Basics: Coq 函数式编程

SOFTWARE FOUNDATIONS Volume 1: 逻辑基础

BasicsCoq 函数式编程

引言

函数式编程风格建立在简单的、日常的数学直觉之上：若一个过程或方法没有副作用，那么在忽略效率的前提下，我们需要理解的一切便只剩下如何将输入映射到输出了 —— 也就是说，我们只需将它视作一种计算数学函数的具体方法即可。这也是 “函数式编程” 中 “函数式” 一词的含义之一。程序与简单数学对象之间这种直接的联系，同时支撑了对程序行为进行形式化证明的正确性以及非形式化论证的可靠性。

函数式编程中 “函数式” 一词的另一个含义是它强调把函数（或方法）作为一**等**的值 —— 即，这类值可以作为参数传递给其它函数，可以作为结果返回，也可以包含在数据结构中等等。这种将函数当做数据的方式，产生了大量强大而有用的编程习语（Idiom）。

其它常见的函数式语言特性包括**代数数据类型（Algebraic Data Type）**，能让构造和处理丰富数据结构更加简单的**模式匹配（Pattern Matching）**，以及用来支持抽象和代码复用的复杂的**多态类型系统（Polymorphic Type System）**。Coq 提供了所有这些特性。

本章的前半部分介绍了 Coq 的函数式编程语言 **Gallina** 中最基本的元素，后半部分则介绍了一些基本**策略（Tactic）**，它可用于证明 Coq 程序的简单属性。

数据与函数

枚举类型

Coq 的一个不同寻常之处在于它内建了**极小**的特性集合。比如，Coq 并未提供通常的原子数据类型（如布尔、整数、字符串等），而是提供了一种极为强大的，可以从头定义新的数据类型的机制 —— 强大到所有常见的类型都是由它定义而产生的实例。

当然，Coq 发行版同时也提供了内容丰富的标准库，其中定义了布尔值、数值，以及如列表、散列表等许多通用的数据结构。不过这些库中的定义并没有任何神秘之处，也没有原语（Primitive）中独有的地方。为了说明这一点，我们并未直接使用库中的数据类型，而是在整个教程中重新定义了它们。

一周七日

让我们从一个非常简单的例子开始，看看这种定义机制是如何工作的。以下声明会告诉 Coq 我们定义了一个新的数据集合，即一个**类型（Type）**。

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

该类型名为 day，成员包括 monday、tuesday 等等。

定义了 day 之后, 我们就能写一些操作星期的函数了。

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday    tuesday
  | tuesday    wednesday
  | wednesday    thursday
  | thursday    friday
  | friday    monday
  | saturday    monday
  | sunday    monday
  end.
```

注意，这里显式声明了函数的参数和返回类型。和大多数函数式编程语言一样，如果没有显式指定类型，Coq 通常会自己通过 **类型推断（Type Inference）**得出。不过我们会标上类型使其更加易读。

定义了函数之后，我们要用一些例子来检验它。实际上，在 Coq 中，检验的方式一共有三种：第一，我们可以用 Compute 指令来计算包含 next_weekday 的复合表达式：

```
Compute (next_weekday friday).
(* ==> monday : day *)

Compute (next_weekday (next_weekday saturday)).
(* ==> tuesday : day *)
```

（我们在注释中写出 Coq 返回的结果。如果你身边就有电脑，不妨自己用 Coq 解释器试一试：选一个你喜欢的 IDE，CoqIde 或 Proof General 都可以。然后从本书附带的 Coq 源码中载入 Basics.v 文件，找到上面的例子，提交给 Coq，然后查看结果。）

第二，我们可以将**期望**的结果写成 Coq 的示例：

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
```

该声明做了两件事：首先它断言 saturday 之后的第二个工作日是 tuesday；然后它为该断言取了名字以便之后引用它。定义好断言后，我们还可以让 Coq 来验证它，就像这样：

```
Proof. simpl. reflexivity. Qed.
```

具体细节并不重要（之后还会解释），不过这段代码基本上可以读作 “经过一番化简后，若等式两边的求值结果相同，该断言即可得证。”

第三，我们可以让 Coq 从 Definition 中**提取（Extract）**出用其它编程语言编写的程序（如 OCaml、Scheme、Haskell 等），它们拥有高性能的编译器。这种能力非常有用，我们可以通过它将 Gallina 编写的 **证明正确** 的算法转译成高效的机器码。（诚然，我们必须信任 OCaml/Haskell/Scheme 的编译器，以及 Coq 提取工具自身的正确性，然而比起现在大多数软件的开发方式，这也是很大的进步了。）实际上，这就是 Coq 最主要的使用方式之一。在之后的章节中我们会回到这一主题上来。

作业提交指南

如果你在课堂中使用《软件基础》，你的讲师可能会用自动化脚本来为你的作业评分。为了让这些脚本能够正常工作（这样你才能拿到全部学分！），请认真遵循以下规则：

- 评分脚本在提取你提交的.v 文件时会用到其中的特殊标记。因此请勿修改练习的 “分隔标记”，如练习的标题、名称、以及末尾的 [] 等等。
- 不要删除练习。如果你想要跳过某个练习（例如它标记为可选或你无法解决它），可以在.v 文件中留下部分证明，这没关系，不过此时请确认它以 Admitted 结尾（不要用 Abort 之类的东西）。
- 你也可以在解答中使用附加定义（如辅助函数，需要的引理等）。你可以将它们放在练习的头部和你要证明的定理之间。

You will also notice that each chapter (like Basics.v) is accompanied by a _test script_ (BasicsTest.v) that automatically calculates points for the finished homework problems in the chapter. These scripts are mostly for the auto-grading infrastructure that your instructor may use to help process assignments, but you may also like to use them to double-check that your file is well formatted before handing it in. In a terminal window either type make BasicsTest.vo or do the following:

```
coqc -Q . LF Basics.v
coqc -Q . LF BasicsTest.v
```

There is no need to hand in BasicsTest.v itself (or Preface.v).

If your class is using the Canvas system to hand in assignments:

- If you submit multiple versions of the assignment, you may notice that they are given different names. This is fine: The most recent submission is the one that will be graded.
- To hand in multiple files at the same time (if more than one chapter is assigned in the same week), you need to make a single submission with all the files at once using the button "Add another file" just above the comment box.

布尔值

通过类似的方式，我们可以为布尔值定义标准类型 bool，它包括 true 和 false 两个成员。

```
Inductive bool : Type :=
  | true
  | false.
```

当然，Coq 的标准库中提供了布尔类型的默认实现，以及大量有用的函数和引理。(有兴趣的话可参考 Coq 库文档中的 Coq.Init.Datatypes。) 我们会尽量让自己的定义和定理的名字与标准库保持一致。

布尔值的函数可按照同样的方式来定义：

```
Definition negb (b:bool) : bool :=
  match b with
  | true  false
  | false  true
  end.


Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true  b2
  | false  false
  end.
```

```
Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true   true
  | false   b2
  end.
```

其中后两段演示了定义多参数函数的语法。以下四个 “单元测试” 则演示了多参数应用的语法，它们构成了 orb 函数的完整规范（Specification），即真值表：

```
Example test_orb1: (orb true false) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb2: (orb false false) = false.
Proof. simpl. reflexivity. Qed.
Example test_orb3: (orb false true) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb4: (orb true true) = true.
Proof. simpl. reflexivity. Qed.
```

我们也可以为刚定义的布尔运算引入更加熟悉的语法。Notation 指令能为既有的定义赋予新的中缀记法。

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).
```

```
Example test_orb5: false || false || true = true.
Proof. simpl. reflexivity. Qed.
```

’ 关于记法的说明：在.v 文件中，我们用方括号来界定注释中的 Coq 代码片段；这种约定也在 coqdoc 文档工具中使用，它能让代码与周围的文本从视觉上区分开来。在 HTML 版的文件中，这部分文本会以**不同的字体**显示。

特殊的指令 Admitted 被用作不完整证明的占位符。我们会在练习中用它来表示留给你的部分。你的练习作业就是将 Admitted 替换为具体的证明。

练习：1 星, standard (nandb)

移除 “Admitted.” 并补完以下函数的定义，然后确保下列每一个 Example 中的断言都能被 Coq 验证通过。（即仿照上文 orb 测试的模式补充证明。）此函数应在两个输入中包含 false 时返回 true 。

```
Definition nandb (b1:bool) (b2:bool) : bool
  (* 将本行替换成 ":= _ 你的 _ 定义 _ ." *). Admitted.
```

Example test_nandb1: (nandb true false) = true.
(* 请在此处解答 *) Admitted.
Example test_nandb2: (nandb false false) = true.
(* 请在此处解答 *) Admitted.
Example test_nandb3: (nandb false true) = true.
(* 请在此处解答 *) Admitted.
Example test_nandb4: (nandb true true) = false.
(* 请在此处解答 *) Admitted.

练习：1 星, standard (andb3)

与此前相同，完成下面的 andb3 函数。此函数应在所有输入均为 true 时返回 true，否则返回 false。

Definition andb3 (b_1:bool) (b_2:bool) (b_3:bool) : bool
  (* 将本行替换成 ":= _ 你的 _ 定义 _ ." *). Admitted.

Example test_andb31: (andb3 true true true) = true.
(* 请在此处解答 *) Admitted.
Example test_andb32: (andb3 false true true) = false.
(* 请在此处解答 *) Admitted.
Example test_andb33: (andb3 true false true) = false.
(* 请在此处解答 *) Admitted.
Example test_andb34: (andb3 true true false) = false.
(* 请在此处解答 *) Admitted.

类型

Coq 中的每个表达式都有类型，它描述了该表达式所计算的东西的类别。Check 指令会让 Coq 显示一个表达式的类型。

Check true.
(* ===> true : bool *)
Check (negb true).
(* ===> negb true : bool *)

像 negb 这样的函数本身也是数据值，就像 true 和 false 一样。它们的类型被称为**函数类型**，用带箭头的类型表示。

```
Check negb.
(* ===> negb : bool -> bool *)
```

negb 的类型写作 bool → bool，读做 “bool 箭头 bool”，可以理解为 “给定一个 bool 类型的输入，该函数产生一个 bool 类型的输出。” 同样，andb 的类型写作 bool → bool → bool，可以理解为 “给定两个 bool 类型的输入，该函数产生一个 bool 类型的输出。”

由旧类型构造新类型

到目前为止，我们定义的类型都是 “枚举类型”：它们的定义显式地枚举了一个元素的有限集，其中每个元素都只是一个裸构造子（译注：即无参数构造子）。下面是一个更加有趣的类型定义，其中有个构造子接受一个参数：

```
Inductive rgb : Type :=
  | red
  | green
  | blue.
```

```
Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

我们来仔细研究一下。

每个归纳定义的类型（如 day、bool、rgb、color 等）包含一个由构造子（如 red、primary、true、false、monday 等）构建的 **构造子表达式**的集合。rgb 和 color 的定义描述了如何构造这两个集合中的元素（即表达式）：

- red、green 和 blue 是 rgb 的构造子；
- black、white 和 primary 是 color 的构造子；
- 表达式 red 属于集合 rgb，表达式 green 与 blue 亦同；
- 表达式 black 和 white 属于集合 color；
- 若 p 为属于集合 rgb 的表达式，则 primary p（读作 “构造子 primary 应用于参数 p”）是属于集合 color 的表达式；以及
- **只有**按照这些方式构造的表达式才属于集合 rgb 和 color。

我们可以像之前的 day 和 bool 那样用模式匹配为 color 定义函数。

```
Definition monochrome (c : color) : bool :=
  match c with
  | black   true
  | white   true
  | primary q   false
  end.
```

鉴于 primary 构造子接收一个参数，匹配到 primary 的模式应当带有一个变量或常量。变量可以取任意名称，如上文所示；常量需有适当的类型，例如：

```
Definition isred (c : color) : bool :=
  match c with
  | black   false
  | white   false
  | primary red   true
  | primary _   false
  end.
```

这里的模式 primary _ 是 “primary 应用到除 red 之外的任何 rgb 构造子” 的简写形式（通配模式 _ 的效果与 monochrome 定义中的哑（dummy）模式变量 p 相同。）

Tuples

A single constructor with multiple parameters can be used to create a tuple type. As an example, consider representing the four bits in a nybble (half a byte). We first define a datatype bit that resembles bool (using the constructors B_0 and B_1 for the two possible bit values), and then define the datatype nybble, which is essentially a tuple of four bits.

```
Inductive bit : Type :=
  | B0
  | B1.
```

```
Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).
```

```
Check (bits B1 B0 B1 B0).
(* ==> bits B1 B0 B1 B0 : nybble *)
```

The bits constructor acts as a wrapper for its contents. Unwrapping can be done by pattern-matching, as in the all_zero function which tests a nybble to see if all its bits are O. Note that we are using underscore (_) as a _wildcard pattern_ to avoid inventing variable names that will not be used.

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
    | (bits B0 B0 B0 B0)   true
    | (bits _ _ _ _)   false
  end.
```

```
Compute (all_zero (bits B1 B0 B1 B0)).
(* ===> false : bool *)
Compute (all_zero (bits B0 B0 B0 B0)).
(* ===> true : bool *)
```

模块

Coq 提供了**模块系统**来帮助组织大规模的开发。在本课程中，我们不太会用到这方面的特性。不过其中有一点非常有用：如果我们将一组定义放在 Module X 和 End X 标记之间，那么在文件中的 End 之后，我们就可以通过像 X.foo 这样的名字来引用，而不必直接用 foo 了。在这里，我们通过此特性在内部模块中引入了 nat 类型的定义，这样就不会覆盖标准库中的同名定义了，毕竟它用了点儿简便的特殊记法。

```
Module NatPlayground.
```

数值

The types we have defined so far, ”enumerated types” such as day, bool, and bit, and tuple types such as nybble built from them, share the property that each type has a finite set of values. The natural numbers are an infinite set, and we need to represent all of them in a datatype with a finite number of constructors. There are many representations of numbers to choose from. We are most familiar with decimal notation (base 10), using the digits 0 through 9, for example, to form the number 123. You may have encountered hexadecimal notation (base 16), in which the same number is represented as 7B, or octal (base 8), where it is 173, or binary (base 2), where it is 1111011. Using an enumerated type to represent digits, we could use any of these to represent natural numbers. There are circumstances where each of these choices can be useful.

Binary is valuable in computer hardware because it can in turn be represented with two voltage levels, resulting in simple circuitry. Analogously, we wish here to choose a representation that makes _proofs_ simpler.

Indeed, there is a representation of numbers that is even simpler than binary, namely unary (base 1), in which only a single digit is used (as one might do while counting days in prison by scratching on the walls). To represent unary with a Coq datatype, we use two constructors. The capital-letter O constructor represents zero. When the S constructor is applied to the representation of the natural number _n_, the result is the representation of _n+1_. (S stands for "successor", or "scratch" if one is in prison.) Here is the complete datatype definition.

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

With this definition, 0 is represented by O, 1 by S O, 2 by S (S O), and so on.

此定义中的从句可读作：

- O 是一个自然数（注意这里是字母 “O”，不是数字 “0”）。
- S 可被放在一个自然数之前产生另一个自然数 —— 也就是说，如果 n 是一个自然数，那么 S n 也是。

同样，我们来仔细观察这个定义。nat 的定义描述了集合 nat 中的表达式是如何构造的：

- O 和 S 是构造子；
- 表达式 O 属于集合 nat；
- 如果 n 是属于集合 nat 的表达式，那么 S n 也是属于集合 nat 的表达式；并且
- 只有按照这两种方式构造的表达式才属于集合 nat。

同样的规则也适用于 day、bool、color 等的定义。

以上条件是精确构成 Inductive 声明的主要推动力。它们蕴含的表达式 O、S O、S (S O)、S (S (S O)) 等等都属于集合 nat，而像 true、andb true false、S (S false) 以及 O (O (O S)) 之类的，由数据构造子构造的表达式则不属于 nat。

关键之处在于，我们目前只是定义了一种数字的**表示**方式，即一种写下它们的方式。名称 O 和 S 是任意的，在这一点上它们没有特殊的意义，它们只是我们能用来写下数字的两个不同的记号（以及一个说明了任何 nat 都能写成一串 S 后跟一个 O 的规则）。如果你喜欢，完全可以将同样的定义写成：

```
Inductive nat' : Type :=
  | stop
  | tick (foo : nat').
```

这些记号的**解释**完全取决于我们如何用它进行计算。

我们可以像之前的布尔值或日期那样，编写一个函数来对上述自然数的表示进行模式匹配。例如，以下为前趋函数：

```
Definition pred (n : nat) : nat :=
  match n with
```

```
  | O    O
  | S n'    n'
end.
```

第二个分支可以读作：“如果 n 对于某个 n' 的形式为 S n'，那么就返回 n'。”

```
End NatPlayground.
```

为了让自然数使用起来更加自然，Coq 内建了一小部分解析打印功能：普通的十进制数可视为 “一进制” 自然数的另一种记法，以代替 S 与 O 构造子；反过来，Coq 也会默认将自然数打印为十进制形式：

```
Check (S (S (S (S O)))).
  (* ===> 4 : nat *)
```

```
Definition minustwo (n : nat) : nat :=
  match n with
    | O    O
    | S O    O
    | S (S n')    n'
  end.
```

```
Compute (minustwo 4).
  (* ===> 2 : nat *)
```

构造子 S 的类型为 nat → nat，与函数 pred 和 minustwo 相同：

```
Check S.
Check pred.
Check minustwo.
```

以上三个函数均可作用于自然数，并产生自然数结果，但第一个 S 与后两者有本质区别：pred 和 minustwo 这类函数定义了 **计算规则**—— 例如 pred 的定义表明 pred 2 可化简为 1——但 S 的定义不表征此类行为。虽然 S 可以作用于参数这点与函数相仿，但其作用仅限于构造数字。（考虑标准的十进制数：数字 1 不代表任何计算，只表示一部分数据。用 111 指代数字一百一十一，实则使用三个 1 符号表示此数各位。）

模式匹配不足以描述很多数字运算，我们还需要递归定义。例如：给定自然数 n，欲判定其是否为偶数，则需递归检查 n-2 是否为偶数。关键字 Fixpoint 可用于定义此类函数。

```
Fixpoint evenb (n:nat) : bool :=
```

```
  match n with
  | O    true
  | S O    false
  | S (S n')    evenb n'
  end.
```

我们可以使用类似的 Fixpoint 声明来定义 odd 函数，不过还有种更简单的定义：

```
Definition oddb (n:nat) : bool := negb (evenb n).
```

```
Example test_oddb1: oddb 1 = true.
Proof. simpl. reflexivity. Qed.
Example test_oddb2: oddb 4 = false.
Proof. simpl. reflexivity. Qed.
```

（如果你逐步检查完这些证明，就会发现 simpl 其实没什么作用 —— 所有工作都被 reflexivity 完成了。我们不久就会看到为什么会这样。）

当然，我们也可以用递归定义多参函数。

```
Module NatPlayground2.
```

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O    m
    | S n'    S (plus n' m)
  end.
```

三加二等于五，正如所料。

```
Compute (plus 3 2).
```

为得出此结论，Coq 所执行的化简步骤如下：

```
(*  plus (S (S (S O))) (S (S O))
==> S (plus (S (S O)) (S (S O))) 根据第二个 match 子句
==> S (S (plus (S O) (S (S O)))) 根据第二个 match 子句
==> S (S (S (plus O (S (S O))))) 根据第二个 match 子句
```

==> S (S (S (S O)))) 根据第一个 match 子句 *)

为了书写方便，如果两个或更多参数具有相同的类型，那么它们可以写在一起。在下面的定义中，(n m : nat) 的意思与 (n : nat) (m : nat) 相同。

```
Fixpoint mult (n m : nat) : nat :=
  match n with
    | O   O
    | S n'   plus m (mult n' m)
  end.
```

```
Example test_mult1: (mult 3 3) = 9.
Proof. simpl. reflexivity. Qed.
```

你可以在两个表达式之间添加逗号来同时匹配它们：

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O , _   O
  | S _ , O   n
  | S n', S m'   minus n' m'
  end.
```

```
End NatPlayground2.
```

```
Fixpoint exp (base power : nat) : nat :=
  match power with
    | O   S O
    | S p   mult base (exp base p)
  end.
```

练习：1 星, standard (factorial)

回想一下标准的阶乘函数：

```
       factorial(0)  =  1
```

```
factorial(n)  =  n * factorial(n-1)     (if n>0)
```

把它翻译成 Coq 语言。

Fixpoint factorial (n:nat) : nat
  (* 将本行替换成 ":= _ 你的 _ 定义 _ ." *). Admitted.

Example test_factorial1: (factorial 3) = 6.
(* 请在此处解答 *) Admitted.
Example test_factorial2: (factorial 5) = (mult 10 12).
(* 请在此处解答 *) Admitted.

我们可以通过引入加法、乘法和减法的**记法（Notation）**来让数字表达式更加易读。

Notation "x + y" := (plus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x - y" := (minus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x * y" := (mult x y)
                    (at level 40, left associativity)
                    : nat_scope.

Check ((0 + 1) + 1).

(level、associativity 和 nat_scope 标记控制着 Coq 语法分析器如何处理上述记法。本课程不关注其细节。有兴趣的读者可参阅本章末尾 “关于记法的更多内容” 一节。)

注意，它们并不会改变我们之前的定义，而只是让 Coq 语法分析器接受用 x + y 来代替 plus x y，并在 Coq 美化输出时反过来将 plus x y 显示为 x + y。

Coq 不包含任何内置定义，以至于数值间相等关系的测试也是由用户来实现。

eqb 函数定义如下：该函数测试自然数 nat 间相等关系 eq，并以布尔值 bool 表示。注意该定义使用嵌套匹配 match （亦可仿照 minus 使用并列匹配）。

Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O   match m with

```
      | O   true
      | S m'   false
      end
  | S n'   match m with
        | O   false
        | S m'   eqb n' m'
        end
  end.
```

类似地，leb 函数检查其第一个参数是否小于等于第二个参数，以布尔值表示。

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O   true
  | S n'
    match m with
    | O   false
    | S m'   leb n' m'
    end
  end.
```

```
Example test_leb1: (leb 2 2) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb2: (leb 2 4) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb3: (leb 4 2) = false.
Proof. simpl. reflexivity. Qed.
```

Since we'll be using these (especially eqb) a lot, let's give them infix notations.

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
Notation "x <=? y" := (leb x y) (at level 70) : nat_scope.
```

```
Example test_leb3': (4 <=? 2) = false.
Proof. simpl. reflexivity. Qed.
```

练习：1 星, standard (ltb)

ltb 函数检查自然数间的小于关系，以布尔值表示。利用前文定义的函数写出该定义，不要使用 Fixpoint 构造新的递归。（只需前文中的一个函数即可实现定义，但亦可两者皆用。）

```
Definition ltb (n m : nat) : bool
  (* 将本行替换成 ":= _ 你的 _ 定义 _ ." *). Admitted.
```

```
Notation "x <? y" := (ltb x y) (at level 70) : nat_scope.
```

```
Example test_ltb1: (ltb 2 2) = false.
(* 请在此处解答 *) Admitted.
Example test_ltb2: (ltb 2 4) = true.
(* 请在此处解答 *) Admitted.
Example test_ltb3: (ltb 4 2) = false.
(* 请在此处解答 *) Admitted.
```

基于化简的证明

至此，我们已经定义了一些数据类型和函数。让我们把问题转到如何表述和证明它们行为的性质上来。其实我们已经开始这样做了：前几节中的每个 Example 都对几个函数在某些特定输入上的行为做出了准确的断言。这些断言的证明方法都一样：使用 simpl 来化简等式两边，然后用 reflexivity 来检查两边是否具有相同的值。

这类 “基于化简的证明” 还可以用来证明更多有趣的性质。例如，对于 “0 出现在左边时是加法 + 的 ‘幺元”’ 这一事实，我们只需读一遍 plus 的定义，即可通过观察 “对于 0 + n，无论 n 的值为多少均可化简为 n” 而得到证明。

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity. Qed.
```

（如果你同时浏览.v 文件和 HTML 文件，那么大概会注意到以上语句在你的 IDE 中和在浏览器渲染的 HTML 中不大一样，我们用保留标识符 “forall” 来表示全称量词 ∀。当.v 文件转换为 HTML 后，它会变成一个倒立的 “A”。）

现在是时候说一下 reflexivity 了，它其实比我们想象得更为强大。在前面的例子中，其实并不需要调用 simpl ，因为 reflexivity 在检查等式两边是否相等时会自动做一些化简；我们加上 simpl 只是为了看到化简之后，证明结束之前的中间状态。下面是对同一定理更短的证明：

```
Theorem plus_O_n' : n : nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.
```

此外，reflexivity 在某些方面做了比 simpl **更多**的化简 —— 比如它会尝试 “展开” 已定义的项，将它们替换为该定义右侧的值。了解这一点会对以后很有帮助。产生这种差别的原因是，当自反性成立时，整个证明目标就完成了，我们不必再关心 reflexivity 化简和展开了什么；而当我们必须去观察和理解新产生的证明目标时，我们并不希望它盲目地展开定义，将证明目标留在混乱的声明中。这种情况下就要用到 simpl 了。

我们刚刚声明的定理形式及其证明与前面的例子基本相同，它们只有一点差别。

首先，我们使用了关键字 Theorem 而非 Example。这种差别纯粹是风格问题；在 Coq 中，关键字 Example 和 Theorem（以及其它一些，包括 Lemma、Fact 和 Remark）都表示完全一样的东西。

其次，我们增加了量词 n:nat，因此我们的定理讨论了**所有的**自然数 n。在非形式化的证明中，为了证明这种形式的定理，我们通常会说 “**假设**存在一个任意自然数 n...”。而在形式化证明中，这是用 intros n 来实现的，它会将量词从证明目标转移到当前假设的**上下文**中。

关键字 intros、simpl 和 reflexivity 都是 **策略（Tactic）**的例子。策略是一条可以用在 Proof（证明）和 Qed（证毕）之间的指令，它告诉 Coq 如何来检验我们所下的一些断言的正确性。在本章剩余的部分及以后的课程中，我们会见到更多的策略。

其它类似的定理可通过相同的模式证明。

```
Theorem plus_1_l : n:nat, 1 + n = S n.
Proof.
  intros n. reflexivity. Qed.
```

```
Theorem mult_0_l : n:nat, 0 * n = 0.
Proof.
  intros n. reflexivity. Qed.
```

上述定理名称的后缀 _l 读作 “在左边”。

跟进这些证明的每个步骤，观察上下文及证明目标的变化是非常值得的。你可能要在 reflexivity 前面加上 simpl 调用，以便观察 Coq 在检查它们的相等关系前进行的化简。

基于改写的证明

下面这个定理比我们之前见过的更加有趣：

```
Theorem plus_id_example : n m:nat,
  n = m →
  n + n = m + m.
```

该定理并未对自然数 n 和 m 所有可能的值做全称断言，而是讨论了仅当 n = m 时这一更加特定情况。箭头符号读作 “蕴含”。

与此前相同，我们需要在能够假定存在自然数 n 和 m 的基础上进行推理。另外我们需要假定有前提 n = m。intros 策略用来将这三条前提从证明目标移到当前上下文的假设中。

由于 n 和 m 是任意自然数，我们无法用化简来证明此定理，不过可以通过观察来证明它。如果我们假设了 n = m，那么就可以将证明目标中的 n 替换成 m 从而获得两边表达式相同的等式。用来告诉 Coq 执行这种替换的策略叫做**改写** rewrite。

```
Proof.
  (* 将两个量词移到上下文中： *)
  intros n m.
  (* 将前提移到上下文中： *)
  intros H.
  (* 用前提改写目标： *)
  rewrite → H.
  reflexivity. Qed.
```

证明的第一行将全称量词变量 n 和 m 移到上下文中。第二行将前提 n = m 移到上下文中，并将其命名为 H。第三行告诉 Coq 改写当前目标（n + n = m + m），把前提等式 H 的左边替换成右边。

(rewrite 中的箭头与蕴含无关：它指示 Coq 从左往右地应用改写。若要从右往左改写，可以使用 rewrite <-。在上面的证明中试一试这种改变，看看 Coq 的反应有何不同。)

练习：1 星, standard (plus_id_exercise)

删除”Admitted.” 并补完证明。

```
Theorem plus_id_exercise : n m o : nat,
  n = m → m = o → n + m = m + o.
Proof.
  (* 请在此处解答 *) Admitted.
```

Admitted 指令告诉 Coq 我们想要跳过此定理的证明，而将其作为已知条件，这在开发较长的证明时很有用。在进行一些较大的命题论证时，我们能够声明一些附加的事实。既然我们认为这些事实对论证是有用的，就可以用 Admitted 先不加怀疑地接受这些事实，然后继续思考大命题的论证。直到确认了该命题确

实是有意义的，再回过头去证明刚才跳过的证明。但是要小心：每次使用 Admitted 或者 admit，你就为 Coq 这个完好、严密、形式化且封闭的世界开了一个毫无道理的后门。

可用的不只有上下文中现有的前提，我们还可以通过 rewrite 策略来运用前期证明过的定理。如果前期证明的定理的语句中包含量词变量，如前例所示，Coq 会通过匹配当前的证明目标来尝试实例化（Instantiate）它们。

```
Theorem mult_0_plus : n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  rewrite → plus_O_n.
  reflexivity. Qed.
```

练习：2 星, standard (mult_S_1)

```
Theorem mult_S_1 : n m : nat,
  m = S n →
  m * (1 + n) = m * m.
Proof.
  (* 请在此处解答 *) Admitted.
```

(* （注意，该命题可用 rewrite 以外的策略证明，不过请使用 rewrite 来做练习。） *)

利用情况分析来证明

当然，并非一切都能通过简单的计算和改写来证明。通常，一些未知的，假定的值（如任意数值、布尔值、列表等等）会阻碍化简。例如，如果我们像以前一样使用 simpl 策略尝试证明下面的事实，就会被卡住。（现在我们用 Abort 指令来放弃证明。）

```
Theorem plus_1_neq_0_firsttry : n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n.
  simpl. (* 无能为力! *)
Abort.
```

原因在于：根据 eqb 和 + 的定义，其第一个参数先被 match 匹配。但此处 + 的第一个参数 n 未知，而 eqb 的第一个参数 n + 1 是复杂表达式，二者均无法化简。

欲进行规约，则需分情况讨论 n 的所有可能构造。如果 n 为 O，则可验算 (n + 1) =? 0 的结果确实为 false；如果 n 由 S n' 构造，那么即使我们不知道 n + 1 的确切结果，但至少知道它的构造子为 S，因而足以得出 (n + 1) =? 0 的结果为 false。

告诉 Coq 分别对 n = 0 和 n = S n' 这两种情况进行分析的策略，叫做 destruct。

```
Theorem plus_1_neq_0 : n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  - reflexivity.
  - reflexivity. Qed.
```

The destruct generates _two_ subgoals, which we must then prove, separately, in order to get Coq to accept the theorem.

The annotation "as [| n']" is called an _intro pattern_. It tells Coq what variable names to introduce in each subgoal. In general, what goes between the square brackets is a _list of lists_ of names, separated by |. In this case, the first component is empty, since the O constructor is nullary (it doesn't have any arguments). The second component gives a single name, n', since S is a unary constructor.

In each subgoal, Coq remembers the assumption about n that is relevant for this subgoal — either n = 0 or n = S n' for some n'. The eqn:E annotation tells destruct to give the name E to this equation. (Leaving off the eqn:E annotation causes Coq to elide these assumptions in the subgoals. This slightly streamlines proofs where the assumptions are not explicitly used, but it is better practice to keep them for the sake of documentation, as they can help keep you oriented when working with the subgoals.)

第二行和第三行中的 - 符号叫做**标号**，它标明了每个生成的子目标所对应的证明部分。（译注：此处的 "标号" 应理解为一个项目列表中每个 **条目**前的小标记，如 或 •。）标号后面的代码是一个子目标的完整证明。在本例中，每个子目标都简单地使用 reflexivity 完成了证明。通常，reflexivity 本身会执行一些化简操作。例如，第二段证明将 at (S n' + 1) 0 化简成 false，是通过先将 (S n' + 1) 转写成 S (n' + 1)，接着展开 beq_nat，之后再化简 match 完成的。

用标号来区分情况完全是可选的：如果没有标号，Coq 只会简单地要求你依次证明每个子目标。尽管如此，使用标号仍然是一个好习惯。原因有二：首先，它能让证明的结构更加清晰易读。其次，标号能指示 Coq 在开始验证下一个目标前确认上一个子目标已完成，防止不同子目标的证明搅和在一起。这一点在大型开发中尤为重要，因为一些证明片段会导致很耗时的排错过程。

在 Coq 中并没有既严格又便捷的规则来格式化证明 —— 尤其指应在哪里断行，以及证明中的段落应如何缩进以显示其嵌套结构。然而，无论格式的其它方面如何布局，只要在多个子目标生成的地方为每行开头标上标号，那么整个证明就会有很好的可读性。

这里也有必要提一下关于每行代码长度的建议。Coq 的初学者有时爱走极端，要么一行只有一个策略语句，要么把整个证明都写在一行里。更好的风格则介于两者之间。一个合理的习惯是给自己设定一个每行

80 个字符的限制。更长的行会很难读，也不便于显示或打印。很多编辑器都能帮你做到。

destruct 策略可用于任何归纳定义的数据类型。比如，我们接下来会用它来证明布尔值的取反是对合（Involutive）的 —— 即，取反是自身的逆运算。

```
Theorem negb_involutive : b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b eqn:E.
  - reflexivity.
  - reflexivity. Qed.
```

注意这里的 destruct 没有 as 子句，因为此处 destruct 生成的子分类均无需绑定任何变量，因此也就不必指定名字。（当然，我们也可以写上 as [|] 或者 as []。）实际上，我们也可以省略 **任何** destruct 中的 as 子句，Coq 会自动填上变量名。不过这通常是个坏习惯，因为如果任其自由决定的话，Coq 经常会选择一些容易令人混淆的名字。

有时在一个子目标内调用 destruct，产生出更多的证明义务（Proof Obligation）也非常有用。这时候，我们使用不同的标号来标记目标的不同 “层级”，比如：

```
Theorem andb_commutative : b c, andb b c = andb c b.
Proof.
  intros b c. destruct b eqn:Eb.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
Qed.
```

每一对 reflexivity 调用和紧邻其上的 destruct 执行后生成的子目标对应。

除了 - 和 +，Coq 证明还可以使用 * 作为第三种标号。我们也可以用花括号将每个子证明目标括起来，这在遇到一个证明生成了超过三层的子目标时很有用：

```
Theorem andb_commutative' : b c, andb b c = andb c b.
Proof.
  intros b c. destruct b eqn:Eb.
  { destruct c eqn:Ec.
    { reflexivity. }
    { reflexivity. } }
  { destruct c eqn:Ec.
    { reflexivity. }
    { reflexivity. } }
```

Qed.

由于花括号同时标识了证明的开始和结束，因此它们可以同时用于不同的子目标层级，如上例所示。此外，花括号还允许我们在一个证明中的多个层级下使用同一个标号：

```
Theorem andb3_exchange :
  b c d, andb (andb b c) d = andb (andb b d) c.
Proof.
  intros b c d. destruct b eqn:Eb.
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
Qed.
```

在本章结束之前，我们最后说一种简便写法。或许你已经注意到了，很多证明在引入变量之后会立即对它进行情况分析：

```
intros x y. destruct y as [|y] eqn:E.
```

This pattern is so common that Coq provides a shorthand for it: we can perform case analysis on a variable when introducing it by using an intro pattern instead of a variable name. For instance, here is a shorter proof of the plus_1_neq_0 theorem above. (You'll also note one downside of this shorthand: we lose the equation recording the assumption we are making in each subgoal, which we previously got from the eqn:E annotation.)

```
Theorem plus_1_neq_0' : n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros [|n].
  - reflexivity.
  - reflexivity. Qed.
```

如果没有需要命名的参数，我们只需写上 [] 即可。

```
Theorem andb_commutative" :
  b c, andb b c = andb c b.
Proof.
  intros [] [].
  - reflexivity.
  - reflexivity.
  - reflexivity.
  - reflexivity.
Qed.
```

练习：2 星, standard (andb_true_elim2)

证明以下断言, 当使用 destruct 时请用标号标出情况（以及子情况）。

```
Theorem andb_true_elim2 : b c : bool,
  andb b c = true → c = true.
Proof.
  (* 请在此处解答 *) Admitted.
```

练习：1 星, standard (zero_nbeq_plus_1)

```
Theorem zero_nbeq_plus_1 : n : nat,
  0 =? (n + 1) = false.
Proof.
  (* 请在此处解答 *) Admitted.
```

关于记法的更多内容 (可选)

(通常，标为可选的部分对于跟进本书其它部分的学习来说不是必须的，除了那些也标记为可选的部分。在初次阅读时，你可以快速浏览这些部分，以便在将来遇到时能够想起来这里讲了些什么。)

回忆一下中缀加法和乘法的记法定义:

```
Notation "x + y" := (plus x y)
                    (at level 50, left associativity)
                    : nat_scope.
```

```
Notation "x * y" := (mult x y)
                    (at level 40, left associativity)
                    : nat_scope.
```

对于 Coq 中的每个记法符号，我们可以指定它的 **优先级**和 **结合性**。优先级 n 用 at level n 来表示，这样有助于 Coq 分析复合表达式。结合性的设置有助于消除表达式中相同符号出现多次时产生的歧义。比如，上面这组对 + 和 * 的参数定义的表达式 1+2*3*4 是 (1+((2*3)*4)) 的简写。Coq 使用 0 到 100 的优先级等级，同时支持 **左结合**、**右结合**和 **不结合**三种结合性。之后我们会看到更多与此相关的例子，比如 **列表**一章。

每个记法符号还与 **记法范围（Notation Scope）** 相关。Coq 会尝试根据上下文来猜测你所指的范围，因此当你写出 S(0*0) 时，它猜测是 nat_scope；而当你写出笛卡尔积（元组）类型 bool*bool 时，它猜测是 type_scope。有时你可能必须百分号记法 (x*y)%nat 来帮助 Coq 确定范围。另外，有时 Coq 打印的结果中也用%nat 来指示记法所在的范围。

记法范围同样适用于数值记法（3、4、5 等等），因此你有时候会看到 0%nat，表示 0（即我们在本章中使用的自然数零 0），而 0%Z 表示整数零（来自于标准库中的另一个部分）。

专业提示：Coq 的符号机制不是特别强大。别期望太多！

不动点 Fixpoint 和结构化递归 (可选)

以下是加法定义的一个副本：

```
Fixpoint plus' (n : nat) (m : nat) : nat :=
  match n with
  | O    m
  | S n'   S (plus' n' m)
  end.
```

当 Coq 查看此定义时，它会注意到 “plus' 的第一个参数是递减的”。这意味着我们对参数 n 执行了**结构化递归**。换言之，我们仅对严格递减的 n 值进行递归调用。这一点蕴含了 “对 plus' 的调用最终会停止”。Coq 要求每个 Fixpoint 定义中的某些参数必须是 “递减的”。

这项要求是 Coq 的基本特性之一，它保证了 Coq 中定义的所有函数对于所有输入都会终止。然而，由于 Coq 的 “递减分析” 不是非常精致，因此有时必须用一点不同寻常的方式来编写函数。

练习：2 星, standard, optional (decreasing)

To get a concrete sense of this, find a way to write a sensible Fixpoint definition (of a simple function on numbers, say) that _does_ terminate on all inputs, but that Coq will reject because of this restriction. (If you choose to turn in this optional exercise as part of a homework assignment, make sure you comment out your solution so that it doesn't cause Coq to reject the whole file!)

(* 请在此处解答 *)

更多练习

Each SF chapter comes with a tester file (e.g. BasicsTest.v), containing scripts that check most of the exercises. You can run make BasicsTest.vo in a terminal and check its output to make sure you didn't miss anything.

练习：1 星, standard (indentity_fn_applied_twice)

用你学过的策略证明以下关于布尔函数的定理。

```
Theorem identity_fn_applied_twice :
  (f : bool → bool),
  ( (x : bool), f x = x) →
  (b : bool), f (f b) = b.
Proof.
  (* 请在此处解答 *) Admitted.
```

练习：1 星, standard (negation_fn_applied_twice)

现在声明并证明定理 negation_fn_applied_twice，与上一个类似，但是第二个前提说明函数 f 有 f x = negb x 的性质。

```
(* 请在此处解答 *)
(* 下一行中的 Import 语句告诉 Coq 使用标准库中的 String 模块。
   在后面的章节中，我们会大量使用字符串，
   不过目前我们只需要字符串字面量的语法来处理评分器的注释。 *)
From Coq Require Export String.
```

```
(* 请勿修改下面这一行： *)
Definition manual_grade_for_negation_fn_applied_twice : option (nat*string) := None.
```

练习：3 星, standard, optional (andb_eq_orb)

请证明下列定理。（提示：此定理的证明可能会有点棘手，取决于你如何证明它。或许你需要先证明一到两个辅助引理。或者，你要记得未必要同时引入所有前提。）

```
Theorem andb_eq_orb :
  (b c : bool),
  (andb b c = orb b c) →
  b = c.
Proof.
  (* 请在此处解答 *) Admitted.
```

练习：3 星, standard (binary)

We can generalize our unary representation of natural numbers to the more efficient binary representation by treating a binary number as a sequence of constructors A and B (representing 0s and 1s), terminated by a Z. For comparison, in the unary representation, a number is a sequence of Ss terminated by an O.

For example:

decimal	binary	unary
0	Z	O
1	B Z	S O
2	A (B Z)	S (S O)
3	B (B Z)	S (S (S O))
4	A (A (B Z))	S (S (S (S O)))
5	B (A (B Z))	S (S (S (S (S O))))
6	A (B (B Z))	S (S (S (S (S (S O)))))
7	B (B (B Z))	S (S (S (S (S (S (S O))))))
8	A (A (A (B Z)))	S (S (S (S (S (S (S (S O)))))))

Note that the low-order bit is on the left and the high-order bit is on the right — the opposite of the way binary numbers are usually written. This choice makes them easier to manipulate.

```
Inductive bin : Type :=
  | Z
  | A (n : bin)
  | B (n : bin).
```

(a) Complete the definitions below of an increment function incr for binary numbers, and a function bin_to_nat to convert binary numbers to unary numbers.

```
Fixpoint incr (m:bin) : bin
```

(* 将本行替换成 ":= _ 你的 _ 定义 _ ." *). Admitted.

Fixpoint bin_to_nat (m:bin) : nat
(* 将本行替换成 ":= _ 你的 _ 定义 _ ." *). Admitted.

(b) Write five unit tests test_bin_incr1, test_bin_incr2, etc. for your increment and binary-to-unary functions. (A "unit test" in Coq is a specific Example that can be proved with just reflexivity, as we've done for several of our definitions.) Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

(* 请在此处解答 *)

(* 请勿修改下面这一行： *)
Definition manual_grade_for_binary : option (nat*string) := None.

(* Fri Mar 15 17:06:28 UTC 2019 *)