Day 2

2.1 交换环上的化简（ring）

上一节我们用 rw 逐步改写等式来做计算,但很多时候我们其实不必手动一条条 rw。比如对于**交换环**（如 $\mathbb{Z},\mathbb{Q},\mathbb{R},\mathbb{C}$）中的大量“纯代数等式”,：mathlib 提供了一个自动化 tactic：ring。

环结构与可用的基本事实 在 Lean 中，[Ring R] 表示类型 R 带有环结构，它提供：

```
variable (R : Type*) [Ring R]
-- 加法
#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)
-- 乘法
#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)
-- 分配律
#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)
```

这些事实既适用于抽象的 R，也会自动适用于具体实例，因为 Lean 会通过类型类机制自动找到相应的环实例。

ring 的使用场景 特别地，当我们考虑交换环时（如上下文有 [CommRing R]或 [Field R]）时，只要目标是“多项式恒等式”，可以尝试 ring （它把等多项式暴力展开，然后看长得一不一样）：

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

-- 当你看到由 加、减、乘、乘方 组成的一团乱麻的等式，并且知道变量满足交换律时，敲 ring 就可以了。
example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b := by
  ring
end
```

直观上，它相当于自动完成一长串 rw [mul_add, add_mul, ...] 的展开、合并同类项与重排。

常见套路： 由于ring 只做**代数恒等式的规范化**，不会自动把你的假设“代入目标”。因此常见写法是先用 rw 把假设展开/代入，再用 ring 处理剩下的纯代数部分：

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

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

相关的代数自动化　mathlib 还提供了与 `ring` 思路类似的 tactic，用来覆盖更广的代数结构：

- `group`：用于乘法群（一般不交换）中的等式化简；
- `abel`：用于交换加法群中的等式化简；
- `noncomm_ring`：用于非交换环的等式化简[1]（更一般，但通常更慢）。

2.2　不等式与格论

虽然 `rw` 和 `ring` 是证明等式的利器，但一旦目标变成**不等式**或更一般的**序结构**，原先那套“把表达式改写到一样”就不够用了。

本节给出两条主线：

- 在线性不等式上，利用 `linarith` 做一键自动化；
- 在更抽象的序结构上，学会把证明组织成“用通用接口引理推进”的套路（尤其是格论里的 `le_antisymm`、`le_inf`、`sup_le` 等）。

线性算术策略：`linarith`

当目标和假设属于**线性算术**（只包含加减、常数乘、$\leq$、$=$, $<$ 等线性关系）时，`linarith` 通常是首选工具。可以把它理解成一个“线性约束求解器”：它会收集当前语境（Context）里的线性等式与不等式，把它们组合、消元、代入，尝试直接推出目标。对初学者而言，一个很实用的经验是：

> 如果你肉眼觉得“这题就是把几条线性不等式加起来/代一下就出结论”，那就先试 `linarith`。

```
variable (a b d : ℝ)

-- linarith 自动利用 h'' 将 d 替换为 2，并组合 h 与 h' 推出目标
example (h : 2 * a ≤ 3 * b) (h' : 1 ≤ a) (h'' : d = 2) : d + a ≤ 5 * b := by
  linarith
```

不适用场景：`linarith` 无法直接处理非线性项（如 a^2、$a \cdot b$）或超越函数（指数、对数等）。此时我们可以先把目标改写/降阶到线性问题。

[1] 命名上看起来有点“反直觉”，主要是历史原因：`ring` 使用最频繁，因此采用了更短的名字。

技巧：将“非线性”转化为“线性”

面对 exp、log 这样的非线性函数，我们可以

先用单调性/等价刻画把“函数不等式”剥离成“自变量不等式”，再用 linarith 收尾。

```
open Real
variable (a b : ℝ)

#check (exp_le_exp : exp a ≤ exp b ↔ a ≤ b)

example (h : a + 2 ≤ b) : exp (a + 1) ≤ exp b := by
  rw [exp_le_exp] -- ⊢ a + 1 ≤ b
  linarith
```

补充：非线性救星 nlinarith

如果你的目标中显式包含**平方**，且无法通过简单的重写消除，那么 linarith 就会失效。此时应使用 nlinarith，它**自动利用了“实数平方非负”**（$x^2 \geq 0$）这一隐含事实。但 nlinarith 并不擅长自动做因式分解。如果你给它一个展开的多项式（如 $a^2 - 2ab + b^2$），最好先手动将其整理为平方形式。

```
example (a b : ℝ) : a ^ 2 - 2 * a * b + b ^ 2 ≥ 0 := by
  -- 1. 先用 ring 帮我们把展开式整理回完全平方的形式
  have h : a ^ 2 - 2 * a * b + b ^ 2 = (a - b) ^ 2 := by ring
  --目标变成了 (a - b) ^ 2 ≥ 0
  nlinarith
```

2.2.1 格论 (Lattice)：统一的序结构接口

我们刚刚讨论了实数上的大小关系 ($\leq$)。但在数学中，“顺序”的概念无处不在：集合有包含关系，逻辑有蕴含关系，数论有整除关系。如果为每一种数学对象都单独定义一套引理（比如 set.subset_trans, nat.dvd_trans, real.le_trans），库会变得极其冗余。Mathlib 的解决方案是引入**格论 (Lattice Theory)**。

Lattice 类型类提供了一个统一的接口，它将“偏序关系”以及“最大下界 / 最小上界”抽象了出来:

- PartialOrder 给出偏序（$\leq$，并带有反身、传递、反对称）；
- 偏序之上额外给出二元运算 inf（$\sqcap$）与 sup（$\sqcup$），分别表示**最大下界与最小上界**。

这一抽象的价值在于：一旦某个对象被证明是一个格，我们就能对它复用同一套“格论通用引理”：

- **全序集 (Total Orders)**：最直观的例子，如实数 $\mathbb{R}$。
  - $\sqcap$ (meet) $\to$ 取最小值 (min)
  - $\sqcup$ (join) $\to$ 取最大值 (max)
- **集合论**：全集的子集族，按 $\subseteq$ 排序。
  - $\sqcap \to$ 集合交集 ($\cap$)：两个集合的公共部分是它们最大的“下界”。
  - $\sqcup \to$ 集合并集 ($\cup$)：包含两个集合的最小集合是它们的“上界”。

- **数论**：自然数，按整除关系 $|$ 排序（注意不是大小！）。
  - $\sqcap \to$ 最大公约数 (gcd)：同时整除 a, b 的最大数。
  - $\sqcup \to$ 最小公倍数 (lcm)：同时被 a, b 整除的最小数。
- **逻辑**：命题或布尔值，按蕴含关系 $\to$ 排序。
  - $\sqcap \to$ 逻辑与 ($\wedge$)
  - $\sqcup \to$ 逻辑或 ($\vee$)
- **代数结构**：群的子群、向量空间的子空间等，通常按包含排序。
  - $\sqcap \to$ 子群的交集（Interscetion）：两个子群的交依然是子群。
  - $\sqcup \to$ 子群的生成和（Generated Subgroup）：两个子群的并集通常不是子群，所以上界是由并集**生成**的最小子群。

格论里的等式　在偏序结构里，想证等式 $u = v$ 时，不像算术那样能“移项/相消”。最稳妥的通用套路是把等式改写成“双向不等式”，也就是用**反对称性**：

$$u = v \quad \Longleftrightarrow \quad (u \leq v) \text{ 且 } (v \leq u).$$

在 Mathlib 里这一步对应引理 `le_antisymm`。

接下来，“看到目标长什么样，就选对应的构造/消去引理”：

- **目标是** $X \leq a \sqcap b$：用 `le_inf`，把目标拆成 $X \leq a$ 和 $X \leq b$。
- （对偶地）**目标是** $a \sqcup b \leq X$：用 `sup_le`，把目标拆成 $a \leq X$ 和 $b \leq X$。
- **目标是** $a \sqcap b \leq a$ **或** $a \sqcap b \leq b$：用投影公理 `inf_le_left` / `inf_le_right`。
- （对偶地）**目标是** $a \leq a \sqcup b$ **或** $b \leq a \sqcup b$：用 `le_sup_left` / `le_sup_right`。

例 1：交运算交换律 $a \sqcap b = b \sqcap a$

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

#check (le_antisymm : a ≤ b → b ≤ a → a = b)
#check (le_inf : c ≤ a → c ≤ b → c ≤ a ⊓ b)
#check (inf_le_left  : a ⊓ b ≤ a)
#check (inf_le_right : a ⊓ b ≤ b)

example : a ⊓ b = b ⊓ a := by
  apply le_antisymm
  · -- 证 a ⊓ b ≤ b ⊓ a: 右边是 inf, 用 le_inf 拆目标
    apply le_inf
    · exact inf_le_right  -- a ⊓ b ≤ b
    · exact inf_le_left   -- a ⊓ b ≤ a
  · -- 对称地再来一遍
    apply le_inf
    · exact inf_le_right
    · exact inf_le_left
```

注：代码中的 •（点号）用于隔离子目标的上下文，是一种良好的代码习惯。

例 2：吸收律 $a \sqcap (a \sqcup b) = a$

思路仍然是反对称性：一边用"⊓ 的投影"，另一边用"要证 $\le \inf$ 就用 `le_inf`"。

```
-- 直观理解：a 和 (a ⊔ b) 取交集，结果还是 a
variable {α : Type*} [Lattice α] (a b : α)

example : a ⊓ (a ⊔ b) = a := by
  apply le_antisymm
  · -- a ⊓ (a ⊔ b) ≤ a: inf 的左投影
    exact inf_le_left
  · -- a ≤ a ⊓ (a ⊔ b): 目标右边是 inf，用 le_inf
    apply le_inf
    · exact le_rfl        -- a ≤ a
    · exact le_sup_left   -- a ≤ a ⊔ b
```

2.3 逻辑结构与证明策略

在 Lean 中，证明的"核心动作"往往不是计算，而是对目标（Goal）与假设（Context）里的逻辑结构进行**拆解**与**构造**。我们将按照"你面对的是**目标**还是**假设**"这一维度，将常用的逻辑连接词（$\land, \lor, \forall, \exists, \to$）对应的 Tactic 进行分类。

2.3.1 第一类：当逻辑符号在"目标"中

当目标（$\vdash$ 右侧）包含逻辑符号时，我们需要使用**构造策略**，将复杂目标拆解为简单的子目标。

1. 构造 $\land$（与）： `constructor`

如果要证明 $P \land Q$，只需分别证明 P 和 Q。

- **Tactic：** `constructor`
- **效果：** 生成两个子目标，一个证明左边，一个证明右边。

```
example (x y : ℝ) (h₀ : x ≤ y) (h₁ : x ≠ y) : x ≤ y ∧ x ≠ y := by
  constructor
  · exact h₀ -- 处理第一个分支
  · exact h₁ -- 处理第二个分支
```

2. 构造 $\lor$（或）： `left` / `right`

如果要证明 $P \lor Q$，你必须做出选择：是证明 P 成立，还是证明 Q 成立。

- **Tactic：** `left` 或 `right`
- **效果：** 目标变为 P（如果选左）或 Q（如果选右）。

```
example (y : ℝ) (h : y > 0) : y > 0 ∨ y < -1 := by
  left      -- 我们手里有 y > 0，所以选择证明左边
  exact h
```

3. 构造 $\to$ 或 $\forall$：`intro`

如果要证明“若 P 则 Q”或“对任意 x”，标准的数学动作是“假设 P 成立”或“任取一个 x”。

- **Tactic**：`intro 变量名 / 假设名`
- **效果**：将前提或变量移入 Context（假设区），目标变为结论部分。

```
-- 目标: ∀ a b, a ≤ b → f a ≤ f b
example (f : ℝ → ℝ) : ∀ a b, a ≤ b → f a ≤ f b := by
  intro a b h_le  -- 引入变量 a, b 和假设 h_le
  -- 现在的目标变成了: f a ≤ f b
  sorry
```

4. 构造 $\exists$（存在）：`use`

要证明“存在一个 x 满足 $P(x)$”，你需要给出一个具体的例子（Witness）。

- **Tactic**：`use 值`
- **效果**：将目标中的存在量词剥离，变为证明该值满足性质。

```
example : ∃ x : ℝ, x > 10 := by
  use 100
  -- 现在的目标变成了: 100 > 10
  norm_num
```

2.3.2 第二类：当逻辑符号在“假设”中

当假设（Context）中包含复杂的逻辑命题（如 $h : P \vee Q$）时，我们需要使用**拆解策略**（Destruction）来提取信息。最通用的拆解工具是 `rcases`（Recursive Cases）。

1. 拆解 $\wedge$（与）与 $\exists$（存在）

如果我们知道“P 且 Q”，或者“存在一个 x 满足 P”，这意味着我们同时拥有了两样东西（两个事实，或者一个值加一个事实）。

- **语法**：`rcases h with ⟨名字 1, 名字 2⟩`
- **效果**：将假设 h 拆成两部分。

```
example (h : ∃ x, x > 5 ∧ x < 10) : True := by
  -- 将 h 拆解为一个具体的值 x₀, 以及两个性质
  rcases h with ⟨x₀, ⟨h_gt, h_lt⟩⟩
  -- 现在 Context 里有了:
  -- x₀ : ℝ
  -- h_gt : x₀ > 5
  -- h_lt : x₀ < 10
  trivial
```

2. 拆解 ∨（或）：分类讨论

如果我们知道“P 或 Q”，则必须进行分类讨论。

- **语法**：`rcases h with h_left | h_right`
- **效果**：生成两个分支。分支一假设 P 成立（命名为 `h_left`）；分支二假设 Q 成立（命名为 `h_right`）。

```
example (x : ℝ) (h : x = 0 ∨ x = 1) : x * (x - 1) = 0 := by
  rcases h with h0 | h1
  · -- 分支 1: 假设 x = 0 (h0)
    rw [h0]; norm_num
  · -- 分支 2: 假设 x = 1 (h1)
    rw [h1]; norm_num
```

2.3.3 逻辑变换与自动化

除了手动的构造与拆解，我们还需要处理“否定”以及使用自动化工具来清理琐碎的逻辑步骤。

1. 处理否定 (¬)：`push_neg`

在数学中，我们经常需要把 $\neg(\forall x, Px)$ 改写为 $\exists x, \neg Px$。手动做这些变换很痛苦，Lean 提供了 `push_neg`。

- `push_neg`：作用于目标，将否定号推入命题最深处。
- `push_neg at h`：作用于假设 h。

```
example (h : ¬ ∀ x, x > 0) : ∃ x, x ≤ 0 := by
  -- h : ¬ ∀ (x : ℝ), x > 0
  push_neg at h
  -- 现在 h 变成了 : ∃ x, x ≤ 0
  exact h
```

2. 数值计算自动化：`norm_num`

我们在前文介绍了处理代数的 `ring` 和处理不等式的 `linarith`。这里补充一个专门处理“具体数字”的工具。`norm_num` 擅长判断如 $2 + 2 = 4$、$(3 :) < 5$ 这类纯数值问题。

```
example : (1 : ℝ) + 2 < 4 := by
  norm_num -- 自动计算并验证
```

3. 纯逻辑自动化：`tauto`

如果一个命题仅仅是逻辑上的重言式（Tautology），不涉及具体的数学计算（如加减乘除），可以直接交给 `tauto`。它就像是自动化真值表。

```
example (p q : Prop) : p ∧ q → p ∨ q := by
  tauto -- 逻辑真理，无需废话
```

2.3.4 总结：逻辑操作速查表

逻辑符号	在目标中（构造）	在假设中（拆解）
∧ (与)	`constructor`	`rcases h with ⟨h1, h2⟩`
∨ (或)	`left` / `right`	`rcases h with h1 | h2`
→ (蕴含)	`intro h`	`apply h` / `have h2 := h h1`
∀ (全称)	`intro x`	`specialize h x` / `apply h`
∃ (存在)	`use value`	`rcases h with ⟨value, h_prop⟩`
¬ (非)	`intro` / `push_neg`	`push_neg at h`

2.3.5 自动化简化：simp

`simp` 是 Lean 中最常用的自动化战术之一。如果说 `rw` 是手动的“精准手术”，那么 `simp` 就像是一个“智能吸尘器”：它会自动在库中寻找并应用那些可以将表达式变得更简单的引理。

工作原理：规范化 (Canonicalization)

直觉上，`simp` 的目标是将表达式**规范化**。它反复应用重写规则，直到目标无法被进一步简化为止。常见的简化包括：

- 算术化简：`x + 0` → `x`, `x * 1` → `x`
- 集合论化简：`x ∈ s ∩ t` → `x ∈ s ∧ x ∈ t`
- 逻辑化简：`p ∧ True` → `p`

自定义简化规则：@[simp]

你可能会问：`simp` 怎么知道哪些引理可以用来化简？答案是：`simp` 维护了一个巨大的数据库。任何标有 `@[simp]` 属性的引理都会被自动加入这个数据库。

你自己也可以定义 `simp` 引理！当你定义了一个新函数后，通常应该立即证明其在基础值上的行为，并标记为 `@[simp]`，这样以后 `simp` 就能自动处理这个函数了。

```
-- 1. 定义一个新函数
def double (n : Nat) : Nat := n + n

-- 2. 告诉 simp 如何化简这个函数的一个特例
-- 使用 @[simp] 属性标记此引理
@[simp] lemma double_zero : double 0 = 0 := rfl

-- 3. 实战：simp 现在变聪明了
example (x : Nat) : double 0 + x = x := by
  -- simp 会自动找到 double_zero，把 double 0 变成 0
  -- 然后利用库里的 add_zero 把 0 + x 变成 x
  simp
```

工程最佳实践：`simp only`

虽然直接输入 `simp` 很爽，但在大型项目（如 Mathlib）中，我们强烈建议使用 `simp only [...]` 显式指定要用的引理。

- **稳定性**：如果未来库里新增了一个 `simp` 引理，可能会改变简化路径，导致你原本能跑通的 `simp` 突然挂掉。使用 `simp only` 可以由你完全掌控局面。
- **速度**：`simp` 需要搜索成千上万条引理，而 `simp only` 只需要查表几次。

```
example {α : Type} {s t : Set α} {x : α} : x ∈ s ∪ t ↔ x ∈ s ∨ x ∈ t := by
  simp only [Set.mem_union]
```

探查利器：`simp?`

- **功能**：它像 `simp` 一样工作，但会在 InfoView 中打印出它具体使用的引理列表。
- **用法**：先写 `simp?`，然后点击 InfoView 中的 **"Try this: ..."** 链接，VS Code 会自动把你的代码替换成稳健的 `simp only [...]` 写法。

```
example (n : Nat) : n + 0 = n := by
  -- 1. 你想化简，但不知道具体引理叫 add_zero 还是 zero_add
  -- 于是你写：
  simp?
  -- 2. Lean 在右侧 InfoView 告诉你：
  -- "Try this: simp only [add_zero]"
  -- 3. 你点击那个蓝色链接，代码自动变为：
  simp only [add_zero]
```

总结流程：开发时用 `simp?` 探路 → 点击替换为 `simp only` → 提交代码。这样既享受了自动化的便利，又保证了代码的健壮性。

位置修饰

同 `rw` 一样，`simp` 也可以指定作用域：

- `simp at h`：仅化简假设 `h`。
- `simp at *`：同时化简所有假设和目标（威力巨大，但容易把上下文搞乱，调试时慎用）。

```
example (x y : Nat) (h : x + 0 = y) : x = y := by
  simp at h
  exact h
```

2.3.6 结构化证明：`calc` 模式

当证明需要进行一连串的代数变形或不等式放缩时，传统的 Tactic 模式（堆砌 `rw` 或 `apply`）往往会让人迷失方向。`calc` 模式旨在模仿人类在黑板上写数学推导的格式：**一步一个台阶，每一步都给出理由。**

基本语法

一个 `calc` 块由多行组成，每一行都遵循 表达式 + 关系符 + 表达式 + 证明的结构：

```
calc
  A = B := by proof1
  _ = C := by proof2
  _ = D := by proof3
```

- **下划线（_）**：这是一个占位符，代表**上一行的右端**。这省去了重复抄写长表达式的麻烦。
- **传递性**：Lean 会自动利用关系的传递性（Transitivity）。如果证明了 $A = B, B = C, C = D$，Lean 自动推出 $A = D$。

示例 1：纯等式推导

我们来证明 $(a + b)^2 = a^2 + 2ab + b^2$。虽然用 `ring` 可以秒杀，但用 `calc` 可以展示推导细节：

```
example (a b : ℝ) : (a + b)^2 = a^2 + 2 * a * b + b^2 := by
  calc
    (a + b)^2
      = (a + b) * (a + b)   := by rw [pow_two]
    _ = a * (a + b) + b * (a + b) := by rw [add_mul]
    _ = a * a + a * b + (b * a + b * b) := by rw [mul_add, mul_add]
    _ = a^2 + 2 * a * b + b^2 := by ring
```

示例 2：混合关系的链式推导

`calc` 强大的地方在于它可以**混合使用不同的关系符**（如 $=, \leq, <$），只要这些关系在逻辑上能连通。Lean 会自动处理这种“混合传递性”：

$$(A = B) \wedge (B < C) \wedge (C \leq D) \implies A < D$$

```
-- 假设我们已知以下事实
variable (a b c d : ℝ)
variable (h1 : a = b)  (h2 : b < c + 1) (h3 : c + 1 ≤ d)

example : a < d := by
  calc
    a = b       := h1      -- 利用等式
    _ < c + 1   := h2      -- 利用小于
    _ ≤ d       := h3      -- 利用小于等于
  -- Lean 自动推断出结论: a < d
```

技巧：配合 `ring` 使用　在处理不等式时，我们经常需要在某一步进行纯代数变形（等式），然后再进行放缩。可以在 `calc` 中随意穿插 `by ring` 来处理繁琐的代数整理。

```
example (n : ℕ) : (n + 1)^2 ≥ 4 * n := by
  calc
```

```
  (n + 1)^2
    = n^2 + 2 * n + 1  := by ring
  _ ≥ 2 * n + 2 * n    := by nlinarith -- 即使中间一步跳跃较大也没关系
  _ = 4 * n            := by ring
```

实践建议

- 逻辑结构优先“结构化处理”（`constructor/intro/rcases`），再考虑自动化（`simp/linarith/tauto`）。
- 自动化优先“可控”（`simp only [...]`、先把目标整理成线性再 `linarith`），避免一次性把 proof state 变得难以预测。
- `calc` 适合“主干推导清晰”的证明；局部技巧（化简、重写、自动化）尽量放在每一步的 `by ...` 里，让结构保持可读。

2.4 寻找 Mathlib 定理

写证明时，最大的瓶颈往往不是 tactic，而是：**你数学上知道要用什么定理，但不知道它在 mathlib 里叫什么/在哪个文件里**。下面给出一套“从快到慢”的找法（优先使用前两条）：

- **（最常用）靠命名习惯 + 自动补全**：mathlib 的定理名高度规律化。比如想找形如 $x + y \le \cdots$ 的引理，通常以 `add_le_` 开头；与指数有关常见是 `exp_`；与最小值有关常见是 `min_` 等。在编辑器中输入前缀后用自动补全（Ctrl-Space / Cmd-Space）往往能直接命中。
- **（非常实用）`#search` / `apply?`**：当你已经写出了目标的形状时，让 Lean 帮你找。

```
-- 例：让 Lean 猜一个能证明 0 ≤ a^2 的引理
example (a : ℝ) : 0 ≤ a ^ 2 := by
  apply?
  -- 常见结果会提示类似 sq_nonneg a 或 pow_two_nonneg a 等
```

- **文档页与源码**：
  - mathlib API 文档：`https://leanprover-community.github.io/mathlib4_docs/`
  - theories 总览页（按领域导览）：`https://leanprover-community.github.io/theories/`
  - GitHub 源码：`https://github.com/leanprover-community/mathlib4`
- **专用搜索工具（按模式搜类型）**[2]：
  - Loogle：`https://loogle.lean-lang.org/`(vscode 里直接右键也行)
  - leansearch：`https://leansearch.net/`(also try `#search`)
  - LeanExplore：`https://www.leanexplore.com/`
- **从一个已知定理“就地扩展”**：在 VS Code 里右键已有定理名，跳转到定义文件；同一文件附近往往有一整组相似引理（例如一串 `add_le_add_*`、`min_le_*`）。

实践建议：先用“命名习惯 + 自动补全”猜一个候选名；不行就用 `#search` / `apply?`；再不行才去文档页/源码/外部工具。

2.5 Lean 中的命名法

为了让代码更接近主流编程语言、并且一眼区分“可运行的程序/数据”和“需要证明的命题/定理”，Lean（尤其是 mathlib 社区）约定了一套比较严格的命名规范。本节内容主要参考 mathlib 的命名指南：`https://leanprover-community.github.io/contribute/naming.html`。

Lean 中最常见的三种命名风格是：

- `snake_case`：全小写，用下划线连接单词；
- `lowerCamelCase`：第一个单词小写，其后每个单词首字母大写，不使用分隔符；
- `UpperCamelCase`：每个单词首字母大写，不使用分隔符。

[2] note their mathlib versions

核心规则（记住这几条就够用）

1. **命题（`Prop`）相关：用** `snake_case`**.**
   典型包括：定理名、引理名、证明名（即类型为 `Prop` 的项）。
   例：`nat.add_comm`、`mul_assoc`、`map_one`。

2. **类型/结构：用** `UpperCamelCase`**.**
   典型包括：归纳类型、`structure`、`class`（以及它们的名字）。
   例：`Nat`、`List`、`MonoidHom`、`TopologicalSpace`。

3. **普通函数与数据：用** `lowerCamelCase`**.**
   也就是既不是类型名、也不是命题/定理名的"普通项"，例如一般函数、构造出的数据等。
   例：`toFun`、`mapMul`、`simpLemma`（示意）。

4. **函数名通常"跟随返回值的风格"。**
   直觉上：一个东西"最终返回什么类型"，它的名字往往就采用那个层级应当使用的风格。例如某个字段返回 `Prop`，那字段名往往是 `snake_case`；返回普通数据则常用 `lowerCamelCase`。

两个常见细节

- **在** `snake_case` **里提到** `UpperCamelCase` **的类型名：**通常把该类型名改成 `lowerCamelCase` 形式嵌入。例如类型叫 `MyType`，相关引理可能写成 `myType_eq_foo` 之类的形式。
- **缩写（Acronym）：**像 `LE`、`LT` 这类缩写一般作为整体处理，具体大小写以首字符为准；mathlib 中也存在少量历史例外（属于"已存在代码的兼容"）。

例子

```
-- 类型/结构：UpperCamelCase
structure OneHom (M : Type _) (N : Type _) [One M] [One N] where
  toFun : M → N          -- 普通字段：lowerCamelCase
  map_one' : toFun 1 = 1 -- 命题字段：snake_case（结尾 ' 常用于避免重名）

-- 定理/引理：snake_case
theorem map_one [OneHomClass F M N] (f : F) : f 1 = 1 := sorry
```

定理名怎么起：结论优先，条件用 `of` 串起来

实践中你最常需要处理的是**定理命名**。mathlib 的风格通常是：

$$\text{结论描述 } \texttt{of} \text{ 条件 } 1 \ \texttt{of} \text{ 条件 } 2 \ \cdots$$

也就是说：**先把结论写在名字里，再用 `of` 把关键假设串起来**，让人只看名字就能大致猜到定理在说什么。

例子

```
#check lt_of_le_of_ne
-- 输出签名：∀ {a b : α}, a ≤ b → a ≠ b → a < b
```

```
-- 命名解析
-- 1. lt (Less Than)    : 结论是 a < b
-- 2. le (Less Equal)   : 条件1是 a ≤ b
-- 3. ne (Not Equal)    : 条件2是 a ≠ b
```

对于极少数“历史上约定俗成、且大家都认识”的大定理，mathlib 也会使用更接近传统名称的命名方式（比如`Cauchy_Schwarz_Inequality`）；但总体上仍会尽量保持可搜索、可读、可预测的风格（尤其在 API 规模很大时，这一点非常重要）。

后续写作业/项目时，建议你尽量遵循上述规则：这样不仅更符合社区习惯，也更利于自动补全、搜索、以及与他人协作（包括 AI 工具的检索与调用）。

2.5.1 常见符号与缩写速查表

Mathlib 的定理命名高度依赖于符号的英文缩写。熟练掌握下表中的对应关系，能够帮助你快速通过名称推测定理内容，或通过内容反推定理名称。

1. 逻辑与集合符号

符号	输入代码	命名片段	说明与备注
$\vee$	`\or`	`or`	
$\wedge$	`\and`	`and`	
$\to$	`\r`	`imp` / `of`	`imp` 指命题本身，`of` 指该命题作为前提
$\leftrightarrow$	`\iff`	`iff`	常省略，尤其是定义等价关系时
$\neg$	`\n`	`not`	
$\exists$	`\ex`	`exists`	`bex` (Bounded Exists) 在 Mathlib 中较少直接用于命名
$\forall$	`\fo`	`forall` / `all`	`ball` (Bounded All) 同上，多用于 Lean Core
$=$		`eq`	在定理名中经常省略
$\neq$	`\ne`	`ne`	
$\circ$	`\o`	`comp`	函数组合 (Composition)
$\in$	`\in`	`mem`	成员关系 (Member)
$\cup$	`\cup`	`union`	二元并集
$\cap$	`\cap`	`inter`	二元交集
$\bigcup$	`\bigcup`	`iUnion`	索引并集 (Indexed Union)
$\bigcap$	`\bigcap`	`iInter`	索引交集 (Indexed Intersection)
$\bigcup_0$	`\bigcup\0`	`sUnion`	集合的并集 ($\bigcup S$)
$\bigcap_0$	`\bigcap\0`	`sInter`	集合的交集 ($\bigcap S$)
$\setminus$	`\\`	`sdiff`	集合差 (Set Difference)
$\{x \mid p\,x\}$		`setOf`	集合构造
$\{x\}$		`singleton`	单元素集
$\{x, y\}$		`pair`	对集

2. 代数与运算符号

符号	输入代码	命名片段	说明与备注
0		`zero`	
1		`one`	
$+$		`add`	
$-$		`neg` / `sub`	`neg` 为取负 $(-a)$，`sub` 为减法 $(a-b)$
$*$		`mul`	一般乘法
$\cdot$	`\bu`	`smul`	数乘 (Scalar Multiplication)
$\hat{}$		`pow`	幂运算
$/$		`div`	除法
$^{-1}$	`\-1`	`inv`	群逆元
$\frac{1}{\cdot}$	`\frac1`	`invOf`	环中的单位元逆
$\mid$	`|`	`dvd`	整除 (Divides)
$\sum$	`\sum`	`sum`	有限和
$\prod$	`\prod`	`prod`	有限积

3. 序与格论符号

重要惯例：在 Mathlib 中，不等式几乎总是优先使用 $\le$ (`le`) 和 $<$ (`lt`)。即便你想表达“大于等于”，也通常会找到名为 `le` 的定理（例如 $b \ge a$ 会被写成 $a \le b$ 处理）。

符号	输入代码	命名片段	说明与备注
$<$		`lt`	Less Than
$\le$	`\le`	`le`	Less Equal
$\sqcup$	`\sup`	`sup`	上确界/并 (二元)
$\sqcap$	`\inf`	`inf`	下确界/交 (二元)
$\bigsqcup$	`\sqcup`	`iSup` / `ciSup`	`c` 表示条件完备 (Conditionally Complete)
$\sqcap$	`\sqcap`	`iInf` / `ciInf`	同上
$\bot$	`\bot`	`bot`	底元素 (Bottom)
$\top$	`\top`	`top`	顶元素 (Top)

4. 拼写惯例 (Spelling Conventions)

Mathlib 严格遵循**美式英语**拼写。

- 使用 **z** 而非 s：`factorization`, `localization` (勿用 *factorisation*)。
- 使用 **er** 而非 re：`fiber`, `center` (勿用 *fibre*, *centre*)。

2.5.2 命名空间 (Namespaces)

在数学形式化过程中，我们经常遇到名称冲突的问题。例如，自然数 $\mathbb{N}$、实数 $\mathbb{R}$ 以及格结构中都有“最小值”的概念，如果我们都将其命名为 `min_le_right`，系统将无法区分。

为了解决这个问题，Lean 引入了**命名空间 (Namespace)** 机制。

- **定义：**使用 `namespace` 关键字可以将一系列定义包裹在一个特定的作用域内。

- **效果**：在命名空间内定义的定理，其全名会自动加上空间名前缀。
- **调用**：在空间外调用时，需使用“点号表示法”（如 `Nat.add`）；或者使用 `open` 关键字打开空间。

```
-- 开启名为 my_theory 的命名空间
namespace my_theory

-- 定义一个定理，全名实际上是 my_theory.and_of_left_of_right
theorem and_of_left_of_right {p q : Prop} (hp : p) (hq : q) : p ∧ q :=
  And.intro hp hq

end my_theory -- 结束命名空间

-- 在外部调用时，必须加前缀
#check my_theory.and_of_left_of_right

-- 或者使用 open 打开它，之后便可直接使用
open my_theory
#check and_of_left_of_right
```

最佳实践：Mathlib 通常通过命名空间来组织不同数学结构的定理。例如，关于列表的定理都在 `List` 空间下，关于自然数的在 `Nat` 空间下。这也是为什么你会看到 `List.map` 和 `Option.map` 这类重名函数能和平共处的原因。

2.6 Mathlib 代码风格指南

Lean 社区维护了一套严格的风格指南（Style Guide）参考 https://leanprover-community.github.io/contribute/style.html，这不仅是为了美观，更是为了保证代码的可读性、可维护性以及编译器的稳定性。除了命名规范外，以下几个方面是编写高质量 Lean 代码的关键。

2.6.1 排版与空白 (Layout & Formatting)

代码的视觉布局应清晰且一致。虽然 Lean 对空白不敏感，但人类读者敏感。

- **行宽限制**：每行代码不应超过 **100 个字符**。过长的行应在适当的运算符处换行。
- **缩进**：
  - 必须使用 **2 个空格**进行缩进。
  - **严禁**使用 Tab 字符。
- **空格的使用**：
  - **二元运算符**前后需加空格。
  - **冒号** (:)、**赋值符** (:=) 前后需加空格。
  - **逗号** (,) 后需加空格。
  - **括号**内侧不加空格，函数调用时不加空格。

```
-- [错误示范] 拥挤、无空格、括号内有多余空格
def f(x:Nat):Nat:=x+1
#check ( f 5 )

-- [正确示范] 清晰的间隔
def f (x : Nat) : Nat := x + 1
#check f 5
```

2.6.2 显式与隐式参数 (Explicit vs Implicit Arguments)

在定义引理或函数时，参数括号的选择（(), {}, []）至关重要。

- **隐式参数** `{ }`：如果一个参数可以通过后续参数或返回类型**被推断出来**，应设为隐式。
- **显式参数** `( )`：如果一个参数无法被推断，或者经常需要用户手动提供，应设为显式。
- **类型类参数** `[ ]`：用于 `Group`, `Ring`, `Fintype` 等结构。

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

-- [错误示范] G 和 x 都能被推断，不应显式传入
-- 用户调用时被迫写 mul_inv_self G x, 非常繁琐
theorem mul_inv_self_bad (G : Type*) [Group G] (x : G) : x * x⁻¹ = 1 := ...

-- [正确示范] G 和群结构自动推断，x 由上下文决定
-- 用户调用时只需写 mul_inv_self x (甚至只写 mul_inv_self)
theorem mul_inv_self {G : Type*} [Group G] (x : G) : x * x⁻¹ = 1 := ...
```

2.6.3 证明风格 (Proof Style)

非终结 Simp (Non-terminal Simp)

这是 Mathlib 中最严格的禁忌之一。不要在证明中间使用不带参数的 `simp`。

- **原因**：`simp` 集合随着库的更新不断扩大。今天能跑通的 `simp`，明天可能会因为多引入了一个引理而把目标化简成不同的样子，导致后续的 `rw` 或 `exact` 失败（代码易碎）。
- **例外**：如果 `simp` 能直接闭合目标（即证明结束），则是允许的。
- **修正**：使用 `simp only [...]` 明确指定要用的引理，或者使用 `squeeze_simp` 辅助生成。

```
-- [脆弱的风格]
example : A = B := by
  simp           -- 危险！如果 simp 行为改变，下面这行可能会挂
  rw [h]

-- [推荐的风格]
example : A = B := by
  simp only [add_comm, add_zero]  -- 明确指定，稳如泰山
  rw [h]
```

结构化证明 (Structured Proofs)

避免写一长串难以理解的 tactic 链。优先使用 `have`, `let`, `calc` 等结构化工具，使证明过程对人类可读。

```
-- [难以阅读] 只有机器知道中间发生了什么
example : a * b * c = d := by
  rw [mul_assoc, mul_comm b, ←mul_assoc, h1, h2, h3]

-- [清晰明了] calc 块展示了推导思路
example : a * b * c = d := by
  calc a * b * c
    _ = a * (b * c) := by rw [mul_assoc]
    _ = a * d       := by rw [h_bc_eq_d]
    _ = d           := by rw [h_ad_eq_d]
```

2.6.4 文档规范 (Documentation)

代码不仅仅是写给编译器看的。

- **模块文档**：每个 Lean 文件的开头必须包含模块级文档，描述该文件的数学定义、主要定理和符号约定。
- **文档字符串 (Docstrings)**：每个顶层定义（`def`, `lemma`, `structure`）上方都应添加文档，使用 `/-- ... -/` 格式。
- **Markdown 支持**：文档中可以使用 Markdown 语法（如反引号引用代码）和 LaTeX 公式。

```
/--
如果一个群元素的平方是 1，则它是自己的逆。
注意：这是一个简单的示例引理。
-/
lemma inv_eq_self_of_sq_eq_one {x : G} (h : x * x = 1) : x⁻¹ = x := by
  ...
```

2.7 类型的层级：Universe

2.7.1 类型的类型

在前文中，我们已经学会了使用 `#check` 命令来查看一个项（Term）的类型：

```
#check 1   -- 1 : ℕ
```

既然 $1 : \mathbb{N}$，那么 $\mathbb{N}$ 本身也是一个对象，它的类型又是什么？同样的，$\mathbb{R}$ 和 `Prop` 的类型是什么？

```
#check ℕ   -- ℕ : Type
#check ℝ   -- ℝ : Type
#check Prop -- Prop : Type
```

结果显示：$\mathbb{N}$、$\mathbb{R}$ 这些"数学对象"在 Lean 中本身也是类型，而它们都属于同一个层级：`Type`。

2.7.2 宇宙层级

如果我们追问 `Type` 的类型是什么，会发现一个有趣的现象：Lean 不允许 `Type : Type` 这种循环包含。如果允许类型包含自身，将会导致类似集合论中"罗素悖论"的逻辑矛盾。

为了保证逻辑一致性，Lean 引入了**宇宙层级（Universe）**的概念，将类型划分为无穷向上的层级：

$$\texttt{Type 0} : \texttt{Type 1} : \texttt{Type 2} : \texttt{...}$$

我们在代码中常见的 `Type`，实际上只是 `Type 0` 的语法糖。为了在不同层级间复用代码（例如定义一个既能处理 `Type 0` 又能处理 `Type 1` 的列表），我们便需要引入**宇宙变量**（如常见的 `universe u`）。

`Sort`：逻辑与数据的统一

在更底层的实现中，Lean 将表示逻辑命题的 `Prop` 和表示数据的 `Type` 统一在了 `Sort` 体系之下。它们的对应关系如下：

$$\texttt{Prop} \equiv \texttt{Sort 0}, \qquad \texttt{Type u} \equiv \texttt{Sort (u+1)}$$

这意味着 `Prop` 和 `Type` 只是 `Sort` 家族在不同层级上的成员。`Sort 0` 是命题的世界，而 `Sort 1` 及以上则是数据类型的世界。我们可以通过以下代码来验证这种等价关系：

```
-- Type 的层级结构
#check Type     -- Type : Type 1
#check Type 1   -- Type 1 : Type 2

-- Prop 和 Type 在 Sort 体系中的位置
#check Sort 0   -- Prop : Type
#check Sort 1   -- Type : Type 1
#check Sort 2   -- Type 1 : Type 2

-- universe 变量实际上是 Sort 的参数
universe u
#check Sort u       -- Sort u : Type u
#check Sort (u+1)   -- Type u : Type (u + 1)
```

2.8 核心机制：函数类型

在类型论中，**函数类型**是构建一切复杂结构（包括逻辑蕴含和全称量词）的基石。

2.8.1 一元函数与 λ 构造

给定任意两个类型 α 与 β，我们可以构造出函数类型 $\alpha \to \beta$。

- 这是一个“从输入 α 得到输出 β”的程序。

要**构造**一个函数，我们使用关键字 `fun` 或符号 `λ`。以下三种写法是完全等价的：

```
variable (A B : Type) (f : A → B) (a : A)
#check f a    -- 函数应用不需要括号，使用空格左结合

-- 1. 标准写法
def f₁ : Nat → Nat := fun n => n + 1
-- 2. 使用 lambda 符号
def f₂ : Nat → Nat := λ n => n + 1
-- 3. 使用 mapsto 箭头
def f₃ : Nat → Nat := fun n ↦ n + 1
```

2.8.2 多元函数的本质：柯里化 (Currying)

在 Lean 中，**本质上不存在“多元函数”**。所有的多元函数实际上都是通过**柯里化**实现的：一个接受两个参数的函数，本质上是一个“接受第一个参数，返回一个‘接受第二个参数并返回结果的函数’的函数”。

理解这一点的关键在于掌握两个相反的结合规则：

- **类型构造是右结合的**：`A → B → C` 等价于 `A → (B → C)`。
- **函数应用是左结合的**：`f a b` 等价于 `(f a) b`。

虽然底层是嵌套的 λ 表达式，但 Lean 提供了语法糖，允许我们将多个参数写在同一个 `fun` 后面：

```
-- 语法糖写法 (推荐)
def add_simple : Nat → Nat → Nat := fun n m => n + m
```

2.8.3 依赖函数类型 (Π-type)

依赖函数是普通函数类型的推广。在普通函数 `A → B` 中，返回值类型 `B` 是固定的。而在依赖函数中，**返回值的类型可以依赖于输入参数的值**。

设 `A : Type`，`B : A → Type` 是一个类型族。对于输入参数 `a : A`，依赖函数的返回值类型为 `B a`。这种类型记作 `(a : A) → B a`，在 Mathlib 中常写作 `Π a : A, B a`。

```
variable (A : Type) (B : A → Type)
-- f 是一个依赖函数：给它不同的 a，返回值的类型 B a 也会随之变化
variable (f : (a : A) → B a)
```

当类型不依赖参数时

依赖函数类型 $\Pi x : A, B(x)$ 涵盖了普通函数类型：如果返回值的类型 B **并不实际依赖**于输入 x（即 B 是个常数类型），那么它就退化成了我们熟悉的普通函数箭头 $\to$。

```
universe u v
variable (A : Type u) (C : Type v)

-- 当 B(x) 恒等于 C 时：
-- (Π x : A, C) 等价于 (A → C)
-- 下划线 _ 表示我们忽略了输入参数 x，因为它不影响返回类型
example : (Π _ : A, C) = (A → C) := rfl
```

全称量词：命题世界里的 Π

在逻辑中，$\forall x : A, P(x)$ 意味着“对于任意的 x，$P(x)$ 都成立”。在 Lean 中，这被实现为一个依赖函数：**你给我一个 x，我通过函数返回给你一个 $P(x)$ 的证明**。

```
-- 命题：对于任意自然数 n，n 等于 n
def forall_example : ∀ n : Nat, n = n :=
  -- 证明就是一个函数：
  -- 输入是 n (自然数)
  -- 输出是 rfl (证明 n = n 的项)
  fun n => rfl
```

2.9 逻辑视角：命题即类型

有了上述函数工具，我们就可以理解类型论中最核心的思想——**Curry-Howard 同构**（Propositions as Types）。

2.9.1 蕴含即函数 ($\to$)

在 Lean 中，逻辑蕴含 $p \to q$ 实际上就是函数类型。

- **参数即假设**：函数的输入参数，对应定理的已知条件。
- **返回值即结论**：函数的返回值，对应定理要证明的目标。

这就是为什么同一个定理可以写得像 C++ 函数，也可以写得像数学公式：

```
-- 写法 1: 函数式风格
theorem logic_func {p q : Prop} (hp : p) (hq : q) : p ∧ q :=
  sorry

-- 写法 2: 逻辑式风格 (使用 →)
theorem logic_arrow : ∀ {p q : Prop}, p → q → p ∧ q :=
  sorry
```

2.10 证明：Term vs Tactic

既然定理是类型，证明就是构造该类型的项（Term）。我们有两种构造方式。

2.10.1 Term Proof：直接构造

当证明逻辑简单（如函数组合）时，我们可以直接写出那个项。这就像在搭积木：

```
variable (p q r : Prop)
-- 逻辑三段论 = 函数组合
example (h₁ : p → q) (h₂ : q → r) (hp : p) : r :=
  h₂ (h₁ hp)
```

解析：`h₁ hp` 产生了一个类型为 q 的项，再把它喂给 `h₂`，最终得到类型为 r 的项。

2.10.2 Tactic Proof：指令式构造

当证明涉及复杂的代数变形时，直接写出 Term 会极其痛苦。Tactic 模式允许我们使用指令（如 `rw`, `ring`）来逐步改变目标。

```
-- 痛苦的 Term 写法 (仅作反面教材)
example (a b c : ℝ) : a * b * c = c * b * a :=
  (mul_assoc a b c).symm ▸ (mul_comm c (a * b)) ▸ (mul_assoc c a b) ▸ ...

-- 清晰的 Tactic 写法
example (a b c : ℝ) : a * b * c = c * b * a := by
  ring
```

2.10.3 `show_term`：揭示真相

Tactic 并不是魔法，它们只是帮你写代码的“宏”。最终它们都会被编译器翻译成一个底层的项。使用 `show_term` 可以查看 Tactic 背后生成的复杂 Term：

```
variable (f : ℝ → ℝ) (a b : ℝ)

example (h : a = b) : f a = f b := by
  show_term {
    rw [h]
  }
  -- InfoView 输出:
  -- exact Eq.mpr (id (congrArg (fun _a => f _a = f b) h)) (Eq.refl (f b))
```

解析：你会发现，简单的 `rw [h]` 实际上构造了一个涉及 `Eq.mpr`（利用等式改写类型）和 `congrArg`（函数的参数全等性）的复杂函数调用。这就是 Tactic 存在的意义：把人类从繁琐的底层构造中解放出来。

2.11 Exercises

2.11.1 命名

请为以下的 examples 起一个合乎规范的名字（其实你们多数也见过了）：

```
variable {α : Type}

example {a : ℝ} : a + 0 = 0 := by sorry

example {a b c : ℝ} : (a + b) * c = a * c + b * c := by sorry

example {a b : ℝ} : a / b = a * b⁻¹ := by sorry

example {a b c : ℝ} : a ∣ b - c → (a ∣ b ↔ a ∣ c) := by sorry

example (s t : Set α) (x : α) : x ∈ s → x ∈ s ∪ t := by sorry

example (s t : Set α) (x : α) : x ∈ s ∪ t → x ∈ s ∨ x ∈ t := by sorry

example {a : α} {p : α → Prop} : a ∈ {x | p x} ↔ p a := by sorry

example {x a : α} {s : Set α} : x ∈ insert a s → x = a ∨ x ∈ s := by sorry

example {x : α} {a b : Set α} : x ∈ a ∩ b → x ∈ a := by sorry

example {a b : ℝ} : a ≤ b ↔ a < b ∨ a = b := by sorry

example {a b : ℤ} : a ≤ b - 1 ↔ a < b := by sorry

example {a b c : ℝ} (bc : a + b ≤ a + c) : b ≤ c := by sorry
```

请根据以下命名猜测并陈述出定理

```
/-
1. mul_add
2. add_sub_right_comm
3. le_of_lt_of_le
4. add_le_add
5. mem_union_of_mem_right
-/
```

2.11.2 tactics

```
variable (a b c d : ℝ)

#check pow_eq_zero
#check pow_two_nonneg
example {x y : ℝ} (h : x ^ 2 + y ^ 2 = 0) : x = 0 := by sorry
```

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

--你可以尝试使用aux来完成这一证明
#check le_antisymm
#check add_le_add_right
#check sub_eq_add_neg
example : min a b + c = min (a + c) (b + c) := by sorry

#check sq_nonneg
theorem fact1 : a * b * 2 ≤ a ^ 2 + b ^ 2 := by sorry

#check pow_two_nonneg
theorem fact2 : -(a * b) * 2 ≤ a ^ 2 + b ^ 2 := by sorry

--你可以使用上面两个定理来完成这一证明
#check le_div_iff₀
example : |a * b| ≤ (a ^ 2 + b ^ 2) / 2 := by sorry

-- Finish the proof using the theorems abs_mul, mul_le_mul, abs_nonneg, mul_lt_mul_right, and one_mul.
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
    _ < 1 * ε := sorry
    _ = ε := sorry

-- 使用calc
example (a b : ℝ) : - 2 * a * b ≤ a ^ 2 + b ^ 2 := by sorry

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

example {x y ε : ℝ} (epos : 0 < ε) (ele1 : ε ≤ 1) (xlt : |x| < ε) (ylt : |y| < ε) : |x * y| < ε := by sorry



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

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

section
variable (f g : ℝ → ℝ) (a b : ℝ)



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

```
  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) := by
  sorry

end
```