1.分布式系统介绍

1.1初识分布式系统

初印象就是一个需要用很大计算量的工作,用很多的服务器一起分担,增加效率

1.1.1 定义

一个仅通过消息传递来通信并协调行动,组件分布在网络计算机上的系统.

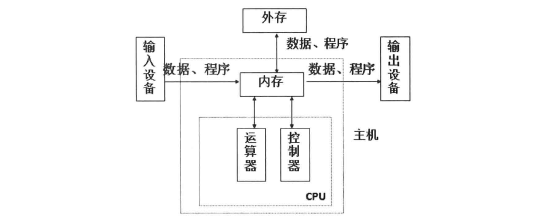
1.1.2 意义

(1) 摩尔定律:当价格不变时，每隔18个月，集成电路上可容纳的晶体管数目会增加一倍，性能也将提升一倍。

这个定律告诉我们，通过更换硬件来提升性能会越来越不划算。同时，处理器也有自己的性能瓶颈。而分布式可以看做是单机系统做容灾备份方案。

1.2 基础

1.2.1 计算机5要素



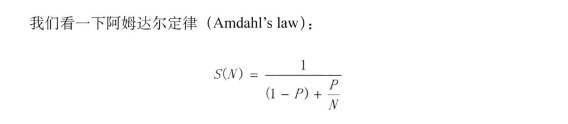
基本要素:输入设备，输出设备，运算器，控制器和存储器（内存和外存）。

1.2.2 线程与进程的执行模式

我们这里的多线程指的是单进程内的多线程

1.2.2.1 阿姆达尔定律

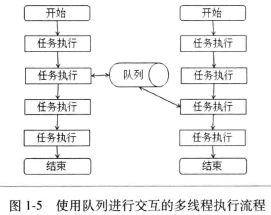
单核时代，程序相对容易随着CPU的更换而变快，而多核，则需要很好的利用多核，更加注重程序的并发和并行。



P：程序中可并行部分的程序在单核上执行时间的占比;

N：处理器的个数;

S(N)：程序在N个处理器相对在单个处理器中的速度提升比

1.2.2.2 互不通信的多线程模式

如果线程之间不需要处理共享的数据，也不需要进行动作协调，那么将会非常简单。

1.2.2.3 基于共享容器协同的多线程模式

很多场景中我们需要在多个线程之间对共享数据进行梳理。多个线程并发访问同一份数据，就需要我们保证访问的正确性。而线程分为安全和不安全，对不安全的，一般通过加锁或者Copy On Write的方式来控制并发。

加锁的方式中，如果数据在多线程中的读写比例很高，一般采用读写锁而非简单的互斥锁。

ConcurrentHashMap是弱一致性的。所以会有这种问题，依赖上一次结果的情况需要强一致性的map，一种是加锁，一种是使用Collections.synchronizedMap()构造一个同步map，其实也是用到synchronized加锁。

# （ConcurrentHashMap 和HashMap）

1.2.2.4 通过事件协同的多线程模式

线程之间存在着协调的需求，A、B两个线程，B线程需要等到某个状态发生后才能继续自己的工作，而这个状态改变和A线程相关。在实际情况下，我们需要避免死锁。

1.2.2.5 多进程模式

使用多进程回避多线程稍微复杂一些，多进程之间可以共享数据，但是其代价比多线程要大，会涉及序列化与反序列化的开销。

1.2.3 网络通信基础知识

网络时代，无论有线还是无线，无论是LAN、MAN还是WAN，都需要解决通信的问题。

1.2.3.1 OSI与TCP/IP网络模型

1.2.3.2 网络IO实现方式

使用Socket套接字进行通信开发三种方式:BIO、NIO和AIO

(1) BIO方式

BIO：Blocking IO,阻塞的方式实现

（2）NIO方式

NIO：Nonblocking IO，事件驱动思想，Reacor模式

（3）AIO方式

AIO：Asynchronous IO，异步IO，Proactor模式。

1.2.4 如何把应用从单机扩展到分布式

从计算机五要素变化维度看单机到分布式

1.2.4.1 输入设备变化

分为两类：一种是互相连接的多个节点，另一种就是传统意义的输入设备

1.2.4.2 输出设备的变化

两种：向其他节点传递信息的节点，另一种传统意义的输出

1.2.4.3 控制器的变化

单机系统，控制器是CPU中的控制器。在分布式中，控制器不是具体电子元件，而是控制方式

分布式系统控制器要做的是协调或控制节点之间的动作和行为。

透明代理——》名称服务——》Master+worker

1.2.4.4 运算器的变化

1.2.2.5 储存器的变化

1.2.5 分布式系统的难点

1.2.5.1 缺乏全局时钟

很多时候我们使用时钟是为了区分两个动作的顺序，而不是一定要知道准确的时间。

1.2.5.2 面对故障独立性

分布式系统中，整个系统一部分有问题而其他部分正常是经常出现的情况，在实现系统的时候，要找到应对办法。

1.2.5.3 处理单点故障

如果某个功能只有某台单机在制程，那么这个节点称为单点，故障称为单点故障。最好把单机变为集群实现，这一般比较困难，另外两种选择：

1，做备份，及时回复，自动回复。

2，降低故障影响范围

1.2.5.4 事务的挑战

分布式中，如何解决事务问题也是一个重要的部分，比如两阶段提交（2pc），最终一致、BASE、CAP、PAXOS

这里先不做展开。

2.大型网站及其架构演进过程

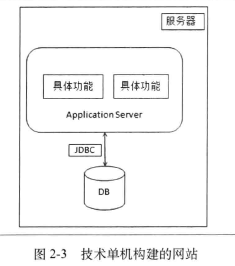
2.1 什么是大型网站

大型网站，海量的数据和高并发的访问量，还要有本身业务和系统的复杂度。

2.2 大型网站的架构演进

大型也是从小网站慢慢发展的，我们从最简单的网站结构，看他们从小到大的变化。

2.2.1 要java技术和单机来构建网站

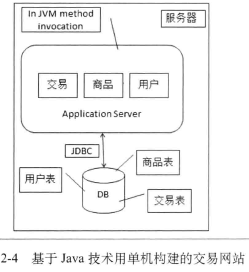


最开始，我们选择一个开源的Server作为容器，直接使用jsp/servlet技术构建我们的应用，选择数据库管理系统存储数据，通过JDBC进行数据库的连接和操作。这样一个最基础的环境就可以工作了。

大型网站，最核心的功能就是计算和存储。

2.2.2 从一个单机的交易网站说起

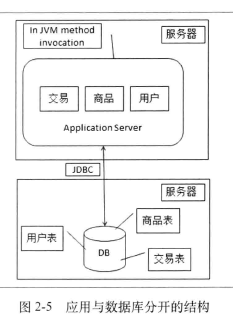
只支持用户、商品和交易，用单机构建，大概是：



各个功能模块之间是通过JVM内部方法调用来进行交互，而应用和数据库之间通过JDBC访问

2.2.3 单机负载告警，数据库与应用分离

对外服务后，访问量增大，负载持续升高，首先可以做的就是把数据库与应用分开，变成下面

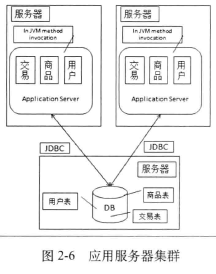
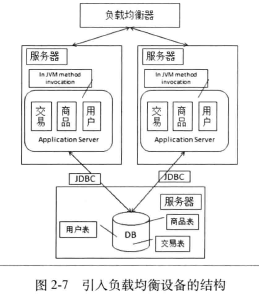


2.2.4 应用服务器负载告警，如何让应用服务器走向集群

应用服务器压力变大，把应用服务器从一台变两台，两个之间没有直接交互。但我们有两个问题需要解决。

（1）用户对两个应用服务器访问的选择问题，可以通过DNS解决，也可以增加负载均衡设备来解决

（2）Session的问题

2.2.4.1 引入负载均衡设备

引入负载均衡，会遇到一个与Session相关的问题

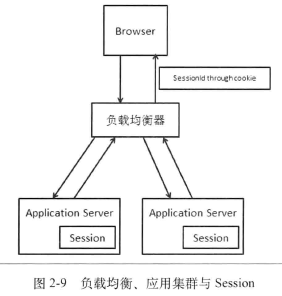
2.2.4.2 解决应用服务器变为集群后的Session问题

什么是session？

使用网站的服务，需要浏览器与Web服务器交互，HTTP协议本身无状态，需要支持会话状态的机制。这个机制使服务器知道哪些请求来自哪个会话的。

具体实现：在会话开始，分配一个唯一会话标识，通过Cookie把标识告诉浏览器，以后每次请求时，浏览器都会带上这个标识。在web服务器上，各个会话有独立的存储，保存不同会话的信息。如果禁用Cookie，一般把这个标识放到URL的参数上。

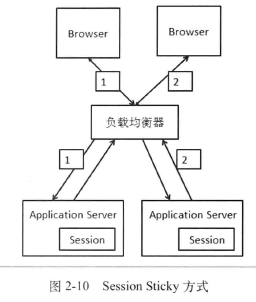
两台服务器，就会遇到一个问题，我们不能保证接下来请求每次都落在同一边的服务器上，这就是Session问题。



解决方案

（1）Session Sticky

保证同一个会话的请求都在同一个Web服务器上处理，需要负载均衡器能够根据每次骑牛的会话标识来进行请求转发。



这个方案带来新的问题：

① 如果有一台web服务器宕机，会话数据会丢失。

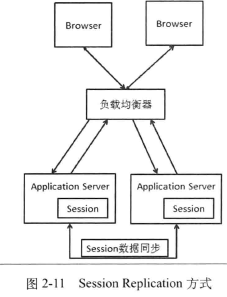
② 会话标识是应用层信息，负载均衡器要将同一个会话都保存到同一个web服务器上需要进行应用层的解析，比第4层的交换开销要大。

③负载均衡器变为了一个有状态的节点，和无状态的节点相比，内存消耗会更大，容灾方面会更麻烦

web服务器时饭店，会话数据是餐具，要保证餐具安全，把自己餐具存在某一家饭店，每次吃都去这一家

（2）Session Replication

我们在每一家饭店都有一套自己餐具，选择饭店就更自由了



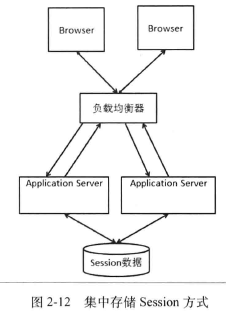
但这个方案也有自己问题

① 同步Session数据造成了网络带宽的开销，机器数越多，开销越大

②每台web服务器都要保存所有Session数据，每台机器用于保存Session数据的内容占用很严重

（3）Session数据集中存储

把Session数据集中存储起来，然后不同WEB服务器从同样的地方来获取Session



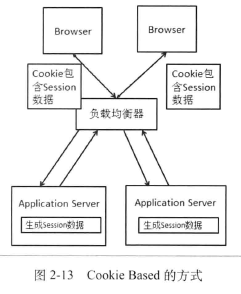
这个模式解决了上述的内存问题，对于网络带宽也比较好，但同时也存在问题

①引入网络操作，相对于本机数据读取，存在时延和不稳定，大多是内网，问题不大

②存储Session的机器有问题，影响我们的应用。

（4）Cookie Based

把Session数据放在Cookie中，相当把自己餐具待在身上



依旧存在不足：

①因为Cookie长度的限制，也限制Session数据的长度

②安全性

③带宽消耗

④性能影响

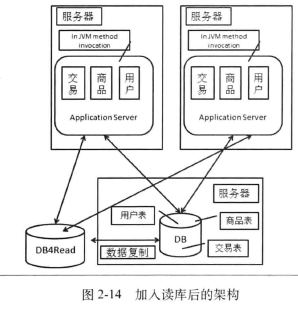
2.2.4.3 小结

大型网站，SessionSticky和Session数据集中存储比较好，具体场景具体分析。

2.2.5 数据读压力变大，读写分离吧

2.2.5.1 采用数据库作为读库

大型网站大多读多写少，我们可以考虑读写分离方式。



带来两个问题

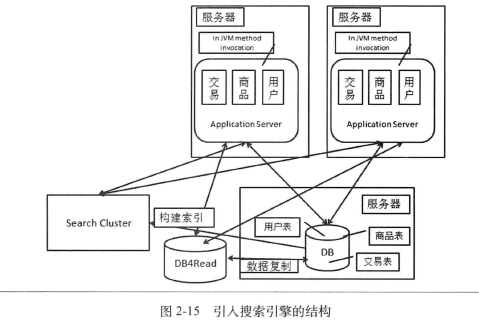
① 数据复制问题

②应用对于数据源的选择问题

数据库系统一般都提供了数据复制的功能，我们可以直接使用它们的自身机制。对于数据复制，还需要考虑数据复制时延问题，以及复制过程中数据的源和目标之间的映射关系及过滤条件的支持问题。

对于应用，增加读库，需要应用分局不同情况来选择不同的数据来源。写操作要走主库，事务种的读也要走主库，还要考虑到备库数据的延迟。

2.2.5.2 搜索引擎其实是一个读库

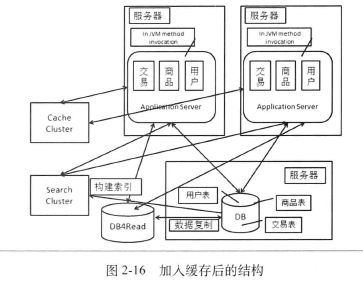


搜索引擎的技术解决了站内搜索时某些场景下读的问题，提供了更好的查询效率，我们可以把搜索引擎当成一个读库。

2.2.5.3 加速数据读取的利器——缓存

介绍缓存的一些用法，看看缓存系统是否可以看做一个读库。

（1）.数据缓存



缓存系统一般用来保存和查询键值对，一般我们在缓存中放的是热数据，而不是全部数据，填充方式通过应用完成的，即应用访问缓存，如果数据不存在，从数据库读出放入缓存，缓存容量不够，最近不被访问的数据被清楚。

还有一种是根据数据库的数据变化，主动把数据放入缓存系统中。好处是，能够及时更新缓存中数据，不会造成读取失效，一般用于全数据缓存。但需要代码能够理解业务逻辑

（2）页面缓存

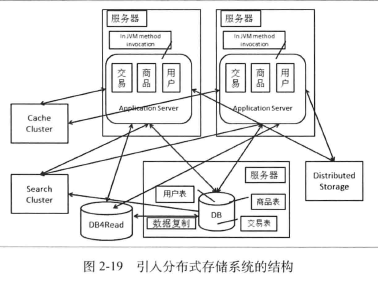
数据缓存加速应用在响应请求时的数据读取速度，但有些页面特别热，需要我们对这些内容进行缓存。

web前端服务器有Apache/Nginx，以Apache举例。具体用ESI类似的思路，也可以把页面缓存与页面渲染放在一起处理。

关键指标是缓存命中率，数据的分布与更新策略也需要结合具体场景考虑。需要有机制避免局部的热点，缓存服务器扩容或缩容尽量平活。数据的更新，会有定时失效，数据变更时失效和数据变更时更新的不同选择。

2.2.6 弥补关系型数据库的部族，引入分布式存储系统

常见的分布式存储系统有分布式文件系统，分布式KEY-VALUE系统和分布式数据库。

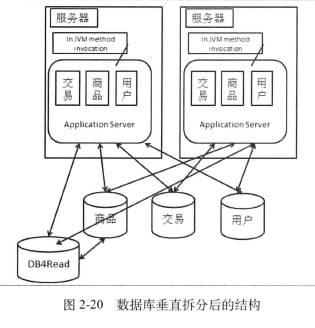


2.2.7 读写分离后，数据库又遇到瓶颈

数据库压力继续增加，我们有数据垂直拆分和水平拆分两种选择。

2.2.7.1 专库专用，数据垂直拆分

垂直拆分就是不同的业务数据拆分到不同的数据库中。

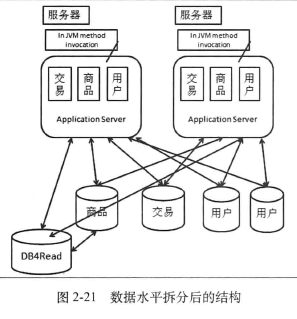


变化？应用需要配置多个数据源，每个数据库连接池需要隔离，需要考虑如何处理原来单机中跨业务的事务。

一种是使用分布式事务，性能明显要低于单机事务；另一种办法就是去掉事务或者不去追求强事务支持。

2.2.7.2 垂直拆分后的单机遇到瓶颈，数据水平拆分。

水平拆分就是把同一个表的数据拆到两个数据库中。读写分离解决的是读压力大的问题，对于数据量大或者更新量的情况并不起作用。



分析影响：

首先，访问用户信息的应用系统需要解决SQL路由的问题，此外，主键的处理也会变得不同，自增字段不能简单的继续使用，也不能直接使用数据库限制来保证主键不重复了。

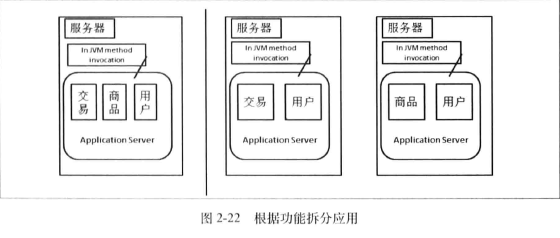
一些查询需要从两个数据库中取数据，如果需要分页，就会比较难处理。

2.2.8 数据库问题解决后，应用面对的新挑战

2.2.8.1 拆分应用

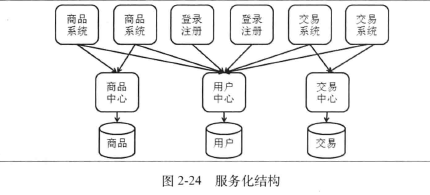
随着业务的发展，应用的功能也会越来越多，我们需要考虑如何不让应用持续变大，这就需要把应用拆开。

（1）根据业务的特性拆分。



这样拆分，不同系统会有一些相似代码，如何保证重复代码的一致以及复用。

2.2.8.2 走服务化的路



①业务功能之间的访问不仅是单机内部的方法调用吗，还引入远程的服务调用

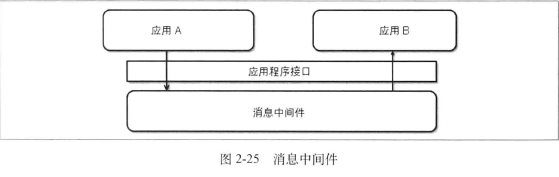
②共享的代码不再是散落，被放在各个服务中心

③把与数据库的交互工作放到服务中心，让前端更加注重于浏览器交互的工作，降低数据库的连接数。

④服务化，可以是有固定小团队来维护，更好保持稳定性

2.2.9 初识消息中间件

消息中间件：在分布式系统中完成消息的发送和接收的基础软件。

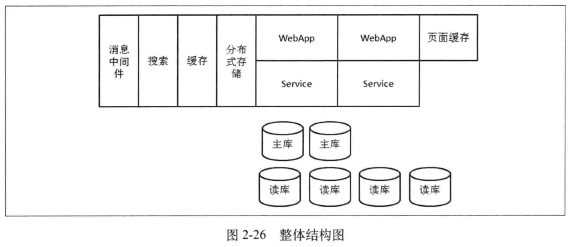


两个好处：异步和解耦。

两个应用不直接联系，这样就完成了解耦，双方不受对方影响，消息的投递都采用了一部的方式。

2.2.10 总结

通过一个例子讲解了网站的架构演进。实际网站没有固定的模式，但基本解决思路大体相同。最后网站的模型。



3. 构建Java中间件

3.1 java中间件定义

中间件是操作系统的一部分，让软件开发者方便的处理通信，输入和输出，能够专注在他们自己应用的部分。

中间件种类很多，我们主要了解下面三类：

①远程过程调用和对象访问中间件

②消息中间件

③数据访问中间件

3.2 构建java中间件的基础知识

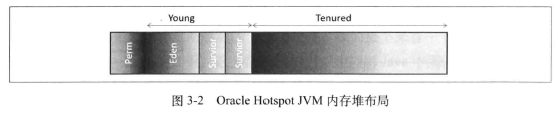
3.2.1 跨平台的java运行环境——JVM

"weite once,run anywhere ",不同平台有不同的虚拟机，但不是不同的虚拟机所识别的是统一格式的中间代码。没有一个万能的调优和问题定位处理方式。

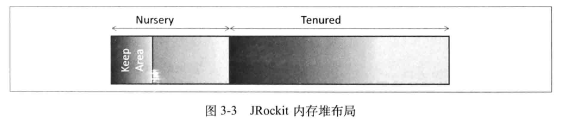
3.2.2 垃圾回收与内存堆布局

java虚拟机规范没有显式地提及垃圾回收的内容，所以我们需要专门的垃圾收集器来完成垃圾回收的工作，不同的虚拟机产品，内存中的堆的布局是不完全一样的，采用的垃圾回收策略也不同。下面是三个常见堆布局。

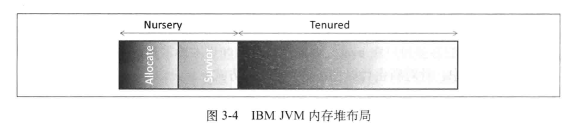
（1）Oracle Hostpot JVM，有Yongoing/Tenured/Perm三块，就是新生代/年老代/持久代。



（2）JRockit



（3）IBM JVM



建议：多花时间去了解针对Hostspot的垃圾回收策略、设置和调优。

3.2.3 java并发编程的类、接口和方法

3.2.3.1 线程池

线程池可以降低创建线程的开销，线程池执行结束后进行的是回收操作，而不是真正销毁线程。

3.2.3.2 synchronized

可以用于声明方法，也可以用于声明代码块

3.2.33 ReentrantLock

类似synchronized，需要显式进行unlock。为何用他：

（1）ReentrantLock提供了tryLock方法。

3.2.3.4 volatile

3.2.3.5 Atomice

这个类主要提供一些相关的原子操作，提升性能。

3.2.3.6 wait、notify和notifyAll

3.2.3.7 CountDownLatch

3.2.3.8 CyclicBarrier

3.2.3.9 Semaphore

3.2.3.10 Exchanger

3.2.3.11 Future和FutureTask

3.2.3.12 并发容器

并发容器是线程安全的容器，不仅追求线程安全，还要考虑并发性，提升容器并发环境下的性能。 加锁互斥，能够完成线程安全，却降低了并发性。并发容器就是尽量不用锁。

CopuOnWrite的思路是在更改容器的时候，把容器复制进行修改，保证正在读的线程不受影响。适用读多写少的场景。而以Concurrent开头的容器，则不完全相同。

想了解并发容器的各种具体实现，可以分析JDK源码。

3.2.4 动态代理

代理有静态和动态之分。

静态是为每个被代理的对象构造对应的代理类。

动态代理是动态地生成具体委托类的代理类实现对象，只需要为一类代理行为写一个具体的实现类就行了。

3.2.5 反射

java反射机制是指在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用他的任意一个方法和属性。

反射用法的示例

① 获取对象属于哪个类 Class clazz = object.getClass();

② 获取类的信息 String className = clazz.getName(); //获取类的名称，通过class对象可以获取更丰富的内容，

③ 构建对象 Class.forName("ClassName").newInstance();

通过newInstance调用来构造对象，要求被构造对象的类一定要有一个无参数的构造函数，否则抛异常。

④动态执行方法

Method method =clazz.getDeclaredMethod("add",int.class,int.class);

method.invoke(this,1,1);

⑤动态操作属性

Field field= clazz.getDeclaredField("name"); field.set(this,"Test");

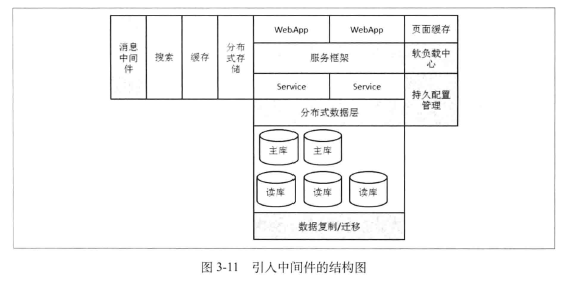
除了java自身提供的动态代理和反射，字节码增强技术和第三方的库也可以直接使用。

Javassist、cglib、asm、bcel可以从网上查询资料。

3.2.6 网络通信实现选择

java中间件要和网络通信打交道，java对于NIO和AIO的支持，具体开发，可以用jdk的api也可以选择诸如MINA/Netty等框架开发。此外，还需要考虑协议的制定。

3.3 分布式系统中的java中间件



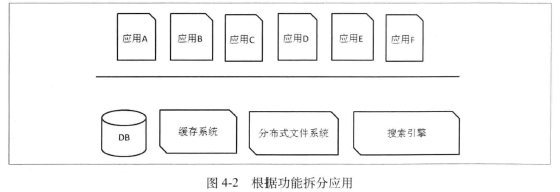
网站演进的变化包括，应用拆分，服务的拆分，数据的拆分和应用的解耦。

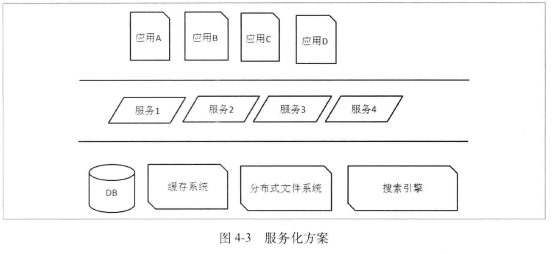
服务框架帮助应用进行拆分没完成服务化，数据处理帮助完成数据的拆分以及数据的管理，扩容，迁移；消息中间件帮助我们完成应用的解耦，提供分布式环境下完成事务的思路。

4.服务框架

4.1 网站功能持续丰富后的困境与应对







最初的网站结构，随着压力上升，第一想到增加应用服务器数量，但同时给数据库连接数带来压力，随着开发人员增多，出现代码冗余，对研发效率和稳定性造成了影响。

然后，想到把应用拆小。但连接数压力还在，多系统之间存在重复代码，使用共享库的方式，应用不太方便。

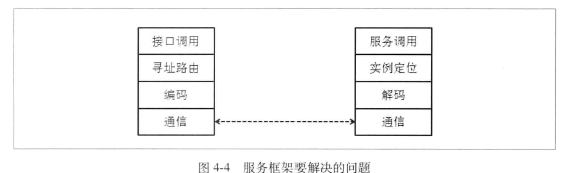
最终选择服务化方案，系统看起来更立体，应用之间有了直接访问。好处是，结构清晰，专人统一维护，提高代码质量，修改和发布次数会减少，底层的资源统一管理，提高效率。

但随着服务化的进行，应用数量分数增长，系统内部的依赖关系变得错综复杂。需要将团队也随之拆分来应对这些变化。

4.2 服务框架的设计与实现

4.2.1 应用从集中式走向分布式所遇到的问题

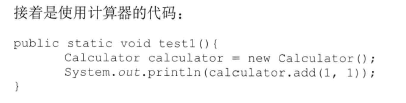
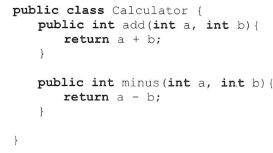
服务化会使原来的本地调用编程远程调用，我们要关心的是提高易用性以及降低性能损失。



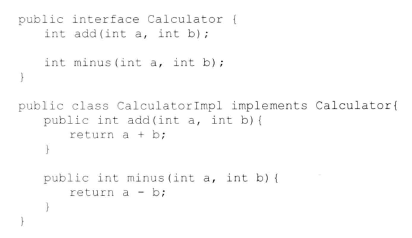
单机调用，只需要把程序计数器指向响应的入口地址，多机之间，需要对请求信息进行编码，然后传给远程的节点，解码后再进行真正的调用。

4.2.2 透过示例看服务框架原型

4.2.2.1 单机方式

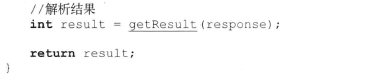


4.2.2.2 实现远程服务的调用客户端



从调用端开始看，重新实现Calculator





首先，根据调用的服务名称获取提供服务的机器地址列表，从可用的地址中选择一个要调用的目标机器。

通过请求发起方和服务提供方之间直连的方式，完成寻址和路由。然后构造请求的数据包，就是序列化后，进行通信。这样就完成了一次调用发起。请求后，需要等待远程服务的执行以及结果的返回，收到结果后，可以对数据进行java反序列化，这样就得到了执行结果。

4.2.2.3 实现服务端



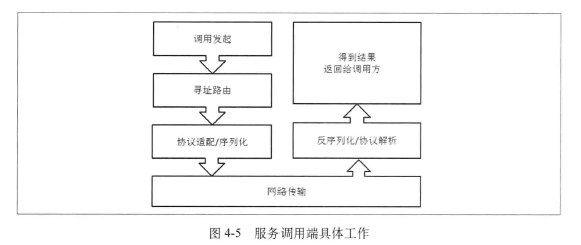
对于服务端，我们需要在启动后就进行监听。对于收到的数据需要一个反序列化的过程。

当我们拿到请求后，在本地定位具体提供的服务。具体实现，有一个服务注册表，根据名称和版本号进行管理。

通过反射进行服务调用，得到结果，序列化，然后通过网络协会给请求发送端。

4.2.3 服务调用端（客户端）的设计与实现

客户端具体实现



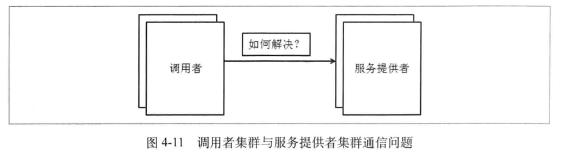
4.2.3.1 确定服务框架的使用方式

如何在客户端引入和使用服务框架

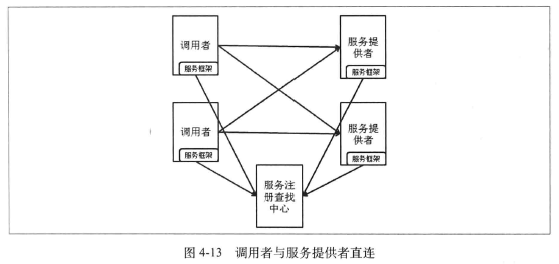
4.2.3.2 服务调用者与服务提供者之间通信方式的选择

使用服务框架是为了把本地对象之间的方法调用变为远程的过程调用（RPC）

(1) 远程通信遇到的问题



（2）采用透明代理与调用者、服务提供者直连的解决方案



服务注册中心对于调用者，只是提供可用的服务提供者的列表。但并不是每次都需要通过这个，把地址缓存在调用者本地，有变化时主动从中心发起通知，告诉调用者可用的服务提供列表的变化。

拿到地址列表后，路由要解决选择问题。首先要考虑的就是负载均衡，随机、轮询、权重是比较常见的实现方式。

4.2.3.3 引入基于接口、方法、参数的路由

一般用接口作为服务的粒度，也就是说一个服务就是一个接口的远程实现，

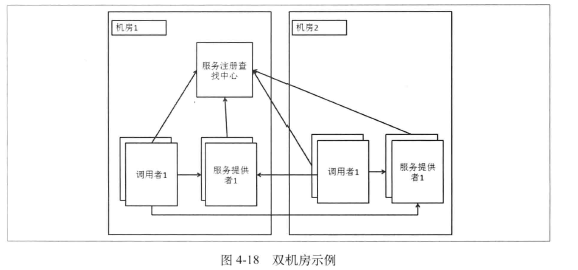
服务提供者执行调用者请求时，内部线程模型是一个线程对应一个请求，总的线程数有一个限制。并发请求过大，所有线程都工作，新的请求就需要排队。某个方法执行时间很长，就会出现所有线程都被这个接口方法占用。

我们可以增加线程数，但单机线程数总是有限，所以引申可以考虑增加机器数。但这种做法不太经济，也不可控。

第二种思路，隔离这些资源，快慢不同，重要级别不同的方法互不影响。

虽然集群代码相同，服务也相同，但是通过路由的策略，可以让请求之请求某一部分机器。

4.2.3.4 多机房场景



先不考虑服务注册中心的集群化，为了避免跨机房调用，有两个方案。第一，通过注册中心甄别不同的机房调用者集群，给他们不同服务提供者地址。另一种，通过路由完成。

4.2.3.5 服务调用端的流控处理

为了应对异常和可运维。需要进行流量控制。控制到服务提供者的请求的流量。基于两个维度考虑

① 根据服务端自身的接口、方法做控制。

②根据来源做控制。

4.2.3.6 序列化与反序列化处理

4.2.3.7 网络通信实现选择

4.2.3.8 支持多种异步服务调用方式

除了同步调用外，我们还需要支持如下几种调用

(1) Oneway ：只管发送请求，不关心结果。

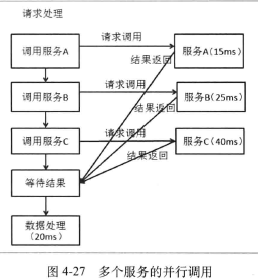
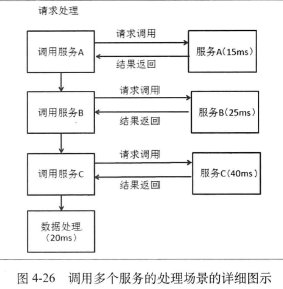
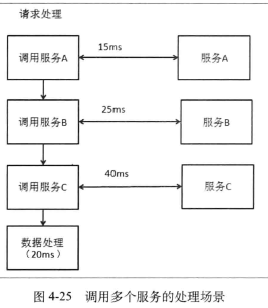
（2）Callback ：发送请求后继续执行自己的操作，等对方有响应时进行一个回调。

（3）Future ：

（4）可靠异步：一般通过消息中间件完成异步请求在远程执行

4.2.3.9 使用Future 方式对远程服务调用的优化

一个请求调用多个远程服务



一个请求处理三个远程服务，服务耗时加上线程自己处理时间，如果改变处理方式，顺序把请求发给服务，等到最后统一等待结果，然后进行本地处理。不过这种方式需要服务之间没有相互依赖，

4..2.4 服务提供端的设计与实现

4.2.4.1 如何暴露远程服务

服务端工作，一是对本地服务的注册管理，二是根据请求定位服务并执行。

4.2.4.2 服务端对请求处理的流程

4.2.4.3 执行不同服务的线程池隔离

把服务提供端工作线程当做一个线程池，我们可以进行线程池隔离，这样工作线程池就不是一个，通过服务名称、方法、参数确定具体执行服务，不会出现争抢线程资源的情况。

4.2.4.4 服务提供端的流控处理

整个服务框架看做产品，让调用变为远程的服务化。具体应用，需要改变，负载均衡在特定场景需要用权重。因此服务框架必须做到模块化且可配置；且模块可替换，并保留一定的扩展点来扩展原有功能。

4.2.5 服务升级

一旦使用服务框架，就意味着一定要做服务升级。对于升级有两种情况，第一是接口不变，对代码本身进行完善。只需要灰度发布的方式验证就行。第二种是修改原有的接口，又分两种情况。

（1）接口中增加方法。

（2）对接口某些方法修改调用的参数列表。

① 对使用原来方法的代码都进行修改，然和和服务端一起发布。

② 通过版本号来解决。

③ 从设计方法上考虑参数的扩展性。参数列表可扩展意味采用类Map方式来传递，不直观，对参数校验比较复杂。

4.3 实战中的优化

通过服务框架，将系统转变为分布式系统，但还有几个问题需要注意

（1）服务的拆分

（2）服务的粒度

（3）优雅和实用的平衡

（4） 分布式环境中的请求合并

4.4 为服务化护航的服务治理

服务治理是系统采用服务框架后，微服务化保驾护航的功能集合。

服务治理分为管理服务和查看服务，管理需要我们去控制、操作整个分布式系统中的服务，查看则是看运行时的状态或者一些具体信息、历史数据等。

服务查看包括

**·**服务信息：服务编码、编码注册、编码定位服务信息

·服务质量：服务排行、质量趋势、查询条件支持

·服务容量：

·服务依赖：

·服务分布：

·服务统计：

·服务元数据：

·服务查询：

·服务报表：

·服务监视：

·服务上下线

·服务路由：

·服务限流降级：

·服务归组：

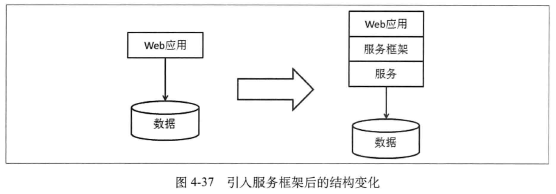
·服务线程池管理：

·机房规则：

·服务授权：

4.5 服务框架与ESB的对比

4.6 总结



5.数据访问层

数据量和访问量的上升，数据库会遇到问题，这就需要数据访问层出场了。

5.1 数据库从单机到分布式的挑战和应对

5.1.1 从应用使用单机数据库开始

网站初期，我们使用JDBC连接数据库，所有业务数据放在一个数据库中管理。

5.1.2 数据库垂直/水平拆分的困难

业务发展，数据量上升，我们更换更好的硬件，这是最简单的解决方案。但增长超过单机的极限，我们不升级硬件的情况下，只能给现有数据库减压。思路一，优化应用；二引入缓存、加搜索引擎等降低数据库压力，最后就是把数据库数据分到多台数据库上。

数据拆分有两种方式，垂直拆分和水平拆分。无论怎么拆分，都会带来影响。

垂直:

·原来事务逻辑被影响,要么放弃原来事务,修改实现,要么引入分布式事务.

·join操作不能用 ·靠外键约束场景被影响

水平

·事务存在被打破情况 ·join被影响 ·外键约束有影响

·不能依赖单库的自增序列生成唯一ID ·单表查询要跨库

5.1.3 单机变为多机后，事务如何处理

5.1.3.1 了解分布式事务的知识

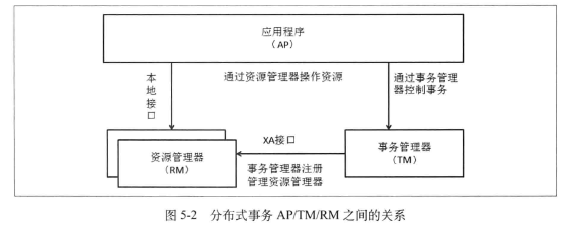
（1）分布式事务模型与规范

X/Open定义事务处理模型DTP模型，有三个组件

Application Program（AP）应用程序，定义事务边界，定义了构成该事务的应用程序的特定操作。

Resource Manager（RM）资源管理器，消息服务器管理系统。对资源进行控制，让它实现XA定义接口，并提供了存储共享资源的支持。

Transaction Manager（TM）,事务管理器，负责协调和管理事务，提供接口。TM向事务指定标识，监视它们的进程，负责处理事务的完成和失败。

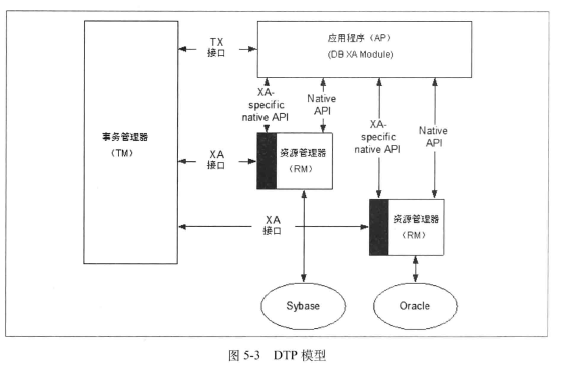


事务：一个事务是一个完整的工作单元，由多个独立的计算任务组成，这多个任务在逻辑上是原子的。

全局事务：一次性操作多个资源管理器的事务就是全局事务。

分支事务：全局事务下，每个资源管理器有自己独立的任务，这些任务的集合是这个资源管理器的分支任务

控制线程：用来表示一个工作线程，用来标识全局事务的分支事务关系的线程。



2 两阶段提交

2PC:Two Phase Commitment Protocol,相对单库，在分布式系统中，在提交之前增加准备阶段。

由于事务管理器自身的稳定性、可用性的影响，以及网络通信问题，还有在多个资源之间进行协调，要进行很多日志记录的工作，交互次数增多，使得开销增大。

5.1.3.2 大型网站一致性的基础理论——CAP/BASE

CAP涵义：

Consistency：所有节点在同一时间读到同样的数据。数据上的一致性。

Availability：无论成功还是失败，每个请求都能收到一个反馈。数据上的可用性。

Partition-Tolerance：系统有部分问题或者消息有丢失，但系统仍能够继续运行。分区容忍性。

因为无法同时满足三项，我们可以选择其中两个提升。在分布式系统中，我们选择AP，加强可用性和分区容忍性，而牺牲一致性。

BASE模型涵义：

Basically Available ：基本可用，允许分区失败

Soft state：软状态，接受一段时间的状态不同步。

Eventually consistent：最终一致。

分布式中选择AP，对于C，我们采用方式是保证最终一致。

5.1.3.3 比两阶段提交更轻量一些的Paxos协议

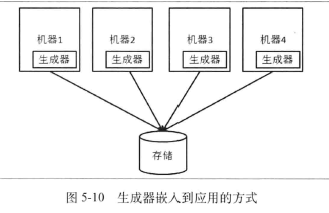
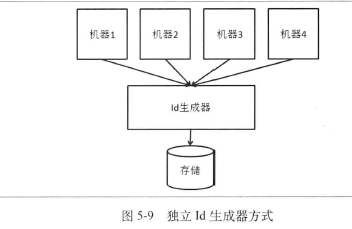
Paxos协议是帮助我们解决分布式系统中一致性问题的一个方案，前提是不存在拜占庭将军问题。是指没有办法保证可信的通信环境的问题。也就是说要求信息都是准确的。

Paxos不容易理解，但核心是少数服从多数。

5.1.3.4 集群内数据一致性的算法实例

5.1.4 多机的Sequence问题与处理

自增需要解决两个问题：唯一性和连续性。

5.1.5 应对多机的数据查询

5.1.5.1 跨库join

在应用层把原来数据库join操作分成多次的数据库操作；数据冗余；借助外部系统解决一些跨库问题。

5.1.5.2 外键约束

5.1.5.3 跨库查询的问题及解决

（1）数据库分库分表的演化

从逻辑上说，独立信息应该放在一起，但是大数据的情况下，就需要分库分表，从物理上分成多个库的多张表上。应用上，对这个表的查询就需要跨库跨表的合并。

（2）从具体例子上看分库分表后查询的问题

假设我们有一个用户表，我们从中查询某个省份一定年龄范围的用户。单表很简单，但分库分表后，就要根据分割方式进行查询。

如果同省份在一个数据库同一张表，这就成了单库单表的问题。但如果查询多省信息，或者多人等复杂操作。

（1）排序，多个来源的数据查询出来后，未排序，就要进行一个全排序。

（2）函数处理，MAX,MIN,SUM,COUNT等函数对多个数据来源的值进行相应的函数处理。

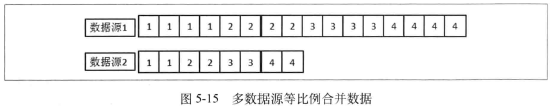
（3）求平均值

（4）非排序分页，有同等步长，同等比例。

举例说明：我们从两个数据源分别得到16和8条，每页4条分页，前四页包含两个数据源各两条，剩下的全是第一个数据源数据。

等比例就是，每个数据源数量不同，但每页占总数比例相同





(5)排序后分页，数据按照某些条件排序并进行分页显示。

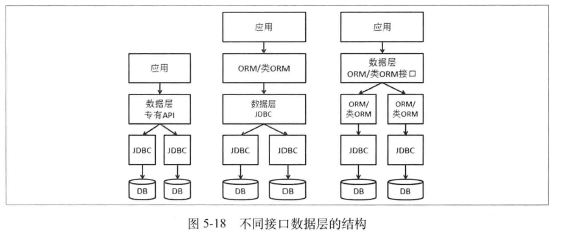
排序分页需要把每个数据源的前两页都取回来进行归并排序，才能等到正确结果，越往后翻页，承受的负担越重。应尽量避免这种方式。

5.2 数据访问层的设计与实现

5.2.1 如何对外提供数据访问层的功能

5.2.1.1 对外提供数据访问层的方式

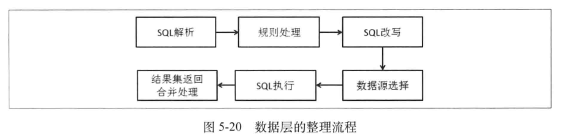
数据以怎样方式呈现应用？（1）为用户提供专有API（2）通用的方式。（3）基于ORM接口方式。



5.2.1.2 不同提供方式之间在合并查询场景下的对比

直接JDBC驱动方式可以减少生成的对象，优势比较明显。

5.2.2 按照数据层流程的顺序看数据层设计



5.2.2.1 SQL解析阶段的处理

SQL解析问题，一是对sql的支持程度，二是支持多少sql方言，都需要结合实际去做选择。解析的缓存需要数量上限，通过解析得到关键信息，表名，字段，where条件。

也可以通过提示实现，不用去解析整个sql，但解析不完备，不带有分库条件。

5.2.2.2 规则处理阶段

(1) 采用固定哈希算法作为规则

根据某个字段去模，然后将数据分散到不同的数据库和表中。规则设置和实现比较简单，但扩容比较复杂。

（2）一致性哈希算法带来的好处

把节点对应的哈希值变为一个范围，不再是离散的。增加或减少一个节点，原来节点都只有一个受影响，但被影响的节点夜里与其他节点不同，似乎需要增加一倍节点或减去一半节点去保持负载均衡。

（3）虚拟节点对一致性哈希的改进

通过一个物理节点对应非常多的虚拟节点，解决负载不均衡问题。

（4）映射表与规则自定义计算方式

这些都是对规则的一个补充，扩展能力最强

5.2.2.3 为什么要改写SQL

5.2.2.4 如何选择数据源

原来的数据库变为一个数据库矩阵，决定分组后，还需决定分组中的那个库，根据当前要执行的SQL特点，是否在事务中以及各个库的权重规则，计算得到这次SQL请求访问的数据库

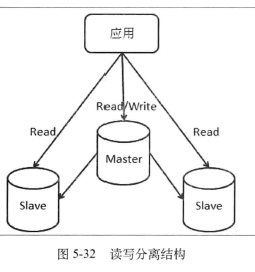
5.2.2.5 执行SQL和结果处理阶段

5.2.2.6 实战经验分享

（1）复杂的连接管理 （2）三层数据源的支持和选择

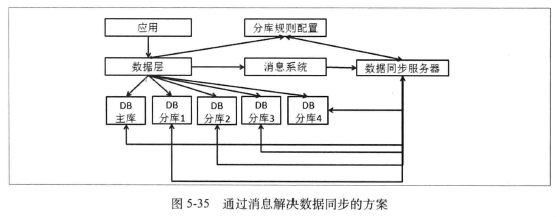
5.2.3 独立部署的数据访问层实现方式

5.2.4 读写分离的挑战和应对



5.2.4.1 主库从库非堆成的场景

（1）数据结构相对，多从库对应一主库的场景



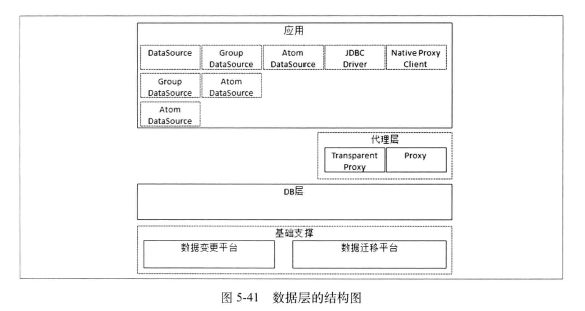
（2）主、从库分库方式不同的数据复制

（3）引入数据变更平台

5.2.4..2 如何做到数据平滑迁移

数据库平滑迁移的最大挑战是，迁移过程中有数据的变化。进行迁移时，记录增量的日志，迁移结束后，在对增量的变化进行处理。最后，把要迁移的数据写暂停，保证日志处理完毕后，再切换规则，放开限制。

5.3 总结



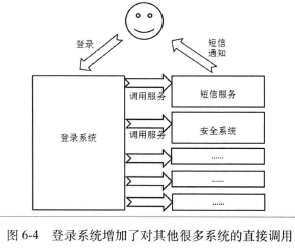
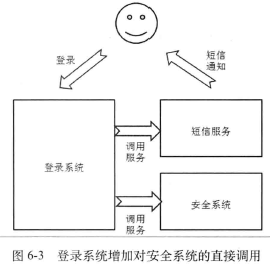
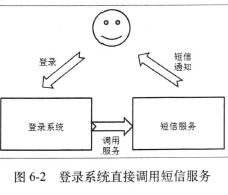
6 消息中间件

6.1 消息中间件的价值

6.1.1 消息中间件的定义

6.1.2 透过示例看消息中间件对应用的解耦

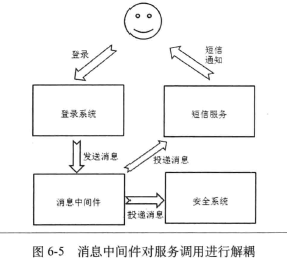
6.1.2.1 透过服务调用让其他系统感知事件发生的方式



在登录成功后增加需要被调用的系统，使登录系统被迫依赖非常多的系统。

6.1.2.2 通过引入消息中间件解耦服务调用

从登录系统角度看，这些系统不是必须，相反，他们依赖登录系统。因此，我们通过消息中间件把上面的结构解耦，



我们必须保证这个登录成功消息发送到了消息中间件

6.2 互联网时代的消息中间件

JMS:Java Message Service，一个关于消息的规范，而ActiveMQ等是对这个规范的实现。

6.2.1 如何解决消息发送一致性

6.2.1.1 消息发送一致性定义：产生消息的业务动作与消息发送的一致。

6.2.1.2 消息发送一致性很难保证吗

6.2.1.3 大家熟知的JMS有办法吗

JMS重要要素

Destination：用来定义消息从发送后要走的通道

ConnectionFactory：用于创建连接的对象

Connction：连接接口，创建Session

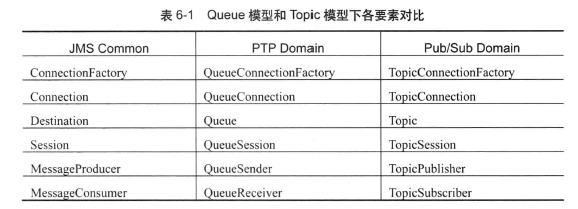
Session：会话接口，创建消息的发送者，接收者以及消息对象本身

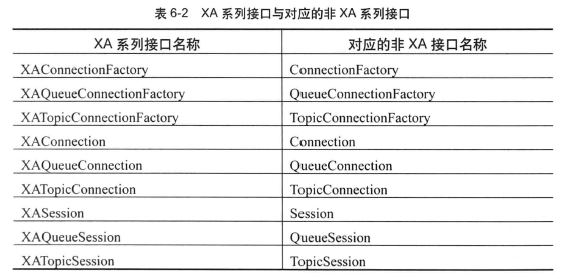
MessageConsumer：消息的消费者，订阅消息并处理的对象

MessageProducer：消息的生产者，发送消息的对象

XXXMessage：各类消息对象，有BytesMessage，MapMessage，ObjectMessage，StreamMessage和TextMessage5种。

JMS有Queue和Topic之分，他的要素都有自己的子接口。





如果不适用XA接口实现，无法直接得到发送消息给消息中间件和业务操作者两个事情的事务保证。而如果使用，就需要引入分布式事务。

6.2.1.4 有其他的办法吗

（1）应用先把消息发给消息中间件，标记消息为待处理状态

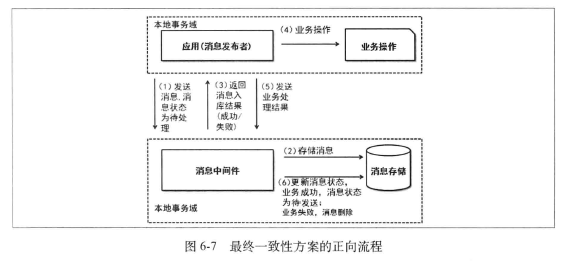
（2）中间件受到消息后，把消息存储在消息存储中，并不投递该消息

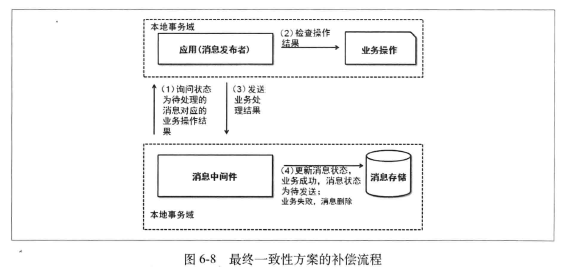
（3）中间件返回消息处理结果，结果是成功或者失败。

（4）业务方受到消息中间件返回的结果并进行处理，失败放弃业务处理，成功进行业务操作

（5）业务操作完成，把操作结果发送给中间件

（6）根据操作结果，进行处理，失败删除消息存储中的消息，成功，也更新消息存储中的消息状态为可发送，进行调度，进行消息的投递。



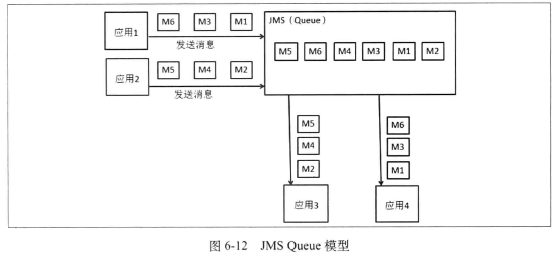


6.2.2 如何解决消息中间件与使用者的强依赖问题

6.2.3 消息模型对消息接收的影响

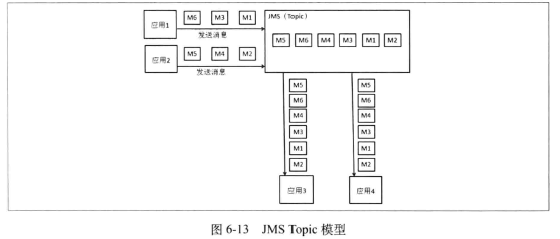
JMS有Queue（点对点）和Topic（发布/订阅）模型

6.2.3.1 Queue模型



应用1、2发送消息到JMS服务器，消息根据到达的顺序形成一个队列，应用3/4进行消息的消费。消息从发送端发送出来不能确定最终会被那个应用消费，但可以确定只有一个会去消费。

6.2.3.2 Topic模型



相比于queue模型，Topic区别在于接收的部分。

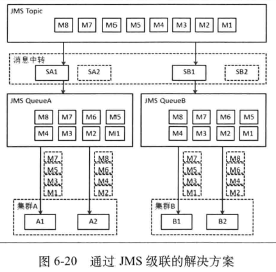
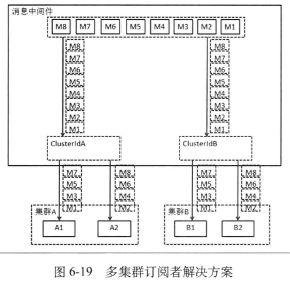
6.2.3.3 JMS中客户端连接的处理和带来的限制

6.2.3.4 我们需要什么样的消息模型

我们的需求：消息发送方和接收方都是集群，同一个消息接收方可能有多个集群进行消息的处理，不同集群对于同一条消息的处理不能相互干扰。

实际上，queue模型接受消息不完整，而Topic却有重复。我们需要跳出这两种模型。

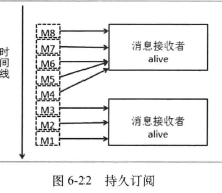
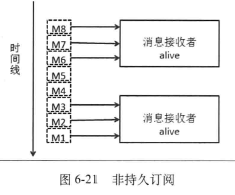
将两者结合，集群对集群用Topic，而集群内部用Queue处理。



6.2.4 消息订阅者订阅消息的方式

消息中间件，提供对于消息的可靠保证是非常重要的事情。

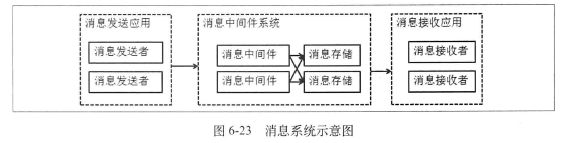
这里介绍持久订阅和非持久订阅两种订阅方式。



非持久：消息接受者启动时，建立订阅关系，可以收到消息，结束时，订阅关系就不存在了。

持久订阅：订阅关系一旦建立，除非显式取消订阅关系，否则订阅关系一直存在。

6.2.5 保证消息可靠性的做法



消息发送者发送消息给消息中间件，消息中间件把消息存入消息存储，消息中间件把消息投递给消息接受者。

要保证这三个阶段都可靠。

6.2.5.1 消息发送端可靠性的保证

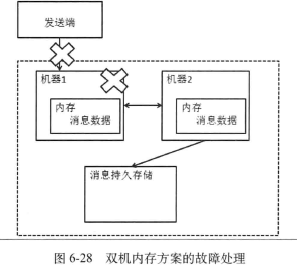
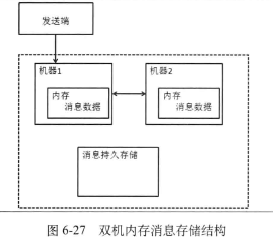
6.2.5.3 消息存储的可靠性保证

消息数据一定要放到外存储器上，成熟存储系统需要积累和沉淀，不建议自主实现持久存储的功能，如果有自信，可以去开发。

（1）实现基于文件的消息存储

（2）采用数据库作为消息存储

（3）基于双机内存的消息存储



6.2.5.3 消息系统的扩容处理

（1）消息中间件自身如何扩容

（2）消息存储的扩容处理

6.2.5.4 消息投递的可靠性保证

（1）消息投递简介

（2）投递处理的优化

6.2.6 订阅者时间的消息重的产生和应对

6.2.6.1 消息重复的产生原因

第一类，发送端重复发送

（1）中间件受到消息并成功存储后出现问题，导致重复发送

（2）中间件响应慢，返回成功结果超时

（3）存储后，网络出现问题

各种原因使发送端没有收到成功的返回结果，又有重试机制，因而导致重复，解决方案：重试使用同样的消息ID.

第二类，消息中间件向外投递时重复

原因是因为消息接收者成功处理后，消息中间件不能及时更新投递状态造成的。分布式事务可以解决，但成本比较高，另一种是要求消息接收者处理重复情况。是幂等操作

幂等（idempotence）数学概念，用于抽象代数，定义如下：

·在某二元运算下，幂等元素是指被自己重复运算的结果等于它自身的元素

·某一元运算为幂等时，其两次作用在任一元素后回合其作用一次的结果相同。

简单来说就是消息接收端的操作，即使重复，结果不会变化。降低了中间件的复杂性，但给接收端带来限制和门槛。

6.2.6.2 JMS的消息确认方式与消息重复的关系

JMS中，消息确认有三种选择

·AUTO\_ACKNOWLEDGE 自动确认，消息不可靠

·CLIENT\_ACKONWLEDGE 客户端确认，将控制权交给了接收消息的客户端应用

·DUPS\_OK\_ACKNOWLEDGE 消息接收方处理函数执行结束后确认。

会出现 至少一次和至多一次两种情况。

6.2.7 消息投递的其他属性支持

（1）消息优先级

（2）订阅者消息处理顺序和分级订阅

（3）自定义属性

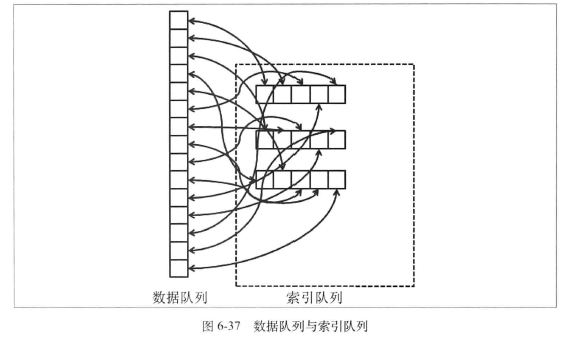
（4）局部顺序

6.2.8 保证顺序的消息队列的设计



6.2.8.1 单机多队列的问题和优化

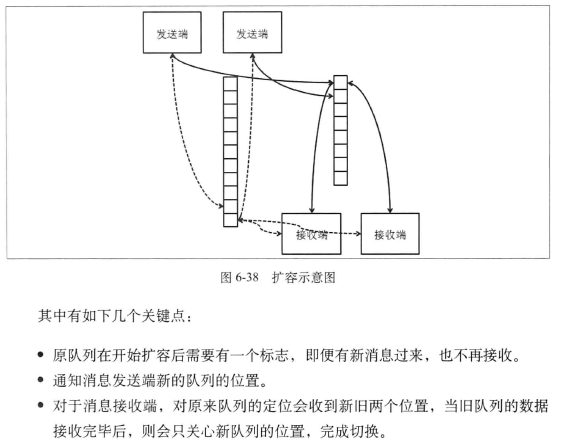
单机多队列的隔离完成了对消息的有序支持，但队列过多，消息写入就接近于随机写了。



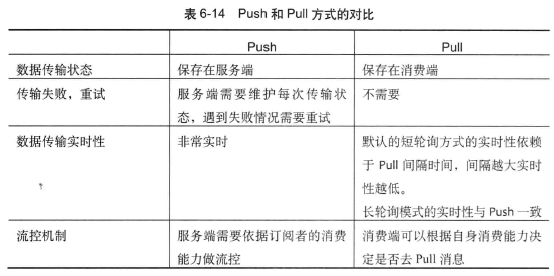
6.2.8.2 解决本地消息存储的可靠性

我们考虑采用消息同步复制的方式解决可靠性的问题

6.2.8.3 如何支持队列的扩容



6.2.9 Push和Pull方式的对比

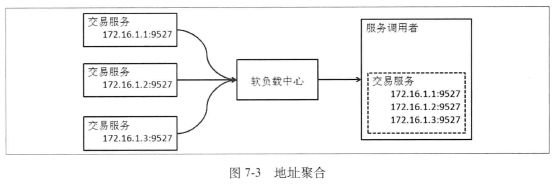


7.软负载中心与集中配置管理

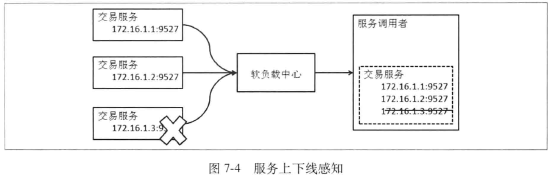
7.1 初识软负载中心

软负载中心有两个最吃出的职责：

（1）聚合地址信息。



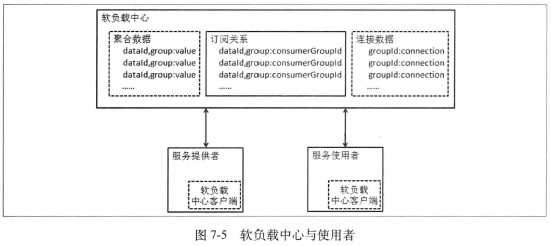
（2）生命周期感知，对服务上下线自动感知，根据变化更新服务地址数据。



7.2 软负载中心的结构

（1）服务端：感知提供服务的机器是否在线，聚合提供者机器信息，把数据传给使用数据的应用。

（2）客户端：把服务提供者的具体信息主动传给服务端，并且随着变化更新数据；想服务端告知自己所需要的数据并负责去更新数据，还要进行本地的数据缓存。



·聚合数据：聚合后的地址信息列表，是一个Key-Value的结构

·订阅关系：

·连接数据：连接到软负载中心的节点和软负载中心已经建立的连接的管理。

7.3 内容聚合功能的设计

·保证数据正确性·高效聚合数据

用一个Map的结构进行数据管理，通过dataId和group定位数据去处理。

（1）并发下的数据正确性的保证：采用加锁或线程安全容器来保证

（2）数据更新、删除的顺序保证：采用NIO方式通信，在插入数据时判断当前产生数据的发布者连接是否还存在。

（3）大量数据同时插入，更新时的性能保证

7.4 解决服务上下线的感知

实现方式：（1）通过客户端与服务端的连接感知

（2）通过对于发布数据中提供的地址端口进行连接的检查

7.5 软负载中心的数据分发的特点和设计

7.5.1 数据分发与消息订阅的区别

（1）消息中间件要保证消息不丢失，而软负载中心只需要保证最新数据送到相关的订阅者。

（2）消息中间件，同一个集群的不同机器处理不同的消息，而软负载中心，要把数据分发给所有的机器。

7.5.2 提升数据分发性能需要注意的问题

·数据压缩

·全量与增量的选择

7.6 针对服务化的特性支持

7.6.1 软负载数据分组

分组是为了进行隔离。

·根据环境进行区分

·分优先级的隔离

7.6.2 提供自动感知以外的上下线开关

·优雅的停止应用

·保持应用场景，用于排错

7.6.3 维护管理路由规则

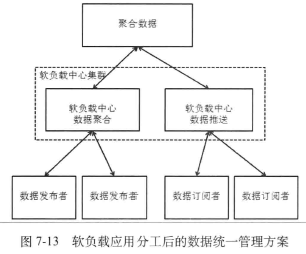
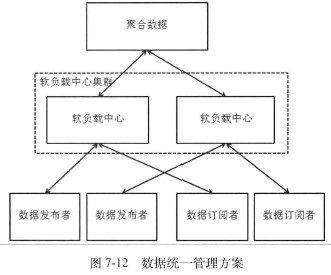
7.7 从单机到集群

软负载中心从单机走向集群，需要解决的问题

·数据管理问题

·连接管理问题

7.7.1 数据统一管理方案



7.7.2 数据对等管理方案

7.8 集中配置管理中心

7.8.1 客户端实现和容灾策略

7.8.2 服务端实现和容灾策略

7.8.3 数据库策略

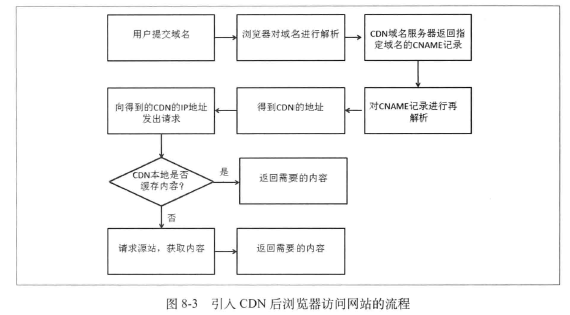
8构建大型网站的其他要素

8.1 加速静态内容访问速度的CDN

Content Delivery NetWork 内容分发网络。把用户需要的内容分发到离用户近的地方。

分为CDN源站和CDN节点，是一种网络缓存技术，节省广域网的带宽消耗，提升用户的访问速度。





CDN关键技术：

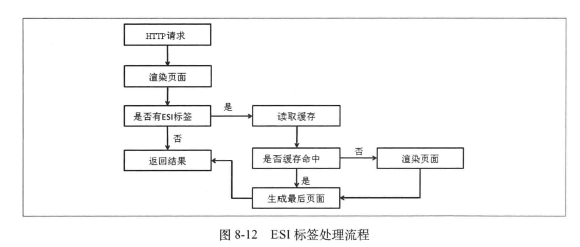
·全局调度 ·缓存技术 ·内容分发 ·带宽优化

8.2 大型网站的存储支持

8.2.1 分布式文件系统

8.2.2 NoSQL

8.2.3 缓存系统



8.3 搜索系统

当网站的数据量和访问量逐步增大时，就需要站内使用搜索技术。

8.3.1 爬虫问题

更新索引

·定时从数据源拉取

·通过数据变更的通知，系统压力大，用于对实时性要求很高 的场景

8.3.2 倒排索引

8.3.3 查询预处理

8.3.4 相关度计算

8.4 数据计算支撑

8.5 发布系统

（1）分发应用

（2）启动校验

（3）灰度发布

（4）产品改版Beta

8.6 应用监控系统

·数据监视维度

·数据记录方式

·数据采集方式

·展现与告警

8.7 依赖管理系统

8.8 多机房问题分析

8.9 系统容量规划

8.10 内部私有云