高级编程

一个简单的求和程序

```
sum_to(1, 1).
sum_to(N, R) :-
        N1 is N - 1,
        sum_to(N1, R1),
        R is N + R1.

?- sum_to(3,X).
```

运行结果是

```
?- sum_to(3,X).
X = 6 ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 1.0Gb, global: 30Kb, trail: 1Kb
ERROR:   Stack depth: 11,183,606, last-call: 0%, Choice points: 3
ERROR:   In:
ERROR:     [11,183,606] user:sum_to(-11183593, _7882)
ERROR:     [11,183,605] user:sum_to(-11183592, _7902)
ERROR:     [11,183,604] user:sum_to(-11183591, _7922)
ERROR:     [11,183,603] user:sum_to(-11183590, _7942)
```

```
ERROR:     [11,183,602] user:sum_to(-11183589, _7962)
ERROR:
ERROR: Use the --stack_limit=size[KMG] command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648). to
double the limit.
```

原因是当运行到sum_to(1, R)的时候，尽管sum_to(1, 1)已经提供解，但是sum_to(N, R)的第二个定义还会运行，称为回溯(Backtracking)。

糟糕的是：这个运行是无法终止的。

解决前面的问题可以用剪切(Cut)技术，将程序改为

```
sum_to(1, 1) :- !.
sum_to(N, R) :-
        N1 is N - 1,
        sum_to(N1, R1),
        R is N + R1.

?- sum_to(3,X).
```

剪切技术体现在“!”上，相应的运行结果是：

```
?- sum_to(3,X).
X = 6.
```

如果不用剪切技术,还有另外一种解决方案，就是加入一句“\+(N = 1)”:

```
sum_to(1, 1).
sum_to(N, R) :-
        \+(N = 1),
        N1 is N - 1,
        sum_to(N1, R1),
        R is N + R1.
```

运行结果是:

```
?- sum_to(3,X).
X = 6 ;
false.
```

一个飞鸟的例子

```
bird(sparrow).
bird(eagle).
bird(duck).
bird(crow).
bird(ostrich).
bird(puffin).
bird(swan).
bird(albatross).
bird(starling).
bird(owl).
bird(kingfisher).
bird(thrush).

can_fly(ostrich):- fail.
can_fly(X):-bird(X).

?- can_fly(ostrich).
true.
```

要纠正这个错误，可以使用cut with failure组合，改为

```
can_fly(ostrich):- !,fail.
can_fly(X):-bird(X).
```

再运行就对了

```
?- can_fly(ostrich).
false.
```

加速器

考虑以下两种方法实现阶乘：

```
fact_simple(0, 1):- !.
fact_simple(N, F) :-
    N1 is N-1,
    fact_simple(N1, F1),
    F is N*F1.
?- fact_simple(6, F).
F = 720.
```

Prolog系统需要维持一个不断增长的堆栈。

使用加速器(Accumulator)，避免维护堆栈。

```
fact_acc(N, F) :- fact_acc(N, 1, F).
fact_acc(0, Acc, Acc):- !.
fact_acc(N, Acc0, F) :-
    N1 is N-1,
    Acc is Acc0 * N,
    fact_acc(N1, Acc, F).
?- fact_acc(6, F).
F = 720.
```

无加速器版的实现不断调用自己完成计算，而有加速器版的实现使用Acc作为加速变量(中间变量).

计算Fibonacci数列

```
fib(1,1):- !.
fib(2,1):- !.
fib(S,N) :-
    A is S-1,
    B is S-2,
    fib(A,C),
    fib(B,D),
    N is C+D.
?- fib(38,X).
X = 39088169.
```

```
fibo(N,F):-
    N > 0,
    fibacc(N,0,1,F).
fibacc(1,_,F,F):- !.
fibacc(N,A1,A2,F) :-
    N1 is N-1,
    Acc is A1+A2,
    fibacc(N1,A2,Acc,F).
?- fibo(38,X).
X = 39088169.
```

加速效果在N=38的时候已经很明显，不用加速器运行时间很长，而使用加速器非常快。