
توثيق الإكسير

يطلق

مجتمع الإكسير

18 سبتمبر 2017

محتويات

1 دليل البدء		3
1.1 مقدمة عن الإكسير		3
المقدمة		3
1.1.1 الأنواع الأساسية		4
1.1.2 المترافقون		4
1.1.3 المترافقون الأساسية		10
1.1.4 المطابقة الأنماط		12
1.1.5 الحال والشرط وإذا		15
1.1.6 الثنائيات والسلسل وقوائم		9
1.1.7 الكلمات الأساسية والخواص والقواميس		1.8
1.1.8 الوحدات النمطية		8
1.1.9 المكتار		19
1.1.10 العناصر القابلة للعد والتذبذبات		10
1.1.11 المكتار		22
1.1.12 الإدخال/الإخراج ونظام الملفات		26
1.1.13 الأسم المستعار والطلب والاستيراد		30
1.1.14 اسماء الوحدة		32
1.1.15 الهياكل		34
1.1.16 بروتوكولات فهم		39
1.1.17 بروتوكولات فهم		42
1.1.18 البرموز		45
1.1.19 المحاولة والإمساك والإنقاذ		49
1.1.20 مواصفات النوع والسلوكيات		50
1.1.21 مقدمة إلى Elixir		54
1.2 Agent		55
1.2.1 مقدمة إلى Mix		59
1.2.2 Mix		62
1.2.3 GenServer		65
1.2.4 GenEvent		65
1.2.5 المشرف والتطبيق		69
1.2.6 ETS		73
1.2.7 التبعيات والمشاريع الشاملة		78
1.2.8 مهمية gen_tcp		82
1.2.9 المستندات والاختبارات وخطوط الأنابيب		89
1.2.10 المهام الموزعة والتكون		98
1.3 إقليدس في Elixir		103
1.3.1 البرمجة الفوقيّة في Elixir		108
1.3.2 وحدات الماكرو		116
1.3.3 إقليدس وإلغاء الاقتراض		124
1.3.4 إقليدس وإلغاء الاقتراض		124
1.3.5 وحدات الماكرو		126

1.3.3 لغات خاصة بال المجال131
2.1 أدلة تقنية	135
2.1.1 أنواع النطاق	136
2.1.2 فضلات Erlang هي معجمية Elixir	138
2.1.3 تعشيش النطاق والتطبيق	138
2.1.4 نطاق المستوى الأعلى	138
2.1.5 نطاق جملة الوظيفة	139
2.1.6 الوظائف والوحدات المسماة	140
2.1.7 جمل حالة الأحرف	142
2.1.8 ككل المحاولة	143
2.1.9 الفهم	143
2.1.10 الطلب والاستيراد والاسم	144
2.1.11 المسبعار	144
الاختلافات عن إرلانج	144

هذه تجربة لتنظيم أنواع مختلفة من وثائق Elixir بطريقة جديدة.

تم إنشاء هذا الموقع من هذه النسخة من المصدر.

الفصل الأول

دليل البدء

مقدمة عن الإكسير

المقدمة 1

toc.html %} تشمل

مرحباً!

في هذا البرنامج التعليمي، ستعلمك أساسيات Elixir، وقواعد اللغة، وكيفية تعريف الوحدات النمطية، وكيفية التعامل مع خصائص هيكل البيانات الشائعة والمزيد. سيركز هذا الفصل على التأكد من تثبيت Elixir وتمكنك من تشغيل Elixir Interactive Shell Elixir Shell، والذي يسمى IEx.

متطلباتنا هي:

Erlang 17.0 وما بعده -

Elixir 1.0.0 وما بعده -

دعونا نبدأ!

إذا وجدت أي أخطاء في البرنامج التعليمي أو على موقع الويب، فيرجي الإبلاغ عن وجود خلل أو إرسال طلب سحب إلى متعقب المشكلات لدينا. إذا كنت تشك في أن الأمر يتعلق بخلل في اللغة، فيرجي إعلامنا بذلك في متعقب مشكلات اللغة.

1.1 التثبيت

إذا لم تقم بتثبيت Elixir بعد، فانتقل إلى صفحة التثبيت الخاصة بنا، بمجرد الانتهاء، يمكنك تشغيل elixir v-7 للحصول على إصدار Elixir الحالي.

1.2 الوضع التفاعلي

عند تثبيت Elixir، سيكون لديك ثلاثة ملفات قابلة للتنفيذ جديدة: elixir و elixirc و elixir.ex. إذا قمت بتنصيب إصداراً معيناً، فيمكنك العثور على هذه الملفات داخل دليل bin.

في الوقت الحالي، نبدأ بتشغيل Elixir (أو iex) إذا كنت تستخدم نظام التشغيل Windows والذى يرمى إلى Interactive Elixir، يمكننا كتابة أي تعليم Elixir والحصول على نتيجته. دعنا نبدأ بعض التعبيرات الأساسية:

الإكسير التفاعلي - اضغط على **Ctrl+Enter** للحصول على المساعدة

ابكx < 2

42

<> "مرحبا" " العالم"

"مرحبا بالعالم"

يبدو أننا جاهزون للانطلاق! سوف نستخدم واجهة التفاعل كثيراً في الفصول التالية لتعرف على تركيب اللغة والأنواع الأساسية بشكل أفضل، بدءاً من الفصل التالي.

1.3 تشغيل البرامج النصية

بعد التعرف على أساسيات اللغة، قد ترغب في محاولة كتابة برامج بسيطة. يمكن تحقيق ذلك من خلال وضع كود Elixir في ملف وتفيذه باستخدام elixir:

Elixir "\$ cat simple.exs IO.puts

ابكx simple.exs

من الإكسير

سنتعلم لاحقاً كيفية تجميع كود Elixir (وكيفية استخدام أداة بناء Mix & OTP). أما الان، فلننتقل إلى الفصل 2.

2 أنواع أساسية

toc.html %} تشمل

في هذا الفصل سوف نتعلم المزيد عن أنواع إلخ Elixir الأساسية: الأعداد الصحيحة، والأعداد العائمة، والقيم المنطقية، والذرارات، والسلسلات، والقوائم، وال الثنائيات. بعض الأنواع الأساسية هي:

```
ابكx < 1 # عدد صحيح
ابكx < 0x1F # عدد عشري
ابكx < 1.0 # عددي عائمة
ابكx < :atom # منطقية
ابكx < :atom # الذرات
ابكx < "elixir" # سلسلة
ابكx < "elixir" # رمز
ابكx < [1, 2, 3] # قائمة
ابكx < {1, 2, 3} # ترجمة
```

2.1 الحساب الأساسي

افتح iex واتكتب التعبيرات التالية:

ابكx < 1 + 2

3

ابكx < 5 * 5

25

ابكx < 10 / 2 5.0

لاحظ أن 2 / 10 أرجعنا عدداً عشرة 5.0 بدلاً من عدد صحيح 5. وهذا متوقع، في Elixir، بعيد المشغل / دائماً عدداً عشرة، إذا كنت تريد إجراء قسمة عدد صحيح أو الحصول على باقي القسمة، فيمكنك استدعاء الدالدين: div و rem:

```
iex> div(10, 2) 5
```

```
iex> div 10, 2 5
```

```
iex> rem 10, 3  
1
```

لاحظ أنه لا يلزم وجود أقواس لاستدعاء وظيفة.

يدعم Elixir أيضاً اختصارات التدوين لإدخال الأرقام الثنائية والثمانية والسداسية عشرية:

ابكس <	Ob1010
10	
ابكس >	0o777
511	
ابكس <	0x1F
31	

تطلب الأرقام العائمة نقطة متبوعة برقم واحد على الأقل وتدعم أيضاً عرقم الأس:

ابكس <	1.0
	1.0
ابكس >	1.0e-10
	1.0e-10

تتميز العواملات في Elixir بدقّة مزدوجة تبلغ 64 بت.

يمكنك استدعاء الدالة round للحصول على أقرب عدد صحيح إلى عدد عشري معين، أو الدالة trunc للحصول على جزء العدد الصحيح من عدد عشري.

3.58 الجولة iex>	4
1.0e-10	
ابكس >	3

القيم المنطقية 2.2

يدعم Elixir القيم المنطقية: eslaf، true، false

صحيح iex>	حقفي
صحيح iex>	خطأ ==
خطأ شنبع	

يوفّر Elixir مجموعة من وظائف التنبؤ للتحقق من نوع القيمة. على سبيل المثال، يمكن استخدام وظيفة boolean/1 للتحقق مما إذا كانت القيمة منطقية أم لا:

ملاحظة: يتم تحديد الدوال في Elixir بالاسم وعدد الوسائط (أي). Elixir، يحدد 1 is_boolean/1، أي دالة تسمى is_boolean، وذلك وسيلة واحدة. يحدد 2 is_booleanarity/1، أي دالة مختلفة (غير موجودة) بنفس الاسم ولكن arity different.

iex> is_boolean(true)	حقفي

iex> is_boolean(1) false

يمكنك أيضًا استخدام `is_integer` أو `is_float` أو `is_number` للتحقق مما إذا كانت الوسيطة عدداً صحيحاً أو عدداً عشررياً أو كليهما على التوالي.

ملاحظة: يمكنك في أي وقت كتابة في shell معلومات حول كيفية استخدام shell، يمكن أيضًا استخدام مساعد whois إلى الوثائق الخاصة بأي دالة.

الذرات 2.3

لذرات عبارة عن ثوابت حيث أن اسمها هو قيمتها الخاصة. بعض اللغات الأخرى تسمى هذه الرموز:

خطأ شبيه

لقيم المنطقية true و false في الواقع ذرات:

`==`: صحيح `<==`: صحيح

١٢٣

تم ادراجه في السلاسل، علامت، اقتباس، مزدوجتين، و يتم تمييزها بتنسق.

"مرحبا"iex>

65001 لاحظة: إذا كنت تستخدم نظام التشغيل Windows، فهناك احتمال أن يستخدم محيطك الطرفي ترميز UTF-8 افتراضياً. يمكنك تغيير ترميز جلستك الحالية بتشغيل الأمر ``chcp``.

يدعم Elixir أيضًا استيفاء السلسلة:

```
jexl> "hello #{'world'}" "hello World"
```

يمكن أن تحتوي السلسل على فوائل للأسطر أو تقديمها باستخدام تسلسلات الهروب:

```
ielx> "hello\nworld" "hello\nworld"  
"hello\nworld"
```

يمكنك طباعة سلسلة باستخدام الدالة `IO.puts/1`:

```
iex> IO.puts "hello\nworld" hello
```

نعم
نعم:

لاحظ أن الدالة `IO.puts/1` تعيد الذرة `\n` كنتيجة بعد الطباعة.

يتم تمثيل السلسل في Elixir داخليا بواسطة ثنائيات عبارة عن تسلسلات من البيانات:

```
(مرحبا")yranib_siex>
```

حققي

يمكننا أيضًا الحصول على عدد البيانات في السلسلة:

```
6 ("مرحبا")ezis_etybiex>
```

لاحظ أن عدد البيانات في هذه السلسلة هو 6، على الرغم من أنها تحتوي على 5 أحرف. وذلك لأن الحرف "ة" يستغرق 2 بait لتمثيله في UTF-8. يمكننا الحصول على الطول الفعلي للسلسلة، بناءً على عدد الأحرف، باستخدام الدالة `String.length/1`:

```
("مرحبا")htgnel.gnirtSiex>
```

5

تحتوي وحدة `String` على مجموعة من الوظائف التي تعمل على السلسل كما هو محدد في معيار Unicode:

```
(مرحبا")esacpu.gnirtSiex>
```

"مرحبا"

2.5 وظائف مجهولة الهوية

يتم تحديد الوظائف بواسطة الكلمات المفاتيحية `fn` و `end`:

```
add = fn a, b -> a + b end #Function<12.71889879/2 in :erl_eval.expr/5> iex> is_function(add) true
```

iex>

`(2) سمح` (اصافة، 2)

`(1) خطأ` (اصافة، 1)

`(1, 2) 3` (اصافة، 3)

تعتبر الدوال "مواطين من الدرجة الأولى" في Elixir، وهذا يعني أنه يمكن تمريرها كحجج إلى دوال أخرى تماماً كما يمكن تمرير الأعداد الصحيحة والسلسل. في المثال، مررت الدالة `is_function/2` إلى المتغير `add` وإلى الدالة `is_function/2` التي أعادت القيمة `true` وبشكل صحيح. يمكننا أيضاً التحقق من عدد الدوال من خلال استدعاء `length/1`:

لاحظ أنه يلزم وجود نقطة(.) بين المتغير والأقواس لاستدعاء وظيفة مجهولة.

الوظائف المجهولة هي عمليات إغلاق، وبالتالي يمكنها الوصول إلى المتغيرات التي تقع ضمن النطاق عند تعريف الوظيفة:

```
add_two = fn a -> add.(a, 2) end #Function<6.71889879/1 in :erl_eval.expr/5> iex> add_two.(2) 4
```

iex>

ضع في اعتبارك أن المتغير المخصص داخل الدالة لا يؤثر على البيئة المحيطة به:

```
iex> x = 42
```

```
42
```

```
()).(النهاية) iex> (fn -> x = 0
```

```
0
```

```
x < ايكس
```

```
42
```

(القوائم المرتبطة) 2.6

يستخدم Elixir أقواساً مربعة لتحديد قائمة القيم. يمكن أن تكون القيم من أي نوع:

```
3] [1, 2, 3] iex> [1, 2,
```

```
[1, 2, 3] 3] iex>
```

يمكن ربط قائمتين وطرحهما باستخدام عامل `/2`:

```
[1, 2, 3, 4, 5, 6] iex> [1, 2, 3] ++ [4, 5, 6] iex> [1,
```

خلال البرنامج التعليمي، ستحدث كثيراً عن رأس وذيل القائمة. الرأس هو العنصر الأول في القائمة والذيل هو باقي القائمة. يمكن استرجاعهما باستخدام الداللين `hd/1` أو `tl/1`:

```
= [1,2,3] iex>
```

```
(القائمة) dhiex>
```

```
1
```

```
[2, 3] (القائمة) tiiex>
```

الحصول على رأس أو ذيل قائمة فارغة هو خطأ:

```
hd[]
```

```
ايكس
```

```
* خطأ في الحجة (ArgumentError)
```

في بعض الأحيان، ستقوم بإنشاء قائمة وستقوم بإرجاع قيمة بين علامتي اقتباس مفردين. على سبيل المثال:

```
iex> [11, 12, 13] '٧٨٩١٣'
```

```
أي> [104, 101, 108, 108, 111]
```

```
'مرحبا'
```

عندما يرى Elixir قائمة بأرقام ASCII القابلة للطباعة، فسوف يقوم بطبعها ذلك كقائمة أحرف (قائمة أحرف حرفية).

قوائم الأحرف شائعة جدًا عند التفاعل مع كود Erlang الحالي.

ضع في اعتبارك أن التمثيلات ذات الاقتباس المفرد والمزدوج ليست متكافئة في Elixir حيث يتم تمثيلها بواسطة أنواع مختلفة:

```
"مرحبا" == "مرحبا" iex>
```

```
خطأ شنيع
```

علامات الاقتباس المفردة هي قوائم أحرف، وعلامات الاقتباس المزدوجة هي سلاسل. سنتحدث أكثر عنهما في الفصل "الثنائيات والسلالس وقوائم الأحرف".

2.7 الثنائيات

يستخدم Elixir الأقواس المترعرجة لتحديد العناصر المتطابقة. ومثل القوائم، يمكن للعناصر المتطابقة أن تحتوي على أي قيمة:

```
iex> {:ok, "hello"} {:ok, "hello"} iex> tuple_size {:ok, "hello"} 2
```

تخزن العناصر في مجموعات متباينة في الذاكرة. وهذا يعني أن الوصول إلى عنصر مجموعه لكل فهرس أو الحصول على حجم المجموعة عملية سريعة (تبدأ الفهارس من الصفر):

```
iex> tuple = {:ok, "hello"} {:ok, "hello"} iex> elem(tuple, 1)
```

"مرحبا"

```
iex> tuple_size(tuple) 2
```

من الممكن أيضًا تعيين عنصر عند مؤشر معين في مجموعة باستخدام `put_elem/3`:

```
put_elem(tuple, 1, "world") {:ok, "world"} iex> tuple {:ok, "hello"}  
iex> tuple = {:ok, "hello"} {:ok, "hello"} iex>
```

لاحظ أن `put_elem/3` أعاد مجموعة جديدة. المجموعة الأصلية المخزنة في متغير المجموعة لم يتم تعديلها لأن أنواع بيانات Elixir غير قابلة للتغيير. ولأنها غير قابلة للتغيير، فإن كود Elixir أسهل في التفكير فيه لأنك لن تحتاج أبداً إلى القلق بشأن ما إذا كان كود معين يغير بنية البيانات لديك في مكانها.

بفضل كونه غير قابل للتغيير، يساعد Elixir أيضاً في التخلص من الحالات الشائعة التي يكون فيها الكود المتزامن به ظروف سباق لأن كيانين مختلفين يحاولان تغيير بنية البيانات في نفس الوقت.

2.8 القوائم أو المجموعات؟

ما هو الفرق بين القوائم والمجموعات؟

يتم تخزين القوائم في الذاكرة كقوائم مرتبطة، مما يعني أن كل عنصر في القائمة يحتفظ بقيمه ويشير إلى العنصر التالي حتى الوصول إلى نهاية القائمة. نطلق على كل زوج من القيمة `cons`: والمؤشر خلية

```
iex> [1 | 2 | 3 | []]] [1, 2, 3]
```

هذا يعني أن الوصول إلى طول القائمة هو عملية خطية: تحتاج إلى اجتياز القائمة بأكملها لمعرفة حجمها. تحديث القائمة سريع طالما أنها نضيف عناصر مسبقاً:

```
قائمة [0, 1, 2, 3] iex> [0] ++ [4] قائمة [0, 1, 2, 3]
```

```
[1, 2, 3, 4]
```

العملية الأولى سريعة لأننا ببساطة نضيف عنصراً جديداً يشير إلى ما تبقى من القائمة. أما العملية الثانية فهي بطيئة لأننا نحتاج إلى إعادة بناء القائمة بالكامل وإضافة عنصر جديد إلى النهاية.

من ناحية أخرى، يتم تخزين العناصر الممتالية في الذاكرة كشكل متاجور. وهذا يعني أن الحصول على حجم العنصر أو الوصول إلى عنصر من خلال الفهرس يتم بسرعة. ومع ذلك، فإن تحديث العناصر أو إضافةها إلى العناصر الممتالية أمر مكلف لأنّه يتطلب نسخ العنصر بالكامل في الذاكرة.

١- تحدد خصائص الأداء هذه استخدام هيكل البيانات هذه. ومن بين حالات الاستخدام الشائعة جدًا للمجموعات استخدامها لإرجاع معلومات إضافية من دالة. على سبيل المثال، `File.read()` هي دالة يمكن استخدامها لقراءة محتويات الملف وهي ترجع مجموعات:

```
iex> File.read("path/to/unknown/file") {:error, :enoent} {"..."} iex> File.read("path/to/existing/file") {:ok, "..."}
```

إذا كان المسار المحدد لملف `1/read` موجوداً، فإنه يعيد مجموعة تحتوي على الذرة `:ko` كعنصر أول ومحطيات الملف كعنصر ثانٍ. وإلا، فإنه يعيد مجموعة تحتوي على `:orre` ووصف الخطأ.

في أغلب الأحيان، سيرشدك Elixir إلى القيام بالشيء الصحيح. على سبيل المثال، توجد دالة `elem` للوصول إلى عنصر من عناصر المجموعة، ولكن لا يوجد معادل مدمج للقواعد:

```
iex> tuple = {:ok, "hello"} {:ok, "hello"} iex> elem(tuple, 1)
```

مِنْهَا

عدد "عد" العناصر في بنية البيانات، يلتزم Elixir أيضًا بقاعدة بسيطة: يجب تسمية الوظيفة بالحجم إذا كانت العملية في وقت ثابت (أي أن القيمة محسوبة مسبقاً) أو الطول إذا كانت العملية تتطلب العدد الصريح.

على سبيل المثال، لقد استخدمنا حتى الآن `length` و`charAt` و`getBytes` في السلاسلة، `length` و`getBytes` و`charAt` هما مجموعتين من الأطوال القائمة، `length` يستخدم على العدد الآخر في السلاسلة، `getBytes` يستخدم على عدد البايتات في السلاسلة، وهو أمر رخيص، ولكن استرداد عدد أحرف `String.length` يطلب أداءً أعلى، السلاسلة أكملها تحتاج إلى التفكير.

يتوفر Elixir أيضًا وDIP وPort وReference كأنواع بيانات (تستخدم عادةً في اتصالات العمليات). وسنلقي نظرة سريعة عليها عند الحديث عن العمليات. في الوقت الحالي، دعنا نلقي

3 مشغلات أساسية

toc.html %} .lo{v{\%

وفـ Elixir أـنـضا ++-- للـتـلاـعـبـ بـالـقـوـائـمـ:

[1,3] **أیکس** < [1,2,3] ++ [4,5,6] [1,2,3,4,5,6] **أیکس** < [2]

يتم ربط السلسلة باستخدام:<>

iex> "foo" <> "bar"

يتوفر Elixir أيضًا ثلاثة عوامل منطقية: `not` و `or` و `and`. وهذه العوامل صارمة بمعنى أنها تتوقع قيمة منطقية (صواب أو خطأ) كحجج أولى لها:

```
iex> is_atom(example) true او iex> false
حقيني
```

سيؤدي توفير قيمة غير منطقية إلى إثارة استثناء:

```
iex> 1 او صحيحة
خطأ في الحجة ***(ArgumentError)
```

أو و هي عوامل تشغيل دائرة قصر، فهي تنفذ الجانب الأيمن فقط إذا لم يكن الجانب الأيسر كافيًّا لتحديد النتيجة:

```
iex> (ان يتم رفع هذا الخطأ أبداً) ororre false
خطأ شنيع
```

```
iex> (ان يتم رفع هذا الخطأ أبداً) ororre true
حقيني
```

ملاحظة: إذا كنت مطورو Erlang، فإن `or` و `and` في Elixir يتم تعبيئهما فعليًا إلى مشغل `eslero` و `andalso` في.

بالإضافة إلى هذه العوامل المنطقية، توفر Elixir أيضًا `|` و `&&` التي تقبل الوسائط من أي نوع. بالنسبة لهذه العوامل، سيتم تقييم جميع القيم باستثناء `nil` و `false` على أنها:

```
# او
1 || صحيحة 1 ||
iex> false || 11
11

# لا شيء و 13
iex> لا شيء
لا شيء
17
17

# ! iex> !true
خطأ شنيع
!ايكس>
خطأ شنيع
! ايكس
خطأ شنيع
! ايكس>
حقيني
```

كقاعدة عامة، استخدم `and` و `or` عندما تتوقع قيمة منطقية، إذا كانت أي من الحجج غير منطقية، فاستخدم `||` و `&&`.

يتوفر Elixir أيضًا <, ==, !=, ===, !==, >=, == و <= كمشغلات مقارنة:

```
1 == 1
حقيني
iex> 1 != 2
حقيني
1 < 2
ايكس
حقيني
```

الفرق بين == و === هو أن الأخير أكثر صرامة عند مقارنة الأعداد الصحيحة والأعداد العشرية:

```
iex> 1 == 1.0
قيفي
iex> 1 === 1.0
خطأ شنيع
```

في Elixir، يمكننا مقارنة نوعين مختلفين من البيانات:

```
iex> 1 < :atom
قيفي
```

السبب الذي يجعلنا قادرين على مقارنة أنواع مختلفة من البيانات هو البراجماتية. لا تحتاج خوارزميات الفرز إلى القلق بشأن أنواع مختلفة من البيانات من أجل الفرز. يتم تعريف ترتيب الفرز الإجمالي أدناه:

```
الرقم < الكرة < المرجع < الدوال < المنفذ < معرف العملية < المجموعة < الخرائط < القائمة < سلسلة البيانات
```

لا تحتاج فعليًا إلى حفظ هذا الترتيب، ولكن من المهم فقط معرفة وجود ترتيب.

حسناً، هذا كل ما في المقدمة. في الفصل التالي، سنناقش بعض الوظائف الأساسية وتحويلات أنواع البيانات وبعض تدفق التحكم.

4 مطابقة الأنماط

`toc.html %` تشمل

في هذا الفصل، سنوضح كيف أن عامل `=` في الواقع عامل مطابقة وكيفية استخدامه لمطابقة النمط داخل هيكل البيانات. وأخيراً، سنتعلم عن عامل `^` المستخدم للوصول إلى القيم المرتبطة سابقًا.

4.1 عامل المطابقة

لقد استخدمنا عامل `=` عدة مرات لتعيين المتغيرات في Elixir:

```
iex> x = 1
1
iex> x
1
```

في Elixir، يُطلق على عامل `=` في الواقع عامل المطابقة. دعنا نرى السبب:

```
iex> 1 = x
1
iex> 2 = x
1** لا يوجد تطابق لقيمة الجانب الأيمن : (MatchError)
```

لاحظ أن `x = 1` عبارة صالحة، وقد تطابقت لأن الجانبين الأيسر والأيمن يساويان. عندما لا يتطابق الجانبان، يتم رفع خطأ `MatchError`.

لا يمكن تعيين متغير إلا على الجانب الأيسر من `=`:

```
iex> 1 =
غير معروف** (RuntimeError) دالة غير محددة : غير معروفة/0
```

نظرًا لعدم وجود متغير غير معروف تم تعريفه مسبقًا، يتخيل Elixir أنك تحاول استدعاء دالة باسم `unknown/0`، ولكن مثل هذه الدالة غير موجودة.

مطابقة الأنماط 4.2

لا يُستخدم عامل المطابقة للمطابقة مع القيم البيسطية فحسب، بل إنه مفيد أيضًا لتفكيك أنواع البيانات الأكثر تعقيدًا. على سبيل المثال، يمكننا مطابقة الأنباط على الثنائيات:

```
iex> {a, b, c} = {:hello, "world", 42} {:hello, "world", 42} iex> a
```

ستحدث أخطاء في مطابقة النمط في حالة عدم تطابق الجوانب، على سبيل المثال، هذه هي الحالة عندما تكون أحجام الثنائيات مختلفة:

```
{"world": "العالٰم", "hello": "مرحبا"}  
MatchError: لا يوجد تطابق لقيمة الجانب الأيمن :(*hello, "world")
```

وأيضا عند مقارنة الأنواع المختلفة:

لا يوجد تطابق لقيمة الجانب الأيمن : [hello, "world", "!"] (ModelError)

الأمر الأكثر إثارة لاهتمام هو أنه يمكننا المطابقة على قيم محددة. يؤكد المثال أدناه أن الجانب الأيسر سوف يتتطابق فقط مع الجانب الأيمن عندما يكون الجانب الأيمن عبارة عن مجموعة تبدأ بالذرة:

= {error, :oops} {النتيجة} {ok, 13} {ok, 13} {النتيجة} {ok, 13}

= {error, :oops} {النتيجة} {ok, 13} {ok, 13} {النتيجة} {ok, 13}

لا يوجد تطابق لقيمة الجانب الأيمن : {error, :oops} (***) (MatchError)

يمكننا مطابقة النمط على القوائم:

أيكس<.j> [1, 2, 3] [1, 2, 3] [ج, ب, .j]> |

ندعم القائمة أيضاً المطابقة على رأسها وذيلها:

```
= [1, 2, 3] [1, 2, 3] [الرأس | الذيل]iex>  
الرأسiex>  
1  
الذيلiex>  
[2, 3]
```

على غرار وظائف `1/1` أو `hd/1` يمكننا مطابقة قائمة فارغة بنمط رأس وذيل:

iex> [h | t] = []
لا يوجد تطابق لقيمة الجانب الأيمن: [] (MatchError)

لا يتم استخدام تنسيق [الرأس | الذيل] في مطابقة الأنماط فحسب، بل أيضاً لإضافة عناصر إلى القائمة:

```
= [1, 2, 3] [1, 2, 3] iex> [0 | 0 | 0]
[0, 1, 2, 3]
```

يتيح مطابقة الأنماط للمطورين تفكيك أنواع البيانات بسهولة مثل الثنائيات والقوائم، وكما سنرى في الفصول التالية، فهي أحد أساس التكرار في Elixir وتنطبيق على أنواع أخرى أيضاً، مثل الخرائط والثنائيات.

4.3 مشغل الدبوس

يمكن استرداد المتغيرات في Elixir:

```
iex> x = 1
1
iex> x = 2
2
```

[^]مشغل الدبوس قبل المباراة:

يمكن استخدامه عندما لا يكون هناك اهتمام بإعادة ربط متغير ولكن بالمطابقة مع قيمته

```
iex> x = 1
1
^x = 2
ايكس = 2
2 iex> {x, ^x} = {2, 1} {2, 1}
ايكس لا يوجد تطابق لقيمة الجانب الأيمن : 1 (MatchError)
```

```
ايكس
2
```

لاحظ أنه إذا تم ذكر متغير أكثر من مرة في نمط، فيجب أن ترتبط جميع المراجع بنفس النمط:

```
iex> {x, x} = {1, 1} 1
```

```
iex> {x, x} = {1, 2}
ايكس لا يوجد تطابق لقيمة الجانب الأيمن : 1, 2 (MatchError)
```

في بعض الحالات، لا يهمك قيمة معينة في نمط ما. ومن الممارسات الشائعة ربط هذه القيم بالشرط السفلية، .. على سبيل المثال، إذا كان رأس القائمة فقط هو الذي يهمنا، فيمكننا تعين ذيل القائمة للتسطير السفلي:

```
1
ايكس = [1, 2, 3] [1, 2, 3] [ ايكس ] ح
```

إنه خاص لأنه لا يمكن قراءته منه أبداً. محاولة القراءة منه تؤدي إلى خطأ متغير غير مقيد:

```
-_-_-_
ايكس
ايكس متغير غير مرتبطة (CompileError) iex:1: -_-_-
```

على الرغم من أن مطابقة الأنماط تسمح لنا بناء هياكت قوية، إلا أن استخدامها محدود. على سبيل المثال، لا يمكنك إجراء مكالمات وظيفية على الجانب الأيسر من المطابقة. المثال التالي غير صالح:

```
([1, [2, 3]] = 3) الطول iex>
ايكس = 3 (CompileError) iex:1: *نمط غير قانوني
```

وبهذا ننهي مقدمتنا حول مطابقة الأنماط. وكما سنرى في الفصل التالي، فإن مطابقة الأنماط شائعة جداً في العديد من بنية اللغة.

5 حالة وشرط وإذا

toc.html %} تشمل {%

في هذا الفصل سوف نتعلم عن هيكل التحكم في التدفق Case و Cond و If.

حالة 5.1

تسمح لنا الحالة بمقارنة قيمة مع العديد من الأنماط حتى نجد القيمة المطابقة:

هذه الفقرة لن تتطابق "ستتطابق هذه الجملة وترتبط بـ 3 في هذه الجملة"
هذه الجملة تتطابق مع أي قيمة "هذا الجملة تتطابق مع أي قيمة"
النهاية ..>

إذا كنت ترييد مطابقة النمط مع متغير موجود، فأنت بحاجة إلى استخدام عامل :

وتسمح البنود أيضًا بتحديد شروط إضافية عبر الحراس:

الحالات `if`

```
if x > 0:
    print("سوف تطابق")
else:
    print("لن تتطابق")
```

ستتطابق الجملة الأولى أعلاه فقط عندما يكون x موجباً.

5.2 التعبيرات في حمل الحراسة.

نسمح آلة Erlang الافتراضية (VM) فقط بمجموعة محدودة من التعبارات في الحالات :

(≡, ≈, ≡≡, ≈≈, ≥, ≤, ≤≡, ≥≡) ði; ləc|| - ləc||

(١٤) ﴿يَوْمَ لِلْفَلَقِ﴾

(+,-,* /) ñ ñ ñ ñ ñ ñ

• ٢٣٦ - الأدبيات واللغات

الآن، ولهذه الأسباب، فالتجارة، وهي النهاية المطلوبة،

is atom/

16-51-5

- هو منطقی/1

- is_float/1

- هي دالة/1

- هي دالة/2

- هو عدد صحيح/1

- is_list/1

- is_map/1

- هو رقم/1

- is_pid/1

- هو المنفذ/1

- هو المرجع/1

- هو elput_1/-

* بالإضافة إلى هذه الوظائف:

(عدد)sba-

(سلسلة البتات)ezis_tib-

-حجم البيانات (سلسلة البتات)

(عدد صحيح, عدد صحيح)vid-

- elem(tuple, n)

(القائمة)dh-

(الطول) القائمة-

-حجم الخريطة

- العقدة ()

- العقدة (معرف العملية | المرجع | المنفذ)

(عدد صحيح, عدد صحيح)mer-

(الجولة) العدد-

- الذات ()

(القائمة) lt-

(الرقم) cnurt-

-حجم المجموعة (المجموعة)

ضع في اعتبارك أن الأخطاء في الحراس لا تسرب ولكنها ببساطة تؤدي إلى فشل الحارس:

ابكش <1> hd(1)

خطأ في الوسيطة : erlang.hd(1) iex> * (ArgumentError)

```
"لن يتطابق"hd(x) -> x
...عندما "#{}x": "...حصلت على" x ->
...نهائية" حصلت على "1"
```

إذا لم تتطابق أي من البنود، فسيتم رفع الخطأ:

```
iex> do ko:case do
  "ان يتطابق" ...> error ->
  "النهاية" ...>
  ko:...> "لا يوجد شرط حالة مطابق: (CaseClauseError)
```

لاحظ أن الوظائف المجهولة يمكن أن تحتوي أيضًا على عدة جمل وحراست:

```
iex> f = fn
x > 0 -> x + yx, y -> x * y
  ...>
  ...>
  "النهاية" ...>
5> iex> f.(1, 3) 4/rpxe.lave_lre: في Function<12.71889879/2
```

```
iex> f.(-1, 3) -3
```

يجب أن يكون عدد الوسائل في كل بند وظيفة مجهولة هو نفسه، وإلا فسيتم رفع خطأ.

5.3 حالة

تكون الحالة مفيدة عندما تحتاج إلى المطابقة مع قيم مختلفة، ومع ذلك، في العديد من الظروف، نريد التحقق من شروط مختلفة والعنور على أول شرط يتم تقييمه على أنه صحيح. في مثل هذه الحالات، يمكن للمرء استخدام الشرط:

```
اشرط القيام به
2 + 2 == 5...>
"هذا لن يكون صحيحا"
...> 2 * 2 == 3...>
"قولاً هذا"
...> 1 + 1 == 2...>
"لوكن هذا سوف"
"النهاية"
"ولكن هذا سوف"
```

هذا يعادل جمل if في العديد من اللغات الآمرة (على الرغم من استخدامها بشكل أقل كثيراً هنا).

إذا لم يتم إرجاع أي من الشروط إلى القيمة true، فسيتم رفع خطأ. لهذا السبب، قد يكون من الضروري إضافة شرط النهائي، يساوي القيمة true، والذي سيتطابق دائمًا مع:

```
اشرط القيام به
...> 2 + 2 == 5...>
"هذا ليس صحيحاً"
...> 2 * 2 == 3...>
"قولاً هذا"
...> ... صحيح
"هذا صحيح دائمًا (مكافن لـ)"
"النهاية"
```

أخيرًا، لاحظ أن `cond` تعتبر أي قيمة بخلاف nil صحيحة:

```
اشرط القيام به
...> hd([1,2,3]) -> "1...يعتبر صحيحاً"
"النهاية"
"1" يعتبر صحيحاً
```

الإصدار Elixir، توثيق

5.4 إذا وما لم

بالإضافة إلى الحالة والشرط، يوفر Elixir أيضًا وحدات الماكرو `/2` `if/unlal` و `o` والتي تكون مفيدة عندما تحتاج إلى التحقق من شرط واحد فقط:

```
<إذا كان الأمر صحيحاً فافعل ex>
  >...هذا يعمل!
<النهاية>
  "هذا يعمل"
<اما لم يكن صحيحاً فافعل ذلك ex>
  >...لن نرى هذا أبداً
<النهاية>
  لا شيء
```

إذا كان الشرط المعطى `L` `/2` أيعود بقيمة `false` أو `nil`، فلن يتم تنفيذ النص المعطى بين `do/end` وسيعود ببساطة بقيمة `nil`. يحدث العكس مع `except/2`.

كما أنها تدعم كتل `else`:

```
<إذا لم يكن هناك شيء، فافعل ذلك ex>
  >...لن يتم رؤية هذا"
  ...آخر
  "هذا سوف"
  ...النهاية
  "هذه سوف"
```

ملاحظة: هناك ملاحظة متبرأة للاهتمام بخصوص `"unless/2"` و `"if/2"` وهي أنهما يتم تنفيذهما كوحدات ماкро في اللغة؛ وهذا ليسا من بنيات اللغة الخاصة كما هو الحال في العديد من اللغات. يمكنك التتحقق من الوثائق ومصدر `"Kernel"` ["/Kernel"](#) `"Kernel"` هي أيضًا المكان الذي يتم فيه تعريف المشغلات مثل `"is_function/2"` وكثيرًا مستوردة تلقائياً ومتوفرة في الكود الخاص بك افتراضيًا.

5.5 كتل النهاية

في هذه المرحلة، تعلمنا أربع هيئات تحكم: `do/end`، `if/unlal`، `o` و `case/2`، وكانت جميعها مغلفة في كتل `do/end`. ويحدث أيضًا أنه يمكننا كتابة أعلى النحو التالي:

```
1 + 2
<إذا كانت صحيحة، قم بما يلي: 3
```

في Elixir، تعتبر كتل `do/end` وسيلة ملائمة لتمرير مجموعة من التعبيرات إلى `..` وهي متكافئة:

```
<إذا كان الأمر صحيحاً فافعل ex>
= 1 + 2 ...
+ 10 ...
<النهاية...
13
<إذا كانت صحيحة، قم بما يلي: )
= 1 + 2 ...
+ 10 ...
...
) 13
```

نقول إن الصيغة الثانية تستخدم قوائم الكلمات الأساسية. يمكننا تمرير `else` باستخدام الصيغة التالية:

taht: taht، siht: siht، حفظ: حفظ، إذا كان: إذا كان، فافعل: فافعل

أحد الأشياء التي يجب وضعها في الاعتبار عند استخدام كتل end/do هي أنها مرتبطة دائمًا بأخر استدعاء للوظيفة. على سبيل المثال، التعبير التالي:

1 + 2 <...> إإذا كانت القيمة صحيحة فافعل ...> is_number

<النهاية...>

سيتم تحليلها على النحو التالي:

1 + 2 <...> rebmun_siiex> إذا كان صحيحًا فافعل ...

<النهاية...>

يؤدي هذا إلى حدوث خطأ في الوظيفة غير المحددة عندما يحاول Elixir استدعاء number/_is إن إضافة أقواس صريحة كافية لحل الغموض:

1 + 2 <...> rebmun_siiex> إذا كان صحيحًا، فافعل ...

<النهاية...>
حقين

تلعب قوائم الكلمات الرئيسية دورًا مهمًا في اللغة وهي شائعة جدًا في العديد من الوظائف والماكرو. سنتشكّلها بمزيد من التفصيل في فصل مستقبلي. الآن حان الوقت للحديث عن "ال الثنائيات والسلسل وقوائم الأحرف".

6 الثنائيات والسلسل وقوائم الأحرف

toc.html %} تشمل {%

في "الأنواع الأساسية"، تعلمنا عن السلسل واستخدمنا الدالة 1_is_binary للتحقق:

سلسلة = "مرحبا" "مرحبا"> rebmun_siiex>

سلسلة is_binary
حقين

في هذا الفصل، سوف نفهم ما هي الثنائيات، وكيف ترتبط بالسلسل، وما معنى القيمة المقتبسة المفردة، "مثل هذا"، في Elixir.

6.1 UTF-8 و Unicode

السلسلة عبارة عن ملف ثنائي مشفر بتنسيق UTF-8 ولكن نفهم بالضبط ما نعنيه بذلك، نحتاج إلى فهم الفرق بين البايتات ونقاط الترميز.

يعين معيار Unicode نقاط ترميز للأحرف التي نعرفها. على سبيل المثال، الحرف له نقطة رمز 97 بينما الحرف له نقطة رمز 104. عند كتابة السلسلة "hello" على القرص، نحتاج إلى تحويل نقطة الرمز هذه إلى بايتات. إذا تبيننا قاعدة تقول إن البايت الواحد يمثل نقطة رمز واحدة، فلنتمكن من كتابة "hello" لأنها تستخدم نقطة الرمز 104، أو البايت الواحد لا يمكن أن يمثل سوى رقم من 0 إلى 255. ولكن بالطبع، نظرًا لأنه يمكنك قراءة "hello" على شاشتك، فيجب تمثيلها بطريقة ما. وهنا يأتي دور الترميزات.

عند تمثيل تعليمات البرمجة بالبايتات، نحتاج إلى ترميزها بطريقة ما. اخترت Elixir ترميز UTF-8 باعتباره الترميز الأساسي والافتراضي. عندما نقول إن السلسلة عبارة عن ملف ثنائي مشفر برميز UTF-8، فإننا نعني أن السلسلة عبارة عن مجموعة من البايتات منظمة بطريقة تمثيل نقاط تعليمات برمجية معينة، كما هو محدد بواسطة ترميز UTF-8.

الإصدارات Elixir توثيق

نظرًا لأن لدينا نقاط رمز مثل `آخ` مخصصة للرقم `322` فبحن في الواقع نحتاج إلى أكثر من بait واحد لتمثيله. لهذا السبب نرى فرقاً عندما نحسب `String.length/1`: `String.length/1`

```
= "hello" "hello"
```

`7` بait الحجم

`5` طول السلسلة

يتطلب UTF-8 بaitًا واحدًا لتمثيل نقاط الترميز `é` و `ö`. ولكن بايتين لتمثيل `é` في Elixir، يمكنك الحصول على قيمة نقطة الترميز باستخدام:

إيكس < 97
إيكس < 322

يمكنك أيضًا استخدام الوظائف الموجودة في وحدة `String` التقسيم السلسلة في نقاط الكود الخاصة بها:

```
iex> String.codepoints("hello") ["h", "e", "l", "l", "o"]
```

ستجد أن Elixir يدعم بشكل ممتاز العمل مع السلسل. كما أنه يدعم العديد من عمليات Unicode. في الواقع، يحتاز Elixir جميع الاختبارات المذكورة في المقال "نوع السلسلة مكسور".

ومع ذلك، فإن السلسل ليست سوى جزء من القصة. إذا كانت السلسلة ثنائية، واستخدمنا الدالة `String.to_binary/1` فيجب أن يحتوي Elixir على نوع أساسى يمكن السلسل. وهذا صحيح. دعنا نتحدث عن الثنائيات!

6.2 الملفات الثنائية (وسلاسل البتات)

في Elixir، يمكنك تعريف ثنائي باستخدام:

```
<<0, 1, 2, 3 >> <<0, 1, 2, 3 >> <<0, 1, 2, 3 >> إيكس < بait_حجم 4
```

الملف الثنائي عبارة عن سلسلة من البايتات. بالطبع، يمكن تنظيم هذه البايتات بأي طريقة، حتى في تسلسل لا يجعلها سلسلة صالحة:

```
iex> String.valid?(<<239, 191, 191>>) false
```

عملية ربط السلسلة هي في الواقع عملية ربط ثنائية:

```
<<0, 1>> <> <<2, 3>> <<0, 1, 2, 3>> أي <
```

الخدعة الشائعة في Elixir هي ربط البايت الفارغ `<>` بسلسلة لرؤيه تمثيله الثنائي الداخلي:

```
iex> "hello" <> <<0>>
<<104, 101, 197, 130, 197, 130, 111, 0 >>
```

كل رقم يتم إعطاؤه لملف ثانٍ يهدف إلى تمثيل بايت وبالتالي يجب أن يصل إلى 255. تسمح الملفات الثنائية بإعطاء تعديلات لتخزين أرقام أكبر من 255 أو تحويل نقطة رمز إلى تمثيلها بتنسيق utf8:

```
<<255>> ايكس
<<255>>
امقطوع<ex> <<256>> #
<<0>>
"-A" البرقم عبارة عن نقطة رمز 16 بت (2 بايت) لتخزين الرقم استخدم<ex> <<256 :: size(16)>> #

<<256 :: utf8, 0>> ايكس
<<196, 128, 0>>
```

إذا كان البايت يحتوي على 8 بتات، فماذا يحدث إذا مررنا حجمًا يبلغ 1 بت؟

```
size(1)>> false امقطوع<ex> <<1 :: size(1)>> <<1::size(1)>> iex> <<2 :: size(1)>> #
<<0::size(1)>> iex> is_binary(<< 1 ::

(<<(1)(جمال)<ex> is_bitstring(<< 1 ::

قيمة<ex> bit_size(<< 1 :: size(1)>>) 1
```

القيمة لم تعد ثنائية، بل سلسلة بتات - مجرد مجموعة من البتات! لذا فإن الثنائي عبارة عن سلسلة بتات حيث يكون عدد البتات قابلاً للقسمة على 8!

يمكننا أيضًا مطابقة النمط على الثنائيات / سلاسل البتات:

```
iex> <<0, 1, x>> = <<0, 1, 2>>
<<0, 1, 2>> iex> x
2
<<0, 1, x>> = <<0, 1, 2, 3>>
أيكس<ex> لا يوجد تطابق لقيمة الجانب الأيمن: <<0, 1, 2, 3>> (MatchError)
```

لاحظ أن كل إدخال في الملف الثنائي من المتوقع أن يتطابق تمامًا مع 8 بتات. ومع ذلك، يمكننا المطابقة مع قيمة معدل الملف الثنائي:

```
= <<0, 1, 2, 3 >> <<0, 1, 2, 3>> iex> x <<0, 1, x ::

<<2, 3>>
```

لا يعمل النمط أعلاه إلا إذا كان الثنائي في نهاية .>>> ويمكن تحقيق نتائج مماثلة باستخدام عاملربط السلسلة:>:

```
"مرحبا"= "هو" يقية<ex>
"مرحبا"= "هو" يقية<ex>
"الباقي<ex>
"لو"
```

وبهذا ننهي جولتنا في سلاسل البتات والثنائيات والسلسلات. السلسلة عبارة عن ملف ثنائي مشفر بتنسيق UTF-8، الثنائي عبارة عن سلسلة بتات حيث يكون عدد البتات قابلاً للقسمة على 8، ورغم أن هذا يوضح المرونة التي يوفرها Elixir للعمل مع البتات والثنائيات، فإنه يستعمل في 99% من الوقت مع الثنائيات واستخدام الداللين.

3. الأحرف قوائم

قائمة الأحرف ليست أكثر من قائمة من الأحرف:

```
[104, 101, 322, 322, 111] iex> is_list 'hello'
```

```
<--> iex>
      'مرحبا'
```

يمكنك أن ترى أنه بدلاً من احتواء البيانات، تحتوي قائمة الأحرف على نقاط رمز الأحرف الموجودة بين علامتي اقتباس مفردين (لاحظ أن `hex` يخرج نقاط رمز فقط إذا كان أي من الأحرف خارج نطاق ASCII)، بينما تمثل علامتا الاقتباس المزدوجتان سلسلة (أي ثنائية)، تمثل علامتا الاقتباس المفردة قائمة أحرف (أي قائمة).

في الممارسة العملية، تُستخدم قوائم الأحرف في الغالب عند التفاعل مع `Erlang` وخاصة المكتبات القديمة التي لا تقبل الثنائيات كحجج، يمكنك تحويل قائمة الأحرف إلى سلسلة والعكس باستخدام الدالتين `list/1` و `to_string/1`

```
iex> to_char_list "hello" [104, 101, 322, 322, 111] iex> to_string 'hello' "مرحبا"
```

```
"مرحبا" iex> to_string :hello
```

```
iex> to_string 1 "1"
```

لاحظ أن هذه الوظائف متعددة الأشكال. فهي لا تقوم بتحويل قوائم الأحرف إلى سلسل فحسب، بل تقوم أيضاً بتحويل الأعداد الصحيحة إلى سلسل، والذرات إلى سلسل، وما إلى ذلك.

بعد أن أوضحنا الثنائيات والسلسل وقوائم الأحرف، حان الوقت للحديث عن هيكل البيانات ذات القيمة الأساسية.

7. كلمات رئيسية وخرائط وقواميس

toc.html %} تشمل

حتى الآن لم نناقش أي هيكل بيانات ارتباطية، أي هيكل البيانات القادرة على ربط قيمة معينة (أو قيم متعددة) بمفتاح، وتطلق اللغات المختلفة على هذه الهياكل أسماء مختلفة مثل القواميس، وعلامات التجزئة، والمصفوفات الارتباطية، والخرائط، وما إلى ذلك.

في `Elixir` لدينا بنية رئيسية للبيانات الترابطية: قوائم الكلمات الرئيسية والخرائط. حان الوقت لمعرفة المزيد عنهم!

7.1. قوائم الكلمات الرئيسية

في العديد من لغات البرمجة الوظيفية، من الشائع استخدام قائمة من العناصر المكونة من عنصرين كمثيل لبنية بيانات ارتباطية. في `Elixir`، عندما يكون لدينا قائمة من العناصر المكونة من عنصرين والعنصر الأول في العنصر المكون من عنصرين (أي المفتاح) هو ذرة، نطلق عليها قائمة الكلمات الأساسية:

```
== [a: 1, b: 2] == صحيح
```

```
[a:][[قائمة]] iex>
```

```
1
```

كما ترى أعلاه، يدعم Elixir صيغة خاصة لتحديد مثل هذه القوائم، وفي الأسفل يتم ربطها بقائمة من الثنائيات. ونظرًا لأنها قوائم ببساطة، فإن جميع العمليات المتاحة للقوائم، بما في ذلك خصائص أدائها، تتطابق أيضًا على قوائم الكلمات الرئيسية.

على سبيل المثال، يمكننا استخدام `+إضافة قيمة جديدة إلى قائمة الكلمات الرئيسية:`

```
++ [c: 3] قائمة iex>
      ++ [:3] iex> :1, :2, :3
      ++ [:1, :2, :3] iex>
```

لاحظ أن القيم المضافة إلى المقدمة هي القيم التي تم جلبها عند البحث:

```
0 قائمة جديدة = [:1, :2] قائمة جديدة iex>
```

تعتبر قوائم الكلمات الرئيسية مهمة لأنها تتمتع بثلاث خصائص خاصة:

- * يجب أن تكون المفاتيح عبارة عن ذرات.

- * يتم ترتيب المفاتيح على النحو الذي حدد المطور.

- * يمكن إعطاء المفاتيح أكثر من مرة.

على سبيل المثال، مكتبة `Ecto` يستخدم كلتا الميزتين لتوفير لغة مجال محددة أنيقة لكتابية استعلامات قاعدة البيانات:

```
استعلام = من w.temp < 20، حيث: w.prcp > 0، حيث: w في الطقس، حيث: siht، إذا كان خطأ، وإلا: taht iex>
```

اختر: و

هذه الميزات هي التي دفعت قوائم الكلمات الرئيسية إلى أن تكون الآلية الافتراضية لتمرير الخيارات إلى الوظائف في Elixir. في الفصل الخامس، عندما ناقشنا الماكرو `if/2`، ذكرنا أن بناء الجملة التالي مدعوم:

```
taht: taht، siht: siht، إذا كان خطأ، وإلا: iex>
```

أزواج `:else` و `:do`: عبارة عن قوائم كلمات رئيسية في الواقع، فإن المكالمات أعلاه تعادل:

```
if(false) then
  أفعال: هذا، وإنما: بذلك
end iex>
```

يشكل عام، عندما تكون قائمة الكلمات الرئيسية هي الوسيطة الأخيرة للدالة، تكون الأقواس المرجعية اختيارية.

من أجل معالجة قوائم الكلمات الرئيسية، توفر Elixir وحدة "الكلمات الرئيسية".
`</docs/stable/elixir/Keyword.html>` تذكر أن قوائم الكلمات الرئيسية هي مجرد قوائم، وبالتالي فهي توفر نفس خصائص الأداء الخطية مثل القوائم، وكلما كانت القائمة أطول، كلما استغرق الأمر وقتًا أطول للعثور على مفتاح ولعد عدد العناصر، وما إلى ذلك. ولهذا السبب، تستخدم قوائم الكلمات الرئيسية في Elixir بشكل أساسي كخيارات. إذا كنت بحاجة إلى تخزين العديد من العناصر أو ضمان ارتباط مفتاح واحد بقيمة واحدة كحد أقصى، فيجب عليك استخدام الخرائط بدلاً من ذلك.

على الرغم من أنه يمكننا مطابقة النمط في قوائم الكلمات الرئيسية، إلا أنه نادرًا ما يتم ذلك في الممارسة العملية نظرًا لأن مطابقة النمط في القوائم تتطلب مطابقة عدد العناصر وترتيبها:

```
1] [:]= [:] iex>
      1]
      ايكس>
      1
      2] [:]= [:] iex>
```

`[a: 1, b: 2] iex> [b: 2, a: a] = [a: 1, b: 2]` لا يوجد تطابق لقيمة الجانب الأيمن: (MatchError)

`[a: 1, b: 2] iex> [a: 1, b: 2]` لا يوجد تطابق لقيمة الجانب الأيمن: (MatchError)

7.2 الخرائط

عندما تحتاج إلى مخزن قيم مفتاحية، فإن الخرائط هي بيئة البيانات "الرئيسية" في Elixir. يتم إنشاء الخريطة باستخدام صيغة: `%{}`

`[a: 1, 2 => :b] %{2 => :b, :a => 1} iex>` الخريطة

```
1
[2] iex>
    ب:
    ب
    لا شيء map[:c]
```

بالمقارنة مع قوائم الكلمات الرئيسية، يمكننا بالفعل رؤية اختلافين:

- تسمح الخرائط بأي قيمة كمفتاح.

- مفاتيح الخرائط لا تتبع أي ترتيب.

إذا قمت بتمرير مفاتيح مكررة عند إنشاء الخريطة، يفوز المفتاح الأخير:

`%{1 => 1, 1 => 2} %{1 => 2}` أي < 2>

عندما تكون جميع المفاتيح في الخريطة عبارة عن ذرات، يمكنك استخدام بناء جملة الكلمات الأساسية للراحة:

`2 {1, 1 => :b, :b: 1} iex>` الخريطة

على النقيض من قوائم الكلمات الرئيسية، تعتبر الخرائط مفيدة جدًا في مطابقة الأنماط:

```
=> 1, 2 => :b) iex> %{a => a} = %{a => 1, 2 => :b} %{a => 1, 2 => :b} iex> a
iex> %{} = %{a => 1, 2 => :b} %{a
```

1

```
iex> %{c => c} = %{a => 1, 2 => :b}
%{2 => :b, :a => 1} iex> لا يوجد تطابق لقيمة الجانب الأيمن: (MatchError)
```

كما هو موضح أعلاه، تتطابق الخريطة طالما أن المفاتيح المحددة موجودة في الخريطة المحددة. وبالتالي، فإن الخريطة الفارغة تتطابق مع جميع الخرائط.

توفر وحدة "الخريطة" `Map.html` واجهة برمجة تطبيقات مشابهة جدًا لوحدة الكلمات الرئيسية مع وظائف ملائمة للتعامل مع الخرائط:

`iex> Map.get(%{a => 1, 2 => :b}, :a)`

`{:a => 1, 2 => :b}) [{2, :b}, {:a, 1}]%)(tsil ot. خريطة iex>`

من الخصائص المثيرة للاهتمام حول الخرائط أنها توفر بناء جملة معيناً لتحديث مفاتيح الذرة والوصول إليها:

```
= %{a => 1, 2 => :b} %{a => 1, 2 => :b} الخريطة
iex>
1
%{2 => :b, :a => 1} فـ: map يتم العثور على المفتاح c:  
| :a => 2) %{a => 2, 2 => :b} الخريطة
iex> %{map | :c => 3}
خطأ** في الحجة (ArgumentError)
```

وتتطلب كل من قواعد الوصول والتحديث أعلاه وجود المفاتيح المحددة. على سبيل المثال، فشل الوصول إلى مفتاح `c` وتحديده، فلا يوجد `c` في الخريطة.

فضل مطورو Elixir عادةً استخدام صيغة `Map.field` عند العمل مع الخرائط لأنها تؤدي إلى أسلوب برمجة حازم. يقدم رؤى وأمثلة حول كيفية الحصول على برامج أكثر إيجازاً وأسرع من خلال كتابة كود حازم في Elixir.

ملاحظة: تم تقديم الخرائط مؤخرًا إلى VM Erlang EEP 43، حيث يتم دعم "الخرائط الصغيرة" فقط. وهذا يعني أن الخرائط تتمتع بخصائص أداء جيدة فقط عند تخزين بعض عشرات المفاتيح كحد أقصى. ولسد هذه الفجوة، يوفر Elixir أيضًا وحدة `HashDict`، وهي تستخدم خوارزمية التجزئة لتوفير قاموس يدعم مئات الآلاف من المفاتيح بأداء جيد.

القواعد 7.3

Elixir هو لغة برمجة فعالة وذات إنتاجية، وهي تسمى قوائم الكلمات الرئيسية والخراطيل بالقوميين. بعبارة أخرى، القاموس يشبه الواجهة (نسميه سلوكيات في Elixir) وتقوم كل من قوائم الكلمات الرئيسية ووحدات الخراطيل بتتنفيذ هذه الواجهة.

وتحت عنوان `Dict` تم تبريره في وحدة `Dict` بـ [برمجة تطبيقات Elixir](#).

= الكلمة الأساسية [] lielex>
= %{} {} {} الخريطة lielex>
1) [a: 1] a: الكلمة الأساسية tup.tciDiex>
1) %{a: 1} a: الخريطة tup.tciDiex>

Dict API مطور تفنيد نسخة الخاصة من Dict، وبخصائص محددة، والارتباط يكود الحالى. توفر وحدة ElixirDict API وأيضاً وظائف تهدف إلى العمل عبر القواميس.

على سبيل المثال، يمكن لـ `Dict.equal` مقارنة قاموسين من أي نوع.

ومع ذلك، قد تتسائل، أي من وحدات Dict أو Map أو Keyword يجب عليك استخدامها في الكود الخاص بك؟ الإجابة هي: يعتمد ذلك على نوع الوحدة.

فاستخدم وحدة **Keyword** **Dict** (وتأكد من كتابة اختبارات تمرن تفاصيل dict مختلفة كحجج).
ومع ذلك، إذا كانتواجهة برمجة التطبيقات الخاصة بك مخصصة للعمل مع أي قاموس، فاستخدم وحدة **Dict** إذا كان الكود الخاص بك يتوقع بيئة بيانات محددة كحجج، فاستخدم الوحدة النمطية ذات الصلة لأنها تؤدي إلى كود أكثر تأكيداً. على سبيل المثال، إذا كنت تتوقع كلمة أساسية كحجج، فاستخدم وحدة **Dict** بدلاً من **Keyword**.

هذا يحتمم مقدمتنا عن هيكل البيانات الترابطية في Elixir. ستكتشف أنه بفضل قوائم الكلمات الرئيسية والخراط، سيكون لديك دائمًا الأداة المناسبة لمعالجة المشكلات التي ترتبط بها البيانات الترابطية في Elixir.

وحدات 8

toc.html %} تشمل

في Elixir تقوم بجمع العديد من الوظائف في وحدات. لقد استخدمنا بالفعل العديد من الوحدات المختلفة في الفصول السابقة مثل وحدة: `_String'`

`"hello" 5 طول السلسلة`

لإنشاء وحداتنا الخاصة في، `defmodule` يستخدم الماكرو `defmacro`. `defmodule` تحدد الوظائف في تلك الوحدة:

```
الرياضيات تفعيل<elixir> defmodule
...> def sum(a, b) do
...>     a + b
...>...النهاية
...>...النهاية
```

`(1. 2) مجموع الرياضيات 3`

في الأقسام التالية، ستصبح أمثلتنا أكثر تعقيداً بعض الشيء، وقد يكون من الصعب كتابتها جميماً في shell. حان الوقت لنتعلم كيفية جماع كود Elixir وكيفية تشغيل نصوص.

التجميع 8.1

في أغلب الأحيان، من الملائم كتابة الوحدات النمطية في ملفات حتى يمكن تجميعها وإعادة استخدامها. لنفترض أن لدينا ملفاً باسم `math.ex` يحتوي على المحتويات التالية:

```
الرياضيات تفعيل<elixir> defmodule
def sum(a, b) do a + b
... نهاية
... نهاية
```

يمكن تجميع هذا الملف باستخدام:

`$ elixirc math.ex`

سيؤدي هذا إلى إنشاء ملف باسم `Math.beam` على الكود الثنائي للوحدة المحددة. إذا بدأنا تشغيل `math.ex` مرة أخرى، فسيكون تعريف الوحدة متاحاً (بشرط بدء تشغيل Elixir). في نفس الدليل الذي يوجد فيه ملف الكود الثنائي):

`(1. 2) مجموع الرياضيات 3`

يتم تنظيم مشاريع Elixir عادةً في ثلاثة أدلة:

- `ebin` - يحتوي على البايت كود المجمّع

- `lib` - يحتوي على كود Elixir (files. xe).

- `sxex` - يحتوي على اختبارات (files. test).

عند العمل على مشاريع فعلية، ستكون أداة البناء المسماة `mix` مسؤولة عن تجميع المسارات المناسبة لك وإعدادها. ولأغراض التعلم، يدعم Elixir أيضًا وضيقاً نسبياً أكثر مرونة ولا يولد أي آثار مجتمعة.

8.2 الوضع النصي

بالإضافة إلى امتداد الملف .ex، يدعم Elixir أيضًا ملفات .sxe للبرمجة النصية. يعامل Elixir كلا الملفين بنفس الطريقة تمامًا، والفرق الوحيد هو في الغرض. ملفات .sxe مخصصة للتجميع بينما تُستخدم ملفات .math.exs على سبيل المثال، يمكننا إنشاء ملف يسمى:

```
defmodule الرياضيات_تفعل
do sum(a, b) do a + b

نهاية
نهاية

2) الرياضيات_مجموع(0.puts
```

ونفذها على النحو التالي:

```
sxe.إكسير_الرياضيات$
```

سيتم تجميع الملف في الذاكرة وتنفيذها، وطباعة "3" كنتيجة لذلك. لن يتم إنشاء ملف بابت كود. في الأمثلة التالية، نوصيك بكتابه الكود الخاص بك في ملفات نصية وتنفيذها كما هو موضح أعلاه.

8.3 الوظائف المسماة

داخل الوحدة النمطية، يمكننا تعريف وظائف خاصة باستخدام `defp/2` ويمكن استدعاء وظيفة محددة باستخدام `def/2` من وحدات نمطية أخرى بينما لا يمكن استدعاء وظيفة خاصة إلا محلياً.

```
defmodule الرياضيات_تفعل
do sum(a, b) do do_sum(a, b)

نهاية
نهاية

defp do_sum(a, b) do a + b

نهاية
نهاية

#=> 3
2) مجموع_الرياضيات(1, 2) #=> ** خطأ وظيفة غير محدد(Math.do_sum(1, 2))
```

دعم إعلانات الوظائف أيضًا الحراس والجمل المتعددة. إذا كانت الوظيفة تحتوي على عدة جمل، فسوف يجرب كل جملة حتى يجد جملة مطابقة، فيما يلي تجربة لوظيفة تتحقق مما إذا كان الرقم المعطى صفرًا أم لا:

```
defmodule الرياضيات_تفعل
do zero?(0) == صفر_صفر؟()
    حقيقى
نهاية

is_number(x) عندما يكون def zero?(x)
    خطأ شنيع
نهاية
نهاية

Math.zero?(0) #=> true Math.zero?(1) #=> false
```

الإصدارات Elixir توثيق

([1,2,3])
الرياضيات، صفر؟
(** جملة الوظيفة) #=> خطأ في

إن تقديم حجة لا تتطابق مع أي من البنود يؤدي إلى حدوث خطأ.

8.4 التقطط الوظيفية

حلل هذا البرنامج التعليمي، كنا نستخدم صيغة الاسم/الصيغة للإشارة إلى الدوال. وقد حدث أن هذه الصيغة يمكن استخدامها بالفعل لاسترداد دالة مسماة كنوع دالة. فلنبدأ تشغيل `iex` وتشغيل ملف `math.exs` المحدد أعلاه:

```
$ iex math.exs
```

```
(0)< iex> الرياضيات، صفر؟(0)< iex>
< الحقيقي>
iex> fun = &Math.zero?/1 &Math.zero?/1 iex>
      دالة fun
```

```
< الحقيقي>
(0)< iex> امتحن(0)< iex>
< الحقيقي>
```

يمكن التقطط الوظائف المحلية أو المستوردة، مثل `is_function/1`، بدون الوحدة النمطية:

```
1 &:erlang.is_function/1 iex> (&is_function/1).
      (0)< iex> &is_function/
```

< الحقيقي>

لاحظ أنه يمكن أيضًا استخدام بناء جملة التقطط كاختصار لإنشاء الوظائف:

```
= &(&1 + 1) iex>
5> iex> fun.(1) 2/rpxe.lave_lre: في #Function<6.71889879/1
```

يمثل `&1` الوسيطة الأولى التي تم تمريرها إلى الدالة. `(&1 + 1)` وأعلاه هو نفس `x + 1`. `end` يمثل الصيغة أعلى `fn` التي تم تعریفات الدالة القصيرة.

وبنفس الطريقة إذا كنت تريد استدعاء دالة من وحدة نمطية، يمكنك القيام بـ `&Module.function()`:

```
iex> fun = &List.flatten(&1, &2)
      2/قائمة_تسطح<
      ([1, [[2], 3]], [4, 5]). [4, 5] iex>
      [1, 2, 3, 4, 5]
```

`Kernel.SpecialForms.html#/1` يمكنك قراءة المزيد حول عامل التقطط `&` في وثائق `_`. `fn(list, tail) -> List.flatten(list, tail) end`. هو نفس كتابة `List.flatten(&1, &2)` "Kernel.SpecialForms" </docs/stable/elixir/

8.5 الحجج الافتراضية

تدعم الوظائف المسماة في `Elixir` أيضًا الوسائل الافتراضية:

```
بربط defmodule
def join(a, b, sep \\ " ") do a <> sep <> b end
```

نهاية

```
(("Hello" \\ "world") \\ "Hello_world")
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

يُسمح لأي تعبير بالعمل كقيمة افتراضية، ولكن لن يتم تقييمه أثناء تعریف الدالة؛ بل سيتم تخزينه ببساطة لاستخدامه لاحقاً. في كل مرة يتم فيها استدعاء الدالة ويجب استخدام أي من قيمها الافتراضية، سيتم تقييم التعبير الخاص بهذه القيمة الافتراضية:

```
بربط defmodule DefaultTest
def dowork(x \\ IO.puts "hello") do
```

س
نهاية
نهاية

```
iex> DefaultTest.dowork
```

مرحبا

نعم:

```
iex> DefaultTest.dowork 123
```

123

```
iex> DefaultTest.dowork
```

مرحبا

نعم:

إذا كانت الدالة ذات القيم الافتراضية تحتوي على عدة فقرات، فمن المستحسن إنشاء رأس دالة (بدون جسم فعلي)، فقط لإعلان القيم الافتراضية:

```
بربط defmodule
def join(a, b \\ nil, sep \\ "")
```

أ
نهاية

```
a <> sep <> b → def join(a, b, sep)
```

نهاية
نهاية

```
("Hello_world" \\ "Hello_world")
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

عند استخدام القيم الافتراضية، يجب توخي الحذر لتجنب تداخل تعریفات الوظائف. ضع في اعتبارك المثال التالي:

```
بربط defmodule
def join(a, b) do
  IO.puts "***First join" a <> b
```

نهاية

```
def join(a, b, sep \\ "") do
```

الإصدار، Elixir توثيق

```
IO.puts "***Second join" a <> sep <> b end
```

نهاية

إذا قمنا بحفظ الكود أعلاه في ملف يسمى "concat.ex" وقمنا بتحميجه، فسوف يصدر Elixir التحذير التالي:

concat.ex:7: يمكن أن تتطابق هذه الجملة لأن الجملة السابقة في السطر 2 تتطابق دائمًا مع :-

يخبرنا المترجم أن استدعاء دالة join سبب دلائلاً إلى اختيار التعريف الأول لـ join بينما سيتم استدعاء التعريف الثاني فقط عند تمرير ثلاثة حجتين:

```
$ iex concat.exs
```

"مرحبا، العالم"iex> Concat.join
"HelloWorld" اولاً إلى انضم ***

"مرحبا، العالم"iex> Concat.join
انضمام الثاني إلى "Hello_world"***

هذا يعني مقدمتنا القصيرة عن الوحدات النمطية. في الفصول التالية، سنتعلم كيفية استخدام الدوال المسممة للتكرار، واستكشاف توجيهات Elixir المعجمية التي يمكن استخدامها لاستيراد الدوال من وحدات نمطية أخرى ومناقشة سمات الوحدات النمطية.

التكرار 9

toc.html %} تشمل {%

9. حلقات التكرار

بسبب عدم قابلية التغيير، تم كتابة الحلقات في Elixir (وفي أي لغة برمجة وظيفية) بشكل مختلف عن اللغات الأمرية. على سبيل المثال، في لغة الأمر (مثل C في المثال التالي)، يمكن كتابة:

```
بالنسبة إلى < i = 0; i < طول المصفوفة; i++) {
    مصفوفة[i] = مصفوفة[i] * 2
}
```

في المثال أعلاه، نقوم بتحويل كل من المصفوفة والمتغير . التحويل غير ممكن في Elixir. بدلاً من ذلك، تعتمد اللغات الوظيفية على التكرار: يتم استدعاء الدالة بشكل متكرر حتى يتم الوصول إلى شرط يمنع استمرار الإجراء التكراري. لا يتم تحويل أي بيانات في هذه العملية. فكر في المثال أدناه الذي يطبع سلسلة عدداً عشوائياً من المرات:

```
defmodule التكرار do
    IO.puts "رسالة نهاية الرسالة"
    IO.puts "فقط بذلك"
    def print_multiple_times(msg, n)
        quando n <= 1 then
            IO.puts msg
        else
            IO.puts msg
            print_multiple_times(msg, n - 1)
        end
    end
end
```

نهاية

```
3) "مرحبا!"(semit_elpitum_tnirp، التكرار.
```

#مرحبا!

#مرحبا!

#مرحبا!

على غرار الحال، قد تحتوي الدالة على العديد من البنود. يتم تنفيذ بند معين عندما تتطابق الوسائل التي يتم تمريرها إلى الدالة مع أنماط وسائط البند ويتم تقييم حارسها على أنه صحيح.

عندما يتم استدعاء `multiple_times/2` في البداية في المثال أعلاه، تكون الحجة `n` متساوية لـ 3.

تحتوي الجملة الأولى على حارس ينص على "استخدم هذا التعريف إذا و فقط إذا كان `n` أقل من أو يساوي 1". نظرًا لأن هذه ليست الحالة، يتنقل Elixir إلى تعريف الجملة التالية.

يتوافق التعريف الثاني مع النمط ولا يحتوي على حارس، لذا سيتم تنفيذه. يقوم أولاً بطباعة رسالتنا ثم يستدعي نفسه وبمرر `(2 - n)` كحجية ثانية.

تم طباعة رسالتنا وتم استدعاء `multiple_times/2` مرة أخرى هذه المرة مع تعيين الحجة الثانية على 1. نظرًا لأن `n` متسوط الآن على 1، فإن الحارس في تعريفنا الأول لـ `multiple_times/2` يتم تقييمه على أنه صحيح، ونقوم بتنفيذ هذا التعريف المحدد. تم طباعة الرسالة، ولا يتبقى شيء لتنفيذها.

لقد قمنا بتعريف `multiple_times/2` بحيث يغض النظر عن الرقم الذي يتم تمريره كحجية ثانية فإنه إما أن يؤدي إلى تشغيل تعريفنا الأول (المعروف باسم "الحالة الأساسية") أو تشغيل تعريفنا الثاني الذي سيضمن أننا نقترب خطوة واحدة بالضبط من حالتنا الأساسية.

9.2 خوارزميات "الاختزال" و"الرسم البياني"

دعونا نرى الآن كيف يمكننا استخدام قوة التكرار لتلخيص قائمة من الأرقام:

```
defmodule الرياضيات_تفعل
  def sum_list([head | tail], accumulator) do
    sum_list(tail, head + accumulator)
  end
```

```
افعل def sum_list([], accumulator)
      افعلنها
      افعلنها
      افعلنها
```

```
الرياضيات_mus.
```

نستدعي `sum_list` مع القائمة `[1, 2, 3]` والقيمة الأولية `0` كحجج. سنحاول كل شرط حتى نجد شرطًا يتطابق وفقًا لقواعد مطابقة النمط. في هذه الحالة، تتطابق القائمة `[1, 2, 3]` مع `[head | tail]`. يتم تحد `head` إلى `1`، وتتم تعيين `tail` إلى `[2, 3]`.

بعد ذلك، نضيف رأس القائمة إلى المتغير `accumulator + head`. نستدعي `sum_list` مرة أخرى، بشكل متكرر، ونمرر ذيل القائمة كحجية أولى لها. ستطابق الذيل مرة أخرى مع `[head | tail]` حتى تصبح القائمة فارغة، كما هو موضح أدناه:

قائمة المجموع 0	1	2	3	قائمة المجموع 3	1	2	3	قائمة المجموع 6
-----------------	---	---	---	-----------------	---	---	---	-----------------

عندما تكون القائمة فارغة، فسوف تتطابق مع الجملة النهائية التي ترجع النتيجة النهائية.

تُعرف عملية أحد قائمة و "تلقيصها" إلى قيمة واحدة باسم خوارزمية "الاختزال" وهي أساسية في البرمجة الوظيفية.

ماذا لو أردنا بدلاً من ذلك مضاعفة جميع القيم في قائمنا؟

```
defmodule الرياضيات_تفعل do
  @spec double_each([head | tail]) :: [head | tail]
  def double_each([head | tail]) do
    [head * 2 | double_each(tail)]
  end
end
```

`double_each([2, 3]) #=> [2, 4, 6]`

نهاية
نهاية

لقد استخدمنا هنا التكرار لجنيز قائمة مضاعفة كل عنصر وإرجاع قائمة جديدة. تُعرف عملية أحد قائمة و "تعيينها" عليها باسم خوارزمية "التعيين".

التكرار واستدعاء `الذيل` تعد عمليات التحسين جزءاً مهماً من Elixir، وُتستخدم عادةً لإنشاء حلقات. ومع ذلك، عند البرمجة في Elixir، نادرًا ما تستخدم التكرار كما هو موضح أعلاه للتلاعب بالقوائم.

توفر وحدة "Enum" التي سنراها في الفصل التالي العديد من المزايا للعمل مع القوائم. على سبيل المثال، يمكن كتابة الأمثلة أعلاه على النحو التالي:

(نهائية) iex> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc

6

[2, 4, 6] (نهائية) iex> Enum.map([1, 2, 3], fn(x) -> x * 2

أو باستخدام صيغة الالتقاط:

iex> Enum.reduce([1, 2, 3], 0, &+/2) 6

iex> Enum.map([1, 2, 3], &(&1 * 2)) [2, 4, 6]

دعونا نلقي نظرة أعمق على العناصر القابلة للعد، وبينما نحن في هذا الشأن، دعونا نلقي نظرة على نظريتها الكسولة، والتي تسمى Streams.

العناصر القابلة للعد والتدفقات

toc.html %} تشمل

10.1 العناصر القابلة للعد

يوفر Elixir مفهوم العناصر القابلة للعد ووحدة `_Enum` للعمل معها. لقد تعلمنا بالفعل عنصررين قابلين للعد: القوائم والخريط.

(نهائية) [2, 4, 6] iex> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k * v) (نهائية)

Enum.map([1, 2, 3], fn x -> x * 2

[2, 12]

توفر وحدة `Enum` مجموعة ضخمة من الوظائف لتحويل العناصر وفرزها وتجميعها وتصفيتها واسترجاعها من العناصر القابلة للعد. وهي إحدى الوحدات التي يستخدمها المطوروون بشكل متكرر في كود Elixir الخاص بهم.

كما تقدم Elixir أيضًا نطاقات:

```
iex> Enum.map(1..3, fn x -> x * 2 end) [2, 4, 6] iex> Enum.reduce(1..3, 0, &+/2) 6
```

نظرًا لأن وحدة `Enum` مصممة للعمل عبر أنواع بيانات مختلفة، فإن واجهة برمجة التطبيقات الخاصة بها تقتصر على الوظائف المفيدة عبر العديد من أنواع البيانات. بالنسبة للعمليات المحددة، قد تحتاج إلى الوصول إلى وحدات محددة لأنواع البيانات. على سبيل المثال، إذا كنت تريد إدراج عنصر في موضع معين في قائمة، فيجب عليك استخدام دالة `List.insert_at/3` حيث لن يكون من المنطقي إدراج قيمة في نطاق، على سبيل المثال.

نقول إن الوظائف في وحدة `Enum` متعددة الأشكال لأنها يمكن أن تعمل مع أنواع بيانات متنوعة. وعلى وجه الخصوص، يمكن للوظائف في وحدة `Enum` العمل مع أي نوع بيانات ينفذ بروتوكول `Enumerable` "ستنقاش البروتوكولات في فصل لاحق، أما الآن فسوف ننتقل إلى نوع محدد من العناصر القابلة للعد والتي تسمى التدفقات.".

الحماس مقابل الكسل 10.2

جميع الوظائف في وحدة `Enum` حريصة. تتوقع العديد من الوظائف وجود عنصر قابل للعد ثم تعيد قائمة:

```
[1, 3] [5فردی] iex> Enum.filter(1..3, /rpxe.lave_lre: في #Function<6.80484245/1>
```

وهذا يعني أنه عند إجراء عمليات متعددة باستخدام `Enum` فإن كل عملية ستولد قائمة وسيطة حتى تصل إلى النتيجة:

```
|> Enum.sum 7500000000 |> Enum.map(&(&1 * 3)) |>
```

يحتوي المثال أعلاه على خط أنابيب للعمليات. نبدأ بـنطاق ثم نضرب كل عنصر في النطاق في 3. ستؤدي هذه العملية الأولى الآن إلى إنشاء قائمة تحتوي على 100000 عنصر وإرجاعها. ثم نحتفظ بكل العناصر الفردية من القائمة، ونشئ قائمة جديدة تحتوي الآن على 50000 عنصر، ثم نجمع كل الإدخالات.

مشغل الأنابيب 10.2.1

الرمز `<|` المستخدم في المقطع أعلاه هو عامل الأنابيب: فهو ببساطة يأخذ الناتج من التعبير على جانبه الأيسر ويمرره كمدخل إلى استدعاء الوظيفة على جانبه الأيمن. وهو مشابه لعامل يونكس `.` والغرض منه هو تسلیط الضوء على تدفق البيانات التي يتم تحويلها بواسطة سلسلة من الوظائف. لمعرفة كيف يمكنه جعل الكود أكثر وضوحاً، ألق نظرة على المثال أعلاه المعاد كتابته دون استخدام عامل `:`:

```
7500000000 ((فردی) iex> Enum.sum(Enum.filter(Enum.map(1..100_000, &(&1 * 3)),
```

[اعرف المزيد عن الأنابيب](http://elixir-lang.org/docs/stable/elixir/Kernel.html#|>/2/) `_.` المشغل من خلال قراءة وثائقه [lang.org/docs/stable/elixir/Kernel.html#|>/2/](http://elixir-lang.org/docs/stable/elixir/Kernel.html#|>/2/)

التدفقات 10.3

كديل لـ `Enum` توفر وحدة `Stream` التي تدعم العمليات الكسولة:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum 7500000000
```

التدفقات عبارة عن كائنات قابلة للترقيم كسلسلة وقابلة للتكون. فبدلاً من إنشاء قوائم وسيطة، تقوم التدفقات بإنشاء سلسلة من العمليات الحسابية التي يتم استدعاؤها فقط عند تمريرها إلى وحدة Stream. مفيدة عند العمل معمجموعات كبيرة وربما لا نهائية.

إنهم كراسى لأنه، كما هو موضح في المثال أعلاه، ((1..100_000 * &&1) |> Stream.map) يعيد نوع بيانات، وهو مجرد فعلي، يمثل حساب الخريطة على النطاق: 1..100_000.

```
iex> 1..100_000 |> Stream.map(&(&1 * 3))
#Stream<[enum: 1..100000, funs: [<#Function<34.16982430/1
    Stream.map/2>]]>
```

علاوة على ذلك، فهي قابلة للتكون لأننا نستطيع توصيل العديد من عمليات التدفق:

```
iex> retlif.maertSiex> 1..100_000 |> Stream.map(&(&1 * 3)) |>
#Stream<[enum: 1..100000, funs: [...]]>
```

تقبل العديد من الوظائف في وحدة Stream أي عنصر قابل للعد كحجة وتعيد تدفقاً كنتيجية. كما توفر وظائف لإنشاء تدفقات، ربما لا نهائية. على سبيل المثال، يمكن استخدام Stream.cycle() لإنشاء تدفق يقوم بدورة عنصر قابل للعد معين إلى ما لا نهاية. احرص على عدم استدعاء وظيفة مثل Stream.map/2 على مثل هذه التدفقات، حيث ستستمر في الدوران إلى الأبد:

```
iex> stream = Stream.cycle([1, 2, 3])
Stream.cycle/1> iex> Enum.take(stream, 10) [1, 2, 3, 1, 2, 3, 1, 2, 3, 1] في #Function<15.16982430/2
```

من ناحية أخرى، يمكن استخدام Stream.unfold/2 لتوليد القيم من قيمة أولية معينة:

```
iex> stream = Stream.unfold("hello", &String.next_codepoint/1)
#Stream.unfold/2> في #Function<39.75994740/2
iex> Enum.take(stream, 3) ["h", "e", "l"]
```

هناك وظيفة أخرى مثيرة لاهتمام وهي Stream.resource/3 والتي يمكن استخدامها للاتفاق حول الموارد، مما يضمن فتحها قبل التعداد مباشرةً وإغلاقها بعد ذلك، حتى في حالة حدوث أخطاء، على سبيل المثال، يمكننا استخدامها بيت ملف:

```
iex> stream = File.stream!("المسار/إلى/الملف")
Stream.resource/3> iex> Enum.take(stream, 10) في #Function<18.16982430/2
```

سيقوم المثال أعلاه بجلب أول 10 أسطر من الملف الذي حددته. وهذا يعني أن التدفقات يمكن أن تكون مفيدة للغاية للتعامل مع الملفات الكبيرة أو حتى الموارد البطيئة مثل موارد الشبكة.

قد يكون عدد الوظائف والوظائف في وحدتي Stream وEnum أكثرًا شاقًا في البداية، ولكنك ستتعاد عليهمما حالة بحالة. وبشكل خاص، ركز على وحدة Stream وأولاً وانتقل إلىEnum فقط في السيناريوهات المحددة التي تتطلب الكسل للتعامل مع الموارد البطيئة أو المجموعات الكبيرة التي ربما لا نهائية.

سنلقي بعد ذلك نظرة على الميزة الأساسية لبرنامجه Elixir، وهي العمليات، التي تسمح لنا بكتابه برامج متزامنة ومتوازية وموزعة بطريقة سهلة ومفهومة.

عملية 11

toc.html %} تشمل

في Elixir، يتم تشغيل كافة التعليمات البرمجية داخل العمليات. يتم عزل العمليات عن بعضها البعض، ويتم تشغيلها بشكل متزامن مع بعضها البعض وتتواصل عبر تمرير الرسائل. لا تتشكل العمليات الأساسية للتزامن في Elixir فحسب، بل إنها توفر أيضًا الوسائل لبناء برامج موزعة ومقاومة للأخطاء.

لا يبني الخلط بين عمليات Elixir وعمليات نظام التشغيل. العمليات في Elixir خفيفة الوزن للغاية من حيث الذاكرة ووحدة المعالجة المركزية (على عكس الخيوط في العديد من لغات البرمجة الأخرى). وبسبب هذا، فليس من غير المألوف أن يكون هناك عشرات الآلاف من العمليات قيد التشغيل في وقت واحد.

في هذا الفصل، سنتعلم عن البنية الأساسية لتوليد عمليات جديدة، بالإضافة إلى إرسال واستقبال الرسائل بين العمليات المختلفة.

11.1 ظهور

الأداة الأساسية لتوليد العمليات الجديدة هي باستخدام وظيفة `spawn/1` المستوردة تلقائيًا:

```
iex> spawn fn -> 1 + 2 end #PID<0.43.0>
```

يأخذ `1` ووظيفة `spawn/1` سيتم تنفيذها في عملية أخرى.

لاحظ أن `1` يعيد معرف العملية (`PID`) في هذه المرحلة، من المرجح جدًا أن تكون العملية التي أنشأتها ميتة. ستنفذ العملية التي أنشأتها الوظيفة المحددة وتخرج بعد الانتهاء من الوظيفة:

```
iex> pid = spawn fn -> 1 + 2 end #PID<0.44.0>
```

```
iex> Process.alive?(pid) false
```

ملاحظة: من المحتمل أن تحصل على معرفات عملية مختلفة عن تلك التي تحصل عليها في هذا الدليل.

يمكننا استرجاع معرف العملية الحالية عن طريق استدعاء `self/0`:

```
iex> self()
<0.14.0>#معرف العملية
iex> Process.alive?(self())
حققت
```

تصبح العمليات أكثر إثارة للاهتمام عندما نتمكن من إرسال واستقبال الرسائل.

11.2 إرسال والاستقبال

يمكننا إرسال الرسائل إلى عملية ما باستخدام `send/2` واستقبالها باستخدام `Receive/1`:

```
iex> do
  ...> [{:hello, "world"}] iex>
  ...> iex> ("العالم" self(), {:hello, "إرسال"}) iex>
  ...> ("النهاية" ...> {:hello, msg} -> msg ...> {:world, msg} ->
  ...> "عالم")
```

عند إرسال رسالة إلى عملية، يتم تخزين الرسالة في صندوق بريد العملية. تمر كتلة الاستلام `1` عبر صندوق بريد العملية الحالي بحثًا عن رسالة تتطابق مع أي من الأنماط المحددة. تدعم كتلة الاستلام `1` العديد من البنود، مثل `2`. بالإضافة إلى الحراس في البنود.

الإصدار Elixir، توثيق

إذا لم تكن هناك رسالة في صندوق البريد مطابقة لأي من الأنماط، فستنتظر العملية الحالية حتى وصول رسالة مطابقة. يمكن أيضاً تحديد مهلة زمنية:

```
iex> استقبل
...> بعد {hello, msg} -> msg ...
...> لا شيء بعد 1_000 ... < النهاية
...> لا شيء بعد 1
```

يمكن إعطاء مهلة زمنية قدرها 0 عندما تتوقع بالفعل أن تكون الرسالة موجودة في صندوق البريد.

دعونا نجمع كل شيء معاً ونرسل الرسائل بين العمليات:

```
= self()#الاصannel<iex>
<0.14.0>#معرف العملية#
iex> spawn fn -> send(parent, {hello, self()})) end #PID<0.48.0>

استقبلxei
...> "تلقيت تحية من" #inspect pid
...> "تلقيت تحية من" #PID<0.48.0>
```

أثناء وجودك في shell، تجد أن الأداة المساعدة flush هي تقوم بمسح وطباعة جميع الرسائل الموجودة في صندوق البريد.

```
self(), :hello :hello |رسالءiex>
```

```
مرحبا(): jhsulfie<iex>
```

نعم:

الروابط 11.3

الشكل الأكثر شيوعاً للتكرار في Elixir هو في الواقع عبر spawn_link/1، دعنا نحاول معرفة ما يحدث عند فشل إحدى العمليات:

```
iex> spawn fn -> raise "oops" end #PID<0.58.0>
[خطأ] في العملية<0.58.0>: بقيمة الخروج: ...
```

لقد سجل خطأ فقط، لكن عملية التفريخ لا تزال جارية. وذلك لأن العمليات معزولة. إذا أردنا أن ينتشر الفشل في عملية واحدة إلى عملية أخرى، فيجب علينا ربطهما. يمكن القيام بذلك باستخدام spawn_link/1:

```
iex> spawn_link fn -> raise "oops" end #PID<0.41.0>
```

** (RuntimeError) oops :erlang.apply/2: رفع استثناء: #PID<0.41.0> من ** (EXIT

عندما يحدث فشل في shell، يقوم بتحجيز الفشل ويعرضه بتنسيق جيد. لفهم ما سيحدث بالفعل في الكود الخاص بنا، دعونا نستخدم spawn_link/1 داخل ملف ونسفله:

```
#spawn.exs spawn_link fn -> raise "oops" end
```

تلقى القيام

ـ-> hello: "دعونا ننتظر حتى تفشل العملية" النهاية

هذه المرة فشلت العملية وأدت إلى تعطل العملية الأصلية لأنها مرتبطة. يمكن أيضًا إجراء الرابط يدويًا عن طريق استدعاء `Process.link/1` ووصلك بإلقاء نظرة على وحدة `Process`.

تلعب العمليات والروابط دورًا مهمًا عند بناء أنظمة مقاومة للأخطاء، في تطبيقات Elixir. غالباً ما نربط عملياتنا بالمشيرين الذين سيكتشفون متى تموت عملية ما ويبدأون عملية جديدة في مكانها. هذا ممكّن فقط لأن العمليات معزولة ولا تشارك أي شيء بشكل افتراضي. وإذا كانت العمليات معزولة، فلا توجد طريقة يمكن أن يؤدي بها فشل في عملية ما إلى تعطل أو إفساد حالة عملية أخرى.

في حين أن اللغات الأخرى قد تتطلب منا اكتشاف/معالجة الاستثناءات، فإننا في Elixir لا نمانع في ترك العمليات تفشل لأنها متوقعة من المشيرين إعادة تشغيل أنظمتنا بشكل صحيح. "الفشل السريع" هي فلسفة شائعة عند كتابة برامج Elixir!

قبل الانتقال إلى الفصل التالي، دعونا نرى إحدى حالات الاستخدام الأكثر شيوعًا لإنشاء العمليات في Elixir.

11.4 المهام

عند تعطل عملياتنا في القسم السابق، ربما لاحظت أن رسائل الخطأ كانت سببية إلى حد ما:

```
iex> spawn fn -> raise "oops" end #PID<0.58.0>
```

[خطأ] خطأ في العملية <0.58.0> بقيمة الخروج: ...

مع الدالتين `spawn_link/1` و `Task.start_link/1` يتم إنشاء رسائل الخطأ مباشرةً بواسطة الآلة الافتراضية وبالتالي تكون مضغوطة وتفتقر إلى التفاصيل. في الممارسة العملية، يفضل المطورون استخدام الدالتين في وحدة المهام، وبشكل أكثر وضوحاً: `Task.start/1` و `Task.start_link/1`:

```
iex(1)> Task.start fn -> raise "oops" end {:ok, #PID<0.55.0>}
```

[خطأ] بدأت المهمة 15:22:33.046 في #Function<20.90072148/0> وتنهي الوظيفة: من #PID<0.53.0> #PID<0.55.0>

: نرجع (Args: [] **)

* (خطأ وقت التشغيل) عفواً

lib/task/supervised.ex:74: Task.Supervised.do_apply/2 (stdlib) proc_lib.erl:239: :proc_lib.init_p_do_apply/3 (لكسيبر)

بالإضافة إلى توفير تسجيل أفضل للأخطاء، هناك اختلافان آخران: `start_link/1` بدلاً من معرف العملية فقط. وهذا ما يتيح استخدام المهام في شجرة الإشراف. علاوة على ذلك، توفر المهام وظائف ملائمة، مثل `Task.await/1` و `Task.async/1` و `Task.await_timeout/1`.

سوف نستكشف هذه الوظائف في دليل Mix and OTP*. وفي الوقت الحالي يكفي أن نتذكر استخدام المهام للحصول على تسجيل أفضل.

11.5 الدولة

لم نتحدث عن الحالة حتى الآن في هذا الدليل. إذا كنت تقوم ببناء تطبيق يتطلب الحالة، على سبيل المثال، لحفظ على تكوين التطبيق، أو كنت بحاجة إلى تحليل ملف وحفظه فيذاكرة، فأين ستخزنها؟

نُعد العمليات هي الإجابة الأكثر شيوعاً على هذا السؤال. يمكننا كتابة عمليات تتكرر بلا حدود، وتحافظ على الحالة، وترسل وتستقبل الرسائل. على سبيل المثال، دعنا نكتب وحدة نمطية تبدأ عمليات جديدة تعامل كمخزن للقيمة الرئيسية في ملف يسمى: kv.exs:

```
KV
elقاء الوحدة النمطية

def start_link do
  -> loop(%{}) end end nf)knil_trats.المهمة

حلاقة (الخريطة) لا تستقبل لا

caller) -> send caller, Map.get(map, key) loop(map) {:put, key, value} ->
  {:get, key,}

حلاقة (الخريطة، المفتاح، القيمة)
نهاية
نهاية
نهاية
```

لاحظ أن دالة start_link تفخر في الأساس عملية جديدة تقوم بتشغيل دالة loop/1 أبداً بخريطة فارغة. تنتظر دالة loop/1 الرسائل ثم تقوم بالإجراء المناسب لكل رسالة. في حالة وجود رسالة: teg، ترسل رسالة مرة أخرى إلى المتصل وتستدعي دالة loop/1 مرة أخرى، لانتظار رسالة جديدة.

في حين أن رسالة tup: تستخدم في الواقع loop/1 لإصدار جديد من الخريطة، مع تخزين المفتاح والقيمة المحددين.

دعنا نحاول تشغيل iex kv.exs:

```
{:get, :hello, #PID<0.41.0>} رسال iex> {:ok, pid} = KV.start_link #PID<0.62.0> iex>
  pid. {:get, :hello, self()}

اندفق
لا شيء
```

في البداية، لا تحتوي خريطة العملية على مفاتيح، لذا فإن إرسال رسالة: teg ثم مسح صندوق الوارد للعملية الحالية يعيد القيمة صفرًا. لترسل رسالة tup: ونحاول مرة أخرى:

```
{:put, :hello, :world} رسال معرف العملية, iex>
{:get, :hello, self()} {:get, :hello, #PID<0.41.0>} #أرسل معرف العملية, iex>

اندفق
:عالم
```

لاحظ كيف تحافظ العملية على حالة معينة ويمكننا الحصول على هذه الحالة وتحديدها عن طريق إرسال رسائل إلى العملية. في الواقع، أي عملية تعرف معرف العملية أعلاه ستكون قادرة على إرسال رسائل إليها والتلاعب بالحالة.

من الممكن أيضاً تسجيل معرف العملية وإعطائه اسمًا والسماح لأي شخص يعرف الاسم بإرسال رسائل إليه:

```
iex> Process.register(pid, :kv)
رسال مسح {:get, :hello, self()} {:get, :hello, #PID<0.41.0>} iex> .vk: رسال iex>
:عالم
```

بعد استخدام العمليات المتعلقة بتسجيل الحالة والاسم من الأنماط الشائعة جداً في تطبيقات Elixir، ومع ذلك، في أغلب الأحيان، لن ننفذ هذه الأنماط يدوياً كما هو مذكور أعلاه، ولكن باستخدام أحد التجريدات العديدة التي

السفن التي تحتوي على Elixir على سبيل المثال، توفر Elixir وكلاء عبارة عن تجريدات بسيطة حول الحالة:

```
elix> {:ok, pid} = Agent.start_link(fn -> %{} end) {:ok, #PID<0.72.0>} iex> Agent.update(pid, fn map -> Map.put(map, :hello, :world) end)
```

نعم:

```
(النهائية<iex> Agent.get(pid, fn map -> Map.get(map, :hello))
```

عالم:

يمكن أيضًا إعطاء خيار `eman` إلى `Agent.start_link` وシステム تسجيله تلقائيًا. بالإضافة إلى الوكالة، يوفر Elixirواجهة برمجة تطبيقات لبناء خوادم عامة (تسمى `GenServer`). ومديري أحداث عامة ومعالجات أحداث (تسمى `GenEvent`). ومهام والمزيد، وكلها مدرومة بالعمليات الموجودة أسفلها. سيتم استكشاف هذه العمليات، جنباً إلى جنب مع أشجار الإشراف، بمزيد من التفصيل في دليل `OTP*` الذي سينتicipateMix and OTP

إلى النهاية.

في الوقت الحالي، دعنا ننتقل إلى استكشاف عالم الإدخال/الإخراج في Elixir.

12.1 ونظام الملفات IO

`toc.html` تشمل {%

يعتبر هذا الفصل مقدمة سريعة للآليات الإدخال/الإخراج والمهام المرتبطة بنظام الملفات، بالإضافة إلى الوحدات النمطية ذات الصلة مثل `Path.html`، `IO`، `File.html`، `Path` و `IO`.

لقد خططنا في الأصل لهذا الفصل ليأتي في وقت مبكر جدًا في دليل البدء. ومع ذلك، لاحظنا أن نظام الإدخال والإخراج يوفر فرصة رائعة لإلقاء الضوء على بعض فلسفات وفضوليات Elixir والألة الافتراضية.

12.1.1 وحدة الإدخال والإخراج

وحدة الإدخال/الإخراج هي الآلية الرئيسية في Elixir لقراءة وكتابة الإدخال/الإخراج القياسي (`:stdio`) والملفات وأجهزة الإدخال/الإخراج الأخرى. استخدام الوحدة بسيط للغاية:

"مرحبا بالعالم" `iex> IO.puts`

نعم:

"نعم أم لا؟ نعم" `iex> IO.gets`

يشكل افتراضي، تقوم الوظائف في وحدة الإدخال والإخراج بالقراءة من الإدخال القياسي والكتابة إلى الإخراج القياسي. يمكننا تغيير ذلك عن طريق تمرير، على سبيل المثال، `rreds` كحجّة (للكتابة إلى جهاز الخطأ القياسي):

"مرحبا بالعالم" `iex> IO.puts :stderr.`

نعم:

12.2.1 وحدة الملف

تحتوي وحدة "ملف" `File.html` على وظائف تسمح لنا بفتح الملفات كأجهزة إدخال/إخراج. بشكل افتراضي، يتم فتح الملفات في الوضع الثنائي، الأمر الذي يتطلب من المطوريين استخدام وظائف `IO.binread/2` و `IO.binwrite/2` المحددة من وحدة `IO`:

```
iex> {:ok, file} = File.open "hello", [write: true, #PID<0.47.0>]
iex> IO.binwrite file, "world"
iex>
```

```
امثلف..إغلاق الملف
نهم
امثلف..قراءة "مرحبا"
{"حسنا، العالم"}</pre>
```

يمكن أيضًا فتح الملف باستخدام تميز: 8ftu، والذي يخبر وحدة الملف بتفسير البيانات المقرؤة من الملف كبيانات مشفرة يتميز UTF-8.

بالإضافة إلى الوظائف الخاصة بفتح الملفات وقراءتها وكتابتها، تحتوي وحدة الملف على العديد من الوظائف للعمل مع نظام الملفات. يتم تسمية هذه الوظائف على اسم نظيراتها في بونكس. على سبيل المثال، يمكن استخدام `File.rm/1` لإزالة الملفات، `File.mkdir_p/1` وإنشاء الدلائل وجميع سلاسلها الأصلية. حتى أن هناك `File.cp_r/2` لنسخ وإزالة الملفات والدلائل بشكل متكرر (أي نسخ وإزالة محتويات الدلائل أيضًا).

ستلاحظ أيضًا أن الوظائف في وحدة الملف لها نسختان: نسخة "عادية" ونسخة أخرى تحمل نفس اسم النسخة العادية ولكن بعلامة تعجب. (!) على سبيل المثال، عندما نقرأ ملف `File.read!/1` أو يمكننا استخدام `File.read/1` في المثال أعلاه، نستخدم "hello" في المثال أعلى.

```
امثلف..قراءة "مرحبا"
{"حسنا، العالم"}</pre>
امثلف..قراءة! "مرحبا"
"عالم"
الملف، اقرأ "غير معروف"
{tneone: خطأ}
امثلف..قراءة! "غير معروف"
"غير معروف"</pre>
```

*+ لم يتمكن من قراءة الملف غير المعروف: لا يوجد مثل هذا الملف أو الدليل (File.Error)

لاحظ أنه عندما لا يكون الملف موجودًا، فإن الإصدار الذي يحمل علامة ! يثير خطأً. يفضل الإصدار الذي لا يحمل علامة ! عندما تريد التعامل مع نتائج مختلفة باستخدام مطابقة الأنماط:

```
read(file) do
  # الحال الملف.
  # افعل شيئاً ما مع الجسم
  `reason` end # معالجة الخطأ الناتج عن ok, body {error, reason} ->
```

مع ذلك، إذا كنت تتوقع وجود الملف هناك، فإن الاختلاف في "النقرة" يكون أكثر فائدًا لأنه يرفع رسالة خطأ ذات معنى. تجنب الكتابة:

```
(ok, body) =
```

كما هو الحال في حالة حدوث خطأ، سيعيد `File.read/1` `{error, reason}` وستفشل عملية مطابقة النمط. ستظل تحصل على النتيجة المطلوبة (خطأً بارز)، لكن الرسالة ستكون حول النمط الذي لا يتتطابق (وبالتالي تكون غامضة فيما يتعلق بما يتعلق به الخطأ بالفعل).

إذا كنت لا ت يريد التعامل مع خطأ محتمل (أي أنك ت يريد ظهوره)، فمن الأفضل استخدام `File.read!/1`.

12.3 وحدة المسار

تتوقع أغلب الوظائف في وحدة الملف مسارات كحجج. وفي أغلب الأحيان، تكون هذه المسارات عبارة عن ملفات ثنائية عادية. توفر وحدة `Path` تسهيلات للعمل مع مثل هذه المسارات:

```
iex> Path.join("foo", "bar") "foo/bar" iex> Path.expand("~/hello")
```

/المستخدمين/مرحبا/جذبها

يُفضل استخدام الوظائف من وحدة Path بدلاً من مجرد معالجة الملفات الثنائية، وذلك لأن وحدة Path تعني بأنظمة التشغيل المختلفة بشفافية. على سبيل المثال، تقوم 2 Windows بربط مسار باستخدام الشرطة المائلة (/) على أنظمة تشبيه Unix وباستخدام الشرطة المائلة العكسية (\) على أنظمة Elixir.

بهذا تكون قد غطينا الوحدات الرئيسية التي يوفرها Elixir للتعامل مع عمليات الإدخال والإخراج والتفاعل مع نظام الملفات. في الأقسام التالية، سنناقش بعض الموضوعات المتقدمة المتعلقة بعمليات الإدخال والإخراج. هذه الأقسام ليست ضرورية لكتابة كود Elixir، لكنها توفر نظرة عامة جيدة حول كيفية تنفيذ نظام الإدخال والإخراج في الجهاز الافتراضي وغير ذلك من الأمور الغريبة.

12.4.1 العمليات وقادرة المجموعات

ربما لاحظت أن `File.open/2` يعيد مجموعة مثل `{:ok, pid}`:

```
iex> {:ok, file} = File.open "hello", [:write] {:ok, #PID<0.47.0>}
```

يحدث ذلك لأن وحدة الإدخال والإخراج تعمل بالفعل مع العمليات (انظر الفصل 11). عندما تكتب `IO.write(pid, binary)`. عند ذلك ترسل وحدة الإدخال والإخراج رسالة إلى العملية التي تم تحديدها بواسطة `pid` بالعملية المطلوبة. دعنا نرى ماذا يحدث إذا استخدمنا عملية خاصة بنا:

```
iex> pid = spawn fn -> ...> Receive do: (msg -> IO.inspect msg) ...> end
```

<0.75.0> #معرف العملية

```
iex> IO.write(pid, "مرحبا"){:io_request, #PID<0.41.0>, #PID<0.57.0>, {:put_chars, :unicode, ("مرحبا")}}
```

erlang : :terminated خطأ ErlangError

بعد `IO.write/2` يمكننا رؤية الطلب الذي أرسلته وحدة الإدخال والإخراج (مجموعة من أربعة عناصر) مطبوعاً. بعد ذلك بفترة وجيزة، نرى أنه فشل لأن وحدة الإدخال والإخراج توقفت نوغاً ما من النتائج التي لم نوفرها.

توفر وحدة `StringIO` تفاصلاً لرسائل جهاز الإدخال/الإخراج أعلى السلاسل:

```
iex> {:ok, pid} = StringIO.open("hello") {:ok, #PID<0.43.0>}
```

"هو"

من خلال نسخة أجهزة الإدخال والإخراج باستخدام العمليات، تسمح آلة Erlang الافتراضية للعقد المختلفة في نفس الشبكة بتبادل عمليات الملفات من أجل قراءة/كتابة الملفات بين العقد. ومن بين جميع أجهزة الإدخال والإخراج، يوجد جهاز واحد خاص بكل عملية: قائد المجموعة.

عندما تكتب إلى `oidts`، فأنت في الواقع ترسل رسالة إلى قائد المجموعة، الذي يكتب إلى موصوف ملف الإدخال القياسي:

```
iex> IO.puts :stdio, "مرحبا"
مرحبا
نعم:
iex> IO.puts Process.group_leader, "مرحبا"
نعم:
```

يمكن تكوين قائد المجموعة لكل عملية على حدة، وُستخدم في مواقف مختلفة. على سبيل المثال، عند تنفيذ التعليمات البرمجية في محطة طرفية بعيدة، فإنه يضمن إعادة توجيه الرسائل في العقدة البعيدة وطباعتها في المحطة الطرفية التي أطلقت الطلب.

12.5. اليودانا والشارданا

في جميع الأمثلة المذكورة أعلاه، استخدمنا الثنائيات عند الكتابة إلى الملفات. في الفصل "الثنائيات والسلسل وقوائم الأحرف"، ذكرنا كيف أن السلسل عبارة عن بايات فقط بينما قوائم الأحرف عبارة عن قوائم تحتوي على نقاط رمز.

تسمح الوظائف في IO وأيضاً بإعطاء القوائم كحجج ليس هذا فحسب، بل تسمح أيضاً بإعطاء قائمة مختلطة من القوائم والأعداد الصحيحة والثنائية:

```
iex> IO.puts "مرحبا بالعالم"
:ok iex> IO.puts ['hello', ?s, "world"]
```

مرحبا بالعالم
نعم:

ومع ذلك، يتطلب هذا بعض الاهتمام، قد تمثل القائمة إما مجموعة من البايات أو مجموعة من الأحرف، ويعتمد استخدام أي منها على تميز جهاز الإدخال/الإخراج، إذا تم فتح الملف بدون تميز، فمن المتوقع أن يكون الملف في وضع غير معالج، ويجب استخدام الوظائف في وحدة الإدخال/الإخراج التي تبدأ بـ*databin*. تتوقع هذه الوظائف أي أنها تتوقع تقديم قائمة من الأعداد الصحيحة التي تمثل الثنائيات والثنائيات.

من ناحية أخرى، تعمل ملفات idts:char_data والملفات المفتوحة يتميز 8ftu: مع الوظائف المتبقية في وحدة الإدخال والإخراج. تتوقع هذه الوظائف أي قائمة من الأحرف أو السلسل.

على الرغم من أن هذا الاختلاف بسيط، إلا أنك لست بحاجة إلى القلق بشأن هذه التفاصيل إلا إذا كنت تنوی تمرير القوائم إلى هذه الوظائف. يتم تمثيل الثنائيات بالفعل بواسطة البايات الأساسية وبالتالي فإن تمثيلها يكون دائماً خاماً.

وبهذا تكون قد أنهينا جولتنا في أجهزة الإدخال والإخراج والوظائف المرتبطة بها. لقد تعلمنا عن أربع وحدات elixir/StringIO.html: - elixir/IO.html: - elixir/Path.html: - elixir/File.html: بالإضافة إلى كيفية استخدام الآلة الافتراضية للعمليات الإدخال والإخراج الأساسية وكيفية استخدام datachar وdataiodata لعمليات الإدخال والإخراج.

13. اسم مستعار، يتطلب ويستورد

toc.html %} تشمل

Elixir توسيعاته، كما سنرى أدناه، تُسمى هذه التوجيهات لأنها ذات نطاق معجمي.

13.1. الاسم المستعار

يتيح لك alias إعداد أسماء مستعارة لأي اسم وحدة معينة. تخيل أن وحدة الرياضيات الخاصة بنا تستخدم تنفيذ قائمة خاصة لإجراء عمليات رياضية محددة:

```
defmodule الرياضيات تفعيل
الاسم المستعار مثلاً: نهاية القائمة
```

من الآن فصاعداً، سيتم توسيع أي مرجع إلى القائمة تلقائياً إلى Math.List. في حالة الرغبة في الوصول إلى القائمة الأصلية، يمكن القيام بذلك عن طريق إضافة بادئة Elixir إلى اسم الوحدة النمطية:

```

Math.List.flatten
قائمة تسلیح
List.flat[إكتشاف قائمة تسلیح]
Math.List.flattenElixir.Math.List.flatten #=>

ملاحظة: يتم تعرف جميع الوحدات النمطية المحددة في Elixir داخل Elixir الرئيسي
و مع ذلك، من أجل الراحة، يمكنك حذف "Elixir." عند الإشارة إليها.

```

تُستخدم الأسماء المستعارة بشكل متكرر لتحديد الاختصارات، في الواقع، يؤدي استدعاء `alias` إلى تعين الاسم المستعار تلقائياً إلى الجزء الأخير من اسم الوحدة النمطية، على سبيل المثال:

`Math.List` الاسم المستعار

هو نفس الشيء:

`List` الاسم المستعار `Math.List` مثل:

لاحظ أن `alias` محدد النطاق معجّماً، مما يسمح لك بتعيين أسماء مستعارة داخل وظائف محددة:

```

defmodule الرياضيات
alias Math.List

# نهاية ...
def minus(a, b) do # ...
# نهاية

```

في المثال أعلاه، نظراً لأننا نستدعي الاسم المستعار داخل الدالة `minus/2`، فسيكون الاسم المستعار صالحًا فقط داخل الدالة `minus/2`. `plus/2` ولن يتأثر على الإطلاق.

13.2 ينطوي

يوفر Elixir وحدات الماكرو كآلية للبرمجة الوصفية (كتابة التعليمات البرمجية التي تولد التعليمات البرمجية).

وحدات الماكرو هي أجزاء من التعليمات البرمجية التي يتم تنفيذها وتوسيعها في وقت التجميع، وهذا يعني أنه لاستخدام وحدة ماكرو، تحتاج إلى ضمان توفر وحدتها وتنفيذها أثناء التجميع. يتم ذلك باستخدام التوجيه `require`:

```

iex> Integer.is_odd(3)
Integer.is_odd/1 iex> require Integer nil ** يجب عليك طلب قبل استخدام الماكرو (CompileError) iex:1:

iex> Integer.is_odd(3)

```

في Elixir، يتم تعریف `Integer.is_odd/1` كوحدة ماكرو حتى يمكن استخدامها كحارس. وهذا يعني أنه لااستدعاء `Integer.is_odd/1` يحتاج أولاً إلى طلب وحدة `Integer`.

بشكل عام، لا يلزم أن تكون الوحدة النمطية مطلوبة قبل الاستخدام، إلا إذا أردنا استخدام وحدات الماكرو المتوفرة في تلك الوحدة النمطية. ستؤدي محاولة استدعاء وحدة ماكرو لم يتم تحميلها إلى ظهور خطأ. لاحظ أنه مثل توجيه `require` فإن `alias` أيضاً له نطاق معجمي، سنتحدث أكثر عن وحدات الماكرو في قصل لاحق.

13.3 استيراد

نستخدم `import` عندما نريد الوصول بسهولة إلى الوظائف أو وحدات الماكرو من وحدات نمطية أخرى دون استخدام الاسم المؤهل بالكامل. على سبيل المثال، إذا أردنا استخدام `List.duplicate/2` من وحدة نمطية `Erlang` (والتي تمثلها `Elixir`)، فيمكننا ببساطة استيرادها:

```
astirad القائمة فقط: [مكرر: 2] شيء ex>
```

```
3 [:ok, :ok, :ok], ko: ex>
```

في هذه الحالة، نقوم باستيراد الدالة المكررة فقط (بقيمة 2) من القائمة. على الرغم من أن `arity` اختياري، إلا أنه يوصى باستخدامه لتجنب استيراد جميع وظائف وحدة معينة داخل مساحة الأسماء. يمكن أيضًا إعطاء `tpecxe`: ك الخيار لاستيراد كل شيء في وحدة ما باستثناء قائمة من الوظائف.

يدعم الاستيراد أيضًا `functions`: على سبيل المثال، لاستيراد كل وحدات الماكرو، يمكن للمرء أن يكتب:

```
astirad عدد صحيح فقط: sorcam
```

أو لاستيراد كافة الوظائف، يمكنك كتابة:

```
astirad عدد صحيح فقط: snoitcnuf
```

لاحظ أن الاستيراد له نطاق معجمي أيضًا. وهذا يعني أنه يمكننا استيراد وحدات ماكرو أو وظائف محددة داخل تعريفات الوظائف:

```
defmodule الرياضيات تفعل
  def some_function do
    استيراد القائمة فقط: [مكرر: 10], ko: ()
```

نهاية
نهاية

في المثال أعلاه، تكون قائمة `List.duplicate/2` المستوردة مرئية فقط داخل هذه الوظيفة المحددة. لن تكون قائمة `String` (أو أي وحدة أخرى في هذا الشأن).

لاحظ أن استيراد الوحدة يتطلب ذلك تلقائيًا.

13.4 الأسماء المستعارة

في هذه المرحلة قد تتساءل: ما هو الاسم المستعار `Elixir` بالضبط وكيف يتم تمثيله؟

الاسم المستعار في `Elixir` هو معرف مكتوب بأحرف كبيرة (مثل `String` أو `Keyword`) يتم تحويله إلى ذرة أثناء التجميع. على سبيل المثال، يتم ترجمة الاسم المستعار `"Elixir.String"` إلى الذرة: `String`

```
سلسلة(mota_siie>
حقيني
سلسلة(gnirts_otie>
سلسلة صحيحة"Elixir.String" iex> :"Elixir.String" ==
```

من خلال استخدام التوجيه `alias/2`، فإننا ببساطة نقوم بتغيير ما يترجم إليه الاسم المستعار.

تعمل الأسماء المستعارة كما هو موضح لأن الوحدات النمطية في `Erlang VM` (والتي تمثلها `Elixir`) يتم ترجمتها بواسطة الذرات. على سبيل المثال، هذه هي الآلية التي نستخدمها لاستدعاء وحدات `Erlang` النمطية:

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

وهذه أيضًا هي الألية التي تسمح لنا باستدعاء وظيفة معينة بشكل ديناميكي في وحدة نمطية:

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

نحن ببساطة نقوم باستدعاء الدالة flatten على الذرة .stsil.

13.5 التعشيش

الآن بعد أن تحدثنا عن الأسماء المستعارة، يمكننا أن نتحدث عن التعشيش وكيفية عمله في Elixir. فكر في المثال التالي:

```
إلغاء الوحدة النمطية fo دو
defmodule shriek
  ^^^
نهاية
نهاية
```

سيحدد المثال أعلاه وحدتين: Foo و Bar. يمكن الوصول إلى الوحدة الثانية باعتبارها Bar داخل Foo بينما أنهما في نفس النطاق المعجمي.

إذا تم نقل وحدة Bar خارج تعريف وحدة Foo، فسوف تحتاج إلى الإشارة إليها باسمها الكامل (Foo.Bar) أو سيتعين تعين اسم مستعار باستخدام توجيه الاسم المستعار الذي تمت مناقشه أعلاه. سيتغير تعريف وحدة Bar أيضًا. هذا الكود يعادل المثال أعلاه:

```
defmodule Foo.Bar do
  ^^^
  ^
  إلغاء الوحدة النمطية fo دو
  ^
  Bar end  مثل: Foo.Bar.
  ^
  الاسم المستعار
```

الكود أعلاه هو نفس الكود السابق تماماً:

```
Elixir.Foo do
  ^^^
  ^
  إلغاء وحدة
Elixir.Foo.Bar
  ^^^
  ^
  نهاية
Bar end  مثل: Elixir.Foo.Bar.
  ^
  الاسم المستعار
```

ملاحظة: في Elixir، ليس عليك تعريف وحدة Foo قبل أن تتمكن من تعريف وحدة Bar، حيث تقوم اللغة بترجمة جميع أسماء الوحدات إلى درات على أي حال. يمكنك تعريف وحدات متداخلة بشكل عشوائي دون تعريف أي وحدة في السلسلة (على سبيل المثال، Foo.Bar.Baz، دون تعريف Foo أو Bar أو Baz).

كما سنرى في الفصول اللاحقة، تلعب الأسماء المستعارة أيضًا دورًا حاسماً في وحدات الماكرو، لضمان صحتها.

بهذا تكون قد أنهينا تقريرًا جولتنا حول وحدات Elixir. الموضوع الأخير الذي سنتناوله هو سمات الوحدات.

14 سمات الوحدة

toc.html %} تشمل {%

تخدم سمات الوحدة النمطية في Elixir ثلاثة أغراض:

1. تستخدم لشرح الوحدة النمطية، غالباً بالمعلومات التي يمكن للمستخدم أو الجهاز الافتراضي استخدامها.

2. تعلم كالثوابت.

3. تعمل كوحدة تخزين مؤقتة يمكن استخدامها أثناء التجميع.

دعونا نتحقق من كل حالة على حدة.

14.1 تعليمات توضيحية

يستعين Elixir بمفهوم سمات الوحدة النمطية من Erlang على سبيل المثال:

```
@vsn 2
defmodule MyServer
```

نهاية

في المثل أعلى، تقوم بتعيين سمة الإصدار صراحةً لتلك الوحدة. يتم استخدام `@vsn` بواسطة آلية إعادة تحميل التعليمات البرمجية في Erlang VM للتحقق مما إذا كانت الوحدة قد تم تحديدها أم لا. إذا لم يتم تحديد أي إصدار، يتم تعيين الإصدار على مجموع الاختبار MD5 لوظائف الوحدة.

يحتوي Elixir على عدد قليل من السمات الممحورة. فيما يلي بعض منها، وهي الأكثر استخداماً:

- يوفر `@moduledoc` توثيقاً للوحدة الحالية.

- يوفر `@doc` توثيقاً للوظيفة أو المacro الذي يتبع السمة.

- `@behaviour` (لاحظ التهجئة البريطانية) تستخدم لتحديد سلوك OTP أو سلوك محدد من قبل المستخدم.

- يوفر خطأً سبتم استدعاؤه قبل تجميع الوحدة النمطية. وهذا يجعل من الممكن حقن الوظائف داخل الوحدة النمطية قبل التجميع تماماً.

`codeword` @`@moduledoc` هما أكثر السمات استخداماً على الإطلاق، وتتوقع منك استخدامهما كثيراً. تتعامل Elixir مع الوثائق باعتبارها من الدرجة الأولى وتتوفر العديد من الوظائف للوصول إلى الوثائق.

دعنا نعود إلى وحدة الرياضيات المحددة في الفصول السابقة، ونضيف بعض الوثائق ونحفظها في ملف `math.ex`:

```
defmodule الرياضيات
  @moduledoc """
  يوفر وظائف متعلقة بالرياضيات.
  # أمثلة
```

```
(1, 2) مجموع الرياضيات iex>
```

3

```
.....
@doc """
بحسب مجموع رقمين.
```

```
.....
def sum(a, b), do: a + b
نهاية
```

يشجع Elixir استخدام تميز `heredocs` مع Markdown لكتابية وثائق قابلة للقراءة. تكون `heredocs` من سلاسل متعددة الأسطر، تبدأ وتنتهي بعلامات اقتباس ثلاثية، مع الحفاظ على تنسيق النص الداخلي. يمكننا الوصول إلى وثائق أي وحدة مجتمعة مباشرة من IEx:

```
$ elixirc math.ex $ iex
```

```
... الوصول إلى المستندات الخاصة بوحدة الرياضيات h Math #
... الوصول إلى المستندات الخاصة بوحدة المجموع h Math.sum #
...
```

نحن نقدم أيضًا أداة تسمى `ExDoc` والتي يتم استخدامها لإنشاء صفحات HTML من الوثائق.

يمكنك إلقاء نظرة على مستندات الوحدة النمطية للحصول على قائمة كاملة بالسمات المدعومة. يستخدم `Elixir` أيضًا السمات لتحديد مواصفات النوع، عبر:

- يوفر مواصفات لوظيفة `@spec` -
- يوفر مواصفات لاستدعاء السلوك `@callback` -
- يحدد النوع الذي سيتم استخدامه في `@type` -
- يحدد نوعًا خاصًا سيتم استخدامه في `@typep` -
- يحدد نوعًا غير شفاف سيتم استخدامه في `@opaque` -

يتناول هذا القسم السمات المضمنة. ومع ذلك، يمكن للمطورين أيضًا استخدام السمات أو توسيعها بواسطة المكتبات لدعم السلوك المخصص.

14.2 كثوابت

غالبًا ما يستخدم مطورو `Elixir` سمات الوحدة النمطية لاستخدامها كثوابت:

```
defmodule MyServer do
  @initial_state %{host: "147.0.0.1", ...}
  IO.inspect @initial_state
```

ملاحظة: على عكس إرلانج، لا يتم تخزين السمات التي يحددها المستخدم في الوحدة النمطية بشكل افتراضي، القيمة موجودة فقط أثناء وقت التجميع. يمكن للمطور تكوين سمة لتتصرف بشكل أقرب إلى إرلانج عن طريق استدعاء `Module.register_attribute/3`

ستؤدي محاولة الوصول إلى سمة لم يتم تعريفها إلى طباعة تحذير:

```
@unknown ا@defmodule MyServer
نهاية
    تحذير: سمة الوحدة غير المحددة. يرجى إزالة الوصول إلى @unknown او اني يعيت الصفر قبل الوصول
```

أخيرًا، يمكن أيضًا قراءة السمات داخل الوظائف:

```
@my_data 14 def first_data, _ do
  @my_data 13 def second_data, _ do
    ...
```

نهاية

```
MyServer.first_data => 14
MyServer.second_data => 13
```

لاحظ أن قراءة سمة داخل دالة ما تأخذ لقطة من قيمتها الحالية. بعبارة أخرى، تم قراءة القيمة في وقت التجميع وليس وقت التشغيل. وكما سنرى، فإن هذا يجعل السمات مفيدة للاستخدام كتخزين أثناء تجميع الوحدة النمطية.

14.3 مؤقت تخزين

أحد المشاريع في منظمة Elixir هو مشروع Plug <https://github.com/elixir-lang/plug> وهو ما يهدف إلى أن يكون بمثابة أساس مشترك لبناء مكتبات الويب وأطر العمل في Elixir.

تتيح مكتبة Plug أيضًا للمطورين تعريف المكونات الإضافية الخاصة بهم والتي يمكن تشغيلها في خادم الويب:

```
Plug.Builder defmodule MyPlug
  . . .
  ko_dnes: المكونات redaeh_tes

  . . .
  فقم بذلك def set_header(conn, _opts)
  put_resp_header(conn, "x-header", "set")
  نهاية

  def send_ok(conn, _opts) do
    send(conn, 200, "ok")
  end

  . . .
نهاية

  http://localhost:4000 "على IO.puts
  مع شغيل MyPlugCowboy "IO.puts
  Plug.Adapters.Cowboy.http MyPlug. []
```

في المثال أعلاه، استخدمنا المacro plug لتوصيل الوظائف التي سيتم استدعاؤها عند وجود طلب ويب. داخليًا، في كل مرة تقوم فيها باستدعاء `1/Plug` تخزن مكتبة Plug الوسيطة المحددة في سمة `@plugs`. يقوم Plug بتشفير معاودة اتصال تحدد طريقة `(call/2)` تتعامل مع طلبات `http://` قبل تجميع الوحدة النمطية مباشرة. ستقوم هذه الطريقة بتشغيل جميع المقاييس الموجودة داخل `MyPlug@plugs` بالترتيب.

لفهم الكود الأساسي، نحتاج إلى وحدات Macro، لذا سنعود إلى هذا النمط في دليل البرمجة الوصفية. ومع ذلك، فإن التركيز هنا ينصب على كيفية استخدام سمات الوحدة النمطية كمخزن للسماح للمطورين بإنشاء لغات مجال محددة.

يأتي مثال آخر من إطار عمل ExUnit الذي يستخدم سمات الوحدة النمطية كتعليق وتخزين:

```
ExUnit.Case defmodule MyTest
  . . .
  اختبار خارجي "اتصالات بالخدمة الخارجية" @tag :
  . . .
  نهاية ...
```

تُستخدم العلامات في ExUnit لشرح الاختبارات. ويمكن استخدام العلامات لاحقًا لتصفية الاختبارات. على سبيل المثال، يمكنك تجنب تشغيل الاختبارات الخارجية على جهازك لأنها بطيئة وتعتمد على خدمات أخرى، بينما لا يزال من الممكن تمكينها في نظام البناء الخاص بك.

نأمل أن يلقي هذا القسم الضوء على كيفية دعم Elixir للبرمجة الوصفية وكيف تلعب سمات الوحدة دورًا مهمًا عند القيام بذلك.

في الفصول التالية، سنستكشف الهياكل والبروتوكولات قبل الانتقال إلى معالجة الاستثناءات والهياكل الأخرى مثل الرموز والفهم.

15. هيكل

toc.html %} تشمل

في الفصل السابع تعلمنا عن الخرائط:

```
2} iex> %{:b1, :b2} = %{:b1, :b2}
          الخريطة
          الخريطة[:b1]
          [:b2]

1
2} %{:b3, :b1} = %{:b1, :b3}
          خريطة | :b1
          خريطة | :b3
```

الهيأكل هي ملحقات مبنية على الخرائط التي توفر عمليات التحقق في وقت التجميع والقيم الافتراضية.

تعريف الهيأكل

لتحديد هيكل، يتم استخدام بنية defstruct:

```
do
  defmodule المستخدم do
    27 ...> end
    ...> defstruct name: "John".
```

قائمة الكلمات الرئيسية المستخدمة مع defstruct تحدد الحقول التي سيحتوي عليها الهيكل بالإضافة إلى قيمها الافتراضية.

تأخذ الهيأكل اسم الوحدة التي تم تعريفها فيها. في المثال أعلاه، قمنا بتعريف هيكل باسم User.

يمكننا الآن إنشاء هيأكل المستخدم باستخدام بناء جملة مشابه لذلك المستخدم لإنشاء الخرائط:

```
 المستخدم{iex>
{"ميج", 27}resU%{resU%}iex>
{"جون", 27}resU%{resU%}iex>
{"جون", 27}resU%
```

توفر الهيأكل صيغات وقت التجميع بأنه سيتم السماح فقط للحقول (وكلها) المحددة من خلال defstruct بالوجود في الهيكل:

```
iex> %User{oops: :field}
      ^
      ** خطأ في التجميع (Unknown field oops)
```

15.2 الوصول إلى الهيأكل وتحديثها

عندما ناقشنا الخرائط، أظهرنا كيف يمكننا الوصول إلى حقول الخريطة وتحديثها. تطبيق نفس التقنيات (ونفس بناء الجملة) على الهيأكل أيضًا:

```
= %User{john: %User{name: "جون", age: 27}}
iex> john.name = "ميج"
      ^
      ** خطأ في التجميع (Unknown field name)
      |
      {dieif: "عفواً", meg: %{age: 27}}
iex> %{meg: "ميج"}resU%
```

* خطأ في الحجة (ArgumentError)

عند استخدام صيغة التحديث، (|=) تدرك الآلة الافتراضية أنه لن يتم إضافة مفاتيح جديدة إلى البنية، مما يسمح للخرائط الموجودة أسفلها بمشاركة بنيتها في الذاكرة. في المثال أعلاه، يتشارك كل من جون وميج نفس بنية المفتاح في الذاكرة.

يمكن أيضًا استخدام الهياكت في مطابقة الأنماط، سواء للمطابقة على قيمة مفاتيح محددة أو للتأكد من أن القيمة المطابقة هي هيكل من نفس نوع القيمة المطابقة.

```
iex> %User{name: name} = john
          "جون"
          =
          %{}% المستخدم iex>
          لا يوجد تطابق لقيمة الجانب الأيمن : {} (MatchError)
```

15.3 الهياكت هي مجرد خرائط عارية أسفلها

في المثال أعلاه، تعمل مطابقة الأنماط لأن الهياكت الموجودة أسفلها عبارة عن خرائط عارية تحتوي على مجموعة ثابتة من الحقول. باعتبارها خرائط، تخزن الهياكت حفلاً "خاصًا" يسمى `__struct__` والذي يحمل اسم الهيكل:

```
صحيح(pam_si) iex>
john = %User{age: 27, name: "جون"}
john.__struct__
```

لاحظ أننا أشرنا إلى الهياكت باعتبارها خرائط عارية لأن أيًا من البروتوكولات التي تم تنفيذها للخرائط غير متوافرة للهياكت. على سبيل المثال، لا يمكنك تعداد هيكل أو الوصول إليه:

```
john = %User{age: 27, name: "جون"} iex>
john.age
Protocol.UndefinedError: لم يتم تنفيذ الوصول إلى البروتوكول
```

الهيكل ليس قاموسًا أيضًا وبالتالي لا يمكن استخدامه مع الوظائف من وحدة `Dict`:

```
iex> Dict.get(%User{}, :name)
User.fetch/2: دالة غير محددة
```

ومع ذلك، نظرًا لأن الهياكت عبارة عن خرائط فقط، فإنها تعمل مع الوظائف من وحدة `Map`:

```
iex> kurt = Map.put(%User{}, :name, "Kurt")
Map.merge(kurt, %User{age: 27, name: "كورت"}) iex>
Map.keys(john) [__struct__, :age, :name]
```

ستتناول كيفية تفاعل الهياكت مع البروتوكولات في الفصل التالي.

16 بروتوكولات

`toc.html` تشمل {%

البروتوكولات هي آلية لتحقيق تعدد الأشكال في `Elixir` الإرسال على بروتوكول متاح لأي نوع بيانات طالما أنه ينفذ البروتوكول. دعونا نرى مثلاً.

في Elixir، يتم التعامل مع القيمتين `nil` و `false` فقط على أنها `false`، قد يكون من المهم تحديد بروتوكول فارغ؟ بعيد قيمة منطقية `true`، يعتمد على شيء آخر على أنه `true`. وبالمثل، يمكن اعتبار القائمة الفارغة أو الثنائي الفارغ فارغين.

يمكننا تعريف هذا البروتوكول على النحو التالي:

```
def blank?(data) do
  # يرجع صحيحاً إذا تم اعتبار البيانات فارغة/فارغة
  # في defprotocol
```

نهاية

يتوقع البروتوكول وظيفة تسمى `blank?` والتي تتلقى وسيطة واحدة ليتم تنفيذها. يمكننا تنفيذ هذا البروتوكول لأنواع بيانات Elixir المختلفة على النحو التالي:

```
# الأعداد الصحيحة لا تكون فارغة أبداً
defimpl Blank, for: Integer do
  def blank?(_), do: false end
```

مجرد قائمة فارغة هي فارغة # بالنسبة إلى: `Blank`

```
# مجرد خريطة فارغة هي فارغة # بالنسبة إلى: Blank
defimpl Blank, for: Map do
  def blank?(_), do: true end
```

مجرد خريطة فارغة هي فارغة # بالنسبة إلى: `Blank`

```
# ضع في اعتبارك أننا لا نستطيع مطابقة النمط على {} لأنه # يتتطابق مع جميع الخرائط. ومع ذلك، يمكننا التتحقق مما إذا كان الحجم # يساوي صفرًا (والحجم عملية سريعة).
def blank?(map), do: map_size(map) == 0
```

نهاية

```
# فقط الذرات false و nil فارغة. بالنسبة لـ :Atom do
```

```
(_) do
  def blank?(atom), do: atom == nil or atom == false
  end
```

أفعال: خطأ

نهاية

وسنفعل ذلك لجميع أنواع البيانات الأساسية. الأنواع المتاحة هي:

* الدرة

* سلسلة بت

* بطاقة

* وظيفة

* عدد صحيح

* قائمة

* خريطة

* معرف المنتج

* ميناء

* مرجع

* متراكبة بيانات

الآن وبعد أن تم تعريف البروتوكول وتنفيذاته في متناول اليد، يمكننا استدعاوه:

```
<فارغ فارغ؟(0)>  
هذا صحيح  
<فارغ فارغ؟([1, 2, 3])>
```

يؤدي تمرير نوع بيانات لا ينفذ البروتوكول إلى حدوث خطأ:

```
"hello" لم يتم تنفيذ البروتوكول الفارغ لـ ("مرجحا")
```

16.1 البروتوكولات والهيآكل

تكمن قوّة قابلية التوسّع في Elixir في استخدام البروتوكولات والهيآكل معاً.

في الفصل السابق، تعلمنا أنه على الرغم من أنّ الهيّاكل عبارة عن خرائط، إلا أنها لا تشتراك في تنفيذات البروتوكول مع الخرائط. دعنا نحدّد هيكل المستخدم كما في الفصل السابق:

```
do المستخدم@iex> defmodule  
 27 ...> end; العمر...> defstruct name: "john".  
  
<<70, 79, 82, ...>>, {__struct__, 0}; المستخدم@module,
```

ومن ثم تحقق:

```
(%User{})>  
هذا صحيح  
(%User{})>  
%User{age: 27, name: "john"} لم يتم تنفيذ البروتوكول الفارغ لـ ("بروتوكول غير معرف") (Protocol.UndefinedError)
```

بدلاً من مشاركة تنفيذ البروتوكول مع الخرائط، تتطلب الهيّاكل تنفيذ بروتوكول خاص بها:

```
do def blank?(_), do: false end defimpl
```

إذا رغبت في ذلك، يمكنك ابتكار دلالات خاصة بك للمستخدم الفارغ. ليس هذا فحسب، بل يمكنك أيضًا استخدام الهيّاكل لبناء أنواع بيانات أكثر قوّة، مثل قوائم الانتظار، وتنفيذ جميع البروتوكولات ذات الصلة، مثل `Blank` و `Enumerable`.

في كثير من الحالات، قد يرغب المطوروون في توفير تنفيذ افتراضي للهيّاكل، حيث إنّ تنفيذ البروتوكول صراحةً لجميع الهيّاكل قد يكون مرهقاً. وهنا يأتي دور `Any`.

16.2 العودة إلى أي

قد يكون من المناسب توفير تنفيذ افتراضي لجميع الأنواع، ويمكن تحقيق ذلك من خلال تعريف `@fallback_to_any` على `true` في تعريف البروتوكول:

```
defprotocol  
  صحيحة@fallback_to_any
```

فargent؟(بيانات) نهاية def

والتي يمكن تنفيذها الآن على النحو التالي:

defimpl فargent لـ أي شيء فargent (.afuel: حطأ ال نهاية

الآن سيتم اعتبار جميع أنواع البيانات (بما في ذلك الهيكل)، التي لم تنفذ لها بروتوكول Blank غير فارغة.

16.3 البروتوكولات المضمنة

يأتي Elixir مزودًا بعض البروتوكولات المضمنة. في الفصول السابقة، ناقشنا وحدة Enumerable التي توفر العديد من الوظائف التي تعمل مع أي بنية بيانات تنفذ بروتوكول:

```
iex> Enum.map [1, 2, 3], fn(x)-> x * 2 end [2,4,6] iex> Enum.reduce 1..3, 0, fn(x, acc)-> x + acc end 6
```

من الأمثلة المفيدة الأخرى بروتوكول String.Chars الذي يحدد كيفية تحويل بنية بيانات تحتوي على أحرف إلى سلسلة. ويمكن عرض ذلك عبر دالة to_string:

```
iex> to_string :hello  
"مرحبا"
```

لاحظ أن استيفاء السلسلة في Elixir يستدعي دالة to_string:

iex> 25"#{(25)"العمر: " العمر:

لا يعمل المقطع أعلاه إلا لأن الأرقام تنفذ بروتوكول String.Chars على سبيل المثال، سيؤدي تمرير مجموعة إلى حدوث خطأ:

"1, 2, 3" # {مجموعه} iex>

iex> {1, 2, 3} JString.Chars لم يتم تنفيذ بروتوكول (Protocol.UndefinedError)

عندما تكون هناك حاجة إلى "طباعة" بنية بيانات أكثر تعقيداً، يمكنك ببساطة استخدام وظيفة التفتيش، استناداً إلى بروتوكول التفتيش:

"tuple: {1, 2, 3}" "# {فحضر المجموعة} iex>

Elixir هو البروتوكول المستخدم لتحويل أي بنية بيانات إلى تمثيل نصي قابل للقراءة. هذا ما تستخدمنه أدوات مثل ExPrint الناتجة:

iex> {1, 2, 3} {1,2,3} iex> %User{}

27} %User{name: "john".

ضع في اعتبارك أنه وفقاً للاتفاقية، عندما تبدأ القيمة التي يتم فحصها بعلامة `#`، فإنها تمثل بنية بيانات في صيغة Elixir غير صالحة. وهذا يعني أن بروتوكول الفحص غير قابل للعكس حيث قد يتم فقد المعلومات على طول الطريق:

```
&(&1+2)>
5>"/rpxe.lave_lre: "#Function<6.71889879/1
```

هناك بروتوكولات أخرى في Elixir ولكن هذا يغطي البروتوكولات الأكثر شيوعاً.

17 فهماً

toc.html %} تشمل

في Elixir، من الشائع تكرار Enumerable مما يؤدي غالباً إلى تصفية بعض النتائج وتعيين القيم في قائمة أخرى. تعتبر الفهميات حلّ لغوناً لمثل هذه البيانات؛ فهي تجمع هذه المهام الشائعة في شكل خاص.

على سبيل المثال، يمكننا تعيين قائمة من الأعداد الصحيحة إلى قيمها الترتيبية:

```
n * n [1, 4, 9, 16] <- [1, 2, 3, 4] iex> بالنسبة إلى
```

يتكون الفهم من ثلاثة أجزاء: المولدات والمرشحات والمقتنيات.

17.1 المولدات والمرشحات

في التعبير أعلاه، `[1, 2, 3, 4] <- n` هو المولد. فهو يقوم حرفيًا بتوليد القيم التي سيتم استخدامها في الفهم. يمكن تمثيل أي عدد قابل للعد في الجانب الأيمن من تعبير المولد:

```
n * n [1, 4, 9, 16] <- 1..4 iex> بالنسبة إلى
```

تدعم عبارات المولد أيضًا مطابقة الأنماط على جانبيها الأيسر؛ ويتم تجاهل جميع الأنماط غير المطابقة. تخيل أنه بدلاً من النطاق، لدينا قائمة كلمات رئيسية حيث يكون المفتاح هو الدرة: `doog` أو `dab` ونريد فقط حساب مربع قيم `:doog`:

```
iex> values = [good: 1, good: 2, bad: 3, good: 4] iex> for {good, n} <- values, do: n * n [1, 4, 16]
```

بدلاً من مطابقة الأنماط، يمكن استخدام المرشحات لتصفية بعض العناصر المعينة. على سبيل المثال، يمكننا تصفية جميع مضاعفات الرقم 3 والحصول على مربع القيم المتبقية فقط:

```
n * n [0, 9] <- 0..5, multiple_of_3?(n) iex> بالنسبة إلى multiple_of_3? = fn(n) -> rem(n, 3) == 0 end iex>
```

تقوم Comprehensions بتصفية جميع العناصر التي يبعد تعبير التصفية لها قيمة `False` أو `:nil` ويتم الاحتفاظ بجميع القيم الأخرى. توفر عمليات الفهم عموماً تمثيلاً أكثر إيجازاً من استخدام الوظائف المكافئة من وحدات Enum وStream. علاوة على ذلك، تسمح عمليات الفهم أيضًا بإعطاء مولدات ومرشحات متعددة. فيما يلي مثال يستقبل قائمة من الدلائل وتحذف جميع الملفات الموجودة في تلك الدلائل:

```
for file in File.ls!(dir),
```

```
do Path.join(dir, file).File.regular?(path)
```

ملف.mr! (المسار) النهاية

ضع في اعتبارك أن تعبيبات المتغيرات داخل الفهم، سواء كانت في المولدات أو المرشحات أو داخل الكتلة، لا تتعكس خارج الفهم.

17.2 مولدات سلسلة البيانات

يتم أيضاً دعم مولدات سلسلة البيانات وهي مفيدة جدًا عندما تحتاج إلى فهم تدفقات سلسلة البيانات. يتلقى المثال أدناه قائمة بالبكسلات من ملف ثانٍ مع قيمها الحمراء والخضراء والزرقاء ويجولها إلى مجموعات من ثلاثة عناصر لكل منها:

```
{r, g, b} [{213, 45, 132}, {64, 76, 32}, {76, 0, 0}, {234, 32, 15}] --> iex> بكسلي<r::8, g::8, b::8 <- _ = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
```

يمكن دمج مولد سلسلة البيانات مع المولدات القابلة للعد "العادية" وتوفير المرشحات أيضًا.

17.3 نتائج غير القوائم

في الأمثلة أعلاه، قامت جميع عمليات الفهم بإرجاع القوائم كنتيجة لها. ومع ذلك، يمكن إدراج نتيجة الفهم في هيكل بيانات مختلفة عن طريق تمرير الخيار: otni إلى الفهم.

على سبيل المثال، يمكن استخدام مولد سلسلة البيانات مع خيار: otni من أجل إزالة جميع المسافات في السلسلة بسهولة:

```
<<إلا بالطبع، إذا؟!>>, إل: "أفعال":>>، "مرحبا بالعالم"
```

يمكن أيضًا من المجموعات والخرائط والقواميس الأخرى لخيار: otni. بشكل عام، يقبل: otni أي بنية تنفذ بروتوكول Collectable.

يمكن أن تكون إحدى حالات الاستخدام الشائعة intو:- لـ هي تحويل القيم في الخريطة، دون لمس المفاتيح:

```
{key, val * val} %{"a" => 1, "b" => 4} --> iex> إلى: {key, val} <- %. {"a" => 1, "b" => 2}, iex>
```

نقم بعمل مثال آخر باستخدام التدفقات. نظرًا لأن وحدة الإدخال والإخراج توفر تدفقات (تنقسم بالقابلية للعد والقابلية للتجميع)، يمكن تنفيذ محطة صدى تردد النسخة المعرفة من أي شيء مكتوب باستخدام الفهم:

```
iex> stream = IO.stream(:stdio, :line) iex> for line <- stream, into: stream do ...> String.upcase(line) <> "\n" ...> end
```

الآن أكتب أي سلسلة في المحطة الطرفية وستجد أن نفس القيمة سُتطيع بأحرف كبيرة. لسوء الحظ، تسبب هذا المثال أيضًا في تعطل واجهة Elixir، لذا ستحتاج إلى الضغط على Ctrl+C مرتين للخروج منها. (:

18 رمزاً

toc.html %} تشمل

لقد تعلمنا بالفعل أن Elixir يوفر سلاسل نصية محاطة بعلامتي اقتباس مزدوجتين وقوائم أحرف محاطة بعلامتي اقتباس مفردين. ومع ذلك، فإن هذا لا يغطي سوى سطح الهيكل التي لها تمثيل نصي في اللغة. على سبيل المثال، يتم إنشاء الـ `mota` في الغالب عبر التمثيل:

أحد أهداف Elixir هو إمكانية التوسيع: يجب أن يكون المطوروون قادرين على توسيع اللغة لتناسب أي مجال معين. لقد أصبح علم الكمبيوتر مجالاً واسعاً لدرجة أنه من المستحبيل أن تتناول لغة ما العديد من المجالات كجزء من جوهيرها. أفضل رهان لدينا هو جعل اللغة قابلة للتوسيع، حتى يمكن المطوروون والشركات والمجتمعات من توسيع اللغة إلى المجالات ذات الصلة.

في هذا الفصل، سنستكشف الرموز، وهي إحدى الآليات التي توفرها اللغة للعمل مع التمثيلات النصية. تبدأ الرموز بعلامة التلدة (~) التي يتبعها حرف (يحدد الرمز) ثم فاصل؛ ويمكن إضافة تعديلات اختيارية بعد الفاصل النهائي.

18.1 التعبيرات العادية

الرمز الأكثر شيوعاً في Elixir هو ~، والذي يستخدم لإنشاء التعبيرات العادية:

```
#تعبير عادي ينطوي على السلاسل التي تحوي على "foo" أو "bar"
"bar": iex> regex = ~r/foo|bar/ -r/foo|bar/ iex> "foo" =~ regex true
```

`iex> "bat" =~`

خطأ شنيع

يوفر Elixir تعبيرات منتظمة متوافقة مع PCRE (Perl regular expressions)، كما تم تنفيذها بواسطة PCRE. على سبيل المثال، يجعل المحدد `regex` التعبير العادي غير حساسة لحالة الأحرف:

```
/مرحبا/~ "مرحبا" =~ مرحبا
خطأ شنيع
=~ -r/hello/i "مرحبا" =~
حقيقة
```

قم بمراجعة وحدة `Regex` للحصول على مزيد من المعلومات حول المعدلات الأخرى والعمليات المدعومة باستخدام التعبيرات العادية.

حتى الآن، استخدمت جميع الأمثلة / لتحديد تعبير عادي. ومع ذلك، تدعم الرموز 8 فواصل مختلفة:

```
-ر/مرحبا/
-ر|مرحبا|
-ر"مرحبا"
-مرحبا'
<(مرحبا)->(مرحبا)
```

السبب وراء دعم الفواصل المختلفة هو أن الفواصل المختلفة قد تكون أكثر ملاءمة لرموز مختلفة. على سبيل المثال، قد يكون استخدام الأقواس للتعبيرات العادية خياراً مربكاً حيث يمكن خلطها بالأقواس داخل التعبيرات العادية. ومع ذلك، يمكن أن تكون الأقواس مفيدة لرموز أخرى، كما سنرى في القسم التالي.

18.2 السلاسل وقوائم الأحرف ورموز الكلمات

بالإضافة إلى التعبيرات العادية، يأتي Elixir مع ثلاثة رموز أخرى.

السلسل 18.2.1

يُستخدم الرمز `s`-لإنشاء سلاسل، مثل علامات الاقتباس المزدوجة. يكون الرمز `s`-مفيداً، على سبيل المثال، عندما تحتوي السلسلة على علامات اقتباس مزدوجة ومفردة:

```
iex> s("هذه سلسلة تحتوي على علامتي اقتباس "مزدوج", وليس "مفرد") "هذه سلسلة تحتوي على علامتي اقتباس "مزدوج", وليس "مفرد"
```

قوائم الأحرف 18.2.2

يتم استخدام `sigil c`-لإنشاء قوائم الأحرف:

```
iex> c("هذه قائمة أحرف تحتوي على 'علامات اقتباس مفردة' " هذه قائمة أحرف تحتوي على 'علامات اقتباس مفردة'"
```

قوائم الكلمات 18.2.3

يُستخدم رمز `w`-لإنشاء قوائم من الكلمات (الكلمات عبارة عن سلاسل عادية فقط). داخل رمز `w`- يتم فصل الكلمات بمسافات بيضاء.

```
iex> ~w(foo bar bat)
["فoo", "bar", "bat"]
```

قبل الرمز `w`-أيضا التعديلات `c` و `w` (قوائم `char` والسلسل والذرارات على التوالي) والتي تحدد نوع بيانات عناصر القائمة الناتجة:

```
iex> ~w(foo barbat)a [:foo, :bar, :bat]
```

الاستيفاء والهروب في الرموز 18.3

بالإضافة إلى الرموز الصغيرة، يدعم Elixir الرموز الكبيرة للتعامل مع الأحرف الفارغة والتدخل. وبينما سيعيد كل من `s`-و`W` سلاسل، يسمح الأول برموز الهروب والتدخل بينما لا يسمح الثاني بذلك:

```
\x26 #{'inter' <> "polation"}~iex>
```

```
"سلسلة تحتوي على أ��اد الهروب"~iex>
```

```
"سلسلة بدون أڪاد الهروب والاستيفاء"~iex>
```

```
"سلسلة بدون أڪاد الهروب وبدون {"# interpolation}
```

يمكن استخدام أ��اد الهروب التالية في السلاسل وقوائم الأحرف:

- * علامتي اقتباس مزدوجتين

- * علامة اقتباس مفردة

- * شرطية مائلة عكسية واحدة

- \a - جرس/تنبيه

- \b - مسافة خلفية

- \d - حذف

- \e - الهروب

- ١٦ • نموذج التغذية
- ١٧ • سطر جديد
- ١٨ - إرجاع العربية
- ١٩ • مسافة
- t _ علامة النبوب
- ٢٠ • علامة تبويب عمودية
- ٢١ • باب فارغ
- ٢٢ • حرف مع تمثيل سداسي عشر (على سبيل المثال، \x13)
- ٢٣ • حرف مع تمثيل سداسي عشر برقم سداسي عشر واحد أو أكثر (على سبيل المثال، \x{D...} \x{abc13})

تدعم `Sigil` أيضًا علامات الاقتباس المزدوجة أو المفردة الثلاثية كفاصلات:

```
ایکس<~S"""
    ...
    ...
...
<...سلسلة مستندة إلى هيرودوك
```

أكمل حالات الاستخدام شيوغا لرموز `heredoc` عند كتابة الوثائق، على سبيل المثال، كتابة أحرف الإفلات في الوثائق ستتحول عرضة للخطأ قريباً بسبب الحاجة إلى الإفلات المزدوج بعض الأحرف:

```
@doc """
تحويل علامات الاقتباس المزدوجة إلى علامات اقتباس مفردة.

# أمثلة

("\\\\"foo\\\\""\")ex>
    "فو"
...
تحويل(...)

def(...)
```

من الممكن تجنب هذه المشكلة تماماً باستخدام `S-`:

```
@doc ~S"""
تحويل علامات الاقتباس المزدوجة إلى علامات اقتباس مفردة.

# أمثلة

("\"foo\""\")ex>
    "فو"
...
تحويل(...)
```

18.4 الرموز المخصصة

كما تم التلميح في بداية هذا الفصل، فإن الرموز في `Elixir` قابلة للتتوسيع. في الواقع، فإن استخدام الرمز `i/r` يعادل استدعاء دالة `sigil_r` مع ثانوي وقائمة أحرف كحجة:

```
iex> sigil_r(<<"foo">>, 'i') ~r"foo"
```

يمكننا الوصول إلى الوثائق الخاصة بـ `sigil_r` عبر الدالة:

```
iex> h sigil_r
```

يمكننا أيضًا توفير رموزنا الخاصة من خلال تفزيذ وظائف تبع نمط `{identifier}`. دعنا ننفذ رمز `~` الذي يعيد عدداً صحيحاً (مع التعديل اختياري `n` لجعله سالباً):

```
iex> defmodule MySigils do ...> def sigil_i(string, []) do: String.to_integer(string) ...> def sigil_i(string, [?n]) do: -String.to_integer(string) ...> end
```

MySigils iex> ~i(13) 13 استيراد

```
iex> ~i(42)n -42
```

يمكن أيضاً استخدام الرموز لتنفيذ العمل في وقت التجميع بمساعدة وحدات الماكرو. على سبيل المثال، يتم تجميع التعبيرات العادي في Elixir في تمثيل الماكرو، بينما يتم تخطي هذه الخطوة في وقت التشغيل. إذا كنت مهتماً بالموضوع، فنوصيك بمعرفة المزيد حول وحدات الماكرو والاطلاع على كيفية تنفيذ الرموز في وحدة Kernel حيث يتم تعريف وظائف `sigil_*`.

19 محاولة، الإمساك والإنقاذ

toc.html %} تشمل

يحتوي برنامج Elixir على ثلاث آليات للخطأ: الأخطاء والرميات والخروجات. في هذا الفصل سوف نستكشف كل منها ونضيف ملاحظات حول متى يجب استخدام كل منها.

19.1 الأخطاء

تُستخدم الأخطاء (أو الاستثناءات) عند حدوث أشياء استثنائية في الكود. يمكن استرداد خطأ نموذجي عن طريق محاولة إضافة رقم إلى ذرة:

```
iex> :foo + 1
** حجة خاطئة في التعبير الحسابي
:erlang.+(:foo, 1)
```

يمكن إثارة خطأ وقت التشغيل في أي وقت باستخدام `raise/1`:

```
iex> raise "أويس"
** خطأ وقت التشغيل (عفواً
```

يمكن إثارة أخطاء أخرى باستخدام `raise/2` عن طريق تمرير اسم الخطأ وقائمة من وسيطات الكلمات الأساسية:

```
iex> raise ArgumentError, "حجة غير صالحة"
** حجة غير صالحة (ArgumentError)
```

يمكنك أيضاً تعريف أخطائك الخاصة عن طريق إنشاء وحدة واستخدام بنية `defexception` بها: بهذه الطريقة، ستتشكل خطأ بنفس اسم الوحدة التي تم تعريفه فيها. الحالة الأكثر شيوعاً هي تعريف استثناء مخصص بحقل رسالة:

```
iex> defmodule MyError do iex> defexception message: "default message" iex> end
```

```
MyError رفع iex>
"رسالة الإقتصاد: الرسالة"** (MyError) raise MyError.
```

رسالة مخصصة** (MyError)

يمكن إنقاذ الأخطاء باستخدام بنية المحاولة/إنقاذ:

```
iex> iex> try...catch...end
      ...> catch RuntimeError e
      ...> e.message
      ...> end
      ...> e.message
      ...> "رسالة: عفواً" (MyError) rorrEemitnuR%
```

ينفذ المثال أعلاه خطأ وقت التشغيل وبعد الخطأ نفسه والذي يتم طباعته بعد ذلك في جلسة. ولكن في الممارسة العملية، نادرًا ما يستخدم مطورو Elixir/rescue على سبيل المثال، تجبرك العديد من اللغات على إنقاذ خطأ عندما لا يمكن فتح ملف بنجاح. يوفر Elixir بدلاً من ذلك دالة `File.read/1` التي تعيد مجموعة تحتوي على معلومات حول ما إذا كان الملف قد تم فتحه بنجاح:

```
iex> iex> File.read("hello", "world")
      ...> {:error, :enoent}
      ...> :ok
      ...> :ok.message
      ...> "Hello, world!"
```

لا توجد محاولة/إنقاذ هنا. في حالة رغبتك في التعامل مع نتائج متعددة لفتح ملف، يمكنك ببساطة استخدام مطابقة الأنماط مع بنية الحال:

```
iex> iex> IO.puts("النهاية الملف، قراءة " مرحبا")
      ...> {:error, reason} ...> IO.puts("خطأ: "##{body})
      ...> reason.message
```

في نهاية المطاف، يعود الأمر إلى تطبيقك لتحديد ما إذا كان الخطأ الذي حدث أثناء فتح ملف ما استثنائياً أم لا. ولهذا السبب لا يفرض Elixir استثناءات على `File.read/1` والعديد من الوظائف الأخرى. بدلاً من ذلك، يترك الأمر للمطور لاختيار أفضل طريقة للمتابعة.

بالنسبة للحالات التي تتوقع فيها وجود ملف (وغياب هذا الملف بعد خطأ حقيقي)، يمكنك ببساطة استخدام `File.read!/1`:

```
iex> iex> File.read!("file.ex:305")
      ...> {:error, reason} ...> IO.puts("خطأ: "##{body})
      ...> reason.message
```

تبعد العديد من الوظائف في المكتبة القياسية نمط وجود نظير يغير استثناء بدلاً من إرجاعمجموعات لمطابقتها. تمثل الاتفاقية في إنشاء دالة `(foo!(error, reason))` أو `(foo(ok, result))` ودالة أخرى `(foo(result))` بنفس الاسم ولكن مع إضافة `!` تأخذ نفس الوسائط مثل `foo` ولكنها تغير استثناء إذا كان هناك خطأ. يجب أن تغير `!foo` النتيجة (غير ملفوقة في مجموعة) إذا سارت الأمور على ما يرام، وحدة الملف `__FILE__` هي مثال على ذلك.

في Elixir، تتجنب استخدام `try/rescue` لأننا لا نستخدم الأخطاء للتحكم في التدفق. تأخذ الأخطاء حرفياً: فهي مخصصة للمواقف غير المتوقعة وأو الاستثنائية. في حالة احتياجك لعمليات إنشاءات للتحكم في التدفق، فيجب استخدام `throws`.

رميات 19.2

في Elixir، يمكن رمي قيمة ثم التقاطها لاحقاً. يتم حجز `try` و `catch` اللهم المواقف التي لا يمكن فيها استرداد القيمة إلا باستخدام `throw` و `catch`.

هذه المواقف نادرة جدًا في الممارسة العملية باستثناء عند التعامل مع المكتبات التي لا توفر واجهة برمجة تطبيقات مناسبة. على سبيل المثال، لتخيل أن وحدة `Enum` لم توفر أي واجهة برمجة تطبيقات للعثور على قيمة وأننا بحاجة إلى العثور على أول مضاعف للرقم 13 في قائمة من الأرقام:

```
iex> try(x, 13) == 0, fn(x) -> iex> catchEnum.each -50..50, fn(x) -> ...>
...>
<...النهاية...
<...النهاية...
...لم أحصل على شيء... فنص<...
-> "حصلت على "#{x}"...
-> ...>
...النهاية...
-> ...
-> "حصلت على "-39"
```

نظرًا لأن `Enum` يوفر واجهة برمجة تطبيقات مناسبة، فمن الناحية العملية فإن `Enum.find/2` هو الحل الأمثل:

```
iex> Enum.find -50..50, &(rem(&1, 13) == 0) -39
```

المخارج 19.3

يتم تشغيل جميع أ Kovاد Elixir داخل العمليات التي تتواصل مع بعضها البعض. عندما تموت عملية ما بسبب "أسباب طبيعية" (على سبيل المثال، استثناءات غير معالجة)، فإنها ترسل إشارة خروج. يمكن أن تموت العملية أيضًا عن طريق إرسال إشارة خروج صراحةً:

```
iex> spawn_link fn -> exit(1) end #PID<0.56.0>
#PID<0.56.0>***(الخروج من 1)
```

في المثال أعلاه، ماتت العملية المرتبطة عن طريق إرسال إشارة خروج بقيمة 1. تتولى واجهة `Elixir shell` معالجة هذه الرسائل تلقائيًا وطبعتها على المحطة الطرفية.

يمكن أيضًا "القبض" على الخروج باستخدام `try/catch`:

```
iex> iex> catchException...> ...> ...> ...>
...> ...> ...> ...>
...> ...> ...> ...>
...> ...> ...> ...>
...> ...> ...> ...>
...> ...> ...> ...>
...> ...> ...> ...>
...> ...> ...> ...>
```

إن استخدام `try/catch` أصبح غير شائع بالفعل، واستخدامه لالتقاط المخارج أصبح أكثر ندرة.

تشكل إشارات الخروج جزءاً مهمًا من نظام مقاومة الأخطاء الذي توفره آلة Erlang الأفتراضية. عادةً ما يتم تشغيل العمليات تحت أشجار الإشراف التي هي نفسها عمليات تتنتظر فقط إشارات الخروج من العمليات الخاضعة للإشراف.

بمجرد تلقي إشارة الخروج، يتم تفعيل استراتيجية الإشراف وإعادة تشغيل العملية الخاضعة للإشراف.

إن نظام الإشراف هذا هو بالتحديد ما يجعل إنشاءات مثل `try/catch` و `try/rescue` غير شائعة في Elixir. فيEDAً من إنقاذ خطأ، نفضل "الفشل بسرعة" لأن شجرة الإشراف ستتضمن عودة تطبيقنا إلى حالة أولية معروفة بعد الخطأ.

بعد 19.4

في بعض الأحيان يكون من الضروري التأكد من تنظيف المورد بعد بعض الإجراءات التي قد تؤدي إلى حدوث خطأ. يتيح لك إنشاء `try/after` للقيام بذلك. على سبيل المثال، يمكننا فتح ملف وضمان إغلاقه (حتى لو حدث خطأ ما) باستخدام كتلة `try/after`:

```
"sample", [:utf8, :write] iex> try do ...> IO.write file, "olá" ...> raise "oops, something went wrong" ...> after
iex> {:ok, file} = File.open
```

>...ملف.إغلاق(ملف)
 >...النهاية
 **عفواً، حدث خطأ ما (RuntimeError)

19.5 نطاق المتغيرات

من المهم أن تضع في اعتبارك أن المتغيرات المحددة داخل كتل `try/catch/rescue/after` لا تسرب إلى السياق الخارجي. وذلك لأن كتلة `try` قد تفشل وبالتالي قد لا يتم ربط المتغيرات في المقام الأول. معنى آخر، هذا الرمز غير صالح:

جاول القيام بـ ...> from_try = true ...> iex>

```
...> from_after = true ...> end
```

من المحاولة iex>
 دالة غير محددة : from_try/0 (RuntimeError)
 دالة غير محددة : from_after/0 (RuntimeError)

هذا يعني مقدمتنا عن المحاولة والصياغة والإنقاذ. ستجد أن هذه العبارات تُستخدم بشكل أقل في Elixir مقارنة باللغات الأخرى، على الرغم من أنها قد تكون مفيدة في بعض المواقف حيث لا تعمل المكتبة أو بعض التعليمات البرمجية المحددة "وفقاً للقواعد".

20 مواصفات النوع والسلوكيات

toc.html %} تشمل

20.1 أنواع المواصفات

Elixir هي لغة ذات أنواع ديناميكية، لذا فإن جميع الأنواع في Elixir يتم استنتاجها من خلال وقت التشغيل. ومع ذلك، تأتي Elixir مع مواصفات النوع، وهي عبارة عن تدوين يستخدم لـ:

1. إعلان أنواع البيانات المخصصة:
2. إعلان توقيعات الوظيفة المكتوبة (المواصفات).

20.1.1 مواصفات الوظيفة

يشكل افتراضي، يوفر Elixir بعض الأنواع الأساسية، مثل `integer` أو `pid`. بالإضافة إلى أنواع أكثر تعقيداً: على سبيل المثال، دالة `round/1` التي تقرب عددًا عشربيًا إلى أقرب عدد صحيح، تأخذ رقمًا كحجja (عدد صحيح أو عدد عشري) وتعيد عددًا صحيحًا. كما ترى في وثائقها، تتم كتابة التوقيع المطبوع الخاص بـ `round/1` على النحو التالي:

```
round(number) ::
```

يعني أن الدالة الموجودة على الجانب الأيسر تعيد قيمة يكون نوعها هو الموجود على الجانب الأيمن. تتم كتابة مواصفات الدالة باستخدام التوجيه `@spec` الذي يتم وضعه قبل تعريف الدالة مباشرةً. يمكن كتابة دالة `round/1` كالتالي:

```
round(number) :: integer def round(number), do: #
```

Elixir يدعم أيضًا أنواع المركبة. على سبيل المثال، تحتوي قائمة الأعداد الصحيحة على النوع `integer`. يمكنك الاطلاع على جميع أنواع التي يوفرها Elixir في مستندات `typespecs`.

20.1.2 تحديد أنواع المخصصة

على الرغم من أن Elixir يوفر العديد من أنواع المضمنة المفيدة، إلا أنه من الملائم تعريف أنواع مخصصة عند الحاجة. ويمكن القيام بذلك عند تعريف وحدات `modules` من خلال `@type`.

لفترض أن لدينا وحدة `LousyCalculator` والتي تقوم بالعمليات الحسابية المعتادة (المجموع، المنتج وما إلى ذلك)، ولكن بدلاً من إرجاع الأرقام، فإنها تقوم بإرجاعمجموعات مع نتيجة عملية كعنصر أول وجزء عشوائية كعنصر ثان.

```
defmodule LousyCalculator do
  @spec add(number, number) :: {number, String.t} def add(x, y), do: {x + y, "هل تحتاج إلى آلة حاسبة للقيام بذلك؟!"}
end
```

```
end do: {x * y, "يا إلهي، تعال!"}
```

كما ترى في المثال، فإن الثنائيات هي نوع مركب ويتم تحديد كل ثنائي من خلال أنواع الموجودة بداخله. لفهم سبب عدم كتابة `String.t` كسلسلة، ألق نظرة أخرى على مستندات `typespecs`.

إن تعريف مواصفات الوظيفة بهذه الطريقة يعمل بشكل جيد، ولكن سرعان ما يصبح الأمر مزعجاً لأننا نكرر النوع `{number, String.t}` مرتين وتكراراً. يمكننا استخدام دليل `@type` لإعلان النوع المخصص الخاص بنا.

```
@typedoc """"@defmodule LousyCalculator
```

مجرد رقم متبع بسلسلة.

.....

```
@type number_with_offense :: {number, String.t}
```

```
  @spec add(number, number) :: number_with_offense def add(x, y), do: {x + y,
```

```
  end do: {x * y, "يا إلهي، تعال!"}
```

يتم استخدام تعليمة `@typedoc` على غرار تعليمتي `@moduledoc` و `@doc` لتوثيق أنواع المخصصة.

يتم تصدير أنواع المخصصة المحددة من خلال `@type` و تكون متاحة خارج الوحدة النمطية التي تم تعريفها فيها:

`PoliteCalculator` إلغاء وحدة

```
make_polite(LousyCalculator.add(x, y))
```

```
  @spec make_polite(LousyCalculator.number_with_offense) ::
```

```
defp make_polite({num, _offense}), do: num end
```

إذا كنت تريد الاحفاظ بنوع مخصص خاصا، فيمكنك استخدام تعليمة `@typepep` بدلاً من `@type`.

تحليل الكود الثابت 20.1.3

لا تعد مواصفات النوع مفيدة للمطوري فقط، بل إنها بمثابة وثائق إضافية. أداة Erlang Dialyzer على سبيل المثال، يستخدم مواصفات النوع لإجراء تحليل ثابت للكود. ولهذا السبب، في مثال، كتبنا مواصفات لوظيفة `make_polite/1` حتى لو تم تعريفها كوظيفة خاصة.

السلوكيات 20.2

تشارك العديد من الوحدات النمطية في نفسواجهة برمجة التطبيقات العامة. ألق نظرة على Plug وهو، كما يشير وصفه، عبارة عن مواصفات لوحدات قابلة للتكون في تطبيقات الويب. كل مكون إضافي عبارة عن وحدة يجب أن تنفذ وظيفتين عامتين على الأقل: `init/2` و `call/1`:

توفر السلوكيات وسيلة لـ:

* تحديد مجموعة من الوظائف التي يتبعن تنفيذها بواسطة وحدة نمطية؛

* التأكد من أن الوحدة تنفذ جميع الوظائف الموجودة في تلك المجموعة.

إذا كان عليك ذلك، يمكنك التفكير في سلوكيات مثل الواجهات في اللغات الموجهة للكائنات مثل Java: مجموعة من توقيعات الوظائف التي يتبعن على الوحدة النمطية تنفيذها.

تحديد السلوكيات 20.2.1

لفترض أننا نريد تنفيذ مجموعة من المحللات، كل منها يحل البيانات المنظمة: على سبيل المثال، محلل JSON ومحلل YAML. سيتصرف كل من هذين المحللين بنفس الطريقة: سيوفر لكاهما دالة `parse/0` و دالة `parse/1` تمثل `parse/0` للبيانات المنظمة، بينما ستعيد دالة `extensions/0` قائمة بامتدادات الملفات التي يمكن استخدامها لكل نوع من البيانات (على سبيل المثال، `.json` لملفات JSON).

يمكننا إنشاء سلوك المحلل:

```
defmodule محلل
  use Application
  alias MyApp.Slack
```

```
  defcallback parse(String.t) :: String.t
  defcallback extensions() :: [String.t]
```

سيتعين على الوحدات النمطية التي تتبعن سلوك المحلل تنفيذ جميع الوظائف المحددة باستخدام `defcallback`. وكما ترى، فإن `defcallback` يتوقع اسم وظيفة ولكن أيضاً مواصفات وظيفة مثل تلك المستخدمة مع التوجيه `@spec`.

تبني السلوكيات 20.2.2

إن تبني السلوك أمر بسيط:

```
defmodule JSONParser
@تحليل السلوك
def extensions, do: ["json"] end
def parse(str), do: # ...
نهاية
```

```
defmodule YAMLParser
@تحليل السلوك
YAML def extensions, do: ["yml"] end
def parse(str), do: # ...
نهاية
```

إذا لم تقم الوحدة النمطية التي تتبع سلوكاً معيناً بتنفيذ إحدى عمليات الاسترجاع المطلوبة بواسطة هذا السلوك، فسيتم إنشاء تحذير وقت التجميع.

Elixir - أدلة بناء لـ Mix

1 مقدمة عن المزيج

toc.html %} تشمل

في هذا الدليل، سنتعلم كيفية إنشاء تطبيق Elixir كامل، مع شجرة الإشراف الخاصة به، والتكون، والاختبارات والمزيد.

يعمل التطبيق كمخزن موزع للقيم الأساسية. سنقوم بتنظيم أزواج القيم الأساسية فيمجموعات وتوزيع هذه المجموعات على عدة عقد. سنقوم أيضاً ببناء عميل بسيط يسمح لنا بالاتصال بأي من هذه العقد وإرسال طلبات مثل:

إنشاء التسوق	نعم
ضع حليب التسوق 1	نعم
ضع بيض التسوق 3	نعم
احصل على الحليب للتسوق	نعم
حذف بيض التسوق	نعم

من أجل بناء تطبيق القيمة الرئيسية الخاص بنا، سنستخدم ثلاثة أدوات رئيسية:

*OTP (منصة الاتصالات المفتوحة) عبارة عن مجموعة من المكتبات التي تأتي مع Erlang. يستخدم مطورو OTP لبناء تطبيقات قوية ومقاومة للأخطاء. في هذا الفصل، سوف نستكشف عدد الجوانب من OTP التي تتكامل مع Elixir، بما في ذلك أشجار الإشراف ومديري الأحداث والمزيد؛

*Mix هي أداة بناء تأتي مع Elixir وتتوفر مهام لإنشاء تطبيقك وتجميده واختباره، إدارة التبعيات وأكثر من ذلك بكثير؛

ExUnit هو إطار عمل يعتمد على وحدة اختبار يتم شحنه مع Elixir

الإصدارات توثيق Elixir

في هذا الفصل، سوف نقوم بإنشاء مشروعنا الأول باستخدام Mix واستكشاف الميزات المختلفة في Mix وOTP وExUnit.

ملاحظة: يتطلب هذا الدليل استخدام Elixir v0.15.0 أو إصدارات أحدث. يمكنك التحقق من إصدار Elixir لديك باستخدام `v`-وتبثبيت إصدار أحدث إذا لزم الأمر باتباع الخطوات الموضحة في الفصل الأول من دليل البدء.

إذا كان لديك أي أسئلة أو تحسينات على الدليل، يرجى إعلامنا بذلك في قائمة البريد الخاصة بنا أو متنبيع القضايا على التوالي. إن مساهمتك مهمة حقاً لمساعدةنا في ضمان إمكانية الوصول إلى الأدلة وتحديثها!

1.1 مشروعنا الأول

عند تثبيت Elixir، بالإضافة إلى الحصول على الملفات القابلة للتنفيذ، `iex` و `elixirc` و `elixir` يسمى `mix`.

لقد بإنشاء مشروعنا الأول من خلال استدعاء `new` من سطر الأوامر، سنمرر اسم المشروع كحجة `kv` (في هذه الحالة)، ونخبر `mix` أن وحدتنا الرئيسية يجب أن تكون `KV` بأحرف كبيرة بالكامل، بدلاً من الوحدة الافتراضية، والتي كانت ستكون `Kv`:

```
kv --module KV $مزبح جديد
```

سيقوم `mix` بإنشاء دليل يسمى `KV` يحتوي على بعض الملفات بداخله:

```
mix.exs * إنشاء README.md * إنشاء erongitig.
lib/kv.ex * إنشاء config.exs * إنشاء lib/config/config.exs * إنشاء
test/kv_test.exs * إنشاء test/test_helper.exs * إنشاء test/test.exs
```

دعونا نلقي نظرة سريعة على تلك الملفات التي تم إنشاؤها.

ملاحظة: `Mix` هو ملف قابل للتنفيذ من Elixir. وهذا يعني أنه لتشغيل `mix`، يجب أن يكون لديك ملف `elixir` القابل للتنفيذ في `PATH`. إذا لم يكن الأمر كذلك، فيمكنك تشغيله عن طريق تمرير البرنامج النصي كحجة إلى `elixir`:

```
kv --module KV $bin/elixirbin/mix
```

لاحظ أنه يمكنك أيضًا تنفيذ أي نص برمجي في `PATH` الخاص بك من Elixir: `5`:

```
kv --module KV $bin/elixir -S
```

عند استخدام `-S`، يبحث `elixir` عن البرنامج النصي أينما كان في `PATH` ويقوم بتنفيذه.

1.2 تجميع المشروع

تم إنشاء ملف باسم `mix.exs` داخل مجلد المشروع الجديد (`kv`) ومسؤوليته الرئيسية هي تكوين مشروعنا. دعونا نلقي نظرة عليه (تم إزالة التعليقات):

```
defmodule KV.Mixfile
  Mix.Project
    def do
      مشروع
```

```
"0.0.1", deps: deps] [التطبيق: .vk: الإصدار: النهائي
```

```
[logger][ التطبيقات: def application do
نهائية
```

```
defp deps do []
نهائية
```

```
نهائية
```

يحدد ملف mix.exs الخاص بنا دالتين عامتين: project الذي يقوم بإرجاع تكوين المشروع مثل اسم المشروع والإصدار، application الذي يستخدم لإنشاء ملف التطبيق.

هناك أيضًا دالة خاصة تسمى deps، والتي يتم استدعاؤها من دالة المشروع، والتي تحدد تبعيات المشروع. ليس من الضروري تعريف deps كدالة منفصلة، ولكنها تساعد في الحفاظ على تكوين المشروع منظماً.

يقوم Mix أيضًا بإنشاء ملف في lib/kv.ex يحتوي على تعريف وحدة بسيط:

```
KV إلغاء الوحدة النمطية
نهائية
```

هذا الهيكل يكفي لتجميع مشروعنا:

```
$ cd kv $
```

سيتم الإخراج:

```
kv.app تم تجميع اوتم إنشاء
```

لاحظ أنه تم تجميع ملف lib/kv.ex وإنشاء ملف kv.app. وقد حدث كل هذا في بنية دليل خاصة به، داخل مجلد build. يتم إنشاء ملف ppa. هذا باستخدام المعلومات من وظيفة application/0 التي في ملف mix.exs. وستكتشف ميزات تكوين mix.exs بشكل أكبر في الفصول المستقبلية.

مجرد تجميع المشروع، يمكنك بدء جلسة iex داخل المشروع عن طريق تشغيل:

```
$ iex -s
```

1.3 تشغيل الاختبارات

كما قام Mix أيضًا بإنشاء البنية المناسبة لتشغيل اختبارات المشروع. تتبع مشروع Mix عادةً اتفاقية وجود ملف _test.exs في دليل الاختبار لكل ملف في دليل lib. ولهذا السبب، يمكننا بالفعل العثور على المقابل لملف test/kv_test.exs.

```
defmodule KVTest
استخدم ExUnit.Case
```

```
اختبار "الحقيقة"
1 + 1 == 2
نهائية
نهائية
```

ومن المهم ملاحظة بضعة أشياء:

1. ملف الاختبار عبارة عن ملف نصي (.exs). وهذا مناسب لأننا لا نحتاج إلى تجميع ملفات الاختبار قبل تشغيلهم؛

2. نقوم بتعريف وحدة اختبار تسمى `ExUnit.Case` (`</docs/stable/ex_unit/ExUnit.Case.html>`) ونستخدم `KVTest` لحقن واجهة برمجة التطبيقات للاختبار وتحديد اختبار بسيط باستخدام الماكرو `test/2`:

قام Mix أيضًا بإنشاء ملف يسمى `test/test_helper.exs` وهو المسئول عن إعداد إطار عمل الاختبار:

الوحدة السابقة، البدء

سيطلب Mix هذا الملف تلقائيًا في كل مرة قبل تشغيل الاختبارات. يمكننا تشغيل الاختبارات باستخدام:

kv.app تم تجميع lib/kv.ex

تم الانتهاء في 0.04 ثانية (0.00 ثانية عند التحميل، 0.00 ثانية عند الاختبارات) اختبار، 0 فشل

عنوان مع الذور 540224

لاحظ أنه من خلال تشغيل اختبار المزيج، قام Mix بتجميع ملفات المصدر وإنشاء ملف التطبيق مرة أخرى. يحدث هذا لأن Mix يدعم بيئات متعددة، وهو ما سينتكرشه في القسم التالي.

علاوة على ذلك، يمكنك أن ترى أن `ExUnit` يطبع نقطة لكل اختبار ناجح ويقوم تلقائيًا بجعل الاختبارات عشوائية أيضًا. دعونا نجعل الاختبار يفشل عمدًا ونرى ماذا سيحدث.

قم بتغيير التأكيد في `test/kv_test.exs` إلى ما يلي:

1 + 1 == 3 أكد

الآن قم بتشغيل اختبار المزيج مرة أخرى (لاحظ هذه المرة أنه لن يكون هناك تجميع):

(KVTest) 1) اختبار الحقيرة
== فشل التأكيد باستخدام test/kv_test.exs:4

اللوك: 1 + 1 == 3
الجانب الأيسر: 2
اليمين: 3
تتبع المكدس:
اختبار kv_test.exs:5

تم الانتهاء في 0.05 ثانية (0.05 ثانية عند التحميل، 0.00 ثانية عند الاختبارات) اختبار، 1 فشل

بالنسبة لكل فشل، تقوم `ExUnit` بطباعة تقرير مفصل، يحتوي على اسم الاختبار مع حالة الاختبار، والرمز الذي فشل وقيم الجانب الأيسر (`lhs`) والجانب الأيمن (`rhs`) من عامل `==`.

في السطر الثاني من الفشل، أسلف اسم الاختبار مباشرةً، يوجد الموقع الذي تم فيه تعريف الاختبار، إذا قمت بنسخ موقع الاختبار في هذا السطر الثاني بالكامل (بما في ذلك الملف ورقم السطر) وإضافته إلى اختبار `test/kv_test.exs`، فسيقوم Mix بتحميل وتشغيل هذا الاختبار المحدد فقط:

مزيج اختبار test/kv_test.exs:4

سيكون هذا الاختصار مفيدًا للغاية أثناء بناء مشروعنا، مما يسمح لنا بالذكراك بسرعة من خلال تشغيل اختبار محدد فقط.

أخيرًا، يرتبط تتبع المكدس بالفشل نفسه، مما يوفر معلومات حول الاختبار وغالبًا المكان الذي تم فيه إنشاء الفشل من داخل ملفات المصدر.

1.4 البيانات

يعد مفهوم "البيانات". فهي تسمح للمطور بتخصيص التجميع والخيارات الأخرى لسيناريوهات محددة. بشكل افتراضي، يفهم Mix ثلاث بيئات:

- هو الذي يتم فيه تشغيل مهام المزج (مثل التجميع) افتراضياً :dev -

- يستخدمه mix test -

- الذي ستستخدمه لوضع مشروعك في الإنتاج :prod -

ملاحظة: إذا قمت بإضافة تبعيات إلى مشروعك، فلن ترث بيئتك مشروعك، بل س يتم تشغيلها بدلاً من ذلك بإعدادات بيئه dorp: الخاصة بها!

بشكل افتراضي، تتصرف هذه البيانات بنفس الطريقة وستؤثر جميع التكوينات التي رأيناها حتى الآن على البيانات الثلاث. يمكن إجراء التخصيص لكل بيئه من خلال الوصول إلى دالة `_mix.exs` في ملف `Mix.env` </docs/stable/mix/Mix.html#env/1>.

`[deps_path: deps_path (Mix.env)]`

نهاية

"deps" قم بما يلي: `defp prod_deps" defp deps_path(_).` قم بما يلي: `defp deps_path(:prod).`

سيتم تعين Mix افتراضياً على بيئه :ved، باستثناء مهمة الاختبار التي سيتم تعينها افتراضياً على بيئه :tset. يمكن تغيير البيئة عبر متغير البيئة: `MIX_ENV`

`VNE_XIM=تجميع مزدوج الإنتاج`

1.5 الاستكشاف

هناك الكثير مما يتعلق بـ Mix، وسنستمر في استكشافه أثناء بناء مشروعنا. توفر نظرة عامة على وثائق Mix.

طبع في اعتبارك أنه يمكنك دائمًا استدعاء مهمة المساعدة لإدراج جميع المهام المتاحة:

`$ مزدوج المساعدة`

يمكنك الحصول على مزيد من المعلومات حول مهمة معينة عن طريق استدعاء `mix help TASK`.

دعونا نكتب بعض التعليمات البرمجية!

2 وكيل

`toc.html %} تشمل {%`

في هذا الفصل، سنقوم بإنشاء وحدة تسمى KV.Bucket. ستكون هذه الوحدة مسؤولة عن تخزين إدخالات القيمة الرئيسية بطريقة تسمح بالقراءة والتعديل بواسطة عمليات مختلفة.

إذا تحطيت دليل البدء أو قرأته منذ فترة طويلة، فتأكد من إعادة قراءة الفصل الخاص بالعمليات. سنتستخدمه كنقطة بداية.

2.1 مشكلة الدولة

Elixir هي لغة ثابتة لا يتم فيها مشاركة أي شيء بشكل افتراضي. إذا أردنا إنشاء دلاء وتخزينها والوصول إليها من أماكن متعددة، فلدينا خياران رئيسيان في:

• العمليات

• تخزين مصطلحات إرلانج ETS

لقد تحدثنا عن العمليات، في حين أن ETS شيء جديد سنستكشفه لاحقاً في هذا الدليل. ولكن عندما يتعلق الأمر بالعمليات، نادراً ما نقوم بتنفيذ عملية خاصة بنا، بل نستخدم بدلاً من ذلك التجريدات المتاحة في PTO و Elixir.

• العميل - غلافات بسيطة حول الحالة.

- GenServer "خوادم عامة" (عمليات) تغلف الحالة وتتوفر مكالمات المزامنة وغير المترافقه وتدعم الكود إعادة التحميل، وأكثر من ذلك.

- GenEvent "مدبر الأحداث العامة" الذين يسمحون بنشر الأحداث إلى معالجات متعددة.

• المهمة - وحدات حسابية غير متزامنة تسمح بإنشاء عملية واسترجاع نتيجتها بسهولة في وقت لاحق.

سنستكشف كل هذه التجريدات في هذا الدليل. ضع في اعتبارك أنها جميئاً يتم تنفيذها أعلى العمليات باستخدام الميزات الأساسية التي توفرها الآلة الافتراضية، مثل الإرسال والاستقبال والتكتائ والربط.

2.2 الوكلاء

الوكاء عبارة عن غلافات بسيطة حول الحالة. إذا كان كل ما تريده من عملية هو الاحتفاظ بالحالة، فإن الوكلاء مناسبون تماماً. دعنا نبدأ جلسة iex #مزدوجة داخل المشروع بـ:

```
iex $ مزدوج
```

والعب قليلاً مع العملاء:

```
iex> {ok, agent} = Agent.start_link fn -> [] end {ok, #PID<0.57.0>} iex> Agent.update(agent, fn list -> ("eggs" | list) end) :ok
```

```
iex> Agent.stop(agent) [~] البيط[iex> Agent.get(agent, fn list -> list end)
```

نعم:

لقد بدأنا عميلاً بحالة أولية لقائمة فارغة. بعد ذلك، أصدرنا أمراً لتحديث الحالة، بالإضافة للعنصر الجديد إلى رأس القائمة. أخيراً، استعدنا القائمة بالكامل. بمجرد الانتهاء من العميل، يمكننا استدعاء Agent.stop/1 لإنهاء عملية العميل.

لنقم بتنفيذ KV.Bucket باستخدام الوكاء. ولكن قبل البدء في التنفيذ، دعنا أولاً نكتب بعض الاختبارات. قم بإنشاء ملف في test/kv/bucket_test.exs (تذكر امتداد .sxe) بما يلي:

```
defmodule KV.BucketTest
  use ExUnit.Case, async: true

  # اختبار " تخزين القيم حسب المفتاح"
  test "نهاية" do
    assert KV.Bucket.start_link assert KV.Bucket.get(bucket, "milk") == nil
  end
end
```

```
KV.Bucket.put(bucket, "milk", 3) assert KV.Bucket.get(bucket, "milk") == 3
```

نهاية

اختبارنا الأول بسيط ومبادر، نبدأ دلو KV جديداً ونقوم ببعض عمليات get و put على عليه، ونؤكّد النتيجة. لا يحتاج إلى إيقاف العميل صراحةً لأنّه مرتبط بعملية الاختبار ويتم إيقاف تشغيل العميل تلقائياً بمجرد انتهاء الاختبار.

لاحظ أيضًا أنها مررنا خيار ExUnit.Case: true يجعل هذا الخيار حالة الاختبار هذه تعمل بالتوالي مع حالات اختبار أخرى تقوم بإعداد خيار cnysa. هذا مفيد للغاية لتسريع مجموعة الاختبار الخاصة بنا باستخدام نوى متعددة في جهازنا. لاحظ أنه يجب تعين خيار cnysa: فقط إذا كانت حالة الاختبار لا تعتمد على أي قيمة عالمية أو تغيرها. على سبيل المثال، إذا كان الاختبار يتطلب الكتابة إلى نظام الملفات أو تسجيل العمليات أو الوصول إلى قاعدة بيانات، فيجب ألا يجعله غير متزامن لتجنب طروف السباق بين الاختبارات.

بغض النظر عن كونه غير متزامن أم لا، فمن الواضح أن اختبارنا الجديد يجب أن يفشل، حيث لم يتم تنفيذ أي من الوظائف.

لصلاح الاختبار الفاشل، دعنا ننشئ ملفًا في lib/kv/bucket.ex بالمحفوظات أدناه. لا تتردد في تجربة تنفيذ وحدة KV.Bucket بنفسك باستخدام الوكاء قبل إلقاء نظرة على التنفيذ أدناه.

```
KV.Bucket إلقاء وحدة
@doc """
بدأ دلوًا جديداً.

def start_link do
  Agent.start_link(fn -> HashDict.new end)

  @doc """
  يحصل على قيمة من `bucket` بواسطة `key`.
  """

  def get(bucket, key) do Agent.get(bucket, &HashDict.get(&1, key)) end

  @doc """
  يضع "القيمة" لـ "المفتاح" المحدد في "الدلو".
  """

  def update(bucket, &HashDict.put(&1, key, value)) do
    Agent.update(bucket, &HashDict.put(&1, key, value))
  end

نهاية
نهاية
```

لاحظ أننا نستخدم HashDict للتخزين حالتنا بدلاً من Map، لأنه في الإصدار الحالي من Elixir تكون الخرائط أقل كفاءة عند الاحتفاظ بعدد كبير من المفاتيح.

الآن بعد أن تم تعريف وحدة KV.Bucket، يجب أن ينجح اختبارنا يمكنك تجربته بنفسك عن طريق تشغيل الأمر: mix test.

2.3 استدعاءات ExUnit

قبل الانتقال إلى إضافة المزيد من الميزات إلى KV.Bucket، دعنا نتحدث عن عمليات الإرجاع الخاصة به. وكما قد تتوقع، ستتطلب جميع اختبارات KV.Bucket بدء تشغيل ExUnit. كما دلوا أنباء الإعداد وإيقافه بعد الاختبار، لحسن الحظ، يدعم ExUnit عمليات الإرجاع التي تسمح لنا بتخطي مثل هذه المهام المتكررة.

دعنا نعيد كتابة حالة الاختبار لاستخدام عمليات الاسترجاع:

```
defmodule KV.BucketTest
  use ExUnit.Case, async: true

  {ok, bucket} = KV.Bucket.start_link []
  # إعداد قم بـ

  KV.Bucket.get(bucket, "milk") == nil %{
    "تحقق من حسب المفتاح"
  }

  # انتهى
```

```
KV.Bucket.put(bucket, "milk", 3) assert KV.Bucket.get(bucket, "milk") == 3
```

نهاية

لقد قمنا أولاً بتعريف معاودة اتصال الإعداد بمساعدة الماكرو `setup/1`. يتم تشغيل معاودة اتصال الإعداد / 1 قبل كل اختبار، بنفس عملية الاختبار نفسه.

لاحظ أننا نحتاج إلى آلية لتمرير معرف دلو العملية من معاودة الاتصال إلى الاختبار، نقوم بذلك باستخدام سياق الاختبار، عندما نعيد `{ok, bucket}` من معاودة الاتصال، سيدمج `ExUnit` العنصر الثاني من المجموعة (القاموس) في سياق الاختبار، سياق الاختبار عبارة عن خريطة يمكننا مطابقتها في تعريف الاختبار، مما يوفر الوصول إلى هذه القيم داخل الكتلة:

```
اختبار "تخزين القيم حسب المفتاح".
%{bucket: bucket}
# هو الان الدلو من نهاية كتلة الإعداد
```

يمكنك قراءة المزيد حول حالات `ExUnit.Callbacks.html` في [وثائق وحدة `ExUnit.Case`](#) والمزيد حول عمليات الاسترجاع في [وثائق `_`](#).
`ExUnit.Callbacks </docs/stable/ex_unit/`

2.4 إجراءات الوكيل الأخرى

بالإضافة إلى الحصول على قيمة وتحديث حالة الوكيل، تسمح لنا الوكالة بالحصول على قيمة وتحديث حالة الوكيل في استدعاء وظيفة واحدة عبر `Agent.get_and_update/2`. دعنا ننفذ وظيفة `delete` التي تحذف مفتاحاً من الدلو، وتغير قيمته الحالية:

```
@doc """
يُحذف 'key' من 'bucket'.
إرجاع القيمة الحالية لـ 'key' إذا كان 'key' موجوداً.

"""
def delete(bucket, key)
  Agent.get_and_update(bucket, &HashDict.pop(&1, key))
```

الآن حان دورك لكتابة اختبار للوظيفة المذكورة أعلاه! تأكد أيضاً من استكشاف الوثائق الخاصة بالوكالء لمعرفة المزيد عنهم.

2.5 العميل/الخادم في الوكالة

قبل أن ننتقل إلى الفصل التالي، دعنا نناقش ثنائية العميل/الخادم في الوكالة. دعنا نوسع وظيفة `2 / delete` التي قمنا بتنفيذها للتتو:

```
def delete(bucket, key)
  Agent.get_and_update(bucket, fn dict-> HashDict.pop(dict, key) end)
```

كل ما هو داخل الدالة التي مررناها إلى العميل يحدث في عملية العميل. في هذه الحالة، بما أن عملية العميل هي الخادم، كل شيء خارج الدالة يحدث في العميل.

هذا التمييز مهم، إذا كان هناك إجراءات مكلفة بحسب القيام بها، فيجب عليك أن تفكير فيما إذا كان من الأفضل تنفيذ هذه الإجراءات على العميل أم على الخادم، على سبيل المثال:

```
Agent.get_and_update(bucket, fn dict -> وضع العميل في وضع السكون def delete(bucket, key) do :timer.sleep(1000) #
```

```
HashDict.pop(dict, key) end) وضع الخادم في وضع السكون :timer.sleep(1000) #
```

عندما يتم تنفيذ إجراء طويل على الخادم، فإن جميع الطلبات الأخرى لهذا الخادم المحدد ستنتظر حتى يتم تنفيذ الإجراء، مما قد يتسبب في انتهاء مهلة بعض العملاء.

في الفصل التالي سوف نستكشف GenServers، حيث يصبح الفصل بين العملاء والخادم أكثر وضوحاً.

خادم 3أجيال

toc.html %} تشمل

في الفصل السابق، استخدمنا وكلاء لتمثيل دلائنا. في الفصل الأول، حددنا أننا نرغب في تسمية كل دلو حتى نتمكن من القيام بما يلي:

إنشاء التسوق
نعم

ضع حليب التسوق
نعم

احصل على الحليب للتسوق

1

نعم

نظرًا لأن الوكلاء عبارة عن عمليات، فإن كل دلو له معرف عملية (pid) ولكنه لا يحمل اسمًا. لقد تعلمنا عن سجل الأسماء في فصل العملية وقد تمثل إلى حل هذه المشكلة باستخدام مثل هذا السجل.

على سبيل المثال، يمكننا إنشاء دلو على النحو التالي:

```
iex> Agent.start_link(fn -> [] end, name: :shopping) {:ok, #PID<0.43.0>} iex> KV.Bucket.put(:shopping, "milk", 1) :ok
```

(":تسوق، "الحليب") teg.tekcuB.VKiex>

1

ومع ذلك، فهذه فكرة سيئة للغاية! يجب أن تكون الأسماء المحلية في Elixir عبارة عن ذرات، مما يعني أننا سنحتاج إلى تحويل اسم الدلو (غالبًا ما يتم استلامه من عميل خارجي) إلى ذرات، ويجب ألا نحول إدخال المستخدم إلى ذرات أبدًا. وذلك لأن الذرات لا يتم جمعها من القماممة. بمجرد إنشاء ذرة، لا يتم استردادها أبدًا. إن إنشاء ذرات من إدخال المستخدم يعني أنه يمكن للمستخدم حقن أسماء مختلفة كافية لاستنفاد ذاكرة نظامنا من الناحية العملية، من المرجح أن تصل إلى حد Erlang VM لأقصى عدد من الذرات قبل نفاد الذاكرة، مما سيؤدي إلى تعطل نظامك على أي حال.

بدلاً من إساءة استخدام مرفق تسجيل الأسماء، سنقوم بدلاً من ذلك بإنشاء عملية تسجيل خاصة بنا تحتوي على قاموس يربط اسم الدلو بعملية الدلو.

يجب أن يضمن السجل تحديث القاموس دائمًا. على سبيل المثال، إذا تعطلت إحدى عمليات المجموعة بسبب خطأ، فيجب على السجل تنظيف القاموس لتجنب تقديم إدخالات قديمة. في Elixir، نصف هذا بالقول إن السجل يحتاج إلى مراقبة كل مجموعة.

GenServer يستخدم لإنشاء عملية تسجيل يمكنها مراقبة عملية الدلو. GenServers هي التجزيد المفضل لبناء خوادم عامة في كل من PTO و Elixir.

أول GenServer لدينا 3.1

يتم تنفيذ GenServer في جزء: واجهة برمجة تطبيقات العميل وعمليات إستدعاء الخادم، وكلها في وحدة واحدة. قم بإنشاء ملف جديد في `lib/kv/registry.ex` على المحتويات التالية:

```
defmodule KV.Registry do
  #واجهة برمجة تطبيقات العميل
  @doc """
  يبدأ التسجيل.
  """
  def start_link(opts \\ []) do
    GenServer.start_link(__MODULE__, :ok, opts)
    #نهاية
  end

  @doc """
  يبحث عن معرف الدلو لـ `name` المخزن في `server`.
  """
  def lookup(server, {:lookup, name}) do
    GenServer.call(server, {:lookup, name})
    #نهاية
  end

  @doc """
  يتتأكد من وجود دلو مرتبط بـ `name` المحدد في "الخادم".
  """
  def create(server, name) do
    #عمليات إستدعاء الخادم
    defp handle_call({:lookup, name}, _from, names) do
      reply = HashDict.fetch(names, name)
      HashDict.put(names, name, reply)
      {:noreply, names}
    end

    defp handle_cast({:create, name}, names) do
      reply = KV.Bucket.start_link(:norelease, HashDict.put(names, name, reply))
      {:noreply, names}
    end
    #نهاية
  end
end
```

الوظيفة الأولى هي `start_link/1`، والتي تبدأ GenServer جديداً عن طريق تمرير ثلاثة وسيطات:

1. الوحدة التي يتم فيها تنفيذ عمليات استدعاء الخادم، في هذه الحالة، `MODULE`، مما يعني التعديل الحالي أولى

2. حجج التهيئة، في هذه الحالة الذاكرة: `ko`.

3. قائمة بالخيارات التي يمكنها، على سبيل المثال، الاحتفاظ باسم الخادم.

هناك نوعان من الطلبات التي يمكنك إرسالها إلى `GenServer`: الاستدعاءات والإرسالات. تكون الاستدعاءات متزامنة ويجب على الخادم إرسال استجابة لهذه الطلبات. وتكون الإرسالات غير متزامنة ولن يرسل الخادم استجابة.

الدالثان `handle_call/2` و `handle_cast/2` و `lookup/2`، وهي مسؤولة عن إرسال هذه الطلبات إلى الخادم. يتم تمثيل الطلبات بواسطة الوسيطة الأولى لـ `handle_call/3` أو `handle_cast/3`. في هذه الحالة، استخدمنا `{create, name}` أو `{lookup, name}` على التوالي. غالباً ما يتم تحديد الطلبات على هيئةمجموعات، مثل هذه، لتوفير أكثر من "وسطية" واحدة في خانة الوسيطة الأولى. من الشائع تحديد الإجراء المطلوب باعتباره العنصر الأول في المجموعة، والوسطيات الخاصة بهذا الإجراء في العناصر المتبقية.

على جانب الخادم، يمكننا تنفيذ مجموعة متنوعة من عمليات الاسترجاع لضمان تهيئة الخادم وإنهائه ومعالجه للطلبات. عمليات الاسترجاع هذه اختيارية، وفي الوقت الحالي قمنا بتنفيذ العمليات التي نهتم بها فقط.

الأول هو استدعاء `GenServer.start_link/3`، حيث تكون `state` عن `HashDict` الجديدة. يمكننا بالفعل ملاحظة كيف يجعل واجهة برمجة تطبيقات `GenServer` الفصل بين العميل والخادم أكثر وضوحاً. يحدث `start_link/3` في العميل، بينما `init/1` هو استدعاء المقابلة الذي يتم تشغيله على الخادم.

بالنسبة لطلبات الاتصال، يجب علينا تنفيذ معاودة الاتصال `handle_call/3` التي تتلقى الطلب والعملية التي تلقينا منها الطلب (`from`) وحالة الخادم الحالية (الأسماء). تعيد معاودة الاتصال مجموعة بتنسيق `{:reply, reply, new_state}`، حيث يكون الرد هو ما سيتم إرساله إلى العميل `new_state` وهو حالة الخادم الجديدة.

بالنسبة لطلبات الإرسال، يجب علينا تنفيذ معاودة الاتصال `handle_cast/2` التي تتلقى الطلب وحالة الخادم الحالية (الأسماء). تعيد معاودة الاتصال `handle_cast/2` مجموعة بتنسيق `{:noreply, new_state}`.

توجد تنسيدات أخرى للعناصر التي قد تعيدها كل من عمليات الاستدعاء `handle_call/3` و `handle_cast/2`، وكما توجد عمليات استدعاء أخرى مثل `code_change/2` و `terminate/3`. يمكننا تنفيذهما، نرحب بك لاستكشاف وتألق `GenServer` الكاملة لمعرفة المزيد عنها.

في الوقت الحالي، دعونا نكتب بعض الاختبارات لضمان عمل `GenServer` الخاص بنا كما هو متوقع.

3.2 اختبار GenServer

لا يختلف اختبار `GenServer` كثيراً عن اختبار وكيل. سننشئ الخادم على استدعاء إعداد ونستخدمه طوال اختباراتنا. أنشئ ملفاً في `test/kv/registry_test.exs` بما يلي:

```
defmodule KV.RegistryTest
  use ExUnit.Case, async: true
```

```
{:ok, registry} = KV.Registry.start_link {:ok, registry: registry} end
```

اختبار "دلة التفريخ". قم بذلك
`KV.Registry.lookup(registry, "shopping") == :error`

```
KV.Registry.create(registry, "shopping") assert {:ok, bucket} = KV.Registry.lookup(registry, "shopping")
```

```
KV.Bucket.put(bucket, "milk", 1) assert KV.Bucket.get(bucket, "milk") == 1
```

نهاية
نهاية

ينبغي أن ينجح اختبارنا فور إخراجه من الصندوق!

لإيقاف تشغيل السجل، نقوم ببساطة بإرسال إشارة nwodtuhs: إلى عمليته عند انتهاء الاختبار. وفي حين أن هذا الحل مناسب لاختبارات، فإذا كانت هناك حاجة لإيقاف كجزء من منطق التطبيق، فمن الأفضل تعريف دالة stop/1 التي ترسل رسالة استدعاء تتسبب في إيقاف الخادم GenServer:

```
#واجهة برمجة تطبيقات العميل
@doc """
يوقف التسجيل.

"""
def stop(server)
  GenServer.call(server, :stop) #في إنتهائه
#عمليات استدعاء الخادم
  def handle_call(:stop, _from, state) do {:stop, :normal, :ok, state} end
```

في المثال أعلاه، تقوم عبارة handle_call/3 بإرجاع الذاكرة pots: إلى جانب سبب توقف الخادم، (:normal) والرد ko: وحالة الخادم.

3.3 الحاجة إلى المراقبة

لقد اكتمل سجلنا تقريرًا. المشكلة الوحيدة المتبقية هي أن السجل قد يصبح قديمًا إذا توقف أحد الدلو أو تعطل. دعنا نضيف اختبارًا إلى KV.RegistryTest يكشف عن هذا الخطأ:

```
"shopping"){:ok, bucket} = KV.Registry.lookup(registry, "shopping") #قم بـ
اختبار إزالة الدلو عند الخروج
KV.Registry.create(registry,
Agent.stop(bucket) assert KV.Registry.lookup(registry, "shopping") == :error end
```

سيفشل الاختبار أعلاه في التأكيد الأخير حيث يظل اسم الدلو في السجل حتى بعد إيقاف عملية الدلو.

لإصلاح هذا الخطأ، نحتاج إلى أن يراقب السجل كل دلو يتم إنشاؤه. بمجرد إعداد المراقبة، سيتلقى السجل إشعارًا في كل مرة يتم فيها خروج دلو، مما يسمح لنا بتنظيف القاموس.

دعونا أولًا نلعب بالشاشات عن طريق بدء تشغيل وحدة تحكم جديدة باستخدام iex -S mix:

```
pid} = KV.Bucket.start_link {:ok, #PID<0.66.0>} iex> Process.monitor(pid)
                                         iex> {:ok,
                                         <155.0.0.0> المرجع#
iex> Agent.stop(pid)
                                         نعم:
                                         iex> flush()
                                         مرجع<0.66.0> عمليه، #معرف العملية<:> عادي{:DOWN,
```

ملحوظة: تعيد Process.monitor(pid) مرجعاً فريداً يسمح لنا بتطابقة الرسائل القادمة مع مرجع العميل، يمكننا مسح جميع الرسائل وملحظة وصول رسالة NWOD، مع المرجع الدقيق الذي تم إرجاعه بواسطة monitor، لإعلامنا بخروج عملية الدلو مع السبب .lamron

دعنا نعيد تفزيذ عمليات استدعاء الخادم لصلاح الخطأ واحتياز الاختبار. أولاً، سنعدل حالة GenServer إلى قاموسين: أحدهما يحتوي على الاسم <-معرف العملية والآخر يحتوي على المرجع <-الاسم، بعد ذلك، نحتاج إلى مراقبة الدلاء على handle_info/2 للتعامل مع رسائل المراقبة. يظهر أدناه تفزيذ عمليات استدعاء الخادم بالكامل:

```
# مطالبات استرجاع الخادم

def init(:ok) do
  names = HashDict.new
  = HashDict.new(المراجع)
  {ok, {names, refs}}
end

def handle_call({:lookup, name}, _from, {names, _} = state) do
  {reply, HashDict.fetch(names, name), state}
end

def handle_call({:stop, _from, state}) do
  {stop, :normal, :ok, state}
end

def handle_cast({:create, name}, {names, refs}) do
  ref = HashDict.put(names, name, pid)
  {noreply, {names, refs}}
end

def handle_cast({:has_key?, name}, {names, refs}) do
  if HashDict.has_key?(names, name) do
    {noreply, {names, refs}}
  else
    {reply, false, {names, refs}}
  end
end

def handle_info({:DOWN, ref, :process, _pid, _reason}, {names, refs}) do
  {names, refs} = HashDict.pop(names, ref)
  HashDict.put(names, ref.pid, ref)
  {noreply, {names, refs}}
end

def handle_info(_msg, state) do
  {noreply, state}
end
```

لاحظ أدناه من تفزيذ الخادم بشكل كبير دون تغيير أي من واجهات برمجة التطبيقات الخاصة بالعميل. هذه إحدى فوائد الفصل الصريح بين الخادم والعميل.

أخيراً، وعلى عكس عمليات الاسترجاع الأخرى، فلدينا بتعريف شرط "catch-all" لـ handle_info/2 والذي يتغافل أي رسالة غير معروفة. لفهم السبب، دعنا ننتقل إلى القسم التالي.

13.4 الاتصال أو البيث أو المعلومات؟

حتى الآن، استخدمنا ثلاثة عمليات رد اتصال: handle_call/3 وhandle_cast/2 وhandle_info/2. إن تحديد متى نستخدم كل واحدة منها أمر بسيط:

1. يجب استخدام handle_call/3 للطلبات المتزامنة. يجب أن يكون هذا هو الخيار الافتراضي لأن انتظار رد الخادم بعد آلية ضغط عكسي مفيدة.

2. يجب استخدام handle_cast/2 للطلبات غير المتزامنة، عندما لا تهتم بالرد. لا يضمن التحويل حتى أن الخادم قد تلقى الرسالة، ولهذا السبب، يجب استخدامه بحذر، على سبيل المثال، كان من المفترض أن تستخدم دالة create/2 التي حددها في هذا الفصل دالة call/2 casts لـ *لأغراض تعليمية*.

3. يجب استخدام handle_info/2 لجميع الرسائل الأخرى التي قد يستقبلها الخادم والتي لا يتم إرسالها عبر GenServer.cast/2 أو GenServer.call/2، بما في ذلك الرسائل العادية المرسلة باستخدام send/2. يُتعذر رسائل المراقبة: NWOD مثلاً متألياً على ذلك.

نقطاً لأن أي رسالة، بما في ذلك التي يتم إرسالها عبر send/2، ستنتقل إلى الخادم. لذلك، إذا لم نحدد شرط "اللتقط الشامل". فقد تؤدي هذه الرسائل إلى تعطل المشرف لدينا، لأنه لن يتم مطابقة أي شرط.

لا داعي للقلق بشأن هذا بالنسبة ل handle_info/2 و handle_cast/2 لأن هذه الطلبات تتم فقط عبر واجهة برمجة تطبيقات GenServer، لذا من المحتمل جدًا أن تكون الرسالة غير المعروفة ناتجة عن خطأ من جانب المطور.

شاشات أو روابط؟

لقد تعلمنا سابقاً عن الروابط في فصل العملية. الآن، بعد اكتمال التسجيل، قد تتساءل: متى يجب علينا استخدام المراقبين ومتى يجب علينا استخدام الروابط؟

الروابط ثنائية الاتجاه. إذا قمت بربط عمليتين وتعطلت إحداهما، فسوف تتعطل الأخرى أيضًا (ما لم تكن تحتجز عمليات الخروج). تكون المراقبة أحادية الاتجاه: ستتلقى عملية المراقبة فقط إشعارات حول العملية التي تم مراقبتها. ببساطة، استخدم الروابط عندما تريد حدوث أخطاء مرتبطة، واستخدم المراقبة عندما تريد فقط معرفة الأخطاء وعمليات الخروج وما إلى ذلك.

بالعودة إلى تنفيذ handle_cast/2 الخاص بنا، يمكنك أن ترى أن السجل يربط الدلاء ويراقبها:

```
{:ok, pid} = KV.Bucket.start_link() ref = Process.monitor(pid)
```

هذه فكرة سيئة، لأننا لا نريد أن يتعطل السجل عند تعطل أحد الدلوان سنتكشف الحلول لهذه المشكلة عندما نتحدث عن المشرفين. باختصار، نتجنب عادةً إنشاء عمليات جديدة بشكل مباشر، بدلاً من ذلك، نفوض هذه المسئولية إلى المشرفين. كما سترى، يعمل المشرفون مع الروابط، وهذا يفسر سبب انتشار واجهات برمجة التطبيقات القائمة على الروابط spawn_link و start_link وما إلى ذلك) في Elixir و PTO.

قبل القفز إلى المشرفين، دعونا أولاً نستكشف مدير الأحداث ومعالجات الأحداث باستخدام GenEvent.

4 أحداث الجيل

toc.html %} تشمل

في هذا الفصل، سوف نستكشف GenEvent، وهو سلوك آخر توفره Elixir و PTO، والذي يسمح لنا بإنشاء مدير أحداث قادر على نشر الأحداث إلى العديد من المعالجات.

هناك حدثان سنقوم بإصدارهما: الأول في كل مرة تتم فيها إضافة دلو إلى السجل والآخر عند إزالته منه.

4.1 مدير الأحداث

لنبدأ جلسة مزيج S-ex الجديدة ونستكشف واجهة برمجة التطبيقات GenEvent قليلاً:

```
ko: (olleh: (مدبر،) yfiton_cnys.tnevEneG= GenEvent.start_link {:ok, #PID<0.83.0>} iex> {مدبر,} iex> {:ok,
```

نعم:
نعم

يبدأ GenEvent.start_link/0 مدبر أحداث جديداً. هذا هو كل ما هو مطلوب لبدء تشغيل مدبر. بعد إنشاء المدبر، يمكننا استدعاء GenEvent.notify/2 و GenEvent.sync_notify/2 لإرسال عدم الإشعارات.

ومع ذلك، نظراً لعدم وجود معالجات أحداث مرتبطة بالمدبر، فلا يحدث الكثير في كل إشعار.

دعنا ننشئ معالجنا الأول، لا يزال موجوداً على Ex، والذي يرسل جميع الأحداث إلى عملية معينة:

```
iex> defmodule GenEvent do
  use GenEvent, handle_event: &handle_event/2
  def handle_event(event, parent)
    ...>
    ...>
    > الـ...>
    الـ...>
    self() |> reldnah_dda.tnevEneGix>
    > نعم
    {:hello, :world} |> yfiton_cnys.tnevEneGix>
  end
  > نعم
  > اندفق
  > {مرحبا، العالم}: حسنا
```

لقد قمنا بإنشاء المعالج الخاص بنا وأضفناه إلى المدبر عن طريق استدعاء GenEvent.add_handler/3 وتمرير:

المدبر الذي بدأه سابقاً وربطناه.

وحدة معالجة الأحداث (المسماة Forwarder) التي قمنا بتعریفها للتوجيه.

3. حالة معالج الحدث: في هذه الحالة، معرف العملية الحالية

بعد إضافة المعالج، يمكننا أن نرى أنه من خلال استدعاء Forwarder بإعادة توجيهه الأحداث إلى صندوق الوارد الخاص بنا بنجاح.

هناك بعض الأشياء المهمة التي يجب تسلیط الضوء عليها في هذه المرحلة:

1. يتم تشغيل معالج الحدث في نفس العملية مثل مدبر الحدث.

2. يقوم sync_notify/2 بتشغيل معالجات الأحداث بشكل متزامن مع الطلب.

3. يقوم sync_notify/2 بتشغيل معالجات الأحداث بشكل غير متزامن.

لذلك، فإن sync_notify/2 و notify/2 يشيران إلى callcast/2 في GenServer. ويوصى عموماً باستخدام sync_notify/2، وهو يعمل كآلية ضغط عكسية في عملية الاستدعاء، لتقليل احتمالية إرسال الرسائل بسرعة أكبر من سرعة إرسالها إلى المعالجات.

تأكد من التحقق من الوظائف الأخرى التي يوفرها GenEvent في وثائق الوحدة النموذجية الخاصة به. في الوقت الحالي، لدينا المعرفة الكافية لإضافة مدبر الأحداث إلى تطبيقنا.

4.2. التسجيل للأحداث

من أجل إصدار الأحداث، نحتاج إلى تغيير السجل للعمل مع مدبر الأحداث. وبينما يمكننا تشغيل مدبر الأحداث تلقائياً عند بدء تشغيل السجل، على سبيل المثال في معاودة الاتصال init/1، الأفضل تمرير معرف/اسم مدبر الأحداث إلى start_link، مما يؤدي إلى فصل بدء مدبر الأحداث عن السجل.

لنبدأ أولاً بتغيير اختباراتنا لإظهار السلوك الذي نريد أن يظهره السجل. افتح `test/kv/registry_test.exs` وقم بتغيير استدعاء الإعداد الحالي إلى استدعاء الإعداد أدناه، ثم أضف الاختبار الجديد:

```
defmodule KV.Registry
  use GenEvent
  alias Forwarder

  def handle_event(event, parent)
  do
    case event do
      {:ok, parent} = ok ->
        Forwarder.start_link(ok, parent)
        :ok
      _other ->
        :stop
    end
  end

  @doc """
  اخبار! إرسال الأحداث عند الإنشاء والتعطّل.
  """
  def start_link(manager) do
    KV.Registry.create(manager, "shopping")
    Forwarder.start_link(manager, self())
    KV.Registry.start_link(manager)
  end

  def add_mon_handler(manager, forwarder, self()) do
    Forwarder.add_mon_handler(manager, forwarder)
  end

  def stop(bucket) do
    Forwarder.stop(bucket)
  end
end
```

لاختبار الوظيفة التي نريد إضافتها، نقوم أولاً بتعريف معالج حدث `Forwarder` مشابهاً للمعالج الذي كتبناه في `Ex` سابقاً. عند الإعداد، نبدأ تشغيل مدير الأحداث، ونمرره كحجة إلى السجل ونضيف معالج `Forwarder` الخاص بنا إلى المدير حتى يمكن إرسال الأحداث إلى عملية الاختبار.

في الاختبار، نقوم بإنشاء عملية دلو وإيقافها ونستخدم `assert_receive` على مهلة زمنية افتراضية تبلغ 500 مللي ثانية، وهو ما ينبغي أن يكون أكثر من كافية لاختباراتنا. لاحظ أيضاً أن `assert_receive` يتوقع نمطاً، وليس قيمة، ولهذا السبب استخدمنا `^bucket` للمطابقة مع معرف دلو.

أخيراً، لاحظ أننا استدعاينا `GenEvent.add_handler/3` بدلاً من `GenEvent.add_mon_handler/3`. تضيف هذه الوظيفة معالجاً، كما نعلم، وتطلب أيضاً من مدير الحدث مراقبة العملية الحالية. إذا ماتت العملية الحالية، فسيتم إزالة معالج الحدث تلقائياً. وهذا منطقي لأنه في حالة `Forwarder` يجب أن توقف عن إعادة توجيه الرسائل إذا لم يعد متلقى هذه الرسائل `(self/0)` (عملية الاختبار) على قيد الحياة.

لنبدأ الآن في تغيير السجل حتى تجتاز الاختبارات بنجاح. افتح `lib/kv/registry.ex` والصق تفاصيل السجل الجديد أدناه (التعليقات المضمنة):

```
defmodule KV.Registry
  use GenServer
  alias Forwarder

  @doc """
  #واجهة برمجة تطبيقات العمل
  """

  ##

  def start_link(__MODULE__, event_manager, opts) do
    Forwarder.start_link(event_manager, opts)
    KV.Registry.start_link(event_manager, opts \\ [])
  end
```

```

@doc """
يبحث عن معرف الدلو لـ `name` المخزن في `server`.

 يتم إرجاع `{ok, pid}` في حالة وجود دلو، `error` وبخلاف ذلك.

"""

def pukoold((الخادم, الاسم))
  ac.revreSneG((الخادم, {البحث, الاسم})) نهاية

@doc """
يتتأكد من وجود دلو مرتبطة بـ "الاسم" المحدد في "الخادم".

"""

def create(server, name)
  tsac.revreSneG((الخادم, {إنشاء, الاسم})) نهاية

## مفاتيح الكلمات استرجاع الخادم

def init(events)
  # تتعلق معاودة الاتصال الأولية الآن مدير الحدث. #
  # لقد قمنا أيضًا بتغيير حالة المدير من مجموعة إلى خريطة، مما يسمح لنا بإضافة خ قول جديدة في المستقبل دون الحاجة إلى إعادة كتابة جميع عمليات الاسترجاع
  names = HashDict.new
  #

  = HashDict.new(المراجع) نهاية

  def handle_call({:lookup, name}, _from, state)
    HashDict.fetch(state.names, name), state}؛#رد،

## فعال إذا كان
def handle_cast({:create, name}, state)
  {noreply, state}

آخر
ref = Process.monitor(pid) refs = HashDict.put(state.refs, ref, name) names = HashDict.put(state.names, name, pid)
{:ok, pid} = KV.Bucket.start_link()

# أرسل إشعارًا إلى مدير الحدث عند إنشاء
GenEvent.sync_notify(state.events, {:create, name, pid}) {noreply, %{state | names: names, refs: refs}} end نهاية # 3.

## فعال إذا كان
def handle_info({:DOWN, ref, :process, pid, _reason}, state)
  tciDhsaH = HashDict.pop(state.refs, {الاسم, المراجع}) الأسماء حذف(المرجع)
  seman.etats = seman.etats نهاية

# أرسل إشعارًا إلى مدير الحدث عند الخروج
GenEvent.sync_notify(state.events, {:exit, name, pid}) {noreply, %{state | names: names, refs: refs}} end نهاية # 4.

## فعال إذا كان
def handle_info(_msg, state)
  {noreply, state} نهاية

```

التغييرات واضحة ومباشرة. نقوم الآن بتمرير مدير الأحداث الذي تلقيناه كحجja إلى start_link إلى تهيئة GenServer. نقوم أيضًا بتغيير كل من عمليات الاستدعاء للبث والمعلومات لاستدعاء GenEvent.sync_notify/2.

وأخيرًا، اغتنمنا الفرصة لتغيير حالة الخادم إلى خريطة، مما يجعل من الأسهل تحسين السجل في المستقبل.

قم بتشغيل مجموعة الاختبار، ويجب أن تصبح جميع الاختبارات باللون الأخضر مرة أخرى.

4.3 تدفقات الأحداث

الوظيفة الأخيرة التي تستحق الاستكشاف هي القدرة على استهلاك أحداثها كدفق:

```
GenEvent.stream(manager). iex> {:ok, manager} = GenEvent.start_link {:ok, #PID<0.83.0>} iex> spawn_link fn -> ...> IO.inspect(x) ...> افعل: <- النهاية
```

نعم:

```
yfiton.tnevEneGiex>
```

نعم:

في المثال أعلاه، قمنا بإنشاء GenEvent.stream(manager) الذي يعيد دفقة (قابلً للعد) من الأحداث التي يتم استهلاكها فور حدوثها. نظرًا لأن استهلاك هذه الأحداث هو إجراء حظر، فإننا ننشئ عملية جديدة مستهلك الأحداث وتطبّعها على المحطة الطرفية، وهذا هو السلوك الذي نراه بالضبط. في كل مرة نستدعي sync_notify/2 أو sync_notify/2، يتم طباعة الحدث على المحطة الطرفية متبعًا بـ :ko (وهو مجرد Ex يطبع النتيجة التي يتم إرجاعها بواسطة وظائف الإشعار).

في كثير من الأحيان توفر تدفقات الأحداث وظائف كافية لاستهلاك الأحداث التي لا تحتاج فيها إلى تسجيل معالجاتها الخاصة. ومع ذلك، عندما تكون هناك حاجة إلى وظيفة مخصصة، أو أثناء الاختبار، فإن تحديد استدعاءات معالج الأحداث الخاصة بنا هي الطريقة الأفضل.

في هذه المرحلة، لدينا مدير أحداث وسجل والعديد من الدلاء المحتملة التي تعمل في نفس الوقت. حان الوقت للبدء في القلق بشأن ما قد يحدث إذا تعطلت أي من هذه العمليات.

5 المشرف والتطبيق

toc.html %} تشمل

حتى الآن، يتطلب تطبيقنا مدير أحداث وسجل. وقد يستخدم عشرات، إن لم يكن مئات، الدلاء. ورغم أنها قد نعتقد أن تنفيذنا حتى الآن جيد جدًا، إلا أن أي برنامج لا يخلو من الأخطاء، ومن المؤكد أن الأخطاء ستحدث.

عندما تفشل الأشياء، قد يكون رد فعلك الأول: "دعنا ننقد تلك الأخطاء". ولكن كما تعلمنا في دليل البدء، في Elixir ليس لدينا عادة البرمجة الداعمة لإنقاذ الاستثناءات، كما هو الحال عادةً في اللغات الأخرى. بدلاً من ذلك، نقول "افشل بسرعة" أو "دعه يتعطل". إذا كان هناك خطأ يؤدي إلى تعطل سجل النظام، فلا داعي للقلق لأننا سنقوم بإعداد مشرف سيبدأ سخة جديدة من السجل.

في هذا الفصل، سنتعرف على المشرفين والتطبيقات أيضًا. لن ننشئ مشرفاً واحداً، بل مشرفين، ونستخدمهما للإشراف على عملياتنا.

5.1 مشرفنا الأول

إن إنشاء مشرف لا يختلف كثيراً عن إنشاء GenServer. سنقوم بتعريف وحدة تسمى KV.Supervisor، داخل الملف lib/kv/supervisor.ex، والتي ستستخدم سلوك المشرف.

```
هل تستخدم المشرف؟ defmodule KV.Supervisor

def start_link do
  (ko: __ELUDOM__knil_trats.
    المشرف.
    نهاية)

KV @registry_name مدير أحداد سجل KV @registry_name

children = [ worker(GenEvent, [[name: @manager_name]]), worker(KV.Registry, [@manager_name, [name: @registry_name]]) ]
def init(:ok) do
]

[ نهاية
الإشراف على (الأطفال، الاستراتيجية: eno_rof_eno) الهاية
```

لدينا مشرفان لديهما طفلان: مدير الأحداث والسجل. من الشائع إعطاء أسماء للعمليات الخاضعة للإشراف حتى تتمكن العمليات الأخرى من الوصول إليها بالاسم دون الحاجة إلى معرفة معرف العملية. هذا مفید لأن العملية الخاضعة للإشراف قد تتعطل، وفي هذه الحالة سيتغیر معرف العملية الخاص بها عندما يعيد المشرف تشغيلها. نعلن عن أسماء أطفال المشرف لدينا باستخدام سمات الوحدة `@manager_name` و `@registry_name`. ثم نشير إلى هذه السمات في تعريفات العامل. في حين أنه ليس مطلوباً أن نعلن عن أسماء عملياتنا الفرعية في سمات الوحدة، إلا أنه مفید، لأن القيام بذلك يساعد في إبرازها لقارئ الكود الخاص بنا.

على سبيل المثال، يتلقى عامل KV.Registry و وسيطين، الأول هي اسم مدير الحدث والثانية هي قائمة كلمات رئيسية للخيارات. في هذه الحالة، نقوم بتعيين خيار الاسم إلى `[name: KV.Registry]` ([استخدام سمة الوحدة النمطية المحددة مسبقاً]). مما يضمن إمكانية الوصول إلى السجل بالاسم.

سجل في جميع أنحاء التطبيق. من الشائع جدًا تسمية أبناء المشرف باسم الوحدة التي تحددهم، حيث يصبح هذا الارتباط مفیداً جدًا عند تصحيح خطأ النظام المباشر.

إن ترتيب إعلان العناصر الفرعية في المشرف له أهمية أيضاً. نظرًا لأن السجل يعتمد على مدير الأحداث، فيجب أن يبدأ الأخير قبل الأول. ولهذا السبب يجب أن يأتي عامل GenEvent قبل عامل KV.Registry في قائمة العناصر الفرعية.

وأخيرًا، نستدعي `supervisor/2` ونمرر قائمة الأطفال واستراتيجية `.eno_rof_eno`:

تحدد استراتيجية الإشراف ما يحدث عندما يتتعطل أحد العناصر الفرعية. إن `eno_rof_eno`: تتعذر في حالة وفاة عنصر فرعي، يتم إعادة تشغيل عنصر فرعي واحد فقط ليحل محله هذه الاستراتيجية منطقية في الوقت الحالي. إذا تعطل مدير الأحداث، فلا يوجد سبب لإعادة تشغيل السجل والعكس صحيح. ومع ذلك، قد تتغير هذه الديناميكيات بمجرد إضافة المزيد من العناصر الفرعية إلى المشرف. يدعم سلوك المشرف العديد من الاستراتيجيات المختلفة وسنناقش ثلثاً منها في هذا الفصل.

إذا بدأنا تشغيل وحدة تحكم داخل مشروعنا باستخدام `iex -S mix`، فيمكننا تشغيل المشرف يدوياً:

```
iex> KV.Supervisor.start_link {:ok, #PID<0.66.0>} iex> KV.Registry.create(KV.Registry, "shopping")
نعم:
{:ok, #PID<0.70.0>} {"السوق" iex> KV.Registry.lookup(KV.Registry,
```

عندما بدأنا شجرة المشرف، تم تشغيل كل من مدير الحدث وعامل التسجيل تلقائياً، مما يسمح لنا بإنشاء دلاء دون الحاجة إلى بدء هذه العمليات يدوياً.

ولكن في الممارسة العملية، نادرًا ما نبدأ تشغيل مشرف التطبيق يدوياً. بل يتم تشغيله كجزء من معاودة الاتصال بالتطبيق.

5.2 التطبيقات فهم

لقد عملنا داخل أحد التطبيقات طوال هذا الوقت. وفي كل مرة قمنا فيها بتحديث ملف تشغيل التجميع المختلط، كان بوسعنا رؤية رسالة Generated kv.app في مخرجات التجميع.

يمكنا العثور على ملف ppa. الناتج في build/_lib/kv/ebin/kv.app. دعنا نلقي نظرة على محتواه:

```
'Elixir.KV.Registry', 'Elixir.KV.Supervisor']}}]. [kernel,stdlib,elixir,logger], {vsn, "0.0.1"}, [{"kv": "Elixir.KV", "Bucket": "Elixir.KV"}]}]
```

يحتوي هذا الملف على مصطلحات Erlang (مكتوبة باستخدام صيغة Erlang). ورغم أننا لستا على دراية بـErlang، فمن السهل تخمين أن هذا الملف يحتوى على تعريف التطبيق الخاص بـErlang. فهو يحتوى على إصدار التطبيق الخاص بـErlang، وجميع الوحدات النمطية التي يحددها، بالإضافة إلى قائمة بالتطبيقات التي تعتمد عليها، مثل دوامة Elixir نفسها، mix.exs والمحدد في قائمة التطبيقات في logger.

سيكون من الممل جدًا تحديث هذا الملف يدوياً في كل مرة نضيف فيها وحدة جديدة إلى تطبيقنا. ولهذا السبب يقوم Mix بإنشاء هذا الملف وصيانته لنا تلقائياً.

يمكنا أيضًا تكوين ملف ppa. الناتج عن طريق تخصيص القيم التي تم إرجاعها بواسطة application.exs داخل ملف مشروع application/exs. ستتناول ذلك في الفصول القادمة.

5.2.1 بدء التطبيقات

عندما نقوم بتعريف ملف ppa، وهو تعريف التطبيق، نتمكن من بدء وإيقاف التطبيق ككل. لم نلقي بشأن هذا الأمر حتى الآن لسببين:

1. يقوم Mix تلقائياً بتشغيل التطبيق الحالي لنا.

2. حتى لو لم يبدأ Mix تطبيقنا نهاية عنا، فلا يحتاج تطبيقنا بعد إلى القيام بأي شيء عند بدء تشغيله.

على أية حال، دعنا نرى كيف يبدأ Mix التطبيق نهاية عنا. دعنا نبدأ وحدة تحكم المشروع باستخدام mix -S ونحاول:

```
iex> Application.start(:kv) {:error, {:already_started, :kv}}
```

أوه، لقد بدأت بالفعل.

يمكنا تمرير خيار إلى mix -S لطلب عدم تشغيل تطبيقنا. دعنا نحاول ذلك بتشغيل iex -S mix run --no-start:

```
iex> Application.start(:kv) {:error, {:not_started, :logger}}
```

الآن نحصل على خطأ لأن التطبيق الذي يعتمد عليه logger: vk (في هذه الحالة) لم يتم تشغيله. يبدأ Mix عادةً التسلسل الهرمي الكامل للتطبيقات المحددة في ملف mix.exs الخاص بـErlang. وي فعل الشيء نفسه لجميع التبعيات إذا كانت تعتمد على تطبيقات أخرى. ولكن نظرًا لأننا مررنا علامة no-start، فنحن بحاجة إما إلى تشغيل كل تطبيق يدوياً بالترتيب الصحيح أو استدعاء Application.ensure_all_started على النحو التالي:

```
iex> Application.ensure_all_started(:kv) {:ok, [:logger, :kv]} iex> Application.stop(:kv) 18:12:10.698 [info]
```

نعم:

لم يحدث شيء مثير حقاً، لكنه يوضح كيف يمكننا التحكم في تطبيقنا.

عند تشغيل الأمر `iex -S mix`، يكون الأمر مماثلاً لتشغيل الأمر `mix run`. كلما احتجت إلى تمرير المزيد من الخيارات إلى الأمر `mix run` عند بدء تشغيل `iex`-`S`، كلما احتجت إلى تمرير المزيد من الخيارات إلى الأمر `mix run`. يمكنك العثور على مزيد من المعلومات حول الأمر `run` من خلال تشغيل الأمر `mix help run` في غلافك.

5.2.2 استدعاء التطبيق

نظرًا لأننا قضينا كل هذا الوقت في الحديث عن كيفية بدء تشغيل التطبيقات وإيقافها، فلا بد أن تكون هناك طريقة للقيام بشيء مفید عند بدء تشغيل التطبيق. وبالفعل، هناك طريقة!

يمكننا تحديد دالة استدعاء التطبيق. هذه هي الدالة التي سيتم استدعاؤها عند بدء تشغيل التطبيق.

يجب أن تقوم الوظيفة بإرجاع نتيجة، حيث `pid` هو معرف العملية لمشرف المشرف.

يمكننا تكون معاودة الاتصال بالتطبيق في خطوتين. أولاً، افتح ملف `mix.exs` وقم بتغيير `def application` إلى ما يلي:

```
def do
  application do
    # ...
    def start(_type, _args) do
      KV.start_link()
      KVSupervisor.start_link()
    end
  end
end
```

يحدد خيار `dom`: "وحدة استدعاء التطبيق"، متبعه بالحجج التي سيتم تمريرها عند بدء تشغيل التطبيق. يمكن أن تكون وحدة استدعاء التطبيق أي وحدة تنفذ سلوك التطبيق.

الآن بعد أن حددنا `KV` كمعاودة الاتصال للوحدة النموذجية، نحتاج إلى تغيير وحدة `KV` المحددة في `lib/kv.ex`:

```
def do
  KV = Application.get_env(:kv, KV)
  KVSupervisor = Application.get_env(:kv, KVSupervisor)

  start(_type, _args) do
    KV.start_link()
    KVSupervisor.start_link()
  end
end
```

عندما نستخدم `Application.get_env`، نحتاج فقط إلى تعريف دالة `start/2`. إذا أردنا تحديد سلوك مخصص عند إيقاف التطبيق، فيمكننا تعريف دالة `stop/1` أيضًا. في هذه الحالة، تكون الدالة المحددة تلقائيًا باستخدام `Application.get_env`.

لنبدأ وحدة التحكم في المشروع مرة أخرى باستخدام الأمر `iex -S mix`. سنرى أن العملية المسماة `KV.Registry` تعمل بالفعل:

```
iex> KV.Registry.create(KV.Registry,
  "التسوق")
:ok
```

ممثراً

5.2.3 مشاريع أو تطبيقات؟

يُفرق بين المشاريع والتطبيقات. بناءً على المحتويات الحالية لملف `mix.exs` الخاص بـ `vk`. يحدد تطبيق `Mix` مشروع `vk`. وكما سنرى في الفصول اللاحقة، هناك مشاريع لا تحدد أي تطبيق.

عندما نقول "مشروع"، يجب أن تفكر في Mix. هي الأداة التي تدير مشروعك. إنها تعرف كيفية تجميع مشروعك واختباره والمزيد. كما أنها تعرف كيفية تجميع التطبيق ذي الصلة بمشروعك وبدء تشغيله.

عندما نتحدث عن التطبيقات، فإننا نتحدث عن OTP. التطبيقات هي الكيانات التي يتم تشغيلها وإيقافها كل بواسطة وقت التشغيل. يمكنك معرفة المزيد عن التطبيقات في مستندات وحدة التطبيق، وكذلك من خلال تشغيل mix help compile.app.

5.3 مشرفين بسيطين من فرد إلى فرد

لقد نجحنا الآن في تحديد المشرف الخاص بنا والذي يتم تشغيله تلقائياً (إيقافه) كجزء من دورة حياة التطبيق لدينا.

لكن تذكر أن KV.Registry الخاص بنا يقوم بربط ومراقبة عمليات الدلو في معاودة الاتصال: handle_cast/2:

```
{:ok, pid} = KV.Bucket.start_link() ref = Process.monitor(pid)
```

الروابط ثنائية الاتجاه، مما يعني أن أي تعطل في أحد الدلو سيؤدي إلى تعطل السجل. رغم أننا لدينا الآن المشرف، الذي يضمن عودة السجل إلى العمل مرة أخرى، فإن تعطل السجل يعني أننا نفقد كل البيانات التي تربط أسماء الدلو بالعمليات الخاصة بها.

عبارة أخرى، نريد أن يستمر السجل في العمل حتى في حالة تعطل أحد الدلو. دعنا نكتب اختباراً:

```
اختبار "إزالة الدلو عند التعطل" . قم بـ KV.Registry.lookup(registry, "shopping") .  
KV.Registry.create(registry, "shopping")
```

#قم بإغلاق الدلو وانتظر الإشعار

```
Process.exit(bucket, :shutdown) assert_receive {:exit, "shopping", ^bucket} assert KV.Registry.lookup(registry, "shopping") == :error end
```

الاختبار مشابه لـ "إزالة الدلو عند الخروج" إلا أننا نكون أكثر صرامة بعض الشيء. فبدلاً من استخدام Agent.stop/1، نرسل إشارة خروج لإغلاق الدلو. ونظرًا لأن الدلو مرتبط بالسجل، والذي يرتبط بدوره بعملية الاختبار، فإن قتل الدلو يتسبب في تعطل السجل، مما يتسبب بدوره في تعطل عملية الاختبار أيضًا:

(1) يقوم الاختبار بإزالة الدلو عند التعطل #PID<0.94.0>#إيقاف التشغيل KV.RegistryTest test/kv/registry_test.exs:52 **

قد يكون أحد الحلول المحتملة لهذه المشكلة هو توفير KV.Bucket.start/0، الذي يستدعي استخدامه من السجل، وإزالة الارتباط بين السجل والدلو، ومع ذلك، قد تكون هذه فكرة سيئة، لأن الدلاء لن تكون مرتبطة بأي عملية بعد هذا التغيير. وهذا يعني أنه إذا أوقف شخص ما تطبيق kv، ففستظل جميع الدلاء نشطة لأنها غير قابلة للوصول.

سنحل هذه المشكلة من خلال تحديد مشرف جديد سيعمل على إنشاء جميع الدلاء والإشراف عليها. هناك استراتيجية إشراف واحدة، تسمى: eno_rof_eno_elpmis، وهي مناسبة تماماً لمثل هذه المواقف: فهي تسمح لنا بتحديد قالب عام والإشراف على العديد من الأبناء بناءً على هذا القالب.

دعونا نقوم بتعريف KV.Bucket.Supervisor على النحو التالي:

```
استخدم المشرف defmodule KV.Bucket.Supervisor
```

```
def start_link(opts \\ [])
```

```

المشرف.(), ko, _ELUDOM__knil_trats) opts) النهائيه
المشرف() قم بذلك tekcub_tratsdef
[]() المشرف.(), dlihc_trats
نهائية

الأطفال def init(:ok)
= []
[工人(yraropmet: إعادة التشغيل: worker(KV.Bucket, [])]
]

الإشراف على (الأطفال، الاستراتيجية: eno_rof_eno_elpmis: ) النهائيه
نهائية

```

هناك تغييرين في هذا المشرف مقارنة بالمشرف الأول.

أولاً، نقوم بتعريف دالة start_bucket/1 التي ستستقبل المشرف وتبدأ عملية دلو ك طفل لهذا المشرف. start_bucket/1 هي الدالة التي سنستدعيها بدلاً من استدعاء KV.Bucket.start_link() مباشرة في السجل.

ثانياً، في معاودة الاتصال ، نقوم بتمييز العامل على أنه مؤقت. وهذا يعني أنه إذا ماتت الدلو، فلن يتم إعادة تشغيلها! وذلك لأننا نريد فقط استخدام المشرف كآلية لتجميع الدلاء، يجب أن يتم إنشاء الدلاء دائماً عبر السجل.

قم بتشغيل mix -S iex حتى نتمكن من تجربة المشرف الجديد لدينا:

```

iex> {:ok, bucket} = KV.Bucket.Supervisor.start_bucket(sup) {:ok, #PID<0.72.0>} iex> KV.Bucket.put(bucket, "eggs", 3)
iex> {:ok, sup} = KV.Bucket.Supervisor.start_link {:ok, #PID<0.70.0>}

نعم:
3 ("البيض") iex> KV.Bucket.get(bucket,

```

دعنا نغير السجل ليعمل مع مشرف الدلاء. ستبقي نفس الإستراتيجية التي أتبعتها مع مدير الأحداث، حيث سنتمرر معرف مشرف الدلاء صراحةً إلى KV.Registry.start_link/3.

لنبدأ بتغيير استدعاء الإعداد في test.exs للقيام بذلك:

```

{:ok, sup} = KV.Bucket.Supervisor.start_link {:ok, manager} = GenEvent.start_link {:ok, registry} = KV.Registry.start_link(manager, sup)

نعم:
GenEvent.add_mon_handler(manager, Forwarder, self()) {:ok, registry: registry}

```

الآن دعنا نغير الوظائف المناسبة في KV.Registry لتأخذ المشرف الجديد في الاعتبار:

```

#واجهة برمجة تطبيقات العميل

@doc """
يبدأ التسجيل.
"""

def start_link(event_manager, buckets, opts \\ [])

```

```

GenServer.start_link(__MODULE__, {event_manager, buckets}, opts) end #قم بتمرير مشرف الدلاء كحجية 1.

#مكالمات استرجاع الخادم#
def init({events, buckets})
  = HashDict.new
    الأسماء
  = HashDict.new
    المراجع
  {ok, %{names: names, refs: refs, events: events, buckets: buckets}} end #قم بتخزين مشرف الدلاء في الحالة 2.

#إلا إذا#
def handle_cast({:create, name}, state)
  = Process.monitor(pid) refs = HashDict.put(state.refs, ref, name) names = HashDict.put(state.names, name, pid) #استخدم مشرف الدلاء بدلاً من نداء الدلاء مباشرةً 3.
  {ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets) ref

#نهاية#
GenEvent.sync_notify(state.events, {:create, name, pid}) {:noreply, %{state | names: names, refs: refs}}

```

يجب أن تكون هذه التعديلات كافية لنجاح اختباراتنا! لإكمال مهمتنا، نحتاج فقط إلى تحديث المشرف الخاص بنا لقبول مشرف الدلاء أيضًا كطفل.

5.4. أشجار الإشراف

لكي نتمكن من استخدام المشرف على الدلاء في تطبيقنا، نحتاج إلى إضافته كطفل لـ KV.Supervisor. لاحظ أننا بدأنا في استخدام المشرفين الذين يشرفون على المشرفين الآخرين، مما يشكل ما يسمى "أشجار الإشراف".

افتتح `lib/kv/supervisor.ex`، وحدة إضافية لاسم مشرف الدلاء، وقم بتغيير `init/1` لتتوافق مع ما يلي:

```

KV @bucket_sup_name KV @registry_name KV @manager_name KV @worker(GenEvent,
  مدبر أحدات سجل دلو مشرف
  [ @manager_name, @bucket_sup_name, .yrtsigeR.VK العامل (@bucket_sup_name)], [.rosivrepuS.tekcuB.VK المشرف (@manager_name)], [.اسم (@اسم [.الاسم (@registry_name))]]]
  ]]

#نهاية#
الإشراف على (الأطفال، الاستراتيجية: eno_r of_eno: eno)

```

هذه المرة، أضفنا مشرفاً كطفل وأعطيته اسم KV.Bucket.Supervisor. كما قمنا بتحديث عامل Registry للتلقي باسم مشرف الدلو كحجية.

نذكر أيضًا أن الترتيب الذي يتم به إعلان الأطفال مهم، نظرًا لأن السجل يعتمد على الدلاء

المشرف، يجب إدراج مشرف الدلاء قبله في قائمة الأطفال.

نظرًا لأننا أضفنا المزيد من العناصر الفرعية إلى المشرف، فمن المهم تقييم ما إذا كانت استراتيجية: eno_rof_eno لا تزال صحيحة. أحد العيوب التي تظهر على الفور هي العلاقة بين السجل ومشرف الدلاء. إذا مات السجل، فيجب أن يموت مشرف الدلاء أيضًا، لأنه بمجرد موته السجل، فقد جميع المعلومات التي تربط اسم الدلاء بعملية الدلاء. إذا تم إيقاع مشرف الدلاء على قيد الحياة، فسيكون من المستحيل الوصول إلى هذه الدلاء.

يمكننا أن نفكر في الانتقال إلى استراتيجية أخرى مثل: eno_rof_eno على قتل وإعادة تشغيل جميع العناصر الفرعية كلما مات أحد العناصر الفرعية. وهذا التغيير ليس مثالياً أيضًا، لأن التعطل في السجل لا يعني أن يؤدي إلى تعطل مدير الأحداث. في الواقع، قد يكون القيام بذلك ضاراً، حيث سيؤدي تعطل مدير الأحداث إلى إزالة جميع معالجات الأحداث المتبقية.

أحد الحلول الممكنة لهذه المشكلة هو إنشاء مشرف آخر يشرف على مشرف السجل والدلاء باستخدام استراتيجية: eno_rof_eno. وجعل المشرف الجذري يشرف على كل من مدير الحدث والمشرف الجديد باستخدام استراتيجية: eno_rof_eno. ستكون الشجرة المقترنة بالتنسيق التالي:

* [one_for_one] مشرف الجذر	* [simple_one_for_one]
* مدير الحدث * [one_for_all]	[mشرف الدلاء *]
* دلاء	
* التسجيل	

يمكنك محاولة إنشاء شجرة الإشراف الجديدة هذه، ولكننا سنتوقف هنا. وذلك لأننا في الفصل التالي سنقوم بإجراء تغييرات على السجل تسمح باستمرار بيانات السجل، مما يجعل استراتيجية eno_rof_eno مناسبة تماماً.

تذكر، أن هناك استراتيجيات أخرى وخيارات أخرى يمكن تقديمها لوظائف supervisor/2 و worker/2، ولذلك لا تنس التحقق من وثائق وحدة المشرف.

نقاط اى اى اس

toc.html %} تشمل {%

في كل مرة نحتاج فيها إلى البحث في أحد الدلائل، نحتاج إلى إرسال رسالة إلى السجل. وفي بعض التطبيقات، يعني هذا أن السجل قد يصبح عقيبة!

في هذا الفصل سوف نتعلم عن، وكيفية استخدامه كآلية تخزين مؤقتة. لاحقًا سوف نوسع استخدامه لاحتفاظ بالبيانات من المشرف إلى أياته، مما يسمح للبيانات بالاستمرار حتى في حالة الأخطاء.

تحذير! لا تستخدم ETS كمخزن مؤقت قبل الأوان! قم بتسجيل وتحليل أداء تطبيقك وتحديد الأجزاء التي تشكل عقبات، حتى تعرف ما إذا كان يجب عليك تخزينها مؤقتًا، وما الذي يجب عليك تخزينه مؤقتًا.
يعتبر هذا الفصل مجرد مثال لكيفية استخدام ETS، بمجرد تحديد الحاجة.

6.1 ETS كذاكرة تخزين مؤقتة

يتيح لنا ETS تخزين أي مصطلح في جدول في الذاكرة. يتم العمل مع جداول ETS عبر وحدة 'ets' في Erlang/Elixir.

```
= :ets.new(:buckets_registry, [:set, :protected])
          ^^^^^^ الجدول
8207
          ^^^^^^
          {"foo", self()} (الجدول, tresni.ste: <xei
          ^^^^^^
          ["foo"])
          ^^^^^^
```

عند إنشاء جداول ETS، يلزم توفر وسيطين: اسم الجدول ومجموعة من الخيارات، ومن الخيارات المتاحة، مررنا نوع الجدول وقواعد الوصول إليه، لقد اختربنا نوع `:ets`، مما يعني أنه لا يمكن تكرار المفاتيح، كما قمنا بتعيين وصول الجدول إلى `:detctorp`، مما يعني أنه لا يمكن إلا للعملية التي أنشأت الجدول الكتابة إليه، ولكن يمكن لجميع العمليات قراءته منه، هذه هي القيمة الافتراضية في الواقع، لذا سنتخطها من الآن فصاعداً.

يمكن أيضًا تسمية جداول ETS، مما يسمح لنا بالوصول إليها باستخدام اسم معين:

```
iex> :ets.new(buckets_registry, [named_table]):buckets_registry iex> :ets.insert(buckets_registry, {"foo", self()})  
  
iex> :ets.lookup(:buckets_registry, "foo") [{"foo", #PID<0.41.0>}]
```

دعنا نغير سجل KV.Registry لاستخدام جداول ETS. سنستخدم نفس التقنية التي استخدمناها مع مدير الأحداث ومشير الدلاء، وستستمر اسم جدول ETS صراحةً على `start_link`. نذكر أنه كما هو الحال مع أسماء الخادم، ستتمكن أي عملية محلية تعرف اسم جدول ETS من الوصول إلى هذا الجدول.

افتح `lib/kv/registry.ex` ولنبدأ في تغيير طريقة تنفيذه. لقد أضفنا تعليقات إلى الكود المصدر لتسلیط الضوء على التغييرات التي أجريناها:

```
defmodule KV.Registry do
  use GenServer
  # يستخدم KV.Registry كجداول ETS
  # واجهة برمجة تطبيقات العميل

  @doc """
  يبدأ التسجيل.
  """

  # قم بذلك
  def start_link(table, event_manager, buckets, opts) do
    GenServer.start_link(__MODULE__, {table, event_manager, buckets}, opts)
  end
  # يتوقع الآن الجدول كحجة ونمرره إلى الخادم #1.

  @doc """
  يبحث عن معرف الدلو لـ `name` في المخزن.
  """

  def lookup(table, name) do
    case :ets.lookup(table, name) do
      [{_, pid}] when pid == self() do
        :ok
      else
        :error
    end
  end
  # يتوقع البحث الآن جدوأ ويبحث مباشرة في #2.

  @doc """
  [-> [":~الاسم, الدلو"]-> [":~حسنا, الدلو"]
  نهاية الخطأ :
  """

  def create(server, name) do
    # يتتأكد من وجود دلو مرتبطة بـ "الاسم" المحدد في "الخادم".
    :tsac.revreSneG.(الخادم, {":~إنشاء, الاسم"})
  end
  # مكالمات استرجاع الخادم ##

  # قم بذلك
  def init({table, events, buckets}) do
    ...
  end
end
```

```
ETS ets = :ets.new(table, [:named_table, readConcurrency: true]) refs = HashDict.newHashDict #3.
```

نهاية{:ok, %{names: ets, refs: refs, events: events, buckets: buckets}}

#4. تم إزالة معاودة الاتصال السابقة handle_call للبحث

```
case lookup(state.names, name) do {ok, _pid} -> {:noreply, state} #5.
```

HashDict

خطا:->

```
ref = Process.monitor(pid) refs = HashDict.put(state.refs, ref, name) :ets.insert(state.names, {name, pid})  
{:ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)
```

```
GenEvent.sync_notify(state.events, {:create, name, pid}) {:noreply, %{state | refs: refs}}
```

نهاية

نهاية

```
HashDict #6. من جدول ETS بدلًا من الحذف def handle_info({:DOWN, ref, :process, pid, _reason}, state) do
```

(الاسم، المراجع) = HashDict.pop(state.refs), {الاسم، etats)eteled.ste: المرجع(

نهايةGenEvent.sync_notify(state.events, {:exit, name, pid}) {:noreply, %{state | refs: refs}}

ذلك قم def handle_info(_msg, state)

نهاية{:noreply, state}

نهاية

لاحظ أنه قبل التغييرات التي أجريناها، كان ملف KV.Registry.lookup يرسل الطلبات إلى الخادم، لكنه الآن يقرأ مباشرةً من جدول ETS، والذي تم مشاركته عبر جميع العمليات. هذه هي الفكرة الرئيسية وراء آلية التخزين المؤقت التي نفذها.

لكي تعمل آلية التخزين المؤقت، يجب أن يكون جدول ETS الذي تم إنشاؤه محميًّا بالوصول (الوضع الافتراضي)، حتى يتمكن جميع العملاء من القراءة منه، بينما تكتب عملية KV.Registry فقط فيه. لقد قمنا أيضًا بتعيين readConcurrency: true.

لقد أدت التغييرات التي أجريناها أعلاه إلى إتلاف اختباراتنا بالتأكيد. بدأ ذي بدء، هناك وسيلة جديدة تحتاج إلى تمريرها إلى KV.Registry.start_link/3. لنبذًا في تعديل اختباراتنا في KV.Registry.start_link/3.

```
{:ok, sup} = KV.Bucket.Supervisor.start_link {:ok, manager} = GenEvent.start_link {:ok, registry} = KV.Registry.start_link(:registry_table, manager, sup) #7.
```

نهايةGenEvent.add_mon_handler(manager, Forwarder, self()) {:ok, registry, ets: :registry_table}

لاحظ أننا نقوم بتمرير اسم الجدول إلى KV.Registry.start_link/3 بالإضافة إلى إرجاع elbat_yrtsiger:registry_table كجزء من سياق الاختبار.

بعد تغيير معاودة الاتصال أعلاه، سنظل نواجه حالات فشل في مجموعة الاختبار الخاصة بنا. وكلها بالتنسيق التالي:

```
1| اختبار تفريغ الدلاء KV.RegistryTest test/kv/registry_test.exs:38 ** (ArgumentError) خطأ وسليمة تتبع المكبس :
```

```
(stdlib) :ets.lookup(#PID<0.99.0>, "shopping") (kv lib/kv/registry.ex:22: KV.Registry.lookup/2 test/kv/registry_test.exs:39
```

يحدث هذا لأننا نمرر معرف التسجيل إلى KV.Registry.lookup/2 بينما يتوقع الآن جدول ETS. يمكننا إصلاح هذا عن طريق تغيير جميع حالات حدوث:

```
(...)(pukool.yrtsigeR.VK(sجل، ...)
```

ل:

```
KV.Registry.lookup(ets, ...)
```

أين سيتم استرجاع ets بنفس الطريقة التي نستعيد بها السجل:

```
%{registry: registry, ets: ets} اختبار "يولد الدلاء".
```

لنغير اختباراتنا لتتمرير ets إلى KV.Registry.lookup/2 بمجرد الانتهاء من هذه التغييرات، ستستمر بعض الاختبارات في الفشل. قد تلاحظ حتى أن الاختبارات تنجح وتفشل بشكل غير متنسق بين التشغيلات. على سبيل المثال، اختبار "spawns buckets":

```
KV.Registry.lookup(ets, "shopping") == :error %{registry: registry, ets: ets} تأكيد اختيار "يولد الدلاء".
```

```
KV.Registry.create(registry, "shopping") assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

```
KV.Bucket.put(bucket, "milk", 1) assert KV.Bucket.get(bucket, "milk") == 1
```

نهاية

قد يكون هناك فشل في هذا الخط:

```
{:ok, bucket} = KV.Registry.lookup(ets, "shopping") تأكيد
```

لكن كيف يمكن لهذا الخط أن يفشل إذا قمنا بإنشاء الدلو في السطر السابق؟

السبب وراء حدوث هذه الإخفاقات هو أننا، لأغراض تعليمية، ارتكبنا خطأين:

- نقوم بالتحسين قبل الأوان (عن طريق إضافة طبقة التخزين المؤقت هذه)

- نستخدم call/2 (بينما يجب علينا استخدام cast/2)

6.2 طروف السياق؟

إن التطوير في Elixir لا يجعل الكود الخاص بك خالياً من حالات التعارض. ومع ذلك، فإن التجربة البسيطة في Elixir حيث لا يتم مشاركة أي شيء بشكل افتراضي تجعل من السهل اكتشاف السبب الجذري لحالة التعارض.

ما يحدث في اختبارنا هو وجود تأخير بين العملية والوقت الذي يمكننا فيه ملاحظة هذا التغيير في جدول ETS. وهذا ما كان متوقع حدوثه:

1. يقوم باستدعاء KV.Registry.create(registry, "shopping") .

2. يقوم السجل بإنشاء الدلو وتحديث جدول ذاكرة التخزين المؤقت

3. يقوم بالوصول إلى المعلومات من الجدول باستخدام KV.Registry.lookup(ets, "shopping") .

4. الأمر أعلاه بعيد {ok, bucket} .

ومع ذلك، نظرًا لأن KV.Registry.create() عبارة عن عملية تحويل، فسوف يعود الأمر قبل أن نكتب فعلًا في الجدول! بعبارة أخرى، يحدث هذا:

1. يقوم باستدعاء KV.Registry.create(registry, "shopping") .

2. يقوم بالوصول إلى المعلومات من الجدول باستخدام KV.Registry.lookup(ets, "shopping") .

3. الأمر أعلاه بعيد `rorre` .

4. يقوم السجل بإنشاء الدلو وتحديث جدول ذاكرة التخزين المؤقت

لصلاح الفشل، تحتاج فقط إلى جعل KV.Registry.create/2 مترافقًا باستخدام `/call` بدلاً من `/cast`. دعنا نغير الوظيفة ووظيفة الاستدعاء الخاصة بها على النحو التالي:

```
def create(server, name)
  # قم بذلك
  # إنشاء الخادم، ([إنشاء، الاسم]) revrSeNneG

  def handle_call({:create, name}, _from, state)
    # قم بذلك
    # البحث عن الحالة
    {ok, pid} = state.names, name) -> {ok, pid}

    pid = Process.monitor(pid)
    refs = HashDict.put(state.refs, ref, name)
    ets.insert(state.names, {name, pid})
    {ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)

    # قم بذلك
    GenEvent.sync_notify(state.events, {:create, name, pid}) # reply, pid, %{state | refs: refs}

  end
  # نهاية
  # نهاية
```

لقد قمنا ببساطة بتغيير معاودة الاتصال من `handle_call/3` إلى `handle_cast/2` وقمنا بتغييرها للرد باستخدام معرف العملية الخاص بالدلو الذي تم إنشاؤه.

لنقم بإجراء الاختبارات مرة أخرى. ولكن هذه المرة سنمرر خيار `--trace`:

```
--trace $
```

بعد خيار `--trace` عندما تكون اختباراتك في طريق مسدود أو توجد ظروف سباق، حيث يقوم بتشغيل جميع الاختبارات بشكل متزامن (لا يؤثر `async: true`) ويعرض معلومات مفصلة حول كل اختبار. هذه المرة يجب أن نكتفي بفشل واحد (قد يكون متقطعاً):

1. يقوم الاختبار بإزالة الدلاء عند الخروج (KV.RegistryTest) test/kv/registry_test.exs:48 ==

الקוד: KV.Registry.lookup(ets, "shopping") == :error lhs: {ok, #PID<0.103.0} .

:shr خطأ:

تنبع المكدس:

اختبار/kv/registry_test.exs:52

وفقاً لرسالة الفشل، توقع أن الدلو لم يعد موجوداً على الجدول، لكنه لا يزال موجوداً! هذه المشكلة هي عكس المشكلة التي حللناها للتو: بينما كان هناك تأخير في السابق بين الأمر بإنشاء دلو وتحديث الجدول، يوجد الآن تأخير بين موت عملية الدلو وإزالة إدخالها من الجدول.

لسوء الحظ، لا يمكننا هذه المرة ببساطة تغيير `handle_info/2` إلى عملية متزامنة. ومع ذلك، يمكننا إصلاح اختباراتنا باستخدام إشعارات مدير الأحداث. دعنا نلقي نظرة أخرى على تنفيذ `handle_info/2` الخاص بنا:

```
HashDict handle_info({:DOWN, ref, :process, pid, _reason}, state) do # 5.
```

```
(الاسم، المراجع) {seman.etats}eteled.state: HashDict.pop(state.refs،
```

```
نهايةGenEvent.sync_notify(state.event، {:exit, name, pid}) {:noreply، %{state | refs: refs}}
```

لاحظ أننا نقوم بالحذف من جدول ETS قبل إرسال الإشعار. هذا مقصوداً وهذا يعني أنه عندما تلقى إشعار `{:exit, name, pid}`، سيكون الجدول محدثاً بالفعل. دعنا نقوم بتحديث الاختبار الفاشل المتبقى على النحو التالي:

```
اختبار "إزالة الدلو عند الخروج" . ets: ets } . KV.Registry.create(registry، "shopping") {:ok، bucket} = KV.Registry.lookup(ets، "shopping")
```

```
KV.Registry.lookup(ets، "shopping") == :error end انتظر الحدثAgent.stop(bucket) assert_receive {:exit، "shopping"، ^bucket} # assert
```

لقد قمنا ببساطة بتعديل الاختبار لضمان استلامنا أولاً رسالة `{:exit, name, pid}` قبل استدعاء `KV.Registry.lookup/2`.

من المهم ملاحظة أننا تمكنا من الحفاظ على استمرارية عمل مجموعتنا دون الحاجة إلى استخدام `peels.remit/1` أو غير ذلك من الجيل. وفي أغلب الأحيان، يمكننا الاعتماد على الأحداث والمراقبة والرسائل للتتأكد على أن النظام في حالة متوقعة قبل تنفيذ التأكيدات.

لتسييل الأمر عليك، إليك حالة الاختبار الناجحة بالكامل:

```
defmodule KV.RegistryTest
  use ExUnit.Case, async: true
```

```
defmodule |عادة التوجيه
  use GenEvent
```

```
def handle_event(event, parent)
  {:ok, parent} = event
```

```
نهاية
نهاية
```

```
sup} = KV.Bucket.Supervisor.start_link {:ok، manager} = GenEvent.start_link {:ok، registry} = KV.Registry.start_link(:registry_table، manager، sup) الإعداد قم
{:ok،
```

```
نهايةGenEvent.add_mon_handler(manager، Forwarder، self()) {:ok، registry: registry، ets: :registry_table}
```

```
اختبار "إرسال الأحداث عند الإنشاء والتعطل" . ets: ets } .
```

```
{:ok, bucket} = KV.Registry.lookup(ets, "shopping") assert_receive {:create, "shopping", ^bucket}
KV.Registry.create(registry, "shopping")
```

```
Agent.stop(bucket) assert_receive {:exit, "shopping", ^bucket}
```

نهاية

KV.Registry.lookup(ets, "shopping") == :error **قم بتأكيد** `%{registry: registry, ets: ets}` اختبار "يولد الدلاء".

```
KV.Registry.create(registry, "shopping") assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

```
KV.Bucket.put(bucket, "milk", 1) assert KV.Bucket.get(bucket, "milk") == 1 end
```

اختبار "إزالة الدلاء عند الخروج".

```
KV.Registry.create(registry, "shopping") {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

KV.Registry.lookup(ets, "shopping") == :error **انتظر الحدث** Agent.stop(bucket) assert_receive {:exit, "shopping", ^bucket} # assert

اختبار "إزالة الدلء عند التعطل".

```
KV.Registry.create(registry, "shopping") {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

قم بإغلاق الدلو وانتظر الإشعار

```
assert_receive {:exit, "shopping", ^bucket} assert KV.Registry.lookup(ets, "shopping") == :error
Process.exit(bucket, :shutdown)
```

نهاية

نهاية

بمجرد نجاح الاختبارات، تحتاج فقط إلى تحديث معاودة الاتصال 1/lib/kv/supervisor.ex التمرين اسم جدول ETS كحجة إلى عامل التسجيل:

```
@ets_registry_name KV @registry_name سجل مدير أحداث
KV.Bucket KV @bucket_sup_name سجل مشرف

= def init(:ok)
  .yrtsigeR.VK(@bucket_sup_name)).:العامل]] .rosivrepuS.tekcuB.VK(@manager_name]]).[[الاسم: المشرف@manager_name]]]]]@worker(GenEvent,
[@ets_registry_name, @manager_name,
@register_name]];@[@bucket_sup_name,
]

[

  (eno_rof_eno على (@الأطفال، الإستراتيجية: eno_rou_eno
نهاية
```

لاحظ أننا نستخدم KV.Registry كاسم لجدول ETS أيضًا، مما يجعل من السهل تصحيح الأخطاء، حيث يشير إلى الوحدة النمطية التي تستخدمه. يتم تخزين أسماء ETS وأسماء العمليات في سجلات مختلفة، لذا لا توجد فرصة

الصراعات.

6.3 ETS كتخزين دائم

حتى الآن، قمنا بإنشاء جدول ETS أثناء تهيئة السجل، ولكننا لم نكلف أنفسنا عباءة إغلاق الجدول عند إنهاء السجل. وذلك لأن جدول ETS "مرتبط" (بشكل مجازي) بالعملية التي أنشأته. وإذا ماتت هذه العملية، فسيتم إغلاق الجدول تلقائياً.

هذا ملائم للغاية كسلوك افتراضي، ويمكننا استخدامه بشكل أكبر لصالحنا. تذكر أن هناك اعتماد بين السجل ومشرف الدلاء. إذا مات السجل، نريد أن يموت مشرف الدلاء أيضاً، لأنه بمجرد موته، فقد جميع المعلومات التي تربط اسم الدلو بعملية الدلو. ومع ذلك، ماذا لو كان بإمكاننا الاحتفاظ ببيانات السجل حتى إذا تعطلت عملية السجل؟ إذا تمكنا من القيام بذلك، فإننا نزيل الاعتماد بين السجل ومشرف الدلاء، مما يجعل استراتيجية eno_rof_eno: الاستراتيجية المثالية لمحارفنا.

سوف يتطلب الأمر إجراء بعض التغييرات لتحقيق ذلك، أولاً، سوف تحتاج إلى بدء تشغيل جدول ETS داخل المشرف. ثانياً، سوف تحتاج إلى تغيير نوع الوصول إلى الجدول protected: إلى cilcup:، لأن المالك هو المشرف، ولكن العملية التي تقوم بالكتابة لا تزال هي المدير.

لنبدأ أولاً بتغيير معاودة الاتصال KV.Supervisor: بـ init/1 الخاصة.

```
def init(:ok)
  ets = :ets.new(@ets_registry_name,
    [:set, :public, :named_table, {:readConcurrency, true}])

  = [ [
    @bucket_sup_name]], .rosivrepus.tekcuB.VK]@الاسم: [المسؤل@manager_name]].worker(GenEvent,
  [ets, @manager_name, .yrtsegeR.VK]@العامل

  @registry_name]]@الاسم:@bucket_sup_name,
]

الإشراف على (الأطفال، الإستراتيجية: eno_rof_eno) النهاية
```

بعد ذلك، نقوم بتغيير استدعاء KV.Registry: بـ init/1 الخاص بـ ، حيث لم يعد هناك حاجة لإنشاء جدول. بل يجب عليه بدلاً من ذلك استخدام الاستدعاء المحدد كحجية:

```
def init({table, events, buckets})
  = HashDict.new

نهاية{:ok, %{names: table, refs: refs, events: events, buckets: buckets}}
```

أخيراً، نحتاج فقط إلى تغيير استدعاء الإعداد في test/kv/registry_test.exs صراحةً. سنستغل هذه الفرصة لتقسيم وظيفة الإعداد إلى وظيفة خاصة ستكون مفيدة قريباً.

```
ets = :ets.new(registry_table, [:set, :public]) registry = start_registry(ets) {ok, registry, ets} end
الإعداد يتم عن طريق الأمر

{:ok, manager} = GenEvent.start_link {:ok, registry} = KV.Registry.start_link(ets, manager, sup)
defp start_registry(ets) do {:ok, sup} = KV.Bucket.Supervisor.start_link
```

```
GenEvent.add_mon_handler(manager, Forwarder, self()) نهاية التسجيل
```

بعد هذه التغييرات، ينبغي لمجموعة الاختبار الخاصة بنا أن تظل خضراء!

يوجد سيناريو آخر يجب مراعاته: بمجرد استلام جدول ETS، قد تكون هناك معرفات موجودة على الجدول. بعد كل شيء، هذا هو الغرض الكامل من هذا التغيير! ومع ذلك، فإن السجل الذي تم تشغيله حديثاً لا يراقب هذه المجموعات، حيث تم إنشاؤها كجزء من السجل السابق الذي لم يعد صالحاً الآن. وهذا يعني أن الجدول قد أصبح قديماً، لأننا لن نزيل هذه المجموعات إذا ماتت.

دعنا نضيف اختباراً إلى `test/kv/registry_test.exs` الذي يُظهر هذا الخطأ:

```
%{registry: registry, ets: ets} do bucket = KV.Registry.create(registry, "shopping") اختبار "مراقبة الإدخالات الموجودة".
```

```
Process.unlink(registry) # قم بإنهاء التسجيل. تقوم بإلغاء الارتباط أولاً، وإلا فسوف يؤدي ذلك إلى إنهاء عملية الاختبار
```

عملية.(السجل، :يقاف التشغيل)

```
start_registry(ets) assert KV.Registry.lookup(ets, "shopping") == {:ok, bucket} # ابدأ سجلاً جديداً بالجدول الموجود وقم بالوصول إلى الدلو
```

```
assert_receive {:exit, "shopping", ^bucket} assert KV.Registry.lookup(ets, "shopping") == :error # بمجرد نفاد الدلو، يجب أن نلقى إشعارات Process.exit(bucket, :shutdown)
```

نهاية

قم بتشغيل الاختبار الجديد وسوف يفشل مع:

```
(KV.RegistryTest)1:1 اختبار مراقبة الإدخالات الموجودة
{:error, "shopping", ^bucket} stacktrace: لا توجد رسالة مطابقة لـ test/kv/registry_test.exs:72
```

اختبار/kv/registry_test.exs:85

هذا ما توقعناه. إذا لم يتم مراقبة الدلو، فلن يتم إخطار السجل عند موته وبالتالي لن يتم إرسال أي حدث. يمكننا إصلاح هذا عن طريق تغيير استدعاء `KV.Registry.init/1` الخاص به إلى `Idlof.init/3` لفحص جميع الإدخالات في الجدول، على غرار `Enum.reduce/3` حيث نستدعي الدالة المعطاة لكل عنصر في الجدول باستخدام المجمع المعطى. في استدعاء الدالة، نراقب كل معرف عملية في الجدول وتحديث قاموس المراجع وفقاً لذلك. إذا كانت أي من الإدخالات ميتة بالفعل، فسنظل نتلقى رسالة `NWOD`. مما يؤدي إلى حذفها لاحقاً.

```
def init({table, events, buckets}) قم بذلك
  acc = <--(الاسم، معرف العملية).refs = :ets.foldl(fn
    HashDict.put(acc, Process.monitor(pid), name) end, HashDict.new, table)
```

نهاية{:ok, %{:names: table, refs: refs, events: events, buckets: buckets}}

نستخدم `Idlof.ste/3` لفحص جميع الإدخالات في الجدول، على غرار `Enum.reduce/3` حيث نستدعي الدالة المعطاة لكل عنصر في الجدول باستخدام المجمع المعطى. في استدعاء الدالة، نراقب كل معرف عملية في الجدول وتحديث قاموس المراجع وفقاً لذلك. إذا كانت أي من الإدخالات ميتة بالفعل، فسنظل نتلقى رسالة `NWOD`. مما يؤدي إلى حذفها لاحقاً.

في هذا الفصل، تمكننا من جعل تطبيقنا أكثر قوّة باستخدام جدول ETS الذي يملكه المشرف ويمرر إلى السجل. كما استكشفنا كيفية استخدام ETS كذاكرة تخزين مؤقتة وناقشنا بعض شروط السوق

قد نواجه مشكلة عندما تصبح البيانات مشتركة بين الخادم وجميع العملاء.

7 التبعيات والمشاريع الشاملة

toc.html %} تشمل

في هذا الفصل، سنناقش بشكل موجز كيفية إدارة التبعيات في Mix.

تم الانتهاء من تطبيق `kv` الخاص بنا، لذاحان الوقت لتنفيذ الخادم الذي سيتعامل مع الطلبات التي حدناها في الفصل الأول:

		إنشاء التسوق
	نعم	
		ضع حليب التسوق 1
	نعم	
		ضع بيض التسوق 3
	نعم	
احصل على الحليب للتسوق	1	
	نعم	
حذف بيض التسوق		
نعم		

ومع ذلك، بدلاً من إضافة المزيد من التعليمات البرمجية إلى تطبيق `kv`، سنقوم ببناء خادم TCP آخر يعمل كعميل لتطبيق `kv`. ونظرًا لأن وقت التشغيل ونظام Elixir بأكمله موجهان نحو التطبيقات، فمن المنطقي تقسيم مشاريعنا إلى تطبيقات أصغر تعمل بدلاً من بناء تطبيق ضخم ومتراوحة.

قبل إنشاء تطبيقنا الجديد، يجب أن نناقش كيفية تعامل Mix مع التبعيات. في الممارسة العملية، هناك نوعان من التبعيات التي نعمل معها عادةً: التبعيات الداخلية والخارجية. يدعم Mix آليات العمل مع كليهما.

7.1 التبعيات الخارجية

التبعيات الخارجية هي تلك التي لا ترتبط بمنطاق عملك. على سبيل المثال، إذا كنت بحاجة إلى واجهة برمجة تطبيقات HTTP لتطبيق `kv` الموزع، فيمكنك استخدام Plug الم مشروع كاعتماد خارجي.

يعد تثبيت التبعيات الخارجية أمراً بسيطاً، في أغلب الأحيان، نستخدم مدير الحزم `Hex` من خلال إدراج التبعية داخل دالة `deps` في ملف `mix.exs` الخاص بنا:

	<code>def deps do [<:plug, "~> 0.5.0"]</code>
نهاية	

يشير هذا التبعية إلى أحد إصدارات `Plug` في سلسلة الإصدارات `0.5.x` التي تم دفعها إلى `Hex`. يتم الإشارة إلى ذلك من خلال `<:plug, "~>` قبل رقم الإصدار. لمزيد من المعلومات حول تحديد متطلبات الإصدار، راجع وثائق وحدة الإصدار.

عادةً، يتم دفع الإصدارات المستقرة إلى `Hex`. إذا كنت تريد الاعتماد على اعتماد خارجي لا يزال قيد التطوير، فإن Mix قادر على إدارة اعتمادات `git` أيضًا:

```
def deps do [{:plug, git: "git://github.com/elixir-lang/plug.git"}] end
```

ستلاحظ أنه عند إضافة تبعة إلى مشروعك، يقوم Mix بإنشاء ملف mix.lock يضم إمكانية تكرار عمليات البناء. يجب تسجيل ملف القفل في نظام التحكم في الإصدار الخاص بك، لضمان أن كل من يستخدم المشروع سيستخدم نفس إصدارات التبعة التي تستخدمها.

يتوفر Mix العديد من المهام للعمل مع التبعيات، والتي يمكن رؤيتها في تعليمات Mix:

<pre>deps.unlock mix deps.update deps.compile mix deps.get mix mix deps mix deps.clean mix \$ mix help</pre>	# قائمة التبعيات وحالتها # قم بإزالة ملفات التبعيات المحددة # تجميع التبعيات # احصل على جميع التبعيات القديمة # فتح التبعيات المحددة # تحديث التبعيات المقدمة
--	--

المهام الأكثر شيوعاً هي mix deps.get و mix deps.update. يمكنك قراءة المزيد حول deps عن طريق كتابة mix help deps. ويأتي Mix.Tasks.Deps. وثائق وحدة.

7. التبعيات الداخلية

التبعيات الداخلية هي تلك الخاصة بمشروعك، وعادةً ما لا يكون لها معنى خارج نطاق مشروعك/شركتك/منظمتك، وفي أغلب الأحيان، ترغب في الحفاظ على خصوصيتها، سواءً لأسباب تقنية أو اقتصادية أو تجارية.

إذا كان لديك اعتماد داخلي، فإن Mix يدعم طريقتين للتعامل معه: مستودعات azo أو مشاريع المظلة.

على سبيل المثال، إذا قمت بدفع مشروع kv إلى مستودع git، فأنت تحتاج فقط إلى إدراجه في كود deps الخاص بك لاستخدامه:

```
def deps do [{:kv, git: "git://github.com/YOUR_ACCOUNT/kv.git"}]
```

نهاية

لا يهم إذا كان مستودع azo عاماً أو خاصاً، فسوف يتمكن Mix من جلب لك طالما أن لديك بيانات الاعتماد المناسبة.

ومع ذلك، فإن استخدام تبعيات azo للتبعيات الداخلية غير مستحسن إلى حد ما في Elixir. تذكر أن وقت التشغيل ونظام البيئي يوفران بالفعل مفهوم التطبيقات. وبالتالي، تتوقع منك تقسيم الكود الخاص بك بشكل متكرر إلى تطبيقات يمكن تنظيمها منطقياً، حتى داخل مشروع واحد.

ومع ذلك، إذا قمت بدفع كل تطبيق كمشروع منفصل إلى مستودع azo، فقد يصبح من الصعب جداً صيانة مشاريعك، لأنه سيعين عليك الآن قضاء الكثير من الوقت في إدارة مستودعات azo بدلاً من كتابة الكود الخاص بك.

لهذا السبب، يدعم Mix "المشاريع الشاملة". تتيح لك المشاريع الشاملة إنشاء مشروع واحد يستضيف العديد من التطبيقات ويدفعها جمِيعاً إلى مستودع azo واحد. هذا هو الأسلوب الذي سنستكشفه بالضبط في الأقسام التالية.

ما سنطلع عليه هو إنشاء مشروع جديد، وسنطلق عليه اسمًا إبداعيًّا، وسيبدو هيكل الدليل على النحو التالي:

```

+ مظلة vk+
+ تطبيقات+
+ kv+
+ خادم+

```

الشيء المثير للاهتمام في هذا النهج هو أن MiX لديه العديد من المزايا للعمل مع مثل هذه المشاريع، مثل القدرة على تجميع واختيار جميع التطبيقات داخل التطبيقات باستخدام أمر واحد. ومع ذلك، على الرغم من إدراجه جميعاً داخل التطبيقات، إلا أنها لا تزال منفصلة عن بعضها البعض، لذلك يمكنك بناء واختيار ونشر كل تطبيق على حدة إذا كنت تريده ذلك.

إذن دعونا نبدأ!

7.3 مشاريع المظلة

لنبأ مشروعًا جديداً باستخدام mix new. سيسُمّي هذا المشروع الجديد umbrella_kv وستحتاج إلى تمرير خيار --عند إنشائه. لا تنسى هذا المشروع الجديد داخل مشروع_kv الحالى!

```

* إنشاء erongitig. $ mix new kv_umbrella --umbrella *
* إنشاء config.exs * إنشاء README.md
* إنشاء config/config.exs

```

من المعلومات المطبوعة، يمكننا أن نرى أن عدد الملفات التي تم إنشاؤها أقل بكثير. ملف mix.exs الناتج مختلف أيضًا.
دعونا نلقي نظرة (تم إزالة التعليقات):

```

defmodule KvUmbrella.Mixfile
  Mix.Project
    do [apps_path: "apps", deps: deps] end
      def project

```

نهاية defp deps do []

نهاية

ما يجعل هذا المشروع مختلفاً عن المشروع السابق هو ببساطة إدخال "apps" في تعريف المشروع، وهذا يعني أن هذا المشروع سيعمل كمظلة. لا تحتوي مثل هذه المشاريع على ملفات مصدرية ولا اختبارات، على الرغم من أنها قد تحتوي على تبعيات متاحة لنفسها فقط. سننشئ مشروع تطبيقات جديدة داخل دليل التطبيقات. نطلق على هذه التطبيقات "أطفال المظلة".

لننتقل إلى داخل دليل التطبيقات ونبدأ في بناء. ستمرر العلم .sup الذي سيخبر Mix بإنشاء شجرة إشراف تلقائياً لنا، بدلاً من بنائها يدوياً كما فعلنا في الفصول السابقة:

```
$ cd kv_umbrella/apps $ mix new kv_server --module KVServer --sup
```

الملفات الناتجة تشبه تلك التي أنشأناها أولاً، مع بعض الاختلافات. لنفتح mix.exs:

```

defmodule KVServer.Mixfile
  # قم بذلك
  # استخدم Mix.Project

  lockfile: "../../mix.lock", elixir: "~> 0.14.1-dev", deps: deps] end

  project do [app: :kv_server, version: "0.0.1", deps_path: "../../deps",
  def

  do [applications: [logger], mod: {KVServer, []}] end
  def application

  # ال نهاية
  defp deps do []
end

```

نهاية

أولاًً وقبل كل شيء، نظرًا لأننا أنشأنا هذا المشروع داخل `apps`،Mix تلقائيًا هيكل المظلة وأضاف سطرين إلى تعريف المشروع:

```

# ملف القفل: "../../mix.lock", deps_path: "../../deps",

```

تعني هذه الخيارات أنه سيتم فحص جميع التبعيات في `kv_umbrella/deps` وستشارك في نفس ملف القفل. يشير هدان السطران إلى أنه إذا كان تطبيقان في المظلة يشتركان في نفس التبعية، فلن يتم جلبهما مرتين. سيتم جلبهما مرة واحدة، وسيضمن Mix تشغيل كلا التطبيقين دائمًا على نفس إصدار التبعية المشتركة.

التغيير الثاني هو في وظيفة التطبيق داخل `mix.exs`:

```

do [applications: [logger], mod: {KVServer, []}] end
def application

```

نظرًا لأننا مررنا علامة `-sup`، فقد أضاف Mix تلقائيًا `mod: {KVServer, []}`، هي وحدة استدعاء التطبيق الخاصة بنا. سيبدأ KVServer شجرة الإشراف على التطبيق الخاص بنا.

في الواقع، دعونا نفتح `lib/kv_server.ex`:

```

defmodule KVServer
  # القيام به
  # استخدم التطبيق

  falseSupervisor.Spec. # تحذير: قم باستيراد
  def start(_type, _args)

  = [الأطفال
  (KVServer.Worker, [arg1, arg2, arg3]) عامل #]
  ]

  KVServer.Supervisor] # الإستراتيجية: eno_rof_eno، الأسماء: المشرف. knl_trats (الأطفال، الخيارات) النهاية

```

نهاية

لاحظ أنه يحدد دالة استدعاء التطبيق، start/2، منتعريف مشرف باسم Supervisor.KVServer. يستخدم وحدة المشرف، فقد قام بتعريف المشرف بشكل ملائم! يمكنك قراءة المزيد حول مثل هذه المشرفين من خلال قراءة وثائق وحدة المشرف.

يمكننا بالفعل تجربة مشروعنا الفرعي الأول. يمكننا تشغيل الاختبارات داخل دليل apps/kv_server، لكن هذا لن يكون ممتنعاً كثيراً. بدلاً من ذلك، انتقل إلى جذر مشروع Umbrella وقم بتشغيل mix test:

اختبار المزيج

إنه يعمل!

نظرًا لأننا نريد أن يستخدم kv_server في النهاية الوظيفة التي حددناها في kv، فنحن بحاجة إلى إضافة kv كعتمادية لتطبيقنا.

7.4 الشاملة التبعيات في

يعد Mix آلية سهلة لجعل أحد عناصر المظلة الفرعية يعتمد على عنصر آخر. افتح deps/app/mix.exs وقم بتغيير دالة 0 إلى ما يلي:

```
defp deps do [{:kv, in_umbrella: true}]
```

يجعل السطر أعلاه `vk`:revres_ `vk`: def. يمكننا استدعاء الوحدات النمطية المحددة في `vk`:revres_ ولكنها لا تبدأ تلقائياً تطبيق `vk`. ولهذا، نحتاج أيضًا إلى إدراج `vk`:revres_ كتطبيق داخل application/

```
defp do
  تطبيق
  [:logger, :kv]
  [ التطبيقات: [:logger, :kv], النهاية mod: {KVServer, []} ]
```

الآن سوف يضمن Mix بدء تشغيل تطبيق `vk` قبل بدء تشغيل `revres_vk`.

أخيرًا، انسخ تطبيق `kv` الذي بنياه حتى الآن إلى دليل التطبيقات في مشروع المظلة الجديد. يجب أن يتتطابق هيكل الدليل النهائي مع الهيكل الذي ذكرناه سابقًا:

```
مظلة vk+
تطبيقات+
+ kv
+ خادم+
```

كل ما نحتاجه الآن هو تعديل ملف `mix.exs` بحيث يحتوي على الإدخالات الشاملة التي رأيناها في ملف `apps/kv_server/mix.exs`. افتح ملف `mix.exs` وافتح ملف `apps/kv/mix.exs` وأضفه إلى دالة المشروع:

```
deps_path: "../../deps", ملف القفل: "../../mix.lock",
```

يمكنك الآن تشغيل الاختبارات لكلا المشروعين من الجذر المظلي باستخدام اختبار المزيج. رائع!

تذكر أن المشاريع الشاملة هي وسيلة مريحة تساعدك على تنظيم وإدارة تطبيقاتك. لا تزال التطبيقات داخل دليل التطبيقات منفصلة عن بعضها البعض. كل تطبيق له تكوينه المستقل، ويجب سرد التبعيات بينهما بشكل صريح. يتيح هذا تطويرها معاً، ولكن تجميعها واختبارها ونشرها بشكل مستقل إذا رغبت في ذلك.

7.5 تلخيص

في هذا الفصل، تعلمنا المزيد عن التبعيات المختلطة والمشاريع الشاملة. لقد قررنا إنشاء مشروع شامل لأننا نعتبر `kv_server` و `kv` تبعيات داخلية لاThem إلا في سياق هذا المشروع.

في المستقبل، سوف تكتب تطبيقات وستلاحظ أنه يمكن استخراجها بسهولة إلى وحدة موجزة يمكن استخدامها من قبل مشاريع مختلفة. في مثل هذه الحالات، يعد استخدام تبعيات `Git` أو `Hex` هو الحل الأمثل.

فيما يلي بعض الأسئلة التي يمكنك طرحها على نفسك عند العمل مع التبعيات. ابدأ بـ هل هذا التطبيق منطقي خارج هذا المشروع؟

إذا لم يكن الأمر كذلك، استخدم مشروع المظلة مع الأطفال المظلبين.

إذا كانت الإجابة بنعم، فهل من الممكن مشاركة هذا المشروع خارج شركتك/مؤسسكتك؟

إذا لم يكن الأمر كذلك، استخدم مستودع `git` أو `Hex`.

إذا كانت الإجابة بنعم، فقم بدفع الكود الخاص بك إلى مستودع `git` وقم بإصدارات متكررة باستخدام `Hex`.

مع مشروع المظلة الخاص بنا جاهزاً للعمل، حان الوقت لبدء كتابة الخادم الخاص بنا.

8.1 المهمة `gen_tcp`

```
toc.html %} تشمل%
```

في هذا الفصل، سوف نتعلم كيفية استخدام وحدة `gen_tcp` لخدمة الطلبات. في الفصول المستقبلية، سنقوم بتوسيع خادمنا حتى يتمكن من خدمة الأوامر بالفعل. سيوفر هذا أيضاً فرصة رائعة لاستكشاف وحدة المهام في Elixir.

8.1 خادم الصدى

سنببدأ تشغيل خادم TCP الخاص بنا من خلال تنفيذ خادم صدى أولًا. سيرسل الخادم ببساطة استجابة بالنص الذي تلقاه في الطلب. سنعمل على تحسين خادمنا ببطء حتى يصبح خاصًا بالإشراف ومستعدًا للتعامل مع اتصالات متعددة.

يقوم خادم TCP، بشكل عام، بتنفيذ الخطوات التالية:

1. يستمع إلى المنفذ حتى يصبح المنفذ متاحًا ويتمكن من الوصول إلى المقابس.

2. ينتظر اتصال العميل على هذا المنفذ ويقبله.

3. يقرأ طلب العميل ويكتب ردًا عليه.

لنبدأ في تنفيذ هذه الخطوات. انتقل إلى تطبيق `kv_server.ex` وافتح `apps/kv_server` وأضف الوظائف التالية:

```
def accept(port) do
  # يستقبل البيانات كملفات ثنائية (بدلاً من القوائم) - 1. # الحبارات أدناه تعني: -
  # يستقبل البيانات سطراً بسطراً - 2. 'packet: :line' - 3. 'active: false'
  {:ok, socket} = :gen_tcp.listen(port, [:binary, packet: :line, active: false])
  IO.puts "قبول الاتصالات على المنفذ #{port}"
```

حتى توفر البيانات #

```
{:ok, socket} = :gen_tcp.listen(port, [:binary, packet: :line, active: false])
```

```
#{port}" loop_acceptor(socket) end "قبول الاتصالات على المنفذ
```

الإصدارات Elixir، توثيق

```

do {ok, client} = :gen_tcp.accept(socket) serve(client) loop_acceptor(socket) end
defp loop_acceptor(socket)

defp serve(client) do
    عميل
    >|قراءة السطر ()|
    >|كتابة سطر(العميل)|
خدمة (العميل) نهاية

defp read_line(socket) do {ok, data} = :gen_tcp.recv(socket, 0) data
نهاية

defp write_line(line, socket) do :gen_tcp.send(socket, line) end

```

نبأً تشغيل خادمنا عن طريق استدعاء . accept(KVServer.accept(4040) هو المنفذ. الخطوة الأولى في accept/1 هي الاستماع إلى المنفذ حتى يصبح المقبس متاخماً ثم استدعاء loop_acceptor/1. loop_acceptor/1. serve/1، نستدعي

إن serve/1 هي عبارة عن حلقة أخرى تقرأ سطراً من المقبس وتكتب هذه الأسطر مرة أخرى في المقبس. لاحظ أن دالة serve/1 مستخدمة عامل خط الأنابيب __>|>|2>'Kernel.html#|>|>|'docs/stable/elixir/>|>'|" للتعبير عن تدفق العمليات هذا. يقوم عامل خط الأنابيب بتقييم الجانب الأيسر ويمرر نتيجته كحجة أولى إلى الدالة الموجودة على الجانب الأيمن. المثال أعلاه:

```
(mcqbs|> read_line() |> enil_etirw|>
```

يعادل:

```
كتابة سطر (قراءة سطر (المقبس) المقبس)
```

عند استخدام عامل ، ``|>|`` من المهم إضافة أقواس إلى استدعاءات الوظيفة بسبب كيفية عمل أولوية العامل. على وجه الخصوص، هذا الكود:

```
::
```

```
&(&1 * 2) |> 1..10 |> مرشح عددي
```

يترجم في الواقع إلى:

```
::
```

```
* 2))(( 1&(&1 * 2) |> 1..10 |> مرشح عددي
```

وهذا ليس ما نريده، لأن الدالة المعطاة لـ "Enum.filter/2" هي الدالة التي تم تمريرها كحجة أولى لـ "Enum.map/2". الحال هو استخدام أقواس صريحة:

```
::
```

```
(&(&1 * 2)) | > 5..10 مرشح عددي (&(&1 <= 5)) | >
```

يتلقى تنفيذ 1/read_line/البيانات من المقبس باستخدام: vcer.pct_neg 2 إلى المقبس باستخدام: dnes.pct_neg 2 ويكتب

هذا هو كل ما نحتاجه تقريرنا لتنفيذ خادم الصدى الخاص بنا. فلنجرها

ابداً جلسة mix-S_kv_server باستعمال:_kv_server داخل التطبيق، Exeix-S mix.

```
ie> KVServer.accept(4040)
```

يعلم الخادم الآن، وستلاحظ أيضاً أن وحدة التحكم محظوظة. دعنا نستخدم عميل telnet<<http://en.wikipedia.org/wiki/Telnet>> للوصول إلى خادمنا. هناك عملاء متاحون على معظم أنظمة التشغيل، وخطوط الأوامر الخاصة بهم متشابهة بشكل عام:

```
$ تيلنت 127.0.0.1 4040
محاولة
localhost. متصل بـ
حرف الهروب هو .". مرحبا
```

```
مرحبا
هل هو اانا
هل هو اانا
هل تبحث عن؟ هل تبحث عن؟
```

أكتب "hello" واضغط على Enter واستلقي "hello" مرة أخرى. ممتازاً

يمكن الخروج من عميل telnet الخاص بي عن طريق كتابة Ctrl + [quit]، ولكن قد يتطلب عميلك خطوات مختلفة.

بمجرد الخروج من عميل telnet، فمن المحموم أن ترى خطأ في جلسة: EX:

```
lib/kv_server.ex:33: KVServer.serve/1 (kv_server) lib/kv_server.ex:27: KVServer.loop_acceptor/1 :MatchError)
{:error, :closed} (kv_server) lib/kv_server.ex:41: KVServer.read_line/1 (kv_server)
```

يرجع ذلك إلى أنها كانت تتوقع بيانات recv/2: tcp.gen: نعم ولكن العميل أغلق الاتصال. نحتاج إلى التعامل مع مثل هذه الحالات بشكل أفضل في المراجعات المستقبلية لخادمنا.

في الوقت الحالي، هناك خطأ أكثر أهمية تحتاج إلى إصلاحه: ماذا يحدث إذا تعطل جهاز الاستقبال TCP الخاص بنا؟ نظرًاً لعدم وجود إشراف، يتوقف الخادم عن العمل ولن نتمكن من تقديم المزيد من الطلبات، لأنه لن يتم إعادة تشغيله. ولهذا السبب يجب علينا نقل خادمنا داخل شجرة الإشراف.

18.2 المهام

لقد تعلمنا عن الوكلاء والخدمات العامة ومديري الأحداث. وكلها مصممة للعمل مع رسائل متعددة أو إدارة الحالة. ولكن ماذا نستخدم عندما نحتاج فقط إلى تنفيذ مهمة ما وهذا كل شيء؟

توفر وحدة المهام هذه الوظيفة بالضبط. على سبيل المثال، تحتوي على دالة start_link/3 التي تستقبل وحدة ودالة وحججاً، مما يسمح لنا بتشغيل دالة معينة كجزء من شجرة الإشراف.

دعنا نحاول ذلك. افتح lib/kv_server.ex، ولنغير المشرف في الدالة start/2 إلى ما يلي:

```
def start(_type, _args) do
  Supervisor.Spec
```

```
الأطفال = [عامل (المهمة، : [4040]))، قبول[KVServer]
```

]

```
الخيارات = [الاسم: eno_rof_eno، الاستراتيجية: KVServer.Supervisor]
```

المشرف (الأطفال، الخيارات) knil_trats.

نهاية

من خلال هذا التغيير، نقول إننا نريد تشغيل KVServer.accept(4040) كعامل. نقوم حالياً بترميز المنفذ بشكل ثابت، ولكننا ستناقش الطرق التي يمكن بها تغيير ذلك لاحقاً.

الآن بعد أن أصبح الخادم جزءاً من شجرة الإشراف، يجب أن يبدأ تلقائياً عند تشغيل التطبيق. اكتب mix run --no-halt في المحطة الطرفية، واستخدم عميل telnet مأمور آخر للتتأكد من أن كل شيء لا يزال يعمل:

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
قل لك
قل لك
قل لي
قل لي
```

نعم، إنه يعمل! إذا قتلت العميل، مما تسبب في تعطل الخادم بالكامل، فسترى عميلاً آخر يبدأ على الفور. ومع ذلك، هل هو قابل للقياس؟

حاول توصيل عميلين telnet في نفس الوقت. عندما تفعل ذلك، ستلاحظ أن العميل الثاني لا يصدر صدى:

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
مرحبا؟
مرحبا؟!
```

لا يبدو أن الأمر يعمل على الإطلاق. وذلك لأننا نخدم الطلبات في نفس العملية التي تقبل الاتصالات. وعندما يتم توصيل عميل واحد، لا يمكننا قبول عميل آخر.

8.3 مشرف المهمة

لكي يتمكن خادمنا من التعامل مع الاتصالات المتزامنة، نحتاج إلى أن يكون لدينا عملية تعمل كمستقبل يولد عمليات أخرى لخدمة الطلبات. أحد الحلول هو تغيير:

```
do {ok, client} = :gen_tcp.accept(socket) serve(client) loop_acceptor(socket) end
    defp loop_acceptor(socket)
```

لاستخدام Task.start_link/1، وهو مشابه لـ Task.start_link/3، لكنه يتلقى دالة مجهولة بدلاً من الوحدة والوظيفة والحجج:

```
defp loop_acceptor(socket) do {ok, client} = :gen_tcp.accept(socket)
```

```
-> serve(client) end nf) knil_trats.
```

```
loop_acceptor(socket)
```

لكتنا ارتکينا هذا الخطأ من قبل. هل تتذكر؟

هذا مشابه للخطأ الذي ارتکبناه عندما قمنا باستدعاء KV.Bucket.start_link/0 من السجل. وهذا يعني أن الفشل في أي دلو من شأنه أن يؤدي إلى تعطل السجل بأكمله.

سيكون لكود أعلاه نفس الخلل: إذا قمنا بربط مهمة (client) بالمستقبل، فإن التعطل عند تقديم الطلب سيؤدي إلى إيقاف المستقبل، وبالتالي جمیع الاتصالات الأخرى.

لقد قمنا بإصلاح المشكلة المتعلقة بالسجل باستخدام طريقة بسيطة واحدة لمشرف واحد. وسنستخدم نفس التكتيك هنا، إلا أن هذا النمط شائع جداً مع المهام التي تأتي بالفعل مع حل: طريقة بسيطة واحدة لمشرف واحد مع عمال مؤقتين يمكننا استخدامها في شجرة الإشراف الخاصة بنا!

دعنا نغير 2/startمرة أخرى، لإضافة مشرف إلى شجرتنا:

```
Supervisor.Spec  قم باستيراد
def start(_type, _args)
  = الأطفال آ
  [4040]]]
  المشرف(مشرف المهمة، [[العامل(m مهمة)، العامل(revrevSVK، قبول، revreSVK]]]
]
```

```
KVServer.Supervisor] = الإستراتيجية؛ eno_rof_eno: الاسم؛
                                               المشرف(knil_trats، الأطفال، الخيارات)
```

نهاية

كل ما علينا فعله هو بدء عملية `_Task.Supervisor` باسم `</docs/stable/elixir/Task.Supervisor.html>`.

الآن نحتاج فقط إلى تغيير `loop_acceptor/2` لخدمة كل طلب:

```
defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  loop_acceptor(socket)
```

المهمة، المشرف، بدء التشغيل الفرعى (KVServer.TaskSupervisor، fn -> serve(client))

ابداً تشغيل خادم جديد باستخدام الأمر `--no-halt mix run --no-halt` ويمكننا الآن فتح العديد من عملاء telnet المتزامنين. ستلاحظ أيضاً أن إغلاق العميل لا يؤدي إلى إيقاف تشغيل جهاز الاستقبال، ممتازاً

فيما يلي تنفيذ خادم الصدى الكامل، في وحدة واحدة:

```
defmodule KVServer
  القيام به
  استخدام التطبيق
  قم باستيراد @doc false def start(_type, _args)
  Supervisor.Spec
```

```
= الأطفال آ
[4040]]]
المشرف(مشرف المهمة، [[العامل(m مهمة)، العامل(revrevSVK، قبول، revreSVK]]]
]
```

```
KVServer.Supervisor] = الإستراتيجية؛ eno_rof_eno: الاسم؛
```

```
المشرف. knil_trats(الأطفال، الخيارات) النهاية
```

```

@doc """
يبدأ قبول الاتصالات على المنفذ المحدد.

"""

def accept(port) do
  {:ok, socket} = :gen_tcp.listen(port, [binary, packet: :line, active: false])

  #{{port}} IO.puts "قبول الاتصالات على المنفذ {{port}}"
  loop_acceptor(socket)
end

defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  KVServer.TaskSupervisor.loop_acceptor(client)
end

defp serve(socket) do
  socket
  |> read_line()
  |> write_line()
end

#{{نهاية الخدمة (المقبس)}}

defp read_line(socket)
  {:ok, data} = :gen_tcp.recv(socket, 0)
  data
  #{{نهاية}}
end

defp write_line(line, socket) do
  :gen_tcp.send(socket, line)
end

```

نهاية

وبما أننا قمنا بتغيير مواصفات المشرف، فنحن بحاجة إلى أن نسأل: هل استراتيجية الإشراف لدينا لا تزال صحيحة؟

في هذه الحالة، الإجابة هي نعم: إذا تعطلت وحدة الاستقبال، فلا توجد حاجة لتعطيل وحدة المشرف. من ناحية أخرى، إذا تعطلت مشرفة المهام، فلا توجد حاجة لتعطيل وحدة الاستقبال أيضًا. وهذا يتناقض مع السجل، حيث كان علينا في البداية تعطل المشرف في كل مرة تعطلت فيها وحدة الاستقبال، حتى استخدمنا ETS للحفاظ على الحالة. ومع ذلك، لا تحتوي المهام على حالة ولن يصبح أي شيء قد يدلي إذا ماتت إحدى هذه العمليات.

في الفصل التالي سنبدأ بتحليل طلبات العملاء وإرسال الاستجابات، وننهي خادمنا.

9 المستندات والاختبارات والأنباب

toc.html %} %

في هذا الفصل، سنقوم بتنفيذ الكود الذي يحل الأوامر التي وصفناها في الفصل الأول:

إنشاء التسوق
نعم

1 ضع حليب التسوق
نعم

<input checked="" type="checkbox"/> نعم <input type="checkbox"/> لا	<input checked="" type="checkbox"/> نعم <input type="checkbox"/> لا
<input checked="" type="checkbox"/> نعم <input type="checkbox"/> لا	<input checked="" type="checkbox"/> نعم <input type="checkbox"/> لا

بعد الانتهاء من التحليل، سنقوم بتحديث خادمنا لإرسال الأوامر المحللة إلى تطبيق: `vk`: الذي قمنا ببنائه مسبقاً.

19.1 الاختبارات الوثائقية

في الصفحة الرئيسية للغة، ذكر أن Elixir يجعل التوثيق مواطناً من الدرجة الأولى في اللغة. لقد استكشفنا هذا المفهوم عدة مرات طوال هذا الدليل، سواء من خلال مساعدة `mix` أو عن طريق كتابة `Enum` أو وحدة أخرى في وحدة `IEx`. تحكم

في هذا القسم، سنتنفذ وظيفة التحليل باستخدام `doctests`، والتي تسمح لنا بكتابة الاختبارات مباشرةً من وثائقنا. وهذا يساعدنا على توفير الوثائق مع عينات أكواد دقيقة.

دعنا نقوم بإنشاء محلل الأوامر الخاص بنا في `lib/kv_server/command.ex`:

```
defmodule KVServer.Command do
  @doc ~S"""
    يقوم بتحليل السطر المحدد إلى أمر.
  """
  # أمثلة
  iex> KVServer.Command.parse "CREATE shopping\r\n" {:ok, {:create, "shopping"}}

  """
  parse(line) do :not_implemented end
  def
  """
  نهاية
```

يتم تحديد الاختبارات الوثائقية في سطر من خلال مسافة بادئة مكونة من أربع مسافات متتابعة بموجة `iex` في سلسلة وثائقية. إذا امتد الأمر إلى عدة سطور، فيمكنك استخدام `...>`، كما في `IEx`. يجب أن تبدأ النتيجة المتوقعة في السطر التالي بعد سطر (أسطر) `iex` أو ...> وتنتهي إما بسطر جديد أو بارنة `iex` جديدة.

لاحظ أيضاً أننا بدأنا سلسلة التوثيق باستخدام `doctest KVServer.Command`. يمنع `-S` تحويل حرف `\n` إلى إرجاع عربة وتغذية سطر حتى يتم تقييمها في الاختبار.

لتتشغيل اختباراتنا الوثائقية، سنقوم بإنشاء ملف في `test/kv_server/command_test.exs` في حالة الاختبار:

```
defmodule KVServer.CommandTest
  use ExUnit.Case, async: true
  doctest KVServer.Command

  نهاية
```

قم بتشغيل مجموعة الاختبار ويجب أن يفشل الاختبار الوثائقى:

```
1|اختبار المستند في KVServer.Command.parse/1 (1) (KVServer.CommandTest)
KVServer.Command.parse "CREATE shopping\r\n" === {:ok, {:create, "shopping"}}
|lhs: :not_implemented stacktrace: Kod فشل test/kv_server/command_test.exs:3
|Doctest :
```

(وحدة نمطية) lib/kv_server/command.ex:11: KVServer.Command

متزماً

الآن الأمر يتعلق فقط بجعل اختبار الوثيقة يجتاز الاختبار. دعنا ننفذ وظيفة parse/1:

```
def parse(line) do
  case String.split(line) do
    ["CREATE", bucket] -> {:ok, {:create, bucket}}
  end
```

نهاية

إن تفيناً تقوم ببساطة بتقسيم السطر على المسافات البيضاء ثم مطابقة الأمر مع قائمة. إن استخدام String.split يعني أن أوامرنا لن تكون حساسة للمسافات البيضاء. ولن يكون للمسافات البيضاء البداية واللاحقة أي أهمية، ولن تكون المسافات المتتالية بين الكلمات أي أهمية. دعنا نضيف بعض الاختبارات الوثائقية الجديدة لاختبار هذا السلوك مع الأوامر الأخرى:

```
@doc ~S"""
يقوم بتحليل السطر المحدد إلى أمر.

#أمثلة
iex> KVServer.Command.parse "CREATE shopping\r\n" {:ok, {:create, "shopping"}}

iex> KVServer.Command.parse {"إنشاء التسوق", "shopping"} {:ok, {:create, "shopping"}}

iex> KVServer.Command.parse " وضع حليب التسوق" {"وضع حليب التسوق", "milk"} {:ok, {:put, "shopping", "milk", "1"}}

iex> KVServer.Command.parse "GET shopping milk\r\n" {:ok, {:get, "shopping", "milk"}}

iex> KVServer.Command.parse "حذف بيض التسوق" {"حذف بيض التسوق", "eggs"} {:ok, {:delete, "shopping", "eggs"}}

iex> KVServer.Command.parse "غير المعروفة التسوق" {"غير المعروفة التسوق", "unrecognized_command"} {:error, :unknown_command}
```

تؤدي الأوامر غير المعروفة أو الأوامر التي تحتوي على عدد خطأ من الوسائط إلى إرجاع خطأ:

```
iex> KVServer.Command.parse "GET shopping\r\n" {:error, :unknown_command}
```

.....

مع وجود الاختبارات في متناول يدك، حان دورك لاجتياز الاختبارات! بمجرد أن تصبح جاهزاً، يمكنك مقارنة عملك بالحل الذي نقدمه أدناه:

```
def parse(line) do
  case String.split(line) do
    ["إنشاء", دلو] >-> {إنشاء, دلو}
    ["ـ", دلو, مفتاح] >-> {"ـ", دلو, مفتاح}
    ["ـ", دلو, مفتاح, GET] >-> {"ـ", دلو, مفتاح, GET}
    ["ـ", دلو, مفتاح, ok] >-> {"ـ", دلو, مفتاح, ok}
    [{"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}] >-> {"ـ", دلو, مفتاح, "ـ", دلو, مفتاح}
    [{"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}] >-> {"ـ", دلو, مفتاح, "ـ", دلو, مفتاح, "ـ", دلو, مفتاح}
    [{"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}] >-> {"ـ", دلو, مفتاح, "ـ", دلو, مفتاح, "ـ", دلو, مفتاح, "ـ", دلو, مفتاح}
    [{"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}] >-> {"ـ", دلو, مفتاح, "ـ", دلو, مفتاح, "ـ", دلو, مفتاح, "ـ", دلو, مفتاح, "ـ", دلو, مفتاح}
    [{"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}] >-> {"ـ", دلو, مفتاح, "ـ", دلو, مفتاح}
    [{"ـ", دلو, مفتاح}, {"ـ", دلو, مفتاح}] >-> {"ـ", دلو, مفتاح, "ـ", دلو, مفتاح}
```

لاحظ كيف تمكنا من تحليل الأوامر بآنقة دون إضافة مجموعة من جمل `else/if`التي تتحقق من اسم الأمر وعدد الوسائط!

أخيراً، بينما لا يلاحظ أن كل اختبار توثقي كان يعتبر اختباراً مختلفاً في حالة الاختبار الخاصة بنا، حيث يبلغ مجموع اختباراتنا الان 7 اختبارات. وذلك لأن `ExUnit` يأخذ في الاعتبار ما يلي لتحديد اختبارين مختلفين:

```
{:error, :unknown_command} > KVServer.Command.parse "بيض التسوق غير المعروف\\n"
```

```
iex> KVServer.Command.parse "GET shopping\r\n" {:error, :unknown_command}
```

بدون أسطر جديدة، كما هو موضح أدناه، يقوم ExUnit بتجميعها في اختبار واحد:

```
"GET shopping\r\n" {:error, :unknown_command} iex> KVServer.Command.parse "\r\n" {:error, :unknown_command} iex> KVServer.Command.parse
```

يمكنك قراءة المزيد عن اختبارات الوثائق في [وثائق ExUnit.DocTest](#).

خطوط الأنابيب 9.2

مع وجود محلل الأوامر في متناول أيدينا، يمكننا أخيراً البدء في تتنفيذ المنطق الذي يقوم بتشغيل الأوامر. دعنا نضيف تعريضاً مدمجاً لهذه الوظيفة في الوقت الحالي:

```
defmodule KVServer.Command do
  @doc """
  يقوم بتشغيل الأمر المحدد.
  """
  def run(command) do
    case command do
      "put" <-> {key, value} = String.split!(System.argv() -- ["put"], " ")
      "get" <-> key = System.argv() -- ["get"] -- [key]
      _ <-> IO.puts("Unknown command")
    end
    IO.puts("OK\r\n")
  end
end
```

قبل أن ننفذ هذه الوظيفة، دعنا نغير خادمنا للبدء في استخدام وظائف `parse` و `run` الجديدة.

تذكرة أن وظيفة `read_line` الخاصة بنا كانت تتغافل تماماً عن إغلاق العميل المقيد، لذا فلنستغل الفرصة لإصلاحها أيضاً. افتح `kv_server.ex` واستبدل تعريف الخادم بالخطي:

```
defp serve(socket) do
  المقبس
    | قراءة السطر ()
    |> كتابة سطر (المقبس)
  نهاية الخدمة (المقبس)
```

```
defp read_line(socket)
  {ok, data} = :gen_tcp.recv(socket, 0) data

نهاية
```

defp write_line(line, socket) do :gen_tcp.send(socket, line) end

مع ما يلي:

```
defp serve(socket) do
  msg = case read_line(socket) do {ok, data} -> case KVServer.Command.parse(data) do
```

{ok, command} -> KVServer.Command.run(command)

-> {خطأ, _}

خطأ

نهاية

-> {_خطأ, خطأ}

خطأ

نهاية

write_line(socket, msg) serve(socket)

نهاية

defp read_line(socket) do :gen_tcp.recv(socket, 0) end

```
defp write_line(socket, msg)
  :gen_tcp.send(socket, format_msg(msg))
```

قام بما يلي: "أمر غير معروف"\n`defp format_msg({error, _}). "ما يلي: " قم بما يلي: النص.`
 بما يلي: "خطأ"

127.0.0.1 telnet \$ جاري المحاولة ... 4040

متصل بـ localhost

حرف الهروب هو ']' .

إنشاء التسوق موافق

مرحبا

أمر غير معروف

وهذا يعني أن تنفيذنا يسير في الاتجاه الصحيح، لكنه لا يبدو أنيقاً جدًا. أليس كذلك؟

استخدم التنفيذ السابق الأنابيب التي جعلت المنطق واضحًا ومبashراً لفهم:

```
read_line(socket) |> KVServer.Command.parse |> KVServer.Command.run()
```

نظرًا لأنه قد يكون لدينا إخفاقات على طول الطريق، فنحن بحاجة إلى أن يتطرق خط الأنابيب لدينا مع مخرجات الأخطاء ويتوقف في حالة حدوثها. ألا يكون من رائع لو تمكنا بدلاً من ذلك من القول: "قم بنقل هذه الوظائف بينما تكون الاستجابة: "ok" أو "failure"؟" بينما تتطابق الاستجابة مع مجموعة {"ok", "failure"}.

ولحسن الحظ، هناك مشروع يسمى `elixir-pipes` الذي يوفر هذه الوظيفة بالضبط! دعنا نحاول ذلك.

افتح ملف `mix.exs` وقم بتغيير الوظيفتين `application/0` و `deps/0` إلى ما يلي:

```
def do
  # تطبيق
  [KVServer, []] = application:locate_nearest([logger, :pipe, :kv])
  [KVServer] = [KVServer]
  defp deps do
    [{"pipe", github: "batate/elixir-pipes"}]
```

قم بتشغيل `mix deps.get` للحصول على التبعية، وأعد كتابة وظيفة `serve/1` لاستخدام وظيفة `pipe_matching/3` الممتاحة لنا الآن:

`def serve(socket)` لا يستورد الأنابيب

```
msg = pipe_matching x, {:ok, x}, read_line(socket)

|> KVServer.Command.parse()
|> KVServer.Command.run()

write_line(socket, msg) serve(socket) end
```

باستخدام `pipe_matching/3` يمكننا أن نطلب من Elixir أن ينقل القيمة `x` من كل خطوة إذا كانت تطابق `.{ok, x}`. نقوم بذلك عن طريق تحويل كل تعبير معطى إلى `case` في خط الأنابيب. بمجرد أن تعيّد أي من الخطوات شيئاً لا يتطابق مع `.{ok, x}` يتم إلغاء خط الأنابيب، ويعيد القيمة غير المطابقة.

ممتنًا لا تتردد في قراءة [أنابيب eksipir](#) توثيق المشروع للتعرف على خيارات أخرى للتعبير عن خطوط الأنابيب. دعونا نستمر في المضي قدماً في تفاصيل الخادم الخاص بنا.

أوامر التشغيل 9.3

الخطوة الأخيرة هي تنفيذ `KVServer.Command.run/1` لتشغيل الأوامر المحللة على تطبيق `vk`. يظهر تنفيذه أدناه:

```
@doc """
...
 يقوم بتشغيل الأمر المحدد.

(الأمر)nurdef run({:create, bucket})
  {:ok, "OK\r\n"}(لـKV.Registry.create(KV.Registry,
```

نهاية

```
bucket. fn pid -> البحث في def run({:get, bucket, key})
```

القيمة = KV.Bucket.get(pid, key) {:ok, "#{value}\r\nOK\r\n"}

نهاية

نهاية

```
def run({:put, bucket, key, value})
```

دلو البحث، pid -> KV.Bucket.put(pid, key, value) {:ok, "OK\r\n"},
النهايةfn

نهاية

```
bucket. fn pid -> البحث في def run({:delete, bucket, key})
```

KV.Bucket.delete(pid, key) {:ok, "OK\r\n"}

نهاية

```
البحث عن callback
```

KV.Registry.lookup(KV.Registry, bucket)

حالة{:ok, pid} -> callback.(pid) :error -> {:error, :not_found}

نهاية

التنفيذ بسيط: نقوم فقط بإرساله إلى خادم KV.Registry الذي سجلناه أثناء بدء تشغيل تطبيق .vk:

لاحظ أننا قمنا أيضًا بتعريف دالة خاصة باسم KV.lookup المساعدة في الوظيفة الشائعة المتمثلة في البحث عن دلو وإرجاع معرف العملية الخاص به إذا كان موجودًا.

:error, :not_found} وبخلاف ذلك

بالمناسبة، بما أننا نرجع الآن، فيجب علينا تعديل دالة KV.Server.format_msg/1 لإظهار رسائل عدم العثور عليها بشكل جيد أيضًا:

```
format_msg({:error, :not_found}), "أمر غير معروف\r\nافعل: نص", defp format_msg({:error, :unknown_command}), "خطأ\r\nافعل: لم يتم العثور عليه", _), "n\r\n" defp format_msg({:error, _})
```

لقد اكتملت وظائف الخادم لدينا تقريبًا! نحتاج فقط إلى إضافة الاختبارات. هذه المرة، تركنا الاختبارات للنهاية لأن هناك بعض الاعتبارات المهمة التي يجب مراعاتها.

تنفيذ KVServer.Command.run/1 هو إرسال الأوامر مباشرة إلى الخادم المسمى KV، الذي يتم تسجيله بواسطة تطبيق .vk. وهذا يعني أن هذا الخادم عالمي وإذا كان لدينا اختباران برسلان رسائل إليه في نفس الوقت، فسوف تتعارض اختباراتنا مع بعضها البعض (ومن المحتمل أن تفشل). نحتاج إلى اتخاذ قرار بين إجراء اختبارات وحدة معزولة ويمكن تشغيلها بشكل غير متزامن، أو كتابة اختبارات تكامل تعمل أعلى الحالة العالمية، ولكنها تمارس المكدس الكامل لتطبيقنا كما هو من المفترض أن يتم ممارسته في الإنتاج.

حتى الآن، اختبرنا نهج اختبار الوحدة. على سبيل المثال، لجعل KVServer.Command.run/1 قابلاً لاختبار كوحدة، نحتاج إلى تغيير تنفيذه بحيث لا يرسل الأوامر مباشرة إلى عملية بل يمرر خادمًا كحجية بدلاً من ذلك. وهذا يعني أننا نحتاج إلى تغيير توقيع التشغيل إلى (command, pid) وسبيدو تنفيذ الأمر etaerc: على النحو التالي:

```
bucket}, pid) do KV.Registry.create(pid, bucket) {:ok, "OK\r\n"} end
def run({:create,
```

ثم في حالة اختبار run/2، على غرار ما فعلناه في KV.Registry، على KVServer.Command تشغيل مثيل لاKVServer.Command.

لقد كان هذا هو النهج الذي اتبناه حتى الان في اختباراتنا، وله بعض الفوائد:

1. تفادي تعيين خادم معين مرتبطة بأي اسم خادم.

2. يمكننا إيقاء اختباراتنا قيد التشغيل بشكل غير متزامن، لأنه لا توجد حالة مشتركة.

ومع ذلك، فإنه يأتي مع الجانب السلبي وهو أن واجهات برمجة التطبيقات لدينا تصيب كبيرة بشكل متزايد من أجل استيعاب جميع المعلومات الخارجية.

البديل هو الاستمرار في الاعتماد على أسماء الخوادم العالمية وإجراء الاختبارات على البيانات العالمية، مع التأكد من تنظيف البيانات بين الاختبارات. في هذه الحالة، نظرًا لأن الاختبار سيعمل على تشغيل المكدس بالكامل، من خادم TCP إلى تحليل الأوامر وتشغيلها، إلى السجل وأخيرًا الوصول إلى الدلو، فإنه يصبح اختبار تكامل.

الجانب السلبي لاختبارات التكامل هو أنها قد تكون أيضًا أكبر من اختبارات الوحدة، وبالتالي يجب استخدامها بشكل أكثر اقتصاداً. على سبيل المثال، لا ينبغي لنا استخدام اختبارات التكامل لاختبار حالة هامشية في تنفيذ تحليل الأوامر.

نظرًا لأننا استخدمنا اختبارات الوحدة حتى الان، فستأخذ هذه المرة الطريق الآخر ونكتب اختبار تكامل. سيحتوي اختبار التكامل على عميل TCP يرسل الأوامر إلى خادمنا وسنؤكّد أنها تحصل على النتيجة المطلوبة.

دعنا ننفذ اختبار التكامل الخاص بنا في test/kv_server_test.exs مما هو موضح أدناه:

```
defmodule KVServerTest
  use ExUnit.Case
  setup do
    :application.stop(:kv) =:application.start(:kv) end
  @opts = %{:binary, packet: :line, active: false}
  {:ok, socket} = :gen_tcp.connect('localhost', 4040, @opts)
  send_and_recv(socket, "UNKNOWN shopping\r\n")
  assert "أمر غير معروف" == receive _ do
    _end
  end
  send_and_recv(socket, "GET /shopping\r\n")
  assert "لم يتم العثور عليه" == receive _ do
    _end
  end
  send_and_recv(socket, "CREATE /shopping egg\r\n")
  assert "تم إنشاء التسوق" == receive _ do
    _end
  end
  send_and_recv(socket, "PUT /shopping egg 3\r\n")
  assert "وضع بعض التسوق" == receive _ do
    _end
  end
  send_and_recv(socket, "GET shopping egg\r\n")
  assert "3\r\n" == receive _ do
    _end
  end
  send_and_recv(socket, "DELETE /shopping egg\r\n")
  assert "# GET" == receive _ do
    _end
  end
end
```

```

send_and_recv(socket, "") == "OK\r\n"
تأكيد " حذف بین التسوق \r\n" == send_and_recv(socket,
" حسنا "
)

assert send_and_recv(socket, "GET shopping egg\r\n") == "\r\n" assert send_and_recv(socket, "") == "OK\r\n" # بعيد سطرين GET

```

نهاية

```

do :ok = :gen_tcp.send(socket, command) {:ok, data} = :gen_tcp.recv(socket, 0, 1000)
defp send_and_recv(socket, command)

```

بيانات
نهاية
نهاية

يتحقق اختبار التكامل لدينا من جميع تفاعلات الخادم، بما في ذلك الأوامر غير المعروفة والأخطاء غير المكتشفة. تجدر الإشارة إلى أنه، كما هو الحال مع جداول ETS والعمليات المرتبطة، ليست هناك حاجة لإغلاق المقاييس. بمجرد خروج عملية الاختبار، يتم إغلاق المقاييس تلقائيا.

هذه المرة، نظرًا لأن اختبارنا يعتمد على بيانات عالمية، لم نعط `ExUnit.Case` علاوة على ذلك، لضمان أن يكون اختبارنا دائمًا في حالة نظيفة، نوقف ونبدأ تطبيق `vk`: قبل كل اختبار، في الواقع، يؤدي إيقاف تطبيق `vk` إلى طباعة تحذير على المحطة الطرفية:

```
deppots: تم الخروج من تطبيق vk [info] 10:698:12:18
```

إذا أردنا ذلك، يمكننا تجنب طباعة هذا التحذير عن طريق إيقاف تشغيل `logger_error!` وتشغيله في إعداد الاختبار:

```

Logger.remove_backend(:console) إعداد
(vk:)trats = تطبيق ko:(vk:pots) تطبيق
Logger.add_backend(:console, flush: true)
نعم: نهاية

```

باستخدام اختبار التكامل البسيط هذا، نبدأ في فهم سبب بطلة اختبارات التكامل. لا يمكن تشغيل هذا الاختبار بشكل غير متزامن فحسب، بل يتطلب أيضًا الإعداد المكلف لإيقاف وتشغيل تطبيق `vk`:

في نهاية المطاف، الأمر متروح لك وفريقك لمعرفة أفضل استراتيجية اختبار لتطبيقاتك. تحتاج إلى تحقيق التوازن بين جودة التعليمات البرمجية والنقة ووقت تشغيل مجموعة الاختبار. على سبيل المثال، قد نبدأ باختبار الخادم فقط باستخدام اختبارات التكامل، ولكن إذا استمر الخادم في النمو في الإصدارات المستقبلية، أو أصبح جزءًا من التطبيق مع وجود أخطاء متكررة، فمن المهم التفكير في تفكيكه وكتابة اختبارات وحدات أكثر كثافة لا تحمل وزن اختبار التكامل.

أنا شخصياً أخاطر في جانب اختبارات الوحدة، وأجري اختبارات التكامل فقط كاختبارات دخان لضمان عمل الهيكل الأساسي للنظام.

في الفصل التالي، سنقوم أخيراً بتوزيع نظامنا عن طريق إضافة آلية توجيه الدلو. وسنتعلم أيضًا عن تكوين التطبيق.

10. التكوين والموزعة المهام

`toc.html %} تشمل`

في هذا الفصل الأخير، سنعود إلى تطبيق `vk` ونضيف طبقة توجيه تسمح لنا بتوزيع الطلبات بين العقد استناداً إلى اسم الدلو.

ستستقبل طبقة التوجيه جدول توجيه بالتنسيق التالي:

```
m, :"foo@computer-name"}, {?n..?z, :"bar@computer-name"}]
[{:?a..?
```

سيتحقق جهاز التوجيه من البايت الأول من اسم الدلو مقابل الجدول ويرسله إلى العقدة المناسبة بناءً على ذلك. على سبيل المثال، سيتم إرسال الدلو الذي يبدأ بالحرف ("a") يمثل رمز Unicode للحرف إلى العقدة `foo@computer-name`.

إذا كانت الإدخالات المطابقة تشير إلى العقدة التي تقوم بتقييم الطلب، فستكون قد انتهينا من التوجيه، وستقوم هذه العقدة بتنفيذ العملية المطلوبة. إذا كانت الإدخالات المطابقة تشير إلى عقدة مختلفة، فسنمرر الطلب إلى هذه العقدة، والتي ستنتظر في جدول التوجيه الخاص بها (والذي قد يكون مختلفاً عن الجدول الموجود في العقدة الأولى) وتتصرف وفقاً لذلك. إذا لم تتطابق أي إدخالات، فسيتم رفع خطأ.

قد تتساءل لماذا نخبر العقدة التي نجدها في جدول التوجيه لدينا ببساطة بتنفيذ العملية المطلوبة مباشرةً، بل نمرر بدلاً من ذلك طلب التوجيه إلى تلك العقدة لمعالجتها. وفي حين أن جدول التوجيه البسيط مثل الجدول أعلاه قد يكون مشتركاً بشكل معقول بين جميع العقد، فإن تمرير طلب التوجيه بهذه الطريقة يجعل من الأسهل بكثير تقسيم جدول التوجيه إلى أجزاء أصغر مع نمو تطبيقنا. ربما في مرحلة ما، سيكون `foo@computer-name` مسؤولاً فقط عن طلبات دلو التوجيه، وسيتم إرسال الدلاء التي يتعامل معها إلى عقد مختلف.

بهذه الطريقة، لا يحتاج `bar@computer-name` إلى معرفة أي شيء عن هذا التغيير.

ملحوظة: سوف نستخدم عقدتين في نفس الجهاز طوال هذا الفصل. أنت حر في استخدام جهازين مختلفين (أو أكثر) في نفس الشبكة ولكنك بحاجة إلى القيام بعض الأعمال التحضيرية. أولاً وقبل كل شيء، تحتاج إلى التأكد من أن جمع الأجهزة لديها ملف `epmd`-بنفس القيمة تمامًا. ثانياً، تحتاج إلى ضمان أن `erlang.cookie` يتم التشغيل على منفذ غير محظوظ (يمكنك تشغيل `epmd -d` للحصول على معلومات التصحيح).

ثالثاً، إذا كنت ترغب في معرفة المزيد عن التوزيع بشكل عام، فتحن نوصيك بهذا الفصل الرائع من [Distribunomicon](#).

أول كود موزع لدينا 10.1

تتضمن [Elixir](#) مراافق لربط العقد وتبادل المعلومات بينها. في الواقع، نستخدم نفس مفاهيم العمليات وتمرير الرسائل واستلام الرسائل عند العمل في بيئة موزعة لأن عمليات [Elixir](#) شفافة الموقع. وهذا يعني أنه عند إرسال رسالة، لا يهم ما إذا كانت عملية المتنقل على نفس العقدة أو على عقدة أخرى، فستكون الآلة الافتراضية قادرة على تسليم الرسالة في كلتا الحالتين.

لتشغيل التعليمات البرمجية الموزعة، تحتاج إلى بدء تشغيل الجهاز الافتراضي باسم. يمكن أن يكون الاسم قصيراً (عند وجوده في نفس الشبكة) أو طويلاً (يطلب عنوان الكمبيوتر الكامل)، لبدأ جلسة `iEx` الجديدة:

```
$ iex --sname foo
```

يمكنك الآن رؤية أن المطالبة مختلفة قليلاً وتنظر اسم العقدة متى باسما الكمبيوتر:

[الإكسبر التفاعلي](#) - اضغط على `Ctrl+C` للخروج (اكتب `ENTER` للحصول على المساعدة)

اسم جهاز الكمبيوتر الخاص بي هو `jalda` وأرى `foo@jv` في المثال أعلاه، ولكنك ستحصل على نتيجة مختلفة. سنسخدم `jv@computer-name` في الأمثلة التالية و يجب عليك تحريرها وفقاً لذلك عند تجربة الكود.

دعنا نقوم بتعريف وحدة تسمى `Hello` في هذا الغلاف:

```
amerjiba@jalda:~/Documents$ iex> defmodule Hello do
...>   def world, do: IO.puts "hello world"
...> end
```

إذا كان لديك جهاز كمبيوتر آخر على نفس الشبكة مثبت عليه كل من [Erlang](#) و [Elixir](#)، فيمكنك بدء تشغيل غلاف آخر عليه. إذا لم يكن الأمر كذلك، فيمكنك ببساطة بدء جلسة `iEx` أخرى في محطة طرفية أخرى. في كلتا الحالتين، أعطها الاسم المختصر `bar`

```
$ iex --sname bar
```

لاحظ أنه داخل جلسة Ex الجديدة هذه، لا يمكننا الوصول إلى: Hello.world/0:

```
iex> bar
               مرحبًا بالعالم<iex>
Hello.world/0: ** دالة غير محددة : (UndefinedFunctionError)
               مرحبًا بالعالم()
```

ومع ذلك، يمكننا إنشاء عملية جديدة على foo@computer-name! من bar@computer-name (حيث يكون name@computer-name هو الذي تراه محلياً):

```
iex> Node.spawn_link :"foo@computer-name", fn -> Hello.world end #PID<9014.59.0>
```

مرحبًا بالعالم

لقد قام Elixir بإنشاء عملية على عقدة أخرى وأعاد معرف العملية الخاص بها. ثم قام بتنفيذ الكود على العقدة الأخرى حيث توجد وظيفة Hello.world/0 واستدعى تلك الوظيفة. لاحظ أن نتيجة "hello world" تمت طباعتها على العقدة الحالية bar وليس على foo. بعبارة أخرى، تم إرسال الرسالة المراد طباعتها من foo إلى bar. يحدث هذا لأن العملية التي تم إنشاؤها على العقدة الأخرى (foo) لا تزال لديها قائد المجموعة للعقدة الحالية (bar).

لقد تحدثنا بشكل مختصر عن قادة المجموعات في الفصل 10.

يمكننا إرسال واستقبال الرسائل من معرف العملية الذي تم إرجاعه بواسطة Node.spawn_link/2 كالمعتاد. دعنا نحاول مثلاً سريعاً للعبة بينج بونج:

```
iex> pid = Node.spawn_link :"foo@computer-name", fn -> ...>
...>
               إرسال العميل: ... gnop< النهائي:>
               إرسال< النهائي:>
               حسنًا: بونج: ...>
...>
               النهائي:>
               معرف العملية< #PID<0.95.4109>>
```

من خلال استكشافنا السريع، يمكننا أن نستنتج أنه ينبغي علينا ببساطة استخدام Node.spawn_link/2 لإنشاء عمليات على عقدة بعيدة في كل مرة نحتاج فيها إلى إجراء عملية حساسية موزعة. ومع ذلك، تعلمنا طوال هذا الدليل أنه ينبغي تجنب إنشاء عمليات خارج أشجار الإشراف إذا أمكن، لذا نحتاج إلى البحث عن خيارات أخرى.

هناك ثلاثة بدائل أفضل لـ Node.spawn_link/2 يمكننا استخدامها في تنفيذنا:

1. يمكننا استخدام Erlang: cpr: في واحدة لتنفيذ الوظائف على عقدة بعيدة. داخل غلاف bar@computer-name، يمكنك استدعاء: ![])-retupmoc@oof";)llac.cpr: .

"hello world"Hello,:world,

2. يمكننا تشغيل خادم على العقدة الأخرى وإرسال طلبات إلى تلك العقدة عبر واجهة برمجة تطبيقات GenServer. على سبيل المثال، يمكنك الاتصال بخادم بعيد باسم باستخدام GenServer.call({name, node}, arg)

3. يمكننا استخدام المهام التي تعلمناها في الفصل السابق، حيث يمكن إنشاؤها على كلا الجهازين المحليتين والعقد البعيدة

الخيارات المذكورة أعلاه لها خصائص مختلفة. فكل من: cpr: واستخدام GenServer من شأنهما تسلسل طلباتك على خادم واحد، بينما يتم تشغيل المهام بشكل غير متزامن على العقد البعيدة، مع كون نقطة التسلسل الوحيدة هي التفريخ الذي يقوم به المشرف.

بالنسبة لطبيقة التوجيه الخاصة بنا، سنستخدم المهام، ولكن لا تتردد في استكشاف البديل الآخر أيضاً.

انتظار/متزامن غير 10.2

حتى الآن استكشنا المهام التي يتم تشغيلها بشكل منعزل، دون مراعاة قيمة إرجاعها، ومع ذلك، من المفيد أحياناً تشغيل مهمة لحساب قيمة وقراءة نتائجها لاحقاً. ولهذا، توفر المهام `async/await` لأنها تُنطَّلق في سياق التحليق.

```
= Task.async(fn -> compute_something_expensive end) res = compute_something_else() res + Task.await(task) المهمة
```

يتوفر `await` `Task.Supervisor.async` كطريقة بسيطة للغابة لحساب القيم في نفس الوقت. ليس هذا فحسب، بل يمكن أيضًا استخدام `Task.Supervisor.start_child/2` بدلاً من `Task.Supervisor.start_child/2` لاستدعاء `Task.Supervisor.async/2` فقط إلى استدعاء `Task.Supervisor.start_child/2` بدلًا من `Task.Supervisor.start_child/2`.
الذي استخدمناه في الفصول السابقة. تحتاج فقط إلى قراءة النتيجة `Task.Supervisor.start_child/2` لفهم ذلك.

10.3 المهام الموزعة

المهام الموزعة هي نفس المهام الخاضعة للإشراف تماماً. الفرق الوحيد هو أننا نمرر اسم العقدة عند إنشاء المهمة على المشرف. افتح `kv/supervisor.ex`:
نصف مثقباً للمهمة آل ، الشحنة:

مشرف(مشرف المهمة، [[الاسم: مهام توجيه، KV]])

لأنه، دعونا نبدأ بعقدتين مسماتين مرة أخرى، ولكن داخل تطبيق: vk

```
$ iex --sname foo -S mix $ iex --sname bar -S mix
```

من داخلا، bar@computer-name، عبد المشفى، العقدة الأخرى، إنشاء مهمة ملائمة على يمكننا الآذن:

```
...> ...> الذهاب...>  
#Reference<0.0.0.400> iex> Task.await(task) المراجع: Task{pid: #PID<12467.88.0>.  
  
{:ok, {"foo@computer-name"}  
...>
```

إن مهمتنا الموزعة الأولى بسيطة: فهي ببساطة تحصل على اسم العقدة التي تعمل عليها المهمة. ومع توفر هذه المعرفة، فلنبدأ أخيراً في كتابة كود التوجيه.

طبقه التوجه 10.4

قم بإنشاء ملف في `lib/kv/router.ex` يحتوي على التالية:

إلغاء وحدة التوجيه KV.

رسال الطلبات `args` و `mod` و `fun` المحددة إلى العقدة المناسبة استناداً إلى `bucket`.

تحتاج إلى إدخال البايت الأول من الملف الثنائي

```
def route(bucket, mod, fun, args) do
    first = :binary.first(bucket)
```

#حاول العثور على إدخال في الجدول أو رفعه

الإصدارات Elixir توثيق

```

enum Enum.find(table, fn {enum, node} ->
  elem(entry, 1) == node() do apply(mod, fun, args)
    # إذا كانت عقدة الإدخال هي العقدة الحالية
    if
      آخر
      meleSup = {KV.RouterTasks,
        fn -> .pus(غير متزامن، المهمة المنشف)
        KV.Router.route(bucket, mod, fun, args)
      }
      Task.await() |_(نهائية)
    else
      نهائية
    end
  end
end

@doc """
جدول التوجيه.
"""

جدول تعريف
[{:a,:m,:"foo@computer-name"},{:n,:z,:"bar@computer-name"}] end
# استبدل اسم الكمبيوتر باسم جهاز المحلب
#({inspect table}) |_(نهائية)

@doc """
جدول التوجيه.
"""

نهائية

```

دعنا نكتب اختباراً للتأكد من عمل جهاز التوجيه الخاص بنا. أنشأ ملفاً باسم `test/kv/router_test.exs` يحتوي على:

```

defmodule KV.RouterTest
  use ExUnit.Case, async: true

  name" تأكيد KV.Router.route(" hello ", Kernel, :node, []) == :"foo@computer-name"
  " تأكيد KV.Router.route("world", Kernel, :node, []) == :"bar@computer-"

  """
  اختبار "الرفع عند الإدخالات غير المعرفة"
  """
  fn -> raise RuntimeError, ~r/
  لم يتمكن من العثور على الإدخال/. |_(نهائية)
  KV.Router.route(<>, Kernel, :node, [])
end

نهائية
نهائية
نهائية

```

يستدعي الاختبار الأول ببساطة `Kernel.node/0` الذي يعيد اسم العقدة الحالية، استناداً إلى أسماء الدلو "hello" و "world". وفقاً لجدول التوجيه الخاص بنا حتى الآن، يجب أن نحصل على `foo@computer-name` وإجابات على التوالي.

الاختبار الثاني يتحقق فقط من أن الكود يتغير الإدخالات غير المعرفة.

لتشغيل الاختبار الأول، نحتاج إلى تشغيل عقدتين. لنعد تشغيل العقدة المسماة `bar`، والتي سيتم استخدامها بواسطة الاختبارات:

```
$ iex --sname bar -S mix
```

والآن قم بتشغيل الاختبارات باستخدام:

```
$ elixir --sname foo -S مزدوج اختبار
```

يجب أن ينجح اختبارنا. ممتازا!

10.5 اختبار المرشحات والعلامات

على الرغم من نجاح اختباراتنا، إلا أن بنية الاختبار لدينا أصبحت أكثر تعقيداً. على وجه الخصوص، يؤدي تشغيل الاختبارات باستخدام اختبار مخلط فقط إلى حدوث أخطاء في مجموعتنا، نظراً لأن اختبارنا يتطلب اتصالاً بعقدة أخرى.

لحسن الحظ، يأتي `ExUnit` مزوداً بميزة لوضع علامات على الاختبارات، مما يسمح لنا بتشغيل مكالمات استرجاعية محددة أو حتى تصفية الاختبارات بالكامل استناداً إلى تلك العلامات.

كل ما نحتاج إلى فعله لوضع علامة على الاختبار هو ببساطة استدعاء `test_kv/routest_test.exs` قبل اسم الاختبار، بالإضافة إلى `@tag` التي نضيف علامة:

"@tag :distributed test" توجيه الطلبات عبر العقد

كتابة `@tag distributor: true` يعادل كتابة `@tag :distributed`

بعد وضع علامة الاختبار بشكل صحيح، يمكننا الآن التتحقق مما إذا كانت العقدة نشطة على الشبكة، وإذا لم تكن كذلك، فيمكننا استبعاد جميع الاختبارات الموزعة. افتح `test_helper.exs` وادخل تطبيق `vk` وأضف ما يلي:

= استبعاد

إذا كان `Node.alive?`، فإن فافعل: []، وإلا: [موقع: صحيح]

(استبعاد: استبعاد) `trats.tinUxE`

قم الآن بتشغيل الاختبارات باستخدام اختبار المزدوج:

اختبار المزدوج

باستثناء العلامات: [توزيع: صحيح]

.....

تم الانتهاء في 0.1 ثانية (0.01 ثانية عند التحميل، 0.01 ثانية عند الاختبارات) 7 اختبارات، 0 فشل

هذه المرة نجحت جميع الاختبارات وحدرنا `ExUnit` من استبعاد الاختبارات الموزعة، إذا قمت بتشغيل الاختبارات باستخدام `elixir --sname foo -S mix test`. فيجب تشغيل اختبار `bar@computer-name` واحد واجتيازه بنجاح طالما أن العقدة متاحة.

يسمح لنا أمر `mix test` أيضاً بتضمين واستبعاد العلامات بشكل ديناميكي. على سبيل المثال، يمكننا تشغيل الاختبارات الموزعة بغض النظر عن القيمة المحددة في `--include distributor` أو `--exclude`. يمكننا أيضاً تمرير `--only` لتشغيل علامة معينة من سطر الأوامر. أخيراً، يمكن استخدام `--only` لتشغيل الاختبارات التي تحتوي على علامة معينة فقط:

`elixir --sname foo -S mix test --` يتم توزيعه فقط

يمكنك قراءة المزيد حول المرشحات والعلامات والعلامات الافتراضية في [`ونائق وحدة`](#).

10.6 بيئة التطبيق والتكتوين

حتى الآن قمنا بتمييز جدول التوجيه في وحدة KV.Router. ومع ذلك، نود أن نجعل الجدول ديناميكيًا. وهذا لا يسمح لنا فقط بتكوين التطوير/الاختبار/الإنتاج، بل يسمح أيضًا بتشغيل عقد مختلفة بإدخالات مختلفة في جدول التوجيه. هناك ميزة في OTP تقوم بذلك بالضبط: بيئة التطبيق.

يحتوي كل تطبيق على بيئة تخزن التكتوين الخاص بالتطبيق حسب المفتاح. على سبيل المثال، يمكننا تخزين جدول التوجيه في بيئة التطبيق: `vk`. وإعطائه قيمة افتراضية والسماح للتطبيقات الأخرى بتغيير الجدول حسب الحاجة.

افتح `application.exs` وقم بتغيير الدالة `0` لإرجاع ما يلي:

```
def do _ do
    #[{"path": "/api/v1/todos", "method": "GET", "handler": "TodosHandler", "response": "TodosResponse"}, {"path": "/api/v1/todos/:id", "method": "PUT", "handler": "TodosHandler", "response": "TodosResponse"}]
end
```

التطبيقات: `[],` [البيئة: [جدول التوجيه: `[],` الوضع: `[],` النهاية]]

لقد أضفنا مفتاحًا جديداً: `vne` إلى التطبيق. وهو يعيد بيئة التطبيق الافتراضية، والتي تحتوي على إدخال المفتاح: `elbat_gnituor` وقيمة قائمة فارغة. ومن المنطقي أن يتم شحن بيئة التطبيق بجدول فارغ، حيث يعتمد جدول التوجيه المحدد على بيئة الاختبار/النشر.

لكي نتمكن من استخدام بيئة التطبيق في الكود الخاص بنا، نحتاج فقط إلى استبدال `KV.table` بالتعريف أدناه:

```
@doc """
# جدول التوجيه.
"""

def __init__(self):
    self.vne = [{"path": "/api/v1/todos", "method": "GET", "handler": "TodosHandler", "response": "TodosResponse"}, {"path": "/api/v1/todos/:id", "method": "PUT", "handler": "TodosHandler", "response": "TodosResponse"}]
```

تطبيقات: `[],` [النهاية (elbat_gnituor:, ,vne_teg)]

نستخدم `Application.get_env/2` لقراءة الإدخال الخاص `:routing_table` بـ `vk`. يمكنك العثور على مزيد من المعلومات والوظائف الأخرى للتعامل مع بيئة التطبيق في وحدة التطبيق.

نظرًا لأن جدول التوجيه الخاص بنا أصبح فارغاً الآن، فيجب أن يفشل اختبارنا الموزع. أعد تشغيل التطبيقات وأعد تشغيل الاختبارات لمعرفة الفشل:

```
iex -S mix $ iex --sname bar -S mix $ elixir --sname foo -S mix test --
```

الشيء المثير للاهتمام حول بيئة التطبيق هو أنه يمكن تكوينها ليس فقط للتطبيق الحالي، ولكن لجميع التطبيقات. يتم إجراء هذا التكوين بواسطة ملف `config/config.exs` على سبيل المثال، يمكننا تكوين موجه `Ex` الافتراضي إلى قيمة أخرى. ما عليك سوى فتح `apps/kv/config/config.exs` وإضافة ما يلى إلى النهاية:

```
iex. default_prompt: ">>>"
```

ابدأ تشغيل `iex -S mix` ويتحقق أن ترى أن موجه `Ex` قد تغير.

يعني هذا أنه يمكننا تكوين `elbat_gnituor`: `elbat_gnituor: config/config.exs` أيًضاً:

```
# استبدل اسم الكمبيوتر بعقد الجهاز المحلية لديك.
config :kv, :routing_table, #
```

```
[{:a..?m, :"foo@computer-name"}, {:n..?z, :"bar@computer-name"}]
```

أعد تشغيل العقد وقم بتشغيل الاختبارات الموزعة مرة أخرى. الآن يجب أن تنجح جميعها.

يحتوي كل تطبيق على ملف config.exs خاص به ولا يتم مشاركته بأي شكل من الأشكال. يمكن أيضًا تعين التكوين لكل بيئة. اقرأ محتويات ملف التكوين لنطبيق `vk` لمزيد من المعلومات حول كيفية القيام بذلك.

نظرًا لعدم مشاركة ملفات التكوين، فإذا قمت بتشغيل الاختبارات من جذر المظلة، فسوف تفشل لأن التكوين الذي أضفناه للتو إلى `vk` غير متاح هناك. ومع ذلك، إذا فتحت config.exs في المظلة، فستجد تعليمات حول كيفية استيراد ملفات التكوين من التطبيقات الفرعية. ما عليك سوى استدعاء:

```
astirad التكوين config.exs ..//apps/kv/config/config.exs
```

يقبل أمر `mix run config`-- والتي تسمح بتقديم ملفات التكوين عند الطلب. يمكن استخدام هذا لبدء تشغيل عقد مختلف، كل منها بتكونتها المحددة (على سبيل المثال، جداول التوجيه المختلفة).

بشكل عام، فإن القدرة المضمنة على تكوين التطبيقات وحقيقة أنها قمنا ببناء برنامجنا كتطبيق شامل يمنحنا الكثير من الخيارات عند نشر البرنامج، يمكننا:

- نشر تطبيق المظلة على عقدة تعمل كخادم TCP وت تخزين القيمة الرئيسية

- نشر تطبيق revres_vk ليعمل فقط كخادم TCP طالما أن جدول التوجيه يشير فقط إلى العقد الأخرى

- نشر تطبيق `vk`: فقط عندما نريد أن نعمل العقدة فقط كتخزين (بدون وصول TCP)

مع إضافة المزيد من التطبيقات في المستقبلي، يمكننا الاستمرار في التحكم في نشرنا بنفس مستوى التفصيل، واختيار التطبيقات التي سيتم نقلها إلى الإنتاج وفقًا لتكونتها. يمكننا أيضًا التفكير في إنشاء إصدارات متعددة باستخدام أدوات مثل `exrm` والتي تقوم بتجميع التطبيقات والتكونيات المختارة، بما في ذلك تكتبات Erlang وإلساير، حتى تتمكن من نشر التطبيق حتى إذا لم يتم تعيين وقت التشغيل مسبقاً على النظام المستهدف.

أخيرًا، تعلمنا بعض الأشياء الجديدة في هذا الفصل، ويمكن تطبيقها على تطبيق revres_vk: أيضًا. سنترك الخطوات التالية كتمرين:

- تغيير تطبيق revres_vk: لقراءة المنفذ من بيئة التطبيق الخاصة به بدلاً من استخدام القيمة المبرمجة 4040

- قم بتغيير تكوين تطبيق revres_vk لاستخدام وظيفة التوجيه بدلاً من الإرسال مباشرةً إلى KV.Registry المحلي، بالنسبة لاختبارات revres_vk. يمكنك جعل جدول التوجيه يشير ببساطة إلى العقدة الحالية نفسها

10.7 تلخيص

في هذا الفصل قمنا ببناء جهاز توجيه بسيط كوسيلة لاستكشاف الميزات الموزعة لبرنامج Erlang والـOTP، وتعلمنا كيفية تكوين جدول التوجيه الخاص به. هذا هو الفصل الأخير في دليل Mix and OTP.

خلال الدليل، قمنا ببناء مخزن قيم مفاتحة موزع بسيط للغاية كفرصة لاستكشاف العديد من البنية مثل الخوادم العامة ومديري الأحداث والمشرفين والمهام والوكلاء والتطبيقات والمزيد. ليس هذا فحسب، بل لقد كتبنا اختبارات للتطبيق بالكامل، ونعرفنا على `ExUnit`، وتعلمنا كيفية استخدام أدوات بناء Mix لإنجاز مجموعة واسعة من المهام.

إذا كنت تبحث عن متجر قيم مفاتحة موزع لاستخدامه في الإنتاج، فيجب عليك بالتأكيد البحث في `Riak`، الذي يعمل أيضًا في VM. يتم تكرار الدلاء لتجنب فقد البيانات، وبدلاً من جهاز التوجيه، يستخدمون [التجزئة المتسلقة](#) لربط دلو بعقدة. تساعد خوارزمية التجزئة المتسلقة في تقليل كمية البيانات التي يلزم نقلها عند إضافة عقد جديدة لتخزين الدلاء إلى البنية الأساسية الخاصة بك.

البرمجة الفوقيّة في Elixir

اقتباس وإلغاء الاقتباس

toc.html %} تشمل

يمكن تمثيل برنامج Elixir من خلال هياكل البيانات الخاصة به. في هذا الفصل، سنتعلم كيف تبدو هذه الهياكل وكيفية تكوينها. المفاهيم التي تعلمناها في هذا الفصل هي اللبنات الأساسية لوحدات الماكرو، والتي سنلقي نظرة أعمق عليها في الفصل التالي.

1.1. الاقتباس

إن اللبنة الأساسية لبرنامج Elixir هي مجموعة مكونة من ثلاثة عناصر. على سبيل المثال، يتم تمثيل استدعاء الدالة `(1, 2, 3) sum` داخلًا على النحو التالي:

```
[]. [1, 2, 3] {مجموع.
```

يمكنك الحصول على تمثيل أي تعبير باستخدام ماكرو الاقتباس:

```
اقتباس do: sum(1, 2, 3) {sum, [], [1, 2, 3]}>
```

العنصر الأول هو اسم الوظيفة، والثاني هو قائمة الكلمات الرئيسية التي تحتوي على البيانات الوصفية والثالث هو قائمة الوسائط.

يتم أيضًا تمثيل المشغلات على هيئةمجموعات من هذا القبيل:

```
اقتباس Kernel]. [1, 2] do: 1 + 2 {:+, [السياق: Elixir, الاستيراد:]>
```

حتى الخريطة يتم تمثيلها على أنها استدعاء لـ `%{}`:

```
اقتباس :%{1 => 2}, [], [{1, 2}]}>
```

يتم أيضًا تمثيل المتغيرات باستخدام مثل هذه الثلاثيات، باستثناء أن العنصر الأخير هو ذرة، بدلاً من القائمة:

```
اقتباس do: x :{x, []}, Elixir>
```

عند اقتباس تعابيرات أكثر تعقيدًا، يمكننا أن نرى أن الكود يتم تمثيله في مثل هذه الثنائيات، والتي غالباً ما تكون متداخلة داخل بعضها البعض في بنية تشبه الشجرة. العديد من اللغات تطلق على مثل هذه التمثلات اسم شجرة بناء الجملة المجردة (AST). يطلق عليها Elixir تعابيرات مقتبسة:

```
اقتباس افعل: Macro.to_string/1 اقتباس Kernel], [2, 3]}. [4] {استيراد: Elixir, السياق: sum(1, 2 + 3, 4) {sum, [], [1, {:+, [2, 3]}, 4]}>
```

في بعض الأحيان، عند العمل مع التعابيرات المقتبسة، قد يكون من المفيد استعادة تمثيل الكود النصي. يمكن القيام بذلك باستخدام `Macro.to_string/1`:

```
اقتباس gnirts_ot.orcaMiex> do: sum(1, 2 + 3, 4)) "sum(1, 2 + 3, 4)">
```

بشكل عام، يتم تنظيم المجموعات أعلاه وفقًا للتسلسلي التالي:

{مجموعة | ذرة، قائمة، قائمة | ذرة}

• العنصر الأول هو ذرة أو مجموعة أخرى في نفس التمثيل؛

• العنصر الثاني هو قائمة الكلمات الرئيسية التي تحتوي على البيانات الوصفية، مثل الأرقام والسيارات؛

• العنصر الثالث هو إما قائمة من الوسائل لاستدعاء الوظيفة أو ذرة. عندما يكون هذا العنصر ذرة، فإنه

يعني أن المجموعة تمثل متغيراً.

بالإضافة إلى المجموعة المحددة أعلاه، هناك خمسة أحرف، والتي عند اقتباسها، تعيد نفسها (وليس مجموعة).

وهم:

مجموع	=> الدارات
1.0	=> الأرقام
	=> القوائم
	[1, 2] => السلاسل
	<=> مجموعات تحتوي على عنصرين {فتح، قيمة} => سلاسل {فتح، قيمة}

تحتوي أغلب أكواد Elixir على ترجمة مباشرة للتعبير المقتبس الأساسي الخاص بها. نوصيك بتجربة عينات أكواد مختلفة ومعرفة النتائج. على سبيل المثال، إلى أي مدى يمتد `String.upcase("foo")`؟

لقد تعلمنا أيضًا أن `(true, do: :this, else: :that) if true do :this else :that end` هي نفس `true do :this end`. كيف ينطبق هذا التأكيد على التعبيرات المقتبسة؟

1.2 عدم الاقتباس

تعلق الاقتباسات باسترداد التمثيل الداخلي لجزء معين من التعليمات البرمجية. ومع ذلك، قد يكون من الضروري في بعض الأحيان حقن جزء معين آخر من التعليمات البرمجية داخل التمثيل الذي نريد استرداده.

على سبيل المثال، تخيل أن لديك متغيراً رقمياً يحتوي على الرقم الذي تريد حقنه داخل تعبير مقتبس.

```
= 13 رقمielex>
(اقتباس من: 11 + رقمgnirts_ot.orcaMiex>
"11 +
```

هذا ليس ما أردناه، حيث لم يتم حقن قيمة متغير الرقم وتم وضع علامة اقتباس بين علامتي الاقتباس في التعبير. ولحقن قيمة متغير الرقم، يجب استخدام علامة الاقتباس داخل التمثيل المقتبس:

```
= 13 رقمielex>
"11 + 13" (اقتباس من: إزالة الاقتباس(رقم))gnirts_ot.orcaMiex>
```

يمكن أيضًا استخدام علامة الاقتباس لحقن أسماء الوظائف:

```
= :hello مرحielex>
(اقتباس إلى: إلغاء الاقتباس(متعدة)(العالم)) "مرحبا:(العالم)"gnirts_ot.orcaMiex>
```

في بعض الحالات، قد يكون من الضروري حقن العديد من القيم داخل القائمة. على سبيل المثال، تخيل أن لديك قائمة تحتوي على [1, 2, 6] ونريد حقن [3, 4, 5] فيها. لن يؤدي استخدام علامات الاقتباس إلى الحصول على النتيجة المطلوبة:

```
do: [1, 2, unquote(inner), 6]) "[1, 2, [3, 4, 5], 6]" (اقتباس gnirts_ot.orcaMiex> inner = [3, 4, 5] ie>
```

هذا هو الوقت الذي يصبح فيه `unquote_splicing` مفيداً:

```
do: [1, 2, unquote_splicing(inner), 6]) "[1, 2, 3, 4, 5, 6]" (اقتباس gnirts_ot.orcaMiex> inner = [3, 4, 5] iex>
```

إن إزالة علامات الاقتباس مفيدة للغاية عند العمل مع وحدات الماكرو، عند كتابة وحدات الماكرو، يمكن المطورو من تلقي أجزاء من التعليمات البرمجية وحقنها داخل أجزاء أخرى من التعليمات البرمجية، والتي يمكن استخدامها لتحويل التعليمات البرمجية أو كتابة التعليمات البرمجية التي تولد التعليمات البرمجية أثناء التجميع.

1.3 الهروب

كما رأينا في بداية هذا الفصل، فإن بعض القيم فقط هي تعبيرات مقتبسة صالحة في Elixir على سبيل المثال، الخريطة ليست تعبيراً مقتبساً صالحاً. وكذلك الأمر بالنسبة لمجموعة مكونة من أربعة عناصر. ومع ذلك، يمكن التعبير عن مثل هذه القيم كتعبير مقتبس:

```
%{1 => 2} %{[], [{1, 2}]} (اقتباس iex>
```

في بعض الحالات، قد تحتاج إلى حقن مثل هذه القيم في التعبيرات المقتبسة. للقيام بذلك، نحتاج أولاً إلى إفلات هذه القيم في التعبيرات المقتبسة بمساعدة Macro.escape/1:

```
{%{مرحبا: العالم}, [%{مرحبا: الخريطة}, []]} (الخريطة iex>
```

تتلقي وحدات الماكرو تعبيرات مقتبسة ويجب أن تعيد التعبيرات المقتبسة. ومع ذلك، في بعض الأحيان أثناء تنفيذ وحدة الماكرو، قد تحتاج إلى العمل بالقيم وسيكون التمييز بين القيم والتعبيرات المقتبسة مطلوباً.

عبارة أخرى، من المهم التمييز بين قيمة العادلة (مثل القائمة أو الخريطة أو العملية أو المرجع وما إلى ذلك) والتعبير المقتبس. تحتوي بعض القيم، مثل الأعداد الصحيحة والدارات والسلسل، على تعبير مقتبس يساوي القيمة نفسها. تحتاج قيم أخرى، مثل الخرائط، إلى تحويل صريح. أخيراً، لا يمكن تحويل قيم مثل الوظائف والمراجع إلى تعبير مقتبس على الإطلاق.

يمكنك قراءة المزيد حول الاقتباس وإلغاء الاقتباس في وحدة Kernel.SpecialForms</docs/stable/elixir/Kernel.SpecialForms.html>. يمكن العثور على وثائق Macro</docs/stable/elixir/Macro.html> Macro.escape/1 والوظائف الأخرى المتعلقة بالتعبيرات المقتبسة في وحدة Macro.escape/1.

في هذه المقدمة وضعنا الأساس لكتابه أول ماكرو لنا أخيراً، لذا دعونا ننتقل إلى الفصل التالي.

2 ماكرو

toc.html %} تشمل

يمكن تعريف وحدات الماكرو في Elixir باستخدام defmacro/2.

في هذا الفصل، سنتستخدم الملفات بدلاً من تشغيل عينات التعليمات البرمجية في Elixir، وذلك لأن عينات التعليمات البرمجية مستمدت على أسطر متعددة من التعليمات البرمجية وكتابتها كلها في Elixir قد يكون غير منتج. يجب أن تكون قادرًا على تشغيل عينات التعليمات البرمجية عن طريق حفظها في ملف macros.exs أو macros.ex وتشغيلها باستخدام elixir macros.exs.

2.1 الماكرو الأول لدينا

من أجل فهم أفضل لكيفية عمل وحدات الماكرو، دعنا ننشئ وحدة جديدة حيث سنقوم بتنفيذ ما لم، والذي يقوم بالعكس من `fa`، كماكرو وكدالة:

```
ما لم يتم القيام بذلك defmodule
(جملة، تعبر عن if! جملة، افعل: تعبر عن nufe defselnu_sse
```

```
(جملة، تعبر عن orcamdefmacro defselnu_sse
إلغاء الاقتباس (جملة)، افعل: إلغاء الاقتباس (تعبر عن))
```

نهاية
نهاية
نهاية

تستقبل الدالة الوسائط وتمررها إلى `azoom` مع ذلك، كما تعلمنا في الفصل السابق، ستنطبق الماكرو التعبيرات المقتبسة، وتحققها في الاقتباس، وأخيراً تعيد تعبيراً مقتبساً آخر.

لنبدأ `iex` بالوحدة أعلاه:

```
$ iex macros.exs
```

واللعب مع تلك التعريفات:

```
ما لم يكن ذلك صحيحًا، فإن IO.puts "لا ينبغي طباعة هذا أبداً" لا شيء<require unless=macro_unless>
```

```
ما لم يكن ذلك صحيحًا، فإن IO.puts "لا ينبغي طباعة هذا أبداً" لا شيء<true unless=macro_unless>
```

لاحظ أنه في تنفيذ الماكرو الخاص بنا، لم تتم طباعة الجملة، على الرغم من طباعتها في تنفيذ الدالة، وذلك لأن وسيطات استدعاء الدالة يتم تقييمها قبل استدعاء الدالة. ومع ذلك، لا تقوم وحدات الماكرو بتقييم وسيطاتها. وبدلاً من ذلك، تلقى الوسيطات كتعبيرات مقتبسة يتم تحويلها بعد ذلك إلى تعبيرات مقتبسة أخرى. في هذه الحالة، قمنا بإعادة كتابة ماكرو ما لم يصبح ما إذا خلف الكواليس.

عبارة أخرى، عند استدعائهما كـ:

```
ما لم يكن macro_unless صحيحًا، فإن IO.puts "لا ينبغي طباعة هذا أبداً"
```

تلقي الماكرو الخاص بـ `macro_unless` ما يلي:

```
٪ خام {٪}
```

```
ما لم يكن macro_unless صحيحًا، فإن IO.puts "لا ينبغي طباعة هذا أبداً" ] macro_unless(true, [{:, []}], [{aliases, []}, [IO], :puts]), [], _
```

```
%{سحب}%
```

ثم أعاد التعبير المقتبس على النحو التالي:

```
٪ خام {٪}
```

```
ما لم يكن macro_unless صحيحًا، فإن IO.puts "لا ينبغي طباعة هذا أبداً" ] macro_unless(true, [{:, []}], [{aliases, []}, [IO], :puts]), [], _
```

```
%{سحب}%
```

يمكنا التتحقق فعلياً من هذه الحالة باستخدام `Macro.expand_once/2`:

```
iex> res = Macro.expand_once(expr, __ENV__)
iex> IO.puts Macro.to_string(res)
  if(!true) do
    expr = quote do: Unless.macro_unless(true, IO.puts
```

لا يبني طباعة هذا أبداً")IO.puts("

نهاية

نعم

يتلقى Macro.expand_once/2 تعبيراً مقتبساً ويوسعه وفقاً للبيئة الحالية. في هذه الحالة، قام بتوسيع/استدعاء الماكرو Unless.macro_unless/2 وأعاد نتيجته. ثم انتقلنا إلى تحويل التعبير المقتبس المعد إلى سلسلة وطباعته (ستحدث عن __ENV__ لاحقاً في هذا الفصل).

هذا هو جوهر وحدات الماكرو. فهي تتعلق بتلقي التعبيرات المقتبسة وتحويلها إلى شيء آخر. في الواقع، يتم تنفيذ ما لم/2 في Elixir كوحدة ماكرو:

```
defmacro ma(m|جملة, خيارات) {  
  اقتباس افعلي(unquote(options)) unless:  
    اقتباس افعلي(unquote(clause)).}
```

نهاية

يتم تنفيذ التراكيب مثل ما لم/2، def/2، defprotocol/2، defmacro/2، والعديد من التراكيب الأخرى المستخدمة في هذا الدليل التمهيدي في الحالص، غالباً كوحدات ماكرو. وهذا يعني أن التراكيب المستخدمة لبناء اللغة يمكن للمطوروين استخدامها لتوسيع اللغة إلى المجالات التي يعملون عليها.

يمكنا تعريف أي وظيفة أو ماكرو نريده، بما في ذلك تلك التي تتجاوز التعرifات المضمنة التي يوفرها Elixir. الاستثناء الوحيدة هي نماذج Elixir الخاصة التي لم يتم تنفيذها في Kernel.SpecialForms.html وبالتالي لا يمكن تجاوزها، والقائمة الكاملة للنماذج الخاصة متوفرة في "Kernel.SpecialForms" </docs/stable/elixir/

2.2. نظافة الماكرو

تتمتع وحدات الماكرو Elixir بدقة متأخرة. وهذا يضمن عدم تعارض متغير محدد داخل علامة اقتباس مع متغير محدد في السياق الذي يتم فيه توسيع هذا الماكرو. على سبيل المثال:

```
=1 end.:_quote _defmacro no_interference _defmodule Hygiene  
a  
  
نهاية  
  
HygieneTest do def go do require Hygiene a = 13  
  defmodule  
  
  النظافة.  
  a  
نهاية  
نهاية  
  
HygieneTest.go # => 13
```

في المثال أعلاه، على الرغم من أن الماكرو يحقن `a = 1` فإنه لا يؤثر على المتغير `a` الذي تحدده دالة `go`. إذا أراد الماكرو التأثير صراحةً على السياق، فيمكنه استخدام `var!`:

```
defmacro interference لـ النظافة defmodule
```

اقتباس do: var!(a) = 1 end

نهاية

```
HygieneTest do def go do require Hygiene a = 13
  defmodule
```

النظافة.تدخل

أ

نهاية

نهاية

HygieneTest.go # => 1

لا تعمل نظافة المتغيرات إلا لأن Elixir يشرح المتغيرات بسياقها. على سبيل المثال، سيتم تمثيل المتغير `x` المحدد في السطر 3 من الوحدة النمطية على النحو التالي:

```
{x, 3} [[السطر: شيء]]
```

ومع ذلك، يتم تمثيل المتغير المقتبس على النحو التالي:

```
do quote do: x end عينة من defmodule
  def quotes
```

نهاية

العينة . مقتبس {x, 3} [[السطر: العينة]]

لاحظ أن العنصر الثالث في المتغير المقتبس هو ذرة العينة، بدلاً من الصفر، مما يشير إلى أن المتغير قادم من وحدة العينة. لذلك، يعتبر Elixir هذين المتغيرينقادمين من سياقات مختلفة ويعامل معهما وفقاً لذلك.

يوفر Elixir آليات مماثلة لاستيراد والأسماء المستعارة أيضاً. وهذا يضمن أن يتصرف الماكرو كما هو محدد بواسطة وحدة المصدر الخاصة به بدلاً من التعارض مع وحدة الهدف حيث يتم توسيع الماكرو، يمكن تجاوز النظافة في مواقع معينة باستخدام وحدات ماكرو مثل `var!/2` أو `alias!/2` على الرغم من أنه يجب توخي الحذر عند استخدام تلك الوحدات لأنها تغير بيئة المستخدم بشكل مباشر.

في بعض الأحيان، قد يتم إنشاء أسماء المتغيرات بشكل ديناميكي. في مثل هذه الحالات، يمكن استخدام `Macro.var/2` لتحديد متغيرات جديدة:

```
defmacro initialize_to_char_count(variables) بـ ( عينة defmodule
```

```
= name |> Atom.to_string |> String.length quote do unquote(var) = unquote(length) end متغيرات
  Enum.map . fn(name) -> var = Macro.var(name, nil) length
```

نهاية

نهاية

تعريف تشغيل القيام به

تهيئة_عدد_الأحرف [أحمر، آخر، آخر، أصفر]

[الأحمر، الأخضر، الأصفر] [النهاية](#)

نهاية

> Sample.run #=> [3, 5, 6]

انته للجة الثانية لـ Macro.var/2. وهذا هو السياق المستخدم وسيحدد النطافة كما هو موضح في القسم التالي.

2.3 البيئة

عند استدعاء Macro.expand_once/2 في وقت سابق من هذا الفصل، استخدمنا النموذج الخاص `_ENV`.

تعيد `_ENV` مثيلاً له بكل Macro.Env الذي يحتوي على معلومات مفيدة حول بيئة التجميع، بما في ذلك الوحدة النمطية الحالية والملف والسطر، وجميع المتغيرات المحددة في النطاق الحالي، بالإضافة إلى الواردات والمتطلبات وما إلى ذلك:

```
iex> _ENV__module
      "iex"__file
_iex> _ENV__requires [IEx.Helpers, Kernel, Kernel.Typespec] iex> require Integer nil
      iex>
      [IEx.Helpers, Integer, Kernel, Kernel.Typespec]_VNE_iex> .يطلب
```

تتوقع العديد من الوظائف في وحدة الماكرو وجود بيئه. يمكنك قراءة المزيد عن "Macro" [ومعرفة المزيد عن](#) "Macro.Env" [في وثائق](#) `_ENV` [في وثائق](#) `Macro`.

2.4 وحدات الماكرو الخاصة

Elixir يدعم أيضًا وحدات الماكرو الخاصة عبر defmacro. وباعتبارها وظائف خاصة، فإن وحدات الماكرو هذه متاحة فقط داخل الوحدة النمطية التي تحددها، وفي وقت التجميع فقط.

من المهم تعريف الماكرو قبل استخدامه. سيؤدي الفشل في تعريف الماكرو قبل استدعائه إلى ظهور خطأ في وقت التشغيل، حيث لن يتم توسيع الماكرو وسيتم ترجمته إلى استدعاء وظيفة:

```
do ...> def four, do: two + two ...> defmacro two, do: 2 ...> end
      iex> defmodule Sample
      iex> two()
      **(CompileError) iex:2: دالة two/0 غير محددة
```

2.5 كتابة وحدات الماكرو بطريقة مسؤولة

Elixir تُعد وحدات الماكرو بنية قوية، ويتوفر العديد من الآليات لضمان استخدامها بشكل مسؤول.

• وحدات الماكرو صحيحة: بشكل افتراضي، لن تؤثر المتغيرات المحددة داخل وحدة الماكرو على كود المستخدم، علاوة على ذلك، لن تسرب استدعاءات الوظائف والأسماء المستعارة المتاحة في سياق وحدة الماكرو إلى سياق المستخدم.

• وحدات الماكرو معجمية: من المستحب حقن التعليمات البرمجية أو وحدات الماكرو عالمياً. لاستخدام وحدة الماكرو، تحتاج إلى تتطلب صراحةً أو تستورد الوحدة النمطية التي تحدد الماكرو.

*وحدات الماكرو صريحة: من المستحبل تشغيل وحدة ماكرو دون استدعائنا صراحةً. على سبيل المثال، تسمح بعض اللغات للمطوريين بإعادة كتابة الوظائف بالكامل خلف الكواليس، غالباً عبر تحويلات التخطيط أو عبر بعض آليات الانعكاس، في Elixir يجب استدعاء وحدة ماكرو صراحةً في المتصل.

لغة وحدات الماكرو واضحة: توفر العديد من اللغات اختصارات بناء الجملة لعلامات الاقتباس والحذف. في Elixir، فضلنا أن تكون هذه الاختصارات واضحة، من أجل تحديد حدود تعريف وحدة الماكرو وتبينها المقتبسة بوضوح.

حتى لو حاولت `Elixir` بذلك قصارى جهدها لتوفير بيئة آمنة، فإن المسؤولة الأكيد تقع على عاتق المطوروين. لهذا السبب فإن القاعدة الأولى لنادي الماكرو هي كتابة الماكرو بطريقة مسؤولة. إن كتابة الماكرو أصعب من كتابة وظائف `Elixir` العادمة، ويعتبر استخدامها عندما لا تكون ضرورية أسلوباً سيئاً. يوفر `Elixir` بالفعل، أدوات أنيقة لكتابية التعليمات البرمجية اليومية الخاصة بك، ويجب حفظ الماكرو كملاذ أخير.

إذا كنت بحاجة إلى اللجوء إلى وحدات الماكرو، فتذكرة أن وحدات الماكرو ليست واجهة برمجة التطبيقات الخاصة بك. احرص على أن تكون تعريفات وحدات الماكرو قصيرة، بما في ذلك محتواها المقتسنة. على سبيل المثال، بدلاً من كتابة وحدة ماكرو مثل، هذا:

```
defmacro my_macro(a, b, c) إلإ defmodule MyModule
    do_this(unquote(a)) إلإ quote
    ...
    افعل ذلك(احذف علامة الاقتباس(ب))
    ...
    و ذلك(((c)etouqnu)) النهاية
نهاية
نهاية
```

يكتب:

```
defmodule MyModule do
  defmacro my_macro(a, b, c) do
    quote do
      do_this_that_and_that(unquote(a), unquote(b), unquote(c))
    end
  end
end
```

احرص على تقليل ما تحتاج إلى القيام به هنا إلى الحد الأدنى #وأنقل كل شيء آخر إلى دالة

يجلب هذا الكود الخاص بك أكثر وضوحاً وأسهل في الاختبار والصيانة، حيث يمكنك استدعاء `this`_that_and_that/3 مباشرةً. كما يساعدك أيضاً في تصميم واجهة برمجة تطبيقات فعلية للمطورين لا تقتصر على مجرد إدخالات الماكرو.

بهذه الدروس، ننهي مقدمتنا حول وحدات الماكرو. الفصل التالي عبارة عن مناقشة موجزة حول لغات المجال المحدد التي توضح كيف يمكننا مزج وحدات الماكرو وسمات الوحدات النمطية لشرح الوحدات النمطية والوظائف وتوضيحها.

لغات خاصة بالمحاج

toc.html \%}, \سایه{\%

اللغات الخاصة بالمجال تسمح للمطوريين بتخصيص تطبيقاتهم لمجال معين. هناك العديد من ميزات اللغة التي، عند استخدامها معاً، يمكنها مساعدة المطوريين على كتابة لغات خاصة بمجال معين. في هذا الفصل، سنركز على كيفية استخدام وحدات الماكرو وسمات الوحدة معاً لإنشاء وحدات خاصة بمجال معين تركز على حل مشكلة معينة. على سبيل المثال، سنكتب وحدة بسيطة للغاية لتحديد الاختبارات وتشغيلها.

الهدف هو بناء وحدة تسمى `TestCase` والتي تسمح لنا بكتابة ما يلي:

```
defmodule MyTest do
  use TestCase

  describe "العمليات الحسابية" do
    test "4 + 2 = 6" do
      assert 4 + 2 == 6
    end
  end

  describe " العمليات القائمة" do
    test "[1, 2, 3] ++ [3, 2, 1] = [1, 2, 3, 2, 1]" do
      assert [1, 2, 3] ++ [3, 2, 1] == [1, 2, 3, 2, 1]
    end
  end
end
```

تشغيل اختباري

في المثال أعلاه، باستخدام `TestCase` يمكننا كتابة اختبارات باستخدام ماكرو الاختبار، الذي يحدد دالة تسمى `run` لتشغيل جميع الاختبارات تلقائياً نيابة عنا. سيعتمد النموذج الأولي لدينا ببساطة على عامل المطابقة (=) كآلية للقيام بالتأكدات.

13.1 اختبار الماكرو

لنبدأ بإنشاء وحدة نمطية تعمل ببساطة على تعريف واستيراد ماكرو الاختبار عند استخدامه:

```
defmodule TestCase do
  @doc """
  # تم استدعاء الاستدعاء بواسطة `use` .
  # في الوقت الحالي، يقوم ببساطة بإرجاع تعبير مقتبس يستورد الوحدة نفسها إلى كود المستخدم.

  defmacro __using__(opts)
  end

  @doc """
  # يقوم بتعريف حالة اختبار بالوصف المحدد.

  # الأمثلة
  describe "العمليات الحسابية" do
    test "4 + 2 = 6" do
      assert 4 + 2 == 6
    end
  end

  @doc """
  # الوصف، قم بـ `function_name` . قم بـ `String.to_atom("test")` quote . قم بـ `tsetdefmacro` .
  # `etouqnu` : كتلة `end` ( ) <> الوصف ( ) <>
  def unquote(function_name) do
    function_name = String.to_atom("test")
    quote
      tsetdefmacro
    end
  end
end
```

نهاية

نهاية

بالافتراض أنا قمنا بتعريف `TestCase` في ملف يسمى `tests.exs`، فمكنا فتحه عن طريق تشغيل `iex tests.exs` وتحديد اختباراتنا الأولى:

```
TestCase iex> defmodule MyTest do ...>
```

```
...>
"اختبار "مرحبا
"مرحبا" = "العالم
...نهائية
...نهائية
```

في الوقت الحالي، ليس لدينا آلية لتشغيل الاختبارات، ولكننا نعلم أن وظيفة تسمى `"test hello"` تم تعريفها خلف الكواليس. عندما نستدعيها، يجب أن تفشل:

```
...".اخبار مرحبا()>
world"> لا يوجد تطابق لقيمة الجانب الأيمن :世界 (MatchError)
```

3. تخزين المعلومات باستخدام السمات

لكي ننهي تنفيذ `TestCase`. تحتاج إلى أن نتمكن من الوصول إلى جميع حالات الاختبار المحددة، إحدى الطرق للقيام بذلك هي استرداد الاختبارات في وقت التشغيل عبر `MODULE___.info_(functions)` والتي تعيد قائمة بجميع الوظائف في وحدة معينة. ومع ذلك، نظرًا لأننا قد نرغب في تخزين المزيد من المعلومات حول كل اختبار بالإضافة إلى اسم الاختبار، فإن الأمر يتطلب نهجًا أكثر مرونة.

عند مناقشة سمات الوحدة النمطية في الفصول السابقة، ذكرنا كيف يمكن استخدامها لتخزين مؤقت. وهذه هي الخاصية بالضبط التي سنطبقها في هذا القسم.

في تنفيذ `1 / __using_` سنقوم بتهيئة سمة وحدة تسمى `@tests` إلى قائمة فارغة، ثم نقوم ب تخزين اسم كل اختبار محدد في هذه السمة حتى يمكن استخدام الاختبارات من وظيفة التشغيل.

فيما يلي الكود المحدث لوحدة `TestCase`:

```
defmodule TestCase do
  @doc """
  افتعل defmacro __using__(opts)
  اقتباس لاستيراد TestCase
  #قم بتهيئة [] إلى قائمة فارغة []
  @tests []

  @before_compile TestCase end #استدعاء /_before_compile
```

نهاية

`@doc """"`
يقوم بتعريف حالة اختبار بالوصف المحدد.

#أمثلة

"اختبار "العمليات الحسابية"
 $4 = 2 + 2$
نهاية

```
"""
|إضافة الاختبار المحدد حديثاً
defmacro test(description, do: block) do function_name = String.to_atom("test " quote do #
@tests [unquote(function_name)] @tests def unquote(function_name)(), do: unquote(block)
 إلى نهاية
```

نهاية
نهاية

#سيتم استدعاء هذا قبل تجميع الوحدة المستهدفة مباشرةً # مما يمنحنا الفرصة المتألية لحقن وطيفة do

اقتباس فعل ديف ران فعل

```
[]() end "Enum.each @tests, fn name -> IO.puts
      #{"name}" apply(__MODULE__, name,
```

نهاية
نهاية
نهاية
نهاية

من خلال بدء جلسة IEx جديدة، يمكننا الآن تحديد اختباراتنا وتشغيلها:

```
iex> defmodule MyTest do
  TestCase...استخدم
  ...>
  >...اختبار "مرحبا"
    "العالم"="مرحبا"
  >...النهاية
  >...النهاية
  world"world" لا يوجد تطابق لقيمة الجانب الأيمن : (ModelError) اختبار  التشغيل iex> MyTest.run
```

على الرغم من أننا تجاهلنا بعض التفاصيل، فهذه هي الفكرة الرئيسية وراء إنشاء وحدات خاصة بالمحال في Elixir. تمكنا وحدات الماكرو من إرجاع التعبيرات المقبسية التي يتم تنفيذها في المتصل، والتي يمكننا استخدامها بعد ذلك لتحويل التعليمات البرمجية وتخزين المعلومات ذات الصلة في الوحدة النمطية المستهدفة عبر سمات الوحدة النمطية. أخيراً، تسمح لنا عمليات الاسترجاع مثل before_compile بحقن التعليمات البرمجية في الوحدة النمطية عند اكتمال تعريفها.

بالإضافة إلى before_compile، يوجد سمات وحدة مفيدة أخرى مثل stable/elixir/Module.html'@before_compile، والتي يمكنك قراءة المزيد عنها في وثائق وحدة Macro.html'@on_definition و@after_compile. يمكنك أيضًا العثور على معلومات مفيدة حول وحدات الماكرو وبيئة التجميع في وثائق وحدة Macro.Env'@Macro.Env.html'. يمكنني ملخصاً ما يلي:

الفصل الثاني

الأدلة الفنية

قواعد تحديد النطاق في Elixir و Erlang

- أنواع النطاق
- نطاقات الإكسير هي معجمية
- نطاق التعشيش والتظليل
- نطاق المستوى الأعلى
- نطاق جملة الوظيفة
- الوظائف والوحدات المسماة
- جمل تشبه الحالة
- جرب الكتل
- الفهم
- تتطلب، وتستورد، وتسمى اسمًا مستعارًا
- الاختلافات عن Erlang

بالنسبة للاستخدام اليومي، يكفي فهم أساسيات قواعد النطاق في Elixir: أن هناك نطاق المستوى الأعلى ونطاق شرط الوظيفة، وأن الوظائف المسماة لها اختلافاتها الخاصة عن الوظائف المجهولة الأكثر تقليدية.

ولكن هناك، في الواقع، عدد لا يأس به من القواعد التي تحتاج إلى معرفتها للحصول على صورة كاملة عن الطريقة التي تعمل بها النطاقات في Elixir. في هذه المقالة الفنية، سنلقي نظرة فاحصة على جميع قواعد النطاق ونتعلم ما هي الاختلافات بينها وبين Erlang.

أنواع النطاق

يوجد في Elixir نوعان من النطاق:

- نطاق المستوى الأعلى

- نطاق جملة الوظيفة

هناك عدد من البنيات التي تخلق نطاقاً جديداً:

- الوحدات النمطية والهياكت الشبيهة بالوحدة النمطية: defmodule, defprotocol, defimpl

- الوظائف: fn, def, defp

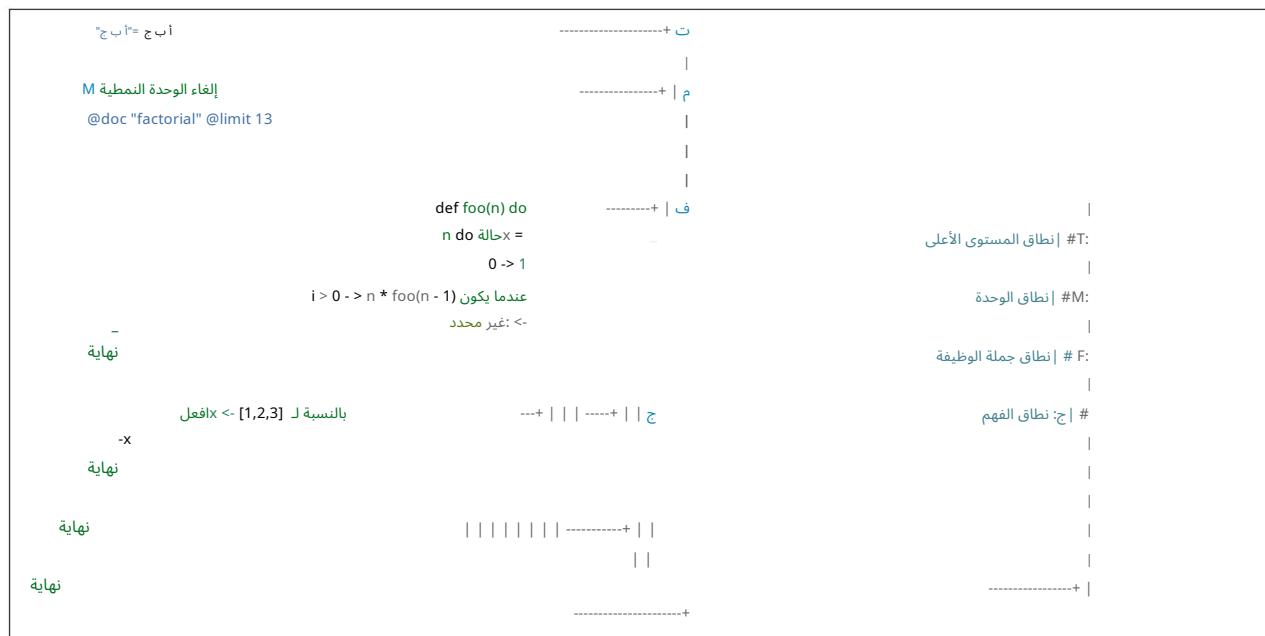
- الفهم: لـ

- حاول إنشاء أجسام كتلة

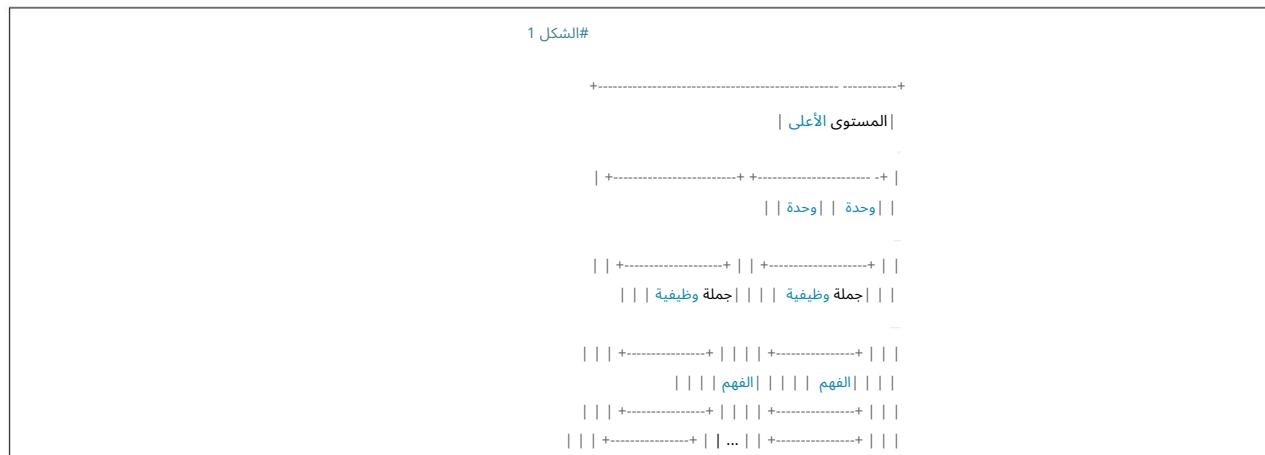
في أغلب الأحيان، يتم تنظيم كود المستخدم في Elixir بالطريقة التالية. في المستوى الأعلى، نقوم بتعريف الوحدات النمطية. كل وحدة نمطية هي:

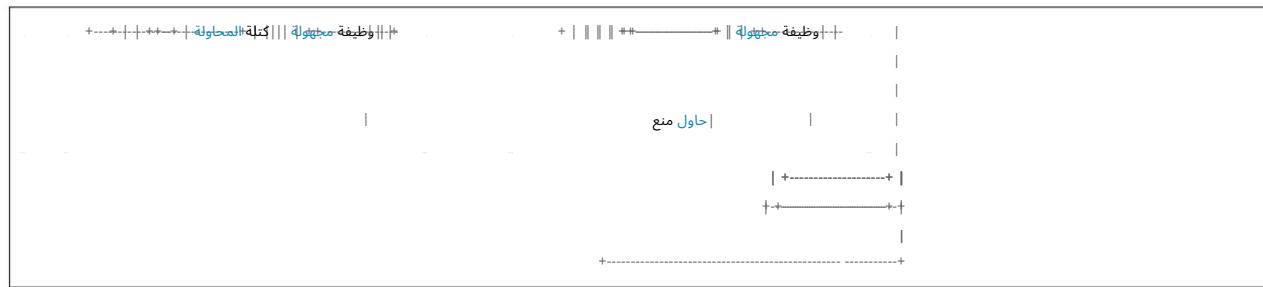
تحتوي الوحدة النمطية على عدد من السمات وحمل الدالة. داخل جملة الدالة يمكن أن يكون هناك عدد عشوائي من

العبارات التي تتضمن تراكيب تدفق التحكم مثل try أو case أو if:

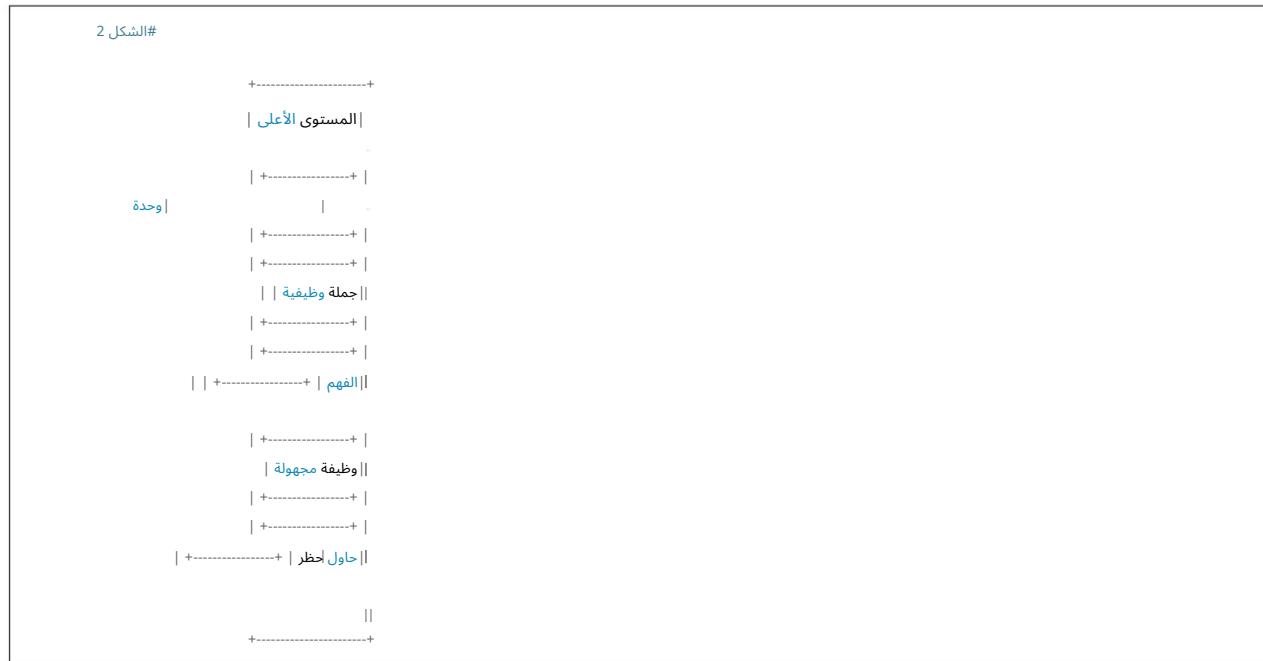


هناك طريقة أخرى لتصور هذا الهيكل، تخطيطياً:



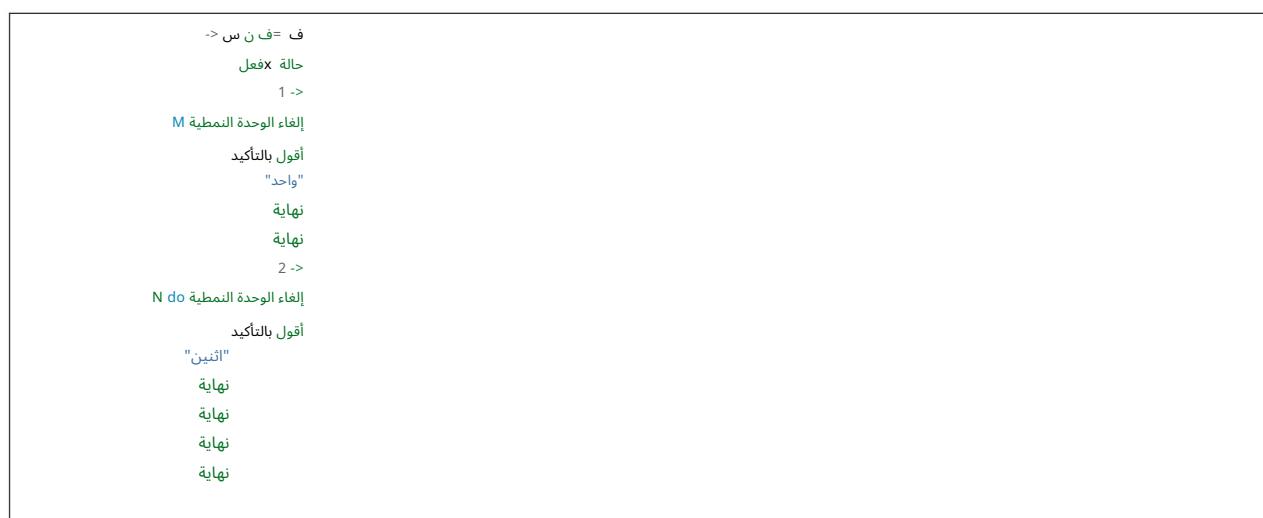


عند العمل في الغلاف التفاعلي، يكون التسلسل الهرمي للنطاق مسطحاً عادةً ("جملة الوظيفة" في الرسم البياني أدناه الآن يشير إلى وظائف مجهرة بدلاً من الوظائف المسمى):



تلك هي الهياكل الأكثر شيوعاً لتنظيم التعليمات البرمجية في Elixir.

في الحالة العامة، مع ذلك، تكون جميع النطاقات قابلة للتعشيش بشكل تعسفي: يمكننا أن نتخيل تعبير حالة داخل فهم أو تعبير أعلى مستوى أعلى يحدد وحدات مختلفة اعتماداً على بعض الشروط. على سبيل المثال:



```

# لم يتم تعريف أي وحدة حتى الآن
#=> دالة غير محددة: say/0
#=> دالة غير محددة: N.say/0

#تعريف م
(1).
ف.ساي
م.ساي
ن.ساي

# واحد
#=> واحد
#=> N.say/0

# حدد ن
(2).
ف.ساي
م.ساي
ن.ساي

# واحد
#=> واحد
#=> إثنان

```

لفهم كيفية عمل المثال أعلاه، يجب أن تكون على دراية بحقيقة أن تعريف الوحدة النمطية ينشئ الوحدة كأثر جانبي لها، لذا ستكون الوحدة نفسها متاحة عالمياً. فقط اسم الوحدة هو المتأثر من خلال تعشيش استدعاء `defmodule` كما سنرى لاحقاً في هذه المقالة.

نطاقات الإكسير هي معجمية

وهذا يعني أنه من الممكن تحديد نطاق كل معرف فقط من خلال النظر إلى الكود المصدر. توفر جميع ارتباطات المتغيرات المقدمة في نطاق معين حتى نهاية هذا النطاق. يحتوي Elixir على بعض الأشكال الخاصة التي تعامل مع النطاقات بشكل مختلف قليلاً (أي تتطلب، وتستورد، وتسمى مستعاراً). سوف نفحصها في نهاية هذه المقالة شرط.

التعشيش والتظليل في النطاق

وفقاً لقواعد النطاق المعجمي، يمكن الوصول إلى أي متغيرات محددة في النطاق المحبط في جميع النطاقات الأخرى إنه يحتوي على.

في الشكل أعلاه، أي متغير محدد في نطاق المستوى الأعلى سيكون متاحاً في نطاق الوحدة وأي نطاق متداخلة بداخله، وهكذا.

يوجد استثناء لهذه القاعدة ينطبق فقط على الوظائف المسممة: أي متغير يأتي من المحبط يجب أن يكون النطاق غير مقتبس داخل نص جملة الوظيفة.

أي متغير في نطاق متداخل ينطابق اسمه مع متغير من النطاق المحبط سوف يحجب ذلك متغير خارجي. بعبارة أخرى، يقوم المتغير الموجود داخل النطاق المتداخل بإخفاء المتغير مؤقتاً عن النطاق المحبط النطاق، ولكن لا يؤثر عليه بأي شكل من الأشكال.

النطاق الأعلى مستوى

يتضمن نطاق المستوى الأعلى كل متغير ومعرف تم تعريفهما خارج أي نطاق آخر.

```

س => دالة غير محددة: x/0
      = 1
      س => 1

      x = fn ->
        #(&:<)

```

لا يمكن تعريف الدوال المسممة على المستوى الأعلى لأن الدالة المسممة تنتهي دائمًا إلى وحدة نمطية. ومع ذلك، يمكن استيراد الدوال المسممة إلى أي نطاق معجمي (بما في ذلك نطاق المستوى الأعلى) على النحو التالي:

استيراد سلسلة فقط: [عكس: 1]

عكس "مرحبا" #=> "olleH"

في الواقع، يتم استيراد جميع الوظائف والمакرو من وحدة Kernel تقليديًا في نطاق المستوى الأعلى بواسطة المترجم.

نطاق جملة الوظيفة

تعرف كل جملة وظيفة نطاقًا معجميًا جديدًا: أي متغير جديد موجود داخلها لن يكون متاحًا خارج تلك الجملة:

إلغاء الوحدة النمطية M

-x: افعل def foo(x).

'foo/1' def bar(x), do: 2*x #هذا 'x' مستقل تمامًا عن الموجود في x

= 1 س

f = fn(x) -> x = x + 1 #التطبيق في العمل: يقوم "x" في قائمة الوسيطات بإنشاء متغير محلي لجسم جملة الدالة وليس له علاقة بـ "x" المحدد مسبقًا

نهاية

ي = ف.(س)

2 == 2 #{{f.(x)}" == #{{y}}#IO.puts الإجابة الصحيحة هي

'y/0' def y(y), do: y*2 #في هذه الحالة، تحجب الحجة 'y' الدالة المسممة

'y/0' def y, do: y*2 # هنا الإشارة إلى 'y' داخل نص الدالة # في الواقع هي استدعاء متكرر إلى

نهاية

3. فو #=> -3 #=> 8

4. بار #=>

#=> -4

#=> حلقة لا نهاية

بلدي 2-بلدي

بصرف النظر عن الوظائف المسممة، يتم إنشاء نطاق شرط وظيفة جديد لكل كتلة تشبه الوحدة النمطية، أو وظيفة مجهولة، أو نص كتلة المحاولة، أو نص الفهم (انظر أدناه).

ف = ف(س) = 1 -

نهاية

أ

a/0 #=> دالة غير محددة:

ج = ف(ف)

<p>f = نهاية</p> <p>ف</p> <p>ج</p>	<p>#(لا نزال الوظيفة المجهولة المحددة أعلاه) =>(الوظيفة المجهولة التي قمنا بتعريفها للتو)</p>
--	--

الوظائف والوحدات المسماة

كما ذكرنا سابقًا، تتمتع الوظيفة المسماة بعض الخصائص.

أولاً، لا يؤدي تعريف وظيفة مسماة إلى تقديم ارتباط جديد إلى النطاق الحالي:

<p>إلغاء الوحدة النمطية M</p> <p>ديف فو، تفعل: "مرحبا"</p>	<pre>foo/0::=CompileError إلى حدوث خطأ في: #</pre>
--	--

ثانياً، لا يمكن للوظائف المسماة الوصول مباشرة إلى النطاق المحلي، ويجب استخدام علامة الاقتباس لتحقيق ذلك:

<p>إلغاء الوحدة النمطية M</p> <p>أ = 1</p> <p>بisher'a'da داخل () لا ليس فيه إلى'a'mحدد # في نطاق الوحدة</p> <p>def a, do: unquote(a) #يشير'a'dا داخل لا ليس فيه إلى الدالة</p> <p>نهاية</p> <p>ما</p> <p>3 ما</p>	<pre>#يشير'a'dا داخل النص بشكل لا ليس فيه إلى الدالة 'a/0' def a(b), do: a + b #يشير'a'dا داخل</pre>
--	--

يعمل نطاق الوحدة النمطية تماماً مثل نطاق شرط الوظيفة: أي متغيرات محددة بين `defprotocol` (أو `defmodule`) وما إلى ذلك) ونهايتها المقابلة لن تكون قابلة للوصول خارج الوحدة النمطية، ولكنها ستكون متاحة في النطاقات المتداخلة لتلك الوحدة النمطية كالمتغير `modulo` (التحذير غير المقتبس للوظائف المسماة المذكورة أعلاه).

من المهم أن نفهم أن نطاق الوحدة النمطية موجود طالما يتم تجميعها، بعبارة أخرى، لا يتم "تجمیع" المتغيرات في الوحدة النمطية. لا ينطبق بناء جملة `Module.function` إلا على الدوال المسماة وهذا شيء آخر يجعل مثل هذه الدوال خاصة:

<p>إلغاء الوحدة النمطية M</p> <p>"مرحبا" X =</p> <p>مرحبا ، افعل: إلغاء الاقتباس (نهاية)</p> <p>م. هي</p> <p>مكس</p>	<pre>X/#=> "مرحبا" #/#=> دالة غير محددة: /0'a'</pre>
--	--

قد تتساءل عن كيفية عمل استدعاءات الوظائف المحلية عندما لا تنتهي الوظائف المسماة ارتباطات بالأسماء ولا تتمتع بالوصول المباشر إلى النطاق المحلي. تكمن الإجابة على هذا السؤال في القاعدة التالية التي يتبعها Elixir عند محاولة حل معرف إلى قيمته:

يتم التعامل مع أي معرف غير مرتبط باعتباره استدعاء وظيفة محلية.

دعونا نرى كيف يعمل هذا في الكود:

```
P do
  إلغاء الوحدة النمطية
```

P" def g, do: f end حرف لـ def f, do:

Q's f" def g, do: f end لـ defmodule Q do def f, do:

يشير كل من الحرفين 'g' و 'f' من الحرف إلى صديقهما المحلي المسمى 'f'.

P's f" أنا "#=> ص ح ج
Q's f" أنا "#=>

دعنا نجعل 'f' محلية في نطاق المستوى الأعلى <=> دالة غير محددة.

استيراد بـ f

P's f" أنا "#=>

ملاحظة أخرى حول تسمية الوحدات النمطية والوحدات النمطية المتداخلة: يتم تعريف الوحدات النمطية دائمًا في المستوى الأعلى، بغض النظر عن النطاق الذي توجد فيه المكالمات الفعلية إلى. وهذا يعني أنه طالما أن الجهاز الافتراضي يمكنه العثور على ملف maeb الذي يحتوي على كود الوحدة النمطية في وقت التشغيل، فلا يهم النطاق الذي تشير فيه إلى اسم الوحدة النمطية.

ما يؤثر عليه النطاق هو الاسم الذي ستحصل عليه الوحدة:

```
P do
  إلغاء الوحدة النمطية
```

P defmodule Q do def q(false), do: "sorry" def q(true) do # سيكون اسم الوحدة الفعلي هو PQ، ولكنه مُشتّت ضمناً من # في نطاق Q

PQM defmodule M do # سيكون اسم الوحدة الفعلي هو PQM

أقول وأفضل: "مرحباً" النهائي

نهاية
نهاية

PQ def foo do # يتم حل إلى Q

نهاية خاطئة

في وقت التشغيل، يكون لهذا نفس التنفيذ الدقيق مثل foo def bar do

نهاية PQq

نهاية

نهاية

بـ فـ
صـ بـ فـ

"آسف" "#=> "آسف" "#=>

"آسف" "#=>

PQMsay # تم تعریف الوحدة النمطية بعد

PQMsay/0 # دالة غير محددة:

بعد هذه المكالمة ستصبح وحدة PQM متاحة

PQq صحيح
بي كيو ام ساي "#=> مرحبا"

جمل تشبه الحالة

تشارک هیاکل تدفق التحكم receive و cond في سمة مشتركة:

- أي متغير تم تقديمها في نمط/شرط البند سيكون متاحًا فقط داخل نص البند
- أي متغير يتم تقديمها داخل بعض نصوص البند (ولكن ليس كلها) سيصبح متاحًا في النطاق المحيط (ربما مع القيمة الافتراضية الصفرية)

وفيما يلي بعض الأمثلة على تطبيق هذه القواعد:

حالة x فعل

```
# يمكن رؤية كل من "result" و "a" فقط داخل نص هذه الفقرة
a : tcepsni.OI=a -> {النتيجة}:ok,
                                         # مرتبطة فعليًا بالنطاق المحيط: ستكون قيمته صفرًا
                                         # إذا لم يتطابق error :x#
:error -> error = true

# التطبيق العادي: هنا "x" مرئي فقط داخل نص الجملة و
# لا يؤثر على "x" من النطاق المحيط
[x] -> IO.inspect(x)
نهاء
```

النتيجة <=> دالة غير محددة: النتيجة/0
أ <=> دالة غير محددة: 0/a

خطأ <=> صحيح إذا كان .error == x ولا فهو لا شيء

ملاحظة: نظرًا لوجود خطأ في سلسلة ، فإن شروط cond تتسرّب فعليًا إلى النطاق المحيط. يجب أن يكون هذا سينم إصلاحه في 0.13.1.

شرط القيام بذلك

```
> a = a0 خطأ
= ب
= ج
= 3 خطأ
نهاء
```

أ
ب
ج
د

إذا x = 3 أفعل

```
y = :ok do
  الحاله
  حسنا -> حسنا
  "إله خطأ":error -> a =
نهاء
آخر
z = 11
نهاء
```

مرتبط بـ false #=> داخل نص الشرط الأول
b/0: دالة غير محددة #=>
`c = 2': لا شيء (الشرط الثاني صحيح، لهذا لم يتم تقييم `d = 3': لا شيء (لم يتم تقييم الجسم الذي يحتوي على ، لذا فإن `d' يتسرّب أيضًا بالقيمة الافتراضية) #

إذا x = 3 أفعل

```
y = :ok do
  الحاله
  حسنا -> حسنا
  "إله خطأ":error -> a =
نهاء
آخر
z = 11
نهاء
```

```

س      #=> 3
ي      <=:#
أ      لا شيء<#
ز      لا شيء<#

```

جرب الكتل

تعمل كتلة المحاولة بشكل مشابه للحالة والاستلام، لكنها تنشئ نطاقًا جديداً، لذا فهي لا تتسرّب أبداً من ارتباطات المتغيرات إلى النطاق المحيط.

```

حاول أن تفعل
#جميع المتغيرات المحددة هنا محلية لهذه الكتلة

#(مثل نطاق جملة الوظيفة)
= ١
+ ١ = ب
ج = د
ينفذ

`case` #تعمل هذه مثل الارتباطات في أنماط
-> y = x في [خطأ وقت التشغيل]
س <- ض = س
نهاية

أ
ب
ج
د
س
ي
ز
لم يتم تسريب أي من المتغيرات
a/0 : دالة غير محددة:>
b/0 : دالة غير محددة:>
c/0 : دالة غير محددة:>
d/0 : دالة غير محددة:>
x/0 : دالة غير محددة:>
y/0 : دالة غير محددة:>
z/0 : دالة غير محددة:>

```

الفهم

يتكون الفهم من جزأين: المولد والجسم.

ستكون المتغيرات التي تم إدخالها في جزء المولد مరئية فقط داخل الجسم.

```

b = {a, x}; a = x <- [1, 2, 3, 4]; #افعل
 بالنسبة إلى [1, 2, 3, 4]
#=> [[1, 1], {2, 2}, {3, 3}, {4, 4}]

أ
س
a/0 : دالة غير محددة:>
x/0 : دالة غير محددة:>

```

يعمل جسم الفهم نفسه مثل نطاق جملة الوظيفة:

```

 بالنسبة إلى ["abc", "def"], -> x #افعل
# يتم تطبيق الاستيراد فقط داخل نص الفهم
استيراد سلسلة فقط: [عكس]: [1]
ب = عكس س
نهاية
#=> ["cba", "fed"]

ب
b/0 : دالة غير محددة:>

```

عکس "مرحبا"

دالة غير محددة: عکس/1=>

تطلب، استيراد، واسم مستعار

تنطبق جميع القواعد الموضحة حتى الان على ارتباطات المتغيرات. وعندما يتعلق الأمر بأحد هذه الأشكال الثلاثة الخاصة، يستمر تأثيرها حتى نهاية كتلة do التي يتم استدعاؤها فيها. في الواقع، تشهد هذه الأشكال تقسيم نطاق مختلف قليلاً حيث تقوم هياكل تدفق التحكم بإنشاء نطاق معجمي جديد:

#نطاق المستوى الأعلى

M إلقاء الوحدة النمطية

#نطاق جديد استيراد سلسلة، فقط: [عکس: 1]

دريف فو دو

#نطاق جديد استيراد سلسلة، فقط: [الشرط: 1]

مورووث من النطاق المحلي IO.puts reverse("abc") # ok:

إذا كان هذا صحيحاً فافعل

#نطاق جديد استيراد سلسلة، فقط: [آخر صغيرة: 1] ولا

#نطاق جديد استيراد سلسلة، فقط: [نهاية upcase: 1]

"مرحبا"

|> strip |> downcase # error: no local function downcase/1 مواتق: تم جعله محلياً في النطاق المحلي

#نفس الشيء

نهاية

شريط تعريف

#نطاق جديد

موروثر من النطاق المحلي IO.puts reverse("abc") # ok:

نهاية

الاختلافات عن Erlang

لقد تم تورث معظم قواعد النطاق الموضحة هنا من Erlang.

أحد الفروق الملحوظة هو أن الوحدات النمطية تحتوي ببساطة على نماذج وجمل وظيفية، وليس لها نطاق ولا تسمح بتعابيرات عشوائية مثل الوحدات النمطية في Elixir.

هناك اختلافان في الطريقة التي يعمل بها نطاق شرط الحالة في Erlang:

1. كل الارتباطين المقدمين في النمط وفي نص اليند يعدلان النطاق المحلي

2. ستعلن المتغيرات المرتبطة ببعض البنود (ولكن ليس كلها) غير مرتبطة بالنطاق المحلي (بدلاً من الحصول على قيمة الصفر كما يفعلون في Elixir)، يطلق عليهم أيضاً اسم المتغيرات غير الآمنة

```

الحالة رقم 1 من
:=> ب = 1
:=> ج = 1
-
نهاية.

%=> 1 . أ
%=> 1 . ب
%=> => ج المتغير 'C' غير مرتبطة

```

يوجد بناء `if` في Erlang يشبه بناء `cond` لكنه يعمل بشكل مختلف. فهو يسمح فقط بتعوييرات الحماية كشرط ولا تسمح لك هذه التعوييرات بتقديم ارتباطات متغيرة. تتسرّب المتغيرات المرتبطبة في نصوص البنود إلى النطاق المحيط بنفس الطريقة التي تتسرّب بها في حالة.

```

1|X = 1.

س => س،
صحيح => ب = س
نهاية.

أ. المتغير 'A' غير مرتبطة
%=> 1 . ب

%%%%

= 7 صحيح، إذا
-> Q = Y -> P = Y:
نهاية.

ص. %=>
س. المتغير 'Q' غير مقيد

```

[راجع هذه الصفحة لمزيد من المعلومات حول هيكل تدفق التحكم Erlang.](#)

قائمة متنوعة من الموارد التي تصف جوانب مختلفة لقواعد نطاق إرلانج:

- [المطابقة والحراسة ونطاق المتغيرات من دليل البدء الخاص ب Erlang.](#)

- [نطاق المتغيرات في دورة إرلانج.](#)

- [قواعد ثابتة لتحديد نطاق المتغيرات في إرلانج ورق](#)

- [نطاق تعبير الحالة سؤال في قائمة البريد الإلكتروني ل Erlang](#)