



The Raindrop-Blue Book

Typst 中文教程

Myriad-Dreamin 等著

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise

transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
- (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
- (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect,

special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

致谢

本文档的主要部分由 [Myriad Dreamin](#) 编写。

[OrangeX4](#) 参与了基本教程部分的勘误。

[GiggleDing](#) 编写了《参考：常用数学符号》。

在此表示感谢。

目录

致谢	5
导引	8
配置 Typst 运行环境	8
如何阅读本书	11
许可证	11
Part.1 基础教程 — 排版 I	12
1.1. 初识标记模式	13
1.2. 初识脚本模式	19
Part.2 基础教程 — 脚本 I	30
2.1. 常量与变量	31
2.2. 块与表达式	41
2.3. 内容与样式	49
Part.3 基础教程 — 排版 II	58
3.1. 度量与布局	59
3.2. 色彩与图表	73
Part.4 基础教程 — 脚本 II	77
4.1. 文件与模块	78
4.2. 使用外部库	81
4.3. 多文件文档	84
Part.5 基础教程 — 排版 III	85
5.1. Typst 架构与原理	86
5.2. 查询文档状态	94
5.3. 维护文档状态	100
Part.6 基础教程 — 附录 I	104
6.1. 参考：语法示例检索表	105
6.2. 参考：函数表（字典序）	113
6.3. 参考：常用数学符号	121
Part.7 基础参考	125
7.1. 参考：基本类型	126
7.2. 参考：内置类型	135
7.3. 参考：时间类型	136
7.4. 参考：数据读写与数据处理	137
7.5. 参考：数据读写与数据处理	138
7.6. 参考：数值计算	139
7.7. 参考：数学模式	140
7.8. 参考：导入和使用参考文献	141
7.9. 参考：WASM 插件	142
Part.8 进阶教程 — 排版 IV	143
8.1. 中文排版	144
Part.9 进阶教程 — 专题	145
9.1. 编写一篇数学文档	146
9.2. 制作一个组件库	147
9.3. 制作一个外部插件	148
9.4. 在 Typst 内执行 Js、Python、Typst 等	149
9.5. 制作一个书籍模板	150
9.6. 制作一个 CV 模板	151
9.7. 制作一个 IEEE 模板	152

Part.10 进阶教程 — 公式和定理	153
10.1. 化学方程式	154
10.2. 伪算法	155
10.3. 定理环境	156
Part.11 进阶教程 — 杂项	157
11.1. 字体设置	158
11.2. 自定义代码高亮规则	159
11.3. 自定义代码主题	160
11.4. 读取外部文件和文本处理	161
Part.12 进阶教程 — 绘制图表	162
12.1. 制表	163
12.2. 立体几何	164
12.3. 拓扑图	166
12.4. 统计图	167
12.5. 状态机	168
12.6. 电路图	169
Part.13 进阶教程 — 附录 II	170
13.1. 参考：语法示例检索表 II	171
13.2. 演示文稿（PPT）	172
13.3. 论文模板	173
13.4. 书籍模板	174
Part.14 进阶参考	175
14.1. 参考：计数器和状态	176
14.2. 参考：长度单位	177
14.3. 参考：布局函数	178
14.4. 参考：表格	179
14.5. 参考：文档大纲	180

导引

本教程面向所有 Typst 用户，循序渐进，供以中文为母语的 Typst 语言初学者和爱好者查阅和参考。本教程希望弥补 Typst 相关资料的缺失，作为官方文档的补充，帮助大家入门和学习 Typst。

本教程的首要定位是，即便你没有学习过任何编程语言，也能通过在本教程中学到的知识上手使用 Typst，在日常生活中使用 Typst 编写各式各样的文档。同时，本教程也会总结在使用 Typst 编写文档的过程中遇到的一系列问题。

为什么学习 Typst?

在开始之前，让我们考虑一下 Typst 到底是什么，以及我们在什么时候应该使用它。Typst 是一种用于排版文档的标记语言，它旨在易于学习、快速且用途广泛。Typst 输入带有标记的文本文件，并将其输出为 PDF 格式。

Typst 是撰写长篇文本（如论文、文章、书籍、报告和作业）的极佳选择。并且，Typst 非常适合书写包含数学公式的文档，例如数学、物理和工程领域的论文。此外，由于其强大的样式和自动化功能，它是编写具有相同样式的一系列文档（例如丛书）的绝佳选择。

阅读本教程前，您需要了解的知识

你不需要了解任何前置知识，所需的仅仅是安装 Typst，并跟着本教程的在线示例一步步学习。

其他参考资料

- [官方文档翻译 - 入门教程](#)
- [非官方中文用户指南](#)
- [官方文档翻译 - 参考](#)
- [Typst Example Books](#)

配置 Typst 运行环境

使用官方的 webapp（推荐）

官方提供了[在线且免费](#)的多人协作编辑器。该编辑器会从远程下载 WASM 编译器，并在你的浏览器内运行编辑器，为你提供预览服务。

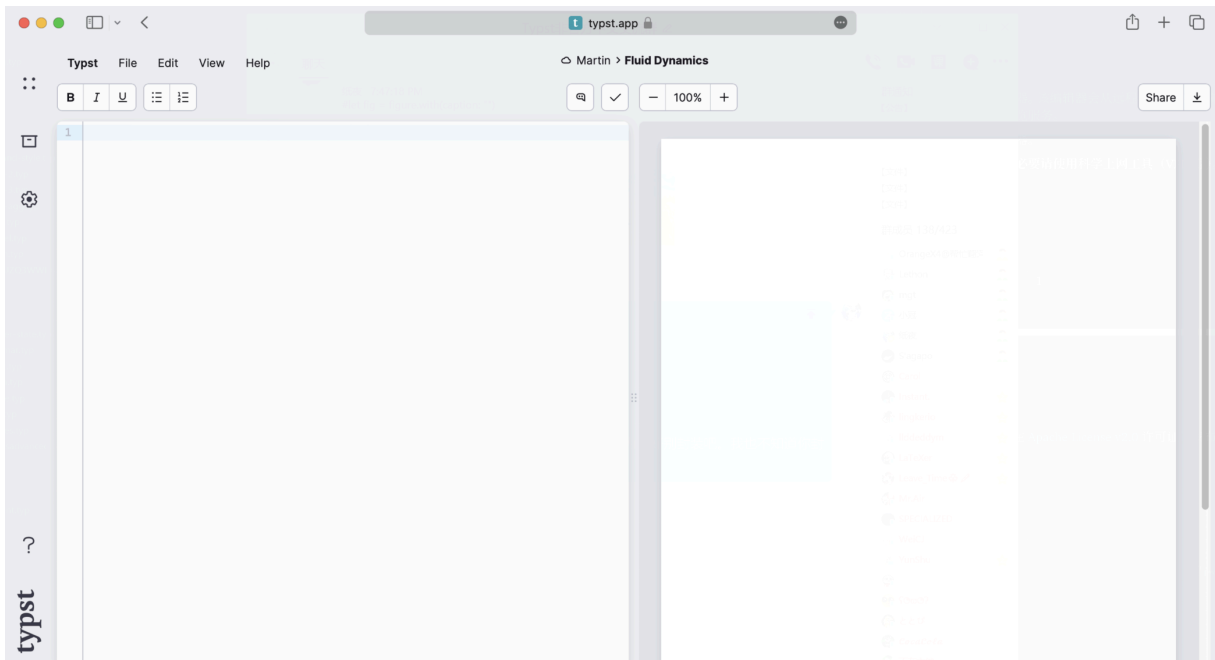


图 1 本书作者在网页中打开 webapp 的瞬间

该编辑器的速度相比许多 LaTeX 编辑器都有显著优势。这是因为：

- 你的大部分编辑操作不会导致阻塞的网络请求。
- 所有计算都在本地浏览器中运行。
- 编译器本身性能极其优越。

你可以注册一个账户并开箱即用该编辑器。

注意：你需要检测你的网络环境，如有必要请使用科学上网工具（VPN 等）。

使用 VSCode 编辑（推荐）

打开扩展界面，搜索并安装 tinymist 插件。它会为你提供语法高亮，代码补全，代码格式化，即时预览等功能。

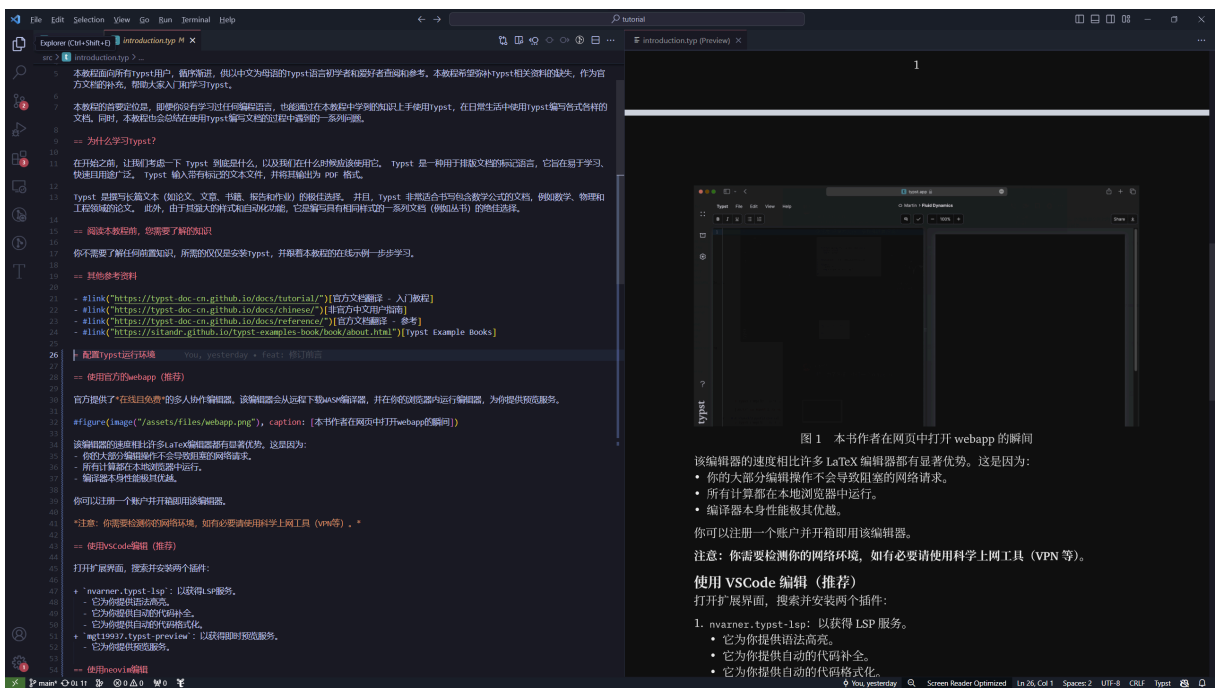


图 1 本书作者在网页中打开 webapp 的瞬间

该编辑器的速度相比许多 LaTeX 编辑器都有显著优势。这是因为：

- 你的大部分编辑操作不会导致阻塞的网络请求。
- 所有计算都在本地浏览器中运行。
- 编译器本身性能极其优越。

你可以注册一个账户并开箱即用该编辑器。

注意：你需要检测你的网络环境，如有必要请使用科学上网工具（VPN 等）。

使用 VSCode 编辑（推荐）

打开扩展界面，搜索并安装两个插件：

1. nvimernest.typst-lsp：以获得 LSP 服务。
 - 它为你提供语法高亮。
 - 它为你提供自动的代码补全。
 - 它为你提供自动的代码格式化。

图 2 本书作者在 VSCode 中预览并编辑本文的瞬间

该编辑器的速度相比 webapp 略慢。这是因为：

- 编译器与预览程序是两个不同的进程，有通信开销。
- 用户事件和文件 IO 有可能增加 E2E 延时。

但是：

- 大部分时间下，你感受不到和 webapp 之间的性能差异。Typst 真的非常快。
- 你可以离线编辑 Typst 文档，无需任何网络连接，例如本书的部分章节是在飞机上完成的。
- 你可以在文件系统中管理你所有的源代码，实施包括但不限于使用 git 等源码管理软件等操作。

使用 neovim 编辑

细节可以问群友。

自去年年底，该编辑器与 VSCode 一样，已经可以有很好的 Typst 编辑体验。

使用 Emacs 编辑

可以问群友。

自今年年初，该编辑器与 VSCode 一样，已经可以有很好的 Typst 编辑体验。

使用 typst-cli 与 PDF 阅读器

Typst 的 CLI 可从不同的来源获得：

- 你可以获得最新版本的 Typst 的源代码和预构建的二进制文件来自[发布页面](#)。下载适合你平台的存档并将其放在“PATH”中的目录中。及时了解未来发布后，你只需运行 `typst update` 即可。
- 你可以通过不同的包管理器安装 Typst。请注意，包管理器中的版本可能落后于最新版本。
 - Linux：查看 [Typst on Repology](#)。
 - macOS： `brew install typst`。
 - Windows： `winget install --id Typst.Typst`。
- 如果您安装了 Rust 工具链，您还可以安装最新开发版本 `cargo --git https://github.com/typst/typst --locked typst-cli`。请注意，这将是一个“夜间”版本，可能已损坏或尚未正确记录。
- Nix 用户可以将 typst 包与 `nix-shell -p typst` 一起使用，或者构建并使用 `nix run github:typst/typst -- --version` 运行前沿版本。
- Docker 用户可以运行预构建的镜像 `docker run -it ghcr.io/typst/typst:latest`。

安装好 CLI 之后，你就可以在命令行里运行 Typst 编译器了：

```
# Creates `file.pdf` in working directory.
typst compile file.typ

# Creates PDF file at the desired path.
typst compile path/to/source.typ path/to/output.pdf
```

为了提供预览服务，你需要让 Typst 编译器运行在监视模式（watch）下：

```
# Watches source files and recompiles on changes.
typst watch file.typ
```

当你启动 Typst 编译器后，你可以使用你喜欢的编辑器编辑 Typst 文档，并使用你喜欢的 PDF 阅读器打开编译好的 PDF 文件。PDF 阅读器推荐使用：

- 在 Windows 下使用 [Sumatra PDF](#)。

各 Typst 运行环境的比较

名称	编辑器	编译器环境	预览方案	是否支持即时编译	语言服务	备注
WebAPP	Code Mirror	wasm	渲染图片	是	优秀	开箱即用
VSCode	VSCode	native	webview	是	良好	简单上手，定制性差
neovim	neovim	native	webview	是	良好	不易安装，定制性好

Emacs	Emacs	native	webview	是	良好	难以安装
typst-cli	任意编辑器	native	任意 PDF 阅读器	否	无	简单上手，灵活组合

如何阅读本书

本书主要分为三个部分：

1. 教程：介绍 Typst 的语法、原理和理念，以助你深度理解 Typst。
2. 参考：完整介绍 Typst 中内置的函数和各种外部库，以助你广泛了解 Typst 的能力，用 Typst 实现各种排版需求。
3. 专题等：与参考等章节的区别是，每个专题都专注解决一类问题，而不会讲解函数的用法。

每个部分都包含大量例子，它们各有侧重：

- 教程中的例子希望切中要点，而避免冗长的全面的展示特性。
- 参考中的例子希望讲透元素和函数的使用，例子多选自过去的一年中大家常问的问题。
- 专题中的例子往往有连贯性，从前往后带你完成一系列专门的问题。专题中的例子假设你已经掌握了相关知识，只以最专业的代码解决该问题。

i 本书会随时夹带一些“Pro Tip”。这些“Pro Tip”由蓝色框包裹。它们告诉你一些较难理解的知识点。

你可以选择在初次阅读时跳过这些框，而不影响对正文的理解。但建议你在阅读完整本书后回头观看所不理解的那些“Pro Tip”。

以下是推荐的阅读方法：

1. 首先阅读《基础教程》排版 I 的两篇文章，既《初识标记模式》和《初识脚本模式》。

经过这一步，你应该可以像使用 Markdown 那样，编写一篇基本不设置样式的文档。同时，这一步的学习难度也与学习完整 Markdown 语法相当。

2. 接着，阅读《基础教程》脚本 I 的三篇文章。

经过这一步，你应该已经基本入门了 Typst 的标记和脚本。此时，你和进阶用户的唯一区别是，你还不太会使用高级的样式配置。推荐：

- 如果你熟练阅读英语，推荐主要参考[官方文档](#)的内容。
- 否则，推荐继续阅读本书剩下的内容，并参考[非官方中文文档](#)的内容。

有什么问题？

- 本书只有《基础教程》完成了校对和润色，后续部分还非常不完善。甚至《基础教程》部分还有待改进。

- [非官方中文文档](#)是 GPT 机翻后润色的的结果，有可能错翻、漏翻，内容也可能有些许迟滞。

3. 接着，阅读《基础教程》脚本 II 和排版 III 的五篇文章。

这两部分分别介绍了如何使用 Typst 的模块机制与状态机制。模块允许你将文档拆分为多个文件。状态则类似于其他编程语言中的全局变量的概念，可用于收集和维护数据。

4. 最后，同时阅读《基础参考》和《进阶教程》。你可以根据你的需求挑选《基础参考》的部分章节阅读。即便不阅读任何《基础参考》中的内容，你也可以继续阅读《进阶教程》。

经过这一步，你应该已经完全学会了目前 Typst 的所有理念与内容。

许可证

typst-tutorial-cn 所有的源码和文档都在 [Apache License v2.0](#) 许可证下发布。



Part.1 基础教程 — 排版 I



初识标记模式

Typst 是一门简明但强大的现代排版语言，你可以使用简洁直观的语法排版出好看的文档。

Typst 希望你总是尽可能少的配置样式，就获得一个排版精良的文档。多数情况下，你只需要专心撰写文档，而不需要在文档内部对排版做任何更复杂的调整。

得益于此设计目标，为了使你可以用 Typst 编写一篇基本文档，本节仍只需涉及最基本的语法。哪怕只依靠这些语法，你已经可以编写满足很多场合需求的文档。

段落

普通文本默认组成一个个段落。

我是一段文本

我是一段文本

另起一行文本不会产生新的段落。为了创建新的段落，你需要空至少一行。

轻轻的我走了，
正如我轻轻的来；

我轻轻的招手，
作别西天的云彩。

轻轻的我走了， 正如我轻轻的来；
我轻轻的招手， 作别西天的云彩。

缩进并不会产生新的空格：

轻轻的我走了，
正如我轻轻的来；

我轻轻的招手，
作别西天的云彩。

轻轻的我走了， 正如我轻轻的来；
我轻轻的招手， 作别西天的云彩。

注意：如下图的蓝框高亮所示，另起一行会引入一个小的空格。该问题会在未来修复。

轻轻的我走了，
正如我轻轻的来；

轻轻的我走了，正如我轻轻的来；

轻轻的我走了， 正如我轻轻的来；
轻轻的我走了，正如我轻轻的来；

标题

你可以使用一个或多个**连续的**「等于号」(=) 开启一个标题。

= 一级标题

我走了。

== 二级标题

我来了。

=== 三级标题

我走了又来了。

一级标题

我走了。

二级标题

我来了。

三级标题

我走了又来了。

等于号的数量恰好对应了标题的级别。一级标题由一个「等于号」开启，二级标题由两个「等于号」开启，以此类推。

注意：正如你所见，标题会强制划分新的段落。

❗ 使用 `show` 规则可以改变“标题会强制划分新的段落”这个默认规则。

```
#show heading.where(level: 3): box
== 一级标题
我走了。

=== 三级标题
我走了又来了。
```

一级标题

我走了。

三级标题 我走了又来了。

着重和强调语义

有许多与「等于号」类似的语法标记。当你以相应的语法标记文本内容时，相应的文本就被赋予了特别的语义和样式。

❗ 与 HTML 一样，Typst 总是希望语义先行。所谓语义先行，就是在编写文档时总是首先考虑标记语义。所有样式都是附加到语义上的。

例如在英文排版中，`strong` 的样式是加粗，`emph` 的样式是倾斜。你完全可以在中文排版中为它们更换样式。

```
#show strong: content => {
  show regex("\p{Hani}"): it =>
    box(place(text(".", size: 1.3em), dx:
      0.3em, dy: 0.5em) + it)
  content.body
}
*中文排版的着重语义用加点表示。*
```

中文排版的着重语义用加点表示。

与许多标记语言相同，Typst 中使用一系列「定界符」(delimiter) 规则确定一段语义的开始和结束。为赋予语义，需要将一个「定界符」置于文本之前，表示某语义的开始；同时将另一个「定界符」置于文本之后，表示该语义的结束。

例如，「星号」(*) 作为定界符赋予所包裹的一段文本以「着重语义」(strong semantics)。

着重语义：这里有一个***重点！***

着重语义：这里有一个**重点！**

与「着重语义」类似，「下划线」(_) 作为定界符将赋予「强调语义」(emphasis semantics)：

强调语义：_emphasis_

强调语义：emphasis

着重语义一般比强调语义语气更重。着重和强调语义可以相互嵌套：

着重且强调：***_strong emph_*** 或 **_*strong emph*_**

着重且强调：**strong emph** 或 **strong emph**

注意：中文排版一般不使用斜体表示着重或强调。

(计算机) 代码片段

Typst 的「代码片段」(raw block) 标记语法与 Markdown 完全相同。

配对的「反引号」(`) 包裹一段内容，表示内容为「代码片段」。

短代码片段：`code`

短代码片段：code

有时候你希望允许代码内容包含换行或「反引号」。这时候，你需要使用至少连续三个「反引号」组成定界符标记「代码片段」：

使用四个反引号包裹： ```` ```` ```` ````

使用四个反引号包裹：` ``

一段有高亮的代码片段: ````javascript function
uninstallLaTeX {}````

一段有高亮的代码片段：`function
uninstallLaTeX {}`

另一段有高亮的代码片段：包含反引号的长代码片段：

1. 使用至少连续三个「反引号」，即其需为长代码片段。

非块代码片段: ````rust trait World````

```
代码片段: ``js
function fibonacci(n) {
  return n <= 1 ? `...`;
}
``
```

块代码片段:

```
function fibonacci(n) {  
  return n <= 1 ? : '...';  
}
```

Typst 的列表语法与 Markdown 非常类似，但不完全相同。

- 一级列表项 1

- 一级列表项 1

+ 一级列表项 1

1. 一级列表项 1

- 一级列表项 1
 - 二级列表项 1.1
 - 三级列表项 1.1.1
 - 二级列表项 1.2
- 一级列表项 2
 - 二级列表项 2.1

- 一级列表项 1
 - ▶ 二级列表项 1.1
 - 三级列表项 1.1.1
 - ▶ 二级列表项 1.2
- 一级列表项 2
 - ▶ 二级列表项 2.1

15


```

+ 一级列表项 1
- 二级列表项 1.1
  + 三级列表项 1.1.1
- 二级列表项 1.2
+ 一级列表项 2
- 二级列表项 2.1

```

```

1. 一级列表项 1
  • 二级列表项 1.1
    1. 三级列表项 1.1.1
  • 二级列表项 1.2
2. 一级列表项 2
  • 二级列表项 2.1

```

和 Markdown 相同，Typst 同样允许使用显式的编号 1. 开启列表。这方便对列表继续编号。

```

1. 列表项 1
1. 列表项 2

```

```

1. 列表项 1
1. 列表项 2

```

```

1. 列表项 1
+ 列表项 2

```

列表间插入一段描述。

```

3. 列表项 3
+ 列表项 4
+ 列表项 5

```

```

1. 列表项 1
2. 列表项 2
列表间插入一段描述。
3. 列表项 3
4. 列表项 4
5. 列表项 5

```

转义序列

你有时希望直接展示标记符号本身。例如，你可能想直接展示一个「下划线」，而非使用强调语义。这时你需要利用「转义序列」(escape sequences) 语法：

在段落中直接使用下划线 >_<!

在段落中直接使用下划线 >_<!

遵从许多编程语言的习惯，Typst 使用「反斜杠」(\) 转义特殊标记。下表给出了部分可以转义的字符：

代码	\\	\/	\[\]	\{	\}	\<	\>	\u{cccc}
效果	\	/	[]	{	}	<	>	책
代码	\(\)	\#	*	_	\+	\=	\~	\u{cCCc}
效果	()	#	*	_	+	=	~	책
代码	\`	\\$	\"	\'	\@	\a	\A		\u{2665}
效果	`	\$	"	'	@	a	A		♥

以上大部分「转义序列」都紧跟单个字符，除了表中的最后一列。

表中的最后一列所展示的 \u{unicode} 语法被称为 Unicode 转义序列，也常见于各种语言。你可以通过将 unicode 替换为 [Unicode 码位](#) 的值，以输出该特定字符，而无需 [输入法支持](#)。例如，你可以这样输出一句话：

```

\u{9999}\u{8FA3}\u{725B}\u{8089}\u{7C89}
\u{597D}\u{5403}\u{2665}

```

香辣牛肉粉好吃♥

诸多「转义序列」无需死记硬背，你只需要记住：

1. 如果其在 Typst 中已经被赋予含义，请尝试在字符前添加一个「反斜杠」。
2. 如果其不可见或难以使用输入法获得，请考虑使用 \u{unicode}。

输出换行符

输出换行符是一种特殊的「转义序列」，它使得文档输出换行。

「反斜杠」后紧接一个任意「空白字符」(whitespace)，表示在此处主动插入一个段落内的换行符：

转义空格可以换行 \ 转义回车也可以换行 \ 换行!

转义空格可以换行
转义回车也可以换行
换行!

空白字符可以取短空格 (U+0020)、长空格 (U+3000)、回车 (U+000D) 等。

速记符号

在「标记模式」(markup mode) 下, 一些符号需要用特殊的符号组合打出, 这种符号组合被称为「速记符号」(shorthand)。它们是:

空格 (U+0020) 的「速记符号」是「波浪号」(~):

AB v.s. A~B

AB v.s. A B

连接号 (en dash, U+2013) 的「速记符号」由两个连续的「连字号」(-) 组成:

北京--上海路线的列车正在到站。

北京-上海路线的列车正在到站。

省略号的「速记符号」由三个连续的「点号」(.) 组成:

真的假的.....

真的假的.....

完整的速记符号列表参考 [Typst Symbols](#)。

注释

Typst 的「注释」(comment) 直接采用 C 语言风格的注释语法, 有两种表示方法。

第一种写法是将注释内容放在两个连续的「斜杠」(/) 后面, 从双斜杠到行尾都属于「注释」。

```
// 这是一行注释  
一行文本 // 这也是注释
```

一行文本

与代码片段的情形类似, Typst 也提供了另外一种可以跨行的「注释」, 形如/*...*/。

你没有看见/* 混入其中 */注释

你没有看见注释

值得注意的是, Typst 会将「注释」从源码中剔除后再解释你的文档, 因此它们对文档没有影响。

以下两个段落等价:

```
注释不会  
// 这是一行注释 // 注释内的注释还是注释  
插入换行 // 这也是注释
```

注释不会 插入换行
注释不会 插入换行

```
注释不会  
插入换行
```

以下三个段落等价:

```
注释不会/* 混入其中 */插入空格
```

注释不会插入空格
注释不会插入空格
注释不会插入空格

```
注释不会/*  
混入其中  
*/插入空格
```

```
注释不会插入空格
```

总结

基于《编写一篇基本文档》前半部分掌握的知识，你应该编写一些非常简单的文档。

习题

1. 使源码至少有两行，第一行包含“欲穷千里目，”，第二行包含“更上一层楼。”，但是输出不换行不空格：

```
欲穷千里目，更上一层楼。
```

2. 输出一个「星号」，期望的输出：

```
*
```

3. 插入代码片段使其包含一个反引号，期望的输出：

```
`
```

4. 插入代码片段使其包含三个反引号，期望的输出：

```
```
```

5. 插入行内的“typc”代码，期望的输出：

```
你可以在 Typst 内通过插件 plugin("typst.wasm") 调用 Typst 编译器。
```

6. 在有序列表间插入描述，期望的输出：

```
约法五章。
1. 其一。
2. 其二。
前两条不算。
3. 其三。
4. 其四。
5. 其五。
```

# 初识脚本模式

从现在开始，示例将会逐渐开始出现脚本。不要担心，它们都仅涉及脚本的简单用法。

## 内容块

有时，文档中会出现连续大段的标记文本。

```
从前有座山，山会讲故事，故事讲的是
从前有座山，山会讲故事，故事讲的是
...
```

从前有座山，山会讲故事，故事讲的是  
从前有座山，山会讲故事，故事讲的是  
...

这可行，但稍显麻烦。如下代码则显得更为整洁，它不必为每段都打上着重标记：

```
#strong[
 从前有座山，山会讲故事，故事讲的是

 从前有座山，山会讲故事，故事讲的是

 ...
]
```

从前有座山，山会讲故事，故事讲的是  
从前有座山，山会讲故事，故事讲的是  
...

例中，`#strong[]` 这个内容的语法包含三个部分：

1. `#` 使解释器进入「脚本模式」(code mode)。
2. `strong` 是赋予「着重语义」(strong semantics) 函数。
3. `[]` 作为「内容块」(content block) 标记一段内容，供 `strong` 使用。

本小节首先讲解第三点，即「内容块」语法。

「内容块」的内容使用中括号包裹，如下所示：

```
#[一段文本]#[两段文本]#[三段文本]
```

一段文本两段文本 三段文本

「内容块」不会影响包裹的内容——Typst 仅仅是解析内部代码作为「内容块」的内容。「内容块」也几乎不影响内容的书写。

❗ 唯一的影响是你需要在内容块内部转义右中括号。

```
#[x\]y]
```

x]y

「内容块」的唯一作用是“界定内容”。它收集一个或多个「内容」(content)，以待后续使用。有了「内容块」，你可以**准确指定**一段内容，并用「脚本」(scripting) 加工。

左 # [一段文本] 右 ..... `text`(blue) ..... 左 一段文本 右  
选中内容                      对内容块应用 `text` 函数                      最终效果

所谓「脚本」，就是对原始内容增删查改，进而形成文档的过程描述。因为有了「脚本」，Typst 才能有远超 Markdown 的排版能力，在许多情况下不逊于 LaTeX 排版，将来有望全面超越 LaTeX 排版。

在接下来两小节你将看到 Typst 作为一门编程语言的核心设计，也是进行更高级排版必须要掌握的知识点。由于我们的目标首先仅是**编写一篇基本文档**，我们将会尽可能减少引入更多知识点，仅仅介绍其中最简单常用的语法。

## 解释模式

`#strong[]` 语法第一点提及：`#` 使解释器进入「脚本模式」。

`#[一段文本]`

一段文本

这个「井号」(`#`) 不属于内容块的语法一部分，而是 Typst 中关于「脚本模式」的定界符。

这涉及到 Typst 的编译原理。Typst 程序包含一个解释器，用其从头到尾查看并「解释」(interpret) 你的文档。

其特殊之处在于，解释器还具备多种「解释模式」(interpreting mode)。借鉴了 LaTeX 的文本和数学模式，在不同的「解释模式」下，解释器以不同的语法规则解释你的文档。Typst 中，标记模式的语法更适合你组织文本，代码模式更适合你书写脚本，而数学模式则最适合输入复杂的公式。

## 标记模式

当解释器从头开始解释文档时，其处于「标记模式」(markup mode)，在这个模式下，你可以使用各种记号创建标题、列表、段落……在这个模式下，Typst 语法几乎就和 Markdown 一样。

当其处于「标记模式」，且遇到一个「井号」时，Typst 会立即将后续的一段代码认作「脚本」并执行，即它进入了「脚本模式」(scripting mode)。

## 脚本模式

在「脚本模式」下，你可以转而主要计算各种内容。例如，你可以计算一个算式的「内容」：

`#(1024*1024*8*7*17+1)` 是一个常见素数。

998244353 是一个常见素数。

当处于「脚本模式」时，解释器在适当的时候从「脚本模式」退回为「标记模式」。如下所示，在「脚本模式」下解析到数字 `2` 后解释器回到了「标记模式」：

`#2` 是一个常见素数。

2 是一个常见素数。

Typst 总是倾向于更快地退出脚本模式。

- ❶ 具体来说，你几乎可以认为解释器至多只会解释一个完整的表达式，之后就会立即退出「脚本模式」。

## 以另一个视角看待内容块

「内容块」的内容遵从标记语法。这意味着，当处于「脚本模式」时，你还可以通过「内容块」语法临时返回「标记模式」，以嵌套复杂的逻辑：

`#([== 脚本模式下创建一个标题] + strong[后接一段文本])`

脚本模式下创建一个标题  
后接一段文本

如此反复，Typst 就同时具备了方便文档创作与脚本编写的能力。

- ❷ 能否直接像使用「星号」那样，让「标记模式」直接将中括号包裹的一段作为内容块的内容？这是可以的，但是存在一些问题。例如，人们也常常在正文中使用中括号等标记：

区间  $[1, \infty)$  上几乎所有有理数都可以表示为  $x^x$ ，其中  $x$  是无理数。

区间  $[1, \infty)$  上几乎所有有理数都可以表示为  $x^x$ ，其中  $x$  是无理数。

如此，「标记模式」下默认将中括号解析为普通文本看起来更为合理。

## 数学模式

Typst 解释器一共有三种模式，其中两种我们之前已经介绍。这剩下的最后一种被称为「数学模式」(math mode)。很多人认为 Typst 针对 LaTeX 的核心竞争点之一就是优美的「数学模式」。

Typst 的数学模式如下：

行内数学公式：`$\sum_x$`

行间数学公式：`$\sum_x$`

行内数学公式： $\sum_x$

行间数学公式：

$$\sum_x$$

由于使用「数学模式」有很多值得注意的地方，且「数学模式」是一个较为独立的模式，本书将其单列为一章参考，可选阅读。有需要在文档中插入数学公式的同学请移步《参考：数学模式》。

## 函数和函数调用

这里仅作最基础的介绍。《基本字面量、变量和简单函数》和《复合字面量、控制流和复杂函数》中有对函数和函数调用更详细的介绍。

在 Typst 中，函数与函数调用同样归属「脚本模式」，所以在调用函数前，你需要先使用「井号」让 Typst 先进入「脚本模式」。

与大部分语言相同的是，在调用 Typst 函数时，你可以向其传递以逗号分隔的「值」(value)，这些「值」被称为参数。

四的三次方为 `#calc.pow(4, 3)`。

四的三次方为 64。

这里 `calc.pow` 是内置的幂计算函数，其接受两个参数：

1. 一为 4，为幂的底
2. 一为 3，为幂的指数。

你可以使用函数修饰「内容块」。例如，你可以使用着重函数 `strong` 标记一整段内容：

```
#strong([
 And every _fair from fair_ sometime
 declines,

 By chance, or nature's changing course
 untrimm'd;

 But thy _eternal summer_ shall not fade,

 Nor lose possession of that fair thou
 ow'st;
])
```

**And every fair from fair sometime declines,**

**By chance, or nature's changing course untrimm'd;**

**But thy eternal summer shall not fade,**

**Nor lose possession of that fair thou ow'st;**

虽然示例很长，但请认真观察，它很简单。首先，中括号包裹的是一大段内容。在之前已经学到，这是一个「内容块」。然后「内容块」在参数列表中，说明它是 `strong` 的参数。`strong` 与幂函数没有什么不同，无非是接受了一个「内容块」作为参数。

类似地，`emph` 可以标记一整段内容为强调语义：

```
#emph([
 And every *fair from fair* sometime
 declines,

])
```

*And every fair from fair sometime declines,*

.....

Typst 强调「一致性」(consistency)，因此无论是通过标记还是通过函数，最终效果都必定是一样的。你可以根据实际情况任意组合方式。

## 内容参数的糖

在许多的语言中，所有函数参数必须包裹在函数调用参数列表的「圆括号」之内。

着重语义：这里有一个 `#strong`([重点!])

着重语义：这里有一个**重点!**

但在 Typst 中，如果将内容块作为参数，内容块可以紧贴在参数列表的「圆括号」之后。

着重语义：这里有一个 `#strong()`[重点!]

着重语义：这里有一个**重点!**

特别地，如果参数列表为空，Typst 允许省略多余的参数列表。

着重语义：这里有一个 `#strong`[重点!]

着重语义：这里有一个**重点!**

所以，示例也可以写为：

```
#strong[
 And every _fair from fair_ sometime
 declines,
]

#emph[
 And every *fair from fair* sometime
 declines,
]
```

**And every *fair from fair* sometime declines,**

*And every **fair from fair** sometime declines,*

❶ 函数调用可以后接不止一个内容参数。例如下面的例子后接了两个内容参数：

```
#let exercise(question, answer) =
 strong(question) + parbreak() + answer

#exercise[
 Question: _turing complete_
][
 Answer: Yes, Typst is.
]
```

**Question: *turing complete?***

Answer: Yes, Typst is.

## 文字修饰

现在你可以使用更多的文本函数来丰富你的文档效果。

### 背景高亮

你可以使用 `highlight` 高亮一段内容：

`#highlight`[高亮一段内容]

高亮一段内容

你可以传入 `fill` 参数以改变高亮颜色。

```
#highlight(fill: orange)[高亮一段内容]
// ^^^^^^^^^^^^^ 具名传参
```

高亮一段内容

这种传参方式被称为「具名传参」。

### 修饰线

你可以分别使用 `underline`、`overline`、或 `strike` 为一段内容添加下划线、上划线或中划线（删除线）：

```
平地翻滚: #underline[□□□□□□□] \
施展轻功: #overline[□□□□□□□] \
泥地打滚: #strike[□□□□□□□] \
```

平地翻滚: □□□□□□□□  
施展轻功: □□□□□□□□  
泥地打滚: □□□□□□□□

值得注意的是，被划线内容需要保持相同字体才能保证线段同时处于同一水平高度。

```
#set text(font: ("Linux Libertine",
"Source Han Serif SC"))
下划线效果: #underline[空格 字体不一致] \
#set text(font: "Source Han Serif SC")
下划线效果: #underline[空格 字体一致] \
```

下划线效果: 空格 字体不一致  
下划线效果: 空格 字体一致

该限制可能会在将来被解除。

`underline` 有一个很有用的 `offset` 参数，通过它你可以修改下划线相对于「基线」的偏移量：

```
#underline(offset: 1.5pt,
underline(offset: 3pt, [双下划线]))
```

双下划线

如果你更喜欢连贯的下划线，你可以设置 `evade` 参数，以解除驱逐效果。

```
带驱逐效果: #underline[Language] \
不带驱逐效果: #underline(evade: false)
[Language]
```

带驱逐效果: Language  
不带驱逐效果: Language

## 上下标

你可以分别使用 `sub` 或 `super` 将一段文本调整至下标位置或上标位置：

```
下标: 威严满满 #sub[抱头蹲防] \
上标: 香風とうふ店 #super[TM] \
```

下标: 威严满满<sub>抱头蹲防</sub>  
上标: 香風とうふ店<sup>TM</sup>

你可以为上下标设置特定的字体大小：

```
上标: 香風とうふ店 #super(size: 0.8em)[TM] \
```

上标: 香風とうふ店<sup>TM</sup>

你可以为上下标设置相对基线的合适高度：

```
上标: 香風とうふ店 #super(size: 1em,
baseline: -0.1em)[TM] \
```

上标: 香風とうふ店<sup>TM</sup>

## 文字属性

文本本身也可以设置一些「具名参数」。与 `strong` 和 `emph` 类似，文本也有一个对应的元素函数 `text`。`text` 接受任意内容，返回一个影响内部文本的结果。

当输入是单个文本时很好理解，返回的就是一个文本元素：

```
#text("一段内容")
```

一段内容

当输入是一段内容时，返回的是该内容本身，但是对于内容中的每一个文本元素，都作相应文本属性的修改。下例修改了「代码片段」元素中的文本元素为红色：

```
#text(fill: red)[``
影响块元素的内容
``]
```

影响块元素的内容

进一步，我们强调，其实际修改了缺省的文本属性。对比以下两个情形：



```
#text[``typ #strong[一段内容] #emph[一段内容]] \
#text(fill: red)[``typ #strong[一段内容] #emph[一段内容]] \
```

```
#strong[一段内容] #emph[一段内容]
#strong[一段内容] #emph[一段内容]
```

可以看见“红色”的设置仅对代码片段中的“默认颜色”的文本生效。对于那些已经被语法高亮的文本，“红色”的设置不再生效。

这说明了为什么下列情形输出了蓝色的文本：

```
#text(fill: red, text(fill: blue, "一段内容"))
```

一段内容

## 设置大小

通过 `size` 参数，可以设置文本大小。

```
#text(size: 12pt)[一斤鸭梨]
#text(size: 24pt)[四斤鸭梨]
```

一斤鸭梨 四斤鸭梨

其中 pt 是点单位。中文排版中常见的号单位与点单位有直接换算关系：

| 字号       | 初号 | 小初   | 一号   | 二号  | 小二  | 三号   | 小三 | 四号    |
|----------|----|------|------|-----|-----|------|----|-------|
| 中国（单位：点） | 42 | 36   | 26   | 22  | 18  | 16   | 15 | 14    |
| 日本（单位：点） | 42 | —    | 27.5 | 21  | —   | 16   | —  | 13.75 |
| 字号       | 小四 | 五号   | 小五   | 六号  | 小六  | 七号   | 八号 |       |
| 中国（单位：点） | 12 | 10.5 | 9    | 7.5 | 6.5 | 5.5  | 5  |       |
| 日本（单位：点） | —  | 10.5 | —    | 8   | —   | 5.25 | 4  |       |

另一个常见单位是 em：

```
#text(size: 1em)[一斤鸭梨]
#text(size: 2em)[四斤鸭梨]
```

一斤鸭梨 四斤鸭梨

1em 是当前设置的文字大小。

关于 Typst 中长度单位的详细介绍，可以挪步《参考：长度单位》。

## 设置颜色

你可以通过 `fill` 参数为文字配置各种颜色：

```
#text(fill: red)[红色鸭梨]
#text(fill: blue)[蓝色鸭梨]
```

红色鸭梨 蓝色鸭梨

你还可以通过颜色函数创建自定义颜色：

```
#text(fill: rgb("ef475d"))[茉莉红色鸭梨]
#text(fill: color.hsl(200deg, 100%, 70%))
[天依蓝色鸭梨]
```

茉莉红色鸭梨 天依蓝色鸭梨

关于 Typst 中色彩系统的详细介绍，详见《参考：颜色、渐变填充与模式填充》。

## 设置字体

你可以通过 `font` 参数为文字配置字体：

```
#text(font: "FangSong")[北京鸭梨]
#text(font: "Microsoft YaHei")[板正鸭梨]
```

北京鸭梨 板正鸭梨

你可以用逗号分隔的「列表」同时为文本设置多个字体。Typst 按顺序优先使用靠前字体。例如可以同时设置西文为 Times New Roman 字体，中文为仿宋字体：



```
#text(font: ("Times New Roman",
"FangSong"))[中西 Pear]
```

中西 Pear

关于如何在不同系统上配置中文、西文、数学等多种字体，详见《[字体设置](#)》。

## 「set」语法

Typst 允许你为元素的「具名参数」设置新的「默认值」，这个特性由「set」语法实现。

例如，你可以这样设置文本字体：

```
#set text(fill: red)
红色鸭梨
```

红色鸭梨

set 关键字后跟随一个与函数调用相同语法的表达式，表示此后所有的元素都具有新的默认值。这比 `#text(fill: red)[红色鸭梨]` 要更易读。

默认情况下文本元素的 fill 参数为黑色，即文本默认为黑色。经过 set 规则，其往后的文本都默认为红色。

```
黑色鸭梨
#set text(fill: red)
红色鸭梨
```

黑色鸭梨 红色鸭梨

之所以说它是默认值，是因为仍然可以在创建元素的时候指定参数值以覆盖默认值：

```
#set text(fill: red)
#text(fill: blue)[蓝色鸭梨]
```

蓝色鸭梨

本节前面讲述的所有「具名参数」都可以如是设置，例如文本大小、字体等。

关于对「set」语法更详细的介绍，详见《[内容、作用域与样式](#)》。

## 图像

图像对应元素函数 `image`。

你可以通过[绝对路径](#)或[相对路径](#)加载一个图片文件：

```
#image("/assets/files/香風とうふ店.jpg")
```



`image` 有一个很有用的 `width` 参数，用于限制图片的宽度：

```
#image("/assets/files/香風とうふ店.jpg",
width: 100pt)
```



你还可以相对于父元素设置宽度，例如设置为父元素宽度的 50%：

```
#image("/assets/files/香風とうふ店.jpg",
width: 50%)
```



同理，你也可以用 height 参数限制图片的高度。

```
#image("/assets/files/香風とうふ店.jpg",
height: 100pt)
```



当同时设置了图片的宽度和高度时，图片默认会被裁剪：

```
#image("/assets/files/香風とうふ店.jpg",
width: 100pt, height: 100pt)
```



如果想要拉伸图片而非裁剪图片，可以同时使用 fit 参数：

```
#image("/assets/files/香風とうふ店.jpg",
width: 100pt, height: 100pt, fit:
"stretch")
```



“stretch”在英文中是拉伸的意思。

## 图形

你可以通过 figure 函数为图像设置标题：

```
#figure(image("/assets/files/香風とうふ
店.jpg"), caption: [上世纪 90 年代，香風とうふ
店送外卖的宝贵影像])
```



图 3 上世纪 90 年代，香風とうふ店送外卖的宝贵影像

`figure` 不仅仅可以接受 `image` 作为内容，而是可以接受任意内容：

```
#figure(``typ
#image("/assets/files/香風とうふ店.jpg")
``, caption: [用于加载香風とうふ店送外卖的宝贵
影像的代码])
```

```
#image("/assets/files/香風とうふ店.jpg")
```

代码 1 用于加载香風とうふ店送外卖的宝贵影像的代码

## 行内盒子

todo: 本节添加 box 的基础使用。

```
在一段话中插入一个 #box(baseline: 0.15em,
image("/assets/files/info-icon.svg",
width: 1em)) 图片。
```

在一段话中插入一个  图片。

## 链接

链接可以分为外链与内链。最简单情况下，你只需要使用 `link` 函数即可创建一个链接：

```
#link("https://zh.wikipedia.org")
```

<https://zh.wikipedia.org>

特别地，Typst 会自动识别文中的 HTTPS 和 HTTP 链接文本并创建链接：

```
https://zh.wikipedia.org
```

<https://zh.wikipedia.org>

无论是内链还是外链，你都可以额外传入一段任意内容作为链接标题：

```
不基于比较方法，#link("https://zh.wikipedia.
org/zh-hans/%E5%9F%BA%E6%95%B0%E6%8E%92%E
5%BA%8F") [排序] 可以做到 $\mathcal{O}(n)$
时间复杂度。
```

不基于比较方法，[排序](#) 可以做到  $\mathcal{O}(n)$  时间复杂度。

请回忆，这其实等价于调用函数：

```
#link("...") [链接] 等价于 #link("...", [链
接])
```

[链接](#) 等价于 [链接](#)

## 内部链接

你可以通过创建标签，标记任意内容：

```
== 一个神秘标题 <myst>
```

一个神秘标题

上例中 `myst` 是该标签的名字。每个标签都会附加到恰在其之前的内容，这里内容即为该标题。

 在脚本模式中，标签无法附加到之前的内容。

```
#show <awa>: set text(fill: red)
#{[a]; [<awa>]}
#[b] <awa>
```

a b

对比上例，具体来说，标签附加到它的「语法前驱」(syntactic predecessor)。

这不是问题，但是易用性有可能在将来得到改善。

你可以通过 `link` 函数在文档中的任意位置链接到该内容：

```
== 一个神秘标题 <mystery>
```

讲述了 `#link(<mystery>)` [一个神秘标题]。

一个神秘标题

讲述了一个神秘标题。

## 表格基础

你可以通过 `table` 函数创建表格。`table` 接受一系列内容，并根据参数将内容组装成一个表格。如下，通过 `columns` 参数设置表格为 2 列，Typst 自动为你生成了一个 2 行 2 列的表格：

```
#table(columns: 2, [111], [2], [3])
```

|     |   |
|-----|---|
| 111 | 2 |
| 3   |   |

你可以为表格设定对齐：

```
#table(columns: 2, align: center, [111],
[2], [3])
```

|     |   |
|-----|---|
| 111 | 2 |
| 3   |   |

其他可选的对齐有 `left`、`right`、`bottom`、`top`、`horizon` 等，详见《参考：布局函数》。

## 使用其他人的模板

虽然这是一片教你写基础文档的教程，但是为什么不更进一步？有赖于 Typst 将样式与内容分离，如果你能找到一个朋友愿意为你分享两行神秘代码，当你粘贴到文档开头时，你的文档将会变得更为美观：

```
#import "latex-look.typ": latex-look
#show: latex-look

= 这是一篇与 LaTeX 样式更接近的文档

Hey there!

Here are two paragraphs. The
output is shown to the right.

Let's get started writing this
article by putting insightful
paragraphs right here!
+ following best practices
+ being aware of current results
 of other researchers
+ checking the data for biases

$
f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i \xi x} d\xi
$
```

## 这是一篇与 LaTeX 样式更接近的文档

Hey there!

Here are two paragraphs. The output is shown to the right.

Let's get started writing this article by putting insightful paragraphs right here!

1. following best practices
2. being aware of current results of other researchers
3. checking the data for biases

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i \xi x} d\xi$$

一般来说，使用他人的模板需要做两件事：

1. 将 `latex-look.typ` 放在你的文档文件夹中。
2. 使用以下两行代码应用模板样式：

```
#import "latex-look.typ": latex-look
#show: latex-look
```

## 总结

基于《编写一篇基本文档》掌握的知识你应该可以：

1. 像使用 Markdown 那样，编写一篇基本不设置样式的文档。
2. 查看《参考：数学模式》和《参考：常用数学符号》，以助你编写简单的数学公式。
3. 查看《参考：时间类型》，以在文档中使用时间。

## 习题

1. 用 `underline` 实现“删除线”效果，其中删除线距离 baseline 距离为 40%:

吾輩は猫である。

2. 攻击者有可能读取你文件系统中的内容，并将其隐藏存储在你的 PDF 中。请尝试将用户密码“I'm the flag”以文本形式存放在 PDF 中，但不可见:

3. 请仅用 `em` 实现以下效果，其中后一个字是前一个字大小的 1.5 倍:

走走走走

4. 请仅用 `em` 实现以下效果，其中后一个字是前一个字大小的 1.5 倍:

走走走 走走走

5. 请仅用 `em` 实现以下效果，其中前一个字是后一个字大小的 1.5 倍。要求代码中不允许出现中括号也不允许出现双引号:

走走走



## Part.2 基础教程 — 脚本 I



# 常量与变量

本章前两节适用于没有编程经验的同学。以前接触过编程的同学可以快速浏览并跳至第三节。

Typst 很快，并非因为它的「解析器」(parser) 和「解释器」(interpreter) 具有惊世的执行性能，而是语言特性本身适合缓存优化。

自本节起，本教程将进入第二阶段，希望不仅让你了解如何使用脚本，还一并讲解 Typst 的执行原理。当然，我们不会陷入 Typst 的细节。所有在教程中出现的原理都是为了更好地了解语言本身。

## 代码表示的自省函数

在开始学习之前，先学习几个与排版无关但非常实用的函数。

`repr` 是一个「自省函数」(introspection function)，可以帮你获得任意值的代码表示，很适合用来在调试代码的时候输出内容。

```
#[一段文本]
```

```
#repr([一段文本])
```

一段文本

sequence([ ], [一段文本], [ ])

❶ 「自省函数」是指那些为你获取解释器内部状态的函数。它们往往接受一些语言对象，而返回存储在解释器内部的相关信息。在这里，`repr` 接受任意值，而返回对应的代码表示。

## 类型的自省函数

与 `repr` 类似，一个特殊的函数 `type` 可以获得任意值的「类型」(type)。所谓「类型」，就是这个值归属的分类。例如：

- 1 是整数数字，类型就对应于整数类型 (integer)。

```
#type(1)
```

integer

- 一段内容是文本内容，类型就对应于内容类型 (content)。

```
#type([一段内容])
```

content

一个值只会属于一种类型，因此类型是可以比较的：

```
str == str \
type("X") == type("Y") \
type("X") == str \
[一段内容] == str
```

true  
true  
true  
false

类型的类型是类型（它自身）：

```
type(str) \
type(type(str)) \
type(type(str)) == type
```

type  
type  
true

## 求值函数

`eval` 函数接受一个字符串，把字符串当作代码执行并求出结果：

```
eval("1"), type(eval("1")) \
eval("[一段内容]"), type(eval("[一段内容]"))
```

1, integer  
一段内容, content

从 `eval("[一段内容]")` 的中括号被解释为「内容块」可以得知, `eval` 默认以「脚本模式」(code mode) 解释你的代码。

你可以使用 `mode` 参数修改 `eval` 的「解释模式」。code 对应为「脚本模式」, markup 对应为「标记模式」(markup mode):

```
代码模式 eval: #eval("[一段内容]", mode:
"code") \
标记模式 eval: #eval("[一段内容]", mode:
"markup")
```

代码模式 eval: 一段内容  
标记模式 eval: [一段内容]

## 基本字面量

本小节我们将具体介绍所有基本字面量, 这是脚本的“一加一”。其实在上一节, 我们已经见过了一部分字面量, 但皆凭直觉使用: `1` 不就是数字吗, 那么在 Typst 中, 它就是数字。(PS: 与之相对, TeX 根本没有数字和字符串的概念。)

如果你学过 Python 等语言, 那么这将对来说不是问题。在 Typst 中, 常用的字面量并不多, 它们是:

1. 「空字面量」(none literal)。
2. 「布尔字面量」(boolean literal)。
3. 「整数字面量」(integer literal)。
4. 「浮点数字面量」(floating-point literal)。
5. 「字符串字面量」(string literal)。

### 空字面量

空字面量是纯粹抽象的概念, 这意味着你在现实中很难找到对应的实体。就像是数学中的零与负数, 空字面量自然产生于运算过程中。

```
#repr((0, 1).find((_) => false)),
#repr(if false [啊?])
```

none, none

上例第一行, 当在「数组」中查找一个不存在的元素时, “没有” 就是 `none`。

上例第二行, 当条件不满足, 且没有 `false` 分支时, “没有内容” 就是 `none`。

`none` 值不会对输出文档有任何影响:

```
#none
```

`none` 的类型是 `none` 类型。 `none` 值不等于 `none` 类型, 因为一个是值而另一个是类型:

```
#type(none), #(type(none) == none),
#type(type(none))
```

type(none), false, type

### 布尔字面量

一个布尔字面量表示逻辑的确否。它要么为 `false` (真) 要么为 `true` (假)。

```
假设 #false 那么一切为 #true。
```

假设 false 那么一切为 true。

一般来说, 我们不直接使用布尔值。当代码做逻辑判断的时候, 会自然产生布尔值。

```
$1 < 2$ 的结果为: #(1 < 2)
```

1 < 2的结果为: true

### 整数字面量

一个整数字面量代表一个整数。相信你一定知道整数的含义。Typst 中的整数默认为十进制:

```
三个值 #(-1)、#0 和 #1 偷偷混入了我们内容之中。
```

三个值 -1、0 和 1 偷偷混入了我们内容之中。



ⓘ 有的时候 Typst 不支持在「井号」(#) 后直接跟一个值。这个时候无论值有多么复杂，都可以将值用一对圆括号包裹起来，从而允许 Typst 轻松解析该值。例如，Typst 无法处理「井号」后直接跟随一个「连字号」(-) 的情况：

```
#(-1), #(0), #(1)
```

-1, 0, 1

有些数字使用其他进制表示更为方便。你可以分别使用 0x、0o 和 0b 前缀加上进制内容表示十六进制数、八进制数和二进制数：

```
十六进制数: #(0xdeadbeef)、#(-0xdeadbeef) \
八进制数: #(0o755)、#(-0o644) \
二进制数: #(0b1001)、#(-0b1001)
```

十六进制数: 3735928559、-3735928559  
八进制数: 493、-420  
二进制数: 9、-9

上例中，当数字被输出到文档时，Typst 将数字都转换成了十进制表示。

整数的有效取值范围是 $[-2^{63}, 2^{63}]$ ，其中 $2^{63} = 9223372036854775808$ 。

## 浮点数字面量

浮点数与整数非常类似。最常见的浮点数由至少一个整数部分或小数部分组成：

三个值 `#(0.001)`、`#(.1)` 和 `#(2.)` 偷偷混入了我们内容之中。

三个值 0.001、0.1 和 2 偷偷混入了我们内容之中。

有些数字使用「指数表示法」(exponential notation) 更为方便。你可以使用标准的「指数表示法」创建浮点数：

```
#(1e2)、#(1.926e3)、#(-1e-3)
```

100、1926、-0.001

Typst 还为你内置了一些特殊的数值，它们都是浮点数：

```
pi=#calc.pi \
tau=#calc.tau \
inf=#calc.inf \
// NaN=#calc.nan \
```

$\pi=3.141592653589793$   
 $\tau=6.283185307179586$   
 $\text{inf}=\infty$

## 字符串字面量

Typst 中所有字符串都是 utf-8 编码的，因此使用时不存在编码转换问题。字符串由一对「英文双引号」界定：

```
#"Hello world!!"
```

Hello world!!

有些字符无法置于双引号之内，例如双引号本身。与「标记模式」中的转义序列语法类似，这时候你需要嵌入字符的转义序列：

```
#"Hello \"world\"!!"
```

Hello "world"!!

字符串中的转义序列与「标记模式」中的转义序列语法相同，但有效转义的字符集不同。字符串中如下转义序列是有效的：

| 代码 | <code>\\</code> | <code>\"</code> | <code>\n</code> | <code>\r</code> | <code>\t</code> | <code>\u{2665}</code> |
|----|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------------|
| 效果 | <code>\</code>  | <code>"</code>  | (换行)            | (回车)            | (制表)            | ♥                     |

你同样可以使用 `\u{unicode}` 格式直接嵌入 Unicode 字符。

#"香辣牛肉粉好吃\u{2665}"

香辣牛肉粉好吃♥

除了使用简单字面量构造，可以使用以下方法从代码块获得字符串：

```
#repr(`包含换行符和双引号的`
"内容"`.text)
```

"包含换行符和双引号的\n\n"内容"

## 类型转换

类型本身也是函数，例如 `int` 类型可以接受字符串，转换成整数。

```
#int("11"), #int("-23")
```

11, -23

从转换的角度，eval 可以将代码字符串转换成值。例如，你可以转换 16 进制数字：

```
#eval("0x3f")
```

63

## 计算标准库

由于该库即将废弃（本文将介绍新的计算 API），如有希望使用的朋友，请参见 [Typst Reference - Calculation](#)。

## 浮点数陷阱

浮点数陷阱是指由于浮点转换或浮点精度问题导致的一系列逻辑问题。

1. 类型比较陷阱。在运行期切记同时考虑到 float 和 int 两种类型:

```
#type(2), #type(2.), #(type(2.) == int)
```

integer, float, false

可见 2. 与 2 类型并不相同。2. 在数学上是整数，但以浮点数存储。

2. 大数类型陷阱。当数字过大时，其会被隐式转换为浮点数存储：

```
#type(9000000000000000000000000000000000000000)
```

float

- ### 3. 整数运算陷阱。整数相除时会被转换为浮点数:

```
#(10 / 4), #type(10 / 4) \n
#(12 / 4), #type(12 / 4) \n
```

```
2.5, float
3, float
```

4. 浮点误差陷阱。哪怕你的运算在数学上理论是可逆的，由于浮点数精度有限，也会误判结果：

```
#(1000000 / 9e21 * 9e21), #((1000000 /
9e21 * 9e21) == 1000000)
```

```
1000000.0000000001, false
```

这提示我们，正确的浮点比较需要考虑误差。例如以上两数在允许  $1e-6$  误差前提下是相等的：

```
#(calc.abs((1000000 / 9e21 * 9e21) -
1000000) < 1e-6)
```

true

5. 整数转换陷阱。为了转换类型，可以使用 `int`，但有可能产生精度损失（就近取整）：

```
#int(10 / 4),
#int(12 / 4)
```

2, 3

或有可能产生「截断」。(todo: typst v0.12.0 已经不适用)

这些都是编程语言中的共通问题。凡是令数字保持有效精度，都会产生如上问题。

## 变量声明

变量是存储“字面量”的一个个容器。它相当于为一个个字面量取名，以方便在脚本中使用。

如下语法，「变量声明」表示使得 `x` 的内容与 `"Hello world!!"` 相等。我们对语法一一翻译：

```
#let x = "Hello world!!"
// ^^^^ ^ ^ ^^^^^^^^^^^^^^^^^^^
// 令 变量名 为 初始值表达式
```

同时我们看见输出的文档为空，这是因为「变量声明」本身的值是 `none`。

「变量声明」一共分为 3 个有效部分。

1. `let` 关键字：告诉 Typst 接下来即将开启一段「声明」。在英语中 `let` 是“令”的意思。
2. 「变量名标识符」(variable identifier)、「等于号」(=)：标识符是变量的“名称”。「变量声明」后续的位置都可以通过标识符引用该变量。
3. 「初始值表达式」(initialization expression)：「等于号」告诉 Typst 变量初始等于一个表达式的值。该表达式在编译领域有一个专业术语，称为「初始值表达式」。这里，「初始值表达式」可以为任意表达式，请参阅

建议标识符简短且具有描述性。尽管标识符中可以包含中文等 unicode 字符，但仍建议标识符中仅含英文与「连字号」。

「变量声明」后续的位置都可以继续使用该变量，取决于「作用域」。

关于作用域，你可以参考《内容、作用域与样式》。

变量可以重复输出到文档中：

```
#let x = "阿吧"
#x#x, #x#x
```

阿吧阿吧，阿吧阿吧

任意时刻都可以将任意类型的值赋给一个变量。上一节所提到的「内容块」也可以赋值给一个变量。

```
#let y = [一段文本]; #y \
#(y = 1) /* 重新赋值为一个整数 */ #y \
```

一段文本  
1

任意时刻都可以重复定义相同变量名的变量，但是之前被定义的变量将无法再被使用：

```
#let y = [一段文本]; #y \
#let y = 1; /* 重新声明为一个整数 */ #y \
```

一段文本  
1

## 函数声明

「函数声明」也由 `let` 关键字开始。如果你仔细对比，可以发现它们在语法上是一致的。

如下语法，「函数声明」表示使得 `f(x, y)` 的内容与右侧表达式的值相等。我们对语法一一翻译：

```
#let f(x, y) = [两个值 #x 和 #y 偷偷混入了我们内容之中。]
// ^^^^ ^^^^^^ ^ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// 令 函数名(参数列表) 为 一段内容
```

「函数声明」一共分为 4 个部分，相较于「变量声明」，多了一个参数列表。

参数列表中参数的个数可以有零个，一个，至任意多个，由逗号分隔。每个参数都是一个单独的「参数标识符」(parameter identifier)。

```
// 零个参数，一个参数，两个参数，..
#f(), #f(x), #f(x, y)
```

「参数标识符」的规则与功能与「变量名标识符」相似。为参数取名是为其能在函数体中使用。

视角挪到等号右侧。「函数体表达式」(function body expression)的规则与功能与「初始值表达式」相似。「函数体表达式」可以任意使用参数列表中的“变量”，并组合出一个表达式。组合出的表达式的值就是函数调用的结果。

结合例子理解函数的作用。将对应参数应用于函数可以取得对应的结果：

```
#let f(x, y) = [两个值 #(x)和 #(y)偷偷混入了我们内容之中。]

#let x = "Hello world!!"
#let y = [一段文本]
#f(repr(x), y)
```

两个值"Hello world!!"和一段文本偷偷混入了我们内容之中。

其中 `#f(repr(x), y)` 的执行过程是这样的：

```
#f(repr(x), y) // 转变为
#f(["Hello world!!"], [一段文本])
```

注意，此时我们进入右侧表达式。

```
#[两个值 #(x)和 #(y)偷偷混入了我们内容之中。] // 转变为
#[两个值 #(["Hello world!!"]) 和 #([一段文本])偷偷混入了我们内容之中。]
```

最后整理式子得到，也即是我们看到的输出：

```
两个值 ["Hello world!!"] 和一段文本偷偷混入了我们内容之中。
```

## 函数闭包

「闭包」是一类特殊的函数，又称为匿名函数。一个简单的闭包如下：

```
#let f = (x, y) => [#(x)和 #(y)。]
#f("狗", "你")
```

狗和你。

我们可以看到其以箭头为体，两侧为参数列表和函数体。不像「函数声明」，它不需要取名。

闭包以其匿名特征，很适合作为参数传入其他函数，即作为「回调」函数。我们将会在下一章经常使用「回调」函数。例如，Typst 提供一个 `locate` 函数，其接受一个函数类型的参数：

```
#let f(loc) = [当前页码为 #loc.page()]
#locate(f)
```

当前页码为 36。

但是，等价地，此时「函数声明」不如「闭包声明」优美：

```
#locate(loc => [当前页码为 #loc.page()])
```

当前页码为 36。

## 成员

Typst 提供了一系列「成员」和「方法」访问字面量、变量与函数中存储的“信息”。

其实在上一节（甚至是第二节），你就已经见过了「成员」语法。你可以通过「点号」，即 ``0v0`.text`，获得代码块的“text”（文本内容）：

```
这是一个代码块: #repr(`0v0`)

这是一段文本: #repr(`0v0`.text)
```

这是一个代码块: `raw(text: "OvO", block: false)`  
这是一段文本: "OvO"

每个类型有哪些「成员」是由 Typst 决定的。你需要逐渐积累经验以知晓这些「成员」的分布，才能更快地通过访问成员快速编写出收集和处理信息的脚本。（todo: 建议阅读）

当然，为防你不知道，大家不都是死记硬背的：有软件手段帮助你使用这些「成员」。许多编辑器都支持 LSP（Language Server Protocol，语言服务），例如 VSCode 安装 `Tinymist LSP`。当你对某个对象后接一个点号时，编辑器会自动为你做代码补全。



图 4 作者使用编辑器作代码补全的精彩瞬间。

从图中可以看出来，该代码片段对象上有七个「成员」。特别是“text”成员赫然立于其中，就是它了。除了「成员」列表，编辑器还会告诉你每个「成员」的作用，以及如何使用。这时候只需要选择一个「成员」作为补全结果即可。

## 方法

「方法」是一种特殊的「成员」。准确来说，如果一个「成员」是一个对象的函数，那么它就被称为该对象的「方法」。

来看以下代码，它们输出了相同的内容，事实上，它们是同一「函数调用」的不同写法：

```
#let x = "Hello World"
#let str-split = str.split
#str-split(x, " ") \
#str.split(x, " ") \
#x.split(" ")
```

```
("Hello", "World")
("Hello", "World")
("Hello", "World")
```

第三行脚本含义对照如下。之前已经学过，这正是「函数调用」的语法：

```
 #(str-split(x, " "))
// 调用 字符串拆分函数，参数为 变量 x 和空格
```

与第三行脚本相比，第四行脚本仍然是在做「函数调用」，只不过在语法上更为紧凑。

第五行脚本则更加简明，此即「方法调用」。约定 `str.split(x, y)` 可以简写为 `x.split(y)`，如果：

1. 对象 `x` 是 `str` 类型，且方法 `split` 是 `str` 类型的「成员」。
2. 对象 `x` 用作 `str.split` 调用的第一个参数。

「方法调用」即一种特殊的「函数调用」规则（语法糖），在各编程语言中广泛存在。其大大简化了脚本。但你也可以选择不用，毕竟「函数调用」一样可以完成所有任务。

❗ 这里有一个问题：为什么 Typst 要引入「方法」的概念呢？主要有以下几点考量。

其一，为了引入「方法调用」的语法，这种语法相对要更为方便和易读。对比以下两行，它们都完成了获取“Hello World”字符串的第二个单词的第一个字母的功能：

```
#"Hello World".split(" ").at(1).split("").at(1)
#array.at(str.split(array.at(str.split("Hello World", " "), 1), ""), 1)
```

```
W W
```

可以明显看见，第二行语句的参数已经散落在括号的里里外外，很难理解到底做了什么事情。

其二，相比「函数调用」，「方法调用」更有利于现代 IDE 补全脚本。你可以通过 `.split` 很快定位到“字符串拆分”这个函数。

其三，方便用户管理相似功能的函数。不仅仅是字符串可以拆分，似乎内容及其他许多类型也可以拆分。如果一一为它们取不同的名字，那可就太头疼了。相比，`str.split` 就简单多了。要知道，很多程序员都非常头痛为不同的变量和函数取名。

## 数组字面量

脚本模式中有两类核心复合字面量。

「数组」是按照顺序存储的一些「值」，你可以在「数组」中存放任意内容而不拘泥于类型。你可以使用圆括号与一个逗号分隔的列表创建一个数组字面量：

```
#(1, "0v0", [一段内容])
```

```
(1, "0v0", [一段内容])
```

## 字典字面量

所谓「字典」即是「键值对」的集合，其每一项是由冒号分隔的「键值对」。如下例所示，冒号左侧，“neko-mimi”等「标识符」或「字符串」是字典的「键」；而冒号右侧分别是对应的「值」。

```
#{neko-mimi: 2, "utterance": "喵喵喵"}
```

```
(neko-mimi: 2, utterance: "喵喵喵")
```

## 数组和字典的成员访问

为了访问数组，你可以使用 `at` 方法。“at”在中文里是“在”的意思，它表示对「数组」使用「索引」操作。在数组中，第 0 个值就是其第一个值，第  $N$  个值就是其第  $N + 1$  个值，以此类推。如下所示：

```
#let x = (1, "0v0", [一段内容])
#x.at(0), #x.at(1), #x.at(2)
```

```
1, 0v0, 一段内容
```

至于「索引」从零开始的原因，这只是约定俗成。等你习惯了，你也会变成计数从零开始的好程序员。

为了访问字典，你可以使用 `at` 方法。但由于「键」都是字符串，你需要使用字符串作为字典的「索引」。

```
#let cat = {attribute: [kawaii~]}
#cat.at("attribute")
```

```
kawaii~
```

为了方便，Typst 允许你直接通过成员方法访问字典对应「键」的值：

```
#let cat = {"attribute": [kawaii~]}
#cat.attribute
```

```
kawaii~
```

## 数组和字典的「存在谓词」

与数组相关的另一个重要语法是 `in`，`x in (...)`，表示判断 `x` 是否存在于某个数组中：

```
#[一段内容 in (1, "0v0", [一段内容])] \
#[另一段内容 in (1, "0v0", [一段内容])]
```

```
true
false
```

字典也可以使用此语法，表示判断 `x` 是否是字典的一个「键」。特别地，你还可以前置一个 `not` 判断 `x` 是否不在某个数组或字典中：

```
#let cat = {neko-mimi: 2}
#("neko-kiki" not in cat)
```

```
true
```

注意：`x in (...)` 与 `"x" in (...)` 是不同的。例如 `neko-mimi in cat` 将检查 `neko-mimi` 变量的内容是否是字典变量 `cat` 的一个「键」，而 `"neko-mimi" in cat` 检查对应字符串是否在其中。

## 数组和字典的「解构赋值」

除了使用字面量「构造」元素，Typst 还支持「构造」的反向操作：「解构赋值」。顾名思义，你可以在左侧用相似的语法从数组或字典中获取值并赋值到对应的变量上。

```
#let (attr: a) = (attr: [kawaii~])
#a
```

```
kawaii~
```

「解构赋值」必须一一对应，但你也可以使用「占位符」(`_`)或「延展符」(`..`)以作部分解构：



```
#let (first, ..) = (1, 2, 3)
#let (.., second-last, _) = (7, 8, 9, 10)
#first, #second-last
```

1, 9

数组的「解构赋值」有一个妙用，那就是重映射内容。

```
#let (a, b, c) = (1, 2, 3)
#let (b, c, a) = (a, b, c); #a, #b, #c
```

3, 1, 2

特别地，如果两个变量相互重映射，这种操作被称为「交换」：

```
#let (a, b) = (1, 2)
#((a, b) = (b, a)); #a, #b
```

2, 1

## 数组和字典的典型构造

特别讲解一些关于数组与字典相关的典型构造：

```
#() \ // 是空的数组
#(:) \ // 是空的字典
#(1) \ // 被括号包裹的整数 1
#(()) \ // 被括号包裹的空数组
#((())) \ // 被括号包裹的被括号包裹的空数组
#(1,) \ // 是含有一个元素的数组
```

()  
(:)  
1  
()  
()  
(1,)

()是空的数组，不含任何值。(:)是空的字典，不含任何键值对。

如果括号内含了一个值，例如(1)，那么它仅仅是被括号包裹的整数 1，仍然是整数 1 本身。

类似的，(())是被括号包裹的空数组，((( )))是被括号包裹的被括号包裹的空数组。

为了构建含有一个元素的数组，需要在列表末尾额外放置一个逗号以示区分，例如(1,)。

这种逗号被称为尾后逗号。

构造数组字面量时，允许尾随一个多余的逗号而不造成影响。

```
#let x = (1, "OvO", [一段内容],); #x
// 这里有一个多余的逗号^^^
```

(1, "OvO", [一段内容])

构造字典字面量时，允许尾随一个多余的逗号。

```
#let cat = (attribute: [kawaii~],); #cat
// 这里有一个尾随的小逗号^^
```

(attribute: sequence([kawaii], [~]))

## 高级参数语法

学会了「数组」和「字典」，我们可以学习更加高级的参数语法。这些高级语法让参数声明更灵活。

### 具名参数声明

以上两种函数都允许包含「具名参数」，其看起来就像字典的一项，冒号右侧则是参数的默认值：

```
#let g(name: "? ") = [你是 #name]
#g(/* 不传就是问号 */); #g(name: "OwO。")
```

你是？ 你是 OwO。

### 变长参数

「函数声明」和「闭包声明」都允许包含「变长参数」。

```
#let g(..args) = [很多个值,
#args.pos().join([、]), 偷偷混入了我们内容之
中。]
#g([一个俩个], [仨个四个], [五六七八个])
```

很多个值, 一个俩个、仨个四个、五六七八个,  
偷偷混入了我们内容之中。

args.pos()的类型是 Argument。

1. 使用 `args.pos()` 得到按顺序传入的参数
2. 使用 `args.at(name)` 访问名称为 `name` 的具名参数。

## 参数解构

todo 参数解构。

## 总结

基于《字面量、变量和函数》掌握的知识你应该可以：

1. 查看《参考：内置类型》，以掌握内置类型的使用方法。
2. 查看《参考：图形与几何元素》，以掌握图形和几何元素的使用方法。
3. 查看《参考：WASM 插件》，以掌握在 Typst 中使用 Rust、JavaScript、Python 等语言编写插件库。
4. 阅读《参考：语法示例检索表》，以检查自己的语法掌握程度。
5. 查看《参考：基本类型》，以掌握基本类型的使用方法。
6. 查看《参考：颜色、色彩渐变与模式》，以掌握色彩的高级管理方法。
7. 查看《参考：数据读写与数据处理》，以助你从外部读取数据或将文档数据输出到文件。
8. 查看《参考：导入和使用参考文献》，以助你导入和使用参考文献。
9. 阅读基本参考部分中的所有内容。

## 习题

1. 使用本节所讲述的语法，计算  $2^{32}$  的值：

4294967296

2. 输出下面的诗句，但你的代码中至多只能出现 17 个汉字：

一帆一桨一渔舟，一个渔翁一钓钩。  
一俯一仰一场笑，一江明月一江秋。

3. 已知斐波那契数列的递推式为  $F_n = F_{n-1} + F_{n-2}$ ，且  $F_0 = 0, F_1 = 1$ 。使用本节所讲述的语法，计算  $F_{75}$  的值：

2111485077978050

4. 编写函数，使用 `table`（表格）元素打印任意  $N \times M$  矩阵，例如：

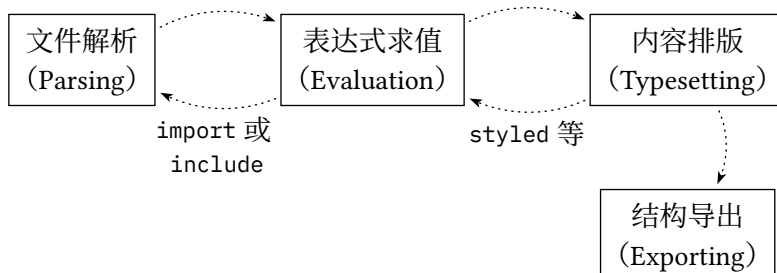
```
#matrix-fmt(
 (1, 2, 3),
 (4, 5, 6),
 (7, 8, 9),
)
```

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |



# 块与表达式

纵览 Typst 的编译流程，其大致分为 4 个阶段，解析、求值、排版和导出。



为了方便排版，Typst 首先使用了一个函数“解析和评估”你的代码。有趣地是，我们之前已经学过了这个函数。事实上，它就是 `eval`。

```
#repr(eval("#[一段内容]", mode: "markup"))
```

[一段内容]

流程图展现了编译阶段间的关系，也包含了本节「块」与「表达式」两个概念之间的关系。

- `eval` 输入：在文件解析阶段，**代码字符串**被解析成一个语法结构，即「表达式」。古人云，世界是一个巨大的表达式。作为世界的一部分，Typst 文档本身也是一个巨大的表达式。事实上，它就是我们在上一章提及的「内容块」。文档的本身是一个内容块，其由一个个标记串联形成。

- `eval` 输出：在内容排版阶段，排版引擎事实不作任何计算。用 TeX 黑话来说，文档被“解析和评估”完了之后，就成为了一个个「材料」(material)。排版引擎将材料。

在求值阶段。「表达式」被计算成一个方便排版引擎操作的值，即「材料」。一般来说，我们所谓的表达式是诸如  $1+1$  的算式，而对其求值则是做算数。

```
#eval("1+1")
```

2

显然，如果意图是让排版引擎输出计算结果，让排版引擎直接排版 2 要比排版 “1+1” 更简单。

但是对于整个文档，要如何理解对内容块的求值？这就引入了「可折叠」的值 (Foldable) 的概念。「可折叠」成为块作为表达式的基础。

❗ Typst 借鉴了 Rust，遵从「面向表达式编程」(expression-oriented programming) 的哲学。它将所有的语句都根据可折叠规则（见后文）设计为表达式。

1. 如果一个语句能产生值，那么该语句的结果是按**控制流**顺序所产生所有值的折叠。
2. 否则，如果一个语句不能产生值，那么该语句的结果是 `none`。
3. 特别地，任意类型  $T$  的值  $v$  与 `none` 折叠仍然是值本身。

$$\forall v \in T \cup \{\text{none}\}, \text{fold}_T(v, \text{none}) = v$$

## 表达式

Typst 中绝大部分语法结构都可作表达式，可以说学习完了所有表达式，则学会了 Typst 所有语法。当然，其吸取了一定历史经验，仅有少量的语句不是表达式。

❗ `show` 语句和 `set` 语句不是表达式。

根据大的分类，可用的表达式可以分为 5 类（它们并非严格术语，而是为了方便教学而分类），他们是：

1. 代数运算表达式。
2. 逻辑比较表达式。
3. 逻辑运算表达式。

4. 赋值表达式。

5. 项。

其中第五项又是一个大类。我们上一章所学的所有常量、变量以及变量的应用都是项。块表达式也是项，将在本节后半部分详细介绍。

## 代数运算表达式

Typst 支持对数字的算数运算，其中浮点运算遵守 IEEE-754 标准。整数和浮点数之间可以混合运算：

```
//加 减 乘 除
#(1 + 2, 1.0 - 2, 1 * 2, 1 / 2 + 1)
```

```
(3, -1.0, 2, 1.5)
```

从上可以看出，整数和整数运算尽可能**保持整数**，但是整数和浮点数运算则结果变为浮点数。除法运算的结果是浮点数。算数之间遵守四则运算规则。

❶ 更高级的运算，例如数字的位运算和数值计算隐藏在类型的方法和 `calc` 标准库中：

```
#(0o755.bit-and(0o644)), // 8 进制位运算
#(calc.pow(9, 4)) // 9 的 4 次方
```

```
420,6561
```

请参考《》。

除了数字运算，字符串、数组等还支持加法和乘法运算。它们的加法实际上是连接操作，它们的乘法则是重复元素的累加。

```
//乘 加
#("1" * 2, "4" + "5",) \
#((1,) * 2, (4,) + (5,))
```

```
("11", "45")
((1, 1), (4, 5))
```

字典只支持加法操作。若有重复键值对，则右侧列表的键值对会覆盖左侧列表的键值对。

```
#((a: 1) + (b: 2),
 (a: 1) + (b: 3, a: 2) + (a: 4, c: 5))
```

```
((a: 1, b: 2), (a: 4, b: 3, c: 5))
```

## 逻辑比较表达式

有三大简单比较关系：

```
//大于 小于 等于
#(1 > 2, 1 < 2, 1 == 2)
```

```
(false, true, false)
```

基于此，可以延伸出三个方便脚本编辑的比较关系。

```
//大于或等于 小于或等于 不等于
#(1 >= 2.0, 1 <= 1, 1 != 2)
```

```
(false, true, true)
```

从数学角度，真正基本的关系另有其系，其中“小于或等于”是数学中的偏序关系，“等于”是数学中的等价关系。这两个关系可以衍生出其他四种关系，但是不太好理解。举例来讲，大于其实就是“小于等于”的否定；而小于则是“小于等于”但是不能“等于”。

注意：不推荐将整数与浮点数相互比较。具体请参考上一节所提及的浮点数陷阱。

字符串、数组和字典之间也是可以比较的，理解他们则必须从偏序关系和等价关系入手。

三种项的等价关系比较容易理解，说两字符串/数组/字典相等，则是在说二者有一样多的子项，且每个子项都一一相等。

```
#((1, 1) == (1, 1)),
#((a: 1, c: (1,)) == (a: 1, c: (1,)))
```

```
true, true
```

字典之间没有偏序关系，便只剩字符串和数组。偏序关系则需要指定一种排序规则。如果两字符串/数组不相等，则首先考虑它们的长度关系，长度小的一方是更小的：

```
#("1" <= "23"),
#((1,) <= (2, 3,))
```

```
true, true
```

否则从前往后依次比较每一个子项，直到找到第一个不相等的子项，进而确定顺序。见下三例：

```
#("1" <= "2", "113" <= "121", "2" <= "12")
```

```
(true, true, false)
```

对于单个字符，我们依照字典序比较，其中容易理解地是数字字符顺序按字面递增。对于前两者，我们可以看到“1”比“2”小；进由此，故“113”比“121”小；尽管如此，优先判断长度关系，故“2”比“12”小。

从两种基础比较关系，其他四种比较关系也能被良好地定义，并在脚本中使用：

```
#("1" > "2", "1" != "2",
"1" < "2", "1" >= "2")
```

```
(false, true, true, false)
```

## 逻辑运算表达式

布尔值之间可以做“且”、“或”和“非”三种逻辑运算，并产生布尔类型的表达式：

```
#(not false), #(false or false), #(true
and false)
```

```
true, false, false
```

真值表如下：

| $p$ | $q$ | $\neg p$ | $p \vee q$ | $p \wedge q$ |
|-----|-----|----------|------------|--------------|
| 0   | 0   | 1        | 0          | 0            |
| 0   | 1   | 1        | 1          | 0            |
| 1   | 0   | 0        | 1          | 0            |
| 1   | 1   | 0        | 1          | 1            |

逻辑运算使用起来很简单，建议入门的同学找一些专题阅读，例如[数理逻辑 \(1\) ——命题逻辑的基本概念](#)。但一旦涉及到对复杂事物的逻辑讨论，你就可能陷入了知识的海洋。关于逻辑运算已经形成一门学科，如有兴趣建议后续找一些书籍阅读，例如[逻辑学概论](#)。

本书自然不负责教你逻辑学。

## 赋值表达式

变量可以被赋予一个表达式的值。事实上，let 表达式后的语法结构就是赋值表达式。

```
#let a = 1; #a,
#(a = 10); #a
```

```
1, 10
```

除此之外，还有先加（减、乘或除）后赋值的变形。所有这些赋值语句都产生 none 值而非返回变量的值。

```
#let a = 1; #a,
#repr(a += 2), #a, #repr(a -= 2), #a,
#repr(a *= 2), #a, #repr(a /= 2), #a
```

```
1, none, 3, none, 1, none, 2, none, 1
```

## 代码块

在 Typst 中，代码块和内容块是等同的。与「代码块」

- 代码块：按顺序包含一系列语句，内部为「脚本模式」（code mode）。
- 内容块：按顺序包含一系列内容，内部为「标记模式」（markup mode）。

内容块（标记模式）内部没有语句的概念，一个个内容或元素按顺序排列。但你可以通过「井号」（#）将解释器的「解释模式」从「标记模式」临时改为「脚本模式」。当执行完脚本后，将脚本结果转换成内容，并放置在「井号」处。

相比，代码块内部则有语句概念。每个语句可以是换行分隔，也可以是「分号」（;）分隔。

```
#{
 "a"
 "b"
}
 \ // 与下表达式等同:
#{ "a"; "b" }
```

```
ab
ab
```

## 「可折叠」的值 (Foldable)

先来看代码块。代码块其实就是一个脚本。既然是脚本，Typst 就可以按照语句顺序依次执行「语句」。

📌 准确地来说，按照控制流顺序。

Typst 按控制流顺序执行代码，将所有结果**折叠**成一个值。所谓折叠，就是将所有数值“连接”在一起。这样讲还是太抽象了，来看一些具体的例子。

### 字符串折叠

Typst 实际上不限制代码块的每个语句将会产生什么结果，只要是结果之间可以**折叠**即可。

我们说字符串是可以折叠的：

```
#{ "Hello"; " "; "World" }
```

```
Hello World
```

实际上折叠操作基本就是「加号」(+) 操作。那么字符串的折叠就是在做字符串连接操作：

```
#{ "Hello" + " " + "World" }
```

```
Hello World
```

再看一个例子：

```
#{
 let hello = "Hello";
 let space = " ";
 let world = "World";
 hello; space; world;
 let destroy = ", Destroy"
 destroy; space; world; "."
}
```

```
Hello World, Destroy World.
```

如何理解将「变量声明」与表达式混写？

回忆前文。对了，「变量声明」表达式的结果为 `none`。

```
#type(let hello = "Hello")
```

```
type(none)
```

并且还有一个重点是，字符串与 `none` 相加是字符串本身，`none` 加 `none` 还是 `none`：

```
#{ "Hello" + none, #(none + "Hello"),
 #repr(none + none) }
```

```
Hello, Hello, none
```

现在可以重新体会这句话了：Typst 按控制流顺序执行代码，将所有结果**折叠**成一个值。对于上例，每句话的执行结果分别是：

```
#{
 none; // let hello = "Hello";
 none; // let space = " ";
 none; // let world = "World";
 "Hello"; " "; "World"; // hello; space; world;
 none; // let destroy = ", Destroy"
 ", Destroy"; " "; "World"; "." // destroy; space; world; "."
}
```

将结果收集并“折叠”，得到结果：

```
#(none + none + none + "Hello" + " " +
"World" + none + ", Destroy" + " " +
"Wrold" + ".")
```

Hello World, Destroy Wrold.

❶ 还有其他可以折叠的值，例如，数组与字典也是可以折叠的：

```
#for i in range(1, 5) { (i, i * 10) }
```

(1, 10, 2, 20, 3, 30, 4, 40)

```
#for i in range(1, 5) { let d = (:);
d.insert(str(i), i * 10); d }
```

("1": 10, "2": 20, "3": 30, "4": 40)

## 其他基本类型的情况

那么为什么说折叠操作基本就是「加号」操作。那么就是说有的“「加号」操作”并非是折叠操作。

布尔值、整数和浮点数都不能相互折叠：

```
// 不能编译
#{ false; true }; #{ 1; 2 }; #{ 1.; 2. }
```

那么是否说布尔值、整数和浮点数都不能折叠呢。答案又是否认的，它们都可以与 `none` 折叠（把下面的加号看成折叠操作）：

```
#{(1 + none)
```

1

所以你可以保证一个代码块中只有一个「语句」产生布尔值、整数或浮点数结果，这样的代码块就又是能编译的了。让我们利用 `let _ =` 来实现这一点：

```
#{ let _ = 1; true },
#{ let _ = false; 2. }
```

true, 2

回忆之前所讲的特殊规则：「占位符」（placeholder）用作标识符的作用是“忽略不必要的语句结果”。

## 内容折叠

Typst 脚本的核心重点就在本段。

内容也可以作为代码块的语句结果，这时候内容块的结果是每个语句内容的“折叠”。

```
#{
 [= 生活在 Content 树上]
 [现代社会以海德格尔的一句“一切实践传统都已经瓦解完了”为噱头。]
 [滥觞于家庭与社会传统的期望正失去它们的借鉴意义。]
 [但面对看似无垠的未来天空，我想循卡尔维诺“树上的男爵”的生活好过过早地振翅。]
}
```

## 生活在 Content 树上

现代社会以海德格尔的一句“一切实践传统都已经瓦解完了”为噱头。滥觞于家庭与社会传统的期望正失去它们的借鉴意义。但面对看似无垠的未来天空，我想循卡尔维诺“树上的男爵”的生活好过过早地振翅。

是不是感觉很熟悉？实际上内容块就是上述代码块的“糖”。所谓糖就是同一事物更方便书写的语法。上述代码块与下述内容块等价：

```
#[
 = 生活在 Content 树上
 现代社会以海德格尔的一句“一切实践传统都已经瓦解完了”为噱头。滥觞于家庭与社会传统的期望正失去它们的借鉴意义。但面对看似无垠的未来天空，我想循卡尔维诺“树上的男爵”的生活好过过早地振翅。
]
```

由于 Typst 默认以「标记模式」开始解释你的文档，这又与省略 `#[]` 的写法等价：

```
= 生活在 Content 树上
现代社会以海德格尔的一句“一切实践传统都已经瓦解完了”为噱头。滥觞于家庭与社会传统的期望正失去它们的借鉴意义。但面对看似无垠的未来天空，我想循卡尔维诺“树上的男爵”的生活好过过早地振翅。
```

❗ 实际上有区别，由于多两个换行，前后各多一个 Space Element。

## none 类型和 if 语句

默认情况下，在逻辑上，Typst 按照顺序执行你的代码，即先执行前面的语句，再执行后面的语句。开发者如果想要控制程序执行的流程，就必须使用流程控制的语法结构，主要是条件执行和循环执行。

if 语句用于条件判断，满足条件时，就执行指定的语句。

```
#if expression { then-block } else { else-block }
#if expression { then-block }
```

上面式子中，表达式 expression 为真（值为布尔值 true）时，就执行 then-block 代码块，否则执行 else-block 代码块。特别地，else 可以省略。

如下所示：

```
#if (1 < 2) { "确实" } else { "啊?" }
```

确实

因为  $1 < 2$  表达式为真，所以脚本仅执行了 then-block 代码块，于是最后文档的内容为“确实”。

if 语句还可以无限串联下去，你可以自行类比推理更长的 if 语句的语义：

```
#if expression { .. } else if expression { .. } else { .. }
#if expression { .. } else if expression { .. }
#if expression { .. } else if expression { .. } else if ..
```

如果只写了 then 代码块，而没写 else 代码块，但偏偏表达式不为真，最终脚本会报错吗？请看：

```
#repr(if (1 > 2) { "啊?" })
```

none

当 if 表达式没写 else 代码块而条件为假时，结果为 none。“none”在中文里意思是“无”，表示“什么都没有”。同时再次强调 none 在「可折叠」的值中很重要的一个性质：none 在折叠过程中被忽略。

见下程序，其根据数组所包含的值输出特定字符串：

```
#let 查成分(成分数组) = {
 "是个"
 if "A" in 成分数组 or "C" in 成分数组 or
 "G" in 成分数组 { "萌萌" }
 if "T" in 成分数组 { "工具" }
 "人"
}

#查成分() \
#查成分(("A", "T"),) \

```

是个人  
是个萌萌工具人

由于 if 也是表达式，你可以直接将 if 作为函数体，例如 fibonacci 函数的递归可以非常简单：

```
#let fib(n) = if n <= 1 { n } else {
 fib(n - 1) + fib(n - 2)
}
#fib(46)
```

1836311903

## while 语句

while 语句用于循环结构，满足条件时，不断执行循环体。

```
#while expression { cont-block }
```

上面代码中，如果表达式 expression 为真，就会执行 cont-block 代码块，然后再次判断 expression 是否为假；如果 expression 为假就跳出循环，不再执行循环体。



```
#let i = 0;
#while i < 10 { (i * 2,); i += 1 }
```

(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)

上面代码中，循环体会执行 10 次，每次将 i 增加 1，直到等于 10 才退出循环。

## for 语句

for 语句也是常用的循环结构，它迭代访问某个对象的每一项。

```
#for X in A { cont-block }
```

上面代码中，对于 A 的每一项，都执行 cont-block 代码块。在执行 cont-block 时，项的内容是 X。例如以下代码做了与之前循环相同的事情：

```
#for i in range(10) { (i * 2,) }
```

(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)

其中 range(10) 创建了一个内容为 (0, 1, 2, ..., 9) 一共 10 个值的数组。

## 使用内容块替代代码块

所有可以使用代码块的地方都可以使用内容块作为替代。

```
#for i in range(4) [阿巴].....
```

阿巴阿巴阿巴阿巴.....

## 使用 for 遍历字典

与数组相同，同理所有字典也都可以使用 for 遍历。此时，在执行 cont-block 时，Typst 将每个键值对以数组的形式交给你。键值对数组的第 0 项是键，键值对数组的第 1 项是对应的值。

```
#let cat = (neko-mimi: 2, "utterance": "喵喵喵", attribute: [kawaii~])
#for i in cat {
 [猫猫的 #i.at(0) 是 #i.at(1)\]
}
```

猫猫的 neko-mimi 是 2  
猫猫的 utterance 是 喵喵喵  
猫猫的 attribute 是 kawaii~

你可以同时使用「解构赋值」让代码变得更容易阅读：

```
#let cat = (neko-mimi: 2, "utterance": "喵喵喵", attribute: [kawaii~])
#for (特色, 这个) in cat [猫猫的 #特色 是 #这个\]
```

## break 语句和 continue 语句

无论是 while 还是 for，都可以使用 break 跳出循环，或 continue 直接进入下一次执行。

基于以下 for 循环，我们探索 break 和 continue 语句的作用。

```
#for i in range(10) { (i,) }
```

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

在第一次执行时，如果我们直接使用 break 跳出循环，但是在 break 之前就已经产生了一些值，那么 for 的结果是 break 前的那些值的「折叠」。

```
#for i in range(10) { (i,); (i + 1926,);
break }
```

(0, 1926)

特别地，如果我们直接使用 break 跳出循环，那么 for 的结果是 none。

```
#for i in range(10) { break }
```

在 break 之后的那些值将会被忽略：

```
#for i in range(10) { break; (i,); (i + 1926,); }
```

以下代码将收集迭代的所有结果，直到 i >= 5：

```
#for i in range(10) {
 if i >= 5 { break }
 (i,)
}
```

(0, 1, 2, 3, 4)

continue 有相似的规则，便不再赘述。我们举一个例子，以下程序输出在 range(10) 中不是偶数的数字：

```
#let 是偶数(i) = calc.even(i)
#for i in range(10) {
 if 是偶数(i) { continue }
 (i,)
}
```

(1, 3, 5, 7, 9)

事实上 break 语句和 continue 语句还可以在参数列表中使用，但本书非常不推荐这些写法，因此也不多做介绍：

```
#let add(a, b, c) = a + b + c
#while true { add(1, break, 2) }
```

3

## 控制函数返回值

你可以通过多种方法控制函数返回值。

### 占位符

早在上节我们就学习过了占位符，这在编写函数体表达式的时候尤为有用。你可以通过占位符忽略不需要的函数返回值。

以下函数获取数组的倒数第二个元素：

```
#let last-two(t) = {
 let _ = t.pop()
 t.pop()
}
#last-two((1, 2, 3, 4))
```

3

### return 语句

你可以通过 return 语句忽略表达式其余所有语句的结果，而使用 return 语句返回特定的值。

以下函数获取数组的倒数第二个元素：

```
#let last-two(t) = {
 t.pop()
 return t.pop()
}
#last-two((1, 2, 3, 4))
```

3



# 内容与样式

这是本章的最后一节。经过两节稍显枯燥的脚本教程，我们继续回到排版本身。

在《初识标记模式》中，我们学到了很多各式各样的内容。我们学到了段落、标题、代码片段……接着我们又花费了三节的篇幅，讲授了各式各样的脚本技巧。我们学到了字面量、变量、闭包……但是它们之间似乎隔有一层厚障壁，阻止了我们进行更高级的排版。是了，如果「内容」也是一种值，那么我们应该也可以更随心所欲地使用脚本操控它们。Typst 以排版为核心，应当也对「内容类型」有着精心设计。

本节主要介绍如何使用脚本排版内容。这也是 Typst 的核心功能，并在语法上与很多其他语言有着不同之处。不用担心，在我们已经学到了很多 Typst 语言的知识的基础上，本节也仅仅更进一步，教你如何真正以脚本视角看待一篇文档。

## 内容类型

我们已经学过很多元素：段落、标题、代码片段等。这些元素在被创建后都会被包装成为一种被称为「内容」的值。这些值所具有的类型便被称为「内容类型」。同时「内容类型」提供了一组公共方法访问元素本身。

乍一听，内容就像是一个“容器”将元素包裹。但内容又不太像是之前所学过的数组或字典那样的复合字面量，或者说这样不方便理解。事实上，每个元素都有各自的特点，但仅仅为了保持动态性，所有的元素都被硬凑在一起，共享一种类型。有两种理解这种类型的视角：从表象论，「内容类型」是一种鸭子类型；从原理论，「内容类型」提供了操控内容的公共方法，即它是一种接口，或称特征 (Trait)。

### 特性一：元素包装于「内容」

我们知道所有的元素语法都可以等价使用相应的函数构造。例如标题：

```
#repr([= 123]) \ // 语法构造
#repr(heading(depth: 1)[123]) // 函数构造
```

```
heading(depth: 1, body: [123])
heading(depth: 1, body: [123])
```

一个常见的误区是误认为元素继承自「内容类型」，进而使用以下方法判断一个内容是否为标题元素：

```
标题是 heading 类型 (伪)? #(type([= 123]) == heading)
```

```
标题是 heading 类型 (伪)? false
```

但两者类型并不一样。事实上，元素是「函数类型」，元素函数的返回值为「内容类型」。

```
标题函数的类型: #(type(heading)) \
标题的类型: #type([= 123])
```

```
标题函数的类型: function
标题的类型: content
```

这引出了一个重要的理念，Typst 中一切皆组合。Typst 中目前没有继承概念，一切功能都是组合出来的，这类似于 Rust 语言的概念。你可能没有学过 Rust 语言，但这里有一个冷知识：

Typst  $\Leftrightarrow$  Typ(setting Ru)st  $\Leftrightarrow$  Typesetting Rust

即 Typst 是以 Rust 语言特性为基础设计出的一个排版 (Typesetting) 语言。

当各式各样的元素函数接受参数时，它们会构造出「元素」，然后将元素包装成一个共同的类型：「内容类型」。heading 是函数而不是类型。与其他语言不同，没有一个 heading 类型继承 content。因此不能使用 `type([= 123]) == heading` 判断一个内容是否为标题元素。

### 特性二：内容类型的 func 方法

所有内容都允许使用 func 得到构造这个内容所使用的函数。因此，可以使用以下方法判断一个内容是否为标题元素：

```
标题所使用的构造函数: #([= 123]).func()
```

```
标题的构造函数是`heading`? #([= 123]).func()
== heading)
```

标题所使用的构造函数: heading

标题的构造函数是 heading? true

### 特性三：内容类型的 **fields** 方法

Typst 中一切皆组合，它将所有内容打包成「内容类型」的值以完成类型上的统一，而非类型继承。

但是这也有坏处，坏处是无法“透明”访问内部内容。例如，我们可能希望知道 heading 的级别。如果不提供任何方法访问标题的级别，那么我们就无法编程完成与之相关的排版。

为了解决这个问题，Typst 提供一个 fields 方法提供一个 content 的部分信息：

```
#([= 123]).fields()
```

(depth: 1, body: [123])

fields() 将部分信息组成字典并返回。如上图所示，我们可以通过这个字典对象进一步访问标题的内容和级别。

```
#([= 123]).fields().at("depth")
```

1

❗ 这里的“部分信息”描述稍显模糊。具体来说，Typst 只允许你直接访问元素中不受样式影响的信息，至少包含语法属性，而不允许你直接访问元素的样式。

### 特性四：内容类型与 **fields** 相关的糖

由于我们经常需要与 fields 交互，Typst 提供了 has 方法帮助我们判断一个内容的 fields 是否有相关的「键」。

```
使用`... in x.fields()`判断: #("text" in
`x`.fields()) \
等同于使用`has`方法判断: #(`x`.has("text"))
```

使用... in x.fields()判断: true  
等同于使用 has 方法判断: true

Typst 提供了 at 方法帮助我们访问一个内容的 fields 中键对应的值。

```
使用`x.fields().at()`获取值:
#(`www`.fields().at("text")) \
等同于使用`at`方法: #(`www`.at("text"))
```

使用 x.fields().at() 获取值: www  
等同于使用 at 方法: www

特别地，内容的成员包含 fields 的键，我们可以直接通过成员访问相关信息：

```
使用`at`方法: #(`www`.at("text")) \
等同于访问`text`成员: #(`www`.text)
```

使用 at 方法: www  
等同于访问 text 成员: www

## 内容的「样式」

我们接下来循着文本样式的脉络学习排版内容的语法。

重点 1：文本是段落的重要组成部分，与之对应的内容函数是 text。

我们知道一个函数可以有各种参数。那么我们从函数视角来看，内容的样式便由创建时参数的内容决定。例如，我们想要获得一段蓝色的文本：

```
#text("一段文本", fill: blue)
```

一段文本

fill: blue 是函数的参数，指定了文本的样式。

这个视角有助于我们更好的将对「样式」的需求转换为对函数的操控。例如，我们可以使用函数的 with 方法，获得一个固定样式的文本函数：

```
#let warning = text.with(fill: orange)
#warning[警告, 你做个人吧]
```

警告, 你做个人吧

## 上下文有关表达式

在介绍重要语法之前, 我们先来一道开胃菜。

```
#context text.size
```

10.5pt

## 「set」语法

重点 2: 一个段落主要是一个内容序列, 其中有可能很多个文本。

```
#repr([不止包含.....一个文本!])
```

sequence([不止包含],[...],[...],[一个文本!])

假设我们想要让一整个段落都显示成蓝色, 显然不能将文本一个个用 `text.with(fill: blue)` 构造好再组装起来。这个时候「set」语法出手了。「set」关键字后可以跟随一个函数调用, 为影响范围内所有函数关联的对应内容设置对应参数。

```
#set text(fill: blue)
一段很长的话可能不止包含.....一个文本!
- 似乎, 列表中也有文本。
```

一段很长的话可能不止包含.....一个文本!  
• 似乎, 列表中也有文本。

重点: 「set」的影响范围是其所在「作用域」内的后续内容。

我们紧接着来讲与之相关的, Typst 中最重要的概念之一: 「作用域」。

## 「作用域」

「作用域」是一个非常抽象的概念。但是理解他也并不困难。我们需要记住一件事, 那就是每个「代码块」创建了一个单独的「作用域」:

```
两只 #{
 [兔]
 set text(rgb("#ffd1dc").darken(15%))
 { [兔白]; set text(orange); [又白] }
 [, 真可爱]
}
```

两只兔兔白又白, 真可爱

从上面的染色结果来看, 粉色规则可以染色到[兔白]、[又白]和[真可爱], 橘色规则可以染色到[又白]但不能染色到[, 真可爱]。以内容的视角来看:

1. [兔]不是相对粉色规则的后续内容, 更不是相对橘色规则的后续内容, 所以它默认是黑色。
2. [兔白]是相对粉色规则的后续内容, 所以它是粉色。
3. [又白]同时被两个规则影响, 但是根据「执行顺序」, 橘色规则被优先使用。
4. [真可爱]虽然从代码先后顺序来看在橘色规则后面, 但不在橘色规则所在作用域内, 不满足「set」影响范围的设定。

我们说「set」的影响范围是其所在「作用域」内的后续内容, 意思是: 对于每个「代码块」, 「set」规则只影响到从它自身语句开始, 到该「代码块」的结束位置。

接下来, 我们回忆: 「内容块」和「代码块」没有什么不同。上述例子还可以以「内容块」的语法改写成:

```
两只 #[兔 #set text(fill:
rgb("#ffd1dc").darken(15%))
 #[兔白 #set text(fill: orange)
 又白], 真可爱
]
```

两只兔兔白又白, 真可爱

由于断行问题, 这不方便阅读, 但从结果来看, 它们确实是等价的。

最后我们再回忆：文件本身是一个「内容块」。

```
两小只, #set text(fill: orange)
真可爱
```

两小只, 真可爱

针对文件，我们仍重申一遍「set」的影响范围。其影响等价于：对于文件本身，顶层「set」规则影响到该文件的结束位置。

也就是说，include 文件内部的样式不会影响到外部的样式。

## 变量的可变性

理解「作用域」对理解变量的可变性有帮助。这原本是上一节的内容，但是前置知识包含「作用域」，故在此介绍。

话说 Typst 对内置实现的所有函数都有良好的自我管理，但总免不了用户打算写一些逆天的函数。为了保证缓存计算仍较为有效，Typst 强制要求用户编写的**所有函数**都是纯函数。这允许 Typst 有效地缓存计算，在相当一部分文档的编译速度上，快过 LaTeX 等语言上百倍。

你可能不知道所谓的纯函数为何物，本书也不打算讲解什么是纯函数。关键点是，涉及函数的**纯度**，就涉及到变量的可变性。

所谓变量的可变性是指，你可以任意改变一个变量的内容，也就是说一个变量默认是可变的：

```
#let a = 1; #let b = 2;
#((a, b) = (b, a)); #a, #b \
#for i in range(10) { a += i }; #a, #b
```

2, 1  
47, 1

但是，一个函数的函数体表达式不允许涉及到函数体外的变量修改：

```
#let a = 1;
#let f() = (a += 1);
#f()
```

error: variables from outside the function are  
read-only and cannot be modified

这是因为纯函数不允许产生带有副作用的操作。

同时，传递进函数的数组和字典参数都会被拷贝。这将导致对参数数组或参数字典的修改不会影响外部变量的内容：

```
#let a = (1,); #a \ // 初始值
#let add-array(a) = (a += (2,));
#add-array(a); #a \ // 函数调用无法修改变量
#(a += (2,)); #a \ // 实际期望的效果
```

(1,)  
(1,)  
(1, 2)

准确地来说，数组和字典参数会被写时拷贝。所谓写时拷贝，即只有当你期望修改数组和字典参数时，拷贝才会随即发生。

为了“修改”外部变量，你必须将修改过的变量设法传出函数，并在外部更新外部变量。

```
#let a = (1,); #a \ // 初始值
#let add-array(a) = { a.push(2); a };
#(a = add-array(a)); #a \ // 返回值更新数组
```

(1,)  
(1, 2)

一个函数是纯的，如果：

1. 对于所有相同参数，返回相同的结果。
2. 函数没有副作用，即局部静态变量、非局部变量、可变引用参数或输入/输出流等状态不会发生变化。

本节所讲述的内容是对第二点要求的体现。

## 「set if」语法

回到「set」语法的话题。假设我们脚本中设置了当前文档是否处于暗黑主题，并希望使用「set」规则感知这个设定，你可能会写：

```
#let is-dark-theme = true
#if is-dark-theme {
 set rect(fill: black)
 set text(fill: white)
}

#rect([wink!])
```



根据我们的知识，这应该不起作用，因为 if 后的代码块创建了一个新的作用域，而「set」规则只能影响到该代码块内后续的代码。但是 if 的 then 和 else 一定需要创建一个新的作用域，这有点难办了。

set if 语法出手了，它允许你在当前作用域设置规则。

```
#let is-dark-theme = true
#set rect(fill: black) if is-dark-theme
#set text(fill: white) if is-dark-theme
#rect([wink!])
```



解读 #set rect(fill: black) if is-dark-theme。它的意思是，如果满足 is-dark-theme 条件，那么设置相关规则。这其实与下面代码“感觉”一样。

```
#let is-dark-theme = true
#if is-dark-theme {
 set rect(fill: black)
}

#rect([wink!])
```



区别仅仅在 set if 语法确实从语法上没有新建一个作用域。这就好像一个“规则怪谈”：如果你想要让「set」规则影响到对应的内容，就想方设法满足「set」影响范围的要求。

## 「内容」是一棵树

重点 3：「内容」是一棵树，这意味着你可以“攀树而行”。

Typst 对代码块有着的一系列语法设计，让代码块非常适合描述内容。又由于作用域的性质，最终代码块让「内容」形成为一棵树。

「内容」是一棵树。一个 main.typ 就是「内容」的一再嵌套。即便不使用任何标记语法，你也可以创建一个文档：

```
#let main-typ() = {
 heading("生活在 Content 树上")
 {
 [现代社会以海德格尔的一句]
 [“一切实践传统都已经瓦解完了”]
 [为嚆矢。]
 } + parbreak()
 [...] + parbreak()
 [在孜孜矻矻以求生活意义的道路上，对自己的期望本就是在与家庭与社会对接中塑型的动态过程。]
 [而我们的底料便是对不同生活方式、不同角色的觉感与体认。]
 [...]
}
#main-typ()
```

# 生活在 Content 树上

现代社会以海德格尔的一句“一切实践传统都已经瓦解完了”为嚆矢。

...

在孜孜矻矻以求生活意义的道路上，对自己的期望本就是在与家庭与社会对接中塑型的动态过程。而我们的底料便是对不同生活方式、不同角色的觉感与体认。...

## plain-text，以及递归函数

如果我们想要实现一个函数 `plain-text`，它将一段文本转换为字符串。它便可以在树上递归遍历：

```
#let plain-text(it) = if it.has("text") {
 it.text
} else if it.has("children") {
 ("", ..it.children.map(plain-text)).join()
} else if it.has("child") {
 plain-text(it.child)
} else { ... }
```

所谓递归是一种特殊的函数实现技巧：

- 递归总有一个不调用其自身的分支，称其为递归基。这里递归基就是返回 `it.text` 的分支。
  - 函数体中包含它自身的函数调用。例如，`plain-text(it.child)`便再度调用了自身。
- 这个函数充分利用了内容类型的特性实现了遍历。首先它使用了 `has` 函数检查内容的成员。

如果一个内容有孩子，那么对其每个孩子都继续调用 `plain-text` 函数并组合在一起：

```
#if it.has("children") { ("", ..it.children.map(plain-text)).join() }
#if it.has("child") { plain-text(it.child) }
```

限于篇幅，我们没有提供 `plain-text` 的完整实现，你可以试着在课后完成。

## 鸭子类型

这里值得注意的是，`it.text` 具有多态行为。即便没有继承，这里通过一定动态特性，允许我们同时访问「代码片段」的 `text` 和「文本」的 `text`。例如：

```
#let plain-mini(it) = if it.has("text")
{ it.text }
#repr(plain-mini(`代码片段中的 text`)) \
#repr(plain-mini([文本中的 text]))
```

"代码片段中的 text"  
"文本中的 text"

这也便是我们在「内容类型」小节所述的鸭子类型特性。如果「内容」长得像文本（鸭子），那么它就是文本。

## 「内容」是一棵树 (Cont.)

① 利用「内容」与「树」的特性，我们可以在 Typst 中设计出更多优雅的脚本功能。

### CeTZ 的「树」

CeTZ 利用内容树制作“内嵌的 DSL”。CeTZ 的 `canvas` 函数接收的不完全是内容，而是内容与其 IR 的混合。

例如它的 `line` 函数的返回值，就完全不是一个内容，而是一个无法窥视的函数。

```
#import "@preview/cetz:0.2.0"
#repr(cetz.draw.line((0, 0), (1, 1), fill:
blue))
```

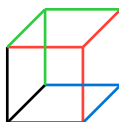
((..) => ...)



当你产生一个“混合”的内容并将其传递给 `cetz.canvas`, CeTZ 就会像 `plain-text` 一样遍历你的混合内容, 并加以区分和处理。如果遇到他自己特定的 IR, 例如 `cetz.draw.line`, 便将其以特殊的方式转换为真正的「内容」。

使用混合语言, 在 Typst 中可以很优雅地画多面体:

```
#import "@preview/cetz:0.2.0"
#align(center, cetz.canvas({
 // 导入 cetz 的 draw 方言
 import cetz.draw: *; import cetz.vector: add
 let neg(u) = if u == 0 { 1 } else { -1 }
 for (p, c) in (
 ((0, 0, 0), black), ((1, 1, 0), red), ((1, 0, 1), blue), ((0, 1, 1), green),
) {
 line(add(p, (0, 0, neg(p.at(2)))), p, stroke: c)
 line(add(p, (0, neg(p.at(1)), 0)), p, stroke: c)
 line(add(p, (neg(p.at(0)), 0, 0)), p, stroke: c)
 }
}))
```



## PNG.typ 的树

我们知道「内容块」与「代码块」没有什么本质区别。

如果我们可以基于「代码块」描述一棵「内容」的树, 那么一张 PNG 格式的图片似乎也可以被描述为一棵「字节」的树。

通过代码块语法, 你可以在 Typst 中拼接字节, 依像素地创建一张 PNG 格式的图片:

```
// Origin: https://typst.app/project/r0SkRmsZYIYNxjs6Q712aP
#import "png.typ": *
#let prelude = (0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A)
#let ihdr(w, h) = chunk("IHDR", be32(w) + be32(h) + (8, 2, 0, 0, 0))
#let idat(lines) = chunk("IDAT", {
 let data = lines.map(line => (0x00,) + line).flatten()
 let len = le32(data.len()).slice(0, 2)
 (0x08, 0x1D, 0x01); len; len.map(xor.with(0xFF)); data; be32(adler32(data))
})
#align(center, box(width: 25%, image.decode(bytes({
 let (w, h) = (8, 8)
 prelude; ihdr(w, h); idat(for y in range(h) { (for x in range(w) {
 (calc.floor(256 * x / w), 128, calc.floor(256 * y / h))
 },) }); chunk("IEND", ())
}))))
```



## 「show」语法

「set」语法是「show set」语法的简写。因此, 「show」语法显然可以比 `set` 更强大。

```
#show: set text(fill: blue)
wink!
```

wink!

我们可以看到「show」语法由两部分组成, 由冒号分隔。

show 的右半部分是一个函数, 表示选择文档的一部分以作修改。

❗ 你可能会问，先姑且不问函数要怎么写，难道 `set text(fill: blue)` 也能算一个函数吗？

事实上，`set` 规则是「内容类型」，它接受一个样式和一个内容，返回一个 `styled` 内容：

```
#let x = [#set text(fill: blue)]
#x.func()
```

styled

以下使用方法非常黑客，请最好不要在你的文档中包含这种代码。仅用于理解：

```
#let styled = [#set text(blue)].func()
#let styles = text("", red).styles
#styled([Red Text], styles)
```

Red Text

1. 第一行代码，我们说 `func` 方法返回内容函数本身，这里便返回了一个内部的函数 `styled`。
2. 第二行代码，这里我们从 `text` 内容上找到了它关于设置红色文本的样式（参数）。
3. 第三行代码，把一个内容及一个无论如何从某处得到了的样式传递给 `styled` 函数。
4. 最终我们构造出了一个真实的红色文本。

`show` 的左半部分是选择器，表示选择文档的一部分以作修改。它作用于「作用域」内的后续所有被选择器选中的内容。

如果选择器为空，则默认选择后续所有内容。这也是「`set`」语法对应规则的原理。如果选择器不为空，那么因为我们还没讲解选择器，所以这里不作过多讲解。

但有一种选择器比较简单易懂。我们可以将内容函数作为选择器，选择相应内容作影响。

以下脚本设置所有代码片段的颜色：

```
#show raw: set text(fill: blue)
被秀了的`代码片段`！
```

被秀了的代码片段！

以下脚本设置所有数学公式的颜色，但同时也修改代码片段的颜色：

```
#show raw: set text(red)
#show math.equation: set text(blue)
#let dif2(x) = math.op(math.Delta + x)
一个公式: $ \sum_{f \in S(x)} (f) \Delta x $
#`refl`; (f) dif2(x) $
```

一个公式：

$$\sum_{f \in S(x)} \text{refl}(f) \Delta x$$

我们说，`show` 的右半部分是一个函数，表示选择文档的一部分以作修改。除了直接应用 `set`，应该可以有其他很多操作。现在是时候解锁 `Typst` 强大能力了。

这个函数接受一个参数：参数是未打包（unpacked）的内容；这个函数返回一个任意内容。

以下示例说明它接受一个未打包的内容。对于代码片段，我们使用「`show`」语法择区其中第二行：

```
#show raw: it => it.lines.at(1)
获取代码片段第二行内容: ```typ
#{
set text(fill: true)
}
```
```

获取代码片段第二行内容：

```
set text(fill: true)
```

在《内容类型的特性》中，我们所接触到的已经打包（packed）的代码片段并不包含 `lines` 字段。在打包后，内部大部分信息已经被屏蔽了。

以下示例说明它可以返回任意内容。这里我们选择语言为 `my-calc` 的代码片段，执行并返回一个非代码片段：

```
#show raw.where(lang: "my-calc"): it =>
eval(it.text)
嵌入一个计算器语言，计算`1*2+2*(2+3)`: ```my-
calc 1*2+2*(2+3)```
```

嵌入一个计算器语言，计算 `1*2+2*(2+3)`：12

由于 `show` 的右半部分只要求接受内容并返回内容，我们可以有非常优雅的写法，使用一些天然满足要求的函数。

以下规则将每个代码片段用方框修饰：

```
#show raw: rect
` ` QwQ ` `
```

QwQ

以下规则将每个代码片段用蓝色方框修饰：

```
#show raw: rect.with(stroke: blue)
` ` QwQ ` `
```

QwQ

总结

本节仅以文本、代码块和内容块为例讲清楚了文件、作用域、「`set`」语法和「`show`」语法。为了拓展广度，你还需要查看《基本参考》中各种元素的用法，这样才能随心所欲排版任何「内容」。

习题

1. 实现内容到字符串的转换 `plain-text`：对于文中出现的 `main-tyt()` 内容，它输出：

生活在 Content 树上现代社会以海德格尔的一句“一切实践传统都已经瓦解完了”为嚆矢。… 在孜孜矻矻以求生活意义的道路上, 对自己的期望本就是在与家庭与社会对接中塑型的动态过程。而我们的底料便是对不同生活方式、不同角色的觉感与体认。…

2. 实现字数统计 `word-count`：对于文中出现的 `main-tyt()` 内容，它输出：

102


3. 思考题： `plain-text` 有何局限性？为什么在 `show` 规则影响下， `word-count` 输出分别为 4 和 5？

```
#let show-me-the(it) = {
  it + [ 的字数统计为 #word-count(it) ]
}
#show-me-the([#show raw: it => {"123";
it}; `一段文本`]) \
#show-me-the([#show: it => {"123"; it};
一段文本])
```

123 一段文本 的字数统计为 4
123 一段文本 的字数统计为 5



Part.3 基础教程 — 排版 II



度量与布局

本章我们再度回到排版专题，拓宽制作文档的能力。

长度类型

Typst 有三种长度单位，它们是：绝对长度（absolute length）、相对长度（relative length）与上下文有关长度（context-sensitive length）。

1. 绝对长度：人们最熟知的长度单位。例如，`1cm` 恰等于真实的一厘米。
 2. 相对长度：与父元素长度相关联的长度单位，例如 `70%` 恰等于父元素高度或宽度的 70%。
 3. 上下文有关长度：与样式等上下文有关的长度单位，例如 `1em` 恰等于当前位置设定的字体长度。
- 掌握不同种类的长度单位对构建期望的布局非常重要。它们是相辅相成的。

绝对长度

目前 Typst 一共提供四种绝对长度。除了公制长度单位 `1cm` 与 `1mm` 与英制长度单位 `1in`，Typst 还提供排版专用的长度单位“点”，即 `1pt`。

点（英语：*point*），*pt*，是印刷所使用的长度单位，用于表示字型的大小，也用于余白（字距、行距）等其他版面构成要素的长度。作为铸字行业内部的一个专用单位，1 点的长度在世界各地、各个时代曾经有过不同定义，并不统一。当代最通行的是广泛应用于桌面排版软件的 DTP 点，72 点等于 1 英寸（ $1\text{ point} = \frac{1}{72}\text{ inch} = \frac{1}{72} \times 25.4\text{ mm} = 0.352777\cdots\text{ mm}$ ）。中国传统字体排印上的字号单位是号，而后采用“点”“号”兼容的体制。

——[维基百科：点 \(印刷\)](#)

Typst 会将你提供的任意长度单位都统一成点单位，以便进行长度运算。

	=?pt	=?mm	=?cm	=?in
<code>1pt</code>	1	0.35	0.04	0.01
<code>1mm</code>	2.83	1	0.1	0.04
<code>1cm</code>	28.35	10	1	0.39
<code>1in</code>	72	25.4	2.54	1

绝对长度的运算

长度单位可以参与任意多个浮点值的运算。一个长度表达式是合法的当且仅当运算结果保持长度量纲。请观察下列算式，它们都可以编译：

```
#(1cm * 3), #(1cm / 3), #(2 * 1cm * 3 / 2), #(1cm + 3in)
```

85.04pt, 9.45pt, 85.04pt, 244.35pt

请观察下列算式，它们都不能编译：

```
#(1cm + 3), #(3 / 1cm), #(1cm * 1cm)
```

所谓保持长度量纲，即它存在一系列判别规则：

- 由于 `1cm` 与 `3in` 量纲均为长度量纲（m），它们之间可以进行加减运算。
- 由于 `3` 无量纲，`1cm` 与 `3` 之间不能进行加减运算。

进一步，通过量纲运算，可以判断一个长度算术表达式是否合法：

长度表达式	量纲运算	检查合法性	判断结果
<code>1cm * 3</code>	$m \cdot 1 = m$	$m = m$	合法
<code>1cm / 3</code>	$m / 1 = m$	$m = m$	

$3 / 1\text{cm}$	$1 / m = m^{-1}$	$m^{-1} = m$	非法
$1\text{cm} * 1\text{cm}$	$m \cdot m = m^2$	$m^2 = m$	

绝对长度的转换

你可以使用 `length` 类型上的「方法」实现不同单位到浮点数的转换：

```
#1cm 是 #1cm.pt() 点 \
#1cm 是 #1cm.inches() 英尺
```

28.35pt 是 28.346456692913385 点
28.35pt 是 0.3937007874015748 英尺

你可以使用乘法实现浮点数到长度上的转换，例如 `28.3465 * 1pt`：

```
#1cm 是 #(28.3465 * 1pt).cm() 厘米
```

28.35pt 是 1.0000015277777776 厘米

相对长度

有两种相对长度（Relative Length），一是百分比（Ratio），一是分数比（Fraction）。

百分比

当「百分比」用作长度时，其实际值取决于父容器宽度：

```
#let p(w, f, ..args) = box(
  width: w, height: 10pt, fill: f, ..args)
4 比 6: #p(100pt, blue, p(40%, red))
```

4 比 6: 

Typst 还支持以「分数比」作长度单位。当分数比作长度单位时，Typst 按比例分配长度。




```
4 比 6: #p(100pt, blue,
  p(4fr, red) + p(6fr, blue))
```

4 比 6: 

结合代码与图例理解，`N fr` 代表：在总的比例中，这个元素应当占有其中 `N` 份长度。

当同级元素既有分数比长度元素，又有其他长度单位元素时，优先将空间分配给其他长度单位元素。

```
绿色先占 60%: #p(100pt, blue, p(1fr, red) +
  p(2fr, blue) + p(60%, green)) \
绿色先占 110%: #p(100pt, blue, p(1fr, red) +
  p(1fr, blue) + p(110%, green)) \
红色先占 30pt: #p(100pt, blue, p(30pt, red)
  + p(1fr, blue) + p(110%, green))
```

绿色先占 60%: 
绿色先占 110%: 
红色先占 30pt: 

建议结合下文中 `grid` 布局关于长度的使用，加深对相对长度的理解。

上下文有关长度

目前 Typst 仅提供一种上下文有关长度，即当前上下文中的字体大小。历史上，定义该字体中大写字母 `M` 的宽度为 `1em`，但是现代排版中，`1em` 可以比 `M` 的宽度要更窄或者更宽。

上下文有关长度是与相对长度相区分的。区别是上下文有关长度的取值从「样式链」获取，而相对长度相对于父元素宽度。事实上 `1em` 的具体值可以通过上下文有关表达式获取：

```
#let _1em = context measure(
  line(length: 1em)).width
#text(size: 10pt, [1em 等于] + _1em) \
#text(size: 20pt, [1em 等于] + _1em) \
```

1em 等于 10pt
1em 等于 20pt

相比较，`1em` 更好用一点，因为 `text.size` 只允许在上下文有关表达式内部使用。

混合长度

以上所介绍的各种长度可以通过「加号」任意混合成单个长度的值，其长度的值为每个分量总和：

```
#(1pt + 1em + 100%)
```

```
100% + 1pt + 1em
```

长度的内省或评估

你可以通过 `measure` 获取一个长度在当前位置的具体值：

```
#let length-of(l) = measure(  
  line(length: l)).width  
#context [长度等于 #length-of(1em+1pt)。]
```

长度等于 11.5pt。

但是该方式是不被推荐的，因为一个长度值中的相对长度分量会被评估为 0pt，从而导致计算失真：

```
长度等于 #context length-of(1em+1pt+100%)。
```

长度等于 11.5pt。

这是因为在评估的时候，`measure` 没有为内容锚定一个“父元素”。

一种更为鲁棒的方法是使用 `layout` 函数获取 `layout` 位置的宽度和高度信息：

```
长度等于 #box(width: 1em+1pt+100%,  
  layout(l => l.width))
```

长度等于 213.02pt

但是使用 `layout` 会导致布局的多轮迭代，有可能严重降低编译性能。

布局概览与布局模型

Typst 的布局引擎仍未完成，其主要缺失或不足的内容为：

1. 修饰段落容器的行或字符的能力。
2. 将段落容器的接口暴露给用户的能力。
3. 浮动布局的能力。
4. 更灵活的 Layout Splitter。目前仅支持 columns（多栏布局）的能力。

Typst 的布局代码风格是用一系列容器函数包装的内容，再将整个内容交给布局引擎反复迭代求解。容器就是一类特殊的「元素」，它并不真正具备具体的内容，而仅仅容纳一个或多个「内容」，**以便布局**。这与 JSX 有些相似。

针对 PDF 页面模型，Typst 构建了完整的容器层次。Todo：用 `cetz` 绘制层次：

1. page，一个 PDF 有很多页。
2. par，一个文档中有很多段。
3. block，一页/段中有很多块。
4. box，一个块中有很多行内元素。
5. sequence，很多个元素可以组成一个元素序列。
6. 各种基础元素。

page

每个 `page` 都单独产生一个单独的页面。并且若你希望将 Typst 文档导出为 PDF，其恰好为 PDF 中单独的一页。

任何内容都会至少产生新的一页：

```
// 第一页  
A
```

每个 `pagebreak` 函数（分页函数）将会产生新的一页：

```
// 第一页  
A  
#pagebreak()  
// 第二页  
B
```

```
// 第一页
A
#pagebreak()
// 第二页
#pagebreak()
// 第三页
B
```

你可以使用 `#pagebreak(weak: true)` 产生弱的分页。这里弱的意思是：假设当前页面并没有包含任何内容，`pagebreak(weak: true)` 就不会创建新的页面。你可以理解为弱的分页意即非必要不分页。

```
// 第一页
A
#pagebreak(weak: true)
#pagebreak(weak: true)
// 第二页
B
```

每个 `page` 函数都会单列新的一页：

```
// 第一页
A
#page[
  // 第二页
  A
]
// 第三页
B
```

如果 `page` 的前后已经是空页，则 `page` 不会导致前后产生空页。

```
// 这里不分页
#page[
  // 第一页
  A
] // 这里不分页
#page[
  // 第二页
  A
] // 这里不分页
```

你可以理解为 `page` 前后自动产生弱的分页，其前后**非必要不分页**，上例相当于：

```
#pagebreak(weak: true)
// 第一页
A
#pagebreak(weak: true)
#pagebreak(weak: true)
// 第二页
A
#pagebreak(weak: true)
```

所以 `page` 与 `pagebreak` 是互通的，你可以随意使用你喜欢的风格创建新的页面。

注意：设置 `page` 样式将导致新的弱分页：

```
// 第一页
A
#set page(..)
// 第二页
B
```

等价于：

```
A
#pagebreak(weak: true)
#set page(..)
#pagebreak(weak: true)
B
```

par

每个 par 都单独产生一个单独的段落。

par 的特性与 page 完全相同，其前后非必要不分段。

```
#par[
  // 第一段
  A
]
```

A

parbreak 与 pagebreak 也是除了同理，除了 parbreak 一定是弱分段，并且没有可以指定的 weak 参数：

```
// 第一段
A
#parbreak()
#parbreak()
// 第二段
B
```

A

B

很多元素都会在其前后自动产生新的段落，例如 figure 和 heading。

block

block 是布局中的块状元素。块状元素会在布局中自成一页。你几乎可以将 block 理解为某种程度上的分段：

```
// 第一段
#block(fill: blue)[A]
#block(fill: blue)[B]
```

A

B

事实上 par 本来就由 block 组成。你可以通过对 block 设置规则影响到段间距：

```
#set block(spacing: 0.5em)
A
#parbreak()
#parbreak()
B
```

A

B

box

box 是布局中的行内块元素。行内块元素会在布局中自成一页。所谓行内块即其自成一页但不会立即导致换行。如下所示：

```
#rect(width: 100pt)[a a a #box(fill: blue)
[A A A A A]a a a]
```

a a a A A A A A a a a

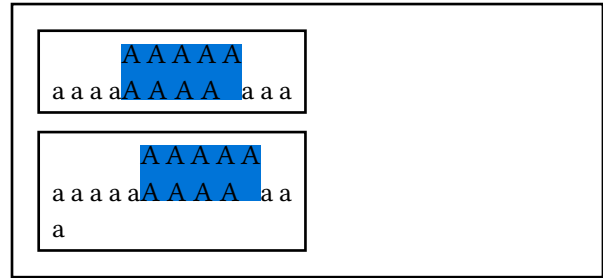
box 并不意味着不换行。box 的宽度默认占满父元素内宽度的 100%。box 内部的文本在其内部继续换行。如下所示，当文字太多时，将导致其在 box 内部换行：

```
#rect(width: 100pt)[#box(fill: blue)[A A A
A A A A A A A A A]]
```

A A A A A A A A A
A A A A

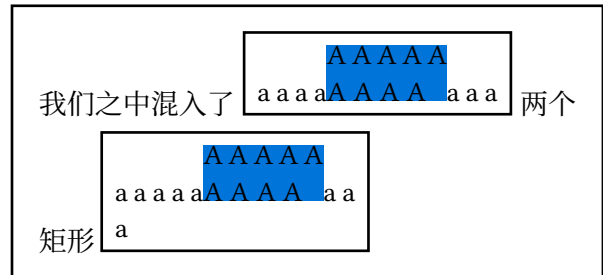
box 整体必须处于段落中的同一行：

```
#rect(width: 100pt)[a a a a#box(fill:
blue, width: 50%)[A A A A A A A A]a a a]
#rect(width: 100pt)[a a a a a#box(fill:
blue, width: 50%)[A A A A A A A A]a a a]
```



一个妙用是，你可以将“块元素”包裹一层 box，使得这些块元素位于段落中的同一行：

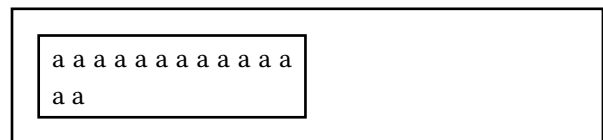
```
我们之中混入了
#box(rect(width: 100pt)[a a a a#box(fill:
blue, width: 50%)[A A A A A A A A]a a
a])
两个矩形
#box(rect(width: 100pt)[a a a a
a#box(fill: blue, width: 50%)[A A A A A A
A A A]a a a])
```



sequence

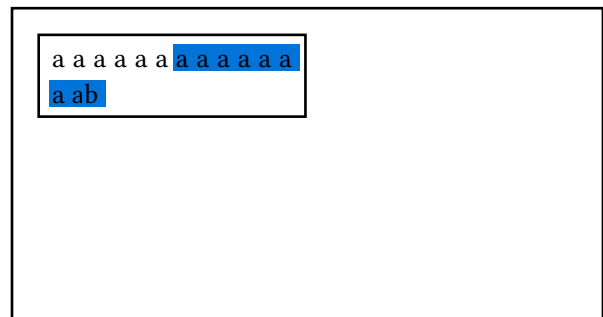
sequence 是布局中最松散的组织。事实上，它正是我们在《基础文档》中就学过的「内容块」。它起不到任何效果，仅仅容纳一系列其他内容。

```
#rect(width: 100pt)[#[a a a a a a] #[a a a
a a a a a] ]
```



我们来温习一下。sequence 主要的作用是可以储存一段内容供函数使用。使用 sequence 的一个好处是允许「标签」准确地选中一段内容。例如，以下代码自制了 highlight 效果：

```
#show <fill-blue>: it => {
  show regex("[\\w]"): it => {
    box(place(dx: -0.1em, dy: -0.15em,
    box(fill: blue, width: 0.85em, height:
    0.95em)) + it)
  }
  it
}
#rect(width: 100pt)[#[a a a a a a] #[a a a
a a a a a ab] <fill-blue> ]
```



「布局分割及其断点」

「布局分割器」(Splitter) 将所持有内容分成多段并放置于页面上。

目前 Typst 唯一提供的分割器是 columns，其实现了多栏布局。因为缺乏好用的分割器，有很多布局都难以在 Typst 中实现，例如浮动布局。但注意，这不代表现在的 Typst 不支持各种特殊布局。已经有外部库帮助你实现了首字母下沉，浮动布局等特殊布局。

你可以使用 columns 在任意容器内创建多栏布局，并用 colbreak 分割每栏。

```
#columns(2)[
  #lorem(13)
  #colbreak()
  #lorem(10)
]
```

Lorem ipsum dolor Lorem ipsum dolor
sit amet, consectetur sit amet, consectetur
adipiscing elit, sed adipiscing elit, sed do.
do eiusmod tempor
incididunt.

尽管还不是很完善，在与页面的结合中，Typst 为 page 提供了一个 columns 参数。该 columns 允许你设置跨页的多栏布局。

「布局编排」

「布局编排器」(Arranger) 将多个内容组成一个新的带有布局的整体。目前，Typst 提供了两种零维布局 pad 和 align，一种一维布局 stack，以及一种二维布局 grid。

pad

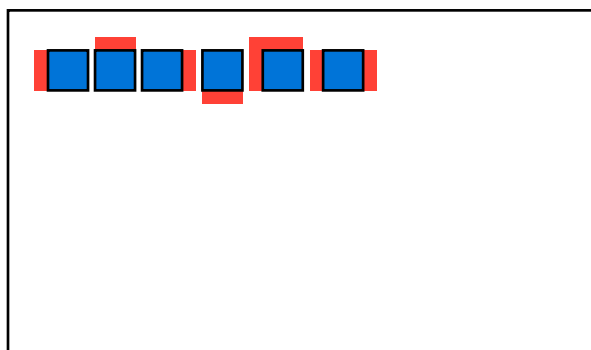
你可以使用 pad 函数为内部元素设置 outer padding。pad 接受一个内容，返回一个添加 padding 的新内容。默认情况下，pad 函数不添加任何 padding。

```
#let square = rect(fill: blue, width: 15pt, height: 15pt)
#box(fill: red, square)
#box(fill: red, pad(square))
```



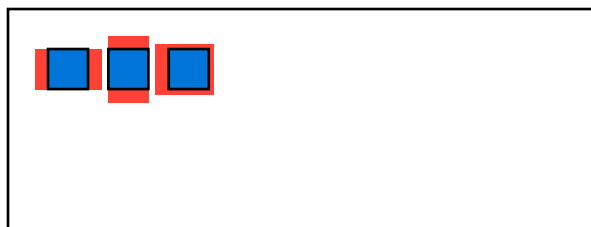
与 CSS 的 padding 属性一样，你可以使用 left、top、right、bottom 设置四个方向的 padding：

```
#let square = rect(fill: blue, width: 15pt, height: 15pt)
#box(fill: red, pad(left: 5pt, square))
#box(fill: red, pad(top: 5pt, square))
#box(fill: red, pad(right: 5pt, square))
#box(fill: red, baseline: 5pt, pad(bottom: 5pt, square))
#box(fill: red, pad(left: 5pt, top: 5pt, square))
#box(fill: red, pad(left: 5pt, right: 5pt, square))
```



特别地，你可以使用 x 同时设置 left 和 right 方向的 padding，y 同时设置 top 和 bottom 方向的 padding，以及 rest 设置“其余”方向的 padding。

```
#let square = rect(fill: blue, width: 15pt, height: 15pt)
#box(fill: red, pad(x: 5pt, square))
#box(fill: red, baseline: 5pt, pad(y: 5pt, square))
#box(fill: red, baseline: 2pt, pad(left: 5pt, rest: 2pt, square))
```



align

你可以使用 align 函数为内部元素设置相对于父元素的对齐方式。默认情况下，align 函数使用默认对齐方式 (start+top)。

```
#let square = rect(fill: red, width: 15pt, height: 15pt)
#let abox = box.with(fill: gradient.radial(..color.map.mako), width: 20pt, height: 20pt)
#abox(square)
#abox(align(square))
```



与上一节介绍的 pad 类似，你可以使用 left、top、right、bottom 设置四个方向的 alignment，并任意使用加号 (+) 组合出你想要的二维对齐方式：

```
#abox(align(left + top, square))
#abox(align(right + top, square)) \
#abox(align(left + bottom, square))
#abox(align(right + bottom, square))
```



除了以上四种基础对齐，Typst 还额外提供两套对齐方式。第一套是居中对齐，你可以使用 `center` 实现水平居中对齐，以及 `horizon` 实现垂直居中对齐。

```
#abox(align(center, square))
#abox(align(horizon, square))
#abox(align(center + horizon, square))
```



第二套与文本流相关。你可以使用 `start` 实现与文本流方向一致的对齐，以及 `end` 实现与文本流方向相反的对齐。

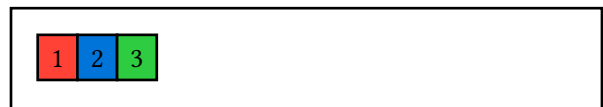
```
#let se = abox(align(start, square)) +
  abox(align(end, square))
#se \
#set text(dir: rtl)
#se
```



stack

与 HTML 中的 `flex` 容器类似，`stack` 可以帮你实现各种一维布局。顾名思义，`stack` 将其内部元素在页面上按顺序按方向排列。

```
#let rects = (rect(fill: red)[1],
  rect(fill: blue)[2], rect(fill: green)[3])
#stack(dir: ltr, ..rects)
```



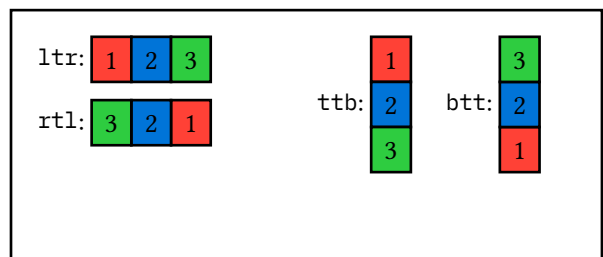
`stack` 主要只有两个可以控制的参数。

其一是排列的方向，参数对应为 `dir`。一共有四种方向可以选择，分别是：

- `ltr`: 即 left to right, 从左至右。
- `rtl`: 即 right to left, 从右至左。
- `tbt`: 即 top to bottom, 从上至下。
- `btt`: 即 bottom to top, 从下至上。

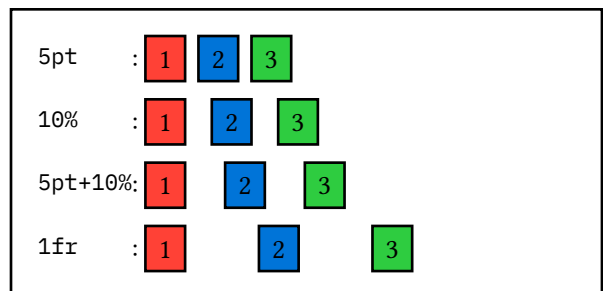
方向默认为 `tbt`。

```
#columns(2)[
  `ltr`: #box(stack(dir: ltr, ..rects)) \
  `rtl`: #box(stack(dir: rtl, ..rects))
  #colbreak()
  `tbt`: #box(stack(/* 默认值为: dir: tbt,
  */ ..rects)) #h(1em) `btt`:
#box(stack(dir: btt, ..rects))
]
```



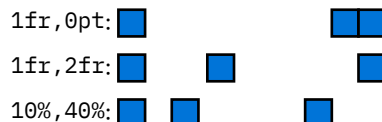
其二是排列的间距，参数对应位 `spacing`。你可以使用各种长度为 `stack` 内部元素制定合适的间距。

```
#let stack = stack.with(dir: ltr)
#let example(s) = box(width: 100pt,
  stack(spacing: s, ..rects))
`5pt` `: #example(5pt) \
`10%` `: #example(10%) \
`5pt+10%` `: #example(5pt+10%) \
`1fr` `: #example(1fr) \
```



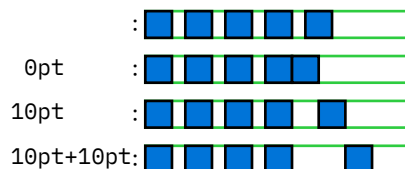
你也可以直接将长度作为参数传入，为每一个元素前后微调间距。

```
#let stack = stack.with(dir: ltr)
#let r = rect(fill: blue)[ ]
`1fr,0pt`: #box(width: 100pt, stack(r,
1fr, r, r)) \
`1fr,2fr`: #box(width: 100pt, stack(r,
1fr, r, 2fr, r)) \
`10%,40%`: #box(width: 100pt, stack(r,
10%, r, 40%, r)) \
```



当同时声明 spacing 参数和长度参数时，长度参数会覆盖 spacing 设置：

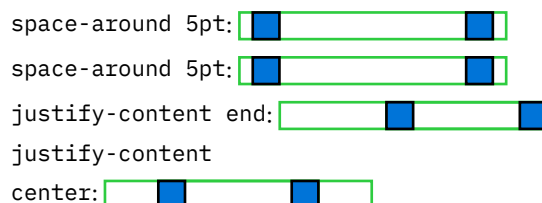
```
#show stack: box.with(width: 100pt,
stroke: green)
#let stack = stack.with(dir: ltr)
#let r = rect(fill: blue)[ ]
` `: #stack(spacing: 5pt, r, r, r,
r, r) \
` 0pt `: #stack(spacing: 5pt, r, r, r,
r, 0pt, r) \
`10pt `: #stack(spacing: 5pt, r, r, r,
r, 10pt, r) \
`10pt+10pt`: #stack(spacing: 5pt, r, r, r,
r, 10pt, 10pt, r) \
```



stack+align/pad

你可以将 stack 与 align/pad 相结合，实现更多一维布局。以下例子对应 CSS 相关属性：

```
#let box = box.with(width: 100pt, stroke:
green)
#let stack = stack.with(dir: ltr, spacing:
1fr)
#let r = rect(fill: blue)[ ]
`space-around 5pt`: #box(stack(5pt, r, r,
5pt)) \
`space-around 5pt`: #box(pad(x: 5pt,
stack(r, r))) \
`justify-content end`: #box(align(end,
box(width: 60pt, stack(r, r)))) \
`justify-content center`:
#box(align(center, box(width: 60pt,
stack(r, r)))) \
```

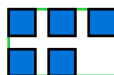


上例提示我们可以组合多种布局容器实现特定的布局效果。

grid

与 HTML 中的 flex 容器类似，grid 可以帮你实现各种二维布局。如下：

```
#show grid: box.with(stroke: green)
#let r = rect(fill: blue, width: 10pt,
height: 10pt)
#grid(columns: 3, gutter: 5pt, r, r, r, r,
r)
```

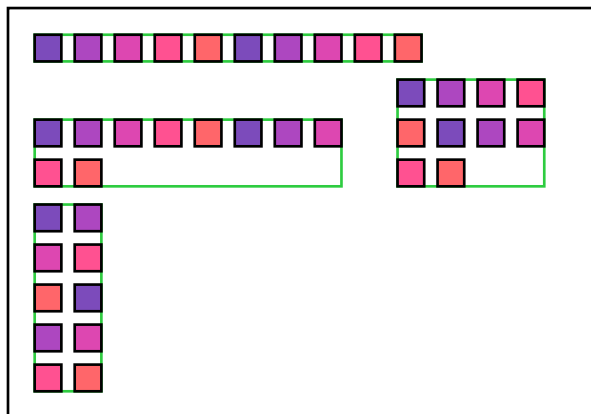


与 stack 相比，grid 要复杂得多。

其一，grid 的前 $N - 1$ 行可容纳的元素是有限的，而最后一行与 stack 一样容纳其余所有元素。你可以通过 columns 参数控制这一特性。

columns 要求你给定一个数组，该数组表示一行中每列的宽度。

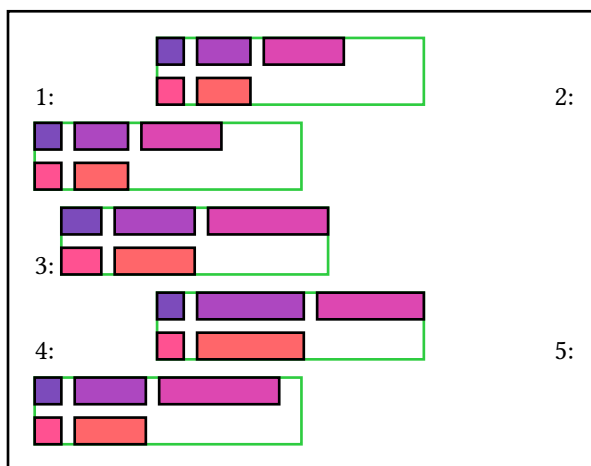
```
#let r(n, width: 10pt, height: 10pt) =
range(n).map(i => rect(fill:
color.map.rainbow.at(calc.rem(i * 16,
80)), width: width, height: height))
#grid(columns: (auto, ) * 10, gutter:
5pt, ..r(10)) #h(1em)
#grid(columns: (auto, ) * 8, gutter:
5pt, ..r(10)) #h(1em)
#grid(columns: (auto, ) * 4, gutter:
5pt, ..r(10)) #h(1em)
#grid(columns: (auto, ) * 2, gutter:
5pt, ..r(10))
```



上例中，当 columns 数组的长度为 N 时，表示每行允许容纳 N 个元素，每列的最大宽度为 auto。长度设定为 auto，则表示每列宽度始终和该行该列内部元素宽度相等。

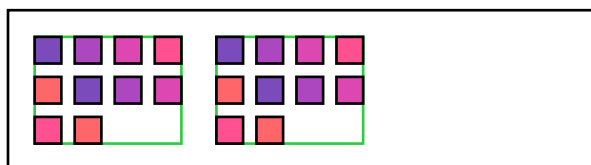
每列宽度不仅可以为 auto，而且可以指定为具体长度：

```
#show grid: box.with(stroke: green, width:
100pt)
#let grid = grid.with(gutter: 5pt, ..r(5,
width: auto))
1: #grid(columns: (10pt, 20pt, 30pt))
#h(1em)
2: #grid(columns: (10%, 20%, 30%)) #h(1em)
3: #grid(columns: (1fr, 2fr, 3fr)) #h(1em)
\
4: #grid(columns: (10pt, 1fr, 1fr))
#h(1em)
5: #grid(columns: (10pt, 1fr, 2fr))
#h(1em)
```



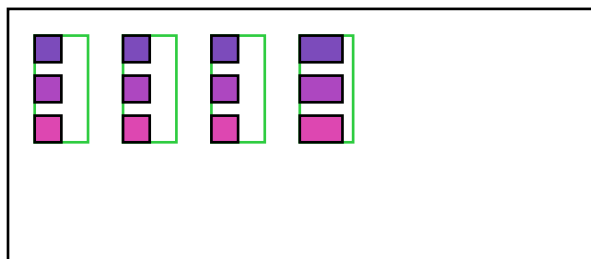
Typst 允许你使用更简便的参数表示。特别地，如果 columns 数组内容恰为 N 个 auto，那么可以直接传入一个数字。以下两种写法是等价的：

```
#grid(columns: 4, gutter: 5pt, ..r(10))
#h(1em)
#grid(columns: (auto, ) * 4, gutter:
5pt, ..r(10)) #h(1em)
```



特别地，如果期望 grid 只有一列，columns 参数允许不传入数组而是单列的长度：

```
#show grid: box.with(stroke: green, width:
20pt)
#let grid = grid.with(gutter: 5pt, ..r(3,
width: auto))
#grid(columns: (10pt, )) #h(1em)
#grid(columns: 10pt) #h(1em)
#grid(columns: 50%) #h(1em)
#grid(columns: 80%) #h(1em)
```

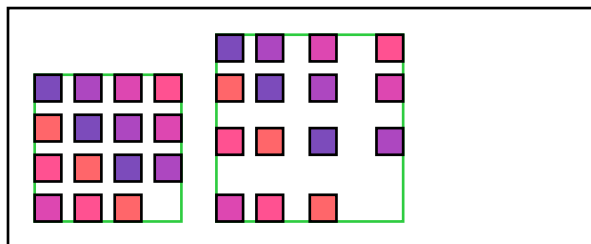


特别地，columns 的默认值可以理解为 auto。

其二，grid 不再允许混杂内容与长度来控制某行或某列元素之间的间隔。取而代之的是三个更为复杂的参数：gutter、row-gutter 和 column-gutter。这是因为“混杂的 gutter”在二维布局中有歧义。

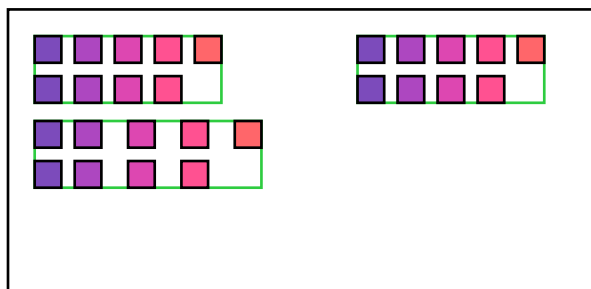
gutter 参数事实上是 spacing 参数的加强版。它允许通过传入一个长度数组为每行每列依次设置间隔：

```
#let grid = grid.with(columns: (auto, ) *
4, ..r(15))
#grid(gutter: 5pt) #h(1em)
#grid(gutter: (5pt, 10pt, 15pt)) #h(1em)
```



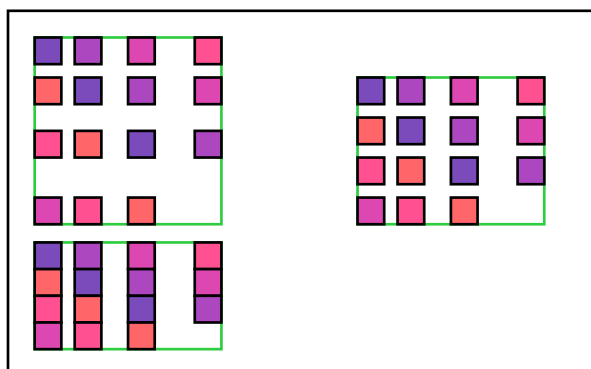
特别地，如果 gutter 参数数组比 rows 参数数组或者 columns 参数数组更短，其余行或列的的间隔以 gutter 参数数组的最后一个长度为准：

```
#let grid = grid.with(columns: (auto, ) *
5, ..r(9))
// 等价于`5pt, 5pt, 5pt, ..`, 一直 5pt 下去
#grid(gutter: 5pt) #h(1em)
// 等价于`5pt, 5pt, 5pt, ..`, 一直 5pt 下去
#grid(gutter: (5pt,)) #h(1em)
// 等价于`5pt, 10pt, 10pt, ..`, 一直 10pt 下
去
#grid(gutter: (5pt, 10pt)) #h(1em)
```



你可以指定 column-gutter 而非 gutter，从而单独为**每列**设置间隔。column-gutter 参数的优先级比 gutter 参数要高。

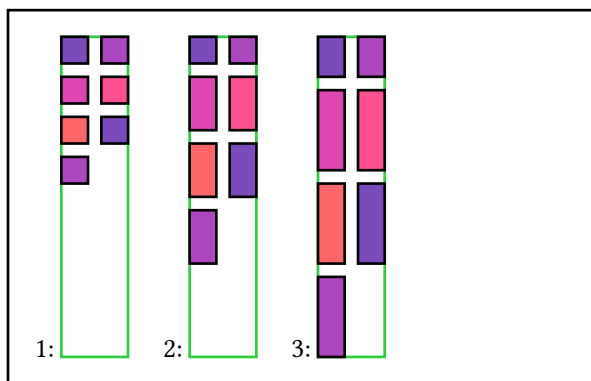
```
#let grid = grid.with(columns: (auto, ) *
4, ..r(15))
#grid(gutter: (5pt, 10pt, 15pt)) #h(1em)
#grid(gutter: 5pt, column-gutter: (5pt,
10pt, 15pt)) #h(1em)
#grid(column-gutter: (5pt, 10pt, 15pt))
#h(1em)
```



与 column-gutter 同理，你也可以指定 row-gutter 而非 gutter，从而单独为**每行**设置间隔。

与 columns 同理，你可以指定 rows 从而为每行设置高度。但是请注意，Typst 对 rows 参数数组的解释方式与 columns 不同，而与 gutter 相同：

```
#show grid: box.with(stroke: green,
height: 120pt)
#let grid = grid.with(columns: 2, gutter:
5pt, ..r(7, height: auto))
// 等价于`10pt, 10pt, 10pt, ..`, 一直 10pt 下
去
1: #grid(rows: (10pt, )) #h(1em)
// 等价于`10pt, 20pt, 20pt, ..`, 一直 20pt 下
去
2: #grid(rows: (10pt, 20pt)) #h(1em)
// 等价于`1fr, 2fr, 2fr, ..`, 一直 2fr 下
去
3: #grid(rows: (1fr, 2fr)) #h(1em)
```



「间距」

此类函数的概念引自 LaTeX，又称胶水函数。一共有两个方向的间距，分别是水平间距与竖直间距。

水平间距（h，horizon）不包含内容，而仅仅在布局中占据一定宽度：

```
两句话之间间隔`0pt`#h(0pt)两句话之间间隔`0pt` \
两句话之间间隔`5pt`#h(5pt)两句话之间间隔`5pt` \
两句话之间间隔`1em`#h(1em)两句话之间间隔`1em` \
间隔`1fr`#h(1fr)间隔`1fr` \
间隔`1fr`#h(1fr)间隔`...`#h(2fr)间隔`2fr` \
#h(1fr)夹心的`1fr`...`#h(1fr) \
```

两句话之间间隔 0pt 两句话之间间隔 0pt
 两句话之间间隔 5pt 两句话之间间隔 5pt
 两句话之间间隔 1em 两句话之间间隔 1em
 间隔 1fr 间隔 1fr
 间隔 1fr 间隔... 间隔 2fr
 夹心的 1fr...

同理竖直间距（v, vertical）不包含内容，而仅仅在布局中占据一定高度。插入竖直间距将会导致强制换行。

```
1 2 \
1 \ 2 \
1 #v(-1em) 2
```

1 2
 1
 2
 1
 2

「空间变换」

一共有三种类型的空间变换，分别是：

- move：移动元素。
- scale：拉伸元素。
- rotate：旋转元素。

它们都不会影响布局，且都会导致强制换行。

move 相对于当前位置移动一段距离。

```
#rect(inset: 0pt, move(
  dx: 6pt, dy: 6pt,
  rect(
    inset: 8pt,
    fill: white,
    stroke: black,
    [Abra cadabra]
  )
))
```

Abra cadabra

你可以使用 box 将其改为行内元素行为：

```
#let TeX(width) = {
  set text(font: "New Computer Modern",
  weight: "regular")
  box(width: width, {
    [T]
    box(stroke: red, move(dx: -0.2em, dy:
    0.22em)[E])
    box(stroke: green, move(dx: -0.4em)
    [X])
  })
}
#TeX(2.2em) is a digital typographical
systems. \
#TeX(2.1em) is a digital typographical
systems.
```

TeX is a digital typographical systems.
 TE
 X is a digital typographical systems.

scale 可以拉伸元素，且 scale 会导致强制换行：

```
#scale(x: 110%)[This is mirrored.]
#box(scale(x: 100%)[This is scaled.]) 1 \
#box(scale(x: 115%)[This is scaled.]) 1
```

This is mirrored.
This is scaled. 1
This is scaled.1

特别地，你可以指定负长度以反转元素：

```
#scale(x: -100%)[This is mirrored.]
```

.bəʊnɪm zɪ zɪdʌ

rotate 可以旋转元素：

```
#rotate(5deg)[This is rotated.]
```

This is rotated.

让「空间变换」函数影响布局

虽然内置的 move 等函数不会影响布局，但是你可以通过 box 模拟子元素对父元素的布局影响：

```
#let ac(a) = calc.abs(calc.cos(a))
#let ax(a) = calc.abs(calc.sin(a))
#let rot(x,y,a) = .5*(x*(ac(a)-1)+y*ax(a))
#let rotx(body, angle) = context {
  let sz = measure(body)
  box(stroke: green, inset: (
    x: rot(sz.width, sz.height, angle),
    y: rot(sz.height, sz.width, angle)),
    rotate(body, angle))
}
#lorem(7) This is a
#rotx([Vertical], -90deg) word, and
#rotx([this], -7deg) is #rotx([skew.], 7deg)
It also works with #rotx($e^(-x)$, -95deg).
#lorem(20)
```

Lorem ipsum dolor sit amet, consectetur
adipiscing. This is a Vertical word, and this is skew. It
also works with skew. Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magnam
aliquam quaerat.

例子使用了 measure 函数评估子元素的大小，并强行改变子元素的边界框（bounding box）。

「绝对定位布局」

你可以使用 place 完成「绝对定位布局」（absolute positioning）。place 相对于父元素移动一段距离，其接受一个 alignment 参数，将元素放置在相对于父元素 alignment 的绝对位置。例如下例相对于父元素的右上角放置了一个矩形框：

```
#box(height: 30pt, width: 100%)[
  Hello, world!
  #place(
    top + right,
    square(
      width: 20pt,
      stroke: 2pt + blue
    ),
  ),
]
```

Hello, world!



place 会使得子元素脱离父元素布局。对比两例：

```

#let TeX(width: 2.2em) = {
  set text(font: "New Computer Modern",
weight: "regular")
  box(width: width, {
    [T]
    box(stroke: red, move(dx: -0.2em, dy:
0.22em)[E])
    box(stroke: green, move(dx: -0.4em)
[X])
  })
}
#TeX() is a digital typographical systems.
\
#let TeX = {
  set text(font: "New Computer Modern",
weight: "regular")
  box(width: 1.7em, {
    [T]
    place(top, dx: 0.56em, dy: 0.22em,
box(stroke: red, [E]))
    place(top, dx: 1.1em, box(stroke:
green, [X]))
  })
}
#TeX is a digital typographical systems.

```

TeX is a digital typographical systems.
TeX is a digital typographical systems.

你可以使用 `place(float: true)` 以创建相对于父元素的浮动布局。(todo)

杂项

1. hide
2. repeat

色彩与图表

颜色类型

Typst 只有一种颜色类型，其由两部分组成。

```
#color.rgb(87, 127, 230)
```

| +-- 色坐标
+-- 色彩空间对应的构造函数

这里有个图注解

「色彩空间」(color space) 是人们主观确定的色彩模型。Typst 为不同的色彩空间提供了专门的构造函数。

「色坐标」(chromaticity coordinate) 是客观颜色在「色彩空间」中的坐标。给定一个色彩空间，如果某种颜色在空间内，那么颜色能分解到不同坐标分量上，并确定每个坐标分量上的数值。反之，选择一个构造函数，并提供坐标分量上的数值，就能构造出这个颜色。

todo chromaticity coordinate 这个名词是对的吗，每种色彩空间中的坐标都是这个名字吗？

习惯上，颜色的坐标分量又称为颜色的「通道」。从物理角度，Typst 使用 f32 存储颜色每通道的值，这允许你对颜色进行较复杂的计算，且计算结果仍然保证较好的误差。

色彩空间

RGB 是我们最使用的色彩空间，对应 Typst 的 `color.rgb` 函数或 `rgb` 函数：

```
#box(square(fill: color.rgb("#b1f2eb")))  
#box(square(fill: rgb(87, 127, 230)))  
#box(square(fill: rgb(25%, 13%, 65%)))
```



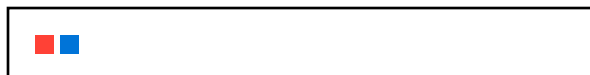
除此之外，还支持 HSL (`hsl`)、CMYK (`cmyk`)、Luma (`luma`)、Oklab (`oklab`)、Oklch (`oklch`)、Linear RGB (`color.linear-rgb`)、HSV (`color.hsv`) 等色彩空间。感兴趣的可以自行搜索并使用。

i 尽管你可以随意使用这些构造器，但是可能会导致 PDF 阅读器或浏览器的兼容性问题。它们可能不支持某些色彩空间（或色彩模式）。

预定义颜色

除此之外，你还可以使用一些预定义的颜色，详见《[Typst Docs: Predefined colors](#)》。

```
#box(square(fill: red, size: 7pt))  
#box(square(fill: blue, size: 7pt))
```



颜色计算

Typst 较 LaTeX 的一个有趣的特色是内置了很多方法对颜色进行计算。这允许你基于某个颜色主题 (Theme) 配置更丰富的颜色方案。这里给出几个常用的函数：

- `lighten/darken`: 增减颜色的亮度，参数为绝对的百分比。
- `saturate/desaturate`: 增减颜色的饱和度，参数为绝对的百分比。
- `mix`: 参数为两个待混合的颜色。

```
#show square: box
#set square(size: 15pt)
#square(fill: red.lighten(20%))
#square(fill: red.darken(20%)) \
#square(fill: red.saturate(20%))
#square(fill: red.desaturate(20%)) \
#square(fill: blue.mix(green))
```



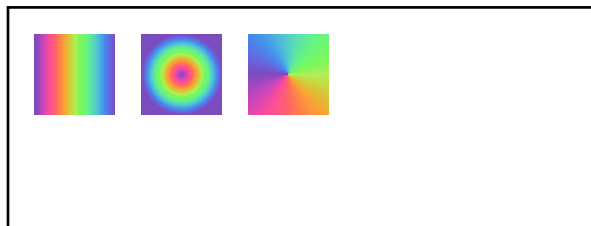
还有一些其他不太常见的颜色计算，详见《[Typst Docs: Color operations](#)》。

渐变色

你可以以某种方式对 Typst 中的元素进行渐变填充。这有时候对科学作图很有帮助。

有三种渐变色构造函数，可以分别构造出线性渐变（Linear Gradient），径向渐变（Radial Gradient），锥形渐变（Conic Gradient）。他们都接受一组颜色，对元素进行颜色填充。

```
#let sqr(f) = square(fill: f(
  ..color.map.rainbow))
#stack(
  dir: ltr, spacing: 10pt,
  sqr(gradient.linear),
  sqr(gradient.radial),
  sqr(gradient.conic ))
```



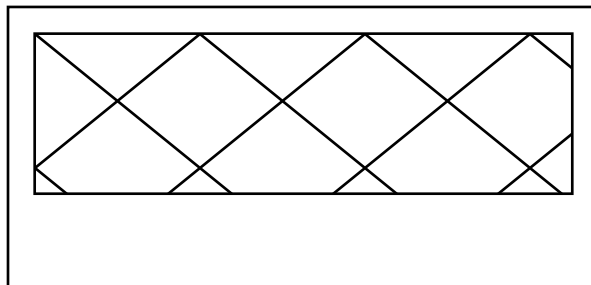
从字面意思理解 `color.map.rainbow` 是 Typst 为你预定义的一个颜色数组，按顺序给出了彩虹渐变的颜色。还有一些其他预定义的颜色数组，详见《[Typst Docs: Predefined color maps](#)》。

填充模式

Typst 不仅支持颜色填充，还支持按照固定的模式将其他图形对元素进行填充。例如下面的 `pat` 定义了一个长为 61.8pt，宽为 50pt 的图形。将其填充进一个矩形中，填充图形从左至右，从上至下布满矩形内容中。

```
#let pat = pattern(size: (61.8pt, 50pt))[
  #place(line(start: (0%, 0%), end: (100%,
    100%)))
  #place(line(start: (0%, 100%), end:
    (100%, 0%)))
]

#rect(fill: pat, width: 100%, height:
  60pt, stroke: 1pt)
```

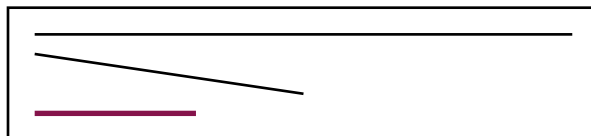


线条

我们学习的第一个图形元素是直线。

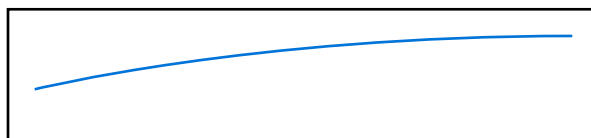
`line` 函数创建一个直线元素。这个函数接受一系列参数，包括线条的长度、起始点、终止点、颜色、粗细等。

```
#line(length: 100%)
#line(end: (50%, 50%))
#line(
  length: 30%, stroke: 2pt + maroon)
```



除了直线，Typst 还支持贝塞尔曲线。贝塞尔曲线是一种光滑的曲线，由一系列点和控制点组成。

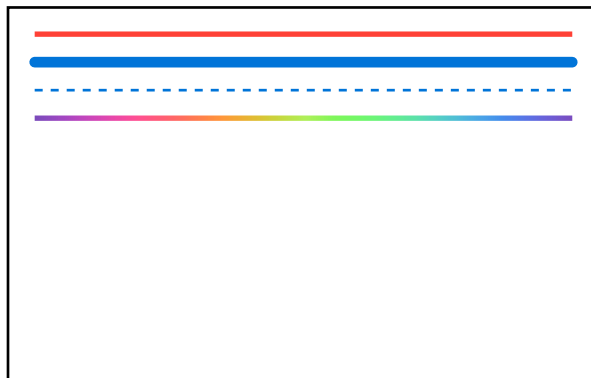
```
#path(
  stroke: blue,
  ((100%, 0pt), (60%, 0pt), (0pt, 20pt),
  )
```



线条样式

与各类图形紧密联系的类型是「线条样式」(stroke)。线条样式可以是一个简单的「字典」, 包含样式数据:

```
#set line(length: 100%)
#let rainbow = gradient.linear(
  ..color.map.rainbow)
#stack(
  spacing: 1em,
  line(stroke: 2pt + red),
  line(stroke: (paint: blue, thickness:
4pt, cap: "round")),
  line(stroke: (paint: blue, thickness:
1pt, dash: "dashed")),
  line(stroke: 2pt + rainbow),
)
```



你也可以使用 stroke 函数来设置线条样式:

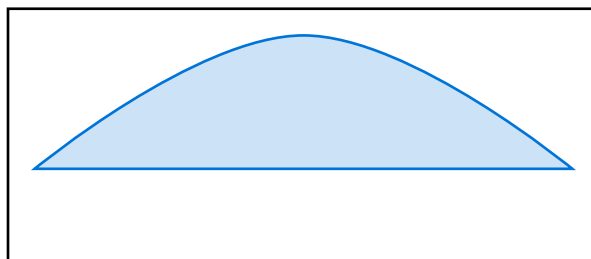
```
#set line(length: 100%)
#let blue-line-style = stroke(
  paint: blue, thickness: 2pt)
#line(stroke: blue-line-style)
```



填充样式

填充样式 (fill) 是另一个重要的图形属性。如果一个路径是闭合的, 那么它可以被填充。

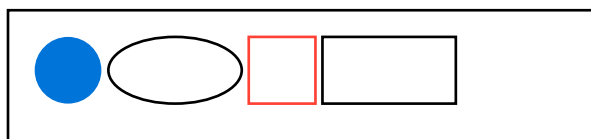
```
#path(
  fill: blue.lighten(80%),
  stroke: blue,
  closed: true,
  (0pt, 50pt),
  (100%, 50pt),
  ((50%, 0pt), (40pt, 0pt)),
)
```



闭合图形

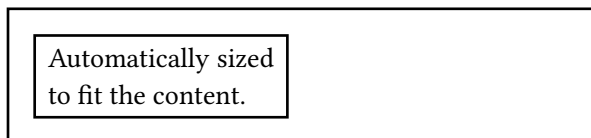
Typst 为你预定义了一些基于贝塞尔曲线的闭合图形元素。下例子种, #circle、#ellipse、#square、#rect、#polygon 分别展示了圆、椭圆、正方形、矩形、多边形的构造方法。

```
#box(circle(radius: 12.5pt, fill: blue))
#box(ellipse(width: 50pt, height: 25pt))
#box(square(size: 25pt, stroke: red))
#box(rect(width: 50pt, height: 25pt))
```



值得注意的是, 这些图形元素都允许自适应一个内嵌的内容。例如矩形作为最常用的边框图形:

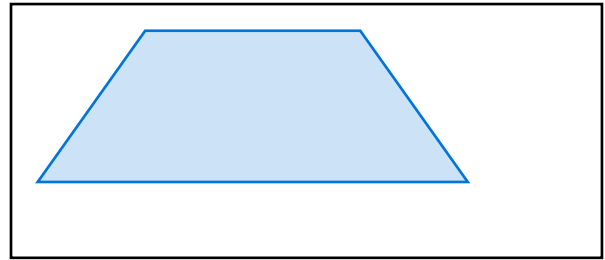
```
#rect[
  Automatically sized \
  to fit the content.
]
```



多边形

多边形是唯一一个使用直线组合而成的闭合图形。你可以使用 polygon 函数构造一个多边形。

```
#polygon(  
  fill: blue.lighten(80%),  
  stroke: blue,  
    (20%, 0pt),  
    (60%, 0pt),  
    (80%, 2cm),  
    (0%, 2cm),  
)
```



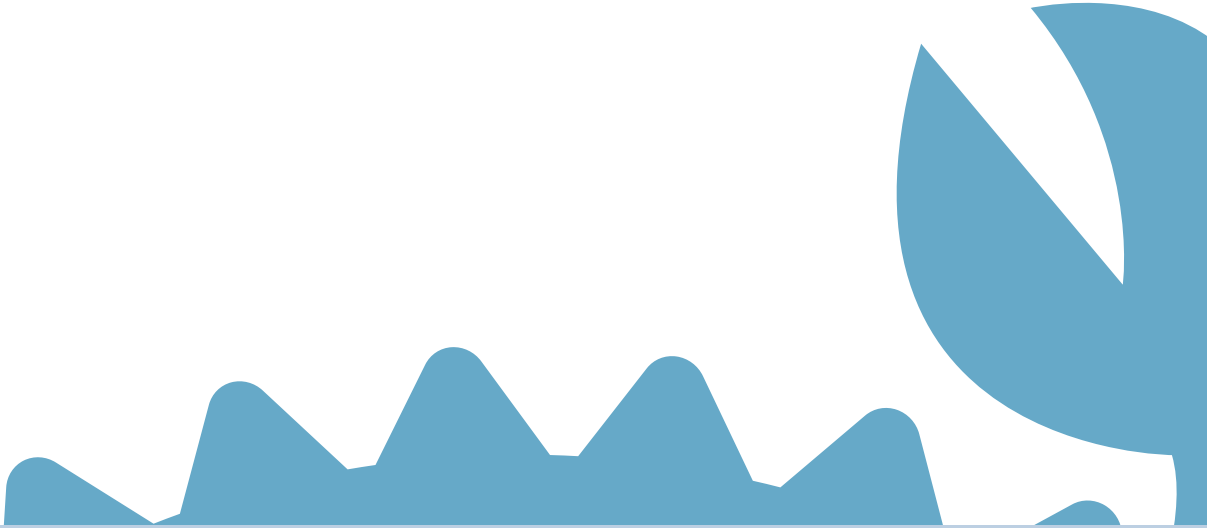
`polygon` 不允许内嵌内容。

外部图形库


很多时候我们只需要在文档中插入分割线（直线）和文本框（带文本的矩形）。但是，若有需求，一些外部图形库可以帮助你绘制更复杂的图形：

1. 树形图： [typst-syntree](#)
2. 拓扑图： [typst-fletcher](#)
3. Canvas 通用图形库： [cetz](#)

这些库也是很好的参考资料，你可以通过查看源代码来学习如何绘制复杂的图形。



Part.4 基础教程 — 脚本 II



文件与模块

正如我们在《初识脚本模式》中所说的，Typst 提供了脚本语言方便排版。但事实上，写作时若能少写甚至不写脚本，这才算真正的便捷。我们总希望 Typst 能够允许我们以一种优雅的方式复制粘贴引入已有代码。理想情况下，只需两行代码便可引入前辈写好的模板：

```
#import "@preview/senpai-no-awesome-template.typ:0.x.x": *
#show: template.with(..)
```

模块便是为复制而生。

Typst 的模块（module）机制非常简单：每个文件都是一个独立的模块。它允许你：

1. 将一份文档分作多个文件编写。
2. 使用本机器或网上的模板或库。

由于 Typst 的模块机制过于简单，本节主要伴随讲解一些日常使用模块机制时所遇到的问题，而不会涉及很多需要理解的地方。

根目录

根目录是一个重要的编译参数。Typst 早在 v0.1.0 时就禁止访问**根目录以外的内容**，以保证安全执行代码。最坏情况下，**其他人编写的 Typst 脚本（来自互联网）**可以访问根目录下所有的文件。

你所使用的工具可能会为你自动配置**根目录**。

- 1. 当你使用 `typst-cli` 程序时，根目录默认为文档所在目录。你也可以通过 `--root` 命令行参数手动指定一个根目录。例如你可以指定当前工作目录的父文件夹作为编译程序的根目录：

```
typst c --root ..
```

- 2. 当你使用 VSCode 的 `tinymist` 或 `typst-preview` 程序时，为了方便多文件的编写，根目录默认为你所打开的文件夹。如果你打开一个 VSCode 工作区，则根目录相对于离你所编辑文件最近的工作目录。

一个常见的错误做法是，为了共享本地代码或模板，在使用 `typst-cli` 的时候将根目录指定为系统的根目录。以下使用方法将有可能导致个人隐私泄露：

```
typst c --root "C:\\\\" # Windows
typst c --root / # Linux 或 macOS
```

并访问你系统中的某处文件。

```
#import "/Users/me/templates/book.typ": book-template
```

本节将介绍另外一种创建本地「库」（package）的方式以在 Typst 项目之间安全地共享代码和资源。

绝对路径与相对路径

我们已经讲解过 `read` 函数和 `include` 语法，其中都有路径的概念。

如果路径以字符 `/` 开头，则其为「绝对路径」。绝对路径相对于「根目录」解析。若设置了根目录为 `/0w0/`，则路径 `/a/b/c` 对应路径 `/0w0/a/b/c`。

```
#include "/src/intermediate/chapter1.typ"
```

我是 `/0w0/src/intermediate/chapter1.typ` 文件！

否则，路径不以字符 `/` 开头，则其「相对路径」。相对路径相对于当前文件的父文件夹解析。若我们正在编辑 `/0w0/src/intermediate/main.typ` 文件，则路径 `d/e/f` 对应路径 `/0w0/src/intermediate/d/e/f`。

```
#include "chapter2.typ"
```

我是 /0w0/src/intermediate/chapter2.typ 文件!

「import」语法与「模块」

Typst 中的模块概念相当简单。你只需记住：每个文件都是一个模块。

假设有一个文件位于 packages/m1.typ:

```
XXX
#let add(x, y) = x + y
YYY
#let sub(x, y) = x - y
```

此时文件名就是所引入模块的名称，那么 packages/m1.typ 就对应 m1 模块。

你可以简单通过绝对路径或相对路径引入一个文件模块：

```
#import "packages/m1.typ"
#repr(m1)
```

<module m1>

你可以通过这个「模块」对象访问到文件顶层作用域的变量或函数。例如在 m1 文件中，你可以访问到其顶层作用域中的 add 或 sub 函数。

```
#import "packages/m1.typ"
#m1.add \
#m1.sub
```

add
sub

你可以在引入模块名的同时将其更名为你想要的名称：

```
#{
  import "packages/m1.typ" as m2
  repr(m2)
} \
// 等价于
#{
  import "packages/m1.typ"
  let m2 = m1
  repr(m2)
}
```

<module m1>
<module m1>

你可以在冒号后添加一个星号表示直接引入模块中所有的函数：

```
// 引入所有函数
#import "packages/m1.typ": *
#type(add), #type(sub)
```

function, function

你也可以在冒号后追加一个逗号分隔的名称列表，部分引入来自其他文件的变量或函数：

```
// 仅仅引入`sub`函数
#import "packages/m1.typ": sub
#type(sub) \
// 引入`add`和`sub`函数
#import "packages/m1.typ": add, sub
#type(add), #type(sub) \
```

function
function, function

注意，本地的变量声明与 import 进来的变量声明都会覆盖 Typst 内置的变量声明。例如 Typst 内置的 sub 函数实际上为取下标函数。你可以在引入外部变量的同时更名以避免可能的名称冲突：

```
1#sub[2]
#import "packages/m1.typ": sub as subtract
#repr(subtract(10, 1))#sub[3]
```

1₂ 9₃

「include」语法与「import」语法的关系

在 Typst 内部，当解析完一个文件时，文件将被分为顶层作用域中的「内容」和「变量声明」，共同组成一个文件模块。include 取其「内容」，import 则取其「变量声明」。

仍然以前文中的 packages/m1.typ 为例：

其「内容」是除了「变量声明」以外的文件内容，连接起来为：

```
XXX
// #let add(x, y) = x + y
YYY
// #let sub = ..
```

其导出所有在文件顶层作用域中的「变量声明」：

```
// XXX
#let add(x, y) = x + y
// YYY
#let sub(x, y) = x - y
```

因此 include 和 import 分别得到以下结果。

使用 include 得到：

```
#repr(include "packages/m1.typ")
```

```
sequence([ ], [XXX], [ ], [ ], [YYY], [ ], [ ])
```

使用 import 得到：

```
#import "packages/m1.typ"
#repr(m1.add), #repr(m1.sub)
```

```
add, sub
```

你可以同时使用 include 和 import 获得同一个文件的内容和变量声明。

控制变量导出的三种方式

有的时候你不希望将一些变量暴露出去，这个时候你可以让这些变量的名称以「下划线」() 开头：

```
#let _factor = 1;
#let add2(x, y) = _factor * (x + y)
```

这样 import 的时候就不会轻易访问到这些变量。

还有一种方法是在非顶层作用域中构造闭包，这样 import 中就不会包含 factor，因为 factor 不在顶层：

```
#let add2 = {
  let factor = 1;
  (x, y) => factor * (x + y)
}
```

值得注意的是，import 进来的变量也可以被重新导出，只要他们也在顶层作用域。这允许你以更优雅的第三种方式为使用者屏蔽内部变量，例如你可以在子文件夹中完成实现并重新导出：

```
// 仅从 packages/m1/add.typ 文件中重新导出`add`函数。
#import "m1/add.typ": add
// 重新导出 packages/m1/sub.typ 文件中所有的函数
#import "m1/sub.typ": *
```


使用外部库

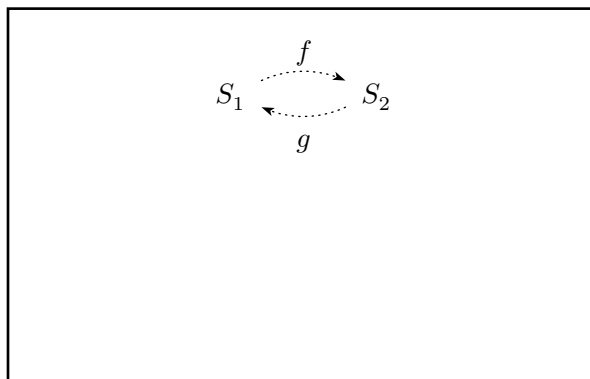
除此之外，可以从特殊的路径从网络导入外部「模块」，即外部「库」(packages)。Typst 的外部库机制可能不是世界上最精简的，但是我所见过当中最精简的。它可能更稍显简陋，但已经有上百个外部库通过官方渠道发布，并足以满足我们日常生活的使用。

你只需要 `import` 特定的路径就能访问其内部的变量声明。例如，导入一个用于绘画自动机的外部库：

```
#import "@preview/fletcher:0.4.0" as
fletcher: node, edge

#align(center, fletcher.diagram(
  node((0, 0), $S_1$),
  node((1, 0), $S_2$),

  edge((1, 0), (0, 0), $g$, "..>", bend:
25deg),
  edge((0, 0), (1, 0), $f$, "..>", bend:
25deg),
))
```



其中，以下一行代码完成了导入外部库的所有工作。我们注意到其完全为 `import` 语法，唯一陌生的是其中的路径格式：

```
#import "@preview/fletcher:0.4.0" as fletcher: node, edge
```

解读路径 `@preview/fletcher:0.4.0` 的格式，它由三部分组成。

「命名空间」，`@preview`

以 `@` 字符开头。目前仅允许使用两个命名空间：

- `@preview`：Typst 仅开放 beta 版本的包管理机制。所有 beta 版本的都在 `preview` 命名空间下。
- `@local`：Typst 建议的本地库命名空间。

「库名」，`fletcher`

必须符合变量命名规范：

1. 以 ASCII 英文字符（a-z 或 A-Z）开头；
2. 跟随任意多个 ASCII 英文或数字字符（a-z 或 A-Z 或 0-9）或下划线（`_`）或中划线（`-`）。

例如 `0v0`、`0_0`、`0-0` 都是合法的库名。但是 `0v0` 不是合法库名，因为其以数字零开头。

「版本号」，`0.4.0`

必须符合 `SemVer` 规范。

如果你不能很好地阅读和理解 `SemVer` 规范，仅记住合法的版本号由三个递增的数字组成，并用「点号」(`.`) 分隔；版本号之间可以相互比较，且比较版本时按顺序比较数字。例如 `0.0.0`、`0.10.0`、`1.5.11`、`1.24.1` 是合法且递增的版本号。

1. `1.0.11` 比 `0.10.0` 大，因为 `1` 大于 `0`；
2. `1.24.1` 比 `1.5.11` 大，因为 `24` 大于 `5`；
3. `1.24.0` 与 `1.24.0` 相等。

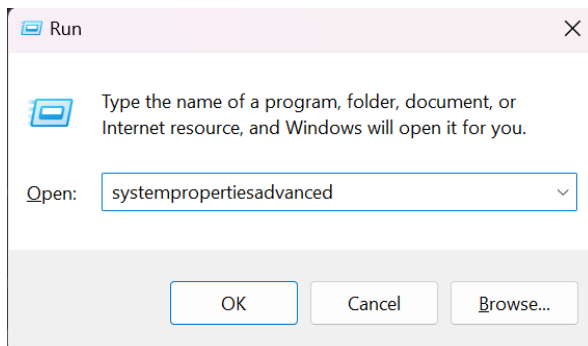
下载外部库

一般来说，使用外部库与导入本地模块一样简单。当你尝试导入一个外部库时，Typst 会立即启动下载线程为你从网络下载外部库代码：

```
#import "@preview/fletcher:0.4.0" as fletcher: node, edge
```

一般情况下，从网络下载外部库的时间不会超过十秒钟，并且不需要任何额外配置。但由于不可描述的原因，你有可能需要为下载线程配置代理。当你希望通过网络代理下载外部库时，请检查全局环境变量 `HTTP_PROXY` 和 `HTTPS_PROXY` 是否成功设置。

1. 如果你的网络代理软件包含环境变量设置，请**优先**在你的网络代理环境中配置。
2. 否则如果你是 Linux 用户，请检查启动 `typst-cli` 或 `vscode` 的 `shell` 中，使用 `echo $HTTP_PROXY` 检查是否包含该环境变量。
3. 否则如果你是 Windows 用户，请使用 `Win + R` 快捷键呼出运行窗口，执行 `systempropertiesadvanced`，调出「环境变量」窗口，并检查是否包含该环境变量。



库文件夹

上一小节中的「命名空间」对应本地的一个或多个文件夹。

首先，Typst 会感知你系统上的“数据文件夹”和“缓存文件夹”并在其中存储库代码。

Typst 将会检查 `{data-dir}/typst/packages` 中是否包含相应的库，随后会在 `{cache-dir}/typst/packages` 中检查和缓存从网络下载的库代码。其中数据文件夹 (`{data-dir}`) 和缓存文件夹 (`{cache-dir}`) 在不同的操作系统上：

操作系统	数据文件夹	缓存文件夹
Linux	<code>\$XDG_DATA_HOME</code> 或 <code>~/.local/share</code>	<code>\$XDG_CACHE_HOME</code> 或 <code>~/.cache</code>
macOS	<code>~/Library/Application Support</code>	<code>~/Library/Caches</code>
Windows	<code>%APPDATA%</code>	<code>%LOCALAPPDATA%</code>

例如，当引入外部库 `@preview/fletcher:0.4.0` 时，Typst 会严格**按顺序**检查并解析路径：数据文件夹中的 `{data-dir}/typst/packages/preview/fletcher/0.4.0` 和缓存文件夹中的 `{cache-dir}/typst/packages/preview/fletcher/0.4.0`。Typst 会将库路径映射到你数据文件夹或缓存文件夹，优先按顺序使用已经存在的库代码，并按需从网络下载外部库。

这意味着你可以拥有以下几条特性。

- 若你已经下载了网络库到本地，再次访问将不会再产生网络请求和检查远程库代码的状态。
- 你可以将本地库存储到数据文件夹。特别地，你可以在数据文件夹中的 `preview` 文件夹中存储库，以覆盖缓存文件夹中已经下载的库。这有利于你调试或临时使用即将发布的外部库。
- 你可以在本地随意创建新的命名空间，对于 `@my-namespace/my-package:version`，Typst 将检查并使用你位于 `{data-dir}/typst/packages/my-namespace/my-package/version` 的库代码。尽管 Typst 对命名空间没有限制，但建议你使用 `@local` 作为所有本地库的命名空间。

构建和注册本地库

本小节教你如何构建和注册本地库。

元数据文件

在库的**根目录**下必须有一个名称为 `typst.toml` 的元数据文件，这是对代码库的描述。内容示例如下：

```
[package]
name = "example"
version = "0.1.0"
entrypoint = "lib.typ"
```

最低限度你仅需要填入以上三个字段，分别对应其名字、版本号和入口文件的路径。

入口文件

一般来说，该文件对应为库的根目录下的 `lib.typ` 文件。你可以使用本节提及的《变量导出的三种方式》编写该文件。当导入该库时，其中的「变量声明」将提供给文档使用。

示例库的文件组织

假设你希望构建一个 `@local/example:0.1.0` 的外部库供本地使用。你应该将 `example` 库的文件夹**拷贝或链接**到 `{data-dir}/typst/packages/local/example/0.1.0` 路径。

创建 `{data-dir}/typst/packages/local/example/0.1.0/typst.toml` 文件，并包含前文所述内容。

创建 `{data-dir}/typst/packages/local/example/0.1.0/lib.typ` 文件，并包含以下内容：

```
#import "add.typ": add
#import "sub.typ": sub
```

这样你就可以在本地的任意文档中使用库中的 `add` 或 `sub` 函数了：

```
#import "@local/example:0.1.0" as example: add
```

发布库到官方源

略

再谈「根目录」

出于安全考虑，每个库都**默认**只能访问其专属的「根目录」，即 `typst.toml` 文件所在的目录。这意味着，库中的绝对路径或相对路径依其专属的「根目录」解析。请回忆《绝对路径与相对路径》小节内容。例如在文件夹 `@local/example:0.1.0` 的内部，设使内部文件 `{data-dir}/{example-lib}/src/add-simd.typ` 中包含这样的代码。

```
#read("/typst.toml")
#read("../typst.toml")
```

则「根目录」被解析为 `{data-dir}/{example-lib}`，绝对路径 `/typst.toml` 被解析为 `{data-dir}/{example-lib}/typst.toml`，相对路径 `../typst.toml` 被解析为 `{data-dir}/{example-lib}/src/../../typst.toml`

函数与闭包中的路径解析

这里有一个微妙的问题。在函数或闭包中请求解析一个绝对路径或相对路径，Typst 会如何做？答案是，任何路径解析都依附于路径解析代码所在文件。这句话有些晦涩，但例子却容易懂。这里举两个例子。

假设函数在文档相对于根目录的 `{example-lib}/src/m1.typ` 中有这样一个函数：

```
#let parse-code(path) = parse-text(read(path))
```

那么无论我们在哪个文件引入了 `parse-code` 函数，其 `read(path)` 的路径解析都是固定的，其绝对路径相对于文档根目录，其相对路径相对于 `{example-lib}/src/m1.typ` 所在目录。例如调用 `parse-code("../def.typ")` 时，其始终读取 `{example-lib}/src/../../def.typ` 文件。

我们上一小节说过，每个库都**默认**只能访问其专属的「根目录」。这种行为在有的时候不是你期望的，但你可以同样通过传递一个来自文档的闭包绕过该限制。例如，`parse-code` 改写为：

```
#let parse-code(path, read-file: read) = parse-text(read-file(path))
```

并且在调用时传入一个读取文件的「回调函数」

```
#parse-code("../def.typ", read-file: (p) => read(p))
```

多文件文档

尽管本书提倡你尽可能将所有文档内容放在单个文件中，本书给出构建一个多文件文档的合理方案。

- 工作区中包含多个主文件
- 每个主文件可以 include 多个子文件

```
typ/packages
└── util.typ
typ/templates
└── book-template.typ
documents/
└── my-book/
    ├── main.typ
    ├── mod.typ
    ├── part1/
    │   ├── mod.typ
    │   └── chap1.typ
    └── part2/
        ├── mod.typ
        └── chapN.typ
```

工作区内的模板与库

使用绝对路径方便引入工作区内的模板与库：

```
#import "/typ/templates/ebook.typ": project as ebook
```

主文件和主库文件

main.typ 文件中仅仅包含模板配置与 include 多个子文件。

```
// 在 mod.typ 文件中：
// #import "/typ/templates/ebook.typ": project as ebook
#import "mod.typ": *

#show: ebook.with(title: "My Book")

#include "part1/chap1.typ"
#include "part2/chap2.typ"
#include "part2/chap3.typ"
```

对于每个子文件，你可以都 import 一个主库文件 mod.typ，以减少冗余。例如在 documents/my-book/part1/chap1.typ 中：

```
#import "mod.typ": *
```

重导出文件

示例文件结构中的 mod.typ 与编写库时的 lib.typ 很类似，都重新导出了大量函数。例如当你希望同时在 chap2.typ 和 chap3.typ 中使用相似的函数时，你可以在 mod.typ 中原地实现该函数或者从外部库中重导出对应函数：

```
#import "@preview/example:0.1.0": add
// 或者原地实现`add`
#let add(x, y) = x + y
```

这时你可以同时 chap2.typ 和 chap3.typ 中直接使用该 add 函数。

依赖管理

你可以不在 main.typ 或者 chap{N}.typ 文件中直接引入外部库，而在 my-book/mod.typ 中引入外部库。由于级联的 mod.typ，相关函数会传递给每个 main.typ 或者 chap{N}.typ 使用。



Part.5 基础教程 — 排版 III



Typst 架构与原理

在上一节中我们理解了作用域，也知道如何简单把「show」规则应用于文档中的部分内容。

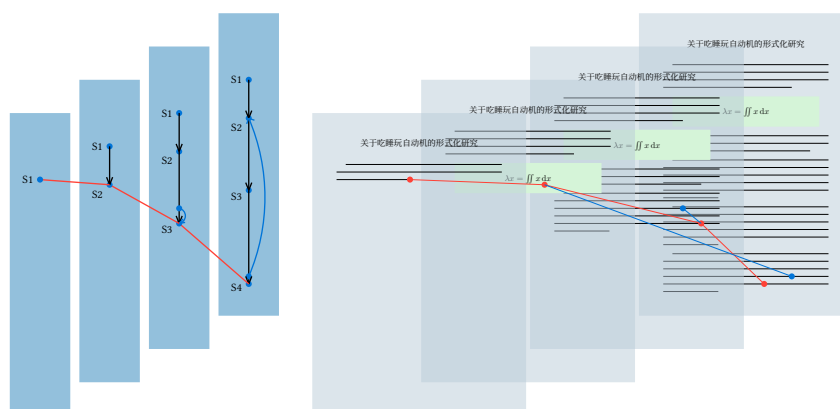
它看起来似乎已经足够强大。但还有一种可能，Typst 可以给你更强大的原语。

我是说有一种可能，Typst 对文档内容的理解至少是二维的。这二维，有一维可以比作空间，另一维可以比作时间。你可以从文档的任意位置，

1. 空间维度 (From Space to Space)：查询文档任意部分的状态（这里的内容和那里的内容）。
2. 时间维度 (From TimeLoc to TimeLoc)：查询脚本执行到文档任意位置的状态（过去的状态和未来的状态）。

这里有一个示意图，红色线段表示 Typst 脚本的执行方向。最后我们形成了一个由 S1 到 S4 的“时间线”。

你可以选择文档中的任意位置，例如你可以在文档的某个位置（蓝色弧线的起始位置），从该处开始查询文档过去某处或未来某处（蓝色弧线的终止位置）。



本节教你使用选择器（selector）定位到文档的任意部分；也教你创建与查询二维文档状态（state）。

自定义标题样式

本节讲解的程序是如何在 Typst 中设置标题样式。我们的目标是：

1. 为每级标题单独设置样式。
2. 设置标题为内容的页眉：
 1. 如果当前页眉有二级标题，则是当前页面的第一个二级标题。
 2. 否则是之前所有页面的最后一个二级标题。

效果如下：

```
#show: set-heading
```

== 雨滴书 v0.1.2

=== KiraKira 样式改进

feat: 改进了样式。

=== FuwaFuwa 脚本改进

feat: 改进了脚本。

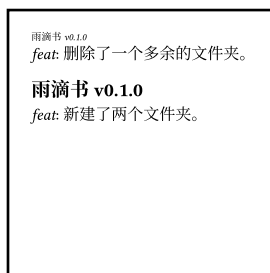
== 雨滴书 v0.1.1

refactor: 移除了 LaTeX。

feat: 删除了一个多余的文件夹。

== 雨滴书 v0.1.0

feat: 新建了两个文件夹。



「样式化」内容

当我们有一个 `repr` 玩具的时候，总想着对着各种各样的对象使用 `repr`。我们在上一节讲解了「`set`」和「`show`」语法。现在让我们稍微深挖一些。

「`set`」是什么，`repr` 一下：

```
#repr({
  [a]; set text(fill: blue); [b]
})
```

```
sequence([a], styled(child: [b], ...))
```

「`show`」是什么，`repr` 一下：

```
#repr({
  [b]; show raw: set text(fill: red)
  [a]
})
```

```
sequence([b], styled(child: [a], ...))
```

我们知道 `set text(fill: blue)` 是 `show: set text(fill: blue)` 的简写，因此「`set`」语法和「`show`」语法都可以统合到第二个例子来理解。

对于第二个例子，我们发现 `show` 语句之后的内容都被重新包裹在 `styled` 元素中。虽然我们不知道 `styled` 做了什么事情，但是简单的事实是：

```
该元素的类型是：#type({show: set text(fill:
blue)}) \
该元素的构造函数是：#({show: set text(fill:
blue)}) .func()
```

```
该元素的类型是：content
该元素的构造函数是：styled
```

原来，你也是内容。从图中，我们可以看到被 `show` 过的内容会被封装成「样式化」内容，即图中构造函数为 `styled` 的内容。

关于 `styled` 的知识便涉及到 `Typst` 的核心架构。

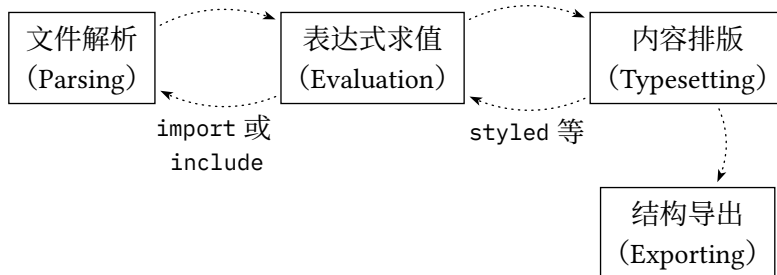
「eval 阶段」与「typeset 阶段」

现在我们介绍 `Typst` 的完整架构。

当 `Typst` 接受到一个编译请求时，他会使用「解析器」（Parser）从 `main` 文件开始解析整个项目；对于每个文件，`Typst` 使用「评估器」（Evaluator）执行脚本并得到「内容」；对于每个「内容」，`Typst` 使用「排版引擎」（Typesetting Engine）计算布局与合成样式。

当一切布局与样式都计算好后，`Typst` 将最终结果导出为各种格式的文件，例如 PDF 格式。

我们回忆上一节讲过的内容，`Typst` 大致上分为四个执行阶段。这四个执行阶段并不完全相互独立，但有明显的先后顺序：



我们在上一节着重讲解了前两个阶段。这里，我们着重讲解“表达式求值”阶段与“内容排版”阶段。事实上，Typst 直接在脚本中提供了对应“求值”阶段的函数，它就是我们之前已经介绍过的函数 `eval`。你可以使用 `eval` 函数，将一个字符串对象「评估」为「内容」：

```

以代码模式评估: #eval("repr(str(1 + 1))") \
以标记模式评估: #eval("repr(str(1 + 1))",
mode: "markup") \
以标记模式评估 2: #eval("#show: it => [c] +
it + [t];a", mode: "markup")

```

```

以代码模式评估: "2"
以标记模式评估: repr(str(1 + 1))
以标记模式评估 2: cat

```

由于技术原因，Typst 并不提供对应“内容排版”阶段的函数，如果有的话这个函数的名称应该为 `typeset`。已经有很多地方介绍了潜在的 `typeset` 函数：

1. [Polylux](#), [Touying](#) 等演示文档（PPT）框架需要将一部分内容固定为特定结果的能力。
2. Typst 的作者在其博客中提及 [Frozen State](#) 的可能性。
 1. 他提及数学公式的编号在演示文档框架。
 2. 即便不涉及用户需求，Typst 的排版引擎已经自然存在 `Frozen State` 的需求。
3. 本文档也需要 `typeset` 的能力为你展示特定页面的最终结果而不影响全局状态。

Typst 的主函数

在 Typst 的源代码中，有一个 Rust 函数直接对应整个编译流程，其内容非常简短，便是调用了两个阶段对应的函数。“求值”阶段（`eval` 阶段）对应执行一个 Rust 函数，它的名称为 `typst::eval`；“内容排版”阶段（`typeset` 阶段）对应执行另一个 Rust 函数，它的名称为 `typst::typeset`。

```

pub fn compile(world: &dyn World) -> SourceResult<Document> {
    // Try to evaluate the source file into a module.
    let module = crate::eval::eval(world, &world.main())?;
    // Typeset the module's content, relayouting until convergence.
    typeset(world, &module.content())
}

```

从代码逻辑上来看，它有明显的先后顺序，似乎与我们所展示的架构略有不同。其 `typst::eval` 的输出为一个文件模块 `module`；其 `typst::typeset` 仅接受文件的内容 `module.content()` 并产生一个已经排版好的文档对象 `typst::Document`。

延迟执行

架构图中还有两个关键的反向箭头，疑问顿生：这两个反向箭头是如何产生的？

我们首先关注与本节直接相关的「样式化」内容。当 `eval` 阶段结束时，「`show`」语法将会对应产生一个 `styled` 元素，其包含了被设置样式的内容，以及设置样式的「回调」：

```

内容是: #repr({show: set text(fill: blue);
[abc]}) \
样式无法描述，但它在这里: #repr({show: set
text(fill: blue); [abc]}.styles)

```

```

内容是: styled(child: [abc], ..)
样式无法描述，但它在这里: ..

```

也就是说 `eval` 并不具备任何排版能力，它只能为排版准备好各种“素材”，并把素材交给排版引擎完成排版。

这里的「回调」术语很关键：它是一个计算机术语。所谓「回调函数」就是一个临时的函数，它会在后续执行过程的合适时机“回过头来被调用”。例如，我们写了一个这样的「show」规则：

```
#repr({
  show raw: content => layout(parent => if
parent.width > 100pt {
  set text(fill: red); content
} else {
  content
})
`a`
})
```

styled(child: raw(text: "a", block: false), ..)

这里 `parent.width > 100pt` 是说当且仅当父元素的宽度大于 100pt 时，才为该代码片段设置红色字体样式。其中，`parent.width` 与排版相关。那么，自然 `eval` 也不知道该如何评估该条件的真正结果。

计算因此被停滞。

于是，`eval` 干脆将整个 `show` 右侧的函数都作为“素材”交给了排版引擎。当排版引擎计算好了相关内容，才回到评估阶段，执行这一小部分“素材”函数中的脚本，得到为正确的内容。我们可以看出，`show` 右侧的函数**被延后执行**可。

这种被延后执行零次、一次或多次的函数便被称为「回调函数」。相关的计算方法也有对应的术语，被称为「延迟执行」。

我们对每个术语咬文嚼字一番，它们都很准确：

1. 「**表达式求值**」阶段仅仅“评估”出「**内容排版**」阶段所需的素材。「**评估器**」并不具备排版能力。
2. 对于依赖排版产生的内容，「表达式求值」产生包含「**回调函数**」的内容，让「排版引擎」在合适的时机“回过头来调用”。
3. 相关的计算方法又被称为「**延迟执行**」。因为现在不具备执行条件，所以延迟到条件满足时才继续执行。

现在我们可以理解两个反向箭头是如何产生的了。它们是下一阶段的回调，用于完成阶段之间复杂的协作。评估阶段可能会 `import` 或 `include` 文件，这时候会重新让解析器解析文件的字符串内容。排版阶段也可能会继续根据 `styled` 等元素产生复杂的内容，这时候依靠评估器执行脚本并产生或改变内容。

模拟 Typst 的执行

我们来模拟一遍上述示例的执行，以加深理解：

```
#show raw: content => layout(parent => if
parent.width < 100pt {
  set text(fill: red); content
} else {
  content
})
#box(width: 50pt, `a`)
`b`
```

a
b

首先进行表达式求值得到：

```
#styled((box(width: 50pt, `a`), `b`), styles: content => ..)
```

排版引擎遇到 ``a``。由于 ``a`` 是 `raw` 元素，它「回调」了对应 `show` 规则右侧的函数。待执行的代码如下：

```
layout(parent => if parent.width < 100pt {
  set text(fill: red); `a`
} else {
  `a`
})
```

此时 parent 即为 box(width: 50pt)。排版引擎将这个 parent 的具体内容交给「评估器」，待执行的代码如下：

```
if box(width: 50pt).width < 100pt {
  set text(fill: red); `a`
} else {
  `a`
}
```

由于此时父元素（box 元素）宽度只有 50pt，评估器进入了 then 分支，其为代码片段设置了红色样式。内容变为：

```
#{box(width: 50pt, {set text(fill: red); `a`}), styled(`b`, ), styles: content => ..)}
```

待执行的代码如下：

```
set text(fill: red); text("a", font: "monospace")
```

排版引擎遇到 `a` 中的 text 元素。由于其是 text 元素，「回调」了 text 元素的「show」规则。记得我们之前说过 set 是一种特殊的 show，于是排版器执行了 set text(fill: red)。

```
#{box(width: 50pt, text(fill: red, "a", ..)), styled(`b`, ), styles: content => ..)}
```

排版引擎离开了 show 规则右侧的函数，该函数调用由 `a` 触发。同时 set text(fill: red) 规则也被解除，因为离开了相关作用域。

回到文档顶层，待执行的代码如下：

```
show raw: ...
`b`
```

排版引擎遇到 `b`，再度「回调」了对应 show 规则右侧的函数。由于此时父元素（page 元素，即整个页面）宽度有 500pt，我们没有为代码片段设置样式。

```
#{box(width: 50pt, text(fill: red, "a", ..)), text("b", ..)}
```

至此，文档的内容已经准备好「导出」(Export) 了。

「样式化」内容的补充

有时候 show 规则会原地执行，这属于一种细节上的优化，例如：

```
repr({ show: it => it; [a] }) \
repr({ show: it => [c] + it + [d]; [a] })
```

```
[a]
sequence([c], [a], [d])
```

这个时候 show 规则不会对应一个 styled 元素。

这种优化告诉你前面手动描述的过程仅作参考。一旦涉及更复杂的环境，Typst 的实际执行过程就会产生诸多变化。因此，你不应该依赖以上某步中排版引擎的瞬间状态。这些瞬间状态将产生「未注明特性」(undocumented details)，并随时有可能在未来被打破。

「可定位」的内容

在过去的章节中，我们了解了评估结果的具体结构，也大致了解了排版引擎的工作方式。

接下来，我们介绍一类内容的「可定位」(Locatable) 特征。你可以与前文中的「可折叠」(Foldable) 特征对照理解。

一个内容是可定位的，如果它可以以某种方式被索引得到。

如果一个内容在代码块中，并未被使用，那么显然这种内容是不可定位的。

```
{ let unused-content = [一段不可定位的内容]; }
```

理论上文档中所有内容都是可定位的，但由于性能限制，Typst 无法允许你定位文档中的所有内容。

我们已经学习过元素函数可以用来定位内容。如下：

```
#show heading: set text(fill: blue)
= 蓝色标题
段落中的内容保持为原色。
```

蓝色标题

段落中的内容保持为原色。

接下来我们继续学习更多选择器。

文本选择器

你可以使用「字符串」或「正则表达式」(regex) 匹配文本中的特定内容, 例如为 c++ 文本特别设置样式:

```
#show "cpp": strong(emph(box("C++")))
在古代, cpp 是一门常用语言。
```

在古代, C++ 是一门常用语言。

这与使用正则表达式的效果相同:

```
#show regex("cp{2}"): strong(emph(box("C+
+")))
在古代, cpp 是一门常用语言。
```

在古代, C++ 是一门常用语言。

关于正则表达式的知识, 推荐在 [Regex 101](#) 中继续学习。

这里讲述一个关于 regex 选择器的重要知识。当文本被元素选中时, 会创建一个不可见的分界, 导致分界之间无法继续被正则匹配:

```
#show "ab": set text(fill: blue)
#show "a": set text(fill: red)
ababababababa
```

ababababababa

因为 "a" 规则比 "ab" 规则更早应用, 每个 a 都被单独分隔, 所以 "ab" 规则无法匹配到任何文本。

```
#show "a": set text(fill: red)
#show "ab": set text(fill: blue)
ababababababa
```

ababababababa

虽然每个 ab 都被单独分隔, 但是 "a" 规则可以继续在内部分界内继续匹配文本。

这个特征在设置文本的字体时需要特别注意:

为引号单独设置字体导致错误的排版结果。因为句号与双引号之间产生了分界, 使得 Typst 无法应用标点挤压规则:

```
#show "": it => {
  set text(font: "KaiTi")
  highlight(it, fill: yellow)
}
“无名, 万物之始也; 有名, 万物之母也。”
```

“无名, 万物之始也; 有名, 万物之母也。”

以下正则匹配也会导致句号与双引号之间产生分界, 因为没有对两个标点进行贪婪匹配:

```
#show regex("[\"。]"): it => {
  set text(font: "KaiTi")
  highlight(it, fill: yellow)
}
“无名, 万物之始也; 有名, 万物之母也。”
```

“无名, 万物之始也; 有名, 万物之母也。”

以下正则匹配没有在句号与双引号之间创建分界。考虑两个标点的字体设置规则, Typst 能排版出这句话的正确结果:

```
#show regex("[\"。]+"): it => {
  set text(font: "KaiTi")
  highlight(it, fill: yellow)
}
“无名, 万物之始也; 有名, 万物之母也。”
```

“无名, 万物之始也; 有名, 万物之母也。”

标签选择器

基本上，任何元素都包含文本。这使得你很难对一段话针对性排版应用排版规则。「标签」有助于改善这一点。标签是「内容」，由一对「尖括号」(<和>)包裹：

```
一句话 <some-label>
```

一句话

「标签」可以选中恰好在它之前的一个内容。示例中，<some-label>选中了文本内容一句话。

也就是说，「标签」无法选中在它之前的多个内容。以下选择器选中了 #[] 后的一句话：

```
#show <一句话>: set text(fill: blue)
#[一句话。]还是一句话。 <一句话>
```

另一句话。

一句话。还是一句话。

另一句话。

这是因为 #[一句话。] 被分隔为了单独的内容。

我们很难判断一段话中有多少个内容。因此为了可控性，我们可以使用内容块将一段话括起来，然后使用标签准确选中这一整段话：

```
#show <一整段话>: set text(fill: blue)
#[
  $lambda$ 语言是世界上最好的语言。#[]还是一句话。
] <一整段话>
```

另一段话。

λ语言是世界上最好的语言。还是一句话。

另一段话。

选择器表达式

任意「内容」可以使用「where」方法创建选中满足条件的选择器。

例如我们可以选中二级标题：

```
#show heading.where(level: 2): set
text(fill: blue)
== 一级标题
== 二级标题
```

一级标题

二级标题

这里 heading 是一个元素，heading.where 创建一个选择器：

```
选择器是: #repr(heading.where(level: 2)) \
类型是: #type(heading.where(level: 2))
```

选择器是: heading.where(level: 2)
类型是: selector

同理我们可以选中行内的代码片段而不选中代码块：

```
#show raw.where(block: false): set
text(fill: blue)
`php` 是世界上最好的语言。
` `
typst 也是。
` `
```

php 是世界上最好的语言。

typst 也是。

回顾其一

针对特定的 feat 和 refactor 文本，我们使用 emph 修饰：

```
#show regex("feat|refactor"): emph
```

雨滴书 v0.1.2

KiraKira 样式改进

feat: 改进了样式。

FuwaFuwa 脚本改进

feat: 改进了脚本。

雨滴书 v0.1.1

refactor: 移除了 LaTeX。

feat: 删除了一个多余的文件夹。

雨滴书 v0.1.0

feat: 新建了两个文件夹。

对于三级标题，我们将中文文本用下划线标记，同时将特定文本替换成 emoji：

```
#let set-heading(content) = {
  show heading.where(level: 3): it => {
    show regex("[p{hani}\s]+"): underline
    it
  }
  show heading: it => {
    show regex("KiraKira"): box("★", baseline: -20%)
    show regex("FuwaFuwa"): box(text("👉", size: 0.5em), baseline: -50%)
    it
  }
  content
}
```

```
#show: set-heading
```

雨滴书 v0.1.2

★ 样式改进

feat: 改进了样式。

👉 脚本改进

feat: 改进了脚本。

雨滴书 v0.1.1

refactor: 移除了 LaTeX。

feat: 删除了一个多余的文件夹。

雨滴书 v0.1.0

feat: 新建了两个文件夹。

制作页眉标题的两种方法

制作页眉标题至少有两种方法。一是直接查询文档内容；二是创建状态，利用布局迭代收敛的特性获得每个页面的首标题。

在接下来的两节中我们将分别介绍这两种方法。

查询文档状态

本节我们讲解制作页眉标题的第一种方法，即通过查询文档状态直接估计当前页眉应当填入的内容。

「locate」

有的时候我们会需要获取当前位置的「位置」信息。

在 Typst 中，获取「位置」的唯一方法是使用「`locate`」函数。

`locate` 的唯一参数是一个「回调函数」，我们在上一节已经学习过这个术语。在排版阶段的过程中，一旦排版引擎为你收集到了足够的信息，就会“回过头来调用”我们传入的该函数。

我们来 `repr` 一下试试看：

```
当前位置的相关信息: #locate(it => repr(it))
```

```
当前位置的相关信息: ..
```

由于「位置」太过复杂了，`repr` 放弃了思考并在这里为我们放了两个点。

我们来简单学习一下 Typst 为我们提供了哪些位置信息：

```
当前位置的坐标: #locate(loc =>
loc.position())
```

```
当前位置的页码: #locate(loc => loc.page())
```

```
当前位置的坐标: (page: 94, x: 396.89pt,
y: 331.02pt)
```

```
当前位置的页码: 94
```

或直接使用方法函数：

```
当前位置的坐标: #locate(location.position)
```

```
当前位置的页码: #locate(location.page)
```

```
当前位置的坐标: (page: 94, x: 396.89pt,
y: 413.97pt)
```

```
当前位置的页码: 94
```

一个常见的问题是：为什么 Typst 提供给我的页码信息是「内容」，我无法在内容上做条件判断和计算！

```
#repr(locate(location.page)) \
#type(locate(location.page))
```

```
locate(func: page)
content
```

上面输出的内容告诉我们 `locate` 不仅是一个函数，而且更是一个元素的构造函数。`locate` 构造出一个 `locate` 内容。

这其中的关系比较复杂。一个比较好理解的原因是：Typst 会调用你的函数多次，因此你理应将所有使用「位置」信息的脚本放在一个内容块中，这样 Typst 才能更好地合成内容。

```
#locate(loc => [ 当前位置的页码是偶
数: #calc.even(loc.page()) ])
// 根据位置信息 计算得到 我们想要的内容
```

```
当前位置的页码是偶数: true
```

❗ 这与 Typst 的缓存原理相关。由于 `locate` 函数接收的闭包 `loc => ..` 是一个函数，且在 Typst 中它被认定为**纯函数**，Typst 只会针对特定的参数执行一次函数。为了强制让用户书写的函数保持纯性，Typst 规定用户必须在函数内部使用「位置」信息。

因此，例如我们希望在偶数页下让内容为“甲”，否则让内容为“乙”，应当这样书写：

```
#locate(loc => if calc.even(loc.page())
[ “甲” ] else [ “乙” ])
```

```
“甲”
```

「query」

使用「`query`」函数你可以获得当前文档状态。

```
#locate(loc => query(heading, loc))
```

```
(
  heading(body: [雨滴书 v0.1.2], level:
2, ..),
  ...
)
```

「`query`」函数有两个参数。

第一个参数是「选择器」，很好理解。它接受一个选择器，并返回被选中的所有元素的列表。

```
#locate(loc => query(heading, loc))
```

```
(
  heading(body: [雨滴书 v0.1.2], level:
2, ..),
  heading(body: [KiraKira 样式改进],
level: 3, ..),
  heading(body: [FuwaFuwa 脚本改进],
level: 3, ..),
  heading(body: [雨滴书 v0.1.1], level:
2, ..),
  heading(body: [雨滴书 v0.1.0], level:
2, ..),
)
```

我们记得，选择器允许继续指定 `where` 条件过滤内容：

```
#locate(loc => query(heading.where(level:
2), loc))
```

```
(
  heading(body: [雨滴书 v0.1.2], level:
2, ..),
  heading(body: [雨滴书 v0.1.1], level:
2, ..),
  heading(body: [雨滴书 v0.1.0], level:
2, ..),
)
```

第二个参数是「位置」，就比较难以理解了。首先说明，`loc` 并没有任何作用，即它是一个「哑参数」(Dummy Parameter)。

如果你学过 C++，以下两个方法分别匹配到前缀自增操作函数和后缀自增操作函数。

```
class Integer {
  Integer& operator++(); // 前缀自增操作函数
  Integer operator++(int); // 后缀自增操作函数
};
```

`class Integer` 类中的 `int` 就是一个所谓的哑参数。

「哑参数」在实际函数执行中并未被使用，而仅仅作为标记以区分函数调用。我们知道以下两点：其一，只有 `locate` 函数会返回「位置」信息；其二，`query` 函数需要我们传入「位置」信息。

有了：那么 Typst 就是在告诉我们，`query` 函数只能在 `locate` 函数内部调用。正如示例中的那样：

```
#locate(loc => query(heading.where(level: 2), loc))
```

这个规则有些隐晦，并且 Typst 的设计者也已经注意到了这一点，所以他们正在计划改进这一点。当然在这之前，你只需要记住：`query` 函数的第二个「位置」参数用于限制该函数仅在 `locate` 函数内部使用。

❗ 这与 Typst 的缓存原理相关。为了加速 `query` 函数，Typst 需要对其缓存。Typst 合理做出以下假设：在文档每处的查询 (`loc`)，都单独缓存对应选择器的查询结果。

更细致地描述如下：将 `query(selector, loc)` 的参数为「键」，执行结果为「值」构造一个哈希映射表。若使用 `(selector, loc)` 作为「键」，查询该表：

1. 未对应结果，则执行查询，缓存并返回结果。
2. 已经存在对应结果，则不会重新执行查询，而是使用表中的值作为结果。

回顾其二

页眉的设置方法是创建一条 `set page(header)` 规则：

```
#set page(header: [这是页眉])
```



既然如此，只需要将 `[这是页眉]` 替换成一个 `locate` 内容，就能通过 `query` 函数完成与「位置」相关的页眉设定：

```
#set page(header: locate(loc => emph(
  get-heading-at-page(loc))))
```

现在让我们来编写 `get-heading-at-page` 函数。

首先，通过 `query` 函数查询得到整个文档的所有二级标题：

```
let headings = query(heading, loc).
  filter(it => it.level == 2)
```

如果你熟记 `where` 方法，你可以更高效地做到这件事。以下函数调用也可以得到整个文档的所有二级标题：

```
let headings = query(heading.where(level: 2), loc)
```

很好，Typst 文档可以很高效，但有些人写出的 Typst 代码天生更高效，而我们正在向他们靠近。

接着，考虑构建这样一个 `fold-headings` 函数，它返回一个数组，数组的内容是每个页面页眉应当显示的内容，即每页的第一个标题。

```
#let fold-headings(headings) = {
  ..
}
```

我们可以对其直接调用以测试：

```
#let fold-headings(headings) = {
  (none, none)
}
#fold-headings((
  (body: "v2", page: 1), (body: "v1", page: 1),
  (body: "v0", page: 2),
))
```

(none, none)

很好，这样我们就可以很方便地进行测试了。

该函数首先创建一个数组，数组的长度为页面的数量。


```
#let fold-headings(headings) = {
  let max-page-num =
    calc.max(..headings.map(it => it.page))
    (none, ) * max-page-num
}
#fold-headings((
  (body:"v2",page:1),(body:"v1",page:1),
  (body:"v0",page: 2),
))
```

```
(none, none)
```

这里，`headings.map(it => it.page)`意即对每个标题获取对应位置的页码；`calc.max(..numbers)`意即取页码的最大值。

由于示例中页码的最大值为2，`fold-headings`针对示例会返回一个长度为2的数组，数组的每一项都是 `none`。

接着，该函数遍历给定的 `headings`，对每个页码，都首先获取第一个标题元素：

```
for h in headings {
  if first-headings.at(h.page - 1) == none
  {
    first-headings.at(h.page - 1) = h
  }
}
```

```
((body: "v2", page: 1), (body: "v0",
page: 2))
```

这里，`first-headings.at(h.page - 1)`意即获取当前页码对应应在数组中的元素；`if` 语句，如果对应页码对应的元素仍是 `none`，那么就将当前的标题元素填入对应的位置中。

同理，可以获得 `last-headings`，存储每页的最后一个标题：

```
let last-headings = (none, ) * max-page-
num
for h in headings {
  last-headings.at(h.page - 1) = h
}
```

```
((body: "v1", page: 1), (body: "v0",
page: 2))
```

这里 `for` 语句意即：无论如何，都将当前的标题元素存入数组中。那么每页的最后一个标题总是能被存入到数组中。

但是我们还没有考虑相邻情况。如果我们希望如果当前页面没有标题元素，则显示之前的标题。接下来我们根据这个思路，组装正确的结果：

```
let res-headings = (none, ) * max-page-num
for i in range(res-headings.len()) {
  res-headings.at(i) = if first-
headings.at(i) != none {
    first-headings.at(i)
  } else {
    last-headings.at(i) = last-
headings.at(
    calc.max(0, i - 1), default: none)
    last-headings.at(i)
  }
}
```

```
((body: "v2", page: 1), (body: "v0",
page: 2))
```

`res-headings` 就是我们想要得到的结果。

由于 `res-headings` 的计算比较复杂，我们先来一些测试用例来理解：

情形一：假设文档的前段没有标题，该函数会将对应下标的结果置空：

```
情形一: #fold-headings((
  (body:"v2",page:3),(body:"v1",page:3),
  (body:"v0",page: 3),
))
```

```
情形一: (none, none, (body: "v2", page:
3))
```

情形二: 假设一页有多个标题, 那么, 对应下表的结果为该页面的首个标题:

```
情形二: #fold-headings((
  (body:"v2",page:2),(body:"v1",page:2),
  (body:"v0",page: 3),
))
```

```
情形二: (none, (body: "v2", page: 2),
(body: "v0", page: 3))
```

情形三: 假设中间有页空缺, 则对应下表的结果为前页的最后一个标题。

```
情形三: #fold-headings((
  (body:"v2",page:1),(body:"v1",page:1),
  (body:"v0",page: 3),
))
```

```
情形三: (
  (body: "v2", page: 1),
  (body: "v1", page: 1),
  (body: "v0", page: 3),
)
```

其中, 情形一其实是情形三的特例: 假设某一页没有标题, 则对应下表的结果为前页的最后一个标题。如果不存在前页包含标题, 则对应下表的结果为 `none`。

于是我们可以给代码加上注释:

```
let res-headings = (none, ) * max-page-num
// 对于每一页, 我们迭代下标
for i in range(res-headings.len()) {
  // 让对应下标的结果等于:
  res-headings.at(i) = {
    // 如果该页包含标题, 则其等于该页的第一个标题
    if first-headings.at(i) != none {
      first-headings.at(i)
    } else {
      // 否则, 我们积累`last-headings`的结果
      last-headings.at(i) = last-headings.at(
        // 始终至少等于前一页的结果
        calc.max(0, i - 1),
        // 默认没有结果
        default: none)
      // 其等于前页的最后一个标题
      last-headings.at(i)
    }
  }
}
```

最后, 我们将 `query` 与 `fold-headings` 结合起来, 便得到了目标函数:

```
#let get-heading-at-page(loc) = {
  let headings = fold-headings(query(
    heading.where(level: 2), loc))
  headings.at(loc.page() - 1)
}
```

这里有一个问题, 那便是 `fold-headings` 没有考虑标题的最后一页仍然存在内容的情况。例如第二页有最后一个标题, 但是我们文档一共有三页。

重新改造一下:

```
#let calc-headings(headings) = {
  // 计算 res-headings 和 last-headings
  ..

  // 同时返回最后一个标题
  (res-headings, if max-page-num > 0 {
    last-headings.at(-1)
  })
}
```

我们来简单测试一下：

```
情形一: #calc-headings((
  (body: "v2", page: 3), (body: "v1", page: 3),
  (body: "v0", page: 3),
)).at(1) \
情形二: #calc-headings((
  (body: "v2", page: 2), (body: "v1", page: 2),
  (body: "v0", page: 3),
)).at(1) \
情形三: #calc-headings((
  (body: "v2", page: 1), (body: "v1", page: 1),
  (body: "v0", page: 3),
)).at(1)
```

```
情形一: (body: "v0", page: 3)
情形二: (body: "v0", page: 3)
情形三: (body: "v0", page: 3)
```

很好，这样，下面的实现就完全正确了：

```
#let get-heading-at-page(loc) = {
  let (headings, last-heading) = calc-headings(
    query(headings.where(level: 2), loc))
  headings.at(loc.page() - 1, default: last-heading)
}
```

❶ 将 `calc-headings` 与 `get-heading-at-page` 分离可以改进脚本性能。这是因为 Typst 是以函数为粒度缓存你的计算。在最后的实现：

1. `query(headings.where(level: 2), loc)` 会被缓存，如果二级标题的结果不变，则 `query` 函数不会重新执行（不会重新查询文档状态）。
2. `calc-headings(..)` 会被缓存。如果查询的结果不变。则其不会重新执行。

最后，让我们适配 `calc-headings` 到真实场景，并应用到页眉规则：

```
// 这里有 get-heading-at-page 的实现..

#set page(header: locate(loc => emph(
  get-heading-at-page(loc)))
```

```
雨滴书 v0.1.2
雨滴书 v0.1.2

★ 样式改进
feat: 改进了样式。

🔧 脚本改进
feat: 改进了脚本。

雨滴书 v0.1.1
refactor: 移除了 LaTeX。
```

```
雨滴书 v0.1.0
feat: 删除了一个多余的文件夹。

雨滴书 v0.1.0
feat: 新建了两个文件夹。
```

维护文档状态

在上一节（法一）中，我们仅靠「`query`」函数就完成制作所要求页眉的功能。

思考下面函数：

```
#let get-heading-at-page(loc) = {  
  let (headings, last-heading) = calc-headings(  
    query(heading.where(level: 2), loc))  
  headings.at(loc.page() - 1, default: last-heading)  
}
```

对于每个页面，它都运行 `query(heading.where(level: 2), loc)`。显然每页的「位置」信息，即 `loc` 对应不相同。因此：

1. 它会每页都重新执行一遍 `heading.where(level: 2)` 查询。
2. 同时，每次 `query` 都是对全文档的线性扫描。

夸张来讲，假设你有一千页文档，文档中包含上千个二级标题；那么他将会使得你的每次标题更新都触发上百万次迭代。虽然 Typst 很快，但这上百万次迭代将会使得包含这种页眉的文档难以实时预览。

那么有没有一种方法避免这种全文档扫描呢？

本节将介绍法二，它基于「`state`」函数，持续维护页眉状态。

Typst 文档可以很高效，但有些人写出的 Typst 代码更高效。本节所介绍的法二，让我们变得更接近这种人。

「`state`」函数

`state` 接收一个名称，并创建该名称对应全局唯一的状态变量。

```
#state("my-state", 1)
```

```
state("my-state", 1)
```

你可以使用 `state.display()` 函数展示其「内容」：

```
#let s1 = state("my-state", 1)  
s1: #s1.display()
```

```
s1: 1
```

你可以使用 `state.update()` 方法更新其状态。`update` 函数接收一个「回调函数」，该回调函数接收 `state` 在某时刻的状态，并返回对应下一时刻的状态：

```
#let s1 = state("my-state", 1)  
s1: #s1.display() \  
#s1.update(it => it + 1)  
s1: #s1.display()
```

```
s1: 1  
s1: 2
```

所有的相同名称的内容将会共享更新：

```
#let s1 = state("my-state", 1)  
s1: #s1.display() \  
#let s2 = state("my-state", 1)  
s1: #s1.display(), s2: #s2.display() \  
#s2.update(it => it + 1)  
s1: #s1.display(), s2: #s2.display()
```

```
s1: 1  
s1: 1, s2: 1  
s1: 2, s2: 2
```

同时，请注意状态的**全局**特性，即便处于不同文件、不同库的状态，只要字符串对应相同，那么其都会共享更新。

这提示我们需要在不同的文件之间维护全局状态的名称唯一性。

另一个需要注意的是，`state` 允许指定一个默认值，但是一个良好的状态设置必须保持不同文件之间的默认值相同。如下所示：

```
#let s1 = state("my-state", 1)
s1: #s1.display() \
#let s2 = state("my-state", 2)
s1: #s1.display(), s2: #s2.display() \
#s2.update(it => it + 1)
s1: #s1.display(), s2: #s2.display()
```

```
s1: 1
s1: 1, s2: 1
s1: 2, s2: 2
```

尽管 `s2` 指定了状态的默认值为 2, 因为之前已经在文档中创建了该状态, 默认值并不会应用。请注意: 你不应该利用这个特性, 该特性是 Typst 中的「未定义行为」。

「state.update」也是「内容」

一个值得注意的地方是, 似乎与 `locate` 函数相似, `state.update` 也接收一个闭包。

事实上, 与 `locate` 函数相同, `state.update` 也具备延迟执行的特性。

让我们检查下列脚本的输出结果:

```
#let s1 = state("my-state", 1)
#((s1.update(it => it + 1), ) * 3).join()
s1: #s1.display()
```

```
s1: 4
```

这告诉我们下面一件事情, 当 eval 阶段结束时, 其对应产生下面的一段内容:

```
(
  state("my-state", 1),
  state("my-state").update(it => it + 1),
  state("my-state").update(it => it + 1),
  state("my-state").update(it => it + 1),
)
```

排版引擎将按照**深度优先的顺序**遍历你的内容, 从文档的开始位置逐渐**积累**状态。

这将帮助我们在多文件之间协助完成状态的更新与计算。

假设我们有两个文件 `s1.typ` 和 `s2.typ`, 文件的内容分别是:

```
// s1.typ
#let s1 = state("my-state", (1, ))
#s1.update(it => it + (3, ))
// s2.typ
#let s1 = state("my-state", (2, ))
#s1.update(it => it + (4, ))
#s1.update(it => it + (5, ))
```

并且我们在 `main.typ` 中引入了上述两个文件:

```
#include "s1.typ"
#include "s2.typ"
```

那么根据我们的经验, 主文件内容其实对应为:

```
// 省略部分内容
#{ { state("my-state", (1, )), .. } + { state("my-state", (2, )), .. } }
```

我们按照顺序执行状态更新, 则状态依次变为:

```
#{ (1, ); (1, 3, ); }
#{ (1, 3, ); (1, 3, 4, ); (1, 3, 4, 5, ); }
```

查询特定时间点的「状态」

Typst 提供两个方法查询特定时间点的「状态」:

一个方法是 `state.at(loc)` 方法, 其接收一个「位置」, 返回在该位置对应的状态「值」。

另一个方法是 `state.final(loc)` 方法，其接收一个「位置」，返回在文档结束一切排版时对应的状态「值」。

熟悉的剧情再次发生了。让我们回想之前介绍 `query` 时讲述的知识点。

这两个方法都只能在 `locate` 内部调用。对于 `state.at` 方法，其「位置」参数是有用的；对于 `state.final` 方法，其「位置」参数仅仅作「哑参数」。

我们回想上一小节，由于文档的每个位置「状态」都会存有对应的值，而且当你使用状态的时候至少会指定一个默认值，我们可以知道在我们文档的任意位置使用文档的任意其他位置的状态的内容。

这就是允许我们进行时光回溯的基础。

「typeset」阶段的迭代收敛

一个容易值得思考的问题是，如果我在文档的开始位置调用了 `state.final` 方法，那么 Typst 要如何做才能把文档的最终状态返回给我呢？

容易推测出，原来 Typst 并不会只对内容执行一遍「typeset」。仅考虑我们使用 `state.final` 方法的情况。初始情况下 `state.final` 方法会返回状态默认值，并完成一次布局。接下来的迭代，`state.final` 方法会返回上一次迭代布局完成时的。直到布局的内容不再发生变化。`state.at` 会导致相似的布局迭代，只不过情况更为复杂，这里便不再展开细节。

所有对文档的查询都会导致布局的迭代：`query` 函数可能会导致布局的迭代；`state.at` 函数可能会导致布局的迭代；`state.final` 函数一定会导致布局的迭代。

回顾其三

本节使用递归的方法完成状态的构建，其更为巧妙。

首先，我们声明两个与法一类似的状态，只不过这次我们将状态定义在全局。

```
#let first-heading = state("first-heading", ())
#let last-heading = state("last-heading", ())
```

然后，我们在每个二级标题后紧接着触发一个更新：

```
show heading.where(level: 2): curr-heading => {
  curr-heading
  locate(loc => ..)
}
```

对于 `last-heading` 状态，我们可以非常简单地如此更新内容：

```
last-heading.update(headings => {
  headings.insert(str(loc.page()), curr-heading.body)
  headings
})
```

每页的最后一个标题总能最后触发状态更新，所以 `str(loc.page())` 总是能对应到每页的最后一个标题的内容。

对于 `first-heading` 状态，稍微复杂但也好理解：

```
first-heading.update(headings => {
  let k = str(loc.page())
  if k not in headings {
    headings.insert(k, curr-heading.body)
  }
  headings
})
```

对于每页靠后的一级标题，都不能使 `if` 条件成立。所以 `str(loc.page())` 总是能对应到每页的第一个一级标题的内容。

接下来便是简单的查询了，我们回忆之前 `get-heading-at-page` 的逻辑，它首先判断是否存在本页的第一个标题，否则取前页的最后一个标题。以下函数完成了前半部分：

```
let get-heading-at-page(loc) = {  
  let page-num = loc.page()  
  let first-headings = first-heading.final(loc)  
  
  first-headings.at(str(page-num))  
}
```

我们为 `at` 函数添加 `default` 函数，其取前页的最后一个标题。

```
let get-heading-at-page(loc) = {  
  ..  
  let last-headings = last-heading.at(loc)  
  
  first-headings.at(str(page-num), default: find-headings(last-headings, page-num))  
}
```

我们使用递归的方法实现 `find-headings`：

```
let find-headings(headings, page-num) = if page-num > 0 {  
  headings.at(str(page-num), default: find-headings(headings, page-num - 1))  
}
```

递归有两个分支：递归的基是，若一直找到文档最前页都找不到相应的标题，则返回 `none`。否则检查 `headings` 中是否有对应页的标题：若有则直接返回其内容，否则继续往前页迭代。

一个细节值得注意，我们对 `first-heading` 使用了 `final` 方法，但对 `last-heading` 使用了 `at` 方法。这是因为：

1. `first-heading` 需要我们支持后向查找，因此需要直接获取文档最终的状态。
2. `last-heading` 仅仅需要前向查找，因此使用 `at` 方法可以改进迭代效率。

这个递归函数是高性能的，因为 `Typst` 会对 `find-headings` 缓存，并且 `Typst` 对于后一页的内容，都总是能命中前一页的缓存。

与之相反，基于 `query` 的实现没有那么好命。它没法很好利用递归完成标题信息的构建。这是因为 `query` 的实现中，`calc-headings` 的首次执行就被要求计算文档的所有标题。

最后让我们设置页眉：

```
// 这里有 get-heading-at-page 的实现..  
  
#set page(header: locate(loc => emph(  
  get-heading-at-page(loc)))
```





Part.6 基础教程 — 附录 I



参考：语法示例检索表

点击下面每行名称都可以跳转到对应章节的对应小节。

分类：基本元素

「名称/术语」	语法	渲染结果
段落	writing-markup	writing-markup
标题	= Heading	Heading
二级标题	== Heading	Heading
着重标记	*Strong*	Strong
强调标记	<i>_emphasis_</i>	<i>emphasis</i>
着重且强调标记	<i>*_emphasis_*</i>	<i>emphasis</i>
有序列表	+ List 1 + List 2	1. List 1 2. List 2
有序列表（重新开始标号）	4. List 1 + List 2	4. List 1 5. List 2
无序列表	- Enum 1 - Enum 2	• Enum 1 • Enum 2
交替有序与无序列表	- Enum 1 + Item 1 - Enum 2	• Enum 1 1. Item 1 • Enum 2
短代码片段	`code`	code
长代码片段	```code```	code
语法高亮	```rs trait World```	trait World
块代码片段	```typ = Heading ```	= Heading
图像	<code>#image("/assets/files/香風とうふ店.jpg", width: 50pt)</code>	
拉伸图像	<code>#image("/assets/files/香風とうふ店.jpg", width: 50pt, height: 50pt, fit: "stretch")</code>	
内联图像（盒子法）	在一段话中插入一个 <code>#box(baseline: 0.15em, image("/assets/files/info-icon.svg", width: 1em))</code> 图片。	在一段话中插入一个  图片。

图像标题	<pre>#figure(``typ #image("/assets/files/香風とうふ 店.jpg") ``, caption: [用于加载香風とうふ店 送外卖的宝贵影像的代码])</pre>	<pre>#image("/assets/files/香風 とうふ店.jpg")</pre> <div>代码 3 用于加载香風とうふ店 送外卖的宝贵影像的代码</div>				
链接	<pre>#link("https://zh.wikipedia. org") [维基百科]</pre>	维基百科				
HTTP(S)链接	<pre>https://zh.wikipedia.org</pre>	https://zh.wikipedia.org				
内部链接	<pre>== 某个标题 <ref-internal-link> #link(<ref-internal-link>) [链接到 某个标题]</pre>	某个标题 链接到某个标题				
表格	<pre>#table(columns: 2, [111], [2], [3])</pre>	<table><tr><td>111</td><td>2</td></tr><tr><td>3</td><td></td></tr></table>	111	2	3	
111	2					
3						
对齐表格	<pre>#table(columns: 2, align: center, [111], [2], [3])</pre>	<table><tr><td>111</td><td>2</td></tr><tr><td>3</td><td></td></tr></table>	111	2	3	
111	2					
3						
行内数学公式	<pre>\$sum_x\$</pre>	\sum_x				
行间数学公式	<pre>\$ sum_x \$</pre>	\sum_x				
标记转义序列	<pre>>_<</pre>	$>_<$				
标记的 Unicode 转义序列	<pre>\u{9999}</pre>	香				
换行符（转义序列）	<pre>A \ B</pre>	A B				
换行符情形二	<pre>A \ B</pre>	A B				
速记符	<pre>北京--上海</pre>	北京-上海				
空格（速记符）	<pre>A~B</pre>	A B				
行内注释	<pre>// 行内注释</pre>					
行间注释	<pre>/* 行间注释 */</pre>					
行内盒子	在一段话中插入一个 <code>#box(baseline: 0.15em, image("/assets/files/info-icon.svg", width: 1em))</code> 图片。	在一段话中插入一个  图片。				

分类：修饰文本

「名称/术语」	语法	渲染结果
背景高亮	<code>#highlight[高亮一段内容]</code>	高亮一段内容

下划线	<code>#underline[Language]</code>	Language
无驱逐效果的下划线	<code>#underline(evade: false)[□□□□]</code>	□□□□
上划线	<code>#overline[□□□□]</code>	□□□□
中划线（删除线）	<code>#strike[□□□□]</code>	□□□□
下标	威严满满 <code>#sub[抱头蹲防]</code>	威严满满 _{抱头蹲防}
上标	香風とうふ店 <code>#super[TM]</code>	香風とうふ店 TM
设置文本大小	<code>#text(size: 24pt)[一斤鸭梨]</code>	一斤鸭梨
设置文本颜色	<code>#text(fill: blue)[蓝色鸭梨]</code>	蓝色鸭梨
设置字体	<code>#text(font: "Microsoft YaHei") [板正鸭梨]</code>	板正鸭梨

分类：脚本声明

「名称/术语」	语法	渲染结果
进入脚本模式	<code>#1</code>	1
代码块	<code>#{"a"; "b"}</code>	ab
内容块	<code>#[内容块]</code>	内容块
空字面量	<code>#none</code>	
假（布尔值）	<code>#false</code>	false
真（布尔值）	<code>#true</code>	true
整数字面量	<code>#(-1), #(0), #(1)</code>	-1, 0, 1
进制数字面量	<code>#(-0xdeadbeef), #(-0o644), #(-0b1001)</code>	-3735928559, -420, -9
浮点数字面量	<code>#(0.001), #(.1), #(2.)</code>	0.001, 0.1, 2
指数表示法	<code>#(1e2), #(1.926e3), #(-1e-3)</code>	100, 1926, -0.001
字符串字面量	<code>#"Hello world!!"</code>	Hello world!!
字符串转义序列	<code>#"\""</code>	"
字符串的 Unicode 转义序列	<code>#"\\u{9999}"</code>	香
数组字面量	<code>#{1, "0v0", [一段内容]}</code>	(1, "0v0", [一段内容])

字典字面量	<code>#(neko-mimi: 2, "utterance": "喵喵")</code>	<code>(neko-mimi: 2, utterance: "喵喵")</code>
空数组	<code>#()</code>	<code>()</code>
空字典	<code>#(:)</code>	<code>(:)</code>
被括号包裹的空数组	<code>#(())</code>	<code>()</code>
含有一个元素的数组	<code>#(1,)</code>	<code>(1,)</code>
变量声明	<code>#let x = 1</code>	
函数声明	<code>#let f(x) = x * 2</code>	
函数闭包	<code>#let f = (x, y) => x + y</code>	
具名参数声明	<code>#let g(named: none) = named</code>	
含变参函数	<code>#let g(..args) = args.pos().join([,])</code>	
数组解构赋值	<code>#let (one, hello-world) = (1, "Hello, World")</code>	
数组解构赋值情形二	<code>#let (_, second, ..) = (1, "Hello, World", []); #second</code>	Hello, World
字典解构赋值	<code>#let (neko-mimi: mimi) = (neko-mimi: 2); #mimi</code>	2
数组内容重映射	<code>#let (a, b, c) = (1, 2, 3) #let (b, c, a) = (a, b, c) #a, #b, #c</code>	3, 1, 2
数组内容交换	<code>#let (a, b) = (1, 2) #((a, b) = (b, a)) #a, #b</code>	2, 1
占位符 (let _ = ..)	<code>#let last-two(t) = { let _ = t.pop() t.pop() } #last-two((1, 2, 3, 4))</code>	3

分类：脚本语句

「名称/术语」	语法	渲染结果
if 语句	<code>#if true { 1 }, #if false { 1 } else { 0 }</code>	1, 0

串联 if 语句	<pre>#if false { 0 } else if true { 1 }, #if false { 2 } else if false { 1 } else { 0 }</pre>	1, 0
while 语句	<pre>#{ let i = 0; while i < 10 { (i * 2,) i += 1; } }</pre>	(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
for 语句	<pre>#for i in range(10) { (i * 2,) }</pre>	(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
for 语句解构赋值	<pre>#for (特色, 这个) in (neko-mimi: 2) [猫猫的 #特色 是 #这个\]</pre>	猫猫的 neko-mimi 是 2
break 语句	<pre>#for i in range(10) { (i,); (i + 1926,); break }</pre>	(0, 1926)
continue 语句	<pre>#for i in range(10) { if calc.even(i) { continue } (i,) }</pre>	(1, 3, 5, 7, 9)
return 语句	<pre>#let never(..args) = return #type(never(1, 2))</pre>	type(none)
include 语句	<pre>#include "other-file.typ"</pre>	一段文本 另一段文本

分类：脚本样式

「名称/术语」	语法	渲染结果
set 语句	<pre>#set text(size: 24pt) 四斤鸭梨</pre>	四斤鸭梨
作用域	<pre>两只 #[兔 #set text(fill: rgb("#ffd1dc").darken(15%)) #[兔白 #set text(fill: orange) 又白], 真可爱]</pre>	两只兔兔白又白, 真可爱
set if 语句	<pre>#let is-dark-theme = true #set rect(fill: black) if is- dark-theme #set text(fill: white) if is- dark-theme #rect([wink!])</pre>	wink!
show set 语句	<pre>#show: set text(fill: blue) wink!</pre>	wink!

show 语句	<pre>#show raw: it => it.lines.at(1) 获取代码片段第二行内容: ```typ #{ set text(fill: true) } ```</pre>	获取代码片段第二行内容: <pre>set text(fill: true)</pre>
文本选择器	<pre>#show "cpp": strong(emph(box("C++"))) 在古代, cpp 是一门常用语言。</pre>	在古代, C++是一门常用语言。
正则文本选择器	<pre>#show regex("[。]+"): it => { set text(font: "KaiTi") highlight(it, fill: yellow) } “无名, 万物之始也; 有名, 万物之母也。”</pre>	“无名, 万物之始也; 有名, 万物之母也。”
标签选择器	<pre>#show <一整段话>: set text(fill: blue) #[\$lambda\$ 语言是世界上最好的语言。] <一整段话> 另一段话。</pre>	<pre>λ语言是世界上最好的语言。 另一段话。</pre>
选择器表达式	<pre>#show heading.where(level: 2): set text(fill: blue) = 一级标题 == 二级标题</pre>	<pre>一级标题 二级标题</pre>
获取位置	<pre>#locate(loc => loc.position())</pre>	(page: 110, x: 377.8pt, y: 423.23pt)
检测当前页面是否为偶数页 (位置相关计算)	<pre>#locate(loc => [页码是偶数: #calc.even(loc.page())])</pre>	页码是偶数: true
查询文档内容	<pre>#locate(loc => query(<ref-internal-link>, loc).at(0).body)</pre>	某个标题
声明全局状态	<pre>#state("my-state", 1)</pre>	<pre>state("my-state", 1)</pre>

分类：脚本表达式

「名称/术语」	语法	渲染结果
函数调用	<pre>#calc.pow(4, 3)</pre>	64
函数调用传递内容参数	<pre>#emph[emphasis]</pre>	<i>emphasis</i>
成员访问	<pre>#`OvO`.text</pre>	OvO
方法调用	<pre>#"Hello World".split(" ")</pre>	("Hello", "World")
字典成员访问	<pre>#let cat = (neko-mimi: 2) #cat.neko-mimi</pre>	2

内容成员访问	<code>#'OvO'.text</code>	OvO
代码表示的自省函数	<code>#repr[一段文本]</code>	sequence([], [一段文本], [])
类型的自省函数	<code>#type[一段文本]</code>	content
求值函数	<code>#type(eval("1"))</code>	integer
求值函数（标记模式）	<code>#eval("== 一个标题", mode: "markup")</code>	一个标题
判断数组内容	<code>#let pol = (1, "OvO", []) #(1 in pol)</code>	true
判断数组内容不在	<code>#let pol = (1, "OvO", []) #([另一段内容] not in pol)</code>	true
判断字典内容	<code>#let cat = (neko-mimi: 2) #("neko-mimi" in cat)</code>	true
逻辑比较表达式	<code> #(1 < 0), #(1 >= 2), #(1 == 2), #(1 != 2)</code>	false, false, false, true
逻辑运算表达式	<code> #(not false), #(false or true), #(true and false)</code>	true, true, false
取正运算	<code> #(+1), #(+0), #(1), #(++1)</code>	1, 0, 1, 1
取负运算	<code> #(-1), #(-0), #(--1), #(-+-1)</code>	-1, 0, 1, 1
算术运算表达式	<code> #(1 + 1), #(1 + -1), #(1 - 1), #(1 - -1)</code>	2, 0, 0, 2
赋值表达式	<code> #let a = 1; #repr(a = 10), #a, #repr(a += 2), #a</code>	none, 10, none, 12
字符串连接	<code> #("a" + "b")</code>	ab
重复连接字符串	<code> #("a" * 4), #(4 * "ab")</code>	aaaa, abababab
字符串比较	<code> #("a" == "b"), #("a" != "b"), #("a" < "ab"), #("a" >= "a")</code>	false, true, true, true
整数转浮点数	<code> #float(1), #(type(float(1)))</code>	1, float
布尔值转整数	<code> #int(true), #(type(int(true)))</code>	1, integer
浮点数转整数	<code> #int(1), #(type(int(1)))</code>	1, integer
十进制字符串转整数	<code> #int("1"), #(type(int("1")))</code>	1, integer

十六进制/八进制/二进制字符串转整数	<pre>#let safe-to-int(x) = { let res = eval(x) assert(type(res) == int, message: "should be integer") res } #safe-to-int("0xf"), #(type(safe-to-int("0xf"))) \ #safe-to-int("0o755"), #(type(safe-to-int("0o755"))) \ #safe-to-int("0b1011"), #(type(safe-to-int("0b1011"))) \</pre>	15, integer 493, integer 11, integer
数字转字符串	<pre>#repr(str(1)), #repr(str(.5))</pre>	"1", "0.5"
整数转十六进制字符串	<pre>#str(501, base:16), #str(0xdeadbeef, base:36)</pre>	1f5, 1ps9wxb
布尔值转字符串	<pre>#repr(false)</pre>	false
数字转布尔值	<pre>#let to-bool(x) = x != 0 #repr(to-bool(0)), #repr(to-bool(1))</pre>	false, true

参考：函数表（字典序）

分类：函数

函数	名称	描述
accent	Accent	Attaches an accent to a base.
align	Align	Aligns content horizontally and vertically.
assert	Assert	Ensures that a condition is fulfilled.
bibliography	Bibliography	A bibliography / reference listing.
block	Block	A block-level container.
box	Box	An inline-level container that sizes content.
cancel	Cancel	Displays a diagonal line over a part of an equation.
cases	Cases	A case distinction.
cbor	CBOR	Reads structured data from a CBOR file.
circle	Circle	A circle with optional content.
cite	Cite	Cite a work from the bibliography.
class	Class	Forced use of a certain math class.
colbreak	Column Break	Forces a column break.
columns	Columns	Separates a region into multiple equally sized columns.
csv	CSV	Reads structured data from a CSV file.
document	Document	The root element of a document and its metadata.
ellipse	Ellipse	An ellipse with optional content.
emph	Emphasis	Emphasizes content by toggling italics.
enum	Numbered List	A numbered list.
equation	Equation	A mathematical equation.
eval	Evaluate	Evaluates a string as Typst code.
figure	Figure	A figure with an optional caption.
footnote	Footnote	A footnote.
frac	Fraction	A mathematical fraction.
grid	Grid	Arranges content in a grid.
h	Spacing (H)	Inserts horizontal spacing into a paragraph.
heading	Heading	A section heading.
here	Here	Provides the current location in the document.
hide	Hide	Hides content without affecting layout.
highlight	Highlight	Highlights text with a background color.
image	Image	A raster or vector graphic.
json	JSON	Reads structured data from a JSON file.
layout	Layout	Provides access to the current outer container's (or page's, if none) size
line	Line	A line from one point to another.

linebreak	Line Break	Inserts a line break.
link	Link	Links to a URL or a location in the document.
list	Bullet List	A bullet list.
locate	Locate	Determines the location of an element in the document.
lorem	Lorem	Creates blind text.
lower	Lowercase	Converts a string or content to lowercase.
mat	Matrix	A matrix.
measure	Measure	Measures the layouted size of content.
metadata	Metadata	Exposes a value to the query system without producing visible content.
move	Move	Moves content without affecting layout.
numbering	Numbering	Applies a numbering to a sequence of numbers.
op	Text Operator	A text operator in an equation.
outline	Outline	A table of contents, figures, or other elements.
overline	Overline	Adds a line over text.
pad	Padding	Adds spacing around content.
page	Page	Layouts its child onto one or multiple pages.
pagebreak	Page Break	A manual page break.
panic	Panic	Fails with an error.
par	Paragraph	Arranges text, spacing and inline-level elements into a paragraph.
parbreak	Paragraph Break	A paragraph break.
path	Path	A path through a list of points, connected by Bezier curves.
place	Place	Places content at an absolute position.
polygon	Polygon	A closed polygon.
primes	Primes	Grouped primes.
query	Query	Finds elements in the document.
quote	Quote	Displays a quote alongside an optional attribution.
raw	Raw Text / Code	Raw text with optional syntax highlighting.
read	Read	Reads plain text or data from a file.
rect	Rectangle	A rectangle with optional content.
ref	Reference	A reference to a label or bibliography.
repeat	Repeat	Repeats content to the available space.
repr	Representation	Returns the string representation of a value.
rotate	Rotate	Rotates content without affecting layout.
scale	Scale	Scales content without affecting layout.
smallcaps	Small Capitals	Displays text in small capitals.
smartquote	Smartquote	A language-aware quote that reacts to its context.
square	Square	A square with optional content.

<code>stack</code>	Stack	Arranges content and spacing horizontally or vertically.
<code>strike</code>	Strikethrough	Strikes through text.
<code>strong</code>	Strong Emphasis	Strongly emphasizes content by increasing the font weight.
<code>style</code>	Style	Provides access to active styles.
<code>sub</code>	Subscript	Renders text in subscript.
<code>super</code>	Superscript	Renders text in superscript.
<code>table</code>	Table	A table of items.
<code>terms</code>	Term List	A list of terms and their descriptions.
<code>text</code>	Text	Customizes the look and layout of text in a variety of ways.
<code>toml</code>	TOML	Reads structured data from a TOML file.
<code>underline</code>	Underline	Underlines text.
<code>upper</code>	Uppercase	Converts a string or content to uppercase.
<code>v</code>	Spacing (V)	Inserts vertical spacing into a flow of blocks.
<code>vec</code>	Vector	A column vector.
<code>xml</code>	XML	Reads structured data from an XML file.
<code>yaml</code>	YAML	Reads structured data from a YAML file.

分类：方法

方法	名称	描述
<code>alignment.axis</code>	Axis	The axis this alignment belongs to.
<code>alignment.inv</code>	Inverse	The inverse alignment.
<code>angle.deg</code>	Degrees	Converts this angle to degrees.
<code>angle.rad</code>	Radians	Converts this angle to radians.
<code>arguments.named</code>	Named	Returns the captured named arguments as a dictionary.
<code>arguments.pos</code>	Positional	Returns the captured positional arguments as an array.
<code>array.all</code>	All	Whether the given function returns <code>{true}</code> for all items in the array.
<code>array.any</code>	Any	Whether the given function returns <code>{true}</code> for any item in the array.
<code>array.at</code>	At	Returns the item at the specified index in the array. May be used on the
<code>array.contains</code>	Contains	Whether the array contains the specified value.
<code>array.dedup</code>	Deduplicate	Deduplicates all items in the array.
<code>array.enumerate</code>	Enumerate	Returns a new array with the values alongside their indices.
<code>array.filter</code>	Filter	Produces a new array with only the items from the original one for which

<code>array.find</code>	Find	Searches for an item for which the given function returns <code>{true}</code> and
<code>array.first</code>	First	Returns the first item in the array. May be used on the left-hand side
<code>array.flatten</code>	Flatten	Combine all nested arrays into a single flat one.
<code>array.fold</code>	Fold	Folds all items into a single value using an accumulator function.
<code>array.insert</code>	Insert	Inserts a value into the array at the specified index. Fails with an
<code>array.intersperse</code>	Intersperse	Returns an array with a copy of the separator value placed between
<code>array.join</code>	Join	Combine all items in the array into one.
<code>array.last</code>	Last	Returns the last item in the array. May be used on the left-hand side of
<code>array.len</code>	Length	The number of values in the array.
<code>array.map</code>	Map	Produces a new array in which all items from the original one were
<code>array.pop</code>	Pop	Removes the last item from the array and returns it. Fails with an error
<code>array.position</code>	Position	Searches for an item for which the given function returns <code>{true}</code> and
<code>array.product</code>	Product	Calculates the product all items (works for all types that can be
<code>array.push</code>	Push	Adds a value to the end of the array.
<code>array.range</code>	Range	Create an array consisting of a sequence of numbers.
<code>array.remove</code>	Remove	Removes the value at the specified index from the array and return it.
<code>array.rev</code>	Reverse	Return a new array with the same items, but in reverse order.
<code>array.slice</code>	Slice	Extracts a subslice of the array. Fails with an error if the start or
<code>array.sorted</code>	Sorted	Return a sorted version of this array, optionally by a given key
<code>array.split</code>	Split	Split the array at occurrences of the specified value.
<code>array.sum</code>	Sum	Sums all items (works for all types that can be added).
<code>array.zip</code>	Zip	Zips the array with other arrays.
<code>bytes.at</code>	At	Returns the byte at the specified index. Returns the default value if
<code>bytes.len</code>	Length	The length in bytes.
<code>bytes.slice</code>	Slice	Extracts a subslice of the bytes. Fails with an error if the start or
<code>color.cmyk</code>	CMYK	Create a CMYK color.

<code>color.components</code>	Components	Extracts the components of this color.
<code>color.darken</code>	Darken	Darkens a color by a given factor.
<code>color.desaturate</code>	Desaturate	Decreases the saturation of a color by a given factor.
<code>color.hsl</code>	HSL	Create an HSL color.
<code>color.hsv</code>	HSV	Create an HSV color.
<code>color.lighten</code>	Lighten	Lightens a color by a given factor.
<code>color.linear-rgb</code>	Linear RGB	Create an RGB(A) color with linear luma.
<code>color.luma</code>	Luma	Create a grayscale color.
<code>color.mix</code>	Mix	Create a color by mixing two or more colors.
<code>color.negate</code>	Negate	Produces the complementary color using a provided color space.
<code>color.oklab</code>	Oklab	Create an Oklab color.
<code>color.oklch</code>	Oklch	Create an Oklch color.
<code>color.rgb</code>	RGB	Create an RGB(A) color.
<code>color.rotate</code>	Rotate	Rotates the hue of the color by a given angle.
<code>color.saturate</code>	Saturate	Increases the saturation of a color by a given factor.
<code>color.space</code>	Space	Returns the constructor function for this color's space:
<code>color.to-hex</code>	To Hex	Returns the color's RGB(A) hex representation (such as <code>#ffaa32</code> or
<code>content.at</code>	At	Access the specified field on the content. Returns the default value if
<code>content.fields</code>	Fields	Returns the fields of this content.
<code>content.func</code>	Func	The content's element function. This function can be used to create the element
<code>content.has</code>	Has	Whether the content has the specified field.
<code>content.location</code>	Location	The location of the content. This is only available on content returned
<code>counter.at</code>	At	Retrieves the value of the counter at the given location. Always returns
<code>counter.display</code>	Display	Displays the current value of the counter with a numbering and returns
<code>counter.final</code>	Final	Retrieves the value of the counter at the end of the document. Always
<code>counter.get</code>	Get	Retrieves the value of the counter at the current location. Always
<code>counter.step</code>	Step	Increases the value of the counter by one.
<code>counter.update</code>	Update	Updates the value of the counter.
<code>datetime.day</code>	Day	The day if it was specified, or <code>{none}</code> for times without a date.
<code>datetime.display</code>	Display	Displays the datetime in a specified format.
<code>datetime.hour</code>	Hour	The hour if it was specified, or <code>{none}</code> for dates without a time.

<code>datetime.minute</code>	Minute	The minute if it was specified, or <code>{none}</code> for dates without a time.
<code>datetime.month</code>	Month	The month if it was specified, or <code>{none}</code> for times without a date.
<code>datetime.ordinal</code>	Ordinal	The ordinal (day of the year), or <code>{none}</code> for times without a date.
<code>datetime.second</code>	Second	The second if it was specified, or <code>{none}</code> for dates without a time.
<code>datetime.today</code>	Today	Returns the current date.
<code>datetime.weekday</code>	Weekday	The weekday (counting Monday as 1) or <code>{none}</code> for times without a date.
<code>datetime.year</code>	Year	The year if it was specified, or <code>{none}</code> for times without a date.
<code>dictionary.at</code>	At	Returns the value associated with the specified key in the dictionary.
<code>dictionary.insert</code>	Insert	Inserts a new pair into the dictionary. If the dictionary already
<code>dictionary.keys</code>	Keys	Returns the keys of the dictionary as an array in insertion order.
<code>dictionary.len</code>	Length	The number of pairs in the dictionary.
<code>dictionary.pairs</code>	Pairs	Returns the keys and values of the dictionary as an array of pairs. Each
<code>dictionary.remove</code>	Remove	Removes a pair from the dictionary by key and return the value.
<code>dictionary.values</code>	Values	Returns the values of the dictionary as an array in insertion order.
<code>direction.axis</code>	Axis	The axis this direction belongs to, either <code>{"horizontal"}</code> or
<code>direction.end</code>	End	The end point of this direction, as an alignment.
<code>direction.inv</code>	Inverse	The inverse direction.
<code>direction.start</code>	Start	The start point of this direction, as an alignment.
<code>duration.days</code>	Days	The duration expressed in days.
<code>duration.hours</code>	Hours	The duration expressed in hours.
<code>duration.minutes</code>	Minutes	The duration expressed in minutes.
<code>duration.seconds</code>	Seconds	The duration expressed in seconds.
<code>duration.weeks</code>	Weeks	The duration expressed in weeks.
<code>float.is-infinite</code>	Is Infinite	Checks if a float is infinite.
<code>float.is-nan</code>	Is Nan	Checks if a float is not a number.
<code>float.signum</code>	Signum	Calculates the sign of a floating point number.
<code>function.where</code>	Where	Returns a selector that filters for elements belonging to this function
<code>function.with</code>	With	Returns a new function that has the given arguments pre-applied.
<code>gradient.angle</code>	Angle	Returns the angle of this gradient.

<code>gradient.conic</code>	Conic	Creates a new conic gradient, in which colors change radially around a
<code>gradient.kind</code>	Kind	Returns the kind of this gradient.
<code>gradient.linear</code>	Linear Gradient	Creates a new linear gradient, in which colors transition along a
<code>gradient.radial</code>	Radial	Creates a new radial gradient, in which colors radiate away from an
<code>gradient.relative</code>	Relative	Returns the relative placement of this gradient.
<code>gradient.repeat</code>	Repeat	Repeats this gradient a given number of times, optionally mirroring it
<code>gradient.sample</code>	Sample	Sample the gradient at a given position.
<code>gradient.samples</code>	Samples	Samples the gradient at multiple positions at once and returns the
<code>gradient.sharp</code>	Sharp	Creates a sharp version of this gradient.
<code>gradient.space</code>	Space	Returns the mixing space of this gradient.
<code>gradient.stops</code>	Stops	Returns the stops of this gradient.
<code>int.bit-and</code>	Bitwise AND	Calculates the bitwise AND between two integers.
<code>int.bit-lshift</code>	Bitwise Left Shift	Shifts the operand' s bits to the left by the specified amount.
<code>int.bit-not</code>	Bitwise NOT	Calculates the bitwise NOT of an integer.
<code>int.bit-or</code>	Bitwise OR	Calculates the bitwise OR between two integers.
<code>int.bit-rshift</code>	Bitwise Right Shift	Shifts the operand' s bits to the right by the specified amount.
<code>int.bit-xor</code>	Bitwise XOR	Calculates the bitwise XOR between two integers.
<code>int.signum</code>	Signum	Calculates the sign of an integer.
<code>length.cm</code>	Centimeters	Converts this length to centimeters.
<code>length.inches</code>	Inches	Converts this length to inches.
<code>length.mm</code>	Millimeters	Converts this length to millimeters.
<code>length.pt</code>	Points	Converts this length to points.
<code>length.to-absolute</code>	To Absolute	Resolve this length to an absolute length.
<code>location.page</code>	Page	Returns the page number for this location.
<code>location.page-numbering</code>	Page Numbering	Returns the page numbering pattern of the page at this location. This
<code>location.position</code>	Position	Returns a dictionary with the page number and the x, y position for this
<code>selector.after</code>	After	Returns a modified selector that will only match elements that occur
<code>selector.and</code>	And	Selects all elements that match this and all of the the other selectors.
<code>selector.before</code>	Before	Returns a modified selector that will only match elements that occur
<code>selector.or</code>	Or	Selects all elements that match this or any of the other selectors.

<code>state.at</code>	At	Retrieves the value of the state at the given selector' s unique match.
<code>state.display</code>	Display	Deprection planned: Use <code>get</code> instead.
<code>state.final</code>	Final	Retrieves the value of the state at the end of the document.
<code>state.get</code>	Get	Retrieves the value of the state at the current location.
<code>state.update</code>	Update	Update the value of the state.
<code>str.at</code>	At	Extracts the first grapheme cluster after the specified index. Returns
<code>str.clusters</code>	Clusters	Returns the grapheme clusters of the string as an array of substrings.
<code>str.codepoints</code>	Codepoints	Returns the Unicode codepoints of the string as an array of substrings.
<code>str.contains</code>	Contains	Whether the string contains the specified pattern.
<code>str.ends-with</code>	Ends With	Whether the string ends with the specified pattern.
<code>str.find</code>	Find	Searches for the specified pattern in the string and returns the first
<code>str.first</code>	First	Extracts the first grapheme cluster of the string.
<code>str.from-unicode</code>	From Unicode	Converts a unicode code point into its corresponding string.
<code>str.last</code>	Last	Extracts the last grapheme cluster of the string.
<code>str.len</code>	Length	The length of the string in UTF-8 encoded bytes.
<code>str.match</code>	Match	Searches for the specified pattern in the string and returns a
<code>str.matches</code>	Matches	Searches for the specified pattern in the string and returns an array of
<code>str.position</code>	Position	Searches for the specified pattern in the string and returns the index
<code>str.replace</code>	Replace	Replace at most count occurrences of the given pattern with a
<code>str.rev</code>	Reverse	Reverse the string.
<code>str.slice</code>	Slice	Extracts a substring of the string.
<code>str.split</code>	Split	Splits a string at matches of a specified pattern and returns an array
<code>str.starts-with</code>	Starts With	Whether the string starts with the specified pattern.
<code>str.to-unicode</code>	To Unicode	Converts a character into its corresponding code point.
<code>str.trim</code>	Trim	Removes matches of a pattern from one or both sides of the string, once or
<code>version.at</code>	At	Retrieves a component of a version.

参考：常用数学符号

推荐阅读：

1. [本科生 Typst 数学英文版](#)，适用于 Typst 0.11.1
2. [本科生 Typst 数学中文版](#)，适用于 Typst 0.9.0

以下表格列出了**数学模式**中的常用符号。

\hat{a}	hat(a)	\check{a}	caron(a)	\tilde{a}	tilde(a)	\grave{a}	grave(a)
\dot{a}	dot(a)	\ddot{a}	dot.double(a)	\bar{a}	macron(a)	\vec{a}	arrow(a)
\acute{a}	acute(a)	\breve{a}	breve(a)				

表 1 数学模式重音符号

α	alpha	θ	theta	o	o	v	upsilon
β	beta	ϑ	theta.alt	π	pi	φ	phi
γ	gamma	ι	iota	ϖ	pi.alt	ϕ	phi
δ	delta	κ	kappa	ρ	rho	χ	chi
ϵ	epsilon.alt	λ	lambda	ϱ	rho.alt	ψ	psi
ε	epsilon	μ	mu	σ	sigma	ω	omega
ζ	zeta	ν	nu	ς	sigma.alt		
η	eta	ξ	xi	τ	tau		
Γ	Gamma	Λ	Lambda	Σ	Sigma	Ψ	Psi
Δ	Delta	Ξ	Xi	Υ	Upsilon	Ω	Omega
Θ	Theta	Π	Pi	Φ	Phi		

表 2 希腊字母

$<$	$<, \text{lt}$	$>$	$>, \text{gt}$	$=$	$=$
\leq	$\leq, \text{lt.eq}$	\geq	$\geq, \text{gt.eq}$	\equiv	equiv
\ll	$\ll, \text{lt.double}$	\gg	$\gg, \text{gt.double}$		
\prec	prec	\succ	succ	\sim	tilde
\preceq	prec.eq	\succeq	succ.eq	\simeq	tilde.eq
\subset	subset	\supset	supset	\approx	approx
\subseteq	subset.eq	\supseteq	supset.eq	\cong	tilde.equiv
\sqsubset	subset.sq	\sqsupset	supset.sq	\boxtimes	join
\sqsubseteq	subset.eq.sq	\sqsupseteq	supset.eq.sq		
\in	in	\ni	in.rev	\propto	prop
\vdash	tack.r	\dashv	tack.l	\models	tack.r.double
$ $	divides	\parallel	parallel	\perp	tack.t
$:$	$:$	\notin	in.not	\neq	$\text{!}, \text{eq.not}$

表 3 二元关系

$+$	$+, \text{plus}$	$-$	$-, \text{minus}$		
\pm	plus.minus	\mp	minus.plus	\triangleleft	lt.tri
\cdot	dot	\div	div	\triangleright	gt.tri
\times	times	\backslash	without	\star	star
\cup	union	\cap	sect	$*$	$*$
\sqcup	union.sq	\sqcap	sect.sq	\circ	$\text{circle.stroked.tiny}$
\vee	or	\wedge	and	\bullet	bullet
\oplus	xor	\ominus	minus.circle	\odot	dot.circle
\oplus	union.plus	\otimes	times.circle	\bigcirc	circle.big
\amalg	product.co	\triangle	$\text{triangle.stroked.t}$	∇	$\text{triangle.stroked.b}$
\dagger	dagger	\triangleleft	lt.tri	\triangleright	gt.tri
\ddagger	dagger.double	\trianglelefteq	lt.tri.eq	\trianglerighteq	gt.tri.eq
$\}$	wreath				

表 4 二元运算符

Σ	sum	\cup	union.big	\vee	or.big
\prod	product	\cap	sect.big	\wedge	and.big
\coprod	product.co	\sqcup	union.sq.big	$\dot{+}$	union.plus.big
\int	integral	\oint	integral.cont	\odot	dot.circle.big
\oplus	plus.circle.big	\otimes	times.circle.big		

表 5 「大」运算符

\leftarrow	arrow.l	\longleftarrow	arrow.l.long
\rightarrow	arrow.r	\longrightarrow	arrow.r.long
\leftrightarrow	arrow.l.r	\longleftrightarrow	arrow.l.r.long
\Leftrightarrow	arrow.l.double	\Longleftrightarrow	arrow.l.double.long
\Rightarrow	arrow.r.double	\Longrightarrow	arrow.r.double.long
\Leftrightarrow	arrow.l.r.double	\Longleftrightarrow	arrow.l.r.double.long
\mapsto	arrow.r.bar	\longmapsto	arrow.r.long.bar
\hookrightarrow	arrow.l.hook	\hookrightarrow	arrow.r.hook
\lhd	harpoon.lt	\rhd	harpoon.rt
\lhd	harpoon.lb	\rhd	harpoon.rb
\Leftrightarrow	harpoons.ltrb	\uparrow	arrow.t
\downarrow	arrow.b	\updownarrow	arrow.t.b
\Uparrow	arrow.t.double	\Downarrow	arrow.b.double
\Updownarrow	arrow.t.b.double	\nearrow	arrow.tr
\searrow	arrow.br	\swarrow	arrow.bl
\nwarrow	arrow.tl	\rightsquigarrow	arrow.r.squiggly

表 6 箭头

...	dots	...	dots.c	:	dots.v
⋮	dots.down	ℏ	planck.reduce	⋮	dotless.i
<i>j</i>	dotless.j	ℓ	ell	ℜ	Re
ℑ	Im	ℵ	aleph	∀	forall
∃	exists	Ω	ohm.inv	∂	diff
'	', prime	∅	emptyset	∞	infinity
∇	nabla	△	triangle.stroked.t	□	square.stroked
◇	diamond.stroked	⊥	bot	⊤	top
∠	angle	♣	suit.club	♠	suit.spade
¬	not				

表 7 其他符号



Part.7 基础参考



参考：基本类型

类型转换

整数转浮点数：

```
#float(1), #(type(float(1)))
```

1, float

布尔值转整数：

```
#int(false), #(type(int(false))) \
#int(true), #(type(int(true)))
```

0, integer
1, integer

浮点数转整数：

```
#int(1), #(type(int(1)))
```

1, integer

该方法是就近取整，并有精度损失（根据规范，超出精度范围时，如何选择较近的值舍入是「与实现相关」）：

```
#int(1.5), #int(1.99),  
// 超出浮点精度范围会就近舍入再转换成整数  
#int(1.9999999999999999)
```

1, 1, 2

为了向下或向上取整，你可以同时使用 `calc.floor` 或 `calc.ceil` 函数（有精度损失）：

```
#int(calc.floor(1.9)),  
#int(calc.ceil(1.9))
```

1, 2

十进制字符串转整数：

```
#int("1"), #(type(int("1")))
```

1, integer

十六进制/八进制/二进制字符串转整数：

```
#let safe-to-int(x) = {  
  let res = eval(x)  
  assert(type(res) == int, message:  
    "should be integer")  
  res  
}  
#safe-to-int("0xf"), #(type(safe-to-int("0xf")) \)  
#safe-to-int("0o755"), #(type(safe-to-int("0o755")) \)  
#safe-to-int("0b1011"), #(type(safe-to-int("0b1011")) \)
```

15, integer
493, integer
11, integer

注意：`assert(type(res) == int)`是必须的，否则是不安全的。

数字转字符串：

```
#repr(str(1)), #(type(str(1)))  
#repr(str(.5)), #(type(str(.5)))
```

"1", string "0.5", string

整数转 *N* 进制字符串：

```
#str(501, base:16), #str(0xdeadbeef,  
base:36)
```

1f5, 1ps9wxb

布尔值转字符串：

```
#repr(false), #(type(repr(false)))
```

```
false, string
```

数字转布尔值:

```
#let to-bool(x) = x != 0
#repr(to-bool(0)), #(type(to-bool(0))) \
#repr(to-bool(1)), #(type(to-bool(1)))
```

```
false, boolean
true, boolean
```

Typst 的基本类型设计大量参考 Python 和 Rust。Typst 基本类型的特点是半纯的 API 设计。其基本类型的方法倾向于保持纯度，但如果影响到了使用的方便性，Typst 会适当牺牲纯度。

如下所示，`array.push` 就是带有副作用的：

```
#let t = (1, 2)
#t.push(3)
#t
```

```
(1, 2, 3)
```

布尔类型

type:bool

Typst 的布尔值是只有两个实例的类型。

```
#false, #true
```

```
false, true
```

布尔值一般用来表示逻辑的确否。

```
 #(1 < 2), #(1 > 2)
```

```
true, false
```

整数

type:int

Typst 的整数是 64 位宽度的有符号整数。

有符号整数的意思是，你可以使用正整数、负整数或零作为整数对象的内容。

正数: `#1`; 负数: `#(-1)`; 零: `#0`。

正数: 1; 负数: -1; 零: 0。

64 位宽度整数的意思是，Typst 允许你使用的整数是有限大的。正数、负数与整数均分 2^{64} 个数字。因此：

- 最大的正整数是 $2^{63} - 1$ 。

```
#int(9223372036854775807)
```

```
9223372036854775807
```

- 最小的负整数是 -2^{63} 。(todo: typst v0.12.0 这里有一个 bug)

```
#int(-9223372036854775808)
```

```
-9223372036854775808
```

Typst 还允许你使用其他「表示法」(representation) 表示整数。借鉴了其他编程语言，使用 `0x`、`0o` 和 `0b` 分别可以指定十六进制、八进制和二进制整数：

```
#0xffff, #0o755, #0b101010
```

```
65535, 493, 42
```

```
fn int(value: bool | int | float | str) -> int
```

func:int

除了「字面量」(literal) 构造整数，你还能使用构造器 `int` 将其他值转换为整数。

- 将布尔值转换为0或1：

```
#int(false), #int(true)
```

0, 1

- 将浮点数向零取整为整数:

```
#int(-1.1), #int(1.1), #int(-0.1),  
#int(-1.9)
```

-1, 1, 0, -1

- 将字符串依照十进制表示转换为整数:

```
#int("42"), #int("0"), #int("-17")
```

42, 0, -17

你可以使用各种操作符进行整数运算:

```
#(1 + 2) \  
#(2 - 5) \  
#(3 + 4 < 8)
```

3
-3
true

详见《基本字面量、变量和简单函数》中的表达式类型。

整数类型上没有任何方法，但是 Typst 正计划将 `calc` 上的方法移动到整数类型和浮点数类型上。

浮点数

type:float

Typst 的浮点数是 64 位宽度的有符号浮点数。

有符号浮点数的意思是，你可以使用正数、负数或零作为浮点数对象的内容。**64 位宽度**浮点数的意思是，Typst 允许你使用的浮点数是有限大的。正数、负数与浮点数均分 2^{64} 个浮点数。

Typst 的浮点数遵守 [IEEE-754 浮点数标准](#)。因此:

- 最大的正数为 $f_{\max} = 2^{1023} \cdot (2 - 2^{-52})$ 。
- 最小的正数为 $f_{\min} = 2^{-1022} \cdot 2^{-52}$ 。
- 最大的负数为 $-f_{\min}$ 。
- 最小的负数为 $-f_{\max}$ 。

Typst 还允许你使用「指数表示法」(exponential notation) 表示浮点数。Typst 允许你使用 `{x}{e}{y}` 格式表示 $x \cdot 10^y$ 的浮点数:

```
#1e1, #1e-1, #(-1e1), #(-1e-1),  
#(1.1e1)
```

10, 0.1, -10, -0.1, 11

```
fn float(value: bool int float ratio str) -> float
```

func:float

除了「字面量」构造浮点数，你还能使用构造器 `float` 将其他值转换为浮点数。

- 将布尔值转换为0.0或1.0:

```
#float(false), #float(true)
```

0, 1

- 将整数转换成「最近」(nearest) 的浮点数:

```
#float(0), #float(-0),  
#float(10000000000)
```

0, 0, 10000000000

- 将百分比数字转换成「最近」的浮点数:

```
#float(100%), #float(1150%),  
#float(1%)
```

1, 11.5, 0.01

- 将字符串依照十进制或「指数表示法」表示转换为浮点数:


```
#float("42.2"), #float("0"),  
#float("-17"), #float("1e5")
```

42.2, 0, -17, 100000

你可以使用各种操作符进行浮点数运算，且浮点数允许与整数混合运算：

```
#(10 / 4) \  
#(3.5 + 4.6 < 8)
```

2.5
false

详见《基本字面量、变量和简单函数》中的表达式类型。

浮点数类型上没有任何方法，但是 Typst 正计划将 `calc` 上的方法移动到整数类型和浮点数类型上。

字符串

`type:str`

Typst 的字符串是「UTF-8 编码」（utf-8 encoding）的字节序列。

UTF-8 编码是一个广泛使用的变长编码规范。这意味着当你使用“abc123”字符串时，实际存储了这样的字节数据：

```
#let f(x) = range(x.len()).map(  
  i => x.at(i))  
#f(bytes("abc123"))
```

(97, 98, 99, 49, 50, 51)

这意味着当你使用“ふ店”字符串时，实际存储了这样的字节数据：

```
#let f(x) = range(x.len()).map(  
  i => x.at(i))  
#f(bytes("ふ店"))
```

(227, 129, 181, 229, 186, 151)

在绝大部分情况下，你不需要关心字符串的编码问题，因为在 Typst 脚本中只有「UTF-8 编码」的字符串。

但是这并不意味着你不需要了解「UTF-8 编码」。Typst 的字符串 API 与底层编码息息相关，因此你需要尽可能多地掌握与「UTF-8 编码」有关的知识。

在编码体系中，与 Typst 相关的主要有三层，它们在原始字节数据视角下的边界并不一样：

1. 字节表示：按字节寻址时，每个偏移量都索引到一个字节。
2. 「码位」（codepoint）表示：「UTF-8 编码」是变长编码。在「UTF-8 编码」中，很多「码位」都由多个字节组成。UTF-8 字符串中「码位」数据按照顺序存放，自然有些字节偏移量在「码位」寻址中不是合法的。
3. 「字素簇」（grapheme cluster）表示：「字素簇」就是人类视觉效果上的“最小字符单位”。在码位之上，Unicode 规范还规定了多个码位组成一个「字素簇」的情况。在这种情况下，自然有些码位（字节）偏移量在「字素簇」寻址中不是合法的。

下表说明了 `ä` 在字节表示下的合法偏移量。

数据	0x61	0xcc	0x83
字节	第 1 个字节	第 2 个字节	第 3 个字节
码位	英文字母 a	「波浪变音符号」（tilde diacritical marks）	非法
字素簇	波浪变音的英文字母 a	非法	非法

❗ 关于「字素簇」的定义，请参考《Unicode 规范：文本分段》。

字符串的字面量由双引号包裹。你可以在字符串字面量中使用与字符串相关的转义序列。

```
#"hello world!" \
#"\"hello\n world\"!" \
```

```
hello world!
"hello
world"!
```

详见《基本字面量、变量和简单函数》中关于字符串类型的描述。

operator: in
of str

你可以使用 in 关键字检查一个字符串是否是另一个字符串的子串：

```
#("fox" in "lazy fox"),
#(" fox" in "lazy fox"),
#("fox " in "lazy fox"),
#("dog" in "lazy fox")
```

```
true, true, false, false
```

in 关键字左侧值还可以是一个正则表达式类型，以检查「模式串」(pattern) 是否匹配字符串：

```
#(regex("\\d+") in "ten euros") \
#(regex("\\d+") in "10 euros")
```

```
false
true
```

in 关键字是以「码位」为粒度检查文本的：

```
#("\\u{0303}" in "ä")
```

```
true
```

```
fn str(value: int float str bytes label version type, base: int) -> str
```

func: str

除了「字面量」构造字符串，你还能使用构造器 str 将其他值转换为字符串。

- 将整数和浮点数转换为十进制格式字符串：

```
#str(0), #str(199),
#str(0.1), #str(9.1)
```

```
0, 199, 0.1, 9.1
```

- 将标签转换为其名称：

```
#str(<owo:this-label>)
```

```
owo:this-label
```

- 将字节数组依照「UTF-8 编码」编码：

```
#str(bytes((72, 0x69, 33)))
```

```
Hi!
```

param: base
of str

特别地，你可以使用 base 参数，将整数依照 $N(2 \leq N \leq 36)$ 进制格式转换为字符串：

```
#str(15, base: 16),
#str(14, base: 14),
#str(0xdeadbeef, base: 36)
```

```
f, 10, 1ps9wxb
```

借鉴 Python 和 JavaScript，Typst 不提供字符类型。相应的，仅具备单个「码位」的字符串就被视作一个「字符」(character)。也就是说，Typst 认定「字符」与「码位」等同。这是符合主流观念的。

- “a”、“我” 是字符。
- “ä” 不是字符。

```
fn str.to-unicode(character: str) -> int
```

method:to-unicode
of str

你可以使用 `str.to-unicode` 函数获得一个字符的「码位」表示:

```
#"a".to-unicode(),  
#"我".to-unicode()
```

97, 25105

❗ 显然, 不是所有的“字符”都可以应用 `to-unicode` 方法。

```
#"ã".to-unicode() /* 不能编译 */
```

这是因为视觉上为单个字符的 `ã` 是一个「字素簇」, 包含多个码位。

```
fn str.from-unicode(value: int) -> str
```

method:from-unicode
of str

你可以使用 `str.from-unicode` 函数将一个数字的「码位」表示转换成字符 (串):

```
#str.from-unicode(97),  
#str.from-unicode(25105)
```

a, 我

```
fn str.len() -> int
```

method:len
of str

你可以使用 `str.len` 函数获得一个字符串的字节表示的长度:

```
#"abc".len(),  
#"香風とうふ店".len(),  
#"ã".len()
```

3, 18, 3

你可能希望得到一个字符串的「码位宽度」(codepoint width), 这时你可以组合以下方法:

```
#"abc".codepoints().len(),  
#"香風とうふ店".codepoints().len(),  
#"ã".codepoints().len()
```

3, 6, 2

你可能希望得到一个字符串的「字素簇宽度」(grapheme cluster width), 这时你可以组合以下方法:

```
#"abc".clusters().len(),  
#"香風とうふ店".clusters().len(),  
#"ã".clusters().len()
```

3, 6, 1

```
fn str.first() -> str
```

method:first
of str

你可以使用 `str.first` 函数获得一个字符串的第一个「字素簇」:

```
#"Wee".first(),  
#"我 们 俩".first(),  
#"\\u{0061}\\u{0303}\\u{0061}".first()
```

W, 我, ã

```
fn str.last() -> str
```

method:last
of str

你可以使用 `str.last` 函数获得一个字符串的最后一个「字素簇」:

```
#"Wee".last(),  
#"我 们 俩".last(),  
#"\\u{0061}\\u{0303}\\u{0061}".last(),  
#"\\u{0061}\\u{0061}\\u{0303}".last()
```

e, 俩, a, ã

```
fn str.at(index: int, default: any) -> any
```

method:at
of str

你可以使用 `str.at` 函数获得一个字符串位于字节偏移量为 `offset` 的「字素簇」:

```
#"我 们 俩".at(0),
#"我 们 俩".at(3),
#"我 们 俩".at(4),
#"我 们 俩".at(8)
```

我, , 们, 俩

上例中, 第一个空格的字节偏移量为 3, “俩”的字节偏移量为 8。

```
fn str.slice(start: int, end: none int, count: int) -> str
```

method:slice of str 你可以使用 `str.slice` 函数截取字节偏移量从 `start` 到 `end` 的子串:

```
#repr("我 们 俩".slice(0, 11)),
#repr("我 们 俩".slice(4, 8)),
```

"我们俩", "们",

param:end of str.slice end 参数可以省略:

```
#repr("我 们 俩".slice(0)),
#repr("我 们 俩".slice(4)),
```

"我们俩", "们俩",

```
fn str.trim(pattern: none str regex, at: alignment, repeat: bool) -> str
```

method:trim of str 你可以使用 `str.trim` 去除字符串的首尾空白字符:

```
#repr(" A ".trim())
```

"A"

param:pattern of str.trim `str.trim` 可以指定 `pattern` 参数。

```
#repr("wwAww".trim("w"))
```

"A"

`pattern` 可以是 `none`、字符串或正则表达式。当不指定 `pattern` 时, Typst 会指定一个模式贪婪地去除首尾空格。

```
#repr(" A ".trim()),
#repr(" A ".trim(none)),
#repr("wwAww".trim("w")),
#repr("abAde".trim(regex("[a-z]+"))),
```

"A", "A", "A", "A",

❗ 当 `pattern` 参数为 `none` 时, Typst 直接调用 Rust 的 `trim_start` 或 `trim_end` 方法, 以默认获得更高的性能。

param:at of str.trim at 参数可以为 `start` 或 `end`, 分别指定则只清除起始位置或结束位置的字符。

```
#repr(" A ".trim(" ", at: start)),
#repr(" A ".trim(" ", at: end))
```

"A ", " A"

param:repeat of str.trim repeat 参数为 `false` 时, 不会重复运行 `pattern`; 否则会重复运行。默认 repeat 为 `true`。

```
#repr(" A ".trim(" ")),
#repr(" A ".trim(" ", repeat:
false))
```

"A", " A "

`pattern` 会在起始位置或结束位置执行至少一次。

```
#repr(" A ".trim(" ", at: start,
repeat: false)),
#repr("abAde".trim(
regex("[a-z]"), repeat: false)),
#repr("abAde".trim(
regex("[a-z]+"), repeat: false))
```

" A ", "bAd", "A"

```
fn str.split(pattern: none str regex) -> array
```

method:split of str 你可以使用 `str.split` 函数将一个字符串依照空白字符拆分:

```
#"我 们仨".split(),
```

```
("我", "们仨"),
```

```
fn str.rev() -> str
```

method: rev
of str

你可以使用 `str.rev` 函数将一个字符串逆转:

```
#"abcdedfg".rev()
```

```
gfdedcba
```

逆转时 Typst 会为你考虑「字素簇」。

```
#"äb".rev()
```

```
bä
```

`str.clusters`, `str.codepoints`

`str.contains`, `str.starts-with`, `str.ends-with`

`str.find`, `str.position`, `str.match`, `str.matches`, `str.replace`

字典

```
#let dict = (  
  name: "Typst",  
  born: 2019,  
)  
  
#dict.name \  
#(dict.launch = 20)  
#dict.len() \  
#dict.keys() \  
#dict.values() \  
#dict.at("born") \  
#dict.insert("city", "Berlin ")  
#("name" in dict)
```

```
Typst
```

```
3
```

```
("name", "born", "launch")
```

```
("Typst", 2019, 20)
```

```
2019
```

```
true
```

`dict.len`, `dict.at`

`dict.insert`, `dict.remove`

`dict.keys`, `dict.values`, `dict.pairs`

数组

```
#let values = (1, 7, 4, -3, 2)  
  
#values.at(0) \  
#(values.at(0) = 3)  
#values.at(-1) \  
#values.find(calc.even) \  
#values.filter(calc.odd) \  
#values.map(calc.abs) \  
#values.rev() \  
#(1, (2, 3)).flatten() \  
#(("A", "B", "C")  
  .join(", ", last: " and "))
```

```
1
```

```
2
```

```
4
```

```
(3, 7, -3)
```

```
(3, 7, 4, 3, 2)
```

```
(2, -3, 4, 7, 3)
```

```
(1, 2, 3)
```

```
A, B and C
```

`array.range`

`array.len`

`array.first`, `array.last`, `array.slice`

`array.push`, `array.pop`, `array.insert`, `array.remove`

`array.contains`, `array.find`, `array.position`, `array.filter`

`array.map`, `array.enumerate`, `array.zip`, `array.fold`, `array.sum`, `array.product`, `array.any`, `array.all`,
`array.flatten`, `array.rev`, `array.split`, `array.join`, `array.intersperse`, `array.sorted`, `array.dedup`

参考：内置类型

空类型

type

content

function

arguments

module

plugin

regex

label

reference

selector

version

参考：时间类型

日期

datetime

日期表示时间长河中的一个具体的时间戳。

一个值 `#datetime(year: 2023, month: 4, day: 19).display()` 偷偷混入了我们内容之中。

一个值 2023-04-19 偷偷混入了我们内容之中。

时间间隔

duration

时间间隔表示两个时间戳之间的时间差。

一个值 `#duration(days: 3, hours: 10).seconds()`s 偷偷混入了我们内容之中。

一个值 295200s 偷偷混入了我们内容之中。

typst-pdf 时间戳

```
set document(date: auto)
```


参考：数据读写与数据处理

read

数据格式

字节数组

image.decode

字符串 Unicode

正则匹配

wasm 插件

metadata

typst query

参考：数据读写与数据处理

read

数据格式

字节数组

image.decode

字符串 Unicode

正则匹配

wasm 插件

metadata

typst query

参考：数值计算

等待 [PR: Begin migration of calc functions to methods](#)

参考：数学模式

参考：导入和使用参考文献

参考：WASM 插件



Part.8 进阶教程 — 排版 IV



中文排版

建议结合 [《Typst 中文文档：中文用户指南》](#) 食用。

中文排版 —— 字体

设置中文字体

如果中文字体不符合 typst 要求，那么它不会选择你声明的字体，例如字体的变体数量不够，参考更详细的 [issue](#)。

1. `typst fonts` 查看系统字体，确保字体名字没有错误。
2. `typst fonts -font-path path/to/your-fonts` 指定字体目录。
3. `typst fonts -variants` 查看字体变体。
4. 检查中文字体是否已经完全安装。

设置语言和区域

如果字体与 `text(lang: ..., region: ...)` 不匹配，可能会导致连续标点的挤压。例如字体不是中国大陆的，标点压缩会出错；反之亦然。

伪粗体

伪斜体

同时设置中西字体（以宋体和 Times New Roman 为例）

中文排版 —— 间距

首行缩进

代码片段与中文文本之间的间距

数学公式与中文文本之间的间距

中文排版 —— 特殊排版

使用中文编号

为汉字和词组注音

为汉字添加着重号

竖排文本

使用国标文献格式

Typst v0.10.0 为止的已知问题及补丁

源码换行导致 linebreak

标点字体 fallback

模板参考

[本书各章节使用的模板](#)

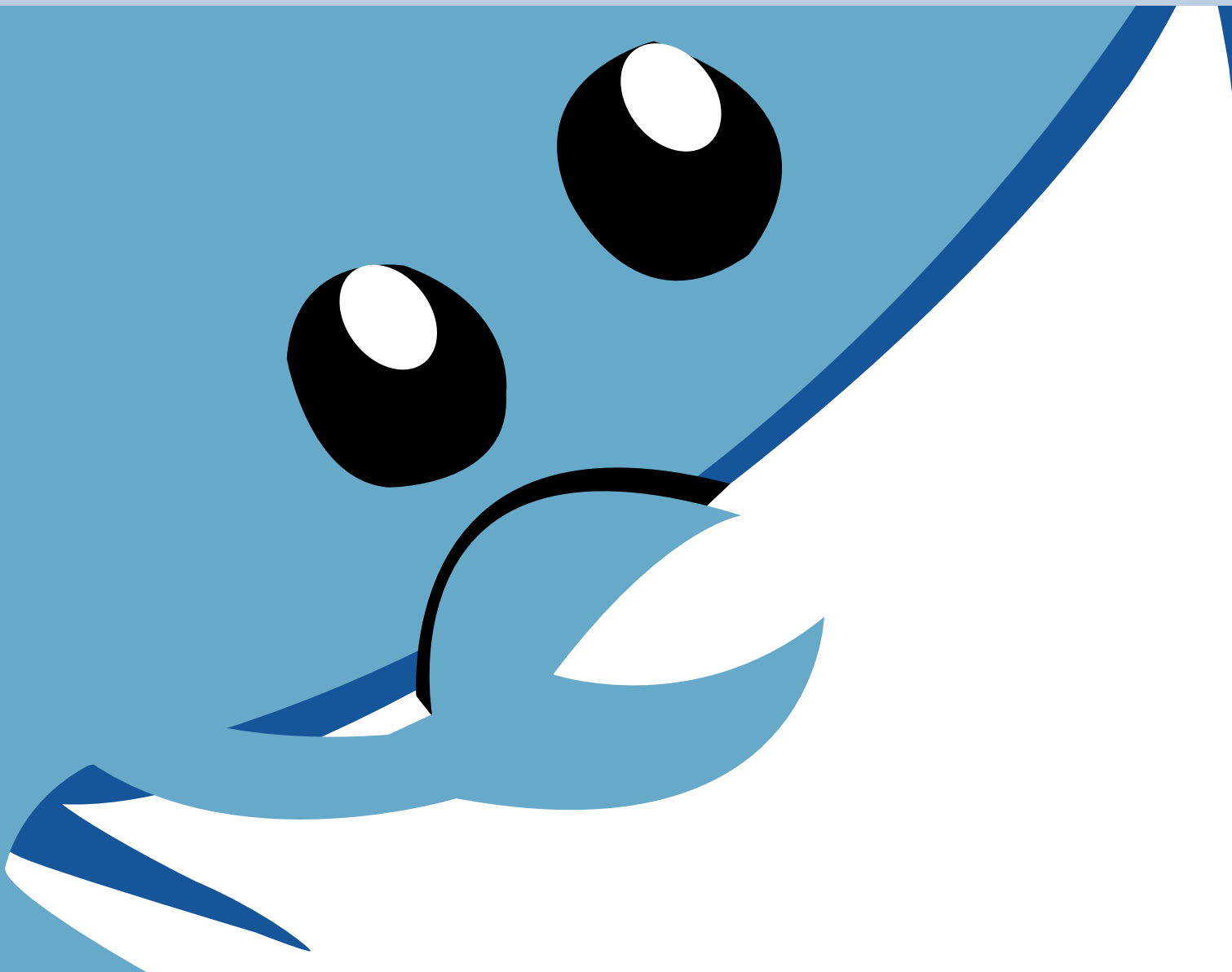
todo: 内嵌和超链接可直接食用的模板

进一步阅读

1. 参考 [《Typst 中文文档：中文用户指南》](#)，包含中文用户常见问题。
2. 参考 [《模块、外部库与多文件文档》](#)，在你的电脑上共享中文文档模板。
3. 推荐使用的[外部库列表](#)



Part.9 进阶教程 — 专题



编写一篇数学文档

制作一个组件库

制作一个外部插件

在 Typst 内执行 Js、Python、Typst 等

制作一个书籍模板

制作一个 CV 模板

制作一个 IEEE 模板



Part.10 进阶教程 — 公式和 定理



化学方程式

伪算法

定理环境

The background is a solid blue color. A horizontal band of a lighter blue shade runs across the middle. Above this band, there are several abstract, jagged blue shapes that look like stylized mountains or a creature's back. To the right, there is a large, stylized blue leaf or petal shape. Below the light blue band, there are large, abstract blue shapes that resemble a creature's face or a large leaf, with two prominent black circles containing white highlights, giving them a 3D appearance.

Part.11 进阶教程 — 杂项

字体设置

自定义代码高亮规则

自定义代码主题

读取外部文件和文本处理

Part.12 进阶教程 — 绘制图表

表



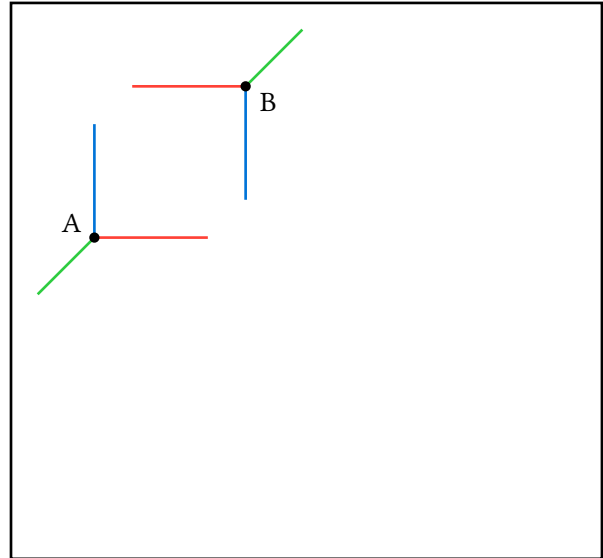
制表

立体几何

变换「坐标系」

在绘制立体图形（以及其他抽象图形）时，最重要的思路是变换「坐标系」（Viewport），以方便绘制。

```
#import "@preview/cetz:0.2.0": *
#canvas({
  import draw: *
  let axis-line(p) = {
    line((0, 0), (x: 1.5), stroke: red)
    line((0, 0), (y: 1.5), stroke: blue)
    line((0, 0), (z: 1.5), stroke: green)
    circle((0, 0, 0), fill: black, radius:
0.05)
    content((-0.4, 0.1, -0.2), p)
  }
  set-viewport((0, 0, 0), (4, 4, -4),
bounds: (4, 4, 4))
  axis-line("A")
  set-viewport((4, 4, 4), (0, 0, 0),
bounds: (4, 4, 4))
  axis-line("B")
})
```



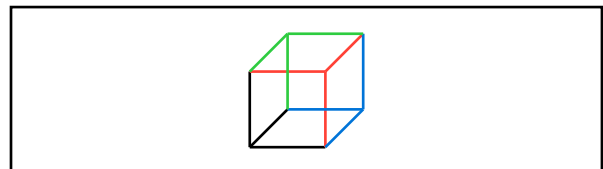
使用「空间变换」

由于「空间变换」（Transformation）的存在，你可以先绘制基本图形，再使用变换完成图形的绘制：

```
#let 六面体 = {
  import draw: *
  let neg(u) = if u == 0 { 1 } else { -1 }
  for (p, c) in (
    ((0, 0, 0), black), ((1, 1, 0), red),
    ((1, 0, 1), blue), ((0, 1, 1), green),
  ) {
    line(vector.add(p, (0, 0, neg(p.at(2)))), p, stroke: c)
    line(vector.add(p, (0, neg(p.at(1)), 0)), p, stroke: c)
    line(vector.add(p, (neg(p.at(0)), 0, 0)), p, stroke: c)
  }
}
```

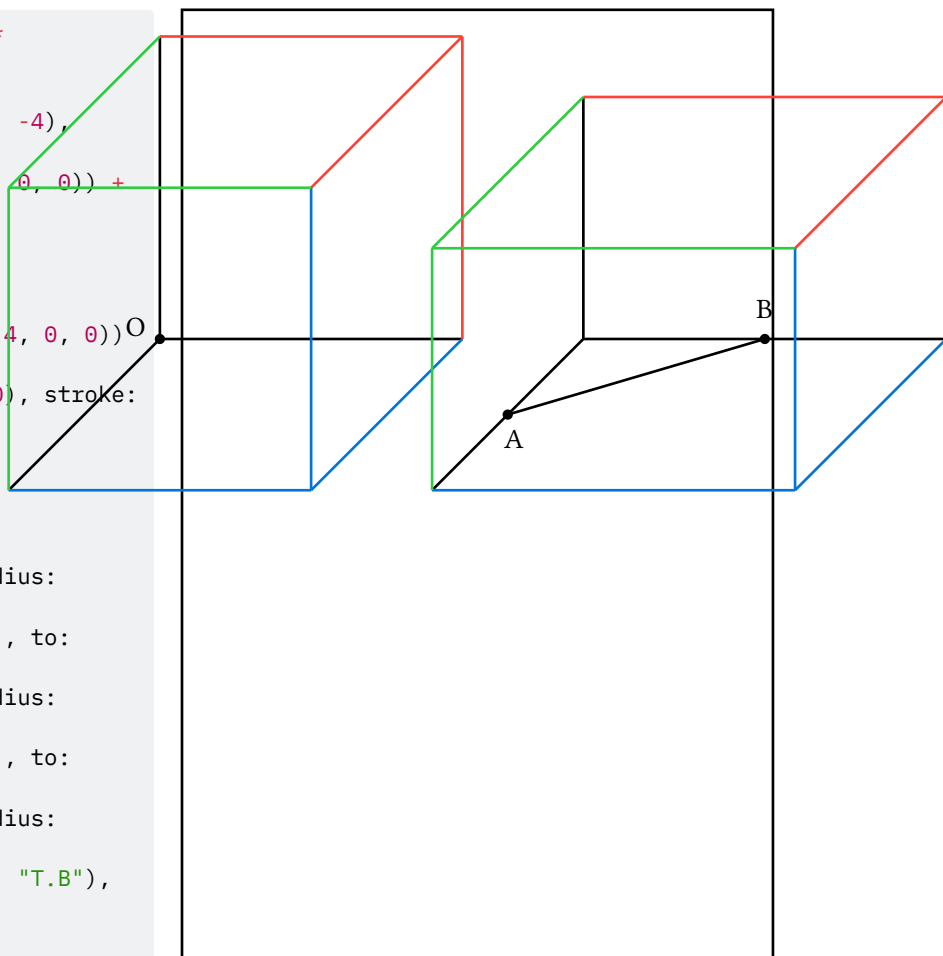
运行以下代码你将获得：

```
#import "@preview/cetz:0.2.0": *
#align(center, canvas(六面体))
```



六面体

```
#import "@preview/cetz:0.2.0": *
#align(center, canvas({
  import draw: *
  set-viewport((0, 0, 0), (4, 4, -4),
  bounds: (1, 1, 1))
  group(name: "S", translate((0, 0, 0)) +
{
  anchor("O", (0, 0, 0))
  六面体
})
  group(name: "T", translate((1.4, 0, 0))O
+ scale(x: 120%, y: 80%) + {
  line((0, 0, 0.5), (0.5, 0, 0), stroke:
black)
  anchor("A", (0, 0, 0.5))
  anchor("B", (0.5, 0, 0))
  六面体
})
  circle("S.O", fill: black, radius:
0.05 / 4)
  content((rel: (-0.08, 0.04, 0), to:
"S.O"), [O])
  circle("T.A", fill: black, radius:
0.05 / 4)
  content((rel: (0.02, -0.08, 0), to:
"T.A"), [A])
  circle("T.B", fill: black, radius:
0.05 / 4)
  content((rel: (0, 0.1, 0), to: "T.B"),
[B])
}))
```



拓扑图

统计图

状态机

电路图



Part.13 进阶教程 — 附录 II



参考：语法示例检索表 II

点击下方每行名称都可以跳转到对应章节的对应小节。

演示文稿（PPT）

论文模板

书籍模板



Part.14 进阶参考



参考：计数器和状态

参考：长度单位

已经移至《度量与布局》。

参考：布局函数

已经移至《度量与布局》。

参考：表格

参考：文档大纲