ListView 中的 RecycleBin 机制

2016-07-22 安卓应用频道

(点击上方公众号，可快速关注)

> 来源：孙群
> 链接：http://blog.csdn.net/iispring/article/details/50967445

在自定义Adapter时，我们常常会重写Adapter的getView方法，该方法的签名如下所示：

```
public abstract View getView(int position, View convertView, ViewGroup parent)
```

此处会传入一个convertView变量，它的值有可能是null，也有可能不是null，如果不为null，我们就可以复用该convertView，对convertView里面的一些控件赋值后可以将convertView作为getView的返回值返回，这么做的目的是减少LayoutInflater.inflate()的调用次数，从而提升了性能（LayoutInflater.inflate()比较消耗性能）。

本文将介绍ListView中的RecycleBin机制，让大家对ListView中的优化机制有个概括的了解，同时也说明convertView的来龙去脉。

首先，我们知道，Adapter是数据源，AdapterView是展示数据源的UI控件，Adapter是给AdapterView使用的，通过调用AdapterView的setAdapter方法就可以让一个AdapterView绑定Adapter对象，从而AdapterView会将Adapter中的数据展示出来。

AdapterView的子类有AbsListView和AbsSpinner等，其中AbsListView的子类又有ListView、GridView等，所以ListView继承自AdapterView。

如果Adapter中有10000条数据，将这个Adapter对象赋给ListView，如果ListView创建10000个子View，那么App肯定崩溃了，因为Android没有能力同时绘制这么多的子View。而且，即便能同时绘制这10000个子View也没什么意义，因为手机的屏幕大小是有限的，有可能ListView的高度只能最多显示10个子View。基于此，Android在设计ListView这个类的时候，引入了RecycleBin机制——对子View进行回收利用，RecycleBin直译过来就是回收站的意思。

RecycleBin基本原理

下面先简要说一下RecycleBin中的工作原理，后面会结合源码详细说明。

在某一时刻，我们看到ListView中有许多View呈现在UI上，这些View对我们来说是可见的，这些可见的View可以称作OnScreen的View，即在屏幕中能看到的View，也可以叫做ActiveView，因为它们是在UI上可操作的。

当触摸ListView并向上滑动时，ListView上部的一些OnScreen的View位置上移，并移除了ListView的屏幕范围，此时这些OnScreen的View就变得不可见了，不可见的View叫做OffScreen的View，即这些View已经不在屏幕可见范围内了，也可以叫做ScrapView，Scrap表示废弃的意思，ScrapView的意思是这些OffScreen的View不再处于可以交互的Active状态了。ListView会把那些ScrapView（即OffScreen的View）删除，这样就不用绘制这些本来就不可见的View了，同时，ListView会把这些删除的ScrapView放入到RecycleBin中存起来，就像把暂时无用的资源放到回收站一样。

当ListView的底部需要显示新的View的时候，会从RecycleBin中取出一个ScrapView，将其作为convertView参数传递给Adapter的getView方法，从而达到View复用的目的，这样就不必在Adapter的getView方法中执行LayoutInflater.inflate()方法了。

RecycleBin中有两个重要的View数组，分别是mActiveViews和mScrapViews。这两个数组中所存储的View都是用来复用的，只不过mActiveViews中存储的是OnScreen的View，这些View很有可能被直接复用；而mScrapViews中存储的是OffScreen的View，这些View主要是用来间接复用的。

上面对mActiveViews和mScrapViews的说明比较笼统，其实在细节上还牵扯到Adapter的数据源发生变化的情况，具体细节后面会讲解。

源码解析

AdapterView是继承自ViewGroup的，ViewGroup中有addView方法可以向ViewGroup中添加子View，但是AdapterView重写了addView方法，如下所示：

```
@Override
public void addView(View child) {
    throw new UnsupportedOperationException("addView(View) is not supported in AdapterView");
}
```

```
@Override
public void addView(View child, int index) {
    throw new UnsupportedOperationException("addView(View, int) is not supported in AdapterView");
}
```

在AdapterView的addView方法中会抛出异常，也就是说AdapterView禁用了addView方法。

在具体讲解之前，我们还是先花一点时间简要说一下View的每一帧的显示流程，当然，ListView也肯定遵循此流程。一个View要想在界面上呈现出来，需要经过三个阶段：measure->layout->draw。

View是一帧一帧绘制的，每一帧绘制都经历了measure->layout->draw这三个阶段，绘制完一帧之后，如果UI需要更新，比如用户滚动了ListView，那么又会绘制下一帧，再次经历measure->layout->draw方法，如果对此不了解，可以参见另一篇博文《 Android中View的量算、布局及绘图机制》。

我们上面说了，AdapterView把addView方法给禁用了，那么ListView怎么向其中添加child呢？奥秘就在layout中，在布局的时候，ListView会执行layoutChildren方法，该方法是ListView对View进行添加以及回收的关键方法，RecycleBin的很多方法都在layoutChildren方法中被调用。在layoutChildren方法中实现对子View的增删，经过layoutChildren方法之后，ListView中所有的子View都是在屏幕中可见的，也就是说layoutChildren方法为接下来的帧绘制把子View准备完善了，这就保证了在后面的draw方法的执行过程中能够正确绘制ListView。

ListView的layoutChildren方法代码比较多，我们只研究和View增删相关的关键代码，主要分以下三个阶段：

1. ListView的children->RecycleBin
2. ListView清空children
3. RecycleBin->ListView的children

在layout这个方法刚刚开始执行的时候，ListView中的children其实还是上一帧中需要绘制的子View的集合，在layout这个方法执行完成的时候，ListView中的children就变成了当前帧马上要进行绘制的子View的集合。

下面对以上这三个阶段分别说明。

1. ListView的children->RecycleBin

该阶段的关键代码如下所示：

```
//mFirstPosition是ListView的成员变量，存储着第一个显示的child所对应的adapter的position
    final int firstPosition = mFirstPosition;
    final RecycleBin recycleBin = mRecycler;
    if (dataChanged) {
        //如果数据发生了变化，那么就把ListView的所有子View都放入到RecycleBin的mScrapViews数组中
        for (int i = 0; i < childCount; i++) {
            //addScrapView方法会传入一个View，以及这个View所对应的position
            recycleBin.addScrapView(getChildAt(i), firstPosition+i);
        }
    } else {
        //如果数据没发生变化，那么把ListView的所有子View都放入到RecycleBin的mActiveViews数组中
        recycleBin.fillActiveViews(childCount, firstPosition);
    }
```

再次强调一下，在上面的代码刚开始的时候，ListView的中的children还是上一帧需要绘制的子View。

- 如果Adapter调用了notifyDataSetChanged方法，那么AdapterView就会知道Adapter的数据源发生了变化，此时dataChanged变量就为true，这种情况下，ListView会认为children中的View都是不合格的了，这时候会用getChildAt方法遍历children中所有的child，并把这些child通过RecycleBin的addScrapView方法将其放入RecycleBin的mScrapViews数组中。
- 如果adapter的数据没有发生变化，那么会调用RecycleBin的fillActiveViews方法将所有的children都放入到RecycleBin的mActiveViews数组中。

经过上面的操作之后，ListView所有的子View都放入到了RecycleBin中，这就实现了ListView的children->RecycleBin的迁移过程，放到RecycleBin的目的是为了分类缓存ListView中的children，以便在后续过程中对这些View进行复用。

2. ListView清空children

然后调用ViewGroup的detachAllViewsFromParent方法，该方法将所有的子View从ListView中分离，也就是清空了children，该方法源码如下所示：

```
protected void detachAllViewsFromParent() {
```

```
    final int count = mChildrenCount;
    if (count <= 0) {
        return;
    }

    final View[] children = mChildren;
    mChildrenCount = 0;

    for (int i = count - 1; i >= 0; i--) {
        children[i].mParent = null;
        children[i] = null;
    }
}
```

3. RecycleBin->ListView的children

然后ListView会根据mLayoutMode进行判断，源码如下所示：

```
switch (mLayoutMode) {
        case LAYOUT_SET_SELECTION:
            if (newSel != null) {
                sel = fillFromSelection(newSel.getTop(), childrenTop, childrenBottom);
            } else {
                sel = fillFromMiddle(childrenTop, childrenBottom);
            }
            break;
        case LAYOUT_SYNC:
            sel = fillSpecific(mSyncPosition, mSpecificTop);
            break;
        case LAYOUT_FORCE_BOTTOM:
            sel = fillUp(mItemCount - 1, childrenBottom);
            adjustViewsUpOrDown();
            break;
        case LAYOUT_FORCE_TOP:
            mFirstPosition = 0;
            sel = fillFromTop(childrenTop);
            adjustViewsUpOrDown();
            break;
        case LAYOUT_SPECIFIC:
```

```
            sel = fillSpecific(reconcileSelectedPosition(), mSpecificTop);
            break;
        case LAYOUT_MOVE_SELECTION:
            sel = moveSelection(oldSel, newSel, delta, childrenTop, childrenBottom);
            break;
        default:
            if (childCount == 0) {
                if (!mStackFromBottom) {
                    final int position = lookForSelectablePosition(0, true);
                    setSelectedPositionInt(position);
                    sel = fillFromTop(childrenTop);
                } else {
                    final int position = lookForSelectablePosition(mItemCount - 1, false);
                    setSelectedPositionInt(position);
                    sel = fillUp(mItemCount - 1, childrenBottom);
                }
            } else {
                if (mSelectedPosition >= 0 && mSelectedPosition < mItemCount) {
                    sel = fillSpecific(mSelectedPosition,
                            oldSel == null ? childrenTop : oldSel.getTop());
                } else if (mFirstPosition < mItemCount) {
                    sel = fillSpecific(mFirstPosition,
                            oldFirst == null ? childrenTop : oldFirst.getTop());
                } else {
                    sel = fillSpecific(0, childrenTop);
                }
            }
            break;
        }
```

在该switch代码段中，会根据不同情况增删子View，这些方法的代码逻辑大部分最终调用了fillDown、fillUp等方法。

fillDown用子View从指定的position自上而下填充ListView，fillUp则是自下而上填充，我们以fillDown方法为例详细说明。

fillDown方法的源码如下所示：

```
    private View fillDown(int pos, int nextTop) {
```

```
    View selectedView = null;

    //end表示ListView的高度
    int end = (mBottom - mTop);
    if ((mGroupFlags & CLIP_TO_PADDING_MASK) == CLIP_TO_PADDING_MASK) {
        end -= mListPadding.bottom;
    }

    //nextTop < end确保了我们只要将新增的子View能够覆盖ListView的界面就可以了
    //pos < mItemCount确保了我们新增的子View在Adapter中都有对应的数据源item
    while (nextTop < end && pos < mItemCount) {
        // is this the selected item?
        boolean selected = pos == mSelectedPosition;
        View child = makeAndAddView(pos, nextTop, true, mListPadding.left, selected);
        //将最新child的bottom值作为下一个child的top值，存储在nextTop中
        nextTop = child.getBottom() + mDividerHeight;
        if (selected) {
            selectedView = child;
        }
        //position自增
        pos++;
    }

    setVisibleRangeHint(mFirstPosition, mFirstPosition + getChildCount() - 1);
    return selectedView;
}
```

- fillDown接收两个参数，pos表示列表中第一个要绘制的item的position，其对应着Adapter中的索引，nextTop表示第一个要绘制的item在ListView中实际的位置， 即该item所对应的子View的顶部到ListView的顶部的像素数。
- 首先将mBottom – mTop的值作为end，end表示ListView的高度。
- 然后在while循环中添加子View，我们先不看while循环的具体条件，先看一下循环体。在循环体中，将pos和nextTop传递给makeAndAddView方法，该方法返回一个View作为child，该方法会创建View，并把该View作为child添加到ListView的children数组中。
- 然后执行nextTop = child.getBottom() + mDividerHeight，child的bottom值表示的是该child的底部到ListView顶部的距离，将该child的bottom作为下一个child的top，也就是说nextTop一直保存着下一个child的top值。
- 最后调用pos++实现position指针下移。现在我们回过头来看一下while循环的条件

while (nextTop < end && pos < mItemCount)。

- nextTop < end确保了我们只要将新增的子View能够覆盖ListView的界面就可以了，比如ListView的高度最多显示10个子View，我们没必要向ListView中加入11个子View。
- pos < mItemCount确保了我们新增的子View在Adapter中都有对应的数据源item，比如ListView的高度最多显示10个子View，但是我们Adapter中一共才有5条数据，这种情况下只能向ListView中加入5个子View，从而不能填充满ListView的全部高度。

经过了上面的while循环之后，ListView对子View的增删就完成了，即children中存放的就是要在后面绘图过程中即将渲染的子View的集合。

上面while循环的方法体中调用了makeAndAddView方法，通过该方法会获得一个子View，并把该子View添加到ListView的children中。该方法的方法签名如下所示：

```
private View makeAndAddView(int position, int y, boolean flow, int childrenLeft,
        boolean selected)
```

其源码如下所示：

```
private View makeAndAddView(int position, int y, boolean flow, int childrenLeft,
        boolean selected) {
    View child;


    if (!mDataChanged) {
        // 如果数据源没发生变化，那么尝试用该position从RecycleBin的mActiveViews中获取可复用的View
        child = mRecycler.getActiveView(position);
        if (child != null) {
            // 如果child 不为空，说明我们找到了一个已经存在的child，这样mActiveViews中存储的View就被直接复用了
            // 调用setupChild，对child进行定位
            setupChild(child, position, y, flow, childrenLeft, selected, true);

            return child;
        }
    }

    // 如果没能够从mActivieViews中直接复用View，那么就要调用obtainView方法获取View，该方法尝试间接复用RecycleBin中的mScrapViews中的View，如果不能间接复用，则创建新的View
    child = obtainView(position, mIsScrap);
```

```
        // 调用setupChild方法，进行定位和量算
        setupChild(child, position, y, flow, childrenLeft, selected, mIsScrap[0]);

        return child;
    }
```

我们重点说一下前两个参数position和y，position表示的是数据源item在Adapter中的索引，y表示要生成的View的top值或bottom值。如果第三个参数flow是true，那么y表示top值，否则表示bottom值。

- 如果数据源没发生变化，那么尝试用该position从RecycleBin的mActiveViews中获取可复用的View。RecycleBin的getActiveView方法接收一个position参数，可以在RecycleBin的mActiveViews数组中查找有没有对应position的View，如果能找到就可以直接复用该View作为child了。举一个例子，假设在某一时刻ListView中显示了10个子View，position依次为从0到9。然后我们手指向上滑动，且向上滑动了一个子View的高度，ListView需要绘制下一帧。这时候ListView在layoutChildren方法中把这10个子View都放入到了RecycleBin的mActiveViews数组中了，然后清空了children数组，然后调用fillDown方法，向ListView中依次添加position1到10的子View，在添加position为1的子View的时候，由于在上一帧中position为1的子View已经被放到mActiveViews数组中了，这次直接可以将其从mActiveViews数组中取出来，**这样就是直接复用子View，所以说RecycleBin的mActiveViews数组主要是用于直接复用的。**

在直接复用了子View后，我们需要调用setupChild方法，该方法会将child添加到ListView的children数组中，并对child进行定位。

- 如果没能够从mActivieViews中直接复用View，那么就要调用obtainView方法获取View，该方法尝试间接复用RecycleBin中的mScrapViews中的View，如果不能间接复用，则创建新的View。

在通过obtainView获取了View之后，调用setupChild方法，该方法会将child添加到ListView的children数组中，并对child进行定位和量算。

下面我们再来看一下obtainView方法，该方法的方法签名如下所示：

```
View obtainView(int position, boolean[] isScrap)
```

该方法接收position参数，其关键的源码有以下两行：

```
final View scrapView = mRecycler.getScrapView(position);
final View child = mAdapter.getView(position, scrapView, this);
```

通过调用RecycleBin的getScrapView方法，从mScrapViews数组中获取一个View，该View是用来间接复用的，该View可能为null，也可能不为null，将其作为我们熟悉的convertView传递给Adapter的getView方法，这样我们就可以在AdapterView的getView方法中通过判断convertView是否为空进行间接复用了。

希望本文对大家理解ListView的RecycleBin机制有所帮助！