脚本

Typst 嵌入了一种强大的脚本语言。您可以自动执行文档并使用代码创建更复杂的样式。下面是脚本概念的概述。

表达式

在 Typst 中，标记和代码合二为一。除了最常见的元素外，所有元素都是使用函数创建的。为了尽可能方便，Typst 提供了紧凑的语法来将代码表达式嵌入到标记中：使用哈希 （） 引入表达式，并在表达式完成后恢复正常的标记解析。如果字符将继续表达式，但应解释为文本，则可以强行以分号 （） 结束表达式。

```
#emph[Hello] \
#emoji.face \
#"hello".len()
```

Hello
😀
5

上面的示例显示了一些可用的表达式，包括函数调用、字段访问和方法调用。本章的其余部分将讨论更多种类的表达式。有几种表达式与哈希语法不兼容（例如二进制运算符表达式）。若要将这些嵌入到标记中，可以使用括号。

块

为了构建代码并将标记嵌入其中，Typst 提供了两种类型的块：

```
#{
  let a = [from]
  let b = [*world*]
  [hello ]
  a + [ the ] + b
}
```

hello from the **world**

代码块：编写代码时，您可能希望将计算拆分为多个语句，创建一些中间变量等。代码块允许您编写多个表达式，其中需要一个表达式。代码块中的各个表达式应用换行符或分号分隔。将代码块中各个表达式的输出值联接起来，以确定该块的值。没有有用输出的表达式（如绑定）可以与任何值联接而不会产生效果。{ let x = 1; x + 2 }letnone

内容块：使用内容块，可以将标记/内容作为编程值进行处理，将其存储在变量中并将其传递给函数。内容块用方括号分隔，可以包含任意标记。内容块生成 content 类型的值。任意数量的内容块可以作为尾随参数传递给函数。也就是说，等价于 。[**Hey** there!]list([A], [B])list[A][B]

内容和代码块可以任意嵌套。在下面的示例中，与 yielding 的输出连接。[hello]a + [the] + b[hello from the **world**]

绑定与解构

如上所述，可以使用绑定来定义变量。为变量分配符号后面的表达式的值。值的赋值是可选的，如果未赋值，则变量将初始化为 。关键字还可用于创建自定义命名函数。可以访问包含块或文档的其余部分的绑定。let=nonelet

```
#let name = "Typst"
This is #name's documentation.
It explains #name.

#let add(x, y) = x + y
Sum is #add(2, 3).
```

This is Typst's documentation. It explains Typst.

Sum is 5.

let 绑定还可用于解构数组和字典。在这种情况下，赋值的左侧应镜像数组或字典。该运算符可以在模式中使用一次，以收集数组或字典项的剩余部分。

```
#let (x, y) = (1, 2)
The coordinates are #x, #y.

#let (a, .., b) = (1, 2, 3, 4)
The first element is #a.
The last element is #b.

#let books = (
  Shakespeare: "Hamlet",
  Homer: "The Odyssey",
  Austen: "Persuasion",
)

#let (Austen,) = books
Austen wrote #Austen.

#let (Homer: h) = books
Homer wrote #h.

#let (Homer, ..other) = books
#for (author, title) in other [
  #author wrote #title.
]
```

The coordinates are 1, 2.

The first element is 1. The last element is 4.

Austen wrote Persuasion.

Homer wrote The Odyssey.

Shakespeare wrote Hamlet. Austen wrote Persuasion.

您可以使用下划线在解构模式中丢弃元素：

```
#let (_, y, _) = (1, 2, 3)
The y coordinate is #y.
```

The y coordinate is 2.

解构也适用于函数的参数列表

```
#let left = (2, 4, 5)
#let right = (3, 2, 6)
#left.zip(right).map(
  ((a,b)) => a + b
)
```

(5, 6, 11)

在正常作业的左侧。这对于交换变量等非常有用。

```
#{
  let a = 1
  let b = 2
  (a, b) = (b, a)
  [a = #a, b = #b]
}
```

a = 2, b = 1

条件

使用条件，您可以根据是否满足某些条件来显示或计算不同的内容。Typst 支持 和 表达式。当条件的计算结果为 时，条件生成 if 的正文生成的值，否则生成 else 的正文生成的值。ifelse ifelsetrue

```
#if 1 < 2 [
  This is shown
] else [
  This is not.
]
```

This is shown

每个分支都可以有一个代码或内容块作为其主体。

```
- if condition {..}
- if condition [..]
- if condition [..] else {..}
- if condition [..] else if condition {..} else [..]
```

循环

使用循环，您可以重复内容或迭代计算某些内容。Typst 支持两种类型的循环：和循环。前者遍历指定的集合，而后者遍历条件，只要条件保持不变。就像块一样，循环将每次迭代的结果合并为一个值。forwhile

在下面的示例中，for 循环创建的三个句子合并为一个内容值，while 循环中的 length-1 数组合并为一个更大的数组。

```
#for c in "ABC" [
  #c is a letter.
]

#let n = 2
#while n < 10 {
  n = (n * 2) - 1
  (n,)
}
```

A is a letter. B is a letter. C is a letter.

(3, 5, 9, 17)

For 循环可以遍历各种集合：

- `for value in array {..}`
  循环访问数组中的项。此处也可以使用 Let 绑定中描述的解构语法。
- `for pair in dict {..}`
  循环访问字典的键值对。也可以使用 对进行解构。它比因为它不创建所有键值对的临时数组更有效。`for (key, value) in dict {..}``for pair in dict.pairs() {..}`
- `for letter in "abc" {..}`
  循环访问字符串的字符。从技术上讲，它遍历字符串的字素簇。大多数情况下，字形集群只是一个代码点。但是，字形簇可以包含多个代码点，例如标志表情符号。
- `for byte in bytes("😀") {..}`
  循环访问字节，这些字节可以从字符串转换或从文件中读取，而无需编码。每个字节值都是介于 和 之间的整数。`0255`

为了控制循环的执行，Typst 提供了 and 语句。前者提前退出循环，而后者则跳到循环的下一个迭代。breakcontinue

```
#for letter in "abc nope" {
  if letter == " " {
    break
  }

  letter
}
```

abc

循环的主体可以是代码块或内容块：

- `for .. in collection {..}`
- `for .. in collection [..]`
- `while condition {..}`
- `while condition [..]`

领域

您可以使用点表示法来访问值上的字段。所讨论的值可以是：

具有指定键的字典， 具有指定修饰符的符号， 包含指定定义的模块， 内容由具有指定字段的元素组成。可用字段与构造元素时给定的元素函数的参数匹配。

```
#let dict = (greet: "Hello")
#dict.greet \
#emoji.face

#let it = [= Heading]
#it.body \
#it.depth
```

Hello
😀
Heading
1

方法

方法调用是调用范围限定为值类型的函数的便捷方法。例如，我们可以通过以下两种等效方式调用 str.len 函数：

```
#str.len("abc") is the same as
#"abc".len()
```

3 is the same as 3

方法调用的结构是 ，其等效的全函数调用是 。每种类型的文档都列出了其作用域函数。当前无法定义自己的方法。value.method(..args)type(value).method(value, ..args)

方法调用的结构是 ，其等效的全函数调用是 。每种类型的文档都列出了其作用域函数。当前无法定义自己的方法。value.method(..args)type(value).method(value, ..args)

```
#let array = (1, 2, 3, 4)
#array.pop() \
#array.len() \

#("a, b, c"
    .split(", ")
    .join[ --- ])

#"abc".len() is the same as
#str.len("abc")
```

4
3
a — b — c
3 is the same as 3

有一些特殊函数可以修改它们被调用的值（例如 array.push）。这些函数必须以方法形式调用。在某些情况下，当调用该方法只是为了它的副作用时，应该忽略它的返回值（并且不参与联接）。丢弃值的规范方法是使用 let 绑定： 。let = array.remove(1)

模块

您可以将 Typst 项目拆分为多个称为*模块的文件*。一个模块可以通过多种方式引用另一个模块的内容和定义：

- **包括：**
  在路径上评估文件并返回生成的内容。`include "bar.typ"`bar.typ
- **导入：**
  计算路径上的文件，并将生成的模块插入到当前作用域中（不带扩展名的文件名）。您可以使用关键字重命名导入的模块：`import "bar.typ"`bar.typbara`simport "bar.typ" as baz`
- **导入项：**
  在路径处计算文件，提取变量和的值（需要在 中定义，例如通过绑定），并在当前文件中定义它们。将模块中定义的所有变量替换为负载。您可以使用关键字重命名各个项目：
  `import "bar.typ": a, b`bar.typabba`r.typleta, b*asimport "bar.typ": a as one, b as two`

还可以使用模块值来代替路径，如以下示例所示：

```
#import emoji: face
#face.grin
```

😀

包

若要跨项目重用构建基块，还可以创建和导入 Typst 包。包导入被指定为命名空间、名称和版本的三元组。

```
#import "@preview/example:0.1.0": add
#add(2, 7)
```

9

命名空间包含社区共享的包。您可以在 Typst Universe 上找到所有可用的社区包。preview

如果您在本地使用 Typst，您还可以创建自己的系统本地包。有关此内容的更多详细信息，请参阅包存储库。

运营商

下表列出了所有可用的一元运算符和二元运算符，包括效果、arity（一元、二进制）和优先级（绑定越强）。

算子	影响	阿里蒂	优先
-	否定	元	7
+	无效果（为对称而存在）	元	7
*	乘法	二元的	6
/	划分	二元的	6
+	加法	二元的	5
-	减法	二元的	5
==	检查相等性	二元的	4
!=	检查不等式	二元的	4
<	检查小于	二元的	4
<=	检查小于或等于	二元的	4
>	检查大于	二元的	4
>=	检查大于或等于	二元的	4
in	检查是否在集合中	二元的	4
not in	检查是否不在集合中	二元的	4
not	逻辑上的“不是”	元	3
and	短路逻辑“和”	二元的	3
or	短路逻辑“或	二元的	2
=	分配	二元的	1
+=	添加赋值	二元的	1
-=	减法-赋值	二元的	1
*=	乘法赋值	二元的	1
/=	分部分配	二元的	1

造型
上一页

上下文
下一页