第 3 章

服务并行设计

在请求到达系统前，我们通过分流设计对用户的请求进行了分流。在请求到达系统之后，我们仍然可以使用反向代理等手段在系统内部对请求进行分流，让系统内的多个节点共同处理用户的请求。

在系统内设置多个节点可以对请求进行进一步的分散，但也会引入数据同步、请求分配等问题。本节我们将详细了解这方面的知识。但在此之前，我们需要先区分并行与并发的概念。

3.1 并行与并发

并行与并发是我们在架构设计与软件开发中经常涉及的概念，不过两者的意义并不完全相同。

并行（Parallelism）是指在同一时刻有多个任务同时进行。例如，你在家中一边读书一边听歌，则“既读书又听歌”描述的就是并行。因为读书和听歌这两件事情是同时发生的，如图 3.1 所示。

并发（Concurrency）是指多个任务中的每个任务都被拆分成细小的任务片，从属于不同任务的任务片被轮番处理。因此，任意时刻都只有一个任务在进行。但是从宏观上看，这些任务像是被同时处理的。例如，你在家中一边读书一边看综艺节目，则“既读书又看综艺节目”描述的就是并发。因为在任意时刻，你要么在低头读书，要么在抬头看综艺节目，这两件事情实际上是交替进行的，如图 3.2 所示。

所以说，并行是一种真正意义上的“并”，而并发只是宏观表现上的“并”。

不过要注意的是，并发和并行的区分仅限于微观。从宏观上看，并发或并行的任务都像是同时开展的。以并发请求为例，在同一时刻向某个系统发送大量请求，这些请求几乎会被同时处理。这些请求在系统内部是被“并行”处理还是被“并发”处理的，则无法从系统外部判断出来。因此，我们宏观上常常将“并行”和“并发”统称为“并发”。

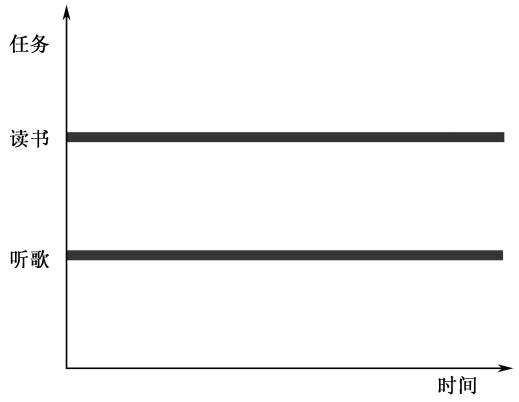

图 3.1　并行

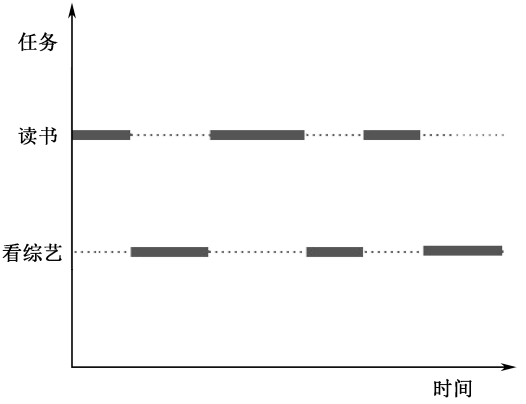

图 3.2　并发

本节介绍如何将单节点系统拆分为多节点系统。在拆分后的系统中，每个节点都有独立的处理器。因此，各个节点之间的处理操作一定是并行的。

3.2　集群系统

集群系统是实现系统内分流的一种最简单的方法。这种集群系统中，可以部署多个节点，每个节点都是同质的（有同样的配置，运行同样的程序），共同对外提供服务。可以通过反向代理等手段将外界请求分配到系统的节点上。集群系统的结构如图 3.3 所示。

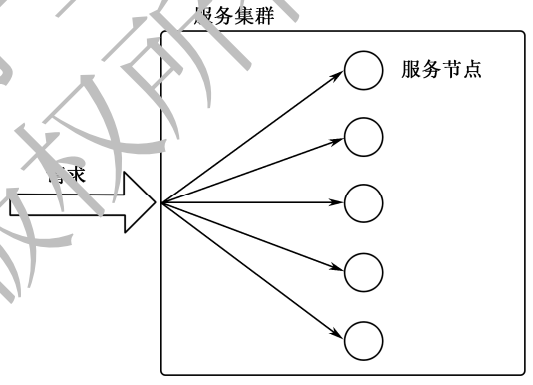

图 3.3　集群系统的结构

这样的集群系统也会带来一些问题。一个最明显的问题是同一个用户发出的多个请求可能会落在不同的节点上，打破服务的连贯性。例如，用户发出 R1、R2 两个请求，且 R2 请求的执行要依赖 R1 请求的信息（例如，R1 请求会触发一个任务，而 R2 请求用来

查询任务的执行结果）。如果 R1 请求和 R2 请求被分配到不同的节点上，则 R2 请求的操作便无法正常执行。

为了解决上述问题，业界研发了以下几种集群方案，我们逐一进行介绍。

3.2.1 无状态的节点集群

最容易实现从单节点到多节点扩展的系统是无状态系统，它可以拆分为多个无状态节点。所谓无状态节点是说，假设用户 U 先后发出 R1、R2 两个请求，则无论 R2 请求和 R1 请求是否落到同一个节点上， R2 请求都能得到同样的结果。某个节点给出的结果与该节点之前是否收到 R1 请求完全无关。

很多节点是有状态的。例如，某个节点接收到外部请求后修改了某对象的属性，那后面的请求在查询对象属性时便可以读取到修改后的结果，如果后面的请求落到了其他节点上，则读取到的是修改前的结果。

要想让系统满足无状态，必须保证其所有的接口都是恒等类接口，即接口被调用前后，系统状态不能发生任何改变。显然只有查询类接口能够满足这个需要。

即便是由多个无状态节点组成的系统，也会出现协作问题。典型的就是并行唤醒问题。例如，我们需要为一个包含多个无状态节点的系统增加定时功能，在每天凌晨对外发送一封邮件。我们会发现该集群中的所有节点都会在凌晨被同时唤醒，并各自发送一封邮件。

我们希望整个系统对外发送一封邮件，而不是让每个节点都发送一封邮件。可集群的节点确实会这样工作，因为所有节点是同质的，它们运行的程序是一致的。

我们可以通过外部请求唤醒来解决无状态节点集群的并行唤醒问题。在指定时刻，由外部系统发送一个请求给服务集群触发定时任务。因为该请求最终只会交给一个节点处理，因此实现了独立唤醒。

无状态节点集群设计简单，可以方便地进行扩展。但其只适合满足无状态要求的系统，应用范围比较受限。

3.2.2 单一服务节点集群

许多服务是有状态的，用户的历史请求在系统中组成了上下文，系统必须结合用户上下文对用户的请求进行回应。在聊天系统中，用户之前的对话（是通过过去的请求实现的）便是上下文；在游戏系统中，用户之前购买的装备、晋升的等级（也是通过过去的请求实现的）便是该用户的上下文。

要想让一个系统是有状态的，则必须要在处理用户的每个请求时能读取和修改用户的上下文信息。这在单一节点的系统中是容易实现的，只要将每个用户的信息都保存在

这个节点上即可，而在节点集群中，这一切就变得复杂起来。其中一个最简单的办法是在节点和用户之间建立对应关系，图 3.4 展示了这种对应关系。

- 任意用户都有一个对应的节点，该节点上保存了该用户的上下文信息。
- 用户的请求总是落在与之对应的节点上的。

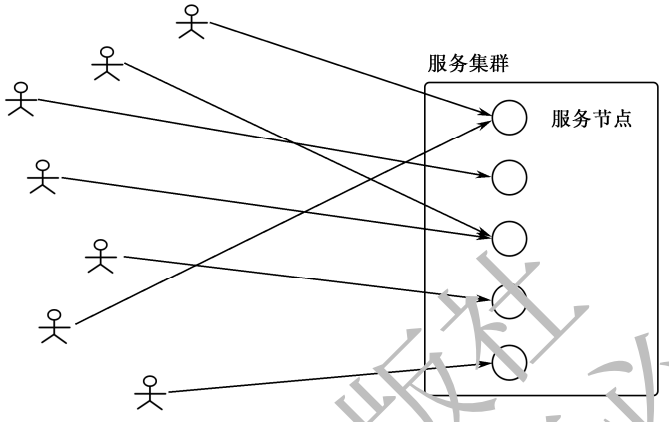

图 3.4　用户与指定节点的对应关系

这种系统的一个非常大的特点就是各个节点是隔离的。这些节点运行同样的代码，有同样的配置，然而保存了不同用户的上下文信息，各自服务自身对应的用户。

虽然集群包含多个节点，但是从用户角度看服务某个用户的始终是同一个节点，因此我们将这种集群称为单一服务节点集群。

实现单一服务节点集群要解决的一个最重要的问题便是如何建立和维护用户与节点之间的对应关系。具体的实现有很多种，下面我们列举常用的几种。

- 在用户注册时由用户选择节点，很多游戏服务就采用这种方式。
- 在用户注册时根据用户所处的网络分配节点，一些邮件服务采用的就是这种方式。
- 在用户注册时根据用户 ID 分配节点，许多聊天系统采用这种方式。
- 在用户登录时随机或根据规则分配节点，然后将分配结果写入 cookie，接下来根据请求中的 cookie 将用户请求分配到指定节点。

其中，最后一种方式与前几种方式略有不同。前几种方式能保证用户对应的节点在整个用户周期内不会改变，而最后一种方式则只保证用户对应的节点在一次会话周期内不会改变。最后一种方式适合用在两次会话之间无上下文关系的场景下。例如，一些登录系统、权限系统等，只需要维护用户这次会话的上下文信息。

无论采用哪种方式，都保证了用户在会话过程中对应的节点不会改变。系统只需要在会话中将用户的请求路由到对应的节点即可。该路由操作根据系统分流方案的不同由反向代理、规则中心等组件完成。

单一服务节点集群方案能够解决有状态服务的问题。但因为各个节点之间是隔离的，无法互相备份，当某个服务节点崩溃时，该节点对应的用户将会失去服务。因此，这种设计方案的容错性比较差。

3.2.3 信息共享的节点集群

有一种方案可以解决有状态服务问题，并且不会因为某个服务节点崩溃而造成某些用户失去服务，那就是信息共享的节点集群。在这种集群中，所有节点连接到一个公共的信息池中，并在这个信息池中存储所有用户的上下文信息，该系统如图 3.5 所示。

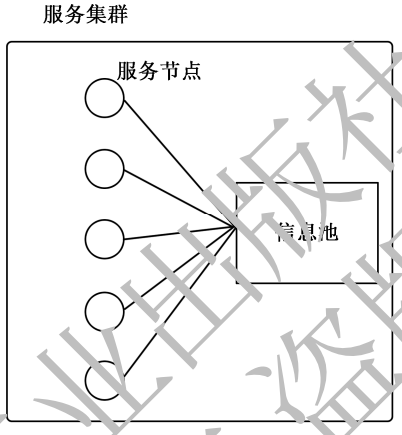

图 3.5 信息共享的节点集群

这是一种非常常见的将单节点系统扩展为多节点系统的方式。通常，服务节点和信息池分别部署在不同的机器上，我们可以通过部署和启动多个服务节点的方式来扩展集群。

数据库常作为信息池使用。任何一个节点接收到用户请求后，都从数据库中读取该用户的上下文信息，然后根据用户请求进行处理。在处理结束后，立刻将新的用户状态写回数据库中。除了传统数据库，也可以是其他类型的信息池。例如，使用 Redis 将信息池作为共享内存，存储用户的 Session 信息。

在信息共享的节点集群中，每个节点都从信息池中读取和写入用户状态，因此对于用户而言，每个节点都是等价的。用户的请求落在任意一个节点上都会得到相同的结果。

在这种集群中，节点之间的信息是互通的，因此可以使用分布式锁解决并发唤醒等节点间的协作问题。一种简单的做法是在定时任务被触发时，每个节点都向信息池中以同样的键写入一个不允许覆盖的数据。最终肯定只有一个节点能够写入成功，这个写入成功的节点获得执行定时任务的权限。

信息共享的节点集群通过增加服务节点而提升了集群的计算能力。但因为多个节点共享信息池，受到信息池容量、读写性能的影响，系统在数据存储容量、数据吞吐能力

等方面的提升并不明显。

3.2.4　信息一致的节点集群

对于信息共享的节点集群而言，其运算能力是分散到各个节点上的，但其存储能力是集中在信息池中的。这使得信息池成了故障单点和性能瓶颈。

为了避免信息池成为整个系统的瓶颈，我们可以让每个节点独立拥有自身的信息池。为了继续保证系统提供有状态的服务，我们必须确保各个信息池中的数据信息是一致的，如图 3.6 所示。

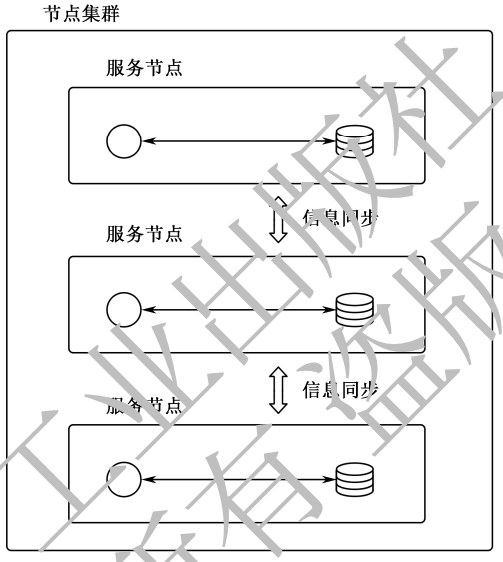

图 3.6　信息一致的节点集群

这种信息一致的节点集群通常也会被称为分布式系统，但从严格意义上讲，它仍然是集群。因为分布式系统中的节点是异构的，不同的节点可能从属系统中的不同模块。而这里的节点是同构的，它们的出现是为了分担并发数高带来的压力。但是，信息一致的节点集群也需要面对分布式系统中经常面对的问题——分布式一致性问题。

分布式一致性要求用户在分布式系统的某个节点上进行了变更操作，并在经过一定时间后能够从系统中的每个节点上读取到这个变更。

我们可以通过图 3.7 所示的例子来简单了解分布式一致性问题。

在图 3.7 中，调用方首先通过节点集群将变量 *a* 的值设置为 5，然后读取变量 *a* 的值，结果读取到变量 *a* 的值为 3。这种情况是完全有可能发生的，因为用户的两次读写操作可能访问的是两个节点，只要节点之间的信息不同步或同步存在时延，便会出现这种情况。

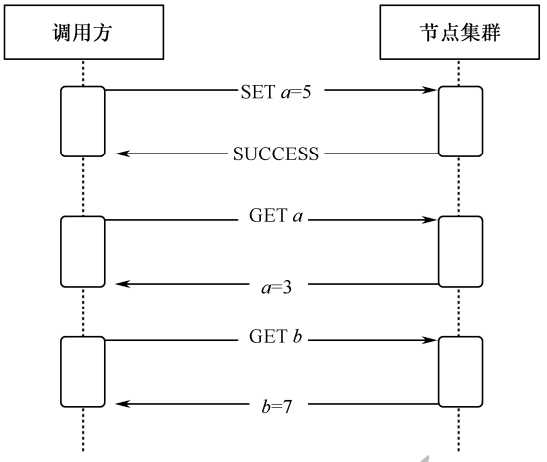

图 3.7　分布式一致性

如果图 3.7 所示的情况可能发生，那么该节点集群便不满足一致性（至少是不满足线性一致性）。如果节点集群不满足一致性，那么从集群中读出的任何值都不是可信的。例如，某个调用方从节点集群中读出 b=7，则这个结果不可信，因为其他的调用方完全可能在同一时刻读到 b=8。

在定义一致性的概念时，我们说在“一定时间后”能够读取到变更。根据“一定时间”的长短可以将一致性分为许多类型，如严格一致性、线性一致性（又叫强一致性、原子一致性）、因果一致性、最终一致性等。

要实现分布式一致性，就是要完成各个节点之间的信息同步。根据要实现一致性的强度不同，其成本也不同。例如，要实现线性一致性，我们可以使用两阶段提交算法、三阶段提交算法等，这些算法的实施会对各个节点的吞吐量造成较大影响；要实现最终一致性，我们可以使用具有重试功能的异步消息中心等，这种方式对节点的吞吐量影响较小，但是集群可能会出现读写不一致的情况。具体采用哪种信息同步方式达到哪种级别的一致性，需要我们根据实际的应用场景定夺。

信息一致的节点集群适合用在读多写少的场景下。在这种场景下，较少发生节点间的信息同步，并且能充分发挥多个信息池的吞吐能力优势。

3.3　分布式系统

集群系统将一个节点的并发请求分散到多个节点上，降低了每个节点的压力。在集群系统中，各个节点是同质的，各自运行一套完整且相同的应用程序。如果应用程序比较复杂，则其性能会受到硬件资源的制约。

应用从诞生之初便不断发展，在这个发展过程中，应用的功能可能会增加，应用的

边界可能会扩展，进而包含越来越多的模块。最终，应用可能会变为一个包含众多功能模块的单体应用（Monolithic Application）。当这样的应用运行在物理节点上时，便会因为 CPU 资源、内存资源、IO 资源等的不足导致性能降低。这种性能的降低不是由并发数高引发的，而是由系统自身的复杂性引发的。

除了效率问题，单体应用也带来了开发维护、可靠性方面的问题。

- 业务逻辑复杂：应用中包含了众多功能模块，而每个模块都可能和其他模块存在耦合。应用开发者必须了解系统的所有模块的业务逻辑后才可以展开开发工作。这给开发者，尤其是新开发者带来了挑战。
- 变更维护复杂：应用中任何一个微小的变动与升级都必须重新部署整个系统，随之而来的还有各种全量测试、回归测试等工作。
- 难以分拆升级：应用中不同组件需要的软硬件资源可能不同，但是因为它们都被整合进了系统中，所以难以对它们进行单独的升级。
- 可靠性变差：任何一个功能模块的异常都可能导致应用宕机，进而使整个应用不可用。但应用模块众多，又会使应用很难在短时间内恢复。

为了解决以上问题，我们可以将单体应用拆分成多个子应用，让每个子应用部署到单独的机器上，以此来提升系统的效率、可靠性。这时，单体应用变成分布式应用，如图 3.8 所示。

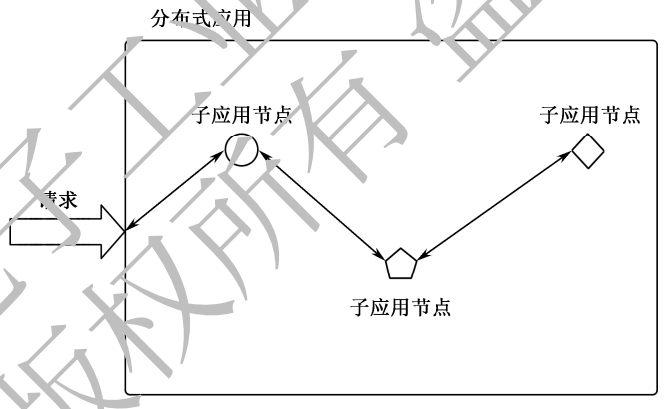

图 3.8　分布式应用

分布式应用通过拆分子应用，将原本集中在一个应用、机器上的压力分散到多个应用、机器上，进一步提升了系统的压力承载能力。分布式应用还便于单体应用内部模块之间的解耦，使得这些子应用可以独立地开发、部署、升级和维护。

当然，分布式应用也要解决分布式一致性问题。例如，在一个分布式系统中，子应用 A1 负责完成商品订单管理功能，子应用 A2 负责完成库存管理功能。当外部购买请求到达分布式应用后，如果子应用 A1 完成了订单生成工作，则子应用 A2 也应该完成库存