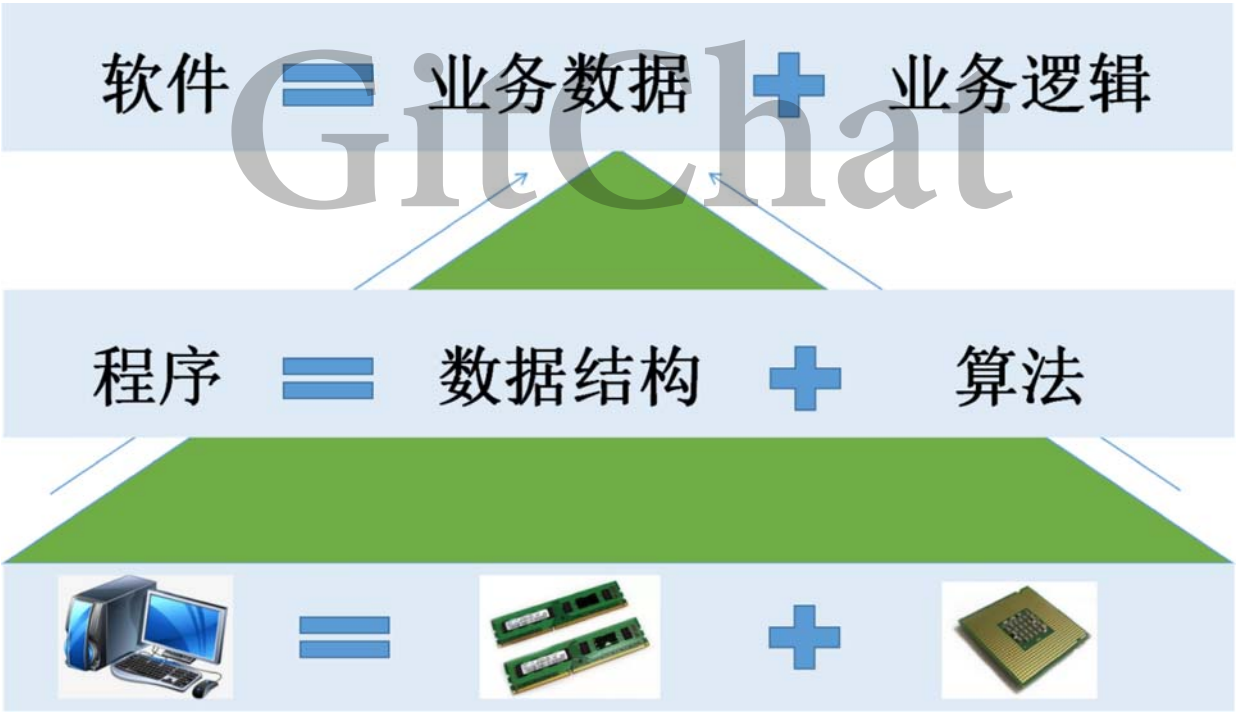
# 面向对象编程的正确姿势

> 世界新生伊始，许多事物还没有名字，提到的时候尚需用手指指点点。
>
> ——《百年孤独》

## 面向过程程序设计

计算机程序=数据结构+算法。这是大学 C 语言教材里非常经典的一句话。这也道出了计算机程序的本质，即通过对一定的数据结构用相应的算法（逻辑）进行处理从而解决用户的问题。这便是 C 之类的过程式语言站在计算机的角度看待编程及软件的经典视角。如下图。





这种基于计算机（实现）视角的软件开发过程通常是这样的：首先梳理业务数据（各种表格、单据、报表等），然后设计数据结构并用代码对其进行表示；接下来分析在真实的业务中对于每一类业务数据需要进行怎样的处理；最后将复杂的处理过程拆解成一个个相对简单的业务步骤并用函数对其进行程序化实现。这样便开发出了满足用户需求的软件。

这种方法的好处显而易见，首先直观明了——用计算机解决现实世界的问题当然要沿着计算机的套路来；其次对于一些业务逻辑不是很复杂的应用，这种方式能提供最简洁的解决方案；但对于业务逻辑比较复杂的项目，这种方式会导致“低内聚，高耦合”的不良

代码。但这种方法是如此符合直觉，以至于今天很多使用 Java 或 Python 等面向对象语言的程序员还在用这种过程式思维在写代码。只不过 C 中的结构体变成了 Java 中的“贫血类”（只有属性和 get、set 方法的类），C 中的函数变成了 Java 中的静态方法或无状态类（没有属性的类）方法。为什么会这样呢？因为大家还是站在实现的角度看待面向对象。比如某著名的 Java 教程上对类的定义——“类是具有相同特性和行为的对象集合”。这便是典型的站在实现的角度看待类的方式。那类到底应该是什么？怎样才是面向对象编程的正确姿势呢？

# 面向对象程序设计

我们进行软件开发时需要不停地在两个空间中转换角色——问题空间（包括用户角色、使用场景和具体需求）和解决方案空间（通过组合各种软件组件来满足用户需求）。面向过程方法的思路是站在解决方案空间遥望问题空间，通过将问题空间中的数据和处理流程分别映射进解决方案空间来解决问题。而面向对象则恰恰相反，它鼓励我们立足问题空间，理解需求涉及的各种概念及其职责，然后通过对象和类对其进行表达从而形成解决方案。按照面向对象的本意，**对象描述了问题领域中的某个概念并具有一定的职责**。如何理解这句话呢？


其实我们日常生活中的世界便是面向对象的。这也是面向对象技术更擅长表达问题空间的原因。举个例子，看下面某公司的组织架构图。我们看到了是 HR 部，财务部，行政部和软件开发部等，**每个部门都有自己的职责范围**。


部门有这么几个特点：

1. 提供一定的职责：比如 HR 部门拥有绩效考核，员工技能提升，新人招聘等职责；
2. 自治：比如当 HR 部门接到招聘新人的请求后，会安排部门内部相关人员通过一定的途径从社会上进行招聘。至于安排什么人通过什么途径去招聘，请求发起方统统不用管；
3. （从实现层面来说）拥有资源和流程，这是履行职责的必要条件。

在面向对象编程中我们设计的类和对象也应该具有这三个特点。在一个具体的项目中，我们该如何定义一个类呢？很多面向对象设计的书籍给出的方法是在项目的需求描述中寻找名字。但这种方法存在的问题是对于业务逻辑很复杂的项目，这些名词只能反映浅层的领域知识。如果想提炼出合理的领域概念，从而设计出一个良好的领域模型，通常需要通过多轮迭代精炼才能做到。而这也是《领域驱动设计》这本书里作者花费了大量篇幅想要告诉我们的。如作者所担心的，很多程序员的问题是将大部分精力都投入在研究技术本身，而忽略了问题领域本身。而后者才是《人月神话》中所谓的软件开发中的根本的困难。作者说到“我认为软件开发中困难的部分是规格说明，设计和测试这些概念上的结构，而不是对概念进行表达和对实现逼真程度进行验证。”让我们从发现概念开始。

## 发现概念

如果你对准确认识问题领域中的概念及其相互关系的困难程度还不以为然的话，让我们再来看一个例子。

下面是我在巴黎拍的一幅照片，从中你看到了什么？


头顶的蓝天和云朵，远处的树木，中间的方尖碑和下面的栏杆等等。我们能轻易识别出的概念有具体的蓝天，云朵，树木，方尖碑和栏杆，同时我们也能识别出更为抽象的空间和距离等概念。这一切是不是很平常？其实一点都不。让我们来看《火星上的人类学家》里提到的一个故事。维吉自幼失明，50岁时哈姆林医生为他做了白内障摘除手术从而得以重见光明。然而，

> 在他初看见的那一刻，他完全不了解自己看见了什么。他看见光、动作与颜色全混杂在一起，都毫无意义，只是一片模糊。后来在一片混沌之中，他听见一个声

> 音说：“怎么样？”那时他才终于明白这一片光与阴影的混合是一张脸，而且正是他主治医生的脸。

作者又写道，

> 我们一般人生来就看得见东西，根本无法想象这种混乱的情形。对我们来说，生来就有的五种知觉，正常而且相互辅助，一开始就建立起一个视觉世界，**对所见的事物有充分的概念，也明白其意义。**我们每天一早张开眼睛所看到的世界，正是我们穷尽一生学习去看的世界。**世界并非就这样送到我们眼前，而是我们通过不断的经验积累、分类、记忆与链接而创造出来的。**但是维吉张开眼睛的时候，他已经失明了45年，他没有视觉记忆来协助他看懂东西，没有一个经验世界与意义世界等待着他。他的视觉经验只比初生婴儿多一点，而即使是这样，那些经验也早已被忘却。他是看见了，但是他所看见的东西却没有连贯性，不具任何意义。他的视网膜与视觉神经十分活泼，传送着刺激，但他的脑子却理不出其中的意义。 这正是神经学家所说的，辨识不能（Agnosic）。

也就是说手术后的维吉“视而不见”，在他的视野里的只是一片混乱。其实我们所有人刚出生时都是这样的，我们用尽一生“通过不断的经验积累、分类、记忆与链接”从而在混乱中发现概念，建立秩序。其实当我们作为系统分析师或程序员，刚接触一个陌生的业务领域时又何曾不是如此呢？一开始我们面对的只是杂乱的数据和复杂的流程，而**我们需要做的便是从混乱中发现概念，定义职责，和建立秩序。**一些对于正常人来说非常平常的图像概念，比如树，方尖碑和人，对于刚恢复视觉的维吉来说却是非常大的挑战。同样一些在领域专家看来非常平常和普通的概念，对程序员来说却是重重困惑。这就造成了程序员和领域专家的沟通有时非常困难，因为一句平常的业务行话背后其实包含了很多**隐含假设**和**深层概念**，而领域专家并没有意识到程序员其实没有这些背景知识。就如在维吉还没有空间和距离的概念时，你让他理解园艺布局的艺术几乎是不可能的。而挖掘出这些隐含假设和深层概念才是设计出一个优良的领域模型的关键。如何做到呢？DDD 给出的方法是基于统一语言（Ubiquitous Language）与领域专家持续交流，通过不断的重构精炼，最终形成挖掘出深层模型。如何保证领域专家在项目中的参与度呢？Scrum 给出的方法是在项目团队中引入 Product Owner，作为客户需求的代表深度介入进项目。但这并不是说领域专家就是发现概念的权威，这里强调的是程序员与领域专家的互动与沟通。因为有时候，领域专家对需求某些方面的理解也是片面或局限的，只有通过双方的反复沟通与碰撞，才能加深彼此对问题领域的理解。在发现了一个概念后，如何确定其职责呢？

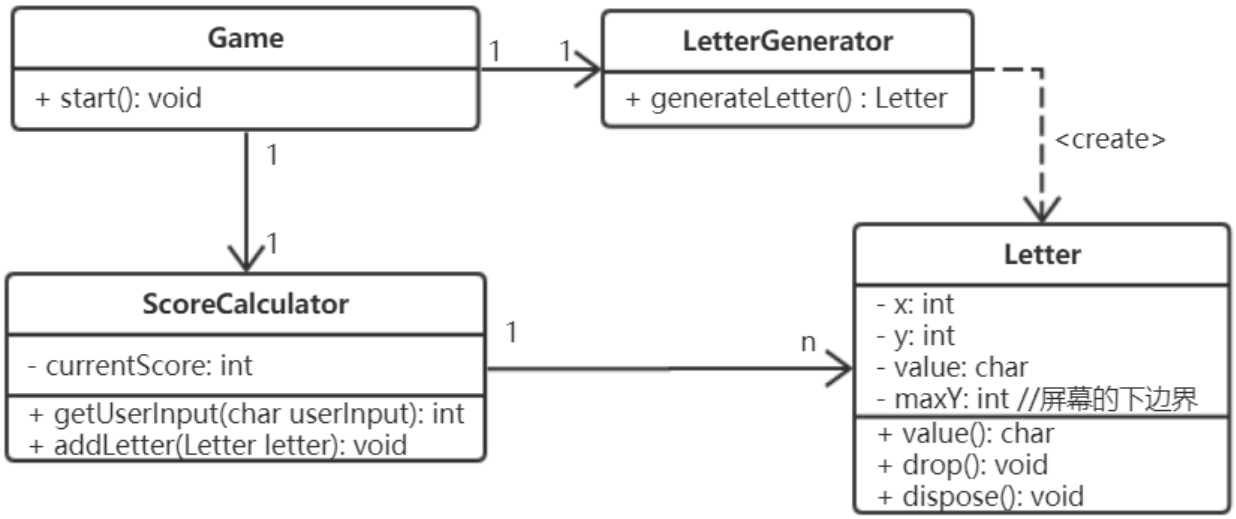
## 定义职责

职责指的是一个概念承担的责任，具体表现为：能响应哪些请求，对于每种请求需要执行哪些动作及返回什么结果。比如在一个图书管理系统中，外界可以向一个图书对象查询其名称，出版社，及当前借阅者等信息，也可以让其处理借阅和归还等请求。一个概念所承担的职责应该是和概念本身同步产生的。有时一些业务领域中的惯用语可能含义模糊，为了使其职责清晰我们甚至需要对其意义进行重新界定。就如当年牛顿在建立力学模型时所做的那样。

> “回首三个世纪前，当时物理学的发展已经到了难以突破的地步，但随着艾萨克·牛顿将一些古老但意义模糊的词（力、质量、运动，甚至时间）赋予新的含义，物理学的新时代开始了。牛顿把这些术语加以量化，以便能够放在数学方程中使用。而在此之前，motion（运动）一词（仅举此一例）的含义就……含混不清。对于当时遵循亚里士多德学说的人们而言，运动可以指代极其广泛的现象：桃子成熟、石头落地、孩童成长、尸体腐烂……但这样，它的含义就太过丰富了。只有将其中绝大多数的运动类型扬弃，牛顿运动定律才能适用，科学革命也才能继续推进。”（摘自《信息简史》）

软件开发也会遇到相似的情形。比如我参与过为一家中小型制造业企业开发人事管理系统，其中涉及两类雇员，一类是生产车间的工人，另一类是车间管理层和办公室工作人员。两者在管理方式、考勤制度、奖金福利和考核机制等各方面完全不同，每一类又有很多子类。最后我们在项目中将员工的概念窄化成专指第一类人，第二类人我们称之为职员，从而精化了项目的统一语言。这显然和我们日常对这两个名词的理解不一样，但在那个具体的项目中却取得了良好的效果。

我们再通过一个简单的打字游戏来理解一下发现概念和定义职责。字母不断从屏幕上方以一定的速度落下，用户通过按键使相应的字母从屏幕上消失从而得分。但如果屏幕上没有与按键对应的字母则扣分。如果当字母已超出了屏幕的下边界时用户仍未按相应的键，则不得分。得分与扣分规则相同，每个字母一分。最后用户关闭窗口，结束游戏。我们来逐个寻找问题空间中的概念及其职责。最明显的概念便是字母（Letter）了，它的职责包括：（1）返回字母值；（2）在屏幕上下落；（3）销毁（当从屏幕上消失或超出边界时，销毁自己，释放内存）。游戏（Game）当然也是一个概念，它的职责是协调整个程序的执行，具体职责包括：1. 启动；2. 捕获按键动作。用户（User）算一个概念吗？这要看相应的问题空间中有没有合适的职责分配给他。如果是一款多用户在线游戏，并且需要记录每个用户的历史成绩，则用户必然是一个非常重要的概念。但在这款简单的单用户游戏中则没有必要引入这个概念。另外我们需要不停的在屏幕上产生字母，这个职责可以分配给字母类吗？不合适。因为字母的生成逻辑和运行逻辑是完全不同的两套机制。字母生成关注的是以什么频率生成哪种类型的字母，这和字母概念的运行逻辑显然是不同的。所以我们再引入一个字母生成器（LetterGenerator）。它的职责很简单：生成字母。另外当用户每次按键时要计算得分，这需要持有当前屏幕上所有字母的引用，于是我们可以再引入一个分数计算器（ScoreCalculator）。它的职责包括：（1）持有游戏中所有字母；（2）执行按键与字母的匹配，并计算得分。最终的 UML 设计图如下（图中省略了一些实现细节）：




## 自治

对象对外通过接口接受请求。接口的范围应该刚好覆盖职责，不多也不少。外部不需要也不应该关注对象内部的实现机制。对象对内应该自治，即只与调用方通过接口发生交互，不用被告知自己份（职责）内的事该怎么做，否则便是职责泄露了。贫血模型便是职责泄露最极端的表现。拿上面的打字游戏来说，Letter 类的移动和销毁完全自己负责，不需要其他的类的参与。

## 继承

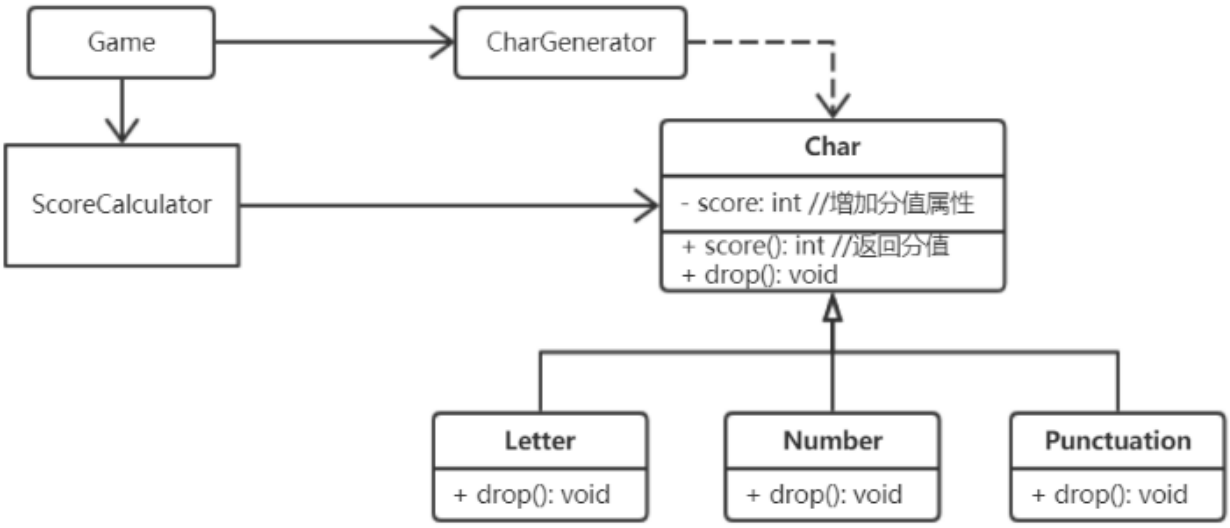


继承是面向对象中非常重要的一个概念。很多地方说继承的主要作用是实现代码复用，即子类会天然继承父类的方法从而便于扩展。与对类的误解类似，这也是完全从“实现”角度理解的结果，并且这种看法是非常有害的。从问题空间来理解，继承的本意是用来表述概念之间“is a”关系的即“子类 is a 父类”，每一个子类是父类的一种变化形式。这样理解的好处便是针对父类所写的代码，可以适用于所有子类。这会大大增加系统可扩展性，即未来可以非常灵活地增加新的子类而不用影响调用方代码。所以说**继承是一种封装方式，封装的是父类的各种变化形式。**

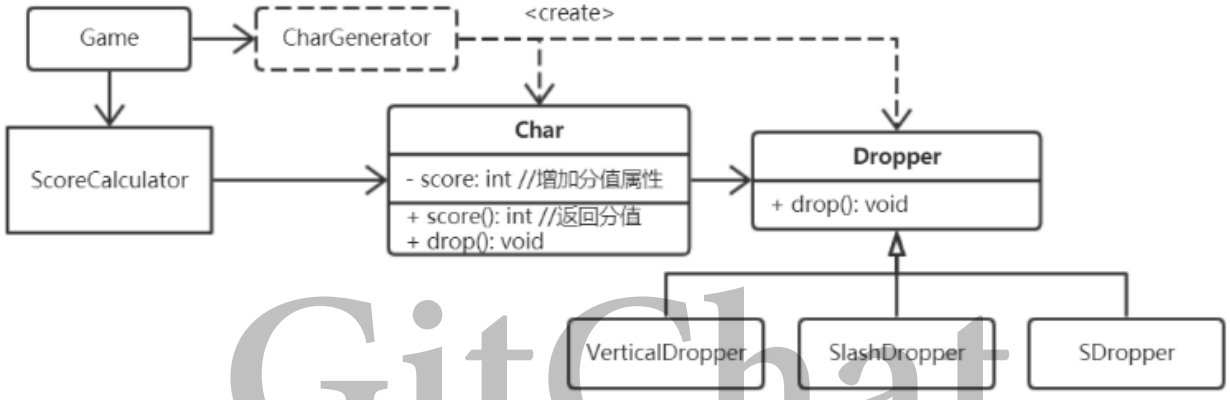
作为例子，我们为游戏程序增加如下特性：

1. 支持数字和标点的打字练习；
2. 计分规则改变：字母一分，数字两分，标点符号三分；
3. 字母垂直下落，数字以斜线的路径下落，而标点符号以 S 型的路径下落。

这个需求变更引入了如下一些概念：数字（Number），标点（Punctuation），分值（Score），下落路径（Dropping Path）。显然这里数字，标点和字母是对等的，是键盘上符号的不同类型而已，于是我们便抽象出了一个新的概念——符号（Char）。然后每一种类型又有不同的分值和下落路径。这样我们便抽象出了一个继承体系，如下图（只标出了与上一图的差异部分）：




还有没有其他的设计方式呢？如下便是一种（总是想想另一种可能性是一种好习惯）。






如果游戏又需要增加汉字打字功能：（1）每个汉字5分；（2）以倒 S 型路径下落。作为作业大家对比一下，以上两种设计方案对这个需求变化的响应度如何？

如果增加的需求是汉字也以斜线下落，以上两种设计方案的响应度又如何？

我们可以在 Chat 交流部分一起来研讨这个问题。

本文中，我们介绍了面向对象的核心思想：

1. 站在问题空间的角度发现概念（即对象）并为其定义职责；
2. 对象应该自治，即封装其内部实现并通过接口与外界交互；
3. 继承是对类型进行封装；装的是父类的各种变化。

本文我们没有谈技“术”，但我们对面向对象设计在“道”的层面做了深层阐述。那如何让这些“道”，以技术的形式落地并形成可行的解决方案呢？欢迎参加下一场 Chat：DDD（领域驱动设计）的正确姿势。

本文参考书目：

- 《设计模式解析》
- 《领域驱动设计》
- 《火星上的人类学家》
- 《信息简史》