1：为什么要用缓存？

磁盘IO开销很大，很容易出现性能瓶颈。

2：数据库的缓存分为哪几种情况？

第一种：单个对象的缓存，这种缓存处理起来非常简单，直接放在本地HashMap中就行（注意线程安全和内存泄露，一般会用实现LRU算法的HashMap），但如果多应用服务器，需要同步各应用服务器之间的缓存，安全和性能会受到一定影响。直接放入本地HashMap这种处理方式，单App Server性能极好（100K的页面如果存入200个对象，大概每秒完成请求1500-2000个，单应用服务器每天“理论”完成5000万PV问题也不大），但扩展性不太好，如果加到10台应用服务器，每修改一个对象都要通知其他9台。所以，可以考虑把单个对象缓存放入集中的memcached/TT中，这样应用服务器扩展容易，各应用服务器之间无需通知，但单应用服务器的性能会下降很多（100K的页面如果存入200个对象，大概每秒完成请求150-200个，单应用服务器每天“理论”完成500万PV问题也不大）。

第二种：列表缓存，如select \* from “商品评价表” where “商品ID”=12344 limit 0,10

这种列表缓存处理起来有点复杂，mysql也好、hibernate也好，如果它不知道业务逻辑，它就无法对列表做很好的缓存，当“商品评价表”一有变动，mysql\hibernate就会清除所有“商品评价表”的列表缓存，这么做其实不厚道，如果一个表更新频繁，这么做列表缓存几乎失效，命中率太底了。比如lafaso有6万个商品，用户只对一个商品发布了评价，不至于其余59999个列表缓存，因为其余59999个商品的评价列表根本就没变。

第三种：长度缓存，如select count(\*) from “商品评价表” where “商品ID”=12344

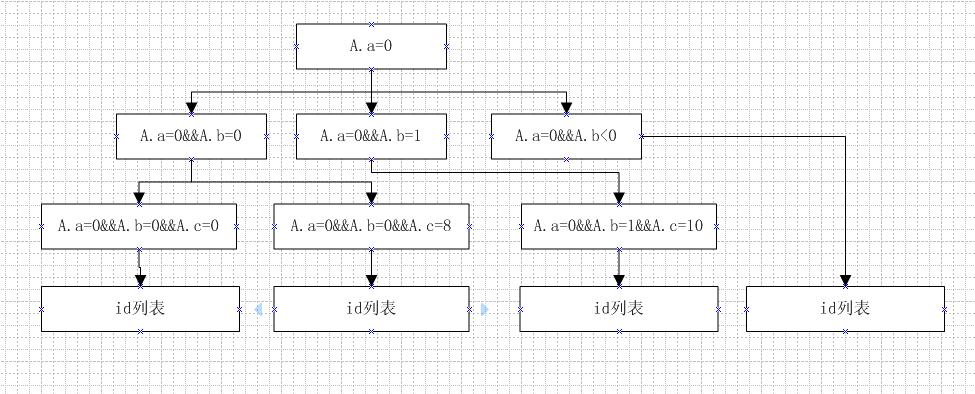
这个缓存也是很有用的，count长度有时会造成很大的性能瓶颈，limit如果只取前面少数记录，索引页建得好的话，性能一般不会有问题，但是count(\*)无法预测，不知道有多长，所以对count的缓存（或者发送到其他服务器上查询）是很有必要的。

3：为什么不用hibernate自带的缓存？

hibernate的确很强大，缓存也做得很好，但它毕竟不了解业务逻辑，没法做出通用的最高效的缓存，所以如果能自己根据业务逻辑写出更高效的缓存那最好了。下面我主要介绍一下列表缓存的改进。

3.1：最早我做了一个“通用”的列表缓存，key就是所有条件拼出来的字符串，value是查询的id列表，比如key是“a=0&&b=0&&c>0&&limit5”这样，如果增加了一个对象A，如果A.a=0，A.b=0，A.c=1（满足a=0,b=0,c>0）的条件，那么这个对象A肯定是这个列表的，所以这个列表需要重新获取。但是这么做开销太大，HashMap的遍历太费性能，分布式也没法做，一秒钟只能完成数百次的遍历，不可取，抛弃了。

3.2：后来我打算做一个“更通用”的列表缓存，用树状结构实现，每个查询条件是树的一个节点。还是如“a=0&&b=0&&c=1”这个列表，把它拆成三级，a=0是跟节点，b=0是a=0下面的一个子节点，c=1是b=0的子节点，如果再有一个查询列表“a=0&&b=0&&c=2”,那么c=2还是b=0的子节点，这样就形成了一个树状结构的列表缓存，当增加一个对象A，如果A.a=0,A.b=0,A.c=1，那么只需要清除一个分支节点即可，如果增加一个对象A，如果A.a=0,A.b=1，那么再在a=0下面开一个二级子节点。这么来缓存列表比较通用，无需配置就能知道哪些列表需要清除。但是需要解析查询条件，可以实现，但是有点复杂，一般没有必要。假设对象K有三个字段a、b、c，对于这三个查询字段，条件组合可能各种各样，最后形成的树状列表结构应该如下：



3.3：最终我选择了简单实用的列表缓存，就是散列缓存，还是如lafaso的“商品评价表”，很明显，每个商品都有自己一套评价列表，如果只正对某个商品作评价，其余商品评价列表根本没有变，还可以继续留在缓存里。散列列表缓存是HashMap<String,HashMap<String,List<String>>的结构，List存放的是id列表，而不是对象，如下：

|  |  |
| --- | --- |
| Key键（String型） | Value值（HashMap） |
| userId=2046 | |  |  | | --- | --- | | Key键（String型） | Value值（ArrayList） | | userId=2046#0,5 | id组成的List | | userId=2046#5,5 | id组成的List | | userId=2046#15,5 | id组成的List | | …… | | |
| userId=2047 | |  |  | | --- | --- | | Key键（String型） | Value值（ArrayList） | | userId=2047#0,5 | id组成的List | | userId=2047#5,5 | id组成的List | | userId=2047#15,5 | id组成的List | | …… | | |
| userId=2048 | |  |  | | --- | --- | | Key键（String型） | Value值（ArrayList） | | userId=2048#topicId=2008#0,5 | id组成的List | | userId=2048#5,5 | id组成的List | | userId=2048#15,5 | id组成的List | | …… | | |
| …… | |

这种列表缓存再很多地方都按照userId来散列就行，比如“我的订单”“我的收件箱”“我的帖子列表”“我的购物车”.......，也有按其他字段散列的，如“商品评论表”、”商品图片列表”就按照商品id来散列，这就需要使用者知道业务逻辑，配置几个散列字段即可。

4：什么时候清除列表缓存？

4.1：增加对象时，根据散列字段清除对应的列表缓存。如增加对象A（假设按照A.userId来散列），如果A.userId=100，那么只需要删除userId=100的列表缓存，直接remove就行，无需遍历。

4.2：删除对象时，同4.1。

4.3：修改对象比较麻烦，分三种情况。

第一种：如果明确知道修改不影响排序，比如修改商品的标题，没有列表会按照标题排序，那么无需清列表缓存。

第二种：如果修改商品的价格，因为列表中有按价格排序的地方，所以需要清除该商品对应的列表缓存。

第三种：如果修改商品的分类，即影响了两个列表，加入从分类“女人”->“男人”，那么即要清除“女人”的列表，也要清除“男人”的列表。如果什么都不知道，修改前删除列表缓存，修改后也删除列表缓存，这也需要程序员控制。

5：分布式列表缓存

上面已经介绍过，分布式对象缓存用集中式的cache存就行。但上面介绍的列表缓存是基于HashMap的，如果列表缓存也做了分布式，就需要各应用服务器之间通讯，这又带来的新的问题，可扩展性不好。所以我再改善了一下，把列表缓存也存入集中的cache中，这个集中的cache可以用jboss的treecache，也可以自己写一个cache，都没有问题，能实现remove功能就行。但我的解决办法还是用memcached，但由于memcached相当于一个巨大的HashMap，它的value是简单对象还好处理，如果value也是HashMap就比较难办了。在memcached中，对象缓存的key是“com.lafaso.bean.Product#1267364”这个样子，列表缓存的key是“com.lafaso.bean.Product#category=1000#price desc#0,100”这个样子，如果增加了一个Product（category=1000），我们无法知道memcached里面到底有多少“category=1000”，没办法一次性把category=1000的列表全部删除。最后我想了个办法，在category=1000这个key上做一个时间轴，这样只需要修改这个时间轴，所有category=1000的列表就不能用了，虽然还在memcached里面，但不用管，memcached最终会gc掉它的。memcached列表缓存的key可能会比较长，在存储的时候我做了Md5加密，统一成16位了。

6：bean和数据库操作的抽象

为了方便用统一的方法处理数据库，我把bean抽象出来一个BaseEntity，这个BaseEntity只包含了一个字段id，也就是primary key，所有其他bean都需要继承这个BaseEntity。这样所有的对象其实都是BaseEntity，这样Dao也可以抽象出来了，而不需要每个bean都要写一个Dao，我写了一个基本的数据库操作工具BaseManager，里面实现了数据库的增删改查和分布式缓存。如果增加一张表，只需要做下面几件事：**1：增加一个java bean；2：修改一下hibernate配置文件，把bean映射进去；3：在managers配置文件里增加一段配置，配置散列字段即可。**无需写一个sql就实现了对该表的增删改查和缓存逻辑，BaseManager的get方法如下：

public BaseEntity get(String id) throws HibernateException {  
   if(id==null||id.length()==0)return null ;  
   BaseEntity br = null;  
   //第一步：去memcached中查找  
    br = getObjectFromMemcached(getBeanVersion()+beanClass.getName()+"#"+id);  
    if(br!=null){  
     return br;  
     }  
    //第二步：从数据库中查找  
    Session s = currentSession();  
    try {  
     Transaction tx= s.beginTransaction();  
     br = (BaseEntity)s.get(beanClass, id);  
    tx.commit();  
}catch(HibernateException he){  
        throw he;  
}finally{  
        closeSession();  
}   
    //第三步：把对象放入memcached中  
    if(br!=null){  
     setObjectToMemcached(br);  
    }  
return br;  
}

传入的beanClass不同就实现了不同表的插入，delete和update方法类似，getList方法稍微复杂点。这样抽象出来，新增加表连DAO都不用写了，直接写一个bean、修改hibernate.cfg.xml、在数据库增加对应的表，就实现了所有常用的增删改查操作和分布式缓存了，还算简单。

6：事务

事务和缓存是一对矛盾，万一事务回滚了，缓存如何回滚也是一个需要处理的问题。如果缓存不回滚就会有问题，比如修改一个商品价格，缓存已经修改，最后数据库报错了没有修改成功，那么用户看到的价格已经修改，数据库还没改。所以为了做缓存回滚，我的BaseEntity实现了java.lang.Cloneable，在事务过程中，用本地线程记录BaseEntity变化前的状态，如果事务回滚，把所有事务中增删改过的BaseEntity缓存都清除一次。

7：表的垂直拆分和水平拆分

垂直拆分很简单，不同的bean有不同的连接池连接到不同数据库，这就实现了垂直拆分，水平拆分也不用想得太复杂，如“交易表”，直接按照“交易表”的userId最后N位拆分成M个表就行，合并结果集只有在非常的情况下才会用到。水平拆分可以这么做，Message继承BaseEntity，Message已经包含了所有字段，Message1再继承Message,Message2也继承Message，Message1和Message2里面什么都没有，就是用来获取不同连接池（库表）用的，所以程序不用做任何修改，取到的对象都是BaseEntity也都是Message。

8：缓存还是静态化

这个要看需求，对于一般对时效性要求比较高的页面一般不用静态化，对于那种很少修改的页面，可用静态化。静态化也有很多问题，和列表缓存一样，如果修改了对象，不知道哪些页面需要重新静态化一次。对于b2c这种网站，除了首页，其余的基本上不需要静态化，对于新闻网站，还是很有必要的。

9：最终的数据库、应用服务器、缓存服务器的构架，网状多对多结构。如下图：

