人人都要学的架构思维

作者：老 K

作者介绍：老 K，“技术领导力”公众号作者，文出过畅销书、武做过 CTO，被生活逼得一身才华。

支持媒体：“技术领导力”、“BAT 架构”公众号：

技术领导力

BAT 架构

目录

1 中台架构的落地方法与实践

架构的本质，是利用分、合、打散、重组等技术手段，对系统进行有序化重构，以达到减少系统“熵”的过程，使系统得以不断进化。即便你不需要在一线撸代码，多了解一些架构原则和思想，感受经典架构背后的哲学与思考，相信对你技术视野的提升大有益处。下面我们先来学习中台架构。

中台架构理念，是阿里巴巴提出，并且发扬光大的一种企业架构治理方法论。阿里巴巴中间件团队，给中台架构做过一个定义：

“中台架构，是将企业的核心能力随着业务不断发展以数字化形式沉淀到平台，形成以服务为中心，由业务中台和数据中台构建起数据闭环运转的运营体系，供企业更高效的进行业务探索和创新, 实现以数字化资产的形态构建企业核心差异化竞争力。”

中台架构理念的底层逻辑是平台思维。平台是指连接两个以上的特定群体，为他们提供互动交流机制，满足所有群体的需求，并从中赢利的商业模式。马歇尔在《平台革命》一书中，对平台进行过描述：“匹配用户，通过商品、服务或社会货币的交换为所有参与者创造价值。”有兴趣的读者，可以深入研究下平台的理念。

阿里中台主要体现为由业务中台和数字中台构成的双中台，并肩扛起了所有前台业务。业务中台将后台资源进行抽象包装整合，转化为前台友好的可重用共享的核心能力，实现了后端业务资源到前台易用能力的转化。

数据中台从后台及业务中台将数据导入，完成海量数据的存储、计算、产品化包装过程，构成企业的核心数据能力，为前台基于数据的定制化创新和业务中台基于数据反馈的持续演进提供了强大支撑。

业务中台与数据中台相辅相成、互相支撑，一起构建起了战场上强大的后方炮火群和雷达阵。如图 4-1 所示。

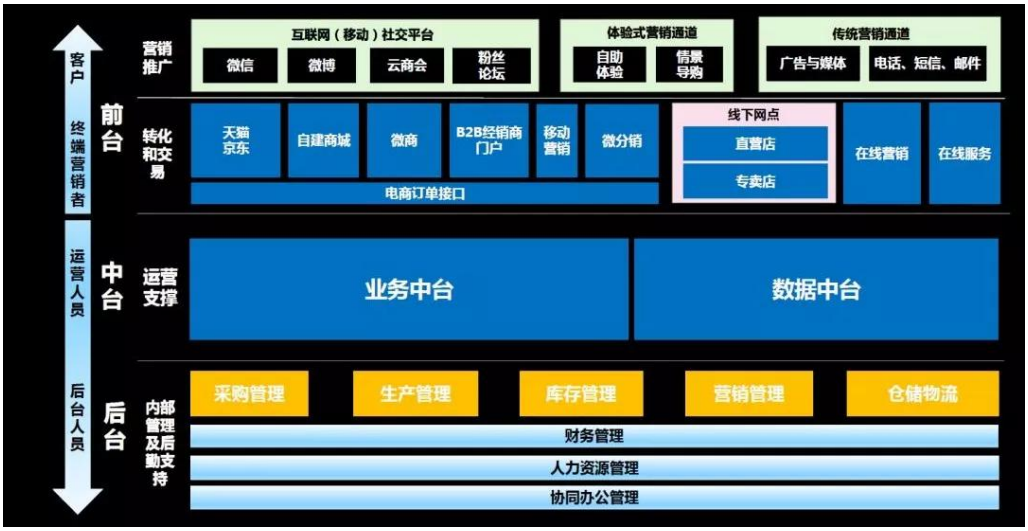

图 4-1 企业中台架构图

中台架构解决什么问题？简单的说，中台架构是解决企业复杂生态协作问题的方法论。中台架构的目标，是通过中台治理的理念和方法，让企业降低成本，提升协作效率。是通过制定符合企业实际情况的标准和规范，来做具体的项目实施。中台治理方法的原则：集中管控，分布式执行。

中台架构的定位问题。定位就是清楚的告诉别人，你有什么？你要什么？以及你不要什么？

以阿里巴巴的业务中台为例，业务中台需要收敛一些基础的业务服务，如会员、商品、交

易、营销和结算等。这些基础的服务会被整个电商业务使用，所以统一管理是很有必要的。

那么业务中台还需要什么？因为中台的目标是要向上层业务提供这些基础的服务，那自然必须能够清楚地描述自己到底有哪些服务、数据和功能，我们可以把它统称为能力。所以还需要能够定义能力（标准和规范）、能力的发现、能力的注册、能力的列表以及能力的评价和更新机制等。

业务中台也不是什么都做，除了有基本的基础服务和服务能力外，还要定义中台的边界。业务中台有它基本的工作范围，它需要能够对接能力，同时又服务好能力使用方，而自己并不负责实现具体的业务。

中台建设方法论的核心要素。前面提到了，中台架构是解决企业复杂生态协作问题的方法论。这个方法论要解决的是企业复杂生态协作的治理问题。

我们可以借鉴建筑行业、互联网行业的基础设施建设的思路。以移动互联网为例，根据 QuestMobile 数据显示，移动互联网月活用户规模达到 11.38 亿。那么，如此庞大的生态体系是如何运作的呢？

我们观察到，任何手机，都有 sim 卡；每个手机的制造工艺，要求满足 3G、4G、5G 协议；无论是联通还是电信，运营商集中控制整个网络。从移动互联网的生态治理可以看到，由三方面因素共同解决一个复杂生态的协作问题：

1、协议标准、运行机制。如 3G、4G、5G 等通信协议。

2、满足标准的分布式执行单元。如每个人的手机设备。

3、中心化的控制单元。如联通、电信等中心化的运营商集中管控。

以上三方面因素，就是中台治理方法论的核心要素。

如何判断一个企业需不需要上中台？企业要不要上中台，不能盲目跟风。别人家上了我也要上，你不清楚别人的战略布局、核心竞争力、战术打法，盲目去学，你不死谁死？

道理都懂，那么有没有一种方法来判断一个企业需不需要上中台？我们参照“美国国家标准技术研究院”的技术战略选择分析流程图，来给企业来做个诊断，如图 4-2 所示：

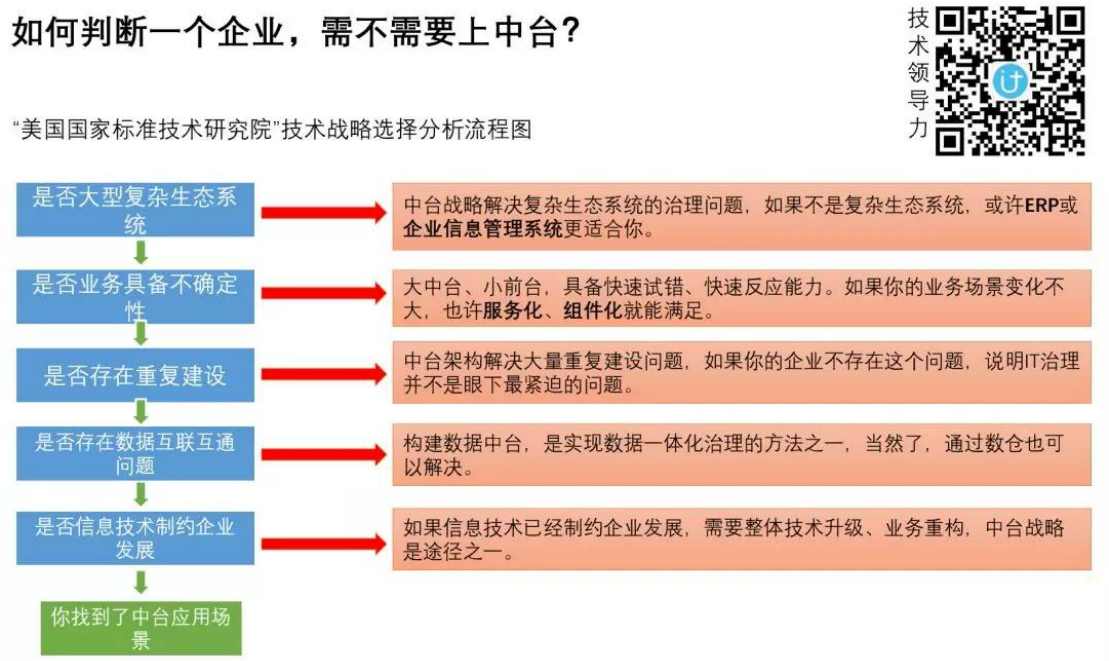

图 4-2 中台战略选择分析流程图

图中列出了 5 个条件判断：

1）是否大型复杂生态系统。如果企业的业态不是大型复杂生态型企业，也许企业信息化系统、ERP 就可以解决企业 IT 治理的问题。何为复杂生态型企业？国内如：BATJ、海尔、华为、小米等跨行业的大型集团公司，都属于这类型企业。

2）是否业务具备不确定性。即市场环境变化快，如互联网行业，以及产业互联网相关行业，都属于这类型行业。

3）是否存在低水平重复建设。先决条件是企业体量够大，不管是自建技术团队还是跟外部第三方公司合作，是否存在重复建设，如整个企业光 CRM 就有 5 套，还不是一家公司的产品，就属于重复建设。

4）是否存在数据互联互通问题。即由于事业部制的组织架构，形成了部门墙，数据和系统也是烟囱式的，阿里的业务中台、数据中台解决的就是这样的问题。

5）是否信息技术制约企业发展。尽管企业的 IT 建设一直在持续，但是在当今产业互联网时代、人工智能时代，企业的发展已经受到严重阻碍，也许你该引入中台架构进行变革了。

以上 5 个条件，任意满足 3 个及以上，我们就认为这个企业适合做中台战略升级。

企业中台建设如何实施？分为以下三个部分：

1）各个中台板块实施难易度分析。首先，我们来看，技术中台、数据中台、业务中台、组织中台，在整个中台建设实施过程中的难易度，如图 4-3 所示：

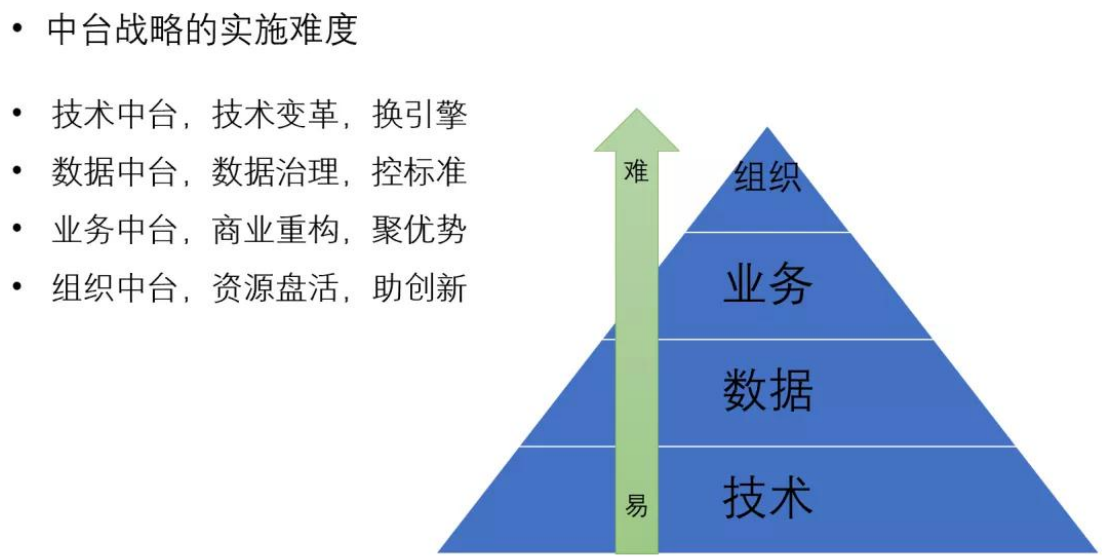

图 4-3 中台建设难易度分析

企业中台建设的技术、数据、业务、组织，建设难度从易到难，总结下来就是：

技术中台，技术变革，换引擎。

数据中台，数据治理，控标准。

业务中台，商业重构，聚优势。

组织中台，资源盘活，助创新。

2）业务中台建设路径。我们借鉴阿里对外部企业输出的中建设路径：

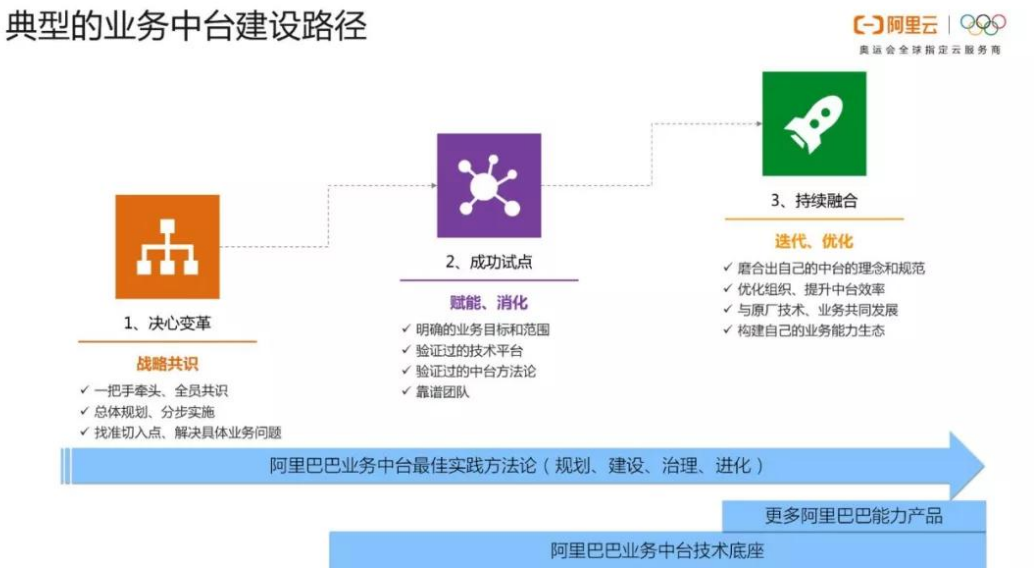

图 4-4 阿里巴巴中台建设路径

决心变革。企业内达成战略共识，一把手牵头，做总体规划、分步实施，找准切入点，解决具体业务问题。

成功试点。通过分析调研，明确业务目标和范围，完成技术平台引入、中台建设方法论宣导，进行试点，梳理标杆，积累经验。

持续融合。总结出适合企业自身的理念和规范，优化组织、提升中台效率。

3）企业中台升级四个方面：

图 4-4 阿里巴巴中台建设路径

阿里建议企业实施中台战略的 4 个升级：

1、战略升级。通过中台建设，落地企业数字化战略。

2、组织升级。组织架构需要与中台架构相匹配，根据企业实际情况优化组织效率。

3、流程升级。将企业现有流程进行梳理，优化及固化企业流程，提升企业运作效率。

4、技术升级。通过互联网技术，对企业基础技术设施进行升级，降本增效。

企业中台建设有哪些坑？

第一，组织架构不匹配中台架构。观察阿里 HR 组织中台，从结构来看，跟阿里的业务中台架构很像，这就是组织架构与业务架构要相匹配。

第二，需求治理缺失，大量浪费资源。建立以价值为导向的需求治理机制，以价值为导向的需求治理机制，其目的是把有限的开发资源，投入到更有价值的项目上，该机制分成几个部分，如图 4-5。

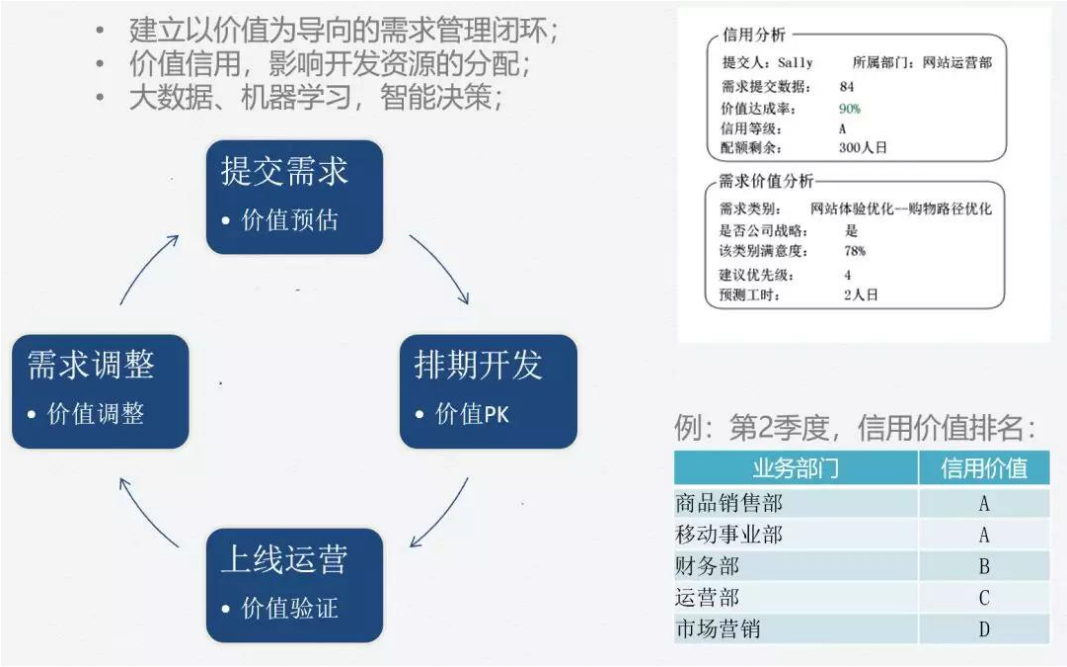

图 4-5 需求治理机制

建立需求管理闭环。以价值为导向的需求治理机制，是在需求提报环节，提出该需求的价值预估，即这个需求将给业务带来什么样的价值，这些价值要能够量化、能够追踪。比如，一个结算系统需求，价值是：提升客户对账效率 20%，上线后会来验证效果，看看是否达到预期。

第三，中台部渐渐脱离一线业务，成为鸡肋中台。

中台化的难度是：技术<数据<业务<组织。因为越往后，越需要业务团队的介入，越要有业务认知才能做好中台。而中台团队，天然就是离业务远的。

各公司跟中台团队的协作，都存在很大的低效问题，一线的前台业务团队，要反复与中台沟通，才能讲清楚业务需求的背景，在大公司，跨部门协作都会徒增成本。另外对于中台业务团队来说，长期离业务很远，没有成长性，在判断需求时也不准确，很难获得前台业务团队的信任。

久而久之，就变成了中台团队纯支撑、被动接需求的状态，成为“鸡肋中台”。更大的问题在于，中台团队的价值就是整合需求、抽象能力的，在对用户和业务不够了解的前提下，做起来就会特别吃力。如果是多个前台业务的需求有了冲突，中台团队未必能做好协调，多方要反复沟通扯皮。

第四，一把手不重视中台建设。导致的问题是，未把中台建设上升到战略高度，资源投入不足，也没有决心进行组织架构变革，导致中台建设不能持之以恒。

第五，幻想着中台战略一步到位。没有中长期投入的准备，想着怎么短平快的进行建设，或者把中台战略当成向公司要资源的新理由。

第六，认为中台包治百病。这类企业把业务增长慢、企业运作效率低、组织架构臃肿、缺乏创新等问题，都想用一套方法来一次性解决掉，结果问题依然存在。中台建设，首先要定义问题，越聚焦越能够产生好的效果。

总之，企业是否要上中台，要根据企业的具体情况做分析，可根据上文“中台战略选择分析流程图”进行判断，不要盲目跟风，认为别人家上了中台，自己也要上。中台是一剂良药，对症下药能治顽疾，用药不当，会送了你的命。

2 CAP 理论在大型互联网系统中的应用

在计算机领域，如果是初入行就算了，如果是多年的老码农还不懂 CAP 定理，那就真的说不过去了。CAP 可是每一名技术架构师都必须掌握的基础原则啊。

现在只要是稍微大一点的互联网项目都是采用分布式结构了，一个系统可能有多个节点组成，每个节点都可能需要维护一份数据。那么如何维护各个节点之间的状态，如何保障各个节点之间数据的同步问题就是大家急需关注的事情了。

CAP 定理是分布式系统中最基础的原则。所以理解和掌握了 CAP，对系统架构的设计至关重要。

CAP 定理（CAP theorem）又被称作布鲁尔定理(Brewer’s theorem)，是加州大学伯克利分校的计算机科学家--埃里克·布鲁尔（Eric Brewer），在 2000 年的 ACM PODC 上提出的一个猜想。2002 年，麻省理工学院的赛斯·吉尔伯特（Seth Gilbert）和南希·林奇（Nancy Lynch）发表了布鲁尔猜想的证明，使之成为分布式计算领域公认的一个定理。

CAP 定理，指的是在一个分布式系统（指互相连接并共享数据节点的集合）中，当涉及读写操作时，只能保证一致性（Consistence）、可用性（Availability）、分区容错性（Partition Tolerance）三者中的两个，另外一个必须被牺牲，如图 4-6 所示。

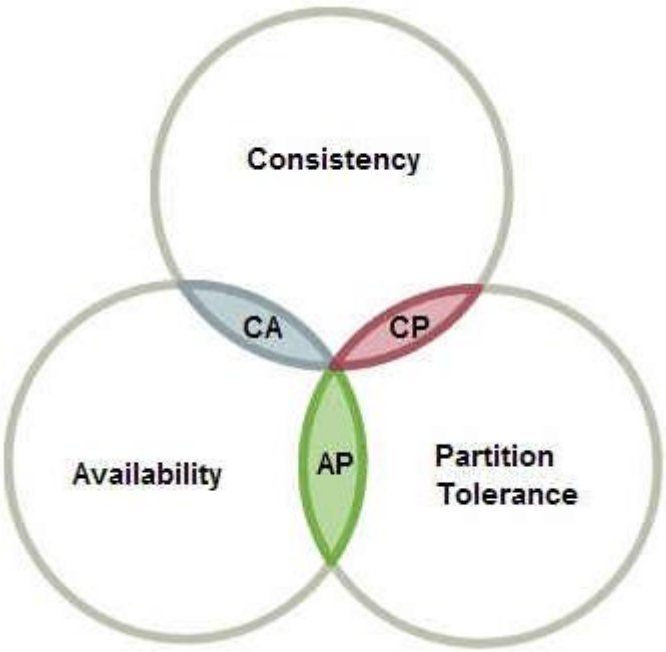

图 4-6 CAP 理论

CAP 分别表示的含义：

一致性（Consistency）， 对某个指定的客户端来说，读操作保证能够返回最新的写操作结果。一致性又可分为：弱一致性、强一致性、最终一致性，感兴趣可以查看相关文档。

可用性（Availability），非故障的节点在合理的时间内返回合理的响应（不是错误和超时的响应）。可用性模式可分成：工作到备用切换（Active-passive）、双工作切换（Active-active）

分区容错性（Partition Tolerance），当出现网络分区后，系统能够继续履行职责。

CAP 定理在互联网大型分布式系统中的应用

虽然 CAP 理论定义是三个要素中只能取两个，但放到分布式环境下来思考的话，我们会发现必须选择 P(分区容错)要素，因为网络本身无法做到 100%可靠，有可能出故障，所以分区是一个必然的选项。

如果我们选择了 CA 而放弃了 P，那么当发生分区现象时，为了保证 C，系统需要禁止写入，当有写入请求时，系统返回 error，这又和 A 冲突了，因为 A 要求返回 no error 和 no timeout。因此，分布式系统理论上不可能选择 CA 架构，只能选择 CP 或者 AP 架构。

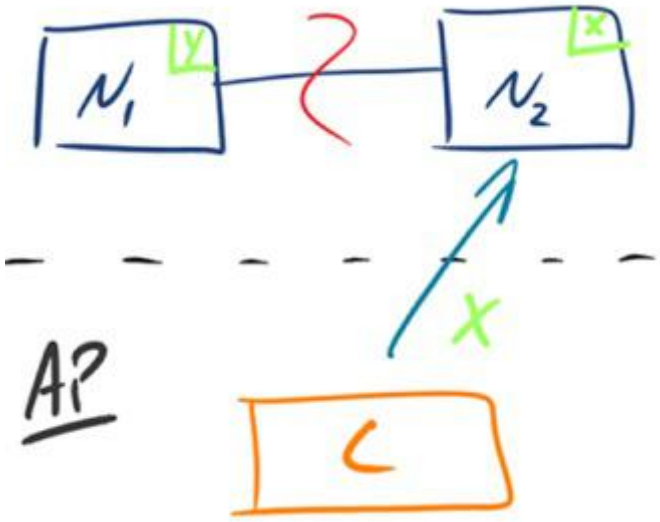

图 4-7 CP - Consistency/Partition Tolerance

如图 4-7 所示，为了保证一致性，当发生分区现象后，N1 节点上的数据已经更新到 y，但由于 N1 和 N2 之间的复制通道中断，数据 y 无法同步到 N2，N2 节点上的数据还是 x。

这时客户端 C 访问 N2 时，N2 需要返回 Error，提示客户端面 C：“系统现在发生错误”，这种处理方式违背了可用性（Availability）的要求，因此 CAP 三者只能满足 CP。

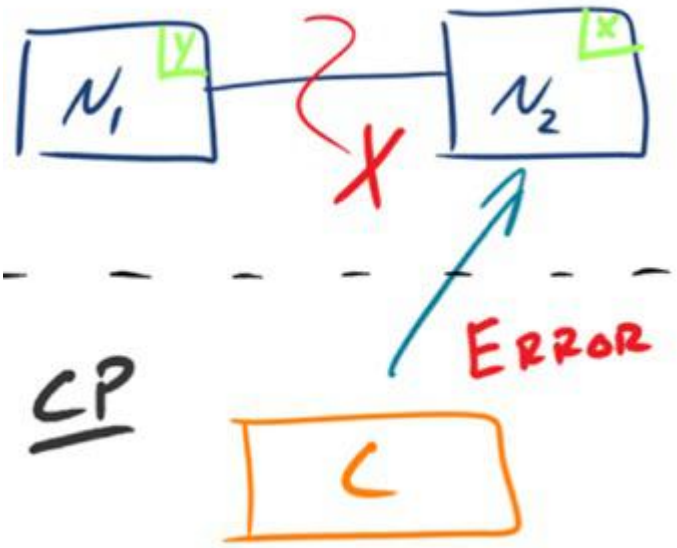

图 4-8 AP - Availability/Partition Tolerance

如图 4-8 所示，为了保证可用性，当发生分区现象后，N1 节点上的数据已经更新到 y，但由于 N1 和 N2 之间的复制通道中断，数据 y 无法同步到 N2，N2 节点上的数据还是 x。

这时客户 C 访问 N2 时，N2 将当前自己拥有的数据 x 返回给客户端 C 了，而实际上当前最新的数据已经是 y 了，这就不满足一致性（Consistency）的要求了，因此 CAP 三者只能满足 AP。

注意：这里 N2 节点返回的 x，虽然不是一个“正确”的结果，但是一个“合理”的结果，因为 x 是旧的数据，并不是一个错乱的值，只是不是最新的数据而已。

CAP 定理的应用，有哪些注意事项？

了解了 CAP 定理后，对于开发者而言，当我们构建服务的时候，就需要根据业务特性进行权衡考虑，哪些点是当前系统可以取舍的，哪些是应该重点保障的。

CAP 关注的粒度是数据，而不是整个系统。

C 与 A 之间的取舍可以在同一系统内以非常细小的粒度反复发生，而每一次的决策可能因为具体的操作，乃至因为牵涉到特定的数据或用户有所不同。

以一个商家管理系统为例，商家管理系统包含商家账号数据（商家 ID、密码）、商家信息数据（行业类别、公司规模、营收规模等）。通常情况下，商家账号数据会选择 CP，而商家信息数据会选择 AP，如果限定整个系统为 CP，则不符合用户信息的应用场景；如果限定整个系统为 AP，则又不符合商家账号数据的应用场景。

所以在 CAP 理论落地实践时，我们需要将系统内的数据按照不同的应用场景和要求进行分类，每类数据选择不同的策略（CP 或 AP），而不是直接限定整个系统所有数据都是同一策略。

CAP 是忽略网络延迟的。

这是一个非常隐含的假设，布鲁尔在定义一致性时，并没有将延迟考虑进去。即当事务提交时，数据能够瞬间复制到所有节点。但实际情况下，尤其是互联网架构之下，从节点 A 复制数据到节点 B，总是需要花费一定时间的。如果在相同机房可能是几毫秒，如果跨机房，可能

是几十毫秒。这也就是说，CAP 理论中的 C 在实践中是不可能完美实现的，在数据复制的过程中，节点 A 和节点 B 的数据并不一致。

正常运行情况下，不存在 CP 和 AP 的选择，可以同时满足 CA。

CAP 理论告诉我们分布式系统只能选择 CP 或者 AP，但其实这里的前提是系统发生了“分区”现象。如果系统没有发生分区现象，也就是说 P 不存在的时候（节点的网络连接一切正常），我们就没有必要放弃 C 或者 A，应该 C 和 A 都可以保证，这就要求架构设计的时候即要考虑分区发生时选择 CP 还是 AP，也要考虑分区没有发生时如何保证 CA。

这里我们还以用户管理系统为例，即使是实现 CA，不同的数据实现方式也可能不一样：用户账号数据可以采用“消息队列”的方式来实现 CA，因为消息队列可以比较好地控制实时性，但实现起来就复杂一些；而用户信息数据可以采用“数据库同步”的方式来实现 CA，因为数据库的方式虽然在某些场景下可能延迟较高，但使用起来简单。

放弃并不等于什么都不做，需要为分区恢复后做准备。

CAP 理论告诉我们三者只能取两个，需要“牺牲”（sacrificed）另外一个，这里的“牺牲”是有一定误导作用的，因为“牺牲”让很多人理解成什么也不做。实际上，CAP 理论的“牺牲”只是说在分区过程中我们无法保证 C 或者 A，但并不意味着什么都不做。分区期间放弃 C 或者 A，并不意味着永远放弃 C 和 A，我们可以在分区期间进行一些操作，从而让分区故障解决后，系统能够重新达到 CA 的状态。

最典型的就是在分区期间记录一些日志，当分区故障解决后，系统根据日志进行数据恢复，使得重新达到 CA 状态。

CAP 是非常重要的架构理论，有志于成为架构师的朋友，对于 CAP、ACID、BASE 等定理，一定要有比较深的理解和认识，把基础打扎实。

3 “缓存”如何应对亿级流量峰值

许多大型互联网系统，如：电商、社交、新闻等 App 或网站，动辄日活千万甚至上亿，每分钟的峰值流量也在数十万以上，架构上如何应对如此高的流量峰值？

本节给大家介绍，通过使用“缓存”技术来给系统减压。流量峰值对系统带来的主要危害在于，它会瞬间造成大量对磁盘数据的读取和搜索，通常的数据源是数据库或文件系统，当数据访问量次数增大的时候，过多的磁盘读取可能会最终成为整个系统的性能瓶颈，甚至是压垮整个数据库，导致系统卡死、服务不可用等严重后果。

常规的应用系统中，我们通常会在需要的时候对数据库进行查找，因此系统的大致结构如下所示：

图 4-9 常规应用系统

当数据量较高的时候，需要减少对于数据库里面的磁盘读写操作，因此通常都会选择在业务系统和数据库之间加入一层缓存从而减少对数据库方面的访问压力，如图 4-10 所示。

图 4-10 增加了缓存层的应用系统

当数据量较高的时候，需要减少对于数据库里面的磁盘读写操作，因此通常都会选择在业务系统和 MySQL 数据库之间加入一个缓存层，从而减少对数据库方面的访问压力。

缓存在实际场景的应用当中并非这么简单，下边我们来通过几个比较经典的缓存应用场景来列举一些问题：

1）缓存和数据库之间数据一致性问题

常用于缓存处理的机制有以下几种：

Cache Aside 模式。这种模式处理缓存通常都是先从数据库缓存查询，如果缓存没有命中则从数据库中进行查找。

这里面会发生的三种情况如下：

缓存命中：当查询的时候发现缓存存在，那么直接从缓存中提取。

缓存失效：当缓存没有数据的时候，则从 database 里面读取源数据，再同步到 cache 里面去。

缓存更新：当有新的写操作去修改 database 里面的数据时，需要在写操作完成之后，让 cache 里面对应的数据失效，作缓存同步。

这种 Cache aside 模式，通常是我们在实际应用开发中最为常用到的模式。但是并非说这种模式的缓存处理就一定能做到完美。

这种模式下依然会存在缺陷。比如，一个是读操作，但是没有命中缓存，然后就到数据库中取数据，此时来了一个写操作，写完数据库后，让缓存失效，然后之前的那个读操作再把老的数据放进去，所以会造成脏数据。

分布式环境中要想完全的保证数据一致性，是一件极为困难的事情，我们只能够尽可能的降低这种数据不一致性问题产生的情况。

Read Through 模式。是指应用程序始终从缓存中请求数据。如果缓存没有数据，则它负责使用底层提供程序插件从数据库中检索数据。检索数据后，缓存会自行更新并将数据返回给调用应用程序。

使用 Read Through 有一个好处，我们总是使用 key 从缓存中检索数据，调用的应用程序不知道数据库，由存储方来负责自己的缓存处理，这使代码更具可读性，代码更清晰。

但是这也有相应的缺陷，开发人员需要编写相关的程序插件，增加了开发的难度性。

Write Through 模式。和 Read Through 模式类似，当数据发生更新的时候，先去 Cache 里面进行更新，如果命中了，则先更新缓存再由 Cache 方来更新 database。如果没有命中的话，就直接更新 Cache 里面的数据。

Write Behind Caching 模式。这种模式通常是先将数据写入到缓存里面，然后再异步的写入到 database 中进行数据同步。

这样的设计既可以直接的减少我们对于数据的 database 里面的直接访问，降低压力，同时对于 database 的多次修改可以进行合并操作，极大的提升了系统的承载能力。

但是这种模式处理缓存数据具有一定的风险性，例如说当 cache 机器出现宕机的时候，数据会有丢失的可能。

缓存的高并发场景中存在哪些问题？

缓存过期后将尝试从后端数据库获取数据，这是一个看似合理的流程。

但是，在高并发场景下，有可能多个请求并发的去从数据库获取数据，对后端数据库造成极大的冲击，甚至导致“雪崩”现象。

此外，当某个缓存 key 在被更新时，同时也可能被大量请求在获取，这也会导致一致性的问题。那如何避免类似问题呢？我们会想到类似“锁”的机制，在缓存更新或者过期的情况下，先尝试获取到锁，当更新或者从数据库获取完成后再释放锁，其他的请求只需要牺牲一定的等待时间，即可直接从缓存中继续获取数据。

缓存穿透问题。缓存穿透也称为“击穿”。很多朋友对缓存穿透的理解是：由于缓存故障或者缓存过期导致大量请求穿透到后端数据库服务器，从而对数据库造成巨大冲击。

这其实是一种误解。真正的缓存穿透应该是这样的：

在高并发场景下，如果某一个 key 被高并发访问，没有被命中，出于对容错性考虑，会尝试去从后端数据库中获取，从而导致了大量请求达到数据库，而当该 key 对应的数据本身就是空的情况下，这就导致数据库中并发的去执行了很多不必要的查询操作，从而导致巨大冲击和压力。

可以通过下面的几种常用方式来避免缓存穿透问题：

1)缓存空对象

对查询结果为空的对象也进行缓存，如果是集合，可以缓存一个空的集合（非 null），如果是缓存单个对象，可以通过字段标识来区分。这样避免请求穿透到后端数据库。同时，也需要保证缓存数据的时效性。

这种方式实现起来成本较低，比较适合命中不高，但可能被频繁更新的数据。

2)单独过滤处理

对所有可能对应数据为空的 key 进行统一的存放，并在请求前做拦截，这样避免请求穿透到后端数据库。

这种方式实现起来相对复杂，比较适合命中不高，但是更新不频繁的数据。

缓存颠簸问题。也被成为“缓存抖动”，可以看做是一种比“雪崩”更轻微的故障，但是也会在一段时间内对系统造成冲击和性能影响。一般是由于缓存节点故障导致。业内推荐的做法是通过一致性 Hash 算法来解决。

缓存的雪崩。是指由于缓存的原因，导致大量请求到达后端数据库，从而导致数据库崩溃，整个系统崩溃，发生灾难。导致这种现象的原因有很多种，上面提到的“缓存并发”，“缓存穿透”，“缓存颠簸”等问题，其实都可能会导致缓存雪崩现象发生。

这些问题也可能会被恶意攻击者所利用。还有一种情况，例如某个时间点内，系统预加载的缓存周期性集中失效了，也可能会导致雪崩。为了避免这种周期性失效，可以通过设置不同的过期时间，来错开缓存过期，从而避免缓存集中失效。

从应用架构角度，我们可以通过限流、降级、熔断等手段来降低影响，也可以通过多级缓存来避免这种灾难。

此外，从整个研发体系流程的角度，应该加强压力测试，尽量模拟真实场景，尽早的暴露问题从而防范。

缓存技术，是大型互联网系统架构中常用的一种技术，在设计缓存架构的过程中，要根据业务场景进行针对性的设计，同步避免缓存延迟、脏数据、缓存雪崩等问题，提升系统的高可用性，健壮性。

4 “消息”，如何给复杂系统解耦

大型互联网系统，业务逻辑较为复杂，或者由于海量、高并发等场景增加了技术架构的复杂性，这时候需要对复杂系统做解耦，这里就要用到消息中间件来给系统做解耦。首先，我们来了解几个概念。

耦合性(Coupling)，也叫耦合度，是对模块间关联程度的一个度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。一般来说，模块间联系越多，其耦合性越强，同时表明其独立性越差。软件设计中通常用耦合度和内聚度，作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

消息中间件，是利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。消息队列主要由以下作用：异步处理、流量削峰、应用解耦。

异步处理。生产端不需要等待消费端响应，直接返回，提高了响应时间和吞吐量。

流量削峰。打平高峰期的流量，消费端可以根据自己的速度处理，同时也无需在高峰期增加太多资源，提高资源利用率。

应用解耦。生产端和消费端不需要相互依赖。

下面分别对消息中间件的应用场景做介绍：

一、异步处理

场景说明：用户下订单后，需要发邮件和短信进行确认,传统的做法有两种，一是串行的方式；二是并行的方式。

1)串行方式：将订单信息写入数据库后，发送邮件，再发送短信，以上三个任务全部完成后才返回给客户端。 这里有一个问题是,邮件、短信并不是必须的，它只是一个通知，而这种做法让客户端等待没有必要等待的事件。

2)并行方式：将订单信息写入数据库后，发送邮件的同时，发送短信，以上三个任务完成后，返回给客户端，并行的方式能提高处理的时间。

假设三个业务节点分别使用 50ms，串行方式使用时间 150ms，并行使用时间 100ms。虽然并行已经提高的处理时间，但是，前面说过，邮件和短信对用户正常使用网站没有任何影响，客户端没有必要等待其发送完成才显示下单成功，应该是写入数据库后就返回。

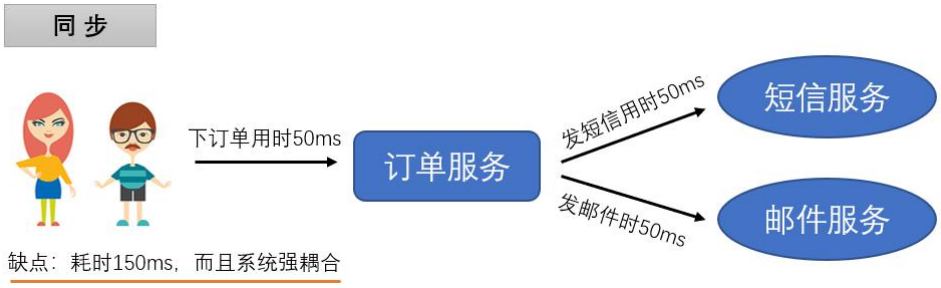

图 4-11 同步方式

图 4-11，如果采用一般的同步方式处理，系统性能会很慢。

3)消息队列。

图 4-12 线程池-异步方式

图 4-13 消息中间件-异步方式

引入消息队列后，把发送邮件，短信等不是必须的业务逻辑，进行异步处理。

由此可以看出，引入消息队列后，用户的响应时间就等于写入数据库的时间+写入消息队列的时间，引入消息队列后处理后，响应时间是串行的 3 倍，是并行的 2 倍。

二、应用解耦

场景：某宝双 11 购物狂欢节，用户下单后，订单系统需要通知库存系统，做锁库存操作，传统的做法就是订单系统调用库存系统的接口。

图 4-14 下单异步处理

这种做法有一个缺点：当库存系统出现故障时，订单就会失败。订单系统和库存系统高耦合，解决的办法是引入消息队列。

订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功。

库存系统：订阅下单的消息，获取下单消息，进行库操作。

如果库存系统出现故障，消息队列也能保证消息的可靠投递，不会导致消息丢失。

三、流量削峰

流量削峰，一般在秒杀系统中应用广泛。

场景：某东购物网站秒杀活动，一般会因为流量过大，导致应用挂掉，为了解决这个问题，可以在应用前端加入消息队列。

图 4-15 流量削峰

作用：

1）可以控制活动人数，超过一定阀值的订单直接丢弃。

2）可以缓解短时间的高流量压垮系统。

改造后的处理流程如下：

用户的请求，服务器收到之后，首先写入消息队列，加入消息队列长度超过最大值，则直接抛弃用户请求或跳转到错误页面。

秒杀业务根据消息队列中的请求信息，再做后续处理。

以上，介绍了通过消息中间件的方式，给复杂系统解耦，通过消息中间件的引入，还可以帮助系统提升响应能力、提高可用性，消息中间件是当今架构技术中，很常用的一种工具，希望大家能够花时间去研究和实践。

5 微服务，大型复杂系统的架构实践

微服务架构已经是大型复杂系统的首选架构方式，我们以大型电商系统应用场景为例，介绍微服务在大型复杂系统的架构实践。

电商是促销拉动式的场景，也是价格战驱动的场景。 618 和双 11，都是典型的促销活动。其实都是在抢用户、提高市场占有率。在这样的场景之下，对秒杀、抢购是很热衷的玩法。

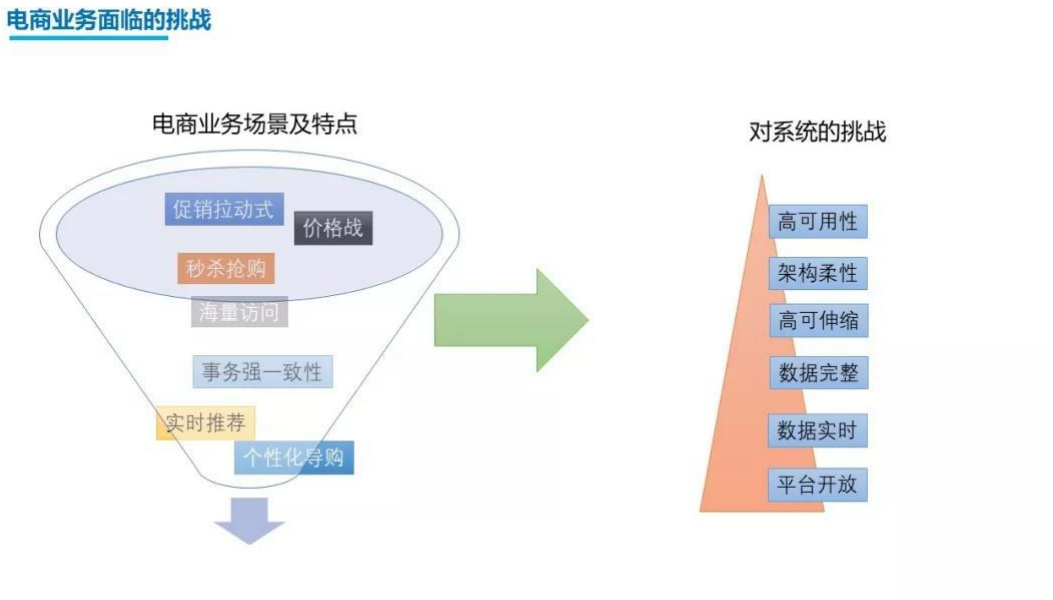

图 4-18 电商业务面临的挑战

促销式的拉动对系统的挑战是什么呢？

可以从上图里看到：对高可用性的要求是非常高的，需要 99.99% 的高可用性。快速迭代，对系统容性的要求很高，从几万单变成几十万单、百万单，架构上不能影响快速迭代，所以有空中加油、或者是高速公路换轮胎的说法。

另外，为了应对瞬间的海量访问（尤其是秒杀场景），系统需要高可伸缩（快速扩容和缩容），这些都是对系统的要求。

大型电商系统的架构

从下往上，数据层，埋点数据把用户行为数据，实时数据存储在 NoSQL 、关系型数据库、大数据平台 。

图 4-19 大型电商系统技术架构

基础架构层

这层实际上是中间件和服务，包括 MQ 的消息、JOB 的调试中心、SSO 联合登陆，还有发消息的，分布式的文件存储，用户上传的一些图片等等，除此之外还有应用监控的整个体系、自动发布的框架，支持到 AB 测试。

基础服务层

再上面一层就是基础服务层，这实际上是用基础架构层提供的组件和服务，加上一些业务逻辑，构建了一些公用的服务，包括 OMS 、PMS 采购，运费模板、配送区域等，这些都是电商最常用的基础服务。

业务服务层

业务服务层我们可以看到的是，比如用户在前台能看到的界面，比如购物车、订单、首页，不管是不是微服务，至少是服务化的。这层就是所有网站应用的核心。除此之外就是第三方平台的 API 对接。

虚拟类目相当于“标签”，比如我们正常的类目叫做“生鲜”、“服装”，还有一些虚拟的类目叫做“ 618 特卖”，里面会聚合很多的商品，可以理解为一个标签，作为展示用。

暴露在最顶层的我们可以看到，这些就是各个端，比如 H5、PC、官网，这就是最终可见的端。

微服务架构的设计

应用的无状态化

很多网站一开始可能不是微服务化的，在早期的一些项目里，我们为了快速上线交付，会做一些单体的应用。随着订单量的发展，我们就开始做所谓的“微服务化”，第一步是把所谓的单体应用，变成应用的无状态化，以登录 SSO 来看，就是一种解决去状态化的方法。我们会拿到一个 Token，每次访问都会带着 Token，这就是所谓的去状态化。之后每一个应用都有横向可扩的能力。当访问量大的时候，就可以通过加服务器来增强水平扩展的能力。

这种应用无状态，其实配置文件还是有状态的。比如访问的数据库和节点，这些是通过配置文件来完成。我说到的案例基本都是基于 Spring Boot 来做微服务化，相关技术框架包括：Dubbo、ZK、Hystrix、RocketMQ、Elasticsearch、Redis 等等。

单体应用的拆分

在做了应用的无状态之后，就是对单体应用的拆分。拆分有几个维度，一个是从系统的维度，最简单的拆法，就是前后台拆开。比如购物车、商品、搜索、首页等属于前端，而后端给网站运营人员用。

还可以按功能的维度来拆分，对于用户服务，从 Service 层到表结构，其实是可以独立部署的，这就是微服务的概念。技术架构反应的，就是组织架构，在这种架构下，开发团队分为用户服务开发组、价格开发组、商品开发组等。

还可以根据读写维度进行拆分。比如搜索和商城的索引，肯定是独立的两个服务。用户注册下单支付，是一个完整的业务流程。这些是由若干个微服务构成。

服务架构搭建

图 4-20 微服务架构设计

数据的异构

在大型电商系统里面的服务架构搭建的经验和技巧，首先是数据的异构，以订单表为例，一般订单都非常庞大，一般按照 ID 来分表分库。这种分法对于查询用户所有订单时就要去各表捞数据，因此可以按用户维度来异构一张表。对于数据的存储，会分为热数据、冷数据和温数据，分别存在不同的地方。同时也会对数据进行聚合。在一些订单详情页，由于有很多 AJAX 请求，由于请求数太多，也需要做一些请求合并。后台的服务也要做一个合并。

以商品详情页为例，使几个接口的数据缓存合并在 Redis 中，从 Redis 中取得聚合好的数据，称为数据闭环。这是优化网络请求的通常做法。

缓存

缓存在大型电商系统中是常用的优化技巧。浏览器级别的缓存通过响应头进行设置。还会用到 App 客户端的缓存，把 H5/CSS/JS/ 图片打包，提前拉到客户端，在客户端做一个代理服务器，但是不会读取数据。可以提升用户体验。缓存的使用在网络上还有常用的 CDN。进到接入层后，如果使用软负载，也可以使用内存级别的缓存。

消息队列的应用

消息队列的应用，是做服务解耦的好方法。也要考虑消息失败和重试的场景，需要来做一些额外补偿来防止数据丢失。还有一个机制，是数据的校验和补偿。很多的场景能做到的，是最终一致性。大型的电商系统和金融系统场景，非常不一样，在设计分布式系统时，这是常用的方式。在电商中大多数情况，只要实现最终一致性就可以了。

高可用的架构设计

高可用的架构设计，对于电商来说，其实高可用是最基本的要求。如果在促销时，引来千万级别的用户，宕机会损失很大。

服务的降级、分组和故障的隔离

基于微服务架构的电商系统，高可用的方案有以下几个部分，首先要支持服务的降级。要做降级的开关，通常写在配置中心里面。比如在大促时，先把订单放在缓存时，再进行落库等操作。同时还要有服务分组和故障的隔离。比如秒杀时，对秒杀的应用单独部署服务，当秒杀的应用挂了之后，不会影响其他服务，因为有服务的隔离。同时要有限流机制，很多的框架都有支持。

流量治理

在极限的场景下， 对流量的治理要从多层面进行。比如在促销当天，会开启对于爬虫和机器人的流量进行限流。一般会在大促前进行封板，如果出现问题，就进行回滚，比如数据版本的回滚，在设置数据结构的时候，要做支持带数据版本号的回滚。

业务设计

业务设计方面的思考。从图中可以看到订单支付的流程。在设计的时候要考虑防重设计，可以采用防重 key 或者防重表的方案，但是耗费和代价很高，会在某些场景使用，比如积分，扣费等和金钱相关的场景下用。

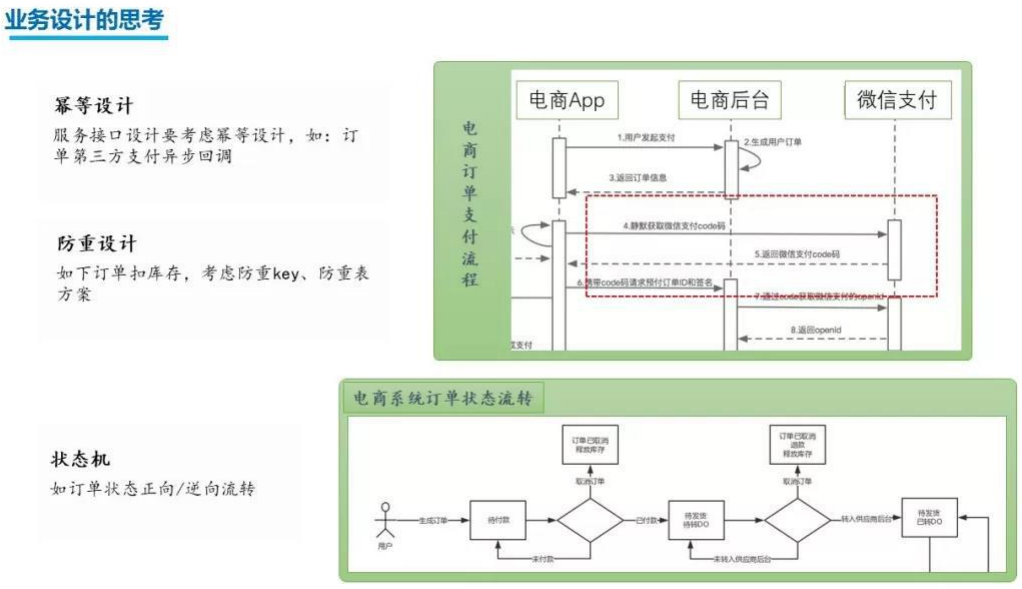

图 4-21 电商业务设计思考

业务设计要考虑状态机。尤其是订单的流转状态里，要做状态机的应用，包括正向和逆向流程，及其产生的结果。

大型移动电商的架构

动态路由

最后来回顾一下大型移动电商的架构。下图是一个移动电商的完整架构。从 App 端，主要做的是，静态文件的缓存、和智能的动态路由。中国的网络环境很复杂，需要在 App 端做智能动态路由。可以上一些 CDN，对动态的内容也做链路优化。会有一些对网络环境检测的机制，可以是 CDN，或者是走域名，也可以暴露 IP。

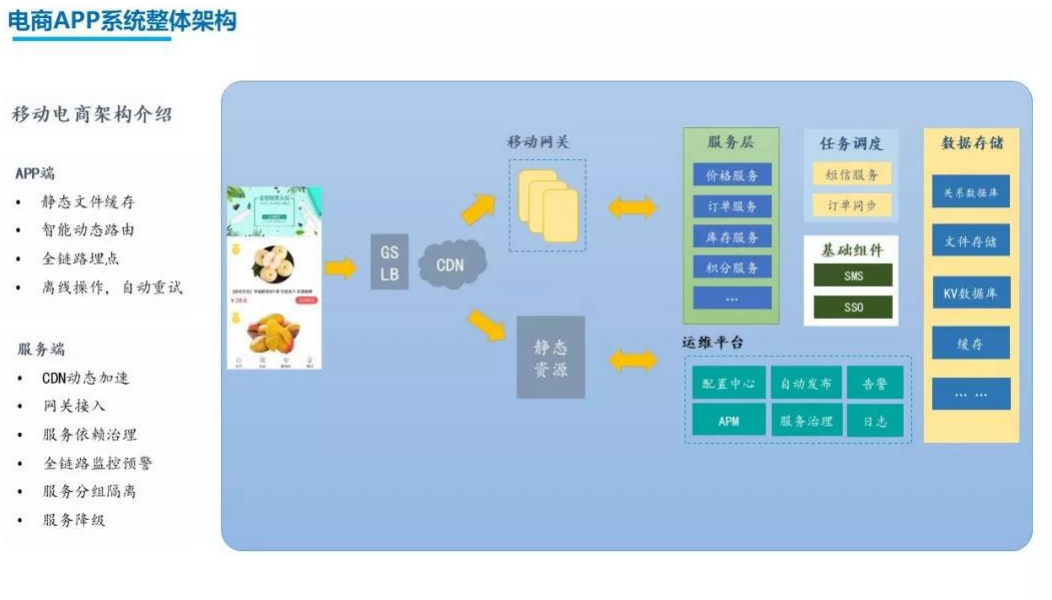

图 4-21 电商 APP 系统整体架构

埋点和网关

移动电商里对 App 来说，还有一个很重要的是埋点，指的是全链路埋点。从 App 里用户的每一个操作，这个操作经过网络、服务层、中间件，整个链路要可以监控。对于快速的定位问题是非常有帮助的，尤其是移动电商性能的优化，第一步就是埋点。

在网络这一层，还有网关的接入。比如限流，动态负载。在网关里没有加太多逻辑，也有不同的做法。对于服务来说，最复杂的，是服务的依赖和治理。服务之间调用的优化，要基于业务场景，比如说购物车的服务，调用到价格、库存、促销等。当依赖的服务，不可用的时候，比如价格不可用，设计依赖的时候，要在购物车服务中做一个缓存，来对缓存调用，最后再对最终一致性进行验证。

全链路监控的做法，需要做到预警，这就是一个基础。通过对数据的监控请求来后，根据场景来做预警方案。

以上就是，微服务在大型电商系统中的应用，许多知识点和架构用法没有展开讲，感兴趣的读者，可以去深入研究这些用法背后的思考逻辑。

6 Serverless 架构初探

什么是 Serverless

技术圈中的人们一般称呼 Serverless 为“无服务器架构”。Serverless 不是具体的一个编程框架、类库或者工具。简单来说，Serverless 是一种软件系统架构思想和方法，它的核心思想是：用户无须关注支撑应用服务运行的底层机制。这种架构的思想和方法将对未来软件应用的设计、开发和运营产生深远的影响。

所谓“无服务器”，并不是说基于 Serverless 架构的软件应用不需要服务器就可以运行，其指的是用户无须关心软件应用运行涉及的底层服务器的状态、资源（比如 CPU、内存、磁盘及网络）及数量。软件应用正常运行所需要的计算资源由底层的云计算平台动态提供。

Serverless 与传统架构区别

在传统的场景里，当用户完成了应用开发后，软件应用将被部署到指定的运行环境，用户会申请一定数量、一定规格（包含一定数量的 CPU、内存及存储空间）的服务器以满足该应用的正常运行。当应用上线后，根据实际的运营情况，用户可能会申请更多的服务器资源进行扩容，以应对更高的访问量。

在 Serverless 架构下，情况则截然不同。当用户完成应用开发后，软件应用将被部署到指定的运行环境，这个运行环境不再是具体的一台或多台服务器，而是支持 Serverless 的云计算平台。有客户端请求到达或特定事件发生时，云计算平台负责将应用部署到某台 Serverless 云计算平台的主机中。Serverless 云计算平台保证该主机提供应用正常运行所需的计算资源。在访问量升高时，云计算平台动态地增加应用的部署实例。当应用空闲一段时间后，云计算平台自动将应用从主机中卸载，并回收资源。

Serverless 带来的价值

一项技术被广泛认可和采纳的原因，往往不是因为这项技术有多新鲜或多酷，最重要的推动力是其能为业务带来巨大的商业或经济价值。Serverless 的价值体现在以下几个方面：

第一，降低运营复杂度

Serverless 架构使软件应用和服务器实现了解耦，服务器不再是用户开发和运营应用的焦点。在应用上线前，用户无须再提前规划服务器的数量和规格。在运维过程中，用户无须再持续监控和维护具体服务器的状态，只需要关心应用的整体状态。

第二，降低运营成本

Serverless 的应用是按需执行的。应用只在有请求需要处理或者事件触发时才会被加载运行，在空闲状态下 Serverless 架构的应用本身并不占用计算资源。

在 Serverless 架构下，用户只需要为处理请求的计算资源付费，而无须为应用空闲时段的资源占用付费。这个特点将为大规模使用公有云服务的用户节省一笔可观的开销。

第三，缩短产品的上市时间

在 Serverless 架构下，应用的功能被解构成若干个细颗粒度的无状态函数，功能与功能之间的边界变得更加清晰，功能模块之间的耦合度大大减小。这使得软件应用的开发效率更高，应用开发的迭代周期更短。

Serverless 架构应用的上市时间（Time To Market，TTM）将大大减少。

第四，增强创新能力

应用的开发和部署效率的提升，使得用户可以用更短的时间、更少的投入尝试新的想法和创意。

Serverless 的技术特点

按需加载。在 Serverless 架构下，应用的加载（load）和卸载（unload）由 Serverless 云计算平台控制。这意味着应用并不总是一直在线的。只有当有请求到达或者有事件发生时才会被部署和启动。

事件驱动。通过将不同事件来源（Event Source）的事件（Event）与特定的函数进行关联，实现对不同事件采取不同的反应动作，这样可以非常容易地实现事件驱动（Event Driven）架构。

状态非本地持久化。应用不再与特定的服务器关联。因此应用的状态不能，也不会保存在其运行的服务器之上，

非会话保持。应用不再与特定的服务器关联。每次处理请求的应用实例可能是相同服务器上的应用实例，也可能是新生成的服务器上的应用实例。因此，Serverless 架构更适合无状态的应用。

自动弹性伸缩。Serverless 应用原生可以支持高可用，可以应对突发的高访问量。应用实例数量根据实际的访问量，由云计算平台进行弹性的自动扩展或收缩。

应用函数化。Serverless 架构下的应用会被函数化，但不能说 Serverless 就是 Function as a Service（FaaS）。

依赖服务化。在 Serverless 架构下的应用依赖，应该服务化和无服务器化，也就是实现 Backend as a Service（BaaS）。所有应用依赖的服务都是一个个后台服务（Backend Service），应用通过 BaaS 方便获取，而无须关心底层细节。和 FaaS 一样，BaaS 是 Serverless 架构实现的另一个重要组件。

Serverless 的应用场景

Web 应用。Serverless 架构可以很好地支持各类静态和动态 Web 应用。

通过 FaaS 的自动弹性扩展功能，Serverless Web 应用，可以很快速地构建出能承载高访问量的站点。

举一个有意思的例子，为了应对每 5 年一次的人口普查，某国政府耗资近 1000 万美元构建了一套在线普查系统。但是由于访问量过大，这个系统不堪重负而崩溃了。在一次编程比赛中，两个大学生用了两天的时间和不到 500 美元的成本在公有云上构建了一套类似的系统。在压力测试中，这两个大学生的系统顶住了类似的压力。

移动互联网。Serverless 应用通过 BaaS 对接后端不同的服务而满足业务需求，提高应用开发的效率。开发者可以通过函数快速地实现业务逻辑，而无须耗费时间和精力开发整个服务端应用。

物联网。物联网（Internet of Things，IoT）应用需要对接各种不同的数量庞大的设备。不同的设备需要持续采集并传送数据至服务端。Serverless 架构可以帮助物联网应用对接不同的数据输入源。

系统集成。Serverless 应用的函数式架构非常适合用于实现系统集成。用户无须像过去一样为了某些简单的集成逻辑而开发和运维一个完整的应用，用户可以更专注于所需的集成逻辑，只编写和集成相关的代码逻辑，而不是一个完整的应用。

Serverless 的局限

控制力。对于一些希望掌控底层计算资源的应用场景，Serverless 架构并不是最合适的选择。

可移植性。Serverless 应用的实现，依赖于 Serverless 平台及该平台上的 FaaS 和 BaaS 服务。不同 IT 厂商的 Serverless 平台和解决方案的具体实现并不相同。

安全性。Serverless 架构下，用户不能直接控制应用实际所运行的主机。不同用户的应用，或者同一用户的不同应用在运行时可能共用底层的主机资源。对于一些安全性要求较高的应用，这将带来潜在的安全风险

性能。应用的首次加载及重新加载的过程将产生一定的延时。对于一些对延时敏感的应用，需要通过预先加载或延长空闲超时时间等手段进行处理。

执行时长。大部分 Serverless 平台对 FaaS 函数的执行时长存在限制。因此 Serverless 应用更适合一些执行时长较短的作业。

总的来说，这种新兴的云计算服务交付模式，为开发人员和管理人员带来了许多好处。它提供了合适的灵活性和控制性级别，因而在 IaaS 和 PaaS 之间找到了一条中间的路，由于服务器端几乎没有什么要管理的，Serverless 架构正在彻底改变软件开发和部署领域，比如推动了 NoOps 模式的发展。

7 大型电商系统架构设计

我们把架构设计分为两部分：应用架构设计和基础架构设计。

应用架构设计，指的是跟业务结合最紧密的业务系统架构设计，这里主要介绍电商网站架构、供应链系统架构、个性化推荐引擎架构、电商搜索引擎架构。

基础架构设计，指的是底层系统支撑中间件的架构设计，应用系统是架设在基础架构之上的，这里主要介绍大数据平台架构设计、云平台架构设计、服务治理平台架构设计、分布式文件存储架构设计。

我们马上开启架构设计之旅，进入奇妙的技术世界，我们先来看看大型电商网站架构设计怎么做？

大型电商网站，指的是每日用户访问量达到数百万，每日页面访问量达到数千万乃至上亿级别的网站，达到这个规模的电商网站在国内不会超过 10 家，系统架构设计的目标和原则是：高可用、易伸缩、低成本。

基于这样的架构设计目标和原则，服务化、分布式是这个架构设计的主要思路。

服务化，是将所有核心业务沉淀下来形成各种服务，供各业务系统共享，基础架构相关

资源也以服务的方式提供，包括消息、文件存储、缓存等。

分布式，系统架构中的每一层及所有的资源都是分布式的，支持平滑的水平扩展。

从技术架构上可以把电商网站系统分为五层：应用层、核心服务层、基础服务层、数据访问层、数据源，如图 9-1 所示。下面我们介绍每一层的作用和包含的主要模块。

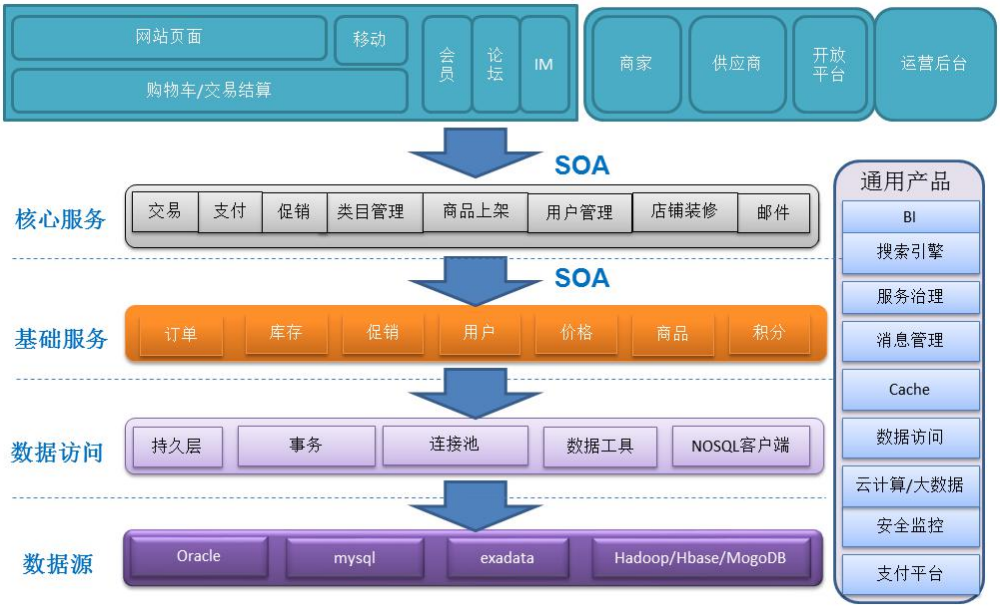

图 9-1　电商系统架构

应用层，是面向用户的应用系统，提供给顾客、商家、员工等角色使用的平台，如网站页面、购物车、结算中心、会员中心、在线客服、商家平台、供应商平台、运营后台等。应用系统通过调用核心服务，实现特定的业务逻辑。

核心服务层，把核心业务逻辑进行封装，以服务的形式提供出来，供各应用系统进行调用。核心服务有交易、支付、促销、类目维护、商品管理、店铺装修、库存操作等。

基础服务层，把原子业务进行封装，以服务的形式提供出来，供核心服务层调用，这里要注意，一般情况下应用层不能够直接调用基础服务层，也就是说不能跨层调用服务。核心服务层在封装某个业务逻辑的时候，常常会调用多个基础服务层的接口。基础服务层包括订单、库存、价格、用户、商品、积分等。

数据访问层，是实现对数据访问的中间件层，功能模块包括持久化组件、事务处理、连接池、NoSQL 客服端、SQL 管理工具箱等。任何数据访问都必须通过数据访问层，不允许绕过数据访问层，直接访问数据库。

数据源，指数据库集群，包括 Oralce、MySQL、Hadoop、Hbase、MangoDB 等，数据库一般是集群部署，实现主备机制、读写分离。

以上介绍了各层要做的事情，大家注意到 SOA 在这个架构中被大量使用，因此需要有服务治理平台，能够对服务进行管理，比如能够支持故障隔离，优雅降级，可以跟踪完整的请求生命周期，可以快速响应和定位问题，可以管理所有服务的依赖关系。

另外，数据的读取还需要有缓存中间件，来减少对数据源的请求次数，缓解数据库的压力，根据场景合理使用多级缓存、本地缓存等，缓存要有主动和被动更新机制，以防止脏数据的产生和被误使用。

同时，还必须有完备的监控预警机制，对硬件、数据库、服务、应用、容器、中间件等进行监控预警，必要时候会发出服务降级指令，来牺牲一些对性能损耗较大的模块，确保主购物流程的正常运行。

最后，整体网站要能够实现多数据中心的部署，来实现性能提升、访问速度优化，以及实现容灾。

8 电商供应链系统架构设计

供应链系统，是在电子商务平台中最重要的系统之一，贯穿了采购管理、仓库管理、配送管理、车辆管理、绩效管理等，供应链系统还需要跟订单系统、客服系统、生产计划系统、BI 等外部系统对接，供应链系统的建设重在作业流程梳理和优化，业务性更强，在本节的介绍中侧重在业务架构层面。

如图 9-2 所示，给大家展示了电商物流业务的闭环，下面我们来梳理其中的业务流程，给大家呈现完整的电商物流业务视图。

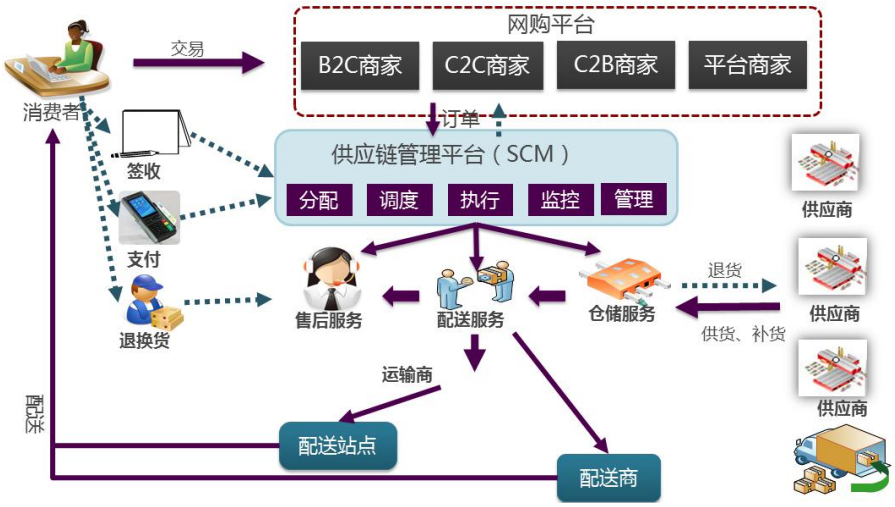

图 9-2 电商物流业务闭环

消费者从网购平台（如淘宝、京东、1 号店等购物网站）中挑选自己需要的商品，完成下单操作，这时候生成了一张订单，订单系统根据消费者的送货地址，通过算法程序，把订单分配到某个或多个仓库中，这里可能涉及到订单拆分，将一个订单拆分成多个子单，并且生成出库单。

供应链管理平台，将接收到的出库单传输给仓库管理系统，仓库管理系统为了提高效率，并不采用来一张出库单，就去拣一次货的方式，而是分波次进行拣货，即累积一定的订单量后，分批次地进行拣货，然后再装箱、打包、出库。

当系统发现某些商品备货不足的时候，就会自动触发一个采购单，发送给供应商，供应商的管理系统收到补货指令，就会通过供应商管理系统进行下单确认、预约送货等操作。

消费者购买的商品包裹出库后，包裹信息就会流转到配送管理系统中，配送管理系统跟踪每一个包裹的状态，包裹会被发往各个配送站点，配送站点接收到包裹后，会在配送管理系统中做确认，按配送范围把包裹分配给每一个配送员，或委托第三方配送公司进行配送。

配送员依据配送系统规划的配送路径，依次前往消费者所在的小区，当顾客签收包裹后，包裹状态信息就会传回配送系统。

订单和包裹的整个流转过程信息，对消费者是透明的，如京东、1 号店等网站，均可以做到及时将订单和包裹的状态信息推送给消费者，提升顾客的消费体验。

根据以上过程，我们可以归纳出，供应链系统的业务架构图如图 9-3 所示，中间部分就是库存管理系统的核心模块。包括供应商预约送货、仓库收货、验收入库、商品上架，订单商品拣货、包裹分拣、包裹包装，包裹出库、缺货补货等。

图 9-3 中，两侧的是辅助模块，功能包括仓库货物调拨、工人绩效管理、仓库优化、仓库服务计费、交叉转运、退换货等逆向物流。图 9-3 下方是与仓库管理系统进行数据交互的系统，这些系统包括订单系统、配送系统、单据状态监控管理系统、物料计划系统、客服系统等。

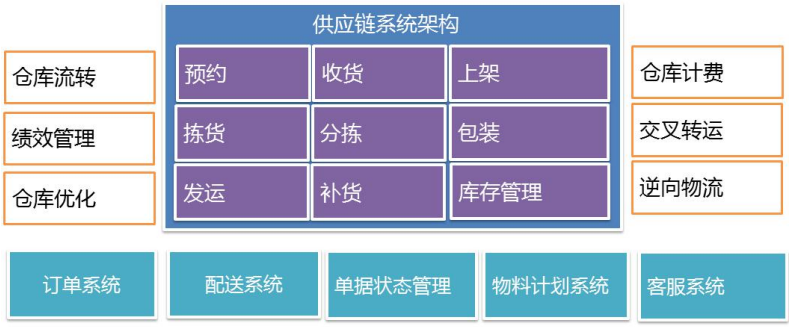

图 9-3　供应链系统架构

仓库管理系统，是整个供应链管理系统中的重要组成部分，如图 9-4 所示。下面我们详细介绍这个系统的主要业务模块。

入库业务，入库有许多种类，如采购入库、调拨入库、退换货入库等，先把商品的 ASN 码导入系统中，通过预约送货模块，进行送货预约。当货物送达仓库后，仓库进行验收入库，对货物的型号、效期、包装等进行检查，通过后才可以被接收。根据系统的建议，进行商品上架。

出库业务，仓库管理系统接收到订单系统传输过来的出库单信息，进行打波次、拣货、分拣、装箱、出库等操作，其中的波次是支持自动波次和人工波次的，拣货支持 RF 拣货、纸单拣货、系统推荐拣货等。

库存管理，是对仓库货物做主要操作的模块，功能包括库存移动、库存盘点、库存锁定等，货物上架以后，不能任意移动货物，否则容易造成系统库存和实物库存对不上，造成损

耗问题。

系统管理、基础管理，是一些基础信息、权限和系统配置的设定模块，通常在系统初始化的时候进行资料的导入，后面再根据实际情况去维护这些信息数据。

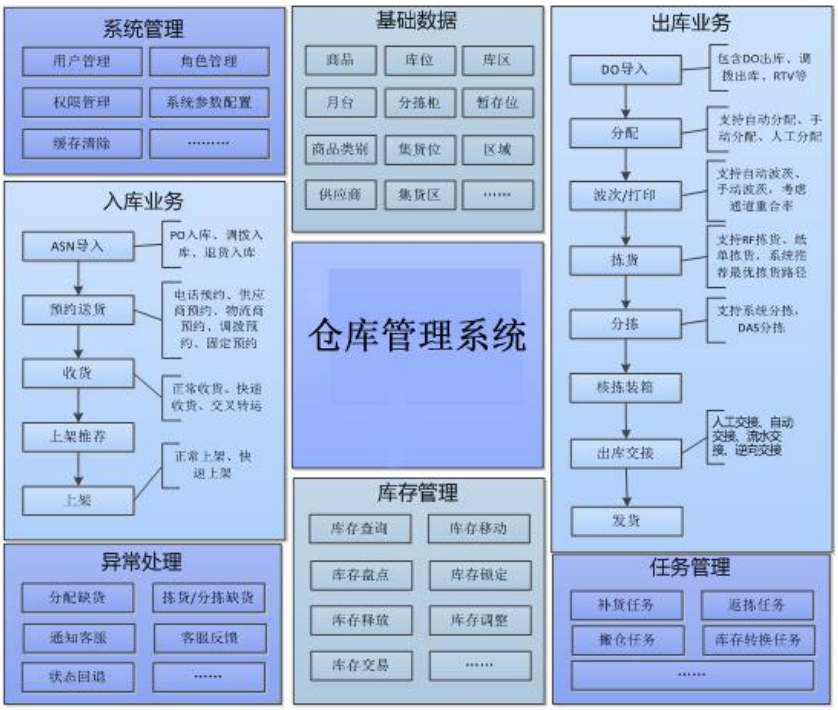

图 9-4　仓库管理系统架构

9　个性化推荐引擎架构设计

个性化推荐引擎，被广泛使用在电商网站、网络广告、资讯网站中，这里我们以电商网站的使用场景为例，给大家介绍个性化推荐引擎的架构设计。

在综合性的电商网站上，商品的数量通常在数十万以上，个性化推荐是帮助顾客在最短的时间内找到想要的商品。还有一种情况是，顾客在购物过程中并没有很明确的购物意图，只是逛逛，看到喜欢的就买，这个时候个性化推荐就能够根据顾客的浏览行为，进行有针对性的商品推荐，帮助顾客挖掘他的购买需求。

个性化推荐，能够帮助电商网站，提升顾客体验、提升销售业绩，通过个性化推荐，可以在顾客即将达到某个购买周期时，给顾客提醒，提升人文关怀。比如，通过个性化推荐引擎的算法，可以在顾客的大米、油盐即将使用完的时候，给顾客提醒信息，让顾客通过“一键购”的便捷方式再次购买商品，有效提升老顾客的复购率。

个性化推荐，能够优化网站展示商品内容，提升销售、提升毛利、提高长尾销售、促进跨品类购买等。经过统计发现，使用了个性化推荐以后，能够提升顾客下单率 2 倍以上，提升订单转化率 20%以上。

了解了个性化推荐引擎的神奇效果之后，我们来学习如何搭建个性化推荐引擎，如图 9-5 所示，这是个性化推荐引擎和应用闭环。新一代的推荐引擎是基于用户画像的，当前被

广泛使用在各大电商网站的推荐引擎中。在详细介绍之前，我们先回顾老一代的推荐引擎的原理，老一代的推荐引擎是基于商品属性关联的，以亚马逊网站为代表，这跟商品的品类是有关系的，最初亚马逊是以卖书为主的，书的主要属性是书目分类、作者、题材等，例如，当顾客购买了狄更斯的作品时，可以推荐作者其他的作品给顾客，一般情况下是有效的。

但是在品类繁多的百货类商品的购物环境中，基于商品属性的推荐就略显单一了，例如顾客购买了飞利浦的剃须刀，再推荐给顾客其他品牌的剃须刀，或推荐飞利浦的其他电子产品，效果不会很好。

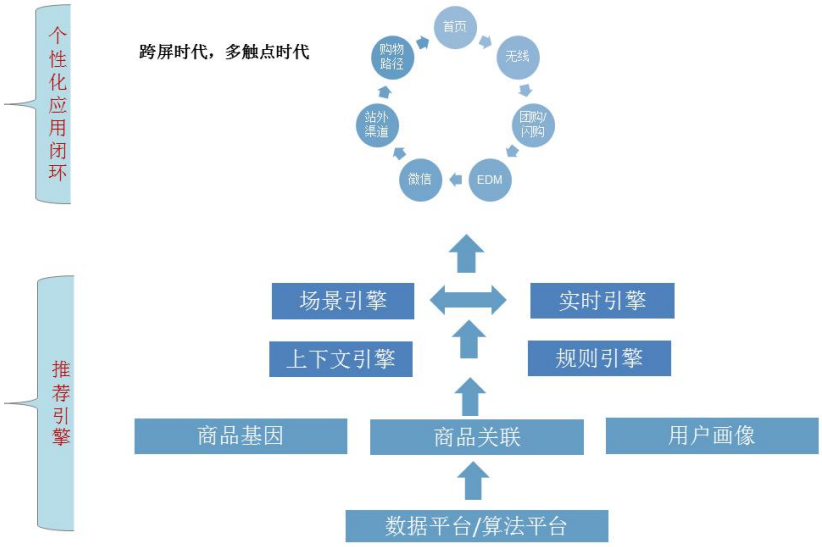

图 9-5　个性化推荐引擎应用闭环

在这类商品的各类繁多的复杂的购物场景中，新一代的基于顾客画像的推荐就取得了非常好的效果，再结合商品关联、商品基因，通常就比较精准了。顾客画像提供了丰富的基础数据，需要通过四个引擎模块对这些顾客画像数据进行处理，这些引擎包括场景引擎、实时引擎、上下引擎、规则引擎。

场景引擎，是根据在购物过程中的不同场景进行推荐，例如顾客刚进入网站、用户浏览商品详情页、购物车页面、订单结算页等场景，在这些场景里，顾客的诉求都是不一样的。比如，顾客浏览商品详情页，此时顾客对这款商品是感兴趣的，可以做同类商品推荐、关联商品推荐等；当顾客把商品加入购物车，此时可以推荐给顾客，“买了这款商品的其他顾客又买了什么”这样的商品推荐列表，引导顾客购买更多的商品。

实时引擎，是根据用户的浏览行为提供实时推荐建议，这对推荐系统的计算能力要求是非常高的，需要有实时计算框架来支持。

上下文引擎，通过顾客的浏览轨迹，结合上下文内容，给顾客推荐与上下文相关的商品。

规则引擎，通过人为配置一些规则，包括节日、季节、热点事件等社会化信息，给顾客推荐更应景的商品，例如在情人节来临之前，推荐鲜花、巧克力、浪漫餐厅等商品。

在上面的描述中，我们了解到了顾客画像的重要性，如图 9-6 所示是顾客画像的组成，包括人口中统计学信息、兴趣图谱、消费类型、忠诚度、第三方网站的顾客画像。通过对这

些信息的分析，个性化推荐引擎就能够做到比用户自己更了解自己。

图 9-6　顾客画像组成

根据顾客画像信息，给顾客“打标签”，每个顾客都有一系列的“标签”，个性化推荐引擎根据场景来选择哪些是主标签、哪些是辅助标签。

以上我们了解了个性化推荐引擎的业务实现，接下来我们来学习个性化推荐引擎的技术架构，如图 9-7 所示。从下往上看，最底层是规则数据层，是对规则数据的存储和加工，它通过数据总线，向上输出规则数据；规则引擎层，包括用户资料、上下文等规则处理模块，向上输出适合场景的规则，将用户画像、商品关联、类目和商品属性数据输入到规则引擎中，得出了初步的推荐结果；再经过场景引擎的规则过滤、去重、结果优化，把最终推荐结果展示给顾客；最后根据顾客的点击情况，再反馈给推荐引擎，用于优化下一次的推荐结果，这也是个机器学习的过程，实现了程序的自我进化，数据和规则积累得越多，个性化推荐引擎的计算结果就越接近顾客的真正需要。

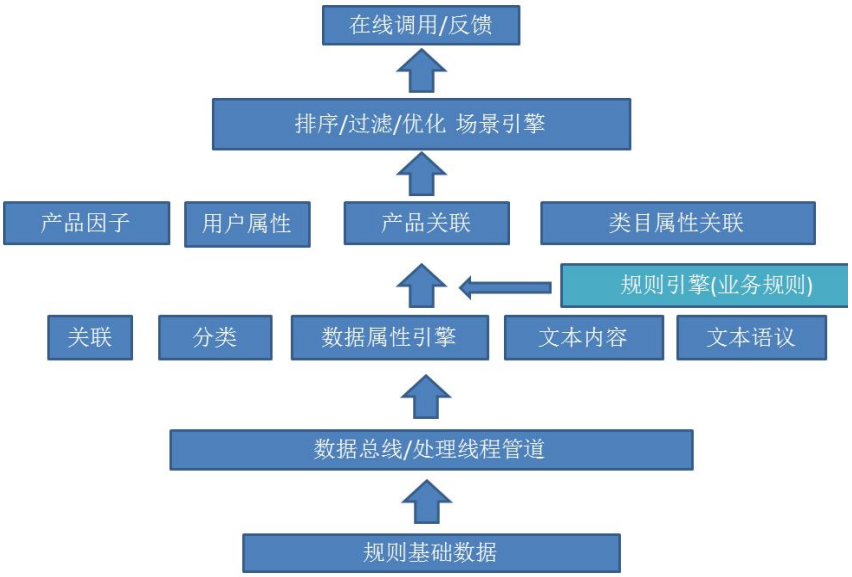

图 9-7　个性化推荐引擎技术架构

10　电商搜索引擎架构设计

电商搜索引擎，是帮助顾客快速找到需要购买的商品的工具，衡量一个电商搜索引擎是否成功的标准是，顾客在一连串的搜索行为当中，是否越来越接近自己的真实需求。顾客越快进入商品页面去浏览商品，越表明搜索引擎推荐的搜索结果越精确。电商搜索引擎，是传统搜索引擎的一个垂直领域，为了更好地学习搜索引擎的相关知识，我们首先要看一个完整的搜索引擎的技术架构。

一个完整的搜索引擎技术框架，如图 9-8 所示，搜索引擎的技术架构，分成 3 个部分：信息采集、建立索引库、提供检索服务，下面我们分别来探讨这 3 部分内容。

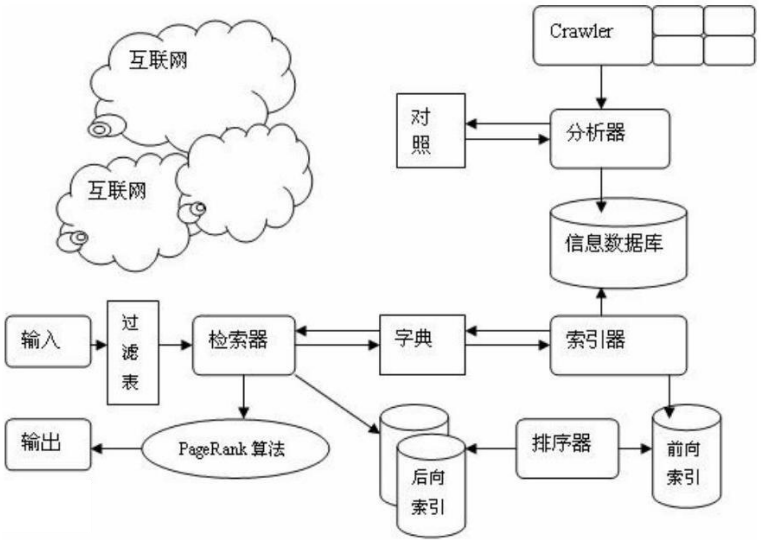

图 9-8　搜索引擎技术架构

信息采集，在互联网中发现、搜集信息和数据。通常，这个步骤是通过爬虫

（Crawler/Spider）抓取网页来实现的。每个独立的搜索引擎都有自己的网页抓取程序爬虫。爬虫 Spider 顺着网页中的超链接，从这个网站爬到另一个网站，通过超链接分析连续访问抓取更多网页。被抓取的网页被称之为网页快照。由于互联网中超链接的应用很普遍，理论上，从一定范围的网页出发，就能搜集到绝大多数的网页。

建立索引库，对收集到的信息进行提取和组织建立索引库。搜索引擎抓到网页后，还要做大量的预处理工作，才能提供检索服务。其中，最重要的就是提取关键词，建立索引库和索引。根据应用场景的不同，其他可能的处理还包括去除重复网页、分词（中文）、判断网页类型、分析超链接、计算网页的重要度/丰富度等。

提供检索服务，由检索器根据用户输入的查询关键字，提供检索服务。接受到关键词后，系统在索引库中快速检出文档，进行文档与查询的相关度评价，对将要输出的结果进行排序，并将查询结果返回给用户。通常，为了用户便于判断，除了网页标题和 URL 外，还会提供一段来自网页的摘要及其他信息。

其实搜索已经是一项非常成熟的技术，这里不打算展开讨论了，只介绍几个在搜索技术架构上比较重要的技术点：分布式索引、分布式搜索。

分布式索引，就是通过很多普通配置的硬件，同时进行索引建立的工作，最后进行索引的合并操作。这样处理的好处在于，具备可扩展性，当数据增加的时候，无须增加单台机器的存储设备，而是通过水平扩展，增加配置普通的机器来解决。建立分布式索引，可采用 Hadoop 这类分布式系统进行构建，Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。HDFS 有高容错性的特点，并且设计用来部署在低廉的硬件上；同时它提供高传输率来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 的上一层是 MapReduce 引擎，用于大规模数据集的并行运算。概念 Map（映射）和 Reduce（规约），和它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。基于这些分布式特性，搜索索引建立可以非常容易地通过它来进行扩展。利用 Hadoop 的平台和 MapReduce 的机制，来实现建立分布式搜索索引，是非常好的实践。

分布式搜索，是将原来的单个索引文件划分成 *n* 个切片（shards）。搜索时，并行的搜索这 *n* 个切片，每个切片返回当前 shard 的 topK 命中结果；然后将 *n* 个切片的局部 topK 进行归并排序，得到全局的 topK 排序结果。分布式搜索的好处在于：更好的可扩展性，在用户访问次数和索引大小两个维度都具有水平扩展能力；更高的稳定性，容许部分失败，调用成功率显著提高；更灵活的全量更新策略，可针对不同类型的数据；更灵活的排序算法，可以针对不同类目，做定制化的排序；更好的可维护性和通用性，支持不同类型的搜索。

11 大数据平台架构设计

近年来，大家对大数据的关注度和使用频率越来越高，软件产品中的各类数据都被记录下来，以便更好地研究和分析。在电商企业中，每天系统记录下来的运营数据，达到几百 GB 增量的规模，为了保证所有数据能集中存储并且可随时访问，越来越多的企业把离线数据体系从商用的 Exadata 等解决方案，全面转向开放的 Hadoop 体系当中，以谋求成本与扩展性的平衡。

有一定技术实力的互联网公司，纷纷搭建自己的大数据平台，如图 9-9 所示是一个典型的大数据平台的技术架构，下面我们一起来学习。从图 9-9 可以看到，大数据平台是由数据存储、数据同步分发、监控、离线计算、平台安全、资源申请等部分构成的。

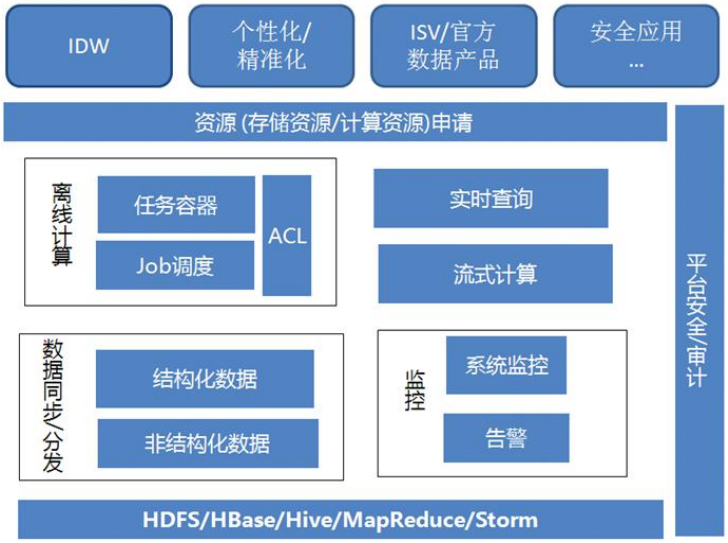

图 9-9　大数据平台技术架构

数据存储，是整个大数据平台的基础，包含如：HDFS、HBase、Hive、MapReduce、Storm 等等。下面，我们对其中的主要框架做些介绍，详细资料大家可以到搜索引擎中获取。

HDFS，分布式文件系统，Hadoop 的核心组成部分。

MapReduce，分布式数据处理，Hadoop 核心之一。

HBase，一个分布式的，列存储数据库，使用 HDFS 作为底层存储，同时支持 MapReduce 的批量式计算和点查询。

Zookeeper，一个分布式的，高可用的协调服务。提供分布式锁之类的基本服务，用于构建分布式应用。

Hive，分布式数据仓库，Hive 管理 HDFS 中存储的数据，并提供基于 SQL 的查询语言用以查询数据。

Hama，建立在 Hadoop 上的分布式并行计算框架，基于 Map/Reduce 和 Bulk Synchronous 的实现框架，运行环境需要关联 Zookeeper、HBase、HDFS 组件。

Mahout，一个基于 MapReduce 的机器学习算法库，运行在 Hadoop 集群上。

Cassandra，一种混合的非关系型数据库，类似于 Google 的 BigTable。

以上就是数据存储层中，用到的一些开源数据框架，我们继续看大数据平台的其他组成部分。

数据同步分发，这个组件对数据同步和分发做统一管理，可实现异步、分布式的数据同步和分发。

监控，指的是对大数据平台的服务和资源，进行监控和预警，包括数据存储的可用性、性能、系统负载、资源请求的响应时效等。

离线计算，处理离线计算任务的模块，包括任务容器、任务调度定时器、异常捕获等模块，确保离线计算任务能够在资源容许的情况下，按计划运行。

平台安全，主要包括对数据访问权限的管理，把数据划分成不同的安全等级进行管理，当访问某些安全级别高的数据时，会触发一个审批流程，经过主管审批后才能访问。

资源申请，指的是对大数据平台的计算或存储资源发起一个使用请求，这里会记录每一个数据操作访问，以供日后审计。

12 云平台架构设计

云平台是个非常宽泛的领域，本节侧重介绍企业私有云平台的架构，大家知道云平台可以分成：IaaS 基础设施即服务、PaaS 平台即服务、SaaS 软件即服务。如图 9-10 所示，这是一个完整的企业级应用平台，从底层的存储资源、计算资源、网络资源，到中间层的容器服务、缓存服务、健康检查服务等，再到最上层的业务应用、接口应用等，这是用云的架构思想，构建的企业级应用。

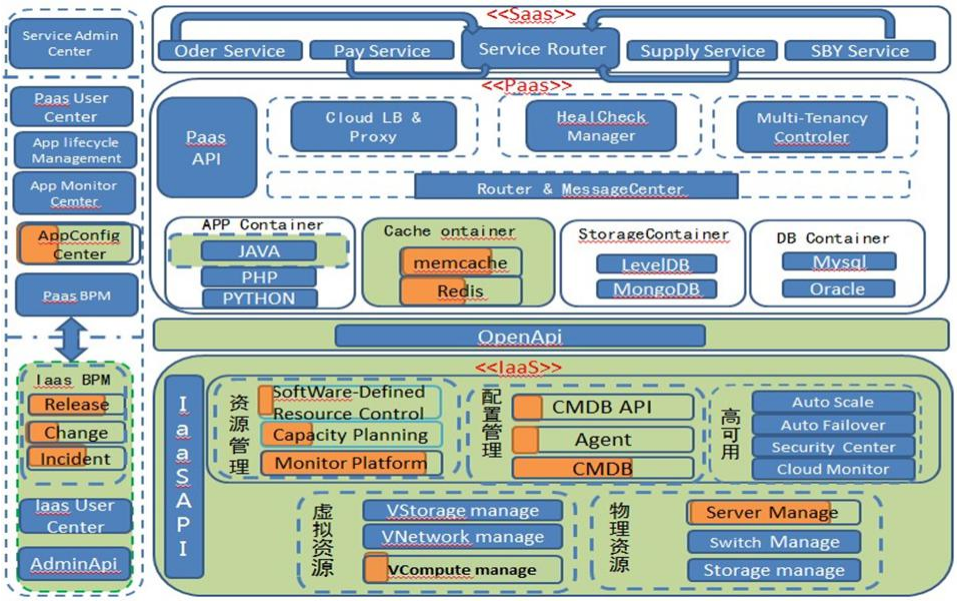

图 9-10 云平台技术架构

下面我们重点介绍 IaaS 部分，通过打造 Iaas 来构建企业级的私有云平台。如图 9-11 所示是 Iaas 的架构图，我们把私有云平台分成：配置管理、ITIL、虚拟资源管理、物理资源管理、自动调度和监控。

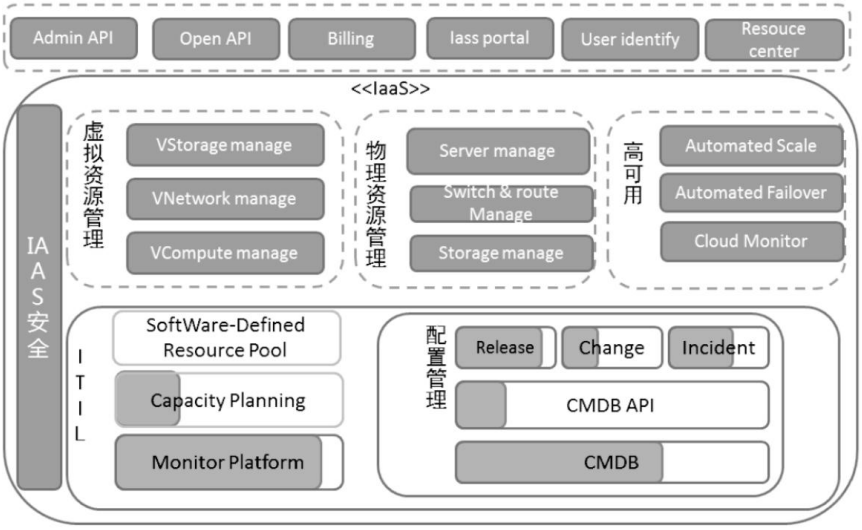

图 9-11　IaaS 架构设计

配置管理，所有 IT 资产进行登记管理，包括服务器硬件、网络设备、虚拟机、网络配置、应用部署管理、变更管理、应用发布信息管理、操作员、权限等，是云平台的基础信息配置管理中心。

ITIL，是软件定义资源池、容量计划、监控门户等，是对资源进行统一治理的模块，制定针对当前资源使用情况的容量计划，监控人员的日常工作平台。

虚拟资源管理，是对虚拟主机、虚拟网络、虚拟存储的管理系统，能够自动化完成虚拟化工作，包括自动化装机、自动化网络配置，对虚拟资源的开通、回收提供统一管理入口，根据应用负载情况，能够触发自动增加虚拟主机，并且部署应用。

物理资源管理，对服务器、存储设备、交换机设备的统一管理系统，基于物理硬件的自动维护、上架和下架、重启等。

自动调度和监控，功能包括自动添加和踢出应用节点，根据负载自动调节资源数量，提供基于云端的监控服务。

以上就是企业私有云平台的架构组成，可提供一个私有云的 Portal，供企业用户一站式地对 IT 资源进行管理，包括成本结算、权限控制、资源分配、部署应用程序等。

13　服务治理平台架构设计

大型电商网站是基于 SOA 架构的，如此大规模的服务架构，需要一个高效、快速、优雅的服务治理平台，本节就给大家介绍，如何搭建一个高效的服务治理平台。

服务治理平台，建立的初衷是：实现对服务健康状况的管理、跟踪每个服务请求的全生命周期，可实现故障隔离、优雅降级，快速响应和定位问题，可管理服务之间的依赖关系。我们将采用分布式架构、无中心、无单点的设计原则来设计这个服务治理平台。

如图 9-12 所示，这就是服务治理平台的架构设计，采用了 ZooKeeper、Detector、消息中间件、MySQL、MongoDB 等开源技术进行搭建。

下面我们来看它们是如何工作的。

步骤 1：服务提供方（Service Provider），首先要向 ZooKeeper Cluster 提交注册申请，注册成功后才可以对外提供服务。

步骤 2：ZooKeeper Cluster 把可用的服务提供方列表，推送给服务使用方（Service Consumer），服务使用方只能使用列表里认证的服务提供方。

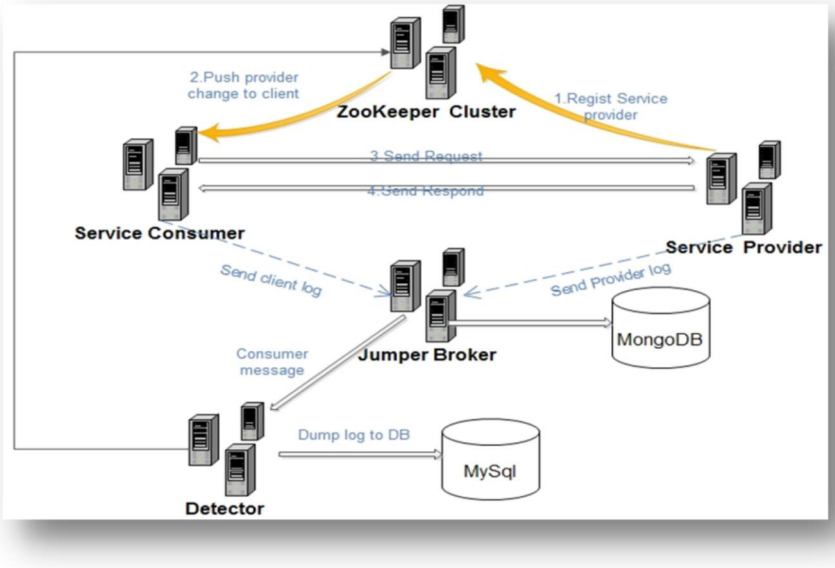

图 9-12　服务治理平台架构

步骤 3：服务使用方，向服务提供方请求服务。

步骤 4：服务提供方，成功回应服务请求方的请求。

同时，服务提供方、服务使用方，都会推送一条调用日志给 Jumper Broker，信息的主要内容是调用频次、响应时间等，Jumper Broker 把这些信息经过分析和处理后，把结果发送给 Detector。

Detector 记录这些信息，并且把这些信息推送给 ZooKeeper Cluster。

如果某个服务的响应时间越来越慢，ZooKeeper Cluster 就会发现，并且及时做出调整，比如，不再给这个服务分配那么多的调用量，直到它的状态恢复正常为止。

从图 9-12 中，注意到 ZooKeeper Cluster、Jumper Broker、Detector 都是集群部署，确保了服务治理平台本身的高可用性，在技术实现上也采用异步消息机制、RPC 框架，使得架构本身无中心、无单点，可支持上万个节点。

部署起来也非常简单，只要把服务治理平台的客户端，跟服务一起部署，做些简单配置就可以了。

更多技术干货资料下载，请关注“技术领导力”、“BAT 架构”公众号

技术领导力

BAT 架构

知识星球：老 K 星际不迷航