Day 1

1.1 Getting started

课程网页：https://github.com/ronggexu/lean4_workshop_26winter
参考 README 安装 lean4:v4.26.0

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

本教程使用 Lean 4 及其主流数学库 **mathlib**。我们默认在文件开头，

```
import Mathlib
```

它会导入 mathlib 中已经形式化好的大量定义、定理与证明工具。

常用链接

- Lean community 主页：`https://leanprover-community.github.io`
- Lean 社群 Zulip 论坛：`https://leanprover.zulipchat.com`
- Lean Game Server：`https://adam.math.hhu.de`
- 在线使用 Lean：`https://live.lean-lang.org/`
- Mathlib 文档：`https://leanprover-community.github.io/mathlib4_docs`

1.2 Overview

我们先建立一个最重要的直觉：
在 Lean 中，任何表达式都有唯一的类型。
例如，数字有类型。可以看到。默认情况下，Lean 会把 `1` 视为自然数，$\sqrt{2}$ 的类型则是实数 $\mathbb{R}$。：

```
#check 1        --1 : ℕ
#check √2       -- √2 : ℝ
```

你可以用 `#check` 命令查看某个表达式的类型：把表达式写在 `#check` 后面，然后在 infoview 中读取输出。Lean 的输出通常是“`A : B`”的形式：冒号左边是项（term），右边是它的类型（type）。

```
-- Every expression has a type, you can use the #check command to print it.
#check 1 + 1


def f (x : ℕ) :=
```

```
  x + 3

#check f
```

例如上面 `1 + 1` 是一个表达式，因此也有类型；而 `f` 是一个函数定义（输入 `x : ℕ`，输出 `x + 3`），因此 `#check f` 会显示它的函数类型。

除了“数据”和“函数”，**命题**在 Lean 中也同样是表达式，并且它们的类型是 `Prop`：

```
-- These are propositions, of type `Prop`.
#check 2 + 2 = 4

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

#check FermatLastTheorem
```

这里 `2 + 2 = 4` 是一个命题，因此它的类型是 `Prop`。同理，`FermatLastTheorem` 虽然看起来像“一个很长的句子”，但它仍然只是一个表达式：它的值是一个命题，所以它的类型也是 `Prop`。

更进一步，Lean 把“证明”也当作表达式：**如果 P 是一个命题（即 P : `Prop`），那么“证明 P”就是一个类型为 P 的项**。换句话说，证明的类型就是它所证明的命题：

```
-- Some expressions have a type, P, where P itself has type Prop.
-- These are proofs of propositions.
theorem easy : 2 + 2 = 4 :=
  rfl

#check easy

theorem hard : FermatLastTheorem :=
  sorry

#check hard
```

例如，`easy` 是一个项，它的类型是命题 `2 + 2 = 4`；这表示 `easy` 就是该命题的一个证明。这里的 `rfl` 可以理解为“反身性证明”：当等式两边在化简后相同，`rfl` 往往就能直接完成证明。类似地，`hard` 的目标类型是 `FermatLastTheorem`，此处先用 `sorry` 占位（表示“暂时承认”）；在正式成果或作业提交中，一般应避免保留 `sorry`。

从这些例子可以看到：**Lean 里“数学对象 / 命题 / 证明”都统一成“项 + 类型”的语言**。接下来我们会逐步解释：常见的数学概念在 Lean 的类型论框架下分别如何表达与使用。

1.3 定理的结构

下面我们看一个“完整的 Lean 定理 + 证明”的例子：

```
theorem The_first_example {G : Type*} [Group G] {a b c d e f : G} (h : a * b = c * d) (h' : e = f) : a * (b * e) = c *
    (d * f) := by
  rw [h']
  rw [← mul_assoc]
  rw [h]
  exact mul_assoc c d f
```

建议你先在本地运行这段代码，并在编辑器中点选不同位置观察 infoview：你会看到当前有哪些参数与假设，以及当前需要证明的目标如何随证明步骤变化。

从整体结构上看，Lean 中一个定理由两部分组成：

- **陈述（statement）**：说明在什么参数与假设下，要证明什么结论；
- **证明（proof）**：给出完成该结论的证明脚本。

本小节先聚焦在**陈述部分**；证明脚本的细节（例如 `rw`、`exact` 等 tactic）会在后续章节讲解。

读懂定理陈述的骨架

把上面第一行按结构拆开，你会发现它基本就是下面的模板：

`theorem` 名称 参数/假设 `:` 目标 `:=` `by` 证明

逐段解释：

- `theorem`：声明接下来要给出一个定理（并提供其证明）。
- `The_first_example`：定理名字，由你自己命名（同一文件/命名空间内不能重名）。
- `{G : Type*} [Group G] {a b c d e f : G} (h : ...) (h' : ...)`：参数与假设。
- `: a * (b * e) = c * (d * f)`：目标（要证明的结论）。注意：**定理的类型就是它的结论**。
- `:= by`：从这里开始进入证明部分。可以暂时把它当作一个固定搭配：`:=` 表示“接下来给出这个定理（这个项）”，`by` 表示“用 tactic 风格写证明”。

就数学含义而言，这个定理在说：“设 G 是一个群，$a, b, c, d, e, f \in G$。给定假设 $h : ab = cd$ 与 $h' : e = f$，则有 $a(be) = c(df)$。”

infoview 在证明过程中显示什么

在证明过程中，infoview 会持续展示“当前上下文”和“当前目标”。例如当光标停在 `rw [h']` 之后，infoview 可能显示：

```
/-
G : Type u_1
inst✝ : Group G
a b c d e f : G
h : a * b = c * d
h' : e = f
⊢ a * (b * f) = c * (d * f)
-/
```

符号 ⊢ 右边是**当前需要证明的目标**，左边（上面）是**当前可用的参数与假设**。随着证明步骤推进，目标会逐步被改写直至出现 `No goals`，表示证明完成。

同一例子：把陈述部分标出来

下面把同一个例子在第一行用注释标出陈述的几个区块（注意:为避免重名,这里把定理名改成 The_first_example'）：

```
theorem /-name-/The_first_example' /-parameters-/{G : Type*} [Group G] (a b c d e f : G) (h : a * b = c * d) (h' : e =
    f) : /-goal-/a * (b * e) = c * (d * f) := by/-proof-/
  sorry
```

小结

一个定理 = 陈述 + 证明。其中陈述部分的基本写法是：

theorem + 名称 + 参数/假设 + : + 目标

理解并熟悉这一骨架后，我们就可以逐步学习：如何写参数（花括号/方括号/圆括号的区别）、如何组织假设，以及如何用 tactic 构造证明。

1.4 定理陈述中的参数

本节解释定理陈述中三种括号的含义：花括号 { }、圆括号 ( )、方括号 [ ]。核心结论先说在前面：

- {...}：**隐式参数**（通常可由 Lean 自动推断，使用时一般不需要手动填写）；
- (...)：**显式参数**（使用时通常需要手动提供）；
- [...]：**类型类参数**（要求某个类型携带特定结构，Lean 会用类型类推断自动寻找实例）。

1.4.1 隐式参数与显式参数

先看一个例子：

```
theorem my_lt_trans {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans
  apply h1
  apply h2
```

这里同时出现了花括号和圆括号：它们都在给定理提供“输入”，区别在于

- {a b c : ℕ} 是**隐式参数**：Lean 通常可以从上下文推断出它们；
- (h1 : a < b) (h2 : b < c) 是**显式参数**：它们是我们明确给定的假设，作为定理的输入。

理解方式可以把“定理”看作一个把若干输入映射到结论的对象：给定所需的假设，就能得到结论。比如 lt_trans 的直觉就是“如果 $a < b$ 且 $b < c$，则 $a < c$”。在 Lean 里，你可以直接把两个证明喂给它：

```
theorem one_lt_two' : 1 < 2 := by sorry
theorem two_lt_three' : 2 < 3 := by sorry

example : 1 < 3 := lt_trans one_lt_two' two_lt_three'
```

注意：这里我们没有手动提供 a, b, c。原因是，当你给出 `one_lt_two' : 1 < 2` 与 `two_lt_three' : 2 < 3` 时，其中已经包含了 $a = 1, b = 2, c = 3$ 这些信息，Lean 可以**自动**从这两个假设中把隐式参数推断出来。因此在陈述 `my_lt_trans` 时，我们把 a, b, c 放进花括号里，表示：**这些参数在使用定理时通常不需要显式填写，让 Lean 自动推断即可。**

顺便一提，上面出现的 `example` 和 `theorem` 类似，区别在于 `example` 不需要命名；我们会在后续再细讲它们的用途差异。

1.4.2 补充：def 的陈述格式

除了定理（`theorem`），我们也经常需要定义函数/对象（`def`）。`def` 的骨架与 `theorem` 很像，同样是“关键词 + 名称 + 参数 + : + 类型 + := + 内容”。区别在于：

- `theorem` 的“`:` 后面”是**要证明的命题**，而“`:= by` 后面”是**证明**；
- `def` 的“`:` 后面”是**返回值的类型**，而“`:=` 后面”是**具体返回的对象/表达式**。

例如：

```
def div_two {a : ℤ} (h : Even a) : ℤ := a / 2
```

这里 `{a : ℤ}` 是隐式参数，`(h : Even a)` 是显式参数；整体表示：给定一个整数 a（通常可从上下文推断）以及它是偶数的证明 `h`，返回整数 `a / 2`。

1.4.3 方括号：结构、class 与类型类推断

现在解释方括号 `[ ]`。它通常出现在形如 `[Group G]`、`[TopologicalSpace X]` 这样的地方，表示：**我们要求某个类型携带一份由 `class` 声明的结构，并让 Lean 自动寻找对应的实例（instance）。**

为了理解这个机制，需要先区分 `structure` 与 `class`：

- `structure` 用来把若干数据/性质“打包”成一个新的结构（你需要显式构造它）；
- `class` 也是打包结构，但它会进入 Lean 的“类型类推断系统”：当你写 `[SomeClass α]` 时，Lean 会尝试自动合成（synthesize）出该结构在 α 上的实例。

用 `structure` 打包信息 例如三维点可以用三个坐标打包：

```
structure Point where
  x : ℝ
  y : ℝ
  z : ℝ
```

再例如“线性”可以理解为一个函数满足若干性质（这里用两个性质做演示）：

```
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
```

用 class + instance 让 Lean 自动“记住”结构 当某些结构在特定类型上是约定俗成的（例如 $\mathbb{N}$ 上的 $\leq$），我们通常希望 Lean 自动使用默认实例，而不是每次都手动传入。这就是 class 的用途：把结构交给类型类系统管理，并用 instance 为具体类型登记实例。

```
-- Remember that a partial order is a binary relation that is transitive, reflexive, and antisymmetric. An even weaker
    notion sometimes arises: a preorder is just a reflexive, transitive relation.

class PreOrder' (α : Type*) where
  le : α → α → Prop -- 它接收两个类型为 α 的元素，并返回一个命题
  le_refl : ∀ a : α, le a a
  le_trans : ∀ a b c : α, le a b → le b c → le a c

instance : PreOrder' ℕ  where
  le := sorry
  le_refl := sorry
  le_trans := sorry

#synth PreOrder' ℕ
```

上面这段的阅读方式是：

- class PreOrder' 定义了“预序”所需的数据与公理；
- instance : PreOrder' ℕ 登记了自然数上的一个预序实例（这里用 sorry 占位）；
- #synth PreOrder' ℕ 让 Lean 尝试自动合成该实例，用来检验登记是否成功。

回到 [Group G] 的含义 因此，当你在定理陈述里写 [Group G] 时，含义就是：**假设 G 携带一个群结构（这是一个由 class 声明的结构），并让 Lean 在需要时自动从上下文中找到它。**这正是第一节例子里 {G : Type*} [Group G] 的意义。

小结

- 花括号 {...}：隐式参数，Lean 通常可推断，使用时一般不需手动填写；
- 圆括号 (...)：显式参数，使用时通常需要你明确提供；
- 方括号 [...]：类型类参数，要求某个 class 结构，并由类型类推断自动寻找 instance。

此外，def 的陈述骨架与 theorem 类似，都是“关键词 + 名称 + 参数 + : + 类型 + := + 内容”，只是 theorem 的内容是证明，而 def 的内容是返回值本身。

1.5 基本计算和等式证明

要在 Lean 中证明定理，你需要先掌握两件事：

- **可用的已有定理来自哪里**：大量基础定义与常用引理都已收录在 **mathlib** 中，并按数学分支分文件组织。
- **如何把这些定理用起来**：Lean 提供了一套面向证明过程的脚本语法，称为 **tactic**。除了 tactic 风格，Lean 也支持直接写“项”的证明（term proof）；我们后面会比较两者的差别与各自适用场景。

本节先介绍五个最常用、足以覆盖大量入门证明的 tactic：exact、rfl、apply、rw 和 have。掌握它们之后，你就可以开始独立形式化许多简单的计算与等式类定理。

1.5.1 exact 和 rfl

`exact`

当我们的目标已经是库中某个定理（或引理）的结论时，可以用 `exact` 直接把它“交上去”，从而一步完成证明。例子如下：

```
#check mul_assoc
example (a b c : ℝ) : a * b * c = a * (b * c) := by
  exact mul_assoc a b c
```

这里 `mul_assoc` 是乘法结合律。通过 `#check mul_assoc` 你会看到它需要三个显式参数，并给出结论 `a * b * c = a * (b * c)`[1]。在上面的 `example` 中，我们的目标恰好就是这个结论（只是把变量具体命名为 `a b c`），因此把参数填好以后，`mul_assoc a b c` 的类型正是我们要证明的目标，于是 `exact` 可以一步结束证明。

需要强调的是：`exact` 只能在**你给出的项的类型与当前目标完全一致**时使用。也就是说，它本质上是“一步收尾”的 tactic：你必须已经拿到了一个能直接证明当前目标的项（通常就是某个定理应用到合适参数后的结果）。如果你希望“先用某个定理把目标化简/转化为另一个子目标”，但还不能立刻结束证明，那么就应当使用 `apply`（`exact` 可以看作 `apply` 的特殊情况），我们马上会介绍它。

`rfl`

另一个最常见的“收尾”tactic 是 `rfl`。它用于证明形如“左右两边本来就是同一个表达式”的等式，例如：

```
example (x y : ℝ) : x + 37 * y = x + 37 * y := by rfl
```

表面上看这有些“平凡”，但 `rfl` 的价值远不止于此：在许多情况下，等式两边经过定义展开或化简后会变得完全相同，此时 `rfl` 也能直接完成证明。我们后面会用更有代表性的例子展示这一点。

1.5.2 apply 和 rw

`apply`

`apply` 与 `exact` 都是“使用已有定理”的 tactic。最简单的情形是：你把定理连同所需参数都填好，使其结论恰好与当前目标一致，那么 `apply` 也能一步结束证明：

```
theorem my_lt_trans' {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans h1 h2
```

回忆 `lt_trans` 表达“$(a < b)$ 且 $(b < c)$ 推出 $(a < c)$”。当我们把 `h1` 和 `h2` 依次喂给它时，就得到了目标 `a < c`，因此证明完成。直觉上，`apply` 做的是一种**反向推理**：它先匹配引理的结论与当前目标，然后把引理的前提变成需要继续证明的子目标（如果这些前提不能立刻由你提供的话）。

与 `exact` 的关键差别在于：`apply` 更灵活——你不必一次把所有参数都填满。Lean 会尝试自动匹配/推断；推不出来的部分，会被转化为新的目标，留待后续继续证明。这就使得 `apply` 适合“先把大目标拆成更小的目标”。

参数填写与占位符

为了准确理解 `apply` 的行为，这里补充几条规则：

- **参数按顺序提供**。使用一个定理时，参数的填写顺序与该定理陈述中的参数顺序一致。

[1]不加括号的时候默认函数应用（Application）左结合 (eg: a * b * c := (a * b) * c), 函数类型（Function Types）右结合 (eg: $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$)

- **显式参数可以暂时不填**。在 apply 中，如果某个显式参数 Lean 无法自动推断，它会变成新的证明目标。
- **隐式参数通常不需要（也不建议）直接按位置填写**。如果你确实要指定某个隐式参数，可以用“命名赋值”的方式，例如 (b := b)。
- 如果你想“跳过某个位置稍后再填”，可以用**占位符**：
  - _：让 Lean 尝试自动推断该位置；
  - ?_：在该位置生成一个新的目标，稍后再证明/填入。

下面的例子分别展示：指定隐式参数、不填显式参数让参数变成子目标、以及用 ?_ 跳过参数创建子目标。

```
#check lt_trans

theorem my_lt_trans1 {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans (b := b) -- 指定（原本可推断的）参数 (此处第一个b是lt_trans中的命名)
  apply h1
  apply h2

-- 不显式提供参数时，Lean 会尝试推断；推不出来的部分会变成新的目标
theorem my_lt_trans2 {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans
  apply h1
  apply h2

-- 需要按顺序填参；用 ?_ 跳过的位置会生成新目标
theorem my_lt_trans3 {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans ?_ h2
  apply h1
```

在实际写证明时，你可以把 apply 的使用理解为：“我想用这个引理把当前目标变成它的前提；前提如果我已经有，就直接提供；如果没有，就让 Lean 把它变成新的目标，我再继续证明。”

rw

下面介绍 rw。它是 *rewrite* 的缩写，用来**按等式（或等价）改写当前目标**。先看最基本的例子：

```
example {a b c d : ℝ} (h1 : a = c) (h2 : b = d) : a * b = c * d := by
  rw [h1]
  rw [h2]
```

rw [h1] 的直觉是：在当前目标中寻找与等式 h1 : a = c 左边 a 相同的部分，并用右边 c 替换它。因此第一步会把目标 a * b = c * d 改写成 c * b = c * d；第二步再用 h2 : b = d 把 b 改写成 d，从而完成证明。

方括号 [...] 在 rw 里的含义 这里的 [...] 表示“改写规则的列表”（可以写一个或多个，用逗号分隔），例如 rw [h1, h2] 等价于连续写两行 rw [h1]、rw [h2]。注意这与类型声明里的 [Group G] 不同：[Group G] 是类型类参数，而 rw [...] 只是 tactic 的参数列表。

改写方向：`rw [h]` 与 `rw [←h]` 默认情况下 `rw [h]` 是“把等式左边替换成右边”。如果你希望反过来用（用右边替换成左边），可以写 `rw [← h]`：

```
#check mul_add
--#check (mul_add a b c : a * (b + c) = a * b + a * c)
example (a : ℝ) : a * 0 + a * 0 = a * 0 := by
  rw [← mul_add, add_zero]
```

这里 `mul_add` 的结论形如 `a * (b + c) = a * b + a * c`。我们的目标左边出现的是 `a * 0 + a * 0`，它更像是该等式的*右边*（`a * b + a * c`）的形状，而我们希望把它改写成*左边*（`a * (b + c)`）的形状，于是使用 `← mul_add` 反向改写。随后 `add_zero` 把 `0 + 0` 化简为 `0`，等式两边就一致了。

不给参数 vs 指定参数：改写发生在哪里？ `rw` 里使用的定理（或假设）通常也带参数。你可以不显式提供参数，让 Lean 自动匹配；也可以给出参数，从而“指向”你希望改写的位置。

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

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

解释如下：

- 第一个例子里写 `rw [mul_comm]`，Lean 会在目标中寻找**第一个**能匹配乘法交换律的乘法子表达式 (这里是`a * _`)，因此它会在最外层把 `a * (b * c)` 改写成 `(b * c) * a`，也就是右边的 `b * c * a`。
- 第二个例子写 `rw [mul_comm b c]`，相当于明确告诉 Lean 你要用的是 `b * c = c * b` 这条等式，于是它会在 `(b * c)` 这个位置进行改写，把 `a * (b * c)` 改写成 `a * (c * b)`。

总之，`rw` 的使用可以概括为：用一个等式（或等价）在当前目标中做替换；默认左到右，必要时可用 `←` 反向；不给参数则自动找位置，指定参数则更精确地控制改写位置。

1.5.3 section 与 variable

`variable` 用来把一些将反复出现的变量提前声明出来，避免在每个定理陈述里重复书写。它常与 `section ... end` 搭配使用：`section` 与 `end` 划定了一个局部作用域——在该区域内声明的 `variable` 只对本段落中的定义与定理有效，不会影响 `end` 之后的内容。

```
section have

variable (a b c d e f : ℝ)

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

#check add_left_cancel
example : 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]

end have
```

这段代码里出现了两个新 tactics:

nth_rw：只改写第 n 处匹配

第一个例子里用到了 `nth_rw`，它可以看作 `rw` 的变体：当目标中有多处都能匹配某个等式时，`nth_rw n [h]` 只改写其中第 n 处匹配的位置。

在该例中，目标里出现了两次 `(a + b)`，我们只想把第二个改写成 `c`，于是写：

- `nth_rw 2 [h]`：把目标中第 2 处与 `a + b` 匹配的子表达式，用 `c` 替换；
- `rw [add_mul]`：再用分配律把 `(a + b) * c` 展开为 `a * c + b * c`。

have：先证一个引理，再回到原目标

第二个例子展示了 `have` 的典型用法。我们要证明 `a * 0 = 0`，一种常见策略是先把它转化为一个可用“加法消去”的形式：构造一个等式 `a * 0 + a * 0 = a * 0 + 0`，然后用 `add_left_cancel` 消去左边公共项得到结论。

`have` 的格式是

`have` 名称`:` 断言`:=` `by` 证明

它会**暂时把当前目标切换为该断言**；当你完成这段 `by` 证明后，`have` 产生的引理会以给定名称加入上下文，随后你回到原目标继续证明。

在例子中：

- `have h : ... := by ...` 先证明中间引理 `h`；
- 证明完成后，`h` 出现在上下文里；
- `rw [add_left_cancel h]` 使用 `add_left_cancel` 把等式两侧的 `a * 0` 消去，从而得到 `a * 0 = 0`。

如果你省略 `have` 的名称（不写 `h`），Lean 会把它默认命名为 `this`，你随后可以用 `this` 引用该引理。

1.6 exercises

1.6.1 natural number game

https://adam.math.hhu.de/#/g/leanprover-community/nng4

1.6.2 check command

看看如下`#check` 命令在 infoview 分别输出什么并理解原因:

```
section
variable (a b c : ℝ)

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

end
```

1.6.3 按定义相等

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

1.6.4 Using rw

只用 `rw` tactic 和`#check` 给出的定理完成如下习题

```
section
variable (a b c d f e: ℝ)

#check lt_trans
#check mul_assoc
#check mul_comm

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

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

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

```
#check sub_self

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

#check mul_add
#check add_mul
#check add_assoc
#check two_mul

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

#check add_zero
#check pow_two
#check mul_sub
#check add_sub
#check sub_sub

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

end
```

1.6.5 rw 结合其他

使用rw, 考虑结合 exact, nth_rw, have, apply完成如下练习

```
example (a b c d : ℝ) (hyp : c = d * a + b) (hyp' : b = a * d) : c = 2 * a * d := by
  sorry

example (a b c : ℕ) (h : a + b = c) : (a + b) * (a + b) = a * c + b * c := by
  sorry

#check add_left_cancel
example (a : ℝ) : 0 * a = 0 := by
  sorry

#check neg_add_cancel_left

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

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

#check inv_mul_cancel
#check one_mul
```

```
theorem my_mul_inv_cancel {G : Type*} [Group G] (a : G) : a * a⁻¹ = 1 := by
  sorry
```

我们会在 Day 2 引入更多和 apply 相关的练习.