*Clean Code*（《代码简洁之道》）读书笔记

这是一本非常经典的书，读了之后，我感觉在如何写出简洁优雅的高质量代码方面对我很有启发。书中从各个方面，列举了许多原则与标准，但我们写代码的时候，不可能将这些东西都清楚记着并时刻提醒自己不能“违规”，这样就如同负重前行，不仅走得慢，还会累死。

一.整洁代码

整洁代码的一些特征

代码逻辑应该直接了当，叫缺陷难以隐藏；

尽量减少依赖关系，使之便于维护；

依据某种分层战略完善错误处理代码；

性能调至最优，省得引诱别人做没规矩的优化，搞出一堆混乱来；

整洁的代码只做好一件事；

有单元测试和验收测试；

有意义的命名；

尽量“少”；

两条重要原则：

尽量让代码易读，开发时减少读的时间。

童子军军规：“让营地比你来时更干净”。

二.有意义的命名

名副其实

避免误导

1.程序员必须避免留下掩藏代码本意的错误线索，应当避免使用与本意相悖的词

2.以同样的方式拼写出同样的概念才是信息，拼写前后不一致就是误导

3.要注意使用小写字母i和大写字母O作为变量名，看起来像“壹”和“零”

做有意义的区分

1.不同东西意义不一样时，一定要在命名上区分，如变量，不要简单的cat1，cat2这样区分

命名读得出来，单词可搜索

避免使用编码

1.不要乱用前缀

避免思维映射

1.明确是王道，不要让人需要想象或者产生联想

类名是名词或名词短语，方法名是动词或者动词短语

别扮可爱

别用双关语

1.避免将同一单词用于不同目的

2.应尽力写出易于理解的代码，把代码写得让别人能一目尽览而不必殚精竭虑地研究

使用解决方案领域名称，使用源自所涉问题领域的名称

添加有意义的语境

1.你需要用有良好命名的类、函数或名称空间来放置名称，给读者提供语境

2.如果没这么做，给名称添加前缀就是最后一招了

不要添加没用的语境

1.只要短名称足够清楚，就要比长名称好

最后的话

1.取好名字最难的地方在于需要良好的描述技巧和共有文化背景

三.函数

短小，只做一件事

每个函数一个抽象层级

1.要确保函数只做一件事，函数中的语句都要在同一抽象层级上

2.自顶向下读代码：向下规则，让代码拥有自顶向下的阅读顺序，让每个函数后面都跟着下一抽象层级的函数，这样一来，在看函数列表时，就能循抽象层级向下阅读了，我把这叫做向下规则

switch语句

1.尽量隐藏switch，不要暴露

使用描述性的名称

1.沃德原则：“如果每个例程都让你感到深合已意，那就是整洁代码”

2.函数越短小，功能越集中，就越便于取个好名字

3.别害怕长名称，长而具有描述性的名称，要比短而令人费解的名称好

4.命名方式要保持一致。使用与模块名一脉相承的短语、名词和动词给函数命名

函数参数

1.最理想的参数数量是零，有足够的理由才能用三个以上参数

2.事件：在这种形式中，有输入参数而无输出参数，程序将函数看作一个事件，使用该参数修改系统状态

3.对于转换，使用输出参数而非返回值令人迷惑，如果函数要对输入参数进行转换操作，转换结果就该体现为返回值

4.向函数传入布尔值会使方法签名立刻变得复杂起来，大声宣布函数不止做一件事

5.如果函数看来需要两个、三个或三个以上参数，就说明其中一些参数应该封装为类了

6.有可变参数的函数可能是一元、二元甚至三元，超过这个数量就可能要犯错了

7.对于一元函数，函数和参数应当形成一种非常良好的动词/名词对形式

无副作用

1.函数承诺只做一件事，但还是会做其他被藏起来的事，会导致古怪的时序性耦合及顺序依赖

2.参数多数会被自然而然地看作是函数的输入

分隔指令与询问

1.函数要么做什么事，要么回答什么事，但二者不可得兼

使用异步替代返回错误码

1.从指令式函数返回错误码轻微违反了指令与询问分隔的规则。它鼓励了在if语句判断中把指令当作表达式使用

2.try/catch代码块把错误处理与正常流程混为一谈，最好把try和catch代码块的主体部分抽离出来，另外形成函数

3.错误处理就是一件事，处理错误的函数不该做其他事

4.依赖磁铁（dependency magnet）：其他许多类都得导入和使用它，如一些错误码枚举类

别重复自己

结构化编程

1.函数中的每个代码块都应该只有一个入口和一个出口

2.尽量少使用break，continue，不能用goto

如何写出这样的函数

1.打磨代码，分解函数、修改名称、消除重复

2.缩短和重新安置方法、拆散类、保持测试通过

四.注释

1.若编程语言足够有表达力，就不需要注释

2.注释的恰当用法是弥补我们在用代码表达意图时遭遇的失败。注释总是一种失败

3.程序员应当负责将注释保持在可维护、有关联、精确的高度，更应该把力气用在写清楚代码上，直接保证无须编写注释

4.不准确的注释要比没注释坏得多

注释不能美化糟糕的代码

用代码来阐述

五.格式

格式的目的

1.代码格式关乎沟通，而沟通是专业开发者的头等大事

垂直格式

1.通过间隔来区分逻辑模块

横向格式

1.适当对齐，间隔

团队规则

1.一组开发者应当认同一种模式风格，每个成员都应该采用那种风格

2.好的软件系统是由一系列读起来不错的代码文件组成的，需要拥有一致和顺畅的风格

六.对象和数据结构

数据、对象的反对称性

1.过程式代码难以添加新的数据结构，因为这要修改所有相关函数；面向对象代码难以添加新函数，因为要修改所有相关类

得墨忒耳律

1.得墨忒耳律（The Law of Demeter）：模块不应了解它所操作对象的内部情形，意味着对象不应通过存取器曝露其内部结构，因为这样更像是曝露而非隐藏其内部结构

2.混合结构，一半是对象，一半是数据结构，应避免这种结构

数据传送对象

1.最为精练的数据结构，是一个只有公共变量、没有函数的类，这种被称为数据传送对象，或DTO（Data Transfer Objects）。在与数据库通信、或解析套接字传递的消息之类场景中存在。

2.JavaBean或Active Record

3.不要塞进业务规则方法，把Active Record当做数据结构。创建包含业务规则、隐藏内部数据（可能就是Active Record的实体）的独立对象。

七.错误信息

错误处理很重要，但如果它搞乱了代码逻辑，就是错误的做法

使用异常而非返回码

1.遇到错误时，最好抛出一个异常。调用代码很整洁，其逻辑不会被错误处理搞乱

先写Try-Catch-Finally语句

1.异常的妙处之一是，它们在程序中定义了一个范围。执行try-catch-finally语句中try部分的代码时，你是在表明可随时取消执行，并在catch语句中接续

2.在某种意义上，try代码块就像是事务，catch代码块将程序维持在一种持续状态

3.在编写可能抛出异常的代码时，最好先写try-catch-finally语句，能帮你定义代码的用户应该期待什么，无论try代码块中执行的代码出什么错都一样

使用不可控异常

1.可控异常，就是指在方法签名中标注异常。但有时候会产生多层波及，有时候你对较底层的代码修改，可能会波及很多上层代码

给出异常发生的环境说明

1.抛出的每个异常，都应当提供足够的环境说明，以便判断错误的来源和处所

2.应创建信息充分的错误消息，并和异常一起传递出去

依调用者需要定义异常类

1.自定义的异常类最重要的考虑是它们如何被捕获

2.将第三方API打包是个良好的实践手段，降低了对每个第三方的依赖，也有助于模拟第三方调用，并将一些复杂的异常捕获过程封装

定义常规流程

1.特例模式（SPECIAL CASE PATTERN,[Fowler]），创建一个类或配置一个对象，用来处理特例，异常行为被封装到特例对象中

尽量别返回null值，也别传递null值

八.边界

使用第三方代码

1.第三方程序包和框架提供者追求普适性，这样就能在多个环境中工作，吸引广泛的用户

2.我们建议不要将Map（或在边界上的其他接口）在系统中传递，把它保留在类或近亲类中，避免从API中返回边界接口，或将接口作为参数传递给公共API

学习性测试的好处不只是免费

1.学习性测试毫无成本，编写测试是获得这些知识（要使用的API）的容易而不会影响其他工作的途径

2.学习性测试确保第三方程序包按照我们想要的方式工作

使用尚不存在的代码

编写我们想得到的接口，好处之一是它在我们控制之下，有助于保持客户代码更可读，且集中于它该完成的工作

整洁的边界

1.边界上的改动，有良好的软件设计，无需巨大投入和重写即可进行修改

2.边界上的代码需要清晰的分割和定义了期望的测试。依靠你能控制的东西，好过依靠你控制不了的东西，免得日后受它控制

3.可以使用ADAPTER模式将我们的接口转换为第三方提供的接口

九.单元测试

TDD三定律

1.在编写能通过的单元测试前，不可编写生产代码(测试先行)

2.只可编写刚好无法通过的单元测试，不能编译也算不通过(测试一旦失败，开始写生产代码)

3.只可编写刚好足以通过当前失败测试的生产代码(老测试一旦通过，返回写新测试)

测试先行的一大好处：

如果先写测试，必然得知道输入是什么，期望输出的是什么。这样就不断促进思考该如何得到这样的输出，即把设计代码也放入了测试前期的准备中。

保持测试整洁

1.脏测试等同于没测试，测试必须随生产代码的演进而修改，测试越脏，就越难修改

2.测试代码和生产代码一样重要，它需要被思考、被设计和被照料，它该像生产代码一般保持整洁

3.如果测试不能保持整洁，你就会失去它们，没有了测试，你就会失去保证生产代码可扩展的一切要素

整洁的测试

1.三个要素：可读性、可读性和可读性，明确、简洁还有足够的表达力

2.构造-操作-检验（BUILD-OPERATE-CHECK）模式，第一个环节构造测试数据，第二个环节操作测试数据，第三个部分检验操作是否得到期望的结果

3.守规矩的开发者也将他们的测试代码重构为更简洁和具有表达力的形式

每个测试一个断言

1.JUnit中每个测试函数都应该有且只有一个断言语句

2.最好的说法是单个测试中的断言数量应该最小化

3.更好一些的规则或许是每个测试函数中只测试一个概念

4.最佳规则是应该尽可能减少每个概念的断言数量，每个测试函数只测试一个概念

F.I.R.S.T

1.快速（Fast）测试应该够快

2.独立（Independent）测试应该相互独立

3.可重复（Repeatable）测试应当可在任何环境中重复通过

4.自足验证（Self-Validating）测试应该有布尔值输出，自己就能给出对错，而不需要通过看日志，比对结果等方式验证

5.及时（Timely）测试应及时编写

十.类

类的组织

1.类应该从一级变量列表开始，如果有公共静态变量，应该先出现，然后是私有静态变量，以及实体变量，很少会有公共变量

2.公共函数应该跟在变量列表之后

3.保持变量和工具函数的私有性，但并不执着于此

类应该短小

1.职责单一，类的名称应当描述其权责，如果无法为某个类命以精确的名称，这个类大概就太长了，类名越含混，该类越有可能拥有过多权责；类或模块应有且只有一条加以修改的理由

2.内聚性强，方法操作的变量越多，就越黏聚到类上，如果一个类的每个变量都被每个方法所使用，则该类具有最大的内聚性

3.保持函数和参数列表短小的策略，有时会导致为一组子集方法所用的实体变量数量增加。出现这种情况时，往往意味着至少有一个类要从大类中挣扎出来。你应当尝试将这些变量和方法分拆到两个或多个类中，让新的类更为内聚

4.将大函数拆为许多小函数，往往也是将类拆分为多个小类的时机

为了修改而组织

1.在整洁的系统中，我们对类加以组织，以降低修改的风险

2.开放-闭合原则（OCP）：类应当对扩展开放，对修改封闭

3.在理想系统中，我们通过扩展系统而非修改现有代码来添加新特性

4.依赖倒置原则（Dependency Inversion Principle,DIP），类应该依赖于抽象而不是依赖于具体细节

十一.系统

如何建造一个城市

1.每个城市都有一组人管理不同的部分，有人负责全局，其他人负责细节

2.深化出恰当的抽象等级和模块，好让个人和他们所管理的“组件”即便在不了解全局时也能有效地运转

将系统的构造与使用分开

1.构造与使用是非常不一样的过程

2.软件系统应将启始过程和启始过程之后的运行时逻辑分离开，在启始过程中构建应用对象，也会存在互相缠结的依赖关系

3.将构造与使用分开的方法之一是将全部构造过程搬迁到main或被称为main的模块中，设计系统的其余部分时，假设所有对象都已正确构造和设置

4.可以使用抽象工厂模式让应用自行控制何时创建对象，但构造的细节却隔离于应用程序代码之外

5.控制反转将第二权责从对象中拿出来，转移到另一个专注于此的对象中，从而遵循了单一权责原则。在依赖管理情景中，对象不应负责实体化对自身的依赖，反之，它应当将这份权责移交给其他“有权力”的机制，从而实现控制的反转

扩容

1.“一开始就做对系统”纯属神话，反之，我们应该只去实现今天的用户故事，然后重构，明天再扩展系统、实现新的用户故事，这就是迭代和增量敏捷的精髓所在。测试驱动开发、重构以及它们打造出的整洁代码，在代码层面保证了这个过程的实现

2.软件系统与物理系统可以类比。它们的架构都可以递增式的增长，只要我们持续将关注面恰当地切分

3.持久化之类关注面倾向于横贯某个领域的天然对象边界

Java代理

纯Java AOP框架

AspectJ的方面

测试驱动系统架构

1.通过方面式（AOP）的手段切分关注面的威力不可低估。假使你能用POJO编写应用程序的领域逻辑，在代码层面与架构关注面分离开，就有可能真正地用测试来驱动架构

2.没必要先做大设计（Big Design Up Front，BDUF），BDUF甚至是有害的，它阻碍改进，因为心理上会抵制丢弃即成之事，也因为架构上的方案选择影响到后续的设计思路

3.我们可以从“简单自然”但切分良好的架构开始做软件项目，快速交付可工作的用户故事，随着规模的增长添加更多基础架构

4.最佳的系统架构由模块化的关注面领域组成，每个关注面均用纯Java（或其他语言）对象实现，不同的领域之间用最不具有侵害性的方面或类方面工具整合起来，这种架构能测试驱动，就像代码一样

优化决策

1.模块化和关注面切分，成就了分散化管理和决策

2.延迟决策至最后一刻也是好手段，它让我们能够基于最有可能的信息做出选择

3.拥有模块化关注面的POJO系统提供的敏捷能力，允许我们基于最新的知识做出优化的、时机刚好的决策，决策的复杂性也降低了

明智使用添加了可论证价值的标准

1.有了标准，就更易复用想法和组件、雇用拥有相关经验的人才、封装好点子，以及将组件连接起来。不过，创立标准的过程有时却漫长到行业等不及的程度，有些标准没能与它要服务的采用者的真实需求相结合

系统需要领域特定语言

1.领域特定语言（Domain-Specific Language, DSL）是一种单独的小型脚本语言或以标准语言写就的API，领域专家可以用它编写读像是组织严谨的散文一般的代码

2.领域特定语言允许所有抽象层级和应用程序中的所有领域，从高级策略到底层细节，使用POJO来表达

十二.迭代

通过迭进设计达到整洁目的

简单规则：

\* 通过所有测试

\* 不可重复

\* 表达了程序员的意图

\* 尽可能减少类和方法的数量

\* 以上规则按其重要程序排列

简单设计原则2～4：重构

1.有了测试，就能保持代码和类的整洁，方法就是递增式地重构代码

2.测试消除了对清理代码就会破坏代码的恐惧

不可重复

表达力

尽可能少的类和方法

十三.并发编程

为什么要并发

1.并发是一种解耦策略，它帮助我们把做什么（目的）和何时（时机）做分解开

2.解耦目的与时机能明显地改进应用程序的吞吐量和结构

3.单线程程序许多时间花在等待web套接字I/O结束上面，通过采用同时访问多个站点的多线程算法，就能改进性能

常见的迷思和误解

\* 并发总能改进性能：只在多个线程或处理器之间能分享大量等待时间的时候管用

\* 编写并发程序无需修改设计：可能与单线程系统的设计极不相同

\* 在采用web或ejb容器时，理解并发问题并不重要

有关编写并发软件的中肯的说法

\* 并发会在性能和编写额外代码上增加一些开销

\* 正确的并发是复杂的，即使对于简单的问题也是如此

\* 并发缺陷并非总能重现，所以常被看做偶发事件而忽略，未被当做真的缺陷看待

\* 并发常常需要对设计策略的根本性修改

并发防御原则

1.单一权责原则

2.限制数据作用域

3.使用数据副本

4.线程应尽可能独立

了解Java库

\* 使用类库提供的线程安全群集

\* 使用executor框架（executor framework）执行无关任务

\* 尽可能使用非锁定解决方案

\* 有几个类并不是线程安全的

了解执行模型

警惕同步方法之间的依赖

保持同步区域微小

很维编写正确的关闭代码

1.平静关闭很难做到，常见问题与死锁有关，线程一直等待永远不会到来的信号

2.建议：尽早考虑关闭问题，尽早令其工作正常

测试线程代码

1.建议：编写有潜力曝露问题的测试，在不同的编程配置、系统配置和负载条件下频繁运行。如果测试失败，跟踪错误。别因为后来测试通过了后来的运行就忽略失败

2.将伪失败看作可能的线程问题：线程代码导致“不可能失败的”失败，不要将系统错误归咎于偶发事件

3.先使非线程代码可工作：不要同时追踪非线程缺陷和线程缺陷，确保代码在线程之外可工作

4.编写可插拔的线程代码，能在不同的配置环境下运行

5.编写可调整的线程代码：允许线程依据吞吐量和系统使用率自我调整

6.运行多于处理器数量的线程：任务交换越频繁，越有可能找到错过临界区域导致死锁的代码

7.在不同平台上运行：尽早并经常地在所有目标平台上运行线程代码

8.装置试错代码：增加对Object.wait()、Object.sleep()、Object.yield()、Object.priority()等方法的调用，改变代码执行顺序，硬编码或自动化

十四.逐步改进

1.要编写清洁代码，必须先写肮脏代码，然后再清理它

2.毁坏程序的最好方法之一就是以改进之名大动其结构