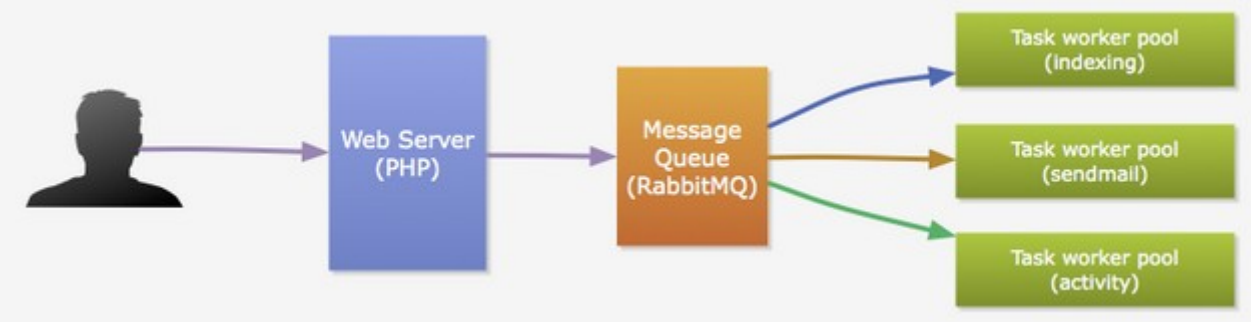
yupoo.com 的数据库架构

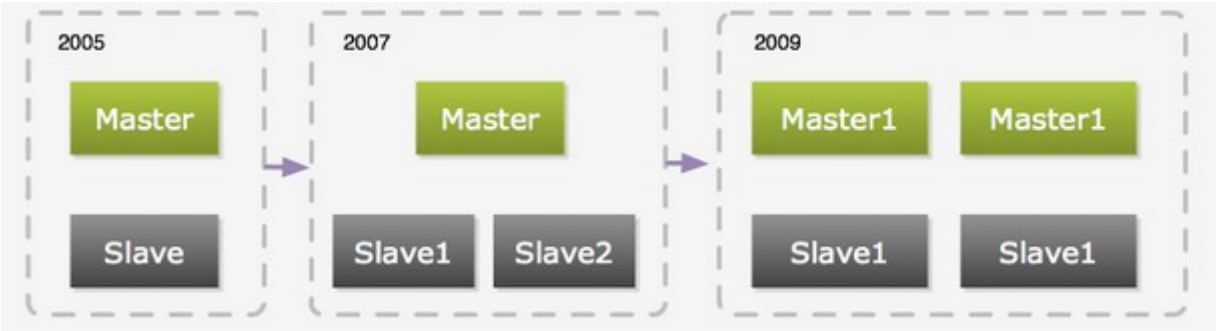
1. 数据情况：
目前为止 260 万注册用户，1.1 亿张照片，日访问量 200 万。

2. 开发语言：
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、数据聚合和好友动态推送等等。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。



PHP 和 Python 的协作，以及 RabbitMQ 的应用场景

3. 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地解决压力问题，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。

4. 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，

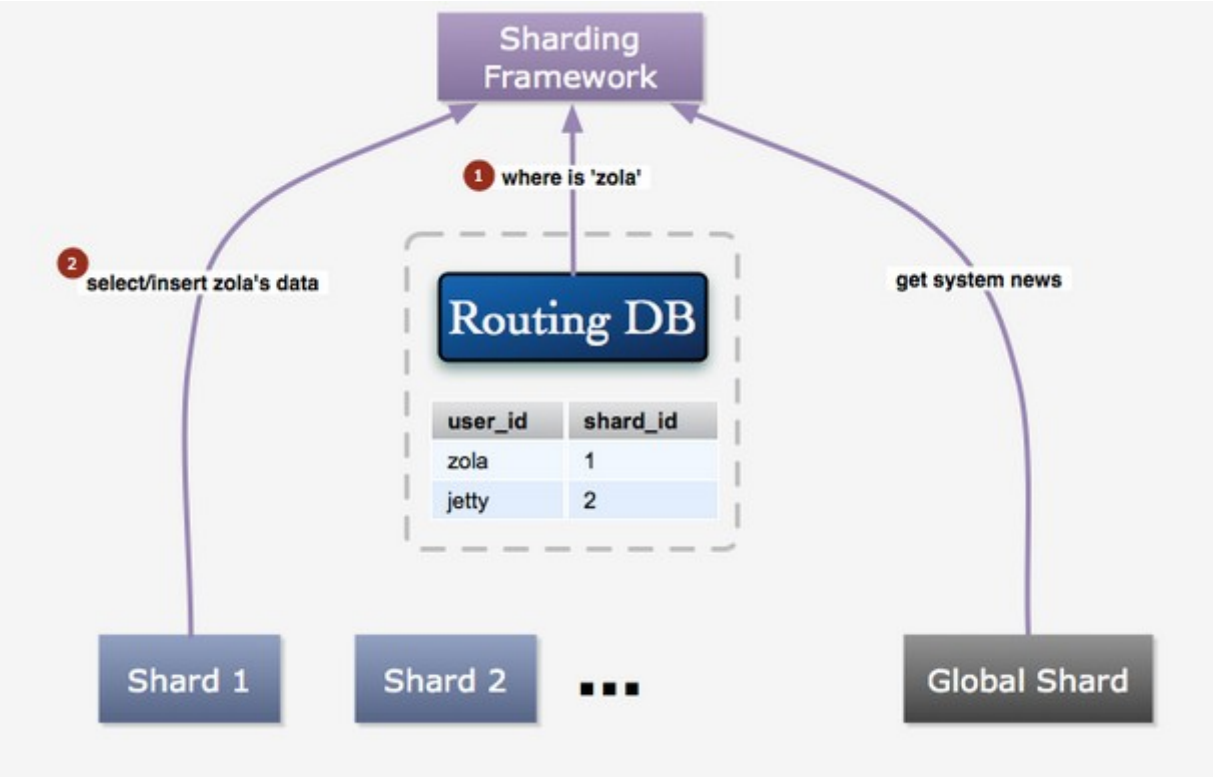
又拍网的核心业务对象是用户上传的照片，而照片数据的增加速度随着用户量的增加越来越快。压力基本上都在照片表上，显然垂直拆分并不能从根本上解决我们的问题，所以，我们采用水平拆分的方式 -> 为什么使用水平拆分的原因。

5. 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：
1) 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。

2) 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。

比较之后，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 -> 利用 memcached 减少索引造成的性能损失。



数据访问过程

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。这个过程通常会在临晨进行，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。

6. 分库之后存在的问题：
分库会给你在应用的开发和部署上都带来很多麻烦。

1) 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

评论表结构
理解这种表结构的设计，可以在一个库中获得某张照片的所有评论，或者在一个库中获得某个用户的所有评论，不需要跨库查询。

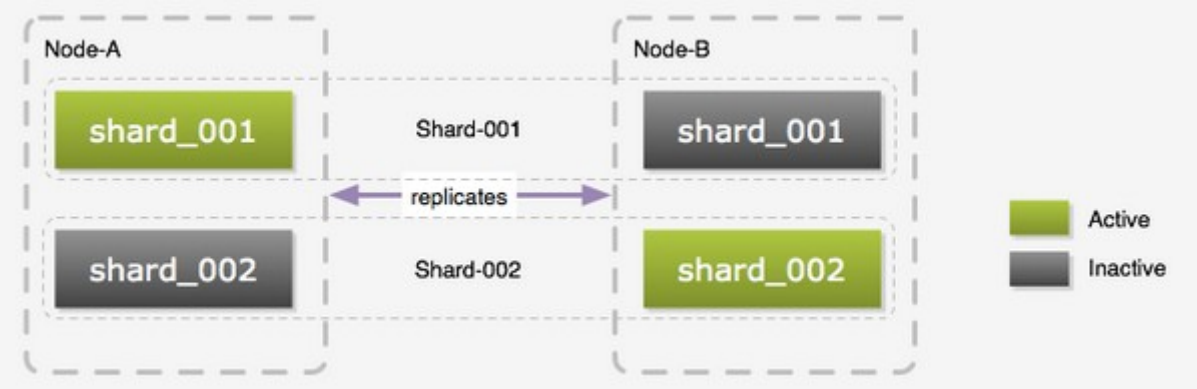
2) 不能保证数据的一致性/完整性
跨库的数据没有外键约束，也没有事务保证。比如上面的评论照片的例子， 很可能出现成功插入 photo_comments 表，但是插入 user_comments 表时却出错了。一个办法是在两个库上都开启事务，然后先插入 photo_comments，再插入 user_comments， 然后提交两个事务。这个办法也不能完全保证这个操作的原子性。

3) 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的 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。

4) 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 的自增字段，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，这种情况很用可能发生。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（标识一行数据）？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， 但是这个结果又让我们实在有点无法接受，太复杂的组合键在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。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。
获取全局唯一 ID 的解决方案：
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。

7. 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 上我们都建立了 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 Inactive 的库，然后升级应用，再将已经升级好的数据库切换成 Active 状态， 原来的 Active 数据库切换成 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。



数据库布局

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，最后到了每台上只有一个逻辑库时，迁移就无法避免了，不过那应该是比较久远的事情了。

8. 缓存：
1) 单个对象的缓存：
比如我们要查找某个用户的某张照片，先在缓存中查询，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。缓存的 Key 也比较容易生成，可以是简单的用户 ID 和照片 ID 的组合。

2) 列表查询结果的缓存：
比如我们要查询某个用户在某个时间段上传的照片，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
第二步再根据照片 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 user_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。因为我们是以整个数据类型的 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。 思路类似，可以通过增加限定来约束 revision 的影响范围，从而合理的利用缓存。

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。