前言

在Android开发中，ImageLoader应该算得上是最重要的开源库之一，由于项目原因（不能使用开源库），前段时间自己也是需要实现一个简单的ImageLoader，因此诞生了这个库，我们暂且叫它为SimpleImageLoader。就目前而言，你上网查ImageLoader资料的时候，基本上能够找到很简单的实现，基本上一个类就把所有的工作给做了，这就显得很不专业了嘛，很多时候我们不只是需要实现功能，而是希望能够在实现功能的同时在设计层面有所提升。

SimpleImageLoader分享出来的主要目的并不是说替代那些著名开源库，而是提供一个简单的、又有一定参考价值的ImageLoader实现让一些需要帮助的人学习，在深入了解实现的同时学到知识，也能够体会到在设计一个开源库时应该要做哪些考虑、做哪些取舍、有什么模式，当然在了解了ImageLoader的实现之后再去使用专业的开源库也会更加的得心应手，出现问题的时候自己也能够不太费力地去究其原因。在提升自己的同时也能够了解一些开源库的设计基本原则，这也是我的博客中一直主张的观点。

当然限于本人水平有限，有bug在所难免，但是我们这里是以学习为目的，了解ImageLoader的设计与实现才是我们最重要的目的，一些细节不必在意，可以在你深入的学习过程中修改你认为不合理或者错误的地方。如果你有好的实现方法或者有好的建议，也都请指教一二；如果你认为我写的东西烂得不值一提，那你就深深地埋藏在心里吧。

基本架构

一般来说，ImageLoader的实现都是基于线程池，在第一版的我也是使用线程池来加载图片，但是后面的版本却换成了跟SimpleNet类似的架构模式，原因是觉得线程池刚启动的时候会稍微慢一些，我感觉不太爽就换了线程模型。当然我也会把线程池的版本在另外的分支上给出，这样给一些需要的朋友参考。

SimpleImageLoader的基本结构如图1所示。

SimpleImageLoader工程结构图

到这幅图看过[SimpleNet网络框架](http://blog.csdn.net/column/details/simple-net.html" \t "http://blog.csdn.net/bboyfeiyu/article/details/_blank)的朋友应该是会熟悉一些，基本结构与SimpleNet很相似，其实主要也是我们比较喜欢把架构图画成这种分层样式，感觉比较好理解，

SimpleImageLoader类是用户的入口，用户在通过配置类初始化SimpleImageLoader之后就可以向SimpleImageLoader提交加载图片的请求了。SimpleImageLoader内部维护了一个请求队列，用户提交的加载图片的请求会在内部被封装成BitmapRequest对象，然后将这些对象放到请求队列中。在创建队列时会创建用户指定数量（默认为CPU个数 + 1）的线程来加载图片，这些线程在内部命名为RequestDispatcher，它们在run函数中不断地获取队列中的加载请求，然后交给对应的Loader加载图片。

为了方便用户的扩展，我们引入了Loader这个抽象，因为在SimpleImageLoader中只支持两种图片uri的加载，即网络图片uri和本地文件的uri。网络图片一般以"http://"或者"https://"开头，而本地图片的uri格式却是"file://"开头，SimpleImageLoader内部通过图片uri的格式的不同使用不同的Loader来加载图片，这样后续用户就可以注册Loader来实现其他格式的加载，例如"drawable:// + 图片名"来加载res/drawable中的图片等。这样保证了SimpleImageLoader可加载图片uri格式的可扩展性。Loader会通过LoaderManager来进行管理，如果需要注册自己的Loader实现，则调用LoaderManager的register函数即可。如果你传递进去的图片uri是无效，例如格式错误，那么LoaderManager会返回一个默认的Loader，这个默认的Loader名为NullLoader，它其实什么也不做，只是为了防止在外部进行判空而已，这种模式成为Null Object设计模式。当然，在加载图片之前我们会从缓存中读取，如果有缓存我们则不加载。

Loader加载完图片之后会先更新UI，即将图片显示到对应的ImageView上，在构造BitmapRequest时内部已经将图片的uri设置为ImageView的tag了。图片加载完成后判断ImageView的tag和uri是否相等，如果相等则将图片显示到ImageView上，否则不更新ImageView。这一步很重要，很多朋友在使用ImageLoader时出现问题基本上就是由于没有设置ImageView的tag。

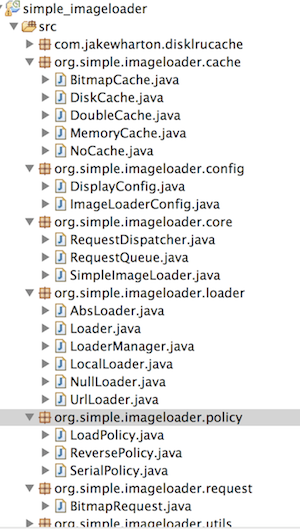
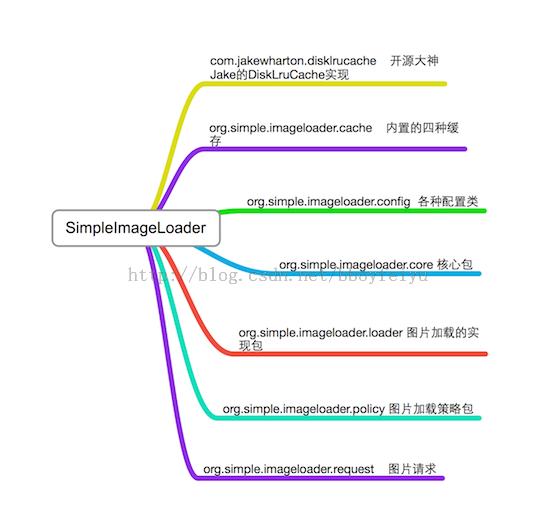
加载图片的先后顺序是由加载策略决定的，策略相关的内容没有在架构图中给出。加载策略决定了请求在队列中的排序，在将请求添加到队列中时会给每个请求设置一个序号，队列将根据这个序号对请求进行排序。这样我们就可以知道哪个请求是先添加进来的，也可以很方便的实现自己的策略类来定制自己的加载策略，比如最后加载到队列中的请求最先加载。比如我们在ListView滚动时，最后添加到队列中的图片请求应该是我们最急需显示的，我们它们就在手机的当前屏幕，而前面的请求对应的ImageView已经被复用，即使它们加载完成它们也不会被显示，因为ImageView的tag已经变化了。因此，策略的灵活性依然很重要。

在加载完图片并且更新UI之后，我们会将图片缓存起来。内置的缓存类型有四种，无缓存、内存缓存、sd卡缓存、内存和sd卡的双缓存，这四种缓存都实现了Cache接口，如果你这四种缓存类型还不能满足你的需求，那么你可以实现Cache接口，然后实现自己的缓存逻辑，然后在配置ImageLoader时设置需要的缓存类型（具体配置后续说明），如果不配置则默认使用的是内存缓存。这里我们又看到了一个面向接口编程的例子，即SimpleImageLoader只依赖于Cache接口的抽象，而不是说依赖于某个具体的缓存类，这样用户就可以很方面的实现自己的缓存逻辑，并且将缓存实现注入到sdk中。当然，上述的Loader、加载策略实现也是基于同样的理论基础，就是说过很多遍的“面向接口编程”。

恩，是时候捋一捋这个执行流程了。

用户调用displayImage请求加载图片，SimpleImageLoader将这个加载图片请求封装成一个Request，然后加入到队列中。几个色眯眯的调度子线程不断地从队列中获取请求，然后根据uri的格式获取到对应的Loader来加载图片。在加载图片之前首先会查看缓存中是否含有目标图片（具体细节在后续的博客再细说），如果有缓存则使用缓存，否则加载目标图片。获取到图片之后，我们会将图片投递给ImageView进行更新，如果该ImageView的tag与图片的uri是一样的，那么则更新ImageView，否则不处理。使用ImageView的tag与图片的uri进行对比是为了防止图片错位显示的问题，这在ImageLoader中是很重要的一步。如果目标图片没有缓存，第一次从uri中加载后会加入缓存中，当然从sdcard中加载的图片我们只会缓存到内存中，而不会再缓存一份到sd卡的另一个目录中。这样，整个加载过程也就完成了。

SimpleImageLoader工程结构图

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

在教你写Android ImageLoader框架之基本架构中我们对SimpleImageLoader框架进行了基本的介绍，今天我们就从源码的角度来剖析ImageLoader的设计与实现。

在我们使用ImageLoader前都会通过一个配置类来设置一些基本的东西，比如加载中的图片、加载失败的图片、缓存策略等等，SimpleImageLoader的设计也是如此。配置类这个比较简单，我们直接看源码吧。

ImageLoaderConfig配置



都是很简单的setter函数，但是不太一样的是这些setter都是返回一个ImageLoaderConfig对象的，在这里也就是返回了自身。这个设计是类似Builder模式的，便于用户的链式调用，例如:

[java] view plaincopy在CODE上查看代码片派生到我的代码片

01.private void initImageLoader() {

02. ImageLoaderConfig config = new ImageLoaderConfig()

03. .setLoadingPlaceholder(R.drawable.loading)

04. .setNotFoundPlaceholder(R.drawable.not\_found)

05. .setCache(new DoubleCache(this))

06. .setThreadCount(4)

07. .setLoadPolicy(new ReversePolicy());

08. // 初始化

09. SimpleImageLoader.getInstance().init(config);

10. }

对于Builder模式不太了解的同学可以参考 Android源码分析之Builder模式(http://blog.csdn.net/bboyfeiyu/article/details/24375481)。构建好配置对象之后我们就可以通过这个配置对象来初始化SimpleImageLoader了。SimpleImageLoader会根据配置对象来初始化一些内部策略，例如缓存策略、线程数量等。调用init方法后整个ImageLoader就正式启动了。



从上述代码中我们可以看到SimpleImageLoader的工作比较少，也比较简单。它就是根据用户传递进来的配置来初始化ImageLoader,并且作为图片加载入口，用户调用displayImage之后它会将这个调用封装成一个BitmapRequest请求,然后将该请求添加到请求队列中。

BitmapRequest图片加载请求

BitmapRequest只是一个存储了ImageView、图片uri、DisplayConfig以及ImageListener的一个对象，封装这个对象的目的在加载图片时减少参数的个数，\*\*\*在BitmapRequest的构造函数中我们会将图片uri设置为ImageView的tag，这样做的目的是防止图片错位显示。\*\*\*BitmapRequest类实现了Compare接口，请求队列会根据它的序列号进行排序，排序策略用户也可以通过配置类来设置，具体细节在加载策略的章节我们再聊吧。



RequestQueue图片请求队列

请求队列我们采用了SImpleNet中一样的模式，通过封装一个优先级队列来维持图片加载队列，mSerialNumGenerator会给每一个请求分配一个序列号，PriorityBlockingQueue会根据BitmapRequest的compare策略来决定BitmapRequest的顺序。RequestQueue内部会启动用户指定数量的线程来从请求队列中读取请求，分发线程不断地从队列中读取请求，然后进行图片加载处理，这样ImageLoader就happy起来了。



RequestDispatcher请求分发

请求Dispatcher,继承自Thread,从网络请求队列中循环读取请求并且执行，也比较简单，直接上源码吧。



第一个重点就是run函数了，不断地从队列中获取请求，然后解析到图片uri的schema，从schema的格式就可以知道它是存储在哪里的图片。例如网络图片对象的schema是http或者https，sd卡存储的图片对应的schema为file。根据schema我们从LoaderManager中获取对应的Loader来加载图片，这个设计保证了SimpleImageLoader可加载图片类型的可扩展性，这就是为什么会增加loader这个包的原因。用户只需要根据uri的格式来构造图片uri，并且实现自己的Loader类，然后将Loader对象注入到LoaderManager即可，后续我们会再详细说明。

## 这里的另一个重点是parseSchema函数，它的职责是解析图片uri的格式,uri格式为: schema:// + 图片路径,例如网络图片的格式为http://xxx.image.jpg,而本地图片的uri为file:///sdcard/xxx/image.jpg。如果你要实现自己的Loader来加载特定的格式，那么它的uri格式必须以schema://开头，否则解析会错误，例如可以为drawable://image,然后你注册一个schema为"drawable"的Loader到LoaderManager中，SimpleImageLoader在加载图片时就会使用你注册的Loader来加载图片，这样就可以应对用户的多种多样的需求。如果不能拥抱变化那就不能称之为框架了，应该叫功能模块。

## 本章总结

   最后我们来整理一下这个过程吧，SimpleImageLoader根据用户的配置来配置、启动请求队列，请求队列又会根据用户配置的线程数量来启动几个分发线程。这几个线程不断地从请求队列（线程安全）中读取图片加载请求，并且执行图片加载请求。这些请求是用户通过调用SimpleImageLoader的displayImage函数而产生的，内部会把这个调用封装成一个BitmapRequest对象，并且将该对象添加到请求队列中。这样就有了生产者（用户）和消费者（分发线程），整个SimpleImageLoader就随着CPU跳动而热血沸腾起来了！

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

在教你写Android ImageLoader框架之初始配置与请求调度中，我们已经讲述了ImageLoader的请求配置与调度相关的设计与实现。今天我们就来深入了解图片的具体加载过程以及加载的策略(包括按顺序加载和逆序加载) ，在这其中我会分享我的一些设计决策，也欢迎大家给我提建议。

图片的加载

Loader与LoaderManager的实现

在上一篇文章教你写Android ImageLoader框架之初始配置与请求调度中，我们聊到了Loader与LoaderManager。 ImageLoader不断地从队列中获取请求，然后解析到图片uri的schema，从schema的格式就可以知道它是存储在哪里的图片。例如网络图片对象的schema是http或者https，sd卡存储的图片对应的schema为file，schemae与Loader有一个对应关系。根据schema我们从LoaderManager中获取对应的Loader来加载图片。这个设计保证了SimpleImageLoader可加载图片类型的可扩展性，这就是为什么会增加loader这个包的原因。用户只需要根据uri的格式来构造图片uri，并且实现自己的Loader类，然后将Loader对象注入到LoaderManager即可。RequestDispatcher中的run函数如下 ：

@Override

public void run() {

try {

while (!this.isInterrupted()) {

final BitmapRequest request = mRequestQueue.take();

if (request.isCancel) {

continue;

}

final String schema = parseSchema(request.imageUri);

// 根据schema获取loader

Loader imageLoader = LoaderManager.getInstance().getLoader(schema);

imageLoader.loadImage(request);

}

} catch (InterruptedException e) {

Log.i("", "### 请求分发器退出");

}

}

Loader只定义了一个接口，只用一个加载图片的方法。

public interface Loader {

public void loadImage(BitmapRequest result);

}

抽象是为了可扩展，定义这个接口，我们就可以注入自己的图片加载实现类。例如从资源、assets中加载。不管从网络还是本地加载图片，我们加载图片的过程有如下几个步骤:

1.判断缓存中是否含有该图片;

2.如果有则将图片直接投递到UI线程，并且更新UI；

3.如果没有缓存，则从对应的地方获取到图片，并且将图片缓存起来，然后再将结果投递给UI线程，更新UI；

我们可以发现，不管从哪里加载图片，这些逻辑都是通用的，因此我抽象了一个AbsLoader类。它将这几个过程抽象起来，只将变化的部分交给子类处理，就相当于AbsLoader封装了一个逻辑框架( 可以思考用了什么设计模式)，大致代码如下 :

/\*\*

\* @author mrsimple

\*/

public abstract class AbsLoader implements Loader {

/\*\*

\* 图片缓存

\*/

private static BitmapCache mCache = SimpleImageLoader.getInstance().getConfig().bitmapCache;

@Override

public final void loadImage(BitmapRequest request) {

// 1、从缓存中获取

Bitmap resultBitmap = mCache.get(request);

Log.e("", "### 是否有缓存 : " + resultBitmap + ", uri = " + request.imageUri);

if (resultBitmap == null) {

showLoading(request);

// 2、没有缓存，调用onLoaderImage加载图片

resultBitmap = onLoadImage(request);

// 3、缓存图片

cacheBitmap(request, resultBitmap);

} else {

request.justCacheInMem = true;

}

// 4、将结果投递到UI线程

deliveryToUIThread(request, resultBitmap);

}

/\*\* 加载图片的hook方法，留给子类处理

\* @param request

\* @return

\*/

protected abstract Bitmap onLoadImage(BitmapRequest request);

// 代码省略

}

代码逻辑如上所述实现了一个模板函数，变化的部分就是onLoadImage，子类在这里实现真正的加载图片的方法。比如从网络上加载图片。



在初始化ImageLoader时我们会默认将几个Loader注入到LoaderManager中，然后在加载图片时ImageLoader会根据图片的schema来获取对应Loader来完成加载功能。

/\*\*

\*

\*/

private LoaderManager() {

register(HTTP, new UrlLoader());

register(HTTPS, new UrlLoader());

register(FILE, new LocalLoader());

}

加载策略

加载策略就是你的图片加载请求提交以后ImageLoader按照一个什么规则来加载你的请求。默认就是SerialPolicy策略(FIFO)，谁在队列前面就是谁优先被执行。但是事情往往没有那么简单，我们在ListView滚动时，我们希望最后添加到请求队列的图片优先得了加载，因此此时它们就在手机屏幕上，所以我们又添加了一个ReversePolicy策略。咦，对于这种存在各种可能性的部分，我们最不能具体化，还是要抽象！于是我定义了LoadPolicy接口，它的作用是compare两个请求，以此来规定排序原则。

public interface LoadPolicy {

public int compare(BitmapRequest request1, BitmapRequest request2);

}1

2

3

因为我们的请求队列使用的是优先级队列PriorityBlockingQueue，因此我们的BitmapRequest都实现了 Comparable 接口，我们在BitmapRequest的函数中将compareTo委托给LoadPolicy对象的compare。

@Override

public int compareTo(BitmapRequest another) {

return mLoadPolicy.compare(this, another);

}

我们看看默认的加载策略，即按顺序加载，先添加到队列的请求先被执行。

/\*\*

\* 顺序加载策略

\*

\* @author mrsimple

\*/

public class SerialPolicy implements LoadPolicy {

@Override

public int compare(BitmapRequest request1, BitmapRequest request2) {

// 那么按照添加到队列的序列号顺序来执行

return request1.serialNum - request2.serialNum;

}

}

逆序加载则为 :

/\*\*

\* 逆序加载策略,即从最后加入队列的请求进行加载

\*

\* @author mrsimple

\*/

public class ReversePolicy implements LoadPolicy {

@Override

public int compare(BitmapRequest request1, BitmapRequest request2) {

// 注意Bitmap请求要先执行最晚加入队列的请求,ImageLoader的策略

return request2.serialNum - request1.serialNum;

}

}

呵，想想这不是策略模式么！原来模式无处不在，当你习惯之后你就会发现模式在无形之中已经运用到你的代码了。如上所示，策略都是简单的实现，这个策略只需要在配置ImageLoader时指定就行了，用户也可以根据自己的需求来实现策略类，并且注入给ImageLoader。这样就保证了灵活性、可扩展性。

总结

通过Loader和LoaderManager保证了可加载图片来源的扩展性，即图片可以存储在网络上、sd卡中、res文件夹中等等，实现一个从特定位置加载图片的Loader,然后给这个Loader注册一个schema，在加载图片的时候根据图片的路径获取schema,再通过schema获取Loader,通过Loader加载图片。

而图片的加载策略又通过LoadPolicy这个抽象来定制，用户可以自行实现加载策略。这样就保证了灵活性，当然还有后期的图片缓存也是需要同样的灵活性。和我在公共技术点之面向对象六大原则所说，面向对象的几大原则最终化为几个简单的关键字: : 抽象、单一职责、最小化。领悟到了这些思想，我想你的代码质量应该会有一个质的提升。

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

我们从基本架构到具体实现已经更新了大部分的内容。今天，我们来讲最后一个关键点，即图片的缓存。为了用户体验，通常情况下我们都会将已经下载的图片缓存起来，一般来说内存和本地都会有图片缓存。那既然是框架，必然需要有很好的定制性，这让我们又自然而然的想到了抽象。下面我们就一起来看看缓存的实现吧。

缓存接口

在教你写Android ImageLoader框架之图片加载与加载策略我们聊到了Loader，然后阐述了AbsLoader的基本逻辑，其中就有图片缓存。因此AbsLoader中必然含有缓存对象的引用。我们看看相关代码:

/\*\*

\* @author mrsimple

\*/

public abstract class AbsLoader implements Loader {

/\*\*

\* 图片缓存

\*/

private static BitmapCache mCache = SimpleImageLoader.getInstance().getConfig().bitmapCache;

// 代码省略

}1

2

3

4

5

6

7

8

9

10

11

12

AbsLoader中定义了一个static的BitmapCache对象，这个就是图片缓存对象。那为什么是static呢？因为不管Loader有多少个，缓存对象都应该是共享的，也就是缓存只有一份。说了那么多，那我们先来了解一下BitmapCache吧。

public interface BitmapCache {

public Bitmap get(BitmapRequest key);

public void put(BitmapRequest key, Bitmap value);

public void remove(BitmapRequest key);

}1

2

3

4

5

6

7

8

9

BitmapCache很简单，只声明了获取、添加、移除三个方法来操作图片缓存。这里有依赖了一个BitmapRequest类，这个类代表了一个图片加载请求，该类中有该请求对应的ImageView、图片uri、显示Config等属性。在缓存这块我们主要要使用图片的uri来检索缓存中是否含有该图片，缓存以图片的uri为key,Bitmap为value来关联存储。另外需要BitmapRequest的ImageView宽度和高度，以此来按尺寸加载图片。

定义BitmapCache接口还是为了可扩展性，面向接口的编程的理念又再一次的浮现在你面前。如果是你，你会作何设计呢？自己写代码来练习一下吧，看看自己作何考虑，如果实现，这样你才会从中有更深的领悟。

内存缓存

既然是框架，那就需要接受用户各种各样的需求。但通常来说框架会有一些默认的实现，对于图片缓存来说内存缓存就其中的一个默认实现，它会将已经加载的图片缓存到内存中，大大地提升图片重复加载的速度。内存缓存我们的策略是使用LRU算法，直接使用了support.v4中的LruCache类，相关代码如下。



sd卡缓存

对于图片缓存，内存缓存是不够的，更多的需要是将图片缓存到sd卡中，这样用户在下次进入app时可以直接从本地加载图片，避免重复地从网络上读取图片数据，即耗流量，用户体验又不好。sd卡缓存我们使用了Jake Wharton的DiskLruCache类，我们的sd卡缓存类为DiskCache,代码如下 :



代码比较简单，也就是实现BitmapCache，然后包装一下DiskLruCache类的方法实现图片文件的增加、删除、获取方法。这里给大家介绍一个类，是我为了简化图片按ImageView尺寸加载的辅助类，即BitmapDecoder。

BitmapDecoder

BitmapDecoder是一个按ImageView尺寸加载图片的辅助类，一般我加载图片的过程是这样的:

1. 创建BitmapFactory.Options options,设置options.inJustDecodeBounds = true，使得只解析图片尺寸等信息;

2. 根据ImageView的尺寸来检查是否需要缩小要加载的图片以及计算缩放比例;

3. 设置options.inJustDecodeBounds = false,然后按照options设置的缩小比例来加载图片.

BitmapDecoder类使用decodeBitmap方法封装了这个过程 ( 模板方法噢 ),用户只需要实现一个子类，并且覆写BitmapDecoder的decodeBitmapWithOption实现图片加载即可完成这个过程(参考DiskCache中的get方法)。代码如下 :



在decodeBitmap中，我们首先创建BitmapFactory.Options对象,并且设置options.inJustDecodeBounds = true，然后第一次调用decodeBitmapWithOption(options)，使得只解析图片尺寸等信息;然后调用calculateInSmall方法，该方法会调用computeInSmallSize来根据ImageView的尺寸来检查是否需要缩小要加载的图片以及计算缩放比例，在calculateInSmall方法的最后将 options.inJustDecodeBounds = false，使得下次再次decodeBitmapWithOption(options)时会加载图片；那最后一步必然就是调用decodeBitmapWithOption(options)啦，这样图片就会按照按照options设置的缩小比例来加载图片了。

我们使用这个辅助类封装了这个麻烦、重复的过程，在一定程度上简化了代码，也使得代码的可复用性更高，也是模板方法模式的一个较好的示例。

## 二级缓存

有了内存和sd卡缓存，其实这还不够。我们的需求很可能就是这个缓存会同时有内存和sd卡缓存，这样上述两种缓存的优点我们就会具备，这里我们把它称为二级缓存。看看代码吧，也很简单。



其实就是封装了内存缓存和sd卡缓存的相关操作嘛~ 那我就不要再费口舌了

自定义缓存

缓存是有很多实现策略的，既然我们要可扩展性，那就要允许用户注入自己的缓存实现。只要你实现BitmapCache，就可以将它通过ImageLoaderConfig注入到ImageLoader内部。

private void initImageLoader() {

ImageLoaderConfig config = new ImageLoaderConfig()

.setLoadingPlaceholder(R.drawable.loading)

.setNotFoundPlaceholder(R.drawable.not\_found)

.setCache(new MyCache())

// 初始化

SimpleImageLoader.getInstance().init(config);

}1

2

3

4

5

6

7

8

MyCache.java

// 自定义缓存实现类

public class MyCache implements BitmapCache {

// 代码

@Override

public Bitmap get(BitmapRequest key) {

// 你的代码

}

@Override

public void put(BitmapRequest key, Bitmap value) {

// 你的代码

}

@Override

public void remove(BitmapRequest key) {

// 你的代码

}

}

Github地址

SimpleImageLoader。

总结

ImageLoader系列到这里就算结束了，我们从基本架构、具体实现、设计上面详细的阐述了一个简单、可扩展性较好的ImageLoader实现过程，希望大家看完这个系列之后能够自己去实现一遍，这样你会发现一些具体的问题，领悟能够更加的深刻。如果你在看这系列博客的过程中，真的能够从中体会到面向对象的基本原则、设计思考等东西，而不是说”我擦，我又找到了一个可以copy来用的ImageLoader”,那我就觉得我做的这些分享到达目的了。