**《软件架构》读书笔记**

**杨凤仪-2021141480325**

在本书中，作者描述软件架构具有动态性。事实也确实如此，随着时代的发展和计算机技术的进步，许多新的架构涌现，几年前完全有效的解决方案如今已无法使用。软件架构似乎是一个很难完全定义的概念，不过在本书中，作者仍旧给了我们一些认识上的指导。

那么究竟什么是系统的架构呢，在本书的描述中，系统架构是由系统结构、架构特征、架构决策和设计原则组成的。而我们常说的“微服务架构”，其实只是一种结构，结构是指系统中的架构风格类型。而架构特征，诸如可用性、可靠性、可测试性、可伸缩性、安全性等，在我看来或许有点类似于非功能性需求。而架构决策，在我看来就是一种规则和约束，限制了哪些接口可以访问哪些后台。不过架构决策不是绝对的，架构审查委员会和首席架构师可以“特事特办”，对一些例外情况权衡利弊后进行“放权”。最后就是设计原则了，设计原则和架构决策像是一正一负的两级，架构决策侧重于限制行为，而设计原则侧重于指导行为，鼓励开发者在多种可选的实现方案中，选择更符合设计原则的方案。



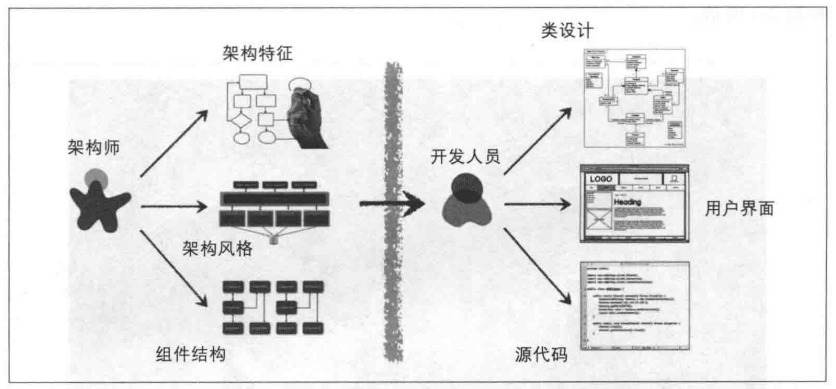
而软件架构师在本书中并没有进行严格的定义，作者认为这是浪费时间的，取而代之的是对于软件架构师的8项核心期望：指定架构决策、持续分析架构、掌握最新趋势、确保决策被遵守、丰富的经历和经验、具备业务领域知识、具备人际交往能力、了解并驾驭政治。通过对这一部分的阅读，我把这些期望归纳为：架构的设计和维护、团队的领导和管理、外界环境的适应和响应。架构的设计和维护，即架构决策、持续分析架构、确保决策被遵守、具备业务领域知识这些与软件架构的技术设计和更新维护直接相关的内容，这些是技术化、工程化的操作，而团队的领导和管理则与丰富的经历和经验、具备人际交往能力有关，这涉及到与开发人员的沟通，与管理层的协商，为自己的决策进行辩护和斗争，最后，外界环境的适应和响应，则与掌握最新的趋势、了解并驾驭政治有关，确保新的技术有益地作用于软件，并且符合法律的规范。

在完成系统的架构和对软件架构师的期望的定义后，我们就进入了工程实践的领域。工程实践与架构具有共生关系。软件开发领域的所有经验都表明，没有什么是静态的，在工程实践开始前，我们或许已经知道了已知的未知，即我们设计了开发人员必须学习将要用到的领域和技术，但该设计必须被实现，于是在开发过程中，未知的未知就会显现，而架构师没有办法对未知的未知进行设计，最后项目就成为了未知的未知的受害者。为了减少这样未知的未知对于软件的损害，Build Evolutionary Architectures引入了适应度函数来保护架构特征，将架构的迭代纳入考虑范围，让构建具有回弹行动的架构变得容易，从而降低了变化所带来的风险。

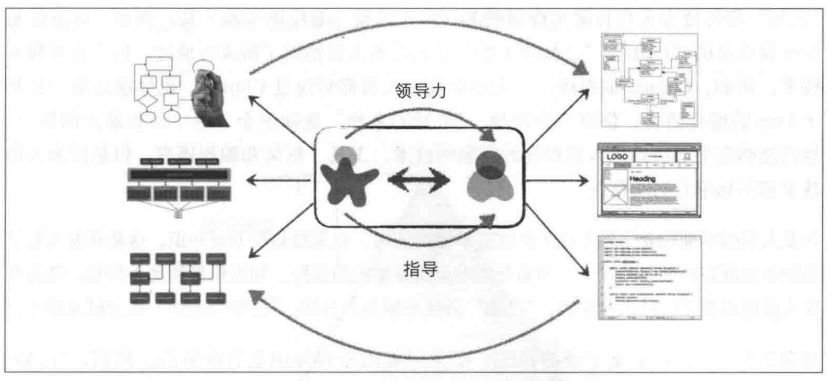
综上描述后，作者提出了软件架构的定律。软件架构第一定律：软件架构中的一切都是在做权衡。这种权衡可能涉及到性能、可扩展性、安全性、成本、开发时间、用户体验等多个方面。优化性能可能会增加复杂性，而设计高度可扩展的架构可能会对性能有所牺牲；增强安全性通常会增加用户的认证和授权步骤，可能会影响用户的使用便利性；增加更多功能可能会增加开发和维护成本，需要权衡功能的优先级和实现的复杂度；单体架构简单易懂，但微服务架构有利于扩展和独立部署，需要根据具体需求权衡选择……

另外，架构不仅仅是各种要素的组合，这点体现在软件架构第二定律中：原因比方法更重要。例如，架构设计的目标是为系统提供一个稳定、高效、可扩展的解决方案。如果忽略了背后的原因和需求，设计出来的架构可能无法解决实际业务问题。例如，过度关注系统的性能优化可能会导致不必要的复杂性，而忽视了系统本身并不需要那么高的性能。

在传统的架构和设计职责中，架构师的工作和开发人员的工作是割裂开的，双方之件缺少沟通。这就导致架构师所做的决策无法完全正确地传达给开发团队，而开发团队对于架构的改动也几乎不会反馈回架构师，最后导致架构落地与最初计划相差巨大。



在现代的项目开发中，通常采用迭代的开发方式，项目在迭代中不断进行改变和演进，每次迭代都保持这架构师和开发团队的沟通和合作，从而打破了传统方式中架构师和开发人员的壁垒，从而保证了项目落地和项目设计的一致性。



开发人员和架构师在技术方向的侧重有所不同，开发人员要注重技术的“深度”，而架构师更注重技术的“广度”，架构师要扩宽“已知的未知”，进而将“已知的未知”传达给开发人员，让开发人员将其转化为“已知”。所以对于架构师而言，更明智的做法是做到“博而不专”，扩宽自己专业知识的覆盖面从而支持架构设计，以便在设计架构时有更丰富的工具箱。

既然软件架构中的一切都是在权衡，那么权衡的因素究竟是什么呢？做权衡时，要理解业务驱动因素。即“看情况”，根据情况和每种架构方案的优点和缺点，判断哪一种架构方案更加符合我们业务驱动因素的要求，从而做出更合理的选择。

总结起来，《软件架构》一书通过详细分析软件架构的核心要素、架构师的职责以及工程实践的动态性，为我们揭示了软件架构在现代软件开发中的关键角色。书中强调架构决策的权衡性以及架构设计背后的原因比具体方法更为重要，帮助我们更好地理解了如何根据业务需求做出合理的架构选择。作为软件工程师，通过对架构知识的学习，我们不仅要掌握技术深度，还应拓宽视野，关注技术的广度，以应对不断变化的技术环境和业务需求。在实践中，架构与开发人员的沟通和合作也是至关重要的，只有通过持续的迭代与协作，才能确保架构设计与项目落地的高度一致性。

在未来的工作中，《软件架构》一书所提供的架构思想将为我提供极大的指导意义。无论是面对新的技术挑战，还是在项目迭代中进行架构的调整和优化，书中提出的架构决策权衡、适应度函数的应用以及架构师的核心期望等具体思想，都将帮助我在做出架构决策时更加从容与全面。例如，书中强调的权衡理念提醒我，在设计系统时要平衡性能、扩展性、开发成本等多方面的因素，避免在追求某一方面时牺牲了其他重要特性。同时，适应度函数让我意识到，架构设计不应只停留在静态阶段，而要随着项目的演进持续优化，以降低未来变化的风险。此外，关于架构师角色的描述，也让我意识到，不仅要具备丰富的技术知识，还需要提高与团队的沟通能力，掌握最新技术趋势，确保项目架构能够应对不断变化的业务需求。

这些具体的理念将为我未来在软件开发中的架构设计提供方向，帮助我在项目中更好地平衡短期需求与长期目标，打造出更具弹性和适应力的系统。前文中，我从软件架构的定义、对软件架构师的期望、工程实践与软件架构的关系、软件架构的定律、软件架构师和软件开发人员的关系，统述了《软件架构》该书的核心内容。接下来，我会走进软件架构的细节，对各个架构细节发表读后感。

在软件架构的过程中，我们或许会常常提到一个概念——“模块化”。如果不对整个软件进行模块的拆分，软件的设计和实现都会变得举步维艰，并且难以维护和扩展。在编程时，我们总是会关注代码中各个模块间的耦合，追其溯源，关注各模块之间的耦合性的责任应当落到软件架构师的身上。在软件架构的模块化中，我们需要关注三个关键概念：内聚性、耦合性、共生性。

内聚性是指软件系统中模块或组件内部各个元素之间的紧密程度，或者说它们在功能上关联的程度。内聚性是衡量模块化设计质量的重要指标之一，通常与模块的功能单一性和职责明确性相关。高内聚的模块意味着其中的各个元素紧密合作，共同完成单一的功能或目标。这种设计通常使得模块更加易于理解、维护和修改，同时减少了模块之间的依赖性。相比之下，低内聚的模块内部包含了许多彼此不相关的功能，这会导致代码复杂化、难以维护，并且更容易引发错误。与此同时，过度分割关联度较高的内容，会导致它们之间存在跨模块的耦合，导致耦合增加和可读性下降。本书介绍了由最佳到最坏的多种内聚结果：功能内聚性、顺序内聚性、联系内聚性、程序内聚性、时间内聚性、逻辑内聚性、偶然内聚性。

耦合性是指软件系统中模块或组件之间相互依赖的程度。它描述了不同模块之间的连接方式以及它们是如何进行交互的。耦合性与内聚性相对，内聚性指模块内部的紧密性，而耦合性则侧重于模块之间的关联。本书提到，相比于内聚，我们由更好的工具来分析代码库中的耦合，其原理一部分基于图论，Structured Design: Fundamentals of a Discipline of Computer Programd Systems Design使用数学分析量度耦合，指导架构师分析代码的耦合特征。

同时在设计软件架构时，我们也要考虑抽象性，抽象性是软件设计中的一种重要原则，指的是通过隐藏系统内部的复杂细节，仅暴露必要的功能或接口，从而简化对系统的理解和使用。抽象性通过提供一种简化的、高层次的视图，使开发者能够专注于系统的主要功能和概念，而不需要关心底层的实现细节。微服务架构是一个典型的例子，服务间通过 API 进行通信，而不需要关心每个服务的内部实现。当一个项目保持了较好的抽象性可以得到很多好处。一方面，隐藏细节使得代码的修改和维护变得更容易，尤其是在实现发生变化时，抽象保证了代码的稳定性。另一方面，通过定义高层的抽象，多个实现可以复用同一个接口或抽象层，大大提高了代码复用的可能性。并且，抽象允许底层实现的变化不影响上层逻辑，使系统在面对需求变化或技术更新时更加灵活。

下面，我将结合《软件架构》一书中所学会的架构知识，对于我将要制作的一个点击向解密游戏进行软件架构。

对于一个游戏的架构，这需要采用面向对象的编程思想。我使用层次化的模型进行架构。在《软件架构》中提到，架构的设计涉及多方面的权衡，例如性能、扩展性、安全性、开发时间等​。分层架构很好地反映了这一思想：它在模块化的基础上，通过清晰的职责划分，提高了扩展性和可维护性。同时，分层设计使得每层可以独立优化。例如，输入层可以优化响应时间，而逻辑层可以专注于确保谜题状态的正确性和一致性，这样在不同需求的变化下，我可以针对性地调整每个层次，而不必对整个系统进行大规模重构。

另外，在《软件架构》一书中，提到系统架构应该具备动态性和弹性，以应对未来变化。分层架构正是这样一种动态的架构设计，它允许各个模块独立演化和迭代。你可以在不影响其他部分的情况下，逐步引入新的功能、优化现有模块，甚至进行技术更新。这样的设计使得你的游戏可以适应未来的需求变化，同时保持较低的维护成本。

下面是我的分层架构的具体内容：

输入层：处理玩家的点击交互行为

逻辑层：处理游戏的核心逻辑，例如谜题的状态管理、游戏进度的存储与更新。

表现层：负责游戏画面、音效、动画等反馈效果。

数据层：存储游戏数据，例如谜题进度、解锁状态、玩家存档等。

图示

中度可信度描述已自动生成

表现层呈现游戏画面，用户点击游戏对象，在输入层通过CursorManager获取用户输入，判断点击对象是Item（游戏道具）还是Interactive（互动点），被点击的道具进入把背包，这些数据由InventoryManager维护一个List，而原本在场景中的道具是否被拾取、场景互动点是否已经被互动过这些数据以键值对的方式在ObjectManager中进行维护。从而实现了表现层到输入层到逻辑层，最后再到数据层的交互。

我的设计具有高内聚性。每个管理器（如InventoryManager、ObjectManager）专注于单一职责，分别负责道具管理、场景数据管理等。例如，InventoryManager只负责管理道具的拾取和存储，而不会涉及与谜题逻辑或场景互动的处理，这种高内聚性确保了模块内部结构清晰，易于维护和修改。这种职责明确的设计意味着每个模块内部功能关联性强，且模块功能相对独立。

我在架构中采用了分层设计（输入层、逻辑层、表现层、数据层），这有助于降低模块间的耦合性。每个层次之间通过特定的接口进行交互，如CursorManager负责输入的处理，InventoryManager负责道具管理，而ObjectManager维护场景数据。这种分层设计能有效减少模块间的直接依赖，降低耦合性。例如，输入层通过CursorManager将用户点击行为传递到逻辑层，而不需要了解逻辑层的具体实现。这种通过接口或中介传递数据和命令的方式，可以避免不同模块之间的强耦合，使得修改某个层的实现不会对其他层产生过多影响。如果我之后想要添加一个可攻击的游戏对象，我只需要在逻辑层新增一个Attacker模块，并更新CursorManager中的点击对象检测相关的内容就可以了，对其他模块不会产生任何影响。

抽象性体现在你的架构中如何通过隐藏实现细节，提供高层次接口。通过引入不同的管理器（如CursorManager、InventoryManager、ObjectManager等），你在不同层次之间划分了清晰的职责。上层模块只需要通过这些抽象的接口调用相应的功能，而无需了解底层的具体实现细节。例如，CursorManager对输入层进行了抽象，表现层只需与CursorManager交互获取用户输入，而不需要知道具体的输入处理逻辑。同时，ObjectManager管理场景中对象的状态，也为逻辑层和数据层之间提供了良好的隔离。

另外，软件架构特征包括可配置性、可扩展性、可安装性、可利用性、本地化、可维护性、可移植性。对于我的这个点击向游戏，拥有设置UI来保证游戏的可配置性，更改游戏的音效、音量、文字流动速度、存档读档等配置；分层的架构和模块化的涉及保证了软件的可扩展性；而Unity提供了游戏的打包功能，并且支持多平台安装，安装只需要下载exe文件就可以安装，具有可安装性；Unity内部也可以通过Package Manager导入通用组件，具有可利用性；游戏也可组织多种语言包，可进行本地化；受益于分层架构和模块化设计的灵活性，保证了软件的可维护性；另外Unity支持的多平台开发也保证了游戏的可移植性，可以在Windows、Mac、手机端进行运行。

通过设计点击向解密游戏的架构，我深入体会到了《软件架构》中许多核心理念的实际应用价值。从分层架构的使用到模块化设计的实施，这一过程不仅验证了高内聚性、低耦合性的重要性，也展现了动态性和适应度函数在实际项目中的作用。例如，通过分层设计，我能够有效管理输入、逻辑、表现和数据等多个维度的需求，而模块化的实现则使得系统易于维护和扩展。

与此同时，案例中对抽象性原则的运用，不仅提高了代码复用和可读性，更让我意识到隐藏实现细节对系统灵活性和稳定性的重要性。在权衡性能、扩展性和开发成本时，我也切身体会到了《软件架构》中强调的“权衡”是一切架构决策的核心。这种架构设计的思维模式，将帮助我在未来的工作中更从容地应对复杂项目中的各种挑战。

未来，我将继续将《软件架构》的理论与实践相结合，探索架构设计的更优解。无论是面临新的技术挑战，还是在项目中不断优化现有架构，书中提到的动态性设计、架构特征的适应度函数、以及架构师的职责和能力要求，都将成为我的重要参考。通过不断积累经验，我希望能够设计出更多具有弹性、适应力和持久价值的软件系统。