Mathematics in Lean

Release 0.1

Jeremy Avigad

Patrick Massot

Jun 18, 2025

CONTENTS

CHAPTER

ONE

引言

1.1 入门指南

本书教你使用 Lean 4 交互式证明助手来形式化数学。你需要一点儿数学知识，但并不需要太多。我们将涵盖从数论到测度论和分析的基础方面，如果你对它们不熟悉，你可以在学习的过程中逐渐掌握。我们也不预设任何对形式化方法的背景知识。形式化可以被视为一种计算机编程：我们将用一种 Lean 可以理解的规范化的语言，类似于编程语言，来编写数学定义、定理和证明。作为回报，Lean 提供反馈和信息，解析表达式并保证它们是形式良好的，并最终验证我们的证明的正确性。

你可以从 Lean 项目页面 和 Lean 社区网页 了解更多关于 Lean 的信息。本教程基于 Lean 庞大且不断增长的库 *Mathlib*。我们也强烈建议加入 Lean Zulip 在线聊天群，如果你还没有加入的话。在那里你会发现一个活跃而友好的 Lean 爱好者社区，愿意回答问题并提供精神支持。

- 项目在 Mathlib 版本 15a391f3007c5c377ba21af3e0a1d53502310685 上测试通过（2024.10.16）。Mathlib 更新迅速，可能会遇到旧版代码在新版 Mathlib 下无法通过的问题。如果读者遇到了版本不兼容问题，请提交 PR 以使本项目跟进最新 Mathlib，或者提交 issue 或联系译者协助解决。

虽然你可以在线阅读本书的 pdf （注：中文版暂不支持 pdf）或 html 版本，但它设计为可以交互式阅读，在 VS Code 编辑器中运行 Lean 代码。开始学习吧：

（中文用户可以参考 中文安装教程 ）

1. 按照这些 安装说明 安装 Lean 4 和 VS Code.
2. 确保你已安装了 git.
3. 按照这些 说明 来获取 `mathematics_in_lean` 存储库并在 VS Code 中打开它。

4. 本书的每个部分都有一个与之相关联的 Lean 文件，其中包含示例和练习。你可以在名为 `MIL` 的文件夹中按章节组织的地方找到它们。我们强烈建议复制该文件夹，并在复制的文件夹中进行实验和练习。这样可以保持原始文件的完整性，并且在存储库改动时更容易同步（见下文）。你可以将复制的文件夹命名为 `my_files` 或其他任何你喜欢的名字，并在其中创建自己的 Lean 文件。

在这之后，你可以按照以下步骤在 VS Code 的侧边栏中打开教科书：

1. 输入 `ctrl-shift-P` （在 macOS 中为 `command-shift-P` ）。

2. 在出现的栏中输入 `Lean 4: Docs: Show Documentation Resources`，然后按回车键。（一旦该选项在菜单中被高亮显示，你就可以按回车键选择它。）
3. 在打开的窗口中，点击 `Mathematics in Lean`.

或者，你还可以在云中运行 Lean 和 VS Code，使用 Gitpod。你可以在 Github 上的 Mathematics in Lean 项目官方页面 或 中文页面 找到如何操作的说明。尽管如此，我们仍然建议按照上述方式在 `MIL` 文件夹的副本中进行操作。

这本教科书及其相关存储库仍在不断完善中。你可以通过在 `mathematics_in_lean` 文件夹内输入 `git pull`，接着输入 `lake exe cache get` 来更新存储库。（这假设你没有改变 `MIL` 文件夹的内容，这也是我们建议你制作副本的原因。）

我们希望你在阅读教科书的同时，在 `MIL` 文件夹中完成练习，该文件夹包含了解释、说明和提示。文本通常会包含示例，就像这个例子一样：

```
#eval "Hello, World!"
```

你应该能够在相关的 Lean 文件中找到相应的示例。如果你点击该行，VS Code 将在 `Lean Goal` 窗口中显示 Lean 的反馈，如果你将光标悬停在 `#eval` 命令上，VS Code 将在弹出窗口中显示 Lean 对此命令的响应。我们鼓励你编辑文件并尝试自己的示例。

此外，本书还提供了许多具有挑战性的练习供你尝试。不要匆忙跳过这些练习！Lean 需要 * 交互式地做 * 数学，而不仅仅是阅读。完成练习是学习的核心内容。你不必完成所有练习；当你觉得已经掌握了相关技能时，可以自由地继续前进。你始终可以将你的解决方案与与每个部分相关的 `solutions` 文件夹中的解决方案进行比较。

1.2 概述

简言之，Lean 是用于构建复杂表达式的工具，它基于一种称为 **依值类型论（Dependent type theory）** 的形式语言。

每个表达式都有一个 **类型（Type）**，你可以使用 *#check* 命令来打印它。一些数学对象表达式的类型可能是像 $\mathbb{N}$ 或 $\mathbb{N} \to \mathbb{N}$ 这样的。

```
#check 2 + 2

def f (x : ℕ) :=
  x + 3

#check f
```

数学命题的类型是 *Prop*。

```
#check 2 + 2 = 4
```

(continues on next page)

(continued from previous page)

```
def FermatLastTheorem :=
  ∀ x y z n : ℕ, n > 2 ∧ x * y * z ≠ 0 → x ^ n + y ^ n ≠ z ^ n

#check FermatLastTheorem
```

设命题 *P* 类型为 *Prop*，你可以构造类型为 *P* 的表达式，它们是命题 *P* 的证明。

```
theorem easy : 2 + 2 = 4 :=
  rfl

#check easy

theorem hard : FermatLastTheorem :=
  sorry

#check hard
```

如果你设法构建了一个类型为 `FermatLastTheorem` 的表达式，并且 Lean 接受它作为该类型的项，那么你已经干了一件伟大的事。（使用 `sorry` 是作弊，Lean 知道这一点。）所以现在你知道了游戏目标。剩下要学习的只有规则了。

- Kevin Buzzard 正在领导这件伟大的事，详见：The Fermat's Last Theorem Project

本书配套教程 Theorem Proving in Lean，中文版 Lean 中的定理证明，提供了对 Lean 的基础逻辑框架和核心语法更全面的介绍。Lean 中的定理证明适用于那些在使用新洗碗机之前更喜欢从头到尾阅读用户手册的人。如果你是那种更喜欢先动手尝试，以后再弄清细节的人，那么从本书开始更合适，需要时随时可以回去参考 Lean 中的定理证明。

Lean 形式化数学与 Lean 中的定理证明的另一个区别在于，本书更加强调 **证明策略 (Tactics)** 的使用。考虑到我们试图构建复杂表达式，Lean 提供了两种方法：直接构造表达式本身，或者向 Lean 提供 *指令*，告诉它如何构建这些表达式。例如，下面构造的证明表达式在说明，如果 `n` 是偶数，则 `m * n` 也是偶数：

```
example : ∀ m n : Nat, Even n → Even (m * n) := fun m n ⟨k, (hk : n = k + k)⟩ ↦
  have hmn : m * n = m * k + m * k := by rw [hk, mul_add]
  show ∃ l, m * n = l + l from ⟨_, hmn⟩
```

这个 **证明项**可以压缩成一行：

```
example : ∀ m n : Nat, Even n → Even (m * n) :=
fun m n ⟨k, hk⟩ ↦ ⟨m * k, by rw [hk, mul_add]⟩
```

以下是同一定理的策略式证明，其中以 `--` 开头的行是注释，因此被 Lean 忽略：

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  -- 设 `m` 和 `n` 是自然数，设 `n = 2 * k`
  rintro m n ⟨k, hk⟩
  -- 需证 `m * n` 是某自然数的两倍。那就设它是 `m * k` 的两倍吧
  use m * k
  -- 代入 `n`
  rw [hk]
  -- 剩下的就很显然了
  ring
```

当你在 VS Code 中输入上述证明的每一行时，Lean 会在一个单独的窗口中显示 **证明状态**，告诉你已经建立了哪些事实，以及要证明你的定理还需要完成什么任务。你可以通过逐行逐步骤回顾证明，因为 Lean 将继续显示光标所在点的证明状态。本例中，证明的第一行引入了 `m` 和 `n`（此时可以重命名它们，如果我们想的话），并且将假设 `Even n` 分解为 `k` 和假设 `n = 2 * k`. 第二行，`use m * k`, 声明我们将通过证明 `m * n = 2 * (m * k)` 来证明 `m * n` 是偶数。下一行使用了 `rw` 策略（rewrite 的缩写）在目标中将 `n` 替换为 `2 * k`，得到的新目标 `m * (2 * k) = 2 * (m * k)` 最终被 `ring` 策略解决。

逐步构建证明并获得增量反馈的能力非常强大。因此，策略证明通常比编写证明项更容易也更快。两者之间没有明显的区别：策略证明可以插入到证明项中，就像我们在上面的例子中使用短语 `by rw [hk, mul_left_comm]` 一样。我们还将看到，反之，将一个简短的证明项插入到策略证明中通常也很有用。虽然如此，但在这本书中，我们会把重点放在策略的使用上。

在我们的例子中，策略证明也可以简化为一行：

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩; use m * k; rw [hk]; ring
```

在这里，我们使用策略来执行每个细节证明步骤。但是 Lean 提供实质性的自动化，并且可以证明更长的计算和更大的推理步骤。例如，我们可以使用特定的规则调用 Lean 的化简器，用于化简关于奇偶性的语句，以自动证明我们的定理。

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  intros; simp [*, parity_simps]
```

两本入门教程之间的另一个重大区别是，Lean 中的定理证明仅依赖于 Lean 内核以及其内置的策略，而 Lean 形式化数学建立在 Lean 强大且不断增长的库 *Mathlib* 的基础上。因此，我们会向您展示如何使用库中的一些数学对象和定理，以及一些非常有用的策略。这本书无意用于对库进行完整描述；社区 网页包含了详尽的文档。不如说，本入门教程是向您介绍形式化背后的思维风格，让您可以轻松浏览库并自行查找内容。

交互式定理证明可能会令人沮丧，学习曲线很陡峭。但是 Lean 社区非常欢迎新人，而且任何时间都有人在 Lean Zulip 聊天群 上在线回答问题。我们希望能在那里见到你，并且毫无疑问，很快你也将能够回答这类问题并为 *Mathlib* 的发展做出贡献。

因此，如果你选择接受这个任务，你的使命如下：投入其中，尝试练习，有问题就来 Zulip 提问，并享受乐

趣。但要注意：交互式定理证明将挑战你以全新的方式思考数学和进行数学推理。这可能改变你的生活。

致谢. Acknowledgments. We are grateful to Gabriel Ebner for setting up the infrastructure for running this tutorial in VS Code, and to Kim Morrison and Mario Carneiro for help porting it from Lean 4. We are also grateful for help and corrections from Takeshi Abe, Julian Berman, Alex Best, Thomas Browning, Bulwi Cha, Hanson Char, Bryan Ginge Chen, Steven Clontz, Mauricio Collaris, Johan Commelin, Mark Czubin, Alexandru Duca, Pierpaolo Frasa, Denis Gorbachev, Winston de Greef, Mathieu Guay-Paquet, Marc Huisinga, Benjamin Jones, Julian Külshammer, Victor Liu, Jimmy Lu, Martin C. Martin, Giovanni Mascellani, John McDowell, Isaiah Mindich, Hunter Monroe, Pietro Monticone, Oliver Nash, Emanuelle Natale, Pim Otte, Bartosz Piotrowski, Nicolas Rolland, Keith Rush, Yannick Seurin, Guilherme Silva, Pedro Sánchez Terraf, Matthew Toohey, Alistair Tucker, Floris van Doorn, Eric Wieser, and others. Our work has been partially supported by the Hoskinson Center for Formal Mathematics.

感谢`**Lean-zh 中文社区 <https://www.leanprover.cn/>**`_ 的伙伴们参与译本的创作和校对。

CHAPTER

TWO

基础

介绍 Lean 中数学推理的基本要素：计算、应用引理和定理，以及对通用结构进行推理。

2.1 计算

通常我们学习数学计算时并不将其视为证明。但是，当我们像 Lean 要求的那样验证计算中的每一步时，最终的结果就是一个证明，我们在证明了算式的左侧等于右侧。

在 Lean 中，陈述一个定理等同于设立证明该定理的目标。Lean 提供了重写（rewrite）策略 `rw`，它会使用一个等式，`rw` 在 **目标**中找到符合等式左侧的部分，然后将这部分替换为等式右侧。例如：如果 `a`、`b` 和 `c` 是实数，那么 `mul_assoc a b c` 是等式 `a * b * c = a * (b * c)`，而 `mul_comm a b` 是等式 `a * b = b * a`。你想运用这些等式时可以只引用它们的名字。在 Lean 中，乘法左结合，因此 `mul_assoc` 的左侧也可以写成 `(a * b) * c`，但是没必要写括号就最好不写。

让我们尝试一下 `rw`。

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
  rw [mul_comm a b]
  rw [mul_assoc b a c]
```

教材源码里的 `import` 行从 Mathlib 中导入了实数理论，以及有用的自动化功能。为简洁计正文中省略，如果你想自己试着运行例子，你可以查询源码来了解。

`ℝ` 字符通过 `\R` 或 `\real` 输入，然后按下空格或 Tab 键。当阅读 Lean 文件时，如果你将光标悬停在一个符号上，VS Code 将显示用于输入该符号的语法。如果你想查看所有可用的缩写，可以按下 Ctrl-Shift-P，然后输入 abbreviations 来访问 `Lean 4: Show all abbreviations` 命令。如果您的键盘上没有方便使用的反斜杠，可以通过更改 `lean4.input.leader` 设置来改变前导字符。

当光标位于策略证明的中间时，Lean 会在 *Lean Infoview* 窗口中报告当前的 **证明状态**。当你将光标移到证明的每一步时，你可以看到状态的变化。Lean 中的典型证明状态可能如下所示：.. code-block:

```
1 goal
x y : ℕ,
h₁ : Prime x,
```

(continues on next page)

(continued from previous page)

```
h₂ : ¬Even x,
h₃ : y > x
⊢ y ≥ 4
```

⊢ 前面的部分是当前位置处我们所拥有的对象和假设，称为 **语境（context）**。在这个例子中，语境包括两个对象，自然数 `x` 和 `y`；包括三个假设，分别具有标识符 `h₁` 、`h₂` 和 `h₃` （下标用 `\1`、`\2` ……键入）。Lean 的语法要求在语境中每个假设都拥有一个名字，叫什么都可以，比如不下标的 `h1` 也可以（实际上这是类型论的要求，例如本例中 `h₁` 这个”名字”其实标记着类型为命题 `Prime x` 的项。），或者 `foo`、`bar` 和 `baz`。最后一行用 ⊢ 标记的代表 **目标（goal）**，即要证明的事实。对于目标这个词，一些人有时人们使用 **目的（target）** 表示要证明的事实，使用 **目标（goal）** 表示语境和目的（target）的组合，不过在特定上下文中不致混淆。

接下来做一些练习！用 `rw` 策略替换掉下面的 `sorry` 。为此再告诉你一个新技巧：你可以用左箭头 `←` （`\l`）来调转一个等式的方向，从而让 `rw` 从另一边做替换操作。例如，`rw ← mul_assoc a b c` 会把目标里的 `a * (b * c)` 替换成 `a * b * c` 。注意这里指的是 `mul_assoc` 等式的从右到左，它与目标的左边或右边无关。

```
example (a b c : ℝ) : c * b * a = b * (a * c) := by
  sorry

example (a b c : ℝ) : a * (b * c) = b * (a * c) := by
  sorry
```

你也可以不带参数使用诸如 `mul_assoc` 或者 `mul_comm` 这些等式。这些情况下重写策略会识别它在目标中匹配到的第一个模式。

```
example (a b c : ℝ) : a * b * c = b * c * a := by
  rw [mul_assoc]
  rw [mul_comm]
```

你还可以只提供一部分参数，例如 `mul_comm a` 识别所有形如 `a * ?` 的模式然后重写为 `? * a` 。下面的练习中你可以试试在第一个里面不给参数，第二个里面只给一个参数。

```
example (a b c : ℝ) : a * (b * c) = b * (c * a) := by
  sorry

example (a b c : ℝ) : a * (b * c) = b * (a * c) := by
  sorry
```

你也可以 `rw` 局部语境里的条件：

```
example (a b c d e f : ℝ) (h : a * b = c * d) (h' : e = f) : a * (b * e) = c * (d *␣
↪f) := by
  rw [h']
```

(continues on next page)

(continued from previous page)

```
  rw [← mul_assoc]
  rw [h]
  rw [mul_assoc]
```

试试看：（第二个练习里面你可以使用定理 `sub_self`，`sub_self a` 代表等式 `a - a = 0`。）

```
example (a b c d e f : ℝ) (h : b * c = e * f) : a * b * c * d = a * e * f * d := by
  sorry

example (a b c d : ℝ) (hyp : c = b * a - d) (hyp' : d = a * b) : c = 0 := by
  sorry
```

现在我们介绍 Lean 的一些有用的特性. 首先，通过在方括号内列出相关等式，可以用一行重写执行多个命令。

```
example (a b c d e f : ℝ) (h : a * b = c * d) (h' : e = f) : a * (b * e) = c * (d *␣
↪f) := by
  rw [h', ← mul_assoc, h, mul_assoc]
```

将光标放在 rewrite 列表中的任意逗号后，仍然可以看到进度。

另一个技巧是我们可以在例子或定理之外一次性地声明变量. 当 Lean 在定理的陈述中看到它们时，它会自动将它们包含进来。

```
variable (a b c d e f : ℝ)

example (h : a * b = c * d) (h' : e = f) : a * (b * e) = c * (d * f) := by
  rw [h', ← mul_assoc, h, mul_assoc]
```

检查上述证明开头的策略状态，可以发现 Lean 确实包含了相关变量，而忽略了声明中没有出现的 g。我们可以把声明的范围放在一个 `section ... end` 块中做成类似其他编程语言中局部变量的效果. 最后，回顾一下第一章，Lean 为我们提供了一个命令来确定表达式的类型：

```
section
variable (a b c : ℝ)

#check a
#check a + b
#check (a : ℝ)
#check mul_comm a b
#check (mul_comm a b : a * b = b * a)
#check mul_assoc c a b
#check mul_comm a
```

(continues on next page)

(continued from previous page)

```
#check mul_comm

end
```

#check 命令对对象和命题都有效。在响应命令 #check a 时，Lean 报告 a 的类型为 ℝ。作为对命令 #check mul_comm a b 的响应，Lean 报告 mul_comm a b 是事实 a * b = b * a 的证明。命令 #check (a : ℝ) 表明我们期望 a 的类型是 ℝ，如果不是这样，Lean 将引发一个错误。稍后我们将解释最后三个 #check 命令的输出，你可以尝试自己写一些 #check 命令。

我们再举几个例子。定理 two_mul a 表示 2 * a = a + a。定理 add_mul 和 mul_add 表示乘法对加法的分配律，定理 add_assoc 表示加法的结合律。使用 #check 命令查看精确的语句。

```
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b := by
  rw [mul_add, add_mul, add_mul]
  rw [← add_assoc, add_assoc (a * a)]
  rw [mul_comm b a, ← two_mul]
```

虽然可以通过在编辑器中逐步检查来弄清楚这个证明中发生了什么，但很难单独阅读。Lean 使用 calc 关键字提供了一种更结构化的方法来编写类似这样的证明。

```
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
  calc
    (a + b) * (a + b) = a * a + b * a + (a * b + b * b) := by
      rw [mul_add, add_mul, add_mul]
    _ = a * a + (b * a + a * b) + b * b := by
      rw [← add_assoc, add_assoc (a * a)]
    _ = a * a + 2 * (a * b) + b * b := by
      rw [mul_comm b a, ← two_mul]
```

请注意，证明 **不**以 by 开头：以 calc 开头的表达式是一个证明项。calc 表达式也可以在策略证明块中使用，Lean 将其解释为使用证明项的结果来解决当前目标的指令。calc 语法必须严格仿照上例格式使用下划线和对齐。Lean 使用缩进来确定策略块或 calc 块开始和结束的地方。试着改变上面证明中的缩进，看看会发生什么。

编写 calc 证明的一种方法是首先使用 sorry 策略填空，确保 Lean 认可中间步骤表达式，然后使用策略对各个步骤进行论证。

```
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
  calc
    (a + b) * (a + b) = a * a + b * a + (a * b + b * b) := by
      sorry
    _ = a * a + (b * a + a * b) + b * b := by
```

(continues on next page)

(continued from previous page)

```
    sorry
  _ = a * a + 2 * (a * b) + b * b := by
    sorry
```

试试用两种方法证明以下等式：只用 `rw` 和用更结构化的 `calc` 。

```
example : (a + b) * (c + d) = a * c + a * d + b * c + b * d := by
  sorry
```

接下来的练习有点挑战性。你可以用下列的定理。

```
example (a b : ℝ) : (a + b) * (a - b) = a ^ 2 - b ^ 2 := by
  sorry

#check pow_two a
#check mul_sub a b c
#check add_mul a b c
#check add_sub a b c
#check sub_sub a b c
#check add_zero a
```

我们还可以在语句集中的假设中执行重写。例如，`rw [mul_comm a b] at hyp` 将假设 `hyp` 中的 `a * b` 替换为 `b * a` 。

```
example (a b c d : ℝ) (hyp : c = d * a + b) (hyp' : b = a * d) : c = 2 * a * d := by
  rw [hyp'] at hyp
  rw [mul_comm d a] at hyp
  rw [← two_mul (a * d)] at hyp
  rw [← mul_assoc 2 a d] at hyp
  exact hyp
```

最后一步中 `exact` 策略使用 `hyp` 来解决目标的原理是，到此 `hyp` 不就是目标本身了吗！(Exactly!)

最后我们介绍一个 Mathlib 提供的强力自动化工具 `ring` 策略，它专门用来解决交换环中的等式，只要这些等式是完全由环公理导出的而不涉及别的假设。

```
example : c * b * a = b * (a * c) := by
  ring

example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b := by
  ring
```

(continues on next page)

(continued from previous page)

```
example : (a + b) * (a - b) = a ^ 2 - b ^ 2 := by
  ring

example (hyp : c = d * a + b) (hyp' : b = a * d) : c = 2 * a * d := by
  rw [hyp, hyp']
  ring
```

ring 策略通过 import Mathlib.Data.Real.Basic 导入，但下一节会看到它不止可以用在实数的计算上。它还可以通过 import Mathlib.Tactic 导入。对于其他常见的代数结构也有类似的策略。

rw 有一种叫做 nth_rewrite 的变体，如果目标中存在多处匹配，nth_rw 允许你指出想要实施替换的位置。匹配项从 1 开始计数，在下面的例子中 nth_rewrite 2 [h] 用 c 替换了第二个 a + b。

```
example (a b c : ℕ) (h : a + b = c) : (a + b) * (a + b) = a * c + b * c := by
  nth_rw 2 [h]
  rw [add_mul]
```

2.2 证明代数结构中的等式

数学中，环由一个对象集合 R、运算 $+$ $\times$、常数 0 和 1、求逆运算 $x \mapsto -x$ 构成，并满足：

- R 与 $+$ 构成阿贝尔群，0 是加法单位元，负数是逆。
- 1 是乘法单位元，乘法满足结合律和对加法的分配律。

在 Lean 中，一组对象被表示为类型 R。环公理如下：

```
variable (R : Type*) [Ring R]

#check (add_assoc : ∀ a b c : R, a + b + c = a + (b + c))
#check (add_comm : ∀ a b : R, a + b = b + a)
#check (zero_add : ∀ a : R, 0 + a = a)
#check (neg_add_cancel : ∀ a : R, -a + a = 0)
#check (mul_assoc : ∀ a b c : R, a * b * c = a * (b * c))
#check (mul_one : ∀ a : R, a * 1 = a)
#check (one_mul : ∀ a : R, 1 * a = a)
#check (mul_add : ∀ a b c : R, a * (b + c) = a * b + a * c)
#check (add_mul : ∀ a b c : R, (a + b) * c = a * c + b * c)
```

一会儿再讲第一行的方括号是什么意思，现在你只需要知道我们声明了一个类型 R 和 R 上的环结构。这样我们就可以表示一般的环中的元素并使用环的定理库。

前一节用过上面的一些定理，所以你应该感觉很熟悉。Lean 不止能在例如自然数和整数这样具体的数学结构上证明东西，也可以在环这样抽象的公理化的结构上证明东西。Lean 支持抽象和具体结构的通用推理，并且

有能力识别符合公理的实例。任何关于环的定理都可以应用于具体的环，如整数 ℤ、有理数 ℚ、复数 ℂ，和抽象的环，如任何有序环或任何域。

然而，并不是所有实数的重要性质在任意环中都成立。例如，实数乘法是可交换的，但一般情况下并不成立。例如实数矩阵构成的环的乘法通常不能交换。如果我们声明 R 是一个交换环 CommRing，那么上一节中的所有关于 R 的定理在 R 中仍然成立。

```
variable (R : Type*) [CommRing R]
variable (a b c d : R)

example : c * b * a = b * (a * c) := by ring

example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b := by ring

example : (a + b) * (a - b) = a ^ 2 - b ^ 2 := by ring

example (hyp : c = d * a + b) (hyp' : b = a * d) : c = 2 * a * d := by
  rw [hyp, hyp']
  ring
```

别的证明也都不需要变，你可以自己试试看。当证明很短时，比如你只用了一个 ring 或者 linarith 或者 sorry，你可以把它们写进 by 的同一行里。好的证明写手需要平衡简洁性和可读性。

本节里面我们会证明更多环定理，它们基本上都在 Mathlib 里面，看完这一节你会对 Mathlib 里面的东西更熟悉。同时这也是训练你的证明能力。

Lean 提供了类似于别的编程语言中的“局域变量”的变量名组织机制。通过命令 namespace bar 创建一个命名空间 bar 并引入定义或者定理 foo，你在命名空间外面引用它时全名为 bar.foo。命令 open bar 可以打开这个命名空间，此时你可以用短名字 foo。下面我们为了不与 Mathlib 中的定理名冲突，我们打开一个名为 MyRing 的命名空间。

下面的例子证明了 add_zero 和 add_right_neg，所以它们不需要作为环公理。

```
namespace MyRing
variable {R : Type*} [Ring R]

theorem add_zero (a : R) : a + 0 = a := by rw [add_comm, zero_add]

theorem add_right_neg (a : R) : a + -a = 0 := by rw [add_comm, neg_add_cancel]

#check MyRing.add_zero
#check add_zero

end MyRing
```

我们重新证明了库中的定理，但是我们可以继续使用库中的版本。但是下面的练习中请不要作弊，我们只能用我们之前证明过的定理。

（如果你仔细注意的话，你可能已经注意到我们把 `(R: Type*)` 中的圆括号改成了 `{R: Type*}` 中的花括号。这里声明 `R` 是一个 **隐式参数（implicit argument）**。稍后会解释这意味着什么。）

下面这个定理很有用：

```
theorem neg_add_cancel_left (a b : R) : -a + (a + b) = b := by
  rw [← add_assoc, neg_add_cancel, zero_add]
```

证明它的配套版本：

```
theorem add_neg_cancel_right (a b : R) : a + b + -b = a := by
  sorry
```

然后用它们证明下面几个（最佳方案仅需三次重写）：

```
theorem add_left_cancel {a b c : R} (h : a + b = a + c) : b = c := by
  sorry

theorem add_right_cancel {a b c : R} (h : a + b = c + b) : a = c := by
  sorry
```

现在解释一下花括号的意思。假设你现在语句集里面拥有变量 `a`、`b`、`c` 和一个假设 `h : a + b = a + c`，然后你想得到结论 `b = c`。在 Lean 中，定理可以应用于假设和事实，就像将它们应用于对象一样，因此你可能会认为 `add_left_cancel a b c h` 是事实 `b = c` 的证明。但其实明确地写出 `a b c` 是多余的，因为假设 `h` 的形式就限定了它们正是我们想使用的对象。现下输入几个额外的字符并不麻烦，但是更复杂的表达式中就会很繁琐。Lean 支持把参数标记为隐式，这意味着它们可以且应该被省略，能通过后面的的命题和假设中推断出来。`{a b c: R}` 中的花括号正是这种隐式参数标记。因此根据定理的表述，正确的表达式是 `add_left_cancel h`。

下面演示个新玩意儿，让我们从环公理中证明 `a * 0 = 0`。

```
theorem mul_zero (a : R) : a * 0 = 0 := by
  have h : a * 0 + a * 0 = a * 0 + 0 := by
    rw [← mul_add, add_zero, add_zero]
  rw [add_left_cancel h]
```

你通过 `have` 策略引入了一个辅助性新目标，`a * 0 + a * 0 = a * 0 + 0`，与原始目标具有相同的语境。这个目标下的“子证明”块需要缩进。证出这个子目标之后我们就多了一个新的命题 `h`，可以用于证明原目标。这里我们看到 `add_left_cancel h` 的结果恰好就是原目标。

我们同样可以使用 `apply add_left_cancel h` 或 `exact add_left_cancel h` 来结束证明。`exact` 策略将能够完整证明当前目标的证明项作为参数，而不创建任何新目标。`apply` 策略是一种变体，它的论证不一定是一个完整的证明。缺失的部分要么由 Lean 自动推断，要么成为需要证明的新目标。虽然 `exact` 策略在技

术上是多余的，因为它严格来说不如 apply 强大，但它增加了可读性。(下一节详细讲解 apply 策略的用法。)

乘法不一定可交换，所以下面的定理也需要证。

```
theorem zero_mul (a : R) : 0 * a = 0 := by
  sorry
```

更多练习：

```
theorem neg_eq_of_add_eq_zero {a b : R} (h : a + b = 0) : -a = b := by
  sorry

theorem eq_neg_of_add_eq_zero {a b : R} (h : a + b = 0) : a = -b := by
  sorry

theorem neg_zero : (-0 : R) = 0 := by
  apply neg_eq_of_add_eq_zero
  rw [add_zero]

theorem neg_neg (a : R) : - -a = a := by
  sorry
```

我们必须在第三个定理中指定 (-0 : R)，因为 Lean 不知道我们想到的是哪个 0，默认情况下它是自然数。

在 Lean 中，环中减去一个元素等于加上它的加法逆元。

```
example (a b : R) : a - b = a + -b :=
  sub_eq_add_neg a b
```

实数的减法就是被如此定义的，因此：

```
example (a b : ℝ) : a - b = a + -b :=
  rfl

example (a b : ℝ) : a - b = a + -b := by
  rfl
```

rfl 是自反性（reflexivity）的缩写。第一个例子中当它作为 a - b = a + -b 的证明项，Lean 展开定义并验证两边是相同的。第二个例子中 rfl 策略也是如此。这是在 Lean 的基础逻辑中所谓的定义相等的一个例子。这意味着不仅可以用 sub_eq_add_neg 重写来替换 a - b = a + -b，而且在某些情况下，当处理实数时，您可以互换使用方程的两边。例如，您现在有足够的信息来证明上一节中的 self_sub 定理：

```
theorem self_sub (a : R) : a - a = 0 := by
  sorry
```

你可以使用 `rw` 来证，不过如果不是任意环 `R` 而是实数的话，你也可以用 `apply` 或者 `exact`。

Lean 知道 `1 + 1 = 2` 对任何环都成立。你可以用它来证明上一节中的定理 `two_mul`：

```
theorem one_add_one_eq_two : 1 + 1 = (2 : R) := by
  norm_num

theorem two_mul (a : R) : 2 * a = a + a := by
  sorry
```

上面的一些定理并不需要环结构甚至加法交换律，有 **群**结构就够了，群公理是下面这些：

```
variable (A : Type*) [AddGroup A]

#check (add_assoc : ∀ a b c : A, a + b + c = a + (b + c))
#check (zero_add : ∀ a : A, 0 + a = a)
#check (neg_add_cancel : ∀ a : A, -a + a = 0)
```

群运算可交换的话习惯上用加号（但是这只是习惯而已，`AddGroup` 并不真的可交换），否则用乘号。Lean 提供乘法版本 `Group` 和加法版本 `AddGroup`，以及它们的可交换版本 `CommGroup` 和 `AddCommGroup`。

```
variable {G : Type*} [Group G]

#check (mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
#check (one_mul : ∀ a : G, 1 * a = a)
#check (inv_mul_cancel : ∀ a : G, a⁻¹ * a = 1)
```

试试用这些群公理证明以下命题。你可以引入一些引理。

```
theorem mul_inv_cancel (a : G) : a * a⁻¹ = 1 := by
  sorry

theorem mul_one (a : G) : a * 1 = a := by
  sorry

theorem mul_inv_rev (a b : G) : (a * b)⁻¹ = b⁻¹ * a⁻¹ := by
  sorry
```

一步一步用这些定理做证明非常麻烦，所以在这些代数结构上 Mathlib 提供了类似 `ring` 的策略：`group` 用于非交换的乘法群，`abel` 用于可交换加法群，`noncomm_ring` 用于非交换环。代数结构 `Ring` 和 `CommRing` 分

别对应的自动化策略被称做 noncomm_ring 和 ring，这似乎很奇怪。这在一定程度上是由于历史原因，但也因为使用更短的名称来处理交换环的策略更方便，因为它使用得更频繁。

2.3 使用定理和引理

重写对于证明等式很有用，但是对于其他类型的定理呢？例如，我们如何证明一个不等式，比如在 $b \le c$ 时 $a + e^b \le a + e^c$？本节我们会着重使用 apply 和 exact 。

考虑库定理 le_refl 和 le_trans：

```
#check (le_refl : ∀ a : ℝ, a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
```

→ 是右结合的，因此 le_trans 应该被解释为 a ≤ b → (b ≤ c → a ≤ c) 。详细规则在 Section 3.1 一节中解释。标准库设计者已经将 le_trans 中的 a , b 和 c 设置为隐式参数，也就是在使用时从语境中推断。(强制显式参数将在后面讨论)。例如，当假设 h : a ≤ b 和 h' : b ≤ c 在语境中时，以下所有语句都有效：

```
variable (h : a ≤ b) (h' : b ≤ c)

#check (le_refl : ∀ a : Real, a ≤ a)
#check (le_refl a : a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (le_trans h : b ≤ c → a ≤ c)
#check (le_trans h h' : a ≤ c)
```

apply 策略的作用规则是：它把被 apply 的表达式的 **结论**与当前的目标相匹配，并将 **前提**（如果有的话）作为新目标。如果给定的证明与目标完全匹配（定义等价），则可以使用 exact 策略代替 apply 。你可以考察下面的例子：

```
example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z := by
  apply le_trans
  -- le_trans : a ≤ b → b ≤ c → a ≤ c, `a ≤ c` 匹配到了目标 `x ≤ z`
  -- 于是 `a` 被重命名（正式地："实例化"）为 `x`, `c`被重命名为 `z`,
  -- `b` 尚未得到它的新名字，因此处在"元变量"（metavariable）状态，表示为 `?b`
  -- 接下来两个前提 `x ≤ ?b`, `?b ≤ z` 成为了新目标
  · apply h₀
  · apply h₁

example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z := by
  apply le_trans h₀
  apply h₁

example (x y z : ℝ) (h₀ : x ≤ y) (h₁ : y ≤ z) : x ≤ z :=
```

(continues on next page)

(continued from previous page)

```
  le_trans h₀ h₁

example (x : ℝ) : x ≤ x := by
  apply le_refl

example (x : ℝ) : x ≤ x :=
  le_refl x
```

在第一个示例中，`apply le_trans` 创建两个目标，我们使用点 `·`（用 `\.` 或 `\centerdot` 键入，或者直接用英文句号 `.` 也可以）来指示对每个目标分别进行证明。这些点并不是语法上必要的，但它们聚焦了目标增加了可读性：在点引入的代码块中，只有一个目标可见，并且必须在代码块结束之前完成证明。另外，点和策略之间其实也可以不用空格。在第三个和最后一个示例中，我们直接构造了证明项 `le_transle_trans h₀ h₁` 和 `le_refl x`。

另一些库定理：

```
#check (le_refl : ∀ a, a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (lt_of_le_of_lt : a ≤ b → b < c → a < c)
#check (lt_of_lt_of_le : a < b → b ≤ c → a < c)
#check (lt_trans : a < b → b < c → a < c)
```

利用这些定理和 `apply` 和 `exact` 策略来证明下面的问题：

```
example (h₀ : a ≤ b) (h₁ : b < c) (h₂ : c ≤ d) (h₃ : d < e) : a < e := by
  sorry
```

实际上 Lean 有一个自动化策略来证明这类问题：

```
example (h₀ : a ≤ b) (h₁ : b < c) (h₂ : c ≤ d) (h₃ : d < e) : a < e := by
  linarith
```

`linarith` 策略用于处理 **线性算术**，也就是仅涉及加法和数乘的等式和不等式。

```
example (h : 2 * a ≤ 3 * b) (h' : 1 ≤ a) (h'' : d = 2) : d + a ≤ 5 * b := by
  linarith
```

除了语境中的等式和不等式之外，你还能把其他式子作为参数传给 `linarith`。在下一个示例中，`exp_le_exp.mpr h'` 对应表达式 `exp b ≤ exp c`，稍后解释原因。在 Lean 中，我们用 `f x` 来表示将函数 `f` 应用于参数 `x`，类似地我们用 `h x` 来表示将事实或定理 `h` 应用到参数 `x`。括号仅用于复合参数，如 `f (x + y)`。如果没有括号，`f x + y` 将被解析为 `(f x) + y`。

```
example (h : 1 ≤ a) (h' : b ≤ c) : 2 + a + exp b ≤ 3 * a + exp c := by
  linarith [exp_le_exp.mpr h']
```

这里列出更多库定理，可以用于实数上的不等式：

```
#check (exp_le_exp : exp a ≤ exp b ↔ a ≤ b)
#check (exp_lt_exp : exp a < exp b ↔ a < b)
#check (log_le_log : 0 < a → a ≤ b → log a ≤ log b)
#check (log_lt_log : 0 < a → a < b → log a < log b)
#check (add_le_add : a ≤ b → c ≤ d → a + c ≤ b + d)
#check (add_le_add_left : a ≤ b → ∀ c, c + a ≤ c + b)
#check (add_le_add_right : a ≤ b → ∀ c, a + c ≤ b + c)
#check (add_lt_add_of_le_of_lt : a ≤ b → c < d → a + c < b + d)
#check (add_lt_add_of_lt_of_le : a < b → c ≤ d → a + c < b + d)
#check (add_lt_add_left : a < b → ∀ c, c + a < c + b)
#check (add_lt_add_right : a < b → ∀ c, a + c < b + c)
#check (add_nonneg : 0 ≤ a → 0 ≤ b → 0 ≤ a + b)
#check (add_pos : 0 < a → 0 < b → 0 < a + b)
#check (add_pos_of_pos_of_nonneg : 0 < a → 0 ≤ b → 0 < a + b)
#check (exp_pos : ∀ a, 0 < exp a)
#check add_le_add_left
```

`exp_le_exp` 、`exp_lt_exp` 和 `log_le_log` 等定理使用双向蕴含，表示“当且仅当”。(用 `\lr` 或者 `\iff` 输入)。我们将在下一章更详细地讨论这个连接词。`rw` 也可以处理双向蕴含，就像等号一样，将目标中匹配到的表达式右侧重写成左侧。

```
example (h : a ≤ b) : exp a ≤ exp b := by
  rw [exp_le_exp]
  exact h
```

实际上，表达式 `h : A ↔ B` 是 `A → B` 和 `B → A` 的合取，我们可以用 `h.mp` “肯定前件式”（modus ponens）指代 `A → B`，而 `h.mpr` “肯定前件式的反向”（modus ponens reverse）指代 `B → A`。另外还可以用 `h.1` 表示 `h.mp` 和用 `` `h.2` `` 表示 `h.mpr`，虽然这样或许会影响可读性。你可以考察下例：

```
example (h₀ : a ≤ b) (h₁ : c < d) : a + exp c + e < b + exp d + e := by
  apply add_lt_add_of_lt_of_le
  · apply add_lt_add_of_le_of_lt h₀
    apply exp_lt_exp.mpr h₁
  apply le_refl
```

第一行，`apply add_lt_add_of_lt_of_le` 创建了两个目标，我们再次使用点将两个证明分开。

试试下面的例子。中间的例子展示了 `norm_num` 策略可用于解决具体数字的问题。

```
example (h₀ : d ≤ e) : c + exp (a + d) ≤ c + exp (a + e) := by sorry

example : (0 : ℝ) < 1 := by norm_num

example (h : a ≤ b) : log (1 + exp a) ≤ log (1 + exp b) := by
  have h₀ : 0 < 1 + exp a := by sorry
  apply log_le_log h₀
  sorry
```

你也许体会到了，寻找你想要的库定理是形式化的重要环节。有以下方式：

- 在 GitHub 存储库 中浏览 Mathlib。
- 在 Mathlib 网页 上查询 API 文档。
- 根据 Mathlib 命名惯例和编辑器智能代码提示功能来猜测定理名称（有时需要手动用 `Ctrl-空格` 或 Mac 键盘上的 `Cmd-空格` 来开启自动补全）。Mathlib 的一种命名惯例是，定理 *A_of_B_of_C* 是以前提 B 和 C 推出 A，其中 A 、B 和 C 差不多是把表达式用人类语言朗读出来的样子（但经常会去掉变量名）。因此，形如 `x + y ≤ ...` 的定理可能会以 *add_le* 开头。键入 `add_le` 然后看看编辑器有没有好建议。请注意，按两次 `Ctrl-空格` 将显示更多可用的信息。（译注：快捷键可能无效，和 VS Code 设置有关，在弹出的提示菜单中可以看到右箭头，点击可展开完整信息。）
- VS Code 中，右键单击定理名称将显示一个菜单，其中包含 “转到定义”，你可以在附近找到类似的定理。
- 你可以使用 `apply?` 策略，这是一个 Mathlib 自带的定理搜索工具，它会自己尝试在库中找到相关的定理。

```
example : 0 ≤ a ^ 2 := by
  -- apply?
  exact sq_nonneg a
```

`--` 是注释行，你可以取消注释来试着用用 `apply?` ，然后你可以尝试用这个工具证明下面的例子：

```
example (h : a ≤ b) : c - exp b ≤ c - exp a := by
  sorry
```

你可以再试试用 `linarith` 而不是 `apply?` 来实现它。其他例子：

```
example : 2 * a * b ≤ a ^ 2 + b ^ 2 := by
  have h : 0 ≤ a ^ 2 - 2 * a * b + b ^ 2
  calc
    a ^ 2 - 2 * a * b + b ^ 2 = (a - b) ^ 2 := by ring
    _ ≥ 0 := by apply pow_two_nonneg
```

(continues on next page)

(continued from previous page)

```
  calc
    2 * a * b = 2 * a * b + 0 := by ring
    _ ≤ 2 * a * b + (a ^ 2 - 2 * a * b + b ^ 2) := add_le_add (le_refl _) h
    _ = a ^ 2 + b ^ 2 := by ring
```

Mathlib 习惯于在二元运算符如 `*` 和 `^` 旁边加空格，不过我不会干扰你的审美喜好。

有几个值得注意的地方。首先，表达式 `s ≥ t` 和 `t ≤ s` 定义等价，对人类来说可以互换，但是 Lean 的某些功能不理解这种等价（so sad），因此在 Mathlib 中习惯于使用 `≤` 而不是 `≥`。其次，`ring` 策略真好用！最后，注意到在第二个 `calc` 证明的第二行中使用了证明项 `add_le_add (le_refl _) h`，没必要写成 `by exact add_le_add (le_refl _) h`。

事实上，上例中的唯一需要人类智慧的地方就是找出假设 `h`，后面其实只涉及线性算术，都交给 `linarith`：

```
example : 2 * a * b ≤ a ^ 2 + b ^ 2 := by
  have h : 0 ≤ a ^ 2 - 2 * a * b + b ^ 2
  calc
    a ^ 2 - 2 * a * b + b ^ 2 = (a - b) ^ 2 := by ring
    _ ≥ 0 := by apply pow_two_nonneg
  linarith
```

好极了！下面是对你的挑战。你可以使用定理 `abs_le'.mpr`。你大概还会用到 `constructor` 策略将一个合取分解为两个目标；参见 Section 3.4。

```
example : |a * b| ≤ (a ^ 2 + b ^ 2) / 2 := by
  sorry

#check abs_le'.mpr
```

如果你连这都解决了，说明你马上要成为形式化大师了！

2.4 apply 和 rw 的更多例子

实数上的 `min` 函数是由以下三个事实定义的：

```
#check (min_le_left a b : min a b ≤ a)
#check (min_le_right a b : min a b ≤ b)
#check (le_min : c ≤ a → c ≤ b → c ≤ min a b)
```

你可以想象 `max` 也使用了类似的定义方式。

注意到我们使用 `min a b` 而不是 `min (a, b)` 来将 `min` 应用于一对参数 `a` 和 `b`。从形式上讲，`min` 是有两个参数的函数，类型为 `ℝ → ℝ → ℝ`，箭头是右结合的，也就是 `ℝ → (ℝ → ℝ)`。其实际效果是，如果 `a` 和 `b` 的

类型是 `ℝ`，那么 `min a` 的类型是 `ℝ → ℝ`，而 `min a b` 的类型是 `ℝ`。以这种方式处理多个参数被称为 **柯里化（currying）**，以逻辑学家 Haskell Curry 的名字命名。

在 Lean 中，运算的顺序也可能需要一些时间来适应。函数应用比中缀运算的优先级更高，因此表达式 `min a b + c` 被解释为 `(min a b) + c`。你会习惯的。

使用定理 `le_antisymm`，我们可以证明两个实数如果彼此小于或等于对方，则它们相等。利用这一点和上述的事实，我们可以证明 `min` 是可交换的：

```
example : min a b = min b a := by
  apply le_antisymm
  · show min a b ≤ min b a
    apply le_min
    · apply min_le_right
    apply min_le_left
  · show min b a ≤ min a b
    apply le_min
    · apply min_le_right
    apply min_le_left
```

这里我们使用点号来分隔不同目标的证明。我们的用法是不一致的：外层对两个目标都使用点和缩进，而内层只对第一个目标使用点。这两种约定都是合理且有用的。我们还使用了 `show` 策略，它只是展示目标，没它也一样，但这样更易于阅读和维护。

你可能觉得这个证明有两段是重复的。避免重复的一种方法是陈述一个局部引理，然后使用它：

```
example : min a b = min b a := by
  have h : ∀ x y : ℝ, min x y ≤ min y x := by
    intro x y
    apply le_min
    apply min_le_right
    apply min_le_left
  apply le_antisymm
  apply h
  apply h
```

我们将在 Section 3.1 中更多地讨论全称量词，但在这里只需说一下，假设 `h` 表示对于任意的 `x` 和 `y`，所需的不等式成立，`intro` 策略引入了任意的 `x` 和 `y` 来证明结论。在 `le_antisymm` 后的第一个 `apply` 隐式使用了 `h a b`，而第二个使用了 `h b a`。

另一个避免重复的方法是使用 `repeat` 策略，它将一个策略（或一个块）尽可能多次地应用。

```
example : min a b = min b a := by
  apply le_antisymm
  repeat
```

(continues on next page)

(continued from previous page)

```
    apply le_min
    apply min_le_right
    apply min_le_left
```

练习。你可以使用刚刚描述的任一技巧来缩短第一个证明。

```
example : max a b = max b a := by
  sorry
example : min (min a b) c = min a (min b c) := by
  sorry
```

你还可以尝试证明 `max` 的结合律。

有趣的是，`min` 在 `max` 上的分配律就像乘法对加法的分配律一样，反之亦然。换句话说，在实数上，我们有等式 `min a (max b c) ≤ max (min a b) (min a c)`，以及将 `max` 和 `min` 交换后的对应版本。但在下一节中，我们将看到这并 **不**是从 `≤` 的传递性和自反性以及上面列举的 `min` 和 `max` 的特性推导出来的。我们需要使用实数上的 `≤` 是 **全序**的事实，也就是说，它满足 `∀ x y, x ≤ y ∨ y ≤ x`。这里的析取符号，`∨`，代表“或”。在第一种情况下，我们有 `min x y = x`，而在第二种情况下，我们有 `min x y = y`。我们将在 Section 3.5 中学习如何根据情况来推理，但现在我们只会使用不需要拆分情况的例子。

例子：

```
theorem aux : min a b + c ≤ min (a + c) (b + c) := by
  sorry
example : min a b + c = min (a + c) (b + c) := by
  sorry
```

显然，`aux` 提供了证明等式所需的两个不等式中的一个，但巧妙地使用它也能证明另一个方向。提示，你可以使用定理 `add_neg_cancel_right` 和 `linarith` 策略。

Mathlib 库中的三角不等式体现了 Lean 的命名规则：

```
#check (abs_add : ∀ a b : ℝ, |a + b| ≤ |a| + |b|)
```

使用它和 `add_sub_cancel_right` 和``sub_add_cancel``来证明以下变形。最好用三行或更少的代码完成。

```
example : |a| - |b| ≤ |a - b| :=
  sorry
end
```

接下来，我们将考察可整除性 `x ∣ y`。可整除符号 **不**是你键盘上的普通的竖线，而是通过输入 `\|` 获得的 Unicode 字符。Mathlib 在定理名称中使用 `dvd` 来指代它。

```
example (h₀ : x ∣ y) (h₁ : y ∣ z) : x ∣ z :=
  dvd_trans h₀ h₁

example : x ∣ y * x * z := by
  apply dvd_mul_of_dvd_left
  apply dvd_mul_left

example : x ∣ x ^ 2 := by
  apply dvd_mul_left
```

在最后一个例子中，对于自然数次幂，应用 `dvd_mul_left` 会强制 Lean 将 `x^2` 的定义展开为 `x^1 * x`。

看看你能否找出证明以下内容所需的定理：

```
example (h : x ∣ w) : x ∣ y * (x * z) + x ^ 2 + w ^ 2 := by
  sorry
end
```

就可整除性而言，**最大公约数**（`gcd`）和最小公倍数（`lcm`）类似于 `min` 和 `max`。由于每个数都可以整除 `0`，因此 `0` 在可整除性的意义下表现得像是最大的元素：

```
variable (m n : ℕ)

#check (Nat.gcd_zero_right n : Nat.gcd n 0 = n)
#check (Nat.gcd_zero_left n : Nat.gcd 0 n = n)
#check (Nat.lcm_zero_right n : Nat.lcm n 0 = 0)
#check (Nat.lcm_zero_left n : Nat.lcm 0 n = 0)
```

试证明：

```
example : Nat.gcd m n = Nat.gcd n m := by
  sorry
```

提示：你可以使用 `dvd_antisymm`，但它有通用定理和专门针对自然数的两个版本，Lean 有可能会抱怨这里的歧义。此时你可以使用 `_root_.dvd_antisymm` 来明确指定使用通用版本，或者 `Nat.dvd_antisymm` 来指定使用自然数版本，任何一个都有效。

2.5 证明关于代数结构的命题

在 Section 2.2 中，我们看到许多关于实数的常见恒等式适用于更一般的代数结构类，比如交换环。我们还有其他代数结构，例如，**偏序**是一个具有二元关系的集合，该关系是自反的、传递的和反对称的，就像实数上的 $\leq$。

```
variable {α : Type*} [PartialOrder α]
variable (x y z : α)

#check x ≤ y
#check (le_refl x : x ≤ x)
#check (le_trans : x ≤ y → y ≤ z → x ≤ z)
#check (le_antisymm : x ≤ y → y ≤ x → x = y)
```

Mathlib 中习惯用希腊字母 `α`、`β` 和 `γ`（用 `\a`、`\b` 和 `\g` 输入）等等表示一般类型，使用 `R` 和 `G` 等来表示如环和群等特殊的代数结构。

对于任何偏序 `≤`，还有一个 **严格偏序** `<`，类似实数上的 `<`，是去掉等于的小于等于。

```
#check x < y
#check (lt_irrefl x : ¬ (x < x))
#check (lt_trans : x < y → y < z → x < z)
#check (lt_of_le_of_lt : x ≤ y → y < z → x < z)
#check (lt_of_lt_of_le : x < y → y ≤ z → x < z)

example : x < y ↔ x ≤ y ∧ x ≠ y :=
  lt_iff_le_and_ne
```

在这个例子中，符号 `∧` 表示“且”，符号 `¬` 表示“非”，而 `x ≠ y` 是 `¬ (x = y)` 的缩写。第 3 章 会讲到如何使用这些逻辑连接词来 **证明** `<` 具有所示的性质。

一个 **格**是给偏序添加运算符 `⊓` 和 `⊔` 的结构，这些运算符类似于实数上的 `min` 和 `max`，称为 **最大下界**和 **最小上界**（greatest lower bound, least upper bound），用 `\glb` 和 `\lub` 输入：

```
variable {α : Type*} [Lattice α]
variable (x y z : α)

#check x ⊓ y
#check (inf_le_left : x ⊓ y ≤ x)
#check (inf_le_right : x ⊓ y ≤ y)
#check (le_inf : z ≤ x → z ≤ y → z ≤ x ⊓ y)
#check x ⊔ y
#check (le_sup_left : x ≤ x ⊔ y)
#check (le_sup_right : y ≤ x ⊔ y)
#check (sup_le : x ≤ z → y ≤ z → x ⊔ y ≤ z)
```

这些符号通常也称为 **下确界**和 **上确界**，在 Mathlib 的定理名称中被记为 `inf` 和 `sup`。与此同时它们也经常被称为 *meet* 和 *join*。因此，如果你使用格，需要记住以下术语：

- `⊓` 是最大下界、下确界或 meet 。

- ⊔ 是最小上界、上确界或 join 。

一些格的实例包括：

- 任何全序上的 min 和 max，例如带有 ≤ 的整数或实数
- 某个域的子集合的 ∩ (\cap) 和 ∪ (\cup)，其中的序是 ⊆ (\subseteq)
- 布尔真值的 ∧ 和 ∨，其中的序是如果 x 是假或 y 是真，则 x ≤ y
- 自然数（或正自然数）上的 gcd 和 lcm，其中的序是 ∣
- 向量空间的线性子空间的集合，其中最大下界由交集给出，最小上界由两个空间的和给出，序是包含关系
- 一个集合（或在 Lean 中，一个类型）上的拓扑的集合，其中两个拓扑的最大下界由它们的并集生成的拓扑给出，最小上界是它们的交集，并且排序是逆包含关系

你可以验证，与 min / max 和 gcd / lcm 一样，你可以仅使用刻画它们的公理以及 le_refl 和 le_trans 来证明下确界和上确界的交换律和结合律。

```
example : x ⊓ y = y ⊓ x := by
  sorry

example : x ⊓ y ⊓ z = x ⊓ (y ⊓ z) := by
  sorry

example : x ⊔ y = y ⊔ x := by
  sorry

example : x ⊔ y ⊔ z = x ⊔ (y ⊔ z) := by
  sorry
```

在 Mathlib 中它们分别叫 inf_comm 、inf_assoc 、sup_comm 和 sup_assoc 。

另一个很好的练习是仅使用这些公理证明 **吸收律**：

```
theorem absorb1 : x ⊓ (x ⊔ y) = x := by
  sorry

theorem absorb2 : x ⊔ x ⊓ y = x := by
  sorry
```

在 Mathlib 中它们分别叫 inf_sup_self 和 sup_inf_self.

格上附加条件 x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z) 和 x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z) 称为 **分配格** (distributive lattice)。

```
variable {α : Type*} [DistribLattice α]
variable (x y z : α)

#check (inf_sup_left x y z : x ⊓ (y ⊔ z) = x ⊓ y ⊔ x ⊓ z)
#check (inf_sup_right x y z : (x ⊔ y) ⊓ z = x ⊓ z ⊔ y ⊓ z)
#check (sup_inf_left x y z : x ⊔ y ⊓ z = (x ⊔ y) ⊓ (x ⊔ z))
#check (sup_inf_right x y z : x ⊓ y ⊔ z = (x ⊔ z) ⊓ (y ⊔ z))
```

使用 ⊓ 和 ⊔ 的交换律易证左右两个版本等价。有兴趣的话你可以构造一个不分配的格。现在的目标是尝试证明在任何格中，一个分配律可以推导出另一个分配律。

```
variable {α : Type*} [Lattice α]
variable (a b c : α)

example (h : ∀ x y z : α, x ⊓ (y ⊔ z) = x ⊓ y ⊔ x ⊓ z) : a ⊔ b ⊓ c = (a ⊔ b) ⊓ (a ⊔␣
↪c) := by
  sorry

example (h : ∀ x y z : α, x ⊔ y ⊓ z = (x ⊔ y) ⊓ (x ⊔ z)) : a ⊓ (b ⊔ c) = a ⊓ b ⊔ a ⊓␣
↪c := by
  sorry
```

可以将结构公理组合成更大的结构。例如，**严格有序环**是一个环上附加偏序，且满足环运算与序是兼容的：

```
variable {R : Type*} [StrictOrderedRing R]
variable (a b c : R)

#check (add_le_add_left : a ≤ b → ∀ c, c + a ≤ c + b)
#check (mul_pos : 0 < a → 0 < b → 0 < a * b)
```

第 3 章 将从 `mul_pos` 和 < 的定义中得出以下定理：

```
#check (mul_nonneg : 0 ≤ a → 0 ≤ b → 0 ≤ a * b)
```

下面是一个拓展练习，展示了任何有序环上的算术和序的常见引理。希望你仅使用环、偏序以及上面两个示例中列举的事实来证明它们（环可能并非交换，因此不能用 `ring` 策略）：

```
example (h : a ≤ b) : 0 ≤ b - a := by
  sorry

example (h: 0 ≤ b - a) : a ≤ b := by
  sorry
```

(continues on next page)

(continued from previous page)

```
example (h : a ≤ b) (h' : 0 ≤ c) : a * c ≤ b * c := by
  sorry
```

最后，**度量空间**是配备了距离 `dist x y` 的集合，距离将任何一对元素映射到一个实数，满足以下公理：

```
variable {X : Type*} [MetricSpace X]
variable (x y z : X)

#check (dist_self x : dist x x = 0)
#check (dist_comm x y : dist x y = dist y x)
#check (dist_triangle x y z : dist x z ≤ dist x y + dist y z)
```

证明距离始终是非负的。你可能会用到 `nonneg_of_mul_nonneg_left` 。在 Mathlib 中这个定理被称为 `dist_nonneg` 。

```
example (x y : X) : 0 ≤ dist x y := by
  sorry
```

CHAPTER

THREE

逻辑

上一节中，我们处理了等式、不等式和类似与“x 整除 y”这样的命题。复杂的数学命题是由简单的命题通过逻辑连接词组装起来的，例如“且”、“或”、“非”、“如果……那么……”、“所有……”和“存在……”。本章我们会处理这些使用到逻辑的命题。

3.1 蕴含和全称量词

考察 `#check` 后面的语句：

```
#check ∀ x : ℝ, 0 ≤ x → |x| = x
```

自然语言表述为“对于每个实数 `x`，若 `0 ≤ x`，则 `x` 的绝对值等于 `x`”。我们还可以有更复杂的语句，例如：

```
#check ∀ x y ε : ℝ, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε
```

自然语言表述为“对于任意的 `x`, `y`, 以及 `ε`，若 `0 < ε ≤ 1`, `x` 的绝对值小于 `ε`, 且 `y` 的绝对值小于 `ε`, 则 `x * y` 的绝对值小于 `ε`.”在 Lean 中，蕴含是右结合的。所以上述表达式的意思是“若 `0 < ε`, 则若 `ε ≤ 1`, 则若 `|x| < ε` …”因此，表达式表示所有假设组合在一起导出结论。

尽管这个语句中全称量词的取值范围是对象，而蕴涵箭头引入的是假设，但 Lean 处理这两者的方式非常相似。如果你已经证明了这种形式的定理，你就可以用同样的方法把它应用于对象和假设。让我们以它为例，稍后会证明这个例子。

```
theorem my_lemma : ∀ x y ε : ℝ, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε :=
  sorry

section
variable (a b δ : ℝ)
variable (h₀ : 0 < δ) (h₁ : δ ≤ 1)
variable (ha : |a| < δ) (hb : |b| < δ)

#check my_lemma a b δ
```

(continues on next page)

(continued from previous page)

```
#check my_lemma a b δ h₀ h₁
#check my_lemma a b δ h₀ h₁ ha hb

end
```

Lean 中，当被量词限定的变量可以从后面的假设中推断出来时，常常使用花括号将其设为隐式，这样一来我们就可以直接将引理应用到假设中，而无需提及对象。

```
theorem my_lemma2 : ∀ {x y ε : ℝ}, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε :=
  sorry

section
variable (a b δ : ℝ)
variable (h₀ : 0 < δ) (h₁ : δ ≤ 1)
variable (ha : |a| < δ) (hb : |b| < δ)

#check my_lemma2 h₀ h₁ ha hb

end
```

如果使用 `apply` 策略将 `my_lemma` 应用于形如 `|a * b| < δ` 的目标，就会把引理的每个假设作为新的目标。

要证明引理，需使用 `intro` 策略。

```
theorem my_lemma3 :
    ∀ {x y ε : ℝ}, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε := by
  intro x y ε epos ele1 xlt ylt
  sorry
```

引入被全称量词限定的变量时可以使用任何名字，并非一定要是 `x`, `y` 和 `ε`。隐式变量也必须被引入，因为隐式变量的意思是，当我们 **使用** `my_lemma` 时可以不写它们，但证明时还得写，因为它组成了待证命题。下面将看到为什么有时候有必要在证明开始之后引入变量和假设。

你来证明后边的部分。提示：可能会用到 `abs_mul`, `mul_le_mul`, `abs_nonneg`, `mul_lt_mul_right` 和 `one_mul`。另外回忆知识点：你可以使用 `.mp` 和 `.mpr` 或者 `.1` 和 `.2` 来提取一个当且仅当语句的两个方向。

```
theorem my_lemma4 :
    ∀ {x y ε : ℝ}, 0 < ε → ε ≤ 1 → |x| < ε → |y| < ε → |x * y| < ε := by
  intro x y ε epos ele1 xlt ylt
  calc
    |x * y| = |x| * |y| := sorry
    _ ≤ |x| * ε := sorry
```

(continues on next page)

(continued from previous page)

```
    _ < 1 * ε := sorry
    _ = ε := sorry
```

全称量词通常隐藏在定义中，Lean 会在必要时展开定义以暴露它们。例如，让我们定义两个谓词，`FnUb f a` 和 `FnLb f a`，其中 `f` 是一个从实数到实数的函数，而 `a` 是一个实数。第一个谓词是说 `a` 是 `f` 的值的一个上界，而第二个是说 `a` 是 `f` 的值的一个下界。

```
def FnUb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, f x ≤ a

def FnLb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, a ≤ f x
```

在下一个例子中，`fun x ↦ f x + g x` 是把 `x` 映射到 ``f x + g x`` 的函数。从表达式 `f x + g x` 构造这个函数的过程在类型论中称为 lambda 抽象（lambda abstraction）。

```
example (hfa : FnUb f a) (hgb : FnUb g b) : FnUb (fun x ↦ f x + g x) (a + b) := by
  intro x
  dsimp
  apply add_le_add
  apply hfa
  apply hgb
```

把 `intro` 应用于目标 `FnUb (fun x ↦ f x + g x) (a + b)` 迫使 Lean 展开 `FnUb` 的定义并从全称量词中引入 `x`。此时目标为 `(fun (x : R) ↦ f x + g x) x ≤ a + b`。把 `(fun x ↦ f x + g x)` 应用到 `x` 上（apply func to var，类型论常用话术，指的就是把 var 代入 func 表达式）的结果应该是 `f x + g x`，`dsimp` 命令执行了这个化简（或者说代入取值）。（其中“d”是指“定义性的”）其实你可以删除这个命令，证明仍然有效；Lean 将不得不自行执行化简，才能使下一个 `apply` 有意义。`dsimp` 命令有更好的可读性。另一种选择是使用 `change` 策略，通过输入 `change f x + g x ≤ a + b`。这也有助于提高证明的可读性，并让你对目标的转换方式有更多的控制。

证明的其余部分都是例行公事。最后两条 `apply` 命令迫使 Lean 展开假设中 `FnUb` 的定义。请尝试进行类似的证明：

```
example (hfa : FnLb f a) (hgb : FnLb g b) : FnLb (fun x ↦ f x + g x) (a + b) :=
  sorry

example (nnf : FnLb f 0) (nng : FnLb g 0) : FnLb (fun x ↦ f x * g x) 0 :=
  sorry

example (hfa : FnUb f a) (hgb : FnUb g b) (nng : FnLb g 0) (nna : 0 ≤ a) :
    FnUb (fun x ↦ f x * g x) (a * b) :=
```

(continues on next page)

(continued from previous page)

```
  sorry
```

虽然我们已对从实数到实数的函数定义了 `FnUb` 和 `FnLb`，但这些定义和证明完全可推广到对任何两个有序结构的类型之间的函数。检查定理 `add_le_add` 的类型，发现它对任何是“有序加法交换幺半群”的结构成立；你可以不在乎它具体是什么，但自然数、整数、有理数和实数都属于这种情况。因此，如果我们能在如此广泛的东西上证明定理 `fnUb_add`，那么它将可用于所有这些实例中。

```
variable {α : Type*} {R : Type*} [OrderedCancelAddCommMonoid R]

#check add_le_add

def FnUb' (f : α → R) (a : R) : Prop :=
  ∀ x, f x ≤ a

theorem fnUb_add {f g : α → R} {a b : R} (hfa : FnUb' f a) (hgb : FnUb' g b) :
    FnUb' (fun x ↦ f x + g x) (a + b) := fun x ↦ add_le_add (hfa x) (hgb x)
```

你在 Section 2.2 中已经看到过像这样的方括号，但我们仍未解释它们的含义。为了具体性，在我们的大多数例子中，我们专注于实数，但 Mathlib 包含一些具有更通用的定义和定理。

再举一个隐藏全称量词的例子，Mathlib 定义了一个谓词 `Monotone`，表示函数相对于参数是非递减的：

```
example (f : R → R) (h : Monotone f) : ∀ {a b}, a ≤ b → f a ≤ f b :=
  @h
```

性质 `Monotone f` 的定义完全就是冒号后的表达式。我们需要在 `h` 之前输入 `@` 符号，不然 Lean 会展开 `h` 的隐式参数并插入占位符。

证明有关单调性的语句需要使用 `intro` 引入两个变量，例如 `a` 和 `b`, 以及假设 `a ≤ b`。

```
example (mf : Monotone f) (mg : Monotone g) : Monotone fun x ↦ f x + g x := by
  intro a b aleb
  apply add_le_add
  apply mf aleb
  apply mg aleb
```

用证明项给出简短的证明往往更方便。描述一个临时引入对象 `a` 和 `b` 以及假设 `aleb` 的证明时，Lean 使用符号 `fun a b aleb ↦ ...`。这类似于用 `fun x ↦ x^2` 这样的表达式来描述一个函数时，先暂时命名一个对象 `x`, 然后用它来描述函数的值。因此，上一个证明中的 `intro` 命令对应于下一个证明项中的 lambda 抽象。`apply` 命令则对应于构建定理对其参数的应用。

```
example (mf : Monotone f) (mg : Monotone g) : Monotone fun x ↦ f x + g x :=
  fun a b aleb ↦ add_le_add (mf aleb) (mg aleb)
```

技巧：如果在开始编写证明项 `fun a b aleb ↦ _` 时，在表达式的其余部分使用下划线，Lean 就会提示错误，表明它无法猜测该表达式的值。如果你检查 VS Code 中的 Lean 目标窗口或把鼠标悬停在标着波浪线的错误标识符上，Lean 会向你显示剩余的表达式需要解决的目标。

尝试用策略或证明项证明它们：

```
example {c : R} (mf : Monotone f) (nnc : 0 ≤ c) : Monotone fun x ↦ c * f x :=
  sorry

example (mf : Monotone f) (mg : Monotone g) : Monotone fun x ↦ f (g x) :=
  sorry
```

其他例子。一个从 $\mathbb{R}$ 到 $\mathbb{R}$ 的函数 f，如果对每个 x, 有 $f(-x) = f(x)$ 则称为 *偶函数*，如果对每个 x, 有 $f(-x) = -f(x)$ 则称为 *奇函数*。下面的例子形式化地定义了这两个概念，证明了一条性质。你来证明其他性质。

```
def FnEven (f : ℝ → ℝ) : Prop :=
  ∀ x, f x = f (-x)

def FnOdd (f : ℝ → ℝ) : Prop :=
  ∀ x, f x = -f (-x)

example (ef : FnEven f) (eg : FnEven g) : FnEven fun x ↦ f x + g x := by
  intro x
  calc
    (fun x ↦ f x + g x) x = f x + g x := rfl
    _ = f (-x) + g (-x) := by rw [ef, eg]


example (of : FnOdd f) (og : FnOdd g) : FnEven fun x ↦ f x * g x := by
  sorry

example (ef : FnEven f) (og : FnOdd g) : FnOdd fun x ↦ f x * g x := by
  sorry

example (ef : FnEven f) (og : FnOdd g) : FnEven fun x ↦ f (g x) := by
  sorry
```

通过使用 `dsimp` 或 `change` 化简 lambda 抽象，可以缩短第一个证明。但你可以验证，除非我们准确地消除 lambda 抽象，否则接下来的 `rw` 不会生效，因为此时它无法在表达式中找到模式 `f x` 和 `g x`。和其他一些策略不同，`rw` 作用于语法层次，它不会为你展开定义或应用还原（它有一个变种称为 `erw`，在这个方向上会努力一点，但也不会努力太多）。

到处都能找到隐式全称量词，只要你知道如何发现它们。

Mathlib 包含一个用于操作集合的优秀的库。回想一下，Lean 并不使用基于集合论的基础，我们在此使用朴素的集合含义，即某个给定类型 `α` 的数学对象的汇集。如果 `x` 具有类型 `α`，而 `s` 具有类型 `Set α`，则 `x ∈ s` 是一个命题，它断言 `x` 是 `s` 的一个元素。若 `y` 具有另一个类型 `β` 则表达式 `y ∈ s` 无意义。这里“无意义”的含义是“没有类型因此 Lean 不认可它是一个形式良好的语句”。这与 Zermelo-Fraenkel 集合论不同，例如其中对于每个数学对象 `a` 和 `b`，`a ∈ b` 都是形式良好的语句。例如，`sin ∈ cos` 是 ZF 中一个形式良好的语句。集合论基础的这一缺陷是证明助手中不使用它的一个重要原因，因为证明助手的目的是帮助我们检出无意义的表达式。在 Lean 中，`sin` 具有类型 `ℝ → ℝ`，而 `cos` 具有类型 `ℝ → ℝ`，它不等于 `Set (ℝ → ℝ)`，即使在展开定义以后也不相等，因此语句 `sin ∈ cos` 无意义。我们还可以利用 Lean 来研究集合论本身。例如，连续统假设与 Zermelo-Fraenkel 公理的独立性就在 Lean 中得到了形式化。但这种集合论的元理论完全超出了本书的范围。

若 `s` 和 `t` 具有类型 `Set α`，那么子集关系 `s ⊆ t` 被定义为 `∀ {x : α}, x ∈ s → x ∈ t`。量词中的变量被标记为隐式，因此给出 `h : s ⊆ t` 和 `h' : x ∈ s`，我们可以把 `h h'` 作为 `x ∈ t` 的证明。下面的例子使用策略证明和证明项，证明了子集关系的反身性，你来类似地证明传递性。

```
variable {α : Type*} (r s t : Set α)

example : s ⊆ s := by
  intro x xs
  exact xs

theorem Subset.refl : s ⊆ s := fun x xs ↦ xs

theorem Subset.trans : r ⊆ s → s ⊆ t → r ⊆ t := by
  sorry
```

就像我们对函数定义了 `FnUb` 一样，假设 `s` 是一个由某类型的元素组成的集合，且它具有一个与之关联的序。我们可以定义 `SetUb s a`，意为 `a` 是集合 `s` 的一个上界。在下一个例子中，证明如果 `a` 是 `s` 的一个上界，且 `a ≤ b`，则 `b` 也是 `s` 的一个上界。

```
variable {α : Type*} [PartialOrder α]
variable (s : Set α) (a b : α)

def SetUb (s : Set α) (a : α) :=
  ∀ x, x ∈ s → x ≤ a

example (h : SetUb s a) (h' : a ≤ b) : SetUb s b :=
  sorry
```

最后，我们以一个重要的例子来结束本节。函数 f 称为 **单射**，若对每个 x_1 和 x_2，如果 $f(x_1) = f(x_2)$，那么 $x_1 = x_2$。Mathlib 定义了 `Function.Injective f`，其中 x_1 和 x_2 是隐式的。下一个例子表明，在实数上，任何由自变量加上非零常数得到的函数都是单射。然后，你可以利用示例中的引理名称作为灵感来源，证明非零常数乘法也是单射的。

```
open Function

example (c : ℝ) : Injective fun x ↦ x + c := by
  intro x₁ x₂ h'
  exact (add_left_inj c).mp h'

example {c : ℝ} (h : c ≠ 0) : Injective fun x ↦ c * x := by
  sorry
```

最后，证明两个单射函数的复合是单射：

```
variable {α : Type*} {β : Type*} {γ : Type*}
variable {g : β → γ} {f : α → β}

example (injg : Injective g) (injf : Injective f) : Injective fun x ↦ g (f x) := by
  sorry
```

3.2 存在量词

存在量词 ∃ （\ex ）用于表示短语 “存在”。Lean 中的形式表达式 `∃ x : ℝ, 2 < x ∧ x < 3` 是说存在一个介于 2 到 3 之间的实数。（我们将在 Section 3.4 探讨合取符号。）证明这种语句的典型方式是给出一个实数并说明它具有语句指出的性质。我们可以用 `5 / 2` 输入 2.5 这个数，或者，当 Lean 无法从上下文推断出我们想输入实数时，用 `(5 : ℝ) / 2` 输入，它具有所需的性质，而 `norm_num` 策略可以证明它符合描述。

我们有一些方式可以把信息聚合在一起。给定一个以存在量词开头的目标，则 `use` 策略可用于提供对象，留下证明其属性的目标。

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 := by
  use 5 / 2
  norm_num
```

你不仅可以给 `use` 策略提供数据，还可以提供证明：

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 := by
  have h1 : 2 < (5 : ℝ) / 2 := by norm_num
  have h2 : (5 : ℝ) / 2 < 3 := by norm_num
  use 5 / 2, h1, h2
```

事实上，`use` 策略同样自动地尝试可用的假设。

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 := by
  have h : 2 < (5 : ℝ) / 2 ∧ (5 : ℝ) / 2 < 3 := by norm_num
```

(continues on next page)

(continued from previous page)

```
  use 5 / 2
```

或者，我们可以使用 Lean 的 **匿名构造子（anonymous constructor）**符号来构造涉及存在量词的证明。

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
  have h : 2 < (5 : ℝ) / 2 ∧ (5 : ℝ) / 2 < 3 := by norm_num
  ⟨5 / 2, h⟩
```

这里没有用 `by`，因为这是证明项。左右尖括号，可以分别用 `\<` 和 `\>` 输入，告诉 Lean 使用任何对当前目标合适的构造方式把给定的数据组织起来。我们可以不在策略模式下使用这种符号：

```
example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
  ⟨5 / 2, by norm_num⟩
```

所以我们现在知道如何证明一个存在语句。但我们要如何 *使用*它？如果我们知道存在一个具有特定性质的对象，我们就可以为任意一个对象命名并对其进行推理。例如，回顾上一节的谓词 `FnUb f a` 和 `FnLb f a`，它们分别是指 `a` 是 `f` 的一个上界或下界。我们可以使用存在量词说明“`f` 是有界的”，而无需指定它的界：

```
def FnUb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, f x ≤ a

def FnLb (f : ℝ → ℝ) (a : ℝ) : Prop :=
  ∀ x, a ≤ f x

def FnHasUb (f : ℝ → ℝ) :=
  ∃ a, FnUb f a

def FnHasLb (f : ℝ → ℝ) :=
  ∃ a, FnLb f a
```

我们可以使用上一节的定理 `FnUb_add` 证明若 `f` 和 `g` 具有上界，则 `fun x ↦ f x + g x` 也有。

```
variable {f g : ℝ → ℝ}

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  rcases ubf with ⟨a, ubfa⟩
  rcases ubg with ⟨b, ubgb⟩
  use a + b
  apply fnUb_add ubfa ubgb
```

`rcases` 策略解包了存在量词中的信息。像 `⟨a, ubfa⟩` 这样长得像是一个匿名构造子的记号，称为 **模式（pattern）**，它们描述了我们在解包主参数时期望找到的信息。给出假设 `ubf`，即 `f` 存在上界，`rcases ubf`

`with ⟨a, ubfa⟩` 在上下文中添加一个新变量 `a` 作为上界，并添加假设 `ubfa`，即该变量有给定的性质。这有点像 `intro`，目标没有变化，但引入了新对象和新假设来证明目标。这是数学推理的一种常规方法：我们解包对象，其存在性被一些假设断言或蕴含，然后使用它论证其他一些东西的存在性。

试着使用这个方法构建下列命题。你可能用到上一节中的一些例子，所以可以给它们起些名字，就像我们对 `fn_ub_add` 做的那样，或者你也可以直接把那些论证插入到证明中。

```
example (lbf : FnHasLb f) (lbg : FnHasLb g) : FnHasLb fun x ↦ f x + g x := by
  sorry

example {c : ℝ} (ubf : FnHasUb f) (h : c ≥ 0) : FnHasUb fun x ↦ c * f x := by
  sorry
```

`rcases` 中的“r”表示“recursive（递归）”，因为它允许我们使用任意复杂的模式解包嵌套数据。`rintro` 策略是 `intro` 和 `rcases` 的组合：

```
example : FnHasUb f → FnHasUb g → FnHasUb fun x ↦ f x + g x := by
  rintro ⟨a, ubfa⟩ ⟨b, ubgb⟩
  exact ⟨a + b, fnUb_add ubfa ubgb⟩
```

事实上，Lean 也支持表达式和证明项中的模式匹配函数：

```
example : FnHasUb f → FnHasUb g → FnHasUb fun x ↦ f x + g x :=
  fun ⟨a, ubfa⟩ ⟨b, ubgb⟩ ↦ ⟨a + b, fnUb_add ubfa ubgb⟩
```

在假设中解包信息的任务非常重要，以至于 Lean 和 Mathlib 提供了多种方式实施。例如，`obtain` 策略提供提示性语法：

```
example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  obtain ⟨a, ubfa⟩ := ubf
  obtain ⟨b, ubgb⟩ := ubg
  exact ⟨a + b, fnUb_add ubfa ubgb⟩
```

将第一条 `obtain` 指令看作是将 `ubf` 的“内容”与给定的模式匹配，并将成分赋值给具名变量。`rcases` 和 `obtain` 可以说是在 `destruct` 它们的参数，但有一点不同，`rcases` 在完成后会清除上下文中的 `ubf`，而在 `obtain` 后它仍然存在。

Lean 还支持与其他函数式编程语言类似的语法：

```
example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  cases ubf
  case intro a ubfa =>
    cases ubg
    case intro b ubgb =>
```

(continues on next page)

(continued from previous page)

```
      exact ⟨a + b, fnUb_add ubfa ubgb⟩

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  cases ubf
  next a ubfa =>
    cases ubg
    next b ubgb =>
      exact ⟨a + b, fnUb_add ubfa ubgb⟩

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  match ubf, ubg with
    | ⟨a, ubfa⟩, ⟨b, ubgb⟩ =>
      exact ⟨a + b, fnUb_add ubfa ubgb⟩

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x :=
  match ubf, ubg with
    | ⟨a, ubfa⟩, ⟨b, ubgb⟩ =>
      ⟨a + b, fnUb_add ubfa ubgb⟩
```

在第一个例子中，如果把光标放在 `cases ubf` 后面，就会看到该策略产生了一个目标，Lean 将其标记为 `intro`（所选的特定名称来自建立存在性语句证明的公理基元的内部名称）。然后，`case` 策略会命名各个组件。第二个例子也是类似的，只是使用了 `next` 而非 `case` 意味着可以避免提及 `intro`。最后两个例子中的 `match` 一词强调了我们在这里做的是计算机科学家所说的“模式匹配”。请注意，第三个证明以 `by` 开头，之后的策略版 `match` 希望在箭头右侧有一个策略证明。最后一个例子是一个证明项，没有策略。

在本书其余部分，我们将坚持使用 `rcases`, `rintros` 和 `obtain`，作为使用存在量词的首选方式。但看看其他语法也没坏处，尤其是当你有机会与计算机科学家合作时。

为了展示 `rcases` 的一种使用方法，我们来证明一个经典的数学结果：若两个整数 `x` 和 `y` 可以分别写成两个平方数之和，那么它们的乘积 `x * y` 也可以。事实上，这个结论对任何交换环都是正确的，而不仅仅适用于整数环。在下一个例子中，`rcases` 同时解包了两个存在量词。然后，我们以列表形式向 `use` 语句提供将 `x * y` 表示为平方和所需的值，并使用 `ring` 来验证它们是否有效。

```
variable {α : Type*} [CommRing α]

def SumOfSquares (x : α) :=
  ∃ a b, x = a ^ 2 + b ^ 2

theorem sumOfSquares_mul {x y : α} (sosx : SumOfSquares x) (sosy : SumOfSquares y) :
    SumOfSquares (x * y) := by
  rcases sosx with ⟨a, b, xeq⟩
  rcases sosy with ⟨c, d, yeq⟩
```

(continues on next page)

(continued from previous page)

```
  rw [xeq, yeq]
  use a * c - b * d, a * d + b * c
  ring
```

这个证明并未给出太多直觉洞见，现在我们展示证明思路。**高斯整数**是形如 $a + bi$ 的数，其中 a 和 b 是整数，而 $i = \sqrt{-1}$ 。根据定义，高斯整数 $a + bi$ 的 *范数* 是 $a^2 + b^2$ 。所以高斯整数的范数是平方和，且任意平方和都可以这样表示。上述定理反映了一个事实，即高斯整数的乘积的范数等于范数的乘积：若 x 是 $a + bi$ 的范数，且 y 是 $c + di$ 的范数，则 xy 是 $(a + bi)(c + di)$ 的范数。我们充满惊人注意力的证明说明了这样一个事实：最容易形式化的证明并不总是最透彻的。在 Section 6.3 中，我们将为你介绍定义高斯整数的方法，并利用它们提供另一种证明。

在存在量词中解包等式，然后用它来重写目标中的表达式的模式经常出现，以至于 `rcases` 策略提供了一个缩写：如果使用关键字 `rfl` 代替新的标识符，`rcases` 就会自动进行重写（这一技巧不适用于模式匹配的 lambda）。

```
theorem sumOfSquares_mul' {x y : α} (sosx : SumOfSquares x) (sosy : SumOfSquares y) :
    SumOfSquares (x * y) := by
  rcases sosx with ⟨a, b, rfl⟩
  rcases sosy with ⟨c, d, rfl⟩
  use a * c - b * d, a * d + b * c
  ring
```

与全称量词一样，存在量词也满地都是。例如，可除性隐含了一个“存在”语句。

```
example (divab : a ∣ b) (divbc : b ∣ c) : a ∣ c := by
  rcases divab with ⟨d, beq⟩
  rcases divbc with ⟨e, ceq⟩
  rw [ceq, beq]
  use d * e; ring
```

这再次为和 `rfl` 一起使用 `rcases` 提供了一个很好的配置。在上面的证明中试试看。感觉还不错！

接下来尝试证明下列定理：

```
example (divab : a ∣ b) (divac : a ∣ c) : a ∣ b + c := by
  sorry
```

另一个重要的例子是，若函数 $f : \alpha \to \beta$ 满足对值域 β 中任意的 y, 存在定义域 α 中的 x, 使得 $f(x) = y$ ，那么我们称这个函数是 *满射*。注意到这个语句既包含全称量词，也包含存在量词，这解释了为什么接下来的例子同时使用了 `intro` 和 `use` 。

```
example {c : ℝ} : Surjective fun x ↦ x + c := by
  intro x
```

(continues on next page)

(continued from previous page)

```
  use x - c
  dsimp; ring
```

试试这个，使用定理 `mul_div_cancel`。：

```
example {c : ℝ} (h : c ≠ 0) : Surjective fun x ↦ c * x := by
  sorry
```

技巧：`field_simp` 策略通常可以有效地去分母。它可以与 `ring` 策略结合使用。

```
example (x y : ℝ) (h : x - y ≠ 0) : (x ^ 2 - y ^ 2) / (x - y) = x + y := by
  field_simp [h]
  ring
```

下一个示例使用了满射性假设，将它应用于一个合适的值。你可以在任何表达式中使用 `rcases`，而不仅仅是在假设中。

```
example {f : ℝ → ℝ} (h : Surjective f) : ∃ x, f x ^ 2 = 4 := by
  rcases h 2 with ⟨x, hx⟩
  use x
  rw [hx]
  norm_num
```

看看你能否用这些方法证明满射函数的复合是满射。

```
variable {α : Type*} {β : Type*} {γ : Type*}
variable {g : β → γ} {f : α → β}

example (surjg : Surjective g) (surjf : Surjective f) : Surjective fun x ↦ g (f x) :=␣
↪by
  sorry
```

3.3 否定

符号 `¬`（`\n`，`\neg`）表示否定，所以 `¬ x < y` 是说 `x` 不小于 `y`，`¬ x = y`（或者，等价地，`x ≠ y`）是说 `x` 不等于 `y`，而 `¬ ∃ z, x < z ∧ z < y` 是说不存在 `z` 使其严格位于 `x` 和 `y` 之间。在 Lean 中，符号 `¬ A` 是 `A → False` 的缩写，你可以认为它表示 `A` 导出矛盾。聪明的你会发现，你可以通过引入假设 `h : A` 并证明 `False` 来证明 `¬ A`，如果你有 `h : ¬ A` 和 `h' : A`，那么把 `h` 应用于 `h'` 就产生 `False`.

为了演示，考虑严格序的反自反律 `lt_irrefl`，它是说对每个 `a` 我们有 `¬ a < a`。反对称律 `lt_asymm` 是说我们有 `a < b → ¬ b < a`。我们证明，`lt_asymm` 可从 `lt_irrefl` 推出。

```
example (h : a < b) : ¬b < a := by
  intro h'
  have : a < a := lt_trans h h'
  apply lt_irrefl a this
```

这个例子引入了两个新技巧。第一，当我们使用 `have` 而不提供名字时，Lean 使用名字 `this` 。因为这个证明太短，我们提供了精确证明项。但你真正需要注意的是这个证明中 `intro` 策略的结果，它留下目标 `False` ，还有，我们最终对 `a < a` 使用 `lt_irrefl` 证明了 `False` 。

这里是另一个例子，它使用上一节定义的谓词 `FnHasUb` ，也就是说一个函数有上界。

```
example (h : ∀ a, ∃ x, f x > a) : ¬FnHasUb f := by
  intro fnub
  rcases fnub with ⟨a, fnuba⟩
  rcases h a with ⟨x, hx⟩
  have : f x ≤ a := fnuba x
  linarith
```

当目标可由语境中的线性等式和不等式得出时，使用 `linarith` 通常很方便。

类似地证明下列定理：

```
example (h : ∀ a, ∃ x, f x < a) : ¬FnHasLb f :=
  sorry

example : ¬FnHasUb fun x ↦ x :=
  sorry
```

Mathlib 提供了一些关于序和否定的定理：

```
#check (not_le_of_gt : a > b → ¬a ≤ b)
#check (not_lt_of_ge : a ≥ b → ¬a < b)
#check (lt_of_not_ge : ¬a ≥ b → a < b)
#check (le_of_not_gt : ¬a > b → a ≤ b)
```

回顾谓词 `Monotone f` ，它表示 `f` 是非递减的。用刚才列举的一些定理证明下面的内容：

```
example (h : Monotone f) (h' : f a < f b) : a < b := by
  sorry

example (h : a ≤ b) (h' : f b < f a) : ¬Monotone f := by
  sorry
```

我们可以说明，如果我们把 < 换成 ≤ ，则上一段的第一个例子无法被证明。我们可以通过给出反例证明一个

全称量词语句的否定。接下来完成证明：

```
example : ¬∀ {f : ℝ → ℝ}, Monotone f → ∀ {a b}, f a ≤ f b → a ≤ b := by
  intro h
  let f := fun x : ℝ ↦ (0 : ℝ)
  have monof : Monotone f := by sorry
  have h' : f 1 ≤ f 0 := le_refl _
  sorry
```

这个例子引入了 `let` 策略，它向语境添加了一个 **局部定义**。如果你把光标移动到目标窗口的 `let` 命令后面的地方，你会看到 `f : ℝ → ℝ := fun x ↦ 0` 已经被添加到语境中。当必须展开定义时，Lean 会展开。特别地，当我们使用 `le_refl` 证明 `f 1 ≤ f 0` 时，Lean 把 `f 1` 和 `f 0` 约化为 `0`.

使用 `le_of_not_gt` 证明下列内容：

```
example (x : ℝ) (h : ∀ ε > 0, x < ε) : x ≤ 0 := by
  sorry
```

我们刚才所做的许多证明都隐含着这样一个事实：如果 `P` 是任何属性，说没有任何事物具有 `P` 属性就等于说一切事物都不具有 `P` 属性，而说并非所有东西都具有 `P` 属性等同于说某些东西不具备 `P` 属性。换句话说，以下四个推理都是有效的（但其中有一个无法使用我们目前已讲解的内容证明）：

```
variable {α : Type*} (P : α → Prop) (Q : Prop)

example (h : ¬∃ x, P x) : ∀ x, ¬P x := by
  sorry

example (h : ∀ x, ¬P x) : ¬∃ x, P x := by
  sorry

example (h : ¬∀ x, P x) : ∃ x, ¬P x := by
  sorry

example (h : ∃ x, ¬P x) : ¬∀ x, P x := by
  sorry
```

第一、二、四个可以使用你已经学到的方法直接证明。鼓励你尝试。然而，第三个更为困难，因为它是从一个对象的不存在可以得出矛盾的这一事实中得出结论，认为该对象是存在的。这是 **经典**数学推理的一个实例。我们可以像下面这样使用反证法证明第三个结果。

```
example (h : ¬∀ x, P x) : ∃ x, ¬P x := by
  by_contra h'
  apply h
```

(continues on next page)

(continued from previous page)

```
  intro x
  show P x
  by_contra h''
  exact h' ⟨x, h''⟩
```

确保你搞懂了示例。`by_contra` 策略允许我们通过假定 `¬ Q` 推出矛盾来证明目标 `Q`。事实上，它等价于使用等价关系 `not_not : ¬ ¬ Q ↔ Q`。确认一下你可以使用 `by_contra` 证明这个等价的正方向，而反方向可从常规的否定法则得出。

```
example (h : ¬¬Q) : Q := by
  sorry

example (h : Q) : ¬¬Q := by
  sorry
```

用反证法证明下面的内容，它是我们在上面证明的一个蕴涵的相反方向。(提示：先使用 `intro`)

```
example (h : ¬FnHasUb f) : ∀ a, ∃ x, f x > a := by
  sorry
```

处理前面带有否定词的复合语句通常很无趣，通常我们希望会把否定词放进里面。好消息，Mathlib 提供了 `push_neg` 策略来找到否定词在里面的等价命题。命令 `push_neg at h` 重述假设 `h`。

```
example (h : ¬∀ a, ∃ x, f x > a) : FnHasUb f := by
  push_neg at h
  exact h

example (h : ¬FnHasUb f) : ∀ a, ∃ x, f x > a := by
  dsimp only [FnHasUb, FnUb] at h
  push_neg at h
  exact h
```

在第二个例子中，我们可以使用 `dsimp` 展开 `FnHasUb` 和 `FnUb` 的定义。(我们需要使用 `dsimp` 而不是 `rw` 来展开 `FnUb`, 因为它出现在量词的辖域中。) 在上面的例子中，你可以验证对于 `¬∃ x, P x` 和 `¬∀ x, P x`，`push_neg` 策略做了我们期望的事情。即使不知道如何处理合取符号，你也应该能使用 `push_neg` 证明如下定理：

```
example (h : ¬Monotone f) : ∃ x y, x ≤ y ∧ f y < f x := by
  sorry
```

Mathlib 还有一个策略 `contrapose`，它把目标 `A → B` 转化为 `¬B → ¬A`。类似地，给定从假设 `h : A` 证明 `B` 的目标，`contrapose h` 会给你留下从假设 `¬B` 证明 `¬A` 的目标。使用 `contrapose!` 将在 `contrapose` 之外对

当前目标（事后假设）应用 push_neg 。

```
example (h : ¬FnHasUb f) : ∀ a, ∃ x, f x > a := by
  contrapose! h
  exact h

example (x : ℝ) (h : ∀ ε > 0, x ≤ ε) : x ≤ 0 := by
  contrapose! h
  use x / 2
  constructor <;> linarith
```

下一节解释 constructor 命令及其后分号的使用。

我们以 爆炸原理（ex falso）结束本节，它是说从自相矛盾可以得出任何命题。在 Lean 中，它用 False.elim 表示，它对于任何命题 P 确认了 False → P 。这似乎是一个奇怪的原理，但它经常出现。我们经常通过分情况来证明定理，有时我们可以证明其中一种情况是矛盾的。在这种情况下，我们需要断言矛盾确证了目标，这样我们就可以继续下一个目标了。（在 Section 3.5 中，我们将看到分情况讨论的实例。）

Lean 提供了多种在出现矛盾时关闭目标的方法。

```
example (h : 0 < 0) : a > 37 := by
  exfalso
  apply lt_irrefl 0 h

example (h : 0 < 0) : a > 37 :=
  absurd h (lt_irrefl 0)

example (h : 0 < 0) : a > 37 := by
  have h' : ¬0 < 0 := lt_irrefl 0
  contradiction
```

exfalso 策略把当前的目标替换为证明 False 的目标。给定 h : P 和 h' : ¬ P ，项 absurd h h' 可证明任何命题。最后，contradiction 策略尝试通过在假设中找到矛盾，例如一对形如 h : P 和 h' : ¬ P 的假设来关闭目标。另外本例也可以用 linarith 。

3.4 合取和双向蕴含

合取符号 ∧ （\and ）用于表示 “且”。constructor 策略允许你通过分别证明 A 和 B 来证明形如 A ∧ B 的定理。

```
example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬y ≤ x) : x ≤ y ∧ x ≠ y := by
  constructor
  · assumption
```

(continues on next page)

(continued from previous page)

```
  intro h
  apply h₁
  rw [h]
```

在这个例子中，`assumption` 策略要求 Lean 寻找一个能解决目标的假设。最后的 `rw` 应用了 ≤ 的自反性。展示一些使用角括号匿名构造子的等价方法。

```
example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬y ≤ x) : x ≤ y ∧ x ≠ y :=
  ⟨h₀, fun h ↦ h₁ (by rw [h])⟩

example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬y ≤ x) : x ≤ y ∧ x ≠ y :=
  have h : x ≠ y := by
    contrapose! h₁
    rw [h₁]
  ⟨h₀, h⟩
```

使用合取命题需要拆开两部分的证明。你可以用 `rcases` 策略做这个，也可以使用 `rintro` 或一个模式匹配函数 `fun`，使用方式和存在量词的情况类似。

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  rcases h with ⟨h₀, h₁⟩
  contrapose! h₁
  exact le_antisymm h₀ h₁

example {x y : ℝ} : x ≤ y ∧ x ≠ y → ¬y ≤ x := by
  rintro ⟨h₀, h₁⟩ h'
  exact h₁ (le_antisymm h₀ h')

example {x y : ℝ} : x ≤ y ∧ x ≠ y → ¬y ≤ x :=
  fun ⟨h₀, h₁⟩ h' ↦ h₁ (le_antisymm h₀ h')
```

像 `obtain` 策略一样，还有一个模式匹配版的 `have`：

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  have ⟨h₀, h₁⟩ := h
  contrapose! h₁
  exact le_antisymm h₀ h₁
```

和 `rcases` 相反，这里 `have` 策略把 `h` 留在了上下文中。另外，我们展示一些专家喜爱的模式匹配语法，尽管我们不会使用它们：

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  cases h
  case intro h₀ h₁ =>
    contrapose! h₁
    exact le_antisymm h₀ h₁

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  cases h
  next h₀ h₁ =>
    contrapose! h₁
    exact le_antisymm h₀ h₁

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  match h with
    | ⟨h₀, h₁⟩ =>
        contrapose! h₁
        exact le_antisymm h₀ h₁
```

与使用存在量词不同，你可以通过输入 `h.left` 和 `h.right` ，或者等价地，`h.1` 和 `h.2` ，提取假设 `h : A ∧ B` 两部分的证明。

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x := by
  intro h'
  apply h.right
  exact le_antisymm h.left h'

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬y ≤ x :=
  fun h' ↦ h.right (le_antisymm h.left h')
```

尝试使用这些技术，想出多种方式证明以下内容：

```
example {m n : ℕ} (h : m ∣ n ∧ m ≠ n) : m ∣ n ∧ ¬n ∣ m :=
  sorry
```

你可以通过匿名构造器，`rintro` 和 `rcases` 嵌套地使用 ∃ 和 ∧ 命题。

```
example : ∃ x : ℝ, 2 < x ∧ x < 4 :=
  ⟨5 / 2, by norm_num, by norm_num⟩

example (x y : ℝ) : (∃ z : ℝ, x < z ∧ z < y) → x < y := by
  rintro ⟨z, xltz, zlty⟩
  exact lt_trans xltz zlty
```

(continues on next page)

(continued from previous page)

```
example (x y : ℝ) : (∃ z : ℝ, x < z ∧ z < y) → x < y :=
  fun ⟨z, xltz, zlty⟩ ↦ lt_trans xltz zlty
```

也可以使用 use 策略：

```
example : ∃ x : ℝ, 2 < x ∧ x < 4 := by
  use 5 / 2
  constructor <;> norm_num

example : ∃ m n : ℕ, 4 < m ∧ m < n ∧ n < 10 ∧ Nat.Prime m ∧ Nat.Prime n := by
  use 5
  use 7
  norm_num

example {x y : ℝ} : x ≤ y ∧ x ≠ y → x ≤ y ∧ ¬y ≤ x := by
  rintro ⟨h₀, h₁⟩
  use h₀
  exact fun h' ↦ h₁ (le_antisymm h₀ h')
```

在第一个例子中，constructor 命令后的分号要求 Lean 对产生的全部两个目标使用 norm_num 策略。

在 Lean 中，A ↔ B 并不定义为 (A → B) ∧ (B → A)，但其实如果这样定义也无妨，而且它的行为几乎与此相同。你可以输入 h.mp 和 h.mpr，或者 h.1 和 h.2，用于表示 h : A ↔ B 的两个方向。你也可以使用 cases 及类似策略。要证明一条当且仅当语句，你可以使用 constructor 或角括号，就像你要证明一个合取一样。

```
example {x y : ℝ} (h : x ≤ y) : ¬y ≤ x ↔ x ≠ y := by
  constructor
  · contrapose!
    rintro rfl
    rfl
  contrapose!
  exact le_antisymm h

example {x y : ℝ} (h : x ≤ y) : ¬y ≤ x ↔ x ≠ y :=
  ⟨fun h₀ h₁ ↦ h₀ (by rw [h₁]), fun h₀ h₁ ↦ h₀ (le_antisymm h h₁)⟩
```

最后一个证明项令人困惑。在编写这样的表达式时，可以使用下划线来查看 Lean 的预期。

尝试你刚才学到的的各种技术和工具以证明下列命题：

```
example {x y : ℝ} : x ≤ y ∧ ¬y ≤ x ↔ x ≤ y ∧ x ≠ y :=
  sorry
```

一个更有意思的练习：请证明，对于任意两个实数 x 和 y，x^2 + y^2 = 0 当且仅当 x = 0 且 y = 0。建议使用 linarith, pow_two_nonneg 和 pow_eq_zero 证明一条辅助性的引理。

```
theorem aux {x y : ℝ} (h : x ^ 2 + y ^ 2 = 0) : x = 0 :=
  have h' : x ^ 2 = 0 := by sorry
  pow_eq_zero h'

example (x y : ℝ) : x ^ 2 + y ^ 2 = 0 ↔ x = 0 ∧ y = 0 :=
  sorry
```

在 Lean 中，双向蕴含具有双重含义。其一是视为合取，分别使用其两个部分。其二是命题之间的一个反射、对称且传递的关系，可以通过 calc 和 rw 使用它。我们经常把一个语句重写为等价语句。在下一个例子中，我们使用 abs_lt 将形如 |x| < y 的表达式替换为等价表达式 - y < x ∧ x < y，在下下个例子中，我们使用 Nat.dvd_gcd_iff 将形如 m ∣ Nat.gcd n k 的表达式替换为等价表达式 m ∣ n ∧ m ∣ k。

```
example (x : ℝ) : |x + 3| < 5 → -8 < x ∧ x < 2 := by
  rw [abs_lt]
  intro h
  constructor <;> linarith

example : 3 ∣ Nat.gcd 6 15 := by
  rw [Nat.dvd_gcd_iff]
  constructor <;> norm_num
```

看看你能否将 rw 与下面的定理结合起来使用来简短地证明“相反数不是一个非递减函数”。(注意 push_neg 不会为你展开定义，所以需要在定理证明中使用 rw [Monotone])。

```
theorem not_monotone_iff {f : ℝ → ℝ} : ¬Monotone f ↔ ∃ x y, x ≤ y ∧ f x > f y := by
  rw [Monotone]
  push_neg
  rfl

example : ¬Monotone fun x : ℝ ↦ -x := by
  sorry
```

关于合取和双向蕴含的进一步练习。偏序是一种具有传递性、反身性和反对称性的二元关系。更弱的概念预序只是一个反身、传递关系。对于任何预序 ≤，Lean 把相应的严格预序公理化定义为 a < b ↔ a ≤ b ∧ ¬ b ≤ a。证明如果 ≤ 是偏序，那么 a < b 等价于 a ≤ b ∧ a ≠ b：

```
variable {α : Type*} [PartialOrder α]
variable (a b : α)
```

(continues on next page)

(continued from previous page)

```
example : a < b ↔ a ≤ b ∧ a ≠ b := by
  rw [lt_iff_le_not_le]
  sorry
```

你只需要 `le_refl` 和 `le_trans` 和逻辑运算。证明即使在只假定 ≤ 是预序的情况下，我们也可以证明严格序是反自反的和传递的。在第二个例子中，为了方便，我们使用了化简器而非 `rw` 把 < 表示为关于 ≤ 和 ¬ 的表达式。我们稍后再讨论化简器，现在你只需要知道它会重复使用指定的引理，即使需要用不同的值将其实例化。

```
variable {α : Type*} [Preorder α]
variable (a b c : α)

example : ¬a < a := by
  rw [lt_iff_le_not_le]
  sorry

example : a < b → b < c → a < c := by
  simp only [lt_iff_le_not_le]
  sorry
```

3.5 析取

证明析取 `A ∨ B` （`\or`）的典型方式是证明 `A` 或证明 `B` 。`left` 策略选择 `A` ，`right` 策略选择 `B` 。

```
variable {x y : ℝ}

example (h : y > x ^ 2) : y > 0 ∨ y < -1 := by
  left
  linarith [pow_two_nonneg x]

example (h : -y > x ^ 2 + 1) : y > 0 ∨ y < -1 := by
  right
  linarith [pow_two_nonneg x]
```

我们不能使用匿名构造子来构造“或”的证明，因为 Lean 猜不出来我们在尝试证明哪个分支。作为替代方式，证明项中可以使用 `Or.inl` 和 `Or.inr` 来做出明确的选择。这里，`inl` 是“引入左项”的缩写，而 `inr` 是“引入右项”的缩写。

```
example (h : y > 0) : y > 0 ∨ y < -1 :=
  Or.inl h
```

(continues on next page)

(continued from previous page)

```
example (h : y < -1) : y > 0 ∨ y < -1 :=
  Or.inr h
```

通过证明一边或另一边来证明析取似乎很奇怪。实际上，哪种情况成立通常取决于假设中隐含或明确的情况区分。

我们通过 `rcases` 策略来使用形如 `A ∨ B` 的假设。与在合取或存在量词中使用 `rcases` 不同，这里的 `rcases` 策略产生 两个目标。它们都有相同的结论，但在第一种情况下，`A` 被假定为真，而在第二种情况下，`B` 被假定为真。换句话说，顾名思义，`rcases` 策略给出一个分情况证明。像往常一样，我们可以把假设中用到的名字报告给 Lean。下例中我们告诉 Lean 对每个分支都使用名字 `h` 。

```
example : x < |y| → x < y ∨ x < -y := by
  rcases le_or_gt 0 y with h | h
  · rw [abs_of_nonneg h]
    intro h; left; exact h
  · rw [abs_of_neg h]
    intro h; right; exact h
```

模式从合取情况下的 $\langle h_0, h_1 \rangle$ 变成了析取情况下的 $h_0 \mid h_1$ 。可以认为，第一种模式匹配包含 h_0 和 h_1 的数据，而第二种模式，也就是带竖线的那种，匹配包含 h_0 或 h_1 的数据。在这种情况下，因为两个目标分离，我们对两种情况可以使用同样的名字 `h` 。

绝对值函数的定义使得我们可以立即证明 `x ≥ 0` 蕴含着 `|x| = x` (这就是定理 `abs_of_nonneg`) 而 `x < 0` 则蕴含着 `|x| = -x`` (这是 ``abs_of_neg` )。表达式 `le_or_gt 0 x` 推出 `0 ≤ x ∨ x < 0` ，使我们可以将这两种情况分开。

析取也支持专家式模式匹配语法。此时最好用 `cases` 策略，因为它允许我们为每个 `cases` 命名，并在更接近使用的地方为引入的假设命名。

```
example : x < |y| → x < y ∨ x < -y := by
  cases le_or_gt 0 y
  case inl h =>
    rw [abs_of_nonneg h]
    intro h; left; exact h
  case inr h =>
    rw [abs_of_neg h]
    intro h; right; exact h
```

名字 `inl` 和 `inr` 分别是“intro left”和“intro right”的缩写。使用 `cases` 的好处是你可以以任意顺序证明每种情况；Lean 使用标签找到相关的目标。如果你不在乎这个，你可以使用 `next` 或者 `match` ，甚至是模式匹配版的 `have` 。

```
example : x < |y| → x < y ∨ x < -y := by
  cases le_or_gt 0 y
  next h =>
    rw [abs_of_nonneg h]
    intro h; left; exact h
  next h =>
    rw [abs_of_neg h]
    intro h; right; exact h

example : x < |y| → x < y ∨ x < -y := by
  match le_or_gt 0 y with
    | Or.inl h =>
      rw [abs_of_nonneg h]
      intro h; left; exact h
    | Or.inr h =>
      rw [abs_of_neg h]
      intro h; right; exact h
```

`match` 下需要用全称 `Or.inl` 和 `Or.inr` 来证明析取。本书通常会使用 `rcases` 来分割析取的情况。

试着用下面前两个定理证明三角不等式。Mathlib 中它也叫这个名字。

```
namespace MyAbs

theorem le_abs_self (x : ℝ) : x ≤ |x| := by
  sorry

theorem neg_le_abs_self (x : ℝ) : -x ≤ |x| := by
  sorry

theorem abs_add (x y : ℝ) : |x + y| ≤ |x| + |y| := by
  sorry
```

如果你喜欢这些“分情况讨论”，可以试试这些。try these.

```
theorem lt_abs : x < |y| ↔ x < y ∨ x < -y := by
  sorry

theorem abs_lt : |x| < y ↔ -y < x ∧ x < y := by
  sorry
```

你也可以将 `rcases` 和 `rintro` 与嵌套析取一起使用。当这些功能导致一个真正包含多个目标的情况划分时，每个新目标的模式都会用竖线隔开。

```
example {x : ℝ} (h : x ≠ 0) : x < 0 ∨ x > 0 := by
  rcases lt_trichotomy x 0 with xlt | xeq | xgt
  · left
    exact xlt
  · contradiction
  · right; exact xgt
```

你仍然可以进行模式嵌套，并使用 `rfl` 来替换等式：

```
example {m n k : ℕ} (h : m ∣ n ∨ m ∣ k) : m ∣ n * k := by
  rcases h with ⟨a, rfl⟩ | ⟨b, rfl⟩
  · rw [mul_assoc]
    apply dvd_mul_right
  · rw [mul_comm, mul_assoc]
    apply dvd_mul_right
```

尝试用一行证明下面的内容。使用 `rcases` 解包假设并分情况讨论，并使用分号和 `linarith` 解决每个分支。

```
example {z : ℝ} (h : ∃ x y, z = x ^ 2 + y ^ 2 ∨ z = x ^ 2 + y ^ 2 + 1) : z ≥ 0 := by
  sorry
```

在实数中，等式 `x * y = 0` 告诉我们 `x = 0` 或 `y = 0` 。在 Mathlib 中，这个事实被命名为 `eq_zero_or_eq_zero_of_mul_eq_zero` ，它是展示生产析取的另一个好例子。使用它证明下列命题（使用 `ring` 策略简化计算）：

```
example {x : ℝ} (h : x ^ 2 = 1) : x = 1 ∨ x = -1 := by
  sorry

example {x y : ℝ} (h : x ^ 2 = y ^ 2) : x = y ∨ x = -y := by
  sorry
```

在任意环 R 中，对于元素 x, 若存在非零元素 y 使得 $xy = 0$ ，则我们把 x 称为一个 *左零因子*，若存在非零元素 y 使得 $yx = 0$, 则我们把 x 称为一个 *右零因子*，是左或右零因子的元素称为 *零因子*。定理 `eq_zero_or_eq_zero_of_mul_eq_zero` 是说实数中没有非平凡的零因子。具有这一性质的交换环称为 *整环*。你对上面两个定理的证明在任意整环中应同样成立：

```
variable {R : Type*} [CommRing R] [IsDomain R]
variable (x y : R)

example (h : x ^ 2 = 1) : x = 1 ∨ x = -1 := by
  sorry
```

(continues on next page)

(continued from previous page)

```
example (h : x ^ 2 = y ^ 2) : x = y ∨ x = -y := by
  sorry
```

你证明第一个定理时可以不使用乘法交换律。在那种情况下，只需假定 `R` 是一个 `Ring` 而非 `CommRing` 。

有时在一个证明中我们想要根据一个语句是否为真来划分情况。对于任何命题 `P` ，我们可以使用 `em P : P ∨ ¬ P` 。名字 `em` 是“excluded middle (排中律)”的缩写。

```
example (P : Prop) : ¬¬P → P := by
  intro h
  cases em P
  · assumption
  · contradiction
```

或者，你也可以使用 `by_cases` 策略。

```
example (P : Prop) : ¬¬P → P := by
  intro h
  by_cases h' : P
  · assumption
  contradiction
```

注意到 `by_cases` 策略要求你为假设提供一个标签，该标签被用于各个分支，在这个例子中，一个分支是 `h' : P` 而另一个是 `h' : ¬ P` 。如果你留空，Lean 默认使用标签 `h` 。尝试证明如下等价性，使用 `by_cases` 解决其中一个方向。

```
example (P Q : Prop) : P → Q ↔ ¬P ∨ Q := by
  sorry
```

3.6 序列和收敛

来做一些真正的数学！在 Lean 中，我们可以把实数序列 $s_0, s_1, s_2, \ldots$ 写成函数 `s : ℕ → ℝ` 。我们称这个序列 *收敛*到数 a ，如果对任意 $\varepsilon > 0$ ，存在一个位置，在该位置之后，序列留在距离 a 不超过 ε 的范围内，也就是说，存在数 N, 使得对于任意 $n \geq N$ ，$|s_n - a| < \varepsilon$ 。

在 Lean 中，我们可以将其表述如下：

```
def ConvergesTo (s : ℕ → ℝ) (a : ℝ) :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, |s n - a| < ε
```

符号 `∀ ε > 0, ...` 是 `∀ ε, ε > 0 → ...` 的方便缩写，类似地，`∀ n ≥ N, ...` 是 `∀ n, n ≥ N → ...` 的缩写。记住 `ε > 0` 的定义是 `0 < ε` ，而 `n ≥ N` 的定义是 `N ≤ n` 。

本节将证明收敛的一些性质。但首先我们将介绍三个用于处理等式的策略。第一个是 ext 策略，它给我们提供了一种证明两个函数相等的途径。令 $f(x) = x + 1$ 和 $g(x) = 1 + x$ 是从实数到实数的函数。当然有 $f = g$，因为对任意 x, 它们返回相同的值。ext 策略允许我们通过证明函数在所有参数下的值都相同来证明函数等式。

```
example : (fun x y : ℝ ↦ (x + y) ^ 2) = fun x y : ℝ ↦ x ^ 2 + 2 * x * y + y ^ 2 := by
  ext
  ring
```

稍后将看到，可以指定 ext 中出现的变量名。例如，你可以尝试在上面的证明中把 ext 替换为 ext u v 。第二个策略是 congr 策略，它允许我们通过调和有差异的部分来证明两个表达式之间的等式：

```
example (a b : ℝ) : |a| = |a - b + b| := by
  congr
  ring
```

在这里 congr 策略剥离两边的 abs ，只留下 a = a - b + b 。

最后，convert 策略用于在定理和结论不完全一致时将定理应用于目标。例如，假定我们想从 1 < a 证明 a < a * a 。库定理 mul_lt_mul_right 让我们证明 1 * a < a * a 。一种可能的方法是逆向工作，重写目标使其具有这种形式。相反，convert 策略原封不动地应用定理，留下证明匹配目标所需等式的任务。

```
example {a : ℝ} (h : 1 < a) : a < a * a := by
  convert (mul_lt_mul_right _).2 h
  · rw [one_mul]
  exact lt_trans zero_lt_one h
```

这个示例还说明了另一个有用的技巧：Lean 无法自动填写带下划线的表达式时，它就会把下划线作为另一个目标留给我们。

下面证明任意常数序列 $a, a, a, \ldots$ 收敛。

```
theorem convergesTo_const (a : ℝ) : ConvergesTo (fun x : ℕ ↦ a) a := by
  intro ε εpos
  use 0
  intro n nge
  rw [sub_self, abs_zero]
  apply εpos
```

策略 simp 经常可以为你省下手写 rw [sub_self, abs_zero] 这种步骤的麻烦。稍后详细介绍。

让我们证明一个更有趣的定理，如果 s 收敛到 a ，t 收敛到 b ，那么 fun n ↦ s n + t n 收敛到 a + b 。在开始写形式证明之前，先建立清晰的纸笔证明是很有帮助的。给定 ε 大于 0 ，思路是利用假设得到 Ns ，使得超出这一位置时，s 在 a 附近 ε / 2 内，以及 Nt ，使得超出这一位置时，s 在 b 附近 ε / 2 内。而后，当 n 大于或等于 Ns 和 Nt 的最大值时，序列 fun n ↦ s n + t n 应在 a + b 附近 ε 内。完成下面的证明。

（提示，你可以使用 `le_of_max_le_left` 和 `le_of_max_le_right`；`norm_num` 可以证明 `ε / 2 + ε / 2 = ε`。还有，使用 `congr` 策略有助于证明 `|s n + t n - (a + b)|` 等于 `|(s n - a) + (t n - b)|`，因为在那之后你就可以使用三角不等式。把全部变量 `s, t, a, b` 标记为隐变量，因为它们可以从假设中推断出来。）

```
theorem convergesTo_add {s t : ℕ → ℝ} {a b : ℝ}
      (cs : ConvergesTo s a) (ct : ConvergesTo t b) :
    ConvergesTo (fun n ↦ s n + t n) (a + b) := by
  intro ε εpos
  dsimp -- this line is not needed but cleans up the goal a bit.
  have ε2pos : 0 < ε / 2 := by linarith
  rcases cs (ε / 2) ε2pos with ⟨Ns, hs⟩
  rcases ct (ε / 2) ε2pos with ⟨Nt, ht⟩
  use max Ns Nt
  sorry
```

证明把加法换成乘法的同样定理需要技巧。我们会先证明一些辅助性的结论。看看你能否同样完成下列证明，它表明若 `s` 收敛于 `a`，则 `fun n ↦ c * s n` 收敛于 `c * a`。根据 `c` 是否等于零来区分不同情况有助于证明。我们已经处理了零的情况，现在让你在另一种假设，即 `c` 非零时证明结论。

```
theorem convergesTo_mul_const {s : ℕ → ℝ} {a : ℝ} (c : ℝ) (cs : ConvergesTo s a) :
    ConvergesTo (fun n ↦ c * s n) (c * a) := by
  by_cases h : c = 0
  · convert convergesTo_const 0
    · rw [h]
      ring
    rw [h]
    ring
  have acpos : 0 < |c| := abs_pos.mpr h
  sorry
```

下一个定理也很有趣：收敛序列在绝对值意义下最终是有界的。完成它。

```
theorem exists_abs_le_of_convergesTo {s : ℕ → ℝ} {a : ℝ} (cs : ConvergesTo s a) :
    ∃ N b, ∀ n, N ≤ n → |s n| < b := by
  rcases cs 1 zero_lt_one with ⟨N, h⟩
  use N, |a| + 1
  sorry
```

事实上，该定理还可以加强，断言存在一个对所有 `n` 值都成立的界 `b`。但这个版本对我们的目的来说已经足够强了，我们将在本节结尾看到它在更一般的条件下成立。

接下来的引理是辅助性的：我们证明若 `s` 收敛到 `a` 且 `t` 收敛到 `0`，则 `fun n ↦ s n * t n` 收敛到 `0`。为了证明它，我们使用上一个定理来找到 `B`，作为 `s` 在超出 N_0 时的界。看看你能否理解我们简述的思路并完成

证明。

```
theorem aux {s t : ℕ → ℝ} {a : ℝ} (cs : ConvergesTo s a) (ct : ConvergesTo t 0) :
    ConvergesTo (fun n ↦ s n * t n) 0 := by
  intro ε εpos
  dsimp
  rcases exists_abs_le_of_convergesTo cs with ⟨N₀, B, h₀⟩
  have Bpos : 0 < B := lt_of_le_of_lt (abs_nonneg _) (h₀ N₀ (le_refl _))
  have pos₀ : ε / B > 0 := div_pos εpos Bpos
  rcases ct _ pos₀ with ⟨N₁, h₁⟩
  sorry
```

如果你已经走到这一步，那么恭喜你！我们现在已经离定理不远了。下面的证明将为我们画上圆满的句号。

```
theorem convergesTo_mul {s t : ℕ → ℝ} {a b : ℝ}
      (cs : ConvergesTo s a) (ct : ConvergesTo t b) :
    ConvergesTo (fun n ↦ s n * t n) (a * b) := by
  have h₁ : ConvergesTo (fun n ↦ s n * (t n + -b)) 0 := by
    apply aux cs
    convert convergesTo_add ct (convergesTo_const (-b))
    ring
  have := convergesTo_add h₁ (convergesTo_mul_const b cs)
  convert convergesTo_add h₁ (convergesTo_mul_const b cs) using 1
  · ext; ring
  ring
```

另一个具有挑战性的练习，请试着填写下面的极限唯一性的证明概要。(如果你觉得自信，可以删除证明概要，然后尝试从头开始证明)。

```
theorem convergesTo_unique {s : ℕ → ℝ} {a b : ℝ}
      (sa : ConvergesTo s a) (sb : ConvergesTo s b) :
    a = b := by
  by_contra abne
  have : |a - b| > 0 := by sorry
  let ε := |a - b| / 2
  have εpos : ε > 0 := by
    change |a - b| / 2 > 0
    linarith
  rcases sa ε εpos with ⟨Na, hNa⟩
  rcases sb ε εpos with ⟨Nb, hNb⟩
  let N := max Na Nb
  have absa : |s N - a| < ε := by sorry
```

(continues on next page)

(continued from previous page)

```
  have absb : |s N - b| < ε := by sorry
  have : |a - b| < |a - b| := by sorry
  exact lt_irrefl _ this
```

在本节的最后，我们要指出，我们的证明是可以推广的。例如，我们使用的唯一一条自然数性质是它们的结构带有含 `min` 和 `max` 的偏序。如果把 `ℕ` 换成任意线性序 `α`，你可以验证一切仍然正确：

```
variable {α : Type*} [LinearOrder α]

def ConvergesTo' (s : α → ℝ) (a : ℝ) :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, |s n - a| < ε
```

在 Section 10.1 中，我们将看到 Mathlib 有更一般的机制来处理收敛，它不仅抽象化了定义域和值域的特定特征，还在抽象意义下统一了不同类型的收敛。

CHAPTER

FOUR

集合和函数

集合、关系和函数为所有数学分支的构造提供了统一的语言。由于函数和关系可以用集合来定义，因此公理集合论可以作为数学的基础。

作为 Lean 基础的原始概念则是 **类型**，包括在类型间定义函数的方法。Lean 中的每个表达式都有一个类型：自然数、实数、从实数到实数的函数、群、向量空间等等。也有的表达式是类型，也就是说，它们的类型是 `Type` 。Lean 和 Mathlib 提供定义新类型的方式，以及定义这些类型的对象的方式。

从概念上讲，你可以把类型看作是一组对象的集合。要求每个对象都有一个类型有一些好处。例如，它使得重载像 + 这样的符号成为可能，有时它还能缩减冗长的输入，因为 Lean 可以从对象的类型中推断出大量信息。当你将函数应用于错误的参数个数，或将函数应用于错误类型的参数时，类型系统让 Lean 可以标记错误。

Lean 的库确实定义了初等集合论概念。与集合论不同的是，在 Lean 中，集合总是某种类型的对象和集合，例如自然数集合或从实数到实数的函数的集合。类型和集合之间的区别需要一些时间来适应，本章将带你了解其中的要点。

4.1 集合

设 `α` 是任意类型，则类型 `Set α` 是 `α` 中的元素组成的集合。这一类型支持常规的集合论运算和关系。例如，`s ⊆ t` 表示 `s` 是 `t` 的子集（`⊆` 符号是 `\ss` 或 `\sub`），`s ∩ t` 指交集（`\i` 或 `\cap`），`s ∪ t` 指并集（`\un` 或 `\cup`）。库中也定义了集合 `univ`，它包含类型 `α` 的全部元素，以及空集 `∅`（`\empty`）。给定 `x : α` 和 `s : Set α`，`x ∈ s` 表示 `x` 属于 `s`（`\in` 或 `\mem`，涉及集合成员关系的定理的名字经常含有 `mem`）。`x ∉ s`（`\notin`）是 `¬ x ∈ s` 的缩写。

证明关于集合的命题的一种方法是使用 `rw` 或 `simp` 来展开定义。在下面的第二个例子中，我们使用 `simp only` 告诉化简器只使用我们提供的列表中的等式，而不是整个数据库中的等式。不同于 `rw`，`simp` 可以在全称或存在量词内实施化简。

```
variable {α : Type*}
variable (s t u : Set α)
open Set
```

(continues on next page)

(continued from previous page)

```
example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  rw [subset_def, inter_def, inter_def]
  rw [subset_def] at h
  simp only [mem_setOf]
  rintro x ⟨xs, xu⟩
  exact ⟨h _ xs, xu⟩

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  simp only [subset_def, mem_inter_iff] at *
  rintro x ⟨xs, xu⟩
  exact ⟨h _ xs, xu⟩
```

本例中，我们打开了 `Set` 命名空间以用更短的定理名访问定理。事实上，我们完全可以不用 `rw` 和 `simp`：

```
example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  intro x xsu
  exact ⟨h xsu.1, xsu.2⟩
```

这叫做 **定义规约（definitional reduction）**：为了理解 `intro` 命令和匿名构造子，Lean 不得不展开定义。下面的例子也说明了这一现象：

```
example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
  fun x ⟨xs, xu⟩ ↦ ⟨h xs, xu⟩
```

为了处理并集，我们可以使用 `Set.union_def` 和 `Set.mem_union` 。由于 `x ∈ s ∪ t` 展开为 `x ∈ s ∨ x ∈ t`，我们也可以使用 `cases` 策略强制要求定义规约。

```
example : s ∩ (t ∪ u) ⊆ s ∩ t ∪ s ∩ u := by
  intro x hx
  have xs : x ∈ s := hx.1
  have xtu : x ∈ t ∪ u := hx.2
  rcases xtu with xt | xu
  · left
    show x ∈ s ∩ t
    exact ⟨xs, xt⟩
  · right
    show x ∈ s ∩ u
    exact ⟨xs, xu⟩
```

交集比并集优先级更高，表达式 `(s ∩ t) ∪ (s ∩ u)` 中不必需使用括号，但使用括号会更易读。下面是更简短的证明。使用 `rintro` 时，有时我们需要使用括号包围析取模式 `h1 | h2` 使得 Lean 能正确解析它。

```
example : s ∩ (t ∪ u) ⊆ s ∩ t ∪ s ∩ u := by
  rintro x ⟨xs, xt | xu⟩
  · left; exact ⟨xs, xt⟩
  · right; exact ⟨xs, xu⟩
```

作为练习，试着证明反向的包含关系：

```
example : s ∩ t ∪ s ∩ u ⊆ s ∩ (t ∪ u) := by
  sorry
```

库里也定义了差集 `s \ t`（反斜线以 \\ 输入）。表达式 `x ∈ s \ t` 展开为 `x ∈ s ∧ x ∉ t`。可以使用 `Set.diff_eq` 和 `dsimp` 或 `Set.mem_diff` 手动重写它，而下面的两个证明展示了这一点和更简洁的方案。

```
example : (s \ t) \ u ⊆ s \ (t ∪ u) := by
  intro x xstu
  have xs : x ∈ s := xstu.1.1
  have xnt : x ∉ t := xstu.1.2
  have xnu : x ∉ u := xstu.2
  constructor
  · exact xs
  intro xtu
  -- x ∈ t ∨ x ∈ u
  rcases xtu with xt | xu
  · show False; exact xnt xt
  · show False; exact xnu xu

example : (s \ t) \ u ⊆ s \ (t ∪ u) := by
  rintro x ⟨⟨xs, xnt⟩, xnu⟩
  use xs
  rintro (xt | xu) <;> contradiction
```

作为练习，证明反向的包含关系：

```
example : s \ (t ∪ u) ⊆ (s \ t) \ u := by
  sorry
```

要证明两个集合相等，只需证明任一集合中的每个元素都是另一个集合中的元素。这个原理称为“外延性”，`ext` 策略可以用于处理它。

```
example : s ∩ t = t ∩ s := by
  ext x
  simp only [mem_inter_iff]
```

(continues on next page)

(continued from previous page)

```
  constructor
  · rintro ⟨xs, xt⟩; exact ⟨xt, xs⟩
  · rintro ⟨xt, xs⟩; exact ⟨xs, xt⟩
```

删除 `simp only [mem_inter_iff]` 一行并不会影响证明。下面的单行证明更简洁但难读：

```
example : s ∩ t = t ∩ s :=
  Set.ext fun x ↦ ⟨fun ⟨xs, xt⟩ ↦ ⟨xt, xs⟩, fun ⟨xt, xs⟩ ↦ ⟨xs, xt⟩⟩
```

甚至还能更短：

```
example : s ∩ t = t ∩ s := by ext x; simp [and_comm]
```

除了使用 `ext` 之外，我们还可以使用 `Subset.antisymm` 这个定理，它可以通过证明 `s ⊆ t` 和 `t ⊆ s` 来证明集合间的等式 `s = t` 。

```
example : s ∩ t = t ∩ s := by
  apply Subset.antisymm
  · rintro x ⟨xs, xt⟩; exact ⟨xt, xs⟩
  · rintro x ⟨xt, xs⟩; exact ⟨xs, xt⟩
```

练习：

```
example : s ∩ t = t ∩ s :=
    Subset.antisymm sorry sorry
```

技巧：你可以用下划线代替 `sorry` ，并将鼠标悬停在它上面，Lean 会向你显示它在这一位置的预期。

下面是一些你可能会喜欢的集合论性质：

```
example : s ∩ (s ∪ t) = s := by
  sorry

example : s ∪ s ∩ t = s := by
  sorry

example : s \ t ∪ t = s ∪ t := by
  sorry

example : s \ t ∪ t \ s = (s ∪ t) \ (s ∩ t) := by
  sorry
```

说到集合的表示，下面就为大家揭秘背后的机制。在类型论中，类型 `α` 上的 **性质**或 **谓词**只是一个函数 `P : α`

→ Prop 。这是有意义的：给定 a : α, P a 就是 P 对 a 成立这一命题。在库中，Set α 定义为 α → Prop 而 x ∈ s 定义为 s x 。换句话说，集合实际上就是性质，只是被当成对象。

库中也定义了集合构造器的符号。表达式 { y | P y } 展开就是 (fun y ↦ P y) 。因此 x ∈ { y | P y } 规约为 P x 。因此我们可以把偶数性质转化为偶数集合：

```
def evens : Set ℕ :=
  { n | Even n }

def odds : Set ℕ :=
  { n | ¬Even n }

example : evens ∪ odds = univ := by
  rw [evens, odds]
  ext n
  simp [-Nat.not_even_iff_odd]
  apply Classical.em
```

你最好能一句一句看看这个证明。尝试删除 rw [evens, odds] 这一行，证明仍然成立。

事实上，集合构造器符号定义

- s ∩ t 为 {x | x ∈ s ∧ x ∈ t}
- s ∪ t 为 {x | x ∈ s ∨ x ∈ t}
- ∅ 为 {x | False}
- univ 为 {x | True}

我们经常需要精确指定 ∅ 和 univ 的类型，因为 Lean 很难猜出我们指的是哪种类型。下面的例子演示了当需要的时候 Lean 如何展开最后两个定义。在第二个例子中，trivial 是库中对 True 的标准证明。

```
example (x : ℕ) (h : x ∈ (∅ : Set ℕ)) : False :=
  h

example (x : ℕ) : x ∈ (univ : Set ℕ) :=
  trivial
```

作为练习，证明下列包含关系。使用 intro n 以展开子集定义，使用 simp 把集合论构造约化为逻辑。我们也推荐使用定理 Nat.Prime.eq_two_or_odd 和 Nat.even_iff 。

```
example : { n | Nat.Prime n } ∩ { n | n > 2 } ⊆ { n | ¬Even n } := by
  sorry
```

有点令人困惑的是，库中有多个版本的谓词 Prime 。最通用的谓词对任何有零元素的交换幺半群都有意义。谓词 Nat.Prime 是针对自然数的。幸运的是，有一个定理指出，在特定情况下这两个概念是一致的，所以你

总是可以把一个谓词改写成另一个谓词。

```
#print Prime

#print Nat.Prime

example (n : ℕ) : Prime n ↔ Nat.Prime n :=
  Nat.prime_iff.symm

example (n : ℕ) (h : Prime n) : Nat.Prime n := by
  rw [Nat.prime_iff]
  exact h
```

`rwa` 策略是在 `rw` 后跟着一个 `assumption` 策略。

```
example (n : ℕ) (h : Prime n) : Nat.Prime n := by
  rwa [Nat.prime_iff]
```

Lean 引入了记号 `∀ x ∈ s, ...`，等价于 `∀ x, x ∈ s → ...`，类似地也引入了 `∃ x ∈ s, ...,`。有时它们被称为 **约束量词（bounded quantifier）**，因为这种构造将 `x` 的意义约束在集合 `s` 内。因此，库中使用这些量词的定理通常在名称中包含 `ball` 或 `bex`。定理 `bex_def` 断言 `∃ x ∈ s, ...` 等价于 `∃ x, x ∈ s ∧ ...,`，但当它们和 `rintro`, `use` 以及匿名构造子一起使用时，这两种量词表达式的行为大致相同。因此，我们通常不需要使用 `bex_def` 来精确转换它们。用例：

```
variable (s t : Set ℕ)

example (h₀ : ∀ x ∈ s, ¬Even x) (h₁ : ∀ x ∈ s, Prime x) : ∀ x ∈ s, ¬Even x ∧ Prime x␣
↪:= by
  intro x xs
  constructor
  · apply h₀ x xs
  apply h₁ x xs

example (h : ∃ x ∈ s, ¬Even x ∧ Prime x) : ∃ x ∈ s, Prime x := by
  rcases h with ⟨x, xs, _, prime_x⟩
  use x, xs
```

尝试下这些略有不同的命题：

```
section
variable (ssubt : s ⊆ t)
```

(continues on next page)

(continued from previous page)

```
example (h₀ : ∀ x ∈ t, ¬Even x) (h₁ : ∀ x ∈ t, Prime x) : ∀ x ∈ s, ¬Even x ∧ Prime x␣
↪:= by
  sorry

example (h : ∃ x ∈ s, ¬Even x ∧ Prime x) : ∃ x ∈ t, Prime x := by
  sorry

end
```

带下标的并集和交集是另一种重要的集合论构造。我们可以把 `α` 中的元素组成的集合的序列 $A_0, A_1, A_2, \ldots$ 建模为函数 `A : ℕ → Set α`，此时 `⋃ i, A i` 表示它们的并集，而 `⋂ i, A i` 表示它们的交集。这里自然数没有什么特别之处，因此 `ℕ` 可以替换为任意作为指标集的类型 `I`。下面说明它们的用法。

```
variable {α I : Type*}
variable (A B : I → Set α)
variable (s : Set α)

open Set

example : (s ∩ ⋃ i, A i) = ⋃ i, A i ∩ s := by
  ext x
  simp only [mem_inter_iff, mem_iUnion]
  constructor
  · rintro ⟨xs, ⟨i, xAi⟩⟩
    exact ⟨i, xAi, xs⟩
  rintro ⟨i, xAi, xs⟩
  exact ⟨xs, ⟨i, xAi⟩⟩

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ ⋂ i, B i := by
  ext x
  simp only [mem_inter_iff, mem_iInter]
  constructor
  · intro h
    constructor
    · intro i
      exact (h i).1
    intro i
    exact (h i).2
  rintro ⟨h1, h2⟩ i
  constructor
```

(continues on next page)

(continued from previous page)

```
  · exact h1 i
  exact h2 i
```

带下标的并集或交集通常需要使用括号，因为与量词一样，绑定变量的范围会尽可能地扩展。

尝试证明下列恒等式。其中一个方向需要经典逻辑！我们建议在证明的适当位置使用 `by_cases xs : x ∈ s` 。

```
example : (s ∪ ⋂ i, A i) = ⋂ i, A i ∪ s := by
  sorry
```

Mathlib 也支持在自定义的下标集中做并集和交集，类似于约束量词。你可以使用 `mem_iUn₂` 和 `mem_iInter₂` 解包它们的含义。正如下面的示例所示，Lean 的化简器也可以执行这些替换。

```
def primes : Set ℕ :=
  { x | Nat.Prime x }

example : (⋃ p ∈ primes, { x | p ^ 2 ∣ x }) = { x | ∃ p ∈ primes, p ^ 2 ∣ x } :=by
  ext
  rw [mem_iUn₂]
  simp

example : (⋃ p ∈ primes, { x | p ^ 2 ∣ x }) = { x | ∃ p ∈ primes, p ^ 2 ∣ x } := by
  ext
  simp

example : (⋂ p ∈ primes, { x | ¬p ∣ x }) ⊆ { x | x = 1 } := by
  intro x
  contrapose!
  simp
  apply Nat.exists_prime_and_dvd
```

试着解决下面这个类似的例子。如果你开始输入 `eq_univ` ，标签补全器会告诉你 `apply eq_univ_of_forall` 是开始证明的好方法。另外推荐使用 `Nat.exists_infinite_primes` 定理。

```
example : (⋃ p ∈ primes, { x | x ≤ p }) = univ := by
  sorry
```

给定一组集合 `s : Set (Set α)` ，它们的并集 $\bigcup_0$ `s` 具有类型 `Set α` ，定义为 `{x | ∃ t ∈ s, x ∈ t}` 。类似地，它们的交集 $\bigcap_0$ `s` 定义为 `{x | ∀ t ∈ s, x ∈ t}` 。这些运算分别称为 `sUnion` 和 `sInter` 。下面的例子展示了它们与下标集合上并集和交集的关系，在库中它们称为 `sUnion_eq_biUnion` 和 `sInter_eq_biInter` 。

```
variable {α : Type*} (s : Set (Set α))

example : ⋃₀ s = ⋃ t ∈ s, t := by
  ext x
  rw [mem_iUnion₂]
  simp

example : ⋂₀ s = ⋂ t ∈ s, t := by
  ext x
  rw [mem_iInter₂]
  rfl
```

4.2 函数

若 `f : α → β` 是一个函数，而 `p` 是一个类型为 `β` 的元素的集合，则 Mathlib 库把 `preimage f p`（记为 `f ⁻¹' p`）定义为 `{x | f x ∈ p}`。表达式 `x ∈ f ⁻¹' p` 简写为 `f x ∈ p`。这经常很方便，就像在下面例子中看到的这样：

```
variable {α β : Type*}
variable (f : α → β)
variable (s t : Set α)
variable (u v : Set β)

open Function
open Set

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v := by
  ext
  rfl
```

若 `s` 是类型为 `α` 的元素的集合，库也把 `image f s`（记为 `f '' s`）定义为 `{y | ∃ x, x ∈ s ∧ f x = y}`。从而假设 `y ∈ f '' s` 分解为三元组 `⟨x, xs, xeq⟩`，其中 `x : α` 满足假设 `xs : x ∈ s` 和 `xeq : f x = y`。`rintro` 策略中的 `rfl` 标签（见 Section 3.2）在这类情况下被明确了。

```
example : f '' (s ∪ t) = f '' s ∪ f '' t := by
  ext y; constructor
  · rintro ⟨x, xs | xt, rfl⟩
    · left
      use x, xs
    right
```

(continues on next page)

(continued from previous page)

```
    use x, xt
  rintro (⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩)
  · use x, Or.inl xs
  use x, Or.inr xt
```

`use` 策略使用了 `rfl` 使得在可以做到时关闭目标。

另一个例子：

```
example : s ⊆ f ⁻¹' (f '' s) := by
  intro x xs
  show f x ∈ f '' s
  use x, xs
```

可以把 `use x, xs` 这一行换成 `apply mem_image_of_mem f xs`，因为这个定理就是为此时设计，不过此处的证明可以提醒你它是用存在量词定义的。

下列等价关系是一个好练习题：

```
example : f '' s ⊆ v ↔ s ⊆ f ⁻¹' v := by
  sorry
```

它表明 `image f` 和 `preimage f` 是 `Set α` 和 `Set β` 之间所谓 *伽罗瓦连接 (Galois connection)* 的一个实例，每个实例都由子集关系作为偏序。在库中，这个等价关系被命名为 `image_subset_iff`。在实践中，右边是更常用的表达方式，因为 `y ∈ f ⁻¹' t` 展开为 `f y ∈ t`，而处理 `x ∈ f '' s` 则需要分解一个存在量词。

这里有一长串集合论定理供您消遣。你不必一次全部做完，只需做其中几个，其余的留待一个空闲的雨天再做。

```
example (h : Injective f) : f ⁻¹' (f '' s) ⊆ s := by
  sorry

example : f '' (f ⁻¹' u) ⊆ u := by
  sorry

example (h : Surjective f) : u ⊆ f '' (f ⁻¹' u) := by
  sorry

example (h : s ⊆ t) : f '' s ⊆ f '' t := by
  sorry

example (h : u ⊆ v) : f ⁻¹' u ⊆ f ⁻¹' v := by
  sorry
```

(continues on next page)

(continued from previous page)

```
example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v := by
  sorry

example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t := by
  sorry

example (h : Injective f) : f '' s ∩ f '' t ⊆ f '' (s ∩ t) := by
  sorry

example : f '' s \ f '' t ⊆ f '' (s \ t) := by
  sorry

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) := by
  sorry

example : f '' s ∩ v = f '' (s ∩ f ⁻¹' v) := by
  sorry

example : f '' (s ∩ f ⁻¹' u) ⊆ f '' s ∩ u := by
  sorry

example : s ∩ f ⁻¹' u ⊆ f ⁻¹' (f '' s ∩ u) := by
  sorry

example : s ∪ f ⁻¹' u ⊆ f ⁻¹' (f '' s ∪ u) := by
  sorry
```

你还可以尝试下一组练习，这些练习描述了像和原像在带下标集合的并集和交集中的性质。在第三个练习中，参数 `i : I` 是必要的，因为要保证指标集非空。要证明其中任何一个，我们推荐使用 `ext` 或 `intro` 展开等式或集合之间的包含关系的定义，然后调用 `simp` 解包集合成员所需的条件。

```
variable {I : Type*} (A : I → Set α) (B : I → Set β)

example : (f '' ⋃ i, A i) = ⋃ i, f '' A i := by
  sorry

example : (f '' ⋂ i, A i) ⊆ ⋂ i, f '' A i := by
  sorry
```

(continues on next page)

(continued from previous page)

```
example (i : I) (injf : Injective f) : (⋂ i, f '' A i) ⊆ f '' ⋂ i, A i := by
  sorry

example : (f ⁻¹' ⋃ i, B i) = ⋃ i, f ⁻¹' B i := by
  sorry

example : (f ⁻¹' ⋂ i, B i) = ⋂ i, f ⁻¹' B i := by
  sorry
```

Mathlib 库定义了谓词 `InjOn f s`, 表示 `f` 在 `s` 上是单射。定义如下：

```
example : InjOn f s ↔ ∀ x₁ ∈ s, ∀ x₂ ∈ s, f x₁ = f x₂ → x₁ = x₂ :=
  Iff.refl _
```

可以证明语句 `Injective f` 等价于 `InjOn f univ` 。类似地，库中将 `range f` 定义为 `{x | ∃y, f y = x}` ，因此可以证明 `range f` 等于 `f '' univ` 。这是 Mathlib 中的一个常见主题：虽然函数的许多属性都是对于其完整域定义的，但通常也有相对化版本，将语句限制为域类型的子集。

下面是一些使用 `InjOn` 和 `range` 的例子：

```
open Set Real

example : InjOn log { x | x > 0 } := by
  intro x xpos y ypos
  intro e
  -- log x = log y
  calc
    x = exp (log x) := by rw [exp_log xpos]
    _ = exp (log y) := by rw [e]
    _ = y := by rw [exp_log ypos]


example : range exp = { y | y > 0 } := by
  ext y; constructor
  · rintro ⟨x, rfl⟩
    apply exp_pos
  intro ypos
  use log y
  rw [exp_log ypos]
```

证明这些：

```
example : InjOn sqrt { x | x ≥ 0 } := by
  sorry

example : InjOn (fun x ↦ x ^ 2) { x : ℝ | x ≥ 0 } := by
  sorry

example : sqrt '' { x | x ≥ 0 } = { y | y ≥ 0 } := by
  sorry

example : (range fun x ↦ x ^ 2) = { y : ℝ | y ≥ 0 } := by
  sorry
```

我们要使用两种新方法定义函数 `f : α → β` 的反函数。第一，我们需要小心 Lean 中任意类型可能为空。为了在没有 `x` 满足 `f x = y` 时定义 `f` 的逆在 `y` 处的取值，我们想要指定 `α` 中的缺省值。将注释 `[Inhabited α]` 添加为变量等于假设 `α` 有一个默认元素，即 `default`。第二，当存在多于一个 `x` 满足 `f x = y` 时，反函数需要 选择其中一个。这需要诉诸 选择公理。Lean 允许使用多种途径访问它：一种方便的途径是使用经典的 `choose` 操作符，如下所示。

```
variable {α β : Type*} [Inhabited α]

#check (default : α)

variable (P : α → Prop) (h : ∃ x, P x)

#check Classical.choose h

example : P (Classical.choose h) :=
  Classical.choose_spec h
```

给定 `h : ∃ x, P x`, 则 `Classical.choose h` 的值是某个满足 `P x` 的 `x`。定理 `Classical.choose_spec h` 表明 `Classical.choose h` 符合要求。

有了这些，我们就可以定义反函数如下：

```
noncomputable section

open Classical

def inverse (f : α → β) : β → α := fun y : β ↦
  if h : ∃ x, f x = y then Classical.choose h else default

theorem inverse_spec {f : α → β} (y : β) (h : ∃ x, f x = y) : f (inverse f y) = y :=␣
```

(continues on next page)

(continued from previous page)

```
↪by
  rw [inverse, dif_pos h]
  exact Classical.choose_spec h
```

之所以需要 `noncomputable section` 和 `open Classical` 这两行，是因为我们正在以一种重要的方式使用经典逻辑。在输入 `y` 时，函数 `inverse f` 返回某个满足 `f x = y` 的 `x` 值（如果有的话），否则返回一个缺省的 `α` 元素。这是一个 *dependent if* 结构的实例，因为在正例中，返回值 `Classical.choose h` 取决于假设 `h`。给定 `h : e` 时，等式 `dif_pos h` 将 `if h : e then a else b` 改写为 `a`，同样，给定 `h : ¬ e`，`dif_neg h` 将它改写为 `b`。定理 `inverse_spec` 表明 `inverse f` 满足这一设定的第一部分。

如果你还不完全理解它们的工作原理，也不用担心。仅 `inverse_spec` 定理就足以说明，当且仅当 `f` 是单射时，`inverse f` 是左逆，当且仅当 `f` 是满射时，`inverse f` 是右逆。通过在 VS Code 双击或右键单击 `LeftInverse` 和 `RightInverse`，或使用命令 `#print LeftInverse` 和 `#print RightInverse`，可以查看它们的定义。然后尝试证明这两个定理。它们很棘手，你最好先打个自然语言草稿。每个定理都能用大约六行短代码证明。附加题：尝试将每个证明压缩成单行证明。

```
variable (f : α → β)

open Function

example : Injective f ↔ LeftInverse (inverse f) f :=
  sorry

example : Surjective f ↔ RightInverse (inverse f) f :=
  sorry
```

在本节的最后，我们将从类型论的角度来阐述康托尔的著名定理，即不存在从集合到其幂集的满射函数。看看你是否能理解这个证明，然后填上缺失的两行。

```
theorem Cantor : ∀ f : α → Set α, ¬Surjective f := by
  intro f surjf
  let S := { i | i ∉ f i }
  rcases surjf S with ⟨j, h⟩
  have h₁ : j ∉ f j := by
    intro h'
    have : j ∉ f j := by rwa [h] at h'
    contradiction
  have h₂ : j ∈ S
  sorry
  have h₃ : j ∉ S
  sorry
```

(continues on next page)

(continued from previous page)

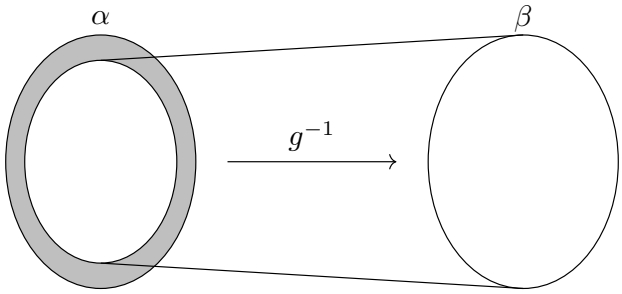
```
contradiction
```

4.3 施罗德-伯恩斯坦定理

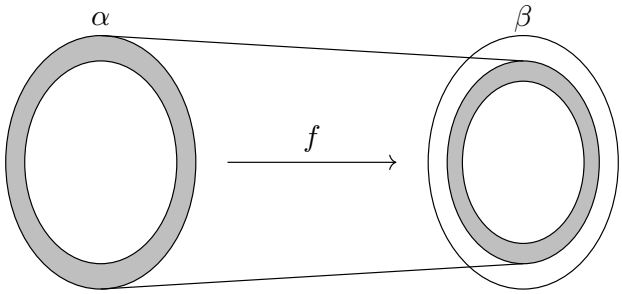
我们用一个初等而非平凡的定理结束本章。令 α 和 β 是集合。（在本节的形式化中，它们实际上会是类型。）假定 $f : \alpha \to \beta$ 和 $g : \beta \to \alpha$ 都是单射。直观上，这意味着 α 不大于 β，反之亦然。如果 α 和 β 是有限的，这意味着它们具有相同的基数，而这相当于说它们之间存在双射。在十九世纪，康托尔（Cantor）指出，即使当 α 和 β 无穷时，同样的结果也成立。这个结果最终由戴德金（Dedekind）、施罗德（Schröder）和伯恩斯坦（Bernstein）各自独立证明。

本节引入的新方法会接下来的章节中更详细地解释，所以不必担心本节讲得太快。本节目标是向你展示你已经可以为真正的数学结果的形式化做出贡献。

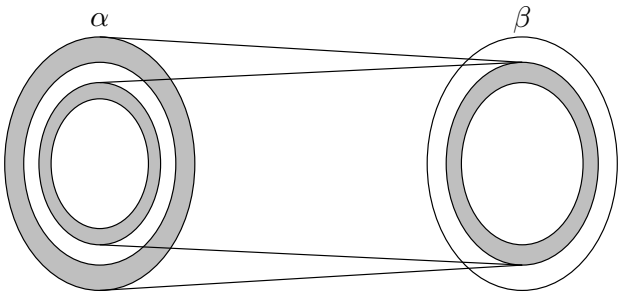
证明的主要目标是构造 α 和 β 之间的双射。现在展示直观上的证明步骤。考虑映射 g 在 α 中的像。在这个像中，可以定义 g 的逆，且是到 β 的双射。



问题是双射不包括图中的阴影区域，如果 g 不是满射，则阴影是非空的。或者，我们可以使用 f 把整个 α 映射到 β，但在那种情况下问题是若 f 不是满射，则它会同样错过 β 内的一些元素。



但现在考虑从 α 到自身的复合映射 $g \circ f$。由于这个复合映射是单射，可以证明它也是 α 到它的像的双射，直观上就是在 α 内部产生了一个缩小的副本。



这个复合将内部阴影环映射到另一个这样的集合，我们可以将其视为更小的同心阴影环，依此类推。这会产生一个阴影环的同心序列，每个都与下一个有双射对应。这实际上得到了 α 和 g 的像之间的双射：α 的白色部分不动，每个阴影圆环映射到下一个。和 g^{-1} 复合一下就产生了所需的 α 和 β 的双射。

形式化地，令 A 为这列阴影区域的并，如下定义 $h : \alpha \to \beta$:

$$h(x) = \begin{cases} f(x) & \text{if } x \in A \\ g^{-1}(x) & \text{otherwise.} \end{cases}$$

换句话说，我们在阴影部分使用 f ，而在其余部分使用 g 的逆。生成的映射 h 是单射，因为每个分支都是单射并且两个分支的像是不相交的。为了看出它是满射，假设给定了 β 中的元素 y ，考虑 $g(y)$ 。若 $g(y)$ 位于阴影区域之一，它不能在第一个环中，因此我们有 $g(y) = g(f(x))$ ，其中 x 在上一个环中。由 g 的单射性，我们有 $h(x) = f(x) = y$ 。若 $g(y)$ 不在阴影区域中，那么由 h 的定义，我们有 $h(g(y)) = y$ 。不论哪种情况，y 都在 h 的像中。

这个论证听起来应该有道理，但细节却很微妙。形式化证明不仅可以提高我们对结果的信心，还可以帮助我们更好地理解它。因为证明使用经典逻辑，所以我们告诉 Lean，我们的定义通常是不可计算的。

```
noncomputable section
open Classical
variable {α β : Type*} [Nonempty β]
```

记号 `[Nonempty β]` 规定 `β` 非空。我们使用它是因为我们将用于构造 g^{-1} 的 Mathlib 原语要求它。定理在 β 为空的情形是平凡的，虽然把形式化推广到覆盖这种情况并不难，但当下就算了。特别地，我们需要假设 `[Nonempty β]` 以使用 Mathlib 中定义的运算 `invFun` 。对于给定的 `x : α`, 如果有的话，`invFun g x` 选择 `x` 在 `β` 中的一个原像，否则就返回 `β` 中的任意元素。若 `g` 是单射，则 `invFun g` 一定是左逆，若 `g` 是满射，则它是右逆。

```
#check (invFun g : α → β)
#check (leftInverse_invFun : Injective g → LeftInverse (invFun g) g)
#check (leftInverse_invFun : Injective g → ∀ y, invFun g (g y) = y)
#check (invFun_eq : (∃ y, g y = x) → g (invFun g x) = x)
```

我们定义对应于阴影区域的并的集合如下。

```
variable (f : α → β) (g : β → α)

def sbAux : ℕ → Set α
  | 0 => univ \ g '' univ
  | n + 1 => g '' (f '' sbAux n)

def sbSet :=
  ⋃ n, sbAux f g n
```

定义 `sbAux` 是 **递归定义**的一个例子，我们将在下一章解释。它定义了一列集合

$$S_0 = \alpha \backslash g(\beta)$$
$$S_{n+1} = g(f(S_n)).$$

定义 `sbSet` 对应于我们的证明概要中的集合 $A = \bigcup_{n\in\mathbb{N}} S_n$ 。上面描述的函数 h 现在定义如下：

```
def sbFun (x : α) : β :=
  if x ∈ sbSet f g then f x else invFun g x
```

我们将需要这样的事实：我们定义的 g^{-1} 是 A 的补集上的右逆，也就是 α 中白色区域上的右逆。这是因为最外层的环 S_0 等于 $\alpha \setminus g(\beta)$ ，因此 A 的补集包含于 $g(\beta)$ 中。所以，对 A 的补集中的每个 x ，存在 y 使得 $g(y) = x$ 。（由 g 的单射性，这个 y 是唯一的，但下一个定理只说了 `invFun g x` 返回 `y` 使得 `g y = x`.）

逐步完成下面的证明，确保你理解发生了什么，并填写剩余部分。你需要在最后使用 `invFun_eq` 。请注意，此处用 `sbAux` 重写替换了 `sbAux f g 0` 与相应定义方程的右侧。

```
theorem sb_right_inv {x : α} (hx : x ∉ sbSet f g) : g (invFun g x) = x := by
  have : x ∈ g '' univ := by
    contrapose! hx
    rw [sbSet, mem_iUnion]
    use 0
    rw [sbAux, mem_diff]
    sorry
  have : ∃ y, g y = x := by
    sorry
  sorry
```

我们现在转向证明 h 是单射。非正式地证明过程如下。首先，假设 $h(x_1) = h(x_2)$ 。若 x_1 在 A 中，则 $h(x_1) = f(x_1)$ ，且我们可以证明 x_2 在 A 中如下。若非如此，则我们有 $h(x_2) = g^{-1}(x_2)$ 。由 $f(x_1) = h(x_1) = h(x_2)$ 我们有 $g(f(x_1)) = x_2$ 。由 A 的定义，既然 x_1 在 A 中，x_2 同样在 A 中，矛盾。因此，若 x_1 在 A 中，则 x_2 也在，这种情况下我们有 $f(x_1) = h(x_1) = h(x_2) = f(x_2)$ 。f 的单射性推出 $x_1 = x_2$ 。对称论证表明若 x_2 在 A 中，则 x_1 也在，这又一次蕴含着 $x_1 = x_2$ 。

剩下的唯一可能性是 x_1 和 x_2 都不在 A 中。这种情况下，我们有 $g^{-1}(x_1) = h(x_1) = h(x_2) = g^{-1}(x_2)$ 。两边使用 g 就得到 $x_1 = x_2$ 。

我们再次鼓励你逐步完成以下证明，观察这个论证在 Lean 中是如何展开的。看看你是否可以使用 `sb_right_inv` 结束证明。

```
theorem sb_injective (hf : Injective f) : Injective (sbFun f g) := by
  set A := sbSet f g with A_def
  set h := sbFun f g with h_def
  intro x₁ x₂
  intro (hxeq : h x₁ = h x₂)
  show x₁ = x₂
  simp only [h_def, sbFun, ← A_def] at hxeq
  by_cases xA : x₁ ∈ A ∨ x₂ ∈ A
  · wlog x₁A : x₁ ∈ A generalizing x₁ x₂ hxeq xA
    · symm
      apply this hxeq.symm xA.symm (xA.resolve_left x₁A)
    have x₂A : x₂ ∈ A := by
      apply _root_.not_imp_self.mp
      intro (x₂nA : x₂ ∉ A)
      rw [if_pos x₁A, if_neg x₂nA] at hxeq
      rw [A_def, sbSet, mem_iUnion] at x₁A
      have x₂eq : x₂ = g (f x₁) := by
        sorry
      rcases x₁A with ⟨n, hn⟩
      rw [A_def, sbSet, mem_iUnion]
      use n + 1
      simp [sbAux]
      exact ⟨x₁, hn, x₂eq.symm⟩
    sorry
  push_neg  at xA
  sorry
```

这个证明引入了一些新策略。首先，`set` 策略引入了缩写 `A` 和 `h` 分别代表 `sbSet f g` 和 `sb_fun f g` 。我们把对应的定义式命名为 `A_def` 和 `h_def` 。这些缩写是定义性的，也就是说，当需要时，Lean 有时会自动展开它们。但并不总是如此；例如，当使用 `rw` 时，我们通常需要精确地使用 `A_def` 和 `h_def` 。所以这些定义带来了一个权衡：它们可以使表达式更短并且更具可读性，但它们有时需要我们做更多的工作。

一个更有趣的策略是 `wlog` 策略，它封装了上面非正式证明中的对称性论证。我们现在不会详细讨论它，但请注意它恰好做了我们想要的工作。如果将鼠标悬停在该策略上，你可以查看其文档。

满射性的论证甚至更容易。给定 β 中的 y ，我们考虑两种情况，取决于 $g(y)$ 是否在 A 中。若是，则它不会在最外层的环 S_0 中，因为根据定义，这个环和 g 的像不相交。因此存在 n 使得 $g(y)$ 是 S_{n+1} 的元素。这意味着它形如 $g(f(x))$ ，其中 x 在 S_n 中。由 g 的单射性，我们有 $f(x) = y$ 。在 $g(y)$ 属于 A 的补集的情况，我们立刻得到 $h(g(y)) = y$, 证毕。

我们再次鼓励您逐步完成证明并填写缺失的部分。策略 `cases n with n` 分解为情况 `g y ∈ sbAux f g 0`

和 `g y ∈ sbAux f g n.succ` 。在两种情况下，通过 `simp [sbAux]` 调用化简器都会应用 `sbAux` 的相应定义式。

```
theorem sb_surjective (hg : Injective g) : Surjective (sbFun f g) := by
  set A := sbSet f g with A_def
  set h := sbFun f g with h_def
  intro y
  by_cases gyA : g y ∈ A
  · rw [A_def, sbSet, mem_iUnion] at gyA
    rcases gyA with ⟨n, hn⟩
    rcases n with _ | n
    · simp [sbAux] at hn
    simp [sbAux] at hn
    rcases hn with ⟨x, xmem, hx⟩
    use x
    have : x ∈ A := by
      rw [A_def, sbSet, mem_iUnion]
      exact ⟨n, xmem⟩
    simp only [h_def, sbFun, if_pos this]
    exact hg hx
  sorry
```

我们现在可以把它们放在一起。最后的论证简短而优雅，证明使用了 `Bijective h` 展开为 `Injective h ∧ Surjective h` 这一事实。

```
theorem schroeder_bernstein {f : α → β} {g : β → α} (hf : Injective f) (hg :␣
↪Injective g) :
    ∃ h : α → β, Bijective h :=
  ⟨sbFun f g, sb_injective f g hf, sb_surjective f g hg⟩
```

CHAPTER

FIVE

初等数论

在这一章中，我们将向您展示如何将数论中的一些基本结果形式化。随着我们处理更复杂的数学内容，证明会变得更长、更复杂，但会建立在您已经掌握的技能之上。

5.1 无理数根

让我们从古希腊人已知的一个事实开始，即根号 2 是无理数。如果我们假设并非如此，那么我们可以将根号 2 写成最简分数形式 $\sqrt{2} = a/b$ 。两边平方得到 $a^2 = 2b^2$ ，这表明 a 是偶数。如果设 a = 2c ，则有 $4c^2 = 2b^2$ ，从而得出 $b^2 = 2c^2$ 。这表明 b 也是偶数，与我们假设 a / b 已化为最简形式相矛盾。

说 a/b 是最简分数意味着 a 和 b 没有公因数，也就是说，它们是 **互质**的。Mathlib 定义谓词 `Nat.Coprime m n` 为 `Nat.gcd m n = 1` 。使用 Lean 的匿名投影符号，如果 `s` 和 `t` 是类型为 `Nat` 的表达式，我们可以写 `s.Coprime t` 而不是 `Nat.Coprime s t` ，对于 `Nat.gcd` 也是如此。和往常一样，Lean 通常会在必要时自动展开 `Nat.Coprime` 的定义，但我们也可以通过重写或简化使用标识符 `Nat.Coprime` 来手动进行。

`norm_num` 策略足够聪明，可以计算出具体的值。

```
#print Nat.Coprime

example (m n : Nat) (h : m.Coprime n) : m.gcd n = 1 :=
  h

example (m n : Nat) (h : m.Coprime n) : m.gcd n = 1 := by
  rw [Nat.Coprime] at h
  exact h

example : Nat.Coprime 12 7 := by norm_num

example : Nat.gcd 12 8 = 4 := by norm_num
```

我们在 Section 2.4 中已经遇到过 *gcd* 函数。对于整数也有一个 *gcd* 版本；我们将在下面讨论不同数系之间的关系。甚至还有适用于一般代数结构类别的通用 *gcd* 函数以及通用的 `Prime` 和 `Coprime` 概念。在下一章中，我们将了解 Lean 是如何处理这种通用性的。与此同时，在本节中，我们将把注意力限制在自然数上。

我们还需要素数的概念，即 `Nat.Prime` 。定理 `Nat.prime_def_lt` 提供了一个常见的特征描述，而 `Nat.Prime.eq_one_or_self_of_dvd` 则提供了另一种。

```
#check Nat.prime_def_lt

example (p : ℕ) (prime_p : Nat.Prime p) : 2 ≤ p ∧ ∀ m : ℕ, m < p → m ∣ p → m = 1 := by
  rwa [Nat.prime_def_lt] at prime_p

#check Nat.Prime.eq_one_or_self_of_dvd

example (p : ℕ) (prime_p : Nat.Prime p) : ∀ m : ℕ, m ∣ p → m = 1 ∨ m = p :=
  prime_p.eq_one_or_self_of_dvd

example : Nat.Prime 17 := by norm_num

-- commonly used
example : Nat.Prime 2 :=
  Nat.prime_two

example : Nat.Prime 3 :=
  Nat.prime_three
```

在自然数中，素数具有不能写成非平凡因数乘积的性质。在更广泛的数学背景下，具有这种性质的环中的元素被称为 **不可约元**。如果一个环中的元素在它整除某个乘积时，就整除其中一个因数，那么这个元素被称为 **素元**。自然数的一个重要性质是，在这种情况下这两个概念是重合的，从而产生了定理 `Nat.Prime.dvd_mul` 。我们可以利用这一事实来确立上述论证中的一个关键性质：如果一个数的平方是偶数，那么这个数本身也是偶数。Mathlib 在 `Algebra.Group.Even` 中定义了谓词 `Even` ，但出于下文将要阐明的原因，我们将简单地使用 `2 ∣ m` 来表示 `m` 是偶数。

```
#check Nat.Prime.dvd_mul
#check Nat.Prime.dvd_mul Nat.prime_two
#check Nat.prime_two.dvd_mul

theorem even_of_even_sqr {m : ℕ} (h : 2 ∣ m ^ 2) : 2 ∣ m := by
  rw [pow_two, Nat.prime_two.dvd_mul] at h
  cases h <;> assumption

example {m : ℕ} (h : 2 ∣ m ^ 2) : 2 ∣ m :=
  Nat.Prime.dvd_of_dvd_pow Nat.prime_two h
```

在接下来的学习过程中，您需要熟练掌握查找所需事实的方法。请记住，如果您能猜出名称的前缀并且已导入相关库，您可以使用制表符补全（有时需要按 `Ctrl + Tab` ）来找到您要查找的内容。您可以在任何标识

符上按 Ctrl + 点击跳转到其定义所在的文件，这使您能够浏览附近的定义和定理。您还可以使用 Lean 社区网页 上的搜索引擎，如果其他方法都行不通，不要犹豫，在 Zulip 上提问。

```
example (a b c : Nat) (h : a * b = a * c) (h' : a ≠ 0) : b = c :=
  -- apply? suggests the following:
  (mul_right_inj' h').mp h
```

我们证明根号二为无理数的核心在于以下定理。试着用 even_of_even_sqr 和定理 Nat.dvd_gcd 来完善证明概要。

```
example {m n : ℕ} (coprime_mn : m.Coprime n) : m ^ 2 ≠ 2 * n ^ 2 := by
  intro sqr_eq
  have : 2 ∣ m := by
    sorry
  obtain ⟨k, meq⟩ := dvd_iff_exists_eq_mul_left.mp this
  have : 2 * (2 * k ^ 2) = 2 * n ^ 2 := by
    rw [← sqr_eq, meq]
    ring
  have : 2 * k ^ 2 = n ^ 2 :=
    sorry
  have : 2 ∣ n := by
    sorry
  have : 2 ∣ m.gcd n := by
    sorry
  have : 2 ∣ 1 := by
    sorry
  norm_num at this
```

实际上，只需稍作改动，我们就可以用任意素数替换 2 。在下一个示例中试一试。在证明的最后，您需要从 p ∣ 1 推导出矛盾。您可以使用 Nat.Prime.two_le ，它表明任何素数都大于或等于 2，以及 Nat.le_of_dvd 。

```
example {m n p : ℕ} (coprime_mn : m.Coprime n) (prime_p : p.Prime) : m ^ 2 ≠ p * n ^␣
↪2 := by
  sorry
```

让我们考虑另一种方法。这里有一个快速证明：如果 p 是质数，那么 $m^2 \neq pn^2$ ：假设 $m^2 = pn^2$ 并考虑 m 和 n 分解成质数的情况，那么方程左边 p 出现的次数为偶数，而右边为奇数，这与假设相矛盾。请注意，此论证要求 n 以及因此 m 不为零。下面的形式化证明确认了这一假设是足够的。唯一分解定理指出，除了零以外的任何自然数都可以唯一地表示为素数的乘积。Mathlib 包含此定理的形式化版本，用函数 Nat.primeFactorsList 来表示，该函数返回一个数的素因数列表，且这些素因数按非递减顺序排列。该库证明了 Nat.primeFactorsList n 中的所有元素都是素数，任何大于零的 n 都等于其素因数的乘积，并且如果 n 等于另一组素数的乘积，那么这组素数就是 Nat.primeFactorsList n 的一个排列。

```
#check Nat.primeFactorsList
#check Nat.prime_of_mem_primeFactorsList
#check Nat.prod_primeFactorsList
#check Nat.primeFactorsList_unique
```

您可以浏览这些定理以及附近的其他定理，尽管我们尚未讨论列表成员、乘积或排列。对于当前的任务，我们不需要这些内容。相反，我们将使用这样一个事实：Mathlib 有一个函数 `Nat.factorization`，它表示与函数相同的数据。具体来说，`Nat.factorization n p`，我们也可以写成 `n.factorization p`，返回 `p` 在 `n` 的质因数分解中的重数。我们将使用以下三个事实。

```
theorem factorization_mul' {m n : ℕ} (mnez : m ≠ 0) (nnez : n ≠ 0) (p : ℕ) :
    (m * n).factorization p = m.factorization p + n.factorization p := by
  rw [Nat.factorization_mul mnez nnez]
  rfl

theorem factorization_pow' (n k p : ℕ) :
    (n ^ k).factorization p = k * n.factorization p := by
  rw [Nat.factorization_pow]
  rfl

theorem Nat.Prime.factorization' {p : ℕ} (prime_p : p.Prime) :
    p.factorization p = 1 := by
  rw [prime_p.factorization]
  simp
```

实际上，在 Lean 中，`n.factorization` 被定义为一个有限支撑的函数，这解释了在您逐步查看上述证明时会看到的奇怪符号。现在不必担心这个问题。就我们此处的目的而言，可以将上述三个定理当作黑箱来使用。下一个示例表明，化简器足够智能，能够将 `n^2 ≠ 0` 替换为 `n ≠ 0`。策略 `simpa` 仅调用 `simp` 后再调用 `assumption`。看看你能否利用上面的恒等式来补全证明中缺失的部分。

```
example {m n p : ℕ} (nnz : n ≠ 0) (prime_p : p.Prime) : m ^ 2 ≠ p * n ^ 2 := by
  intro sqr_eq
  have nsqr_nez : n ^ 2 ≠ 0 := by simpa
  have eq1 : Nat.factorization (m ^ 2) p = 2 * m.factorization p := by
    sorry
  have eq2 : (p * n ^ 2).factorization p = 2 * n.factorization p + 1 := by
    sorry
  have : 2 * m.factorization p % 2 = (2 * n.factorization p + 1) % 2 := by
    rw [← eq1, sqr_eq, eq2]
  rw [add_comm, Nat.add_mul_mod_self_left, Nat.mul_mod_right] at this
  norm_num at this
```

这个证明的一个妙处在于它还能推广。这里的 2 没有什么特殊之处；稍作改动，该证明就能表明，无论何时我们写出 m^k = r * n^k ，那么在 r 中任何素数 p 的幂次都必须是 k 的倍数。要使用 Nat.count_factors_mul_of_pos 来处理 r * n^k ，我们需要知道 r 是正数。但当 r 为零时，下面的定理是显然成立的，并且很容易通过简化器证明。所以证明是分情况来进行的。rcases r with _ | r 这一行将目标替换为两个版本：一个版本中 r 被替换为 0 ，另一个版本中 r 被替换为 r + 1 。在第二种情况下，我们可以使用定理 r.succ_ne_zero ，它表明 r + 1 ≠ 0 （succ 表示后继）。还要注意，以 have : npow_nz 开头的那行提供了 n^k ≠ 0 的简短证明项证明。要理解其工作原理，可以尝试用策略证明替换它，然后思考这些策略是如何描述证明项的。试着补全下面证明中缺失的部分。在最后，你可以使用 Nat.dvd_sub' 和 Nat.dvd_mul_right 来完成证明。请注意，此示例并未假定 p 为素数，但当 p 不是素数时结论是显而易见的，因为根据定义此时 r.factorization p 为零，而且无论如何该证明在所有情况下都成立。

```
example {m n k r : ℕ} (nnz : n ≠ 0) (pow_eq : m ^ k = r * n ^ k) {p : ℕ} :
    k ∣ r.factorization p := by
  rcases r with _ | r
  · simp
  have npow_nz : n ^ k ≠ 0 := fun npowz ↦ nnz (pow_eq_zero npowz)
  have eq1 : (m ^ k).factorization p = k * m.factorization p := by
    sorry
  have eq2 : ((r + 1) * n ^ k).factorization p =
      k * n.factorization p + (r + 1).factorization p := by
    sorry
  have : r.succ.factorization p = k * m.factorization p - k * n.factorization p := by
    rw [← eq1, pow_eq, eq2, add_comm, Nat.add_sub_cancel]
  rw [this]
  sorry
```

我们或许想要通过多种方式改进这些结果。首先，关于根号二为无理数的证明应当提及根号二，这可以理解为实数或复数中的一个元素。并且，声称其为无理数应当说明有理数的情况，即不存在任何有理数与之相等。此外，我们应当将本节中的定理推广到整数。尽管从数学角度显而易见，如果能将根号二写成两个整数的商，那么也能写成两个自然数的商，但正式证明这一点仍需付出一定努力。在 Mathlib 中，自然数、整数、有理数、实数和复数分别由不同的数据类型表示。将注意力限制在不同的域上通常是有帮助的：我们会看到对自然数进行归纳很容易，而且在不考虑实数的情况下，关于整数的可除性进行推理是最简单的。但在不同域之间进行转换是一件令人头疼的事，这是我们必须应对的问题。我们将在本章后面再次讨论这个问题。我们还应当能够将最后一个定理的结论加强，即表明数字 r 是 k 的幂，因为其 k 次方根恰好是每个整除 r 的素数的 r 中该素数的幂次除以 k 后的乘积。要做到这一点，我们需要更好的方法来处理有限集合上的乘积和求和问题，这也是我们之后会再次探讨的一个主题。事实上，本节中的所有结果在 Mathlib 的 Data.Real.Irrational 中都有更广泛的证明。对于任意交换幺半群，都定义了 重数（multiplicity）这一概念，其取值范围为扩展自然数 enat ，即在自然数的基础上增加了无穷大这一值。在下一章中，我们将开始探讨 Lean 如何支持这种泛化的方法。

5.2 归纳与递归

自然数集 $\mathbb{N} = \{0, 1, 2, \ldots\}$ 不仅本身具有根本的重要性，而且在新数学对象的构建中也起着核心作用。Lean 的基础允许我们声明 **归纳类型**，这些类型由给定的 **构造子**列表归纳生成。在 Lean 中，自然数是这样声明的。

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

您可以在库中通过输入 `#check Nat` 然后 `Ctrl + 点击`标识符 `Nat` 来找到它。该命令指定了 `Nat` 是由两个构造函数 `zero : Nat` 和 `succ : Nat → Nat` 自然且归纳地生成的数据类型。当然，库中为 `nat` 和 `zero` 分别引入了记号 `ℕ` 和 `0`。(数字会被转换为二进制表示，但现在我们不必担心这些细节。)

对于数学工作者而言，“自然”意味着类型 `Nat` 有一个元素 `zero` 以及一个单射的后继函数 `succ`，其值域不包含 `zero`。

```
example (n : Nat) : n.succ ≠ Nat.zero :=
  Nat.succ_ne_zero n

example (m n : Nat) (h : m.succ = n.succ) : m = n :=
  Nat.succ.inj h
```

对于数学工作者而言，“归纳”这个词意味着自然数附带有一个归纳证明原则和一个递归定义原则。本节将向您展示如何使用这些原则。

以下是一个阶乘函数的递归定义示例。

```
def fac : ℕ → ℕ
  | 0 => 1
  | n + 1 => (n + 1) * fac n
```

这种语法需要一些时间来适应。请注意第一行没有 `:=`。接下来的两行提供了递归定义的基础情况和归纳步骤。这些等式是定义性成立的，但也可以通过将名称 `fac` 给予 `simp` 或 `rw` 来手动使用。

```
example : fac 0 = 1 :=
  rfl

example : fac 0 = 1 := by
  rw [fac]

example : fac 0 = 1 := by
  simp [fac]

example (n : ℕ) : fac (n + 1) = (n + 1) * fac n :=
```

(continues on next page)

(continued from previous page)

```
  rfl

example (n : ℕ) : fac (n + 1) = (n + 1) * fac n := by
  rw [fac]

example (n : ℕ) : fac (n + 1) = (n + 1) * fac n := by
  simp [fac]
```

阶乘函数实际上已经在 Mathlib 中定义为 `Nat.factorial` 。您可以通过输入 `#check Nat.factorial` 并使用 `Ctrl + 点击`跳转到它。为了便于说明，我们在示例中将继续使用 `fac` 。定义 `Nat.factorial` 前面的注释 `@[simp]` 指定定义方程应添加到简化的默认等式数据库中。归纳法原理指出，我们可以通过证明某个关于自然数的一般性陈述对 0 成立，并且每当它对某个自然数 n 成立时，它对 $n+1$ 也成立，从而证明该一般性陈述。因此，在下面的证明中，行 `induction' n with n ih` 会产生两个目标：在第一个目标中，我们需要证明 `0 < fac 0` ；在第二个目标中，我们有额外的假设 `ih : 0 < fac n` ，并且需要证明 `0 < fac (n + 1)` 。短语 `with n ih` 用于为归纳假设命名变量和假设，您可以为它们选择任何名称。

```
theorem fac_pos (n : ℕ) : 0 < fac n := by
  induction' n with n ih
  · rw [fac]
    exact zero_lt_one
  rw [fac]
  exact mul_pos n.succ_pos ih
```

该 `induction` 策略足够智能，能够将依赖于归纳变量的假设作为归纳假设的一部分包含进来。接下来，我们可以逐步执行一个示例，以具体说明这一过程。

```
theorem dvd_fac {i n : ℕ} (ipos : 0 < i) (ile : i ≤ n) : i ∣ fac n := by
  induction' n with n ih
  · exact absurd ipos (not_lt_of_ge ile)
  rw [fac]
  rcases Nat.of_le_succ ile with h | h
  · apply dvd_mul_of_dvd_right (ih h)
  rw [h]
  apply dvd_mul_right
```

以下示例为阶乘函数提供了一个粗略的下界。结果发现，从分情况证明入手会更容易些，这样证明的其余部分就从 $n = 1$ 的情况开始。尝试使用 `pow_succ` 或 `pow_succ'` 通过归纳法完成论证。

```
theorem pow_two_le_fac (n : ℕ) : 2 ^ (n - 1) ≤ fac n := by
  rcases n with _ | n
  · simp [fac]
```

(continues on next page)

(continued from previous page)

```
  sorry
```

归纳法常用于证明涉及有限和与乘积的恒等式。Mathlib 定义了表达式 `Finset.sum s f`，其中 `s : Finset α` 是类型为 `α` 的元素的有限集合，而 `f` 是定义在 `α` 上的函数。

`f` 的值域可以是任何支持交换、结合加法运算且具有零元素的类型。

如果您导入 `Algebra.BigOperators.Basic` 并执行命令 `open BigOperators`，则可以使用更直观的符号 `∑ x in s, f x`。当然，对于有限乘积也有类似的运算和符号。

我们将在下一节以及稍后的章节中讨论 `Finset` 类型及其支持的操作。目前，我们仅使用 `Finset.range n`，它表示小于 `n` 的自然数的有限集合。

```
variable {α : Type*} (s : Finset ℕ) (f : ℕ → ℕ) (n : ℕ)

#check Finset.sum s f
#check Finset.prod s f

open BigOperators
open Finset

example : s.sum f = ∑ x in s, f x :=
  rfl

example : s.prod f = ∏ x in s, f x :=
  rfl

example : (range n).sum f = ∑ x in range n, f x :=
  rfl

example : (range n).prod f = ∏ x in range n, f x :=
  rfl
```

事实 `Finset.sum_range_zero` 和 `Finset.sum_range_succ` 为求和至 n 提供了递归描述，乘积的情况也是如此。

```
example (f : ℕ → ℕ) : ∑ x in range 0, f x = 0 :=
  Finset.sum_range_zero f

example (f : ℕ → ℕ) (n : ℕ) : ∑ x in range n.succ, f x = ∑ x in range n, f x + f n :=
  Finset.sum_range_succ f n

example (f : ℕ → ℕ) : ∏ x in range 0, f x = 1 :=
```

(continues on next page)

(continued from previous page)

```
  Finset.prod_range_zero f

example (f : ℕ → ℕ) (n : ℕ) : ∏ x in range n.succ, f x = (∏ x in range n, f x) * f n␣
↪:=
  Finset.prod_range_succ f n
```

每对中的第一个恒等式是定义性的，也就是说，您可以将证明替换为 `rfl` 。

以下表达的是我们定义为乘积形式的阶乘函数。

```
example (n : ℕ) : fac n = ∏ i in range n, (i + 1) := by
  induction' n with n ih
  · simp [fac, prod_range_zero]
  simp [fac, ih, prod_range_succ, mul_comm]
```

我们将 `mul_comm` 作为简化规则包含在内这一事实值得评论。使用恒等式 `x * y = y * x` 进行简化似乎很危险，因为这通常会导致无限循环。不过，Lean 的简化器足够聪明，能够识别这一点，并且仅在结果项在某些固定但任意的项排序中具有较小值的情况下应用该规则。下面的示例表明，使用 `mul_assoc` 、`mul_comm` 和 `mul_left_comm` 这三条规则进行简化，能够识别出括号位置和变量顺序不同但实质相同的乘积。

```
example (a b c d e f : ℕ) : a * (b * c * f * (d * e)) = d * (a * f * e) * (c * b) :=␣
↪by
  simp [mul_assoc, mul_comm, mul_left_comm]
```

大致来说，这些规则的作用是将括号向右推移，然后重新排列两边的表达式，直到它们都遵循相同的规范顺序。利用这些规则以及相应的加法规则进行简化，是个很实用的技巧。

回到求和恒等式，我们建议按照以下证明步骤来证明自然数之和（从 1 加到 n）等于 $n(n+1)/2$。证明的第一步是消去分母。这在形式化恒等式时通常很有用，因为涉及除法的计算通常会有附加条件。（同样，在可能的情况下避免在自然数上使用减法也是有用的。）

```
theorem sum_id (n : ℕ) : ∑ i in range (n + 1), i = n * (n + 1) / 2 := by
  symm; apply Nat.div_eq_of_eq_mul_right (by norm_num : 0 < 2)
  induction' n with n ih
  · simp
  rw [Finset.sum_range_succ, mul_add 2, ← ih]
  ring
```

我们鼓励您证明类似的平方和恒等式，以及您在网上能找到的其他恒等式。

```
theorem sum_sqr (n : ℕ) : ∑ i in range (n + 1), i ^ 2 = n * (n + 1) * (2 * n + 1) / 6␣
↪:= by
  sorry
```

在 Lean 的核心库中，加法和乘法本身是通过递归定义来定义的，并且它们的基本性质是通过归纳法来确立的。如果您喜欢思考诸如基础性的话题，您可能会喜欢证明乘法和加法的交换律和结合律以及乘法对加法的分配律。您可以在自然数的副本上按照下面的提纲进行操作。请注意，我们可以对 `MyNat` 使用 `induction` 策略；Lean 足够聪明，知道要使用相关的归纳原理（当然，这与 `Nat` 的归纳原理相同）。

我们先从加法的交换律讲起。一个不错的经验法则是，由于加法和乘法都是通过在第二个参数上递归定义的，所以通常在证明中对处于该位置的变量进行归纳证明是有利的。在证明结合律时，决定使用哪个变量有点棘手。

在没有通常的零、一、加法和乘法符号的情况下书写内容可能会令人困惑。稍后我们将学习如何定义此类符号。在命名空间 `MyNat` 中工作意味着我们可以写 `zero` 和 `succ` 而不是 `MyNat.zero` 和 `MyNat.succ`，并且这些名称的解释优先于其他解释。在命名空间之外，例如下面定义的 `add` 的完整名称是 `MyNat.add`。

如果您发现自己确实喜欢这类事情，不妨试着定义截断减法和幂运算，并证明它们的一些性质。请记住，截断减法在结果为零时会停止。要定义截断减法，定义一个前驱函数 `pred` 会很有用，该函数对任何非零数减一，并将零固定不变。函数 `pred` 可以通过简单的递归实例来定义。

```
inductive MyNat where
  | zero : MyNat
  | succ : MyNat → MyNat

namespace MyNat

def add : MyNat → MyNat → MyNat
  | x, zero => x
  | x, succ y => succ (add x y)

def mul : MyNat → MyNat → MyNat
  | x, zero => zero
  | x, succ y => add (mul x y) x

theorem zero_add (n : MyNat) : add zero n = n := by
  induction' n with n ih
  · rfl
  rw [add, ih]

theorem succ_add (m n : MyNat) : add (succ m) n = succ (add m n) := by
  induction' n with n ih
  · rfl
  rw [add, ih]
  rfl

theorem add_comm (m n : MyNat) : add m n = add n m := by
```

(continues on next page)

(continued from previous page)

```
  induction' n with n ih
  · rw [zero_add]
    rfl
  rw [add, succ_add, ih]

theorem add_assoc (m n k : MyNat) : add (add m n) k = add m (add n k) := by
  sorry
theorem mul_add (m n k : MyNat) : mul m (add n k) = add (mul m n) (mul m k) := by
  sorry
theorem zero_mul (n : MyNat) : mul zero n = zero := by
  sorry
theorem succ_mul (m n : MyNat) : mul (succ m) n = add (mul m n) n := by
  sorry
theorem mul_comm (m n : MyNat) : mul m n = mul n m := by
  sorry
end MyNat
```

5.3 无穷多个素数

让我们继续探讨归纳法和递归法，这次以另一个数学标准为例：证明存在无穷多个素数。一种表述方式是：对于每一个自然数 n ，都存在一个大于 n 的素数。要证明这一点，设 p 是 $n! + 1$ 的任意一个素因数。如果 p 小于或等于 n ，那么它能整除 $n!$ 。由于它也能整除 $n! + 1$ ，所以它能整除 1 ，这与事实相悖。因此 p 大于 n 。

要使该证明形式化，我们需要证明任何大于或等于 2 的数都有一个质因数。要做到这一点，我们需要证明任何不等于 0 或 1 的自然数都大于或等于 2。这让我们看到了形式化的一个奇特之处：往往正是像这样的简单陈述最难形式化。这里我们考虑几种实现方式。

首先，我们可以使用 `cases` 策略以及后继函数在自然数上保持顺序这一事实。

```
theorem two_le {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m := by
  cases m; contradiction
  case succ m =>
    cases m; contradiction
    repeat apply Nat.succ_le_succ
    apply zero_le
```

另一种策略是使用 `interval_cases` 这一策略，它会在所讨论的变量处于自然数或整数的某个区间内时，自动将目标分解为多个情况。请记住，您可以将鼠标悬停在它上面以查看其文档说明。

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m := by
```

(continues on next page)

(continued from previous page)

```
  by_contra h
  push_neg at h
  interval_cases m <;> contradiction
```

回想一下，`interval_cases m` 后面的分号表示接下来的策略将应用于它生成的每个情况。另一个选择是使用 `decide` 策略，它尝试找到一个决策过程来解决问题。Lean 知道，对于以有界量词 `∀ x, x < n → ...` 或 `∃ x, x < n ∧ ...` 开头的陈述，可以通过决定有限多个实例来确定其真值。

```
example {m : ℕ} (h0 : m ≠ 0) (h1 : m ≠ 1) : 2 ≤ m := by
  by_contra h
  push_neg at h
  revert h0 h1
  revert h m
  decide
```

有了 `two_le` 这个定理，让我们先证明每个大于 2 的自然数都有一个素数因子。Mathlib 包含一个函数 `Nat.minFac`，它会返回最小的素数因子，但为了学习库的新部分，我们将避免使用它，直接证明这个定理。

在这里，普通的归纳法不够用。我们想用 **强归纳法**，它允许我们通过证明对于每个自然数 n，如果 P 对所有小于 n 的值都成立，那么 P 在 n 处也成立，从而证明每个自然数 n 都具有性质 P。在 Lean 中，这个原理被称为 `Nat.strong_induction_on`，我们可以使用 `using` 关键字告诉归纳法策略使用它。请注意，当我们这样做时，就没有了基本情况；它被包含在一般的归纳步骤中。

论证过程很简单。假设 $n \geq 2$，如果 n 是质数，那么证明就完成了。如果不是，那么根据质数的定义之一，它有一个非平凡因子 m，此时我们可以对这个因子应用归纳假设。步进证明，看看具体是如何进行的。

```
theorem exists_prime_factor {n : Nat} (h : 2 ≤ n) : ∃ p : Nat, p.Prime ∧ p ∣ n := by
  by_cases np : n.Prime
  · use n, np
  induction' n using Nat.strong_induction_on with n ih
  rw [Nat.prime_def_lt] at np
  push_neg at np
  rcases np h with ⟨m, mltn, mdvdn, mne1⟩
  have : m ≠ 0 := by
    intro mz
    rw [mz, zero_dvd_iff] at mdvdn
    linarith
  have mgt2 : 2 ≤ m := two_le this mne1
  by_cases mp : m.Prime
  · use m, mp
  · rcases ih m mltn mgt2 mp with ⟨p, pp, pdvd⟩
    use p, pp
```

(continues on next page)

(continued from previous page)

```
    apply pdvd.trans mdvdn
```

现在我们可以证明我们定理的以下表述形式。看看你能否完善这个概要。你可以使用 `Nat.factorial_pos` 、`Nat.dvd_factorial` 和 `Nat.dvd_sub'` 。

```
theorem primes_infinite : ∀ n, ∃ p > n, Nat.Prime p := by
  intro n
  have : 2 ≤ Nat.factorial (n + 1) + 1 := by
    sorry
  rcases exists_prime_factor this with ⟨p, pp, pdvd⟩
  refine ⟨p, ?_, pp⟩
  show p > n
  by_contra ple
  push_neg  at ple
  have : p ∣ Nat.factorial (n + 1) := by
    sorry
  have : p ∣ 1 := by
    sorry
  show False
  sorry
```

让我们考虑上述证明的一个变体，其中不使用阶乘函数，而是假设我们得到一个有限集合 $\{p_1, \ldots, p_n\}$ ，并考虑 $\prod_{i=1}^{n} p_i + 1$ 的一个质因数。该质因数必须与每个 p_i 都不同，这表明不存在包含所有质数的有限集合。

要将此论证形式化，我们需要对有限集合进行推理。在 Lean 中，对于任何类型 `α` ，类型 `Finset α` 表示类型为 `α` 的元素的有限集合。从计算角度对有限集合进行推理需要有一个在 `α` 上测试相等性的过程，这就是下面代码片段包含假设 `[DecidableEq α]` 的原因。对于像 `ℕ` 、`ℤ` 和 `ℚ` 这样的具体数据类型，该假设会自动满足。在对实数进行推理时，可以使用经典逻辑并放弃计算解释来满足该假设。

我们使用命令 `open Finset` 来使用相关定理的更短名称。与集合的情况不同，涉及有限集的大多数等价关系并非定义上的成立，因此需要手动使用诸如 `Finset.subset_iff` 、`Finset.mem_union` 、`Finset.mem_inter` 和 `Finset.mem_sdiff` 这样的等价关系来展开。不过，`ext` 策略仍然可以用于通过证明一个有限集的每个元素都是另一个有限集的元素来证明两个有限集相等。

```
open Finset

section
variable {α : Type*} [DecidableEq α] (r s t : Finset α)

example : r ∩ (s ∪ t) ⊆ r ∩ s ∪ r ∩ t := by
  rw [subset_iff]
  intro x
```

(continues on next page)

(continued from previous page)

```
  rw [mem_inter, mem_union, mem_union, mem_inter, mem_inter]
  tauto

example : r ∩ (s ∪ t) ⊆ r ∩ s ∪ r ∩ t := by
  simp [subset_iff]
  intro x
  tauto

example : r ∩ s ∪ r ∩ t ⊆ r ∩ (s ∪ t) := by
  simp [subset_iff]
  intro x
  tauto

example : r ∩ s ∪ r ∩ t = r ∩ (s ∪ t) := by
  ext x
  simp
  tauto

end
```

我们使用了一个新技巧：`tauto` 策略（还有一个更强的 `tauto!`，这个使用经典逻辑）可以用来省去命题逻辑中的重言式。看看你能否用这些方法证明下面的两个例子。

```
example : (r ∪ s) ∩ (r ∪ t) = r ∪ s ∩ t := by
  sorry
example : (r \ s) \ t = r \ (s ∪ t) := by
  sorry
```

定理 `Finset.dvd_prod_of_mem` 告诉我们，如果一个数 `n` 是有限集合 `s` 的一个元素，那么 `n` 能整除 `∏ i in s, i`。

```
example (s : Finset ℕ) (n : ℕ) (h : n ∈ s) : n ∣ ∏ i in s, i :=
  Finset.dvd_prod_of_mem _ h
```

我们还需要知道，在 `n` 为素数且 `s` 为素数集合的情况下，其逆命题也成立。要证明这一点，我们需要以下引理，您应该能够使用定理 `Nat.Prime.eq_one_or_self_of_dvd` 来证明它。

```
theorem _root_.Nat.Prime.eq_of_dvd_of_prime {p q : ℕ}
      (prime_p : Nat.Prime p) (prime_q : Nat.Prime q) (h : p ∣ q) :
    p = q := by
  sorry
```

我们可以利用这个引理来证明，如果一个素数 p 整除有限个素数的乘积，那么它就等于其中的一个素数。

Mathlib 提供了一个关于有限集合的有用归纳原理：要证明某个性质对任意有限集合 s 成立，只需证明其对空集成立，并证明当向集合 s 中添加一个新元素 a ∉ s 时该性质仍成立。

这个原理被称为 Finset.induction_on 。当我们告诉归纳策略使用它时，还可以指定名称 a 和 s ，以及归纳步骤中假设 a ∉ s 的名称和归纳假设的名称。表达式 Finset.insert a s 表示集合 s 与单元素集合 a 的并集。恒等式 Finset.prod_empty 和 Finset.prod_insert 则提供了乘积相关的重写规则。在下面的证明中，第一个 simp 应用了 Finset.prod_empty 。逐步查看证明的开头部分，以了解归纳过程的展开，然后完成证明。

```
theorem mem_of_dvd_prod_primes {s : Finset ℕ} {p : ℕ} (prime_p : p.Prime) :
    (∀ n ∈ s, Nat.Prime n) → (p ∣ ∏ n in s, n) → p ∈ s := by
  intro h₀ h₁
  induction' s using Finset.induction_on with a s ans ih
  · simp at h₁
    linarith [prime_p.two_le]
  simp [Finset.prod_insert ans, prime_p.dvd_mul] at h₀ h₁
  rw [mem_insert]
  sorry
```

我们需要有限集合的一个最后性质。给定一个元素 s : Set α 和一个关于 α 的谓词 P ，在 第 4 章 中，我们用 { x ∈ s | P x } 表示集合 s 中满足 P 的元素。对于 s : Finset α ，类似的概念写作 s.filter P 。

```
example (s : Finset ℕ) (x : ℕ) : x ∈ s.filter Nat.Prime ↔ x ∈ s ∧ x.Prime :=
  mem_filter
```

我们现在证明关于存在无穷多个素数的另一种表述，即对于任意的 s : Finset ℕ ，都存在一个素数 p 不属于 s 。为了得出矛盾，我们假设所有的素数都在 s 中，然后缩小到一个只包含素数的集合 s' 。将该集合的元素相乘，加一，然后找到结果的一个素因数，这将得出我们所期望的矛盾。看看你能否完成下面的概要。在第一个 have 的证明中，你可以使用 Finset.prod_pos 。

```
theorem primes_infinite' : ∀ s : Finset Nat, ∃ p, Nat.Prime p ∧ p ∉ s := by
  intro s
  by_contra h
  push_neg at h
  set s' := s.filter Nat.Prime with s'_def
  have mem_s' : ∀ {n : ℕ}, n ∈ s' ↔ n.Prime := by
    intro n
    simp [s'_def]
    apply h
  have : 2 ≤ (∏ i in s', i) + 1 := by
    sorry
  rcases exists_prime_factor this with ⟨p, pp, pdvd⟩
```

(continues on next page)

(continued from previous page)

```
  have : p ∣ ∏ i in s', i := by
    sorry
  have : p ∣ 1 := by
    convert Nat.dvd_sub' pdvd this
    simp
  show False
  sorry
```

因此，我们看到了两种表述素数有无穷多个的方式：一种是说它们不受任何 n 的限制，另一种是说它们不在任何有限集合 s 中。下面的两个证明表明这两种表述是等价的。在第二个证明中，为了形成 s.filter Q，我们必须假设存在一个判定 Q 是否成立的程序。Lean 知道存在一个判定 Nat.Prime 的程序。一般来说，如果我们通过写 open Classical 来使用经典逻辑，就可以省去这个假设。在 Mathlib 中，Finset.sup s f 表示当 x 遍历 s 时 f x 的上确界，如果 s 为空且 f 的值域为 ℕ，则返回 0。在第一个证明中，我们使用 s.sup id，其中 id 是恒等函数，来表示 s 中的最大值。

```
theorem bounded_of_ex_finset (Q : ℕ → Prop) :
    (∃ s : Finset ℕ, ∀ k, Q k → k ∈ s) → ∃ n, ∀ k, Q k → k < n := by
  rintro ⟨s, hs⟩
  use s.sup id + 1
  intro k Qk
  apply Nat.lt_succ_of_le
  show id k ≤ s.sup id
  apply le_sup (hs k Qk)

theorem ex_finset_of_bounded (Q : ℕ → Prop) [DecidablePred Q] :
    (∃ n, ∀ k, Q k → k ≤ n) → ∃ s : Finset ℕ, ∀ k, Q k ↔ k ∈ s := by
  rintro ⟨n, hn⟩
  use (range (n + 1)).filter Q
  intro k
  simp [Nat.lt_succ_iff]
  exact hn k
```

对证明存在无穷多个素数的第二种方法稍作修改，即可证明存在无穷多个模 4 余 3 的素数。论证过程如下。首先，注意到如果两个数 m 和 n 的乘积模 4 余 3，那么这两个数中必有一个模 4 余 3。毕竟，这两个数都必须是奇数，如果它们都模 4 余 1，那么它们的乘积也模 4 余 1。利用这一观察结果，我们可以证明，如果某个大于 2 的数模 4 余 3，那么这个数有一个模 4 余 3 的素因数。现在假设只有有限个形如 4n + 3 的素数，设为 $p_1, \ldots, p_k$。不失一般性，我们可以假设 $p_1 = 3$。考虑乘积 $4 \prod_{i=2}^{k} p_i + 3$。显然，这个数除以 4 的余数为 3，所以它有一个形如 4n + 3 的素因数 p。p 不可能等于 3；因为 p 整除 $4 \prod_{i=2}^{k} p_i + 3$，如果 p 等于 3，那么它也会整除 $\prod_{i=2}^{k} p_i$，这意味着 p 等于 p_i 中的一个（i = 2，⋯，k），但我们已将 3 排除在该列表之外。所以 p 必须是其他 p_i 中的一个。但在这种情况下，p 会整除 $4 \prod_{i=2}^{k} p_i$ 以及 3，这与 p 不是 3 这一事实相矛盾。在 Lean 中，记号 n % m，读作 n 模 m，表示 n 除以 m 的余数。

```
example : 27 % 4 = 3 := by norm_num
```

然后我们可以将“n 除以 4 余 3”这一表述写成 `n % 4 = 3`。下面的示例和定理总结了我们接下来需要用到的关于此函数的事实。第一个命名定理是通过少量情况推理的又一示例。在第二个命名定理中，请记住分号表示后续的策略块应用于前面策略生成的所有目标。

```
example (n : ℕ) : (4 * n + 3) % 4 = 3 := by
  rw [add_comm, Nat.add_mul_mod_self_left]

theorem mod_4_eq_3_or_mod_4_eq_3 {m n : ℕ} (h : m * n % 4 = 3) : m % 4 = 3 ∨ n % 4 =␣
↪3 := by
  revert h
  rw [Nat.mul_mod]
  have : m % 4 < 4 := Nat.mod_lt m (by norm_num)
  interval_cases m % 4 <;> simp [-Nat.mul_mod_mod]
  have : n % 4 < 4 := Nat.mod_lt n (by norm_num)
  interval_cases n % 4 <;> simp

theorem two_le_of_mod_4_eq_3 {n : ℕ} (h : n % 4 = 3) : 2 ≤ n := by
  apply two_le <;>
    · intro neq
      rw [neq] at h
      norm_num at h
```

我们还需要以下事实，即如果 `m` 是 `n` 的非平凡因数，那么 `n / m` 也是。试着用 `Nat.div_dvd_of_dvd` 和 `Nat.div_lt_self` 完成证明。

```
theorem aux {m n : ℕ} (h₀ : m ∣ n) (h₁ : 2 ≤ m) (h₂ : m < n) : n / m ∣ n ∧ n / m < n␣
↪:= by
  sorry
```

现在把所有部分整合起来，证明任何模 4 余 3 的数都有一个具有相同性质的素因数。

```
theorem exists_prime_factor_mod_4_eq_3 {n : Nat} (h : n % 4 = 3) :
    ∃ p : Nat, p.Prime ∧ p ∣ n ∧ p % 4 = 3 := by
  by_cases np : n.Prime
  · use n
  induction' n using Nat.strong_induction_on with n ih
  rw [Nat.prime_def_lt] at np
  push_neg at np
  rcases np (two_le_of_mod_4_eq_3 h) with ⟨m, mltn, mdvdn, mne1⟩
  have mge2 : 2 ≤ m := by
```

(continues on next page)

(continued from previous page)

```
    apply two_le _ mne1
    intro mz
    rw [mz, zero_dvd_iff] at mdvdn
    linarith
  have neq : m * (n / m) = n := Nat.mul_div_cancel' mdvdn
  have : m % 4 = 3 ∨ n / m % 4 = 3 := by
    apply mod_4_eq_3_or_mod_4_eq_3
    rw [neq, h]
  rcases this with h1 | h1
  . sorry
  . sorry
```

我们已接近尾声。给定一个素数集合 `s`，我们需要讨论从该集合中移除 3（如果存在的话）的结果。函数 `Finset.erase` 可以处理这种情况。

```
example (m n : ℕ) (s : Finset ℕ) (h : m ∈ erase s n) : m ≠ n ∧ m ∈ s := by
  rwa [mem_erase] at h

example (m n : ℕ) (s : Finset ℕ) (h : m ∈ erase s n) : m ≠ n ∧ m ∈ s := by
  simp at h
  assumption
```

我们现在准备证明存在无穷多个模 4 余 3 的素数。请补全下面缺失的部分。我们的解法会用到 `Nat.dvd_add_iff_left` 和 `Nat.dvd_sub'`。

```
theorem primes_mod_4_eq_3_infinite : ∀ n, ∃ p > n, Nat.Prime p ∧ p % 4 = 3 := by
  by_contra h
  push_neg at h
  rcases h with ⟨n, hn⟩
  have : ∃ s : Finset Nat, ∀ p : ℕ, p.Prime ∧ p % 4 = 3 ↔ p ∈ s := by
    apply ex_finset_of_bounded
    use n
    contrapose! hn
    rcases hn with ⟨p, ⟨pp, p4⟩, pltn⟩
    exact ⟨p, pltn, pp, p4⟩
  rcases this with ⟨s, hs⟩
  have h₁ : ((4 * ∏ i in erase s 3, i) + 3) % 4 = 3 := by
    sorry
  rcases exists_prime_factor_mod_4_eq_3 h₁ with ⟨p, pp, pdvd, p4eq⟩
  have ps : p ∈ s := by
    sorry
```

(continues on next page)

(continued from previous page)

```
  have pne3 : p ≠ 3 := by
    sorry
  have : p ∣ 4 * ∏ i in erase s 3, i := by
    sorry
  have : p ∣ 3 := by
    sorry
  have : p = 3 := by
    sorry
  contradiction
```

如果您成功完成了证明，恭喜您！这是一项了不起的形式化成就。

CHAPTER

SIX

结构体（STRUCTURES）

现代数学广泛使用了代数结构，这些代数结构封装了可在不同环境中实例化的模式，而且往往有多种方法来对它们进行定义和实例化。

因此，Lean 提供了相应的方法来形式化定义这些结构并对其进行操作。此前你已经接触过一些代数结构的示例，比如在 Chapter 2 中的环（rings）和格（lattices）。本章将解释之前出现过的方括号语法，比如 `[Ring α]`、`[Lattice α]`，并介绍如何创建和使用自定义的代数结构。

如需了解更多技术细节，可以参考 Theorem Proving in Lean，以及 Anne Baanen 的这篇论文 Use and abuse of instance parameters in the Lean mathematical library。

6.1 定义结构体

广义上来说，结构体是对特定形式数据集合的约定，包括包含数据的形式以及这些数据要满足的一些约束条件。而结构体的实例则是某一组满足约束的具体数据。例如，我们可以规定一个点是由三个实数组成的三元组：

```
@[ext]
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ
```

上面用到的 `@[ext]` 注解让 Lean 自动生成一个定理，内容是当该结构体的两个实例各组成部分对应相同时，这两个实例相等。该属性也称为外延性（extensionality）。

```
#check Point.ext

example (a b : Point) (hx : a.x = b.x) (hy : a.y = b.y) (hz : a.z = b.z) : a = b := by
  ext
  repeat' assumption
```

接着我们定义一个 `Point` 结构体的实例。Lean 提供多种实例化方式来达成目的。

```
def myPoint1 : Point where
  x := 2
  y := -1
  z := 4

def myPoint2 : Point :=
  ⟨2, -1, 4⟩

def myPoint3 :=
  Point.mk 2 (-1) 4
```

第一个例子中，我们明确地指明了结构体的各个字段。在定义 `myPoint3` 时用到的函数 `Point.mk` 叫做 `Point` 结构体的构造函数（constructor），用于构造结构体成员。你也可以为构造函数指定一个不同的名字，比如 `build`。

```
structure Point' where build ::
  x : ℝ
  y : ℝ
  z : ℝ

#check Point'.build 2 (-1) 4
```

接下来的两个例子展示了如何定义结构体的函数。第二个例子中明确指出了构造函数 `Point.mk` 中的字段名，而第一个例子则使用更为简洁的匿名构造函数。Lean 能根据 `add` 的目标类型推断出所需的构造函数。通常，我们会将与结构体（例如这里的 `Point`）相关的定义和定理放在一个同名的命名空间（namespace）中。在下面的示例中，由于我们启用了 `Point` 命名空间，所以 `add` 的完整名称实际是 `Point.add`。当命名空间没被启用时，就得使用完整名称。不过这时也可以使用匿名投影记号（anonymous projection notation），它允许我们用 `a.add b` 代替 `Point.add a b`。因为 a 的类型是 `Point`，Lean 能在没有开启对应命名空间的时候将 `a.add b` 推断为 `Point.add a b`。

```
namespace Point

def add (a b : Point) : Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def add' (a b : Point) : Point where
  x := a.x + b.x
  y := a.y + b.y
  z := a.z + b.z

#check add myPoint1 myPoint2
```

(continues on next page)

(continued from previous page)

```
#check myPoint1.add myPoint2

end Point

#check Point.add myPoint1 myPoint2
#check myPoint1.add myPoint2
```

接下来我们继续在相关命名空间中添加定义，但在引用的代码片段中会省略开关命名空间相关的指令。在证明加法函数性质时，可使用 `rw` 来展开定义，并用 `ext` 将结构体两个实例之间的等号转化为它们各个组成部分之间的等号。下面的代码使用 `protected` 关键字，使得在命名空间打开的情况下定理的名字依然是 `Point.add_comm`。当我们希望避免与泛型定理如 `add_comm` 产生歧义时，这样做是有帮助的。

```
protected theorem add_comm (a b : Point) : add a b = add b a := by
  rw [add, add]
  ext <;> dsimp
  repeat' apply add_comm

example (a b : Point) : add a b = add b a := by simp [add, add_comm]
```

因为 Lean 能在内部自动展开定义并简化投影，所以有时我们需要的等式在定义上就自动成立。

```
theorem add_x (a b : Point) : (a.add b).x = a.x + b.x :=
  rfl
```

与在 Section 5.2 中定义递归函数类似，我们定义函数时也可以使用模式匹配。下面定义的 `addAlt` 和 `addAlt'` 本质上是相同的，形式上的的区别在于我们在后者的定义中使用了匿名构造函数。虽然以此方式定义函数有时会显得更加简洁，并且结构化 η-规约（structural eta-reduction）也确保了这种简写在定义上的等价性，但这么做可能会为后续的证明带来不便。

```
def addAlt : Point → Point → Point
  | Point.mk x₁ y₁ z₁, Point.mk x₂ y₂ z₂ => ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

def addAlt' : Point → Point → Point
  | ⟨x₁, y₁, z₁⟩, ⟨x₂, y₂, z₂⟩ => ⟨x₁ + x₂, y₁ + y₂, z₁ + z₂⟩

theorem addAlt_x (a b : Point) : (a.addAlt b).x = a.x + b.x := by
  rfl

theorem addAlt_comm (a b : Point) : addAlt a b = addAlt b a := by
  rw [addAlt, addAlt]
  -- the same proof still works, but the goal view here is harder to read
```

(continues on next page)

(continued from previous page)

```
  ext <;> dsimp
  repeat' apply add_comm
```

数学构造通常涉及将捆绑的信息拆分开来，并以不同的方式重新组合。这也是 Lean 和 Mathlib 提供如此多方法来高效处理这些操作的原因。作为练习，请尝试证明 `Point.add` 是满足结合律的。然后请定义点的标量乘法并证明其满足对加法的分配律。

```
protected theorem add_assoc (a b c : Point) : (a.add b).add c = a.add (b.add c) := by
  sorry

def smul (r : ℝ) (a : Point) : Point :=
  sorry

theorem smul_distrib (r : ℝ) (a b : Point) :
    (smul r a).add (smul r b) = smul r (a.add b) := by
  sorry
```

使用结构体是走向代数抽象的第一步。我们目前还没有方法将 `Point.add` 与泛型的 `+` 符号联系起来，抑或是将 `Point.add_comm`、`Point.add_assoc` 与泛型的 `add_comm`、`add_assoc` 定理联系起来。这些任务属于结构体在代数层面的应用，我们将在下一节介绍如何具体实现。对现在来说，我们只需要把结构体视作一种捆绑对象和信息的方法。

在结构体中我们不仅可以指定数据类型，还可以指定数据需要满足的约束。在 Lean 中，后者被表示为类型 `Prop` 的字段。例如，标准 2-单纯形（standard 2-simplex）定义为满足 $x \geq 0$，$y \geq 0$，$z \geq 0$，$x + y + z = 1$ 的点 (x, y, z) 的集合。如果你不熟悉这个概念，可以画个图，其实这个集合就是三维空间中以 $(1, 0, 0)$，$(0, 1, 0)$，和 $(0, 0, 1)$ 为顶点的等边三角形以及其内部。在 Lean 中可以这样形式化表示：

```
structure StandardTwoSimplex where
  x : ℝ
  y : ℝ
  z : ℝ
  x_nonneg : 0 ≤ x
  y_nonneg : 0 ≤ y
  z_nonneg : 0 ≤ z
  sum_eq : x + y + z = 1
```

请注意，最后四个字段用到的 `x`，`y`，`z` 指的就是前三个字段。我们可以定义一个从 2-单纯形到其自身的映射，该映射交换 `x` 和 `y`：

```
def swapXy (a : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := a.y
```

(continues on next page)

(continued from previous page)

```
  y := a.x
  z := a.z
  x_nonneg := a.y_nonneg
  y_nonneg := a.x_nonneg
  z_nonneg := a.z_nonneg
  sum_eq := by rw [add_comm a.y a.x, a.sum_eq]
```

有趣的来了，我们可以计算单纯形内两个点的中点。为此需要先在文件开头加上 `noncomputable section` 语句以便使用实数上的除法。

```
noncomputable section

def midpoint (a b : StandardTwoSimplex) : StandardTwoSimplex
    where
  x := (a.x + b.x) / 2
  y := (a.y + b.y) / 2
  z := (a.z + b.z) / 2
  x_nonneg := div_nonneg (add_nonneg a.x_nonneg b.x_nonneg) (by norm_num)
  y_nonneg := div_nonneg (add_nonneg a.y_nonneg b.y_nonneg) (by norm_num)
  z_nonneg := div_nonneg (add_nonneg a.z_nonneg b.z_nonneg) (by norm_num)
  sum_eq := by field_simp; linarith [a.sum_eq, b.sum_eq]
```

上面的代码中，对于 `x_nonneg`，`y_nonneg`，`z_nonneg`，我们用简洁的证明项即可建立。而对于 `sum_eq`，我们选择使用 `by` 在策略模式下进行证明。

给定参数 λ 满足 $0 \le \lambda \le 1$，可定义标准 2-单纯形中 a 和 b 两点的加权平均为 $\lambda a + (1 - \lambda)b$。请尝试参考前面的 `midpoint` 形式化定义该加权平均。

```
def weightedAverage (lambda : Real) (lambda_nonneg : 0 ≤ lambda) (lambda_le : lambda␣
↪≤ 1)
    (a b : StandardTwoSimplex) : StandardTwoSimplex :=
  sorry
```

结构体还可以依赖于参数。例如，可以将标准 2-单纯形推广到任意维数 n 下的 n -单纯形。目前阶段，你不需要对 `Fin n` 了解太多，只需知道它有 n 个元素，并且 Lean 直到如何在其上进行就和操作即可。

```
open BigOperators

structure StandardSimplex (n : ℕ) where
  V : Fin n → ℝ
  NonNeg : ∀ i : Fin n, 0 ≤ V i
  sum_eq_one : (∑ i, V i) = 1
```

(continues on next page)

(continued from previous page)

```
namespace StandardSimplex

def midpoint (n : ℕ) (a b : StandardSimplex n) : StandardSimplex n
    where
  V i := (a.V i + b.V i) / 2
  NonNeg := by
    intro i
    apply div_nonneg
    · linarith [a.NonNeg i, b.NonNeg i]
    norm_num
  sum_eq_one := by
    simp [div_eq_mul_inv, ← Finset.sum_mul, Finset.sum_add_distrib,
      a.sum_eq_one, b.sum_eq_one]
    field_simp

end StandardSimplex
```

作为额外练习，请尝试定义一下标准 n -单纯形中两点的加权平均。可以使用 `Finset.sum_add_distrib` 和 `Finset.mul_sum` 来实现相关的求和操作。

前面我们已经了解到结构体可以用来将数据和属性打包在一起。有趣的是，结构体也可以用来打包脱离具体数据的抽象属性。例如，下面的结构体 `IsLinear`，将线性性（linearity）的两个主要组成部分打包在了一起。

```
structure IsLinear (f : ℝ → ℝ) where
  is_additive : ∀ x y, f (x + y) = f x + f y
  preserves_mul : ∀ x c, f (c * x) = c * f x

section
variable (f : ℝ → ℝ) (linf : IsLinear f)

#check linf.is_additive
#check linf.preserves_mul

end
```

值得一提的是，结构体并不是打包数据的唯一方法。前面提到的 `Point` 数据结构也可以用泛型乘积类型来定义，而 `IsLinear` 可以使用简单的 `and` 来定义。

```
def Point'' :=
  ℝ × ℝ × ℝ
```

(continues on next page)

(continued from previous page)

```
def IsLinear' (f : ℝ → ℝ) :=
  (∀ x y, f (x + y) = f x + f y) ∧ ∀ x c, f (c * x) = c * f x
```

泛型类型构造甚至可以替代包含依赖关系的结构体。例如，子类型（subtype）构造结合了一段数据以及一条属性。你可以将下面例子中的 `PReal` 看作正实数类型。每个 `x : PReal` 都有两个分量：它的值，以及“它是正的”这一属性。你可以分别用 `x.val` 与 `x.property` 来访问这两个分量，其中前者的类型为 `ℝ`，后者是一个属性 `0 < x.val`。

```
def PReal :=
  { y : ℝ // 0 < y }

section
variable (x : PReal)

#check x.val
#check x.property
#check x.1
#check x.2

end
```

我们也可以用子类型来定义前面的标准 2-单纯形，以及任意维数 n 下的标准 n -单纯形。

```
def StandardTwoSimplex' :=
  { p : ℝ × ℝ × ℝ // 0 ≤ p.1 ∧ 0 ≤ p.2.1 ∧ 0 ≤ p.2.2 ∧ p.1 + p.2.1 + p.2.2 = 1 }

def StandardSimplex' (n : ℕ) :=
  { v : Fin n → ℝ // (∀ i : Fin n, 0 ≤ v i) ∧ (∑ i, v i) = 1 }
```

类似地，Sigma 类型是有序对的推广，其中第二个分量的类型依赖于第一个分量。

```
def StdSimplex := Σ n : ℕ, StandardSimplex n

section
variable (s : StdSimplex)

#check s.fst
#check s.snd

#check s.1
```

(continues on next page)

(continued from previous page)

```
#check s.2

end
```

给定 `s : StdSimplex`，它的第一个分量 `s.fst` 是个自然数，第二个分量是对应维数下标准单纯形中的一个元素 `StandardSimplex s.fst`。Sigma 类型与子类型的区别在于 Sigma 类型的第二个分量是数据而非属性。

尽管可以使用乘积类型、子类型、Sigma 类型等替代结构体，但使用结构体其实有许多好处。定义结构体可以抽象出底层的表达，且能为成员提供自定义的名称以供访问。这使得证明更加健壮：对于那些仅依赖结构体接口的证明，通常只需用新接口替换旧接口，就能让证明代码在结构体定义变更后依然有效。此外，正如我们即将看到的，Lean 支持将各种结构体编织成一个丰富的、相互关联的层次体系，以管理它们之间的交互关系。

6.2 代数结构

为了阐明我们所说的 **代数结构**这一短语的含义，考虑一些例子会有所帮助。

1. **偏序集**由集合 P 以及定义在 P 上的二元关系 $\leq$ 组成，该关系具有传递性和自反性。
2. **群**由集合 G 以及其上的一个结合二元运算、一个单位元 1 和一个将 G 中每个元素 g 映射为其逆元 g^{-1} 的函数 $g \mapsto g^{-1}$ 构成。若运算满足交换律，则称该群为阿贝尔群或交换群。
3. **格**是一种具有交和并运算的部分有序集。
4. **环**由一个（加法表示的）阿贝尔群 $(R, +, 0, x \mapsto -x)$ 以及一个结合的乘法运算 $\cdot$ 和一个单位元 1 组成，并且乘法对加法满足分配律。如果乘法是可交换的，则称环为 **交换环**。
5. 一个 **有序环** $(R, +, 0, -, \cdot, 1, \leq)$ 由一个环以及其元素上的一个偏序关系组成，满足以下条件：对于 R 中的任意 a、b 和 c，若 $a \leq b$，则 $a + c \leq b + c$；对于 R 中的任意 a 和 b，若 $0 \leq a$ 且 $0 \leq b$，则 $0 \leq ab$。

#. **度量空间**由一个集合 X 和一个函数 $d : X \times X \to \mathbb{R}$ 组成，满足以下条件：- 对于集合 X 中的任意 x 和 y，有 $d(x, y) \geq 0$ 。- 当且仅当 x = y 时，$d(x?y) = 0$ 。- 对于集合 X 中的任意 x 和 y，有 $d(x?y) = d(y?x)$ 。- 对于集合 X 中的任意 x、y 和 z，有 $d(x, z) \leq d(x, y) + d(y?z)$ 。

#. **拓扑空间**由集合 X 以及 X 的子集所构成的集合 $\mathcal{T}$ 组成，这些子集被称为 X 的开子集，且满足以下条件：- 空集和 X 是开集。- 两个开集的交集是开集。- 任意多个开集的并集是开集。

在上述每个例子中，结构的元素都属于一个集合，即 **载体集**，有时它可代表整个结构。例如，当我们说“设 G 是一个群”，然后说“设 $g \in G$”，这里 G 既代表结构本身，也代表其载体。并非每个代数结构都以这种方式与单个载体集相关联。例如，**二部图**涉及两个集合之间的关系，**伽罗瓦连接**也是如此。**范畴**也涉及两个感兴趣的集合，通常称为 **对象**和 **态射**。这些示例表明了证明助手为了支持代数推理需要完成的一些工作。首先，它需要识别结构的具体实例。数系 $\mathbb{Z}$、$\mathbb{Q}$ 和 $\mathbb{R}$ 都是有序环，我们应当能够在这些实例中的任何一个上应用关于有序环的一般定理。有时，一个具体的集合可能以不止一种方式成为某个结构的实例。例如，除了构成实分析基础的通常的 $\mathbb{R}$ 上的拓扑外，我们还可以考虑 $\mathbb{R}$ 上的 **离散**拓扑，在这种拓扑中，每个集合都是开集。其次，证明助手需要支持结构上的通用符号表示。在 Lean 中，符号 `*` 用于所有常见数系中的乘法，也用于泛指群和环中的乘法。当我们使用像 `f x * y` 这样的表达式时，Lean 必须利用关于 `f`、`x` 和 `y` 的类型信

息来确定我们所指的乘法运算。第三，它需要处理这样一个事实，即结构可以通过多种方式从其他结构继承定义、定理和符号。有些结构通过添加更多公理来扩展其他结构。交换环仍然是环，因此在环中有意义的任何定义在交换环中也有意义，任何在环中成立的定理在交换环中也成立。有些结构通过添加更多数据来扩展其他结构。例如，任何环的加法部分都是加法群。环结构添加了乘法和单位元，以及管理它们并将其与加法部分相关联的公理。有时我们可以用另一种结构来定义一种结构。任何度量空间都有一个与之相关的标准拓扑，即“度量空间拓扑”，并且任何线性序都可以关联多种拓扑。最后，重要的是要记住，数学使我们能够像使用函数和运算来定义数字一样，使用函数和运算来定义结构。群的乘积和幂次仍然是群。对于每个 n，模 n 的整数构成一个环，对于每个 $k > 0$，该环中系数的 $k \times k$ 多项式矩阵再次构成一个环。因此，我们能够像计算其元素一样轻松地计算结构。这意味着代数结构在数学中有着双重身份，既是对象集合的容器，又是独立的对象。证明助手必须适应这种双重角色。在处理具有代数结构相关联的类型的元素时，证明助手需要识别该结构并找到相关的定义、定理和符号。这一切听起来似乎工作量很大，确实如此。但 Lean 使用一小部分基本机制来完成这些任务。本节的目标是解释这些机制并展示如何使用它们。第一个要素几乎无需提及，因为它太过显而易见：从形式上讲，代数结构是 Section 6.1 中所定义的那种结构。代数结构是对满足某些公理假设的数据束的规范，我们在 Section 6.1 中看到，这正是 `structure` 命令所设计要容纳的内容。这简直是天作之合！给定一种数据类型 `α`，我们可以按如下方式定义 `α` 上的群结构。

```
structure Group₁ (α : Type*) where
  mul : α → α → α
  one : α
  inv : α → α
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  inv_mul_cancel : ∀ x : α, mul (inv x) x = one
```

请注意，在 `group₁` 的定义中，类型 `α` 是一个 **参数**。因此，您应当将对象 `struc : Group₁ α` 视为是在 `α` 上的一个群结构。我们在 Section 2.2 中看到，与 `inv_mul_cancel` 相对应的 `mul_inv_cancel` 可以从其他群公理推导出来，所以无需将其添加到定义中。

这个群的定义与 Mathlib 中的 `Group` 定义类似，我们选择使用 `Group₁` 这个名称来区分我们的版本。如果您输入 `#check Group` 并点击定义，您会看到 Mathlib 版本的 `Group` 被定义为扩展了另一个结构；我们稍后会解释如何做到这一点。如果您输入 `#print Group`，您还会看到 Mathlib 版本的 `Group` 具有许多额外的字段。出于稍后会解释的原因，有时在结构中添加冗余信息是有用的，这样就有额外的字段用于从核心数据定义的对象和函数。现在先别担心这个问题。请放心，我们的简化版本 `Group₁` 在本质上与 Mathlib 使用的群定义相同。

有时将类型与结构捆绑在一起是有用的，Mathlib 中也包含一个与以下定义等价的 `GroupCat` 结构定义：

```
structure Group₁Cat where
  α : Type*
  str : Group₁ α
```

Mathlib 版本位于 `Mathlib.Algebra.Category.GroupCat.Basic` 中，如果您在示例文件开头的导入中添加此内容，可以使用 `#check` 查看它。

出于下文会更清楚说明的原因，通常更有用的做法是将类型 `α` 与结构 `Group α` 分开。我们将这两个对象一起称为 **部分捆绑结构**，因为其表示将大部分但并非全部的组件组合成一个结构。在 Mathlib 中，当某个类型被用作群的载体类型时，通常会使用大写罗马字母如 `G` 来表示该类型。

让我们构建一个群，也就是说，一个 $\texttt{Group}_1$ 类型的元素。对于任意的类型 α 和 β，Mathlib 定义了类型 `Equiv α β`，表示 α 和 β 之间的 **等价关系**。Mathlib 还为这个类型定义了具有提示性的记号 `α ≃ β`。一个元素 `f : α ≃ β` 是 α 和 β 之间的双射，由四个部分表示：从 α 到 β 的函数 `f.toFun`，从 β 到 α 的逆函数 `f.invFun`，以及两个指定这些函数确实是彼此逆函数的性质。

```
variable (α β γ : Type*)
variable (f : α ≃ β) (g : β ≃ γ)

#check Equiv α β
#check (f.toFun : α → β)
#check (f.invFun : β → α)
#check (f.right_inv : ∀ x : β, f (f.invFun x) = x)
#check (f.left_inv : ∀ x : α, f.invFun (f x) = x)
#check (Equiv.refl α : α ≃ α)
#check (f.symm : β ≃ α)
#check (f.trans g : α ≃ γ)
```

请注意最后三个构造的创造性命名。我们将恒等函数 `Equiv.refl`、逆运算 `Equiv.symm` 和复合运算 `Equiv.trans` 视为明确的证据，表明处于双射对应关系的性质是一个等价关系。

还要注意，`f.trans g` 要求将前向函数按相反的顺序组合。Mathlib 已经声明了从 `Equiv α β` 到函数类型 `α → β` 的一个 **强制转换**，因此我们可以省略书写 `.toFun`，让 Lean 为我们插入。

```
example (x : α) : (f.trans g).toFun x = g.toFun (f.toFun x) :=
  rfl

example (x : α) : (f.trans g) x = g (f x) :=
  rfl

example : (f.trans g : α → γ) = g ∘ f :=
  rfl
```

Mathlib 还定义了 `α` 与其自身的等价关系类型 `perm α`。

```
example (α : Type*) : Equiv.Perm α = (α ≃ α) :=
  rfl
```

显然，`Equiv.Perm α` 在等价关系的组合下形成了一个群。我们将乘法定义为 `mul f g` 等于 `g.trans f`，其前向函数为 `f ∘ g`。换句话说，乘法就是我们通常所认为的双射的组合。这里我们定义这个群：

```
def permGroup {α : Type*} : Group₁ (Equiv.Perm α)
    where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

实际上，Mathlib 在文件 `GroupTheory.Perm.Basic` 中为 `Equiv.Perm α` 精确地定义了这个 `Group` 结构。和往常一样，您可以将鼠标悬停在 `permGroup` 定义中使用的定理上以查看其陈述，并且可以跳转到原始文件中的定义以了解它们是如何实现的。

在普通数学中，我们通常认为符号表示与结构是相互独立的。例如，我们可以考虑群 $(G_1, \cdot, 1, \cdot^{-1})$、$(G_2, \circ, e, i(\cdot))$ 和 $(G_3, +, 0, -)$。在第一种情况下，我们将二元运算写为 $\cdot$，单位元为 1，逆函数为 $x \mapsto x^{-1}$。在第二种和第三种情况下，我们使用所示的符号替代形式。然而，在 Lean 中对群的概念进行形式化时，符号表示与结构的联系更为紧密。在 Lean 中，任何 `Group` 的组成部分都命名为 `mul` 、`one` 和 `inv` ，稍后我们将看到乘法符号是如何与它们关联的。如果我们想使用加法符号，则使用同构结构 `AddGroup` （加法群的基础结构）。其组成部分命名为 `add` 、`zero` 和 `neg` ，相关符号也是您所期望的那样。

回想一下我们在 Section 6.1 中定义的类型 `Point` ，以及在那里定义的加法函数。这些定义在本节附带的示例文件中有所重现。作为练习，请定义一个类似于我们上面定义的 $\texttt{Group}_1$ 结构的 $\texttt{AddGroup}_1$ 结构，但使用刚刚描述的加法命名方案。在 `Point` 数据类型上定义否定和零，并在 `Point` 上定义 $\texttt{AddGroup}_1$ 结构。

```
structure AddGroup₁ (α : Type*) where
  (add : α → α → α)
  -- fill in the rest
@[ext]
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ

namespace Point

def add (a b : Point) : Point :=
  ⟨a.x + b.x, a.y + b.y, a.z + b.z⟩

def neg (a : Point) : Point := sorry

def zero : Point := sorry
```

(continues on next page)

(continued from previous page)

```
def addGroupPoint : AddGroup₁ Point := sorry

end Point
```

我们正在取得进展。现在我们知道了如何在 Lean 中定义代数结构，也知道如何定义这些结构的实例。但我们还希望将符号与结构相关联，以便在每个实例中使用它。此外，我们希望安排好，以便可以在结构上定义一个操作，并在任何特定实例中使用它，还希望安排好，以便可以在结构上证明一个定理，并在任何实例中使用它。

实际上，Mathlib 已经设置好使用通用群表示法、定义和定理来处理 `Equiv.Perm α`。

```
variable {α : Type*} (f g : Equiv.Perm α) (n : ℕ)

#check f * g
#check mul_assoc f g g⁻¹

-- 群幂，定义适用于任何群
#check g ^ n

example : f * g * g⁻¹ = f := by rw [mul_assoc, mul_inv_cancel, mul_one]

example : f * g * g⁻¹ = f :=
  mul_inv_cancel_right f g

example {α : Type*} (f g : Equiv.Perm α) : g.symm.trans (g.trans f) = f :=
  mul_inv_cancel_right f g
```

您可以检查一下，对于上面要求您定义的 `Point` 上的加法群结构，情况并非如此。我们现在要做的就是理解幕后发生的神奇操作，以便让 `Equiv.Perm α` 的示例能够像预期那样运行。

问题在于，Lean 需要能够根据我们输入的表达式中所包含的信息来 **找到**相关的符号表示和隐含的群结构。同样地，当我们输入 `x + y` 且表达式 `x` 和 `y` 的类型为 `ℝ` 时，Lean 需要将 `+` 符号解释为实数上的相关加法函数。它还必须识别类型 `ℝ` 是交换环的一个实例，这样所有关于交换环的定义和定理才能被使用。再举个例子，连续性在 Lean 中是相对于任意两个拓扑空间来定义的。当我们有 `f : ℝ → ℂ` 并输入 `Continuous f` 时，Lean 必须找到 `ℝ` 和 `ℂ` 上的相关拓扑。

这种魔力是通过三者的结合实现的。

1. **逻辑**。在任何群组中都应如此解释的定义，其参数应包含群组的类型和群组结构。同样，关于任意群组元素的定理，其开头应包含对群组类型和群组结构的全称量词。
2. **隐式参数**。类型和结构的参数通常被隐式省略，这样我们就不必书写它们，也不必在 Lean 的信息窗口中看到它们。Lean 会默默地为我们补充这些信息。

3. **类型类推断**。也称为 **类推断**，这是一种简单但强大的机制，使我们能够为 Lean 注册信息以供日后使用。当 Lean 被要求为定义、定理或符号填写隐式参数时，它可以利用已注册的信息。

而注释 `(grp : Group G)` 告诉 Lean 它应该期望明确给出该参数，注释 `{grp : Group G}` 告诉 Lean 它应该尝试从表达式中的上下文线索中推断出来，注释 `[grp : Group G]` 则告诉 Lean 应该使用类型类推断来合成相应的参数。由于使用此类参数的全部意义在于我们通常无需明确引用它们，Lean 允许我们写成 `[Group G]` 并将名称匿名化。您可能已经注意到，Lean 会选择诸如 `_inst_1` 这样的名称。自动地。当我们使用带有 `variables` 命令的匿名方括号注释时，只要变量仍在作用域内，Lean 就会自动为提及 `G` 的任何定义或定理添加参数 `[Group G]` 。

我们如何注册 Lean 进行搜索所需的信息？回到我们的群示例，我们只需做两个改动。首先，我们不用 `structure` 命令来定义群结构，而是使用关键字 `class` 来表明它是一个类推断的候选对象。其次，我们不用 `def` 来定义特定实例，而是使用关键字 `instance` 来给 Lean 注册特定实例。与类变量的名称一样，我们也可以让实例定义的名称保持匿名，因为通常我们希望 Lean 能够找到它并加以利用，而不必让我们操心其中的细节。

```
class Group₂ (α : Type*) where
  mul : α → α → α
  one : α
  inv : α → α
  mul_assoc : ∀ x y z : α, mul (mul x y) z = mul x (mul y z)
  mul_one : ∀ x : α, mul x one = x
  one_mul : ∀ x : α, mul one x = x
  inv_mul_cancel : ∀ x : α, mul (inv x) x = one

instance {α : Type*} : Group₂ (Equiv.Perm α) where
  mul f g := Equiv.trans g f
  one := Equiv.refl α
  inv := Equiv.symm
  mul_assoc f g h := (Equiv.trans_assoc _ _ _).symm
  one_mul := Equiv.trans_refl
  mul_one := Equiv.refl_trans
  inv_mul_cancel := Equiv.self_trans_symm
```

以下说明了它们的用法。

```
#check Group₂.mul

def mySquare {α : Type*} [Group₂ α] (x : α) :=
  Group₂.mul x x

#check mySquare

section
```

(continues on next page)

(continued from previous page)

```
variable {β : Type*} (f g : Equiv.Perm β)

example : Group₂.mul f g = g.trans f :=
  rfl

example : mySquare f = f.trans f :=
  rfl

end

``#check`` 命令显示， ``Group₂.mul`` 有一个隐式参数 ``[Group₂ α]`` ，
我们期望它通过类推断找到，其中 ``α`` 是 ``Group₂.mul`` 参数的类型。换句话说， ``{α : ␣
↪Type*}`` 是群元素类型的隐式参数，而 ``[Group₂ α]`` 是 ``α`` 上的群结构的隐式参数。同样地，
当我们为 ``Group₂`` 定义一个通用的平方函数 ``my_square`` 时，我们使用隐式参数 ``{α :
Type*}`` 来表示元素的类型，使用隐式参数 ``[Group₂ α]`` 来表示 ``α`` 上的 ``Group₂`` 结构。
```

在第一个示例中，当我们编写 `Group₂.mul f g` 时，f 和 g 的类型告诉 Lean 在 `Group₂.mul` 的参数 `α` 中必须实例化为 `Equiv.Perm β` 。这意味着 Lean 必须找到 `Group₂ (Equiv.Perm β)` 中的一个元素。前面的 `instance` 声明确切地告诉 Lean 如何做到这一点。问题解决了！

这种用于注册信息以便 Lean 在需要时能够找到它的简单机制非常有用。这里有一个例子。在 Lean 的基础中，数据类型 “α” 可能是空的。然而，在许多应用中，知道一个类型至少有一个元素是有用的。例如，函数 `List.headI` ，它返回列表的第一个元素，可以在列表为空时返回默认值。为了实现这一点，Lean 库定义了一个类 `Inhabited α` ，它所做的只是存储一个默认值。我们可以证明 `Point` 类型是其一个实例：

```
instance : Inhabited Point where default := ⟨0, 0, 0⟩

#check (default : Point)

example : ([] : List Point).headI = default :=
  rfl
```

类推断机制也用于泛型表示法。表达式 `x + y` 是 `Add.add x y` 的缩写，你猜对了————其中 `Add α` 是一个存储 `α` 上二元函数的类。书写 `x + y` 会告诉 Lean 查找已注册的 `[Add.add α]` 实例并使用相应的函数。下面，我们为 `Point` 注册加法函数。

```
instance : Add Point where add := Point.add

section
variable (x y : Point)
```

(continues on next page)

(continued from previous page)

```
#check x + y

example : x + y = Point.add x y :=
  rfl

end
```

通过这种方式，我们也可以将符号 + 用于其他类型的二元运算。但我们还能做得更好。我们已经看到，* 可以用于任何群，+ 可以用于任何加法群，而两者都可以用于任何环。当我们在 Lean 中定义一个新的环实例时，我们不必为该实例定义 + 和 * ，因为 Lean 知道这些运算符对于每个环都是已定义的。我们可以使用这种方法为我们的 *Group₂* 类指定符号表示法：

```
instance hasMulGroup₂ {α : Type*} [Group₂ α] : Mul α :=
  ⟨Group₂.mul⟩

instance hasOneGroup₂ {α : Type*} [Group₂ α] : One α :=
  ⟨Group₂.one⟩

instance hasInvGroup₂ {α : Type*} [Group₂ α] : Inv α :=
  ⟨Group₂.inv⟩

section
variable {α : Type*} (f g : Equiv.Perm α)

#check f * 1 * g⁻¹

def foo : f * 1 * g⁻¹ = g.symm.trans ((Equiv.refl α).trans f) :=
  rfl

end
```

在这种情况下，我们必须为实例提供名称，因为 Lean 很难想出好的默认值。使这种方法奏效的是 Lean 进行递归搜索的能力。根据我们声明的实例，Lean 可以通过找到 `Group₂ (Equiv.Perm α)` 的实例来找到 `Mul (Equiv.Perm α)` 的实例，而它能够找到 `Group₂ (Equiv.Perm α)` 的实例是因为我们已经提供了一个。Lean 能够找到这两个事实并将它们串联起来。

我们刚刚给出的例子是危险的，因为 Lean 的库中也有一个 `Group (Equiv.Perm α)` 的实例，并且乘法在任何群上都有定义。所以，找到的是哪个实例是不明确的。实际上，除非您明确指定不同的优先级，否则 Lean 会倾向于更近的声明。此外，还有另一种方法可以告诉 Lean 一个结构是另一个结构的实例，即使用 `extends` 关键字。这就是 Mathlib 指定例如每个交换环都是环的方式。您可以在 Section 7 以及 *Theorem Proving in Lean* 中的 关于类推断的章节 中找到更多信息。

一般来说，对于已经定义了记号的代数结构的实例，指定值为 * 是一个不好的主意。在 Lean 中重新定义 Group 的概念是一个人为的例子。然而，在这种情况下，群记号的两种解释都以相同的方式展开为 Equiv.trans 、Equiv.refl 和 Equiv.symm 。

作为一项类似的构造性练习，仿照 Group₂ 定义一个名为 AddGroup₂ 的类。使用 Add 、Neg 和 Zero 类为任何 AddGroup₂ 定义加法、取反和零的常规表示法。然后证明 Point 是 AddGroup₂ 的一个实例。尝试一下并确保加法群的表示法对 Point 的元素有效。

```
class AddGroup₂ (α : Type*) where
  add : α → α → α
  -- 请将剩余部分填写完整
```

我们上面已经为 Point 声明了实例 Add 、Neg 和 Zero ，这并不是什么大问题。再次强调，这两种符号合成方式应该得出相同的答案。类推断是微妙的，在使用时必须小心谨慎，因为它配置了无形中控制我们所输入表达式解释的自动化机制。然而，如果明智地使用，类推断则是一个强大的工具。正是它使得在 Lean 中进行代数推理成为可能。

6.3 构建高斯整数

现在我们将通过构建一个重要的数学对象——**高斯整数**，来说明在 Lean 中代数层次结构的使用，并证明它是欧几里得域。换句话说，按照我们一直在使用的术语，我们将定义高斯整数，并证明它们是欧几里得域结构的一个实例。

在普通的数学术语中，高斯整数集 $\mathbb{Z}[i]$ 是复数集 $\{a + bi \mid a, b \in \mathbb{Z}\}$。但我们的目标不是将其定义为复数集的一个子集，而是将其定义为一种独立的数据类型。为此，我们将高斯整数表示为一对整数，分别视为其 **实部** 和 **虚部**。

```
@[ext]
structure GaussInt where
  re : ℤ
  im : ℤ
```

我们首先证明高斯整数具有环的结构，其中 0 定义为 ⟨0, 0⟩ ，1 定义为 ⟨1, 0⟩ ，加法按点定义。为了确定乘法的定义，记住我们希望由 ⟨0, 1⟩ 表示的元素 i 是 -1 的平方根。因此，我们希望

$$\begin{aligned}(a + bi)(c + di) &= ac + bci + adi + bdi^2 \\ &= (ac - bd) + (bc + ad)i.\end{aligned}$$

这解释了下面对 Mul 的定义。

```
instance : Zero GaussInt :=
  ⟨⟨0, 0⟩⟩

instance : One GaussInt :=
```

(continues on next page)

(continued from previous page)

```
  ⟨⟨1, 0⟩⟩

instance : Add GaussInt :=
  ⟨fun x y ↦ ⟨x.re + y.re, x.im + y.im⟩⟩

instance : Neg GaussInt :=
  ⟨fun x ↦ ⟨-x.re, -x.im⟩⟩

instance : Mul GaussInt :=
  ⟨fun x y ↦ ⟨x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re⟩⟩
```

正如在 Section 6.1 中所提到的，将与某种数据类型相关的所有定义放在同名的命名空间中是个好主意。因此，在与本章相关的 Lean 文件中，这些定义是在 `GaussInt` 命名空间中进行的。请注意，这里我们直接定义了符号 `0` 、 `1` 、 `+` 、 `-` 和 `*` 的解释，而不是将它们命名为 `GaussInt.zero` 之类的名称并将其分配给这些名称。通常，为定义提供一个明确的名称很有用，例如，用于与 `simp` 和 `rewrite` 一起使用。

```
theorem zero_def : (0 : GaussInt) = ⟨0, 0⟩ :=
  rfl

theorem one_def : (1 : GaussInt) = ⟨1, 0⟩ :=
  rfl

theorem add_def (x y : GaussInt) : x + y = ⟨x.re + y.re, x.im + y.im⟩ :=
  rfl

theorem neg_def (x : GaussInt) : -x = ⟨-x.re, -x.im⟩ :=
  rfl

theorem mul_def (x y : GaussInt) :
    x * y = ⟨x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re⟩ :=
  rfl
```

对计算实部和虚部的规则进行命名，并将其声明给简化器，这也是很有用的。

```
@[simp]
theorem zero_re : (0 : GaussInt).re = 0 :=
  rfl

@[simp]
theorem zero_im : (0 : GaussInt).im = 0 :=
  rfl
```

(continues on next page)

(continued from previous page)

```
@[simp]
theorem one_re : (1 : GaussInt).re = 1 :=
  rfl

@[simp]
theorem one_im : (1 : GaussInt).im = 0 :=
  rfl

@[simp]
theorem add_re (x y : GaussInt) : (x + y).re = x.re + y.re :=
  rfl

@[simp]
theorem add_im (x y : GaussInt) : (x + y).im = x.im + y.im :=
  rfl

@[simp]
theorem neg_re (x : GaussInt) : (-x).re = -x.re :=
  rfl

@[simp]
theorem neg_im (x : GaussInt) : (-x).im = -x.im :=
  rfl

@[simp]
theorem mul_re (x y : GaussInt) : (x * y).re = x.re * y.re - x.im * y.im :=
  rfl

@[simp]
theorem mul_im (x y : GaussInt) : (x * y).im = x.re * y.im + x.im * y.re :=
  rfl
```

现在令人惊讶地容易证明高斯整数是交换环的一个实例。我们正在很好地利用结构概念。每个特定的高斯整数都是 `GaussInt` 结构的一个实例，而 `GaussInt` 类型本身，连同相关操作，则是 `CommRing` 结构的一个实例。`CommRing` 结构反过来又扩展了符号结构 `Zero` 、`One` 、`Add` 、`Neg` 和 `Mul` 。

如果您输入 `instance : CommRing GaussInt := _` ，点击 VS Code 中出现的灯泡图标，然后让 Lean 自动填充结构定义的框架，您会看到大量的条目。然而，跳转到结构的定义会发现，其中许多字段都有默认定义，Lean 会自动为您填充。下面的定义中列出了关键的几个。

如果您输入 `instance : CommRing GaussInt := _` ，点击 VS Code 中出现的灯泡图标，然后让 Lean 填充

结构定义的骨架，您会看到令人望而生畏的条目数量。然而，跳转到该结构的定义后会发现，许多字段都有默认定义，Lean 会自动为您填充。以下定义中展示的是其中关键的部分。`nsmul` 和 `zsmul` 是一个特殊情况，暂时可以忽略，它们将在下一章中解释。在每种情况下，相关的恒等式都是通过展开定义、使用 `ext` 策略将恒等式简化为其实部和虚部、进行简化，并在必要时在整数中执行相关的环计算来证明的。需要注意的是，我们本可以轻松避免重复所有这些代码，但这并不是当前讨论的主题。

```
instance instCommRing : CommRing GaussInt where
  zero := 0
  one := 1
  add := (· + ·)
  neg x := -x
  mul := (· * ·)
  nsmul := nsmulRec
  zsmul := zsmulRec
  add_assoc := by
    intros
    ext <;> simp <;> ring
  zero_add := by
    intro
    ext <;> simp
  add_zero := by
    intro
    ext <;> simp
  neg_add_cancel := by
    intro
    ext <;> simp
  add_comm := by
    intros
    ext <;> simp <;> ring
  mul_assoc := by
    intros
    ext <;> simp <;> ring
  one_mul := by
    intro
    ext <;> simp
  mul_one := by
    intro
    ext <;> simp
  left_distrib := by
    intros
    ext <;> simp <;> ring
```

(continues on next page)

(continued from previous page)

```
  right_distrib := by
    intros
    ext <;> simp <;> ring
  mul_comm := by
    intros
    ext <;> simp <;> ring
  zero_mul := by
    intros
    ext <;> simp
  mul_zero := by
    intros
    ext <;> simp
```

Lean 的库将具有至少两个不同元素的类型定义为 **非平凡**类型。在环的上下文中，这等同于说零不等于一。由于一些常见的定理依赖于这一事实，我们不妨现在就证明它。

```
instance : Nontrivial GaussInt := by
  use 0, 1
  rw [Ne, GaussInt.ext_iff]
  simp
```

我们现在将证明高斯整数具有一个重要的额外性质。一个 **欧几里得整环（Euclidean domain）**是一个环 R，配备了一个范数函数 $N : R \to \mathbb{N}$，满足以下两个性质：

-对于 R 中的任意 a 和非零 b，存在 R 中的 q 和 r，使得 $a = bq + r$，并且要么 $r = 0$，要么 $N(r) < N(b)$。-对于任意 a 和非零 b，$N(a) \leq N(ab)$。整数环 $\mathbb{Z}$ 与范数 $N(a) = |a|$ 是欧几里得整环的一个典型例子。在这种情况下，我们可以将 q 取为 a 除以 b 的整数除法结果，并将 r 取为余数。这些函数在 Lean 中定义，并满足以下性质：

```
example (a b : ℤ) : a = b * (a / b) + a % b :=
  Eq.symm (Int.ediv_add_emod a b)

example (a b : ℤ) : b ≠ 0 → 0 ≤ a % b :=
  Int.emod_nonneg a

example (a b : ℤ) : b ≠ 0 → a % b < |b| :=
  Int.emod_lt a
```

在任意环中，若一个元素 a 能整除 1，则称其为 **单位元**。若一个非零元素 a 不能写成 $a = bc$ 的形式，其中 b 和 c 均不是单位元，则称其为 **不可约元**。在整数环中，每个不可约元 a 都是 **素元**，也就是说，每当 a 能整除乘积 bc 时，它就能整除 b 或 c 。但是在其他环中，这一性质可能会失效。在环 $\mathbb{Z}[\sqrt{-5}]$ 中，我们有

$$6 = 2 \cdot 3 = (1 + \sqrt{-5})(1 - \sqrt{-5}),$$

并且元素 2 、3 、$1+\sqrt{-5}$ 以及 $1-\sqrt{-5}$ 都是不可约的，但它们不是素元。例如，2 能整除乘积 $(1+\sqrt{-5})(1-\sqrt{-5})$ ，但它不能整除这两个因数中的任何一个。特别地，我们不再具有唯一分解性：数字 6 可以用不止一种方式分解为不可约元素。

相比之下，每个欧几里得域都是唯一分解域，这意味着每个不可约元素都是素元。欧几里得域的公理意味着可以将任何非零元素表示为不可约元素的有限乘积。它们还意味着可以使用欧几里得算法找到任意两个非零元素 `a` 和 `b` 的最大公约数，即能被任何其他公因数整除的元素。这反过来又意味着，除了乘以单位元素外，分解为不可约元素是唯一的。

我们现在证明高斯整数是一个欧几里得域，其范数定义为 $N(a+bi) = (a+bi)(a-bi) = a^2 + b^2$ 。高斯整数 $a - bi$ 被称为 $a + bi$ 的共轭。不难验证，对于任何复数 x 和 y ，我们都有 $N(xy) = N(x)N(y)$ 。

要证明这个范数的定义能使高斯整数构成欧几里得域，只有第一个性质具有挑战性。假设我们想写成 $a+bi = (c+di)q + r$ 的形式，其中 q 和 r 是合适的数。将 $a + bi$ 和 $c + di$ 视为复数，进行除法运算。

$$\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{(c+di)(c-di)} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i.$$

实部和虚部可能不是整数，但我们可将其四舍五入到最接近的整数 u 和 v 。然后我们可以将右边表示为 $(u + vi) + (u' + v'i)$ ，其中 $u' + v'i$ 是剩余的部分。请注意，我们有 $|u'| \leq 1/2$ 和 $|v'| \leq 1/2$ ，因此

$$N(u' + v'i) = (u')^2 + (v')^2 \leq 1/4 + 1/4 \leq 1/2.$$

乘以 $c + di$，我们有

$$a + bi = (c + di)(u + vi) + (c + di)(u' + v'i).$$

Setting $q = u + vi$ and $r = (c + di)(u' + v'i)$, we have $a + bi = (c + di)q + r$, and we only need to bound $N(r)$: 设 $q = u + vi$ 以及 $r = (c + di)(u' + v'i)$ ，则有 $a + bi = (c + di)q + r$ ，我们只需对 $N(r)$ 进行限制：

$$N(r) = N(c + di)N(u' + v'i) \leq N(c + di) \cdot 1/2 < N(c + di).$$

我们刚刚进行的论证需要将高斯整数视为复数的一个子集。因此，在 Lean 中对其进行形式化的一种选择是将高斯整数嵌入到复数中，将整数嵌入到高斯整数中，定义从实数到整数的舍入函数，并且要非常小心地在这些数系之间进行适当的转换。实际上，这正是 Mathlib 中所采用的方法，其中高斯整数本身是作为 **二次整数**环的一个特例来构造的。请参阅文件 GaussianInt.lean.

在这里，我们将进行一个完全在整数范围内进行的论证。这说明了在将数学形式化时人们通常面临的一种选择。当一个论证需要库中尚未包含的概念或工具时，人们有两个选择：要么形式化所需的这些概念或工具，要么调整论证以利用已有的概念和工具。从长远来看，当结果可以在其他情境中使用时，第一个选择通常是对时间的良好投资。然而，从实际角度来看，有时寻找一个更基础的证明会更有效。

整数的常规商余定理指出，对于任意整数 a 和非零整数 b，都存在整数 q 和 r 使得 $a = bq + r$ 且 $0 \leq r < b$。这里我们将使用以下变体，即存在整数 q' 和 r' 使得 $a = bq' + r'$ 且 $|r'| \leq b/2$。您可以验证，如果第一个陈述中的 r 满足 $r \leq b/2$，则可以取 $q' = q$ 和 $r' = r$，否则可以取 $q' = q + 1$ 和 $r' = r - b$。我们感谢 Heather Macbeth 提出以下更优雅的方法，该方法避免了按情况定义。我们只需在除法前将 `b / 2` 加到 `a` 上，然后从余数中减去它。

```
def div' (a b : ℤ) :=
  (a + b / 2) / b

def mod' (a b : ℤ) :=
  (a + b / 2) % b - b / 2

theorem div'_add_mod' (a b : ℤ) : b * div' a b + mod' a b = a := by
  rw [div', mod']
  linarith [Int.ediv_add_emod (a + b / 2) b]

theorem abs_mod'_le (a b : ℤ) (h : 0 < b) : |mod' a b| ≤ b / 2 := by
  rw [mod', abs_le]
  constructor
  · linarith [Int.emod_nonneg (a + b / 2) h.ne']
  have := Int.emod_lt_of_pos (a + b / 2) h
  have := Int.ediv_add_emod b 2
  have := Int.emod_lt_of_pos b zero_lt_two
  revert this; intro this -- 待修复，这应该是不必要的
  linarith
```

请注意我们老朋友 `linarith` 的使用。我们还需要用 `div'` 来表示 `mod'` 。

```
theorem mod'_eq (a b : ℤ) : mod' a b = a - b * div' a b := by linarith [div'_add_mod'␣
↪a b]
```

我们将使用这样一个事实，即 $x^2 + y^2$ 等于零当且仅当 x 和 y 都为零。作为练习，我们要求您证明在任何有序环中都成立。

```
theorem sq_add_sq_eq_zero {α : Type*} [LinearOrderedRing α] (x y : α) :
    x ^ 2 + y ^ 2 = 0 ↔ x = 0 ∧ y = 0 := by
  sorry
```

我们将本节中剩余的所有定义和定理都放在 `GaussInt` 命名空间中。首先，我们定义 `norm` 函数，并请您证明其部分性质。这些证明都很简短。

```
def norm (x : GaussInt) :=
  x.re ^ 2 + x.im ^ 2

@[simp]
theorem norm_nonneg (x : GaussInt) : 0 ≤ norm x := by
  sorry
theorem norm_eq_zero (x : GaussInt) : norm x = 0 ↔ x = 0 := by
```

(continues on next page)

(continued from previous page)

```
  sorry
theorem norm_pos (x : GaussInt) : 0 < norm x ↔ x ≠ 0 := by
  sorry
theorem norm_mul (x y : GaussInt) : norm (x * y) = norm x * norm y := by
  sorry
```

接下来我们定义共轭函数：

```
def conj (x : GaussInt) : GaussInt :=
  ⟨x.re, -x.im⟩

@[simp]
theorem conj_re (x : GaussInt) : (conj x).re = x.re :=
  rfl

@[simp]
theorem conj_im (x : GaussInt) : (conj x).im = -x.im :=
  rfl

theorem norm_conj (x : GaussInt) : norm (conj x) = norm x := by simp [norm]
```

最后，我们为高斯整数定义除法运算，记作 `x / y`，将复数商四舍五入到最近的高斯整数。为此我们使用自定义的 `Int.div'`。正如我们上面计算的那样，如果 `x` 是 $a+bi$，而 `y` 是 $c+di$，那么 `x / y` 的实部和虚部分别是如下式子最近的整数

$$\frac{ac+bd}{c^2+d^2} \quad \text{和} \quad \frac{bc-ad}{c^2+d^2},$$

这里分子是 $(a+bi)(c-di)$ 的实部和虚部，而分母都等于 $c+di$ 的范数。

```
instance : Div GaussInt :=
  ⟨fun x y ↦ ⟨Int.div' (x * conj y).re (norm y), Int.div' (x * conj y).im (norm y)⟩⟩
```

定义了 `x / y` 之后，我们定义 `x % y` 为余数，即 `x - (x / y) * y`。同样地，我们将这些定义记录在定理 `div_def` 和 `mod_def` 中，以便使用 `simp` 和 `rewrite`。

```
instance : Mod GaussInt :=
  ⟨fun x y ↦ x - y * (x / y)⟩

theorem div_def (x y : GaussInt) :
    x / y = ⟨Int.div' (x * conj y).re (norm y), Int.div' (x * conj y).im (norm y)⟩ :=
  rfl
```

(continues on next page)

(continued from previous page)

```
theorem mod_def (x y : GaussInt) : x % y = x - y * (x / y) :=
  rfl
```

这些定义立即得出对于每个 `x` 和 `y` 都有 `x = y * (x / y) + x % y`，所以我们需要做的就是证明当 `y` 不为零时，`x % y` 的范数小于 `y` 的范数。

我们刚刚将 `x / y` 的实部和虚部分别定义为 `div' (x * conj y).re (norm y)` and `div' (x * conj y).im (norm y)`。计算得出，我们有

```
(x % y) * conj y = (x - x / y * y) * conj y = x * conj y - x / y * (y * conj y)
```

右侧实部和虚部恰好为 `mod' (x * conj y).re (norm y)` 和 `mod' (x * conj y).im (norm y)`。根据 `div'` 和 `mod'` 的性质，可以保证它们都小于或等于 `norm y / 2`。因此我们有

```
norm ((x % y) * conj y) ≤ (norm y / 2)^2 + (norm y / 2)^2 ≤ (norm y / 2) * norm
y.
```

另一方面，我们有

```
norm ((x % y) * conj y) = norm (x % y) * norm (conj y) = norm (x % y) * norm y.
```

两边同时除以 `norm y`，我们得到 `norm (x % y) ≤ (norm y) / 2 < norm y`，如所要求的那样。

这个杂乱的计算将在接下来的证明中进行。我们鼓励您仔细查看细节，看看能否找到更简洁的论证方法。

```
theorem norm_mod_lt (x : GaussInt) {y : GaussInt} (hy : y ≠ 0) :
    (x % y).norm < y.norm := by
  have norm_y_pos : 0 < norm y := by rwa [norm_pos]
  have H1 : x % y * conj y = ⟨Int.mod' (x * conj y).re (norm y), Int.mod' (x * conj␣
↪y).im (norm y)⟩
  · ext <;> simp [Int.mod'_eq, mod_def, div_def, norm] <;> ring
  have H2 : norm (x % y) * norm y ≤ norm y / 2 * norm y
  · calc
      norm (x % y) * norm y = norm (x % y * conj y) := by simp only [norm_mul, norm_
↪conj]
      _ = |Int.mod' (x.re * y.re + x.im * y.im) (norm y)| ^ 2
          + |Int.mod' (-(x.re * y.im) + x.im * y.re) (norm y)| ^ 2 := by simp [H1,␣
↪norm, sq_abs]
      _ ≤ (y.norm / 2) ^ 2 + (y.norm / 2) ^ 2 := by gcongr <;> apply Int.abs_mod'_le _
↪ _ norm_y_pos
      _ = norm y / 2 * (norm y / 2 * 2) := by ring
      _ ≤ norm y / 2 * norm y := by gcongr; apply Int.ediv_mul_le; norm_num
  calc norm (x % y) ≤ norm y / 2 := le_of_mul_le_mul_right H2 norm_y_pos
    _ < norm y := by
        apply Int.ediv_lt_of_lt_mul
```

(continues on next page)

(continued from previous page)

```
      · norm_num
      · linarith
```

我们即将完成。我们的 `norm` 函数将高斯整数映射到非负整数。我们需要一个将高斯整数映射到自然数的函数，我们通过将 `norm` 与将整数映射到自然数的函数 `Int.natAbs` 组合来获得它。接下来的两个引理中的第一个确立了将范数映射到自然数然后再映射回整数不会改变其值。第二个引理重新表述了范数递减这一事实。

```
theorem coe_natAbs_norm (x : GaussInt) : (x.norm.natAbs : ℤ) = x.norm :=
  Int.natAbs_of_nonneg (norm_nonneg _)

theorem natAbs_norm_mod_lt (x y : GaussInt) (hy : y ≠ 0) :
    (x % y).norm.natAbs < y.norm.natAbs := by
  apply Int.ofNat_lt.1
  simp only [Int.natCast_natAbs, abs_of_nonneg, norm_nonneg]
  apply norm_mod_lt x hy
```

我们还需要确立范数函数在欧几里得域上的第二个关键性质。

```
theorem not_norm_mul_left_lt_norm (x : GaussInt) {y : GaussInt} (hy : y ≠ 0) :
    ¬(norm (x * y)).natAbs < (norm x).natAbs := by
  apply not_lt_of_ge
  rw [norm_mul, Int.natAbs_mul]
  apply le_mul_of_one_le_right (Nat.zero_le _)
  apply Int.ofNat_le.1
  rw [coe_natAbs_norm]
  exact Int.add_one_le_of_lt ((norm_pos _).mpr hy)
```

现在我们可以将其整合起来，证明高斯整数是欧几里得域的一个实例。我们使用已定义的商和余数函数。Mathlib 中欧几里得域的定义比上面的更通用，因为它允许我们证明余数相对于任何良基度量都是递减的。比较返回自然数的范数函数的值是只需这样一个度量的一个实例，在这种情况下，所需的性质就是定理 `natAbs_norm_mod_lt` 和 `not_norm_mul_left_lt_norm` 。

```
instance : EuclideanDomain GaussInt :=
  { GaussInt.instCommRing with
    quotient := (· / ·)
    remainder := (· % ·)
    quotient_mul_add_remainder_eq :=
      fun x y ↦ by simp only; rw [mod_def, add_comm] ; ring
    quotient_zero := fun x ↦ by
      simp [div_def, norm, Int.div']
      rfl
```

(continues on next page)

(continued from previous page)

```
    r := (measure (Int.natAbs ∘ norm)).1
    r_wellFounded := (measure (Int.natAbs ∘ norm)).2
    remainder_lt := natAbs_norm_mod_lt
    mul_left_not_lt := not_norm_mul_left_lt_norm }
```

一个直接的好处是，我们现在知道，在高斯整数中，素数和不可约数的概念是一致的。

```
example (x : GaussInt) : Irreducible x ↔ Prime x :=
  irreducible_iff_prime
```

CHAPTER

SEVEN

层次结构

在 第 6 章 中，我们已经了解了如何定义群类并创建此类的实例，以及如何创建交换环类的实例。但当然这里存在一个层次结构：交换环尤其是一种加法群。在本章中，我们将研究如何构建这样的层次结构。它们出现在数学的所有分支中，但在本章中，重点将放在代数示例上。在更多地讨论如何使用现有的层级结构之前，就来探讨如何构建层级结构似乎为时尚早。但要使用层级结构，就需要对其底层技术有一定的了解。所以您或许还是应该读一下这一章，不过在第一次阅读时不必太努力记住所有内容，先读完后面的章节，然后再回过头来重读这一章。在这一章中，我们将重新定义（更简单的版本）出现在 Mathlib 中的许多内容，因此我们将使用索引来区分我们的版本。例如，我们将把我们的版本的 `Ring` 称为 $\texttt{Ring}_1$ 。由于我们将逐步介绍更强大的形式化结构的方法，这些索引有时会超过 1。

7.1 基础

在 Lean 的所有层次结构的最底层，我们能找到承载数据的类。下面这个类记录了给定类型 `α` 具有一个被称为 `one` 的特殊元素。在现阶段，它没有任何属性。

```
class One₁ (α : Type) where
  /-- The element one -/
  one : α
```

由于在本章中我们将更频繁地使用类，所以我们需要进一步了解 `class` 命令的作用。首先，上述的 `class` 命令定义了一个结构体 $\texttt{One}_1$ ，其参数为 `α : Type` ，且只有一个字段 `one` 。它还标记此结构体为类，这样对于某些类型 `α` 的 $\texttt{One}_1$ `α` 类型的参数，只要它们被标记为实例隐式参数（即出现在方括号中），就可以通过实例解析过程进行推断。这两个效果也可以通过带有 `class` 属性的 `structure` 命令来实现，即写成 `@[class] structure` 的形式。但 `class` 命令还确保了 $\texttt{One}_1$ `α` 在其自身的字段中以实例隐式参数的形式出现。比较：

```
#check One₁.one -- One₁.one {α : Type} [self : One₁ α] : α

@[class] structure One₂ (α : Type) where
  /-- The element one -/
  one : α

#check One₂.one
```

在第二次检查中，我们可以看到 `self : One₂ α` 是一个显式参数。让我们确保第一个版本确实可以在没有任何显式参数的情况下使用。

```
example (α : Type) [One₁ α] : α := One₁.one
```

备注：在上述示例中，参数 `One₁ α` 被标记为实例隐式，这有点蠢，因为这仅影响该声明的 **使用**，而由 `example` 命令创建的声明无法被使用。不过，这使我们能够避免为该参数命名，更重要的是，它开始养成将 `One₁ α` 参数标记为实例隐式的良好习惯。

另一个需要注意的地方是，只有当 Lean 知道 α 是什么时，上述所有内容才会起作用。在上述示例中，如果省略类型标注： α，则会生成类似以下的错误消息：

```
typeclass instance problem is stuck, it is often due to metavariables One₁ (?m.263
α)
```

其中 `?m.263 α` 表示“依赖于 α 的某种类型”（263 只是一个自动生成的索引，用于区分多个未知事物）。另一种避免此问题的方法是使用类型注解，例如：

```
example (α : Type) [One₁ α] := (One₁.one : α)
```

如果您在 Section 3.6 中尝试处理数列的极限时遇到过这个问题，那么您可能已经遇到过这种情况，比如您试图声明 `0 < 1`，但没有告诉 Lean 您是想表示自然数之间的不等式还是实数之间的不等式。

我们的下一个任务是为 `One₁.one` 指定一个符号。由于我们不想与内置的 `1` 符号发生冲突，我们将使用 `𝟙`。这是通过以下命令实现的，其中第一行告诉 Lean 使用 `One₁.one` 的文档作为符号 `𝟙` 的文档。

```
@[inherit_doc]
notation "𝟙" => One₁.one

example {α : Type} [One₁ α] : α := 𝟙

example {α : Type} [One₁ α] : (𝟙 : α) = 𝟙 := rfl
```

我们现在想要一个记录二元运算的数据承载类。目前我们不想在加法和乘法之间做出选择，所以我们将使用菱形。

```
class Dia₁ (α : Type) where
  dia : α → α → α

infixl:70 " ⋄ "   => Dia₁.dia
```

与 `One₁` 示例一样，此时该运算没有任何属性。现在让我们定义一个半群结构类，其中运算用 ⋄ 表示。目前，我们手动将其定义为具有两个字段的结构，一个 `Dia₁` 实例和一个值为 `Prop` 的字段 `dia_assoc`，用于断言 ⋄ 的结合性。

```
class Semigroup₁ (α : Type) where
  toDia₁ : Dia₁ α
  /-- Diamond is associative -/
  dia_assoc : ∀ a b c : α, a ⋄ b ⋄ c = a ⋄ (b ⋄ c)
```

请注意，在声明 `dia_assoc` 时，先前定义的字段 `toDia₁` 在本地上下文中，因此在 Lean 搜索 `Dia₁ α` 的实例以理解 `a ⋄ b` 时可以使用它。但是这个 `toDia₁` 字段不会成为类型类实例数据库的一部分。因此，执行 `example {α : Type} [Semigroup₁ α] (a b : α) : α := a ⋄ b` 将会失败。错误消息 `failed to synthesize instance Dia₁ α` 。

我们可以通过稍后添加 `instance` 属性来解决这个问题。

```
attribute [instance] Semigroup₁.toDia₁

example {α : Type} [Semigroup₁ α] (a b : α) : α := a ⋄ b
```

在构建之前，我们需要一种比显式编写诸如 *toDia₁* 这样的字段并手动添加实例属性更方便的方式来扩展结构。`class` 使用如下语法通过 `extends` 支持这一点：

```
class Semigroup₂ (α : Type) extends Dia₁ α where
  /-- Diamond is associative -/
  dia_assoc : ∀ a b c : α, a ⋄ b ⋄ c = a ⋄ (b ⋄ c)

example {α : Type} [Semigroup₂ α] (a b : α) : α := a ⋄ b
```

请注意，这种语法在 `structure` 命令中也是可用的，尽管在这种情况下，它仅解决了编写诸如 *toDia₁* 这样的字段的麻烦，因为在那种情况下没有实例需要定义。

现在让我们尝试将一个菱形运算和一个特殊操作结合起来，并用公理说明这个元素在两边都是零元。

```
class DiaOneClass₁ (α : Type) extends One₁ α, Dia₁ α where
  /-- 存在一个对于菱形运算的左零元。 -/
  one_dia : ∀ a : α, 1 ⋄ a = a
  /-- 存在一个对于菱形运算的右零元。 -/
  dia_one : ∀ a : α, a ⋄ 1 = a
```

在下一个示例中，我们告诉 Lean `α` 具有 `DiaOneClass₁` 结构，并声明一个使用 `Dia₁` 实例和 `One₁` 实例的属性。为了查看 Lean 如何找到这些实例，我们设置了一个跟踪选项，其结果可以在 Infoview 中看到。默认情况下，该结果相当简略，但可以通过点击以黑色箭头结尾的行来展开。它包括 Lean 在拥有足够的类型信息之前尝试查找实例但未成功的尝试。成功的尝试确实涉及由 `extends` 语法生成的实例。

```
set_option trace.Meta.synthInstance true in
example {α : Type} [DiaOneClass₁ α] (a b : α) : Prop := a ⋄ b = 1
```

请注意，在组合现有类时，我们不需要包含额外的字段。因此，我们可以将半群定义为：

```
class Monoid₁ (α : Type) extends Semigroup₁ α, DiaOneClass₁ α
```

虽然上述定义看起来很简单，但它隐藏了一个重要的微妙之处。`Semigroup₁ α` 和 `DiaOneClass₁ α` 都扩展了 `Dia₁ α`，所以有人可能会担心，拥有一个 `Monoid₁ α` 实例会在 `α` 上产生两个不相关的菱形运算，一个来自字段 `Monoid₁.toSemigroup₁`，另一个来自字段 `Monoid₁.toDiaOneClass₁`。

实际上，如果我们尝试通过以下方式手动构建一个单子类：

```
class Monoid₂ (α : Type) where
  toSemigroup₁ : Semigroup₁ α
  toDiaOneClass₁ : DiaOneClass₁ α
```

那么我们会得到两个完全不相关的菱形运算

`Monoid₂.toSemigroup₁.toDia₁.dia` 和 `Monoid₂.toDiaOneClass₁.toDia₁.dia`。

使用 `extends` 语法生成的版本不存在此缺陷。

```
example {α : Type} [Monoid₁ α] :
  (Monoid₁.toSemigroup₁.toDia₁.dia : α → α → α) = Monoid₁.toDiaOneClass₁.toDia₁.dia
↪:= rfl
```

所以 `class` 命令为我们做了些神奇的事（`structure` 命令也会这样做）。查看类的构造函数是了解其字段的简便方法。比较：

```
/- Monoid₂.mk {α : Type} (toSemigroup₁ : Semigroup₁ α) (toDiaOneClass₁ : DiaOneClass₁
↪α) : Monoid₂ α -/
#check Monoid₂.mk

/- Monoid₁.mk {α : Type} [toSemigroup₁ : Semigroup₁ α] [toOne₁ : One₁ α] (one_dia : ∀
↪(a : α), 1 * a = a) (dia_one : ∀ (a : α), a * 1 = a) : Monoid₁ α -/
#check Monoid₁.mk
```

所以我们看到 `Monoid₁` 按预期接受 `Semigroup₁ α` 参数，但不会接受可能重叠的 `DiaOneClass₁ α` 参数，而是将其拆分并仅包含不重叠的部分。它还自动生成了一个实例 `Monoid₁.toDiaOneClass₁`，这并非字段，但具有预期的签名，从最终用户的角度来看，这恢复了两个扩展类 `Semigroup₁` 和 `DiaOneClass₁` 之间的对称性。

```
#check Monoid₁.toSemigroup₁
#check Monoid₁.toDiaOneClass₁
```

我们现在离定义群非常接近了。我们可以在单子结构中添加一个字段，断言每个元素都存在逆元。但那样的话，我们需要努力才能访问这些逆元。实际上，将其作为数据添加会更方便。为了优化可重用性，我们定义一个新的携带数据的类，然后为其提供一些符号表示。

```
class Inv₁ (α : Type) where
  /-- The inversion function -/
  inv : α → α

@[inherit_doc]
postfix:max "⁻¹" => Inv₁.inv

class Group₁ (G : Type) extends Monoid₁ G, Inv₁ G where
  inv_dia : ∀ a : G, a⁻¹ * a = 1
```

上述定义可能看起来太弱了，我们只要求 `a⁻¹` 是 `a` 的左逆元。但另一侧是自动的。为了证明这一点，我们需要一个预备引理。

```
lemma left_inv_eq_right_inv₁ {M : Type} [Monoid₁ M] {a b c : M} (hba : b * a = 1)␣
↪(hac : a * c = 1) : b = c := by
  rw [← DiaOneClass₁.one_dia c, ← hba, Semigroup₁.dia_assoc, hac, DiaOneClass₁.dia_
↪one b]
```

在这个引理中，给出全名相当烦人，尤其是因为这需要知道这些事实是由层次结构中的哪一部分提供的。解决这个问题的一种方法是使用 `export` 命令将这些事实作为引理复制到根名称空间中。

```
export DiaOneClass₁ (one_dia dia_one)
export Semigroup₁ (dia_assoc)
export Group₁ (inv_dia)
```

然后我们可以将上述证明重写为：

```
example {M : Type} [Monoid₁ M] {a b c : M} (hba : b * a = 1) (hac : a * c = 1) : b =␣
↪c := by
  rw [← one_dia c, ← hba, dia_assoc, hac, dia_one b]
```

现在轮到你来证明我们代数结构的性质了。

```
lemma inv_eq_of_dia [Group₁ G] {a b : G} (h : a * b = 1) : a⁻¹ = b :=
  sorry

lemma dia_inv [Group₁ G] (a : G) : a * a⁻¹ = 1 :=
  sorry
```

在这个阶段，我们想继续定义环，但有一个严重的问题。一个类型的环结构包含一个加法群结构和一个乘法半群结构，以及它们相互作用的一些性质。但到目前为止，我们为所有操作硬编码了一个符号 `*`。更根本的是，类型类系统假定每个类型对于每个类型类只有一个实例。有多种方法可以解决这个问题。令人惊讶的是，Mathlib 使用了一个朴素的想法，借助一些代码生成属性，为加法和乘法理论复制了所有内容。结构和类以加

法和乘法两种记法定义，并通过属性 `to_additive` 相互关联。对于像半群这样的多重继承情况，自动生成的“恢复对称性”的实例也需要进行标记。这有点技术性，您无需理解细节。重要的是，引理仅以乘法记法陈述，并通过属性 `to_additive` 标记，以生成加法版本为 `left_inv_eq_right_inv'` ，其自动生成的加法版本为 `left_neg_eq_right_neg'` 。为了检查这个加法版本的名称，我们在 `left_inv_eq_right_inv'` 之上使用了 `whatsnew in` 命令。

```
class AddSemigroup₃ (α : Type) extends Add α where
/-- Addition is associative -/
  add_assoc₃ : ∀ a b c : α, a + b + c = a + (b + c)

@[to_additive AddSemigroup₃]
class Semigroup₃ (α : Type) extends Mul α where
/-- Multiplication is associative -/
  mul_assoc₃ : ∀ a b c : α, a * b * c = a * (b * c)

class AddMonoid₃ (α : Type) extends AddSemigroup₃ α, AddZeroClass α

@[to_additive AddMonoid₃]
class Monoid₃ (α : Type) extends Semigroup₃ α, MulOneClass α

attribute [to_additive existing] Monoid₃.toMulOneClass

export Semigroup₃ (mul_assoc₃)
export AddSemigroup₃ (add_assoc₃)

whatsnew in
@[to_additive]
lemma left_inv_eq_right_inv' {M : Type} [Monoid₃ M] {a b c : M} (hba : b * a = 1)␣
↪(hac : a * c = 1) : b = c := by
  rw [← one_mul c, ← hba, mul_assoc₃, hac, mul_one b]

#check left_neg_eq_right_neg'
```

借助这项技术，我们可以轻松定义交换半群、幺半群和群，然后定义环。

```
class AddCommSemigroup₃ (α : Type) extends AddSemigroup₃ α where
  add_comm : ∀ a b : α, a + b = b + a

@[to_additive AddCommSemigroup₃]
class CommSemigroup₃ (α : Type) extends Semigroup₃ α where
  mul_comm : ∀ a b : α, a * b = b * a
```

(continues on next page)

(continued from previous page)

```
class AddCommMonoid₃ (α : Type) extends AddMonoid₃ α, AddCommSemigroup₃ α

@[to_additive AddCommMonoid₃]
class CommMonoid₃ (α : Type) extends Monoid₃ α, CommSemigroup₃ α

class AddGroup₃ (G : Type) extends AddMonoid₃ G, Neg G where
  neg_add : ∀ a : G, -a + a = 0

@[to_additive AddGroup₃]
class Group₃ (G : Type) extends Monoid₃ G, Inv G where
  inv_mul : ∀ a : G, a⁻¹ * a = 1
```

我们应当在适当的时候用 simp 标记引理。

```
attribute [simp] Group₃.inv_mul AddGroup₃.neg_add
```

然后我们需要重复一些工作，因为我们切换到标准记法，但至少 to_additive 完成了从乘法记法到加法记法的转换工作。

```
@[to_additive]
lemma inv_eq_of_mul [Group₃ G] {a b : G} (h : a * b = 1) : a⁻¹ = b :=
  sorry
```

请注意，可以要求 to_additive 为一个引理添加 simp 标签，并将该属性传播到加法版本，如下所示。

```
@[to_additive (attr := simp)]
lemma Group₃.mul_inv {G : Type} [Group₃ G] {a : G} : a * a⁻¹ = 1 := by
  sorry

@[to_additive]
lemma mul_left_cancel₃ {G : Type} [Group₃ G] {a b c : G} (h : a * b = a * c) : b = c␣
↪:= by
  sorry

@[to_additive]
lemma mul_right_cancel₃ {G : Type} [Group₃ G] {a b c : G} (h : b*a = c*a) : b = c :=␣
↪by
  sorry

class AddCommGroup₃ (G : Type) extends AddGroup₃ G, AddCommMonoid₃ G
```

(continues on next page)

(continued from previous page)

```
@[to_additive AddCommGroup₃]
class CommGroup₃ (G : Type) extends Group₃ G, CommMonoid₃ G
```

现在我们准备讨论环。为了演示目的，我们不会假设加法是交换的，然后立即提供一个 `AddCommGroup₃` 的实例。Mathlib 不会玩这种把戏，首先是因为在实践中这不会使任何环实例变得更容易，其次是因为 Mathlib 的代数层次结构通过半环进行，半环类似于环但没有相反数，因此下面的证明对它们不起作用。在这里，我们不仅收获了一个不错的实践机会（尤其是对于初次接触者而言），更能通过这一过程掌握一个实例构建的范例，其中运用了能够同时指定父结构及额外字段的语法技巧。

```
class Ring₃ (R : Type) extends AddGroup₃ R, Monoid₃ R, MulZeroClass R where
  /-- Multiplication is left distributive over addition -/
  left_distrib : ∀ a b c : R, a * (b + c) = a * b + a * c
  /-- Multiplication is right distributive over addition -/
  right_distrib : ∀ a b c : R, (a + b) * c = a * c + b * c

instance {R : Type} [Ring₃ R] : AddCommGroup₃ R :=
{ Ring₃.toAddGroup₃ with
  add_comm := by
    sorry }
```

当然，我们也可以构建具体的实例，比如在整数上构建环结构（当然下面的实例利用了 Mathlib 中已经完成的所有工作）。

```
instance : Ring₃ ℤ where
  add := (· + ·)
  add_assoc₃ := add_assoc
  zero := 0
  zero_add := by simp
  add_zero := by simp
  neg := (- ·)
  neg_add := by simp
  mul := (· * ·)
  mul_assoc₃ := mul_assoc
  one := 1
  one_mul := by simp
  mul_one := by simp
  zero_mul := by simp
  mul_zero := by simp
  left_distrib := Int.mul_add
  right_distrib := Int.add_mul
```

作为练习，您现在可以为序关系设置一个简单的层次结构，包括一个有序交换幺半群的类，它同时具有偏序关系和交换幺半群结构，满足 `∀ a b : α, a ≤ b → ∀ c : α, c * a ≤ c * b`。当然，您需要为以下类添加字段，可能还需要添加 `extends` 子句。

```
class LE₁ (α : Type) where
  /-- The Less-or-Equal relation. -/
  le : α → α → Prop

@[inherit_doc] infix:50 " ≤₁ " => LE₁.le

class Preorder₁ (α : Type)

class PartialOrder₁ (α : Type)

class OrderedCommMonoid₁ (α : Type)

instance : OrderedCommMonoid₁ ℕ where
```

我们现在要讨论涉及多个类型的代数结构。最典型的例子是环上的模。如果您不知道什么是模，您可以假装它指的是向量空间，并认为我们所有的环都是域。这些结构是配备了某个环的元素的标量乘法的交换加法群。

我们首先定义由某种类型 `α` 在某种类型 `β` 上进行标量乘法的数据承载类型类，并为其赋予右结合的表示法。

```
class SMul₃ (α : Type) (β : Type) where
  /-- Scalar multiplication -/
  smul : α → β → β

infixr:73 " • " => SMul₃.smul
```

然后我们可以定义模（如果您不知道什么是模，可以先想想向量空间）。

```
class Module₁ (R : Type) [Ring₃ R] (M : Type) [AddCommGroup₃ M] extends SMul₃ R M␣
↪where
  zero_smul : ∀ m : M, (0 : R) • m = 0
  one_smul : ∀ m : M, (1 : R) • m = m
  mul_smul : ∀ (a b : R) (m : M), (a * b) • m = a • b • m
  add_smul : ∀ (a b : R) (m : M), (a + b) • m = a • m + b • m
  smul_add : ∀ (a : R) (m n : M), a • (m + n) = a • m + a • n
```

这里有一些有趣的事情。虽然 `R` 上的环结构作为此定义的参数并不令人意外，但您可能期望 `AddCommGroup₃ M` 像 `SMul₃ R M` 一样成为 `extends` 子句的一部分。尝试这样做会导致一个听起来很神秘的错误消息：`cannot find synthesization order for instance Module₁.toAddCommGroup₃ with type (R : Type) → [inst : Ring₃ R] → {M : Type} → [self : Module₁ R M] → AddCommGroup₃ M all remaining`

arguments have metavariables: Ring₃ ?R @Module₁ ?R ?inst✝ M。要理解此消息，您需要记住，这样的 extends 子句会导致一个标记为实例的字段 Module₃.toAddCommGroup₃ 。此实例具有错误消息中出现的签名：

```
(R : Type) → [inst : Ring₃ R] → {M : Type} → [self : Module₁ R M] → AddCommGroup₃ M
```

。在类型类数据库中有这样一个实例，每次 Lean 要为某个 M 查找一个 AddCommGroup₃ M 实例时，它都需要先去寻找一个完全未指定的类型 R 以及一个 Ring₃ R 实例，然后才能开始寻找主要目标，即一个 Module₁ R M 实例。这两个支线任务在错误消息中由元变量表示，并在那里用 ?R 和 ?inst✝ 标注。这样的一个 Module₃.toAddCommGroup₃ 实例对于实例解析过程来说会是一个巨大的陷阱，因此 class 命令拒绝设置它。

那么 extends SMul₃ R M 又如何呢？它创建了一个字段

```
Module₁.toSMul₃ : {R : Type} → [inst : Ring₃ R] → {M : Type} → [inst_1 : AddCommGroup₃ M] → [self : Module₁ R M] → SMul₃ R M
```

其最终结果 SMul₃ R M 同时提到了 R 和 M，所以这个字段可以安全地用作实例。规则很容易记住：在 extends 子句中出现的每个类都应提及参数中出现的每个类型。

让我们创建我们的第一个模块实例：一个环自身就是一个模块，其乘法作为标量乘法。

```
instance selfModule (R : Type) [Ring₃ R] : Module₁ R R where
  smul := fun r s ↦ r*s
  zero_smul := zero_mul
  one_smul := one_mul
  mul_smul := mul_assoc₃
  add_smul := Ring₃.right_distrib
  smul_add := Ring₃.left_distrib
```

作为第二个例子，每个阿贝尔群都是整数环上的模块（这是推广向量空间理论以允许非可逆标量的原因之一）。首先，对于任何配备零和加法的类型，都可以定义自然数的标量乘法：n • a 定义为 a + ⋯ + a，其中 a 出现 n 次。然后通过确保 (-1) • a = -a 将其扩展到整数的标量乘法。

```
def nsmul₁ [Zero M] [Add M] : ℕ → M → M
  | 0, _ => 0
  | n + 1, a => a + nsmul₁ n a

def zsmul₁ {M : Type*} [Zero M] [Add M] [Neg M] : ℤ → M → M
  | Int.ofNat n, a => nsmul₁ n a
  | Int.negSucc n, a => -nsmul₁ n.succ a
```

证明这会产生一个模块结构有点繁琐，且对当前讨论来说不那么有趣，所以我们很抱歉地略过所有公理。您 ** 无需 ** 用证明来替换这些抱歉。如果您坚持这样做，那么您可能需要陈述并证明关于 nsmul₁ 和 zsmul₁ 的几个中间引理。

```
instance abGrpModule (A : Type) [AddCommGroup₃ A] : Module₁ ℤ A where
  smul := zsmul₁
  zero_smul := sorry
  one_smul := sorry
  mul_smul := sorry
  add_smul := sorry
  smul_add := sorry
```

一个更为重要的问题是，我们目前对于整数环 ℤ 本身存在两种模块结构：首先，由于 ℤ 是阿贝尔群，因此可以定义其为 abGrpModule ℤ；其次，鉴于 ℤ 作为环的性质，我们也可以将其视作 selfModule ℤ。这两种模块结构虽然对应相同的阿贝尔群结构，但它们在标量乘法上的一致性并不显而易见。实际上，这两者确实是相同的，但这一点并非由定义直接决定，而需要通过证明来确认。这一情况对类型类实例解析过程而言无疑是个不利消息，并且可能会让使用此层次结构的用户感到颇为沮丧。当我们直接请求找到某个实例时，Lean 会自动选择一个，我们可以通过以下命令查看所选的是哪一个：

```
#synth Module₁ ℤ ℤ -- abGrpModule ℤ
```

但在更间接的上下文中，Lean 可能会先推断出一个实例，然后感到困惑。这种情况被称为坏菱形。这与我们上面使用的菱形运算无关，而是指从 ℤ 到其 Module₁ ℤ 的路径，可以通过 AddCommGroup₃ ℤ 或 Ring₃ ℤ 中的任意一个到达。

重要的是要明白，并非所有的菱形结构都是不好的。实际上，在 Mathlib 中以及本章中到处都有菱形结构。在最开始的时候我们就看到，可以从 Monoid₁ α 通过 Semigroup₁ α 或者 DiaOneClass₁ α 到达 Dia₁ α，并且由于 class 命令所做的工作，这两个 Dia₁ α 实例在定义上是相等的。特别是，底部为 Prop 值类的菱形结构不可能是不好的，因为对同一陈述的任何两个证明在定义上都是相等的。

但是我们用模块创建的菱形肯定是有问题的。问题出在 smul 字段上，它是数据而非证明，并且我们有两个构造在定义上并不相等。解决这个问题的稳健方法是确保从丰富结构到贫乏结构的转换总是通过遗忘数据来实现，而不是通过定义数据。这种众所周知的模式被称为“遗忘继承”，在 https://inria.hal.science/hal-02463336 中有大量讨论。

在我们的具体案例中，我们可以修改 AddMonoid₃ 的定义，以包含一个 nsmul 数据字段以及一些值为 Prop 类型的字段，以确保此操作确实是我们在上面构造的那个。在下面的定义中，这些字段在其类型后面使用 := 给出了默认值。由于这些默认值的存在，大多数实例的构造方式与我们之前的定义完全相同。但在 ℤ 的特殊情况下，我们将能够提供特定的值。

```
class AddMonoid₄ (M : Type) extends AddSemigroup₃ M, AddZeroClass M where
  /-- Multiplication by a natural number. -/
  nsmul : ℕ → M → M := nsmul₁
  /-- Multiplication by `(0 : ℕ)` gives `0`. -/
  nsmul_zero : ∀ x, nsmul 0 x = 0 := by intros; rfl
  /-- Multiplication by `(n + 1 : ℕ)` behaves as expected. -/
  nsmul_succ : ∀ (n : ℕ) (x), nsmul (n + 1) x = x + nsmul n x := by intros; rfl
```

(continues on next page)

(continued from previous page)

```
instance mySMul {M : Type} [AddMonoid₄ M] : SMul ℕ M := ⟨AddMonoid₄.nsmul⟩
```

让我们检查一下，即使不提供与 `nsmul` 相关的字段，我们是否仍能构造一个乘积单群实例。

```
instance (M N : Type) [AddMonoid₄ M] [AddMonoid₄ N] : AddMonoid₄ (M × N) where
  add := fun p q ↦ (p.1 + q.1, p.2 + q.2)
  add_assoc₃ := fun a b c ↦ by ext <;> apply add_assoc₃
  zero := (0, 0)
  zero_add := fun a ↦ by ext <;> apply zero_add
  add_zero := fun a ↦ by ext <;> apply add_zero
```

现在让我们处理 `ℤ` 的特殊情况，在这种情况下，我们希望使用从 `ℕ` 到 `ℤ` 的强制转换以及 `ℤ` 上的乘法来构建 `nsmul` 。请注意，特别是证明字段包含的工作比上面的默认值要多。

```
instance : AddMonoid₄ ℤ where
  add := (· + ·)
  add_assoc₃ := Int.add_assoc
  zero := 0
  zero_add := Int.zero_add
  add_zero := Int.add_zero
  nsmul := fun n m ↦ (n : ℤ) * m
  nsmul_zero := Int.zero_mul
  nsmul_succ := fun n m ↦ show (n + 1 : ℤ) * m = m + n * m
    by rw [Int.add_mul, Int.add_comm, Int.one_mul]
```

让我们检查一下我们是否解决了问题。因为 Lean 已经有一个自然数和整数的标量乘法定义，而我们希望确保使用我们的实例，所以我们不会使用 `•` 符号，而是调用 `SMul.mul` 并明确提供上面定义的实例。

```
example (n : ℕ) (m : ℤ) : SMul.smul (self := mySMul) n m = n * m := rfl
```

接下来的故事中，会在群的定义中加入一个 `zsmul` 字段，并采用类似的技巧。现在您已经准备好阅读 Mathlib 中关于幺半群、群、环和模的定义了。它们比我们在这里看到的要复杂，因为它们属于一个庞大的层级结构，但上面已经解释了所有原则。

作为练习，您可以回到上面构建的序关系层次结构，并尝试引入一个携带“小于”符号 $<_I$ 的类型类 LT_I，并确保每个预序都带有一个 $<_I$，其默认值由 $\leq_I$ 构建，并且有一个 *Prop* 值字段，断言这两个比较运算符之间的自然关系。-/

7.2 态射

到目前为止，在本章我们讨论了如何创建数学结构的层次结构。但定义结构的工作尚未完成，除非我们有了态射。这里主要有两种方法。最显而易见的方法是在函数上定义一个谓词。

```
def isMonoidHom₁ [Monoid G] [Monoid H] (f : G → H) : Prop :=
  f 1 = 1 ∧ ∀ g g', f (g * g') = f g * f g'
```

在这个定义中，使用合取有点不自然。特别是用户在想要访问这两个条件时，需要记住我们选择的顺序。所以我们可以使用一个结构来替代。

```
structure isMonoidHom₂ [Monoid G] [Monoid H] (f : G → H) : Prop where
  map_one : f 1 = 1
  map_mul : ∀ g g', f (g * g') = f g * f g'
```

一旦到了这里，不禁让人考虑将其设计为一个类，并使用类型类实例解析过程自动推断复杂函数的 `isMonoidHom₂` ，基于简单函数的实例。例如，两个幺半群态射的复合也是一个幺半群态射，这似乎是一个有用的实例。然而，这样的实例对于解析过程来说会非常棘手，因为它需要在每个地方寻找 `g ∘ f` 。在 `g (f x)` 中找不到它会非常令人沮丧。更一般地说，必须始终牢记，识别给定表达式中应用了哪个函数是一个非常困难的问题，称为“高阶合一问题”。因此，Mathlib 并不采用这种类的方法。

一个更根本的问题在于，我们是像上面那样使用谓词（无论是使用 `def` 还是 `structure` ），还是使用将函数和谓词捆绑在一起的结构。这在一定程度上是一个心理问题。考虑两个幺半群之间的函数，而该函数不是同态的情况极为罕见。感觉“幺半群同态”不是一个可以赋予裸函数的形容词，而是一个名词。另一方面，有人可能会说，拓扑空间之间的连续函数实际上是一个恰好连续的函数。这就是为什么 Mathlib 有一个 `Continuous` 谓词的原因。例如，您可以这样写：

```
example : Continuous (id : ℝ → ℝ) := continuous_id
```

我们仍然有连续函数的束，这在例如为连续函数空间赋予拓扑时很方便，但它们不是处理连续性的主要工具。

相比之下，单子（或其他代数结构）之间的态射是捆绑在一起的，例如：

```
@[ext]
structure MonoidHom₁ (G H : Type) [Monoid G] [Monoid H]  where
  toFun : G → H
  map_one : toFun 1 = 1
  map_mul : ∀ g g', toFun (g * g') = toFun g * toFun g'
```

当然，我们不想到处都输入 `toFun` ，所以我们使用 `CoeFun` 类型类来注册一个强制转换。它的第一个参数是我们想要强制转换为函数的类型。第二个参数描述目标函数类型。在我们的例子中，对于每个 `f : MonoidHom₁ G H` ，它总是 `G → H` 。我们还用 `coe` 属性标记 `MonoidHom₁.toFun` ，以确保它在策略状态中几乎不可见，仅通过 `↑` 前缀显示。

```
instance [Monoid G] [Monoid H] : CoeFun (MonoidHom₁ G H) (fun _ ↦ G → H) where
  coe := MonoidHom₁.toFun

attribute [coe] MonoidHom₁.toFun
```

让我们检查一下我们是否确实可以将捆绑的单子态射应用于一个元素。

```
example [Monoid G] [Monoid H] (f : MonoidHom₁ G H) : f 1 = 1 :=  f.map_one
```

于其他类型的态射，我们也可以做同样的事情，直到我们遇到环态射。

```
@[ext]
structure AddMonoidHom₁ (G H : Type) [AddMonoid G] [AddMonoid H]  where
  toFun : G → H
  map_zero : toFun 0 = 0
  map_add : ∀ g g', toFun (g + g') = toFun g + toFun g'

instance [AddMonoid G] [AddMonoid H] : CoeFun (AddMonoidHom₁ G H) (fun _ ↦ G → H)␣
↪where
  coe := AddMonoidHom₁.toFun

attribute [coe] AddMonoidHom₁.toFun

@[ext]
structure RingHom₁ (R S : Type) [Ring R] [Ring S] extends MonoidHom₁ R S,␣
↪AddMonoidHom₁ R S
```

这种方法存在一些问题。一个小问题是，我们不太清楚在哪里放置 `coe` 属性，因为 `RingHom₁.toFun` 并不存在，相关函数是 `MonoidHom₁.toFun ∘ RingHom₁.toMonoidHom₁`，这并不是一个可以标记属性的声明（但我们仍然可以定义一个 `CoeFun (RingHom₁ R S) (fun _ ↦ R → S)` 实例）。一个更重要的问题是，关于单子态射的引理不能直接应用。这将归结为要么每次应用单群同态引理时都与 `RingHom₁.toMonoidHom₁` 打交道，要么为环同态重新陈述每个这样的引理。这两种选择都不吸引人，因此 Mathlib 在这里使用了一个新的层次结构技巧。其想法是定义一个至少为单群同态的对象的类型类，用单群同态和环同态实例化该类，并使用它来陈述每个引理。在下面的定义中，`F` 可以是 `MonoidHom₁ M N`，或者如果 `M` 和 `N` 具有环结构，则为 `RingHom₁ M N`。

```
class MonoidHomClass₁ (F : Type) (M N : Type) [Monoid M] [Monoid N] where
  toFun : F → M → N
  map_one : ∀ f : F, toFun f 1 = 1
  map_mul : ∀ f g g', toFun f (g * g') = toFun f g * toFun f g'
```

然而，上述实现存在一个问题。我们尚未注册到函数实例的强制转换。让我们现在尝试这样做。

```
def badInst [Monoid M] [Monoid N] [MonoidHomClass₁ F M N] : CoeFun F (fun _ ↦ M → N)␣
↪where
  coe := MonoidHomClass₁.toFun
```

将此设为实例是不好的。当面对类似 `f x` 的情况，而 `f` 的类型不是函数类型时，Lean 会尝试查找一个 `CoeFun` 实例来将 `f` 转换为函数。上述函数的类型为：

```
{M N F : Type} → [Monoid M] → [Monoid N] → [MonoidHomClass₁ F M N] → CoeFun F (fun
x ↦ M → N)
```

因此，在尝试应用它时，Lean 无法预先确定未知类型 `M`、`N` 和 `F` 应该以何种顺序进行推断。这与我们之前看到的情况略有不同，但归根结底是同一个问题：在不知道 `M` 的情况下，Lean 将不得不在未知类型上搜索单子实例，从而无望地尝试数据库中的每一个单子实例。如果您想看看这种实例的效果，可以在上述声明的顶部输入 `set_option synthInstance.checkSynthOrder false in`，将 `def badInst` 替换为 `instance`，然后在这个文件中查找随机出现的错误。

在这里，解决方案很简单，我们需要告诉 Lean 先查找什么是 `F`，然后再推导出 `M` 和 `N`。这可以通过使用 `outParam` 函数来实现。该函数被定义为恒等函数，但仍会被类型类机制识别，并触发所需的行为。因此，我们可以重新定义我们的类，注意使用 `outParam` 函数：

```
class MonoidHomClass₂ (F : Type) (M N : outParam Type) [Monoid M] [Monoid N] where
  toFun : F → M → N
  map_one : ∀ f : F, toFun f 1 = 1
  map_mul : ∀ f g g', toFun f (g * g') = toFun f g * toFun f g'

instance [Monoid M] [Monoid N] [MonoidHomClass₂ F M N] : CoeFun F (fun _ ↦ M → N)␣
↪where
  coe := MonoidHomClass₂.toFun

attribute [coe] MonoidHomClass₂.toFun
```

现在我们可以继续执行我们的计划，实例化这个类了。

```
instance (M N : Type) [Monoid M] [Monoid N] : MonoidHomClass₂ (MonoidHom₁ M N) M N␣
↪where
  toFun := MonoidHom₁.toFun
  map_one := fun f ↦ f.map_one
  map_mul := fun f ↦ f.map_mul

instance (R S : Type) [Ring R] [Ring S] : MonoidHomClass₂ (RingHom₁ R S) R S where
  toFun := fun f ↦ f.toMonoidHom₁.toFun
  map_one := fun f ↦ f.toMonoidHom₁.map_one
  map_mul := fun f ↦ f.toMonoidHom₁.map_mul
```

正如所承诺的那样，我们对 `f : F` 在假设存在 `MonoidHomClass₁ F` 的实例的情况下所证明的每个引理，都

将同时适用于幺半群同态和环同态。让我们来看一个示例引理，并检查它是否适用于这两种情况。

```
lemma map_inv_of_inv [Monoid M] [Monoid N] [MonoidHomClass₂ F M N] (f : F) {m m' : M}␣
↪(h : m*m' = 1) :
    f m * f m' = 1 := by
  rw [← MonoidHomClass₂.map_mul, h, MonoidHomClass₂.map_one]

example [Monoid M] [Monoid N] (f : MonoidHom₁ M N) {m m' : M} (h : m*m' = 1) : f m *␣
↪f m' = 1 :=
map_inv_of_inv f h

example [Ring R] [Ring S] (f : RingHom₁ R S) {r r' : R} (h : r*r' = 1) : f r * f r' =␣
↪1 :=
map_inv_of_inv f h
```

乍一看，可能会觉得我们又回到了之前那个糟糕的想法，即把 `MonoidHom₁` 设为一个类。但其实并非如此。一切都提升了一个抽象层次。类型类解析过程不会寻找函数，而是寻找 `MonoidHom₁` 或者 `RingHom₁` 。

我们方法中仍存在的一个问题是在围绕 `toFun` 字段以及相应的 `CoeFun` 实例和 `coe` 属性的地方存在重复代码。另外，最好记录下这种模式仅用于具有额外属性的函数，这意味着对函数的强制转换应该是单射的。因此，Mathlib 在这个基础层之上又增加了一层抽象，即基础类 `DFunLike` （其中“DFun”代表依赖函数）。让我们基于这个基础层重新定义我们的 `MonoidHomClass` 。

```
class MonoidHomClass₃ (F : Type) (M N : outParam Type) [Monoid M] [Monoid N] extends
    DFunLike F M (fun _ ↦ N) where
  map_one : ∀ f : F, f 1 = 1
  map_mul : ∀ (f : F) g g', f (g * g') = f g * f g'

instance (M N : Type) [Monoid M] [Monoid N] : MonoidHomClass₃ (MonoidHom₁ M N) M N␣
↪where
  coe := MonoidHom₁.toFun
  coe_injective' _ _ := MonoidHom₁.ext
  map_one := MonoidHom₁.map_one
  map_mul := MonoidHom₁.map_mul
```

当然，态射的层次结构不止于此。我们还可以继续定义一个类 `RingHomClass₃` 来扩展 `MonoidHomClass₃` ，然后将其实例化为 `RingHom` ，之后再实例化为 `AlgebraHom` （代数是具有某些额外结构的环）。不过，我们已经涵盖了 Mathlib 中用于态射的主要形式化思想，您应该已经准备好理解 Mathlib 中态射的定义方式了。作为练习，您应当尝试定义有序类型之间的保序函数类，以及保序幺半群同态。这仅用于训练目的。与连续函数类似，在 Mathlib 中保序函数主要是未打包的，它们通过 `Monotone` 预言来定义。当然，您需要完成下面的类定义。

```
@[ext]
structure OrderPresHom (α β : Type) [LE α] [LE β] where
  toFun : α → β
  le_of_le : ∀ a a', a ≤ a' → toFun a ≤ toFun a'

@[ext]
structure OrderPresMonoidHom (M N : Type) [Monoid M] [LE M] [Monoid N] [LE N] extends
MonoidHom₁ M N, OrderPresHom M N

class OrderPresHomClass (F : Type) (α β : outParam Type) [LE α] [LE β]

instance (α β : Type) [LE α] [LE β] : OrderPresHomClass (OrderPresHom α β) α β where

instance (α β : Type) [LE α] [Monoid α] [LE β] [Monoid β] :
    OrderPresHomClass (OrderPresMonoidHom α β) α β where

instance (α β : Type) [LE α] [Monoid α] [LE β] [Monoid β] :
    MonoidHomClass₃ (OrderPresMonoidHom α β) α β
  := sorry
```

7.3 子对象

定义了一些代数结构及其态射之后，下一步是考虑继承这种代数结构的集合，例如子群或子环。这在很大程度上与我们之前的话题重叠。实际上，`X` 中的一个集合是作为从 `X` 到 `Prop` 的函数来实现的，因此子对象是满足特定谓词的函数。因此，我们可以重用许多导致 `DFunLike` 类及其后代的想法。我们不会重用 `DFunLike` 本身，因为这会破坏从 `Set X` 到 `X → Prop` 的抽象屏障。取而代之的是有一个 `SetLike` 类。该类不是将一个嵌入包装到函数类型中，而是将其包装到 `Set` 类型中，并定义相应的强制转换和 `Membership` 实例。

```
@[ext]
structure Submonoid₁ (M : Type) [Monoid M] where
  /-- 子幺半群的载体。 -/
  carrier : Set M
  /-- 子幺半群中两个元素的乘积属于该子幺半群。 -/
  mul_mem {a b} : a ∈ carrier → b ∈ carrier → a * b ∈ carrier
  /-- 单位元素属于子幺半群。 -/
  one_mem : 1 ∈ carrier

/-- `M` 中的子幺半群可以被视为 `M` 中的集合 -/
instance [Monoid M] : SetLike (Submonoid₁ M) M where
  coe := Submonoid₁.carrier
```

(continues on next page)

(continued from previous page)

```
  coe_injective' _ _ := Submonoid₁.ext
```

借助上述的 `SetLike` 实例，我们已经可以很自然地表述子幺半群 `N` 包含 `1` ，而无需使用 `N.carrier` 。我们还可以将 `N` 作为 `M` 中的一个集合来处理，或者在映射下取其直接像。

```
example [Monoid M] (N : Submonoid₁ M) : 1 ∈ N := N.one_mem

example [Monoid M] (N : Submonoid₁ M) (α : Type) (f : M → α) := f '' N
```

我们还有一个到 `Type` 的强制转换，它使用 `Subtype` ，因此，给定一个子幺半群 `N` ，我们可以写一个参数 `(x : N)` ，其可以被强制转换为属于 `N` 的 `M` 的一个元素。

```
example [Monoid M] (N : Submonoid₁ M) (x : N) : (x : M) ∈ N := x.property
```

利用这种到 `Type` 的强制转换，我们还可以解决为子幺半群配备幺半群结构的任务。我们将使用上述与 `N` 相关联的类型的强制转换，以及断言这种强制转换是单射的引理 `SetCoe.ext` 。这两者均由 `SetLike` 实例提供。

```
instance SubMonoid₁Monoid [Monoid M] (N : Submonoid₁ M) : Monoid N where
  mul := fun x y ↦ ⟨x*y, N.mul_mem x.property y.property⟩
  mul_assoc := fun x y z ↦ SetCoe.ext (mul_assoc (x : M) y z)
  one := ⟨1, N.one_mem⟩
  one_mul := fun x ↦ SetCoe.ext (one_mul (x : M))
  mul_one := fun x ↦ SetCoe.ext (mul_one (x : M))
```

请注意，在上述实例中，我们本可以使用解构绑定器，而不是使用到 `M` 的强制转换并调用 `property` 字段，如下所示。

```
example [Monoid M] (N : Submonoid₁ M) : Monoid N where
  mul := fun ⟨x, hx⟩ ⟨y, hy⟩ ↦ ⟨x*y, N.mul_mem hx hy⟩
  mul_assoc := fun ⟨x, _⟩ ⟨y, _⟩ ⟨z, _⟩ ↦ SetCoe.ext (mul_assoc x y z)
  one := ⟨1, N.one_mem⟩
  one_mul := fun ⟨x, _⟩ ↦ SetCoe.ext (one_mul x)
  mul_one := fun ⟨x, _⟩ ↦ SetCoe.ext (mul_one x)
```

为了将关于子幺半群的引理应用于子群或子环，我们需要一个类，就像对于同态一样。请注意，这个类将一个 `SetLike` 实例作为参数，因此它不需要载体字段，并且可以在其字段中使用成员符号。

```
class SubmonoidClass₁ (S : Type) (M : Type) [Monoid M] [SetLike S M] : Prop where
  mul_mem : ∀ (s : S) {a b : M}, a ∈ s → b ∈ s → a * b ∈ s
  one_mem : ∀ s : S, 1 ∈ s

instance [Monoid M] : SubmonoidClass₁ (Submonoid₁ M) M where
```

(continues on next page)

(continued from previous page)

```
  mul_mem := Submonoid₁.mul_mem
  one_mem := Submonoid₁.one_mem
```

作为练习，您应该定义一个 `Subgroup₁` 结构，为其赋予一个 `SetLike` 实例和一个 `SubmonoidClass₁` 实例，在与 `Subgroup₁` 相关联的子类型上放置一个 `Group` 实例，并定义一个 `SubgroupClass₁` 类。

在 Mathlib 中，关于给定代数对象的子对象，还有一点非常重要，那就是它们总是构成一个完备格，这种结构被大量使用。例如，您可能会寻找一个关于子幺半群交集仍是子幺半群的引理。但这不会是一个引理，而是一个下确界构造。我们来看两个子幺半群的情况。

```
instance [Monoid M] : Inf (Submonoid₁ M) :=
  ⟨fun S₁ S₂ ↦
    { carrier := S₁ ∩ S₂
      one_mem := ⟨S₁.one_mem, S₂.one_mem⟩
      mul_mem := fun ⟨hx, hx'⟩ ⟨hy, hy'⟩ ↦ ⟨S₁.mul_mem hx hy, S₂.mul_mem hx' hy'⟩ }⟩
```

这使得两个子幺半群的交集能够作为一个子幺半群得到。

```
example [Monoid M] (N P : Submonoid₁ M) : Submonoid₁ M := N ⊓ P
```

您可能会觉得在上面的例子中使用下确界符号 `⊓` 而不是交集符号 `∩` 很遗憾。但想想上确界。两个子幺半群的并集不是子幺半群。然而，子幺半群仍然构成一个格（甚至是完备格）。实际上，`N ⊔ P` 是由 `N` 和 `P` 的并集生成的子幺半群，当然用 `N ∪ P` 来表示会非常令人困惑。所以您可以看到使用 `N ⊓ P` 要一致得多。在各种代数结构中，这种表示法也更加一致。一开始看到两个子空间 `E` 和 `F` 的和用 `E ⊔ F` 而不是 `E + F` 表示可能会觉得有点奇怪。但您会习惯的。很快你就会觉得 `E + F` 这种表示法是一种干扰，它强调的是一个偶然的事实，即 `E ⊔ F` 的元素可以写成 `E` 的一个元素与 `F` 的一个元素之和，而不是强调 `E ⊔ F` 是包含 `E` 和 `F` 的最小向量子空间这一根本事实。

本章的最后一个主题是商的概念。同样，我们想要解释在 Mathlib 中如何构建方便的符号表示法以及如何避免代码重复。这里的主要手段是 `HasQuotient` 类，它允许使用诸如 `M ⧸ N` 这样的符号表示法。请注意，商符号 `⧸` 是一个特殊的 Unicode 字符，而不是普通的 ASCII 除法符号。

例如，我们将通过一个子幺半群来构建一个交换幺半群的商，证明留给你。在最后一个示例中，您可以使用 `Setoid.refl` ，但它不会自动获取相关的 `Setoid` 结构。您可以使用 `@` 语法提供所有参数来解决此问题，例如 `@Setoid.refl M N.Setoid` 。

```
def Submonoid.Setoid [CommMonoid M] (N : Submonoid M) : Setoid M  where
  r := fun x y ↦ ∃ w ∈ N, ∃ z ∈ N, x*w = y*z
  iseqv := {
    refl := fun x ↦ ⟨1, N.one_mem, 1, N.one_mem, rfl⟩
    symm := fun ⟨w, hw, z, hz, h⟩ ↦ ⟨z, hz, w, hw, h.symm⟩
    trans := by
      sorry
```

(continues on next page)

(continued from previous page)

```
  }

instance [CommMonoid M] : HasQuotient M (Submonoid M) where
  quotient' := fun N ↦ Quotient N.Setoid

def QuotientMonoid.mk [CommMonoid M] (N : Submonoid M) : M → M ⧸ N := Quotient.mk N.
↪Setoid

instance [CommMonoid M] (N : Submonoid M) : Monoid (M ⧸ N) where
  mul := Quotient.map₂' ( · * · ) (by
      sorry
        )
  mul_assoc := by
      sorry
  one := QuotientMonoid.mk N 1
  one_mul := by
      sorry
  mul_one := by
      sorry
```

CHAPTER

EIGHT

群与环

在 Section 2.2 中，我们已经学习了如何推导群和环中的运算规则。而在稍后的 Section 6.2 中，我们探讨了定义抽象代数结构的方法，例如群的结构，以及像高斯整环这样的具体实例。Chapter 7 详细说明了 Mathlib 是如何处理这些抽象结构的层级关系的。

本章将更深入地探讨群和环的相关内容。由于 Mathlib 仍在不断发展，我们无法全面覆盖所有内容，但会提供使用相关库的切入点，并展示如何运用其核心概念。虽然本章的部分内容与 Chapter 7 有所重叠，但我们的重点在于指导如何使用 Mathlib，而非分析其设计背后的理念。因此，为了更好地理解本章的部分示例，可能需要回顾 Chapter 7。

8.1 幺半群与群

8.1.1 幺半群和同态

抽象代数课程通常从群开始，逐步深入到环、域直至向量空间。这种教学顺序常会导致讨论环上的乘法等非群结构运算时出现不必要的曲折：许多群定理证明的方法同样适用，但我们却要重新证明。一般来说，当你用书本学习数学时，多数作者用一行“以下留作习题”化解此窘境。不过，在 Lean 中，我们有另一种虽然不那么方便，但更安全，更适于形式化的方法：引入幺半群 (monoid)。

类型 M 上的 **幺半群**是一个内部具有结合律和单位元的复合法则 (composition law)。幺半群引入的初衷是同时描述群和环的乘法结构。一些自然的例子如：自然数与加法构成一个幺半群。

从实用的角度而言，你几乎可以忘记 Mathlib 中的幺半群。但了解其存在是有益的，否则在寻找那些实际上并不需要元素可逆的引理时，你可能忽略它们在幺半群而非群的文件中。

类型 `M` 上的幺半群被写作 `Monoid M`. 函数 `Monoid` 是一个类型类，所以通常作为隐式实例参数而出现。(即在方括号中) `Monoid` 默认用乘号作运算符号。如需使用加号，可以使用 `AddMonoid`。若需满足交换律，可使用 `CommMonoid`.

```
example {M : Type*} [Monoid M] (x : M) : x * 1 = x := mul_one x

example {M : Type*} [AddCommMonoid M] (x y : M) : x + y = y + x := add_comm x y
```

注意：虽然库中确实定义了 `AddMonoid`，但将加号用于非交换的运算可能会导致混淆。

幺半群 `M` 与 `N` 间的同态的类型称为 `MonoidHom M N`，可写作 `M →* N`. 在将一个同态作用于类型 `M` 的元素时，Lean 将视其为一个由 `M` 到 `N` 的函数。相应的加法版本被称为 `AddMonoidHom`, 写作 `M →+ N`. `M →* N`.

```
example {M N : Type*} [Monoid M] [Monoid N] (x y : M) (f : M →* N) : f (x * y) = f x␣
↪* f y :=
  f.map_mul x y

example {M N : Type*} [AddMonoid M] [AddMonoid N] (f : M →+ N) : f 0 = 0 :=
  f.map_zero
```

同态其实是一系列映射，即：同态映射本身和它的一些性质。Section 7.2 中对这样的系列映射有解释。现在，我们发现这也产生了些不妙的效果：我们无法使用常规的函数复合来复合两个映射。对此，有 `MonoidHom.comp` 和 `AddMonoidHom.comp`.

```
example {M N P : Type*} [AddMonoid M] [AddMonoid N] [AddMonoid P]
    (f : M →+ N) (g : N →+ P) : M →+ P := g.comp f
```

8.1.2 群和同态

对于群，我们有更多可以探讨的。群，就是幺半群加上每一个元素都有逆元的性质。

```
example {G : Type*} [Group G] (x : G) : x * x⁻¹ = 1 := mul_inv_cancel x
```

正如之前我们看到的 `ring` 策略，我们有 `group` 策略用来证明所有群所共同满足的恒等式。(即自由群所满足的恒等式)

```
example {G : Type*} [Group G] (x y z : G) : x * (y * z) * (x * z)⁻¹ * (x * y * x⁻¹)⁻¹␣
↪= 1 := by
  group
```

对满足交换律的群，还有 `abel` 策略.

```
example {G : Type*} [AddCommGroup G] (x y z : G) : z + x + (y - z - x) = y := by
  abel
```

有趣的是，群同态所需满足的实际上与幺半群别无二致。所以我们之前的例子可以照搬过来。

```
example {G H : Type*} [Group G] [Group H] (x y : G) (f : G →* H) : f (x * y) = f x *␣
↪f y :=
  f.map_mul x y
```

当然也有点新性质：

```
example {G H : Type*} [Group G] [Group H] (x : G) (f : G →* H) : f (x⁻¹) = (f x)⁻¹ :=
  f.map_inv x
```

你也许会担心构造一个群同态需要枉费些不必要的工夫：幺半群同态的定义需要映射保持幺元，可这是群的情况下由第一条保持运算的性质就能自动得到的。虽然在实际中多做这一步并不困难，但我们可以避免它。接下来的函数可以由保持运算的群间映射给出群同态.

```
example {G H : Type*} [Group G] [Group H] (f : G → H) (h : ∀ x y, f (x * y) = f x * f
↪y) :
    G →* H :=
  MonoidHom.mk' f h
```

同样对于群（幺半群）同构，我们有类型 `MulEquiv`，记为 `≃*` (对应加号版本 `AddEquiv` 记为 `≃+`).

> `f : G ≃* H` 的逆是 `MulEquiv.symm f : H ≃* G`, `f` 和 `g` 的复合是 `MulEquiv.trans f g`, `G` 到自身的恒等同构 `MulEquiv.refl G`.

使用匿名投影子记号, 前两个可对应写作 `f.symm` 和 `f.trans g`. 这些类型的元素将在必要时自动转换为同态或函数.

```
example {G H : Type*} [Group G] [Group H] (f : G ≃* H) :
    f.trans f.symm = MulEquiv.refl G :=
  f.self_trans_symm
```

你可以用 `MulEquiv.ofBijective` 从一个也是双射的同态构造同构. 同时这样做会使逆映射被标记为不可计算的 (noncomputable).

```
noncomputable example {G H : Type*} [Group G] [Group H]
    (f : G →* H) (h : Function.Bijective f) :
    G ≃* H :=
  MulEquiv.ofBijective f h
```

8.1.3 子群

就像群同态是一系列映射一样，`G` 的子群也是一个由类型 `G` 的集合和相应的封闭性质所共同构成结构。

```
example {G : Type*} [Group G] (H : Subgroup G) {x y : G} (hx : x ∈ H) (hy : y ∈ H) :
    x * y ∈ H :=
  H.mul_mem hx hy

example {G : Type*} [Group G] (H : Subgroup G) {x : G} (hx : x ∈ H) :
    x⁻¹ ∈ H :=
  H.inv_mem hx
```

在以上的例子中, 重要的一点是要理解 `Subgroup G` 是 `G` 的子群的类型，而不是对一个 `Set G` 中元素 `H` 的附加的断言 `IsSubgroup H`. `Subgroup G` 类型已经被赋予了到 `Set G` 的类型转换和一个与 `G` 间的包含关系的判断。参见 Section 7.3 以了解这是为何要以及如何完成的。

当然，两个子群相同当且仅当它们包含的元素完全相同。这一事实被注册到了 `ext` 策略, 所以你可以像证明两个集合相等一样来证明两个子群相等。

当我们论证类似 `ℤ` 是 `ℚ` 的一个加性子群这样的命题时, 我们真正想要的其实相当于是构造一个 `AddSubgroup ℚ` 类型的项，该项到 `Set ℚ` 的投影为 `ℤ`，或者更精确的说，`ℤ` 在 `ℚ` 中的像.

```
example : AddSubgroup ℚ where
  carrier := Set.range ((↑) : ℤ → ℚ)
  add_mem' := by
    rintro _ _ ⟨n, rfl⟩ ⟨m, rfl⟩
    use n + m
    simp
  zero_mem' := by
    use 0
    simp
  neg_mem' := by
    rintro _ ⟨n, rfl⟩
    use -n
    simp
```

通过使用类型类，Mathlib 知道群的子群继承了群结构。

```
example {G : Type*} [Group G] (H : Subgroup G) : Group H := inferInstance
```

这一个例子隐含了一些信息. 对象 `H` 并不是一个类型, 但 Lean 通过将其解释为 `G` 的子类型自动将其转换为了一个类型. 以上例子还可以用更清晰的方式来表述:

```
example {G : Type*} [Group G] (H : Subgroup G) : Group {x : G // x ∈ H} :=␣
↪inferInstance
```

使用类型 ``Subgroup G`` 而不是断言 `IsSubgroup : Set G → Prop` 的一个重要优势在于可以为 `Subgroup G` 轻松地赋予额外的结构. 重要的是, 它具有对于包含关系的完备格结构 (lattice structure). 例如, 相较于用额外的引理来说明两个 `G` 的子群的交仍然是一个子群, 我们可以使用格运算符 `⊓` 直接构造出这个交集构成的子群. 我们可以将有关格的任意引理应用到这样的构造上.

现在我们来检验, 两个子群的下确界导出的集合, 从定义上来说, 确实是它们的交集.

```
example {G : Type*} [Group G] (H H' : Subgroup G) :
    ((H ⊓ H' : Subgroup G) : Set G) = (H : Set G) ∩ (H' : Set G) := rfl
```

为实际上给出了集合交集的运算使用另一个记号可能有些奇怪, 但要知道，这样的对应关系在上确界与并集

运算之间不再成立. 因为一般来说, 子群的并不再构成一个子群. 我们需要的是这样的并生成的子群, 这可以使用 `Subgroup.closure` 来得到.

```
example {G : Type*} [Group G] (H H' : Subgroup G) :
    ((H ⊔ H' : Subgroup G) : Set G) = Subgroup.closure ((H : Set G) ∪ (H' : Set G)) :=
  rw [Subgroup.sup_eq_closure]
```

另一个微妙的地方在于 `G` 本身并不具有类型 `Subgroup G`, 所以我们需要一种方式来将 `G` 视作它自身的子群. 这同样由格结构来证明: 全集构成的子群是格中的最大元.

```
example {G : Type*} [Group G] (x : G) : x ∈ (⊤ : Subgroup G) := trivial
```

类似的，格中的最小元是只包含有单位元的子群.

```
example {G : Type*} [Group G] (x : G) : x ∈ (⊥ : Subgroup G) ↔ x = 1 := Subgroup.mem_bot
```

作为操作群与子群的练习，你可以定义一个子群与环境群中的元素得到的共轭子群.

```
def conjugate {G : Type*} [Group G] (x : G) (H : Subgroup G) : Subgroup G where
  carrier := {a : G | ∃ h, h ∈ H ∧ a = x * h * x⁻¹}
  one_mem' := by
    dsimp
    sorry
  inv_mem' := by
    dsimp
    sorry
  mul_mem' := by
    dsimp
    sorry
```

将前两个主题结合在一个, 我们就可以使用群同态来 “前推” (push forward) 或 “拉回” (pull back) 子群. Mathlib 中的命名习惯是将这两个操作称为 `map` 和 `comap`. 它们并不是常见的数学名词, 但它们的优势在于较 “push-forward” 和 “direct image” 更为简洁.

```
example {G H : Type*} [Group G] [Group H] (G' : Subgroup G) (f : G →* H) : Subgroup H :=
  Subgroup.map f G'

example {G H : Type*} [Group G] [Group H] (H' : Subgroup H) (f : G →* H) : Subgroup G :=
  Subgroup.comap f H'
```

(continues on next page)

(continued from previous page)

```
#check Subgroup.mem_map
#check Subgroup.mem_comap
```

特别的, 最小子群（即只含单位元的群）在同态 `f` 下的前像构成一个被称为同态 `f` 的 **核**的子群, 而 `f` 的值域同样也是一个子群.

```
example {G H : Type*} [Group G] [Group H] (f : G →* H) (g : G) :
    g ∈ MonoidHom.ker f ↔ f g = 1 :=
  f.mem_ker

example {G H : Type*} [Group G] [Group H] (f : G →* H) (h : H) :
    h ∈ MonoidHom.range f ↔ ∃ g : G, f g = h :=
  f.mem_range
```

作为操作群同态和子群的练习, 我们来证明一些初等性质. Mathlib 已经证明了它们, 所以如果你想真正锻炼自己，最好不要依赖于 `exact?`。

```
section exercises
variable {G H : Type*} [Group G] [Group H]

open Subgroup

example (φ : G →* H) (S T : Subgroup H) (hST : S ≤ T) : comap φ S ≤ comap φ T := by
  sorry

example (φ : G →* H) (S T : Subgroup G) (hST : S ≤ T) : map φ S ≤ map φ T := by
  sorry

variable {K : Type*} [Group K]

-- 记得你可以用 `ext` 策略证明子群的相等性.
example (φ : G →* H) (ψ : H →* K) (U : Subgroup K) :
    comap (ψ.comp φ) U = comap φ (comap ψ U) := by
  sorry

-- 将一个子群先后通过两个群同态 “前推” 相当于通过这些同态的复合 “前推” .
example (φ : G →* H) (ψ : H →* K) (S : Subgroup G) :
    map (ψ.comp φ) S = map ψ (S.map φ) := by
  sorry
```

(continues on next page)

(continued from previous page)

```
end exercises
```

让我们用两个非常经典的例子来结束对 Mathlib 中子群的介绍. 拉格朗日定理 (Lagrange theorem) 说明了有限群的子群的阶可以整除该群的阶. 西罗第一定理 (Sylow’s first theorem)，是拉格朗日定理的一个著名部分逆定理.

虽然 Mathlib 的这一部分是部分为了允许计算而设置的，但我们可以通过以下的 open scoped 命令告诉 Lean 使用非构造性逻辑。

```
open scoped Classical


example {G : Type*} [Group G] (G' : Subgroup G) : Nat.card G' ∣ Nat.card G :=
  ⟨G'.index, mul_comm G'.index _ ▸ G'.index_mul_card.symm⟩

open Subgroup

example {G : Type*} [Group G] [Finite G] (p : ℕ) {n : ℕ} [Fact p.Prime]
    (hdvd : p ^ n ∣ Nat.card G) : ∃ K : Subgroup G, Nat.card K = p ^ n :=
  Sylow.exists_subgroup_card_pow_prime p hdvd
```

接下来的两个练习推导出拉格朗日引理的一个推论。(Mathlib 也已经有证明，所以也不要太快地使用 `exact?`)

```
lemma eq_bot_iff_card {G : Type*} [Group G] {H : Subgroup G} :
    H = ⊥ ↔ Nat.card H = 1 := by
  suffices (∀ x ∈ H, x = 1) ↔ ∃ x ∈ H, ∀ a ∈ H, a = x by
    simpa [eq_bot_iff_forall, Nat.card_eq_one_iff_exists]
  sorry

#check card_dvd_of_le

lemma inf_bot_of_coprime {G : Type*} [Group G] (H K : Subgroup G)
    (h : (Nat.card H).Coprime (Nat.card K)) : H ⊓ K = ⊥ := by
  sorry
```

8.1.4 具体群

在 Mathlib 中，也可以操作具体的群，尽管这通常比处理抽象理论更为复杂。例如, 对任意一个类型 `X`, `X` 的置换群为 `Equiv.Perm X`. 特别的，对称群 $\mathfrak{S}_n$ 是 `Equiv.Perm (Fin n)`. 你也可以论述关于这些群的抽象事实，比如 `Equiv.Perm X` 在 `X` 有限时是由轮换生成的.

```
open Equiv

example {X : Type*} [Finite X] : Subgroup.closure {σ : Perm X | Perm.IsCycle σ} = ⊤ :=
  Perm.closure_isCycle
```

可以完全具体化并计算轮换的实际乘积. 下面使用 `#simp` 命令, 其调用 `simp` 策略作用于表达式. 记号 `c[]` 用于表示轮换表示下的置换. 本例中是 `ℕ` 上的置换. 也可以在第一个元素上用 `(1 : Fin 5)` 这样的类型标注使其在 `Perm (Fin 5)` 上计算.

```
#simp [mul_assoc] c[1, 2, 3] * c[2, 3, 4]
```

另一种处理具体群的方式是使用自由群及其相关表示. 类型 `α` 上的自由群是一个 `FreeGroup α`, 对应包含映射为 `FreeGroup.of : α → FreeGroup α`. 例如，可以定义类型 `S`，其中有三个元素 `a`, `b` 和 `c`, 这些元素可以构成自由群中的元素 ab^{-1}.

```
section FreeGroup

inductive S | a | b | c

open S

def myElement : FreeGroup S := (.of a) * (.of b)⁻¹
```

我们给出了预期的类型，所以 Lean 能推断出 `.of` 代表着 `FreeGroup.of`.

自由群的整体性质体现在等价性 `FreeGroup.lift` 中. 例如，可以定义 `FreeGroup S` 到 `Perm (Fin 5)` 的群同态: 映射 `a` 到 `c[1, 2, 3]`, `b` 到 `c[2, 3, 1]`, `c` 到 `c[2, 3]`,

```
def myMorphism : FreeGroup S →* Perm (Fin 5) :=
  FreeGroup.lift fun | .a => c[1, 2, 3]
                     | .b => c[2, 3, 1]
                     | .c => c[2, 3]
```

最后一个例子, 让我们通过定义一个元素立方为单位来生成一个群 (也即同构于 $\mathbb{Z}/3$) 并构造从其到 `Perm (Fin 5)` 的态射.

`Unit` 类型只有一个元素，表示为 `()`. 函数 `PresentedGroup` 接受一系列关系, (即一系列某个自由群中的元素), 并将这个自由群模去由这些关系 (元素) 生成的正规子群, 返回得到的商群. (在 Section 8.1.6 中将展示如何处理更一般的商关系) 为简化定义, 我们使用 `deriving Group` to 直接生成 `myGroup` 上的群实例.

```
def myGroup := PresentedGroup {.of () ^ 3} deriving Group
```

presented groups 的整体性质确保了可以由那些将关系元都映射到目标群的单位元的函数构造群同态. 我们只需要一个函数和其满足该性质的证明, 就能用 `PresentedGroup.toGroup` 得到所需的群态射.

```
def myMap : Unit → Perm (Fin 5)
| () => c[1, 2, 3]

lemma compat_myMap :
    ∀ r ∈ ({.of () ^ 3} : Set (FreeGroup Unit)), FreeGroup.lift myMap r = 1 := by
  rintro _ rfl
  simp
  decide

def myNewMorphism : myGroup →* Perm (Fin 5) := PresentedGroup.toGroup compat_myMap

end FreeGroup
```

8.1.5 群作用

群作用是群论与其它数学相作用的一项重要方式. 群 `G` 在某类型 `X` 上的一个作用也正是 `G` 到 `Equiv.Perm X` 的一个态射. 所以某种意义上来说群作用已经为前文的讨论所涉及了. 不过我们并不想显式地给出这个态射; 它应该更多可能的由 Lean 自动推断得到. 对此, 我们有类型类 `MulAction G X`. 这种方式的缺点是, 当我们有同一个群在同一类型上的多个群作用时, 会产生一些曲折, 比如将定义携带不同类型类实例的同义类型.

实例化类型类之后，我们可以使用 `g • x` 表示群中的 `g` 作用在点 `x` 上.

```
noncomputable section GroupActions

example {G X : Type*} [Group G] [MulAction G X] (g g': G) (x : X) :
    g • (g' • x) = (g * g') • x :=
  (mul_smul g g' x).symm
```

对应的加法版本为 `AddAction`, 记号是 `+ᵥ`. 仿射空间的定义中会用到这个实例.

```
example {G X : Type*} [AddGroup G] [AddAction G X] (g g' : G) (x : X) :
    g +ᵥ (g' +ᵥ x) = (g + g') +ᵥ x :=
  (add_vadd g g' x).symm
```

底层的群态射为 `MulAction.toPermHom`.

```
open MulAction

example {G X : Type*} [Group G] [MulAction G X] : G →* Equiv.Perm X :=
  toPermHom G X
```

为了说明这一点，让我们看看如何将任意群 `G` 的 Cayley 同构嵌入定义为一个置换群，即 `Perm G`。

```
def CayleyIsoMorphism (G : Type*) [Group G] : G ≃* (toPermHom G G).range :=
  Equiv.Perm.subgroupOfMulAction G G
```

以下是这段话的翻译：

请注意，在上述定义之前，并不需要一定是群，而是可以是幺半群或者任何具有乘法操作的类型。

群的条件真正开始发挥作用是在我们需要将 X 分成轨道（orbits）时。对应于 X 的等价关系被称为 MulAction.orbitRel。这个等价关系并没有被声明为一个全局实例。

```
example {G X : Type*} [Group G] [MulAction G X] : Setoid X := orbitRel G X
```

利用这一点，我们可以说明在群 G 的作用下，集合 X 被划分为多个轨道 (orbits) 。

更具体地说，我们得到一个集合 X 与一个依赖乘积类型 (dependent product) (ω : orbitRel.Quotient G X) × (orbit G (Quotient.out' ω)) 之间的双射, 其中, Quotient.out' ω 简单地选择一个元素，该元素映射到 ω。

请注意，该依赖积的元素是形如 ⟨ω, x⟩ 的对，其中 x 的类型 orbit G (Quotient.out' ω) 依赖于 ω。

```
example {G X : Type*} [Group G] [MulAction G X] :
    X ≃ (ω : orbitRel.Quotient G X) × (orbit G (Quotient.out' ω)) :=
  MulAction.selfEquivSigmaOrbits G X
```

特别是，当 X 是有限集时，可以结合 Fintype.card_congr 和 Fintype.card_sigma 推导出 X 的基数等于其轨道 (orbits) 基数之和。

此外，这些轨道与 G 在稳定子 (stabilizers) 的左平移作用下的商集 (quotient) 一一对应 (存在双射关系)。这种通过子群的左平移作用定义的商集通常使用符号 / 来表示，因此可以用以下简洁的表述来说明。

```
example {G X : Type*} [Group G] [MulAction G X] (x : X) :
    orbit G x ≃ G ⧸ stabilizer G x :=
  MulAction.orbitEquivQuotientStabilizer G x
```

将上述两个结果结合的一个重要特殊情况是当 X 是一个通过平移作用配备了一个子群 H 的群 G 。在这种情况下，所有稳定子都是平凡的，因此每个轨道都与 H 一一对应，我们得到：

```
example {G : Type*} [Group G] (H : Subgroup G) : G ≃ (G ⧸ H) × H :=
  groupEquivQuotientProdSubgroup
```

这是我们前面看到的拉格朗日定理版本的概念变体。请注意，这个版本并不假设有限性。

作为本节的练习，让我们使用前面练习中的 conjugate 定义，构建一个群对其子群通过共轭作用的群作用。

```
variable {G : Type*} [Group G]
```

(continues on next page)

(continued from previous page)

```
lemma conjugate_one (H : Subgroup G) : conjugate 1 H = H := by
  sorry

instance : MulAction G (Subgroup G) where
  smul := conjugate
  one_smul := by
    sorry
  mul_smul := by
    sorry

end GroupActions
```

8.1.6 商群

在上述关于子群作用于群的讨论中，我们看到了商 `G ⧸ H` 的出现。一般来说，这只是一个类型。只有当 `H` 是正规子群时，它才能被赋予群结构，使得商映射是一个群态射（group morphism）（并且这种群结构是唯一的）。

正规性假设是一个类型类 `Subgroup.Normal`，因此类型类推断可以使用它来导出商上的群结构。

```
noncomputable section QuotientGroup

example {G : Type*} [Group G] (H : Subgroup G) [H.Normal] : Group (G ⧸ H) :=␣
↪inferInstance

example {G : Type*} [Group G] (H : Subgroup G) [H.Normal] : G →* G ⧸ H :=
  QuotientGroup.mk' H
```

通过 `QuotientGroup.lift` 可以使用商群的整体性质：当且仅当群态射 `φ` 的核包含 `N` 时，`φ` 可以下降为 `G ⧸ N` 上的态射。

```
example {G : Type*} [Group G] (N : Subgroup G) [N.Normal] {M : Type*}
    [Group M] (φ : G →* M) (h : N ≤ MonoidHom.ker φ) : G ⧸ N →* M :=
  QuotientGroup.lift N φ h
```

目标群被称为 `M` 这一事实表明，`M` 拥有幺半群结构就足够了。

一个重要的特殊情况是当 `N = ker φ` 时。在这种情况下，下降的态射是单射，并且我们得到一个从群到其像的群同构。这个结果通常被称为 **第一同构定理**。

```
example {G : Type*} [Group G] {M : Type*} [Group M] (φ : G →* M) :
    G ⧸ MonoidHom.ker φ →* MonoidHom.range φ :=
  QuotientGroup.quotientKerEquivRange φ
```

将整体性质应用于一个态射 `φ : G →* G'` 与商群投影 `Quotient.mk' N'` 的组合时，我们也可以构造一个从 `G ⧸ N` 到 `G' ⧸ N'` 的态射。

对 `φ` 的条件通常表述为“`φ` 应该将 `N` 映射到 `N'` 内部。”但这相当于要求 `φ` 应该将 `N'` 拉回到 `N` 上，而后者的条件更便于处理，因为拉回（pullback）的定义不涉及存在量词。

```
example {G G': Type*} [Group G] [Group G']
    {N : Subgroup G} [N.Normal] {N' : Subgroup G'} [N'.Normal]
    {φ : G →* G'} (h : N ≤ Subgroup.comap φ N') : G ⧸ N →* G' ⧸ N':=
  QuotientGroup.map N N' φ h
```

需要注意的一个细微点是，类型 `G ⧸ N` 实际上依赖于 `N``（在定义等同的范围内），因此证明两个正规子群 ``N` 和 `M` 是相等的并不足以使相应的商群相等。然而，在这种情况下，整体性质确实给出了一个同构。

```
example {G : Type*} [Group G] {M N : Subgroup G} [M.Normal]
    [N.Normal] (h : M = N) : G ⧸ M ≃* G ⧸ N := QuotientGroup.quotientMulEquivOfEq h
```

作为本节的最后一组练习，我们将证明：如果 `H` 和 `K` 是有限群 `G` 的不相交正规子群，且它们的基数的乘积等于 `G` 的基数，那么 `G` 同构于 `H × K`。请记住，这里的“不相交”意味着 `H ⊓ K = ⊥`。

我们从拉格朗日引理的一些操作开始，此时不假设子群是正规或不相交的。

```
section
variable {G : Type*} [Group G] {H K : Subgroup G}

open MonoidHom

#check Nat.card_pos -- 参数将从子群条件中隐式推导出
#check Subgroup.index_eq_card
#check Subgroup.index_mul_card
#check Nat.eq_of_mul_eq_mul_right

lemma aux_card_eq [Finite G] (h' : Nat.card G = Nat.card H * Nat.card K) :
    Nat.card (G ⧸ H) = Nat.card K := by
  sorry
```

从现在开始，我们假设我们的子群是正规且不相交的，并假设基数条件。现在，我们构造所需同构的第一个部分。

```
variable [H.Normal] [K.Normal] [Fintype G] (h : Disjoint H K)
  (h' : Nat.card G = Nat.card H * Nat.card K)

#check Nat.bijective_iff_injective_and_card
#check ker_eq_bot_iff
```

(continues on next page)

(continued from previous page)

```
#check restrict
#check ker_restrict

def iso₁ [Fintype G] (h : Disjoint H K) (h' : Nat.card G = Nat.card H * Nat.card K) :␣
↪K ≃* G ⧸ H := by
  sorry
```

现在我们可以定义第二个部分。我们将需要 `MonoidHom.prod`，它可以根据从 G_0 到 G_1 和 G_2 的态射构建一个从 G_0 到 $G_1 \times G_2$ 的态射。

```
def iso₂ : G ≃* (G ⧸ K) × (G ⧸ H) := by
  sorry
```

再将这两部分结合起来.

```
#check MulEquiv.prodCongr

def finalIso : G ≃* H × K :=
  sorry
```

8.2 环

8.2.1 环、环上的单位元、态射和子环

类型 `R` 上环结构的类型是 `Ring R`。乘法交换的变体为 `CommRing R`。我们已经看到，`ring` 策略会证明任何基于交换环公理的等式。

```
example {R : Type*} [CommRing R] (x y : R) : (x + y) ^ 2 = x ^ 2 + y ^ 2 + 2 * x * y␣
↪:= by ring
```

更为奇特的变体不要求 `R` 上的加法形成群，而仅需是加法幺半群。对应的类型类是 `Semiring R` 和 `CommSemiring R`。

自然数类型是 `CommSemiring R` 的一个重要实例，任何以自然数为值的函数类型也是如此。另一个重要的例子是环中的理想的类型，这将在下面讨论。

`ring` 策略的名称是双重误导性的，因为它假设了交换性，但也适用于半环。换句话说，它适用于任何 `CommSemiring`。

```
example (x y : ℕ) : (x + y) ^ 2 = x ^ 2 + y ^ 2 + 2 * x * y := by ring
```

还有一些环类和半环类的变体不假设乘法单位元的存在或乘法的结合性。我们在这里不讨论这些。

某些传统上在环论入门中教授的概念实际上是关于底层乘法幺半群的。一个突出的例子是环的单位元的定义。每个（乘法）幺半群 M 都有一个谓词 IsUnit : M → Prop，用来断言存在双边逆元，一个类型 Units M 表示单位元，并用记号 $M^\times$ 表示，以及到 M 的强制转换。类型 Units M 将一个可逆元素与其逆元以及确保它们彼此互为逆元的性质一起打包。此实现细节主要与定义可计算函数相关。在大多数情况下，可以使用 IsUnit.unit {x : M} : IsUnit x → $M^\times$ 构造一个单位元。在交换情况下，还可以使用 Units.mkOfMulEqOne (x y : M) : x * y = 1 → $M^\times$ 构造 x 作为单位元。

```
example (x : ℤˣ) : x = 1 ∨ x = -1 := Int.units_eq_one_or x

example {M : Type*} [Monoid M] (x : Mˣ) : (x : M) * x⁻¹ = 1 := Units.mul_inv x

example {M : Type*} [Monoid M] : Group Mˣ := inferInstance
```

两个（半）环 R 和 S 之间环态射的类型是 RingHom R S，记号为 R →+* S。

```
example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) (x y : R) :
    f (x + y) = f x + f y := f.map_add x y

example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) : Rˣ →* Sˣ :=
  Units.map f
```

同构的变体是 RingEquiv，记号为 ≃+*。

与子幺半群和子群类似，环 R 的子环有一个类型 Subring R，但这个类型远不如子群类型有用，因为环不能通过子环进行商构造。

```
example {R : Type*} [Ring R] (S : Subring R) : Ring S := inferInstance
```

值得注意的是，RingHom.range 产生一个子环。

8.2.2 理想与商

由于历史原因，Mathlib 仅为交换环提供了理想的理论。（环库最初的开发是为了快速推进现代代数几何的基础。）因此，在本节中我们将讨论交换（半）环。R 的理想被定义为将 R 视为 R-模的子模。模将在线性代数的章节中讨论，但这一实现细节基本上可以安全忽略，因为大多数（但不是全部）相关引理都已在理想的特殊背景中重新叙述。但是匿名投影记号并不总是像预期的那样工作。例如，不能将 Ideal.Quotient.mk I 替换为 I.Quotient.mk，因为下面的代码片段中有两个 .，因此它会被解析为 (Ideal.Quotient I).mk；但单独的 Ideal.Quotient 并不存在。

```
example {R : Type*} [CommRing R] (I : Ideal R) : R →+* R ⧸ I :=
  Ideal.Quotient.mk I

example {R : Type*} [CommRing R] {a : R} {I : Ideal R} :
```

(continues on next page)

(continued from previous page)

```
    Ideal.Quotient.mk I a = 0 ↔ a ∈ I :=
  Ideal.Quotient.eq_zero_iff_mem
```

商环的整体性质是 `Ideal.Quotient.lift`。

```
example {R S : Type*} [CommRing R] [CommRing S] (I : Ideal R) (f : R →+* S)
    (H : I ≤ RingHom.ker f) : R ⧸ I →+* S :=
  Ideal.Quotient.lift I f H
```

特别的，其导出了环的第一同构定理。

```
example {R S : Type*} [CommRing R] [CommRing S](f : R →+* S) :
    R ⧸ RingHom.ker f ≃+* f.range :=
  RingHom.quotientKerEquivRange f
```

理想在包含关系下形成一个完备格结构，同时也具有半环结构。这两个结构相互作用良好。

```
variable {R : Type*} [CommRing R] {I J : Ideal R}

example : I + J = I ⊔ J := rfl

example {x : R} : x ∈ I + J ↔ ∃ a ∈ I, ∃ b ∈ J, a + b = x := by
  simp [Submodule.mem_sup]

example : I * J ≤ J := Ideal.mul_le_left

example : I * J ≤ I := Ideal.mul_le_right

example : I * J ≤ I ⊓ J := Ideal.mul_le_inf
```

可以使用环态射分别通过 `Ideal.map` 和 `Ideal.comap` 将理想前推（push forward）或拉回（pull back）。通常情况下，后者更方便使用，因为它不涉及存在量词。这也解释了为何它被用来表达构造商环之间态射的条件。

```
example {R S : Type*} [CommRing R] [CommRing S] (I : Ideal R) (J : Ideal S) (f : R␣
↪→+* S)
    (H : I ≤ Ideal.comap f J) : R ⧸ I →+* S ⧸ J :=
  Ideal.quotientMap J f H
```

一个需要注意的细微点是，类型 `R ⧸ I` 实际上依赖于 `I``（在定义等同的范围内），因此证明两个理想 ``I` 和 `J` 相等并不足以使相应的商环相等。然而，在这种情况下，整体性质确实提供了一个同构。

```
example {R : Type*} [CommRing R] {I J : Ideal R} (h : I = J) : R ⧸ I ≃+* R ⧸ J :=
  Ideal.quotEquivOfEq h
```

我们现在可以将中国剩余定理的同构作为一个示例呈现。请注意，索引下确界符号 ⨅ 与类型大乘积符号 Π 的区别。取决于你的字体，它们可能很难区分。

```
example {R : Type*} [CommRing R] {ι : Type*} [Fintype ι] (f : ι → Ideal R)
    (hf : ∀ i j, i ≠ j → IsCoprime (f i) (f j)) : (R ⧸ ⨅ i, f i) ≃+* Π i, R ⧸ f i :=
  Ideal.quotientInfRingEquivPiQuotient f hf
```

初等版本的中国剩余定理（关于 ZMod 的表述）可以轻松地从前述定理推导出来：

```
open BigOperators PiNotation

example {ι : Type*} [Fintype ι] (a : ι → ℕ) (coprime : ∀ i j, i ≠ j → (a i).Coprime
↪(a j)) :
    ZMod (∏ i, a i) ≃+* Π i, ZMod (a i) :=
  ZMod.prodEquivPi a coprime
```

作为一系列练习，我们将在一般情况下重新证明中国剩余定理。

我们首先需要定义定理中出现的映射，作为一个环态射，利用商环的整体性质。

```
variable {ι R : Type*} [CommRing R]
open Ideal Quotient Function

#check Pi.ringHom
#check ker_Pi_Quotient_mk

/-- 从 ``R ⧸ I i`` 到 ``Π i, R ⧸ I i`` 的同态映射，该映射在中国剩余定理中出现。 -/
def chineseMap (I : ι → Ideal R) : (R ⧸ ⨅ i, I i) →+* Π i, R ⧸ I i :=
  sorry
```

确保以下两个引理可以通过 rfl 证明。

```
lemma chineseMap_mk (I : ι → Ideal R) (x : R) :
    chineseMap I (Quotient.mk _ x) = fun i : ι ↦ Ideal.Quotient.mk (I i) x :=
  sorry

lemma chineseMap_mk' (I : ι → Ideal R) (x : R) (i : ι) :
    chineseMap I (mk _ x) i = mk (I i) x :=
  sorry
```

下一个引理证明了中国剩余定理的简单部分，对于理想族没有任何假设。该证明不到一行即可完成。

```
#check injective_lift_iff

lemma chineseMap_inj (I : ι → Ideal R) : Injective (chineseMap I) := by
  sorry
```

我们现在准备进入定理的核心部分，它将展示我们的 chineseMap 的满射性。首先，我们需要了解几种表达互素性（也称为互为最大假设 co-maximality assumption ）的方法。以下仅需要使用前两种表达方式。

```
#check IsCoprime
#check isCoprime_iff_add
#check isCoprime_iff_exists
#check isCoprime_iff_sup_eq
#check isCoprime_iff_codisjoint
```

我们借此机会对 Finset 使用归纳法。以下列出了关于 Finset 的相关引理。请记住，ring 策略适用于半环，并且环的理想构成一个半环。

```
#check Finset.mem_insert_of_mem
#check Finset.mem_insert_self

theorem isCoprime_Inf {I : Ideal R} {J : ι → Ideal R} {s : Finset ι}
    (hf : ∀ j ∈ s, IsCoprime I (J j)) : IsCoprime I (∏ j ∈ s, J j) := by
  classical
  simp_rw [isCoprime_iff_add] at *
  induction s using Finset.induction with
  | empty =>
      simp
  | @insert i s _ hs =>
      rw [Finset.iInf_insert, inf_comm, one_eq_top, eq_top_iff, ← one_eq_top]
      set K := ∏ j ∈ s, J j
      calc
        1 = I + K                 := sorry
        _ = I + K * (I + J i)     := sorry
        _ = (1 + K) * I + K * J i := sorry
        _ ≤ I + K ⊓ J i           := sorry
```

我们现在可以证明在中国剩余定理中出现的映射的满射性。

```
lemma chineseMap_surj [Fintype ι] {I : ι → Ideal R}
    (hI : ∀ i j, i ≠ j → IsCoprime (I i) (I j)) : Surjective (chineseMap I) := by
```

(continues on next page)

(continued from previous page)

```
  classical
  intro g
  choose f hf using fun i ↦ Ideal.Quotient.mk_surjective (g i)
  have key : ∀ i, ∃ e : R, mk (I i) e = 1 ∧ ∀ j, j ≠ i → mk (I j) e = 0 := by
    intro i
    have hI' : ∀ j ∈ ({i} : Finset ι)ᶜ, IsCoprime (I i) (I j) := by
      sorry
    sorry
  choose e he using key
  use mk _ (∑ i, f i * e i)
  sorry
```

将这些部分结合起来：

```
noncomputable def chineseIso [Fintype ι] (f : ι → Ideal R)
    (hf : ∀ i j, i ≠ j → IsCoprime (f i) (f j)) : (R ⧸ ⨅ i, f i) ≃+* Π i, R ⧸ f i :=
  { Equiv.ofBijective _ ⟨chineseMap_inj f, chineseMap_surj hf⟩,
    chineseMap f with }
```

8.2.3 代数与多项式

给定一个交换（半）环 `R`，一个 `R` 上的 *代数* (algebra) `R` 是一个半环 `A`，其配备了一个环态射，其像与 `A` 的每个元素可交换。这被编码为一个类型类 `Algebra R A`。从 `R` 到 `A` 的态射称为结构映射，并在 Lean 中记作 `algebraMap R A : R →+* A`。对某个 `r : R`，将 `a : A` 与 `algebraMap R A r` 相乘被称为 `a` 被 `r` 的标量乘法，记为 `r • a`。请注意，这种代数的概念有时称为 *结合幺代数* (associative unital algebra)，以强调存在更一般的代数概念。

`algebraMap R A` 是一个环态射的事实打包了许多标量乘法的性质，例如以下内容：

```
example {R A : Type*} [CommRing R] [Ring A] [Algebra R A] (r r' : R) (a : A) :
    (r + r') • a = r • a + r' • a :=
  add_smul r r' a

example {R A : Type*} [CommRing R] [Ring A] [Algebra R A] (r r' : R) (a : A) :
    (r * r') • a = r • r' • a :=
  mul_smul r r' a
```

`R`-代数 `A` 和 `B` 之间的态射是环态射，它们与 `R` 元素的标量乘法可交换。它们是具有类型 `AlgHom R A B` 的打包态射，记号为 `A →ₐ[R] B`。

非交换代数的重要示例包括自同态代数和方阵代数，这两个将在线性代数一章中讨论。在本章中，我们将讨论最重要的交换代数之一，即多项式代数。

系数在 R 中的一元多项式代数称为 Polynomial R，当打开 Polynomial 命名空间时，它可以写作 R[X]。从 R 到 R[X] 的代数结构映射记为 C，它表示“常数”，因为相应的多项式函数始终是常数。未定元记为 X。

```
open Polynomial

example {R : Type*} [CommRing R] : R[X] := X

example {R : Type*} [CommRing R] (r : R) := X - C r
```

在上面的第一个示例中，至关重要的是要为 Lean 提供预期的类型，因为它无法从定义的主体中推断出来。在第二个示例中，目标多项式代数可以通过我们对 C r 的使用推断出来，因为已知 r 的类型。

由于 C 是从 R 到 R[X] 的环态射，我们可以在 R[X] 环中计算之前，使用所有环态射引理，例如 map_zero、map_one、map_mul 和 map_pow。例如：

```
example {R : Type*} [CommRing R] (r : R) : (X + C r) * (X - C r) = X ^ 2 - C (r ^ 2)␣
↪:= by
  rw [C.map_pow]
  ring
```

可以用 Polynomial.coeff 获取系数

```
example {R : Type*} [CommRing R] (r:R) : (C r).coeff 0 = r := by simp

example {R : Type*} [CommRing R] : (X ^ 2 + 2 * X + C 3 : R[X]).coeff 1 = 2 := by simp
```

定义多项式的次数总是比较棘手，因为零多项式是一个特殊情况。Mathlib 有两个变体：Polynomial.natDegree : R[X] → ℕ 将零多项式的次数指定为 0，而 Polynomial.degree : R[X] → WithBot ℕ 将其指定为 ⊥。

在后者中，WithBot ℕ 可以视为 ℕ ∪ {-∞}，只不过 -∞ 被表示为 ⊥，与完备格中的底元素同一个符号。此特殊值用于零多项式的次数，并且在加法中是吸收元。（在乘法中它几乎是吸收元，但 ⊥ * 0 = 0 除外。）

理论而言，degree 版本是正确的那一个。例如，它允许我们陈述关于乘积次数的预期公式（假设基环没有零因子）。

```
example {R : Type*} [Semiring R] [NoZeroDivisors R] {p q : R[X]} :
    degree (p * q) = degree p + degree q :=
  Polynomial.degree_mul
```

而对于 natDegree 的版本，则需要假设多项式非零。

```
example {R : Type*} [Semiring R] [NoZeroDivisors R] {p q : R[X]} (hp : p ≠ 0) (hq : q␣
↪≠ 0) :
```

(continues on next page)

(continued from previous page)

```
    natDegree (p * q) = natDegree p + natDegree q :=
  Polynomial.natDegree_mul hp hq
```

然而，ℕ 的使用要比 WithBot ℕ 更友好，因此 Mathlib 提供了这两种版本并提供了在它们之间转换的引理。此外，当计算复合多项式的次数时，natDegree 是更方便的定义。多项式的复合是 Polynomial.comp，我们有：

```
example {R : Type*} [Semiring R] [NoZeroDivisors R] {p q : R[X]} :
    natDegree (comp p q) = natDegree p * natDegree q :=
  Polynomial.natDegree_comp
```

多项式产生多项式函数：任何多项式都可以通过 Polynomial.eval 在 R 上进行求值。

```
example {R : Type*} [CommRing R] (P: R[X]) (x : R) := P.eval x

example {R : Type*} [CommRing R] (r : R) : (X - C r).eval r = 0 := by simp
```

特别地，有一个谓词 IsRoot，它用于表示当一个多项式在 R 中的某些元素 r 处取零时成立。

```
example {R : Type*} [CommRing R] (P : R[X]) (r : R) : IsRoot P r ↔ P.eval r = 0 :=␣
↪Iff.rfl
```

我们希望能够说明，在假设 R 没有零因子的情况下，一个多项式的根（按重数计算）最多不超过其次数。然而，零多项式的情况再次变得麻烦。

因此，Mathlib 定义了 Polynomial.roots，它将一个多项式 P 映射到一个多重集合（multiset），即：- 如果 P 为零多项式，该集合被定义为空集；- 否则，该集合为 P 的根，并记录其重数。

此定义仅适用于底层环是整域的情况，因为在其他情况下，该定义不具有良好的性质。

```
example {R : Type*} [CommRing R] [IsDomain R] (r : R) : (X - C r).roots = {r} :=
  roots_X_sub_C r

example {R : Type*} [CommRing R] [IsDomain R] (r : R) (n : ℕ):
    ((X - C r) ^ n).roots = n • {r} :=
  by simp
```

Polynomial.eval 和 Polynomial.roots 都仅考虑系数环。它们不能让我们说明 X ^ 2 - 2 : ℚ[X] 在 R 中有根，或 X ^ 2 + 1 : R[X] 在 ℂ 中有根。为此，我们需要 Polynomial.aeval，它可以在任意 R-代数中对 P : R[X] 进行求值。

更具体地说，给定一个半环 A 和 Algebra R A 的实例，Polynomial.aeval 会沿着在元素 a 处的 R-代数态射将多项式的每个元素发送出去。由于 AlgHom 可以强制转换为函数，因此可以将其应用于多项式。

但 aeval 并没有一个多项式作为参数，因此不能像在上面使用 P.eval 那样使用点符号表示法。

```
example : aeval Complex.I (X ^ 2 + 1 : ℝ[X]) = 0 := by simp
```

在这种情况下，与 roots 对应的函数是 aroots，它接受一个多项式和一个代数，并输出一个多重集合（关于零多项式的警告与 roots 相同）。

```
open Complex Polynomial

example : aroots (X ^ 2 + 1 : ℝ[X]) ℂ = {Complex.I, -I} := by
  suffices roots (X ^ 2 + 1 : ℂ[X]) = {I, -I} by simpa [aroots_def]
  have factored : (X ^ 2 + 1 : ℂ[X]) = (X - C I) * (X - C (-I)) := by
    have key : (C I * C I : ℂ[X]) = -1 := by simp [← C_mul]
    rw [C_neg]
    linear_combination key
  have p_ne_zero : (X - C I) * (X - C (-I)) ≠ 0 := by
    intro H
    apply_fun eval 0 at H
    simp [eval] at H
  simp only [factored, roots_mul p_ne_zero, roots_X_sub_C]
  rfl

-- Mathlib 知晓达朗贝尔-高斯定理：``ℂ`` 是代数闭域。
example : IsAlgClosed ℂ := inferInstance
```

更一般地说，给定一个环态射 f : R →+* S，可以使用 Polynomial.eval₂ 在 S 中的一个点上对 P : R[X] 进行求值。由于它不假设存在 Algebra R S 实例，因此它生成从 R[X] 到 S 的实际函数，因此点符号可以像预期那样正常工作。

```
#check (Complex.ofRealHom : ℝ →+* ℂ)

example : (X ^ 2 + 1 : ℝ[X]).eval₂ Complex.ofRealHom Complex.I = 0 := by simp
```

我们最后简要提一下多变量多项式。给定一个交换半环 R，系数在 R 且不定元通过类型 σ 索引的多项式所构成的 R-代数为 MVPolynomial σ R。

给定 i : σ，对应的不定元记为 MvPolynomial.X i。（通常可以打开 MVPolynomial 命名空间以将其缩写为 X i。）

例如，如果我们需要两个不定元，可以使用 Fin 2 作为 σ 并写出定义单位圆的多项式（在 $\mathbb{R}^2$ 中）如下：

```
open MvPolynomial

def circleEquation : MvPolynomial (Fin 2) ℝ := X 0 ^ 2 + X 1 ^ 2 - 1
```

函数应用具有非常高的优先级，因此上述表达式可以读取为 (X 0) ^ 2 + (X 1) ^ 2 - 1。

我们可以对其进行求值，以确保坐标为 $(1, 0)$ 的点位于圆上。此外，`![...]` 表示符号表示元素属于 `Fin n → X`，其中自然数 `n` 是由参数的数量决定的，而某种类型 `X` 是由参数的类型决定的。

```
example : MvPolynomial.eval ![0, 1] circleEquation = 0 := by simp [circleEquation]
```

CHAPTER

NINE

线性代数

9.1 向量空间与线性映射

9.1.1 向量空间

我们将直接从抽象线性代数开始，讨论定义在任意域上的向量空间。不过，你也可以在 Section 9.4.1 中找到关于矩阵的信息，该部分在逻辑上并不依赖于本抽象理论。Mathlib 实际上处理的是更广义的线性代数，使用模（module）一词，但现在我们暂时当作这只是一个古怪的拼写习惯。

“设 K 是一个域，且 V 是定义在 K 上的向量空间”（并将其作为后续结论的隐式参数）的表达方式是：

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]
```

我们在 Chapter 7 中解释了为什么需要两个独立的类型类 *[AddCommGroup V]* 和 *[Module K V]*。

简而言之，数学上我们想表达的是：拥有一个域 K 上的向量空间结构意味着拥有一个加法交换群结构。虽然可以告诉 Lean 这一点，但如果这样做，每当 Lean 需要在类型 V 上寻找该群结构时，它就会尝试去寻找带有完全未指定域 K 的向量空间结构，而这个 K 无法从 V 中推断出来。这会严重影响类型类合成系统的效率和稳定性。

向量 v 与标量 a 的乘法记作 $a \bullet v$。下面示例列举了该操作与加法之间的若干代数规则。当然，*simp* 或 *apply?* 策略可以自动找到这些证明。还有一个 *module* 策略，类似于在交换环中使用 *ring* 策略、在群中使用 *group* 策略，可以根据向量空间和域的公理自动解决相关目标。但需要记住的是，在引理名称中，标量乘法通常缩写为 *smul*。

```
example (a : K) (u v : V) : a • (u + v) = a • u + a • v :=
  smul_add a u v

example (a b : K) (u : V) : (a + b) • u = a • u + b • u :=
  add_smul a b u

example (a b : K) (u : V) : a • b • u = b • a • u :=
  smul_comm a b u
```

作为对更高级读者的简要提示，正如术语所暗示的那样，Mathlib 的线性代数不仅涵盖了定义在（不一定是交换）环上的模（modules）。事实上，它甚至涵盖了半环上的半模（semi-modules）。如果你认为不需要如此广泛的泛化，可以思考下面这个例子，它很好地体现了理想对子模作用的许多代数规则：

```
example {R M : Type*} [CommSemiring R] [AddCommMonoid M] [Module R M] :
    Module (Ideal R) (Submodule R M) :=
  inferInstance
```

9.1.2 线性映射

接下来我们需要引入线性映射。类似于群同态，Mathlib 中的线性映射是“打包”的映射，即由一个映射和其线性性质证明组成的整体。这些打包映射在应用时会自动转换为普通函数。关于这一设计的更多信息，请参见 Chapter 7。

两个域 `K` 上的向量空间 `V` 和 `W` 之间的线性映射类型记作 `V →ₗ[K] W`，其中下标的 *l* 表示“linear”（线性）。一开始在符号中显式指定 `K` 可能感觉有些奇怪，但当涉及多个域时，这一点尤为重要。例如，从复数域 到 的实线性映射形式为 $z \mapsto az + b\bar{z}$，而复线性映射则仅限于形式为 $z \mapsto az$ 的映射，这一区别在复分析中极为关键。

```
variable {W : Type*} [AddCommGroup W] [Module K W]

variable (φ : V →ₗ[K] W)

example (a : K) (v : V) : φ (a • v) = a • φ v :=
  map_smul φ a v

example (v w : V) : φ (v + w) = φ v + φ w :=
  map_add φ v w
```

注意，*V →ₗ[K] W* 本身也携带了丰富的代数结构（这也是将这些映射打包的动机之一）。它是一个 *K*-向量空间，因此我们可以对线性映射进行加法运算，也可以对它们进行标量乘法。

```
variable (ψ : V →ₗ[K] W)

#check (2 • φ + ψ : V →ₗ[K] W)
```

使用打包映射的一个缺点是，无法直接使用普通函数的组合运算。我们需要使用专门的组合函数 *LinearMap.comp*，或者使用专用符号 *∘ₗ* 来表示线性映射的组合。

```
variable (θ : W →ₗ[K] V)

#check (φ.comp θ : W →ₗ[K] W)
#check (φ ∘ₗ θ : W →ₗ[K] W)
```

构造线性映射主要有两种方式。第一种是通过提供函数本体和线性性质的证明来构建该结构。通常，可以借

助代码操作快速完成：你只需输入 `example : V →[K] V := _` 然后对下划线使用“生成骨架”（Generate a skeleton）代码操作，即可自动生成所需的结构框架。

```
example : V →[K] V where
  toFun v := 3 • v
  map_add' _ _ := smul_add ..
  map_smul' _ _ := smul_comm ..
```

你可能会好奇为什么 *LinearMap* 结构中用于证明的字段名称都带有撇号。这是因为这些证明是在函数强制转换（coercion）之前定义的，因此它们是基于 *LinearMap.toFun* 表达的。随后，它们又被重新表达为基于函数强制转换的 *LinearMap.map_add* 和 *LinearMap.map_smul*。事情还没结束。我们还需要一个适用于任意保持加法的（打包）映射的 *map_add* 版本，比如加法群同态、线性映射、连续线性映射、*K*-代数映射等。这个版本是定义在根命名空间的 *map_add*。而中间版本 *LinearMap.map_add* 虽有些冗余，但支持点记法（dot notation），在某些情况下使用起来更方便。*map_smul* 也有类似情况。整体框架请参见 Chapter 7 的详细解释。

```
#check (φ.map_add' : ∀ x y : V, φ.toFun (x + y) = φ.toFun x + φ.toFun y)
#check (φ.map_add : ∀ x y : V, φ (x + y) = φ x + φ y)
#check (map_add φ : ∀ x y : V, φ (x + y) = φ x + φ y)
```

我们也可以利用 Mathlib 中已有的线性映射，通过各种组合子（combinators）来构造新的线性映射。例如，上述例子已经在 Mathlib 中以 *LinearMap.lsmul K V 3* 的形式存在。这里 *K* 和 *V* 是显式参数，原因有几个：最主要的是，如果只写 *LinearMap.lsmul 3*，Lean 无法推断出 *V*，甚至无法推断出 *K*。此外，*LinearMap.lsmul K V* 本身也是一个有趣的对象：它的类型是 `K →[K] V →[K] V`，意味着它是一个从域 *K*`（视为其自身上的向量空间）到从 `*V* 到 *V* 的 *K*-线性映射空间的线性映射。

```
#check (LinearMap.lsmul K V 3 : V →[K] V)
#check (LinearMap.lsmul K V : K →[K] V →[K] V)
```

还有一个线性同构类型 *LinearEquiv*，用符号表示为 `V ≃[K] W`，对于 *f : V ≃[K] W*，它的逆映射是 `f.symm : W ≃[K] V`。两个线性同构 *f* 和 *g* 的复合是 `f.trans g`，也可用符号 `f ≪≫ g` 表示。*V* 的恒等同构是 `LinearEquiv.refl K V`，该类型的元素在需要时会自动强制转换为对应的同态映射或函数。

```
example (f : V ≃[K] W) : f ≪≫ f.symm = LinearEquiv.refl K V :=
  f.self_trans_symm
```

可以使用 *LinearEquiv.ofBijective* 从一个双射的同态构造线性同构。这样构造的逆映射函数是不可计算的。

```
noncomputable example (f : V →[K] W) (h : Function.Bijective f) : V ≃[K] W :=
  .ofBijective f h
```

请注意，在上述例子中，Lean 利用声明的类型自动识别 *.ofBijective* 是指 *LinearEquiv.ofBijective*，无需显式打开命名空间。

9.1.3 向量空间的直和与直积

我们可以利用已有的向量空间，通过直和和直积构造新的向量空间。从两个向量空间开始，在该情形下直和与直积无区别，我们可以直接使用积类型。下面的代码片段展示了如何将所有结构映射（含入（inclusion）映射和投影映射）构造为线性映射，以及构造映射到积空间和从和空间出的万有性质（universal properties）。如果你对范畴论中和与积的区别不熟悉，可以忽略“万有性质”相关术语，专注于示例中的类型即可。

```
section binary_product

variable {W : Type*} [AddCommGroup W] [Module K W]
variable {U : Type*} [AddCommGroup U] [Module K U]
variable {T : Type*} [AddCommGroup T] [Module K T]

-- 第一投影映射
example : V × W →ₗ[K] V := LinearMap.fst K V W

-- 第二投影映射
example : V × W →ₗ[K] W := LinearMap.snd K V W

-- 直积的万有性质
example (φ : U →ₗ[K] V) (ψ : U →ₗ[K] W) : U →ₗ[K]  V × W := LinearMap.prod φ ψ

-- 直积映射满足预期行为，第一分量
example (φ : U →ₗ[K] V) (ψ : U →ₗ[K] W) : LinearMap.fst K V W ∘ₗ LinearMap.prod φ ψ =␣
↪φ := rfl

-- 直积映射满足预期行为，第二分量
example (φ : U →ₗ[K] V) (ψ : U →ₗ[K] W) : LinearMap.snd K V W ∘ₗ LinearMap.prod φ ψ =␣
↪ψ := rfl

-- 我们也可以并行组合映射
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] T) : (V × W) →ₗ[K] (U × T) := φ.prodMap ψ

-- 这只是通过将投影与万有性质结合实现
example (φ : V →ₗ[K] U) (ψ : W →ₗ[K] T) :
  φ.prodMap ψ = (φ ∘ₗ .fst K V W).prod (ψ ∘ₗ .snd K V W) := rfl

-- 第一包含（inclusion）映射
example : V →ₗ[K] V × W := LinearMap.inl K V W

-- 第二含入映射
example : W →ₗ[K] V × W := LinearMap.inr K V W
```

(continues on next page)

(continued from previous page)

```
-- 直和（又称余积）的万有性质
example (φ : V →[K] U) (ψ : W →[K] U) : V × W →[K] U := φ.coprod ψ

-- 余积映射满足预期行为，第一分量
example (φ : V →[K] U) (ψ : W →[K] U) : φ.coprod ψ ∘ LinearMap.inl K V W = φ :=
  LinearMap.coprod_inl φ ψ

-- 余积映射满足预期行为，第二分量
example (φ : V →[K] U) (ψ : W →[K] U) : φ.coprod ψ ∘ LinearMap.inr K V W = ψ :=
  LinearMap.coprod_inr φ ψ

-- 余积映射的定义符合预期
example (φ : V →[K] U) (ψ : W →[K] U) (v : V) (w : W) :
    φ.coprod ψ (v, w) = φ v + ψ w :=
  rfl

end binary_product
```

现在我们转向任意族向量空间的直和与直积。这里我们将展示如何定义一个向量空间族，并访问直和与直积的万有性质。请注意，直和符号限定在 *DirectSum* 命名空间中，且直和的万有性质要求索引类型具备可判定等价性（这在某种程度上是实现上的偶然情况）。

```
section families
open DirectSum

variable {ι : Type*} [DecidableEq ι]
         (V : ι → Type*) [∀ i, AddCommGroup (V i)] [∀ i, Module K (V i)]

-- 直和的万有性质将从各个直和因子映射出的映射组装成从直和映射出的映射
example (φ : Π i, (V i →[K] W)) : (⊕ i, V i) →[K] W :=
  DirectSum.toModule K ι W φ

-- 直积的万有性质将映射到各个因子的映射组装成映射到直积
example (φ : Π i, (W →[K] V i)) : W →[K] (Π i, V i) :=
  LinearMap.pi φ

--  来自直积的投影映射
example (i : ι) : (Π j, V j) →[K] V i := LinearMap.proj i
```

(continues on next page)

(continued from previous page)

```
-- 含入映射到直和
example (i : ι) : V i →ₗ[K] (⊕ i, V i) := DirectSum.lof K ι V i

-- 含入映射到直积
example (i : ι) : V i →ₗ[K] (Π i, V i) := LinearMap.single K V i

-- 若 `ι` 是有限类型，直和与直积之间存在线性同构
example [Fintype ι] : (⊕ i, V i) ≃ₗ[K] (Π i, V i) :=
  linearEquivFunOnFintype K ι V

end families
```

9.2 子空间和商空间

9.2.1 子空间

正如线性映射是打包结构，向量空间 V 的线性子空间同样是一个打包结构，包含 V 中的一个子集，称为子空间的载体（carrier），并具备相关的闭合性质。这里依然使用“模（module）”一词，而非向量空间，是因为 Mathlib 在线性代数中采用了更广泛的理论框架。

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]

example (U : Submodule K V) {x y : V} (hx : x ∈ U) (hy : y ∈ U) :
    x + y ∈ U :=
  U.add_mem hx hy

example (U : Submodule K V) {x : V} (hx : x ∈ U) (a : K) :
    a • x ∈ U :=
  U.smul_mem a hx
```

在上述例子中，需要理解的是，*Submodule K V* 表示 V 上的 K-线性子空间的类型，而非 *Set V* 中元素 U 的谓词 *IsSubmodule U*。*Submodule K V* 配备了到 *Set V* 的强制类型转换（coercion）以及对 V 的成员关系谓词。有关该设计的原因及实现方式，可参见 Section 7.3。

当然，两个子空间相等当且仅当它们包含相同的元素。此性质已被注册用于 *ext* 策略，后者可用来证明两个子空间相等，方式与证明两个集合相等相同。

例如，要表述并证明实数域 $\mathbb{R}$ 是复数域 $\mathbb{C}$ 上的一个 $\mathbb{R}$-线性子空间，实质上是构造一个类型为 *Submodule* $\mathbb{R}$ $\mathbb{C}$ 的项，使其强制转换为 *Set* $\mathbb{C}$ 后正是 $\mathbb{R}$，或者更准确地说，是 $\mathbb{R}$ 在 $\mathbb{C}$ 中的像。

```
noncomputable example : Submodule ℝ ℂ where
```

(continues on next page)

(continued from previous page)

```
  carrier := Set.range ((↑) : ℝ → ℂ)
  add_mem' := by
    rintro _ _ ⟨n, rfl⟩ ⟨m, rfl⟩
    use n + m
    simp
  zero_mem' := by
    use 0
    simp
  smul_mem' := by
    rintro c - ⟨a, rfl⟩
    use c*a
    simp
```

Submodule 中证明字段末尾的撇号与 *LinearMap* 中的类似。这些字段以 *carrier* 字段为基础定义，因为它们在 *Membership* 实例之前被定义。随后，它们被我们之前看到的 *Submodule.add_mem*、*Submodule.zero_mem* 和 *Submodule.smul_mem* 所取代。

作为操作子空间和线性映射的练习，你将定义线性映射下子空间的原像（pre-image）（当然，后面会看到 Mathlib 已经包含了相关定义）。记住，可以使用 *Set.mem_preimage* 来对涉及成员资格和原像的陈述进行重写。除了上述关于 *LinearMap* 和 *Submodule* 的引理外，这是你唯一需要用到的引理。

```
def preimage {W : Type*} [AddCommGroup W] [Module K W] (φ : V →ₗ[K] W) (H : Submodule␣
↪K W) :
    Submodule K V where
  carrier := φ ⁻¹' H
  zero_mem' := by
    dsimp
    sorry
  add_mem' := by
    sorry
  smul_mem' := by
    dsimp
    sorry
```

使用类型类，Mathlib 知道向量空间的子空间继承了向量空间结构。

```
example (U : Submodule K V) : Module K U := inferInstance
```

这个例子很微妙。对象 `U` 不是一个类型，但 Lean 会通过将其解释为 `V` 的一个子类型来自动将其强制转换为一个类型。因此，上面的例子可以更明确地重述为：

```
example (U : Submodule K V) : Module K {x : V // x ∈ U} := inferInstance
```

9.2.2 完全格结构和内直和

拥有类型 *Submodule K V* 而非谓词 *IsSubmodule : Set V → Prop* 的一个重要好处是，可以方便地赋予 *Submodule K V* 额外的结构。其中最重要的是它在包含关系下具有完备格结构。例如，不再需要单独的引理来说明两个子空间的交集仍是子空间，我们可以直接使用格的运算符 来构造交集。随后便可对该结构应用任意格论相关的引理。

下面我们验证，两个子空间下确界对应的底层集合，正是它们的（定义上的）交集。

```
example (H H' : Submodule K V) :
    ((H ⊓ H' : Submodule K V) : Set V) = (H : Set V) ∩ (H' : Set V) := rfl
```

对底层集合的交集使用不同符号表示可能看起来有些奇怪，但这种对应关系并不适用于上确界运算和集合的并集，因为子空间的并集通常并不是子空间。取而代之的是，需要使用由并集生成的子空间，这通常通过 *Submodule.span* 来实现。

```
example (H H' : Submodule K V) :
    ((H ⊔ H' : Submodule K V) : Set V) = Submodule.span K ((H : Set V) ∪ (H' : Set␣
↪V)) := by
  simp [Submodule.span_union]
```

另一个细节是，向量空间本身 *V* 并不属于类型 *Submodule K V*，因此需要一种方式来表示将 *V* 视为自身的子空间。这也由格结构提供支持：整个空间作为该格的最大元存在。

```
example (x : V) : x ∈ (⊤ : Submodule K V) := trivial
```

同样，这个格的底部元素是唯一元素为零元素的子空间。

```
example (x : V) : x ∈ (⊥ : Submodule K V) ↔ x = 0 := Submodule.mem_bot K
```

特别地，我们可以讨论处于（内部）直和的子空间的情况。在两个子空间的情况下，我们使用通用谓词 `IsCompl`，它适用于任何有界偏序类型。在一般子空间族的情况下，我们使用 `DirectSum.IsInternal`。

```
-- 如果两个子空间是直和，则它们生成整个空间。
example (U V : Submodule K V) (h : IsCompl U V) :
  U ⊔ V = ⊤ := h.sup_eq_top

-- 如果两个子空间是直和，则它们的交集仅为零。
example (U V : Submodule K V) (h : IsCompl U V) :
  U ⊓ V = ⊥ := h.inf_eq_bot
```

(continues on next page)

(continued from previous page)

```
section
open DirectSum
variable {ι : Type*} [DecidableEq ι]

-- 如果子空间是直和，则它们生成整个空间。
example (U : ι → Submodule K V) (h : DirectSum.IsInternal U) :
  ⨆ i, U i = ⊤ := h.submodule_iSup_eq_top

-- 如果子空间是直和，则它们的交集仅为零。
example {ι : Type*} [DecidableEq ι] (U : ι → Submodule K V) (h : DirectSum.IsInternal
↪U)
    {i j : ι} (hij : i ≠ j) : U i ⊓ U j = ⊥ :=
  (h.submodule_independent.pairwiseDisjoint hij).eq_bot

-- 这些条件表征了直和。
#check DirectSum.isInternal_submodule_iff_independent_and_iSup_eq_top

-- 与外直和的关系：如果一族子空间是内直和，则它们的外直和映射到 `V` 的映射是线性同构。
noncomputable example {ι : Type*} [DecidableEq ι] (U : ι → Submodule K V)
    (h : DirectSum.IsInternal U) : (⨁ i, U i) ≃ₗ[K] V :=
  LinearEquiv.ofBijective (coeLinearMap U) h
end
```

9.2.3 由集合生成的子空间

除了从现有子空间构建子空间，我们还可以使用 `Submodule.span K s` 从任何集合 `s` 构建子空间，该函数构建包含 `s` 的最小子空间。在纸面上，通常使用该空间由 `s` 的所有线性组合构成的事实。但通常更有效的方法是使用其万有性质，即 `Submodule.span_le`，以及整个 Galois 连接理论。

```
example {s : Set V} (E : Submodule K V) : Submodule.span K s ≤ E ↔ s ⊆ E :=
  Submodule.span_le

example : GaloisInsertion (Submodule.span K) ((↑) : Submodule K V → Set V) :=
  Submodule.gi K V
```

当上述方法不够时，可以使用相关的归纳原理 *Submodule.span_induction*，它保证只要某性质对零元和集合 *s* 中的元素成立，且在加法和标量乘法下封闭，那么该性质对 *s* 的张成子空间中的每个元素都成立。

作为练习，我们重新证明 *Submodule.mem_sup* 的一个推论。记住你可以使用 *module* 策略，来解决基于 *V* 上各种代数运算公理的证明目标。

```
example {S T : Submodule K V} {x : V} (h : x ∈ S ⊔ T) :
    ∃ s ∈ S, ∃ t ∈ T, x = s + t  := by
  rw [← S.span_eq, ← T.span_eq, ← Submodule.span_union] at h
  induction h using Submodule.span_induction with
  | mem y h =>
      sorry
  | zero =>
      sorry
  | add x y hx hy hx' hy' =>
      sorry
  | smul a x hx hx' =>
      sorry
```

9.2.4 拉回和推出子空间

如前所述，我们现在描述如何通过线性映射来拉回和推出子空间。在 Mathlib 中，第一个操作称为 `map`，第二个操作称为 `comap`。

```
section

variable {W : Type*} [AddCommGroup W] [Module K W] (φ : V →ₗ[K] W)

variable (E : Submodule K V) in
#check (Submodule.map φ E : Submodule K W)

variable (F : Submodule K W) in
#check (Submodule.comap φ F : Submodule K V)
```

这些函数定义在 *Submodule* 命名空间内，因此可以使用点记法写作 `E.map φ` 而不是 ``Submodule.map φ E``。不过这种写法阅读起来相当别扭（虽然有些 Mathlib 贡献者会这样写）。特别地，线性映射的像集和核都是子空间，这些特殊情况重要到有专门的声明支持。

```
example : LinearMap.range φ = .map φ ⊤ := LinearMap.range_eq_map φ

example : LinearMap.ker φ = .comap φ ⊥ := Submodule.comap_bot φ -- or `rfl`
```

请注意，我们不能写成 *φ.ker* 来替代 *LinearMap.ker φ*，因为 *LinearMap.ker* 不仅适用于线性映射，还适用于保持更多结构的映射类，因此它不期望接收类型以 *LinearMap* 开头的参数，所以点记法在这里无法使用。然而，在右侧我们能够使用另一种点记法。因为 Lean 在展开左侧后，期望得到类型为 *Submodule K V* 的项，因此它会将 *.comap* 解析为 *Submodule.comap*。

以下引理描述了这些子模与线性映射 φ 的关键关系。

```
open Function LinearMap

example : Injective φ ↔ ker φ = ⊥ := ker_eq_bot.symm

example : Surjective φ ↔ range φ = ⊤ := range_eq_top.symm
```

作为练习，我们来证明 `map` 和 `comap` 的 Galois 连接性质。可以使用以下引理，但这不是必需的，因为它们是根据定义成立的。

```
#check Submodule.mem_map_of_mem
#check Submodule.mem_map
#check Submodule.mem_comap

example (E : Submodule K V) (F : Submodule K W) :
    Submodule.map φ E ≤ F ↔ E ≤ Submodule.comap φ F := by
  sorry
```

9.2.5 商空间

商向量空间使用通用的商记号（输入为 *quot*，而非普通的 /）。投影到商空间的映射是 *Submodule.mkQ*，其万有性质由 *Submodule.liftQ* 给出。

```
variable (E : Submodule K V)

example : Module K (V ⧸ E) := inferInstance

example : V →ₗ[K] V ⧸ E := E.mkQ

example : ker E.mkQ = E := E.ker_mkQ

example : range E.mkQ = ⊤ := E.range_mkQ

example (hφ : E ≤ ker φ) : V ⧸ E →ₗ[K] W := E.liftQ φ hφ

example (F : Submodule K W) (hφ : E ≤ .comap φ F) : V ⧸ E →ₗ[K] W ⧸ F := E.mapQ F φ hφ

noncomputable example : (V ⧸ LinearMap.ker φ) ≃ₗ[K] range φ := φ.quotKerEquivRange
```

作为练习，我们来证明商空间子空间的对应定理。Mathlib 中有一个稍微更精确的版本，称为 `Submodule.comapMkQRelIso`。

```
open Submodule

#check Submodule.map_comap_eq
#check Submodule.comap_map_eq

example : Submodule K (V ⧸ E) ≃ { F : Submodule K V // E ≤ F } where
  toFun := sorry
  invFun := sorry
  left_inv := sorry
  right_inv := sorry
```

9.3 自同态

线性映射中的一个重要特例是自同态（endomorphisms）：即从向量空间自身映射到自身的线性映射。自同态特别有趣，因为它们构成了一个 K-代数。具体来说，我们可以在其上对系数属于 K 的多项式进行求值，它们也可能具有特征值和特征向量。

Mathlib 使用简称 `Module.End K V := V →ₗ[K] V`，这在大量使用此类映射时非常方便，尤其是在打开了 *Module* 命名空间后。

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]

variable {W : Type*} [AddCommGroup W] [Module K W]


open Polynomial Module LinearMap

example (φ ψ : End K V) : φ * ψ = φ ∘ₗ ψ :=
  LinearMap.mul_eq_comp φ ψ -- `rfl` 也可以

-- evaluating `P` on `φ`
example (P : K[X]) (φ : End K V) : V →ₗ[K] V :=
  aeval φ P

-- evaluating `X` on `φ` gives back `φ`
example (φ : End K V) : aeval φ (X : K[X]) = φ :=
  aeval_X φ
```

作为练习，结合自同态、子空间和多项式的操作，让我们证明二元核引理：对于任意自同态 及互素多项式 P 和 Q，有 $\ker P() \oplus \ker Q() = \ker \left(PQ()\right)$。

注意，*IsCoprime x y* 的定义为 `∃ a b, a * x + b * y = 1` 。

```
#check Submodule.eq_bot_iff
#check Submodule.mem_inf
#check LinearMap.mem_ker

example (P Q : K[X]) (h : IsCoprime P Q) (φ : End K V) : ker (aeval φ P) ⊓ ker (aeval␣
↪φ Q) = ⊥ := by
  sorry

#check Submodule.add_mem_sup
#check map_mul
#check LinearMap.mul_apply
#check LinearMap.ker_le_ker_comp

example (P Q : K[X]) (h : IsCoprime P Q) (φ : End K V) :
    ker (aeval φ P) ⊔ ker (aeval φ Q) = ker (aeval φ (P*Q)) := by
  sorry
```

我们现在转向本征空间和特征值的讨论。对于自同态 和标量 a ，与之对应的本征空间是 $-aId$ 的核空间。本征空间对所有 a 都有定义，但只有在本征空间非零时才有意义。然而，本征向量定义为本征空间中的非零元素。对应的谓词是 *End.HasEigenvector*。

```
example (φ : End K V) (a : K) : φ.eigenspace a = LinearMap.ker (φ - a • 1) :=
  End.eigenspace_def
```

然后有一个谓词 `End.HasEigenvalue` 和对应的子类型 `End.Eigenvalues`。

```
example (φ : End K V) (a : K) : φ.HasEigenvalue a ↔ φ.eigenspace a ≠ ⊥ :=
  Iff.rfl

example (φ : End K V) (a : K) : φ.HasEigenvalue a ↔ ∃ v, φ.HasEigenvector a v  :=
  ⟨End.HasEigenvalue.exists_hasEigenvector, fun ⟨_, hv⟩ ↦ φ.hasEigenvalue_of_
↪hasEigenvector hv⟩

example (φ : End K V) : φ.Eigenvalues = {a // φ.HasEigenvalue a} :=
  rfl

-- 特征值是最小多项式的根
example (φ : End K V) (a : K) : φ.HasEigenvalue a → (minpoly K φ).IsRoot a :=
  φ.isRoot_of_hasEigenvalue

-- 有限维情况下，反之亦然（我们将在下面讨论维度）
```

(continues on next page)

(continued from previous page)

```
example [FiniteDimensional K V] (φ : End K V) (a : K) :
    φ.HasEigenvalue a ↔ (minpoly K φ).IsRoot a :=
  φ.hasEigenvalue_iff_isRoot

-- Cayley-Hamilton
example [FiniteDimensional K V] (φ : End K V) : aeval φ φ.charpoly = 0 :=
  φ.aeval_self_charpoly
```

9.4 矩阵、基和维度

9.4.1 矩阵

在介绍抽象向量空间的基之前，我们先回到更基础的线性代数设置——定义在某域 K 上的 K^n 。这里的主要对象是向量和矩阵。对于具体向量，可以使用 *![···]* 记法，分量之间用逗号分隔。对于具体矩阵，可以使用 *!![···]* 记法，行之间用分号分隔，行内分量用冒号分隔。当矩阵元素类型是可计算类型（例如 $\mathbb{N}$ 或 $\mathbb{Q}$）时，可以使用 *eval* 命令来进行基本运算操作的试验。

```
section matrices

-- Adding vectors
#eval ![1, 2] + ![3, 4]  -- ![4, 6]

-- Adding matrices
#eval !![1, 2; 3, 4] + !![3, 4; 5, 6]  -- !![4, 6; 8, 10]

-- Multiplying matrices
#eval !![1, 2; 3, 4] * !![3, 4; 5, 6]  -- !![4, 6; 8, 10]
```

需要理解的是，*#eval* 的使用主要用于探索和试验，并不旨在替代如 Sage 这类计算机代数系统。这里矩阵的数据表示方式 *并不*在计算上高效，它采用函数而非数组，优化目标是证明而非计算。而且，*#eval* 所用的虚拟机也未针对这种用途进行优化。

请注意，矩阵表示中列出了多行，而向量表示既不是行向量也不是列向量。矩阵与向量的乘法，若从左乘，则将向量视为行向量；若从右乘，则将向量视为列向量。

对应的操作为：

- *Matrix.vecMul*，符号为 *⬝ᵥ*
- *Matrix.mulVec*，符号为 *ₘ*

这些符号限定在 *Matrix* 命名空间，因此我们需要打开该命名空间才能使用。

```
open Matrix

-- 矩阵左作用于向量
#eval !![1, 2; 3, 4] *ᵥ ![1, 1] -- ![3, 7]

-- 矩阵左作用于向量，结果为一维矩阵
#eval !![1, 2] *ᵥ ![1, 1]  -- ![3]

-- 矩阵右作用于向量
#eval  ![1, 1, 1] ᵥ* !![1, 2; 3, 4; 5, 6] -- ![9, 12]
```

为了生成由某向量指定的相同行或列的矩阵，我们使用 *Matrix.row* 和 *Matrix.column*，它们的参数分别是用于索引行或列的类型以及该向量。例如，可以得到单行矩阵或单列矩阵（更准确地说，是行或列由 *Fin 1* 索引的矩阵）。

```
#eval row (Fin 1) ![1, 2] -- !![1, 2]

#eval col (Fin 1) ![1, 2] -- !![1; 2]
```

其它熟悉的操作包括向量点积、矩阵转置，以及对于方阵的行列式和迹。

```
-- 向量点积
#eval ![1, 2] ⬝ᵥ ![3, 4] -- `11`

-- 矩阵转置
#eval !![1, 2; 3, 4]ᵀ -- `!![1, 3; 2, 4]`

-- 行列式
#eval !![(1 : ℤ), 2; 3, 4].det -- `-2`

-- 迹
#eval !![(1 : ℤ), 2; 3, 4].trace -- `5`
```

当矩阵元素类型不可计算时，比如实数，*#eval* 就无能为力了。而且，这类计算也不能直接用于证明中，否则需要大幅扩展受信任代码库（即在检验证明时必须信任的 Lean 核心部分）。

因此，在证明中推荐使用 *simp* 和 *norm_num* 策略，或者它们对应的命令形式来进行快速探索和简化。

```
#simp !![(1 : ℝ), 2; 3, 4].det -- `4 - 2*3`

#norm_num !![(1 : ℝ), 2; 3, 4].det -- `-2`
```

(continues on next page)

(continued from previous page)

```
#norm_num !![(1 : ℝ), 2; 3, 4].trace -- `5`

variable (a b c d : ℝ) in
#simp !![a, b; c, d].det -- `a * d - b * c`
```

下一个关于方阵的重要操作是求逆。类似于数的除法总是定义的，且对除以零的情况返回人为设定的零值，矩阵求逆操作也定义在所有矩阵上，对于不可逆矩阵返回零矩阵。

更具体地，存在一个通用函数 *Ring.inverse*，它在任意环中实现此功能；对于任意矩阵 A，其逆矩阵定义为 `Ring.inverse A.det • A.adjugate` 根据 Cramer’s rule，当矩阵行列式不为零时，上述定义确实是矩阵 A 的逆矩阵。

```
#norm_num [Matrix.inv_def] !![(1 : ℝ), 2; 3, 4]⁻¹ -- !![-2, 1; 3 / 2, -(1 / 2)]
```

当然这种定义确实只对可逆矩阵有用。存在一个通用类型类 `Invertible` 来帮助记录这一点。例如，下面例子中的 `simp` 调用将使用 `inv_mul_of_invertible` 引理，该引理具有 `Invertible` 类型类假设，因此只有在类型类合成系统能够找到它时才会触发。在这里，我们使用 `have` 语句使这一事实可用。

```
example : !![(1 : ℝ), 2; 3, 4]⁻¹ * !![(1 : ℝ), 2; 3, 4] = 1 := by
  have : Invertible !![(1 : ℝ), 2; 3, 4] := by
    apply Matrix.invertibleOfIsUnitDet
    norm_num
  simp
```

在这个完全具体的例子中，我们也可以使用 `norm_num` 工具，和 `apply?` 来找到最后一行：

```
example : !![(1 : ℝ), 2; 3, 4]⁻¹ * !![(1 : ℝ), 2; 3, 4] = 1 := by
  norm_num [Matrix.inv_def]
  exact one_fin_two.symm
```

之前所有具体矩阵的行和列均由某个 *Fin n* 索引（行列的 n 不一定相同）。但有时使用任意有限类型作为矩阵的行列索引更为方便。例如，有限图的邻接矩阵，其行和列自然由图的顶点索引。

实际上，如果仅定义矩阵而不涉及运算，索引类型的有限性甚至不是必须的，矩阵元素的类型也无需具备任何代数结构。因此，Mathlib 直接将 *Matrix m n α* 定义为 $m \to n \to \alpha$，其中 m、n、α 是任意类型。我们之前使用的矩阵类型多如 *Matrix (Fin 2) (Fin 2) ℝ*。当然，代数运算对 m、n 和 α 有更严格的假设。

我们不直接使用 $m \to n \to \alpha$ 的主要原因，是让类型类系统能正确理解我们的意图。举例来说，对于环 R，类型 $n \to R$ 自带逐点乘法（point-wise multiplication），而 $m \to n \to R$ 也有类似操作，但这并不是我们想在矩阵上使用的乘法。

下面第一个示例中，我们强制 Lean 展开 *Matrix* 定义，接受陈述的意义，并通过逐个元素检查完成证明。

而后面两个示例展示了 Lean 对 *Fin 2 → Fin 2 → ℤ* 使用逐点乘法，而对 *Matrix (Fin 2) (Fin 2) ℤ* 使用矩阵乘法。

```
section

example : (fun _ ↦ 1 : Fin 2 → Fin 2 → ℤ) = !![1, 1; 1, 1] := by
  ext i j
  fin_cases i <;> fin_cases j <;> rfl

example : (fun _ ↦ 1 : Fin 2 → Fin 2 → ℤ) * (fun _ ↦ 1 : Fin 2 → Fin 2 → ℤ) = !![1, 1;
↪ 1, 1] := by
  ext i j
  fin_cases i <;> fin_cases j <;> rfl

example : !![1, 1; 1, 1] * !![1, 1; 1, 1] = !![2, 2; 2, 2] := by
  norm_num
```

为了定义矩阵作为函数而不失去 `Matrix` 在类型类合成中的好处，我们可以使用函数和矩阵之间的等价关系 `Matrix.of`。这个等价关系是通过 `Equiv.refl` 隐式定义的。

例如，我们可以定义与向量 `v` 对应的范德蒙德矩阵。

```
example {n : ℕ} (v : Fin n → ℝ) :
    Matrix.vandermonde v = Matrix.of (fun i j : Fin n ↦ v i ^ (j : ℕ)) :=
  rfl
end
end matrices
```

9.4.2 基

我们现在讨论向量空间的基。非正式地说，这一概念有多种定义方式：

- 可以通过万有性质来定义；
- 可以说基是一族线性无关且张成全空间的向量；
- 或结合这两个性质，直接说基是一族向量，使得每个向量都能唯一地表示为基向量的线性组合；
- 另一种说法是，基提供了与基域 `K` 的某个幂（作为 `K` 上的向量空间）的线性同构。

实际上，Mathlib 在底层采用的是最后一种同构的定义，并从中推导出其他性质。对于无限基的情况，需稍加注意“基域的幂”这一说法。因为在代数环境中，只考虑有限线性组合有意义，所以我们参考的向量空间不是基域的直积，而是直和。我们可以用 $\square i : \iota, K$ 表示某个类型 ι 索引的基向量集合对应的直和，但更常用的是更专门的表示法 $\iota \to_0 K$，表示“具有有限支集的函数”，即在 ι 上除有限点外值为零的函数（这个有限集合不是固定的，依赖于函数本身）。

对于基 B 中的函数，给定向量 v 和索引 $i : \iota$，函数的值即为 v 在第 i 个基向量上的坐标分量。

以类型 ι 索引的向量空间 V 的基类型为 $Basis\ \iota\ K\ V$，对应的同构称为 $Basis.repr$。

```
variable {K : Type*} [Field K] {V : Type*} [AddCommGroup V] [Module K V]

section

variable {ι : Type*} (B : Basis ι K V) (v : V) (i : ι)

-- 索引为 ``i`` 的基向量
#check (B i : V)

-- 与模空间 ``B`` 给出的线性同构
#check (B.repr : V ≃[K] ι →₀ K)

-- ``v`` 的分量函数
#check (B.repr v : ι →₀ K)

-- ``v`` 在索引为 ``i`` 的基向量上的分量
#check (B.repr v i : K)
```

与其从同构出发，也可以从一族线性无关且张成的向量族 b 开始构造基，这就是 *Basis.mk*。“张成”的假设表达为 `⊤ ≤ Submodule.span K (Set.range b)`，这里的 `⊤` 是 V 的最大子模，即 V 自身作为自身的子模。这个表达看起来有些拗口，但下面我们将看到它与更易理解的表达式 `∀ v, v ∈ Submodule.span K (Set.range b)` 几乎是定义等价的（下面代码中的下划线表示无用信息 $v \in \top$）。

```
noncomputable example (b : ι → V) (b_indep : LinearIndependent K b)
    (b_spans : ∀ v, v ∈ Submodule.span K (Set.range b)) : Basis ι K V :=
  Basis.mk b_indep (fun v _ ↦ b_spans v)

-- 该基的底层向量族确实是 ``b``。
example (b : ι → V) (b_indep : LinearIndependent K b)
    (b_spans : ∀ v, v ∈ Submodule.span K (Set.range b)) (i : ι) :
    Basis.mk b_indep (fun v _ ↦ b_spans v) i = b i :=
  Basis.mk_apply b_indep (fun v _ ↦ b_spans v) i
```

特别地，模型向量空间 $\iota \to_0 K$ 具有所谓的标准基（canonical basis），其 *repr* 函数在任意向量上的值即为恒等同构。该基称为 *Finsupp.basisSingleOne*，其中 *Finsupp* 表示有限支集函数，*basisSingleOne* 指的是基向量是除单个输入点外，其余点值皆为零的函数。更具体地，索引为 $i : \iota$ 的基向量是 `Finsupp.single i 1`，这是一个有限支集函数，在点 i 取值为 *1*，其余点取值为 *0*。

```
variable [DecidableEq ι]

example : Finsupp.basisSingleOne.repr = LinearEquiv.refl K (ι →₀ K) :=
```

(continues on next page)

(continued from previous page)

```
  rfl

example (i : ι) : Finsupp.basisSingleOne i = Finsupp.single i 1 :=
  rfl
```

当索引类型是有限集合时，有限支集函数的概念就不再需要。这时，我们可以使用更简单的 *Pi.basisFun*，它直接构造了整个 $\iota \to K$ 的一组基。

```
example [Finite ι] (x : ι → K) (i : ι) : (Pi.basisFun K ι).repr x i = x i := by
  simp
```

回到抽象向量空间基的通用情况，我们可以将任何向量表示为基向量的线性组合。让我们首先看看有限基的简单情况。

```
example [Fintype ι] : ∑ i : ι, B.repr v i • (B i) = v :=
  B.sum_repr v
```

当索引类型 ι 不是有限集时，上述直接对 ι 求和的说法在理论上没有意义，因为无法对无限集合进行直接求和。不过，被求和的函数的支集是有限的（即 *B.repr v* 的支集），这就允许对有限支集进行求和。为了处理这种情况，Mathlib 使用了一个特殊的函数，虽然需要一些时间适应，它是 *Finsupp.linearCombination*`（建立在更通用的 `*Finsupp.sum* 之上）。

给定一个从类型 ι 到基域 K 的有限支集函数 c，以及任意从 ι 到向量空间 V 的函数 f，*Finsupp.linearCombination* K f c 定义为对 c 的支集上所有元素的 $c \bullet f$ 进行标量乘法后求和。特别地，我们也可以将求和范围替换为任何包含 c 支集的有限集合。

When `ι` is not finite, the above statement makes no sense a priori: we cannot take a sum over `ι`. However the support of the function being summed is finite (it is the support of `B.repr v`). But we need to apply a construction that takes this into account. Here Mathlib uses a special purpose function that requires some time to get used to: `Finsupp.linearCombination` (which is built on top of the more general `Finsupp.sum`). Given a finitely supported function `c` from a type `ι` to the base field `K` and any function `f` from `ι` to `V`, `Finsupp.linearCombination K f c` is the sum over the support of `c` of the scalar multiplication `c • f`. In particular, we can replace it by a sum over any finite set containing the support of `c`.

```
example (c : ι →₀ K) (f : ι → V) (s : Finset ι) (h : c.support ⊆ s) :
    Finsupp.linearCombination K f c = ∑ i ∈ s, c i • f i :=
  Finsupp.linearCombination_apply_of_mem_supported K h
```

也可以假设 f 是有限支集函数，这样依然能得到良定义的求和结果。但 *Finsupp.linearCombination* 所作的选择更贴合我们关于基的讨论，因为它支持陈述 *Basis.sum_repr* 的推广版本。

```
example : Finsupp.linearCombination K B (B.repr v) = v :=
  B.linearCombination_repr v
```

你可能会好奇，为什么这里的 K 是显式参数，尽管它可以从 c 的类型中推断出来。关键在于，部分应用的 *Finsupp.linearCombination K f* 本身就是一个有趣的对象。它不仅仅是一个从 $\iota \to_0 K$ 到 V 的普通函数，更是一个 K-线性映射。

```
variable (f : ι → V) in
#check (Finsupp.linearCombination K f : (ι →₀ K) →ₗ[K] V)
```

上述细节也解释了为什么不能用点记法写成 *c.linearCombination K f* 来代替 *Finsupp.linearCombination K f c*。事实上，*Finsupp.linearCombination* 本身并不直接接收 c 作为参数，而是先部分应用 K 和 f，得到一个被强制转换为函数类型的对象，随后该函数才接收 c 作为参数。

回到数学讨论，理解基下向量的具体表示在形式化数学中往往没有你想象的那么有用非常重要。实际上，直接利用基的更抽象性质通常更加高效。特别是，基的通用性质将其与代数中的其他自由对象连接起来，使得我们可以通过指定基向量的像来构造线性映射。这就是 *Basis.constr*。对于任意 K-向量空间 W，我们的基 B 提供了一个线性同构 *Basis.constr B K : (ι → W) ≃ₗ[K] (V →ₗ[K] W)*。该同构的特点是，对于任意函数 $u : \iota \to W$，它对应的线性映射将基向量 $B\, i$ 映射为 $u\, i$，其中 $i : \iota$。

```
section

variable {W : Type*} [AddCommGroup W] [Module K W]
         (ψ : V →ₗ[K] W) (u : ι → W)

#check (B.constr K : (ι → W) ≃ₗ[K] (V →ₗ[K] W))

#check (B.constr K u : V →ₗ[K] W)

example (i : ι) : B.constr K u (B i) = u i :=
  B.constr_basis K u i
```

这个性质实际上是特征性的，因为线性映射是由它们在基上的值决定的：

```
example (φ ψ : V →ₗ[K] W) (h : ∀ i, φ (B i) = ψ (B i)) : φ = ψ :=
  B.ext h
```

如果我们在目标空间上也有一个基 `B'`，那么我们可以将线性映射与矩阵进行识别。这种识别是一个 `K`-线性同构。

```
variable {ι' : Type*} (B' : Basis ι' K W) [Fintype ι] [DecidableEq ι] [Fintype ι']␣
↪[DecidableEq ι']

open LinearMap

#check (toMatrix B B' : (V →ₗ[K] W) ≃ₗ[K] Matrix ι' ι K)
```

(continues on next page)

(continued from previous page)

```
open Matrix -- 获取对矩阵和向量之间乘法的 ``*ᵥ`` 记法的访问。

example (φ : V →ₗ[K] W) (v : V) : (toMatrix B B' φ) *ᵥ (B.repr v) = B'.repr (φ v) :=
  toMatrix_mulVec_repr B B' φ v


variable {ι'' : Type*} (B'' : Basis ι'' K W) [Fintype ι''] [DecidableEq ι'']

example (φ : V →ₗ[K] W) : (toMatrix B B'' φ) = (toMatrix B' B'' .id) * (toMatrix B B'
↪φ) := by
  simp

end
```

作为本主题的练习，我们将证明一部分定理，保证自同态的行列式是良定义的。具体来说，我们要证明：当两个基由相同类型索引时，它们对应任意自同态的矩阵具有相同的行列式。然后，结合所有基的索引类型彼此线性同构，便可得到完整结论。

当然，Mathlib 已经包含了这一结论，且 *simp* 策略可以立即解决此目标，因此你不应过早使用它，而应先尝试使用相关引理进行证明。

```
open Module LinearMap Matrix

-- 一些来自于 `LinearMap.toMatrix` 是一个代数同态的事实的引理。
#check toMatrix_comp
#check id_comp
#check comp_id
#check toMatrix_id

-- 一些来自于 ``Matrix.det`` 是一个乘法单元环同态的事实的引理。
#check Matrix.det_mul
#check Matrix.det_one

example [Fintype ι] (B' : Basis ι K V) (φ : End K V) :
    (toMatrix B B φ).det = (toMatrix B' B' φ).det := by
  set M := toMatrix B B φ
  set M' := toMatrix B' B' φ
  set P := (toMatrix B B') LinearMap.id
  set P' := (toMatrix B' B) LinearMap.id
  sorry
```

(continues on next page)

(continued from previous page)

```
end
```

9.4.3 维度

回到单个向量空间的情况，基也用于定义维度的概念。在这里，我们再次遇到有限维向量空间的基本情况。对于这样的空间，我们期望维度是一个自然数。这就是 `Module.finrank`。它将基域作为显式参数，因为给定的阿贝尔群可以是不同域上的向量空间。

```
section
#check (Module.finrank K V : ℕ)

-- `Fin n → K` 是维度为 `n` 的典型空间。
example (n : ℕ) : Module.finrank K (Fin n → K) = n :=
  Module.finrank_fin_fun K

-- 作为自身的向量空间，`ℂ` 的维度为 1。
example : Module.finrank ℂ ℂ = 1 :=
  Module.finrank_self ℂ

-- 但作为实向量空间，它的维度为 2。
example : Module.finrank ℝ ℂ = 2 :=
  Complex.finrank_real_complex
```

注意到 `Module.finrank` 是为任何向量空间定义的。它对于无限维向量空间返回零，就像除以零返回零一样。

当然，许多引理都需要有限维度的假设。这就是 `FiniteDimensional` 类型类的作用。例如，考虑下一个例子在没有这个假设的情况下是如何失败的。

```
example [FiniteDimensional K V] : 0 < Module.finrank K V ↔ Nontrivial V  :=
  Module.finrank_pos_iff
```

在上述陈述中，*Nontrivial V* 表示 *V* 至少包含两个不同元素。注意，*Module.finrank_pos_iff* 没有显式参数。从左向右使用时没问题，但从右向左使用时会遇到困难，因为 Lean 无法从 *Nontrivial V* 这一命题中推断出域 *K*。此时，使用命名参数语法会很有帮助，前提是确认该引理是在一个名为 *R* 的环上陈述的。因此我们可以写成：

```
example [FiniteDimensional K V] (h : 0 < Module.finrank K V) : Nontrivial V := by
  apply (Module.finrank_pos_iff (R := K)).1
  exact h
```

上述写法看起来有些奇怪，因为我们已经有了假设 *h*，因此完全可以直接写成 *Module.finrank_pos_iff.1 h* 来使用该引理。不过，了解命名参数的用法对于更复杂的情况仍然很有帮助。

根据定义，*FiniteDimensional K V* 可以通过任意一组基来判定。

```
variable {ι : Type*} (B : Basis ι K V)

example [Finite ι] : FiniteDimensional K V := FiniteDimensional.of_fintype_basis B

example [FiniteDimensional K V] : Finite ι :=
  (FiniteDimensional.fintypeBasisIndex B).finite
end
```

使用与线性子空间对应的子类型具有向量空间结构，我们可以讨论子空间的维度。

```
section
variable (E F : Submodule K V) [FiniteDimensional K V]

open Module

example : finrank K (E ⊔ F : Submodule K V) + finrank K (E ⊓ F : Submodule K V) =
    finrank K E + finrank K F :=
  Submodule.finrank_sup_add_finrank_inf_eq E F

example : finrank K E ≤ finrank K V := Submodule.finrank_le E
```

在第一条语句中，类型注释的目的是确保对 `Type*` 的强制转换不会过早触发。

我们现在准备进行一个关于 `finrank` 和子空间的练习。

```
example (h : finrank K V < finrank K E + finrank K F) :
    Nontrivial (E ⊓ F : Submodule K V) := by
  sorry
end
```

我们现在转向一般的维数理论情形。此时，*finrank* 就不再适用，但我们依然知道，对于同一向量空间的任意两组基，其索引类型之间存在一个双射。因此，我们仍然可以期望将秩定义为基数（cardinal），即“类型集合在存在双射的等价关系下的商集”中的元素。

在讨论基数时，难以像本书其他部分那样忽略类似罗素悖论的基础性问题。因为不存在“所有类型的类型”，否则会导致逻辑矛盾。这一问题通过宇宙层级（universe hierarchy）来解决，而我们通常会尝试忽略这些细节。

每个类型都有一个宇宙层级，该层级行为类似自然数。特别地，有一个零层级，对应的宇宙 *Type 0* 简写为 *Type*，这个宇宙足以容纳几乎所有经典数学对象，比如 $\mathbb{N}$ 和 $\mathbb{R}$ 都属于类型 *Type*。每个层级 *u* 有一个后继层级 *u + 1*，且 *Type u* 的类型是 *Type (u + 1)*。

但宇宙层级不是自然数，它们本质不同且没有对应的类型。例如，无法在 Lean 中声明 $u \neq u + 1$，因为不存在一个类型可以表达这个命题。即使声明 *Type u* ≠ *Type (u+1)* 也没有意义，因为两者属于不同的类型。

当我们写 *Type** 时，Lean 会自动插入一个名为 *u_n* 的宇宙变量，其中 *n* 是一个数字，允许定义和陈述在所有

宇宙层级中成立。

给定宇宙层级 *u*，我们可以在 *Type u* 上定义等价关系：当且仅当两个类型 α 和 β 存在双射时它们等价。商类型 *Cardinal.{u}* 属于 *Type (u + 1)*，花括号表示这是一个宇宙变量。类型 *α : Type u* 在商中的像为 *Cardinal.mk α : Cardinal.{u}*。

但我们不能直接比较不同宇宙层级中的基数。因此技术上讲，不能将向量空间 *V* 的秩定义为索引 *V* 基的所有类型的基数。相反，秩定义为 *Module.rank K V*，即 *V* 中所有线性无关集基数的上确界。如果 *V* 属于宇宙层级 *u*，则其秩属于 *Cardinal.{u}* 类型。

```
#check V -- Type u_2
#check Module.rank K V -- Cardinal.{u_2}
```

这个定义仍然可以与基联系起来。实际上，宇宙层级上存在一个交换的 *max* 运算，对于任意两个宇宙层级 *u* 和 *v*，存在一个操作 *Cardinal.lift.{u, v} : Cardinal.{v} → Cardinal.{max v u}*，该操作允许将基数提升到共同的宇宙层级中，从而陈述维数定理。

```
universe u v -- `u` 和 `v` 将表示宇宙层级

variable {ι : Type u} (B : Basis ι K V)
         {ι' : Type v} (B' : Basis ι' K V)

example : Cardinal.lift.{v, u} (.mk ι) = Cardinal.lift.{u, v} (.mk ι') :=
  mk_eq_mk_of_basis B B'
```

我们可以将有限维情况与此讨论联系起来，使用从自然数到有限基数的强制转换（更准确地说，是生活在 `Cardinal.{v}` 中的有限基数，其中 `v` 是 `V` 的宇宙层级）。

```
example [FiniteDimensional K V] :
    (Module.finrank K V : Cardinal) = Module.rank K V :=
  Module.finrank_eq_rank K V
```

CHAPTER

TEN

拓扑学

微积分建立在函数的概念之上，旨在对相互依赖的量进行建模。例如，研究随时间变化的量是一个常见的应用场景。**极限（limit）**这一概念同样基础。我们可以说，当 x 趋近于某个值 a 时，函数 $f(x)$ 的极限是一个值 b，或者说 $f(x)$ 当 x 趋近于 a 时 **收敛于（converges to）** b。等价地，我们可以说当 x 趋近于某个值 a 时，$f(x)$ 趋近于 b，或者说它当 x 趋向于 a 时 **趋向于（tends to）** b。我们在 Section 3.6 中已开始考虑此类概念。

拓扑学是对极限和连续性的抽象研究。在第 2 章到第 6 章中，我们已经介绍了形式化的基础知识，在本章中，我们将解释在 Mathlib 中如何形式化拓扑概念。拓扑抽象不仅适用范围更广，而且有些矛盾的是，这也使得在具体实例中推理极限和连续性变得更加容易。

拓扑概念建立在多层数学结构之上。第一层是朴素集合论，如第 Chapter 4 所述。接下来的一层是 **滤子**理论，我们将在 Section 10.1 中进行描述。在此之上，我们再叠加 **拓扑空间**、**度量空间**以及一种稍显奇特的中间概念——**一致空间**的理论。

虽然前面的章节所依赖的数学概念您可能已经熟悉，但“滤子”这一概念对您来说可能不太熟悉，甚至对许多从事数学工作的人员来说也是如此。然而，这一概念对于有效地形式化数学来说是必不可少的。让我们解释一下原因。设 `f : ℝ → ℝ` 为任意函数。我们可以考虑当 `x` 趋近于某个值 x_0 时 `f x` 的极限，但也可以考虑当 `x` 趋近于正无穷或负无穷时 `f x` 的极限。此外，我们还可以考虑当 `x` 从右趋近于 x_0（通常记为 x_0^+）或从左趋近于 x_0（记为 x_0^-）时 `f x` 的极限。还有些变化情况是 `x` 趋近于 x_0 或 x_0^+ 或 x_0^-，但不允许取值为 x_0 本身。这导致了至少八种 `x` 趋近于某个值的方式。我们还可以限制 `x` 为有理数，或者对定义域施加其他约束，但让我们先只考虑这八种情况。

在值域方面，我们也有类似的多种选择：我们可以指定 `f x` 从左侧或右侧趋近某个值，或者趋近正无穷或负无穷等等。例如，我们可能希望说当 `x` 从右侧趋近 x_0 但不等于 x_0 时，`f x` 趋近于 $+\infty$。这会产生 64 种不同的极限表述，而且我们甚至还没有开始处理序列的极限，就像我们在 Section 3.6 中所做的那样。

当涉及到辅助引理时，问题变得更加复杂。例如，极限的复合：如果当 `x` 趋近于 x_0 时，`f x` 趋近于 y_0，且当 `y` 趋近于 y_0 时，`g y` 趋近于 z_0，那么当 `x` 趋近于 x_0 时，`g ∘ f x` 趋近于 z_0。这里涉及三种“趋近于”的概念，每种概念都可以按照上一段描述的八种方式中的任何一种来实例化。而这会产生 512 个引理，要添加到库中实在太多了！非正式地说，数学家通常只证明其中两三个，然后简单地指出其余的可以“以相同方式”进行证明。形式化数学需要明确相关“相同”的概念，而这正是布尔巴基（Bourbaki）的滤子理论所做到的。

10.1 滤子

类型 X 上的 **滤子**是 X 的集合的集合，满足以下三个条件（我们将在下面详细说明）。该概念支持两个相关的想法：

- **极限**，包括上述讨论过的各种极限：数列的有限极限和无穷极限、函数在某点或无穷远处的有限极限和无穷极限等等。
- **最终发生的事情**，包括对于足够大的自然数 n : ℕ 发生的事情，或者在某一点 x 足够近的地方发生的事情，或者对于足够接近的点对发生的事情，或者在测度论意义上几乎处处发生的事情。对偶地，滤子也可以表达 **经常发生的事情**的概念：对于任意大的 n，在给定点的任意邻域内存在某点发生，等等。

与这些描述相对应的滤子将在本节稍后定义，但我们现在就可以给它们命名：

- (atTop : Filter ℕ)，由包含 {n | n ≥ N} 的 ℕ 的集合构成，其中 N 为某个自然数
- 𝓝 x，由拓扑空间中 x 的邻域构成
- 𝓤 X，由一致空间的邻域基构成（一致空间推广了度量空间和拓扑群）
- μ.ae，由相对于测度 μ 其补集测度为零的集合构成

一般的定义如下：一个滤子 F : Filter X 是集合 F.sets : Set (Set X) 的一个集合，满足以下条件：

- F.univ_sets : univ ∈ F.sets
- F.sets_of_superset : ∀ {U V}, U ∈ F.sets → U ⊆ V → V ∈ F.sets
- F.inter_sets : ∀ {U V}, U ∈ F.sets → V ∈ F.sets → U ∩ V ∈ F.sets

第一个条件表明，集合 X 的所有元素都属于 F.sets。第二个条件表明，如果 U 属于 F.sets，那么包含 U 的任何集合也属于 F.sets。第三个条件表明，F.sets 对有限交集是封闭的。在 Mathlib 中，滤子 F 被定义为捆绑 F.sets 及其三个属性的结构，但这些属性不携带额外的数据，并且将 F 和 F.sets 之间的区别模糊化是方便的。我们因此，将 U ∈ F 定义为 U ∈ F.sets。这就解释了为什么在一些提及 U ∈ F 的引理名称中会出现 sets 这个词。

可以将滤子视为定义“足够大”集合的概念。第一个条件表明 univ 是足够大的集合，第二个条件表明包含足够大集合的集合也是足够大的集合，第三个条件表明两个足够大集合的交集也是足够大的集合。

将类型 X 上的一个滤子视为 Set X 的广义元素，可能更有用。例如，atTop 是“极大数的集合”，而 𝓝 x。是“非常接近 x。的点的集合”。这种观点的一种体现是，我们可以将任何 s : Set X 与所谓的“主滤子”相关联，该主滤子由包含 s 的所有集合组成。此定义已在 Mathlib 中，并有一个记号 𝓟（在 Filter 命名空间中本地化）。为了演示的目的，我们请您借此机会在此处推导出该定义。

```
def principal {α : Type*} (s : Set α) : Filter α
    where
  sets := { t | s ⊆ t }
  univ_sets := sorry
  sets_of_superset := sorry
  inter_sets := sorry
```

对于我们的第二个示例，我们请您定义滤子 `atTop : Filter ℕ`（我们也可以使用任何具有预序关系的类型来代替 `ℕ`）

```
example : Filter ℕ :=
  { sets := { s | ∃ a, ∀ b, a ≤ b → b ∈ s }
    univ_sets := sorry
    sets_of_superset := sorry
    inter_sets := sorry }
```

我们还可以直接定义任意实数 `x : ℝ` 的邻域滤子 `𝓝 x`。在实数中，`x` 的邻域是一个包含开区间 $(x_0 - \varepsilon, x_0 + \varepsilon)$ 的集合，在 Mathlib 中定义为 `Ioo (x₀ - ε) (x₀ + ε)`。(Mathlib 中的这种邻域概念只是更一般构造的一个特例。)

有了这些例子，我们就可以定义函数 `f : X → Y` 沿着某个 `F : Filter X` 收敛到某个 `G : Filter Y` 的含义，如下所述：

```
def Tendsto₁ {X Y : Type*} (f : X → Y) (F : Filter X) (G : Filter Y) :=
  ∀ V ∈ G, f ⁻¹' V ∈ F
```

当 `X` 为 `ℕ` 且 `Y` 为 `ℝ` 时，`Tendsto₁ u atTop (𝓝 x)` 等价于说序列 `u : ℕ → ℝ` 收敛于实数 `x`。当 `X` 和 `Y` 均为 `ℝ` 时，`Tendsto f (𝓝 x₀) (𝓝 y₀)` 等价于熟悉的概念 $\lim_{x \to x_0} f(x) = y_0$。介绍中提到的所有其他类型的极限也等价于对源和目标上适当选择的滤子的 `Tendsto₁` 的实例。

上述的 `Tendsto₁` 概念在定义上等同于在 Mathlib 中定义的 `Tendsto` 概念，但后者定义得更为抽象。`Tendsto₁` 的定义存在的问题是它暴露了量词和 `G` 的元素，并且掩盖了通过将滤子视为广义集合所获得的直观理解。我们可以通过使用更多的代数和集合论工具来隐藏量词 `∀ V`，并使这种直观理解更加突出。第一个要素是与任何映射 `f : X → Y` 相关的 **前推**操作 f_*，在 Mathlib 中记为 `Filter.map f`。给定 `X` 上的滤子 `F`，`Filter.map f F : Filter Y` 被定义为使得 `V ∈ Filter.map f F ↔ f ⁻¹' V ∈ F` 成立。在这个示例文件中，我们已经打开了 `Filter` 命名空间，因此 `Filter.map` 可以写成 `map`。这意味着我们可以使用 `Filter Y` 上的序关系来重写 `Tendsto` 的定义，该序关系是成员集合的反向包含关系。换句话说，给定 `G H : Filter Y`，我们有 `G ≤ H ↔ ∀ V : Set Y, V ∈ H → V ∈ G`。

```
def Tendsto₂ {X Y : Type*} (f : X → Y) (F : Filter X) (G : Filter Y) :=
  map f F ≤ G

example {X Y : Type*} (f : X → Y) (F : Filter X) (G : Filter Y) :
    Tendsto₂ f F G ↔ Tendsto₁ f F G :=
  Iff.rfl
```

可能看起来滤子上的序关系是反向的。但请回想一下，我们可以通过将任何集合 `s` 映射到相应的主滤子的 `𝓟 : Set X → Filter X` 的包含关系，将 `X` 上的滤子视为 `Set X` 的广义元素。这种包含关系是保序的，因此 `Filter` 上的序关系确实可以被视为广义集合之间的自然包含关系。在这个类比中，前推类似于直接像 (direct image)。而且，确实有 `map f (𝓟 s) = 𝓟 (f '' s)`。

现在我们可以直观地理解为什么一个序列 `u : ℕ → ℝ` 收敛于点 x_0 当且仅当 `map u atTop ≤ 𝓝 x₀` 成立。这

个不等式意味着在“u 作用下”的“非常大的自然数集合”的“直接像”包含在“非常接近 x_0 的点的集合”中。

正如所承诺的那样，Tendsto₂ 的定义中没有任何量词或集合。它还利用了前推操作的代数性质。首先，每个 Filter.map f 都是单调的。其次，Filter.map 与复合运算兼容。

```
#check (@Filter.map_mono : ∀ {α β} {m : α → β}, Monotone (map m))

#check
  (@Filter.map_map :
    ∀ {α β γ} {f : Filter α} {m : α → β} {m' : β → γ}, map m' (map m f) = map (m' ∘
↪m) f)
```

这两个性质结合起来使我们能够证明极限的复合性，从而一次性得出引言中描述的 256 种组合引理变体，以及更多。您可以使用 Tendsto₁ 的全称量词定义或代数定义，连同上述两个引理，来练习证明以下陈述。

```
example {X Y Z : Type*} {F : Filter X} {G : Filter Y} {H : Filter Z} {f : X → Y} {g :
↪Y → Z}
    (hf : Tendsto₁ f F G) (hg : Tendsto₁ g G H) : Tendsto₁ (g ∘ f) F H :=
  sorry
```

前推构造使用一个映射将滤子从映射源推送到映射目标。还有一个“拉回”操作，即 Filter.comap，其方向相反。这推广了集合上的原像操作。对于任何映射 f，

> Filter.map f 和 Filter.comap f 构成了所谓的 **伽罗瓦连接**，也就是说，它们满足
>
> filter.map_le_iff_le_comap：f 映射下的 F 小于等于 G 当且仅当 F 小于等于 G 在 f 下的逆像。

对于每一个 F 和 G。此操作可用于提供“Tendsto”的另一种表述形式，该形式可证明（但不是定义上）等同于 Mathlib 中的那个。

> comap 操作可用于将滤子限制在子类型上。例如，假设我们有 f : ℝ → ℝ、x_0 : ℝ 和 y_0 : ℝ，并且假设我们想要说明当 x 在有理数范围内趋近于 x_0 时，f x 趋近于 y_0。我们可以使用强制转换映射 (↑) : ℚ → ℝ 将滤子 𝓝 x_0 拉回到 ℚ，并声明 Tendsto (f ∘ (↑) : ℚ → ℝ) (comap (↑) (𝓝 x_0)) (𝓝 y_0)。

```
variable (f : ℝ → ℝ) (x₀ y₀ : ℝ)

#check comap ((↑) : ℚ → ℝ) (𝓝 x₀)

#check Tendsto (f ∘ (↑)) (comap ((↑) : ℚ → ℝ) (𝓝 x₀)) (𝓝 y₀)
```

拉回操作也与复合运算兼容，但它具有 **逆变性**，也就是说，它会颠倒参数的顺序。

```
section
variable {α β γ : Type*} (F : Filter α) {m : γ → β} {n : β → α}
```

(continues on next page)

(continued from previous page)

```
#check (comap_comap : comap m (comap n F) = comap (n ∘ m) F)

end
```

现在让我们将注意力转向平面 `ℝ × ℝ`，并尝试理解点 `(x₀, y₀)` 的邻域与 `𝓝 x₀` 和 `𝓝 y₀` 之间的关系。存在一个乘积运算 `Filter.prod : Filter X → Filter Y → Filter (X × Y)`，记作 `×ˢ`，它回答了这个问题：

```
example : 𝓝 (x₀, y₀) = 𝓝 x₀ ×ˢ 𝓝 y₀ :=
  nhds_prod_eq
```

该乘积运算通过拉回运算和 `inf` 运算来定义：

```
F ×ˢ G = (comap Prod.fst F) ⊓ (comap Prod.snd G)
```

这里的 `inf` 操作指的是对于任何类型 X 的 `Filter X` 上的格结构，其中 `F ⊓ G` 是小于 `F` 和 `G` 的最大滤子。因此，`inf` 操作推广了集合交集的概念。

在 Mathlib 中的许多证明都使用了上述所有结构（`map`、`comap`、`inf`、`sup` 和 `prod`），从而给出关于收敛性的代数证明，而无需提及滤子的成员。您可以在以下引理的证明中练习这样做，如果需要，可以展开 `Tendsto` 和 `Filter.prod` 的定义。

```
#check le_inf_iff

example (f : ℕ → ℝ × ℝ) (x₀ y₀ : ℝ) :
    Tendsto f atTop (𝓝 (x₀, y₀)) ↔
      Tendsto (Prod.fst ∘ f) atTop (𝓝 x₀) ∧ Tendsto (Prod.snd ∘ f) atTop (𝓝 y₀) :=
  sorry
```

有序类型 `Filter X` 实际上是一个 **完备**格，也就是说，存在一个最小元素，存在一个最大元素，并且 X 上的每个滤子集都有一个 `Inf` 和一个 `Sup`。

请注意，根据滤子定义中的第二个性质（如果 `U` 属于 `F`，那么任何包含 `U` 的集合也属于 `F`），第一个性质（`X` 的所有元素组成的集合属于 `F`）等价于 `F` 不是空集合这一性质。这不应与更微妙的问题相混淆，即空集是否为 `F` 的一个 **元素**。滤子的定义并不禁止 `∅ ∈ F`，但如果空集在 `F` 中，那么每个集合都在 `F` 中，也就是说，`∀ U : Set X, U ∈ F`。在这种情况下，`F` 是一个相当平凡的滤子，恰好是完备格 `Filter X` 的最小元素。这与布尔巴基对滤子的定义形成对比，布尔巴基的定义不允许滤子包含空集。

由于我们在定义中包含了平凡滤子，所以在某些引理中有时需要明确假设非平凡性。不过作为回报，该理论具有更优的整体性质。我们已经看到，包含平凡滤子为我们提供了一个最小元素。它还允许我们定义 `principal : Set X → Filter X`，将 `∅` 映射到 `⊥`，而无需添加预条件来排除空集。而且它还允许我们定义拉回操作时无需预条件。实际上，有可能 `comap f F = ⊥` 尽管 `F ≠ ⊥`。例如，给定 `x₀ : ℝ` 和 `s : Set ℝ`，`𝓝 x₀` 在从与 `s` 对应的子类型强制转换下的拉回非平凡当且仅当 `x₀` 属于 `s` 的闭包。

为了管理确实需要假设某些滤子非平凡的引理，Mathlib 设有类型类 `Filter.NeBot`，并且库中存在假设（`F`

`: Filter X)` `[F.NeBot]` 的引理。实例数据库知道，例如，`(atTop : Filter ℕ).NeBot`，并且知道将非平凡滤子向前推进会得到一个非平凡滤子。因此，假设 `[F.NeBot]` 的引理将自动应用于任何序列 `u` 的 `map u atTop`。

我们对滤子的代数性质及其与极限的关系的探讨基本上已经完成，但我们尚未证明我们所提出的重新捕捉通常极限概念的主张是合理的。表面上看，`Tendsto u atTop (𝓝 x₀)` 似乎比在:numref: `sequences_and_convergence` 中定义的收敛概念更强，因为我们要求 `x₀` 的 **每个**邻域都有一个属于 `atTop` 的原像，而通常的定义仅要求对于标准邻域 `Ioo (x₀ - ε) (x₀ + ε)` 满足这一条件。关键在于，根据定义，每个邻域都包含这样的标准邻域。这一观察引出了 **滤子基（filter basis）** 的概念。

给定 `F : Filter X`，如果对于每个集合 `U`，我们有 `U ∈ F` 当且仅当它包含某个 `s i`，那么集合族 `s : ι → Set X` 就是 `F` 的基。换句话说，严格说来，`s` 是基当且仅当它满足 `∀ U : Set X, U ∈ F ↔ ∃ i, s i ⊆ U`。考虑在索引类型中仅选择某些值 `i` 的 `ι` 上的谓词会更加灵活。对于 `𝓝 x₀`，我们希望 `ι` 为 `ℝ`，用 `ε` 表示 `i`，并且谓词应选择 `ε` 的正值。因此，集合 `Ioo (x₀ - ε) (x₀ + ε)` 构成 `ℝ` 上邻域拓扑的基这一事实可表述如下：

```
example (x₀ : ℝ) : HasBasis (𝓝 x₀) (fun ε : ℝ ↦ 0 < ε) fun ε ↦ Ioo (x₀ - ε) (x₀ + ε)␣
↪:=
  nhds_basis_Ioo_pos x₀
```

还有一个很好的 `atTop` 滤子的基础。引理 `Filter.HasBasis.tendsto_iff` 允许我们在给定 `F` 和 `G` 的基础的情况下，重新表述形式为 `Tendsto f F G` 的陈述。将这些部分组合在一起，就基本上得到了我们在:numref: `sequences_and_convergence` 中使用的收敛概念。

```
example (u : ℕ → ℝ) (x₀ : ℝ) :
    Tendsto u atTop (𝓝 x₀) ↔ ∀ ε > 0, ∃ N, ∀ n ≥ N, u n ∈ Ioo (x₀ - ε) (x₀ + ε) := by
  have : atTop.HasBasis (fun _ : ℕ ↦ True) Ici := atTop_basis
  rw [this.tendsto_iff (nhds_basis_Ioo_pos x₀)]
  simp
```

现在我们展示一下滤子如何有助于处理对于足够大的数字或对于给定点足够近的点成立的性质。在 Section 3.6 中，我们经常遇到这样的情况：我们知道某个性质 `P n` 对于足够大的 `n` 成立，而另一个性质 `Q n` 对于足够大的 `n` 也成立。使用两次 `cases` 得到了满足 `∀ n ≥ N_P, P n` 和 `∀ n ≥ N_Q, Q n` 的 `N_P` 和 `N_Q`。通过 `set N := max N_P N_Q`，我们最终能够证明 `∀ n ≥ N, P n ∧ Q n`。反复这样做会让人感到厌烦。

我们可以通过注意到“对于足够大的 **n**，`P n` 和 `Q n` 成立”这一表述意味着我们有 `{n | P n} ∈ atTop` 和 `{n | Q n} ∈ atTop` 来做得更好。由于 `atTop` 是一个滤子，所以 `atTop` 中两个元素的交集仍在 `atTop` 中，因此我们有 `{n | P n ∧ Q n} ∈ atTop`。书写 `{n | P n} ∈ atTop` 不太美观，但我们可以用更具提示性的记号 `∀𝓝 n in atTop, P n`。这里的上标 `f` 表示“滤子”。你可以将这个记号理解为对于“非常大的数集”中的所有 `n`，`P n` 成立。`∀𝓝` 记号代表 `Filter.Eventually`，而引理 `Filter.Eventually.and` 利用滤子的交集性质来实现我们刚才所描述的操作：

```
example (P Q : ℕ → Prop) (hP : ∀𝓝 n in atTop, P n) (hQ : ∀𝓝 n in atTop, Q n) :
    ∀𝓝 n in atTop, P n ∧ Q n :=
  hP.and hQ
```

这种表示法如此方便且直观，以至于当 `P` 是一个等式或不等式陈述时，我们也有专门的表示形式。例如，设 `u` 和 `v` 是两个实数序列，让我们证明如果对于足够大的 `n`，`u n` 和 `v n` 相等，那么 `u` 趋向于 x_0 当且仅当 `v` 趋向于 x_0。首先我们将使用通用的 `Eventually`，然后使用专门针对等式谓词的 `EventuallyEq`。这两个陈述在定义上是等价的，因此在两种情况下相同的证明都适用。

```
example (u v : ℕ → ℝ) (h : ∀ᶠ n in atTop, u n = v n) (x₀ : ℝ) :
    Tendsto u atTop (𝓝 x₀) ↔ Tendsto v atTop (𝓝 x₀) :=
  tendsto_congr' h

example (u v : ℕ → ℝ) (h : u =ᶠ[atTop] v) (x₀ : ℝ) :
    Tendsto u atTop (𝓝 x₀) ↔ Tendsto v atTop (𝓝 x₀) :=
  tendsto_congr' h
```

从 `Eventually` 这一概念的角度来审视滤子的定义是很有启发性的。给定滤子 `F : Filter X`，对于 `X` 上的任意谓词 `P` 和 `Q`，

条件 `univ ∈ F` 确保了 `(∀ x, P x) → ∀ᶠ x in F, P x`，条件 `U ∈ F → U ⊆ V → V ∈ F` 确保了 `(∀ᶠ x in F, P x) → (∀ x, P x → Q x) → ∀ᶠ x in F, Q x`，并且条件 `U ∈ F → V ∈ F → U ∩ V ∈ F` 确保了 `(∀ᶠ x in F, P x) → (∀ᶠ x in F, Q x) → ∀ᶠ x in F, P x ∧ Q x`。

```
#check Eventually.of_forall
#check Eventually.mono
#check Eventually.and
```

第二个项目，对应于 `Eventually.mono`，支持使用滤子的优雅方式，尤其是与 `Eventually.and` 结合使用时。`filter_upwards` 策略使我们能够将它们组合起来。比较：

```
example (P Q R : ℕ → Prop) (hP : ∀ᶠ n in atTop, P n) (hQ : ∀ᶠ n in atTop, Q n)
    (hR : ∀ᶠ n in atTop, P n ∧ Q n → R n) : ∀ᶠ n in atTop, R n := by
  apply (hP.and (hQ.and hR)).mono
  rintro n ⟨h, h', h''⟩
  exact h'' ⟨h, h'⟩

example (P Q R : ℕ → Prop) (hP : ∀ᶠ n in atTop, P n) (hQ : ∀ᶠ n in atTop, Q n)
    (hR : ∀ᶠ n in atTop, P n ∧ Q n → R n) : ∀ᶠ n in atTop, R n := by
  filter_upwards [hP, hQ, hR] with n h h' h''
  exact h'' ⟨h, h'⟩
```

熟悉测度论的读者会注意到，补集测度为零的集合构成的滤子 `μ.ae`（即“几乎每个点构成的集合”）作为 `Tendsto` 的源或目标并不是很有用，但它可以方便地与 `Eventually` 一起使用，以表明某个性质对几乎每个点都成立。

存在 `∀ᶠ x in F, P x` 的对偶版本，有时会很有用：用 `∃ᶠ x in F, P x` 表示

`{x | ¬P x} ∉ F`。例如，`∃ᶠ n in atTop, P n` 意味着存在任意大的 `n` 使得 `P n` 成立。`∃ᶠ` 符号

代表 `Filter.Frequently` 。

对于一个更复杂的示例，请考虑以下关于序列 `u` 、集合 `M` 和值 `x` 的陈述：

如果序列 `u` 收敛于 `x` ，且对于足够大的 `n` ， `u n` 属于集合 `M` ，那么 `x` 就在集合 `M` 的闭包内。

这可以形式化表述如下：

```
Tendsto u atTop (𝓝 x) → (∀ᶠ n in atTop, u n ∈ M) → x ∈ closure M.
```

这是拓扑库中定理 `mem_closure_of_tendsto` 的一个特殊情况。试着利用所引用的引理来证明它，利用 `ClusterPt x F` 意味着 `(𝓝 x ⊓ F).NeBot` 这一事实，以及根据定义，`∀ᶠ n in atTop, u n ∈ M` 这一假设意味着 `M ∈ map u atTop` 。

```
#check mem_closure_iff_clusterPt
#check le_principal_iff
#check neBot_of_le

example (u : ℕ → ℝ) (M : Set ℝ) (x : ℝ) (hux : Tendsto u atTop (𝓝 x))
    (huM : ∀ᶠ n in atTop, u n ∈ M) : x ∈ closure M :=
  sorry
```

10.2 度量空间

在上一节中的示例主要关注实数序列。在本节中，我们将提高一点一般性，关注度量空间。度量空间是一种类型 `X` ，它配备了一个距离函数 `dist : X → X → ℝ` ，这是在 `X = ℝ` 情形下函数 `fun x y ↦ |x - y|` 的一种推广。

引入这样一个空间很简单，我们将检验距离函数所需的所有性质。

```
variable {X : Type*} [MetricSpace X] (a b c : X)

#check (dist a b : ℝ)
#check (dist_nonneg : 0 ≤ dist a b)
#check (dist_eq_zero : dist a b = 0 ↔ a = b)
#check (dist_comm a b : dist a b = dist b a)
#check (dist_triangle a b c : dist a c ≤ dist a b + dist b c)
```

请注意，我们还有其他变体，其中距离可以是无穷大，或者 `dist a b` 可以为零而不需要 `a = b` 或者两者皆是。它们分别被称为 `EMetricSpace` 、`PseudoMetricSpace` 和 `PseudoEMetricSpace` （这里“e”代表“扩展”）。

请注意，我们从实数集 `ℝ` 到度量空间的旅程跳过了需要线性代数知识的赋范空间这一特殊情况，这部分内容将在微积分章节中进行解释。

10.2.1 收敛与连续性

利用距离函数，我们已经能够在度量空间之间定义收敛序列和连续函数。实际上，它们是在下一节所涵盖的更一般的情形中定义的，但我们有一些引理将定义重新表述为距离的形式。

```
example {u : ℕ → X} {a : X} :
    Tendsto u atTop (𝓝 a) ↔ ∀ ε > 0, ∃ N, ∀ n ≥ N, dist (u n) a < ε :=
  Metric.tendsto_atTop

example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} :
    Continuous f ↔
      ∀ x : X, ∀ ε > 0, ∃ δ > 0, ∀ x', dist x' x < δ → dist (f x') (f x) < ε :=
  Metric.continuous_iff
```

很多引理都有一些连续性假设，所以我们最终要证明很多连续性结果，并且有一个专门用于此任务的 `连续性` 策略。让我们证明一个连续性陈述，它将在下面的一个练习中用到。请注意，Lean 知道如何将两个度量空间的乘积视为一个度量空间，因此考虑从 `X × X` 到 `ℝ` 的连续函数是有意义的。特别是距离函数（未卷曲的版本）就是这样一种函数。

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) := by continuity
```

这种策略有点慢，所以了解如何手动操作也是有用的。我们首先需要利用 `fun p : X × X ↦ f p.1` 是连续的这一事实，因为它是连续函数 `f`（由假设 `hf` 给出）与投影 `prod.fst` 的复合，而 `prod.fst` 的连续性正是引理 `continuous_fst` 的内容。复合性质是 `Continuous.comp`，它在 `Continuous` 命名空间中，所以我们可以用点表示法将 `Continuous.comp hf continuous_fst` 压缩为 `hf.comp continuous_fst`，这实际上更易读，因为它确实读作将我们的假设和引理进行复合。我们对第二个分量做同样的操作，以获得 `fun p : X × X ↦ f p.2` 的连续性。然后，我们使用 `Continuous.prod_mk` 将这两个连续性组合起来，得到 `(hf.comp continuous_fst).prod_mk (hf.comp continuous_snd) : Continuous (fun p : X × X ↦ (f p.1, f p.2))`，并再次复合以完成我们的完整证明。

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) :=
  continuous_dist.comp ((hf.comp continuous_fst).prod_mk (hf.comp continuous_snd))
```

通过 `Continuous.comp` 将 `Continuous.prod_mk` 和 `continuous_dist` 结合起来的方式感觉很笨拙，即便像上面那样大量使用点标记也是如此。更严重的问题在于，这个漂亮的证明需要大量的规划。Lean 接受上述证明项是因为它是一个完整的项，证明了一个与我们的目标定义上等价的陈述，关键在于要展开的定义是函数的复合。实际上，我们的目标函数 `fun p : X × X ↦ dist (f p.1) (f p.2)` 并未以复合的形式给出。我们提供的证明项证明了 `dist ∘ (fun p : X × X ↦ (f p.1, f p.2))` 的连续性，而这恰好与我们的目标函数定义上相等。但如果尝试从 `apply continuous_dist.comp` 开始逐步使用战术构建这个证明，Lean 的繁

饰器将无法识别复合函数并拒绝应用此引理。当涉及类型乘积时，这种情况尤其糟糕。

这里更适用的引理是

```
Continuous.dist {f g : X → Y} : Continuous f → Continuous g → Continuous (fun x ↦
dist (f x) (g x))
```

它对 Lean 的繁饰器更友好，并且在直接提供完整证明项时也能提供更简短的证明，这一点从以下两个对上述陈述的新证明中可以看出：

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) := by
  apply Continuous.dist
  exact hf.comp continuous_fst
  exact hf.comp continuous_snd

example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) :=
  (hf.comp continuous_fst).dist (hf.comp continuous_snd)
```

请注意，如果不考虑来自组合的详细说明问题，压缩我们证明的另一种方法是使用 `Continuous.prod_map`，它有时很有用，并给出一个替代的证明项 `continuous_dist.comp (hf.prod_map hf)`，这个证明项甚至更短，输入起来也更方便。

由于在便于详细阐述的版本和便于输入的较短版本之间做出选择令人感到遗憾，让我们以 `Continuous.fst'` 提供的最后一点压缩来结束这个讨论，它允许将 `hf.comp continuous_fst` 压缩为 `hf.fst'`（`snd` 也是如此），从而得到我们的最终证明，现在已接近晦涩难懂的程度。

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] {f : X → Y} (hf : Continuous f)␣
↪:
    Continuous fun p : X × X ↦ dist (f p.1) (f p.2) :=
  hf.fst'.dist hf.snd'
```

现在轮到你来证明一些连续性引理了。在尝试了连续性策略之后，你将需要使用 `Continuous.add`、`continuous_pow` 和 `continuous_id` 手动完成证明。

```
example {f : ℝ → X} (hf : Continuous f) : Continuous fun x : ℝ ↦ f (x ^ 2 + x) :=
  sorry
```

到目前为止，我们把连续性视为一个整体概念，但也可以定义某一点处的连续性。

```
example {X Y : Type*} [MetricSpace X] [MetricSpace Y] (f : X → Y) (a : X) :
    ContinuousAt f a ↔ ∀ ε > 0, ∃ δ > 0, ∀ {x}, dist x a < δ → dist (f x) (f a) < ε :=
```

(continues on next page)

(continued from previous page)

```
  Metric.continuousAt_iff
```

10.2.2 球、开集与闭集

一旦我们有了距离函数，最重要的几何定义就是（开）球和闭球。

```
variable (r : ℝ)

example : Metric.ball a r = { b | dist b a < r } :=
  rfl

example : Metric.closedBall a r = { b | dist b a ≤ r } :=
  rfl
```

请注意，这里的 *r* 是任意实数，没有符号限制。当然，有些陈述确实需要半径条件。

```
example (hr : 0 < r) : a ∈ Metric.ball a r :=
  Metric.mem_ball_self hr

example (hr : 0 ≤ r) : a ∈ Metric.closedBall a r :=
  Metric.mem_closedBall_self hr
```

一旦我们有了球，就可以定义开集。实际上，它们是在下一节所涵盖的更一般的情形中定义的，但我们有一些引理将定义重新表述为用球来表示。

```
example (s : Set X) : IsOpen s ↔ ∀ x ∈ s, ∃ ε > 0, Metric.ball x ε ⊆ s :=
  Metric.isOpen_iff
```

那么闭集就是其补集为开集的集合。它们的重要性质是它们在极限运算下是封闭的。一个集合的闭包是包含它的最小闭集。

```
example {s : Set X} : IsClosed s ↔ IsOpen (s) :=
  isOpen_compl_iff.symm

example {s : Set X} (hs : IsClosed s) {u : ℕ → X} (hu : Tendsto u atTop ( a))
    (hus : ∀ n, u n ∈ s) : a ∈ s :=
  hs.mem_of_tendsto hu (Eventually.of_forall hus)

example {s : Set X} : a ∈ closure s ↔ ∀ ε > 0, ∃ b ∈ s, a ∈ Metric.ball b ε :=
  Metric.mem_closure_iff
```

请在不使用 *mem_closure_iff_seq_limit* 的情况下完成下一个练习。

```
example {u : ℕ → X} (hu : Tendsto u atTop (𝓝 a)) {s : Set X} (hs : ∀ n, u n ∈ s) :
    a ∈ closure s :=
  sorry
```

请记住，在滤子部分中提到，邻域滤子在 Mathlib 中起着重要作用。在度量空间的背景下，关键在于球体为这些滤子提供了基。这里的主要引理是 `Metric.nhds_basis_ball` 和 `Metric.nhds_basis_closedBall` ，它们分别表明具有正半径的开球和闭球具有这一性质。中心点是一个隐式参数，因此我们可以像下面的例子那样调用 `Filter.HasBasis.mem_iff` 。

```
example {x : X} {s : Set X} : s ∈ 𝓝 x ↔ ∃ ε > 0, Metric.ball x ε ⊆ s :=
  Metric.nhds_basis_ball.mem_iff

example {x : X} {s : Set X} : s ∈ 𝓝 x ↔ ∃ ε > 0, Metric.closedBall x ε ⊆ s :=
  Metric.nhds_basis_closedBall.mem_iff
```

10.2.3 紧致性

紧性是一个重要的拓扑概念。它区分了度量空间中的子集，这些子集具有与实数中的线段相同的性质，而与其他区间不同：

- 任何取值于紧集中的序列都有一个子序列在该紧集中收敛。
- 在非空紧集上取实数值的任何连续函数都是有界的，并且在某个地方达到其界值（这被称为极值定理）。
- 紧集是闭集。

首先让我们验证实数中的单位区间确实是一个紧集，然后验证一般度量空间中紧集的上述断言。在第二个陈述中，我们只需要在给定的集合上连续，因此我们将使用 `ContinuousOn` 而不是 `Continuous` ，并且我们将分别给出最小值和最大值的陈述。当然，所有这些结果都是从更一般的形式推导出来的，其中一些将在后面的章节中讨论。

```
example : IsCompact (Set.Icc 0 1 : Set ℝ) :=
  isCompact_Icc

example {s : Set X} (hs : IsCompact s) {u : ℕ → X} (hu : ∀ n, u n ∈ s) :
    ∃ a ∈ s, ∃ φ : ℕ → ℕ, StrictMono φ ∧ Tendsto (u ∘ φ) atTop (𝓝 a) :=
  hs.tendsto_subseq hu

example {s : Set X} (hs : IsCompact s) (hs' : s.Nonempty) {f : X → ℝ}
      (hfs : ContinuousOn f s) :
    ∃ x ∈ s, ∀ y ∈ s, f x ≤ f y :=
  hs.exists_isMinOn hs' hfs

example {s : Set X} (hs : IsCompact s) (hs' : s.Nonempty) {f : X → ℝ}
```

(continues on next page)

(continued from previous page)

```
    (hfs : ContinuousOn f s) :
  ∃ x ∈ s, ∀ y ∈ s, f y ≤ f x :=
  hs.exists_isMaxOn hs' hfs

example {s : Set X} (hs : IsCompact s) : IsClosed s :=
  hs.isClosed
```

我们还可以通过添加一个额外的 `Prop` 值类型类来指定度量空间是全局紧致的：

```
example {X : Type*} [MetricSpace X] [CompactSpace X] : IsCompact (univ : Set X) :=
  isCompact_univ
```

在紧致度量空间中，任何闭集都是紧致的，这就是 `IsClosed.isCompact` 。一致连续函数 ^^^^^^^^^^

现在我们来探讨度量空间上的均匀性概念：一致连续函数、柯西序列和完备性。同样，这些概念是在更一般的背景下定义的，但在度量空间中我们有引理来获取它们的基本定义。我们先从一致连续性讲起。

```
example {X : Type*} [MetricSpace X] {Y : Type*} [MetricSpace Y] {f : X → Y} :
    UniformContinuous f ↔
      ∀ ε > 0, ∃ δ > 0, ∀ {a b : X}, dist a b < δ → dist (f a) (f b) < ε :=
  Metric.uniformContinuous_iff
```

为了练习运用所有这些定义，我们将证明从紧致度量空间到度量空间的连续函数是一致连续的（在后面的章节中我们将看到更一般的形式）。

我们首先给出一个非正式的概述。设 `f : X → Y` 是从一个紧致度量空间到一个度量空间的连续函数。我们固定 `ε > 0` ，然后开始寻找某个 `δ` 。

令 `φ : X × X → ℝ := fun p ↦ dist (f p.1) (f p.2)` 以及 `K := { p : X × X | ε ≤ φ p }` 。注意到由于 `f` 和距离函数都是连续的，所以 `φ` 也是连续的。并且 `K` 显然是闭集（使用 `isClosed_le` ），因此由于 `X` 是紧致的，所以 `K` 也是紧致的。

然后我们使用 `eq_empty_or_nonempty` 来讨论两种可能性。如果集合 `K` 为空，那么显然我们已经完成了（例如，我们可以设 `δ = 1` ）。所以假设 `K` 不为空，利用极值定理选择 (x_0, x_1) 使得距离函数在 `K` 上达到最小值。然后我们可以设 $\delta = \text{dist } x_0\ x_1$ 并检查一切是否都正常。

```
example {X : Type*} [MetricSpace X] [CompactSpace X]
    {Y : Type*} [MetricSpace Y] {f : X → Y}
    (hf : Continuous f) : UniformContinuous f :=
  sorry
```

10.2.4 完备性

度量空间中的柯西序列是指其各项越来越接近的序列。表述这一概念有几种等价的方式。特别是收敛序列是柯西序列。但反过来只有在所谓的 **完备**空间中才成立。

```
example (u : ℕ → X) :
    CauchySeq u ↔ ∀ ε > 0, ∃ N : ℕ, ∀ m ≥ N, ∀ n ≥ N, dist (u m) (u n) < ε :=
  Metric.cauchySeq_iff

example (u : ℕ → X) :
    CauchySeq u ↔ ∀ ε > 0, ∃ N : ℕ, ∀ n ≥ N, dist (u n) (u N) < ε :=
  Metric.cauchySeq_iff'

example [CompleteSpace X] (u : ℕ → X) (hu : CauchySeq u) :
    ∃ x, Tendsto u atTop (𝓝 x) :=
  cauchySeq_tendsto_of_complete hu
```

我们将通过证明一个方便的判别式来练习使用这个定义，该判别式是 Mathlib 中出现的一个判别式的特殊情况。这也是一个在几何背景下练习使用大求和符号的好机会。除了滤子部分的解释外，您可能还需要使用 `tendsto_pow_atTop_nhds_zero_of_lt_one` 、`Tendsto.mul` 和 `dist_le_range_sum_dist` 。

```
theorem cauchySeq_of_le_geometric_two' {u : ℕ → X}
    (hu : ∀ n : ℕ, dist (u n) (u (n + 1)) ≤ (1 / 2) ^ n) : CauchySeq u := by
  rw [Metric.cauchySeq_iff']
  intro ε ε_pos
  obtain ⟨N, hN⟩ : ∃ N : ℕ, 1 / 2 ^ N * 2 < ε := by sorry
  use N
  intro n hn
  obtain ⟨k, rfl : n = N + k⟩ := le_iff_exists_add.mp hn
  calc
    dist (u (N + k)) (u N) = dist (u (N + 0)) (u (N + k)) := sorry
    _ ≤ ∑ i in range k, dist (u (N + i)) (u (N + (i + 1))) := sorry
    _ ≤ ∑ i in range k, (1 / 2 : ℝ) ^ (N + i) := sorry
    _ = 1 / 2 ^ N * ∑ i in range k, (1 / 2 : ℝ) ^ i := sorry
    _ ≤ 1 / 2 ^ N * 2 := sorry
    _ < ε := sorry
```

我们已准备好迎接本节的最终大 Boss：完备度量空间上的贝尔纲定理（Baire’s theorem）！下面的证明框架展示了有趣的技术。它使用了感叹号形式的 `choose` 策略（您应该尝试去掉这个感叹号），并且展示了如何在证明过程中使用 `Nat.rec_on` 来递归定义某些内容。

```
open Metric
```

(continues on next page)

(continued from previous page)

```
example [CompleteSpace X] (f : ℕ → Set X) (ho : ∀ n, IsOpen (f n)) (hd : ∀ n, Dense␣
↪(f n)) :
    Dense (⋂ n, f n) := by
  let B : ℕ → ℝ := fun n ↦ (1 / 2) ^ n
  have Bpos : ∀ n, 0 < B n
  sorry
  /- 将密度假设转化为两个函数 `center` 和 `radius`，对于任意的 n、x、δ、δpos，这两个函数分别
关联一个中心和一个正半径，使得 `closedBall center radius` 同时包含在 `f n` 和 `closedBall␣
↪x δ` 中。我们还可以要求 `radius ≤ (1/2)^(n+1)`，以确保之后能得到一个柯西序列。-/
  have :
    ∀ (n : ℕ) (x : X),
      ∀ δ > 0, ∃ y : X, ∃ r > 0, r ≤ B (n + 1) ∧ closedBall y r ⊆ closedBall x δ ∩ f␣
↪n :=
    by sorry
  choose! center radius Hpos HB Hball using this
  intro x
  rw [mem_closure_iff_nhds_basis nhds_basis_closedBall]
  intro ε εpos
  /- 设 `ε` 为正数。我们需要在以 `x` 为圆心、半径为 `ε` 的球内找到一个点，该点属于所有的␣
↪`f n`。为此，我们递归地构造一个序列 `F n = (c n, r n)`，使得闭球 `closedBall (c n) (r␣
↪n)` 包含在前一个球内且属于 `f n`，并且 `r n` 足够小以确保 `c n` 是一个柯西序列。那么 `c␣
↪n` 收敛到一个极限，该极限属于所有的 `f n`。-/
  let F : ℕ → X × ℝ := fun n ↦
    Nat.recOn n (Prod.mk x (min ε (B 0)))
      fun n p ↦ Prod.mk (center n p.1 p.2) (radius n p.1 p.2)
  let c : ℕ → X := fun n ↦ (F n).1
  let r : ℕ → ℝ := fun n ↦ (F n).2
  have rpos : ∀ n, 0 < r n := by sorry
  have rB : ∀ n, r n ≤ B n := by sorry
  have incl : ∀ n, closedBall (c (n + 1)) (r (n + 1)) ⊆ closedBall (c n) (r n) ∩ f n␣
↪:= by
    sorry
  have cdist : ∀ n, dist (c n) (c (n + 1)) ≤ B n := by sorry
  have : CauchySeq c := cauchySeq_of_le_geometric_two' cdist
  -- 由于序列 `c n` 在完备空间中是柯西序列，所以它收敛于极限 `y`。
  -- 根据完备空间中柯西序列收敛的定理，存在 `y` 使得 `c n` 收敛于 `y`，记为 `ylim`。
  -- 这个点 `y` 就是我们想要的点。接下来我们要验证它属于所有的 `f n` 以及 `ball x ε`。
  use y
  have I : ∀ n, ∀ m ≥ n, closedBall (c m) (r m) ⊆ closedBall (c n) (r n) := by sorry
```

(continues on next page)

(continued from previous page)

```
  have yball : ∀ n, y ∈ closedBall (c n) (r n) := by sorry
  sorry
```

10.3 拓扑空间

10.3.1 基础

我们现在提高一般性，引入拓扑空间。我们将回顾定义拓扑空间的两种主要方式，然后解释拓扑空间范畴比度量空间范畴表现得要好得多。请注意，这里我们不会使用 Mathlib 的范畴论，只是采用一种稍微范畴化一点的观点。

从度量空间到拓扑空间转变的第一种思考方式是，我们只记住开集（或等价地，闭集）的概念。从这个角度来看，拓扑空间是一种配备了被称为开集的集合族的类型。这个集合族必须满足下面给出的若干公理（这个集合族稍有冗余，但我们忽略这一点）。

```
section
variable {X : Type*} [TopologicalSpace X]

example : IsOpen (univ : Set X) :=
  isOpen_univ

example : IsOpen (∅ : Set X) :=
  isOpen_empty

example {ι : Type*} {s : ι → Set X} (hs : ∀ i, IsOpen (s i)) : IsOpen (⋃ i, s i) :=
  isOpen_iUnion hs

example {ι : Type*} [Fintype ι] {s : ι → Set X} (hs : ∀ i, IsOpen (s i)) :
    IsOpen (⋂ i, s i) :=
  isOpen_iInter_of_finite hs
```

闭集被定义为补集是开集的集合。拓扑空间之间的函数是（全局）连续的，当且仅当所有开集的原像都是开集。

```
variable {Y : Type*} [TopologicalSpace Y]

example {f : X → Y} : Continuous f ↔ ∀ s, IsOpen s → IsOpen (f ⁻¹' s) :=
  continuous_def
```

根据这个定义，我们已经可以看出，与度量空间相比，拓扑空间仅保留了足够的信息来讨论连续函数：如果且仅如果两种拓扑结构具有相同的连续函数，则类型上的两种拓扑结构相同（实际上，当且仅当两种结构具有相同的开集时，恒等函数在两个方向上都是连续的）。

然而，一旦我们转向某一点的连续性，就会看到基于开集的方法的局限性。在 Mathlib 中，我们经常将拓扑空间视为每个点 x 都附带一个邻域滤子 `𝓝 x` 的类型（相应的函数 `X → Filter X` 满足进一步说明的某些条件）。回想一下滤子部分的内容，这些工具发挥着两个相关的作用。首先，`𝓝 x` 被视为 X 中接近 `x` 的点的广义集合。其次，它被视为一种方式，用于说明对于任何谓词 `P : X → Prop`，该谓词对于足够接近 x 的点成立。让我们来陈述 `f : X → Y` 在 `x` 处连续。纯粹基于滤子的说法是，`f` 下 `x` 附近点的广义集合的直接像包含在 `f x` 附近点的广义集合中。回想一下，这可以写成 `map f (𝓝 x) ≤ 𝓝 (f x)` 或者 `Tendsto f (𝓝 x) (𝓝 (f x))`。

```
example {f : X → Y} {x : X} : ContinuousAt f x ↔ map f (𝓝 x) ≤ 𝓝 (f x) :=
  Iff.rfl
```

还可以使用被视为普通集合的邻域和被视为广义集合的邻域滤子来拼写它：“对于 `f x` 的任何邻域 `U`，所有靠近 `x` 的点都被发送到 `U`”。请注意，证明又是 `iff.rfl`，这种观点在定义上与前一种观点等价。

```
example {f : X → Y} {x : X} : ContinuousAt f x ↔ ∀ U ∈ 𝓝 (f x), ∀ᶠ x in 𝓝 x, f x ∈ U
↪:=
  Iff.rfl
```

现在我们来解释如何从一种观点转换到另一种观点。就开集而言，我们可以简单地将 `𝓝 x` 的成员定义为包含一个包含 `x` 的开集的集合。

```
example {x : X} {s : Set X} : s ∈ 𝓝 x ↔ ∃ t, t ⊆ s ∧ IsOpen t ∧ x ∈ t :=
  mem_nhds_iff
```

要朝另一方向进行，我们需要讨论 `𝓝 : X → Filter X` 成为拓扑的邻域函数必须满足的条件。

第一个约束条件是，将 `𝓝 x` 视为广义集合时，它将包含被视为广义集合 `pure x` 的集合 `{x}`（解释这个奇怪的名字会离题太远，所以我们暂时接受它）。另一种说法是，如果一个谓词对靠近 `x` 的点成立，那么它在 `x` 处也成立。

```
example (x : X) : pure x ≤ 𝓝 x :=
  pure_le_nhds x

example (x : X) (P : X → Prop) (h : ∀ᶠ y in 𝓝 x, P y) : P x :=
  h.self_of_nhds
```

然后一个更微妙的要求是，对于任何谓词 `P : X → Prop` 以及任何 `x`，如果 `P y` 对于接近 `x` 的 `y` 成立，那么对于接近 `x` 的 `y` 以及接近 `y` 的 `z`，`P z` 也成立。更确切地说，我们有：

```
example {P : X → Prop} {x : X} (h : ∀ᶠ y in 𝓝 x, P y) : ∀ᶠ y in 𝓝 x, ∀ᶠ z in 𝓝 y, P z
↪:=
  eventually_eventually_nhds.mpr h
```

这两个结果描述了对于集合 X 上的拓扑空间结构而言，函数 `X → Filter X` 成为邻域函数的特征。仍然存在一个函数 `TopologicalSpace.mkOfNhds : (X → Filter X) → TopologicalSpace X`，但只有当输入满

足上述两个约束条件时，它才会将其作为邻域函数返回。更确切地说，我们有一个引理 `TopologicalSpace.nhds_mkOfNhds`，以另一种方式说明了这一点，而我们的下一个练习将从上述表述方式推导出这种不同的表述方式。

```
example {α : Type*} (n : α → Filter α) (H₀ : ∀ a, pure a ≤ n a)
    (H : ∀ a : α, ∀ p : α → Prop, (∀ᶠ x in n a, p x) → ∀ᶠ y in n a, ∀ᶠ x in n y, p x)␣
↪:
    ∀ a, ∀ s ∈ n a, ∃ t ∈ n a, t ⊆ s ∧ ∀ a' ∈ t, s ∈ n a' :=
  sorry
```

请注意，`TopologicalSpace.mkOfNhds` 并不经常使用，但了解拓扑空间结构中邻域滤子的精确含义仍然是很有好处的。

要在 Mathlib 中高效地使用拓扑空间，接下来需要了解的是，我们大量使用了 `TopologicalSpace : Type u → Type u` 的形式属性。从纯粹的数学角度来看，这些形式属性是解释拓扑空间如何解决度量空间存在的问题的一种非常清晰的方式。从这个角度来看，拓扑空间解决的问题在于度量空间的函子性非常差，而且总体上具有非常糟糕的范畴性质。这还不包括前面已经讨论过的度量空间包含大量拓扑上无关的几何信息这一事实。

我们先关注函子性。度量空间结构可以诱导到子集上，或者等价地说，可以通过单射映射拉回。但也就仅此而已。它们不能通过一般的映射拉回，甚至不能通过满射映射推前。

特别是对于度量空间的商空间或不可数个度量空间的乘积，不存在合理的距离。例如，考虑类型 `ℝ → ℝ`，将其视为由 `ℝ` 索引的 `ℝ` 的副本的乘积。我们希望说函数序列的逐点收敛是一种值得考虑的收敛概念。但在 `ℝ → ℝ` 上不存在能给出这种收敛概念距离。与此相关的是，不存在这样的距离，使得映射 `f : X → (ℝ → ℝ)` 是连续的当且仅当对于每个 `t : ℝ`，`fun x ↦ f x t` 是连续的。

现在我们来回顾一下用于解决所有这些问题的数据。首先，我们可以使用任何映射 `f : X → Y` 将拓扑从一侧推到另一侧或从另一侧拉到这一侧。这两个操作形成了一个伽罗瓦连接。

```
variable {X Y : Type*}

example (f : X → Y) : TopologicalSpace X → TopologicalSpace Y :=
  TopologicalSpace.coinduced f

example (f : X → Y) : TopologicalSpace Y → TopologicalSpace X :=
  TopologicalSpace.induced f

example (f : X → Y) (T_X : TopologicalSpace X) (T_Y : TopologicalSpace Y) :
    TopologicalSpace.coinduced f T_X ≤ T_Y ↔ T_X ≤ TopologicalSpace.induced f T_Y :=
  coinduced_le_iff_le_induced
```

这些操作与函数的复合兼容。通常，前推是协变的，后拉是逆变的，参见 `coinduced_compose` 和 `induced_compose`。在纸上，我们将使用记号 f_*T 表示 `TopologicalSpace.coinduced f T`，使用记号 f^*T 表示 `TopologicalSpace.induced f T`。接下来的一个重要部分是在给定结构下对 拓扑空间 `X` 建立一个完

整的格结构。如果您认为拓扑主要是开集的数据，那么您会期望 拓扑空间 `X` 上的序关系来自 `Set (Set X)`，即您期望 `t ≤ t'` 当且仅当对于 `t'` 中的开集 `u`，它在 `t` 中也是开集。然而，我们已经知道 Mathlib 更侧重于邻域而非开集，因此对于任何 `x : X`，我们希望从拓扑空间到邻域的映射 `fun T : TopologicalSpace X ↦ @nhds X T x` 是保序的。而且我们知道 `Filter X` 上的序关系是为确保 `principal : Set X → Filter X` 保序而设计的，从而可以将滤子视为广义集合。所以我们在 `TopologicalSpace X` 上使用的序关系与来自 `Set (Set X)` 的序关系是相反的。

```
example {T T' : TopologicalSpace X} : T ≤ T' ↔ ∀ s, T'.IsOpen s → T.IsOpen s :=
  Iff.rfl
```

现在，我们可以通过将推进（或拉回）操作与序关系相结合来恢复连续性。

```
example (T_X : TopologicalSpace X) (T_Y : TopologicalSpace Y) (f : X → Y) :
    Continuous f ↔ TopologicalSpace.coinduced f T_X ≤ T_Y :=
  continuous_iff_coinduced_le
```

有了这个定义以及前推和复合的兼容性，我们自然地得到了这样一个通用性质：对于任何拓扑空间 Z，函数 $g : Y \to Z$ 对于拓扑 $f_* T_X$ 是连续的，当且仅当 $g \circ f$ 是连续的。

$$\begin{aligned} g \text{ continuous} &\Leftrightarrow g_*(f_* T_X) \leq T_Z \\ &\Leftrightarrow (g \circ f)_* T_X \leq T_Z \\ &\Leftrightarrow g \circ f \text{ continuous} \end{aligned}$$

```
example {Z : Type*} (f : X → Y) (T_X : TopologicalSpace X) (T_Z : TopologicalSpace Z)
      (g : Y → Z) :
    @Continuous Y Z (TopologicalSpace.coinduced f T_X) T_Z g ↔
      @Continuous X Z T_X T_Z (g ∘ f) := by
  rw [continuous_iff_coinduced_le, coinduced_compose, continuous_iff_coinduced_le]
```

因此，我们已经得到了商拓扑（使用投影映射作为 `f`）。这并没有用到对于所有 `X`，`TopologicalSpace X` 都是一个完备格这一事实。现在让我们看看所有这些结构如何通过抽象的废话证明积拓扑的存在性。我们上面考虑了 `ℝ → ℝ` 的情况，但现在让我们考虑一般情况 `Π i, X i`，其中 `ι : Type*` 且 `X : ι → Type*`。我们希望对于任何拓扑空间 `Z` 以及任何函数 `f : Z → Π i, X i`，`f` 是连续的当且仅当 `(fun x ↦ x i) ∘ f` 对于所有 `i` 都是连续的。让我们在纸上用符号 `p_i` 表示投影 `(fun (x : Π i, X i) ↦ x i)` 来探究这个约束条件：

$$\begin{aligned} (\forall i, p_i \circ f \text{ continuous}) &\Leftrightarrow \forall i, (p_i \circ f)_* T_Z \leq T_{X_i} \\ &\Leftrightarrow \forall i, (p_i)_* f_* T_Z \leq T_{X_i} \\ &\Leftrightarrow \forall i, f_* T_Z \leq (p_i)^* T_{X_i} \\ &\Leftrightarrow f_* T_Z \leq \inf \left[(p_i)^* T_{X_i} \right] \end{aligned}$$

因此我们看到，对于 `Π i, X i`，我们想要的拓扑结构是什么：

```
example (ι : Type*) (X : ι → Type*) (T_X : ∀ i, TopologicalSpace (X i)) :
    (Pi.topologicalSpace : TopologicalSpace (∀ i, X i)) =
      ⨅ i, TopologicalSpace.induced (fun x ↦ x i) (T_X i) :=
  rfl
```

这就结束了我们关于 Mathlib 的探讨，其如何认为拓扑空间通过成为更具函子性的理论与对于任何固定类型都具有完备格结构的特性，从而弥补度量空间理论的缺陷。

10.3.2 分离性与可数性

我们看到拓扑空间范畴具有非常良好的性质。为此付出的代价是存在一些相当病态的拓扑空间。你可以对拓扑空间做出一些假设，以确保其行为更接近度量空间。其中最重要的是 `T2` 空间，也称为“豪斯多夫空间”，它能确保极限是唯一的。更强的分离性质是 `T3` 空间，它还确保了正则空间性质：每个点都有一个闭邻域基。

```
example [TopologicalSpace X] [T2Space X] {u : ℕ → X} {a b : X} (ha : Tendsto u atTop␣
↪(𝓝 a))
    (hb : Tendsto u atTop (𝓝 b)) : a = b :=
  tendsto_nhds_unique ha hb

example [TopologicalSpace X] [RegularSpace X] (a : X) :
    (𝓝 a).HasBasis (fun s : Set X ↦ s ∈ 𝓝 a ∧ IsClosed s) id :=
  closed_nhds_basis a
```

请注意，根据定义，在每个拓扑空间中，每个点都有一个开邻域基。

```
example [TopologicalSpace X] {x : X} :
    (𝓝 x).HasBasis (fun t : Set X ↦ t ∈ 𝓝 x ∧ IsOpen t) id :=
  nhds_basis_opens' x
```

我们现在的主要目标是证明基本定理，该定理允许通过连续性进行延拓。从布尔巴基学派的《一般拓扑学》一书，第 I 卷第 8.5 节，定理 1（仅取非平凡的蕴含部分）：

设 X 为拓扑空间，A 是 X 的一个稠密子集，$f : A \to Y$ 是将 A 映射到 T_3 空间 Y 的连续映射。若对于 X 中的每个点 x，当 y 趋近于 x 且始终处于 A 内时，$f(y)$ 在 Y 中趋于一个极限，则存在 f 在 X 上的连续延拓 。

实际上，Mathlib 包含了上述引理的一个更通用的版本 `DenseInducing.continuousAt_extend`，但在这里我们将遵循布尔巴基的版本。

请记住，对于 `A : Set X`，`↥A` 是与 `A` 相关联的子类型，并且在需要时，Lean 会自动插入那个有趣的上箭头。而（包含）强制转换映射为 `(↑) : A → X`。假设“趋向于 x 且始终处于 A 中”对应于拉回滤子 `comap (↑) (𝓝 x)`。

我们首先证明一个辅助引理，将其提取出来以简化上下文（特别是这里我们不需要 Y 是拓扑空间）。

```
theorem aux {X Y A : Type*} [TopologicalSpace X] {c : A → X}
      {f : A → Y} {x : X} {F : Filter Y}
      (h : Tendsto f (comap c (𝓝 x)) F) {V' : Set Y} (V'_in : V' ∈ F) :
    ∃ V ∈ 𝓝 x, IsOpen V ∧ c ⁻¹' V ⊆ f ⁻¹' V' := by
  sorry
```

现在让我们来证明连续性延拓定理的主要内容。

当需要在 `↥A` 上定义拓扑时，Lean 会自动使用诱导拓扑。唯一相关的引理是

```
nhds_induced (↑) : ∀ a : ↥A, 𝓝 a = comap (↑) (𝓝 ↑a)
```

（这实际上是一个关于诱导拓扑的一般引理）。

证明的大致思路是：

主要假设和选择公理给出一个函数 φ，使得

```
∀ x, Tendsto f (comap (↑) (𝓝 x)) (𝓝 (φ x))
```

（因为 `Y` 是豪斯多夫空间，`φ` 是完全确定的，但在我们试图证明 `φ` 确实扩展了 `f` 之前，我们不需要这一点）。

首先证明 `φ` 是连续的。固定任意的 `x : X`。由于 `Y` 是正则的，只需检查对于 `φ x` 的每个**闭**邻域 `V'`，`φ⁻¹' V' ∈ 𝓝 x`。极限假设（通过上面的辅助引理）给出了某个 `V ∈ 𝓝 x`，使得 `IsOpen V ∧ (↑)⁻¹' V ⊆ f⁻¹' V'`。由于 `V ∈ 𝓝 x`，只需证明 `V ⊆ φ⁻¹' V'`，即 `∀ y ∈ V, φ y ∈ V'`。固定 `V` 中的 `y`。因为 `V` 是**开**的，所以它是 `y` 的邻域。特别是 `(↑)⁻¹' V ∈ comap (↑) (𝓝 y)` 且更进一步 `f⁻¹' V' ∈ comap (↑) (𝓝 y)`。此外，`comap (↑) (𝓝 y) ≠ ⊥` 因为 `A` 是稠密的。因为我们知道 `Tendsto f (comap (↑) (𝓝 y)) (𝓝 (φ y))`，这表明 `φ y ∈ closure V'`，并且由于 `V'` 是闭的，我们已经证明了 `φ y ∈ V'`。

接下来要证明的是 `φ` 延拓了 `f`。这里就要用到 `f` 的连续性以及 `Y` 是豪斯多夫空间这一事实。

```
example [TopologicalSpace X] [TopologicalSpace Y] [T3Space Y] {A : Set X}
    (hA : ∀ x, x ∈ closure A) {f : A → Y} (f_cont : Continuous f)
    (hf : ∀ x : X, ∃ c : Y, Tendsto f (comap (↑) (𝓝 x)) (𝓝 c)) :
    ∃ φ : X → Y, Continuous φ ∧ ∀ a : A, φ a = f a := by
  sorry

#check HasBasis.tendsto_right_iff
```

除了分离性之外，您还可以对拓扑空间做出的主要假设是可数性假设，以使其更接近度量空间。其中最主要的是第一可数性，即要求每个点都有一个可数的邻域基。特别是，这确保了集合的闭包可以通过序列来理解。

```
example [TopologicalSpace X] [FirstCountableTopology X]
      {s : Set X} {a : X} :
    a ∈ closure s ↔ ∃ u : ℕ → X, (∀ n, u n ∈ s) ∧ Tendsto u atTop (𝓝 a) :=
  mem_closure_iff_seq_limit
```

10.3.3 紧致性

现在让我们来讨论一下拓扑空间的紧致性是如何定义的。和往常一样，对此有多种思考方式，而 Mathlib 采用的是滤子版本。

我们首先需要定义滤子的聚点。给定拓扑空间 `X` 上的一个滤子 `F`，若将 `F` 视为广义集合，则点 `x : X` 是 `F` 的聚点，当且仅当 `F` 与广义集合中所有接近 `x` 的点的集合有非空交集。

那么我们就可以说，集合 `s` 是紧致的，当且仅当包含于 `s` 中的每一个非空广义集合 `F`，即满足 `F ≤ 𝓟 s` 的集合，都在 `s` 中有一个聚点。

```
variable [TopologicalSpace X]

example {F : Filter X} {x : X} : ClusterPt x F ↔ NeBot (𝓝 x ⊓ F) :=
  Iff.rfl

example {s : Set X} :
    IsCompact s ↔ ∀ (F : Filter X) [NeBot F], F ≤ 𝓟 s → ∃ a ∈ s, ClusterPt a F :=
  Iff.rfl
```

例如，如果 `F` 是 `map u atTop`，即 `u : ℕ → X` 在 `atTop` 下的像，其中 `atTop` 是非常大的自然数的广义集合，那么假设 `F ≤ 𝓟 s` 意味着对于足够大的 `n`，`u n` 属于 `s`。说 `x` 是 `map u atTop` 的聚点意味着非常大的数的像与接近 `x` 的点的集合相交。如果 `𝓝 x` 有一个可数基，我们可以将其解释为说 `u` 有一个子序列收敛于 `x`，这样我们就得到了度量空间中紧致性的样子。

```
example [FirstCountableTopology X] {s : Set X} {u : ℕ → X} (hs : IsCompact s)
    (hu : ∀ n, u n ∈ s) : ∃ a ∈ s, ∃ φ : ℕ → ℕ, StrictMono φ ∧ Tendsto (u ∘ φ) atTop␣
↪(𝓝 a) :=
  hs.tendsto_subseq hu
```

聚点与连续函数的性质相容。

```
variable [TopologicalSpace Y]

example {x : X} {F : Filter X} {G : Filter Y} (H : ClusterPt x F) {f : X → Y}
    (hfx : ContinuousAt f x) (hf : Tendsto f F G) : ClusterPt (f x) G :=
  ClusterPt.map H hfx hf
```

作为练习，我们将证明连续映射下的紧集的像是紧集。除了我们已经看到的内容外，您还应该使用 `Filter.push_pull` 和 `NeBot.of_map`。

```
example [TopologicalSpace Y] {f : X → Y} (hf : Continuous f) {s : Set X} (hs :␣
↪IsCompact s) :
    IsCompact (f '' s) := by
```

(continues on next page)

(continued from previous page)

```
  intro F F_ne F_le
  have map_eq : map f (𝓝 s ⊓ comap f F) = 𝓝 (f '' s) ⊓ F := by sorry
  have Hne : (𝓝 s ⊓ comap f F).NeBot := by sorry
  have Hle : 𝓝 s ⊓ comap f F ≤ 𝓝 s := inf_le_left
  sorry
```

也可以用开覆盖来表述紧性：s 是紧的，当且仅当覆盖 s 的每一个开集族都有一个有限的覆盖子族。

```
example {ι : Type*} {s : Set X} (hs : IsCompact s) (U : ι → Set X) (hUo : ∀ i, IsOpen␣
↪(U i))
    (hsU : s ⊆ ⋃ i, U i) : ∃ t : Finset ι, s ⊆ ⋃ i ∈ t, U i :=
  hs.elim_finite_subcover U hUo hsU
```

CHAPTER

ELEVEN

微分学

我们现在考虑来自 **分析学**概念的形式化，从本章的微分开始，然后在下一章讨论积分和测度论。在 Section 11.1 中，我们仍采用从实数到实数的函数这一常见于任何微积分入门课程的设定。在 Section 11.2 中，我们则在更广泛的背景下考虑导数的概念。

11.1 初等微分学

设 `f` 为从实数到实数的函数。讨论 `f` 在单个点处的导数与讨论导数函数是有区别的。在 Mathlib 中，第一个概念表示如下。

```
open Real

/-- 正弦函数在 0 处的导数为 1 。 -/
example : HasDerivAt sin 1 0 := by simpa using hasDerivAt_sin 0
```

我们也可以在不指明某点处导数的情况下表达函数 `f` 在该点可微，方法是写成 `DifferentiableAt ℝ` 。我们明确写出 `ℝ` 是因为在稍更一般的语境中，当我们讨论从 `ℂ` 到 `ℂ` 的函数时，我们希望能够在实可微和复可微（即复导数的意义下可微）之间做出区分。

```
example (x : ℝ) : DifferentiableAt ℝ sin x :=
  (hasDerivAt_sin x).differentiableAt
```

每次想要提及导数时都得提供可微性的证明，这会很不方便。因此，Mathlib 提供了一个函数 `deriv f : ℝ → ℝ` ，它适用于任何 `f : ℝ → ℝ` 的函数，但在 `f` 不可微的任何点上，该函数被定义为取值 `0` 。

```
example {f : ℝ → ℝ} {x a : ℝ} (h : HasDerivAt f a x) : deriv f x = a :=
  h.deriv

example {f : ℝ → ℝ} {x : ℝ} (h : ¬DifferentiableAt ℝ f x) : deriv f x = 0 :=
  deriv_zero_of_not_differentiableAt h
```

当然，关于 `deriv` 的许多引理确实需要可微性假设。例如，您应该思考一下在没有可微性假设的情况下，下一个引理的反例。

```
example {f g : ℝ → ℝ} {x : ℝ} (hf : DifferentiableAt ℝ f x) (hg : DifferentiableAt ℝ␣
↪g x) :
    deriv (f + g) x = deriv f x + deriv g x :=
  deriv_add hf hg
```

有趣的是，然而，存在一些语句能够通过利用函数不可微时 `deriv` 的值默认为零这一事实来避免可微性假设。因此，理解下面的语句需要确切了解 `deriv` 的定义。

```
example {f : ℝ → ℝ} {a : ℝ} (h : IsLocalMin f a) : deriv f a = 0 :=
  h.deriv_eq_zero
```

我们甚至可以在没有任何可微性假设的情况下陈述罗尔定理（Rolle’s theorem），这似乎更奇怪。

```
open Set

example {f : ℝ → ℝ} {a b : ℝ} (hab : a < b) (hfc : ContinuousOn f (Icc a b)) (hfI : f␣
↪a = f b) :
    ∃ c ∈ Ioo a b, deriv f c = 0 :=
  exists_deriv_eq_zero hab hfc hfI
```

当然，这个技巧对一般的中值定理并不适用。

```
example (f : ℝ → ℝ) {a b : ℝ} (hab : a < b) (hf : ContinuousOn f (Icc a b))
    (hf' : DifferentiableOn ℝ f (Ioo a b)) : ∃ c ∈ Ioo a b, deriv f c = (f b - f a) /␣
↪(b - a) :=
  exists_deriv_eq_slope f hab hf hf'
```

Lean 可以使用 `simp` 策略自动计算一些简单的导数。

```
example : deriv (fun x : ℝ ↦ x ^ 5) 6 = 5 * 6 ^ 4 := by simp

example : deriv sin π = -1 := by simp
```

11.2 赋范空间中的微分学

11.2.1 赋范空间

利用 **赋范向量空间**的概念，可以将微分推广到 `ℝ` 之外，该概念同时涵盖了方向和距离。我们从 **赋范群**的概念开始，它是一个加法交换群，配备了一个实值范数函数，满足以下条件。

```
variable {E : Type*} [NormedAddCommGroup E]
```

(continues on next page)

(continued from previous page)

```
example (x : E) : 0 ≤ ‖x‖ :=
  norm_nonneg x

example {x : E} : ‖x‖ = 0 ↔ x = 0 :=
  norm_eq_zero

example (x y : E) : ‖x + y‖ ≤ ‖x‖ + ‖y‖ :=
  norm_add_le x y
```

每个赋范空间都是一个度量空间，其距离函数为 $d(x, y) = \|x - y\|$，因此它也是一个拓扑空间。Lean 和 Mathlib 都知道这一点。

```
example : MetricSpace E := by infer_instance

example {X : Type*} [TopologicalSpace X] {f : X → E} (hf : Continuous f) :
    Continuous fun x ↦ ‖f x‖ :=
  hf.norm
```

为了在范数的概念中引入线性代数中的概念，我们在 `NormedAddGroup E` 的基础上添加了 `NormedSpace ℝ E` 这一假设。这表明 `E` 是 `ℝ` 上的向量空间，并且标量乘法满足以下条件。

```
variable [NormedSpace ℝ E]

example (a : ℝ) (x : E) : ‖a • x‖ = |a| * ‖x‖ :=
  norm_smul a x
```

完备的赋范空间被称为 **巴拿赫空间**。每个有限维向量空间都是完备的。

```
example [FiniteDimensional ℝ E] : CompleteSpace E := by infer_instance
```

在前面的所有示例中，我们都使用实数作为基域。更一般地说，我们可以在任何 **非平凡赋范域**上的向量空间中理解微积分。这些域配备了实值范数，该范数具有乘法性质，并且不是每个元素的范数都为零或一（等价地说，存在范数大于一的元素）。

```
example (𝕜 : Type*) [NontriviallyNormedField 𝕜] (x y : 𝕜) : ‖x * y‖ = ‖x‖ * ‖y‖ :=
  norm_mul x y

example (𝕜 : Type*) [NontriviallyNormedField 𝕜] : ∃ x : 𝕜, 1 < ‖x‖ :=
  NormedField.exists_one_lt_norm 𝕜
```

在一个非平凡赋范域上的有限维向量空间，只要域本身是完备的，那么该向量空间就是完备的。

```
example (𝕜 : Type*) [NontriviallyNormedField 𝕜] (E : Type*) [NormedAddCommGroup E]
    [NormedSpace 𝕜 E] [CompleteSpace 𝕜] [FiniteDimensional 𝕜 E] : CompleteSpace E :=
  FiniteDimensional.complete 𝕜 E
```

11.2.2 连续线性映射

现在我们来讨论赋范空间范畴中的态射，即连续线性映射。在 Mathlib 中，赋范空间 `E` 和 `F` 之间的 `𝕜` 线性连续映射的类型写作 `E →L[𝕜] F`。它们被实现为 **捆绑映射**，这意味着该类型的元素包含映射本身以及线性和连续的性质。Lean 会插入一个强制转换，使得连续线性映射可以当作函数来处理。

```
variable {𝕜 : Type*} [NontriviallyNormedField 𝕜] {E : Type*} [NormedAddCommGroup E]
  [NormedSpace 𝕜 E] {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

example : E →L[𝕜] E :=
  ContinuousLinearMap.id 𝕜 E

example (f : E →L[𝕜] F) : E → F :=
  f

example (f : E →L[𝕜] F) : Continuous f :=
  f.cont

example (f : E →L[𝕜] F) (x y : E) : f (x + y) = f x + f y :=
  f.map_add x y

example (f : E →L[𝕜] F) (a : 𝕜) (x : E) : f (a • x) = a • f x :=
  f.map_smul a x
```

连续线性映射具有算子范数，其特征在于以下性质。

```
variable (f : E →L[𝕜] F)

example (x : E) : ‖f x‖ ≤ ‖f‖ * ‖x‖ :=
  f.le_opNorm x

example {M : ℝ} (hMp : 0 ≤ M) (hM : ∀ x, ‖f x‖ ≤ M * ‖x‖) : ‖f‖ ≤ M :=
  f.opNorm_le_bound hMp hM
```

还有一种连续线性同构的 **成束**概念。这种同构的类型表示为 `E ≃L[𝕜] F`。

作为一项具有挑战性的练习，您可以证明巴拿赫-斯坦因豪斯定理，也称为一致有界性原理。该原理指出，从巴拿赫空间到赋范空间的一族连续线性映射在逐点有界的情况下，这些线性映射的范数是一致有界的。主要依据是贝尔纲定理 `nonempty_interior_of_iUnion_of_closed`（您在拓扑学章节中证明过其

一个版本)。次要依据包括 `continuous_linear_map.opNorm_le_of_shell` 、`interior_subset` 、`interior_iInter_subset` 和 `isClosed_le` 。

```
variable {𝕜 : Type*} [NontriviallyNormedField 𝕜] {E : Type*} [NormedAddCommGroup E]
  [NormedSpace 𝕜 E] {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

open Metric

example {ι : Type*} [CompleteSpace E] {g : ι → E →L[𝕜] F} (h : ∀ x, ∃ C, ∀ i, ‖g i x‖
↪≤ C) :
    ∃ C', ∀ i, ‖g i‖ ≤ C' := by
  -- 由范数 `‖g i x‖` 被 `n` 所限制那些的 `x : E` 组成的子集序列
  let e : ℕ → Set E := fun n ↦ ⋂ i : ι, { x : E | ‖g i x‖ ≤ n }
  -- 这些集合每个都是闭集
  have hc : ∀ n : ℕ, IsClosed (e n)
  sorry
  -- 并集是整个空间；这就是我们使用 `h` 的地方
  have hU : (⋃ n : ℕ, e n) = univ
  sorry
  /- 应用贝尔纲定理得出结论：存在某个 `m : ℕ` ，使得 `e m` 包含某个 `x` -/
  obtain ⟨m, x, hx⟩ : ∃ m, ∃ x, x ∈ interior (e m) := sorry
  obtain ⟨ε, ε_pos, hε⟩ : ∃ ε > 0, ball x ε ⊆ interior (e m) := sorry
  obtain ⟨k, hk⟩ : ∃ k : 𝕜, 1 < ‖k‖ := sorry
  -- 证明球内所有元素在应用任何 `g i` 后范数均不超过 `m`
  have real_norm_le : ∀ z ∈ ball x ε, ∀ (i : ι), ‖g i z‖ ≤ m
  sorry
  have εk_pos : 0 < ε / ‖k‖ := sorry
  refine ⟨(m + m : ℕ) / (ε / ‖k‖), fun i ↦ ContinuousLinearMap.opNorm_le_of_shell ε_
↪pos ?_ hk ?_⟩
  sorry
  sorry
```

11.2.3 渐近比较

定义可微性也需要渐近比较。Mathlib 拥有一个涵盖大 O 和小 o 关系的广泛库，其定义如下。打开 `asymptotics` 域允许我们使用相应的符号。在这里，我们将仅使用小 o 来定义可微性。

```
open Asymptotics

example {α : Type*} {E : Type*} [NormedGroup E] {F : Type*} [NormedGroup F] (c : ℝ)
    (l : Filter α) (f : α → E) (g : α → F) : IsBigOWith c l f g ↔ ∀𝓝 x in l, ‖f x‖ ≤
↪c * ‖g x‖ :=
```

(continues on next page)

(continued from previous page)

```
  isBigOWith_iff

example {α : Type*} {E : Type*} [NormedGroup E] {F : Type*} [NormedGroup F]
    (l : Filter α) (f : α → E) (g : α → F) : f =O[l] g ↔ ∃ C, IsBigOWith C l f g :=
  isBigO_iff_isBigOWith

example {α : Type*} {E : Type*} [NormedGroup E] {F : Type*} [NormedGroup F]
    (l : Filter α) (f : α → E) (g : α → F) : f =o[l] g ↔ ∀ C > 0, IsBigOWith C l f g␣
↪:=
  isLittleO_iff_forall_isBigOWith

example {α : Type*} {E : Type*} [NormedAddCommGroup E] (l : Filter α) (f g : α → E) :
    f ~[l] g ↔ (f - g) =o[l] g :=
  Iff.rfl
```

11.2.4 可微性

我们现在准备讨论赋范空间之间的可微函数。与基本的一维情况类似，Mathlib 定义了一个谓词 `HasFDerivAt` 和一个函数 `fderiv` 。这里的字母“f”代表 **弗雷歇（Fréchet）**。

```
open Topology

variable {𝕜 : Type*} [NontriviallyNormedField 𝕜] {E : Type*} [NormedAddCommGroup E]
  [NormedSpace 𝕜 E] {F : Type*} [NormedAddCommGroup F] [NormedSpace 𝕜 F]

example (f : E → F) (f' : E →L[𝕜] F) (x₀ : E) :
    HasFDerivAt f f' x₀ ↔ (fun x ↦ f x - f x₀ - f' (x - x₀)) =o[𝓝 x₀] fun x ↦ x - x₀␣
↪:=
  hasFDerivAtFilter_iff_isLittleO ..

example (f : E → F) (f' : E →L[𝕜] F) (x₀ : E) (hff' : HasFDerivAt f f' x₀) : fderiv 𝕜␣
↪f x₀ = f' :=
  hff'.fderiv
```

我们还有取值于多重线性映射类型 `E [×n]→L[𝕜] F` 的迭代导数，并且我们有连续可微函数。类型 `WithTop ℕ` 是在自然数 `ℕ` 的基础上添加了一个比任何自然数都大的元素 `⊤` 。因此，$\mathcal{C}^\infty$ 函数是满足 `ContDiff 𝕜 ⊤ f` 的函数 `f` 。

```
example (n : ℕ) (f : E → F) : E → E[×n]→L[𝕜] F :=
  iteratedFDeriv 𝕜 n f
```

(continues on next page)

(continued from previous page)

```
example (n : WithTop ℕ) {f : E → F} :
    ContDiff ℝ n f ↔
      (∀ m : ℕ, (m : WithTop ℕ) ≤ n → Continuous fun x ↦ iteratedFDeriv ℝ m f x) ∧
        ∀ m : ℕ, (m : WithTop ℕ) < n → Differentiable ℝ fun x ↦ iteratedFDeriv ℝ m f␣
↪x :=
  contDiff_iff_continuous_differentiable
```

存在一种更严格的可微性概念，称为 `HasStrictFDerivAt` ，它用于逆函数定理和隐函数定理的表述，这两个定理都在 Mathlib 中。在 `ℝ` 或 `ℂ` 上，连续可微函数都是严格可微的。

```
example {𝕂 : Type*} [RCLike 𝕂] {E : Type*} [NormedAddCommGroup E] [NormedSpace 𝕂 E]
↪{F : Type*}
    [NormedAddCommGroup F] [NormedSpace 𝕂 F] {f : E → F} {x : E} {n : WithTop ℕ}
    (hf : ContDiffAt 𝕂 n f x) (hn : 1 ≤ n) : HasStrictFDerivAt f (fderiv 𝕂 f x) x :=
  hf.hasStrictFDerivAt hn
```

局部逆定理是通过一种运算来表述的，该运算从一个函数生成其反函数，并且假定该函数在点 `a` 处严格可微，且其导数为同构映射。

下面的第一个例子得到了这个局部逆。接下来的一个例子表明，它确实是从左和从右的局部逆，并且它是严格可微的。

```
section LocalInverse
variable [CompleteSpace E] {f : E → F} {f' : E ≃L[𝕂] F} {a : E}

example (hf : HasStrictFDerivAt f (f' : E →L[𝕂] F) a) : F → E :=
  HasStrictFDerivAt.localInverse f f' a hf

example (hf : HasStrictFDerivAt f (f' : E →L[𝕂] F) a) :
    ∀ᶠ x in 𝓝 a, hf.localInverse f f' a (f x) = x :=
  hf.eventually_left_inverse

example (hf : HasStrictFDerivAt f (f' : E →L[𝕂] F) a) :
    ∀ᶠ x in 𝓝 (f a), f (hf.localInverse f f' a x) = x :=
  hf.eventually_right_inverse

example {f : E → F} {f' : E ≃L[𝕂] F} {a : E}
  (hf : HasStrictFDerivAt f (f' : E →L[𝕂] F) a) :
    HasStrictFDerivAt (HasStrictFDerivAt.localInverse f f' a hf) (f'.symm : F →L[𝕂]␣
↪E) (f a) :=
  HasStrictFDerivAt.to_localInverse hf
```

(continues on next page)

(continued from previous page)

```
end LocalInverse
```

这只是对 Mathlib 中微分学的一个快速浏览。该库包含许多我们未讨论过的变体。例如，在一维情况下，您可能希望使用单侧导数。在 Mathlib 中，您可以在更一般的上下文中找到实现此目的的方法；请参阅 `HasFDerivWithinAt` 或更通用的 `HasFDerivAtFilter` 。

CHAPTER

TWELVE

积分和测度论

12.1 初等积分

我们首先关注函数在 $\mathbb{R}$ 上有限区间的积分。我们可以积分初等函数。

```
open MeasureTheory intervalIntegral

open Interval
-- 这里引入了记号`[[a, b]]` 来表示区间`min a b`到`max a b`。

example (a b : ℝ) : (∫ x in a..b, x) = (b ^ 2 - a ^ 2) / 2 :=
  integral_id

example {a b : ℝ} (h : (0 : ℝ) ∉ [[a, b]]) : (∫ x in a..b, 1 / x) = Real.log (b / a)␣
↪:=
  integral_one_div h
```

微积分基本定理联系了微分和积分。下面我们给出这个定理的两个部分的一种简单情形。第一部分说明积分是微分的逆运算，第二部分说明了如何计算微元的累积。(这两个部分非常密切相关，但它们的最好版本 (没有写在这里) 并不等价。)

```
example (f : ℝ → ℝ) (hf : Continuous f) (a b : ℝ) : deriv (fun u ↦ ∫ x : ℝ in a..u, f␣
↪x) b = f b :=
  (integral_hasStrictDerivAt_right (hf.intervalIntegrable _ _) (hf.
↪stronglyMeasurableAtFilter _ _)
        hf.continuousAt).hasDerivAt.deriv

example {f : ℝ → ℝ} {a b : ℝ} {f' : ℝ → ℝ} (h : ∀ x ∈ [[a, b]], HasDerivAt f (f' x) x)
    (h' : IntervalIntegrable f' volume a b) : (∫ y in a..b, f' y) = f b - f a :=
  integral_eq_sub_of_hasDerivAt h h'
```

在 Mathlib 中也定义了卷积，并证明了卷积的基本性质。

```
open Convolution

example (f : ℝ → ℝ) (g : ℝ → ℝ) : f ⋆ g = fun x ↦ ∫ t, f t * g (x - t) :=
  rfl
```

12.2 测度论

Mathlib 中积分的数学基础是测度论。甚至前一节的初等积分实际上也是 Bochner 积分。Bochner 积分是 Lebesgue 积分的推广，目标空间可以是任意的 Banach 空间，不一定是有限维的。

测度论的第一部分是集合的 σ -代数的语言，被称作 * 可测集 *。*MeasurableSpace* 类型族提供了带有这种结构的类型。空集 *empty* 和单元素集 *univ* 是可测的，可测集的补集是可测的，可数交和可数并是可测的。注意，这些公理是冗余的；如果你 *#print MeasurableSpace*，你会看到 Mathlib 用来构造可测集的公理。可数性条件可以使用 *Encodable* 类型族来表示。

```
variable {α : Type*} [MeasurableSpace α]

example : MeasurableSet (∅ : Set α) :=
  MeasurableSet.empty

example : MeasurableSet (univ : Set α) :=
  MeasurableSet.univ

example {s : Set α} (hs : MeasurableSet s) : MeasurableSet (s⁻) :=
  hs.compl

example : Encodable ℕ := by infer_instance

example (n : ℕ) : Encodable (Fin n) := by infer_instance

variable {ι : Type*} [Encodable ι]

example {f : ι → Set α} (h : ∀ b, MeasurableSet (f b)) : MeasurableSet (⋃ b, f b) :=
  MeasurableSet.iUnion h

example {f : ι → Set α} (h : ∀ b, MeasurableSet (f b)) : MeasurableSet (⋂ b, f b) :=
  MeasurableSet.iInter h
```

如果一个类型是可测的，那么我们就可以测量它。字面上，对配备 σ -代数的集合（或者类型）的测量是一个函数，它是从可测集到扩展（即允许无穷）非负实数的函数，并且满足可数无交并集合上可加性。在 Mathlib 中，我们不希望每次测量集合时都带着写一个集合可测。因此我们把这个测度推广到任何集合 s，作为包含

s 的可测集合的测度的最小值。当然，许多引理仍然需要可测假设，但不是全部。

```
open MeasureTheory
variable {μ : Measure α}

example (s : Set α) : μ s = ⨅ (t : Set α) (_ : s ⊆ t) (_ : MeasurableSet t), μ t :=
  measure_eq_iInf s

example (s : ι → Set α) : μ (⋃ i, s i) ≤ ∑' i, μ (s i) :=
  measure_iUnion_le s

example {f : ℕ → Set α} (hmeas : ∀ i, MeasurableSet (f i)) (hdis : Pairwise (Disjoint␣
↪on f)) :
    μ (⋃ i, f i) = ∑' i, μ (f i) :=
  μ.m_iUnion hmeas hdis
```

一旦一个类型有了与它相关联的测度，我们就说，如果性质 *P* 只在一个测度为 0 的集合上失效，则 *P* “几乎处处”成立 (almost everywhere, ae)。几乎处处的性质集合形成了一个过滤器 (filter)，但是 Mathlib 引入了特殊的符号来表示一个性质几乎处处成立。

```
example {P : α → Prop} : (∀ᵐ x ∂μ, P x) ↔ ∀ᵐ x in ae μ, P x :=
  Iff.rfl
```

12.3 积分

现在我们有了测度和可测空间，我们就可以考虑积分了。正如前文所讲，Mathlib 使用非常一般的积分记号，支持任意的 Banach 空间。像往常一样，我们不希望我们的记号带有假设，所以我们这样约定：如果函数不可积，那么积分等于零。大多数与积分有关的引理都有可积性假设。

```
section
variable {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E] [CompleteSpace E] {f :␣
↪α → E}

example {f g : α → E} (hf : Integrable f μ) (hg : Integrable g μ) :
    ∫ a, f a + g a ∂μ = ∫ a, f a ∂μ + ∫ a, g a ∂μ :=
  integral_add hf hg
```

作为我们做出的各种约定之间复杂交互的一个例子，让我们看看如何积分常值函数。回顾一下测度 μ 是在扩展的非负实数 $\mathbb{R}\geq 0\infty$ 上取值的，存在一个函数 *ENNReal.toReal* : $\mathbb{R}\geq 0\infty \to \mathbb{R}$ 把无穷点 $\square$ 映到 0。对任意 *s* : *Set* α，如果 $\mu\ s = \square$，则非零的常值函数在 *s* 上不可积，因此根据约定积分值为 0，刚好是 *(μ s).toReal* 的结果。所以我们有下面的引理。

```
example {s : Set α} (c : E) : ∫ x in s, c ∂μ = (μ s).toReal • c :=
  setIntegral_const c
```

现在我们快速地解释如何获得积分理论中最重要的定理，从控制收敛定理开始（dominated convergence theorem）。Mathlib 中有几个版本，这里我们只展示最基本的一个。

```
open Filter

example {F : ℕ → α → E} {f : α → E} (bound : α → ℝ) (hmeas : ∀ n,␣
↪AEStronglyMeasurable (F n) μ)
    (hint : Integrable bound μ) (hbound : ∀ n, ∀ᵐ a ∂μ, ‖F n a‖ ≤ bound a)
    (hlim : ∀ᵐ a ∂μ, Tendsto (fun n : ℕ ↦ F n a) atTop (𝓝 (f a))) :
    Tendsto (fun n ↦ ∫ a, F n a ∂μ) atTop (𝓝 (∫ a, f a ∂μ)) :=
  tendsto_integral_of_dominated_convergence bound hmeas hint hbound hlim
```

还有一个积类型上的积分的 Fubini 定理：

```
example {α : Type*} [MeasurableSpace α] {μ : Measure α} [SigmaFinite μ] {β : Type*}
    [MeasurableSpace β] {ν : Measure β} [SigmaFinite ν] (f : α × β → E)
    (hf : Integrable f (μ.prod ν)) : ∫ z, f z ∂ μ.prod ν = ∫ x, ∫ y, f (x, y) ∂ν ∂μ :=
  integral_prod f hf
```

有一个非常一般的版本的卷积适用于任何连续双线性形式。

```
open Convolution

variable {𝕜 : Type*} {G : Type*} {E : Type*} {E' : Type*} {F : Type*}␣
↪[NormedAddCommGroup E]
  [NormedAddCommGroup E'] [NormedAddCommGroup F] [NontriviallyNormedField 𝕜]␣
↪[NormedSpace 𝕜 E]
  [NormedSpace 𝕜 E'] [NormedSpace 𝕜 F] [MeasurableSpace G] [NormedSpace ℝ F]␣
↪[CompleteSpace F]
  [Sub G]

example (f : G → E) (g : G → E') (L : E →L[𝕜] E' →L[𝕜] F) (μ : Measure G) :
    f ⋆[L, μ] g = fun x ↦ ∫ t, L (f t) (g (x - t)) ∂μ :=
  rfl
```

最后，Mathlib 有一个非常一般的换元公式。下面的命题中，*BorelSpace E* 意为由开集 E 生成的 E 上的 σ-代数，*IsAddHaarMeasure μ* 意为测度 μ 是左不变的 (left-invariant)，在紧集上有限，在开集上为正数。

```
example {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E] [FiniteDimensional ℝ E]
    [MeasurableSpace E] [BorelSpace E] (μ : Measure E) [μ.IsAddHaarMeasure] {F :␣
↪Type*}
    [NormedAddCommGroup F] [NormedSpace ℝ F] [CompleteSpace F] {s : Set E} {f : E → E}
    {f' : E → E →L[ℝ] E} (hs : MeasurableSet s)
    (hf : ∀ x : E, x ∈ s → HasFDerivWithinAt f (f' x) s x) (h_inj : InjOn f s) (g : E␣
↪→ F) :
    ∫ x in f '' s, g x ∂μ = ∫ x in s, |(f' x).det| • g (f x) ∂μ :=
  integral_image_eq_integral_abs_det_fderiv_smul μ hs hf h_inj g
```

CHAPTER

THIRTEEN

INDEX

INDEX

L

M

N

O

P

R

S

T

U

V