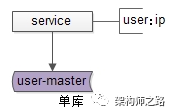
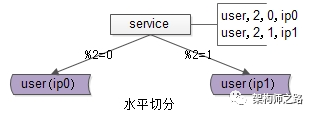
**数据库扩展性架构设计**

1. **数据库水平切分，秒级扩容**
2. ***问题起源***

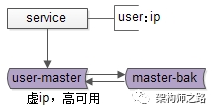
* *并发量大，流量大的互联网架构*，一般来说，数据库上层都有一个服务层，服务层记录了“业务库名”与“数据库实例”的映射关系，通过数据库连接池向数据库路由sql语句以执行，服务层配置用户库user对应的数据库实例物理位置为ip（其实是一个内网域名）：



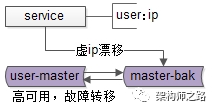
* *随着数据量的增大*，数据要进行水平切分，分库后将数据分布到不同的数据库实例（甚至物理机器）上，以达到降低数据量，增强性能的扩容目的，用户库user分布在两个实例上，ip0和ip1，服务层通过用户标识uid取模的方式进行寻库路由，模2余0的访问ip0上的user库，模2余1的访问ip1上的user库。



* 互联网架构需要保证数据库高可用，常见的一种方式—*使用双主同步+keepalived+虚ip*的方式保证数据库的可用性

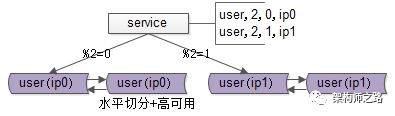


如上图：两个相互同步的主库使用相同的虚ip



如上图：当主库挂掉的时候，虚ip自动漂移到另一个主库，整个过程对调用方透明，通过这种方式保证数据库的高可用。

* 综合上两条，实际的架构，既有水平切分，又有高可用保证，所以实际的数据库架构是这样的：



出现的问题：

* 数据量持续增大，可以继续水平拆分，拆成更多的库，降低单库数据量，增加库主库实例（机器）数量，提高性能。
* 但是在分成x个库后，随着数据量的增加，要增加到y个库，数据库扩容的过程中，能否平滑，持续对外提供服务，保证服务的可用性，我们可以采用以下方案

1. ***停服务方案***

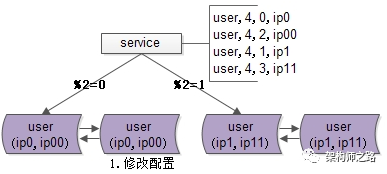
* *方案步骤*
* 站点挂一个公告“为了为广大用户提供更好的服务，本站点/游戏将在今晚00:00-2:00之间升级，届时将不能登录，用户周知”
* 停服务
* 新建y个库，做好高可用
* 数据迁移，重新分布，写一个数据迁移程序，从x个库里导入到y个库里，路由规则由%x升级为%y【最费时的一步】
* 修改服务配置，原来x行配置升级为y行
* 重启服务，连接新库重新对外提供服务
* *回滚*

若数据迁移或者迁移后的测试出错，则将配置改回x库，恢复服务，改天再挂公告。

* *缺点*
* 停服务，不高可用
* 技术同学压力大，所有工作要在规定时间内做完，根据经验，压力越大约容易出错（这一点很致命）
* 如果有问题第一时间没检查出来，启动了服务，运行一段时间后再发现有问题，难以回滚，需要回档，可能会丢失一部分数据

1. ***秒级扩容方案(三步走)***

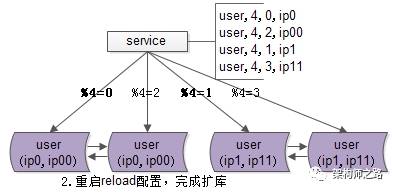
* *改配置*



* 数据库实例所在的机器做双虚ip，原来%2=0的库是虚ip0，现在增加一个虚ip00，%2=1的另一个库同理
* 修改服务的配置（不管是在配置文件里，还是在配置中心），将2个库的数据库配置，改为4个库的数据库配置，修改的时候要注意旧库与新库的映射关系：
  + %2=0的库，会变为%4=0与%4=2；
  + %2=1的部分，会变为%4=1与%4=3；

这样修改是为了保证，拆分后依然能够路由到正确的数据。

* *reload配置，实例扩容*



服务层reload配置可能有以下几种方式：

* + 比较原始的，重启服务，读新的配置文件
  + 高级一点的，配置中心给服务发信号，重读配置文件，重新初始化数据库连接池

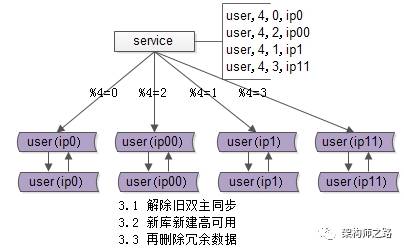
不管哪种方式，reload之后，数据库的实例扩容就完成了，原来是2个数据库实例提供服务，现在变为4个数据库实例提供服务，这个过程一般可以在秒级完成。

整个过程可以桌布重启，对服务的可用性和正确性完全没有影响

* + 即使%2寻库和%4寻库同时存在，也不影响数据的正确性，因为此时仍然是双主数据同步的
  + 服务reload之前是不对外提供服务的，冗余的服务能够保证高可用

完成了实例的扩展，会发现每个数据库的数据量依然没有下降，所以第三个步骤还要做一些收尾工作。

* *数据收缩*



* 把双虚ip修改回单虚ip
* 解除旧的双主同步，让成对库的数据不再同步增加
* 增加新的双主同步，保证高可用
* 删除掉冗余数据，例如：ip0里%4=2的数据全部干掉，只为%4=0的数据提供服务啦

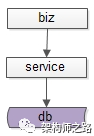
这样下来，每个库的数据量就降为原来的一半，数据收缩完成。

1. ***总结***

秒级扩容方案能够实现n库扩2n库的秒级、平滑扩容，增加数据库服务能力，降低单库一半的数据量，其核心原理是：成倍扩容，避免数据迁移。

1. **数据库水平切分，平滑扩容**
2. ***问题起源***

互联网有很多“数据量较大，并发量较大，业务复杂度较高”的业务场景，其典型系统分层架构如下：

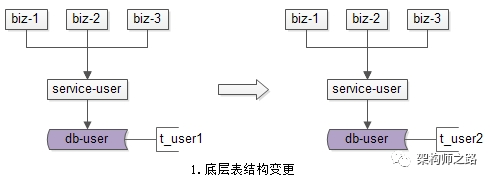


* 上游是业务层biz，实现个性化的业务逻辑
* 中游是服务层service，封装数据访问
* 下游是数据层db，存储固化的业务数据

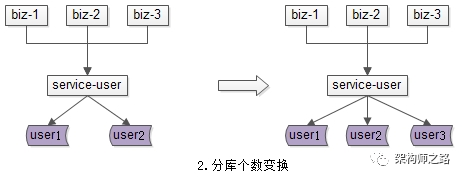
服务化分层架构的好处是，服务层屏蔽下游数据层的复杂性，例如缓存、分库分表、存储引擎等存储细节不需要向调用方暴露，而只向上游提供方便的RPC访问接口，当有一些数据层变化的时候，所有的调用方也不需要升级，只需要服务层升级即可。

有时会有这样的需求：

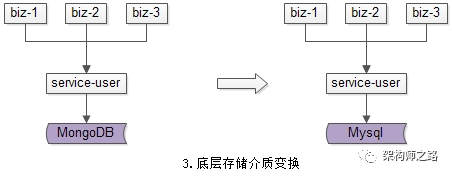
* 底层表结构变更：数据量非常大的情况下，数据表增加了一些属性，删除了一些属性，修改了一些属性。



* 分库个数变换：由于数据量的持续增加，底层分库个数非成倍增加。



* 底层存储介质变换：底层存储引擎由一个数据库换为另一个数据库

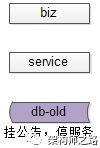


种种需求，都需要进行数据迁移，提出一种平滑迁移数据，迁移过程不停机，保证系统持续服务的方案

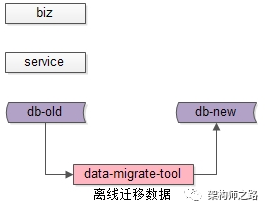
1. ***停机方案***

在讨论平滑迁移数据方案之前，先看下不平滑的停机数据迁移方案，主要分三个步骤。

* 挂一个类似“为了给广大用户提供更好的服务，服务器会在凌晨0:00-4:00进行停机维护”的公告，并在对应时段进行停机，这个时段系统没有流量进入。



* 停机后，研发一个离线的数据迁移工具，进行数据迁移。针对第一节的三类需求，会分别开发不同的数据迁移工具。
* 底层表结构变更需求：开发旧表导新表的工具
* 分库个数变换需求：开发2库导3库的工具
* 底层存储介质变换需求：开发Mongo导Mysql工具

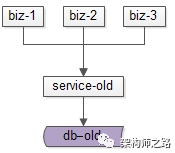
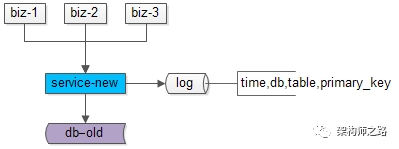


* 恢复服务，并将流量切到新库，不同的需求，可能会涉及不同服务升级
* 底层表结构变更需求：服务要升级到访问新表
* 分库个数变换需求：服务不需要升级，只需要改寻库路由配置
* 底层存储介质变换需求：服务升级到访问新的存储介质

总的来说，停机方案是相对直观和简单的，但对服务的可用性有影响，许多游戏公司的服务器升级，游戏分区与合区，可能会采用类似的方案。

除了影响服务的可用性，这个方案还有一个缺点，就是必须在指定时间完成升级，这个对研发、测试、运维同学来说，压力会非常大，一旦出现问题例如数据不一致，必须在规定时间内解决，否则只能回滚。根据经验，人压力越大越容易出错，这个缺点一定程度上是致命的。

1. ***平滑迁移-追日志法***

* *步骤一*

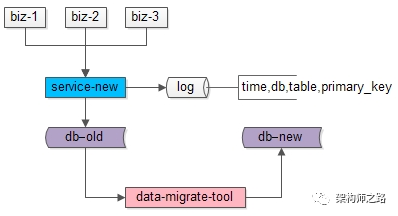
服务进行升级，记录“对旧库上的数据修改”的日志（这里的修改，为数据的insert, delete, update），这个日志不需要记录详细数据，主要记录：

* 被修改的库
* 被修改的表
* 被修改的唯一主键

具体新增了什么行，修改后的数据格式是什么，不需要详细记录。这样的好处是，不管业务细节如何变化，日志的格式是固定的，这样能保证方案的通用性。

这个服务升级风险较小：

* 写接口是少数接口，改动点较少
* 升级只是增加了一些日志，对业务功能没有任何影响
* *步骤二*



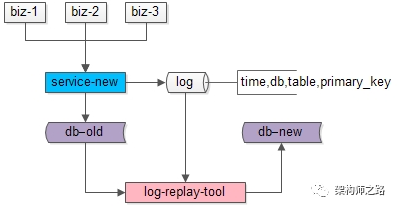
研发一个数据迁移工具，进行数据迁移。这个数据迁移工具和离线迁移工具一样，把旧库中的数据转移到新库中来。

这个小工具风险很小

* + 整个过程中旧库依然对线上提供服务
  + 小工具的复杂度较低
  + 任何时间发现问题，都可以把新库中的数据干掉重来
  + 可以限速慢慢迁移，技术没有时间压力

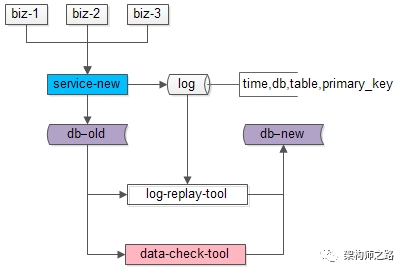
数据迁移的过程中，旧库依然对线上提供着服务，库中的数据随时可能变化，这个变化并没有反映到新库中来，于是旧库和新库的数据并不一致，所以不能直接切库，需要将数据追平。接下来步骤一所做的准备工作开始起作用。

* *步骤三*



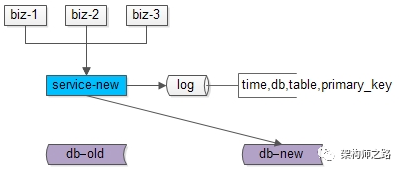
研发一个读取日志并迁移数据的小工具，要把步骤二迁移数据过程中产生的差异数据追平。这个小工具需要做的是：

* + 读取日志，得到哪个库、哪个表、哪个主键发生了变化
  + 把旧库中对应主键的记录读取出来
  + 把新库中对应主键的记录替换掉
* *步骤四*



在持续重放日志，追平数据的过程中，研发一个数据校验的小工具，将旧库和新库中的数据进行比对，直到数据完全一致。

* *步骤五*

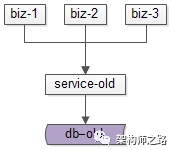
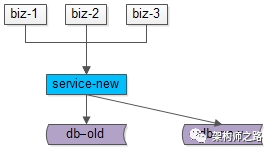


在数据比对完全一致之后，将流量迁移到新库，新库提供服务，完成迁移。

如果步骤四数据一直是99.9%的一致，不能完全一致，也是正常的，可以做一个秒级的旧库readonly，等日志重放程序完全追上数据后，再进行切库切流量。

1. ***平滑迁移-双写法***

* *步骤一*

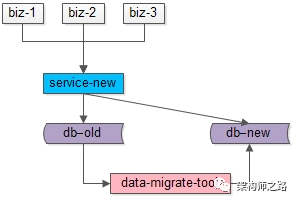
服务进行升级，对“对旧库上的数据修改”（这里的修改，为数据的insert, delete, update），在新库上进行相同的修改操作，这就是所谓的“双写”，主要修改操作包括：

* + 旧库与新库的同时insert
  + 旧库与新库的同时delete
  + 旧库与新库的同时update

由于新库中此时是没有数据的，所以双写旧库与新库中的affect rows可能不一样，不过这完全不影响业务功能，只要不切库，依然是旧库提供业务服务。

这个服务升级风险较小：

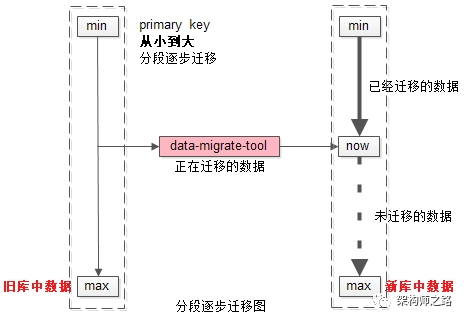
* + 写接口是少数接口，改动点较少
  + 新库的写操作执行成功与否，对业务功能没有任何影响
* *步骤二*



研发一个数据迁移工具，进行数据迁移，把旧库中的数据转移到新库中来。这个小工具的风险较小：

* + 整个过程依然是旧库对线上提供服务
  + 小工具的复杂度较低
  + 任何时间发现问题，都可以把新库中的数据干掉重来
  + 可以限速慢慢迁移，技术同学没有时间压力

数据迁移完成之后，就能够切到新库提供服务。因为前置步骤进行了双写，所以理论上数据迁移完之后，新库与旧库的数据应该完全一致。由于迁移数据的过程中，旧库新库双写操作在同时进行，下面可以证明数据迁移完成之后数据是完全一致的：



如上：

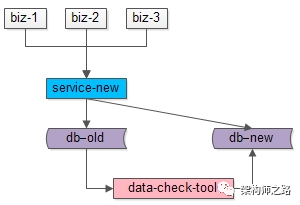
* + 左侧是旧库中的数据，右侧是新库中的数据
  + 按照primary key从min到max的顺序，分段，限速进行数据的迁移，假设已经迁移到now这个数据段

数据迁移过程中的修改操作分别讨论：

* + 假设迁移过程中进行了一个双insert操作，旧库新库都插入了数据，数据一致性没有被破坏
  + 假设迁移过程中进行了一个双delete操作，这又分为两种情况
    - 假设这delete的数据属于[min,now]范围，即已经完成迁移，则旧库新库都删除了数据，数据一致性没有被破坏
    - 假设这delete的数据属于[now,max]范围，即未完成迁移，则旧库中删除操作的affect rows为1，新库中删除操作的affect rows为0，但是数据迁移工具在后续数据迁移中，并不会将这条旧库中被删除的数据迁移到新库中，所以数据一致性仍没有被破坏
  + 假设迁移过程中进行了一个双update操作，可以认为update操作是一个delete加一个insert操作的复合操作，所以数据仍然是一致的
  + 在一种非常非常非常极限的情况下：
    - date-migrate-tool刚好从旧库中将某一条数据X取出
    - 在X插入到新库中之前，旧库与新库中刚好对X进行了双delete操作
    - date-migrate-tool再将X插入到新库中

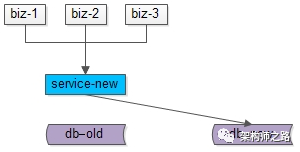
这样，会出现新库比旧库多出一条数据X。所以不论怎样都需要步骤三来进行检查。

* *步骤三*



在数据迁移完成之后，需要使用数据校验的小工具，将旧库和新库中的数据进行比对，完全一致则符合预期，如果出现步骤二中的极限不一致情况，则以旧库中的数据为准。

* 步骤四



数据完全一致之后，将流量切到新库，完成平滑数据迁移。

1. **互联网在线表结构变更**
2. ***问题起源***

* 数据量大、并发量高场景，在线数据库属性扩展
* 数据库表结构扩展性设计

1. ***禁术***

* alter table add column；要坚持这个方案的，也不多解释了，大数据高并发情况下，一定不可行
* 通过增加表的方式扩展，通过外键join来查询；大数据高并发情况下，join性能较差，一定不可行
* 通过增加表的方式扩展，通过视图来对外；一定不可行，大数据高并发情况下，互联网不怎么使用视图，至少58禁止使用视图
* 必须遵循“第x范式”的方案；一定不可行。互联网的主要矛盾之一是吞吐量，为了保证吞吐量甚至可能牺牲一些事务性和一致性，通过反范式的方式来确保吞吐量的设计是很常见的，例如：冗余数据。互联网的主要矛盾之二是可用性，为了保证可用性，常见的技术方案也是数据冗余。在互联网数据库架构设计中，第x范式真的没有这么重要

1. ***可用方案***

* *版本号+通用列*



以上面的用户表为例，假设只有uid和name上有查询需求，表可以设计为user(uid, name, version, ext)

* uid和name有查询需求，必须设计为单独的列并建立索引
* version是版本号字段，它对ext进行了版本解释
* ext采用可扩展的字符串协议载体，承载被查询的属性

这样做的优点在于：

* 可以随时动态扩展属性
* 新旧两种数据可以同时存在
* 迁移数据方便，写个小程序将旧版本ext的改为新版本的ext，并修改version

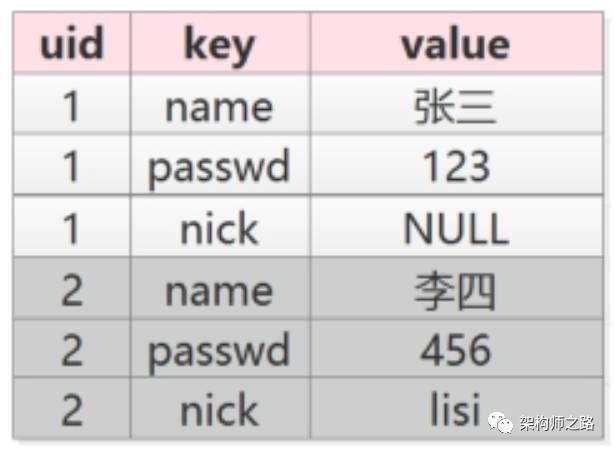
这样做的缺点在于

* ext里的字段无法建立索引
* ext里的key值有大量冗余，建议key短一些

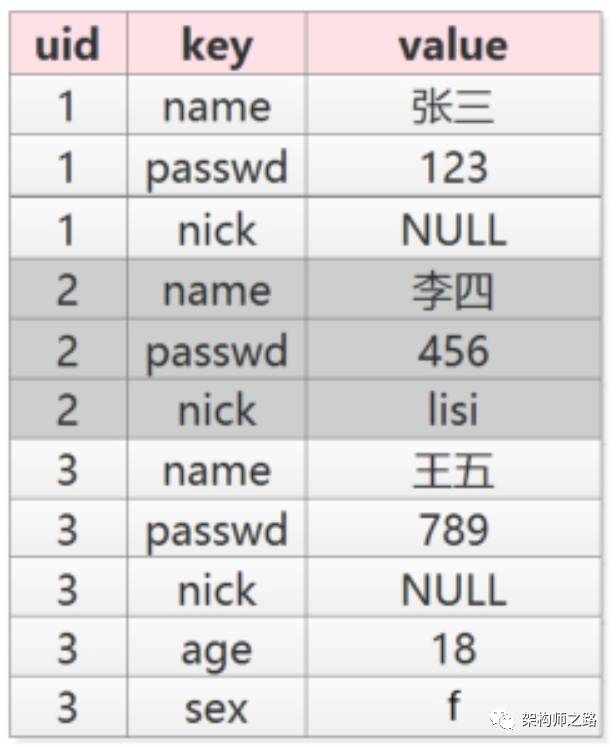
这样改进：如果ext里的属性有索引需求，可能Nosql的如MongoDB会更适合。

* *通过扩展行的方式扩展属性*

以上面的用户表为例，可以设计为user(uid, key, value)。初期有name, passwd, nick三个属性，那么数据为：



未来扩展了age和sex两个属性，数据变为：



这样做的优点在于

* 可以随时动态扩展属性
* 新旧两种数据可以同时存在
* 迁移数据方便，写个小程序可以将新增的属性加上
* 各个属性上都可以查询

缺点在于

* key值有大量冗余，建议key短一些
* 本来一条记录很多属性，会变成多条记录，行数会增加很多

应用场景

* 服务端，wordpress，EAV，配置，统计项等都经常使用这个方案。
* 客户端（APP或者PC），保存个人信息也经常使用这个方案。
* *提前预留一些reserved字段。*这个是可以的。

但如果预留过多，会造成空间浪费，预留过少，不一定达得到扩展效果。

* *通过增加表的方式扩展列*，*上游通过service来屏蔽底层的细节。*

这个也是可以的。 UserExt(uid, newCol1, newCol2)就是这样的方案（但join连表和视图是不行的）

* **新表+触发器+迁移数据+rename（pt-online-schema-change）**

【以user(uid, name, passwd)扩展到user(uid, name, passwd, age, sex)为例】

* 基本原理
  + 先创建一个扩充字段后的新表user\_new(uid, name, passwd, age, sex)
  + 在原表user上创建三个触发器，对原表user进行的所有insert/delete/update操作，都会对新表user\_new进行相同的操作
  + 分批将原表user中的数据insert到新表user\_new，直至数据迁移完成
  + 删掉触发器，把原表移走（默认是drop掉）
  + 把新表user\_new重命名（rename）成原表user

扩充字段完成。

* 优点：整个过程不需要锁表，可以持续对外提供服务
* 注意事项
  + 变更过程中，最重要的是冲突的处理，一条原则，以触发器的新数据为准，这就要求被迁移的表必须有主键（这个要求基本都满足）
  + 变更过程中，写操作需要建立触发器，所以如果原表已经有很多触发器，方案就不行（互联网大数据高并发的在线业务，一般都禁止使用触发器）
  + 触发器的建立，会影响原表的性能，所以这个操作建议在流量低峰期进行

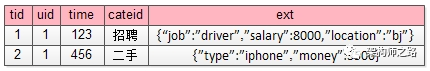
1. **应用：100亿数据1万属性数据架构设计(帖子中心为例)**
2. ***帖子中心信息的特点***

* 每个品类的属性千差万别，不同类型的帖子属性完全不同同一类型不同细类的属性又完全不同(部分相同)，假设有近万个属性
* 帖子量很大，100亿级别
* 每个属性上都有查询需求（各组合属性上都可能有组合查询需求）
* 查询量很大，每秒几10万级别

1. ***解决方案***

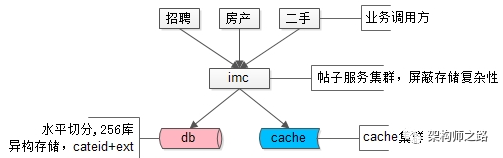
* *统一帖子中心服务：version+ext*

tiezi(tid,uid, time, title, cate, subcate, xxid, ext)



* + 一些通用的字段抽取出来单独存储
  + 通过cate, subcate, xxid等来定义ext是何种含义
  + 通过ext来存储不同业务线的个性化需求

帖子中心最核心的帖子数据，100亿的数据量，分256库，异构数据mysql存储，上层架了一个服务，使用memcache做缓存，是最核心的统一服务IMC（Imformation Management Center）



这样就解决了海量异构数据的存储问题，遇到的新问题是：

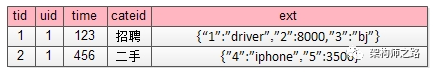
* + 每条记录ext内key都需要重复存储，占据了大量的空间，能否压缩存储
  + cateid已经不足以描述ext内的内容，品类有层级，深度不确定，ext能否具备自描述性
  + 随时可以增加属性，保证扩展性
* *统一类目属性服务*

抽象出一个统一的类目、属性服务，单独来管理这些信息，而帖子库ext字段里json的key，统一由数字来表示，减少存储空间。

通过



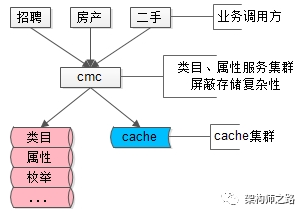
把ext做更改结果如下：



如果ext里某个key的value不是正则校验的值，而是枚举值时，需要有一个对值进行限定的枚举表来进行校验



协助解释帖子中心最核心的帖子数据，描述品类层级关系，保证各类目属性扩展性，保证各属性值合理性校验，就是帖子中心另一个统一的核心服务CMC（Category Management Center）。

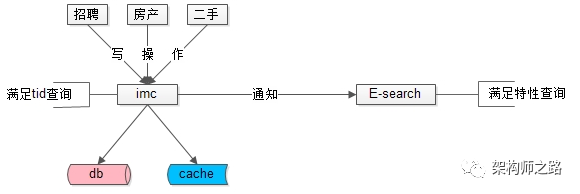


这样解决了解决了key压缩，key描述，key扩展，value校验，品类层级的问题，但是，每个品类下帖子的属性各不相同，查询需求各不相同，必须解决100亿数据量，1万属性的查询需求。

* *统一检索服务*

数据量很大的时候，不同属性上的查询需求，不可能通过组合索引来满足所有查询需求，这时提出“外置索引，统一检索服务”的技术方案：

* 数据库提供“帖子id”的正排查询需求
* 所有非“帖子id”的个性化检索需求，统一走外置索引

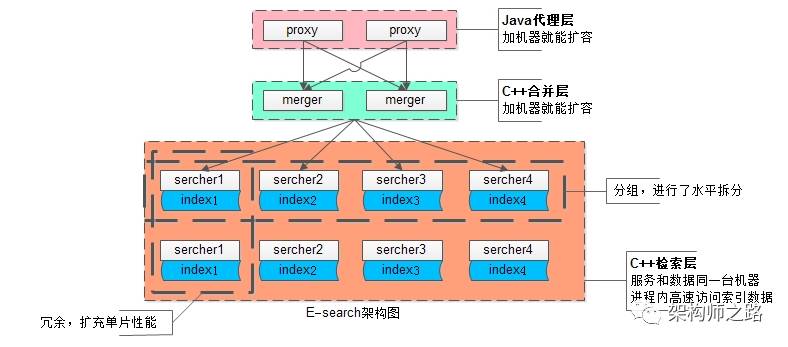


元数据与索引数据的操作遵循：

* 对帖子进行tid正排查询，直接访问帖子服务
* 对帖子进行修改，帖子服务通知检索服务，同时对索引进行修改
* 对帖子进行复杂查询，通过检索服务满足需求

E-search搜索引擎(不同于elasticsearch)

为应对100亿级别数据量、几十万级别的吞吐量，业务线各种复杂的复杂检索查询，扩展性是设计重点



* 统一的Java代理层集群，其无状态性能够保证增加机器就能扩充系统性能
* 统一的合并层C服务集群，其无状态性也能够保证增加机器就能扩充系统性能
* 搜索内核检索层C服务集群，服务和索引数据部署在同一台机器上，服务启动时可以加载索引数据到内存，请求访问时从内存中load数据，访问速度很快
  + 为了满足数据容量的扩展性，索引数据进行了水平切分，增加切分份数，就能够无限扩展性能
  + 为了满足一份数据的性能扩展性，同一份数据进行了冗余，理论上做到增加机器就无限扩展性能

系统时延，100亿级别帖子检索，包含请求分合，拉链求交集，从merger层均可以做到10ms返回。【一致性不是主要矛盾，E-search会定期全量重建索引，以保证即使数据不一致，也不会持续很长的时间】

* **综合**，面对100亿数据量，1万列属性，10万吞吐量的业务需求采用了元数据服务、属性服务、搜索服务来解决的。

