Part1　什么是软件架构（SA）

第一章 什么是架构

1、架构是什么？

（1）架构作为名词来解释时，概括起来都与结构有关：将产品分解为一系列组件、模块和交互。

（2）架构作为动词来解释时，包括了理解你需要构建什么、设定愿景以便于进行构建和做出恰当的设计决策。所有这些都要以需求为基础，因为需求驱动架构。

第二章 架构的种类

2、架构的种类/工厂领域都有哪些架构？它们的共同之处是什么？

❑基础设施；❑ 安全；❑ 技术；❑ 解决方案；网络；❑ 数据；❑ 硬件；❑ 企业；❑ 应用程序；❑ 系统；❑ 集成；❑ IT；数据库；❑ 信息；❑ 流程；❑ 商务；❑ 软件。

这其中重点为：

（1）解决方案架构：对一些组织来说，“解决方案架构师”就是“软件架构师”，而有些组织则有一个特定的专注于整体“方案”设计（但不包括实施细节的讨论）的角色。

（2）技术架构：通常指软件、硬件，或者两者兼有。

（3）软件架构

<共同之处>

除了都以“架构”或“架构师”结尾之外，所有架构类型都具有结构和愿景。

第三章 软件架构是什么

3、软件架构的标准定义？

对我而言，软件结构就是应用程序和系统架构的结合。换句话说，从代码结构和基础到将代码成功部署到生产环境，与一个软件系统重要元素相关的所有东西就是软件架构。

4、什么是低层的细节？什么是高层次的的问题？（仅了解）

<低层>

在这里，我们考虑的是有助于构架更好软件的东西，比如面向对象的原则、类、接口、控制反转、重构、自动化单元测试、代码整洁和其他不胜枚举的技术实践。

<高层>

❑ 横切关注点，比如登录和异常处理；❑ 安全性，包括认证、授权和敏感数据保密；

❑ 性能、可伸缩性、可用性和其他质量属性；❑ 审计及其他监管需求；

❑ 客观环境的约束；❑ 互操作性、与其他软件系统的集成；

❑ 运营、支持和维护的需求；❑ 结构和整个代码库解决问题、实现特性的方法的一致性；

❑ 评估正在构建的基础有助于交付按计划进行。

第四章 敏捷软件架构是什么

5、什么是敏捷软件架构？

人们用“敏捷”一词指代的往往不止一件事情。首当其冲就是软件开发的敏捷方法；快速行动，拥抱变化，持续交付，接收反馈，不一而足。先把这些“肤浅的东西”放到一边，在我看来，给软件架构打上“敏捷”的标签就意味着它能够应对所处环境中的变化，适应人们提出的不断变化的需求。

6、OODA 循环（仅了解）

美国空军战斗机飞行员约翰·博伊德（JohnBoyd）提出了一个名为 OODA 循环的概念。本质上，这个循环构成了基本的决策过程。

OODA 是 Observation（观察）、Orientation（判断/调整）、Decision（决策）、Action（行动）四个单词的首字母组合。这一循环过程强调通过不断地观察环境、判断形势、制定决策并迅速执行，以适应环境的变化并占据主动。

7、知识点

（1）敏捷是相对的，且按时间来衡量。如果你的软件团队交付的软件跟不上所处环境的变化，就不算敏捷。
（2）好的软件架构能带来敏捷。
（3）不同的软件架构提供不同层次的敏捷。

第 5 章 架构对上设计

8、知识点

（1）**需求**是架构的驱动力。
（2）**所有架构都是设计，但并非所有设计都是架构。**
（3）架构反映了使一个系统成型的重要设计决策，而重要性则通过改变的成本来衡量。
（4）从本质上讲，格雷迪认为**重要决策**即“架构”，其他的都是“设计”。

9、为软件项目/软件系统的架构决策做一个清单，并举一个具体的例子。

在我们的软件系统中哪些可能是重要的（或者说“架构的”）。比如说，这可能包括：

- ❑ 系统的形态（例如，客户端－服务器、基于 Web、原生移动客户端、分布式、异步，等等）；
- ❑ 软件系统的结构（例如，组件、层、交互，等等）；
- ❑ 技术选择（即编程语言、部署平台、数据库，等等）；
- ❑ 框架选择（例如，Web MVC[插图]框架、持久性/ORM 框架，等等）；
- ❑ 设计方法/模式选择（例如，针对性能、可伸缩性、可用性等的方法）。

<具体例子>

电子商务平台架构决策清单示例

1. **系统形态决策**
   - **决策**：我们的软件系统将采用基于 Web 的形态，用户将通过 Web 浏览器访问服务。
   - **理由**：基于 Web 的系统无需用户安装额外的软件，易于访问且跨平台兼容性好。
2. **软件系统结构决策**
   - **决策**：系统将分为前端和后端两部分，前端负责用户界面展示，后端负责业务逻辑处理和数据存储。
   - **理由**：前后端分离可以提高系统的可维护性和可扩展性，前端可以专注于用户体验，后端可以专注于数据处理。
3. **技术选择决策**
   - **编程语言**：前端使用 JavaScript（结合 React 框架），后端使用 Java（结合 Spring Boot 框架）。
   - **部署平台**：选择云服务提供商（如 AWS、Azure）进行部署，以便灵活扩展和降低运维成本。
   - **数据库**：使用 MySQL 作为关系型数据库，存储用户信息和订单数据。
   - **理由**：这些技术成熟稳定，社区支持广泛，且能够满足我们的业务需求。
4. **框架选择决策**
   - **前端框架**：选择 React，因为它组件化、性能优越，且生态丰富。
   - **后端框架**：选择 Spring Boot，因为它简化了 Java 应用的开发、配置和部署。
   - **持久性/ORM 框架**：使用 MyBatis 作为持久层框架，因为它轻量级且易于与 MySQL 集成。
   - **理由**：这些框架能够加速开发过程，减少重复劳动，提高代码质量。
5. **设计方法/模式选择决策**
   - **性能优化**：采用缓存机制（如 Redis）减少数据库访问压力，提高响应速度。
   - **可伸缩性**：设计微服务架构，每个服务独立部署和扩展，以适应不同业务量的需求。
   - **可用性**：实施负载均衡（如 Nginx）和故障转移机制，确保系统在高并发和故障情况下仍能稳定运行。
   - **理由**：这些方法/模式能够提升系统的性能、可伸缩性和可用性，满足用户对高质量服务的需求。

第 6 章 软件架构重要吗

10、软件架构重要吗？它能带来哪些好处？

- ❑ 让团队跟随一个清晰的愿景和路线图，无论这个愿景是一人所有还是整个团队共有；
- ❑ 技术领导力和更好的协调；
- ❑ 与人交流的刺激因素，以便回答与重要决策、非功能需求、限制和其他横切关注点相关的问题；
- ❑ 识别和减轻风险的框架；
- ❑ 方法和标准的一致性，随之而来的结构良好的代码库；
- ❑ 正在构建的产品的坚实基础；
- ❑ 对不同的听众，以不同层次的抽象来交流解决方案的结构。

缺乏软件结构将引发的问题

- ❑ 你的软件系统有良好定义的结构吗？
- ❑ 团队里每个人都以一致的方式实现特性吗？
- ❑ 代码库的质量水平一致吗？
- ❑ 对于如何构建软件，团队有共同的愿景吗？
- ❑ 团队里每个人都得到了足够的技术指导吗？
- ❑ 有适当的技术领导力吗？

11、所有软件项目都需要软件架构吗？

我不会给出“看情况”这种典型的咨询式回答，相反我会说答案毫无疑问是**肯定的**，并提醒每个软件项目都应该考虑多种因素，以评估必需多少软件架构的思考。

Part2　软件架构（SA）的角色

第 8 章　软件架构的角色

12、软件开发者和软件架构师的区别？

软件开发者负责具体功能，偏底层；软件架构师从总体角度出发，偏技术类。

13、软件架构角色有哪些？（重点）

注意，我这里说的是“角色”；它可以是一个人，也可以由团队共同扮演。

<简略版>

(1) 架构驱动力：捕捉和挑战一套复杂的非功能需求，还是简单地假设它们的存在。（客户会主动提到功能需求）

(2) 设计软件：从零开始设计一个软件系统，还是扩展已有的。

(3) 技术风险：证明你的架构能够工作，还是盲目乐观。

(4) 架构演化：持续参与和演化你的架构，还是把它交给“实现团队”。

(5) 编写代码：参与交付的实践部分，还是袖手旁观。
(6) 质量保证：保证质量并选择标准，还是反其道而行之或无所作为。

<详细版>

（1）架构驱动力

这个角色首先要理解业务目标和管理架构驱动力，其中包括需求（功能性需求和非功能性需求）和环境的限制。

（2）设计软件

这涉及要理解如何解决架构驱动力带来的问题，创建软件系统的整体结构，并为交付设定一个愿景。软件设计的一个关键部分是技术选择，这通常是一个有趣的练习，但也有一定的挑战。

（3）技术风险

技术选择其实就是风险管理，当复杂度或不确定性高的时候降低风险，有利可图时再冒点险。技术选择不能太新也不能太旧。概括地说，就是主动发现、减轻和承担高优先级的技术风险，保证架构的运转。

（4）架构演化

在整个交付过程中依据不断变化的需求和团队反馈来对其演化。如果架构师创建了一个架构，为什么在整个交付过程的其他时候不自己拥有和演化这个架构？这关乎持续的技术领导，而不是仅仅参与生命周期的开始阶段，然后泰然处之、袖手旁观。

（5）编写代码

编码为架构师提供了一种与团队分享软件开发经验的方式，从而帮助他们更好地理解如何从开发的角度看待架构。许多公司都有阻止软件架构师参与编码工作的政策，因为他们的架构师“太宝贵了，不该承担日常编码工作”，这显然是错误的。不应该因为“我是架构师”，就把自己排除在编码之外。

（6）质量保证

质量保证应该是软件架构角色的一部分，但它的内容不只是代码评审。你要保证一条基线，它可以是引入一些标准和工作实践，如编码标准、设计原则和工具。质量保证也包括确保团队对架构实现的一致。管它叫架构服从还是架构一致取决于你，但都要遵循技术愿景。

可以肯定地说，大多数项目没有做足够的质量保证，因此，你要弄清楚什么是重要的，并确保它有充分的保证。对我来说，只要是架构上显著的、业务上关键的、复杂的和高度可见的，都是一个项目的重要组成部分。

14、知识点（重要）

（1）**合作**很重要。一个软件系统很少孤立存在，可能有不少人要为整个架构过程作贡献。这包括了从需要**理解和认同架构的直接开发团队**，一直到那些对**安全**性、数据库、运营、维护或支持感兴趣的人组成的**扩展团队**。

（2）软件领导是一个角色而非级别。软件架构的角色基本上就是向软件团队引入**技术领导**，有必要重申的是，我这里谈论的是一个角色，而非职务级别。晋升软件架构师。

（3）多种对 SA 角色的称呼：架构师、技术主管、首席设计师等。

第 9 章 软件架构师应该编码吗

15、软件架构师应该编码吗？

软件架构师应该编码，保持自己的技术水准，**先完成自己的本职工作，有余力时编码**，别把时间都花在编码上。我的建议是让编码成为你作为软件架构师角色的一部分，不见得要成为团队里写代码最厉害的，只要把自己当作软件开发团队的一份子就行了，参与到实践和交付流程的好处非常大。

软件架构师：负责设计软件系统最高层次结构的专业人士。工作包括：定义软件系统的整体结构，选择合适的技术栈，制定设计模式和需求。

16、知识点

（1）我也不推荐把代码评审作为一个长期的战略，但参与或做**代码评审**至少能让你**了解新技术及其应用**。

（2）团队不让写代码的

参与方式：如果是这样的话，**对软件系统中有疑问的概念构建原型和证明**是一个很好的参与方式。这让你可以和团队建立起融洽的关系，也是评估你的架构是否管用的好办法。

替代方式：作为替代，你可以帮助建立团队可用的框架和基础。

（3）架构师要保持一定的技术知识（开源项目，最新语言，框架等）

第 10 章 软件架构师应该是建造大师

17、知识点

（1）成长路径：软件工程师->设计师->架构师

（2）石匠大师，就是石头的操作者、艺术家和设计师。这句话同样适用于我们软件开发者。

“象牙塔”通常用来形容那些过于理想化、脱离实际、不关心社会现实或缺乏实践经验的学术环境或学者。软件架构不是象牙塔。

（3）软件架构师被看作是首席设计师，是把项目的每件事**凝聚**在一起的人。

第 11 章 从开发者到架构师

18、区分软件架构和软件设计的关键因素?

软件开发和架构之间的界线某种程度上是模糊的，区分软件架构和软件设计的关键因素包括:

规模的扩大、抽象层级的增加、做出正确设计决策的意义等。软件架构就是总览全貌，看清**“大局”**，才能理解软件系统整体如何工作。

软件开发者升级为软件架构师要做到：

经验，看的更深，满足需求

(1) 架构驱动力：捕捉和挑战一套复杂的非功能需求，还是简单地假设它们的存在。

(2) 设计软件：从零开始设计一个软件系统，还是扩展已有的。

(3) 技术风险：证明你的架构能够工作，还是盲目乐观。

(4) 架构演化：持续参与和演化你的架构，还是把它交给“实现团队”。

(5) 编写代码：参与交付的实践部分，还是袖手旁观。

(6) 质量保证：保证质量并选择标准，还是反其道而行之或无所作为。

其中大部分可以归结为是承担寻找方案的责任还是推诿问题。

第 12 章 拓展 T

“T”指的是**技术（Technique）**，这也正是优秀的软件构架师应该懂得的。作为软件开发者，我们倾向于去**搞懂编程语言语法、API、框架、设计模式、自动化单元测试**和其他所有日常使用的底层技术。对一个软件构架师来说，这些也是基础知识。因为扮演软件构架角色的人要懂技术，这样他们至少能如实回答以下类型的问题。**该方案是否有效？我们要这样去构建吗？**

避免受困于单一的技术栈。

19、软件构架角色的要求有哪些？

技术知识面宽对软件构架师来说也很重要。**大局观和底层细节间自由切换，技术领导中的编码最好者，码农中领导最好者。**软件架构师是**通才型专家**。当然，他们可能是 Java 或者 Oracle 专家，但软件构架角色的要求更高。例如，他们要能够回答以下类型的问题。

- ❑ 和其他可选技术相比，我们所选的是否最合适？
- ❑ 对该系统的设计和构建，还有哪些选择？
- ❑ 是否应该采用一种通用的架构模式？
- ❑ 我们是否明白所做决策的利弊？
- ❑ 我们照顾到了品质属性的需求吗？
- ❑ 如何证明这种架构行之有效？

第 13 章 软技能

20、有哪些软技能？

- ❑ 领导力：简单来说，领导力就是创造共有的愿景，并带领人们向着共同目标前行的能力。

❑ 沟通：你有世界上最好的想法和愿景，但如果不能有效地传达给其他人，也是死路一条。这包括了软件开发团队内外的人，要使用适合受众的语言和细节水平。
❑ 影响力：这是重要的领导技能，从毫不掩饰的劝说到神经语言编程或绝地控心术，它能够以多种途径实现。通过妥协和谈判也可以达到这样的目的。每个人都有自己的想法和计划，你在处理时还得让他们都不反感，并主动地去追求你需要的结果。好的影响力也要求好的倾听和探索能力。
❑ 信心：信心很重要，是有效的领导力、影响力和沟通的基础。但信心不代表傲慢。
❑ 合作：软件架构角色不应该被孤立，（与其他人）合作想出更好的方案是一项值得实践的技能。这意味着倾听、谦虚和响应反馈。
❑ 指导：不是每个人都对你正尝试做的事情有经验，你需要对他们进行角色、技术等方面的指导。
❑ 辅导：辅导是对人进行学习方面的指引，而非告诉他们怎么做一件事。作为领导，你可能会被要求去辅导团队中的其他人。
❑ 动力：这说的是保持团队愉快、开朗和积极。团队要有积极性，才会跟随你这个软件架构师所创建的任何愿景。你还要面对团队中一些人不买账的局面。
❑ 润滑剂：你经常需要退后一步，促进讨论，特别是团队内有不同意见时。这需要探索、客观，帮助团队达成共识。
❑ 政治：每个组织都少不了政治。我的咒语是，离得越远越好，但你至少应该明白周围发生了什么，这样才能做出更可靠的决策。
❑ 责任感：你不能因为失败就责备软件开发团队中的其他人，有责任感对你而言很重要。如果软件架构不能满足业务目标，无法交付非功能性需求或技术品质很差，那都是你的问题。
❑ 授权：授权对任何领导角色来说都是一个重要部分，作壁上观和事必躬亲之间有一条模糊的界线。你应该学会在适当的时候授权，但请记住，你授权的可不是责任。

21、知识点：

（1）根据是否具有硬性指标，区分软科学和硬科学。
（2）根据是否看得见摸得着，区分软件和硬件。
（3）根据什么区分硬技能和软技能？
（4）**软件架构师要保持积极**

第 14 章 软件架构不是接力运动

22、知识点

（1）软件架构不是接力运动。
（2）更敏捷的软件团队的组成
①一项核心专长，更多综合知识和经验的通才
②收集需求和架构到编码和部署
③团队大，需要技术领导
（3）有很多人，特别是在大型组织里，自称“**解决方案架构师**”或“技术架构师”。他们**设计好了软件**，为自己的**方案编写文档**，然后扔给一个单独的**开发团队**。这个方案一旦“完成”，架构师就会去别的地方重复这个过程，甚至往往对开发团队的进展看都不看一眼。这种行为是错误的，应该盯着整个项目，要有人负责大局。**到新项目，原项目开发团队推脱架构责任。**
（4）**要有人负责大局**：创建愿景，交流，演化，避免接力运动后不管不顾是否顺利交付。

第 15 章 软件架构要引入控制吗

23、软件架构要引入控制吗？

（1）引入控制和约束才能**提供指导，追求一致性**。
若缺乏控制，则项目混乱，不一致；引入控制和约束才能实现指导和一致性。
（2）控制也可以只是保证你的代码库有一个清晰一致的结构。
（3）控制的程度很重要。一种方法是**独裁**，任何人都不能擅自做决策；另一种方法是**放手**，没有人会得

到任何指导。这两个极端都不对。

24、软件架构要引入多少控制？（操纵杆而非按钮）

我只能给出一个咨询式的回答，在不清楚的情况下，这取决于以下这些事：

- ❑ 团队是否经验丰富？
- ❑ 团队以前一起工作过吗？（合作）
- ❑ 团队有多大？（规模）
- ❑ 项目有多大？
- ❑ 项目的需求复杂吗？
- ❑ 有没有需要考虑的复杂的非功能需求或限制？
- ❑ 日常的讨论是什么样的？
- ❑ 团队或已有的代码库是否看起来已经混乱不堪？（代码整洁性）

我的建议是，先**从部分控制开始，倾听反馈**，以便随着项目的推进再**微调**。如果团队老是问“为什么”和“怎么办”，那可能就需要更多指导。如果团队好像总是在和你对着干，可能你就是把操纵杆推得太多了。

理论上可以从完全控制开始，但实际上行不通。会造成逆反心理。

第 16 章 小心鸿沟

25、知识点

（1）开发者关注底层细节。我们都梦想在这样的团队中工作：所有人都经验丰富，从代码到架构，对软件考虑得面面俱到。然而现实并非如此，**成员经验参差不齐**，如果你的团队**只关注底层细节而忽略大局**，一切就毫无用处。（**团队内的鸿沟**）

（2）不要成为**闭门造车的独裁架构师**。虽有大局，但脱离团队造成鸿沟；造成团队不认可，不执行架构。

26、如果你是软件架构师，如何消除鸿沟？

让团队理解大局共同愿景和决策理由，讨论架构的合理性，参与项目的日常开发工作。

- ❑ 包容与合作：让开发团队参与软件架构的过程，帮助他们了解大局，认同你所做的决策。确保每个人都明白决策背后的原理和目的，会对此有所帮助。
- ❑ 动手：如果可能的话，参与一些项目的日常开发工作来提高你对架构交付的理解。了解软件的底层如何工作会让你更透彻地了解开发团队对架构的感受，也会为你提供有价值的信息，可以用来更好地塑造架构。

第 17 章 未来的软件架构师在哪里

27、知识点

（1）指导、辅导和师徒关系：我认识的大多数架构师正是因为他们在一个或多个技术领域有大量的经验，才当上架构师的。架构师应该通过分享一些自己的经验来帮助别人。指导和辅导提供了一种方式来增强人们的技能，帮助他们完善自己的职业生涯。这种协助有时候是技术性的，有时候是软技能。

（2）我们正在失去技术导师：可悲的是，在我们的行业里许多开发者为了在企业晋升机制中有所发展，被迫转向非技术的管理岗位。**技术刚到管理岗，如：雷军**

（3）软件团队需要休息：最资深的技术导师流失，技术员和项目风潮压力大，为了提高，软件开发团队需要一些时间**远离日常工作，进行思考**。

第 18 章 每个人都是架构师，除非他们有其他身份

28、知识点

（1）每个人都是架构师：经验丰富、能够在**大局内外自如切换**的软件开发者，真正**会动手的架构师**，能成功**解决非功能需求、约束**，等等问题，不会漏掉任何一样。从**技术**的角度来看，这就是一个**自组织**（主动去学）的团队。

（2）除非他们有其他身份：罗伊·奥谢洛夫描述了他的“**弹性领导**”概念，这种领导风格需要根据团队的成熟度有所变化。罗伊用一个简单的模型对团队成熟度做了分类，每个等级需要不同的领导风格。

a. 生存型（混乱）：需要一种直接指挥和控制的领导风格。

b. 学习型：需要一种指导的领导风格。

c. 自组织型：需要简易化来确保平衡不受影响。（既是学习型又是自我指导型）

（3）**敏捷软件项目仍然需要架构**，因为那些围绕**复杂非功能需求和约束**的棘手问题不会消失，只是对架构角色的执行不同。集体代码所有制，每个人在架构层次上工作。

第 19 章 软件架构咨询师

29、知识点

（1）对**业务领域知识**的了解必不可少。对业务领域的了解可以帮助你更好地理解目标和建立成功的软件产品。你需要敏锐的分析能力来理解业务领域的关键部分，而不陷入分析瘫痪的恶性循环。

比如在金融行业：基金管理，投资银行，零售银行等。

方法：**由单一领域知识扩张到多领域知识；需要敏锐的分析能力来理解业务领域。**

（2）软件架构的角色意味着**技术领导力**，也就是你要让整个团队朝着同一方向前进。如果你负责一个软件系统的软件架构和技术交付，就必须有**决策权**。这就是施展**软技能**的地方，特别是**建设关系、建立信任和激励团队**相关的软技能。

Part3 设计软件

第 21 章 架构驱动力（<-需求<功能性需求和非功能性需求>）

1、驱动、影响和塑造了最终的软件架构的主要要素有哪些？

1. 功能需求（容易识别，比较清楚）：特性或用户故事清单（比如 Scrum 产品订单），即使粗糙短小，也是必不可少的。**需求驱动架构。**

2. 质量属性（非功能需求）（需要识别，不清楚）：非功能需求代表的质量属性反映了服务等级，如**性能、可伸缩性、可用性、安全性**等。这些属性主要是技术方面的，可以对最终的架构产生巨大影响。

3. 约束：你任职的组织可能对**技术选型**（时间、预算）、**部署平台**等有一系列细致的约束，能做什么，不能做什么。

4. 原则：原则是你为了将一致性和清晰度引入最终代码库而想采用的原则即**代码原则（例如编码规范、自动化测试的使用等）或架构的原则（如分层策略，架构模式等**）。

5. 理解影响：软件架构谈论的是重要的设计决策，其重要性以变动的成本来衡量。对于那些从根本上塑造了最终软件架构的重要决策而言，**起点是在高层次上对需求、约束和原则的理解。**

2、知识点

（1）需求驱动架构。需求分为功能性需求和非功能需求。

第 22 章 质量属性（非功能需求）

1、知识点

（1）功能需求：功能愿望清单，用户故事、用例、传统的需求规格书、验收标准等

（2）非功能性需求通常被看作是“能力”，主要跟服务质量有关。服务质量（SaaS（软件即服务）/AaaS）

2、常见的质量属性有哪些？

1. 性能：性能就是一个东西有多快，通常指响应时间或延迟。

2. 可伸缩性：可伸缩性基本上就是软件处理更多用户、请求、数据、消息等的能力（用户规模）。**可伸缩性和并发机制**密不可分，因此能在相同的时间内处理更多的东西（比如每秒的请求）。

3. 可用性：可用性是软件对服务请求的可操作和可见程度。你常会看到用“9”来衡量或指代可用性，如 99.99%或 **99.999%**。这些数字指的是**正常运行**时间的百分比。

4. 安全性：安全性涵盖了从**认证**和**授权**到**数据在运输和储存**中的**机密性**的所有事情。

5. 灾难恢复：人为，非人为的天灾

6. 可访问性：让视觉障碍之类的残疾人也能使用你的软件。（**残疾人友好**）

7．监测：将软件与平台特定的监测功能（比如 Java 平台的 **JMX**）集成，或发生故障时向集中监测仪表发送警报（比如通过 **SNMP**）。
8．管理：监测通常提供一个软件系统的只读视图，有时会有运行时管理需求。
9．审计：通常这些日志需要捕获与变动由**谁做出、什么时候做出以及为什么做出**相关的信息。
10．灵活性：非技术人员修改软件内部使用的业务规则的能力。
11．可扩展性：扩展软件使其可以做一些现在还不能做的事的能力，也许是通过**使用插件和 API**（扩展服务）。
12．可维护性：代码库以后将由谁来维护，遵循的**架构和开发原则**
13．法律法规：必须遵守一些规则（如**反洗钱**）。
14．国际化（i18n）：国际化是指以多种语言交付软件中用户可见元素的能力。有些语言是**从右向左**写。
15．本地化（l10n）：数字格式等
有时候，国际化和本地化也统称为“全球化”。

第 23 章 处理非功能需求

1．捕捉
客户明确给出非功能需求信息的次数屈指可数。**往往只提出功能性需求或者隐含的非功能需求**，面对这种情况，**架构师要**主动出击，自己去捕捉它们，**主动思考、问、捕获**。
2．提炼
模糊到量化（快->快的指标；安全->EU）
3. 沟通
成本 vs 实现高规格非功能需求

第 24 章 约束

1、约束的形态和大小不尽相同，有几类？
1.时间和预算（以及法律法规，道德）的约束
2.技术约束
- ❑ 批准的技术清单
- ❑ 系统整合和批准的技术清单
- ❑ 目标部署平台（这包括嵌入式设备、微软的 Windows 或 Linux 服务器的可用性，以及云。）
- ❑ 技术成熟度
- ❑ 开放源代码
- ❑ 供应商“关系
- ❑ 内部知识产权

3.人员约束：（团队规模，技能，扩展性，维护团队匹配性）
- ❑ 你的开发团队有多大？
- ❑ 他们有什么技能？
- ❑ 如果你的开发团队需要扩展的话，能有多快？
- ❑ 如果需要的话，你能够提供培训、咨询和专家吗？
- ❑ 如果在交付后转交你的软件，接手的维护团队拥有和你的开发团队相同的技能吗？

4.组织约束
- ❑ 软件系统是战术或战略实施的一部分吗？
- ❑ 组织政治有时能阻碍你实现真正想要的解决方案。

2、知识点
（1）约束可以划分优先级（动态的）。
（2）在时间紧迫时，时间优先级约束高于技术清单

第 25 章 原则

1、约束和原则的区别？

约束是强加于你的，而原则是你为了将标准方法和一致性引入构建软件的方式而想采用的。通用的原则很多，有些跟开发相关，其他则跟架构相关。

2、开发原则有哪些？

- ❑ 编码标准和规范
- ❑ 自动化单元测试
- ❑ 静态分析工具

3、架构原则有哪些？

- ❑ 分层策略（你可以把软件系统解构为 UI（User Interface，用户界面）层，业务层和数据访问层。）
- ❑ 业务逻辑的位置
- ❑ 高内聚（组件之间）、低耦合（组件内部）、SOLID 等
- ❑ 无状态组件（可以通过复制组件来对系统进行横向扩展从而分担负载）
- ❑ 存储过程
- ❑ 域模型：丰富与贫瘠
- ❑ HTTP 会话的使用（会话（session）支持对象到底存储在哪里，服务器出现故障时会发生什么）
- ❑ 始终一致与最终一致（用数据一致性换取性能或可伸缩性）

第 26 章 技术不是实现细节

1、架构图应该包括技术的选择/技术选型，技术不是实现细节，原因如下：

1．你有**复杂的非功能需求**吗

考虑：比如高性能或可伸缩性，不是所有技术都可

不考虑：选择任何技术都可

2．你有**约束**吗

约束能（且会）影响你能给出的软件架构。可以用各种手段挑战约束，但不能忽视

3．你**有一致性**吗

你的代码库真的应该只有一个日志、依赖注入或对象关系映射框架。缺乏一致性的方法会导致代码库难以理解、维护和增强。

2、一种技术选型的例子

第 27 章 更多分层等于更高复杂度

1、知识点

（1）分层不是越多越好。

2、金融风险系统案例

需求：分发数据到企业局域网用户的一个子集

解决方案：允许用户通过一个内部的 Web 应用程序访问数据。

ASP.NET or Silverlight（2007-2021）

替代 1:Angularjs-动态 Web 应用框架

替代 2:React--Facebook 构建 Ul 的 JavaScript 库

替代 3: Vue.js-构建 UI 的轻量框架

但这导致了更多的问题：

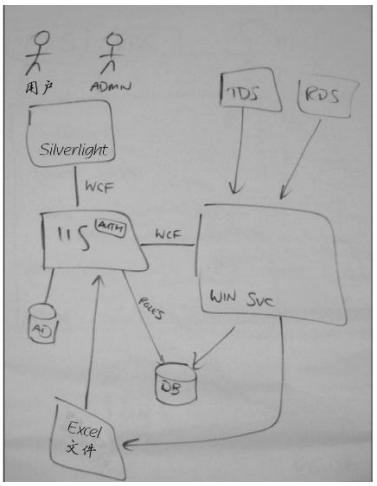
(1) 需要向 Silverlight 客户端暴露什么操作？

(2) 使用哪些技术捆绑和协议？

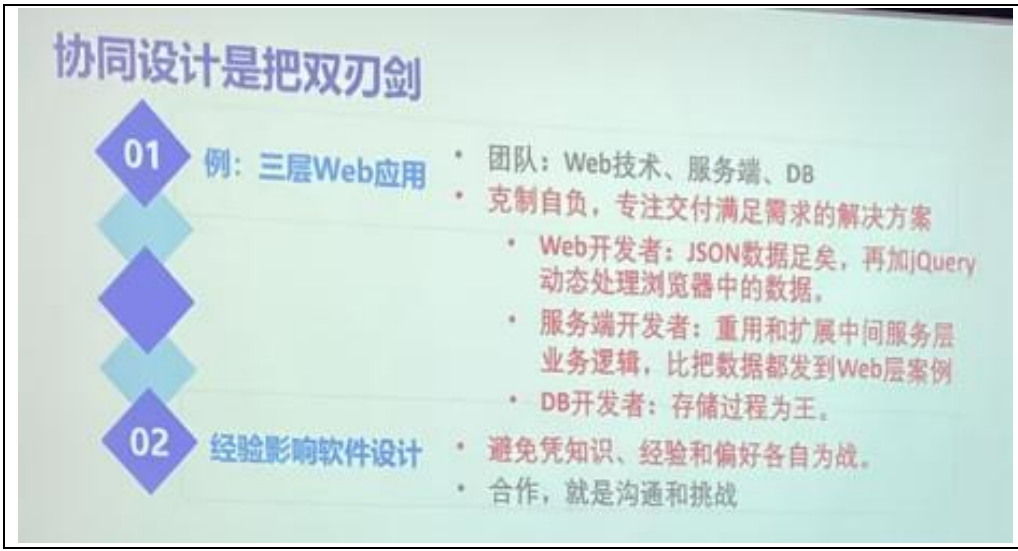
(3) 如何确保人们不能插入自己定制的 WCF 客户端并消费服务？

(4) 如何部署和测试？

(5) 时间和预算增加。



第 28 章 协同设计是一把双刃剑



第 29 章 软件架构是对话的平台

1、知识点

（1）敏捷方法：告诉我们，要定期与最终用户或他们的代表沟通，防止跑偏，才能确保我们构建的软件符合他们的需要。

（2）软件开发不只是交付特性。

软件的使用者（即客户）只是利益相关者的一类。通常还有很多其他的类型。

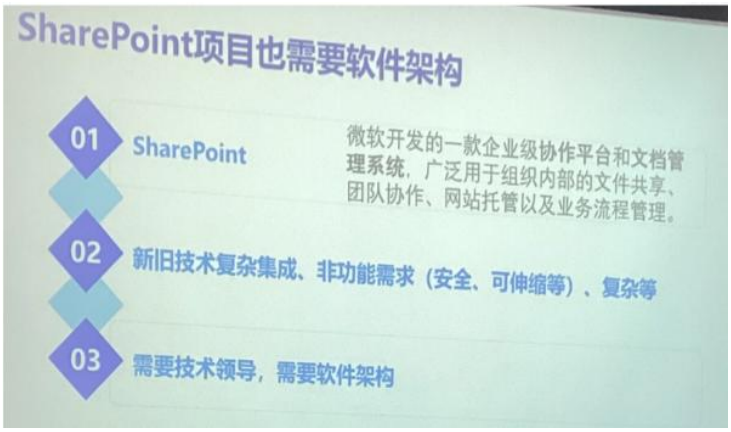
2、和架构有利害关系的利益相关者有哪些？

- ❑ 软件的使用者（即客户）
- ❑ 当前的开发团队：驱动力解决方案才会与架构一致。
- ❑ 未来的开发团队：明白解决方案如何运作，才能以一致的方式修改它。
- ❑ 其他团队：你的软件往往需要和环境中的其他系统集成，达成共识。
- ❑ 数据库管理员
- ❑ 执行/支持人员
- ❑ 安全团队

第 30 章 SharePoint 项目也需要软件架构

1、知识点

（1）SharePoint 项目，类似于钉钉、飞书，主要进行内部协作。



第 31 章 课后问题

1、什么是非功能性需求，它为什么重要？什么时候应该考虑非功能需求？（可能 5-6 分）

非功能性需求是指软件产品为满足用户业务需求而必须具有且除功能需求以外的特性。非功能性需求不直接与系统的具体功能相关，而是与系统的总体特性相关，如可靠性、响应时间、存储空间等。它定义了对于系统功能的服务或功能的约束，包括时间约束、空间约束、开发过程约束及应遵循的标准等。这些约束来源于用户的限制，包括预算的约束、机构政策、与其他软硬件系统间的互操作，以及如安全规章、隐私权保护的立法等外部因素。

非功能性需求对于软件产品的重要性不言而喻。以下是几个关键原因：

（1）影响用户体验：非功能性需求如响应时间、易用性等直接影响用户对软件产品的使用体验。如果软件响应时间过长或操作过于复杂，用户可能会对产品失去耐心和信心。

（2）关系软件稳定与扩展：非功能性需求如可靠性、可扩展性等对于软件的长期稳定运行和未来的扩展至关重要。如果软件在这些方面存在问题，可能会导致频繁的系统故障或无法满足未来业务增长的需求。

（3）增加开发风险：如果在软件开发后期或软件交付后才考虑非功能性需求，往往需要进行颠覆性的改动，这不仅会增加开发成本，还可能引入新的错误和风险。

综上所述，非功能性需求是软件产品中不可或缺的一部分，它对于软件产品的用户体验、稳定性和扩展性等方面都具有重要影响。因此，在软件开发的最初阶段就需要充分考虑非功能性需求，并将其纳入软件架构的设计之中。

非功能性需求应该在软件开发的最初阶段，即设计软件架构的时候就纳入考量。具体来说，在需求定义阶段，就需要确认各个非功能性需求的被需求程度；在开发阶段，则需要从软件架构的设计阶段开始就将非功能性需求纳入结构之中。这样做可以确保软件在满足功能需求的同时，也具备良好的非功能性特性，从而提高软件的整体质量和用户体验。

Part4 可视化软件

第 32 章 沟通障碍

1、知识点

（1）敏捷团队：有一个故事墙或看板，可视化了将要开始的、进行中的和已完成的工作。（**高效表达与沟通**）

（2）可视化软件开发流程是一个引入透明的奇妙方式，因为任何人都能从一个较高层次一眼看清当前的进度。

（3）抛弃 UML（统一建模语言）：一种用来表达软件系统设计的正规的标准化符号。

例子：**Rational Unified Process 统一过程（RUP）和结构化系统分析与设计方法（SSADM）（强调：架构驱动保持整体一致性）**

（4）抛弃 UML 没什么问题，但在敏捷比赛中，很多软件开发团队都**失去了视觉化的沟通能力。（没有统一标准）**

（5）敏捷需要良好的沟通。

第 33 章 对草图的需要

1、知识点

（1）软件架构的名声已经很差了，一说“大局”往往会让人想起**分析瘫痪和一堆很少有人真正理解的 UML 图。**

（2）测试驱动开发与图表：

• **测试驱动开发（TDD）**，即先做测试，有很多人使用 TDD 作为设计软件的方式，但它不见得适合每个人。无论布道者说什么，TDD 都不是**银弹（sliver bullet）**。

•Red-Green-Refactor:是测试驱动开发（TDD）中的核心概念，也被称为“红-绿-重构”循环。以下是关于 Red-Green-Refactor 的详细解释：

Red（红色）:在编写实际代码之前，先编写一个测试用例，并确保它失败。这个失败的测试用例揭示了所需功能的缺陷。可以明确当前代码存在的问题，并为后续的开发工作提供明确的方向。

Green（绿色）:编写足够的代码来使测试用例通过，从而验证代码的正确性。在这个阶段，把缺失补上，需要确保所有测试都通过，包括之前编写的测试。

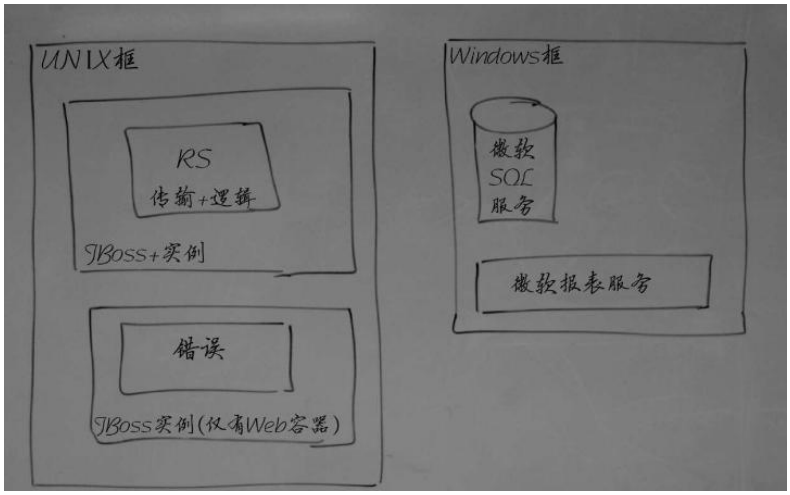
Refactor（重构）:在测试通过的基础上，对代码进行优化和重构，通过重构代码，可以进一步提高代码的质量，降低维护成本，并为未来的开发工作提供良好的基础。

（3）对于软件架构草图，我的目标是确保大家理解高层次结构，而不是类的设计细节。

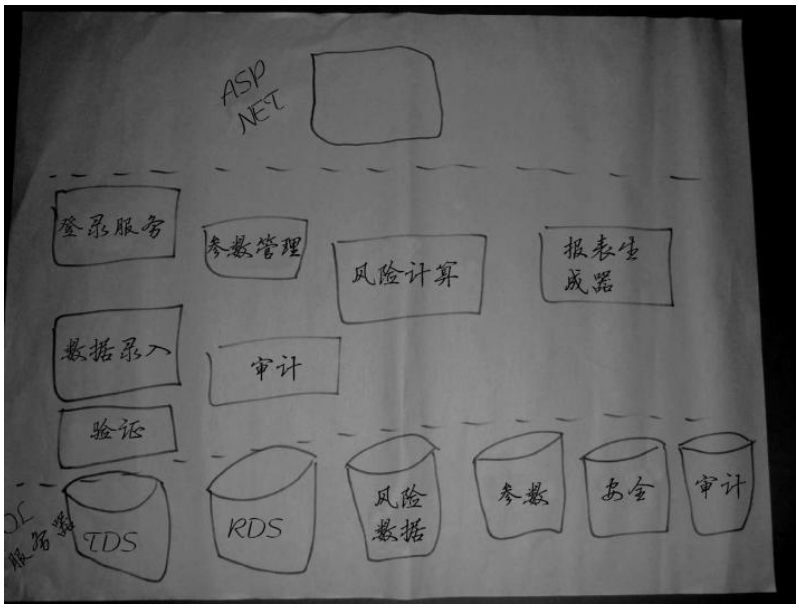
（4）画草图：要尽可能标准，否则会引起沟通障碍。**草图非 UML，而是一种地图，注重高层次机构，而非技术细节。**

第 34 章 无效的草图

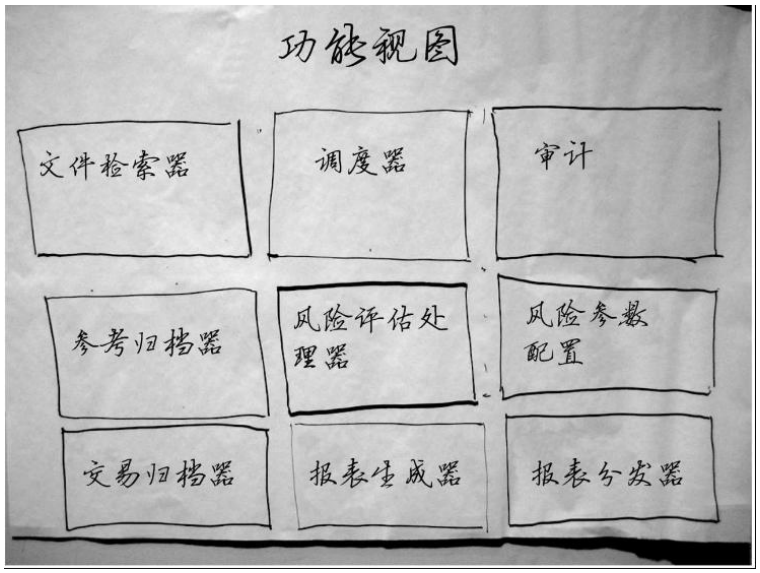
（1）**采购清单**：只是一个技术采购清单。我不知道那些产品做了些什么，**UNIX 框和 Windows 框**之间似乎也缺少某种联系



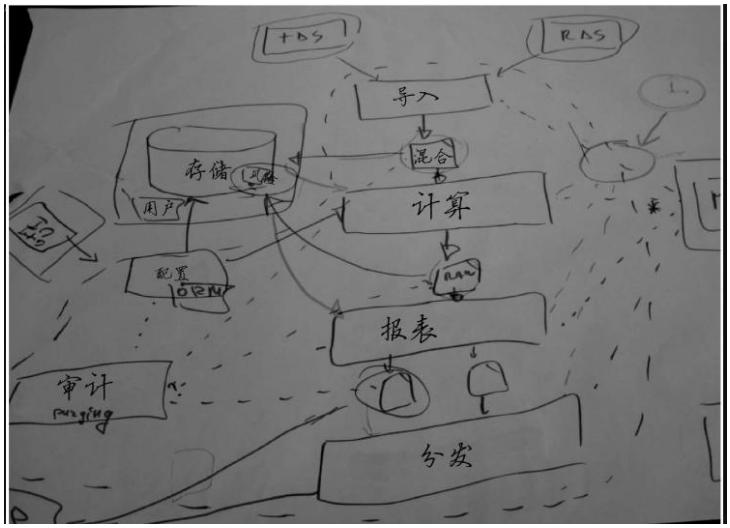
（2）只有框没有线



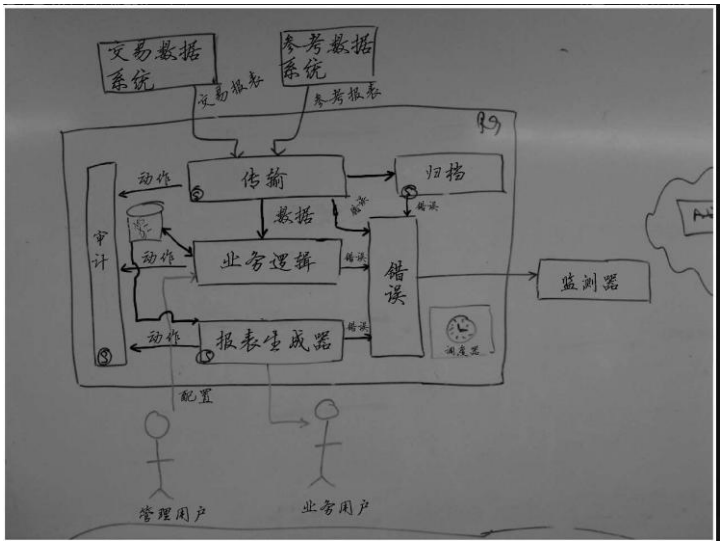
（3）“功能视图”



（4）航线图

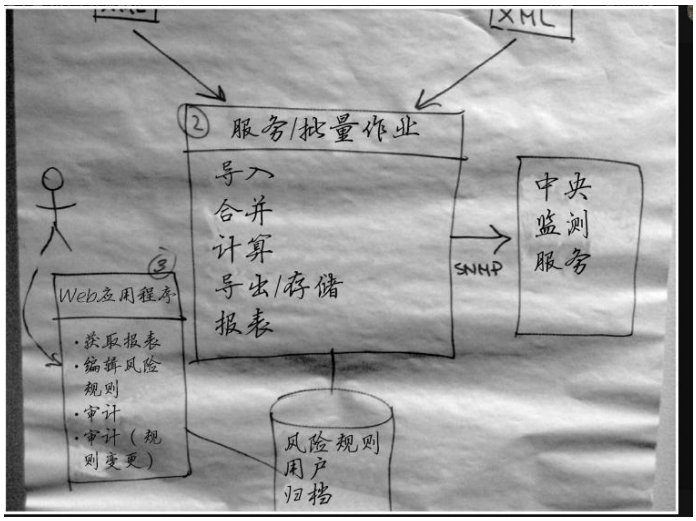


（5）一般正确，软件架构图

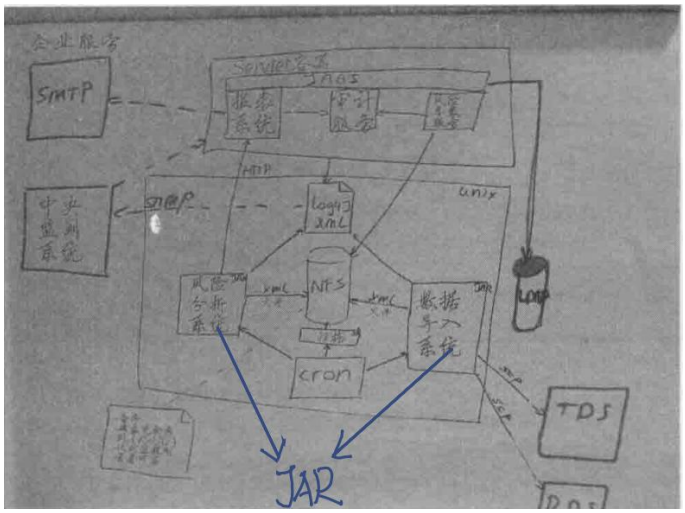


（6）推迟技术

它展示了软件架构的整体形态（包括职责），但把技术选择留给了你的想象力。



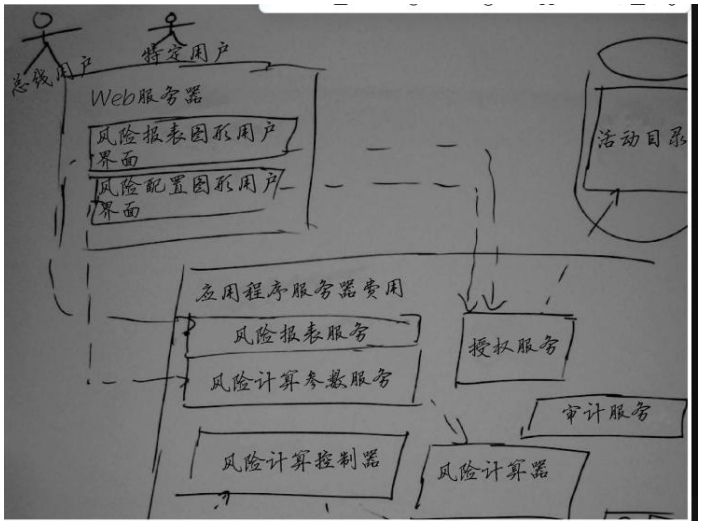
（7）部署和执行上下文（缺少信息）



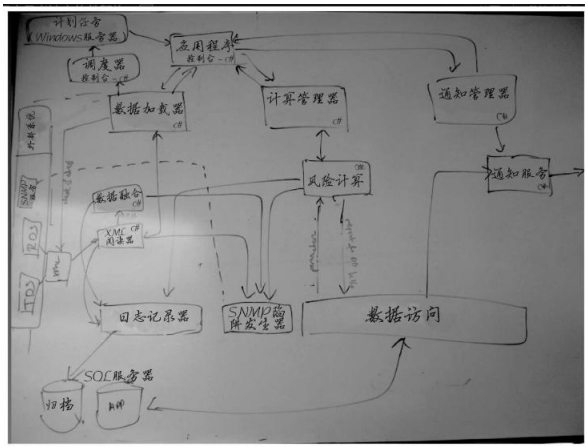
这两个框上注明了“JAR”，这是一个包含编译后 Java 字节码的压缩文件。

（8）太多假设

它**忽略**了一些重要的细节。对于**通信**是如何发生的，**Web 服务器**和**应用程序服务器**之间的连线没有提供任何信息。

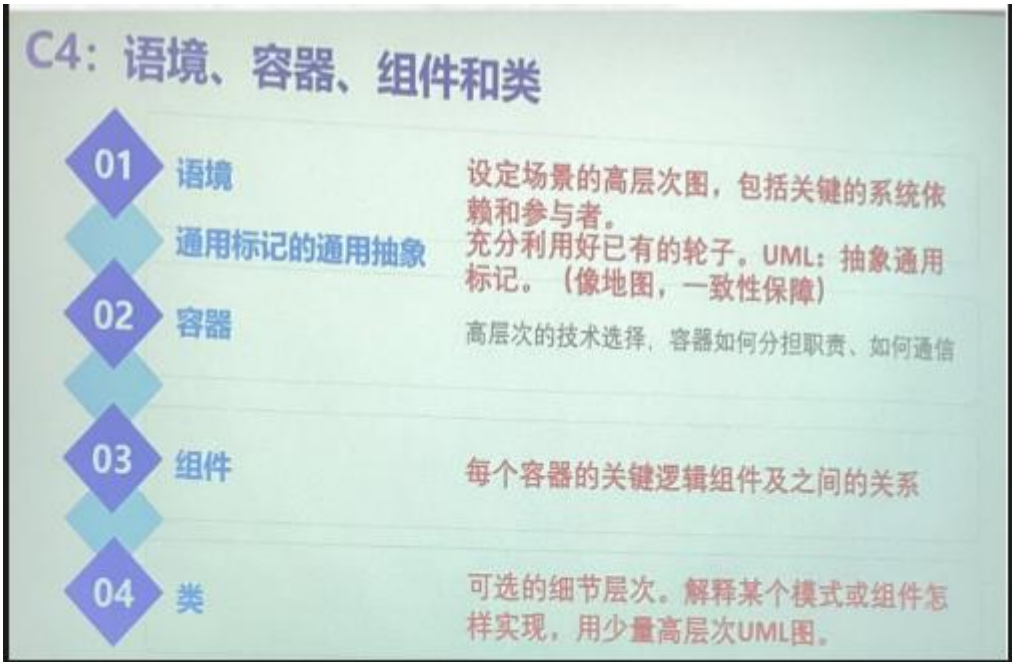


（9）无家可归的 C#对象（HOCO）



（10）应该用白板，白板是很有用的工具。

第 35 章 C4：语境、容器、组件和类



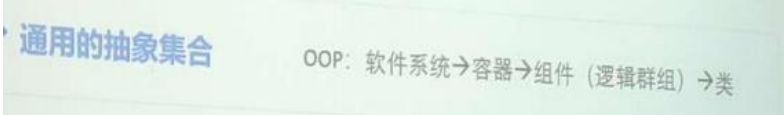
1、知识点

（1）架构作图工具：Microsoft Visio、Rational Software Architect 或 SparxEnterprise Architect，用于制作一些抽象层次各异的图。

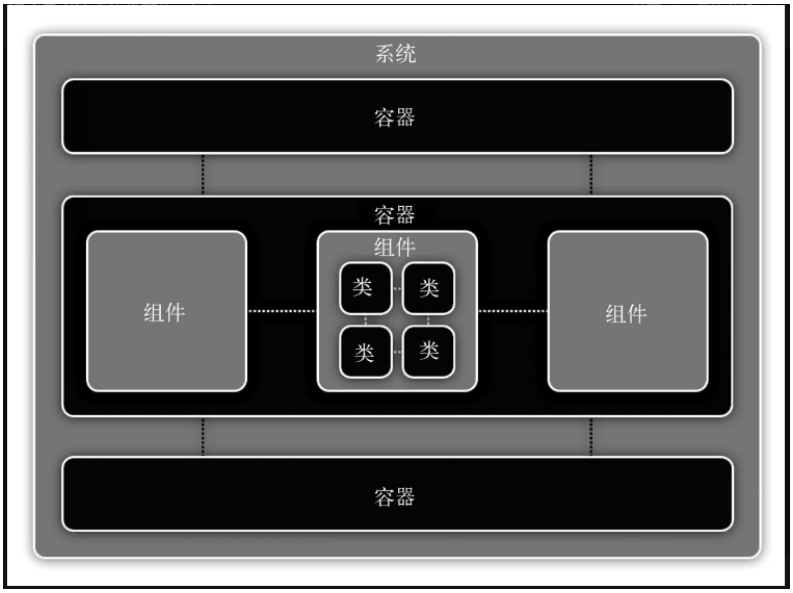
（2）通用的抽象集合：

软件系统由多个容器构成，容器又由多个组件构成，组件由一个或多个类实现。（OOP）

- **类：**对我们大多数人来说，在一个面向对象的世界里，类是软件系统的最小结构单元。
- **组件：**组件可以想象成一个或多个类组成的逻辑群组。
- **容器：**容器是指一个在其内部可以执行组件或驻留数据的东西。它可以是从网络或应用服务器直到富客户端应用或数据库的任何东西。作为整个系统的一部分，容器通常是可执行文件，但未必是各自独立的流程。
- **系统：**系统是最高的抽象层次，代表了能够提供价值的东西。一个系统由多个独立的容器构成，例如金融风险管理系统、网络网银行系统、网站等。



（3）架构结构的简单模型



(3) 总结软件的静态视图（C4）

我总结软件的静态视图时，大概会凭借脑海中的抽象集合，画出如下几类图。

- **语境** context：设定场景的高层次图，包括关键的系统依赖和参与者。
- **容器** container：高层次的技术选择，容器如何分担职责、如何通信。
- **组件** component：每个容器的关键逻辑组件及之间的关系。
- **类** class：可选的细节层次。解释某个模式或组件将（或已经）被怎样实现，少量高层次 UML 类的图。

(4) UML 提供了一套通用的抽象和用于描述它们的通用标记（像地图，一致性保障）

第 36 章 语境图

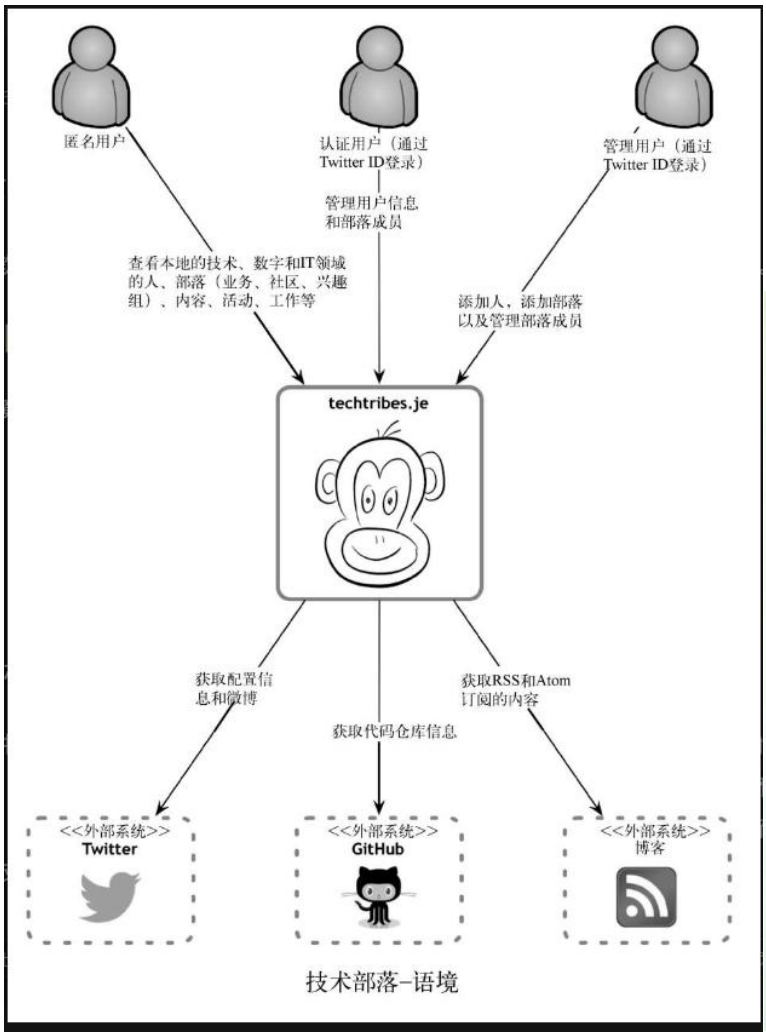
(1)意图

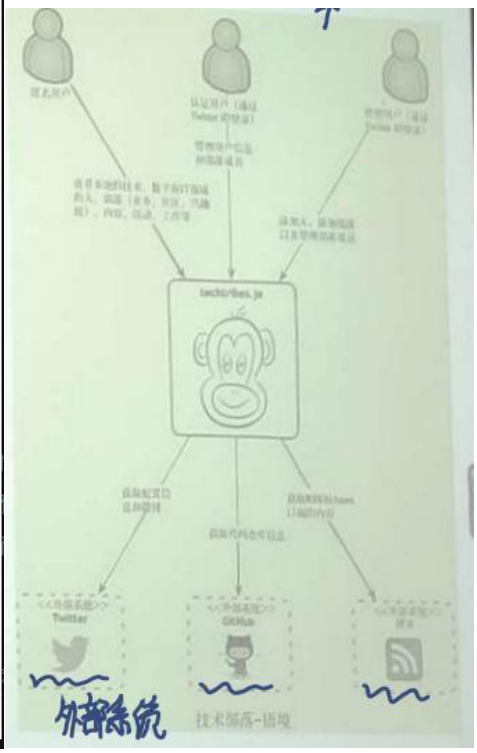
语境图能帮你回答下面这些问题：

- 我们构建的（或已经构建的）软件系统是什么？
- 谁会用它？（用户）
- 如何融入已有的 IT 环境？（和外部系统如何交流？）

(2)结构

在中间画一个简单的框图展示你的系统，周围是它的用户和其他与之相互作用的系统。





第 37 章 容器图

通过容器图说明高层次的技术选择。

(1)意图:

容器图可以帮助你回答下面的问题

- 软件**系统的整体形态**是什么样的?
- **高层次技术决策**有哪些?
- **职责**在系统中如何**分布**?
- **容器**之间如何**相互交流/交互**?
- 为了实现特性，作为一个开发者，我需要**在哪里写代码?**

（2）结构

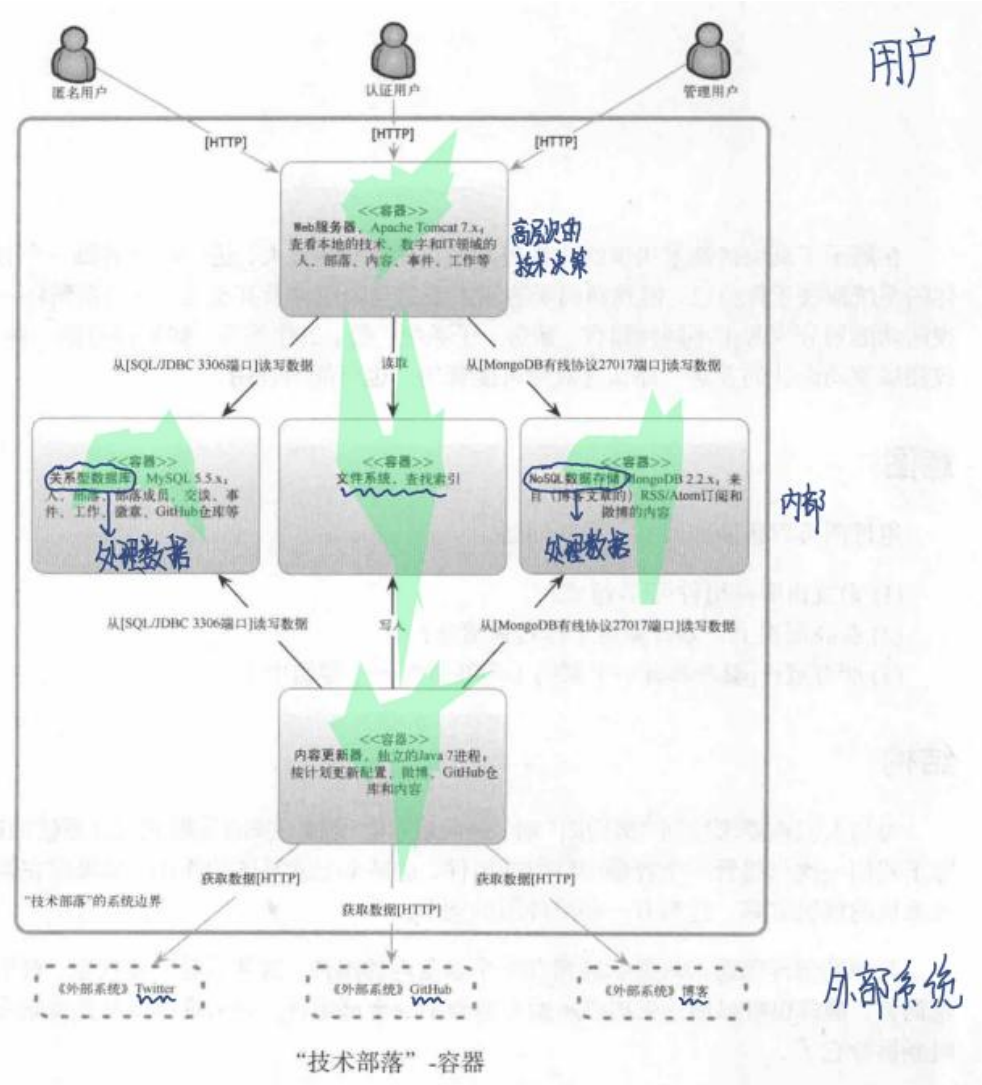
画一个简单的框图来展示你的关键技术选择

（3）容器

这里说的“容器”，指的是组成软件系统的逻辑上的可执行文件或过程，诸如:

- Web 服务器（比如 Apache HTTP 服务器、Apache Tomcat、微软 IIS、WEBrick 等);
- 应用服务器（如 IBM WebSphere、BEA/Oracle WebLogic、JBoss AS 等);
- 企业服务总线和业务流程编排引擎（如 Oracle Fusion 中间件等);
- SQL 数据库（如 Oracle、Sybase、微软 SQL 服务器、MySQL、PostgreSQL 等);
- NoSQL 数据库（如 MongoDB、CouchDB、RavenDB、Redis、Neo4j 等);
- 其他存储系统（如亚马逊 S3 等);

❑ 文件系统（特别是如果你在数据库以外读/写数据）；
❑ Windows 服务；
❑ 独立/控制台应用程序（即“public static void main”风格的应用程序）；
❑ Web 浏览器和插件；
❑ cron 和其他计划的工作容器。
图上的每一个容器都可以指定下面这些项。
❑ 名称：容器的逻辑名称（如“面向互联网的 Web 服务器”、“数据库”等）。
❑ 技术：容器的技术选择（如 Apache Tomcat 7、Oracle 11g 等）。
❑ 职责：容器职责的高层次声明或清单。你也可以展示一张驻留在每个容器中关键组件的小图，但我发现这通常会把图搞得很乱。
（4）示例： “技术部落”-容器



“技术部落”-容器

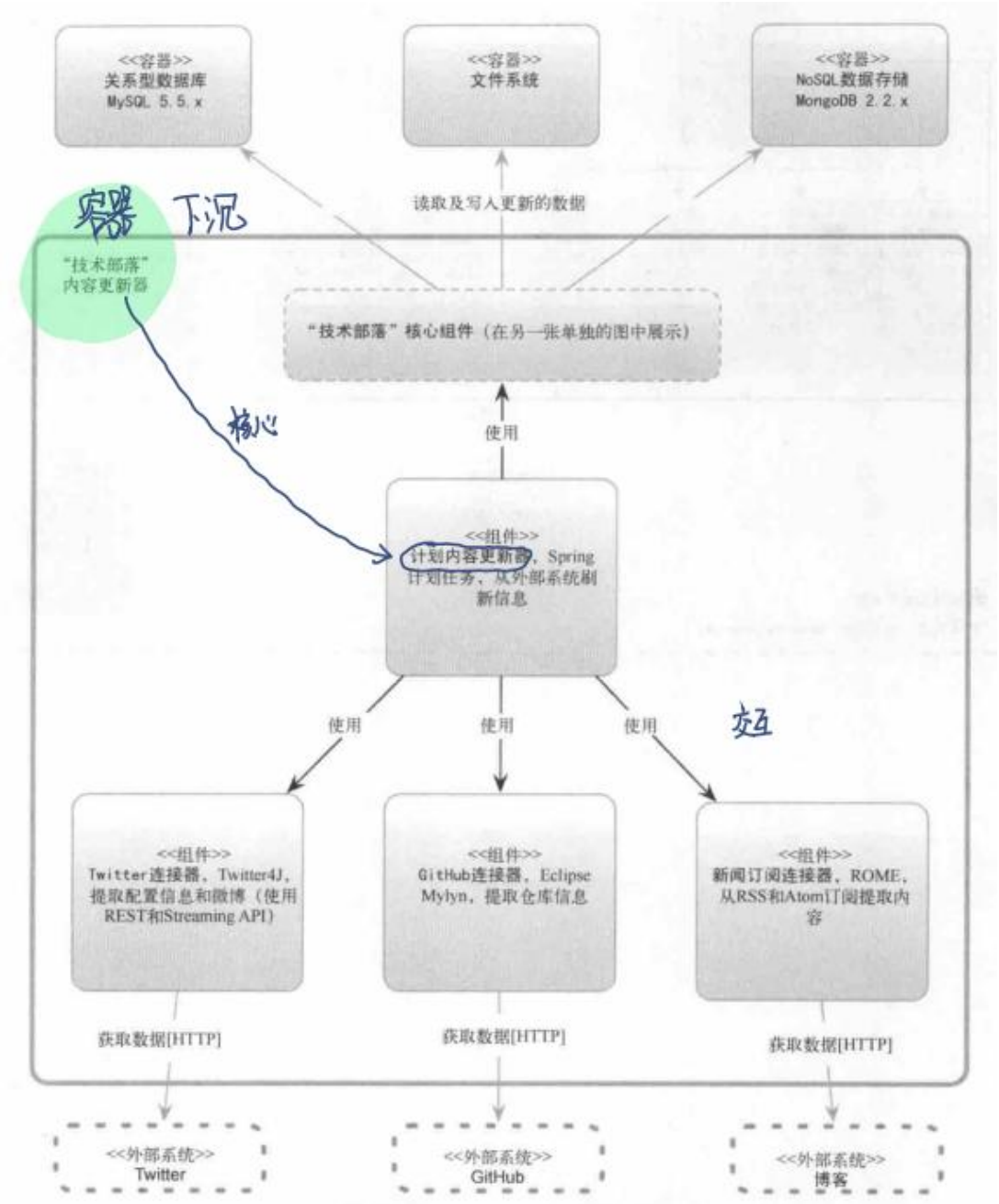
第 38 章 组件图

（1）意图
组件图可以帮助你回答下面的问题。
➢ **系统**由哪些**组件/服务**组成？
➢ **在高层次上，系统如何工作是否清晰**？
➢ **所有组件/服务都有一个家吗**（即驻留在一个**容器**中）？（组件/服务所在容器）

（2）结构

展示关键技术选择

（3）示例（“技术部落”-组件-内容更新器独立 Java 进程）



第 39 章 是否包含技术选择

（1）应有技术选择

（2）但不是实现细节

第 40 章 你会那样编码吗

（1）软件架构：既是逻辑上高层次的概念和表达，又是物理抽象。

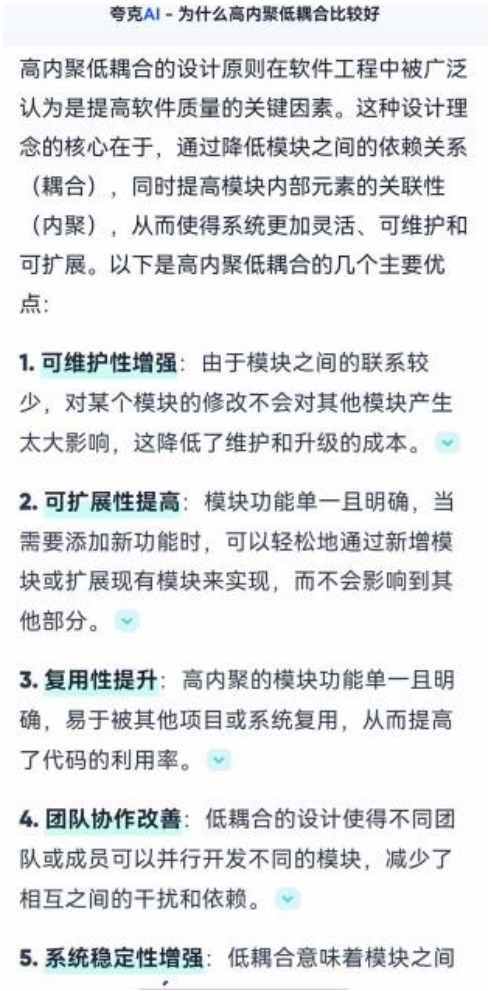
e.g.,Web server 和 App server 都用 log 组件，不能架构可视化时做在两者之外，因为编码时不合理。应标为共享组件。

（2）图应该反映现实

第 41 章 软件架构和编码

（1）职责驱动设计和组件分解：

- 深入组件、服务或模块的层次，但不是实现细节。
- 组件是接口背后的一组相关行为，可以用一个或多个协作类实现
- 好组件和好类共性：**高内聚、低耦合**，有良好定义的公共接口、良好封装。



（2）我们往往以组件为单位来思考。

（3）用层封装代码：很多软件团队用分层来结构化他们的代码。

（4）用特性封装代码

（5）用组件封装代码

（6）对齐软件架构和代码：从架构到代码有一个简单而明确的映射还有进一步的作用。结构、功能放一起保持平整性。

第 42 章 你不需要 UML 工具

1、不同类型的 UML 工具

(1) 只有图：Visio（UML 模板）、OmniGraffle

(2) 逆向工程：从代码创建 UML 图，但太过细节

(3) 往返工程：code->UML & UML 改变->code

(4) 模型驱动： 模型驱动架构（Model Driven Architect，MDA）。在图上用可执行 UML（xUML）或对象约束语言（OCL）标注出所需特性和行为。工具自动提供端到端的解决方案

2、知识点

（1）敏捷：简单、有效

技术含量不高的非 UML 方法有效、规约、一致、可追溯。（比如，箭头是依赖还是流向？）

（2）No Silver Bullet（即可以解决所有问题的技术）

第 43 章 有效的草图

1、知识点

（1）标题短而表意；图:有图号和图题，标签避免缩略语，legend

（2）形状：框和线；线条:样式(实虚点线、箭头)；职责:标注；颜色

（3）边框：明确意义(标签或解释)；布局:Visio；方向:上下左右

（4）Target：understanding

第 44 章 C4 的常见问题

1、C4 的常见问题

（1）语境图上标注短而表意的系统名称

（2）尽量避免混合的抽象层次(避免混合容器和组件图)

（3）共享组件画在每个组件图上并标注

（4）多系统的企业语境，由 C4 变 C5

注：架构到类（边界线），不能再往下降了。

Part5 为软件生成文档

第 46 章 代码不会讲述完整的故事——生成文档的原因

01 好代码

（1）TDD（测试驱动开发）包括：红灯/绿灯/重构阶段

（2）①重构方法要求变得更小和可复用以及自动化；

②根据个人偏好，选择注释/自注释

③易读理解和维护的好代码，却不是完整故事

（3）没有文档可能会有以下问题：

❑ 软件系统如何融入已有的系统形态；❑ 为什么会选择正在使用的技术；

❑ 软件系统的整体结构；❑ 各个组件在运行时部署在哪里，如何相互沟通；

❑ Web 层如何“知道”在哪里找到中间层；

❑ 日志/配置/错误的处理/其他采用了什么方法，在代码库中是否一致；

❑ 代码库中是否使用了通用的模式和原则；❑ 如何添加新功能，在哪里添加；

❑ 栈的安全性是如何实现的；❑ 如何实现可伸缩性；❑ 与其他系统的接口如何工作

02 好代码不能取代文档

（1）从文档反推问题答案的任务繁重。

（2）阅读代码的作用有限，向团队请教存在问题。

03 辅助信息与代码互补

（1）任何软件系统，在代码之上都有另一个可以回答这些类型以及更多问题的信息层。

这类信息与代码是互补的。

第 47 章 软件文档即指南（P131）

01 敏捷也需要文档

（1）不能走极端：没有文档或繁杂的文档

（2）有职责的划分

（3）敏捷也需要文档，但相比于 SDLC 较少。（系统开发生命周期 Systems development life cycle，SDLC）

02 文档即指南

（1）考虑把辅助文档当做一个不断变化的旅游指南。

（2）辅助文档应有：

- 图、表：导航作用（适当的放缩比例）
- C4、UML

03 文档应短小简洁

可能包含在软件指南中的事：

(1) 语境；(2) 功能性概览；(3) 质量属性；(4) 约束；(5) 原则；(6) 软件架构；(7) 外部接口；(8) 代码；(9) 数据；(10) 基础设施架构；(11) 部署；(12) 运营和支持；(13) 决策日志。

第 48 章 语境

01 意图，动机

语境部分应回答以下几类问题：

❑ 这个软件项目/产品/系统是关于什么的？

❑ 构建的是什么？

❑ 它如何融入现有环境？（比如，系统、业务流程等）

❑ 谁在使用？（用户、角色、参与者、人物等）

02 结构

简洁明了，不需要太长，1-2 页就够了。

03 受众

直接开发软件的团队内外的技术和非技术人员

04 是必须的

第 49 章 功能性概览

01 意图

（1）这部分让你总结系统的关键功能是什么。

（2）功能性概览应该回答以下几类问题：

• 系统实际上做什么是否清楚？
• 哪些重要特性，功能，用例，用户故事等对架构是重要的？原因是否清楚？
• 重要用户是谁？他们的的需求如何满足？
• 系统的主要流程和信息流是什么？

02 结构

（1）功能规格，用例文档甚至用户故事清单
（2）图表（UML 用例图等）很有帮助。
（3）流程图或 UML 活动图展示流程中较小的步骤。

03 受众

开发团队内外技术和非技术人员

04 是必须的

第 50 章 质量属性（非功能性需求）

01 意图

1、质量属性应回答以下问题：
• 对于架构必须满足的质量属性是否有清晰的认识？
• 质量属性是否满足 SMART 原则？
（specific 具体、measurable 可衡量、achievable 可达成、relevant 相关、time-bounded 及时）
• 如果通常理所当然的质量属性并无必要，是否会明确标示为超出范围（比如，“用户界面元素只用英语呈现”就表明并没有明确考虑多语言支持）？
或者说：显然的质量属性是否必要（如用户界面 UI 为英文）
• 有无不切实际的质量属性？

02 结构

（1）质量属性列表（第 22 章已出现过）
❑ 性能（比如延迟和吞吐）；❑ 可伸缩性（比如数据和流量）；❑ 可用性（比如运行时间、停机时间、定期维护、全天候、99.9%等）；❑ 安全性（比如认证、授权、数据保密性等）；❑ 可扩展性；❑ 灵活性；
❑ 审计；❑ 监测和管理；❑ 可依赖性；❑ 故障转移/灾难恢复的目标（比如手工还是自动化，要花多长时间）；❑ 业务连续性；❑ 互操作性；❑ 遵守法律法规（比如数据保护法）；
❑ 国际化（i18n）和本地化（l10n）；❑ 可访问性；❑ 易用性；❑ 等等。
（2）每一个质量属性都应该是精确的，不要让读者来解释。

03 受众

软件开发团队中的技术人员。

04 是必须的

第 51 章 约束

01 意图

（1）约束两面性，约束可减少设计工作
（2）约束通常是强加于你的，但不一定是“坏的”，因为减少可用的选项数目常常会让你的工作——设计软件——更容易。这一部分可以让你明确地总结工作中受到的约束，以及已经为你作出的决策。

02 结构

（1）约束列表
❑ 时间、预算和资源；❑ 允许使用的技术清单和技术约束；❑ 目标部署平台；❑ 已有系统和继承标准；
❑ 局部标准（比如开发、编码等）；❑ 公共标准（比如，HTTP、SOAP、XML、XML 结构、WSDL 等）；
❑ 标准协议；❑ 标准消息格式；❑ 软件开发团队的规模；❑ 软件开发团队的技能配置；
❑ 所构建软件的本质（比如战术或战略）；❑ 政治约束；❑ 内部知识产权的使用；❑ 等等。

03 受众

参与开发过程的每一个人

04 是必须的

第 52 章 原则

01 意图

这个部分的目的就是明确你要遵循的原则，可以是由利益相关者明确提出的要求，或者软件开发团队想要采用和遵循的原则。

02 结构

（1）原则的例子包括：

原则列表+简短注释或链接。

❑ 架构分层策略；❑ 视图中没有业务逻辑；❑ 视图中没有数据访问；❑ 接口的使用；❑ 始终使用 ORM；
❑ 依赖注入；❑ 好莱坞原则（不要给我们打电话，我们会给你打电话）；❑ 高内聚，低耦合；
❑ 遵循 SOLID（单一职责原则、开闭原则、里氏代换原则、接口隔离原则、依赖倒置原则）；
❑ DRY（don't repeat yourself，不要重复自己）；❑ 确保所有组件都是无状态的（比如，让伸缩更容易）；
❑ 选择一个富域模型；❑ 先择一个贫血域模型；❑ 始终选择存储过程；❑ 绝不使用存储过程；
❑ 不要重新发明轮子；❑ 错误处理、日志等的通用方法；❑ 购买而非构建；❑ 等等。

03 受众

软件开发团队中的技术人员。

04 是必须的

第 53 章 软件架构

01 意图

（1）这个部分的目的是总结你的软件系统的软件架构，这样就能回答以下问题：

- 宏观视图能看明白吗？
- 结构是否清晰？
- 展示了主要容器及技术选型，组件及交互吗？
- 关键的内部接口是哪些？

02 结构

- C4 中容器图和组件图作为这个部分的重点，附上关于内容的小段注释和容器/组件的总结。
- 展示组件交互的 UML 序列或协作图是描绘软件如何满足主要用例/用户故事/等的一个很有用的方法。

03 受众

软件开发团队中的技术人员。

04 是必须的

第 54 章 外部接口

01 外部接口

（1）这个部分的目的是回答下面几类问题。

❑ 关键的外部接口有哪些？

❑每个接口是否都经过技术审查？

- 接口的技术定义是什么？
- 如果使用了消息，哪些队列（点对点）和话题（发布-订阅）是用于通信的组件？
- 消息的格式是什么（比如，纯文本或 DTD/Schema 定义的 XML）？
- 同步还是异步？
- 异步消息的连接有保障吗？
- 如果必要，人们会长期订阅吗？
- 消息能否打乱顺序接收，这是一个问题吗？
- 接口是否幂等？

• 接口是否总是可用，或者比如说你是否需要在本地缓存数据？
• 性能/可伸缩性/安全性/其他是如何满足的？
❑每个接口是否都经过非技术审查？
•接口所有权属谁？
•接口多久会有变化，版本怎么处理？
•是否有服务级别的协议？

02 结构

接口列表和注释；包含一个强调接口的简化版容器或组件图。

03 受众

软件开发团队中的技术人员。

04 不是必须的

第 55 章 代码

01 意图

代码部分的目的是描述软件系统中重要、复杂、意义重大部分的实现细节，比如：

❑ 生成/渲染 HTML：对生成 HTML 的内部框架的简短描述，包括主要的类和概念。
❑ 数据绑定：根据 HTTP POST 请求更新业务对象的方法。
❑ 多页数据采集：简短描述构建跨网页表单的内部框架。
❑ Web MVC：正在使用的 Web MVC 框架的一个使用示例。
❑ 安全性：使用 Windows 身份基础（WIF）进行认证和授权的方法。
❑ 域模型：域模型重要部分的概览。
❑ 组件框架：简短描述为了在运行时重新配置组件而构建的框架。
❑ 配置：简短描述代码库中使用的标准组件配置机制。
❑ 架构分层：分层策略和用来实现的模式的概览。
❑ 异常和日志：总结在各个架构分层中处理异常和记录日志的方法。
❑ 模式和原则：解释模式和原则如何实现。
❑ 等等。

02 结构

•保持简单，可包含图表。
•高层次的 UML 类或序列图，内部定制框架工作原理。
•严控细节，保持在代码变化时图表依然有效。

03 受众

软件开发团队中的技术人员。

04 不是必须的

第 56 章 数据

01 意图

从数据文档审视，即数据部分的目的是记录任何从数据的角度来看重要的东西，回答下面几类问题：

❑ 数据模型看起来是什么样？❑ 数据存储在哪里？❑ 谁拥有数据？
❑ 数据需要多少存储空间？（比如，特别是如果你在处理“大数据”）
❑ 归档和备份策略是什么？❑ 业务数据的长期归档是否有法规要求？
❑ 日志文件和审计跟踪是否有类似的要求？❑ 是否用简单文件来存储？如果是，用的是哪种格式？

02 结构

•简单，让任何图都停留在较高的抽象层次。
•域模型（OOD+Domin-Driven Design/DDDD）或 E-R 图。

03 受众

软件开发团队中的技术人员以及其他可以协助部署、支持和运营软件系统的人。

04 不是必须的

第 57 章 基础设施架构

01 意图

这个部分描述软件将会部署到的物理/虚拟硬件和网络。这个部分的目的是回答下面几类问题：

- ❑ 是否有清晰的物理架构？
- ❑ 在所有的层中，什么硬件（虚拟或物理）做了这件事？
- ❑ 如果适用，它是否满足冗余、故障转移和灾难恢复？
- ❑ 选择的硬件组件如何改变大小和被选中是否清楚？
- ❑ 如果使用了多个服务器和网站，它们之间的网络联系是什么？
- ❑ 谁负责基础设施的支持和维护？
- ❑ 有照管通用基础架构（比如，数据库、消息总线、应用程序服务器、网络、路由器、交换机、负载均衡器、反向代理、互联网连接等）的中心团队吗？
- ❑ 谁拥有资源？
- ❑ 开发、测试、验收、试制、生产等是否有合适的环境？

02 结构

这个部分的主要关注点通常是展示各种硬件/网络组件以及如何相互融合的基础设施/网络图，配合简短的叙述。

03 受众

软件开发团队中的技术人员以及其他可以协助部署、支持和运营软件系统的人。

04 是必须的

第 58 章 部署

01 意图

这个部分是用来描述软件（比如容器）和基础设施之间的映射。这个部分回答下面几类问题：

- ❑ 软件安装和配置软件在哪里，怎么做？
- ❑ 软件如何部署到基础设施架构部分描述的基础设施元素上是否清楚？（比如，一对一映射、每个服务器多个容器等。）
- ❑ 如果这仍待决定，有哪些选项，是否做了文档？
- ❑ 内存和 CPU 在运行于单块基础设施上的进程间如何分配是否清楚？
- ❑ 有容器或组件以主动-主动、主动-被动、热备用、冷备用等形态运行吗？
- ❑ 部署和回滚策略是否已经定义？
- ❑ 软件或基础设施出现故障时会发生什么？
- ❑ 跨站点的数据如何复制是否清楚？

02 结构

有几种方式来组织这个部分的结构。

(1) 表格：展示软件容器和组件之间映射以及它们将被部署到的基础设施的简单文本表格。

(2) 图表：展示软件在哪里运行的 UML 部署图或者基础设施架构部分图的修改版。

03 受众

软件开发团队中的技术人员以及其他可以协助部署、支持和运营软件系统的人。

04 是必须的

第 59 章 运营和支持

01 意图

大多数系统都会受到所支持的运营需求的限制，特别是关于如何进行监测、管理和执行。这个部分应该处理下面几类问题：

❑ 软件如何为运营/支持团队提供监测和管理系统的能力是否清楚？
❑ 在架构的各个分层中这是如何实现的？
❑ 运营人员要如何诊断问题？
❑ 错误和信息记录在哪里？（比如，日志文件、Windows 事件日志、SNMP、JMX、WMI、自定义诊断等。）
❑ 更改配置是否需要重新启动？
❑ 有需要定期执行的手动管理任务吗？
❑ 旧数据需要定期归档吗？

02 结构

每个需求写一段简洁的说明，这个部分本质上是叙事性的，每个标题对应一组相关的信息。

03 受众

开发团队中的技术人员以及其他可以协助部署、支持和运营软件系统的人。

04 是必须的

第 60 章 决策日志

01 意图

这个部分的目的是简单记录所做的重要决策，包括技术选择（比如，产品、框架等）和整体架构（比如，软件的结构、架构风格、分解、模式等）

❑ 你为什么选择技术或框架 X，而不是 Y 和 Z？
❑ 你是怎么做的？产品评估还是概念证明？
❑ 你是否根据公司政策或企业架构战略而被迫做出关于 X 的决策？
❑ 你为什么选择所采用的软件架构？你考虑过其他哪些选项？
❑ 你怎么知道解决方案满足主要的非功能性需求？
❑ 等等。

02 结构

用一小段文字描述你要记录的每个决策，保持简单。

03 受众

软件开发团队中的技术人员以及其他可以协助部署、支持和运营软件系统的人。

04 不是必须的

Part6 开发生命周期中的软件架构

第 62 章 敏捷和架构的冲突：神话还是现实

1、敏捷和架构总体兼容

①架构就是结构和愿景，关键在于理解重要设计决策。
②敏捷强调兼容与快速迭代。

2、具体冲突

①团队结构：架构师 VS 少文档化
②流程和产出

大型预先设计目标是在蓝图（通常是一个计划）到位前，对全部事情达成共识；敏捷推崇“随机应变”浮现式设计”或“演化架构”

3、软件架构提供了 TDD、BDD、DDD、RDD 和代码整洁的分界线

了解：TDD、BDD、DDD 和 RDD 在软件开发和数据处理领域各自扮演着重要的角色，它们各自代表不同的开发方法或数据处理方式。它们分别重点关注测试、行为、领域模型和分布式数据处理，共同推动软件开发的高效性和软件产品的质量提升。

①有哪些架构的驱动力？

❑ 功能需求：需求驱动架构。不管怎么捕捉和记录需求（比如，用户故事、用例、需求规格书、验收测试

等），你都要大概知道你在构建什么。

❑ 质量属性：非功能需求（比如，性能、可扩展性、安全等）通常是技术方面的，也很难改造。理论上，这些都需要体现在初始的设计中，忽视这些属性会导致软件系统要么做得不够，要么做得太过。

❑ 约束：约束普遍存在于现实世界，包括批准的技术清单、规定的集成标准、目标部署环境、团队规模等。再说一次，不考虑这些会导致你交付的软件系统与环境不匹配，增加不必要的摩擦。

❑ 原则：是在试图为软件提供一致性和清晰度时你想要采用的东西。从设计的角度来看，这包括你的分解策略（比如，层、组件和微服务的对比）、关注点分离、架构模式等。明确概述一套初始的原则至关重要，这样构建软件的团队才会朝着同一方向出发。

4、知识点

TDD ： Test-driven_ development 测试驱动开发

BDD ： Behavior-driven_ development 行为驱动开发

DDD ： Domain-driven_ design 领域驱动设计

RDD ： Responsibility-driven_design 责任驱动设计

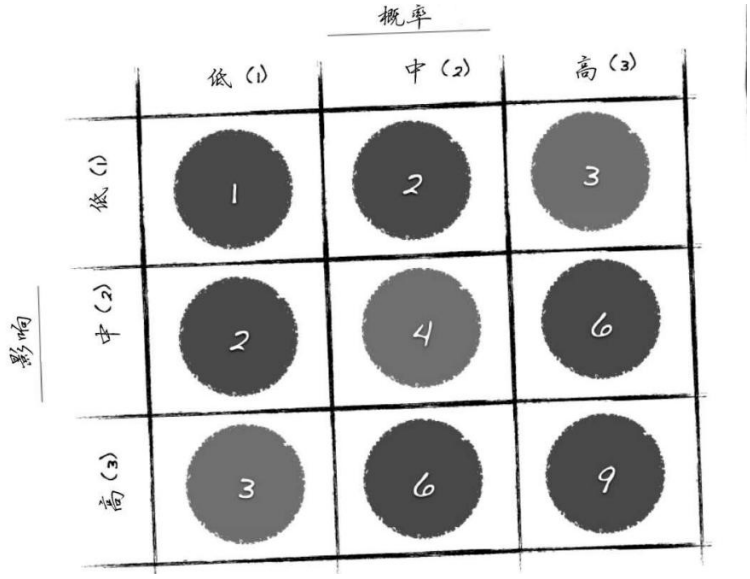
第 63 章 量化风险

1、作用

识别风险是恰如其分的预先设计的一个关键的部分。

2、风险发生的概率与影响

风险量化的概率/影响矩阵：概率和影响都可以量化为低、中、高或者就是一个数值。如果你把概率和影响分开考虑，就可以像下图一样，将矩阵中的两项分数相乘得到整体的评分。



3、设定风险的优先级

概率低影响小的风险可以被设定为低优先级风险。相反，概率高影响大的风险则需要设定为高优先级。

第 64 章 风险风暴

1、风险风暴是一种快速、有趣、协作和视觉的风险识别技术，整个团队都能参与。它有 4 个步骤。

步骤 1、画一些架构图

C4（语境、容器、组件、类）可用来标出架构中的不同风险。

步骤 2、分别识别风险

团队每人识别风险并量化（概率和影响）。

这里是一些要寻找的风险的例子：

❑ 第三方系统的数据格式意外变更；❑ 外部系统不可用；❑ 组件运行过慢；❑ 组件无法伸缩；

❑ 关键组件崩溃；❑ 单点故障；❑ 数据被破坏；❑ 基础设施故障；❑ 磁盘填满；

❑ 新技术未按预期工作；❑ 新技术使用过于复杂；❑ 等等。

步骤 3：汇总图中的风险

把风险小贴士置于正确的地方。

步骤 4：对风险设定优先级

贴量概率与影响，现在你们可以拿下每一张便利贴（或一堆便利贴），就如何量化已识别的风险达成共识。

2、缓解策略

根据风险的类型，有一些适用的缓解策略，包括下面这些：

(1) 教育：训练团队，重组团队，或者在你缺乏经验的领域（比如新的技术）招聘新成员。

(2) 原型：在需要通过证明某些事能否工作来缓解技术风险的地方创建原型。由于风险风暴是一种可视的技术，它可以让你很容易地看到软件系统中的可能应该结合原型更详细查看的部分。

(3) 修订：改变你的软件架构，以消除或减少已识别风险的概率/影响（比如，移除单点故障、增加一个缓存以免受到第三方系统中断的影响等）。如果你决定改变架构，可以重新进行风险风暴，以检验变化是否达到预期的效果。

（4）招聘

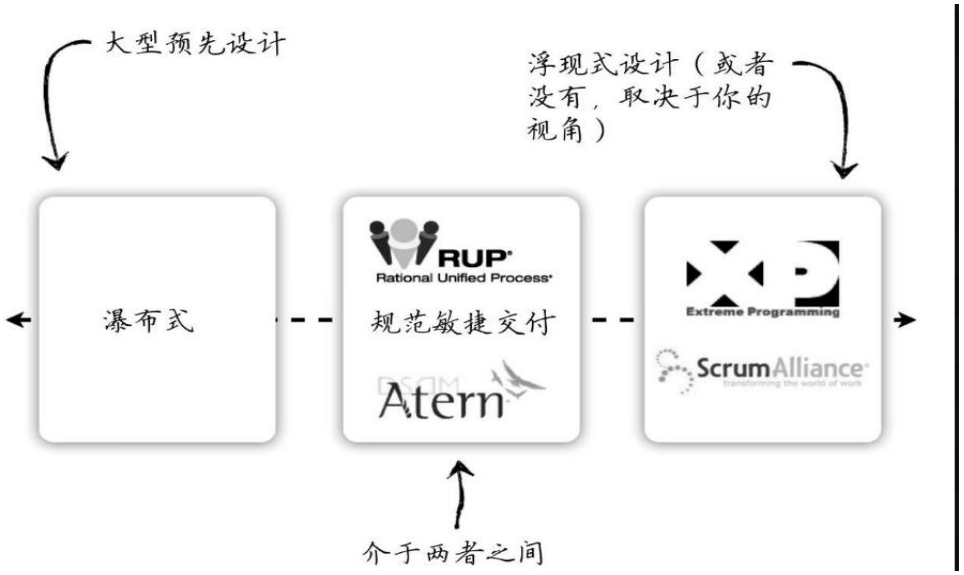
3、集体所有制

风险归谁所有？一个更好的方法是将技术风险的所有权交给软件架构角色。

第 65 章 恰如其分的预先设计

1、如果从提倡多少预先设计来对常见的软件开发方法进行比较，你会得到如下的图。一头是瀑布式，它的典型形式是大型预先设计，推崇在开始编写代码之前，每件事都必须决定、评审和签发。另一头则是表面看来回避架构的敏捷方法。

2、两头中间的是像 Rational 统一过程（RUP Rational_Unified_Process）、规范敏捷交付（DAD Disciplined_Agile_Delivery）和动态系统开发方法（DSDM ）Atern 这样的方法。这些是灵活的过程框架，实现中可以全部或部分采用。三者都是风险驱动的方法学，基本上就是“集中主要的高层次关键需求，规避风险，然后迭代和增量”。



3、恰如其分的设计

避免预先设计太少，避免预先设计太多。刚好够的预先设计为软件产品及其交付创造坚实的基础。

Part7 金融风险系统

第 68 章 金融风险系统

1、背景

交易数据系统（TDS　Transaction Data System）
交易数据系统存储了银行进行的所有交易。它已经配置为在纽约的交易关闭时间（下午 5 点）生成一个 XML 输出文件。输出包括银行进行的每一次交易的以下信息：❑ 交易 ID；❑ 日期；❑ 当前的美元交易价格；❑ 交易对手 ID。
参考数据系统（RDS　Reference Data System）
参考数据系统维护了银行需要的所有参考数据。这包括了交易对手的信息，每一个都代表一个个体、一家银行等。它也生成 XML 输出文件，包括了每个交易对手的基本信息。一个新的全组织参考数据系统将在未来 3 个月内完工，而当前的系统最终将停用。

2、功能需求

以下是新的风险系统的高层次功能需求：(1) 从交易数据系统导入交易数据；(2) 从参考数据系统导入交易对手数据；(3) 合并两个数据集，用交易对手的信息丰富交易数据；(4) 对每个交易对手计算银行面临的风险；(5) 生成一个可以导入微软 Excel 的报表，包含银行所有已知的交易对手的风险指数；(6) 在新加坡的下一个交易日开始（上午 9 点）之前将报表分发给业务用户；(7) 为业务用户子集提供一种配置和维护风险计算使用的外部参数的方法。