01 编程语言概述 补充阅读

TA 谢润烁
2024/2/27

《01 编程语言概述》这一讲中有非常多新概念，而为了更好地理解这些概念，阅读相关材料是有必要的。由于维基百科的质量要远高于百度百科（百度百科有很多是直接抄维基百科的），英文维基百科的词条质量也要高于其它语言维基百科的词条质量（尤其是科技类词条）。因此，我们选用英文维基百科上的一些词条作为我们的补充读物。

为了方便大家的阅读，我对其中部分专有名词和可能大家不太熟悉的词汇加了中文注释（没有形成固定中文翻译的词汇则没有加），也在一些我觉得可以解释或者拓展一下的地方做了注解。这些材料会非常有利于更好地理解这些概念，也没有晦涩难懂的词汇，所以大家认真读的话并不会感觉到吃力。在日后课程的学习中，我们接触到的很多书籍、手册和课件都会是全英文的，所以我们要从现在开始养成阅读英文材料的习惯，培养阅读英文材料的能力。

1 通用编程语言(GPL)与领域特定语言(DSL)[1]

In computer software, a **general-purpose programming language (GPL)** is a programming language for building software in a wide variety of application domains. Conversely, a **domain-specific programming language (DSL)** is used within a specific area. For example, Python is a GPL, while SQL is a DSL for querying relational databases.

> SQL语言应用于数据库操作，大家会在数据库课程中学习到

The distinction between general-purpose programming languages and domain-specific programming languages is not always clear. A programming language may be created for a specific task, but used beyond that original domain and thus be considered a general purpose programming language. … Inversely, a language may be designed for general use but only applied in a specific area in practice. A programming language that is well suited for a problem, whether it be general-purpose language or DSL, should minimize the level of detail required while still being expressive enough in the problem domain. As the name suggests, general-purpose language is "general" in that it cannot provide support for domain-specific notation while DSLs can be designed in diverse problem domains to handle this problem. General-purpose languages are preferred to DSLs when an application domain is not well understood enough to warrant its own language. In this case, a general-purpose language with an appropriate library of data types and functions for the domain may be used instead. While DSLs are usually smaller than GPL in that they offer a smaller range of notations of abstractions, some DSLs actually contain an entire GPL as a sublanguage. In these instances, the DSLs are able to offer domain-specific expressive power along with the expressive power of GPL.

General Purpose programming languages are all Turing complete(图灵完备), meaning that they can theoretically solve any computational problem. Domain-specific languages are often similarly Turing complete but are not exclusively so(不完全都是图灵完备的).

> 图灵完备：即语言的能力与图灵机的能力等价

2 编译与编译器[2]

In computing, a **compiler(编译器)** is a computer program that translates computer code written in one programming language (the **source language(源语言)**) into another language (the **target language(目标语言)**). The name "compiler" is primarily used for programs that translate source code from a **high-level** programming language to a **low-level** programming

language (e.g. **assembly language(汇编语言)**, **object code(目标语言)**, or **machine code(机器语言)**) to create an **executable program(可执行程序)**.

There are many different types of compilers which produce output in different useful forms. A **cross-compiler(交叉编译器)** produces code for a different CPU or operating system than the one on which the cross-compiler itself runs. A **bootstrap(自举) compiler** is often a temporary compiler, used for compiling a more permanent or better optimised compiler for a language.

Related software include **decompilers(反编译器)**, programs that translate from low-level languages to higher level ones; programs that translate between high-level languages, usually called **source-to-source compilers** or **transpilers**; **language rewriters**, usually programs that translate the form of expressions without a change of language; and **compiler-compilers**, compilers that produce compilers (or parts of them), often in a generic and reusable way so as to be able to produce many differing compilers.

A compiler is likely to perform some or all of the following operations, often called **phases**: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and machine specific code generation. Compilers generally implement these phases as **modular components**, promoting efficient design and correctness of transformations of source input to target output. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness.

Compilers are not the only language processor used to transform source programs. An **interpreter(解释器)** is computer software that transforms and then executes the indicated operations. The translation process influences the design of computer languages, which leads to a preference of compilation or interpretation. In theory, a programming language can have both a compiler and an interpreter. In practice, programming languages tend to be associated with just one (a compiler or an interpreter).

3 解释与解释器[3]

In computer science, an **interpreter(解释器)** is a computer program that directly executes instructions written in a **programming(编程语言/程序设计语言)** or **scripting(脚本语言) language**, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:

1. **Parse** the source code and perform its behavior directly;

2. Translate source code into some efficient **intermediate representation(IR, 中间表示)** or **object code(目标代码)** and immediately execute that;
3. Explicitly execute stored precompiled **bytecode(字节码)** made by a compiler and matched with the interpreter's Virtual Machine.

Early versions of Lisp programming language and minicomputer and microcomputer BASIC dialects would be examples of the first type. Perl, Raku, Python, MATLAB, and Ruby are examples of the second, while UCSD Pascal is an example of the third type. Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and/or compiler (for JIT systems). Some systems, such as Smalltalk and contemporary versions of BASIC and Java, may also combine two and three types. Interpreters of various types have also been constructed for many languages traditionally associated with compilation, such as Algol, Fortran, Cobol, C and C++.

While interpretation and compilation are the two main means by which programming languages are implemented, **they are not mutually exclusive(互斥)**, as most interpreting systems also perform some translation work, just like compilers. **The terms "interpreted language" or "compiled language" signify that the canonical implementation of that language is an interpreter or a compiler, respectively**. **A high-level language is ideally an abstraction independent of particular implementations**.

4 命令式编程

In computer science, **imperative programming(命令式编程)** is a **programming paradigm(编程范式)** of software that uses statements that change a program's state. In much the same way that the imperative mood(语气) in natural languages expresses commands, an imperative program consists of commands for the computer to perform. **Imperative programming focuses on describing how a program operates step by step, rather than on high-level descriptions of its expected results**.

The term is often used in contrast to **declarative programming(声明式编程)**, which **focuses on what the program should accomplish without specifying all the details of how the program should achieve the result**.

4.1 过程式编程(面向过程编程)

Procedural programming(过程式编程) is a type of **imperative programming** in which the program is built from one or more **procedures(过程)** (also termed **subroutines(子程序)** or **functions(函数)**). The terms are often used as synonyms(同义词), but the use of procedures has a dramatic effect on how imperative programs appear and how they are

constructed. Heavy procedural programming, in which state changes are localized to procedures or restricted to **explicit arguments(显式实参)** and returns from procedures, is a form of **structured programming(结构化编程)**. Since the 1960's, **structured programming** and **modular programming(模块化编程)** in general have been promoted as techniques to improve the **maintainability(可维护性)** and overall quality of imperative programs. The concepts behind **object-oriented programming(面向对象编程)** attempt to extend this approach.

Procedural programming could be considered a step toward **declarative programming**. A programmer can often tell, simply by looking at the names, arguments, and return types of procedures (and related comments), what a particular procedure is supposed to do, without necessarily looking at the details of how it achieves its result. At the same time, a complete program is still imperative since it fixes the statements to be executed and their order of execution to a large extent.

> 这里指出了结构化编程所带来的效果其实已经有点像声明式编程——也就是说，通过把一系列的操作封装成一个函数，然后你可以在不知道这个函数内部实现机制的情况下调用它，你只需要知道这个函数的功能和接口（参数列表，返回值）即可。

4.2 面向对象编程[4]

Object-oriented programming (OOP, 面向对象编程) is a **programming paradigm(编程范式)** based on the concept of **objects**, which can contain **data** and **code**: data in the form of **fields(域)** (often known as **attributes** or **properties**), and code in the form of **procedures** (often known as **methods(方法)**). In OOP, computer programs are designed by **making them out of objects that interact with one another**.

> procedure, method, function, subroutine, … 这些词其实本质上都是同个意思

Many of the most widely used programming languages (such as C++, Java, Python, etc.) are **multi-paradigm** and **they support object-oriented programming to a greater or lesser degree**, typically in combination with **imperative programming**, **procedural programming** and **functional programming**.

Significant object-oriented languages include Ada, ActionScript, C++, Common Lisp, C#, Dart, Eiffel, Fortran 2003, Haxe, Java, JavaScript, Kotlin, Logo, MATLAB, Objective-C, Object Pascal, Perl, PHP, Python, R, Raku, Ruby, Scala, SIMSCRIPT, Simula, Smalltalk, Swift, Vala and Visual Basic.NET.

4.3 过程式与面向对象的历史

The earliest imperative languages were the machine languages of the original computers. In these languages, instructions were very simple, which made hardware implementation easier but hindered the creation of complex programs. FORTRAN, developed by John Backus at International Business Machines (IBM) starting in 1954, was the first major programming language to remove the obstacles presented by machine code in the creation of complex programs. FORTRAN was a compiled language that allowed **named variables(命名变量)**, **complex expressions(表达式)**, **subprograms(子程序)**, and many other features now common in imperative languages. The next two decades saw the development of many other major high-level imperative programming languages. In the late 1950s and 1960s, ALGOL was developed in order to allow mathematical algorithms to be more easily expressed and even served as the operating system's target language for some computers. MUMPS (1966) carried the imperative paradigm to a logical extreme, by not having any statements at all, relying purely on commands, even to the extent of making the IF and ELSE commands independent of each other, connected only by an intrinsic variable named $TEST. COBOL (1960) and BASIC (1964) were both attempts to make programming syntax look more like English. In the 1970s, Pascal was developed by Niklaus Wirth, and C was created by Dennis Ritchie while he was working at Bell Laboratories. Wirth went on to design Modula-2 and Oberon. For the needs of the United States Department of Defense, Jean Ichbiah and a team at Honeywell began designing Ada in 1978, after a 4-year project to define the requirements for the language. The **specification(规约/规格说明)** was first published in 1983, with revisions in 1995, 2005, and 2012.

The 1980s saw a rapid growth in interest in **object-oriented programming**. These languages were imperative in style, but added features to support objects. The last two decades of the 20th century saw the development of many such languages. Smalltalk-80, originally conceived by Alan Kay in 1969, was released in 1980, by the Xerox Palo Alto Research Center (PARC). Drawing from concepts in another object-oriented language—Simula (which is considered the world's first object-oriented programming language, developed in the 1960s)—Bjarne Stroustrup designed C++, an object-oriented language based on C. Design of C++began in 1979 and the first implementation was completed in 1983. In the late 1980s and 1990s, the notable imperative languages drawing on object-oriented concepts were Perl, released by Larry Wall in 1987; Python, released by Guido van Rossum in 1990; Visual Basic and Visual C++ (which included Microsoft Foundation Class Library (MFC) 2.0), released by Microsoft in 1991 and 1993 respectively; PHP, released by Rasmus Lerdorf in 1994; Java, by James Gosling (Sun Microsystems) in 1995, JavaScript, by Brendan Eich (Netscape), and Ruby, by Yukihiro "Matz" Matsumoto, both released in 1995. Microsoft's .NET Framework (2002) is imperative at its core, as are its main target

languages, VB.NET and C# that run on it; however Microsoft's F#, a functional language, also runs on it.

5 声明式编程

In computer science, **declarative programming(声明式编程)** is a **programming paradigm**—a style of building the structure and elements of computer programs—that **expresses the logic of a computation without describing its control flow(控制流)**.

Many languages that apply this style attempt to minimize or eliminate **side effects(副作用)** by describing *what* the program must accomplish in terms of the problem domain, rather than describing how to accomplish it as a sequence of the programming language **primitives(原语)** (the *how* being left up to the language's implementation). This is in contrast with imperative programming, which implements algorithms in explicit steps.

> 函数的副作用(side effects)会在课程接下来的lambda calculus和SICP部分来学习；
> 原语指的是原子操作，不能再分解的操作

Declarative programming often considers programs as theories of a **formal logic(形式逻辑)**, and computations as **deductions(演绎推理)** in that logic space. Declarative programming may greatly simplify writing parallel programs.

> 从某种程度上讲，人类做reasoning(推理)主要就是两种：induction(归纳)和deduction(演绎)

Common declarative languages include those of **database query languages(数据库查询)** (e.g., SQL, XQuery), **regular expressions(正则表达式)**, **logic programming** (e.g. Prolog, Datalog, answer set programming), **functional programming**, and **configuration management systems(配置管理系统)**.

> 正则表达式是很有用的工具，大家会在编译原理课程中接触到，主要可以用于模式匹配等任务

The term is often used in contrast to imperative programming, which **dictates(命令) the transformation steps of its state explicitly**.

5.1 逻辑式编程

由于这部分比较超纲，也比较小众，故不整理相关阅读材料

5.2 函数式编程[5]

In computer science, **functional programming(函数式编程)** is a programming paradigm where programs are constructed by applying and composing functions. It is a **declarative programming(声明式编程)** paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

In functional programming, functions are treated as *first-class citizens*(一等公民), meaning that **they can be bound(被绑定) to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can**. This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

Functional programming is sometimes treated as synonymous with **purely functional programming**, a subset of functional programming which treats all functions as deterministic mathematical functions, or **pure functions**. When a pure function is called with some given arguments, it will always return the same result, and cannot be affected by any **mutable(可变) state** or other **side effects**. This is in contrast with impure procedures, common in imperative programming, which can have side effects (such as modifying the program's state or taking input from a user). Proponents of purely functional programming claim that by restricting side effects, programs can have fewer bugs, be easier to debug and test, and be more suited to formal verification.

> 在课程接下来的SICP部分，我们会接触到mutable与immutable的值，也会接触到pure function和有side effect的function

Functional programming has its roots in academia, evolving from the **lambda calculus(lambda演算)**, a formal system of computation based only on functions. Functional programming has historically been less popular than imperative programming, but many functional languages are seeing use today in industry and education, including Common Lisp, Scheme, Clojure, Wolfram Language, Racket, Erlang, Elixir, OCaml, Haskell, and F#. Functional programming is also key to some languages that have found success in specific domains, like JavaScript in the Web, R in statistics, J, K and Q in financial analysis, and XQuery/XSLT for XML. Domain-specific declarative languages like SQL and Lex/Yacc use some elements of functional programming, such as not allowing mutable values. In addition,

many other programming languages support programming in a functional style or have implemented features from functional programming, such as C++11, C#, Kotlin, Perl, PHP, Python, Go, Rust, Raku, Scala, and Java (since Java 8).

1. Wikipedia contributors, "General-purpose programming language," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=General-purpose_programming_language&oldid=1206017538 (accessed February 26, 2024). ↩

2. Wikipedia contributors, "Compiler," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Compiler&oldid=1202531084 (accessed February 26, 2024). ↩

3. Wikipedia contributors, "Interpreter (computing)," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Interpreter_(computing)&oldid=1209168351 (accessed February 26, 2024). ↩

4. Wikipedia contributors, "Object-oriented programming," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=1209497648 (accessed February 26, 2024). ↩

5. Wikipedia contributors, "Functional programming," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Functional_programming&oldid=1201599060 (accessed February 27, 2024). ↩