Manejo de Errores e Introducción a la Supervisión

Borrador

Franco Bulgarelli

En objetos uno normalmente tiene tres patrones de comunicación para señalizar errores:

- call & return: enviar un mensaje y que este devuelva algo que represente al error, ya sea un código o un objeto más rico
- continuaciones
- excepciones: el método que falla lanza la excepción, abortando el envío del mensaje, y propagandola hasta quien lo inició (que en última instancia, es el main, si ninguna porción de código en el medio captura la excepción)

Con algunas excepciones dadas en general por restricciones de la plataforma (ios, node) el mecanismos de excepciones es el más usado

Hasta acá no hay nada nuevo.

Lo interesante es que en actores tenemos también los tres mecanismos, y los tres tienen cierta aplicabilidad. En erlang en particular tenemos que:

- call and return es usado frecuentemente para señalizar ausencias de valores o errores que deben ser manejados de forma cercana a quien lo propagó, usando algunos de los siguientes formatos:
    - {ok, Value} / error
    - {value, Value} / error
    - Value / error

  Es bastante evidente que todos estos tipos algebraicos describen functores. Sin embargo, más por convención de la comunidad que por limitaciones propias del lenguaje, se estila manejar esto con pattern matching, en lugar de con combinadores como el fmap. Y además de ello, el lenguaje no posee una sintaxis especial como la do-syntax de Haskell o las for-comprehensions de Scala, lo que aún con combinadores su uso no es transparente. Por ello entonces es que decimos que son usados para manejar errores que deben ser tratados en un lugar cercano a su causa, dado que propogar estos errores es engorroso.
- Continuaciones: teóricamente no hay limitaciones sobre su uso, pero el API estándar no presenta muchos ejemplos de estas para el manejo de errores.
- Excepciones: son la base del fail-fast y let-it-crash de Erlang. En Erlang y derivados se diferencian al menos dos tipos de excepciones
    - las excepciones utilizadas para indicar errores (error) y terminaciones de procesos (exit). Lo que indican es que hubo un error y el proceso debe ser detenido.
    - los retornos no locales. La diferencia semántica es sutil, y se ve en lo que ocurre al no capturar un retorno no local (throw): el error que se produce por no capturar una excepción de tipo error es justamente ese error. Pero el error de no capturar un throw es.... {nocatch,Value}. Es decir, throw no está indicando que el proceso deba ser detenido, sino que alguien debe manejar ese throw; y el error está en no haberlo manejado (en contraposición con error, donde el error es el lanzamiento propiamente dicho).
      Mientras que error tiene una semántica de "lanzo esto sin esperar que alguien lo

capture", throw tiene una semántica de "lanzo esto porque SE que alguien lo va a manejar".

Pero el diablo está en los detalles.

Más allá de la discusión throw/error, que es un poco anecdótica (podríamos haber vivido sin conocer a throw, y sin usar retornos no locales), lo que vale la pena analizar son las diferencias entre la excepciones de actores y de objetos, o mejor dicho, analizar que son idénticas pero que el modelo de procesos independientes modifica algunas consecuencias.

Volvamos a la idea original: las excepciones cortan el flujo de ejecución del envio del mensaje, hasta llegar a quien lo inició (el main), es decir, se propagan a través del stack.

En Erlang secuencial esto se va a comportar de la misma manera claro. Pero cambia cuando introducimos los procesos:

- No hay un main, hay N "mains", uno por cada actor.
- El envio de mensajes de un actor a otro actor no ocurre a través del stack, y es asincrónico (y aun los mensajes sincrónicos se conforman a través de mensajes asincrónicos).
- El actor es un proceso ejecutable, en contraposición con objetos, donde el método es el código ejecutable.
- A diferencia del código de un método de un objeto que es ejecutado por el proceso (hilo, fibra, etc) en el que el envio de mensajes ocurre, el código de un recieve es ejecutado por el proceso asociado al actor receptor.

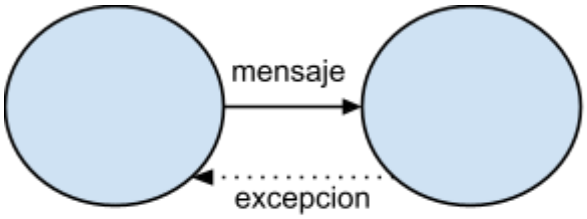
Consecuencias:

- una excepción dentro del actor no "destruye" a la cadena de envio de mensajes, sino que destruye al actor mismo.
- en el paso de mensajes entre actores, las excepciones que ocurran como parte de un envío de mensajes NO serán reportadas al actor que envió el mensajes (en contraposición a "en el paso de mensajes entre objetos, las excepciones que ocurran como parte de un envío de mensajes serán reportadas al (método del) objeto que envió el mensaje")
- Qué significa esto, que el manejador de la excepción ya no es el cliente que inició el pedido (como en objetos). Lema: si yo envio un mensaje y ese mensaje provoca la muerte del actor, yo no soy responsable de manejar esa situación.

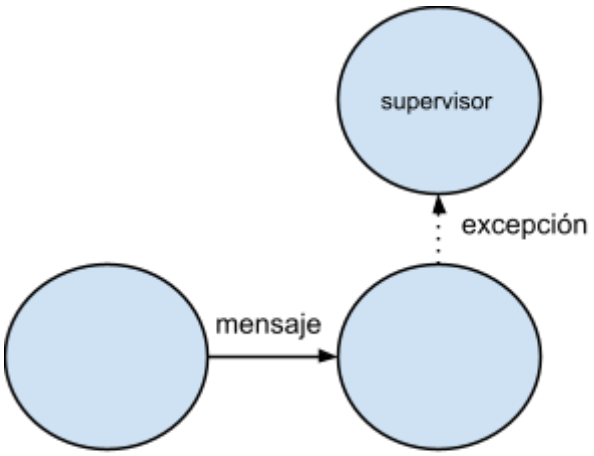
Entonces, ¿quien la maneja?. Simple:

- Por defecto, nadie. El actor muere. Y esa funcionalidad del sistema queda deshabilitada (ups, eso es malo, pero al menos no muere el programa completo, como lo sería en un ambiente de objetos secuencial)
- Pero podemos designar a otro actor para que se encargue de la misma, y tome una acción. Por una cuestión de separación de responsabilidades, este actor responsable de manejar la falla de otro actor se dedicará sólo a eso: será su supervisor. Y la acciones que puede tomar son:
    - reiniciarlo
    - dejarlo morir
    - morir también junto con el proceso supervisado (y si alguien lo supervisa, podrá tomar las mismas decisiones)

Es decir, en actores habrá dos estructuras paralelas: la estructura que sirve para resolver la funcionalidad en sí, y la estructura de manejo de error. Por eso, es que en actores el manejo de error es una idea tan importante.



Excepciones en Objetos



Excepciones en Actores

Links:

- http://elixir-lang.org/getting-started/mix-otp/supervisor-and-application.html