缓存是存取数据的临时地，因为取原始数据代价太大了，加了缓存，可以取得快些。缓存可以认为是原始数据的子集，它是从原始数据里复制出来的，并且为了能被取回，被加上了标志。   
  
在android开发中，经常要访问网络数据比如大量网络图片，如果每次需要同一张图片都去网络获取，这代价显然太大了。可以考虑设置本地文件缓存和内存缓存，存储从网络取得的数据；本地文件缓存空间并非是无限大的，容量越大读取效率越低，可设置一个折中缓存容量比如10M，如果缓存已满，我们需要采用合适的替换策略换掉一个已有的数据对象，并替之已一个新的数据对象；内存缓存作为最先被读取的数据，应该存储那些经常使用的数据对象，且内存容量有限，内存缓存的容量也应该限定。依照这样的做法，取得一个图片（总图片数为N）的流程应该是这样的：   
a.先在内存缓存取(设存储K个)，若取到则返回（命中率为K/N，时间为tA），否则进行b；   
b.在本地文件缓存（设能存储M个）中取，若取到则返回并更新内存缓存（命中率为（M-K）/N,时间为tB），否则进行c；   
c.通过网络下载图片，并更新本地文件缓存和内存缓存(命中率为(N-M)/N，时间为tC)；   
  
取一张图片的时间期望为：W = tA \* (K/N) + tB \* (M-K)/N + tC \* (N-M)/N ，其中tA < tB < tC ,为使W代价小，即尽可能快的取得数据，我们应该提高内存缓存的命中率和本地文件缓存的命中率，但两者的容量都是有限制的，所以必须使用适合替换算法来更新两者所存储的对象。选择合适的替换算法是缓存的难点所在。   
  
2、常见缓存交换算法介绍   
  
  
最理想的替换算法是每次调换出的数据对象是所有缓存中最迟将被使用的，这可以最大限度的推迟数据对象调换。可惜的是，我们无法预知用户的访问，所以这种算法是无法实现的。   
实际中常见的缓存算法有以下几种（参考文档a）：   
  
Least Frequently Used（LFU）   
对每个缓存对象计算他们被使用的频率。把最不常用的缓存对象换走。   
  
Least Recently User（LRU）   
把最近最少使用的缓存对象给换走。总是需要去了解在什么时候，用了哪个缓存对象。如果有人想要了解为什么总能把最近最少使用的对象踢掉，是非常困难的。浏览器就是使用了LRU作为缓存算法。新的对象会被放在缓存的顶部，当缓存达到了容量极限，我会把底部的对象踢走，而技巧就是：我会把最新被访问的缓存对象，放到缓存池的顶部。   
所以，经常被读取的缓存对象就会一直呆在缓存池中。可以用数据或者链表实现。其改进算法有LRU2 和 2Q。   
  
Least Recently Used 2（LRU2）   
把被两次访问过的对象放入缓存池，当缓存池满了之后，我会把有两次最少使用的缓存对象踢走。因为需要跟踪对象2次，访问负载就会随着缓存池的增加而增加。如果用在大容量的缓存池中，就会有问题。另外，还需跟踪那么不在缓存的对象，因为他们还没有被第二次读取。这比LRU好。   
  
Two Queues（2Q）   
把被访问的数据放到LRU的缓存中，如果该对象再一次被访问，就把他转移到第二个更大的LRU缓存。替换掉缓存对象是为了保持第一个缓存池是第二个缓存池的1/3。当缓存的访问负载是固定的时候，把 LRU 换成 LRU2，就比增加缓存的容量更好。这种机制使得该算法比 LRU2 更好。   
  
Adaptive Replacement Cache（ARC）   
这种算法介于 LRU 和 LFU 之间，由2个 LRU 组成，第一个，也就是 L1，包含的条目是最近只被使用过一次的，而第二个 LRU，也就是L2，包含的是最近被使用过两次的条目。因此，L1 放的是新的对象，而 L2 放的是常用的对象。该算法是是性能最好的缓存算法之一，能够自调，并且是低负载的。保存着历史对象，这样，就可以记住那些被移除的对象，同时，也可以看到被替换掉的对象是否可以留下，取而代之的是替换别的对象。该算法记忆力很差，但是很快，适用性也强。   
  
Most Recently Used（MRU）   
该算法与 LRU是对应的。它替换掉最近最多被使用的对象，你一定会问为什么。原因是，当一次访问过来的时候，有些事情是无法预测的，并且在缓存系统中找出最少最近使用的对象是一项时间复杂度非常高的运算。该算法在数据库内存缓存中很见！每当一次缓存记录的使用，会把它放到栈的顶端。当栈满了的时候，会把栈顶的对象给换成新进来的对象！   
  
First in First out（FIFO）   
这是一个低负载的算法，并且对缓存对象的管理要求不高。通过一个队列去跟踪所有的缓存对象，最近最常用的缓存对象放在后面，而更早的缓存对象放在前面，当缓存容量满时，排在前面的缓存对象会被踢走，然后把新的缓存对象加进去。很快，但是不适用。   
  
Second Chance   
改进的FIFO算法，比 FIFO 好的地方是改善了 FIFO 的成本。一样是在观察队列的前端，但是很FIFO的立刻替换不同，它会检查即将要被踢出的对象有没有之前被使用过的标志（1一个bit表示），如果没有被使用过，就把他换出；否则，把这个标志位清除，然后把这个缓存对象当做新增缓存对象加入队列。你可以想象就这就像一个环队列。当再一次在队头碰到这个对象时，由于它已经没有标志位，可以立刻就它换出。在速度上比FIFO快。   
  
CLock   
这是一个更好的FIFO，也比 second chance更好。因为它不会像second chance那样把有标志的缓存对象放到队列的尾部，但是也可以达到second chance的效果。它持有一个装有缓存对象的环形列表，头指针指向列表中最老的缓存对象。当缓存miss发生并且没有新的缓存空间时，它会根据指针指向的缓存对象的标志位去决定应该怎么做。如果标志是0，直接用新的缓存对象替代这个缓存对象；如果标志位是1，把头指针递增，然后重复这个过程，直到新的缓存对象能够被放入。   
  
Simple time-based   
通过绝对的时间周期去失效那些缓存对象。对于新增的对象，保存特定的时间。很快，但不适用。   
  
Extended time-based expiration   
通过相对时间去失效缓存对象的；对于新增的缓存对象，保存特定的时间，比如是每5分钟，每天的12点。   
  
Sliding time-based expiration   
被管理的缓存对象的生命起点是在这个缓存的最后被访问时间算起。很快，不太适用。   
  
缓存算法主要考虑到了下面几点：   
  
成本。如果缓存对象有不同的成本，应该把那些难以获得的对象保存下来。   
容量。如果缓存对象有不同的大小，应该把那些大的缓存对象清除，这样就可以让更多的小缓存对象进来了。  
时间。一些缓存还保存着缓存的过期时间。电脑会失效他们，因为他们已经过期了。 