架构笔记

===方法论===

# 单点系统架构的可用性与性能优化

## 典型互联网高可用架构：

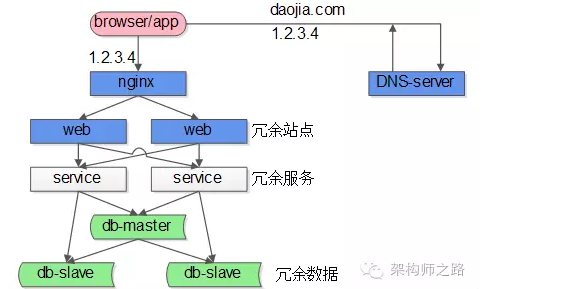
（1）客户端层，这一层是浏览器或者APP，第一步先访问DNS-server，由域名拿到nginx的外网IP

（2）负载均衡层，nginx是整个服务端的入口，负责反向代理与负载均衡工作

（3）站点层，web-server层，典型的是tomcat或者apache

（4）服务层，service层，典型的是dubbo或者thrift等提供RPC调用的后端服务

（5）数据层，包含cache和db，典型的是主从复制读写分离的db架构



## 单点架构存在的问题

单点系统一般来说存在两个很大的问题：

（1）非高可用：既然是单点，master一旦发生故障，服务就会受到影响

（2）性能瓶颈：既然是单点，不具备良好的扩展性，服务性能总有一个上限，这个单点的性能上限往往就是整个系统的性能上限

## shadow-master解决单点高可用问题

shadow-master是一种很常见的解决单点高可用问题的技术方案。

“影子master”，顾名思义，服务正常时，它只是单点master的一个影子，在master出现故障时，shadow-master会自动变成master，继续提供服务。

shadow-master它能够解决高可用的问题，并且故障的转移是自动的，不需要人工介入，但不足是它使服务资源的利用率降为了50%，业内经常使用keepalived+vip的方式实现这类单点的高可用。

## 减少与单点的交互，是存在单点的系统优化的核心方向

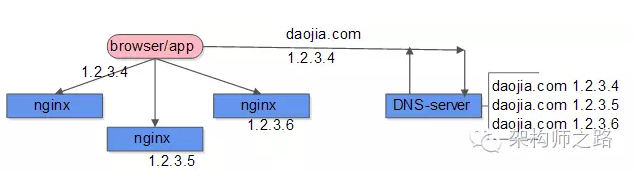
#### 批量写

批量写是一种常见的提升单点性能的方式。

客户端缓存也是一种降低与单点交互次数，提升系统整体性能的方法。

## 水平扩展是提升单点系统性能的好方案

第一步的DNS解析，只能返回一个nginx外网IP么？答案显然是否定的，“DNS轮询”技术支持DNS-server返回不同的nginx外网IP，这样就能实现nginx负载均衡层的水平扩展。

 DNS-server部分，一个域名可以配置多个IP，每次DNS解析请求，轮询返回不同的IP，就能实现nginx的水平扩展，扩充负载均衡层的整体性能。

数据库单点写库也是同样的道理，在数据量很大的情况下，可以通过水平拆分，来提升写入性能。

遗憾的是，并不是所有的业务场景都可以水平拆分，例如秒杀业务，商品的条数可能不多，数据库的数据量不大，就不能通过水平拆分来提升秒杀系统的整体写性能。

# 秒杀系统架构优化思路

**读写冲突，锁非常严重，这是秒杀业务难的地方！**

## 优化方向

### 将请求尽量拦截在系统上游

不要让锁冲突落到数据库上去）。传统秒杀系统之所以挂，请求都压倒了后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，流量虽大，下单成功的有效流量甚小。以12306为例，一趟火车其实只有2000张票，200w个人来买，基本没有人能买成功，请求有效率为0。

### 充分利用缓存

秒杀买票，这是一个典型的读多些少的应用场景，大部分请求是车次查询，票查询，下单和支付才是写请求。一趟火车其实只有2000张票，200w个人来买，最多2000个人下单成功，其他人都是查询库存，写比例只有0.1%，读比例占99.9%，非常适合使用缓存来优化。

## ****常见秒杀架构****

数据

服务

站点

浏览器

## 各层次优化细节

### 第一层，客户端怎么优化（浏览器层，APP层）

（a）产品层面，用户点击“查询”或者“购票”后，按钮置灰，禁止用户重复提交请求；

（b）JS层面，限制用户在x秒之内只能提交一次请求；

APP层面，可以做类似的事情，虽然你疯狂的在摇微信，其实x秒才向后端发起一次请求。这就是所谓的“将请求尽量拦截在系统上游”，越上游越好，浏览器层，APP层就给拦住，这样就能挡住80%+的请求，这种办法只能拦住普通用户（但99%的用户是普通用户）对于群内的高端程序员是拦不住的。firebug一抓包，http长啥样都知道，js是万万拦不住程序员写for循环，调用http接口的，这部分请求怎么处理？

### 第二层，站点层面的请求拦截

怎么拦截？怎么防止程序员写for循环调用，有去重依据么？ip？cookie-id？…想复杂了，这类业务都需要登录，用uid即可。在站点层面，对uid进行请求计数和去重，甚至不需要统一存储计数，直接站点层内存存储（这样计数会不准，但最简单）。一个uid，5秒只准透过1个请求，这样又能拦住99%的for循环请求。

5s只透过一个请求，其余的请求怎么办？缓存，页面缓存，同一个uid，限制访问频度，做页面缓存，x秒内到达站点层的请求，均返回同一页面。同一个item的查询，例如车次，做页面缓存，x秒内到达站点层的请求，均返回同一页面。如此限流，既能保证用户有良好的用户体验（没有返回404）又能保证系统的健壮性（利用页面缓存，把请求拦截在站点层了）。

页面缓存不一定要保证所有站点返回一致的页面，直接放在每个站点的内存也是可以的。优点是简单，坏处是http请求落到不同的站点，返回的车票数据可能不一样，这是站点层的请求拦截与缓存优化。

好，这个方式拦住了写for循环发http请求的程序员，有些高端程序员（黑客）控制了10w个肉鸡，手里有10w个uid，同时发请求（先不考虑实名制的问题，小米抢手机不需要实名制），这下怎么办，站点层按照uid限流拦不住了

### 第三层 服务层来拦截（反正就是不要让请求落到数据库上去）

服务层怎么拦截？大哥，我是服务层，我清楚的知道小米只有1万部手机，我清楚的知道一列火车只有2000张车票，我透10w个请求去数据库有什么意义呢？没错，请求队列！

对于写请求，做请求队列，每次只透有限的写请求去数据层（下订单，支付这样的写业务）

1w部手机，只透1w个下单请求去db

3k张火车票，只透3k个下单请求去db

如果均成功再放下一批，如果库存不够则队列里的写请求全部返回“已售完”。

对于读请求，怎么优化？cache抗，不管是memcached还是redis，单机抗个每秒10w应该都是没什么问题的。如此限流，只有非常少的写请求，和非常少的读缓存mis的请求会透到数据层去，又有99.9%的请求被拦住了。

当然，还有业务规则上的一些优化。回想12306所做的，分时分段售票，原来统一10点卖票，现在8点，8点半，9点，...每隔半个小时放出一批：将流量摊匀。

其次，数据粒度的优化：你去购票，对于余票查询这个业务，票剩了58张，还是26张，你真的关注么，其实我们只关心有票和无票？流量大的时候，做一个粗粒度的“有票”“无票”缓存即可。

第三，一些业务逻辑的异步：例如下单业务与 支付业务的分离。

### 最后是数据库层

浏览器拦截了80%，站点层拦截了99.9%并做了页面缓存，服务层又做了写请求队列与数据缓存，每次透到数据库层的请求都是可控的。db基本就没什么压力了，闲庭信步，单机也能扛得住，还是那句话，库存是有限的，小米的产能有限，透这么多请求来数据库没有意义。

全部透到数据库，100w个下单，0个成功，请求有效率0%。透3k个到数据，全部成功，请求有效率100%。

## 总结

（1）**尽量将请求拦截在系统上游**（越上游越好）；

（2）**读多写少的常用多使用缓存**（缓存抗读压力）；

浏览器和APP：做限速

站点层：按照uid做限速，做页面缓存

服务层：按照业务做写请求队列控制流量，做数据缓存

数据层：闲庭信步

并且：结合业务做优化

## Q&A

### 问题1

按你的架构，其实压力最大的反而是站点层，假设真实有效的请求数有1000万，不太可能限制请求连接数吧，那么这部分的压力怎么处理？

答：每秒钟的并发可能没有1kw，假设有1kw，解决方案2个：

（1）站点层是可以通过加机器扩容的，最不济1k台机器来呗。

（2）如果机器不够，抛弃请求，抛弃50%（50%直接返回稍后再试），原则是要保护系统，不能让所有用户都失败。

### 问题2

“控制了10w个肉鸡，手里有10w个uid，同时发请求” 这个问题怎么解决哈？

答：上面说了，服务层写请求队列控制

### 问题3

限制访问频次的缓存，是否也可以用于搜索？例如A用户搜索了“手机”，B用户搜索“手机”，优先使用A搜索后生成的缓存页面？

答：这个是可以的，这个方法也经常用在“动态”运营活动页，例如短时间推送4kw用户app-push运营活动，做页面缓存。

### 问题4

如果队列处理失败，如何处理？肉鸡把队列被撑爆了怎么办？

答：处理失败返回下单失败，让用户再试。队列成本很低，爆了很难吧。最坏的情况下，缓存了若干请求之后，后续请求都直接返回“无票”（队列里已经有100w请求了，都等着，再接受请求也没有意义了）

### 问题5

站点层过滤的话，是把uid请求数单独保存到各个站点的内存中么？如果是这样的话，怎么处理多台服务器集群经过负载均衡器将相同用户的响应分布到不同服务器的情况呢？还是说将站点层的过滤放到负载均衡前？

答：可以放在内存，这样的话看似一台服务器限制了5s一个请求，全局来说（假设有10台机器），其实是限制了5s 10个请求，解决办法：

1）加大限制（这是建议的方案，最简单）

2）在nginx层做7层均衡，让一个uid的请求尽量落到同一个机器上

### 问题6

服务层过滤的话，队列是服务层统一的一个队列？还是每个提供服务的服务器各一个队列？如果是统一的一个队列的话，需不需要在各个服务器提交的请求入队列前进行锁控制？

答：可以不用统一一个队列，这样的话每个服务透过更少量的请求（总票数/服务个数），这样简单。统一一个队列又复杂了。

### 问题7

秒杀之后的支付完成，以及未支付取消占位，如何对剩余库存做及时的控制更新？

答：数据库里一个状态，未支付。如果超过时间，例如45分钟，库存会重新会恢复（大家熟知的“回仓”），给我们抢票的启示是，开动秒杀后，45分钟之后再试试看，说不定又有票哟~

### 问题8

不同的用户浏览同一个商品 落在不同的缓存实例显示的库存完全不一样 请问老师怎么做缓存数据一致或者是允许脏读？

答：目前的架构设计，请求落到不同的站点上，数据可能不一致（页面缓存不一样），这个业务场景能接受。但数据库层面真实数据是没问题的。

### 问题9

就算处于业务把优化考虑“3k张火车票，只透3k个下单请求去db”那这3K个订单就不会发生拥堵了吗？

答：（1）数据库抗3k个写请求还是ok的；（2）可以数据拆分；（3）如果3k扛不住，服务层可以控制透过去的并发数量，根据压测情况来吧，3k只是举例；

### 问题10

如果在站点层或者服务层处理后台失败的话，需不需要考虑对这批处理失败的请求做重放？还是就直接丢弃？

答：别重放了，返回用户查询失败或者下单失败吧，架构设计原则之一是“fail fast”。

### 问题11

对于大型系统的秒杀，比如12306，同时进行的秒杀活动很多，如何分流？

答：垂直拆分

### 问题12

额外又想到一个问题。这套流程做成同步还是异步的？如果是同步的话，应该还存在会有响应反馈慢的情况。但如果是异步的话，如何控制能够将响应结果返回正确的请求方？

答：用户层面肯定是同步的（用户的http请求是夯住的），服务层面可以同步可以异步。

### 问题13

秒杀群提问：减库存是在那个阶段减呢？如果是下单锁库存的话，大量恶意用户下单锁库存而不支付如何处理呢？

答：数据库层面写请求量很低，还好，下单不支付，等时间过完再“回仓”，之前提过了。

===数据库与缓存===

# 缓存架构设计细节二三事

本文主要讨论这么几个问题：

（1）“缓存与数据库”需求缘起

（2）“淘汰缓存”还是“更新缓存”

（3）缓存和数据库的操作时序

（4）缓存和数据库架构简析

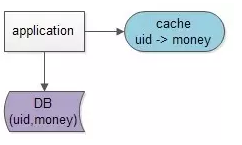
## 需求缘起

缓存是一种提高系统读性能的常见技术，对于读多写少的应用场景，我们经常使用缓存来进行优化。

例如对于用户的余额信息表account(uid, money)，业务上的需求是：

（1）查询用户的余额，SELECT money FROM account WHERE uid=XXX，占99%的请求

（2）更改用户余额，UPDATE account SET money=XXX WHERE uid=XXX，占1%的请求

由于大部分的请求是查询，我们在缓存中建立uid到money的键值对，能够极大降低数据库的压力。

## 读操作流程

有了数据库和缓存两个地方存放数据之后（uid->money），每当需要读取相关数据时（money），操作流程一般是这样的：

（1）读取缓存中是否有相关数据，uid->money

（2）如果缓存中有相关数据money，则返回【这就是所谓的数据命中“hit”】

（3）如果缓存中没有相关数据money，则从数据库读取相关数据money【这就是所谓的数据未命中“miss”】，放入缓存中uid->money，再返回

**缓存的命中率** = 命中缓存请求个数/总缓存访问请求个数 = hit/(hit+miss)

上面举例的余额场景，99%的读，1%的写，这个缓存的命中率是非常高的，会在95%以上。

**那么问题来了**

当数据money发生变化的时候：

（1）是更新缓存中的数据，还是淘汰缓存中的数据呢？

（2）是先操纵数据库中的数据再操纵缓存中的数据，还是先操纵缓存中的数据再操纵数据库中的数据呢？

（3）缓存与数据库的操作，在架构上是否有优化的空间呢？

这是本文关注的三个核心问题。

## 更新缓存 VS 淘汰缓存

什么是更新缓存：数据不但写入数据库，还会写入缓存

什么是淘汰缓存：数据只会写入数据库，不会写入缓存，只会把数据淘汰掉

更新缓存的优点：缓存不会增加一次miss，命中率高

淘汰缓存的优点：简单

那到底是选择更新缓存还是淘汰缓存呢，主要取决于“更新缓存的复杂度”。

例如，上述场景，只是简单的把余额money设置成一个值，那么：

（1）淘汰缓存的操作为deleteCache(uid)

（2）更新缓存的操作为setCache(uid, money)

更新缓存的代价很小，此时我们应该更倾向于更新缓存，以保证更高的缓存命中率

如果余额是通过很复杂的数据计算得出来的，例如业务上除了账户表account，还有商品表product，折扣表discount account(uid, money)

product(pid, type, price, pinfo) discount(type, zhekou)

业务场景是用户买了一个商品product，这个商品的价格是price，这个商品从属于type类商品，type类商品在做促销活动要打折扣zhekou，购买了商品过后，这个余额的计算就复杂了，需要：

（1）先把商品的品类，价格取出来：SELECT type, price FROM product WHERE pid=XXX

（2）再把这个品类的折扣取出来：SELECT zhekou FROM discount WHERE type=XXX

（3）再把原有余额从缓存中查询出来money = getCache(uid)

（4）再把新的余额写入到缓存中去setCache(uid, money-price\*zhekou)

更新缓存的代价很大，此时我们应该更倾向于淘汰缓存。

however，淘汰缓存操作简单，并且带来的副作用只是增加了一次cache miss，建议作为通用的处理方式。

## 先操作数据库 vs 先操作缓存

当写操作发生时，假设淘汰缓存作为对缓存通用的处理方式，又面临两种抉择：

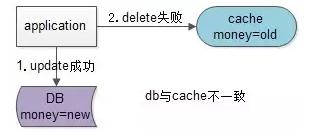
（1）先写数据库，再淘汰缓存 （2）先淘汰缓存，再写数据库

究竟采用哪种时序呢？

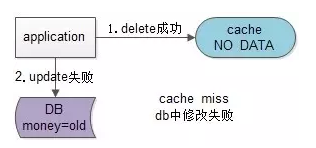
对于一个**不能保证事务性**的操作，一定涉及“哪个任务先做，哪个任务后做”的问题，解决这个问题的方向是：

如果出现不一致，谁先做对业务的影响较小，就谁先执行。

由于写数据库与淘汰缓存不能保证原子性，谁先谁后同样要遵循上述原则。



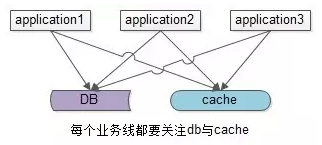
假设先写数据库，再淘汰缓存：第一步写数据库操作成功，第二步淘汰缓存失败，则会出现DB中是新数据，Cache中是旧数据，数据不一致。



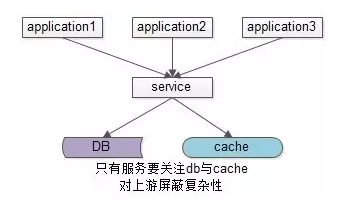
假设先淘汰缓存，再写数据库：第一步淘汰缓存成功，第二步写数据库失败，则只会引发一次Cache miss。

结论：数据和缓存的操作时序，结论是清楚的：先淘汰缓存，再写数据库。

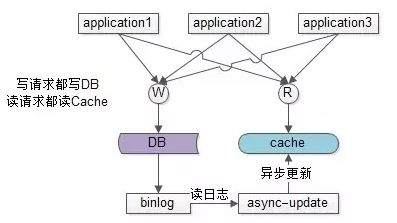
## 缓存架构优化



上述缓存架构有一个缺点：业务方需要同时关注缓存与DB，有没有进一步的优化空间呢？有两种常见的方案，一种主流方案，一种非主流方案



主流优化方案是服务化：加入一个服务层，向上游提供帅气的数据访问接口，向上游屏蔽底层数据存储的细节，这样业务线不需要关注数据是来自于cache还是DB。



非主流方案是异步缓存更新：业务线所有的写操作都走数据库，所有的读操作都总缓存，由一个异步的工具来做数据库与缓存之间数据的同步，具体细节是：

（1）要有一个init cache的过程，将需要缓存的数据全量写入cache

（2）如果DB有写操作，异步更新程序读取binlog，更新cache

在（1）和（2）的合作下，cache中有全部的数据，这样：

（a）业务线读cache，一定能够hit（很短的时间内，可能有脏数据），无需关注数据库

（b）业务线写DB，cache中能得到异步更新，无需关注缓存

这样将大大简化业务线的调用逻辑，存在的缺点是，如果缓存的数据业务逻辑比较复杂，async-update异步更新的逻辑可能也会比较复杂。

## 结论强调

（1）淘汰缓存是一种通用的缓存处理方式

（2）先淘汰缓存，再写数据库的时序是毋庸置疑的

（3）服务化是向业务方屏蔽底层数据库与缓存复杂性的一种通用方式

# 缓存与数据库一致性保证

本文主要讨论这么几个问题：

（1）啥时候数据库和缓存中的数据会不一致

（2）不一致优化思路

（3）如何保证数据库与缓存的一致性

“先操作缓存，在写数据库成功之前，如果有读请求发生，可能导致旧数据入缓存，引发数据不一致”，这就是本文要讨论的主题。

## 为什么数据会不一致

写流程：

（1）先淘汰cache

（2）再写db

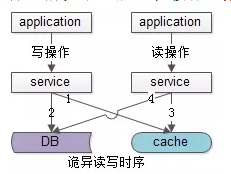
读流程：

（1）先读cache，如果数据命中hit则返回

（2）如果数据未命中miss则读db

（3）将db中读取出来的数据入缓存

什么情况下可能出现缓存和数据库中数据不一致呢？



在分布式环境下，数据的读写都是并发的，上游有多个应用，通过一个服务的多个部署（为了保证可用性，一定是部署多份的），对同一个数据进行读写，在数据库层面并发的读写并不能保证完成顺序，也就是说后发出的读请求很可能先完成（读出脏数据）：

（a）发生了写请求A，A的第一步淘汰了cache（如上图中的1）

（b）A的第二步写数据库，发出修改请求（如上图中的2）

（c）发生了读请求B，B的第一步读取cache，发现cache中是空的（如上图中的步骤3）

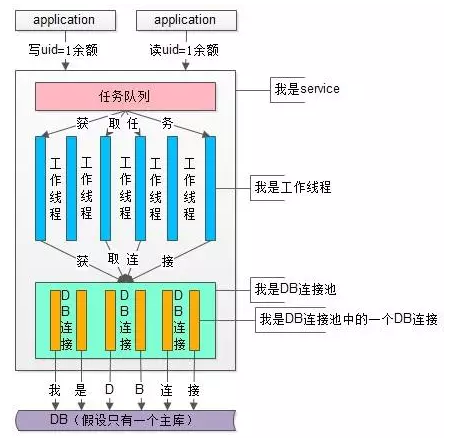
（d）B的第二步读取数据库，发出读取请求，此时A的第二步写数据还没完成，读出了一个脏数据放入cache（如上图中的步骤4）

即在数据库层面，后发出的请求4比先发出的请求2先完成了，读出了脏数据，脏数据又入了缓存，缓存与数据库中的数据不一致出现了

## 不一致优化思路

能否做到先发出的请求一定先执行完成呢？常见的思路是“串行化”，今天将和大家一起探讨“串行化”这个点。

先一起细看一下，在一个服务中，并发的多个读写SQL一般是怎么执行的



上图是一个service服务的上下游及服务内部详细展开，细节如下：

（1）service的上游是多个业务应用，上游发起请求对同一个数据并发的进行读写操作，上例中并发进行了一个uid=1的余额修改（写）操作与uid=1的余额查询（读）操作

（2）service的下游是数据库DB，假设只读写一个DB

（3）中间是服务层service，它又分为了这么几个部分

（3.1）最上层是任务队列

（3.2）中间是工作线程，每个工作线程完成实际的工作任务，典型的工作任务是通过数据库连接池读写数据库

（3.3）最下层是数据库连接池，所有的SQL语句都是通过数据库连接池发往数据库去执行的

工作线程的典型工作流是这样的：

void work\_thread\_routine(){

Task t = TaskQueue.pop(); // 获取任务

// 任务逻辑处理，生成sql语句

DBConnection c = CPool.GetDBConnection(); // 从DB连接池获取一个DB连接

c.execSQL(sql); // 通过DB连接执行sql语句

CPool.PutDBConnection(c); // 将DB连接放回DB连接池

}

提问：任务队列其实已经做了任务串行化的工作，能否保证任务不并发执行？

答：不行，因为

（1）1个服务有多个工作线程，串行弹出的任务会被并行执行

（2）1个服务有多个数据库连接，每个工作线程获取不同的数据库连接会在DB层面并发执行

提问：假设服务只部署一份，能否保证任务不并发执行？

答：不行，原因同上

提问：假设1个服务只有1条数据库连接，能否保证任务不并发执行？

答：不行，因为

（1）1个服务只有1条数据库连接，只能保证在一个服务器上的请求在数据库层面是串行执行的

（2）因为服务是分布式部署的，多个服务上的请求在数据库层面仍可能是并发执行的

提问：假设服务只部署一份，且1个服务只有1条连接，能否保证任务不并发执行？

答：可以，全局来看请求是串行执行的，吞吐量很低，并且服务无法保证可用性

完了，看似无望了，

1）任务队列不能保证串行化

2）单服务多数据库连接不能保证串行化

3）多服务单数据库连接不能保证串行化

4）单服务单数据库连接可能保证串行化，但吞吐量级低，且不能保证服务的可用性，几乎不可行，那是否还有解？

退一步想，其实不需要让全局的请求串行化，而只需要“让同一个数据的访问能串行化”就行。

在一个服务内，如何做到“让同一个数据的访问串行化”，只需要“让同一个数据的访问通过同一条DB连接执行”就行。

如何做到“让同一个数据的访问通过同一条DB连接执行”，只需要“在DB连接池层面稍微修改，按数据取连接即可”

获取DB连接的CPool.GetDBConnection()【返回任何一个可用DB连接】改为

CPool.GetDBConnection(longid)【返回id取模相关联的DB连接】

这个修改的好处是：

（1）简单，只需要修改DB连接池实现，以及DB连接获取处

（2）连接池的修改不需要关注业务，传入的id是什么含义连接池不关注，直接按照id取模返回DB连接即可

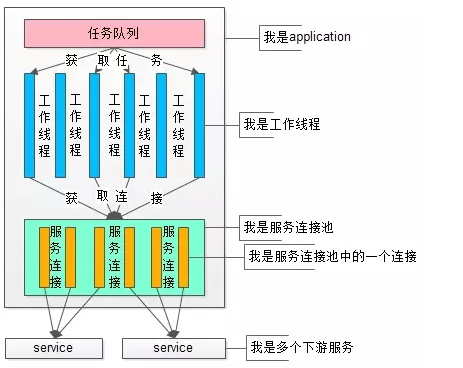
（3）可以适用多种业务场景，取用户数据业务传入user-id取连接，取订单数据业务传入order-id取连接即可

这样的话，就能够保证同一个数据例如uid在数据库层面的执行一定是串行的

稍等稍等，服务可是部署了很多份的，上述方案**只能保证同一个数据在一个服务上的访问，在DB层面的执行是串行化的**，实际上服务是分布式部署的，在全局范围内的访问仍是并行的，怎么解决呢？能不能做到同一个数据的访问一定落到同一个服务呢？

## 能否做到同一个数据的访问落在同一个服务上？

上面分析了服务层service的上下游及内部结构，再一起看一下应用层上下游及内部结构



上图是一个业务应用的上下游及服务内部详细展开，细节如下：

（1）业务应用的上游不确定是啥，可能是直接是http请求，可能也是一个服务的上游调用

（2）业务应用的下游是多个服务service

（3）中间是业务应用，它又分为了这么几个部分

（3.1）最上层是任务队列【或许web-server例如tomcat帮你干了这个事情了】

（3.2）中间是工作线程【或许web-server的工作线程或者cgi工作线程帮你干了线程分派这个事情了】，每个工作线程完成实际的业务任务，**典型的工作任务是通过服务连接池进行RPC调用**

（3.3）最下层是服务连接池，所有的RPC调用都是通过服务连接池往下游服务去发包执行的

工作线程的典型工作流是这样的：

voidwork\_thread\_routine(){

Task t = TaskQueue.pop(); // 获取任务

// 任务逻辑处理，组成一个网络包packet，调用下游RPC接口

ServiceConnection c = CPool.GetServiceConnection(); // 从Service连接池获取一个Service连接

c.Send(packet); // 通过Service连接发送报文执行RPC请求

CPool.PutServiceConnection(c); // 将Service连接放回Service连接池

}

似曾相识吧？没错，只要对服务连接池进行少量改动：

获取Service连接的CPool.GetServiceConnection()【返回任何一个可用Service连接】改为

CPool.GetServiceConnection(longid)【返回id取模相关联的Service连接】

这样的话，就能够保证同一个数据例如uid的请求落到同一个服务Service上。

## 总结

由于数据库层面的读写并发，引发的数据库与缓存数据不一致的问题（本质是后发生的读请求先返回了），可能通过两个小的改动解决：

（1）修改服务Service连接池，id取模选取服务连接，能够保证同一个数据的读写都落在同一个后端服务上

（2）修改数据库DB连接池，id取模选取DB连接，能够保证同一个数据的读写在数据库层面是串行的

六、遗留问题

提问：取模访问服务是否会影响服务的可用性？

答：不会，当有下游服务挂掉的时候，服务连接池能够检测到连接的可用性，取模时要把不可用的服务连接排除掉。

提问：取模访问服务与 取模访问DB，是否会影响各连接上请求的负载均衡？

答：不会，只要数据访问id是均衡的，从全局来看，由id取模获取各连接的概率也是均等的，即负载是均衡的。

===服务化与微服务===

===消息系统===

===架构实践===