Para facilitar la reutilización del software, Elixir proporciona tres directivas (alias, requerir e importar) más una macro llamada uso resumida a continuación:

```
# Alias del módulo para que se pueda llamar como Bar en lugar de Foo.Bar
alias Foo.Bar, as: Bar

# Requiere que el módulo use sus macros
require Foo

# Importa funciones de Foo para poder llamarlas sin el prefijo `Foo.`
import Foo

# Invoca el código personalizado definido en Foo como un punto de extensión
use Foo
```

alias

Permite configurar alias para cualquier nombre de módulo dado.

Imagine que un módulo utiliza una lista especializada implementada en Math.List. La directiva alias permite hacer referencia a Math.List como List dentro de la definición del módulo:

```
defmodule Stats do
  alias Math.List, as: List
end
```

Es igual que:

```
defmodule Stats do
  alias Math.List
end
```

Ten en cuenta que el alias tiene un ámbito léxico, lo que le permite establecer alias dentro

de funciones específicas:

```
defmodule Math do
  def plus(a, b) do
    alias Math.List
    # ...
  end

  def minus(a, b) do
    # ...
  end
end
```

En el ejemplo anterior, estamos invocando el alias dentro de la función plus/2, el alias será válido solo dentro de la función plus/2. minus/2 no se verá afectado en absoluto.

require

Elixir proporciona macros como mecanismo para la metaprogramación (escribir código que genera código). Las macros se expanden en tiempo de compilación.

Las funciones públicas en los módulos están disponibles globalmente, pero para usar macros, debe optar por solicitar el módulo en el que están definidas.

```
iex> Integer.is_odd(3)
** (CompileError) iex:1: you must require Integer before invoking the macro Integer.is_odd/1
    (elixir) src/elixir_dispatch.erl:97: :elixir_dispatch.dispatch_require/6
iex> require Integer
Integer
iex> Integer.is_odd(3)
true
```

En Elixir, Integer.is_odd/1 se define como una macro para que pueda usarse como guard. Esto significa que, para invocar Integer.is_odd/1, primero necesitamos el módulo Integer.

Tenga en cuenta que, al igual que la directiva alias, require también tiene un alcance léxico. Hablaremos más sobre macros en un capítulo posterior.

import

Usamos import siempre que deseamos acceder a funciones o macros de otros módulos sin usar el nombre completo. Tenga en cuenta que solo podemos importar funciones públicas, ya que las funciones privadas nunca son accesibles externamente.

Por ejemplo, si queremos usar la función duplicate/2 del módulo List varias veces, podemos importarla:

```
iex> import List, only: [duplicate: 2]
List
iex> duplicate :ok, 3
[:ok, :ok, :ok]
```

Importamos solo la función duplicate/2 de List. Aunque: solo es opcional, se recomienda su uso para evitar importar todas las funciones de un módulo dado dentro del alcance actual. :except también podría darse como una opción para importar todo en un módulo, excepto una lista de funciones.

Ten en cuenta que la importación también tiene un ámbito léxico. Esto significa que podemos importar macros o funciones específicas dentro de las definiciones de funciones:

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    duplicate(:ok, 10)
  end
end
```

En el ejemplo anterior, el List.duplicate/2 importado solo es visible dentro de esa función específica. duplicate/2 no estará disponible en ninguna otra función en ese módulo (o cualquier otro módulo para el caso).

Tenga en cuenta que la importación de un módulo lo requiere automáticamente.

use

La macro de use con frecuencia es un punto de extensión. Esto significa que, cuando usa un módulo FooBar, permite que ese módulo inyecte cualquier código en el módulo actual, como importarse a sí mismo u otros módulos, definir nuevas funciones, establecer un estado del módulo, etc.

Por ejemplo, para escribir pruebas usando el marco ExUnit, un desarrollador debe usar el módulo ExUnit.Case:

```
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    assert true
  end
end
```

Detrás de escena, el uso requiere el módulo dado y luego llama a la devolución de llamada __using__/1, lo que permite que el módulo inyecte algo de código en el contexto actual. Algunos módulos (por ejemplo, el ExUnit.Case anterior, pero también Supervisor y GenServer) usan este mecanismo para llenar su módulo con un comportamiento básico, que su módulo está destinado a anular o completar.

En general, el siguiente módulo:

```
```

se compila en

```
defmodule Example do
  require Feature
  Feature.__using__(option: :value)
end
```

Como el uso permite que se ejecute cualquier código, no podemos conocer realmente los efectos secundarios del uso de un módulo sin leer su documentación. Por esta razón, a menudo se prefiere la importación y el alias, ya que su semántica está definida por el lenguaje.

Entendiendo los alias

Un alias en Elixir es un identificador en mayúscula (como String, Keyword, etc.) que se convierte en un átomo durante la compilación. Por ejemplo, el alias String se traduce por defecto al átomo: «Elixir.String»:

```
iex> is_atom(String)
true
iex> to_string(String)
"Elixir.String"
iex> :"Elixir.String" == String
true
```

Al usar la directiva alias/2, estamos cambiando el átomo al que se expande el alias.

Los alias se expanden a átomos porque en los módulos Erlang VM (y, en consecuencia, Elixir) siempre están representados por átomos. Por ejemplo, ese es el mecanismo que usamos para llamar a los módulos de Erlang:

```
iex> :lists.flatten([1, [2], 3])
[1, 2, 3]
```

Anidamiento de módulos

Fíjate en el siguiente ejemplo:

```
defmodule Foo do
  defmodule Bar do
  end
end
```

El ejemplo anterior definirá dos módulos: Foo y Foo.Bar. Se puede accedeFrom Elixir v1.2, it is possible to alias, import or require multiple modules at once. This is particularly useful once we start nesting modules, which is very common when building Elixir applications. For example, imagine you have an application where all modules are nested under MyApp, you can alias the modules MyApp.Foo, MyApp.Bar and MyApp.Baz at once as follows:r al segundo como Bar dentro de Foo siempre que estén en el mismo ámbito léxico. El código anterior es exactamente el mismo que:

```
defmodule Elixir.Foo do
  defmodule Elixir.Foo.Bar do
  end
  alias Elixir.Foo.Bar, as: Bar
end
```

O que:

```
defmodule Elixir.Foo do
  alias Elixir.Foo.Bar, as: Bar
end

defmodule Elixir.Foo.Bar do
```

```
end
```

Multi alias/import/require/use

Desde Elixir v1.2, es posible crear alias, importar o requerir múltiples módulos a la vez. Esto es particularmente útil una vez que comenzamos a anidar módulos, lo cual es muy común al construir aplicaciones Elixir. Por ejemplo, imagine que tiene una aplicación en la que todos los módulos están anidados en MyApp, puede crear un alias de los módulos MyApp.Foo, MyApp.Bar y MyApp.Baz de la siguiente manera:

```
alias MyApp.{Foo, Bar, Baz}
```