开始

印记

我们已经了解到 Elixir 提供双引号字符串和单引号字符列表。但是，这仅涵盖在语言中具有文本表示的结构的表面。例如，原子主要是通过表示创建的。`:atom`

Elixir 的目标之一是可扩展性：开发人员应该能够扩展语言以适应任何特定领域。计算机科学已经成为一个如此广泛的领域，以至于一种语言不可能将其的各个方面作为其核心的一部分来处理。相反，Elixir 旨在使自己可扩展，以便开发人员，公司和社区可以将语言扩展到他们的相关领域。

在本章中，我们将探讨符号，这是语言提供的处理文本表示的机制之一。符号以波浪号 （） 字符开头，后跟一个字母（标识符号），然后是分隔符；（可选）可以在最终分隔符之后添加修饰符。`~`

正则表达式

新闻：Elixir v1.15 发布

搜索。。。

Elixir 中最常见的符号是，用于创建正则表达式：`~r`

```
# A regular expression that matches strings which contain
"foo" or "bar":
iex> regex = ~r/foo|bar/
~r/foo|bar/
iex> "foo" =~ regex
true
iex> "bat" =~ regex
false
```

Elixir 提供了与 Perl 兼容的正则表达式（regexes），由 PCRE 库实现。正则表达式也支持修饰符。例如，修饰符使正则表达式不区分大小写：`i`

```
iex> "HELLO" =~ ~r/hello/
false
iex> "HELLO" =~ ~r/hello/i
true
```

查看 `Regex` 模块，了解有关其他修饰符和支持的正则表达式操作的更多信息。

到目前为止，所有示例都用于分隔正则表达式。但是，符号支持 8 种不同的分隔符：`/`

```
~r/hello/
~r|hello|
~r"hello"
~r'hello'
~r(hello)
~r[hello]
~r{hello}
~r<hello>
```

支持不同分隔符的原因是提供一种在没有转义分隔符的情况下编写文本的方法。例如，带有正斜杠（如 read）的正则表达式可以说比。同样，如果正则表达式具有正斜杠和捕获组（使用 ），则可以选择双引号而不是括号。`~r(^https?://)` `~r/^https?:\/\//` `()`

字符串、字符列表和单词列表符号

除了正则表达式外，长生不老药还附带了另外三个符号。

字符串

符号用于生成字符串，就像双引号一样。当字符串包含双引号时，符号很有用：`~s` `~s`

```
iex> ~s(this is a string with "double" quotes, not
'single' ones)
"this is a string with \"double\" quotes, not 'single'
ones"
```

字符列表

符号是表示字符的常规方式。`~c`

```
iex> [?c, ?a, ?t]
~c"cat"
iex> ~c(this is a char list containing "double quotes")
~c"this is a char list containing \"double quotes\""
```

单词列表

符号用于生成单词列表（*单词*只是常规字符串）。在符号内，单词由空格分隔。`~w` `~w`

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

符号还接受 和修饰符（分别用于字符列表、字符串和原子），它们指定结果列表元素的数据类型：`~w` `c` `s` `a`

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

字符串符号中的插值和转义

Elixir 支持一些符号变体来处理转义字符和插值。特别是，大写字母符号不执行插值或转义。例如，尽管两者都将返回字符串，但前者允许转义码和插值，而后者不允许：`~s` `~S`

```
iex> ~s(String with escape codes \x26 #{"inter" <>
"polation"})
"String with escape codes & interpolation"
iex> ~S(String without escape codes \x26 without #
{interpolation})
"String without escape codes \\x26 without \#
{interpolation}"
```

以下转义代码可用于字符串和字符列表：

- `\\` – 单反斜杠
- `\a` – 铃声 / 警报
- `\b` – 退格
- `\d` - 删除
- `\e` - 逃
- `\f` - 表单馈送
- `\n` – 换行符
- `\r` – 回车
- `\s` –空间
- `\t` –标签
- `\v` – 垂直标签
- `\0` - 空字节
- `\xDD` - 表示十六进制的单个字节（例如 `\x13` ）
- `\uDDDD` 和 - 表示十六进制的 Unicode 代码点（例如 `\u{D...}` `\u{1F600}` ）

除此之外，双引号字符串中的双引号需要转义为 ，类似地，单引号字符列表中的单引号需要转义为 。尽管如此，更改如上所示的分隔符比转义分隔符更好。 `\"` `\'`

符号还支持 heredocs，即三个双引号或单引号作为分隔符：

```
iex> ~s"""
...> this is
...> a heredoc string
...> """
```

heredoc 符号最常见的用例是在编写文档时。例如，在文档中编写转义字符很快就会变得容易出错，因为需要对某些字符进行双重转义：

```
@doc """
Converts double-quotes to single-quotes.

## Examples

    iex> convert("\\\"foo\\\"")
    "'foo'"

"""
def convert(...)
```

通过使用 ，可以完全避免此问题： `~S`

```
@doc ~S"""
Converts double-quotes to single-quotes.

## Examples

    iex> convert("\"foo\"")
    "'foo'"

"""
def convert(...)
```

日历符号

Elixir 提供了几种符号来处理各种口味的时间和日期。

日期

%Date{} 结构包含字段 、 和 。您可以使用符号创建一个：
`year` `month` `day` `calendar` `~D`

```
iex> d = ~D[2019-10-31]
~D[2019-10-31]
iex> d.day
31
```

时间

%Time{} 结构包含字段 、 、 和 。您可以使用符号创建一个： `hour` `minute` `second` `microsecond` `calendar` `~T`

```
iex> t = ~T[23:00:07.0]
~T[23:00:07.0]
iex> t.second
7
```

朴素日期时间

%NaiveDateTime{} 结构包含来自 和 的字段。您可以使用符号创建一个： `Date` `Time` `~N`

```
iex> ndt = ~N[2019-10-31 23:00:07]
~N[2019-10-31 23:00:07]
```

为什么叫幼稚？因为它不包含时区信息。因此，给定的日期时间可能根本不存在，或者在某些时区中可能存在两次 - 例如，当我们在夏令时前后移动时钟时。

UTC 日期时间

%DateTime{} 结构包含与 相同的字段，但添加了用于跟踪时区的字段。符号允许开发人员在 UTC 时区创建日期时间： `NaiveDateTime` `~U`

```
iex> dt = ~U[2019-10-31 19:59:03Z]
~U[2019-10-31 19:59:03Z]
iex> %DateTime{minute: minute, time_zone: time_zone} = dt
~U[2019-10-31 19:59:03Z]
iex> minute
59
```

```
iex> time_zone
"Etc/UTC"
```

自定义符号

正如本章开头所暗示的，Elixir 中的符号是可扩展的。实际上，使用 signgil 等效于使用二进制和字符列表作为参数进行调用：`~r/foo/i` `sigil_r`

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

我们可以通过以下方式访问符号的文档：`~r` `sigil_r`

```
iex> h sigil_r
...
```

我们还可以通过实现遵循该模式的函数来提供我们自己的符号。例如，让我们实现返回整数的符号（使用可选的修饰符使其为负数）：`sigil_{character}` `~i` `n`

```
iex> defmodule MySigils do
...>   def sigil_i(string, []), do:
String.to_integer(string)
...>   def sigil_i(string, [?n]), do: -
String.to_integer(string)
...> end
iex> import MySigils
iex> ~i(13)
13
iex> ~i(42)n
-42
```

符号也可用于在宏的帮助下进行编译时工作。例如，Elixir 中的正则表达式在编译源代码期间被编译成有效的表示，因此在运行时跳过此步骤。如果你对这个主题感兴趣，我们建议你了解有关宏的更多信息，并查看如何在模块（定义函数的地方）中实现符号。`Kernel` `sigil_*`

有什么不对吗？ 在 GitHub 上编辑此页面。