android studio 解决actionbar永远是overflow

在项目结构里吧v7支持包删掉就好 前提api level>=14

用其他命名没有用

解决actionbar在4.0系统上overflow无图标

源码

@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
 getMenuInflater().inflate(R.menu.*mainactivity\_menu*, menu);  
 return true;  
}  
  
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
 switch (item.getItemId()) {  
 case android.R.id.*home*:  
 GotoHome();  
 return true;  
 default:  
 return super.onOptionsItemSelected(item);  
 }  
}

添加

@Override  
 public boolean onMenuOpened(int featureId, Menu menu) {  
 if (featureId == Window.*FEATURE\_ACTION\_BAR* && menu != null) {  
 if (menu.getClass().getSimpleName().equals("MenuBuilder")) {  
 try {  
 Method m = menu.getClass().getDeclaredMethod(  
 "setOptionalIconsVisible", Boolean.*TYPE*);  
 m.setAccessible(true);  
 m.invoke(menu, true);  
 } catch (Exception e) {  
 }  
 }  
 }  
 return super.onMenuOpened(featureId, menu);  
 }  
  
 private void setOverflowShowingAlways() {  
// *TODO Auto-generated method stub* try {  
 ViewConfiguration config = ViewConfiguration.*get*(this);  
 Field menuKeyField = ViewConfiguration.class  
 .getDeclaredField("sHasPermanentMenuKey");  
 menuKeyField.setAccessible(true);  
 menuKeyField.setBoolean(config, false);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }

创建抽屉式导航栏布局

要添加抽屉式导航栏，请将包含 [DrawerLayout](https://developer.android.com/reference/android/support/v4/widget/DrawerLayout.html) 对象的用户界面声明为布局的根视图。在 [DrawerLayout](https://developer.android.com/reference/android/support/v4/widget/DrawerLayout.html)内，添加一个包含屏幕主内容（当抽屉式导航栏处于隐藏状态时为主要布局）的视图和另一个包含抽屉式导航栏内容的视图。

例如，以下布局使用包含两个子视图的 [DrawerLayout](https://developer.android.com/reference/android/support/v4/widget/DrawerLayout.html)：包含主内容的 [FrameLayout](https://developer.android.com/reference/android/widget/FrameLayout.html)（在运行时由[Fragment](https://developer.android.com/reference/android/app/Fragment.html) 填充）和抽屉式导航栏的 [ListView](https://developer.android.com/reference/android/widget/ListView.html)。

<android.support.v4.widget.DrawerLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/drawer\_layout"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
    <!-- The main content view -->  
    <FrameLayout  
        android:id="@+id/content\_frame"  
        android:layout\_width="match\_parent"  
        android:layout\_height="match\_parent" />  
    <!-- The navigation drawer -->  
    <ListView android:id="@+id/left\_drawer"  
        android:layout\_width="240dp"  
        android:layout\_height="match\_parent"  
        android:layout\_gravity="start"  
        android:choiceMode="singleChoice"  
        android:divider="@android:color/transparent"  
        android:dividerHeight="0dp"  
        android:background="#111"/>  
</android.support.v4.widget.DrawerLayout>

此布局演示了一些重要的布局特性：

* 在 [DrawerLayout](https://developer.android.com/reference/android/support/v4/widget/DrawerLayout.html) 中，主内容视图（上面的 [FrameLayout](https://developer.android.com/reference/android/widget/FrameLayout.html)）**必须是第一个子视图**，因为 XML 顺序意味着按 z 序（层叠顺序）排序，并且抽屉式导航栏必须位于内容顶部。
* 主内容视图设置为匹配父视图的宽度和高度， 因为在抽屉式导航栏处于隐藏状态时， 它代表整个 UI。

所以编译器抛出异常： java.lang.ClassCastException: android.app.Application cannot be cast to 类名  
  <application android:icon="@drawable/icon" android:label="@string/app\_name"  
                android:name=".你所新建的lication类名">

ViewHolder通常出现在适配器里，为的是listview滚动的时候快速设置值，而不必每次都重新创建很多对象，从而提升性能。  
  
在android开发中Listview是一个很重要的组件，它以列表的形式根据数据的长自适应展示具体内容,用户可以自由的定义listview每一列的布局，但当listview有大量的数据需要加载的时候，会占据大量内存，影响性能，这时候就需要按需填充并重新使用view来减少对象的创建。  
ListView加载数据都是在public View getView(int position, View convertView,   
ViewGroup parent)   
{}方法中进行的(要自定义listview都需要重写listadapter:如BaseAdapter，SimpleAdapter,CursorAdapter的等的getvView方法),优化listview的加载速度就要让convertView匹配列表类型，并最大程度上的重新使用convertView。  
getview的加载方法一般有以下三种种方式：  
最慢的加载方式是每一次都重新定义一个View载入布局，再加载数据  
public View getView(int position, View convertView, ViewGroup parent)   
{  
 View item = mInflater.inflate(R.layout.list\_item\_icon\_text,   
null);  
 ((TextView)   
item.findViewById(R.id.text)).setText(DATA[position]);  
 ((ImageView) item.findViewById(R.id.icon)).setImageBitmap(  
 (position & 1) == 1 ? mIcon1 : mIcon2);  
 return item;  
}  
正确的加载方式是当convertView不为空的时候直接重新使用convertView从而减少了很多不必要的View的创建，然后加载数据  
public View getView(int position, View convertView, ViewGroup parent)   
{  
 if (convertView == null) {  
 convertView = mInflater.inflate(R.layout.item, parent, false);  
 }  
 ((TextView)   
convertView.findViewById(R.id.text)).setText(DATA[position]);  
 ((ImageView)   
convertView.findViewById(R.id.icon)).setImageBitmap(  
 (position & 1) == 1 ? mIcon1 : mIcon2);  
 return convertView;  
 }  
最快的方式是定义一个ViewHolder，将convetView的tag设置为ViewHolder,不为空时重新使用即可  
static class ViewHolder {  
TextView text;  
ImageView icon;  
}  
public View getView(int position, View convertView, ViewGroup parent)   
{  
 ViewHolder holder;  
 if (convertView == null) {  
 convertView = mInflater.inflate(R.layout.list\_item\_icon\_text,  
 parent, false);  
 holder = new ViewHolder();  
 holder.text = (TextView) convertView.findViewById(R.id.text);  
 holder.icon = (ImageView) convertView.findViewById(R.id.icon);  
 convertView.setTag(holder);  
} else {  
holder = (ViewHolder) convertView.getTag();  
}  
holder.text.setText(DATA[position]);  
holder.icon.setImageBitmap((position & 1) == 1 ? mIcon1 :   
mIcon2);  
return convertView;  
}

/\*\*

\* 这个类是主线程，它每隔两秒就发送一个Message给线程，线程收到Message后将发送一个Message给主线程，主线程收到Message后会将Message内容Toast出来。

\* org.fneg.HandlerTestActivity.java

\* Create at: 2012-6-4 下午9:39:19

\* @author:feng

\* Email:fengcunhan@gmail.com

\*

\*/

public class HandlerTestActivity extends Activity {

private Handler uiHandler;

private ThreadWithLooper thread;

private Runnable showRunable;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

uiHandler=new Handler(){

@Override

public void handleMessage(Message msg) {

switch(msg.what){

case Messages.MSG\_HELLO:

Toast.makeText(HandlerTestActivity.this, (String)msg.obj, Toast.LENGTH\_SHORT).show();

break;

}

}

};

thread=new ThreadWithLooper(uiHandler);

thread.start();

showRunable=new Runnable() {

@Override

public void run() {

//給线程发送一个Message

thread.getHandler().sendEmptyMessage(Messages.MSG\_HELLO);

uiHandler.postDelayed(this, 2\*1000);

}

};

uiHandler.post(showRunable);

}

@Override

protected void onStop() {

super.onStop();

uiHandler.removeCallbacks(showRunable);

}

}

import android.os.Handler;

import android.os.Looper;

import android.os.Message;

/\*\*

\* 从线程发送消息到UI线程（主线程）

\* org.fneg.ThreadWithLooper.java

\* Create at: 2012-6-4 下午4:58:11

\* @author:feng<br/>

\* Email:fengcunhan@gmail.com

\*

\*/

public class ThreadWithLooper extends Thread {

private Handler handler;

private Handler uiHandler;

public ThreadWithLooper(Handler mHandler){

this.uiHandler=mHandler;

//初始化Handler，接收到主线程发送过来的Message就回复一个Message给主线程，消息内容是 一个字符串和当前时间

handler =new Handler(){

@Override

public void handleMessage(Message msg) {

switch(msg.what){

case Messages.MSG\_HELLO:

Message message=new Message();

message.what=Messages.MSG\_HELLO;

message.obj="Yes!I get a hello"+System.currentTimeMillis();

uiHandler.sendMessage(message);

break;

}

}

};

}

public Handler getHandler() {

return handler;

}

public void setHandler(Handler handler) {

this.handler = handler;

}

@Override

public void run() {

Looper.prepare();

Looper.loop();

}

}

/\*\*

\*

\* org.fneg.Messages.java

\* Create at: 2012-6-4 下午4:51:20

\* @author:feng

\* Email:fengcunhan@gmail.com

\*

\*/

public class Messages {

public static final int MSG\_HELLO=0X1;

}