> 移动开发本质上就是手机和服务器之间进行通信，需要从服务端获取数据。反复通过网络获取数据是比较耗时的，特别是访问比较多的时候，会极大影响了性能，Android中可通过缓存机制来减少频繁的网络操作，减少流量、提升性能。

实现原理

把不需要实时更新的数据缓存下来，通过时间或者其他因素　来判别是读缓存还是网络请求，这样可以缓解服务器压力，一定程度上提高应用响应速度，并且支持离线阅读。

Bitmap的缓存

在许多的情况下(像 ListView, GridView 或 ViewPager 之类的组件)我们需要一次性加载大量的图片，在屏幕上显示的图片和所有待显示的图片有可能需要马上就在屏幕上无限制的进行滚动、切换。

像ListView, GridView 这类组件，它们的子项当不可见时，所占用的内存会被回收以供正在前台显示子项使用。垃圾回收器也会释放你已经加载了的图片占用的内存。如果你想让你的UI运行流畅的话，就不应该每次显示时都去重新加载图片。保持一些内存和文件缓存就变得很有必要了。

使用内存缓存

通过预先消耗应用的一点内存来存储数据，便可快速的为应用中的组件提供数据，是一种典型的以**空间换时间**的策略。
LruCache 类（Android v4 Support Library 类库中开始提供）非常适合来做图片缓存任务 ，它可以使用一个LinkedHashMap 的强引用来保存最近使用的对象，并且当它保存的对象占用的内存总和超出了为它设计的最大内存时会把**不经常使用**的对象成员踢出以供垃圾回收器回收。

给LruCache 设置一个合适的内存大小，需考虑如下因素：

- 还剩余多少内存给你的activity或应用使用
- 屏幕上需要一次性显示多少张图片和多少图片在等待显示
- 手机的大小和密度是多少（密度越高的设备需要越大的 缓存）
- 图片的尺寸（决定了所占用的内存大小）
- 图片的访问频率（频率高的在内存中一直保存）
- 保存图片的质量（不同像素的在不同情况下显示）

具体的要根据应用图片使用的具体情况来找到一个合适的解决办法，一个设置 LruCache 例子：

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // 获得虚拟机能提供的最大内存，超过这个大小会抛出OutOfMemory的异常
    final int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);

    // 用 1 / 8 的内存大小作为内存缓存
    final int cacheSize = maxMemory / 8;

    mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // 这里返回的不是item的个数，是cache的size（单位1024个字节）
            return bitmap.getByteCount() / 1024;
        }
    };
    ...
}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromMemCache(String key) {
    return mMemoryCache.get(key);
}
```

当为ImageView加载一张图片时，会先在LruCache 中看看有没有缓存这张图片，如果有的话直接更新到ImageView中，如果没有的话，一个后台线程会被触发来加载这张图片。

```
public void loadBitmap(int resId, ImageView imageView) {
    final String imageKey = String.valueOf(resId);

    // 查看下内存缓存中是否缓存了这张图片
    final Bitmap bitmap = getBitmapFromMemCache(imageKey);
    if (bitmap != null) {
        mImageView.setImageBitmap(bitmap);
    } else {
        mImageView.setImageResource(R.drawable.image_placeholder);
BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }
}
```

在图片加载的Task中，需要把加载好的图片加入到内存缓存中。

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...
    // 在后台完成
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final Bitmap bitmap = decodeSampledBitmapFromResource(
                getResources(), params[0], 100, 100));
    addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);
        return bitmap;
    }
    ...
}
```

使用磁盘缓存

内存缓存能够快速的获取到最近显示的图片，但不一定就能够获取到。当数据集过大时很容易把内存缓存填满（如GridView ）。你的应用也有可能被其它的任务（比如来电）中断进入到后台，后台应用有可能会被杀死，那么相应的内存缓存对象也会被销毁。 当你的应用重新回到前台显示时，你的应用又需要一张一张的去加载图片了。

磁盘文件缓存能够用来处理这些情况，保存处理好的图片，当内存缓存不可用的时候，直接读取在硬盘中保存好的图片，这样可以有效的减少图片加载的次数。读取磁盘文件要比直接从内存缓存中读取要慢一些，而且需要在一个UI主线程外的线程中进行，因为磁盘的读取速度是不能够保证的，磁盘文件缓存显然也是一种以**空间换时间**的策略。

如果图片使用非常频繁的话，一个 ContentProvider 可能更适合代替去存储缓存图片，比如图片gallery 应用。

下面是一个DiskLruCache的部分代码：

```
private DiskLruCache mDiskLruCache;
private final Object mDiskCacheLock = new Object();
private boolean mDiskCacheStarting = true;
private static final int DISK_CACHE_SIZE = 1024 * 1024 * 10; // 10MB
private static final String DISK_CACHE_SUBDIR = "thumbnails";

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // 初始化内存缓存
    ...
    // 在后台线程中初始化磁盘缓存
    File cacheDir = getDiskCacheDir(this, DISK_CACHE_SUBDIR);
    new InitDiskCacheTask().execute(cacheDir);
    ...
}

class InitDiskCacheTask extends AsyncTask<File, Void, Void> {
    @Override
    protected Void doInBackground(File... params) {
        synchronized (mDiskCacheLock) {
            File cacheDir = params[0];
  mDiskLruCache = DiskLruCache.open(cacheDir, DISK_CACHE_SIZE);
   mDiskCacheStarting = false; // 结束初始化
   mDiskCacheLock.notifyAll(); // 唤醒等待线程
        }
        return null;
    }
}

class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    ...
    // 在后台解析图片
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final String imageKey = String.valueOf(params[0]);

        // 在后台线程中检测磁盘缓存
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);

        if (bitmap == null) { // 没有在磁盘缓存中找到图片
 final Bitmap bitmap = decodeSampledBitmapFromResource(
                    getResources(), params[0], 100, 100));
        }

        // 把这个final类型的bitmap加到缓存中
        addBitmapToCache(imageKey, bitmap);

        return bitmap;
    }
    ...
}

public void addBitmapToCache(String key, Bitmap bitmap) {
    // 先加到内存缓存
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }

    //再加到磁盘缓存
    synchronized (mDiskCacheLock) {
        if (mDiskLruCache != null && mDiskLruCache.get(key) == null) {
            mDiskLruCache.put(key, bitmap);
        }
    }
}

public Bitmap getBitmapFromDiskCache(String key) {
    synchronized (mDiskCacheLock) {
        // 等待磁盘缓存从后台线程打开
        while (mDiskCacheStarting) {
            try {
                mDiskCacheLock.wait();
            } catch (InterruptedException e) {}
        }
        if (mDiskLruCache != null) {
            return mDiskLruCache.get(key);
        }
    }
```

```
    return null;
}

public static File getDiskCacheDir(Context context, String uniqueName) {
    // 优先使用外缓存路径，如果没有挂载外存储，就使用内缓存路径
final String cachePath =
            Environment.MEDIA_MOUNTED.equals(Environment.getExternalStorageState()) ||
!isExternalStorageRemovable() ?getExternalCacheDir(context).getPath():context.getCacheDir().getPath()

    return new File(cachePath + File.separator + uniqueName);
}
```

不能在UI主线程中进行这项操作，因为初始化磁盘缓存也需要对磁盘进行操作。上面的程序片段中，一个锁对象确保了磁盘缓存没有初始化完成之前不能够对磁盘缓存进行访问。

内存缓存在UI线程中进行检测，磁盘缓存在UI主线程外的线程中进行检测，当图片处理完成之后，分别存储到内存缓存和磁盘缓存中。

设备配置参数改变时加载问题

由于应用在运行的时候设备配置参数可能会发生改变，比如设备朝向改变，会导致Android销毁你的Activity然后按照新的配置重启，这种情况下，我们要避免重新去加载处理所有的图片，让用户能有一个流畅的体验。

使用Fragment 能够把内存缓存对象传递到新的activity实例中，调用setRetainInstance(true)) 方法来保留Fragment实例。当activity重新创建好后， 被保留的Fragment依附于activity而存在，通过Fragment就可以获取到已经存在的内存缓存对象了，这样就可以快速的获取到图片，并设置到ImageView上，给用户一个流畅的体验。

下面是一个示例程序片段：

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
RetainFragment mRetainFragment =            RetainFragment.findOrCreateRetainFragment(getFragmentMana
    mMemoryCache = RetainFragment.mRetainedCache;
    if (mMemoryCache == null) {
        mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
            ... //像上面例子中那样初始化缓存
        }
        mRetainFragment.mRetainedCache = mMemoryCache;
    }
    ...
}

class RetainFragment extends Fragment {
    private static final String TAG = "RetainFragment";
    public LruCache<String, Bitmap> mRetainedCache;

    public RetainFragment() {}

    public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {
        RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);
        if (fragment == null) {
            fragment = new RetainFragment();
        }
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 使得Fragment在Activity销毁后还能够保留下来
        setRetainInstance(true);
    }
}
```

可以在不适用Fragment（没有界面的服务类Fragment）的情况下旋转设备屏幕。在保留缓存的情况下，你应该能发现填充图片到Activity中几乎是瞬间从内存中取出而没有任何延迟的感觉。任何图片优先从内存缓存获取，没有的话再到硬盘缓存中找，如果都没有，那就以普通方式加载图片。　　参考：

Caching Bitmaps

LruCache

使用SQLite进行缓存

网络请求数据完成后，把文件的相关信息（如url（一般作为唯一标示），下载时间，过期时间）等存放到数据库。下次加载的时候根据url先从数据库中查询，如果查询到并且时间未过期，就根据路径读取本地文件，从而实现缓存的效果。

注意：缓存的数据库是存放在/data/data//databases/目录下，是占用内存空间的，如果缓存累计，容易浪费内存，需要及时清理缓存。

文件缓存

思路和一般缓存一样，把需要的数据存储在文件中，下次加载时判断文件是否存在和过期（使用File.lastModified()方法得到文件的最后修改时间，与当前时间判断），存在并未过期就加载文件中的数据，否则请求服务器重新下载。

注意，无网络环境下就默认读取文件缓存中的。