```
#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

8 term，type 和 universe

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 : ℕ”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 ℕ，ℝ，Prop 等多个类型，那么既然任何东西都有类型，ℕ，ℝ，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
```

答案是–“Type 1”！这是一件非常神奇的事情，其中的问题类似于罗素悖论–“包含全部集合的集合存在吗?”，在类型论中同样存在这样的问题，如果我们把类型的类型设置为类型，会导致悖论，我们不打算在这里解释这件事，有兴趣的同学可以自行上网搜索。但总之，人们为了解决这个问题，提出了这样的方案，如果一个东西是类型的类型，那么我们让它升级，变成“Type 1”，而聪明的同学想必已经想到了，“Type 1”的类型是“Type 2”，这些东西就这么子子孙孙无穷匮也地增长下去。这种方法将不同的内容分成了无穷的层次，我们将每个层次称为一个“宇宙（universe）”，“Type”本身处在第 0 层宇宙之中，而“Type 1”则位于第 1 层宇宙，“Type 2”则更高一层……

但“Prop”在这其中的地位是有些特殊的，因为对于任何的`p : Prop`来说，“p”本身仍然可以构成一个类型，但是对于其他位于第 0 层宇宙的内容来说，这件事是不成立的，例如“1： ”，我们完全没有理由设定“1”可以是一个

类型。让我们考虑一下这件事，“p : Prop” 对于我们来说，意义是 “p 是一个命题”，此时它可以是真命题或是假命题，例如：`2 + 2 = 4 : Prop`和`2 + 2 = 5`都是成立的。

而如果我们继续写 “h : p”，这对于我们来说意味着 “h 是 p 为真的一个证明”，按这种的方式理解更合乎 Lean 中类型论的思想。为了让叙述更简单，人们又额外称 Prop 为 “Sort 0”，而 Type 则在 “Sort 1”，按照同样的方法迭代下去。

考虑一些第 0 层宇宙的内容，比如 “ℕ”，“1 : ℕ” 对我们来说意味着 “1 是 ℕ 的一个元素”，按照上面的想法来解释 “1 是 ℕ 的一个证明” 似乎是很奇怪的，但我们可以将这理解为 “1 是 ℕ 非空的一个证明”，在 Lean 中的类型论中，我们可以统一地认为一个 Term 的意义是它对应的 Type 的证明。注意到，虽然我们在很多时候可以认为 “1:ℕ” 的含义是 “1 是 ℕ 这个集合的一个元素”，但在类型论中，类型并不等同于集合，在 Lean 中，我们额外给出了集合的定义。一个区别在于：在集合中我们可以谈论从属关系，而在类型论中则没有所谓的从属关系，在集合中，我们可以通过比较两个集合的元素来说明它们是否相等，但在类型论中，我们根本没办法谈论这件事。

8.1 定理的陈述与函数类型简介

在类型论的课程中，我们认识到，不仅任何东西都有类型，同时，我们可以使用一些方法来构造新的类型。今天将简单地介绍其中的一种–函数类型。

关于函数类型，目前我们需要知道的内容非常简单，有如下两点：

第一，如果你有任何两个类型 α 以及 β 那么你可以使用这两个类型来构造一个函数类型 $\alpha \to \beta$，同样的道理，如果你有任何三个类型，或者任意有限多的类型，你都可以用这个方式来构造一个新的类型。

第二，如果你已经构造了一个新的类型 $\alpha \to \beta$，并且你有一个以它为类型的 Term f，以及一个以 α 为类型的 Term a，那么 f a 的类型是_。同样的道理，如果你有一个三个类型构造的函数类型 $\alpha \to \beta \to \gamma$，并且你有一个以它为类型的 Term f，以及一个以 α 为类型的 Term a，那么 f a 的类型是 $\beta \to \gamma$。你同样可以对有限多个类型构造的函数类型来说明这件事。

在理解了这些时候，我们应该可以更好地理解在 Lean 中使用一个函数时发生了什么。我们已经学过，在 Lean 中，一个定理是一个需要填入它的显式参数，才能得到结论的东西。实际上，Lean 中每个定理的类型都是一个函数类型，例如下面这个定理，所展示的两种写法具有相同的含义：

```
theorem function_type {p q : Prop} (hp : p) (hq : q) : p ∧ q := by sorry


theorem function_type' : ∀ {p q : Prop}, p → q → p ∧ q := by sorry
```

它们的类型是完全一致的：$p \to q \to p \wedge q$，而这个定理相当于一个以此为类型的 Term。根据我们前面所说，如果我们有一个以 p 为类型的 Term hp，和一个以 q 为类型的 Term hq，那么`function_type hp hq`为一个以 $p \wedge q$ 为类型的 Term。在定理的陈述时，我们使用的各种括号实际上是一种 Lean 提供的替代的写法，我们事实上创造了一个以它的全部参数构成的函数类型为类型的 Term。

在了解了这件事后，我们可以重新看一下 Lean 中的证明过程具体是在做什么。

8.2 Term proof 与 Tactic proof

先说明我们的结论，证明一个定理的过程，实际上是在构造一个特定类型的 Term，这个特定的类型是定理要证明的结论。所以 Lean 中的定理证明过程实际上是一个 Term 构造的过程。“:=” 这个符号表达的含义是 “后面的内容以前面的内容作为类型”。因此，事实上我们可以像这样完成一个证明，这种证明方法称为 Term proof：

```
#check add_comm
example (a b : ℝ) : a + b = b + a := add_comm a b
```

你可以使用`#check`来检查`add\_comm a b`的类型，会发现这正是我们所需要的内容，我们曾经使用`exact`来解决这个问题，然而这样的写法现在看起来更加简洁，事实上这两种写法的含义是完全一样的。

但 Term proof 的写法在定理本身较为复杂的时候则会变得很不方便，例如：

```
example (a b c d e f : ℝ) (h₁ : b * d = f * e) (h₂ : f * c = 1) : a * b * c * d = a * e :=
    (mul_assoc (a * b) c d).symm ▸ (mul_comm d c) ▸ (mul_assoc (a * b) d c) ▸ (mul_assoc a b d).symm ▸ h₁.symm ▸
     (mul_comm e f) ▸ (mul_assoc a e f) ▸ (mul_assoc (a * e) f c).symm ▸ h₂.symm ▸ (mul_one (a * e))
```

证明中的“▶”可以被理解为rw的意思，但是这样书写的代码会有很长的一行，这会导致很难进行思考，也很难进行纠错，使用起来并不方便，于是 Lean 提供了 tactic 的写法，可以允许我们在“:=”后添加by关键词，来说明后续使用的内容我们会使用 tactic 进行写作，tactic 模式允许我们对目标进行分步转化，直到出现我们想要的形式，这是一种更为方便的写法。当然，在有些情况下，适当使用 Term proof 的模式来进行证明会使我们的代码更为简洁。下面展示了同一个例子使用 tactic 的写法，这种写法显然更为方便并且更为易懂。

一个相关的关键词是show_term，你可以在by之后加上show_term关键词，Lean 会为你展示这个证明的 Term proof 形式。

```
example (a b c d e f : ℝ) (h₁ : b * d = f * e) (h₂ : f * c = 1) : a * b * c * d = a * e := by
  have h1 : a * b * c * d = a * c * (b * d) := by ring
  have h2 : a * c * (f * e) = a * e * (f * c) := by ring
  rw [h1, h₁, h2, h₂, mul_one]
```

9 Lean 中的命名法

下面我们介绍如何为我们所形式化的定理与定义进行命名，Lean 中有规定一套规范化的命名方式，使用 `snake_case`、`lowerCamelCase` 和 `UpperCamelCase` 的组合来对任何内容进行命名。

其中，`snake_case` 命名法中，单词之间用下划线（_）分隔，所有字母通常为小写。`lowerCamelCase` 命名法中，第一个单词的首字母小写，后续单词的首字母大写，且单词之间没有分隔符。`UpperCamelCase` 命名法中，每个单词的首字母都大写，且单词之间没有分隔符。

我们根据以下命名方案使用 `snake_case`、`lowerCamelCase` 和 `UpperCamelCase` 的组合。

规则

1. **以 Prop 为类型的内容（如证明、定理名称）**：使用 `snake_case`。
2. **Prop 和 Type**：使用 `UpperCamelCase`。包括归纳类型、structure 和 class。目前有一些罕见的例外：某些 structure 的字段错误地使用了小写（见 `LT` 类的示例）。
3. **函数类型**：函数类型的命名方式与其返回值一致。例如，类型为 `A → B → C` 的函数被命名为类型 `C` 的项。
4. **其他类型项**：基本上所有其他类型项都使用 `lowerCamelCase`。
5. **在 `snake_case` 中引用 `UpperCamelCase`**：当 `UpperCamelCase` 命名的内容作为 `snake_case` 的一部分时，使用 `lowerCamelCase` 引用。
6. **缩写词**：像 `LE` 这样的缩写词作为一个整体，其大小写取决于首字符的大小写。
7. **例外情况**：一些罕见的例外情况是为了保持局部命名的对称性。例如，我们使用 `Ne` 而不是 `NE` 以遵循 `Eq` 的示例；`outParam` 的输出是 `Sort`，但它不是 `UpperCamelCase`。其他例外包括区间（如 `Set.Icc`、`Set.Iic` 等），其中 `I` 大写，尽管根据规范它应该是 `lowerCamelCase`。

下面是一些例子：

```
-- follows rule 2
structure OneHom (M : Type _) (N : Type _) [One M] [One N] where
  toFun : M → N -- follows rule 4 via rule 3 and rule 7
  map_one' : toFun 1 = 1 -- follows rule 1 via rule 7

-- follows rule 2 via rule 3
class CoeIsOneHom [One M] [One N] : Prop where
  coe_one : (↑(1 : M) : N) = 1 -- follows rule 1 via rule 6

-- follows rule 1 via rule 3
theorem map_one [OneHomClass F M N] (f : F) : f 1 = 1 := sorry

-- follows rules 1 and 5
theorem MonoidHom.toOneHom_injective [MulOneClass M] [MulOneClass N] :
  Function.Injective (MonoidHom.toOneHom : (M →* N) → OneHom M N) := sorry
-- manual align is needed due to `lowerCamelCase` with several words inside `snake_case`
#align monoid_hom.to_one_hom_injective MonoidHom.toOneHom_injective

-- follows rule 2
class HPow (α : Type u) (β : Type v) (γ : Type w) where
  hPow : α → β → γ -- follows rule 3 via rule 6; note that rule 5 does not apply

-- follows rules 2 and 6
class LT (α : Type u) where
  lt : α → α → Prop -- this is an exception to rule 2

-- follows rules 2 (for `Semifield`) and 4 (for `toIsField`)
theorem Semifield.toIsField (R : Type u) [Semifield R] :
    IsField R -- follows rule 2

-- follows rules 1 and 6
theorem gt_iff_lt [LT α] {a b : α} : a > b ↔ b < a := sorry

-- follows rule 2; `Ne` is an exception to rule 6
class NeZero : Prop := sorry

-- follows rules 1 and 5
theorem neZero_iff {R : Type _} [Zero R] {n : R} : NeZero n ↔ n ≠ 0 := sorry
-- manual align is needed due to `lowerCamelCase` with several words inside `snake_case`
#align ne_zero_iff neZero_iff
```

定理的命名问题会最常遇到，定理命名使用 snake_case，它的模式通常是“结论 +of+ 条件 1+of+ 条件 2...”，通过“of”来连接多个条件，我们已经见过一些例子。当然，一些著名的定理有它们自己的名字，比如Cauchy_Schwarz_Inequality，这些定理的命名一般会采用它们更为人所知的名字。下面展示一些常用的命名缩写：

逻辑符号

符号	代码	名称	备注
$\vee$	`\or`	or	
$\wedge$	`\and`	and	
$\to$	`\r`	of / imp	结论在前，假设通常省略
$\leftrightarrow$	`\iff`	iff	有时省略，连同等价的右侧
$\neg$	`\n`	not	
$\exists$	`\ex`	exists / bex	`bex` 表示“有界存在”
$\forall$	`\fo`	all / forall / ball	`ball` 表示“有界全称”
$=$		eq	通常省略
$\neq$	`\ne`	ne	
$\circ$	`\o`	comp	

集合符号

符号	代码	名称	备注
$\in$	`\in`	mem	
$\cup$	`\cup`	union	
$\cap$	`\cap`	inter	
$\bigcup$	`\bigcup`	iUnion / biUnion	`i` 表示“索引”，`bi` 表示“有界索引”
$\bigcap$	`\bigcap`	iInter / biInter	`i` 表示“索引”，`bi` 表示“有界索引”
$\bigcup_0$	`\bigcup\0`	sUnion	`s` 表示“集合”
$\bigcap_0$	`\bigcap\0`	sInter	`s` 表示“集合”
$\setminus$	`\\`	sdiff	
$\{x \mid p\, x\}$		setOf	
$\{x\}$		singleton	
$\{x, y\}$		pair	

代数符号

符号	代码	名称	备注
0		zero	
+		add	
−		neg / sub	neg 表示一元函数，sub 表示二元函数
1		one	
*		mul	
^		pow	
/		div	
·	\bu	smul	
$^{-1}$	\-1	inv	
$\frac{1}{\cdot}$	\frac1	invOf	
|	\\|	dvd	
∑	\sum	sum	
∏	\prod	prod	

格符号

符号	代码	名称	备注
<		lt	
≤	\le	le	
⊔	\sup	sup	二元运算符
⊓	\inf	inf	二元运算符
⨆	\sqcup	iSup / biSup / ciSup	c 表示“条件完备”
⊓	\sqcap	iInf / biInf / ciInf	c 表示“条件完备”
⊥	\bot	bot	
⊤	\top	top	

9.1 命名空间

关于 Lean 中的命名法，我们还需要解释另一个概念：命名空间。这个概念在很多编程语言中都有出现，如果你学习过某种编程语言，应该对此并不陌生。命名空间的出现是为了解决这样的问题：我们会为很多不同的东西起同样的名字。比如，在整数和自然数上，我们都想给出min_le_right这个定理。但这两个定理的含义是不同的。因此，我们希望在这种情况下，能够说明这两个定理是在不同的结构上提出的，因此我们创造了 namespace 的概念，先看一个例子：

```
namespace my_name

theorem and_of_left_of_right {p q : Prop} (hp : p) (hq : q) : p ∧ q := by sorry

#check my_name.and_of_left_of_right
```

我们使用namespace关键词来开启一个命名空间，在这个命名空间中所定义的所有定理的名称前都会带有这个命名空间的名字作为前缀，当你想要使用这个定理的时候，你需要写成命名空间名称.定理名称的形式，或者，你可以在你的文件一开始使用open关键词来打开一些命名空间，这样你可以不加前缀的使用这些命名空间中的定理。

下面是关于 Lean 的命名法的一些练习题：

```
/-
# 请为以下的example起一个合乎规范的名字：
-/
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
-/
```

10 一些好用的资源网站

1. https://leanprover-community.github.io/：这是 Lean 社区的官方网站，是一个各种 Lean 资源的集合，你可以在上面找到几乎所有你想要的资源。

2. https://leanprover-community.github.io/mathlib4_docs/：这是 Mathlib 库的官方检索网站，可以在这里搜索到 Mathlib 中的所有定理，缺点是只能使用定理名进行搜索，所以在搜索前需要先猜测出想要搜索的定理名大概是什么。

3. https://leanprover.zulipchat.com/：这是一个 Lean 社区的讨论区，你可以在里面提出各种 Lean 相关的问题，会有各种大佬帮忙解决问题或者加入讨论。新手可以在 new members 区域提问，如果是关于代码的问题的话需要提供一个可以运行的简单例子。

4. https://leansearch.net/：这是一个 lean 定理搜索的网页，你可以使用自然语言进行搜索，比官方提供的检索要好用一些。

11 suffices，assumption 和 refine

下面介绍几个新的 tactic：

```
example (P Q R : Prop) (h₁ : P → R) (h₂ : Q) (h₃ : Q → P) : R := by
  suffices h : P from h₁ h
  exact h₃ h₂
```

这个例子中使用了suffices，它的含义是“要证明目标，只需证明 h”。此时，需要说明两件事，第一是“如何通过 h 证明目标”，第二件事是“如何证明 h”，因此，suffices的使用格式是“suffices 名称（h）：只需证明的结论 from 如何通过 h 证明目标（换行）如何证明 h”，在“from”后的部分，我们需要通过suffices提出的假设“h（此处可以自己命名）”，以及原有的假设，来证明目标。在证明完目标后，我们需要通过原有的假设，来证明提出的假设“h”。注意到，suffices的用法和have的用法是非常类似的，实际上这两者完全可以交换使用，只是它们证明问题的顺序不同，have会先证明引理的结论，再通过引理来证明目标，suffices则是先证明目标，再证明引理，我们可以在不同的情况下选择用起来更顺手的一个来使用。同样的，在suffices中，如果我们不给出名称，则会被默认命名为this。

下面是一些使用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
```

```
example (a b : ℝ) (h : a = b) : a = b := by
  assumption
```

这个例子中使用了assumption，它的作用非常简单，使用assumption将会搜索假设的列表，如果找到与当前目标相匹配的内容，则完成此证明。

一个相关的 tactic 是rwa，它的用法类似于rw，但它会在使用完rw之后使用一次assumption。

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

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

这个例子中使用了refine，它的作用类似于我们介绍过的apply，事实上，它们也可以在很多情况下替换使用，区别在于，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。但我们可以在部分的内容上填充占位符。另外，refine的一个非常方便的功能如最后一个例子中所展示的，可以使用⟨...,...⟩的格式来直接证

明且命题，在尖括号内需要填入对应的两个 Term 来完成证明，或者，对于较长的证明，你也可以使用by关键词来开启 tactic 模式，这里的尖括号事实上是实现了一个构造函数，这部分内容我们将在后续进一步介绍。

在使用refine时，我们也可以使用占位符来暂时忽略那些可能较为复杂的部分，如下面的例子：

```
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)
```

下面是一些使用refine的练习题：

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

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
```

12 calc

最后我们再介绍一个非常好用的 tactic：

```
open Real

example (x y : ℝ) (h1 : 0 < x) (h2 : 0 < y) (h : log x = log y) : x = y := by
  calc
    _ = exp (log x) := by rw [exp_log h1]
    _ = exp (log y) := by rw [h]
    _ = _ := by rw [exp_log h2]
```

如例子中所演示的，“calc” 的作用是将一个难以在一步之内证明的等式分成多步来完成，其中 “_” 符号作为占位符，代表了上一个等式的结果，例如，第一行中 “_” 的位置代表 “x”，请注意它的格式和缩进。事实上，“calc” 也可以用于证明不等式的结论，使用方法是一样的，只需把证明中的等号改为不等号：

```
example (a b : ℝ) : 2 * a * b ≤ a ^ 2 + b ^ 2 := by
  have h : 0 ≤ a ^ 2 - 2 * a * b + b ^ 2 := by
    calc
      a ^ 2 - 2 * a * b + b ^ 2 = (a - b) ^ 2 := by ring
      _ ≥ 0 := pow_two_nonneg (a - b)
  calc
    2 * a * b = 2 * a * b + 0 := by ring
    _ ≤ 2 * a * b + (a ^ 2 - 2 * a * b + b ^ 2) := (add_le_add_iff_left (2 * a * b)).2 h
    _ = a ^ 2 + b ^ 2 := by ring
```

下面是一些使用 “calc” 的练习题：

```
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
```