二

01 中间件生态（上）：有哪些类型的中间件?

你好，我是丁威。

最近十年是互联网磅礴发展的十年，IT 系统从单体应用逐渐向分布式架构演变，高并发、高可用、高性能、分布式等话题变得异常火热，中间件也在这一时期如雨后春笋般涌现出来，那到底什么是中间件呢？存在哪些类型的中间件呢？同一类型的中间件，我们该怎么选择？接下来的两节课，我们就来聊聊这些问题。

中间件的种类很多，我们无法把所有类型和产品列出来逐一讲解。但是每个类别的中间件在设计原理、使用上有很多共同的考量标准，只要了解了最重要、最主流的几种中间件，我们就可以方便地进行知识迁移，举一反三了，然后学习其他中间件将变得非常简单。

所以呢，你可以把这两节课看作是提纲挈领的知识清单。下面我们讲到的中间件你不一定都能够用上，但在需要的时候，可以帮你从更加高屋建瓴的角度迅速决策。

什么是中间件?

先来说说什么是中间件，我认为中间件是游离于业务需求之外，专门为了处理项目中涉及高可用、高性能、高并发等技术需求而引入的一个个技术组件。它的一个重要作用就是能够实现业务代码与技术功能之间解耦合。

这么说是不是还有点抽象？在这里定义里，我提到了业务需求和技术需求，关于这两个词我需要再解释一下。

业务需求，笼统地说就是特定用户的特定诉求。以我们快递行业为例：人与人之间需要跨城市传递物品，逢年过节我们需要给远方的亲人寄礼物，这就是所谓的业务需求。

技术需求，就是随着业务的不断扩展，形成规模效应后带来的使用上的需求。例如上面提到的寄件服务，原先只需要服务 1 万个客户，用户体验非常好，但现在需要服务几个亿的用户，用户在使用的过程中就会出现卡顿、系统异常等问题，因此产生可用性、稳定性方面的技术诉求。

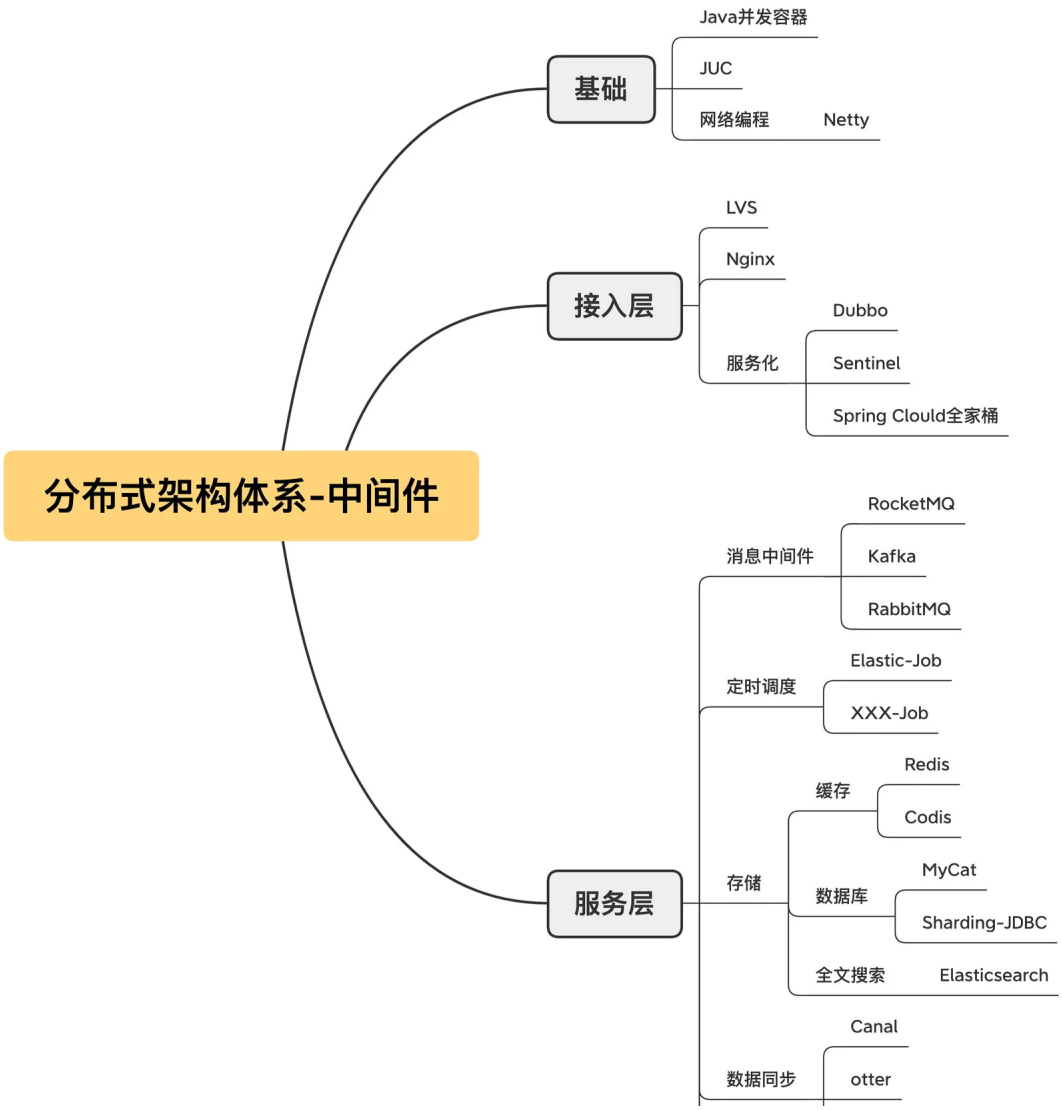
为了解决各式各样的业务和技术诉求，代码量会越来越多。如果我们任凭业务代码与技术类

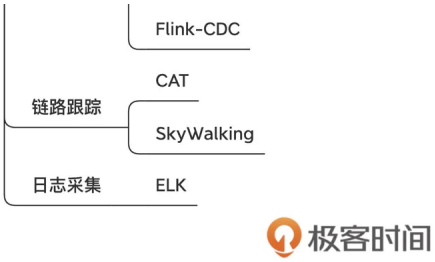
代码没有秩序地纠缠在一起，系统会变得越来越不可维护，运营成本也会成指数级增加，故障频发，最终直接导致项目建设失败。

怎么解决这个问题呢？计算机领域有一个非常经典的分层架构思想，还有这样一句话“计算机领域任何一个问题都可以通过分层来解决，如果不行，那就再增加一层。”要想让系统做得越来越好，我们**通常会基于分层的架构思想引入一个中间层，**专门来解决可用性、稳定性、高性能方面的技术类诉求，这个中间层就是中间件，这也正是“中间件”这个词的来源。

中间件生态漫谈

明白了中间件的内涵，我们再来看看市面上有哪些中间件。我在开篇词中已经提到过了，中间件的种类繁多，我整理了一版分布式架构体系中常见的中间件，你可以先打开图片仔细看一看。





结合我 10 多年的从业经验，特别是对互联网主流分布式架构体系的研读，我发现**微服务中间件、消息中间件、定时调度的使用频率极高，在解决分布式架构相关问题中是排头兵，具有无可比拟的普适性。**这三者的设计理念和案例能对分布式、高可用和高并发等理念实现全覆盖。

所以，在专栏的第三章到第五章，我会深度剖析微服务、消息中间件和定时调度这三个方向，结合生产级经典案例深入剖析它们的架构设计理念，带你扎实地掌握分布式架构设计相关的基本技能。

- 微服务

具体而言，作为软件架构从单体应用向分布式演进出现的第一个新名词，微服务涉及分布式领域中服务注册、服务动态发现、RPC 调用、负载均衡、服务聚合等核心技术，而 Dubbo 在微服务领域是当仁不让的王者。所以在微服务这一部分，我们会以 Dubbo 为例进行实战演练。

- 消息中间件

随着微服务的蓬勃发展，系统的复杂度越来越高，加上互联网秒杀、双十一、618 等各种大促活动层出不穷，我们急切需要对系统解耦和应对突发流量的解决办法，这时候消息中间件应运而生了，它同样成为我们架构设计工作中最常用的工具包。常用的消息中间件包括 RocketMQ、Kafka，它们在适用性上有所不同，如何保障消息中间件的稳定性是一大挑战。

- 定时调度

而定时调度呢？我们既可以认为它是个技术需求，也可以认为它是一个业务类需求，通过研读 ElasticJob、XXL-Job 等定时调度框架，可以很好地提升我们对业务需求的架构设计能力。

这三部分我们会在后面的模块中重点展开，所以这一模块不做深入讲解。接下来，为了让你对主流中间件有一个更全面的认知，我会分两节课对另外的几类中间件（数据库、缓存、搜索、日志等）进行简要阐述，以补全你的中间件知识图谱，帮助你更加有底气、有效率地进行决策。这节课，我们先来看看数据库中间件。

数据库中间件

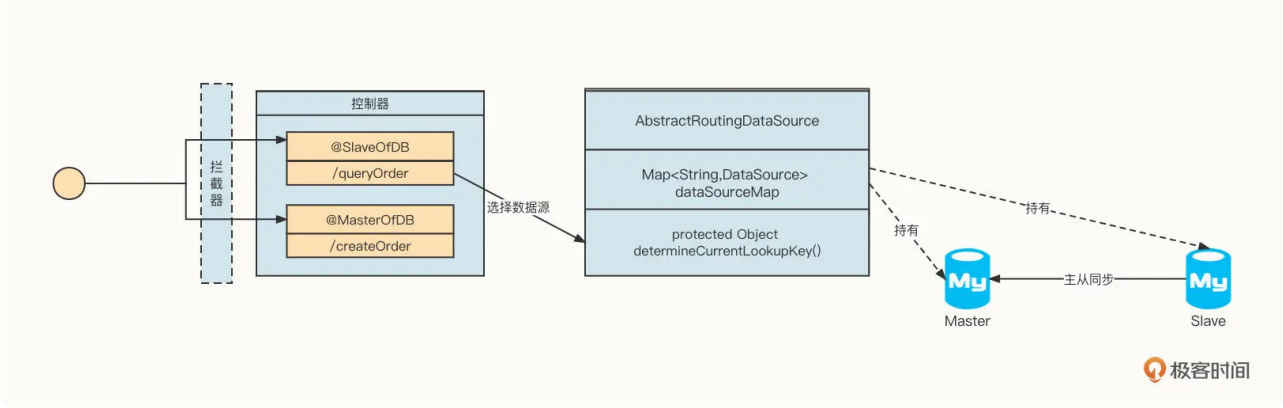
数据库中间件应该是我们接触得最早也是最为常见的中间件，在引入数据库中间件之前，由于单体应用向分布式架构演进的过程中单表日数据急速增长，单个数据库的节点很容易成为系统瓶颈，无法提供稳定的服务。因此，为了解决可用性问题，在技术架构领域通常有如下两种解决方案：

- 读写分离
- 分裤分表

我们先分别解析下这两个方案。最后再来看一看，引入数据库中间件给技术带来的简化。

读写分离

这是我在没有接触中间件之前，在一个项目中使用过的方案：



这个方案的实现要点有三个。

第一，在编写业务接口时，要通过在接口上添加注解来指示运行时应该使用的数据源。例如，@SlaveofDB 表示使用 Slave 数据库，@MasterOfDB 表示使用主库。

第二，当用户发起请求时，要先经过一个拦截器获取用户请求的具体接口，然后使用反射机制获取该方法上的注解。举个例子，如果存在 @SlaveofDB，则往线程上下文环境中存储一个名为 dbType 的变量，赋值为 slave，表示走从库；如果存在 @MasterOfDB，则存储为 master，表示走主库。

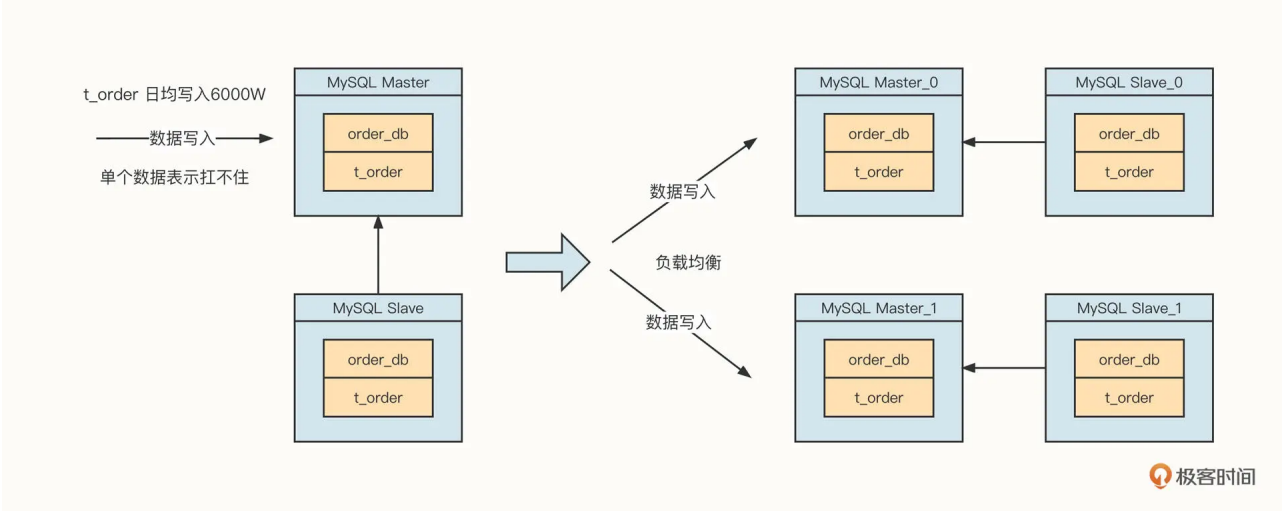
第三，在 Dao 层采用 Spring 提供的路由选择机制，继承自 AbastractRoutingDataSource。应用程序启动时自动注入两个数据源 (master-slave)，采用 key-value 键值对的方式存储。在真正需要获取链接时，根据上下文环境中存储的数据库类型，从内部持有的 dataSourceMap 中获取对应的数据源，从而实现数据库层面的**读写分离**。

总结一下，读写分离的思路就是通过降低写入节点的负载，将耗时的查询类请求转发到从节点，从而有效提升写入的性能。

但是，当业务量不断增加，单个数据库节点已无法再满足业务需求时，我们就要对数据进行切片，分库分表的技术思想就应运而生了。

分库分表

分库分表是负载均衡在数据库领域的应用，主要的原理你可以参考下面这张图。

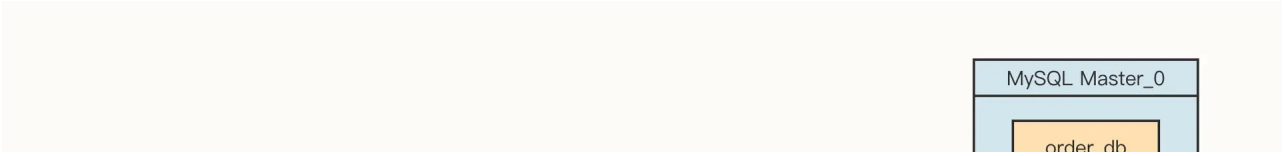


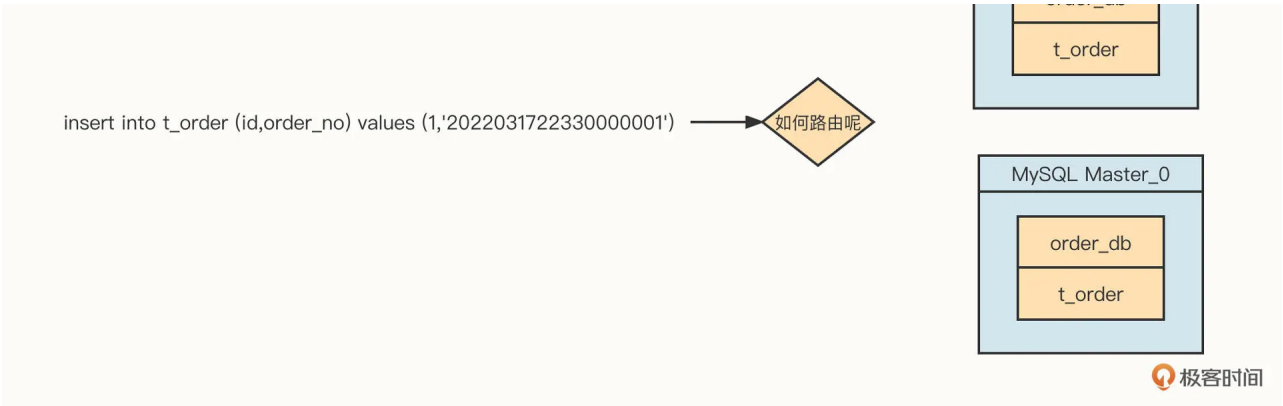
简单说明一下。分库分表主要是通过引入多个写入节点来缓解数据压力的。因此，在接受写入请求后，负载均衡算法会将数据路由到其中一个节点上，多个节点共同分担数据写入请求，降低单个节点的压力，提升扩展性，解决单节点的性能瓶颈。

不过，要实现数据库层面的分库分表还是存在一定技术难度的。**因为分库分表和读写分离一样，最终要解决的都是如何选择数据源的问题。**所以在分库分表方案中，首先我们要有两个算法。

- 一个分库字段和分库算法，即在进行数据查询、数据写入时，根据分库字段的值算出要路由到哪个数据库实例上；
- 一个分表字段和分表算法，即在进行数据查询、数据写入时，根据分表字段的值算出要路由到哪个表上。

不管是上面的分库、还是分表都需要解决一个非常关键的问题：SQL 解析。你可以看下面这张图。





如果订单库的分库字段设置为 order_no，要想正确执行这条 SQL 语句，我们首先要解析这条 SQL 语句，提取 order_no 的字段值，再根据分库算法 (负载均衡算法) 计算应该发送到哪一个具体的库上执行。

SQL 语句语法非常复杂，要实现一套高性能的 SQL 解析引擎绝非易事，如果按照上面我提供的解决方案，将会带来几个明显的弊端。

- 技术需求会污染业务代码，维护成本高

在业务控制器中需要使用注解来声明读写分离按相关的规则进行，随着业务控制的不断增加、或者读写分离规则的变化，我们需要对系统所有注解进行修改，但业务逻辑其实并没有改变。这就造成两者之间相互影响，后期维护成本较高。

- 技术实现难度大，极大增加开发成本

由于 SQL 语句的格式太复杂、太灵活，如果不是数据库专业人才，很难全面掌握 SQL 语法。在这样的情况下，你写出的 SQL 解析引擎很难覆盖所有的场景，容易出现遗漏最终导致故障的发生；这也给产品的性能带来极大挑战。

那怎么办呢？其实，我们完全可以使用业界大神的开源作品来解决问题，这就要说到数据库中间件了。

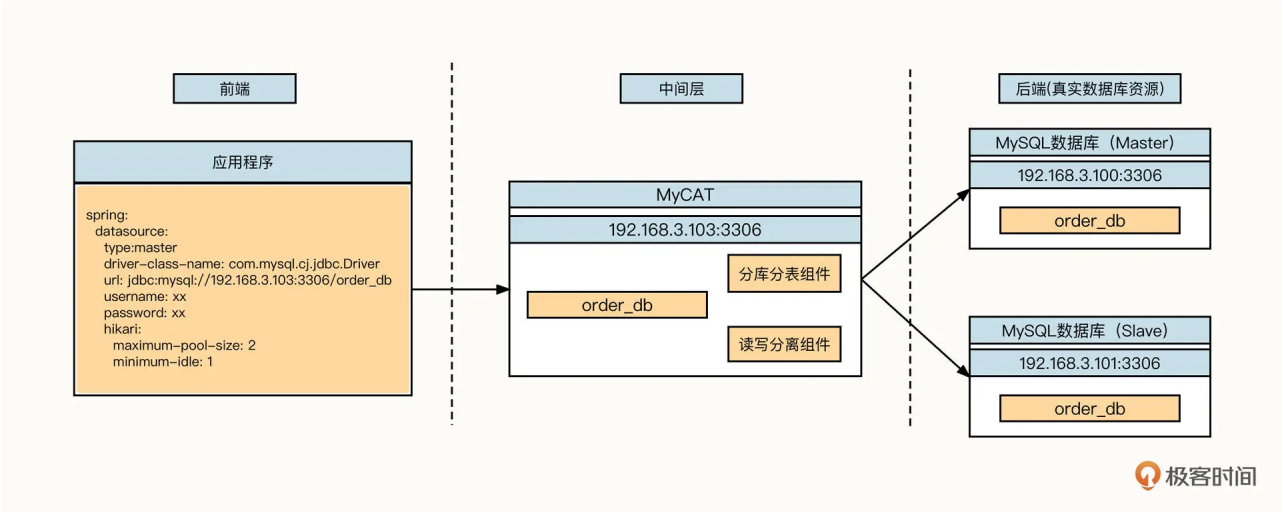
引进数据库中间件

技术类诉求往往是相通的，极具普适性，为了解决上面的通病，根据分层的架构理念，我们通常会引入一个中间层，专门解决数据库方面的技术类需求。

MyCat 和 ShardingJDBC/ShardingSphere 是目前市面最主流的两个数据库中间件，二者各有优势。

MyCat 服务端代理模式

先来看下 MyCat 代理数据库。它的工作模式可以用下面这张图概括：



面对应用程序，MyCat 会伪装成一个数据库服务器 (例如 MySQL 服务端)。它会根据各个数据库的通信协议，从二进制请求中根据协议进行解码，然后提取 SQL，并根据配置的分库分表、读写分离规则计算出需要发送到哪个物理数据库。

随后，面对真实的数据库资源，MyCat 会伪装成一个数据库客户端。它会根据通信协议将 SQL 语句封装成二进制流，发送请求到真实的物理资源，真实的物理数据库收到请求后解析请求并进行对应的处理，再将结果层层返回到应用程序。

这种架构的优势是它对业务代码无任何侵入性，应用程序只需要修改项目中数据库的连接配置就可以了，而且使用简单，易于推广。同时它也有劣势：

- 存在性能损耗

数据库中间件需要对应用程序发送过来的请求进行解码并计算路由，随后它还要再次对请求进行编码并转发到真实的数据库，这就增加了性能开销。

- 高度中心化，数据库中间件容易成为性能瓶颈

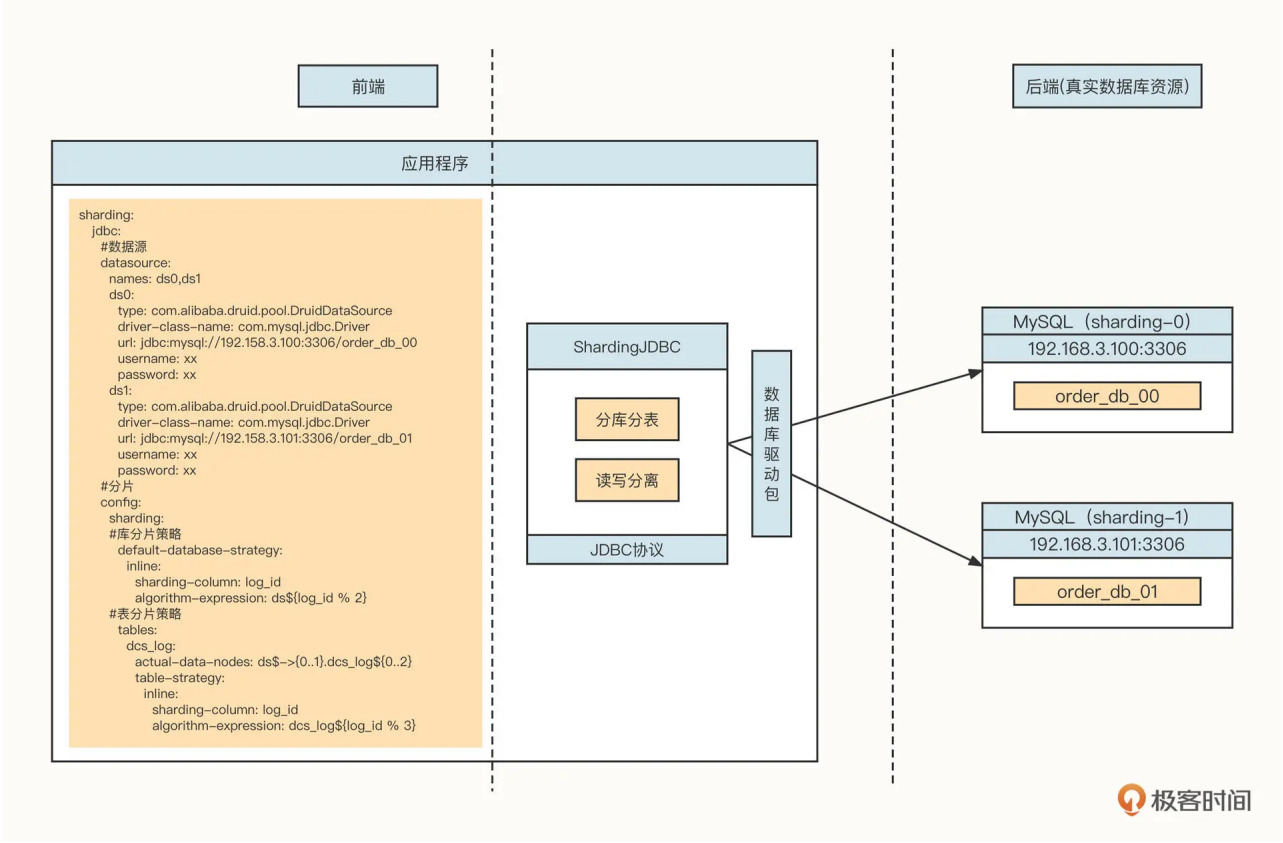
数据库中间件需要处理所有的数据库请求，返回结果都需要在数据库中进行聚合，虽然减少了后端数据库的压力，但中间件本身很容易成为系统的瓶颈，扩展能力受到一定制约。

- 代理层实现复杂，普适性差

数据库中间件本身的实现比较复杂，需要适配市面上各主流数据库，例如 MySQL、Oracle 等，通用性大打折扣。

ShardingJDBC 客户端代理模式

下面我们再来看下 ShardingJDBC 客户端代理数据库。ShardingJDBC 的工作模式如下图所示：



ShardingJDBC 主要实现的是 JDBC 协议。实现 JDBC 协议，其实主要是面向 java.sql.Datasource、Connection、ResultSet 等对象编程。它通常以客户端 Jar 包的方式嵌入到业务系统中，ShardingjJDBC 根据分库分表的配置信息，初始化一个 ShardingJdbcDatasource 对象，随后解析 SQL 语句来提取分库、分表字段值，再根据配置的路由规则选择正确的后端真实数据库，最后，ShardingJDBC 用各种类型数据库的驱动包将 SQL 发送到真实的物理数据库上。

我们同样来分析一下这个方案的优缺点。

主要的优势有如下几点：

- 无性能损耗

ShardingJDBC 使用的是基于客户端的代理模式，不需要对 SQL 进行编码解码等操作，只要根据 SQL 语句进行路由选择就可以了，没有太多性能损耗。

- 无单点故障，扩展性强

ShardingJDBC 以 Jar 包的形式存在于项目中，其分布式特性随着应用的增加而增加，扩展

性极强。

- 基于JDBC协议，可无缝支持各主流数据库

JDBC 协议是应用程序与关系型数据库交互的业界通用标准，市面上所有关系型数据库都天然支持 JDBC，故不存在兼容性问题。

当然缺点也很明显，对于分库分表，它没有一个统一的视图，运维类成本较高。举个例子，如果订单表被分成了 1024 个表，这时候如果你想根据订单编号去查询数据，必须人为计算出这条数据存在于哪个库的哪个表中，然后再去对应的库上执行 SQL 语句。

为了解决 ShardingJDBC 存在的问题，官方提供了 ShardingSphere，其工作机制基于代理模式，与 MyCat 的设计理念一致，作为数据库的代理层，提供统一的数据聚合层，可以有效弥补 ShardingJDBC 在运维层面的缺陷，因此**项目通常采用 ShardingDBC 的编程方式，然后再搭建一套 ShardingSphere 供数据查询。**

在没有 ShardingSphere 之前，使用 MyCat 也有一定优势。MyCat 对业务代码无侵入性，接入成本也比较低。但 ShardingSphere 弥补了 ShardingJDBC 对运维的不友好，而且它的性能损耗低、扩展性强、支持各类主流数据库，可以说相比 MyCat 已经占有明显的优势了。

所以如果要在实践生产中选择数据库中间件，我更加推荐 ShardingJDBC。

除了上面的原因，从资源利用率和社区活跃度的角度讲，首先，MyCat 的“前身”是阿里开源的 Cobar，是数据库中间件的开山鼻祖，技术架构稍显古老，而 ShardingJDBC 在设计之初就可以规避 MyCat 的固有缺陷，摒弃服务端代理模式。代理模式需要额外的机器搭建 MyCat 进程，引入了新的进程，**势必需要增加硬件资源的投入。**

其次，ShardingJDBC 目前已经是 Apache 的顶级项目，它的社区活跃度也是 MyCat 无法比拟的。一个开源项目社区越活跃，寻求帮助后问题得到解决的概率就会越大，越多人使用，系统中存在的 Bug 也更容易被发现、被修复，这就使得中间件本身的稳定性更有保障。

	MyCat	ShardingJDBC
优势	• 对业务代码无任何侵入性，使用简单，易于推广。	• 无性能损耗。 • 无单点故障，可用性、扩展性高。 • 基于JDBC协议，可无缝支持各主流数据库。 • 无需增加额外的硬件成本。 • Apache顶级项目，社区活跃度高。
劣势	• 存在性能损耗。 • 高度中心化，数据库中间件容易成为性能瓶颈。	• 对于分库分表，没有一个统一的视图，运维类成本较高

	• 代理层实现复杂，普适性差。 • 社区活跃度、规模较小，技术支持度低。	（可以通过搭建ShardingSphere来弥补）。
总结	综合考虑，在实际工作中，优先考虑ShardingJDBC。	

极客时间

总结

好了，这节课就讲到这里，我们来做个小结。通过刚才的学习，我们知道了中间件的概念，它是为了解决系统中的技术需求，将技术需求与业务需求进行解耦，让我们专注于业务代码开发的一个个技术组件。中间件的存在，就是为了解决高并发、高可用性、高性能等各领域的技术难题。

在项目中，合理引用中间件能极大提升我们系统的稳定性、可用性，但同时也会提升系统维护的复杂度，对我们的技术能力提出了更高的要求，我们必须要熟练掌握项目中引用的各种中间件，深入理解其工作原理、实现细节，提高对中间件的驾驭能力，否则一旦运用不当，很可能给系统带来灾难性的故障。

为了让你对中间件有一个更加宏观的认识，我给你列举了市面最为常用的中间件。虽然现在新的中间件层出不穷，但在我看来，大都不超过我列的这几类。这节课我们重点讲了两个主流的数据库中间件，下节课，我们再来解读缓存、全文索引、分布式日志这几类中间件。

课后题

学完这节课，我也给你出两道课后题吧！

1. 从数据库中间件的演变历程中，你能提炼出哪些分布式架构设计理念？
2. 请你以订单业务场景，搭建一个 2 库 2 表的 ShardingSphere 集群，实现数据的插入、查询功能。

如果你想要分享你的修改或者想听听我的意见，可以提交一个 GitHub的 push 请求或 issues，并把对应地址贴到留言里。我们下节课见！

上一页 下一页