Atitit **编程范式总结**

目录

[1.1.1. IP(Imperative Programming) 1](#_Toc4762)

[1.1.2. SP(Structured Programming) 2](#_Toc19176)

[1.1.3. PP(Procedure Programming) 3](#_Toc17567)

[1.1.4. DP(Declarative Programming) 4](#_Toc24582)

[1.1.5. LP(Logic Programming) 5](#_Toc25309)

[1.1.6. FP(Functional Programming) 即函数式编程，也是 DP 的子集, 5](#_Toc20259)

[1.1.7. FRP(Functional Reactive Programming) 6](#_Toc4128)

[1.1.8. MP(Meta Programming) 6](#_Toc20269)

编程范式(programming paradigms)，是在编程的理论与实践当中提炼出的概念模型。

Programming Paradigm: A conceptual model underlying the theories and practice of programming

### **IP(Imperative Programming)**

即指令式编程。程序由一系列指令和流程控制语句组成，运行过程中指令不断改变程序的状态，由此达到最终的编码意图。 ****IP**** 范式会显式地指定代码的执行流程，以及运算逻辑。汇编是典型的使用 ****IP**** 范式的编程语言。

result = []

i = 0

start:

numPeople = length(people)

if i >= numPeople goto finished

p = people[i]

nameLength = length(p.name)

if nameLength <= 5 goto nextOne

upperName = toUpper(p.name)

addToList(result, upperName)

nextOne:

i = i + 1

goto start

finished:

return sort(result)

### **SP(Structured Programming)**

即结构化编程，在 ****IP**** 的基础上，我们可以将用 ****goto**** 来控制流程的代码，以 ****for**** 语句， ****while****语句等此类结构化的代码块(block structure)组织起来，使得代码的可读性更高，那么此种编码方式即为结构化范式。 ****SP**** 是现代语言都支持的一种基础范式。

result = [];

for i = 0; i < length(people); i++ {

p = people[i];

if length(p.name)) > 5 {

addToList(result, toUpper(p.name));

}

}

return sort(result);

### **PP(Procedure Programming)**

即过程式编程，单看中文可能难以理解， *procedure* 来源于 *procedure call* , 即函数调用，主要是因为 ****PP**** 在 ****IP**** 的基础上引入了函数及函数调用, 将可提炼的逻辑用函数封装起来，以复用代码和提高可读性。

OOP(Object-oriented Programming)

即我们常说的面向对象编程。在 ****SP**** 和 ****PP**** 的范畴里，数据类型是松散的，数据结构和算法之间也是松散的，而 ****OOP**** 则提供了一种类似于人类对现实世界建模的方法，对二进制世界的类型和逻辑进行建模和封装，并在此基础上提供了更多的类型和语法特性。 ****OOP**** 的优点简列如下(封装，继承，多态):

* 当我们对一组数据类型进行抽象，封装成类(class, 类是 ****OOP**** 的基本概念)时，我们可以定义该类的子类，来共享它的数据类型和逻辑，此方式称为继承(inheritance), 能够有效减少我们的开发时间。
* 当类被定义后，通常它只需要关注它自身的数据和逻辑，通过语法特性，一般是 *public* / *private* 关键字，将这类数据和逻辑隐藏起来，避免被非法(或者说不合理的, 不应当的)访问，提升程序和系统的安全性。
* 一个封装好的类，不仅能被它的创建者使用，也可以分发(在网络上)给其他人使用, 比如 *Java*的jar包。
* 一门语言不可能把开发者所需要的所有的类型都定义好，class的概念则很好地解决了这个问题，开发者可以定义任意自己想要的数据类型。
* ****OOP**** 的多态性质可以让代码更加灵活。

### **DP(Declarative Programming)**

即，描述性范式。和 ****IP**** 相反，此类语言只描述最终的编码意图，不描述达到意图的过程。举个例子，如何用程序来回答你是怎么回家的？

*IP* :

Go out of the north exit of the parking lot and take a left. Get on I-15 south until you get to the Bangerter Highway exit. Take a right off the exit like you’re going to Ikea. Go straight and take a right at the first light. Continue through the next light then take your next left. My house is #298.

*DP* :

My address is 298 West Immutable Alley, Draper Utah 84020

典型的 ****DP**** 范式语言如 *SQL* , 仅描述目的，达到目的的逻辑被隐藏。

SELECT \* FROM Users WHERE Country=’Mexico’;

### **LP(Logic Programming)**

即逻辑编程，它属于 ****DP**** 的范畴。逻辑编程的要点是将数学中的逻辑风格带入计算机程序设计之中。它设置匹配规则来解决问题(rule-based)，而非设置步骤来解决问题, 即事实+规则=结果。*Prolog* 是典型的 ****LP**** 范式语言，此类语言主要应用在人工智能，专家系统等领域。

### **FP(Functional Programming)** 即函数式编程，也是 ****DP**** 的子集,

即函数式编程，也是 ****DP**** 的子集, 在函数式编程里，所有的计算都是通过函数调用完成的，函数里的 ****SP**** 逻辑尤其是控制流逻辑，被隐藏了起来. 假设我们要编写一个函数，将一个数组的每个元素都乘以2， ****PP**** 风格的代码如下:

// TypeScript

function double (arr) {

let results = []

for (let i = 0; i < arr.length; i++){

results.push(arr[i] \* 2)

}

return results

}

上述代码，详细地写明了整个计算过程，包括迭代过程和计算方法。所以 ****IP**** 范畴的范式会详细描述计算机是如何完成这件事的，有篇文章是这么描述 ****IP**** 的

First do this, then do that.

FP则不会描述数组是如何迭代的，也不会显式地修改变量, 仅仅描述了我们想要什么，我们想要将元素乘以2.

function double (arr) {

return arr.map((item) => item \* 2)

}

FP将开发者从机器的执行模型切换到了人的思维模型上，可读性会更高。需要注意的是，某些支持 ****FP**** 的语言本身是属于 ****IP**** 的，同时也可以认为其属于 ****DP**** , 不必过于纠结。

### **FRP(Functional Reactive Programming)**

即，函数式响应型编程。

### **MP(Meta Programming)**

即元编程, 也写做 *Metaprogramming* 。元编程是一种可以将程序当作数据来操作的技术，元编程能够读取，生成，分析或转换其他的程序代码，甚至可以在运行时修改自身. *C++* 的template即属于 *meta programming* 的范畴，编译器在编译时生成具体的源代码。在web框架 *Ruby on Rails*里，元编程被普遍使用。比如，在SQL数据库的表里，有一个表users，在ruby中用类 *User* 表示，你需要根据user的email字段来获取相应的结果，通常的做法是写个sql查询语句去完成，但是*Ruby on Rails* 在元编程的加持下，会让这件事变得异常简单。

User.find\_by\_email('songtianyi630@163.com')

*find\_by\_email* 并不是我们自己的定义的函数，它是如何完成这件事的呢？框架会根据函数名 *find\_by\_email* 动态生成一个查询函数去查询数据库。除了这种黑魔法，元编程还能够动态修改一个类的成员函数，在运行时创建函数等等，这里不展开讲，在 *Ruby* 或者 *Groovy* 的相关书籍里会有详细介绍。语言的反射特性和对模板的支持是实现元编程的主要基础，虽然 *c++* 并不支持反射，但 *c++* 的模板提供了元编程的能力。

编程范式还有很多细分项，比如 *Event-driven programming* , *Distributed programming* [10] 等, 这里不再一一列举。