第1部分 软件架构概述

第 1 章　软件架构是什么

第 2 章　软件结构

第 3 章　关键技术、支撑技术与技术路线

第 4 章　质量属性

第 5 章　软件架构设计的原则

第 6 章　如何完成软件架构设计

第 1 章

软件架构是什么

横看成岭侧成峰，远近高低各不同。
不识庐山真面目，只缘身在此山中。

——《题西林壁》宋·苏轼

软件架构是从建筑架构类比过来的概念，我们不能完全按照建筑架构来解释软件架构。本章将讨论软件架构的定义、作用以及软件架构设计的一般方法。

1.1 难以定义的软件架构

软件是按特定顺序组织的计算机数据和指令，是计算机中的非有形部分。软件是信息系统的灵魂，在相同硬件上运行不同软件会产生截然不同的效果，这正是软件的魅力所在。由于软件是无形的，在设计和研究软件时，我们通常通过类比（或称隐喻）将某些有形、易于理解的概念应用于软件理论，例如“容器”“适配器”“工厂”等。这些术语大多源自其他行业，用来类比软件中的某种结构或概念。软件架构也是如此，如果将软件比作建筑物，软件架构就如同建筑物的架构，起到骨架和支撑作用。然而，这两者有很大不同，建筑物的架构是可见的，物理存在的。在建筑工地，我们能很容易看到建筑物的骨架，这些骨架先于其他部分搭建。而软件本身是柔性的，软件架构在软件的生命周期中是不断变化和发展的。虽然类比可以帮助我们理解软件架构，但这不足以作为软件架构的定义。

1.1.1 针对软件架构定义的不同观点

给软件架构（也称为软件构架或软件体系结构[34]）下定义是一项困难的工作，业界对此有许多争论。Martin Fowler[1]总结了几种观点，我们逐一分析。

一种观点认为，软件架构是系统的基础部分，或是顶层模块的组织方式。然而，这种观点的问题在于缺乏客观的标准来定义什么是“基本的”或“顶层的”，这两个概念具有很强的主

观性。Bob 大叔（Robert C. Martin）在《架构整洁之道》[2]一书中指出，“高层的”架构与“低层的”设计之间没有明确界限——“所谓低层和高层本身只是一系列决策组成的连续体，并没有清晰的分界线”。

还有一种观点认为，软件架构是“项目早期需要做出的设计决策”，但这更像是“希望在项目早期可以做出的正确决策”。“项目早期”这一概念存在问题，软件开发是一个循环迭代的过程，每次循环都需要做出设计决策。在实际项目中，对比项目初期的决策与若干次迭代后的开发结果，我们常常发现初期的设计已经面目全非。

根据 Ralph Johnson 的说法:“架构是那些重要的东西……无论它具体是什么”（Architecture is about the important stuff. Whatever that is）。也就是说，所有重要的成分都是架构的一部分。这种说法虽然正确，但过于宽泛，不容易把握。架构师需要能够识别出哪些是项目中的关键元素，这些元素如果得不到适当控制，可能会导致严重问题。然而，什么是“重要”的成分，带有很强的主观性，且在设计初期不易验证。

还有一种观点认为，软件架构主要关乎软件的结构[8]。这种观点有一定的道理，比如分层架构和微服务架构描述的就是软件的结构。但这种观点并不全面，因为一些架构模式，如事件驱动架构，关注的是协作方式，而非结构。除此之外，与质量属性相关的战术性机制也是架构设计的一部分，但这些机制并不属于软件结构的范畴。

笔者认为，软件架构的定义之所以难以确定，正是由于软件的多样性和技术的快速发展所致。软件架构是多维的，在不同的软件开发阶段、不同层次上发挥着不同的作用、表现形式也是不同的，综合考虑各个视角，才能给出一个相对完整的描述。

1.1.2　不同视角的软件架构

从不同的视角观察，看到的软件架构是不同的。用户通常从业务角度出发来考量软件架构。以生产管理系统为例，它包括“数据采集”“车间管理”“调度”“计划”和“辅助决策与综合查询”等子系统。如果从业务分层的角度看，这些子系统可以归类为“操作层”“管理层”和“决策层”，形成面向业务的分层架构。

从软件技术的角度来看，这些应用子系统采用的技术基本相同。按三层结构或四层结构划分，它们可分为表示层、应用层和数据访问层，或者表示层、应用层、领域层和基础设施层，这是从技术角度的分层架构。

如果将不同视角的系统架构综合在一起，就需要使用多维架构图来进行描述。图 1-1 取自实际项目（金陵石化炼油厂 NR_CIMS 工程，2000 年）[4]。

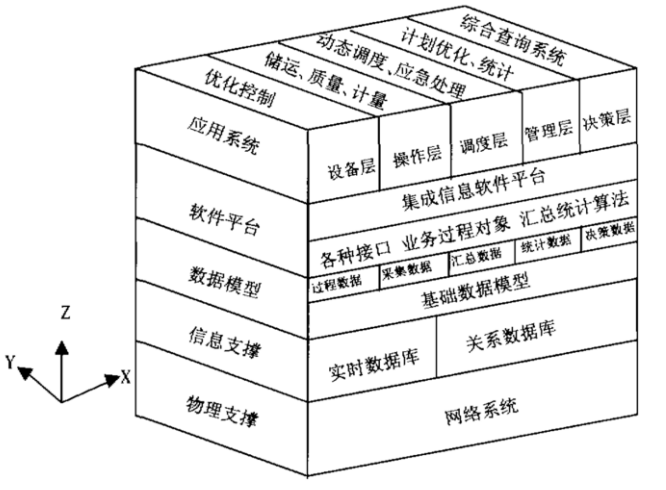

图 1-1　金陵石化炼油厂 NR_CIMS 工程总体架构图

同一应用软件系统，从不同

角度观察所得到的架构视图是不同的。只有当这些视图相互补充时，才能展现出系统的全貌。

1.1.3 不同层次的软件架构

软件架构是复杂的，一个大型应用系统需要分解为若干子系统，每个子系统可能包含多个独立运行的模块，模块内部也需要进行逻辑分层……，从不同的层次观察，软件架构的表现是不同的。图1-2所描述的架构是属于哪种架构呢？

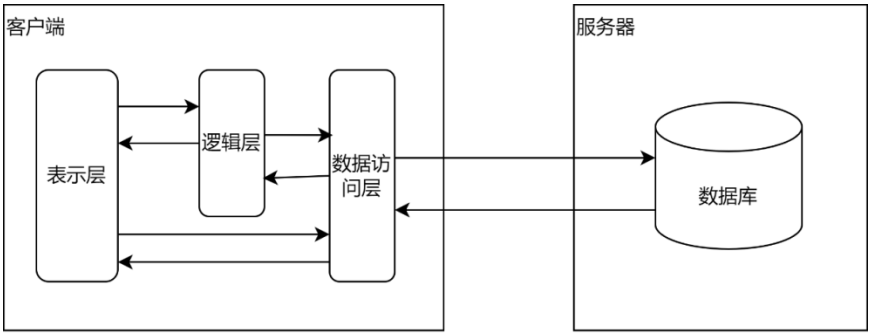

图 1-2　不同层次的软件架构

图 1-2 展示了两个层面的架构：系统的整体架构和客户端软件的内部架构。系统的整体架构采用了“客户端/服务器（C/S）”架构模式，而客户端软件内部采用了三层架构模式。

这是一个简单的例子，实际应用中可能涉及更多层次的架构，并且每个层次内部也可能存在嵌套。针对不同的层次，软件架构的描述方式也会不同。

在设计软件架构时，我们不能仅仅关注顶层架构设计而忽略其他层次的架构设计。这会导致架构设计不完整，许多代码可能脱离架构设计的控制，形成一些非设计的“自发架构”。如果软件需要升级或扩展，这些“自发架构”就会成为障碍：它们起到了架构的作用，但却在架构设计之外，除原作者外，其他人难以理解。因此，架构设计需要避免这些“自发架构”的出现。

1.1.4 不同开发阶段的软件架构

软件架构之所以难以精确定义，主要原因在于其在软件开发生命周期中的各个阶段呈现出不同的形态和作用。脱离了特定开发阶段的背景，软件架构的定义就会变得模糊不清。因此，准确理解软件架构，需要将其置于相应的开发阶段中进行考量。

在项目规划阶段，软件架构主要关注宏观的技术路线选择，包括整体架构模式、后端数据库技术选型、前端架构风格以及认证方式等关键决策。这些选择为项目的后续发展奠定了技术基础。

进入概要设计阶段后，软件架构开始具体化。此时，需要对规划阶段确定的技术路线进行细化。架构设计图也从抽象的技术路线展示转变为包含具体技术细节的描述。

到了开发实施阶段，软件架构与具体的设计和实现过程紧密相连。在这一阶段，软件架构需要在不同层面上解决不同类型的问题，这些层面大致可以分为系统层面、子系统层面以及子系统内部三个层次。在系统层面上，主要关注各个子系统及模块之间的集成方式和通信协议。在子系统或模块层面上，则需要确定具体的开发方式和架构模式，如领域驱动开发、四层架构等。而在子系统内部层面，架构设计将更加关注微观的实现细节和组件交互。

因此，软件架构是一个随着开发阶段不断推进而逐渐具体化和细化的过程。要准确理解和定义软件架构，必须紧密结合其所处的开发阶段和上下文环境。

1.1.5　不断发展变化的软件架构理论

软件架构作为独立的研究领域，吸引了许多研究人员和学者的关注，他们试图建立一套完整的软件架构理论体系。然而，软件技术的发展过于迅猛，导致理论始终难以跟上实际的进展。

早期，软件开发常常与建房子进行类比，“万丈高楼平地起”，房子是由一砖一瓦盖起来的，软件则是通过一行行代码编写而成。建房需要设计，要确定主体结构，然后在此基础上设计其他细节。软件也是如此，在开始编写代码之前，需要通过某种方式对将要实现的软件进行描述，这就是软件架构。在 1995 年之前甚至更早之前，软件开发中的设计工作至关重要。在编写代码之前，必须绘制流程图，然后根据流程图编写代码。这样做的原因在于，当时缺乏成熟的 IDE（Integrated Development Environment，集成开发环境），只能通过编译时才能发现程序中的错误，且没有完善的调试工具。在这种情况下，如果没有设计就直接编写代码，那么即使是小规模的程序，也需要花费大量时间。早期的架构设计包括代码实现的微观层面。

随着软件开发技术的进步，出现了完善的 IDE 工具、测试工具和调试工具，这些工具能够实现即时编译、测试和调试。因此，代码编写阶段的形式化设计不再重要，而设计的主要内容转向了模块和组件的划分。软件需求的不确定性，设计内容需要频繁变更，导致软件开发的周期延长，费用超支，并且无法完全满足用户需求。在这种背景下，极限编程、敏捷开发[45]以及领域驱动设计[22][23][24][25]等新型软件开发方法应运而生，这些方法将设计与编码融为一体，去除了大量的形式化设计。这时的软件架构可以采用成熟的架构模式或风格进行描述。

如今，软件架构设计主要面向大型分布式应用，预先考虑应用的可用性、安全性、性能和可修改性等因素。随着技术的不断进步，软件架构设计的内容也在持续变化。

从与软件架构相关的权威著作之一——《软件架构实践》的不同版本[5][6][7][8]中，我们可以看到软件架构理论的不断发展变化。

《软件架构实践》已出版 4 个版本，跨越了 20 年，对比各个版本可以发现，内容有很大的变化，这些变化反映了软件架构理论的演进。在第一版[5]（2002 年）中，书中详细列出了完整的软件架构风格（Software Architecture Style，或称样式）的分类和说明，并提出了创建软件架构的设计语言和设计方法，软件架构作为一个完整的理论体系进行设计。然而，在第四版[8]（2023 年）中，这些内容已经被删减，取而代之的是包括质量属性、软件架构的评估以及与软件工程相关的内容。这是因为，软件架构设计不可能脱离软件开发过程独立存在，架构设计本身就是软件开发过程的一部分。软件技术和软件开发方法的进步，极大地影响了软件架构理论的演变。

因此，软件架构理论在不断发展和变化，软件架构所起的作用和设计方法也在不断变化，这对人们对软件架构的理解产生了深远影响。

1.2　软件架构的范围

回到软件架构的定义，Ralph Johnson 的说法（架构是那些重要的东西……无论它具体是什么）

在实践中更具可操作性。我们不必拘泥于软件架构的具体定义，只要能找到“那些重要的东西”即可。本节将根据鸭子理论（Duck Test）来确定软件架构的范围和内容。

1.2.1 使用鸭子理论划定软件架构范围

鸭子理论是一种思维方式，它的基本逻辑是：如果一只动物看起来像鸭子，走路像鸭子，叫起来像鸭子，那么它就是一只鸭子。类似的，如果难以给软件架构下一个明确的定义，那么借鉴鸭子理论，通过寻找软件架构的一些特征来判断。如果某个问题符合这些特征，就可以认为它是需要在架构层面解决的问题，相关的工作也属于软件架构设计的一部分。我们根据实践中常遇到的问题进行总结，认为以下三方面的内容是软件架构设计需要关注的重点：

- 软件的结构。
- 关键技术与支撑技术相融合。
- 与软件质量属性相关的机制和解决方案。

如果遇到涉及这三方面内容的问题，就可以认为这些问题属于软件架构范畴，应该从架构设计的层面加以解决。

这种定义方式可能不完美（许多原本不属于架构范畴的问题，可能一开始被作为架构问题进行了处理，这类偏差易于纠正），但它具有很强的操作性，易于在项目实践中应用。

1.2.2 软件架构描述了软件的结构

软件架构是对软件结构的描述，涉及软件的组成结构以及各部分之间的集成关系，均属于架构范畴。如果一个企业级应用被分解为若干子系统，那么这些子系统间的数据交互关系和调用关系就是该企业级应用宏观架构的一部分。若某个子系统被分解为表示层、应用层、领域层和基础设施层，那么这些层次间的接口调用关系和装配关系就是该子系统的架构。该子系统表示层中的组件划分和集成关系则构成表示层内部的微观架构。

在 1.1 节中提到的从不同视角、不同层次和不同开发阶段观察软件，会得到不同的架构描述。仔细分析后发现，观察结果的差异与观察尺度有一定的关系，“不识庐山真面目，只缘身在此山中”。从宏观到微观，或从整体到局部，所看到的软件架构也有所不同。

回到前面的用户视角，这是大尺度的结构划分，在这个尺度上，软件架构关注的是各个子系统之间的数据交换方式。而从技术角度看，分层架构关注的是每个子系统的内部构成方式。如果将这两种方式结合起来，可以用图 1-3 来表示。

图 1-3 展示了宏观架构和微观架构之间的关系。在实际项目中，我们通常不会使用这种复合图形进行架构描述。然而，在进行架构设计时，一定要明确架构所处的尺度，清楚地认识到所设计的架构是宏观还是微观。

再来看不同层次的架构图。若不同视角的架构是从业务角度出发的，那么不同层次的架构是从技术角度出发的。1.1 节中的 C/S 架构图就是一个例子。图 1-4 展示了另一个例子，描述了使用微服务架构构建的简单电商应用。

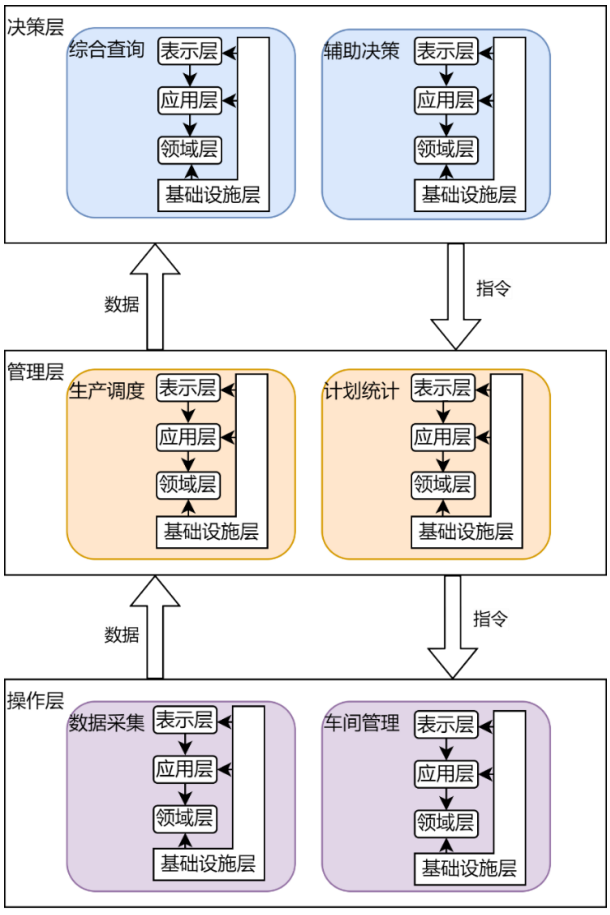

图 1-3　宏观架构和微观架构的关系

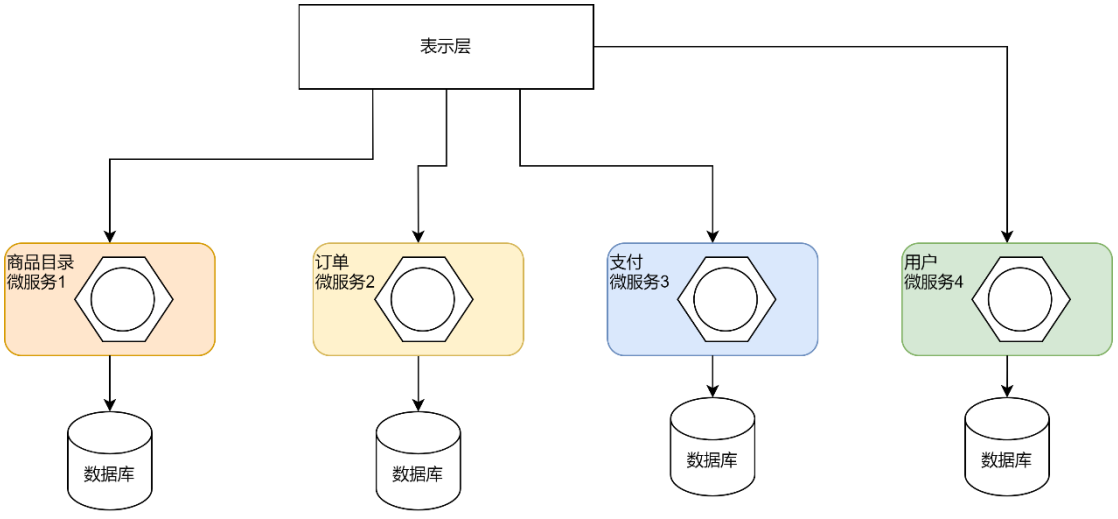

图 1-4　使用微服务架构的电商应用

图 1-4 中略去了微服务之间的调用关系。整体应用架构采用的是微服务架构，而每个微服务内

部则采用六边形架构。

除此之外，我们还可以发现，如果划分得当，从宏观层面设计的软件架构，技术和业务是可以对齐的：商品目录、订单、支付、用户这些按照业务划分的服务，恰好也是技术实现的模块。在上述例子中，可以看到两个层面的软件架构：一是针对软件整体的架构（宏观层面），二是针对软件内部模块组成的架构（微观层面）。

我们可以这样理解宏观和微观两个层面的架构：一个软件由若干有相互联系的业务模型组成，这些模型之间的关系构成软件的整体架构（宏观架构），也可以称为“模型之间的架构”；而围绕某个业务模型的软件模块，构成微观架构，即“模型周围的架构”。在上面的微服务架构中，存在 4 个模型：“商品目录”“订单”“支付”和“用户”。整体架构描述了这 4 个模型之间的关系，而每个微服务内部的架构都描述了围绕这些模型的架构。

现在来看宏观架构和微观架构之间的界限。宏观架构指的是构成应用系统的可独立运行部分及其之间的关系。所谓可独立运行，是指这些部分在运行时为独立的进程，可能运行在同一台硬件服务器上，也可能在不同的容器或独立服务器上运行。这些独立运行的部分具有物理边界，它们之间的互相调用需要通过网格协议进行，不能在同一进程内直接完成。微观架构则是这些独立运行部分的内部结构。在微观架构中，如果有分层，也是逻辑分层，运行时通过某种方式实现内存中的互相引用。

单体架构和微服务架构是两个极端情况。如果一个应用系统采用单体结构，那么宏观架构会非常简单，因为所有代码都运行在同一进程中。而微观架构则相对复杂，需要在内部进行逻辑分层。假设我们使用领域驱动设计的方法开发这个应用，那么可以将该应用按照业务分解为多个子域，每个子域的解决方案就是一个限界上下文，每个限界上下文内部有独立的领域模型。由于采用单体应用架构，这些限界上下文之间的界限只能是逻辑边界。

与单体架构对应的另一个极端是完全采用微服务架构。如果将一个应用的所有功能都拆解成独立的服务并部署为微服务，那么微服务内部的微观架构会相对简单，但宏观架构则非常复杂，因为所有服务之间都需要通过网络进行通信。在实际项目中，架构通常介于两者之间：一个大型应用既不会采用独立纯粹的单体结构，也不会采用分解到极致的微服务架构。我们通常采用按业务规则划分的方式，使得限界上下文的边界与微服务的物理边界一致。微服务可以独立开发，最后将它们集成在一起，形成完整的应用。

1.2.3 软件架构的关键技术和支撑技术

关键技术是指所开发软件的基础技术，是软件的竞争力所在。如果没有这些技术，软件就无法实现。因此，关键技术决定了软件的存在价值，必须在设计之初确定。

以 Docker 为例进行说明。使用 Docker，我们可以在一台宿主机上创建若干相互隔离的容器，在容器中运行独立的应用。创建相互隔离的容器是 Docker 的核心功能，而这个功能建立在 Linux 的 namespace、controlled groups 和 rootfs 基础之上。因此，Docker 的关键技术是使用 namespace、controlled groups 和 rootfs 来创建互相隔离的容器。

不同类型的软件有着不同的关键技术，主要包括以下几种类型：

- 底层技术：许多软件的关键技术取决于其所依赖的类库、框架、组件等提供的功能。区分是否为关键技术的标准主要看该技术是否具有可替换性。只有不具有可替换性的技术才算作关键技术，而可替换的技术通常可以在架构层面解决。例如，Docker 的关键技术是基于

Linux 内核的技术，这些技术不具备可替换性。

- 关键算法：与计算相关的软件通常有关键算法，当软件所需要解决的问题可以抽象为计算模型时，实现这个计算模型的算法即为关键技术。例如，我们可以将业务审批流程抽象为有限状态机，有限状态机的算法就是关键算法，也是软件的关键技术。
- 业务模型：许多信息处理类的软件看似没有明确的关键技术，例如人力资源管理系统和设备管理系统。从软件结构和使用的软件技术上看，它们几乎没有区别。那么它们的区别在哪里呢？就是它们具有不同的业务模型。在这类软件中，业务模型就是关键技术之一。业务模型的存在形式决定了软件的结构、可扩展性、可维护性等质量属性。

在确定关键技术后，接下来需要确定软件实现的途径，这就是技术路线上要解决的问题。技术路线是指为实现软件目标所采取的技术手段、具体步骤及解决关键性问题的方法。技术栈的选择就是技术路线的一部分。当我们说“前端使用 Vue3+Element Plus，TypeScript 作为编程语言，后端使用 Spring Boot 架构，采用 RESTful API 方式提供服务”时，实际上是在描述技术路线。

技术路线中还包括一些重要决策，也属于软件架构的范畴，这些决策决定了后续设计的软件结构和质量属性。常见的决策是业务模型的存在形式，可能选择表模式、活动记录或领域模型，所选择模式直接影响后续的软件架构风格。有些决策涉及实现细节，如关键字的类型、生成方式和生成时机、日期时间的处理，以及本地化与国际化等。

技术路线中涉及的技术属于支撑技术范畴，这些技术决定了软件如何实现，因此需要在架构设计中明示。

在第 3 章中，我们将详细讨论关键技术与支撑技术在软件架构中的作用。

1.2.4　软件架构决定了软件的质量属性

软件不仅需要提供用户所需的功能，还应具备其他特性，以满足用户的整体需求。读者可能曾遇到过崩溃的网络应用，回想一下当你填写完表单数据，单击“提交”按钮后，出现“500 错误”时的心情。由此可见，软件在运行时应具备可靠性，特别是在外部条件不佳（例如网速缓慢或时断时续）的情况下；软件应具备安全性，确保个人信息不被泄露；软件还应具备易用性，使用户能够流畅地使用。这些需求不属于功能需求，因此曾被称为“非功能性”需求。然而，使用“非功能性”来描述并不完全准确，因此现在通常使用“质量属性”来描述软件的这些特性。软件质量属性的范围广泛，包括可度量的属性（如性能）和不可度量的属性（如可扩展性、可集成性等）。

解决质量属性问题需要从架构的宏观层面和微观层面入手。在宏观层面，主要解决结构性问题。大多数软件质量属性都与软件的结构密切相关。例如，软件的结构支持模块化设计，那么该软件通常会具有良好的可修改性、可测试性和可集成性，而这些质量属性的提升将直接影响软件的可用性和性能。在微观层面，针对特定的质量属性，已有各种成熟的机制可供选择，相关内容将在第 12 章中详细介绍。

1.3　软件架构的作用

软件架构在软件开发过程中起着多方面的作用。在软件的可行性论证阶段，软件架构向用户及

其他利益相关者展示未来软件的结构和工作原理；在软件开发阶段初期，架构帮助确定开发团队的组织和分工；在软件开发过程中，软件架构作为开发的基础和测试的依据；在软件运行和维护期间，软件架构为运维工作提供指导。

1.3.1 体现软件开发的早期设计决策

在软件规划阶段，通常不涉及具体的编码工作，这一阶段的工作可能是可行性研究或初步设计，且具体的开发合同可能尚未敲定。因此，需要采用某种形式来描述未来的软件，而软件架构设计是这一阶段的主要工作之一。

在早期设计决策期间，软件架构通常以架构图的形式展示，表现高层次的结构规划，描述待开发软件的内部结构及其与外部环境的交互方式，并对软件质量属性（如性能、安全性等）提出解决方案。

1.3.2 用于沟通与交流

软件架构的另一个重要作用是用于沟通与交流，帮助软件开发的利益相关方及风险承担者理解软件。

作为一种可视化和文档化的表达形式，软件架构是项目团队内部及与客户和其他利益相关者之间的重要沟通媒介。通过架构图、模型和文档，架构师能够向非技术人员解释软件的结构和功能，确保各方对项目的理解和期望保持一致。这种沟通有助于及时发现和解决问题，减少误解和返工。

1.3.3 软件质量属性的保证

软件具有业务属性与质量属性。如果将软件的业务属性剥离，剩下部分可以看作是软件的架构属性，质量属性是架构属性的一部分。从需求角度来看，与具体业务无关的需求，可以视为对软件架构的需求。例如，安全性、性能和可维护性等需求，在软件架构设计时必须予以解决。

在架构设计阶段，需要考虑这些质量属性需求，并采取相应的设计策略和技术手段来满足。例如，通过引入缓存机制来提高性能，或采用模块化设计来提升可维护性。

1.3.4 软件工程管理的抓手

软件开发过程伴随着工程管理，开发管理者需要组织团队、分配任务，利益相关者需要了解软件的开发周期和成本等。这些都是软件工程管理需要完成的工作，而软件架构在这一过程中扮演着至关重要的角色。

在软件工程管理中，软件架构为项目管理者提供了一个清晰的框架，用于组织团队、分配任务、监控进度和评估风险。它帮助管理者更好地理解项目的复杂性和工作量，从而制定更准确的计划和预算。同时，架构的稳定性和可扩展性也直接影响软件的开发周期和成本。一个设计良好的架构能够减少开发过程中的返工和修改成本，从而提高开发效率和质量。

1.4 软件架构和软件架构模式（风格）

在日常交流中，软件架构常常和软件架构风格或软件架构模式混淆。在很多场景下，“软件架

构”实际上指的是“软件架构风格”。我们常听到有人问“这个软件的架构是什么？”实际上，提问人希望了解的是该软件属于什么类型的“软件架构风格”。

软件架构指的是一个具体软件产品或软件项目的架构。就像不同的建筑具有不同的架构一样，不同的软件也有各自的软件架构。如前文所述，软件架构包括软件的结构、质量属性相关的解决方案和关键技术。在许多情况下，软件结构需要多层次地描述，既包括宏观结构，也包括微观结构。

软件架构风格或软件架构模式是软件架构的抽象形式，类似于建筑风格（如中式风格、欧式风格等）。一个软件可以同时采用多种架构模式或架构风格，也可以在宏观层面采用某种架构模式，而在构成软件的模块中采用其他架构模式。大多数软件架构模式或风格关注软件的结构，例如分层架构、插件架构、管道-过滤架构等；也有一些架构模式或风格关注组成软件各部分之间的协作方式，例如事件驱动架构；还有些架构既关注结构，又关注协作方式，如事件驱动的微服务架构。

在设计软件架构时，可以采用软件架构模式或风格来辅助描述架构设计，但它们不能代替软件架构设计。

本书第 2 部分（第 7~14 章）主要介绍常用的软件架构模式和架构风格。

1.5　软件架构和软件框架

软件框架类似于建筑中的预制件，是一种软件制品，或者说是半成品软件。软件框架本身并不提供完整的业务功能，但它提供了基于某种技术栈的基本技术实现。在软件架构的基础上开发软件可以减少工作量。通常，软件框架符合某种软件架构模式。

软件架构可以使用软件框架来实现，但需要注意的是，不能用软件框架代替软件架构设计。良好的软件架构设计应该以业务为中心，并且应当与框架无关。

然而，在实际项目中，总是会遇到各种框架的诱惑。很多重量级的框架的确实现了从数据库访问到用户界面的技术功能，并且集成了权限认证、日志等常用功能，只需要补充相关的业务，一个应用系统便可快速开发出来。这些框架很有吸引力。

如果选择使用重量级的软件框架，必须对框架有深入的了解，特别是要注意软件的业务模型如何与所使用的软件框架集成。如果需要分解业务模型以适应软件框架，就需要格外小心。一旦业务模型被框架碎片化，开发出来的软件将变得难以理解，从而引发不易扩展和不易维护等问题。

知识拓展

在各种开发生态中，都有优秀的重量级软件框架可供使用：.NET 环境下有 ABP vNext、Orchard Core，Python 世界有 Django，等等。使用这些框架可以帮助我们快速开发应用系统，特别是在业务较为简单、易于理解的应用中。然而，在使用这些框架开发软件时，经常会陷入一种悖论：我们的初衷是节约时间，但开发过程中遇到的某些“小”问题，通常需要深入理解框架的工作原理才能解决，而这需要花费大量的时间来阅读框架文档或分析框架源代码。结果是，我们在了解框架的过程中花费的时间，往往远大于开发自身业务所需的时间。

第 6 章将详细讨论如何在软件架构设计中选择合适的框架。

1.6 本章小结

很难为软件架构提供一个非常确切的定义，因为同一个软件从不同的视角、层次和开发阶段观察，看到的内容是不一样的。因此，笔者更赞同 Ralph Johnson 的说法：“架构是那些重要的东西……无论它具体是什么”。

总结一下，这些“重要的东西”包括以下三方面内容：首先是软件的结构，软件架构需要描述软件各个层次的结构；其次是软件所涉及的关键技术和支撑技术；最后，软件架构决定了软件的质量属性，如可用性、性能、可修改性、安全性等。

软件架构有多重作用。首先，它体现了软件开发早期的决策；其次，它用于沟通与交流，向软件的利益相关者提供必要的说明；再次，它保证软件质量属性；最后，它是软件工程管理的抓手。

软件架构的内容包括关键技术的解决方案、技术路线说明、描述软件结构的框架图、软件质量属性的解决方案以及体现软件架构设计的骨架代码。

软件架构设计既包括描述软件整体的宏观架构，又包括决定架构设计落地的微观部分，这两部分缺一不可。

第 2 章

软件结构

彼节者有间，而刀刃者无厚；以无厚入有间，恢恢乎其于游刃必有余地矣。

——《庄子·养生主》

软件不是以实物的形式存在的，没有物理边界，因此软件的结构是逻辑结构。在软件的开发、部署和运行过程中，软件的组成形式会有所不同，描述软件结构的方式也会有所差异。同时，软件的结构还与软件的规模和范围密切相关。因此，在进行软件架构设计时，我们需要从多视角、多层次对软件结构进行描述。

2.1 软件的结构

一个物件的结构包括两方面的内容：物件由哪些部分（构件）组成，以及这些构件通过什么样的关系组合在一起，构成这个物件。物件的结构较为直观，因为存在可视的物理边界，构成物件的构件也具有自然可视的边界。然而，软件的结构则要复杂得多。软件由代码构成，不具备自然可视的物理边界，软件的边界是逻辑边界，结构则是代码的组成结构。而且，随着设计、部署和运行阶段的不同，软件的逻辑结构往往也会有所变化。因此，要了解软件的结构，需要从软件的边界、存在周期、规模与范围等方面入手。

2.1.1 开发边界、运行边界和部署边界

在讨论软件的边界时，通常指代以下三方面的内容：开发时的代码范围、软件的部署范围以及软件运行时涉及的范围。开发边界、部署边界和运行边界之间的不当搭配，可能会影响软件架构对软件结构的描述。反过来，在软件架构设计中，明确区分开发边界、部署边界和运行边界，可以使设计更清晰，更易于理解。

首先，需要明确的是软件的开发边界。一个项目若涉及多个开发人员或开发团队，就需要划清每个开发人员或团队的工作边界。最常用的办法是采用模块化开发，将开发工作拆解为可以独立交付的模块，每个独立模块都可以采用独立的代码库进行管理。

独立开发的组件或模块需要集成在一起才能够运行，这就需要在架构层面解决组件的集成关系。

通常有两种集成方式，一种是在设计阶段明确组件之间的依赖关系，高层组件调用低层组件，这就是“主-控”架构模式。该模式的特点是结构简单，但缺点是高层组件依赖低层组件，耦合性较高；另一种方式是引入接口隔离，采用依赖反转模式。在这种模式下，高层组件定义低层组件的调用接口，高层组件不直接调用低层组件，而是调用接口，低层组件实现这些接口。这种模式实现了组件之间的解耦，但缺点是结构复杂，需要在运行之前完成组件的装配，通常需要引入依赖注入技术。包含软件运行入口的模块是主模块，其他模块不能独立运行。

接着，需要明确软件的部署边界。软件的部署边界决定了软件的部署位置和部署组成：在何处部署哪些组件，需要进行何种配置，这些都是部署过程中需要解决的问题。软件的部署边界与开发边界息息相关，通常由具有运行入口的主模块和相关组件构成一个部署单元。这个部署单元可能部署在软件运行的主机上，也可能部署在远程主机上，并在运行时动态加载。

最后，我们需要明确软件的运行边界。在软件设计时，我们会假设软件的运行边界，并通过软件的部署边界来规定软件的运行边界。例如，在开发微服务架构的软件时，可以使用 Docker Compose 来编排微服务。在运行时，Docker 环境根据 Compose 文件创建容器并启动容器。从这一角度来看，软件的运行边界与我们的预期一致。然而，在许多情况下，软件的运行边界是动态的。软件运行时，软件与其设计时的外部环境构成一个有机的整体，软件可能通过与外部环境交互获取执行逻辑，这些逻辑与软件自身逻辑叠加，可能产生意想不到的结果。

在接下来的部分，我们将讨论 Log4j 的安全漏洞。在该安全漏洞中，Log4j、JNIN 和 RMI 服务构成了一个并非设计时预见的闭环架构。这个架构在软件的运行时存在，并形成了对应用软件的攻击闭环，如图 2-1 所示。

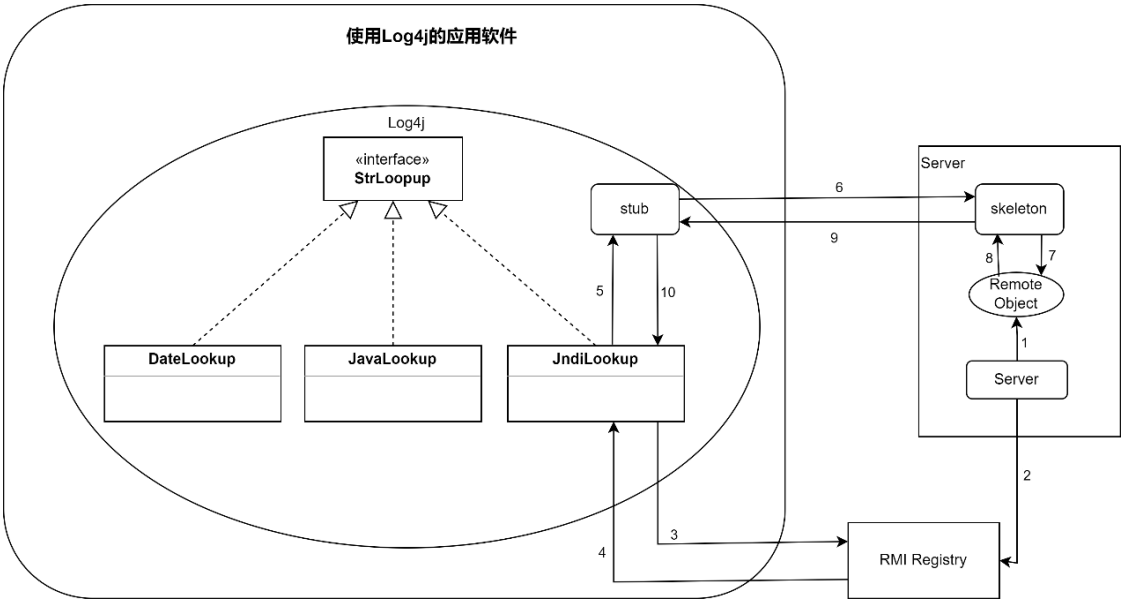

图 2-1　并非设计时预见的闭环架构，导致对应用软件的攻击

在架构设计中，首先需要明确开发边界和部署边界，这是设计阶段可以控制的内容；然后，应结合运行环境对软件的运行边界进行评估，并进行相应的测试，确保软件的运行边界可控。

2.1.2　架构的三种结构

在 2.1.1 节中，我们介绍了软件的三种边界，对应的则是软件的三种结构，即开发时结构、运行时结构和部署结构。这三种结构通常使用模块结构、构件及连接器（Component-and-Connector，C&C）结构和分配结构[8]来描述。

- 模块结构：描述系统的实现单元，即系统如何由代码或数据单元组成。模块代表了静态考虑系统的方式，通常与代码的实现方式相对应，包括包、类、层等，这些都可以视为模块的形式。模块结构通常使用 UML、ER 图等图形化建模语言来描述。
- 构件及连接器结构：聚焦于元素在运行时如何相互交互以执行系统功能。构件可以是服务、端点、客户机、服务器、过滤器或其他类型的运行时元素。连接器则是构件之间的通信媒介，例如调用返回、进程同步操作、管道等。这种结构通常使用“构件”和“连接器”来描述。
- 分配结构：建立从软件结构到系统的非软件结构（如相关组织、执行环境等）的映射。前面提到的部署结构就是分配结构的一种。分配结构通常使用 UML 部署图等进行描述。

这三种结构从不同侧面对软件进行描述，综合在一起，形成了完整的软件架构。

2.1.3　软件结构和软件边界的变化

最初，软件只是独立运行的程序，通过顺序执行的代码完成任务。当代码量逐渐增大时，程序变得越来越难以理解，修改和维护也变得更加困难。解决这一问题的一种办法是对代码进行归类，将关系密切的代码组织到一起，形成一个模块，进而将一个庞大的程序划分为若干相对独立的模块。每个模块内部的代码量会大幅减少，而模块之间的关系则可以比较清晰地描述出来。这种方法叫作模块化，是架构设计中的常用方法之一。模块化将软件从混沌状态转变为可描述的结构，完成了软件内部逻辑的划分，在开发时，所有逻辑模块的代码仍然在同一代码库中。软件在部署和执行时，仍然作为一个整体。

随着软件技术的发展，组件的概念应运而生。软件内部逻辑划分的模块可以独立编译为组件，组件可以独立开发和部署，并作为产品分发，供多个软件使用。然而，在运行时，组件仍需要作为软件的一部分被加载和调用，整个软件仍然从一个入口进入，并在一个进程中运行。这种类型的架构称为单体架构，根据内部逻辑的划分模式，可以分为插件架构、管道-过滤器架构和分层架构等。

网络的出现解决了不同主机之间的通信问题，使得运行在不同主机上的程序可以通过网络交换数据。软件的结构也从逻辑划分转变为物理划分。一个软件可以分解成不同的部分，部署在不同的主机上，通过网络协作来完成软件的功能。这种结构称为分布式架构，可以进一步细分为面向服务的架构、事件驱动架构和微服务架构等。

浏览器的出现扩展了软件在运行时的边界。传统的软件通常运行在其所部署的主机中，而使用现代浏览器作为客户端的应用，在软件运行时，客户端代码会被动态加载到浏览器中运行，从而使得运行时的软件边界远远超过了软件的部署边界。更极端的情况是，基于单页面的Serverless应用，甚至只需要在文件服务器上部署前端的静态代码，前端在用户使用应用时动态加载到浏览器中运行，而后端完全依赖开放的云计算平台，无须单独部署。

2.1.4 软件的范围和规模与软件架构

在第 1 章中提到，从不同的视角和不同的层次观察软件架构，会得到不同的结果。这是因为软件的规模与范围决定了软件架构所关注的内容。从范围和规模的角度，可以把软件分为三个不同层次：企业级应用软件、独立应用软件和软件组件。针对这三个不同的层次，软件架构有着不同的关注点和描述方式。

- 企业级应用软件架构：主要关注大型企业或组织的业务需求。它通常包含多个模块、子系统和服务，以支持复杂的业务流程、数据管理和用户交互。企业级应用软件架构的设计目标包括可集成性、可维护性、高可用性、安全性和性能优化。为实现这些设计目标，需要涵盖面向整个企业的业务框架、应用框架、数据框架和技术框架，并关注作为一个整体的标准规范设计、系统集成设计和技术底座设计。企业级应用软件架构主要面向整个企业的宏观架构。完成独立功能的应用软件在企业应用中需要与企业的这些框架有机集成，使用企业统一的数据平台和技术底座，同时这些应用软件具有自己的架构，属于独立应用软件架构。
- 独立应用软件架构：主要针对某一软件产品或软件开发项目，根据给定的业务需求和外部资源条件进行设计。独立应用软件架构设计关注软件的内部结构，软件架构在这个层次上可以使用架构模式或架构风格进行描述，例如分层架构、微内核架构、管道-过滤器架构以及微服务架构等。独立应用软件架构的落地需要更底层的架构设计作为支撑，这就是组件层次的架构设计，也称为微观架构设计。
- 组件软件架构：主要关注软件组件的设计和实现。软件组件是具有独立功能、可重用和可替换的软件单元。它们可以在不同的软件产品、系统或平台中重复使用，以提高开发效率和降低维护成本。组件架构的设计目标包括可重用性、可替换性、可维护性和可扩展性。软件组件通常需要通过特定的接口与其他组件或系统进行交互，是更大粒度软件架构的组成部分。

不同规模和范围的软件需要采用相应的软件架构描述方法。这些描述方法包括架构蓝图、构件和连接器组成的架构图，以及 UML 模型图等。

2.2 软件结构的描述方法

描述软件结构是软件架构的核心任务。由于软件是由代码组成的，如何将代码以直观的形式表现出来是必须解决的首要问题。在设计软件架构时，尤其需要注意这一点，因为此时组成软件的代码尚未产生，所以需要使用代表这些代码的抽象设计元素来描述软件的结构。

针对不同层次的架构，需要使用不同的描述方法。企业应用架构、独立应用软件以及软件内部的微观结构，均需要采用不同的描述方式。

2.2.1 架构蓝图

企业应用的架构蓝图是对企业应用整体结构和设计的可视化描述，展示了应用的主要组件、服

务、交互以及它们之间的关系。一个典型的企业应用架构蓝图可能包含以下几个关键部分：

- 业务架构：业务架构是整个架构的顶层设计，它从企业战略出发，定义了企业如何高效地创造价值。它涵盖业务领域、业务运营模式、组织结构管理以及关键业务流程体系的建设。
- 应用架构：应用架构将业务架构转化为所需的应用系统和服务。它定义了各个功能模块交互集成的方式以及它们之间的数据流。应用架构的目的是支持业务架构的运转，确保业务需求能够被有效实现。
- 数据架构：数据架构关注数据的结构、存储、管理和使用。它定义了数据的来源、格式、存储位置以及如何在应用中进行访问和使用。数据架构为应用架构提供数据支撑，确保数据的准确性和一致性。
- 技术架构：技术架构是指将应用和数据架构中定义的各种组件映射为相应的技术组件。它涵盖基础设施、硬件、操作系统、数据库、中间件、网络等技术层面的选择和设计。技术架构的目标是确保应用能够高效、稳定地运行，并支持业务的发展和变化。

架构蓝图中还包括企业应用需要遵循的标准和运营策略等内容，如图 2-2 所示。

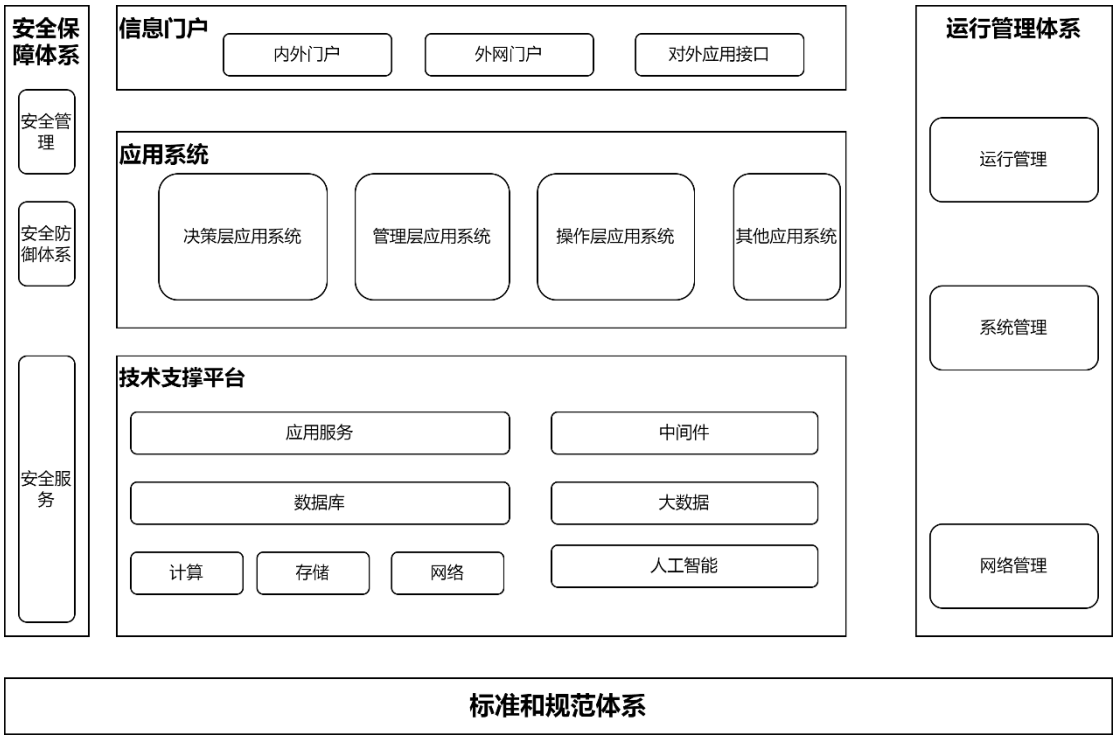

图 2-2　使用架构蓝图描述企业应用架构参考模型

图 2-2 展示了使用架构蓝图描述的企业应用架构参考模型，在本书的最后一章将进一步介绍这个模型。

2.2.2　“构件”和“连接器”

最常使用的软件架构描述方法是通过“构件”和“连接器”来描述软件架构，利用这两种元素

可以完成对软件运行时结构的描述。

构件代表软件系统中的独立单元，表示实现特定功能的代码和数据，构件具有与其他构件协同工作的接口。构件是一个高度抽象的概念，可以代表任何粒度的独立单元。在微观层面，构件可以是界面组件、实例、模块；在宏观层面，构件可以是服务、子系统。构件强调的是，在其描述的范围内，每个构件都具有明确定义的职责和边界。一个构件应该具有如下特点：

- 独立性：构件描述的应该是独立单元，而不是其他单元的一部分。
- 可重用性：构件描述的独立单元，只要满足条件，就可以在其他架构中使用。
- 可替换性：构件描述的独立单元，可以被其他构件替换。

连接器是软件架构中用于连接不同构件的机制。它定义了构件之间如何交互、传递数据和消息。连接器负责构件之间的协作关系，可以是函数调用、请求-应答、数据共享、消息传递等。

构件-连接器通常用于描述软件的运行时结构，具有简单明了、易于理解的特点，是架构设计中主要使用的描述方法。

2.2.3 图形化建模语言

如前文所述，使用构件-连接器的方式描述架构简单明了，但其缺点是缺乏表现实现细节的手段。通常，构件-连接器的描述方式较为粗线条，适用于顶层架构的概念设计。当需要指导架构落地实施时，在架构描述中需要展示更多的细节，此时构件-连接器的描述方式就显得力不从心。为了解决这一问题，需要引入更精确、更严密的描述方式。我们可以使用图形化的建模语言来描述面向实施的微观架构。

对于软件设计时的结构描述或更细节的运行时结构描述，可以采用多种标准的建模语言，如软件架构描述语言（Architecture Description Languages，ADL）、统一建模语言（Unified Modeling Language，UML[20]）、实体关系图（ER 图）和业务流程建模符号（Business Process Modeling Notation，BPMN[33]）等。标准的建模语言能够描述从微观到宏观的结构组成。下面简单介绍这几种语言。

1. 软件架构描述语言

类似于软件设计理论，架构设计理论也是在不断发展和变化的。有许多关于软件架构设计的理论，其中一些理论期望通过使用严格的语言来描述软件架构，从而规范架构设计，这种语言称为软件架构描述语言（ADL）。

ADL 的目的是提供清晰、精确且无歧义的软件架构描述，使架构师和开发人员能够使用一种共同的语言来沟通和理解系统的结构及设计决策。ADL 通过提供严格的语法和语义规则，使架构描述更加准确和一致，进而提高开发效率和软件质量。然而，ADL 并未被大多数开发人员所接受，原因在于 ADL 尚未形成统一标准，目前存在多种不同的 ADL，且它们之间的语法和语义差异较大，这增加了学习和使用的难度。同时，尽管 ADL 是完备的语言，它仍然无法直接转化为可执行的代码，转换过程需要人工完成。随着软件技术的迅猛发展，ADL 难以跟上变化，因此在开发项目时，我们不可能使用陈旧的描述语言来构建现代系统。

在实际项目中，软件架构设计应以需求为导向，在设计方法上要因地制宜，采用最适应的软件开发流程，而非追求理论上的标准化方法。如果开发团队熟悉某种 ADL，可在此基础上使用它对软件架构进行描述，但需注意与现代软件技术的衔接。

2. UML 及其扩展语言

我们可以使用 UML 等建模语言对软件架构进行描述。UML 包括以下基本图形：

- 类图（Class Diagram）：类图展示了系统中的类、接口以及它们之间的关系（如继承、实现、关联、聚合、组合等）。它有助于开发人员理解类的结构、属性、方法和它们之间的交互。
- 对象图（Object Diagram）：对象图是类图的一个实例，展示了类的对象实例（即对象）以及它们之间的关系。它可以在运行时的上下文中表示类的状态和行为。
- 时序图（Sequence Diagram）：时序图描述了对象之间的交互顺序，显示了消息如何在对象之间传递以及它们如何响应这些消息。这对于理解类的方法调用顺序和对象的协同工作非常有用。
- 活动图（Activity Diagram）：活动图用于描述系统中的业务流程和动作流。在微观层面，它可以用于描述一个类或对象内部的方法或操作的执行流程。
- 状态图（State Diagram）：状态图展示了类的对象在其生命周期中的不同状态，以及这些状态之间的转换。它有助于开发人员理解对象的内部行为，特别是在响应外部事件时。
- 组件图（Component Diagram）：虽然组件图通常用于描述宏观架构中的组件和它们之间的依赖关系，但在微观层面，也可用于展示更小的、可重用的软件组件（如库、框架、服务等）以及它们如何协同工作。

UML 可以描述各种粒度的系统架构，但一般来说，描述更贴近实现的微观架构更为适合。在 UML 的基础上，引入特定元素和语法，可以创建新的建模语言，SysML 就是一种扩展语言。在进行架构设计时，采用 UML 可以更精确地描述架构结构和解决方案。

3. ER 图

ER 图（Entity-Relationship Diagram，实体-关系图）是一种用于数据库设计的图形化表示方法，它展示了数据库中实体（Entity）、属性（Attribute）以及实体之间的关系（Relationship）。ER 图是软件设计过程中非常重要的工具，可以帮助设计师清晰地理解和描述数据模型的结构。

- 实体（Entity）：在 ER 图中，实体通常用一个矩形来表示，代表数据库中要存储信息的对象或事物，如人、地点、事件、物品等。每个实体都有一组属性来描述它的特征。例如，人的实体可能包含姓名、年龄、性别等属性。
- 属性（Attribute）：属性是描述实体特征的数据项，在 ER 图中通常不作为单独的图形元素表示，而是直接列在相应实体的矩形内部或旁边。每个属性都有一个名称和数据类型，例如姓名是字符串类型，年龄是整数类型。
- 关系（Relationship）：关系表示实体之间的联系，在 ER 图中通常用一个菱形来表示。关系可以是一对一（1:1）、一对多（1:N）、多对一（N:1）或多对多（M:N）。关系也可以有属性，用于描述关系本身的特征。例如，订单与顾客之间的关系可能包含一个日期属性，表示订单的下单日期。

4. BPMN 等业务建模语言

如果软件涉及大量的业务过程处理，可以使用 BPMN 等业务建模语言进行描述。

BPMN 是一种用于建模业务流程的标准化符号和语法。它由对象管理组织（Object Management Group，OMG）制定并发布的国际标准，用于描述业务流程的各个环节和活动。BPMN 的核心要素包括流对象（Flow Objects），如事件、活动和网关，这些元素共同构成了业务流程模型。

许多业务流程引擎支持 BPMN 语言。如果在架构设计时决定使用类似的引擎，那么在设计中优先使用 BPMN。

2.3 软件架构模式与软件架构风格

在第 1 章中提到，软件架构首先决定了软件的结构。反过来，将实践中构建的软件结构抽象出来并加以总结，就形成了若干有代表性的软件架构模型，这些模型可以作为新开发软件架构设计的基础。软件架构模式和软件架构风格是描述这些抽象架构模型的两种方式。

从作用上来看，软件架构模式[9]和软件架构风格[8]基本相同，但二者在描述方式和观察架构的角度上有所不同。软件架构模式采用模式化的方式进行描述，既包括描述软件体系结构的宏观架构模式，又包括描述软件实现的微观架构模式。而软件架构风格采用了独特的描述方式，主要关注描述软件的总体架构，基本不涉及实现层面的微观架构。

在本书的第 2 部分，我们将重点介绍软件架构模式与软件架构风格。

2.4 示例 1——Docker 的软件架构分析

Docker[21]是流行的应用容器引擎，允许开发者将应用及其依赖包以镜像的形式发布，并以容器的方式在安装有 Linux 操作系统的机器上运行。本节通过分析 Docker 的软件架构，说明如何描述从宏观到微观的软件结构。

2.4.1 Docker 的作用

首先，我们需要了解 Docker 的作用。应用软件在 Docker 环境下运行，不需要了解底层操作系统和基础设施的细节。各个应用软件之间具有独立的运行环境，互不干扰，从而提高了应用的可靠性、可扩展性和可维护性。图 2-3 描述了基础设施、操作系统、Docker 和应用之间的关系。

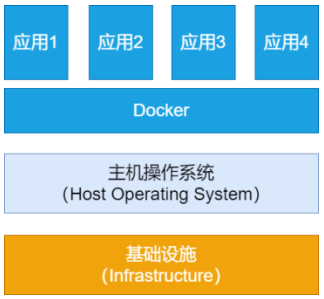

图 2-3 基础设施、操作系统、Docker 和应用之间的关系

Docker 的 Logo 形象化地表现了这种结构，如图 2-4 所示。

Logo 中的集装箱代表一个个应用软件，鲸鱼船则代表 Docker 平台和基础设施。

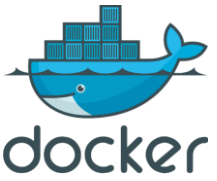

图 2-4　Docker 的 Logo

在上述架构描述中，Docker 是应用软件运行的整体环境的组成部分，但并未涉及 Docker 的内部结构。该架构的目的在于说明 Docker 的作用（是什么），而若要了解 Docker 的运作原理（如何做），则需要探索 Docker 本身的结构。

2.4.2　Docker 的顶层架构

Docker 的顶层架构，也可以称为宏观架构，属于典型的“客户机/服务器（C/S）”架构风格。图 2-5 是 Docker 架构的示意图，描述了 Docker 的基本结构和工作原理，属于概念层面的架构。

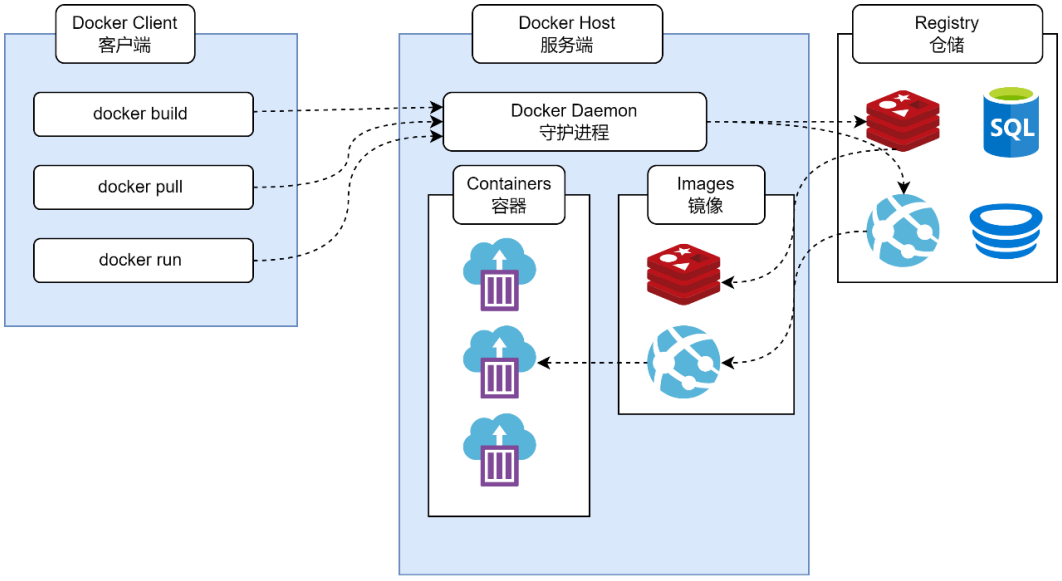

图 2-5　Docker 的概念架构

- Docker Host：Docker Host 是服务端，运行 Docker Daemon。Docker Daemon 侦听 Docker 客户端请求，这些请求包括 docker run、docker build 等命令。Docker Daemon 还管理诸如镜像、容器、网络等 Docker 对象。
- Docker Client：Docker Client 是客户端，发送 Docker 请求，这些请求由 Docker Daemon 完成。Docker Client（客户端）与 Docker Daemon（守护进程）通过交互，后者负责完成 Docker 容器的构建、运行以及销毁等功能。Docker Client 和 Docker Daemon 可以运行在同一个主机中，也可以分别运行在不同的主机中。Docker Client 和 Docker Daemon 可以通过 REST API、UNIX 套接字或网络接口进行交互。
- Docker Registries（Docker 镜像仓库）：Docker 镜像仓库是无状态的、具有高伸缩性的服务端应用，用于保存和发布 Docker 镜像。Docker 有很多公共的镜像仓库，如 Docker Hub、Docker Cloud 等。Docker Hub 是默认的镜像仓库，也可以创建私有镜像仓库。Docker Host

可以从镜像库中拉取镜像，也可以向镜像库提交镜像。

架构图中还包括容器、镜像等 Docker 对象。架构图中的虚线描述了镜像和容器的创建过程。当 Docker Client 发出 docker build 请求后，Docker Daemon 会根据 Dockerfile 中的定义，创建镜像并保存在 Docker Host 中。在创建镜像的过程中，Docker Daemon 会根据 Dockerfile 中的描述，从 Docker 镜像仓库中拉取需要的基础镜像，在此基础上构建新的镜像。当 Docker Client 发出 docker run 请求后，Docker Daemon 会在本地查找需要创建容器的镜像，如果在本地没有找到，则会向默认的 Docker 镜像仓库发出请求，使用镜像创建并运行容器。

顶层架构可以帮助我们理解 Docker 的基本结构和运行过程，如果需要进一步了解 Docker 的结构，则需要进行更深层次的架构分析。

2.4.3 顶层架构的展开

2.4.2 节展示了 Docker 的基本工作过程，对于 Docker 的使用者来说，这些信息已经基本够用。但对于希望了解其实现过程的开发者来说，需要更细节地描述。这时就需要进行更细粒度的架构描述，对 Dock Host 部分进一步展开，以更深入地揭示其内部结构，如图 2-6 所示。

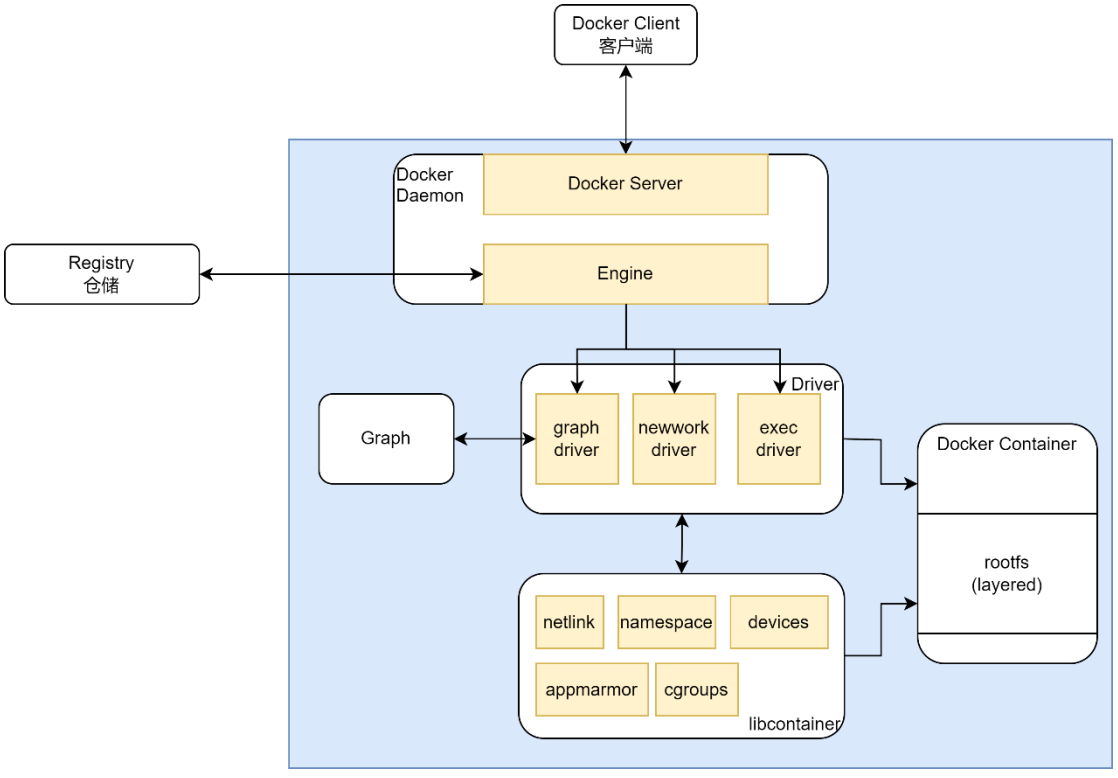

图 2-6 Docker 的总体架构

上述架构更详细地描述了 Docker 的结构，其工作过程如下：

（1）用户使用 Docker Client 与 Docker Daemon 建立通信，并向后者发送请求。

（2）Docker Daemon作为Docker架构中的核心部分，提供Server的功能，接受Docker Client的请求。

（3）Engine 执行 Docker 内部的一系列工作，每项工作都是以 Job 的形式存在。

（4）在 Job 运行过程中，当需要容器镜像时，从 Docker Registry 中下载镜像，并通过镜像管理驱动 graphdriver 将下载的镜像以 Graph 的形式存储。

（5）当需要为 Docker 创建网络环境时，通过网络管理驱动 networkdriver 创建并配置 Docker 容器的网络环境。

（6）当需要限制 Docker 容器的运行资源或执行用户指令等操作时，通过 execdriver 来完成。

（7）libcontainer 是一个独立的容器管理包，networkdriver 和 execdriver 都通过 libcontainer 来实现对容器的具体操作。

如果需要深入了解组件内部的工作过程，可以展开每个组件，分析各个组件的内部结构。

2.4.4　组件架构

Docker 的总体架构描述了各个组件之间的关系。要进一步理解 Docker 的工作原理，需要深入分析这些组件及其架构。

1. Docker Daemon 架构

Docker Daemon 是 Docker 的守护进程，负责接收来自 Docker Client 的命令，并执行相关操作。Docker Daemon 的架构如图 2-7 所示。

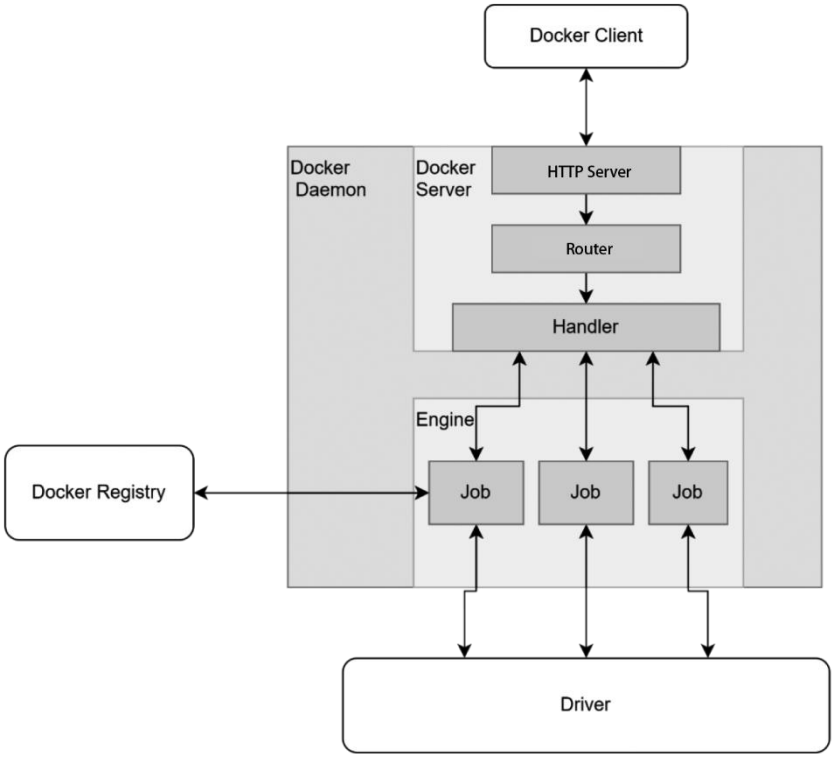

图 2-7　Docker Daemon 架构

Docker Server 相当于 C/S 架构中的服务端，主要功能是接收并调度分发来自 Docker Client 的请

求。在接收到请求后，Server 通过路由与调度机制，找到相应的 Handler 来执行请求。

Engine 是 Docker 架构中的运行引擎，也是 Docker 运行的核心模块。它作为 Docker Container 的存储仓库，通过执行 Job 的方式来管理和操作这些容器。Job 可以视为 Docker 架构中 Engine 内部的最基本工作执行单元。Docker 执行的每一项操作都可以抽象为一个 Job。

2. Graph 架构

Graph 是 Docker 的内部数据库，负责存储下载的镜像，包括 Repository 和 GraphDB 两个部分，如图 2-8 所示。

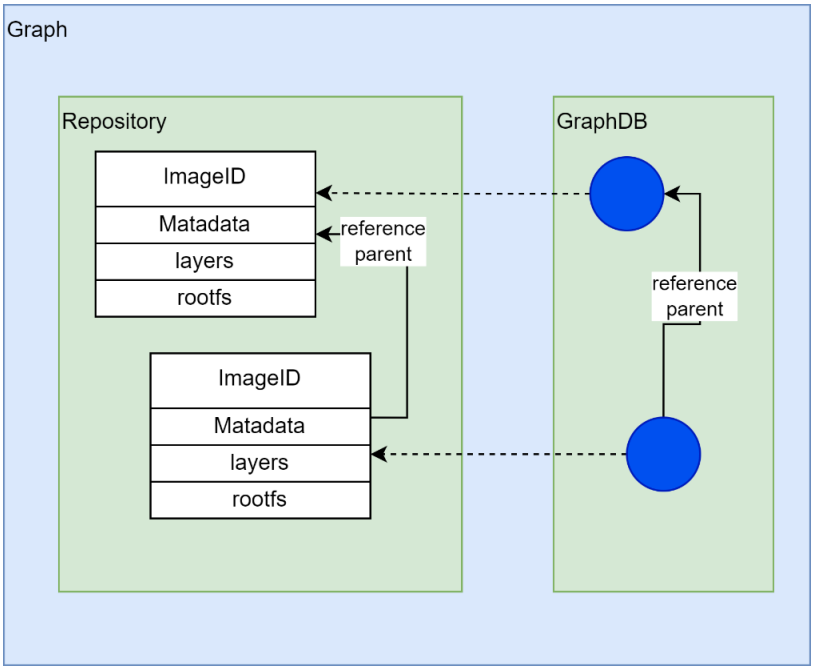

图 2-8　Docker Graph 架构

Repository 保管已下载的镜像和使用 Dockerfile 构建的镜像，每个镜像使用 tag 进行区分。

GraphDB 是一个构建在 SQLite 之上的小型图数据库，用于实现节点命名以及记录节点之间的关联关系。

3. Driver 架构

Driver 是 Docker 架构中的驱动模块。通过 Driver，Docker 能够定制 Docker 容器的执行环境。也就是说，Graph 负责镜像的存储，而 Driver 负责容器的执行。Driver 包括 graphdriver、networkdriver 和 execdriver 等。

graphdriver 主要用于管理容器镜像，包括存储与获取。其架构如图 2-9 所示。

docker pull 命令用于将下载的镜像从 graphdriver 存储到本地的指定目录（Graph 中）。当执行 docker run（或 create）命令用镜像来创建容器时，由 graphdriver 到本地 Graph 中获取镜像。

networkdriver 用于配置 Docker 容器的网络环境。其架构如图 2-10 所示。

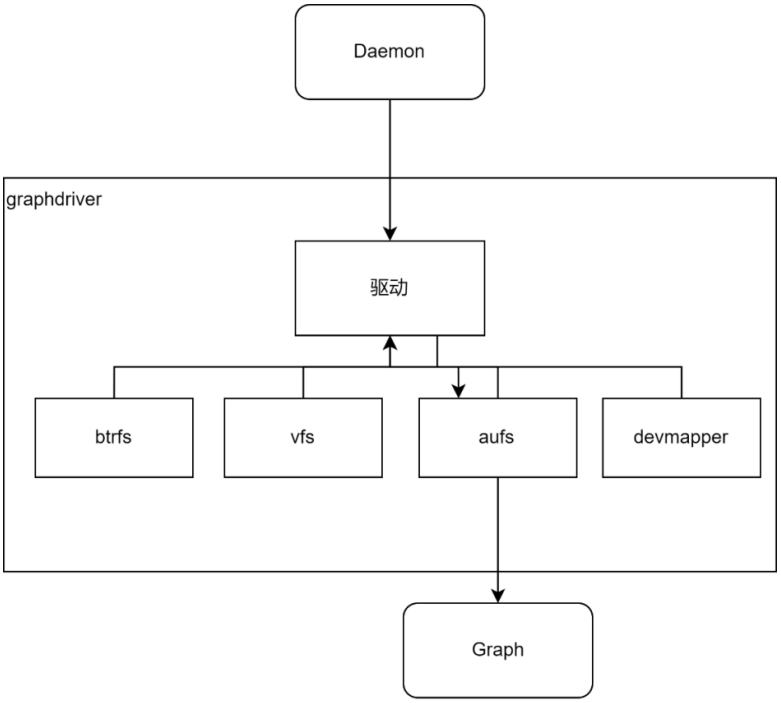

图 2-9　graphdriver 架构

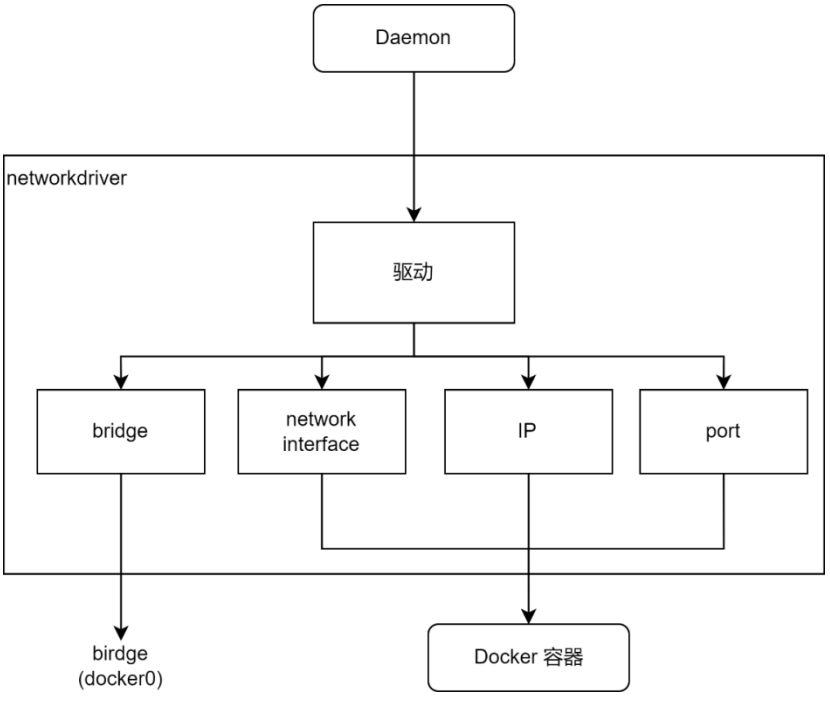

图 2-10　networkdriver 架构

networkdriver 在 Docker 启动时为 Docker 环境创建网桥；在 Docker 容器创建时为其创建专属的虚拟网卡设备；为 Docker 容器分配 IP 和端口，并与宿主机进行端口映射，设置容器的防火墙策略等。

execdriver 作为 Docker 容器的执行驱动，负责创建容器运行的命名空间，统计和限制容器资源的使用，也负责容器内部进程的实际运行等。其架构如图 2-11 所示。

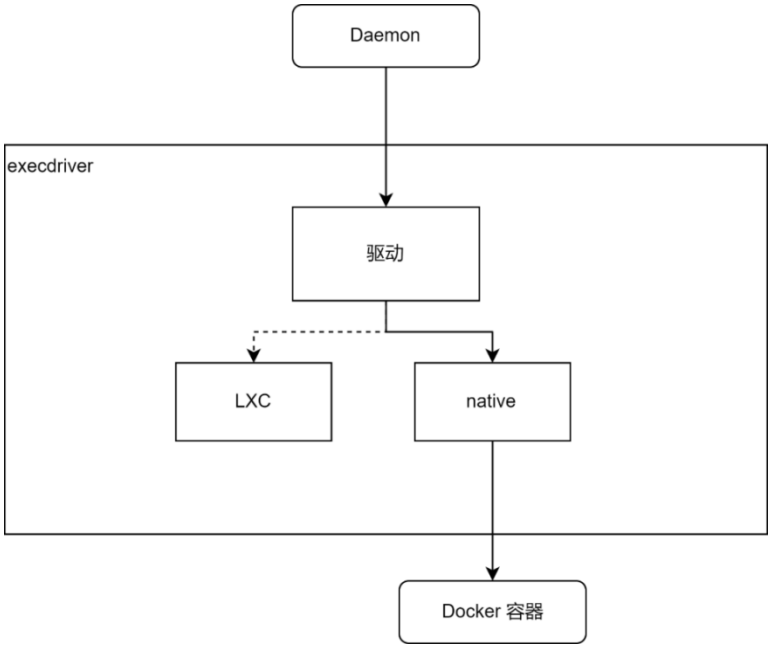

图 2-11　execdriver 架构

execdriver 负责存储容器的配置信息。实现这一功能有两种方式：一种是使用 LXC（Linux Container，一种轻量级的虚拟化技术），另一种是直接使用本地驱动。当前，execdriver 默认使用 native 驱动，而不再依赖于 LXC。

4. Libcontainer 架构

Libcontainer 负责与 Linux 内核交互，这部分封装了 Docker 容器实现的关键技术，具体内容将在第 3 章中介绍。

2.4.5　Docker 架构分析总结

在前文中，我们逐层分析了 Docker 的软件结构，这些层次包括：

（1）描述软件结构的作用：在这个层次中，Docker 是系统中的一个组件。

（2）描述软件整体运作过程和原理的结构：从这个层次可以看到，Docker 采用了 C/S 架构风格。

（3）Docker 服务端内部结构：这个层次说明了服务端内部的组成和工作原理。

（4）组件结构：说明了构成软件的各个组件的内部构成和工作过程。

以上 4 个层次从宏观到微观逐层描述了软件的结构。如果我们在开发类似 Docker 的软件时，必须在软件架构设计中包含这 4 个层次，才能实现完整的软件系统，缺一不可。我们必须关注软件结构从微观到宏观的各个方面，而不能仅仅依赖于顶层结构来替代整体的架构设计。

2.5　示例 2——设计时结构与运行时结构的关系

我们以一个简单的示例来说明软件设计时结构与运行时结构的关系，并探讨如何通过引入接口来改变模块之间的依赖关系。

在设计时，模块之间的依赖关系属于设计时依赖关系，是指某个模块或组件在设计时对其他模块或组件的依赖关系。如果依赖的组件不存在，就会出现编译错误，模块或组件无法正常构建。在设计时，组件通常只依赖某个接口，而不需要这个接口的具体实现即可通过编译，这就是为什么在编写程序时要遵守“依赖接口，不依赖实现”的原则。

在运行时，所有的组件和模块必须就位，程序才能正确运行。此时，不仅接口需要存在，接口的实现也必须就位。在运行时，接口及其实现需要装配在一起，以完成接口提供的服务。

接下来，我们通过一个简单的示例来说明这两种依赖与设计时结构和运行时结构的关系。

假设我们设计了两个类：GamePlayService 和 PoemService，其中 GamePlayService 使用了 PoemService 的某些功能。这两个类分别封装在两个不同的组件中，分别为 PoemGame.Domain 和 PoemService。如果不创建任何接口，GamePlayService 将直接调用 PoemService，从而形成依赖关系。这些关系可以用图 2-12 所示的 UML 模型来描述。

这种关系表明，在设计时 GamePlayService 依赖于 PoemService。而 GamePlayService 存在于程序集 PoemGame.Domain 中，由于前面两个类之间的依赖关系，导致组件之间也产生了依赖关系。PoemGame.Domain 依赖于 PoemService，如图 2-13 所示。

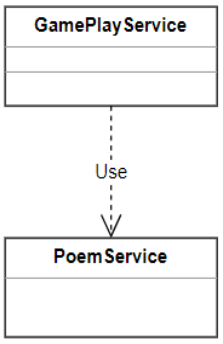

图 2-12　GamePlayService 直接调用 PoemService

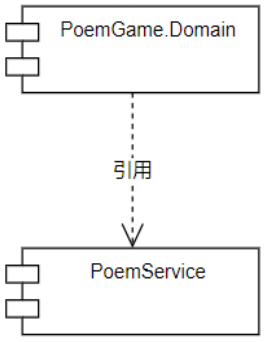

图 2-13　直接调用导致程序集之间的依赖

组件 PoemGame.Domain 需要引用 PoemService 才能编译通过，这就是设计时依赖。

在运行时，GamePlayerService 的实例（假设实例名称为 gamePlayService）将使用 PoemService 的实例（假设实例名称为 poemService），如图 2-14 所示。

也就是说，在运行时，gamePlayService 依赖于 poemService。在这种情况下，设计时依赖关系与运行时依赖关系是一致的，设计时结构与运行时结构也是一致的。

现在，我们希望 PoemGame.Domain 能够独立发布，而不需要引用其他组件或类库，因此需要对当前结构进行改造。

首先，将 PoemService 中 GamePlayService 调用的功能抽象为接口，命名为 IPoemService。这个接口在 PoemGame.Domain 中定义。这样，GamePlayerService 就可以依赖这个接口，而不再依赖 PoemService，如图 2-15 所示。

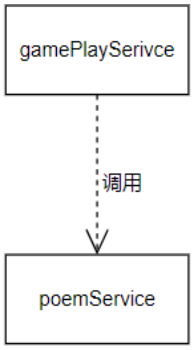

图 2-14　直接调用的运行时依赖

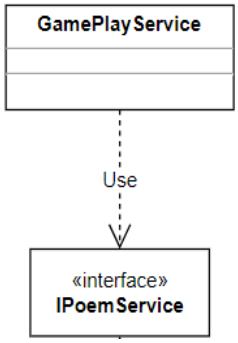

图 2-15　GamePlayService 依赖于接口而不是实现

在移除对 PoemService 的依赖后，即使没有 PoemService，PoemGame.Domain 依然可以正常编译和发布。然而，由于 PoemService 需要实现 IPoemService 接口，它反而需要依赖 PoemGame.Domain，如图 2-16 所示。这表明依赖关系发生了反转。

同时，程序集之间的依赖关系也发生了变化，如图 2-17 所示。

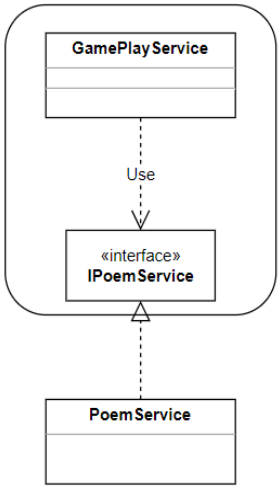

图 2-16　接口使依赖关系出现反转

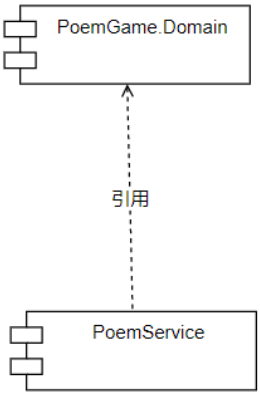

图 2-17　程序集之间的依赖关系也发生了变化

可以看出，程序集之间的依赖关系也发生了反转。在设计阶段，由于接口的引入，依赖关系发生了倒置。这就是依赖反转原则（Dependency Inversion Principle，DIP），相关内容将在第 5 章进行详细介绍。

那么在运行时呢？在实际运行时，GamePlayService 的实例（假设为 gamePlayService）需要调用 IPoemService 接口的某个实现实例（这里假设为 poemService）。也就是说，在运行时，gamePlayService 仍然依赖于 poemService，如图 2-18 所示。

现在的问题是，在运行时，gamePlayService、IPoemServie 和 poemService 之间需要进行组装，以便 gamePlayService 能够正确访问 poemService。这个在装配过程图中没有显示，应该由

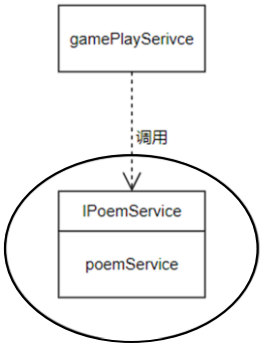

图 2-18　运行期的依赖关系

创建 gamePlayService 的第三方负责。

如果将前面几幅图拼接在一起，可以看出引入接口产生的依赖倒置过程，如图 2-19 所示。

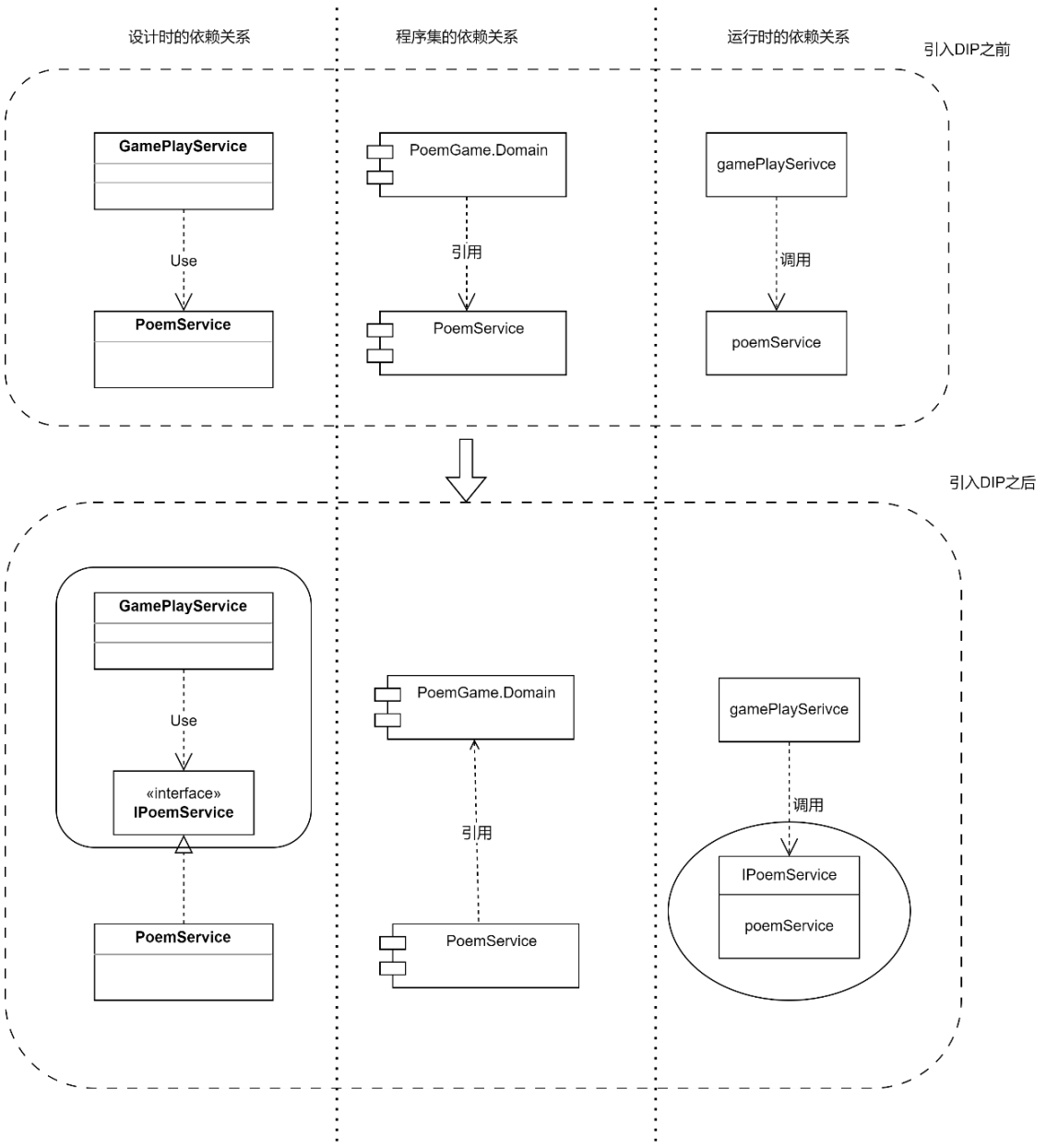

图 2-19　DIP 引入前后的比较（设计时、运行时和程序集）

使用 DIP（依赖反转原则）设计的结构，在设计时和运行时的依赖关系是不一样的，软件的结构也因此有所不同。

现在，简单总结一下，通过依赖反转改变依赖关系和软件结构的步骤如下：

步骤 01　存在互相依赖的两个类之间，根据被依赖方的方法抽象出一个接口。

步骤02 依赖方由依赖原来的类，修改为依赖新创建的接口，并通过构造函数传入新创建接口的实例。

步骤03 原来的被依赖方实现新创建的接口。

步骤04 由第三方负责进行组装。

如果要设计新的系统，步骤基本一样，只是可以先设计接口：

步骤01 为需要的功能设计一个接口。

步骤02 使用这个接口的类从构造函数中传入接口的实例。

步骤03 实现该接口的类可以在独立的程序集中。

步骤04 由第三方负责进行组装。

引入接口的另一个好处是开发团队可以解耦：使用接口和实现接口的部分可以独立开发，独立测试。

本示例说明，在进行架构设计时必须同时关注软件的设计结构、部署结构和运行结构，只有这样才能完整地描述软件的架构。

2.6 本章小结

软件架构首先描述了软件的结构。不同规模和范围的软件，其架构描述方式有所不同。对于大尺度的企业级应用软件，可以采用架构蓝图来描述其宏观结构；而对于一般的应用软件，则可以使用“构件”和“连接器”来描述其整体结构。在更细粒度的微观结构层面，可以采用 UML 等图形化建模语言进行详细描述。

通过对软件结构的抽象和总结，形成了软件架构模式和软件架构风格。从架构模式或架构风格出发进行软件架构设计，可以充分利用前人积累的经验，从而提高设计效率，使架构设计更易理解，也便于交流。

无论是分析现有软件的架构，还是设计全新软件的架构，都需要从顶层到底层逐层分析和设计。本章以 Docker 架构为例进行了说明。首先，需要了解 Docker 作为运行环境的架构，这一架构说明了 Docker 的作用以及它与其他基础设施的关系。接着，对 Docker 的整体架构进行分析，从整体架构来看，Docker 属于典型的客户端/服务器架构风格。然后，进一步展开整体架构，使用更细粒度的组件来描述 Docker 的内部结构。最后，分析构成 Docker 的各个组件的内部结构。

本章对 Docker 的分析过程可应用于其他软件的架构分析，也可以在架构设计过程中作为参考借鉴。

第 3 章

关键技术、支撑技术与技术路线

“你们所说的……主，为什么这样害怕纳米材料呢？”汪淼问。

“因为它能够使人类摆脱地球引力，大规模进入太空。”

“太空电梯？”汪淼立刻想到了。

“是的，那种超高强度的材料一旦能够大规模生产，建设从地表直达地球同步轨道的太空电梯就有了技术基础。对主而言，这只是一项很小的发明，但对地球人类却意义重大。地球人类可以凭借这项技术轻易地进入近地空间，在太空建立起大规模的防御体系便成为可能，所以，必须扑灭这项技术。”

——《三体》刘慈欣

关键技术的确定和技术路线的制订对软件开发过程至关重要。通常在项目规划阶段，就需要确定项目的关键技术和技术路线。

技术路线决定了实现软件所需使用的技术，这些技术的选择不仅是软件架构设计的基础，也是软件架构设计的约束。如果在开发过程中引入架构设计规定以外的技术，将大幅增加软件架构的复杂度，从而增加软件开发的成本和难度。

3.1 关键技术

关键技术是软件中不可替代的技术，离开了关键技术，软件就无法实现。

3.1.1 什么是关键技术

关键技术是指在一个系统、一个环节或一个技术领域中，占据核心地位且不可或缺的技术或环节。关键技术包括软件的核心算法和模型，或者其依赖的外部技术等。

1. 算法

对于需要完成某种计算的软件来说，算法就是这个软件的关键技术。需要注意，“计算”是一个很宽泛的概念，通常需要一定的抽象才能将某种需求转换为计算模型。有些场景中，“计算”需

求显而易见：例如，在油品储存运输系统中，需要计算油罐的储量；在财务系统中，需要根据税率计算费用等等。而大多数应用场景中，计算需求是隐性的，需要通过分析将软件需要解决的问题转换为算法。

举个例子，如果我们希望开发一款简单的人与人对战的围棋游戏，关键技术是什么呢？人机交互、网络通信等都有现成的解决方案，这些都不是这款游戏的关键技术。可以这样思考，围棋游戏和五子棋游戏的最大区别是什么？就是游戏的规则不同，这些规则包括落子合法性判断、游戏结束判断、输赢判断，还有围棋特有的吃子判断等。这些规则需要抽象为算法实现，这些算法就是围棋游戏的关键技术。

2. 业务模型

业务模型用于描述软件针对的业务。对于以业务处理为主的软件来说，业务模型就是关键技术，如大多数的管理信息类软件。如果比较“人力资源管理系统”和“设备信息管理系统”的软件架构和技术实现，会发现，除数据库结构不同外，涉及的软件实现方法和使用的软件技术几乎完全相同。数据库结构正是业务模型的一种表现形式。

业务模型在软件中有多种存在方式。在面向对象技术出现之前，业务模型通常以数据模型的形式出现，使用“实体-关系”模型进行描述。“实体-关系”模型可以转换为关系数据库的结构。数据库结构成为软件开发的抓手，在软件设计时，先设计数据模型，再根据数据模型生成数据库结构，再由数据库结构驱动软件代码开发，这就是数据驱动的开发方法。

数据模型的缺点是无法描述业务的动态特性。为了解决这个问题，产生了针对数据模型的多种业务模型的实现方式，包括表模式、活动记录等。对于复杂业务，需要比数据模型更加复杂的模型来描述，领域模型就是其中一种。领域模型既可以描述业务实体的属性，也可以描述业务实体的行为，根据业务实体的不同性质，还引入了更多的描述方法，如实体、值对象、聚合根、领域服务、领域事件等。领域模型使用存储库模式实现持久化，存储库采用接口和实现分离原则进行设计，使领域模型摆脱了对持久化框架的依赖。

需要注意的是，业务模型的实现方式大多需要特定的软件框架作为支撑。在实际项目中，业务模型实现方式的选择往往是被动的：当选定了某种软件框架时，业务模型的实现方式也就被选定了。例如，如果确定使用 Django 作为开发的基础框架，那么就需要使用活动记录作为业务模型的实现方式，因为 Django 框架基于活动记录模式，并提供了活动记录的完整实现。

3. 软件依赖的底层技术

很多软件的关键技术取决于其所依赖的类库、框架、组件等提供的功能。当所使用的技术无法使用其他技术进行替代时，这种技术就是关键技术。

我们以页面的“所见即所得（WYSIWYG）”编辑为例来说明关键技术。“所见即所得”的编辑页面要求页面上的元素可以通过拖动来完成布局，元素中包括基础元素和容器元素，基础元素可以包含在容器元素中，容器元素可以嵌套。这是典型的“组合设计模式”，也是这种页面的基础模型。因此，实现这种类型的页面的关键技术就是底层的技术框架需要支持基于“组合设计模式”的页面组件结构。反过来说，如果使用的底层技术框架不支持这种结构，那么如果不借助其他技术，就没有办法实现“所见即所得”的编辑方式。

DNN Plat（DotNetNuke）的发展和衰落与“所见即所得”的编辑技术密切相关。DotNetNuke 曾

经是风靡一时的CMS 应用框架，基于 ASP.NET 的 Web Form 技术进行设计，“所见即所得（WYSIWYG）”的编辑界面是其重要特色之一，最终用户可以通过鼠标拖拽实现 Web 页面布局。然而，随着 Web Form 技术不再发展，MVC 等技术成为主流，问题出现了：MVC 等流式输出不再支持基于组合模式的服务端组件，在这些新的框架下，不存在 DotNetNuke 所需要的技术基础，导致其无法向新的技术平滑迁移，只能在现有技术上维持，结果是社区逐渐萎缩，失去了继续发展的动力。

3.1.2　关键技术的确定与识别

任何软件都具有属于自身的关键技术，而关键技术具有不可替代性（或者替代成本过高）。因此，关键技术的定义与识别变得非常重要。如果某项技术没有被识别为关键技术，但在实现时却严重依赖这项技术，就会有隐含的风险。因此，关键技术的确定与识别需要作为软件架构设计的一个步骤加以管理。

需要注意的是，对于具体项目而言，虽然某种技术具有可替代性，但若替换成本过高，这项技术仍然可以被视为关键技术，尽管在设计初期未被主动标识。这种情况在实际项目中经常发生，尤其当软件项目使用的是重量级框架、库或组件时，便容易出现这种情况，我们称之为被动关键技术。

避免被动关键技术的主要方法是加强微观层面的架构设计。可以采用某种设计模式，降低对框架的依赖性，将被动关键技术降级为支撑技术。

关系数据库经常成为项目的关键技术之一。在项目之初，我们往往未能意识到这一点。随着项目的进行，经常会用到某种关系数据库的特定功能，而这些功能在其他类型的关系数据库中要么不存在，要么实现方式完全不同。例如，Oracle 数据库提供了强大的 PL/SQL 语言，能够编写复杂的逻辑。当项目中需要使用 PL/SQL 编写关键业务逻辑时，Oracle 数据库便成为该项目的关键技术。

3.1.3　关键技术的验证

关键技术需要在软件正式开发之前进行验证。尽管涉及关键技术的开发量可能只占软件整体开发量的很小一部分，但其重要性远超软件的其他部分。反之，如果关键技术未经过验证就在项目中使用，而到开发后期才发现不可行，整个项目就会面临失败的风险。

验证关键技术的最佳方法是创建系统原型，通过系统原型完成关键技术涉及的功能并进行测试。系统原型可以是简单的控制台应用，亦或是一组单元测试，所产生的代码不是抛弃型的，而是在后续开发迭代中继续使用。在确定技术路线和进行结构设计之前，必须完成关键技术的验证。

3.2　支撑技术

软件的实现一定基于具体的编程语言和开发环境，架构设计必须依赖相关的软件技术才能落地，这些技术便是支撑技术。

3.2.1　软件架构落地需要特定的软件技术作为支撑

软件架构设计并非凭空进行，而是以软件技术为基础。例如，如果项目限定了使用面向过程的

编程语言（如 C 语言或者 Fortran）进行开发，那么就无法在该项目中采用面向对象技术。如果我们在架构设计中使用了某种方法、范式或模型，就需要确保我们的选择在技术上易于实现。例如，若选择领域驱动设计为主要开发方法，则架构设计要求领域层与基础设施层解耦，这便需要支持存储库模式的技术。如果所使用的编程环境或框架没有成熟的解决方案，架构设计将很难落地。

2006 年，笔者承担了一个大型项目的设计工作。当时，领域驱动设计刚刚兴起。笔者被这一新兴开发方式所吸引，并决定在项目初始设计阶段尝试在一个子系统中使用这种方式进行开发：先建立领域模型，然后使用存储库模式完成持久化。建模过程顺利，使用内存存储库也可以完成模拟测试工作，但在编写针对关系数据库的存储库时遇到了麻烦。当时使用的技术是.NET Framework 2.0，而.NET 当时提供的数据库访问技术是 ADO.NET，基于数据集和数据表（DataSet 和 DataTable）完成持久化并与数据库交互。由于没有 Entity Framework 等 ORM 框架，我们需要自己编写对象的“开箱”和“装箱”代码。起初，这似乎不是什么大问题，但随着编写工作的进行，发现没有那么简单。其中一个问题是信息隐藏，如果希望保护实体的属性，便不能暴露这个属性的 set 方法，但当从数据库恢复实体时，又需要使用 set 方法为属性赋值。虽然可以使用反射技术，但这显著增加了技术复杂度，导致开发人员的注意力从业务逻辑转移到对特定技术细节的研究上，这与项目的整体要求不符。最终，我们决定采用当时的主流技术来完成项目，领域驱动设计作为阶段试验被暂停了。

在进行软件架构设计时，必须明确所需的软件技术是现实可行的，这些技术构成了项目的支撑技术。

3.2.2 软件技术对软件架构设计的刚性约束

如果用户因某些原因已选定某项技术，这项技术可能会成为架构设计的约束。在某些情况下，整个架构设计甚至会围绕这项技术展开。

20 世纪 90 年代中期，随着局域网的广泛使用和办公自动化需求的增加，Lotus Notes 协同办公平台开始流行。许多企业和组织开始使用该平台快速构建自己的办公自动化系统。那时，许多项目要求要么在这种平台上开发，要么具备与其数据集成的能力。然而，Lotus Notes 的服务器基于一种文档型的非结构化数据库（Domino），它的开放性较差，且与其他系统的集成在可靠性和性能方面往往难以保障。因此，在很多情况下，技术路线只能选择在 Notes 环境下进行二次开发，软件架构也因此受到限制，只能使用 Notes/Domino 平台，无法使用其他架构。

当已存在的软件技术是重量级的框架或产品时，它们会对架构设计产生刚性约束，例如企业级的工作流引擎、基于特定企业服务总线（Enterprise Service Bus，ESB）的 SOA 架构等。因此，必须重视这些已选定技术对架构设计的影响，将它们作为架构设计基础技术的一部分，而不是仅仅作为后期的集成对象。

3.2.3 软件架构设计与软件技术选择

在实际项目中，缺乏软件技术支撑，架构设计很容易沦为空谈。因此，在软件架构设计的初期，就必须确定所需的支撑技术。技术的选择一旦确定，也就为软件架构设计增添了约束。

有些软件技术是客户方的硬性要求，例如采用的运行平台、数据库的类型、权限认证方式等。另一些则是我们在软件架构设计时需要使用的技术。如果这些技术需要用户投入额外的资源，则必须在设计初期明确提出，并与用户协商。

如果我们的目标是开发一个软件产品，那么希望对具体的软件技术的依赖尽可能少。在这种情况下，需要明确支撑软件技术的类型，例如，是否需要同时支持关系数据库与非关系数据库，是否需要配备特定的消息中间件等。

下面是一些常见的支撑技术，许多架构设计中都会用到这些技术。

- 与分布式系统相关的技术：如果软件需要处理大量数据和高并发请求，在架构设计时需要考虑使用分布式系统。与分布式系统相关的技术包括负载均衡、服务降级、失效转移、超时重试等，这些技术可以提高系统的稳定性和可用性。
- 与微服务相关的技术：对于业务复杂的应用，将不同的业务领域划分为不同的微服务，可以获得良好的可扩展性和可维护性。与微服务相关的技术包括容器技术、服务发现与注册等。
- 与数据库和数据存储相关的技术：业务数据的持久化离不开数据库和数据存储设计。合理的数据存储设计对于确保系统性能和数据安全性至关重要。数据库相关的技术包括数据模型的选择（关系型、非关系型、图数据库等）、数据一致性、事务处理、索引优化等。
- 与缓存相关的技术：缓存技术是提高系统性能的关键手段之一，包括应用缓存、HTTP 缓存、分布式缓存等。通过缓存频繁访问的数据，可以减少数据库的访问次数，从而降低系统响应时间。
- 异步与并发处理技术：异步与并发处理技术对于处理大量用户请求、提高系统吞吐量和响应速度非常重要。这包括多线程编程、异步任务处理、消息队列等技术。
- 安全性设计：确保软件系统的安全性是架构设计中的关键任务之一。这包括数据加密、用户身份验证、访问控制、防止 SQL 注入、XSS 攻击和 CSRF 攻击等安全性措施。

在进行架构设计时，可以使用上述列表作为检查单，确保架构中涉及的技术是否已经得到充分考虑并解决。

3.3　技术路线

如果说关键技术解决的是“点”的问题（关键点），则技术路线解决的是“路径”的问题（完整的技术栈和实现步骤）。

3.3.1　什么是技术路线

技术路线是指为实现软件目标而采取的技术手段、具体步骤以及解决关键性问题的方法等所形成的路径。

技术路线的制定是一个渐进且迭代的过程，通常贯穿架构设计的全过程。在项目还处于蓝图阶段时，技术路线的工作内容主要是确定所使用的技术范围。例如，在项目设计初期，我们可以确定采用前后端分离的策略，前端采用基于响应式编程的单页面应用，后端通过 RESTful API 提供服务，并采用微服务架构。此时的设计属于概念性设计，为后续设计指出大致的方向。接下来，我们需要为前后端选择具体的技术，逐步将概念设计转化为具体的概要设计。技术路线的描述类似素描绘画：先画出大致的轮廓，在此基础上逐步细化，每一步都为下一步提供支持。

3.3.2 确定技术路线时需要考虑的因素

在确定技术路线时，需要综合考虑环境现状、团队能力、行业发展趋势和可能的风险等因素。

首先，要考虑用户现有的环境，这是技术路线选择的刚性要求。用户环境包括当前使用的网络、数据库、桌面系统等。例如，如果用户已经在使用某种类型的关系数据库系统，那么在选择数据库时，应优先考虑与之兼容的数据库，这有助于降低后续数据库维护的总体成本。

同时，还需考虑开发团队的能力，并结合项目的实际情况选择团队熟悉的技术。如今，成熟的技术栈能够覆盖从后端到前端的各种应用，每种技术在不同的技术栈中都有对应的框架，技术选择通常取决于团队的技术偏好。以 Web 开发为例，.NET 有 ASP.NET Core 框架，Java 有 Spring MVC 框架，Python 有 Django 框架，Node.js 有 Nest.js 和 egg.js 框架等，这些框架都能够有效支持 Web 应用的开发。

此外，在技术路线选择方面，还需要关注行业内的技术发展趋势和最佳实践，尽量选择符合行业发展的技术栈，避免选择已过时的技术。例如，在前端开发中，虽然 jQuery 曾是前端开发中流行的库，依然可以胜任大部分前端功能的开发，但它已不再符合前端技术的发展趋势，因此需要逐步引入 Vue.js 等新兴技术。

3.3.3 技术路线与架构设计落地

架构设计落地需要选择具体的技术实现方式，这可能是某种软件框架，也可能是某种第三方产品，或者是自行开发的组件。一方面，从架构的角度，我们应尽量将技术与具体实现解耦，避免过度依赖这些实现；另一方面，需要选择符合技术需求的产品，确保软件项目的顺利完成。

1. 从架构设计上减少框架依赖

在软件结构设计时，架构设计应确保减少对实现技术（框架、库或组件）的依赖。这不仅有助于提升可维护性，还为未来替换当前技术实现提供了可能，更是为实现可测试性提供了保障。后者在当前项目开发中更具有现实意义。

为了实现这一目的，需要遵守架构设计的一般原则，为每种技术定义访问接口，使构成软件的各个组件只依赖这些接口，而不是具体的技术产品或技术实现。第 5 章将详细介绍架构设计的原则。

2. 选择合适的技术产品

接下来，需要为每种支撑技术选择适合项目的产品，这些产品既可能是商业软件，也可能是开源项目。在选择技术产品时，需要考虑技术因素和非技术因素，如成本、合规性等。

首先要考虑技术因素，产品必须满足架构设计的技术需求，包括功能需求和质量属性需求（如安全性、性能等）。

非技术因素同样不可忽视。当选择开源软件时，合规性是首先需要考虑的因素，必须了解所选择产品的使用许可，并确保该使用许可与正在开发的软件的商业目标没有冲突。成本也是重要的考虑因素之一，这不仅仅是指购买商业软件许可的费用，还包括使用软件所需的培训等其他成本。

3. 充分了解所使用的技术和产品

当确定了所使用的软件技术后，首先需要对这些技术有充分的了解。要了解这些技术能做什么，

更重要的是了解它们不能做什么。

对软件技术的初步了解最好的办法是实践。编写一些针对特定场景的简单程序进行试验，可以帮助我们迅速找到“感觉”。

如果要进一步了解软件技术，就需要系统地阅读这项技术的说明文档。编程实践虽然能够帮助我们快速熟悉软件技术的基本功能，但往往缺乏系统性。完整阅读说明文档是编程实践的必要补充。

最后，在实践中使用这项技术时，需要加入有关社区，与其他使用者进行交流。在必要时，还可以购买相关服务，请专业人员帮助解决特定问题。

3.4　关键技术和支撑技术的区别和联系

在技术路线中包括多种支持软件架构设计落地的支撑技术，那么这些技术是否就是关键技术？

关键技术最重要的特性是其不可替代性，也就是说，如果缺乏关键技术，软件就无法实现特定的功能。而技术路线中的支撑技术大多数是可替换的，正因为它们是可替换的，所以在制定技术路线时需要进行选择。例如，前端框架的选择既可以选择 React，也可以选择 Vue 或 Angular。如果选择 Vue 作为前端框架，那么可以选择 Element Plus 或 iView 作为前端组件库。在制定技术路线时，选择哪种具体的技术，往往不取决于技术本身，而是取决于其他非技术因素，例如用户的要求、团队的开发能力等。

3.5　示例 1——Docker 的关键技术

在第 2 章的实例中，我们介绍了 Docker 的架构，侧重于结构介绍。然而，Docker 之所以能够实现容器间的隔离，其所依赖的关键技术在结构图中无法明确描述出来，因为这些关键技术是由 Linux 提供的，是 Docker 依赖的，而非 Docker 所实现的。

3.5.1　Docker 关键技术概述

Docker 依赖的关键技术有三个部分：Namespace、Controlled groups 和 Rootfs，如图 3-1 所示。

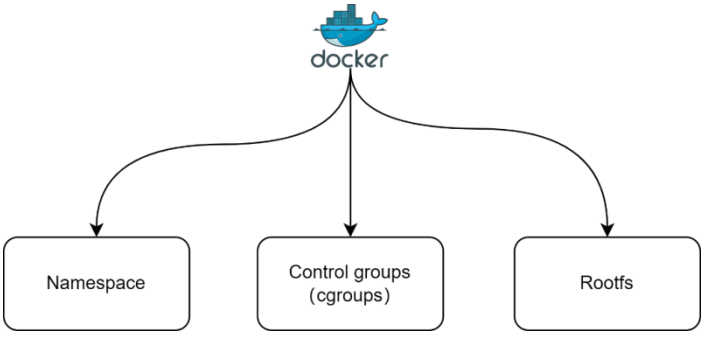

图 3-1　Docker 所依赖的关键技术

这些技术由 Linux 内核提供，能够确保创建的容器可以隔离运行。它们不可替代，属于 Docker 的关键技术。

1. Namespace

Namespace 技术是一种内核级别的特性，它允许将全局系统资源隔离成独立的视图，使得在不同 Namespace 中运行的进程看到的资源是不同的。这为容器化技术提供了基础，使得多个进程或容器可以在同一台主机上独立运行而不会相互干扰。

Namespace 主要有以下几种类型。

- Mount Namespace：用于隔离文件系统的挂载点，不同的 Mount Namespace 拥有各自独立的挂载点信息。
- UTS Namespace：用于隔离系统的主机名、Hostname 和 NIS 域名。
- IPC Namespace：使容器内的所有进程之间的数据传输、共享数据、通知以及资源共享等操作仅限于所属容器内部，不会对宿主机或其他容器产生干扰。
- PID Namespace：用于隔离进程的 ID 空间，使得不同容器中的进程 ID 可以重复，互不影响。
- Network Namespace：用于隔离网络，每个 Namespace 可以拥有自己独立的网络栈、路由表、防火墙规则等。

2. Cgroups

Cgroups 是 Linux 内核提供的一种机制，用于限制、记录和隔离进程组（Process Groups）所使用的物理资源（如 CPU、内存、磁盘 I/O 等）。通过 Cgroups，系统管理员可以更有效地管理系统资源，提高系统稳定性，并防止某个进程组占用过多资源。

Cgroups 包括以下几个主要部分：

- Cgroup 本身：用于对进程进行分组。
- Hierarchy：将 Cgroup 形成树形结构。
- Subsystem：真正起到限制作用的部件，不同的 Subsystem 可以限制不同类型的资源，如 CPU、内存、磁盘 I/O 等。

3. Rootfs

Rootfs 是容器的根目录，挂载一个完整的文件系统，为容器提供隔离后的执行环境，即容器镜像。容器镜像中包括各种目录和文件，如 bin、dev、etc、home、lib 等。

Rootfs 只包括操作系统的文件和目录，并不包含内核。通过 Mount Namespace 和 Rootfs，可以构建出一个完善的文件系统隔离环境，使得容器内的进程只能看到和操作属于自己的文件系统，而无法访问或修改宿主机的文件系统。

3.5.2 关键技术在架构中的位置

Docker 通过使用 Namespace、Controlled Groups 和 Rootfs 实现容器的创建，这些关键技术存在于 Linux 内核中。在具体实现上，Docker 引擎并不直接调用 Linux 内核，而是通过封装的 Libcontainer

库来完成这个工作。其架构如图 3-2 所示。

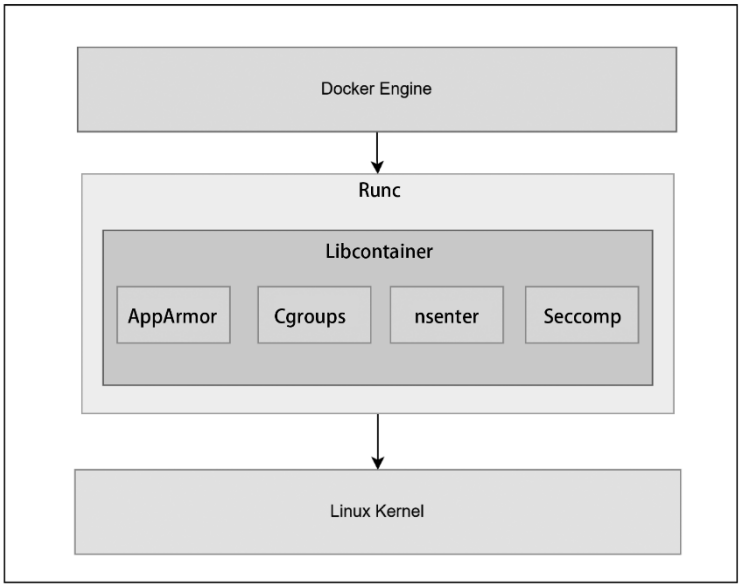

图 3-2　Docker 关键技术实现

因此，Docker 容器的实现主要依赖 Linux 的 Namespace 的隔离能力、Linux Cgroups 的限制能力以及基于 Rootfs 的文件系统。

3.5.3　是否可以替换关键技术之外的部分

关键技术对于软件的目标而言具有不可替代性，那么修改和替换关键技术之外的部分，是否同样可以实现软件的既定目标呢？我们仍然以 Docker 为例继续讨论。

Docker的目的是创建可以独立运行的容器，支撑这个目标实现的技术是Linux内核提供的Namespace、Controlled Groups和Rootfs技术，架构中的其他部分都是为这个目标服务的，因此属于可替换的技术。实际上，在Docker之外还有许多其他容器技术，如Podman、AWS ACI等。它们的关键技术相同，但在其他方面各具特色，以Podman为例，它与Docker的架构有显著不同，最明显的区别是Podman架构中没有后台守护进程（Daemon）。

3.6　示例 2——图形展示软件：关键技术与支撑技术的区别

许多软件都有将业务数据图形化展示的需求，数据可视化可以使用户对业务数据一目了然。这样的例子有很多：在办公系统中将审批流程以流程图的形式展现；在生产调度系统中，将供应链数据以物流图的形式展现；在软件运行监测系统中，将数据流以图形方式展示等。具体的业务领域虽然不同，但处理的过程是相似的，通常包括以下几个部分：

- 数据抽取：抽取需要显示的数据，并以适当的数据结构进行存储。

- 可视化模型转换：抽取出的业务数据通常不能完全满足可视化的需求，我们需要将不同的数据类型转换为对应的图元，并且需要使用一定的算法确定这些图元的显示位置。
- 图形显示：采用某一种图形显示框架，实现可视化模型的显示。

在上面 3 个步骤中，涉及关键技术、支撑技术和一般技术，我们逐一分析。

数据抽取属于一般技术，只要了解数据源的数据结构基本就可以完成工作，这部分可以放到实施阶段进行。

可视化模型转换属于关键技术，图元的位置计算涉及业务中数据上下游的关系，因此算法依赖于数据的业务含义，不同业务含义的模型转换算法是不同的，具备不可替代性。例如，供应链图形化算法与数据流图形化的算法完全不同，不能互相替代。

图形显示属于支撑技术，我们需要一种图形显示技术来展示可视化模型。换句话说，如果没有合适的图形显示技术，就无法实现这一功能。然而，图形显示技术不是关键技术，因为有很多种图形展示框架可供选择，我们可以选择 x6，也可以选择 GoJs 等。

3.7 本章小结

关键技术是软件实现业务目标不可替代的技术，因此在软件架构设计中，首先需要识别并解决关键技术问题。以 Docker 为例，实现应用容器化运行的关键技术，依赖于 Linux 内核提供的 Namespace、Controlled Groups 和 Rootfs 技术。

除关键技术外，软件架构在落地时还需要其他技术的支撑，这些技术就是支撑技术。常见的支撑技术包括数据存储、异步与并发处理、安全性设计、缓存、分布式技术等。

在架构设计过程中，还需要确定软件实现的技术路线，技术路线规定了软件实现所需采用的技术手段和具体步骤。

本章通过实例说明了什么是关键技术，以及关键技术和支撑技术的区别和联系。