开始

# 调试

1 IO.inspect/2

2 dbg

3 断点

4 观察者

5 其他工具和社区

有许多方法可以在 Elixir 中调试代码。在本章中，我们将介绍一些更常见的方法。

## IO.inspect/2

在调试中真正有用的是，它返回传递给它的参数，而不会影响原始代码的行为。让我们看一个例子。`IO.inspect(item, opts \\ []) item`

```
(1..10)
|> IO.inspect
|> Enum.map(fn x -> x * 2 end)
|> IO.inspect
|> Enum.sum
|> IO.inspect
```

指纹：

```
1..10
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
110
```

新闻：Elixir v1.15 发布

搜索。。。

接口文档

开始

1. 介绍
2. 基本类型
3. 基本运算符
4. 模式匹配
5. 案例、cond 和 if
6. 二进制文件、字符串和字符列表
7. 关键字列表和地图
8. 模块和功能
9. 递归
10. 枚举项和流
11. 过程
12. IO 和文件系统
13. 别名、要求和导入
14. 模块属性
15. 结构体
16. 协议
17. 理解
18. 印记
19. 尝试、捕捉和救援
20. 可选语法表
21. Erlang 库

如您所见，可以在代码中的几乎任何位置“监视”值，而无需更改结果，这使得它在管道中非常有用，就像上面的情况一样。`IO.inspect/2`

`IO.inspect/2`还提供了使用选项装饰输出的功能。标签将在检查前打印：`label` `item`

```
[1, 2, 3]
|> IO.inspect(label: "before")
|> Enum.map(&(&1 * 2))
|> IO.inspect(label: "after")
|> Enum.sum
```

指纹：

```
before: [1, 2, 3]
after: [2, 4, 6]
```

与 `binding()` 一起使用也很常见，它返回所有变量名及其值：`IO.inspect/2`

```
def some_fun(a, b, c) do
  IO.inspect binding()
  ...
end
```

当使用 ， 调用时，它将打印：`some_fun/3` `:foo` `"bar"` `:baz`

```
[a: :foo, b: "bar", c: :baz]
```

请参阅 IO.inspect/2 以了解有关使用此函数的其他方式的更多信息。此外，为了查找可以一起使用的其他格式选项的完整列表，请参阅 Inspect.Opts。`IO.inspect/2`

## dbg

药剂 v1.14 介绍。与 类似，但专门针对调试而定制。它打印传递给它的值并返回它（就像 一样），但它也会打印代码和位置。


22. 调试
23. 类型规格和行为
24. 下一步去哪里

混合和一次性密码

1. 混音简介
2. 代理
3. GenServer
4. 主管和申请
5. 动态主管
6. 电子交易体系
7. 依赖项和伞形项目
8. 任务和 gen_tcp
9. 文档测试，模式和
10. 分布式任务和标签
11. 配置和发布

ELIXIR 中的元编程

1. 报价和取消报价
2. 宏
3. 域特定语言

`dbg/2` `dbg` `IO.inspect/2` `IO.inspect/2`

```
# In my_file.exs
feature = %{name: :dbg, inspiration: "Rust"}
dbg(feature)
dbg(Map.put(feature, :in_version, "1.14.0"))
```

上面的代码打印了以下内容：

```
[my_file.exs:2: (file)]
feature #=> %{inspiration: "Rust", name: :dbg}
[my_file.exs:3: (file)]
Map.put(feature, :in_version, "1.14.0") #=> %{in_version:
"1.14.0", inspiration: "Rust", name: :dbg}
```

在谈论时，我们提到了它放置在管道步骤之间的有用性。 做得更好：它理解Elixir代码，因此它将在*管道的每一步*打印值。 `IO.inspect/2` `|>` `dbg`

```
# In dbg_pipes.exs
__ENV__.file
|> String.split("/", trim: true)
|> List.last()
|> File.exists?()
|> dbg()
```

此代码打印：

```
[dbg_pipes.exs:5: (file)]
__ENV__.file #=> "/home/myuser/dbg_pipes.exs"
|> String.split("/", trim: true) #=> ["home", "myuser",
"dbg_pipes.exs"]
|> List.last() #=> "dbg_pipes.exs"
|> File.exists?() #=> true
```

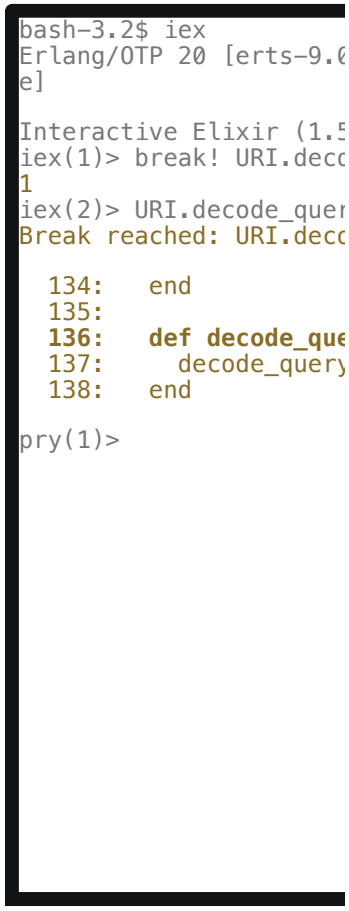
注意仅支持单步执行管道（换句话说，它只能单步执行它看到的代码）。 对于函数的常规单步执行，需要使用 设置断点。 `dbg` `IEx.break!/4`

# 断点

当通过 执行代码调用时，IEx 将要求您“停止”调用所在的代码执行。如果您接受，您将能够直接从 IEx 访问所有变量，以及代码中的导入和别名。这称为“撬动”。当 pry 会话运行时，代码执行将停止，直到 或 被调用。请记住，您始终可以使用 在项目的上下文中运行。

`dbg` `iex` `dbg` `continue` `next` `iex` `iex -S mix TASK`


`dbg` 调用要求我们更改要调试的代码，并且具有有限的步进功能。幸运的是，IEx还提供了一个 `break! /2` 函数，允许你设置和管理任何Elixir代码上的断点，而无需修改其源代码：




与类似，一旦到达断点，代码执行将停止，直到或被调用。但是，无法访问别名和从调试的代码导入，因为它适用于编译的项目而不是源代码。`dbg` `continue` `next` `break!/2`

## 观察者

对于调试复杂系统，跳跃代码是不够的。有必要了解整个虚拟机，进程，应用程序，以及设置跟踪机制。幸运的是，这可以在 Erlang 中使用。在您的应用程序中：`:observer`

```
$ iex
iex> :observer.start()
```

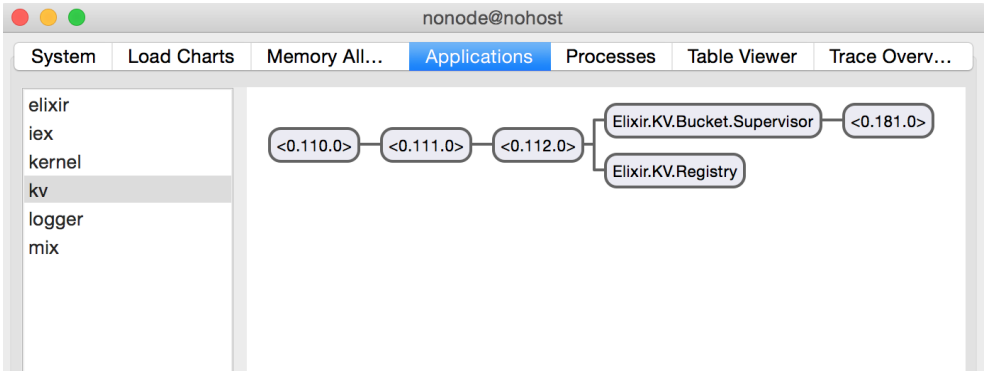
> *使用 在项目中运行时，将不会作为依赖项使用。为此，您需要在之前调用以下函数：* *`iex` `iex -S mix` `observer`*
>
> ```
> iex> Mix.ensure_application!(:wx)
> iex> Mix.ensure_application!(:runtime_tools)
> ```

```
iex> Mix.ensure_application!(:observer)
iex> :observer.start()
```

*如果上述任何调用失败，则可能发生以下情况：某些包管理器默认安装最小化的 Erlang，而不安装用于 GUI 支持的 WX 绑定。在某些软件包管理器中，你可以用更完整的软件包替换无头的 Erlang（在 Debian/Ubuntu/Arch 上查找名为 vs 的软件包）。在其他管理器中，您可能需要安装单独的（或类似名称的）软件包。* `erlang` `erlang-nox` `erlang-wx`

*在将来的版本中，有一些对话可以改进此体验。*

上面将打开另一个图形用户界面，该界面提供了许多窗格来完全理解和导航运行时和项目：




我们在Mix & OTP 指南的动态主管章节中实际项目中探讨观察者。这是 Phoenix 框架用于在一台机器上实现 2 万个连接的调试技术之一。

如果您使用的是 Phoenix Web 框架，它会附带 Phoenix LiveDashboard，这是一个用于生产节点的 Web 仪表板，提供与 Observer 类似的功能。

最后，请记住，您还可以通过直接在 IEx 中调用来获取运行时信息的迷你概述。`runtime_info/0`

# 其他工具和社区

我们刚刚触及了 Erlang VM 必须提供的表面，例如：

- 除了观察器应用程序，Erlang 还包括一个查看崩溃转储 `:crashdump_viewer`

- 与操作系统级别跟踪器集成，如 Linux Trace Toolkit、DTRACE 和 SystemTap

- 微状态记帐测量运行时在短时间间隔内在多个低级任务上花费的时间

- Mix 附带命名空间下的许多任务，例如 `profile` `cprof` `fprof`

- 对于更高级的用例，我们推荐优秀的 Erlang in Anger，它以免费电子书的形式提供

调试愉快！

← 上一页　返回页首　下一→

有什么不对吗？ 在 GitHub 上编辑此页面。

© 2012–2023 长生不老药团队。

Elixir和Elixir标志是The Elixir Team 的注册商标。