

# Music, Coffee & Programme

Home

Posts

About

## 又拍网架构中的分库设计

June 10, 2010

本文已经首发于[InfoQ中文站](#)，原文为[又拍网架构中的分库设计](#)如需转载，请附带本声明，谢谢。

又拍网是一个照片分享社区，从2005年6月至今积累了260万用户，1.1亿张照片，目前的日访问量为200多万。5年的发展历程里经历过许多起伏，也积累了一些经验，在这篇文章里，我要介绍一些我们在技术上的积累。

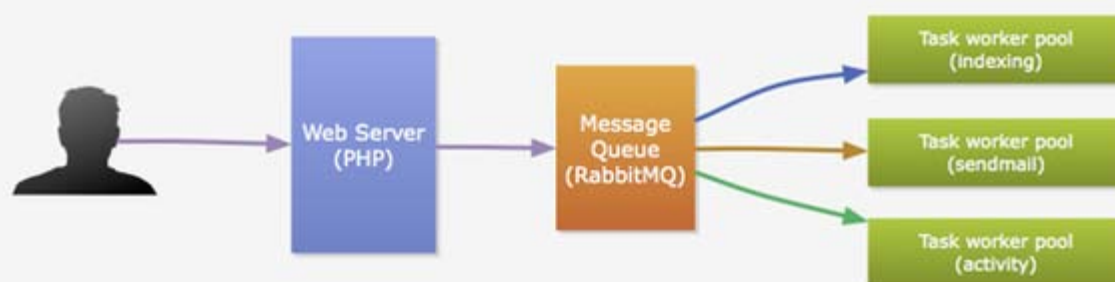
又拍和大多数Web2.0站点一样，构建于大量开源软件之上，包括MySQL, PHP, nginx, Python, memcached, redis, Solr, Hadoop, RabbitMQ等等。

又拍的服务器端开发语言主要是PHP和Python, PHP用于编写Web逻辑(通过HTTP和用户直接打交道)，而Python则主要用于开发内部服务和后台任务。而在客户端使用了大量的Javascript，要感谢一下MooTools这个JS框架使得我们很享受前端开发过程。另外，我们把图片处理过程从PHP进程里独立出来变成一个服务。这个服务基于nginx, 作为nginx的一个模块而开放REST API。



由于PHP的单线程模型，我们把耗时较久的运算和I/O操作从HTTP请求周期中分离出来，交给由Python实现的任务进程来完成，以保证请求响应速度。这些任务主要包括：邮件发送，数据索引，数据聚合，好友动态推送（稍候会有介绍）等等。通常这些任务由用户触发，并且，用户的一个行为可能会触发多种任务的执行。比如，用户上传了一张新的照片，我们需要更新索引，也需要向他的朋友推送一条新的动态。PHP通过消息

队列（我们用的是RabbitMQ）来触发任务执行。



数据库一向是网站架构中最具挑战性的，瓶颈通常出现在这里。又拍网的照片数据量很大，数据库也几度出现严重的压力问题。因此，这里我主要介绍一下又拍网在分库设计这方面的一些尝试。

## 分库设计

和很多使用MySQL的2.0站点一样，又拍网的MySQL集群经历了从最初的一个主库一个从库、到一个主库多个从库、然后到多个主库多个从库的一个发展过程。



最初是由一台主库和一台从库组成，当时从库只用作备份和容灾，当主库出现故障时，从库就手动变成主库，一般情况下，从库不作读写操作(同步除外)。随着压力的增加，我们加上了memcached，当时只用其缓存单行数据。但是，单行数据的缓存并不能很好的解决压力问题，因为单行数据的查询通常很快。所以我们把一些实时性要求不高的Query放到从库去执行。后面又通过添加多个从库来分流查询压力，不过随着数据量的增加，主库的写压力也越来越大。

在参考了一些相关产品和其它网站的做法后，我们决定进行数据库拆分。也就是将数据存放到不同的数据库服务器中，一般可以按两个纬度来拆分数据：

- 垂直拆分

是指按功能模块拆分，比如可以将群组相关表和照片相关表存放在不同的数据库中，这种方式多个数据库之间的表结构不同。

- 水平拆分

而水平拆分是将同一个表的数据进行分块保存到不同的数据库中，这些数据库中的表结构完全相同。

## 拆分方式

一般都会先进行垂直拆分，因为这种方式拆分方式实现起来比较简单，根据表名访问不同的数据库就可以了。但是垂直拆分方式并不能彻底解决所有压力问题，另外，也要看应用类型是否合适这种拆分方式。如果合适的话，也能很好的起到分散数据库压力的作用。比如对于豆瓣我觉得比较适合采用垂直拆分，因为豆瓣的各核心业务/模块(书籍、电影、音乐)相对独立，数据的增加速度也比较平稳。不同的是，又拍网的核心业务对象是用户上传的照片，而照片数据的增加速度随着用户量的增加越来越快。压力基本上都在照片表上，显然垂直拆分并不能从根本上解决我们的问题，所以，我们采用水平拆分的方式。

## 拆分规则

水平拆分实现起来相对复杂，我们要先确定一个拆分规则，也就是按什么条件将数据进行切分。一般2.0网站都以用户为中心，数据基本都跟随用户，比如用户的照片，朋友和评论等等。因此一个比较自然的选择是根据用户来切分。每个用户都对应一个数据库，访问某个用户的数据时，我们要先确定他/她所对应的数据库，然后连接到该数据库进行实际的数据读写。

那么，怎么样对应用户和数据库呢？我们有这些选择：

- 按算法对应

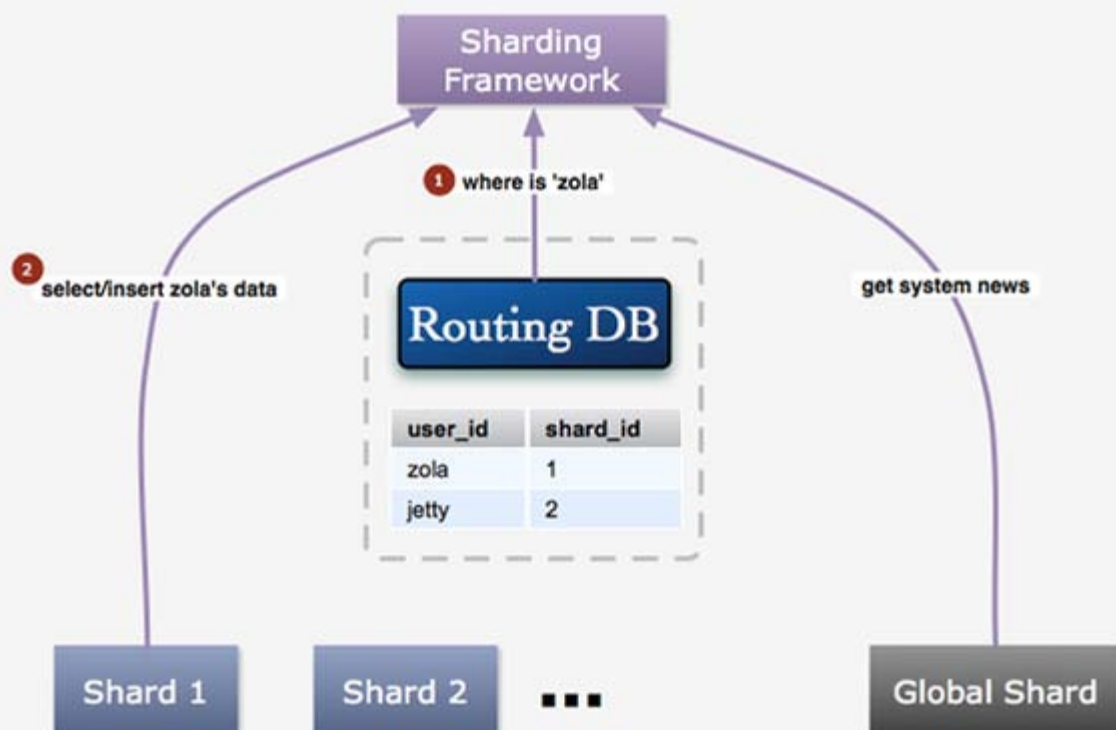
最简单的算法是按用户ID的奇偶性来对应，将奇数ID的用户对应到数据库A，而偶数ID的用户则对应到数据库B。这个方法的最大问题是，只能分成两个库。另一个算法是按用户ID所在区间对应，比如ID在0-10000之间的用户对应到数据库A，

ID在10000-20000这个范围的对应到数据库B，以此类推。按算法分实现起来比较方便，也比较高效，但是不能满足后续的伸缩性要求，如果需要增加数据库节点，必需调整算法或移动很大的数据集，比较难做到在不停止服务的前提下进行扩充数据库节点。

#### ■ 按索引/映射表对应

这种方法是指建立一个索引表，保存每个用户的ID和数据库ID的对应关系，每次读写用户数据时先从这个表获取对应数据库。新用户注册后，在所有可用的数据库中随机挑选一个为其建立索引。这种方法比较灵活，有很好的伸缩性。一个缺点是增加了一次数据库访问，所以性能上没有按算法对应好。

我们采用的是索引表的方式，我们愿意为其灵活性损失一些性能，更何况我们还有memcached，因为索引数据基本不会改变的缘故，缓存命中率非常高。所以能很大程度上减少了性能损失。



索引表的方式能够比较方便地添加数据库节点，在增加节点时，只要将其添加到可用数据库列表里即可。当然如果需要平衡各个节点的压力的话，还是需要进行数据的迁移，但是这个时候的迁移是少量的，可以逐步进行。要迁移用户A的数据，首先要将其状态

置为迁移数据中，这个状态的用户不能进行写操作，并在页面上进行提示。然后将用户A的数据全部复制到新增加的节点上后，更新映射表，然后将用户A的状态置为正常，最后将原来对应的数据库上的数据删除。这个过程通常会在凌晨进行，所以，所以很少会有用户碰到迁移数据中的情况。

当然，有些数据是不属于某个用户的，比如系统消息、配置等等，我们把这些数据保存在一个全局库中。

## 问题

分库会给你在应用的开发和部署上都带来很多麻烦。

- 不能执行跨库的关联查询

如果我们需要查询的数据分布于不同的数据库，我们没办法通过JOIN的方式查询获得。比如要获得好友的最新照片，你不能保证所有好友的数据都在同一个数据库里。一个解决办法是通过多次查询，再进行聚合的方式。我们需要尽量避免类似的需求。有些需求可以通过保存多份数据来解决，比如User-A和User-B的数据库分别是DB-1和DB-2，当User-A评论了User-B的照片时，我们会同时在DB-1和DB-2中保存这条评论信息，我们首先在DB-2中的photo\_comments表中插入一条新的记录，然后在DB-1中的user\_comments表中插入一条新的记录。这两个表的结构如下图所示。这样我们可以通过查询photo\_comments表得到User-B的某张照片的所有评论，也可以通过查询user\_comments表获得User-A的所有评论。另外可以考虑使用全文检索工具来解决某些需求，我们使用Solr来提供全站标签检索和照片搜索服务。

photo_comments	
column	type
photo_id	int
comment_id	int
author_id	int
posted_at	datetime
content	text

user_comments	
column	type
user_id	int
comment_id	int
photo_owner_id	int
photo_id	int
posted_at	datetime

- 不能保证数据的一致/完整性

跨库的数据没有外键约束，也没有事务保证。比如上面的评论照片的例子，很可能出现成功插入`photo_comments`表，但是插入`user_comments`表时却出错了。一个办法是在两个库上都开启事务，然后先插入`photo_comments`，再插入`user_comments`，然后提交两个事务。这个办法也不能完全保证这个操作的原子性。

- 所有查询必须提供数据库线索

比如要查看一张照片，仅凭一个照片ID是不够的，还必须提供上传这张照片的用户ID(也就是数据库线索)，才能找到它实际的存放位置。因此，我们必须重新设计很多URL地址，而有些老的地址我们又必须保证其仍然有效。我们把照片地址改成`/photos/{username}/{photo_id}/`的形式，然后对于系统升级前上传的照片ID，我们又增加一张映射表，保存`photo_id`和`user_id`的对应关系。当访问老的照片地址时，我们通过查询这张表获得用户信息，然后再重定向到新的地址。

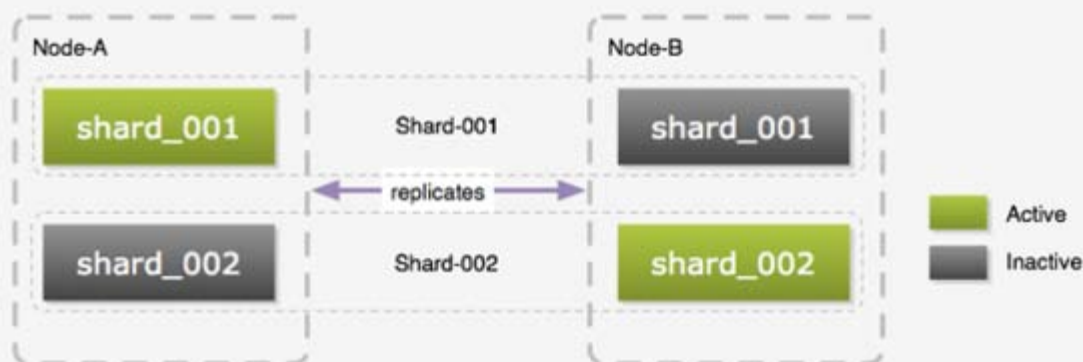
- 自增ID

如果要在节点数据库上使用自增字段，那么我们就不能保证全局唯一。这倒不是很严重的问题，但是当节点之间的数据发生关系时，就会使得问题变得比较麻烦。我们可以再来看看上面提到的评论的例子。如果`photo_comments`表中的`comment_id`是自增字段，当我们在`DB-2.photo_comments`表插入新的评论时，得到一个新的`comment_id`，假如值为**101**，而`User-A`的ID为**1**，那么我们还需要在`DB-1.user_comments`表中插入`(1, 101 ...)`。`User-A`是个很活跃的用户，他又评论了`User-C`的照片，而`User-C`的数据库是`DB-3`。很巧的是这条新评论的ID也是**101**，这种情况很有可能发生。那么我们又在`DB-1.user_comments`表中插入一行像这样`(1, 101 ...)`的数据。那么我们要怎么设置`user_comments`表的主键呢？可以不设啊，不幸的是有的时候(框架、缓存等原因)必需设置。那么可以以`user_id`、`comment_id`和`photo_id`为组合主键，但是`photo_id`也有可能一样(的确很巧)。看来只能再加上`photo_owner_id`了，但是这个结果让我实在有点无法接受，太复杂的组合键在数据写入时会带来一定的性能影响，这样的自然键看起来也很不自然。更重要的是我们需要在节点之间移动数据。所以，我们放弃了在节点上使用自增字段，想办法让这些ID变成全局唯一。为此增加了一个专门用来生成ID的数据库，这个库中的表结构都很简单，只有一个自增字段`id`。当我们要插入新的评论时，我们先在ID库的`photo_comments`表里插入一条空的记录以获得一个唯一的评论ID。当然这些逻辑都已经封装在我们的框架里了，对于开发人员是透明的。为什么

不用其它方案呢，比如一些支持incr操作的Key-Value数据库。我们还是比较放心把数据放在MySQL里。另外，我们会定期清理ID库的数据，以保证获取新ID的效率。

## 实现

我们称前面提到的一个数据库节点为Shard，一个Shard由两个台物理服务器组成，我们称它们为Node-A和Node-B，Node-A和Node-B之间是配置成Master-Master相互复制的。虽然是Master-Master的部署方式，但是同一时间我们还是只使用其中一个，原因是复制的延迟问题，当然在Web应用里，我们可以在用户会话里放置一个A或B来保证同一用户一次会话里只访问一个数据库，这样可以避免一些延迟问题。但是我们的Python任务是没有任何状态的，不能保证和PHP应用读写相同的数据库。那么为什么不配置成Master-Slave呢？我们觉得只用一台太浪费了，所以我们在每台服务器上都创建多个逻辑数据库。如下图所示，在Node-A和Node-B上都建立了shard\_001和shard\_002两个逻辑数据库，Node-A上的shard\_001和Node-B上的shard\_001组成一个Shard，而同一时间只有一个逻辑数据库处于Active状态。这个时候如果需要访问Shard-001的数据时，我们连接的是Node-A上的shard\_001，而访问Shard-002的数据则是连接Node-B上的shard\_002。以这种交叉的方式将压力分散到每台物理服务器上。以Master-Master方式部署的另一个好处是，我们可以不停止服务的情况下进行表结构升级，升级前先停止复制，升级前先停止复制，升级Inactive的库，然后升级应用，再将已经升级好的数据库切换到Active状态，原来的Active数据库切换到Inactive状态，然后升级它的表结构，最后恢复复制。当然这个步骤不一定适合所有升级过程，如果表结构的更改会导致数据复制失败，那么还是需要停止服务再升级的。



前面提到过添加服务器时，为了保证负载的平衡，我们需要迁移一部分数据到新的服务器上。为了避免短期内迁移的必要，我们在实际部署的时候，每台机器上部署了8个逻



辑数据库，添加服务器后，我们只要将这些逻辑数据库迁移到新服务器就可以了。最好是每次添加一倍的服务器，然后将每台的1/2逻辑数据迁移到一台新服务器上，这样能很好的平衡负载。当然，最后到了每台只有一个逻辑库时，迁移就无法避免了，不过那应该还是比较久远的事情了。

我们把分库逻辑都封装在我们的PHP框架里了，开发人员基本上不需要被这些繁琐的事情困扰。下面是使用我们的框架进行照片数据的读写的一些例子：

```
$Photos = new ShardedDBTable('Photos', 'yp_photos', 'user_id', array(
    'photo_id' => array('type' => 'long', 'primary' =>
true, 'global_auto_increment' => true),
    'user_id' => array('type' => 'long'),
    'title' => array('type' => 'string'),
    'posted_date' => array('type' => 'date'),
));

$photo = $Photos->new_object(array('user_id' => 1, 'title' =>
'Workforme'));
$photo->insert();

// 加载ID为10001的照片，注意第一个参数为用户ID
$photo = $Photos->load(1, 10001);

// 更改照片属性
$photo->title = 'Database Sharding';
$photo->update();

// 删除照片
$photo->delete();

// 获取ID为1的用户在2010-06-01之后上传的照片
$photos = $Photos->fetch(array('user_id' => 1, 'posted_date__gt' =>
'2010-06-01'));
```

首先要定义一个ShardedDBTable对象，所有的API都是通过这个对象开放。第一个参数是对象类型名称，如果这个名称已经存在，那么将返回之前定义的对象。你也可以通过get\_table('Photos')这个函数来获取之前定义的Table对象。第二个参数是对应的数据库表名，而第三个参数是数据库线索字段，你会发现在后面的所有API中全部需要指定这个字段的值。第四个参数是字段定义，其中photo\_id字段的global\_auto\_increment属性被置为true，这就是前面所说的全局自增ID，只要指定了这个属性，框架会处理好ID的事情。

如果我们要访问全局库中的数据，我们需要定义一个DBTable对象。



```
$Users = new DBTable('Users', 'yp_users', array(
    'user_id' => array('type' => 'long', 'primary' => true,
    'auto_increment' => true),
    'username' => array('type' => 'string'),
));
```

DBTable是ShardedDBTable的父类，除了定义时参数有些不同(DBTable不需要指定数据库线索字段)，它们提供一样的API。

## 缓存

我们的框架提供了缓存功能，对开发人员是透明的。

```
$photo = $Photos->load(1, 10001);
```

比如上面的方法调用，框架先尝试以Photos-1-10001为Key在缓存中查找，未找到的话再执行数据库查询并放入缓存。当更改照片属性或删除照片时，框架负责从缓存中删除该照片。这种单个对象的缓存实现起来比较简单。稍微麻烦的是像下面这样的列表查询结果的缓存。

```
$photos = $Photos->fetch(array('user_id' => 1, 'posted_date__gt' =>
    '2010-06-01'));
```

我们把这个查询分成两步，第一步先查出符合条件的照片ID，然后再根据照片ID分别查找具体的照片信息。这么做可以更好的利用缓存。第一个查询的缓存Key为Photos-list-{shard\_key}-{md5(查询条件SQL语句)}，Value是照片ID列表（逗号间隔）。其中shard\_key为用户\_id的值1。目前来看，列表缓存也不麻烦。但是如果用户修改了某张照片的上传时间呢，这个时候缓存中的数据就不一定符合条件了。所以，我们需要一个机制来保证我们不会从缓存中得到过期的列表数据。我们为每张表设置了一个revision，当该表的数据发生变化时(调用insert/update/delete方法)，我们就更新它的revision，所以我们把列表的缓存Key改为Photos-list-{shard\_key}-{md5(查询条件SQL语句)}-{revision}，这样我们就不会再得到过期列表了。

revision信息也是存放在缓存里的，Key为Photos-revision。这样做看起来不错，但是好像列表缓存的利用率不会太高。因为我们是整个数据类型的revision为缓存Key的后缀，显然这个revision更新的非常频繁，任何一个用户修改或上传了照片都会导致它的更新，哪怕那个用户根本不在我们要查询的Shard里。要隔离用户的动作对其他用户的影

响，我们可以通过缩小revision的作用范围来达到这个目的。所以revision的缓存Key变成Photos-{shard\_key}-revision，这样的话当ID为1的用户修改了他的照片信息时，只会更新Photos-1-revision这个Key所对应的revision。

因为全局库没有shard\_key，所以修改了全局库中的表的一行数据，还是会导致整个表的缓存失效。但是大部分情况下，数据都是有区域范围的，比如我们的帮助论坛的主题帖子，帖子属于主题。修改了其中一个主题的一个帖子，没必要使所有主题的帖子缓存都失效。所以我们在DBTable上增加了一个叫isolate\_key的属性。

```
$GLOBALS['Posts'] = new DBTable('Posts', 'yp_posts', array(
    'topic_id' => array('type' => 'long', 'primary' => true),
    'post_id' => array('type' => 'long', 'primary' => true,
    'auto_increment' => true),
    'author_id' => array('type' => 'long'),
    'content' => array('type' => 'string'),
    'posted_at' => array('type' => 'datetime'),
    'modified_at' => array('type' => 'datetime'),
    'modified_by' => array('type' => 'long'),
), 'topic_id');
```




注意构造函数的最后一个参数topic\_id就是指以字段topic\_id作为isolate\_key，它的作用和shard\_key一样用于隔离revision的作用范围。

ShardedDBTable继承自DBTable，所以也可以指定isolate\_key。ShardedDBTable指定了isolate\_key的话，能够更大幅度缩小revision的作用范围。比如相册和照片的关联表yp\_album\_photos，当用户往他的其中一个相册里添加了新的照片时，会导致其它相册的照片列表缓存也失效。如果我指定这张表的isolate\_key为album\_id的话，我们就把这种影响限制在了本相册内。

我们的缓存分为两级，第一级只是一个PHP数组，有效范围是Request。而第二级是memcached。这么做的原因是，很多数据在一个Request周期内需要加载多次，这样可以减少memcached的网络请求。另外我们的框架也会尽可能的发送memcached的gets命令来获取数据，从而减少网络请求。

## 总结

这个架构使得我们在很长一段时期内都不必再为数据库压力所困扰。我们的设计很多地方参考了netlog和flickr的实现，因此非常感谢他们将一些实现细节发布出来。

 喜欢   and 6 others liked this.

 DISQUS

## 添加新的评论

[登录](#)



Please wait...

## 显示 14 评论

排序 受欢迎的



xxd

请问更新Memcached的机制是什么，为什么不用MySQL UDF+Trigger更新？  
谢谢  
另，代码自动折开和高亮做的很棒

11 月前

[喜欢](#) [回复](#)



zolazhou

MySQL UDF是自定义类似 MAX() 之类的Function的吧，不是很了解，不知道怎么用来更新Memcached？  
Trigger就更不可能了，如果有一天memcached整合在MySQL里了，那还是有点可能的，不过容量是个问题。

11 月前 in reply to xxd

[喜欢](#) [回复](#)



xxd

我的意思是使用Trigger来调用这些函数更新Memcached。  
mysql memcached UDF 其实就是通过libmemcached来使用memcache的一系列函数，通过这些函数，能对memcache进行get, set, cas, append, prepend, delete, increment, decrement objects操作，如果通过mysql trigger来使用这些函数，那么就能通过mysql更好的，更自动的管理memcache。

11 月前 in reply to zolazhou

[喜欢](#) [回复](#)



zolazhou

哦，明白了，原来你是指配合使用两者。  
我不知道有 mysql memcached UDF 的存在，如果要自己去实现比起我们的方式要麻烦许多。  
这的确是一个不错的方法，回头研究一下。

11 月前 in reply to xxd

[喜欢](#) [回复](#)



Footman265

支持!

[11 月前](#)

[喜欢](#) [回复](#)



Chuanshuang Liucs

about页面里面联系你的，a标签的href都没设置。

[11 月前](#)

[喜欢](#) [回复](#)



zolazhou

的确，谢谢提醒

[11 月前](#) [in reply to Chuanshuang Liucs](#)

[喜欢](#) [回复](#)



Ivor Horton

请问楼主文中的示意图用什么软件画的？很漂亮。

[11 月前](#)

[喜欢](#) [回复](#)



zolazhou

OmniGraffle

[11 月前](#) [in reply to Ivor Horton](#)

[喜欢](#) [回复](#)



Luis Ashurei

关于文中描述的revision机制不是很明白：

->我们为每张表设置了一个revision，当该表的数据发生变化时，我们就更新它的revision，这样我们就不会再得到过期列表了。

这个和直接把当前指定缓存设为失效有什么区别呢？

[11 月前](#)

[喜欢](#) [回复](#)



zolazhou

对于一张表的查询有很多种，每一种的条件不一样也会产生不同的缓存key，那么要怎么样找出这些key，并使它们对应memcached里的entry失效呢？

revision机制不是使memcached里的entry失效，而是让你不再使用这些key（因为key里包含当前revision值），也就相当使所有老的entry失效了。

[11 月前](#) [in reply to Luis Ashurei](#)

[喜欢](#) [回复](#)



Fjctlzy

那么之前的`revision`的版本的数据已经放在缓存中了，失效了，如何删除呢？

9 月前 in reply to zolazhou

[喜欢](#) [回复](#)



zolazhou

在内存用到最大限额后，`memcached`会自动删除最久没有访问的`entry`

9 月前 in reply to Fjctlzy

[喜欢](#) [回复](#)



Fjctlzy

这个就让我纳闷了，难道不进行`memcached`使用量的监控吗？还是就是故意让她用满，然后自动覆盖？这样会不会有可能导致覆盖掉有用的数据？

9 月前 in reply to zolazhou

[喜欢](#) [回复](#)

[M 通过邮件订阅](#) [S RSS](#)

Copyright © 2010 Zola Zhou. All rights reserved.

本站采用HTML5和CSS3构建，所以请使用支持这些特性的浏览器浏览，推荐：[Safari](#), [Chrome](#), [Firefox](#)