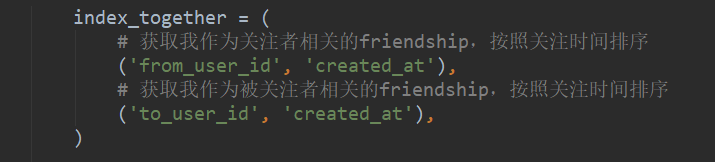
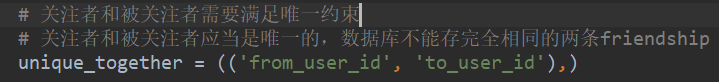
原先的Friendship存储在mysql中，只需要一张表单，因为mysql支持索引，之前建立了两个索引：

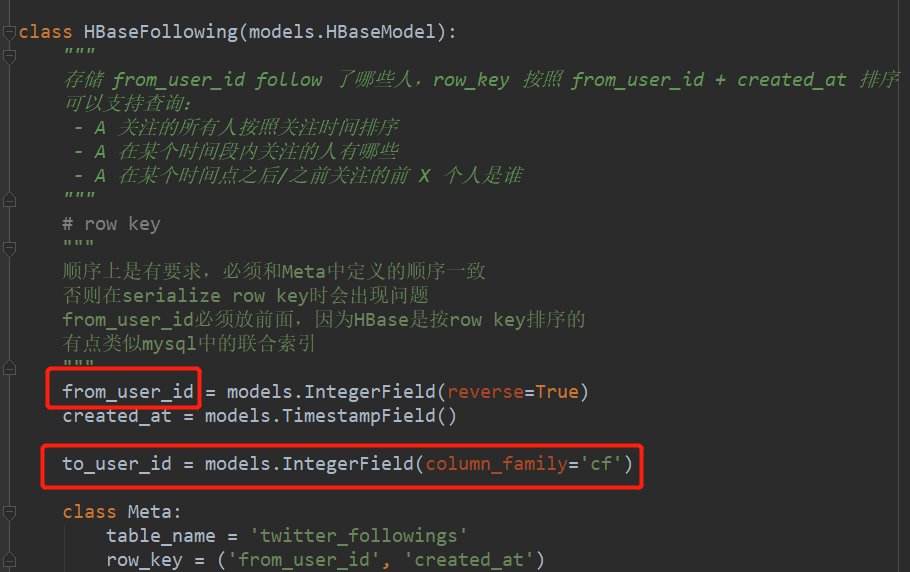


这样就可以查询关注某个人的所有人和某个人关注的所有人这两种情况。而且数据也只用存一份，由unique\_together来保证：



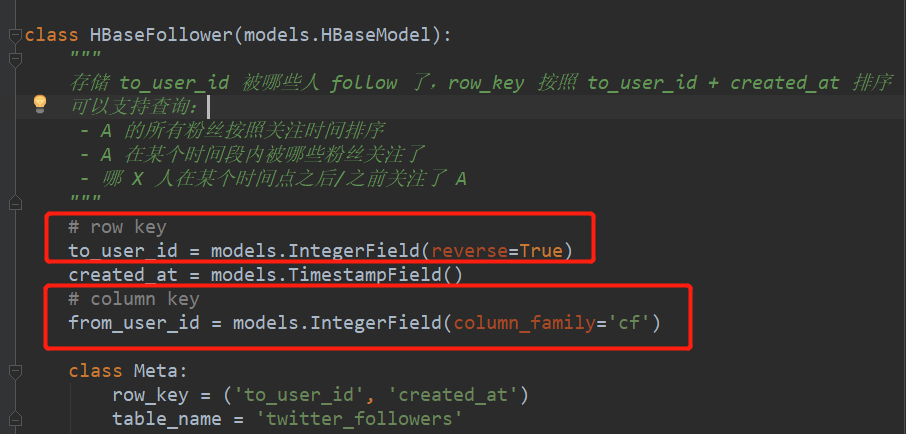
将Friendship迁移到HBase后，要拆成两张表单，因为HBase按Row Key排序，如果要查询关注某个用户的人和某个用户关注的人这两种情况，就必须拆成两个表单，用不同的Row Key。

HBaseFollowing：



from\_user\_id在Row Key中，to\_user\_id在column key中。

HBaseFollower：



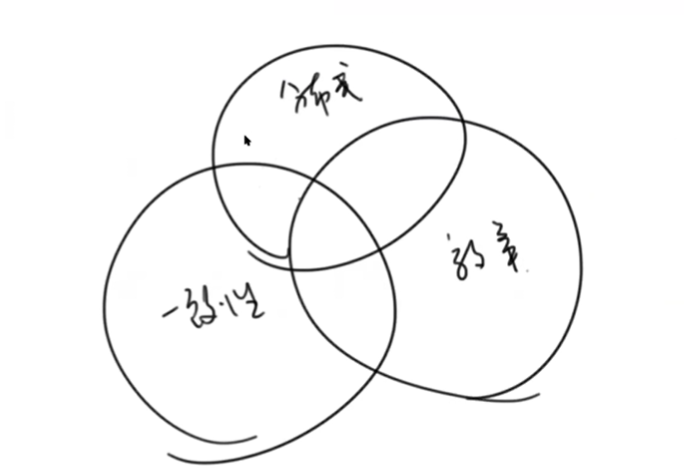
to\_user\_id在Row Key中，from\_user\_id在column key中。

Mysql中虽然只有一个表单，但它的索引其实类似一张看不见的表单，而HBase中查询可以更灵活，如果有某种查询需要，那就新建一张表单，这是人为可控的。

Friendship拆成两个表单后，可能会出现一致性的问题。两个不同的表单其实本质上没有什么约束关联，但在逻辑层却是有关联的。例如id=1的用户关注了id=2的用户，HBaseFollowing中要插入一条记录，同时HBaseFollower当中也要插入一条id=2的用户被id=1的用户关注了。如果某条记录插入成功，而另一条记录插入不成功，就会出现不一致的问题。

对于这类问题，目前没有很好的解决办法，HBase不支持事务机制，只能在实际应用时，增加类似try except的操作，发现某条记录有，而对应的记录不存在，就添加进去或者抛出异常提示，不进行其它的过多的保护机制。

选择数据库时，有3个重要的指标：分布式、一致性，效率。通常这三者没法共生，所以需要有取舍。选择HBase，它天然支持分布式，需要在一致性和效率之间取舍，对于社交系统，一致性要求没那么高，反而对效率要求较高。



HBase的Row Key是字符串排序，所以前缀查询和Range query是一致的，如果是整数排序，就不行了，因为1和10这两个Row Key会隔的很远，但字符串排序下1和10会排在一起。

目前HBase的Friendship表单HBaseFollowing和HBaseFollower并不支持unique\_together，这样导致可以在HBaseFollowing和HBaseFollower插入逻辑层面有相同意义的数据，例如在HBaseFollowing中插入id=1在时刻1关注了id=2，然后又插入id=1在时刻2关注了id=2，由于timestamp不同，所以Row Key不同，是允许插入的，但这两条数据在逻辑意义上是相同，都是id=1关注了id=2，这是重复的。

这种问题其实也没有什么统一的做法，一种解决办法是再新建一张表单：from\_user\_id和to\_user\_id在Row Key中，插入HBaseFollowing和HBaseFollower时先看看是否在HBaseFriendship中。



带来的问题时，之前提到的一致性问题会大幅度提示，因为你要多增加一条插入，会提高不一致性的概率。

如果需要查询A是否关注了B的话，可以直接在HBaseFollowing中查询A所有关注的对象，然后判断B是否在其中，这样效率会有所降低，但可以利用缓存Redis进行加速，利用Redis中的Set或Hash来存储A的所有关注对象。

灰度发布(Gray Release)：

1. 线上环境切换数据库
2. 提交某个功能，但对该功能不是很有信心，希望一旦出问题就快速撤回。把新功能一点一点的发布给用户，而不是一次性让所有用户都能使用。

大规模集群的部署是分批的，例如1000台机器，分10组，每组100台，第1组先停止接收正常的web请求，进行部署操作，包括代码拉取，重启等操作，其它组机器是正常的，等第1组完成部署后，重新接入网络，接收正常请求，第2组再执行之前类似的操作，依次类推，这样始终有10%的机器没有正常工作，而剩余的90%的机器是正常的。

如果通过更新代码进行某些新功能的发布或撤回，需要的时间比较长，尤其是大规模集群的更新部署，可以在代码中加入一个开关，通过类似if else这样的判断，来决定是否执行新功能或撤销。把开关需要读取的配置放置在类似Redis的缓存当中，这样可以在后台快速的修改。

实际情况下的底层数据库切换：

1. 双写，既要写到mysql，又要写到HBase中，读取数据从mysql中，用两个开关来操作
2. 写一个脚本，进行迁移操作，把mysql中之前所有的数据for循环写到HBase，这个时间非常长，可能有的数据会更改，不过对于HBase来说，它的每条数据都有时间戳，到可以区分新旧。
3. 迁移完成后，要重新执行一遍迁移，进行校验
4. 既写入mysql，又写HBase，读取时，既要从mysql读取，也要从HBase读取，两边的读取数据进行比较，看看两边的数据是否有不一致，如果存在不一致，记录在日志中，使用时间戳更新一点的数据，并且把数据修改成一致的。
5. 监控比较长的一段时间，例如1个月后，发现没有不一致的情况，写入到HBase，并从HBase读取。

整个流程都需要用到开关来控制，代码是一开始就写好的，通过开关来控制执行第几步操作。

灰度发布中，用户从5%->10%-20%....->100%进行增加，可以用user\_id % 100<N来筛选user\_id，N为指定的百分比，这样做的好处是从5%增加到10%时，原来在5%的用户一定在10%中。

A/B测试：

是否用nosql：

1. 数据结构简单
2. 数据量非常大
3. 数据查询要求简单
4. 数据有分布式需求

缓存不更新，而是删除：

参考：<https://segmentfault.com/a/1190000038572546>

1. 先删除缓存，再更新数据库，为了保证数据一致性，可以采用延时双删的方式
2. 先更新数据库，再删除缓存，为了保证数据一致性，可以采用监控mysql的binlog或消息队列等方式，不推荐。

以上两种方式都不建议更新缓存，因为如果先更新缓存，如果有多个request都更新，会出现数据不全的问题，缓存为了提高速度，一般是没有锁的，所以没法保证完整性， mysql数据库有行锁，可以保证最终更新到mysql中的数据是完整的，这样就会造成数据不一致。另外，有可能缓存中的数据被访问的频率并不高，如果每次修改数据都更新缓存，会造成资源浪费，可以只更新数据，并删除缓存，等到需要访问时，再把它从数据库中取出来，加入缓存中，这样也能保证一致性。

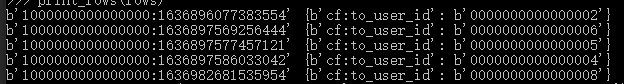
后端对response的格式进行调整，处理的流程：

1. 如果是大版本的改动，可以提供新的api，例如在v1，v2，v3这样的api
2. 如果是小范围的改动，则可以在api中添加参数，例如format，通过format参数来指定response的格式，在这个项目在，可以通过format来获取对应的serializer\_class来确定response的返回格式。

旧的格式不能删除掉，遗留代码不能删，有些客户端一直不更新，所以必须一直支持。

参考：<https://happybase.readthedocs.io/en/latest/api.html>，happybase中常用的scan()：

不指定任何参数，全局扫描整个表：



row\_start表示从某个row key开始查找，包含这个row key

table.scan(limit=2, row\_start=b'1000000000000000:1636897569256444')



row\_stop则稍微复杂点，如果没有指定row\_start，只有row\_stop，它表示从表单的第一条数据开始，查找指定limit的数据，直到碰到row\_stop，这里不包含row\_stop对应的row key。row\_stop相当于设置了一个终止的标志。

table.scan(limit=2, row\_stop=b'1000000000000000:1636982681535954')



从输出可以看出，尽管row\_stop设置的是最后一条row key，但它仍然输出了前两个数据，因为指定了limit=2

如果希望查找某个row key之前的指定limit条数据，可以设置row\_start和reverse=True来实现。

table.scan(limit=2, row\_start=b'1000000000000000:1636982681535954', reverse=True)



从输出可以看出，设置了row\_start和reverse=True后，找到的两条数据是从倒数第一条row key开始往上查找，同时因为是row\_start，所以包含了倒数第一条row key。