10 | 如何利用依赖管控来提升写服务的性能和可用性?

本模块的前几讲讨论了在存储上如何分库分表、如何构建无状态存储集群，来打造一个高可用的、支持海量数据存储的写业务的系统架构。

在写业务的系统架构里，除了需要关注存储上的高可用，写链路上的各项外部依赖的管控同样十分重要。因为即使存储的高可用做好了，也可能会因为外部依赖的不可用进而导致系统故障。比如写链路上依赖的某一个接口性能抖动或者接口故障，都会导致你的系统不可用。对于此问题，本讲将介绍一个提升写服务性能和可用性的升级架构方案，详细讲解如何对写链路依赖进行精细化管控。

链路依赖的全貌

完成一个写请求时，不仅需要依赖存储，大部分场景还需要依赖各种外部第三方提供的接口。比如：

1. 当你发布一条微博，在数据存储至数据库前，不仅需要依赖用户模块校验用户的有效性、还需要依赖安全过滤非法内容等；
2. 在创建订单时，同样是先要校验用户有效性、再校验用户的收货地址合法性，以及获取最新价格、扣减库存、扣减支付金额等。完成上述的校验和数据获取，最后一步才是写存储。

其他的写场景，比如发布短视频、发布博客等亦是如此。上述几种场景的架构如下图 1 所示：

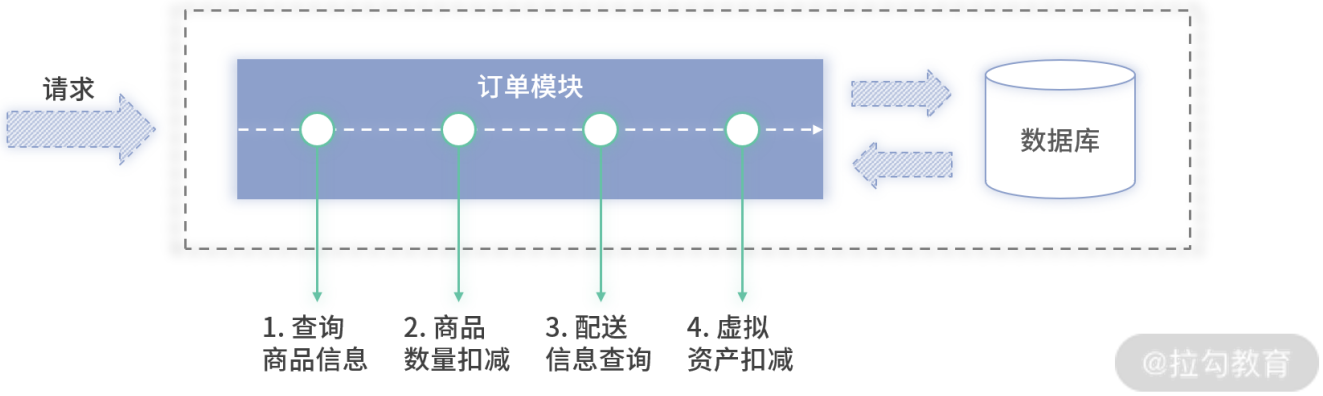

图 1：外部依赖的架构常见场景

对于整个链路依赖的各项外部接口，可能是出现了以下几个问题，导致系统不可用：

1. 外部接口性能抖动严重，比如从 50ms 飙升至 500 ms，进而导致你的接口超时，此时会影响你的系统可用率；
2. 完成上述某一个写业务时，如果需要依赖外部的接口过多，也会导致你的接口性能太差；
3. 外部接口可用率下降，也会影响你的系统的可用率。

依赖并行化

当依赖外部接口过多时，可以从几个方面进行优化，来提升整体的性能和写接口的稳定性。

将依赖的串行改并行

假设一次写请求要依赖二十个外部接口，可以将这些依赖全部并行化，优化的架构如下图 2 所示：

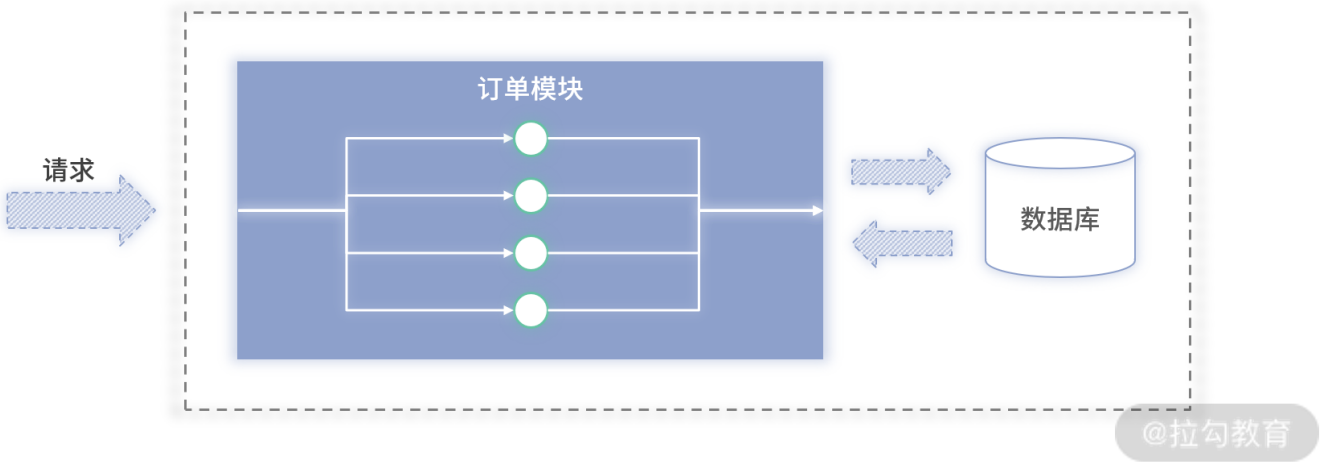

图 2：串行改并行的架构方案图

如果一个依赖接口的性能为 10ms，以串行执行的方式，请求完所有外部依赖就需要 200ms（10ms*20）。但改为并行执行后，只需要 10ms 即可完成。上述情况中，我们假设每个接口的性能都是 10ms，但在实际场景中并没有这么精确的数字，有的外部依赖可能快一点、有的可能慢一点。**实际并行执行的耗时，等于最慢的那个接口的性能**。

全部外部依赖的接口都可以并行是一种理想情况。**接口能否并行执行有一个前置条件，即两个接口间没有任何依赖关系**，如果 A 接口执行的前置条件是需要 B 接口返回的数据才能执行，那么这两个接口则不能并行执行。按相互依赖梳理后的并行执行方案如下图 3 所示。对于并行中存在相互依赖的场景，并行化后的性能等于最长子串（下图 3 中红色框）的性能总和。

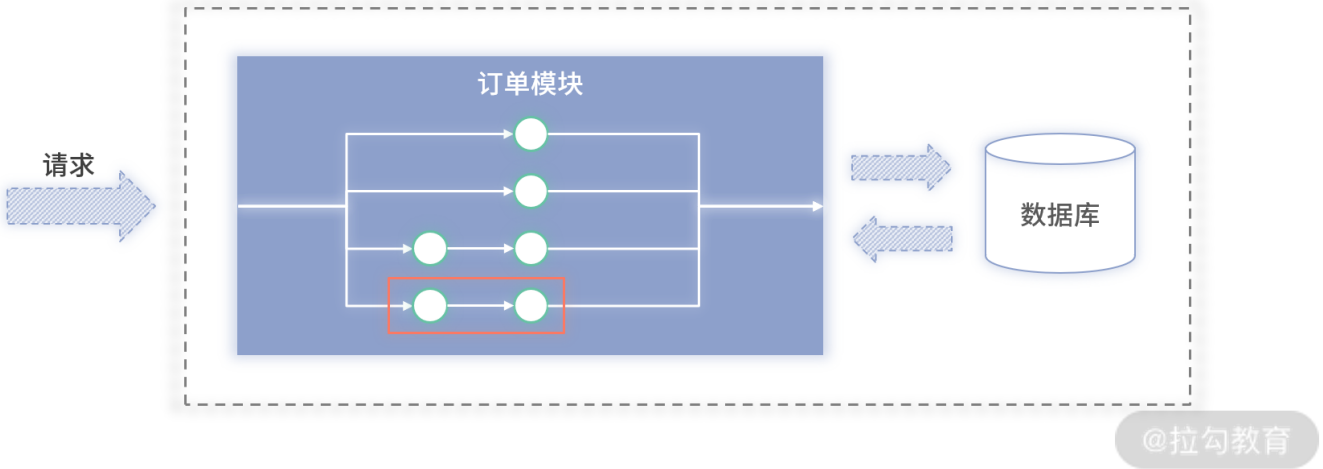

图 3：并行中需串行执行的架构方式

依赖后置化

此外，**虽然整个链路上会有较多外部接口，但大部分场景里，很多接口都是可以后置化的**。后置化是指当接口里的业务流程处理完成并返回给用户之后，后置去处理一些非重要且对实时性无要求的场景。

比如在提交订单后，用户只需要查看订单是否下单成功，以及对应的价格、商品和数量是否正确。而对于商品的详细描述信息、所归属的商家名称等信息并不会特别关心，如果在提单的同时还需要获取这些用户不太关心的信息，会给整个提单的性能和可用率带来非常大的影响。鉴于这种情况，可以在提单后异步补齐这些仅供展示的信息。

采用依赖后置化后，需要增加一个异步 Worker 进行数据补齐。架构如下图 4 所示：

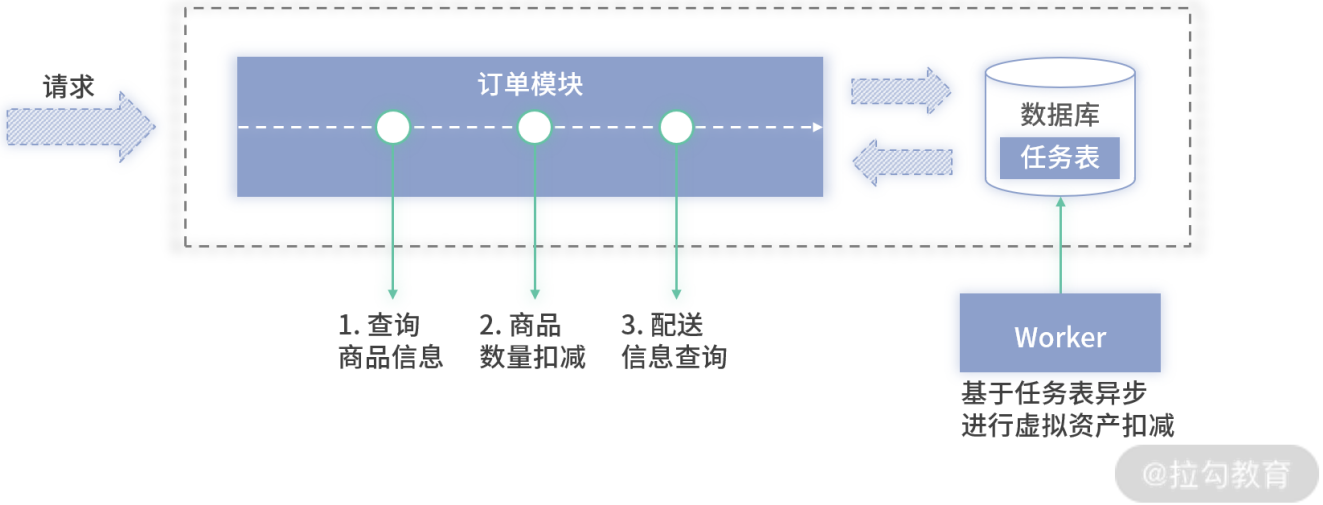

图 4：依赖后置化架构

对于一些可以后置补齐的数据，可以在写请求完成时将原始数据写入一张任务表。然后启动一个异步 Worker，异步 Worker 再调用后置化的接口去补齐数据，以及执行相应的后置流程（比如发送 MQ 等）。

通过依赖后置化移除一些不必要的接口调用，会提升你的写接口的整体性能和可用性。

显式设置超时和重试

即使是使用了后置化的方案，仍然会有一些外部接口需要同步调用。如果这些同步调用的接口出现性能抖动或者可用率下降，就需要通过显式设置超时和重试来规避上述问题。

超时设置

设置超时是为了防止依赖的外部接口性能突然变得太差，比如从几十毫秒飙升至十几秒及以上，进而导致你的请求被阻塞，此请求线程得不到释放，还会导致你的微服务的 RPC 线程池被占满。此时又会带来新的问题，进程的 RPC 线程池被占满之后，就无法再接受任何新的请求，你的系统基本上也就宕机了。导致上述问题的架构如下图 5 所示：

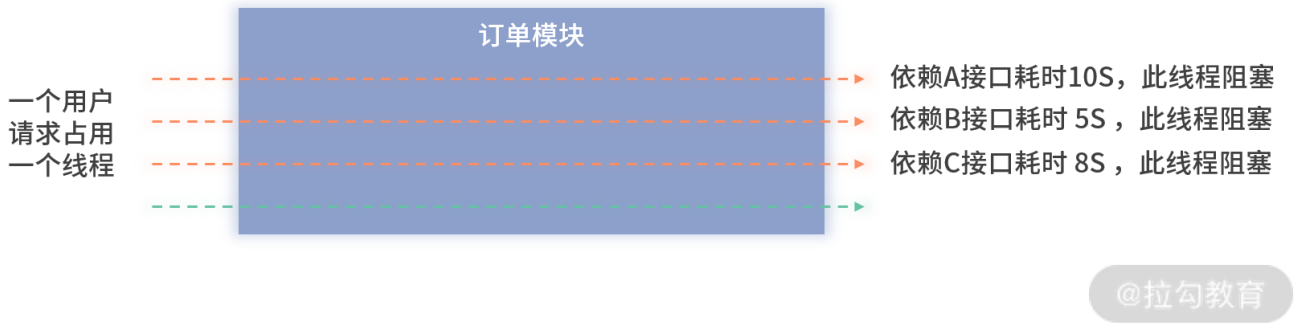

图 5：超时导致请求线程阻塞问题

在设置依赖的接口的超时阈值时，很多时候为了简便快速，大家都习惯设置一个不会太大，但下游接口实际执行时间远小于它的值，比如设置 3s 或者 5s。**我建议在设置此值时，通过系统上线后的性能监控图进行设置，设置超时时间等于 Max 的性能值，依据数据说话而不是“拍脑袋”做决定。**

如果你依赖的下游接口毛刺特别严重，表现就是它的接口性能的 Max 和 TP999，或与 TP99 相差特别大，比如 TP999 在 200ms 左右，但 Max 在 3~5s 左右，如下图 6 所展示的情况：

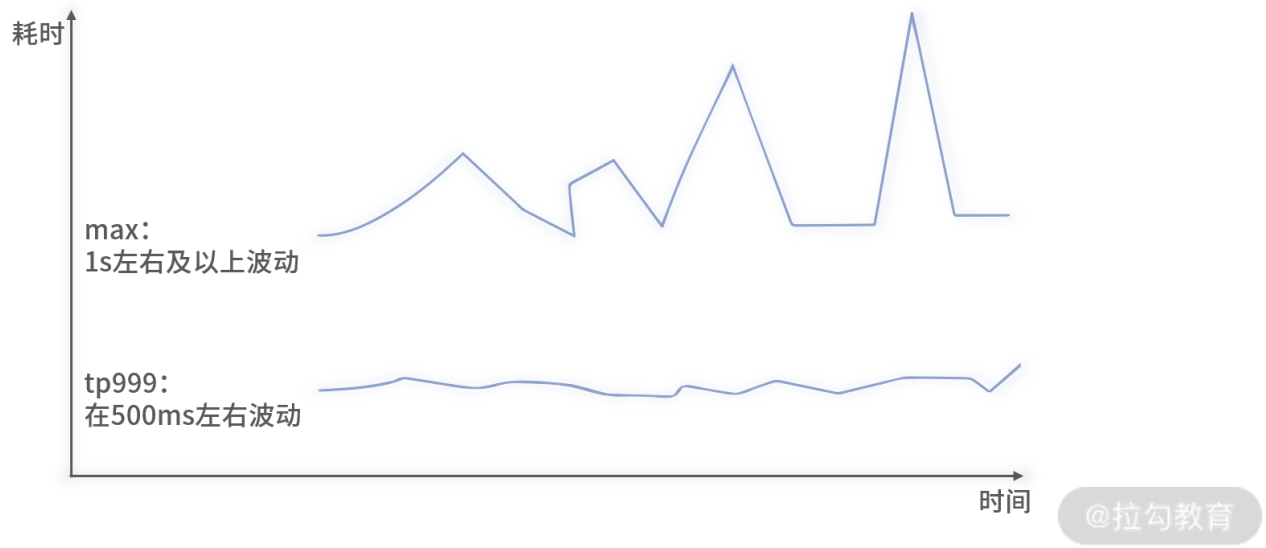

图 6：TP999 和 Max 差距太大图示

产生此现象的原因可能是网络环境不好，偶尔会抖动，导致 Max飙高。遇到此种情况，**为了防止接口因下游太高的 Max 导致线程阻塞，你可以将此接口的超时时间设置为 TP999 和 Max 之间的值**。但此时也会带来一个问题，就是超时时间控制在此区间值范围之后，TP999 之外的 0.1% 请求都会因为超时而失败，应对方案见下述“重试设置”小节。

重试设置

除了超时之外，还可以对依赖的读接口设置调用失败自动重试，重试次数设置为一次。

自动重试只能设置读接口，我在模块二里介绍过，读接口是无副作用的，重试对被依赖方无数据上的影响。而写接口是有状态的，如果你的依赖方没有做好幂等，设置自动重试可能会导致脏数据产生。

设置自动重试是为了提高接口的可用性。因为依赖的外部接口的某一台机器可能会因为网络波动、机器重启等导致当次调用超时进而失败。如果设置了自动重试，就可能重试到另外一台正常的机器，保障服务的可用性。

上一小节里提到为了保证接口性能，将超时时间设置为 TP999 和 Max 之间的值，但因此可能会带来 0.1% 的失败。如果搭配重试，可以将失败的比例降低到 0.0001%（即两次都失败，0.1%*0.1%）。即使使用了重试一次，你的接口性能也会较好。比如设置超时时间为大于上述 TP999 的值，比如 500ms，重试一次最大的耗时才为 1s，远比上述的 Max 低。

通过超时并灵活搭配重试，可以极大地提升接口的性能，但仍然存在非常低概率的失败（0.01%）。对于此问题，很多人的处理方式是简单粗暴地设置一个非常大的超时时间，这种做法并不能解决根本问题。我建议你去寻找导致毛刺的根源，比如：

1. 是否为某一台机器的网卡年老失修，丢包率高?
2. 缓存里是否存在数据量比较大的 Key，导致一请求就是几秒的耗时?
3. 是否调用不合法，每次请求获取上百条数据，网络消耗太大?

降级方式

现在业界有很多开源工具，比如 Hystrix 等，均可实现服务熔断和触发降级的功能。但此类技术框架并不提供业务如何降级，以及降级到哪里。比如你依赖的接口可用率下降了，Hystrix 可以设置可用率持续多久都低于具体某个阈值时，可以自动进行降级。但降级方案如何实现，是直接报错？还是调用替代接口？这个都需要你自己去考虑。

依赖系统故障时，有以下一些降级方式可供你选择。

1. 当依赖的是读接口，同时该接口返回的数据只用来补齐本次请求的数据时，可以对其返回的数据**采用前置缓存。** 当出现故障时，可以使用前置缓存顶一段时间，给依赖提供方提供一定的时间去修复缓存。
2. **对产生故障的依赖进行后置处理**。比如发布微博前需要判断是否为非法内容，可能要依赖风控的接口进行合规性判断。当风控接口故障后，可以直接降级，先将新微博数据写入存储并标记未校验。但此数据可能是不合规的，可以在业务上进行适当降

级，未校验的数据只允许用户自己看，待风控故障恢复后再进行数据校验，校验通过后再允许所有人可见。**通过有损+异步最终校验，也是一种常见的降级方案**。

3. 对于需要写下游的场景，比如提单时扣减库存，当库存不够便不能下单的场景，处理方式和上述第二点类似。当库存故障时，可降级直接跳过库存扣减，但需要提示用户后续可能无货。修复故障后进行异步校验库存，如果校验不通过，系统取消订单或发送消息通知客户进行人工判断是否要等待商家补货。此方式是一种预承接，但最终有可能失败的有损降级方案。

总结

本讲介绍了在完成一个写请求时，除了保障存储高可用之外，对于外部依赖，如何保障高可用，以及在出现故障时的可选降级措施。**当你在实现一个高可用写服务时，可以参考依赖并行、显式的设置超时和重试来保障性能和可用性**。

本讲介绍的内容不仅适用于写接口，对于读接口和扣减接口依然适用。只是大部分场景里，写接口的外部依赖较多且写接口担负一个公司的营收重任（外卖下单、购买电影票等），故将此讲内容放到此模块内。

最后，留一道讨论题给你，你使用过的降级方式和具体业务场景有哪些，欢迎写在留言区，我们一起讨论学习。

下一讲将介绍11 | 分库分表化后如何满足多维度查询?