Day 4

4.1 集合与子类型

4.1.1 集合

初学者使用 Lean 时往往容易产生这样一个直觉：可以将类型论中的“类型”类比为集合论中的“集合”，将类型的元素（Term）类比为集合中的元素。这种直觉虽然在一定程度上是自然的，但在严谨的数学形式化中，二者有着本质的区别。

我们以自然数为例。在标准集合论中，表达式 $(1,3) \in \mathbb{N}$ 是一个合法的命题（虽然它的真值为假）。这是因为在集合论中，我们可以讨论任何对象是否属于某个集合。然而，在类型论中，表达式 `(1, 3) : ℕ` 不仅是错误的，甚至是不合法的语法（ill-formed）。类型检查是在编译期进行的，这意味着我们无法构造出“类型不匹配”的表达式。因此，Lean 中的“类型”缺乏集合论中那种“从全集中选取特定子集”的灵活性。

为了在 Lean 中进行标准的集合论推理，我们需要在类型之上显式地定义“集合”的概念。

集合的定义

为了给 Lean 中的集合下定义，我们不妨从最基本的**属于关系**（membership）出发进行反向推导。

假设我们要在类型 `α` 上定义一个集合 `s`。对于任意元素 `a : α`，表达式 `a ∈ s` 在数学上是一个命题（可能是真，也可能是假）。根据 Lean 的类型规则，我们可以得出两点事实：

- 表达式 `a ∈ s` 的类型必须是 `Prop`。
- 这里的 `s` 实际上扮演了一个“判定者”的角色：它接收一个元素 `a`，并返回一个关于该元素的命题。

因此，从函数的角度看，集合 `s` 的行为完全等同于一个函数：

$$\text{输入类型：}\alpha \longrightarrow \text{输出类型：}\texttt{Prop}$$

这意味着，**集合本质上就是类型 `α` 上的一个谓词（Predicate）**。

- 如果函数 `s` 作用于 `a` 得到的命题 `s a` 为真，我们说 `a` 在集合中；
- 如果命题 `s a` 为假（或无法证明），则 `a` 不在集合中。

基于这种逻辑，Lean 直接将集合定义为这个函数类型本身。为了使代码更具数学语义，我们定义 `Set α` 作为 `α → Prop` 的别名。

```
variable (α : Type)

-- Set α 的定义本质上就是 α → Prop
#check Set α      -- Set α : Type
```

```
#print Set        -- def Set.{u} : Type u → Type u := fun α => α → Prop

example : (Set α) = (α → Prop) := rfl

variable (s : Set α)
#check s          -- s : Set α
```

在上述代码中，`#print` 命令展示了 `Set` 的具体定义是 `fun α => α → Prop`。即：当我们写下 `s : Set α` 时，我们实际上是在定义一个函数 `s : α → Prop`。这个定义简洁地捕捉了集合论中“外延公理”的精神：一个集合完全由它所包含的元素（即满足性质的元素）所决定。

根据上述定义，给定一个类型 `α`，我们如何定义一个包含该类型全部元素的“全集”?（答案将在后续给出）

集合的性质与运算

基于 `Set α := α → Prop` 这一核心定义，我们可以自然地导出集合论中的常见概念。所有的集合运算本质上都是逻辑运算的体现。

1. **属于关系 (Membership)**

对于元素 `a : α` 和集合 `s : Set α`，属于关系 `a ∈ s` 定义为性质 `s` 作用于参数 `a`，即 `s a`。

```
variable (α : Type) (a : α) (s : Set α)

#check a ∈ s      -- a ∈ s : Prop

-- 属于关系仅仅是函数应用的语法糖
example : (a ∈ s) = (s a) := rfl
```

Remark：如果你在编辑器中按住 `Ctrl` 点击 `∈` 符号，可能会跳转到 `Membership.mem` 等复杂的类定义。这是 Lean 为了支持多种结构的属于关系（如列表、数组等）所做的抽象，但在集合（Set）的语境下，它就是简单的函数应用 `s a`。

同理，不属于关系 `a ∉ s` 定义为属于关系的逻辑否定：

```
example : (a ∉ s) = ¬(a ∈ s) := rfl
```

2. **子集关系 (Subset Relation)**

在数学上，如果集合 s 中的任意元素都属于集合 t，我们就称 s 是 t 的子集，记作 $s \subseteq t$。

在 Lean 中，我们希望定义同样的表达式 `s ⊆ t`。由于这也是一个关于集合的断言，其类型应为 `Prop`。结合子集的数学含义——“对于任意 x，若 $x \in s$ 则 $x \in t$”，我们可以直接将其转化为 Lean 的全称命题形式。

```
variable (t : Set α)

#check s ⊆ t      -- s ⊆ t : Prop

-- 子集的定义等价于全称命题
example : (s ⊆ t) = (∀ {x}, x ∈ s → x ∈ t) := rfl
```

Remark: 关于双花括号

细心的读者可能注意到了，上述定义中全称量词的变量 `x` 被 `⦃ ⦄` 括了起来，而不是我们常见的花括号 `{ }`。这被称为**严格隐式参数** (Strict-implicit binder)。

为了理解它的作用，我们需要对比一下它与普通隐式参数的区别：

- **普通隐式参数** `{x}`：只要你提到了包含该参数的函数名，Lean 就会立刻尝试推断 `x` 是什么。如果推断不出，它就会生成一个元变量（metavariable，通常显示为 `?m`）占位。
- **严格隐式参数** `⦃x⦄`：Lean 会等到你提供了后续的显式参数（在这里是 `x ∈ s` 这一项）之后，才去推断 `x`。

在谓词逻辑中，使用 `⦃x⦄` 的好处在于：当我们只引用子集关系的证明 `h : s ⊆ t` 而不传入参数时，Lean 不会自作聪明地把 `h` 展开成带问号的 `?m ∈ s → ?m ∈ t`，而是保持 `h` 原本整洁的样子。这使得 Infoview 中的显示更加清晰。

那么，当我们拥有一个子集关系的证明 `h : s ⊆ t` 时，该如何使用它呢？根据定义，`s ⊆ t` 本质上是一个函数：它接收“元素 x”和“$x \in s$ 的证据”，返回“$x \in t$ 的证据”。由于 x 是隐式参数（无论是严格还是普通），我们在使用时只需要提供后者。

```
-- 假设我们知道 s 是 t 的子集，且 x 在 s 中
example (h : s ⊆ t) (x : α) (mem_s : x ∈ s) : x ∈ t := by
  -- h 的类型是 ∀ ⦃x⦄, x ∈ s → x ∈ t
  -- 我们像调用函数一样调用 h，只需传入 mem_s，Lean 会自动推断出 x
  exact h mem_s
```

3. **交集 (Intersection)**

我们定义两个集合 s 和 t 的交集 $s \cap t$ 为：一个元素属于该交集，当且仅当它同时属于 s 和 t。

```
#check s ∩ t      -- s ∩ t : Set α

-- 交集的定义：x ∈ s ∧ x ∈ t
example : (s ∩ t) = (fun x => x ∈ s ∧ x ∈ t) := rfl
```

Lean 中，交集本质上就是两个谓词的**合取**（Logical And, ∧）但这里交集运算从定义上来讲并不是可交换的。即 `fun x => x ∈ s ∧ x ∈ t` 和 `fun x => x ∈ t ∧ x ∈ s` 虽然逻辑等价，但参数顺序不同，因此不是**定义上相等**的。

```
-- rfl会失败，因为左右两边并不是 definitionally equal
example : (s ∩ t) = (fun x => x ∈ t ∧ x ∈ s) := by rfl
```

由于元素 `a` 与交集 `s ∩ t` 之间的属于关系 `a ∈ s ∩ t`，展开后就是 `a ∈ s ∧ a ∈ t`。因此我们可以按照处理“与”命题（And）的方法来处理它。

```
-- 1. 属于关系展开为合取
example : (a ∈ s ∩ t) = (a ∈ s ∧ a ∈ t) := by rfl

-- 2. 从交集中提取信息（拆解）
example (h : a ∈ s ∩ t) : a ∈ s := by
  rcases h with ⟨mem_s, mem_t⟩  -- 将 h 拆解为两个假设
  exact mem_s

-- 或者直接访问项
example (h : a ∈ s ∩ t) : a ∈ s := h.left

-- 3. 证明元素属于交集（构造）
```

```
example (h₁ : a ∈ s) (h₂ : a ∈ t) : a ∈ s ∩ t := by
  constructor       -- 目标分裂为两个子目标
  • exact h₁        -- 证明 a ∈ s
  • exact h₂        -- 证明 a ∈ t
```

4. **并集 (Union)**

并集 `s ∪ t` 由属于 `s` 或者属于 `t` 的元素组成。逻辑上，这对应于命题的 ** 析取 **（Or, `∨`）。

```
example : (s ∪ t) = (fun x => x ∈ s ∨ x ∈ t) := by rfl
```

与交集的情形类似, 一个元素 `a : α` 与并集 `s ∪ t` 之间的属于关系实际上就是一个 “或” 命题 `a ∈ s ∨ a ∈ t`，通常需要使用 `rcases` 进行分类讨论：

```
-- 从并集中推理（分类讨论）
example (r : Set α) (h : a ∈ s ∪ t) (h₁ : s ⊆ r) (h₂ : t ⊆ r) : a ∈ r := by
  rcases h with mem_s | mem_t
  • exact h₁ mem_s -- 情况 1: a ∈ s
  • exact h₂ mem_t -- 情况 2: a ∈ t

-- 构造并集成员
example (h : a ∈ s) : a ∈ s ∪ t := by
  left -- 选择左侧分支
  exact h
```

5. **差集 (Difference)**

差集 `s \ t` 包含属于 `s` 但不属于 `t` 的元素。逻辑上，这是 “与” 和 “非” 的组合。

```
example : (s \ t) = (fun x => x ∈ s ∧ x ∉ t) := rfl

example (h : a ∈ s \ t) : a ∉ t := h.right

example (h₁ : a ∈ s) (h₂ : a ∉ t) : a ∈ s \ t := ⟨h₁, h₂⟩
```

6. **补集 (Complement)**

补集 s^c 包含所有不属于 `s` 的元素。逻辑上，这对应于命题的**否定**。

```
#check sᶜ         -- sᶜ : Set α
example : sᶜ = (fun x => x ∉ s) := rfl
```

处理补集即处理否定命题，常涉及反证或通过否定引入法（`exfalso`）：

```
-- 矛盾推导
example (h₁ : a ∈ s) (h₂ : a ∈ sᶜ) : 1 = 2 := by
  exfalso
  exact h₂ h₁

-- 利用子集关系推导补集性质
example (h : a ∉ t) (h₂ : s ⊆ t) : a ∈ sᶜ := by
  intro mem_s     -- 假设 a ∈ s
```

```
  apply h         -- 目标转变为证明 a ∈ t
  exact h₂ mem_s  -- 利用子集关系得证
```

全集与空集

现在我们来回答本节前面提出的问题：**如何定义一个包含类型 α 中全部元素的集合？**

回想集合的定义：类型 α 上的集合 s 本质上是一个判定函数 s : α → Prop。如果我们希望集合包含所有元素，意味着这个判定函数对于任何输入 x : α 都应当返回“真”。在 Lean 中，最简单的恒真命题是 True（其证明为 trivial）。因此，全集可以定义为 fun _ => True。

Lean 标准库中已经为我们定义好了这个概念，记作 Set.univ (Universal Set)。

```
variable {α : Type}
-- 定义全集
def SET : Set α := fun _ => True

-- 证明任意元素都属于该集合
example : ∀ x : α, x ∈ SET := by
  intro x
  exact trivial

-- Lean 内置的全集
#check Set.univ     -- Set.univ : Set α

example : ∀ x : α, x ∈ Set.univ := by
  intro x
  exact trivial
```

相对地，我们可以定义数学中的**空集**。不难想到，空集对应的性质应当是对所有元素都为“假”。Lean 使用符号 (∅ : Set α) 来表示空集，其定义等价于 fun _ => False。

```
def EMPTY : Set α := fun _ => False

-- Lean 内置的空集符号
#check (∅ : Set α)

-- 证明没有任何元素属于空集
example : ∀ x : α, x ∉ (∅ : Set α) := by
  intro x mem_empty
  -- mem_empty 的类型是 x ∈ ∅, 即 False
  exact mem_empty
```

集合描述法 (Set Builder Notation)

为了贴近数学的自然语言习惯，Lean 提供了一套强大的**集合描述法**记号。给定类型 α 和其上的性质 p : α → Prop，我们可以使用 {x | p x} 来定义由满足性质 p 的元素构成的集合。

```
variable (p : α → Prop)
#check {x : α | p x}      -- {x | p x} : Set α
```

在这个记号中，竖线左边代表集合中的元素变量，右边代表该元素需满足的条件。

此外，Lean 还支持更高级的语法糖，允许我们在竖线左侧写构造表达式。例如，如果我们想定义“所有偶数的集合”，标准的写法是 `{x | ∃ n, 2 * n = x}`。但在 Lean 中，我们可以简写为 `{2 * n | n}`。

```
-- 标准写法：所有满足“存在 n 使得 x = 2n”的 x
#check {x | ∃ n, 2 * n = x}

-- 语法糖写法：由 n 构造出的 2n 构成的集合
#check {2 * n | n}
```

注：第二种写法 `{2 * n | n}` 在底层会被 Lean 自动展开为第一种带存在量词的形式。这种写法虽然简洁，但在某些复杂类型推断场景下可能会受限。

函数的像与原像

函数与集合之间的交互是数学中的常见操作。在 Lean 中，我们主要关注**像**（Image）和**原像**（Preimage）。

1. **像 (Image)** 对于函数 `f : α → β` 和集合 `s : Set α`，`f` 在 `s` 上的像记作 `f '' s`。它是一个 `β` 上的集合，包含所有形式为 `f a`（其中 `a ∈ s`）的元素。从逻辑定义上看，这涉及到一个**存在量词**：$y \in f(s) \iff \exists x \in s, f(x) = y$。

   ```
   variable (β : Type) (f : α → β) (s : Set α)

   #check f '' s     -- f '' s : Set β

   example (b : β) : (b ∈ f '' s) = (∃ a ∈ s, f a = b) := rfl
   ```

2. **原像 (Preimage)** 对于函数 `f : α → β` 和集合 `t : Set β`，`t` 在 `f` 下的原像记作 `f ⁻¹' t`。它是一个 `α` 上的集合，包含所有映射后落在 `t` 中的元素。原像的定义通常比像更简单，因为它直接利用了函数的性质，不涉及存在量词：$x \in f^{-1}(t) \iff f(x) \in t$。

   ```
   variable (t : Set β)

   #check f ⁻¹' t    -- f ⁻¹' t : Set α

   example (a : α) : (a ∈ f ⁻¹' t) = (f a ∈ t) := rfl
   ```

有界量词 (Bounded Quantifiers)

在数学中，我们经常写 $\forall x > 0, P(x)$ 或 $\exists x > A, P(x)$。在逻辑底层，这两者的处理方式是不同的：

- **全称量词与蕴含：**`∀ x ∈ s, p x` 实际上是 `∀ x, x ∈ s → p x` 的缩写。即：“对于任意 x，**如果** x 在 s 中，**那么** px 成立”。
- **存在量词与合取：**`∃ x ∈ s, p x` 实际上是 `∃ x, x ∈ s ∧ p x` 的缩写。即：“存在一个 x，它**既**在 s 中，**且**满足 px”。

这对应 Lean 中常用的一种语法糖：有界量词

```
variable (p : ℕ → Prop)

-- 全称量词的限制条件转化为蕴含 (→)
```

```
example : (∀ x > 0, p x) = (∀ x, x > 0 → p x) := by rfl

-- 存在量词的限制条件转化为合取 (∧)
example : (∃ x > 0, p x) = (∃ x, x > 0 ∧ p x) := by rfl
```

4.2 子类型 (Subtype) 与类型转换

集合与类型的界限

按照我们最初介绍的类型论原则，类型 `α : Type` 上的一个集合 `s : Set α` 本身是一个**项**（Term），而不是一个**类型**（Type）。从严格的语法角度来看，表达式 `a : s`（意指 `a` 是类型 `s` 的一个实例）应当是无意义的，因为 `:`右侧必须是一个类型。

然而，如果我们尝试在 Lean 中通过代码实现这一看似错误的操作，会发生什么呢？

```
variable (α : Type) (s : Set α)

variable (a : s)    -- 竟然没有报错!!

#check a            -- a : ↑s
```

我们惊讶地发现，Lean 接受了 `a : s` 这样的表达式。但请注意 `#check` 的输出结果：变量 `a` 的类型在 Infoview 中被显示为 `↑s`，而不是简单的 `s`。这里的上箭头 `↑` 暗示了某种隐式的转换正在发生。

4.2.1 子类型的定义与结构

为了理解上述现象，我们需要引入**子类型**（Subtype）的概念。

对于一个类型 `α` 和其上的性质 `p : α → Prop`，我们可以从 `α` 中筛选出所有满足性质 `p` 的元素，构成一个新的类型。我们将其记作 `{x : α // p x}`，称之为 `α` 的一个子类型。

现在回到集合的问题上。回顾定义，集合 `s : Set α` 本质上就是谓词 `s : α → Prop`。因此，Lean 通过一种被称为**强制类型转换**（Coercion to Sort）的机制，自动将集合 `s` 提升为了子类型 `{x : α // x ∈ s}`。这也解释了为什么 `a : ↑s` 是合法的——它实际上代表 `a : {x : α // x ∈ s}`。

子类型的内部结构

一个子类型的元素 `a : {x : α // p x}` 实际上是一个包含两个分量的“包”：

1. **数值分量** `a.val : α`。这是来自原类型 `α` 的具体元素。
2. **性质分量** `a.property : p a.val`。这是证明该元素满足性质 `p` 的证据（Proof）。

Lean 为访问这两个分量提供了简便的记号（语法糖）：`a.1` 等价于 `a.val`，`a.2` 等价于 `a.property`。

```
variable (α : Type) (p : α → Prop) (a : {x : α // p x})

#check a.val        -- ↑a : α
#check a.property   -- a.property : p ↑a
#check a.1          -- ↑a : α
#check a.2          -- a.property : p ↑a

-- 底层定义
#check Subtype.val      -- ... (self : Subtype p) : α
#check Subtype.property -- ... (self : Subtype p) : p ↑self
```

实战与陷阱：多层子类型

让我们通过实数集的例子来深入理解。假设我们想定义非负实数类型，我们可以使用性质 `fun r => r ≥ 0`：

```
#check {x : ℝ // x ≥ 0}     -- { x // x ≥ 0 } : Type
variable (y : {x : ℝ // x ≥ 0})

#check y.val                -- ↑y : ℝ
#check y.property           -- y.property : ↑y ≥ 0
```

如果我们要更进一步，从这个“非负实数类型”中再取出“正实数”构成一个新的类型，会发生什么？这在定义上当然是可行的，但需要格外小心谓词的写法。

错误写法：`{x : ({x : ℝ // x ≥ 0}) // x > 0}` 这里的 `x` 本身是一个子类型（非负实数），我们不能直接拿它和 `0`（实数）做比较。

正确写法：`{x : ({x : ℝ // x ≥ 0}) // x.val > 0}` 我们需要取出 `x` 所代表的实数值 `x.val` 来与 `0` 进行比较。

```
-- 定义嵌套子类型：非负实数中的正数
#check {x : {x : ℝ // x ≥ 0} // x.val > 0}  -- { x // ↑x > 0 } : Type
variable (z : {x : {x : ℝ // x ≥ 0} // x.val > 0})

-- 第一层解包：得到非负实数
#check z.val            -- ↑z : { x // x ≥ 0 }
-- 第一层性质：证明该非负实数的“值”大于 0
#check z.property       -- z.property : ↑↑z > 0

-- 第二层解包：得到原始实数
#check z.val.val        -- ↑↑z : ℝ
-- 第二层性质：证明该实数非负
#check z.val.property   -- (↑z).property : ↑↑z ≥ 0
```

嵌套子类型的代价

虽然 Lean 允许通过 `z.val.val` 这种方式层层剥洋葱，但在实践中，**不推荐**定义这种多层嵌套的子类型。层数越多，访问底层数据和性质就越繁琐（如 `x.val.val.val...`），这会给证明和代码维护带来巨大的认知负担。

特别是当我们在集合上定义集合时，很容易陷入这个陷阱：比如对于类型 `α` 来说, 我们令 `s : Set α` 是 `α` 上的一个集合, 那么 `t : Set s` 相当于先把 `s` 视作 `α` 的子类型, 然后再定义了这个子类型上的一个集合 `t`, 这时候, 如果我们使用 `x : t` 把 `t` 再变成 `s` 的子类型:

```
variable (α : Type) (s : Set α) (t : Set s) (x : t)

-- x 是 t 的元素，t 是 s 上的集合
-- 要想拿到 x 对应的 α 中的元素，我们需要剥两层皮
#check x.val.val        -- ↑↑x : α
#check x.val.property   -- (↑x).property : ↑↑x ∈ s
```

实践建议：在使用 `term : type` 这种表达式时，请务必审视 `type` 的本质。如果需要表达“满足性质 A 且满足性质 B 的元素”，通常更好的做法是直接在原始类型 `α` 上定义一个组合性质 `fun x => A x ∧ B x`，从而构造一个单层的子类型，而不是通过嵌套构建。

4.2.2 类型转换 (Coercion)

现在我们来解释上一节中遇到的箭头符号 `↑` 到底是什么意思。

问题引入：数学直觉与类型严格性

在常规数学中，我们习惯于存在一个自然的包含链：$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$。这意味着一个自然数（如 `1`）在任何需要实数的场合，都可以被无条件地当作实数 `1` 来使用。然而，Lean 是建立在严格类型论基础上的。在 Lean 中，`1 : ℕ` 和 `1 : ℝ` 是完全不同的两个对象，它们的类型截然不同。

解决方案：隐式强制转换

为了在保持类型系统严格性的同时提供便利，Lean 引入了**强制类型转换**（Coercion）机制。

当 Lean 遇到类型不匹配的表达式时（例如需要 `ℝ` 却得到了 `ℕ`），它并不会立即报错，而是会尝试在后台搜索是否存在一个已注册的“转换函数”，能将当前类型转化为目标类型。

如果找到了这样的函数，Lean 就会自动插入它。在 Infoview 的显示中，这种自动插入的转换函数通常被标记为一个向上的箭头 `↑`。

```
 -- 定义一个实数上的函数
def f' : ℝ → ℝ := fun x => x + 1
-- 数学上 f(1) = 2 是直接的，但在 Lean 中，1 默认为自然数
#check f' (1 : ℕ)    -- f ↑1 : ℝ
```

在上例中，Lean 自动插入了从 $\mathbb{N}$ 到 $\mathbb{R}$ 的转换函数，`↑1` 实际上代表“应用了转换函数后的 1”。这个 `↑` 并不是一个具体的函数名，而是**发生了一次类型转换**的视觉标识。

在 Lean 的 Infoview 中，根据类型转换的目标不同，我们会看到三种不同形状的箭头。它们均表示发生了类型转换，但具体含义有所不同.

1. `↑`. 这个箭头符号没有更多特殊含义，它只表示在箭头处发生了一个类型转换，至于转换成了什么类型，我们无法从记号上直接看出. 值得注意的一点是，`↑x + ↑y` 表达式和 `↑(x + y)` 表示的含义不同，前者表示先将 `x` 和 `y` 转到某个目标类型后，再做加法，而后者表示现在原类型上将 `x` 和 `y` 相加，然后把相加的结果转到目标类型.

2. `↥`. 这个箭头符号也表示发生了类型转换，并且类型转换的目标类型是 `Type`，即把原表达式变成了一个类型.

3. `↟`. 这个箭头符号也表示发生了类型转换，并且类型转换的目标类型是一个函数类型.

实践建议：显式标注

虽然隐式转换非常方便，但在复杂的数学表达式中，过度依赖 `↑` 可能会导致 Lean 推断出的目标类型不符合你的预期，或者让表达式难以阅读。

在实际工作中，如果类型转换的目标不明显，我们建议使用**显式类型标注**（Type Ascription）来指明意图。

```
-- 显式指明我们想要有理数
#check (1 : ℚ)

-- 显式指明我们想要实数
#check (1 : ℝ)
```

4.2.3 类型转换处理 tactic

在处理涉及自然数、整数、有理数等混合运算时，类型转换（Coercion, `↑`）经常会让证明变得繁琐。Lean 提供了两个专门的 tactic 来处理这种情况，它们的行为恰好相反，分别对应着“化简”与“展开”两种策略。

`norm_cast` **向外提**。它会尝试把所有的类型转换符号 `↑` 移动到表达式的最外层。例如，将 `↑x + ↑y` 转化为 `↑(x + y)`。通常情况下，这是我们首选的策略。因为它倾向于消除表达式内部的类型转换，将问题还原为原始类型（如 `ℤ` 或 `ℕ`）上的纯粹运算，从而简化证明。

`push_cast` **向内推**。它会尝试把类型转换符号 `↑` 推入到表达式的内部，作用于每一个子项。例如，将 `↑(x + y)` 转化为 `↑x + ↑y`。虽然这看起来会让表达式变复杂（增加了 `↑` 的数量），但在某些特定场景下——例如当我们需要在目标类型（如 `ℚ`）上应用某些特定的引理，或者需要对齐等式两边的结构时——它是必不可少的。

```
-- norm_cast 示例：消除类型转换，回到整数加法
example (a b : ℤ) (h : (a : ℚ) + b < 10) : a + b < 10 := by
  norm_cast at h


example (a b : ℤ) (h : (a : ℚ) + (b : ℚ) = 1) : ((a + b : ℤ) : ℚ) = 1 := by
 -- ⊢ ↑(a + b) = 1
  push_cast
  -- ⊢ ↑a + ↑b = 1
  exact h
```

从上面的例子可以看到，这两个 tactic 不仅可以作用于目标（Goal），也可以通过 `at h` 作用于假设（Hypothesis），非常灵活。

4.3 更多自动化 Tactic

本节将介绍几个通用的自动化工具。它们不局限于集合论，在处理涉及数值、代数运算的问题时也非常常用。

4.3.1 外延性 (ext)

在数学中，许多对象是由其“成分”唯一确定的。例如：

- 两个集合相等，当且仅当它们包含相同的元素。
- 两个函数相等，当且仅当它们对所有输入都有相同的输出。
- 两个有序对相等，当且仅当它们的每一个分量都相等。

`ext` 用于将对象的相等性证明转化为其成分的相等性证明。Lean 会自动在库中查找标记为 `@[ext]` 的外延性引理并应用它。

```
variable (α β : Type)

-- 1. 用于集合：转化为元素互推
example (s t : Set α) (h : ∀ x, x ∈ s ↔ x ∈ t) : s = t := by
  ext x
  exact h x

-- 2. 用于函数：转化为函数值相等
example (f g : α → β) (h : ∀ x, f x = g x) : f = g := by
  ext x
  exact h x

-- 3. 用于结构体（如子群）：转化为底层集合相等 (子群相等本质上就是它们包含的元素相等)
example {G : Type*} [Group G] (H K : Subgroup G) (h : ∀ x, x ∈ H ↔ x ∈ K) : H = K := by
  ext x
  exact h x
```

注：

- 名字 `x` 是可选的: 可以定义成其他名字, 也可以不加名字, 后者仍然会自动生成一个被 † 标记的不可访问的变量名.
- 与 `funext` 的区别：`funext` 是专门用于证明**函数**相等性的策略（Function Extensionality）。相比之下，`ext` 是一个更强大的通用工具。适用于函数、集合、结构体等所有注册了 `@[ext]` 的类型。另外 `ext` 通常会递归地应用，一次性剥离多层结构，而 `funext` 通常只剥离一层参数。

4.3.2 数值判定与代数化简

这一类 Tactic 主要用于处理具体的数值符号判定和代数式的结构化简。

`positivity` 用于自动化证明形如 `0 ≤ x`, `0 < x` 和 `x ≠ 0` 的情形. 这是一个结束性 tactic, 因此, 它要么解决了目标, 要么报告一个错误. 其作用原理有些复杂, 通俗地讲, `positivity` 会对复杂表达式结合变量数值下界进行解析, 拆分成一个个简单的以上三种情形的数值表达式, 然后分别进行证明.

```
example {a b : ℝ} (h : b ≠ 0) : 0 < a^2 + |b| + 1 := by
  positivity
```

面对上述目标，`positivity` 会执行以下步骤：

1. 识别出顶层运算是加法（$x+y+z$）。要证明和为正，策略试图证明每一项均非负，且至少有一项严格为正。
2. 识别出这是**平方**形式。根据内置规则（平方数总是非负），判定 $a^2 \geq 0$。
3. 识别出这是**绝对值**形式。虽然绝对值天然只保证 ≥ 0，但策略扫描了**局部上下文**，发现了假设 $h : b \neq 0$。结合绝对值的性质（非零数的绝对值严格为正），判定 $|b| > 0$。
4. 识别出这是**数字字面量**。策略内部调用 `norm_num` 进行计算，判定 $1 > 0$。
5. 将结果合并，$(\geq 0) + (> 0) + (> 0)$ 显然是严格大于 0 的。

`positivity` 的能力依赖于表达式的结构。对于像 $0 < 5 - 2$ 这种涉及减法的表达式，由于“正数减正数”符号不确定，它通常无法直接处理（除非配合 `norm_num` 先进行化简）。

```
example (x : ℝ) (hx : 0 < x) : 0 < 5 - (-x) := by
  -- positivity  -- 失败！因为它看不出 5 - (-x) 是正的
  norm_num       -- 化简为 0 < 5 + x
  positivity     -- 成功
```

`field_simp` 用于化简包含除法（逆元）的表达式。它通常会将分式通分，消除分母。常与 `ring`（处理环上的加减乘）配合使用来解决四则运算问题。

```
example (x : ℝ) (h : x > 0) : 1 / x + 1 = (x + 1) / x := by
  field_simp
  ring

-- 可以通过参数列表传入自定义引理
example {a b : ℝ} (h : b ≠ 1) : a = (a * b - a) / (b - 1) := by
  field_simp [sub_ne_zero_of_ne h] -- 告诉 tactic 分母不为 0
  ring
```

4.3.3 综合性求解器

这一类 Tactic 功能更为强大，通常用于直接解决某一类特定领域的复杂命题。

算术自动化： `omega` 是 Lean 4 中功能极为强大的自动化 tactic，用于处理自然数（`Nat`）和整数（`Int`）的一切命题。

它的底层机制基于 Presburger 算术的决策过程。通俗来讲，只要你的命题只涉及加法、减法、乘常数（如 `2 * x`）以及比较关系（`≤`, `<`, `=`），`omega` 几乎都能秒杀。

`omega` 的能力目前对于除法和取模有所限制：

- **支持：**自然数的除法（下取整）。
- **受限：**取模运算（`%`）。由于取模引入了非线性特性，`omega` 目前只能处理非常简单的取模情形。

尽管如此，随着 Lean 的迭代，omega 的功能正在不断完善，它在未来可能会成为处理整数相关命题的神器。

```
-- 线性不等式：轻松解决
example (x : Nat) : x ≥ 1 → x + 1 ≤ 2 * x := by omega

-- 包含除法：可以解决
example (x : Int) : x ≥ 2 → x / 2 ≤ x - 1 := by omega

-- 简单的数值取模：可以解决
example : 5 % 2 = 1 := by omega

-- 涉及定义的取模：需要辅助
example (x : ℕ) (h : Odd x) : x % 2 ≠ 0 := by
  rw [Odd] at h  -- 先展开 Odd 的定义
  omega          -- 然后交给 omega

-- 复杂的取模：需要提供定理
example (x : Nat) : x * (x + 1) % 2 = 0 := by
  have h : Even (x * (x + 1)) := by
    exact Nat.even_mul_succ_self x
  rw [Even] at h -- 利用偶数定义将取模转化为存在性命题
  omega
```

可以看到，在最后两个例子中，面对 omega 无法直接“看穿”的高级概念（如 Odd、Even），我们需要手动展开定义或引入辅助引理，将问题转化为 omega 能理解的算术形式。

通用证明搜索： aesop (Automated Extensible Search for Obvious Proofs) 是 Lean 4 中一个通用的自动化证明搜索 tactic，旨在替代 Coq 中的 auto 或 Isabelle 中的 auto/blast。

与 omega 不同，aesop的底层机制是一种“最优优先的树搜索”（Best-first Tree Search）。在搜索过程中，它会自动尝试应用各种规则，并将这些规则分为两类：

- 安全规则 (Safe Rule)：应用后肯定不会导致证明“走入死胡同”的操作（如 intro, split）。**策略：** aesop 会贪婪地应用所有可用的安全规则。这相当于“无脑推进”，不需要回溯。
- 非安全规则 (Unsafe Rule)：应用后可能会导致证明无法完成的操作（如尝试 apply 某个特定的定理，或者进行某个特定的分类讨论）。**策略：** 这是搜索树产生**分支**的地方。aesop 会根据启发式评分，优先尝试可能性最大的分支。如果走不通，它会自动**回溯**并尝试其他路径。

此外，aesop 的另一个强大之处在于它**内置了** simp。在搜索树的每一个节点，它都会先尝试用 simp 进行标准化和化简。因此，它实际上是“逻辑拆解（Safe/Unsafe）”与“代数化简（Simp）”的结合体，非常适合消灭那些“逻辑琐碎但步骤繁多”的证明

```
-- 已知：存在一个 x，使得 P x 且 Q x ；并且：对所有 x，P x → R x；证明：存在一个 x，使得 R x ∧ Q x
example  (α : Type*)  (P Q R : α → Prop)  (h₁ : ∃ x, P x ∧ Q x)  (h₂ : ∀ x, P x → R x) :  ∃ x, R x ∧ Q x := by
  aesop -- 一阶逻辑 + 构造性目标

open Topology
example {α β : Type*} [TopologicalSpace α] [TopologicalSpace β]
    (f g : α → β) (hf : Continuous f) (hg : Continuous g) :
```

```
  Continuous fun x => (f x, g x) := by
aesop -- aesop 自动把目标匹配到库里现成引理然后 apply 掉
```

值得一提的是，如果你想知道 `aesop` 到底干了什么，可以使用 `aesop?`。它会在 Infoview 中打印出搜索到的具体证明步骤（Try this: ...），方便你学习或将其替换为显式的 Tactic 脚本。

4.4 Mathlib 中的群结构

在 Mathlib 中，群论的构建并非从零开始，而是建立在**幺半群（Monoid）**的层级之上。这种设计主要是为了兼容环（Ring）的乘法结构，因为环的乘法通常只构成幺半群而非群。

从 Monoid 到 Group

Mathlib 通过类型类（Type Class）定义代数结构。`Group G` 本质上是一个拥有逆元（`Inv`）的 `Monoid G`。这意味着我们在群论中使用的许多引理实际上继承自 `Monoid` 库。

- `Monoid M`：满足结合律且有单位元的结构（通常用于环的乘法或自然数加法）。
- `Group G`：在 Monoid 基础上增加了 `div` 或 `inv` 结构，即每个元素均可逆。

我们可以先通过以下等效代码来理解 Lean 中群的核心定义：

```
namespace Demo
class Group (G : Type*) extends Mul G, One G, Inv G :=
  (mul_assoc : ∀ a b c : G, (a * b) * c = a * (b * c))
  (one_mul : ∀ a : G, 1 * a = a)
  (mul_one : ∀ a : G, a * 1 = a)
  (mul_left_inv : ∀ a : G, a⁻¹ * a = 1)
end Demo
```

这里 `extends` 的意思是，要成为一个 `Group`，类型 G 必须先拥有乘法（`Mul`）、单位元（`One`）和求逆（`Inv`）这三种基本运算结构。

符号系统与对偶性

Mathlib 严格区分**乘法记号**（Multiplicative，默认）和**加法记号**（Additive）。这两套系统通过元程序 `to_additive` 自动关联，但在代码层面是两个独立的类型类实例。

```
-- 整数加法构成一个交换加法群
#check AddCommGroup ℤ
-- 非零实数乘法构成一个交换群
#check CommGroup ℝˣ
```

4.5 群同态、子群与商群

本节我们将深入探讨群论中的结构性概念：群同态、子群以及商群。Lean 通过精巧的类型设计（如打包结构、强制转换和类型类）来处理这些概念。

4.5.1 群同态 (Group Homomorphisms)

在介绍群同态之前，我们需要先理解 Mathlib 中处理代数同态的核心设计理念。

打包映射 (Bundled Maps)

在纸笔数学中，我们通常说“令 f 为一个从 G 到 H 的同态”。但在 Lean 中，为了计算机能自动管理这些性质，同态被定义为**打包结构**（Bundled Structure）。

这意味着一个对象 `f : G →* H` 不仅仅是一个函数，它是一个包含两部分信息的**结构体**：

1. 底层的映射函数（Map）。
2. 该映射保持运算性质的证明（Properties）。

我们使用 `MonoidHom M N`（记作 `M →* N`）来表示乘法同态，用 `AddMonoidHom M N`（记作 `M →+ N`）表示加法同态。

```
variable {M N : Type*} [Monoid M] [Monoid N]

-- 使用点记法 (Dot Notation) 调用性质
-- f.map_mul x y 等价于 MonoidHom.map_mul f x y
example (x y : M) (f : M →* N) : f (x * y) = f x * f y :=  f.map_mul x y

variable {A B : Type*} [AddMonoid A] [AddMonoid B]

-- 加法版本：保持零元
example (f : A →+ B) : f 0 = 0 :=  f.map_zero
```

群同态的组合

由于同态是打包结构，我们**不能**直接使用普通的函数组合符号 `∘`，因为 `g ∘ f` 的类型仅仅是一个普通函数，它丢失了“保持代数结构”的信息。

因此，Mathlib 提供了专用的组合函数 `MonoidHom.comp`。

```
variable {P : Type*} [Monoid P]

-- g.comp f 意味着先应用 f，再应用 g (对应数学符号：g ∘ f)
example (f : M →* N) (g : N →* P) : M →* P := g.comp f
```

群环境下的性质、构造与策略

群同态本质上就是 `MonoidHom`。但得益于群的**可逆性**，我们在处理时拥有更多工具和简化的构造方式。

1. 自动化策略 (Tactics)

- `group`：用于简化任意群中的恒等式。
- `abel`：专用于交换加法群（Abelian Groups）。

3. 简化的构造 (Simplified Construction)

虽然 `MonoidHom` 定义中只包含 `map_mul` 和 `map_one`，但在群中，保持乘法自动蕴含保持逆元。

构造群同态时，我们不需要手动证明“保单位元”。

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

-- 证明群恒等式
example (x y z : G) : x * (y * z) * (x * z)⁻¹ * (x * y * x⁻¹)⁻¹ = 1 := by
  group

-- 自动保持逆元
example (x : G) (f : G →* H) : f (x⁻¹) = (f x)⁻¹ :=
  f.map_inv x

-- 仅通过保乘法构造群同态
def myGroupHom (f : G → H) (h : ∀ x y, f (x * y) = f x * f y) : G →* H :=
  MonoidHom.mk' f h
```

群同构 (Group Isomorphisms)

当两个群之间存在双射同态时，我们称其为同构。Lean 使用 `MulEquiv`（记作 $\simeq *$）来表示。这种对象比同态更强，它包含反向映射，并且满足同类关系的三个基本性质：自反性、对称性和传递性。

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

-- 自反性
#check MulEquiv.refl G      -- G ≃* G

-- 对称性：如果 G ≃* H，则 H ≃* G (即逆映射)
variable (f : G ≃* H)
#check f.symm               -- H ≃* G

-- 传递性：如果 G ≃* H 且 H ≃* K，则 G ≃* K
variable (g : H ≃* K)
#check f.trans g            -- G ≃* K

-- 同构与其逆映射组合后等于恒等映射
example : f.trans f.symm = MulEquiv.refl G := f.self_trans_symm
```

从双射构造同构：

```
-- 如果你有一个群同态 f，而且它是双射，那么 G 和 H 是同构的。
noncomputable def isoFromBij (f : G →* H) (h : Function.Bijective f) :
  G ≃* H := MulEquiv.ofBijective f h
```

注： Lean 是一个**构造性**证明助手，这意味着定义的函数通常需要是“可计算”的。虽然在数学上“双射必有逆映射”是显而易见的，但在逻辑上，从“存在一个逆映射”（命题）提取出“具体的逆函数”（数据）往往需要依赖**选择公理**。这一过程无法生成可执行的计算机代码，因此必须标记为 `noncomputable`。

4.5.2 子群 (Subgroups)

定义与基本性质

与群同态类似，Lean 中的子群也是一个**打包结构**（Bundled Structure）。一个 `Subgroup G` 不仅仅是一个集合，它还打包了封闭性公理（包含单位元、乘法封闭、逆元封闭）。

```
variable {G : Type*} [Group G] (H : Subgroup G)

-- 1. 乘法封闭性
example {x y : G} (hx : x ∈ H) (hy : y ∈ H) : x * y ∈ H :=
  H.mul_mem hx hy

-- 2. 逆元封闭性
example {x : G} (hx : x ∈ H) : x⁻¹ ∈ H :=
  H.inv_mem hx
```

两个子群相等当且仅当它们包含相同的元素。我们可以使用 `ext` 将子群相等的证明转化为元素归属关系的双向推导。

```
-- 证明两个子群相等，等价于证明 x ∈ H ↔ x ∈ K
example (H K : Subgroup G) (h : ∀ x, x ∈ H ↔ x ∈ K) : H = K := by
  ext x
  exact h x
```

构造实例：有理数中的整数加法群

下面的例子展示了如何手动构造一个子群。我们将定义“由整数构成的有理数加法子群”(即 在 中的像)。这里涉及：

- **升格 (Coercion, ↑)**：整数 $\mathbb{Z}$ 和有理数 $\mathbb{Q}$ 是不同的类型。我们需要使用 `Int.cast` 将整数“投射”到有理数中。
- **值域 (Range)**：`Set.range f` 表示函数 f 的像集。

```
example : AddSubgroup ℚ where
  -- 1. 定义底层集合：整数映射到有理数的值域
  carrier := Set.range (Int.cast : ℤ → ℚ)
  -- 2. 加法封闭性
  -- 设 a = cast n, b = cast m, 证明 a + b 也在值域中
  add_mem' := by
    rintro _ _ ⟨n, rfl⟩ ⟨m, rfl⟩
    use n + m
    simp
  -- 3. 包含零元
  zero_mem' := by
    use 0
    simp
  -- 4. 取负封闭性
  neg_mem' := by
    rintro _ ⟨n, rfl⟩
```

```
  use -n
  simp
```

子群作为类型

虽然子群 H 被定义为 G 的一项（Term），但 Lean 允许通过强制转换将其视为一个类型（即子类型 `{x : G // x ∈ H}`）。由于子群本身也是群，Mathlib 已经注册了相关实例，我们可以直接使用 `inferInstance` 获取它。

```
-- 子群 H 继承了 G 的群结构
example (H : Subgroup G) : Group H := inferInstance
```

`inferInstance` 是一个指示 Lean 启动**类型类推断（Type Class Inference）**的函数。简单来说，它告诉 Lean：“请在已知的数据库中自动寻找并填入所需的实例”，而无需我们手动书写证明。

子群的格结构 (Lattice)

回忆 Day2 关于格的知识。Lean 中的所有子群构成了**完全格**，因此，我们可以直接应用任意关于格的通用引理（例如交集的结合律、单调性等）来处理子群。

- **偏序关系 (≤)**：对应集合的包含关系 (⊆)。
- **最大与最小**：`⊤` (全群) 和 `⊥` (平凡子群 {1})。
- **交集 (⊓)**：对应集合的交集。
- **生成子群 (⊔)**：两个子群的并集通常不是子群。因此，$H \sqcup K$ 被定义为并集的闭包 (`Subgroup.closure`)。

```
-- 1. 偏序关系即集合包含. 在 Lean 中, 我们通常写 H ≤ K 而非 H ⊆ K
example (H K : Subgroup G) : H ≤ K ↔ ∀ x, x ∈ H → x ∈ K :=
  SetLike.le_def -- 把“类似集合的结构”关系拆解为元素的包含关系。

-- 2. 交集的底层集合 = 集合的交集
example (H K : Subgroup G) :
  ((H ⊓ K : Subgroup G) : Set G) = (H : Set G) ∩ (K : Set G) := rfl

-- 3. 并集生成的子群 = 集合并集的闭包
example {G : Type*} [Group G] (H K : Subgroup G) :
    ((H ⊔ K : Subgroup G) : Set G) = Subgroup.closure ((H : Set G) ∪ (K : Set G)) := by
  rw [Subgroup.sup_eq_closure]

-- 4. ⊥ 仅包含单位元
example (x : G) : x ∈ (⊥ : Subgroup G) ↔ x = 1 := Subgroup.mem_bot
```

同态对子群的作用：Map 与 Comap

我们可以利用群同态将子群在不同的群之间“推”和“拉”：

- **Map (推)**：`Subgroup.map f H` 是 H 在 f 下的像（前推）。
- **Comap (拉)**：`Subgroup.comap f K` 是 K 在 f 下的原像（拉回）。

特别地，核 (Kernel) 和值域 (Range) 可以看作是 $\bot$ 和 $\top$ 的拉回与前推：

```
variable {N : Type*} [Group N] (f : G →* N)

-- Ker f = f⁻¹({1})
example : MonoidHom.ker f = (⊥ : Subgroup N).comap f := rfl

-- Range f = f(G)
example : MonoidHom.range f = (⊤ : Subgroup G).map f := MonoidHom.range_eq_map f
```

4.5.3 正规子群与商群 (Quotient Groups)

在群论中，商群 G/N 的构造依赖于子群 N 的正规性。在 Lean 中，我们将其作为一个**类型类 (Type Class)** 实例来处理, 即只要我们在上下文中引入了 `[N.Normal]`，Mathlib 就会自动推断出商群 G/N 上应当具备的群结构。

商映射与泛性质

首先，我们来看最基础的构造。商群在 Lean 中写作 `G ⧸ N`。我们可以定义从原群 G 到商群的自然投影（Canonical Projection），通常记为 π。更重要的是商群的**泛性质 (Universal Property)**：这是我们定义商群上同态的主要手段。如果有一个从 G 出发的同态 φ，且它的核包含 N，那么 φ 就可以唯一地“下降[1]”为一个定义在商群 G/N 上的同态。

```
variable (N : Subgroup G) [N.Normal]

-- 1. 商映射 π: G → G/N
-- QuotientGroup.mk' 构造了这一自然投影
def π : G →* G ⧸ N := QuotientGroup.mk' N

-- 2. 泛性质 (Lift)
-- 如果同态 φ 的核包含 N (即 N ≤ Ker φ)，则 φ 可以下降到商群
variable {M : Type*} [Group M] (φ : G →* M)
example (h : N ≤ MonoidHom.ker φ) : G ⧸ N →* M :=
  QuotientGroup.lift N φ h
```

商群之间的映射 (Functoriality)

如果我们有两个商群 G/N 和 G'/N'，以及一个原本在 G 和 G' 之间的同态 φ，我们如何构造出商群之间的映射呢？

虽然数学上我们常说“φ 将 N 映入 N'”，但在 Lean 中，Mathlib 推荐使用 **拉回 (Pullback)** `comap` 的视角：即 $N \le \varphi^{-1}(N')$ 来避免∃。

```
variable {G' : Type*} [Group G'] (N' : Subgroup G') [N'.Normal]
variable (φ : G →* G')

-- 若 φ 将 N 映射入 N' (即 N ≤ φ⁻¹(N'))，则诱导商群间的映射
-- 如果我们写 N.map φ ≤ N'，底层定义包含“存在量词”。而N ≤ N'.comap φ 是全称量词蕴含，底层定义等价于 ∀ x, x ∈ N →
    φ x ∈ N'
```

[1]这种将‘定义在元素上的函数’转换为‘定义在商类型上的函数’的操作，在类型论中统称为提升 (Lift)

```
example (h : N ≤ N'.comap φ) : G / N →* G' / N' :=
  QuotientGroup.map N N' φ h
```

注：在数学上，如果两个正规子群 M 和 N 是同一个集合 $(M = N)$，我们会认为 G/M 和 G/N 是同一个群。但在 Lean 的类型论中，**它们是不同的类型**。因此，我们不能直接划等号，而是需要构造一个显式的**同构**。

```
variable (M N : Subgroup G) [M.Normal] [N.Normal]

-- 即使 M = N，G / M 与 G / N 也是不同的类型
example (h : M = N) : G / M ≃* G / N :=
  QuotientGroup.quotientMulEquivOfEq h
```

4.5.4 构造积群同态

最后补充一个实用的工具。在有限群论中，我们经常需要证明一个群同构于两个群的直积（例如 $G \cong H \times K$）。要构造这样的同构，第一步往往是构造一个从 G 射向 $H \times K$ 的同态。

Mathlib 提供了 `MonoidHom.prod` 函数，它允许我们将两个同态 $f : G \to H$ 和 $g : G \to K$ 组合成一个积同态。

```
variable {K : Type*} [Group K]
variable (f : G →* H) (g : G →* K)

-- 将两个同态组合成 G → H × K
-- 对应数学映射：x ↦ (f(x), g(x))
def prodHom' : G →* H × K := MonoidHom.prod f g

-- 典型应用：利用商映射构造 G → (G / K) × (G / H)
-- 假设我们有正规子群 H, K，这是证明 G ≅ H × K 的关键一步
variable (N1 N2 : Subgroup G) [N1.Normal] [N2.Normal]
def quotientProd : G →* (G / N1) × (G / N2) :=
  MonoidHom.prod (QuotientGroup.mk' N1) (QuotientGroup.mk' N2)

-- 进阶：如果我们要证明同构，可以使用 MulEquiv.prodCongr
#check MulEquiv.prodCongr  -- 它表示：如果 G ≅ H 且 G' ≅ K，则 G × G' ≅ H × K
```

4.6 Exercises

```
section exercise₁
variable (α : Type) (s t u : Set α)
/- 请尽量少使用 tactic 且不使用`constrcutor`完成以下四个题目. -/
example : s ∩ (s ∪ t) = s := sorry
example : s ∪ s ∩ t = s := sorry
example : s \ t ∪ t = s ∪ t := sorry
example (h : s ⊆ t) : s \ u ⊆ t \ u := sorry


/- 以下四个题目允许使用 tactic 或搜索定理来完成证明, 但仍请做到结构尽量精简. -/
example : s \ t ∪ t \ s = (s ∪ t) \ (s ∩ t) := by
  sorry
example : s ∩ t ∪ s ∩ u ⊆ s ∩ (t ∪ u) := by
  sorry
example : s \ (t ∪ u) ⊆ (s \ t) \ u := by
  sorry

variable (f : α → β) (v : Set β)
example : f '' s ⊆ v ↔ s ⊆ f ⁻¹' v := by
  sorry
end exercise₁



section exercise₂
example (a b c d x y : ℂ) (hx : x ≠ 0) (hy : y ≠ 0) :
    a + b / x + c / x ^ 2 + d / x ^ 3 = a + x⁻¹ * (y * b / y + (d / x + c) / x) := by
  sorry

example {x : ℝ} (h : x = 1 ∨ x = 2) : x ^ 2 - 3 * x + 2 = 0 := by
  sorry

example {n m : ℕ} : (n + m) ^ 3 = n ^ 3 + m ^ 3 + 3 * m ^ 2 * n + 3 * m * n ^ 2 := by
  sorry

example {a b c : ℝ} (ha : a ≤ b + c) (hb : b ≤ a + c) (hc : c ≤ a + b) :
    ∃ x y z, x ≥ 0 ∧ y ≥ 0 ∧ z ≥ 0 ∧ a = y + z ∧ b = x + z ∧ c = x + y := by
  set x := (b - a + c) / 2 with hx_def
  set y := (a - b + c) / 2 with hy_def
  set z := (a + b - c) / 2 with hz_def
  sorry

example (x : ℕ) (h : (x : ℚ) = 1) : x = 1 := by
  sorry
```

```
example {b : ℤ} : 0 ≤ max (-3) (b ^ 2) := by
  sorry

example (x : ℕ) : x ≥ 2 → x / 2 ≥ 1 := by
  sorry
end exercise₂
```

尝试定义由元素 x 诱导的共轭子群 xHx^{-1}。

```
def conjugate (x : G) (H : Subgroup G) : Subgroup G where
  carrier := {a : G | ∃ h, h ∈ H ∧ a = x * h * x⁻¹}
  one_mem' := by
    dsimp
    use 1
    simp
  inv_mem' := by
    dsimp
    rintro _ ⟨h, h_in, rfl⟩
    use h⁻¹
    simp [h_in] -- 利用群的共轭性质
  mul_mem' := by
    dsimp
    rintro _ _ ⟨a, ha, rfl⟩ ⟨b, hb, rfl⟩
    use a * b
    simp [ha, hb] -- 利用 x(ab)x⁻¹ = (xax⁻¹)(xbx⁻¹)
```

The next two exercises derive a corollary of Lagrange’ s lemma.

```
  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
```