Image Caching Scheme Based on Android

Xia Xiaoling and Wang Yunlin

***Abstract*—Loading and displaying of the images is one of the commonly used features of Android applications. Every time re-download images from the server not only affect the user experience, excessive amount of pictures but also easily lead to OOM (out of memory). So it is necessary to cache image for applications. Image caching scheme first to search the memory cache, if it returns no resource then go to search the local cache, if it still has no resource, then turn on asynchronous download to get the images. To different types of images, this scheme use different caching strategies. It also optimizes the memory cache and the local cache at the same time. Experimental results show that this scheme can effectively improve loading efficiency and user experience, and avoid the emergence of the OOM phenomenon of the application process.**

***Index Terms*—Android, cache, image, OOM.**

I. Introduction

With the popularity of mobile devices such as mobile phones and tablets, users' requirements for mobile applications is not only functional, but also have more and more demands in the field of custom requirements such as user experience [1], [2]. Since 2010 Google's Android platform is showing a momentum of rapid development and has a share soared to more than 50% from less than 10% in the global smart phone market. It seems to have seized the "half" [3]. Android is a truly open mobile device platform, which is composed of operating system, middle ware, user interface and application software. It is known as the first mobile software which is truly open and complete for mobile terminal [4], [5].

Displaying an image in a mobile application is one of the most common tasks for app developers. Nearly every application displays some kind of graphics. As reading the images from the network every time is not only a waste of time but also traffic and slow, affecting the user experience. If the images are downloaded to memory for the first time, although it can be called quickly, the size of a single image may be up to a few hundred K while memory of mobile devices is limited(usually 16M). if the image is too large, memory is often not enough and it will easily lead to OOM exceptions [6]. Surprisingly, it can be quite challenging to efficiently load and display an image on Android.

Based on the above contradictions, this paper presents a general Android platform based image caching scheme. The scheme not only uses different cache recovery strategies for images of different characteristics of the Android application, but also takes three caching strategies, optimizes memory and local cache. Images of the application display smoothly. It improves the user experience and is to avoid OOM at the same time.

The structure of this paper is as follows. Section I introduces the characteristics and default approach of images of Android. Section II focuses on the principle of the proposed image cache scheme. Section III describes the specific implementation of the scheme. The fourth quarter verifies the effectiveness of the scheme by experiments and practical applications. Finally, it concludes.

Manuscript received April 11, 2013; revised June 19, 2013.

The authors are with the Computer Science and Technology Institute of Donghua University, ShangHai 201620, China (e-mail: wangyl_7017@126.com).

II. The Characteristics and Default Approach of Images of Android

As an important user experience elements in Android applications, images have the following characteristics:

1) **As the skin of the application:** such images are generally embedded within the application, released together with the application.
2) **As permanent data resources:** such images can be embedded into the application, and can also be obtained from the network. Generally in order to reduce the size of the published application, they will be accessed from the network.
3) **As temporary data resources:** such images are obtained from the network.

The default handling of the images of Android is shown as below:

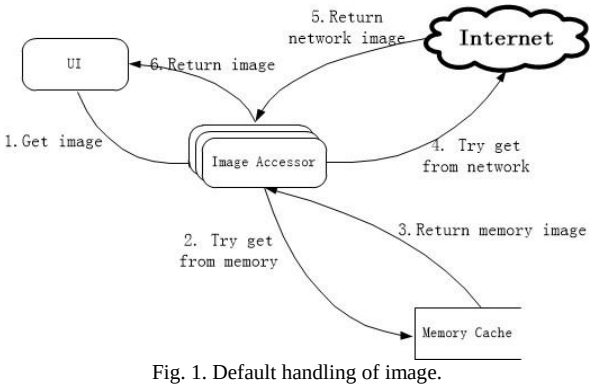

Fig. 1. Default handling of image.

III. Principles of Android-Based Image Cache Scheme

For the above defects of supporting network images in Android application , which makes the application waste a lot flows and time to download the repetitive images, this paper presents a more favorable image processing scheme.

Fig. 2 is the basic idea of the proposed scheme: When the application tries to request a network Image, it first attempts to access image from memory. If it is in memory cache then as shown in steps 1, 2, 3, 10, directly returns. If the image is not cached in memory or cached image is recovered by

DOI: 10.7763/IJIET.2013.V3.325

system, try to get the image from the local cache. From the Fig. 2. we can see, the application cannot obtain the required image from memory cache in step 2, it will turn to step 4, trying to get the image from local cache. If the local cache has the image then return it, storing it to the memory cache for the next fetch. As shown, perform steps 1, 2, 4, 5, 6, 10 to return the local cached image resources. If the local cache also has no such image or cached image is too old for our request, then try to access the network image. As Fig. 2 shown if the application cannot return the image in step 4, then jump to step 7, directly downloading the image via a network connection. If it is successful, encrypt it's version number by MD5, and together with the image are stored in the local cache, as well as in memory cache. As shown, the application will perform step 1, 2, 4, 7, 8, 9, 6, 10 to return the latest picture of the network resources.

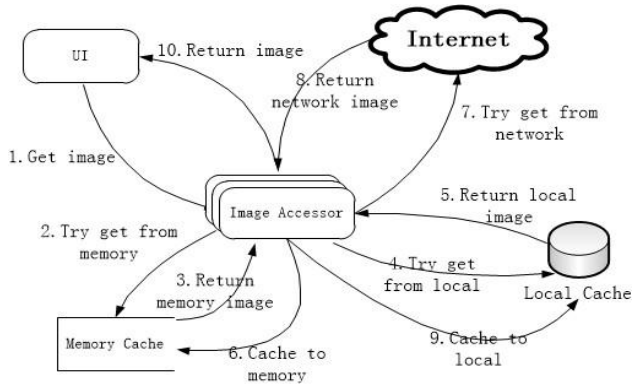

Fig. 2. Principles of image cache scheme.

According to the proposed scheme, when the application requires for image resources, it firstly tries to search the memory cache, if the memory is not cached then goes to the local cache, if it still has no resource, then downloads from the network. This strategy won't cause OOM and it ensures optimal efficiency at the same time. It greatly improves the the images' loading speed and user experience.

Besides, in terms of handling exceptions, if the application is OOM, it will trigger GC operating. System will recycle the memory resources which are not applicable, and then re-try to get images to ensure that the images presented to the user rightly.

Meanwhile, in the memory cache, this article provides a way to limit the size of memory cache. As long as the image cache size exceeds the limit, the program will deletes the long-unused images from memory as to minimize memory occupation.

In addition, for the image resource for local cache, programs of this article also provides a strategy to limit the size of the cache. if the local cache exceeds the setted maximum cache size, images of old version or low utilization rate will be removed according to certain strategy so that you can minimize the space occupied of disc.

For the previously mentioned problem of cached images may be out of date, the solution provided by this paper is a version management. When caching the images, application will store the resource with the version number of the image (generally last modified time of the image) with simple encryption. When getting local cached images it will decrypt first and then compare the version number with the requested, if it is not the images that are requested, the scheme will turn to network to obtain the latest picture. So that both the update of images and prevention of image stolen are guaranteed.

IV. Specific Implementation of the Scheme

When application starts, it will start the downloader according to the stetted image type [7]. For example, if the application is only configured skin image type downloader, the application will only start this downloader, and then accompanies it according to the supplied configuration. After the downloader starts, then waiting for the users to request a image.

When user requests a image, the application will obtain a task from the task cache or re-generate a unique download task *ImageTask* based on each URL of the image. Before starting the task, the program will call the callback event *ImageFetchingCallback.onFetchingStarted (ImageTask)* to notice the client that task has begun, and then check memory cache according to *ImageTask*. If the image has been cached in memory, directly fetch it from memory and call the callback event *ImageFetchingCallback.onFetchingCompleted (ImageTask, Bitmap)* to notify the client that image has been obtained, and then ends this mission.

If it does not find the requested image in the memory cache, the program will first retrieve the local cache directory. If the local cache contains the requested resources, *ImageTask* will be added to the local download queue *ImageTaskQueue*, and then notify the local downloader *CachedPoolExecutor* that a new task has been added, otherwise it will be added to the network download task queue. For local downloader *CachedPoolExecutor*, when it receives the task it will start a worker thread *ImageTaskWorker*. Worker thread will first retrieve the task queue, obtaining the highest priority task according to *ImageTaskContext*, and then start to get a local image. If it is failed to obtain local image then task will be canceled and added to network image queue. If it causes OOM when obtaining, the program will intercept the OOM error, then handle OOM according to the configured *OutOfMemoryAttempt*. Default handling is not going to clean up the memory cache list, but merely calling *System.gc ()* to trigger system recovery. For network downloader *RemotePoolExecutor*, the process is essentially the same with *CachedPoolExecutor*, the only difference is just when the network image obtaining is failed at last, callback event *ImageTaskHandler.notifyFetchingFailed* is invoked to tell the client image fetching failure. If success image resources will be stored to the local cache and memory cache.

A. Task Queue

Task queue *ImageTaskQueue* is a list of *ImageTask* whose main purpose is to obtain the priority task. Worker thread *ImageTaskWorker* which is access to local or network image will call task list *ImageTaskQueue.dequeue ()* (dequeue contains the main access algorithm) to get performed *ImageTask*. Each time of obtaining *ImageTask*, *ImagetTaskQueue* will traverse all of the tasks to be performed currently, and then determine the priority of the task by *ImagetTaskContext* of each task.

B. OOM Handling

If it causes OOM when obtaining a image, the program will intercept the error, then handle OOM according to the configured *OutOfMemoryAttempt*. OOM will be handled in the following ways:

1) ATTEMPT_END: only perform *System.gc ()*, do not clean up the images of the memory cache or re-try to get images.
2) ATTEMPT_CLEAN: clean up the the memory cache, call *System.gc ()* but do not re-try to get image resources.
3) ATTEMPT_BEGIN: clean up the memory cache, call *System.gc ()* then re-try to obtain the image resources every 2 seconds .

C. Image Version Management

Each image resource file in local cache is divided into two parts, header information *ImageHeader* and the image resource. Header information includes the version number and the last modification time of the image. Each of the header information occupy 12 bytes, 4 bytes used to keep the version and 8 byte to save the *lastModified*.

When download is complete, image resources will be stored to the local cache. If version management is needed, the version number of the image will be stored in the header information.

When reading for image resource in a local cache with version management, the program first reads the header information, the first 12 bytes of the file. If the version number equals *ImageTask.version*, read the rest image resource, and then returns the obtained resource.

D. Memory Cache Strategies

Mentioned solutions for the memory cache management, this paper provides the following management strategies:

1) *WeakMemoryCache*: cache without any limitation.
2) *LargestLimitedMemoryCache*: limit the size of the cache, and to recover the largest resource when the cache reaches the maximum size limit .
3) *FIFOLimitedMemoryCache*: limit the size of the cache, when the cache reaches the maximum size limit it will be recovered by the first-in, first-out strategy.

E. Local Cache Strategies

Mentioned solutions for the local cache management, this paper provides the following management strategies:

1) *UnlimitedDiscCache*: without limiting the size of the local cache, unlimited occupation of Disc space.
2) *FileCountLimitedDiscCache*: limit the number of cache files, the most-recently-used image file will be delete when cached files are more than the set maximum number.
3) *CacheSizeLimitedDiscCache*: limit the size of the cache directory, if the number of files in the cache exceeds the set maximum number, the most-recently-used image files will be deleted.

V. Experiment and Analysis

The main purpose of the image caching scheme proposed in this paper is to accelerate the speed of loading, so this section is to illustrate the advantages of our method by contrast the loading time of the same number of pictures.

Besides our method *ImageFetcher*, we also achieved a method called *BitMap* which do not cache images, for every loading go directly to the server-side to download resources, and caching scheme proposed by [8] called *AsyncImageLoader*. With the same conditions of network and mobile devices, we use the three ways to load the same number and size of images. By measuring the load time, we illustrate the efficiency of the various options. The experimental results are shown as Fig. 3:

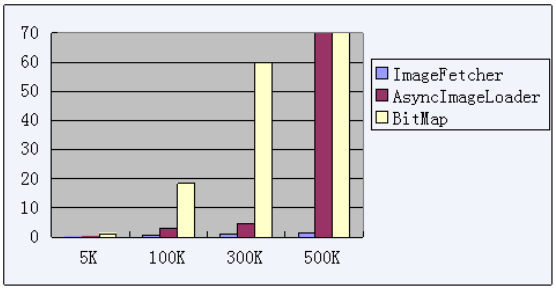

Fig. 3. Loaded 50 pictures, the time required by the three methods.

The abscissa of the Fig. 3 above shows the size of the images, and the vertical axis shows the average time required by the load (experiment was repeated 20 times, an average). As we can see in the case of small images(5K), the efficiency of the three methods has no significant difference, especially *ImageFetcher* and *AsyncImageLoader*, both do cache, the loading speed is very fast, the time nearly can be ignored. As the images grow larger, time overhead of *AsyncImageLoader* slowly increases, while our *ImageFetcher* is essentially the same. As *BitMap* re-request resources from the server and load each time, it causes a sharp increase in time consumption. When the size of images reach 500k, *AsyncImageLoader* and *BitMap* both cause OOM, time consumption in the Fig. 3 is infinite. The reason for *AsyncImageLoader* is that it does no optimization for memory cache and images cached in the memory beyond the memory capacity, while for *BitMap* the reason is images downloaded to the memory are too big to cause a memory leak. Using our *ImageFetcher* method do not have to worry about this situation. Unified memory cache management can not only maximize the reuse of images to speed up the loading speed, but also avoid memory overflow problems.

In actual Android application testing we also found that using *BitMap* to loaded 50 images to a ListView, the application will become very card and user experience is very bad. While applications using *ImageFetcher* or *AsyncImageLoader* are very smooth, particularly applications using *ImageFetcher*, user experience is very good. Application downloaded and displayed hundreds of pictures is amazingly smooth, not even with a slightest pause. Also the memory remains under control; there is no occurrence of OOM.

Lot of practice has proved that the image cache scheme proposed in this paper can not only effectively improve the efficiency of loading, but also with reliable stability. Fig. 4 shows a sports application with this scheme , the application has been formally launched in the Google Market, and has good user feedback.

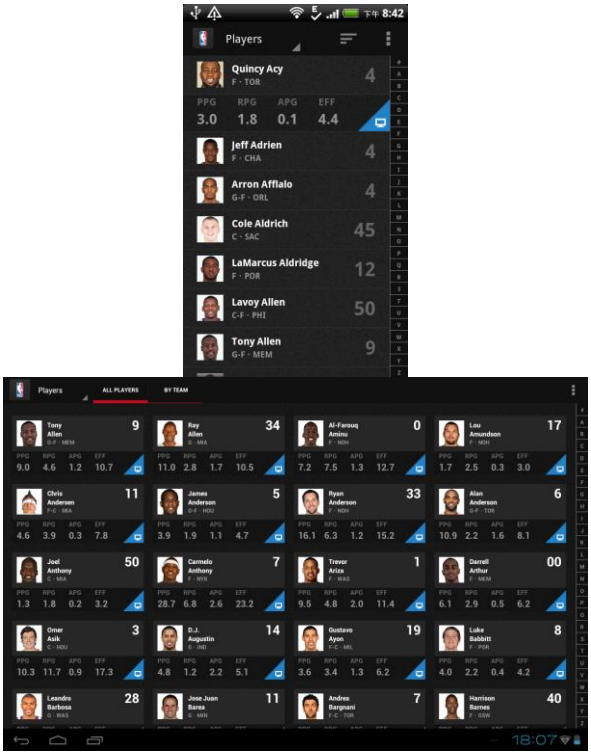

Fig. 4. Android application interface for a sporting event.

VI. Conclusion

When Android applications loading and displaying images, for every request if they re-download images from the server not only affect the user experience, but also easily lead to OOM .This paper presents an Android image caching scheme, using asynchronous thread to download image resources and cache them in both memory and local. When obtaining resources, it would read the memory first then local cache before network download. The scheme uses different caching strategies for images of different characteristics. Verified by experiments and real applications, the scheme effectively improves the efficiency of image displaying and user experience, and optimizes the caches, avoiding OOM at the same time.

References

[1] F. Lu, "Research of development trend of mobile intelligent terminal," *Modern Media*, 2011, vol. 11, pp. 139-140.
[2] Z. Z. Gong and Z. L. Lu, "Analysis and research of the android development," *Value Engineering*, 2013, vol. 2, pp. 185-186.
[3] X. Y. Liu, "Mobile terminal open platform-Android," *Information and Communication Technology*.2011, vol. 4, pp. 50-52.
[4] M. Yao and W. G. Liu, "Research of Android architecture and application development," *Computer Systems & Applications*, 2008, vol. 11, pp. 110-112.
[5] Y. H. Xu and C. Y. Xiong, "Android mobile development optimization strategy," *Computer age*, 2011, vol. 12, pp. 23-24.
[6] RyanHoo. (Dec. 01, 2012). Build Android cache module. [Online]. Available: http://my.oschina.net/ryanhoo/blog/93285.
[7] J. Song, L. C. Dang, Z. C Guo, and M. Zhao, "The security mechanism analysis and applied research of Android OS mobile platform," *Computer Technology and Development*, 2010, vol. 6, pp. 31-33.
[8] H. B. Li and M. Ma, "Research and design of the multi-threaded downloader based on Android," *Information & Communications*, 2012, vol. 28.

Xia Xiaoling is an associate professor. She received her Ph.D. in July 1994 at the Shanghai Jiao Tong University, image processing and pattern recognition. Her main research directions are image processing and data visualization. Now she is an associate professor of Computer Software and Theory Department of Computer Science and Technology Institute of Donghua University, master tutor.

Prof. Xia has received Sangma Award in 2000, and was awarded the title of Donghua University "eight" in 2008. She participated in the international engineering Computer Professional Training Mode Reform and Shanghai teaching achievement in 2009 second prize, and also participated in the construction of the 2008 Shanghai courses and key course "principles of Database Systems".

Wang Yunlin is a master. Now she is studing at Computer Science and Technology Institute of Donghua University. Her main research direction is new media.