语境

有时，我们希望创建对其在文档中的位置做出反应的内容。这可以是取决于配置的文本语言的本地化短语，也可以是像标题编号这样简单的东西，它根据前面有多少个标题打印正确的值。然而，Typst 代码并不直接知道它在文档中的位置。源文本开头的某些代码可能会产生最终出现在文档后面的内容。

为了生成对其周围环境做出反应的内容，我们必须专门指示 Typst：我们使用关键字来执行此操作 context，该关键字位于表达式之前并确保它是根据其环境的知识进行计算的。作为回报，上下文表达式本身最终变得不透明。我们无法在代码中直接访问它的任何结果，正是因为它是上下文相关的：没有一个正确的结果，文档的不同位置可能有多个结果。因此，依赖于上下文数据的所有内容都必须发生在上下文表达式内部。

除了显式上下文表达式之外，上下文还在某些位置隐式建立，这些位置也知道它们在文档中的位置：显示规则提供上下文 1 和大纲中的编号，例如，还提供正确的上下文来解析计数器。

风格背景

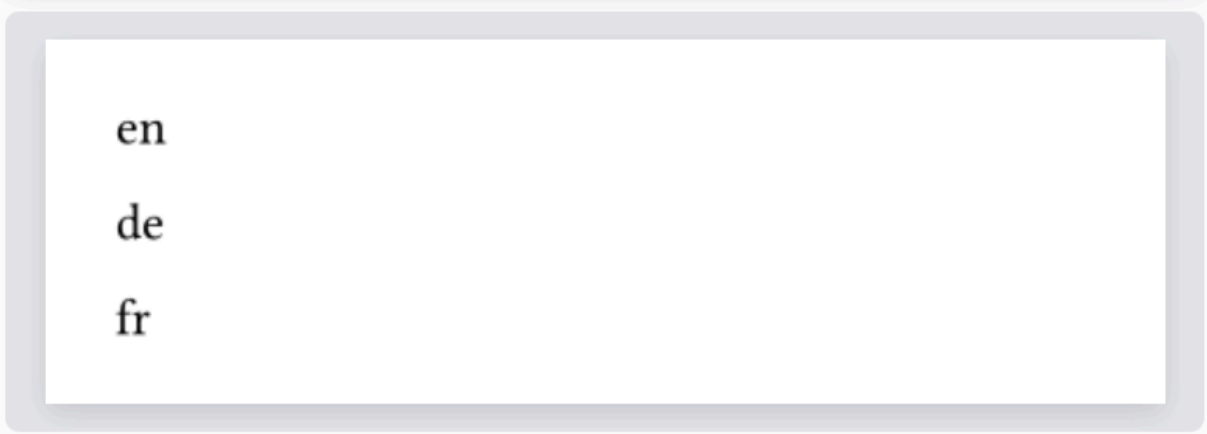
通过设置规则，我们可以调整部分或整个文档的样式属性。如果没有已知的上下文，我们就无法访问这些内容，因为它们可能会在文档的整个过程中发生变化。当上下文可用时，我们只需将它们作为相应元素函数上的字段进行访问即可检索它们。

```
#set text(lang: "de")
#context text.lang
```

de

如上所述，上下文表达式对其所处的不同环境做出反应。在下面的示例中，我们创建一个上下文表达式，将其存储在 value 变量中并多次使用它。每次使用都会对当前环境做出适当的反应。

```
#let value = context text.lang
#value

#set text(lang: "de")
#value

#set text(lang: "fr")
#value
```



至关重要的是，在创建后，内容就变成了我们无法窥视的 value 不透明内容。它只有在放置在某个地方时才能得到解决，因为只有这样才能知道上下文。上下文表达式的主体可以被计算零次、一次或多次，具体取决于它被放入多少个不同的位置。

位置背景

上下文不仅可以让我们访问设置的规则值。它还可以让我们知道当前在文档中的位置（相对于其他元素）以及绝对在页面上的位置。我们可以使用此信息在不同文档部分之间创建非常灵活的交互。这支撑了标题编号、目录或依赖于章节标题的页眉等功能。

有些函数例如 counter.get 隐式访问当前位置。在下面的示例中，我们要检索标题计数器的值。由于它在整个文档中都会发生变化，因此我们需要首先输入上下文表达式。然后，我们用来 get 检索计数器的当前值。该函数从上下文访问当前位置以解析计数器值。计数器有多个级别，并 get 返回一个包含已解析数字的数组。因此，我们得到以下结果：

```
#set heading(numbering: "1.")

= Introduction
#lorem(5)

#context counter(heading).get()

= Background
#lorem(5)

#context counter(heading).get()
```

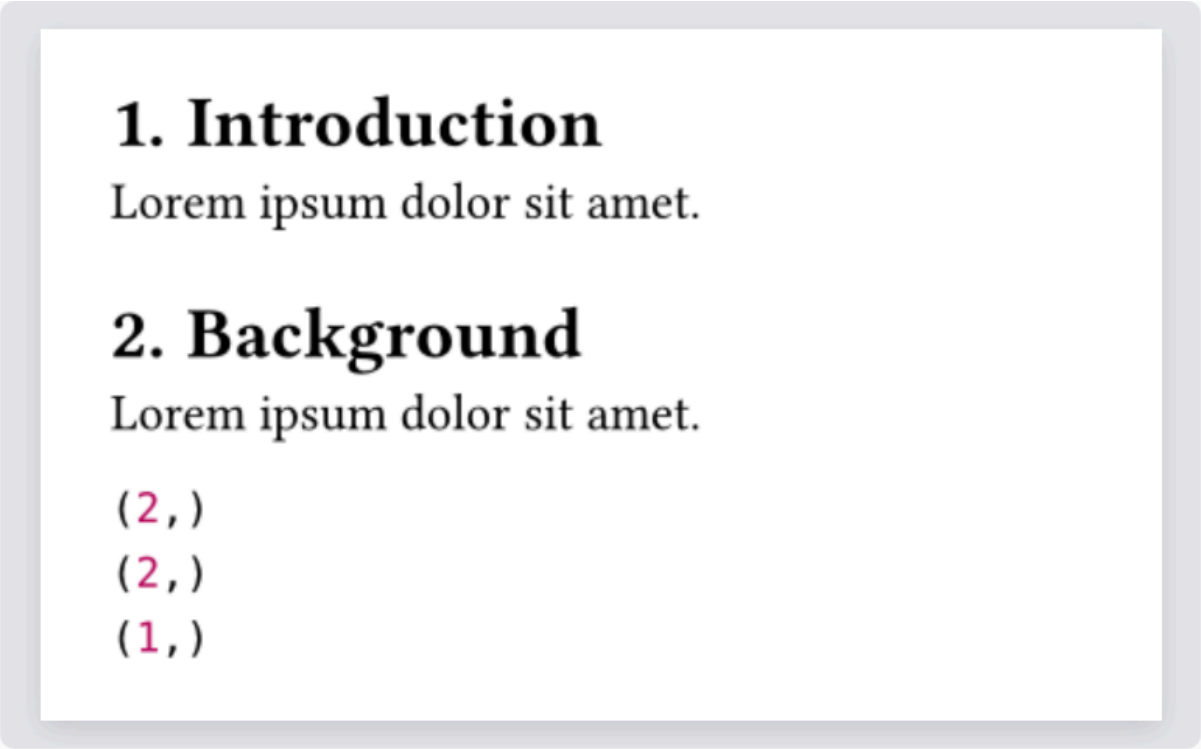
为了更加灵活，我们还可以使用该 here 函数直接从上下文中提取当前位置。下面的示例演示了这一点：

– 我们首先有，它像以前一样解析为。counter(heading).get()(2,)

– 然后我们使用更强大的 counter.atwith here，它的组合相当于 get，从而得到。(2,)

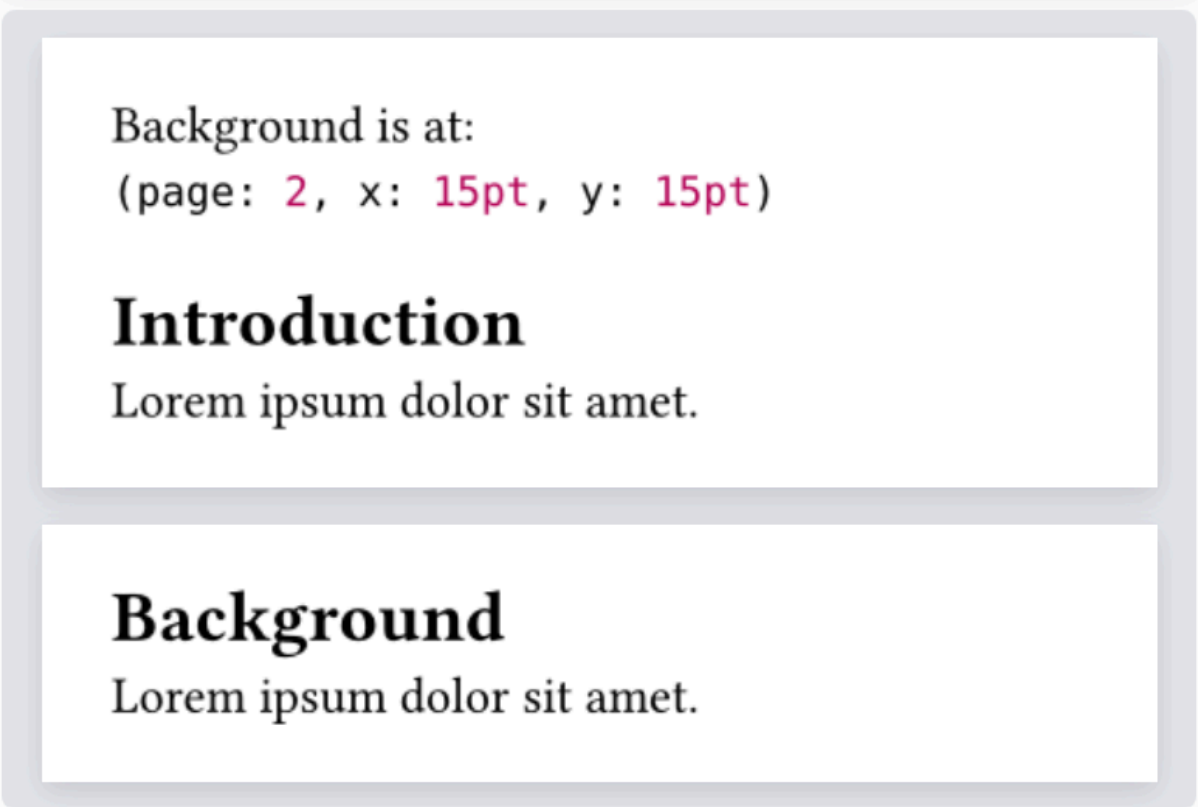
– 最后，我们使用 at 标签来检索文档中不同位置的计数器值，在我们的例子中是介绍标题。这产生. Typst 的上下文系统为我们提供了时间旅行能力，并让我们能够检索文档中任何位置的任何计数器和状态的值。(1,)

```
#set heading(numbering: "1.")

= Introduction <intro>
#lorem(5)

= Background <back>
#lorem(5)

#context [
  #counter(heading).get() \
  #counter(heading).at(here()) \
  #counter(heading).at(<intro>)
]
```



如前所述，我们还可以使用上下文来获取元素在页面上的物理位置。我们使用该函数执行此 locate 操作，其工作原理类似于 counter.at：它采用一个位置或其他选择器来解析为唯一元素（也可以是标签）并返回该元素在页面上的位置。

```
Background is at: \
#context locate(<back>).position()

= Introduction <intro>
#lorem(5)
#pagebreak()

= Background <back>
#lorem(5)
```



还有其他一些函数也利用了位置上下文，其中最为突出的是 query。查看内省类别以获取有关这些内容的更多详细信息。

嵌套上下文

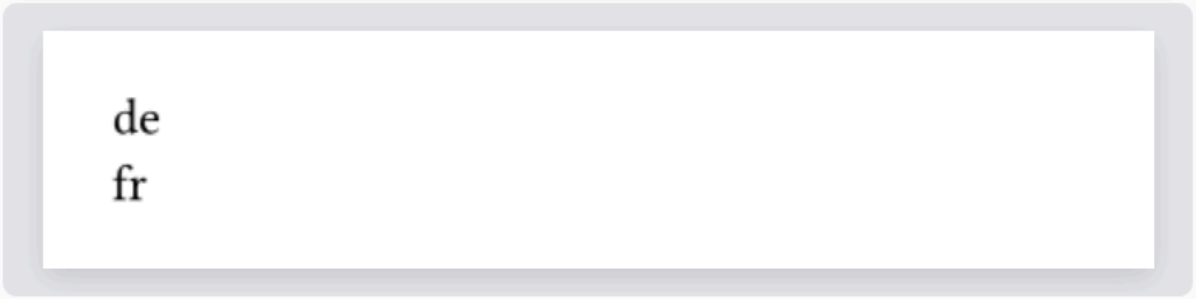
还可以从嵌套在上下文块中的函数调用内访问上下文。在下面的示例中，foo 它本身成为一个上下文函数，就像 to-absoluteis 一样。

```
#let foo() = 1em.to-absolute()
#context {
  foo() == text.size
}
```

```
true
```

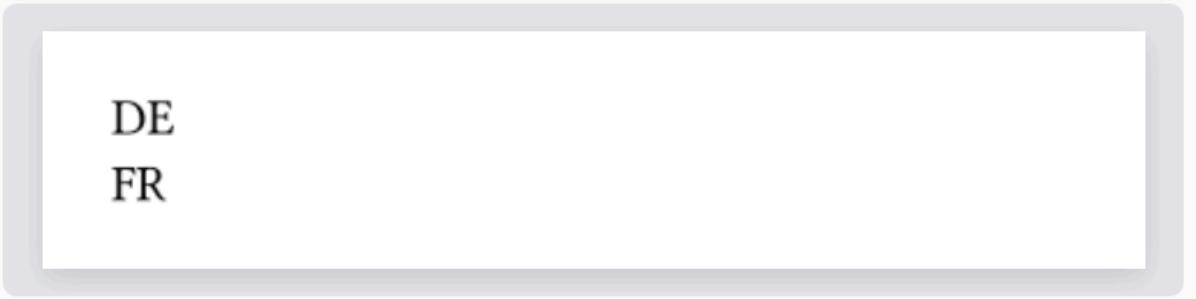
上下文块可以嵌套。然后，上下文代码将始终访问最内部的上下文。下面的示例演示了这一点：第一个 text.lang 将访问外部上下文块的样式，因此，它将看不到的效果。然而，第二个周围的嵌套上下文块在设置的规则之后开始，因此将显示其效果。set text(lang: “fr”)text.lang

```
#set text(lang: "de")
#context [
  #set text(lang: "fr")
  #text.lang \
  #context text.lang
]
```



text.lang 您可能想知道为什么 Typst 在计算上例中的第一个时忽略 French set 规则。原因是，在一般情况下，Typst 无法知道将应用的所有样式，因为设定的规则可以在构建后应用于内容。下面，text.lang 是在应用模板函数时已经计算出来的。因此，它不可能知道模板中的语言更改为法语。

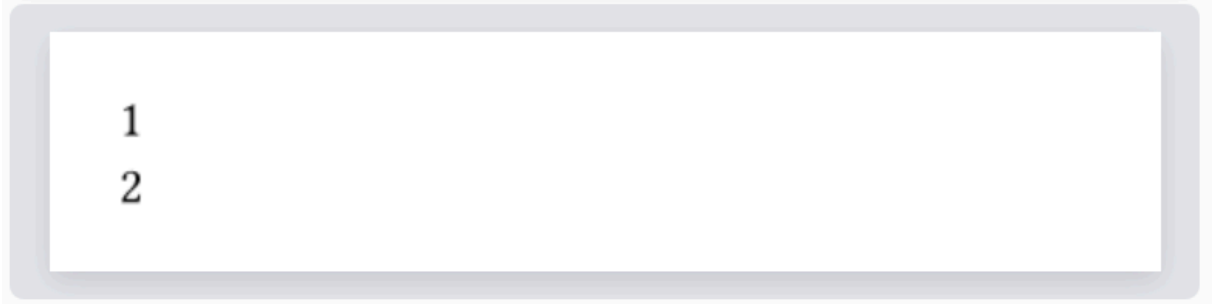
```
#let template(body) = {
  set text(lang: "fr")
  upper(body)
}

#set text(lang: "de")
#context [
  #show: template
  #text.lang \
  #context text.lang
]
```



text.lang 然而，第二个确实对语言变化做出反应，因为对其周围上下文块的评估被推迟，直到它的样式已知为止。这说明了为上下文选择正确插入点以访问精确正确样式的重要性。

对于位置上下文来说也是如此。下面，第一个调用将访问外部上下文块，因此不会看到第二个调用访问内部上下文的效果，因此会看到它。c.display()c.update(2)c.display()

```
#let c = counter("mycounter")
#c.update(1)
#context [
  #c.update(2)
  #c.display() \
  #context c.display()
]
```



编译器迭代

为了解决上下文交互，Typst 编译器会多次处理您的文档。例如，为了解决 locate 调用，Typst 首先提供占位符位置，布局文档，然后使用完成布局中的已知位置重新编译。采用相同的方法来解析计数器、状态和查询。在某些情况下，Typst 甚至可能需要两次以上的迭代来解决所有问题。虽然这有时是必要的，但它也可能是滥用上下文功能（例如 state）的迹象。如果 Typst 在五次尝试内无法解决所有问题，它将停止并输出警告“布局在 5 次尝试内未收敛”。

非常细心的读者可能已经注意到，并非上面介绍的所有函数实际上都使用了当前位置。虽然肯定取决于它，但例如，却并非如此。然而，它仍然需要上下文。虽然它的值在一次编译迭代中始终相同，但它可能会在多次编译迭代过程中发生变化。如果可以直接在模块的顶层调用它，则整个模块及其导出可能会在多次编译器迭代的过程中发生变化，这是不可取的。counter(heading).get()-counter(heading).at()

[1] 目前，所有显示规则都提供样式上下文，但只有可定位元素上的显示规则才提供位置上下文。

脚本编写
上一页

基础
下一页