[TOC]

附录: 编程指南

本附录包含了有助于指导你进行低级程序设计和编写代码的建议。

当然，这些只是指导方针，而不是规则。我们的想法是将它们用作灵感，并记住偶尔会违反这些指导方针的特殊情况。

设计

1. **优雅总是会有回报**。从短期来看，似乎需要更长的时间才能找到一个真正优雅的问题解决方案，但是当该解决方案第一次应用并能轻松适应新情况，而不需要数小时，数天或数月的挣扎时，你会看到奖励（即使没有人可以测量它们）。它不仅为你提供了一个更容易构建和调试的程序，而且它也更容易理解和维护，这也正是经济价值所在。这一点可以通过一些经验来理解，因为当你想要使一段代码变得优雅时，你可能看起来效率不是很高。抵制急于求成的冲动，它只会减慢你的速度。

2. **先让它工作，然后再让它变快**。即使你确定一段代码非常重要并且它是你系统中的主要瓶颈 **，也是如此。不要这样做。使用尽可能简单的设计使系统首先运行。然后如果速度不够快，请对其进行分析。你几乎总会发现 “你的” 瓶颈不是问题。节省时间，才是真正重要的东西。

3. **记住 “分而治之” 的原则**。如果所面临的问题太过混乱 **，就去想象一下程序的基本操作，因为存在一个处理困难部分的神奇 “片段”（piece）。该 “片段” 是一个对象，编写使用该对象的代码，然后查看该对象并将其困难部分封装到其他对象中，等等。

4. **将类创建者与类用户（客户端程序员）分开**。类用户是 “客户”，不需要也不想知道类幕后发生了什么。类创建者必须是设计类的专家，他们编写类，以便新手程序员都可以使用它，并仍然可以在应用程序中稳健地工作。将该类视为其他类的*服务提供者*（service provider）。只有对其它类透明，才能很容易地使用这个类。

5. **创建类时，给类起个清晰的名字，就算不需要注释也能理解这个类**。你的目标应该是使客户端程序员的接口在概念上变得简单。为此，在适当时使用方法重载来创建直观，易用的接口。

6. **你的分析和设计必须至少能够产生系统中的类、它们的公共接口以及它们与其他类的关系，尤其是基类**。如果你的设计方法产生的不止于此，就该问问自己，该方法生成的所有部分是否在程序的生命周期内都具有价值。如果不是，那么维护它们会很耗费精力。对于那些不会影响他们生产力的东西，开发团队的成员往往不会去维护，这是许多设计方法都没有考虑的生活现实。

7. **让一切自动化**。首先在编写类之前，编写测试代码，并将其与类保持一致。通过构建工具自动运行测试。你可能会使用事实上的标准 Java 构建工具 Gradle。这样，通过运行测试代码可以自动验证任何更改，将能够立即发现错误。因为你知道自己拥有测试框架的安全网，所以当发现需要时，可以更大胆地进行彻底的更改。请记住，语言的巨大改进来自内置的测试，包括类型检查，异常处理等，但这些内置功能很有限，你必须完成剩下的工作，针对具体的类或程序，去完善这些测试内容，从而创建一个强大的系统。

8. **在编写类之前，先编写测试代码，以验证类的设计是完善的**。如果不编写测试代码，那么就不知道类是什么样的。此外，通过编写测试代码，往往能够激发出类中所需的其他功能或约束。而这些功能或约束并不总是出现在分析和设计过程中。测试还会提供示例代码，显示了如何使用这个类。

9. **所有的软件设计问题，都可以通过引入一个额外的间接概念层次（extra level of conceptual indirection）来解决**。这个软件工程的基本规则[1]是抽象的基础，是面向对象编程的主要特征。在面向对象编程中，我们也可以这样说："如果你的代码太复杂，就要生成更多的对象。"

10. **间接（indirection）应具有意义（与准则 9 一致）**。这个含义可以简单到 "将常用代码放在单个方法中。" 如果添加没有意义的间接（抽象，封装等）级别，那么它就像没有足够的间接性那样糟糕。

11. **使类尽可能原子化**。为每个类提供一个明确的目的，它为其他类提供一致的服务。如果你的类或系统设计变得过于复杂，请将复杂类分解为更简单的类。最直观的指标是尺寸大小，如果一个类很大，那么它可能是做的事太多了，应该被拆分。建议重新设计类的线索是：

    - 一个复杂的 *switch* 语句：考虑使用多态。
    - 大量方法涵盖了很多不同类型的操作：考虑使用多个类。
    - 大量成员变量涉及很多不同的特征：考虑使用多个类。

[1] Andrew Koenig 向我解释了它。

- 其他建议可以参见 Martin Fowler 的 *Refactoring: Improving the Design of Existing Code*（重构：改善既有代码的设计）（Addison-Wesley 1999）。

12. **注意长参数列表**。那样方法调用会变得难以编写，读取和维护。相反，尝试将方法移动到更合适的类，并且（或者）将对象作为参数传递。

13. **不要重复自己**。如果一段代码出现在派生类的许多方法中，则将该代码放入基类中的单个方法中，并从派生类方法中调用它。这样不仅可以节省代码空间，而且可以轻松地传播更改。有时，发现这个通用代码会为接口添加有价值的功能。此指南的更简单版本也可以在没有继承的情况下发生：如果类具有重复代码的方法，则将该重复代码放入一个公共方，法并在其他方法中调用它。

14. **注意 *switch* 语句或链式 *if-else* 子句**。一个*类型检查编码*（type-check coding）的指示器意味着需要根据某种类型信息选择要执行的代码（确切的类型最初可能不明显）。很多时候可以用继承和多态替换这种代码，多态方法调用将会执行类型检查，并提供了更可靠和更容易的可扩展性。

15. **从设计的角度，寻找和分离那些因不变的事物而改变的事物**。也就是说，在不强制重新设计的情况下搜索可能想要更改的系统中的元素，然后将这些元素封装在类中。

16. **不要通过子类扩展基本功能**。如果一个接口元素对于类来说是必不可少的，则它应该在基类中，而不是在派生期间添加。如果要在继承期间添加方法，请考虑重新设计。

17. **少即是多**。从一个类的最小接口开始，尽可能小而简单，以解决手头的问题，但不要试图预测类的所有使用方式。在使用该类时，就将会了解如何扩展接口。但是，一旦这个类已经在使用了，就无法在不破坏客户端代码的情况下缩小接口。如果必须添加更多方法，那很好，它不会破坏代码。但即使新方法取代旧方法的功能，也只能是保留现有接口（如果需要，可以结合底层实现中的功能）。如果必须通过添加更多参数来扩展现有方法的接口，请使用新参数创建重载方法，这样，就不会影响到对现有方法的任何调用。

18. **大声读出你的类以确保它们合乎逻辑**。将基类和派生类之间的关系称为 “is-a”，将成员对象称为 “has-a”。

19. **在需要在继承和组合之间作决定时，问一下自己是否必须向上转换为基类型**。如果不是，则使用组合（成员对象）更好。这可以消除对多种基类型的感知需求（perceived need）。如果使用继承，则用户会认为他们应该向上转型。

20. **注意重载**。方法不应该基于参数的值而有条件地执行代码。在这里，应该创建两个或多个重载方法。

21. **使用异常层次结构**，最好是从标准 Java 异常层次结构中的特定适当类派生。然后，捕获异常的人可以为特定类型的异常编写处理程序，然后为基类型编写处理程序。如果添加新的派生异常，现有客户端代码仍将通过基类型捕获异常。

22. **有时简单的聚合可以完成工作**。航空公司的 "乘客舒适系统" 由独立的元素组成：座位，空调，影视等，但必须在飞机上创建许多这样的元素。你创建私有成员并建立一个全新的接口了吗？如果不是，在这种情况下，组件也应该是公共接口的一部分，因此应该创建公共成员对象。这些对象有自己的私有实现，这些实现仍然是安全的。请注意，简单聚合不是经常使用的解决方案，但确实会有时候会用到。

23. **考虑客户程序员和维护代码的人的观点**。设计类以便尽可能直观地被使用。预测要进行的更改，并精心设计类，以便轻松地进行更改。

24. **注意 "巨型对象综合症"**（giant object syndrome）。这通常是程序员的痛苦，他们是面向对象编程的新手，总是编写面向过程程序并将其粘贴在一个或两个巨型对象中。除应用程序框架外，对象代表应用程序中的概念，而不是应用程序本身。

25. **如果你必须做一些丑陋的事情，至少要把类内的丑陋本地化**。

26. **如果必须做一些不可移植的事情，那就对这个事情做一个抽象，并在一个类中进行本地化**。这种额外的间接级别可防止在整个程序中扩散这种不可移植性。（这个原则也体现在桥接模式中，等等）。

27. **对象不应该仅仅只是持有一些数据**。它们也应该有明确的行为。有时候，"数据传输对象"（data transfer objects）是合适的，但只有在泛型集合不合适时，才被明确用于打包和传输一组元素。

28. **在从现有类创建新类时首先选择组合**。仅在设计需要时才使用继承。如果在可以使用组合的地方使用继承，那么设计将会变得很复杂，这是没必要的。

29. **使用继承和覆盖方法来表达行为的差异，而不是使用字段来表示状态的变化**。如果发现一个类使用了状态变量，并且有一些方法是基于这些变量切换行为的，那么请重新设计它，以表示子类和覆盖方法中的行为差异。一个极端的反例是继承不同的类来表示颜色，而不是使用 "颜色" 字段。

30. **注意协变（variance）**。两个语义不同的对象可能具有相同的操作或职责。为了从继承中受益，会试图让其中一个成为另一个的子类，这是一种很自然的诱惑。这

称为协变，但没有真正的理由去强制声明一个并不存在的父子类关系。更好的解决方案是创建一个通用基类，并为两者生成一个接口，使其成为这个通用基类的派生类。这仍然可以从继承中受益，并且这可能是关于设计的一个重要发现。

31. **在继承期间注意限定（limitation）**。最明确的设计为继承的类增加了新的功能。含糊的设计在继承期间删除旧功能而不添加新功能。但是规则是用来打破的，如果是通过调用一个旧的类库来工作，那么将一个现有类限制在其子类型中，可能比重构层次结构更有效，因此新类适合在旧类的上层。

32. **使用设计模式来消除“裸功能”（naked functionality）**。也就是说，如果类只需要创建一个对象，请不要推进应用程序并写下注释“只生成一个。”应该将其包装成一个单例（singleton）。如果主程序中有很多乱七八糟的代码去创建对象，那么找一个像工厂方法一样的创建模式，可以在其中封装创建过程。消除“裸功能”不仅会使代码更易于理解和维护，而且还会使其能够更加防范应对后面的善意维护者（well-intentioned maintainers）。

33. **注意“分析瘫痪”（analysis paralysis）**。记住，不得不经常在不了解整个项目的情况下推进项目，并且通常了解那些未知因素的最好、最快的方式是进入下一步而不是尝试在脑海中弄清楚。在获得解决方案之前，往往无法知道解决方案。Java有内置的防火墙，让它们为你工作。你在一个类或一组类中的错误不会破坏整个系统的完整性。

34. **如果认为自己有很好的分析，设计或实施，请做一个演练**。从团队外部带来一些人，不一定是顾问，但可以是公司内其他团体的人。用一双新眼睛评审你的工作，可以在一个更容易修复它们的阶段发现问题，而不仅仅是把大量时间和金钱全扔到演练过程中。

实现

36. **遵循编码惯例**。有很多不同的约定，例如，谷歌使用的约定（本书中的代码尽可能地遵循这些约定）。如果坚持使用其他语言的编码风格，那么读者就会很难去阅读。无论决定采用何种编码约定，都要确保它们在整个项目中保持一致。集成开发环境通常包含内置的重新格式化（reformatter）和检查器（checker）。

37. **无论使用何种编码风格，如果你的团队（甚至更好是公司）对其进行标准化，它就确实会产生重大影响**。这意味着，如果不符合这个标准，那么每个人都认为修复别人的编码风格是公平的游戏。标准化的价值在于解析代码可以花费较少的脑力，因此可以更专注于代码的含义。

38. **遵循标准的大写规则**。类名的第一个字母大写。字段，方法和对象（引用）的第一个字母应为小写。所有标识符应该将各个单词组合在一起，并将所有中间单词的首字母大写。例如：

    - **ThisIsAClassName**
    - **thisIsAMethodOrFieldName**

    将 **static final** 类型的标识符的所有字母全部大写，并用下划线分隔各个单词，这些标识符在其定义中具有常量初始值。这表明它们是编译时常量。

    - **包是一个特例**，它们都是小写的字母，即使是中间词。域扩展（com，org，net，edu 等）也应该是小写的。这是 Java 1.1 和 Java 2 之间的变化。

39. **不要创建自己的 “装饰” 私有字段名称**。这通常以前置下划线和字符的形式出现。匈牙利命名法（译者注：一种命名规范，基本原则是：变量名 = 属性 + 类型 + 对象描述。Win32 程序风格采用这种命名法，如 `WORD wParam1;LONG lParam2;HANDLE hInstance`）是最糟糕的例子，你可以在其中附加额外字符用于指示数据类型，用途，位置等，就好像你正在编写汇编语言一样，编译器根本没有提供额外的帮助。这些符号令人困惑，难以阅读，并且难以执行和维护。让类和包来指定名称范围。如果认为必须装饰名称以防止混淆，那么代码就可能过于混乱，这应该被简化。

40. 在创建一般用途的类时，**遵循 “规范形式”**。包括 **equals()**，**hashCode()**，**toString()**，**clone()** 的定义（实现 **Cloneable**，或选择其他一些对象复制方法，如序列化），并实现 **Comparable** 和 **Serializable**。

41. **对读取和更改私有字段的方法使用 “get”，“set” 和 “is” 命名约定**。这种做法不仅使类易于使用，而且也是命名这些方法的标准方法，因此读者更容易理解。

42. **对于所创建的每个类，请包含该类的 JUnit 测试**（请参阅 *junit.org* 以及第十六章：代码校验中的示例）。无需删除测试代码即可在项目中使用该类，如果进行更改，则可以轻松地重新运行测试。测试代码也能成为如何使用这个类的示例。

43. **有时需要继承才能访问基类的 protected 成员**。这可能导致对多种基类型的感知需求（perceived need）。如果不需要向上转型，则可以首先派生一个新类来执行受保护的访问。然后把该新类作为使用它的任何类中的成员对象，以此来代替直接继承。

44. **为了提高效率，避免使用 *final* 方法**。只有在分析后发现方法调用是瓶颈时，才将 **final** 用于此目的。

45. **如果两个类以某种功能方式相互关联（例如集合和迭代器），则尝试使一个类成为另一个类的内部类**。这不仅强调了类之间的关联，而且通过将类嵌套在另一个类中，可以允许在单个包中重用类名。Java 集合库通过在每个集合类中定义内部 **Iterator** 类来实现此目的，从而为集合提供通用接口。使用内部类的另一个原因是作为**私有**实现的一部分。这里，内部类将有利于实现隐藏，而不是上面提到的类关联和防止命名空间污染。

46. **只要你注意到类似乎彼此之间具有高耦合，请考虑如果使用内部类可能获得的编码和维护改进**。内部类的使用不会解耦类，而是明确耦合关系，并且更方便。

47. **不要成为过早优化的牺牲品**。过早优化是很疯狂的行为。特别是，不要担心编写（或避免）本机方法（native methods），将某些方法设置为 **final**，或者在首次构建系统时调整代码以使其高效。你的主要目标应该是验证设计。即使设计需要一定的效率，也先让它工作，然后再让它变快。

48. **保持作用域尽可能小，以便能见度和对象的寿命尽可能小**。这减少了在错误的上下文中使用对象并隐藏了难以发现的 bug 的机会。例如，假设有一个集合和一段迭代它的代码。如果复制该代码以用于一个新集合，那么可能会意外地将旧集合的大小用作新集合的迭代上限。但是，如果旧集合比较大，则会在编译时捕获错误。

49. **使用标准 Java 库中的集合**。熟练使用它们，将会大大提高工作效率。首选 **ArrayList** 用于序列，**HashSet** 用于集合，**HashMap** 用于关联数组，**LinkedList** 用于堆栈（而不是 **Stack**，尽管也可以创建一个适配器来提供堆栈接口）和队列（也可以使用适配器，如本书所示）。当使用前三个时，将其分别向上转型为 **List**，**Set** 和 **Map**，那么就可以根据需要轻松更改为其他实现。

50. **为使整个程序健壮，每个组件必须健壮**。在所创建的每个类中，使用 Java 所提供的所有工具，如访问控制，异常，类型检查，同步等。这样，就可以在构建系统时安全地进入下一级抽象。

51. **编译时错误优于运行时错误**。尝试尽可能在错误发生点处理错误。在最近的处理程序中尽其所能地捕获它能处理的所有异常。在当前层面处理所能处理的所有异常，如果解决不了，就重新抛出异常。

52. **注意长方法定义**。方法应该是简短的功能单元，用于描述和实现类接口的离散部分。维护一个冗长而复杂的方法是很困难的，而且代价很大，并且这个方法可能是试图做了太多事情。如果看到这样的方法，这表明，至少应该将它分解为多种

方法。也可能建议去创建一个新类。小的方法也可以促进类重用。（有时方法必须很大，但它们应该只做一件事。）

53. **保持“尽可能私有”**。一旦公开了你的类库中的一个方面（一个方法，一个类，一个字段），你就永远无法把它拿回来。如果这样做，就将破坏某些人的现有代码，迫使他们重写和重新设计。如果你只公开了必须公开的内容，就可以轻易地改变其他一切，而不会对其他人造成影响，而且由于设计趋于发展，这是一个重要的自由。通过这种方式，更改具体实现将对派生类造成的影响最小。在处理多线程时，私有尤其重要，只有**私有**字段可以防止不同步使用。具有包访问权限的类应该仍然具有**私有**字段，但通常有必要提供包访问权限的方法而不是将它们**公开**。

54. **大量使用注释，并使用 *Javadoc commentdocumentation* 语法生成程序文档**。但是，注释应该为代码增加真正的意义，如果注释只是重申了代码已经清楚表达的内容，这是令人讨厌的。请注意，Java 类和方法名称的典型详细信息减少了对某些注释的需求。

55. **避免使用“魔法数字”**。这些是指硬编码到代码中的数字。如果后续必须要更改它们，那将是一场噩梦，因为你永远不知道“100”是指“数组大小”还是“完全不同的东西”。相反，创建一个带有描述性名称的常量并在整个程序中使用常量标识符。这使程序更易于理解，更易于维护。

56. **在创建构造方法时，请考虑异常**。最好的情况是，构造方法不会做任何抛出异常的事情。次一级的最佳方案是，该类仅由健壮的类组成或继承自健壮的类，因此如果抛出异常则不需要处理。否则，必须清除 **finally** 子句中的组合类。如果构造方法必然失败，则适当的操作是抛出异常，因此调用者不会认为该对象是正确创建的而盲目地继续下去。

57. **在构造方法内部，只需要将对象设置为正确的状态**。主动避免调用其他方法（**final** 方法除外），因为这些方法可以被其他人覆盖，从而在构造期间产生意外结果。（有关详细信息，请参阅第六章：初始化和清理章节。）较小，较简单的构造方法不太可能抛出异常或导致问题。

58. **如果类在客户端程序员用完对象时需要进行任何清理，请将清理代码放在一个明确定义的方法中**，并使用像 **dispose()** 这样的名称来清楚地表明其目的。另外，在类中放置一个 **boolean** 标志来指示是否调用了 **dispose()** ，因此 **finalize()** 可以检查“终止条件”（参见第六章：初始化和清理章节）。

59. ***finalize()* 的职责只能是验证对象的“终止条件”以进行调试**。（参见第六章：初始化和清理一章）在特殊情况下，可能需要释放垃圾收集器无法释放的内存。因为

可能无法为对象调用垃圾收集器，所以无法使用 **finalize()** 执行必要的清理。为此，必须创建自己的 **dispose()** 方法。在类的 **finalize()** 方法中，检查以确保对象已被清理，如果没有被清理，则抛出一个派生自 **RuntimeException** 的异常，以指示编程错误。在依赖这样的计划之前，请确保 **finalize()** 适用于你的系统。（可能需要调用 **System.gc()** 来确保此行为。）

60. **如果必须在特定范围内清理对象（除了通过垃圾收集器），请使用以下准则：**初始化对象，如果成功，立即进入一个带有 **finally** 子句的 **try** 块，并在 **finally** 中执行清理操作。

61. **在继承期间覆盖 *finalize()* 时，记得调用 *super.finalize()***。（如果是直接继承自 **Object** 则不需要这样做。）调用 **super.finalize()** 作为重写的 **finalize()** 的最终行为而不是在第一行调用它，这样可以确保基类组件在需要时仍然有效。

62. **创建固定大小的对象集合时，将它们转换为数组，**尤其是在从方法中返回此集合时。这样就可以获得数组编译时类型检查的好处，并且数组的接收者可能不需要在数组中强制转换对象来使用它们。请注意，集合库的基类 **java.util.Collection** 有两个 **toArray()** 方法来完成此任务。

63. **优先选择 接口而不是 抽象类**。如果知道某些东西应该是基类，那么第一选择应该是使其成为一个接口，并且只有在需要方法定义或成员变量时才将其更改为抽象类。一个接口关心客户端想要做什么，而一个类倾向于关注（或允许）实现细节。

64. **为了避免非常令人沮丧的经历，请确保类路径中的每个名称只对应一个不在包中的类**。否则，编译器可以首先找到具有相同名称的其他类，并报告没有意义的错误消息。如果你怀疑自己有类路径问题，请尝试在类路径的每个起始点查找具有相同名称的 **.class** 文件。理想情况下，应该将所有类放在包中。

65. **注意意外重载**。如果尝试覆盖基类方法但是拼写错误，则最终会添加新方法而不是覆盖现有方法。但是，这是完全合法的，因此在编译时或运行时不会获得任何错误消息，但代码将无法正常工作。始终使用 **@Override** 注释来防止这种情况。

66. **注意过早优化**。先让它工作，然后再让它变快。除非发现代码的特定部分存在性能瓶颈。除非是使用分析器发现瓶颈，否则过早优化会浪费时间。性能调整所隐藏的额外成本是代码将变得难以理解和维护。

67. **请注意，相比于编写代码，代码被阅读的机会更多**。清晰的设计可能产生易于理解的程序，但注释，详细解释，测试和示例是非常宝贵的，它们可以帮助你和你的所有后继者。如果不出意外，试图从 JDK 文档中找出有用信息的挫败感应该可以说服你。