性能设计篇之"异步处理"

2018-05-15 陈皓

性能设计篇之"异步处理"
朗读人：柴巍　11′43″ | 5.37M

在弹力设计篇中我们讲过，异步通讯的设计模式有助于提高系统的稳定性和容错能力。其实，异步通讯在分布式系统中还可以增加整个系统的吞吐量，从而可以面对更高的并发，并可以更多地利用好现有的系统资源。为什么这么说呢？

我们试想一下，在你的工作中，有很多人会来找你，让你帮着做事。如果你是这种请求响应式的工作方式，那么本质上来说，你是在被动工作，也就是被别人驱动的工作方式。

当你在做一件事的时候，如果有别人来找你做其它事，你就会被打断而要去干别的事。另外，没办法把这些事统筹安排。如果可以统筹安排，本来五件事只需要 2 个小时，如果不能，你可能要做出 5 个小时来。异步处理任务可以让你更好地利用好时间和资源。利用好了时间和资源，性能自然就会提升上来。

这就好像邮递业务一样，你寄东西的时候，邮递公司会把大量的去往同一个方向的订单合并处理，并统一地调配物流交通工具，从而在整体上更为节省资源和时间。

在分布式架构中，我们的系统被拆成了很多的子系统。如果想把这堆系统合理地用好，并更快地处理大量的任务，我们就需要统一地规划和统筹整体，这样可以达到整体的最优。本质上，这和邮递公司处理邮件一样，是相同的道理。

在计算机的世界里，到处都是异步处理。比如：当程序读写文件时，我们的操作系统并不会真正同步地去操作硬盘，而是把硬盘读写请求先在内存中 hold 上一小会儿（几十毫秒），然后，对这些读写请求做 merge 和 sort。

也就是说，merge 是把相同的操作合并，相同的读操作只读一次，相同的写操作，只写最后一次，而 sort 是把不同的操作排个序，这样可以让硬盘向一个方向转一次就可以把所有的数据读出来，而不是来来回回地转。这样可以极大地提高硬盘的吞吐率。

再如，我们的 TCP 协议向网络发包的时候，会把我们要发的数据先在缓冲区中进行囤积，当囤积到一定尺寸时（MTU），才向网络发送，这样可以最大化利用我们的网络带宽。而传输速度和性能也会变得很快。

这就是异步系统所带来的好处——让我们的系统可以统一调度。

另外，我举上面这两个例子是想告诉你，我们可能会觉得异步通讯慢，其实并不然，我们同样也可以把异步做得比较实时。

多说一句，就算是有延时，异步处理在用户体验上也可以给用户带来一个不错的用户体验，那就是用户可以有机会反悔之前的操作。

异步处理的设计

之前，我们在弹力设计中讲的是异步通讯，这里，我们想讲的是异步任务处理。当然，这里面没有什么冲突的，只不过是，异步通讯讲的是怎么把系统连接起来，而我们这里想讲的是怎么处理任务。

首先，我们需要一个前台系统，把用户发来的请求一一记录下来。这有点像请求日志。这样，我们的操作在数据库或是存储上只会有追加的操作，性能会很高。我们收到请求后，给客户端返回 " 收到请求，正在处理中 "。

然后，我们有个任务处理系统来真正地处理收到的这些请求。为了解耦，我们需要一个任务派发器，这里就会出来两个事，一个是推模型 Push，一个是拉模型 Pull。

所谓 Push 推模型，就是把任务派发给相应的人去处理，有点像是一个工头的调度者的角色。而 Pull 拉模型，则是由处理的人来拉取任务处理。这两种模型各有各的好坏。一般来说，Push 模型可以做调度，但是它需要知道下游工作结点的情况。

除了要知道哪些是活着的，还要知道它们的忙闲程度。这样一来，当下游工作结点扩容缩容或是有故障需要维护等一些情况发生时，Push 结点都需要知道，这会增加一定的系统复杂度。而 Pull 的好处则是可以让上游结点不用关心下游结点的状态，只要自己忙得过来，就会来拿任务处理，这样可以减少一定的复杂度，但是少了整体任务调度。

一般来说，我们构建的都是推拉结合的系统，Push 端会做一定的任务调度，比如它可以像物流那样把相同商品的订单都合并起来，打成一个包，交给下游系统让其一次处理掉；也可以把同一个用户的订单中的不同商品给拆成多个订单。然后 Pull 端来订阅 Push 端发出来的异步消息，处理相应的任务。

事件溯源

在这里，我们需要提一下 Event Sourcing（事件溯源）这个设计模式。

所谓 Event Sourcing，其主要想解决的问题是，我们可以看到数据库中的一个数据的值（状态），但我们完全不知道这个值是怎么得出来的。就像银行的存折一样，我们可以在银行的存折看到我们收支的所有记录，也能看得到每一笔记录后的余额。

当然，如果我们有了所有的收支流水账的记录，我们完全不需要保存余额，因为我们只需要回放一下所有的收支事件，就可以得到最终的数据状态。这样一来，我们的系统就会变得非常简单，只需要追加不可修改的数据操作事件，而不是保存最终状态。除了可以提高性能和响应时间之外，还可以提供事务数据一致性，并保留了可以启用补偿操作的完整记录和历史记录。

还有一个好处，就是如果我们的代码里有了 bug，在记录状态的系统里，我们修改 bug 后还需要做数据修正。然而，在 Event Sourcing 的系统里，我们只需要把所有事件重新播放一遍就好了，因为整个系统没有状态了。

事件不可变，并且可使用只追加操作进行存储。 用户界面、工作流或启动事件的进程可继续，处理事件的任务可在后台异步运行。 此外，处理事务期间不存在争用，这两点可极大提高应用程序的性能和可伸缩性。

事件是描述已发生操作的简单对象以及描述事件代表的操作所需的相关数据。 事件不会直接更新数据存储，只会对事件进行记录，以便在合适的时间进行处理。 这可简化实施和管理。

事件溯源不需要直接更新数据存储中的对象，因而有助于防止并发更新造成冲突。

最重要的是，异步处理 + 事件溯源的方式，可以很好地让我们的整个系统进行任务的统筹安排、批量处理，可以让整体处理过程达到性能和资源的最大化利用。

关于 Event Sourcing 一般会和 CQRS 一起提。另外，你可以去 GitHub 上看看这个项目的示例以得到更多的信息。

异步处理的分布式事务

在前面的《分布式系统的本质》一文中，我们说过，对于分布式事务，在强一致性下，在业务层上只能做两阶段提交，而在数据层面上需要使用 Raft/Paxos 的算法。但是，我想说，在现实生活中，需要用到强一致性的场景实在不多，真是不是所有的场景都必须要强一致性的事务的。

我们仔细想想现实生活当中的很多例子。比如，我们去餐馆吃饭，先付钱，然后拿个小票去领吃的。这种情况下，把交钱和取货这两个动作分开，可以让我们的餐馆有更高的并发和接客能力。如果要做成两阶段提交，顾客锁定好钱，餐馆锁定好食材，最后一手交钱一手交饭，那么这是一件非常恐怖的事。

是的，你可以看到，我们的现实世界中有很多这样先付钱，拿小票去领货的场景，也有先消费，然后拿一个账单去付钱的场景。总之，完全不需要两阶段提交这种方式。我们完全可以使用异步的方式来达到一致性，当然，是最终一致性。

要达到最终一致性，我们需要有个交易凭证。也就是说，如果一个事务需要做 A 和 B 两件事，比如，把我的钱转给我的朋友，首先先做扣钱交易，然后，记录下扣钱的凭证，拿这个凭证去给我朋友的账号上加钱。

在达成这个事务的过程中，有几点需要注意。

- 凭证需要非常好地保存起来，不然会导致事务做不下去。
- 凭证处理的幂等性问题，不然在重试时就会出现多次交易的情况。
- 如果事务完成不了，需要做补偿事务处理。

异步处理的设计要点

异步处理中的事件驱动和事件溯源是两个比较关键的技术。

异步处理可能会因为一些故障导致我们的一些任务没有被处理，比如消息丢失，没有通知到，或通知到了，没有处理。有这一系列的问题，异步通知的方式需要任务处理方处理完成后，给任务发起方回传状态，这样确保不会有漏掉的。

另外，发起方也需要有个定时任务，把一些超时没有回传状态的任务再重新做一遍，你可以认为这是异步系统中的＂对账＂功能。当然，如果要重做的话，就需要处理方支持幂等性处理。

异步处理的整体业务事务问题，也就是说，异步处理在处理任务的时候，并不知道能否处理成功，于是其会一步一步地处理，如果到最后一步不能成功，那么你就需要回滚。这个时候，需要走我们在弹力设计中说的补偿事务的流程。

并不是所有的业务都可以用异步的方式，比如一些需要强一致性的业务，使用异步的方式可能就不适合，这里需要我们小心地分析业务。我相信绝大多数的业务场景都用不到强一致性，包括银行业务。另外，在需要性能的时候，需要牺牲强一致性，变为最终一致性。

在运维时，我们要监控任务队列里的任务积压情况。如果有任务积压了，要能做到快速地扩容。如果不能扩容，而且任务积压太多，可能会导致整个系统挂掉，那么就要开始对前端流量进行限流。

最后，还想强调一下，异步处理系统的本质是把被动的任务处理变成主动的任务处理，其本质是在对任务进行调度和统筹管理。

小结

好了，我们来总结一下今天分享的主要内容。首先，我介绍了异步通讯，它在弹力设计中的作用是提高系统的稳定性和容错能力，而其实我们还可以在异步通讯的基础上统筹任务来提高系统的吞吐量。接着，我讲了异步通讯的设计，包括推拉结合的模型。异步处理配合事件溯源一起使用，将大大简化 bug 修复后的数据恢复，也能用于实现存储的事务一致性。

我将餐馆吃饭作为比喻，介绍了异步处理的事务一致性一般不是强一致性，而是最终一致性，这样才能取得高的吞吐量。最后，我指出了异步处理的设计要点。下篇文章中，我们讲述数据库扩展。希望对你有帮助。

也欢迎你分享一下你的异步处理过程是怎样统筹安排来提高执行效率的？异步事务又是怎样实现的？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
  - 认识故障和弹力设计
  - 隔离设计 Bulkheads
  - 异步通讯设计 Asynchronous
  - 幂等性设计 Idempotency
  - 服务的状态 State
  - 补偿事务 Compensating Transaction
  - 重试设计 Retry
  - 熔断设计 Circuit Breaker
  - 限流设计 Throttle
  - 降级设计 degradation
  - 弹力设计总结
- 管理设计篇
  - 分布式锁 Distributed Lock
  - 配置中心 Configuration Management
  - 边车模式 Sidecar

- 服务网格 Service Mesh
- 网关模式 Gateway
- 部署升级策略

- 性能设计篇
  - 缓存 Cache
  - 异步处理 Asynchronous
  - 数据库扩展
  - 秒杀 Flash Sales
  - 边缘计算 Edge Computing

精选留言

浪荡居士 7

耗子哥的文章没得说...引经据典...但有一点小建议，耗子哥能否把相关的开源框架也引用一下？我们小公司的小弟见识小...不知道哪些

2018-05-15

天真有邪 3

很失望

2018-05-15

1angxi 0

阿里面试的时候特别喜欢问这类问题□

2018-06-03

kingeasternsun 0

皓哥文中讲到推拉结合的例子，是否可以这么理解，上游将数据push到下游，但是下游只在从上游pull到处理数据的命令时进行处理？

2018-05-30

圣诞使者 0

耗子哥讲的这个操作系统的merge和sort和linux系统编程中有出入，书中的意思是如果读请求的序号是2341，内核先sort成1234然后merge成一个请求，固态盘一般是noop算法，只merge不sort，上一个请求就会是两个读请求1和234。不知道我的理解对不对。

2018-05-24

颇忒妥 0

Event sourcing 需要snapshot 否则启动时每次回放日志太恐怖。
如果要自制分布式系统可以看一下atomix项目

2018-05-17

作者回复

是的，要做snapshot

2018-05-22

50infivedays 0

想了解下数据整形相关的内容

2018-05-16

曹铮 0

实践的例子，柔性事务的最大努力通知算么？

2018-05-15

K 0

请教一下老师，用事务性的MQ来做最终一致性，这是一个好的实践吗？

2018-05-15

作者回复

可以的，但具体要看业务场景了

2018-05-22

Xg huang 0

皓哥，想问下你们在实现event souring 模式的时候，用了什么中间件做存储？我最常接触的主要是rabit mq和kafka, 前者主要是push 模型我觉得不太适合做es,后者虽然是pull模型，但更多的是强调性能而不是数据可靠性，所以问下皓哥是怎样选型的，谢谢

2018-05-15