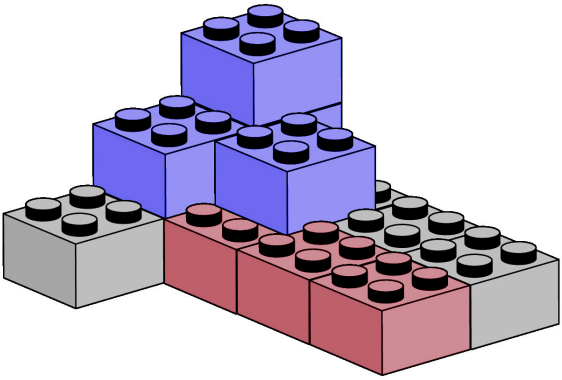
Elixir/OTP

ALQUIMIA CON OTP



MANUEL ÁNGEL RUBIO JIMÉNEZ

Elixir

Alquimia con OTP

Manuel Angel Rubio Jimenez

Este capítulo corresponde al libro
Elixir/OTP: Alquimia con OTP

Puedes adquirir el libro completo aquí:
https://altenwald.com/book/elixir-otp

Elixir
Alquimia con OTP

Manuel Angel Rubio Jimenez

Resumen

OTP es uno de los frameworks más potentes para desarrollar aplicaciones de sistemas para alta carga, necesidad de alta capacidad de procesamiento, tolerancia a fallos y escalabilidad.

Elixir es un lenguaje a medio camino entre Erlang y Ruby. Ha sabido aprovechar las potencias de cada uno y desarrollar un lenguaje sobre BEAM capaz de sacar el máximo partido. En conjunción con OTP provee un sistema de desarrollo versátil, rápido y elimina la complejidad inherente en sistemas de alta concurrencia.

Aprender OTP sobre Elixir te proporciona lo mejor de ambos mundos. Por un lado la potencia de OTP para el desarrollo de soluciones basadas en alta capacidad, concurrencia y tolerancia a fallos. Por otro lado Elixir para ayudar a implementar cada aspecto con un entorno entre funcional y de meta-programación ideal para reducir el código repetitivo (***boilerplate***).

Por lo tanto, en este libro veremos a través de un par de videojuegos (Zero y Leprechaun) las bases para desarrollar empleando aplicaciones, supervisores, servidores, máquinas de estados y gestores de eventos.

Depósito legal CO-2122-2022.

9 788494 552359

[1] http://altenwald.org/curriculum-vitae/manuel
[2] https://creativecommons.org/licenses/by-nc-sa/3.0/deed.es

Capítulo 5. Servidores Genéricos

No íbamos a implementar un lenguaje funcional.
No íbamos a implementar el modelo actor.
Intentábamos solventar el problema.
— Robert Virding

Una de las piezas fundamentales cuando construimos sistemas usando OTP es el servidor. Cuando comentamos los elementos del Modelo Actor situábamos en el centro del sistema a los actores. Tanto los servidores como los actores son los elementos clave de sus paradigmas. En la Programación Orientada a la Concurrencia el servidor se emplea para construir el resto de elementos y de hecho, los elementos vistos hasta el momento son servidores.

Es decir, un supervisor no es sino un servidor en espera por peticiones de un usuario para obtener la información de los procesos bajo su supervisión u obtener información de esos procesos bajo supervisión para reaccionar de una cierta forma. Al mismo tiempo una aplicación mantiene una configuración inicial y mantiene la base para determinar el inicio del árbol de supervisión y obtener todos los procesos bajo una misma aplicación.

En similitud con el Modelo Actor cada servidor creado posee las siguientes características:

- proceso propio para la ejecución del servidor,
- encapsulación fuerte de los datos dentro del estado del proceso,
- facilidad para la recepción de mensajes de tres formas diferentes: llamada síncrona (***call***), envío asíncrono (***cast***) e información sin formato (***info***).

En nuestros ejemplos podremos ver el servidor de juego en ***Leprechaun*** e incluso un bot para jugar a este mismo juego de forma automática. El juego del buscaminas (***Mine***) también está construido usando un servidor genérico. Por último, el proyecto ***Hemdal*** implementa la gestión de máquinas (***hosts***) a través de un servidor genérico.

En definitiva, todo es un servidor y sobre el servidor lo construiremos todo. Veamos qué nos ofrece este elemento.

1. El servidor y el supervisor

Los servidores genéricos están preparados para iniciarse dentro de las estructuras de supervisión provistas por los supervisores, no obstante

nosotros debemos proporcionar en nuestro módulo la función base `start_link/1` necesaria para el inicio.

Pongamos el ejemplo del buscaminas, su servidor de juego en sus dos primeras líneas de código eliminando comentarios nos muestra:

```
defmodule Mine.Game.Worker do
  use GenServer, restart: :transient
```

La segunda línea hace referencia a la implementación del comportamiento, esta línea en verdad agrega bastante contenido a nuestro módulo, vamos a enumerarlo:

- Implementa la función `child_spec/1`. Esta función se agrega para proporcionar la especificación necesaria para el supervisor. A través de opciones como la mostrada (`restart`) podemos sobrecargar las opciones dentro de la especificación o podemos directamente implementar la función por nosotros mismos si lo necesitamos.

- Agrega el comportamiento. Esta línea es como sigue. Le dice al compilador el uso del comportamiento por parte del módulo y nos avisa si no implementamos alguna de las funciones requeridas por el comportamiento.

```
@behaviour GenServer
```

- Implementa la retro-llamada `init/1`. Eso sí, nos arroja un aviso y nos recomienda implementarla nosotros mismos. Veremos en qué consiste esta función en la Sección3, "La función de inicio" .

- Implementa la retro-llamada `handle_call/3` errónea. Si intentas enviar un mensaje síncrono obtendrás un error indicando la falta de esta función y la necesidad de implementarla. Veremos estas llamadas en la Sección6.1, "Llamada síncrona (***call***)".

- Implementa la retro-llamada `handle_cast/2` errónea. Si intentas enviar un mensaje asíncrono obtendrás un error indicando la falta de esta función y la necesidad de implementarla. Veremos estas llamadas en la Sección6.2, "Mensaje asíncrono (***cast***)".

- Implementa la retro-llamada `handle_info/2`. Si un mensaje sin formato llega a nuestro servidor y lo atiende esta implementación veremos un mensaje como anotación (o ***log***) indicando la falta de esta función y el contenido de la información recibida. Veremos esta llamada en la Sección6.3, "Información sin formato" .

- Implementa una retro-llamada `terminate/2` sin acción alguna. Esta retro-llamada es opcional y por tanto puede ignorarse. Veremos

qué podemos hacer al término de los servidores en la Sección 5, "Finalizando un servidor".

- Implementa una retro-llamada `code_change/3` sin acción alguna. Esta retro-llamada es opcional y se emplea para el cambio de código en caliente. Si no hay contenido el cambio se realiza manteniendo el mismo estado interno de la función. Veremos cómo hacer cambios de estado en la Sección10, "Cambio en caliente" .

Comenzaremos por el principio o el inicio.

2. Iniciando el servidor

Cada código debe proporcionar un sentido semántico y por tanto al crear un servidor debemos portar nuestra semántica sobre la funcionalidad a implementar en lugar de dejarnos llevar por la estructura genérica. Es decir, aunque es lógico que un servidor puede iniciarse y puede detenerse, podemos enviarle información de diferentes formas, este no es siempre el caso de los servidores implementados por el comportamiento genérico.

Por ejemplo, si queremos implementar un servidor para mantener una conexión de red como cliente, nuestra semántica sería muy extraña si proporcionamos una función llamada ***start*** en lugar de ***connect*** o una función llamada ***stop*** en lugar de ***disconnect***.

Este es uno de los motivos por los que cada servidor creado se mantiene en un módulo con un nombre, una responsabilidad y unas funcionalidades y emplearemos las funciones proporcionadas por este módulo y no las proporcionadas por el servidor genérico fuera del módulo.

En el ejemplo del juego de Leprechaun tiene sentido mantener las funciones para comenzar el juego como ***start*** e incluso ***stop*** para detenerlo. Mantenemos las funciones y las implementamos desde el servidor genérico de esta forma:

```
@spec start_link(opts()) :: {:ok, pid}
def start_link(opts) do
  name = Keyword.fetch!(opts, :name) ❶
  {:ok, _pid} =
    GenServer.start_link(
      __MODULE__, ❷
      opts, ❸
      name: via(name) ❹
    )
end

defp via(board) do
```

```
  {:via, Registry, {Leprechaun.Game.Registry, board}}
end
```

❶ Primero nos aseguramos de obtener el nombre, si no está entre las opciones la función fallaría en ese momento.

❷ El módulo a emplear para la implementación del servidor genérico será el actual. Esta función se encuentra en el módulo ***Leprechaun.Game***.

❸ Se proporcionan las opciones pasadas como parámetro a la función para ser usadas en el inicio del servidor. Lo veremos más adelante.

❹ Damos al servidor el nombre. La forma de registrar los nombres la veremos en la Sección 3, "Registro de Procesos" del Capítulo 8, ***Registro***.

El servidor se inicia a través de las funciones `GenServer.start_link/2` o `GenServer.start/2`. Como todos los comportamientos están preparados para iniciarse bajo un árbol de supervisión es obligatoria la implementación de la función `start_link/1` dentro del módulo y la funcionalidad de esta función está ligada a la función `GenServer.start_link/2`. Veamos sus parámetros:

```
@spec start_link(module, init_args) :: on_start
@spec start_link(module, init_args, options) :: on_start

@type init_args() :: [any()]
@type options() :: [option()]
@type option() ::
  {:debug, debug()}
  | {:name, name()}
  | {:timeout, timeout()}
  | {:spawn_opt, [Process.spawn_opt()]}
  | {:hibernate_after, timeout()}

@type debug() :: [
  :debug
  | :trace
  | :log
  | :statistics
  | {:log_to_file, Path.t()}
]

@type name() ::
  atom()
  | {:global, term()}
  | {:via, module(), term()}

@type on_start() ::
  {:ok, pid()}
```

```
  | :ignore
  | {:error, {:already_started, pid()} | term()}
```

El inicio se lleva a cabo indicando el módulo donde se implementa el comportamiento pasado como primer parámetro, las opciones de inicio serán enviadas así se indique como único parámetro a la retro-llamada `init/1` y las opciones que podemos indicar son las siguientes.

2.1. `:name`

Hay dos formas de crear servidores con y sin nombre. Cuando proporcionamos un nombre mantenemos al proceso identificado mientras que si lanzamos los procesos sin proporcionar un nombre podemos lanzar tantos como necesitemos. Por ejemplo, si lanzamos un juego de Leprechaun de la siguiente forma dos veces nos encontraremos una colisión con respecto al nombre y en el segundo caso nos remite al proceso empleando ese nombre y no creará un nuevo servidor:

```
iex> Leprechaun.Game.start_link(name: "mi juego")
19:07:09.999 [info] [board] started #PID<0.541.0>
{:ok, #PID<0.541.0>}

iex> Leprechaun.Game.start_link(name: "mi juego")
** (MatchError) no match of right hand side value: {:error,
 {:already_started, #PID<0.541.0>}}
    (leprechaun 1.2.3) lib/leprechaun/game.ex:192:
 Leprechaun.Game.start_link/1
```

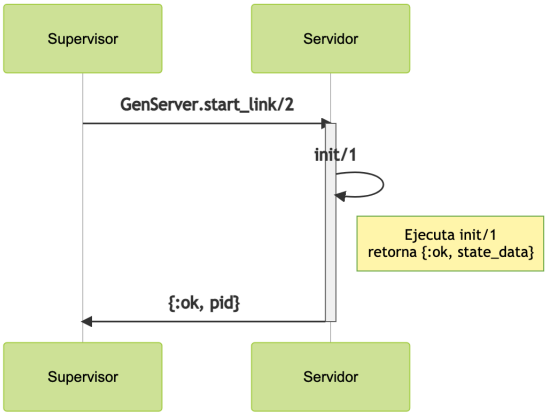
De hecho el código no está preparado para obtener un error por colisión por nombres y falla indicando el retorno proporcionado por la función `GenServer.start_link/3`: `{:error, {:already_started, #PID<0.541.0>}}`.

Curiosidad

Podemos emplear muchas átomos para proporcionar un nombre local a nuestro servidor, una tupla indicando como primer elemento `:global` y como segundo elemento cualquier término y tendremos un nombre registrado de forma global a través de todos los nodos conectados en clúster BEAM y podemos proporcionar una tupla `:via` para indicar un módulo para encargarse del registro del nombre y el identificador de proceso. No importa la forma en que lo hagamos, si necesitamos obtener el identificador de proceso de vuelta podemos emplear la función `GenServer.whereis/1`. Ver más sobre ***:via*** en la Sección 3, "Registro de Procesos" del Capítulo8, ***Registro***.

2.2. `:timeout`

Indica el tiempo total dedicado al inicio del servidor. Este tiempo puede ser necesario en caso de realizar demasiadas acciones en la retro-llamada `init/1`. Esto se explica mejor con un diagrama:



El servidor comienza su ejecución cuando el supervisor realiza la ejecución de la función `GenServer.start_link/3`. Internamente esta función crea un nuevo proceso iniciando su ejecución en la retro-llamada `init/1` y dependiendo de la ejecución y retorno de esta función se determina el retorno de `GenServer.start_link/3`. Es decir, hasta que `init/1` no finaliza su ejecución no avanza la ejecución y por tanto podemos determinar un tiempo máximo a través de `:timeout`. Por defecto el valor es de 5_000 mili-segundos (o 5 segundos).

Si nuestro proceso requiere de un tiempo de inicio prolongado por unas acciones muy pesadas pero no requeridas desde el punto de vista del supervisor. Estas pueden moverse bajo la retro-llamada opcional `handle_continue/2`, veremos esta dinámica en la Sección4, "Continúa el Inicio".

2.3. `:spawn_opt`

Nos permite indicar opciones para la creación del proceso. Estas opciones las podemos encontrar en la documentación para la función `Process.spawn/4` en su cuarto parámetro. Las listaré pero sin entrar en muchos detalles:

`:link`

Enlaza el proceso. Por defecto si empleamos `GenServer.start_link/3` esta opción viene dada.

`{:priority, :low | :normal | :high}`

La prioridad del proceso desde el punto de vista del planificador de tareas. En momentos de carga los procesos con prioridad baja se ejecutarán con menos frecuencia y por tanto se ralentizarán mientras que los procesos con una prioridad más alta se mantendrán más activos y rápidos.

El resto de opciones son referentes a tratar el rendimiento del sistema y uso de la memoria. En caso de requerir modificar o usar estas opciones, te recomiendo echar un vistazo a la documentación oficial y al libro [FH16].

Importante

La opción `:monitor` aparece en el listado pero no funciona. A partir de la versión de OTP 23 surgió una nueva función en Erlang para `start_monitor` que junto con `start_link` ofrecen las dos variantes posibles en contraste con `start`. Sin embargo, estos cambios no han llegado a Elixir en su versión 1.14 por lo que recomiendo seguir empleando `Process.monitor/1` para la monitorización.

Si quieres saber más echa un vistazo a [MR18].

3. La función de inicio

El servidor arranca a través de la retro-llamada `init/1`. El parámetro obtenido es exactamente el segundo parámetro de la función `GenServer.start/2` o `GenServer.start_link/2`. La especificación de la retro-llamada es como sigue:

```
@callback init(init_arg :: term()) ::
  {:ok, state()}
  | {:ok, state(), timeout() | :hibernate}
  | {:ok, state(), {:continue, term()}}
  | :ignore
  | {:stop, reason()}

@type state() :: any()
@type reason() :: any()
```

En esencia pueden darse tres posibles casos cuando realizamos el inicio de un servidor:

Correcto

Retornamos una tupla donde se indican los datos de estado a persistir mientras el servidor esté en ejecución y posiblemente un temporizador, un mandato de pasar a hibernación o continuar la ejecución a través de la retro llamada `handle_continue/2`. Ver en la Sección4, "Continúa el Inicio" .

Ignorar

Hay veces donde un servidor puede permanecer detenido o inactivo. La ejecución no tiene éxito pero se espera el comportamiento y un supervisor no intentará reiniciar el proceso aunque figure como permanente hasta que se requiera de forma explícita.

Por ejemplo, si tenemos nuestro inicio de aplicación por fases, podemos introducir servidores en el árbol de supervisión iniciándose como ***:ignorar*** si no detectan los parámetros correctos o el entorno propicio para iniciarse y después en otra fase configurar el entorno y entonces solicitar al supervisor que reinicie ese servidor.

Detener

La detención de un servidor puede tener dos finales dependiendo de la razón de la detención. Si se produce por un motivo ***:normal*** entonces se reacciona como en el caso de ignorar. Si embargo, si la razón es otra se toma como un error y en caso de recibir este mensaje un supervisor, intentará volver a iniciar el servidor de nuevo.

Por ejemplo, estamos conectando a una base de datos y obtenemos un error. Podemos esperar un tiempo prudencial antes de retornar el error correspondiente y así minimizar el impacto de reinicios en el supervisor probando con cada vez más tiempo de distancia pero sin sobrepasar el umbral del tiempo máximo de espera para iniciar el servidor.

La implementación de la retro-llamada `init/1` para el proceso de inicio del servidor de juego ***Leprechaun.Game*** es así:

```
@doc false
@impl GenServer
def init(opts) do ❶
  if pieces = opts[:pieces], do: Piece.set(pieces)
  board_x = opts[:board_x] || @board_x
  board_y = opts[:board_y] || @board_y
```

```
  turns = opts[:turns] || @init_turns
  board = Board.new(board_x, board_y)

  max_running_time = opts[:max_running_time] ||
    :timer.hours(@max_running_hours)
  Process.send_after(self(), :stop, max_running_time) ❷
  Logger.info("[board] started #{inspect(self())}")

  {:ok, %__MODULE__{board: board, turns: turns}} ❸
end
```

❶ Definimos la función y las opciones recibidas.

❷ Configuramos un temporizador para marcar el final del juego en caso de prolongarse, por seguridad y para no mantener procesos "infinitos".

❸ Retornamos de forma correcta con el estado del juego, en este caso la estructura de datos del propio módulo.

Puedes echar un vistazo al resto de servidores de los otros proyectos. No hay mucha diferencia. Empleamos principalmente esta función para iniciar el estado interno del servidor.

Nota

Debemos tener presente la gestión de la supervisión como vimos en el Capítulo4, ***Supervisores*** en la Sección4, "Tolerancia a fallos" .

Hay veces donde puede ser mejor dar el proceso como iniciado y continuar el proceso de inicio hasta conseguir conectar. Veamos cómo liberar al proceso lanzador o supervisor y seguir con el inicio.

4. Continúa el Inicio

Imaginemos por un momento un servidor para realizar el cambio de divisa. El servidor almacenará un mapa con tuplas donde se indica la divisa de origen, la de destino y su valor será un dato decimal indicando el valor de cambio. El dato tendría la siguiente forma:

```
%{
  {:eur, :usd} => 1.05,
  {:usd, :eur} => 0.95
}
```

Si extraemos esta información de un recurso externo tendremos un riesgo al asumir la actualización de los datos de forma periódica pero

también al inicio del servidor cuando debemos iniciar los datos antes de dar el servidor como disponible.

En este caso definiríamos la función de inicio con un bloque de continuación y dentro de bloque de continuación procederíamos a realizar la carga de los datos remotos. La retro-llamada tiene esta especificación:

```
@callback handle_continue(continue(), state()) ::
  {:noreply, new_state :: any()}
  | {:noreply, new_state :: any(), timeout() | :hibernate}
  | {:noreply, new_state :: any(), {:continue, term()}}
  | {:stop, reason :: any()}
```

Este retorno lo verás repetido a lo largo de todas las funciones de manipulación (***handle***) de los comportamientos donde especificamos si hay o no respuesta (***:reply*** o ***:noreply***) o si detenemos el servidor.

En este caso podemos realizar una segunda recursividad indicando un estado modificado y de nuevo la tupla de continuación, podemos configurar el estado y un temporizador, o pasar el proceso a hibernación o detener el proceso completamente con una razón ***:normal*** (sin errores) o cualquier otra razón resultando en un error.

Ahora ya tenemos nuestro servidor iniciado y con el estado configurado. Incluso hemos obtenido información de cómo detener el servidor de diferentes formas en caso de no iniciar el estado de forma correcta. Veamos las formas de detener un servidor de que disponemos.

5. Finalizando un servidor

Nuestro servidor está en ejecución y una vez ha cumplido su cometido es hora de detener su ejecución. En el caso de los juegos de buscaminas y Leprechaun los servidores equivalen a una partida de un juego. Cuando iniciamos el juego se crea el servidor y podemos jugar y cuando el juego finaliza podemos detener el proceso.

La detención del servidor la llevamos a cabo con al función `GenServer.stop/1`. Esta función tiene la siguiente especificación:

```
@spec stop(server()) :: :ok
@spec stop(server(), reason :: term()) :: :ok
@spec stop(server(), reason :: term(), timeout()) :: :ok

@type server() ::
  pid() ❶
  | atom() ❷
  | {:global, term()} ❸
```

```
  | {:via, module(), term()} ❹
  | {atom(), node()} ❺
```

❶ Un identificador de proceso (PID).

❷ Un nombre local registrado (solo puede ser un átomo).

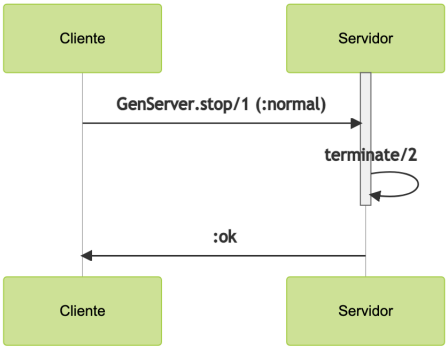
❸ Un nombre global registrado (por el módulo ***:global***).

❹ Una implementación específica para registro flexible y dinámico. En los ejemplos vemos el uso de ***Registry*** para este caso.

❺ Un nombre registrado localmente y el nodo donde se registró.

Es común emplear la función con un solo parámetro. Indicamos dónde se encuentra el servidor y se detiene. Si no indicamos la razón por defecto se enviará `:normal`. El tiempo de espera por defecto es `:infinity`.

Podemos ver en el siguiente diagrama cómo se produce la finalización del proceso:



En el diagrama vemos cómo la función `GenServer.stop/1` actúa de forma síncrona. Hasta la detención del proceso no retorna el valor ***:ok***. De esta forma podemos estar seguros de la finalización del proceso antes de continuar con otra acción.

El proceso de parada ejecuta la retro-llamada `terminate/2` si ha sido implementada. Esta función debe implementarse con la siguiente especificación:

```
@spec terminate(reason(), state :: term()) :: term()

@type reason() ::
  :normal
  | :shutdown
  | {:shutdown, term()}
  | term()
```

Los dos parámetros recibidos son la razón de terminación y los datos de estado del proceso. Las razones más comunes son `:normal` para la detención del proceso sin errores, `:shutdown` y `{:shutdown, term()}` es la razón empleada por el supervisor para indicar un reinicio del proceso.

Esta función puede retornar cualquier valor porque no será tenido en cuenta.

Importante

Si el proceso es detenido por un error de código durante su ejecución o enviando la señal `:kill` la retro-llamada `terminate/2` no será ejecutada. De hecho, cualquier interrupción normal en la ejecución del código dentro del proceso (`raise`, `throw` o `Process.exit/1`) evita la ejecución del código dentro de la retro-llamada `terminate/2`.

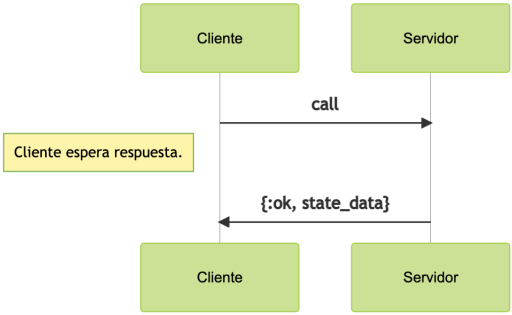
6. Comunicación con el Servidor

Sabemos cómo iniciar nuestro servidor, como iniciar su estado de datos interno y cómo finalizar su ejecución. Ahora vamos a comunicarnos con el servidor.

Disponemos de tres formas para enviar información a nuestro servidor: llamada síncrona, mensaje asíncrono o información sin formato. Cada uno de estos métodos tiene sus beneficios y perjuicios. Veremos cada uno de ellos, cómo emplearlos y algunos trucos.

6.1. Llamada síncrona (*call*)

En BEAM toda comunicación es asíncrona y realizar una comunicación síncrona nos obliga a implementar esta sincronía a nosotros mismos. El comportamiento del servidor genérico tiene una implementación con muchos detalles y beneficios. Pero antes de proseguir, vamos a ver con un diagrama cómo se produce esta comunicación:



De forma muy burda podemos ilustrar el código encargado de la llamada (***call***) del cliente como:

```
send(servidor, mensaje)
respuesta =
  receive do
    {:ok, resp} -> resp
  end
```

Es decir, dejamos al cliente bloqueado por la recepción de un mensaje por parte del servidor. Pero como puedes imaginar, esto no es solo así. Hay involucrados muchos más elementos porque, ¿qué sucedería si el servidor no responde? ¿si el servidor termina su ejecución durante el tiempo de espera o incluso antes? ¿y si hicimos una petición anterior y en esta petición recibimos la respuesta anterior? ¿cómo sabe el servidor dónde debe enviar la respuesta?

Como ves, son muchos detalles de los que se encarga el servidor. Nosotros podemos ocuparnos únicamente de emplear la función correspondiente al envío del mensaje síncrono al servidor:

```
@spec call(server(), request :: any()) :: term()
@spec call(server(), request :: any(), timeout()) :: term()

@type server() ::
  pid()
  | atom()
  | {:global, term()}
  | {:via, module(), term()}
  | {atom(), node()}
```

En esta llamada localizamos al servidor de igual forma que cuando le solicitamos detenerse en el apartado anterior para hacerle llegar el mensaje. El mensaje no tiene un formato específico. Podemos hacerle llegar al servidor cualquier tipo de dato e incluso una función anónima,

un identificador de proceso (acrony::[PID]) o una referencia. De igual forma el servidor puede enviar de vuelta cualquier tipo de dato.

Podemos ver ejemplos de llamadas a servidor dentro del módulo de buscaminas. Por ejemplo:

```
def show(game_id), do:
  GenServer.call(via(game_id), :show)
```

La función `Mine.Game.show/1` realiza el envío del mensaje `:show` y espera por la información desde el servidor. Solo nos queda ver el código atendiendo esta petición en el lado del servidor. La especificación es la siguiente:

```
@spec handle_call(msg(), from(), state()) ::
  {:reply, reply(), state()}
  | {:reply, reply(), state(), timeout()}
  | {:reply, reply(), state(), :hibernate}
  | {:reply, reply(), state(), {:continue, term()}}
  | {:noreply, state()}
  | {:noreply, state(), timeout()}
  | {:noreply, state(), :hibernate}
  | {:noreply, state(), {:continue, term()}}
  | {:stop, reason(), reply(), state()}
  | {:stop, reason(), state()}

@type msg() :: term()
@type from() :: {pid(), tag :: term()}
@type state() :: term()
@type reply() :: term()
@type reason() :: term()
```

Tenemos varias opciones dentro de nuestro código cuando recibimos una llamada síncrona. Podemos responder cambiando en ese momento el estado interno para obtener ese cambio en la siguiente llamada, podemos configurar un tiempo de espera o podemos mandar el proceso a hibernar. También existe la posibilidad de no responder nada e incluso detener el proceso del servidor respondiendo o no al cliente.

Veamos las opciones. Primero la más frecuente, enviamos una respuesta sin más:

```
def handle_call(:show, _from, state) do
  cells = Board.get_naive_cells(state.board)
  {:reply, cells, state}
end
```

Recibimos el mensaje, obtenemos del tablero almacenado en el estado del proceso los datos, las celdas del juego y las retornamos en el retorno de la retro-llamada. Así le indicamos al comportamiento la información a retornar, el nuevo estado y la intención de continuar con la ejecución.

En verdad el estado insertado en el retorno como tercer elemento de la tupla es el mismo y con esto indicamos que no queremos cambios en el estado.

Hablaremos de las opciones de tiempo de espera agotado e hibernación más adelante.

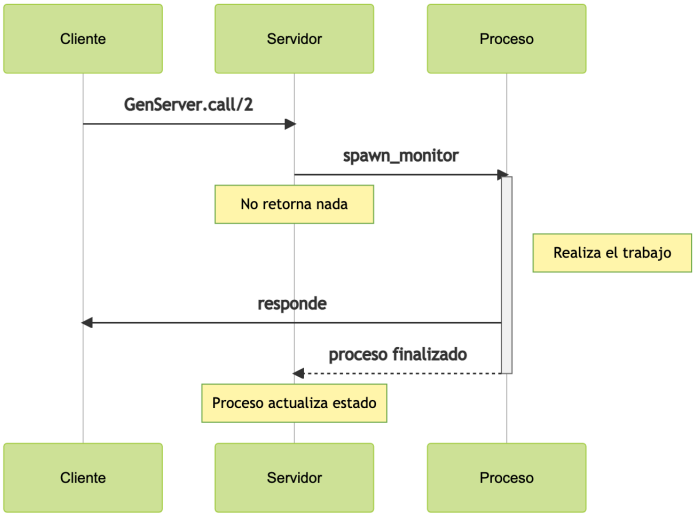
¿Qué sucedería si no enviamos una respuesta? En principio si no enviamos una respuesta e ignoramos completamente la petición el tiempo de espera para la función `GenServer.call/2` en cliente agota su tiempo de espera (5 segundos) y produce un error. ¿Qué sentido tiene entonces no responder?

En el proyecto Hemdal dentro del módulo ***Hemdal.Host*** recibimos peticiones síncronas pero en lugar de atenderlas una a una liberamos el proceso servidor para atender más peticiones y agregamos en una cola la información para realizar el retorno manteniendo siempre al servidor libre para atender peticiones entrantes de forma rápida. Esto lo conseguimos así:

```
def handle_call({:exec, cmd, args}, from,
     %__MODULE__{max_workers: :infinity} = state) do
  spawn_monitor(fn ->
    run_in_background(cmd, args, from, state)
  end)
  workers = state.workers + 1
  {:noreply, %__MODULE__{state | workers: workers}}
end
```

La primera clausula de la función ejecuta siempre el nuevo worker porque el número `max_workers` ha sido configurado como `:infinity`. En este caso lanza el proceso con la tarea a realizar y ahí puedes observar el paso de `from` a la función. Actualizamos el estado para reflejar el incremento de trabajadores activos y cuando recibamos el mensaje de fin de proceso entonces procederemos a decrementar, pero ese mensaje lo obtendremos como información sin formato.

En principio solo tienes que ver la siguiente dinámica:



Como dijimos inicialmente, todas las comunicaciones en BEAM son asíncronas. La sensación de sincronía la genera el comportamiento del servidor y la emplea para poder responder al proceso llamante de forma simple a través del retorno de la retro-llamada con la tupla de tipo `:reply` pero también nos permite emplear la función `GenServer.reply/2` y realizar la respuesta en otro momento y desde otro proceso diferente.

6.2. Mensaje asíncrono (*cast*)

El mensaje asíncrono es mucho más simple. Se basa en enviar el mensaje y proseguir. La función `GenServer.cast/2` se encarga de enviar el mensaje pero no espera por una respuesta y tampoco valida si el proceso existe o no. Solo falla si empleamos algún nombre y el nombre no conduce a ningún identificador de proceso.

La especificación de la función `GenServer.cast/2` es como sigue:

```
@spec cast(server(), request :: any()) :: :ok
@spec cast(server(), request :: any(), timeout()) :: :ok

@type server() ::
  pid()
  | atom()
  | {:global, term()}
  | {:via, module(), term()}
  | {atom(), node()}
```

Siempre recibimos la misma respuesta. No hay mucho más que hacer. Podemos ver algunos ejemplos en el juego del buscaminas:

```
def sweep(game_id, x, y),
  do: GenServer.cast(via(game_id), {:sweep, x, y})

def flag(game_id, x, y),
  do: GenServer.cast(via(game_id), {:flag, x, y})
```

Estas opciones para descubrir una posición o poner una bandera no requieren de un retorno por parte del servidor y por tanto las implementamos de esta forma.

Veamos cómo manejamos el mensaje recibido en el proceso del servidor mediante la retro-llamada `handle_cast/2`. La especificación de esta función es la siguiente:

```
@spec handle_cast(msg(), state()) ::
  {:noreply, state()}
  | {:noreply, state(), timeout()}
  | {:noreply, state(), :hibernate}
  | {:noreply, state(), {:continue, term()}}
  | {:stop, reason(), state()}

@type msg() :: term()
@type state() :: term()
@type reason() :: term()
```

No tenemos muchas opciones. En este caso solo podemos no responder y modificar el estado interno del proceso o detener el proceso.

La implementación del código de buscaminas para atender el mensaje `{:sweep, x, y}` es la siguiente:

```
def handle_cast({:sweep, _, _},
      %__MODULE__{status: :gameover} = state) do
  {:noreply, state}
end

def handle_cast({:sweep, _, _},
      %__MODULE__{status: :pause} = state) do
  {:noreply, state}
end

def handle_cast({:sweep, _, _} = msg,
      %__MODULE__{timer: nil} = state) do
  {:ok, timer} = :timer.send_interval(1_000, self(), :tick)
  handle_cast(msg, %__MODULE__{state | timer: timer})
end

def handle_cast({:sweep, x, y}, state) do
  state.board
  |> Board.get_cell(x, y)
  |> process_sweep(x, y, state)
```

```
end
```

En este caso vemos cuatro bloques diferentes. Empleamos el contenido del estado para elegir la funcionalidad a realizar según el mensaje recibido. Si el juego está en estado `:gameover` o en `:pause` no hacemos nada. En caso de no haber iniciado el temporizador lo lanzamos en ese momento y el caso general se encarga de realizar la acción sobre el tablero y procesar la acción para actualizar apropiadamente el estado interno del proceso con respecto al tablero y la puntuación.

6.3. Información sin formato

Un servidor no deja de ser una ejecución recursiva donde vamos modificando el estado y siempre esperamos por un mensaje entrante en el proceso. Cuando hablamos de lanzar un proceso monitorizado y la recepción del mensaje cuando el proceso terminase su ejecución dijimos que ese mensaje llegaría al servidor como información sin formato. Es aquí donde manejamos esa información.

Cuando realizamos un envío de información sin emplear las funciones `GenServer.call/2` o `GenServer.cast/2`, es decir, lo hacemos mediante `send/2` y es cuando el mensaje no tiene ningún formato. Este tipo de mensajes son también generados por la función de monitorización cuando nos hace saber de la terminación de un proceso, de las funciones de tiempo (del módulo `:timer`) o incluso de otros módulos como `:gen_tcp` o `:gen_udp`.

La implementación en Hemdal para cuando recibimos el fin de un proceso lanzado es la siguiente:

```
def handle_info(
      {:DOWN, _ref, :process, _pid, _reason},
      %__MODULE__{queue: []} = state
    ) do
  workers = state.workers - 1
  {:noreply, %__MODULE__{state | workers: workers}}
end

def handle_info({:DOWN, _ref, :process, _pid, _}, state) do
  [{from, cmd, args} | queue] = state.queue
  state = %__MODULE__{state | queue: queue}
  spawn_monitor(fn ->
    run_in_background(cmd, args, from, state)
  end)
  {:noreply, state}
end
```

Cuando recibimos el mensaje del sistema de monitorización sobre la finalización de un proceso procedemos a revisar si hay alguna otra tarea esperando a ser lanzada. Si no la hay entonces incrementamos el número de trabajadores máximo y no hacemos nada más.

La especificación para el manejador de la información sin formato es la siguiente:

```
@spec handle_info(info(), state()) ::
  {:noreply, state()}
  | {:noreply, state(), timeout()}
  | {:noreply, state(), :hibernate}
  | {:noreply, state(), {:continue, term()}}
  | {:stop, reason(), state()}

@type info() :: term()
@type state() :: term()
@type reason() :: term()
```

Al igual que la retro-llamada `handle_cast/2` podemos mantener la ejecución y agregar un temporizador, hibernar el proceso o disparar la ejecución de la retro-llamada::[handle_continue/2] o podemos detener el proceso.

7. Temporizadores

En cada uno de los manejadores nos dan las opciones de agregar un temporizador o incluso hibernar el proceso. Los temporizadores vienen a indicar un tiempo de espera de este estilo:

```
receive do
  {:"$cast", msg} -> handle_cast(msg, state)
  {:"$call", msg} -> handle_call(msg, from, state)
  info -> handle_info(info, state)
after timeout ->
  handle_info(:timeout, state)
end
```

Esta es solo una aproximación y obviamente el código real maneja muchas más excepciones, manejo de retorno y detención del sistema, pero sirve para hacernos una idea.

En esencia, el temporizador es válido mientras no llegue ningún otro mensaje al proceso porque en ese caso lanzaría la ejecución de algún manejador y vuelta a empezar.

Por ese motivo el juego del buscaminas tiene un temporizador externo al proceso del servidor para recibir un evento cada segundo y nos permita decrementar el tiempo hasta llegar a cero.

8. Hibernación

La hibernación permite eliminar un proceso del planificador de tareas y comprime su memoria para ocupar menos espacio. Si disponemos de un proceso que será empleado con poca frecuencia y manejará

pocos mensajes o con mucho espacio de tiempo entre una petición y la siguiente puede ser buena idea hibernar el proceso.

Si proporcionamos la opción `:hibernate_after` en la función `GenServer.start_link/3` o `GenServer.start/3` el servidor se mantendrá a la espera durante el tiempo especificado y si ningún mensaje es recibido entonces pasa a modo hibernado:

```
GenServer.start_link(__MODULE__, [], hibernate_after: 500)
```

En este ejemplo, si el proceso no recibe ninguna petición en 500 milisegundos entrará en modo hibernación. A nivel de programación no hay ningún cambio. Un proceso hibernado reacciona igual que un proceso normal porque es despertado de su hibernación en el momento de recibir un nuevo mensaje.

El comportamiento de ***GenServer*** nos proporciona las mecánicas para simplificar esta acción a través de un simple retorno en los manejadores o una configuración al inicio del servidor tal y como hemos visto.

9. Observabilidad

Una de las grandes potencias de BEAM es la capacidad para proporcionar información de cada proceso en ejecución. Un comportamiento no solo proporciona las mecánicas necesarias para implementar un servidor sino también funciones para permitirnos trazar u observar qué sucede dentro de cada uno de los servidores.

Esta información nos puede servir de ayuda para implementar pruebas de nuestro código y observar procesos en producción de forma poco intrusiva.

Vamos a tomar de ejemplo el servidor de juego del buscaminas ***Mine.Game.Worker***. Queremos obtener información sobre qué acciones realiza el proceso. Emplearemos la función `:sys.trace/2` de la siguiente forma:

```
iex> {:ok, pid} = Mine.Game.start "juego"
#PID<0.641.0>

iex> :sys.trace pid, true
:ok

iex> Mine.Game.sweep "juego", 1, 1
*DBG* {'Elixir.Mine.Game.Registry','Elixir.Mine'} got cast
{sweep,1,1}
:ok
*DBG* {'Elixir.Mine.Game.Registry','Elixir.Mine'} new state
#{'__struct__' => 'Elixir.Mine.Game.Worker', board => ...}
*DBG* {'Elixir.Mine.Game.Registry','Elixir.Mine'} got tick
*DBG* {'Elixir.Mine.Game.Registry','Elixir.Mine'} new state
```

```
#{'__struct__' => 'Elixir.Mine.Game.Worker', board => ...}
```

Al activar las trazas para el proceso podemos ver por pantalla cada mensaje obtenido dentro del servidor y cómo afecta al estado interno. Uno de los problemas de obtener esta información por pantalla es la cantidad de información almacenada en el estado del servidor. Con cada `:tick` (producido cada segundo) vemos la representación completa del estado y hace difícil su seguimiento.

Si necesitas ver el estado puedes simplemente emplear la función `:sys.get_state/1` o `:sys.get_status/1`. Ambas funciones son diferentes porque la primera (***state***) retorna el estado tal como lo ves dentro de los manejadores. La segunda (***status***) muestra más información y ejecuta la retro-llamada `format_status/2` para modificar la forma de representar el estado antes de presentarlo.

Esta retro-llamada nos puede ayudar para proporcionar una información más clara sobre el estado interno del juego, por ejemplo, modificando el tablero para mostrar una versión más pequeña:

```
def format_status(_reason, [_pdict, state]) do
  %__MODULE__{state |
    board: Board.get_naive_cells(board)
  }
end
```

De esta forma, en lugar de ver la representación en forma de mapa, veremos una lista de listas sin número de índice y por tanto con una representación más compacta.

Para sistemas como `:observer` donde podemos obtener el estado interno de un proceso a través de esta función nos facilita modificar y adaptar la información para obtener una mejor legibilidad es un punto a favor. Para dar algunas ideas, estas modificaciones pueden incluir el cambio de una marca de tiempo (***timestamp***) por una fecha en un formato más claro o la conversión átomo o número identificando un estado a un texto más representativo.

10. Cambio en caliente

El comportamiento del servidor nos posibilita realizar cambios en caliente y aunque veremos cómo realizar estos cambios más adelante sí podemos ver qué hacer para modificar el estado interno de nuestro servidor en caso de modificar la estructura de su estado.

Por ejemplo, si tenemos el juego de buscaminas con este estado inicial:

```
defstruct board: nil,
```

```
          flags: 0,
          score: 0,
          status: :play,
          timer: nil,
          time: nil,
          consumers: [],
          username: nil
```

Las estructuras son mapas internamente y realmente la única comprobación sobre si una estructura es correcta la determina el valor de la clave `_\_struct\_\_`. No obstante, siempre es buena realizar los cambios precisos para no encontrar errores imprevistos por la falta de estos datos.

Por ejemplo, si la nueva versión elimina el campo `username` y lo cambia por `user_id` porque agregamos finalmente un sistema para dar de alta usuarios y proporcionar sesiones el cambio de un juego en curso debe ejecutar la retro-llamada `code_change/3` y debe tener esta forma:

```
def code_change(_old_vsn, state, _extra) do
  user_id = Users.search_by_username(state.username)

  state
  |> Map.delete(:username)
  |> Map.put(:user_id, user_id)
end
```

De esta forma eliminamos el campo antiguo y rellenamos el nuevo con el nuevo dato permitiendo al usuario continuar con su partida sin interrupción.

La especificación de esta retro-llamada es la siguiente:

```
@spec code_change(old_vsn(), state(), extra()) ::
  {:ok, state()}
  | {:error, reason()}

@type old_vsn() :: term()
@type state() :: term()
@type extra() :: term()
@type reason() :: term()
```

La versión del módulo puede indicarse explícitamente a través del atributo de módulo `@vsn` aunque normalmente no se emplea y el compilador termina generando esta versión como un valor por defecto. Si ejecutamos la función `module_info/0` para nuestro módulo podremos ver dentro de la sección ***attributes*** el atributo ***vsn*** y el valor auto-generado para nuestro módulo.

En caso de un cambio en caliente, a la función `code_change/3` le llegará como primer parámetro este atributo correspondiente al módulo antiguo y en caso de conocer esta numeración podemos realizar algún

tipo de concordancia para actualizar correspondientemente el estado del servidor.

Esto se ve mejor con un ejemplo en consola. Vamos por pasos. Primero definimos el módulo y lanzamos el proceso. Podemos comprobar el estado dentro del proceso también:

```
iex> defmodule Prueba do
...>   @vsn 1
...>   use GenServer
...>   defstruct [:name, :address]
...>   def init([]), do: {:ok, %__MODULE__{}}
...> end
{:module, Prueba, <<...>>, ...}

iex> {:ok, pid} = GenServer.start_link(Prueba, [])
{:ok, #PID<0.173.0>}

iex> :sys.get_state(pid)
%Prueba{name: nil, address: nil}
```

Suspendemos la ejecución del proceso:

```
iex> :sys.suspend(pid)
:ok
```

Una vez tenemos suspendido el proceso realizamos el cambio del código. Sobrecargamos el código de nuestro módulo con una nueva versión:

```
iex> defmodule Prueba do
...>   @vsn 2
...>   use GenServer
...>   defstruct [:name, :age]
...>   def init([]), do: {:ok, %__MODULE__{}}
...>   def code_change(1, state, []) do
...>     {:ok, Map.delete(state, :address) |> Map.put(:age,
 nil)}
...>   end
...> end
{:module, Prueba, <<...>>, ...}
```

Ahora indicamos el cambio de código. Al ejecutar la función `:sys.change_code/4` esta función se encargará de realizar los cambios pertinentes internamente y de ejecutar la retro-llamada `code_change/3` si fue definida para nuestro módulo. Además, definimos la versión a eliminar y parámetros extra en caso de ser necesarios:

```
iex> :sys.change_code(pid, Prueba, 1, [])
:ok
```

Por último, reanudamos la ejecución del proceso y comprobamos el estado del proceso:

```
iex> :sys.resume(pid)
:ok

iex> :sys.get_state(pid)
%Prueba{name: nil, age: nil}
```

Este cambio es manual y veremos más adelante cómo realizar el cambio de forma automática para el conjunto de todos los comportamientos y todos los procesos dentro de nuestro árbol de supervisión.

11. Todo es un servidor

Comentamos al inicio de este capítulo que una de las piezas fundamentales al construir los sistemas en OTP es el servidor. También dijimos en el capítulo sobre supervisores que el supervisor está construido sobre un servidor y vimos el funcionamiento básico de los manejadores de mensajes y los tipos de mensajes.

A lo largo de los siguientes capítulos veremos otros comportamientos y haremos comparaciones de nuevo con el servidor. Nos referiremos a las dinámicas aquí explicadas para los mensajes, los temporizadores, la hibernación porque en esencia para los otros comportamientos el funcionamiento es el mismo salvo que se indique lo contrario de forma explícita.