# Day 3

## 3.1 命名空间与点方法

本节介绍 Lean 里“点号 .”的两种截然不同但外观相似的用途。阅读代码时，我们需要根据上下文迅速判断它是指**层级结构**还是**函数调用**：

- **命名空间前缀 (Namespace Prefix)**：表示层级归属。例如 `Nat.add` 表示 `add` 定义在 `Nat` 命名空间下。
- **点方法 (Field/Dot Notation)**：一种函数调用的语法糖。例如 `x.bar` 实际上是 `Foo.bar x` 的简写。

### 3.1.1 命名空间 (Namespaces)

Day2 简要介绍了命名空间用于组织代码，防止命名冲突。除了基本的 `namespace` 和 `open` 指令外，还有几个实用的细节。

#### 1) 嵌套与简写

命名空间可以像文件夹一样嵌套。要在特定的命名空间内定义函数，我们既可以使用 `namespace` 代码块，也可以直接在定义名中使用点号。

```
namespace ns₁

-- 1. 标准写法：在命名空间块内定义
namespace ns₂
  def funcA : Nat → Nat := fun n => n
end ns₂

-- 2. 语法糖：直接使用点号前缀
-- 这等同于手动打开 namespace ns₂ ... end ns₂
def ns₂.funcB : Nat → Nat := fun n => n + 1

end ns₁

-- 由于funcA，funcB被嵌套在两层namespace里，访问时需写全完整路径
#check ns₁.ns₂.funcA
#check ns₁.ns₂.funcB
```

#### 2) `open` 的作用域

使用 `open` 指令可以省略前缀，但要注意它受当前 `section` 或 `namespace` 的作用域限制。

```
section
  open ns₁.ns₂
  #check funcA  -- OK: 在 section 内可以直接访问
end

-- #check funcA -- Error: 出了 section, open 失效, 必须写全名
```

**3)** scoped notation 与 open scoped

这是 Mathlib 中非常常见的模式。普通的定义（如函数）通常是局部的，但**记号（Notation）**默认是全局的。

```
namespace Physics
def energy (m c : Nat) := m* c * c

notation "E" => energy
end Physics

#check E 10 1 -- Physics.energy 10 1 : Nat

namespace Logic -- 假设我又想在逻辑学里定义 "E" 表示“存在 (Exists)”
def exists_prop := True

notation "E" => exists_prop

#check E -- 报错,  ambiguous, possible interpretations

end Logic
```

为了防止记号污染全局环境（例如不想让 $\infty$ 到处都是），我们使用 scoped notation。

**关键区别：**

- open MySpace：打开该空间下的所有普通定义（定理、函数），但不打开 scoped 记号。
- open scoped MySpace：专门用于打开该空间下定义的 scoped 记号。

```
namespace MyMath
  -- 定义一个 scoped 记号, 只有显式开启时才生效
  scoped notation "∞" => (0 : Nat)

  def myFunc (x : Nat) := x
end MyMath

-- #check ∞      -- Error: 记号未开启

section
  -- 只打开记号, 不打开 namespace 里的函数
  open scoped MyMath

  #check ∞       -- 0 : Nat
```

```
  -- #check myFunc -- Error: myFunc 依然需要前缀，因为没写 `open MyMath`
end
```

### 3.1.2 点方法 (Field Notation)

这是 Lean 中提升代码可读性的重要语法糖。如果一个命名空间名即为类型名，则对于该命名空间下的定理或定义，若其需要一个类型为该命名空间名的参数，我们可以直接写该参数调用相关定理，而不是在前面加上命名空间前缀。这使得代码更接近面向对象语言的写法。

其核心规则是：

**如果`x : T`，那么`x.f`会被解析为`T.f x`**

#### 1) 基本例子

当 Lean 看到 `x.bar` 时，它会检查 `x` 的类型（假设是 `Foo`），然后查找 `Foo.bar` 并将 `x` 作为第一个参数传入。

```
axiom Foo : Type

-- 命名空间名为类型名
def Foo.bar (_ : Foo) : String := "hello world!"

variable (a : Foo)

-- 以下两种写法完全等价：
#check Foo.bar a  -- 标准函数调用
#check a.bar      -- 点方法调用
```

第一行的 `axiom` 是用来声明公理的, **所谓公理, 就是无法证明对错的假设**, 这里相当于不管不顾、强行声明有这样一个类型 `Foo : Type` 存在, 为了下文的演示[1].

#### 2) 参数穿透

点方法不仅限于第一个参数。如果函数定义中有多个参数，Lean 会尝试把点号前的变量匹配给类型符合的位置，其余参数照常写在后面。

```
variable (f : Foo)

-- 参数在前
def Foo.add (self : Foo) (_ : Nat) := self
#check f.add 42       -- 等价于 Foo.add f 42

-- 类型的参数不在第一位
-- bar 定义为：先接收 Nat，再接收 Foo
def Foo.bar₁ (_ : Nat) (self : Foo) := self

-- Lean 能识别出 f 是 Foo 类型应该填入第二个位置
#check f.bar₁ 10       -- 等价于 Foo.bar₁ 10 f
```

[1]在正式的数学证明（如 Mathlib 开发）中，极度不推荐随意使用 axiom。因为如果你写了 axiom H : 1 = 0，虽然 Lean 会接受它（因为是公理），但这会导致整个逻辑系统崩溃

**3) 管道符与链式调用**

在处理多层嵌套时，可以使用管道符配合点方法。|> 是**管道运算符**，x |>.f 等价于 (x).f
例如：计算 0-99 之间偶数的平方，丢弃前 5 个，取接下来的 3 个求和。

- **普通写法**（逻辑反向，括号地狱）：

```
-- 必须从最里层的 range 100 开始读，视线由于括号不断跳跃
#eval List.sum (List.take 3 (List.drop 5 (List.map (fun x => x^2) (List.filter (fun x => x % 2 == 0) (List.range
    100)))))
```

- **点管道写法**（流式阅读）：

```
#eval List.range 100
  |>.filter (fun x => x % 2 == 0) -- 1. 选偶数
  |>.map (fun x => x^2)           -- 2. 平方
  |>.drop 5                       -- 3. 丢弃前5个
  |>.take 3                       -- 4. 取前3个
  |>.sum                          -- 5. 求和
```

**本节小结：**

- 看到 List.map 这种形式，它是**命名空间**索引。
- 看到 L.map f （其中 L : List），它是**点方法**，等价于 List.map f L。
- 使用 open scoped 来管理那些特殊的数学符号，保持环境整洁。

## 3.2 Lean 中的逻辑表达

### 3.2.1 命题逻辑

我们已经知道一个命题 P : Prop 在 Lean 中被视为一个类型：

- 如果命题 P 为真，意味着类型 P 非空，即存在一个元素 h : P。我们称 h 为命题 P 的**证明**。
- 如果命题 P 为假，意味着类型 P 为空，不存在任何元素。

命题之间可以通过逻辑运算符组合。Lean 支持以下各类运算符：

- 零元运算符（Prop）：True (真), False (假)
- 一元运算符（Prop → Prop）：¬ (否定)
- 二元运算符（Prop → Prop → Prop）：∧ (合取), ∨ (析取), → (蕴含), ↔ (等价)

下面我们介绍这些运算符在 Lean 中的定义及其对应的证明构造与使用方法。

1. True **(真)**

True 表示逻辑上的恒真命题。在 Lean 中，它有一个名为 trivial 的平凡证明。

```
#check True     -- True : Prop
#check trivial  -- trivial : True
```

2. `False` (假)

   `False` 表示逻辑上的恒假命题。在 Lean 中，它没有任何证明（即类型为空）。

   根据经典逻辑中的**爆炸原理**（Ex Falso Quodlibet），如果假命题成立，则可以推出任意命题。在 Lean 中，这体现为 `False.elim`：如果我们拥有一个 `False` 的证明 `h`（即导出了矛盾），就可以利用 `h` 构造出任意命题的证明。

```
#check False    -- False : Prop

example (h : False) : 1 = 2 := by
  exact False.elim h
```

3. `→` (蕴含)

   在 Lean 的类型论中，蕴含关系本质上是一个函数类型。要证明 `P → Q`，等价于构造一个函数：它接受一个 `P` 的证明作为输入，并返回一个 `Q` 的证明作为输出。反之，如果我们已知 `h : P → Q` 且拥有 `p : P`，通过函数应用 `h p` 即可得到 `Q` 的证明。

```
variable (P Q : Prop)
#check P → Q    -- P → Q : Prop
example (p : P) (h : P → Q) : Q := h p
```

4. `¬` (否定)

   命题 `¬P` 表示 `P` 的否定。在 Lean 中，否定并不是一个原语，而是被定义为“蕴含假”：

$$\neg P \equiv P \to \mathtt{False}$$

   这表示：如果你能提供 `P` 的证明，我就能推出矛盾（`False`）。因此，证明 `¬P` 就是构造一个从 `P` 到 `False` 的函数。

```
#check Not    -- Prop → Prop

variable (P : Prop)
#check ¬P       -- ¬P : Prop
example : (¬P) = (P → False) := rfl

example (p : P) (h : ¬P) : False := h p
```

   利用上一章介绍的点号记法，（对于蕴含关系 `h : P → Q`，我们可以直接使用 `h.mt` (Modus Tollens) 其本来面貌是 `Function.mt`，来获取其**逆否命题**的证明。

```
variable (h : P → Q)
#check h.mt           -- Function.mt h : ¬Q → ¬P
#check Function.mt    -- Function.mt {a b : Prop} : (a → b) → ¬b → ¬a
```

5. ∧ (合取)

P ∧ Q 表示 P 与 Q 同时成立。

**构造证明**：要证明 P ∧ Q，我们需要同时提供 P 的证明和 Q 的证明。可以使用构造函数 And.intro，或者更常用的尖括号语法 ⟨p, q⟩。

```
#check And    -- Prop → Prop → Prop

variable (P Q : Prop)
example (p : P) (q : Q) : (P ∧ Q) := by
  exact And.intro p q

-- 使用尖括号语法的等价写法
example (P Q : Prop) (p : P) (q : Q) : P ∧ Q := by
  exact ⟨p, q⟩
```

**使用证明**：如果我们已知 h : P ∧ Q，可以通过 And.left (或 h.left, h.1) 提取出 P 的证明，通过 And.right (或 h.right, h.2) 提取出 Q 的证明。

```
variable (h₁ : P ∧ Q)
#check h₁.left    -- h₁.left : P
#check h₁.right   -- h₁.right : Q
#check h₁.1       -- h.left : P
#check h₁.2       -- h₁.right : Q
```

6. ∨ (析取)

P ∨ Q 表示 P 成立或者 Q 成立 [2]。

**构造证明**：只需要证明其中一个命题成立即可。

- 使用 Or.inl (left) 通过 P 的证明来构造 P ∨ Q；
- 使用 Or.inr (right) 通过 Q 的证明来构造 P ∨ Q。

```
variable (P Q : Prop)
example (p : P) : P ∨ Q := by
  exact Or.inl p

example (q : Q) : P ∨ Q := by
  exact Or.inr q
```

**使用证明**：如果我们已知 h : P ∨ Q，想要证明目标 R，则必须进行分类讨论。在逻辑上，这意味着我们需要证明“如果 P 成立则 R 成立”以及“如果 Q 成立则 R 成立”。在 Lean 中，这一过程由 Or.elim 实现。

Or.elim 接收三个显式参数：1. 一个“或”命题 h : P ∨ Q（讨论的基础）；2. 从 P 推出的证明 P → R；3. 从 Q 推出的证明 Q → R。

[2]Lean 默认合取的优先级高于析取，比如 p ∧ q ∨ p ∧ r 其实是 (p ∧ q) ∨ (p ∧ r)

```
#check Or.elim  -- Or.elim {a b c : Prop} (h : a ∨ b) (left : a → c) (right : b → c) : c

example (P Q R : Prop) (h : P ∨ Q) (h₁ : P → R) (h₂ : Q → R) : R := by
  apply Or.elim h
  • exact h₁ -- 分支 1: 证明 P → R
  • exact h₂ -- 分支 2: 证明 Q → R
```

7. ↔ (等价)

`P ↔ Q` 表示 `P` 当且仅当 `Q`。从逻辑上讲，它等价于 `(P → Q) ∧ (Q → P)`。

**构造证明：**需要同时提供“前推后”和“后推前”两个证明。可以使用 `Iff.intro hpq hqp` 或尖括号 `⟨hpq, hqp⟩`。

```
variable (P Q : Prop)
example (hpq : P → Q) (hqp : Q → P) : P ↔ Q := by
  exact Iff.intro hpq hqp

-- 使用尖括号语法的等价写法
example (hpq : P → Q) (hqp : Q → P) : P ↔ Q := by
  exact ⟨hpq, hqp⟩
```

**使用证明：**如果已知 `h : P ↔ Q`，我们可以使用 `h.mp` (Modus Ponens) 获取 `P → Q`，使用 `h.mpr` (Modus Ponens Reverse) 获取 `Q → P`。

```
variable (h₂ : P ↔ Q)

#check h₂.mp      -- h₂.mp : P → Q
#check h₂.mpr     -- h₂.mpr : Q → P
#check h₂.1       -- h₂.mp : P → Q
#check h₂.2       -- h₂.mpr : Q → P
```

类似于合取，这里的 `.mp` 和 `.mpr` 也是点号记法的应用，对应定理 `Iff.mp` 和 `Iff.mpr`。

从上述代码的检查结果可以看出，所有这些逻辑运算符操作的结果仍然是 `Prop` 类型，这保证了命题系统的封闭性。

## 3.3 一些进阶 Tactics

### 3.3.1 suffices, assumption 与 refine

在这一节中，我们补充几个们能让我们的证明脚本更加简洁、灵活的 tactics。

#### 逆向推理：suffices

suffices顾名思义，它的含义是“为了证明当前目标，只需证明命题 $h$ 成立即可”。使用它会将证明过程分为两步：

1. 说明充分性：首先说明“为什么有了假设 $h$ 就能推出原目标”。
2. 证明新目标：然后转而去证明这个新的假设 $h$ 本身。

其基本语法格式为：

suffices 名称 (h) : 新的结论 from 通过 h 证明原目标

```
example (P Q R : Prop) (h₁ : P → R) (h₂ : Q) (h₃ : Q → P) : R := by
  -- 我们的目标是 R。
  -- 我们意识到：只要有了 P，配合已有的 h₁ : P → R，就能得到 R。
  suffices h : P from h₁ h

  -- 现在，原目标 R 已经被“解决”了（通过 from 部分）。
  -- 剩下的新目标变成了证明 P。
  exact h₃ h₂
```

suffices 与我们之前学过的 have 非常相似，两者的区别主要在于 ** 思维顺序 **：

- have 是前向推理（Forward）：先证明引理 $h$ 成立，再利用 $h$ 去证明最终目标。（“我有工具 $h$，所以我能解决问题。”）
- suffices 是后向推理（Backward）：先宣称引理 $h$ 足够解决问题，再去回头寻找 $h$。（“只要有工具 $h$ 就够了，现在我去造个 $h$。”）

我们可以在不同的情况下，根据直觉选择用起来更顺手的一个。

#### 自动搜索：assumption 与 rwa

assumption 是一个非常直观且省力的 tactic。它的作用是搜索当前上下文中的所有假设，如果发现其中有一个假设与当前的目标完全匹配，则自动完成证明。

最简单的例子如下：

```
example (a b : ℝ) (h : a = b) : a = b := by
  assumption -- 上下文中已经有了 h : a = b，直接匹配成功
```

稍微复杂一点的例子，结合了构造证明和反证法：

```
example {x y : ℝ} (h₀ : x ≤ y) (h₁ : ¬y ≤ x) : x ≤ y ∧ x ≠ y := by
  constructor
  · assumption -- 第一个分支 x ≤ y，上下文中已有 h₀，直接解决
```

```
  -- 反证法: x ≠ y 定义为 x = y → False
  intro h -- ⊢ False
  • apply h₁ -- ⊢ y ≤ x
    rw [h]   -- y ≤ y by rfl
```

**组合技：**与assumption相关的一个 tactic 是 rwa。它的名称来源于 Rewrite + Assumption。它的行为类似于 rw，但在执行完重写操作后，会自动尝试调用一次 assumption。这在“重写一步后目标恰好变成已知条件”的场景下非常有用。

```
example (n : ℕ) (h : Prime n) : Nat.Prime n := by
  -- Nat.prime_iff : Nat.Prime p ↔ Prime p
  -- 重写后目标变为 Prime n，正好与假设 h 匹配，rwa 一步到位
  rwa [Nat.prime_iff]
```

### 精准控制：refine

refine 的作用类似于apply, 但 refine 在使用定理时，需要提供所有的参数，而没有像 apply 一样的自动推断功能。

```
example (a b c : ℝ) : a + b + c = a + (b + c) := by
  refine add_assoc a b c
```

但它可以配合匿名构造子 ⟨...⟩ 使用，直接构建出目标的结构（如“且”、“存在”的骨架），而 apply 无法直接“应用”一个构造子。

```
example (P Q R : Prop) (h1 : P) (h2 : Q) (h3 : P → R) : Q ∧ R := by
  refine ⟨h2, h3 h1⟩
```

#### 利用占位符延迟证明

我们还可以通过 refine 先写出证明的“形状”，通过混合使用引理和占位符 ?_延迟证明，把注意力集中在当前能解决的部分，将复杂的逻辑留给后续步骤。

```
example (p q r t : Prop) (h1 : p → q) (h2 : q → r) (h3 : p) (h4 : t) : r ∧ t := by
  refine ⟨?_, h4⟩
  exact h2 (h1 h3)
```

#### 嵌套构造

我们甚至可以在一个 refine 留下的坑里，再次使用 refine 一层一层地剥洋葱。

```
example (p q r : Prop) (h : p ∧ (q ∧ r)) : (p ∧ q) ∧ r := by
  refine ⟨?_, ?_⟩
  -- 子目标 1: 证明 p ∧ q
  • refine ⟨?_, ?_⟩
    -- 子目标 1.1: 证明 p
    • exact h.left
    -- 子目标 1.2: 证明 q
    • exact h.right.left

  -- 子目标 2: 证明 r
  • exact h.right.right
```

**命名占位符**

比如当证明目标是 $\leftrightarrow$ 时，我们可以通过起名 `?yourname` 清晰地拆分“前推后”和“后推前”两个目标。

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  refine ⟨?forward, ?backward⟩
  -- 此时 Lean 生成了两个标记为 forward 和 backward 的子目标,我们可以自由选择先证明哪一个
  case forward =>
    intro h
    -- 将 h : p ∧ (q ∨ r) 拆解为 hp 和 (hq | hr)
    rcases h with ⟨hp, hq | hr⟩
    • left; exact ⟨hp, hq⟩
    • right; exact ⟨hp, hr⟩

  case backward =>
    intro h
    -- 将 h : (p ∧ q) ∨ (p ∧ r) 分情况拆解
    rcases h with ⟨hp, hq⟩ | ⟨hp, hr⟩
    • exact ⟨hp, Or.inl hq⟩
    • exact ⟨hp, Or.inr hr⟩
```

### 3.3.2 函数相关：funext，apply_fun 与 gcongr

本节介绍在处理函数相等性、函数应用以及不等式同余时的常用 tactic。

**函数相等性：funext**

在数学中，要证明两个函数 $f, g : A \to B$ 相等，我们通常诉诸于**函数外延性公理**（Function Extensionality）：如果对于任意的 $x \in A$，都有 $f(x) = g(x)$，那么 $f = g$。

在 Lean 中，`funext` (function extensionality) 正是用来实现这一逻辑的。

```
def f' (n : ℕ) := n + 1
def g (n : ℕ) := 1 + n

example : f' = g := by
  funext x  -- 目标变为: f x = g x
  rw [f', g]
  rw [Nat.add_comm]
```

如果我们直接使用 `funext` 而不提供变量名，Lean 会自动引入一个无法直接访问的匿名变量（通常在 InfoView 中显示为带剑号的变量，如 `x†`）。为了在后续证明中引用这个变量，我们需要使用 `rename` 将其“重命名”为可用的标识符。

```
example (A B : Type) (f g : A → B) : f = g := by   -- 目标: f = g
  funext                                           -- 目标变为: f x† = g x†
  rename A => x                                    -- 将类型为 A 的匿名变量重命名为 x
  sorry
```

`rename` 的语法格式为 `rename 类型名 => 新变量名`。它会在上下文中查找指定类型的匿名变量，并将其赋予新的名字。

### 函数应用：apply_fun

apply_fun 用于在等式或不等于的两边同时应用一个函数。根据它是作用于**假设**（正向）还是**目标**（逆向），其背后的数学逻辑和对函数的要求完全不同。

**1. 作用于假设 (at hypothesis)**

这是**正向推理**：我们已知某种关系，想推导出函数作用后的新关系。

**对于等式 $(x = y)$**：这是总是合法的。因为函数的定义保证了“若 $x = y$ 则 $f(x) = f(y)$”。

```
example (A B : Type) (x y : A) (f : A → B) (h : x = y)
  : f x = f y := by
  apply_fun f at h    -- 假设 h 变成了 `f x = f y`
  exact h
```

**2. 作用于目标 (at goal)**

这是**逆向推理**：我们想通过证明变换后的结论来反推原目标。这通常需要函数具有更强的性质，以保证逻辑的可逆性。

- **对于“不等于” $(x \neq y)$**：这是总是合法的。逻辑基础是逆否命题：“若 $f(x) \neq f(y)$ 则 $x \neq y$”。不需要额外条件。

```
example (A B : Type) (x y : A) (f : A → B) : x ≠ y := by
  apply_fun f   -- 目标变为 `f x ≠ f y`
  sorry
```

- **对于等式 $(x = y)$**：如果我们想将目标转化为 $f(x) = f(y)$，则必须保证 $f$ 是**单射**（Injective）。Lean 会自动生成证明单射性的子目标。

```
example (A B : Type) (x y : A) (f : A → B) : x = y := by
  apply_fun f
  • -- 子目标 1: f x = f y
    sorry
  • -- 子目标 2: Injective f
    sorry
```

### 处理不等式：gcongr

在处理不等式（如 $\leq$ 或 $<$）时，我们通常不使用 apply_fun，而是使用更现代、更强大的 gcongr (Generalized Congruence)。

当我们的目标形如 $f(a) \leq f(b)$，且我们希望将其简化为 $a \leq b$ 时，gcongr 会自动分析函数 $f$ 的结构并应用相应的单调性引理（它是一个自动化工具，在库里检索有 @[gcongr] 标签索引）。

**场景：剥离函数的“外壳”**

```
example (a b : ℝ) (h : a ≤ b) : a + 1 ≤ b + 1 := by
  -- 目标是证明 a + 1 ≤ b + 1
  -- gcongr 自动识别出两边都加了 1，这保持了不等式方向
  gcongr
```

```
example (a b : ℝ) (h : a ≤ b) : Real.exp a ≤ Real.exp b := by
  -- gcongr 自动识别出 Real.exp 是单调增的
  gcongr
```

**注**：这种“逆向”思维在 Lean 中非常常见。我们不是把假设 $a \le b$ 变成 $e^a \le e^b$（正向），而是把目标 $e^a \le e^b$ 简化回 $a \le b$（逆向），从而与假设匹配。

### 3.3.3 逻辑命题相关的实用 Tactic

在处理逻辑命题时，我们经常需要对假设进行拆解、利用矛盾进行推导，或者进行分类讨论。Lean 提供了一系列专门的 Tactic 来简化这些操作。我们将它们分为三类：结构拆解、反证与矛盾、以及分类讨论。

#### 1. 结构拆解：obtain 与 rintro

这两个 Tactic 的核心功能是**从复杂的命题中提取信息**，它们都支持类似模式匹配的语法。

1. `obtain`

   `obtain` 用于从现有的假设中“提取”出具体的成分。通俗地讲，它相当于 `have`（定义中间结论）加上 `rcases`（拆解）的组合。

   **典型场景**：当你有一个结构复杂的假设，你想把它拆成几个简单的小假设 (还是前面 refine 的例子):

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  refine ⟨?forward, ?backward⟩

  case forward =>
    intro h
    obtain ⟨hp, (hq | hr)⟩ := h  --  rcases h with ⟨hp, hq | hr⟩
    · left; exact ⟨hp, hq⟩
    · right; exact ⟨hp, hr⟩

  case backward =>
    intro h
    obtain (⟨hp, hq⟩ | ⟨hp, hr⟩) := h -- rcases h with ⟨hp, hq⟩ | ⟨hp, hr⟩
    · exact ⟨hp, Or.inl hq⟩
    · exact ⟨hp, Or.inr hr⟩
```

2. `rintro`

   `rintro` 是 `intro` 和 `rcases` 的组合体。它在引入假设的同时直接进行模式匹配拆解。优势：

   - `intro` 虽然支持简单的构造拆解（如 `⟨hp, hq⟩`），但**不支持**直接处理“或”命题（`∨`）。
   - `rcases` 只能作用于**上下文中已有**的假设。
   - `rintro` 则是在**引入**假设的同时直接完成拆解，并且支持处理 `∨`（使用 `(h1 | h2)` 语法），相当于前两者的结合。

   (还是这个例子，此时我们可以合并`intro` 和 `rcases`)

```
example (p q r : Prop) : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  refine ⟨?forward, ?backward⟩
  case forward =>
    -- 拆解外层的 ∧ 为 ⟨hp, ...⟩；拆解内层的 ∨ 为 hq | hr，并自动分两个分支
    rintro ⟨hp, hq | hr⟩
    • left; exact ⟨hp, hq⟩
    • right; exact ⟨hp, hr⟩
  case backward =>
    -- 因为顶层就是 ∨，所以用 ( ... | ... ) 括起来；同时把两边的 ∧ 也拆了
    rintro (⟨hp, hq⟩ | ⟨hp, hr⟩)
    • exact ⟨hp, Or.inl hq⟩
    • exact ⟨hp, Or.inr hr⟩
```

## 2. 反证与矛盾：exfalso, by_contra 与 contrapose

这一组 Tactic 处理的是否定（Negation）和假命题（False），是经典逻辑证明中的利器。

1. `exfalso`

   源自拉丁语 *ex falso quodlibet*（爆炸原理）。当你的假设中已经出现矛盾（或者你能轻易推出 `False`）时，使用 `exfalso` 将当前目标直接变为 `False`。这通常用于“不论目标是什么，只要前提矛盾就能得证”的情况。

```
example (p : Prop) (h : p ∧ ¬p) : 1 = 2 := by
  exfalso  -- 目标从 1 = 2 变为 False
  exact h.right h.left -- 利用 ¬p 和 p 构造 False
```

2. `by_contra`

   对应经典的**反证法**（Proof by Contradiction）。如果你想证明 $P$，`by_contra h` 会让你假设 $\neg P$（命名为 $h$），并将目标变为 `False`。

```
example (p q : Prop) (h : ¬q → ¬p) : p → q := by
  intro hp
  -- 我们想证 q。尝试反证法：假设 ¬q。
  by_contra hnq
  -- 现在我们需要寻找矛盾
  have hnp : ¬p := h hnq
  exact hnp hp -- p 和 ¬p 矛盾
```

3. `contrapose`. 它是指定了矛盾来源的反证法。使用 `contrapose hq` 会将当前目标与假设 `hq` **互换并取反**。即：它将证明目标变为 `¬q`，同时将假设 `hq` 更新为 `¬p` (利用了逆否命题 $q \to p \iff \neg p \to \neg q$)。

```
example (p q r : Prop) (hq : q) (hr : r) : p := by
  -- hq : q; hr : r ⊢ p
  contrapose hq
```

```
  -- hr : r; hq : ¬p ⊢ ¬q
  -- 目标与假设 hq 发生了互换并取反
  sorry
```

若加上叹号 (如 by_contra! 或 contrapose!)，Tactic 会自动对新生成的假设执行 push_neg，将否定符号推入命题内部，通常能得到更易处理的形式。

```
example (p q : Prop) (h : p → q) : ¬q → ¬p := by
  contrapose!
  exact h
```

### 3. 排中律与分歧：by_cases

by_cases 是最基础的分类讨论工具。对于任意命题 $P$，它利用排中律将证明分为 $P$ 成立和 $\neg P$ 成立两个分支。

**注：**Lean 默认采用构造主义逻辑，但在处理一般数学命题时，by_cases 会自动调用经典逻辑公理（Classical Logic），因此我们可以对任意命题使用排中律，而不必担心其是否可计算。

```
example (p q : Prop) : (p → q) ∨ (q → p) := by
  -- 这是一个经典的非构造性证明，必须讨论 p 是否成立
  by_cases h : p
  • -- 情况 1: 假设 p 成立 (h : p)
    right
    intro _ -- 忽略 q
    exact h
  • -- 情况 2: 假设 p 不成立 (h : ¬p)
    left
    intro hp
    exfalso -- 从 hp : p 和 h : ¬p 推出矛盾
    exact h hp
```

- 建议总是写成 by_cases h : p 的形式，以便给假设命名。
- **典型应用：边界讨论**。在处理除法或分段函数时，经常需要讨论变量是否为 0。

## 3.4 归纳与递归

### 3.4.1 归纳类型

在 Lean 的类型系统中，我们已经熟悉了如 `Prop`、`Type` 等宇宙层级，以及函数类型 `→`。那么，诸如自然数 `Nat`、布尔值 `Bool` 这样具体的基础数据类型是如何产生的呢？它们都属于**归纳类型** (Inductive Types)。

归纳类型可以被直观地理解为“上帝造物法则”：**定义一个类型，本质上就是在定义这个类型的所有可能值的构造方式**。只要列出了所有的“构造子”(Constructor)，也就穷尽了这个类型的所有可能性。

本节我们由简入繁展示如何定义归纳类型。

#### 1. 枚举类型：有限的集合

最简单的归纳类型是只包含有限个值的类型。在其他编程语言中，这通常被称为“枚举”(enum)。例如 `Bool` 只有 `true` 和 `false` 两个值。

我们以定义“三原色”为例，使用 `inductive` 关键字：

```
inductive Color where
  | Red    : Color
  | Yellow : Color
  | Blue   : Color
-- deriving Repr
```

**语法解析：**

- `inductive`：声明这是一个归纳类型。
- `where`：引出构造子列表（此处可省略，但建议保留以增加可读性）。
- `|` (竖线)：用于分隔不同的构造子。
- `Red`, `Yellow`, `Blue`：这是三个构造子的名字。在这里，它们不接受任何参数，直接成为了 `Color` 类型的值。

#### 2. 递归类型：从有限到无限

枚举类型只能处理有限的情况。而我们完全可以有更复杂的 constructor，只需将其改为函数类型。比如要表示无穷集合（如自然数），我们需要引入**递归**。如果一个构造子接受该类型本身作为参数，就能派生出无穷的值。

最经典的例子是自然数 `Nat` 的皮亚诺定义：

```
inductive Nat' where
  | zero : Nat'        -- 基础情况：0
  | succ : Nat' → Nat' -- 归纳步骤：后继
```

这里我们用 `Nat'` 以区别于标准库的 `Nat`。这个定义告诉我们：

1. `zero` 是一个 `Nat'`（对应数字 0）。
2. 如果你有一个 `Nat'` (假设记为 $n$)，那么 `succ n` 也是一个 `Nat'`。

通过这种方式，所有的自然数都可以被构造出来：

- $0 \Rightarrow$ `Nat'.zero`
- $1 \Rightarrow$ `Nat'.succ Nat'.zero`

- $2 \Rightarrow$ `Nat'.succ (Nat'.succ Nat'.zero)`
- ...

这种“不同的构造方式对应不同的值”的特性，是归纳类型的核心。

### 3. 携带数据：链表与参数化

构造子不仅可以递归，还可以携带其他类型的数据。让我们定义一个专门存储自然数的链表 `NatList`：

```
inductive NatList where
  | nil  : NatList                    -- 空链表
  | cons : Nat → NatList → NatList    -- 头部存一个 Nat，尾部接续 NatList
```

这虽然有用，但不够通用。我们显然不希望为 `Bool`、`String` 都单独定义一种链表。Lean 允许我们使用**类型参数** (Type Parameters) 来定义通用的归纳类型。

我们可以定义一个对任意类型 `α` 都通用的列表 `List'`：

```
inductive List' (α : Type) where
  | nil  : List' α
  | cons : α → List' α → List' α
```

此时，`List'` 本身成为了一个函数 `Type → Type`。只有当我们填入具体的类型参数（如 `List' Nat`）时，它才成为一个具体的归纳类型。

另一个常见的参数化类型是 `Option` (选项类型)，它用于处理可能为空的值（类似其他语言中的 null/nil 处理机制）：

```
inductive Option' (α : Type) where
  | none : Option' α      -- 表示“无值”
  | some : α → Option' α  -- 表示“有值”，并携带该值
```

### 4. 结构体：单构造子的特例

有些时候，我们需要把多个不同类型的数据打包在一起（逻辑上的“且”关系）。例如，一个学生信息包含姓名、年龄**和**成绩。

虽然我们可以用 `inductive` 定义一个只有一个构造子的类型来实现：

```
inductive Student where
  | mk : String → Nat → Nat → Student
```

但这有一个明显的缺点：参数是位置相关的。我们无法直观地知道这两个 `Nat` 哪个是年龄，哪个是成绩。

对于这种“单构造子”情形，Lean 提供了语法糖 `structure`。

```
structure Student where
  name  : String
  age   : Nat
  grade : Nat
  deriving Repr
```

使用 `structure` 后，我们可以清晰地构造实例并访问数据：

```
def s : Student := { name := "Alice", age := 2, grade := 1 }

#eval s.name -- 输出 "Alice"
```

- **字段访问**：Lean 会自动为每个字段生成投影函数。写法 `s.name` 本质上是函数调用 `Student.name s` 的语法糖。这避免了我们手写模式匹配来提取数据。
- **底层实现**：它依然是一个归纳类型。Lean 会自动生成一个默认名为 `mk` 的构造子（全称 `Student.mk`），其类型为 `String → Nat → Nat → Student`。

**5. 命题与类型的统一：同构与截断**

回顾“命题即类型”(Curry-Howard 同构)，归纳类型在数据构造与逻辑证明中扮演了相同的代数角色。我们可以通过下表直观地看到这种统一性与差异：

| | **Type (数据宇宙)** | **Prop (命题宇宙)** |
|---|---|---|
| **乘法结构 ($\times$)** | Product / Structure | And ($\wedge$) |
| **加法结构 ($+$)** | Sum ($\oplus$) | Or ($\vee$) |

我们以**加法结构**为例：

```
-- Sum: 两个类型的不交并 (Disjoint Union)
inductive MySum (α β : Type) where
  | inl : α → MySum α β
  | inr : β → MySum α β

-- Or: 两个命题的析取 (Disjunction)
inductive MyOr (a b : Prop) where
  | inl : a → MyOr a b
  | inr : b → MyOr a b
```

1. 可以看到两者的构造子 (`inl`/`inr`) 完全一致，都对应集合论中的**不交并** $A \sqcup B$。本质上，它们都是通过“注入”(Injection) 来构造对象。
2. 区别在于它们所处的**宇宙**不同，导致了信息的处理方式不同：
   - **Sum (保留信息)**：在 `Type` 中，`inl x` 与 `inr y` 是严格区分的。我们可以通过模式匹配提取出 $x$ 或 $y$，这对程序计算至关重要。
   - **Or (擦除信息)**：在 `Prop` 中，Lean 强制执行**证明无关性**。这相当于对不交并进行了**命题截断** (Quotient)。无论证明来自左侧还是右侧，其携带的信息都被“商”掉了，仅保留“存在证明”这一事实。

### 3.4.2 模式匹配

现在，我们讨论如何使用这些归纳类型。这涉及到函数式编程中核心的机制：**模式匹配** (Pattern Matching)。在 Lean 中，模式匹配主要通过 `match` 关键字实现。值得注意的是，这种逻辑在编程和证明中是统一的：

- **在函数定义中**：我们使用 `match` 来根据数据的不同构造形式返回不同的值。
- **在定理证明中** (Tactic Mode)：我们使用 `cases` 策略。它的底层逻辑与 `match` 完全一致，都是对目标进行拆解和分类讨论。

我们对上一节中的例子逐一应用模式匹配，以展示其用法。

### 1. 基础匹配与相等性判定

以仅有三个值的 `Color` 类型为例。如果我们想要定义一个函数，以一个 `Color` 作为输入，对不同的颜色输出不同的结果，初学者可能会想到使用 `if...then...else`：

```
def color2num (c : Color) :=
  if c == Color.Red then 1 else
    if c == Color.Yellow then 2 else 3
```

如果你直接运行这段代码，Lean 会报错。这是因为在 Lean 中，当你定义一个新的类型时，它默认是没有任何能力的。它并不知道如何判断两个 `Color` 是否相等。为了让 `==` 生效，我们需要在定义类型时派生 (derive) 相应的实例：

```
inductive Color where
  | Red     : Color
  | Yellow  : Color
  | Blue    : Color
deriving BEq, DecidableEq
```

- `BEq` (Boolean Equality)：允许我们使用 `==` 运算符进行计算，返回 `Bool`。
- `DecidableEq` (Decidable Equality)：允许我们判定命题层面的相等 `=`，返回 `Decidable` 证据。

虽然 `if` 语句行得通，但它写起来显得冗长且不够直观。使用 `match` 关键字，我们甚至不需要上述的 `BEq` 实例，也不需要进行相等性运算，而是直接根据**构造子** (Constructor) 的结构进行拆解：

```
def color2num (c : Color) : Nat :=
  match c with
  | .Red    => 1
  | .Yellow => 2
  | .Blue   => 3
```

**语法细节：**

- **点号前缀：** `.Red` 中的点号是 Lean 的语法糖，它允许 Lean 根据上下文 (`c` 的类型) 自动推断命名空间，省略了 `Color.Red` 中的前缀。
- **穷举性：** `match` 必须覆盖所有可能的情况。对于有限的枚举类型，这很容易做到。

### 2. 通配符与简写

如果我们只关心部分情况，可以使用下划线 `_` 作为**通配符**，匹配“所有剩余的情况”：

```
def isRedOrBlue (c : Color) : Bool :=
  match c with
  | .Red  => true
  | .Blue => true
  | _     => false  -- 匹配 Yellow
```

Lean 还提供了一种更紧凑的写法，允许我们在定义函数时直接省略 `match` 关键字，直接在参数列表后进行匹配。这在多参数匹配时尤为方便：

```
-- 这是一个类型为 Color → Color → Bool 的函数
def isRedAndBlue : Color → Color → Bool
  | .Red, .Blue => true
  | _, _        => false
```

### 3. 提取构造子参数

当归纳类型的构造子携带数据时（如 Nat 的 succ，或者结构体的字段），模式匹配不仅能区分情况，还能**提取**其中的数据。

**递归类型的解构：**以自然数的前驱函数 pred 为例：

```
def pred : Nat → Nat
-- 情况 1: 如果输入是 0
  | .zero   => .zero
-- 情况 2: 如果输入是“某个 k 的后继”
  | .succ k => k

#eval pred 0
#eval pred 3
```

在这里，.succ k 模式成功匹配后，变量 k 就代表了被包裹在 succ 中的那个自然数。

**结构体的解构：**

对于我们之前用 inductive 定义的 Student 类型（非 structure 语法糖版），我们无法直接用点号访问字段。但通过模式匹配，我们可以手动提取数据：

```
-- Student 的定义回顾: | mk : String → Nat → Nat → Student

def Student.name : Student → String
  | .mk name _ _ => name  -- 提取第一个参数，忽略后两个

def Student.age : Student → Nat
  | .mk _ age _ => age    -- 提取第二个参数

def Student.grade : Student → Nat
  | .mk _ _ grade => grade -- 提取第三个参数

def s := Student.mk "Alice" 20 100
#eval s.name
```

这揭示了 structure 的本质：structure只是归纳类型的一个特例（只有一个构造子），而 .name 等访问器本质上就是 Lean 自动生成的上述模式匹配函数。

### 4. 综合示例：数据与逻辑

最后，我们来看一些依赖归纳类型的进阶例子。观察它们是如何处理类型参数和递归结构的。

**和类型 (Sum) 与列表 (List)：**

```
-- 这是一个处理 MySum Nat (List Nat) 的函数
-- 如果左边是数字则返回该数字，如果右边是列表则计算列表和
```

```
def sum (x : MySum Nat (List Nat)) : Nat :=
  match x with
  | .inl n => n
  | .inr l => l.sum

-- 获取列表的最后一个元素（如果存在）
-- 这里展示了递归匹配 (recursive matching)
def List'.end {α : Type} : List' α → Option' α
  | .nil          => .none
  | .cons x .nil  => .some x    -- 匹配只有一个元素的情况
  | .cons _ l     => l.end      -- 递归调用
```

**逻辑命题 (Or)：**

既然 `MyOr` (即命题层面的 `v`) 也是归纳类型，我们同样可以用 `match` 来使用它。在逻辑上，这对应于**析取消除** (Disjunction Elimination)。

```
-- 已知 A 或 B 成立，且 A 不成立，则 B 必然成立
def MyOr.resolve_left {a b : Prop} (h : MyOr a b) (ha : ¬a) : b :=
  match h with
  | .inl h_a => absurd h_a ha  -- 情况1: h 是 A 的证明。与 ¬A 矛盾，推出任意结论
  | .inr h_b => h_b            -- 情况2: h 是 B 的证明。直接返回
```

这里的 `absurd` 是一个利用矛盾 (`False`) 推出任意结论的函数，这在处理“不可能发生的分支”时非常有用。通过这些例子，我们可以深刻体会到 Lean 中“证明即程序”的统一美感。

## 3.5 Exercises

自己定义一个归纳类型，并使用 inductive 关键词与额外的函数定义实现 structure 类似效果。

使用refine的练习题：

```
example {p q : Prop} (h1 : p → q) (h2 : q → p) : p ↔ q := by sorry

#check Or.inl
example {p q r t : Prop} (h1 : q) (h2 : r) (h3 : t) : (((p ∨ q) ∧ r) ∨ p) ∧ t := by sorry

#check ne_of_lt
example {x y : ℝ} : x ≤ y ∧ ¬y ≤ x ↔ x ≤ y ∧ x ≠ y := by sorry
```

使用suffices的练习题

```
#check Nat.even_pow'
example (n : ℕ) : Odd (n ^ 2 + n + 1) := by sorry

example (n : ℕ) : Even ((n ^ 3 + 97 * n ^ 2 + 38 * n + 8) * (n ^ 2 + n)) := by sorry

example (s : Set ℕ) (h : {n | 3 ∣ n} ⊆ s) : ∃ x ∈ s, Even x := by sorry
```