面向对象五十弦

Lesley Lai
原作者

CAIMEO
翻译提议

Chuigda Whitegive
编译
Doki Doki λ Club!

Cousin Ze
编译
Doki Doki λ Club!

Gemini
校对
Google Deepmind

Claude
校对
Anthropic

译者前言

本文是 Fifty Shades of OOP 的中文**编译**版本。译者基于个人判断对部分术语选择、代码示例和章节标题进行了调整，并以脚注形式添加了较多译注。所有对原文的实质性修改均已在脚注中以“译注”标注。原文观点以原作者为准。术语 *(terminology)* 在正文中第一次出现的地方以仿宋体（中文）或 *Italic (English)* 呈现，如果某个术语不易辨识，则总是会以仿宋体呈现。如遇翻译或排版质量问题，请在 https://github.com/chuigda/CoreBlogPHP-NG/issues 向译者报告。

本文另有 Amisto 的译本。

前言

如今，唱衰面向对象程序设计 *(Object Oriented Programming, OOP)* 似乎成为了某种潮流。在接连看到 Lobsters 上的两篇关于面向对象程序设计的文章之后，我决定写下这篇文章。我无意捍卫或是抨击面向对象程序设计，但我希望能发表一些浅见，提供一个更细致入微的视角。

工业界和学术界用“面向对象”一词来表示太多不同的含义。而相关讨论之所以收效甚微，正是因为人们对“面向对象程序设计究竟是什么”各执一词。

何谓面向对象程序设计？维基百科将其定义为“基于对象概念的程序设计范式”。这一定义并不尽如人意，它既依赖于“对象”的进一步定义，也未能涵盖这一术语在工业界五花八门的用法。Alan Kay 也提出过他对面向对象程序设计的构想。然而，大多数人使用这一术语的方式现已渐行渐远。我也不想为了强求某种“真正的”含义而陷入本质主义或者词源学谬误。

> 有意思的是，本文发布之后，部分评论针对“对象”的确切定义展开了争论，且各自基于截然不同的标准，如：(i) Alan Kay 与消息传递 *(message passing)*；(ii) 任何能提供封装 *(encapsulation)* 的事物（包括闭包 *closure* 与模块 *module*）；(iii) 动态分派 *(dynamic dispatch)*；(iv) 方法 *(method)*。

与其执着于单一定义，不如把面向对象程序设计当作一系列彼此关联的思想的混合体，并逐一考察每种思想。接下来，我将考察一些和面向对象相关的思想，并（主观地）探讨其优缺点。

类

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.
>
> 面向对象程序设计是一种实现方法：程序被组织为相互协作的对象集合，每个对象代表某个类 *(class)* 的一个实例 *(instance)*，并且其类都是通过继承 *(inheritance)* 关系组织起来的类层次结构 *(hierarchy of classes)* 的成员。
>
> — Grady Booch

类 *(class)* 扩展了“结构体 *(struct)*”或者“记录 *(record)*”的概念，增加了对方法语法、信息隐藏和继承的支持。我们稍后再分别讨论这些具体特性。

类也可以视作对象的蓝图。这并非定义/组织对象的唯一方式——原型 *(prototype)* 是 Self 语言首创的方案，并因被 JavaScript 采用而广为人知。就我个人的感受而言，原型相较于类更难理解，以至于连 JavaScript 都引入了 ES6 `class` 以向初学者隐藏其背后的原型机制。

> 在学习 JavaScript 时，原型继承和 `this` 的语义是最令我困扰的两个主题。

方法语法

> In Japanese, we have sentence chaining, which is similar to method chaining in Ruby.
>
> 日语中有句子接续 *(sentence chaining,* 連用形接続*)*，这和 Ruby 中的方法链 *(method chaining)* 很像。
>
> — Yukihiro Matsumoto

方法 *(method)* 语法是面向对象程序设计中争议较少的特性之一，它准确反映了一类常见的程序设计场景：对特定主体 *(subject)* 执行操作。即使在没有方法的语言中，函数 *(function)* 也常被当作方法使用：将相关的数据作为函数的第一个参数（或者在支持柯里化 *currying* 的语言中，作为最后一个参数）。

> 我们将在讨论封装时回顾“对特定主体执行操作”（本质上即捆绑数据和行为）这一思想。

方法语法包括方法定义和方法调用。支持方法的语言一般两者都有——除非把函数式语言中的管道运算符 *(pipe operator)* 当作一种方法调用。

方法调用语法有利于 IDE 自动补全，且方法链比嵌套函数调用更符合人体工程学（类似于函数式语言中的管道运算符）。

方法语法也有值得商榷之处。首先，许多语言不允许在类外定义方法，这使得方法与函数地位不对等。也有一些例外，如 Rust（方法总是在结构体外定义的）、Scala、Kotlin 和 C♯（扩展方法）。

其次，在许多语言中，自指 `this` 或 `self` 是隐式的。这让代码更加简洁，但也可能造成混淆，并增加意外发生名称遮蔽 *(name shadowing)* 的风险。隐式自指的另一缺点则是自指总是以指针的形式传递，且其类型不能更改。这就导致自指不能被按值/按拷贝传递，而指针引入的间接性有时会导致性能问题。更重要的是，因为自指的类型是固定的，你不能编写接受不同 `this` 类型的泛型函数。Python 和 Rust 从一开始就正确地设计了自指，而 C++ 也在 C++23 中引入了 Deducing this 以解决这一问题。

第三，若语言同时支持自由函数 *(free function)* 和方法，函数和方法就成了做同一件事的两种方式，殊途同归却互不兼容，这在泛型代码中可能造成问题。Rust 允许完全限定方法名称并将其视为函数来解决这一问题。

第四，大多数语言都将点号语法 *(dot notation)* 兼用于实例变量访问和方法调用。这是有意为之，旨在让方法和实例变量[1]看起来更统一。在一些动态类型语言中，方法本就是实例变量，这么做顺理成章，甚至算不上刻意的设计。但在 C++ 和 Java 这样的语言中，这种做法就可能导致混淆，并引入名称遮蔽问题。

[1]译注：“实例变量”原作“对象”，译者依个人判断改。

信息隐藏

> Its interface or definition was chosen to reveal as little as possible about its inner workings.
>
> 接口 *(interface)* 或定义 *(definition)* 应尽可能少地透露其内部运作方式。
>
> — [Parnas, 1972b]

在 Smalltalk 中，所有实例变量都不能在对象外直接访问，而所有方法都是对外暴露的。现代的面向对象语言则通过 `private` 这样的访问说明符 *(access specifier)* 支持类级别的信息隐藏。即使是非面向对象语言，通常也支持某种形式的信息隐藏，例如模块系统、不透明类型 *(opaque type)* 乃至 C 语言的头文件分离。

信息隐藏是防止不变式 *(invariant)*[2]被破坏的有效手段，也是将频繁变动的实现细节与稳定的接口分离开来的好方法。

> 遗憾的是，我见过很多大学课程教授流于形式的 `private` 和 getter/setter 方法，却不讲论其中缘由。

尽管如此，激进地隐藏信息会增加不必要的样板代码，并且可能引发抽象倒置 *(abstraction inversion)*。另一种批评则来自函数式程序员，他们认为若数据不可变 *(immutable)*，则无须维护不变式，进而也就不需要隐藏太多信息[3]。而从某种意义上说，面向对象恰恰是在鼓励人们编写必须维护其不变式的可变对象。

> 不过，若语言对不可变数据支持不佳，“隐藏数据、仅暴露 getter 方法”确实是让对象不可变的一种方式。

信息隐藏还鼓励人们创建小巧且自包含 *(self-contained)* 的对象，让它们懂得“如何自我管理”，这就直接引出了封装这一话题。

封装

> If you can, just move all of that behavior into the class it helps. After all, OOP is about letting objects take care of themselves.
>
> 如果可以的话，把所有相关的行为都移到其服务的类中。毕竟面向对象程序设计就是要让对象自我管理。
>
> — Bob Nystrom, Game Programming Patterns

封装常与信息隐藏混淆，但它们确是不同的概念。封装指的是捆绑数据和操作数据的函数。面向对象语言通过对象/类和方法语法直接支持了封装，但也有其他的封装方式。许多现代语言也支持闭包 *(closure)*（事实上，闭包和对象可以相互模拟[4]）。还有一些不那么广为人知的方式，例如 ML 系语言中的模块系统。

面向数据设计 *(Data-oriented design, DOD)*[5] 对于“将数据和功能捆绑在一起”这一做法颇有微词。当存在大量对象时，批量处理它们通常比逐个处理要高效得多。让众多小对象各自拥有独立行为会损害数据局部性 *(data locality)*、引入更多的间接性并减少并行优化的机会。当然，面向数据设计并不完全排斥封装，而是提倡一种更粗粒度的封装形式，根据数据和功能的实际使用方式来组织代码，而不是依照领域模型在概念上的结构来组织代码。

---

[2] 译注：请注意，不变式这一术语指的**不是**不可变数据 *(immutable data)*。
[3] 译注：此观点有待商榷。即使数据不可变，在构造和操作时依然需要校验并维护逻辑上的不变式。
[4] 译注：对象是穷人的闭包，闭包是穷人的对象。
[5] 译注：请注意，面向数据设计这一术语指的**不是**数据驱动设计 *(Data-driven design)*。

接口

> No part of a complex system should depend on the internal details of any other part.
>
> 复杂系统中的任何部分都不应依赖其他部分的内部细节。
>
> — Daniel Ingalls, "The Smalltalk-76 Programming System Design and Implementation"

接口与实现分离的思想由来已久，与信息隐藏、封装和抽象数据类型 *(abstract data type)* 密切相关。某种程度上讲，即使是 C 语言的头文件也可以视作一种接口。不过面向对象语境中的"接口"通常指用以支持多态性的一组特定的语言构造（常以继承的形式实现）。接口通常不能包含数据，并且在更严格的语言（如早期 Java）中也不能包含方法实现。接口的思想也常见于非面向对象语言：Haskell 的类型类 *(type class)*、Rust 的特型 *(traits)* 和 Go 的 `interface` 都被用于描述一组独立于实现的抽象操作。

接口常被视作完整类继承的一个更简单、更规范的替代方案。接口只有一种用途，且不像多重继承那样受到菱形继承问题的困扰。

接口在与参数化多态 *(parametric polymorphism)* 合用时尤其有价值，它能约束类型参数，限制其必须支持某些操作。动态类型语言（以及 C++/D 模板）通过鸭子类型实现了类似的效果，但即使是支持鸭子类型的语言，往往也会在后期引入接口结构（如 C++ 的 `concept` 或 TypeScript 的 `interface`）以更明确地表达约束。

面向对象语言中实现的接口通常有运行时开销，但也不总是如此。例如，C++ 的 `concept` 只能用于编译期约束，而 Rust 的特型则仅通过 `dyn` 提供可选的运行时多态支持。

延迟绑定

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.
>
> 于我而言，"面向对象程序设计"这个词的意思只有：消息传递，对内部状态和行为的保留、保护和隐藏，以及对所有事物的极度延迟绑定。
>
> — Alan Kay

延迟绑定指的是将方法或成员的查找推迟到运行时。这是大部分动态类型语言的默认行为，在这些语言中，方法调用常被实现成哈希表查找，但也有其他的实现方式，如动态加载和函数指针。

延迟绑定的关键特性之一是软件可在运行时改变其行为，从而支持各种热重载和猴子补丁 *(monkey-patching)*。例如，考虑以下 Python 代码：

```python
def bar():
    return foo() + 1

def foo():
    return 42

print(bar()) # 43

def foo():
    return 100

print(bar()) # 101
```

注意到，当定义 bar 时，foo 尚未被定义；当 bar 被调用时，foo 的名称查找才发生。当然，“使用先于定义”在其他语言中也可通过支持互递归的语言结构（如前向声明或 letrec）来实现。

> 延迟绑定天然地支持互递归函数调用，我们将在*开放递归 (open recursion)* 中进一步讨论这一点。

例子的第二部分才是重头戏：在第二次调用 bar 之前，foo 的实现被修改了，而调用 bar 会自动反映这一变化。如果没有延迟绑定，这是无法实现的。

延迟绑定的缺点在于其不容忽视的性能开销。此外，它还可能成为破坏不变式、甚至导致接口不匹配的隐患。其可变性也可能引入一些更微妙的问题，例如 Python 中著名的“延迟绑定闭包”陷阱。

动态分派

> A programming paradigm in C++ using Polymorphism based on runtime function dispatch using virtual functions.
>
> 这是一种 C++ 编程范式，它利用多态机制，通过虚函数来实现运行时的函数分派。
>
> — Back to Basics: Object-Oriented Programming - Jon Kalb - CppCon 2019

动态分派是一个与延迟绑定相关的概念，它指的是在运行时选择多态操作的具体实现。两个概念有所重叠，但动态分派更侧重于在多个已知的多态操作实现中作选择，而非运行时名称查找。

在动态类型语言中，一切都是延迟绑定的，动态分派自然就是默认行为。而在静态类型语言中，动态分派常通过*虚函数表 (virtual function table/virtual table/vtable)* 实现，其底层结构大致如下：

```
struct VTable {
  // 用于销毁 BaseClass 对象的函数指针
  void (*destroy)(BaseClass&);

  // 指向某个方法实现的函数指针
  void (*foo)(void);

  // 指向另一个方法实现的函数指针
  int (*bar)(int);
};

struct BaseClass {
  VTable* vtable;
};
```

这些语言还在编译时保证虚函数表包含该类型的有效操作。

动态分派可与继承解耦。例如，动态分派可以靠手动构造虚函数表来实现（如 Rust 的 `RawWaker` / `RawWakerVTable` 类型[6]），也可以用接口/特型/类型类这种语言结构来实现。不使用继承的动态分派通常不被称作“面向对象”。

另一点需要注意的是，指向虚函数表的指针可以直接位于对象内部（如 C++），也可嵌入到宽指针 *(wide pointer)* 中（如 Go 和 Rust）。

对动态分派的抱怨主要集中于性能问题：尽管虚函数调用本身很快，但虚函数会阻碍内联、降低缓存命中率和分支预测成功率。

[6]译注：原文举例为 C++ 的 `std::function`，译者依个人判断改。

继承

> Programming using class hierarchies and virtual functions to allow manipulation of objects of a variety of types through well-defined interfaces and to allow a program to be extended incrementally through derivation.
>
> 利用类层次结构和虚函数进行编程，可以通过定义良好的接口操作各种类型的对象，并允许程序通过派生 *(derivation)* 进行增量扩展。
>
> — Bjarne Stroustrup

继承[7]的历史源远流长，上可追溯到 Simula 67。它可能是面向对象程序设计最具标志性的特性：几乎所有标榜“面向对象”的语言都包含它，而规避面向对象程序设计的语言则通常省略它。

有时候继承确实很[redacted]方便——而其他写法会显著增加样板代码[8]。

> 我惭愧地承认，我写 Rust 的时候会时不时地怀念继承。

但另一方面，继承是一个非常不正交 *(non-orthogonal)* 的特性，它融合了动态分派、子类型多态、接口/实现分离和代码复用。它很灵活，但灵活性使其易被误用，故一些现代语言倾向于用更严格的语言结构取而代之。

继承还有一些其他问题。首先，使用继承几乎肯定意味着要承担动态分派和堆分配带来的性能开销。在某些语言——例如 C++——中，继承可以在没有动态分派和堆分配的情况下使用，并且也存在一些合理的用例（例如用 *CRTP* 实现代码复用）。但继承的主流用途确是运行时多态（因此也依赖动态分派）。

其次，继承以一种不够严谨[9]的方式实现了子类型，里氏替换原则 (Liskov substitution principle) 的执行全靠程序员自觉[10]。

最后，继承结构是刚性的，会受到对角线问题 *(diagonal problem)*[11] 等问题的困扰。这些不灵活之处正是“组合优于继承”这一设计理念流行的重要原因之一。《游戏编程模式》中的“组件模式”一章给出了一个很好的例子。

子类型

> If for each object o_1 of type S there is another object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.
>
> 若对于任意 S 类型的对象 o_1，都存在 T 类型的对象 o_2，使得对于所有以 T 定义的程序 P，当用 o_1 替换 o_2 时，P 的行为不变，则称 S 是 T 的子类型。
>
> — Barbara Liskov, “Data Abstraction and Hierarchy”

[7] 译注：“传统”的面向对象理论常把继承分为接口继承 *(interface inheritance)* 和 实现继承 *(implementation inheritance)*。而本节的主题主要是实现继承。

[8] 译注：值得一提的是，即便是臭名昭著的多重继承也有其用武之地。

[9] 译注：此处原作“unsound”，直译为“不健全”。严格来说，继承实现的子类型在类型系统层面是健全 *(sound)* 的，只是语言无法在类型层面保证里氏替换原则这一语义性质。故依译者个人判断意译。

[10] 译注：当然，Rust 的特型、Go/TypeScript 的接口乃至 Haskell 的类型类在这方面也是一样的，契约的执行事实上也全靠程序员自觉。结构子类型还存在微妙的名义性问题，使用不当则容易引出许多的危险。

[11] 译注：译者未能找到这个词汇的确切定义和出处，推测它 (i) 可能是拼错了菱形 (diamond) 继承问题；(ii) 可能是指表达式问题 *(expression problem)*；(iii) 可能是指多维正交分类问题。

[12]子类型描述了两种**类型**之间的“是 *(is-a)*”关系。里氏替换原则定义了安全的子类型关系必须满足的属性。

面向对象语言通常通过继承来支持子类型。但请注意，继承并不总是对子类型关系的建模，也不是子类型关系的唯一形式。许多非面向对象语言中的接口/特型等结构都支持子类型。而且除了显式声明子类型关系的名义子类型 *(nominal subtyping)* 之外，还有结构子类型 *(structural subtyping)*：若一个类型 S 包含了另一个类型 T 的全部特性，则 S 就隐式地成为了 T 的子类型。OCaml 中的对象和多态变体、TypeScript 中的 `interface` 都是结构子类型的优秀案例。子类型还体现在各种细微之处，例如 Rust 的生存期 *(lifetime)*[13]、TypeScript 的可空性[14]以及依值类型语言中的类型宇宙层级[15]。

型变 *(variance，包括协变 covariance 和逆变 contravariance)*[16] 是与子类型相关的重要概念之一，它连接了参数化多态和子类型。解释这一概念需要一整篇文章，故此处不再赘述[17]。型变极大地提高了子类型的易用性（例如，如果 C++ 指针不是协变的，它们就无法多态地使用了），但大多数语言都只实现了有限的、硬编码的型变规则，因其难以理解且容易出错。特别地，可变数据类型 *(mutable data type)* 通常应该是无型变 *(invariance / invariant)*[18] 的，而 Java/C♯ 的协变数组类型就是典型的反面教材。只有少数语言允许程序员显式控制型变，如 Scala 和 Kotlin。

经由子类型关系的类型转换通常是隐式的。虽说隐式转换声名狼藉，但从子类型隐式转换到超类型确实符合人体工程学，并且可能是最合乎情理的隐式转换。子类型也可看作隐式转换的对偶：隐式转换可被用于“仿造”子类型关系。例如，C++ 模板是无型变的，但 `std::unique_ptr` 却允许从 `std::unique_ptr<Derived>` 隐式转换到 `std::unique_ptr<Base>`，从而实现了协变的效果。《Go 有子类型吗?》这篇文章很好地进一步探讨了这一思想。

实现上的复杂度是语言设计者们试图规避子类型的原因之一。整合双向类型推断 *(bidirectional type inference)* 和子类型难如登天。Stephen Dolan 于 2016 年发表的博士论文《代数子类型》在这一问题上取得了长足的进步。

消息传递

> I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages.
>
> 我把对象看作生物细胞或是网络上相互独立的计算机，它们之间只能通过消息通信。
>
> — Alan Kay

消息传递是指使用相互发送“消息”的对象来执行操作。这是 Alan Kay 的面向对象程序设计理念的核心，虽说这个定义相当模糊。需要注意的是，消息名是延迟绑定的，并且消息的结构未必在编译期确定。

许多早期面向对象概念受到过分布式和模拟系统的启发，在这些系统中，消息传递是自然而然的。然而，在大多数人都还写单线程代码的年代，这一理念在 C++ 和 Java 等语言中逐渐被遗忘。与消息传递相比，方法语法的优势有限（Bjarne Stroustrup 肯定从 Simula 那里了解到了消息传递的思想，但在实践中需要考虑如何高效实现）。真正的消息传递仍然存在，但仅限于特定领域，如进程间通信和高度事件驱动 *(event-driven)* 的系统。

---

[12]译注：本章标题原作子类型多态 *(Subtyping polymorphism)*，译者依个人判断改。
[13]译注：允许将长生存期类型当作短生存期类型使用。
[14]译注：允许将不可空类型当作可空类型使用。
[15]译注：允许将低层宇宙的类型当作高层宇宙的类型使用；这个例子是译者自己加的。
[16]和不变式无关。
[17]译注：不过译者碰巧找到了一篇极好的文章。
[18]译注：也和不变式/不可变数据无关。

消息传递在并发编程中迎来了复兴。讽刺的是，这复兴并非来自面向对象语言，而是来自 Erlang 和 Go 等非面向对象语言，通过参与者 *(actor)* 和信道 *(channel)* 之类的结构实现。这种无共享 *(shared-nothing)* 并发机制消除了一系列数据竞争和竞态错误。结合监督机制，参与者模式还提供了容错性，一个参与者发生故障不会影响整个程序。

开放递归

> In general, a rule of thumb is: use classes and objects in situations where open recursion is a big win.
>
> 一般来说，经验法则是：若开放递归能带来显著优势，则使用类和对象。
>
> — Real World OCaml

开放递归的概念源于著名的《类型与程序设计语言》一书，却可能是本文中最鲜为人知的术语。然而，这一术语描述的不过是面向对象中的常见特性之一：对象的方法可以相互调用，即使它们在继承层次结构中的不同类中定义。

这一术语有一定误导性，因为“开放递归”中未必存在递归调用。这里“递归”一词指的是*互递归 (mutually recursive)*[19]，而“开放”指的则是“对扩展开放”，通常通过继承来实现。举个例子最容易理解：

```
struct Animal {
  void print_name() const {
    // 此时函数 name 尚未在类 Animal 中定义
    std::print("{}\n", name());
  }

  virtual std::string name() const = 0;
};

struct Cat: Animal {
  std::string name() const override {
    return "Kitty";
  }
};

int main() {
  Cat cat;
  // print_name 并未在类 Cat 中定义
  cat.print_name();
}
```

这个例子原本是用 Python 写的。但 Python 的延迟绑定语义让我无法进一步阐述观点，故改用 C++。

熟悉面向对象程序设计的人可能会理所当然地认为开放递归是天赋人权，甚至察觉不到这是一个独立的概念。然而，不是所有的语言结构都有这种属性。例如，在许多语言中，函数默认不是互递归的：

[19]译注：或者说，互调用。

```cpp
// 在 C++ 中不能编译，因为 name 尚未被定义
void print_name(const Animal& animal) {
    std::print("{}\n", name(animal));
}

std::string name(const Cat& cat) {
    return "Kitty";
}
```

而在支持延迟绑定函数的语言（如 Python 和 JavaScript）中，同模块内的函数总是能互调用。在另一些语言（如 Rust）中，函数默认就是支持互递归的。还有一些语言通过前向声明（C）或 `letrec` 结构（Scheme 和 ML 系语言）来支持互递归。这解决了“递归”的部分，但“开放”的部分尚未解决：

```cpp
std::string name(const Cat& cat);

void print_name(const Animal& animal) {
    // 还是不能编译，因为 Animal 不能向下转型成 Cat
    std::print("{}\n", name(animal));
}

std::string name(const Cat& cat) {
    return "Kitty";
}
```

这一问题可以用回调函数解决：

```cpp
struct Animal {
  std::function<std::string()> get_name;

  void print_name() const {
    std::print("{}\n", get_name());
  }
};

Animal make_cat() {
    return Animal {
      .get_name = []() { return "Kitty"; },
    };
}

int main() {
  Animal cat = make_cat();
  cat.print_name();
}
```

哒哒~ 我们刚刚重新发明了原型风格的分派！

总之，我想通过上面的简单示例说明，开放递归是面向对象程序设计自带的特性，而在没有内置支持的语言中实现开放递归会比较棘手。开放递归允许分别定义对象中相互依赖的部分，这一特性有很多应用场景。例如，装饰器模式 *(decorator pattern)* 的思想就依赖于开放递归。

面向对象最佳实践

或许人们对面向对象程序设计最常见的抱怨并非针对具体的语言特性，而是针对它所提倡的程序设计风格。许多实践被当作放之四海而皆准的最佳实践来教授，有时还会给出理由，但其弊端却往往被忽略。以下是一些例子：

实践	优点	缺点
优先选择（子类型）多态，而非带标签联合体 *(tagged union)* / `if` / `switch` / 模式匹配	• 对扩展开放 • 更易添加新的子类型 / 分支	• 性能开销 • 彼此相关的行为分散于多处 • 很难在一个地方看到完整的控制流 • 表达式问题意味着更难添加新的方法 / 行为
将所有数据成员设为私有	保护类不变式	• 更多样板代码 • 如果没有不变式，通常无须隐藏数据 • 在没有属性语法的语言中，getter/setter 不如直接访问实例变量方便
偏好小型的“自管理”对象，而不是集中式的“管理者”	• 更难违反不变式 • 代码组织更清晰	• 潜在的数据局部性损害 • 阻碍并行化 • 重复引用共享数据（“反向指针 *back pointer*”）
扩展而非修改 / 开-闭原则	• 防止新特性破坏旧特性 • 防止破坏 API	• 没有必要“封闭”受控使用的非公共模块 • 不必要的复杂性和继承链 • 设计不良的接口不能改变 • 可能导致抽象倒置
针对抽象接口工作，而非针对具体实现工作	系统更易于替换、更易测试	• 过度使用会降低代码可读性和可调试性 • 额外的间接调用会增加性能开销

这篇文章已经很长，故此处不再展开更多细节。你可以对以上列出的“优缺点”提出不同意见。我想传达的核心观点是：几乎所有这些实践都有权衡和折衷[20]。

结语

恭喜你读完了这篇文章！我还有一些想讨论的话题，比如资源获取即初始化 *(Resource Acquisition Is Initialisation, RAII)* 和设计模式。但这篇文章已经够长了，所以就留给你自己去探索吧。

查看 Lobsters 上的相关讨论。

[20] 译注：“一切有为法，如梦幻泡影，如露亦如电，应作如是观。”