



支付平台架构设计评审核心要点与最佳实践



(/gitchat/author/58ace254777c3f2d3b6e253f)

李艳鹏 (/gitchat/auth...

蚂蚁金服高级
技术专家，著

有《分布式服务架构：原理、设计与实战》和《可伸缩服务架构：框架与中间件》，曾经在易宝支付、花旗银行、甲骨文、新浪微博、路透社等大型IT互联网公司担任技术负责人和架构师的工作，现专注于区块链平台的研发与推广，擅长大规模高并发的线上与线下相结合的第三方支付平台的架构规划与实施。

[向作者提问 \(/m/mazi/author/58ace254777c3f2d3b6e253f/question\)](/m/mazi/author/58ace254777c3f2d3b6e253f/question)

查看本场Chat



(/gitchat/activity/5a7271dceb14c2636d904648)

互联网平台架构日益成为互联网发展的基石，对于 Java 开发者和架构师而言，只有在了解架构背后的原理后，才能写出更高质量的代码，才能设计出更好的方案，才能在错综复杂平台架构下产出价值，能在各种场景下快速发现问题、快速定位问题、快速解决问题。

本场 Chat 会带领大家从支付平台架构设计评审入手，讲解设计评审的核心要点，为读者带去现实中的案例，帮助读者理解设计评审的重要性、核心要点和最佳实现。在这场 Chat 中将学到如下内容：

1. 揭秘支付系统中数据库锁的应用实践。
2. 如何科学的设置线程池。
3. 缓存使用的最佳实践。
4. 数据库设计要点。
5. 一行代码引起的“血案”。
6. 幂等和防重。
7. 实现分布式任务调度的多种方法。

揭秘支付系统中数据库锁的应用实践

锁通常应用在多个线程对一个共享资源进行同时操作，用来保证操作的有序性和正确性的同步设施。⁽⁷⁾在笔者看来，锁的本质其实是排队，不同的锁排队的空间和时间不同而已，例如，Java 的 Synchronized 的锁是在应用处理业务逻辑的时候在对象头上进行排队，数据库的锁是在数据库上进行数据库操作的时候进行排队，而分布式锁是在处理业务逻辑的时候在一个公用的存储服务上排队。



乐观锁

乐观锁是基于一种具有“乐观”的思想，假设数据库操作的并发非常少，多数情况下是没有并发的，更新是按照顺序执行的，少有的一些并发通过版本控制来防止脏数据的产生。具体过程为，在操作数据库数据的时候，对数据不加显式的锁，而是通过对数据的版本或者时间戳的对比来保证操作的有序性和正确性。一般是在更新数据之前，先获取这条记录的版本或者时间戳，在更新数据的时候，对比记录的版本或者时间戳，如果版本或者时间戳一样，则继续更新，如果不一样，则停止更新数据记录，这说明数据已经被其他线程或者其他客户端更新过了。这时候需要获取最新版本的数

据，进行业务逻辑的操作，再次进行更新。

其伪代码如下。

```
int version = executeSql("select version from... where id = $i  
  
// process business logic  
  
boolean succ = executeSql("update ... where id = $id and versi  
  
if (!succ) {  
    // try again  
}
```

乐观锁在同一时刻，只有一个更新请求会成功，其他的更新请求会失败，因此，适用于并发不高的场景，通常是在传统的行业里应用在 ERP 系统，防止多个操作员并发修改同一份数据。在某些互联网公司里，使用乐观锁在失败的时候再尝试多次更新，导致并发量始终上不去，是一个反模式。而且这种模式是应用层实现的，阻止不了其他程序对数据库数据的直接更新。

悲观锁

悲观锁是基于一种具有“悲观”的思想，假设数据库操作的并发很多，多数情况下是有并发的，在更新数据之前对数据上锁，更新过程中防止任何其他的请求更新数据而产生脏数据，更新完成之后，再释放锁，这里的锁是数据库级别的锁。

通常使用数据库的 `for update` 语句来实现，代码如下。



```
executeSql("select ... where id = $id for update");

try {
    // process business logic

    commit();
} catch (Exception e) {
    rollback();
}
```

悲观锁是在数据库引擎层次实现的，它能够阻止所有的数据库操作。但是为了更新一条数据，需要提前对这条数据上锁，直到这条数据处理完成，事务提交，别的请求才能更新数据，因此，悲观锁的性能比较低下，但是由于它能够保证更新数据的强一致性，是最安全的处理数据库的方式，因此，有些账户、资金处理系统仍然使用这种方式，牺牲了性能，但是获得了安全，规避了资金风险。

行级锁

不是所有更新操作都要加显示锁的，数据库引擎本身有行级别的锁，本身在更新行数据的时候是有同步和互斥操作的，我们可以利用这个行级别的锁，控制锁的时间窗口最小，一次来保证高并发的场景下更新数据的有效性。

行级锁是数据库引擎中对记录更新的时候引擎本身上的锁，是数据库引擎的一部分，在数据库引擎更新一条数据的时候，本身就会对记录上锁，这时候即使有多个请求更新，也不会产生脏数据，行级锁的粒度非常细，上锁的时间窗口也最少，只有更新数据记录的那一刻，才会对记录上锁，因此，能大大减少数据库操作的冲突，发生锁冲突的概率最低，并发度也最高。

通常在扣减库存的场景下使用行级锁，这样可以通过数据库引擎本身对记录加锁的控制，保证数据库更新的安全性，并且通过 `where` 语句的条件，保证库存不会被减到0以下，也就是能够有效的控制超卖的场景，如下代码。

```
boolean result = executeSql("update ... set amount = amount -

if (result) {
    // process sucessful logic
} else {
    // process failure logic
}
```

另外一种场景是在状态转换的时候使用行级锁，例如交易引擎中，状态只能从 init 流转到 doing 状态，任何重复的从 init 到 doing 的流转，或者从 init 到 finished 等其他状态的流转都会失败，代码如下。



```
boolean result = executeSql("update ... set status = 'doing' v

if (result) {
    // process sucessful logic
} else {
    // process failure logic
}
```

行级锁的并发性较高，性能是最好的，适用于高并发下扣减库存和控制状态流转的方向的场景。

但是，有人说这种方法是不能保证幂等的，比如说，在扣减余额场景，多次提交可能会扣减多次，这确实是实际存在的，但是，我们是有应对方案的，我们可以记录扣减的历史，如果有非幂等的场景出现，通过记录的扣减历史来核对并矫正，这种方法也适用于账务历史等场景，代码如下。

```
boolean result = executeSql("update ... set amount = amount -

if (result) {
    int amount = executeSql("select amount ... where id = $id'

    executeSql("insert into hist (pre_amount, post_amount) val

    // process successful logic
} else {
    // process failure logic
}
```

在支付平台架构设计评审中，通常对交易和支付系统的流水表的状态流转的控制、对账户系统的状态控制，分账和退款余额的更新等，都推荐使用行级锁，而单独使用乐观锁和悲观锁是不推荐的。

如何科学的设置线程池

线上高并发的服务就像默默的屹立在大江大河旁边的大堤一样，随时准备着应对洪水带来了冲击，线上高并发服务的线程池导致的问题也颇多，例如：线程池涨满、CPU 利用率高、服务线程挂死等，这些都是因为线程池的使用不当，或者没有做好保护、

降级的工作而导致的。



当然，有些小伙伴是有保护线程池的想法的，但是，大家是不是有过这样的经验和印象，线程池的线程有时候设置多了性能低，设置少了还是性能低，到底应该怎么设置线程池呢？

在经历过这些年对小伙伴的设计评审，得知小伙伴们都是凭经验、凭直觉来设置线程池的线程数的，然后根据线上的情况调整数量多少，最后找到一个最合适的值，这是通过经验的，有时候管用，有时候不管用，有时候虽然管用但是牺牲了很大的代价才找到最佳的设置数量。

其实，线程池的设置是有据可依的，可以根据理论计算来设置的。

首先，我们看一下理想的情况，也就是所有要处理的任务都是计算任务，这时，线程数应该等于 CPU 核数，让每个 CPU 运行一个线程，不需要线程切换，效率是最高的，当然这是理想情况。

这种情况下，如果我们要达到某个数量的 QPS，我们使用如下的计算公式。

$$\text{设置的线程数} = \text{目标 QPS} / (1 / \text{任务实际处理时间})$$

举例说明，假设目标 QPS=100，任务实际处理时间 0.2s， $100 * 0.2 = 20$ 个线程，这里的20个线程必须对应物理的20个 CPU 核心，否则将不能达到预估的 QPS 指标。

但实际上我们的线上服务除了做内存计算，更多的是访问数据库、缓存和外部服务，大部分的时间都是在等待 IO 任务。

如果 IO 任务较多，我们使用阿姆达尔定律来计算。

$$\text{设置的线程数} = \text{CPU 核数} * (1 + \text{io/computing})$$

举例说明，假设4核 CPU，每个任务中的 IO 任务占总任务的80%， $4 * (1 + 4) = 20$ 个线程，这里的20个线程对应的是4核心的 CPU。

线程中除了线程数的设置，线程队列大小的设置也很重要，这也是可以通过理论计算得出，规则为按照目标响应时间计算队列大小。

$$\text{队列大小} = \text{线程数} * (\text{目标相应时间} / \text{任务实际处理时间})$$

举例说明，假设目标相应时间为0.4s，计算阻塞队列的长度为 $20 * (0.4 / 0.2) = 40$ 。



另外，在设置线程池数量的时候，我们有如下最佳实践。

1. 线程池的使用要考虑线程最大数量和最小数量。
2. 对于单部的服务，线程的最大数量应该等于线程的最小数量，而混布的服务，适当的拉开最大最小数量的差距，能够整体调整 CPU 内核的利用率。
3. 线程队列大小一定要设置有界队列，否则压力过大就会拖垮整个服务。
4. 必要时才使用线程池，须进行设计性能评估和压测。
5. 须考虑线程池的失败策略，失败后的补偿。
6. 后台批处理服务须与线上面向用户的服务进行分离。

缓存使用的最佳实践

笔者在做设计评审的过程中，总结了一些开发人员在设计缓存系统时的优秀实践。

最佳实践1

缓存系统主要消耗的是服务器的内存，因此，在使用缓存时必须先对应用需要缓存的数据大小进行评估，包括缓存的数据结构、缓存大小、缓存数量、缓存的失效时间，然后根据业务情况自行推算未来一定时间的容量的使用情况，根据容量评估的结果来申请和分配缓存资源，否则会造成资源浪费或者缓存空间不够。

最佳实践2

建议将使用缓存的业务进行分离，核心业务和非核心业务使用不同的缓存实例，从物理上进行隔离，如果有条件，则请对每个业务使用单独的实例或者集群，以减少应用之间互相影响的可能性。笔者经常听说有的公司应用了共享缓存，造成缓存数据被覆盖，以及缓存数据错乱的线上事故。

最佳实践3

根据缓存实例提供的内存大小推送应用需要使用的缓存实例数量，一般在公司里会成立一个缓存管理的运维团队，这个团队会将缓存资源虚拟成多个相同内存大小的缓存实例，例如，一个实例有 4GB 内存，在应用申请时可以按需申请足够的实例数量来使用，对这样的应用需要进行分片。这里需要注意，如果我们使用了 RDB 备份机制，每

个实例使用 4GB 内存，则我们的系统需要大于 8GB 内存，因为 RDB 备份时使用 copy-on-write 机制，需要 fork 出一个子进程，并且复制一份内存，因此需要双份的内存存储大小。



最佳实践4

缓存一般是用来加速数据库的读操作的，一般先访问缓存，后访问数据库，所以缓存的超时时间的设置是很重要的。笔者曾经在一家互联网公司遇到过由于运维操作失误导致缓存超时设置得较长，从而拖垮服务的线程池，最终导致服务雪崩的情况。

最佳实践5

所有的缓存实例都需要添加监控，这是非常重要的，我们需要对慢查询、大对象、内存使用情况做可靠的监控。

最佳实践6

如果多个业务共享一个缓存实例，当然我们不推荐这种情况，但是由于成本控制的原因，这种情况经常出现，我们需要通过规范来限制各个应用使用的 key 一定要有唯一的前缀，并进行隔离设计，避免缓存互相覆盖的问题产生。

最佳实践7

任何缓存的 key 都必须设定缓存失效时间，且失效时间不能集中在某一点，否则会导致缓存占满内存或者缓存穿透。

最佳实践8

低频访问的数据不要放在缓存中，如我们前面所说的，我们使用缓存的主要目的是提高读取性能，曾经有个小伙伴设计了一套定时的批处理系统，由于批处理系统需要对一个大的数据模型进行计算，所以该小伙伴把这个数据模型保存在每个节点的本地缓存中，并通过消息队列接收更新的消息来维护本地缓存中模型的实时性，但是这个模型每个月只用了一次，所以这样使用缓存是很浪费的，既然是批处理任务，就需要把任务进行分割，进行批量处理，采用分而治之、逐步计算的方法，得出最终的结果即可。

最佳实践9

缓存的数据不易过大，尤其是 Redis，因为 Redis 使用的是单线程模型，单个缓存 key 的数据过大时，会阻塞其他请求的处理。



最佳实践10

对于存储较多 value 的 key，尽量不要使用 HGETALL 等集合操作，该操作会造成请求阻塞，影响其他应用的访问。

最佳实践11

缓存一般用于交易系统中加速查询的场景，有大量的更新数据时，尤其是批量处理，请使用批量模式，但是这种场景较少。

最佳实践12

如果对性能的要求不是非常高，则尽量使用分布式缓存，而不要使用本地缓存，因为本地缓存在服务的各个节点之间复制，在某一时刻副本之间是不一致的，如果这个缓存代表的是开关，而且分布式系统中的请求有可能会重复，就会导致重复的请求走到两个节点，一个节点的开关是开，一个节点的开关是关，如果请求处理没有做到幂等，就会造成处理重复，在严重情况下会造成资金损失。

最佳实践13

写缓存时一定写入完全正确的数据，如果缓存数据的一部分有效，一部分无效，则宁可放弃缓存，也不要部分数据写入缓存，否则会造成空指针、程序异常等。

最佳实践14

在通常情况下，读的顺序是先缓存，后数据库；写的顺序是先数据库，后缓存。

最佳实践15

当使用本地缓存（如 Ehcache）时，一定要严格控制缓存对象的个数及生命周期。由于 JVM 的特性，过多的缓存对象会极大影响 JVM 的性能，甚至导致内存溢出等问题出现。

最佳实践16

在使用缓存时，一定要有降级处理，尤其是对关键的业务环节，缓存有问题或者失效时也要能回源到数据库进行处理。



关于缓存使用的最佳实践和线上案例，请参考《可伸缩服务架构：框架与中间件》一书的第4章的内容，预计在2018年3月份上市。

数据库设计要点

索引

提起数据库的设计要点，我们首先要说的就是数据库索引的使用，在线上的服务中，任何数据库的查询都要走索引，这个是底线，不能因为数据量暂时较小就不使用索引，久而久之可能数据量增大就导致了性能问题，一般每个开发者都有建立索引和使用索引的意识，然而，问题出现在开发者使用索引的方法上。我们要保证建立的索引的有效性，一定要确保线上的查询最后走到了索引，曾经就出现过这样的低级错误，某个场景需要根据 A、B、C 三个字段联合查询，开发者分别在 A、B 和 C 上建立了3个索引，看似也符合规范，但是实际上只用了 A 这个索引，B 和 C 的都没有用上，后来由于产生了性能问题，代码走查的时候才发现。

我们建议每个开发者对使用的 SQL 都要查看执行计划，另外，SQL 和索引要经过 DBA 的审阅才能上线。

另外，对于一般的数据库，>=、BETWEEN、IN、LIKE 等都可以走索引，而 NOT IN 不能走索引，如果匹配的字符以 % 开头，是不能走索引的，这些必须记住了。

范围查询

任何针对数据库的范围查询，都要有最大结果集条数的限制，然后进行分页处理，不能因为暂时数据量小而采用开发式的 SQL 语句，如果这样的话，在数据上量以后，会导致结果集太大，而让应用 OOM。

下面是主流数据库限制结果集大小的方法。

DB2

```
FETCH FIRST 100 ROWS ONLY
SELECT id FROM( SELECT ROW_NUMBER() OVER() AS num,id FROM TABL
```

MySQL



()

limit 1, 100



Oracle

`rownum`

Schema 变更

对于数据库的 Schema 变更，我们推荐只能增加字段，而不要修改字段，也不要删除字段，修改和删除字段的风险太高了，尤其是在应用比较复杂，数据库和应用的设计都是做加法加上来的，对于使用数据库的应用了解不清楚，不要轻易更改原有的数据结构，修改字段就有可能导致代码和数据库不兼容的情况。

即使是只允许添加字段，我们也做如下的规定。

新代码要兼容老数据，老代码要兼容新数据。

要尽量让新老代码和新老数据库 Schema 完全兼容，这在数据库升级前、中、后都不会产生问题。

字段枚举值的增加，或者数据库字段的含义、格式、限制的改变，必须考虑准生产和线上导致的不一致的行为或者上线过程中新老版本的不一致的行为。曾经就出现过，版本更新的时候增加了枚举值，由于 Boss 后台先上线，产生了新的枚举值，结果交易程序没有更新，不认识新的枚举值就出现了处理异常，因此枚举值要慎用。

事务

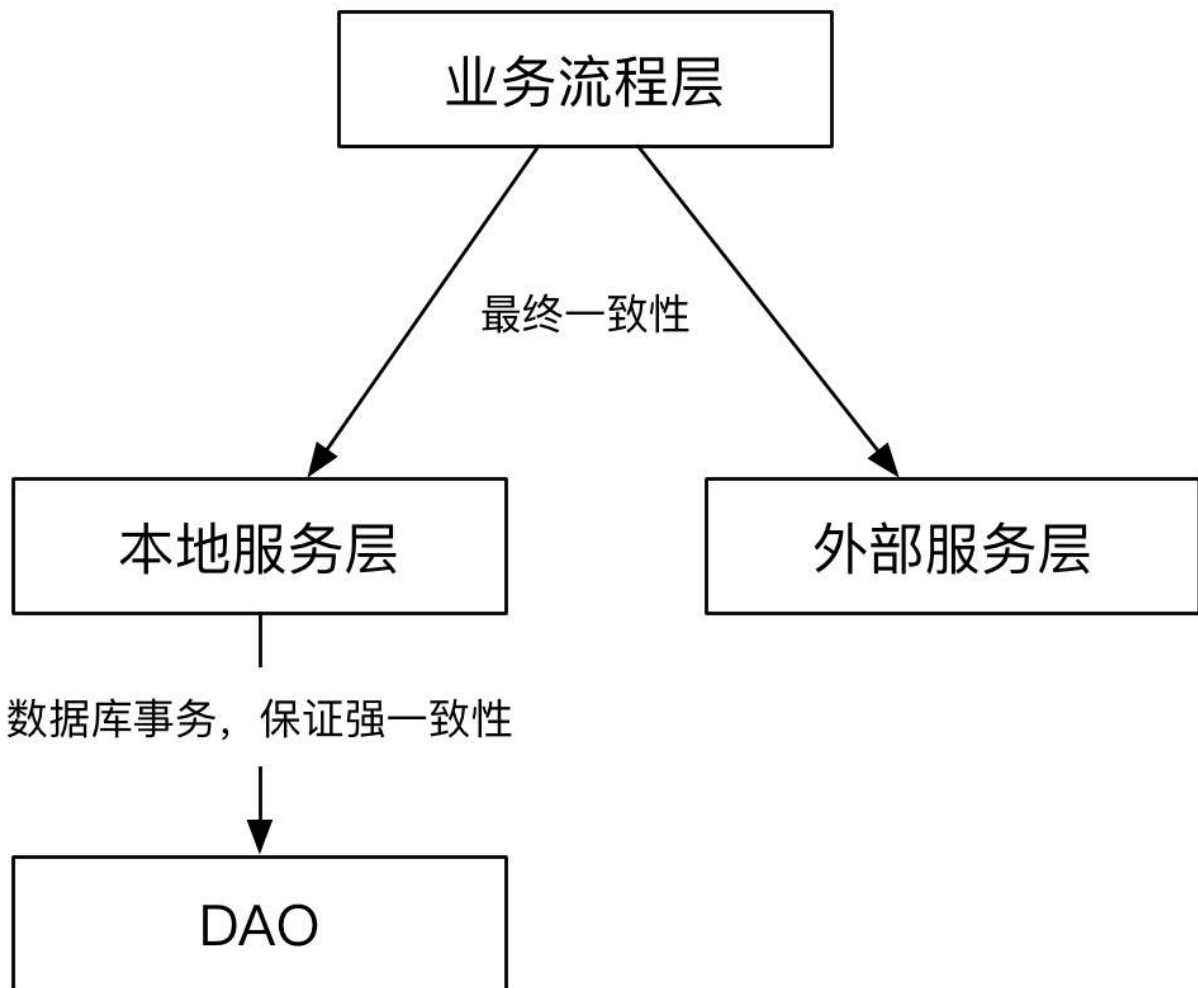
经常会出现数据库事务中调用远程服务，由于远程服务超时而拉长事务，导致数据库瘫痪的情况，因此，在事务处理过程中，禁止执行可能产生线程阻塞的调用，例如：锁等待、远程调用等。

另外，事务要尽可能保持短事务，一个事务中不要有太多的操作，或者做太多的事情，长时间操作事务会影响或堵塞其他的请求，累积可造成数据库故障，同一事务中大量的数据操作会引起锁的范围和影响扩大，易造成数据库的其他操作阻塞而导致短暂的不可用。

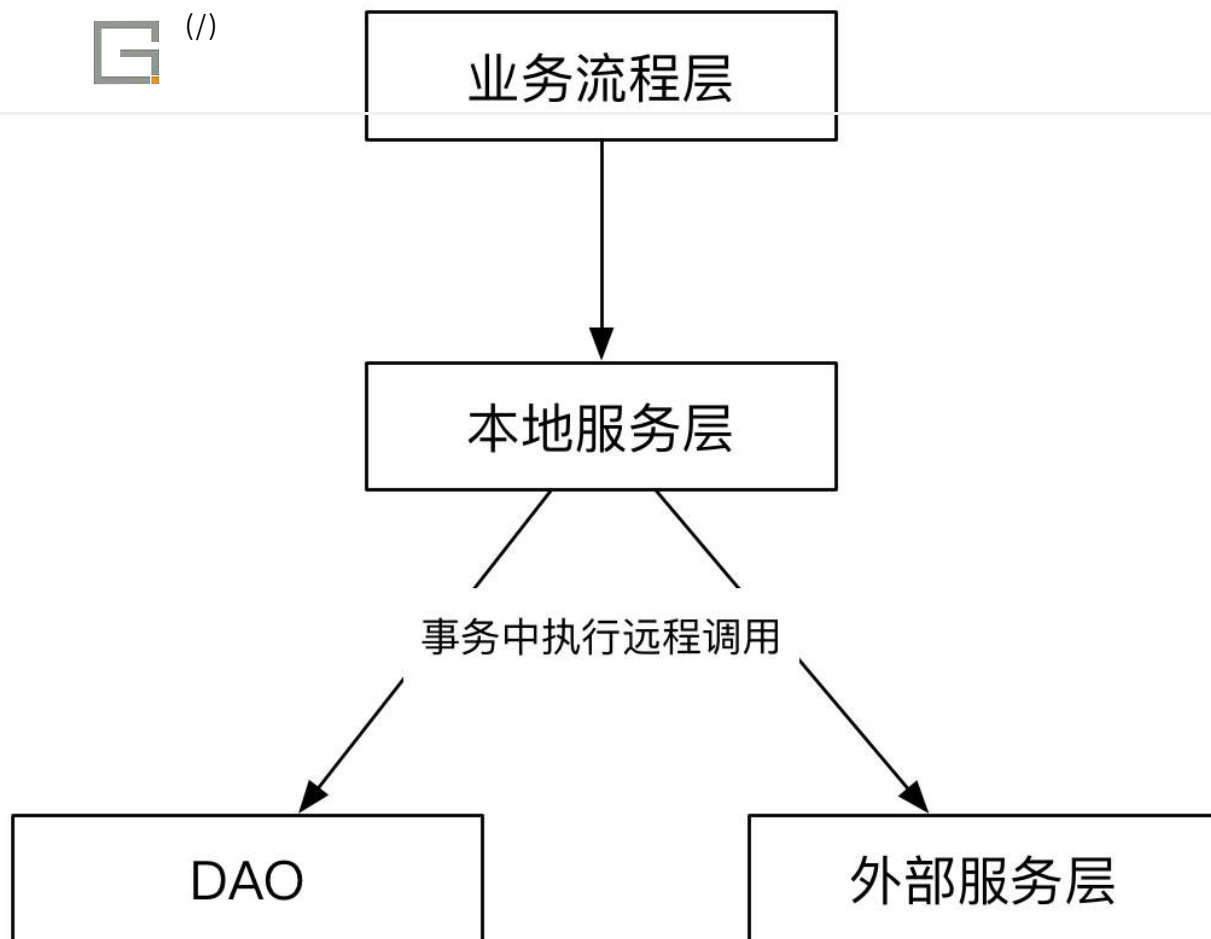
因此，如果业务允许，要尽可能用短事务来代替长事务，降低事务执行时间，减少锁的时长，使用最终一致性来保证数据的一致性原则。



我们推荐下图中的这种结构。



一定不能使用如下图中的这种结构。



SQL 安全

所有的 SQL 必须使用参数化的 SQL，防止 SQL 注入，这是一条不能妥协的底线原则。

一行代码引起的“血案”

在做支付平台的设计评审的时候，我们一定要格外仔细，因为一不注意可能会出现问題，甚至导致资金损失，笔者就经历一次增加一行打印日志的代码导致的“血案”。

在一次查问題的过程中，发现缺少一个日志，于是，增加了一行日志。

```
log.info(... + obj);
```

很不巧，上线以后应用就全面出现问題，交易出现失败，查看代码发现不时的有 `NullPointerException`，分析代码发现，出现 `NullPointerException` 的代码在 `obj.toString()` 方法里。

`object.toString()` 方法代码如下所示。

```
private Object fld1;
```

```
.....
```

```
public String toString() {  
    return ... + this.fld1;  
}
```



我们看见，在 `obj.toString()` 方法里面，直接使用了本地的变量 `fld1`，由于返回值是 `String` 类型，所以，Java 会试图将 `fld1` 转化成字符串，但是这个时候发生了 `NullPointerException`，那么，`fld1`就一定为 `null`，查明原因发现，这个对象是从缓存中反序列化而来的，反序列化的时候这个字段就为 `null`。

因此，我们看到线上的代码和环境是十分复杂的，在做设计评审的时候，一定要考虑到所有的情况，尽可能的将影响想得全面些，充分的降低代码变更带来的降低可用性的风险。

幂等和防重

幂等和防重虽然说起来挺复杂，但是实现起来很简单，这也就应了笔者的一句话：凡是能够有效解决问题的方法都是看起来很挫的方法”。

幂等是一个特性，一个操作执行多次，产生的结果是一样的，就成为幂等，用数学公式表达如下。

$$f(f(x)) = f(x)$$

对于某些业务具有的特点，操作本身就是幂等的，例如：删除一个资源、增加一个资源、获得一个资源等。

防重是实现幂等的一种方法，防重有多种方法。

1. 使用数据库表的唯一键进行滤重，拒绝重复的请求，这通常用在增加记录上，只要记录有唯一的主键，这种方法失踪奏效。
2. 使用状态流转的方向性来滤重，通常使用上面的行级锁来实现，这通常是在接受到回调消息的时候，要对记录的状态进行更新，可以使用行级锁来更新数据库的状态，然后根据更新的成功与否来判断继续处理的业务逻辑，例如，收到支付成

功消息，会先把支付记录从 init 更新成 pay_finished，如果有重复的请求，第二个更新的请求会失败。



3. 使用分布式存储对请求进行滤重，这个实现起来成本比较高。

实现分布式任务调度的多种方法

使用成熟的框架

可以使用成熟的开源分布式任务调用系统，例如 TBSchedule、ElasticJob 等等。

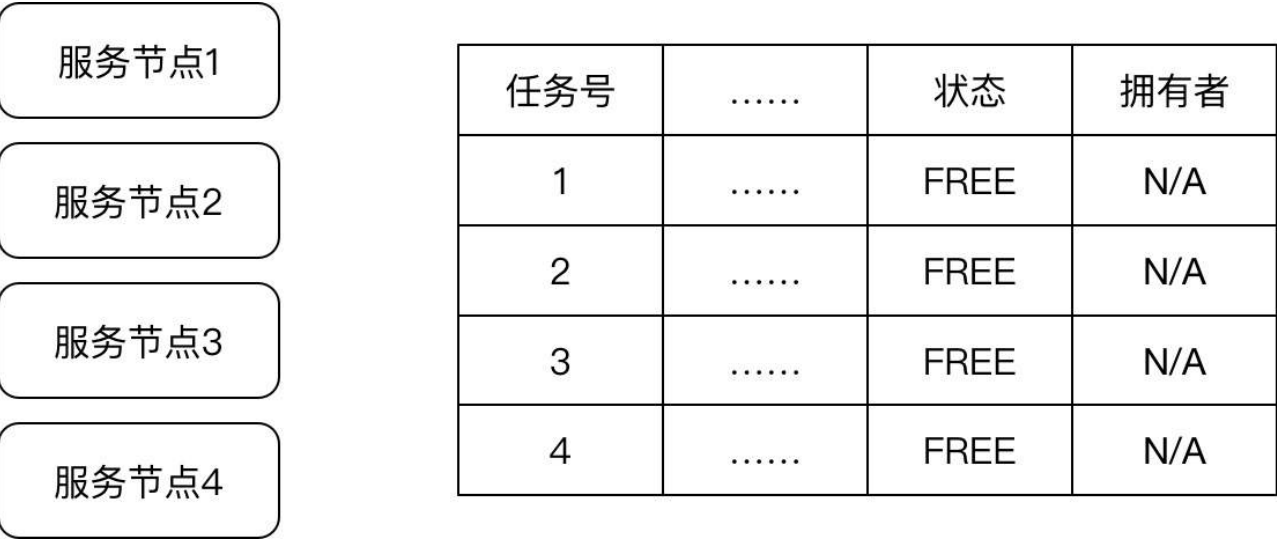
详细内容，请参考《可伸缩服务架构：框架与中间件》的第6章的内容。

代码自行实现

如果不喜欢使用成熟的框架，喜欢重复发明轮子，或者平台有要求，不准引入外部的开源项目，那么这个时候就是我们大显身手的时候了，我们可以自己开发一套分布式任务调度系统。

其实，分布式任务调度系统的核心就是任务的抢占，这和操作系统的任务调度类似，只不过应用的场景不同而已，操作系统处理各个应用进程提交的任务，而我们的分布式任务调度系统处理服务化系统中的后台定时任务。

假设，我们有4个后台定时的服务节点，以及4个任务存储在数据库的任务表中，如下图所示，所有的任务都处于空闲状态，拥有者为空，4台服务器都没有工作可做。



到了某个时间点，激活服务节点的定时任务，服务节点开始抢占任务，抢占任务需要更新数据库里面的记录状态字段和拥有者，一般会使用数据库的行级别锁，代码如下。



```
boolean result = executeSql("update ... set status = 'occupied'

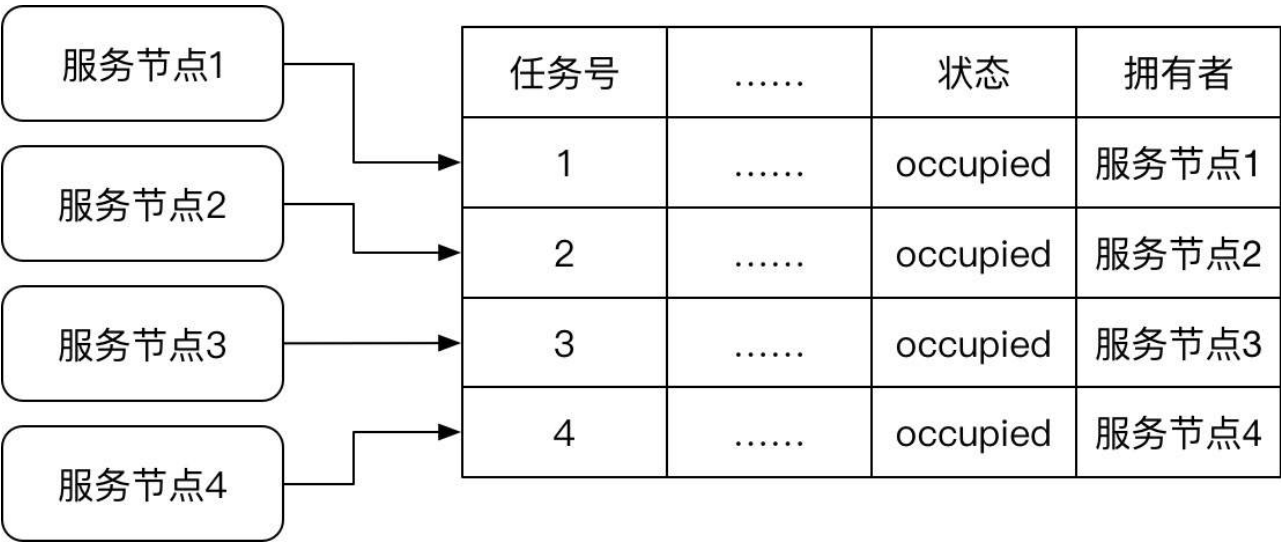
if (result) {
    Task t = executeSql("select ... where status = 'occupied'

    // process task t

    executeSql("update ... set status = 'finished' and owner =

}
```

假设服务节点1抢占了任务号1，服务节点2抢占了任务号2，服务节点3抢占了任务号3，服务节点4抢占了任务号4，如下图所示，这样各自开始处理自己的任务，处理后，将任务状态设置成 finished，其他服务节点就不会抢占这个任务了。



当然，这里描述的只是核心思想，具体实现的时候需要详细的设计，要考虑到任务如何调度、任务超时如何处理等等。

利用 Dubbo 服务化或者具有负载均衡的服务化平台来实现

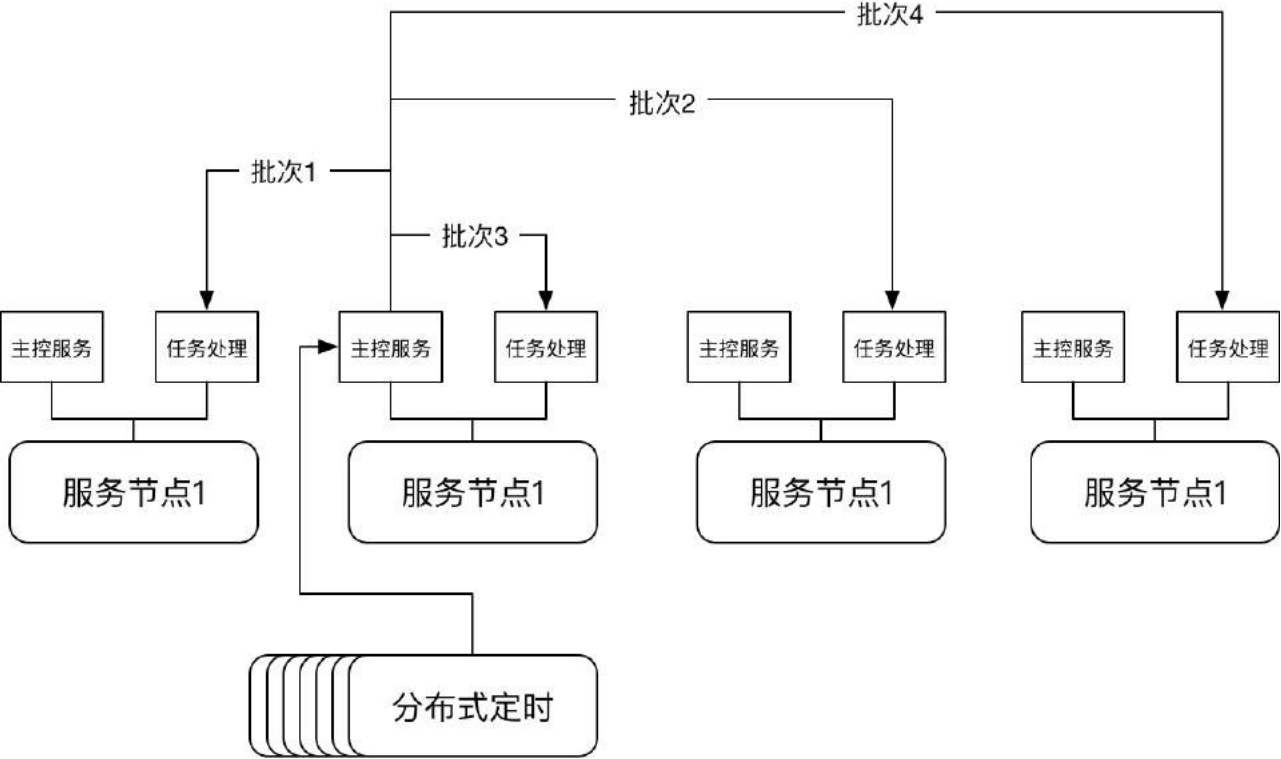
假如说平台规定不能使用第三方开源组件，自己开发又比较耗时耗力，那么还有一种办法，这种办法虽然看起来不是最佳的，但是能够帮助你快速实现任务的分片。

我们可以借助 Dubbo 服务化或者具有负载均衡的服务来实现，我们在服务节点上开发两个服务，一个总控服务，用来接受分布式定时的触发事件，总控服务从数据库里面捞取任务，然后分发任务，分发任务利用 Dubbo 服务化或者具有负载均衡的服务化平

台来实现，也就是调用服务节点的任务处理服务，通过服务化的负载均衡来实现。



例如，下图中分布式定时调用服务节点2的主控服务，主控服务从数据库里面捞取任务，并且分成4个分片，然后通过服务化调用任务处理接口，由于服务化具有负载均衡的功能，因此，4个分片会均衡的分布在服务节点1、服务节点2、服务节点3、服务节点4上。



当然，这种方法需要把后台的定时任务与前台的服务相互隔离，不能影响正常的线上服务是底线。

本文首发于GitChat，未经授权不得转载，转载需与GitChat联系。



16



0

写评论

向作者提问 (/m/mazi/author/58ace254777c3f2d3b6e253f/que



方维

希望能多些实际的案例，而非只是原则性的描述，有没架构评审文件可以放出来



29天前

0

0



白雪

希望能多些实际的案例，而非只是原则性的描述，有没架构评审文件可以放出来 +1 github上可以放代码

29天前

0

0



小白

有没有混合支付相关的设计供参考

28天前

0

0



陌

希望能多些实际的案例，而非只是原则性的描述，有没架构评审文件可以放出来

27天前

0

0



Shaw

"笔者曾经在一家互联网公司遇到过由于运维操作失误导致缓存超时设置得较长，从而拖垮服务的线程池，最终导致服务雪崩的情况。" PS：这里是否描述有误，应该是缓存超时设置得较短才会有这个问题吧？

27天前

0

0



尽情折叠我吧

层次太高，有些部分能看懂，但是理解不到你想表达的意思。有没有具体分析的案例拿出来把你讲的套一套，比如防重设计。

19天前

0

0