Elixir

INTRODUCCIÓN PARA ALQUIMISTAS

MANUEL ÁNGEL RUBIO JIMÉNEZ

Elixir

Introducción para Alquimistas

MANUEL ÁNGEL RUBIO JIMÉNEZ

Elixir nació como la solución a un gran problema que recaía sobre BEAM de ser una gran plataforma pero con un lenguaje *raro*. Con influencia de otros lenguajes y una gran ejecución por parte de José Valim comenzaron a salir las primeras versiones del lenguaje en 2011 y muchos entusiastas comenzaron a dedicarse a mejorar y hacerlo crecer.

Este libro te ayuda a adentrarte en el mundo de Elixir. Cómo nació, cómo es su comunidad, su ecosistema y por supuesto el lenguaje. Las ventajas y lo que lo hacen único e incluso sus debilidades. Todo para ayudarte a dar tus primeros pasos en Elixir o aprender toda la base en caso de tener ya conocimientos previos.

Aprenderás cómo crear aplicaciones cliente-servidor y las mejores prácticas para desarrollar con Elixir, crear proyectos, integrar tu código con el de otros a través de la instalación de dependencias, publicar tus propias librerías y desplegar tus proyectos en producción.

Este libro cubre todos los aspectos principales de Elixir 1.7.

ISBN 978-84-945523-4-2

Elixir

Introducción para Alquimistas

Manuel Angel Rubio Jiménez

Elixir
Introducción para Alquimistas

Manuel Angel Rubio Jiménez

Resumen

Elixir nació como la solución a un gran problema que recaía sobre BEAM de ser una gran plataforma pero con un lenguaje ***raro***. Con influencia de otros lenguajes y una gran ejecución por parte de José Valim comenzaron a salir las primeras versiones del lenguaje en 2011 y muchos entusiastas comenzaron a dedicarse a mejorar y hacerlo crecer.

Este libro te ayuda a adentrarte en el mundo de Elixir. Cómo nació, cómo es su comunidad, su ecosistema y por supuesto el lenguaje. Las ventajas y lo que lo hacen único e incluso sus debilidades. Todo para ayudarte a dar tus primeros pasos en Elixir o aprender toda la base en caso de tener ya conocimientos previos.

Aprenderás cómo crear aplicaciones cliente-servidor y las mejores prácticas para desarrollar con Elixir, crear proyectos, integrar tu código con el de otros a través de la instalación de dependencias, publicar tus propias librerías y desplegar tus proyectos en producción.

Este libro cubre todos los aspectos principales de Elixir 1.7.

Depósito legal CO-2236-2018.

ISBN 978-84-945523-4-2

[1] http://books.altenwald.com/elixir-i
[2] https://creativecommons.org/licenses/by-nc-sa/3.0/deed.es

Tabla de contenidos

Introducción viii
1. Acerca del autor viii
2. Acerca del libro ix
3. Objetivo del libro ix
4. ¿A quién va dirigido este libro? x
5. Nomenclatura usada x
6. Agradecimientos xi
7. Más información en la web xii
1. Lo que debes saber sobre Elixir 1
1. ¿Qué es Elixir? 1
2. Características de Elixir 2
2.1. Características de BEAM 2
2.2. Características de Elixir 4
3. Historia de BEAM 5
4. Comenzando la idea de Elixir 7
5. Desarrollos con Elixir 9
6. Soportando 2 Millones de Usuarios Conectados 9
2. El lenguaje 12
1. El intérprete de Elixir (**iex**) 13
2. Azúcar sintáctico 14
3. Comentarios 15
4. Tipos de Datos 15
4.1. Átomos 15
4.2. Booleanos o Lógicos 17
4.3. Valor nulo ***nil*** 18
4.4. Números Enteros y Reales 18
4.5. Variables e Inmutabilidad 20
4.6. Listas 22
4.7. Colecciones 25
4.8. Cadenas de texto 26
4.9. Trabajando con Binarios 28
4.10. Trabajando con Bits 28
4.11. Tuplas 31
4.12. Mapas 33
4.13. Cambios en Profundidad 35
4.14. Conjuntos 36
4.15. Rangos 36
4.16. Sigilos 37
5. Conversión de datos 39
6. Imprimiendo por pantalla 40
3. Expresiones, Estructuras y Errores 42
1. Expresiones 42
1.1. Expresiones Aritméticas 42

1.2. Expresiones Lógicas 43
1.3. Expresiones regulares 44
1.4. Precedencia de Operadores 45
1.5. Sobrecarga de operadores 47
2. Estructuras de Control 48
2.1. Concordancia 48
2.2. Operador ***pipe*** 50
2.3. Estructura ***if...else*** y ***unless..else*** 51
2.4. Estructura ***case*** 52
2.5. Estructura ***cond*** 54
2.6. Estructura ***with*** 54
2.7. Estructura ***for*** o Listas de Comprensión 55
3. Errores 56
4. Estructura ***try..catch*** 58
5. Salida 58
4. Las funciones y los módulos 60
1. Organización del código 60
2. Definición de Módulos 61
3. Atributos del Módulo 63
4. Aliases e Importación 66
5. Funciones 68
6. Polimorfismo y Concordancia 69
7. Argumentos Opcionales 70
8. Guardas 71
9. Registros 72
10. Estructuras 73
11. Protocolos 75
12. Sigilos 77
13. Clausuras 78
14. Programación Funcional 80
14.1. Transparencia referencial o Inmutabilidad 80
14.2. Funciones de Primera Clase y Orden Superior 80
14.3. Sin Efectos Colaterales 80
14.4. Expresiones frente a Mandatos 81
14.5. Evaluación perezosa 81
15. Recursividad 82
15.1. Ordenación por mezcla (***mergesort***) 83
15.2. Ordenación rápida (***quicksort***) 84
16. Comportamientos 85
17. Interoperatividad y BIFs 87
5. Meta-Programación 88
1. ¿Cómo compila Elixir? 89
2. Retro-llamadas del Compilador 92
3. Macros 95
4. Higiene en las Macros 99
5. Extendiendo Módulos 100

6. Tiempo Real 104
1. Tipos de datos de Fechas y Horas 104
2. Cálculo de Fechas y Horas 105
3. Tiempo monótono 106
4. Identificadores únicos 107
5. Túneles de tiempo 108
6. ¿Cómo mantener el código seguro? 109
7. Procesos y Nodos 110
1. Anatomía de un Proceso 110
2. Ventajas e inconvenientes 111
3. Lanzando Procesos 112
4. Bautizando Procesos 114
5. Comunicación entre Procesos 115
6. Procesos Enlazados 118
7. Monitorización de Procesos 122
8. Recarga de código 125
9. Gestión de Procesos 128
10. Nodos 129
11. Nodos ocultos 131
12. Procesos Remotos 131
13. Diccionario del Proceso 133
8. Ficheros y Directorios 134
1. Ficheros 134
1.1. Abriendo y Cerrando Ficheros 135
1.2. Fichero manejado o en ***crudo*** 137
1.3. Lectura de Ficheros de Texto 137
1.4. Escritura de Ficheros de Texto 138
1.5. Lectura de Ficheros Binarios 139
1.6. Escritura de Ficheros Binarios 140
1.7. Acceso aleatorio de Ficheros 140
2. Flujos de datos 141
3. Gestión de Ficheros 142
3.1. Nombre del fichero 143
3.2. Copiar, Mover y Eliminar Ficheros 143
3.3. Permisos, Propietarios y Grupos 144
4. Gestión de Directorios 146
4.1. Directorio de Trabajo 146
4.2. Creación y Eliminación de Directorios 147
4.3. ¿Es un fichero? 147
4.4. Contenido de los Directorios 148
9. Comunicaciones y Servidores 150
1. Conceptos básicos de Redes 150
1.1. Direcciones IP 151
1.2. Puertos 153
2. Servidor y Cliente UDP 154
3. Servidor y Cliente TCP 158
4. Servidor TCP Concurrente 161

5. Ventajas de ***:inet*** 165
10. Un vistazo a OTP 169
1. Ejecución Asíncrona con Task 170
2. Servidores 173
3. Agentes 176
3.1. ¿Cuándo usar un Agente y cuando un Servidor? 177
3.2. Creando y Usando Agentes 177
3.3. Actualización segura 178
3.4. Acción de las Clausuras 178
4. Registros 178
4.1. Creando y Usando un Registro 179
4.2. Desregistro de información 180
4.3. Filtrado de contenido 180
4.4. Registro de Nombres 182
4.5. Entregas (***dispatch***) 184
4.6. Publicación/Suscripción 185
4.7. Particiones 186
11. Aplicaciones y Supervisores 187
1. ¿Qué es una Aplicación? 187
1.1. La Versión del Proyecto 190
1.2. ¿Qué versión de Elixir empleamos? 191
1.3. Entorno o Configuración 192
1.4. Inicio y Fases 193
1.5. Aplicaciones Incluidas y ***Umbrella*** 194
2. Estructura de una Aplicación 195
3. ¿Qué es un Supervisor? 197
4. Aplicación ***cuatro*** 200
5. Probando el Juego 209
12. Ecosistema Elixir 211
1. El comando **mix** 211
2. Dependencias y Hex 215
3. Cambios en el Código 217
4. Preparando un Lanzamiento 224
5. Entornos 226
6. Puesta en Producción 227
7. Actualizaciones en Caliente 228
8. Agregando tareas a mix 230
9. ***Scripting*** con Elixir 232
Apéndices 236
A. Instalación de Elixir 237
1. Instalación en Windows 237
2. Instalación en GNU/Linux 237
3. Instalación en Mac 238
4. Otros métodos 238
4.1. Kiex 238
4.2. Desde código fuente 239

B. La línea de comandos 241
1. Módulos 241
2. Histórico 242
3. Procesos 242
4. Directorio de trabajo 243
5. Ayudantes 243
6. Modo JCL 244
7. Salir de la consola 246
C. Compilador 247
1. Configuración en `mix.exs` 248
2. Configuración para un módulo específico 249

Introducción

La aventura vale la pena en sí misma.
—Amelia Earhart

1. Acerca del autor

La programación es un tema que me ha fascinado desde siempre. A partir del año 2002, a la edad de 22 años, me centré en perfeccionar mis conocimientos sobre C++, el paradigma de la orientación a objetos y sus particularidades de implementación en este lenguaje.

Ese mismo año comencé a interesarme por Java. Al año siguiente aprendí SQL, Perl y PHP, comenzando así una aventura que me ha llevado al aprendizaje de nuevos lenguajes de programación regularmente, siempre con el interés de analizar sus potencias y debilidades. Así es como experimenté también con lenguajes clásicos como Basic, Pascal, Modula-2 y otro tipo de lenguajes de scripting para la gestión de sistemas informáticos como Perl o lenguajes de shell.

En los siguientes 8 años, después de haber tratado con lenguajes imperativos, tanto estructurados como orientados a objetos, con sus particularidades y ecosistemas como son C/C++, Java, Perl, Python, PHP, Ruby, Pascal y Modula-2 entre otros, descubrí Erlang y Elixir.

La máquina virtual de Erlang llamada BEAM me permitió y me permite desarrollar estructuras complejas cliente-servidor con un alto grado de concurrencia, tolerancia a fallos y alta disponibilidad entre otras cosas de forma fácil y sencilla.

A diferencia de Erlang en Elixir encuentro una versatilidad y facilidad para escribir código cada vez más compacto y con una mayor potencia. Desde que comencé a aprender Elixir siempre lo he considerado un superconjunto de Erlang. Lo que puedes hacer con Erlang puedes hacerlo con Elixir y mucho más.

En 2016 tras varios años de experiencia en el desarrollo de sistemas Erlang/OTP decidí centrarme en desarrollar algunas soluciones con Elixir. Desde entonces he podido aprender mucho por el camino y desarrollar no solo en Elixir sino también en Phoenix Framework lo que facilita enormemente el desarrollo de sitios web.

Quizás mi pasión por la plataforma BEAM y encontrar un lenguaje tan agradable de escribir me ha llevado a querer explicar y compartir cómo desarrollar empleando Elixir y embarcarme en la realización de este libro. Espero que podamos disfrutar del camino de la ***alquimia*** juntos a través del análisis y aprendizaje de este lenguaje.

2. Acerca del libro

Siempre es difícil adentrarse en una nueva tecnología. Elixir ha contado casi desde el principio con gran documentación y los libros sobre el lenguaje se han multiplicado a medida que las nuevas versiones comenzaban a salir. En estos momentos la versión estable es 1.7 y existen más de una docena de libros hablando de Elixir y otros aspectos de Elixir como la meta-programación o la puesta en producción. No obstante no hay aún nada en castellano.

He querido aportar al mundo de Elixir un poco más de documentación en castellano sobre cómo desarrollar con este lenguaje y sus herramientas.

Aún así esta es una introducción al lenguaje. Algunos aspectos avanzados se han quedado fuera. En este libro sentamos las bases y planteamos preguntas y lugares de referencia donde podéis seguir o completar vuestro aprendizaje con más información.

3. Objetivo del libro

El objetivo del libro es guiar en el conocimiento del lenguaje y el uso de la plataforma que proporciona Elixir para el desarrollo de soluciones. Para conseguir una mejor visión del lenguaje considero importante:

Explicar los aspectos básicos del lenguaje para comenzar a programar.

> Ya que Elixir no es un lenguaje imperativo, puede ocurrir que su sintaxis sea paradójicamente más fácil para el que no sabe programar que para desarrolladores avanzados de lenguajes como C, Java o PHP.

Conocer las fortalezas y debilidades del lenguaje.

> Como en el uso de cualquier tecnología, es importante tener la capacidad de seleccionar un lenguaje o entorno frente a otro dependiendo del trabajo que se vaya a realizar. En este texto analizamos qué es Elixir y en qué se puede emplear. De esta forma obtendrás una idea clara de posibles casos de uso al enfrentarte a un nuevo desarrollo.

Es importante elegir bien la herramienta y solución a implementar. El objetivo del primer capítulo se centrará en este tema dando una visión de qué es Elixir en qué se suele emplear y los casos de éxito que ha tenido hasta el momento.

El resto de capítulos se centran en temas concretos del lenguaje para introducir al lector en cada aspecto y mostrar cómo funcionan Elixir y BEAM.

4. ¿A quién va dirigido este libro?

Este libro está dirigido a todo aquel que quiera aprender a programar en un lenguaje funcional con control de concurrencia y distribuido. Al que le fascine un poco de meta-programación y aplicar al máximo el principio DRY[1]. Permite igualmente ampliar el vocabulario de programación del lector con nuevas ideas sobre el desarrollo de programas y la resolución de problemas. Esto se aplica tanto a los que comienzan a programar como a los que ya tienen experiencia en la programación, y a aquellos que quieren saber qué puede hacer este lenguaje para tomarlo en consideración en las decisiones tecnológicas de su empresa o proyecto.

Para ***los programadores neófitos*** ofrece una guía de aprendizaje base, una forma rápida de adentrarse en el conocimiento del lenguaje que permite comenzar a desarrollar directamente. Propone ejemplos y preguntas que el programador puede realizar, resolver y responder.

Para ***el programador experimentado*** ofrece un nexo hacia un lenguaje diferente, si el lector proviene del mundo imperativo, o bien relativamente similar a otros vistos (si se tienen conocimientos de Ruby o Erlang). Provee un acercamiento detallado a las entrañas de un sistema desarrollado con una ideología concreta y para un fin concreto. Incluso si ya se conoce Elixir supone un recorrido por lo que ya se sabe, pero desde otro enfoque y con características o detalles que probablemente no se conozcan.

Para ***el desarrollador, analista o arquitecto***, ofrece el punto de vista de una herramienta, un lenguaje y un entorno, en el que se pueden desarrollar un cierto abanico de soluciones de forma rápida y segura. Elixir es aún un lenguaje bastante joven pero sobre una plataforma con muchos años de desarrollo muy probada en producción por muchas empresas conocidas y desconocidas. Permite realizar un recorrido por las potencias del lenguaje y obtener el conocimiento de sus debilidades. Lo suficiente como para saber si es una buena herramienta para desarrollar una solución específica.

Para ***administradores de sistemas*** puede ofrecer un conocimiento de BEAM importante, cómo se gestiona la configuración, configuración de nodos, configuración de aplicaciones y la construcción del artefacto para puesta en producción.

5. Nomenclatura usada

A lo largo del libro encontrarás muchos ejemplos y fragmentos de código. Los códigos aparecen de una forma visible y con un formato distinto al del resto del texto. Tendrán este aspecto:

[1] Don't Repeat Yourself o su traducción: No te repitas a ti mismo.

```
defmodule Hola do
  def mundo do
    IO.puts "Hola mundo!"
  end
end
```

Además de ejemplos con código Elixir, en los distintos apartados del libro hay diferentes bloques que contienen notas informativas o avisos importantes. Sus formatos son los siguientes:

Nota

Esta es la forma que tendrán las notas informativas. Contienen detalles o información adicional sobre el texto para satisfacer la curiosidad del lector.

Importante

Estas son las notas importantes que indican usos específicos y detalles importantes que hay que tener muy en cuenta. Se recomienda su lectura.

Un poco de azúcar...

Empleamos este bloque para indicar cuándo el lenguaje nos permite realizar atajos o emplear otra forma más simple de escribir el código.

Sobre el código nos referimos a las funciones siempre indicando el módulo en el que podemos encontrarlas, el nombre y aridad[2], pudiendo darse esta última como un número simple o un rango: `Kernel.spawn/1-2`.

6. Agradecimientos

Agradecer a mi familia, Marga, Juan Antonio y Ana María, por ser pacientes y dejarme el tiempo suficiente para escribir, así como su amor y cariño. A mis padres por enseñarme a defenderme en esta vida, así como a competir conmigo mismo para aprender y superarme en cada reto personal y profesional.

[2] La aridad es el número de parámetros que soporta una función.

7. Más información en la web

Para obtener información sobre las siguientes ediciones, fe de erratas, comentarios, contactos y ayuda sobre el libro puedes acceder a la siguiente dirección:

https://books.altenwald.com/elixir

Además puedes acceder a más contenido relacionado con Erlang/OTP, BEAM y Elixir en la web editorial de Altenwald:

https://books.altenwald.com/

Capítulo 1. Lo que debes saber sobre Elixir

Todo gran logro requiere tiempo.
—Maya Angelou

Elixir se ha expandido muy rápidamente desde sus inicios. Los servicios en la nube requieren de soluciones de alta capacidad como Erlang o Elixir para soportar la inmensa cantidad de peticiones a un solo sistema llegando desde distintos puntos del planeta. Al principio solo las empresas de telecomunicaciones junto con la creciente demanda por parte de startups dedicadas a la mensajería instantánea parecían emplear este tipo de soluciones. Poco a poco otro tipo de empresas están usando soluciones en sectores como el de las apuestas deportivas, el sector financiero, los videojuegos online o publicidad son claros ejemplos de sectores con estas necesidades.

En este capítulo vamos a dar un repaso rápido a la historia de BEAM para continuar en el punto en el cual José Valim comenzó a elaborar Elixir, las decisiones de diseño que decidió adoptar y finalmente casos de uso que respaldan las buenas elecciones tomadas.

Comenzaremos antes definiendo cada elemento para no confundirnos cuando nos adentremos un poco más en el mundo de Elixir. Enumeraremos las características de su arquitectura, del lenguaje, sus ventajas y sus debilidades.

1. ¿Qué es Elixir?

Podríamos definir Elixir como un lenguaje funcional con capacidad de meta-programación. La mayor parte del lenguaje Elixir está definido en Elixir. El lenguaje permite construcciones muy flexibles como veremos a lo largo de los siguientes capítulos.

Junto con otros lenguajes como Ruby se dice que Elixir tiene mucho azúcar sintáctico[1]. Este término fue acuñado por Peter J. Landin en 1964 para referirse a todas esas construcciones del lenguaje que no aportan ninguna nueva funcionalidad pero ayudan enormemente en la facilidad de escribir código.

El azúcar sintáctico presente en Elixir es muy abundante. Como dijimos antes, la mayor parte de Elixir está escrito en Elixir y es gracias a su sistema de meta-programación. Todo gira en torno a las funciones, las macros y los flujos de datos.

[1] https://es.wikipedia.org/wiki/Az%C3%BAcar_sint%C3%A1ctico

En otros lenguajes estructuras condicionales como ***if*** pertenecen al propio lenguaje. En Elixir esta estructura condicional está definida a través del propio lenguaje. Veremos esto en profundidad en el Capítulo4, ***Las funciones y los módulos***.

Los lenguajes reservan una serie de palabras clave. No podemos usar estas palabras reservadas como nombres de variables, funciones, clases u otro tipo de construcción que nos permita el lenguaje. En lenguajes como Cobol podemos encontrar hasta 400 palabras reservadas, C++ dispone de 84 palabras reservadas, Java y Rust tienen 52, Ruby 41, Erlang tan solo 26 pero uno de los lenguajes con menos palabras reservadas es precisamente Elixir con tan solo 8.

Al igual que Erlang en Elixir podemos emplear los procesos de BEAM. Esto dota al lenguaje de capacidad para el desarrollo de soluciones de alta concurrencia, alta capacidad para aceptar gran cantidad de peticiones simultáneas, tolerancia a fallos y además nos permite emplear código realizado en Erlang/OTP dada la gran interoperatividad que tiene Elixir con Erlang/OTP.

En resumen podemos decir que Elixir es un lenguaje orientado a la meta-programación, con base funcional, diseñado para construir aplicaciones a nivel de servidor tolerantes a fallos, concurrentes y de alta capacidad. Pero aún hay mucho más porque como veremos más adelante una de las potencias de Elixir es su extensibilidad y esto nos proporciona una versatilidad a la hora de desarrollar increíble.

2. Características de Elixir

Para definir Elixir hemos nombrado muchas de las características de que dispone el lenguaje. En este sentido me gustaría separar lo que nos permite realizar el lenguaje de lo que nos permite realizar su máquina virtual (BEAM) y es común a todos los lenguajes compilados y que se ejecutan sobre ella.

2.1. Características de BEAM

Comenzando desde la base podemos decir que BEAM es una máquina virtual desarrollada y mantenida en producción por muchas empresas desde hace muchos años. Nació para dar soporte a sistemas de telecomunicaciones con unas características basadas en la tolerancia a fallos, alto nivel de concurrencia, alta capacidad, tiempo real blando y actualización en caliente para evitar la parada de los sistemas.

Vamos a explicar un poco más qué significa cada una de estas características:

Tolerancia a fallos

El código se aisla completamente en unidades mínimas de ejecución llamadas procesos. Un proceso es iniciado y debido a un fallo puede ser terminado. Al mantener cada proceso aislado y sin memoria compartida BEAM nos garantiza la consistencia en la memoria interna de cada uno de los procesos no afectados por el error. De esta forma la parte afectada por el error puede ser desechada y/o reiniciada y el sistema seguir funcionando correctamente.

Alto Nivel de Concurrencia

Al no existir memoria compartida los recursos no son compartidos en ningún momento. El modelo de protección de sección crítica se basa en mantener la unidad de ejecución conjutamente con los datos que maneja y ningún otro código puede acceder a esa información directamente. Solo a través de ese código. Eso nos garantiza un control de acceso a información compartida y evita situaciones de volatilidad de datos.

Alta Capacidad

BEAM define unos procesos internos mucho más ligeros que los proporcionados por el sistema operativo. Esto nos permite generar tantos procesos como necesitemos. De hecho la mayoría de sistemas operativos limita el número de procesos a 65536 (16 bits) mientras que los procesos que pueden ser creados en BEAM son un poco más de 134 millones. Esta característica dota a BEAM de la capacidad para poder atender a miles y millones de peticiones con un coste computacional menor.

Tiempo Real Blando

Al no contener memoria compartida el recolector de basura de BEAM no necesita bloquear ningún proceso para poder actuar. Esto se traduce en la eliminación de cortes en el funcionamiento normal de la máquina virtual y poder trabajar en tiempo real blando. Además veremos más adelante cómo mide el tiempo BEAM para poder proporcionar mediciones reales de tiempo y no basadas en los continuos cambios de hora que realizan los sistemas informáticos.

Actualización en Caliente

BEAM es de las únicas máquinas virtuales que dispone de un sistema seguro para el cambio de código en caliente. Permite modificar el código de un módulo aún manteniendo una copia

antigua para los procesos que aún se mantienen en ejecución e ir migrando de forma segura a la nueva versión del módulo sin tirar el sistema[2].

2.2. Características de Elixir

En la definición adelantamos muchas de las características de Elixir como lenguaje. Una de sus grandes potencias es la meta-programación que nos ayuda a evitar repetir una y otra vez los mismos códigos y reduce al mínimo las típicas plantillas de código a completar cuando se usan frameworks. Parte del código se ejecuta en tiempo de compilación con objetivo de generar otro código más completo o preciso y así mejorar el rendimiento en tiempo de ejecución.

En palabras de José Valim el lenguaje Elixir es un lenguaje dinámico enfocado en la productividad, en la extensibilidad y en la concurrencia.

Lenguaje Dinámico

Es de tipado dinámico y no es fuertemente tipado. Los tipos son opcionales y con fines de comprobación y documentación. Al igual que en Erlang podemos definirlos como una especificación adicional. Veremos esto más adelante.

Productividad

Al escribir código que genera código permite eliminar la necesidad de escribir muchas partes repetitivas y agregar construcciones específicas en lógica de negocio facilitándonos y reduciendo la generación de código. Pero con esta declaración José va aún más allá e incluye la necesidad de tener documentación de primera clase y un conjunto de herramientas como **mix** para la creación, administración y construcción de los proyectos, ***ExUnit*** para la generación de pruebas unitarias, **iex** y el repositorio de paquetes *Hex*.

Extensibilidad

A través de la definición de macros, guardas, funciones, sigilos y sobretodo protocolos nos permite agregar muchas más construcciones al lenguaje. Además José hace hincapié en las estructuras y el polimorfismo como otra piedra angular para la extensibilidad. Este aspecto es muy importante porque el foco de

[2]El cambio en caliente no es una característica que se use mucho actualmente pero para sistemas ***non-stop*** puede eliminar muchos dolores de cabeza y permitirnos cumplir mejor los SLA si diseñamos bien el sistema.

Elixir está en proveer a los programadores de las herramientas necesarias para adaptar el lenguaje a su modelo de dominio[3].

Concurrencia

Dada por BEAM. A través de comportamientos definidos desde la base a través de OTP[4] el lenguaje construye elementos permitiendo mantener la base de paso de mensajes, mantener la sección crítica y facilitar la generación de miles y millones de procesos.

Por mi parte considero la interoperatividad con Erlang otra de las grandes características del lenguaje. Aún teniendo muchas más características que el primero es posible sin mucho problema agregar código desde Erlang y emplearlo desde Elixir de forma fácil.

Además otra de las grandes características facilitadas en el lenguaje Elixir es la evaluación perezosa y/o la creación de generadores. Mientras que otros lenguajes requieren de obtener toda la información y todos los datos para poder proceder al siguiente paso del algoritmo, Elixir nos da la posibilidad de evaluar los elementos uno a uno en la cadena de proceso del algoritmo ahorrando así memoria y tiempo de procesador.

3. Historia de BEAM

Joe Armstrong asistió a la conferencia de Erlang Factory de Londres, en 2010, donde explicó la historia de la máquina virtual de Erlang. En sí, es la propia historia de Erlang/OTP. Sirviéndome de las diapositivas[5] que proporcionó para el evento, vamos a dar un repaso a la historia de Erlang/OTP.

La idea de Erlang surgió por la necesidad de Ericsson de acotar un problema que había surgido en su plataforma AXE, que estaba siendo desarrollada en PLEX, un lenguaje propietario. Joe Armstrong junto a dos colegas, Elshiewy y Robert Virding, desarrollaron una lógica concurrente de programación para canales de comunicación. Esta álgebra de telefonía permitía a través de su notación describir el sistema público de telefonía (POTS) en tan solo quince reglas.

A través del interés de llevar esta teoría a la práctica desarrollaron modelos en Ada, CLU, Smalltalk y Prolog entre otros. Así descubrieron que el álgebra telefónica se procesaba de forma muy rápida en sistemas de alto nivel, es decir, en Prolog, con lo que comenzaron a desarrollar un sistema determinista en él.

[3] El modelo de dominio es el ***lenguaje*** propio de negocio hablado por la empresa o programadores. Puedes encontrar una definición un poco más extensa aquí: https://altenwald.org/2010/04/26/modelo-de-dominio-la-importancia-de-los-nombres/.
[4] OTP son las siglas para ***Open Telecom Platform*** construido como framework de Erlang y provee la base para supervisores, servidores, máquinas de estados, generadores de eventos y aplicaciones.
[5] http://www.erlang-factory.com/upload/presentations/247/erlang_vm_1.pdf

La conclusión a la que llegó el equipo fue que, si se puede resolver un problema a través de una serie de ecuaciones matemáticas y portar ese mismo esquema a un programa de forma que el esquema funcional se respete y entienda tal y como se formuló fuera del entorno computacional, puede ser fácil de tratar por la gente que entiende el esquema, incluso mejorarlo y adaptarlo. Las pruebas realmente se realizan a nivel teórico sobre el propio esquema, ya que algorítmicamente es más fácil de probarlo con las reglas propias de las matemáticas que computacionalmente con la cantidad de combinaciones que pueda tener.

Prolog no era un lenguaje pensado para concurrencia, por lo que se decidieron a realizar uno que satisficiera todos sus requisitos, basándose en las ventajas que habían visto de Prolog para conformar su base. Erlang vio la luz en 1986, después de que Joe Armstrong se encerrase a desarrollar la idea base como intérprete sobre Prolog, con un número reducido de instrucciones que rápidamente fue creciendo gracias a su buena acogida. Básicamente, los requisitos que se buscaban cumplir eran:

- Los procesos debían ser una parte intrínseca del lenguaje, no una librería o framework de desarrollo.
- Debía poder ejecutar desde miles a millones de procesos concurrentes y cada proceso ser independiente del resto, de modo que si alguno de ellos se corrompiese no dañase el espacio de memoria de otro proceso. Es decir, el fallo de los procesos debe ser aislado del resto del programa.
- Debe poder ejecutarse de modo ininterrumpido, lo que obliga a no detener su ejecución para actualizar el código del sistema. Recarga en caliente.

En 1989, el sistema estaba comenzando a dar sus frutos, pero surgió el problema de que su rendimiento no era el adecuado. Se llegó a la conclusión de que el lenguaje era adecuado para la programación que se realizaba pero tendría que ser unas 40 veces más rápido como mínimo.

Mike Williams se encargó de escribir el emulador, cargador, planificador y recolector de basura (en lenguaje C) mientras que Joe Armstrong escribía el compilador, las estructuras de datos, el heap de memoria y la pila; por su parte Robert Virding se encargaba de escribir las librerías. El sistema desarrollado se optimizó a un nivel en el que consiguieron aumentar su rendimiento en 120 veces de lo que lo hacía el intérprete en Prolog.

En los años 90, tras haber conseguido desarrollar productos de la gama AXE con este lenguaje, se le potenció agregando elementos como distribución, estructura OTP, HiPE, sintaxis de bit o compilación de patrones para ***matching***. Erlang comenzaba a ser una gran pieza de

software, pero tenía varios problemas para que pudiera ser adoptado de forma amplia por la comunidad de programadores. Desafortunadamente para el desarrollo de Erlang, aquel periodo fue también la década de Java y Ericsson decidió centrarse en ***lenguajes usados globalmente*** por lo que prohibió seguir desarrollando en Erlang.

Nota

HiPE es el acrónimo de ***High Performance Erlang*** (Erlang de Alto Rendimiento) que es el nombre de un grupo de investigación sobre Erlang formado en la Universidad de Uppsala en 1998. El grupo desarrolló un compilador de código nativo de modo que la máquina (BEAM) virtual de Erlang no tenga que interpretar ciertas partes del código si ya están en lenguaje máquina mejorando así su rendimiento.

Con el tiempo, la imposición de no escribir código en Erlang se fue olvidando y la comunidad de programadores de Erlang comenzó a crecer fuera de Ericsson. El equipo OTP se mantuvo desarrollando y soportando Erlang que, a su vez, continuó sufragando del proyecto HiPE y aplicaciones como EDoc o Dialyzer.

Antes de 2010 Erlang agregó capacidad para SMP y más recientemente para multi-core. La revisión de 2010 del emulador de BEAM se ejecuta con un rendimiento 300 veces superior al de la versión del emulador en C, por lo que es 36.000 veces más rápido que el original interpretado en Prolog. Cada vez más sectores se hacen eco de las capacidades de Erlang y cada vez más empresas han comenzado desarrollos en esta plataforma por lo que se augura que el uso de este lenguaje siga al alza.

4. Comenzando la idea de Elixir

El nombre de Elixir no tiene un origen definido. José Valim dijo en la Elixir Conf de 2014[6] que no tuvo una razón específica para elegir el nombre. No obstante y gracias a su símil con la mezcla de elementos para la fabricación de elixires los programadores se atribuyen el nombre de alquimistas, la herramienta de construcción el nombre de ***mix*** (mezclar) y el gestor de paquetes ***hex*** (maleficio). Además del nombre de conocidas librerías como ***poison*** (veneno) para el tratamiento de JSON o ***distillery*** (destilería) para el empaquetado de un lanzamiento.

El autor del lenguaje ha concedido cientos de entrevistas y ha hablado sobre el lenguaje, cada liberación y cada nueva característica de forma repetida en varios medios de comunicación. Sobre todo en charlas como la Erlang User Conference de 2013[7] y más recientemente las ElixirConf[8].

[6] https://www.youtube.com/watch?v=aZXc11eOEpI
[7] https://www.meetup.com/es-ES/Stockholm-Erlang-Meetup/events/117945802/
[8] https://www.elixirconf.eu/elixirconf2015

José Valim ha sido uno de los contribuidores más activos a Rails y ante los problemas existentes para poder hacer funcionar Rails con websockets o la seguridad entre hilos al ejecutar Rails en un entorno con múltiples núcleos se decidió a probar nuevas vías. Nuevas máquinas virtuales. Ruby es un lenguaje que desde hace bastante tiempo tiene varias implementaciones y sus usuarios según sus necesidades y gustos pueden elegir una u otra base para desarrollar sus aplicaciones.

José Valim mencionó en su charla un estudio de Herb Sutter que acuñó la famosa frase ***se acabó la comida gratis***[9] haciendo mención al hecho del cambio de paradigma al crear los nuevos procesadores donde no se incrementaría más la velocidad del procesador sino el número de núcleos.

José Valim pertenecía al núcleo de desarrollo de Ruby on Rails por esa época y se encontró de frente con el problema. Ruby no estaba preparado para emplear diferentes núcleos en el sistema operativo y las soluciones desarrolladas hasta 2009 eran simples parches para intentar solventar el problema sin mucho éxito.

Al principio la idea de Elixir fue construir un Ruby sobre BEAM. Tal y como había intentado hacer Reia[10] anteriormente. José Valim se basó en Reia para iniciar Elixir[11] y cuando Elixir comenzó a incrementar la velocidad de su desarrollo fue cuando el desarrollador principal de Reia consideró oportuno abandonar su proyecto y dar su apoyo a Elixir.

Hay que considerar importante el apoyo de la empresa Plataformatec[12] donde José Valim trabajó durante la gestación del lenguaje. Según cuenta José Valim en charlas como la mencionada Elixir Conf de 2014 la empresa financió su trabajo en el lenguaje durante dos años hasta convertirlo en un lenguaje de alto interés y muy aceptado por muchos programadores del sector.

Desde mi punto de vista considero que hubo un antes y un después de la lectura del artículo de Joe Armstrong[13] en mayo de 2013 hablando de la excitación de Dave Thomas por Elixir y la gestación del primer libro sobre Elixir[14]. En el mes siguiente O'Reilly a través de Simon Saint Laurent publicó Introducing Elixir[15]. El lenguaje comenzó a ganar masa crítica y comienzan a aparecer muchos más eventos sobre el lenguaje.

[9] Traducido de la frase en inglés original: ***free lunch is over***.
[10] http://reia-lang.org/
[11] http://www.unlimitednovelty.com/2011/06/why-im-stopping-work-on-reia.html
[12] http://plataformatec.com.br/
[13] https://joearms.github.io/published/2013-05-31-a-week-with-elixir.html
[14] https://pragprog.com/book/elixir/programming-elixir
[15] http://shop.oreilly.com/product/0636920030584.do

5. Desarrollos con Elixir

A nivel empresarial Elixir es usado por empresas tan grandes como Adobe, AdRoll, Lexmark, PagerDuty, Pinterest o Slack[16].

Dado su origen el principal uso de Elixir está en la elaboración de servicios web y websocket para servicios en Internet. Uno de los grandes frameworks que posibilita y facilita esta adopción es Phoenix Framework[17]. Quizás por su influencia este framework tiene mucha semejanza con Ruby on Rails[18] pero difiere en muchos aspectos y ha conseguido hitos bastante notables como veremos en la siguiente sección.

6. Soportando 2 Millones de Usuarios Conectados

A finales de 2015 Chris McCord[19] comenzó a publicar una serie de tuits donde mencionaba cómo estaba probando e intentando alcanzar 2 millones de usuarios concurrentes[20] conectados por websocket en Elixir y más específicamente en Phoenix Framework.

Todo comenzó cuando Gary Rennie[21] estaba haciendo pruebas sobre cuántos canales simultáneos podría soportar Phoenix Framework. Conseguía un máximo de mil (1000) en su máquina local. Al comentarlo vía IRC y ante la falta de bancos de pruebas los desarrolladores del núcleo de Phoenix Framework quisieron arrojar un poco de luz sobre cómo realizar las pruebas y aportar números.

Rackspace aportó tres máquinas de 15GB I/O v1 con 15GB de RAM y 4 núcleos cada una. También tuvieron acceso a un OnMetal I/O de 128GB con 40 núcleos.

Las pruebas para generar el número crítico de clientes se realizaron a través de Tsung[22], un sistema para realizar bancos de pruebas en diferentes protocolos y generar una alta carga. La primera configuración probada fue con un servidor configurado para aceptar las peticiones y los otros dos para generar el tráfico cliente usando Tsung.

Se alcanzaron 27 mil conexiones simultáneas. José Valim hizo una corrección en el código de Phoenix para acelerar la entrada de usuarios.

[16] http://elixir-companies.com/
[17] http://phoenixframework.org/
[18] https://rubyonrails.org/
[19] https://github.com/chrismccord
[20] http://phoenixframework.org/blog/the-road-to-2-million-websocket-connections
[21] https://github.com/Gazler
[22] http://tsung.erlang-projects.org/

Se repitieron las pruebas y esta vez se alcanzaron 50 mil conexiones simultáneas.

A través de una visualización de los procesos mediante ***observer***[23] Chris pudo detectar otro cuello de botella. Chris eliminó el sistema de heartbeat de la conexión de websocket al ver que se duplicaba el número de temporizadores en uso. Tanto el sistema de websocket como cowboy implementaban un temporizador. Dejó únicamente el provisto por cowboy en una nueva modificación de Phoenix Framework. De nuevo ejecutaron las pruebas y esta vez llegaron a 100 mil conexiones simultáneas.

Debido a las limitaciones de tener solo dos máquinas para clientes y solo poder generar 40 mil y 60 mil conexiones simultáneas respectivamente, emplearon otra máquina de 128GB disponible para conseguir más clientes y seguir las pruebas. Esta vez consiguieron 330 mil conexiones simultáneas.

Gabi Zuniga[24] echó un vistazo y agregó una corrección al sistema. Empleando otro tipo de dato para las tablas ETS consiguieron llegar a 450 mil conexiones simultáneas.

La empresa Live Help Now[25] aportó a la prueba 45 máquinas en Rackspace para proseguir gracias a la participación de Justin Schneck[26] en las pruebas. Configuraron Tsung en unas cuantas máquinas y lo limitaron a 1 millón de conexiones simultáneas para hacer las pruebas de nuevo. Terminaron de configurar el resto de las 45 máquinas para probar si era posible llegar a 2 millones de conexiones en una sola máquina. Desafortunadamente se quedaron en 1,3 millones de conexiones. Nada mal no obstante.

Pero algo no estaba bien. A esos niveles de carga los mensajes para los suscriptores se demoraban más de 5 segundos. Tras varios análisis Chris tuvo la idea de generar un concentrador (pool) para pubsub y Justin la de fraccionar (shard) la tabla ETS donde se almacenaba todo. Esta combinación permitió reducir el tiempo de difusión a un segundo y alcanzar 2 millones de conexiones simultáneas.

Cada escenario concreto tiene sus propias limitaciones y como hemos visto pasar de mil conexiones simultáneas a dos millones ha tenido una travesía de análisis y correcciones permitiendo a los desarrolladores ir mejorando cada vez más el sistema. José, Chris, Gary y todos los implicados quedaron satisfechos con la cifra alcanzada aunque comenta

[23] Una aplicación de Erlang/OTP que permite obtener información en tiempo real del sistema en ejecución a través de una interfaz gráfica de usuario (GUI).
[24] https://twitter.com/gabiz
[25] http://www.livehelpnow.net/
[26] https://twitter.com/mobileoverlord

también que tras la publicación siguieron encontrando otras mejoras para poder aplicar.

La buena noticia para todos es que estas mejoras y las futuras que se han ido descubriendo han ido formando parte de Elixir y/o Phoenix Framework y eso nos garantiza un sistema robusto y con un alto grado de escalabilidad horizontal.

Capítulo 2. El lenguaje

Un mar tranquilo nunca hizo a un marinero hábil.
—Proverbio

La sintaxis de Elixir está basada en otros lenguajes como Ruby y Erlang y agrega el suficiente azúcar sintáctico para dar flexibilidad al programador para escribir código simple. Permite expresar fácilmente el código escrito. No obstante debemos recordar que Elixir es un lenguaje funcional. La especificación de la solución al problema a resolver mediante código difiere un poco al planteamiento en otros lenguajes imperativos como Java, C/C++, Perl o PHP.

En la programación funcional el código parece una función matemática:

```
def area(base, altura), do: base * altura
```

En este ejemplo definimos una función en Elixir. El área de un rectángulo. Los parámetros son ***base*** y ***altura***. El contenido de la función en una sola línea nos retorna la multiplicación de la base por la altura.

Para códigos imperativos como el siguiente no hay una correlación exacta:

```
para i <- 1 hasta 10 hacer
    si clavar(i) = 'si' entonces
        martillea_clavo(i)
    fsi
fpara
```

Podemos obtener algo parecido a través de la construcción ***for*** de Elixir de la siguiente forma:

```
for i <- 1..10,
    clavar(i) == "si",
    do: martillea_clavo(i)
```

En estas construcciones simplificamos las expresiones definiendo qué queremos conseguir y no los pasos explícitos uno a uno. Otra construcción típica de Elixir para este problema podría ser:

```
1..10
|> Enum.filter(&(clavar(&1) == "si"))
|> Enum.each(&martillea_clavo/1)
```

Obteniendo los números del rango ***1..10*** filtramos llamando a la función `clavar/1` para cada uno de los elementos y para la lista restante llamamos a `martillea_clavo/1` para cada elemento.

También podemos expresar con evaluación perezosa el mismo código y obtenemos un sistema de ejecución en cadena:

```
1..10
|> Stream.filter(&(clavar(&1) == "si"))
|> Enum.each(&martillea_clavo/1)
```

Esto nos garantiza recorrer la lista de elementos una única vez aplicando todas las funciones por cada elemento en el momento que se va obteniendo del rango. Es muy recomendable para acciones como procesar las líneas de un fichero, los registros de una base de datos o el flujo de información proveniente desde una conexión de red.

No te preocupes si de momento no entiendes estos códigos a lo largo de los siguientes capítulos iremos abordando cada uno de estos conceptos.

1. El intérprete de Elixir (iex)

La mayoría del código de ejemplo lo vamos a ejecutar en la línea de comandos o el intérprete de Elixir. Antes de continuar te recomiendo que instales Elixir y pruebes a acceder al intérprete ejecutando **iex** en una consola de tu sistema operativo. Puedes ver cómo instalar Elixir en el ApéndiceA, ***Instalación de Elixir***.

Al acceder al intérprete deberías ver algo como lo siguiente:

```
$ iex
Erlang/OTP 21 [erts-10.1.3] [source] [64-bit] [smp:8:8]
 [ds:8:8:10] [async-threads:1] [hipe] [kernel-poll:false]
 [dtrace] ❶

Interactive Elixir (1.7.4) - press Ctrl+C to exit (type h()
 ENTER for help) ❷
iex(1)> ❸
```

❶ La primera línea es información sobre BEAM indicando la versión base de Erlang/OTP. En este caso ejecutamos una OTP 21 y según la versión de erts (10.1.3) es OTP 21.1. Nos indica la instalación desde código fuente (source) ejecutándose en un sistema de 64 bits, con uso de SMP[1], seguido por la configuración del programador de tareas (ds:8:8:10), los

[1]SMP es Symmetric Multi-Processing o Multiprocesamiento Simétrico. Indica que hay varias unidades de procesamiento (cores o núcleos) acceden a la misma memoria.

hilos asíncronos lanzados, el uso de HiPE[2], poll del kernel[3] y dtrace[4].

❷ Información del intérprete de Elixir. Nos proporciona la versión de Elixir que estamos usando (1.7.4), la forma de salir (**Ctrl**+**C**) y cómo obtener ayuda sobre los comandos especiales que podemos emplear dentro del intérprete. Puedes escribir **help** para revisar la ayuda.

❸ El símbolo del sistema nos da información de dónde estamos (**iex**) y el número del comando que vamos a escribir. Esto es útil para poder repetir comandos u obtener una salida anterior en un nuevo comando si olvidamos asignarlo a una variable:

```
iex(1)> 1 + 1
2
iex(2)> v(1) * 2
4
```

En lo sucesivo mostraremos los ejemplos pero recortando el número de comando de este símbolo del sistema para no distraer del código en sí.

Puedes revisar la sección anterior y probar a escribir los códigos de ejemplo proporcionados y jugar un poco antes de proseguir.

Para más información puedes ver el ApéndiceB, ***La línea de comandos***.

2. Azúcar sintáctico

La definición de las funciones de ayuda es **h()** y **help**. Ambas son funciones y pueden incluso ejecutarse de forma diferente como **h** y **help()**. Esto forma parte del azúcar sintáctico. Es una construcción que no agrega ninguna nueva funcionalidad al lenguaje pero facilita la lectura y escritura. Es más fácil escribir **help** como comando en el intérprete que **help()**.

Donde más notaremos las mejoras que propone la sintaxis propia de Elixir es en el uso de funciones, en los bloques ***do..end*** y en la especificación de clausuras o funciones anónimas.

[2]High Performance Erlang (HiPE) o Erlang de Alto Rendimiento es un proyecto de BEAM para dotar a la máquina virtual de compilación nativa y por lo tanto mayor rendimiento.
[3]Poll es una técnica de E/S para delegar la espera por entrada de datos al núcleo del sistema operativo y liberar el programador de tareas.
[4]Herramienta originaria de Solaris pero disponible en BSD y Linux que permite obtener eventos del sistema operativo sobre el uso de recursos como la red, disco, memoria y mucho más.

Iremos viendo más adelante cada uno de estos edulcorantes del lenguaje.

3. Comentarios

Los comentarios son esa parte del código no ejecutable que sirve al programador para aclarar una parte del código. En elixir podemos agregar comentarios empleando el símbolo almohadilla (#). Todo lo que escribamos tras este símbolo será tomado como comentario e ignorado por el compilador.

Podemos ver un ejemplo de comentario en el siguiente código:

```
# Este código calcula el área del rectángulo:
area = base * altura  # multiplica área por base
```

Obviamente no necesitamos tantos comentarios en un código tan pequeño pero nos sirve para ilustrar cómo podemos agregar los comentarios.

4. Tipos de Datos

En Elixir disponemos de distintos tipos de datos desde simples átomos o números hasta estructuras compuestas por otros tipos de datos, mapas y listas. Cada tipo de dato tiene sus propias particularidades y su cometido.

En esta sección nos adentraremos en los tipos de datos para ver cada uno de ellos respondiendo tres preguntas básicas: ¿qué son?, ¿para qué se usan?, ¿cómo se usan?

Cada tipo de dato además tiene asociado un módulo que permite realizar acciones sobre el mismo o conversiones desde otros tipos de datos. No podemos ver en detalle cada función de cada módulo pero agregaremos un enlace a la documentación en línea de cada uno de ellos para que puedas revisar qué otras acciones puedes realizar con ese tipo de dato o qué opciones pueden ser usadas con las funciones mencionadas aquí.

4.1. Átomos

Los átomos son una unidad muy pequeña y auto-explicativa que podemos usar para agregar código más legible y errores más comprensibles en consola al emplear texto en lugar de números para indicar estados, códigos de error o indicadores para configuración entre otros.

Este tipo de dato nos ayuda a dar nombre al contenido de variables sin necesidad de ocupar mucho tamaño en memoria. Por ejemplo es

trivial crear valores numéricos en otros lenguajes para indicar estados, algo como asignar 0 a "nuevo", 1 a "en proceso", 2 a "cancelado" y 3 a "pagado". La lectura del código se puede dificultar e incluso si podemos crear constantes, tenemos el engorro de tener que crearlas y mantenerlas.

Otra solución sería emplear cadenas de texto para obtener mejor código para leer pero con mayor uso de memoria y más lento para procesar si un texto es igual a otro o no.

Para solventar este problema surgieron los átomos. Este tipo de dato se expresa con un texto comprensible para quien lee el código. En el ejemplo anterior, podemos usar los átomos: ***:nuevo***, ***:en_proceso***, ***:cancelado*** y ***:pagado***. El uso de memoria es incluso mejor en comparación con emplear números y el rendimiento mucho mayor que si empleásemos cadenas de texto.

Los átomos se pueden obtener de dos formas diferentes. La primera es escribiendo con la primera letra mayúscula el átomo y la segunda es anteponiendo al texto que emplearemos como átomo el símbolo de dos puntos (:).

```
iex> is_atom Atom
true
iex> is_atom :atom
true
iex> is_atom :my_atom_12
true
```

Hay varios textos que definen un átomo válido como la mezcla de letras, números, signo de subrayado (_) y arroba (@) como únicos valores válidos para formar el átomo. Pero de hecho hay bastantes excepciones. Cualquier operador puede ser un átomo también (***:===*** es un átomo válido). El signo de interrogación se permite pero solo como terminador: ***:is_binary?***.

Para el resto de casos podemos siempre emplear las dobles comillas (") y encerrar el código entre ellas para que nos permita crear átomos como ***:"12"***.

Importante

Cada átomo definido se inserta en una tabla. El número máximo de átomos por defecto es 1.048.576. Si se intentan definir más de este número se produce un error crítico y BEAM termina su ejecución al completo.

Podemos en este ejemplo lo que sucede:

```
iex> 0..2000000
  |> Enum.each(&(String.to_atom("a#{&1}")))
no more index entries in atom_tab (max=1048576)

Crash dump is being written to:
 erl_crash.dump...done
```

El código intenta generar 2 millones de átomos y finaliza la ejecución de BEAM al llegar al límite máximo de átomos.

Para evitar esto es importante no emplear átomos generados de forma dinámica y si en algún caso deben emplearse es preferible usar la función `String.to_existing_atom/1` siempre que sea posible. Esta función fallará si la cadena a transformar no encuentra un átomo ya existente en la tabla de átomos ya empleados.

El módulo ***Atom*** provee únicamente un par de funciones para convertir un átomo a una lista de caracteres con `Atom.to_charlist/1` o una cadena de texto `Atom.to_string/1`.

4.2. Booleanos o Lógicos

Este tipo de dato solo tiene dos representaciones posibles ***true*** o ***false***. Se obtienen principalmente a través del uso de los literales y los operadores de comparación además de los nexos lógicos.

Debido a ser un tipo de dato tan pequeño no hay disponible un módulo para él.

Un poco de azúcar...

En sí la representación interna de este dato no existe. Si pruebas a emplear los átomos ***:true*** y ***:false*** en su lugar verás que obtienes los mismos resultados. Incluso puedes hacer una comparación estricta para obtener la prueba de que en realidad Elixir emplea esos átomos internamente:

```
iex> true === :true
true
```

4.3. Valor nulo *nil*

El valor nulo o ***nil*** es otro átomo con significado especial que permite escribirse si el uso de los dos puntos (:). El significado de este átomo es la ausencia de tipo. Cuando asignamos este valor a una variable estamos indicando que no dispone de tipo.

Podemos verlo de la siguiente forma:

```
iex> :nil
nil
iex> :nil == nil
true
```

Hay que tener presente que en funciones como `Kernel.is_atom/1` usando una variable o el valor directo obtenemos siempre ***true*** ya que se trata de un átomo.

4.4. Números Enteros y Reales

En Elixir puedes trabajar con dos tipos de números: enteros y reales. Los números enteros no tienen decimales y el tamaño representado puede ser tan grande como memoria tengas en el sistema.

Un poco de azúcar...

Para representar número grandes como 1000000 podemos emplear el símbolo de subrayado (_) como separador y mejorar así la legibilidad: 1_000_000. El compilador ignora estos símbolos y toma el número como si lo hubiésemos escrito sin ellos. En librerías como Money[5] si creamos un dato en euros el número entero a emplear lo escribimos en céntimos, por ejemplo así 10_50 para indicar 10 euros y 50 céntimos.

Podemos ver algunos ejemplos:

```
iex> 1 * 2
2
iex> 12_000_500
12000500
iex> 1_000 * 1_000 == 1_000_000
true
```

El módulo ***Integer*** nos provee de funciones útiles para convertir a cadena `Integer.to_string/1`, a lista de caracteres `Integer.to_charlist/1`, y otras más exóticas como

[5] https://github.com/liuggio/money

`Integer.digits/1` para separar los dígitos de un número o su contrapuesta `Integer.undigits/1` u obtener el máximo común divisor de dos números con `Integer.gcd/2`:

```
iex> Integer.to_string(1_000)
"1000"
iex> Integer.to_charlist(1_024)
'1024'
iex> Integer.digits 101
[1, 0, 1]
iex> Integer.undigits v(-1)
101
iex> Integer.gcd 10, 15
5
```

Podemos especificar la base en la que representamos los números enteros anteponiendo un prefijo específico a cada número escrito. Por ejemplo para los números hexadecimales anteponemos ***0x***, para los números binarios ***0b*** y para los octales el prefijo ***0o***. Podemos ver algunos ejemplos a continuación:

```
iex> 0o10
8
iex> 0x10
16
iex> 0b10
2
```

Los números reales se representan en su forma científica si exceden una cierta dimensión. A la hora de escribirlos podemos optar por escribir el número completo empleando como separador decimal el punto (.) o podemos escribirlos de forma científica especificando la base en formato decimal, la letra ***e*** (para indicar exponente en base a 10) y el número del exponente para la base 10. Escribir ***2.0e1*** (o 2 por 10 elevado a 1) nos da como resultado ***20.0***.

En el módulo ***Float*** podemos encontrar funciones para conversión a lista de caracteres `Float.to_charlist/1` o a cadena de texto `Float.to_string/1` funciones para redondear hacia abajo `Float.floor/1-2` y hacia arriba `Float.ceil/1-2`. También disponemos de un redondeo estándar `Float.round/1-2` y una función para obtener dos números enteros que en fracción generen el número dado: `Float.ratio/2`.

```
iex> Float.to_charlist 0.10
'0.1'
iex> Float.to_string 0.10
"0.1"
iex> Float.floor 0.10
0.0
iex> Float.ceil 0.10
```

```
1.0
iex> Float.round 0.10
0.0
iex> Float.round 0.51
1.0
iex> Float.round 0.52
1.0
iex> Float.ratio 0.5
{1, 2}
```

Por último, para convertir de cadena a número realizamos un análisis de la cadena. Debemos tener en cuenta que una cadena puede contener cualquier cosa y la conversión podría ser errónea. El retorno de las funciones `Float.parse/1` e `Integer.parse/1` retorna una tupla con dos elementos. El primer elemento será el número encontrado y segundo elemento el resto de texto que no corresponde al número:

```
iex> Float.parse "0.10"
{0.1, ""}
iex> Float.parse "a1"
:error
iex> Float.parse "1"
{1.0, ""}
```

Como podéis comprobar, si al principio de la cadena no se encuentra ningún número se retorna el átomo ***:error***.

Antes de terminar con los números me gustaría mostrar un ejemplo para ver la magnitud que un número entero puede tomar. Si ejecutamos el siguiente código veremos varias pantallas de números:

```
iex> List.duplicate(2048, 1000) |> Enum.reduce(&(&1 * &2))
213772730161759466399474188010951...
```

En resumen, el código se encarga de repetir mil veces 2048 dentro de una lista y la reducción se encarga de multiplicar cada elemento por el siguiente. Es lo mismo que elevar 2048 a 1000. Por pantalla obtenemos el resultado.

4.5. Variables e Inmutabilidad

Las variables son nombres a los que se asigna un valor. Podemos emplear un nombre que esté compuesto por letras (mayúsculas y/o minúsculas), números y guion bajo o subrayado (_) aunque con algunas salvedades. No podemos emplear una letra mayúscula al inicio del nombre ni podemos emplear ninguna de las palabras reservadas por Elixir.

En el ejemplo inicial definimos una función para calcular el área de un rectángulo. Si en el intérprete de Elixir definimos un par de variables

conteniendo un número entero cada una y realizamos una operación aritmética con ellas veremos el resultado:

```
iex> base = 10
10
iex> altura = 5
5
iex> base * altura
50
```

Las variables referencian valores. Estos valores pueden ser literales, el dato en sí como hemos visto en el ejemplo anterior, pueden ser otras variables o expresiones que contengan llamadas a funciones y cálculos aritméticos o lógicos.

La variable referenciará el valor asignado hasta que se asigne otro diferente.

A diferencia de otros lenguajes las variables en Elixir referencian espacios de memoria y cuando se asigna otro valor referencian a ese nuevo espacio de memoria. La variable en sí no apunta en ningún momento a un espacio mutable de memoria.

En Elixir existe la inmutabilidad para los datos. Es decir, un tipo de dato como una lista, un mapa, una cadena de texto o incluso un número no cambian. Su espacio de memoria no varía una vez se ha reservado para contenerlo. Si queremos modificar un dato se crea un espacio de memoria nuevo con la modificación. Esta forma de tratar la información interna da seguridad al proceso de control de concurrencia y tiende a ser más rápido que modificar partes de un dato existente.

```
iex> a = 10 ❶
10
iex> b = a ❷
10
iex> a = 5 ❸
5
iex> b
10 ❹
```

❶ Asignamos a la variable ***a*** el valor 10. Elixir lo que hace es reservar un espacio de memoria para un número entero y almacena el número 10.

❷ Copiamos la referencia de ***a*** a ***b***. En esta ocasión como el dato no varía de una variable a otra se copia la referencia y ambas apuntan al mismo espacio de memoria.

❸ Asignamos a la variable ***a*** un nuevo espacio de memoria que contiene el valor 5. El espacio de memoria anterior queda inmutable. No se modifica. Pero la variable ***a*** comienza a apuntar al nuevo espacio de memoria que contiene 5.

❹ La variable ***b*** sigue apuntando al espacio de memoria anterior y por lo tanto sigue conteniendo 10.

El uso de memoria no suele ser un problema porque cada proceso tiene su espacio de memoria y cuando una función termina la reserva de memoria se desecha completamente quedando únicamente los datos del retorno de la función. Si el lenguaje trabajase con referencias a otros datos o incluso retornase referencias, no sería seguro realizar esta limpieza o sería mucho más complicado de realizar.

Por este motivo Elixir puede realizar tareas de tiempo real blando. Al finalizar la ejecución de una función todas las variables internas se desechan y son recogidas por el recolector de basura.

4.6. Listas

Las listas en Elixir son vectores de información heterogénea, es decir, pueden contener información de distintos tipos, ya sean números, átomos, tuplas, estructuras, mapas u otras listas.

Las listas son una de las potencias de Elixir y otros lenguajes funcionales. Al igual que en Lisp, Elixir maneja las listas como lenguaje de alto nivel, en modo declarativo, permitiendo el uso de herramientas como las listas de comprensión o la agregación y eliminación de elementos específicos como si de conjuntos se tratase.

4.6.1. ¿Qué podemos hacer con una lista?

Podemos definir una lista de elementos de forma directa tal y como la presentamos a continuación:

```
iex> [ 1, 2, 3, 4, 5 ]
[1,2,3,4,5]
iex> [ 1, "Hola", 5.0, :hola ]
[1,"Hola",5.0,:hola]
```

Podemos realizar modificaciones de estas listas agregando o sustrayendo elementos con los operadores especiales **++** y **--** tal y como podemos ver en los siguientes ejemplos:

```
iex> [1,2,3] ++ [4]
[1,2,3,4]
iex> [1,2,3] -- [2]
```

```
[1,3]
```

Otra ventaja de las listas es la forma en la que podemos ir tomando elementos de la cabeza de la lista dejando el resto en otra sublista. Podemos realizar esta acción con esta sencilla sintaxis:

```
iex> [cabeza|cola] = [1,2,3,4]
[1,2,3,4]
iex> cabeza
1
iex> cola
[2,3,4]
iex> [cabeza1,cabeza2|cola] = [1,2,3,4]
[1,2,3,4]
iex> cabeza1
1
iex> cabeza2
2
iex> cola
[3,4]
```

De esta forma tan sencilla podemos implementar los algoritmos de manejo de pilas como son ***push*** (empujar a la pila) y ***pop*** (extraer de la pila) de la siguiente forma:

```
iex> lista = []
[]
iex> lista = [1|lista]
[1]
iex> lista = [2|lista]
[2,1]
iex> [extrae|lista] = lista
[2,1]
iex> extrae
2
iex> lista
[1]
```

De esta forma hemos modificado nuestra lista agregando elementos primero y extrayendo un elemento después en nuestra implementación básica de una cola LIFO[6].

4.6.2. Listas de Caracteres

Las listas de caracteres son un tipo específico de lista. Se trata de una lista homogénea de elementos representables como caracteres. Elixir detecta si una lista en su totalidad cumple con esta premisa para tratarla como lista de caracteres.

Por tanto, la representación de la palabra ***Hola*** en forma de lista, se puede hacer como lista de enteros que representan a cada una de las letras o como el texto encerrado entre comillas simples ('). Una demostración:

[6] ***Last In First Out***, último en entrar primero en salir.

```
iex> 'Hola' = [72,111,108,97]
'Hola'
```

Como puede apreciarse, la asignación no da ningún error ya que ambos valores, a izquierda y derecha, son lo mismo para Elixir.

Importante

Esta forma de tratar las cadenas es muy similar a la que se emplea en lenguaje C, donde el tipo de dato ***char*** es un dato de 8 bits en el que podemos almacenar un valor de 0 a 255 y que las funciones de impresión tomarán como representaciones de la tabla de caracteres en uso por el sistema. En Elixir la única diferencia es que cada dato no es de 8 bits sino que es un entero lo que conlleva un mayor consumo de memoria pero mejor soporte de nuevas tablas como la de UTF-16 o las extensiones de UTF-8 y similares.

Si empleamos librerías nativas de BEAM y relativas a Erlang es muy posible que empleemos este tipo de dato en lugar de las cadenas de caracteres.

Al igual que con el resto de listas, las listas de caracteres soportan también la agregación de elementos, de modo que la concatenación se podría realizar de la siguiente forma:

```
iex> 'Hola, ' ++ 'mundo!'
'Hola, mundo!'
```

Una de las ventajas de la asignación es la capacidad de realizar concordancias. Si en una asignación se encuentra una variable que no ha sido enlazada a ningún valor, automáticamente cobra el valor necesario para que la ***ecuación*** sea cierta. Elixir intenta hacer siempre que los elementos a ambos lados del signo de asignación sean iguales. Un ejemplo:

```
iex> 'Hola, ' ++ quien = 'Hola, mundo!'
'Hola, mundo!'
iex> quien
'mundo!'
```

Esta notación tiene sus limitaciones, en concreto la variable no asignada debe estar al final de la expresión, ya que de otra forma el código para realizar la concordancia sería mucho más complejo.

Las listas de caracteres también nos permiten realizar interpolación. La interpolación es la capacidad de agregar la salida de un código dentro de una parte de la lista de caracteres. Por ejemplo:

```
iex> IO.puts 'Una hora son #{60 * 60} segundos'
Una hora son 3600 segundos
:ok
```

El resultado de la operación es intercambiado por el código dentro de la cadena. Todo lo que esté dentro de las llaves y con el símbolo de almohadilla delante (#{}) es intercambiado por el resultado de evaluar ese código.

4.7. Colecciones

En general las listas son consideradas colecciones de datos en Elixir. Pero no solo las listas. Podemos encontrarnos otros tipos de datos que se consideran colecciones y sobre ellos podemos aplicar las funciones del módulo ***Enum***.

Las funciones contenidas en ***Enum*** nos permiten realizar acciones sobre las colecciones como iterar cada elemento retornando un resultado por elemento (`Enum.map/2`) o sin retorno (`Enum.each/2`). Además de ordenar (`Enum.sort/1`), obtener solo los valores únicos (`Enum.uniq/1`), buscar un elemento (`Enum.find/2`) y otras funciones muy potentes y configurables.

El requisito para que un dato pueda emplear las funciones de ***Enum*** es que implemente el protocolo ***Enumerable***. Los módulos que implementan ***Enumerable*** son: Date.Range, File.Stream, Function, GenEvent.Stream, HashDict, HashSet, IO.Stream, List, Map, MapSet, Range y Stream.

Por ejemplo si estamos realizando una salida por pantalla de una lista de elementos y queremos numerarlos:

```
iex> ["pan", "leche", "huevos"] |>
...> Enum.with_index() |>
...> Enum.each(fn({item, i}) -> IO.puts("#{i}.- #{item}") end)
0.- pan
1.- leche
2.- huevos
:ok
```

O si estamos desarrollando una página HTML y queremos agregar tres capas (***div***) dentro de otra que las contenga a modo de simular filas y columnas:

```
iex> ["erlang", "elixir", "golang", "rust"] |>
...> Enum.map(&("    <div>#{&1}</div>\n")) |>
...> Enum.chunk_every(2) |>
...> Enum.map(&("<div>\n#{Enum.join(&1)}</div>\n")) |>
...> Enum.join() |>
...> IO.puts()
```

```
<div>
    <div>erlang</div>
    <div>elixir</div>
</div>
<div>
    <div>golang</div>
    <div>rust</div>
</div>

:ok
```

Más adelante veremos más usos de este módulo.

4.8. Cadenas de texto

Las cadenas de texto o cadenas de caracteres son similares a las listas de caracteres. Permiten almacenar cadenas de caracteres con tamaño de byte y permite realizar trabajos específicos con secuencias de bytes o incluso a nivel de bit.

Los literales los escribimos encerrados en comillas dobles ("). La sintaxis es como sigue:

```
iex> "Hola"
"Hola"
iex> <<72,111,?l,?a>>
"Hola"
```

La cadena de texto puede ser representada como lista binaria tal y como hemos visto en la segunda sintaxis. Estas listas no tiene las mismas funcionalidades que las listas vistas anteriormente. Podemos concatenar cadenas unas a otras con el operador **<>** de la siguiente forma:

```
iex> "Hola " <> "mundo!"
"Hola mundo!"
```

En las cadenas de texto también podemos realizar la interpolación. Este método es más sencillo para concatenar elementos dentro de una cadena. Por ejemplo en el caso anterior si uno de los trozos de la cadena está en una variable y queremos construir la cadena completa podemos hacer:

```
iex> quien = "mundo"
"mundo"
iex> "Hola #{quien}!"
"Hola mundo!"
```

La sintaxis de la almohadilla seguida por las llaves (#{}) dentro de una cadena de texto nos permite intercambiar todo lo encerrado entre esas llaves por la evaluación del código encerrado.

Por último, para trabajar con cadenas de texto disponemos del módulo ***String***. Este módulo nos proporciona funciones para poder cambiar el texto a mayúscula (`String.upcase/1`) o a minúscula (`String.downcase/1`) e incluso capitalizar[7] el texto (`String.capitalize/1`). Podemos darle la vuelta al texto (`String.reverse/1`), obtener su tamaño (`String.length/1`), tomar el primer carácter (`String.first/1`), el último (`String.last/1`) u otro de la cadena especificando posición (`String.at/2`). Veamos algunos ejemplos:

```
iex> titulo = "don Quijote de la Mancha"
"don Quijote de la Mancha"
iex> String.upcase titulo
"DON QUIJOTE DE LA MANCHA"
iex> String.downcase titulo
"don quijote de la mancha"
iex> String.capitalize titulo
"Don quijote de la mancha"
iex> String.reverse titulo
"ahcnaM al ed etojiuQ nod"
iex> String.length titulo
24
iex> String.first titulo
"d"
iex> String.last titulo
"a"
iex> String.at titulo, 4
"Q"
```

Además disponemos en el módulo de funciones que nos permiten comprobar si el texto termina con un sufijo dado (`String.ends_with?/2`), o si comienza con un prefijo dado (`String.starts_with?/2`) o incluso si contiene un trozo de texto dado (`String.contains?/2`). Podemos ver cómo funcionan con estos ejemplos:

```
iex> String.ends_with? titulo, "Mancha"
true
iex> String.ends_with? titulo, "Celestina"
false
iex> String.starts_with? titulo, "Don"
false
iex> String.starts_with? titulo, "don"
true
iex> String.contains? titulo, "Quijote"
true
iex> String.contains? titulo, "De"
false
```

Puedes ver estas y más funciones en la documentación del módulo ***String***[8].

[7] Un texto capitalizado es en el que la primera letra está en mayúscula y el resto en minúsculas.
[8] https://hexdocs.pm/elixir/String.html

4.9. Trabajando con Binarios

Las cadenas de caracteres nos permiten trabajar con cadenas de texto como listas binarias. Esta funcionalidad nos permite empaquetar en forma binaria un conjunto de información teniendo control incluso a nivel de bit cada uno de sus componentes.

Por ejemplo la forma en la que tomábamos la cabeza de la lista en una variable y el resto lo dejábamos en otra variable se puede simular así:

```
iex> <<cabeza::binary-size(1), cola::binary>> = "Hola"
"Hola"
iex> cabeza
"H"
iex> cola
"ola"
```

Para obtener el tamaño de la lista binaria empleamos la función `byte_size/1`. En el caso anterior para cada una de las variables empleadas:

```
iex> byte_size(cabeza)
1
iex> byte_size(cola)
3
```

Esta sintaxis es un poco más elaborada que la de las listas, pero se debe a que nos adentramos en la verdadera potencia que tienen las listas binarias: el manejo de bits.

4.10. Trabajando con Bits

En la sección anterior vimos la sintaxis básica para simular el comportamiento de la cadena al tomar la cabeza de una pila. Esta sintaxis se basa en el siguiente formato: ***var::tipo-tamaño(n)***.

En caso de que el tamaño no se indique, se asume que es tanto como el tipo soporte y/o hasta concordar el valor al que debe de igualarse (si es posible), por ello en el ejemplo anterior la variable ***cola*** se queda con el resto de la lista binaria.

Como tipo podemos indicar varios elementos. Los tipos tienen una forma compleja pero siguen una sintaxis: ***endian-signo-tipo-unidad***; vamos a ver los posibles valores para cada uno de ellos:

endian

La forma en que los bits son leídos en la máquina. Si es en formato Intel o Motorola, es decir, ***little*** o ***big*** respectivamente. Además de

estos dos, es posible elegir ***native***, que empleará el formato nativo de la máquina en la que se esté ejecutando el código.

```
iex> <<1215261793::big-size(32)>>
"Hola"
iex> <<1215261793::little-size(32)>>
"aloH"
iex> <<1215261793::native-size(32)>>
"aloH"
```

En este ejemplo se ve que la máquina de la prueba es de tipo ***little*** u ordenación Intel.

signo

Se indica si el número se almacenará en formato con signo o sin él, es decir, ***signed*** o ***unsigned***, respectivamente.

tipo

Es el tipo con el que se almacena el dato en memoria. Según el tipo el tamaño es relevante para indicar precisión o número de bits. Los tipos disponibles son: ***integer***, ***float***, ***binary***, ***bits*** (alias para bitstring), ***bitstring***, ***bytes*** (alias para binary), ***utf8***, ***utf16*** y ***utf32***. El tipo por defecto es ***integer***.

unidad

Este es el valor de la unidad por el que multiplicará el tamaño. En caso de enteros y coma flotante el valor por defecto es 1, y en caso de binario es 8. Por lo tanto: ***Tamaño x Unidad = Número de bits***; por ejemplo, si la unidad es 8 y el tamaño es 2, los bits que ocupa el elemento son 16 bits.

Si queremos almacenar tres datos de color rojo, verde y azul en 16 bits, tomando para cada uno de ellos 5, 5 y 6 bits respectivamente, tenemos la partición de los bits algo dificultosa. Con este manejo de bits, componer la cadena de 16 bits (2 bytes) correspondiente a los valores 20, 0 y 6, sería así:

```
iex> <<20::size(5), 0::size(5), 60::size(6)>>
<<160, 60>>
```

Nota

Para obtener el tamaño de la lista binaria en bits podemos emplear la función `Kernel.bit_size/1` que nos retornará el tamaño de la lista binaria:

```
iex> bit_size("Hola mundo!")
88
```

En el módulo ***Bitwise*** podemos encontrar no solo funciones sino también operadores para trabajar con bits. Al agregar el módulo para su uso automáticamente se definen en el ámbito actual todos los operadores y funciones. Podemos ver algunos ejemplos de Y (`Bitwise.band/2` o el operador ***&&&***), O inclusivo (`Bitwise.bor/2` o el operador ***|||***), O exclusivo (`Bitwise.bxor/2` o el operador **^^^**) y NO (`Bitwise.bnot/1` o el operador **~~~**):

```
iex> use Bitwise
Bitwise
iex> band 0b101, 0b110
4
iex> 0b101 &&& 0b110
4
iex> bor 0b101, 0b010
7
iex> 0b101 ||| 0b010
7
iex> bnot 0b111
-8
iex> ~~~0b111
-8
iex> bxor 0b101, 0b111
2
iex> 0b101 ^^^ 0b111
2
```

También podemos realizar el desplazamiento a la derecha o izquierda de los bits de un número con las funciones `Bitwise.bsl/2` (mueve el número de bits indicado como segundo parámetro a la izquierda) y `Bitwise.bsr/2` (mueve el número de bits indicado como segundo parámetro a la derecha). Igualmente en lugar de las funciones podemos emplear los operadores **<<<** y **>>>** respectivamente. Podemos ver un ejemplo:

```
iex> bsl 0b011, 1
6
iex> 0b011 <<< 1
6
iex> bsr 0b110, 1
3
iex> 0b110 >>> 1
3
```

4.11. Tuplas

Las tuplas son tipos de datos organizativos en Elixir. Se pueden crear listas de tuplas para conformar conjuntos de datos homogéneos de elementos individuales heterogéneos.

Las tuplas, a diferencia de las listas, no pueden incrementar ni decrementar su tamaño salvo por la redefinición completa de su estructura. Se emplean para agrupar datos con un propósito específico. Por ejemplo, imagina que tenemos un directorio con unos cuantos ficheros. Queremos almacenar esta información para poder tratarla y sabemos que va a ser: ruta, nombre, tamaño y fecha de creación.

Esta información se podría almacenar en forma de tupla de la siguiente forma:

```
{ "/home/user", "texto.txt", 120, {{2011, 11, 20}, {0, 0,
 0}} }
```

Las llaves indican el inicio y fin de la definición de la tupla, y los elementos separados por comas conforman su contenido. También hemos escrito la fecha y hora a través de tuplas.

4.11.1. Modificación dinámica de tuplas

Hay momentos en los que necesitamos agregar un valor más a una tupla, eliminar un valor o tomar el valor que está en una posición sin necesidad de realizar una concordancia. Los módulos ***Kernel*** y ***Tuple*** nos ayudan a realizar estas operaciones. Veamos cómo:

Kernel.put_elem/3

Cambia el elemento de una tupla sin modificar el resto de los elementos:

```
iex> put_elem {:a, :b, :c}, 1, B
{:a, B, :c}
```

Tuple.append/2

Agrega un elemento al final de la tupla:

```
iex> Tuple.append {:a, :b, :c}, :d
{:a, :b, :c, :d}
```

Kernel.elem/2

Obtiene un elemento de la tupla dado su índice:

```
iex> elem {:a, :b, :c}, 1
:b
```

Tuple.delete_at/2

Elimina un elemento de una tupla:

```
iex> Tuple.delete_at {:a, :b, :c}, 1
{:a, :c}
```

Importante

No obstante la modificación de tuplas no es rápida en comparación con las listas. Podemos emplear estos métodos de modificación de tuplas en actualizaciones en caliente de datos para convertir de una versión a otra los estados internos pero no es recomendable realizar este tipo de acciones en el funcionamiento normal de la aplicación.

4.11.2. Listas de Propiedades

Aunque las listas de propiedades puedan verse en muchas ocasiones como un tipo de dato completamente diferente internamente se componen de una lista de tuplas de dos elementos. Es muy útil para la especificación de opciones y configuración.

Podemos emplear esta sintaxis para especificar una lista de propiedades:

```
iex> opts = [enabled: true, updated_at: {2018, 5, 20}]
[enabled: true, updated_at: {2018, 5, 20}]
```

La única restricción es que las claves sean átomos. Los valores pueden contener cualquier valor.

Al tratarse de una lista podemos emplear las funciones de lista para tratar la lista de propiedades. No obstante la lista de propiedades tiene agregado el acceso a sus elementos de forma simplificada a través de *Access*. Veremos en detalle este módulo más adelante en el Capítulo 5, *Meta-Programación*.

Un poco de azúcar...

La sintaxis de la lista de propiedades parte de la forma original de lista conteniendo tuplas de dos elementos donde el primer elemento es siempre un átomo. Lo podemos ver como:

```
iex> [{:enabled, true}]
[enabled: true]
```

El cambio para esta sintaxis se componen del movimiento de los dos puntos (:) al final del átomo y la eliminación de la coma (,) y las llaves ({}). Facilita y simplifica la escritura.

Por ejemplo podemos acceder a los elementos de la lista anterior de la siguiente forma:

```
iex> opts[:enabled]
true
iex> opts[:updated_at]
{2018, 5, 20}
iex> opts[:created_at]
nil
```

Al acceder a cada elemento obtenemos su valor y cuando el elemento no existe obtenemos en respuesta el valor nulo (***nil***).

4.12. Mapas

Los mapas son una estructura de datos donde cada elemento se almacena bajo una clave. La clave puede ser de cualquier tipo igual que su contenido. Podemos definir un mapa de la siguiente forma:

```
iex> m = %{ :nombre => "Manuel" }
%{nombre: "Manuel"}
```

Podemos además cambiar la información contenida en el mapa así:

```
iex> m = %{m | :nombre => "Miguel" }
#{nombre: "Miguel"}
```

Esta sintaxis solo nos servirá para modificar el contenido de una clave existente dentro del mapa. Si necesitamos agregar un nuevo elemento al mapa tenemos dos funciones bajo el módulo ***Map***. La función `Map.put_new/3` agrega el elemento solo si no existe previamente en la lista. La función `Map.put/3` agrega o actualiza una clave para darle un nuevo valor.

Veamos unos ejemplos de uso de estas funciones:

```
iex> m = %{ :nombre => "Manuel" }
%{nombre: "Manuel"}
iex> m = Map.put(m, :apellido, "Rubio")
%{apellido: "Rubio", nombre: "Manuel"}
iex> m = Map.put_new(m, :nombre, "Miguel")
%{apellido: "Rubio", nombre: "Manuel"}
iex> m = Map.put_new(m, :edad, 38)
%{apellido: "Rubio", edad: 38, nombre: "Manuel"}
```

Para extraer un valor podemos proceder de varias formas. Podemos usar concordancia como veremos más adelante en la Sección 2.1, "Concordancia" del Capítulo 3, ***Expresiones, Estructuras y Errores***, también podemos usar los corchetes ([]) para acceder al dato o usando un punto (.) a través del nombre (siempre que sea un átomo).

La sintaxis usando ***Access*** (o los corchetes) nos permite acceder a una clave incluso cuando su clave no es un átomo:

```
iex> m[12]
nil
iex> m[:nombre]
"Manuel"
```

En caso de no existir la clave (como el primer caso) el retorno es ***nil***. Probamos ahora la otra sintaxis:

```
iex> m.nombre
"Manuel"
iex> m.direccion
** (KeyError) key :direccion not found in: %{apellido:
 "Rubio", edad: 38, nombre: "Manuel"}
```

Podemos acceder sin problemas a las claves existentes pero cuando indicamos una clave no existente obtenemos una excepción. Si en el código que estás desarrollando necesitas obtener un valor y este valor debe existir este es un buen método para fallar rápido y poder hacer una corrección temprana.

El módulo ***Map*** contiene las funciones que permiten eliminar una clave, buscar usando una función en lugar de la concordancia, obtener el número de claves y algunas otras opciones más. No obstante otras funciones como `Kernel.is_map/1` o `Kernel.map_size/1` pertenecen al módulo ***Kernel***.

Algunos ejemplos:

```
iex> Map.delete m, :nombre
#{apellido: "Rubio", edad: 38}
iex> Map.get m, :nombre
"Manuel"
iex> is_map m
true
```

```
iex> map_size m
3
iex> Maps.keys m
[:apellido, :edad, :nombre]
```

4.13. Cambios en Profundidad

Uno de los problemas de la inmutabilidad y la anidación de la información es cómo modificar un dato dentro de una lista, a su vez dentro de otra lista, y otra lista y así sucesivamente. No hay forma simple. Debemos extraer el dato a modificar y después modificar todos sus antecesores para introducir las modificaciones.

Los creadores de Elixir han tenido este problema en cuenta y han desarrollado un par de funciones para ayudar a modificar información anidada. Se trata de las funciones `Kernel.get_in/2,3` y `Kernel.put_in/3,4`.

Estas funciones nos permiten especificar una ruta como parámetro y el valor final a modificar. La ruta puede especificarse de dos formas:

```
iex> data = %{
...>   "alpha" => [
...>     primero: %{"one" => 1, "two" => 3},
...>     segundo: %{"yksi" => 1, "kaksi" => 2}
...>   ],
...>   "beta" => 2
...> }
%{
  "alpha" => [
    primero: %{"one" => 1, "two" => 3},
    segundo: %{"kaksi" => 2, "yksi" => 1}
  ],
  "beta" => 2
}
iex> put_in(data["alpha"][:primero]["two"], 2)
%{
  "alpha" => [
    primero: %{"one" => 1, "two" => 2},
    segundo: %{"kaksi" => 2, "yksi" => 1}
  ],
  "beta" => 2
}
iex> put_in(data, ["alpha", :primero, "two"], 2)
%{
  "alpha" => [
    primero: %{"one" => 1, "two" => 2},
    segundo: %{"kaksi" => 2, "yksi" => 1}
  ],
  "beta" => 2
}
```

En la primera especificamos la forma de acceder al elemento a modificar y en la segunda empleamos una lista con cada una de las claves. En el primer caso optamos por la forma provista por ***Access*** empleando

los corchetes ([]) y en el segundo caso el sistema emplea de forma automática también ***Access*** con las claves dadas.

4.14. Conjuntos

Otro tipo de dato que podemos emplear en Elixir son los conjuntos. Estos datos no tienen una sintaxis específica para poder crearse. Debemos emplear siempre el módulo ***MapSet*** para manejarlos.

Veamos un ejemplo con código de cómo trabajar con conjuntos:

```
iex> set = MapSet.new ❶
#MapSet<[]>
iex> set = MapSet.put set, "rojo" ❷
#MapSet<["rojo"]>
iex> set = MapSet.put set, "azul"
#MapSet<["azul", "rojo"]>
iex> set = MapSet.put set, "rojo" ❸
#MapSet<["azul", "rojo"]>
iex> MapSet.member? set, "verde" ❹
false
iex> set = MapSet.delete set, "rojo" ❺
#MapSet<["azul"]>
```

❶ Creamos un mapa vacío.

❷ Agregamos un elemento dentro del conjunto.

❸ Al intentar agregar un elemento duplicado se retorna el conjunto sin modificaciones.

❹ Podemos comprobar si un elemento está dentro del conjunto a través la función `MapSet.member?/2`.

❺ Eliminamos un elemento del conjunto.

Podemos realizar operaciones de más alto nivel para conjuntos como la unión a través de `MapSet.union/2`, la intersección `MapSet.intersection/2`, la resta de conjuntos `MapSet.difference/2` y otras funciones para determinar si dos conjuntos son iguales (`MapSet.equal?/2`), si uno es subconjunto de otro (`MapSet.subset?/2`) e incluso si son disjuntos (que no comparten ningún elemento `MapSet.disjoint?/2`).

4.15. Rangos

Una de las ventajas de Elixir es la evaluación perezosa. Los rangos emplean la evaluación perezosa para definir un conjunto de números

enteros y permitir a las funciones su iteración generando cada número cuando es necesario. La creación del rango lo podemos realizar a través de la función `Range.new/2` especificando el número inicial y final para el rango o a través del operador **..** tal y como podemos ver en estos ejemplos:

```
iex> 1..10
1..10
iex> Range.new 20, 100
20..100
```

Los rangos pueden emplearse como origen de datos en la mayoría de funciones del módulo ***Enum***.

4.16. Sigilos

Los sigilos[9] son representaciones especiales para dar una mayor semántica a un dato específico. Los sigilos pueden ser definidos por el programador y existen algunos predefinidos en Elixir.

La sintaxis de los sigilos comienza con la virgulilla (~)[10] seguida de una letra y uno de los 8 delimitadores válidos para contener la información. Los delimitadores válidos son: /, |, ", ', (), [], {}, <>. Los 4 últimos tienen un símbolo diferente para apertura y cierre mientras que los 4 primeros usan el mismo símbolo tanto para abrir como para cerrar.

Para muchos de los sigilos la ventaja es la posibilidad de escribir un texto dentro sin necesidad de tener que realizar el escapado de cada uno de los caracteres que actúan normalmente delimitadores:

```
iex> ~c[<input type='text' name='nombre'/>]
'<input type=\'text\' name=\'nombre\'/>'
```

Los sigilos definidos en Elixir son:

~c, ~C

Lista de caracteres. Es equivalente a encerrar un texto entre comillas simples ('):

```
iex> ~c(Esto es un texto)
'Esto es un texto'
```

[9] Del latín ***sello***, un símbolo usado en magia: https://es.wikipedia.org/wiki/Sigilo_(magia).
[10] En teclados españoles este símbolo no suele aparecer. Según el sistema operativo puede obtenerse presionando **AltGr** + **4** o la combinación de teclas **Option** + **Ñ** (en Mac además habrá que presionar espacio).

~D

Fecha. Gestionado desde el módulo ***Date***. Veremos más adelante este dato.

~N

Fecha "ingenua"[11]. Gestionado desde el módulo ***NaiveDateTime***. Veremos más adelante este dato.

~r, ~R

Expresiones regulares. Este sigilo nos permite construir una expresión regular y emplearla en comparaciones de forma sencilla. Veremos más adelante las expresiones regulares.

~s, ~S

Cadena de texto. Es equivalente a encerrar un texto entre comillas dobles ("):

```
iex> ~s[Esto es un texto]
"Esto es un texto"
```

~T

Hora. Gesionado desde el módulo ***Time***. Veremos más adelante este dato.

~w, ~W

Lista de palabras. Permite generar una lista de palabras. Cada palabra será una cadena de caracteres:

```
iex> ~w/rojo azul verde/
["rojo", "azul", "verde"]
```

Hemos visto que hay posibilidad en algunos sigilos de emplear letras mayúsculas o minúsculas. Para estos sigilos (***s***, ***c***, ***r*** y ***w***) es una forma de diferenciar la posibilidad de agregar caracteres de escape e interpolación (en caso de las minúsculas) o no permitir ninguno de estos (las letras en mayúscula).

Podemos ver un ejemplo de esto mismo a continuación:

```
iex> ~w{rojo\nverde azul\namarillo gris\nnegro}
["rojo", "verde", "azul", "amarillo", "gris", "negro"]
iex> ~W{rojo\nverde azul\namarillo gris\nnegro}
["rojo\\nverde", "azul\\namarillo", "gris\\nnegro"]
```

[11] Es llamada "ingenua" por no contener información de la zona horaria.

Los caracteres de escape más habituales son los siguientes:

Caracter	**Descripción**
\\	Barra invertida
\a	Sonido de alerta (beep).
\b	Retroceso. Retrocede un caracter.
\d	Suprimir. Simula la pulsación de la tecla suprimir.
\e	Escape. Simula la pulsación de la tecla escape.
\f	Salto de página. No suele tener efecto.
\n	Salto de línea. Mueve el cursor a la siguiente línea.
\r	Retorno de carro. Mueve el cursor al inicio de la línea actual.
\s	Espacio. Escribe un espacio.
\t	Tabulador. Escribe un conjunto de espacios (normalmente 8).
\v	Tabulador vertical. No suele tener efecto.
\0	Caracter nulo.
\xDD	Imprime un caracter dado su índice en hexadecimal.
\uDDDD o \u{DD...}	Imprime un caracter dado su índice en la tabla Unicode.

Veremos en la Sección 12, "Sigilos" del Capítulo 4, ***Las funciones y los módulos*** más sobre cómo definir nuestros propios sigilos.

5. Conversión de datos

A lo largo del capítulo hemos visto los tipos de datos de que disponemos. Hemos visto también algunas conversiones entre cadenas de texto y números y estos dos en átomos y viceversa.

No obstante no todos los datos pueden convertirse en otros tipos. Una cadena de texto cualquiera no puede convertirse en una lista de propiedades o un mapa ni viceversa.

Para convertir cadenas conteniendo números a enteros y reales hemos visto las funciones `Integer.parse/1` y `Float.parse/1`. De la misma forma hemos visto funciones como `Integer.to_string/1` o `Float.to_string/1` para la conversión de números a cadena.

Ahora veremos cómo podemos convertir un mapa en una lista de propiedades y una lista de propiedades en un mapa.

Para convertir a lista de propiedades un mapa la forma más fácil es emplear la función `Map.to_list/1`. Esta función exporta el contenido del mapa en formato a una lista. Si todas las claves del mapa son átomos entonces veremos el formato saliente como una lista de propiedades. Si el tipo de una o más claves fuese diferente entonces veríamos únicamente una lista con tuplas de dos elementos:

```
iex> l = Map.to_list(m)
[apellido: "Rubio", edad: 38, nombre: "Manuel"]
iex> Map.to_list(%{ "inicia" => 10 })
[{"inicia", 10}]
```

En el primer caso todas las claves son átomos y obtenemos una lista de propiedades. En el segundo caso sin embargo la única clave existente es una cadena de caracteres y obtenemos por tanto una lista simple.

Para convertir de nuevo las listas en mapas debemos usar una nueva función del módulo ***Enum***. Esta función es `Enum.into/2`. Como primer parámetro pasamos el dato a convertir y como segundo dato el tipo de dato al que queremos convertirlo.

Los tipos de datos aceptados y vistos de momento son listas y mapas. Veamos cómo funciona para convertir la lista a mapa:

```
iex> Enum.into l, %{}
%{apellido: "Rubio", edad: 38, nombre: "Manuel"}
```

En caso de querer convertir un mapa en lista a través de esta función podemos pasar como segundo parámetro unos corchetes ([]).

6. Imprimiendo por pantalla

La impresión por pantalla nos permite mostrar texto y mensajes en la salida estándar (también conocida en sistemas tipo Unix como ***stdio***[12]). Podemos hacer una impresión por pantalla simple usando:

```
iex> IO.puts "Hola mundo!"
"Hola mundo!"
```

[12] Acortación de ***Standard Input/Output*** o Entrada/Salida Estándar.

```
:ok
```

La función debe retornar siempre ***:ok***. Podemos hacer salida por pantalla de textos, números enteros y reales y átomos. Otras estructuras de datos más complejas deben ser convertidas para poder ser impresas. No obstante disponemos de la función `Kernel.inspect/1`. Esta función presenta como cadena de texto cualquier estructura de datos compleja. Su formato es el mismo que podemos ver por consola:

```
iex> IO.puts "map => #{inspect %{}}"
map => %{}
:ok
```

Como puedes ver podemos emplear la función `Kernel.inspect/1` dentro de la interpolación. De esta forma la salida queda más compacta. También podemos emplear la función `IO.inspect/1` para realizar la salida directa de un dato por pantalla:

```
iex> IO.inspect %{}
%{}
%{}
```

El retorno es el valor pasado como parámetro y por ello podemos ver en la salida primero el mapa impreso y después el valor retornado.

Dentro del módulo ***IO.ANSI*** podemos encontrar muchos comandos para facilitar la visualización de la salida. La función `IO.puts/1` nos permite enviar una lista con cadenas de texto en su interior. Si queremos escribir un mensaje de error resaltando parte del texto podemos hacer:

```
iex> IO.puts [IO.ANSI.red(),
                                      IO.ANSI.blink_slow(),
                                      "¡ERROR!",
                                      IO.ANSI.reset(),
                                      " no pudimos leer el
 fichero correctamente"]
```

La función ***IO.ANSI.reset/0*** nos permite volver al modo de texto inicial después de haber impreso en rojo y con parpadeos el texto ***¡ERROR!***.

Puedes ver más opciones de este módulo en la documentación oficial[13].

[13] https://hexdocs.pm/elixir/IO.ANSI.html

Capítulo 3. Expresiones, Estructuras y Errores

Mi objetivo es simplificar la complejidad. Solo deseo construir cosas que realmente simplifiquen la base de nuestra interacción humana.
—Jack Dorsey

En este capítulo ampliamos lo visto en el capítulo anterior con el conocimiento de las expresiones lógicas, las expresiones aritméticas, las estructuras de control y el manejo de errores.

1. Expresiones

Las expresiones se componen de operadores y datos para obtener una sentencia válida para el lenguaje con significado para el compilador. De esta forma ofrece en tiempo de ejecución una representación a nivel de código máquina del resultado que se pretende obtener.

Las expresiones pueden ser de tipo aritmético o lógico. Las aritméticas buscan un valor a través de operaciones matemáticas simples o complejas. De un conjunto de datos dados con las operaciones indicadas y el orden representado por la expresión se obtiene un resultado. En las lógicas se busca una conclusión lógica (o binaria) a la conjunción de los predicados expuestos.

Las operaciones aritméticas actúan sobre los dos tipos de números existentes. Siempre en conversión hacia el más general. Cuando realizamos una suma, resta o multiplicación de dos números si uno de ellos es un número real el resultado será real. Sin embargo en el caso de la división es indiferente que ambos números sean reales o no, el resultado debe ser siempre un número real.

En Elixir por defecto las operaciones lógicas son perezosas. Esto significa que evaluando el primer operador y según su resultado si podemos entenderlo como verdadero y la operación es un O lógico no necesitamos evaluar el resto de la operación. Lo mismo ocurre si el resultado del primer operando es falso y la operación es un Y lógico.

1.1. Expresiones Aritméticas

Con los números, de forma nativa, se pueden llevar a cabo expresiones aritméticas. Operaciones básicas como la suma, resta, multiplicación y división son de sobra conocidas. Otras operaciones como la división entera o el remanente (o módulo) se implementan en cada lenguaje de una forma distinta. Haremos un repaso rápido con un breve ejemplo:

```
iex> 2 + 2
4
iex> 2 - 2
0
iex> 2 * 3
6
iex> 10 / 3
3.3333333333333335
iex> div 10, 3
3
iex> rem 10, 3
1
```

Se puede hacer uso de los paréntesis para establecer una relación de precedencia de operadores. De este modo podemos anteponer una suma a una multiplicación. También podemos realizar operaciones encadenadas multiplicando más de dos operandos. Ejemplos de todo esto:

```
iex> 2 * 3 + 1
7
iex> 2 * (3 + 1)
8
iex> 3 * 3 * 3
27
```

1.2. Expresiones Lógicas

En la Sección 4.10, "Trabajando con Bits" del Capítulo 2, ***El lenguaje*** hablamos sobre el tratamiento de bits y las operaciones lógicas a nivel de bits. El álgebra de Boole. En este capítulo vamos a ver únicamente la comparación y generación de resultados comparativos para respuestas simples ***true*** o ***false***.

Para almacenar el resultado lógico de una comparación podemos hacer:

```
iex> c1 = 2 > 1
true
iex> c2 = 1 > 2
false
iex> c1 and c2
false
iex> c1 or c2
true
iex> c3 = 3 == (1 + 2)
true
iex> c1 and (c2 or c3)
true
```

Podemos construir todas las expresiones lógicas que queramos. A nivel de comparación obtenemos un resultado lógico (verdadero o falso). En la siguiente sección se mencionan todos los operadores de comparación que podemos emplear para realizar comparaciones.

También es posible emplear los símbolos ***&&*** para emplear en lugar de ***and*** y ***||*** en lugar de ***or***. Al igual que se emplean en otros lenguajes como Java o C.

En realidad entre estos símbolos y las palabras hay diferencias. Mientras que las palabras clave ***and*** y ***or*** necesitan que ambos operadores sean de tipo lógico los símbolos aceptan otros valores haciendo una conversión a valor lógico pero manteniendo el valor original. Por ejemplo:

```
iex> a || IO.puts("error!")
error!
:ok
iex> a = 100
100
iex> a || IO.puts("error!")
100
iex> a or IO.puts("error!")
** (BadBooleanError) expected a boolean on left-side of "or",
 got: 100
```

Por ello es común ver estos símbolos sirviendo por ejemplo para la asignación de valores por defecto. Si el valor a asignar a la variable es falso o nulo (nil) entonces se emplea el valor tras el operador. Veremos más adelante más usos de estos operadores.

1.3. Expresiones regulares

Las expresiones regulares permiten analizar un texto y extraer información o comprobar si cumple con un patrón escrito con un formato específico.

El formato de las expresiones regulares es bastante extenso y requeriría de un capítulo completo para comentar su funcionamiento. Las expresiones regulares se basan en PCRE[1].

Como vimos en la Sección 4.16, "Sigilos" del Capítulo 2, ***El lenguaje*** las expresiones regulares se construyen a través de un sigilo. Podemos representar una expresión regular para comprobar si una dirección de correo electrónico es correcta de la siguiente forma:

```
iex> email = ~r/^[-\w.%+]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,125}
[A-Z]{2,63}$/i
```

La expresión regular podemos emplearla en funciones que requieran de expresiones regulares o podemos emplearla en conjunción con el operador **=~**. Este operador se emplea también en otros lenguajes como Perl y Ruby. Lo empleamos para realizar la concordancia entre una expresión regular y un texto:

[1] https://www.pcre.org/

```
iex> "manuel@altenwald.com" =~ email
true
iex> "no-es-email" =~ email
false
```

A través del uso de otras funciones del módulo ***Regex*** podemos compilar las expresiones regulares para emplearlas de forma más rápida en nuestro código, emplear las expresiones para realizar reemplazos y capturar trozos entre otras opciones. Puedes revisar el módulo Regex[2] para más información.

1.4. Precedencia de Operadores

El orden de los operadores para Elixir de más prioritario a menos prioritario es el siguiente:

Operador	**Descripción**	**Asociación**
@	Prefijo para constantes.	Unitario.
.	Ejecución de funciones.	Izquierda a Derecha.
+ - ! ^ not ~~~	Signo y negaciones.	Unitario.
/ *	División y Multiplicación.	Izquierda a Derecha.
+ -	Suma y resta.	Izquierda a Derecha.
++ -- .. <>	Agrega/Sustrae de listas, rangos y concatenación.	Derecha a Izquierda.
in, not in	Comprueba si un elemento se encuentra en una colección o no.	Izquierda a Derecha.
|> <<< >>> ~>> <<~ ~> <~ <~> <|>	Operador ***pipe*** y otros personalizables.	Izquierda a Derecha.
< > <= >=	Comparaciones relativas (mayores o menores).	Izquierda a Derecha.
== != =~ === !==	Comparaciones de igualdad.	Izquierda a Derecha.
&& &&& and	Operador lógico Y.	Izquierda a Derecha.

[2] https://hexdocs.pm/elixir/Regex.html

Operador	Descripción	Asociación
|| ||| or	Operador lógico O.	Izquierda a Derecha.
&	Clausuras.	Unitario.
=	Asignación.	Derecha a Izquierda.
=>	Relación entre clave y valor para mapas.	Derecha a Izquierda.
|	Separador para mapas y estructuras.	Derecha a Izquierda.
::	Separador para especificación de tipos.	Derecha a Izquierda.
when	Especifica cuándo comienza el cuerpo de una guarda.	Derecha a Izquierda.
<- \\	Iterador para ***for*** y separador de valores opcionales en funciones.	Izquierda a Derecha.

En la lista de operadores aparecen muchos que realmente podrían no considerarse operadores ya que su semántica hace verlos como partes de código normalmente no relativas a las expresiones matemáticas o lógicas. Sin embargo en Elixir todo es una expresión y la comprobación de precedencia se calcula en todo momento para todo el código escrito.

Hay muchos operadores que son considerados operadores pero no tienen un código asociado. Si recuerdas la Sección 4.10, "Trabajando con Bits", la definición de los operadores de esa sección solo pueden emplearse si se importan del módulo correspondiente. Si no se importan Elixir los detecta como operadores pero sin funcionamiento específico y genera un error.

Algo similar sucede con los operadores personalizables. Se pueden emplear para crear código y emplear esos operadores pero por defecto no tienen código asignado ni asociado y fallarán si intentas utilizarlos. Veremos en la siguiente sección cómo agregar código a estos operadores y cambiar código de otros.

Los operadores de comparación pueden emplearse con elementos de diferentes tipos. La conversión de esos tipos no se realiza, en su lugar se comparan los tipos de la siguiente forma:

```
número < átomo < referencia < clausura < puerto < pid <
tupla < mapa < lista < bitstring
```

Esto quiere decir que sin comparamos un número con un átomo el átomo siempre será mayor que el número.

1.5. Sobrecarga de operadores

La sobrecarga de operadores se realiza teniendo en cuenta los operadores que podemos sobrecargar y el tipo que tienen. Dependerá si son operadores binarios o unitarios. En la tabla de la sección anterior indicamos los operadores y su asociación. Los operadores con asociación izquierda a derecha o derecha a izquierda son binarios. El resto son unitarios.

Un operador binario se sobrecarga indicando la definición de la función de la siguiente forma:

```
defmodule BuscaTexto do
  def trozo ~> texto do
    String.contains? texto, trozo
  end
end
```

Para que los operadores funcionen hay que importar los módulos donde están definidos. Podemos hacerlo de la siguiente forma:

```
iex> import BuscaTexto
BuscaTexto
iex> "a" ~> "abcdef"
true
iex> "ab" ~> "abcdef"
true
iex> "fg" ~> "abcdef"
false
```

Como puedes observar conseguimos obtener la funcionalidad deseada en el operador definido. Si queremos sobrecargar el operador de suma por ejemplo para realizar la concatenación en caso de encontrarse dos binarios, podemos hacer lo siguiente:

```
defmodule Concatena do
  def a + b when is_binary(a) and is_binary(b) do
    a <> b
  end
  def a + b, do: Kernel.'+' a, b
end
```

En este caso hemos comprobado los valores de entrada. En caso de encontrarnos dos binarios realizamos la concatenación. Sin embargo si encontramos otros tipos de datos entonces dejamos a la operación nativa trabajar.

La prueba sin embargo nos arroja un error:

```
iex> import Concatena
Concatena
iex> "hola" + " mundo!"
** (CompileError) iex:11: function +/2 imported from both
 Concatena and Kernel, call is ambiguous
    (elixir) src/
elixir_dispatch.erl:111: :elixir_dispatch.expand_import/6
    (elixir) src/
elixir_dispatch.erl:81: :elixir_dispatch.dispatch_import/5
```

Esto sucede porque ambos operadores están importados. Kernel importa por defecto todo su contenido siempre. Podemos cambiar esto haciendo lo siguiente:

```
iex> import Kernel, except: ['+': 2]
Kernel
iex> "hola" + " mundo!"
"hola mundo!"
iex> 2 + 2
4
```

De esta forma indicamos que no queremos importar de Kernel el operador de la suma y al haber importad anteriormente nuestro operador ya podemos emplear sin problemas nuestra versión.

Importante

Ten cuidado con la sobrecarga de operadores. Si cambias el funcionamiento de los operadores por defecto podrías obtener errores en tu código sin saber a qué se pueden deber exactamente.

2. Estructuras de Control

Erlang es un lenguaje funcional y como veremos en esta sección dispone de concordancia. Igualmente la mayoría de estructuras que utilizaremos no serán las típicas de otros lenguajes imperativos.

Muchas de estas estructuras se basan en la concordancia de sus expresiones. Ambas tienen que realizar una concordancia positiva con una expresión y ejecutar un código que retorne un valor.

Como la concordancia es tan importante para estas estructuras y en Elixir en general vamos a dedicar una sección para estudiar la ***concordancia*** y seguidamente veremos las estructuras donde se aplica.

2.1. Concordancia

La concordancia es una herramienta muy útil. Fácil de leer y ahorra escribir mucho código. Ya vimos un poco de esta concordancia en

la asignación con cadenas de caracteres, listas binarias y listas de caracteres, pero podemos aplicarla sobre la mayoría de tipos de datos.

Por ejemplo para tuplas. Si necesitamos extraer el segundo valor de una tupla de dos elementos. Podemos hacer una asignación imitando la estructura interna almacenada en la variable y colocando la variable a enlazar con el valor deseado justo en la posición donde se encuentre el valor deseado:

```
iex> a = {:clave, "valor"}
{:clave, "valor"}
iex> {_, valor} = a
{:clave, "valor"}
iex> valor
"valor"
```

De esta forma la asignación toma un papel más importante en el desarrollo de código para Elixir. Podemos emplear la concordancia para extraer elementos de los mapas de esta forma:

```
iex> d = %{apellido: "Rubio", edad: 38, nombre: "Manuel"}
%{apellido: "Rubio", edad: 38, nombre: "Manuel"}
iex> %{edad: edad} = d
%{apellido: "Rubio", edad: 38, nombre: "Manuel"}
iex> edad
38
```

Igualmente para las listas podemos extraer el primer elemento o los primeros elementos de la siguiente forma:

```
iex> [a, b|_] = [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
iex> a
1
iex> b
2
iex> [_, _|c] = [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
iex> c
[3, 4, 5]
```

En el ejemplo también podemos ver que hemos empleado el símbolo de tubería (|) y el subrayado (_). El primer símbolo nos permite separar los primeros elementos del resto. Vimos esto ya en la Sección 4.6, "Listas" del Capítulo 2, ***El lenguaje***. Tras la tubería siempre habrá una lista con el resto de elementos y si no hay más elementos la lista estará vacía.

El símbolo de subrayado lo empleamos cuando queremos ignorar el dato. En la sintaxis podemos ver en el primer ejemplo cómo ignoramos el resto de la lista porque solo nos interesan los dos primeros elementos. En el segundo ejemplo no nos interesan los dos primeros elementos y sí el resto de la cadena.

También podemos emplear la concordancia en las tuplas:

```
iex> d = {"Manuel", "Rubio", 38}
{"Manuel", "Rubio", 38}
iex> {_nombre, _apellido, edad} = d
{"Manuel", "Rubio", 38}
iex> edad
38
```

El símbolo de subrayado puede emplearse como prefijo para una variable que no se vaya a emplear. Esto es útil cuando queremos tener información sobre el contenido de la tupla incluso cuando el dato no vaya a ser empleado.

Importante

El uso del subrayado como prefijo no debe emplearse como normal general para usarlo con variables que sí se empleen en el código. Esta sintaxis específica proporciona información sobre un dato no necesario para saber a qué pertenece por si en un futuro pudiera ser útil.

Cuando queremos hacer concordancia con el contenido de una variable y que no se enlace al nuevo valor concordante podemos usar el arco circunflejo (^) antes de la variable. Este operador es conocido también como operador pin:

```
iex> d = {"Manuel", "Rubio", 38}
{"Manuel", "Rubio", 38}
iex> nombre = "Manuel"
"Manuel"
iex> {^nombre, apellido, _edad} = d
{"Manuel", "Rubio", 38}
iex> {^nombre, apellido, _edad} = {"Miguel", "Rubio", 40}
** (MatchError) no match of right hand side value: {"Miguel",
 "Rubio", 40}
```

Como vemos, si el contenido de la variable no concuerda se produce un error de concordancia (***MatchError***).

2.2. Operador *pipe*

El operador ***pipe***[3] se emplea para simplificar y hacer más legibles construcciones como la siguiente:

```
iex>
 Enum.sort(Enum.uniq(String.graphemes(Integer.to_string(424))))
["2", "4"]
```

[3]En inglés ***pipe*** significa tubería. El operador hace mención a una tubería porque los datos van recorriendo un flujo.

Si te fijas, cada retorno es el primer parámetro de la siguiente función encadenada. Podríamos escribirlo también así:

```
iex> r = Integer.to_string(424)
"424"
iex> r = String.graphemes(r)
["4", "2", "4"]
iex> r = Enum.uniq(r)
["4", "2"]
iex> r = Enum.sort(r)
["2", "4"]
```

Pero Elixir tiene como principal objetivo evitar la repetición y acciones tan repetitivas. Por eso crearon este operador ***|>***. El operador es una simplificación a tomar el retorno de la función anterior y emplearla como primer parámetro de la siguiente. Los comandos anteriores quedarían así en consola:

```
iex> 424 |>
     Integer.to_string() |>
     String.graphemes() |>
     Enum.uniq() |>
     Enum.sort()
["2", "4"]
```

Realmente en un fichero de código lo escribiríamos de esta forma:

```
defmodule Numeros do
  def numeros(int) when is_integer(int) do
    int
    |> Integer.to_string()
    |> String.graphemes()
    |> Enum.uniq()
    |> Enum.sort()
  end
end
```

Visualmente se ve cómo se va conectando cada línea a la siguiente. Este operador es muy empleado en los códigos de Elixir para el tratamiento de listas y colecciones de datos principalmente pero también para modificar datos de mapas o estructuras.

Iremos empleando este operador en nuestros códigos de ejemplo a lo largo del libro.

2.3. Estructura *if...else* y *unless..else*

Esta es sin duda la construcción más simple de Elixir. La estructura condicional. Se basa en evaluar una expresión y si su resultado es verdadero (***true***) ejecutar el bloque de código bajo ***do***. Si la expresión fuese falsa y se ha definido un bloque de código bajo ***else*** entonces ejecutaría este bloque en su lugar.

Podemos ver algunos ejemplos:

```
if (2 == 2), do: "OK"
```

Como dijimos al principio, en Elixir todo es una expresión y esta estructura no es una excepción. El bloque de código solo contiene la cadena de caracteres ***"OK"*** y es lo que retorna. Podemos agregar también un bloque para el caso en el que no se cumpla la expresión:

```
if (2 == 3), do: "NO", else: "OK"
```

Volvemos a obtener de nuevo la cadena ***"OK"***. Podemos escribir esta estructura como un bloque:

```
if (2 == 3) do
  "NO"
else
  "OK"
end
```

Como puedes ver hay dos bloques. La palabra clave ***else*** se encarga de separar ambos bloques y por lo tanto solo necesitamos un ***end*** al final.

Si queremos evaluar negativamente una expresión y queremos ahorrarnos la negación e incluso utilizar una estructura condicional más semántica podemos emplear ***unless*** en lugar de ***if***.

Como ejemplo, la estructura anterior podría escribirse como:

```
unless (2 == 3), do: "OK"
```

De la misma forma ***unless*** tiene opcionalmente el bloque ***else***.

2.4. Estructura *case*

La estructura ***case*** es una de las más útiles en Elixir. La razón de su potencia es la capacidad para emplear concordancia en sus valores. Nos permite evaluar una variable o el valor retornado por una función con un grupo de valores. Por ejemplo, imagina que tenemos un texto y queremos realizar la ejecución de un código dependiendo del prefijo encontrado:

```
case nombre do
  "Mr." <> nombre_ing -> {:hombre, nombre_ing}
  "Mrs." <> nombre_ing -> {:mujer, nombre_ing}
  "Ms." <> nombre_ing -> {:mujer, nombre_ing}
  "Sr." <> nombre_esp -> {:hombre, nombre_esp}
  "Sra." <> nombre_esp -> {:mujer, nombre_esp}
  "Srita." <> nombre_esp -> {:mujer, nombre_esp}
  _ -> {:desconocido, nombre}
```

```
end
```

Como podéis ver analizamos la cadena contra cada una de las entradas hasta que una concuerda con el contenido de la variable retornando el resultado. Podemos ver más formas de realizar concordancia en la Sección2.1, "Concordancia" .

Además de la concordancia podemos también agregar condiciones o guardas. Las guardas las veremos más en detalle en la Sección 8, "Guardas" del Capítulo4, ***Las funciones y los módulos***. Es decir, imagina que tenemos una estructura de usuarios y no solo queremos hacer concordancia sobre el usuario sino también indicar si un usuario es o no mayor de edad. Esto lo conseguimos agregando la palabra clave ***when*** tras la concordancia a realizar y entonces las condiciones a cumplir:

```
case usuario do
  %Usuario{edad: edad} when edad >= 18 ->
    :ok
  %Usuario{edad: edad} when edad >= 16 ->
    {:warn, "presenta ID"}
  _ ->
    {:error, "no puede comprar alcohol"}
end
```

Podemos probar de la siguiente forma:

```
iex> defmodule Usuario do
...>     defstruct [:nombre, :edad]
...> end
{:module, Usuario,
 <<70, 79, 82, 49, 0, 0, 5, 156, 66, 69, 65, 77, 65, 116,
   85, 56, 0, 0, 0, 184, 0, 0, 0, 18, 14, 69, 108, 105,
   120, 105, 114, 46, 85, 115, 117, 97, 114, 105,
   111, 8, 95, 95, 105, 110, 102, 111, 95, ...>>,
 %Usuario{edad: nil, nombre: nil}}
iex> usuario = %Usuario{nombre: "Manuel", edad: 38}
%Usuario{edad: 38, nombre: "Manuel", }
iex> case usuario do
...>   %Usuario{edad: edad} when edad >= 18 -> :ok
...>   _ -> {:error, "no puede comprar alcohol"}
...> end
:ok
iex> usuario = %Usuario{nombre: "Juan Antonio", edad: 11}
%Usuario{edad: 11, nombre: "Juan Antonio"}
iex> case usuario do
...>   %Usuario{edad: edad} when edad >= 18 -> :ok
...>   _ -> {:error, "no puede comprar alcohol"}
...> end
{:error, "no puede comprar alcohol"}
```

La potencia de ***case*** como hemos visto es la capacidad de enlazar condiciones y comprobaciones diferentes sin necesidad de anidar las condiciones unas en otras. Todas las comprobaciones quedan al mismo nivel de anidamiento y podemos revisarlas de un vistazo.

2.5. Estructura *cond*

Si nuestra construcción no requiere de concordancia y queremos ahorrarnos escribir una y otra vez el nombre de la variable podemos emplear la estructura ***cond*** en lugar de ***case*** o ***if***s anidados.

La sintaxis de ***cond*** es como sigue:

```
cond do
  condicion1 -> bloque1
  condicion2 -> bloque2
  condicionN -> bloqueN
end
```

La ventaja de esta estructura es la versatilidad que nos da al poder incluir cualquier condición. Podemos hacer, por ejemplo:

```
cond do
  usuario.genero == :hombre ->
    IO.puts("Sr. #{usuario.nombre}")
  usuario.edad >= 18 ->
    IO.puts("Sra. #{usuario.nombre}")
  true ->
    IO.puts("Srita. #{usuario.nombre}")
end
```

Las condiciones se evalúan de arriba a abajo. Si la primera condición (en este caso la comprobación de género masculino) es válida, se ejecuta solo el primer bloque y obtenemos el texto ***Sr. Manuel*** si no evaluase como correcta esa comprobación pasa a la siguiente. No hace falta que repitamos la comprobación de género, es binaria por lo que si no es ***:hombre*** debe ser ***:mujer***. La última comprobación nos deja claro que debe ser ***:mujer*** y menor de edad por lo que no hace falta que comprobemos nada.

Esta estructura al igual que ***case*** requiere de evaluar como verdadera al menos una opción. Por ello es común encontrar la última entrada como ***true***.

2.6. Estructura *with*

La estructura ***with*** permite construir una serie de comprobaciones o líneas a cumplir, una tras otra hasta concluir en el bloque de ejecución. Si no cumplimos con alguna de las condiciones el código entra en el bloque de errores y busca el error concreto.

La sintaxis de esta estructura es como sigue:

```
with
```

```
  concordancia1 <- funcion_o_variable1,
  concordancia2 <- funcion_o_variable2,
  concordanciaN <- funcion_o_variableN
do
  bloque_codigo
else
  concordancia_error1 -> bloque_error1
  concordancia_error2 -> bloque_error2
  concordancia_errorN -> bloque_errorN
end
```

Por ejemplo, imagina que tenemos dos funciones: ***valida_usuario/1*** y ***guarda_usuario/1***. Ambas funciones pueden retornar o el átomo ***:ok*** o una tupla ***{:error, reason}***. Podemos escribir el código de la siguiente forma:

```
with
  :ok <- valida_usuario(usuario),
  :ok <- guarda_usuario(usuario)
do
  IO.puts("usuario creado.")
else
  {:error, reason} ->
    IO.puts("error al crear: #{reason}")
end
```

Esta forma de apilar las funciones es muy parecida a la metodología de programación orientada a raíles[4]. Mantener el código base alineado y el código a emplear en caso de error aparte.

Para que esta forma funcione de forma correcta es muy posible que las funciones a emplear o las variables a evaluar deban ser tratadas en funciones hechas a medida para poder aprovechar al máximo esta forma de trabajar.

2.7. Estructura *for* o Listas de Comprensión

Las listas de comprensión son una forma rápida de iterar sobre cada elemento de una lista, filtrar para extraer únicamente los elementos deseados y realizar incluso una modificación antes de obtener la lista definitiva. Puede considerarse equivalente a la unión de las funciones `Enum.filter/2` y `Enum.map/2`.

Esta funcionalidad la obtenemos en Elixir a través de la estructura ***for***. La estructura ***for*** funciona como un iterador pero a diferencia de otros lenguajes la estructura retorna cada elemento procesado dentro de una lista.

La sintaxis es como sigue:

[4] https://altenwald.org/2018/07/17/programacion-orientada-a-ferrocarril/

```
iex> for i in 1..10, rem(i, 2) == 0 do: i
[2, 4, 6, 8, 10]
```

La construcción tiene dos partes. La primera es un grupo de generadores y condiciones separados por comas donde puedes especificar un generador como en el caso del ejemplo o una condición tal y como aparece también arriba. La segunda parte es el cuerpo de la lista de comprensión. El código que se ejecutará por cada elemento generado y validado. El código puede ser una única expresión tal y como vemos en el ejemplo o un bloque completo encerrado entre ***do..end***.

El resultado del ejemplo como vemos es cada elemento procesado que concuerda con la validación. En este caso obtenemos una lista con los números divisibles por dos (valores pares) para los números del 1 al 10.

Los generadores pueden ser de dos tipos. El visto en el ejemplo es un generador de lista. Genera una lista de elementos. El otro es un generador bitstring o simplemente de cadena de caracteres:

```
iex> for <<s::binary-size(1) <- "Hola!">> do
...>   String.upcase(s)
...> end
["H", "O", "L", "A"]
```

Como puedes ver se ha tomado un elemento cada vez de la cadena (o bitstring) y se ha procesado. También podemos agregar antes del bloque de código la configuración ***into*** para indicar el formato en el que queremos almacenar el resultado:

```
iex> for <<s::binary-size(1) <- "Hola!">>, into: "" do
...>   String.upcase(s)
...> end
"HOLA!"
```

El requisito para indicar el formato al que queremos exportarlo es que ese formato esté implementado en el protocolo ***Collectable***. Por defecto Elixir implementa los siguientes: BitString, File.Stream, HashDict, HashSet, IO.Stream, List, Map y MapSet.

3. Errores

Los errores son disparados desde dentro del flujo normal de ejecución para evitar que el resto del código sea ejecutado con datos inconsistentes.

Por ejemplo si una suma obtiene dos valores para ser sumados pero uno de ellos no es un número el resultado es impreciso e inconsistente. Un error es lanzado:

```
iex> "hola" + 10
** (ArithmeticError) bad argument in arithmetic expression
    :erlang.+("hola", 10)
```

En este caso el error es llamado ***ArithmeticError*** (o error aritmético). Hay muchos errores diferentes definidos por defecto en Elixir pero podemos definir algunos más para dar un mayor significado a los errores provocados por nuestro código.

Algunos de los errores más comunes son: ArgumentError, ArithmeticError, BadArityError, CaseClauseError, FunctionClauseError o SyntaxError.

Podemos definir nuestros propios errores de la misma forma que hacemos con las estructuras a través de la definición de un módulo:

```
defmodule MenorEdadError do
  defexception message: "menor de edad"
end
```

Podemos crear cualquier error como si fuese una estructura o a través de la función `Kernel.raise/1-2`:

```
iex> error = %RuntimeError{message: "no funciona"}
%RuntimeError{message: "no funciona"}
iex> raise error
** (RuntimeError) no funciona

iex> raise MenorEdadError
** (MenorEdadError) menor de edad
```

Podemos capturar estos errores si se producen dentro de un bloque ***try..rescue***. Capturamos de esta forma el error de la minoría de edad:

```
iex> try do
...>   raise MenorEdadError
...> rescue
...>   error in MenorEdadError ->
...>     IO.puts("error: #{error.message}")
...> end
error: menor de edad
:ok
```

De esta forma podemos rescatar el flujo de programa e intentar volver a un punto conocido. Si solo queremos realizar algún procesamiento y seguir lanzando el error hacia arriba podemos emplear en ese momento la función `Kernel.reraise/2-3`. Esta función se encarga de volver a enviar el error hacia un nivel superior pero manteniendo el volcado de pila intacto para en caso de tener que imprimir el error por pantalla podamos indicar exactamente dónde se produjo originalmente:

```
iex> try do
```

```
...>   raise MenorEdadError
...> rescue
...>   error in MenorEdadError ->
...>     IO.puts("error: #{error.message}")
...>     reraise(error, System.stacktrace)
...> end
error: menor de edad
** (MenorEdadError) menor de edad
```

La sección ***rescue*** es un bloque donde se ejecuta una concordancia entre el error recibido y el código a ejecutar en caso de concordar. Si no hay concordancia el error prosigue hasta encontrar otro ***try..rescue*** anterior o finalizar el flujo de ejecución.

4. Estructura *try..catch*

Además de provocar errores podemos lanzar otros valores. En Elixir es posible lanzar un valor dentro de un bloque ***try*** y capturarlo en la sección ***catch***. Podemos verlo a continuación:

```
iex(21)> try do
...>   for i <- 0..50 do
...>     <<prev :: binary-size(i),
...>       c :: binary-size(1),
...>       _ :: binary()>> = nombre
...>     if c == " ", do: throw(prev)
...>   end
...> catch
...>   primer_nombre ->
...>     IO.puts("primer: #{primer_nombre}")
...> end
primer: Manuel
:ok
```

Con este ejemplo vamos recorriendo cada vez un elemento de un binario. Siempre el mismo y de forma incremental hasta encontrar un espacio. En ese momento rompemos el flujo y lanzamos la parte anterior a ese espacio.

Este código puede ser útil para búsqueda de elementos de un flujo de datos y poder interrumpir el proceso en el momento que un valor ha sido encontrado.

5. Salida

En Elixir todo se ejecuta dentro de un proceso y aunque veremos los procesos en el Capítulo 7, ***Procesos y Nodos*** tenemos la función `Kernel.exit/1` que nos permite realizar la salida del proceso actual.

Esta función admite un parámetro. Este parámetro es la razón de la salida del proceso. Esta razón puede ser capturada por otros procesos o por una

construcción de tipo ***try..catch*** como la vista en la sección anterior. Tiene un formato diferente ya que emite dos valores: el átomo ***:exit*** y la razón empleada en la función.

Podemos ver un ejemplo:

```
iex> try do
...>   exit "adios"
...> catch
...>   :exit, razon -> IO.puts("salir: razon=#{razon}")
...> end
salir: razon=adios
:ok
```

Podemos ver que no se genera ningún error ni se indica la finalización por error del proceso. La señal de salida es capturada y se imprime el mensaje en su lugar.

Capítulo 4. Las funciones y los módulos

El diseño no es lo que ves, sino cómo funciona.
—Steve Jobs

Podemos decir sin miedo a equivocarnos que en Elixir las funciones son ciudadanos de primera clase. Es más, todo gira en torno a las funciones. Todo se expresa a través de funciones y todas las especificaciones de todas las estructuras condicionales, iterativas y de bucle se construyen sobre las funciones.

En este capítulo veremos cómo se organiza el código. Cómo podemos escribir las unidades mínimas y ascender en complejidad hasta llegar a los módulos. A lo largo de otros capítulos como el Capítulo 11, ***Aplicaciones y Supervisores*** o el Capítulo12, ***Ecosistema Elixir*** seguiremos viendo el ascenso de esta organización a aplicaciones y más tarde a proyectos.

1. Organización del código

El código se organiza en módulos. Los módulos deben tener como nombre un átomo. Para facilitar la nomenclatura se eligen nombres que comiencen por mayúscula como por ejemplo los módulos ***String*** o ***Map*** vistos anteriormente.

Los ficheros que contienen código en Elixir tienen la extensión ***ex***. Cualquier fichero con esta extensión es entendido como un código que debe ser compilado dentro del sistema. Hay otras extensiones pero las veremos más adelante. Los nombres de los ficheros se escriben en minúsculas y si el nombre lo forman más de una palabra la unión entre estas palabras se realiza a través de un subrayado (_). Por ejemplo el fichero que contiene el módulo ***MapSet*** tiene el nombre `map_set.ex`.

Los módulos se organizan en jerarquía. La jerarquía no está ligada a los nombres de los ficheros pero siempre ayuda mantener el mismo nombre de fichero y directorios acorde al nombre del módulo. Por ejemplo, si creamos el módulo ***Users*** esto nos dice que el fichero donde debe almacenarse el módulo debe ser `users.ex`.

Para especificar un nivel de jerarquía en Elixir empleamos el punto (.). Este símbolo nos permite separar un nombre general donde se agruparán varios módulos del nombre del módulo elegido. Por ejemplo si en nuestro código vamos a disponer de distintos módulos llamados ***Users*** podemos agruparlos según su función. Podemos encontrarnos

Account.Users dentro del fichero `account/users.ex` y ***Web.Users*** dentro del fichero `web/users.ex`.

2. Definición de Módulos

Los módulos en Elixir se definen a través de la construcción ***defmodule***. Esta construcción nos permite indicar el inicio de la especificación de un módulo: su nombre y contenido.

Podemos definir un módulo ***Vacio*** de la siguiente forma:

```
iex> defmodule Vacio do
...> end
{:module, Vacio,
 <<70, 79, 82, 49, 0, 0, 3, 76, 66, 69, 65, 77, 65, 116,
   85, 56, 0, 0, 0, 119, 0, 0, 0, 12, 12, 69, 108, 105,
   120, 105, 114, 46, 86, 97, 99, 105, 111, 8, 95, 95,
   105, 110, 102, 111, 95, 95, 3, ...>>, nil}
```

Al ejecutar en consola la definición se crea el módulo de forma inmediata. El retorno de la expresión es la información del módulo donde podemos ver una tupla de 4 elementos. El segundo elemento es el nombre del módulo y el tercer elemento es el código compilado del módulo.

Esta definición la podemos introducir dentro de un fichero y compilar el fichero con el comando de consola `IEx.Helpers.c/1`. Suponiendo que el fichero se llame `vacio.ex` y se encuentre en el directorio donde ejecutamos **iex**:

```
iex> c "vacio.ex"
[Vacio]
```

Nota

Recuerda que los asistentes (***helpers***) de la consola están disponibles a través del módulo ***IEx.Helpers*** pero no hace falta escribir ese módulo porque está importado en el contexto de ejecución de la consola.

Para ver más sobre la consola puedes revisar el Apéndice B, ***La línea de comandos***.

La compilación nos da una lista de nombres de módulos compilados. Esto nos indica que dentro del mismo fichero podemos agregar tantos módulos como necesitemos. No obstante no es una buena práctica realizar esto.

Podríamos agregar un segundo nivel a través de la especificación de un módulo dentro de otro. Si escribimos nuestro código en el fichero de esta forma:

```
defmodule Vacio do
  defmodule Subvacio do
  end
end
```

Podemos ver un módulo llamado ***Subvacio*** dentro del módulo ***Vacio***. Como dijimos anteriormente Elixir es jerárquico. Esto nos proporciona la posibilidad de agregar ***Subvacio*** dentro de la jerarquía del módulo superior. El retorno de la compilación será:

```
iex> c "vacio.ex"
[Vacio, Vacio.Subvacio]
```

Como puedes ver el módulo ***Subvacio*** toma su nombre completo como ***Vacio.Subvacio*** por encontrarse dentro del módulo ***Vacio***.

Esta es una forma no muy difundida de escribir submódulos. Se prefiere escribir cada módulo en su propio fichero. Pero si el módulo a escribir es muy pequeño y concreto puede agregarse tal y como hemos visto.

Un poco de azúcar...

En realidad la llamada a ***defmodule*** es una llamada a función que puede escribirse de esta forma:

```
defmodule(Vacio, do: nil)
```

Los paréntesis son opcionales y el parámetro de lista de propiedades se expande para crear el bloque ***do...end***.

La opción ideal para realizar una organización del código mejor es crear el directorio `vacio` y agregar dentro de este directorio el fichero `subvacio.ex`. El contenido del fichero es como sigue:

```
defmodule Vacio.Subvacio do
end
```

En este caso se especifica explícitamente la jerarquía completa a la que pertenece el módulo. Es necesario ya que no hemos definido este módulo dentro del anterior sino en un fichero nuevo.

Importante

Aunque creamos directorios donde contener el código y organizarlo. Realmente Elixir no tiene en cuenta esta organización y en caso de dejar todos los ficheros juntos en el mismo directorio e incluso asignarles otros nombres no dará ningún tipo de aviso ni error. Es nuestra tarea mantener organizados los ficheros y directorios.

3. Atributos del Módulo

Los atributos son datos especificados dentro de un módulo que pueden proporcionar información para la compilación de un módulo o servir como constantes. Los atributos más empleados en Elixir son los siguientes:

@vsn

Permite definir manualmente la versión del módulo. Si no se especifica la versión del módulo será generada como el cálculo de las funciones del módulo con algoritmo de hash. Debe ser una cadena de caracteres. Será útil para la carga y cambio del código en caliente. Puedes ver más en la Sección8, "Recarga de código" del Capítulo7, ***Procesos y Nodos***.

@moduledoc

Esta entrada se especifica al principio del módulo y permite agregar documentación para el módulo. Será visualizada con el comando ***h*** de la línea de comandos de Elixir y por ExDoc para la generación de la documentación.

@doc

Documentación general para funciones y macros. Esta información puede ser revisada a través del comando ***h*** de la línea de comandos de Elixir y por ExDoc para la generación de la documentación.

@typedoc

Documentación para los tipos de datos definidos en el código. Esta información puede ser revisada en la documentación generada por ExDoc.

@behaviour

Usado para indicar que un módulo cumple las funciones de comportamiento. Puedes revisar la Sección16, "Comportamientos" para más información.

@impl

Acepta un parámetro booleano. Normalmente debe ser ***true*** e indica la implementación de una retro-llamada en la implementación de un comportamiento. De esta forma la implementación de las retro-llamadas de los comportamientos es explícita y ayuda a detectar fallos de implementación. Puedes revisar la Sección 16, "Comportamientos" para más información.

@after_compile

Indica el módulo o una tupla de dos elementos donde el primero será el módulo y el segundo el nombre de la función a llamar. Después de compilar el código se ejecutará la función indicada. En caso de no especificar la función esta será `__after_compile__/1`. Puedes revisar la Sección 2, "Retro-llamadas del Compilador" del Capítulo 5, ***Meta-Programación*** para más información.

@before_compile

Permite definir un módulo o una tupla de dos elementos donde el primer elemento es un módulo y el segundo el nombre de una función o macro que será ejecutada justo antes de comenzar la compilación. Puedes revisar la Sección 2, "Retro-llamadas del Compilador" del Capítulo 5, ***Meta-Programación*** para más información.

@compile

Permite pasar parámetros de configuración para la compilación. Principalmente es empleada para indicar las funciones que deben ser compiladas en línea. Puedes revisar las opciones de compilación en el Apéndice C, ***Compilador*** para más información.

@on_load

Permite definir el nombre de una función como átomo para llamar a esa función cuando el módulo sea cargado por el sistema en tiempo de ejecución. La función debe tener aridad 0 y retornar siempre el átomo ***:ok***.

@deprecated

Nos permite marcar funciones o macros como desfasadas y que aparezca un aviso en el proceso de compilación cada vez que las usemos. Como valor se puede especificar el texto para el mensaje

de aviso. Este mensaje debe dar una solución o alternativa para quien vea el mensaje pueda subsanar la situación.

@dialyzer

Nos permite especificar parámetros de configuración para Dialyzer[1].

Nota

Hay muchos más atributos que puedes revisar en la documentación oficial de Elixir a través de este enlace[2]. Muchos de ellos son acerca de conceptos que no cubre este libro.

Podemos definir nuestros propios atributos también. Estos atributos pueden ser tratados como constantes o como datos acumulativos. En el primer caso no habrá que hacer nada salvo definir un valor y después emplearlos donde los necesitemos:

```
defmodule Matematicas do
  @pi 3.141592653589793

  def pi, do: @pi
end
```

Estos atributos son sobrecargables. Cuando definimos el valor de un atributo este es mantenido hasta encontrar otro lugar donde se redefina. Podemos ver un ejemplo:

```
defmodule Temporizadores do
  @timeout 200
  def peticion_local do
    # código
    Process.sleep @timeout
    # código
  end

  @timeout 500
  def peticion_remota do
    # código
    Process.sleep @timeout
    # código
  end
end
```

En el caso de la petición local el valor del tiempo de espera es configurado como 200 milisegundos y antes de la petición remota el valor de tiempo de espera es redefinido y se emplean 500 milisegundos para la petición remota.

[1] Dialyzer es una herramienta para análisis del código. Esta herramienta nos permite comprobar la exactitud del código escrito comprobando tipos, parámetros, llamadas y flujos de ejecución en frío.
[2] https://hexdocs.pm/elixir/Module.html

En el caso de los datos acumulativos debemos primero definir las características del atributo para poder emplearlo después. Por ejemplo, imagina que quieres crear un atributo que actúe como decorador haciendo que cada función pueda ser exportable como un servicio web. Si agregamos los datos necesarios para realizar la publicación tenemos:

```
defmodule Matematicas do
  Module.register_attribute __MODULE__, :api,
    accumulate: true

  @pi :math.pi()

  @api {:pi, 0, "/api/pi"}
  def pi, do: @pi

  @api {:pow, 2, "/api/pow/:base/:exp"}
  def pow(base, exp), do: :math.pow(base, exp)

  def info, do: @api
end
```

Podemos ver la definición del atributo acumulativo al inicio del módulo. Antes de cada función hemos agregado información sobre la función y una URI además de una última función que nos retorna toda la información acumulada.

4. Aliases e Importación

Cuando la jerarquía de módulos es muy grande tenemos dos opciones para emplear funciones de otros módulos: ***alias*** o ***import***.

Cuando agregamos un alias al inicio de un módulo estamos diciendo al compilador que cada ocurrencia de ese alias debe intercambiarse por el nombre completo. Así por ejemplo un módulo como ***Skirnir.Backend.PostgreSQL*** podemos agregarlo con un alias al inicio y emplear ***PostgreSQL*** únicamente:

```
defmodule IMAP do
  alias Skirnir.Backend.PostgreSQL

  def query(data), do: PostgreSQL.query(data)
end
```

Si queremos emplear un nombre diferente porque pueda colisionar con otro podemos agregar la opción ***as*** y especificar el nombre a emplear. Esto es útil también cuando queremos usar un módulo de Erlang/OTP y queremos darle un nombre más al estilo Elixir:

```
defmodule IMAP do
  alias Skirnir.Backend.PostgreSQL, as: PSQL
  alias :timer, as: Timer
```

```
  def query(data), do: PSQL.query(data)
  def sleep(msec), do: Timer.sleep(msec)
end
```

Por otro lado la importación de funciones desde otro módulo nos evita escribir el nombre del módulo del que procede cada vez que la empleamos la función. Actúa como si la función estuviese definida dentro del módulo. Esto también puede ser un problema por no saber si la función que usamos es local o proviene de otro módulo.

Nota

Las funciones importadas serán importadas como privadas. No están disponibles para ser llamadas desde fuera del módulo.

Si no especificamos ninguna opción se importan todas las funciones y macros del módulo especificado. Si agregamos la opción ***:only*** o ***:except*** limitamos la inclusión a solo lo especificado o todo menos lo especificado según el caso.

Para incluir la función `:timer.sleep/1` en el módulo solo tendríamos que hacer:

```
import :timer, only: [sleep: 1]
```

Para indicar que queremos todas las funciones pero no las macros:

```
import :timer, only: :functions
```

En caso opuesto para importar las macros pero no las funciones:

```
import :timer, only: :macros
```

Por último, para incluir todas las funciones y macros a excepción de una función o varias, podemos escribir:

```
import :timer, except: [start: 0, start_link: 1, init: 1]
```

Importante

Ten cuidado al importar puesto que hay módulos muy grandes y con muchísimas funciones definidas. Lo ideal es emplear ***only*** para asegurarse de incluir únicamente las funciones que necesitemos.

5. Funciones

Como dijimos al principio del capítulo las funciones en Elixir son ciudadanos de primera clase. La mayoría de construcciones son funciones tal y como vimos con ***defmodule***.

Las funciones se definen a través del uso de dos palabras clave ***def*** o ***defp***. La primera es la definición de una función pública mientras que la segunda es una función privada. Las funciones deben definirse siempre dentro de un módulo. Las funciones privadas solo serán accesibles dentro de sus módulos.

Los nombres de las funciones deben comenzar por minúscula como las variables. Pueden contener letras, números, subrayados (_) y de forma opcional una función puede terminar con un símbolo de exclamación (!) o un símbolo de interrogación (?). Esto permite definir funciones más expresivas. Mientras que la exclamación se emplea para funciones que pueden causar excepciones en lugar de retornar errores el símbolo de interrogación se emplea con funciones que responden preguntas. Generalmente con respuestas booleanas.

Lo siguiente en la definición de una función son los argumentos en caso de definir alguno. Podemos agregar paréntesis y tantas variables como argumentos vayamos a recibir. El número de argumentos está delimitado entre 0 y 255.

El cuerpo de la función se encierra entre las palabras ya vistas anteriormente ***do...end***. Todo lo que vaya entre estas palabras será tomado como cuerpo de la función.

Dentro de la función podemos agregar tantas expresiones como necesitemos. El resultado de la última expresión será el retorno de la función:

```
defmodule Area do
  def rectangulo(base, altura) do
    base * altura / 2
  end
  def cuadrado(base) do
    base * base
  end
end
```

Hemos definido las funciones `Area.rectangulo/2` y `Area.cuadrado/1`. La primera acepta dos argumentos llamados ***base*** y ***altura***. La segunda acepta tan solo un argumento llamado ***base***.

Un poco de azúcar...

La definición de funciones parte de la llamada a la macro `Kernel.def/1-2`. Esta macro actúa como una función y acepta uno o dos parámetros dependiendo de si hay argumentos definidos o no. La sintaxis completa y sin azúcar sería algo como:

```
def(rectangulo(base, altura), do: base *
 altura / 2)
```

No es usual desarrollarlo con todos los paréntesis, pero sí de esta forma:

```
def rectangulo(base, altura), do: base *
 altura / 2
```

También podemos emplear ***defp*** y definir funciones internas para el módulo. Estas funciones tienen la capacidad de optimizarse de diferente forma a nivel de compilación. Si una función se emplea solo dentro de un módulo es buena idea definirla como privada.

6. Polimorfismo y Concordancia

Una de las grandes potencias de las funciones de Elixir reside en el polimorfismo. El polimorfismo en Elixir no solo consiste en poder emplear el mismo nombre de función con diferente aridad[3] sino también poder definir un contenido diferente para cada uno de los argumentos.

Elixir no es un lenguaje de tipado estático. Esto quiere decir que no necesitamos declarar las variables ni los argumentos de las funciones empleando un tipo. Por ejemplo no es necesario decir que un argumento de una función o una variable es de tipo cadena. Pero el contenido de una variable o un dato puede emplearse para realizar concordancias. Esto facilita escribir funciones como la siguiente:

```
defmodule Area do
  def calcula({:rectangulo, base, altura}), do: base *
 altura / 2
  def calcula({:cuadrado, base}), do: base * base
  def calcula({:ciruclo, radio}), do: radio * radio * :math.pi
end
```

Como puedes ver hemos realizado polimorfismo sobre la función `Area.calcula/1`. Esta función recibe siempre una tupla con el primer elemento siendo un átomo e identificando el tipo de operación a realizar y los siguientes elementos los datos para ese cálculo.

[3] La aridad es el número de argumentos de una función.

Podemos emplear todo lo visto en la Sección 2.1, "Concordancia" del Capítulo 3, ***Expresiones, Estructuras y Errores*** para completar y realizar mejores concordancias para simplificar y acortar nuestro código.

7. Argumentos Opcionales

Cuando uno de los parámetros es opcional pudiendo obtener un valor por defecto pero sin modificar el resto del código podríamos definir estas funciones de la siguiente forma:

```
def request(uri), do: request(uri, @timeout)
def request(uri, timeout) do
  # código de petición
end
```

En este código vemos la primera línea donde especificamos el valor por defecto dentro de la función al llamar a la función más completa. Esta forma podemos acortarla en Elixir de la siguiente forma:

```
def request(uri, timeout \\ @timeout) do
  # código de petición
end
```

Las dobles barras inclinadas inversas (\\) nos proporcionan la sintaxis para definir la función con uno o dos parámetros. De este modo la función puede ser llamada especificando un solo parámetro o los dos. En el primer caso la llamada se completará empleando como segundo parámetro el valor que hayamos indicado tras las barras inclinadas inversas.

Uno de los puntos más confusos al emplear las barras inversas es cuando empleamos concordancia con el primer parámetro y especificamos la función más de una vez. En esta caso el valor por defecto habría que definirlo todas las veces de la misma forma. Sin embargo eso nos obligaría a escribir de forma repetitiva el código y es una de las prácticas que los creadores de Elixir intentan evitar. Por tanto ese código se escribe de otra forma diferente:

```
def request(uri, timeout \\ @timeout)
def request("/", timeout) do
  # código para request /
end
def request("/users", timeout) do
  # código para request /users
end
```

De esta forma definimos el valor por defecto únicamente en una especificación inicial antes de cada función de concordancia. Al no dar

cuerpo a la función se entiende que esta función debe emplear otras funciones definidas más abajo.

Los parámetros opcionales pueden indicarse en cualquier lugar, ya sea al inicio en mitad del listado de argumentos o al final.

8. Guardas

Además de la concordancia podemos especificar condiciones más generales a cumplirse para poder ejecutar el código del cuerpo de la función. Estas condiciones se especifican entre la definición de la función y el cuerpo de la función tras la palabra clave ***when***.

Veamos algunos ejemplos de guardas:

```
def to_str(i) when is_integer(i), do: Integer.to_string(i)
def to_str(f) when is_float(f), do: Float.to_string(f)
def to_str(a) when is_atom(a), do: Atom.to_string(a)
def to_str(l) when is_list(l), do: List.to_string(l)
def to_str(s) when is_binary(s), do: s

def division(dividendo, divisor) when is_number(divisor) and
                                      is_number(dividendo) and
                                      divisor > 0 do
  dividendo / divisor
end
```

No solo podemos asegurarnos del tipo de dato recibido por la función sino también agregar comprobaciones como en el caso de la función `division/2`.

Nota

Las funciones que podemos emplear para las guardas son limitadas. Las guardas están diseñadas para ser rápidas y no tener efectos colaterales en su ejecución. Puedes ver una lista de funciones permitidas en esta URL:

https://hexdocs.pm/elixir/guards.html#list-of-allowed-expressions

Para casos donde se repite mucho una serie de comprobaciones, por ejemplo si queremos crear una función de guarda específica para comprobar que el primer elemento de una cadena de texto es una letra mayúscula:

```
defmodule Greet do
  defguard is_capital(name) when
```

```
    binary_part(name, 0, 1) >= "A" and
    binary_part(name, 0, 1) <= "Z"

  def greet(name) when is_capital(name), do: "Hello, #{name}"
  def greet(thing_name), do: "Hello, to the #{thing_name}"
end
```

En este código podemos ver la definición de la guarda `Greet.is_capital/1`. Esta guarda comprueba el primer carácter de la cadena de texto pasada como parámetro. Si está dentro del rango de letras mayúsculas se evalúa como verdadera.

La función `Greet.greet/1` emplea la guarda en su primera forma y emite un saludo acorde. La segunda función no implementa esta comprobación y obtenemos un texto diferente.

9. Registros

Los registros son una forma estructurada de almacenar información para un módulo. Esta información se estructura como una tupla internamente y puede tratarse como una lista de propiedades a través de la función implementada automáticamente a través de su definición. Veamos cómo podemos definir un registro dentro de un módulo:

```
defmodule Usuario do
  import Record
  defrecord :usuario, [:nombre, :edad, :clave]
end
```

Para poder emplear los registros debemos importar el módulo ***Record*** para poder emplear la macro `Record.defrecord/2`. Esta macro nos permite especificar en el primer parámetro el nombre del registro y como segundo parámetro una lista de átomos con los nombres de los campos del registro.

La macro genera una serie de funciones que nos facilitan el uso del registro. Podemos ver a continuación cómo podemos crear un registro, asignarle datos y modificarlo:

```
iex> import Usuario
Usuario
iex> Usuario.usuario()
{:usuario, nil, nil, nil}
iex> u = Usuario.usuario(nombre: "Manuel", edad: 38)
{:usuario, "Manuel", 38, nil}
iex> Usuario.usuario(u, clave: "miclave")
{:usuario, "Manuel", 38, "miclave"}
iex> Usuario.usuario(u)
[nombre: "Manuel", edad: 38, clave: nil]
```

Un poco de azúcar...

Los registros son tuplas en realidad por lo que cada elemento dentro de la tupla ocupa un espacio. Si escribimos como entrada una tupla con el mismo número de elementos y como primer elemento el átomo que da nombre a la tupla las funciones entenderán que se trata de un registro:

```
iex> Usuario.usuario({:usuario, nil, nil, nil})
[nombre: nil, edad: nil, clave: nil]
```

Si queremos saber la posición de un elemento en concreto podemos escribir:

```
iex> Usuario.usuario(:nombre)
1
```

Esto puede resultar útil para realizar búsquedas en listas de registros cuando queramos buscar por nombre:

```
iex> ["Manuel", "Miguel", "Juan"] |>
...> Enum.map(&(Usuario.usuario(nombre: &1)))
[
  {:usuario, "Manuel", nil, nil},
  {:usuario, "Miguel", nil, nil},
  {:usuario, "Juan", nil, nil}
]
iex> usuarios |>
...> List.keyfind("Juan", Usuario.usuario(:nombre))
{:usuario, "Juan", nil, nil}
```

La función `List.keyfind/3` necesita como tercer parámetro la posición en la tupla con la que debe comparar el valor a buscar y así encontrar la tupla deseada.

10. Estructuras

Aunque las estructuras (***struct*** en inglés) son otro tipo de dato están muy ligadas a los módulos. Es más, el nombre de una estructura se lo da el módulo donde está contenida. Por ejemplo, si queremos crear una estructura de datos para almacenar los datos de nuestros usuarios podemos escribir un módulo llamado ***Usuario*** conteniendo lo siguiente:

```
defmodule Usuario do
  defstruct [:nombre, :edad, :clave]
end
```

En realidad podríamos definir las estructuras incluso como una lista de propiedades donde la clave es el nombre y el valor es el valor por defecto asignado a ese dato de la estructura:

```
defmodule Usuario do
  defstruct nombre: nil, edad: nil, clave: "1234"
end
```

Para crearlos usamos una sintaxis muy parecida a la de los mapas:

```
iex> usuario = %Usuario{}
%Usuario{clave: "1234", edad: nil, nombre: nil}
iex> usuario = %Usuario{nombre: "Manuel"}
%Usuario{clave: "1234", edad: nil, nombre: "Manuel"}
```

Para acceder y modificar los elementos de la estructura usamos la siguiente forma:

```
iex> usuario.edad
nil
iex> usuario = %Usuario{usuario | edad: 38}
%Usuario{clave: "1234", edad: 38, nombre: "Manuel"}
iex> usuario.edad
38
```

Podemos ver cómo accedemos y modificamos valores de las estructuras fácilmente. Al igual que con los mapas las estructuras facilitan también la concordancia en la llamada de funciones por ejemplo. Son muy empleadas para guardar estados y tipos de datos fijos y sin variación.

Un poco de azúcar...

Internamente y en realidad las estructuras se almacenan como mapas. La diferencia es que almacenan además un campo especial llamado ***__struct__*** donde se indica el nombre de la estructura.

Esto quiere decir que sobre las estructuras y en principio podemos aplicar todas las funciones que podemos aplicar sobre un mapa.

Otro requisito que podemos agregar a una estructura son las claves obligatorias. Es decir, datos que debemos agregar a la estructura y no deben quedar vacíos o se generará un error.

Vamos a reimplementar la estructura de ***Usuario*** e indicar el dato ***nombre*** como obligatorio:

```
defmodule Usuario do
  @enforce_keys [:nombre]
```

```
  defstruct [:nombre, :edad, :clave]
end
```

Si intentamos definir una estructura nueva indicando solo la edad obtenemos un error de tipo ***ArgumentError***.

```
iex> %Usuario{edad: 38}
** (ArgumentError) the following keys must also be given when
 building struct Usuario: [:nombre]
    expanding struct: Usuario.__struct__/1
    iex:1: (file)
```

El mensaje nos indica que debemos proporcionar el dato nombre. Por tanto no podemos olvidar especificar los campos que forzamos a riesgo de obtener una excepción en caso de olvidarlos.

Estructuras vs Registros

La utilidad de las estructuras es la misma que la de los registros. Ambos se emplean para definir datos estructurados dentro de un módulo. Debido a su composición interna y a su facilidad para realizar concordancias en Elixir se emplean más las ***Estructuras***.

11. Protocolos

Otra forma de conseguir polimorfismo es el uso los protocolos. El protocolo define el uso de una función de forma genérica a través del uso de la macro ***defprotocol/2***. En el cuerpo del protocolo se definen las cabeceras de las funciones a implementar. Podemos ver un ejemplo con una función que convierte los datos a JSON[4]:

```
defmodule JSON do
  defprotocol Encoder do
    def encode(data)
  end
end
```

La especificación nos dice que existe una función ***JSON.Encoder.encode/1*** que debe ser implementada. Estas implementaciones se suelen agregar por otras aplicaciones y módulos. Si intentamos ejecutar la función sin implementaciones veremos lo siguiente:

```
iex> JSON.Encoder.encode "hola mundo!"
** (Protocol.UndefinedError) protocol JSON.Encoder not
 implemented for "hola mundo!"
    iex:2: JSON.Encoder.impl_for!/1
```

[4]JSON son las siglas para ***JavaScript Object Notation*** o Notación de Objeto JavaScript. Es un formato muy empleado en servicios web para comunicación entre máquinas.

```
    iex:3: JSON.Encoder.encode/1
```

Podemos realizar la implementación para los datos básicos de la siguiente forma:

```
defimpl JSON.Encoder, for: BitString do
  def encode(data), do: "\"#{data}\""
end
```

Al volver a ejecutar la línea anterior obtendremos:

```
iex> JSON.Encoder.encode "hola mundo!"
"\"hola mundo!\""
```

Podemos agregar tantos bloques ***defimpl*** como tipos de datos necesitemos implementar. Los tipos de datos siempre se referencian con el nombre de algún módulo que contenga información sobre ellos. Los tipos de datos básicos en Elixir son los siguientes:

- Atom
- BitString (para cadenas y binarios)
- Float
- Function (clausuras)
- Integer
- List
- Map
- PID
- Port (manejadores de ficheros, conexiones y otros)
- Reference
- Tuple

Las estructuras son tratadas como tipos de datos en sí. El nombre del módulo que las contiene es empleado como tipo. En el caso de ***%Usuario{}*** visto en la sección anterior podemos realizar la implementación de la siguiente forma:

```
defimpl JSON.Encoder, for: Usuario do
  def encode(%Usuario{nombre: nombre, edad: edad}) do
    "{\"nombre\": #{JSON.Encoder.encode nombre}, \"edad\":
 #{edad}}"
  end
end
```

Podemos probarlo escribiendo el siguiente código:

```
iex> usuario = %Usuario{nombre: "Manuel", edad: 38}
%Usuario{clave: nil, edad: 38, nombre: "Manuel"}
iex> JSON.Encoder.encode usuario
"{\"nombre\": \"Manuel\", \"edad\": 38}"
```

Esto nos garantiza extensibilidad en nuestro código pudiendo agregar en futuros proyectos soporte para estructuras que aún no han sido creadas.

12. Sigilos

En los tipos de datos de la Sección 4.16, "Sigilos" en el Capítulo 2, ***El lenguaje*** vimos los sigilos. Este tipo de dato es realmente una sintaxis endulzada para presentar y escribir la información de forma más semántica en principio. Por ejemplo el sigilo existente para la fecha:

```
iex> ~D/2018-01-01/
~D[2018-01-01]
iex> ~D/January 1st, 2018/
** (ArgumentError) cannot parse "January 1st, 2018" as date,
 reason: :invalid_format
    (elixir) lib/calendar/date.ex:277: Date.from_iso8601!/2
    (elixir) expanding macro: Kernel.sigil_D/2
    iex:1: (file)
```

Como vemos el dato es pasado a través de la función `Date.from_iso8601!/2` para crear la representación interna de la fecha. Podemos tomar el sigilo ***~F*** y aprovechar para usar el formato español de fecha. Primero implementamos las funciones para sigilos en un módulo:

```
defmodule Fecha do
  defstruct [:year, :month, :day]
  def sigil_F(<<day::binary-size(2), "/",
                month::binary-size(2), "/",
                year::binary-size(4)>>, []) do
    %Fecha{year: String.to_integer(year),
           month: String.to_integer(month),
           day: String.to_integer(day)}
  end
end
```

De esta forma cada vez que queramos escribir fechas, en lugar de tener que rellenar los datos de la estructura podemos hacer:

```
iex> import Fecha
Fecha
iex> ~F|01/01/2018|
%Fecha{day: 1, month: 1, year: 2018}
```

Podemos agregar también modificadores al final del sigilo y serán recibidos como lista de caracteres y segundo parámetro de la función. Supongamos que queremos la salida en formato ***Date*** en caso de enviar el modificador ***D***:

```
defmodule Fecha do
  defstruct [:year, :month, :day]
  def sigil_F(<<day::binary-size(2), "/",
                month::binary-size(2), "/",
                year::binary-size(4)>>, []) do
    %Fecha{year: String.to_integer(year),
           month: String.to_integer(month),
           day: String.to_integer(day)}
  end
  def sigil_F(<<day::binary-size(2), "/",
                month::binary-size(2), "/",
                year::binary-size(4)>>, [?D]) do
    Date.from_iso8601! "#{year}-#{month}-#{day}"
  end
end
```

La ejecución con el modificador sería como sigue:

```
iex> import Fecha
Fecha
iex> ~F|01/01/2018|
%Fecha{day: 1, month: 1, year: 2018}
iex> ~F|01/01/2018|D
~D[2018-01-01]
```

Podemos ver cómo con el modificador la salida del sigilo es diferente. Tal y como esperábamos.

Un poco de azúcar...

Este sistema es una dulcificación en realidad. Supone una forma más corta de escribir un dato y procesarlo a algo más complejo y largo por debajo. Por ejemplo si queremos realizar un sistema para realizar una decodificación de JSON o XML y necesitamos agregar XML o JSON en nuestro código los sigilos nos ayudan a que esta escritura sea más legible procesando este código y obteniendo una representación mejor para nivel de ejecución.

13. Clausuras

Las clausuras son funciones anónimas que se almacenan en variables y pueden ser pasadas y retornadas de otras funciones. En Elixir el uso de las clausuras es bastante común cuando empleamos ciertas funciones de módulos como ***Enum*** y para lanzar procesos.

La sintaxis para crear una clausura es la siguiente:

```
iex> m = fn(a, b) -> a * b end
#Function<12.99386804/2 in :erl_eval.expr/5>
```

La palabra clave ***fn*** marca el principio. Tras la flecha (->) todo lo siguiente es considerado el cuerpo de la función hasta encontrar la palabra clave ***end***.

Podemos usar la clausura así:

```
iex> m.(2, 3)
6
```

Para no confundir nombres de funciones con variables conteniendo clausuras debemos especificar tras el nombre de la variable un punto y después los argumentos si hubiese.

También podemos crear referencias a otras funciones existentes de la siguiente forma:

```
iex> output = &IO.puts/1
&IO.puts/1
iex> output.("hola mundo!")
"hola mundo!"
:ok
```

De esta forma se puede tratar como clausura cualquier función.

Importante

Para la ejecución de las variables conteniendo una clausura los paréntesis son obligatorios.

Como curiosidad Joe Armstrong (uno de los creadores de Erlang) comentó a José Valim (creador de Elixir) que esa sintaxis con uso del punto entre los paréntesis y el nombre de la variable daría lugar a muchas dudas y preguntas en el futuro.

Por último vamos a ver un poco de azúcar sintáctico para acortar la creación de algunas de las clausuras más comunes. Normalmente las clausuras suelen procesar de alguna forma los parámetros de entrada. En nuestro ejemplo anterior creamos una clausura para retornar la multiplicación de dos elementos. Podemos crear las clausuras de forma rápida con ***et*** (& o ***ampersand*** en inglés):

```
iex> suma = &(&1 + &2)
&:erlang.+/2
iex> doble = &(&1 * 2)
#Function<6.99386804/1 in :erl_eval.expr/5>
iex> suma.(1, 2)
3
```

```
iex> doble.(10)
20
```

Podemos apreciar cómo creamos las clausuras indicando únicamente si vamos a emplear el primer argumento (&1) y/o el segundo (&2) y teniendo presente que el resultado de la expresión será el resultado retornado por la clausura.

14. Programación Funcional

Al pensar en programación funcional podemos pensar en listas de comprensión, clausuras, currying y/o MapReduce. Hay muchos elementos que definen lo que un lenguaje funcional debe hacer y Elixir cumple muchas de ellas.

14.1. Transparencia referencial o Inmutabilidad

La transparencia referencial indica que una función puede intercambiarse por su valor. La función requiere la falta de estado para ser idempotente[5] o determinista.

La inmutabilidad juega un papel muy importante porque garantiza en parte la eliminación del estado global en caso de existir o la existencia de variables estáticas locales.

14.2. Funciones de Primera Clase y Orden Superior

Lo que se entiende en los lenguajes de programación como ***funciones de primera clase*** es cuando tratamos las funciones como si fuesen otras variables. Básicamente debe cumplir con tres premisas:

1. Una función debe poder pasarse como argumento de otra función.

2. Una función debe poder asignarse a una variable.

3. Una función debe poder ser retornada desde otra función.

Todas estas premisas las revisamos en la Sección13, "Clausuras" por lo que concluimos que Elixir cumple con tener funciones de primera clase.

14.3. Sin Efectos Colaterales

Desgraciadamente hay pocos lenguajes que cumplan esta característica. La teoría dice que al llamar a una función siempre con los mismos

[5]La idempotencia en informática se refiere a obtener siempre el mismo resultado haciendo la misma operación.

parámetros se deben obtener los mismos resultados. No deben existir efectos colaterales que puedan provocar una salida diferente.

Realmente es difícil garantizar esto si necesitamos obtener valores aleatorios, el contenido de un fichero, una entrada de usuario o la fecha y hora por citar algunos ejemplos.

Esto puede conseguirse empleando mónadas y meta-programación para conseguir la implementación funcional completa pero de forma nativa no está implementado y aún implementándolo podríamos saltarnos en cualquier momento la implementación.

14.4. Expresiones frente a Mandatos

Una de las ventajas de los lenguajes funcionales es la tendencia a convertir cada línea de código en una expresión. Cada trozo de código tiene un valor retornado. Pero además de esta expresividad las construcciones tienden a ser más declarativas. Indican qué se quiere conseguir sobre el conjunto de los datos.

Por ejemplo cuando queremos obtener los números pares de un rango de números podemos hacer el filtrado fácilmente empleando `Enum.filter/2`. Este filtrado nos da la posibilidad de validar cada elemento e indicar si entra o no en la lista resultante:

```
iex> Enum.filter 1..10, &(rem(&1, 2) == 0)
[2, 4, 6, 8, 10]
```

O incluso una acción a emplear sobre cada elemento de una lista:

```
iex> Enum.map 1..10, &(&1 * 2)
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Funciones de acumuladores para modificar el resultado obtenido por el análisis de un conjunto de datos. Por ejemplo:

```
iex> Enum.reduce 1..10, &(&1 + &2)
55
iex> Enum.reduce 1..10, &(&1 * &2)
3628800
```

Estas construcciones y otras dentro de ***Enum*** ayudan a entender mejor el código escrito y hacerlo más semántico.

14.5. Evaluación perezosa

Hablamos en la Sección2.2, "Características de Elixir" sobre la evaluación perezosa. Elixir cumple con esta condición. A través de construcciones

como ***Stream*** podemos evaluar de forma perezosa un conjunto de elementos haciendo óptimo su tratamiento sobretodo cuando el conjunto o la colección de información a tratar es muy grande y no puede mantenerse en memoria. Mucho menos varias copias para su tratamiento.

Por ejemplo si queremos tratar un millón de elementos:

```
iex> 1..1_000_000 |>
...> Enum.map(fn(i) -> rem(i, 500) + 1 end) |>
...> Enum.filter(fn(i) -> rem(i, 2) == 0 end) |>
...> Enum.reduce({0, 1}, fn(i, {s, p}) -> {s + i, p * i} end)
```

El rango generará una lista de un millón de elementos para ser procesada por la siguiente función `Enum.map/2` que a su vez genera otra lista resultado de otro millón de elementos para ser procesada por `Enum.filter/2` que elimina solo la mitad resultando otra lista de medio millón de elementos para ser procesada por último por `Enum.reduce/3`.

Toma mucho tiempo procesar cada parte y necesita mucho espacio en memoria. Si cambiamos las funciones `Enum.map/2` y `Enum.filter/2` para emplear en su lugar `Stream.map/2` y `Stream.filter/2` conseguimos un procesamiento perezoso del rango de números:

```
iex> 1..1_000_000 |>
...> Stream.map(fn(i) -> rem(i, 500) + 1 end) |>
...> Stream.filter(fn(i) -> rem(i, 2) == 0 end) |>
...> Enum.reduce({0, 1}, fn(i, {s, p}) -> {s + i, p * i} end)
```

La diferencia básicamente es que ***Stream*** no realiza la acción. Retorna el dato de lo que debe hacerse por cada elemento y cada nueva llamada a otra función de ***Stream*** apila una nueva ejecución para cada elemento. Pero nada se procesa de momento. Al llegar al final y ejecutar `Enum.reduce/3` Es cuando cada elemento se extrae uno a uno del rango, se ejecuta cada clausura para cada elemento en ese momento y se pasa a la reducción.

De esta forma cada elemento extraído del rango es pasado por las clausuras apiladas con antelación y tomada por `Enum.reduce/3`. Por lo que en realidad en ningún momento tenemos en memoria un millón de elementos.

15. Recursividad

Cuando hablamos de lenguajes funcionales siempre nos llega la imagen de la recursividad. El uso de nuevas llamadas a la misma función para reducir el problema poco a poco en lugar de resolver internamente el problema mediante un bucle.

En lenguajes como Elixir ayuda de forma muy positiva el pequeño impacto que agrega una nueva llamada a función. Gracias a disponer de inmutabilidad para sus datos las nuevas variables simplemente referencian el estado y datos ya existentes. Además disponemos de un método conocido en inglés como ***tail recursion*** (recursividad de cola) que nos garantiza no acumular las llamadas recurrentes en la pila de llamadas y por lo tanto poder realizar de forma mucho más económica en memoria la recursividad.

Para mostrar cómo realizar la recursividad he pensado en detallar dos ejemplos bastante característicos de ordenación. El método por mezcla o ***mergesort*** y el método de ordenación rápida o ***quicksort***.

15.1. Ordenación por mezcla (*mergesort*)

Este algoritmo se basa en hacer una partición de los elementos simple, una recursividad sobre cada parte para descomponer el problema lo más que se pueda y una mezcla en la que se va realizando combinación de las partes ordenadas. Este algoritmo es simple en las dos primeras partes y deja la complejidad para la tercera. Primero partimos la lista en trozos de tamaño similar, idealmente igual:

```
iex> lista = [5, 2, 8, 4, 3, 2, 1]
[5, 2, 8, 4, 3, 2, 1]
iex> {L1, L2} = Enum.split lista, div(length(lista), 2)
{[5,2,8],[4,3,2,1]}
```

Esto lo podemos dejar dentro de una función que se llame `MergeSort.separa/1` para dar semántica al código y diferenciarlo dentro del algoritmo. La mezcla podemos hacerla a través de recursividad también. Dadas dos listas ordenadas podemos definirla así:

```
defp mezcla([], lista), do: lista
defp mezcla(lista, []), do: lista
defp mezcla([h1|t1], [h2|_] = l2) when h1 <= h2 do
  [h1|mezcla(t1, l2)]
end
defp mezcla(l1, [h2|t2]) do
  [h2|mezcla(l1, t2)]
end
```

La mezcla la realizamos tomando en cada paso de los datos de cabecera de las listas. El que cumpla con la condición indicada (el que sea menor), concatenando el elemento y llamando a la función con los elementos restantes. Para que este algoritmo funcione ambas listas deben de estar ordenadas, por lo que hay que ir separando elementos hasta llegar al caso particular, que será la comparación de un elemento con otro elemento (uno con uno). Para conseguir esto, realizamos la siguiente recursividad:

```
def ordena([]), do: []
def ordena([h]), do: [h]
def ordena(lista) do
  {lista1, lista2} = separa(lista)
  mezcla(ordena(lista1), ordena(lista2))
end
```

Como puedes observar, para cada sublista y antes de llamar a la mezcla se vuelve a llamar a la función `MergeSort.ordena/1` con lo que llega hasta la comparación de un solo elemento con otro. Después un nivel más alto de dos con dos, tres con tres, y así hasta poder comparar la mitad de la lista con la otra mitad para acabar con la ordenación de la lista de números.

Como dijimos al principio, la complejidad se presenta en la combinación, o función `MergeSort.mezcla/2`, que de forma recursiva se encarga de comparar los elementos de una lista con la otra para conformar una sola en la que estén todos ordenados.

El código completo del algoritmo es el siguiente:

```
defmodule MergeSort do
  def ordena([]), do: []
  def ordena([h]), do: [h]
  def ordena(lista) do
    {lista1, lista2} = separa(lista)
    mezcla(ordena(lista1), ordena(lista2))
  end

  defp separa(lista) do
    Enum.split lista, div(length(lista), 2)
  end

  defp mezcla([], lista), do: lista
  defp mezcla(lista, []), do: lista
  defp mezcla([h1|t1], [h2|_] = l2) when h1 <= h2 do
    [h1|mezcla(t1, l2)]
  end
  defp mezcla(l1, [h2|t2]) do
    [h2|mezcla(l1, t2)]
  end
end
```

Hemos exportado solamente la función `ordena/1`. Para emplear el algoritmo podemos escribir lo siguiente:

```
iex> MergeSort.ordena([1, 7, 5, 3, 6, 2])
[1, 2, 3, 5, 6, 7]
```

15.2. Ordenación rápida (*quicksort*)

En este ejemplo, vamos a llevarnos la complejidad de la parte de combinación a la parte de separación. Esta función se llama ***quicksort*** por

lo rápida que es ordenando elementos. Se basa en la ordenación primaria de las listas para que la mezcla sea trivial.

Este algoritmo se basa en tomar un elemento de la lista como ***pivote*** y separar la lista en dos sublistas. Una primera lista con los elementos menores al pivote y una segunda lista con los elementos mayores y vuelve a llamar al algoritmo para cada sublista.

Esta parte de código la simplificamos con la función `Enum.split_with/2`. Esta función separa los elementos de una lista en dos dependiendo si cumplen una premisa o no. La premisa se evalúa a través de la ejecución para cada elemento de una clausura:

```
iex> [pivote|_] = lista = [5,2,6,4,3,2,1]
iex> {menor, mayor} = Enum.split_with lista, &(&1 <= pivote)
{[5, 2, 4, 3, 2, 1], [6]}
```

La parte de la mezcla es trivial puesto que se recibirán listas ya ordenadas como parámetros. La mezcla consiste solo en concatenar las sublistas. La parte de la recursividad, es muy parecida a la de ***mergesort***. Viendo el código al completo:

```
defmodule QuickSort do
  def ordena([]), do: []
  def ordena([h]), do: [h]
  def ordena(lista) do
    {lista1, [pivote], lista2} = separa(lista)
    ordena(lista1) ++ [pivote] ++ ordena(lista2)
  end

  defp separa([pivote|t]) do
    {menor, mayor} = Enum.split_with t, &(&1 <= pivote)
    {menor, [pivote], mayor}
  end
end
```

Se puede ver que la estrategia de divide y vencerás se mantiene. Por un lado separamos la lista en dos sublistas seleccionando un ***pivote***, retornando ambas sublistas y el pivote. Las sublistas se ordenan mediante recursividad sobre cada sublista por separado.

La ejecución de este código sería así:

```
iex> QuickSort.ordena([1, 7, 5, 3, 6, 2])
[1, 2, 3, 5, 6, 7]
```

16. Comportamientos

Antes de finalizar con los módulos me gustaría introducir la idea de los comportamientos. Esta idea viene del Modelo Actor. Este paradigma es

muy parecido a la programación orientada a objetos y de hecho deriva de la descripción original de Alan Kay sobre la orientación a objetos de Smalltalk.

Nota

En el libro Erlang/OTP Volumen II: Las Bases de OTP[6] introduzco un capítulo completo sobre este concepto de forma agnóstica al lenguaje por lo que puede ser leído incluso si tu finalidad no es aprender Erlang.

La idea de los comportamientos es crear una interfaz común. Este comportamiento define únicamente retro-llamadas y estas deben ser definidas en los módulos implementando este comportamiento.

Vamos a definir el comportamiento ***Crypto***. Este comportamiento definirá una única función que se encargará de tomar un dato de entrada y generar una cadena criptográfica de salida:

```
defmodule Crypto do
  @callback generate(input :: binary) :: binary
end
```

Por ejemplo la implementación de este comportamiento para el algoritmo MD5 podemos hacerla en el módulo ***Crypto.MD5*** de la siguiente forma:

```
defmodule Crypto.MD5 do
  @behaviour Crypto

  @impl true
  def generate(input) do
    :crypto.hash(:md5, input)
    |> Base.encode16()
  end
end
```

Es un ejemplo muy sencillo y vemos que gracias al módulo ***:crypto*** podemos realizar las operaciones sin problema. No obstante si quisiésemos implementar MD5 a través de otra librería manteniendo el comportamiento no habría problema. Igual si queremos implementar otro algoritmo como SHA-1.

Habrás podido ver en el código que hemos empleado ***@impl true***. Este atributo nos permite obtener un grado más de implicación por parte del compilador. Si una función es la implementación de una retro-llamada de un comportamiento y la definimos así el compilador puede comprobar que realmente exista la retro-llamada y concuerde con la definición.

[6] https://books.altenwald.com/book/erlang-ii

Nota

También podemos definir retro-llamadas para macros con ***@macrocallback*** y definir retro-llamadas opcionales con ***@optional_callbacks***. Además queremos otorgar un funcionamiento por defecto a una retro-llamada opcional podemos emplear ***defoverridable***.

17. Interoperatividad y BIFs

Una de las ventajas de Elixir es tener completa interoperatividad con Erlang/OTP y por lo tanto poder emplear cualquier módulo y aplicación escritos en Erlang. Incluso veremos más adelante en el Capítulo 12, ***Ecosistema Elixir*** cómo agregar dependencias desarrolladas en Erlang/OTP y compilarlas con nuestros proyectos.

En Erlang/OTP además se permite realizar la inclusión de funciones escritas en lenguaje C o Rust más recientemente. Estos lenguajes compilan su código de forma que BEAM pueda cargar como librería dinámica el código compilado y llamarlos de diferentes formas. Una de ellas es como ***Built-In Function*** (BIF o Función Integrada en su traducción) o ***Native Implemented Function*** (NIF o Función Nativa Implementada).

Hay muchas BIFs disponibles en Erlang por defecto. Todas las funciones dentro del módulo ***:erlang*** son nativas.

Otra forma sería a través de drivers (manejadores) permitiendo emplear un puerto para realizar una ejecución paralela de ese código.

Capítulo 5. Meta-Programación

Nadie conoce todos los usos de la meta-programación.
—Bjarne Stroustrup

Meta-Programación es una forma de extender el lenguaje de programación usado agregando elementos al mismo que anteriormente no estaban. Es decir, construir semánticas dentro del lenguaje que permitan desarrollar mejores formas de programar. Elixir potencia mucho esta característica para evitar repetir código.

En sí, la creación de estructuras como los bucles, funciones e incluso objetos y clases en otros lenguajes es una forma de intentar escribir menos líneas de código y aún así mantener la misma funcionalidad. Estructuramos mejor el código, lo hacemos comprensible y a la vez más compacto cuando usamos estos elementos.

Estos elementos no tienen porqué ser generales. Pueden cumplir una función semántica únicamente para tu lógica de negocio. Pueden ser construcciones que ayuden a entender mejor tu código. La misión de la meta-programación es facilitar y reducir la cantidad de código a escribir incrementando la legibilidad y comprensión y sobretodo acercando el código a la lógica de negocio que se pretende escribir.

Por ejemplo puedes pensar en código como SQL[1]. Este lenguaje declarativo permite al programador escribir una sentencia indicando qué quiere conseguir de un conjunto de datos seleccionado y aísla a quien lo escribe del tratamiento propio del dato, su forma de almacenamiento y la forma en la que es buscado entre otras características.

Ahora imagina que estás desarrollando un programa que requiere realizar muchos servicios web. Cada servicio web puede tener métodos de tipo GET, PUT, POST o DELETE principalmente. Cada servicio tiene una funcionalidad diferente pero el problema es que para llegar a esa funcionalidad debes de escribir una y otra vez lo mismo. Imagina que fuese este el código:

```
def request(%Request{mehtod: :get, uri: "/"}) do
  {200, "Hola mundo!"}
end
def request(%Request{method: :get, uri: "/farewell"}) do
  {200, "Hasta luego!"}
end
def request(_), do: {404, "No encontrado"}
```

[1]SQL (Structured Query Language) es un lenguaje muy extendido y usado en sistemas gestores de base de datos como SQL Server, MySQL o PostgreSQL.

Aunque la repetición realmente no es muy tediosa si podemos escribirlo de la siguiente forma, ¿no sería mejor?

```
get "/", do: {200, "Hola Mundo!"}
get "/farewell", do: {200, "Hasta luego!"}
```

En principio ahorramos escribir código para procesar la estructura ***Request***, definir la función `request/1` y especificar la última concordancia de la función `request/1`. Reducimos todo a dos líneas y no solo sigue entendiéndose su misión sino que la clarifica aún más.

Lo mejor es que la meta-programación tiene lugar en tiempo de compilación. Cualquier modificación realizada en el código no afectará al tiempo de ejecución por muy pesada que pueda ser y por ello muchas veces Elixir resulta mejor en rendimiento para ciertas tareas.

Revisaremos cómo realiza Elixir los pasos para compilar su código. Después podremos explotarlo en nuestro beneficio gracias a las construcciones ***quote***, ***unquote*** y las macros.

Nota

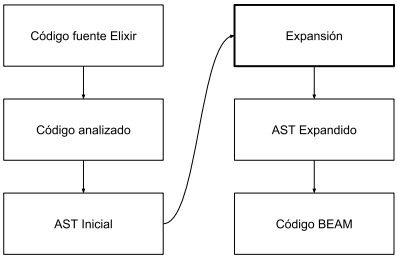
La meta-programación es un tema muy importante en Elixir y muy extenso. En este capítulo repaso cada uno de los aspectos más importantes sabiendo que muchos otros quedarán fuera por su extensión. Por ejemplo el tratamiento a nivel de AST y funciones como `Macro.prewalk/2` o `Macro.postwalk/2` por citar algunas. Solo tienes que echar un vistazo a la documentación (en inglés) de Elixir para las Macros[2] y te harás una idea de lo que hablo.

1. ¿Cómo compila Elixir?

El estado inicial del código está escrito en Elixir. Es el fichero fuente que guardamos en el fichero o los comandos que escribimos en la consola de Elixir. El resultado final es un fichero ***beam*** con el código binario que BEAM entiende.

Entre el estado inicial y el fichero en BEAM se suceden una serie de pasos. Los pasos al completo son los siguientes:

[2] https://hexdocs.pm/elixir/Macro.html



Código fuente en Elixir

El código fuente de Elixir. Como hemos dicho este es el fichero en código Elixir o los comandos ejecutados directamente en la consola.

Código analizado

El código se analiza generando pequeños trozos identificados y más fáciles de procesar en los siguientes pasos. En este nivel el sistema encuentra muchos de los errores de escritura y genera errores de compilación.

AST Inicial

Se genera el AST[3] inicial. Esta información permite analizar las reglas del código y detectar errores semánticos en la escritura del código.

Expansión

Aquí se encuentra gran parte de la potencia de Elixir. En este paso se expanden las macros y se ejecuta el meta-código para generar el código final. Este meta-código son generalmente macros y construcciones de tipo ***quote*** y ***unquote***.

Cada meta-código genera nuevo código que a su vez puede contener nuevo meta-código por lo que esta operación en sí es recursiva y solo finaliza cuando se ha convertido todo el meta-código en código final.

[3]AST son las siglas para Abstract Syntax Tree o Árbol Sintáctico Abstracto.

AST Expandido

Finalmente tenemos un nuevo árbol sintáctico para revisar. Este será el árbol final a compilar en código BEAM si es correcto. Si no es correcto generará un error de compilación.

Código BEAM

Todo se compila a código BEAM y puede ser ejecutado directamente por BEAM a través de consola o de la llamada de sus funciones.

La parte importante y extra que permite la meta-programación es la expansión. La expansión no solo se encarga de modificar e intercambiar código por plantillas o trozos especificados en otras partes sino también ejecuta parcialmente trozos de código y su resultado es empleado dentro del código final.

A través del retorno de la función `Kernel.quote/1` podemos ver cómo es el código en su forma AST:

```
iex> quote do: a = 1
{:=, [], [{:a, [], Elixir}, 1]}

iex> quote do: IO.puts("hello world!")
{{:., [], [{:__aliases__, [alias: false], [:IO]}, :puts]},
 [], ["hello world!"]}
```

Podemos ver la forma en la que Elixir compila cada bloque de código pasado como parámetro y lo deja compilado en modo AST. Podemos probar cada combinación.

Como vemos en el primer ejemplo, cuando empleamos una variable Elixir la codifica como el nombre de esa variable. Si escribimos esto:

```
iex> b = 5
5
iex> q = quote do: a = b + 1
{:=, [],
 [
   {:a, [], Elixir},
   {:+, [context: Elixir, import: Kernel], [{:b, [], Elixir},
 1]}
 ]}
```

La variable ***b*** no existe pero Elixir dice que la asignación constará de la asignación de la suma entre ***b*** y ***1***. Si intentamos evaluar esto obtendremos un error:

```
iex> Code.eval_quoted q
warning: variable "b" does not exist and is being expanded
 to "b()", please use parentheses to remove the ambiguity or
 change the variable name
  nofile:1
```

```
** (CompileError) nofile:1: undefined function b/0
    (stdlib) lists.erl:1354: :lists.mapfoldl/3
```

El contenido de la variable no existe y obtenemos un error de compilación. La variable existe en el contexto donde ejecutamos pero no en el contexto donde se evalúa el código. Veremos más adelante la higiene y los contextos. En estos momentos solo nos vale saber que podemos inyectar el valor de la variable ***b*** gracias a `Kernel.unquote/1`:

```
iex> q = quote do: a = unquote(b) + 1
{:=, [], [{:a, [], Elixir},
          {:+, [context: Elixir, import: Kernel], [5, 1]}]}
iex> Code.eval_quoted q
{6, [{{:a, Elixir}, 6}]}
```

Podemos emplear `Kernel.unquote/1` en cualquier parte de un código dentro de `Kernel.quote/1`. Esto nos permite definir funciones con nombres contenidos en una variable para generar código en lugar de tener que agregarlo a mano en ese momento. Es muy útil para la generación de DSL[4] que nos permita escribir el código de una forma más semántica y ligada a la lógica de negocio.

Otra función igualmente útil para esta función es `Kernel.unquote_splicing/1`. Se encarga de escribir el contenido de una lista como una sucesión de variables, expresiones o literales separados por comas. Esta función se suele emplear para generar los argumentos de funciones. Veremos algunos ejemplos con las macros.

2. Retro-llamadas del Compilador

Cuando se realiza la compilación hay ciertos puntos en los que ciertas retro-llamadas pueden ser empleadas si están definidas en el módulo. En la Sección 3, "Atributos del Módulo" del Capítulo 4, ***Las funciones y los módulos*** pudimos ver los atributos necesarios para configurar el lanzamiento de las retro-llamadas a ejecutar antes y después de la compilación. En esta sección vamos a ver qué podemos hacer con estas retro-llamadas.

Vamos a ver un ejemplo donde se emplea ***before_compile***. En este ejemplo vamos a crear dos módulos el primero llamado ***Hooks*** para contener toda la meta-información necesaria para el segundo módulo llamado ***Publish***.

En el código definiremos unos atributos ***@publish :name*** donde configuramos el nombre de la función a publicar para el módulo.

[4]Domain Specific Language o Lenguaje de Dominio Específico.

Podemos hacer mucho más en lugar de tan solo pasar un átomo pero vamos a intentar mantener el código lo más simple posible:

```
defmodule Hooks do
  defmacro __using__(_data) do
    quote do
      Module.register_attribute __MODULE__, :publish,
                                accumulate: true,
                                persist: false
      @before_compile Hooks
    end
  end

  defmacro __before_compile__(_env) do
    quote do
      def published, do: @publish
    end
  end
end

defmodule Publish do
  use Hooks

  @publish :hello
  def hello, do: "hello world!"

  def hidden, do: "unpublished"

  @publish :bye
  def bye, do: "bye bye!"
end
```

Al compilar y ejecutar vemos:

```
iex> c "examples/cap5/publish.ex"
[Publish, Hooks]
iex> Publish.published
[:bye, :hello]
```

Hemos acumulado los valores y generado una función tras la compilación que agrega esta información. Hasta aquí fácil. Pero si nos fijamos en la información pasada y cambiamos los datos vemos que no hay enlace entre la función y los atributos definidos. Podríamos tener información errónea y el compilador no nos ayudaría.

Podemos hacer otra función usando el atributo ***@on_definition***. Con este atributo en cada definición de función se ejecuta una función para analizar y realizar algunas acciones. Podemos valernos de las funciones definidas en el módulo ***Module*** para los atributos, definir un acumulador y generar información por función. Vamos a repetir el código pero cambiando un poco la forma de funcionamiento.

Esta vez el atributo será siempre igual ***@publish true***. Así indicamos nuestro deseo de publicar la siguiente función que aparezca. Para

cada ocurrencia de función obtenemos una llamada a la función `Hooks.on_def/6`. Esta función obtendrá el atributo del módulo en compilación (tomado del entorno, primer parámetro), hacemos concordancia en el segundo parámetro para obtener solo las funciones (***:def***) y acumulamos en otro atributo (***@published***) cada función.

La restricción que agregamos para poder acumular funciones en ***@published*** es la existencia previa del atributo ***@publish***. Eliminamos el atributo para asegurarnos de no agregar ninguna función siguiente a menos que el atributo se vuelva a definir.

Podemos ver el código completo aquí:

```
defmodule Hooks do
  defmacro __using__(_data) do
    quote do
      Module.register_attribute __MODULE__, :published,
                                accumulate: true,
                                persist: false
      @before_compile Hooks
      @on_definition {Hooks, :on_def}
    end
  end

  def on_def(env, :def, name, _args, _guards, _body) do
    if Module.get_attribute(env.module, :publish) do
      Module.delete_attribute(env.module, :publish)
      Module.put_attribute(env.module, :published, name)
    end
  end

  defmacro __before_compile__(_env) do
    quote do
      def published, do: @published
    end
  end
end

defmodule Publish do
  use Hooks

  @publish true
  def hello, do: "hello world!"

  def hidden, do: "unpublished"

  @publish true
  def bye, do: "bye bye!"
end
```

Podemos agregar también el uso de la retro-llamada ***after_compile***. Esta retro-llamada nos da la posibilidad de ejecutar código de un código recién compilado. No obstante no podemos modificar el código. La retro-llamada de hecho tiene los ***bytecodes*** (el binario compilado) como segundo parámetro de la función.

3. Macros

Podemos definir las macros como funciones pero su ejecución no se lleva a cabo en tiempo de ejecución sino en tiempo de compilación. Su labor es modificar el código escrito y generar otro de mayor extensión o repetitivo.

La macro debe definirse con un nombre, unos parámetros opcionales y el cuerpo a ejecutar. El retorno de la macro debe ser un AST que será integrado en el AST en compilación al momento de llamar a la macro. Se puede retornar una expresión, una estructura, la definición de una función o incluso la definición de un módulo completo.

Podemos considerar una macro como la forma de realizar plantillas de código. Escribiendo las plantillas e insertando citas (***quotes***) donde se requiera para completar esas plantillas en el lugar donde se vayan a emplear.

Las macros nos permiten definir estructuras para generar código. Por ejemplo podemos generar estructuras que nos permitan escribir salidas de tipo SQL o HTML de forma que garanticen un mejor control del código escrito. Un código como el siguiente:

```
div do
  p, do: "Texto"
end
```

Puede generar el texto:

```
<div><p>Texto</p></div>
```

Las macros creadas en este ejemplo son las siguientes:

```
defmodule Html do
  defmacro div(do: block) do
    quote do
      "<div>" <> unquote(block) <> "</div>"
    end
  end
  defmacro p(do: block) do
    quote do
      "<p>" <> unquote(block) <> "</p>"
    end
  end
end
```

La implementación es bastante sencilla pero vemos que va siendo muy repetitiva. Elixir nació para ser extensible y permitirnos escribir código

elegante pero además para escribir lo menos posible. Por ello podemos realizar una reducción más en el código:

```
defmodule Html.Tag do
  defmacro new(name) do
    quote do
      def unquote(name)(do: block) do
        "<#{unquote(name)}>#{block}</#{unquote(name)}>"
      end
    end
  end
end

defmodule Html do
  require Html.Tag

  Html.Tag.new :div
  Html.Tag.new :p
end
```

En este caso hemos combinado macros con funciones. La macro se encarga de crear la función específica para cada etiqueta con el contenido necesario para generar la salida. Podemos mejorarlo mucho más agregando la gestión de atributos para las etiquetas. Una llamada a función que nos permita cambiar una lista de propiedades (***Keyword***) a un texto:

```
defmodule Html.Tag do
  defmacro tag(name) do
    quote do
      def unquote(name)(do: block) do
        "<#{unquote(name)}>#{block}</#{unquote(name)}>"
      end
      def unquote(name)(attrs) do
        attrs_txt = Html.Tag.attrs_to_str(attrs)
        "<#{unquote(name)} #{attrs_txt}/>"
      end
      def unquote(name)(attrs, do: block) do
        attrs_txt = Html.Tag.attrs_to_str(attrs)
        "<#{unquote(name)} #{attrs_txt}>#{block}</
#{unquote(name)}>"
      end
    end
  end

  def attrs_to_str(attrs) do
    convert = fn({key, val}) ->
      "#{Atom.to_string(key)}='#{val}'"
    end
    attrs
    |> Enum.map(convert)
    |> Enum.join(" ")
  end
end

defmodule Html do
  require Html.Tag
```

```
  Html.Tag.tag :div
  Html.Tag.tag :p
end
```

Con estos cambios en el código podemos escribir ahora también atributos para nuestras etiquetas:

```
iex> Html.div class: "texto" do
...>   Html.p do: "Texto"
...>   Html.div class: "clear"
...> end
"<div class='texto'><p>Texto</p><div class='clear'/></div>"
```

Para disponer de macros definidas en otro módulo o en la línea de comandos debemos emplear la macro `Kernel.SpecialForms.require/1-2`. Esta macro nos ayuda a incluir todas las macros del módulo especificado dentro del módulo donde ejecutamos `require/1-2`. Este uso ya lo vimos en el módulo ***Html*** anterior.

Como vimos en la Sección4, "Aliases e Importación" del Capítulo4, ***Las funciones y los módulos*** podemos emplear ***import*** en lugar de ***require***. De esta forma no es necesario anteponer el nombre del módulo:

```
defmodule Html do
  import Html.Tag, only: :macros

  tag :div
  tag :p
end
```

En esencia estas macros son como funciones. Mientras que las funciones mantienen su código en solo una sección específica y es ejecutada en los lugares donde es usada la función la macro se diferencia en que su contenido es incrustado en todos los lugares donde es empleada. Al ahorrarnos el salto la ejecución del código gana un poco más de velocidad. Sin embargo si la macro es muy extensa y se emplea en muchas partes del código, la extensión del mismo se incrementa y con ello el espacio en memoria necesario para almacenar el código.

Uno de los problemas con los parámetros de las macros es que igualmente son incrustados en el código final. En el ejemplo anterior empleamos `Kernel.unquote/1` varias veces con el parámetro ***name***. Si el parámetro proviene de una variable o de un literal no hay mayor problema, pero si proviene de una expresión que obtiene su valor de la llamada a funciones podemos obtener dos veces el resultado generando efectos colaterales indeseados. Por ejemplo:

```
defmodule Nombre do
```

```
  defmacro dar(nombre) do
    quote do
      "<#{unquote(nombre)}>nombre</#{unquote(nombre)}>"
    end
  end
end
```

En esta macro empleamos dos veces `Kernel.unquote/1` para generar la cadena de salida. Empleando una expresión podemos ver:

```
iex> f = fn() -> IO.puts("dando un texto!"); "texto" end
#Function<20.99386804/0 in :erl_eval.expr/5>
iex> require Nombre
Nombre
iex> Nombre.dar(f.())
dando un texto!
dando un texto!
"<texto>nombre</texto>"
```

La expresión se evalúa dos veces. Una vez por cada uso. En lugar de incrustar la expresión podemos emplear solo el valor resultante si empleamos el parámetro ***bind_quoted*** de `Kernel.quote/2`:

```
defmodule Nombre do
  defmacro dar(nombre) do
    quote bind_quoted: [nombre: nombre] do
      "<#{nombre}>nombre</#{nombre}>"
    end
  end
end
```

El código enlaza el resultado de la expresión a una variable que se empleará en la macro. De esta forma solo se evaluará una única vez y se empleará el resultado de la expresión para el resto de lugares donde se emplee. Como ves hemos eliminado el uso de `Kernel.unquote/1` porque empleamos la variable enlazada en lugar de la expresión que nos llega desde fuera. Ahora escribiendo el mismo código:

```
iex> Nombre.dar(f.())
dando un texto!
"<texto>nombre</texto>"
```

Solo se evalúa la expresión una única vez aunque ha sido empleado su resultado dos veces al igual que antes.

Un poco de azúcar...

Emplear ***bind_quoted*** en lugar del uso de `Kernel.unquote/1` es considerado una buena práctica. Siempre que sea posible es mejor emplear este método para evitar problemas difíciles de trazar. Pero es un poco de azúcar sintáctico. En realidad lo que realiza el compilador para el código anterior es:

```
quote do
  nombre = unquote(nombre)
  "<#{nombre}>nombre</#{nombre}>"
end
```

Todas las variables enlazadas generan una línea donde se crea el una variable para asignara el valor requerido. Así a lo largo del resto del código de la macro solo se emplea esta variable y no se vuelve a evaluar el valor de la expresión.

4. Higiene en las Macros

El término higiene se refiere a emplear de forma limpia los contextos en los que son definidos de forma diferente la macro y el código donde se inyecta. De este modo evitamos efectos colaterales no deseados. Por ejemplo:

```
iex> defmodule Datos do
...(15)>   defmacro a do
...(15)>     quote do: a = 1
...(15)>   end
...(15)> end
{:module, Datos,
 <<70, 79, 82, 49, 0, 0, 4, 48, 66, 69, 65, 77, 65, 116,
   85, 56, 0, 0, 0, 127, 0, 0, 0, 13, 12, 69, 108, 105,
   120, 105, 114, 46, 68, 97, 116, 111, 115, 8,
   95, 95, 105, 110, 102, 111, 95, 95, 6, ...>>, {:a, 0}}
iex> require Datos
Datos
iex> a = 10
10
iex> Datos.a
1
iex> a
10
```

Podemos ver cómo la macro define un código en el que asigna el valor ***1*** a la variable ***a***. Al estar los contextos separados por higiene la variable ***a*** definida en la macro es diferente a la variable ***a*** definida en la línea de comandos.

Esta restricción puede ser saltada de forma explícita a través del uso de `Kernel.var!/1`. Esta función nos permite emplear una variable de un contexto superior desde dentro de una macro. En el ejemplo anterior:

```
iex> defmodule Datos do
...>   defmacro a do
...>     quote do: var!(a) = 1
...>   end
...> end
{:module, Datos,
 <<70, 79, 82, 49, 0, 0, 4, 132, 66, 69, 65, 77, 65, 116,
   85, 56, 0, 0, 0, 127, 0, 0, 0, 13, 12, 69, 108, 105,
   120, 105, 114, 46, 68, 97, 116, 111, 115, 8,
   95, 95, 105, 110, 102, 111, 95, 95, 6, ...>>, {:a, 0}}
iex> require Datos
Datos
iex> a = 10
10
iex> Datos.a
1
iex> a
1
```

En este caso como la variable empleada dentro de la macro ***a*** es la variable del contexto donde se incrusta el código de la macro vemos que la variable ***a*** cambia su valor.

5. Extendiendo Módulos

En todo este capítulo hemos revisado poco a poco cómo una de las fortalezas de este lenguaje se va desarrollando haciendo cada vez más extensible el propio lenguaje al agregar personalizaciones. En esta sección abordaremos un paso más a importar funciones de otro módulo mediante `Kernel.use/1-2`. Esta construcción nos permite importar el contenido de un módulo y además ejecutar una macro especial llamada `__using__/1` para implementar código en ese módulo.

Por ejemplo, estamos creando una serie de módulos los cuales siempre importarán una serie de funciones de distintos módulos y requerirán las macros de otros. Si escribimos un módulo base para este tipo de módulos llamado ***ModuloBase*** con este contenido:

```
defmodule ModuloBase do
  defmacro __using__([]) do
    quote do
      require Logger
      import Record
      use Bitwise
    end
  end
end
```

Este módulo nos proporciona las macros de ***Logger***, importa las funciones necesarias para crear registros con ***Record*** e incluso extiende su funcionalidad con el módulo ***Bitwise*** para poder emplear los operadores de tratamiento de bits.

Creamos ahora otro módulo llamado ***Usuario***:

```
defmodule Usuario do
  use ModuloBase

  defrecord :usuario, [:nombre, :edad]

  def nuevo() do
    Logger.info "crea nuevo usuario vacío"
    usuario()
  end
end
```

Podemos emplear tanto los registros como las anotaciones (mediante `Logger.info/1`) gracias a usar el módulo ***ModuloBase***. Un ejemplo del uso de la función:

```
iex> Usuario.nuevo()

14:21:20.687 [info]  crea nuevo usuario vacío
{:usuario, nil, nil}
```

Podemos hacer algo un poco más útil. Por ejemplo cuando creamos servidores mediante el uso de ***GenServer*** (lo veremos en el Capítulo 7, ***Procesos y Nodos***). La forma de implementar las funciones para llamar enviar peticiones al proceso de ***GenServer***, ejecutar el código y obtener un resultado suele tener esta forma:

```
def agrega(nombre, edad) do
  GenServer.call(__MODULE__, {:agrega, nombre, edad})
end
def handle_call({:agrega, nombre, edad}, _from, st) do
  {:reply, :ok, state(st, nombre: nombre, edad: edad)}
end
```

Se definen siempre dos funciones. Una llamada con el nombre semántico de la acción a realizar con los parámetros a ser pasados al proceso del servidor y la función de servidor que deberá manejar la petición dentro del proceso servidor y actualizando o tomando información del estado del servidor.

No te preocupes si aún no entiendes la dinámica de ***GenServer***. Lo importante a ver en este punto es el acto tedioso de tener que escribir siempre dos funciones por cada entrada de API.

Para simplificar esto vamos a crear un módulo base del que extender la funcionalidad. Este módulo será el siguiente:

```
defmodule Call do
  defmacro __using__([]) do
    quote do
      import Call
```

```
      use GenServer

      def start_link() do
        GenServer.start_link(__MODULE__, [], [name:
 __MODULE__])
      end

      def stop() do
        GenServer.stop(__MODULE__)
      end
    end
  end

  defmacro persist(args) do
    quote do
      import Record
      defrecord :state, [unquote_splicing(args)]

      def init([]) do
        {:ok, state()}
      end
    end
  end

  defmacro api(name, args, state, do: code) do
    quote do
      def unquote(name)(unquote_splicing(args)) do
        GenServer.call(__MODULE__, {unquote(name),
 unquote_splicing(args)})
      end
      def handle_call({unquote(name), unquote_splicing(args)},
 _from, var!(state)) do
        unquote(code)
      end
    end
  end
end
```

En este módulo hemos creado tres macros. La primera nos permite usar el módulo en un nuevo módulo y disponer de toda la funcionalidad integrada: importa el módulo ***Call***, extiende la funcionalidad de ***GenServer*** y define las funciones de inicio y parada para el servidor.

La segunda y tercera macros nos permiten definir el estado e implementar cada función del servidor respectivamente. El uso de este módulo nos permite generar código como el siguiente:

```
defmodule User do
  use Call

  persist [:nombre, :edad]

  api :agrega, [nombre, edad], state do
    {:reply, :ok, state(state, nombre: nombre, edad: edad)}
  end

  api :obten, [], state do
    {:reply, state(state), state}
  end
```

```
end
```

Este código persiste los datos de ***nombre*** y ***edad*** para el estado interno del servidor y define para nosotros el registro de estado.

Por otro lado las macros ***Call.api/4*** nos permiten definir la funcionalidad de cada llamada al servidor especificando el nombre de la función, los argumentos que se pasarán, el nombre de la variable de estado y el bloque de código.

La especificación de la variable a emplear para el estado es importante porque así aseguramos que la higiene nos permite emplear el estado desde el bloque de código definido fuera de las macros. Tal y como vimos en la sección anterior.

Podemos probar el ejemplo en la consola:

```
iex> c "call.ex"
warning: variable "state" is unused
  call.ex:28

[Call]
iex> c "user.ex"
[User]
iex> User.start_link
{:ok, #PID<0.101.0>}
iex> User.agrega "Manuel", 38
:ok
iex> User.obten
[nombre: "Manuel", edad: 38]
iex> User.stop
:ok
```

Podemos ver cómo podemos iniciar el servidor, agregar información a través de la función `User.agrega/2`, obtener información con `User.obten/0` y finalmente parar el servidor con `User.stop/0`.

Capítulo 6. Tiempo Real

No importa lo lento que vayas siempre y cuando no te detengas.
—Confucio

Una de las características de BEAM es la capacidad para desarrollar aplicaciones en tiempo real blando (soft realtime). BEAM es una máquina virtual con gestión de procesos, memoria y acceso a recursos, pero lo que no había citado hasta el momento es su capacidad para controlar los eventos horarios.

En principio repasaremos los tipos de datos para representar las fechas y horas. Después analizaremos el uso que hace Elixir o BEAM del tiempo real y las funciones disponibles para este uso.

1. Tipos de datos de Fechas y Horas

En Elixir disponemos del módulo ***Calendar***. Este es un módulo que contiene el comportamiento básico para todos los demás módulos que implementan el uso de fechas y/o horas con o sin zona horaria.

Elixir nos proporciona los siguientes módulos:

Calendar.ISO

Esta implementación de ***Calendar*** según el calendario gregoriano proléptico[1] y el estándar ISO-8601[2] para la representación de fechas y horas.

Date

Este módulo implementa el sigilo ***~D*** para facilitar la representación de la fecha. Se emplea solo para trabajar con fechas como su nombre indica.

DateTime

Similar a ***Date*** permite no solo agregar la hora sino también la zona horaria. El tipo de dato en uso es el definido por el comportamiento ***Calendar.datetime/0***.

[1] Este calendario fue adoptado en 1582. Todas las fechas anteriores a este momento están fuera del calendario pero se convierten al usarlo.
[2] https://es.wikipedia.org/wiki/ISO_8601

NaiveDateTime

Implementa el sigilo ***~N*** para facilitar la representación de la fecha y hora. Este módulo representa la fecha y hora pero a diferencia de ***DateTime*** lo hace sin zona horaria.

Time

Implementa el sigilo ***~T*** para facilitar la representación de la hora. Al no disponer de fecha tampoco dispone de zona horaria.

Además de estos módulos también podemos optar por otras representaciones como la fecha en formato BEAM o en formato de época de Unix. El primero consta de una tupla de dos elementos y cada uno de estos elementos a su vez otra tupla de tres elementos:

```
{{2018, 8, 15}, {18, 30, 15}}
```

Esta sería la representación de fecha y hora. Podemos obtenerla a través del módulo ***NaiveDateTime*** o ***DateTime*** de esta forma:

```
iex> NaiveDateTime.to_erl ~N[2018-08-15 18:30:15]
{{2018, 8, 15}, {18, 30, 0}}
```

La segunda representación no está implementada en todos los módulos. Puedes verla en ***DateTime*** por ejemplo y permite convertir fechas y horas de un formato de época Unix a Elixir y viceversa:

```
iex> naive = ~N[2018-08-15 18:30:15]
~N[2018-08-15 18:30:15]

iex> dt = DateTime.from_naive!(naive, "Etc/UTC")
#DateTime<2018-08-15 18:30:15Z>

iex> DateTime.to_unix dt
1534357815
```

Podemos ver cómo transformamos la representación de una fecha y hora con zona horaria a una fecha y hora ingenua[3] para después obtener la representación de segundos según la época Unix[4].

2. Cálculo de Fechas y Horas

Una de las acciones más comunes cuando trabajamos con fechas es obtener diferencias entre fechas y horas con respecto a otras anteriores. Por ejemplo para obtener el tiempo transcurrido entre un evento y la fecha y hora actual.

[3] El término ***naive*** podemos traducirlo como ingenuo. Es considerada una fecha y hora ingenua porque no dispone de zona horaria.
[4] La época Unix se mide en segundos desde el 1 de enero de 1970.

Para obtener esta información podemos emplear la función `DateTime.diff/2-3`. Esta función nos permite obtener la diferencia de tiempo entre dos fechas y horas por ejemplo:

```
iex> a1 = DateTime.utc_now
#DateTime<2018-09-13 15:00:40.596283Z>
iex> DateTime.diff DateTime.utc_now(), a1, :microseconds
7996178
iex> DateTime.diff DateTime.utc_now(), a1, :milliseconds
16204
iex> DateTime.diff DateTime.utc_now(), a1
20
iex> DateTime.diff DateTime.utc_now(), a1
33
```

Por defecto obtenemos el tiempo en segundos. Pero también podemos especificar otras unidades como ***:second***, ***:millisecond***, ***:microsecond*** o ***:nanosecond***. Pudiendo usar tanto el nombre en singular como en plural.

Nota

Al igual que con ***DateTime*** al emplear `NaiveDateTime.diff/2-3` podemos obtener la diferencia en segundos u otra unidad más pequeña (hasta nanosegundos) de las dos fechas pasadas como parámetro.

Por otro lado el módulo ***Date*** nos permite obtener la diferencia entre fechas cuantizando en número de días a través de la función `Date.diff/2`. Podemos ver cómo funciona a través de los siguientes ejemplos:

```
iex> a1 = Date.utc_today
~D[2018-09-13]
iex> Date.diff ~D[2018-09-20], a1
7
iex> Date.diff ~D[2021-01-01], a1
841
```

Puedes ver cómo obtenemos para la primera una diferencia de 7 días y la segunda de 841 días. En este caso no nos permite obtener la diferencia en otra unidad diferente, solo días.

3. Tiempo monótono

El tiempo real de un ordenador se ve modificado constantemente, se reajusta saltando hacia adelante o a veces hacia atrás para obtener siempre la hora del día. Esto sucede incluso con NTP[5], un cliente que

[5] https://es.wikipedia.org/wiki/Network_Time_Protocol

puede ajustar el reloj local a la hora ***real*** medida con precisión por otro sistema.

Si necesitamos que un evento tarde un tiempo específico o tenga un tiempo de duración máxima (timeout), necesitamos que el tiempo se mantenga constante, avanzando siempre a una misma velocidad. Decimos que necesitamos una medición de tiempo monótona (en ingles ***monotonic time***).

Además, podemos decir que un sistema emplea un tiempo monótono estricto únicamente si siempre que es llamado nos proporciona un valor diferente y ascendente con respecto al anterior.

En Elixir se puede emplear la función `System.monotonic_time/0` para obtener este tiempo monótono:

```
iex> Syste.monotonic_time
-575862592706288779
```

Este valor puede ser positivo o negativo. Establece un número entero de nanosegundos. Si ejecutamos dos veces consecutivas y realizamos la resta entre ambas obtendremos el tiempo real transcurrido entre ambas ejecuciones:

```
iex> t1 = System.monotonic_time
-575862402567925779
iex> t2 = System.monotonic_time
-575862399438399779
iex> t2 - t1
3129526000
```

En mi caso han sido 3,129526 segundos. Recuerda que el valor por defecto está en nanosegundos. Si queremos cambiar la precisión podemos especificar un parámetro en la función y pasar: ***:second***, ***:millisecond***, ***:microsecond*** o ***:nanosecond***.

Los átomos aceptados por `System.monotonic_time/1` pueden estar en singular o plural e incluso puedes emplear un número del tamaño a recibir. Por ejemplo si queremos solo segundos pasamos 1. Si queremos además milisegundos entonces 1000.

Este tiempo monótono será el mecanismo interno que nos ayude a obtener eventos en el tiempo esperado y configurar apropiadamente los temporizadores.

4. Identificadores únicos

Otra de las funciones del temporizador en BEAM es obtener un identificador único basado en el tiempo: `System.unique_integer/1`.

Para obtener este valor BEAM bloquea al menos un nanosegundo el sistema para garantizarnos la propiedad única del valor obtenido. Dicho de otra forma, la misma llamada a la función nunca retornará dos o más veces el mismo valor para la misma instancia en ejecución de BEAM.

La función `System.unique_integer/1` acepta un parámetro. Este parámetro es una lista de átomos. Estos átomos se consideran modificadores del comportamiento. Los dos únicos modificadores disponibles hasta el momento son:

:positive

Nos garantiza el retorno de un valor siempre positivo. Por defecto el valor retornado por la función puede ser tanto positivo como negativo.

:monotonic

Garantiza un valor ascendente en cada retorno. Cada valor será siempre inferior a los siguientes que obtengamos en futuras llamadas a la misma función con los mismos modificadores.

Para información sobre cómo y cuando implementar y usar esta función ver la Sección6, "¿Cómo mantener el código seguro?" .

5. Túneles de tiempo

Básicamente es la configuración del desplazamiento que se empleará para ajustar el reloj de la máquina virtual de Erlang con la hora del sistema cada vez que haya un cambio de hora en el sistema.

Tenemos varias configuraciones posibles que se ajustan con la opción ***+C*** y la opción ***+c*** ambas de BEAM. Entre ellas (tomado de las definiciones del libro Learn You Some Erlang...[6]):

- ***+C no_time_warp +c true***: es la opción por defecto y se mantiene por retrocompatibilidad con las versiones anteriores de Erlang. No se usa túnel de tiempo y se mantiene activo el ajuste.
- ***+C no_time_warp +c false***: es la misma opción que si deshabilitamos el túnel de tiempo, o si empleamos solo ***+c false***.
- ***+C multi_time_warp +c true***: el desplazamiento temporal es ajustado hacia adelante y hacia atrás para coincidir con la hora del sistema operativo. El reloj monótono permanece estable y preciso.
- ***+C multi_time_warp +c false***: el desplazamiento temporal es ajustado hacia adelante y hacia atrás para coincidir con la hora del sistema

[6] http://learnyousomeerlang.com/time#time-warp

operativo. No hay corrección de tiempo por lo que el reloj monótono podría pausarse brevemente.

- ***+C single_time_warp +c true***: no se realizan ajustes de tiempo pero intenta mantenerse el reloj monótono lo más estable posible.
- ***+C single_time_warp +c false***: se mantiene igual que ***no_time_warp***.

En el caso de ***single_time_warp*** se puede llamar a `:erlang.system_flag(:time_offset, :finalize)` para indicar al sistema que puede tomar el offset en ese momento. Es útil para sistemas embebidos donde se emplea el ajuste del reloj solo al inicio del sistema y podemos o queremos ajustar el reloj tras este ajuste.

De todas formas, por la complejidad del ajuste, se recomienda mantener el código seguro o emplear en su lugar siempre ***no_time_warp***.

6. ¿Cómo mantener el código seguro?

La documentación de Erlang nos explica cómo mantener el código seguro[7] en estos casos. En el libro Learn You Some Erlang...[8] nos sintetizan esta información en unos sencillos puntos a seguir cuando queramos utilizar el tiempo, ya sea por eventos o por el uso de fechas y horas:

- Obtener la fecha y/o hora del sistema: `System.os_time/0-1`.
- Medir diferencias de tiempo: `System.monotonic_time/0-1` llamando antes y después del evento a medir y realizar la resta.
- Definir un orden absoluto entre eventos en un nodo: `System.unique_integer([:monotonic])`.
- Medir el tiempo y asegurar que un orden absoluto ha sido definido: ***{System.monotonic_time(), System.unique_integer([:monotonic])}***.
- Crear un nombre único: ***System.unique_integer([:positive])***.
- Crear un nombre único en un cluster (igual que el anterior pero agregar el nombre del nodo).

Siguiendo estas guías, el código debe mantenerse seguro respecto a los túneles de tiempo.

[7] http://www.erlang.org/doc/apps/erts/time_correction.html#id62952
[8] http://learnyousomeerlang.com/time#survive-time-warps

Capítulo 7. Procesos y Nodos

Lanzar y aprender. Todo es progreso.
—Danielle LaPorte

Para comenzar analizaremos la ***anatomía de un proceso*** para comprender lo que es, los mecanismos de comunicación de que dispone y sus características de monitorización y enlazado con otros procesos. Muchas de estas características están presentes en los procesos nativos de sistemas operativos como Unix o derivados (BSD, Linux, Solaris, ...) y otras se pueden desarrollar sin estar a priori integradas dentro del proceso.

Repasaremos también las ventajas e inconvenientes que tienen los procesos de BEAM. Su estructura aporta ventajas como la posibilidad de lanzar millones de procesos por nodo, teniendo en cuenta que cada máquina puede ejecutar más de un nodo. También presenta inconvenientes como la velocidad de procesamiento frente a los procesos nativos del sistema operativo.

Por último, el sistema de compartición de información entre procesos programados para la concurrencia emplea el paso de mensajes en lugar de emplear mecanismos como la memoria compartida y semáforos, o monitores. Para ello proporciona a cada proceso un buzón y la capacidad de enviar mensajes a otros procesos de una forma simple.

1. Anatomía de un Proceso

Un proceso cualquiera, no solo los que son propios de BEAM, tiene unas características específicas que lo distingue, por ejemplo, de un hilo. Los procesos son unidades de un programa en ejecución que tienen un código propio y un espacio de datos propio (normalmente llamado ***heap*** o montículo).

Se podría decir que un proceso cumple los principios del ser vivo, ya que puede nacer (crearse), crecer (ampliando sus recursos asignados), reproducirse (generar otros procesos) y morir (terminar su ejecución). El planificador de procesos de BEAM se encarga de dar paso a cada proceso a su debido tiempo y de aprovechar los recursos propios de la máquina, como son los procesadores disponibles, para intentar paralelizar y optimizar la ejecución de los procesos. Esta sería la vida útil de un proceso.

En BEAM el proceso es además un ***animal social***. Tiene mecanismos que le permiten comunicarse con el resto de procesos y enlazarse a otros procesos de forma ***vital*** o ***informativa***. En caso de que un proceso muera (ya sea debido a un fallo o porque ya no haya más código que ejecutar), el proceso que está enlazado con él de forma vital muere también,

mientras que el que está enlazado de forma informativa es notificado de su muerte.

Para esta comunicación, el proceso dispone de un buzón. En este buzón otros procesos pueden dejar mensajes encolados, de modo que el proceso puede procesar estos mensajes en cualquier momento. El envío de estos mensajes no solo se puede realizar de forma local, dentro del mismo nodo, sino que también es posible entre distintos nodos que estén interconectados entre sí, ya sea dentro de la misma máquina o en la misma red.

Nota

Cuando se lanza un proceso, en consola podemos ver su representación, en forma de cadena, como #PID<X.Y.Z>. Los valores que se representan en esta forma equivalen a:

- *X* es el número del nodo, siendo cero el nodo local.
- *Y* son los primeros 15 bits del número del proceso, un índice a la tabla de procesos.
- *Z* son los bits 16 a 28 del número del proceso.

El hecho de que los valores Y y Z estén representados como dos valores aparte, viene de las versiones de Erlang/OTP R9B y anteriores, donde Y era de 15 bits y Z era un contador de reutilización. Actualmente Y y Z se siguen representando de forma separada para no romper esa compatibilidad.

2. Ventajas e inconvenientes

Hemos realizado una introducción rápida y esquemática de lo que es un proceso en general y un proceso BEAM, para dar una visión a alto nivel del concepto. Como dijimos al principio, los procesos en BEAM no son los del sistema operativo y, por tanto, tienen sus diferencias, sus características especiales y sus ventajas e inconvenientes. En este apartado concretaremos esas ventajas e inconvenientes para saber manejarlos y conocer las limitaciones y las potencias que proporcionan.

Desde el principio hemos remarcado siempre que una de las potencias de BEAM son sus procesos, y es porque fue la primera plataforma con una máquina virtual que genera procesos propios y no del sistema operativo. Esto confiere las siguientes ventajas:

La limitación de procesos lanzados se amplia.

La mayoría de sistemas operativos que se basan en procesos o hilos limitan su lanzamiento a unos 64 mil aproximadamente (16 bits).

BEAM gestiona la planificación de los procesos en ejecución y eleva ese límite a 134 millones[1].

La comunicación entre procesos es más simple y más nutrida.

La programación concurrente se basa en la compartición de datos, bien mediante mecanismos como la memoria compartida y el bloqueo de la misma a través de semáforos, o bien mediante la existencia de secciones críticas de código que manipulan los datos compartidos a través de monitores. BEAM sin embargo emplea el paso de mensajes. Existe un buzón en cada proceso al que se le puede enviar información (cualquier dato) y el código del proceso puede trabajar con ese dato de cualquier forma que necesite.

Son procesos y no hilos.

Cada proceso tiene su propia memoria y por tanto no comparte nada con el resto de procesos. La ventaja principal de tener espacios de memoria exclusiva es que cuando un proceso falla y deja su memoria inconsistente, este hecho no afecta al resto de procesos que pueden seguir trabajando con normalidad. Si el proceso vuelve a levantarse y queda operativo el sistema se autorecupera del error. En el caso de hilos, es posible que un fallo en la memoria (que sí es compartida) afecte a más de un hilo, e incluso al programa entero.

No obstante, no todo es perfecto y siempre hay inconvenientes en las ventajas que se pintan. Por un lado, el hecho de que BEAM se encargue de los procesos y del planificador de procesos tiene su coste. Aunque BEAM está optimizada y el rendimiento de la máquina ha ido en incremento con cada versión lanzada, cualquier lenguaje que emplee directamente los procesos nativos del sistema operativo será más rápido en tareas sencillas.

3. Lanzando Procesos

El lanzamiento de los procesos en Elixir se realiza con una función llamada `Kernel.spawn/1`. Esta función se encarga de lanzar un proceso para ejecutar el código pasado como parámetro, junto con la configuración para lanzar el proceso. El retorno a esta llamada es el identificador del proceso lanzado.

[1] El límite es más bajo por defecto, con el parámetro +P se puede configurar un número mayor, siendo el valor de procesos máximo por defecto de 32.768, y pudiéndose ajustar este valor de 16 a 134.217.727.

Importante

BEAM provee en forma de BIFs las funciones para generar nuevos procesos enviar y recibir mensajes. Elixir las encapsula en el módulo Kernel. En las siguientes secciones veremos solo las funciones disponibles como BIF principalmente. Comentaremos un poco las funciones dentro del módulo Process pero no las emplearemos por ser más limitadas.

La identificación de la función, pasada como parámetro a `Kernel.spawn/1` puede realizarse de varias formas distintas. Se puede emplear una clausura o indicar, a través de una tripleta de datos (módulo, función y argumentos), la función que se ejecutará.

Las opciones que acepta `Kernel.spawn/1` se refieren sobretodo al nodo BEAM en el que se lanza el proceso y al código para ser ejecutado. La primera parte la veremos un poco más adelante. Ahora nos centraremos en el lanzamiento del código en el nodo actual.

Por ejemplo, si queremos ejecutar en un proceso separado la impresión de un dato por pantalla, podremos ejecutar lo siguiente:

```
iex> spawn(IO, :puts, ["hola mundo!"])
```

Podemos hacer lo mismo en forma de clausura, obteniendo el mismo resultado de la siguiente forma:

```
iex> spawn(fn -> IO.puts("hola mundo!") end)
```

Si almacenamos el identificador de proceso llamado comúnmente PID[2] en una variable podemos ver que el proceso ya no está activo mediante la función `Process.alive?/1`:

```
iex> pid = spawn(fn -> IO.puts("hola mundo!") end)
hola mundo!#PID<0.38.0>
iex> Process.alive? pid
false
```

Como dijimos en su definición un proceso se mantiene vivo mientras tiene código que ejecutar. Obviamente, la llamada a la función `IO.puts/1` termina en el momento en el que imprime por pantalla el texto que se le pasa como parámetro, por lo tanto, el proceso nuevo finaliza en ese momento.

Si el código se demorase más tiempo la función `Process.alive?/1` devolvería un resultado diferente. Podemos comprobarlo así:

[2] PID, siglas de Process ID o Identificador de Proceso.

```
iex> pid = spawn(fn -> Process.sleep(10_000) end)
#PID<0.42.0>
iex> Process.alive? pid
true
```

Tras esperar 10 segundos (10_000 milisegundos) el proceso finaliza la ejecución de `Process.sleep/1` y el proceso finaliza.

4. Bautizando Procesos

Otra de las ventajas disponibles en BEAM sobre los procesos, es poder darles un nombre. Esto facilita mucho la programación ya que solo necesitamos conocer el nombre de un proceso para poder acceder a él. No es necesario que tengamos el identificador que se ha generado en un momento dado para ese proceso.

El registro de los nombres de procesos se realiza a través de otra función llamada `Process.register/2`. Esta función se encarga de realizar la asignación entre el nombre del proceso y el PID para que a partir de ese momento el sistema pueda emplear el nombre como identificador del proceso.

El nombre debe de suministrarse como átomo, y cuando se emplee, debe ser también como átomo. Un ejemplo de esto sería el siguiente:

```
iex> pid = spawn(fn -> Process.sleep(60_0000) end)
#PID<0.53.0>
iex> Process.register pid, :timer
true
iex> Process.whereis :timer
#PID<0.53.0>
iex> Process.whereis(:timer) |> Process.alive?()
true
```

Ejecutamos una espera dentro del proceso de 60 segundos (60_000 milisegundos) y aprovechamos para registrar el proceso y comprobar que aún sigue vivo. Si tardamos más de 60 segundos el proceso muere y el nombre vuelve a quedar libre. Hemos empleado la función `Process.whereis/1` para traducir el nombre al proceso.

Nota

Como comentamos en la Sección 6, "Soportando 2 Millones de Usuarios Conectados" del Capítulo 1, ***Lo que debes saber sobre Elixir*** el soporte de dos millones de usuarios se consiguió a través del uso de sistemas que eliminaban cuellos de botella. Para el registro de procesos se empleo el módulo ***Registry***[3] que permite dar nombres a los procesos de BEAM con un mejor rendimiento. Está recomendado cuando se emplea junto con comportamientos como ***GenServer*** o ***Agent***. Esto lo veremos más adelante.

5. Comunicación entre Procesos

Una vez que sabemos cómo lanzar procesos y bautizarlos para poder localizarlos sin necesidad de conocer su identificador de proceso, veamos cómo establecer una comunicación entre procesos. Esta sería la faceta social de nuestros procesos.

Para que un proceso pueda recibir un mensaje debe permanecer en ***escucha***. Esto quiere decir que debe mantenerse en un estado especial, en el que se toman los mensajes recibidos en el buzón del proceso o en caso de que esté vacío espera hasta la llegada de un nuevo mensaje. El comando que realiza esta labor es ***receive***. Tiene una sintaxis análoga a ***case***. En este ejemplo puedes observar la sintaxis que presenta ***receive***:

```
iex> receive do
...>   dato -> IO.puts("recibido: #{inspect(dato)}")
...> end
```

Si ejecutamos esto en la consola, veremos que se queda ***bloqueada***. Esto ocurre porque el proceso está a la espera de recibir un mensaje de otro proceso. La consola de Elixir es también un proceso BEAM, si escribimos `Kernel.self/0` obtenemos su PID.

El envío de un mensaje desde otro proceso se realiza a través de la función `Kernel.send/2`. Vamos a probar con un el siguiente código:

```
iex> pid = spawn(fn ->
...>   receive do
...>     any -> IO.puts("recibido: #{inspect(any)}")
...>   end
...> end)
#PID<0.49.0>
iex> send(pid, "hola")
recibido: "hola"
"hola"
```

[3] https://hexdocs.pm/elixir/master/Registry.html

Usamos la función `Kernel.send/2` para enviar el mensaje al proceso. La información enviada puede ser de cualquier tipo, ya sea un átomo, una lista, una estructura, un mapa o una tupla con la complejidad interna que deseemos.

Nota

Cada proceso BEAM tiene una cola de mensajes que almacena los mensajes recibidos durante la vida del proceso, para que cuando se ejecute ***receive***, el mensaje pueda ser desencolado y procesado.

Para poder realizar una comunicación bidireccional, el proceso que envía el mensaje debe agregar su PID. Si queremos enviar información y recibir una respuesta podemos realizar la siguiente prueba:

```
iex> pid = spawn(fn ->
...>   receive do
...>     {from, message} ->
...>       IO.puts("recibido: #{inspect message}")
...>       send(from, "adios")
...>   end
...> end)
#PID<0.40.0>
iex> send(pid, {self(), "hola"})
recibido: "hola"
{#PID<0.32.0>, "hola"}
iex> receive do
...>   result -> IO.puts "retorno: #{inspect result}"
...> end
retorno: "adios"
:ok
```

Con este código, el proceso hijo creado con `Kernel.spawn/1` se mantiene a la escucha desde el momento de su nacimiento. Cuando recibe una tupla con la forma ***{from, message}***, imprime el mensaje por pantalla y envía ***adios*** al proceso ***from***.

El proceso de la consola es quien se encarga de realizar el envío del primer mensaje hacia el proceso con identificador ***pid*** agregando su propio identificador (obtenido mediante la función `Kernel.self/0`) a la llamada. A continuación se mantiene a la escucha de la respuesta que le envía el proceso hijo, en este caso ***adios***.

Importante

Las secciones de opción dentro de ***receive*** pueden tener también ***guardas***. En caso de que el mensaje recibido no concuerde con ninguna de las opciones dadas será ignorado y se seguirá manteniendo el proceso en modo de escucha.

Como opción de salida para evitar posibles bloqueos en caso de que un evento nunca llegue, o nunca concuerde, o si simplemente se quiere escuchar durante un cierto período de tiempo, podemos emplear la sección especial ***after***. En esta sección podemos indicarle al sistema un número de milisegundos a esperar antes de cesar la escucha, pudiendo indicar un código específico en este caso.

Si por ejemplo, en el código anterior, queremos que el proceso que lanzamos se mantenga solo un segundo en escucha y si no le llega ningún mensaje finalice indicando este hecho, podemos reescribirlo de la siguiente forma:

```
iex> pid = spawn(fn ->
...>   receive do
...>     {from, message} ->
...>       IO.puts("recibido: #{message}")
...>       send(from, "adios")
...>     after 1000 ->
...>       IO.puts("tiempo de espera agotado")
...>   end
...> end)
#PID<0.47.0>
tiempo de espera agotado
```

Si ponemos más segundos y realizamos el envío del mensaje antes de cumplir el tiempo de espera, el comportamiento es exactamente igual al anterior.

Desarrollado en forma de módulo, para aprovechar la recursividad y que el proceso se mantenga siempre activo, podríamos hacerlo así:

```
defmodule Escucha do
  def escucha do
    receive do
      {desde, mensaje} ->
        IO.puts("recibido: #{inspect mensaje}")
        Process.send(desde, :ok, [])
        escucha()
      :stop ->
        IO.puts("proceso terminado")
      after 5000 ->
        IO.puts("dime algo!")
        escucha()
    end
  end

  def para(pid) do
    Process.send(pid, :stop, [])
  end

  def dime(pid, algo) do
    Process.send(pid, {self(), algo}, [])
    receive do
      :ok -> :ok
    end
    :ok
```

```
  end

  def init do
    spawn(Escucha, :escucha, [])
  end
end
```

La función `Escucha.escucha/0` se mantiene a la espera de mensajes. Acepta dos tipos de mensajes. Por un lado el que ya habíamos visto antes, una tupla ***{proceso, mensaje}*** que recibirá desde otro proceso que se comunica con éste (se presentará por pantalla). El otro tipo es un simple mensaje de ***:stop***. Cuando se recibe, como ya no volvemos a ejecutar la función de `Escucha.escucha/0`, el proceso finaliza su ejecución.

Además, cada 5 segundos desde el último mensaje enviado, o desde el último tiempo agotado, o desde el inicio de la ejecución, se imprime el mensaje ***dime algo!***, ejecutando recursivamente la función `Escucha.escucha/0` para seguir con el proceso activo.

La función ***Escucha.dime/2*** no solo envía un mensaje, también espera por una respuesta. Esto garantiza una sincronización entre procesos. Hasta que el proceso preguntado no responde con el parámetro acordado, el proceso que pregunta no puede continuar.

El código para utilizar este módulo podría ser algo como:

```
iex> pid = Escucha.init()
#PID<0.34.0>
dime algo!
iex> Escucha.dime(pid, "hola")
recibido: "hola"
:ok
dime algo!
iex> Escucha.dime(pid, "hola a todos")
recibido: "hola a todos"
:ok
dime algo!
iex> Escucha.para(pid)
proceso terminado
```

Con este ejemplo queda claro que lanzar un proceso es una actividad trivial, al igual que el intercambio de mensajes entre procesos. Esta es la base sobre la que se fundamenta una de las aplicaciones más importantes de BEAM y Elixir, la solución de problemas en entornos concurrentes. También es la base de la mayoría de código que se escribe en este lenguaje. A continuación iremos ampliando y matizando aún más lo visto en este apartado.

6. Procesos Enlazados

Otra de las funcionalidades que proporciona BEAM respecto a los procesos es la capacidad para enlazarlos funcionalmente. Es posible

establecer una vinculación o enlace vital entre procesos de modo que si a cualquiera de ellos le sucede algo, el otro es inmediatamente finalizado por el sistema.

Completando el ejemplo anterior, si el código contuviera un fallo (no de compilación, sino de ejecución), el proceso lanzado moriría pero al proceso lanzador no le sucedería absolutamente nada.

El siguiente fragmento de código contiene un error:

```
iex> pid = spawn(fn -> a = 5; case a do 6 -> :no end end)
#PID<0.39.0>

21:21:01.832 [error] Process #PID<0.39.0> raised an exception
** (CaseClauseError) no case clause matching: 5
    (stdlib) :erl_eval.expr/3

nil
```

El error aparece en la consola provocando que el proceso termine inmediatamente. Sin embargo, al proceso de la consola no le sucede absolutamente nada. Ni tan siquiera se entera porque el proceso fue lanzado sin vinculación.

Nota

La consola está diseñada con diferentes procesos, uno para interactuar con el usuario y otro para ejecutar las expresiones escritas por el usuario. Cuando un código genera una excepción el proceso de ejecución se ve afectado y finaliza. El proceso principal que interactúa con el usuario recibe la información presenta la información de la excepción por pantalla y genera un nuevo proceso para las siguientes ejecuciones.

Cambiando `Kernel.spawn/1` por `Kernel.spawn_link/1` el lanzamiento del proceso se realiza con vinculación, produciendo:

```
> pid = spawn_link(fn -> a = 5; case a do 6 -> :no end end)
** (EXIT from #PID<0.34.0>) shell process exited with reason:
 an exception was raised:
    ** (CaseClauseError) no case clause matching: 5
        (stdlib) :erl_eval.expr/3

21:36:31.870 [error] Process #PID<0.42.0> raised an exception
** (CaseClauseError) no case clause matching: 5
    (stdlib) :erl_eval.expr/3
```

Ahora no solo muere el proceso lanzado sino que con la vinculación también muere nuestro proceso de consola. No obstante, la consola está preparada y relanza otro proceso. Podemos comprobar este hecho ejecutando antes y después la función: `Kernel.self/0`.

Vamos a hacer un ejemplo más completo en un módulo. Tenemos dos procesos que se mantienen a la escucha por un tiempo limitado y uno de ellos en su código tiene un error. En este caso ambos procesos, aunque independientes, finalizarán, ya que uno depende del otro (así se indica al lanzarlos enlazados).

El código sería así:

```
defmodule Gemelos do
  def lanza do
    spawn(Gemelos, :crea, [])
    :ok
  end

  def crea do
    spawn_link(Gemelos, :zipi, [0])
    Process.sleep(500)
    zape(0)
  end

  def zipi(a) do
    IO.puts("zipi - #{a}")
    Process.sleep(1000)
    zipi(a + 1)
  end

  def zape(a) do
    IO.puts("zape - #{a}")
    Process.sleep(1000)
    case a do
      a when a < 5 -> :ok
    end
    zape(a + 1)
  end
end
```

Al ejecutar la función `Gemelos.lanza/0`, se genera un nuevo proceso independiente (sin enlazar). Este proceso a su vez genera otro enlazado que ejecuta la función `Gemelos.zipi/1`. Después se mantiene ejecutando la función `Gemelos.zape/1`. Tenemos tres procesos: el de la consola generado por la llamada a `Gemelos.lanza/0`, el proceso que ejecuta `Gemelos.zipi/1` y el proceso que ejecuta `Gemelos.zape/1`; todos ellos enlazados.

Revisando `Gemelos.zape/1`, podemos ver que cuando el contador llegue a 5, no habrá concordancia posible en la sentencia ***case*** lo que generará un error que terminará con el proceso. Como está enlazado a `Gemelos.zipi/1`, este proceso también finalizará su ejecución.

Visto desde la consola:

```
iex> Gemelos.lanza()
zipi - 0
:ok
zape - 0
```

```
zipi - 1
zape - 1
zipi - 2
zape - 2
zipi - 3
zape - 3
zipi - 4
zape - 4
zipi - 5
zape - 5
zipi - 6
iex>
21:45:18.351 [error] Process #PID<0.86.0> raised an exception
** (CaseClauseError) no case clause matching: 5
    gemelos.ex:22: Gemelos.zape/1
```

Analizando la salida, vemos que se imprime ***zape*** por pantalla hasta que al evaluar el código se produce un error que termina ese proceso y su enlace, es decir, el proceso ***zipi***.

Los enlaces se puede establecer o eliminar a través de las funciones `Process.link/1` y `Process.unlink/1`. El parámetro que esperan ambas funciones es el PID del proceso a enlazar con el actual en el que se ejecutan.

Volviendo sobre nuestro ejemplo anterior, podemos crear un proceso que se encargue de lanzar a los otros manteniendo un enlace con cada uno de ellos. De este modo si uno de ellos finaliza su ejecución el enlace con el proceso lanzador hará que éste finalice por lo que el resto de procesos serán también finalizados en cascada.

El código del lanzador podría crearse en un módulo que usara la función `Process.link/1` de esta forma:

```
defmodule Lanzador do
  def init do
    spawn(Lanzador, :loop, [])
  end

  def loop do
    receive
      {:link, pid} ->
        link(pid)
      :error ->
        throw(:error)
    end
    loop()
  end

  def agrega(lanzador, pid) do
    Process.send(lanzador, {:link, pid}, [])
  end
end
```

Ahora el módulo ***Gemelos*** se simplifica de la siguiente forma:

```
defmodule GemelosLanzador do
  def lanza do
    lanzador_pid = Lanzador.init()
    zipi = spawn(GemelosLanzador, :zipi, [0])
    Lanzador.agrega(lanzador_pid, zipi)
    Process.sleep(500)
    zape = spawn(GemelosLanzador, :zape, [0])
    Lanzador.agrega(lanzador_pid, zape)
    lanzador_pid
  end

  def zipi(a) do
    IO.puts("zipi - #{a}")
    Process.sleep(1000)
    zipi(a + 1)
  end

  def zape(a) do
    IO.puts("zape - #{a}")
    Process.sleep(1000)
    zape(a + 1)
  end
end
```

En este caso, no hemos introducido un error en el código del módulo ***GemelosLanzador*** sino que el error se produce durante el procesamiento de uno de los mensajes del lanzador. En concreto, al enviarle el mensaje ***:error*** al lanzador éste lanza una excepción produciendo la caída automática de los tres procesos.

Importante

Para que la finalización de un proceso provoque que todos sus enlaces también finalicen, debe producirse una finalización por error. Si un proceso finaliza su ejecución de forma normal y satisfactoria, queda finalizado y desenlazado del resto de procesos pero los demás no finalizan. En otras palabras, para que un proceso enlazado sea finalizado por otro, el proceso que provoca la caída de los procesos en cascada debe de haber acabado con un error de ejecución.

7. Monitorización de Procesos

En contraposición al enlace vital, el enlace informativo o monitorización tal y como se conoce en BEAM, permite recibir el estado de cada proceso como mensaje. Este mecanismo permite que podamos conocer si un proceso sigue activo o si ha finalizado su ejecución, ya sea por un error o de forma normal. Este tipo de enlace es diferente al anterior que simplemente propaga los errores haciendo que se produzcan en todos los procesos enlazados.

Un ejemplo simple del paso de mensajes cuando un proceso finaliza se puede ver a través de este sencillo código:

```
iex> {pid, mon_ref} = spawn_monitor(fn -> receive do
...>   any ->
...>     IO.puts("recibido: #{inspect any}")
...>   end
...> end)
{#PID<0.58.0>, #Reference<0.3578407714.2630877190.189326>}
iex> send pid, "hola"
"hola"
recibido: "hola"
iex> flush
{:DOWN, #Reference<0.3578407714.2630877190.189413>, :process,
 #PID<0.95.0>,
 :normal}
:ok
```

El primer proceso tiene un ***receive*** que lo mantiene en espera hasta que le llegue un mensaje. Al enviarle ***hola***, el proceso finaliza satisfactoriamente. La función `Kernel.spawn_monitor/1` se encarga de lanzar el nuevo proceso y enlazarle el monitor al proceso de la consola. Cuando ejecutamos la función `IEx.Helpers.flush/0` podemos ver los mensajes que ha recibido la consola, entre ellos el de finalización del proceso lanzado anteriormente.

Si queremos lanzar un monitor sobre un proceso ya creado tendríamos que recurrir a la función `Process.monitor/1`. El parámetro de esta función es el PID del proceso a monitorizar. Empleando el ejemplo anterior:

```
iex> pid = spawn(fn -> receive do
...>   any ->
...>     IO.puts("recibido: #{any}")
...>   end
...> end)
#PID<0.58.0>
iex> monitor pid
#Reference<0.3578407714.2630877190.189413>
iex> send pid, "hola"
"hola"
recibido: "hola"
iex> flush
{:DOWN, #Reference<0.3578407714.2630877190.189413>, :process,
 #PID<0.58.0>,
 :normal}
:ok
```

El mensaje de finalización enviado por el proceso es una tupla que consta de las siguientes partes:

```
{:DOWN, monitor_ref, :process, pid, reason}
```

La referencia, ***monitor_ref***, es la misma que retorna la función `Process.monitor/1`, el ***pid*** se refiere al identificador del proceso que se está monitorizando y ***reason*** es la razón de terminación. Si la razón es ***:normal*** es que el proceso ha finalizado de forma correcta, en caso contrario, será debido a que encontró algún fallo o se detuvo intencionadamente.

El uso de monitores nos puede servir para crear un ***lanzador*** como el del apartado anterior pero que, al morir un proceso, sea capaz de relanzarlo cuando se recibe la notificación de terminación. Se trata de un ***monitor*** que se puede implementar de la siguiente forma:

```
defmodule Monitor do
  def init do
    spawn(fn -> loop([]) end)
    |> Process.register(:monitor)
    :ok
  end

  def loop(state) do
    receive do
      {:monitor, from, name, fun} ->
        pid = lanza(name, fun)
        send from, {:ok, name}
        loop(Keyword.put(state, pid, [name, fun]))
      {:DOWN, _ref, :process, pid, _reason} ->
        [name, fun] = state[pid]
        new_pid = lanza(name, fun)
        IO.puts("reavivando hijo en #{inspect new_pid}")
        state
        |> Keyword.delete(pid)
        |> Keyword.put(new_pid, [name, fun])
    end
  end

  def lanza(name, fun) do
    pid = spawn_monitor(fun)
    Process.register pid, name
    Process.monitor pid
    pid
  end

  def agrega(name, fun) do
    send :monitor, {:monitor, self(), name, fun}
    receive do
      {:ok, pid} -> pid
    end
  end
end
```

Como ejemplo, podemos utilizar este código en consola de la siguiente forma:

```
iex> Monitor.init()
:ok
iex> Monitor.agrega(:hola_mundo, fn ->
...>   receive do
```

```
...>     any -> IO.puts("Hola #{any}!")
...>   end
...> end)
:hola_mundo
iex> send :hola_mundo, "Manuel"
Hola Manuel!
"Manuel"
reavivando hijo en #PID<0.38.0>
iex> send :hola_mundo, "Miguel"
Hola Miguel!
"Miguel"
reavivando hijo en #PID<0.40.0>
```

El código presente en la clausura no mantiene ningún bucle. Cuando recibe un mensaje se ejecuta presentando por pantalla el texto ***Hola ...!*** y finaliza. El proceso monitor recibe la salida del proceso y vuelve a lanzarlo de nuevo, tal y como se observa en los mensajes ***reavivando hijo en #PID<0.40.0>***.

8. Recarga de código

Uno de los requisitos con los que se desarrolló BEAM fue que el código pudiese cambiar en caliente sin afectar su funcionamiento. El mecanismo para cambiar el código es parecido al que se realiza con los lenguajes de ***scripting*** con algunos matices.

Quizás sea un poco extraño encontrar este tema en un capítulo dedicado a procesos, pero me parece apropiado abordarlo aquí porque la recarga de código afecta directamente a los procesos. La recarga de código afecta más a un proceso que lo emplea de forma continua (como es el código base del proceso), que a otro que lo emplea de forma eventual (funciones aisladas que se emplean en muchos sitios).

Pondremos un ejemplo. Teniendo este código:

```
defmodule Prueba do
  def init, do: loop()
  def loop do
    receive do
      any -> IO.puts("original: #{inspect any}")
    end
    __MODULE__.loop()
  end
end
```

Desde una consola podemos compilar y ejecutar el código como de costumbre:

```
iex> c "prueba.ex"
[Prueba]
iex> pid = spawn(Prueba, :init, [])
#PID<0.39.0>
iex> send pid, "hola"
```

```
original: "hola"
"hola"
```

Se genera un proceso que mantiene su ejecución a través de `Prueba.loop/0` y atiende a cada petición que se le envía al proceso. La función `Prueba.loop/0` a su vez llama a la función `Prueba.code_change/0` de forma ***fully qualified***. Esta forma de llamar a la función le permite a BEAM revisar si hay una nueva versión del código en el fichero binario y recargarla en caso de ser así.

Importante

BEAM puede mantener hasta dos instancias de código en ejecución. Si tenemos un código ejecutándose que no se llama de forma ***fully qualified***, aunque cambiemos el código compilado no se recargará. Pero si se lanza otro proceso nuevo, se hará con la nueva versión del código. En ese momento habrá dos versiones diferentes de un mismo código ejecutándose en la máquina. Si se volviese a modificar el código, el sistema debe de extinguir la versión más antigua del código para quedarse solo con las dos últimas, por lo que los procesos antiguos con el código más antiguo serían eliminados.

Si cambiamos el código del listado anterior para mostrar otro mensaje:

```
  def loop() do
    receive do
      any -> IO.puts("cambio: #{inspect any}")
    end
    __MODULE__.code_change()
  end
```

Vamos a la consola de nuevo y recompilamos. En esta ocasión vamos a emplear el comando de consola ***r*** que nos permite recompilar y recargar un módulo existente:

```
iex> r Prueba
warning: redefining module Prueba (current version defined in
 memory)
  prueba.ex:1

[Prueba]
iex> send pid, "hola"
original: "hola"
"hola"
iex> send pid, "hola"
cambio: "hola"
"hola"
```

Dado que el proceso está ya en ejecución, hasta que no provocamos una segunda ejecución no se produce la recarga del código ni comienza a ejecutar el nuevo código.

El código modificado quedaría así:

```
defmodule Prueba do
  def init, do: loop()

  def code_change do
    :code.purge __MODULE__
    :code.load_file __MODULE__
    __MODULE__.loop()
  end

  def loop do
    receive do
      :update ->
        __MODULE__.code_change()
      any ->
        IO.puts("cambio: #{inspect any}")
        loop()
    end
  end
end
```

Si queremos cargar el código bajo demanda. Podemos compilar el módulo fuera de consola mediante el comando **elixirc**. El proceso para cambiar el código con seguridad desde consola y teniendo acceso al nuevo fichero compilado sería el siguiente:

```
iex> :erlang.suspend_process pid
true
iex> :code.purge Prueba
false
iex> :code.load_file Prueba
{:module, Prueba}
iex> :erlang.resume_process pid
true
iex> send pid, "hola"
original: "hola"
"hola"
iex> send pid, "hola"
cambio: "hola"
"hola"
```

Al bloquear el proceso aseguramos que el código no seguirá en ejecución mientras realizamos los cambios. Todos los mensajes recibidos por el proceso se encolarán y esperaran hasta que el proceso sea reanudado. Las líneas para la eliminación del código antiguo mediante `:code.purge/1` y `:code.load_file/1` pueden ser reemplazadas por `IEx.Helpers.r/1`.

No obstante vemos que la primera petición es igualmente atendida por el código antiguo. Para evitar esto debemos preparar nuestro código. En la segunda modificación puedes ver que hemos agregado el código de cambio bajo la petición ***:update***. Esto nos garantiza que un mensaje de este tipo realizará todo el cambio sin bloqueos. Si compilamos en otra

consola del sistema operativo de nuevo el módulo prueba de la primera versión podemos hacer lo siguiente:

```
iex> send pid, "hola"
cambio: "hola"
"hola"
iex> send pid, :update
:update
iex> send pid, "hola"
original: "hola"
"hola"
```

Sin embargo podemos ver en el código que si no enviamos ***:update*** el bucle se realiza llamando a la función internamente. No hay posibilidad de cambiar el código para el proceso hasta no realizar la petición ***:update***.

Vamos a probar qué sucede si no hacemos la actualización y recargamos el código por segunda vez:

```
iex> send pid, "hola"
cambio: "hola"
"hola"
iex> r Prueba
warning: redefining module Prueba (current version defined in
 memory)
  prueba.ex:1

{:reloaded, Prueba, [Prueba]}
iex> send pid, "hola"
cambio: "hola"
"hola"
iex> r Prueba
warning: redefining module Prueba (current version defined in
 memory)
  prueba.ex:1

{:reloaded, Prueba, [Prueba]}
iex> send pid, "hola"
"hola"
iex> Process.alive? pid
false
```

Aún siendo el mismo código recargado dos veces BEAM lo toma como código diferente porque Elixir realiza una nueva purga y recarga del módulo. El límite de dos versiones del mismo código lo sobrepasamos al realizar la recarga del módulo por segunda vez y nuestro proceso se queda sin código que ejecutar. Entonces finaliza.

9. Gestión de Procesos

Como hemos dicho desde el principio, Elixir ejecuta su código dentro de una máquina virtual, por lo que posee su propia gestión de procesos, de la que ya comentamos sus ventajas e inconvenientes.

En este apartado revisaremos las características de que disponemos para la administración de procesos dentro de un programa. Aunque ya hemos visto muchas de estas características como la creación, vinculación y monitorización, nos quedan otras como el listado, comprobación y eliminación.

Comenzaremos por lo más básico, la eliminación. Elixir provee una función llamada `Process.exit/2` para permitirnos enviar mensajes de terminación a los procesos. Los procesos aceptan estas señales y finalizan su ejecución. El primer parámetro es el PID que es el dato que requiere `Process.exit/2` para finalizar el proceso. El segundo parámetro es opcional y representa el motivo de la salida. Por defecto se envía el átomo ***:normal***. Su sintaxis por tanto es:

```
Process.exit(pid, reason)
```

Por otro lado `Process.list/0` nos proporciona una lista de procesos activos. Con `Process.info/1` obtenemos la información sobre un proceso dado el PID e incluso mediante `Process.info/2` con un parámetro que indica la información específica de la lista de propiedades[4]: enlaces con otros procesos (links), información de la memoria usada por el proceso (memory), la cola de mensajes (messages), por quién está siendo monitorizado (monitored_by) o a quién monitoriza (monitors), el nombre del proceso (registered_name), etc.

10. Nodos

BEAM no solo tiene la capacidad de gestionar millones de procesos en un único nodo, o de facilitar la comunicación entre procesos a través de paso de mensajes implementado a nivel de proceso, sino que también facilita la comunicación entre lo que se conoce como nodos, dando al programador la transparencia suficiente para que dos procesos comunicándose entre nodos diferentes se comporten como si estuviesen dentro del mismo.

Cada nodo es una instancia en ejecución de BEAM. Esta máquina virtual posee la capacidad de poder comunicarse con otros nodos siempre y cuando se cumplan unas características concretas:

- El nodo se debe haber lanzado con un nombre de nodo válido.
- La ***cookie*** debe de ser la misma en ambos nodos.
- Deben poder conectarse entre ellos. Se recomienda que estén en la misma red por seguridad y fiabilidad en la comunicación.

[4]Toda esta información puede ser consultada, con mayor detalle de la siguiente dirección: http://www.erlang.org/doc/man/erlang.html#process_info-2

BEAM dispone de un mecanismo de seguridad de conexión por clave a la que se conoce como ***cookie***. La ***cookie*** es una palabra de paso que permite a un nodo conectarse con otros nodos.

Un ejemplo de lanzamiento de un nodo desde una terminal sería el siguiente:

```
iex --sname test1 --cookie mitest
```

Nota

La opción ***--cookie*** es opcional y en caso de especificar el nombre del nodo, la cookie puede generarse automáticamente y guardarse en el directorio del usuario en un fichero llamado `.erlang.cookie` que solo contendrá una cadena de texto con la ***cookie*** elegida por el sistema. Si el fichero existe se usa la cookie existente en el fichero. Los permisos del fichero no deben permitir a otros usuarios leer el fichero.

Si lanzamos esta línea para ***test1*** y ***test2***, veremos que el símbolo de sistema de la consola de Elixir se modifica adoptando el nombre del nodo de cada uno. En caso de que el nombre de la máquina en la que ejecutamos esto fuese por ejemplo ***altenwald***, tendríamos dos nodos en estos momentos levantados: ***test1@altenwald*** y ***test2@altenwald***.

Desde la consola podemos obtener el nombre del nodo a través de la función `Kernel.node/0` o `Node.self/0`. Los nodos a los que está conectado ese nodo se obtienen con la función `Node.list/0`. Los nodos de un clúster se obtienen con la forma:

```
[Node.self()|Node.list()]
```

O también podemos escribirlo así:

```
[self()|Node.list()]
```

Nota

Las funciones `Node.self/0` y `Kernel.self/0` son equivalentes y realizan exactamente la misma acción con el mismo resultado. Para evitar ambigüedad emplearemos únicamente `Kernel.self/0` y así nos ahorramos escribir el módulo.

Desde la consola podemos usar el siguiente comando para conectar los dos nodos:

```
iex(test1@altenwald)> Node.list
[]
iex(test1@altenwald)> Node.connect :test2@altenwald
true
iex(test1@altenwald)> Node.list
[:test2@altenwald]
```

11. Nodos ocultos

Cuando levantamos una consola de Elixir o una instancia de BEAM, podemos hacerlo de dos formas. Hasta el momento lo hemos hecho de forma visible. Cuando ejecutamos `Node.list/0` obtenemos el nombre de todos estos nodos visibles. Pero también podemos iniciar en modo oculto (***hidden***).

Una consola en modo oculto no se lista por defecto ejecutando `Node.list/0` desde otro nodo. Si nuestro código se basa en el envío de código para ser procesado a todos los nodos conectados esta forma debe ser empleada siempre al conectar con una consola remota. Lo haremos así:

```
iex --sname remote --hidden --remsh test1@altenwald
```

Este comando creará una nueva instancia de BEAM llamada ***remote@altenwald*** y se conectará remotamente al nodo ***test1@alenwald***. Dentro podemos ejecutar:

```
iex(test1@altenwald)> Node.list
[:test2@altenwald]
iex(test1@altenwald)> Node.list :hidden
[:remote@altenwald]
iex(test1@altenwald)> Node.list :connected
[:test2@altenwald, :remote@altenwald]
```

Como podemos observar `Node.list/0` retorna una lista solo con ***test2@altenwald*** ya que ***remote@altenwald*** es una instancia oculta. Sin embargo llamando a `Node.list/1` ya sea con el parámetro ***hidden*** (para obtener solo los ocultos) o con ***connected*** (para obtener todos los nodos conectados, ocultos o no) podemos obtener los dos nodos conectados.

12. Procesos Remotos

Hasta ahora, cuando empleamos la función `Kernel.spawn/1` generamos un proceso local ejecutándose en el nodo local. Si tenemos otros nodos conectados podemos realizar programación paralela o distribuida lanzando la ejecución de los procesos en otros nodos. Es lo que se conoce como un proceso remoto.

Se puede lanzar un proceso remoto con la misma función `Kernel.spawn/2` agregando como primer parámetro el nombre del nodo donde queremos lanzar el proceso. Por ejemplo, si queremos lanzar un proceso que se mantenga a la escucha para dar información en el clúster montado por los dos nodos que lanzamos en el apartado anterior, podemos hacerlo con el siguiente código:

```
defmodule Hash do
  def get(pid, key) do
    Process.send(pid, {:get, self(), key})
    receive do
      any -> any
    end
  end

  def set(pid, key, value) do
    Process.send(pid, {:set, key, value})
  end

  def init(node_name) do
    IO.puts("iniciado")
    spawn(node_name, fn ->
      Keyword.new()
      |> Keyword.put("hi", "hola")
      |> Keyword.put("bye", "adios")
      |> loop()
    end)
  end

  def loop(data) do
    receive do
      {:get, from, key} ->
        Process.send(from, data[key], [])
        loop(data)
      {:set, key, value} ->
        loop(Keyword.put(data, key, value))
      :stop ->
        :ok
    end
  end
end
```

En la función `Hash.init/1`, se agrega el nombre del nodo que se pasa como parámetro a `Kernel.spawn/2`. La comunicación la podemos realizar normalmente como en todos los casos anteriores que hemos visto sin problemas. No obstante y a diferencia de lo visto anteriormente, el PID devuelto tiene su primera parte distinta de cero. Esto indica que está corriendo en otro nodo.

Los procesos en nodos remotos no se puede registrar con la función `Process.register/2`, es decir, no le podemos asociar un nombre y por ello son solo accesibles desde el nodo que los lanzó.

Nota

Si queremos saber el nodo donde se está ejecutando un proceso y disponemos de su PID, podemos emplear `Kernel.node/1` pasando como parámetro el PID. La función no solo acepta PIDs sino también ***ports*** y ***references*** para conocer el nodo donde se abrieron o generaron.

13. Diccionario del Proceso

Para finalizar y dar por terminado este capítulo, indicar que BEAM dispone de un diccionario de datos que puede ser empleado para mantener datos propios del proceso. Podríamos considerarlo como atributos propios del proceso.

Estos datos pueden ser manejados a través del uso de las siguientes funciones:

Process.get/0, Process.get/1

Cuando se indica sin parámetros se obtienen todos los datos contenidos dentro de ese proceso. El formato de esta devolución es una lista de propiedades.

Cuando se indica un parámetro, se toma como clave y se retorna únicamente el valor solicitado.

Process.get_keys/0, get_keys/1

Se emplea para obtener todas las claves o todas las claves cuyos valores son los indicados como único parámetro de la llamada a la función respectivamente.

Process.put/2

Almacena el par clave-valor pasados como parámetros, siendo el primero la clave y el segundo el valor.

Process.delete/1

Se encarga de eliminar el valor correspondiente a la clave pasada como parámetro.

Este diccionario suele emplearse para almacenar meta-información del proceso como cuál es el proceso antecesor o creador (***$ancestors***) y la función con la que se inició el proceso (***$initial_call***).

Capítulo 8. Ficheros y Directorios

Los datos son el activo estratégico de las compañías
por excelencia
—Christian Gardiner

El manejo de datos dentro de la aplicación, dónde almacenarlos y los modos de persistencia disponibles en Elixir son la base para este nuevo capítulo. Estos modos los veremos a través de ficheros. Cómo trabajar con los ficheros, directorios y rutas. El conocimiento de esta información nos puede ser de gran ayuda al escribir guiones (***scripts***) para arranque del sistema o tareas de la herramienta **mix**.

Nota

Como empleamos BEAM y trae consigo Erlang/OTP podemos emplear otros módulos como ***:ets***, ***:dets*** o ***:mnesia***. De los dos primeros hablé en el libro Erlang/OTP Volumen I: Un Mundo Concurrente[1]. Si quieres saber más sobre estos te recomiendo su lectura.

1. Ficheros

En este apartado revisaremos lo que Elixir puede hacer con los ficheros. Para ello, vamos a diferenciar el tratamiento de los ficheros entre los dos tipos existentes: binarios y de texto.

Los ficheros de texto tienen un tratamiento especial, ya que se interpretan algunos caracteres especiales como fin de fichero o salto de línea.

En cambio, los ficheros binarios, tienen un tamaño definido y todos sus bytes son iguales, es decir, no tienen ningún significado especial, aunque se pueden agrupar para definir formas de datos estructuradas.

Pero antes de comenzar con este tratamiento de los ficheros trataremos inicialmente la forma en que BEAM trata los ficheros. BEAM permite trabajar de dos formas diferentes con los ficheros. La primera es a través de un proceso. De esta forma el proceso puede ser referenciado desde otro nodo y obtener acceso al fichero a través de la red. No obstante esta forma pierde un poco en rendimiento y cuando necesitamos incrementar el rendimiento en la lectura de ficheros debemos optar por la segunda forma que es tratando el fichero en ***crudo*** (raw) o directamente.

Además de estas formas, Elixir agrega nuevas mejoras para el tratamiento de los ficheros pudiendo emplearlos a través de un ***Stream*** y cuando

[1] https://books.altenwald.com/book/erlang-i/

tratamos con ficheros grandes pudiendo especificar opciones como *:read_ahead* o *:delayed_write* que veremos más adelante.

1.1. Abriendo y Cerrando Ficheros

Los ficheros que podemos abrir o crear son ficheros que pueden contener texto, imágenes, audio, vídeo, documentos formateados como los RTF, o ficheros binarios de cualquier otro tipo.

La función de que disponemos para poder abrir o crear ficheros es la siguiente:

```
File.open(path, modes_or_function \\ []) -> {:ok, io_device()}
 | {error, posix()}
```

Como primer parámetro tenemos el nombre del fichero junto con su ruta (***path***), y el segundo parámetro corresponde a una lista donde podemos agregar tantas opciones de las siguientes como necesitemos:

:read | :write | :append | :exclusive

> El fichero se puede abrir en modo de lectura (***:read***), escritura (***:write***) o para agregación al final (***:append***). El parámetro ***:exclusive*** es usado para crear el fichero y en caso de existir nos devuelva un error.

:raw

> Abre el fichero sin proceso manejador. Veremos esto más adelante.

:binary

> Las operaciones de lectura retornarán cadenas de texto en lugar de listas de caracteres.

{:delayed_write, size, delay}

> Los datos se mantienen en un buffer hasta que se alcanza el tamaño indicado por ***size*** o hasta que el dato más antiguo en el buffer ha permanecido ahí por más tiempo del especificado en ***delay*** (milisegundos) entonces se escriben a disco. Esta opción se emplea para decrementar los accesos a disco e intentar incrementar el rendimiento del sistema.

{:read_ahead, size}

> Activa el buffer de lectura para las operaciones de lectura que son inferiores al tamaño definido en ***size***. Igual que en el caso anterior, intenta decrementar el número de llamadas al sistema para acceso a disco, por lo que debe aumentar el rendimiento.

:compressed

Crea o abre ficheros comprimidos con ***gzip***. Esta opción puede combinarse con ***:read*** o ***:write***, pero no ambas.

{:encoding, encoding}

Realiza la conversión automática de caracteres para y desde un tipo específico. La codificación por defecto es ***:latin1***. Las tablas de codificación permitidas se pueden revisar en la documentación oficial de la función `file:open/2` (Erlang)[2].

Un ejemplo de apertura de un fichero tan famoso[3] como `/etc/debian_version`, para lectura o escritura y el resultado obtenido:

```
iex> File.open "/etc/debian_version", [:read]
{:ok, #PID<0.52.0>}
iex> File.open "/etc/debian_version", [:write]
{:error, :eacces}
```

Al intentar abrir un fichero para escritura hemos obtenido un error de acceso (***eaccess***) debido a la falta de permisos, ya que un usuario normal no tiene permisos para escribir en ese fichero.

Para cerrar el fichero, y con ello liberar el proceso que se mantiene a la espera de indicaciones para tratar dicho fichero, debemos de emplear la función `File.close/1`. En los ejemplos anteriores haríamos lo siguiente:

```
iex> {:ok, pid} = File.open("/etc/debian_version", [:read])
{:ok, #PID<0.34.0>}
iex> File.close(pid)
:ok
```

Importante

Es importante que cerremos todos los ficheros que abramos. Esto no es solo un uso innecesario de los recursos de los descriptores de ficheros[4] sino también de procesos de BEAM.

La función `File.open/1-2` también dispone de otra forma:

```
File.open!(path, mode_or_function \\ []) -> io_device()
```

[2] http://www.erlang.org/doc/man/file.html#open-2
[3] Para los que usan Debian o Ubuntu, o alguna distribución derivada de estas, es frecuente encontrar el fichero `/etc/debian_version` en el sistema de ficheros.
[4] Son las estructuras del sistema operativo que se emplean para designar que un programa tiene un fichero abierto.

Esta forma nos permite obtener el descriptor directamente para poder emplearlo en flujos de ejecución (para el operador pipe) y lanzar un error del tipo ***File.Error*** en caso de sucederse no poder abrir o crear el fichero por algún motivo.

1.2. Fichero manejado o en *crudo*

Cuando empleamos la opción ***:raw*** al abrir un fichero en lugar de obtener un PID obtenemos una tupla conteniendo información del descriptor de fichero. El fichero puede ser manejado localmente por el nodo que abrió el fichero y de forma simultánea desde diferentes sitios.

Importante

Esto tiene el peligro que en algún punto el descriptor sea cerrado y otro proceso intente usarlo. En ese caso se producirá un error.

La ventaja de abrir el fichero de esta forma es evitar el trasiego de información pasando de un proceso a otro. La información leída mediante el descriptor de fichero será leída inmediatamente por el proceso solicitante y de forma más rápida.

1.3. Lectura de Ficheros de Texto

Los ficheros de texto son los que el sistema interpreta dando un significado concreto a ciertos caracteres especiales. El carácter de avance de línea (\n), es tomado como un salto de línea y el carácter de retorno de carro (\r), en caso de ir seguido al avance de línea, es ignorado y tomado como parte del salto de línea.

Para procesar la lectura de un fichero abierto con la función `File.open/1-2` o `File.open!/1-2` debemos emplear las funciones disponibles en el módulo ***IO***. Estas funciones nos permiten trabajar no solo con los ficheros (manejados por procesos o en crudo) sino también con los flujos de entrada y salida estándar[5].

Funciones como `IO.gets/2` o `IO.read/2` se encargan de leer una línea del fichero. Leen tantos bytes como sean necesarios hasta llegar a un salto de línea o el final del fichero. En el caso de `IO.read/2` podemos indicar como segundo parámetro ***:line*** para obtener este funcionamiento, ***:all*** para leer el fichero completo o un número entero referenciando los bytes máximos a ser leídos.

Como el fichero que vimos en el ejemplo de apertura de ficheros es de tipo texto, podemos ir leyendo línea a línea alternando las funciones

[5]Los dispositivos de entrada y salida estándar suelen ser el teclado y la pantalla o ***stdin*** y ***stdout*** respectivamente.

`IO.gets/2` y `IO.read/2` hasta el fin de fichero y luego cerrarlo, como se ve en el siguiente ejemplo:

```
iex> {:ok, pid} = File.open("/etc/motd", [:read])
{:ok, #PID<0.34.0>}
iex> IO.gets pid, nil
"Linux barbol 3.1.0-1-amd64 Tue Jan 10 05:01:58 UTC..."
iex> IO.read pid, :line
"\n"
iex> IO.gets pid, nil
"The programs included with the Debian GNU/Linux..."
iex> IO.read pid, :line
"the exact distribution terms for each program are..."
iex> IO.gets pid, nil
"individual files in /usr/share/doc/*/copyright.\n"
iex> IO.read pid, :line
"\n"
iex> IO.gets pid, nil
"Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY..."
iex> IO.read pid, :line
"permitted by applicable law.\n"
iex> IO.gets pid, nil
:eof
iex> File.close pid
:ok
```

El uso de la función `IO.gets/2` queda un poco extraño teniendo que agregar como segundo parámetro una valor ***nil***. Esto es así porque la función está preparada para realizar preguntas en la interfaz y obtener una respuesta del usuario. A este nivel y para evitar agregar ese parámetro es mejor emplear únicamente `IO.read/2`.

Si lo que queremos es leer todo el contenido del fichero para almacenarlo en una variable de texto, podemos emplear la función `File.read/1`:

```
iex> File.read("/etc/debian_version")
{:ok, "wheezy/sid\n"}
```

De esta forma evitamos tener que abrir y cerrar el fichero.

1.4. Escritura de Ficheros de Texto

La escritura de ficheros de texto la podemos realizar simplemente a través de las funciones `IO.puts/2` o `IO.write/2`. Ambas funciones son similares pero la primera agrega siempre un retorno de línea al final de la línea. Podemos ver un ejemplo:

```
iex> {:ok, pid} = File.open("mifile.txt", [:write])
{:ok, <0.42.0>}
iex> IO.puts pid, "\nSaldo: 12.3\nTotal: 20.1"
:ok
iex> IO.write pid, "fichero de texto"
:ok
```

```
iex> File.close pid
:ok
```

Hemos empleado ambas funciones para escribir dos mensajes diferentes. La diferencia en el primero se ha agregado un salto de línea dejando cada línea separada y la última línea del fichero ha sido agregada sin salto de línea.

1.5. Lectura de Ficheros Binarios

La lectura de estos archivos la realizaremos expresamente con la función `IO.binread/2`. Como primer parámetro emplearemos el identificador para el fichero abierto y, como segundo parámetro, el tamaño del fichero que será leído.

Aunque podemos emplear otras funciones, esta es la más genérica y nos permitirá realizar las lecturas de los ficheros sin problemas. Si queremos leer la totalidad del fichero es más aconsejable emplear la función `File.read/1` o ***:all*** como segundo parámetro de la función `IO.binread/2`. Al emplear `File.read/1` no es necesario abrir ni cerrar el fichero lo cual es una ventaja.

Un ejemplo de lectura de una imagen sería la siguiente:

```
iex> filename = "/usr/share/pixmaps/debian-logo.png"
"/usr/share/pixmaps/debian-logo.png"
iex> {:ok, pid} = File.open(filename, [:read, :binary])
{:ok, #PID<0.34.0>}
iex> IO.binread pid, 16
<<137, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, 13, 73, 72, 68,
 82>>
iex> File.close pid
:ok
```

Los bytes leídos de nuestro fichero binario imagen se muestran como una lista binaria de enteros. En concreto hemos leído 16 bytes del principio del archivo PNG. Podemos leer 8 bytes que conforman la firma del PNG y los 8 que contienen la cabecera del PNG en la siguiente forma (como podemos ver en la wikipedia[6]):

```
iex> filename = "/usr/share/pixmaps/debian-logo.png"
"/usr/share/pixmaps/debian-logo.png"
iex> {:ok, pid} = File.open(filename, [:read, :binary])
{:ok, #PID<0.34.0>}
iex> <<137, "PNG", 13, 10, 26, 10>> = IO.binread(pid, 8)
<<137, 80, 78, 71, 13, 10, 26, 10>>
iex> <<length::size(32), "IHDR">> = IO.binread(pid, 8)
<<0, 0, 0, 13, 73, 72, 68, 82>>
iex> <<width::size(32), height::size(32),
...>   depth::size(8), color::size(8),
...>   compression::size(8), filter::size(8),
```

[6] http://en.wikipedia.org/wiki/Portable_Network_Graphics

```
...>   interlace::size(8)>> = IO.binread(pid, length)
<<0, 0, 0, 48, 0, 0, 0, 48, 8, 6, 0, 0, 0>>
iex> IO.puts "Image #{width}x#{height} pixels"
Image 48x48 pixels
:ok
iex> File.close pid
:ok
```

Siguiendo las directrices de la especificación de los ficheros PNG, hemos podido extraer, gracias a los binarios y al tratamiento de bits, el tamaño de la imagen y muchos otros datos que podríamos también pasar por pantalla.

Es aconsejable realizar el tratamiento de ficheros binarios siempre a través de binarios. En el ejemplo de la lectura del PNG hemos visto cómo se desempaqueta el entramado de bytes para obtener la información codificada en el fichero. Teniendo la definición de otros tipos de documentos binarios se podría hacer lo mismo para ficheros de audio como los WAV o MP3, o para ficheros de vídeo como los AVI o MOV.

1.6. Escritura de Ficheros Binarios

La escritura de ficheros binarios se realiza con la función `IO.binwrite/2` igual que hemos hecho con los archivos de texto.

Vamos a realizar un ejemplo simple copiando un fichero. Abrimos el fichero que queremos leer y el creamos el fichero destino donde queremos escribir. A continuación hacemos una lectura completa del fichero con la función `File.read/1`:

```
iex> {:ok, contenido} = File.read("logo.png")
{:ok, <<137, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, 13, 73,
        72, 68, 82, 0, 0, 0, 48, 0, 0, 0, 48, 8, 6,
        0, ...>>}
iex> {:ok, destino} = File.open("logo2.png",
 [:write, :binary])
{:ok, #PID<0.35.0>}
iex> File.binwrite destino, contenido
:ok
iex> File.close destino
:ok
```

1.7. Acceso aleatorio de Ficheros

Hasta ahora hemos eliminado el contenido de los archivos que hemos escrito. Si lo que quisiéramos es modificar un archivo binario (en nuestro ejemplo modificaremos la cabecera PNG), la apertura del fichero debería agregar el parámetro ***:read*** además de ***:write***, tal y como se ve en este ejemplo:

```
File.open "file.bin", [:read, :write, :binary]
```

Para modificar una parte específica del fichero, habrá que desplazar el puntero al punto exacto donde queremos escribir. Esto es lo que se conoce como escrituras y/o lecturas aleatorias (o no secuenciales).

La función que permite realizar este tipo de movimientos por el fichero es `:file.position/2`[7]. Esta función nos permite desplazarnos por el fichero a posiciones absolutas o relativas al principio del fichero, también llamado ***:bof***, o ***begin of file***, relativas a la posición actual: ***:cur***, o relativas al final del fichero: ***:eof***, o ***end of file***.

Los parámetros que podemos emplear son:

```
iex> {:ok, pid} = File.open("logo.png",
 [:binary, :write, :read])
{:ok, #PID<0.99.0>}
iex> :file.position pid, 1024
{:ok, 1024}
iex> :file.position pid, {:cur, -24}
{:ok, 1000}
iex> :file.position pid, :eof
{:ok, 1718}
iex> :file.position pid, {:eof, 24}
{:ok, 1742}
iex> :file.position pid, {:eof, -24}
{:ok, 1694}
iex> :file.position pid, :bof
{:ok, 0}
iex> File.close pid
:ok
```

Vemos que el tamaño total del fichero es 1718 bytes al movernos al final del mismo e incluso podemos desplazarnos más allá del tamaño del fichero. Esto es posible porque abrimos el fichero para escritura y se nos permite agregar más información para ampliar el tamaño del fichero.

2. Flujos de datos

Al tratar con ficheros grandes introducir todo el contenido dentro de la memoria no es una opción. En lenguajes imperativos se suele emplear un bucle o una iteración donde en cada ejecución se ejecuta el bloque interior hasta llegar al final del fichero. En Elixir esto no es una opción. Tendríamos que construir conjuntos de funciones para leer y procesar la información cada vez. Aprovechando la funcionalidad del operador pipe lo ideal es convertir el fichero abierto en un flujo (***stream***) y tratarlo.

La función que emplearemos para esto es `File.stream!/1-3`. Los parámetros son los mismos que ya vimos para `File.open/1-2` agregando un tercer parámetro que es equivalente al segundo parámetro

[7] En este caso debemos emplear una función traída de Erlang porque no hay ninguna disponible para hacer lo mismo en Elixir.

de la función `IO.read/2` o `IO.binread/2`. De esta forma podemos indicar el fichero a abrir, las opciones de apertura y por último la forma en la que obtener en cada petición de flujo los datos.

Supongamos que queremos abrir un fichero de texto. Un fichero de anotaciones (***logs***) como puede ser `access.log`. Estos ficheros pueden ser muy grandes. Queremos abrirlo y analizarlo línea a línea:

```
File.stream! "access.log", [:read], :line
```

Supongamos que tenemos este fichero `access.log`:

```
127.0.0.1 [2018-10-10 13:55:36] "GET / HTTP/1.1" 200 2326
127.0.0.5 [2018-10-10 13:57:11] "GET / HTTP/1.1" 200 2326
127.0.0.2 [2018-10-10 13:57:45] "GET / HTTP/1.1" 200 2326
```

Incluso podemos generar un fichero similar con cientos de líneas. Ahora este código:

```
iex> File.stream!("access.log", [:read], :line) |>
...> Enum.each(&(IO.puts "=> #{String.trim &1}"))
=> 127.0.0.1 [2018-10-10 13:55:36] "GET / HTTP/1.1" 200 2326
=> 127.0.0.5 [2018-10-10 13:57:11] "GET / HTTP/1.1" 200 2326
=> 127.0.0.2 [2018-10-10 13:57:45] "GET / HTTP/1.1" 200 2326
:ok
```

De esta forma podemos realizar un tratamiento del fichero. Supongamos que queremos obtener todos los valores únicos de direcciones IP que aparecen al principio de cada línea:

```
def get_addresses(file) do
  File.stream!(file, [:read], :line)
  |> Stream.map(&String.trim/1)
  |> Stream.map(&(String.split(&1, " ")))
  |> Stream.map(&Kernel.hd/1)
  |> Stream.uniq()
  |> Enum.sort()
end
```

Esta función nos permite tratar el fichero y obtener la información sin necesidad de cargar todo el fichero en memoria. Al agregar funciones del módulo ***Stream*** es como si las fuésemos encadenando en una sola función.

3. Gestión de Ficheros

Además de todo lo visto anteriormente para la creación, modificación y lectura de un fichero, podemos realizar administración de nuestros

ficheros: renombrarlos, cambiar sus permisos, propietario[8], copiar el fichero, truncarlo o eliminarlo.

3.1. Nombre del fichero

Doy por supuesto que todos conocemos que, los nombres de los ficheros se componen por la ruta en la que se ubica el fichero, su nombre y extensión. En Elixir, el módulo ***Path*** nos permite obtener la información correspondiente a un nombre de fichero: su ruta, su nombre, su nombre raíz (sin extensión) y/o su extensión.

Para esto, disponemos de varias funciones:

```
iex> filename = "/home/bombadil/logo.png"
"/home/bombadil/logo.png"
iex> Path.basename filename
"logo.png"
iex> Path.rootname filename
"/home/bombadil/logo"
iex> Path.dirname filename
"/home/bombadil"
iex> Path.extname filename
".png"
```

Como en otros lenguajes `Path.basename/1` nos retorna el nombre del fichero sin ruta. `Path.dirname/1` nos devuelve la ruta sin el nombre del fichero. También tenemos `Path.rootname/1` que nos retorna el nombre del fichero (con ruta si dispone de ella) y `Path.extname/1` que nos da únicamente la extensión (con el punto incluido).

También disponemos de la función `Path.absname/1` que retorna siempre el nombre del fichero de forma absoluta. Si le pasamos una ruta relativa o un fichero sin ruta, obtenemos un nombre de fichero absoluto, completado con la ruta de trabajo actual:

```
iex> Path.absname "logo.png"
"/home/bombadil/logo.png"
```

3.2. Copiar, Mover y Eliminar Ficheros

Una de las acciones básicas cuando se trata con ficheros son estas tres que encabezan la sección actual. Para estas acciones Elixir provee de tres funciones: `File.copy/2`, `File.rename/2` y `File.rm/1`.

Un ejemplo de cómo podemos emplear estas funciones:

```
iex> File.copy "logo.png", "logo2.png"
```

[8] El cambio de permisos y propietario depende de cada sistema operativo, sistema de ficheros y los permisos en sí que tenga el usuario que lanzó la ejecución del programa.

```
{:ok, 1718}
iex> File.rename "logo2.png", "milogo.png"
:ok
iex> File.rm "milogo.png"
:ok
```

La función de `File.rename/2`, además de para cambiar el nombre del fichero, nos puede servir para cambiar la ubicación del fichero si indicamos una ruta distinta, por ejemplo:

```
File.rename "logo.png", "/tmp/logo.png"
```

Nota

Las operaciones se realizan sobre ficheros específicos, no sobre grupos de ficheros como los comandos de consola de los sistemas operativos, por lo que el uso de comodines como asterisco (*) o interrogante (?) no se tienen en cuenta como tal, sino que son interpretados como parte del nombre del fichero.

3.3. Permisos, Propietarios y Grupos

Otro de los aspectos relevantes cuando gestionamos ficheros, son sus permisos y su pertenencia a un usuario o grupo. El cambio de los permisos se puede realizar mediante la función `File.chmod/2`. El cambio de propietario se hace mediante `File.chown/2` y el cambio de grupo a través de `File.chgrp/2`.

La función `File.chmod/2` permite cambiar los permisos del fichero. Como primer parámetro se pasa el nombre del fichero y como segundo parámetro el modo que se desea establecer, en modo numérico. En modo octal, tenemos esta tabla de permisos:

Valor numérico	Permiso	Usuario
0o00400	Lectura	Propietario
0o00200	Escritura	Propietario
0o00100	Ejecución	Propietario
0o00040	Lectura	Grupo
0o00020	Escritura	Grupo
0o00010	Ejecución	Grupo
0o00004	Lectura	Otros
0o00002	Escritura	Otros

Valor numérico	Permiso	Usuario
0o00001	Ejecución	Otros

Por lo que si queremos que el fichero `logo.png` tenga permisos de lectura y escritura para su propietario y lectura para el grupo y otros, tendremos que ejecutar:

```
File.chmod "logo.png", 0o00644
```

El cambio de propietario se puede realizar a través de la función `File.chown/2`. Los parámetros de UID se dan en formato entero[9].

Hay una función que engloba todas las funciones del módulo ***File*** para la gestión de usuarios, grupos y permisos y permite realizar todas las modificaciones en una sola acción, tanto para la lectura: `File.stat/1`; como para la escritura: `File.write_stat/2`. Vamos a verlas un poco más en detalle:

`File.stat/1-2`

Permite leer las propiedades de un fichero retornando un registro en el que aparecen datos como la fecha y hora de creación, fecha y hora de modificación y fecha y hora del último acceso, además de los permisos, tipo de fichero y tamaño del mismo. Por ejemplo:

```
iex> File.stat "logo.png", :size
{:ok,
 %File.Stat{
   access: :read_write,
   atime: {{2012, 7, 18}, {14, 23, 1}},
   ctime: {{2012, 7, 18}, {14, 23, 1}},
   gid: 1000,
   inode: 11150880,
   links: 1,
   major_device: 2049,
   minor_device: 0,
   mode: 33188,
   mtime: {{2012, 7, 18}, {14, 23, 1}},
   size: 1718,
   type: :regular,
   uid: 1000
 }}
```

`File.write_stat/2`

Permite modificar cualquiera de los datos del fichero, para ello, se debe de especificar, como segundo parámetro, una estructura de tipo ***%File.Stat{}*** y rellenarlo con los datos del fichero que deseemos modificar.

[9] En los sistemas de tipo Unix este dato se puede ver en `/etc/passwd` donde hay una correspondencia entre el nombre del usuario y su UID o con la ejecución de ***ls -ln***.

4. Gestión de Directorios

Hasta el momento hemos visto como trabajar con ficheros, su contenido ya sea de tipo texto o de tipo binario, así como la gestión propia de los ficheros (copia, renombrado, eliminación, ...), ahora vamos a tratar la gestión de los directorios.

Los directorios nos permiten organizar nuestros ficheros de una forma más categorizada. Para sistemas que trabajan con miles de ficheros esto no es una opción sino una necesidad. En el terreno informático no hay recursos infinitos e incluso el número de ficheros que pueden albergarse en un directorio está limitado[10].

Como la gestión de los directorios, e incluso la de los ficheros se puede realizar desde un programa, un sistema que cree muchos ficheros puede particionar estos en directorios y subdirectorios, de modo que el acceso a cada directorio sea más rápido que en el caso de tener un directorio con miles de ficheros.

Veremos a continuación las funciones relativas a la gestión de directorios bajo los conceptos en los que se emplean.

4.1. Directorio de Trabajo

Las rutas que indicamos para los nombres de ficheros las podemos indicar de forma absoluta o relativa tanto para su apertura como para su gestión. La ruta absoluta nos indica dónde se encuentra un fichero mientras que la ruta relativa se basa en la ruta activa en la que se esté trabajando.

Elixir establece una ruta de trabajo que puede ir cambiando a través de llamadas específicas al sistema. La ruta de trabajo podemos extraerla con `File.cwd/0` o `File.cwd!/0`. Cualquier referencia a fichero que hagamos de forma relativa será siempre relativa a esta ruta.

Podemos cambiar la ruta de trabajo mediante la función `File.cd/1`, donde especificamos cuál será la nueva ruta de trabajo. Esto afectará a todas las rutas relativas que se empleen a partir del cambio.

Es un método bastante frecuente el cambiar la ruta para la ejecución de un código específico y volver inmediatamente a la ruta anterior, algo como esto:

```
dir = File.cwd
File.cd "/miruta"
```

[10] Hay sistemas de ficheros que establecen este límite a 1024 y otros que permiten miles o millones de ficheros por directorio, pero esto no es nada aconsejable. El tratamiento y gestión del propio directorio o de los propios ficheros se ralentiza si el número de ficheros que contiene es muy elevado.

```
%% ejecuta_codigo...
File.cd dir
```

Este es un truco que podemos emplear para ejecutar nuestro código en una ruta nueva y volver a la ruta anterior al finalizar.

4.2. Creación y Eliminación de Directorios

Una de las acciones básicas con los directorios es la de su creación y eliminación. Comenzaremos con el primer caso, la creación. La creación se puede indicar de forma absoluta o relativa. Un ejemplo:

```
File.mkdir "logos"
```

En sistemas como la ***shell*** de los sistemas tipo Unix, se permite realizar el comando:

```
mkdir -p /miruta/nuevo1/nuevo2/midir
```

Con lo que no solo se crea un directorio sino todos los necesarios hasta llegar al último indicado en la ruta pasada como parámetro. Esto no lo realiza `File.mkdir/1`. Esta función debe de recibir una ruta existente y creará el último directorio que se indique en la ruta, siempre que no exista ya.

Para obtener la creación de toda la ruta como el comando de consola podemos emplear la función `File.make_p/1`. Podemos ver un ejemplo:

```
iex> File.mkdir "/tmp/prueba/dir1"
{:error, :enoent}
iex> File.mkdir_p "/tmp/prueba/dir1/"
:ok
```

Partimos de que `/tmp/prueba` no existe. Por este motivo la función `File.mkdir/1` no puede crear el directorio final ***dir1***. La función `File.mkdir_p/1` crea ambos directorios. Primero crea el directorio ***prueba*** y después el directorio ***dir1*** dentro de ***prueba***.

Para eliminar un directorio junto con todos sus subdirectorios podemos emplear la función `File.rm_rf/1`. Esta función actúa como el comando de línea de comandos ***rm -rf*** eliminando de forma recursiva y forzosa la ruta pasada como parámetro.

4.3. ¿Es un fichero?

Para emplear en las guardas al igual que disponemos de las funciones `Kernel.is_list/1` podemos hacer uso de las funciones `File.dir?/1` o `File.regular?/1`.

Esto nos permite realizar funciones que nos permitan realizar un proceso previo de validación, por ejemplo, en caso de las configuraciones en las que se nos proporciona una ruta:

```
def temp_dir_config(dir) when not File.dir?(dir) do
  File.mkdir_p! dir
end
def temp_dir_config(_dir), do: :ok
```

Igualmente podemos emplear las funciones `File.exists?/1` (si el fichero o la ruta existe) para agregar más semántica o funcionalidad al código.

4.4. Contenido de los Directorios

Hay momentos en los que queremos tener un listado de todos los ficheros que se encuentran dentro de un directorio, para realizar un listado dentro del programa ya sea para tener un control del número de ficheros que se van generando, para realizar búsquedas o por cualquier otro motivo.

Una forma rápida de obtener los ficheros que queremos es emplear la función `File.ls/1`, lo cual nos retorna una lista de todos los nombres de ficheros que se encuentran en la ruta pasada como parámetro:

```
iex> File.ls "/home"
{:ok, ["bombadil", "bayadeoro"]}
```

Otra forma de obtener los ficheros que nos interesan que podemos emplear la función `Path.wildcard/1`. Esta función nos permite no solo poner una ruta sino además emplear los comodines para obtener los ficheros que concuerden:

```
iex> Path.wildcard "*.png"
["logo.png"]
```

Con estos listados, a través de funciones sobre listas como `Enum.map/2`, podemos realizar un procesado individualizado de los ficheros que nos retornen las funciones. Por ejemplo, si queremos extraer, además del nombre del fichero el tamaño y mostrarlo:

```
iex> Path.wildcard("*.png") |>
...> Enum.map(fn file ->
...>            %{size: size} = File.stat!(file)
...>            {file, size}
...>          end)
{"logo.png", 1718}
```

Esto nos abre una cantidad de posibilidades para obtener información de los ficheros albergados en un directorio, o incluso para poder recorrer directorios a través de funciones recursivas, en las que poder emplear las guardas vistas referentes a los ficheros.

Capítulo 9. Comunicaciones y Servidores

La mayoría de conversaciones son simples monólogos desarrollados en presencia de un testigo.
—Margaret Miller

Uno de los principales cometidos de un servidor, es establecer puertos de comunicación para recibir conexiones entrantes. La comunicación se establece a varios niveles, empleando en cada uno de los niveles un protocolo específico para la comunicación. En este capítulo nos centraremos en el protocolo IP, TCP y UDP para la pila de conexiones más popular: TCP/IP.

Nota

El material de este capítulo ha sido traído de Erlang/OTP Volumen I: Un Mundo Concurrente[1] pero adaptado para Elixir.

1. Conceptos básicos de Redes

Cuando se establece una conexión, la información que se percibe en el más alto nivel (o nivel de aplicación), es una representación que se ha ido resolviendo de un sistema de empaquetado anterior, encargado de agregar información sobre el paquete y enviarlo a través de la red.

Las capas que se distinguen en el envío de información de un punto a otro, en el modelo de Internet denominado TCP/IP, son:

Nivel físico

En este nivel se encuentran las conexiones físicas y sus protocolos específicos, según la tecnología en uso: Ethernet, 802.11, Fibre Channel, etc. A través de los ***drivers*** (o módulos del kernel) estos protocolos son transparentes para las aplicaciones. El nivel físico siempre se emplea punto a punto, cada máquina se conecta a través de un cable o de forma inalámbrica con otra y establecen una comunicación uno a uno.

Nivel de red

Es el nivel en el que se establece la base de la red, la identificación de los sistemas y el transporte hacia los mismos. En este nivel y en el

[1] https://books.altenwald.com/book/erlang-i

alcance que nos hemos propuesto para este libro, solo nos importa el protocolo IP. Este protocolo proporciona una dirección dentro de una red y permite establecer una comunicación a través de diversos dispositivos hasta encontrar la dirección de la máquina que debe recibir el mensaje.

Nivel de transporte

Este nivel es el que establece la forma de conexión entre las máquinas, la forma en la que se envían y trocean los paquetes para que lleguen a su destino y los acuses de recibo para asegurar que el paquete es recibido correctamente. En este nivel veremos dos protocolos: TCP y UDP.

Nivel de aplicación

Este es el nivel más alto que podemos encontrar en comunicación. Aquí se definen y usan protocolos como: HTTP, FTP, SMTP, POP3, IMAP, etc.

1.1. Direcciones IP

Una dirección IP se representa mediante un número binario de 32 bits (según IPv4). Las direcciones IP se pueden expresar como números de notación decimal: se dividen los 32 bits de la dirección en cuatro octetos. El valor decimal de cada octeto puede estar entre 0 y 255[2].

En la expresión de direcciones IPv4 en decimal se separa cada octeto por un punto. Cada uno de estos octetos puede estar comprendido entre 0 y 255, salvo algunas excepciones. Los ceros iniciales, si los hubiera, se pueden obviar. Ejemplo de representación de dirección IP: 164.12.123.65

Importante

Las direcciones IP en Elixir se emplean a través de un formato de tupla formada por cuatro elementos enteros. Esta forma es en la que generalmente trabaja el módulo ***inet*** que es el que se encarga de las comunicaciones, tanto para conexiones cliente, como para servidor:

```
{127, 0, 0, 1}
```

Este sería el formato de IP para 127.0.0.1. La función `:inet.ip/1` nos permite realizar la conversión del formato de texto al formato de tupla.

[2] El número binario de 8 bits más alto es 11111111 y esos bits, de derecha a izquierda, tienen valores decimales de 1, 2, 4, 8, 16, 32, 64 y 128, lo que suma 255 en total.

Hay tres clases de direcciones IP que una organización puede recibir de parte de la ***Internet Corporation for Assigned Names and Numbers*** (ICANN): clase A, clase B y clase C. En la actualidad, ICANN reserva las direcciones de clase A para los gobiernos de todo el mundo[3] y las direcciones de clase B para las medianas empresas. Se otorgan direcciones de clase C para todos los demás solicitantes. Cada clase de red permite una cantidad fija de equipos (hosts).

- En una red de clase A, se asigna el primer octeto para identificar la red, reservando los tres últimos octetos (24 bits) para que sean asignados a los equipos y a su vez reservando el primero (0) para la dirección de red y el último (255.255.255) para la dirección de multidifusión (broadcast), es decir, 16.777.214 equipos.

- En una red de clase B, se asignan los dos primeros octetos para identificar la red, reservando los dos octetos finales (16 bits) para que sean asignados a los equipos, de modo que la cantidad máxima de equipos (de nuevo menos dos) equivale a 65.534 equipos.

- En una red de clase C, se asignan los tres primeros octetos para identificar la red, reservando el octeto final (8 bits) para que sea asignado a los equipos, de modo que la cantidad máxima de equipos (menos dos) es de 254 equipos.

- Las direcciones 127.x.x.x se reservan para pruebas de retroalimentación. Se denomina dirección de bucle local o loopback y son muy útiles para probar elementos de red dentro de una misma máquina o para conectar dos elementos de forma local sin pasar por ninguna interfaz de red real.

Hay ciertas direcciones en cada clase de dirección IP que no están asignadas y que se denominan direcciones privadas. Las direcciones privadas pueden ser utilizadas por los equipos que usan traducción de dirección de red (NAT) para conectarse a una red pública o por los equipos que no se conectan a Internet. En una misma red no pueden existir dos direcciones iguales, pero sí se pueden repetir en dos redes privadas que no tengan conexión entre sí directamente. Las direcciones privadas son:

- Clase A: 10.0.0.0 a 10.255.255.255 (8 bits red, 24 bits equipos)

- Clase B: 172.16.0.0 a 172.31.255.255 (16 bits red, 16 bits equipos)

- Clase C: 192.168.0.0 a 192.168.255.255 (24 bits red, 8 bits equipos)

Muchas aplicaciones requieren conectividad dentro de una sola red, y no necesitan conectividad externa. En las redes de gran tamaño a menudo se usa TCP/IP. Por ejemplo, los bancos pueden utilizar TCP/

[3]Aunque en el pasado se le hayan otorgado a empresas de gran envergadura como, por ejemplo, Hewlett Packard.

IP para conectar los cajeros automáticos que no se conectan a la red pública, de manera que las direcciones privadas son ideales para ellos. Las direcciones privadas también se pueden utilizar en una red en la que no hay suficientes direcciones públicas disponibles.

Las direcciones privadas se pueden utilizar junto con un servidor de traducción de direcciones de red (NAT) para suministrar conectividad a todos los equipos de una red que tiene relativamente pocas direcciones públicas disponibles. Según lo acordado, cualquier tráfico que posea una dirección destino dentro de uno de los intervalos de direcciones privadas no se encaminará a través de Internet.

1.2. Puertos

Los puertos de comunicaciones son la base sobre la que se sustentan los protocolos de transporte TCP y UDP. Estos protocolos establecen conexiones salientes y entrantes en puertos denominados activos o pasivos respectivamente.

Los puertos se representan como números en rango de 16 bits, que pueden ir desde el 0 hasta el 65535. Los puertos por debajo del 1024 se denominan puertos privilegiados y en sistemas como los UNIX se requieren permisos de súper-usuario (o root) para poder emplear estos puertos[4].

Nota

En la mayoría de sistemas operativos existe un fichero de texto plano denominado `services` que contiene, formateado en dos columnas: el nombre de servicio y el puerto que emplea dicho servicio. La columna del puerto, además, viene formateada de forma que se indica el número, una barra inclinada y el tipo de transporte que se emplea:

```
http            80/tcp
http            80/udp
ftp-data        20/tcp
ftp             21/tcp
domain          53/tcp
domain          53/udp
```

Cada protocolo de transporte puede hacer uso del rango de numeración sin colisionar con ningún otro. TCP puede hacer uso del puerto 22 e igualmente UDP podría usar el mismo sin provocar colisión. La asignación de puertos es realizada por los protocolos de transporte, cada uno mantiene su propia numeración.

[4]Esto es debido también a que la mayoría de servicios que se prestan están en este rango, de modo que en un servidor un usuario sin privilegios no pueda establecer un puerto pasivo en el puerto 80 (dedicado a HTTP), 25 (de SMTP), 22 (de SSH) o 21 (de FTP) entre otros.

El puerto activo toma de la numeración de puertos un número y lo reserva para establecer la comunicación con el puerto pasivo. El número del puerto activo puede ser elegido o no dependiendo del protocolo de transporte.

La comunicación se identifica para el sistema operativo con el par de puertos activo/pasivo además de la dirección IP tanto de origen como de destino. Esto hace posible que a un puerto pasivo se pueda conectar más de un puerto activo.

Nota

En la jerga de los sistemas de comunicación existen algunas palabras clave que se emplean para determinar ciertos aspectos de la comunicación o los elementos que la componen. La palabra anglosajona ***socket*** (traducida como zócalo o conector) es la palabra que se suele emplear para indicar una conexión. Cuando se establece un puerto pasivo mediante TCP se suele decir que el servidor ***escucha*** mientras que si es mediante UDP se dice que el servidor está ***enlazado*** a ese puerto.

En el esquema cliente-servidor existe un servidor que se mantiene a la espera de una petición entrante y un cliente de forma activa realizas las peticiones al servidor. El servidor emplearía un puerto pasivo para establecer la comunicación mientras que el cliente usaría uno activo.

Como hemos ido comentando a lo largo de la sección hay dos protocolos para transporte que emplean los puertos de comunicación:

TCP

Es orientado a conexión. Requiere que el cliente formalice la conexión con el servidor y esta se mantiene hasta que una de las dos partes solicita la desconexión o durante un envío se produzca un tiempo de espera agotado.

UDP

UDP es un protocolo de datagramas. Un datagrama puede ser enviado hacia un servidor pero no se comprueba el estado de recepción. No se espera respuesta del mismo. El tratamiento de este tipo de paquetes carga menos la red y los sistemas operativos pero si la red no es fiable puede existir pérdida de información.

2. Servidor y Cliente UDP

En Elixir podemos hacer uso de un módulo llamado ***:gen_udp*** que nos permite establecer un puerto pasivo para mantenernos enlazados y

recibir paquetes entrantes. También permite emplear un puerto activo para el envío de un paquete hacia un servidor que se mantenga enlazado. Resumiendo, permite tanto programar servidores como clientes UDP.

En UDP tanto cliente como servidor deben enlazar un puerto. El servidor lo hará de forma pasiva para recibir mensajes. El cliente lo hará de forma activa para enviarlos y tener la posibilidad de recibir respuesta del servidor.

El módulo ***:gen_udp*** aprovecha las capacidades propias de los procesos enviando cada paquete recibido al proceso que solicitó el enlace con el puerto.

Hay tres funciones que emplearemos con mucha frecuencia en la construcción de servidores y clientes UDP: `:gen_udp.open/1-2`, `:gen_udp.send/4` y `:gen_udp.close/1`.

La función `:gen_udp.open/2` se presenta así:

```
:gen_udp.open(port, opts) -> {:ok, socket} | {:error, reason}
```

Puede recibir como segundo parámetro opciones que permiten variar la forma en la que establecer el enlace con el puerto. Las opciones son:

:list | :binary

Indica si se quiere recibir el paquete como una lista o un binario. El valor por defecto es ***:list***.

{:ip | :ifaddr, ip_address}

La dirección IP en la que enlazar el puerto.

:inet | :inet6

Emplea IPv4 o IPv6 para las conexiones. El valor por defecto es ***:inet***.

{:active, true | false | :once}

Indica si todos los mensajes recibidos por red serán pasados al proceso (***true***) o no (***false***) o si se hará solo la primera vez (***:once***). El valor por defecto es ***true***.

{:reuseaddr, true | false}

Permite reutilizar el puerto. No lo bloquea. Por defecto esta opción está deshabilitada.

Nota

Hay disponibles muchas opciones más a las que no entraremos ya que son conceptos más avanzados o muy específicos, con lo que salen del ámbito de este capítulo. Si desea más información sobre la función `:gen_udp.open/2` puede revisar la siguiente dirección:

http://www.erlang.org/doc/man/gen_udp.html#open-2

Pondremos en práctica lo aprendido. Vamos a escribir un módulo que enlace el puerto 2020 y cada paquete que reciba lo pase por pantalla:

```
defmodule UdpSrv do

  def start(port) do
    {:ok, spawn(__MODULE__, :init, [port])}
  end

  def init(port) do
    IO.puts "init #{port}"
    {:ok, socket} = :gen_udp.open(port)
    loop(socket)
  end

  def loop(socket) do
    IO.puts "waiting for input info... #{inspect socket}"
    receive do
      :stop ->
        :gen_udp.close(socket)
      {:udp, socket, ip, port, msg} ->
        IO.puts "recibido(#{inspect ip}): #{inspect msg}"
        :gen_udp.send socket, ip, port, "recibido"
        loop(socket)
    end
  end
end
```

El código se inicia mediante la función `UdpSrv.start/1` indicando un número entero correspondiente al puerto enlazado. Lanza un nuevo proceso que ejecuta la función `UdpSrv.init/1` encargada de abrir el puerto y pasar el control a la función `UdpSrv.loop/1` que se encarga de atender los paquetes que vayan llegando.

En el ejemplo hemos usado un registro formado por los cuatro datos que nos envía cada paquete UDP además del identificador:

```
{:udp, socket, ip, in_port_no, packet}
```

La definición de estos datos es la siguiente:

socket

El manejador retornado por la función `:gen_udp.open/1-2`.

ip

La dirección IP en formato de tupla.

in_port_no

El puerto origen del paquete recibido.

packet

El paquete recibido.

Podemos abrir una consola de Elixir y lanzar el servidor. Para saber el puerto donde se encuentra enlazado empleamos la función `:inet.i/0`. Esta función nos proporciona información sobre las comunicaciones:

```
iex(5)> UdpSrv.start(2020)
{:ok, #PID<0.100.0>}
iex(6)> :inet.i
Port  [...] Recv Sent Owner     Local Address  [...] State
 Type
26992 [...] 0    0    <0.100.0> *:2020         [...] IDLE
 DGRAM
:ok
```

La salida nos muestra el proceso (Owner) que tiene el puerto 2020 (Local) de tipo UDP (Type =:= DGRAM). Nos proporciona también otros datos estadísticos como los bytes recibidos (Recv) y enviados (Sent).

Podemos escribir estas líneas en la consola de Elixir para probar el código del servidor:

```
iex> c "examples/cap9/udp_srv.ex"
[UdpSrv]
iex> UdpSrv.start 2020
init 2020
waiting for input info... #Port<0.3329>
{:ok, #PID<0.96.0>}
iex> {:ok, socket} = :gen_udp.open(0)
{:ok, #Port<0.3348>}
iex> :gen_udp.send socket, {127,0,0,1}, 2020, "hola mundo!"
:ok
recibido({127, 0, 0, 1}): 'hola mundo!'
waiting for input info... #Port<0.3329>
```

La operación de conexión se ha realizado abriendo una comunicación con la función `:gen_udp.open/1` pasando como parámetro el puerto 0. El puerto 0 se usa para indicar que queremos que el sistema operativo seleccione un puerto automáticamente por nosotros. Podemos recurrir a ejecutar de nuevo `:inet.i/0` para ver las conexiones abiertas y las estadísticas:

```
iex> :inet.i
Port [...] Recv Sent Owner    Local Address [...] State Type
```

```
593  [...] 11   8    <0.34.0> *:2020        [...] IDLE  DGRAM
618  [...] 8    11   <0.38.0> *:33361       [...] IDLE  DGRAM
:ok
```

Ahora vemos dos líneas. La primera sigue siendo la del servidor y la segunda pertenece al cliente. Por los datos estadísticos vemos que la información que ha enviado el cliente (columna Sent) es la que ha recibido el servidor (columna Recv).

Comprobamos que hay también tráfico en la otra dirección. Se han transmitido desde el servidor al cliente 8 bytes, es decir, el texto "recibido". Podemos ejecutar ***flush*** para obtener ese dato. Para obtener la información desde código podemos variar la ejecución del cliente de esta forma:

```
iex> :gen_udp.close socket
:ok
iex> {:ok, socket} = :gen_udp.open(0, active: false)
{:ok, #Port<0.3372>}
iex> :gen_udp.send socket, {127,0,0,1}, 2020, "hola mundo!"
:ok
recibido({127, 0, 0, 1}): 'hola mundo!'
waiting for input info... #Port<0.3329>
iex> :gen_udp.recv(socket, 1024)
{:ok, {{127, 0, 0, 1}, 2020, 'recibido'}}
```

En la función `:gen_udp.open/2` hemos empleado el parámetro de opciones para indicar a ***:gen_udp*** que no envíe el paquete recibido al proceso. Al ejecutar `gen_udp:recv/2` es cuando se obtiene la información que llega del servidor.

3. Servidor y Cliente TCP

Para establecer comunicaciones TCP en Elixir disponemos del módulo ***:gen_tcp***. En TCP el servidor escucha de un puerto y se mantiene aceptando peticiones entrantes que quedan conectadas tras su aceptación. La dinámica está protocolizada de forma que un servidor establece una escucha en un puerto específico con posibilidad de envío de opciones a través de la función `:gen_tcp.listen/2` cuya definición es:

```
:gen_tcp.listen(port, opts) -> {:ok, socket} | {:error,
 reason}
```

Las opciones disponibles son iguales a las mostradas en la función `:gen_udp.open`. Son las siguientes:

:list | :binary

Indica si se quiere recibir el paquete como una lista o un binario. El valor por defecto es ***:list***.

{:ip | :ifaddr, ip_address}

La dirección IP en la que enlazar el puerto.

:inet | :inet6

Emplea IPv4 o IPv6 para las conexiones. El valor por defecto es ***:inet***.

{:active, true | false | :once}

Indica si todos los mensajes recibidos por red serán pasados al proceso (***true***) o no (***false***) o solo la primera vez (***:once***). El valor por defecto es ***true***.

{:reuseaddr, true | false}

Permite reutilizar el puerto. No lo bloquea. Por defecto esta opción está deshabilitada.

Nota

Hay disponibles muchas opciones más a las que no entraremos ya que son conceptos más avanzados o muy específicos, con lo que salen del ámbito de explicación de este capítulo. Si deseas más información sobre la función `:gen_tcp.listen/2` puedes revisar este enlace:

http://www.erlang.org/doc/man/gen_tcp.html#listen-2

Si ponemos en escucha el puerto 2020 y volvemos a listar el estado de la red podemos ver que el proceso (en Owner) pasa a escuhar (State =:= LISTEN) en el puerto 2020 (en Local) y para el tipo TCP (Type =:= STREAM):

```
iex> {:ok, socket} = :gen_tcp.listen(2020, reuseaddr: true)
{:ok, #Port<0.609>}
iex> :inet.i
Port [...] Recv Sent Owner    Local Address [...] State  Type
609  [...] 0    0    <0.32.0> *:2020        [...] LISTEN
 STREAM
:ok
```

El siguiente paso será mantener el proceso a la espera de una conexión entrante. Es el proceso que se llama aceptación y se realiza con la función `:gen_tcp.accept/1`. Si lo ejecutamos en la consola de Elixir el sistema se quedará bloqueado a la espera de una conexión entrante:

```
iex> {:ok, client_socket} = :gen_tcp.accept(socket)
```

La función emplea la variable ***socket*** creada en `:gen_tcp.listen/2` para esperar nuevas conexiones entrantes. Cuando llega una conexión

entrante la función `:gen_tcp.accept/1` acepta la conexión y finaliza su ejecución retornando otra variable ***client_socket***. Esta conexión se empleará para comunicarse con el cliente y obtener información del mismo.

En otra consola podemos realizar la conexión entrante. Emplearemos la función `:gen_tcp.connect/3` que tiene la siguiente sintaxis:

```
:gen_tcp.connect(address, port, opts) ->
    {:ok, socket} | {:error, reason}
```

Como parámetros se indica la dirección IP a la que conectarse (***address*** en formato tupla), el puerto al que conectarse (***port***) y las opciones para establecer la comunicación (***opts***). Las opciones son las mismas que se describieron para la función `:gen_tcp.listen/2`.

Abrimos otra consola y establecemos una comunicación local entre ambos puntos escribiendo lo siguiente:

```
iex> {:ok, socket} = :gen_tcp.connect({127,0,0,1}, 2020, [])
{:ok, #Port<0.599>}
iex> :inet.i
Port [...] Recv Sent [...] Local   Foreign State     Type
599  [...] 0    0    [...] *:38139 *:2020  CONNECTED STREAM
:ok
```

La nueva consola de Elixir solo tiene constancia de una conexión existente que está en estado conectada (State) y va del puerto local 38139 al puerto 2020.

Desde la nueva consola podemos enviar información al servidor a través de la función `:gen_tcp.send/2` la cual tiene la forma:

```
:gen_tcp.send(socket, packet) -> :ok | {:error, reason}
```

Como ***socket*** emplearemos el que nos retornó la función `:gen_tcp.connect/2`. El ***packet*** es la información que queremos enviar. El paquete permite formatos de lista de caracteres y lista binaria. Podemos ejecutarlo de la siguiente forma:

```
iex> :gen_tcp.send socket, "hola mundo!"
```

En el servidor no vemos de momento nada por consola. Si ejecutamos `IEx.Helpers.flush/0` aparecerán los mensajes recibidos al proceso de la consola[5].

[5]Recordemos que la consola es un proceso que se mantiene a la espera de recibir eventos. Todos los eventos que reciba la consola son interceptados por esta. Véase el Apéndice B, ***La línea de comandos*** para más información.

Si agregamos a las opciones de la función `:gen_tcp.listen/2` ***active: false*** el mensaje no es enviado al proceso sino que espera a que lo recibamos a través del uso de la función `:gen_tcp.recv/2`. Esta función tiene la siguiente sintaxis:

```
:gen_tcp.recv(socket, length) -> {:ok, packet} | {:error,
 reason}
```

El ***socket*** es el valor de retorno obtenido tras la ejecución de la función `:gen_tcp.accept/1`. El valor ***length*** es el tamaño máximo del paquete que esperamos recibir. En caso de que el paquete sea mayor que el tamaño una nueva ejecución de `:gen_tcp.recv/2` recogerá el siguiente trozo de información. En caso de indicar un tamaño cero se recibe todo el paquete sin limitación de tamaño.

Para establecer el diálogo entre servidor y cliente solo necesitamos emplear las funciones `:gen_tcp.send/2` y `:gen_tcp.recv/2` para realizar la comunicación bidireccional. En el momento en el que deseemos finalizar la comunicación por cualquiera de las dos partes emplearíamos la función `:gen_tcp.close/1`.

Importante

El ***socket*** que establecimos para escucha lo podemos cerrar igualmente con `:gen_tcp.close/1`. Conviene cerrar los puertos antes de finalizar la ejecución de los servidores para liberar los puertos empleados.

4. Servidor TCP Concurrente

La comunicación TCP se basa en la interconexión de dos puntos. En BEAM se conecta cada ***socket*** a un proceso para recibir información por lo que hasta que la conexión entre cliente y servidor no se cierre ningún otro cliente puede ser atendido por el servidor. Este problema no se presenta en comunicaciones UDP. En TCP cada conexión servidora genera un ***socket*** de conexión con el cliente específico.

Para que el proceso de servidor no permanezca bloqueado atendiendo la conexión del primer cliente que conecte generamos un nuevo proceso para atender esa petición entrante. De esta forma el proceso principal queda liberado para aceptar más peticiones y generar nuevos procesos a medida que vayan llegando nuevas peticiones de clientes.

Para que el nuevo ***socket*** generado sepa que tiene que enviar sus paquetes al nuevo proceso hay que emplear la función `:gen_tcp.controlling_process/2`, que tiene la forma:

```
:gen_tcp.controlling_process(socket, pid) -> :ok | {:error,
 reason}
```

Escribiremos un pequeño módulo para comprobar cómo funciona:

```
defmodule TcpSrv do
  use GenServer

  @accept_timeout 1_000

  def start(port) do
    GenServer.start __MODULE__, [port], name: __MODULE__
  end

  @impl true
  def init([port]) do
    opts = [reuseaddr: true, active: false]
    {:ok, socket} = :gen_tcp.listen(port, opts)
    GenServer.cast self(), :accept
    {:ok, socket}
  end

  @impl true
  def handle_cast(:accept, socket) do
    case :gen_tcp.accept(socket, @accept_timeout) do
      {:ok, sock_cli} ->
        task = Task.async(fn -> worker_loop sock_cli end)
        :gen_tcp.controlling_process sock_cli, task.pid
        :inet.setopts sock_cli, active: true
      {:error, :timeout} ->
        :ok
    end
    GenServer.cast self(), :accept
    {:noreply, socket}
  end

  @impl true
  def terminate(_reason, socket) do
    :gen_tcp.close(socket)
  end

  @impl true
  def handle_info(_msg, socket) do
    # ignore unformatted messages
    {:noreply, socket}
  end

  def worker_loop(socket) do
    receive do
      {:tcp, ^socket, msg} ->
        IO.puts "Recibido #{inspect self()}: #{inspect msg}"
        IO.puts "Espera 5 segundos y entonces responde"
        Process.sleep 5_000
        salida = "eco: #{msg}"
        :gen_tcp.send socket, salida
        worker_loop socket
      {:tcp_closed, ^socket} ->
        IO.puts "Finalizado."
      any ->
        IO.puts "Mensaje no reconocido: #{inspect any}"
```

```
    end
  end
end
```

La función `TcpSrv.start/1` se encarga de lanzar un servidor que se inicia con la ejecución de la función `TcpSrv.init/1`. El cometido de esta función inicializadora es establecer la escucha en un puerto TCP. Realizamos el lanzamiento periódico del mensaje ***:accept*** que es interceptado por `TcpSrv.handle_cast/2`. En esta función el sistema espera un segundo por la conexión entrante y vuelve a lanzar de nuevo el evento.

En caso de recibir una conexión entrante se lanza una tarea con la función `worker_loop`. Esta función integra el protocolo a nivel de aplicación para interactuar con el cliente. En nuestro ejemplo esperamos recibir un mensaje y lo retornamos precedido de la palabra ***Eco***.

El siguiente código es un ejemplo para probar el servidor. Para facilitar las pruebas en consola vamos a crear otro módulo:

```
defmodule TcpCli do
  def send(port, msg) do
    opts = [active: true]
    {:ok, socket} = :gen_tcp.connect({127,0,0,1}, port, opts)
    :gen_tcp.send(socket, msg)
    receive do
      {:tcp, ^socket, msg_srv} ->
        IO.puts "Retornado #{inspect self()}: #{msg_srv}"
      any ->
        IO.puts "Mensaje no reconocido: #{inspect any}"
    end
    :gen_tcp.close socket
  end
end
```

La función `TcpCli.send/2` permite conectarse a un puerto local[6], enviar un mensaje y esperar por el retorno antes de finalizar la comunicación. Podemos ver su funcionamiento:

```
iex> c "examples/cap9/tcp_srv.ex"
[TcpSrv]
iex> c "examples/cap9/tcp_cli.ex"
[TcpCli]
iex> {:ok, server} = TcpSrv.start 2020
{:ok, #PID<0.96.0>}
iex> TcpCli.send 2020, "hola mundo!"
Recibido #PID<0.98.0>: 'hola mundo!'
Espera 5 segundos y entonces responde
Retornado #PID<0.84.0>: eco: hola mundo!
Finalizado.
```

[6]En este ejemplo no hemos empleado direcciones IP, por lo que se emplea por defecto la IP local o 127.0.0.1.

```
:ok
```

Hasta el momento todo funciona como esperábamos. No obstante, hemos agregado en el servidor un retraso de 5 segundos que nos ayudará a ver la concurrencia en ejecuciones múltiples de `TcpCli.send/2`. Lo podemos realizar con varias consolas o a través de un código como el siguiente:

```
defmodule TcpCliParallel do
  def concurrent_send(port, num \\ 5) do
    f = fn(i) -> TcpCli.send(port, "i=#{inspect i}") end
    1..num
    |> Task.async_stream(f, timeout: 10_000)
    |> Enum.to_list()
  end
end
```

Este código genera 10 procesos que ejecutan la función `TcpCli.send/2` enviando el mensaje ***i=#{i}***, siendo i el valor pasado por `Enum.each/2` a cada uno de los procesos:

```
iex> c "examples/cap9/tcp_cli_parallel.ex"
[TcpCliParallel]
iex> TcpCliParallel.concurrent_send 2020
Recibido #PID<0.113.0>: 'i=1'
Espera 5 segundos y entonces responde
Recibido #PID<0.114.0>: 'i=2'
Recibido #PID<0.116.0>: 'i=3'
Espera 5 segundos y entonces responde
Espera 5 segundos y entonces responde
Recibido #PID<0.118.0>: 'i=4'
Recibido #PID<0.117.0>: 'i=5'
Espera 5 segundos y entonces responde
Espera 5 segundos y entonces responde
Retornado #PID<0.109.0>: eco: i=1
Retornado #PID<0.110.0>: eco: i=2
Retornado #PID<0.111.0>: eco: i=3
Retornado #PID<0.115.0>: eco: i=5
Finalizado.
Retornado #PID<0.112.0>: eco: i=4
Finalizado.
Finalizado.
Finalizado.
Finalizado.
[ok: :ok, ok: :ok, ok: :ok, ok: :ok, ok: :ok]
```

La ejecución muestra como todos los procesos llegan a recibir el mensaje en el servidor y quedan esperando por el resultado 5 segundos después.

Hemos empleado para este ejemplo el módulo ***Task*** que veremos más adelante con más detalle en la Capítulo 10, ***Un vistazo a OTP*** del Sección1, “Ejecución Asíncrona con Task“ .

5. Ventajas de *:inet*

BEAM no solo dispone de funciones para manejar las comunicaciones a nivel transporte. El módulo ***:inet*** a través de la función `:inet.setopts/2` nos permite modificar cómo los paquetes son recibidos a través de TCP o UDP y enviados como mensaje al proceso ya procesados.

Según la documentación de ***:inet***[7] los formatos que procesa son: CORBA, ASN-1, SunRPC, FastCGI, Line, TPKT y HTTP.

Nota

La decodificación la realiza únicamente a nivel de recepción, el envío deberemos de componerlo nosotros mismos y enviarlo con la función `:gen_tcp.send/2` o `:gen_udp.send/4`.

Para construir nuestro propio servidor HTTP y aprovechar la característica que nos provee ***:inet*** solo tendríamos que agregar la opción a la función `:gen_tcp.listen/2`. Vamos a verlo con un ejemplo:

```
iex> opts = [reuseaddr: true, active: true, packet: :http]
[reuseaddr: true, active: true, packet: :http]
iex> {:ok, socket} = :gen_tcp.listen(8080, opts)
{:ok, #Port<0.604>}
iex> {:ok, socket_cli} = :gen_tcp.accept(socket)
```

En este momento el sistema queda en espera de que llegue una petición. Como hemos levantado un puerto TCP y le hemos configurado las características de HTTP, vamos a abrir un navegador con la siguiente URL:

http://localhost:8080/

En la consola veremos que ya prosigue la ejecución:

```
{:ok, #Port<0.605>}
iex> flush
{:http, #Port<0.605>,
    {:http_request, :GET, {:abs_path, '/''}, {1, 1}}}
{:http, #Port<0.605>,
    {:http_header, 14, :Host, :undefined, 'localhost:8080'}}
[...]
{:http, #Port<0.605>, :http_eoh}
:ok
iex> msg = "HTTP/1.0 200 OK
...> Content-length: 1
...> Content-type: text/plain
...>
...> H"
```

[7] http://www.erlang.org/doc/man/inet.html#setopts-2

```
"HTTP/1.0 200 OK\nContent-length: 1\nContent-type: text/plain
\n\nH"
iex> :gen_tcp.send socket_cli, msg
:ok
```

Los mensajes recibidos por el sistema son tuplas que tienen como primer elemento ***:http***. Como en los casos de ***:tcp*** el segundo parámetro es ***socket***. Como tercer parámetro puede aparecer otra tupla cuyo primer parámetro es:

:http_request

Si se trata de la primera línea de petición. Esta tupla tendrá 4 campos: ***:http_request***, método HTTP (GET, POST, PUT o DELETE entre otros), URI y versión HTTP en forma de tupla de dos elementos. Un ejemplo:

```
{:http_request, :GET, {:abs_path, "/"}, {1, 1}}
```

:http_header

Las siguientes líneas a la petición son las líneas de cabecera. Que se estructuran en una tupla de 5 campos: ***:http_header***, bit de cabecera, nombre de la cabecera, valor reservado (***:undefined***) y valor de la cabecera.

:http_eoh

Este dato se transmite en forma de átomo. Indica que la recepción de cabeceras ha finalizado.

A continuación vemos un ejemplo completo. Presenta las peticiones recibidas por pantalla junto con su contenido:

```
defmodule HttpSrv do
  use GenServer

  @resp """
        HTTP/1.1 200 OK
        Content-Length: 2
        Content-Type: text/plain

        OK
        """
  @accept_timeout 1_000

  def start(port) do
    GenServer.start __MODULE__, [port], name: __MODULE__
  end

  @impl true
  def init([port]) do
```

```
    opts = [reuseaddr: true, active: :once, packet: :http]
    {:ok, socket} = :gen_tcp.listen(port, opts)
    GenServer.cast self(), :accept
    {:ok, socket}
  end

  @impl true
  def handle_cast(:accept, socket) do
    case :gen_tcp.accept(socket, @accept_timeout) do
      {:ok, sock_cli} ->
        IO.puts "conexión entrante: #{inspect sock_cli}"
        task = Task.async(fn -> worker_loop sock_cli end)
        :gen_tcp.controlling_process sock_cli, task.pid
        :inet.setopts sock_cli, active: true
        :inet.setopts socket, active: :once
      {:error, :timeout} ->
        :ok
    end
    GenServer.cast self(), :accept
    {:noreply, socket}
  end

  @impl true
  def terminate(_reason, socket) do
    :gen_tcp.close(socket)
  end

  @impl true
  def handle_info(_msg, socket) do
    # ignore unformatted messages
    {:noreply, socket}
  end

  def worker_loop(socket) do
    receive do
      {:http, ^socket, :http_eoh} ->
        IO.puts "Fin cabeceras (http_eoh)"
        :inet.setopts socket, [packet: :raw]
        :gen_tcp.send socket, @resp
        :gen_tcp.close socket
      {:http, ^socket, {:http_request, method, uri, _}} ->
        IO.puts "Recibido (http) #{inspect self()}: " <>
                "#{method} #{inspect uri}"
        worker_loop socket
      {:http, ^socket, {:http_header, _, name, _, _val}} ->
        IO.puts "Recibido (http) #{inspect self()}: " <>
                "header #{name}"
        worker_loop socket
      {:tcp_closed, ^socket} ->
        IO.puts "Finalizado"
        :gen_tcp.close socket
      any ->
        IO.puts "Mensaje no reconocido: #{inspect any}"
        :gen_tcp.close socket
    end
  end
end
```

El servidor es bastante simple ya que siempre retorna el mismo resultado. Si accedemos desde un navegador veremos en modo texto el mensaje *OK*.

En la función `HttpSrv.worker_loop/1` cuando se recibe ***:http_eoh*** se puede ver que se modifica el tipo de paquete para poder recibir el contenido. Además vemos que se diferencian bien los mensajes que se reciben de tipo ***HTTP*** de los que son de tipo ***TCP***.

Nota

Si empleamos el parámetro ***active: false*** para emplear la función `:gen_tcp.recv/2` en lugar de ***receive/1*** hay que tener presente que el retorno de la función `:gen_tcp.recv/2` será: ***{:ok, http_packet}***, mientras que el retorno de ***receive/1*** será: ***{:http, socket, http_packet}***.

Capítulo 10. Un vistazo a OTP

Funciones + Mensajes + Concurrencia = Erlang/OTP
—Joe Armstrong

En este capítulo vamos a revisar un poco del mundo OTP[1]. Este framework proviene de Erlang y contiene muchos elementos que facilitan el desarrollo de soluciones a nivel de backend para servidores.

Aunque OTP tiene unas dos décadas de antigüedad la revolución que ha introducido Elixir ha propiciado la aparición de otras estructuras también muy útiles y necesarias para la forma de programar que propone Elixir. Entre ellas podemos ver Agent, Registry o Task que se suman a las que provee Erlang como son Application, Supervisor y GenSever entre otras.

El capítulo lo he titulado OTP pero en realidad las soluciones propuestas por Elixir no están estrictamente dentro de OTP. Quizás podríamos decir que están relacionadas pero poco más. Además en Elixir se han introducido o cambiado algunos de los comportamientos no tan de base como GenEvent[2] que se marcó como desfasado en Elixir 1.5 por surgir algo mucho mejor como es GenStage[3] que aunque aún no se ha integrado en el código base de Elixir (y todo apunta a que no se hará), es una solución mucho más robusta, elegante y estable a la que proponía GenEvent.

También en la versión de Elixir 0.12.5 fue marcada como desfasada GenFsm[4]. Poco tiempo después incluso en Erlang/OTP 20 el comportamiento relacionado en Erlang ***gen_fsm*** fue marcado también como desfasado y eliminado en Erlang/OTP 21. En este caso Erlang/OTP lo hizo para introducir una mejor solución a través de ***gen_statem*** y para Elixir al igual que el caso anterior se desarrolló de forma externa GenStateMachine[5] como una forma de mantener la implementación de la máquina virtual en Elixir de alguna manera.

Como el estudio de todos los comportamientos nos llevaría un libro completo (tal y como demuestro en Erlang/OTP Volumen II: Las Bases de OTP[6]) vamos a ver principalmente los comportamientos y soluciones propuestas por Elixir y el comportamiento GenServer considerado la base para la construcción de soluciones en el Modelo Actor.

[1] OTP son las siglas de Open Telecom Platform (Plataforma de Telecomunicaciones Abierta).
[2] https://hexdocs.pm/elixir/1.5.0/GenEvent.html
[3] https://github.com/elixir-lang/gen_stage
[4] https://github.com/elixir-lang/elixir/commit/455eb4c4ace81ce60b347558f9419fe3c33d8bf7
[5] https://hexdocs.pm/gen_state_machine/GenStateMachine.html
[6] https://books.altenwald.com/book/erlang-ii

Nota

Los comportamientos no provistos de forma nativa por Elixir como GenStateMachine[7] o GenStage[8] no serán tratados en este libro por requerir de mucho más espacio para su desarrollo de forma adecuada.

1. Ejecución Asíncrona con Task

En esta sección vamos a adentrarnos en un módulo que facilita la generación de procesos para la ejecución en paralelo de un código específico.

Hasta ahora para ejecutar un código en un proceso local hemos empleado la función `Kernel.spawn/1`. Obtener sincronización o la información resultante de la ejecución del código lo conseguimos a través de enviar el PID del proceso padre al nuevo proceso y esperar una respuesta.

El módulo ***Task***[9] nos permite generar tareas a realizar por un proceso en paralelo. La ejecución de la función tendrá un resultado y este resultado es enviado de vuelta al proceso lanzador.

Veamos un ejemplo:

```
iex> task = Task.async(fn -> 1 + 1 end)
%Task{
  owner: #PID<0.84.0>,
  pid: #PID<0.220.0>,
  ref: #Reference<0.2138592675.769654785.257698>
}
iex> Task.await task
2
```

En el ejemplo podemos ver la ejecución asíncrona de una función a través de `Task.async/1` y cómo obtenemos su código resultante con `Task.await/1`.

A diferencia de emplear la forma aprendida anteriormente con `spawn/1` y `receive/1` el uso de ***Task*** nos aporta mucha más funcionalidad escribiendo mucho menos. En principio y por ser ambas funciones podemos emplear el operador pipe:

```
iex> Task.async(fn -> 1 + 1 end) |> Task.await()
```

[7] https://hexdocs.pm/gen_state_machine/GenStateMachine.html
[8] https://github.com/elixir-lang/gen_stage
[9] Traducimos del inglés la palabra Task como Tarea.

```
2
```

Además el módulo ***Task*** nos proporciona otras funciones útiles como `Task.yield/1-2` o `Task.shutdown/1-2`.

La función `Task.yield/1-2` nos permite esperar un resultado durante un tiempo determinado y dependiendo del retorno podemos saber si la tarea finalizó o si debemos volver a preguntar. Los posibles retornos son ***{:ok, resultado}***, ***nil*** o ***{:exit, razón}***. La primera sucede en el caso ideal de que la tarea termine y envíe de vuelta un valor. La segunda cuando aún no ha terminado y se ha cumplido el tiempo de espera. La última cuando el proceso ha muerto y obtenemos como segundo elemento de la tupla el motivo o la razón de la terminación. Como segundo parámetro indicamos el tiempo de espera. Por defecto son 5 segundos si omitimos el parámetro. Podemos realizar un pequeño código de ejemplo:

```
iex> task = Task.async Process, :sleep, [60_000]
%Task{
  owner: #PID<0.99.0>,
  pid: #PID<0.101.0>,
  ref: #Reference<0.1647150028.391380993.137883>
}
iex> Task.yield task, 5_000
nil
iex> Task.yield task, 60_000
{:ok, :ok}
```

De esta forma podemos preguntar a la tarea si finalizó, y esperar incluso durante un tiempo determinado y realizar otras tareas mientras tanto. También podemos forzar su finalización:

```
iex> task = Task.async Process, :sleep, [60_000]
%Task{
  owner: #PID<0.99.0>,
  pid: #PID<0.124.0>,
  ref: #Reference<0.1647150028.391380993.139029>
}
iex> Process.alive? task.pid
true
iex> Task.shutdown task
nil
iex> Process.alive? task.pid
false
```

Por último comentamos la forma de realizar el procesamiento paralelo de un conjunto de datos a través de la ejecución de una función. El procesamiento se lleva a cabo en paralelo y el retorno se puede obtener a modo de flujo de datos (***Stream***). Vamos a obtener la tabla de multiplicar del 5 de forma asíncrona:

```
iex> 1..10 |>
```

```
...> Task.async_stream(fn x -> x * 5 end) |>
...> Enum.map(fn {:ok, n} -> n end)
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

De esta forma obtenemos cada procesamiento individual.

Un código similar lo empleamos en un capítulo anterior en la Sección4, "Servidor TCP Concurrente" donde creábamos múltiples tareas de forma asíncrona esperando por el resultado de sus ejecuciones.

Sin embargo si algo sale mal nos quedaríamos sin el proceso remoto en ejecución ni el local porque están enlazados:

```
iex> Task.async fn() -> Process.sleep(1_000); 1 / 0 end
%Task{
  owner: #PID<0.99.0>,
  pid: #PID<0.167.0>,
  ref: #Reference<0.1647150028.391380993.140344>
}
** (EXIT from #PID<0.99.0>) shell process exited with reason:
 an exception was raised:
    ** (ArithmeticError) bad argument in arithmetic expression
        :erlang./(1, 0)
        (elixir) lib/task/supervised.ex:88:
 Task.Supervised.do_apply/2
        (elixir) lib/task/supervised.ex:38:
 Task.Supervised.reply/5
        (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3

Interactive Elixir (1.6.6) - press Ctrl+C to exit (type h()
 ENTER for help)

20:28:45.878 [error] Task #PID<0.167.0> started from
 #PID<0.99.0> terminating
** (ArithmeticError) bad argument in arithmetic expression
    :erlang./(1, 0)
    (elixir) lib/task/supervised.ex:88:
 Task.Supervised.do_apply/2
    (elixir) lib/task/supervised.ex:38:
 Task.Supervised.reply/5
    (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
Function: #Function<20.99386804/0 in :erl_eval.expr/5>
    Args: []
```

Para evitar esto tendríamos que desenlazarlos o emplear ***Task.Supervisor***. Este módulo nos permite crear un supervisor para los procesos y ejecutar de forma no enlazada las tareas:

```
iex> fun = fn() -> Process.sleep(1_000); 1 / 0 end
#Function<20.99386804/0 in :erl_eval.expr/5>
iex> self
#PID<0.169.0>
iex> Task.Supervisor.async_nolink supervisor, fun
%Task{
  owner: #PID<0.169.0>,
  pid: #PID<0.176.0>,
```

```
  ref: #Reference<0.1647150028.391380993.142765>
}
iex>
20:41:34.044 [error] Task #PID<0.176.0> started from
 #PID<0.169.0> terminating
** (ArithmeticError) bad argument in arithmetic expression
    :erlang./(1, 0)
    (elixir) lib/task/supervised.ex:88:
 Task.Supervised.do_apply/2
    (elixir) lib/task/supervised.ex:38:
 Task.Supervised.reply/5
    (stdlib) proc_lib.erl:247: :proc_lib.init_p_do_apply/3
Function: #Function<20.99386804/0 in :erl_eval.expr/5>
    Args: []

iex> self
#PID<0.169.0>
iex> Task.yield task
nil
iex> Process.alive? task.pid
false
```

Podemos ver cómo en esta ocasión el proceso de la tarea falla pero no afecta a la ejecución del proceso lanzador. El proceso lanzador sigue siendo el mismo y puede comprobar si funciona o no el proceso de la tarea.

Nota

Ten en cuenta en estos casos si empleamos la función `Task.await/1` como el proceso lanzado falló con esta función el fallo se propagará y también fallará el proceso lanzador. Sin embargo como hemos empleado en su lugar la función `Task.yield/1` solo obtenemos ***nil*** al no obtener nada sobre el proceso lanzado. Como en el ejemplo en estos casos conviene comprobar también si el proceso sigue vivo.

2. Servidores

Uno de los comportamientos más empleado en sistemas BEAM es el servidor genérico. El servidor genérico inicia con un proceso en un estado, recibe durante su vida tres tipos de mensajes: llamadas síncronas (call), llamadas asíncronas (cast) e información (info).

El servidor se mantiene en ejecución de forma constante y mantiene su estado aislado. El proceso es el único capaz de modificar su estado y puede hacerlo al recibir una llamada o información desde otros procesos o un evento de tiempo.

Nota

Además el servidor nos permite cambiar el código en caliente a través del mecanismo de cambio de código integrado a través de una retro-llamada que podemos implementar. Esto queda fuera del ámbito marcado para este libro pero es muy parecido a cómo se realiza en Erlang/OTP por lo que puede ser de ayuda leer los ejemplos y cómo se realiza en este lenguaje y sobre BEAM a través del libro Erlang/OTP Volumen II: Las Bases de OTP[10].

Pondremos un ejemplo básico de un servidor que almacena un conjunto de números en su estado, acepta una petición para obtener un número dado el tipo (media, máximo, mínimo y sumatorio) y otra petición asíncrona donde acepta un número para realizar las operaciones con él. El código es el siguiente:

```
defmodule Estadistica do
  use GenServer

  defmodule State do
    defstruct cuenta: 0,
              sumatorio: 0,
              media: 0,
              max: 0,
              min: nil
  end

  alias Estadistica.State

  def start_link do
    GenServer.start_link __MODULE__, [], name: __MODULE__
  end

  def stop do
    GenServer.stop __MODULE__
  end

  def get(name) do
    GenServer.call(__MODULE__, name)
  end

  def add(num) do
    GenServer.cast(__MODULE__, {:num, num})
  end

  @impl true
  def init([]) do
    {:ok, %State{}}
  end

  @impl true
  def handle_call(:sum, _from, state) do
    {:reply, state.sumatorio, state}
  end
```

[10] https://books.altenwald.com/book/erlang-ii/

```
  @impl true
  def handle_call(:avg, _from, state) do
    {:reply, state.media, state}
  end

  @impl true
  def handle_call(:max, _from, state) do
    {:reply, state.max, state}
  end

  @impl true
  def handle_call(:min, _from, state) do
    {:reply, state.min, state}
  end

  @impl true
  def handle_cast({:num, num}, state) do
    cuenta = state.cuenta + 1
    sumatorio = state.sumatorio + num
    media = (sumatorio + num) / cuenta
    state = %State{state | cuenta: cuenta,
                           sumatorio: sumatorio,
                           media: media,
                           max: max(state.max, num),
                           min: min(state.min, num)}
    {:noreply, state}
  end
end
```

Puedes ver cómo empleamos y usamos el módulo ***GenServer*** para extender el funcionamiento del servidor por defecto y realizamos las llamadas al servidor empleando ***GenServer*** para las funciones `Estadistica.start_link/0`, `Estadistica.stop/0`, `Estadistica.get/1` y `Estadistica.add/1`. El código empleado por el servidor está definido por las implementaciones y las retro-llamadas `Estadistica.init/1`, `Estadistica.handle_call/3` y `Estadistica.handle_cast/2`.

Tenemos una API[11] definida para utilizar nuestro código basado en cuatro funciones: `Estadistica.start_link/0` para iniciar el proceso del servidor, `Estadistica.stop/0` para detener el proceso, `Estadistica.get/1` para obtener la información especificada y `Estadistica.add/1` para agregar un nuevo número al estado del servidor.

El estado del servidor se actualiza cuando llega la petición de llamada asíncrona (`Estadistica.handle_cast/2`) y obtenemos información a través de llamadas síncronas. Hemos dejado sin implementar otras retro-llamadas como `handle_info/2` (para la información sin formato enviada al proceso del servidor con `Kernel.send/2`), `terminate/2` (para ser ejecutada cuando el proceso es detenido) o `code_change/3`

[11] Application Programming Interface o Interfaz de Programación de Aplicaciones es un término empleado para la especificación de funciones disponibles para su uso en otro código.

(ejecutada cuando el código del servidor es cambiado por una nueva versión).

Podemos lanzar el código escrito y probar algunas llamadas:

```
iex> c "examples/cap10/estadistica.ex"
[Estadistica, Estadistica.State]
iex> Estadistica.start_link
{:ok, #PID<0.93.0>}
iex> [:sum, :avg, :max, :min] |>
...> Enum.map(&Estadistica.get/1)
[0, 0, 0, nil]
iex> 1..10 |> Enum.each(&Estadistica.add/1)
:ok
iex> [:sum, :avg, :max, :min] |>
...> Enum.map(&Estadistica.get/1)
[55, 6.5, 10, 1]
```

Los servidores ofrecen la posibilidad de desarrollar código que interactúa con otros servidores y permite mantener información y recursos disponibles solo para el proceso del servidor. En otros proyectos de BEAM los servidores suelen emplearse para contener conexiones entrantes, una conexión por servidor y proceso. Mantener siempre la información del proceso en el estado facilita y simplifica el diseño y el código.

3. Agentes

Antes hemos visto el uso de los servidores como medida para mantener información y trabajar con ella a través de un código y unos datos ejecutados dentro de un proceso para garantizar su aislamiento. Además, en la Sección13, "Diccionario del Proceso" mostramos cómo almacenar información en el diccionario del proceso.

Estas medidas son bastante empleadas en BEAM pero existen más y Elixir provee las suyas propias a través de la implementación de sus propios módulos. En principio y partiendo del paradigma de los servidores podemos revisar la idea de agentes.

Un agente es un código encargado únicamente de interactuar entre un proceso que solicita información para ser guardada, actualizada o leída y los datos. Es agnóstico con respecto a la información almacenada. Es una forma rápida de generar un servidor con estado donde solo necesitamos acceso a ese estado y ninguna propiedad específica solo disponible en el servidor como la gestión de eventos entrantes al proceso que contiene los datos.

Cada acción que realicemos con el agente tomará una clausura para tratar la información y realizar la adecuación del estado tanto para guardarlo o modificarlo como para extraerlo.

3.1. ¿Cuándo usar un Agente y cuando un Servidor?

Los servidores son procesos con estado que mantenemos en ejecución para interactuar con ellos. La misión de los servidores es proveer una forma de interacción personalizada y en algunos casos ligada a los datos pero no en torno a estos datos. En ocasiones la necesidad no es mantener los datos compartidos sino también un recurso como un fichero o la ejecución de un código de forma exclusiva.

Algunos ejemplos son los manejadores de ficheros. Estos servidores mantienen el descriptor del fichero y son los únicos que pueden acceder al fichero abierto. Otro ejemplo puede ser un trabajador en un contenedor de trabajadores para limitar la realización de una tarea específica a un número de trabajadores. De esta forma cada proceso realiza de forma exclusiva la ejecución de un código y se limita a través del propio proceso.

El empleo del agente está más ligado al almacenamiento de los datos en sí. Por ejemplo si necesitamos almacenar información accesible a través del nombre del proceso y que no requiera de un procesamiento pesado y exclusivo. Almacenar información de caché, sesiones o temporal para un procesamiento genérico o ligero pueden ser algunos de los usos más cotidianos.

3.2. Creando y Usando Agentes

Vamos a crear un agente y jugar un poco con él para insertar datos y obtenerlos después. En principio la creación la realizaremos a través de la función `Agent.start_link/1`. Esta función acepta una clausura cuyo retorno será empleado como estado inicial del agente:

```
iex> {:ok, pid} = Agent.start_link fn -> 0 end
{:ok, #PID<0.89.0>}
iex> Agent.update pid, &(&1+1)
:ok
iex> Agent.get pid, &(&1)
1
iex> Agent.update pid, &(&1*5)
:ok
iex> Agent.get pid, &(&1)
5
iex> Agent.stop pid
:ok
```

Como vemos el código nos permite actualizar el valor almacenado y obtener el valor en cualquier momento. Obtener un valor cuando es considerado de caché o un dato estadístico y relativo puede

ser manejado de esta forma. Por ejemplo si tomamos los datos almacenados en el agente como estadísticos podemos actualizarlos con `Agent.update/2` y obtener la información estadística en cualquier momento a través de `Agent.get/2`. Sin problema.

3.3. Actualización segura

No obstante si el dato almacenado corresponde a un valor autoincrementado y debe ser único para proporcionarnos un valor incremental único la forma de conseguir esto es a través de `Agent.get_and_update/2`. Esta función nos permite obtener el valor del estado y modificarlo:

```
iex> {:ok, pid} = Agent.start_link fn -> 0 end
{:ok, #PID<0.89.0>}
iex> Agent.get_and_update(pid, &({&1, &1+1}))
0
iex> Agent.get_and_update(pid, &({&1, &1+1}))
1
iex> Agent.get_and_update(pid, &({&1, &1+1}))
2
iex> Agent.stop pid
:ok
```

De esta forma nuestro proceso es el único que puede conseguir el valor almacenado en el agente ya que en la misma llamada lo obtenemos y modificamos.

3.4. Acción de las Clausuras

Como hemos visto todas las funciones relacionadas con el estado del agente emplean una clausura. Esta clausura es ejecutada por el agente dentro de su proceso. Si la clausura realiza alguna acción muy pesada ese tiempo de procesamiento pesará en las otras peticiones hacia el agente para manejar su estado con nuevas clausuras.

Lo ideal para evitar cuellos de botella es simplificar al máximo posible las clausuras haciendo todos los procesamientos pesados fuera de la clausura siempre y cuando sea posible.

4. Registros

Los registros son muy parecidos a los agentes pero difieren en la forma de tratar la información tanto para su acceso y modificación como para su almacenamiento interno. Mientras que los agentes están basados en servidores para mantener la información interna los registros están basados en ETS fraccionadas para almacenar tanta información como sea posible y garantizar una buena velocidad de acceso a los datos.

El caso más característico de uso para los registros es como diccionario de procesos para emplear cuando se inician servidores. De hecho fue creado precisamente con esa intención. En la Sección6, "Soportando 2 Millones de Usuarios Conectados" comentamos cómo mejorando este elemento consiguieron obtener un registro de 2 millones de elementos accesible de forma rápida para mantener el diccionario de procesos con 2 millones de entradas y lecturas constantes.

Sin embargo el registro tiene sus limitaciones para poder dar ese nivel de potencia. Solo podemos registrar y desregistrar información. No es posible modificar información almacenada solo insertar y eliminar. Además todo dato registrado está ligado al proceso que lo registró. Si el proceso que realizó los registros muere la información registrada por este proceso es eliminada.

Esto limita los casos de uso como base de datos clave-valor pero incrementa sus posibilidades en otros aspectos como sistemas de chat o publicación/suscripción.

4.1. Creando y Usando un Registro

Vamos a ver cómo crear un registro además de agregar y obtener información de él. Las funciones que vamos a necesitar son `Registry.start_link/1` para crear el registro, `Registry.register/3` para registrar información y `Registry.lookup/2` para obtener esa información:

```
iex> self
#PID<0.84.0>
iex> {:ok, pid} = Registry.start_link(keys: :unique, name:
 Diccionario)
{:ok, #PID<0.86.0>}
iex> {:ok, _} = Registry.register(Diccionario, "hello",
 "hola")
{:ok, #PID<0.87.0>}
iex> {:ok, _} = Registry.register(Diccionario, "bye", "adiós")
{:ok, #PID<0.87.0>}
iex> Registry.count Diccionario
2
iex> Registry.lookup Diccionario, "hello"
[{#PID<0.84.0>, "hola"}]
```

La respuesta de `Registry.lookup/2` es siempre una lista con los elementos encontrados para la clave dada. Como iniciamos el registro diciendo que manejamos claves únicas (***keys: :unique***) solo será posible obtener un valor en caso de encontrar alguno:

```
iex> Registry.lookup Diccionario, "no"
[]
```

En caso de no encontrar ningún valor el retorno es una lista vacía.

Además el retorno se forma por una tupla que contiene como primer elemento el PID del proceso ligado a ese valor y como segundo elemento el dato almacenado mediante la llamada a la función `Registry.register/3`.

Por último la función `Registry.count/1` nos da información sobre el número de elementos almacenados en el registro.

4.2. Desregistro de información

La eliminación de la información registrada puede darse de dos formas diferentes. La primera la conseguimos empleamos la función `Registry.unregister/2`. La segunda es el efecto de que el proceso que mandó registrar el dato muera.

Podemos probar cómo los elementos desaparecen con este simple ejemplo:

```
iex> Registry.count Diccionario
2
iex> spawn fn -> Registry.register Diccionario, "good", "bien"
...>             IO.puts "count => #{Registry.count
 Diccionario}"
...>       end
count => 3
#PID<0.116.0>
iex> Registry.count Diccionario
2
iex> Registry.unregister Diccionario, "hello"
:ok
iex> Registry.unregister Diccionario, "bye"
:ok
iex> Registry.count Diccionario
0
```

Creamos un nuevo proceso donde se registra la clave ***good***. Comprobamos que el registro se ha incrementado y en ese momento tiene tres claves registradas y finaliza el proceso. Al finalizar el proceso el registro recibe el evento y elimina todas las claves registradas por ese proceso. Por eso cuando volvemos a ejecutar `Registry.count/1` obtenemos de nuevo el valor de 2 claves registrada únicamente.

Por último desregistramos manualmente las otras claves.

4.3. Filtrado de contenido

En caso de emplear claves múltiples podemos encontrarnos con muchos resultados cuando obtenemos una clave. Si queremos filtrar estos

resultados para obtener únicamente los pertinentes para nosotros en un momento dado podemos emplear la función `Registry.match/3` o `Registry.match/4`. Esta función nos permite filtrar el contenido para obtener únicamente los datos relevantes.

Supongamos que creamos un registro donde agregamos jugadores. Cada jugador será una tupla, el primer elemento será el nombre del jugador y el segundo la edad. Las claves donde se almacenan son las partidas que están disputando o donde han sido agregados:

```
iex> {:ok, juego} = Registry.start_link keys: :duplicate,
 name: Juego
{:ok, #PID<0.97.0>}
iex> Registry.register Juego, :lago, {"Gabriel", 16}
{:ok, #PID<0.98.0>}
iex> Registry.register Juego, :lago, {"Laura", 21}
{:ok, #PID<0.98.0>}
iex> Registry.register Juego, :lago, {"Miguel", 23}
{:ok, #PID<0.98.0>}
iex> Registry.register Juego, :lago, {"Ana", 18}
{:ok, #PID<0.98.0>}
iex> Registry.lookup Juego, :lago
[
  {#PID<0.84.0>, {"Gabriel", 16}},
  {#PID<0.84.0>, {"Laura", 21}},
  {#PID<0.84.0>, {"Miguel", 23}},
  {#PID<0.84.0>, {"Ana", 18}}
]
iex> Registry.match Juego, :lago, {:_, 18}
[{#PID<0.84.0>, {"Ana", 18}}]
```

Hemos creado un nuevo registro para nuestro juego. Agregamos jugadores a la partida ***:lago*** y vemos cómo podemos obtener los componentes con la función `Registry.lookup/2`. A través de una concordancia especial podemos conseguir los usuarios que tienen 18 años.

Podemos emplear `Registry.match/4` para agregar guardas y conseguir datos más específicos como todos los mayores de edad:

```
iex> Registry.match Juego, :lago, {:_, :"$1"}, [{:>=, :"$1",
 18}]
[
  {#PID<0.84.0>, {"Laura", 21}},
  {#PID<0.84.0>, {"Miguel", 23}},
  {#PID<0.84.0>, {"Ana", 18}}
]
```

De esta forma podemos obtener información específica de los usuarios y si cada dato lo agrega el proceso que controla al usuario en caso de desconexión la información puede actualizarse automáticamente en lugar de tener que preocuparnos de desregistrarla nosotros mismos.

Nota

El sistema de concordancia empleado para hacer las búsquedas de contenido se basa en la función `:ets.match/2`[12]. También se comentó más en detalle en el libro Erlang/OTP Volumen I: Un Mundo Concurrente[13].

4.4. Registro de Nombres

Cuando vimos los servidores en la Sección2, "Servidores" vimos cómo asignar un nombre al servidor. ***GenServer*** nos permite además poder emplear otro sistema como registro de nombres y podemos emplear Registry para eso como habíamos dicho en varias ocasiones.

Vamos a modificar el servidor de estadística que creamos en el capítulo anterior:

```
defmodule Estadistica do
  use GenServer

  defmodule State do
    defstruct cuenta: 0,
              sumatorio: 0,
              media: 0,
              max: 0,
              min: nil
  end

  alias Estadistica.State

  defp via(stats_name) do
    {:via, Registry, {Estadistica.Registry, stats_name}}
  end

  def start_link(stats_name) do
    GenServer.start_link __MODULE__, [], name: via(stats_name)
  end

  def stop(stats_name) do
    GenServer.stop via(stats_name)
  end

  def get(stats_name, name) do
    GenServer.call(via(stats_name), name)
  end

  def add(stats_name, num) do
    GenServer.cast(via(stats_name), {:num, num})
  end

  @impl true
  def init([]) do
    {:ok, %State{}}
```

[12] http://erlang.org/doc/man/ets.html#match-2
[13] https://books.altenwald.com/book/erlang-i

```
  end

  @impl true
  def handle_call(:sum, _from, state) do
    {:reply, state.sumatorio, state}
  end

  @impl true
  def handle_call(:avg, _from, state) do
    {:reply, state.media, state}
  end

  @impl true
  def handle_call(:max, _from, state) do
    {:reply, state.max, state}
  end

  @impl true
  def handle_call(:min, _from, state) do
    {:reply, state.min, state}
  end

  @impl true
  def handle_cast({:num, num}, state) do
    cuenta = state.cuenta + 1
    sumatorio = state.sumatorio + num
    media = (sumatorio + num) / cuenta
    state = %State{state | cuenta: cuenta,
                           sumatorio: sumatorio,
                           media: media,
                           max: max(state.max, num),
                           min: min(state.min, num)}
    {:noreply, state}
  end
end
```

Hemos agregando la función `Estadistica.via/1` para obtener la tupla necesaria y requerida por ***GenServer***. De esta forma el servidor puede encontrar el PID del proceso a través de la búsqueda de la clave asignada. Vamos a probarlo:

```
iex> Registry.start_link keys: :unique, name:
 Estadistica.Registry
{:ok, #PID<0.148.0>}
iex> c "examples/cap10/estadistica_registro.ex"
[Estadistica, Estadistica.State]
iex> Estadistica.start_link "altenwald.es"
{:ok, #PID<0.151.0>}
iex> Estadistica.add "altenwald.es", 100
:ok
iex> Estadistica.get "altenwald.es", :sum
100
```

La clave elegida es ***altenwald.es*** y a través de su uso en las funciones podemos acceder al proceso y modificar los datos. Esto puede ser útil cuando obtenemos llamadas HTTP y queremos apuntar estadísticas sobre el dominio basado en la cabecera ***Host*** provista por la llamada.

También puede emplearse con conexiones websocket donde cada conexión se equipara con un usuario dentro del sistema y se ha creado un sistema de chat donde pueden recibir mensajes en tiempo real. Veremos un poco más sobre cómo enviar mensajes específicos y simular un sistema de chat en las siguientes secciones.

4.5. Entregas (*dispatch*)

Existe una función en el registro llamada `Registry.dispatch/3` diseñada para iterar sobre el conjunto de datos registrados bajo la misma clave y realizar alguna acción sobre ellos. En nuestro ejemplo del juego imagina que queremos presentar la información de todos los jugadores de un juego por pantalla. Podemos hacerlo de esta forma:

```
iex> Registry.dispatch Juego, :lago,
...>   fn entries ->
...>     for {_, {nombre, edad}} <- entries do
...>       IO.puts """
...>               player =>
...>                   nombre -> #{nombre}
...>                   edad -> #{edad}
...>               """
...>     end
...>   end
player =>
    nombre -> Gabriel
    edad -> 16
player =>
    nombre -> Laura
    edad -> 21
player =>
    nombre -> Miguel
    edad -> 23
player =>
    nombre -> Ana
    edad -> 18
:ok
```

De la misma forma si el proceso registrador para cada usuario fuese un proceso que gestiona la comunicación con el usuario podríamos emplear esta función para poder enviar un mensaje a todos los participantes.

La función `Registry.dispatch/3` tiene una forma que podemos simular con el uso de `Registry.lookup/2` y `Enum.each/2` de esta forma:

```
iex> Registry.lookup(Juego, :lago) |>
...> Enum.each(fn {_, {nombre, edad}} ->
...>             IO.puts """
...>                     player =>
...>                         nombre -> #{nombre}
...>                         edad -> #{edad}
...>                     """
```

```
...>           end)
```

Las diferencias se aprecian con el uso de las particiones, lo veremos más adelante en la Sección4.7, "Particiones" .

4.6. Publicación/Suscripción

Siguiendo con la funcionalidad de `Registry.dispatch/3` vamos a probar otro ejemplo donde cada proceso se registra para recibir información publicada por otros procesos. En este caso crearemos el registro ***RSS*** de tipo duplicado y bajo la clave ***:blog*** agregaremos varios procesos que permanecerán a la espera de la recepción de algún evento:

```
iex> Registry.start_link keys: :duplicate, name: RSS
{:ok, #PID<0.229.0>}
iex> process_fun = fn ->
...>   Registry.register(RSS, :blog, [])
...>   receive do data ->
...>     IO.puts "received #{inspect self} " <>
...>             "=> #{inspect data}"
...>   end
...> end
#Function<20.99386804/0 in :erl_eval.expr/5>
iex> 1..10 |>
iex> Enum.each(fn _ -> spawn process_fun end)
:ok
iex> Registry.lookup RSS, :blog
[
  {#PID<0.249.0>, []},
  {#PID<0.250.0>, []},
  {#PID<0.251.0>, []},
  {#PID<0.252.0>, []},
  {#PID<0.254.0>, []},
  {#PID<0.255.0>, []},
  {#PID<0.256.0>, []},
  {#PID<0.253.0>, []},
  {#PID<0.257.0>, []},
  {#PID<0.258.0>, []}
]
iex> Registry.dispatch RSS, :blog,
...>   fn entries ->
...>     for {pid, []} <- entries do send(pid, "hey!") end
...>   end
received #PID<0.249.0> => "hey!"
received #PID<0.253.0> => "hey!"
received #PID<0.250.0> => "hey!"
received #PID<0.251.0> => "hey!"
received #PID<0.255.0> => "hey!"
received #PID<0.254.0> => "hey!"
received #PID<0.256.0> => "hey!"
received #PID<0.252.0> => "hey!"
received #PID<0.257.0> => "hey!"
:ok
received #PID<0.258.0> => "hey!"
iex> Registry.lookup RSS, :blog
[]
```

Podemos ver cómo cada proceso recibe el mensaje y lo imprime por pantalla. Finalizando tras la ejecución y desregistrándose de ***:blog*** en ***RSS***.

4.7. Particiones

Cuando el registro crece es buena idea contar con diferentes particiones. Las particiones fraccionan internamente el almacenamiento de las claves en diferentes tablas ETS y permiten acelerar las acciones de almacenamiento, eliminación y búsqueda.

Por defecto el número de particiones es 1. Es una buena práctica configurar el número de particiones acorde al número de programadores de tareas[14] disponibles en el sistema. A través de la función `Registry.start_link/1` podemos configurar esta opción:

```
Registry.start_link(
  keys: :duplicate,
  name: Juego,
  partitions: System.schedulers_online()
)
```

Podemos realizar acciones en paralelo sobre cada partición a través de `Registry.dispatch/4` e indicando como opción ***parallel: true***. Retomando la ejecución del juego podemos agregar la opción para paralelizar la ejecución de esta forma:

```
Registry.dispatch Juego, :lago,
  fn entries ->
    for {_, {nombre, edad}} <- entries do
      IO.puts """
              player =>
                  nombre -> #{nombre}
                  edad -> #{edad}
              """
    end
  end,
  parallel: true
```

De esta forma ejecutamos la función en paralelo en cada una de las particiones. En muestras de datos pequeñas la mejora de tiempo puede no ser apreciable pero para cantidades mayores de información será la diferencia entre obtener un resultado de forma rápida o generar cuellos de botella importantes en nuestro programa.

[14] Veremos la configuración de BEAM más adelante en el ApéndiceB, ***La línea de comandos***.

Capítulo 11. Aplicaciones y Supervisores

Las ideas son fáciles, implementarlas es lo difícil
—Guy Kawasaki

Vamos a implementar un pequeño juego: Conecta Cuatro™. En principio quiero implementar la idea como un servidor genérico para interactuar desde una consola con él. La escritura del código no será muy diferente de cómo lo hacíamos en capítulos anteriores con los servidores genéricos no obstante nos servirá para introducir el concepto de aplicación y supervisor además de la creación de un proyecto.

El servidor genérico encargado de llevar la lógica del juego se encargará de crear el tablero, inscribir a los dos jugadores (su ID de proceso) y aceptar los movimientos cuando les toque su turno. Además comprobaremos el estado del tablero tras cada movimiento para saber si tenemos un ganador o no.

Fijaremos el tamaño del tablero en 7 casillas horizontales y 6 verticales. Los colores de los jugadores serán rojo para el primer jugador y amarillo para el segundo. En principio solo podremos disputar un juego cada vez. Esto nos facilita la primera versión ligando el nombre del módulo al proceso del juego.

Un jugador gana cuando ha conseguido colocar 4 piezas seguidas en vertical, horizontal o diagonal. El juego también puede acabar en empate si todo el tablero está relleno y no tenemos ningún ganador.

Comenzaremos con la implementación rápida del servidor para pasar directamente a la gestión de la aplicación donde estará incluido este servidor genérico. De la aplicación nos interesa saber cómo se genera una aplicación, las posibilidades que tenemos para manejar nuestras aplicaciones y cómo iniciar y detenerlas.

En el diseño de la aplicación veremos también los supervisores. En este caso veremos únicamente un supervisor estático manejando un único trabajador. Veremos la dinámica del supervisor de reiniciar el proceso del juego para garantizarnos tener siempre un juego activo.

1. ¿Qué es una Aplicación?

En BEAM una aplicación es un conjunto de módulos con un nombre, descripción y versión además de otra información como dependencias

y configuración. Dentro de BEAM existen diferentes aplicaciones. Cada una de las aplicaciones se inicia a través de un comportamiento definido en un fichero de especificación y un módulo con retro-llamadas para completar la funcionalidad de inicio de la aplicación y ajustarla a nuestras necesidades.

La información de una aplicación se especifica normalmente en el fichero `mix.exs`. El fichero `mix.exs` tiene la siguiente forma:

```
defmodule Cuatro.Mixfile❶ do
  use Mix.Project ❷

  def project do ❸
    [
      app: :cuatro,
      version: "1.0.0",
      elixir: "~> 1.7.0"
    ]
  end

  def application do ❹
    [
      mod: {Cuatro.Application, []},
      extra_applications: [:logger, :runtime_tools],
      registered: [Cuatro.Juego]
    ]
  end
end
```

❶ El nombre de este módulo se forma a través del nombre de la aplicación agregando ***Mixfile***.

❷ Este módulo usará ***Mix.Project*** para ayudarle a generar todos los entresijos necesarios de la aplicación BEAM.

❸ Debemos proveer información del proyecto. El retorno de la función `project` proporciona esa lista de propiedades con toda la información. Veremos más adelante toda la información a proveer.

❹ La función `application/0` nos permite especificar información sobre la aplicación principal de nuestro proyecto.

Hemos visto en el fichero de configuración de nuestro proyecto dos bloques bien diferenciados. El primer bloque proporciona información general sobre el proyecto y su construcción. El segundo bloque proporciona información más concreta sobre la aplicación.

Las opciones disponibles para el proyecto son las siguientes:

app

El nombre de la aplicación. Debe ser un átomo con el nombre de la aplicación. Por convención en BEAM todas las aplicaciones se escriben en minúsculas. Esto está ligado a que en Erlang el formato por defecto de los átomos es en letras minúsculas. Es un parámetro obligatorio.

version

Especifica la versión de la aplicación principal y del proyecto en general. Es importante mantenerlo actualizado sobretodo para utilizarlo en lanzamientos progresivos para realizar despliegues continuos[1]. Es un parámetro obligatorio.

elixir

Información sobre la versión mínima y/o máxima de Elixir a emplear para construir el proyecto. La forma de especificar la versión la veremos más adelante. Este parámetro es obligatorio.

La función `application/0` también retorna una lista de propiedades con unos parámetros de configuración. Estos parámetros están más ligados a la aplicación principal y a su comportamiento. Los parámetros de configuración son:

mod

Módulo. Especifica una tupla con el primer elemento indicando el nombre de un módulo que debe implementar el comportamiento de aplicación y un segundo elemento con una lista de parámetros a pasar a la función de la aplicación.

extra_applications

Aplicaciones extra que serán incluidas dentro del proyecto. Normalmente dependencias propias de Elixir como la aplicación ***:logger*** o propias de Erlang como la aplicación ***:crypto***.

registered

Los nombres registrados provistos por la aplicación. No es obligatorio pero es de ayuda para detectar colisiones con otras aplicaciones. Es una lista de átomos con los nombres regisrados para los procesos que empleará la aplicación. El valor por defecto es una lista vacía ([]).

[1] En inglés se suele emplear el término ***Continuos Deployments***.

env

Una lista de propiedades con información para la configuración de la aplicación. En este punto podemos agregar la configuración por defecto para la aplicación. El valor por defecto es una lista vacía ([]).

start_phases

El inicio de una aplicación puede estructurarse en fases. Bajo esta configuración podemos especificar las fases como una lista de propiedades constituida por el nombre de la fase y los argumentos para esa fase. El valor por defecto es una lista vacía ([]).

included_applications

Una lista de átomos representando las aplicaciones incluidas dentro de nuestra aplicación. Es una forma de especificar dependencias con un comportamiento diferente. Mientras que las dependencias deben estar en ejecución antes de proceder al inicio de nuestra aplicación las aplicaciones incluidas se ejecutan justo tras nuestra aplicación. Lo veremos más adelante. El valor por defecto es una lista vacía ([]).

maxT

El tiempo máximo de ejecución para la aplicación expresado en milisegundos. El valor por defecto es ***:infinity***.

Importante

Es posible que en proyectos algo más antiguos de Elixir encuentres en `mix.exs` la configuración ***applications*** en lugar de ***extra_applications***. Antiguas versiones de **mix** necesitaban la declaración de las aplicaciones a incluir en el proyecto. Actualmente se autodetectan basándose en las dependencias. Aunque ***applications*** se mantiene por compatibilidad su uso está obsoleto y nada recomendado.

La función `application/0` no es obligatoria. En nuestro caso es necesaria para informar al sistema de lanzamiento de la existencia de la implementación del comportamiento y por tanto la posibilidad de iniciar el supervisor y nuestro servidor.

1.1. La Versión del Proyecto

Las versión es una cadena de texto y podemos emplear cualquier formato pero os recomiendo emplear Semantic Versioning 2[2] por ser

[2] https://semver.org/lang/es/

ampliamente usado tanto por las dependencias de Elixir y el propio Elixir como por la mayoría de software libre existente.

Se compone de tres números separados por puntos especificando:

- Primer número (o número mayor) para especificar un lanzamiento del proyecto completado. Un hito conseguido en la línea del proyecto o una refactorización grande.
- Segundo número (o número medio) para especificar la adición de características. Cada vez que una característica es agregada al proyecto podemos incrementar este número.
- Tercer número (o número menor) para especificar corrección de errores, documentación o cambios muy pequeños en el código.

Existen también sufijos para agregar más información sobre posibles bifurcaciones en el código original. Recomiendo la lectura en la página oficial para mayor información.

La versión es muy importante para el cambio en caliente de las aplicaciones. Os recomiendo incrementar la versión siempre al realizar un lanzamiento para así tener la posibilidad de emplear el cambio en caliente de la versión anterior.

1.2. ¿Qué versión de Elixir empleamos?

En el proyecto debemos indicar la versión o versiones de Elixir a emplear. El formato de las versiones nos permite elegir diferentes formas para indicar la versión. Podemos elegir una versión única. Por ejemplo si nos limitamos a la versión 1.7.3 podemos indicarla así:

```
{:elixir, "1.7.3"}
```

En el módulo ***Version***[3] podemos encontrar información de cómo se procesa la versión. Al principio del texto podemos agregar los comparadores: **>=**, **<=**, **>**, **==**. De esta forma podemos indicar:

```
# igual o superior a 1.4.5
">= 1.4.5"

# superior a 1.5
"> 1.5"

# inferior o igual a 1.5.3
"<= 1.5.3"
```

Además podemos indicar condiciones con ***and*** y ***or*** de modo que podamos realizar estas composiciones:

[3] https://hexdocs.pm/elixir/Version.html

```
# igual o superior a 1.4 pero inferior a 1.5.0
">= 1.4 and < 1.5.0"
```

Esta combinación es bastante común por lo que se regularizó a través de otro operador **~>**. Este operador equivale a la construcción vista antes pero con ciertas restricciones:

```
# igual o superior a 1.4.5 pero inferior a 1.5.0
"~> 1.4.5"

# igual o superior a 1.4.0 pero inferior a 2.0.0
"~> 1.4"
```

De esta forma podemos garantizar la compatibilidad con la versión actual y asegurar que una nueva versión no será empleada hasta que podamos probar que realmente funciona con nuestro código.

1.3. Entorno o Configuración

Existen dos fuentes para la configuración del entorno de una aplicación. La primera es la configuración ***env*** del fichero `mix.exs` vista anteriormente. Es la inicial y menos prioritaria. De ahí se toman todas las configuraciones y se cargan en el entorno. Después se revisa el fichero de configuración. De este fichero se toman todas las configuraciones y se cargan en el entorno sobrescribiendo las que colisionen con las ya existentes y provenientes de la especificación de la aplicación.

En nuestro código la configuración la tomamos a través del uso de la función `Application.get_env/2,3` donde el primer parámetro es el nombre de la aplicación, el segundo corresponde a la clave de la configuración a obtener y el tercero y opcional nos permite proveer un valor por defecto para en caso de no haber sido definida la clave obtenemos ese valor por defecto.

En nuestro ejemplo hemos empleado la configuración para almacenar el número de puerto del que escuchará el sistema y la familia de protocolo de Internet a emplear (***:inet*** para IPv4 o ***:inet6*** para IPv6). Esta configuración no es útil ahora pero lo será en el siguiente capítulo cuando integremos la interfaz web para acceder a nuestro juego.

Hemos definido la configuración en la especificación de la aplicación así si el fichero no proporciona estos valores, estos serán los valores empleados. Para realizar una prueba he creado un servicio en http://cuatro.altenwald.com donde el ejemplo (la segunda versión a ver en el siguiente capítulo) funciona con esta configuración:

```
use Mix.Config
```

```
config :cuatro, port: 80,
                family: :inet
```

Podemos ver que el fichero de configuración requiere el uso de ***Mix.Config*** para proporcionarnos la función `Mix.Config.config/2`. El módulo de configuración también dispone de la macro `Mix.Config.import_config/1` que en combinación con el entorno puede ayudarnos para separar la configuración en diferentes ficheros según el entorno. El archivo de configuración solo tendría que agregar:

```
import_config "#{Mix.env}.exs"
```

Al ser una macro en ese punto se agrega realmente el contenido del fichero a incluir y se trata todo como un solo fichero. Hablaremos en la Sección 5, "Entornos" del Capítulo 12, ***Ecosistema Elixir*** sobre los entornos.

Nota

La configuración realizada en este fichero será transformada en un fichero entendible por BEAM para el momento de ejecución. Ten presente este dato porque todo el código y las configuraciones proporcionadas solo serán ejecutadas en tiempo de compilación para generar la configuración final a emplear en tiempo de ejecución.

1.4. Inicio y Fases

El inicio de una aplicación va acompañada de una serie de comprobaciones. Al principio se comprueban las dependencias, los módulos que componen la aplicación, se carga la configuración, se cargan los módulos y se ejecuta el comportamiento de la aplicación.

La ejecución de las funciones sigue el siguiente orden, explicado a modo de código Elixir:

```
def start do
  MyApp.start(:normal, [])
  if phases = Application.spec(:kernel)[:start_phases] do
    phases
    |> Enum.each(fn({phase, args}) ->
                   MyApp.start_phase(phase, :normal, args)
                 end)
  end
end
```

De esta forma podemos emplear la retro-llamada `start_phase/3` para agrupar una serie de acciones y sincronizar el inicio.

Obviamente este no es el código que emplea ***Application*** para el inicio, pero nos da una idea de cómo procede.

Pongamos un ejemplo. Vamos a imaginar que tenemos un servicio web. Este servicio se conecta con una base de datos, inicia el servidor web, la conexión con el sistema de caché y realizamos una carga preliminar de datos útiles para acelerar la interacción con el usuario.

Aquí podemos iniciar el sistema y crear las fases: datos, carga y web. Siendo la primera la encargada de realizar las conexiones de base de datos y sistemas de caché. La fase de carga la encargada de realizar la carga de los datos preliminares de algún proveedor, ficheros o similar a los sistemas de caché, memoria o base de datos y por último levantamos o habilitamos el servidor web para comenzar a aceptar peticiones.

De este modo el inicio es ordenado y nos aseguramos de tener todo el sistema funcionando antes de que comiencen a producirse peticiones web de clientes.

1.5. Aplicaciones Incluidas y *Umbrella*

Las aplicaciones incluidas se inician justo después de las aplicaciones. Una aplicación incluida es dependiente de la principal al mismo tiempo que la principal depende de esta aplicación incluida. Se necesitan para completar el conjunto o el proyecto. Por ejemplo si tenemos un programa contable y creamos una aplicación llamada clientes. Nuestra aplicación principal llamada contabilidad tiene módulos y servicios empleados en clientes y la aplicación clientes tiene la necesidad de interactuar también con contabilidad, emplear módulos comunes o recursos compartidos.

El tipo de aplicaciones para ser agregadas dentro del proyecto suele ser del tipo ***umbrella***. Cuando creamos una aplicación ***umbrella*** realmente lo que hacemos es generar dentro del directorio `apps` un directorio por aplicación y mantenemos todos los recursos compartidos en el directorio base: el directorio de construcción `_build`, el directorio de dependencias `deps`, los ficheros de configuración y de bloqueo de dependencias instaladas (`mix.lock`).

Este fraccionamiento está aconsejado por los creadores de Elixir para mantener en aplicaciones separadas dentro del mismo proyecto las unidades funcionales semejantes ordenadas. Esto también propicia la inclusión de nuevas aplicaciones dentro de un proyecto como pequeños bloques interconectados.

Este tema es bastante extenso de tratar y bien merece su propio capítulo y un buen ejemplo que pueda mostrar toda la complejidad y posibilidades. Queda de momento fuera del objetivo de este libro de ofrecer una base clara y simple para comenzar a trabajar con Elixir. Pero puedes recurrir a lecturas como la siguiente (en inglés): Umbrella

projects[4]; y aunque es de Erlang, esta guía orienta también lo que son las aplicaciones incluidas para Elixir: Included Applications[5].

2. Estructura de una Aplicación

Como he comentado hasta el momento la aplicación se conforma de una estructura de directorios y ficheros. La herramienta **mix** nos permite generar esta estructura de forma sencilla. Para crear nuestro ejemplo podemos ejecutar lo siguiente:

```
$ mix new cuatro --sup
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/cuatro.ex
* creating lib/cuatro/application.ex
* creating test
* creating test/test_helper.exs
* creating test/cuatro_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd cuatro
    mix test

Run "mix help" for more commands.
```

Vemos cómo el comando nos ha creado la estructura básica donde encontramos los ficheros base ya comentados anteriormente además de otros ficheros útiles pero no indispensables como `.gitignore`.

Nota

El fichero `.formatter.exs` corresponde a una adición nueva de la versión 1.7 enfocada a formatear el código de forma automática. Este enfoque nos permite definir el estilo a emplear en nuestro código y al ejecutarlo todo nuestro código es revisado y cambiado acorde a esos parámetros.

Si abrimos el fichero `lib/cuatro/application.ex` veremos la plantilla básica propuesta por Elixir para la aplicación. Es la siguiente:

```
defmodule Cuatro.Application ❶ do
  # See https://hexdocs.pm/elixir/Application.html
```

[4] https://elixir-lang.org/getting-started/mix-otp/dependencies-and-umbrella-projects.html#umbrella-projects
[5] http://erlang.org/doc/design_principles/included_applications.html

```
  # for more information on OTP Applications
  @moduledoc false

  use Application ❷

  def start(_type, _args) do ❸
    # List all child processes to be supervised
    children = [ ❹
      # Starts a worker by calling:
      #   Cuatro.Worker.start_link(arg)
      # {Cuatro.Worker, arg},
    ]

    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, ❺
            name: Cuatro.Supervisor]
    Supervisor.start_link(children, opts) ❻
  end
end
```

❶ Por conveniencia se suele emplear el nombre de la aplicación como primera parte y ***Application*** como parte final. De esta forma es fácil localizar el módulo conteniendo las retro-llamadas para la aplicación.

❷ Este módulo empleará ***Application*** para implementar el código necesario para las retro-llamadas cuando la aplicación esté siendo iniciada o parada.

❸ La retro llamada `Cuatro.Application.start/2` es llamada al inicio de la aplicación. Suele recibir dos parámetros: el tipo de inicio y los parámetros para el inicio. El tipo de inicio suele ser ***:normal***. Solo cambia cuando empleamos facilidades de BEAM como la alta disponibilidad de aplicaciones.

❹ La aplicación normalmente levantará un supervisor y retornará el mismo valor que retorne la función `Supervisor.start_link/2`. La especificación de los procesos supervisados será realizada a través del primer parámetro de esta función y deberá ser una lista conteniendo una tupla con el nombre del servidor y los argumentos a pasar a su función `start_link`.

❺ Las opciones del supervisor las veremos más adelante en detalle.

❻ La función `Supervisor.start_link/2` es la última normalmente en la retro-llamada. Esta función nos permite

crear un supervisor para todos los procesos generados en nuestra aplicación. Este supervisor debe ser retornado para ser incluido al árbol de supervisión de BEAM.

La plantilla agrega algunos comentarios con enlaces por si queremos obtener más ayuda.

Para nuestro ejemplo vamos a modificar únicamente la lista de hijos. Agregaremos el nombre del módulo:

```
children = [Cuatro.Juego]
```

De esta forma el supervisor se encargará de levantar un proceso con el juego y cuando el proceso se detenga el supervisor se encargará de volver a levantarlo. Vamos a repasar antes cómo funciona el supervisor.

3. ¿Qué es un Supervisor?

A lo largo de este capítulo hemos ido comentando la necesidad de un supervisor para la aplicación y el árbol de supervisión de BEAM. La definición más exacta para supervisor es un proceso encargado de recibir información sobre el estado de una serie de procesos y la forma en la que son iniciados.

La acción básica del supervisor es volver a levantar el proceso en caso de finalización y siempre dependiendo del estado de finalización del proceso y de la estrategia del supervisor.

Como primer parámetro pasamos una lista de hijos a ser empleada por el supervisor para saber cómo iniciar los procesos hijos. Suele ser un mapa con 6 claves definidas dentro:

id

El identificador para el hijo. Este puede ser cualquier dato no duplicado que pueda ayudarnos a identificar el proceso supervisado cuando queramos acceder a él a través del supervisor. También puede ser de ayuda cuando veamos errores producidos por el proceso y notificados por el supervisor a través de las anotaciones (o logs).

start

Una tupla del tipo MFA (módulo, función y argumentos) donde se define la información necesaria para lanzar el proceso a través de la ejecución de esa tripleta.

restart

Un átomo que define la forma de un proceso de ser levantado o no. Los valores válidos son ***:permanent*** para levantar siempre el proceso, ***:temporary*** para no levantar nunca el proceso y ***:trasient*** para levantarlo únicamente si su salida no fue ***:normal***.

shutdown

Permite definir cómo actuar cuando se detiene un proceso. El supervisor envía una señal ***:shutdown*** y dependiendo del valor de este proceso: si especificamos ***:infinity*** el supervisor esperará de forma indefinida a que el proceso finalice. Si especificamos un número entero este será tomado como los milisegundos a esperar antes de proceder con la finalización forzosa del proceso. Si especificamos ***:brutal_kill*** entonces el proceso es finalizado directamente con la señal ***:kill***.

type

El tipo de proceso. Pueden ser dos: ***:worker*** para los procesos servidores y ***:supervisor*** para supervisar otros supervisores.

modules

No suele emplearse. Indica los módulos necesarios a comprobar y que estén cargados antes de proceder con el inicio del proceso. Si alguno de los módulos especificados en la lista no estuviese disponible el supervisor no iniciaría.

No obstante de entre estas opciones es común que finalmente solo se empleemos las dos primeras que son obligatorias y obviemos el resto. Para facilitar la definición Elixir nos proporciona dos formas abreviadas: tupla de dos elementos tal y como viene definida en los comentarios del ejemplo de la aplicación y la segunda forma es especificar tan solo el módulo. Esta última es la forma elegida para lanzar el proceso de nuestro juego.

Como segundo parámetro podemos especificar algunas opciones para el supervisor. En nuestro ejemplo levantamos un supervisor pasando las opciones: ***strategy*** y ***name***. Las opciones que podemos emplear son las siguientes:

strategy

La estrategia a emplear para levantar o no un proceso cuando recibimos el mensaje de finalización. Veremos un poco más adelante las posibles estrategias.

max_restarts

Cuando un proceso es reiniciado el supervisor guarda registro del incidente y en un tiempo determinado, si se producen más del número definido de reinicios el supervisor finaliza su ejecución deteniendo el resto de procesos en supervisión.

max_seconds

En combinación con ***max_restarts*** especifica el tiempo en el que estos reintentos son contados. Si definimos 5 reintentos en 2 segundos esto quiere decir que permitimos la caída de hasta 5 procesos en un tiempo de 2 segundos.

name

El nombre que daremos al proceso del supervisor.

Las posibles estrategias del supervisor definen el comportamiento ante la caída de procesos. Como sabemos pueden darse varios tipos de finalización de los procesos. Solo cuando un proceso finaliza con el mensaje ***:normal*** es considerado como un final apropiado. El resto de mensajes son tomados como erróneos.

Las estrategias disponibles para el supervisor son las siguientes:

one_for_one

Cada proceso es independiente. Si un proceso muere el supervisor actúa sobre él como deba hacerlo y deja el resto de procesos inalterados.

rest_for_one

Este es conocido como el efecto dominó. Teniendo en cuenta el orden de definición de los hijos si un proceso falla y debe ser reiniciado el supervisor actúa deteniendo todos los que le siguen en la lista y los vuelve a levantar en el mismo orden.

one_for_all

Uno para todos y todos para uno. Esta estrategia actúa sobre todos los procesos. Si uno es finalizado y debe ser reiniciado el supervisor detiene todos los demás procesos y los vuelve a levantar todos de nuevo.

Nuestro ejemplo de momento tiene tan solo un proceso para supervisar por lo que realmente todas las estrategias se comportarían de la misma forma. En el siguiente capítulo agregaremos más elementos a este supervisor.

4. Aplicación *cuatro*

Vamos a comenzar a comentar nuestra aplicación desde el fichero `mix.exs` donde vemos la especificación del proyecto y la aplicación:

```
defmodule Cuatro.Mixfile do
  use Mix.Project

  def project do
    [
      app: :cuatro,
      version: "1.0.0",
      elixir: "~> 1.7",
      elixirc_paths: ["lib"],
      start_permanent: true,
      deps: deps()
    ]
  end

  def application do
    [
      mod: {Cuatro.Application, []},
      extra_applications: [:logger, :runtime_tools]
    ]
  end

  defp deps do
    []
  end
end
```

El fichero es exactamente igual al visto anteriormente. Hemos definido nuestra aplicación ***cuatro*** y de momento no tenemos dependencias.

Lo siguiente a modificar es el fichero de la aplicación:

```
defmodule Cuatro.Application do
  use Application
  require Logger

  def start(_type, _args) do
    import Supervisor.Spec

    children = [
      worker(Cuatro.Juego, [])
    ]

    Logger.info "[app] iniciada aplicación"

    opts = [strategy: :one_for_one, name: Cuatro.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

A diferencia de la aplicación generada por defecto vemos una línea de anotación puesta para informarnos de cuándo la aplicación es iniciada y en este caso he empleado las facilidades del módulo ***Supervisor.Spec***

a través de la función `Supervisor.Spec.worker/2`. Esta función nos proporciona la tupla de configuración basada en los parámetros proporcionados. El primer parámetro es el módulo del proceso a supervisar y el segundo parámetro son los argumentos pasados a la función `Cuatro.Juego.start_link/0`. En nuestro caso no tenemos argumentos así que proporcionamos una lista vacía.

Si ejecutamos en consola este comando podemos ver la información generada:

```
iex> Supervisor.Spec.worker(Cuatro.Juego, [])
{Cuatro.Juego, {Cuatro.Juego, :start_link, []}, :permanent,
 5000, :worker,
 [Cuatro.Juego]}
```

Además podemos agregar un tercer parámetro a la función a modo de lista de propiedades para modificar alguno de los parámetros en caso de ser necesario.

El último fichero necesario es el servidor del juego. Este código es el que sigue:

```
defmodule Cuatro.Juego do
  use GenServer

  @vsn 1

  @moduledoc """
  El  juego de Conecta Cuatro es un juego donde dos
  jugadores deben insertar una pieza en una de las columnas.
  Esta pieza cae a modo de pila sobre las ya existentes o
  en la última posición y gana quien consigue conectar
  cuatro piezas de su mismo color.
  """

  defmodule Tablero do
    @moduledoc """
    Información del tablero. Nos da información del tamaño
    máximo del tablero, las columnas y las piezas ya
    insertadas así como de quién es el turno.
    """
    defstruct cols: [],
              turno: nil,
              jugadores: {nil, nil},
              num_jugadores: 0,
              ganador: nil
  end

  alias Cuatro.Juego.Tablero

  @jugador1 0
  @jugador2 1

  @max_x 7
  @max_y 6

  @doc "Inicia el juego"
```

```
  def start_link do
    GenServer.start_link __MODULE__, [], name: __MODULE__
  end

  def stop do
    GenServer.stop __MODULE__
  end

  @doc "Retorna información de quién es el siguiente"
  def quien_juega? do
    GenServer.call __MODULE__, :quien_juega
  end

  def quien_soy? do
    GenServer.call __MODULE__, :quien_soy
  end

  def quien_gana? do
    GenServer.call __MODULE__, :quien_gana
  end

  def inscribe_jugador do
    GenServer.call __MODULE__, {:jugador, self()}
  end

  def existe? do
    is_pid(Process.whereis(__MODULE__))
  end

  @doc "Inserta una ficha en el tablero"
  def inserta(col) when is_integer(col) and col >= 0 do
    GenServer.call __MODULE__, {:col, col}
  end

  def jugadores do
    GenServer.call __MODULE__, :jugadores
  end

  @doc "Muestra las columnas"
  def muestra() do
    GenServer.call __MODULE__, :muestra
  end

  @impl true
  def init([]) do
    cols = List.duplicate([], @max_x)
    {:ok, %Tablero{cols: cols,
                   turno: @jugador1}}
  end

  defp cambia_turno(@jugador1), do: @jugador2
  defp cambia_turno(@jugador2), do: @jugador1

  defp agrega_jugador(tablero, id) do
    jugadores = case tablero.jugadores do
      {nil, id2} -> {id, id2}
      {id2, nil} -> {id2, id}
    end
    num_jugadores = tablero.num_jugadores + 1
    %Tablero{tablero | jugadores: jugadores,
                       num_jugadores: num_jugadores}
  end
```

```
@impl true
def handle_info({:DOWN, _ref, :process, id, _reason},
                tablero) do
  num = tablero.num_jugadores - 1
  case tablero.jugadores do
    {^id, id2} ->
      {:noreply, %Tablero{tablero | jugadores: {nil, id2},
                                    num_jugadores: num}}
    {id2, ^id} ->
      {:noreply, %Tablero{tablero | jugadores: {id2, nil},
                                    num_jugadores: num}}
  end
end

@impl true
def handle_call(:quien_gana, _from, tablero) do
  {:reply, tablero.ganador, tablero}
end

def handle_call(:quien_soy, {pid, _} = from, tablero) do
  reply = case tablero.jugadores do
    {^pid, _} -> 0;
    {_, ^pid} -> 1
    {nil, nil} -> :esperando_jugadores
  end
  {:reply, reply, tablero}
end

@impl true
def handle_call(:jugadores, _from, tablero) do
  {:reply, Tuple.to_list(tablero.jugadores), tablero}
end

@impl true
def handle_call({:jugador, id}, _from, tablero) do
  Process.monitor id
  case tablero.jugadores do
    {_, ^id} -> {:reply, @jugador1, tablero}
    {^id, _} -> {:reply, @jugador2, tablero}
    {nil, _} ->
        {:reply, @jugador1, agrega_jugador(tablero, id)}
    {_, nil} ->
        {:reply, @jugador2, agrega_jugador(tablero, id)}
    _ ->
        {:reply, :partida_ocupada, tablero}
  end
end

@impl true
def handle_call(_msg, _from,
                %Tablero{num_jugadores: num} = tablero)
                  when num < 2 do
  {:reply, :esperando_jugadores, tablero}
end

@impl true
def handle_call(:quien_juega, _from, tablero) do
  {:reply, tablero.turno, tablero}
end

@impl true
```

```
  def handle_call({:col, ix}, {pid, _},
                  %Tablero{jugadores: {_, pid},
                           turno: @jugador2} = tablero)
                    when ix >= 0 and ix < @max_x do
    juega(ix, tablero)
  end

  @impl true
  def handle_call({:col, ix}, {pid, _},
                  %Tablero{jugadores: {pid, _},
                           turno: @jugador1} = tablero)
                    when ix >= 0 and ix < @max_x do
    juega(ix, tablero)
  end

  @impl true
  def handle_call({:col, _ix}, _from, tablero) do
    {:reply, :turno_de_otro, tablero}
  end

  @impl true
  def handle_call(:muestra, _from, tablero) do
    cols = tablero.cols
           |> Enum.map(&normaliza/1)
    {:reply, cols, tablero}
  end

  defp juega(ix, tablero) do
    col = Enum.at(tablero.cols, ix)
    if length(col) < @max_y do
      col = col ++ [tablero.turno]
      cols = List.replace_at(tablero.cols, ix, col)
      tablero = %Tablero{tablero | cols: cols}
      if gana?(tablero) do
        {:reply, {:gana, tablero.turno},
                 %Tablero{tablero | ganador: tablero.turno}}
      else
        if lleno?(tablero) do
          {:reply, :lleno, tablero}
        else
          turno = cambia_turno(tablero.turno)
          {:reply, :sigue, %Tablero{tablero | turno: turno}}
        end
      end
    else
      {:reply, :col_llena, tablero}
    end
  end

  defp normaliza(col) when length(col) == @max_y, do: col
  defp normaliza(col) when length(col) < @max_y do
    normaliza(col ++ [nil])
  end

  defp lleno?(tablero) do
    Enum.all? tablero.cols, &(length(&1) == @max_y)
  end

  defp gana?(tablero) do
    f = fn({inc_x, inc_y}, {pos_x, pos_y}) ->
      comprueba(tablero, pos_x, pos_y, inc_x, inc_y, 4)
    end
```

```
    paths = [{-1, -1}, {-1, 0}, {-1, 1},
             {0, -1}, {0, 1},
             {1, -1}, {1, 0}, {1, 1}]
    for(pos_x <- 0..(@max_x - 1),
        pos_y <- 0..(@max_y - 1),
        do: {pos_x, pos_y})
    |> Enum.any?(fn(pos) ->
                  Enum.any?(paths, &(f.(&1, pos)))
                end)
  end

  defp comprueba(_tablero, _pos_x, _pos_y,
                 _inc_x, _inc_y, 0), do: true
  defp comprueba(_tablero, pos_x, pos_y, _inc_x, _inc_y, _i)
                  when pos_x < 0 or
                       pos_y < 0 or
                       pos_x >= @max_x or
                       pos_y >= @max_y, do: false
  defp comprueba(tablero, pos_x, pos_y, inc_x, inc_y, i) do
    if coord(tablero, pos_x, pos_y) == tablero.turno do
      comprueba(tablero, pos_x + inc_x, pos_y + inc_y,
                inc_x, inc_y, i - 1)
    else
      false
    end
  end

  defp coord(tablero, x, y) do
    tablero.cols
    |> Enum.at(x)
    |> Enum.at(y)
  end
end
```

Se trata de un servidor genérico (***GenServer***) extendido para mantener la información de un juego. Acepta las inscripciones de jugadores a través del almacenamiento del identificador de proceso (PID) desde donde se inscribe y mantiene la información del turno, el tablero y el código para comprobar si las inserciones de fichas son correctas así como saber si algún usuario ganó o no al realizar un movimiento.

Con estos tres ficheros podemos arrancar nuestra aplicación. Podemos hacerlo fácilmente a través del comando **mix run** o abriendo una consola al mismo tiempo que lo iniciamos con el comando **iex -S mix run**. No obstante como suponemos que debe haber dos jugadores vamos a iniciar dos consolas. La primera:

```
$ iex --name cuatro1@localhost -S mix run
Erlang/OTP 21 [erts-10.1.3] [source] [64-bit] [smp:8:8]
 [ds:8:8:10] [async-threads:1] [hipe] [dtrace]

Compiling 3 files (.ex)
Generated cuatro app

17:21:19.599 [info]  [app] iniciada aplicación
Interactive Elixir (1.7.4) - press Ctrl+C to exit (type h()
 ENTER for help)
```

```
iex(cuatro1@localhost)>
```

Podemos ejecutar algunos comandos para ver el entorno, las aplicaciones cargadas y la configuración. Por ejemplo, para ver las aplicaciones cargadas:

```
iex(cuatro1@localhost)> Application.loaded_applications
[
  {:elixir, 'elixir', '1.7.4'},
  {:mix, 'mix', '1.7.4'},
  {:compiler, 'ERTS  CXC 138 10', '7.2.7'},
  {:logger, 'logger', '1.7.4'},
  {:cuatro, 'cuatro', '1.0.0'},
  {:kernel, 'ERTS  CXC 138 10', '6.1'},
  {:stdlib, 'ERTS  CXC 138 10', '3.6'},
  {:runtime_tools, 'RUNTIME_TOOLS', '1.13.1'},
  {:iex, 'iex', '1.7.4'}
]
```

De esta forma obtenemos un listado de las aplicaciones en ejecución. Entre ellas vemos nuestra aplicación ***:cuatro*** además de ***:logger*** y ***:runtime_tools*** agregadas en el fichero `mix.exs`.

Podemos obtener la configuración de nuestra aplicación:

```
iex(cuatro1@localhost)> Application.get_all_env :cuatro
[]
```

En este caso no tenemos nada que ver. No hay ninguna configuración aún en el fichero de configuración relativa a nuestra aplicación. Pero podemos ver que ***:logger*** sí que tiene configuración por defecto:

```
iex(cuatro1@localhost)> Application.get_all_env :logger
[
  translators: [{Logger.Translator, :translate}],
  compile_time_application: nil,
  truncate: 8096,
  backends: [:console],
  discard_threshold: 500,
  handle_sasl_reports: false,
  handle_otp_reports: true,
  utc_log: false,
  discard_threshold_for_error_logger: 500,
  translator_inspect_opts: [],
  console: [],
  compile_time_purge_matching: [],
  compile_time_purge_level: :debug,
  sync_threshold: 20,
  level: :debug
]
```

El último módulo necesario para facilitar el juego es el módulo ***Cuatro***. Este módulo con el nombre de la aplicación suele crearse para realizar

una interfaz simple para interactuar con esta aplicación desde otras o en nuestro caso para facilitar los comandos en la consola:

```
defmodule Cuatro do
  require Logger
  alias Cuatro.Juego

  @jugador1 0
  @jugador2 1

  @jugador1_simbolo IO.ANSI.format(["[", :red_background,
                                    "O", :black_background,
                                    "]"])
  @jugador2_simbolo IO.ANSI.format(["[", :yellow_background,
                                    :black, "O", :white,
                                    :black_background, "]"])
  @nojugado_simbolo IO.ANSI.format(["[ ]"])

  @jugador1_color "rojo"
  @jugador2_color "amarillo"

  @msg_ganador "¡¡¡ Ganaste !!!"
  @msg_perdedor "Perdiste :-("

  defp muestra_a_todos do
    Juego.jugadores()
    |> Enum.each(fn(jugador) ->
                   info = Process.info(jugador)
                   muestra(info[:group_leader])
                 end)
  end

  defp muestra_a(jugador, msg) do
    leader = Process.info(jugador)[:group_leader]
    IO.puts(leader, msg)
  end

  @doc """
       Realiza el movimiento y retorna el estado, si ha
       ganado muestra el tablero y detiene el juego.
       """
  def mueve(col) do
    case Juego.inserta(col) do
      {:gana, quien} ->
        [ganador, perdedor] = case quien do
          @jugador1 -> Juego.jugadores()
          @jugador2 -> Enum.reverse Juego.jugadores()
        end
        muestra_a_todos()
        muestra_a ganador, @msg_ganador
        muestra_a perdedor, @msg_perdedor
        Juego.stop
      :esperando_jugadores ->
        IO.puts "esperando jugadores, inscríbase!"
      :sigue ->
        muestra_a_todos()
      :turno_de_otro ->
        IO.puts "es el turno del otro jugador... espere"
        :ok
      :col_llena ->
        IO.puts "columna llena, pruebe otra"
```

```
        :ok
    end
  end

  defp color(@jugador1), do: @jugador1_color
  defp color(@jugador2), do: @jugador2_color

  def inscribe_jugador() do
    case Juego.inscribe_jugador() do
      :partida_ocupada ->
        IO.puts "error: partida ocupada"
      jugador ->
        IO.puts "jugando como #{color(jugador)}"
    end
  end

  defp simbolo(@jugador1), do: @jugador1_simbolo
  defp simbolo(@jugador2), do: @jugador2_simbolo
  defp simbolo(nil), do: @nojugado_simbolo

  defp traduce_simbolos(col) do
    col
    |> Enum.map(&simbolo/1)
    |> Enum.join()
  end

  defp traspone([[]|_]), do: []
  defp traspone(matriz) do
    [ Enum.map(matriz, &hd/1) |
      traspone(Enum.map(matriz, &tl/1)) ]
  end

  @doc "Muestra el tablero"
  def muestra(device \\ :stdio) do
    cols = Juego.muestra()
    imprime = &(IO.puts(device, &1))

    cols
    |> Enum.map(&Enum.reverse/1)
    |> traspone()
    |> Enum.map(&traduce_simbolos/1)
    |> Enum.join("\n")
    |> imprime.()

    for(i <- 0..(length(cols) - 1),
      do: if(i<10, do: " #{i} ", else: "#{i} "))
    |> Enum.join()
    |> imprime.()
  end

end
```

Vamos a facilitar el acceso a las funciones de ***Cuatro*** escribiendo lo siguiente:

```
iex(cuatro1@localhost)> import Cuatro
Cuatro
iex(cuatro1@localhost)> inscribe_jugador
jugando como rojo
:ok
```

Podemos ver cómo ahora no necesitamos escribir el módulo para acceder a sus funciones. Nos inscribimos para jugar y el juego nos asigna el color rojo. Veamos cómo podemos realizar una partida de prueba.

5. Probando el Juego

Necesitamos un segundo jugador. Vamos a conectarnos al mismo nodo. Esto lo haremos con la opción ***remsh*** de esta forma:

```
iex --remsh cuatro1@localhost --sname cuatro2@localhost
Erlang/OTP 21 [erts-10.1.3] [source] [64-bit] [smp:8:8]
 [ds:8:8:10] [async-threads:1] [hipe] [dtrace]

Interactive Elixir (1.7.4) - press Ctrl+C to exit (type h()
 ENTER for help)
iex(cuatro1@localhost)> import Cuatro
Cuatro
iex(cuatro1@localhost)> inscribe_jugador
jugando como amarillo
:ok
```

Ambas consolas están conectadas al mismo nodo como puedes ver en el símbolo del sistema y se han inscrito como jugadores. Vamos ahora a la primera consola, la del jugador rojo para comenzar a insertar fichas:

```
iex(cuatro1@localhost)> mueve 4
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][O][ ][ ]
 0  1  2  3  4  5  6
:ok
```

Hemos insertado la ficha en la columna 4. En ambas consolas se muestra el tablero con la pieza roja insertada en la columna correspondiente. Esto lo conseguimos gracias a la impresión hacia el líder. Puedes verlo en este pequeño trozo de código:

```
  defp muestra_a(jugador, msg) do
    leader = Process.info(jugador)[:group_leader]
    IO.puts(leader, msg)
  end
```

Obtenemos del proceso el identificador de proceso correspondiente al ***:group_leader*** que es el proceso encargado de enviar toda la salida que le llega a la consola correspondiente y a través del uso de `IO.puts/2` hacemos la impresión.

Podemos ver otro trozo de código donde empleamos algo similar para imprimir un mensaje diferente tanto para el ganador como para el perdedor:

```
{:gana, quien} ->
  [ganador, perdedor] = case quien do
    @jugador1 -> Juego.jugadores()
    @jugador2 -> Enum.reverse Juego.jugadores()
  end
  muestra_a_todos()
  muestra_a ganador, @msg_ganador
  muestra_a perdedor, @msg_perdedor
```

En el momento de producirse el fin del juego el proceso finaliza. Para poder comenzar otro juego debemos volver a inscribirnos.

Nota

Recuerda que si te equivocas y el proceso de algún jugador es finalizado el juego elimina a ese jugador y debes volver a inscribirte.

Capítulo 12. Ecosistema Elixir

El hombre debe configurar sus herramientas a su forma.
—Arthur Miller

En este capítulo comenzamos a escribir proyectos y emplearemos **mix**[1] como herramienta base para la construcción, compilación, liberación y empaquetado de nuestra aplicación.

En el capítulo anterior ya pudimos ver la definición del fichero `mix.exs` y usamos el comando **mix** para compilar y ejecutar el proyecto del juego Cuatro. En este capítulo vamos a ir un paso más allá y veremos las dependencias. Gracias a esto podremos implementar ***websockets*** y podremos jugar en un navegador web a nuestro juego.

El principal objetivo de este capítulo es mostrar las dinámicas de trabajo de un proyecto Elixir a través del uso de algunos comandos, dependencias y la realización de las acciones básicas para construir, lanzar y actualizar un proyecto.

1. El comando mix

En el capítulo de aplicaciones y supervisores nos adentramos en el uso del comando **mix** para la creación base del proyecto. También vimos cómo nos puede servir para arrancar nuestro proyecto pero hay muchas más acciones que puede realizar **mix**.

Empleando **mix help** podemos ver la salida de todas las opciones de que dispone por defecto. Pero al igual que Elixir este comando también es extensible. Hay muchas dependencias como ***ecto*** o ***phoenix*** que agregan sus propias tareas e incluso nosotros mismos podemos agregar otras tareas nuevas tanto a través de nuestro proyecto como a través de la configuración en el fichero `mix.exs`.

Hay además una acción especifica llamada ***do*** que nos permite enumerar una serie de acciones a realizar en una sola línea separadas por comas. Por ejemplo si queremos limpiar el proyecto y después compilarlo:

```
mix do clean, compile
```

Esto será equivalente a si escribimos primero el comando para limpiar y después el comando para compilar con la ventaja de detenerse en el paso que cause algún problema.

[1] La herramienta ***mix*** recibe ese nombre por su significado ***mezclar*** que es la acción básica cuando se crean elixires.

Algunos de los comandos básicos de mix son los siguientes:

app.start, run

Inicia la aplicación. Normalmente el nodo cuando inicia la operación finaliza porque la aplicación se ejecuta en segundo plano. Si queremos mantener la aplicación en ejecución debemos de ejecutar esta acción dentro de una consola o mediante el comando:

```
iex -S mix app.start
```

La tarea ***run*** es equivalente a ***app.start***.

app.tree

Muestra el árbol de aplicaciones del proyecto. En el proyecto del capítulo anterior tal y como lo dejamos tenemos estas aplicaciones:

```
$ mix app.tree --format plain
cuatro
|-- elixir
|-- logger
|   `-- elixir
`-- runtime_tools
```

Vemos como la aplicación principal es ***cuatro*** y depende directamente de las aplicaciones: ***elixir***, ***logger*** y ***runtime_tools***. En estos listados es común ver como todas las aplicaciones realizadas en Elixir dependen de la aplicación ***elixir***.

archive, archive.build, archive.install, archive.uninstall

Estos archivos son aplicaciones incluidas para extender el funcionamiento de **mix**. Si ejecutamos **mix archive** veremos los que tenemos en estos momentos instalados. Por mi parte tengo tan solo ***hex***.

```
$ mix archive
* hex-0.18.2
Archives installed at: /.../archives/elixir-1.7.4
```

Si jugamos con Phoenix Framework nos sugieren instalar su archivo y así poder crear proyectos. Podemos instalar un archivo así:

```
$ mix archive.install hex phx_new 1.4.0
Resolving Hex dependencies...
Dependency resolution completed:
New:
  phx_new 1.4.0
* Getting phx_new (Hex package)
All dependencies up to date
Compiling 10 files (.ex)
Generated phx_new app
```

```
Generated archive "phx_new-1.4.0.ez" with MIX_ENV=prod
Are you sure you want to install "phx_new-1.4.0.ez"? [Yn]
* creating /.../archives/elixir-1.7.4/phx_new-1.4.0

$ mix archive
* hex-0.18.2
* phx_new-1.4.0
Archives installed at: /.../archives/elixir-1.7.4
```

Podemos ver cómo ahora disponemos del archivo ***phx_new-1.4.0*** y ejecutando **mix help** obtenemos nuevos comandos para ejecutar con **mix**. Si no queremos ese archivo o funcionalidad podemos emplear el comando **mix uninstall** para eliminar cualquiera de los archivos instalados. El comando **archive.build** nos permite construir un archivo propio.

deps, deps.clean, deps.compile, deps.update, deps.tree, deps.unlock, deps.get

Estos comandos nos permiten trabajar con dependencias. Todos estos comandos los veremos más en detalle en la siguiente sección.

compile, clean

Hemos visto con anterioridad estos comandos. Nos permiten compilar el proyecto o eliminar los ficheros generados por la compilación respectivamente. Estos comandos además tienen algunas opciones más que puedes ver ejecutando **mix help compile** y **mix help clean**.

escript, escript.build, escript.install, escript.uninstall

Al igual que podemos instalar archivos para extender la funcionalidad de **mix** podemos crear proyectos que generen un archivo ejecutable y disponer de un nuevo comando para nuestro sistema. Veremos esto más adelante en la Sección9, " ***Scripting*** con Elixir".

format

Este comando nos permite formatear todos los ficheros de código de nuestro proyecto. Esta fue una agregación para la versión 1.7 de Elixir. La idea es mantener un único estilo por proyecto pudiendo configurar el estilo a través del fichero `.formatter.exs`.

hex, hex.*

Esta colección de comandos está pensada para la publicación de código a modo de librerías en el repositorio de hex[2]. Veremos un poco más adelante más sobre Hex.

[2] https://hex.pm

local, local.rebar, local.hex, local.public_keys

Estos comandos muestran tareas locales. Las tareas locales son las instaladas como archivos que no están presentes de forma nativa en **mix**. Cada vez que instalemos un archivo aparecerá un nuevo comando local. En nuestro caso, si seguiste la ejecución anterior e instalaste el archivo de Phoenix Framework, ejecutando **mix help** podrás ver en el listado las tareas de Phoenix Framework para crear nuevos proyectos pero además la tarea **mix local.phx**. Estos comandos nos facilitan actualizar estos archivos:

```
$ mix local.hex
Found existing entry: /.../elixir-1.7.4/phx_new-1.4.0
Are you sure you want to replace it with "https://
github.com/phoenixframework/archives/raw/master/
phx_new.ez"? [Yn]
* creating /.../archives/elixir-1.7.4/phx_new
```

También disponemos del acceso a las claves públicas que nos permiten el acceso a Hex a través del comando **mix local.public_keys**.

new

Vimos con anterioridad esta tarea para crear proyectos de Elixir desde cero. Esta tarea no solo nos permite crear un proyecto estándar sino también uno aún más básico, sin aplicación definida o más complejo como puede ser un proyecto ***umbrella***. Puedes ver más información ejecutando **mix help new**.

profile.cprof, profile.eprof, profile.fprof

BEAM dispone de tres tipos de perfiladores[3]: cprof, eprof y fprof. Podemos usar cualquiera de ellos para obtener información de la ejecución de nuestra aplicación. La forma de emplearlos queda fuera del alcance de este libro.

test

El sistema de pruebas (tests) es una de las potencias de Elixir. Está integrado en el lenguaje y al generar un proyecto siempre se genera el directorio de pruebas. El sistema de pruebas además tiene muchas ventajas y está muy avanzado. Aún siendo muy fácil y bastante importante no he podido dedicarle tiempo dentro de este libro.

[3]Traduciré profile como perfilador por tratarse de una herramienta que nos permite perfilar o ajustar el rendimiento de nuestra aplicación diciéndonos dónde están los cuellos de botella.

xref

Analiza el código e informa de llamadas a funciones no existentes. Esta comprobación es muy útil en combinación con las pruebas unitarias que puedas desarrollar para garantizar un mínimo de estabilidad en el código.

Finalmente y antes de cerrar este capítulo dejo mi recomendación para la compilación de un proyecto desde cero o para realizar una compilación completa:

```
mix do clean --deps, deps.get, compile, xref, test --cover
```

De esta forma nos aseguramos de limpiar bien el proyecto para proceder con la descarga de dependencias, una compilación completa del proyecto y después una comprobación del código primero de funciones llamadas (xref) y después de pruebas unitarias que deberemos implementar a través de ficheros dentro del directorio `tests` activando la salida de los porcentajes de cobertura del código.

2. Dependencias y Hex

Vamos a continuar con nuestro proyecto. El juego Conecta Cuatro™ a nivel de consola no es óptimo para poder jugar con otros jugadores e incluso para nosotros mismos puede resultar un poco molesto no disponer de una interfaz propia donde se faciliten los comandos.

Crearemos nuestra segunda versión cambiando esto. No obstante no vamos a desarrollar todo desde cero nosotros solos. Uno de los principios del desarrollo de software es la reutilización. Es por eso que todos los lenguajes de programación han desarrollado un ecosistema donde los programadores pueden publicar y compartir código con otros. Sitios colaborativos donde además puedes recibir sugerencias, propuestas de cambios, correcciones y/o mejoras.

Para Elixir esta red se denomina Hex y podemos encontrar todo el catálogo de librerías disponibles a través de su página web: hex.pm[4]. La web de Hex nos permite además crear nuestro propio usuario. Generar nuestro propio usuario nos permitirá poder agregar al catálogo de Hex nuestras propias librerías.

La creación se basa en unos requisitos fijos. Debemos de proveer no solo el código sino también la licencia del código, información del creador e importante, la documentación. Si nos fijamos en cualquier librería disponible para Elixir (y hecha en Elixir) dentro de Hex veremos el enlace

[4] https://hex.pm

a su documentación. Cuando subimos el código también se sube la colección de páginas HTML generadas de la documentación del proyecto.

Vamos a agregar un par de dependencias a nuestro proyecto. Como a día de hoy la interfaz más común para proveer un software es la interfaz web instalaremos cowboy[5]. Esta librería nos proporciona una interfaz HTTP para publicar una página básica y websockets para establecer la comunicación entre los usuarios y el juego.

La comunicación entre cliente y servidor se realizará empleando JSON[6] así que agregamos otra librería: jason[7]. Esta librería nos ayudará para codificar/decodificar los mensajes enviados/recibidos en la comunicación con el cliente.

Además para generar lanzamientos y paquetes de nuestra aplicación y así poder llevarla a servidores donde se ejecute de forma fácil emplearemos distillery[8]. Esta librería nos proporcionará los comandos necesarios para realizar el empaquetado y las actualizaciones en caliente que veremos más adelante.

Agregamos las dependencias y dejamos nuestro fichero `mix.exs` así:

```
defmodule Cuatro.Mixfile do
  use Mix.Project

  def project do
    [
      app: :cuatro,
      version: "2.0.0",
      elixir: "~> 1.7",
      elixirc_paths: ["lib"],
      start_permanent: true,
      deps: deps()
    ]
  end

  def application do
    [
      mod: {Cuatro.Application, []},
      extra_applications: [:logger, :runtime_tools]
    ]
  end

  defp deps do
    [
      {:jason, "~> 1.1"},
      {:cowboy, "~> 2.5"},
      {:distillery, "~> 2.0"}
    ]
```

[5] https://hex.pm/packages/cowboy
[6] JSON son las siglas de JavaScript Object Notation o Notación de Objetos JavaScript. Es una notación bastante simple y fácil de codificar para JavaScript por lo que se ha difundido mucho en entornos web.
[7] https://hex.pm/packages/jason
[8] https://hex.pm/packages/distillery

```
end
```

Como podemos apreciar además de agregar las dependencias hemos incrementado la versión a 2.0.0. Podemos proceder a compilar para asegurarnos de que las dependencias se bajan y compilan correctamente. Además ***distillery*** nos proporciona nuevas tareas para **mix**. Las iremos viendo en las siguientes secciones.

Ejecutamos ahora la tarea **mix deps.get** para obtener las dependencias. Estas se descargan y descomprimen en el directorio `deps`. Este directorio no es necesario agregarlo en nuestro control de versiones porque gracias al sistema de descarga desde Hex podemos reproducir este paso siempre que queramos.

Si miramos el directorio base del proyecto veremos que ha aparecido además un nuevo fichero: `mix.lock`. Este fichero contiene en formato de Elixir información sobre las dependencias instaladas. La principal función de este fichero es instalar siempre la misma versión de las librerías y así evitar al compartir el código que otro desarrollador descargue otras versiones diferentes de las librerías pudiendo sucederse errores por el empleo de otro código diferente.

Podemos desbloquear algunos paquetes con la tarea **mix deps.unlock**. De esta forma el fichero es eliminado de `mix.lock` y nos permite descargar una nueva versión basada en la última disponible que concuerde con la versión especificada en `mix.exs`.

Si nuestra intención es actualizar la librería podemos hacerlo en un solo paso con la tarea **mix deps.update**. De esta forma se desbloquea la librería y busca por una nueva versión disponible que concuerde con la restricción especificada en `mix.exs`.

3. Cambios en el Código

Antes de continuar con el proceso del lanzamiento y seguir viendo las tareas de **mix** para ello vamos a centrarnos en los cambios de nuestra segunda versión.

El código del juego lo desarrollé de forma agnóstica a cómo jugar. De esta forma el módulo ***Cuatro.Juego*** no sufre casi ninguna modificación. Los jugadores siguen siendo procesos solo que esta vez cada proceso será una conexión websocket iniciada desde cowboy. Esa parte no cambia pero sí la forma de localizar los procesos. En lugar de lanzar un único juego con el nombre del módulo vamos a implementar un ***Registry*** tal y como vimos en la Sección 4, "Registros" del Capítulo 10, ***Un vistazo a OTP***. Simplemente cambiamos la API para interactuar con el servidor y la forma de inicio:

```
defp via(juego) do
  {:via, Registry, {Cuatro.Registry, juego}}
end

@doc "Inicia el juego"
def start_link(juego) do
  GenServer.start_link __MODULE__, [], name: via(juego)
end

def stop(juego) do
  GenServer.stop via(juego)
end

@doc "Retorna información de quién es el siguiente"
def quien_juega?(juego) do
  GenServer.call via(juego), :quien_juega
end

def quien_soy?(juego) do
  GenServer.call via(juego), :quien_soy
end

def quien_gana?(juego) do
  GenServer.call via(juego), :quien_gana
end

def inscribe_jugador(juego) do
  GenServer.call via(juego), {:jugador, self()}
end

def existe?(juego) do
  case Registry.lookup(Cuatro.Registry, juego) do
    [{_pid, nil}] -> true
    [] -> false
  end
end

@doc "Inserta una ficha en el tablero"
def inserta(juego, col)
    when is_integer(col) and col >= 0 do
  GenServer.call via(juego), {:col, col}
end

def jugadores(juego) do
  GenServer.call via(juego), :jugadores
end

@doc "Muestra las columnas"
def muestra(juego) do
  GenServer.call via(juego), :muestra
end
```

De esta forma cada vez que un jugador solicite crear un nuevo juego proveyendo un nombre, el sistema podrá buscar ese nombre en el registro y crearlo si no existe. Para saber si existe o no disponemos de la función nueva `Cuatro.Juego.existe?/1`.

Importante

La configuración de cowboy es bastante simple. La librería dispone de buena documentación. Solo debes asegurarte de leer la documentación correspondiente a la versión que estés usando porque la API y los datos usados por la librería han sufrido muchas modificaciones de una versión a otra.

El fichero de aplicación también ha sufrido modificaciones. La principal es el uso de la configuración para flexibilizar el puerto y el tipo de conexión permitida para escuchar:

```
defmodule Cuatro.Application do
  use Application

  require Logger

  @port 1234
  @family :inet

  # See https://hexdocs.pm/elixir/Application.html
  # for more information on OTP Applications
  def start(_type, _args) do
    import Supervisor.Spec

    port = Application.get_env(:cuatro, :port, @port)
    family = Application.get_env(:cuatro, :family, @family)

    children = [
      supervisor(Registry, [:unique, Cuatro.Registry]),
      worker(Cuatro.Http, [port, family])
    ]

    Logger.info "[app] iniciada aplicación"

    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: Cuatro.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Además vemos que los procesos supervisados ahora son el registro donde se irán anotando todos los juegos que se creen y un nuevo módulo ***Cuatro.Http*** al que se le pasa la configuración del puerto y la familia para establecer la escucha.

El módulo ***Cuatro.Http*** es usado por cowboy para cada petición entrante vía HTTP y no solo configuramos las peticiones HTTP sino también la gestión de la comunicación vía websocket que correrá a cargo de otro nuevo módulo: ***Cuatro.Websocket***.

Las conexiones HTTP atenderán solo peticiones para ciertos ficheros agregados en el directorio `priv`. Este directorio también nuevo en nuestra base del proyecto tendrá esta forma:

```
priv
|-- app.css
|-- app.js
|-- favicon.ico
`-- index.html
```

Nota

Estos ficheros puedes descargarlos de la sección de recursos del libro: https://books.altenwald.com/book/elixir

El directorio `priv` es un directorio especial. BEAM entiende este directorio como parte de la aplicación y el lugar donde se situarán datos necesarios para la aplicación por lo que será copiado dentro del lanzamiento generado por distillery.

El módulo para gestión de las peticiones HTTP tendrá esta forma:

```
defmodule Cuatro.Http do
  require Logger

  defp priv(file), do: priv('/' ++ file, file)
  defp priv(path, file) do
    {path, :cowboy_static, {:priv_file, :cuatro, file}}
  end

  def start_link(port_number, family) do
    dispatch = :cowboy_router.compile [
      {:_, [
        priv('/', 'index.html'),
        priv('favicon.ico'),
        priv('app.css'),
        priv('app.js'),
        {'/websession', Cuatro.Websocket, []}
      ]}
    ]
    opts = %{env: %{dispatch: dispatch}}
    port = [{:port, port_number}, family]
    {:ok, _} = :cowboy.start_clear(Cuatro.Http, port, opts)
  end

  def init(req, opts) do
    {:cowboy_websocket, req, opts}
  end

  def handle(req, state) do
    Logger.debug "Petición no esperada: #{inspect req}"
    headers = %{"content-Type" => "text/html"}
    {:ok, req} = :cowboy_req.reply(404, headers)
    {:ok, req, state}
  end
```

```
  def terminate(_reason, _req, _state) do
    Logger.info "terminate (#{inspect self()})"
    :ok
  end
end
```

El módulo para la gestión de las peticiones websocket es algo más largo y actúa como el controlador entre el juego y el jugador. Guarda bastantes similitudes con el anterior módulo ***Cuatro*** pero orientado a la información recibida mediante el websocket:

```
defmodule Cuatro.Websocket do
  require Logger
  alias Cuatro.Juego

  @jugador1 0
  @jugador2 1

  @jugador1_color "rojo"
  @jugador2_color "amarillo"

  @jugador1_simbolo "<td class='rojo'>&#9673;</td>"
  @jugador2_simbolo "<td class='amarillo'>&#9673;</td>"
  @nojugado_simbolo "<td class='vacio'>&#9673;</td>"

  @msg_ganador "<h1>¡¡¡ Ganaste !!!</h1>"
  @msg_perdedor "<h1>Perdiste :-(</h1>"
  @msg_empate "<h1>¡Empate!</h1>"

  def init(req, opts) do
    {:cowboy_websocket, req, opts}
  end

  def websocket_init(_opts) do
    {:ok, %{}}
  end

  def websocket_handle({:text, msg}, state) do
    msg
    |> Jason.decode!()
    |> process_data(state)
  end

  def websocket_handle(_any, state) do
    {:reply, {:text, "eh?"}, state}
  end

  def websocket_info({:send, data}, state) do
    {:reply, {:text, data}, state}
  end
  def websocket_info({:timeout, _ref, msg}, state) do
    {:reply, {:text, msg}, state}
  end

  def websocket_info(info, state) do
    Logger.info "info => #{inspect info}"
    {:ok, state}
  end

  def websocket_terminate(reason, _state) do
```

```
  Logger.info "reason => #{inspect reason}"
  :ok
end

defp send_to_all(juego, data) do
  data_json = Jason.encode!(data)
  juego
  |> Juego.jugadores()
  |> Enum.each(fn(jugador) ->
                 send(jugador, {:send, data_json})
               end)
end

defp reply(data, state) do
  data_json = Jason.encode!(data)
  {:reply, {:text, data_json}, state}
end

defp process_data(%{"type" => "inserta", "pos" => pos},
                  state) do
  if Juego.existe?(state.juego) do
    case Juego.inserta(state.juego, pos) do
      {:gana, quien} ->
        html = muestra state.juego
        msg = %{"type" => "gana", "html" => html}
        [ganador, perdedor] = case quien do
          @jugador1 -> Juego.jugadores(state.juego)
          @jugador2 -> Juego.jugadores(state.juego)
                       |> Enum.reverse()
        end
        gana = Map.put(msg, "msg", @msg_ganador)
               |> Jason.encode!()
        pierde = Map.put(msg, "msg", @msg_perdedor)
                 |> Jason.encode!()
        send ganador, {:send, gana}
        send perdedor, {:send, pierde}
        Juego.stop state.juego
      :lleno ->
        html = muestra state.juego
        msg = %{"type" => "gana", "html" => html,
                "msg" => @msg_empate}
        send_to_all state.juego, msg
        Juego.stop state.juego
      :sigue ->
        html = muestra state.juego
        msg = %{"type" => "dibuja", "html" => html}
        send_to_all state.juego, msg
        quien = state.juego
                |> Juego.quien_juega?()
                |> color()
        turno = %{"type" => "turno", "quien" => quien}
        send_to_all state.juego, turno
      :col_llena ->
        :ok
      :turno_de_otro ->
        quien = state.juego
                |> Juego.quien_juega?()
                |> color()
        turno = %{"type" => "turno", "quien" => quien}
        send_to_all state.juego, turno
        :ok
    end
```

```
  end
  {:ok, state}
end

defp process_data(%{"type" => "muestra"}, state) do
  if Juego.existe?(state.juego) do
    html = muestra(state.juego)
    unless is_atom(html) do
      msg = %{"type" => "dibuja", "html" => html}
      send_to_all state.juego, msg
    end
  end
  {:ok, state}
end

defp process_data(%{"type" => "login", "juego" => juego},
                  state) do
  unless Juego.existe?(juego) do
    Juego.start_link juego
  end
  case Juego.inscribe_jugador(juego) do
    :partida_ocupada ->
      msg = %{"type" => "login", "result" => "busy"}
      reply msg, state
    jugador ->
      state = %{juego: juego}
      reply %{"type" => "login",
              "result" => "ok",
              "color" => color(jugador)}, state
  end
end

defp color(@jugador1), do: @jugador1_color
defp color(@jugador2), do: @jugador2_color

defp muestra(juego) do
  case Juego.muestra(juego) do
    atom when is_atom(atom) -> atom
    cols ->
      "<table id='juego'><tr>"
      |> add(for(i <- 0..(length(cols) - 1), do: th(i))
             |> Enum.join())
      |> add("</tr><tr>")
      |> add(cols
             |> Enum.map(&Enum.reverse/1)
             |> traspone()
             |> Enum.map(&traduce_simbolos/1)
             |> Enum.join("</tr><tr>"))
      |> add("</tr><tr>")
      |> add(for(i <- 0..(length(cols) - 1), do: boton(i))
             |> Enum.join())
      |> add("</tr></table>")
  end
end

defp add(str1, str2), do: str1 <> str2

defp th(i) do
  "<th>#{i + 1}</th>"
end

defp boton(i) do
```

```
    "<td><button class='drop' id='drop_#{i}'>" <>
    "&uarr;</button></td>"
  end

  defp traspone([[]|_]), do: []
  defp traspone(matriz) do
    [ Enum.map(matriz, &hd/1) |
      traspone(Enum.map(matriz, &tl/1)) ]
  end

  defp simbolo(@jugador1), do: @jugador1_simbolo
  defp simbolo(@jugador2), do: @jugador2_simbolo
  defp simbolo(nil), do: @nojugado_simbolo

  defp traduce_simbolos(col) do
    col
    |> Enum.map(&simbolo/1)
    |> Enum.join()
  end
end
```

Con estas modificaciones ya podemos proceder a realizar una liberación y probar nuestro juego a través de un navegador web.

4. Preparando un Lanzamiento

Para generar lanzamientos ***distillery*** nos solicita disponer de un directorio nuevo en la base del proyecto llamado `rel`. Dentro de este directorio debe aparecer el fichero `config.exs`.

Por suerte no necesitamos generar toda esta estructura por nosotros mismos porque ***distillery*** dispone de una tarea de **mix** para realizar esta acción. Vamos a generar el directorio junto con toda la información necesaria:

```
mix release.init
```

Ahora disponemos de los ficheros necesarios para generar el lanzamiento. Podemos echar un vistazo al fichero de configuración generado. Por nuestra parte y gracias a que el proyecto es bastante simple no necesitamos modificar nada. He eliminado los comentarios para dejar solo la parte del código y configuración relevante:

```
~w(rel plugins *.exs)
|> Path.join()
|> Path.wildcard()
|> Enum.map(&Code.eval_file(&1))

use Mix.Releases.Config,
    default_release: :default,
    default_environment: Mix.env()

environment :dev do
```

```
  set dev_mode: true
  set include_erts: false
  set cookie: :"PALCx)hn$ZB&1),
$RY(usmSR8Tex.uaS>[Qia~r(>MP89EBYa[n`VKscGc!cY~Dj"
end

environment :prod do
  set include_erts: true
  set include_src: false
  set cookie: :"{2EQ$SA5l,QCwF@YIx3W*5Wz@Tw,QHY%!
8Q6M@4_;w5W[j1(kW:pp3s%ojquX]Bc"
end

release :cuatro do
  set version: current_version(:cuatro)
  set applications: [
    :runtime_tools
  ]
end
```

Puedes probar ahora a realizar un lanzamiento ejecutando lo siguiente:

```
$ mix do clean --deps, release
===> Compiling ranch
==> jason
Compiling 8 files (.ex)
Generated jason app
===> Compiling cowlib
===> Compiling cowboy
==> artificery
Compiling 10 files (.ex)
Generated artificery app
==> distillery
Compiling 33 files (.ex)
warning: variable "err" is unused
  lib/mix/lib/releases/runtime/pidfile.ex:50

Generated distillery app
==> cuatro
Compiling 4 files (.ex)
Generated cuatro app
==> Assembling release..
==> Building release cuatro:2.0.0 using environment dev
==> You have set dev_mode to true, skipping archival phase
Release successfully built!
To start the release you have built, you can use one of the
 following tasks:

    # start a shell, like 'iex -S mix'
    > _build/dev/rel/cuatro/bin/cuatro console

    # start in the foreground, like 'mix run --no-halt'
    > _build/dev/rel/cuatro/bin/cuatro foreground

    # start in the background, must be stopped with the 'stop'
 command
    > _build/dev/rel/cuatro/bin/cuatro start

If you started a release elsewhere, and wish to connect to it:

    # connects a local shell to the running node
```

```
    > _build/dev/rel/cuatro/bin/cuatro remote_console

    # connects directly to the running node's console
    > _build/dev/rel/cuatro/bin/cuatro attach

For a complete listing of commands and their use:

    > _build/dev/rel/cuatro/bin/cuatro help
```

En este paso vemos la compilación y generación del lanzamiento. Por último el sistema nos informa de la existencia de un script generado para nuestro proyecto que nos permite iniciar en primer plano y a través de una consola nuestra aplicación (console) o en primer plano pero sin consola (foreground) para poder emplear con Docker o supervisord o incluso en segundo plano y sin consola (start) que podemos emplear con scripts tipo init.d o systemd para iniciar nuestra aplicación cuando se inicie el sistema.

Además para cuando iniciamos en segundo plano disponemos de otros comandos para poder conectarnos a una consola auxiliar (remote_console) o a la misma consola mediante conexión Unix (attach) y nos informa que para más comandos y ayuda disponemos del comando **cuatro help**.

5. Entornos

Mix gestiona diferentes entornos y nos permite configurar nuestro entorno, la configuración y la generación de nuestro código acorde a estos entornos. Al mismo tiempo que podemos querer dependencias disponibles solo para el entorno de desarrollo o el entorno de test, también podemos querer una configuración específica para el entorno de producción diferente a la del resto de entornos. Esto lo conseguimos gracias a Mix y el uso de la función `Mix.env/0`. Esta función siempre nos retornará un átomo con el entorno en uso.

Esta función puede ser empleada en los ficheros de configuración (normalmente `config.exs`) y en `mix.exs`. De esta forma podemos cambiar y modificar opciones dependiendo del entorno.

Además de esto las dependencias presentan una configuración extra para instalar y usar dependencias dependiendo del entorno donde estemos. Por ejemplo si queremos usar una dependencia para mostrar mejor la información de cobertura de los tests no tiene sentido emplear esta dependencia en producción y quizás tampoco en desarrollo. Esto lo podemos conseguir así:

```
  def deps do
    [
      {:excoveralls, "~> 0.10.3", only: :test}
```

```
    ]
  end
```

De esta forma incluso aunque la dependencia se descargue y se agregue al fichero `mix.lock` no será nunca agregada como dependencia a la aplicación ni llevada al lanzamiento de producción.

Para indicar los entornos en los que construimos normalmente empleamos las variables de entorno. En este caso la variable para especificar el entorno es ***MIX_ENV***. Si queremos construir en entorno producción el proyecto en una consola bash podemos hacerlo así:

```
MIX_ENV=prod mix do clean --deps, deps.get, compile
```

Vemos en esta ocasión que tras la compilación está disponible un nuevo directorio dentro de `_build` llamado `prod` donde se ha realizado la compilación de todo el proyecto acorde a la configuración del entorno.

Importante

Hemos visto que distillery tiene entornos definidos para generar los lanzamientos. Cuando realizamos la ejecución del comando especificando el entorno de producción para distillery el código y la construcción de todo seguía estando sin embargo aún bajo `_build/dev` esto es porque distillery y mix manejan sus entornos por separado. Para generar una compilación de producción y un lanzamiento de producción habría que ejecutar:

```
$ MIX_ENV=prod mix release --env=prod
```

De esta forma generamos en el entorno ***prod*** un lanzamiento acorde a la configuración ***prod*** de distillery.

6. Puesta en Producción

En principio nuestro lanzamiento se encuentra preparado y podemos ejecutar esos comandos con las rutas dadas para probarlo. No obstante esta versión no es portable. Si queremos generar una versión portable deberemos emplear el entorno de producción de distillery:

```
mix release --env=prod
```

La salida es prácticamente la misma pero nos informa esta vez de estar usando el entorno de producción tomado del fichero de configuración `rel/config.exs` y esta vez sí genera un lanzamiento portable en el directorio `_build/dev/rel/cuatro`.

Una vez generado el lanzamiento podemos encontrar en la siguiente ruta un fichero comprimido con todo nuestro software portable[9]:

```
_build/dev/rel/cuatro/releases/2.0.0/cuatro.tar.gz
```

Si descomprimimos este paquete en otro directorio podemos apreciar que se crean cuatro directorios: `lib`, `releases`, `bin` y `erts-10.1.3`. El directorio `bin` contiene el script necesario para iniciar el sistema tal y como nos presentaba la ayuda de distillery tras la generación del lanzamiento. Podemos ejecutarlo en primer plano para ver cómo funciona:

```
$ bin/cuatro console
Erlang/OTP 21 [erts-10.1.3] [source] [64-bit] [smp:8:8]
 [ds:8:8:10] [async-threads:1] [hipe] [dtrace]


13:11:08.994 [info]  [app] iniciada aplicación
Interactive Elixir (1.7.4) - press Ctrl+C to exit (type h()
 ENTER for help)
iex(cuatro@127.0.0.1)1> :inet.i :tcp, [:local_address]
Local Address
*:51301
localhost:51302
*:1234
:ok
```

Por defecto el puerto de escucha es `1234` y vemos que ha sido establecido. Si vamos a un navegador y escribimos la URL podremos ver nuestro juego:

http://localhost:1234

Puedes probar también a interrumpir la consola e iniciar con la opción **bin/cuatro start** y después entrar en la consola mediante una conexión remota con la opción **bin/cuatro remote_console**. Por último también puedes detener el servidor con la opción **bin/cuatro stop**.

7. Actualizaciones en Caliente

Una de las ventajas de distillery es la generación automática para las actualizaciones de código. El único requisito es tener construido un lanzamiento de la versión anterior en `_build`. Así que realizaremos un cambio en la interfaz del juego. Este cambio lo dejo a tu elección, puedes modificar el fichero HTML o agregar alguna función, modificar la salida de los mensajes o cualquier modificación que quieras realizar.

[9]Un software portable se refiere normalmente a un programa ejecutable sin dependencias externas a instalar que puede ser llevado a otro sistema y ser ejecutado sin instalación.

Lo más importante es que modifiques también el número de versión en `mix.exs` al número 2.0.1.

Una vez tengas la modificación y la hayas probado ejecutándola de forma local tal y como comentábamos en la Sección4, "Aplicación ***cuatro***" del Capítulo11, ***Aplicaciones y Supervisores*** procedemos al lanzamiento:

```
$ mix release --env=prod --upgrade
Compiling 4 files (.ex)
Generated cuatro app
==> Assembling release..
==> Building release cuatro:2.0.1 using environment prod
==> Generated .appup for cuatro 2.0.0 -> 2.0.1
==> Relup successfully created
==> Including ERTS 10.1.3 from /usr/local/Cellar/
erlang/21.1.3/lib/erlang/erts-10.1.3
==> Packaging release..
Release successfully built!
To start the release you have built, you can use one of the
 following tasks:

    # start a shell, like 'iex -S mix'
    > _build/dev/rel/cuatro/bin/cuatro console

    # start in the foreground, like 'mix run --no-halt'
    > _build/dev/rel/cuatro/bin/cuatro foreground

    # start in the background, must be stopped with the 'stop'
 command
    > _build/dev/rel/cuatro/bin/cuatro start

If you started a release elsewhere, and wish to connect to it:

    # connects a local shell to the running node
    > _build/dev/rel/cuatro/bin/cuatro remote_console

    # connects directly to the running node's console
    > _build/dev/rel/cuatro/bin/cuatro attach

For a complete listing of commands and their use:

    > _build/dev/rel/cuatro/bin/cuatro help
```

Vemos cómo se ha generado una actualización de 2.0.0 a 2.0.1. En realidad las actualizaciones son ficheros con extensión appup que se sitúan en el directorio `ebin`. El contenido del fichero está escrito en Erlang para ser entendido por BEAM. Distillery se encarga de realizar la labor de comprobar la versión de cada módulo de la versión anterior y escribir en este fichero lo necesario para realizar el cambio de una versión a la siguiente.

Si queremos especificar cambios algo más complejos podemos generar un fichero de actualización mediante la tarea **mix release.gen.appup --app=cuatro** y escribir dentro del fichero generado en la ruta `rel/appups/cuatro/2.0.0_to_2.0.1.appup` los pasos necesarios para subir a esa versión y los pasos para volver a la versión anterior.

Nota

La especificación de los pasos y los pasos posibles para dar es un tema bastante extenso. Afortunadamente está cubierto en el libro Erlang/OTP Volumen II: Las Bases de OTP[10]. Puedes revisar el capítulo 9 sobre aplicaciones y el capítulo 11 sobre lanzamientos donde obtendrás más información sobre las aplicaciones, lanzamientos y actualización en caliente dentro de BEAM.

Una vez tenemos nuestro nuevo paquete preparado la forma de actualizar es copiar ese nuevo fichero un nuevo directorio del sistema en ejecución y ejecutar el comando de despliegue a través del script **bin/cuatro**. El fichero a copiar se encuentra en esta ruta:

```
_build/dev/rel/cuatro/releases/2.0.1/cuatro.tar.gz
```

El fichero debe ser accesible para el sistema en ejecución a través de la ruta que mostramos a continuación. Deberás crear el directorio 2.0.1 y luego copiar el fichero dentro:

```
releases/2.0.1/cuatro.tar.gz
```

El último paso es ejecutar el script con la opción ***upgrade*** para notificar al sistema de la existencia de una nueva versión y se proceda a cambiarla:

```
$ bin/cuatro upgrade 2.0.1
Release cuatro:2.0.1 not found, attempting to unpack
 releases/2.0.1/cuatro.tar.gz
Unpacked '2.0.1' successfully!
Release cuatro:2.0.1 is already unpacked, installing..
Installed release cuatro:2.0.1
Made release cuatro:2.0.1 permanent
```

Si accedemos ahora a nuestra interfaz web veremos los cambios que hayamos realizado.

8. Agregando tareas a mix

Antes de finalizar la exposición del entorno vamos a hablar de cómo agregar tareas a Mix. Una de las formas más sencillas es a través de los aliases. Dentro de un proyecto podemos crear o sobrescribir una tarea para que realice otras diferentes.

Por ejemplo, comenté inicialmente las tareas para una compilación completa desde cero. Para evitar tener que escribir una y otra vez lo mismo podemos hacer lo siguiente:

[10] https://books.altenwald.com/erlang-ii

```
def aliases do
  [
    full: ["clean --deps",
           "deps.get",
           "compile",
           "xref",
           "test"],
    release: ["release --env=prod --upgrade"]
  ]
end
```

De esta forma estamos creando la nueva tarea ***full*** que realizará todas las tareas referidas en la lista y aprovechamos para sobrecargar la tarea ***release*** para que emplee siempre los parámetros especificados.

Igualmente podemos agregar comandos de consola mediante la tarea ***cmd*** o la función `Mix.Shell.cmd/1` y crear funciones o clausuras para realizar más acciones. Supongamos que queremos copiar todas las liberaciones al directorio base en la ruta `releases`:

```
def release_copy(_) do
  File.mkdir "releases/#{project()[:version]}"
  "_build/dev/rel/cuatro/releases/*/*.tar.gz"
  |> Path.wildcard()
  |> Enum.each(fn(file) ->
                [dir, f] = file
                           |> Path.split()
                           |> Enum.reverse()
                           |> Enum.slice(0, 2)
                           |> Enum.reverse()
                target = "releases/#{dir}"
                File.mkdir_p! target
                File.cp! file, Path.join(target, f)
              end)
end
```

Hay que reconocer que quedó un poco extenso. La idea es obtener los ficheros que concuerden con la ruta de la primera línea. Debemos extraer el directorio inmediato donde se encuentra y realizar la copia a la nueva localización dentro de un directorio con el número de versión.

Agregamos esta tarea a los aliases:

```
def aliases do
  [
    "release.copy": &release_copy/1,
    full: ["clean --deps",
           "deps.get",
           "compile",
           "xref",
           "test",
           "release.copy"],
    release: ["release --env=prod --upgrade"]
  ]
end
```

No debemos olvidar agregar la configuración de aliases en el proyecto:

```
def project do
  [
    app: :cuatro,
    version: "2.0.1",
    elixir: "~> 1.7",
    elixirc_paths: ["lib"],
    start_permanent: true,
    deps: deps(),
    aliases: aliases()
  ]
end
```

De esta forma ya queda configurado y podemos usar nuestros aliases.

La segunda forma de crear tareas es mediante la creación de un módulo que debe tener como raíz ***Mix.Tasks*** y usar o extender el módulo ***Mix.Task***. La única función necesaria para la ejecución de la tarea será `run/1`.

La creación de tareas de esta forma está más ligada al código interno y se realiza de esta forma para tener acceso a otros módulos del código y configuración. Es ideal para la realización de tareas para modificación de datos en una base de datos, uso de servicios web configurados en la aplicación o en definitiva el uso de algún recurso específico de la aplicación.

9. *Scripting* con Elixir

Una de las ventajas de Elixir es su formato. Nos permite poder escribir código de forma muy simple y tanto es así que podemos incluso escribir ficheros de scripting. Estos ficheros son compilados al vuelo y ejecutados al instante.

De forma simple y empleando la extensión ***exs***[11] podemos escribir un fichero que nos permita realizar acciones al más propio estilo de bash.

Además podemos crear un proyecto para generar un script. Esto es ligeramente diferente porque el script generado es un paquete donde se agrega todo el código compilado. Vamos a hacer una de estas aplicaciones simples para implementar una calculadora. Aceptando tres parámetros donde el primero y tercero serán números y el segundo la operación a realizar. Creamos el proyecto:

```
$ mix new calc
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
```

[11] La extensión ***exs*** es una contracción de Elixir Scripting y la hemos visto en ficheros como `mix.exs` o `config.exs`.

```
* creating config
* creating config/config.exs
* creating lib
* creating lib/calc.ex
* creating test
* creating test/test_helper.exs
* creating test/calc_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd calc
    mix test

Run "mix help" for more commands.
```

No necesitamos la mayor parte de los ficheros generados, vamos a hablar sobre los que sí necesitamos. En principio debemos modificar el fichero `mix.exs`. Los proyectos para scripts deben agregar el bloque escript tal y como se ve en el fichero:

```
defmodule Calc.MixProject do
  use Mix.Project

  def project do
    [
      app: :calc,
      version: "0.1.0",
      elixir: "~> 1.7",
      start_permanent: Mix.env() == :prod,
      escript: [main_module: Calc]
    ]
  end
end
```

En esta ocasión he eliminado el bloque de aplicación porque no generamos ninguna y no requerimos de ninguna dependencia. De igual forma tampoco necesitamos ningún alias ni nada más.

El código será muy simple en tan solo un módulo:

```
defmodule Calc do
  def main([op1, "+", op2]), do: IO.puts num(op1) + num(op2)
  def main([op1, "-", op2]), do: IO.puts num(op1) - num(op2)
  def main([op1, "*", op2]), do: IO.puts num(op1) * num(op2)
  def main([op1, "/", op2]), do: IO.puts num(op1) / num(op2)
  def main(_), do: IO.puts "Syntax: calc <n1> <+|-|*|/> <n2>"

  defp num(string) do
    {number, ""} = Float.parse(string)
    number
  end
end
```

Se llama a la función `Calc.main/1` pasando una lista de binarios con todos los parámetros pasados en la línea de comandos. Si no pasamos

exactamente 3 parámetros y el segundo parámetro es una operación reconocible nos presenta la sintaxis.

Vamos a compilar todo y generar el script a través de **escript.build**:

```
$ mix escript.build
Compiling 1 file (.ex)
Generated escript calc with MIX_ENV=dev
```

En el directorio base habrá aparecido nuestro script `calc`. Podemos ejecutarlo para comprobar que funcione correctamente:

```
$ ./calc 10 + 2
12.0
$ ./calc 10 - 2
8.0
$ ./calc 10 / 2
5.0
$ ./calc 10 * 2
Syntax: calc <n1> <op> <n2>
$ ./calc 10 '*' 2
20.0
```

Como podemos apreciar el comando funciona. Hay restricciones con respecto a la multiplicación por el significado que ciertos intérpretes de comandos le dan y debe ser puesto entre comillas simples o dobles para evitar ese problema, pero por lo general funciona correctamente.

Ahora probaremos a instalar el script. Normalmente los scripts son instalados en `~/.mix/escripts` como podemos ver en la instalación de nuestro escript:

```
$ mix escript.install
Generated escript calc with MIX_ENV=dev
Are you sure you want to install "calc"? [Yn]
* creating /Users/manuel.rubio/.mix/escripts/calc

warning: you must append "/Users/manuel.rubio/.mix/escripts"
to your PATH if you want to invoke escripts by name

$ mix escript
* calc
Escripts installed at: /Users/manuel.rubio/.mix/escripts
```

Ya tenemos instalado el escript. El sistema nos recomienda introducir en el PATH de nuestro sistema la ruta para poder emplear los escripts que generemos desde cualquier ruta simplemente ejecutando **calc**.

Por último podemos eliminar el escript con el comando:

```
$ mix escript.uninstall calc
Are you sure you want to uninstall /Users/manuel.rubio/.mix/
escripts/calc? [Yn]
```

De esta forma tenemos control sobre los escripts instalados pudiendo listarlos, instalarlos y desinstalarlos de forma fácil.

Podemos escribir este mismo código como fichero `calc.exs` de esta forma:

```
#!/usr/bin/env elixir

num = fn string ->
        {number, ""} = Float.parse(string)
        number
      end

case System.argv() do
  [op1, "+", op2] -> IO.puts num.(op1) + num.(op2)
  [op1, "-", op2] -> IO.puts num.(op1) - num.(op2)
  [op1, "*", op2] -> IO.puts num.(op1) * num.(op2)
  [op1, "/", op2] -> IO.puts num.(op1) / num.(op2)
  _ -> IO.puts "Syntax: calc <n1> <+|-|*|/> <n2>"
end
```

Hacemos uso de `System.argv/0` para obtener los parámetros y no definimos ni módulos ni funciones. Podemos darle permisos de ejecución al fichero y probarlo de esta forma:

```
$ examples/cap12/calc.exs
Syntax: calc <n1> <+|-|*|/> <n2>
$ examples/cap12/calc.exs 1 + 2
3.0
```

La extensión en realidad no se requiere. Puedes crear scripts sin extensión. Además puedes agregar el código Elixir de la forma que he mostrado y agregar módulos para usarlos sin problema.

Apéndices

Apéndice A. Instalación de Elixir

Elixir puede instalarse de muchas formas dependiendo del sistema. Repasaremos los pasos para tener el sistema funcional de las formas más usadas según el sistema operativo. Cubriremos: Windows, GNU/Linux y Mac.

Este apéndice no es una copia exacta pero sí se basa mucho en esta página (en inglés) donde se menciona cómo instalar el sistema en las distintas plataformas:

https://elixir-lang.org/install.html

Ante cualquier duda puedes revisar ese enlace o dejar un comentario en la web del libro:

https://books.altenwald.com/comments/elixir

1. Instalación en Windows

Puedes descargar el instalador para Windows desde el siguiente enlace:

https://repo.hex.pm/elixir-websetup.exe

El instalador te ayuda a instalar tanto Erlang como Elixir para dejarlo disponible bajo Windows. Después desde cualquier consola tendrás accesibles los comandos **elixir**, **elixirc** y **iex** principalmente.

2. Instalación en GNU/Linux

Normalmente a través del gestor de paquetes de la distribución que uses deberías poder instalar Elixir. La inclusión dentro de las distintas distribuciones es la siguiente:

Distribución	**Elixir**	**Erlang/OTP**
Debian Stretch 9	1.3.3	19.2
Debian Buster 10	1.7.4	21.1
Ubuntu Xenial 16.04LTS	1.1.0	18.3
Ubuntu Bionic 18.04LTS	1.3.3	20.2
Ubuntu Cosmic 18.10	1.6.5	20.3

En sistemas CentOS 5, 6 y 7 la versión de Erlang/OTP es demasiado antigua para soportar Elixir. Se recomienda una instalación a través de alguna de las alternativas mostradas más abajo.

De las distribuciones mostradas ahora mismo la más actual o conforme al programa de liberaciones de Erlang y Elixir es Debian Buster 10. Puedes instalar simplemente usando (como usuario root o empleando **sudo**):

```
apt-get install elixir
```

También puedes emplear otros comandos como **aptitude**.

3. Instalación en Mac

La instalación en Mac la puedes realizar fácilmente a través de Homebrew[1]. Esta herramienta te permite instalar cualquier paquete de software libre disponible de forma rápida y sencilla mediante un comando de consola.

Para instalar Elixir solo tienes que ejecutar:

```
$ brew install elixir
```

Si no tienes instaladas sus dependencias se instalarán estas primero y después le tocará el turno a Elixir. El proceso puede durar bastante tiempo. Cuando acabe tendrás accesibles los comandos **elixir**, **elixirc** y **iex** principalmente.

4. Otros métodos

Si ya tienes instalado Erlang 20 o Erlang 21 en el mejor de los casos puedes proceder a instalar Elixir por otros cauces que los mostrados más arriba.

4.1. Kiex

En principio uno de los métodos es empleando kiex[2]. Esta herramienta no solo nos permite instalar Elixir sino también varias versiones diferentes de Elixir y seleccionar por consola cual queremos emplear cada vez manteniendo una por defecto.

De esta forma es bastante seguro actualizar a una nueva versión y comprobar si el código en el que trabajas funciona correctamente con los

[1] https://brew.sh/index_es
[2] https://github.com/taylor/kiex

nuevos cambios introducidos. Si no es así puedes volver con un comando rápido a la versión anterior y seguir trabajando.

Este sistema es el que suelo emplear. Mi listado de instalaciones se ve así:

```
$ kiex list

kiex elixirs

   elixir-1.4.5
   elixir-1.5.3
   elixir-1.6.6
=* elixir-1.7.4

# => - current
# =* - current && default
#  * - default
```

En estos momentos tan solo tengo las últimas versiones de las 4 últimas ramas. Puedo cambiar a cualquiera en cualquier momento con el comando:

```
$ kiex use 1.6.6
```

Este sistema es bastante simple y útil y es compatible con sistemas GNU/Linux y Mac.

4.2. Desde código fuente

Otro método de más bajo nivel sería instalar desde código fuente. No es muy complicado aunque sí requiere de más tiempo y un poco más de esfuerzo. En este caso tendríamos que ir a github para descargar el código fuente:

```
$ git clone https://github.com/elixir-lang/elixir
$ cd elixir
$ make clean test
```

Si todo va bien en el directorio `bin` encontraremos los comandos **elixir**, **elixirc** y **iex** entre otros. Podemos agregar esa ruta a nuestro PATH y así tendríamos el sistema instalado.

Ten en cuenta que no hemos cambiado de rama. Al descargar así Elixir estamos posicionados en la rama master que es lo último que se está haciendo con Elixir y posiblemente pueda no ser estable en algún punto.

Lo ideal sería ejecutar el siguiente comando antes de compilar para usa la versión 1.7.4:

```
$ git checkout v1.7.4
```

De esta forma nos aseguramos de estar construyendo esa versión. Además para actualizar cuando una nueva versión sea liberada (por ejemplo 1.7.5) solo nos requerirá realizar estos pasos:

```
$ git pull
$ git checkout v1.7.5
$ make clean test
```

No es una forma tan simple como las anteriores pero tampoco es complicada.

Apéndice B. La línea de comandos

El código de la mayoría de ejemplos del libro han sido desarrollados en la consola o línea de comandos. Elixir como lenguaje dentro de BEAM dispone de esta línea de comandos para facilitar su gestión y demostrar su versatilidad permitiendo conectar una consola a un nodo que se encuentre en ejecución y permitir al administrador obtener información del servidor en ejecución.

La línea de comandos es por tanto uno de los principales elementos de BEAM y Elixir. En este apéndice veremos las opciones que nos ofrece este intérprete de comandos para facilitar la tarea de gestión. Muchas de estas funciones ya se han ido mostrando a través de los capítulos del libro por lo que este compendio será una referencia útil para nuestro trabajo del día a día.

Importante

Las funciones que se listan a continuación están pensadas solo para la línea de comandos, no es intención de los desarrolladores el empleo de estas funciones en el código de un programa convencional. Hay funciones mejores para realizar cada una de estas acciones y a lo largo de este libro hemos visto muchas de estas funciones. Otras pueden ser encontradas en las guías de referencia de los módulos de Elixir y/o Erlang/OTP.

1. Módulos

Indicaremos todos los comandos referentes a la compilación, carga e información para los módulos:

c/1-2

Compila un fichero pasando su nombre como parámetro. El nombre proporcionado será un átomo con el nombre del módulo o una cadena de texto con el nombre del fichero, opcionalmente con su ruta. Como segundo parámetro podemos indicar la ruta donde se almacenará la compilación o el átomo ***:in_memory*** para no almacenarlo en ningún sitio.

r/1

Recompila y recarga un módulo pasando el nombre del fichero donde está contenido como cadena de texto. Todos los módulos

contenidos en el fichero serán recargados. El fichero original compilado (beam) no se verá afectado.

recompile/0

Recompila la aplicación principal cargada. Cuando trabajamos en un proyecto y tenemos la consola cargada y en ejecución con el código desarrollado, si realizamos un cambio en el código, en lugar de parar BEAM, recompilar y volver a lanzar, si los cambios son solo a nivel de ficheros de código (no configuración) podemos ejecutar este comando en consola y realizará la carga de todos los módulos que cambiaron.

2. Histórico

La consola dispone de un histórico que nos permite repetir comandos ya utilizados en la consola. El histórico es configurable y contendrá los últimos comandos tecleados. El símbolo del sistema (o prompt) nos indicará el número de orden que estamos ejecutando.

Además de los comandos, la consola de Elixir también almacena los últimos resultados. El número de resultados almacenados también es configurable.

Estas son las funciones que pueden emplearse:

v/1

Obtiene el resultado de la línea correspondiente pasada como parámetro.

v/0

Obtiene el resultado de la línea anterior.

Para repetir comandos podemos navegar por el histórico empleando los cursores, las flechas arriba y abajo así como la combinación de teclas **Ctrl**+**R** para buscar algún comando específico.

3. Procesos

Estas son funciones rápidas y de gestión sobre los temas que ya se revisaron en el Capítulo7, ***Procesos y Nodos***:

flush/0

Muestra todos los mensajes enviados al proceso de la consola.

pid/1,3

Obtiene el tipo de dato PID usando o una cadena de texto conteniendo el proceso en representación textual o tres parámetros numéricos dados para formar el proceso.

ref/1,4

Al igual que `pid/1,3` esta función nos permite convertir a una referencia o una cadena de texto con la representación textual de una referencia o los cuatro valores numéricos que conforman una referencia pasados como parámetros.

4. Directorio de trabajo

En cualquier momento podemos modificar el directorio de trabajo dentro de la consola. Las siguientes funciones nos ayudan en esta y otras tareas relacionadas:

cd/1

Cambia el directorio de trabajo. Se indica una lista de caracteres con la ruta relativa o absoluta para el cambio.

ls/0-1

Lista el directorio actual u otro indicado como parámetro de forma relativa o absoluta a través de una lista de caracteres.

pwd/0

Imprime el directorio de trabajo actual.

5. Ayudantes

Uno de los pilares de Elixir es la documentación. Todos los módulos compilados agregan la documentación a su módulo de forma que pueda ser obtenida en la consola. Tenemos estas funciones para ayudarnos a mostrar información sobre datos, módulos y funciones (coloreados y explicados):

b/1

Muestra las retro-llamadas a implementar para un módulo que define un comportamiento o la definición de una retro-llamada si es especificada. Muy útil cuando tenemos dudas sobre qué retro-llamadas deben implementarse para un comportamiento que usamos en otro módulo.

clear/0

Borra la pantalla. Nada más. Podemos conseguir normalmente lo mismo con la combinación de teclas **Ctrl**+**L**.

exports/1

Muestra las funciones públicas de un módulo. Conseguimos lo mismo si escribimos el módulo, un punto y presionamos la tecla **Tab** (tabulador).

h/0

Muestra el mensaje de ayuda para la línea de comandos.

h/1

Muestra el mensaje de ayuda para un módulo o función pasada como parámetro.

i/1

Muestra información detallada del dato pasado como parámetro.

i/0

Muestra información detallada del dato retornado por el último comando ejecutado en la consola.

open/1

Llama localmente al comando **open** para abrir el fichero fuente del código relativo al módulo que queremos editar.

runtime_info/0

Muestra información sobre el sistema. Información sobre las versiones de Elixir, Erlang, los parámetros de configuración principales de BEAM, memoria usada, CPU, tiempo en ejecución y más información.

6. Modo JCL

Cuando se presiona la combinación de teclas **Ctrl**+**G** se accede a una nueva consola. Esta consola es denominada JCL (***Job Control Mode*** o modo de control de trabajos). Este modo nos permite lanzar una nueva consola, conectarnos a una consola remota, detener una consola en ejecución o cambiar de una a otra consola.

Importante

Cada trabajo que se lanza es una consola (shell). Este modo nos permite gestionar estas consolas. Cada nodo puede tener tantas consolas como se quiera. No obstante el sistema está realizado con Erlang en mente por lo que la creación de una nueva shell lanzará una consola Erlang por defecto.

Estos son los comandos que podemos emplear en este modo:

c [nn]

Conectar a una consola. Si no se especifica un número vuelve al actual.

i [nn]

Detiene la consola actual o la que corresponda al número que se indique como argumento. Es útil cuando se quiere interrumpir un bucle infinito sin perder las variables empleadas.

k [nn]

Mata la consola actual o la que corresponda al número que se indique como argumento.

j

Lista las consolas en ejecución. La consola actual se indicará con un asterisco (*).

s [shell]

Inicia una consola nueva. Si se indica el nombre de un módulo como argumento se intentará lanzar un proceso con ese módulo como consola alternativa.

r [node [shell]]

Indica que deseamos crear una consola en un nodo al que se tiene conexión. Se lanza una consola en ese nodo y queda visible en el listado de consolas. Se puede indicar también una consola alternativa en caso de disponer de ella.

q

Finaliza la ejecución del nodo Erlang en el que estemos ejecutando el modo JCL.

7. Salir de la consola

Para salir de la consola hay varias formas. Se puede salir presionando dos veces la combinación de teclas **Ctrl**+**C** o a través del modo JCL y su comando ***q***.

Apéndice C. Compilador

La compilación es una tarea que hemos delegado 100% a Elixir y no hemos revisado ninguna configuración a lo largo del libro. En este apéndice vamos a dar una referencia de todas las opciones que podemos agregar y tener en cuenta para compilar nuestro código.

Estas opciones se pueden especificar dentro del fichero `mix.exs` en la configuración del proyecto a través de ***erlc_options***, a través del atributo de módulo ***@compile*** o si compilamos manualmente con **elixirc** a través de la opción ***--erl*** o las distintas opciones independientes. Puedes ejecutar **elixirc --help** para más información.

Si ejecutamos la función `Code.available_compiler_options/0` obtenemos todas las posibles opciones para la compilación:

:docs

Indica compilar incluyendo la documentación. Puede ser útil para versiones de producción sea extraño el acceso de administradores o no se requiera. Por defecto está activado.

:debug_info

Indica agregar información para depuración dentro del fichero compilado. Útil para realizar trazas y depurar. Por defecto está activado.

:ignore_module_conflict

Nos evita ver el mensaje cuando un módulo es redefinido. Normalmente no aparece en compilaciones normales a menos que sobreescribamos un módulo existente dentro de nuestro código. Por defecto está desactivado.

:relative_paths

Principalmente presenta los ficheros en anotaciones (logs) con ruta relativa en lugar de absoluta en caso de estar activado. Por defecto está activado.

:warnings_as_errors

Detiene la compilación generando un error si aparece un aviso (warning). Esta opción fuerza al código a presentarse libre de mensajes de aviso. Por defecto está desactivado.

1. Configuración en `mix.exs`

Por defecto si no elegimos nada la configuración tomada es tal que así:

```
def project do
  [
    app: :cuatro,
    version: "1.0.0",
    elixir: "~> 1.7",
    elixirc_paths: ["lib"],
    start_permanent: true,
    deps: deps(),
    erlc_options: [docs: true,
                   ignore_module_conflict: false,
                   debug_info: true,
                   warnings_as_errors: false,
                   relative_paths: true],
  ]
end
```

Puedes modificarlo de este modo para tener una configuración según el entorno:

```
def project do
  [
    app: :cuatro,
    version: "1.0.0",
    elixir: "~> 1.7",
    elixirc_paths: ["lib"],
    start_permanent: true,
    deps: deps(),
    erlc_options: erlc_options(Mix.env()),
  ]
end

def erlc_options(:prod) do
  [
    docs: false,
    ignore_module_conflict: false,
    debug_info: false,
    warnings_as_errors: true,
    relative_paths: false
  ]
end
def erlc_options(_) do
  [
    docs: true,
    ignore_module_conflict: false,
    debug_info: true,
    warnings_as_errors: true,
    relative_paths: true
  ]
end
```

Por mi parte considero una buena práctica tener siempre activados los mensajes de aviso como si se tratara de errores y así forzar a tener una mayor limpieza en el código.

2. Configuración para un módulo específico

Si necesitamos por ejemplo desactivar los errores por mensajes de avisos o no queremos agregar documentación para un módulo específico podemos agregar estas opciones directamente en el módulo a través del atributo ***@compile*** tal y como vimos en el Capítulo4, ***Las funciones y los módulos***.

En el ejemplo de no querer ni documentación ni mensajes de aviso como errores podemos agregar la línea:

```
@compile [warnings_as_errors: false, docs: false]
```

De esta forma conseguimos saltar las opciones generales para el módulo.