《架构整洁之道》读书笔记

撰写人：Relph Hu

1 说明

本书用到的编程语言：C、Java、Clojure、C++、PDP-8，其实本书用到的这些编程语言，只是为了说明架构与语言无关。

2 章节笔记

2.1 第 1 部分：概述

（1） 软件架构的**终极目标**是，用**最小**的**人力成本**来满足**构建**和**维护**该系统的**需求**。

（2） 过度自信只会使得重构设计陷入和原项目一样的困局中。所以对于架构师来说，千万不要过度自信!!!

（3） 对于每个软件系统，可以通过**行为**和**架构**两个维度来体现它的实际价值。行为价值：最直观的价值，根据需求，让系统功能可以运行。架构价值：软件系统必须保存灵活，好的系统架构设计应该尽可能做到与“形状”无关。

（4） 软件系统的第一个价值维度：**系统行为**，是紧急的，但是并不总是特别重要。软件系统的第二个价值维度：**系统架构**，是重要的，但是并不总是特别紧急。

2.2 第 2 部分：从基础构件开始：编程范式

（1） **结构化编程**：对程序控制权的直接转移进行了限制和规范，限制了 goto 语句。最有价值的地方就是，它赋予了我们创造可证伪程序单元的能力。在架构设计领域，功能性降解拆分是最佳实践之一。

（2） **面向对象编程**：对程序控制权的间接转移进行了限制和规范，限制了函数指针。要求程序员尽量避免破坏数据的封装性，但相对于 C 这种完美封装的语言，其封装性被削弱了，而不是加强了。面向对象编程就是以多态为手段来对源代码中的依赖关系进行控制的能力，让高层策略性组件与底层实现性组件相分离，底层组件可以被编译成插件，实现独立于高层组件的开发与部署。

（3） 一个架构设计良好的应用程序应该将状态修改的部分和不需要修改状态的部分隔离成单独的组件，然后用合适的机制来保护可变量。

（4） **函数式编程**：对程序中的赋值进行了限制和规范，λ演算法的一个核心思想是不可变性——某个符号所对应的值是永远不变的。

2.3 第 3 部分：设计原则

（1） SRP **单一职责原则**：任何一个软件模块都应该只对某一类行为者负责。主要讨论的是函数和类之间的关系，在组件层面，成为 CCP 共同闭包原则，在软件架构层面，用于奠定架构边界的变更轴心。

（2） OCP **开闭原则**：是我们进行系统架构设计的主导原则，其主要目标是让系统易于扩展，同时限制其每次被修改所影响的范围。实现方式是通过将系统划分为一系列组件，并将这些组件间的依赖关系按层次结构进行组织，使得高阶组件不会因低阶组件被修改而受到影响。

（3） LSP **里氏替换原则**：可以且应该被用于软件架构层面，因为一旦违背了可替换性，该系统架构就不得不为此增添大量复杂的应对机制。

（4） ISP **接口隔离原则**：在一般情况下，任何层次的软件设计如果依赖于不需要的东西，都会是有害的。

（5） DIP **依赖反转原则**：如果想要设计一个灵活的系统，在源代码层次的依赖关系中就应该多引用抽象类型，而非具体实现。**具体的编码守则**：应在代码中多使用抽象接口，尽量避免使用那些多变的具体实现类。不要在具体实现类上创建衍生类。不要覆盖包含具体实现的函数。应避免在代码中写入与任何具体实现相关的名字，或者是其他容易变动的事物的名字。

2.4 第 4 部分：组件构建原则

（1） REP **复用/发布等同原则：**软件复用的最小粒度应等同于其发布的最小粒度。

（2） CCP **共同闭包原则：**我们应该将那些会同时修改，并且为相同目的而修改的类放到同一个组件中，而将不会同时修改，并且不会为了相同目的而修改的那些类放到不同的组件中。

（3） 以上两个原则可以用一句话来概括：**将由于相同原因而修改，并且需要同时修改的东西放在一起。将由于不同原因而修改，并且不同时修改的东西分开。**

（4） CRP **共同复用原则：**不要强迫一个组件的用户依赖他们不需要的东西。将经常共同复用的类和模块放在同一个组件中。CRP 原则实际上是 ISP 原则的一个普适版。**不要依赖不需要用到的东西。**

（5） ADP **无依赖环原则：**组件依赖关系图中不应该出现环。打破循环依赖的两种主要机制：应用依赖反转原则，创建一个新的组件。

（6） 组件结构图中一个重要目标是知道如何隔离频繁的变更。

（7） SDP **稳定依赖原则：**依赖关系必须要指向更稳定的方向。

（8） SAP **稳定抽象原则：**一个组件的抽象化程度应该与其稳定性保持一致。

2.5 第 5 部分：软件架构

（1） 如果想设计一个便于推进各项工作的系统，其策略就是要在设计中尽可能长时间地保留尽可能多的可选项。

（2） 软件架构师的**目的**是创建一种系统形态，该形态会以策略为最基本的元素，并**让细节与策略脱离关系**，以允许在具体决策过程中推迟或延迟与细节相关的内容。

（3） 一个设计良好的软件架构必须支持以下几点：系统的用例与正常运行、系统的维护、开发和部署。

（4） 优秀的架构师应该将用例的 UI 部分与其业务逻辑部分隔离，这样这两个部分就既可以各自进行变更，也能保证用例的完整清晰。

（5） 按照变更原因的不同对系统进行解耦，就可以持续地向系统内添加新的用例，而不会影响旧有的用例。如果同时对用例的 UI 和数据库进行分组，每个用例使用的就是不同面向的 UI 和数据库，因此增加新用例就更不太可能会影响旧有的用例了。

（6） 高吞吐量的用例和需要低吞吐量的用例互相自然分开。

（7） 按水平分层和用例解耦一个系统有很多种方式。**源码层次：**控制源代码模块之间的依赖关系，以此来实现一个模块的变更不会导致其他模块也需要变更或重新编译。**部署层次：**控制部署单元之间的依赖关系。**服务层次：**将组件间的依赖关系降低到数据结构级别，然后仅通过网络数据包来进行通信。

（8） 业务逻辑需要的是有一组可以用来查询和保存数据的函数。

（9） 软件开发技术发展的历史就是一个如何想法设法方便增加插件，从而构建一个可扩展、可维护的系统架构的故事。

（10） 为了在软件架构中画边界线，需要先将系统分割成组件，其中一部分是系统的核心业务逻辑组件，另一部分是与核心业务逻辑无关但负责提供必要功能的插件。然后通过对源代码的修改，让这些非核心组件依赖于系统的核心业务逻辑组件。

（11） 在运行时，跨边界调用是指边界线一侧的函数调用另一侧的函数，并同时传递数据的行为。

（12） 业务实体只要求我们将关键业务数据和关键业务逻辑绑定在一个独立的软件模块内。

（13） 这些业务逻辑应该保持纯净，不要掺杂用户界面或者所使用的数据库相关的东西。在理想情况下，这部分代表业务逻辑的代码应该是整个系统的核心，其他低层概念的实现应该以插件形式接入系统中。业务逻辑应该是系统中最独立、复用性最高的代码。

（14） 一个系统的架构应该着重于展示系统本身的设计，而并非该系统所使用的框架。

（15） 源码中的依赖关系必须指向同心圆的内层，即由低层机制指向高层策略。

（16） 业务实体属于系统中最不容易受外界影响而变动的部分。

（17） 软件的用例层中通常包含的是特征应用场景下的业务逻辑，这里面封装

并实现了整个系统的所有用例。

（18） 虽然软件质量本身并不会随时间推移而损耗，但是未妥善管理的硬件依赖和固件依赖却是软件的头号杀手。

2.6 实现细节

（1） 一个优秀的架构师是不会让实现细节污染整个系统架构的。

（2） 将业务规则与 UI 解耦

（3） 我们与框架作者之间的关系是非常不对等的。我们要采用某个框架就意味着自己要遵守一大堆约定，但框架作者却完全不需要为我们遵守什么约定。

（4） 当我们面临框架选择时，尽量不要草率地做出决定。在全身心投入之前，应该首先看看是否可以部分地采用以增加了解。

（5） 如果不考虑具体实现细节，再好的设计也无法长久。

（6） 必须要将设计映射到对应的代码结构上，考虑如何组织代码树，以及在编译期和运行期采用哪种解耦合的模式。

（7） 保持开放，但是一定要务实，同时要考虑到团队的大小、技术水平，以及对应的时间和预算限制。

3 架构设计的总结

（1） 在架构设计领域，功能性降解拆分是最佳实践之一。

（2） 软件架构师应该着力于将大部分处理逻辑都归于不可变组件中，可变状态组件的逻辑应该越少越好。

（3） 软件架构师有必要设计并构造出一套组件依赖关系图，以便将稳定的高价值组件与常变的组件隔离开，从而起到保护作用。

（4） 设计软件架构的**目的**是为了在工作中更好地对这些组件进行研发、部署、运行以及维护。

（5） 软件架构设计最高优先级的目标就是保持系统正常工作。

（6） 软件架构设计的**主要目标**是支撑软件系统的全生命周期，设计良好的架构可以让系统便于理解、易于修改、方便维护，并且能轻松部署。软件架构

的**终极目标**就是最大化程序员的生产力，同时最小化系统的总运营成本。

（7） 优秀的架构师会小心地将软件的**高层策略与其底层实现隔离开**，让高层策略与实现细节**脱钩**，使其策略部分完全不需要关心底层细节，当然也不会对这些细节有任何形式的依赖。所设计的策略应该允许系统尽可能地推迟与实现细节相关的决策，越晚做决策越好。

（8） 一个系统所适用的解耦模式可能会随着时间而变化，优秀的架构师应该能预见这一点，并做出相应的对策。

（9） 软件架构设计的工作重点之一就是，将策略彼此分离，然后将它们按照变更的方式进行重新分组。

（10）一个良好的架构设计应该围绕着用例来开展。

（11）架构师的职责之一就是**预判未来**哪里有可能会需要设置**架构边界**，并决定应该以完全形式还是不完全形式来实现它们。

（12）架构师必须持续观察**系统的演进**，时刻注意哪里可能需要**设计边界**，然后仔细观察这些地方会由于不存在边界而出现哪些问题。

（13）架构设计的任务就是找到高层策略与低层细节之间的架构边界，同时保证这些边界遵守依赖关系规则。

（14）系统的架构是由系统内部的架构边界，以及边界之间的依赖关系所定义的，与系统中各组件之间的调用和通信方式无关。

4 个人总结

本书阅读完之后，笔者发现如果在考取高级系统架构师之前看，就会对架构风格有更好的理解。比如像管道-过滤器风格，隐式调用风格，其实这些风格都是为了将可变与不可变进行隔离，然后在调用的时候采用 DIP 原则，并时刻注意 ISP 原则。当初笔者学习设计模式的时候，总是认为这些只能用于源代码层面，对架构层面的理解只能限于表面。读完本书后，觉得 SOLID 的原则不仅对源码层面的设计模式有支撑，也能应用于架构设计。

另外，本书一再强调要划分边界，其实也是为了更好的架构设计。在“拾遗”部分，可以将细节实现用四种模式架构设计：按层封装、按功能封装、端口和适

配器、按组件封装，上述这些封装都在强调解耦合的模式重要性。笔者强烈建议先看《代码整洁之道》，再阅读本书，因为 Bob 大叔很多知识在前者中都有所提及，看起来有连贯感。

本书的附录 A 架构设计考古，也是很有意思的内容。该附录将了 Bob 大叔早年做系统开发的一些趣事，从笨重的电脑，半个房间的硬盘，软件还需要通过打孔机输入到内存中，然后交换读取数据，到最后 CleanCoders.com 网站的架构设计，针对每个项目都有很不错的架构设计经验。《代码整洁之道》和《架构整洁之道》两本可被称作为从程序员到架构师的入门经典书籍，建议所有未来希望做系统架构师的同学都学习一下。