Coq 中的逻辑证明

1 关于“并且”的证明

要证明“某命题甲并且某命题乙”成立，可以在 Coq 中使用 `split` 证明指令进行证明。该指令会将当前的证明目标拆成两个子目标。

```
Lemma and_intro: forall A B: Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB.
  (** 拆分前，需要证明的结论是_[A /\ B]_ *)
  split.
  + (** 第一个分支的待证明结论是_[A]_ *)
    apply HA.
    (** 上面的_[apply]_指令表示在证明中使用一条前提，或者使用一条已经经过证明的
        定理或引理。*)
  + (** 第二个分支的待证明结论是_[B]_ *)
    apply HB.
Qed.
```

如果当前一条前提假设具有“某命题甲并且某命题乙”的形式，我们可以在 Coq 中使用 `destruct` 指令将其拆分成为两个前提。

```
Lemma proj1: forall P Q: Prop,
  P /\ Q -> P.
Proof.
  intros.
  (** 拆分前，当前证明目标有一个前提
      - H: P /\ Q *)
  destruct H as [HP HQ].
  (** 拆分后，当前证明目标有两个前提
      - HP: P
      - HQ: Q *)
  apply HP.
Qed.
```

另外，`destruct` 指令也可以不指名拆分后的前提的名字，Coq 会自动命名。

```
Lemma proj2: forall P Q: Prop,
  P /\ Q -> Q.
Proof.
  intros.
  destruct H.
  (** 拆分后，当前证明目标有两个前提
      - H: P
      - H0: Q *)
  apply H0.
Qed.
```

当前提与结论中，都有 `/\` 的时候，我们就既需要使用 `split` 指令，又需要使用 `destruct` 指令。

```
Theorem and_comm: forall P Q: Prop,
  P /\ Q -> Q /\ P.
Proof.
  intros.
  destruct H as [HP HQ].
  split.
  + apply HQ.
  + apply HP.
Qed.
```

Coq 证明脚本 1. apply 指令. 如果待证明结论与某个前提或者定理是相同的命题，那么就可以用 `apply` 指令完成证明。如果前提或定理中还包含额外的概称量词，`apply` 也能完成证明。例如，当前提 `H` 与结论分别为下面命题时

```
H: forall x: Z, x < x + 1
结论: u + 1 < (u + 1) + 1
```

`apply H` 或者 `apply (H (u + 1))` 都可以完成证明。除此之外，`apply` 指令还可以处理前提或者定理中包含一些附加条件的情况。例如，`Z.mul_nonneg_nonneg` 这条 Coq 标准库中的定理说的是:

```
forall n m, 0 <= n -> 0 <= m -> 0 <= n * m
```

于是，如果要证明 `0 <= 5 * n` 就可以使用下面的证明指令之一：

```
apply Z.mul_nonneg_nonneg
apply (Z.mul_nonneg_nonneg 5)
apply (Z.mul_nonneg_nonneg 5 n)
```

它们的效果都是将当前证明目标拆分两个：分别证明 `0 <= 5` 与 `0 <= n` 这两个结论。还可以使用下面证明指令：

```
apply (Z.mul_nonneg_nonneg 5 n ltac:(lia))
```

这样原结论就被规约为 `0 <= n` 这一个命题了。

习题 1. 请在不使用 `tauto` 指令的情况下证明下面结论。

```
Theorem and_assoc1: forall P Q R: Prop,
  P /\ (Q /\ R) -> (P /\ Q) /\ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem and_assoc2: forall P Q R: Prop,
  (P /\ Q) /\ R -> P /\ (Q /\ R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

2 关于“或”的证明

“或”是另一个重要的逻辑连接词。如果“或”出现在前提中，我们可以用 Coq 中的 `destruct` 指令进行分类讨论。在下面的例子中，我们对于前提 `P \/ Q` 进行分类讨论。要证明 `P \/ Q` 能推出原结论，就需要证明 `P` 与 `Q` 中的任意一个都可以推出原结论。

```
Fact or_example:
  forall P Q R: Prop, (P -> R) -> (Q -> R) -> (P \/ Q -> R).
Proof.
  intros.
  (** 拆分前，Coq证明目标中有三个前提：
      - H: P -> R
      - H0: Q -> R
      - H1: P \/ Q
      待证明结论是_[R]_。下面的_[destruct]_指令对_[H1]_分类讨论。 *)
  destruct H1 as [HP | HQ].
  + (** 第一种情况，_[P]_成立（前提_[HP]_） *)
    pose proof H HP.
    apply H1.
  + (** 第一种情况，_[Q]_成立（前提_[HQ]_） *)
    pose proof H0 HQ.
    apply H1.
Qed.
```

相反的，如果要证明一条形如 `A \/ B` 的结论整理，我们就只需要证明 `A` 与 `B` 两者之一成立就可以了。在 Coq 中的指令是：`left` 与 `right` 。例如，下面是选择左侧命题的例子。

```
Lemma or_introl: forall A B: Prop, A -> A \/ B.
Proof.
  intros.
  (** 选择前，要证明的结论是_[A \/ B]_。 *)
  left.
  (** 选择左侧后，要证明的结论是_[A]_。 *)
  apply H.
Qed.
```

下面是选择右侧命题的例子。

```
Lemma or_intror: forall A B: Prop, B -> A \/ B.
Proof.
  intros.
  (** 选择前，要证明的结论是_[A \/ B]_。 *)
  right.
  (** 选择右侧后，要证明的结论是_[B]_。 *)
  apply H.
Qed.
```

习题 2. 请在不使用 `tauto` 指令的情况下证明下面结论。

```
Theorem or_comm: forall P Q: Prop,
  P \/ Q  -> Q \/ P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 3. 请在不使用 `tauto` 指令的情况下证明下面结论。

```
Theorem or_assoc1: forall P Q R: Prop,
  P \/ (Q \/ R)  -> (P \/ Q) \/ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem or_assoc2: forall P Q R: Prop,
  (P \/ Q) \/ R -> P \/ (Q \/ R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 关于“当且仅当”的证明

在 Coq 中，`<->` 符号对应的定义是 `iff` ，其将 `P <-> Q` 定义为

`(P -> Q) /\ (Q -> P)`

因此，要证明关于“当且仅当”的性质，首先可以使用其定义进行证明。

```
Theorem iff_refl: forall P: Prop, P <-> P.
Proof.
  intros.
  (** 展开前，待证明的目标是_[P <-> P]_ *)
  unfold iff.
  (** 展开后，待证明的目标是_[(P -> P) /\ (P -> P)]_ *)
  split.
  + (** 第一个分支需要证明_[P -> P]_ *)
    intros.
    apply H.
  + (** 第二个分支也需要证明_[P -> P]_ *)
    intros.
    apply H.
Qed.
```

Coq 也允许在不展开“当且仅当”的定义时就使用 `split` 或 `destruct` 指令进行证明。

```
Theorem and_dup: forall P: Prop, P /\ P <-> P.
Proof.
  intros.
  (** 拆分前，待证明的目标是_[P /\ P <-> P]_ *)
  split.
  + (** 拆分后，第一个分支需要证明_[P /\ P -> P]_ *)
    intros.
    destruct H.
    apply H.
  + (** 拆分后，第二个分支需要证明_[P -> P /\ P]_ *)
    intros.
    split.
    - apply H.
    - apply H.
Qed.
```

```
Theorem iff_imply: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros P Q H.
  (** 拆分后，前提_[H]_是命题_[P <-> Q]_ *)
  destruct H.
  (** 拆分后，得到两个前提：
      - H: P -> Q
      - H0: Q -> P *)
  apply H.
Qed.
```

习题 4. 请在不使用 `tauto` 指令的情况下证明下面结论。

```
Theorem or_dup: forall P: Prop, P \/ P <-> P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 命题逻辑综合应用

下面是证明“并且”、“或”与“当且仅当”时常用的证明指令汇总与拓展。

Coq 证明脚本 2. left 指令与 right 指令. 如果待证明结论具有 `P \/ Q` 的形式，那么 `left` 可以将该结论规约为 `P` ，`right` 可以将该结论规约为 `Q` 。

Coq 证明脚本 3. 命题逻辑证明中的 split 指令. 如果待证明结论具有 `P /\ Q` 的形式，那么 `split` 可以将当前证明目标规约为两个更简单的证明目标，它们的前提与原证明目标的前提相同，它们的结论分别为 `P` 与 `Q` 。

Coq 证明脚本 4. 命题逻辑证明中的 destruct 指令. 如果证明时前提 `H` 具有形式 `P /\ Q` 或具有形式 `P \/ Q` ，则可以使用 `destruct H` 指令。当 `H` 具有形式 `P /\ Q` 时，该指令会将此前提分解为两个前提 `P` 与 `Q` 。当 `H` 具有形式 `P \/ Q` 时，该指令会将当前证明目标规约为两个证明目标，其中一个将前提 `H` 被改为 `P` ，另一个将前提 `H` 改为 `Q` 。

Coq 允许用户使用 `destruct ... as ...` 指令对 destruct 得到的新前提手动重命名。例如当 `H` 具有形式 `P /\ Q` 时，`destruct H as [H1 H2]` 指令将生成下面两个证明前提：

```
H1: P
H2: Q
```

又例如当 `H` 具有形式 `P \/ Q` 时，`destruct H as [H1 | H2]` 指令也可以用于手动命名。与 intros 指令中的规定一样，当需要对 destruct 结果中的一部分手动命名而对另一部分自动命名时，可以使用问号 `?` 表示那些需要由 Coq 自动命名的名字。

Coq 允许用户对多个前提同时执行 destruct 指令,例如 `destruct H, H0` 就表示先 `destruct H` 再 `destruct H0` 。Coq 也允许用户在一条 destruct 指令中对 destruct 的结果再进一步分解或进一步分类讨论，但具体需要分解多少层，需要使用 `destruct ... as ...` 指令做具体说明。例如，当 `H` 具有形式

```
(P /\ Q) /\ (R /\ S)
```

时，`destruct H as [? [? ?]]` 指令会将 `H` 分解为 `P /\ Q` 、`R` 与 `S` ；而 `destruct H as [[? ?] ?]` 指令会将 `H` 分解为 `P` 、`Q` 与 `R /\ S` 。另外，对“并且”与“或”的分解与分类讨论也可以相互嵌套，例如，当 `H` 具有形式

```
(P /\ Q) \/ R
```

时，可以使用 `destruct H as [[HP HQ] | HR]` 指令在分类讨论的同时对其中一个分类讨论的分支对前提做进一步分解。

Coq 证明脚本 5. 引入模式. 前面已经介绍，`destruct ... as ...` 指令可以一次性完成若干次的命题拆解或分类讨论。在这一指令中，`as` 之后的结构成为“引入模式”（intro-pattern）。以下是与目前所学相关的几种引入模式：

- 针对“并且”的命题拆解 `[intro_pattern1 intro_pattern2]` ；
- 针对“或”的分类讨论 `[intro_pattern1 | intro_pattern2]` ；
- 新引入的名字，例如 `H` ；

- 表示由 Coq 系统自动命名的问号 `?` ；
- 表示直接丢弃拆分结果的下划线 `_` ；

除了 destruct 指令之外，intros 指令与 pose proof 指令也可以使用引入模式，在完成原有功能的基础上，再进行一次 destruct。例如，`intros [H1 H2] [H3 | H3]` 相当于依次执行：

```
intros H1 H2
destruct H1 as [H1 H2]
destruct H3 as [H3 | H3] ;
```

而 `pose proof H x y as [H1 H2]` 相当于依次执行：

```
pose proof H x y as H1
destruct H1 as [H1 H2] 。
```

下面罗列了一些命题逻辑中的常见性质。请有兴趣的读者综合前几节所学内容，在不使用 `tauto` 证明指令的限制下完成下面证明。

习题 5. 请在 Coq 中证明“假言推理”规则。

```
Theorem modus_ponens: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 6. 请在 Coq 中证明“并且”对“或”的分配律。

```
Theorem and_or_distr_l: forall P Q R: Prop,
  P /\ (Q \/ R) <-> P /\ Q \/ P /\ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 7. 请在 Coq 中证明“或”对“并且”的分配律。

```
Theorem or_and_distr_l: forall P Q R: Prop,
  P \/ (Q /\ R) <-> (P \/ Q) /\ (P \/ R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 8. 请在 Coq 中证明“并且”对“或”的吸收律。

```
Theorem and_or_absorb: forall P Q: Prop,
  P /\ (P \/ Q) <-> P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 9. 请在 Coq 中证明“或”对“并且”的吸收律。

```
Theorem or_and_absorb: forall P Q: Prop,
  P \/ (P /\ Q) <-> P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 10. 请在 Coq 中证明“并且”能保持逻辑等价性。

```
Theorem and_congr: forall P1 Q1 P2 Q2: Prop,
  (P1 <-> P2) ->
  (Q1 <-> Q2) ->
  (P1 /\ Q1 <-> P2 /\ Q2).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 11. 请在 Coq 中证明“或”能保持逻辑等价性。

```
Theorem or_congr: forall P1 Q1 P2 Q2: Prop,
  (P1 <-> P2) ->
  (Q1 <-> Q2) ->
  (P1 \/ Q1 <-> P2 \/ Q2).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 12. 请在 Coq 中证明“或”能保持逻辑等价性。

```
Theorem imply_congr: forall P1 Q1 P2 Q2: Prop,
  (P1 <-> P2) ->
  (Q1 <-> Q2) ->
  ((P1 -> Q1) <-> (P2 -> Q2)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 13. 请在 Coq 中证明“并且”与“如果-那么”之间的关系。提示：必要时可以使用 `assert` 指令证明辅助性质。

```
Theorem and_imply: forall P Q R: Prop,
  (P /\ Q -> R) <-> (P -> Q -> R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 14. 请在 Coq 中证明“或”与“如果-那么”之间的关系。提示：必要时可以使用 `assert` 指令证明辅助性质。

```
Theorem or_imply: forall P Q R: Prop,
  (P \/ Q -> R) <-> (P -> R) /\ (Q -> R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

本章之后的内容中将介绍关于“任意”、“存在”与“非”的证明方式，在相关逻辑命题的证明过程中，涉及命题逻辑的部分将会灵活使用上面介绍的各种证明方式，包括 `tauto` 指令。习题中也不再限制使用 `tauto` 指令。

5 关于“存在”的证明

当待证明结论形为：“存在一个 `x` 使得...”，那么可以用 `exists` 指明究竟哪个 `x` 使得该性质成立。

```
Lemma four_is_even: exists n, 4 = n + n.
Proof. exists 2. lia. Qed.
```

习题 15.

```
Lemma six_is_not_prime: exists n, 2 <= n < 6 /\ exists q, n * q = 6.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

当某前提形为：“存在一个 `x` 使得...”，那么可以使用 Coq 中的 `destruct` 指令进行证明。这一证明指令相当于数学证明中的：任意给定一个这样的 `x` 。

```
Theorem dist_exists_and: forall (X: Type) (P Q: X -> Prop),
  (exists x, P x /\ Q x) -> (exists x, P x) /\ (exists x, Q x).
Proof.
  intros.
  destruct H as [x [HP HQ]].
  (** 拆分后的前提为：
      - HP: P x
      - HQ: Q x *)
  split.
  + (** 第一个分支需要证明：_[exists x0: X, P x0]_，这里Coq系统自动对结论中原先的
        _[exists x]_进行了重命名，将_[x]_改为了_[x0]_避免与_[destruct]_指令产生的
        _[x]_重名。*)
    exists x.
    apply HP.
  + (** 第二个分支需要证明：_[exists x0: X, Q x0]_。*)
    exists x.
    apply HQ.
Qed.
```

从上面证明可以看出，Coq 中可以使用引入模式将一个存在性的命题拆分，并且与其他引入模式嵌套使用。

习题 16. 请在 Coq 中证明下面性质：

```
Theorem exists_exists: forall (X Y: Type) (P: X -> Y -> Prop),
  (exists x y, P x y) <-> (exists y x, P x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

6 关于“任意”的证明

在逻辑中，与“存在”相对偶的量词是“任意”，即 Coq 中的 `forall` 。其实我们已经在 Coq 中证明了许多关于 `forall` 的命题，最常见的证明方法就是使用 `pose proof` 指令。下面是一个简单的例子。

```
Example forall_ex1: forall (X: Type) (P Q R: X -> Prop),
  (forall x: X, P x -> Q x -> R x) ->
  (forall x: X, P x /\ Q x -> R x).
Proof.
  intros X P Q R H x [HP HQ].
  (** 现在的证明环境中有三个前提:
      - HP: P x
      - HQ: Q x
      - H: forall x: X, P x -> Q x -> R x *)
  pose proof H x HP HQ.
  (** 现在的证明环境中有四个前提:
      - HP: P x
      - HQ: Q x
      - H: forall x: X, P x -> Q x -> R x
      - H0: R x *)
  apply H0.
Qed.
```

有时在证明中想要将前提中的命题与结论一起重新组成推断式或者概称句。这可以通过使用 `revert` 指令完成。某种意义上说，`revert` 指令是 `intros` 指令的反操作。

```
Example forall_ex1_alter_proof: forall (X: Type) (P Q R: X -> Prop),
  (forall x: X, P x -> Q x -> R x) ->
  (forall x: X, P x /\ Q x -> R x).
Proof.
  intros X P Q R H x [HP HQ].
  (** 现在的证明环境中有三个前提:
      - HP: P x
      - HQ: Q x
      - H: forall x: X, P x -> Q x -> R x
      同时结论为: R x *)
  revert x HP HQ.
  (** 通过revert指令HP、HQ与x都从前提中被移除了，而结论则变为了:
      forall x: X, P x -> Q x -> R x *)
  apply H.
Qed.
```

有时在证明中，我们不需要反复使用一个概称的前提，而只需要使用它的一个特例，此时如果能在 pose proof 指令后删去原命题，能够使得证明目标更加简洁。这可以使用 Coq 中的 specialize 指令实现。

```
Example forall_ex2: forall (X: Type) (P Q R: X -> Prop),
  (forall x: X, P x /\ Q x -> R x) ->
  (forall x: X, P x -> Q x -> R x).
Proof.
  intros X P Q R H x HP HQ.
  (** 现在的证明环境中有三个前提:
      - HP: P x
      - HQ: Q x
      - H: forall x: X, P x -> Q x -> R x *)
  specialize (H x ltac:(tauto)).
  (** 现在的证明环境中还是只有三个前提:
      - HP: P x
      - HQ: Q x
      - H: R x *)
  apply H.
Qed.
```

在上面的证明中，specialize 指令并没有生成新的前提，而是把原有的前提 `H` 改为了特化后的 `R x`，换言之，原有的概称命题被删去了。在 Coq 证明中，我们可以灵活使用 Coq 提供的自动化证明指令，例如，

在下面证明中，我们使用 `tauto` 指令大大简化了证明。尽管我们并不能在一开始就使用 `tauto` 直接完成证明，因为

`forall a: A, P a /\ Q a`

`forall a: A, P a`

`forall a: A, Q a`

这三个命题之间的逻辑联系不能仅仅通过命题逻辑推导得到，而要用到概称量词 `forall` 的性质。

```
Theorem forall_and: forall (A: Type) (P Q: A -> Prop),
  (forall a: A, P a /\ Q a) <-> (forall a: A, P a) /\ (forall a: A, Q a).
Proof.
  intros.
  split. (** 对_[<->]_拆分。*)
  + intros.
    split. (** 对_[/\]_拆分。*)
    - (** 这一分支的证明目标是：
          - H: forall a: A, P a /\ Q a
          - 结论：forall a: A, P a。*)
      intros a.
      specialize (H a).
      (** 此时证明目标中的前提和结论为：
          - H: P a /\ Q a
          - 结论：P a
          因此就可以用_[tauto]_完成剩余证明了，另外两个分支也是类似。*)
      tauto.
    - (** 这一分支的证明目标是：
          - H: forall a: A, P a /\ Q a
          - 结论：forall a: A, Q a。*)
      intros a.
      specialize (H a).
      tauto.
  + intros [HP HQ].
    (** 这一分支的证明目标是：
        - HP: forall a: A, P a
        - HQ: forall a: A, Q a
        - 结论：forall a: A, P a /\ Q a。*)
    intros a.
    specialize (HP a).
    specialize (HQ a).
    tauto.
Qed.
```

习题 17. 请在 Coq 中证明下面性质：

```
Theorem forall_forall : forall (X Y: Type) (P: X -> Y -> Prop),
  (forall x y, P x y) -> (forall y x, P x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 18. 请在 Coq 中证明下面性质：

```
Theorem forall_iff : forall (X: Type) (P Q: X -> Prop),
  (forall x: X, P x <-> Q x) ->
  ((forall x: X, P x) <-> (forall x: X, Q x)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

7 关于“非”的证明

“非”是一个命题逻辑连接词，它符合两条重要性质：排中律与矛盾律。排中律说的是，对于任意一个命题 P，P 与非 P 中必有至少有一个为真。在 Coq 标准库中排中律称为 `classic`，下面例子展示了在 Coq 应用排中律的方法。

```
Example not_ex1: forall n m: Z, n < m \/ ~ n < m.
Proof.
  intros.
  pose proof classic (n < m).
  apply H.
Qed.
```

矛盾律说的是，对于任意命题 P，P 与非 P 不能都为真。在 Coq 中，如果能从前提中同时推导出 P 与非 P 就意味着导出了矛盾。这一般可以用 `tauto` 完成证明。

```
Example not_ex2: forall P Q: Prop,
  P -> ~ P -> Q.
Proof. intros. tauto. Qed.
```

下面是一些关于“非”的重要性质，它们中的一部分可以直接使用 `tauto` 证明。

```
Theorem not_and_iff: forall P Q: Prop,
  ~ (P /\ Q) <-> ~ P \/ ~ Q.
Proof. intros. tauto. Qed.
```

```
Theorem not_or_iff: forall P Q: Prop,
  ~ (P \/ Q) <-> ~ P /\ ~ Q.
Proof. intros. tauto. Qed.
```

```
Theorem not_imply_iff: forall P Q: Prop,
  ~ (P -> Q) <-> P /\ ~ Q.
Proof. intros. tauto. Qed.
```

```
Theorem double_negation_iff: forall P: Prop,
  ~ ~ P <-> P.
Proof. intros. tauto. Qed.
```

在证明“非”与量词（`exists`，`forall`）的关系时，有时可能需要使用排中律。下面证明的是，如果不存在一个 `x` 使得 `P x` 成立，那么对于每一个 `x` 而言，都有 `~ P x` 成立。证明中，我们根据排中律，对于每一个特定的 `x` 进行分类讨论，究竟是 `P x` 成立还是 `~ P x` 成立。如果是后者，那么我们已经完成了证明。如果是前者，则可以推出与前提（不存在一个 `x` 使得 `P x` 成立）的矛盾。

```
Theorem not_exists: forall (X: Type) (P: X -> Prop),
  ~ (exists x: X, P x) -> (forall x: X, ~ P x).
Proof.
  intros.
  (** 此时的前提为_[H: ~ exists x, P x]_ *)
  pose proof classic (P x) as [H0 | H0].
  + assert (exists x: X, P x).
    { exists x. apply H0. }
    (** 上面_[assert]_得到的结论与前提_[H]_矛盾。 *)
    tauto.
  + apply H0.
Qed.
```

下面再证明，如果并非每一个 `x` 都满足 `P x` ，那么就存在一个 `x` 使得 `~ P x` 成立。我们还是选择利用排中律进行证明。

```
Theorem not_forall: forall (X: Type) (P: X -> Prop),
  ~ (forall x: X, P x) -> (exists x: X, ~ P x).
Proof.
  intros.
  pose proof classic (exists x: X, ~ P x) as [H0 | H0].
  + tauto.
  + pose proof not_exists _ _ H0.
    assert (forall x: X, P x <-> ~ ~ P x). {
      intros.
      tauto.
    }
    pose proof forall_iff _ P (fun x => ~ ~ P x) H2.
    tauto.
Qed.
```

利用前面的结论，我们还可以证明 `not_all` 的带约束版本。

```
Corollary not_forall_imply: forall (X: Type) (P Q: X -> Prop),
  ~ (forall x: X, P x -> Q x) -> (exists x: X, P x /\ ~ Q x).
Proof.
  intros.
  pose proof not_forall _ _ H.
  destruct H0 as [x H0].
  exists x.
  pose proof not_imply_iff (P x) (Q x).
  tauto.
Qed.
```