Westlakelean2025

emailboxofjjd

December 2024

1 Introduction

2 前言

这份教程将为你提供一份 Lean 语言的基本介绍。Lean 语言是一门形式化语言，使用这一语言让人们可以把数学内容–包括各种定义与定理（以及定理的证明），“翻译”成为计算机语言。对我而言，学习 Lean 的过程更像是在学习一门语言而不是一项计算机技术，你需要通过了解更多“词汇”以及“语法”来把你在数学世界中所看到的一切“翻译”出来，更具体地说，我们在数学中遇到的内容，往往由一些定义以及一些定理组成，我们将会逐步学习与这些内容相对应的 Lean 语法。

我们的侧重点是 Lean 的使用，因此我们会介绍更多的 tactic，而忽略一些原理性的讲述。这份教程不需要任何的前置知识，就算你认为自己的数学水平还相对较低，也没有任何计算机基础，也可以放心读下去。希望这份教程能够为你带来一些帮助:)

3 一点点类型论

在正式开始所有内容之前，让我们回顾一些目前我们需要使用到的简单的类型论知识。目前来说，你需要记住的全部事情是：任何东西都有类型！让我们看几个简单的例子：

```
#check 1         --1 : ℕ
#check (1 : ℝ)   -- 1 : ℝ
#check √2        -- √2 : ℝ
#check 2 + 2     -- 2 + 2 : ℕ
#check 2 + 2 = 4 -- 2 + 2 = 4 : Prop
#check 2 +2 =5   -- 2 + 2 = 5 : Prop
#check mul_comm  -- mul_comm.{u_1} {G : Type u_1} [CommMagma G] (a b : G) : a * b = b * a
```

运行这段代码，你会看到 Lean 代码的操作界面分为两个部分，其中左边是代码的写作区，而右边是 infoview 的部分，代码运行的各种结果会展示在这一部分。注意到，代码中出现了第一个我们没有见过的内容`#check`，它的作用是检查任何内容的类型，如你所见，你只需要把你想要检查的内容写在`#check`后面即可，点击对应的位置，你可以在 Lean 语言的 infoview 部分看到元素的类型，在第一个例子中，你会看到这样的输出“1 : ℕ”，这代表你所检查的数字 1 的类型是自然数，Lean 中的类型总是以这样的格式展示出来，以“:”分隔，之前是内容，之后是内容的类型。第二个例子中，我们同样检查数字 1 的类型，但我们按照上述格式，将 1 声明成类型“ℝ”，这是一种强制类型转换的写法，让我们可以强制要求我们所写的内容的类型，我们得到输出“1 : ℝ”。第三个例子中，我们检查了 √2 的类型，得到了“√2 : ℝ”。第四个例子中，我们检查了一个加法表达式的类型，它的类型仍然是“ℕ”。第五个例子中，我们检查一个等式的类型，得到“2 + 2 = 4 : Prop”，意味着“2 + 2 = 4”的类型是一个命题。第六个例子中，我们检查了`mul_assoc`的类型，这是 Lean 中已经定义好的乘法交换律的名字，我们在 infoview 中看到“a * b = b * a”，这

是乘法交换律的结论。在 Lean 中，一个定理的类型是它的结论。这是一件方便的事情，因为如果你不了解一个定理的具体内容，你总是可以 #check 它来得到有关的信息。

4 第一个例子

下面我们看一个完整的 Lean 形式化的定理的例子，在我们做出解释之前，希望你能先运行并观察一下这段代码，尝试尽可能多地理解其中你能理解的部分，并且尽可能多地找到你不能理解的部分，你可以点击代码中的不同位置，观察 infoview 的内容，来获得更多信息：

```
theorem The_frist_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
```

我们逐步观察这段代码，首先是一个关键词theorem，这说明我们将要给出一个定理的陈述，接下来是the_first_example，这是我们给这个定理的命名，定理的命名可以随意修改，再之后直到a * (b * e) = c * (d * f)之前，是这个定理的所需要的参数，以及假设，在这一段内容里，同时出现了花括号，方括号和圆括号，我们会在之后解释它们分别具有什么含义，之后是“:”作为分隔，“:”后是我们希望证明的结论a * (b * e) = c * (d * f)。之后由“:= by”说明我们已经完成了定理的陈述，将开始证明这个定理。

在证明的部分，你可以点击不同的位置来观察 infoview 的结果，你会发现，在定理的证明过程中，infoview 总会给出当前所有的参数与假设，以及需要证明的目标，而随着证明的过程，你会发现目标在逐步改变，直到“No goals”代表证明完全完成。

第一个例子给出了一个完整的定理陈述与证明，现在我们已经知道，我们可以按照这样的格式来完成各种定理的陈述与证明，下面我们将逐步介绍各个部分具体应该怎样实现。

```
theorem /-name-/The_frist_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-/
  rw [h']
  rw [← mul_assoc]
  rw [h]
  exact mul_assoc c d f
```

当你把鼠标放在rw [h']后时，infoview 会展示这些内容：

```
/-
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)
-/
```

“ ”后的内容是我们需要证明的目标，“ ”之前是定理的参数部分，包含变量和假设，不难发现参数与假设的部分和我们在定理的陈述中所写的内容是一致的。

5 定理与定义的陈述

首先我们说明定理陈述中的三种括号分别具有什么意义，先看一个例子：

5.1 隐式参数与显式参数

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

注意到在这个定理的陈述中出现了花括号和圆括号，它们都给出了这个定理所需要的一些参数。区别在于，由花括号括起来的内容称为隐式参数，而由圆括号括起来的内容，称为显式参数。

为了解释这两个概念，让我们先说明 Lean 中的定理如何使用，以上面例子中的定理为例，注意到在定理的陈述中我们给出了三个参数a，b，c以及两个假设h1，h2，因此，在应用这个定理时，按照我们对于数学的理解，我们也应该需要指出这些参数与假设分别是什么，才能获得正确的定理结论。事实上，在 Lean 中，上面的定理可以被理解成这样的形式“$h1 \to h2 \to a < c$”，在填入“h1”后，则成为“$h2 \to a < c$”的形式。类比地说，一个定理是一个需要填入假设，输出结论的机器，这和我们在数学上的认知也是一致的。可以参考下面的例子：

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

#check lt_trans one_lt_two' two_lt_three' --lt_trans one_lt_two' two_lt_three' : 1 < 3
example : 1 < 3 := lt_trans one_lt_two' two_lt_three'
```

而在上面的例子my_lt_trans中，我们注意到，在使用例子中的定理时，我们没有要求填入a，b，c三个参数，这是因为在给出h1时，其中已经包含了a，b的信息（要知道a < b总得知道a，b分别是什么），同样地，h2中也包含了b，c的信息。因此，我们在陈述定理时希望这部分可以由 Lean 自动推断出来的信息不需要使用者每次都手动填入，因此，我们使用花括号括起这部分内容，来告知 Lean“这部分内容在使用本定理时不要求填入，请自动推断。”，这就是隐式参数的含义。总结一下，Lean 中的定理需要填入参数，才能输出结论。隐式参数（花括号括起的部分）是那些使用定理时不需要填入的参数，而显式参数（圆括号括起的部分）则反之。

5.2 定义的陈述

在讲解另一种方括号的使用之前，我们先介绍定义的陈述，它的写法和定理是类似的，我们需要使用到关键词def，先看一个例子：

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

def的书写格式与theorem是类似的，前三个部分都是“关键词 + 名称 + 参数”，而在theorem中需要写结论的部分，def则是我们想要这个函数返回的内容的类型，在theorem中写证明的部分，def中则是我们希望这个函数返回的内容。在例子中，我们将a声明成了一个整数，在它是偶数的情况下，希望返回它的 1/2。

5.3 structure 与 class

现在我们已经了解了关于定理陈述部分的显式参数和隐式参数，它们分别对应于圆括号和花括号，在之前的例子中还使用了一种方括号，下面我们将解释它的含义。

引入方括号的原因来自于下面的想法：在数学中，有一些结构是我们需要频繁使用的，这些结构由一些参数和假设“打包”形成，比如说：一个三维的点可以由它的三个坐标值的信息组成，线性映射的定义由它的两个假设组成

(关于加法和关于数乘可交换)。因此，Lean 中提供了关键词structure来解决这个需求。使用structure关键词，我们可以打包一些信息，例如:

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

最简单的例子是使用 structure 来给出一个三维实空间中点的定义，它由它的三个坐标值来给出，我们将三个坐标值（分别为一个实数）的信息进行了“打包”，定义了一个新的结构，我们将其命名为Point。注意 structure 的写法格式。x，y，z是我们给三个坐标值的名称（你可以尝试任意改动这三个名称），而它们的类型我们要求是实数，下一个例子是线性函数:

```
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
```

structure的写法同样允许参数，这个例子中我们给出了一个函数作为参数，它是一个实数集到实数集的映射，这个structure描述了这个映射是线性需要满足的两个条件，is_additive和preserves_mul是我们给这两个条件的名称。它们的类型分别是一个命题，分别是线性性的其中一个条件。

而在另一些情况下,我们希望使用一些特定集合上的特定数学结构,它们往往是约定俗成的,我们不希望在每次使用时都说明我们选取了怎样的结构,比如说,自然数集上的序关系,实数集上的拓扑结构,等等,这些结构可以有很多种定义的方式,但当我们每次使用时,我们总是使用特定的一种。出于这种需求,Lean 中提供了关键词class。使用class关键词，我们可以让 Lean 将一系列信息进行登记，作为一个数学结构，而对于class，Lean 中允许使用instance关键词来解释这些数学结构在特定的集合上是如何形成的。例如:

```
class PreOrder' (α : Type*) where
  le : α → α → Prop
  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
```

在例子中，我们使用class声明了一个预序结构，它要求有序关系，并且序关系具有反身性和传递性。我们使用instance关键词“登记”了自然数集上具有这个预序关系，我们可以在sorry的部分填充具体的内容我们可以使用#synth关键词来检查 Lean 是否成功“登记”了这个信息。在用instance关键词让 Lean“记住”这个信息后，我们可以在后续的内容里使用这里的内容。这些部分将会在后面的课程中展示。

现在回到我们最初的问题，当我们在定理的陈述中使用方括号时，我们事实上要求了定理中的内容具有我们以某个class声明的结构，比如在第一个例子中，我们要求G上具有群的结构。

5.4 小结

至此，我们对于 Lean 中定理的陈述部分已经较为熟悉，我们介绍了 Lean 中定理陈述的格式，“关键词 theorem+命名 + 参数 + 结论”，另外，我们介绍了三种括号的含义，其中花括号对应隐式参数，圆括号对应显式参数，而方

括号对应一个由关键词 class “打包”的数学结构。

类似的，我们可以使用def关键词，通过“关键词 def+ 命名 + 参数 + 返回值的类型 + 返回值”的格式来陈述一个定义，在定义的陈述中我们同样可以使用三种括号来写参数。

6 定理的证明

下面我们进入定理的证明部分，介绍更多的 tactic：

6.1 exact 和 rfl

我们首先介绍两个最为简单的 tactic，exact与rfl。exact的作用是使用一个定理，如果定理的结论与我们想要证明的目标完全一致，那么完成这一证明，我们来看一个例子：

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

注意“mul_assoc”，它是乘法结合律的定义，在#check中我们可以看到，它有三个显式参数需要填入，结论是乘法结合律“a * b * c = a * (b * c)”这和我们希望证明的结论是一致的，因此exact会完成这个证明。注意，exact只有在可以在定理的结论与目标完全一致时才能使用，也就是说，exact只允许我们在一步之内完成证明。

这个例子中的“example”关键词的意义和theorem是一样的，但是允许我们不提供名字。

而rfl的作用是，完成一个两侧相等的等式的证明，例如：

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

这看上去并不是很有用，我们将在后面说明rfl同样可以证明一些“看上去相等”的内容。

6.2 apply，rw 和 have

一个与exact类似的 tactic 是apply，它的作用同样是使用一个定理，首先我们看一个最简单的例子：

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

在例子中，我们通过apply使用定理并填入了参数，获得了和目标一致的结论，因此完成了证明。这和exact的使用方法是一样的。

但apply对于参数的使用更为灵活，也不要求我们在一步之内可以完成证明，这里需要额外对于定理的隐式参数与显式参数做出一些补充解释：

在使用定理时，我们需要按顺序填入定理的参数。被声明为隐式参数的内容，即使我们希望填入这部分内容，Lean 也不会接受。如果在某些情况下我们确实需要额外指明隐式参数的内容，我们可以使用“(在定理陈述中的参数名:= 参数)”的格式来填入。而反过来，在以apply使用定理时，被声明为显式参数的内容，我们也可以不填入，Lean 会做一些自动匹配，而无法自动匹配的内容，将会成为新的目标。注意，参数的填写必须按照定理陈述中的顺序，如果我们希望跳过一些参数暂时不填写，比如说，我希望跳过第一个参数，先填写第二个参数，此时，可以在第一个位置上填入“_”或“?_”作为占位符，下面的几个例子分别展示了这些内容：

```
#check lt_trans
-- 强行填入隐式参数
theorem my_lt_trans1 {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
```

```
  apply lt_trans (b := b)
  apply h1
  apply h2

-- 不填入的参数，如果无法自动推断，则成为新的目标
theorem my_lt_trans2 {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans
  apply h1
  apply h2

-- 填入参数需要按照顺序，可以使用占位符来跳过，跳过的部分会变成新的目标
theorem my_lt_trans3 {a b c : ℕ} (h1 : a < b) (h2 : b < c) : a < c := by
  apply lt_trans ?_ h2
  apply h1
```

下面是使用apply的一些练习题：

```
#check mul_assoc
example (a b c : ℕ) : a * b * c = a * (b * c) := by sorry

#check abs_le'
example (a b : ℝ) (h : a ≤ b ∧ -a ≤ b) : |a| ≤ b := by sorry

#check add_lt_add_of_lt_of_le
#check exp_lt_exp
#check le_refl
example (h₀ : a ≤ b) (h₁ : c < d) : a + exp c + e < b + exp d + e := by sorry

#check dvd_mul_of_dvd_left
#check dvd_mul_left
example (x y z : ℤ) : x ∣ y * x * z := by sorry

#check dvd_add
#check dvd_mul_of_dvd_right
#check dvd_mul_left
#check pow_two
example (x y z w : ℤ) (h : x ∣ w) : x ∣ y * (x * z) + x ^ 2 + w ^ 2 := by sorry

#check le_antisymm
#check le_min
#check min_le_right
example (a b : ℝ) : min a b = min b a := by sorry

#check le_trans
example : min (min a b) c = min a (min b c) := by sorry
```

下面我们介绍rw，同样先看一个例子：

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

```
  rw [h2]
```

rw是“rewrite”的缩写，它的格式是“rw [...]”，它的作用是依据方括号内填写的内容，来改写当前的目标，具体来说，你可以在方括号内填写一些结论是等式的定理，或者如你在例子中看到的，结论是等式的假设，rw会在当前目标中寻找与等式左边相匹配的内容，将其替换为等式右边的内容。或者，你可以填写类似等式的内容，在一些情况下rw也能完成转换，一个例子是填写一个当且仅当的命题，rw会将与当且仅当左边的内容一致的目标转换为与当且仅当右边内容一致的目标。同时，rw允许同时接受多个定理，你可以通过“rw [..., ..., ...]”的格式以逗号分隔各个定理，将多行的内容写在同一行内。

而在一些情况下，我们不能将等式左边的内容与目标匹配，而是希望将等式右边的内容与目标匹配，此时，我们可以使用“rw [← ...]”的格式，例如：

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

在例子中，我们希望将“a * 0 + a * 0”的部分改写为“a * (0 + 0)”的形式，来将两侧转化为相同的形式，注意到我们想要转化的内容和“mul_add”的结论等式右侧是相匹配的，而我们希望将它转化成“mul_add”的结论等式左侧的形式，于是我们使用“rw [← ...]”的格式，换句话说，我们想要将定理的结论“反过来”使用的时候可以使用这一格式。

而rw中使用的定理也需要填入参数，这些参数如果不填入，rw会尝试自动匹配，在可以匹配的第一个位置使用定理，而如果填入了参数，rw则会在和我们填写的参数匹配的位置进行改写，我们看下面两个例子：

```
#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]
```

第一个例子中，我们没有为定理提供参数，因此，它在最外层的乘法的位置应用了定理，而第二个例子中，我们为定理提供了参数，因此，rw在我们按照我们提供的参数进行匹配，改写了我们希望的位置。

下面是一些使用rw的练习题：

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

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

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

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

#check neg_add_cancel_left
example {a b : ℝ} (h : a + b = 0) : -a = b := by sorry
```

下面我们再介绍一些新的 tactic：

```
section a

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 a
```

在上面两个例子中，出现了一些新的关键词和两个新的 tactic，我们逐一进行说明，首先，第一个关键词 variable 的作用是将原本需要写在定理陈述中的参数提前写出来，这些参数会对 variable 后的所有函数起作用，它的常用使用方法是与 section 和 end 一起用，一个 section （名称）...end（名称）的部分会划定一个区域，在这个区域中给出的 variable 只会影响 end 之前的内容，而不会影响 end 之后的内容，variable 的具体作用细节较为复杂，我们将留到后续部分进行讲解。

在第一个例子中，出现了一个新的 tactic：“nth_rw” 它是rw的一种变体，它的作用是更改目标中与给定等式相匹配的第 n 项，例如，例子中改变了目标中的第二个 “a + b”，根据 “h” 使用 “c” 来替代它。这适用于我们只想改变目标中的特定位置时使用。

第二个例子中，出现了另一个新的 tactic：have，它的作用是暂时改变目标先证明一个引理，在例子中，我们不太方便直接证明目标 “a * 0 = 0”，所以我们使用have来先证明引理 “h”，在使用have后，我们的目标会暂时变为have所提出的内容，直到我们完成这一证明，才会回到原本的目标，而此时have提出的引理会作为一个新的参数存在，我们可以在证明时使用。它的格式是 “have 名称: 内容:= by 证明”。在使用have时，如果我们不提供名称，则have会自动将它命名为this。

6.3 与逻辑学相关的 tactic 简介

下面将展示更多的例子简单介绍一些用来处理简单逻辑学命题的 tactic。我们会在后续进一步介绍这些 tactic。在进行讲解之前，希望你能自己阅读这些例子，并且使用鼠标检查各个部分，通过阅读代码，来初步认识它们的作用，首先是逻辑学的命题出现在目标中时：

```
variable {x y : ℝ}

example (h₀ : x ≤ y) (h₁ : x ≠ y) : x ≤ y ∧ x ≠ y := by
  constructor
  · apply h₀
  · apply h₁

example (h : (y > 0) ∧ (y < 8)) : y > 0 ∨ y < -1 := by
  left
  apply h.left

example (f : ℝ → ℝ) (h : Monotone f) : ∀ {a b}, a ≤ b → f a ≤ f b := by
  intro a b hab
  apply h
```

```
  apply hab

example : ∃ x : ℝ, 2 < x ∧ x < 4 := by
  use 3
  constructor
  · norm_num
  · norm_num

example (h : x < 0) : ¬ 0 ≤ x := by
  push_neg
  apply h
```

前两个例子关于且命题和或命题，为了证明一个且命题，我们可以使用constructor将它们分成两个目标来证明。为了证明一个或命题，我们可以使用left或right来选择一边进行证明。

后两个例子是关于全称量词和存在量词，出现全称量词的命题，可以使用intro，来把“任取”部分的内容转化成参数，intro后的内容为按顺序给这些参数的命名，注意到，intro会把“→”之前的部分也转化成参数。出现存在量词的命题，可以使用use来实际给出一个例子。

最后一个例子是关于否命题，可以使用push_neg来把一个命题的否定转化为它的否命题。例如，例子中的“$\neg 0 \leq x$”会被转化为“$\mathrm{x} < 0$”。

下面使用一些例子介绍这些逻辑命题出现在条件中时的处理方法：

```
example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : x < y := by
  apply lt_of_le_of_ne
  apply h.left
  apply h.right

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : x < y := by
  rcases h with ⟨h1, h2⟩
  apply lt_of_le_of_ne
  apply h1
  apply h2

example {x : ℝ} (h : x = 3 ∨ x = 5) : x = 3 ∨ x = 5 := by
  rcases h with h | h
  · left
    apply h
  · right
    apply h

example {x : ℝ} (h : ¬ 0 ≤ x) : x < 0 := by
  push_neg at h
  exact h
```

在第一个例子中，我们使用了且命题中的两个部分，只需要使用.left与.right的写法就可以使用一个且命题的左半部分与右半部分。更为简单的写法是.1与.2，类似地，带有“ ”的命题也可以使用.1与.2来获取“→”与“←”的命题。

在第二个例子中，我们展示了另一种使用且命题的方式，使用rcases...with ⟨..., ...⟩的方法来拆分且命题。第三个命题中我们类似地展示了使用rcases...with ... ...| 的方法也可以拆分或命题，它会将目标拆分为两部分，分

别将或命题条件 $p \vee q$ 改变为 p 和 q。

最后一个例子中，我们使用了push_neg at的写法，在假设“h”处做出了转换，实际上“at”不仅可以接在push_neg后使用，它可以应用在很多的 tactic 后，对一些假设做出转换，我们学过的rw和apply，都可以衔接 “at” 来使用。

最后我们简单介绍一些能够自动完成证明的 tactic，下面是一些例子：

```
variable (a b c d : ℝ)

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

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

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

第一个是ring，它能够解决一些简单的关于加法，减法和乘法的运算问题，第二个是linarith，它能够解决简单的不等式问题。

另一个能够自动完成证明的 tactic 是simp，在 Mathlib 库中会一部分定理带有[@simp]的记号，例如：

```
@[simp] theorem Nat.dvd_one {n : ℕ} : n ∣ 1 ↔ n = 1 := ...
@[simp] theorem mul_eq_zero {a b : ℕ} : a * b = 0 ↔ a = 0 ∨ b = 0 := ...
@[simp] theorem List.mem_singleton {a b : α} : a ∈ [b] ↔ a = b := ...
@[simp] theorem Set.setOf_false : {a : α | False} = ∅ := ...
```

这些定理的结论往往是等式，或是 “ $\iff$ ” 的形式，当你在定理的证明中使用simp的时候，Lean 会自动搜索所有这些定理，并依次将这些定理的左侧与你的目标相匹配，如果能够匹配，Lean 会将目标替换为等式右侧的形式。注意，使用simp后，Lean 会无脑地尝试匹配所有定理，直到无法继续匹配为止。因此，你得到的最终形式或许不是你原本希望的。一个好的方式是使用simp only，它的格式和rw类似，你可以填写你希望使用的定理，Lean 将会只使用这部分定理进行匹配。下面是一些例子：

```
example {n : ℕ} (h : Nat.Prime n → 2 < n → ¬Even n) : n ∈ {n | Nat.Prime n} ∩ {n | n > 2} → n ∈ {n | ¬Even n} := by
  simp
  exact h

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

下面是本部分内容的综合练习：

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

7 更多的类型论

这节课我们的目标有两个，一是让大家（理论上）能够独立地完成大部分数学内容的形式化工作，二是让大家学习更多的 Lean 中的类型论知识，以便能够更清楚当我们谈论代码的时候，我们在谈论什么。

关于第一点，真正对数学进行形式化的工作当然还是比较复杂并且比较有些时候比较繁琐的，没有一定的经验积累很难说可以独立地完成相对困难的工作，因此，更精确的说，这节课的目标是教会大家足以形式化大部分数学内容的 tactic 和它们的基础使用方法，并且展示一些在形式化工作中很有用的资源，最后我们会处理一些相对复杂的题目，并在这个过程中展示形式化工作的大体流程。

而关于第二点，我们会讲解一些理论，但是仍然不会很多，理论的内容我们会在后续的课程中持续地进行补充和深入，但是（至少目前而言）对你来说更为重要的事情不是纠结于这些理论本身，而是使用它们更好地理解你阅读的代码与写作的代码，以便能够更为顺利地理解他人的代码以及写出更简洁和优雅的代码。

最后我想要说明的是，如果你一开始不能理解某一部分理论，没办法把它和实际的代码很好地对应起来，不必担心这件事，因为即使你对这些理论完全没有概念，也可以完成相当一部分的形式化工作，你有很多的时间来一边提高你的实际操作水平，一边完善自己对于理论部分的理解。

首先我们回顾一下对于 Lean 中的类型论我们的全部知识– “任何东西都有类型!”。以及我们在最开始学习到的工具：#check，它能够很方便地帮助我们检查各种东西的类型。我们用形如 “1： ” 地形式来表示一个元素的类型，在 Lean 的类型论中，我们将冒号前的内容称为元素（Term），冒号后的内容称为类型（Type）。我们已经见过一些例子：

```
#check 1
#check (1 : ℝ)
#check √2
#check 2 + 2
#check 2 + 2 = 4
#check mul_comm
```

这其中有 ， ，Prop 等多个类型，那么既然任何东西都有类型， ， ，Prop 本身的类型是什么？让我们检查一下：

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

我们得到了它们的类型– “Type”，事实上，这是 Lean 中绝大部分内容的类型，如果你继续追问：“Type” 的类型是什么呢?

```
#check Type
```