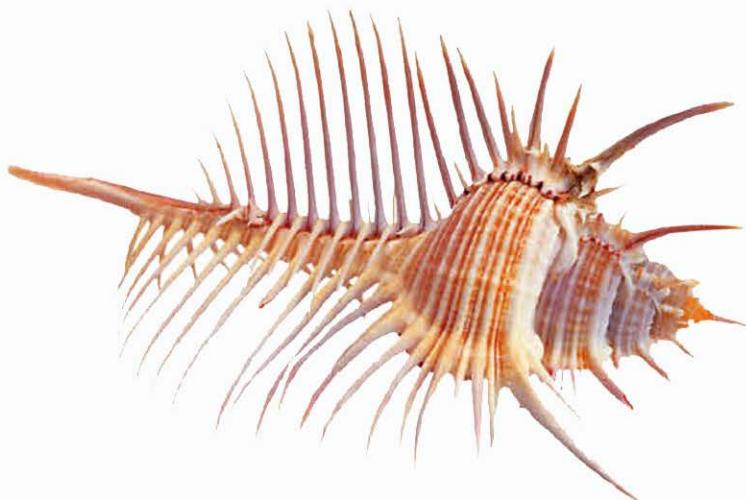


# 软件系统架构

## 使用视点和视角与利益相关者合作

(原书第2版)

(英) Nick Rozanski Eoin Woods 著  
侯伯薇 译



机械工业出版社  
China Machine Press

InfoQ 软件开发丛书

# 译 者 序

作为一名程序员，对未来的职业规划会有很多种。有人想要走管理路线，基本目标是成为项目主管或者项目经理；有人对数据库感兴趣，期望成为DBA；有人对测试感兴趣，期望成为测试方面的专家或者质量保证方面的高手；还有很多人希望做架构师，能够从总体上把握一个系统或者一个组织中的所有系统。

和其他角色相比，在一个系统的设计和开发过程中，架构师的地位显得尤为重要，颇有“运筹帷幄之中，决胜于千里之外”的架势。但是，很多时候，在很多组织中，架构师只不过是一个职位的概念，很多人即便在拥有这个头衔之后，还是会有很多的疑问亟待解答，比如：

- 架构师需要哪些能力？
  - 架构师需要从事哪些工作？
  - 想要设计出良好的架构，需要考虑哪些问题？
  - 如何与系统的各种利益相关者相互协调？他们的关注点一般又会有哪些？
  - 如何编写出能够让相关人员都很好了解的架构文档？
  - 如何评估设计出的架构？
  - 在解决某些特定系统问题时，应该采用何种合适的方法？
- .....

很少有一本书能够系统地回答这么多问题，能够为迷茫的架构师做出有效的指导。我自己也是一样，作为一名程序员，对架构方面还是如履薄冰，了解了一些皮毛，绝不敢以架构师来自居。

直到偶然与这样的一本书相遇，我才发现，其实想要成为架构师，或者在成为架构师之后，想要更好地完成相应的工作，这本书都能够提供很好的指导和帮助。其中提出的各种视图、视点和视角，可以帮助我们更好地了解架构的诸多方面，从而在设计架构的时候，能够根据最重要的利益相关者的关注点，更好地进行平衡。

也正因为如此，我才争取到这样的机会，把这么好的一本书翻译成中文，呈献给国内的程序员朋友们，希望他们能够通过阅读这本书，更好地了解架构设计和架构师这种职业，也更好地为他们解决架构方面的问题。

不得不说，翻译这样一本大部头的確是很辛苦的事情，这也是我所翻译的最厚的一本书，所以首先要感谢我的家人对我的理解和支持，让我抽出晚上和周末的时间来从事该书的翻译工作。还要感谢关敏老师，容忍我一再延期交稿。还要感谢张勇等人的参与，正是大家的帮助，才让我最终完成了这本书的翻译。

愿这本书能够帮助更多有志成为架构师或者已经是架构师的朋友们！

# 第一部分

## 架构的基本原则

第2章 软件架构概念

第3章 视点和视图

第4章 架构视角

第5章 软件架构师的角色

## 第②章

# 软件架构概念

当我们谈及软件系统的架构时，一个重要问题就是，术语都是从其他领域（像建筑架构或者船舶架构）借用过来的，并已经在各种情况下广泛但不一致地使用。例如，“架构”这个术语可以指的是微处理器的内部结构、计算机的内部结构、网络的组织情况、软件程序的结构等，不一而足。

本章会定义和回顾一些核心概念，它们会支持本书其余部分的讨论。这些概念包括：软件架构（software architecture）、架构元素（architectural element）、利益相关者（stakeholder）以及架构描述（architectural description）。

## 2.1 软件架构

现代社会计算机随处可见，不仅仅是在数据中心或者办公桌面上，甚至在汽车、洗衣机、手机和信用卡里面都能够见到。不论它们是大是小，是简单还是复杂，所有计算机系统都由三个基本部分构成：软件（像程序或类库）；数据，可能是临时的（在内存中），也可能是持久存在的（在磁盘或 ROM 中）；硬件（像处理器、内存、磁盘、网卡等）。

**定义** 当提及计算机系统时，我们指的是你需要指定和（或）设计的软件元素，从而满足特定的一系列需求，以及你需要在其中运行那些软件的硬件。

当你试图理解一个系统的时候，你会对很多方面感兴趣，包括每个部分的实际工作，各个部分如何协同工作，以及它们如何与周围的环境——换言之，它的架构——交互。我们可以在最新的国际标准 ISO/IEC 42010 “系统和软件工程——架构描述”（Systems and Software Engineering—Architecture Description）[ISO11] 中找到对软件架构广泛接受的定义。

**定义** 系统的架构是一系列基本概念或者系统在其环境中表现出来的属性，体现在它的元素、关系以及设计和发展的原则中。

让我们来详细看一下这个定义中的三个关键部分，也就是系统的元素和关系、它的基本属性以及它的设计和发展的原则。

### 2.1.1 系统元素和关系

任何一个系统都是由很多部分组成的，这些部分可以称为模块、组件、部件或者子系统。我

们会尽量避免使用这些术语，因为它们都包含了特定类型的实现或者部署技术。我们更喜欢遵循 ISO 标准和大量其他标准，使用大家不太熟悉但是在语义上中立的术语元素，来表示组成系统的部分。我们会在本章稍后更正式地定义架构元素这个术语，但现在让我们先约定，元素就是系统在架构上的重要组成部分。

组成系统的元素以及它们之间的关系定义了包含它们的系统结构。软件架构师感兴趣的结构类型有两种：静态结构（设计时元素的组织形式）和动态结构（运行时元素的组织形式）。

1) 系统的静态结构展示了系统的设计时形式，也就是它的元素有哪些，这些元素是如何组合从而提供系统所需要的特性的。

**定义** 系统的静态结构定义了它的内部设计时元素及其排列方式。

内部设计时（internal design-time）软件元素可能是程序、面向对象的类或包、数据库存储过程、服务或者任何一种自包含的代码单元。内部数据元素包括类、关系型数据库实体或表以及数据文件。内部硬件元素包括计算机或者它的组成部分（如磁盘或 CPU 等）以及网络元素（如线缆、路由器或集线器等）。

基于所处的情境，这些元素的静态排列方式定义了这些元素之间的关联、关系或者连接性。例如，对于软件模块来说，静态关系可能包括元素的层级关系（A 模块是基于 B 模块和 C 模块构建的），或者元素之间的依赖关系（A 模块依赖于 B 模块的服务）。对于类、关系型实体或者其他数据元素来说，关系会定义一种数据项目是如何与另一种链接的。对于硬件来说，关系定义的是系统的各种硬件元素之间所需要的物理连接。

2) 系统的动态结构表示的是系统实际上如何工作，也就是说，在运行时的情况以及系统对外部（或者内部）的刺激会做出什么样的响应。

**定义** 系统的动态结构定义了它的运行时元素及其之间的交互。

这些内部交互可能是元素之间的信息流（元素 A 向元素 B 发送消息），或者内部任务以并行或串行的方式执行（元素 X 调用元素 Y 上的例程），或者根据它们对数据造成的影响（创建数据项 D，进行多次更新，最终销毁）来说明。

当然，系统的静态结构和动态结构彼此紧密相连。例如，没有程序或数据库之类的静态结构元素，也就不会有任何动态结构元素可以让信息在之间传递。然而，这两种类型的结构并不相同。例如简单的客户端 / 服务器系统，其中有一个面向客户端的元素，它会处理所有与用户之间的交互。这会作为静态结构元素出现一次，但是它会在动态结构模型中多次出现（对每个活动用户都会出现一次）。动态结构模型还需要说明是什么让客户端元素的实例的状态变为活动或者停用的（例如，用户登录之后又注销）。

## 2.1.2 基本系统属性

系统的基本属性表现为两种不同的方式：外部可见行为（系统所做的工作）和质量属性（系统如何完成工作）。

1) 外部可见行为从外部观察者的角度告诉你系统做了哪些工作。

**定义** 系统的外部可见行为定义了系统和环境之间的功能交互。

这些外部交互与我们考虑动态结构时的情况类似。其中包括输入系统和从系统输出的信息，系统对外部刺激做出响应的方式，以及架构向外部环境发布的“契约”(contract)和API。

我们可以把系统认为是黑盒来对外部行为建立模型，这样你对其内部一无所知（如果你向按照架构构建的系统发送请求P，那么系统就会返回响应Q）。或者，我们可以认为这是内部系统状态响应外部刺激所做出的改变（提交请求R导致创建了内部数据项D）。

2) 质量属性从外部观察者的角度告诉你系统的行为方式（经常指的是非功能性特征）。

**定义** 质量属性是系统外部可见的、非功能性的属性，例如性能、安全性或者可伸缩性等。

你可能会对很多质量属性感兴趣：系统在负载下性能如何？对于特定的硬件峰值吞吐量有多少？对于恶意用户系统如何保护信息的安全性？宕机的频率有多高？管理、维护和改善的难度有多大？对于残障人士易用性如何？这些特征哪些有意义取决于你的情况，以及利益相关者的关注点和优先级。

### 2.1.3 设计和发展的原则

对于一个结构良好、可维护的系统来说，有一种情况显而易见，那就是它的实现是一致的，并且遵循了系统范围内的一系列结构规定。这让系统更易于理解，并支持以一致和逻辑的方式对系统进行扩展，满足系统总体的形式，而不会引入不必要的复杂性。

想要达到这种内部实现的一致性，有一件事非常必要，那就是要有一系列清晰的原则来指引系统的设计和发展。

在《牛津英文词典》中，对原则的一般定义是：基本的事实或定义，可以作为看法和行为的基础。在架构设计的情境中，我们对这个定义稍加扩展，把架构原则定义为：对指引架构定义的看法、方法或者意图的基本声明。

对于为一致、架构良好的架构确立决定性框架来说，定义并遵循架构原则这种方式非常有用。原则会暴露出底层的假设，并对其进行冷静考虑，换言之，它们让隐藏的内容显露出来。它们是开始架构项目很好的方式，特别是在动机或范围还没有确定的时候。如果你怀疑在提出的架构需求中有重大但未发现的冲突和矛盾，这也会非常有用。我们会在第8章中详细讨论设计原则。

### 2.1.4 系统属性和内部组织形式

让我们通过下面的简单示例来探究一下系统属性的想法，以及它是如何与系统的内部组织形式相关联的。

**示例** 一个航空公司订票系统支持多种不同事务，包括订座、更新或取消座位、转让座位、升级座位等。图2-1显示了这个系统的情境（在此我们使用了简化的用例图：矩形代表系统，“火柴人”代表与系统交互的用户，而通知框提供额外的支持信息。）

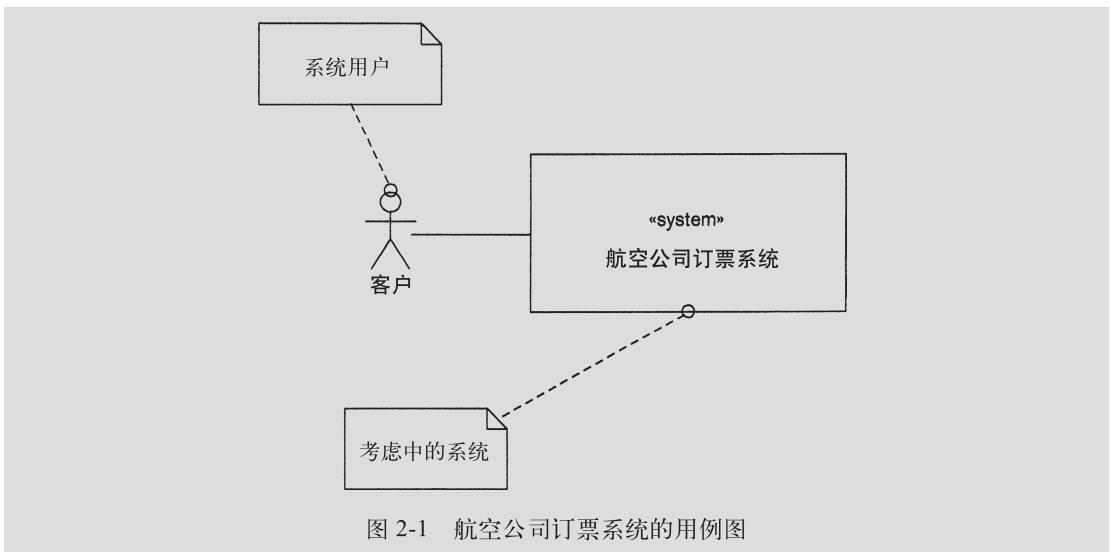


图 2-1 航空公司订票系统的用例图

系统的外部可见行为（所做的工作）是它对客户提交的事务所做出的响应，例如订座、升级服务或者取消订票等。系统的质量属性（如何工作）包括：在特定负载下一项事务的平均响应时间；系统能够支持的最大吞吐量；系统的可用性；修复缺陷所需要的时间、技能和成本。

面对这些需求，架构师可以用多种方式来设计系统。在接下来的内容中，我们会概述针对这个系统的两种架构方法。

架构师可以基于两层的客户端 / 服务器方法来为航空公司订票系统设计一种解决方案。（事实上，这是应用一种架构样式（architectural style）的例子，你会在第二部分看到详细的内容。）如图 2-2 所示，在这种方法中，大量客户端（向客户展示信息，并接受他们的输入）会通过广域网（WAN）与中央服务器（它会在关系型数据库中存储数据）通信。已经确定的架构样式（如客户端 / 服务器）的优缺点已

经是众所周知的，因此在开始时我们使用这种大家已经理解的方法，有助于避免向设计中引入不必要的风险。

正如图 2-2 所说明的，这种客户端 / 服务器架构的静态结构（设计时组织形式）由客户端程序（在这个示例中进一步分解为表现层、业务逻辑层、数据库层和网络层）、服务器和两者之间的连接组成。相关的架构图显示出，动态结构（运行时组织形式）基于请求 / 响应模型：请求是由客户端通过 WAN 向服务器提交的，

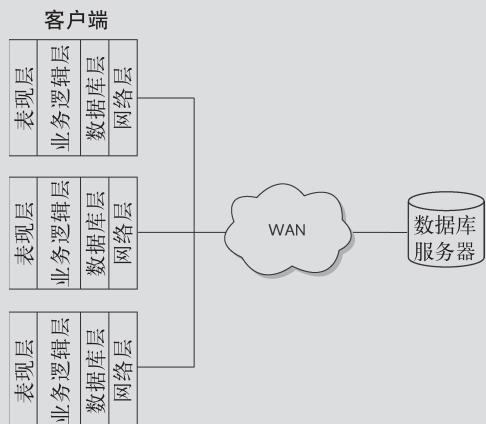


图 2-2 针对航空公司订票系统的两层客户端 / 服务器架构

而响应是由服务器返回给客户端的。架构的静态元素提供了一种机制，通过这种机制才能够发生动态交互（例如，客户端程序代表客户提交请求，然后接受并显示结果）。

另外，架构师还可以选择采用三层的客户端 / 服务器结构，其中客户端只执行表现处理，业务逻辑和数据库访问都在应用程序服务器中执行，如图 2-3 所示。

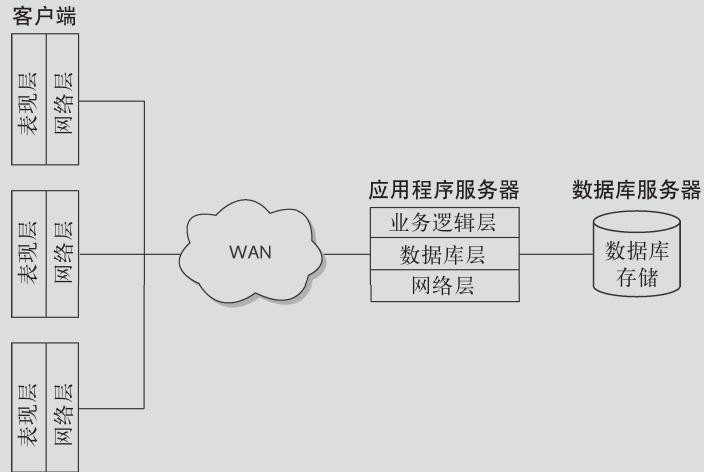


图 2-3 针对航空公司订票系统的三层客户端 / 服务器架构

这种架构的静态结构由客户端程序（在这个示例中进一步分解为表现层和网络层）、应用程序服务器（在此包括业务逻辑层、数据库层和网络层）、数据库服务器以及它们之间的连接组成。动态结构基于三层请求 / 响应模型：客户端会通过 WAN 向应用程序服务器提交请求，应用程序服务器会在需要的时候把请求提交给数据库服务器，然后响应会由应用程序服务器返回给客户端。

架构师可能会认为两层的架构更合适，因为这种架构的运维相对简单，能够由公司的软件开发者快速开发出来，交付所需的成本比其他方案更低，或者基于其他的各种原因。

架构师也可能认为三层架构才是正确的选择，因为在负载增加时它具有更好的可伸缩性，对客户端硬件能力要求更低，可能会提供更高的安全性，或者基于各种其他原因。

不管架构师认为哪种架构更为合适，之所以选择那种，都是因为它能够提供方法所承诺的系统属性与系统需求之间的最佳契合。

在这个示例中，对于问题有两种可能的解决方案，分别是基于两层和三层的方式。我们将其称为候选架构（candidate architecture）。

**定义** 系统的候选架构是静态结构和动态结构的一种特定组合，可能会表现出系统所需要的外部可见行为和质量属性。

尽管候选架构拥有不同的静态结构和动态结构，但每种架构都能够满足系统的总体需求，及

时、有效地处理航空公司的订票业务。然而，尽管我们相信所有候选架构都具有相同的外部可见行为（在这个案例中，对订票事务做出响应）和一般的质量属性（例如可接受的响应时间、吞吐量、可用性和修复时间），但是它们在表现出来的特定质量属性上还是有区别的（例如比另一种更容易维护但构建成本更高）。

不管是哪种情况，候选架构实际上表现出这些行为和属性的程度必须由对静态结构和动态结构的进一步分析决定。例如，两层的候选架构可能更好地符合功能需要，因为它支持功能更丰富的客户端；而三层的候选架构可能有更好的吞吐量和响应时间，因为各层之间的耦合更松散。

架构师的职责之一就是要获得每种候选架构的静态结构和动态结构，理解它们能够以何种程度表现所需要的行为和质量属性，并做出最佳选择。当然，“最佳”的意思总是不明确，我们会在第二部分再讨论这个问题。

我们能够捕获系统的外部可见属性和内部结构及组织形式之间的关系，如下所示：

- 系统的外部可见行为（它所做的事情）是由内部元素的组合功能行为决定的。
- 系统的质量属性（它如何工作）（如性能、可伸缩性以及弹性）都来自于内部元素的质量属性。（通常，系统的整体质量属性和表现取决于最差或者最弱的内部元素的属性情况。）

当然，事实上并非那么简单。例如，如果一台服务器无法扩展以处理提交过来的负载，它在功能上可能会受限（例如，用户可能无法登录，或者执行一些耗费大量资源的功能）。然而，我们还是发现这种非常单纯的区别很有用，它会表现出我们的想法。

### 2.1.5 软件架构的重要性

不论大小，所有计算机系统都是由彼此连接在一起的部分组成的。可能只有少量这样的部分，也可能只有一个，也可能有几十上百个。这种连接可能微不足道，也可能非常复杂，或者位于二者之间。

此外，所有系统都是由许多部分组成的，它们会以确定的（可预测的）方式彼此交互以及与外界交互。行为可能很简单且易于理解，也可能非常费解，没有人能够了解所有的方面。然而，这种行为就在那里，仍然（至少在理论上）是我们所需要的。

换言之，每个系统都拥有架构，就像每栋楼房、每座桥、每艘战舰都有架构一样——就像每个人的身体都有各种机能。

这个概念非常重要，我们在此要正式将其声明为一种原则。

**原则** 每个系统都有架构，不管是否有相关文档，也不管是否有人理解。

系统的架构是固有的、本质的属性，不管是否有相关文档，也不管是否有人理解。每个系统都只能拥有唯一的架构——尽管我们会看到，这个架构会以多种方式展现。

## 2.2 架构元素

正如之前说明的，我们会对架构元素做出标准定义，用来指代构建系统的部分。

**定义** 架构元素（简称为元素）是我们认为可以构建系统的基本组成部分。

架构元素的情况与你想要构建的系统类型以及考虑的元素所处的情境有很大关系。程序类库、子系统、可部署的软件单元（如 Enterprise Java Beans 或者 .NET 库）、可重用的软件产品（如数据库管理系统）、或者整个应用程序都可能是信息系统中架构元素的形式，这依赖于所构建的系统。

架构元素应该拥有以下关键属性：

- 一系列清晰定义的责任。
- 清晰定义的边界。
- 一系列清晰定义的接口，它会定义元素为其他架构元素所提供的服务。

我们经常会把架构元素非正式地称为组件或者模块，但是这些术语已经被广泛使用，并确定了特定的意义。组件（component）更多的是指使用编程级别的组件模型（如 J2EE 或 .NET），而模块（module）更多指的是一种编程语言的概念。尽管在某些情境下它们可能是有效的架构元素，但是在另一种情境下就可能不是那种基本的系统元素了。

正因为如此，我们从现在开始要尽量不使用这些术语。相反，贯穿本书会始终使用元素（element）这个术语以避免歧义（这遵循了其他理论，包括 ISO 42010 和 Bass、Clements 和 Kazman[BASS03]，请阅读 2.7 节以获得更多细节信息）。

## 2.3 利益相关者

传统软件开发是由交付软件以满足用户需求的需要驱动的。尽管对术语用户（user）有各种各样的定义，但是所有软件开发方法都以某种方式基于这种原则创建。

然而，软件系统所影响的人群并不仅限于使用它的人。软件系统并不仅仅是被使用：它会被构建并被测试，它需要被运维，它还可能需要修复，它经常会改善，当然它会被支付。上述每种活动都涉及很多（可能是大量）用户以外的人。每种人群都有他们自己的需求、兴趣和需要，需要软件系统来满足。

我们把这些人统称为利益相关者。理解利益相关者的角色是理解架构师在开发软件产品或系统中角色的基础。我们对利益相关者做出了如下定义。

**定义** 系统架构中的利益相关者是能够从实现系统获得利益的个人、团队、组织或者一类人。

这个定义基于 ISO 标准 42010 的定义，我们会在第二部分对其深入讨论。现在，让我们先看一下这个定义中的几个关键概念。

### 2.3.1 个人、团队或组织

首先，考虑一下“个人、团队或组织”这个短语。正如我们将会在这本书中看到的，与系统架构有利益关系的远远不只是开发者，甚至不只是开发者和用户。将架构实现为系统这个过程的

影响范围非常广，包括那些为其提供支持、部署系统或者为其付钱的人。

指定架构是利益相关者决定系统情况和方向的关键机会。然而，你会发现有些利益相关者只关心自己的角色，基于各种原因不参与架构相关的工作。因此，作为架构师，你的部分工作就是要管理和激励他们，让他们重视参与的机会，并获得他们对任务的意见。

正如定义所提到的，利益相关者通常会代表一类人，如用户或者开发者，而不会代表特定的某个人。这就存在问题，因为我们可能在给定的时间中无法获得这类人中所有成员（所有用户、所有开发者）的需求，并达成一致。另外，你周围可能不会有利益相关者（比方说，当开发新产品的时候）。不管在哪种情况下，你都需要选择一些有代表性的利益相关者来做一组人的代言。我们会在第二部分再讨论这个问题。

### 2.3.2 兴趣和关注点

现在思考一下“从实现系统获得利益”这句话。这个标准的确是很宽泛，而对它的解释完全是由项目而异。当你开始创建架构的时候，会发现你处理的是发现的过程，而不是捕获的过程，换言之，在这个系统开发生命周期的早期过程中，你的利益相关者很可能还无法精确地描述出他们的需求是什么。

有时我们解释这个观点的另一种方式是说，我们对那些关注系统的利益相关者感兴趣。我们认为关注点这个词特别合适，因为系统会涉及各种各样类型的利益相关者。

**定义** 对架构的关注点是利益相关者对于架构的需求、目标、约束、意图或愿望。

很多关注点在利益相关者中是通用的，但也有一些特殊甚至相互冲突的关注点。找到一种能够让利益相关者都满意的解决冲突方式是一项很重大的挑战。

**示例** 我们经常用三角形来表示软件开发项目的一些重要属性，其中的三个角分别代表成本、质量以及推向市场的时间。理想状况下，我们希望项目拥有高质量、零成本，并可以立即交付，但那是不可能的。图 2-4 所示的质量三角形显示，我们必须对这三种属性进行折中处理，最好的情况是你能够达到其中的两个目标。在这个图中，三角形的顶点分别代表这些所要达到的质量，我们也在图中显示了质量的一些可能的组合情况，以说明它们是如何相互影响的。

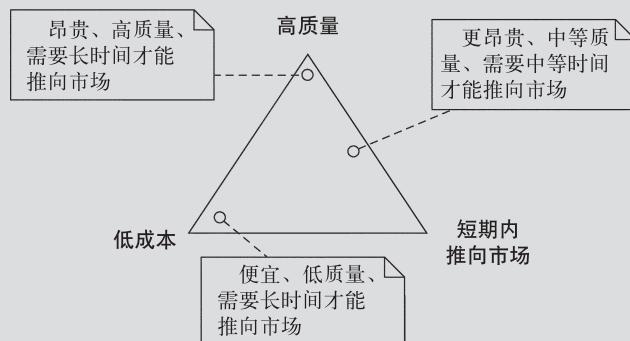


图 2-4 质量三角形

例如，构建高质量的系统可能会花费更长的时间，并且花费更多成本。我们经常可以减少最初定下的开发时间，但是，假设成本是不可变的常量，那么这就需要以付出降低交付软件的质量为代价。

这些属性中的一种或多种可能对于不同的利益相关者很重要，而架构师的职责就是要知道哪些属性对于哪些人重要，并在必要的时候达成一种可接受的折中方案。我们会在第二部分进一步讨论这个问题。

### 2.3.3 利益相关者的重要性

利益相关者（显性或隐性的）会把握架构的整体状况和方向，而创建这个架构只是为了构建能够让他们获利并满足他们需求的系统。利益相关者最终会对最终产品或系统的范围、功能、操作特性和结构做出决定，当然这要在架构师的指导之下进行。没有利益相关者，就没有创建架构的必要性，因为那样就没有人需要这个系统，也没有人来构建它、部署它或者为它付钱了。

**原则** 创建架构就是要满足利益相关者的需要。

以上原则遵循这样的观点：如果系统没有充分满足利益相关者的需要，那么就不能认为是成功的，不管它遵从了多么好的架构实践。也就是说，架构不仅要根据抽象的架构和软件工程原则来评估，还要根据利益相关者的需求来评估。

正如我们已经看到的，不同利益相关者的需求彼此冲突的情况会经常出现。对于这种问题没有简单的答案，而在这种情况下，通常都要由架构师进行有效平衡（例如，在对性能要求高的系统中接受高维护成本的要求，这是由优化和整合所需要的系统元素以减少请求处理延迟的等级导致的）。

**原则** 好的架构要成功地解决利益相关者的关注点，当关注点冲突时，要以利益相关者能够接受的方式进行平衡。

第二部分会详细探究利益相关者这个概念，并说明在创建架构的过程中如何对其进行分类、识别、选择和管理。

## 2.4 架构描述

软件系统的架构是一个非常复杂的事物。架构师职责的一部分就是要向需要理解它的人们描述这种复杂的事物。架构师可以通过架构描述来达到这个目的。

**定义** 架构描述（Architectural Description, AD）是一系列产出物，它们会以利益相关者能够理解的形式编写架构的文档，并说明架构如何满足他们的关注点。

上述“产出物”包括很多内容：不仅有架构模型，还有范围定义、约束和原则。我们会在第二部分、第三部分详细讨论这些内容。

对架构的描述需要同时表现出架构的本质和细节，也就是说，必须提供能够总结这个系统的整体概况，还必须分解为能够验证的、足够细的详细内容，并且所描述的系统能够基于此构建。

尽管每个系统的确都会有架构，但是遗憾的是，并非每个系统都拥有架构描述。即便是为架构编写了文档，它也可能只是对一部分编写了文档，或者那些文档已经过时或不再使用。

因此，从严格意义上来说，我们的定义描述的是好的架构描述。然而，如果一个架构无法被利益相关者理解，或者无法向他们说明能够满足关注点，那么这个架构也就没有存在的价值了——事实上，它更多是一种债务而不是资产。架构描述需要包含所有（理想状况下只包含）与那些需要理解架构的利益相关者沟通所需要的信息。

**原则** 尽管每个系统都有架构，但并非每个系统都拥有能够通过架构描述进行有效沟通的架构。

当然，如果架构描述不充分，那么如你所想实现架构方面的想法的机会也会大大降低。

**示例** 之前提到的航空公司订票系统的架构设计主要关注的是静态结构（关键的软件和硬件元素，以及它们是如何组织在一起的），而对外部行为（与那些元素的交互方式，从而能够对用户的请求做出响应）的关注则不够。因为大多数用户的桌子对面都会坐着客户，或者是在电话的另一端，因此快速响应时间和系统可用性非常重要。

对于这样一种系统的架构设计，如果没有考虑到任何系统质量属性的细节，特别是，如果没有清晰定义响应时间需求和性能模型，那么当部署系统以后，系统性能的表现会很差，特别是在负载高峰的时候。

这个问题的解决方案就是要识别出一组用户，他们可以对性能需求达成一致，这样架构师就可以根据分析和测试的实际情况进行平衡。这有助于在系统的生命周期中出现性能问题的时候，减少需要做大量改善和调优的工作。

架构师会编写架构描述，而且他也是架构描述的主要用户之一。你可以把它用作记忆辅助、分析的基础、决策的记录等。然而，你只是架构描述的用户之一。所有利益相关者都需要理解与之相关的架构（或者至少是一部分）。如果架构描述对此没有帮助，那么就意味着你的这项工作是失败的。

**原则** 好的架构描述能够与合适的利益相关者有效并一致地沟通架构的关键内容。

现在有很多技术、模型、架构描述语言以及其他方式来编写架构文档。为特定的系统开发过程选择合适的方式也具有很大的挑战性。需要考虑到系统的特点，以及利益相关者的技能和能力。

第二部分会探究架构描述概念的细节，而第三部分、第四部分会说明架构描述中的不同元素，以及如何创建这些元素。

## 2.5 核心概念之间的关系

图 2-5 中的 UML 类图说明了我们的核心概念之间的重要关系。这个图说明了我们到现在已经讨论过的概念之间的联系，如下所示：

- 构建系统是为了应对利益相关者的需求、关注点、目标以及目的。
- 系统架构由大量架构元素以及元素之间的关系组成。
- 可以通过架构描述文档来说明系统的架构（所有、部分或没有）。事实上，对于给定的架构有多种可能的架构描述，有些好一些，有些比较差。
- 架构描述会为利益相关者提供架构文档，并说明它是如何满足需求的。

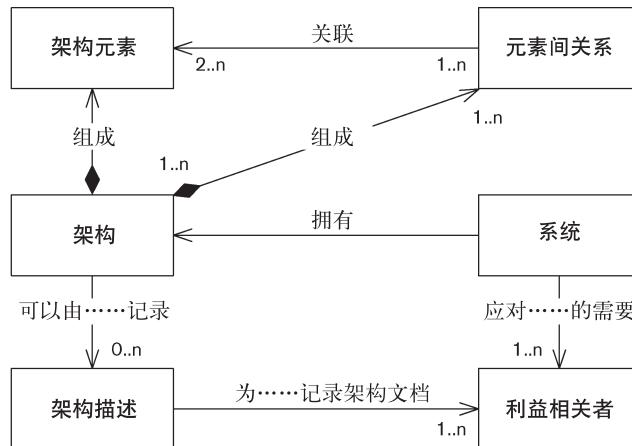


图 2-5 核心概念关系

我们在图 2-5 中以及本书中都会使用标准的 UML 规约。其中矩形表示架构概念，带有箭头的直线表示从一个概念到另一个概念的关系。线的起始端带有黑色菱形块表示“由……组成”的关系。每种关系的基数（一个概念能够关联到多少个另一种概念）显示在每条线的结束端。我们还为关系添加了注释，以简要说明它的意义。

## 2.6 小结

本章中我们定义并讨论了一些在本书以后的内容中都会使用的概念和术语，从而为后续章节奠定了基础。

- 系统的架构定义了它的静态结构、动态结构、外部可见行为、质量属性以及应该引导其设计和发展的原则。尽管不会一直声明，但每个概念都非常 important。每个计算机系统都有架构，即便我们并不了解它的架构。
- 系统的候选架构是有可能展现出系统所需要的外部可见行为和质量属性的架构。大多数问题都会有多个候选架构，而选择最佳方案则是架构师的职责所在。
- 架构元素是系统的组成部分，我们可以清晰地识别出它，并且它对于架构很有意义。
- 利益相关者是对架构的实现感兴趣或者关注的个人、小组或实体。利益相关者包括用户以及很多其他类型的人，像开发者、运维人员以及银行等。创建架构就是为了满足利益相关者的需要。

- 架构描述是一系列产出物，它们会以利益相关者能够理解的方式制定架构的文档，并说明架构如何满足他们的关注点。尽管每个系统都有架构，但并非每个系统都拥有有效的架构描述。

## 2.7 延伸阅读

我们已经让本章中的语言和概念与所知的软件架构领域最新的通用标准——也就是 ISO/IEC 标准 42010[ISO11]（它是针对架构描述从 IEEE 标准 1471-20000 发展而来的）——一致。根据它自己的介绍，这个标准说明了“通过使用架构描述创建、分析和维持系统架构”。我们的概念模型也基于该标准中所提出的一个模型。

我们关于软件架构概念的大多数想法都基于软件工程协会的软件架构小组的工作成果。Bass、Clements 和 Kazman 所著的书 [BASS03] 详细介绍了软件架构领域的主要观点，并为我们所提出的基本概念提供了深度和背景内容。

关于软件架构最早的一本书是 Shaw 和 Garlan 撰写的 [SHAW96]。该书对软件架构的基本观点给出了最小化和精致的介绍，包括对架构描述、架构样式以及可能的工具支持的概述。而在此之前，Perry 和 Wolf 在软件架构领域的一篇论文 [PERR92] 也非常值得一读，它专注于规则的重要元素。

如果你想要对软件架构有更多了解，那么跨学科的著作《系统架构的艺术》(Art of System Architecting) [MAIE09] 会非常有用。该书的新奇之处在于对架构的观点进行了介绍和讨论，认为它是一系列可以在所有复杂系统领域中使用的原则和技术。该书的重点特别放在了架构启发式方法上，并且提供了一系列有趣的启发式方法。这些例子来自于建筑、工程、社会系统、IT 以及协作系统。

在本书第 1 版和第 2 版出版的几年中，关于架构这个主题出现了大量值得推荐的书籍。我们在所有的作品中都试图强调，很重要的是要把你的架构工作专注于解决所面临问题的重要方面，而不是在每个案例中都试图使用所有视点和视角。George Fairbanks 的著作《Just Enough Software Architecture》[FAIR10] 是对这项工作的实践指南，其中展示了如何实践“风险驱动架构”(risk-driven architecting) 来根据面临的风险剪裁架构。Ian Gorton 的著作《Essential Software Architecture》[GORT06] 对很多重要的软件架构话题进行了简明而具有实践意义的介绍；Richard Taylor、Neno Medvidovic 和 Eric Dashofy 在 [TAYL09] 中对主题做了非常易于理解的介绍。

如果你有兴趣为你的软件架构工作定义正规的过程，那么 Peter Eeles 和 Perter Cripps 的著作《The Process of Software Architecting》[EELE09] 会很有帮助。

# 第二部分

## 软件架构过程

第6章 软件架构过程简介

第7章 架构定义过程

第8章 关注点、原则和决定

第9章 确定并引入利益相关者

第10章 识别并使用场景

第11章 使用样式和模式

第12章 创建架构模型

第13章 创建架构描述

第14章 评估架构

## 第⑥章

# 软件架构过程简介

本书的目的并不是要定义另一种软件开发方法，也不是要从根本上改变现存的软件开发生命周期模型。然而，很多软件开发方法都无法清晰地定义开发生命周期中软件架构的角色。如果说讨论的话，他们经常会把架构定义仅仅视为软件设计的第一部分，我们希望这本书会告诉你，这种观点过于简单了。

正如我们已经看到的，架构定义是一种广泛的、有创造性的、动态的活动，它不仅仅要获取信息，而且需要找到利益相关者的关注点、评估各种可选方法并做出权衡。开始时，你的利益相关者可能对于范围、目的和优先级有着完全不同的意见。你可能会根据工作中得到的信息，在中途改变方向，这种可能性还是非常大的。

尽管每种形式都有区别，但是通常在任何项目中都会执行一系列核心活动，将其作为架构定义的一部分。我们会在接下来的章节中描述这些活动。你可能需要在架构定义过程中做其他工作，但是你可能会执行大多数在第二部分中描述的活动，以避免将来可能出现的问题。

在第二部分中，我们首先会展示一般且简单的架构定义过程，你可以通过它帮助你制订自己的架构定义工作计划，并使其与开发的其他部分的计划保持一致。接下来的章节会带你了解过程中的各个关键活动，它们分别是：

- 对于范围和情境、约束以及基本的架构原则达成一致意见；
- 确定并引入利益相关者；
- 确定并使用架构场景；
- 使用架构样式和模式；
- 创建架构模型；
- 编写架构文档；
- 验证架构。

对于上述的任何一种活动，我们都提供了实践性的建议和指导，包括帮助你确保不会忘记任何内容的检查列表，以及延伸阅读内容。

# 第三部分

## 视点类型

第15章 视点类型简介

第16章 情境视点

第17章 功能视点

第18章 信息视点

第19章 并发视点

第20章 开发视点

第21章 部署视点

第22章 运维视点

第23章 保持视图一致性

## 第 15 章

# 视点类型简介

第三部分集中了我们的七种核心视点：情境、功能、信息、并发、开发、部署和运维。有很多方法都可以用来指定架构描述的结构，但是我们认为这一系列的视点很好地把架构描述分为数量可管理的部分，同时确保广泛覆盖关注点。

图 15-1 显示了使用这些视点创建的视图之间的关系。

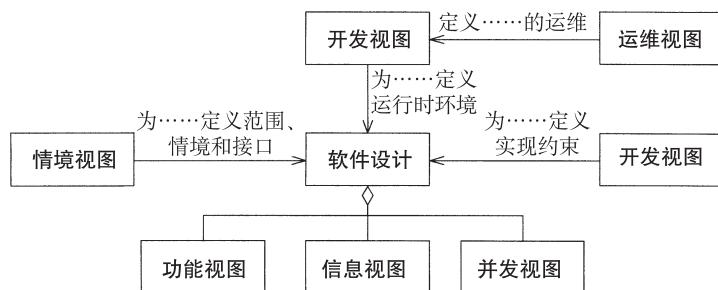


图 15-1 视图关系

为方便起见，我们在表 15-1 中再次说明了最初在第一部分展示的视点分类方法。

对于每种视点，我们展现了以下细节：

- 视点所应对的最重要关注点，并识别出最可能对它的视图感兴趣的利益相关者。
- 你可能构建并展示视图的最重要模型，还有使用的标记法以及构建系统的活动。
- 要知道的一些问题和缺陷，以及缓解它们从而降低风险的技术。
- 当创建视点和对其进行评审时所要考虑问题的检查列表，这有助于确保它的正确、完整和精确。

篇幅所限，我们只能展示每种视点的概况，以及某些复杂和详细的主题。第三部分的大多数章节都可以扩展成一本书。我们的目标是要让你开始行动，因此，每个视点的章节都包含了大量对延伸信息的资源引用。

表 15-1 视点目录

视点	定义
情境	描述系统与其环境（与之交互的人、系统和外部实体）之间的关系、依赖和交互。很多系统利益相关者都会对情境视图感兴趣，而该视图在帮助他们理解其责任以及如何与他们的组织相关方面会起到重要的作用
功能	描述系统的运行时功能元素，以及它们的责任、接口和主要交互关系。功能视图是大多数架构描述的基础，通常是利益相关者想要阅读的第一部分。它会驱动系统结构（如信息结构、并发结构、部署结构等）的形式。它还会对系统的质量属性（如变更能力、安全防护能力、运行时性能等）起到重大影响
信息	描述架构存储、操作、管理和分发信息的方式。所有可视化计算机系统的最终目的是用某种形式来操作信息，并且这个视点会对静态数据结构和信息流创建完整但高层次的视图。这项分析的目的在于回答关于内容、结构、所有权、延迟、引用和数据迁移的重大问题
并发	描述系统的并发结构，并把功能元素映射到并发单元上，以清晰地确定系统能够并发执行的各个部分，以及这是如何协作和控制的。这需要创建模型，用来展示系统将会使用的进程和线程结构，以及用于协调它们的操作的进程间的沟通机制
开发	描述支持软件开发过程的架构。开发视图会和利益相关者沟通感兴趣的架构内容，这些利益相关者会参与系统的构建、测试、维护和改善活动
部署	描述系统将要部署的环境，以及系统对它的依赖关系。这个视图会获得系统需要的硬件环境（主要是处理节点、网络连接以及所需要的磁盘存储设备）、每个元素的技术环境需求，以及软件元素和执行它们的运行时环境之间的映射
运维	描述当系统在生产环境中运行时，将如何对其进行运维、管理和支持。除非是最简单系统，否则安装、管理和运维系统都是非常重要的任务，必须在设计时就考虑并制订计划。运维视点的目标就是确定系统级的、系统利益相关者关于运维方面的关注点，并确定应对这些关注点的解决方案

## 第 24 章

# 视角类型简介

第四部分会集中描述各种视角。我们会详细描述表 24-1 中所列的各种视角，每章描述一种。我们已经选择更彻底地研究这一系列视角，因为它们所应对的质量属性对于大多数信息系统都非常重要。

我们会在第 29 章概述表 24-2 中所列的视角。由于篇幅所限，我们以更加简要的方式来说明这些视角，尽管它们通常也很重要，但是它们并不像表 24-1 中所列的那些视角被普遍使用。

表 24-1 详细描述的视角

视角	要求的质量
安全性	系统可靠地控制、监控和审计谁能够在哪种资源上执行哪种动作的能力，以及检测安全漏洞并从中恢复的能力
性能和可伸缩性	系统以可预测的方式在指定的性能指标下执行的能力，以及将来处理逐渐增长的处理量的能力（如果需要）
可用性和弹性	系统在需要的时候完全或者部分操作，并能够有效地处理影响系统可用性的故障的能力。
演进性	系统在面临不可避免的变更时（这在所有系统部署之后都会遇到）保持灵活的能力，需要平衡成本来提供这样的灵活性

表 24-2 简要描述的视角

视角	需要的质量
可访问性	系统被残疾人士使用的能力
开发资源	系统在已知的约束（与人员、预算、事件和材料相关）下设计、构建、部署和操作的能力
国际化	系统独立于特定语言、国家和文化的能力
位置	系统克服元素的绝对位置以及它们之间的距离所带来的问题的能力
法规	系统遵守本地或国际法律、准法律规则、公司政策以及其他规则和标准的能力
易用性	与系统交互的用户的有效工作的轻松程度

对于每个视角，我们都描述了以下细节：

- 视角对视图的适用性，也就是哪些视图最可能被应用这个视角所影响。
- 视角解决的最重要问题。
- 描述向架构应用视角所需活动。
- 当架构没有表现出视角所说明的所需质量属性时，需要考虑作为可能解决方案的关键架构策略。
- 需要知道的某些问题和缺陷，以及缓解问题、降低风险的技术。

- 当应用视角和对其进行评审时，需要考虑的问题的检查列表，以确保正确性、完整性和精确性。

在视角类型目录中，我们只是对某些复杂和详细的话题进行了概述。我们的目标是提供每个领域的基本信息，因此，每个视角的章节都包含了大量资源或者延伸信息的引用。

正如我们在第一部分所说的，可能有很多视角都可以应用到特定的架构上，不过在所有视图的情境下考虑所有视角是不可行的。并非所有视角与所有视图都相关，甚至不会与所有系统相关，在很多实例中，你根本不需要考虑某些视角。充分利用视角的关键在于，考虑每个视角对于架构的重要程度，并依此对方法进行剪裁。