程序扩展

谢志杰 1352975

* 分布式
* 数据一致性
* 负载均衡
* ID分配
* 通信可靠高效
* 消息队列

分布式

* 性能可扩展：性能无法完全实现线性扩展，但要尽量使用具有并发性和异步性的组件。具备完成通知功能的工作队列要优于同步连接到数据库。
* 可用性可扩展：CAP理论表明，分布式系统无法同时提供一致性、可用性和分区容错性保证。许多大规模Web应用程序都为了可用性和分区容错性而牺牲了强一致性，而后者则有赖于最终一致性来保证。
* 维护可扩展：软件和服务器都需要维护。在使用平台&工具监控和更新应用程序时，要尽可能地自动化。
* 成本可扩展：总拥有成本包括开发、维护和运营支出。在设计一个系统时，要在重用现有组件和完全新开发组件之间进行权衡。现有组件很少能完全满足需求，但修改现有组件的成本还是可能低于开发一个完全不同的方案。另外，使用符合行业标准的技术使组织更容易聘到专家，而发布独有的开源方案则可能帮助组织从社区中挖掘人才。

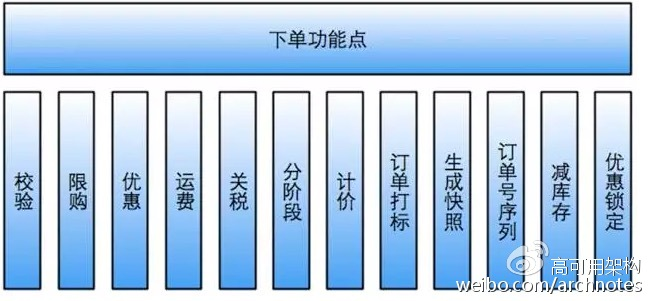
以上各项，他们在设计应用程序时都会考虑和权衡。下面是他们根据上述内容总结出的10个设计原则：

1. 避免单点故障：任何东西都要有两个。这增加了成本和复杂度，但却能在可用性和负载性能上获益。而且，这有助于设计者采用一种分布式优先的思维。
2. 横向扩展，而不是纵向扩展：升级服务器（纵向）的成本是指数增长的，而增加另一台商用服务器（横向）的成本是线性增长的。
3. 尽量减少应用程序核心所需要完成的工作。
4. API优先：将应用程序视为一个提供API的服务，而且，不假定服务的客户端类型（手机应用、Web站点、桌面应用程序）。
5. 总是缓存。
6. 提供尽可能新的数据：用户可能不需要立即看到最新的数据，最终一致性可以带来更高的可用性。
7. 设计时要考虑维护和自动化：不要低估应用程序维护所需要的时间和工作量。软件首次公开发布是一个值得称赞的里程碑，但也标志着真正的工作要开始了。
8. 宁异步，不同步。
9. 努力实现无状态：状态信息要保存在尽可能少的地方，而且要保存在专门设计的组件中。
10. 为故障做好准备：将故障对终端用户的影响最小化。

数据一致性

蘑菇街交易创建过程中的分布式一致性方案

交易创建的一般性流程

我们把交易创建流程抽象出一系列可扩展的功能点，每个功能点都可以有多个实现（具体的实现之间有组合/互斥关系）。把各个功能点按照一定流程串起来，就完成了交易创建的过程。

面临的问题

每个功能点的实现都可能会依赖外部服务。那么如何保证各个服务之间的数据是一致的呢？比如锁定优惠券服务调用超时了，不能确定到底有没有锁券成功，该如何处理？再比如锁券成功了，但是扣减库存失败了，该如何处理？

方案选型

服务依赖过多，会带来管理复杂性增加和稳定性风险增大的问题。试想如果我们强依赖 10 个服务，9 个都执行成功了，最后一个执行失败了，那么是不是前面 9 个都要回滚掉？这个成本还是非常高的。

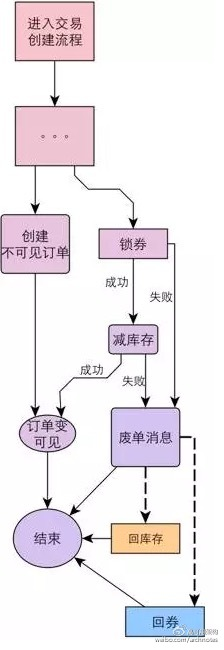
所以在拆分大的流程为多个小的本地事务的前提下，对于非实时、非强一致性的关联业务写入，在本地事务执行成功后，我们选择发消息通知、关联事务异步化执行的方案。

消息通知往往不能保证 100% 成功；且消息通知后，接收方业务是否能执行成功还是未知数。前者问题可以通过重试解决；后者可以选用事务消息来保证。

但是事务消息框架本身会给业务代码带来侵入性和复杂性，所以我们选择基于 DB 事件变化通知到 MQ 的方式做系统间解耦，通过订阅方消费 MQ 消息时的 ACK 机制，保证消息一定消费成功，达到最终一致性。由于消息可能会被重发，消息订阅方业务逻辑处理要做好幂等保证。

所以目前只剩下需要实时同步做、有强一致性要求的业务场景了。在交易创建过程中，锁券和扣减库存是这样的两个典型场景。

要保证多个系统间数据一致，乍一看，必须要引入分布式事务框架才能解决。但引入非常重的类似二阶段提交分布式事务框架会带来复杂性的急剧上升；在电商领域，绝对的强一致是过于理想化的，我们可以选择准实时的最终一致性。

我们在交易创建流程中，首先创建一个不可见订单，然后在同步调用锁券和扣减库存时，针对调用异常（失败或者超时），发出废单消息到MQ。如果消息发送失败，本地会做时间阶梯式的异步重试；优惠券系统和库存系统收到消息后，会进行判断是否需要做业务回滚，这样就准实时地保证了多个本地事务的最终一致性。

负载均衡

负载均衡在不同的领域有不同的解决方案，在这里我只讨论对于大型网站的负载均衡解决方案。

1 DNS负载均衡

最早的负载均衡技术，利用域名解析实现负载均衡，在DNS服务器，配置多个A记录，这些A记录对应的服务器构成集群。大型网站总是部分使用DNS解析，作为第一级负载均衡。

优点

使用简单：负载均衡工作，交给DNS服务器处理，省掉了负载均衡服务器维护的麻烦

提高性能：可以支持基于地址的域名解析，解析成距离用户最近的服务器地址，可以加快访问速度，改善性能；

缺点

可用性差：DNS解析是多级解析，新增/修改DNS后，解析时间较长；解析过程中，用户访问网站将失败；

扩展性低：DNS负载均衡的控制权在域名商那里，无法对其做更多的改善和扩展；

维护性差：也不能反映服务器的当前运行状态；支持的算法少；不能区分服务器的差异（不能根据系统与服务的状态来判断负载）

实践建议

将DNS作为第一级负载均衡，A记录对应着内部负载均衡的IP地址，通过内部负载均衡将请求分发到真实的Web服务器上。一般用于互联网公司，复杂的业务系统不合适使用。

2 IP负载均衡

在网络层通过修改请求目标地址进行负载均衡。

用户请求数据包，到达负载均衡服务器后，负载均衡服务器在操作系统内核进程获取网络数据包，根据负载均衡算法得到一台真实服务器地址，然后将请求目的地址修改为，获得的真实ip地址，不需要经过用户进程处理。

真实服务器处理完成后，响应数据包回到负载均衡服务器，负载均衡服务器，再将数据包源地址修改为自身的ip地址，发送给用户浏览器。

IP负载均衡，真实物理服务器返回给负载均衡服务器，存在两种方式：（1）负载均衡服务器在修改目的ip地址的同时修改源地址。将数据包源地址设为自身盘，即源地址转换（snat）。（2）将负载均衡服务器同时作为真实物理服务器集群的网关服务器。

优点：

（1）在内核进程完成数据分发，比在应用层分发性能更好；

缺点：

（2）所有请求响应都需要经过负载均衡服务器，集群最大吞吐量受限于负载均衡服务器网卡带宽；

3 链路层负载均衡

在通信协议的数据链路层修改mac地址，进行负载均衡。

数据分发时，不修改ip地址，指修改目标mac地址，配置真实物理服务器集群所有机器虚拟ip和负载均衡服务器ip地址一致，达到不修改数据包的源地址和目标地址，进行数据分发的目的。

实际处理服务器ip和数据请求目的ip一致，不需要经过负载均衡服务器进行地址转换，可将响应数据包直接返回给用户浏览器，避免负载均衡服务器网卡带宽成为瓶颈。也称为直接路由模式（DR模式）。

优点：性能好；

缺点：配置复杂；

实践建议：DR模式是目前使用最广泛的一种负载均衡方式。

4 混合型负载均衡

由于多个服务器群内硬件设备、各自的规模、提供的服务等的差异，可以考虑给每个服务器群采用最合适的负载均衡方式，然后又在这多个服务器群间再一次负载均衡或群集起来以一个整体向外界提供服务（即把这多个服务器群当做一个新的服务器群），从而达到最佳的性能。将这种方式称之为混合型负载均衡。

此种方式有时也用于单台均衡设备的性能不能满足大量连接请求的情况下。是目前大型互联网公司，普遍使用的方式。

以上模式适合有动静分离的场景，反向代理服务器（集群）可以起到缓存和动态请求分发的作用，当时静态资源缓存在代理服务器时，则直接返回到浏览器。如果动态页面则请求后面的应用负载均衡（应用集群）。

以上模式，适合动态请求场景。

因混合模式，可以根据具体场景，灵活搭配各种方式，以上两种方式仅供参考。

ID分配

前言

 生产系统随着业务增长总会经历一个业务量由小变大的过程，可扩展性是考量数据库系统高可用性的一个重要指标;在单表/数据库数据量过大，更新量不断飙涨时，MySQL DBA往往会对业务系统提出sharding的方案。既然要sharding，那么不可避免的要讨论到sharding key问题，在有些业务系统中，必须保证sharding key全局唯一,比如存放商品的数据库等，那么如何生成全局唯一的ID呢，下文将从DBA的角度介绍几种常见的方案。

方案一

专门建一个库/表分配全局唯一的ID

1.创建一个生成ID的表(auto\_increment id），要插入新业务数据时，开启事务，获取自增ID，使用此自增ID进行业务数据插入。 这个方法的问题在于每次生成ID都要开事务生成和查询新的ID，在高并发系统下资源竞争剧烈，效率极低，分配ID的表会成为整个系统的瓶颈。

2.使用max(id)方案，创建一个表存放每个表的max id,需要插入新的业务数据时，查看max\_id表找到当前ID中最大的ID，将此ID+1分配给新插入的业务数据，同时更新该业务数据所表对应的表的max id。这个方法的问题同上，唯一的好处是不会使分配ID的表的size变的很大。

方案二

每张表划分区域

  创建表时为每张表指定不同的AUTO\_INCREMENT起始值。比如第一张表从1开始，第二张表从100000开始,当然这样显然存在一个问题，当第一张表的数据量，或者分配的ID超过100000时，就会导致ID重复，所以必须监控每个表的自增ID增长状况，当ID增长过快时，需要通过命令调整AUTO\_INCREMENT起始值。

方案三

错位分配

 在设计阶段，通过指定每个sharding 表的 AUTO\_INCREMENT 和步长来实现错位分配。需要注意的时，步长值要大于sharding的数目， 否在会导致重复分配ID，步长值-sharding数目=未来可扩展的sharding数量，这种方案的局限性也就在此，当布长值等于sharding数目时，想要再进行扩展是很复杂的。

方案四

组合ID

1.使用sharding编号作为前缀+自增ID生成全局ID,需要再每个sharding的库或者实例下创建一个生产ID的表。这种实现的缺点在于每个sharding的实例或者逻辑下需要一个单独生产ID的表。

2.组合主键，sharding表中增加一列存储sharding标识，用这一列和自增列共同作为主键，实现全局唯一。这种实现的缺点在于联表操作比较复杂，增加了存储成本。

其他

使用GUID\UUID或者twitter的snowflake等实现。

总结

方案一的两种实现都是非分布式的，也就是说所有的sharding都要依赖这个系统，那么这个系统很容易成为瓶颈，适合业务和并发量不大的系统,方案二和方案三都是通过调整和设计MySQL的AUTO\_INCREMENT实现，但是都有各自的局限性。方案四是一种分布式的方案，各个sharding之间互不影响,不存在方案一的单点瓶颈问题，是比较好的实现方案。以上几种方案都是从DBA的角度提供的方案，UUID或者snowflake是业务端实现的方式,这里就不在赘述。

通信可靠高效

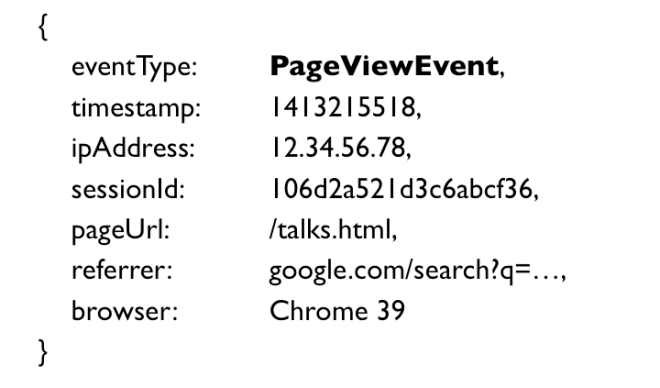
IBM 消息中间件概述：

MQ 消息中间件为用户和应用开发人员提供了一种直接，简单的手段以实现应用系统在不同操作系统平台之间稳定可靠地传递，交换重要的数据和信息，确保消息不丢失/不复传。 MQ 消息传输产品和企业 IT 应用的关系，就如同电子邮件和人一样，是 SOA 时代企业 IT 应用之间相互传递消息的最重要工具。但是，与由于网络原因而经常丢失消息的脆弱的电子邮件系统不同的是，MQ 消息中间件强大而稳定，永远能够确保每一个字节的消息都能够被正确地传送到目的应用之中。

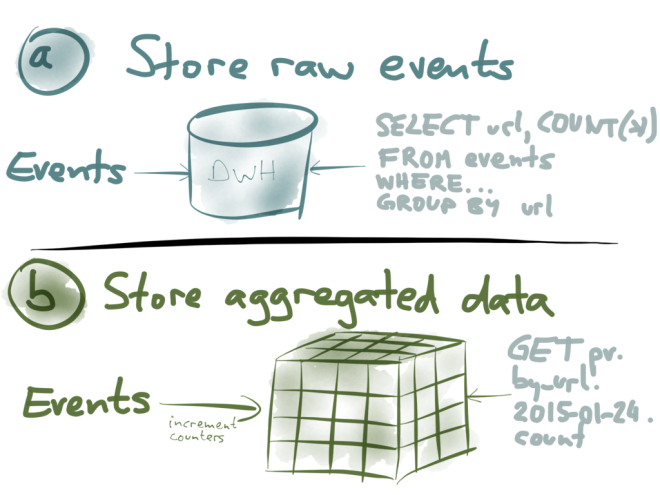
消息队列

关于事件流处理，在不同的场景中有不同的概念。有人称之为流处理，有人称之为事件溯源或CQRS，还有人称之为“复杂事件处理（Complex Event Processing）”。不管名称是什么，它们的基本原则都是一样的。Martin Kleppmann是Apache Samza的贡献者。在本文中，我们将跟随他的思路深入理解这些概念，以便帮助我们设计更好的系统。

“流处理（stream processing）”源于LinkedIn构建大规模数据系统的经验，并在开源项目Apache Kafka和Apache Samza中实现。Martin以Google Analytics为例具体介绍了这一概念。Google Analytics是一小段JavaScript代码，可以追踪哪个访问者访问了哪个网页。然后，系统管理员可以研究这些数据，并按照时间段、URL等划分这些数据。为了实现这个目的，每次用户访问一个页面时，就需要记录一个事件来反映这个事实。页面访问事件可能是（图1）这样的结构：



每个事件都是包含上述信息的一个简单不变的事实。它只简单地记录已发生的事情。然后，我们就可以从这些页面访问事件中生成图形仪表板。通常来说，这些事件可以使用（图2）所示的其中一种方式存储：

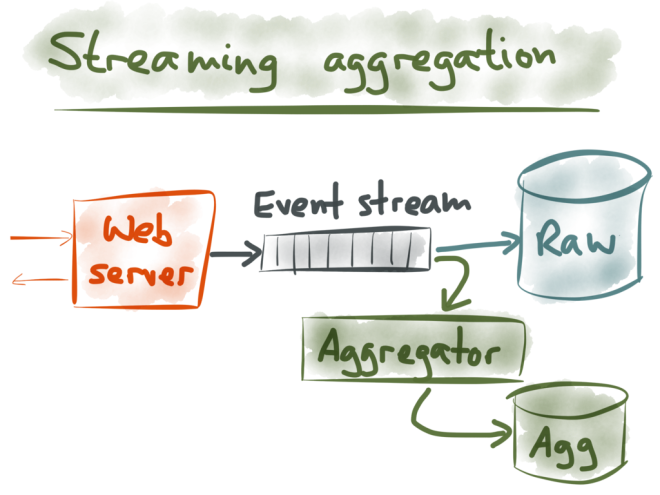


选项（a）：在每个事件进来的时候将其存储，并把它们全部转存到一个大型的数据库、数据仓库或Hadoop集群中。在需要时，就可以在数据集上执行查询。这个过程会扫描所有事件，或者至少是某个大型的数据子集，并动态地完成聚合。

选项（b）：如果每个事件都存储数据量太大的话，可以选择存储事件的聚合结果。比如，如果要记录某个事件的发生次数，那么就可以在这个事件进来时将计数器加1。我们还可以将多个计数器保存在OLAP立方中。有了OLAP立方，当需要查找一个URL在某一天的访问量时，直接读取相应URL和日期组合的计数器就可以了。这样就只需要读取一个值，而不需要扫描一个很长的事件列表。

选项（a）的好处是，存储原始事件数据可以最大化分析的灵活性。比如，可以跟踪某个人以什么顺序访问了哪些页面，采用选项（b）就无法实现。这种分析对于一些离线处理任务非常重要，比如训练一个推荐系统。在这种应用场景下，最好是保存原始事件。

不过，选项（b）也有它的用途，尤其是需要实时决策或响应的时候。比如，为了防止别人破坏网站，可能需要引入一个访问频率限制，在一个小时内一个特定的IP只允许请求100次；如果客户端超出这个限制，就阻塞它。这时，通过原始数据存储实现效率将非常低下，因为系统需要不断地重新扫描事件历史才能确定某个人是否超出了限制。而针对每个IP每个时间窗口维护一个计数器将会更高效。总之，存储原始事件和存储聚合结果都是有用的，只不过应用场景不同。

对于选项（b），在最简单的情况下，可以让Web服务器直接更新聚合结果。这时，可以将计数器保存在像memcached或Redis这样具有原子增量操作的缓存中。每次Web服务器处理一个请求，就直接向缓存发送一条增量命令。更复杂一点，可以引入事件流（如图3），或者消息队列，或者事件日志。流上的事件与（图1）中PageViewEvent记录相同。

Reference:

* 分布式：http://www.infoq.com/cn/news/2015/11/Elastisys-cloud
* 数据一致性：http://blog.csdn.net/zheng0518/article/details/51194942
* 负载均衡：http://www.cnblogs.com/itfly8/p/5043435.html
* ID分配：http://chaubeau.github.io/blog/2014/06/10/mysql-shardingzai-gao-bing-fa-xia-de-quan-ju-idsheng-cheng/
* 通信可靠高效：http://www.cnblogs.com/ammy714926/p/4194073.html
* 消息队列：http://www.infoq.com/cn/articles/eventflow-app-scalability-reliability-maintainability