递归函数

作者：thzt

目录

一、开篇

提到函数式编程，人们最多想到的可能是它的某些性质，

例如，不可变性，无副作用，惰性求值，类型推导，等等。

然而，这些性质可能并不是它能吸引粉丝的根本原因，
而是它从工业界触手可及的直接应用出发，带我们看到了人类能力的边界，
函数式编程仿佛一座桥梁，让我们普通程序员也能窥探计算机科学的奥秘。

λ 演算是一个简洁的演算系统，但是它的计算能力却能比肩复杂的现代计算机，
因为它的简洁性质，以及与 Lisp 语言的紧密关系，对 λ 演算有所了解的程序员也比较多。

提到编程语言，上下文无关文法，以及正则文法，熟悉的人可能会更多，
书写正则表达式，开发和应用 DSL，甚至阅读编程语言的语法规范，都离不开它们。

然而，逻辑系统和递归函数论却鲜有人提及，
对如何书写递归函数人们可能仍旧心存畏惧，对类型的推导过程和原理也如坠迷雾。

其实，这是一块广袤的领域，其背景知识可能涉及了数学归纳法，良基或结构归纳法，
可计算性理论，不动点算子，哥德尔定理，以及证明论，模型论，等等。
数学背景，我们可能还需要补充，抽象代数，集合论，数理逻辑等离散数学的内容。

然而，我们的收获将是巨大的，
我们会看到用有限表示无限的递归思想，以及由这种思想导致的各种计算模型的能力限制。
递归与其他领域触类旁通之后，我们将走到数学，逻辑，计算机科学的交叉点上，
我认为这是一件值得高兴的事情。

二、编写递归函数的思路和技巧

递归，是一个熟悉而陌生的概念，说它熟悉，是因为人们经常提起它，

而说它陌生，指的是人们在实际编程中几乎不会主动使用它。

给定一个问题，如果本质上它能看做一个调用自身的规模较小的一个子问题来求解，
那么给出一个递归的算法解，就是很自然的。
然而，即使是这样，编制一个递归函数也是一件令人头疼的事情。

本系列文章的目的，可能并不局限于指出如何编写一个递归函数，
而是期望想从递归函数开始，了解它相关的科学知识，以达到对不同领域触类旁通的效果。

1. 从一个简单的例子开始

首先，我们来重温一下递归的概念，
维基百科上是这样描述的，

> 递归（recursion），在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。

我们来看一个简单的例子吧。（Haskell 代码

```
fact :: Int -> Int
fact 1 = 1
fact n = n * fact (n-1)
```

在这个例子中，
第一行 `fact :: Int -> Int` 表示了 `fact` 函数的类型，
第二行和第三行定义了函数 `fact`，
我们看到第三行，在对 `fact` 函数定义的时候，等式右边又出现了 `fact`，
这样定义的函数 `fact` 是递归的。

我们调用一下 `fact`，来看看结果，

```
fact 10
3628800
```

嗯嗯，`fact` 就是阶乘函数。

2. 写递归函数的步骤

那么，给定一个问题，我们编写一个递归函数，要如何开始呢？

（1）递推式

首先，我们要找到“递推式”。

例如，在数学上阶乘的定义是，$f(n)=n!$，这样的表述形式，不具有递推性。
我们先要想办法把 $f(n)$用 $f(n-1)$表示出来。

经过思考之后，我们可以证明，$f(n)=n*f(n-1)$，
于是，我们就走出了关键的第一步，得到了“递推式”。

（2）找出终止条件

有了“递推式”还不行，我们还需要确定递推在什么时候终止。
我们知道 $f(1)=1$，$f(2)=2*f(1)$，$f(3)=3*f(2)$，等等，
因此，我们只需要指定 $f(1)=1$，那么递推就会在 $f(n)$，当 $n=1$ 的时候终止了。

终止，就是不再调用规模更小的问题了。
这时，终止条件是 $f(1)=1$。

（3）用数学归纳法证明解的正确性

这一步是很重要的，有很多人都缺少证明递推式正确性的环节，
但是，考虑到介绍数学归纳法及其扩展会占用不少篇幅，
这里先略去，下一篇我们再回来讨论它。

这里，我们先假定，根据“递推式”和“终止条件”，使用数学归纳法，
我们已经证明了这样定义的 $f(n)$就是 $n!$。

（4）根据递推式和终止条件，编写程序

有了“递推式”和“终止条件”，再编写程序就水到渠成了。
很多人一上来就开始编码，就会感觉毫无头绪。

我们再来看下那段程序，

```
fact :: Int -> Int
fact 1 = 1
fact n = n * fact (n-1)
```

这不就是“递推式”和“终止条件”的忠实表示吗？
我们用 `fact 1 = 1` 表示了 $f(1)=1$，
用 `fact n = n * fact (n-1)` 表示了 $f(n)=n*f(n-1)$。

3. 小技巧

我们再看个复杂一点的例子。

在实际项目中，我们可能会遇到循环 n 次的场景，
在循环过程中，我们会根据索引进行运算，然后将某些符合条件的运算放到最终的结果中。

例如，我们选择 10 以内的所有偶数，

```
[x|x <- [0..9], x `mod` 2 == 0]
[0,2,4,6,8]
```

使用以上列表解析（list comprehension）的方法，我们可以快速得到结果。
但是这里，我们想要拿它来举例，介绍一个编写递归函数常用的小技巧。

为了通用性，我们考虑循环 `n` 次，将索引传入函数 `fn`，根据 `fn` 的返回值，将结果放入一个列表中。

（1）困境

根据前文介绍的编写步骤，我们需要先找到“递推式”和“终止条件”。

“终止条件”怎么写呢？

假如我们定义的递归函数称为 myLoop，那么 $myLoop(0,fn)$就是终止条件，它应该返回一个列表。
但是这个列表在参数中没有，它随着递归调用的过程“积累”得到的。

好吧，那我们看“递推式”。
$myLoop(n,fn)$要用 $myLoop(n-1,fn)$的结果计算出来，
我们需要先用索引调用 fn，然后再根据 fn 的返回值，放入结果列表，再继续调用 $myLoop(n-1,fn)$。
可是，索引从哪来呢？（n 不是索引，因为索引从 0 开始，而 n 是逐渐变小的。

这是两个典型的困难，
其一，我们在递归的过程中“积累”了某些东西，
其二，我们需要传递和递归过程相关的“索引”。

（2）解法

这时候，我们的小技巧就有用武之地了。

> 我们可以编写一个辅助的递归函数，通过增加参数的办法，提高灵活性。

例如，我们可以编写一个辅助函数 myLoop'，然后用 myLoop'来实现 myLoop。

```
myLoop :: Int -> (Int -> Maybe a) -> [a]
myLoop n fn = myLoop' n 0 fn []
myLoop' :: Int -> Int -> (Int -> Maybe a) -> [a] -> [a]
myLoop' 0 i fn lst = lst
myLoop' n i fn lst = case fn i of
  Just x -> myLoop' (n-1) (i+1) fn (lst++[x])
  Nothing -> myLoop' (n-1) (i+1) fn lst
```

以上，我们为 myLoop'增加了参数 i 和 lst，分别表示“索引”和“积累”的列表。
然后，myLoop 就可以用 myLoop'来实现了。

别忘了测试一下最终的结果，

```
myLoop 10 (\x -> if x `mod` 2 == 0 then Just x else Nothing)
[0,2,4,6,8]
```

（3）其他考虑

合理的利用递归函数的返回值，会减少附加参数的数量，例如，

```
myLoop :: Int -> (Int -> Maybe a) -> [a]
myLoop n fn = myLoop' n 0 fn
myLoop' :: Int -> Int -> (Int -> Maybe a) -> [a]
myLoop' 0 i fn = []
myLoop' n i fn = case fn i of
    Just x -> x:(myLoop' (n-1) (i+1) fn)
    Nothing -> myLoop' (n-1) (i+1) fn

```

但最终得到的递归函数就不是尾递归了，
关于尾递归，我们将在后续文章中讨论它。

参考

维基百科 - 递归
维基百科 - 递推关系式

三、归纳原理

自然数归纳法

自然数归纳法，是一种数学证明方法，通常被用于证明某个给定命题在整个（或者局部）自然数范围内成立。
它可以用一个有限的方式写出一个无限的证明。
后续文章中我们会看到，这种用有限表示无限的方法，其实是有局限性的，并不能用来解决所有的问题，
它能处理的只是无限中的一个子集罢了。

自然数归纳法，我们可以描述如下：
为证明对每一个自然数 n，命题 $P(n)$为真，只需要证明两件事，
（1）对于自然数 1，命题 $P(1)$为真
（2）如果对于自然数 m，命题 $P(m)$为真，那么对于自然数 $m+1$，命题 $P(m+1)$也为真

其中，第(1)条称为起始条件，第(2)条称为递推条件，或者称为归纳步骤。
第(2)条中，为了证明 $P(m+1)$而假设的 $P(m)$，称为归纳假设。

这似乎是很显然的事情，我们可以在一张无限长的纸带开头写上初始条件 $P(1)$，
接着根据递推条件，由 $P(1)$我们可以证明 $P(2)$成立，
重复这种思想，我们可以由 $P(2)$证明 $P(3)$成立，如此不断的进行下去，
最终，对于每个自然数 n，我们都能证明 $P(n)$成立。

但是，这样并不算是一个有效的证明。
要证明自然数归纳法的正确性，我们还需要补充一些集合论方面的知识。
然而，在此之前，我们还是先来看自然数归纳法的一个例子吧。

例子

在上一篇，我们在定义递归函数 `fact` 的时候，
先找到了“递推式”，再找到了“终止条件”，然后写出了 `fact` 的定义：

```
fact :: Int -> Int
fact 1 = 1
fact n = n * fact (n-1)
```

我们还提到，有一个步骤是必不可少的，

那就是证明 `fact` 的正确性，即证明这样定义的 `fact` 就是阶乘函数 $f(n)=n!$。

现在，我们正好可以用自然数归纳法来证明它。
我们假设命题 $P(n)$为：`fact n` 的值为 $n!$

（1）对于自然数 1，命题 $P(1)$：`fact 1` 的值为 $1!=1$，成立
（2）假设对于自然数 m，命题 $P(m)$：`fact m` 的值为 $m!$，成立；
那么，我们可以得到 `fact (m+1) = (m+1) * fact m`，值为$(m+1)*m!=(m+1)!$，也成立。

所以，对于任意自然数（$n\geqslant 1$），`fact n` 的值就是 $n!$。
于是，我们证明了 `fact` 就是阶乘函数 $f(n)=n!$。

自然数归纳法还有另外一种等价形式，
如果要证明 $P(n)$对每一个自然数 n 为真，
只要证明对于任意自然数 m，如果 $P(i)$当 $i<m$ 为真，那么 $P(m)$也为真。

集合上的关系

关系是一个日常生活用语，例如，“同学关系”，“我们的关系很好”之类的。
然而，它也是一个集合论中的概念，这给我们带来了很多困扰。
为了避免歧义，本文中从这里开始，我们开始谈论数学，我们要对集合上的“关系”进行定义。

直观的说，集合 A 的元素和集合 B 的元素之间的关系是一个二元性质 R，
使得对于每个 $a\in A$ 和 $b\in B$ 而言，$R(a,b)$要么为真，要么为假。

关系通常表示为一个集合，它是笛卡尔积的子集，即，
集合 A 和集合 B 之间的关系 R 是它们笛卡尔积的一个子集 $R\subseteq A\times B$。
如果序对(a,b)属于子集 R，则认为 a 与 b 之间的关系为真，否则认为 a 与 b 之间的关系为假。
通常关系直接描述为 $R(a,b)$，或者 aRb，而不用$(a,b)\in R$。

除了二元关系之外，对任何正整数 k，还可以定义 k 元关系。
如果 $A1,\cdots,Ak$为集合，则在 $A1,\cdots,Ak$上的 k 元关系是笛卡尔积 $A1\times\cdots\times Ak$的一个子集。

某个集合上的二元关系有很多性质，例如自反性，对称性，反对称性，传递性。

一个关系 $R \subseteq A \times A$ 是自反的，如果 $R(a,a)$对于所有的 $a \in A$ 成立；
是对称的，如果 $R(a,b)$就有 $R(b,a)$，对于所有的 $a,b \in A$ 都成立；
是反对称的，如果 $R(a,b)$且 $R(b,a)$，则 a,b 是同一个元素，对于所有的 $a,b \in A$ 都成立；
是传递的，如果 $R(a,b)$和 $R(b,c)$能推出 $R(a,c)$，对于所有的 $a,b,c \in A$ 都成立。
（注意，反对称性不是对称性的否定。

等价关系是同时具有自反性，对称性和传递性的关系。
偏序关系是具有自反性，反对称性和传递性的关系。
等价关系的一个例子就是相等性，相等性关系 $R(a,b)$当且仅当 a,b 是同一个元素。
偏序关系，例如通常的序关系 $R \subseteq N \times N$，$R(a,b)$当且仅当 $a \leqslant b$。

良基关系

归纳法有各种各样的形式，自然数归纳法只是其中的一种应用，
在数理逻辑和形式语言理论中，用的最多的是结构归纳法，在树形结构上进行归纳，后续文章中我们会提到。

人们总结了各种归纳法的共性，提出了良基关系的概念，
于是，自然数归纳法和结构归纳法都变成了在良基关系上通用归纳法的具体应用了。

集合 A 上的良基关系（well-founded relation），是 A 上的一个二元关系 $\prec$，
如果不存在无限下降序列（infinite descending sequence）$a0 \succ a1 \succ a2 \cdots$。
例如，自然数上的关系 $<$，就是一个良基关系。
但是 $\leqslant$ 却不是，因为存在一个无限下降序列 $a0 \geqslant a1 \geqslant a2 \cdots$。

根据良基关系，我们可以定义集合中的最小元，
$a \in A$ 为最小元，如果不存在 $a' \in A$，使得 $a' \prec a$。

对于良基关系，有一个等价的定义，
A 上的二元关系 $\prec$ 是良基的，当且仅当 A 的每一个非空子集 B 有最小元。

我们可以证明一下这两种说法等价性。
要证当且仅当，我们需要证明充分性和必要性，
（1）充分性：
要证：A 上的二元关系 $\prec$ 是良基的，则 A 的每一个非空子集 B 有最小元。
使用反证法，如果 B 没有最小元，则对于每个 $a \in B$，总可以找到 $a' \in B$，使得 $a' \prec a$。

但是，如果这样的话，我们就可以对任何 $a0 \in B$，以 $a0$开始构造一个无限下降序列 $a0 > a1 > a2 \cdots$，
这与$<$是一个良基关系矛盾。充分性证毕。

（2） 必要性：
要证：如果 A 的每一个非空子集 B 都有最小元，则 A 上用于比较的二元关系$<$是良基的。
由于 A 的每一个非空子集 B 都有最小元，则不可能存在无限下降序列 $a0 > a1 > a2 \cdots$，
因此，$<$是良基的。必要性证毕。

因此，A 上的二元关系$<$是良基的，当且仅当 A 的每一个非空子集 B 有最小元。

良基归纳法

设$<$为集合 A 上的良基二元关系，并且设 P 为关于 A 中元素的某个命题，
如果 $P(b)$对于所有的 $b < a$ 成立，就必然有 $P(a)$成立，
那么 $P(a)$就对所有的 $a \in A$ 成立。

我们看到$<$确实是自然数集上的良基关系，因此自然数归纳法只是良基归纳法的一种特例。

现在我们有了足够的能力来证明自然数归纳法的正确性了，
只要我们证明了良基归纳法是正确的。

还是用反证法：
我们期望证明，
前提：如果 $P(b)$对于所有的 $b < a$ 成立，必然有 $P(a)$成立，
结论：那么对于所有的 $a \in A$，$P(a)$都成立。

如若不然，假设存在 $x \in A$，使得 $P(x)$不成立，
则集合 $B = \{a \in A \mid \neg P(a)\}$非空，
因此根据良基关系的等价定义，集合 B 必有最小元 $m < B \subseteq A$，
而且，$\neg P(m)$成立。

则根据前提的逆否命题，一定存在 $b < m$，使得$\neg P(b)$成立，
所以，我们有 $b \in B$，且 $b < m$，与 m 是 B 的最小元矛盾。

证毕。

由此，我们证明了良基归纳法的正确性。
理解良基关系和偏序关系，是理解递归和不动点算子的第一步。

总结

本文从自然数归纳法出发，补充了一些集合论方面的知识，
让我们熟悉了集合上的几种常用关系，例如，等价关系，偏序关系和良基关系，
这些关系在以后的文章中还会被再次提到。

最后，我们证明了良基归纳法，从而证明了自然数归纳法的正确性。
不知道是否很明显了，递归的步骤和归纳的步骤，简直是太像了，
这一定不是偶然。

在 The Little Prover 一书中，为了证明递归函数是否全函数（total function），
作者使用了测度（measure）的概念，这实际上定义了参数集上的一个良基关系。
全函数是可计算理论中一个很重要的概念，
到底什么是全函数，什么是测度？下文我们再详细讨论。

参考

维基百科 - 数学归纳法
维基百科 - 二元关系
维基百科 - 良基关系
程序设计语言的形式语义

四、全函数与计算的可终止性

回顾

上文我们讨论了集合上的关系，还讨论了数学归纳法的一种普遍形式，称为良基归纳法，
它建立在集合上的良基关系之上。

本文开始讨论函数，我们将回顾函数的定义，
然后解释什么是全函数（total function），什么是部分函数（partial function）。

我们会看到，在证明一个递归函数是全函数时，
良基归纳法起到了重要作用。

在分析学中，人们似乎很少关心函数的完全性，
只关心它的连续性，可导性，可微性与可积性，等等。
而在计算机科学领域中，人们更在意计算的可终止性，
因此一个函数在某个点是否有定义将经常被提及。

程序中定义的函数，往往对应于某个集合上的数学函数，
为了描述程序的非终止性，就得扩充这个数学函数的定义域和值域。
为了理解这些事情，我们先要从函数的定义开始。

函数

集合 A,B 上的关系，是笛卡尔积 $A\times B$ 的一个子集。

而函数 $f:A\to B$，则是集合 A,B 上的一种特殊关系，
它要求 A 中的每一个元素，都有 B 中唯一确定的元素与之对应。
其中，集合 A 称为函数 f 的定义域，集合 B 称为函数的值域。

函数是我们熟悉的概念，这里只是提到了它本质上是集合上的一个关系。

（1）部分函数（partial function）
如果 f 是从 A 到 B 的二元关系，且 $\forall a\in A$，$f(a)=\varnothing$ 或 $\{b\}$，
则称 f 是从 A 到 B 的部分函数，或 A 上的部分函数。

其中，如果 $f(a)=\{b\}$，则称 $f(a)$ 有定义，记为 $f(a)\downarrow$，

也称 b 为 f 在 a 点的函数值，记为 $f(a)=b$。
如果 $f(a)=\varnothing$，则称 $f(a)$无定义，记为 $f(a)\uparrow$。

（2）全函数（total function）
如果$\forall a\in A$ 都有 $f(a)\downarrow$，则称 f 是 A 上的全函数，
此时，可以记为 $f:A\to B$。

可见，我们熟悉的函数，指是全函数。
值得注意的是，部分函数的定义已经包含了我们学过的“函数”的定义，
后文中，我们提到的“函数”如果不强调它的完全性的话，都泛指部分函数。

非终止性

部分函数在计算机科学中是非常重要的，
因为对于每一个 $a\in A$，一个算法可以表示为，计算出集合 B 中与之对应元素的过程，
这个算法可能对于某些值 $a\in A$ 不会终止，而这种情况是很常见的。

例如：

```
f :: Int -> Int
f 1 = 1
f n = n + f(n-2)
```

这样定义的函数 `f`，对应了数学上的一个部分函数 f，它只在某些情况下有意义，
只有当 `n` 是奇数时，我们才能得到终止性的结果。
而当 `n` 是偶数时，算法会无限的递归下去，直到堆栈溢出。

因此，将 `Int` 解释为整数集 N，将 `f :: Int -> Int` 解释为整数集上的函数，似乎是有问题的。
因为，$f(2)$并不是一个整数，它的计算不能终止。

为了描述非终止性，就需要对整数集进行扩充，
我们给整数集加上一个特殊元素“$\perp$”，称为 bottom，来表示非终止性，
而将 `f :: Int -> Int` 解释为集合 $N\cup\{\perp\}$上一个的数学函数。

像这种通过构造表达程序含义的数学对象，来对程序进行分析的方法，来自指称语义学。

指称语义中，人们会区分函数的严格性，一个函数称为严格的（strict），
如果接受一个非终止的输入表达式，函数的计算仍然不会终止，
即，$f(\perp)=\perp$。

否则，称函数为不严格的（non-strict）。

原始递归函数

我们看到在程序中使用递归，可能会导致非终止性的计算，而有些递归又不会。
这是为什么呢？

我们可以从递归函数论中找到一些线索。
递归函数论是和图灵机以及 λ 演算相等价的计算模型，它从另一个角度刻画了可计算性。
可计算性是一个有趣的话题，后续文章中，我们会详细讨论。

在递归函数论中，人们把函数划分为了 3 个层次，
原始递归函数，递归函数，和其他的不能用递归函数表示的“函数”。
这些函数集合的范围越来越大。

本文我们先介绍原始递归函数，
为此，我们需要先定义两种运算。

（1）合成运算
设 f 是 k 元部分函数，$g1,g2,\cdots,gk$ 是 k 个 n 元部分函数，令，
$h(x1,\cdots,xn)=f(g1(x1,\cdots,xn),\cdots,gk(x1,\cdots,xn))$
则称 h 是由 f 和 $g1,g2,\cdots,gk$，经过合成运算得到的。

（2）原始递归运算
设 f 是一个 n 元全函数，g 是 $n+2$ 元全函数，令，
$h(x1,\cdots,xn,0)=f(x1,\cdots,xn)$
$h(x1,\cdots,xn,t+1)=g(t,h(x1,\cdots,xn,t),x1,\cdots,xn)$
则称 h 是由 f 和 g 经过原始递归运算得到的。

于是，我们就可以定义原始递归函数了。

设初始函数包括，
（1）零函数 $n(x)=0$
（2）后继函数 $s(x)=x+1$
（3）投影函数 $uin(x1,\cdots,xn)=xi$，$i\leqslant i\leqslant n$
则由初始函数经过有限次合成运算和原始递归运算得到的函数，称为原始递归函数。

原始递归函数有以下这些性质：
由原始递归函数经过合成或原始递归得到的函数仍为原始递归函数，
因此，原始递归函数的集合在合成与原始递归运算下是封闭的。

此外，每一个原始递归函数都是全函数。
这是因为合成运算虽然是在部分函数上定义的，
但是如果 f 和 $g1,g2,\cdots,gk$ 是全函数，那么 h 也一定是全函数。

另一方面，在进行原始递归运算时，如果 f 和 g 是全函数，则 h 也一定是全函数，
这是因为原始递归运算在 h 的参数集上的定义了一个良基关系，由良基归纳法可证，h 是全函数。

总结

本文介绍了全函数与部分函数，以及计算可终止性相关的概念，
我们对程序中函数的指称，进行了定义域和值域的扩充，
随后，我们进一步了解了原始递归函数，以及它的完全性，良基归纳法起到了关键作用。

下文，我们将深入到可计算性理论，
讨论部分可计算函数和可计算函数的区别，讨论递归函数与原始递归函数的关系，
引出递归可枚举集这个重要的概念。

参考

function (mathematics)
strict function
可计算性与计算复杂性导引

五、递归集与递归可枚举集

回顾

上文中我们讨论了全函数和部分函数，以及计算的可终止性。
本文我们从数论函数开始，给原始递归函数集增加一种新的运算，得到了一个更大的集合。
然后根据递归函数，我们可以定义递归集和递归可枚举集，
为以后讨论可计算性与可判定性打好基础。

数论函数

自然数集一般记为 $N=\{0,1,2,\cdots\}$，那么 n 个自然数集的笛卡尔积记为 N^n，
于是，我们称集合 N^n到 N 的部分函数为 n 元部分数论函数。
作为数论函数，$2x$ 是一个全函数，而 $x/2$，$x-y$，$\sqrt{x}$只是部分函数，
它们的计算结果，$3/2$，$4-6$，$\sqrt{5}$都不在 N 中，于是相应定义域中的点可视为没有定义。

为什么讨论数论函数呢，其一是因为它是一个典型数学的问题，
另外一点，则是因为我们经常把其他数学问题转换成数论问题，例如，哥德尔编码。
本文中，使用数论函数，可以简化我们的描述方式。

一个谓词，指的是返回布尔值的函数，
我们还可以将谓词看做值域为$\{0,1\}$的一个数论函数。
0 代表 `True`，1 代表 `False`。

极小化算子

在前一篇中，我们从三个初始函数出发，
通过合成运算和原始递归运算，得到了原始递归函数集，
递归函数集是相对于这两种运算封闭的。

然而，这样定义的原始递归函数，并不能包括所有的数论函数，
一个典型的例子就是，阿克曼函数，

```
ackermann :: Int -> Int -> Int
ackermann 0 x = x+1
ackermann k 0 = ackermann (k-1) 1
ackermann k x = ackermann (k-1) $ ackermann k x-1
```

它并不是一个原始递归函数，（证略
因此原始递归函数集并不足以表示计算机程序中的所有函数。

为此，我们需要对原始递归函数集进行扩充，我们定义一个新的运算，称为极小化运算，
设 $P(x1,\cdots,xn,t)$是一个谓词，令
$f(x1,\cdots,xn)=min\ P(x1,\cdots,xn,t)$

$f(x1,\cdots,xn)$的值，或者是使 $P(x1,\cdots,xn,t)$为真的最小 t 值，
或者无定义，此时不存在 t 使得 $P(x1,\cdots,xn,t)$为真。
这样通过 min 得到 $f(x1,\cdots,xn)$的过程称为极小化运算，
也称部分函数 $f(x1,\cdots,xn)$是由谓词经过极小化运算得到的。

以上我们给谓词定义了极小化运算，现在我们将极小化运算推广到一般的函数上面，
设 $g(x1,\cdots,xn,t)$是一个 $n+1$ 元函数，令
$f(x1,\cdots,xn)=min\{g(x1,\cdots,xn,t)=0\}$
则称部分函数 $f(x1,\cdots,xn)$是由函数 $g(x1,\cdots,xn,t)$经过极小化运算得到的。

递归函数集

和定义原始递归函数集一样，我们从以下三个初始函数出发，
（1）零函数 $n(x)=0$
（2）后继函数 $s(x)=x+1$
（3）投影函数 $uin(x1,\cdots,xn)=xi$，$i\leqslant i\leqslant n$
由初始函数，经过有限次合成运算，原始递归运算，以及极小化运算，得到的函数称为递归函数。

递归函数并不一定是全函数，因为极小化运算可能会导致结果函数在某些点无定义，
递归的部分函数称为部分递归函数。

可以证明阿克曼函数是递归函数，但不是原始递归函数，
因此，原始递归函数集是递归函数集的真子集。

递归可枚举集

在具体实践中，我们经常会遇到这样的问题，

给定一个元素，我们需要判断这个元素是否属于某个集合。
这种问题，称为集合的成员资格问题。

沿用这一思路，我们可以使用一个谓词 χB来定义相应的集合 $B \subseteq N$，
$B=\{x \in N \mid \chi B(x)\}$
谓词 $\chi B(x)$为真，则 $x \in B$。
这个谓词 $\chi B(x)$，通常称为集合 B 的特征函数。

如果特征函数 χB是第一个递归的全函数，
则我们总是可以判断 $\chi B(x)$等于 0 还是 1，
这样的集合 B 称为递归集。

如果存在部分递归函数 g，使得 $B=\{x \in N \mid g(x)\downarrow\}$，
即，$x \in B$ 当且仅当 g 在 x 处有定义，
则称集合 B 是一个递归可枚举集。

因此，对于每一个自然数 $x \in N$，
我们总是可以通过递归集 B 的特征函数 χB，来判断 x 是否 B 的成员。
而对于递归可枚举集，就不容乐观了，
如果某个自然数 $x \in N$ 是 B 的成员，那么我们可以断定这件事，因为 $g(x)$有定义，
但是如果某个自然数 $y \in N$ 不是 B 的成员，我们就不能确定，因为这时候 $g(x)$无定义。
（$g(x)$无定义，则它对应的图灵机不停机，后文我们详细讨论

因此，集合 B 是递归的当且仅当 B 和B^-是递归可枚举的，
其中B^-为 B 的补集。

总结

本文介绍了数论函数，递归函数集，然后用递归函数分别定义了递归集和递归可枚举集，
可是为什么递归可枚举集是“可枚举”的呢？

是因为每一个递归可枚举集可以一一对应一个自然数，这是怎样做到的呢？
这需要我们理解总共有多少个可能的程序，以及什么是通用程序。

参考

递归集
递归可枚举集

六、最多有多少个程序

回顾

上一篇中，我们通过引入极小化算子定义了递归函数，
使用递归函数，我们又定义了递归集与递归可枚举集，
本文我们要讨论，为什么递归可枚举集是“可枚举”的，以及什么是可计算函数。

可计算性

我们听说过，现代计算机在计算能力上是与图灵机等价的，
什么叫做计算能力呢?
它指的是图灵机可计算的函数集，与现代计算机可计算的函数集是相等的。

为了简单起见，我们不去讨论图灵机，而是从现代计算机直接说起，
设 P 是一段程序，n 是一个正整数，
我们称数论函数 $\psi(x1,x2,\cdots,xn)$为程序 P 所计算的 n 元部分函数，
如果对于相同的输入，
要么：（1）程序 P 的计算可以终止，此时计算结果等于 $\psi(x1,x2,\cdots,xn)$的相应函数值；
要么：（2）程序 P 的计算不能终止，此时 $\psi(x1,x2,\cdots,xn)$无定义。

设 $f(x1,x2,\cdots,xn)$是一个部分函数，如果存在程序 P 可计算 f，则称 f 是部分可计算的。
如果一个函数，既是部分可计算的，又是全函数，则称这个函数是可计算的。

可以证明，所有的原始递归函数和递归函数都是部分可计算的。

通用程序

我们使用现代计算机进行编程的时候，并不是直接把程序的输入传给程序，
而是将程序本身以及它的输入，传给计算机，最后由计算机得到计算结果，
像这种接受任何程序和它的输入作为自己的输入，返回程序执行结果的程序，称为通用程序。
为此，通用程序需要把输入的程序进行编码。

常用的编码方法，涉及配对函数和哥德尔编码。
为了不引入太多的复杂性，我们可以将程序的编码理解为存储程序的二进制数据，

不同的程序会有不同的二进制表示，每一个二进制表示可以对应一段程序（虽然可能不合法）。
哥德尔编码做的事情就是将程序和自然数集一一对应起来。

因此，所有程序的个数是可数的，而这些程序可计算的函数个数也一定是可数的，
它们可能是全函数，也可能是部分函数。
（其中，“可数”指的是可数集，可数集是与自然数集之间存在一一映射的集合。

然而，自然数集上的函数全体并不可数，（证略
所以肯定存在程序不可计算的函数。

集合个数的可枚举性

程序 P 所计算的函数，我们可以记为 $\psi(x1,x2,\cdots,xn)$，
由此，我们可以定义通用程序 Φ，则有，
$\Phi(x1,x2,\cdots,xn,y)=\psi(x1,x2,\cdots,xn)$
其中，y 是程序 P 的编码。

因为，所有的程序与自然数集一一对应，
所以，$\Phi(x1,x2,\cdots,xn,0),\Phi(x1,x2,\cdots,xn,1),\cdots$
枚举了所有的 n 元可计算函数。

我们定义 $Wy=\{x\in N|\Phi(x,y)\downarrow\}$，
根据递归可枚举集的定义，每一个 Wy是一个递归可枚举集，
又因为 $\Phi(x,0),\Phi(x,1),\cdots$枚举了所有的可计算函数，
所以，$W0,W1,\cdots$枚举了所有的递归可枚举集。

因此，集合 B 是递归可枚举的，当且仅当存在 $y\in N$，使得 $B=Wy$，
称为枚举定理，这就是“枚举”的含义。

令 $K=\{n\in N|n\in Wn\}$，
则 K 是递归可枚举的，但不是递归的，（证略
因此，K^-不是递归可枚举的，否则 K 就是递归集了。
（根据，集合 B 是递归的当且仅当 B 和B^-是递归可枚举的，见上一篇

因此，我们找到了一个非递归的递归可枚举集 K，

以及一个非递归可枚举集K^-。

停机问题

任给一个程序和一个自然数，问该程序对这个自然数输入的计算是否停止，
这个问题称为停机问题。

我们可以用谓词 $H(x,y)$描述这个问题，
$H(x,y)$，表示以 y 为代码的程序对输入 x 的计算最终停止。
那么，$H(x,y)$是不可计算的，即，不存在一个程序来计算 $H(x,y)$。

我们来证明一下，假设有一个程序可以计算 $H(x,y)$，
那么我们就能用它来构造一个新程序 P，它的输入是 x，
这段程序当 $H(x,x)$为真时，计算不停止，而当 $H(x,x)$为假时，计算停止。

程序 P 也可以进行编码，假设为 $y0$，现在我们来判断 $H(y0,y0)$。

如果 $H(y0,y0)$为真，意味着编码为 $y0$的程序以 $y0$作为输入最终停止，
即程序 P，输入为 $y0$时，最终停止，
可是根据 P 的定义，此时 $H(x,x)=H(y0,y0)$为假才会停止，矛盾。

如果 $H(y0,y0)$为假，意味着编码为 $y0$的程序以 $y0$作为参数最终不会停止，
即程序 P，输入为 $y0$时，最终不停止，
可是根据 P 的定义，此时 $H(x,x)=H(y0,y0)$为真才不会停止，矛盾。

$H(y0,y0)$不能为真也不能为假，矛盾，
因此，计算 $H(x,x)$的程序不存在，我们也无法用它来构造程序 P。

可判定性

可判定性问题，指的是一个询问真或者假的问题是否可以被回答。
如果我们总能回答出这个问题是真或者是假，就称该问题是可判定的，
如果我们只能当问题为真的时候确定为真，为假的时候所进行的计算可能不会终止，那么就称该问题是半可判定的。

某元素是否属于一个递归集，是可判定的，
某元素是否属于一个递归可枚举集，是半可判定的。

因为，递归集使用一个递归的全函数定义的，
而递归可枚举集是使用第一个部分递归函数定义的，
我们无法判断某个部分递归函数，在接受某参数时，是没有定义，还是计算尚未停止。
即，判断元素是否属于某递归可枚举集的程序可能永不停机。

总结

本文介绍了函数的可计算性，通用程序，以及最多有多少个程序，
还了解了停机问题和可判定性问题。

这些都是可计算性理论的基础，我们清晰的看到了人类的计算能力，
以及用递归所能计算的函数范围，后文中我们开始讨论不动点理论，
这同样是一个有趣的话题。

附

配对函数和哥德尔数，是对数偶和有穷数列的一种编码方式。

（1）配对函数
令$\langle x,y\rangle=2x(2y+1)-1$，称$\langle x,y\rangle$为配对函数，它是一个原始递归函数。
任给一个数 z，存在唯一的一对数 x 和 y，使得$\langle x,y\rangle=z$。
x 是 $z+1$ 含有因子 2 的个数，即使得 $2^t|(z+1)$的 t 的最大值。
$(z+1)/2x$必为奇数，y 是 $2y+1=(z+1)/2x$的唯一解。
一般的，记 $l(z)=x$，$r(z)=y$，则 $l(z)$和 $r(z)$也是原始递归函数。

（2）哥德尔数
记$[a1,a2,\cdots,an]=\prod i=1npiai$，
$[a1,a2,\cdots,an]$称为有穷数列$(a1,a2,\cdots,an)$的哥德尔数。
其中，pi是第 i 个素数。

例如，$[2,0,1,3]=2^2\cdot3^0\cdot5^1\cdot7^3=6860$。
对于每一个固定的 n，$[a1,a2,\cdots,an]$是原始递归函数，并且这种编码具有唯一性。

参考

配对函数
哥德尔数
可数集
可判定性
康托尔定理

七、不动点算子

回顾

以上几篇文章中，我们讨论了可计算性理论相关的一些内容，
可计算性与递归函数论存在着千丝万缕的联系，
不动点理论也是这样的，我们定义的递归函数一定存在吗？
在什么情况下它是存在的？

要回答以上这些问题，还要从方程，不动点，不动点算子说起。

约束方程

在中学时代，我们学过“方程”的概念，
方程可以简单表述为含有未知数的等式，例如，$3x+3=2$。
未知数可以同时出现在等式的两边，例如，$2x+3=2-x$。
通过合并同类项，我们可以求解 x。

在大学时代，我们还学过线性方程组和微分方程，
例如，求解矩阵的特征值和特征向量，$Av=\lambda v$
二阶常微分方程（贝塞尔方程），$x2y''+xy'+(x2-\alpha 2)y=0$。

在计算机科学中，同样的未知“数”的思想，
还出现在了类型推导（例如：unification）与递归函数的定义中。

以上这些例子，方程是“约束”的一种表现形式。

我们回到最简单的阶乘函数 `fact` 的定义式，

```
fact :: Int -> Int
fact 1 = 1
fact n = n * fact (n-1)
```

去掉语法糖，稍微修改一下，

```
fact :: Int -> Int
fact n = case n of
  1 -> 1
  n -> n * fact (n-1)
```

我们发现，`fact` 的定义和“方程”十分相似，`fact` 同时出现在了等式的两边，
阶乘函数，就是这个“方程”的“解”。

函数的不动点

在中学数学中，我们已经学过不动点了，只是当时印象不是那么深刻，
函数的不动点，是指被这个函数映射到其自身的那些点。
例如：$f(x)=x2-3x+4$，
则 2 是函数 f 的一个不动点，$f(2)=2$。

并不是每一个函数都有不动点，
例如，实数域上的函数 $f(x)=x+1$，就没有不动点，对于任意实数 x，永远都不等于 $x+1$。
（不动点是和定义域有关的，以后我们还会重新讨论 $f(x)=x+1$ 的不动点。

一般的，函数 $f(x)$的不动点，指的是这样的 x，使得 $x=f(x)$。

重新温习了不动点相关的知识之后，
我们就可以对上面的阶乘函数进行改造了，
我们要把阶乘函数看做另外一个函数的不动点。

定义函数 g，

```
g :: (Int -> Int) -> Int -> Int
g f n = case n of
  1 -> 1
  n -> n * f (n-1)
```

我们可以得到，`g fact = fact`，
因此，`fact` 实际上就是函数 `g` 的不动点。

于是，在“方程”中求解 `fact` 的过程，
就转换成了求解函数 `g` 的不动点的过程了。

不动点算子

我们怎样求解函数 `g` 的不动点呢?
在 Haskell 中，可以很方便的定义一个高阶函数 `fix`，它可以用来求解任意函数的不动点，

```
fix :: (a -> a) -> a
fix g = let x = g x in x
```

我们试验一下 `fix` 的强大威力，

```
fact 10
> 3628800
fix g 10
> 3628800
```

注意，`fix g` 得到了 `g` 的不动点，即`(fix g) = g (fix g)`。

有了 `fix`，我们就可以构造匿名递归函数了，

```
fact' :: Int -> Int
fact' = fix $ \fact -> \n -> case n of
  1 -> 1
  n -> n * fact (n-1)
```

`fix` 后面跟的函数没有名字，它是匿名的，但是经过 `fix` 作用后，可以产生一个递归函数。
也就是说，为了实现递归，函数是可以没有名字的。

Y 组合子

Y 组合子，是 Haskell B. Curry 在研究 λ 演算时发现的，
它的表现形式如下，
$Y := \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$

在 λ 演算中，（α 转换和 β 规约
我们可以证明，对于任何函数 `g`，$(Yg)=(g(Yg))$。

因此，Y 组合子是一个不动点算子，它可以得到任意函数的不动点。
其他的不动点组合子还有图灵不动点组合子 Θ，
$\Theta := (\lambda x.\lambda y.(y\ (x\ x\ y)))\ (\lambda x.\lambda y.(y\ (x\ x\ y)))$

讨论 Y 组合子在 Haskell 中的表示方式是有趣的，因为直接翻译过去会报类型错误，

```
y :: (a -> a) -> a
y = \f -> (\x -> f (x x)) (\x -> f (x x))
-- Occurs check: cannot construct the infinite type: r0 ~ r0 -> a
```

```
4
  -- Expected type: r0 -> a
5
  --   Actual type: (r0 -> a) -> a
6
  -- In the first argument of ‘x’, namely ‘x’
7
  -- In the first argument of ‘f’, namely ‘(x x)’
8
```

类型系统无法确定 `x` 的类型。

问题出在表达式 `x x` 上面，
假设 `x x` 的类型为 `a`，则第一个 `x` 的类型就应该为`? -> a`，
于是，第二个 `x` 的类型肯定也应该是`? -> a`。（因为都是 `x`

又因为 `x x` 的类型为 `a`，
所以第一个 `x` 的类型`? -> a` 中，`?`的类型就应该是`? -> a`，
（因为`((? -> a) -> a)`作用到`(? -> a)`才能得到 `a`

`?`的类型是`? -> a`，因此`?`应该是一个递归类型。

下面我们来定义递归类型 `Mu`，来帮助编译器进行恰当的类型推导，

```
1
  newtype Mu a = Mu (Mu a -> a)
2
  y :: (a -> a) -> a
3
  y f = (\h -> h <span data-katex=" Mu h) (\x -> f . (\(Mu g) -> g) x "></span> x)
4
```

最后，`fact'`就可以使用 Y 组合子来定义了。

```
1fact' :: Int -> Int
2fact' = y $ \fact -> \n -> case n of
3   1 -> 1
4   n -> n * fact (n-1)
```

总结

本文从简单的“方程”思想出发，引出了不动点的概念，
然后把递归函数看做了另外一个函数的不动点，
最后，我们讨论了 Y 组合子这样一个具体的不动点算子。

可是，这里隐藏着一个问题，我们看到 `fix` 是可以求解任意函数的不动点的，
而对于以下递归函数 `succ`，即 $f(x)=x+1$，

```
succ :: Int -> Int
succ n = n+1
```
在实数域上是显然没有不动点的。

那么 `fix succ` 是什么呢？
这个问题，我们将在后文中继续讨论。

参考

方程
特征值和特征向量
微分方程
不动点
不动点组合子
Haskell/Fix and recursion
Y Combinator in Haskell

八、偏序结构

回顾

上一篇我们介绍了不动点算子和 Y 组合子，以及 Y 组合子的具体表现形式，
这一篇我们根据不动点算子的性质来证明 `fact` 函数就是 `g` 函数的不动点。
随后，我们回归到了数学中，讨论集合上的一种偏序结构，
这为下文完全偏序集，以及完全偏序集上连续函数的不动点定理做好准备。

不动点算子的性质

上文我们介绍了不动点算子 `fix`，
它可以用来求取任意函数的不动点。

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

并且我们说以下函数的不动点为 `fact = fix g`，

```
g :: (Int -> Int) -> Int -> Int
g f n = case n of
  1 -> 1
  _ -> n * f (n-1)
```

但是上文中，我们只是对它们的计算结果进行比对，
并没有对它进行证明。

考虑到 `fix` 的性质，`fix g = g (fix g)`，
（因为 `fix g` 是 `g` 的不动点，令 `h = fix g`，上式为 `h = g h`
我们可以使用数学归纳法，证明对于任意的自然数 n，`fact n = fix g n`。

我们先证初始条件，

```
fix g 1
= g (fix g) 1
= case 1 of
    1 -> 1
    _ -> ...
= 1

```

```
  = fact 1
```

然后再证递推条件，假设 `fact k = fix g k`，
我们要推出 `fact (k+1) = fix g (k+1)`，其中，$k>0$。

```
fix g (k+1)
= g (fix g) (k+1)
= case (k+1) of
    1 -> 1
    _ -> (k+1) * (fix g) k
= (k+1) * (fix g) k
= (k+1) * fact k
= fact (k+1)
```

因此，对于任意的自然数 n，`fact n = fix g n`。
即，`fact = fix g`。

不动点算子的有限展开

根据上一节 `fact = fix g` 的证明，我们看到，
每一步递推，我们都使用了不动点算子 `fix` 的性质 `fix g = g (fix g)`，
但是对于一个具有有限存储空间的机器来说，递推的步骤不可能是无限的，
为了界定最多使用多少次递推，我们定义，$fix[n+1]\ g=g\ (fix[n]g)$。
并且认为 $fix[0]\ g$ 对于任意的 n 无定义。

因此，$fix[1]\ g\ 1=1$，而 $fix[1]\ g\ n$ 在 $n>1$ 时没有定义。
$fix[2]\ g\ 1=1$，$fix[2]\ g\ 2=2$，而 $fix[2]\ g\ n$ 在 $n>2$ 时没有定义。

所以，$fix[n]\ g$ 是一个部分函数，且，
$fix[n+1]\ g$ 所表示的函数，总是比 $fix[n]\ g$ 的计算能力更强一些，离 `fact` 更近一些。
当 $n\to\infty$时，$fix[\infty]\ g$ 就是阶乘函数 `fact`。

即，$\{fix[n]g|\ n\geqslant 0\}$的最小上界，就是 `g` 的不动点。
那么，什么样的 `g` 才能保证这个集合具有最小上界呢？
序理论指出，完全偏序集上的序保持自映射具有最小不动点。

为此，我们需要先认识什么是偏序集，什么是连续函数。

使用完全偏序集上的连续函数解释程序中函数的方式，称为域论模型。

偏序集与哈斯图

在第三篇中，我们讨论过偏序关系，
一个偏序集$(D,\leqslant)$是一个集合D，
并且在这个集合上定义了一个偏序关系$\leqslant$。

设A为实数集的一个非空子集，我们定义A上的偏序关系为$\leqslant$，
$x\leqslant y$当且仅当x是小或等于y的实数。
则，$(A,\leqslant)$是一个偏序集。

偏序集反映了集合上的一种偏序结构，它比我们想象中的更为常见，
例如，一个集合A，对于任意两个元素$x,y\in A$，我们定义$x\leqslant y$当且仅当$x=y$。
那么$(A,\leqslant)$是一个偏序集。
因此，如果某个集合构成了一个偏序集，这完全取决于我们怎样定义偏序关系。

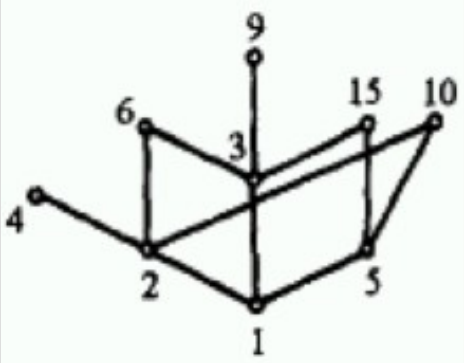
设$(D,\leqslant)$是一个偏序集，
对于任意的$x,y\in D$，如果总是有$x\leqslant y$或者$y\leqslant x$成立，
则称x和y是可比的。

$x\leqslant y$，且$x\neq y$，则记为$x<y$。
如果x和y是可比的，且$x<y$，如果不存在$z\in D$，使得$x<z<y$，则称y覆盖x。

根据可比性和覆盖性，我们就可以将偏序关系用无向图表示出来了，
其中，顶点表示元素，边表示覆盖关系，并且省去图中每个顶点处的环，
y覆盖x就将代表y的顶点放在代表x的顶点之上，并在x和y之间连线，
如果$x<y$，但是y不覆盖x，就省掉x与y之间的连线。

这样用来表示有限偏序集的无向图，称为哈斯图。

例如，易证整除关系是整数集上的一种偏序关系，
我们可以画出偏序集{1,2,3,4,5,6,9,10,15}对应的哈斯图，如下，



全序集与拟序集

设$(D,\leqslant)$是一个偏序集，如果对于任意 $x,y\in D$，x 和 y 都可比，
则称$\leqslant$为 D 上的全序关系，此时称$(D,\leqslant)$为全序集。

可见，实数集以及实数集上的小于等于关系$\leqslant$，构成了一个全序集。
哈斯图为从下至上的“一条线”，是全序集的充要条件。

设 R 是非空集合 A 上的，反自反的和传递的二元关系，
则称 R 为 A 上的拟序关系，常将拟序关系记为$<$，并称$(A,<)$为拟序集。
拟序关系自然具有反对称性。
其中，反自反关系，指的是不存在 $x\in A$，使得 $x<x$。

拟序关系与偏序关系的哈斯图在画法上完全相同，只是拟序关系的哈斯图的各顶点都没有环。

设$(A,<)$是一个拟序集，如果对于任意的 $x,y\in A$，
$x<y$，$x=y$，$y<x$ 三式有且仅有一式成立，则称$<$具有三歧性，
这样的拟序关系$<$，称为拟全序关系，这样的拟序集$(A,<)$，称为拟全序集。
拟全序集的哈斯图也是“一条线”。

最小元与上确界

对于偏序集$(D,\leqslant)$，以及它的一个子集 $S\subseteq D$，
如果存在 $y\in S$，且对于任意的 $x\in S$，有 $y\leqslant x$，
则称 y 为 S 的最小元。（相似的我们可以定义最大元

如果存在 $y \in S$，对于任意的 $x \in S$，
如果 $x \leqslant y$ 那么就有 $x=y$，
则称 y 为 S 的极小元。（相似的我们可以定义极大元

如果 S 是有穷集，则 S 的极小元一定存在，并且可能有多个，
但是最小元却不一定存在。

上文中，我们画出了偏序集 $A=\{1,2,3,4,5,6,9,10,15\}$对应的哈斯图，
我们取 $B1=\{1,2,3\}$，$B1=\{3,5,15\}$，$B3=A$，
则 1 是 $B1$的最小元，也是极小元，2,3 是 $B1$的极大元，但 $B1$没有最大元。
3,5 是 $B2$的极小元，但 $B2$没有最小元，15 是 $B2$的最大元，也是极大元。
1 是 $B3$的最小元，也是极小元，4,6,9,10,15 是 $B3$的极大元，但是 $B3$没有最大元。

对于偏序集$(D, \leqslant)$，以及它的一个子集 $S \subseteq D$，
如果存在 $y \in D$，（注意，不一定是 $y \in S$
使得对于任意的 $x \in S$，$x \leqslant y$，则称 y 为 S 的上界，
如果 S 的所有上界存在最小元，则称它为 S 最小上界，或上确界。
（相似的可以定义下确界

S 的上界和下界不一定存在，即使存在，上确界和下确界也不一定存在。

设$(A,<)$是一个拟全序集，如果对于 A 中的任何非空子集 S 都有最小元，
则称$<$是一个良序关系，$(A,<)$是一个良序集。
例如，自然数集以及自然数集上的小于关系，构成了一个良序集$(N,<)$，
但是，整数集以及整数集上的小于关系，并不构成良序集，而仅仅是一个拟全序集。

总结

本文从一个证明出发，我们了解了不动点算子的工作原理，
然后引出了一些数学概念，序关系在不动点算子理论中占有很重要的地位，
所以，这里给出了详细的介绍，下文我们开始讨论最小不动点定理。

参考

序理论
域论模型
偏序关系
离散数学教程

九、最小不动点定理

回顾

上文我们讨论了集合上的偏序结构，之所以谈论它们是因为，
完全偏序集上的连续函数具有最小不动点，这称之为最小不动点定理，
除了集合论的一些知识之外，我们还要讨论到底什么是连续函数，以及什么是完全偏序集。

有向子集与完全偏序

偏序集$(D,\leqslant)$的非空子集$S\subseteq D$叫做有向子集（directed subset），
当且仅当，对于S中的任意元素$a,b\in S$，
存在S中的一个元素c，有$a\leqslant c$且$b\leqslant c$。

如果一个偏序集$(D,\leqslant)$的每个有向子集$S\subseteq D$都有上确界（记为$\bigvee S$）
就称它是一个有向完全偏序集，
此外，如果它还有最小元，就称它是一个完全偏序集。

注意，完全偏序集并不是每一个子集都有上确界，
而是它的每一个有向子集都有上确界。

连续函数

假设$(D,\leqslant)$与$(E,\leqslant)$是完全偏序集，
$f:D\to E$是集合上定义的一个函数，
如果，$S\subseteq D$，则$f(S)$为E的子集，
其中$f(S)=\{f(d)\mid d\in S\}$。

对于任意的$d,d'\in D$，如果$d\leqslant d'$就有$f(d)\leqslant f(d')$，
我们就说f是单调的。
可以看出，如果f是单调的，且S是D的有向子集，那么$f(S)$也是E的有向子集。

如果f是单调的，且对于任意有向子集$S\subseteq D$，
有$f(\bigvee S)=\bigvee f(S)$，就称$f$是连续的。

连续函数集上的偏序结构

完全偏序集$(D,\leqslant)$和$(E,\leqslant)$上的连续也可以定义偏序结构，
我们称$f\leqslant g$，当且仅当对于每一个$d\in D$，我们有$f(d)\leqslant g(d)$。
这样我们就得到了一个元素为连续函数的偏序集，$(D\to E,\leqslant)$。
可以证明，$(D\to E,\leqslant)$也是一个完全偏序集。（证略

最小不动点定理

如果$(D,\leqslant)$是一个完全偏序集，且$f:D\to D$是连续的，
则f有最小不动点，$fix D\ f=\bigvee\{fn(\bot)|\ n\geqslant 0\}$。

因为$\bot$是D中的最小元，且$f(\bot)\in D$，所以，$\bot\leqslant f(\bot)$，
因为f是连续的，所以f也一定是单调的，所以，$f(\bot)\leqslant f2(\bot)$，
继而，$fn(\bot)\leqslant fn+1(\bot)$，对于任意的$n\geqslant 0$都成立。

$\{fn(\bot)|\ n\geqslant 0\}$构成了一个全序集，
所以，它是完全偏序集$(D\to E,\leqslant)$的一个有向子集，必有上确界。

设a是上确界，$a=\bigvee\{fn(\bot)|\ n\geqslant 0\}$，
首先a是f的不动点，因为，由f的连续性，
$f(a)=f(\bigvee\{fn(\bot)|\ n\geqslant 0\})$
$f(a)=\bigvee\{f(n+1)(\bot)|\ n\geqslant 0\}$，
但是由于$\{fn(\bot)\}$与$\{f(n+1)(\bot)\}$，有同样的上确界，
所以，$f(a)=a$。

其次，a是f的最小不动点，
假设$b=f(b)$是f的任意不动点，因为$\bot\leqslant b$，所以$f(\bot)\leqslant f(b)$，
类似的，对于任意的$n\geqslant 0$，$fn(\bot)\leqslant fn(b)$。
但是，由于b是f的不动点，所以$fn(b)=b$，
因此b是集合$\{fn(\bot)|\ n\geqslant 0\}$的上界。
由于集合的最小上界是a，所以有$a\leqslant b$。

后继函数的不动点

```
succ :: Int -> Int
succ n = n+1
```

在第七篇中，我们说 `fix` 可以得到任意函数的不动点，
那么这个 `succ` 函数呢？它有没有不动点？
`fix succ` 是什么？

现在我们有了最小不动点定理，
就得验证 `succ` 所指称的数学函数，是不是一个完全偏序集上的连续函数，
如果是的话，它就有不动点。

在第四篇为了表示计算的不可终止性，我们对自然数集进行了扩充，$N \cup \{\bot\}$，
然后用这个集合作为程序中 `Int` 类型的值的解释。

然而，$N \cup \{\bot\}$不是一个完全偏序集，原因是它没有上界，
如果我们额外加入一个比其他的自然都大的元素$+\infty$，
则我们就得到了一个完全偏序集，$N \cup \{\bot\} \cup \{+\infty\}$。

然后，我们看 `succ` 连续吗？
首先，它是单调的，如果我们再定义 $succ(+\infty)=+\infty$，
就有 $succ(\bigvee N)=\bigvee succ(N)$，
因此，`succ` 是一个连续函数。

根据最小不动点定理，`succ` 的最小不动点是，$fix\ succ=\bigvee\{succn(\bot) \mid n \geqslant 0\}$。

由于 $succ(\bot)=\bot$，即对于非终止的输入 `succ` 的计算也不会终止，
所以 $succn(\bot)=\bot$，
因此，$fix\ succ=\bigvee\{succn(\bot) \mid n \geqslant 0\}=\bot$，
即，`succ` 的最小不动点是⊥，`fix succ` 的计算不会终止。

求解阶乘函数

上一篇中，我们证明了阶乘函数 `fact` 是以下函数的不动点，`fact = fix g`，

```
g :: (Int -> Int) -> Int -> Int
g f n = case n of
```

```
  1 -> 1
  _ -> n * f (n-1)
```

现在我们来看一下，如何运用最小不动点定理来得到这个答案。
为了避免篇幅过长，关于 g 函数的连续性证明暂略，
我们直接使用公式，
$fix\ g=\bigvee\{gn(\bot)|\ n\geqslant 0\}$，
即，g 函数的最小不动点，就是集合 $D=\{gn(\bot)|\ n\geqslant 0\}$ 的上确界。

我们先来看一下这个集合 D 中有哪些元素，
其中，$g(\bot)\in D$，我们将 $\bot$ 传入 g 中，看看会得到什么，

```
f1 = \n -> case n of
  1 -> 1
  _ -> ...
```

这个函数能计算 f1 1，但是不能计算 f1 2，恰好是 fact 函数有限展开的一阶项。

我们再来看 $g2(\bot)\in D$，它等于 $g(f1)$，

```
f2 = \n -> case n of
  1 -> 1
  _ -> n * f1 (n-1)
```

这个函数能计算 f2 1 以及 f2 2，但是不能计算 f2 3，恰好是 fact 函数展开的二阶项。

由此，我们看出了规律，
$gn(\bot)\in D$ 就是 fact 函数有限展开的第 n 阶项。
上一篇中，我们已经证明了，当 $n\to\infty$ 时，它就是 fact 函数，
考虑到 $f1,f2,\cdots$ 这些函数的序关系，
因此，我们有 $fact=\bigvee\{gn(\bot)|\ n\geqslant 0\}$。

即，fact 函数就是 g 函数的最小不动点。

总结

到此为止，我想这个系列的文章已经讨论完了，
本系列文章一直围绕着递归函数和不动点进行分析，
在内容上可以分为两个部分，前半部分主要与可计算性理论有关，
后半部分与不动点定理有关，希望对大家有所帮助。

参考

有向集合
完全偏序
Kleene fixed-point theorem
Foundations for Programming Languages