第 1 章
从对象到 Web 服务

Web 服务在现实中已经应用了很多年。在此期间，就如何使用 Web 服务而言，开发人员和架构师们经常会遇到一些反复出现的设计挑战。我们了解到，在解决某些特定的问题时，有些特定的设计方法的确比其他方法更有效。本书的读者应该是那些正在使用 Web 服务或者正在考虑使用 Web 服务的软件开发人员和架构师。本书的目的是让他们了解一些最常见的和基础的 Web 服务设计方案，并帮助他们决定何时使用这些设计方案。这里讨论的所有概念都源于现实生活中的经验教训，并通过一些代码示例来演示这些经过实践验证的方案。

服务开发人员会遇到一系列问题，如下所述。

- 如何创建服务 API，普通风格的 API 是什么样的，特殊风格的 API 应该在什么时候使用？
- 在复杂会话中，多个参与方可以在一段持续的时间内交换数据。那么，客户端和服务如何进行通信，创建复杂会话的基础是什么？
- 实现服务逻辑有哪些可供选择的方法，什么时候应该使用特定的某种方法？
- 如何减少客户端与服务所使用的底层系统的耦合？
- 如何发现关于服务的信息？
- 如何在客户端或服务端提供对身份验证、数据校验、缓存和日志记录等通用功能的支持？
- 对服务进行的哪些改变会中断客户端的应用？
- 服务的版本管理有哪些常用的方法？
- 如何设计服务，才能让它既支持业务逻辑的不断演化，又不必强制客户端频繁地升级？

以上这些问题只是我们必须回答的问题中的一小部分。本书将会帮助你找到合适你的应用场景的解决方案。

在本章中，你将会学习服务到底是什么，Web 服务是如何解决之前服务的不足之处的。

1.1 Web 服务是什么

从技术角度来说，服务（service）这个术语可以指任何软件功能，比如执行一项业务任务、提供文件访问（如文本、文档、图片、视频、音频等），或者执行一些通用功能（如身份验证或日志记录）。对于这些目标，服务实现可以使用自动化的工作流引擎、属于领域模型（Domain Model）的对象 [POEAA]、商业软件包、传统应用程序的 API、面向消息的中间件（Message-Oriented Middleware，MOM），当然，还有数据库。用许多方法都可以实现服务。事实上，各种各样的技术，例如 CORBA 和 DCOM、为 REST 和 SOAP/WSDL 开发的新式软件框架等，都可以用来创建服务。

本书主要介绍如何使用服务在不同应用之间共享逻辑功能，以及如何使运行于不同计算平台上的软件互相协作。这些平台可以是硬件、操作系统（如 Linux、Windows、z/OS、Android、iOS）、软件框架（如 Java、.NET、Rails）和编程语言的任意组合。本书所讨论的服务，我们都假设它们是在处理客户端应用之外的机器上执行的。服务可以在与客户端在同一台机器上进行处理，不过，服务通常都是在另一台机器上进行处理的。虽然可以用诸如 CORBA 和 DCOM 之类的技术来创建服务，但本书的重点是 Web 服务。Web 服务为集成不同的系统提供了方法，并通过 HTTP 来输出可重用的业务功能。这些服务或者是将 HTTP 作为一种简单的信息运输工具，通过它来承载数据（如 SOAP/WSDL 服务）；或者是将 HTTP 作为一种完整的应用控制协议，为服务行为定义各种语义（如 RESTful 服务）。

术语

Web 服务开发人员经常使用不同的术语来称呼角色相当的概念。遗憾的是，这已经导致了很多混淆。因此，我们用下面这两列澄清这些术语，并将其作为参考。第一列列举了几个用于表示发送请求或触发事件的软件处理过程的名称。第二列包含了用于描述对这些请求和事件做出响应或反应的软件功能的术语。每列包含的这些术语都是同义词。

客户端（Client）	服务（Service）
请求者（Requestor）	提供者（Provider）

服务消费者（Service consumer）　　服务提供者（Service provider）

本书使用“客户端”和“服务”这两个术语，因为它们经常出现在 SOAP/WSDL 服务和 RESTful 服务中。

在很大程度上，Web 服务是一种用于解决分布式对象缺点的技术。因此，先回顾一下历史，认清使用 Web 服务的动机，这对我们理解 Web 服务会有所帮助。

1.2　从本地对象到分布式对象

在大多数现代编程语言中，对象（Object）是一种典范式的结构，用于封装行为（如业务逻辑）和数据。对象通常是“细粒度”的，也就是说对象具有很多细致的属性（如 `Firstname`、`Lastname`）或方法（如 `getAddress`、`setAddress`）。对于使用对象的开发人员来说，他们经常都可以访问到对象实现的内部细节，所以通常将对象提供的重用形式称为白盒重用（white-box reuse）。客户端在使用对象时，首先要实例化对象，再按顺序调用它们的属性和方法，来完成某种任务。一旦对象实例化完毕，它们就可以在客户端调用之间保持一定的状态。不幸的是，在不同编程语言和平台之间使用这些类并不总是件容易的事。因此，组件技术就逐渐发展起来，以部分地解决这一问题。

设计组件，是为了在不同的编程语言之间提供促进软件重用的手段（如图 1.1 所示）。其目的是为了将软件单元组装成复杂的应用程序，就像用电子元器件组装电路板一样。因为使用组件的开发人员无法看到或修改组件的内部实现，所以称组件提供的重用方式为黑盒重用（black-box reuse）。组件将相互关联的对象集中到可部署的二进制软件单元中，在应用程序中可以直接插入这些软件单元。在这一概念的基础上，软件供应商又创建了 ActiveX 控件，这种控件可以很容易地集成到桌面和基于 Web 的应用程序中。20 世纪 90 年代，Windows 平台上的完整组件产业也因此应运而生。这种模式规定，应用程序不能直接访问组件内部的对象，组件为应用程序提供二进制接口，用这些接口来描述对象的方法、属性和事件。这种二进制接口一般是用平台特定的接口定义语言（Interface Definition Language，IDL）创建的，如 Microsoft 接口定义语言（Microsoft Interface Definition Language，MIDL），而使用组件的客户端也通常只能运行在相同的计算平台上。

为了共享或重用对象封装的逻辑处理，最终都需要将对象部署到远程服务器上

（如图 1.2 所示）。这意味着，为客户端和分布式对象分配的内存，不但位于单独的地址空间中，而且还会出现在不同的机器上。像组件一样，分布式对象也支持黑盒重用。想使用分布式对象的客户端，可以利用许多远程化的技术，如 CORBA、DCOM、Java 远程方法调用（Remote Method Invocation，RMI）以及 .NET Remoting。这些技术的编译过程会生成一些二进制库文件，其中包含一个远程代理（Remote Proxy[GoF]），它包含了与远程对象进行通信所需的逻辑处理。只要客户端和分布式对象使用了同一种技术，那么所有事情都会运行得相当正常。但是，这些技术也有缺点。对于开发人员来说，实现这些技术相当复杂。在不同软件供应商的实现中，对象的序列化和反序列化处理也没有一定的标准。对于用不同的软件供应商工具集创建的对象，客户端在和它们进行通信时经常会遇到麻烦。此外，分布式对象一般通过 TCP 端口进行通信，而各软件供应商的实现中对这些端口也没有一定的标准。但防火墙通常会限制软件供应商选择的这些端口。为了解决这个问题，IT 管理员可以配置防火墙，允许访问请求的端口。在某些情况下，这样做可能会打开大量端口，给黑客留下更多可以利用的网络路径，降低了网络的安全性。如果已经允许访问特定的端口，那么它经常是已经另有其他用途。

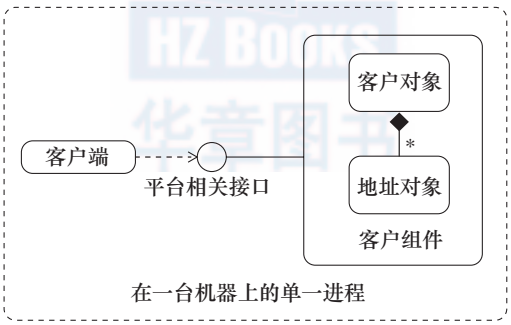

图 1.1 组件是一种在不同编程语言之间实现重用的便捷方法。但是，组件通常只是针对特定计算平台而创建的

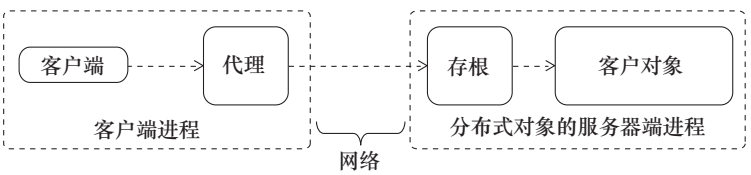

图 1.2 在分布式场景下经常使用对象。当客户端调用代理接口上的某个方法时，代理对象会通过网络将调用分发到远程存根（remote stub），再调用分布式对象上的相应方法。只要客户端和分布式对象使用相同的技术，整个调用过程就可以正常进行

分布式对象通常需要维护客户端调用之间的状态，这会导致许多降低系统可伸缩性的问题。

- 随着客户端负载的增加，服务器内存的利用率就会降低。
- 由于需要经常为客户端保存会话状态，有效的负载均衡技术就更加难以实现和管理。在默认情况下，后续请求会被转发回客户端会话建立时所在的服务器。这样就不能均匀地分布客户请求负载，除非采用复杂的基础结构（如共享内存缓存），这种结构可以从任何服务器上访问客户端会话。
- 服务器必须实现一种策略，释放为特定的客户端实例分配的内存。在大多数情况下，服务器要依赖客户端通知它什么时候释放内存。不幸的是，如果客户端崩溃了，那么为客户端分配的服务器内存就可能永远无法释放。

除了这些问题以外，如果负责维护客户端会话的进程崩溃了，那么客户端“正在处理的工作”就会丢失。

1.3　为什么使用 Web 服务

通过使用 Web 服务，在不同类型的客户端（如移动设备、桌面 PC、Web 应用程序）之间重用和共享共用逻辑就会变得相对容易。Web 服务能够触及的范围有可能非常广泛，因为它们依赖于普遍使用的开放标准，在不同的计算平台之间可以互操作，并且独立于底层的执行技术。所有的 Web 服务，至少都要使用 HTTP，并利用一些数据交换标准（如 XML、JSON）和常见的媒体类型。除此之外，Web 服务可以按两种完全不同的方式来使用 HTTP：一种是将 HTTP 作为一种应用协议，用于定义标准的服务行为；另一种只是简单地将 HTTP 作为一种传输数据的运输机制。无论如何，这两种 Web 服务都有助于快速应用集成，因为与之前的技术相比，它们更容易学习和实现。基于 Web 服务固有的互操作性和简单性，通过服务组合可以方便地创建复杂的业务处理。服务组合就是把多个较简单的服务组装成一个工作流，从而得到一个组合服务。

Web 服务建立了一层屏障，很自然地将客户端和用于完成客户端请求的方法隔离开来（如图 1.3 所示）。这样，只要服务的公共接口不发生破坏性变化（breaking change），客户端和服务就可能在一定程度上互相独立地发展（有关破坏性变化的更多细节，可以参考 7.2 节）。例如，服务的开发人员可以重新设计服务，用开源库代替原来自己定制开发的库，而客户端不必为此进行修改。

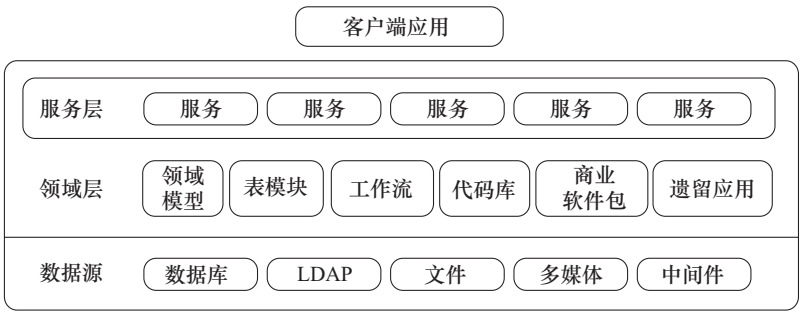

图 1.3 Web 服务有助于将客户端和用于完成客户端请求处理的业务逻辑相隔离。Web 服务建立了一层自然的屏障，使得客户端和领域实体（如工作流逻辑、表模块、领域模型 [POEAA] 等）可以各自独立地发展

1.4 Web 服务的考虑因素和替代方案

虽然 Web 服务的确适用于很多情况，但它也不是万能的。Web 服务的调用代价“昂贵”。在每次请求 Web 服务时，客户端必须将所有输入数据序列化（serialize）成字节流，再通过计算机处理（即地址空间）后将该字节流传输出去。Web 服务端也必须将这个字节流反序列化（deserialize）成它可以理解的数据格式和结构，再执行服务。如果服务提供的响应是某种“复杂类型”（即响应结果不仅仅是一个简单的 HTTP 状态码），那么 Web 服务就必须序列化并传输其响应结果，客户端也必须将接收到的字节流反序列化成它可以理解的格式和结构。所有这些处理活动都要花费时间。如果 Web 服务和客户端不在同一台机器上，那么调用 Web 服务花费的时间可能要比进程内调用所需时间高出几个数量级。

比延迟问题更重要的是，Web 服务调用通常必需依赖分布式通信。这意味着，客户端和服务端开发人员都必须准备好处理局部故障 [Waldo，Wyant，Wollrath，Kendall]。当客户端、服务或网络自身发生异常，但其他部分仍然继续正常运行时，这种情况就是局部故障。网络原本就不可靠，而且很多原因都可以导致问题出现。网络连接偶尔会超时或掉线，服务器也会经常超负荷运行，结果就是服务器不能接收和处理所有请求。甚至在处理某个请求时，服务也有可能会崩溃。客户端也可能崩溃，在这种情况下，服务可能没有办法返回响应。因此，必须采取多种策略来检测并处理这些局部故障。

由于这些固有的风险，开发人员和架构师首先应该明白还有哪些替代方案。在很多情况下，创建在客户端进程中可以导入、调用和执行的“服务库”（如 JAR、.NET 程

序集），可能是更好的办法。如果客户端和服务是为不同平台（如 Java 和 .NET）创建的，仍然有很多技术能够使完全不同的客户端和服务在同一个进程内互相合作。例如，可以在客户端中集成服务的运行时引擎，将服务加载到客户端环境中，再直接调用目标服务。我们可以在一个 .NET 客户端中集成一个 Java 虚拟机（Java Virtual Machine，JVM），将 Java 类库加载到 JVM，再通过 Java 本地接口（Java Native Interface，JNI）与目标类进行通信。你还可以使用第三方“桥接技术”。不过，这些替代方案可能会非常复杂，通常会加深客户端与服务实现技术的耦合。

因此，当进程外和跨多台机器的调用有意义时，就应该考虑 Web 服务。以下就是属于这种情况的几个例子。

- 客户端和服务属于不同的应用领域，无法简单地将“服务功能”导入到客户端。
- 客户端是一个复杂的业务处理过程，需要整合来自多个应用领域的功能。逻辑服务由不同的组织拥有和管理，变化频率也各不相同。
- 客户端和服务器之间的边界是自然划分的。例如，客户端可以是使用相同业务功能的移动设备或桌面应用程序。

即使跨机器的调用看起来很合理，开发人员也应该明智地考虑一下 Web 服务的替代方案。

- 可以使用 MOM 技术（如 MSMQ、WebSphere MQ、Apache ActiveMQ 等）来集成应用程序。不过，最好在一个安全环境中使用这些技术，如深藏在企业防火墙之后的位置。此外，它们还需要采用某种异步通信机制，这使得各方面都要面对一些新的设计挑战。MOM 解决方案经常使用特定平台的专有技术。有关这一主题的完整介绍，可以参考《Enterprise Integration Patterns：Designing，Building，and Deploying Messaging Solutions》[EIP] 一书。Web 服务经常会将请求转发至 MOM。
- 使用 HTTP 会有一定的开销，因为客户端和服务器建立连接需要时间。在某些高性能 / 高负载应用场景下，可能无法接受这种增加的时间开销。而像用户报文协议（User Datagram Protocol，UDP）之类的无连接协议，可能就是这种场合下可行的替代方案。但是这样做的代价是传输的数据可能会丢失、重复或接收顺序紊乱。
- 大多数 Web 服务框架可以配置为支持流式数据传输。对于数据发送方和接收

方来说，这种传输方式有助于最小化内存使用，因为不必缓存数据。响应时间也得以最小化，因为只要收到数据，接收方就可以使用它，而不用等待整个数据集传输完。不过，这种方式最适合传输大型文档或消息，而不适合实时传输大型多媒体文件（如视频和音频）。对于多媒体文件的传输，如实时流传输协议（Real Time Streaming Protocol，RTSP，http://www.ietf.org/rfc/rfc2326.txt）、实时传输协议（Real Time Transport Protocol，RTP，http://tools.ietf.org/html/rfc3550）以及实时控制协议（Real Time Control Protocol，RTCP，http://tools.ietf.org/html/rfc3605）通常都要比 HTTP 更合适。

1.5 服务和松散耦合的承诺

服务通常被描述成是松散耦合的。不过，“耦合”这一术语的定义不是一成不变的，它涉及到很多因素。耦合是指某个实体（如客户端）对另一个实体的依赖程度。当依赖很多时，就称耦合是高的或紧密的（如高耦合、紧密耦合）；相反，当依赖很少时，就认为耦合是低的或松散的（如低耦合、松散耦合）。

Web 服务确实可以消除客户端对实现服务所用的底层技术的依赖。但是，客户端和服务绝不可能完全没有一点耦合。一定程度的耦合总是存在的，而且通常是必需的。以下列举了几种服务设计者必须考虑的耦合形式。

- **功能耦合**：对于特定场景下的特定输入类型，客户端期望服务能够生成一致的特定结果。因此，客户端间接地依赖于 Web 服务实现的逻辑。如果这种逻辑实现得不正确，或者实际情况发生了变化，没有按照客户端的期望生成结果，那么客户端通常都会受到影响。
- **数据结构耦合**：客户端必须理解服务接收和返回的数据结构、这些结构中使用的数据类型，以及消息中使用的字符编码（如 Unicode）。如果数据结构提供了到相关服务的链接，那么客户端也必须知道如何解析这种信息的结构。客户端还需要知道服务会返回哪种 HTTP 状态码。服务开发人员必须谨慎，避免在数据结构中包含平台特定的数据类型（如日期类型）。
- **时间耦合**：如果收到请求，就必须马上处理，那么就是存在高度的时间耦合。这也意味着，支撑服务的系统（如数据库、原有的或打包的应用系统等）必须始终保持运行。如果能够延迟对请求的处理，则可以降低这种时间耦合。如果使

用请求 / 确认（Request/Acknowledge）模式（参见 3.3 节），则 Web 服务就能够取得这一效果。如果客户端必须阻塞并等候响应，那么时间耦合也会很高。为了减少这种形式的耦合，客户端可以采用异步响应处理器（Asynchronous Response Handler）模式（参见 6.4 节）。

- **URI 耦合**：客户端往往与服务 URI 高度耦合。客户端要么使用指向服务的静态 URI，要么按照一套简单的规则来构造服务 URI。遗憾的是，这会让服务所有者很难移动或重命名服务 URI，也无法采用新的 URI 构造模式，因为这些操作很可能会让客户端崩溃。以下这些模式有助于减少客户端对服务 URI 和位置的耦合：链接服务（参见 3.5 节）、服务连接器（参见 6.2 节）、服务注册表（参见 6.7.1 节），以及虚拟服务（参见 6.7.2 节）。

1.6　SOA 是什么

面向服务的架构（Service-Oriented Architecture，SOA），在业内已经有很多定义。有些人将它看作是一种架构的技术风格，为离散系统的集成和可重用业务功能的输出提供有效手段。而另一些人则从更广的角度看待它：

面向服务的架构是一种设计风格，在业务服务创建和使用的整个生命周期（从构思到废弃）中，它对各个方面均有指导意义 [Newcomer，Lomow，p.13]。

SOA 是一种用于组织和利用分布式能力的范式，这些分布式能力可能归属于不同的所有域 [OASIS Ref Model]。

这些观点表明，SOA 是一种设计范式或方法学，其中将“业务功能”看作是服务，将不同的服务组织成逻辑域，以某种方式管理它们的生命周期。虽然和面向对象的分析方法相比，SOA 可以帮助业务人员更自然地描述他们的需求，但仍然还有很多方法可以用于实现服务。本书将重点介绍几种可用于创建 SOA 的技术解决方案。

1.7　总结

通过消除与特定计算平台的耦合，Web 服务已经帮助我们清除了软件重用中的一个重大障碍。不过，还有很多实现服务设计的方法，开发人员仍然面临着一系列必须解决的问题。本书将帮助你找到最适合你的应用场合的解决方案。