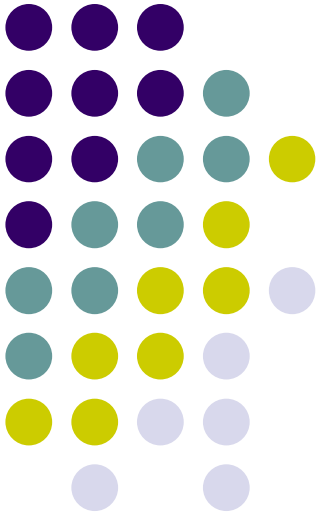
南京大学
Nanjing University

第四讲- coq入门



内容提要

- coq的概述
  - 历史
  - 功能
  - 逻辑系统
- 基本语法
  - 函数式编|数据类型|函数
- 基本证明策略
  - 简化|自反|重写|分类讨论

Part1- coq概述

coq概述

- 由INRIA的**Thierry Coquand**和**Gérard Huet**开发
- 是一种基于**构造演算（Calculus of Constructions）**的交互式定理证明器

C: > Users > borak > Desktop > 数理逻辑大纲 > examples > ch4 > exp1.v

```
1  Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
2  Proof.
3    intros P Q H.  (* 引入假设 *)
4    destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
5    split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
6    - apply HQ.    (* 证明第一个子目标 Q *)
7    - apply HP.    (* 证明第二个子目标 P *)
8  Qed.
```

MAIN 1 SHELVED 0 GIVEN UP 0

```
P, Q :  Prop
H :  P /\ Q

(1/1)
Q /\ P
```

代码编辑器

定义数据类型和函数

基于策略的证明脚本

证明视图

上半部分：上下文（已知变量和假设）

下半部分：当前需要证明的结论

coq的历史

- 1985年：基于**CoC（Calculus of Construction）**的早期版本（~ V4.10）
- 1991年：基于**CIC（Calculus of Inductive Construction）**的（V5）
- 1998年：引入了自动化策略，支持自动化的证明过程
- 2005年：**四色定理**证明
- 2006年：经过coq形式化验证的C编译器**CompCert**
- 2012年：奇数阶定理证明
- 2013年：**ACM 软件系统奖**
- 2017年~：基于coq验证/生成的工业级软件系统

功能1：证明助手

- 证明助手coq的核心机制
  - 用户输入指令，系统反馈当前目标
  - 系统自动管理上下文，每一步推理均基于显式的推理策略

exp1.v

C: > Users > borak > Desktop > 数理逻辑大纲 > examples > ch4 > exp1.v

```
1  Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
2  Proof.
3    intros P Q H.  (* 引入假设 *)
4    destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
5    split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
6    - apply HQ.    (* 证明第一个子目标 Q *)
7    - apply HP.    (* 证明第二个子目标 P *)
8  Qed.
```

证明的结构


```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.          (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.     (* 证明第一个子目标 Q *)
  - apply HP.     (* 证明第二个子目标 P *)
Qed.
```

关键词	名称	说明
Theorem	定理声明	类似编程中的 function 定义。也可以用 Lemma (引理), Example (例子)。
:	类型标注	在 coq 中，定理的内容 = 类型。
Proof	开始证明	此时界面会切换，显示 goal 和 context。
Qed	结束证明	只有当所有 goal 都解决后才能执行。
.	语句结束符	coq 的每一行命令都要以句号结尾。

证明的结构

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.          (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.     (* 证明第一个子目标 Q *)
  - apply HP.     (* 证明第二个子目标 P *)
Qed.
```

符号	说明	符号	说明
forall	∀ (全称量词)	exists	∃ (存在量词)
\/	∨ (逻辑或)	/\	∧ (逻辑与)
->	→ (蕴含)	~	¬ (逻辑非)
<->	↔ (等价)	=	= (相等)
<>	≠ (不等)		

证明的结构

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.          (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.     (* 证明第一个子目标 Q *)
  - apply HP.     (* 证明第二个子目标 P *)
Qed.
```

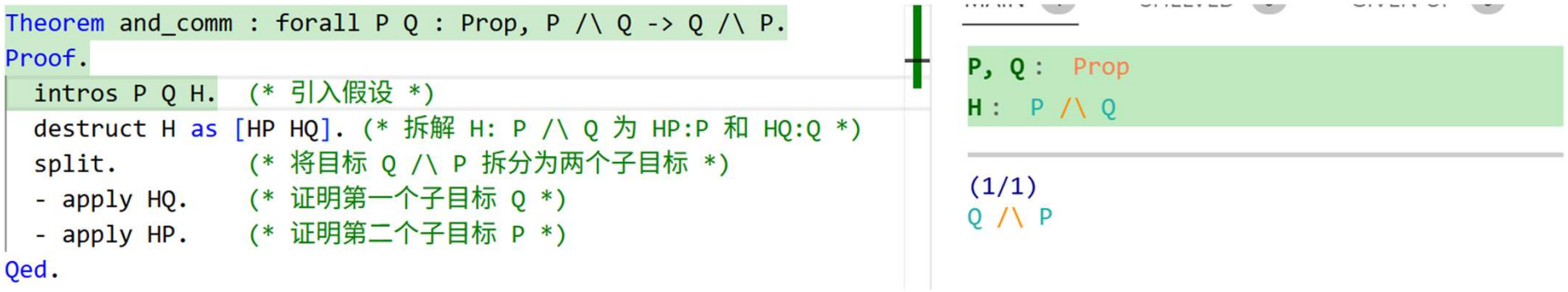
策略	作用方向	功能说明
intros	goal => Context	引入假设。将全称量词和前提移动到上下文。
destruct	操作 goal	拆解类型。把复杂类型（如 H）拆开。
split	操作 goal	拆分目标。将复杂的结论（如 P /\ Q ）分为两个子目标。
apply	context => goal	应用证据。用已有的证据直接匹配并解决当前目标。

证明过程

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.    (* 证明第一个子目标 Q *)
  - apply HP.    (* 证明第二个子目标 P *)
Qed.
```

```
(1/1)
forall P Q : Prop, P /\ Q -> Q /\ P
```

证明的过程



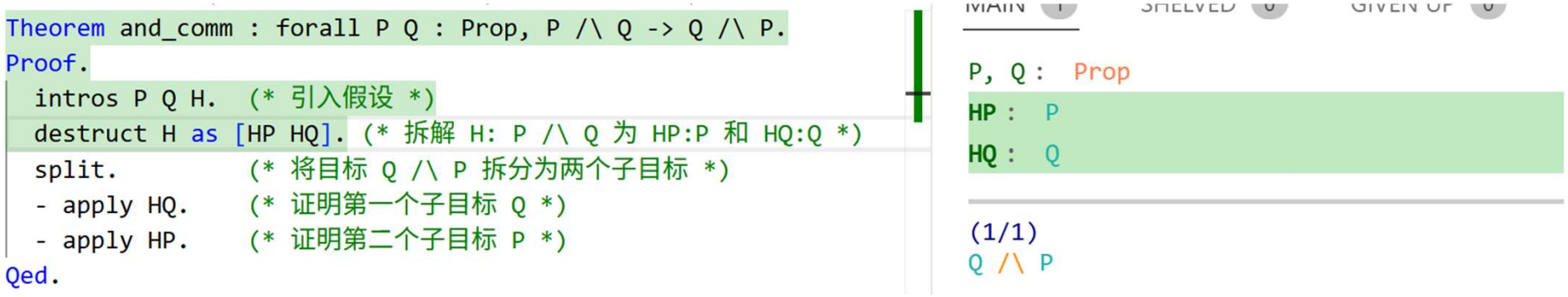
```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.    (* 证明第一个子目标 Q *)
  - apply HP.    (* 证明第二个子目标 P *)
Qed.
```

```
P, Q :  Prop
H :  P /\ Q

(1/1)
Q /\ P
```

- **操作含义：**将全称量词 `forall` 和蕴含前件 `->` 移动到上下文中。
- **逻辑解读：**假设存在命题`P`和`Q`，并且假设我们有一个证据`H`证明了`P /\ Q`成立。

证明的过程



- **操作含义：**消去规则（**/\ E**）。既然已知**H**是 **P /\ Q** 的证据，那么必然包含两部分：**P**的证据和**Q**的证据。进一步地，我们把它们拆出来，分别命名为**HP**和**HQ**。
- **逻辑解读：**既然我知道**P**和**Q**都为真，那我就把‘P为真’记在**context**中，叫**HP**，把‘**Q**为真’记在**context**中，叫 **HQ**。

证明的过程

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.          (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.     (* 证明第一个子目标 Q *)
  - apply HP.     (* 证明第二个子目标 P *)
Qed.
```

```
P, Q :  Prop
HP :  P
HQ :  Q

(1/2)
Q

(2/2)
P
```

- **操作含义：**引入规则（**/\ I**）。当前目标是**P/\Q**，要证明它，我们需要分别证明**Q**和**P**。这会将一个目标分裂成两个子目标。
- **逻辑解读：**要把这个大任务搞定，我得先搞定**Q**，再搞定**P**。

证明的过程

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.    (* 证明第一个子目标 Q *)
  - apply HP.    (* 证明第二个子目标 P *)
Qed.
```

```
P, Q :  Prop
HP :  P
HQ :  Q

(1/1)
Q
```

- **操作含义：** `goal 1`需要证明`Q`。检查`context`，发现手里正好有`HQ:Q`。直接应用它。

- **逻辑解读：** 你要证明`Q`？我手里刚好有`Q`的证据`HQ`，拿去吧。

证明的过程

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.    (* 证明第一个子目标 Q *)
  - apply HP.    (* 证明第二个子目标 P *)
Qed.
```

```
P, Q :  Prop
HP :  P
HQ :  Q

(1/1)
P
```

- **操作含义：** goal 2需要证明P。检查context，发现手里正好有HP:P。直接应用它。

- **逻辑解读：** 你要证明P？我手里刚好有P的证据HP，拿去吧。

证明的过程

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\ P.
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和 HQ:Q *)
  split.         (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.    (* 证明第一个子目标 Q *)
  - apply HP.    (* 证明第二个子目标 P *)
Qed.
```

Proof finished

- **操作含义：**检查完成，将定理保存到库中。

自动化证明


Settings
exp7.v 1
exp6.v
ProofView: exp7.v
ProofView: exp6.v
C: > Users > borak > Desktop > 数理逻辑大纲 > examples > ch4 > exp6.v
(* 一个看起来有点绕的算术问题 *)
Theorem math_magic : forall x y z : nat,
x > 10 -> (* 假设 x 大于 10 *)
y = x + 5 -> (* 假设 y 等于 x + 5 *)
z > y * 2 -> (* 假设 z 大于 y 的两倍 *)
z > 30. (* 结论：z 一定大于 30 *)
Proof.
(* 如果手动证明，你需要：
1. 把 y 展开为 x + 5
2. z > (x + 5) * 2
3. z > 2x + 10
4. 因为 x > 10，所以 2x > 20
5. 所以 2x + 10 > 30
6. 根据传递性 z > 30
在 Coq 里，只需要三个字母：
*)
intros. (* 把所有变量和假设引入上下文 *)
lia. (* 见证奇迹：Coq 自动计算并解决了所有线性不等式 *)
Qed.
MAIN 1
SHELVED 0
GIVEN UP 0
x, y, z : nat
H : x > 10
H0 : y = x + 5
H1 : z > y * 2
(1/1)
z > 30

功能2：编程语言

- coq的语言：Gallina
  - 一种**纯函数式编程语言**
    - 没有变量赋值，数据一经创建不可修改
    - 函数，而非语句/对象，作为程序的一等公民
  - 全函数性质
    - coq中的**所有函数必须在有限步内终止**
    - 逻辑系统的可验证性
  - 丰富的类型系统
    - 代数数据类型
    - **依赖类型**

斐波拉契数列

```
(* 1. 定义阶乘函数（类似于 Python 的递归）*)
Fixpoint fact (n : nat) : nat :=
  match n with
  | 0 => 1              (* 0! = 1 *)
  | S p => n * fact p (* n! = n * (n-1)! *)
  end.

Example fact_5_is_120 : fact 5 = 120.
Proof.
  reflexivity.  (* 策略：两边算一下，一样就行！  *)
Qed.
```

Proof finished

阶乘

```
Require Import Arith.

Fixpoint fact (n : nat) : nat :=
  match n with
  | 0 => 1
  | S p => n * fact p
  end.

Example fact_3_manual : fact 3 = 6.
Proof.
  replace (fact 3) with (3 * fact 2).
  replace (fact 2) with (2 * fact 1).
  replace (fact 1) with (1 * fact 0).
  replace (fact 0) with 1.

  reflexivity.
  reflexivity.
  reflexivity.
  reflexivity.
  reflexivity.
Qed.
```

Proof finished

功能3：严格的逻辑系统

- **Curry-Howard Correspondence**
  - 逻辑系统和编程语言在数学上是同构的
  - 命题即类型，证明即程序

    Propositions as Types,

    Proofs as Programs

- 逻辑任务：证明如果 **A→B** 且 **B→C**，那么 **A→C**
- 编程任务：如果我有一个把“苹果变成香蕉”的机器（函数f），还有一个把“香蕉变成西瓜”的机器（函数g），我就能造出一个“把苹果直接变成西瓜”的机器

Curry-Howard Correspondence

逻辑概念	代码对应	代码示例
命题 (Proposition P)	类型 (Type T)	Definition P : Prop := ...
证明 (Proof of P)	值/对象 (Instance of T)	Definition p_proof : P := ...
蕴含 (A -> B)	函数 (Function A -> B)	fun (x:A) => ...
真 (True)	单元类型 (Unit Type)	类型 True, 值 I
假 (False)	空类型 (Empty Type)	类型 False (无值)
且 (And /\)	积类型/元组 (Product/Pair)	(proof_A, proof_B)
或 (Or \/)	和类型/联合体 (Sum/Union)	left proof_A 或 right proof_B
全称 (Forall x)	依赖积 (Dependent Product)	forall x:nat, ...
存在 (Exists x)	依赖偶/结构体 (Dependent Pair)	exists x, ...

Part2- 基本语法

函数式编程 Functional programming

- 核心理念：通过定义描述数据之间关系的函数来实现计算过程
  - 命令式/过程式编程：通过修改数据来实现计算过程

```
public static int fib(int n) {
    // 1. 状态初始化 (State Initialization)
    // 我们定义了两个“slot” a 和 b
    int a = 0;
    int b = 1;
    // 2. 过程控制 (Control Flow)
    // 告诉计算机“如何跑圈”
    for (int i = 0; i < n; i++) {
        int temp = a;
        // 3. 状态修改 (Mutation / Side Effect)
        // 关键点：变量 a 的值被覆盖了，旧值消失
        a = b;
        b = temp + b;
    }
    return a;
}
```

```
(* 引入算术库以使用 + 符号 *)
Require Import Nat.

(* 定义函数：输入一个自然数，返回一个自然数 *)
(* Fixpoint 表示这是一个递归函数 *)
Fixpoint fib (n : nat) : nat :=
  (* 模式匹配：看看 n 长什么样？ *)
  match n with
  | O => 0
  (* 情况1：如果是 0，结果就是 0 *)
  | S O => 1
   (* 情况2：如果是 1 (0的后继)，结果就是 1 *)
  | S (S n') =>  fib n' + fib (S n')
   (* 情况3：如果是 n'+2，结果是：前两项之和 *)
  end.
```

函数式编程 Functional programming

- 特性1：不可变性 (Immutability)
  - 数据一旦创建，永不改变

```
| S (S n') =>  fib n' + fib (S n')
 (* 情况3: 如果是 n'+2，结果是: 前两项之和 *)
```

- 特性2：函数是一等公民 (First-Class Functions)
  - 函数和数据一样，可以作为参数/返回值/组成复杂结构

```
Fixpoint fib (n : nat) : nat :=
```

- 特性3：纯函数 (Pure Functions)
  - 函数内部不修改变量值，输入-输出确定

Gallina程序=数据类型 + 函数

数据类型

- 作用：**定义“存在”**
- 构成：归纳法组装数据
- 可以理解成是C/Java中的**recursive enum**

函数

- 作用：**定义“变换”**
- 构成：递归调用
- 可以理解成是C/Java中的**finite loop**

模式匹配

- 作用：**实现数据与函数的交互**
  - 数据如何被构造，就应该如何被匹配拆解
- 构成：分支逻辑 `match n with XXXX end.`
- 可以理解成是C/Java中的**switch-case**

归纳类型定义

- Coq 中几乎所有数据类型都是通过 **Inductive** 定义的

```
Inductive type_name : Type :=
  | constructor1
  | constructor2
  | ...
  | constructorN.
```

- **type_name**: 该数据类型的名称
- **Type**: 这个数据类型是个“类型”
  - **Sort = {Type, Prop, Set}**
- **constructor**: 这个类型的构造方法

枚举类型

- 可以直接列出**所有元素**的数据类型

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

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

复合类型

- 通过参数实现不同数据的构造

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

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
  (* primary 构造子携带一个 rgb 参数 *)
```

递归类型

- 如何定义自然数？

- 皮亚诺公理
  - 基础：0是一个自然数
  - 归纳：如果n是自然数，则n的后继S n也是自然数

- coq中的自然数类型 nat

```
Inductive nat : Type :=
  | O : nat
  | S (n : nat) : nat.
  (* 引用了自身！  *)
```

递归类型：有理数

```
(* 1. 先定义整数 Z（这也是归纳定义的） *)
Inductive Z : Type :=
  | Z0 : Z
  | Zpos : positive -> Z
  | Zneg : positive -> Z.

(* 2. 再定义有理数 Q，就是一个“分数”结构 *)
Inductive Q : Type :=
  | Qmake (num : Z) (den : positive).
  (* num / den *)
```

- 是否可以构造实数？
  - **Inductive** 要求**有限步构造序列**
  - coq处理实数的方式：**公理化定义** or **共归纳函数类型**

非递归函数 Definition

- 关键字：**Definition**

```
Definition name (arg1 : Type1) (arg2 : Type2) : ReturnType :=
  body.
```

- 定义一个逻辑“与”函数 **andb**

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

函数的核心机制：模式匹配

- coq 唯一的处理复杂数据的机制

```
match val with
| constructor1 vars => result1
| constructor2 vars => result2
...
end
```

  - `constructor vars`是一个完整的语法结构，如果当前构造子是常量，则无需声明`vars`
  - 必须覆盖所有可能的构造子
  - 从上到下匹配，匹配成功即停止
  - 对于不关心的值，可以用下划线_代替

递归函数 Fixpoint

- `S n`描述了 `n+1`，那么如何描述`n+m`？

- 关键字：`Fixpoint`
  - 即调用了自身的递归函数

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  (* 如果 n 是 0，结果就是 m *)
  | S n' => S (plus n' m)
  (* 如果 n 是 S n'，结果是 n'+m 的后继 *)
  end.
```

函数的特性

- 结构化递归
  - **核心限制：**所有 coq 函数必须终止
  - **递减原则：**递归调用的参数必须比当前参数“小”

```
| S n' => S (plus n' m)
```

- 一等公民
  - 函数可以作为参数传递
  - **关键字：fun**

```
Definition do_twice (f : nat -> nat) (x : nat) : nat :=
  f (f x).
(* 调用两次f *)
Compute (do_twice (fun x => x * x) 2).
(*  (2*2) * (2*2)  *)
```

中缀符号

- 为什么我们可以在coq中使用算术表达式？
  - `plus (plus 1 2) 3`

- 关键词：**Notation**

```
Notation "x + y" := (plus x y)
                     (at level 50, left associativity)
                    (* 解析规则：优先级与结合性 *)
                     : nat_scope.
                    (* 作用域：消除符号歧义 *)
```

  - 优先级等级越小，优先度越高
  - 左结合：减法；右结合：乘方

Part3- 基本证明策略

coq的证明策略

- coq证明的三要素
  - 声明：**Theorem / Lemma / Example** + 名字 + 命题
  - 推导：**Proof.** + 一系列策略
  - 证毕：**Qed.**

```
Theorem and_comm : forall P Q : Prop, P /\ Q -> Q /\
Proof.
  intros P Q H.  (* 引入假设 *)
  destruct H as [HP HQ]. (* 拆解 H: P /\ Q 为 HP:P 和
  split.          (* 将目标 Q /\ P 拆分为两个子目标 *)
  - apply HQ.     (* 证明第一个子目标 Q *)
  - apply HP.     (* 证明第二个子目标 P *)
Qed.
```

当前条件
context

```
P, Q :  Prop
HP :  P
HQ :  Q
```

```
(1/1)
Q /\ P
```

目标结论
goal

引入假设 intros

- 将全称量词forall或蕴含前件 ->从goal移动到context
  - intros x：移动全称量词forall x的变量x
  - intros H：移动蕴含前件到context，并将命名其为H

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.  (* <--- 关注这一行 *)
  ...
```

```
(1/1)
forall n : nat, 0 + n = n
```

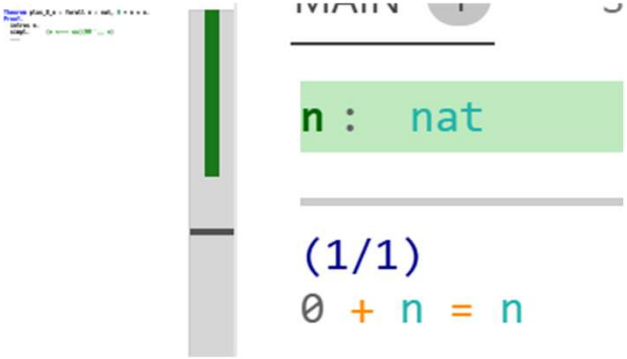
```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.  (* <--- 关注这一行 *)
  ...
```

```
n : nat
(1/1)
0 + n = n
```

简化计算 simpl

- 根据涉及到的函数定义进行计算/归约

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.      (* <--- 关注这一行 *)
  ...
```

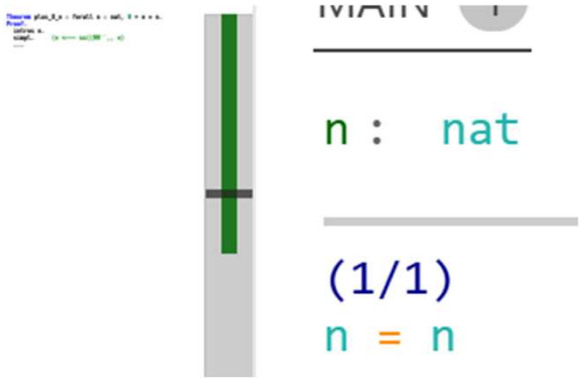


```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.      (* <--- 关注这一行 *)
  ...
```

自反性 reflexivity

- 证明当前的**context**等于**goal**
  - 公理 $\Gamma, A, \Delta \vdash \Theta, A, \Lambda$
  - 使用**simpl**对涉及的函数进行计算/规约

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.      (* <--- 关注这一行 *)
  ...
```



```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.
  reflexivity. (* <--- 关注这一行 *)
```

等式重写 rewrite

- 用等式假设进行替换
  - **H: XXX = YYY**
  - **rewrite -> H.** 将目标中的**XXX**替换为**YYY**
  - **rewrite <- H.** 将目标中的**YYY**替换为**XXX**

```
Theorem rewrite_example : forall n m : nat,
  n = m -> n + n = m + m.
Proof.
  intros n m H. (* H 是假设 n = m *)
  rewrite -> H. (* 把 n 替换成 m *)
  reflexivity.
Qed.
```

```
n, m : nat
H : n = m
(1/1)
n + n = m + m
```

```
Theorem rewrite_example : forall n m : nat,
  n = m -> n + n = m + m.
Proof.
  intros n m H. (* H 是假设 n = m *)
  rewrite -> H. (* 把 n 替换成 m *)
  reflexivity.
Qed.
```

```
n, m : nat
H : n = m
(1/1)
m + m = m + m
```

The Road Ahead

- 语言特性：
  - 更复杂的数据：结构化数据`Lists`
  - 更抽象的函数：多态与高阶函数
  - 更强大的策略：`apply, injection, discriminate, destruct, induction`

- 逻辑系统
  - 如何描述逻辑：命题 `Prop`
  - 如何理解逻辑：柯里-霍华德同构 `Curry-Howard correspondence`
  - 如何进行推理：自动化策略`auto, Ltac, eapply`