天方夜谭

一、什么是程序设计语言

我也不知道什么时候开始关注各种各样的计算机程序设计语言的，但是我确实一点点这样做了：从最初对 begin..end 语句块式的模式的厌恶，到对 Lisp 这种神奇的语言的崇拜……我不喜欢太过于深究细节，但是对于语言的特性与选择这方面，还是看到了不少东西的。

后来继续翻看的时候就不只是这么简单了，更多的关注了语言发展和演化，以及某些更深层次的东西，我喜欢没事看着玩，但不喜欢没事写着玩，所以，我看过的代码要远比我写的多得多得多得多。

无论是流行的 Python、Perl、Ruby，还是大神级的 Lisp、Scheme（当然也是 Lisp 系的）、Haskell，还是平民级的 C/C++、Java、PHP，还是说 Javascript 和 Lua 甚至是已经不怎么被使用的 Pascal，我都曾在某种程度上有些涉猎。对于编程范式这概念也有一定程度的理解，所以 OO、FP 还可以算是掌握了的。

但是有时候经常会发现这种问题，我究竟该怎么选择一门语言，究竟哪一种该真正适合我？各种语言之间究竟有着什么样的区别？孰优孰劣？……等等等等关于计算机程序设计语言的相关的问题。

好吧，要想弄清这个问题，就要从非常原始的地方开始说起。

1.1 什么是程序设计语言

几乎任何人都已经了解了语言是从机器指令到汇编语言到高级语言这三个层次的，然而究其演化的过程，我们不清楚，只能通过史料来了解：但我记忆中没有史料说过些什么细节方面的东西，所以，大概 08 年，我十五岁的时候，想到过这样一段话：

> 世上本没有编程，需要解决的问题多了，也就有了编程。
> 世上本没有循环，需要重复的问题多了，也就有了循环。
> 世上本没有死循环，出错的循环多了，也就有了死循环。

笑了哈，这本来感觉就是在拿鲁迅先生开涮：抱着他老人家经典的话语修改来修改去的。但后来我发现这恰恰是当年程序设计语言发展的过程啊！

设计程序本来就是为了解决问题的，这个是毋庸置疑的，而最初的程序设计语言设计的时候也肯定是一条线下来的那种：一步步的照着本来设计好的程序执行，然后得出结果（试想下算盘吧，大概就是这个样子的，这里珠算口诀和算子可以看作编程语言）。

可是大家都知道大概发展到现在所出现的三种结构：顺序、分支、循环。只有顺序执行显然不可能解决太多的问题（这也就是为什么算盘仅仅只能用于简单的基础数学运算的原因），所以，跳转语句出现了：类似 C 语言里面的 goto 和各种汇编语言里面的 jump 指令。没错，最初出现之前没有分支或循环这一说，就好象 x86 汇编语言里面没有 if 和 while 这么高阶的语句，而只有 jnz/jne 这种跳转语句；进一步如果你在汇编代码下调试过 C 语言程序的话也会明显看到 if 和 while 语句其实也被解释成了这种形式。只是当更多人发现跳转最常用也最普遍的抽象是这两种，于是就把他们单独拿了出来，形成了固定的模式：于是就出现了循环。

那么，于是，现在看：高阶程序设计语言其实是对低阶的某些设计的更高层次的抽象：于是在汇编之上有了 C，再之上有了 OO 和 FP。

可是，真的只有这么简单么？

1.2 机器与语言

对于特定机器，其对应的都有自己的机器语言（如果是虚拟机的话则是字节码），而这些机器语言有着相对应的汇编指令，然后，那些能够稳定运行在这些机器上的高阶语言则只不过是通过编译器（Compiler）和解释器（Interpreter）来将自身转化为对应的汇编指令并运行罢了。

可能说了半天大家还是满头雾水。说这些有什么用？这些跟什么是编程语言有什么关系？

好吧，前面说到过编程语言是用来解决问题的工具，但是，工具也有工具的用法，工具也有工具的平台，x86 汇编就只能用于 x86 平台，ARM 汇编就只能用于 ARM 特定平台，Java 编译生成的字节码也就只能用于各种 JVM 平台。

当然，随着平台这东西一步步细分以后，阶层也一步步细分了：

于是出现了高阶语言和汇编语言之分。（某些地方高阶语言也被称为高级语言）

那么，高级语言究竟高级在那儿？

1.3 高级在哪儿？

很多人提到高阶语言的时候大概都会这么说：高阶语言大家就那么几种模式，长得都差不多的。可是，真的差不多？许多习惯了 Java 的人在往 P/R 转的时候会特别不适应：至少语法就令他感觉难受，特别是 Python 的强制缩进：但是这往往并不是因为语言本身的优缺点的问题，而全是自身受其影响所致。

那么，且从上面我列出来的几种语言看一下吧：

对于 C/C++我不想多说什么，因为这是两个让我敬畏的语言：一个做的如此精美，以至于从早期到现在一直是主流；一个却能在保持向下兼容（特指兼容 C 语言）的情况下能够做的如此复杂（试比较语言关键字的数量你或许就能明白更多）！特别是后者出现模板和泛型以后，这将是两个我始终都要进行仰视的语言。

然后是，好吧，追随下主流的目光吧：Java（1995）。Paul Graham 曾经在提到面向对象时说过这样的话：某些语言允许你使用这种风格，某些语言强制你必须要这样编程；他说后者是不可取的，而 Java 则是后一种的典型代表（C#在某种程度上也被看作 Java 系的语言了）。

好吧，某种程度上我们感觉，止步于 Java 和 C#（.NET）就差不多了：学校目前也是这样做的：更注重 J2EE 和.NET 的企业级应用。或许这是一种找到某些工作好找的原因。是啊，有这么多的成功的例子摆在我们面前，为何不向着他们走去呢？但然而当你真正发现写起来程序的时候其实这些是很令人恼火的不停地类型声明，不停地设计类与对象。如果我看我都会抓狂：写一个“Hello world.”都要好多行代码。那么，究竟有没有 Paul Graham 提到过的第一种语言呢？

1.4 更加自由的语言

如果你真正的了解脚本语言，那么一定会知道 Larry Wall，这是一个大神级的人物，创造了大神级别的语言：Perl（1987）。请不要说我搞 XX 崇拜，这个本来就是如此：Perl 某种程度上就是你想要的那种语言；虽然自由的代价一般都是牺牲效率，但是对于今天的计算机系统来讲，这些被牺牲掉的效率已经值得忽略了。

Perl 带来的一种叫做 Quick & Dirty 的东西：有些时候即便你写过的 Perl 代码，你回头翻看的时候都不一定理解的透彻（当然除非你能一直保持良好的习惯来）。动态类型、OO、FP，想要的都已经具备了，更可贵的是 Larry Wall 开创了这么一种新的模式：如果你有想法有能力，自己也可以推出一个新语言。

于是新语言们就来了：Python、Ruby（1993）。

Python 之火现在虽然在中国的高校内部不太常见，但是在开源社区，在国外高校，Python 已经不只是一个编程语言这么简单了：去看下 The Zen of Python（Python 之禅），那是一种哲学意义上的东西了近乎。语法的优雅与高的执行效率为其带来了极高的关注度。甚至是 Python 之父的一句“I wrote python”都让其在 Google 谋得一个职位。

Python 表现的更要自由一些，并且免除了很多 Perl 的弊病，至少 Q&D 的事情出现后少有看不懂自己的代码的时候了，而且 Python 中加入了更多的函数式编程的元素，更自由的迭代器系统以及更加宽松自由的扩展方式。于是今天会出现如此之多的 Python 模块供我们使用，并且可以自由添加我们自己的模块。

Ruby 则是在一个叫做松本行弘的日本人手中设计出来：经典著作《松本行弘的程序世界》中讲了很多相关的东西：甚至涉及到了某些更深入的讨论。但是 Ruby 的诞生进一步出现了这样一个情况（至少现在仍是如此）：Perl 程序员嘲笑 Java 程序员，Python 程序员鄙视 Perl 程序员，Ruby 和 Python 两家又经常争论不休。

某种意义上讲，Ruby 设计的还是要比较好的：也许正是因为它更年轻的原因（虽然 Java 比他还要年轻），考虑到了更多深层的因素，并且从不同地方取长补短，既继承了 Perl 的思想又延续了 Python 的部分优点（但凡能做的都能做，而且还有一套优雅的语法），同时又不失效率。而这三家也分别在促进中，造就了开源社区的兴盛。

而试做这样一个比较：Java->Perl->Python->Ruby，这期间我一直提到的一个关键词是：函数式编程。这样一个顺序的话，越靠后其实对函数式编程的倾向越明显。甚至根据 Paul Graham 的说法，Ruby 的语法如果换一下形式就可以转换成为 Lisp。

没错，Lisp。这才是一切的根源。

二、根源

2.1 Lisp

John McCarthy，Lisp 的创始人。

在还是各种过程式语言的时代（Lisp 出现在 1958 年，经典高阶语言 Fortran 出现在 1957 年），凭空（几乎可以这么说了）这么一个想法，并且如此经久不衰，还对其他的如今还是主流的语言有着如此深远的影响。

这是 Lisp 的一部分资料（来自维基百科）：

Lisp

Paradigm(s) Multi-paradigm: functional, procedural, reflective, meta
Appeared in 1958
Designed by John McCarthy
Developer Steve Russell, Timothy P. Hart, and Mike Levin
Typing discipline Dynamic, strong
Dialects Arc, AutoLISP, Clojure, Common Lisp, Emacs Lisp, ISLISP, Newlisp, Scheme, SKILL
Influenced by IPL
Influenced CLU, Dylan, Falcon, Forth, Haskell, Io, Ioke, JavaScript, Logo, Lua, Mathematica, MDL, ML, Nu, OPS5, Perl, Python, Qi, Rebol, Racket, Ruby, Smalltalk, Tcl

好吧，我确实不知道 IPL 是个什么东西，但是 Influenced 列表中出现了我列出的绝大部分语言。开源社区的语言更占绝对优势。或许不知道 Lisp 是个什么东西，但是那一大坨语言中你至少也听说过这么一两个。

Lisp 的名字来源于“LISt Processing”，而其主要的数据结构就是表（Linked list），而其程序代码都统一写成 S-表达式（S-Expressions）的形式。进一步，Lisp 带来的不只是函数式编程这么一个范式，更出现了动态类型（Dynamic Type）、垃圾回收（Garbage Collection）等等动态语言的特性。如果说 Fortran 是高阶语言始祖的话，那么 Lisp 则给动态语言奠定了一个很好的原型（甚至可以说是一个模范，这就是后来各种语言被其影响这么大并且逐渐向它靠拢的原因）。

这就是根源：如果你想寻找更自由的更体贴程序员的程序设计语言，从他开始就给提供了。

2.2 大神与奇葩

Paul Graham 列出过九条 Lisp 诞生后创造的新思想：

- 条件结构（if-then-else）：因为早期的 Fortran 只依靠 goto 来实现跳转的
- 函数是一种类型：笑，听起来真的略微有些不可思议，但确实如此，正因为这样，才有可函数作为参数的高阶函数和闭包成为可能。
- 递归：Lisp 是第一个支持递归函数的语言。
- 变量的动态类型：Lisp 中变量是没有类型的，只是作为一个类似指针的东西出现，而只有其指向的值才有类型之分。
- 垃圾回收机制（GC）。
- 程序由表达式组成：Lisp 是一些表达式树的哦集合，每个表达式都会返回一个值。也就是说没有 Fortran 或者 C 之类的表达式和语句之分。
- 符号类型。
- 代码使用符号和常量组成的树形表示法。
- 无论什么时候，整个语言都是可用的。Lisp 不会真正的区分读取期、编译期、运行期。

那些时候看来，这一坨几乎全是奇葩的思想：因为当时主流的语言都是要依靠一定的硬件基础的，而 Lisp 则是将自己提升到了数学的高度，很多借鉴了来自数学方面的思想，McCarthy 最初就没有决定把她给做成编程语言，而是想用一种简洁的方式定义图灵机，直

到后来当 Steve Russell 给他实现出来的时候，他才意识到这居然是一个十分强大的编程语言。

当年的奇葩莫名其妙间成了奇葩，然后大家才逐步发现后一步步的模仿，然后奇葩就成了大神。

试看下大神的创造被改成了什么样子：

条件结构就不说了，Brainfuck 这种闪闪发光的语言都可以某种程度上给实现了。

函数是一种值，这方面在 Lua 和 JavaScript 上面体现的更加彻底，Lua 中直接可以将一个函数定义作为直接量赋值给变量。而很多其他语言都会有 lambda 表达式作为匿名函数作为参数传递或者作为结果返回，也可以算是一定程度上的把函数看成一种值来对待。

因为后来 Fortran 借鉴了**递归**这个思想，所以后来的编程语言基本上都加入了这一点。

绝大部分脚本语言都已经实现了**动态类型**和**垃圾回收**的，而且几乎某种程度上保持了当年 Lisp 的特点，稍微有一些不同的话，也大致是类型的内部表示方法和 GC 的算法不同罢了（Lua 的源代码十分小巧，有兴趣的话去看一下）。

程序由表达式组成而非表达式+语句组成。这个在主要的程序语言中还是比较少见，所以很是多情况下还会看到 return 等让 Lisp 大神们感觉起来不爽的东西的（Ruby 又一个奇葩的例外，你就不会怎么看到他有 return 这东西，很多时候就留一个表达式在那里就结束一个函数了）。

后面几个暂时还没怎么被主流所接受，不过看发展趋势的话应该会很快了。

大神就是这样一步步成为大神的。

2.3 另一个大神

计算机最初设计的时候，所需要的都是一些非常简单的指令，这就是当年第一版 Fortran 中为什么连 if-then-else 等语句都没有（上面说过，Lisp 创造的这个概念），简单的跳转语句（最经典的是 goto——虽然现在很少用了，但围绕着他有着尚有很多的可说的内容）就能够搞定他们所需要的东西了。

Edsger Dijkstra 提倡了程序控制的三种组合，后来也变成了我们入门必须掌握的东西：

- 顺序——程序按照顺序执行；
- 循环——一定的条件成立时程序反复执行；
- 分支——满足条件时执行某部分，否则执行另外的部分；

这就是结构化程序设计的思想，知道现在也是至理。但关键在于，很大程度上只实现了控制流程结构化，而数据还没有被结构化（不要扯什么数据结构的概念，两个差得离谱的很）。面向对象的程序设计思想就是在结构化编程对流程控制实现结构化后，又加上了对数据的结构化。面向对象的思想基本包含以下两点内容：

- 不需要知道内部的详细处理就可以进行操作（封装、数据抽象）
- 根据不同的数据类型自动选择适当的方法（多态性）

封装的概念是非常重要的，数据会被当成黑盒（Blackbox）来看待，每一步具体的细节操作是对外不可见的，只需要调用相应的接口就可以（因为有些时候我们没必要深究他的实现）。

多态性则使得程序员摆脱了设计不同数据类型的不同处理方式的复杂的分支处理方法；如果没有多态，则程序中会到处都是分支处理，并且，在变更和追加数据类型的时候会变得非常之困难。

或许对于很多人来说这个还是难以理解，但不可否认面向对象确实可以提高生产效率；

而如果把面向对象看成是结构化编程的扩展，那么，对象是否是现实世界中具体物体的抽象化反应就不是多么重要了，即便是如此，对象也只能表现现实物体的某一侧边的抽象概念而已，并不需要考虑绝大多数本来就无用的属性。

C++并不是第一个加入 OO（Object-Oriented，面向对象）的语言（Lisp 加入 OO 都要比 C++早），但是要比比他早的和比他迟的那些 OO 语言都比较成功，Bjarne Stroustrup 一不小心就把来自 Simula 的 OO 特性加到了 C 上面，形成了 C with classes，不久进而进化成了 C++，然后不断加入各种概念：多继承、模板、友元、抽象类、虚基类等等；而最关键的一点是 Stroustrup 几乎完整保留了与 ANSI C 的完整的兼容性，使得程序更好的移植和更好的复用 C 的代码。

OO 不能不说是另一个神奇的创造（在 C 里面加入这一个概念更是一个非常漂亮的组合，C++做到了）。然后这个神奇的创造被 Stroustrup/C++给发展了起来，进而影响了很多后来的程序设计语言，OO 思想深入人心也正是因为他。

所以，这是另一个大神，另一个奇葩。

三、抽象（Abstraction）

3.0 未有对象之前（Age before Object-Oriented）

软件开发的最大敌人就是复杂性。人类的大脑无法做太复杂的处理，记忆和理解力也是有限的。但计算机运行软件却没有这样限制，无论多么复杂的软件，无论有多少数据，无论要运行多长时间，计算机都可以处理，而根据 Andy-Bill 定律和摩尔定律，计算机硬件是越来越复杂的，随之软件业变得越来越复杂。计算机性能是有一定程度限制的，但人类的理解力局限性给软件生产力带来的限制则更大，即便是牺牲一部分性能也在所不惜。

最初挑战这种复杂性的是结构化编程（想想之前提到的 Edsger Dijkstra 大神），前面也提到过他相应的思想，通过顺序-条件-循环三种结构控制程序流程，在降低了复杂度的情况下保留了程序的实现能力。

结构化编程中，我们可以通过方法调用来实现程序的，而不必追究程序的内部的具体实现，也是实现了一定程度的黑盒化的。但是前面提到过，流程控制的结构化虽然实现，但还没有实现数据的结构化，这个时候面向对象就出现了。

面向对象这东西有些时候确实很难理解，因为他表示的就是一个抽象概念，更不用提他还是一种抽象方法了。越抽象的东西就越难理解。而且在很多时候在讲面向对象时所作出的比喻往往和实际编程之间差距太大。而且面向对象的语言又各自有着不同的特色，一下理解也要费很大的劲，所以不妨一步步的分开来理解。

3.1 多态性（Polymorphism）

多态性可能是 OO 中最重要的一个特性。

多态，是指 OO 的程序执行时，会把不同种类的东西使用相同的方法来处理；而不同的对象可能会对一个特定的行为有着公共的接口，OO 程序只需要调用这个接口就可以了，而

不必去考虑这个对象究竟是什么。

试考虑下之前结构化程序设计的时候处理不同类型数据的方法：要用一个复杂的分支语句来判断该数据的类型，然后再在该数据类型的基础上进行相应的操作；而当需要增加更多的数据类型的时候，就需要对这个方法进行修改，而且出现的数据类型越多程序就越复杂，需要修改的也就越多，出错的可能性也就越大，结果可能甚至导致程序根本就无法运行。

OO 的方法则是针对每个类型各自实现一套公共接口，而免去程序员们手动去判断数据类型这个环节，当需要改行为的时候，程序调用下该接口即可。这样就把本来该由人完成的东西交由计算机去完成，效率和质量都会提高很多（具体代码实现可以参见维基百科 的 Polymorphism 词条）。

那么，单说是能够提高效率和质量是不行的，多态性的好处究竟在哪儿呢？

首先，各种数据可以统一的处理。多态性可以让程序员只关注要处理什么（What），而不是怎么去处理（How）。

其次，根据对象的不同自动选择最合适的方法，而程序内部则不会发生冲突。不管调用哪一类对象对应的方法，他都能够自动处理，不必担心内部错误，这样就减轻了程序员的负担。

再次，如果有新数据需要对应的处理的话，通过简单的追加就可以实现了，而不需要改动以前的程序，这样就对程序具备了扩展性。

综上：多态性是在保证质量的基础上提高了开发效率的，所以说 OO 的最重要的一个概念就是多态。

3.2 数据抽象（Data Abstraction/Encapsulation）

多态、数据抽象和继承被列为 OOP 的三原则（可能通常会有别的称呼，如数据抽象之于封装等等）。数据抽象也是 OO 中非常重要一个特性。

因为程序设计不只会有流程控制，还要有对数据的处理。而前面说过结构化程序设计没有处理数据复杂性的问题，OO 的数据封装就是为这个而产生的。

为了得到正确的结果，就要保持处理和数据的一致性，这在结构化编程中是非常难实现的，而数据抽象针对这个问题的解决方案。

数据抽象是数据和数据处理方法的结合。对数据内容的处理和操作必须通过事先定义好的方法来进行。数据和处理方法结合起来就成了黑盒。这样内部实现就有了一致性，如果需要修改就可以具有针对性的修改。而抽象数据能够对特定的操作产生反应，是一种智能化的数据，使用抽象数据可以更好的模拟现实中的实体。

有了数据抽象，程序处理的数据就不再是单纯的数值或文字这些概念性的东西，而变成了人脑容易想象的具体事物。而代码的“抽象化”则是把想象的过程“具体化”。这种智能数据可以模拟现实中的实体，因而被称为“对象”，面向对象编程也因此得名。

3.3 继承（Inheritance）

出现在程序中的对象，通常具有相同的动作。从抽象的原则来说，多个相同/相似的事物出现时，应该组合在一起。而且，程序的重复是很多问题的根源。重复的程序在需要修改的时候所涉及的范围就会更广，费用也就会更高。多个重复的地方需要修改时，哪怕漏掉其中之一，程序都无法正常工作。所以，重复降低了程序的可靠性。

避免重复就要提高复用度，为了避免重复，在 OOP 中，有下面几种方法来管理对象：

一种是基于原型（Prototype-based）：典型的例子就是 Javascript，通过原始对象的副本来作为新的相同的对象。

另外一种则是基于类（Class-based）：相同类型的对象通过同属于一个类，同样类型对象分别属于同样的类，操作方法和属性可以共享。与基于原型的面向对象不同，这一种通常是采用比较多的。

但是，当随着软件规模扩大的时候，也会逐渐产生更多相似的类，这样仍然会产生重复，所以如果能够通过一种方法再度解决这种重复，降低复杂度，就更好了。

于是，继承就出现了。

继承既保留了原有类的特性，又在此基础上产生相应的新类。原来的类成为父类，生成的新类称为子类。子类继承父类的所有方法，如果需要还可以增加新的方法（或者重写父类的方法）。在子类进行的任何修改都不会影响到父类，从而不会影响到其他的子类。

于是有这样两种形式来实现继承：一种是在现有类的基础上提取抽象相同/相似的部分来生成父类，另一种则是通过现有的类来派生子类。前一种方法称为自底向上，后一种则叫做自顶向下。

但是自底向上的抽象过程中，通常一个子类可能跟多种不同的类被抽象出几个父类来，这样一个子类就不只有一个父类了（不要想成一个孩子有多个父亲！），于是很多语言中加入了这样一个特性：多重继承。

单继承是纯粹的树形继承结构，这样确实能够使得类之间的关系变得简单不混乱，但也会对继承关系进行限制，一定程度上还是会使程序重复（如 iostream 类型，在单一继承中，只能挂靠于 istream 或者 ostream，而需要再次实现另外一个类的代码，多继承就可以比较好的解决这个问题）。多重继承的优点就体现了出来了。

松本行弘有这么一个对比，说多重继承和 goto 语句类似：功能强大、易于使用、却使得程序变得极为复杂不利于结构化。而且多重继承还会出现更多问题：比如当继承多个类时的优先级和功能冲突。C++中通过虚基类等来解决该问题，Java 中通过 interface（接口）来解决该问题，动态语言中则可以通过 Mix-in 实现（《松本行弘的程序世界》中有介绍）。

四、具体（Concreteness）

更具体请参见：
http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions) 。

4.1 值与类型（Values and types）

第一个让我能够感到值的类型的区别较大的是 Lua，在他的类型体系中只有仅有的几种：number、string、table、function、boolean、nil、以及 userdata 和 thread（我们讨论的时候忽略后两种）。在 javascript 中也只有这么几种基本类型：undefined、number、string、boolean、function 和 object。如果类比一下 table 和 object，nil 和 undefined 就会发现这两个是如此的相似；而实际上也是这样的，这两者都受 Lisp 的影响特别大，都是嵌入进宿主中的语言，基于原型的面向对象和把函数看成第一类值等特性使得两者除了语法上写起来略有不同，而在其他方面几乎就是同一类型的。

对于值与类型的讨论是无穷尽的，你可以照着你的习惯任意去定义：Lisp 中只有函数和表、C++选择的是继承 C 的类型系统并扩展、各种不同的动态语言大致都有一套类似的系统

等等。

如果说要讨论的话，第一个要讨论的就是，**数值类型**。

首先，数字在数学中是绝对重要的一个类型，而且其分类也是极为严谨明确的，但到了计算机中，因为硬件的限制，所以必须要对其进行某种程度上的归类与取舍，才能够更好的进行计算或者数据处理。

C 语言在这方面做的比较好，从一个字节长度到双精度浮点类型都有相关的类型关键字，还有符号和无符号之分（而刚好这也是他为什么能够这么强大的其中一个原因），而等到一步步抽象到高阶语言如 Python/Ruby 的时候，就没有这么细化了，整数与小数的分别还在，但是处理起来就比较简单了。而到了 Lua 和 Javascript 中，则只有一个 number 了：他就是一个数，不是什么别的乱七八糟的东西，但凡符合数的操作都可以对他使用。

而像 Ruby/Python 这种比较正式点的语言，其中还内建了 Complex（复数）对象/类型，甚至可以直接支持相应的直接量表示，所以这些语言有些时候用于做科学计算的话在开发效率上要高出 C/C++很多，在丰富度上要比 Lua 这种小语言高出很多，是一个不错的选择（NumPy 已经在进行相应的应用了）。

字符串也是一个重要的类型。

C 语言中是没有字符串这东西的，而在其后的很多语言中却成了标配（如 C++标准库中的<string>），在 C 语言看来，字符串不过是一坨用 char 类型堆砌起来的数组所以你的对字符串的操作其实质上是对一个个 8bit 的内存空间进行操作；但在其他许多主流语言里面，字符串就是字符串，它可以拥有很多特性的：你可以在 Python 中执行下 dir(str)，他会告诉你很多。

说到这里不得不提到 Larry Wall 大神的 Perl 了，Perl 有一个十分抽象的类型，叫 Scalar，就是 number 和 string 的合体，Perl 认为一定程度上，他们俩是一致的，所以给归为了一类。这层一度让我思考过究竟 string 该算是什么东西，究竟该怎么规划，这个问题估计以后可能会提到吧。

再往下来大致就是**空类型**和**布尔类型**了，空类型分大概那么几种，NULL，nil，undefined。大致表示值未定义，或者是值不存在，或者是值为空。而 boolean 更为简单，其就是用来代表逻辑运算和关系运算的结果的：true（真）或者 false（假）。ANSI C 中没有定义布尔型，逻辑运算和关系运算的结果用 1 或者 0 来表示，所以某种程度上来说，false，0，和 NULL（nil/undefined，其中 ANSI C 里面 NULL 的定义是((void *)0)）是等价的。

然后就是**容器**（STL 看多了，就不禁用了这个名字）。

C 语言中的容器基本上就是数组和结构体（由指针能够构成的其他复杂数据结构暂且忽略）：一个允许连续存放固定数目的同一类型的数据，另一个则允许连续存放固定顺序的不同类型的数据。然后这样一步步衍生，到了 Python 中变成了列表/元组、集合和字典（List、Tuple、Set、Dict），而且内置了基础的类似堆栈和队列的操作（append、insert、pop、reverse 等），既有可变长又有不可变长，既序列式也有关联式，同时又有无序性容器，这样为科学计算辅助了不少。而再进一步演变，到了 Javascript 和 Lua 中，一切变得更加简单了。Javascript 中有 Array 对象和 object 之分，一个是序列式的，另外一个是关联式 key-value 型，这种简单不失可用性的特性造就了 JSON 这种数据封装形式。Lua 中只有表（table）这种东西，无论是通过数值还是通过键来索引都可以，并且没有不可变长这一说，随时都可以改动，对容器的精简几乎达到了极致（好吧，不要忘了 Lisp，Lua 可是参考着他来设计的）。

好吧，终于到**函数**了。

在学习 C/C++的过程中，很多时候函数对我们来说只是结构化编程中的一个组成部件，简单的用于流程控制会使用参数并影响参数然后有或者没有返回值返回的一个东西。最多也就在读一些高端的代码的时候会看到 C 语言中使用函数指针作为参数（stdlib.h 中的 qsort

函数），那种时候也不会太有函数是值的概念，只是说把这个函数的地址传给他，到时候会被调用罢了。

但是，javascript 及 Lua 中甚至可以允许这种写法：

```
x=function(){....}
x=function()....end
```

这样，x 的值就是一个函数，赋值语句左边的可以看成函数的直接量。x 的值同时也可以赋给其他变量真正可以吧函数作为值的感觉来对待：Javascript 甚至还内置了很多方法来对函数对象进行操作，然后各种语言都有支持函数作为参数传递及 Lambda 表达式（本质是，匿名函数？）和闭包。然后的然后，一步步的，就不再把函数当成一个简单的东西来对待了。也貌似就这样，函数式编程的某些概念就深入人心了。

4.2 条件与控制（Condition & Control）

4.2.0 我了个去（goto/jump）

这是一个很好玩的语句，不是么？

我们现在明明知道有了它会糟糕很多，但是还是有很多语言不舍得丢弃它。

嗯，在只有 goto/jump 的时代，确实写出来会糟糕很多：且不说回到早期语言的代码去翻看了，只看 x86 的汇编语言你就会懂得许多：jz/jnz、je/jne、jmp 等指令多用几个就会让你修改起来十分头疼了（大牛们除外）。可是，在看 C 语言代码的时候，你可能会看到不少大神们会写不少 goto 语句的，而且还是点睛之笔的那种加入，令你不得不佩服。

高德纳（Donald Knuth）等大神们都曾经对 goto 进行过十分深入的讨论，特别是关于是否保留的问题上，当然最终的结果是保留（只听说过高先生有一个特别长的论文，从来没有拜读过），但是，不推荐使用，所以，后来的语言很多就干脆忽略了他。

但是，如果没有早期的 goto 语句的话，程序如果只是连续执行的话。那么我们要比现在苦逼更多。正是 goto 语句的反复利用，才被总结出三种经典流程控制语句的，才会有我们后来的更多的创造的。

4.2.1 如果，那么……（if/then/else）

前面有说过，Lisp 创造了这一流程控制语句（好吧现在看来 C 中的那个三元运算符跟 Lisp 中的方法感觉差不多），然后被 Fortran 抄去了，然后被大多数语言使用了，然后就这个样子了。大多数人除了简单的 Hello,world.程序外，下一个接触的估计就是他了。就我感觉，那种稍微像样子一点的项目里面，每一个负责逻辑控制的源码文件里都应该至少有一句 if 的。

有了 if，有了 else/else if，那其实就可以创造很多分支了。但是，大神们说这样还是不够的，于是一步步就出现了 switch/case。我曾想过，人都是趋于懒惰的，于是懒惰的大神们就这样创造了他（懒惰的大神们不止创造了这一个东西的）。

4.2.2 当/为了（while/for）

这两个字眼翻译的有点小锉。

C 语言老师教给我们的时候就强加了一个先入为主的思想：while 是条件循环，for 是计数循环。可是，貌似不是这样的。

Go 语言以他有较少的关键字为荣，while 也被他整合到 for 里面去了，这个的感觉就好象直接把高校的老师们打击的彻头彻尾的。但是有一点是没有变的：我们使用循环大部分也就是这两个目的，一个是条件式使用，一个是遍历/迭代某容器的时候（这就是后期的 foreach 这么流行和大多数动态语言直接把 for 改成了迭代式的原因）。

然后在这里我又想到了当年那几句我无聊时总结出来的话（见 1.1）：循环的出现就是这样子的，问题的出现也是这样子的。有时候死循环的出现并不一定是因为程序员出错了，甚至死循环是一个非常值得好好利用的东西，循环是死的，程序是活的。

4.2.3 直到/否则（until/unless）

当时在 Ruby 中看到 unless 这个用法的时候感觉松本实在是太懒了，if not 这种东西都要改成另外一个单独的关键字。后来才发现我竟然能够错的这么严重。

until 循环当时就感觉是一个不太怎么有用的设定。这个的缘起我已经找不到了，但是很明显能够感觉到，当型/直到型的循环这种区分我认为设置起来没什么必要，很多人不一直是 while 或者 for 就用下去了么？Go 语言的果断让我比较喜欢，对不必要的东西直接给予无视。

但是，当我第一次看到了 Ruby 的 unless（好吧，准确地说应该是 Perl 的 unless），当真是觉得程序员们有些可爱了。懒惰是人的天性，很多地方都看得出来。我想，除了 Larry Wall 大神是语言学家外，另一个加入 unless 的原因估计就是想省一些功夫吧。

4.2.4 表达式修饰符（Modifier）

这也是 Perl 中的创造吧估计。

我就想，一个语言学家能够给计算机编程带来多大的影响：修饰符这种方式估计也算是非常瑰丽的一个创造了。

在没有修饰符的年代，你需要这样写东西：

```
if ($n < 0){
    print "$n is a negative numbers.\n";
}
```

可是，一不小心你就会发现，Perl 中可以允许你这样来完成同样的事情：

```
print "$n is a negative numbers.\n"  if $n < 0;
```

这个瞬间感觉就不同了：你用一行代码完成了原来三行代码完成的事情（好吧，不考虑其他人可能还会有的各种不良语法习惯），然后达到了完全一样的效果，而且看起来还要比三行代码舒服很多，又能够容易理解，这明显就像是自然语言中的倒装句式嘛。

4.2.5 中断执行（break/continue/return）

这三个单独几乎不能怎么使用（除了 return 能够作为函数返回值），需要结合各种控制语句，并辅助他们表现得更好，而 break 和 continue 等也可以看作为 goto 的特例化（至于 Javascript 中的那个几乎就能拿来当作 goto 使用的 break 除外）。

然后，这三个出现以后，goto 就逐渐消退了，if 和 for 就逐渐流行了，他们能够一直在默默的执行着 goto 的任务，默默的做那个幕后的推手。

无论如何，这三者的存在都是为了更好地服务结构化程序设计，而为了避免 goto 的乱入和泛滥，都尽了十分的力气来做到自己应尽的责任。特别是 return，他的出现导致了函数可以在任意地方直接中断执行而后返回，很大程度上方便了 goto 之后的程序员们；虽然缺陷也很重，但这是后话了。

4.3 过程定义（Procedure* Definition）

（*这里的过程[procedure]，也可以在某些说法里面被称为函数[function]，这里我们可能会混用）

4.3.1 关键字（def[ine]/fun[c[tion]]/let/sub[routine]）

首先要说的其实不是上面那一大坨关键字，因为很多人最初接触的时候遇到的是 C 语言系的过程定义，直接返回值类型另外附上一堆东西就可以定义一个过程（函数），所以就省去了 def 或者 function 什么的，易读性也就高了很多。但这样子要定义匿名函数就很困难，C++因为这个原因还专门加入了 lambda 关键字（C++11 标准）。这样看来，并不是某一种做法一定就是好的，总要权衡选择最合适的。

let 出入于 ML 系语言中，Haskell、Ocaml 和 F#都有出现；**def** 是 Python 和 Ruby 用的；**define** 则属于 scheme（另外一个奇葩的是 Common Lisp，用的是 **defun**）；**fun** 在 ML 中和 Erlang 中见到过，而到了 Go 那里就变成了 **func**，JavaScript 和 Lua 则选用了最长的 **function**；**sub** 则存在于 Perl 和 Basic 中。

这些关键字里面更喜欢的是 **fun** 和 **let**，**fun** 给人的感觉就好象是他的字面意思，有趣，而且简化了 **function** 这么一长串，而 **let** 比起 **define** 来的感觉更像是在定义东西（好吧，虽然 sheme 中这两者都会出现过，但是意义完全不一样）。重点在于，首先一个关键字要能表现出他究竟要做什么，而不一定非得用最专业的表达方式：

```
let square x = x*x
```

在 GHCi 里面写出来看起来怎么读怎么顺口。而

```
int square(int x){return x*x;}
```

或者

```
function square(x){return x*x;}
```

则就是另外一种完全不一样的感觉了。

4.3.2 块（do..end/begin..end/do..done/{..}/(..)）

块也是一个很影响风格的东西从而惯坏程序员们的东西。

在命令式程序设计语言里面，很多都是用的 begin/do..end/done 和花括号的风格。一方面对语法逻辑有一个清晰的分割和控制，另一方面能够限定变量的作用域。但是有些时候，有些语言，是可以避免这些问题的。

Python 就是用的缩进来表示的语句块，而这种方式省去了各种块结束标志，然后程序的清晰度和可以变得很高。Haskell 等 ML 系语言也是如此。

而另外一个奇葩就是 Lisp 系语言。他们本身就没有语句这个概念，万物皆表达式，只要能获取到表达式的结果就好。所以，语句块也是一个表达式，也可以用 S-expression 来表示，所以，除了成对的括号堆叠之外，没有什么可以分隔他们的语言元素的东西。