# 启动服务

## 调起

Android壁纸服务WallpaperManagerService启动在SystemServer中。当NonCoreServices没有被禁止且config\_enableWallpaperService的config值为true时，启动WallpaperManagerService。

/frameworks/base/services/java/com/android/server/SystemServer.java

...

if (!disableNonCoreServices && context.getResources().getBoolean(

R.bool.config\_enableWallpaperService)) {

try {

Slog.i(TAG, "Wallpaper Service");

wallpaper = new WallpaperManagerService(context);

ServiceManager.addService(Context.WALLPAPER\_SERVICE, wallpaper);

} catch (Throwable e) {

reportWtf("starting Wallpaper Service", e);

}

}

...

## WallpaperManagerService

Sdf

/frameworks/base/services/core/java/com/android/server/wallpaper/WallpaperManagerService.java

public WallpaperManagerService(Context context) {

if (DEBUG) Slog.v(TAG, "WallpaperService startup");

mContext = context;

mImageWallpaper = ComponentName.unflattenFromString(

context.getResources().getString(R.string.image\_wallpaper\_component));

mIWindowManager = IWindowManager.Stub.asInterface(

ServiceManager.getService(Context.WINDOW\_SERVICE));

mIPackageManager = AppGlobals.getPackageManager();

mMonitor = new MyPackageMonitor();

mMonitor.register(context, null, UserHandle.ALL, true);

getWallpaperDir(UserHandle.USER\_OWNER).mkdirs();

loadSettingsLocked(UserHandle.USER\_OWNER);

}

mImageWallpaper的值是一个ComponentName，其值是”com.android.systemui/com.android.systemui.ImageWallpaper”。getWallpaperDir则在设备上创建/data/system/users/0的目录。loadSettingsLocked用来读取保存在xml文件里面的壁纸信息。

*<!-- Component name of the built in wallpaper used to display bitmap wallpapers. This must not be null. -->*<**string name="image\_wallpaper\_component" translatable="false"**>com.android.systemui/com.android.systemui.ImageWallpaper</**string**>

## loadSettingsLocked

loadSettingsLocked中，先是读取/data/system/users/0/wallpaper\_info.xml的内容，将一些信息例如壁纸的宽度（width），壁纸的高度（height），名字（name），下一个壁纸组件（component）等信息记录在WallpaperData中，这个WallpaperData和userid组成一个键值对保存在mWallpaperMap中。这样，不同的userid就可以拥有不同的壁纸配置信息。最后，getMaximumSizeDimension返回的是当前屏幕逻辑高度和逻辑宽度的最大值，若壁纸信息记录的宽度或高度小于这个值，则把壁纸的宽度或高度设置成getMaximumSizeDimension的返回值。屏幕逻辑高度和逻辑宽度根据当前的分辨率而定，可能1920x1080的屏幕物理分辨率是1920x1080，但是修改分辨率后，例如1280x720，那么逻辑宽度和逻辑高度就分别为1280和720

2|bullhead:/data/system/users # cat 0/wallpaper\_info.xml

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>

<wp id="1" width="2160" height="1920" cropLeft="0" cropTop="0" cropRight="1920" cropBottom="1920" name="" backup="true" />

## systemRunning

## switchWallpaper

asd

void switchWallpaper(WallpaperData wallpaper, IRemoteCallback reply) {

synchronized (mLock) {

RuntimeException e = null;

try {

ComponentName cname = wallpaper.wallpaperComponent != null ?

wallpaper.wallpaperComponent : wallpaper.nextWallpaperComponent;

if (bindWallpaperComponentLocked(cname, true, false, wallpaper, reply)) {

return;

}

} catch (RuntimeException e1) {

e = e1;

}

Slog.w(TAG, "Failure starting previous wallpaper", e);

clearWallpaperLocked(false, wallpaper.userId, reply);

}

}

switchWallpaper主要是调用bindWallpaperComponentLocked来启动真正的壁纸组件”com.android.systemui/com.android.systemui.ImageWallpaper”。

只看bindWallpaperComponentLocked的关键部分

## bindWallpaperComponentLocked

简而言之就是用bindServiceAsUser来启动一个WallpaperService，而SystemUI中的ImageWallpaper继承自WallpaperService，可以借此启动。若bindWallpaperComponentLocked没有出错返回，则更新WallpaperData的wallpaperComponent和connection为启动的壁纸组件的包名和ServiceConnection。最后，往WindowManagerService添加一个类型为TYPE\_WALLPAPER的WindowToken，将mLastWallpaper设为当前的WallpaperData。

boolean bindWallpaperComponentLocked(ComponentName componentName, boolean force,

boolean fromUser, WallpaperData wallpaper, IRemoteCallback reply) {

...

// Bind the service!

if (DEBUG) Slog.v(TAG, "Binding to:" + componentName);

WallpaperConnection newConn = new WallpaperConnection(wi, wallpaper);

intent.setComponent(componentName);

intent.putExtra(Intent.EXTRA\_CLIENT\_LABEL,

com.android.internal.R.string.wallpaper\_binding\_label);

intent.putExtra(Intent.EXTRA\_CLIENT\_INTENT, PendingIntent.getActivityAsUser(

mContext, 0,

Intent.createChooser(new Intent(Intent.ACTION\_SET\_WALLPAPER),

mContext.getText(com.android.internal.R.string.chooser\_wallpaper)),

0, null, new UserHandle(serviceUserId)));

if (!mContext.bindServiceAsUser(intent, newConn,

Context.BIND\_AUTO\_CREATE | Context.BIND\_SHOWING\_UI,

new UserHandle(serviceUserId))) {

String msg = "Unable to bind service: "

+ componentName;

if (fromUser) {

throw new IllegalArgumentException(msg);

}

Slog.w(TAG, msg);

return false;

}

if (wallpaper.userId == mCurrentUserId && mLastWallpaper != null) {

detachWallpaperLocked(mLastWallpaper);

}

wallpaper.wallpaperComponent = componentName;

wallpaper.connection = newConn;

newConn.mReply = reply;

try {

if (wallpaper.userId == mCurrentUserId) {

if (DEBUG)

Slog.v(TAG, "Adding window token: " + newConn.mToken);

mIWindowManager.addWindowToken(newConn.mToken,

WindowManager.LayoutParams.TYPE\_WALLPAPER);

mLastWallpaper = wallpaper;

}

} catch (RemoteException e) {

}

} catch (RemoteException e) {

String msg = "Remote exception for " + componentName + "\n" + e;

if (fromUser) {

throw new IllegalArgumentException(msg);

}

Slog.w(TAG, msg);

return false;

}

return true;

}

---------------------

我们知道，当通过bindService启动一个Service时，就会回调ServiceConnection的onServiceConnected函数。看看实现了ServiceConnection接口的WallpaperConnection对onServiceConnected的实现。

  服务端的WallpaperService通过onBind返回一个IBinder对象，作为onServiceConnected的第二个参数传入，这个IBinder对象是一个IWallpaperServiceWrapper对象。asInterface调用返回一个IWallpaperService.Stub的代理对象并保存在mService中。attachServiceLocked

用来启动壁纸的Engine。Engine类提供了许多壁纸呈现的流程的回调接口及属性获取接口，并通过updateSurface来实现添加壁纸窗口和申请Surface的操作，当壁纸的属性发生变化时，我们可以在对应的流程回调接口加入例如drawFrame（画图渲染）等的操作。saveSettingsLocked用来将Engine启动过程中对WallpaperData的修改写回到配置文件wallpaper\_info.xml中

## WallMS. onServiceConnected

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

@Override

public final IBinder onBind(Intent intent) {

return new IWallpaperServiceWrapper(this);

}

1

2

3

4

/frameworks/base/services/core/java/com/android/server/wallpaper/WallpaperManagerService.java

void attachServiceLocked(WallpaperConnection conn, WallpaperData wallpaper) {

try {

conn.mService.attach(conn, conn.mToken,

WindowManager.LayoutParams.TYPE\_WALLPAPER, false,

wallpaper.width, wallpaper.height, wallpaper.padding);

} catch (RemoteException e) {

Slog.w(TAG, "Failed attaching wallpaper; clearing", e);

if (!wallpaper.wallpaperUpdating) {

bindWallpaperComponentLocked(null, false, false, wallpaper, null);

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

@Override

public void attach(IWallpaperConnection conn, IBinder windowToken,

int windowType, boolean isPreview, int reqWidth, int reqHeight, Rect padding) {

new IWallpaperEngineWrapper(mTarget, conn, windowToken,

windowType, isPreview, reqWidth, reqHeight, padding);

}

1

2

3

4

5

6

  IWallpaperEngineWrapper的构造函数中，发送了一个DO\_ATTACH的消息到WallpaperService的Looper的MessageQueue中。

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

IWallpaperEngineWrapper(WallpaperService context,

IWallpaperConnection conn, IBinder windowToken,

int windowType, boolean isPreview, int reqWidth, int reqHeight, Rect padding) {

mCaller = new HandlerCaller(context, context.getMainLooper(), this, true);

mConnection = conn;

mWindowToken = windowToken;

mWindowType = windowType;

mIsPreview = isPreview;

mReqWidth = reqWidth;

mReqHeight = reqHeight;

mDisplayPadding.set(padding);

Message msg = mCaller.obtainMessage(DO\_ATTACH);

mCaller.sendMessage(msg);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

public void executeMessage(Message message) {

switch (message.what) {

case DO\_ATTACH: {

try {

mConnection.attachEngine(this);

} catch (RemoteException e) {

Log.w(TAG, "Wallpaper host disappeared", e);

return;

}

Engine engine = onCreateEngine();

mEngine = engine;

mActiveEngines.add(engine);

engine.attach(this);

return;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

  WallpaperConnection将mEngine成员设为传过来的IWallpaperEngineWrapper对象。当我们还没调用attachEngine来attach一个Engine到WallpaperConnection时，而我们希望修改attach到Engine后改变Engine的宽度和高度（mDimensionsChanged），或边距（mPaddingChanged），则可以通过WallpaperManager#suggestDesiredDimensions（改变宽度和高度）和WallpaperManager#setDisplayPadding（改变边距）来实现，这样，这些改变属性的操作将会延迟到attach之后才会执行。WallpaperManager对象可以通过ContextImpl#getSystemService(Context.WINDOW\_SERVICE)获得.。

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

@Override

public void attachEngine(IWallpaperEngine engine) {

synchronized (mLock) {

mEngine = engine;

if (mDimensionsChanged) {

try {

mEngine.setDesiredSize(mWallpaper.width, mWallpaper.height);

} catch (RemoteException e) {

Slog.w(TAG, "Failed to set wallpaper dimensions", e);

}

mDimensionsChanged = false;

}

if (mPaddingChanged) {

try {

mEngine.setDisplayPadding(mWallpaper.padding);

} catch (RemoteException e) {

Slog.w(TAG, "Failed to set wallpaper padding", e);

}

mPaddingChanged = false;

}

}

}

## ImageWallpaper

ImageWallpaper类复写了onCreateEngine方法，创建了一个继承自Engine类的DrawableEngine。这个DrawableEngine将被保存在IWallpaperEngineWrapper的mEngine成员中。

/frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

@Override

public Engine onCreateEngine() {

mEngine = new DrawableEngine();

return mEngine;

}

1

2

3

4

5

6

  mSurfaceHolder是Engine的一个内部类BaseSurfaceHolder对象，其继承了BaseSurfaceHolder类，提供了管理Surface对象和保存Surface生命周期回调接口的能力。onCreate函数在子类DrawableEngine有实现，

用来初始化壁纸使用的Bitmap和使用的Bitmap及Surface的大小，见于updateSurfaceSize函数。最后调用updateSurface进行首次的添加Window,获取Surface和渲染的操作。

/frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

void attach(IWallpaperEngineWrapper wrapper) {

if (DEBUG) Log.v(TAG, "attach: " + this + " wrapper=" + wrapper);

if (mDestroyed) {

return;

}

mIWallpaperEngine = wrapper;

mCaller = wrapper.mCaller;

mConnection = wrapper.mConnection;

mWindowToken = wrapper.mWindowToken;

mSurfaceHolder.setSizeFromLayout();

mInitializing = true;

mSession = WindowManagerGlobal.getWindowSession();

mWindow.setSession(mSession);

mDisplayManager = (DisplayManager)getSystemService(Context.DISPLAY\_SERVICE);

mDisplayManager.registerDisplayListener(mDisplayListener, mCaller.getHandler());

mDisplay = mDisplayManager.getDisplay(Display.DEFAULT\_DISPLAY);

if (DEBUG) Log.v(TAG, "onCreate(): " + this);

onCreate(mSurfaceHolder);

mInitializing = false;

mReportedVisible = false;

updateSurface(false, false, false);

}

  getDefaultDisplaySize获取了屏幕的逻辑宽高。forgetLoadedWallpaper表示忘记原来的壁纸Bitmap，即取消一些变量对Bitmap的引用，防止Bitmap无法被回收造成内存泄露。updateWallpaperLocked用来获得壁纸对应的Bitmap及其宽高数据。我们在壁纸宽度和屏幕逻辑宽度，壁纸高度和屏幕逻辑高度中均选出一个较大值，作为Surface的宽高。

## openDefaultWallpaper

壁纸的选取地方有（以下按查找顺序排列）：1./data/system/users/{userid}/wallpaper；2.ro.config.wallpaper属性配置的壁纸路径；3.默认壁纸default\_wallpaper.jpg，存在于/frameworks/base/core/res/res/drawable-xxx/default\_wallpaper.jpg。

res/res/

public static InputStream openDefaultWallpaper(Context context, @SetWallpaperFlags int which) {

final String whichProp;

final int defaultResId;

if (which == FLAG\_LOCK) {

/\* Factory-default lock wallpapers are not yet supported

whichProp = PROP\_LOCK\_WALLPAPER;

defaultResId = com.android.internal.R.drawable.default\_lock\_wallpaper;

\*/

return null;

} else {

whichProp = PROP\_WALLPAPER;

defaultResId = com.android.internal.R.drawable.default\_wallpaper;

## Qita

Dsf

/frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

private void updateWallpaperLocked() {

Throwable exception = null;

try {

Log.d(TAG,"choose a bitmap for wallpaper.");

mBackground = null;

mBackgroundWidth = -1;

mBackgroundHeight = -1;

mBackground = mWallpaperManager.getBitmap();

mBackgroundWidth = mBackground.getWidth();

mBackgroundHeight = mBackground.getHeight();

} catch (RuntimeException e) {

exception = e;

} catch (OutOfMemoryError e) {

exception = e;

}

Log.v(TAG,"mBackgroundWidth is "+mBackgroundWidth+",mBackgroundHeight is "+mBackgroundHeight);

if (exception != null) {

mBackground = null;

mBackgroundWidth = -1;

mBackgroundHeight = -1;

// Note that if we do fail at this, and the default wallpaper can't

// be loaded, we will go into a cycle. Don't do a build where the

// default wallpaper can't be loaded.

Log.w(TAG, "Unable to load wallpaper!", exception);

try {

mWallpaperManager.clear();

} catch (IOException ex) {

// now we're really screwed.

Log.w(TAG, "Unable reset to default wallpaper!", ex);

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

  当强制layout或Window未创建或Surface未创建或属性改变或Window属性改变或需要重绘时，updateWallpaperLocked分为以下几步：1.更新Engine的对应属性为Surface的属性，更新Window属性；2.当Window尚未添加到WMS时（mCreated为false），使用addToDisplay添加窗口；3.relayout，申请Surface保存在mSurfaceHolder.mSurface中；4.根据具体的属性变化进行BaseSurfaceHolder中CallBack的回调和DrawableEngine的回调。在ImageWallpaper的实现中，会在onVisibilityChanged（可视性改变），onOffsetsChanged（壁纸窗口发生偏移），onSurfaceChanged（Surface属性改变），onSurfaceRedrawNeeded（壁纸窗口需要重绘）中调用drawFrame函数进行绘图渲染。

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

void updateSurface(boolean forceRelayout, boolean forceReport, boolean redrawNeeded) {

if (mDestroyed) {

Log.w(TAG, "Ignoring updateSurface: destroyed");

}

boolean fixedSize = false;

int myWidth = mSurfaceHolder.getRequestedWidth();

Log.v(TAG,"mRequestedWidth is "+myWidth);

if (myWidth <= 0) myWidth = ViewGroup.LayoutParams.MATCH\_PARENT;

else fixedSize = true;

int myHeight = mSurfaceHolder.getRequestedHeight();

Log.v(TAG,"mRequestedHeight is "+myHeight);

if (myHeight <= 0) myHeight = ViewGroup.LayoutParams.MATCH\_PARENT;

else fixedSize = true;

final boolean creating = !mCreated;//Window是否需要添加

final boolean surfaceCreating = !mSurfaceCreated;//Surface是否已得到

final boolean formatChanged = mFormat != mSurfaceHolder.getRequestedFormat();//像素格式是否改变

boolean sizeChanged = mWidth != myWidth || mHeight != myHeight;//Surafce大小是否改变

boolean insetsChanged = !mCreated;//边衬区域是否改变

final boolean typeChanged = mType != mSurfaceHolder.getRequestedType();//Surface buffer类型改变

final boolean flagsChanged = mCurWindowFlags != mWindowFlags ||

mCurWindowPrivateFlags != mWindowPrivateFlags;//Window的Flags或PrivateFlags是否改变

if (forceRelayout || creating || surfaceCreating || formatChanged || sizeChanged

|| typeChanged || flagsChanged || redrawNeeded

|| !mIWallpaperEngine.mShownReported) {

if (DEBUG) Log.v(TAG, "Changes: creating=" + creating

+ " format=" + formatChanged + " size=" + sizeChanged);

try {

mWidth = myWidth;

mHeight = myHeight;

mFormat = mSurfaceHolder.getRequestedFormat();

mType = mSurfaceHolder.getRequestedType();

//初始化传给WindowManagerService的WindowManager.LayoutParams参数

mLayout.x = 0;

mLayout.y = 0;

mLayout.width = myWidth;

mLayout.height = myHeight;

mLayout.format = mFormat;

mCurWindowFlags = mWindowFlags;

mLayout.flags = mWindowFlags

| WindowManager.LayoutParams.FLAG\_LAYOUT\_NO\_LIMITS

| WindowManager.LayoutParams.FLAG\_LAYOUT\_IN\_SCREEN

| WindowManager.LayoutParams.FLAG\_NOT\_FOCUSABLE

;

mCurWindowPrivateFlags = mWindowPrivateFlags;

mLayout.privateFlags = mWindowPrivateFlags;

mLayout.memoryType = mType;

mLayout.token = mWindowToken;

if (!mCreated) {

// Retrieve watch round and outset info

final WindowManager windowService = (WindowManager)getSystemService(

Context.WINDOW\_SERVICE);

TypedArray windowStyle = obtainStyledAttributes(

com.android.internal.R.styleable.Window);

final Display display = windowService.getDefaultDisplay();

final boolean shouldUseBottomOutset =

display.getDisplayId() == Display.DEFAULT\_DISPLAY;

if (shouldUseBottomOutset && windowStyle.hasValue(

R.styleable.Window\_windowOutsetBottom)) {

if (mOutsetBottom == null) mOutsetBottom = new TypedValue();

windowStyle.getValue(R.styleable.Window\_windowOutsetBottom,

mOutsetBottom);

} else {

mOutsetBottom = null;

}

mWindowIsRound = getResources().getBoolean(

com.android.internal.R.bool.config\_windowIsRound);

windowStyle.recycle();

// detect emulator

mIsEmulator = Build.HARDWARE.contains("goldfish");

mIsCircularEmulator = SystemProperties.getBoolean(

ViewRootImpl.PROPERTY\_EMULATOR\_CIRCULAR, false);

// Add window

mLayout.type = mIWallpaperEngine.mWindowType;

mLayout.gravity = Gravity.START|Gravity.TOP;

Log.v(TAG,"title is "+WallpaperService.this.getClass().getName());

mLayout.setTitle(WallpaperService.this.getClass().getName());

mLayout.windowAnimations =

com.android.internal.R.style.Animation\_Wallpaper;

mInputChannel = new InputChannel();

if (mSession.addToDisplay(mWindow, mWindow.mSeq, mLayout, View.VISIBLE,

Display.DEFAULT\_DISPLAY, mContentInsets, mStableInsets,

mInputChannel) < 0) {

Log.w(TAG, "Failed to add window while updating wallpaper surface.");

return;

}

mCreated = true;

mInputEventReceiver = new WallpaperInputEventReceiver(

mInputChannel, Looper.myLooper());

}

//上锁

mSurfaceHolder.mSurfaceLock.lock();

mDrawingAllowed = true;

if (!fixedSize) {

//宽高为非固定值的情况下，更新Surface边距(Surface与Window的边距）

mLayout.surfaceInsets.set(mIWallpaperEngine.mDisplayPadding);

} else {

//宽高为固定值的情况下，不设边距

mLayout.surfaceInsets.set(0, 0, 0, 0);

}

//relayout

final int relayoutResult = mSession.relayout(

mWindow, mWindow.mSeq, mLayout, mWidth, mHeight,

View.VISIBLE, 0, mWinFrame, mOverscanInsets, mContentInsets,

mVisibleInsets, mStableInsets, mConfiguration, mSurfaceHolder.mSurface);

if (DEBUG) Log.v(TAG, "New surface: " + mSurfaceHolder.mSurface

+ ", frame=" + mWinFrame);

int w = mWinFrame.width();

int h = mWinFrame.height();

if (!fixedSize) {

//宽高非固定值的情况下，更新各种边距值

final Rect padding = mIWallpaperEngine.mDisplayPadding;

w += padding.left + padding.right;

h += padding.top + padding.bottom;

mOverscanInsets.left += padding.left;

mOverscanInsets.top += padding.top;

mOverscanInsets.right += padding.right;

mOverscanInsets.bottom += padding.bottom;

mContentInsets.left += padding.left;

mContentInsets.top += padding.top;

mContentInsets.right += padding.right;

mContentInsets.bottom += padding.bottom;

mStableInsets.left += padding.left;

mStableInsets.top += padding.top;

mStableInsets.right += padding.right;

mStableInsets.bottom += padding.bottom;

}

if (mCurWidth != w) {

sizeChanged = true;

mCurWidth = w;

}

if (mCurHeight != h) {

sizeChanged = true;

mCurHeight = h;

}

insetsChanged |= !mDispatchedOverscanInsets.equals(mOverscanInsets);

insetsChanged |= !mDispatchedContentInsets.equals(mContentInsets);

insetsChanged |= !mDispatchedStableInsets.equals(mStableInsets);

mSurfaceHolder.setSurfaceFrameSize(w, h);

mSurfaceHolder.mSurfaceLock.unlock();

if (!mSurfaceHolder.mSurface.isValid()) {

reportSurfaceDestroyed();

if (DEBUG) Log.v(TAG, "Layout: Surface destroyed");

return;

}

boolean didSurface = false;

try {

mSurfaceHolder.ungetCallbacks();

if (surfaceCreating) {

//Surface创建成功后的操作

mIsCreating = true;

didSurface = true;

if (DEBUG) Log.v(TAG, "onSurfaceCreated("

+ mSurfaceHolder + "): " + this);

onSurfaceCreated(mSurfaceHolder);

SurfaceHolder.Callback callbacks[] = mSurfaceHolder.getCallbacks();

if (callbacks != null) {

for (SurfaceHolder.Callback c : callbacks) {

c.surfaceCreated(mSurfaceHolder);

}

}

}

redrawNeeded |= creating || (relayoutResult

& WindowManagerGlobal.RELAYOUT\_RES\_FIRST\_TIME) != 0;

if (forceReport || creating || surfaceCreating

|| formatChanged || sizeChanged) {

//Surface发生改变的操作

if (DEBUG) {

RuntimeException e = new RuntimeException();

e.fillInStackTrace();

Log.w(TAG, "forceReport=" + forceReport + " creating=" + creating

+ " formatChanged=" + formatChanged

+ " sizeChanged=" + sizeChanged, e);

}

if (DEBUG) Log.v(TAG, "onSurfaceChanged("

+ mSurfaceHolder + ", " + mFormat

+ ", " + mCurWidth + ", " + mCurHeight

+ "): " + this);

didSurface = true;

onSurfaceChanged(mSurfaceHolder, mFormat,

mCurWidth, mCurHeight);

SurfaceHolder.Callback callbacks[] = mSurfaceHolder.getCallbacks();

if (callbacks != null) {

for (SurfaceHolder.Callback c : callbacks) {

c.surfaceChanged(mSurfaceHolder, mFormat,

mCurWidth, mCurHeight);

}

}

}

if (insetsChanged) {

//边距发生改变的操作

mDispatchedOverscanInsets.set(mOverscanInsets);

mDispatchedContentInsets.set(mContentInsets);

mDispatchedStableInsets.set(mStableInsets);

final boolean isRound = (mIsEmulator && mIsCircularEmulator)

|| mWindowIsRound;

mFinalSystemInsets.set(mDispatchedOverscanInsets);

mFinalStableInsets.set(mDispatchedStableInsets);

if (mOutsetBottom != null) {

final DisplayMetrics metrics = getResources().getDisplayMetrics();

mFinalSystemInsets.bottom =

( (int) mOutsetBottom.getDimension(metrics) )

+ mIWallpaperEngine.mDisplayPadding.bottom;

}

WindowInsets insets = new WindowInsets(mFinalSystemInsets,

null, mFinalStableInsets, isRound);

onApplyWindowInsets(insets);

}

if (redrawNeeded) {

//需要重绘的操作

onSurfaceRedrawNeeded(mSurfaceHolder);

SurfaceHolder.Callback callbacks[] = mSurfaceHolder.getCallbacks();

if (callbacks != null) {

for (SurfaceHolder.Callback c : callbacks) {

if (c instanceof SurfaceHolder.Callback2) {

((SurfaceHolder.Callback2)c).surfaceRedrawNeeded(

mSurfaceHolder);

}

}

}

}

if (didSurface && !mReportedVisible) {

//Surface发生了改变且壁纸尚不可见

// This wallpaper is currently invisible, but its

// surface has changed. At this point let's tell it

// again that it is invisible in case the report about

// the surface caused it to start running. We really

// don't want wallpapers running when not visible.

if (mIsCreating) {

//属于创建Surface的情况

// Some wallpapers will ignore this call if they

// had previously been told they were invisble,

// so if we are creating a new surface then toggle

// the state to get them to notice.

if (DEBUG) Log.v(TAG, "onVisibilityChanged(true) at surface: "

+ this);

onVisibilityChanged(true);

}

if (DEBUG) Log.v(TAG, "onVisibilityChanged(false) at surface: "

+ this);

onVisibilityChanged(false);

}

} finally {

mIsCreating = false;

mSurfaceCreated = true;

if (redrawNeeded) {

//需要重绘的情况，一般发生在首次relayout或者首次获得Surface,需要显示一个新的窗口

mSession.finishDrawing(mWindow);

}

mIWallpaperEngine.reportShown();

}

} catch (RemoteException ex) {

}

if (DEBUG) Log.v(

TAG, "Layout: x=" + mLayout.x + " y=" + mLayout.y +

" w=" + mLayout.width + " h=" + mLayout.height);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

 &esmp;drawFrame作为壁纸服务的绘图渲染函数。

/frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

void drawFrame() {

try {

int newRotation = ((WindowManager) getSystemService(WINDOW\_SERVICE)).

getDefaultDisplay().getRotation();

// Sometimes a wallpaper is not large enough to cover the screen in one dimension.

// Call updateSurfaceSize -- it will only actually do the update if the dimensions

// should change

if (newRotation != mLastRotation) {

// Update surface size (if necessary)

//如果屏幕的旋转发生了变化，更新SUrface的大小

updateSurfaceSize(getSurfaceHolder());

}

SurfaceHolder sh = getSurfaceHolder();

final Rect frame = sh.getSurfaceFrame();

final int dw = frame.width();

final int dh = frame.height();

//Surface中的Window的frame的宽和高对比上次保存的此值是否有变化

boolean surfaceDimensionsChanged = dw != mLastSurfaceWidth

|| dh != mLastSurfaceHeight;

boolean redrawNeeded = surfaceDimensionsChanged || newRotation != mLastRotation;

if (!redrawNeeded && !mOffsetsChanged) {

//当Surface中的Window的frame的宽和高没有变化&旋转状态没有变化&壁纸偏移没有变化，直接返回

if (DEBUG) {

Log.d(TAG, "Suppressed drawFrame since redraw is not needed "

+ "and offsets have not changed.");

}

return;

}

mLastRotation = newRotation;

// Load bitmap if it is not yet loaded or if it was loaded at a different size

if (mBackground == null || surfaceDimensionsChanged) {

if (DEBUG) {

Log.d(TAG, "Reloading bitmap: mBackground, bgw, bgh, dw, dh = " +

mBackground + ", " +

((mBackground == null) ? 0 : mBackground.getWidth()) + ", " +

((mBackground == null) ? 0 : mBackground.getHeight()) + ", " +

dw + ", " + dh);

}

//没有壁纸图片或者当Surface中的Window的frame的宽和高有变化

mWallpaperManager.forgetLoadedWallpaper();

updateWallpaperLocked();

if (mBackground == null) {

if (DEBUG) {

Log.d(TAG, "Unable to load bitmap");

}

return;

}

if (DEBUG) {

if (dw != mBackground.getWidth() || dh != mBackground.getHeight()) {

Log.d(TAG, "Surface != bitmap dimensions: surface w/h, bitmap w/h: " +

dw + ", " + dh + ", " + mBackground.getWidth() + ", " +

mBackground.getHeight());

}

}

}

// Center the scaled image

//mScale表示壁纸图片的拉伸比例

mScale = Math.max(1f, Math.max(dw / (float) mBackground.getWidth(),

dh / (float) mBackground.getHeight()));

final int availw = dw - (int) (mBackground.getWidth() \* mScale);

final int availh = dh - (int) (mBackground.getHeight() \* mScale);

int xPixels = availw / 2;

int yPixels = availh / 2;

// Adjust the image for xOffset/yOffset values. If window manager is handling offsets,

// mXOffset and mYOffset are set to 0.5f by default and therefore xPixels and yPixels

// will remain unchanged

final int availwUnscaled = dw - mBackground.getWidth();

final int availhUnscaled = dh - mBackground.getHeight();

if (availwUnscaled < 0)

xPixels += (int) (availwUnscaled \* (mXOffset - .5f) + .5f);

if (availhUnscaled < 0)

yPixels += (int) (availhUnscaled \* (mYOffset - .5f) + .5f);

mOffsetsChanged = false;

mRedrawNeeded = false;

if (surfaceDimensionsChanged) {

mLastSurfaceWidth = dw;

mLastSurfaceHeight = dh;

}

if (!redrawNeeded && xPixels == mLastXTranslation && yPixels == mLastYTranslation) {

if (DEBUG) {

Log.d(TAG, "Suppressed drawFrame since the image has not "

+ "actually moved an integral number of pixels.");

}

return;

}

mLastXTranslation = xPixels;

mLastYTranslation = yPixels;

if (DEBUG) {

Log.d(TAG, "Redrawing wallpaper");

}

if (mIsHwAccelerated) {

//一般情况下都启动硬件加速

if (!drawWallpaperWithOpenGL(sh, availw, availh, xPixels, yPixels)) {

//OpenGL渲染失败则使用canvas绘制

drawWallpaperWithCanvas(sh, availw, availh, xPixels, yPixels);

}

} else {

drawWallpaperWithCanvas(sh, availw, availh, xPixels, yPixels);

}

} finally {

if (FIXED\_SIZED\_SURFACE && !mIsHwAccelerated) {

// If the surface is fixed-size, we should only need to

// draw it once and then we'll let the window manager

// position it appropriately. As such, we no longer needed

// the loaded bitmap. Yay!

// hw-accelerated renderer retains bitmap for faster rotation

mBackground = null;

mWallpaperManager.forgetLoadedWallpaper();

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

  关于壁纸偏移。首先关注的是mOffsetsChanged变量。在对壁纸的窗口进行relayout时，传入了一个继承自IWindow.Stub的BaseIWindow对象，里面实现了WMS对壁纸窗口进行调整后的回调。其中有resized（ContentInsets，VisibleInsets，Surface的大小改变或者Configuration改变时的回调），moved（窗口坐标改变的回调），dispatchAppVisibility（窗口可见性改变的回调），dispatchWallpaperOffsets（窗口偏移发生改变的回调）等。那mOffsetsChanged什么时候为true表示壁纸窗口偏移发生了改变呢？1.调用setDimensionHints来设置Engine的期望壁纸宽高时（没有设置则为传过来的WallpaperData记录的壁纸宽高值），因为壁纸窗口偏移的值需要根据Engine的期望壁纸宽高来决定，但是在SystemUI的ImageWallpaper的相应实现里没有利用到Engine的期望壁纸宽高。2.dispatchWallpaperOffsets回调来更新偏移值时。在Surface已创建的情况下，若壁纸窗口处于可见状态，onOffsetsChanged（ImageWallpaper的实现）会检查传入的xOffset和yOffset是否与mXOffset和mYOffset保存的值相同，若有一个不同，则认为是壁纸窗口偏移发生了变化，设置mOffsetsChanged为true。若壁纸窗口不处于可见状态，则直接设置mOffsetsChanged为true，而不必要更新mXOffset，mYOffset，更不需要调用drawFrame重画一帧,这些等到壁纸壁纸窗口可见以后再更新就可以了。3.dispatchAppVisibility回调更新壁纸窗口可见性时，如果在壁纸窗口不可见时有人申请了更新窗口偏移值（当时没更新mXOffset，mYOffset，只是设置mOffsetsChanged为true），则再壁纸窗口变得可见后再重新处理这些请求，见onOffsetsChanged的实现。

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

void doOffsetsChanged(boolean always) {

if (mDestroyed) {

return;

}

if (!always && !mOffsetsChanged) {

return;

}

float xOffset;

float yOffset;

float xOffsetStep;

float yOffsetStep;

boolean sync;

synchronized (mLock) {

xOffset = mPendingXOffset;

yOffset = mPendingYOffset;

xOffsetStep = mPendingXOffsetStep;

yOffsetStep = mPendingYOffsetStep;

sync = mPendingSync;

mPendingSync = false;

mOffsetMessageEnqueued = false;

}

if (mSurfaceCreated) {

if (mReportedVisible) {

if (DEBUG) Log.v(TAG, "Offsets change in " + this

+ ": " + xOffset + "," + yOffset);

final int availw = mIWallpaperEngine.mReqWidth-mCurWidth;

final int xPixels = availw > 0 ? -(int)(availw\*xOffset+.5f) : 0;

final int availh = mIWallpaperEngine.mReqHeight-mCurHeight;

final int yPixels = availh > 0 ? -(int)(availh\*yOffset+.5f) : 0;

onOffsetsChanged(xOffset, yOffset, xOffsetStep, yOffsetStep, xPixels, yPixels);

} else {

mOffsetsChanged = true;

}

}

if (sync) {

try {

if (DEBUG) Log.v(TAG, "Reporting offsets change complete");

mSession.wallpaperOffsetsComplete(mWindow.asBinder());

} catch (RemoteException e) {

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

/frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

@Override

public void onOffsetsChanged(float xOffset, float yOffset,

float xOffsetStep, float yOffsetStep,

int xPixels, int yPixels) {

if (DEBUG) {

Log.d(TAG, "onOffsetsChanged: xOffset=" + xOffset + ", yOffset=" + yOffset

+ ", xOffsetStep=" + xOffsetStep + ", yOffsetStep=" + yOffsetStep

+ ", xPixels=" + xPixels + ", yPixels=" + yPixels);

}

if (mXOffset != xOffset || mYOffset != yOffset) {

if (DEBUG) {

Log.d(TAG, "Offsets changed to (" + xOffset + "," + yOffset + ").");

}

mXOffset = xOffset;

mYOffset = yOffset;

mOffsetsChanged = true;

}

drawFrame();

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

/frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

void reportVisibility() {

if (!mDestroyed) {

boolean visible = mVisible

& mDisplay != null && mDisplay.getState() != Display.STATE\_OFF;

if (mReportedVisible != visible) {

mReportedVisible = visible;

if (DEBUG) Log.v(TAG, "onVisibilityChanged(" + visible

+ "): " + this);

if (visible) {

// If becoming visible, in preview mode the surface

// may have been destroyed so now we need to make

// sure it is re-created.

doOffsetsChanged(false);

updateSurface(false, false, false);

}

onVisibilityChanged(visible);

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

  但实际上壁纸的分辨率和屏幕的分辨率不相同，我们应该将壁纸画在哪个位置也是一个问题。有3种情形：1.壁纸可以完全包含屏幕；2.屏幕可以完全包含壁纸；3.壁纸和屏幕相交但不互相完全包含。

  对于壁纸可以完全包含屏幕的情况，就是根据mXOffset和mYOffset的值把屏幕的位置放在壁纸的适当位置，当mXOffset和mYOffset的值均为0.5时，屏幕放在壁纸的中央位置，壁纸坐标（left,top,right,bottom）为（-（屏幕宽-壁纸宽）/2,-（屏幕高-壁纸高）/2,（屏幕宽-壁纸宽）/2+壁纸宽，（屏幕高-壁纸高）/2+壁纸高），mXOffset的值从0到1增长时，会将壁纸往左边移动（假设屏幕位置不动），mYOffset的值从0到1增长时，会将壁纸往上边移动，也就是说，mXOffset和mYOffset的值越大，屏幕占用的区域就越接近壁纸的右下方；对于屏幕可以完全包含壁纸的情况，需要计算壁纸的拉伸值mScale（屏幕宽/壁纸宽，屏幕高/壁纸高中的最大值），屏幕宽高值分别减去壁纸的宽高乘以拉伸值得到的值得到availw和availh，这个时候availw和availh必然有一个为0，有一个为负数，则把壁纸放在负数对应的那条边的中间位置以屏幕边的中点位置进行以mScale值为比例的拉伸。对于壁纸和屏幕相交但不互相完全包含的情况，方法同屏幕可以完全包含壁纸的情况基本一致，都是先拉伸壁纸到恰好可以可以完全包含屏幕后，将壁纸贴合沿着拉伸后屏幕中长度还未和壁纸一致的边进行移动，直到屏幕恰好在在拉伸后壁纸中央。

## BUG

<https://blog.csdn.net/yh_dreamsky/article/details/39932111>

## REF

[Android壁纸服务WallpaperManagerService](https://blog.csdn.net/Invoker123/article/details/82382050)

# UI更新

Ds

在Android系统中，壁纸窗口和输入法窗口一样，都是一种特殊类型的窗口，而且它们都是喜欢和一个普通的Activity窗口缠绵在一起。大家可以充分地想象这样的一个3W场景：输入法窗口在上面，壁纸窗口在下面，Activity窗口夹在它们的中间。在前面一篇文章中，我们已经分析过输入法窗口是如何压在Activity窗口上面的了。在这篇文章中，我们就将继续分析壁纸窗口是如何贴在Activity窗口下面的。

 一个Activity窗口如果需要显示壁纸，那么它必须满足以下两个条件：

1. 背景是半透明的，例如，它在AndroidManifest.xml文件中的android:theme属性设置为Theme.Translucent：
2. **<activity** android:name=".WallpaperActivity"
3. android:theme="@android:style/Theme.Translucent"**>**
4. ......
5. **</activity>**

2. 窗口属性中的WindowManager.LayoutParams.FLAG\_SHOW\_WALLPAPER位设置为1：

getWindow().addFlags(WindowManager.LayoutParams.FLAG\_SHOW\_WALLPAPER);

图2中的对象的关系如下所示：

        1. 在ActivityManagerService服务内部的Activity组件堆栈顶端的ActivityRecord对象N描述的是系统当前激活的Activity组件。

        2. ActivityRecord对象N在WindowManagerService服务内部的窗口令牌列表顶端对应有一个AppWindowToken对象N。

        3. AppWindowToken对象N在WindowManagerService服务内部的窗口堆栈中对应有一个WindowState对象N，用来描述系统当前激活的Activity组件窗口。

        4. WindowState对象N下面有一个WindowState对象WP，用来描述系统中的壁纸窗口。

        5. 系统中的壁纸窗口在WindowManagerService服务内部中对应的窗口令牌是由WindowToken对象WP来描述的。

        6. WindowToken对象WP在WallpaperManagerService服务中对应有一个Binder对象。

        总的来说，就是图2描述了系统当前激活的Activity窗口需要显示壁纸的情景。WindowManagerService服务的职能之一就是要时刻关注系统中是否有窗口需要显示壁纸。WindowManagerService服务一旦发现有窗口需要显示壁纸，那么就会调整壁纸窗口在窗口堆栈中的位置，使得它放置在需要显示壁纸的窗口的下面。此外，需要显示壁纸的窗口还可以设置壁纸窗口在X轴和Y轴上的偏移位置，以便可以将壁纸窗口的某一部分指定为它的背景。

        接下来，我们就首先分析两个需要调整壁纸窗口在窗口堆栈中的位置的情景，然后再分析壁纸窗口在X轴和Y轴上的偏移位置的调整过程，最后分析壁纸窗口在窗口堆栈中的位置调整过程。

        一. 调整壁纸窗口在窗口堆栈中的位置的情景

        第一个需要调整壁纸窗口在窗口堆栈中的位置的情景是增加一个窗口到WindowManagerService服务去的时候。从前面[Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/8275938)一文可以知道，增加一个窗口到WindowManagerService服务最终是通过调用WindowManagerService类的成员函数addWindow来实现的。接下来我们就主要分析这个函数中与壁纸窗口调整相关的逻辑，如下所示：

1. addWindowToListInOrderLocked(win, **true**);
2. **if** (attrs.type == TYPE\_WALLPAPER) {
3. ......
4. adjustWallpaperWindowsLocked();

dfg

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        如果当前增加到WindowManagerService服务来的是一个壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，那么就要求与该壁纸窗口所对应的类型为WindowToken的窗口令牌已经存在，否则的话，WindowManagerService类的成员函数addWindow就会直接返回一个错误码WindowManagerImpl.ADD\_BAD\_APP\_TOKEN给调用者。这个类型为WindowToken的窗口令牌是WallpaperManagerService服务请求WindowManagerService服务创建的，即调用WindowManagerService类的成员函数addWindowToken来创建的，具体可以参考前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)一文。

        如果当前增加到WindowManagerService服务来的既不是一个输入法窗口，也不是一个输入法对话框，那么WindowManagerService类的成员函数addWindow就会调用另外一个成员函数addWindowToListInOrderLocked来将前面为它所创建的一个WindowState对象win增加到窗口堆栈的合适位置上去。

        如果前面增加到窗口堆栈中的窗口是一个壁纸窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_WALLPAPER，或者是一个需要显示壁纸的窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的值的FLAG\_SHOW\_WALLPAPER位等于1，那么就说明需要调整壁纸窗口在窗口堆栈中的位置，使得它位于需要显示壁纸的窗口的下面，这是通过调用WindowManagerService类的成员函数adjustWallpaperWindowsLocked来实现的。

        最后，由于增加了一个窗口到窗口堆栈中，以及窗口堆栈的窗口位置发生了变化，因此，就需要重新各个窗口的Z轴位置，这是通过调用WindowManagerService类的成员函数assignLayersLocked来实现的。

        在这个情景中，主要涉及到了WindowManagerService类的三个成员函数addWindowToListInOrderLocked、adjustWallpaperWindowsLocked和assignLayersLocked，其中，成员函数addWindowToListInOrderLocked的实现可以参考前面前面[Android窗口管理服务WindowManagerService组织窗口的方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908" \t "_blank)一文，成员函数assignLayersLocked的实现在接下来的一篇文章中再分析，本文主要是关注成员函数

## adjustWallpaperWindowsLocked

第二个需要调整壁纸窗口在窗口堆栈中的位置的情景是一个应用程序进程请求WindowManagerService服务重新布局一个窗口的时候。从前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文可以知道，应用程序进程请求WindowManagerService服务重新布局一个窗口最终是通过调用WindowManagerService类的成员函数relayoutWindow来实现的。接下来我们就主要分析这个函数中与壁纸窗口调整相关的逻辑，如下所示：

这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        应用程序进程在请求WindowManagerService服务重新布局一个窗口的时候，这个窗口的一些布局参数可能会发生变化，而这些变化可能会引发系统的壁纸窗口在窗口堆栈中的位置发生变化。如果系统的壁纸窗口在窗口堆栈中的位置发生了变化，那么就需要调整它们在窗口堆栈中的位置。

        WindowManagerService类的成员函数relayoutWindow首先调用根据参数session和client来调用另外一个成员函数windowForClientLocked，以便可以获得用来描述要重新布局的窗口的一个WindowState对象win。

        WindowState对象win的成员变量mViewVisibility描述的是窗口上一次布局时的可见性，而参数viewVisibility描述的是窗口当前的可见性，当它们的值不相等时，就意味着窗口的可见性发生了变化。在窗口的可见性发生了变化的情况下，如果正在请求重新布局的是一个需要显示壁纸的窗口，即WindowState对象win的成员变量mAttrs所指向的是一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SHOW\_WALLPAPER位等于1，那么就说明可能需要调整壁纸窗口在窗口堆栈中的位置，以便它可以位于WindowState对象win所描述的窗口的下面，这时候变量wallpaperMayMove的值就会等于true。

        WindowManagerService类的成员函数relayoutWindow执行了一系列的其它操作之后，接下来就会判断变量wallpaperMayMove的值是否等于true。如果等于true的话，那么就会调用另外一个成员函数adjustWallpaperWindowsLocked来调整壁纸窗口在窗口堆栈中的位置，以便它可以位于需要显示壁纸的窗口的下面。WindowManagerService类的成员函数adjustWallpaperWindowsLocked的返回值是一个整数，当它的ADJUST\_WALLPAPER\_LAYERS\_CHANGED位等于1的时候，就说明壁纸窗口在窗口堆栈的位置发生了变化，于是就会将变量assignLayers的值设置为true，以便接下来可以调用WindowManagerService类的成员函数assignLayersLocked来重新计算系统中各个窗品的Z轴位置。

        变量displayed用来描述WindowState对象win所描述的窗口在当前布局中是由不可见变为可见的。在满足以下的条件之下，WindowState对象win所描述的窗口是由不可见变为可见的：

        1. 参数viewVisibility的值等于View.VISIBLE，即应用程序进程请求显示WindowState对象win所描述的窗口。

        2. WindowState对象win描述的是一个Activity窗口，即它的成员变量mAppToken不等于null，并且它所指向的AppWindowToken对象的成员变量clientHidden的值等于false，即WindowState对象win的窗口所对应的Activity组件当前是可见的。注意，如果WindowState对象win描述的不是一个Activity窗口，即它的成员变量mAppToken等于null，那么就可以忽略条件2。

        3. WindowState对象win所描述的窗口上一次是不可见的，即调用WindowState对象win的成员函数isVisibleLw的返回值等于false。

        此外，在满足条件1和条件2的情况下，如果WindowState对象win所描述的窗口的像素格式发生了变化，那么就需要将该窗口的绘图表面销毁掉，然后再重新创建一个，这时候也会认为该窗口由不可见变为了可见。

        参数attrs所指向的一个WindowManager.LayoutParams对象是用来保存WindowState对象win所描述的窗口在当前布局中所使用的布局参数的，而WindowState对象win的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象是用来保存WindowState对象win所描述的窗口在上一次布局所使用的布局参数的。在将参数attrs所指向的一个WindowManager.LayoutParams对象的内容拷贝到WindowState对象win的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的过程中，如果某些布局参数发生了变化，那么就会记录在变量attrChanges中。当变量attrChanges的WindowManager.LayoutParams.FORMAT\_CHANGED位等于1时，就说明WindowState对象win所描述的窗口的像素格式发生了变化，因此，WindowManagerService类的成员函数relayoutWindow就会调用WindowState对象win的成员函数destroySurfaceLocked来销毁该窗口的绘图表面，并且将变量displayed的值设置为true。

        WindowManagerService类的成员函数relayoutWindow调用另外一个成员函数performLayoutAndPlaceSurfacesLocked来对WindowState对象win所描述的窗口进行了布局之后，如果发现变量displayed的值等于true，并且WindowState对象win描述的是一个壁纸窗口，即它的成员变量mIsWallpaper的值等于true，那么还需要调用另外一个成员函数updateWallpaperOffsetLocked来重新计算该壁纸窗口在X轴和Y轴上的偏移位置，以便可以将它的某一部分区域指定在需要显示壁纸的窗口的背景。

        在这个情景中，主要涉及到了WindowManagerService类的四个成员函数adjustWallpaperWindowsLocked、updateWallpaperOffsetLocked、performLayoutAndPlaceSurfacesLocked和assignLayersLocked，其中，成员函数performLayoutAndPlaceSurfacesLocked的实现框架可以参考前面[Android窗口管理服务WindowManagerService计算Activity窗口大小的过程分析](http://blog.csdn.net/luoshengyang/article/details/8479101)一文，成员函数assignLayersLocked的实现如上所述在接下来的一篇文章中再分析，本文主要是关注成员函数adjustWallpaperWindowsLocked和updateWallpaperOffsetLocked的实现。

        从上面的分析就可以知道，在布局一个窗口的过程中，可能需要调用WindowManagerService类的成员函数updateWallpaperOffsetLocked和adjustWallpaperWindowsLocked来调整壁纸窗口在X轴和Y轴上的偏移位置和在窗口堆栈中的位置。接下来我们就分别分析壁纸窗口在X轴和Y轴上的偏移位置和在窗口堆栈中的位置的调整过程。

        二. 调整壁纸窗口在X轴和Y轴上的偏移位置

        壁纸窗口的大小是可以大于屏幕大小的。在这种情况下，需要显示壁纸的Activity窗口就需要指定壁纸在X轴和Y轴上的偏移位置，以便可以将壁纸的某一部分作为窗口的背景。

        假设壁纸窗口的大小为（WallpaperWidth, WallpaperHeight），屏幕的大小为（DisplayWidth, DisplayHeight），并且壁纸在X轴和Y轴上的偏移位置为WallpaperX和WallpaperY，其中，WallpaperWidth > DisplayWidth，WallpaperHeight > DisplayHeight，0.0 <= WallpaperX <= 1.0，0.0 <= WallpaperY <= 1.0，如图3所示：

图3 指定壁纸窗口在X轴和Y轴上的偏移位置

         这时候壁纸窗口在X轴和Y轴上的偏移位置的绝对值XOffset和YOffset就分别等于（WallpaperWidth - DisplayWidth）\* WallpaperX和（WallpaperHeight - DisplayHeight）\* WallpaperY。这意味道着：

        1. 当WallpaperX = WallpaperY = 0.0时，取壁纸窗口的左上角区域作为窗口背景。

        2. 当WallpaperX = WallpaperY = 0.5时，取壁纸窗口的中间区域作为窗口背景。

        3. 当WallpaperX = WallpaperY = 1.0时，取壁纸窗口的右下角区域作为窗口背景。

        除了使用WallpaperX和WallpaperY来描述壁纸窗口在X轴和Y轴上的偏移位置之外，WindowManagerService服务还使用WallpaperXStep和WallpaperYStep来描述壁纸窗口跨越了多少个虚拟屏幕。例如，假设一个Activity窗口在X轴上有3个虚拟屏幕，即它的实际宽度是屏幕宽度的3倍，而在Y轴上有一个屏幕，即它的实际高度刚好等于屏幕高度，并且壁纸窗口的宽度也刚好是屏幕宽度的3倍，而高度也刚好是等于屏幕高度，那么WallpaperXStep和WallpaperYStep的值就可以分别指定为0.5和0，这意味着：

        1. 第1个虚拟屏幕取壁纸窗口的左边三分之一的区域作为窗口背景，相当于是将壁纸窗口在X轴和Y轴上的偏移位置WallpaperX和WallpaperY的值分别设置为0.0和0.0。

        2. 第2个虚拟屏幕取壁纸窗口的中间三分之一的区域作为窗口背景，相当于是将壁纸窗口在X轴和Y轴上的偏移位置WallpaperX和WallpaperY的值分别设置为0.5和0.0。

        3. 第3个虚拟屏幕取壁纸窗口的右边三分之一的区域作为窗口背景，相当于是将壁纸窗口在X轴和Y轴上的偏移位置WallpaperX和WallpaperY的值分别设置为1.0和0.0。

        一般地，如果一个Activity窗口在X轴上有N个虚拟屏幕，而在Y轴上有M个虚拟屏幕，那么它就会将壁纸窗口的WallpaperXStep和WallpaperYStep值分别设置为1.0 / (N - 1)和1.0 / (M - 1)。对于WindowManagerService服务来说，它并不关心壁纸窗口的WallpaperXStep和WallpaperYStep值，而只关心壁纸窗口的WallpaperX和WallpaperY值，因为通过后两者，它就可以知道怎么显示壁纸窗口了。壁纸窗口的WallpaperXStep和WallpaperYStep值是用来传递给提供壁纸的服务的。提供壁纸的服务一旦知道壁纸窗口的WallpaperXStep和WallpaperYStep值是多少，就可以知道当前需要显示避纸的窗口有多少个虚拟屏幕。

        上面提到的与壁纸窗口在X轴和Y轴上的偏移位置相关的六个状态WallpaperX、WallpaperY、WallpaperXStep、WallpaperYStep、XOffset和YOffset由WindowManagerService服务来统一维护，它们分别对应于WindowState类的六个成员变量mWallpaperX、mWallpaperY、mWallpaperXStep、mWallpaperYStep、mXOffset和mYOffset，如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **float** mLastWallpaperX = -1;
6. **float** mLastWallpaperY = -1;
7. **float** mLastWallpaperXStep = -1;
8. **float** mLastWallpaperYStep = -1;
9. ......
11. **private** **final** **class** WindowState **implements** WindowManagerPolicy.WindowState {
12. ......
14. // If a window showing a wallpaper: the requested offset for the
15. // wallpaper; if a wallpaper window: the currently applied offset.
16. **float** mWallpaperX = -1;
17. **float** mWallpaperY = -1;
19. // If a window showing a wallpaper: what fraction of the offset
20. // range corresponds to a full virtual screen.
21. **float** mWallpaperXStep = -1;
22. **float** mWallpaperYStep = -1;
24. // Wallpaper windows: pixels offset based on above variables.
25. **int** mXOffset;
26. **int** mYOffset;
28. ......
29. }
31. ......
32. }

        这段代码定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        此外，WindowManagerService类还使用四个成员变量mLastWallpaperX、mLastWallpaperY、mLastWallpaperXStep和mLastWallpaperYStep来记录壁纸窗口上一次所使用的WallpaperX、WallpaperY、WallpaperXStep和WallpaperYStep值。

        在Android系统中，提供壁纸功能的组件叫做WallpaperService，它是一个Service组件，是由壁纸管理服务WallpaperManagerService负责启动的。WallpaperService有两个内部类BaseIWindow和Engine，其中，BaseIWindow是一个实现了IWindow接口的Binder本地对象类，用来和WindowManagerService服务通信，而Engine是一个真正用来实现壁纸功能的类。当一个Activity窗口需要指定壁纸窗口的某一部分区域作为它的背景时，它就会通过WallpaperManager类来通知WallpaperService设置壁纸窗口在X轴和Y轴上的偏移位置，这个过程如图4所示：

图4 壁纸窗口在X轴和Y轴上的偏移位置的设置过程

        这个过程大概如下所示：

        Step 1. 需要显示壁纸的Activity组件调用WallpaperManager类的成员函数setWallpaperOffsetSteps来设置壁纸窗口的WallpaperXStep和WallpaperYStep值。

        Step 2. 需要显示壁纸的Activity组件调用WallpaperManager类的成员函数setWallpaperOffsets来设置壁纸窗口的WallpaperX和WallpaperY值。

        Step 3. 一个类型为Session的Binder代理对象的成员函数setWallpaperPosition会被调用来通知WindowManagerService服务来重新计算壁纸窗口在X轴和Y轴上的偏移位置，传递的参数包括在Step 1和Step 2中所设置的WallpaperXStep、WallpaperYStep、WallpaperX和WallpaperY四个值。

        Step 4. WindowManagerService类的成员函数setWindowWallpaperPositionLocked会被调用来保存从前面Step 3传递过来的WallpaperXStep、WallpaperYStep、WallpaperX和WallpaperY值。

        Step 5. WindowManagerService类的成员函数updateWallpaperOffsetLocked会被调用来计算壁纸窗口在X轴和Y轴上的偏移位置的绝对值XOffset和YOffset，是根据壁纸窗口的大小（WallpapperWidth, WallpaperHeight）、屏幕的大小（DisplayWidth, DisplayHeight），以及保存在前面Step 4中的WallpaperX和WallpaperY来计算的。

        Step 6. 在WallpaperService类内部的一个BaseIWindow对象的成员函数dispatchWallpaperOffsets会被调用来通知WallpaperService服务，壁纸窗口在X轴和Y轴上的偏移位置发生改变了，传递过来的参数包括壁纸窗口的XOffset、YOffset、WallpaperXStep和WallpaperYStep值。

        Step 7. 在WallpaperService类内部的一个Engine对象的成员函数doOffsetsChanged会被调用来处理壁纸窗口在X轴和Y轴上的偏移位置变化事件。

        Step 8. Engine类的成员函数doOffsetsChanged会调用另外一个成员函数onOffsetsChanged来分发壁纸窗口在X轴和Y轴上的偏移位置变化事件。Engine类的成员函数onOffsetsChanged一般是由其子类来重写的，以便子类可以实现自己的壁纸效果。

        本文不打算详细这八个步骤，而主要关注Step 3、Step 4和Step 5这三步是如何计算壁纸窗口在X轴和Y轴上的偏移位置的，即主要关注Session类的成员函数setWallpaperPosition，以及WindowManagerService类的成员函数setWindowWallpaperPositionLocked和updateWallpaperOffsetLocked的实现。

         Session类的成员函数setWallpaperPosition的实现如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **class** Session **extends** IWindowSession.Stub
6. **implements** IBinder.DeathRecipient {
7. ......
9. **public** **void** setWallpaperPosition(IBinder window, **float** x, **float** y, **float** xStep, **float** yStep) {
10. **synchronized**(mWindowMap) {
11. **long** ident = Binder.clearCallingIdentity();
12. **try** {
13. setWindowWallpaperPositionLocked(
14. windowForClientLocked(**this**, window, **true**),
15. x, y, xStep, yStep);
16. } **finally** {
17. Binder.restoreCallingIdentity(ident);
18. }
19. }
20. }
22. ......
23. }
25. ......
26. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        Session类的成员函数setWallpaperPosition首先调用WindowManagerService类的成员函数windowForClientLocked来找到与参数window所对应的一个WindowState对象，这个WindowState对象描述的是要改变壁纸窗口位置的窗口，接着再调用WindowManagerService类的另外一个成员函数setWindowWallpaperPositionLocked来执行设置壁纸窗口在X轴和Y轴的偏移位置的操作。

        WindowManagerService类的成员函数setWindowWallpaperPositionLocked的实现如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **void** setWindowWallpaperPositionLocked(WindowState window, **float** x, **float** y,
6. **float** xStep, **float** yStep) {
7. **if** (window.mWallpaperX != x || window.mWallpaperY != y)  {
8. window.mWallpaperX = x;
9. window.mWallpaperY = y;
10. window.mWallpaperXStep = xStep;
11. window.mWallpaperYStep = yStep;
12. **if** (updateWallpaperOffsetLocked(window, **true**)) {
13. performLayoutAndPlaceSurfacesLocked();
14. }
15. }
16. }
18. ......
19. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类的成员函数setWindowWallpaperPositionLocked首先检查参数window所描述的WindowState对象上一次所设置的壁纸窗口的偏移位置与参数x和y所描述的偏移位置是否不一样。如果不一样的话，那么就会分别将参数x、y、xStep和yStep分别保存在参数window所描述的WindowState对象的成员变量mWallpaperX、mWallpaperY、mWallpaperXStep和mWallpaperYStep中，并且调用WindowManagerService类的成员函数updateWallpaperOffsetLocked来更新系统中的壁纸窗口的偏移位置。

        如果WindowManagerService类的成员函数updateWallpaperOffsetLocked的返回值等于true，那么就说明它更新了系统中的壁纸窗口的偏移位置，因此，就需要调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来刷新系统的UI。

        接下来我们继续分析WindowManagerService类的成员函数updateWallpaperOffsetLocked的实现，如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **final** ArrayList<WindowToken> mWallpaperTokens = **new** ArrayList<WindowToken>();
7. // If non-null, this is the currently visible window that is associated
8. // with the wallpaper.
9. WindowState mWallpaperTarget = **null**;
10. ......
12. **boolean** updateWallpaperOffsetLocked(WindowState changingTarget, **boolean** sync) {
13. **final** **int** dw = mDisplay.getWidth();
14. **final** **int** dh = mDisplay.getHeight();
16. **boolean** changed = **false**;
18. WindowState target = mWallpaperTarget;
19. **if** (target != **null**) {
20. **if** (target.mWallpaperX >= 0) {
21. mLastWallpaperX = target.mWallpaperX;
22. } **else** **if** (changingTarget.mWallpaperX >= 0) {
23. mLastWallpaperX = changingTarget.mWallpaperX;
24. }
25. **if** (target.mWallpaperY >= 0) {
26. mLastWallpaperY = target.mWallpaperY;
27. } **else** **if** (changingTarget.mWallpaperY >= 0) {
28. mLastWallpaperY = changingTarget.mWallpaperY;
29. }
30. }
32. **int** curTokenIndex = mWallpaperTokens.size();
33. **while** (curTokenIndex > 0) {
34. curTokenIndex--;
35. WindowToken token = mWallpaperTokens.get(curTokenIndex);
36. **int** curWallpaperIndex = token.windows.size();
37. **while** (curWallpaperIndex > 0) {
38. curWallpaperIndex--;
39. WindowState wallpaper = token.windows.get(curWallpaperIndex);
40. **if** (updateWallpaperOffsetLocked(wallpaper, dw, dh, sync)) {
41. wallpaper.computeShownFrameLocked();
42. changed = **true**;
43. // We only want to be synchronous with one wallpaper.
44. sync = **false**;
45. }
46. }
47. }
49. **return** changed;
50. }
52. ......
53. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        当WindowManagerService类的成员变量mWallpaperTarget的值不等于null时，它所指向的一个WindowState对象描述的是系统当前可见的并且需要显示壁纸的窗口。在这种情况下，要将这个WindowState对象当前正在使用的壁纸窗口在X轴和Y轴上的偏移位置分别保存在WindowManagerService类的成员变量mLastWallpaperX和mLastWallpaperY，以便接下来可以用来计算壁纸窗口在X轴和Y轴上的偏移位置的绝对值。

        注意，如果WindowManagerService类的成员变量mWallpaperTarget所指向的一个WindowState对象的成员变量mWallpaperX（mWallpaperY）的值小于0，那么就说明这个WindowState对象所描述的窗口还没有设置过壁纸窗口在X轴上（Y轴上）的偏移位置，这时候就需要将参数changingTarget所指向的一个WindowState对象的成员变量mWallpaperX（mWallpaperY）的值保存在WindowManagerService类的成员变量mLastWallpaperX（mLastWallpaperY）中，前提也是它的值大于等于0，即它描述的是一个有效的偏移值。

        WindowManagerService类的成员变量mWallpaperTokens保存的是一系列与壁纸相关的窗口令牌，与这些窗口令牌所对应的窗口就是系统当前所设置的壁纸窗口。WindowManagerService类的成员函数updateWallpaperOffsetLocked依次调用另外一个四参数版本的成员函数updateWallpaperOffsetLocked来更新系统当前所设置的每一个壁纸窗口在X轴和Y轴上的偏移位置。

        注意， WindowManagerService类的四个参数版本的成员函数updateWallpaperOffsetLocked的最后一个参数sync是一个布尔值，用来表示在更新壁纸窗口在X轴和Y轴上的偏移位置的时候，是否需要同步等待提供壁纸窗口的服务处理完成壁纸窗口在X轴和Y轴上的偏移位置变化事件。参数sync本身也是由两个参数版本的成员函数updateWallpaperOffsetLocked的调用者传进来的，它的值即使等于true，两个参数版本的成员函数updateWallpaperOffsetLocked也只会同步等待提供第一个壁纸窗口的服务处理完成壁纸窗口在X轴和Y轴上的偏移位置变化事件。

        WindowManagerService类的四个参数版本的成员函数updateWallpaperOffsetLocked的实现如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **boolean** updateWallpaperOffsetLocked(WindowState wallpaperWin, **int** dw, **int** dh,
6. **boolean** sync) {
7. **boolean** changed = **false**;
8. **boolean** rawChanged = **false**;
9. **float** wpx = mLastWallpaperX >= 0 ? mLastWallpaperX : 0.5f;
10. **float** wpxs = mLastWallpaperXStep >= 0 ? mLastWallpaperXStep : -1.0f;
11. **int** availw = wallpaperWin.mFrame.right-wallpaperWin.mFrame.left-dw;
12. **int** offset = availw > 0 ? -(**int**)(availw\*wpx+.5f) : 0;
13. changed = wallpaperWin.mXOffset != offset;
14. **if** (changed) {
15. ......
16. wallpaperWin.mXOffset = offset;
17. }
18. **if** (wallpaperWin.mWallpaperX != wpx || wallpaperWin.mWallpaperXStep != wpxs) {
19. wallpaperWin.mWallpaperX = wpx;
20. wallpaperWin.mWallpaperXStep = wpxs;
21. rawChanged = **true**;
22. }
24. **float** wpy = mLastWallpaperY >= 0 ? mLastWallpaperY : 0.5f;
25. **float** wpys = mLastWallpaperYStep >= 0 ? mLastWallpaperYStep : -1.0f;
26. **int** availh = wallpaperWin.mFrame.bottom-wallpaperWin.mFrame.top-dh;
27. offset = availh > 0 ? -(**int**)(availh\*wpy+.5f) : 0;
28. **if** (wallpaperWin.mYOffset != offset) {
29. ......
30. changed = **true**;
31. wallpaperWin.mYOffset = offset;
32. }
33. **if** (wallpaperWin.mWallpaperY != wpy || wallpaperWin.mWallpaperYStep != wpys) {
34. wallpaperWin.mWallpaperY = wpy;
35. wallpaperWin.mWallpaperYStep = wpys;
36. rawChanged = **true**;
37. }
39. **if** (rawChanged) {
40. **try** {
41. ......
42. **if** (sync) {
43. mWaitingOnWallpaper = wallpaperWin;
44. }
45. wallpaperWin.mClient.dispatchWallpaperOffsets(
46. wallpaperWin.mWallpaperX, wallpaperWin.mWallpaperY,
47. wallpaperWin.mWallpaperXStep, wallpaperWin.mWallpaperYStep, sync);
48. **if** (sync) {
49. **if** (mWaitingOnWallpaper != **null**) {
50. **long** start = SystemClock.uptimeMillis();
51. **if** ((mLastWallpaperTimeoutTime+WALLPAPER\_TIMEOUT\_RECOVERY)
52. < start) {
53. **try** {
54. ......
55. mWindowMap.wait(WALLPAPER\_TIMEOUT);
56. } **catch** (InterruptedException e) {
57. }
58. ......
59. **if** ((start+WALLPAPER\_TIMEOUT)
60. < SystemClock.uptimeMillis()) {
61. ......
62. mLastWallpaperTimeoutTime = start;
63. }
64. }
65. mWaitingOnWallpaper = **null**;
66. }
67. }
68. } **catch** (RemoteException e) {
69. }
70. }
72. **return** changed;
73. }
75. ......
76. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类的四个参数版本的成员函数updateWallpaperOffsetLocked首先计算参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置，接着再向提供该壁纸窗口的服务发向一个壁纸窗口在X轴和Y轴上的偏移位置变化事件通知。

        参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置的计算过程是一样的，这里我们只要结合前面的图3来分析壁纸窗口在X轴上的偏移位置的计算过程。

        在图3中，壁纸窗口的WallpaperX、WallpaperXStep、WallpaperWidth和DisplayWidth值分别等于这里的mLastWallpaperX、mLastWallpaperXStep、wallpaperWin.mFrame.right - wallpaperWin.mFrame.left和dw。有了这些值之后，就可以计算得到参数wallpaper所描述的壁纸窗口在X轴上的偏移位置的绝对值XOffset了。

        如果计算得到的XOffset、WallpaperX、WallpaperXStep的值与原来保存在参数wallpaper所指向的一个WindowState对象的成员变量mXOffset、mWallpaperX、mWallpaperXStep的值不相等，那么就会将计算得到的XOffset、WallpaperX、WallpaperXStep的值分别保存在这个WindowState对象的成员变量mXOffset、mWallpaperX、mWallpaperXStep，并且相应地将变量changed和rawChanged的值设置为true，表示参数wallpaper所描述的壁纸窗口在X轴上的偏移位置发生了变化。

        有四个地方需要注意：

        1. 当mLastWallpaperX的值小于0的时候，那么就说明系统中的壁纸窗口还没有被设置一个有效的X轴偏移位置，这时候计算壁纸窗口在X轴上的偏移位置所采用的WallpaperX值就会取为0.5，即默认将壁纸窗口的中间区域指定为需要显示壁纸的窗口的背景。

        2. 当mLastWallpaperXStep的值小于0的时候，那么就说明需要显示壁纸的窗口还没有告诉WindowManagerService服务它有多少个虚拟屏幕，这时候就会将壁纸窗口的WallpaperXStep值设置为-1.0，用来告诉提供壁纸窗口的服务，需要显示壁纸的窗口没有指定虚拟屏幕的个数。

        3. 当壁纸窗口的宽度小于等于屏幕宽度的时候，即变量availw的值小于等于0的时候，那么就说明不需要设置壁纸窗口在X轴上的偏移位置，也就是说，这时候壁纸窗口在X轴上的偏移位置始终保持为0。

        4. 当壁纸窗口的宽度大于屏幕宽度的时候，即变量availw的值大于0的时候，壁纸窗口在X轴上的偏移值等于availw \* wps，加上0.5是为了向上取整，向上取整后需要取反，因为负数才能正确表达出壁纸窗口相对屏幕的偏移。

        计算完成参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置之后，如果变量rawChanged的值等于true，那么就说明参数wallpaper所描述的壁纸窗口在X轴和Y轴上的偏移位置发生了变化，这时候就需要向提供该壁纸窗口的服务发送一个事件通知，这是通过调用参数wallpaperWin所指向的一个WindowState对象的成员变量mClient所描述的一个实现了IWindow接口的Binder代理对象的成员函数dispatchWallpaperOffsets来实现的，同时传递给壁纸窗口的服务的参数有壁纸窗口当前所使用的WallpaperX、WallpaperY、WallpaperXStep和WallpaperYStep值，以及另外一个同步参数sync。

        当参数sync的值等于true的时候，就表示WindowManagerService服务需要等待提供壁纸窗口wallpaperWin的服务处理完成前面所发送的偏移位置变化事件通知，等待的最长时间为WALLPAPER\_TIMEOUT。如果提供壁纸窗口wallpaperWin的服务不能在WALLPAPER\_TIMEOUT时间内向WindowManagerService服务发送一个事件处理完成通知，那么WindowManagerService服务就会将这次事件通知发送时间start保存在WindowManagerService类的成员变量mLastWallpaperTimeoutTime中。

        如果上一次发送的壁纸窗口偏移位置变化事件通知发生了超时，那么在上次发送这个事件通知起的WALLPAPER\_TIMEOUT\_RECOVERY时间内，是不允许再次发送壁纸窗口偏移位置变化事件通知的。这是因为在上一次事件通知超时的情况下，在短时间内再次发送相同的事件通知也是非常有可能是超时的，因此，就不允许短时间内重复发送相同的事件通知，避免出现雪崩现象。

        关于互联网的雪崩现象，可以举一个常见的例子来说明。假设现在有一个Web页面正在现场直播一项非常热门的体育赛事，这时候就会有海量的用户访问这个页面。一旦访问量快要达到Web服务器的承受能力的时候，Web页面的打开速度就会越来越慢。Web页面打开速度变慢的时候，用户就会下意识地不断按F5刷新。越是不断地按F5刷新，Web页面的请求量就越大，而当请求量大于Web服务器的承受能力的时候，Web服务器就会宕机了，这个就是雪崩现象。为了避免雪崩现象，就需要在请求量快要达到Web服务器的承受能力的时候，避免用户发送更多的访问请求，以使得Web服务器有喘息的机会。

        废话少说了，当WindowManagerService服务在等待壁纸窗口wallpaper所属的服务处理它的偏移位置变化事件通知时，会将该壁纸窗口wallpaper保存在WindowManagerService类的成员变量mWaitingOnWallpaper中，用来表示WindowManagerService服务正在处于等待壁纸服务处理完成一个壁纸窗口偏移位置变化事件通知。一旦壁纸服务处理完成该事件通知，WindowManagerService类的成员变量mWaitingOnWallpaper的值就会被设置为null。

        壁纸服务处理壁纸窗口在X轴和Y轴上的偏移位置变化事件通知的过程就如图4的Step 6至Step 8所示。

        至此，我们就分析完成壁纸窗口在X轴和Y轴上的偏移位置的调整过程了，接下来我们就继续分析壁纸窗口在窗口堆栈中的位置调整过程。

        三. 调整壁纸窗口在窗口堆栈中的位置

        调整壁纸窗口在窗口堆栈中的位置实际上就是将壁纸窗口放置在需要显示壁纸的窗口的下面，这是是通过调用WindowManagerService类的成员函数adjustWallpaperWindowsLocked来实现的。

        WindowManagerService类的成员函数adjustWallpaperWindowsLocked的实现框架如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **int** adjustWallpaperWindowsLocked() {
6. **int** changed = 0;
8. **final** **int** dw = mDisplay.getWidth();
9. **final** **int** dh = mDisplay.getHeight();
11. // First find top-most window that has asked to be on top of the
12. // wallpaper; all wallpapers go behind it.
13. **final** ArrayList<WindowState> localmWindows = mWindows;
14. **int** N = localmWindows.size();
15. WindowState w = **null**;
16. WindowState foundW = **null**;
17. **int** foundI = 0;
18. WindowState topCurW = **null**;
19. **int** topCurI = 0;
20. **int** i = N;
21. //Label #1:
22. **while** (i > 0) {
23. //从上到下遍历窗口堆栈，查找需要显示壁纸的窗口foundW，foundI为窗口foundW在窗口堆栈中
24. //的位置如果没有找到需要显示壁纸的窗口，并且系统中存在壁纸窗口，那么topCurW就指向Z轴
25. //位置最大的壁纸窗口，topCurI为窗口topCurW在窗口堆栈中的位置，这时候foundW一定等于
26. //null。
27. ......
28. }
30. //Label #2:
31. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
32. //如果系统当前正在窗口切换的过程中，并且系统当前存在一个需要显示壁纸的Activity窗口，
33. //那么就认为当前正在执行的窗口切换涉及到了这个需要显示壁纸的Activity窗口，
34. //因此，就暂时不要调整壁纸窗口的位置了，等到窗口切换过程完成了再说。
35. //系统当前存在一个需要显示壁纸的Activity窗口，意味着mWallpaperTarget不等于null，
36. //或者foundW不等于null。
37. ......
38. }
40. **if** (mWallpaperTarget != foundW) {
41. //上一次显示壁纸的窗口和接下来要显示壁纸的窗口发生了变化
43. mLowerWallpaperTarget = **null**;
44. mUpperWallpaperTarget = **null**;
46. WindowState oldW = mWallpaperTarget;
47. mWallpaperTarget = foundW;
49. // Now what is happening...  if the current and new targets are
50. // animating, then we are in our super special mode!
51. **if** (foundW != **null** && oldW != **null**) {
52. **boolean** oldAnim = oldW.mAnimation != **null**
53. || (oldW.mAppToken != **null** && oldW.mAppToken.animation != **null**);
54. **boolean** foundAnim = foundW.mAnimation != **null**
55. || (foundW.mAppToken != **null** && foundW.mAppToken.animation != **null**);
56. ......
58. //Label #3:
59. **if** (foundAnim && oldAnim) {
60. //上一次显示壁纸的窗口oldW和接下来要显示壁纸的窗口foundW正在显示动画的
61. //过程中，那么就将Z轴位置较高的窗口保存在mUpperWallpaperTarget中，而将
62. //Z轴位置较低的窗口保存在mLowerWallpaperTarget中，并且将变量foundW指向
63. //Z轴位置较高的窗口，这样就可以在这两个窗口的动画显示过程中都能看到壁
64. //纸窗口.
65. ......
66. }
67. }
68. ｝**else** **if** (mLowerWallpaperTarget != **null**) {
69. //Label #4:
70. //检查mUpperWallpaperTarget和mLowerWallpaperTarget所指向的窗口的动画显示过程
71. //是否已经结束，如果已经结束，那么就将mUpperWallpaperTarget和
72. //mLowerWallpaperTarget的值置null。
73. // Is it time to stop animating?
74. ......
75. }
77. **boolean** visible = foundW != **null**;
78. //Label #5:
79. **if** (visible) {
80. //前面找到了一个需要显示壁纸的窗口foundW，并且存在其它窗口与它关联，这些关联的
81. //窗口包括：
82. //1. 在与该窗口所对应的窗口令牌的其它窗口
83. //2. 该窗口所设置的启动窗口
84. //3. 附加在该窗口的其它窗口
85. //在上述这些关联的窗口中，如果存在一些Z轴位置比窗口foundW小，那么就将需要将壁纸
86. //窗口放在Z轴位置最小的那个窗口下面，即将变量foundW指向Z轴位置最小的那个窗口。
87. ......
88. }
90. //让变量foundW指向前面找到的需要显示壁纸的窗口的下一个窗口，
91. //这时候变量foundI记录的仍是需要显示壁纸的窗口在窗口堆栈中的位置，
92. //接下来会根据这两个变量来调整壁纸窗口在窗口堆栈中的位置
93. **if** (foundW == **null** && topCurW != **null**) {
94. //前面提到，如果没有找到需要显示壁纸的窗口，并且系统中存在壁纸窗口，那么foundW一
95. //定等于null，并且topCurW一定不等于null，这时候就不需要调整壁纸窗口在窗口堆栈中的
96. //位置。为了与其它情况统一处理，这时候假设位于壁纸窗口上面的那个窗口就是需要显示
97. //壁纸的窗口。因此，就会将foundI的值设置为(topCurI+1)，而将foundW的值设置为
98. //topCurW。
99. // There is no wallpaper target, so it goes at the bottom.
100. // We will assume it is the same place as last time, if known.
101. foundW = topCurW;
102. foundI = topCurI+1;
103. } **else** {
104. //前面找到了需要显示壁纸的窗口，因此，就将它的下一个窗口保存在foundW中，变量foundI
105. //的值不需要修改。
106. // Okay i is the position immediately above the wallpaper.  Look at
107. // what is below it for later.
108. foundW = foundI > 0 ? localmWindows.get(foundI-1) : **null**;
109. }
111. //如果前面找到的需要显示壁纸的窗口是可见的，并且当前正在显示壁纸的窗口设置了壁纸窗口
112. //在X轴和Y轴上的偏移位置，那么就将用来描述壁纸窗口在X轴和Y轴上的偏移位置的WallpaperX、
113. //WallpaperY、WallpaperXStep和WallpaperYStep值记录在mLastWallpaperX、
114. //mLastWallpaperXStep、mLastWallpaperY和mLastWallpaperYStep中。
115. **if** (visible) {
116. **if** (mWallpaperTarget.mWallpaperX >= 0) {
117. mLastWallpaperX = mWallpaperTarget.mWallpaperX;
118. mLastWallpaperXStep = mWallpaperTarget.mWallpaperXStep;
119. }
120. **if** (mWallpaperTarget.mWallpaperY >= 0) {
121. mLastWallpaperY = mWallpaperTarget.mWallpaperY;
122. mLastWallpaperYStep = mWallpaperTarget.mWallpaperYStep;
123. }
124. }
126. //Label #6:
127. // Start stepping backwards from here, ensuring that our wallpaper windows
128. // are correctly placed.
129. **int** curTokenIndex = mWallpaperTokens.size();
130. **while** (curTokenIndex > 0) {
131. //一切准备就绪，开始调整系统中的壁纸窗口在窗口堆栈的位置，[算法](https://www.xuebuyuan.com/category/%E7%AE%97%E6%B3%95)如下所示。
132. //对于从Z轴位置从高到低的每一个壁纸窗口wallpaper：
133. //1. 如果它与变量foundW指向的不是同一个壁纸窗口，那么就说明它在窗口堆栈中
134. //的位置不对，这时候就需要将它调整到窗口堆栈中的第foundI个位置上。
135. //2. 如果它与变量foundW指向的是同一个壁纸窗口，那么就说明它在窗口堆栈中的
136. //位置是正确，这时候就不需要对它进行调整，不过要让变量foundI的值减1，并且将
137. //在窗口堆栈第(foundI - 1)个位置的窗口记录在变量foundW中。
138. //注意，变量foundW一开始就指向Z轴位置最高的壁纸窗口，而变量foundI记录的是
139. //位于Z轴位置最高的壁纸窗口上面的那个窗口在窗口堆栈中的位置。
140. //上述算法实际上是用状态机的方法将系统中的所有壁纸窗口（假设数量为N）按照Z轴
141. //位置从高到底的顺序放置在窗口堆栈中的第（foundI - 1）、（foundI - 2）、
142. //（foundI - 3）、......、（foundI - N）个位置上。
143. ......
144. }
146. **return** changed;
147. }
149. ......
150. }

        这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

        WindowManagerService类的成员函数adjustWallpaperWindowsLocked是按照以下流程来调整壁纸窗口在窗口堆栈中的位置的：

        1. 通过一个while循环来从上到下地遍历窗口堆栈，找到需要显示壁纸的窗口foundW，其中，foundI为窗口foundW在窗口堆栈中的位置。如果没有找到需要显示壁纸的窗口，并且系统中存在壁纸窗口，那么topCurW就指向Z轴位置最大的壁纸窗口，其中，topCurI为窗口topCurW在窗口堆栈中的位置。在这种情况下，变量foundW的值一定等于null的。

        2. 如果WindowManagerService类的成员变量mNextAppTransition的值不等于WindowManagerPolicy.TRANSIT\_UNSET，那么就说明系统当前正在窗口切换的过程中。在这种情况下，如果系统当前存在一个需要显示壁纸的Activity窗口，即WindowManagerService类的成员变量mWallpaperTarget的值不等于null，或者前面得到的变量foundW的值不等于null，那么就认为当前正在执行的窗口切换操作涉及到了这个需要显示壁纸的Activity窗口。这时候就不需要调整壁纸窗口的位置，要等到窗口切换过程完成了之后再调整。

        3. 如果WindowManagerService类的成员变量mWallpaperTarget和前面得到的变量foundW指向的不是同一个WindowState对象，那么就说明上一次显示壁纸的窗口和接下来要显示壁纸的窗口发生了变化。在这种情况下，就会使用变量oldW来描述上一次显示壁纸的窗口，而接下来要显示壁纸的窗口通过WindowManagerService类的成员变量mWallpaperTarget以及变量foundW来描述。这时候如果检查发现上一次显示壁纸的窗口和接下来要显示壁纸的窗口都处于显示动画的过程中，那么就会将Z轴位置较高的窗口保存在WindowManagerService类的成员变量mUpperWallpaperTarget中，而将Z轴位置较低的窗口保存在WindowManagerService类的成员变量mLowerWallpaperTarget中，并且将变量foundW指向Z轴位置较高的窗口。这样就能保证在这两个窗口的动画显示过程中都能看到壁纸窗口，实际上就是保证在两个窗口的切换过程中看到壁纸窗口。

        4. 如果WindowManagerService类的成员变量mWallpaperTarget和前面得到的变量foundW指向的是同一个WindowState对象，并且WindowManagerService类的成员变量mLowerWallpaperTarget的值不等于null，那么就说明需要检查系统的窗口切换过程完成了没有。如果已经完成，那么就需要将WindowManagerService类的成员变量mUpperWallpaperTarget和mLowerWallpaperTarget的值设置为null。由此可以，WindowManagerService类的成员变量mUpperWallpaperTarget和mLowerWallpaperTarget的作用就是用来记录两个处于切换状态的需要显示壁纸的窗口。

        5. 如果变量foundW的值不等于null，那么就说明前面找到了一个接下来要显示壁纸的窗口。在这种情况下，需要做两件事情。第一件事情是判断接下来要显示壁纸的窗口是否是可见的。如果是的话，那么就会将变量visible的值设置为true。第二件事情是在与接下来要显示壁纸的窗口相关联的窗口中，即与变量foundW所描述的窗口相关联的窗口中，找到一个Z轴位置最小的窗口，因为壁纸窗口最终是要放置在这个Z轴位置最小的窗口的下面，而不是最初找到的那个窗口的下面。与变量foundW所描述的窗口相关联的窗口包括：A.  
 与变量foundW所描述的窗口具有相同窗口令牌的其它窗口；B. 与变量foundW所描述的窗口附加在同一个窗口的其它窗口；C. 为变量foundW所描述的窗口所设置的启动窗口；D. 附加在变量foundW所描述的窗口上的其它窗口。一旦找到这样的一个窗口，那么就会让重新让变量foundW指向它。

        6. 再次重新调整变量foundW的值，让它指向位于前面所找到的需要显示壁纸的窗口的下面的一个窗口。注意，这个窗口有可能就是壁纸窗口。这时候变量foundI记录的然是前面所找到的需要显示壁纸的窗口在窗口堆栈中的位置。这样做的目的是为了接下来可以方便地调整壁纸窗口在窗口堆栈中的位置。但是如果变量foundW的值等于null，那么就说明前面根本没有找到需要显示壁纸的窗口。在这种情况下，如果变量topCurW的值不等于null，那么就说明系统中存在壁纸窗口。这种情况其实就不需要调整壁纸窗口在窗口堆栈中的位置了，但是为了接下来的逻辑可以统一处理，就假定位于壁纸窗口上面的那个窗口是需要显示壁纸的窗口。因此，就会将变量foundI的值设置为(topCurI+1)，而将变量foundW的值设置为topCurW。

        7. 如果前面所找到的需要显示壁纸的窗口是可见的，即变量visible的值等于true，并且当前正在显示壁纸的窗口设置了壁纸窗口在X轴和Y轴上的有效偏移位置，即WindowManagerService类的成员变量mWallpaperTarget所指向的一个WindowState对象的成员变量mWallpaperX和mWallpaperY的值大于等于0，那么就将用来描述壁纸窗口在X轴和Y轴上的偏移位置的WallpaperX、WallpaperY、WallpaperXStep和WallpaperYStep值记录在WindowManagerService类的成员变量mLastWallpaperX、mLastWallpaperY、mLastWallpaperXStep和mLastWallpaperYStep中。

        8. 经过上面的一系列操作之后，现在一切准备就绪，因此就可以按照以下的算法来调整系统中的壁纸窗口在窗口堆栈的位置。对于从Z轴位置从高到低的每一个壁纸窗口wallpaper：(1). 如果它与变量foundW指向的不是同一个壁纸窗口，那么就说明它在窗口堆栈中的位置不对，这时候就需要将它调整到窗口堆栈中的第foundI个位置上；(2). 如果它与变量foundW指向的是同一个壁纸窗口，那么就说明它在窗口堆栈中的位置是正确，这时候就不需要对它进行调整，不过要让变量foundI的值减1，并且将在窗口堆栈第(foundI  
- 1)个位置的窗口记录在变量foundW中；(3). 重复执行第(1)和第(2)步的操作，直到系统所有的壁纸窗口都检查完成为止。注意，在上述算法中，变量foundW一开始就指向Z轴位置最高的壁纸窗口，而变量foundI记录的是位于Z轴位置最高的壁纸窗口上面的那个窗口在窗口堆栈中的位置。每当Z轴位置最高的壁纸窗口在窗口堆栈中的位置调整完成之后，变量foundW就会指向Z轴位置次高的壁纸窗口，而变量foundI的值也会相应的地减少1。这个算法其实就是用状态机的方法来将系统中的所有壁纸窗口（假设数量为N）按照Z轴位置从高到底的顺序放置在窗口堆栈中的第（foundI  
- 1）、（foundI - 2）、（foundI - 3）、......、（foundI - N）个位置上。

        上述流程可能还是比较抽象，接下来我们就通过在标号为Label #1、Label #2、Label #3、Label #4、Label #5和Label #6处所忽略的代码来详细分析壁纸窗口在窗口堆栈中的位置的调整过程。

        标号为Label #1的代码如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **while** (i > 0) {
2. i--;
3. w = localmWindows.get(i);
4. **if** ((w.mAttrs.type == WindowManager.LayoutParams.TYPE\_WALLPAPER)) {
5. **if** (topCurW == **null**) {
6. topCurW = w;
7. topCurI = i;
8. }
9. **continue**;
10. }
11. topCurW = **null**;
12. **if** (w.mAppToken != **null**) {
13. // If this window's app token is hidden and not animating,
14. // it is of no interest to us.
15. **if** (w.mAppToken.hidden && w.mAppToken.animation == **null**) {
16. ......
17. topCurW = **null**;
18. **continue**;
19. }
20. }
22. ......
24. **if** ((w.mAttrs.flags&FLAG\_SHOW\_WALLPAPER) != 0 && w.isReadyForDisplay()
25. && (mWallpaperTarget == w
26. || (!w.mDrawPending && !w.mCommitDrawPending))) {
27. ......
29. foundW = w;
30. foundI = i;
31. **if** (w == mWallpaperTarget && ((w.mAppToken != **null**
32. && w.mAppToken.animation != **null**)
33. || w.mAnimation != **null**)) {
34. // The current wallpaper target is animating, so we'll
35. // look behind it for another possible target and figure
36. // out what is going on below.
37. ......
38. **continue**;
39. }
40. **break**;
41. }
42. }

        这段代码从上到下遍历保存在窗口堆栈中的窗口，目的是要找到一个Z轴位置最大的并且需要显示壁纸的窗口。一个窗口如果需要显示壁纸，那么用来描述它的一个WindowState对象w的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的值的FLAG\_SHOW\_WALLPAPER位就不等于0。

        一个需要显示壁纸的窗口只有准备就绪显示并且UI也已经绘制完成之后，WindowManagerService服务才会将壁纸窗口放置在它的下面。 一个需要显示壁纸的窗口如果已经准备就绪显示，那么用来描述它的一个WindowState对象w的成员函数isReadyForDisplay的返回值等于true。另一方面，如果一个窗口的UI还没有绘制，那么用来描述它的一个WindowState对象w的成员变量mDrawPending的值就会等于true。一个窗口的UI虽然绘制好了，但是还没有提交给SurfaceFlinger服务处理，即用来描述它的一个WindowState对象w的成员变量mCommitDrawPending的值等于true，那么它的UI也是认为还没有绘制完成的。

        在遍历的过程中，如果发现一个窗口w刚好就是当前正在显示壁纸的窗口mWallpaperTarget，那么就会继续检查该窗口是否正处于显示动画的过程中。如果是的话，那么就需要跳过该窗口，因为我们的目标是要找到另外一个接下来要显示壁纸的窗口。对于Activity窗口和非Activity窗口来说，判断它们是否是正处于显示动画的过程中的方法是不一样的。对于一个处于显示动画过程的Activity窗口来说，用来描述它的一个WindowState对象w的成员变量mAppToken的值不等于null，并且指向了一个AppWindowToken对象，并且这个AppWindowToken对象的成员变量animation的值不等于null。对于一个处于显示动画过程的非Activity窗口来说，用来描述它的一个WindowState对象w的成员变量mAnimation的值不等于null。这就是说，AppWindowToken类的成员变量animation和WindowState类的成员变量mAnimation都是用来描述一个动画对象的。

       在遍历的过程中，有两种类型的窗口是需要跳过的。第一种类型的窗口是壁纸窗口，即用来描述它的一个WindowState对象w的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量type的值等于WindowManager.LayoutParams.TYPE\_WALLPAPER。第二种类型的窗口是Activity窗口，但是与之所对应的Activity组件处于不可见状态，这意味着这种类型的窗口也是不可见的。前面提到，对于Activity窗口来说，用来描述它的一个WindowState对象w的成员变量mAppToken的值是不等于null的，并且指向了一个AppWindowToken对象。当这个AppWindowToken对象的成员变量hidden的值等于true的时候，就意味着对应的Activity组件是不可见的。有时候一个AppWindowToken对象的成员变量hidden的值虽然等于true，但是如果这个AppWindowToken对象的成员变量animation的值不等于null，那么隐含着对应的Activity组件其实还是可见的，因为它还处于显示动画的过程中。

        遍历完成之后，有可能找到了接下来要显示壁纸的窗口，也有可能找不到接下来要显示壁纸的窗口。

        如果找到了接下来要显示壁纸的窗口，那么变量foundW的值就不等于null，并且指向了这个接下来要显示壁纸的窗口，另外一个变量foundI记录的是该窗口在窗口堆栈中位置。这时候变量topCurW的值一定等于null，但是变量topCurI的值却不一定等于0，它有可能指向了Z轴位置最大的那个壁纸窗口。

        假设foundW的值不等于null，并且变量topCurI的值等于0.，那么窗口堆栈的状态就如图5所示：

图5 foundw != null & topCurI == 0

        假设foundW的值不等于null，并且变量topCurI的值大于0.，那么窗口堆栈的状态就如图6所示：

图6 foundW != null & topCurI != 0

        如果没有找到接下来要显示壁纸的窗口，那么变量foundW的值就等于null，并且另外一个变量foundI的值等于0。这时候变量topCurW的值始终等于null，而变量topCurI的值可能不等于0，取决于系统中是否存在壁纸窗口。

        为了方便描述，我们假设系统中是存在壁纸窗口，那么这时候topCurI的值就不等于0，并且它记录的是Z轴位置最大的那个壁纸窗口在窗口堆栈中的位置，如图7所示：

图7 foundW == null && topCurI != 0

        标号为Label #2的代码如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **if** (mNextAppTransition != WindowManagerPolicy.TRANSIT\_UNSET) {
2. // If we are currently waiting for an app transition, and either
3. // the current target or the next target are involved with it,
4. // then hold off on doing anything with the wallpaper.
5. // Note that we are checking here for just whether the target
6. // is part of an app token...  which is potentially overly aggressive
7. // (the app token may not be involved in the transition), but good
8. // enough (we'll just wait until whatever transition is pending
9. // executes).
10. **if** (mWallpaperTarget != **null** && mWallpaperTarget.mAppToken != **null**) {
11. ......
12. **return** 0;
13. }
14. **if** (foundW != **null** && foundW.mAppToken != **null**) {
15. ......
16. **return** 0;
17. }
18. }

        WindowManagerService类的成员变量mNextAppTransition的值不等于WindowManagerPolicy.TRANSIT\_UNSET意味着系统当前正在窗口切换的过程中。这里说的窗口切换其实就是由Activity组件切换引起来的，即切换的是Activity窗口。如果正在切换的Activity窗口是是需要显示壁纸的，那么WindowManagerService类的成员函数adjustWallpaperWindowsLocked就要等到切换过程结束后，才能调整重新调整壁纸窗口在窗口堆栈中的位置。

        这里本来是要判断正在发生切换的Activity窗口是否是当前壁纸窗口的目标窗口或者前面所找到的接下来要显示壁纸的窗口的，但是却没有这样做。这段代码采取了一种比较激进的方法，即主要发现当前壁纸窗口的目标窗口是一个Activity窗口，或者前面所找到的接下来要显示壁纸的窗口是一个Activity窗口，那么就认为当前正在执行的窗口切换过程涉及到了壁纸窗口，因此，就要等到切换过程结束后，再来重新调整壁纸窗口在窗口堆栈中的位置。

        WindowManagerService类的成员变量mWallpaperTarget描述的就是当前壁纸窗口的目标窗口，当它的值不等于null时，并且它所指向的一个WindowState对象的成员变量mAppToken的值不等于null，那么就说明当前壁纸窗口的目标窗口是一个Activity窗口。同样，如果前面得到的变量foundW的值不等于null，并且它所指向的一个WindowState对象的成员变量mAppToken的值不等于null，那么就说明前面所找到的接下来要显示壁纸的窗口是一个Activity窗口。

        标号为Label #3的代码如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **if** (foundAnim && oldAnim) {
2. **int** oldI = localmWindows.indexOf(oldW);
3. ......
5. **if** (oldI >= 0) {
6. ......
8. // Set the new target correctly.
9. **if** (foundW.mAppToken != **null** && foundW.mAppToken.hiddenRequested) {
10. ......
11. mWallpaperTarget = oldW;
12. }
14. // Now set the upper and lower wallpaper targets
15. // correctly, and make sure that we are positioning
16. // the wallpaper below the lower.
17. **if** (foundI > oldI) {
18. // The new target is on top of the old one.
19. ......
20. mUpperWallpaperTarget = foundW;
21. mLowerWallpaperTarget = oldW;
22. foundW = oldW;
23. foundI = oldI;
24. } **else** {
25. // The new target is below the old one.
26. ......
27. mUpperWallpaperTarget = oldW;
28. mLowerWallpaperTarget = foundW;
29. }
30. }
31. }

        当变量foundAnim和oldAnim的值均等于true的时候，就说明当前正在显示壁纸的窗口oldW和接下来要显示壁纸的窗口foundW均处于显示动画的过程中，那么就分别将它们记录在WindowManagerService类的成员变量mLowerWallpaperTarget和mUpperWallpaperTarget中，其中，前者用来描述Z轴位置较低的窗口，而后者用来描述Z轴位置较高的的窗口。

         变量foundI和oldI记录的分别是窗口foundW和oldW在窗口堆栈中的位置。因此，当变量foundI的值大于变量oldI的值的时候，窗口foundW就是Z轴位置较高的的窗口，而窗口oldW就是Z轴位置较低的的窗口。相反，当变量foundI的值小于等于变量oldI的值的时候，窗口oldW就是Z轴位置较高的的窗口，而窗口foundW就是Z轴位置较低的的窗口。

         这里有三个地方是需要注意的：

         1. 当前正在显示壁纸的窗口oldW其实就是WindowManagerService类的成员变量mWallpaperTarget所描述的那个窗口。

         2. 变量foundW和foundI记录的始终都是Z轴位置较低的那个窗口及其在窗口堆栈的位置，因此，当变量foundI的值大于变量oldI的值的时候，要将变量foundW和foundI的值分别设置为oldW和oldI，这样做的目的是为了接下来可以将壁纸窗口放置在Z轴位置较低的窗口的下面，以便可以在两个窗口的动画显示过程中看到壁纸。

         3. 如果前面找到的接下来要显示壁纸的窗口是一个Activity窗口，即变量foundW所描述的一个WindowState对象的成员变量mAppToken的值不等于null，并且它所指向的一个AppWindowToken对象的成员变量hiddenRequested的值等于true，那么就说明与窗口foundW所对应的一个Activity组件已经被请求隐藏起来了。在这种情况下，当前正在显示壁纸的窗口就会仍然被当作是接下来壁纸窗口的目标窗口。由于此前我们已经将WindowManagerService类的成员变量mWallpaperTarget的值设置了为foundW，因此，这时候就需要将它的值修改为oldW。

         这段代码执行完成之后，窗口堆栈的状态就如图8所示：

图8 mUpperWallpaperTarget、mLowerWallpaperTarget、foundW和foundI的关系

        标号为Label #4的代码如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. // Is it time to stop animating?
2. **boolean** lowerAnimating = mLowerWallpaperTarget.mAnimation != **null**
3. || (mLowerWallpaperTarget.mAppToken != **null**
4. && mLowerWallpaperTarget.mAppToken.animation != **null**);
5. **boolean** upperAnimating = mUpperWallpaperTarget.mAnimation != **null**
6. || (mUpperWallpaperTarget.mAppToken != **null**
7. && mUpperWallpaperTarget.mAppToken.animation != **null**);
8. **if** (!lowerAnimating || !upperAnimating) {
9. ......
10. mLowerWallpaperTarget = **null**;
11. mUpperWallpaperTarget = **null**;
12. }

        这段代码检查WindowManagerService类的成员变量mLowerWallpaperTarget和mUpperWallpaperTarget所描述的两个窗口的动画是否已经显示结束。如果已经显示结束，那么就会将这两个成员变量的值设置为null。

        注意，如果一个窗口的动画已经显示结束，那么用来描述它的一个WindowState对象的成员变量mAnimation的值就会等于null。另外，如果一个Activity窗口的动画已经显示结束，那么用来描述它的WindowState对象的成员变量mAppWindowToken所指向的一个AppWindowToken对象的成员变量animation的值也会等于null。

        标号为Label #5的代码如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. **boolean** visible = foundW != **null**;
2. **if** (visible) {
3. // The window is visible to the compositor...  but is it visible
4. // to the user?  That is what the wallpaper cares about.
5. visible = isWallpaperVisible(foundW);
6. ......
8. // If the wallpaper target is animating, we may need to copy
9. // its layer adjustment.  Only do this if we are not transfering
10. // between two wallpaper targets.
11. mWallpaperAnimLayerAdjustment =
12. (mLowerWallpaperTarget == **null** && foundW.mAppToken != **null**)
13. ? foundW.mAppToken.animLayerAdjustment : 0;
15. **final** **int** maxLayer = mPolicy.getMaxWallpaperLayer()
16. \* TYPE\_LAYER\_MULTIPLIER
17. + TYPE\_LAYER\_OFFSET;
19. // Now w is the window we are supposed to be behind...  but we
20. // need to be sure to also be behind any of its attached windows,
21. // AND any starting window associated with it, AND below the
22. // maximum layer the policy allows for wallpapers.
23. **while** (foundI > 0) {
24. WindowState wb = localmWindows.get(foundI-1);
25. **if** (wb.mBaseLayer < maxLayer &&
26. wb.mAttachedWindow != foundW &&
27. wb.mAttachedWindow != foundW.mAttachedWindow &&
28. (wb.mAttrs.type != TYPE\_APPLICATION\_STARTING ||
29. wb.mToken != foundW.mToken)) {
30. // This window is not related to the previous one in any
31. // interesting way, so stop here.
32. **break**;
33. }
34. foundW = wb;
35. foundI--;
36. }
37. }

        当变量foundW的值不等于null时，就说明前面找到了一个接下来要显示壁纸的窗口。在这种情况下，需要做三件事件：

        1. 判断窗口foundW是否是可见的，这是通过调用WindowManagerService类的成员函数isWallpaperVisible来实现的。如果可见，那么变量visible的值就会等于true，否则就会等于false。后面在调整壁纸窗口在窗口堆栈中的位置时，会根据变量visible的值来决定要显示壁纸窗口还是隐藏壁纸窗口。

        2. 检查窗口foundW是否是一个Activity窗口。如果是的话，那么就会将用来描述它的一个WindowState对象的成员变量mAppToken所指向的一个AppWindowToken对象的成员变量animLayerAdjustment的值保存在WindowManagerService类的成员变量mWallpaperAnimLayerAdjustment中。在计算壁纸窗品的Z轴位置的时候，需要使用到WindowManagerService类的成员变量mWallpaperAnimLayerAdjustment，用来调整壁纸窗品的Z轴位置。在后面一篇文章分析窗口的Z轴位置的计算方法时，我们再详细分析壁纸窗口的Z轴位置是如何计算的。注意，如果这时候系统的壁纸窗口有两个目标窗口，即WindowManagerService类的成员变量mLowerWallpaperTarget的值不等于null，那么就说明壁纸窗口的目标窗口正在显示动画的过程中。在这种情况下，就不需要调整壁纸窗品的Z轴位置，即会将WindowManagerService类的成员变量mLowerWallpaperTarget的值设置为0。等到壁纸窗口的目标窗口结束动画显示过程之后，再来调整它的Z轴位置。

        3. 检查窗口foundW的下面是否存在一些关联的窗口。如果存在的话，就需要将壁纸窗口放置在这些关联的窗口中Z轴位置最低的窗口的下面。这段代码通过一个while循环从窗口foundW的下面一个窗口开始往下检查，直到找到一个没有关联的窗口为止。在检查的过程中，每碰到一个关联的窗口，那么就让变量foundW指向它，并且将变量foundI的值减少1。这样最终得到的变量foundW和foundI就是用来描述与窗口foundW有联的、Z轴位置最低的窗口及其在窗口堆栈中的位置。

        前面提到，窗口foundW所关联的窗口四种，即对于一个窗口wb来，如果它满足以下四个条件，那么它就与窗口foundW有关联：

        A.  窗口wb与窗口foundW对应的是同一个窗品令牌，即分别用来描述窗口wb和窗口foundW的两个WindowState对象的成员变量mToken指向的是同一个WindowToken对象。

        B.  窗口wb附加在窗口foundW上，即用来描述窗口wb的一个WindowState对象的成员变量mAttachedWindow与变量foundW指向的是同一个WindowState对象。

        C.  窗口wb与窗口foundW附加在同一个窗口上，即分别用来描述窗口wb和窗口foundW的两个WindowState对象的成员变量mAttachedWindow指向的是同一个WindowState对象。

        D. 窗口wb是窗口foundW的启动窗口，即用来描述窗口wb的一个WindowState对象的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_APPLICATION\_STARTING。

        此外，WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象会规定系统中的壁纸窗口的Z轴位置不能大于某一个值，也就是说，壁纸窗口的Z轴位置有一个最大值限制。这个限制值可以通过调用WindowManagerService类的成员变量mPolicy所指向的一个PhoneWindowManager对象的成员函数getMaxWallpaperLayer来获得。获得了这个限制值之后，还需要乘以一个窗口类型因子TYPE\_LAYER\_MULTIPLIER，最后再加一个窗口类型偏移值TYPE\_LAYER\_OFFSET，就可以得到壁纸窗口的最大Z轴位置限制值maxLayer。这时候如果在窗口foundW的下面找到一个窗口wb，它的Z轴位置大于等于maxLayer，即用来描述它的一个WindowState对象的成员变量mBaseLayer的值大于maxLayer，那么也会认为窗口wb是与窗口foundW有关联的。

       我们通过图9和图10来说明查找与窗口foundW关联的、Z轴位置最小的窗口的过程：

图9 查找与窗口foundW关联的窗口之前

图10 查找与窗口foundW关联的窗口之后

        标号为Label #6的代码如下所示：

**[java]** [view  
plaincopy](http://blog.csdn.net/luoshengyang/article/details/8550820)

1. // Start stepping backwards from here, ensuring that our wallpaper windows
2. // are correctly placed.
3. **int** curTokenIndex = mWallpaperTokens.size();
4. **while** (curTokenIndex > 0) {
5. curTokenIndex--;
6. WindowToken token = mWallpaperTokens.get(curTokenIndex);
7. **if** (token.hidden == visible) {
8. changed |= ADJUST\_WALLPAPER\_VISIBILITY\_CHANGED;
9. token.hidden = !visible;
10. // Need to do a layout to ensure the wallpaper now has the
11. // correct size.
12. mLayoutNeeded = **true**;
13. }
15. **int** curWallpaperIndex = token.windows.size();
16. **while** (curWallpaperIndex > 0) {
17. curWallpaperIndex--;
18. WindowState wallpaper = token.windows.get(curWallpaperIndex);
20. **if** (visible) {
21. updateWallpaperOffsetLocked(wallpaper, dw, dh, **false**);
22. }
24. // First, make sure the client has the current visibility
25. // state.
26. **if** (wallpaper.mWallpaperVisible != visible) {
27. wallpaper.mWallpaperVisible = visible;
28. **try** {
29. ......
30. wallpaper.mClient.dispatchAppVisibility(visible);
31. } **catch** (RemoteException e) {
32. }
33. }
35. wallpaper.mAnimLayer = wallpaper.mLayer + mWallpaperAnimLayerAdjustment;
36. ......
38. // First, if this window is at the current index, then all
39. // is well.
40. **if** (wallpaper == foundW) {
41. foundI--;
42. foundW = foundI > 0
43. ? localmWindows.get(foundI-1) : **null**;
44. **continue**;
45. }
47. // The window didn't match...  the current wallpaper window,
48. // wherever it is, is in the wrong place, so make sure it is
49. // not in the list.
50. **int** oldIndex = localmWindows.indexOf(wallpaper);
51. **if** (oldIndex >= 0) {
52. i......
53. localmWindows.remove(oldIndex);
54. mWindowsChanged = **true**;
55. **if** (oldIndex < foundI) {
56. foundI--;
57. }
58. }
60. // Now stick it in.
61. ......
63. localmWindows.add(foundI, wallpaper);
64. mWindowsChanged = **true**;
65. changed |= ADJUST\_WALLPAPER\_LAYERS\_CHANGED;
66. }
67. }

        这段代码就是用来调整系统中的壁纸窗口在窗口堆栈中的位置的，目标就是要将它们放置在前面所找到的接下来要显示壁纸的窗口的下面。

        WindowManagerService类的成员变量mWallpaperTokens保存的是一系列WindowToken对象，它们描述的是系统中的壁纸窗口令牌。这些WindowToken对象都有一个成员变量windows，里面保存的是一系列WindowState对象，它们描述的是系统中的壁纸窗口。这段代码就目标就要通过两个嵌套的while循环来将这些WindowState对象调整到前面所找到的接下来要显示壁纸的窗口的下面去。

        在调整壁纸窗口在窗口堆栈中的位置的过程中，还会做以下四件事情：

        1. 设置壁纸窗口令牌的可见性。也就是说，如果一个用来描述壁纸窗口令牌的WindowToken对象token的成员变量hidden的值不等于前面得到的变量visible的值，那么就说明该壁纸窗口令牌的可见性发生了变化。由于WindowToken类的成员变量hidden是用来表示壁纸窗口令牌的不可见状态的，而变量visible是用来表示接下来要显示壁纸的窗口是可见的，因此，当一个壁纸窗口令牌的可见性发生变化时，就要将用来描述它的WindowToken对象token的成员变量hidden的值设置为!visbile。壁纸窗口令牌的可见性发生了变化之后，需要重新刷新系统的UI，因此，就需要将WindowManagerService类的成员变量mLayoutNeeded  
的值设置为true，并且将函数返回值changed的ADJUST\_WALLPAPER\_VISIBILITY\_CHANGED位设置为1。

        2. 在前面所找到的接下来要显示壁纸的窗口是可见的情况下，即在变量visible的值等于true的情况下，重新计算每一个壁纸窗口wallpaper在X轴和Y轴上的偏移位置，这是通过调用WindowManagerService类的成员函数updateWallpaperOffsetLocked来实现的。

        3. 如果一个壁纸窗口之前是不可见的，现在变得可见了，或者之前是可见的，现在变得不可见了，具体就表现在用来描述该壁纸窗口的一个WindowState对象的成员变量mWallpaperVisible的值不等于变量visible的值，那么就需要该WindowState对象的成员变量mWallpaperVisible的值设置为visible，并且向提供该壁纸窗口的服务发送一个可见性变化事件通知。

        4. 调整每一个壁纸窗口的Z轴位置。一个壁纸窗口的Z轴位置保存在用来描述它的一个WindowState对象的成员变量mLayer中，用这个成员变量的值加上前面已经计算好的壁纸窗口的Z轴位置调整值，即保存在WindowManagerService类的成员变量mWallpaperAnimLayerAdjustment中的值，就可以得到一个壁纸窗口的最终Z轴位置值，并且保存WindowState对象的成员变量mAnimLayer中。

        前面在分析WindowManagerService类的成员函数adjustWallpaperWindowsLocked的实现框架时提到，在调整系统中的壁纸窗口在窗口堆栈中的位置之前，变量foundW描述的应该是Z轴位置最大的壁纸窗口，而变量foundI记录的是需要显示壁纸的窗口在窗口堆栈中的位置，如图11所示：

图11 调整壁纸窗口前的窗口堆栈状态

       在图11中，接下来需要显示壁纸的是窗口A，在它下面依次是窗口B、C和D，并且系统中存在着三个壁纸窗口，它们的编号分别为1、2和3。假设窗口B和编号为3的壁纸窗口是同一个窗口，那么就说明编号为3的壁纸窗口已经在窗口堆栈中的正确位置了，因此，就不需要调整它在窗口堆栈中的位置了。这时候窗口堆栈中的状态如图12所示：

图12 处理完成编号为3的壁纸窗口后的窗口堆栈状态

       在图12中，假设窗口C和编号为2的壁纸窗口不是同一个窗口，那么就需要将编号为2的壁纸窗口放置在窗口C的位置上，如图13所示：

图13 处理完成编号为2的壁纸窗口后的窗口堆栈状态

        在图13中，假设窗口C和编号为1的壁纸窗口也不是同一个窗口，那么就需要将编号为1的壁纸窗口放置在窗口C的位置上，如图14所示：

图14 处理完成编号为1的壁纸窗口的窗口堆栈状态

       处理完成编号为1的壁纸窗口之后，系统中所有的壁纸窗口都调整到窗口A的下面去了，这样在下一次在刷新系统UI时，就可以将系统中的壁纸窗口作为窗口A的背景了。

       至此，我们就分析完成壁纸窗口在窗口堆栈中的位置调整过程了，WindowManagerService服务对壁纸窗口的管理也分析完成了。结合前面[Android窗口管理服务WindowManagerService对窗口的组织方式分析](http://blog.csdn.net/luoshengyang/article/details/8498908)和[Android窗口管理服务WindowManagerService对输入法窗口（Input  
Method Window）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8526644)这两篇文章，我们就可以对WindowManagerService服务在内部所维护的窗口堆栈有一个清晰的认识了。

       当系统中的所有窗口都在窗口堆栈排列好之后，WindowManagerService服务就可以计算每一个窗口的Z轴坐标了，以便可以传递给SurfaceFlinger服务做可见性计算，从而正确地将系统的UI渲染出来

# app用法

fgs

原文：[https://www.zybuluo.com/Tyhj/note/765583](https://link.jianshu.com?t=https%3A%2F%2Fwww.zybuluo.com%2FTyhj%2Fnote%2F765583)

最近搞了一下Android的动态壁纸，像实现“萤火视频桌面”那样，本来以为很难的，但是了解了一下感觉还是很容易的。首先想要做好一个东西必须先了解其原理，不能代码随便抄一下，改一下就行了，这也是最近去阿里和贝贝网几个大公司面试的感悟。

效果图：[http://ac-fgtnb2h8.clouddn.com/82f4e474384b28b5739a.gif](https://link.jianshu.com?t=http%3A%2F%2Fac-fgtnb2h8.clouddn.com%2F82f4e474384b28b5739a.gif)

### Android壁纸的实现和管理分为三层：

只想了解动态壁纸的看第一个就好了

#### WallpaperService与Engine

壁纸运行在一个Android服务之中，这个服务的名字叫做WallpaperService。当用户选择了一个壁纸之后，此壁纸所对应的WallpaperService便会启动并开始进行壁纸的绘制工作。Engine是WallpaperService中的一个内部类，实现了壁纸窗口的创建以及Surface的维护工作。这一层次的内容主要体现了壁纸的实现原理。

#### WallpaperManagerService

这个系统服务用于管理壁纸的运行与切换，并通过WallpaperManager类向外界提供操作壁纸的接口。这一层次主要体现了Android对壁纸的管理方式。

#### WindowManagerService

用于计算壁纸窗口的Z序、可见性以及为壁纸应用窗口动画。这一层次主要体现了Android对壁纸窗口的管理方式。

### 实现

首先静态壁纸是很简单的，大概就是如下几种方法，我也没有试过，

* 使用WallpaperManager的setResource(int ResourceID)方法
* 使用WallpaperManager的setBitmap(Bitmap bitmap)方法
* 使用WallpaperManager的setStream(InputStream data)方法

//需要权限

<uses-permission android:name = "android.permission.SET\_WALLPAPER"/>

WallpaperManager wallpaperManager = WallpaperManager.getInstance(this);

try {

wallpaperManager.setResource(R.drawable.picture);

} catch (IOException e) {

e.printStackTrace();

}

### 动态壁纸

刚才讲了，动态壁纸就是一个服务，我们先创建一个服务并继承**WallpaperService**。这个服务里面有个内部类**Engine**，实现了壁纸窗口的创建以及Surface的维护工作。就是说我们可以获取到一个**SurfaceHolder**，拿到这个东西就好办了，我们可以在上面画自己想要的东西或者把视频输出到上面去。

我就直接上代码了

#### 设置视频桌面

public class VideoWallpaper extends WallpaperService {

private MediaPlayer mp;

private int progress = 0;

//这里就是返回我们自定义的Engine

@Override

public Engine onCreateEngine() {

return new VideoEngine();

}

//自定义Engine

class VideoEngine extends Engine {

@Override

public void onCreate(SurfaceHolder surfaceHolder) {

super.onCreate(surfaceHolder);

//可以设置点击事件

setTouchEventsEnabled(true);

}

@Override

public void onSurfaceCreated(SurfaceHolder holder) {

//把视频输出到SurfaceHolder上面

if (mp != null && mp.isPlaying())

return;

//可以设置SD卡的视频

mp = MediaPlayer.create(getApplicationContext(), R.raw.bird);

//这句话并不简单

mp.setSurface(holder.getSurface());

//重复播放

mp.setLooping(true);

mp.start();

}

//当桌面不可见的时候的处理

@Override

public void onVisibilityChanged(boolean visible) {

if (visible) {

if (mp != null)

return;

mp = MediaPlayer.create(getApplicationContext(), R.raw.bird);

mp.setSurface(getSurfaceHolder().getSurface());

mp.setLooping(true);

//获取进度播放

mp.seekTo(progress);

mp.start();

} else {

if (mp != null && mp.isPlaying()) {

//保存进度

progress = mp.getCurrentPosition();

mp.stop();

mp.release();

mp = null;

}

}

}

@Override

public void onDestroy() {

if (mp != null) {

mp.stop();

mp.release();

}

super.onDestroy();

}

}

}

上面的代码并不复杂，只是自己做的时候会遇到一些问题，首在不能在Engine的onCreate的方法里面设置视频播放，应该是SurfaceHolder还没有创建吧，还有mediaPlayer设置输出的Surface

//正确设置代码

mediaPlayer.setSurface(holder.getSurface());

//一般是这样设置，这里这样设置报错

mediaPlayer.setDisplay(holder);

是Service那肯定要注册的

<service

android:name="ui.service.VideoWallpaper"

android:permission="android.permission.BIND\_WALLPAPER">

<!-- 为动态壁纸配置intent-filter -->

<intent-filter>

<action android:name="android.service.wallpaper.WallpaperService" />

</intent-filter>

<!-- 为动态壁纸配置meta-data -->

<meta-data

android:name="android.service.wallpaper"

android:resource="@xml/livewallpapervideo" />

</service>

然后关于"livewallpapervideo.xml"，就是一个正常的布局文件

<?xml version="1.0" encoding="utf-8"?>

<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"

android:settingsActivity="ui.activity.LiveWallPreference"

android:thumbnail="@mipmap/ic\_video"

android:description="@string/wallpaper\_description4"

/>

里面有个settingsActivity，这个东西我不知道有什么用，就是随便一个Activity，你也要注册这个Activity，

#### 其他玩法

除了将视频作为壁纸以为还可以将GIF作为壁纸，将摄像头获取到的图象作为壁纸，甚至直接自己在SurfaceHolder自己画东西都可以，道理都是一样的

作者：Tyhj  
链接：https://www.jianshu.com/p/5a5f95f6cd59

# FallbackHome

## DirectBoot功能介绍

当手机已经通电开机但是用户并有解锁锁屏的时候，Android N运行于一个安全的模式，也就是Dierect Boot模式。

<https://blog.csdn.net/fu_kevin0606/article/details/65437594>

<!-- Triggered when user-selected home app isn't encryption aware -->

adb shell am start -n com.android.settings/com.android.settings.FallbackHome

Android N平台新加入阻塞开机分析

<http://www.it619.net/index.php?doc-view-2610.htm>

Android7.0 DirectBoot阻塞开机分析

https://blog.csdn.net/fu\_kevin0606/article/details/65437594

# WallpaperPicker

rm500:/ $ dumpsys activity | grep mFoc

mFocusedActivity: ActivityRecord{bee84ec u0 com.android.wallpaperpicker2/com.android.wallpaperpicker.WallpaperPickerActivity t475}

C:\Users\key.guan\Desktop>adb pull /data/user/0/com.android.wallpaperpicker2/files/\

DefaultWallpaperInfo getDefaultWallpaper(Context context) {

android WallpaperPicker7.0源码分析

https://blog.csdn.net/A771642/article/details/77948878

# REF

《深入理解Android 卷III》第八章深入理解Android壁纸

https://blog.csdn.net/innost/article/details/47660645