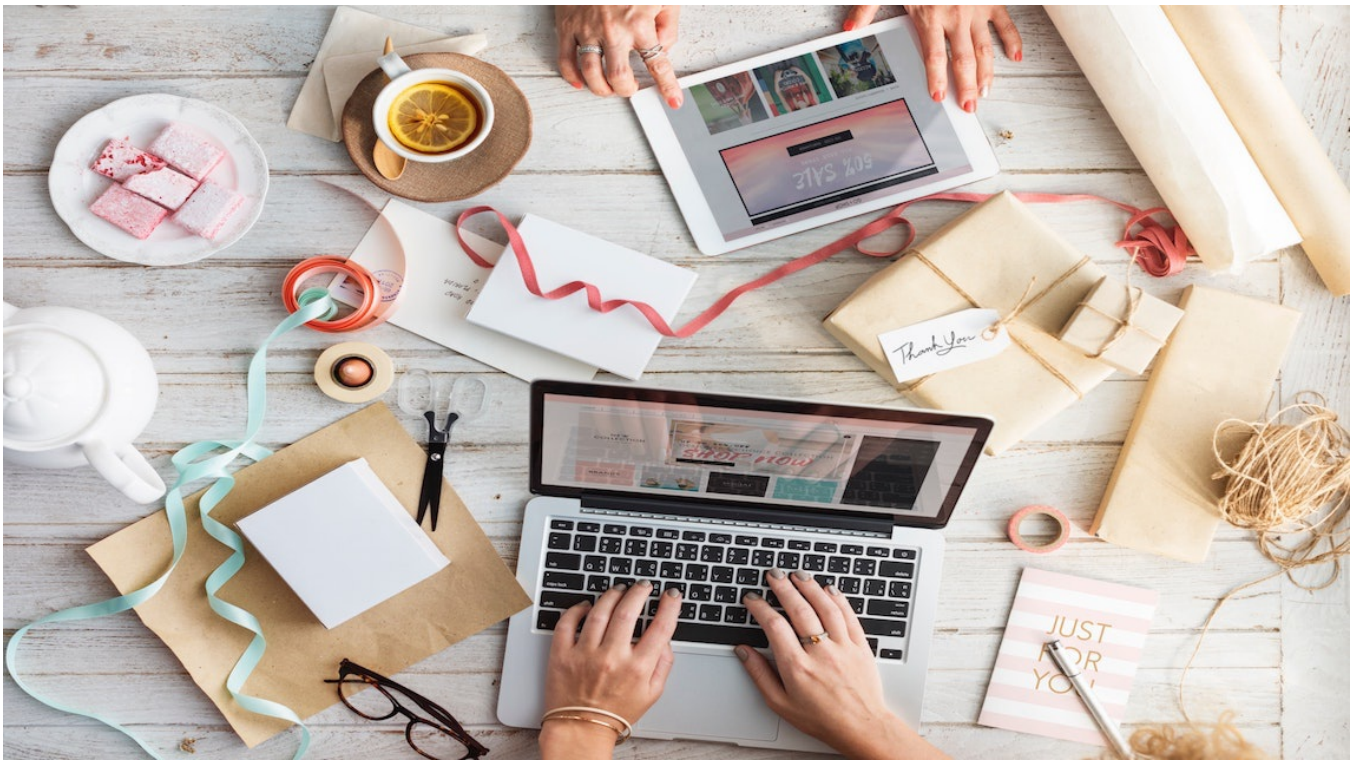
08 | 代码与分类：工业级编程的代码分类与特征

2018-08-20 胡峰



编程，就是写代码，那么在真实的行业项目中你编写的这些代码可以如何分类呢？回顾我曾经写过的各种系统代码，按代码的作用，大概都可以分为如下三类：

- 功能
- 控制
- 运维

如果你想提高编程水平，写出优雅的代码，那么就必须要清晰地认识清楚这三类代码。

一、功能

功能代码，是实现需求的业务逻辑代码，反映真实业务场景，包含大量领域知识。

一个程序软件系统，拥有完备的功能性代码仅是基本要求。因为业务逻辑的复杂度决定了功能性代码的复杂度，所以要把功能代码写好，最难的不是编码本身，而是搞清楚功能背后的需求并得到正确的理解。之后的编码活动，就仅是一个“翻译”工作了：把需求“翻译”为代码。

当然，“翻译”也有自己独有的技术和积累，并不简单。而且“翻译”的第一步要求是“忠于原文”，也即真正地理解并满足用户的原始需求。可这个第一步的要求实现起来就很困难。

为什么搞清楚用户需求很困难？因为从用户心里想要的，到他最后得到的之间有一条长长的链条，如下所示：

> 用户心理诉求 -> 用户表达需求 -> 产品定义需求 -> 开发实现 -> 测试验证 -> 上线发布 -> 用户验收

需求信息源自用户的内心，然后通过表达显性地在这个链条上传递，最终固化成了代码，以程序系统的形态反馈给了用户。

但信息在这个链条中的每个环节都可能会出现偏差与丢失，即使最终整个链条上的各个角色都貌似达成了一致，完成了系统开发、测试和发布，但最终也可能发现用户的心理诉求要么表达错了，要么被理解错了。

因为我近些年一直在做即时通讯产品（IM），所以在这儿我就以微信这样一个国民级的大家都熟悉的即时通讯产品为样本，举个例子。

微信里有个功能叫：消息删除。你该如何理解这个功能背后的用户心理诉求呢？用户进行删除操作的期待和反馈又是什么呢？从用户发消息的角度，我理解其删除消息可能的诉求有如下几种：

1. 消息发错了，不想对方收到。
2. 消息发了后，不想留下发过的痕迹，但期望对方收到。
3. 消息已发了，对于已经收到的用户就算了，未收到的最好就别收到了，控制其传播范围。

对于第一点，微信提供了两分钟内撤回的功能；而第二点，微信提供的删除功能正好满足；第三点，微信并没有满足。我觉着第三点其实是一个伪需求，它其实是第一点不能被满足情况下用户的一种妥协。

用户经常会把他们的需要，表达成对你的行为的要求，也就是说不真正告诉你要什么，而是告诉你要做什么。所以你才需要对被要求开发的功能进行更深入的思考。有时，即使是日常高频使用的产品背后的需求，你也未必能很好地理解清楚，而更多的业务系统其实离你的生活更远，努力去理解业务及其背后用户的真实需求，才是写好功能代码的基本能力。

程序存在的意义就在于实现功能，满足需求。而一直以来我们习惯于把完成客户需求作为程序开发的主要任务，当功能实现了便感觉已经完成了开发，但这仅仅是第一步。

二、控制

控制代码，是控制业务功能逻辑代码执行的代码，即业务逻辑的执行策略。

编程领域熟悉的各类设计模式，都是在讲关于控制代码的逻辑。而如今，很多这些常用的设计模式基本都被各类开源框架固化了进去。比如，在 Java 中，Spring 框架提供的控制反转（IoC）、依赖注入（DI）就固化了工厂模式。

通用控制型代码由各种开源框架来提供，程序员就被解放出来专注写好功能业务逻辑。而现今分布式领域流行的微服务架构，各种架构模式和最佳实践也开始出现在各类开源组件中。比如微服务架构模式下关注的控制领域，包括：通信、负载、限流、隔离、熔断、异步、并行、重试、降

级。

以上每个领域都有相应的开源组件代码解决方案，而进一步将控制和功能分离的“服务网格（Service Mesh）”架构模式则做到了极致，控制和功能代码甚至运行在了不同的进程中。

控制代码，都是与业务功能逻辑不直接相关的，但它们和程序运行的性能、稳定性、可用性直接相关。提供一项服务，功能代码满足了服务的功能需求，而控制代码则保障了服务的稳定可靠。

有了控制和功能代码，程序系统终于能正常且稳定可靠地运行了，但难保不出现异常，这时最后一类“运维”型代码便要登场了。

三、运维

运维代码，就是方便程序检测、诊断和运行时处理的代码。它们的存在，才让系统具备了真正工业级的可运维性。

最常见的检测诊断性代码，应该就是日志了，打日志太过简单，因此我们通常也就疏于考虑。其实即使是打日志也需要有意识的设计，评估到底应该输出多少日志，在什么位置输出日志，以及输出什么级别的日志。

检测诊断代码有一个终极目标，就是让程序系统完成运行时的自检诊断。这是完美的理想状态，却很难在现实中完全做到。

因为它不仅仅受限于技术实现水平，也与实现的成本和效益比有关。所以，我们可以退而求其次，至少在系统异常时可以具备主动运行状态汇报能力，由开发和运维人员来完成诊断分析，这也是我们常见的各类系统或终端软件提供的机制。

在现实中，检测诊断类代码经常不是一开始就主动设计的。但生产环境上的程序系统可能会偶然出现异常或故障，而因为一开始缺乏检测诊断代码输出，所以很难找到真实的故障原因。现实就这样一步一步逼着你去找到真实原因，于是检测诊断代码就这么被一次又一次地追问为什么而逐渐完善起来了。

但如果一开始你就进行有意识地检测诊断设计，后面就会得到更优雅的实现。有一种编程模式：面向切面编程（AOP），通过早期的有意设计，可以把相当范围的检测诊断代码放入切面之中，和功能、控制代码分离，保持优雅的边界与距离。

而对于特定的编程语言平台，比如 Java 平台，有字节码增强相关的技术，可以完全干净地把这类检测诊断代码和功能、控制代码彻底分离。

运维类代码的另一种类，是方便在运行时，对系统行为进行改变的代码。通常这一类代码提供方便运维操作的 API 服务，甚至还会有专门针对运维提供的服务和应用，例如：备份与恢复数据、实时流量调度等。

功能、控制、运维，三类代码，在现实的开发场景中优先级这样依次排序。有时你可能仅仅完成了第一类功能代码就迫于各种压力上线发布了，但你要在内心谨记，少了后两类代码，将来都会是负债，甚至是灾难。而一个满足工业级强度的程序系统，这三类代码，一个也不能少。

而对三类代码的设计和实现，越是优雅的程序，这三类代码在程序实现中就越是能看出明显的边界。为什么需要边界？因为，“码以类聚，人以群分”。功能代码易变化，控制代码固复杂，运维代码偏繁琐，这三类不同的代码，不仅特征不同，而且编写它们的人（程序员）也可能分属不同群组，有足够的边界与距离才能避免耦合与混乱。

而在程序这个理性世界中，优雅有时就是边界与距离。