【经典必读】web网站架构演变过程，电商网站升级打怪

2018-02-27 小M java思维导图

> 公众号回复“**加群**”，添加小编微信拉你进群一起交流学习。
> ---
> 文章来自：http://www.cnblogs.com/xiaoMzjm

前言

我们以javaweb为例，来搭建一个简单的电商系统，看看这个系统可以如何一步步演变。

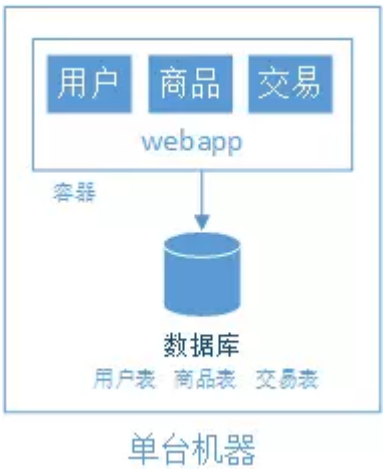
该系统具备的功能：

用户模块：用户注册和管理
商品模块：商品展示和管理
交易模块：创建交易和管理

阶段一、单机构建网站

网站的初期，我们经常会在单机上跑我们所有的程序和软件。此时我们使用一个容器，如tomcat、jetty、jboos，然后直接使用JSP/servlet技术，或者使用一些开源的框架如maven+spring+struct+hibernate、maven+spring+springmvc+mybatis；最后再选择一个数据库管理系统来存储数据，如mysql、sqlserver、oracle，然后通过JDBC进行数据库的连接和操作。

把以上的所有软件都装载同一台机器上，应用跑起来了，也算是一个小系统了。此时系统结果如下：

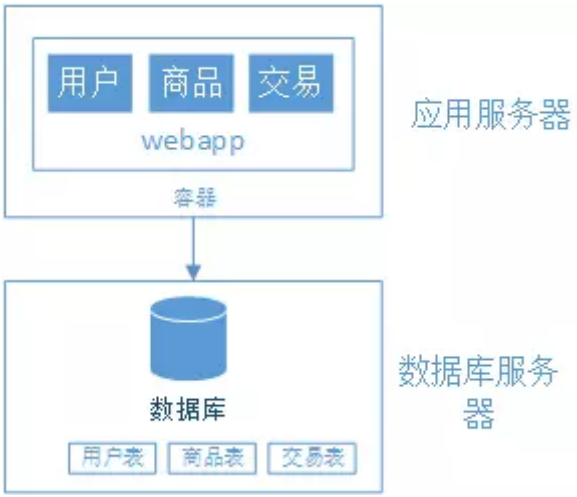


阶段二、应用服务器与数据库分离

随着网站的上线，访问量逐步上升，服务器的负载慢慢提高，在服务器还没有超载的时候，我们应该就要做好准备，提升网站的负载能力。假如我们代码层面已难以优化，在不提高单台机器的性能的情况下，增加机器是一个不错的方式，不仅可以有效地提高系统的负载能力，而且性价比高。

增加的机器用来做什么呢？此时我们可以把数据库，web服务器拆分开来，这样不仅提高了单台机器的负载能力，也提高了容灾能力。

应用服务器与数据库分开后的架构如下图所示：

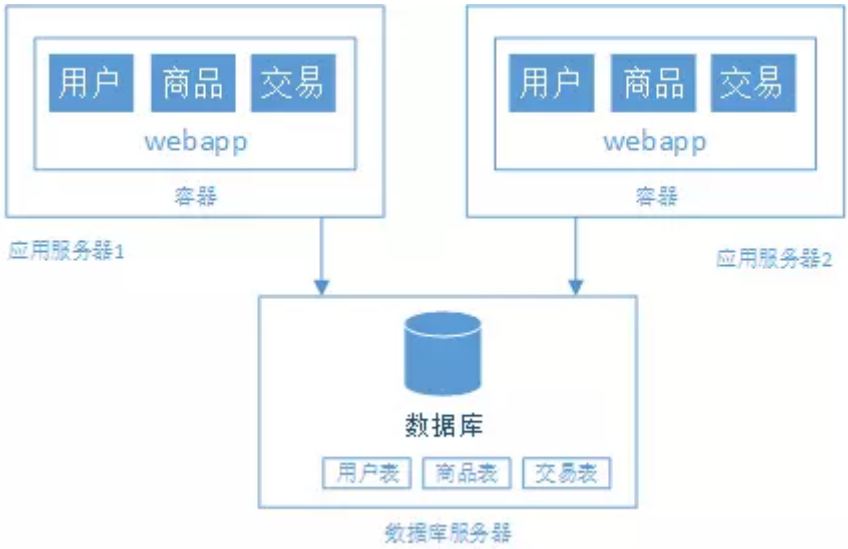


阶段三、应用服务器集群

随着访问量继续增加，单台应用服务器已经无法满足需求了。在假设数据库服务器没有压力的情况下，我们可以把应用服务器从一台变成了两台甚至多台，把用户的请求分散到不同的服务器中，从而提高负载能力。多台应用服务器之间没有直接的交互，他们都是依赖数据库各自对外提供服务。

著名的做故障切换的软件有keepalived，keepalived是一个类似于layer3、4、7交换机制的软件，他不是某个具体软件故障切换的专属品，而是可以适用于各种软件的一款产品。keepalived配合上ipvsadm又可以做负载均衡，可谓是神器。

我们以增加了一台应用服务器为例，增加后的系统结构图如下：



系统演变到这里，将会出现下面四个问题：

1. 用户的请求由谁来转发到到具体的应用服务器
2. 有什么转发的算法
3. 应用服务器如何返回用户的请求
4. 用户如果每次访问到的服务器不一样，那么如何维护session的一致性

我们来看看解决问题的方案：

1、第一个问题即是负载均衡的问题，一般有5种解决方案：

http重定向。HTTP重定向就是应用层的请求转发。用户的请求其实已经到了HTTP重定向负载均衡服务器，服务器根据算法要求用户重定向，用户收到重定向请求后，再次请求真正的集群

优点：简单。

缺点：性能较差。

DNS域名解析负载均衡。DNS域名解析负载均衡就是在用户请求DNS服务器，获取域名对应的IP地址时，DNS服务器直接给出负载均衡后的服务器IP。

优点：交给DNS，不用我们去维护负载均衡服务器。

缺点：当一个应用服务器挂了，不能及时通知DNS，而且DNS负载均衡的控制权在域名服务商那里，网站无法做更多的改善和更强大的管理。

反向代理服务器。在用户的请求到达反向代理服务器时（已经到达网站机房），由反向代理服务器根据算法转发到具体的服务器。常用的apache，nginx都可以充当反向代理服务器。

优点：部署简单。

缺点：代理服务器可能成为性能的瓶颈，特别是一次上传大文件。

IP层负载均衡。在请求到达负载均衡器后，负载均衡器通过修改请求的目的IP地址，从而实现请求的转发，做到负载均衡。

优点：性能更好。

缺点：负载均衡器的宽带成为瓶颈。

数据链路层负载均衡。在请求到达负载均衡器后，负载均衡器通过修改请求的mac地址，从而做到负载均衡，与IP负载均衡不一样的是，当请求访问完服务器之后，直接返回客户。而无需再经过负载均衡器。

2、第二个问题即是集群调度算法问题，常见的调度算法有10种。

rr 轮询调度算法。顾名思义，轮询分发请求。

优点：实现简单

缺点：不考虑每台服务器的处理能力

wrr 加权调度算法。我们给每个服务器设置权值weight，负载均衡调度器根据权值调度服务器，服务器被调用的次数跟权值成正比。

优点：考虑了服务器处理能力的不同

sh 原地址散列：提取用户IP，根据散列函数得出一个key，再根据静态映射表，查处对应的value，即目标服务器IP。过目标机器超负荷，则返回空。

dh 目标地址散列：同上，只是现在提取的是目标地址的IP来做哈希。

优点：以上两种算法的都能实现同一个用户访问同一个服务器。

lc 最少连接。优先把请求转发给连接数少的服务器。

优点：使得集群中各个服务器的负载更加均匀。

wlc 加权最少连接。在lc的基础上，为每台服务器加上权值。算法为：（活动连接数*256+非活动连接数）÷权重 ，计算出来的值小的服务器优先被选择。

优点：可以根据服务器的能力分配请求。**sed 最短期望延迟。**其实sed跟wlc类似，区别是不考虑非活动连接数。算法为：（活动连接数+1)*256÷权重，同样计算出来的值小的服务器优先被选择。

nq 永不排队。改进的sed算法。我们想一下什么情况下才能“永不排队”，那就是服务器的连接数为0的时候，那么假如有服务器连接数为0，均衡器直接把请求转发给它，无需经过sed的计算。

LBLC 基于局部性的最少连接。均衡器根据请求的目的IP地址，找出该IP地址最近被使用的服务器，把请求转发之，若该服务器超载，最采用最少连接数算法。

LBLCR 带复制的基于局部性的最少连接。均衡器根据请求的目的IP地址，找出该IP地址最近使用的“服务器**组**”，注意，并不是具体某个服务器，然后采用最少连接数从该组中挑出具体的某台服务器出来，把请求转发之。若该服务器超载，那么根据最少连接数算法，在集群的**非**本服务器组的服务器中，找出一台服务器出来，加入本服务器组，然后把请求转发之。

3、第三个问题是集群模式问题，一般3种解决方案：

NAT：负载均衡器接收用户的请求，转发给具体服务器，服务器处理完请求返回给均衡器，均衡器再重新返回给用户。

DR：负载均衡器接收用户的请求，转发给具体服务器，服务器出来玩请求后直接返回给用户。需要系统支持IP Tunneling协议，难以跨平台。

TUN：同上，但无需IP Tunneling协议，跨平台性好，大部分系统都可以支持。

4、第四个问题是session问题，一般有4种解决方案：

Session Sticky。session sticky就是把同一个用户在某一个会话中的请求，都分配到固定的某一台服务器中，这样我们就不需要解决跨服务器的session问题了，常见的算法有ip_hash法，即上面提到的两种散列算法。

优点：实现简单。

缺点：应用服务器重启则session消失。

Session Replication。session replication就是在集群中复制session，使得每个服务器都保存有全部用户的session数据。

优点：减轻负载均衡服务器的压力，不需要要实现ip_hasp算法来转发请求。

缺点：复制时宽带开销大，访问量大的话session占用内存大且浪费。

Session数据集中存储：session数据集中存储就是利用数据库来存储session数据，实现了session和应用服务器的解耦。

优点：相比session replication的方案，集群间对于宽带和内存的压力减少了很多。

缺点：需要维护存储session的数据库。

Cookie Base：cookie base就是把session存在cookie中，有浏览器来告诉应用服务器我的session是什么，同样实现了session和应用服务器的解耦。

优点：实现简单，基本免维护。

缺点：cookie长度限制，安全性低，宽带消耗。

值得一提的是：

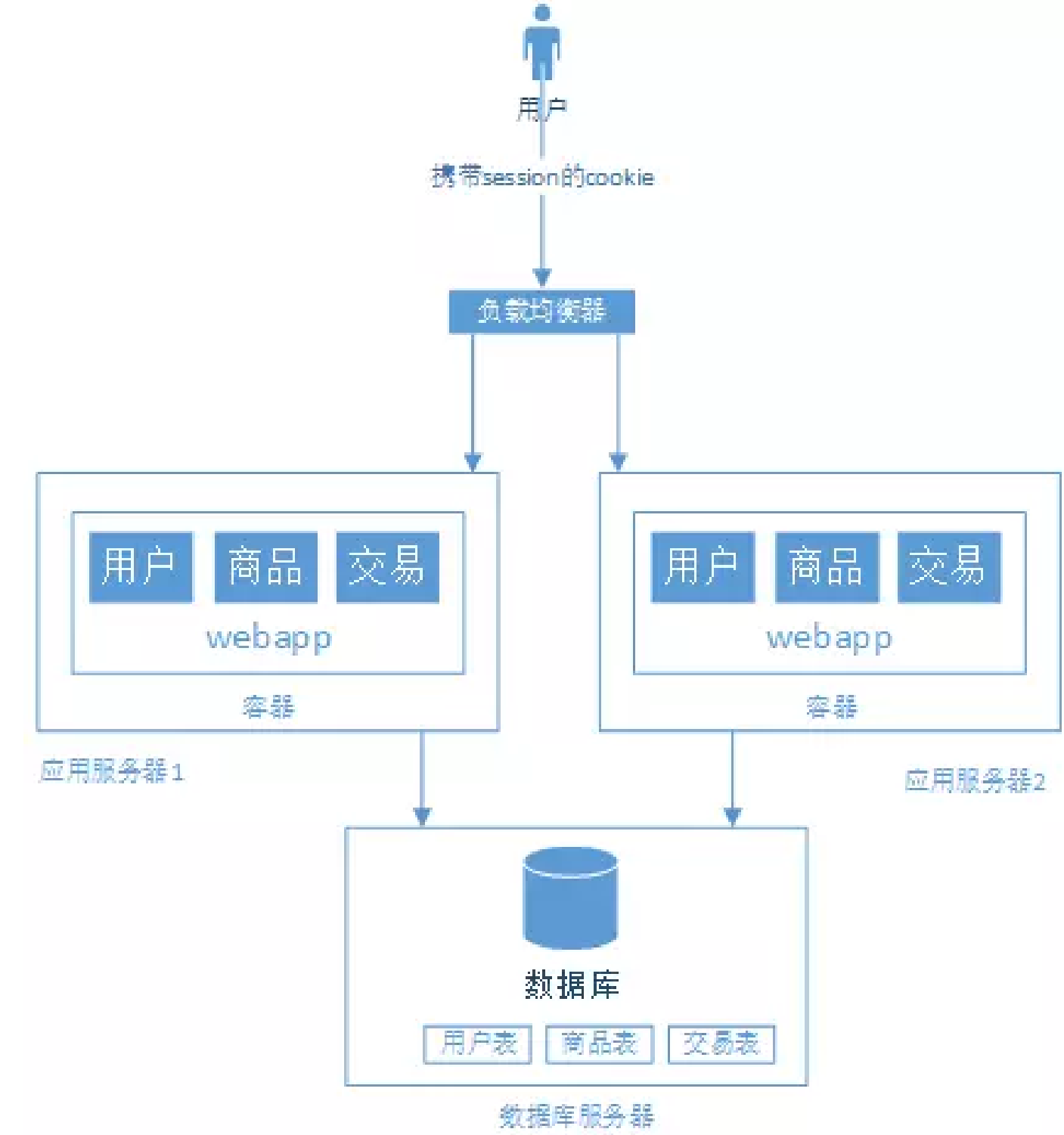
nginx目前支持的负载均衡算法有wrr、sh（支持一致性哈希）、fair（本人觉得可以归结为lc）。但nginx作为均衡器的话，还可以一同作为静态资源服务器。

keepalived+ipvsadm比较强大，目前支持的算法有：rr、wrr、lc、wlc、lblc、sh、dh

keepalived支持集群模式有：NAT、DR、TUN

nginx本身并没有提供session同步的解决方案，而apache则提供了session共享的支持。

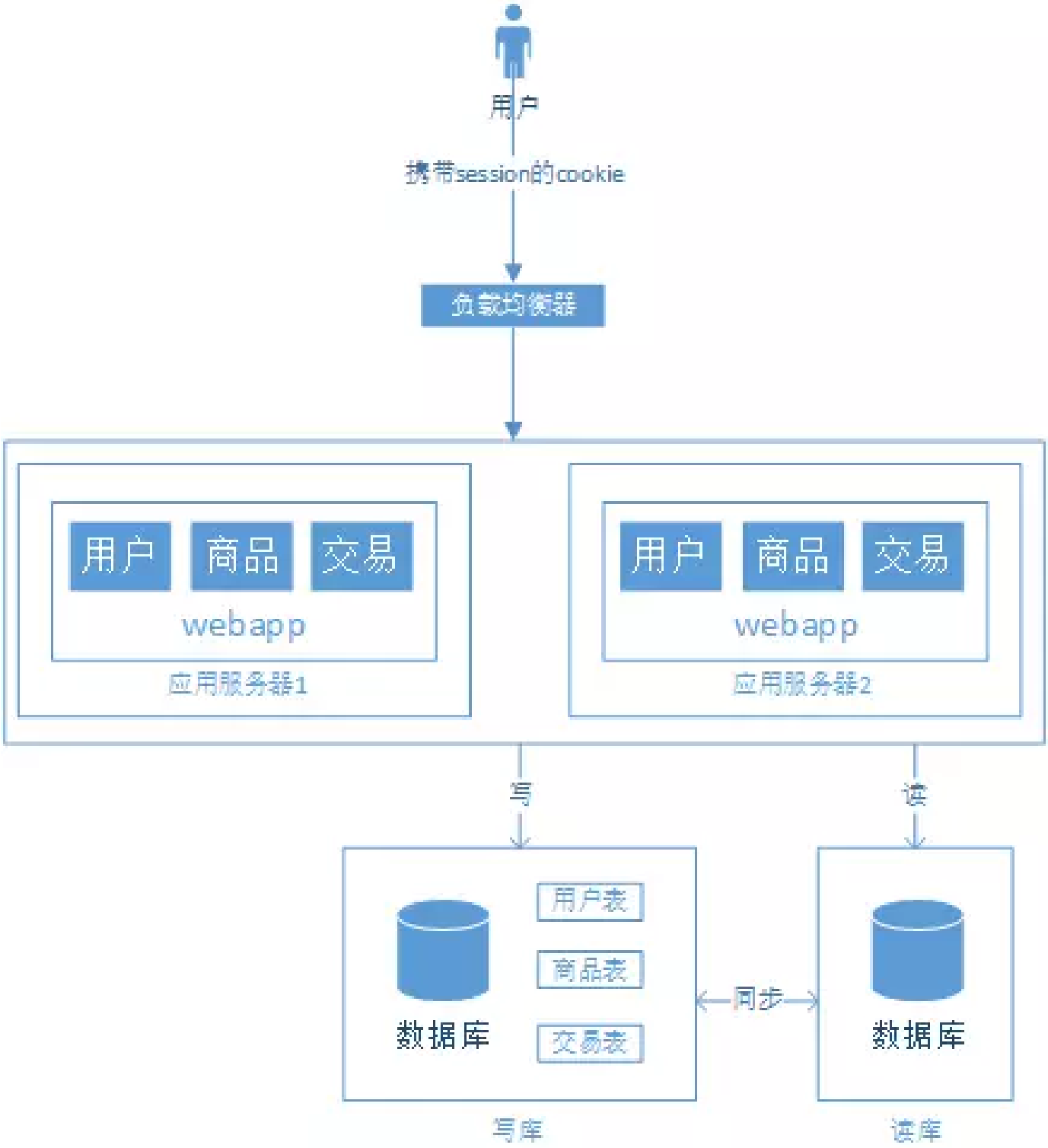
好了，解决了以上的问题之后，**系统的结构如下**：



阶段四、数据库读写分离化

上面我们总是假设数据库负载正常，但随着访问量的的提高，数据库的负载也在慢慢增大。那么可能有人马上就想到跟应用服务器一样，把数据库一份为二再负载均衡即可。但对于数据库来说，并没有那么简单。假如我们简单的把数据库一分为二，然后对于数据库的请求，分别负载到A机器和B机器，那么显而易见会造成两台数据库数据不统一的问题。那么对于这种情况，我们可以先考虑使用读写分离的方式。

读写分离后的数据库系统结构如下：



这个结构变化后也会带来两个问题：

1. 主从数据库之间数据同步问题
2. 应用对于数据源的选择问题

解决问题方案：

1. 我们可以使用MYSQL自带的master+slave的方式实现主从复制。
2. 采用第三方数据库中间件，例如mycat。mycat是从cobar发展而来的，而cobar是阿里开源的数据库中间件，后来停止开发。mycat是国内比较好的mysql开源数据库分库分表中间件。

阶段五、用搜索引擎缓解读库的压力

数据库做读库的话，常常对模糊查找力不从心，即使做了读写分离，这个问题还未能解决。以我们所举的交易网站为例，发布的商品存储在数据库中，用户最常使用的功能就是查找商品，尤其是根据商品的标题来查找对应的商品。对于这种需求，一般我们都是通过like功能来实现的，但是这种方式的代价非常大。此时我们可以使用搜索引擎的倒排索引来完成。

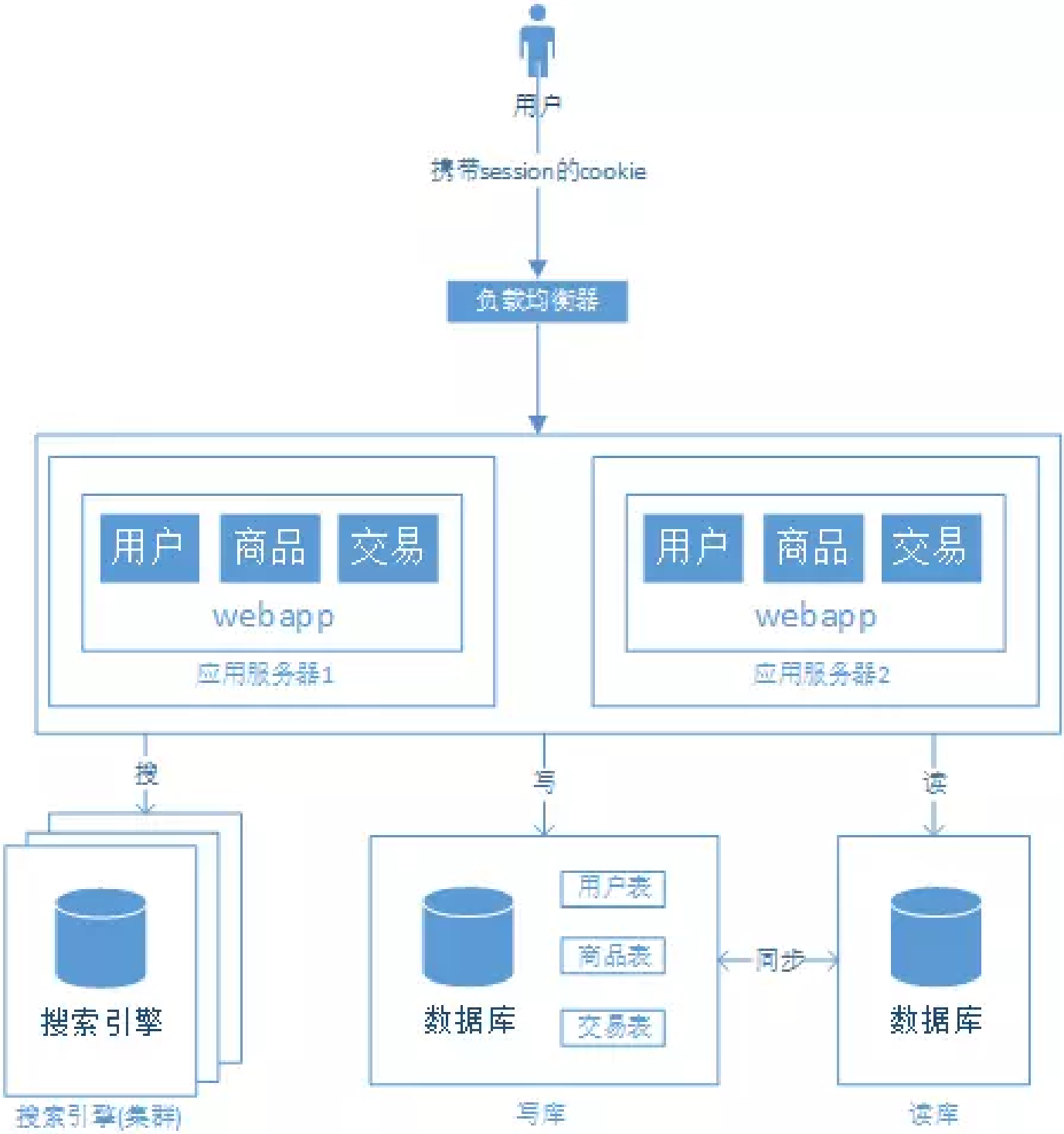
搜索引擎具有以下优点：
它能够大大提高查询速度。

引入搜索引擎后也会带来以下的开销：

带来大量的维护工作，我们需要自己实现索引的构建过程，设计全量/增加的构建方式来应对非实时与实时的查询需求。

需要维护搜索引擎集群

搜索引擎并不能替代数据库，他解决了某些场景下的“读”的问题，是否引入搜索引擎，需要综合考虑整个系统的需求。引入搜索引擎后的系统结构如下：



阶段六、用缓存缓解读库的压力

1、后台应用层和数据库层的缓存

随着访问量的增加，逐渐出现了许多用户访问同一部分内容的情况，对于这些比较热门的内容，没必要每次都从数据库读取。我们可以使用缓存技术，例如可以使用google的开源缓存技术guava或者使用memcacahe作为应用层的缓存，也可以使用redis作为数据库层的缓存。

另外，在某些场景下，关系型数据库并不是很适合，例如我想做一个“每日输入密码错误次数限制”的功能，思路大概是在用户登录时，如果登录错误，则记录下该用户的IP和错误次数，那么这

个数据要放在哪里呢？假如放在内存中，那么显然会占用太大的内容；假如放在关系型数据库中，那么既要建立数据库表，还要简历对应的java bean，还要写SQL等等。而分析一下我们要存储的数据，无非就是类似{ip:errorNumber}这样的key:value数据。对于这种数据，我们可以用NOSQL数据库来代替传统的关系型数据库。

2、**页面缓存**除了数据缓存，还有页面缓存。比如使用HTML5的localstroage或者cookie。

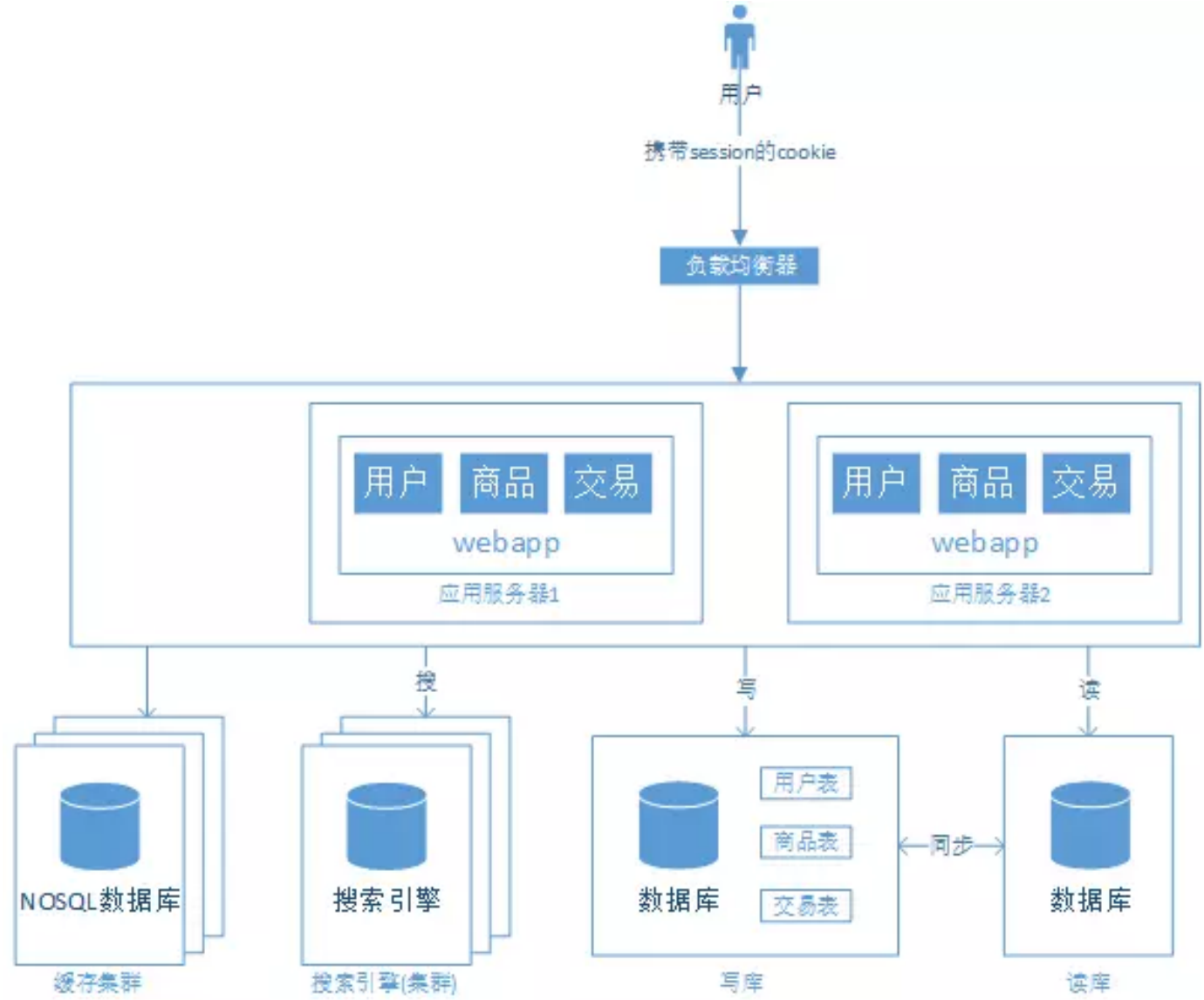
优点：
减轻数据库的压力
大幅度提高访问速度

缺点：
需要维护缓存服务器
提高了编码的复杂性

值得一提的是：

缓存集群的调度算法不同与上面提到的应用服务器和数据库。最好采用“一致性哈希算法”，这样才能提高命中率。这个就不展开讲了，有兴趣的可以查阅相关资料。

加入缓存后的结构：



阶段七、数据库水平拆分与垂直拆分

我们的网站演进到现在，交易、商品、用户的数据都还在同一个数据库中。尽管采取了增加缓存，读写分离的方式，但随着数据库的压力继续增加，数据库的瓶颈越来越突出，此时，我们可以有数据垂直拆分和水平拆分两种选择。

7.1、数据垂直拆分

垂直拆分的意思是把数据库中不同的业务数据拆分道不同的数据库中，结合现在的例子，就是把交易、商品、用户的数据分开。

优点：
解决了原来把所有业务放在一个数据库中的压力问题。
可以根据业务的特点进行更多的优化

缺点：
需要维护多个数据库

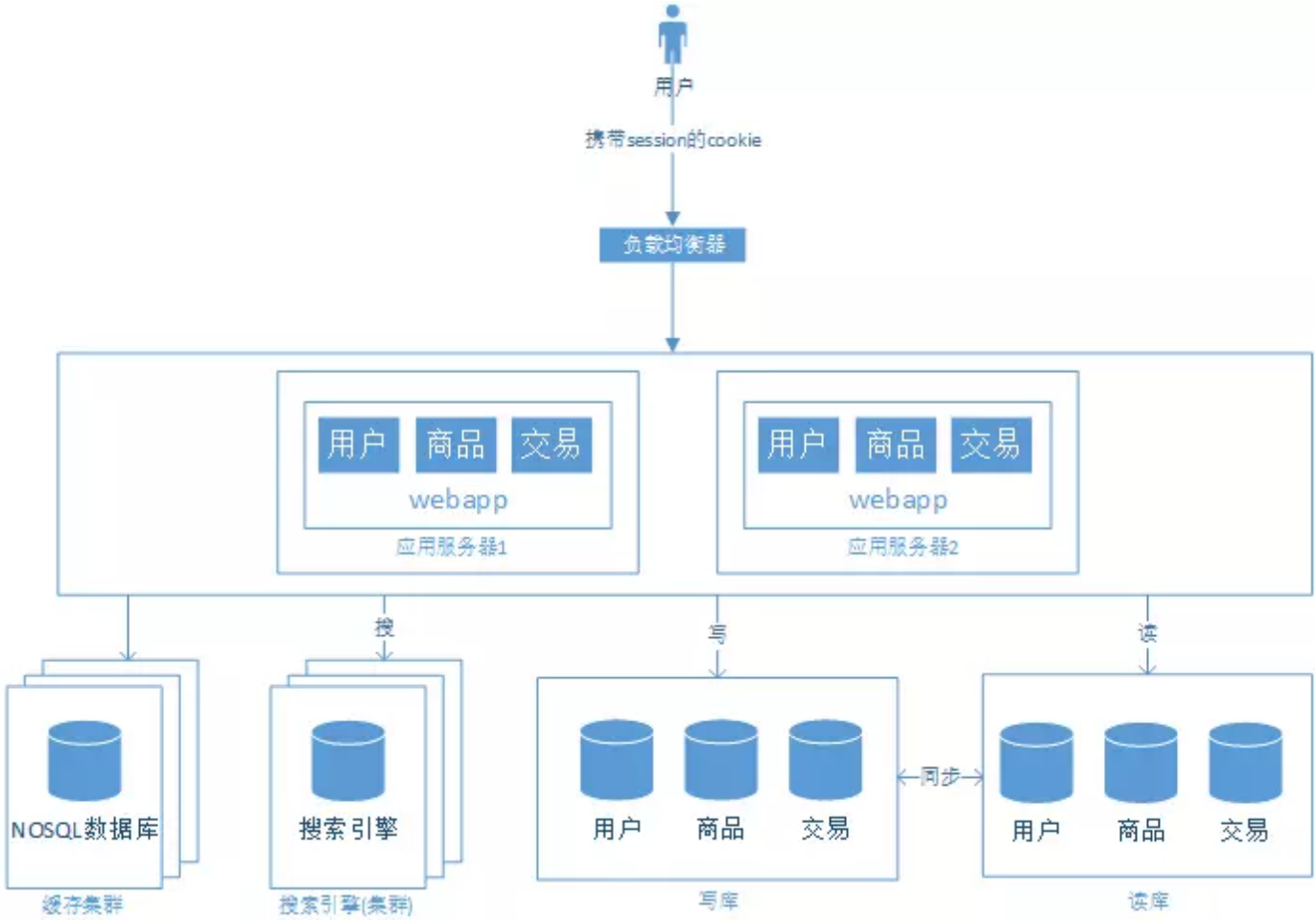
问题：

1. 需要考虑原来跨业务的事务
2. 跨数据库的join

解决问题方案：

1. 我们应该在应用层尽量避免跨数据库的事物，如果非要跨数据库，尽量在代码中控制。
2. 我们可以通过第三方应用来解决，如上面提到的mycat，mycat提供了丰富的跨库join方案，详情可参考mycat官方文档。

垂直拆分后的结构如下：



7.2、数据水平拆分

数据水平拆分就是把同一个表中的数据拆分到两个甚至多个数据库中。产生数据水平拆分的原因是某个业务的数据量或者更新量到达了单个数据库的瓶颈，这时就可以把这个表拆分到两个或更多个数据库中。

优点：

如果我们能客服以上问题，那么我们将能够很好地对数据量及写入量增长的情况。

问题：

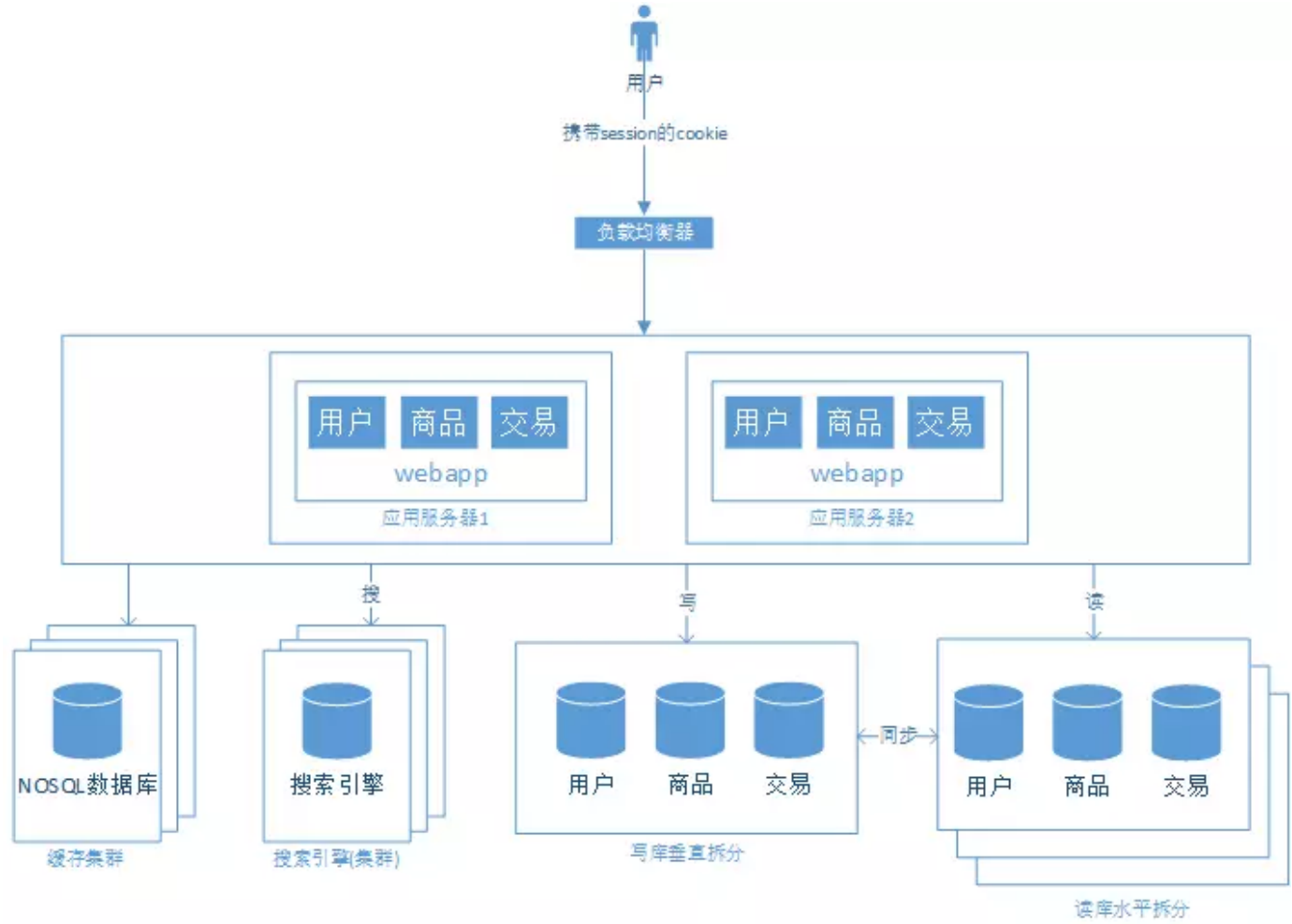
1. 访问用户信息的应用系统需要解决SQL路由的问题，因为现在用户信息分在了两个数据库中，需要在进行数据操作时了解需要操作的数据在哪里。

2. 主键的处理也变得不同，例如原来自增字段，现在不能简单地继续使用了。
3. 如果需要分页，就麻烦了。

解决问题方案：

1. 我们还是可以通过可以解决第三方中间件，如mycat。mycat可以通过SQL解析模块对我们的SQL进行解析，再根据我们的配置，把请求转发到具体的某个数据库。
2. 我们可以通过UUID保证唯一或自定义ID方案来解决。
3. mycat也提供了丰富的分页查询方案，比如先从每个数据库做分页查询，再合并数据做一次分页查询等等。

数据水平拆分后的结构：

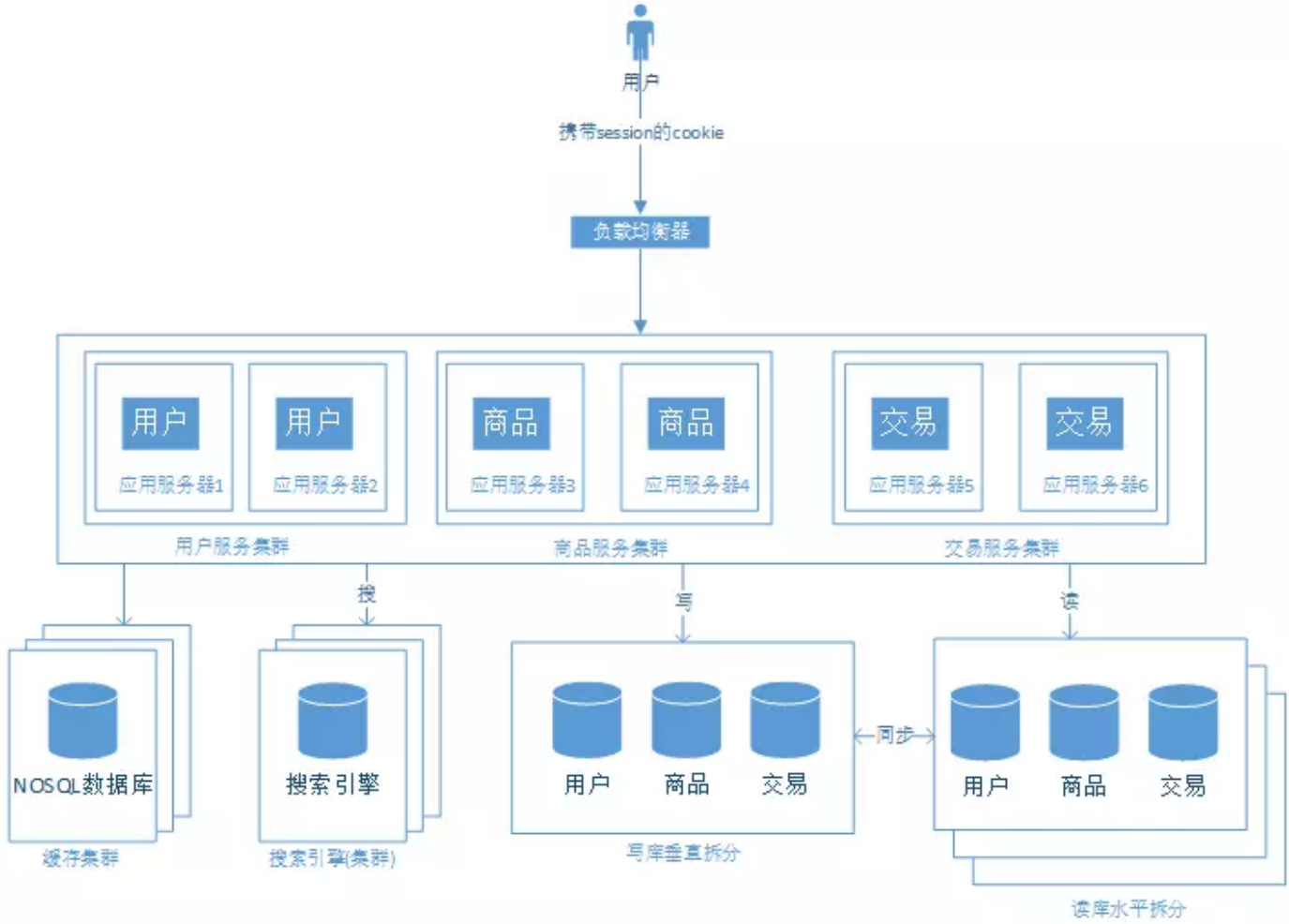


阶段八、应用的拆分

8.1、拆分应用

随着业务的发展，业务越来越多，应用越来越大。我们需要考虑如何避免让应用越来越臃肿。这就需要把应用拆开，从一个应用变为俩个甚至更多。还是以我们上面的例子，我们可以把用户、商品、交易拆分开。变成“用户、商品”和“用户，交易”两个子系统。

拆分后的结构：



问题：

1. 这样拆分后，可能会有一些相同的代码，如用户相关的代码，商品和交易都需要用户信息，所以在两个系统中都保留差不多的操作用户信息的代码。如何保证这些代码可以复用是一个需要解决的问题。

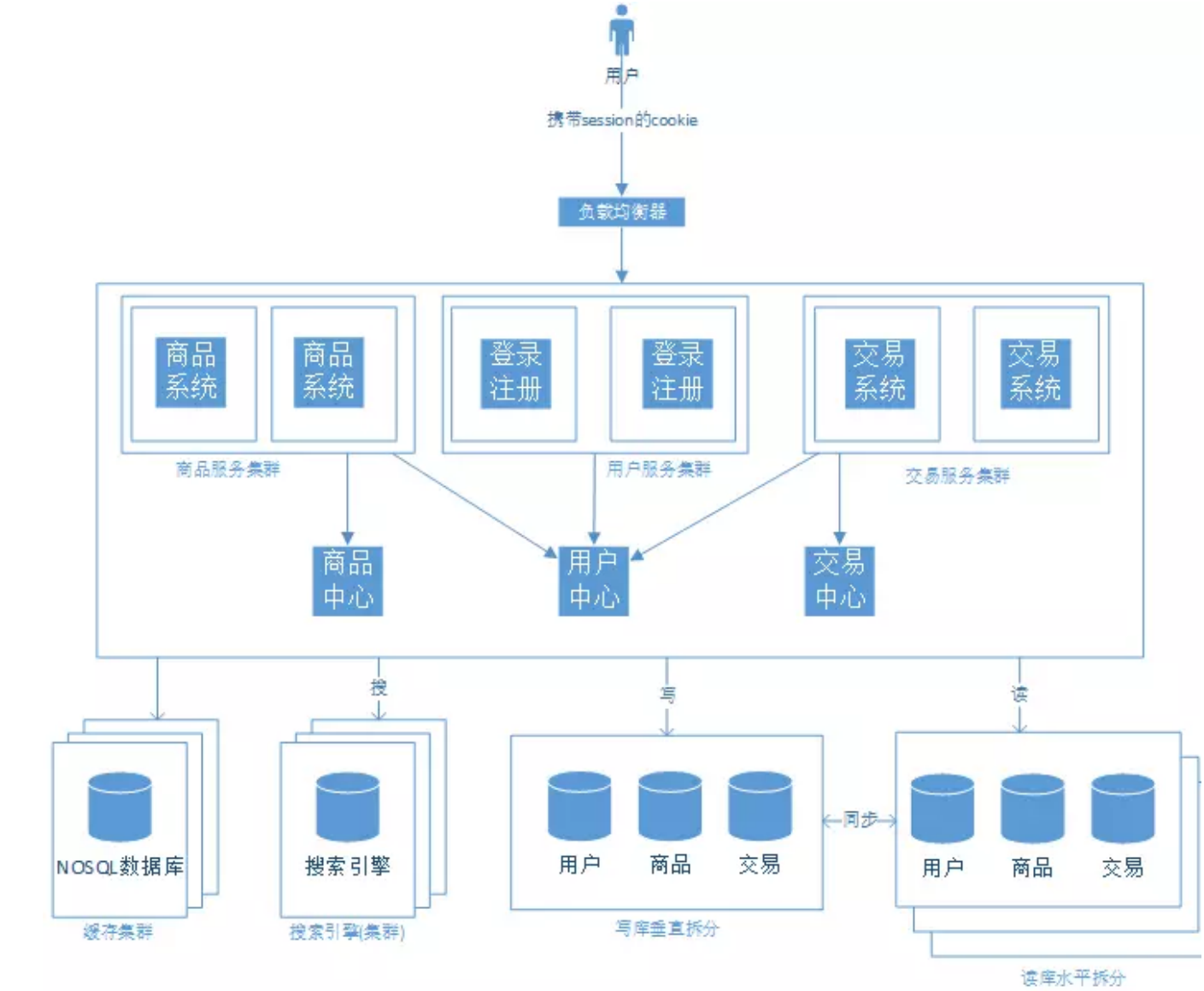
解决问题：

1. 通过走服务化的路线来解决

8.2、走服务化的道路

为了解决上面拆分应用后所出现的问题，我们把公共的服务拆分出来，形成一种服务化的模式，简称SOA。

采用服务化之后的系统结构：



优点：

相同的代码不会散落在不同的应用中了，这些实现放在了各个服务中心，使代码得到更好的维护。

我们把对数据库的交互放在了各个服务中心，让”前端“的web应用更注重与浏览器交互的工作。

问题：

1. 如何进行远程的服务调用

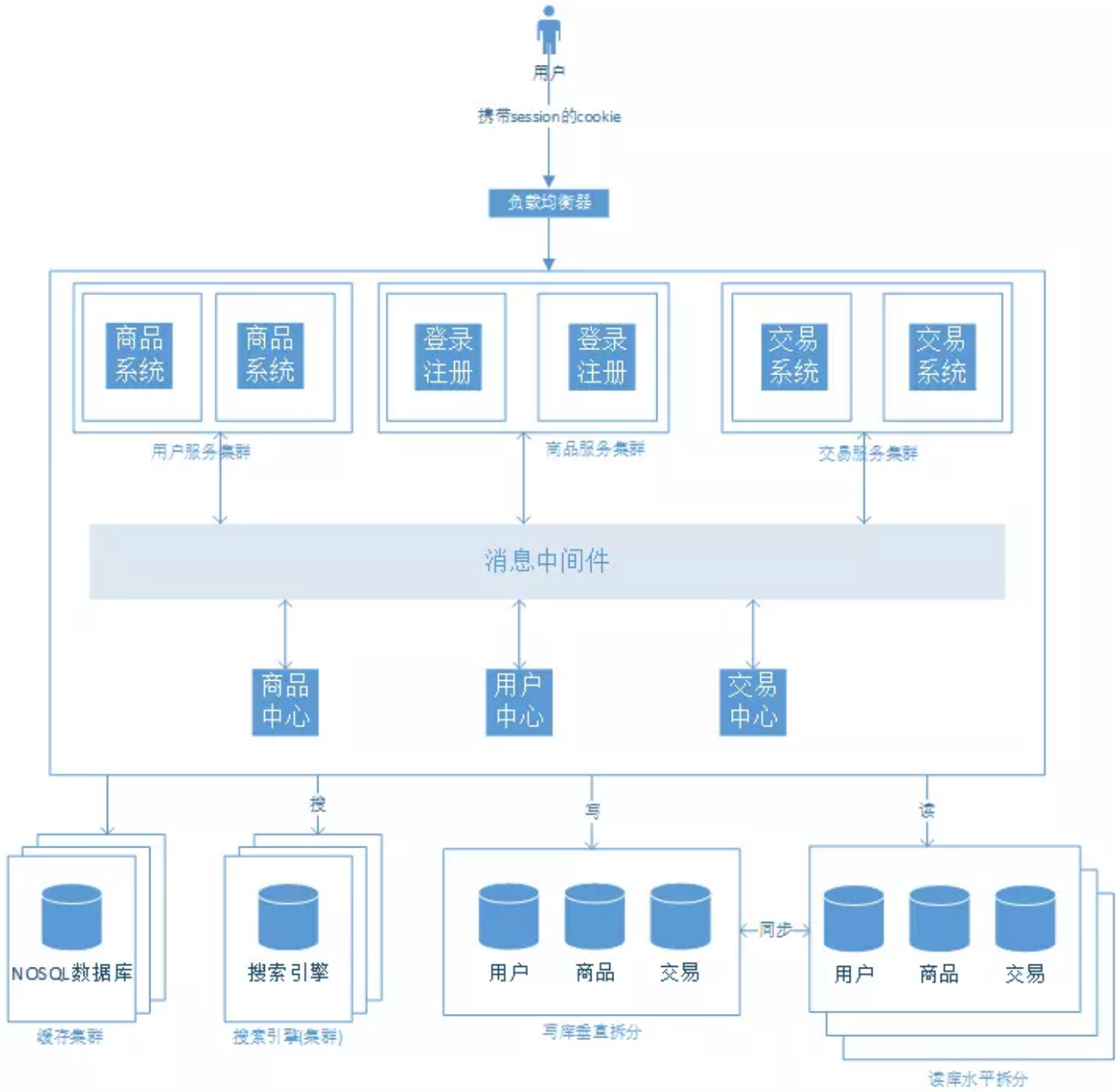
解决方法：

1. 我们可以通过下面的引入消息中间件来解决

阶段九、引入消息中间件

随着网站的继续发展，我们的系统中可能出现不同语言开发的子模块和部署在不同平台的子系统。此时我们需要一个平台来传递可靠的，与平台和语言无关的数据，并且能够把负载均衡透明化，能在调用过程中收集调用数据并分析之，推测出网站的访问增长率等等一系列需求，对于网站应该如何成长做出预测。开源消息中间件有阿里的dubbo，可以搭配Google开源的分布式程序协调服务zookeeper实现服务器的注册与发现。

引入消息中间件后的结构：



十、总结

以上的演变过程只是一个例子，并不适合所有的网站，实际中网站演进过程与自身业务和不同遇到的问题有密切的关系，没有固定的模式。只有认真的分析和不断地探究，才能发现适合自己网站的架构。

【程序猿必备小程序】程序员技能包

最近文章：

rpc思维导图，让rpc不再难懂

架构师必备的几项技能（上）

架构师必备的几项技能（下）

快速回顾，浅谈mvc思想

阅读原文