 在前面的一系列文章中，我们学习了Android应用程序与SurfaceFlinger服务的关系，以及SurfaceFlinger服务的启动过程、初始化硬件帧缓冲区的过程、线程模型。SurfaceFlinger服务所做的一切都是为了给Android应用程序提服务的，即为Android应用程序渲染它们的UI。在本文中，我们就详细分析SurfaceFlinger服务渲染Android应用程序UI的过程。

        从前面[Android系统Surface制的SurfaceFlinger服务的线程模型分析](http://blog.csdn.net/luoshengyang/article/details/8062945)一文可以知道，SurfaceFlinger服务是通过它的UI渲染线程来将应用程序的UI渲染到硬件帧缓冲区中去的，因此，接下来我们就通过分析SurfaceFlinger服务的UI渲染线程的执行过程来分应用程序UI的渲染过程，这个过程如图1所示。

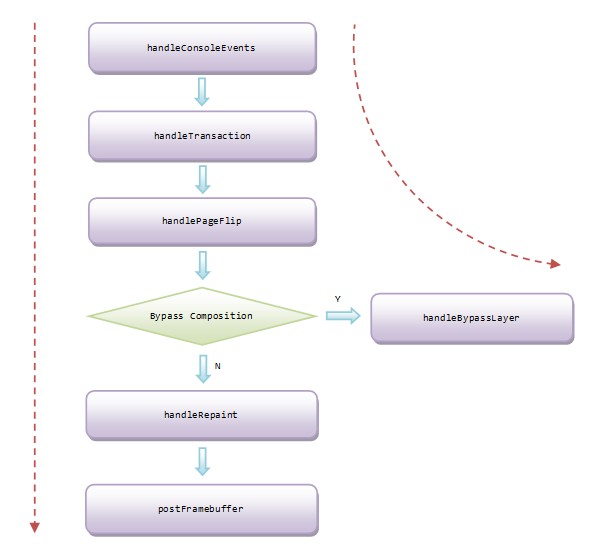


图1 SurfaceFlinger服务渲染应用程序UI的示意图

        从图1就可以看出，SurfaceFlinger服务的UI渲染线程的执行过程如下所示：

        1. 调用SurfaceFlinger类的成员函数handleConsoleEvents来处理控制台事件。

        2. 调用SurfaceFlinger类的成员函数handleTransaction来处理系统显示屏以及应用程序窗口的属性变化，例如大小、旋转方向变化等。

        3. 调用SurfaceFlinger类的成员函数handlePageFlip来让各个应用程序窗口设置它们当前所要渲染的图形缓冲区。

        4. 如果SurfaceFlinger服务在编译的时候指定了USE\_COMPOSITION\_BYPASS宏，并且当前需要渲染的应用程序窗口只有一个，那么就会调用SurfaceFlinger类的成员函数handleBypassLayer来直接将这个应用程序窗口的图形缓冲区渲染到硬件帧缓冲区中去，否则的话，就要调用SurfaceFlinger类的成员函数handleRepaint来合成所有的应用程序窗口的图形缓冲区到一个主图形缓冲区中去。

        5. 调用SurfaceFlinger类的成员函数postFramebuffer将前面得到的主图形缓冲区渲染到硬件帧缓冲区中去。

        前面[Android系统Surface制的SurfaceFlinger服务的线程模型分析](http://blog.csdn.net/luoshengyang/article/details/8062945)一文中，我们已经分析过第1步的实现了，而通过前面[Android应用程序与SurfaceFlinger服务的关系概述和学习计划](http://blog.csdn.net/luoshengyang/article/details/7846923)这一系列文章的学习，我们也已经了解了应用程序窗口的图形缓冲区的创建过程，因此，接下来我们就在这些知识的基础上来详细分析第2步到第5的实现，即分别分析SurfaceFlinger类的成员函数handleTransaction、handlePageFlip、handleBypassLayer和postFramebuffer的实现。

       1. handleTransaction

       SurfaceFlinger类的成员函数handleTransaction是用来处理系统显示屏以及应用程序窗口的属性变化的，这个过程如图2所示。

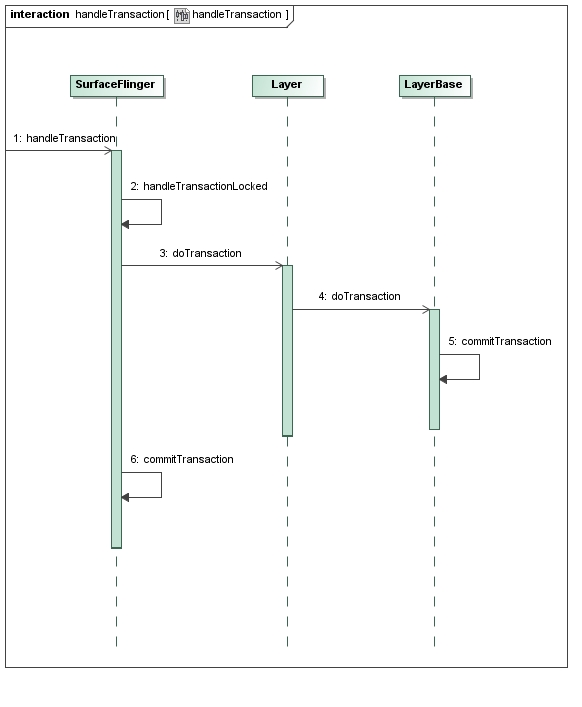


图2 系统显示屏以及应用程序窗口的属性变化处理过程

         这个过程可以分为6个步骤，接下来我们就详细分析每一个步骤。

         Step 1. SurfaceFlinger.handleTransaction

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::handleTransaction(uint32\_t transactionFlags)
2. {
3. Vector< sp<LayerBase> > ditchedLayers;
5. /\*
6. \* Perform and commit the transaction
7. \*/
9. { // scope for the lock
10. Mutex::Autolock \_l(mStateLock);
11. const nsecs\_t now = systemTime();
12. mDebugInTransaction = now;
13. handleTransactionLocked(transactionFlags, ditchedLayers);
14. mLastTransactionTime = systemTime() - now;
15. mDebugInTransaction = 0;
16. // here the transaction has been committed
17. }
19. /\*
20. \* Clean-up all layers that went away
21. \* (do this without the lock held)
22. \*/
23. const **size\_t** count = ditchedLayers.size();
24. for (**size\_t** i=0 ; i<count ; i++) {
25. if (ditchedLayers[i] != 0) {
26. //LOGD("ditching layer %p", ditchedLayers[i].get());
27. ditchedLayers[i]->ditch();
28. }
29. }
30. }

void SurfaceFlinger::handleTransaction(uint32\_t transactionFlags)

{

Vector< sp<LayerBase>> ditchedLayers;

/\*

\* Perform and commit the transaction

\*/

{ // scope for the lock

Mutex::Autolock \_l(mStateLock);

const nsecs\_t now = systemTime();

mDebugInTransaction = now;

handleTransactionLocked(transactionFlags, ditchedLayers);

mLastTransactionTime = systemTime() - now;

mDebugInTransaction = 0;

// here the transaction has been committed

}

/\*

\* Clean-up all layers that went away

\* (do this without the lock held)

\*/

const size\_t count = ditchedLayers.size();

for (size\_t i=0 ; i<count ; i++) {

if (ditchedLayers[i] != 0) {

//LOGD("ditching layer %p", ditchedLayers[i].get());

ditchedLayers[i]->ditch();

}

}

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

        SurfaceFlinger类的成员函数handleTransaction是通过调用另外一个成员函数handleTransactionLocked来处理系统显示屏以及应用程序窗口的属性变化的，而SurfaceFlinger类的成员函数handleTransactionLocked在处理完成系统显示屏以及应用程序窗口的属性变化之后，会返回系统中那些已经销毁了的应用程序窗口。

        从[Android应用程序与SurfaceFlinger服务的关系概述和学习计划](http://blog.csdn.net/luoshengyang/article/details/7846923)这一系列文章可以知道，在SurfaceFlinger服务这一侧，应用程序窗口一般是使用一个Layer对象来描述的，又由于Layer类是从LayerBase类继承下来的，因此，我们可以那些已经销毁了的应用程序窗口保存在一个类型为sp<LayerBase>的向量ditchedLayers中。

        SurfaceFlinger类的成员函数handleTransaction最后就调用保存在向量ditchedLayers中的每一个LayerBase对象的成员函数dtich来执行被销毁的应用程序窗口的清理操作，接下来我们就继续分析SurfaceFlinger类的成员函数handleTransactionLocked，看看它是如何处理系统显示屏以及应用程序窗口的属性变化的。

        Step 2. SurfaceFlinger.handleTransactionLocked

        SurfaceFlinger类的成员函数handleTransactionLocked定义在文件rameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中，我们分三段来阅读：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::handleTransactionLocked(
2. uint32\_t transactionFlags, Vector< sp<LayerBase> >& ditchedLayers)
3. {
4. const LayerVector& currentLayers(mCurrentState.layersSortedByZ);
5. const **size\_t** count = currentLayers.size();
7. /\*
8. \* Traversal of the children
9. \* (perform the transaction for each of them if needed)
10. \*/
12. const **bool** layersNeedTransaction = transactionFlags & eTraversalNeeded;
13. if (layersNeedTransaction) {
14. for (**size\_t** i=0 ; i<count ; i++) {
15. const sp<LayerBase>& layer = currentLayers[i];
16. uint32\_t trFlags = layer->getTransactionFlags(eTransactionNeeded);
17. if (!trFlags) continue;
19. const uint32\_t flags = layer->doTransaction(0);
20. if (flags & Layer::eVisibleRegion)
21. mVisibleRegionsDirty = true;
22. }
23. }

void SurfaceFlinger::handleTransactionLocked(

uint32\_t transactionFlags, Vector< sp<LayerBase>>& ditchedLayers)

{

const LayerVector& currentLayers(mCurrentState.layersSortedByZ);

const size\_t count = currentLayers.size();

/\*

\* Traversal of the children

\* (perform the transaction for each of them if needed)

\*/

const bool layersNeedTransaction = transactionFlags & eTraversalNeeded;

if (layersNeedTransaction) {

for (size\_t i=0 ; i<count ; i++) {

const sp<LayerBase>& layer = currentLayers[i];

uint32\_t trFlags = layer->getTransactionFlags(eTransactionNeeded);

if (!trFlags) continue;

const uint32\_t flags = layer->doTransaction(0);

if (flags & Layer::eVisibleRegion)

mVisibleRegionsDirty = true;

}

}

        这段代码用来处理应用程序窗口的属性变化。

        参数transactionFlags最开始是从SurfaceFlinger类的成员函数threadLoop传进来的。从前面[Android系统Surface制的SurfaceFlinger服务的线程模型分析](http://blog.csdn.net/luoshengyang/article/details/8062945)一文可以知道，SurfaceFlinger类的成员函数threadLoop在调用另外一个成员函数handleTransaction来处理系统显示屏以及应用程序窗口的属性变化之前，首先会调用成员函数getTransactionFlags来检查系统显示屏或者应用程序窗口的属性是否发生了变化。如果系统显示屏的属性发生了变化，那么传到这里的参数transactionFlags的eTransactionNeeded位就会等于1，而如果有应用程序窗口的属性发生了变化，那么传到这里的参数transactionFlags的eTraversalNeeded位就会等于1。为了方便描述，我们假设系统显示屏以及应用程序窗口的属性都发生了变化。

       SurfaceFlinger类的成员变量mCurrentState指向了一个State对象，用来描述SufaceFlinger服务的当前状态，其中，这个State对象的成员变量layersSortedByZ是一个类型为LayerVector的向量，它里面保存了SufaceFlinger服务当前所需要渲染的应用程序窗口，而这些应用程序窗口都是使用一个LayerBase对象来描述的。

       这段代码首先获得SufaceFlinger服务当前所需要渲染的应用程序窗口，接着再通过一个for循环来依次检查每一个应用程序窗口的属性是否发生了变化。如果某一个应用程序窗口的属性被修改过，那么调用用来描述这个应用程序窗口的一个LayerBase对象的成员函数getTransactionFlags得到的返回值trFlags就不会等于0，在这种情况下，这段代码就会调用这个LayerBase对象的成员函数doTransaction来处理对应的应用程序窗口的属性变化。

       在LayerBase类中，有一个类型为int32\_t的成员变量mTransactionFlags，每当SurfaceFlinger服务修改某一个应用程序窗口的属性时，都会将与其对应的LayerBase的成员变量mTransactionFlags的相应的位设置为1，这样LayerBase类的成员函数getTransactionFlags就可以通过这个成员变量来判断一个应用程序窗口的属性是否发生变化了。

       如果一个应用程序窗口发生的属性变化是可见区域发生了改变，那么对应的LayerBase对象的成员函数doTransaction的返回值flags的Layer::eVisibleRegion位就会等于1。在这种情况下，这段代码就会将 SurfaceFlinger类的成员变量mVisibleRegionsDirty的值设置为true，表示后面要重新计算各个应用程序窗口的可见区域。

      为了方便描述，我们假设发生了属性变化的应用程序窗口是一个普通类型的Surface，即用来描述它的实际是一个从LayerBase类继承下来的Layer对象。在这种情况下，前面实际上调用了Layer类的成员函数doTransaction来处理一个应用程序窗口的属性变化。在接下来的Step 3中，我们再详细分析Layer类的成员函数doTransaction的实现，现在我们接着往下阅读SurfaceFlinger类的成员函数handleTransactionLocked的代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. /\*
2. \* Perform our own transaction if needed
3. \*/
5. if (transactionFlags & eTransactionNeeded) {
6. if (mCurrentState.orientation != mDrawingState.orientation) {
7. // the orientation has changed, recompute all visible regions
8. // and invalidate everything.
10. const **int** dpy = 0;
11. const **int** orientation = mCurrentState.orientation;
12. const uint32\_t type = mCurrentState.orientationType;
13. GraphicPlane& plane(graphicPlane(dpy));
14. plane.setOrientation(orientation);
16. // update the shared control block
17. const DisplayHardware& hw(plane.displayHardware());
18. volatile display\_cblk\_t\* dcblk = mServerCblk->displays + dpy;
19. dcblk->orientation = orientation;
20. dcblk->w = plane.getWidth();
21. dcblk->h = plane.getHeight();
23. mVisibleRegionsDirty = true;
24. mDirtyRegion.set(hw.bounds());
25. }
27. if (mCurrentState.freezeDisplay != mDrawingState.freezeDisplay) {
28. // freezing or unfreezing the display -> trigger animation if needed
29. mFreezeDisplay = mCurrentState.freezeDisplay;
30. if (mFreezeDisplay)
31. mFreezeDisplayTime = 0;
32. }
34. if (currentLayers.size() > mDrawingState.layersSortedByZ.size()) {
35. // layers have been added
36. mVisibleRegionsDirty = true;
37. }
39. // some layers might have been removed, so
40. // we need to update the regions they're exposing.
41. if (mLayersRemoved) {
42. mLayersRemoved = false;
43. mVisibleRegionsDirty = true;
44. const LayerVector& previousLayers(mDrawingState.layersSortedByZ);
45. const **size\_t** count = previousLayers.size();
46. for (**size\_t** i=0 ; i<count ; i++) {
47. const sp<LayerBase>& layer(previousLayers[i]);
48. if (currentLayers.indexOf( layer ) < 0) {
49. // this layer is not visible anymore
50. ditchedLayers.add(layer);
51. mDirtyRegionRemovedLayer.orSelf(layer->visibleRegionScreen);
52. }
53. }
54. }
55. }

/\*

\* Perform our own transaction if needed

\*/

if (transactionFlags & eTransactionNeeded) {

if (mCurrentState.orientation != mDrawingState.orientation) {

// the orientation has changed, recompute all visible regions

// and invalidate everything.

const int dpy = 0;

const int orientation = mCurrentState.orientation;

const uint32\_t type = mCurrentState.orientationType;

GraphicPlane& plane(graphicPlane(dpy));

plane.setOrientation(orientation);

// update the shared control block

const DisplayHardware& hw(plane.displayHardware());

volatile display\_cblk\_t\* dcblk = mServerCblk->displays + dpy;

dcblk->orientation = orientation;

dcblk->w = plane.getWidth();

dcblk->h = plane.getHeight();

mVisibleRegionsDirty = true;

mDirtyRegion.set(hw.bounds());

}

if (mCurrentState.freezeDisplay != mDrawingState.freezeDisplay) {

// freezing or unfreezing the display -> trigger animation if needed

mFreezeDisplay = mCurrentState.freezeDisplay;

if (mFreezeDisplay)

mFreezeDisplayTime = 0;

}

if (currentLayers.size() > mDrawingState.layersSortedByZ.size()) {

// layers have been added

mVisibleRegionsDirty = true;

}

// some layers might have been removed, so

// we need to update the regions they're exposing.

if (mLayersRemoved) {

mLayersRemoved = false;

mVisibleRegionsDirty = true;

const LayerVector& previousLayers(mDrawingState.layersSortedByZ);

const size\_t count = previousLayers.size();

for (size\_t i=0 ; i<count ; i++) {

const sp<LayerBase>& layer(previousLayers[i]);

if (currentLayers.indexOf( layer ) < 0) {

// this layer is not visible anymore

ditchedLayers.add(layer);

mDirtyRegionRemovedLayer.orSelf(layer->visibleRegionScreen);

}

}

}

}

        这段代码用来处理系统显示屏的属性变化。

        在分析这段代码之前，我们首先了解SurfaceFlinger类的另外一个成员变量mDrawingState的含义。SurfaceFlinger类的成员变量mDrawingState与前面所介绍的成员变量mCurrentState类似，它的类型也为State，不过它是用来描述SufaceFlinger服务的上一次渲染状态的。通过这两个成员变量的比较，我们就可以知道系统显示屏的哪一个属性发生了变化。

        前面提到，当系统显示屏的属性发生了变化，那么参数transactionFlags的eTransactionNeeded位就会等于1，在这种情况，这段代码就需要完成四件事情。

        第一件事情是判断系统显示屏的旋转方向是否发生变化。State类的成员变量orientation用来描述显示屏的方向，因此，当SurfaceFlinger类的成员变量mCurrentState所描述的一个State对象的成员变量orientation的值不等于SurfaceFlinger类的成员变量mDrawingState所描述的一个State对象的成员变量orientation的值时，就说明系统显示屏的旋转方向发生了变化。在这种情况下，我们就需要将系统显示屏的旋转方向设置为SurfaceFlinger类的成员变量mCurrentState所描述的一个State对象的成员变量orientation的值，这是通过调用编号为0的一个GraphicPlane对象的成员函数setOrientation来实现的。

       SurfaceFlinger服务的UI渲染线程在初始化的过程中，除了会初始化硬件帧缓冲区之外，还会创建一个类型为surface\_flinger\_cblk\_t的对象，用来描述系统显示屏的信息，例如大小和旋转方向等，以便其它进程可以通过这个surface\_flinger\_cblk\_t对象来获得系统显示屏的信息。这个surface\_flinger\_cblk\_t对象就保存在SurfaceFlinger类的成员变量mServerCblk中。因此，当系统显示屏的旋转方向发生了变化时，我们还需要将变化后的旋转方向保存在SurfaceFlinger类的成员变量mServerCblk所描述的一个surface\_flinger\_cblk\_t对象中。由于系统显示屏的旋转方向变化一般意味着宽度和高度也会发生变化，因此，我们还需要将旋转发生变化后得到的系统显示屏的宽度和高度值保存在SurfaceFlinger类的成员变量mServerCblk所描述的一个surface\_flinger\_cblk\_t对象中。

       系统显示屏的旋转方向同时也意味着我们需要重新计算各个应用程序窗口的可见区域以及重新绘制整个显示屏，因此，在这种情况下，我们还需要将SurfaceFlinger类的成员变量mVisibleRegionsDirty的值设置为true，以及将SurfaceFlinger类的成员变量mDirtyRegion的大小设置为整个显示屏的大小，即将系统UI的脏区域设置为整个显示屏的大小。

       第二件事情是判断系统显示屏的冻结状态是否发生变化。State类的成员变量freezeDisplay用来描述显示屏的冻结状态，因此，当SurfaceFlinger类的成员变量mCurrentState所描述的一个State对象的成员变量freezeDisplay的值不等于SurfaceFlinger类的成员变量mDrawingState所描述的一个State对象的成员变量freezeDisplay的值时，就说明系统显示屏的冻结状态发生了变化。在这种情况下，我们就需要将SurfaceFlinger类的成员变量mFreezeDisplay的值设置为SurfaceFlinger类的成员变量mCurrentState所描述的一个State对象的成员变量freezeDisplay的值。如果显示屏的是由解冻状态变化为冻结状态的，那么还需要将显示屏的冻结时间设置为0，即将SurfaceFlinger类的成员变量mFreezeDisplayTime的值设置为0，以便可以将显示屏进入到冻结状态的最长时间设置为一个默认值，这一点可以参考前面[Android系统Surface制的SurfaceFlinger服务的线程模型分析](http://blog.csdn.net/luoshengyang/article/details/8062945)一文。

       第三件事情是判断是否新增了应用程序窗口。State类的成员变量layersSortedByZ是一个类型LayerVector的向量，里面保存的是SurfaceFlinger服务在某一个状态下所拥有的应用程序窗口，因此，当SurfaceFlinger类的成员变量mCurrentState所描述的一个State对象的成员变量layersSortedByZ所指向的一个向量的大小值大于SurfaceFlinger类的成员变量mDrawingState所描述的一个State对象的成员变量layersSortedByZ所指向的一个向量的大小值时，就说明系统新增了应用程序窗口。在这种情况下，我们就需要将SurfaceFlinger类的成员变量mVisibleRegionsDirty的值设置为true，以表示我们需要重新计算各个应用程序窗口的可见区域。

       第四件事情是判断是否移除了某些应用程序窗口。SurfaceFlinger类的成员变量mLayersRemoved用来描述是否有应用程序窗口被移除了。如果有有应用程序窗口被移除的话，那么这个成员变量的值就会等于true。在这种情况下，我们就需要是哪些应用程序窗口被移除了。计算的方法很简单，如果一个应用程序窗口存在于SurfaceFlinger类的成员变量mDrawingState所描述的一个State对象的成员变量layersSortedByZ所指向的一个向量中，但是不存在于SurfaceFlinger类的成员变量mCurrentState所描述的一个State对象的成员变量layersSortedByZ所指向的一个向量中，那么就说明这个应用程序窗口被移除了，因此，就需要将它保存输出参数ditchedLayers所描述的一个向量中，以便可以返回给上一步来处理。SurfaceFlinger类的成员变量mDirtyRegionRemovedLayer用来描述那些被移除了的应用程序窗口所占用的区域，因此，每当我们移除一个应用程序窗口的时候，都需要将它所占用的区域增加到SurfaceFlinger类的成员变量mDirtyRegionRemovedLayer所描述的一个区域去。

      当处理完成那些被移除的应用程序窗口之后，我们就需要将SurfaceFlinger类的成员变量mLayersRemoved的值设置为false，并且将SurfaceFlinger类的成员变量mVisibleRegionsDirty的值设置为true，以表示我们需要重新计算现存的各个应用程序窗口的可见区域。

      处理完成系统显示屏的属性变化之后，我们接着向下阅读SurfaceFlinger类的成员函数handleTransactionLocked的最后一行代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. commitTransaction();

commitTransaction();

}

      这段代码只有一行，即调用SurfaceFlinger类的成员函数commitTransaction来告诉SurfaceFlinger服务，系统显示屏以及各个应用程序窗口的属性变化已经处理完毕，这时候SurfaceFlinger服务就可以切换状态了。在后面的Step 6中，我们再详细分析SurfaceFlinger类的成员函数commitTransaction的实现。

      接下来，我们就继续分析Layer类的成员函数doTransaction的实现，以便可以了解应用程序窗口的属性变化的处理过程。

      Step 3. Layer.doTransaction

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. uint32\_t Layer::doTransaction(uint32\_t flags)
2. {
3. const Layer::State& front(drawingState());
4. const Layer::State& temp(currentState());
6. const **bool** sizeChanged = (front.requested\_w != temp.requested\_w) ||
7. (front.requested\_h != temp.requested\_h);
9. if (sizeChanged) {
10. ......
12. if (!isFixedSize()) {
13. // we're being resized and there is a freeze display request,
14. // acquire a freeze lock, so that the screen stays put
15. // until we've redrawn at the new size; this is to avoid
16. // glitches upon orientation changes.
17. if (mFlinger->hasFreezeRequest()) {
18. // if the surface is hidden, don't try to acquire the
19. // freeze lock, since hidden surfaces may never redraw
20. if (!(front.flags & ISurfaceComposer::eLayerHidden)) {
21. mFreezeLock = mFlinger->getFreezeLock();
22. }
23. }
25. // this will make sure LayerBase::doTransaction doesn't update
26. // the drawing state's size
27. Layer::State& editDraw(mDrawingState);
28. editDraw.requested\_w = temp.requested\_w;
29. editDraw.requested\_h = temp.requested\_h;
31. // record the new size, form this point on, when the client request
32. // a buffer, it'll get the new size.
33. setBufferSize(temp.requested\_w, temp.requested\_h);
35. ClientRef::Access sharedClient(mUserClientRef);
36. SharedBufferServer\* lcblk(sharedClient.get());
37. if (lcblk) {
38. // all buffers need reallocation
39. lcblk->reallocateAll();
40. }
41. } else {
42. // record the new size
43. setBufferSize(temp.requested\_w, temp.requested\_h);
44. }
45. }
47. if (temp.sequence != front.sequence) {
48. if (temp.flags & ISurfaceComposer::eLayerHidden || temp.alpha == 0) {
49. // this surface is now hidden, so it shouldn't hold a freeze lock
50. // (it may never redraw, which is fine if it is hidden)
51. mFreezeLock.clear();
52. }
53. }
55. return LayerBase::doTransaction(flags);
56. }

uint32\_t Layer::doTransaction(uint32\_t flags)

{

const Layer::State& front(drawingState());

const Layer::State& temp(currentState());

const bool sizeChanged = (front.requested\_w != temp.requested\_w) ||

(front.requested\_h != temp.requested\_h);

if (sizeChanged) {

......

if (!isFixedSize()) {

// we're being resized and there is a freeze display request,

// acquire a freeze lock, so that the screen stays put

// until we've redrawn at the new size; this is to avoid

// glitches upon orientation changes.

if (mFlinger->hasFreezeRequest()) {

// if the surface is hidden, don't try to acquire the

// freeze lock, since hidden surfaces may never redraw

if (!(front.flags & ISurfaceComposer::eLayerHidden)) {

mFreezeLock = mFlinger->getFreezeLock();

}

}

// this will make sure LayerBase::doTransaction doesn't update

// the drawing state's size

Layer::State& editDraw(mDrawingState);

editDraw.requested\_w = temp.requested\_w;

editDraw.requested\_h = temp.requested\_h;

// record the new size, form this point on, when the client request

// a buffer, it'll get the new size.

setBufferSize(temp.requested\_w, temp.requested\_h);

ClientRef::Access sharedClient(mUserClientRef);

SharedBufferServer\* lcblk(sharedClient.get());

if (lcblk) {

// all buffers need reallocation

lcblk->reallocateAll();

}

} else {

// record the new size

setBufferSize(temp.requested\_w, temp.requested\_h);

}

}

if (temp.sequence != front.sequence) {

if (temp.flags & ISurfaceComposer::eLayerHidden || temp.alpha == 0) {

// this surface is now hidden, so it shouldn't hold a freeze lock

// (it may never redraw, which is fine if it is hidden)

mFreezeLock.clear();

}

}

return LayerBase::doTransaction(flags);

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/Layer.cpp中。

       和SurfaceFlinger服务类似，每一个应用程序窗口在内部也分别使用两个类型为State的成员变量mDrawingState和mCurrentState来描述上一次的渲染状态和下一次的渲染状态。这两个成员变量是从LayerBase类继承下来的，用来描述应用程序窗口的Layer类可以分别通过从父类LayerBase继承下来的成员函数drawingState和currentState来访问它们。注意，这里所说的State类是定义在LayerBase类内部的，而SurfaceFlinger服务使用的State类是定义在SurfaceFlinger类内部的，它们是不一样的。

       Layer类的成员函数doTransaction首先调用从父类LayerBase继承下来的成员函数drawingState和currentState来获得当前正在处理的应用程序窗口的上一次的渲染状态和下一次的渲染状态，并且分别保存在两个类型为State的变量front和temp中。

       State类的成员变量requested\_w和requested\_h分别用来描述应用程序窗口的宽度和高度，因此，当State变量front的成员变量requested\_w和requested\_h不等于State变量temp的成员变量requested\_w和requested\_h时，我们就会得到变量sizeChanged的值等于true，表示当前正在处理的应用程序窗口的大小发生了变化。在分析Layer类的成员函数doTransaction处理应用程序窗口的大小变化时，我们先介绍Layer类的成员变量mFixedSize的含义。

       Layer类的成员变量mFixedSize是一个布尔变量，它的值可以通过Layer类的成员函数isFixedSize来获得。从前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文可以知道，当Android应用程序请求SurfaceFlinger服务分配一块图形缓冲区时， Layer类的成员函数requestBuffer就会被调用。这时候Android应用程序会传递两个参数reqWidth和reqHeight过来，表示请求分配的图形缓冲区的宽度和高度。这两个参数是可以同时等于0的，表示使用默认的宽度和高度值来创建所请求的图形缓冲区。这两个默认的宽度和高度值即等于当前所处理的应用程序窗口的宽度和高度值，而后者的宽度和高度值是在其创建的时候指定的，这一点可以参考[Android应用程序请求SurfaceFlinger服务创建Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7884628)一文。

       Layer类的成员函数requestBuffer的参数reqWidth和reqHeight的值等于0意味着什么呢？从前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文还可以知道，Android应用程序在请求SurfaceFlinger服务分配一块图形缓冲区之前，会通过在Surface类内部定义的BufferInfo类的成员函数validateBuffer来检查当前所处理的应用程序窗口的大小是否发生了变化。如果发生了变化，那么Android应用程序就会忽略掉缓存自己一侧的图形缓冲区，而去SurfaceFlinger服务请求新的图形缓冲区，因为那些缓存的图形缓冲区由于与它们所关联的应用程序窗口大小发生了变化而变为无效了。但是有一种特殊情况，在Android应用程序这一侧，用来描述应用程序窗口的Surface类可以不维护应用程序窗口的大小值。在这种情况下，Surface类就会将与它所关联的应用程序窗口的大小值设置为0，这意味着Android应用程序每次为这个应用程序窗口向SurfaceFlinger服务请求分配图形缓冲区之前，都认为这个应用程序窗口的大小值没有发生变化，同时传递给Layer类的成员函数requestBuffer的参数reqWidth和reqHeight的值会等于0。事实上，一个应用程序窗口的大小是随时都可以发生变化的，比如，我们可以通过调用用来在Android应用程序和SurfaceFlinger服务建立连接的一个类型为Client的Binder对象的成员函数setState来改变一个应用程序窗口的大小，而一旦一个应用程序窗口的大小发生了变化，我们正在分析Layer类的成员函数doTransaction就会被调用。

      从上面的分析就可以得出一个结论，Layer类的成员函数doTransaction在处理应用程序窗口大小变化时，需要考虑Android应用程序每次在为该应用程序窗口向SurfaceFlinger服务请求分配图形缓冲区之前，是否有能力去判断之前为该应用程序窗口缓存的图形缓冲区的有效性。如果没有的话，那么Layer类的成员函数doTransaction就需要将为该应用程序窗口缓存的图形缓冲区设置为无效，以便以后Android应用程序可以请求SurfaceFlinger服务分配新的、大小正确的图形缓冲区。从前面的分析还可以知道，当Android应用程序没有能力去判断之前为一个应用程序窗口所缓存的图形缓冲区的有效性时，那么之前在请求分配这些图形缓冲区时，传递给Layer类的成员函数requestBuffer的参数reqWidth和reqHeight的值就会等于0，这时候Layer类的成员函数requestBuffer就会将Layer类的成员变量mFixedSize的值设置为false。

      接下来，我们就分别根据Layer类的成员变量mFixedSize是等于true还是false来分析Layer类的成员函数doTransaction处理一个应用程序窗口大小发生变化的过程。

      当Layer类的成员变量mFixedSize的值等于true时，Layer类的成员函数doTransaction的处理很简单，它只是调用另外一个成员函数setBufferSize来将新的应用程序窗口大小记录下来，即保存在Layer类的成员变量mWidth和mHeight中。

      当Layer类的成员变量mFixedSize的值等于false时，由于Android应用程序没有能力去判断之前为一个应用程序窗口所缓存的图形缓冲区的有效性，因此，Layer类的成员函数doTransaction除了会调用外一个成员函数setBufferSize来将新的应用程序窗口大小记录下来之外，还会通过一个SharedBufferServer对象的成员函数reallocateAll来将为当前正在处理的应用程序窗口所缓存的图形缓冲区设置为无效。在前面[Android应用程序请求SurfaceFlinger服务创建Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7884628)一文中，我们已经分析过SharedBufferServer类的作用了，它是用来维护Android应用程序与SurfaceFlinger服务之间的共享UI元数据的，通过它可以将对应的图形缓冲区设置为无效。

      当Layer类的成员变量mFixedSize的值等于false时，Layer类的成员函数doTransaction还会提前将成员变量mCurrentState所描述的一个State对象的成员变量requested\_w和requested\_h的值保存到成员变量mDrawingState所描述的一个State对象的成员变量requested\_w和requested\_h中去，这是为了避免后面调用父类LayerBase的成员函数doTransaction时，会返回一个Layer::eVisibleRegion位不等于0的标志值给前面的Step 2，而这将会导致SurfaceFlinger服务马上重新计算各个应用程序窗口的可见区域。现在不返回一个Layer::eVisibleRegion位不等于0的标志值给前面的Step 2，就会等到下次渲染当前正在处理的应用程序窗口时再重新计算各个应用程序窗口的可见区域。

      此外，当Layer类的成员变量mFixedSize的值等于false时，Layer类的成员函数doTransaction还会检查系统显示屏是否正处于冻结的状态，这是通过调用Layer类的成员变量mFlinger所指向的SurfaceFlinger服务的成员函数hasFreezeRequest来实现的。如果系统显示屏处于冻结的状态中，并且当前正在处理的应用程序窗口处于可见状态，即变量front所描述的State对象的成员变量flags的ISurfaceComposer::eLayerHidden位等于0，那么Layer类的成员函数doTransaction还会请求获得一个用来冻结系统显示屏的类型为FreezeLock的锁，并且将这个锁保存在Layer类的成员变量mFreezeLock中。这样就可以等到下次渲染当前正在处理的应用程序窗口时，再来刷新系统的UI。

      State类的另外一个成员变量sequence用来描述一个应用程序窗口的其它属性是否发生过变化，例如，X、Y和Z轴位置、透明度等除了大小之外的属性。每当这些属性发生了变化，那么State类的成员变量sequence的值就会比原来增加1。Layer类的成员函数doTransaction再接下来就通过比较变量temp和front所描述的State对象的成员变量sequence的值是否相等来判断当前正在处理的应用程序窗口的其它属性是否发生变化。如果发生过变化，并且当前正在处理的应用程序窗口处于不可见状态或者处于完全透明的状态，那么Layer类的成员函数doTransaction就会释放之前所持有的用来冻结系统显示屏的一个类型为FreezeLock的锁，这是通过调用Layer类的成员变量mFreezeLock所指向的一个FreezeLock对象的成员函数clear来实现的，这样就可以避免阻止SurfaceFlinger服务更新系统UI。

      最后，Layer类的成员函数doTransaction调用父类LayerBase的成员函数doTransaction来继续处理应用程序窗口的属性变化。

      Step 4. LayerBase.doTransaction

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. uint32\_t LayerBase::doTransaction(uint32\_t flags)
2. {
3. const Layer::State& front(drawingState());
4. const Layer::State& temp(currentState());
6. if ((front.requested\_w != temp.requested\_w) ||
7. (front.requested\_h != temp.requested\_h))  {
8. // resize the layer, set the physical size to the requested size
9. Layer::State& editTemp(currentState());
10. editTemp.w = temp.requested\_w;
11. editTemp.h = temp.requested\_h;
12. }
14. if ((front.w != temp.w) || (front.h != temp.h)) {
15. // invalidate and recompute the visible regions if needed
16. flags |= Layer::eVisibleRegion;
17. }
19. if (temp.sequence != front.sequence) {
20. // invalidate and recompute the visible regions if needed
21. flags |= eVisibleRegion;
22. this->contentDirty = true;
24. // we may use linear filtering, if the matrix scales us
25. const uint8\_t type = temp.transform.getType();
26. mNeedsFiltering = (!temp.transform.preserveRects() ||
27. (type >= Transform::SCALE));
28. }
30. // Commit the transaction
31. commitTransaction();
32. return flags;
33. }

uint32\_t LayerBase::doTransaction(uint32\_t flags)

{

const Layer::State& front(drawingState());

const Layer::State& temp(currentState());

if ((front.requested\_w != temp.requested\_w) ||

(front.requested\_h != temp.requested\_h)) {

// resize the layer, set the physical size to the requested size

Layer::State& editTemp(currentState());

editTemp.w = temp.requested\_w;

editTemp.h = temp.requested\_h;

}

if ((front.w != temp.w) || (front.h != temp.h)) {

// invalidate and recompute the visible regions if needed

flags |= Layer::eVisibleRegion;

}

if (temp.sequence != front.sequence) {

// invalidate and recompute the visible regions if needed

flags |= eVisibleRegion;

this->contentDirty = true;

// we may use linear filtering, if the matrix scales us

const uint8\_t type = temp.transform.getType();

mNeedsFiltering = (!temp.transform.preserveRects() ||

(type >= Transform::SCALE));

}

// Commit the transaction

commitTransaction();

return flags;

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/LayerBase.cpp中。

       与前面的Step 3一样，LayerBase类的成员函数doTransaction首先通过成员函数drawingState和currentState来获得用来描述当前正在处理的应用程序窗口的当前渲染状态的一个State对象front和下一次渲染状态的一个State对象temp，接下来就可以通过这两个State对象的成员变量requested\_w和requested\_h来判断当前正在处理的应用程序窗口的大小是否发生了变化。如果发生了变化，那么就将新的应用程序窗口大小值保存在用来描述下一次渲染状态的State对象editTemp的成员变量w和h中。

       State类的成员变量requested\_w、requested\_h和w、h的区别在于，前两者用来描述应用程序窗口的大小，后两者虽然也是用来描述应用程序窗口的大小，不过它们的作用是用来判断是否需要重新计算系统中的各个应用程序窗口的可见区域的。一旦用来描述应用程序窗口的当前渲染状态的State对象front和下一次渲染状态的State对象temp的成员变量w和h不相等，那么就说明需要系统中的各个应用程序窗口的可见区域，这是通过将将返回值flags的Layer::eVisibleRegion位设置为1来实现的。

       LayerBase类的成员函数doTransaction接下来还会继续判断State对象front和temp的成员变量sequence的值是否相等。如果不相等，那么就说明当前正在处理的应用程序窗口的其它属性，例如位置和透明度等发生了变化。在这种情况下，SurfaceFlinger服务也是需要重新计算系统中的各个应用程序窗口的可见区域的。因此，这时候 LayerBase类的成员函数doTransaction就会将它的返回值flags的Layer::eVisibleRegion位设置为1，并且将Layer类的成员变量contentDirty的值设置为true，表示当前正在处理的应用程序窗口的内容是脏的，需要重新渲染。

       LayerBase类的成员函数doTransaction最后还会继续调用另外一个成员函数commitTransaction来结束对当前正在处理的应用程序窗口的属性变化处理，以及将返回值flags返回给前面的Step 2，即SurfaceFlinger类的成员函数handleTransactionLocked。SurfaceFlinger类的成员函数handleTransactionLocked一旦发现这个返回值的Layer::eVisibleRegion位的值为1，那么就会将SurfaceFlinger类的成员变量mVisibleRegionsDirty的值设置为true，以便接下来可以重新计算系统中的各个应用程序窗口的可见区域。

       接下来，我们就继续分析LayerBase类的成员函数commitTransaction的实现。

       Step 5. LayerBase.commitTransaction

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void LayerBase::commitTransaction() {
2. mDrawingState = mCurrentState;
3. }

void LayerBase::commitTransaction() {

mDrawingState = mCurrentState;

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/LayerBase.cpp中。

      由于此时当前正在处理的应用程序窗口的属性变化已经处理完成了，因此，LayerBase类的成员函数commitTransaction就将用来描述下一次渲染状态的成员变量mCurrentState所描述的一个State对象保存在另外一个用来描述当前渲染状态的成员变量mDrawingState中去。

      这一步执行完成之后，返回到前面的Step 2中，即SurfaceFlinger类的成员函数handleTransactionLocked中，这时候系统显示屏的属性变化以及各个应用程序的属性变化就都已经处理完成了，SurfaceFlinger类的成员函数handleTransactionLocked最后就会调用另外一个成员函数commitTransaction来结束整个属性变化处理过程。

      Step 6. SurfaceFlinger.commitTransaction

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::commitTransaction()
2. {
3. mDrawingState = mCurrentState;
4. mResizeTransationPending = false;
5. mTransactionCV.broadcast();
6. }

void SurfaceFlinger::commitTransaction()

{

mDrawingState = mCurrentState;

mResizeTransationPending = false;

mTransactionCV.broadcast();

}

      这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

      前面提到，SurfaceFlinger类的成员变量mDrawingState用来描述SufaceFlinger服务的当前状态，而成员变量mCurrentState用来描述SufaceFlinger服务的下一个状态，由于这时候SufaceFlinger服务的状态变化已经处理完成了，因此，SurfaceFlinger类的commitTransaction就将成员变量mCurrentState所指向的一个State对象保存在成员变量mDrawingState中，接着又将成员变量mResizeTransationPending的值设置为false，表示SufaceFlinger服务已经处理过系统显示屏或者应用程序窗口的大小变化了，最后调用成员变量mTransactionCV所描述的一个条件变量的成员函数broadcast来通知其它线程，例如Binder线程，SufaceFlinger服务已经处理完成一次属性变化了。

      至此，我们就分析完成系统显示屏以及应用程序窗口的属性变化的处理过程，接下来我们继续分析各个应用程序窗口是如何设置它们当前所要渲染的图形缓冲区的，即SurfaceFlinger类的成员函数handlePageFlip的实现。

      2. handlePageFlip

      SurfaceFlinger类的成员函数handlePageFlip是用来让各个应用程序窗口设置它们当前所要渲染的图形缓冲区的，在这个过程中，SurfaceFlinger服务还会计算各个应用程序窗口的可见区域，如图3所示。

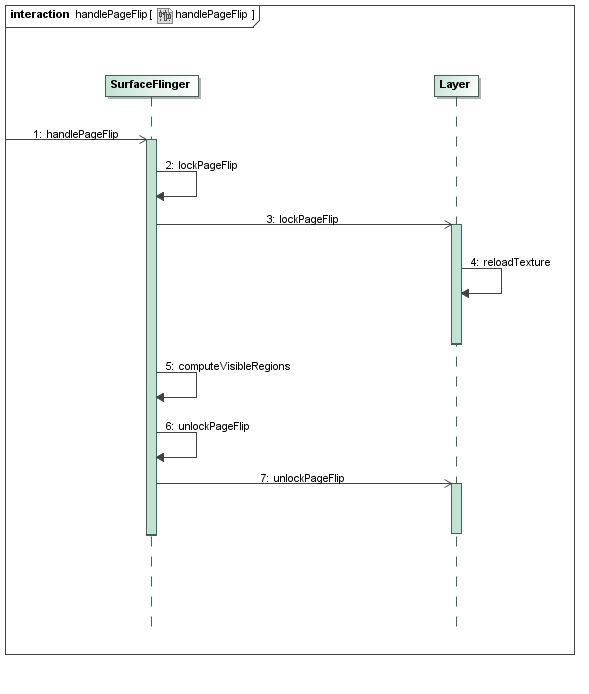


图3 应用程序窗口设置当前所要渲染的图形缓冲区的过程

        这个过程可以分为7个步骤，接下来我们就详细分析每一个步骤。

        Step 1. SurfaceFlinger.handlePageFlip

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::handlePageFlip()
2. {
3. **bool** visibleRegions = mVisibleRegionsDirty;
4. LayerVector& currentLayers = const\_cast<LayerVector&>(
5. mDrawingState.layersSortedByZ);
6. visibleRegions |= lockPageFlip(currentLayers);
8. const DisplayHardware& hw = graphicPlane(0).displayHardware();
9. const Region screenRegion(hw.bounds());
10. if (visibleRegions) {
11. Region opaqueRegion;
12. computeVisibleRegions(currentLayers, mDirtyRegion, opaqueRegion);
14. /\*
15. \*  rebuild the visible layer list
16. \*/
17. mVisibleLayersSortedByZ.clear();
18. const LayerVector& currentLayers(mDrawingState.layersSortedByZ);
19. **size\_t** count = currentLayers.size();
20. mVisibleLayersSortedByZ.setCapacity(count);
21. for (**size\_t** i=0 ; i<count ; i++) {
22. if (!currentLayers[i]->visibleRegionScreen.isEmpty())
23. mVisibleLayersSortedByZ.add(currentLayers[i]);
24. }
26. #ifdef USE\_COMPOSITION\_BYPASS
27. sp<LayerBase> bypassLayer;
28. const **size\_t** numVisibleLayers = mVisibleLayersSortedByZ.size();
29. if (numVisibleLayers == 1) {
30. const sp<LayerBase>& candidate(mVisibleLayersSortedByZ[0]);
31. const Region& visibleRegion(candidate->visibleRegionScreen);
32. const Region reminder(screenRegion.subtract(visibleRegion));
33. if (reminder.isEmpty()) {
34. // fullscreen candidate!
35. bypassLayer = candidate;
36. }
37. }
38. setBypassLayer(bypassLayer);
39. #endif
41. mWormholeRegion = screenRegion.subtract(opaqueRegion);
42. mVisibleRegionsDirty = false;
43. }
45. unlockPageFlip(currentLayers);
46. mDirtyRegion.andSelf(screenRegion);
47. }

void SurfaceFlinger::handlePageFlip()

{

bool visibleRegions = mVisibleRegionsDirty;

LayerVector& currentLayers = const\_cast<LayerVector&>(

mDrawingState.layersSortedByZ);

visibleRegions |= lockPageFlip(currentLayers);

const DisplayHardware& hw = graphicPlane(0).displayHardware();

const Region screenRegion(hw.bounds());

if (visibleRegions) {

Region opaqueRegion;

computeVisibleRegions(currentLayers, mDirtyRegion, opaqueRegion);

/\*

\* rebuild the visible layer list

\*/

mVisibleLayersSortedByZ.clear();

const LayerVector& currentLayers(mDrawingState.layersSortedByZ);

size\_t count = currentLayers.size();

mVisibleLayersSortedByZ.setCapacity(count);

for (size\_t i=0 ; i<count ; i++) {

if (!currentLayers[i]->visibleRegionScreen.isEmpty())

mVisibleLayersSortedByZ.add(currentLayers[i]);

}

#ifdef USE\_COMPOSITION\_BYPASS

sp<LayerBase> bypassLayer;

const size\_t numVisibleLayers = mVisibleLayersSortedByZ.size();

if (numVisibleLayers == 1) {

const sp<LayerBase>& candidate(mVisibleLayersSortedByZ[0]);

const Region& visibleRegion(candidate->visibleRegionScreen);

const Region reminder(screenRegion.subtract(visibleRegion));

if (reminder.isEmpty()) {

// fullscreen candidate!

bypassLayer = candidate;

}

}

setBypassLayer(bypassLayer);

#endif

mWormholeRegion = screenRegion.subtract(opaqueRegion);

mVisibleRegionsDirty = false;

}

unlockPageFlip(currentLayers);

mDirtyRegion.andSelf(screenRegion);

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

        函数首先将SurfaceFlinger类的成员变量mVisibleRegionsDirty的值保存在变量visibleRegions中。从前面第1部分的内容可以知道，当SurfaceFlinger服务在处理系统显示屏以及各个应用程序窗口的属性变化时，如果发现需要重新计算各个应用程序窗口的可见区域，那么就会将SurfaceFlinger类的成员变量mVisibleRegionsDirty的值设置为true。

       函数接下来通过SurfaceFlinger类的成员变量mDrawingState所描述的一个State对象的成员变量layersSortedByZ来获得系统当前所有的应用程序窗口，并且保存在一个类型为LayerVector的向量currentLayers中。有了系统当前所有的应用程序窗口之后，就可以通过调用SurfaceFlinger类的成员函数lockPageFlip来让它们设置自己当前所要渲染  
的图形缓冲区。在后面的Step 2中，我们再详细分析SurfaceFlinger类的成员函数lockPageFlip的实现。

      系统中的各个应用程序窗口在设置自己当前所要渲染的图形缓冲区的过程中，有可能会改变自己的大小。在这种情况下，它们就会通过SurfaceFlinger类的成员函数lockPageFlip来返回一个true值来给SurfaceFlinger类的成员函数handlePageFlip，这时候得到的变量visibleRegions的值就会等于true，表示需要重新计算各个应用程序窗口的可见区域。

      综合上述情况，无论变量visibleRegions的值是由于什么原因等于true，都说明SurfaceFlinger服务需要重新计算各个应用程序窗口的可见区域，这个计算的工作是通过调用SurfaceFlinger类的另外一个成员函数computeVisibleRegions来完成的。在后面的Step 5中，我们再详细分析SurfaceFlinger类的成员函数computeVisibleRegions的实现。

      SurfaceFlinger类的成员函数computeVisibleRegions在计算完成各个应用程序窗口的可见区域之后，会得到一个全局不透明区域，保存在输出参数opaqueRegion中。这个全局不透明区域就是接下来需要渲染的，一般情况下，它的大小就应该等于显示屏的大小，即变量screenRegion所描述的区域。在异常情况下，可能会导致显示屏区域screenRegion大于全局不透明区域opaqueRegion，这时候前者减去后者就可以得到一些称为“虫洞”的区域，它们保存在SurfaceFlinger类的成员变量mWormholeRegion中。由于这些虫洞区域不会被各个应用程序窗口覆盖，因此，SurfaceFlinger服务需要对它们进行特殊处理，即以一种特殊的方式来渲染它们。在后面的第4部分内容中，我们就会看到SurfaceFlinger服务是通过调用SurfaceFlinger类的成员函数drawWormhole来渲染这些虫洞的。

      SurfaceFlinger类的成员函数computeVisibleRegions在计算完成各个应用程序窗口的可见区域之后，各个应用程序窗口的可见区域就会保存在用来描述它们的一个LayerBase对象的成员变量visibleRegionScreen中，因此，SurfaceFlinger类的成员函数handlePageFlip就会通过这个成员变量来排除掉那些可见区域为空的应用程序窗口，并且将所有可见区域不为空的应用程序窗口按照它们的Z轴大小保存在SurfaceFlinger类的成员变量mVisibleLayersSortedByZ所描述的一个向量中。

      经过前面的操作之后，SurfaceFlinger类的成员函数handlePageFlip就可以将成员变量mVisibleRegionsDirty的值设置为false了，因为这时候系统中各个应用程序窗口的可见区域都已经重新计算过了。

      SurfaceFlinger类的成员函数handlePageFlip最后就调用另外一个成员函数unlockPageFlip来让各个应用程序窗口执行一些善后的工作，例如，让各个应用程序窗口检查自己的可见区域是否等于空，如果等于空的话，那么就需要将它们之前所获得的用来冻结显示屏的锁释放掉，以避免阻止SurfaceFlinger服务渲染系统UI。在后面的Step 6中，我们再详细分析SurfaceFlinger类的成员函数unlockPageFlip的实现。

      SurfaceFlinger类的成员函数handlePageFlip最后还需要做的另外一件事情是将设置系统的脏区域，这个脏区域保存在SurfaceFlinger类的成员变量mDirtyRegion中，它同样是作为一个输出参数来传递给SurfaceFlinger类的成员函数computeVisibleRegions的，以便后者在计算各个应用程序窗口的可见区域时，可以将获得的系统脏区域保存在它里面。我们同样是在后面的Step 5中分析SurfaceFlinger类的成员函数computeVisibleRegions的实现时，再看看系统的脏区域是如何计算的。

      有一种特殊情况，即SurfaceFlinger类的成员函数handlePageFlip在重新计算完成各个应用程序窗口的可见区域后，如果满足以下三个条件：

     1. SurfaceFlinger服务在编译时指定了宏USE\_COMPOSITION\_BYPASS；

     2. 当前要渲染的应用程序窗口只有一个，即SurfaceFlinger类的成员变量mVisibleLayersSortedByZ所描述的一个向量的大小等于1；

     3. 当前要渲染的一个唯一的应用程序窗口的可见区域的大小不为空。

     那么SurfaceFlinger服务就会直接将这个唯一需要渲染的应用程序窗口的图形缓冲区渲染到硬件帧缓冲区中去，以跳过后面的合成各个应用程序窗口的图形缓冲区的操作。在这种情况下，这个唯一需要渲染的应用程序窗口会通过SurfaceFlinger类的成员函数setBypassLayer记录起来。

      SurfaceFlinger类的成员函数setBypassLayer的实现如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::setBypassLayer(const sp<LayerBase>& layer)
2. {
3. // if this layer is already the bypass layer, do nothing
4. sp<Layer> cur(mBypassLayer.promote());
5. if (mBypassLayer == layer) {
6. if (cur != NULL) {
7. cur->updateBuffersOrientation();
8. }
9. return;
10. }
12. // clear the current bypass layer
13. mBypassLayer.clear();
14. if (cur != 0) {
15. cur->setBypass(false);
16. cur.clear();
17. }
19. // set new bypass layer
20. if (layer != 0) {
21. if (layer->setBypass(true)) {
22. mBypassLayer = static\_cast<Layer\*>(layer.get());
23. }
24. }
25. }

void SurfaceFlinger::setBypassLayer(const sp<LayerBase>& layer)

{

// if this layer is already the bypass layer, do nothing

sp<Layer> cur(mBypassLayer.promote());

if (mBypassLayer == layer) {

if (cur != NULL) {

cur->updateBuffersOrientation();

}

return;

}

// clear the current bypass layer

mBypassLayer.clear();

if (cur != 0) {

cur->setBypass(false);

cur.clear();

}

// set new bypass layer

if (layer != 0) {

if (layer->setBypass(true)) {

mBypassLayer = static\_cast<Layer\*>(layer.get());

}

}

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

       从这里就可以看出，在指定了编译宏USE\_COMPOSITION\_BYPASS的情况下，SurfaceFlinger服务唯一需要渲染的应用程序窗口就保存在SurfaceFlinger类的一个类型为Layer的成员变量mBypassLayer中，并且会调用用来这个Layer对象的成员函数setBypass来将它里面的一个成员变量mBypassState的值设置为true。在后面的第3部分内容中，我们再详细分析这个唯一需要渲染的应用程序窗口的图形缓冲区是如何直接渲染到硬件帧缓冲区中去的。

      接下来，我们就继续分析SurfaceFlinger类的成员函数lockPageFlip、computeVisibleRegions和unlockPageFlip的实现，以便可以分别了解各个应用程序窗口是如何设置自己当前需要渲染的图形缓冲区的、SurfaceFlinger服务是如何计算各个应用程序窗口的可见区域的，以及各个应用程序窗口的可见区域计算完成之后的善后工作是什么。

      Step 2. SurfaceFlinger.lockPageFlip

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. **bool** SurfaceFlinger::lockPageFlip(const LayerVector& currentLayers)
2. {
3. **bool** recomputeVisibleRegions = false;
4. **size\_t** count = currentLayers.size();
5. sp<LayerBase> const\* layers = currentLayers.array();
6. for (**size\_t** i=0 ; i<count ; i++) {
7. const sp<LayerBase>& layer(layers[i]);
8. layer->lockPageFlip(recomputeVisibleRegions);
9. }
10. return recomputeVisibleRegions;
11. }

bool SurfaceFlinger::lockPageFlip(const LayerVector& currentLayers)

{

bool recomputeVisibleRegions = false;

size\_t count = currentLayers.size();

sp<LayerBase> const\* layers = currentLayers.array();

for (size\_t i=0 ; i<count ; i++) {

const sp<LayerBase>& layer(layers[i]);

layer->lockPageFlip(recomputeVisibleRegions);

}

return recomputeVisibleRegions;

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

        在前面[Android应用程序与SurfaceFlinger服务的关系概述和学习计划](http://blog.csdn.net/luoshengyang/article/details/7846923)的一系列文章中，我们假设在SurfaceFlinger服务这一侧，每一个应用程序窗口都是使用一个Layer对象来描述，这些Layer对象都是从LayerBase继承下来的，因此它们可以保存在一个类型为LayerBase的向量中。

        从前面的调用过程可以知道，参数currentLayers里面保存的一系列LayerBase对象正是用来描述系统当前的各个应用程序窗口的，SurfaceFlinger类的成员函数lockPageFlip依次调用这些LayerBase对象的成员函数lockPageFlip来让它们设置当前需要渲染的图形缓冲区。由于前面我们假设这些LayerBase对象的实际类型为Layer，因此，前面调用的实际上就是Layer类的成员函数lockPageFlip。接下来，我们就继续分析Layer类的成员函数lockPageFlip的实现。

        Step 3. Layer.lockPageFlip

        这个函数定义在文件frameworks/base/services/surfaceflinger/Layer.cpp中。在前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文中，我们已经分析过这个函数的实现了，不过当时分析得不是太完整，现在我们再重新来阅读一下。这个函数比较长，我们分段来阅读：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void Layer::lockPageFlip(**bool**& recomputeVisibleRegions)
2. {
3. ClientRef::Access sharedClient(mUserClientRef);
4. SharedBufferServer\* lcblk(sharedClient.get());
5. if (!lcblk) {
6. // client died
7. recomputeVisibleRegions = true;
8. return;
9. }
11. ssize\_t buf = lcblk->retireAndLock();
12. if (buf == NOT\_ENOUGH\_DATA) {
13. // NOTE: This is not an error, it simply means there is nothing to
14. // retire. The buffer is locked because we will use it
15. // for composition later in the loop
16. return;
17. }
19. if (buf < NO\_ERROR) {
20. LOGE("retireAndLock() buffer index (%d) out of range", **int**(buf));
21. mPostedDirtyRegion.clear();
22. return;
23. }
25. // we retired a buffer, which becomes the new front buffer
26. if (mBufferManager.setActiveBufferIndex(buf) < NO\_ERROR) {
27. LOGE("retireAndLock() buffer index (%d) out of range", **int**(buf));
28. mPostedDirtyRegion.clear();
29. return;
30. }

void Layer::lockPageFlip(bool& recomputeVisibleRegions)

{

ClientRef::Access sharedClient(mUserClientRef);

SharedBufferServer\* lcblk(sharedClient.get());

if (!lcblk) {

// client died

recomputeVisibleRegions = true;

return;

}

ssize\_t buf = lcblk->retireAndLock();

if (buf == NOT\_ENOUGH\_DATA) {

// NOTE: This is not an error, it simply means there is nothing to

// retire. The buffer is locked because we will use it

// for composition later in the loop

return;

}

if (buf < NO\_ERROR) {

LOGE("retireAndLock() buffer index (%d) out of range", int(buf));

mPostedDirtyRegion.clear();

return;

}

// we retired a buffer, which becomes the new front buffer

if (mBufferManager.setActiveBufferIndex(buf) < NO\_ERROR) {

LOGE("retireAndLock() buffer index (%d) out of range", int(buf));

mPostedDirtyRegion.clear();

return;

}

       这段代码首先通过Layer类的成员变量mUserClientRef来获得一个SharedBufferServer对象lcblk。这个SharedBufferServer对象lcblk是用来描述Android应用程序与SurfaceFlinger服务之间的共享UI元数据，通过调用它的成员函数retireAndLock，Layer类的成员函数lockPageFlip就可以知道当前需要渲染的图形缓冲区的编号。有了这个编号之后，Layer类的成员函数lockPageFlip就可以在成员变量mBufferManager所描述的一个BufferManager中找到一个对应的图形缓冲区。

       这段代码接着将当前需要渲染的图形缓冲区的编号保存在Layer类的成员变量mBufferManager所描述的一个BufferManager中，这是通过调用这个BufferManager的成员函数setActiveBufferIndex来实现的。在接下来的Step 4中，我们就可以通过这个BufferManager的成员函数getActiveBufferIndex来重新获得这个编号，以便可以找到对应的图形缓冲区。

       我们接着往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. sp<GraphicBuffer> newFrontBuffer(getBuffer(buf));
2. if (newFrontBuffer != NULL) {
3. // get the dirty region
4. // compute the posted region
5. const Region dirty(lcblk->getDirtyRegion(buf));
6. mPostedDirtyRegion = dirty.intersect( newFrontBuffer->getBounds() );
8. // update the layer size and release freeze-lock
9. const Layer::State& front(drawingState());
10. if (newFrontBuffer->getWidth()  == front.requested\_w &&
11. newFrontBuffer->getHeight() == front.requested\_h)
12. {
13. if ((front.w != front.requested\_w) ||
14. (front.h != front.requested\_h))
15. {
16. // Here we pretend the transaction happened by updating the
17. // current and drawing states. Drawing state is only accessed
18. // in this thread, no need to have it locked
19. Layer::State& editDraw(mDrawingState);
20. editDraw.w = editDraw.requested\_w;
21. editDraw.h = editDraw.requested\_h;
23. // We also need to update the current state so that we don't
24. // end-up doing too much work during the next transaction.
25. // NOTE: We actually don't need hold the transaction lock here
26. // because State::w and State::h are only accessed from
27. // this thread
28. Layer::State& editTemp(currentState());
29. editTemp.w = editDraw.w;
30. editTemp.h = editDraw.h;
32. // recompute visible region
33. recomputeVisibleRegions = true;
34. }
36. // we now have the correct size, unfreeze the screen
37. mFreezeLock.clear();
38. }
40. // get the crop region
41. setBufferCrop( lcblk->getCrop(buf) );
43. // get the transformation
44. setBufferTransform( lcblk->getTransform(buf) );
46. } else {
47. // this should not happen unless we ran out of memory while
48. // allocating the buffer. we're hoping that things will get back
49. // to normal the next time the app tries to draw into this buffer.
50. // meanwhile, pretend the screen didn't update.
51. mPostedDirtyRegion.clear();
52. }

sp<GraphicBuffer> newFrontBuffer(getBuffer(buf));

if (newFrontBuffer != NULL) {

// get the dirty region

// compute the posted region

const Region dirty(lcblk->getDirtyRegion(buf));

mPostedDirtyRegion = dirty.intersect( newFrontBuffer->getBounds() );

// update the layer size and release freeze-lock

const Layer::State& front(drawingState());

if (newFrontBuffer->getWidth() == front.requested\_w &&

newFrontBuffer->getHeight() == front.requested\_h)

{

if ((front.w != front.requested\_w) ||

(front.h != front.requested\_h))

{

// Here we pretend the transaction happened by updating the

// current and drawing states. Drawing state is only accessed

// in this thread, no need to have it locked

Layer::State& editDraw(mDrawingState);

editDraw.w = editDraw.requested\_w;

editDraw.h = editDraw.requested\_h;

// We also need to update the current state so that we don't

// end-up doing too much work during the next transaction.

// NOTE: We actually don't need hold the transaction lock here

// because State::w and State::h are only accessed from

// this thread

Layer::State& editTemp(currentState());

editTemp.w = editDraw.w;

editTemp.h = editDraw.h;

// recompute visible region

recomputeVisibleRegions = true;

}

// we now have the correct size, unfreeze the screen

mFreezeLock.clear();

}

// get the crop region

setBufferCrop( lcblk->getCrop(buf) );

// get the transformation

setBufferTransform( lcblk->getTransform(buf) );

} else {

// this should not happen unless we ran out of memory while

// allocating the buffer. we're hoping that things will get back

// to normal the next time the app tries to draw into this buffer.

// meanwhile, pretend the screen didn't update.

mPostedDirtyRegion.clear();

}

        这段代码首先是用来设置当前正在处理的应用程序窗口的脏区域、纹理坐标和旋转方向的。

        一个应用程序窗口当前的脏区域、纹理坐标，以及旋转方向是保存在一块匿名共享内存中的，用来在Android应用程序和SurfaceFlinger服务之间共享UI元数据。这块匿名共享内存就是使用前面所获得的SharedBufferServer对象lcblk来描述的，即我们可以通过调用SharedBufferServer对象lcblk的成员函数getDirtyRegion、getCrop以及getTransform来获得当前正在处理的应用程序窗口的脏区域、纹理坐标以及旋转方向。其中，脏区域保存在Layer类的成员变量mPostedDirtyRegion中，纹理坐标通过调用父类LayerBase的成员函数setBufferCrop保存在其成员变量mBufferCrop中，而旋转方向通过调用父类LayerBase的成员函数setBufferTransform保存在其成员变量mBufferTransform中。

        从前面的执行过程可以知道，变量buf保存的是当前需要渲染的图形缓冲区的编号。当我们以这个编号为参数来调用Layer类的成员函数getBuffer时，就可以得到一个对应的图形缓冲区。这个图形缓冲区使用一个GraphicBuffer对象来描述，并且保存在变量newFrontBuffer中。

        当前需要渲染的图形缓冲区newFrontBuffer的大小可以通过调用它的成员函数getBounds来获得。我们把当前需要渲染的图形缓冲区的大小与当前正在处理的应用程序窗口的脏区域执行一个相交操作之后，才可以得到最终需要渲染的脏区域。这个最终需要渲染的脏区域就保存在前面提到的Layer类的成员变量mPostedDirtyRegion中。

        这段代码接下来还会检查是否需要修改当前正在处理的应用程序窗口的大小。本来修改应用程序窗口的大小是在处理应用程序窗口的属性变化时进行的，如前面第1部分的内容所示。但是在前面第1部分的内容的Step 3中提到，有一种特殊情况，即Android应用程序在请求SurfaceFlinger服务分配图形缓冲区时，传递过来的图形缓冲区的宽度和高度值等于0。在这种情况下，Layer类的成员函数doTransaction会暂时忽略掉应用程序窗口当前所发生的大小变化，而是等到该应用程序窗口大小发生了变化后所申请分配的新图形缓冲区需要渲染时，再来修改它的大小。

        从前面第1部分的内容的Step 3可以知道，调用Layer类的成员函数drawingState所获得的一个State对象是用来描述一个应用程序窗口的当前渲染状态的，而这个State对象的成员变量requested\_w和requested\_h描述的就正好是该应用程序窗口的最新宽度和高度值。如果一个应用程序窗口的最新宽度和高度值正好等于它当前所要渲染的图形缓冲区newFrontBuffer的宽度和高度值，那么就说明它当前所要渲染的图形缓冲区newFrontBuffer就正好是在它的大小发生变化之后所申请分配的，因此，这时候Layer类的成员函数lockPageFlip就需要继续检查它之前是否有被忽略掉的大小变化未被处理。如果有的话，现在就是时候对它的大小变化进行处理了。

       从前面第1部分的内容的Step 4可以知道，如果用来描述一个应用程序窗口的当前渲染状态的一个State对象的两个成员变量w和h，与另外两个成员变量requested\_w和requested\_h不相等的话，那么就说明该应用程序窗口有被忽略掉的大小变化未被处理，因此，在这种情况下，Layer类的成员函数lockPageFlip就会分别将该State对象的成员变量requested\_w和requested\_h的值分别保存在成员变量w和h中，同时还会保存用来描述该应用程序窗口的下一次渲染状态的一个State对象的成员变量w和h中，以表示前面被忽略掉的窗口大小变化已经得到处理了。

       由于每当有应用程序窗口的大小发生变化之后，SurfaceFlinger服务都需要重新计算各个应用程序窗口的可见区域，因此，Layer类的成员函数lockPageFlip在处理了当前正在处理的应用程序窗口上一次被忽略掉的大小变化之后，需要将输出参数recomputeVisibleRegions的值设置为true，以便可以用来通知SurfaceFlinger服务更新各个应用程序窗口的可见区域，如前面的Step 1所示。

       此外，Layer类的成员函数lockPageFlip在处理了当前正在处理的应用程序窗口上一次被忽略掉的大小变化之后，如果之前获取过用来冻结系统显示屏的锁，那么现在也是时间释放这个锁了，以避免阻止SurfaceFlinger服务接下来重新渲染系统的UI。这个用来冻结系统显示屏的锁是保存在Layer类的成员变量mFreezeLock中的，通过调用它的成员函数clear就可以释放它所描述的用来冻结系统显示屏的锁。

       我们接着往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. if (lcblk->getQueuedCount()) {
2. // signal an event if we have more buffers waiting
3. mFlinger->signalEvent();
4. }
6. /\* a buffer was posted, so we need to call reloadTexture(), which
7. \* will update our internal data structures (eg: EGLImageKHR or
8. \* texture names). we need to do this even if mPostedDirtyRegion is
9. \* empty -- it's orthogonal to the fact that a new buffer was posted,
10. \* for instance, a degenerate case could be that the user did an empty
11. \* update but repainted the buffer with appropriate content (after a
12. \* resize for instance).
13. \*/
14. reloadTexture( mPostedDirtyRegion );

if (lcblk->getQueuedCount()) {

// signal an event if we have more buffers waiting

mFlinger->signalEvent();

}

/\* a buffer was posted, so we need to call reloadTexture(), which

\* will update our internal data structures (eg: EGLImageKHR or

\* texture names). we need to do this even if mPostedDirtyRegion is

\* empty -- it's orthogonal to the fact that a new buffer was posted,

\* for instance, a degenerate case could be that the user did an empty

\* update but repainted the buffer with appropriate content (after a

\* resize for instance).

\*/

reloadTexture( mPostedDirtyRegion );

}

        这段代码首先检查当前正在处理的应用程序窗口是否还有其它图形缓冲区在等待被渲染。如果有的话，那么就会通过Layer类的成员变量mFlinger的成员函数signalEvent来向SurfaceFlinger服务的UI渲染线程的消息队列发送另外一个类型为MessageQueue::INVALIDATE的消息，以便SurfaceFlinger服务的UI渲染线程在刷新了当前的系统UI之后，可以马上再次刷新系统的UI。

        执行完成以上操作之后，Layer类的成员函数lockPageFlip就调用另外一个成员函数reloadTexture来为当前正在处理的应用程序窗口加载新的纹理，以便接下来可以将这个新的纹理渲染合成和渲染到硬件帧缓冲区中去。

        接下来，我们就继续分析Layer类的成员函数reloadTexture是如何获得应用程序窗口即将要合成和渲染的纹理的。

        Step 4. Layer.reloadTexture

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void Layer::reloadTexture(const Region& dirty)
2. {
3. sp<GraphicBuffer> buffer(mBufferManager.getActiveBuffer());
4. if (buffer == NULL) {
5. // this situation can happen if we ran out of memory for instance.
6. // not much we can do. continue to use whatever texture was bound
7. // to this context.
8. return;
9. }
11. if (mGLExtensions.haveDirectTexture()) {
12. EGLDisplay dpy(mFlinger->graphicPlane(0).getEGLDisplay());
13. if (mBufferManager.initEglImage(dpy, buffer) != NO\_ERROR) {
14. // not sure what we can do here...
15. goto slowpath;
16. }
17. } else {
18. slowpath:
19. GGLSurface t;
20. if (buffer->usage & GRALLOC\_USAGE\_SW\_READ\_MASK) {
21. status\_t res = buffer->lock(&t, GRALLOC\_USAGE\_SW\_READ\_OFTEN);
22. LOGE\_IF(res, "error %d (%s) locking buffer %p",
23. res, strerror(res), buffer.get());
24. if (res == NO\_ERROR) {
25. mBufferManager.loadTexture(dirty, t);
26. buffer->unlock();
27. }
28. } else {
29. // we can't do anything
30. }
31. }
32. }

void Layer::reloadTexture(const Region& dirty)

{

sp<GraphicBuffer> buffer(mBufferManager.getActiveBuffer());

if (buffer == NULL) {

// this situation can happen if we ran out of memory for instance.

// not much we can do. continue to use whatever texture was bound

// to this context.

return;

}

if (mGLExtensions.haveDirectTexture()) {

EGLDisplay dpy(mFlinger->graphicPlane(0).getEGLDisplay());

if (mBufferManager.initEglImage(dpy, buffer) != NO\_ERROR) {

// not sure what we can do here...

goto slowpath;

}

} else {

slowpath:

GGLSurface t;

if (buffer->usage & GRALLOC\_USAGE\_SW\_READ\_MASK) {

status\_t res = buffer->lock(&t, GRALLOC\_USAGE\_SW\_READ\_OFTEN);

LOGE\_IF(res, "error %d (%s) locking buffer %p",

res, strerror(res), buffer.get());

if (res == NO\_ERROR) {

mBufferManager.loadTexture(dirty, t);

buffer->unlock();

}

} else {

// we can't do anything

}

}

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/Layer.cpp中。

        函数首先调用Layer类的成员变量mBufferManager所描述的一个BufferManager对象的成员函数getActiveBuffer来获得当前正在处理的应用程序窗口接下来需要渲染的图形缓冲区，并且保存在变量buffer中。

        函数接着调用Layer类的成员变量mGLExtensions所描述的一个GLExtensions对象的成员函数haveDirectTexture来判断系统是否支持在硬件上直接创建纹理对象。如果支持的话，那么就调用Layer类的成员变量mBufferManager所描述的一个BufferManager对象的成员函数initEglImage来根据图形缓冲区buffer的内容在硬件上直接创建一个纹理对象。如果这个纹理对象创建成功的话，那么就会保存在Layer类的成员变量mBufferData所描述的一个BufferData中。

        如果系统不支持在硬件上直接创建纹理对象，或者前面在硬件上直接创建纹理对象失败，那么函数就会退而求之，转换为用软件的方式来根据图形缓冲区buffer的内容来创建一个纹理对象，这是通过调用Layer类的成员变量mBufferManager所描述的一个BufferManager对象的成员函数loadTexture来实现的。创建出来的这个纹理对象就保存在Layer类的成员变量mFailoverTexture中。不过，能够使用软件的方式来创建纹理对象的一个前提是图形缓冲区buffer的内容可以通过软件的方式来读取，即它的成员变量usage的GRALLOC\_USAGE\_SW\_READ\_MASK位等于1。在这种情况下，我们就可以调用图形缓冲区buffer的成员函数lock来获得它的内容访问地址，并且保存在GGLSurface对象t，以便接下来可以通过这个GGLSurface对象t来创建一个纹理对象。

        Layer类的成员函数reloadTexture首选在硬件上创建纹理对象，如果这种方式不可行，那么再使用软件方式来创建纹理对象。这种技术在计算机领域中就称为失效备援技术，即Failover技术。

        这一步执行完成之后，返回到前面的Step 1中，即SurfaceFlinger类的成员函数handlePageFlip中，接下来就会继续调用SurfaceFlinger类的另外一个成员函数computeVisibleRegions来计算各个应用程序窗口的可见区域。

        Step 5. SurfaceFlinger.computeVisibleRegions

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中，主要用来计算各个应用程序窗口的可见区域。在分析这个函数的实现之前，我们首先解释一些与应用程序窗口相关的概念：可见区域（Visible Region）、透明区域（Transparent Region）、半透明区域（Translucent Region）、完全不透明区域（Opaque Region）和被覆盖区域（Covered Region）。

        假设一个应用程序窗口的宽度和高度分别为w和h，如图4所示：

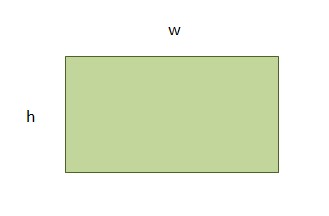


图4  应用程序窗口的可见区域

       那么我们就可以将由(0,0)、(0, w)、(0, h)和(w,h)四个点组成的区域称为应用程序窗口的可见区域。

       接下来，我们可以在一个应用程序窗口的可见区域挖一个洞出来，如图5所示：

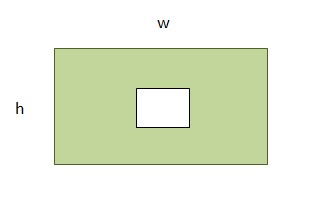


图5 应用程序窗口的透明区域

      这时候应用程序窗口真正的可见区域就需要减去中间被挖出来的洞。这个被挖出来的洞就称为应用程序窗口的透明可见区域。

      如果应用程序窗口的可见区域的Alpha通道大于0并且小255，那么我们就认为应用程序窗口的可见区域是半透明的。有两种极端情况，即当应用程序窗口的可见区域的Alpha通道等于0或者255的时候。当等于0的时候，我们就认为应用程序窗口的可见区域是透明的，就如图5所示的洞一样，而当等于255的时候，我们就认为应用程序窗口的可见区域是完全不透明的。

      上面我们讨论的应用程序窗口的可见区域是基于单个应用程序窗口而言的，当多个应用程序窗口叠加在一起的时候，在讨论一个应用程序窗口的可见区域的时候，就需要考虑位于它上面的其它应用程序窗口的可见区域的影响了。注意，一个应用程序窗口的可见区域只受位于它上面的其它应用程序窗口影响，而不会受到位于它下面的其它的应用程序窗口影响，因此，我们是按照从上到下的顺序来计算系统中各个应用程序窗口的可见区域的。

       为了方便描述，我们假设位于一个应用程序窗口上面的所有应用程序窗口组成了一个整体的可见区域（Above Covered Layers），并且这个可见区域与我们所要讨论的应用程序窗口相交，即它们叠加在一起，如图6所示：

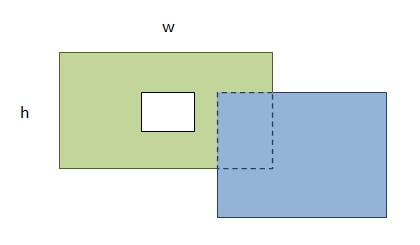


图6 应用程序窗口的被覆盖区域

       由蓝色矩形组成的区域即为上层所有应用程序窗口所组成的一个整体可见区域，这个整体可见区域与下面绿色矩形组成的一个应用程序窗口相交的部分，即由虚线所围成的区域，就是下面的一个应用程序窗口的被覆盖区域。

       一个应用程序窗口的被覆盖区域有可能是半透明的，也有可能是完全不透明的，但是不可能是透明的，如图7所示：

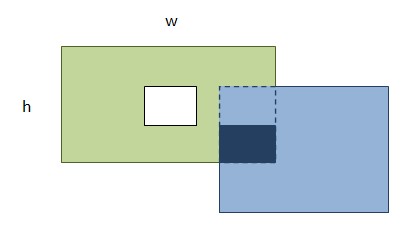


图7 应用程序窗口的被覆盖完全不透明区域

       在原来由虚线围成的区域中，深蓝色的那部分区域就是完全不透明的（Above Opaque  Layers），这时候由绿色矩形组成的应用程序窗口的可见区域除了要减去中间的洞（透明区域）之外，还要减去被覆盖的完全不透明区域，如下图8所示：

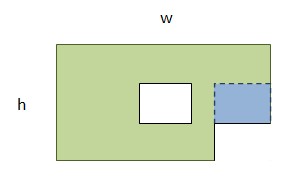


图8 应用程序窗口的最终可见区域

      从上面的讨论我们就可以清楚地知道，为了计算一个应用程序窗口的最终可见区域，我们需要知道：

      1. 应用程序窗口的左上角位置，以及宽度和高度，以便可以获得应用程序窗口的原始可见区域。

      2. 应用程序窗口的透明区域。

      3. 应用程序窗口的被覆盖完全不透明区域。

      用第1步到的原始可见区域减去第2步的透明区域和第3步的被覆盖完全不透明区域，就可以得到一个应用程序窗口的最终可见区域。

      为了获得第3步的被覆盖完全不透明区域，我们在计算一个应用程序窗口的最终可见区域的过程中，还需要将此前得到的应用程序窗口的完全不透明区域组合起来，形成一个覆盖完全不透明区域（Above Opaque  Layers），因此，我们还需要知道：

      4. 应用程序窗口的完全不透明区域。

      此外，由于一个应用程序窗口的被覆盖半透明区域是需要与上层的应用程序窗口可见区域执行混合计算的，因此，我们在计算系统中各个应用程序窗口的可见区域的过程中，还需要将所有上层的应用程序窗口可见区域组合起来形成一个覆盖区域（Above Covered Layers）。

      有了这些背景知识之后，接下来我们就可以分析SurfaceFlinger类的成员函数computeVisibleRegions的实现了。由于SurfaceFlinger类的成员函数computeVisibleRegions的实现比较长，我们分段来阅读：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::computeVisibleRegions(
2. LayerVector& currentLayers, Region& dirtyRegion, Region& opaqueRegion)
3. {
4. const GraphicPlane& plane(graphicPlane(0));
5. const Transform& planeTransform(plane.transform());
6. const DisplayHardware& hw(plane.displayHardware());
7. const Region screenRegion(hw.bounds());
9. Region aboveOpaqueLayers;
10. Region aboveCoveredLayers;
11. Region dirty;
13. **bool** secureFrameBuffer = false;

void SurfaceFlinger::computeVisibleRegions(

LayerVector& currentLayers, Region& dirtyRegion, Region& opaqueRegion)

{

const GraphicPlane& plane(graphicPlane(0));

const Transform& planeTransform(plane.transform());

const DisplayHardware& hw(plane.displayHardware());

const Region screenRegion(hw.bounds());

Region aboveOpaqueLayers;

Region aboveCoveredLayers;

Region dirty;

bool secureFrameBuffer = false;

        这段代码首先计算得到屏幕区域，保存在变量screenRegion中，接着定义了另外两个区域aboveOpaqueLayers，分别用来描述上层覆盖完全不透明区域（Above Opaque  Layers）和上层覆盖区域（Above Covered Layers），最后定义了一个布尔变量secureFrameBuffer，用来描述系统中是否存在界面受到安全保护的应用程序窗口。

       界面受到安全保护的应用程序窗口的内容是不可以在进程间传输的，这个属性主要是应用在屏幕截图中。例如，如果系统中存在一个界面受到安全保护的应用程序窗口，那么我们就不可以请求SurfaceFlinger服务执行截屏功能，因为SurfaceFlinger服务截取下来的屏幕会被传输给请求的进程使用。

      我们接着往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. **size\_t** i = currentLayers.size();
2. while (i--) {
3. const sp<LayerBase>& layer = currentLayers[i];
4. layer->validateVisibility(planeTransform);

size\_t i = currentLayers.size();

while (i--) {

const sp<LayerBase>& layer = currentLayers[i];

layer->validateVisibility(planeTransform);

       这段代码是一个while循环的前面几行。系统中所有需要计算可见区域的应用程序窗口都保存在参数currentLayers所描述的一个向量中。这段代码的while循环就是用来逐个地这些应用程序窗口的可见区域的。注意，这个while是先计算是按照从上到下的顺序来计算系统中各个应用程序窗口的可见区域的。

       在计算一个应用程序窗口layer的可见区域之前，我们首先要验证它的可见性，这是通过调用它的成员函数validateVisibility来实现的，即调用LayerBase类的成员函数validateVisibility来实现的。

       LayerBase类的成员函数validateVisibility的实现如下所示：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void LayerBase::validateVisibility(const Transform& planeTransform)
2. {
3. const Layer::State& s(drawingState());
4. const Transform tr(planeTransform \* s.transform);
5. const **bool** transformed = tr.transformed();
7. uint32\_t w = s.w;
8. uint32\_t h = s.h;
9. tr.transform(mVertices[0], 0, 0);
10. tr.transform(mVertices[1], 0, h);
11. tr.transform(mVertices[2], w, h);
12. tr.transform(mVertices[3], w, 0);
13. if (UNLIKELY(transformed)) {
14. // NOTE: here we could also punt if we have too many rectangles
15. // in the transparent region
16. if (tr.preserveRects()) {
17. // transform the transparent region
18. transparentRegionScreen = tr.transform(s.transparentRegion);
19. } else {
20. // transformation too complex, can't do the transparent region
21. // optimization.
22. transparentRegionScreen.clear();
23. }
24. } else {
25. transparentRegionScreen = s.transparentRegion;
26. }
28. // cache a few things...
29. mOrientation = tr.getOrientation();
30. mTransformedBounds = tr.makeBounds(w, h);
31. mLeft = tr.tx();
32. mTop  = tr.ty();
33. }

void LayerBase::validateVisibility(const Transform& planeTransform)

{

const Layer::State& s(drawingState());

const Transform tr(planeTransform \* s.transform);

const bool transformed = tr.transformed();

uint32\_t w = s.w;

uint32\_t h = s.h;

tr.transform(mVertices[0], 0, 0);

tr.transform(mVertices[1], 0, h);

tr.transform(mVertices[2], w, h);

tr.transform(mVertices[3], w, 0);

if (UNLIKELY(transformed)) {

// NOTE: here we could also punt if we have too many rectangles

// in the transparent region

if (tr.preserveRects()) {

// transform the transparent region

transparentRegionScreen = tr.transform(s.transparentRegion);

} else {

// transformation too complex, can't do the transparent region

// optimization.

transparentRegionScreen.clear();

}

} else {

transparentRegionScreen = s.transparentRegion;

}

// cache a few things...

mOrientation = tr.getOrientation();

mTransformedBounds = tr.makeBounds(w, h);

mLeft = tr.tx();

mTop = tr.ty();

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/LayerBase.cpp中。

       参数planeTransform用来描述系统显示屏旋转方向，它是一个变换矩阵，而用来描述当前正在处理的应用程序窗口的当前渲染状态的一个State对象s的成员变量transform指向的也是一个变换矩阵，用来描述当前正在处理的应用程序窗口的位置、旋转方向和缩放因子等。将这两者相乘，就可以得到当前正在处理的应用程序窗口相对于系统显示屏的一个变换矩阵tr。

        函数首先计算当前正在处理的应用程序窗口的四个角在显示屏中的位置，并且分别保存在LayerBase的成员变量mVertices所描述的一个数组中，这是通过调用变换矩阵tr的成员函数transform来实现的。

        函数接着判断当前正在处理的应用程序窗口是否被旋转过或者被缩放过。如果是的话，那么前面调用变换矩阵tr的成员函数transformed的返回值就会等于true，即变量transformed的等于true。在这种情况下，函数就要相应地对当前正在处理的应用程序窗口的透明区域进行旋转或者缩放。但是有一种特殊情况，即当当前正在处理的应用程序窗口被旋转和缩放得不规则时，这时候对应用程序窗口的透明区域进行旋转或者缩放就会很复杂，于是函数就干脆将它的透明区域忽略掉。判断当前正在处理的应用程序窗口是否被旋转和缩放得不规则是通过调用变换矩阵tr的成员函数preserveRects来实现的，当它的返回值等于true的时候，就说明当前正在处理的应用程序窗口是否被旋转和缩放之后还是规则，否则就是不规则的。

        当前正在处理的应用程序窗口的透明区域保存在State对象s的成员变量transparentRegion中，按照上述原理，函数按照以下规则来对它进行处理：

        1. 当变量transformed的等于false时，说明当前正在处理的应用程序窗口的透明区域就不需要进行旋转或者缩放，这时候就可以将这个透明区域保存在LayerBase类的成员变量transparentRegionScreen中。

        2. 当变量transformed的等于true，并且变换矩阵tr的成员函数preserveRects的返回值也等于true时，那么就说明当前正在处理的应用程序窗口的透明区域需要进行旋转或者缩放，这时候通过调用变换矩阵tr的成员函数transform来实现的。最终得到的透明区域同样是保存在LayerBase类的成员变量transparentRegionScreen中。

        3.  当变量transformed的等于true，并且变换矩阵tr的成员函数preserveRects的返回值等于false时，那么就说明需要忽略掉当前正在处理的应用程序窗口的透明区域，这是通过LayerBase类的成员变量transparentRegionScreen所描述的一个Region对象的成员函数clear来实现的。

        最后，函数就计算当前正在处理的应用程序窗口的方向，左上角位置，以及包含了当前正在处理的应用程序窗口的一个矩形区域，这些值分别保存在在LayerBase类的成员变量mOrientation，mLeft和mTop，以及mTransformedBounds中。

        返回到SurfaceFlinger类的成员函数computeVisibleRegions中，我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. // start with the whole surface at its current location
2. const Layer::State& s(layer->drawingState());
4. /\*
5. \* opaqueRegion: area of a surface that is fully opaque.
6. \*/
7. Region opaqueRegion;
9. /\*
10. \* visibleRegion: area of a surface that is visible on screen
11. \* and not fully transparent. This is essentially the layer's
12. \* footprint minus the opaque regions above it.
13. \* Areas covered by a translucent surface are considered visible.
14. \*/
15. Region visibleRegion;
17. /\*
18. \* coveredRegion: area of a surface that is covered by all
19. \* visible regions above it (which includes the translucent areas).
20. \*/
21. Region coveredRegion;

// start with the whole surface at its current location

const Layer::State& s(layer->drawingState());

/\*

\* opaqueRegion: area of a surface that is fully opaque.

\*/

Region opaqueRegion;

/\*

\* visibleRegion: area of a surface that is visible on screen

\* and not fully transparent. This is essentially the layer's

\* footprint minus the opaque regions above it.

\* Areas covered by a translucent surface are considered visible.

\*/

Region visibleRegion;

/\*

\* coveredRegion: area of a surface that is covered by all

\* visible regions above it (which includes the translucent areas).

\*/

Region coveredRegion;

        这段代码首先获得用来描述当前正在处理的应用程序窗口的当前渲染状态的一个State对象s，接着再定义了三个Region对象opaqueRegion、visibleRegion和coveredRegion，分别用来描述当前正在处理的应用程序窗口的完全不透明区域、可见区域和被覆盖区域。这三个区域的含义和作用如前所述。

         我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. // handle hidden surfaces by setting the visible region to empty
2. if (LIKELY(!(s.flags & ISurfaceComposer::eLayerHidden) && s.alpha)) {
3. const **bool** translucent = layer->needsBlending();
4. const Rect bounds(layer->visibleBounds());
5. visibleRegion.set(bounds);
6. visibleRegion.andSelf(screenRegion);
7. if (!visibleRegion.isEmpty()) {
8. // Remove the transparent area from the visible region
9. if (translucent) {
10. visibleRegion.subtractSelf(layer->transparentRegionScreen);
11. }
13. // compute the opaque region
14. const int32\_t layerOrientation = layer->getOrientation();
15. if (s.alpha==255 && !translucent &&
16. ((layerOrientation & Transform::ROT\_INVALID) == false)) {
17. // the opaque region is the layer's footprint
18. opaqueRegion = visibleRegion;
19. }
20. }
21. }

// handle hidden surfaces by setting the visible region to empty

if (LIKELY(!(s.flags & ISurfaceComposer::eLayerHidden) && s.alpha)) {

const bool translucent = layer->needsBlending();

const Rect bounds(layer->visibleBounds());

visibleRegion.set(bounds);

visibleRegion.andSelf(screenRegion);

if (!visibleRegion.isEmpty()) {

// Remove the transparent area from the visible region

if (translucent) {

visibleRegion.subtractSelf(layer->transparentRegionScreen);

}

// compute the opaque region

const int32\_t layerOrientation = layer->getOrientation();

if (s.alpha==255 && !translucent &&

((layerOrientation & Transform::ROT\_INVALID) == false)) {

// the opaque region is the layer's footprint

opaqueRegion = visibleRegion;

}

}

}

        这段代码用来计算当前正在处理的应用程序窗口的可见区域和完全不透明区域。只有在当前正在处理的应用程序窗口处于可见状态，并且它不是完全透明时，才需要计算这两个区域。当State对象s的成员变量flags的ISurfaceComposer::eLayerHidden位等于0时，就说明当前正在处理的应用程序窗口是处于可见状态的，而当它的另外一个成员变量alpha的值不等于0的时候，就说明当前正在处理的应用程序窗口不是完全透明的。

        我们首先来看当前正在处理的应用程序窗口的可见区域的计算过程。

        LayerBase对象layer的成员函数visibleBounds返回的是包围当前正在处理的应用程序窗口的一个矩形，即LayerBase类的成员变量mTransformedBounds，这是在前面调用LayerBase类的成员函数validateVisibility时计算得到的。函数首先将这个矩形与屏幕区域screenRegion执行一个与操作，以便可以得到当前正在处理的应用程序窗口的初始可见区域visibleRegion。

        当得到当前正在处理的应用程序窗口的初始可见区域visibleRegion不为空时，函数接着判断该窗口是否是半透明的，即它是否要与其它的窗口执行混合操作。如果是半透明的话，那么前面调用LayerBase对象layer的成员函数needsBlending的返回值translucent就会等于true。在这种情况下，我们就需要将应用程序窗口的初始可见区域visibleRegion减去它的透明区域，即LayerBase对象layer的成员变量transparentRegionScreen所描述的区域。这样我们进一步得到当前正在处理的应用程序窗口的可见区域visibleRegion。

        我们接着来看当前正在处理的应用程序窗口的完全不透明区域的计算过程。

        当当前正在处理的应用程序窗口是完全不透明，并且旋转方向也是规则时，那么它的完全不透明区域opaqueRegion就等于前面计算所得到的可见区域visibleRegion。当当前正在处理的应用程序窗口的Alpha通道等于255，即当State对象s的成员变量alpha的值等于255，并且变量translucent的值等于false时，就说明它是完全不透明的，而当当前正在处理的应用程序窗口的旋转方向layerOrientation的Transform::ROT\_INVALID位等于0的时候，就说明它的旋转方向是规则的。

        我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. // Clip the covered region to the visible region
2. coveredRegion = aboveCoveredLayers.intersect(visibleRegion);
4. // Update aboveCoveredLayers for next (lower) layer
5. aboveCoveredLayers.orSelf(visibleRegion);
7. // subtract the opaque region covered by the layers above us
8. visibleRegion.subtractSelf(aboveOpaqueLayers);

// Clip the covered region to the visible region

coveredRegion = aboveCoveredLayers.intersect(visibleRegion);

// Update aboveCoveredLayers for next (lower) layer

aboveCoveredLayers.orSelf(visibleRegion);

// subtract the opaque region covered by the layers above us

visibleRegion.subtractSelf(aboveOpaqueLayers);

        这段代码用来计算当前正在处理的应用程序窗口的被覆盖区域，以及再进一步计算它的可见区域，主要考虑是否被上层的不透明区域覆盖了。

        变量aboveCoveredLayers用来描述当前正在处理的应用程序窗口的所有上层应用程序窗口所组成的可见区域，将这个区域与当前正在处理的应用程序窗口的可见区域visibleRegion相交，就可以得到当前正在处理的应用程序窗口的被覆盖区域coveredRegion，而将这个区域与当前正在处理的应用程序窗口的可见区域visibleRegion相或一下，就可以得到下一个应用程序窗口的所有上层应用程序窗口所组成的可见区域aboveCoveredLayers。

        变量aboveOpaqueLayers用来描述当前正在处理的应用程序窗口的所有上层应用程序窗口所组成的完全不透明区域，这个区域从当前正在处理的应用程序窗口的可见区域visibleRegion减去后，就可以得到当前正在处理的应用程序窗口的最终可见区域visibleRegion。

        我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. // compute this layer's dirty region
2. if (layer->contentDirty) {
3. // we need to invalidate the whole region
4. dirty = visibleRegion;
5. // as well, as the old visible region
6. dirty.orSelf(layer->visibleRegionScreen);
7. layer->contentDirty = false;
8. } else {
9. /\* compute the exposed region:
10. \*   the exposed region consists of two components:
11. \*   1) what's VISIBLE now and was COVERED before
12. \*   2) what's EXPOSED now less what was EXPOSED before
13. \*
14. \* note that (1) is conservative, we start with the whole
15. \* visible region but only keep what used to be covered by
16. \* something -- which mean it may have been exposed.
17. \*
18. \* (2) handles areas that were not covered by anything but got
19. \* exposed because of a resize.
20. \*/
21. const Region newExposed = visibleRegion - coveredRegion;
22. const Region oldVisibleRegion = layer->visibleRegionScreen;
23. const Region oldCoveredRegion = layer->coveredRegionScreen;
24. const Region oldExposed = oldVisibleRegion - oldCoveredRegion;
25. dirty = (visibleRegion&oldCoveredRegion) | (newExposed-oldExposed);
26. }
27. dirty.subtractSelf(aboveOpaqueLayers);
29. // accumulate to the screen dirty region
30. dirtyRegion.orSelf(dirty);

// compute this layer's dirty region

if (layer->contentDirty) {

// we need to invalidate the whole region

dirty = visibleRegion;

// as well, as the old visible region

dirty.orSelf(layer->visibleRegionScreen);

layer->contentDirty = false;

} else {

/\* compute the exposed region:

\* the exposed region consists of two components:

\* 1) what's VISIBLE now and was COVERED before

\* 2) what's EXPOSED now less what was EXPOSED before

\*

\* note that (1) is conservative, we start with the whole

\* visible region but only keep what used to be covered by

\* something -- which mean it may have been exposed.

\*

\* (2) handles areas that were not covered by anything but got

\* exposed because of a resize.

\*/

const Region newExposed = visibleRegion - coveredRegion;

const Region oldVisibleRegion = layer->visibleRegionScreen;

const Region oldCoveredRegion = layer->coveredRegionScreen;

const Region oldExposed = oldVisibleRegion - oldCoveredRegion;

dirty = (visibleRegion&oldCoveredRegion) | (newExposed-oldExposed);

}

dirty.subtractSelf(aboveOpaqueLayers);

// accumulate to the screen dirty region

dirtyRegion.orSelf(dirty);

        这段代码用来计算屏幕的脏区域。我们首先解释一下屏幕的脏区域是如何计算的。将所有应用程序窗口的脏区域都组合起来，就可以得到屏幕的脏区域，这个脏区域就是需要重新执行渲染操作的。因此，为了得到屏幕的脏区域，我们要知道当前正在处理的应用程序窗口的脏区域，以及之前已经处理了的应用程序窗口脏区域组合。前者使用变量dirty来描述，而后者使用输出参数dirtyRegion来描述。从前面的调用过程可以知道，输出参数dirtyRegion指向的就正好是SurfaceFlinger类的成员变量mDirtyRegion，因此，当这一步执行完成之后，SurfaceFlinger类的成员变量mDirtyRegion就代表了SurfaceFlinger服务所要渲染的脏区域。

        我们首先来看当前正在处理的应用程序窗口的脏区域dirty是如何计算的。我们分两种情况来考虑。

        首先考虑当前正在处理的应用程序窗口上一次的状态还未来得及处理的情况，即它当前的内容是脏的。在这种情况下，LayerBase对象layer的成员变量contentDirty的值就会等于true。这时候我们就需要将该应用程序窗口的上一次可见区域，以及当前的可见区域合并起来，形成一个大的脏区域，这样就可以将两次渲染操作合并成一次来执行。当前正在处理的应用程序窗口的上一次可见区域保存在LayerBase对象layer的成员变量visibleRegionScreen中，而它前的可见区域保存在变量visibleRegion中。将这两者相或一下，就可以得到当前正在处理的应用程序窗口的脏区域dirty。

        接着考虑当前正在处理的应用程序窗口上一次的状态已经处理了的情况，即它当前的内容不是脏的，这意味着它所要显示的内容没有发生变化。在这种情况下，就不需要重新渲染所有的可见区域。那么那些区域是需要重新渲染的呢？第一部分区域是之前是被覆盖的，现在不被覆盖了，第二部分是由于窗口大小变化而引发的新增不被覆盖区域。接下来，我们就来看看这两部分区域是如何计算的。

        将一个应用程序窗口的当前可见区域减去被覆盖区域，就可以它的当前不被覆盖的区域newExposed，按照同样的方法，我们可以也可以得到它的上一次不被覆盖的区域oldExposed。注意，一个应用程序窗口的上一次可见区域和被覆盖区域分别保存与它相对应的一个LayerBase对象的成员变量visibleRegionScreen和coveredRegionScreen中。这样，将一个应用程序窗口的当前不被覆盖的区域newExposed减去它的上一次不被覆盖的区域oldExposed，就可以得到新增的不被覆盖区域，即可以得到第二部分需要重新渲染的区域。另一方面，将一个应用程序窗口的当前可见区域visibleRegion与它的上一次被覆盖区域oldCoveredRegion相交，就可以得到之前是被覆盖的而现在不被覆盖了的区域，即可以得到第一部分需要重新渲染的区域。将第一部分和第二部分需要重新渲染的区域组合起来，就可以得到当前正在处理的应用程序窗口的脏区域dirty。

       得到了当前正在处理的应用程序窗口的脏区域dirty，接下来的事情就好办了。首先从该脏区域dirty减去上层的完全不透明区域，因为后者的渲染不需要当前应用程序窗口来参与，接着最将得到的新的脏区域dirty累加到输出参数dirtyRegion中去，这样就可以得到目前为止，SurfaceFlinger服务需要渲染的脏区域。

      我们接着往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. // Update aboveOpaqueLayers for next (lower) layer
2. aboveOpaqueLayers.orSelf(opaqueRegion);
4. // Store the visible region is screen space
5. layer->setVisibleRegion(visibleRegion);
6. layer->setCoveredRegion(coveredRegion);
8. // If a secure layer is partially visible, lock-down the screen!
9. if (layer->isSecure() && !visibleRegion.isEmpty()) {
10. secureFrameBuffer = true;
11. }

// Update aboveOpaqueLayers for next (lower) layer

aboveOpaqueLayers.orSelf(opaqueRegion);

// Store the visible region is screen space

layer->setVisibleRegion(visibleRegion);

layer->setCoveredRegion(coveredRegion);

// If a secure layer is partially visible, lock-down the screen!

if (layer->isSecure() && !visibleRegion.isEmpty()) {

secureFrameBuffer = true;

}

}

        这段代码是前面的while循环的几行结束代码，主要用来做三件事情。

        第一件事情是计算到目前为止所得到的上层应用程序窗口的完全不透明区域，这是通过组合当前正在处理的应用程序窗口的完全不透明区域与位于它上面的的所有应用程序窗口的完全不透明区域aboveOpaqueLayers来得到的，并且最终结果保存在变量aboveOpaqueLayers中。

       第二件事情是调用LayerBase对象layer的成员函数setVisibleRegion和setCoveredRegion来保存当前正在处理的应用程序窗口的可见区域和被覆盖区域。

       第三件事情是判断当前正在处理的应用程序窗口的内容是否受安全保护的。如果是的话，并且它的可见区域不为空，那么就需要将变量secureFrameBuffer的值设置为true，以表示当前SurfaceFlinger服务不可以执行截屏功能。

       我们接着往下阅读最后一段代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. // invalidate the areas where a layer was removed
2. dirtyRegion.orSelf(mDirtyRegionRemovedLayer);
3. mDirtyRegionRemovedLayer.clear();
5. mSecureFrameBuffer = secureFrameBuffer;
6. opaqueRegion = aboveOpaqueLayers;

// invalidate the areas where a layer was removed

dirtyRegion.orSelf(mDirtyRegionRemovedLayer);

mDirtyRegionRemovedLayer.clear();

mSecureFrameBuffer = secureFrameBuffer;

opaqueRegion = aboveOpaqueLayers;

}

       由于前面我们得到SurfaceFlinger服务需要重新渲染的脏区域dirtyRegion只考虑了那些新增或者本来已经存在的应用程序窗口的，而没有考虑那些已经被删除了的应用程序窗口。那些已经被删除了的应用程序窗口所占据的区域保存在SurfaceFlinger类的成员变量mDirtyRegionRemovedLayer中，因此，将它从输出参数dirtyRegion减去之后得到的才是SurfaceFlinger服务最终需要重新渲染的脏区域。

       此外，函数还将变量secureFrameBuffer的值保存在urfaceFlinger类的成员变量mSecureFrameBuffer中，以便SurfaceFlinger服务可以知道自己可以执行截屏功能。

       最后，函数还将前面所有的应用程序窗口组成的完全不透明区域aboveOpaqueLayers保存在输出参数opaqueRegion，以便可以返回给调用者使用。

       这一步执行完成之后，返回到前面的Step 1中，即SurfaceFlinger类的成员函数handlePageFlip中，接下来就会继续调用SurfaceFlinger类的另外一个成员函数unlockPageFlip来让各个应用程序窗口执行一些清理工作。

       接下来，我们就继续分析SurfaceFlinger类的成员函数unlockPageFlip的实现。

       Step 6. SurfaceFlinger.unlockPageFlip

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::unlockPageFlip(const LayerVector& currentLayers)
2. {
3. const GraphicPlane& plane(graphicPlane(0));
4. const Transform& planeTransform(plane.transform());
5. **size\_t** count = currentLayers.size();
6. sp<LayerBase> const\* layers = currentLayers.array();
7. for (**size\_t** i=0 ; i<count ; i++) {
8. const sp<LayerBase>& layer(layers[i]);
9. layer->unlockPageFlip(planeTransform, mDirtyRegion);
10. }
11. }

void SurfaceFlinger::unlockPageFlip(const LayerVector& currentLayers)

{

const GraphicPlane& plane(graphicPlane(0));

const Transform& planeTransform(plane.transform());

size\_t count = currentLayers.size();

sp<LayerBase> const\* layers = currentLayers.array();

for (size\_t i=0 ; i<count ; i++) {

const sp<LayerBase>& layer(layers[i]);

layer->unlockPageFlip(planeTransform, mDirtyRegion);

}

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

        前面提到，我们假设在SurfaceFlinger服务这一侧，每一个应用程序窗口都是使用一个Layer对象来描述，这些Layer对象都是从LayerBase继承下来的，因此它们可以保存在一个类型为LayerBase的向量中。

        从前面的调用过程可以知道，参数currentLayers里面保存的一系列LayerBase对象正是用来描述系统当前的各个应用程序窗口的，SurfaceFlinger类的成员函数unlockPageFlip依次调用这些LayerBase对象的成员函数unlockPageFlip来让它们有机会执行一些清理工作。由于我们假设这些LayerBase对象的实际类型为Layer，因此，前面调用的实际上就是Layer类的成员函数unlockPageFlip。接下来，我们就继续分析Layer类的成员函数unlockPageFlip的实现。

        Step 7. Layer.unlockPageFlip

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void Layer::unlockPageFlip(
2. const Transform& planeTransform, Region& outDirtyRegion)
3. {
4. Region dirtyRegion(mPostedDirtyRegion);
5. if (!dirtyRegion.isEmpty()) {
6. mPostedDirtyRegion.clear();
7. // The dirty region is given in the layer's coordinate space
8. // transform the dirty region by the surface's transformation
9. // and the global transformation.
10. const Layer::State& s(drawingState());
11. const Transform tr(planeTransform \* s.transform);
12. dirtyRegion = tr.transform(dirtyRegion);
14. // At this point, the dirty region is in screen space.
15. // Make sure it's constrained by the visible region (which
16. // is in screen space as well).
17. dirtyRegion.andSelf(visibleRegionScreen);
18. outDirtyRegion.orSelf(dirtyRegion);
19. }
20. if (visibleRegionScreen.isEmpty()) {
21. // an invisible layer should not hold a freeze-lock
22. // (because it may never be updated and therefore never release it)
23. mFreezeLock.clear();
24. }
25. }

void Layer::unlockPageFlip(

const Transform& planeTransform, Region& outDirtyRegion)

{

Region dirtyRegion(mPostedDirtyRegion);

if (!dirtyRegion.isEmpty()) {

mPostedDirtyRegion.clear();

// The dirty region is given in the layer's coordinate space

// transform the dirty region by the surface's transformation

// and the global transformation.

const Layer::State& s(drawingState());

const Transform tr(planeTransform \* s.transform);

dirtyRegion = tr.transform(dirtyRegion);

// At this point, the dirty region is in screen space.

// Make sure it's constrained by the visible region (which

// is in screen space as well).

dirtyRegion.andSelf(visibleRegionScreen);

outDirtyRegion.orSelf(dirtyRegion);

}

if (visibleRegionScreen.isEmpty()) {

// an invisible layer should not hold a freeze-lock

// (because it may never be updated and therefore never release it)

mFreezeLock.clear();

}

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/Layer.cpp中。

       参数planeTransform用来描述系统显示屏的旋转方向，它是一个变换矩阵，而另外一个参数outDirtyRegion是一个输出参数，指向了SurfaceFlinger类的成员函数mDirtyRegion，即它描述的是SurfaceFlinger服务需要渲染的脏区域。

       Layer类的成员变量mPostedDirtyRegion用来描述当前正在处理的应用程序窗口的脏区域，即可能需要渲染的区域。从前面的Step 5可以知道，Layer类的成员变量visibleRegionScreen是从父类LayerBase继续下来的，用来描述当前正在处理的应用程序窗口的可见区域。这两个区域相关的那部分区域才是当前正在处理的应用程序窗口需要重新渲染的区域，因为一个区域如果是脏的，但是它同时也是不可见的，那么我们是不需要去渲染的。注意，Layer类的成员变量visibleRegionScreen的所描述的区域是相对于显示屏的，而Layer类的成员变量mPostedDirtyRegion所描述的区域是相对于当前正在处理的应用程序窗口的，因此，需要将它转换成相对于显示屏的区域之后，才能将它与前者执行一个相交操作，从而得到当前正在处理的应用程序窗口真正需要渲染的脏区域dirtyRegion。有了这个脏区域dirtyRegion之后，就可以将它组合到输出参数outDirtyRegion中去，以便可以得到SurfaceFlinger服务需要渲染的总脏区域。

       当Layer类的成员变量visibleRegionScreen所描述的区域为空时，就说明当前正在处理的应用程序窗口是不需要参与本次的渲染操作的，因此，这时候就要判断当前正在处理的应用程序窗口是否拥有一个用来冻结系统显示屏的锁。如果有的话，那么就要将这个锁释放掉，避免阻止SurfaceFlinger服务渲染其它应用程序的UI。释放当前正在处理的应用程序窗口所拥有的一个用来冻结系统显示屏的锁是通过调用Layer类的成员变量mFreezeLock所描述的一个FreezeLock对象的成员函数clear来完成的。

       至此，我们就分析完成各个应用程序窗口是如何设置它们当前所要渲染的图形缓冲区以及计算它们的可见区域的了，接下来我们继续分析当当前需要渲染的应用程序窗口只有一个，并且SurfaceFlinger服务在编译的时候指定了USE\_COMPOSITION\_BYPASS宏时，这个唯一的应用程序窗口是如何被渲染的，即分析SurfaceFlinger类的成员函数handleBypassLayer的实现。

      3. handleBypassLayer

      SurfaceFlinger类的成员函数handleBypassLayer跳过应用程序窗口的图形缓冲区合成阶段，它直接将系统中唯一的一个应用程序窗口的图形缓冲区渲染到硬件帧缓冲区中去，它的执行过程如图9所示：

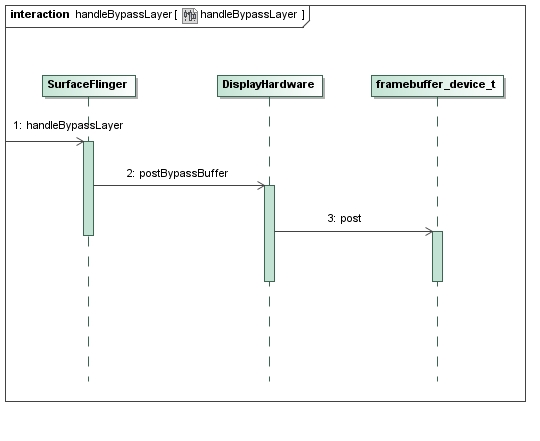


图9 SurfaceFlinger服务直接渲染应用程序窗口的图形缓冲区的过程

      这个过程可以分为3个步骤，接下来我们就详细分析每一个步骤。

      Step 1. SurfaceFlinger.handleBypassLayer

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. **bool** SurfaceFlinger::handleBypassLayer()
2. {
3. sp<Layer> bypassLayer(mBypassLayer.promote());
4. if (bypassLayer != 0) {
5. sp<GraphicBuffer> buffer(bypassLayer->getBypassBuffer());
6. if (buffer!=0 && (buffer->usage & GRALLOC\_USAGE\_HW\_FB)) {
7. const DisplayHardware& hw(graphicPlane(0).displayHardware());
8. hw.postBypassBuffer(buffer->handle);
9. return true;
10. }
11. }
12. return false;
13. }

bool SurfaceFlinger::handleBypassLayer()

{

sp<Layer> bypassLayer(mBypassLayer.promote());

if (bypassLayer != 0) {

sp<GraphicBuffer> buffer(bypassLayer->getBypassBuffer());

if (buffer!=0 && (buffer->usage & GRALLOC\_USAGE\_HW\_FB)) {

const DisplayHardware& hw(graphicPlane(0).displayHardware());

hw.postBypassBuffer(buffer->handle);

return true;

}

}

return false;

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

       从前面第2部分的内容的Step 1可以知道，SurfaceFlinger类的成员变量mBypassLayer指向了系统中唯一需要渲染的应用程序窗口，当它的值不等于0的时候，函数接下来就会检查它当前需要渲染的图形缓冲区buffer是否是直接在硬件帧缓冲区中分配的，即图形缓冲区buffer的成员变量usage的GRALLOC\_USAGE\_HW\_FB位是否等于1。如果是直接在硬件帧缓冲区中分配的话，那么函数最后就会先获得用来描述系统主显示屏的一个DisplayHardware对象hw的成员函数postBypassBuffer来直接渲染图形缓冲区buffer。

      在前面第2部分的内容的Step 1中提到，用来描述系统中唯一需要渲染的应用程序窗口的一个Layer对象的成员变量mBypassState的值会被设置为true。这样做的目的是为了让SurfaceFlinger服务以后在为这个应用程序窗口分配图形缓冲区时，都直接在硬件帧缓冲区中分配，这一点可以参考前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文，这样SurfaceFlinger服务以后就可以直接将该图形缓冲区渲染到硬件帧缓冲区上。与那些在匿名共享内存中分配的图形缓冲区的渲染过程相比，直接在硬件帧缓冲区中分配的图形缓冲区的渲染过程要高效得多，因从就可以提高系统UI的显示性能。

      接下来，我们就继续分析DisplayHardware类的成员函数postBypassBuffer的实现，以便可以了解系统中唯一需要渲染的应用程序窗口的图形缓冲区是如何被直接渲染到硬件帧缓冲区中去的。

      Step 2. DisplayHardware.postBypassBuffer

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. status\_t DisplayHardware::postBypassBuffer(const native\_handle\_t\* handle) const
2. {
3. framebuffer\_device\_t \*fbDev = (framebuffer\_device\_t \*)mNativeWindow->getDevice();
4. return fbDev->post(fbDev, handle);
5. }

status\_t DisplayHardware::postBypassBuffer(const native\_handle\_t\* handle) const

{

framebuffer\_device\_t \*fbDev = (framebuffer\_device\_t \*)mNativeWindow->getDevice();

return fbDev->post(fbDev, handle);

}

      这个函数定义在文件frameworks/base/services/surfaceflinger/DisplayHardware/DisplayHardware.cpp中。

      从前面[Android系统Surface机制的SurfaceFlinger服务对帧缓冲区（Frame Buffer）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8046659)一文可以知道，DisplayHardware类的成员变量mNativeWindow指向的是一个FramebufferNativeWindow对象，调用这个FramebufferNativeWindow对象的成员函数getDevice就可以获得它里面的一个framebuffer\_device\_t对象fbDev。这个framebuffer\_device\_t对象fbDev是在HAL层的模块Gralloc中打开的，通过调用它的成员函数post就可以将指定的图形缓冲区渲染到硬件帧缓冲区中去。

      接下来，我们就继续分析framebuffer\_device\_t类的成员函数post的实现，以便可以了解SurfaceFlinger是如何将一个图形缓冲区渲染到硬件帧缓冲区中去的。

      Step 3. framebuffer\_device\_t.post

      从前面[Android帧缓冲区（Frame Buffer）硬件抽象层（HAL）模块Gralloc的实现原理分析](http://blog.csdn.net/luoshengyang/article/details/7747932)一文可以知道，framebuffer\_device\_t类的成员函数post指向了HAL层的Gralloc模块的函数fb\_post中，后者定义在文件hardware/libhardware/modules/gralloc/framebuffer.cpp。

      HAL层的Gralloc模块的函数fb\_post的作用就是将一个指定的图形缓冲区的内容渲染到硬件帧缓冲区中去。它会分两种情况来考虑。第一种情况是指定的图形缓冲区是直接在硬件帧缓冲区中分配的，这时候该函数就使用IO控制命令FBIOPUT\_VSCREENINFO来直接在硬件帧缓冲区中渲染该图形缓冲区的内容。第二种情况是指定的图形缓冲区是直接在匿名共享内存中分配，这时候该函数就会调用函数memcpy来将图形缓冲区的内容从匿名共享内存拷贝硬件帧缓冲区中去。在我们这个场景中，指定要渲染的图形缓冲区是直接在硬件帧缓冲区中分配的，因此，它最终就会通过IO控制命令FBIOPUT\_VSCREENINFO渲染到硬件帧缓冲区中去。这个过程可以参考[Android帧缓冲区（Frame Buffer）硬件抽象层（HAL）模块Gralloc的实现原理分析](http://blog.csdn.net/luoshengyang/article/details/7747932)一文，这里不再详述。

       至此，我们就分析完成系统中唯一一个需要渲染的应用程序窗口的图形缓冲区的渲染过程了，接下来我们继续分析系统中存在多个需要渲染的应用程序窗口时，SurfaceFlinger服务是如何渲染它们的图形缓冲区的，即分析SurfaceFlinger类的成员函数handleRepaint的实现。

       4. handleRepaint

       SurfaceFlinger类的成员函数handleRepaint是用来合成系统中各个应用程序窗口的图形缓冲区的，以便可以将它们的内容一起渲染到硬件帧缓冲区中去，它的执行过程如图10所示：

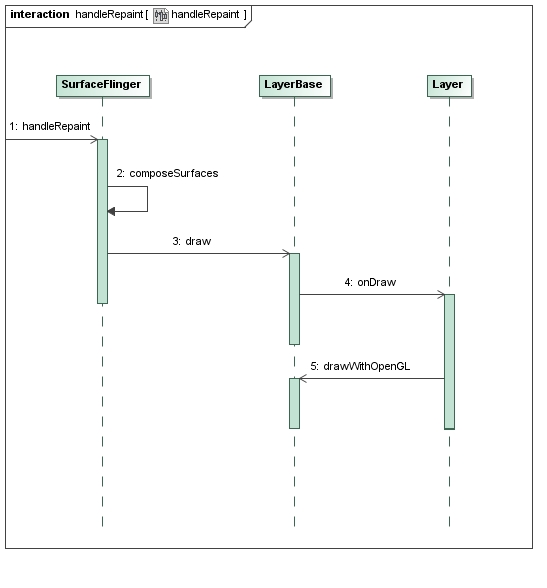


图10 SurfaceFlinger服务合成应用程序窗口的图形缓冲区的过程

        这个过程可以分为5个步骤，接下来我们就详细分析每一个步骤。

        Step 1. SurfaceFlinger.handleRepaint

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::handleRepaint()
2. {
3. // compute the invalid region
4. mInvalidRegion.orSelf(mDirtyRegion);
5. if (mInvalidRegion.isEmpty()) {
6. // nothing to do
7. return;
8. }
10. if (UNLIKELY(mDebugRegion)) {
11. debugFlashRegions();
12. }
14. // set the frame buffer
15. const DisplayHardware& hw(graphicPlane(0).displayHardware());
16. glMatrixMode(GL\_MODELVIEW);
17. glLoadIdentity();
19. uint32\_t flags = hw.getFlags();
20. if ((flags & DisplayHardware::SWAP\_RECTANGLE) ||
21. (flags & DisplayHardware::BUFFER\_PRESERVED))
22. {
23. // we can redraw only what's dirty, but since SWAP\_RECTANGLE only
24. // takes a rectangle, we must make sure to update that whole
25. // rectangle in that case
26. if (flags & DisplayHardware::SWAP\_RECTANGLE) {
27. // TODO: we really should be able to pass a region to
28. // SWAP\_RECTANGLE so that we don't have to redraw all this.
29. mDirtyRegion.set(mInvalidRegion.bounds());
30. } else {
31. // in the BUFFER\_PRESERVED case, obviously, we can update only
32. // what's needed and nothing more.
33. // NOTE: this is NOT a common case, as preserving the backbuffer
34. // is costly and usually involves copying the whole update back.
35. }
36. } else {
37. if (flags & DisplayHardware::PARTIAL\_UPDATES) {
38. // We need to redraw the rectangle that will be updated
39. // (pushed to the framebuffer).
40. // This is needed because PARTIAL\_UPDATES only takes one
41. // rectangle instead of a region (see DisplayHardware::flip())
42. mDirtyRegion.set(mInvalidRegion.bounds());
43. } else {
44. // we need to redraw everything (the whole screen)
45. mDirtyRegion.set(hw.bounds());
46. mInvalidRegion = mDirtyRegion;
47. }
48. }
50. // compose all surfaces
51. composeSurfaces(mDirtyRegion);
53. // clear the dirty regions
54. mDirtyRegion.clear();
55. }

void SurfaceFlinger::handleRepaint()

{

// compute the invalid region

mInvalidRegion.orSelf(mDirtyRegion);

if (mInvalidRegion.isEmpty()) {

// nothing to do

return;

}

if (UNLIKELY(mDebugRegion)) {

debugFlashRegions();

}

// set the frame buffer

const DisplayHardware& hw(graphicPlane(0).displayHardware());

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

uint32\_t flags = hw.getFlags();

if ((flags & DisplayHardware::SWAP\_RECTANGLE) ||

(flags & DisplayHardware::BUFFER\_PRESERVED))

{

// we can redraw only what's dirty, but since SWAP\_RECTANGLE only

// takes a rectangle, we must make sure to update that whole

// rectangle in that case

if (flags & DisplayHardware::SWAP\_RECTANGLE) {

// TODO: we really should be able to pass a region to

// SWAP\_RECTANGLE so that we don't have to redraw all this.

mDirtyRegion.set(mInvalidRegion.bounds());

} else {

// in the BUFFER\_PRESERVED case, obviously, we can update only

// what's needed and nothing more.

// NOTE: this is NOT a common case, as preserving the backbuffer

// is costly and usually involves copying the whole update back.

}

} else {

if (flags & DisplayHardware::PARTIAL\_UPDATES) {

// We need to redraw the rectangle that will be updated

// (pushed to the framebuffer).

// This is needed because PARTIAL\_UPDATES only takes one

// rectangle instead of a region (see DisplayHardware::flip())

mDirtyRegion.set(mInvalidRegion.bounds());

} else {

// we need to redraw everything (the whole screen)

mDirtyRegion.set(hw.bounds());

mInvalidRegion = mDirtyRegion;

}

}

// compose all surfaces

composeSurfaces(mDirtyRegion);

// clear the dirty regions

mDirtyRegion.clear();

}

       这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

       SurfaceFlinger类的成员变量mInvalidRegion用来描述SurfaceFlinger服务需要渲染到硬件帧缓冲区中去的脏区域的。前面提到，SurfaceFlinger类的成员变量mDirtyRegion也是用来描述SurfaceFlinger服务需要渲染的脏区域的，不过，它的作用是用来合成系统中各个应用程序窗口的图形缓冲区的，也就是说，当系统中各个应用程序窗口的图形缓冲区被合成之后，这个成员变量所描述的区域就会被清空，而成员变量mInvalidRegion会一直等到它的内容被渲染到硬件帧缓冲区中去之后，才会被清空。这样就可能会出现这种情况：上一次合成的图形缓冲区还未被渲染到硬件帧缓冲区中去，SurfaceFlinger服务又会执行新一轮的系统中各个应用程序窗口的图形缓冲区的合并操作。在这种情况下，SurfaceFlinger服务就需要将前面两次需要渲染到硬件帧缓冲区中去的区域合并在一起，以便可以正确地反映被刷新的UI。因此，函数在开头的地方，就会先SurfaceFlinger类的成员变量mDirtyRegion所描述的区域组合到成员变量mInvalidRegion所描述的区域中去。

       函数接下来调用用来描述系统主显示屏的一个DisplayHardware对象hw的成员函数getFlags来获得系统所支持的渲染方式，并且保存在一个uint32\_t变量flags中。接下来，我们就分四种情况来讨论系统所支持的渲染方式：

      1. 变量flags的DisplayHardware::PARTIAL\_UPDATES位等于1。在这种情况下，系统在硬件上直接支持部分区域更新功能，不过，这个部分被更新的区域必须要是一个矩形区域。

      2. 变量flags的DisplayHardware::SWAP\_RECTANGLE位等于1。在这种情况下，系统在软件上支持部分区域更新功能，同样，这个部分被更新的区域必须要是一个矩形区域。

      3. 变量flags的DisplayHardware::BUFFER\_PRESERVED位等于1。在这种情况下，系统支持不规则的部分区域更新功能。所谓不规则，就是这个被更新的区域不必是一个矩形区域。

      4. 变量flags的值等于0。在这种情况下，系统不支持部分更新区域，这时候就需要更新整个屏幕的内容。

      在第1种和第2种情况中，由于被更新的区域都必须是一个矩形区域，因此，函数就需要将SurfaceFlinger类的成员变量mDirtyRegion所描述的一个区域设置为包含了所有脏区域的一个最小矩形区域。在第4种情况中，由于需要更新的是整个屏幕的内容，因此，函数就需要将SurfaceFlinger类的成员变量mDirtyRegion所描述的一个区域设置为等于屏幕大小的一个矩形区域。在第3种情况中，就不需要调用被更新的区域。对于第1种、第2种和第3种渲染方式的更多描述，可以参考前面[Android系统Surface机制的SurfaceFlinger服务对帧缓冲区（Frame Buffer）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8046659)一文。

      得到了最终要合成的脏区域mDirtyRegion之后，SurfaceFlinger类的成员函数handleRepaint最后就调用另外一个成员函数composeSurfaces来合成系统中各个应用程序窗口的图形缓冲区，并且在合成操作完成之后，将脏区域mDirtyRegion设置为一个空区域。

      接下来，我们就继续分析SurfaceFlinger类的成员函数composeSurfaces的实现。

      Step 2. SurfaceFlinger.composeSurfaces

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::composeSurfaces(const Region& dirty)
2. {
3. if (UNLIKELY(!mWormholeRegion.isEmpty())) {
4. // should never happen unless the window manager has a bug
5. // draw something...
6. drawWormhole();
7. }
8. const Vector< sp<LayerBase> >& layers(mVisibleLayersSortedByZ);
9. const **size\_t** count = layers.size();
10. for (**size\_t** i=0 ; i<count ; ++i) {
11. const sp<LayerBase>& layer(layers[i]);
12. const Region clip(dirty.intersect(layer->visibleRegionScreen));
13. if (!clip.isEmpty()) {
14. layer->draw(clip);
15. }
16. }
17. }

void SurfaceFlinger::composeSurfaces(const Region& dirty)

{

if (UNLIKELY(!mWormholeRegion.isEmpty())) {

// should never happen unless the window manager has a bug

// draw something...

drawWormhole();

}

const Vector< sp<LayerBase>>& layers(mVisibleLayersSortedByZ);

const size\_t count = layers.size();

for (size\_t i=0 ; i<count ; ++i) {

const sp<LayerBase>& layer(layers[i]);

const Region clip(dirty.intersect(layer->visibleRegionScreen));

if (!clip.isEmpty()) {

layer->draw(clip);

}

}

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

        参数dirty用来描述即将要被渲染的脏区域，它是从前面的第1步传进来的，即为SurfaceFlinger类的成员变量mDirtyRegion所描述的区域。

        SurfaceFlinger类的成员变量mVisibleLayersSortedByZ保存了系统当前所有的可见应用程序窗口，这些应用程序窗口是前面第2部分内容的Step 1中计算得到的，而且都是需要渲染的。 SurfaceFlinger类的成员函数composeSurfaces依次检查这些应用程序窗口的可见区域是否与即将要渲染的的脏区域有交集。如果有的话，即变量clip所描述的区域不为空，那么接下来SurfaceFlinger类的成员函数composeSurfaces就会分别调用与这些应用程序窗口对应的一个LayerBase对象的成员函数draw来将它们需要渲染的图形缓冲区合成到系统的主显示屏来。

       接下来，我们就继续分析LayerBase类的成员函数draw的实现，以便了解SurfaceFlinger服务合成各个应用程序窗口的图形缓冲区的过程。

       Step 3. LayerBase.draw

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void LayerBase::draw(const Region& clip) const
2. {
3. // reset GL state
4. glEnable(GL\_SCISSOR\_TEST);
6. onDraw(clip);
7. }

void LayerBase::draw(const Region& clip) const

{

// reset GL state

glEnable(GL\_SCISSOR\_TEST);

onDraw(clip);

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/LayerBase.cpp中。

        LayerBase类的成员函数draw的实现很简单，它只是调用了另外一个成员函数onDraw来通知各个应用程序窗口重新绘制参数clip所描述的一个区域。LayerBase类的成员函数onDraw是一个虚函数，这是由其子类来重写的。前面我们假设系统中的所有应用程序窗口都是使用一个Layer对象来描述的，而Layer类是从LayerBase类继承下来的，并且重写了其成员函数onDraw。因此，接下来我们就继续分析Layer类的成员函数onDraw的实现。

        Step 4. Layer.onDraw

        这个函数定义在文件frameworks/base/services/surfaceflinger/Layer.cpp中，用来绘制一个应用程序窗口的指定区域，我们分段来阅读：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void Layer::onDraw(const Region& clip) const
2. {
3. Texture tex(mBufferManager.getActiveTexture());
4. if (tex.name == -1LU) {
5. // the texture has not been created yet, this Layer has
6. // in fact never been drawn into. This happens frequently with
7. // SurfaceView because the WindowManager can't know when the client
8. // has drawn the first time.
10. // If there is nothing under us, we paint the screen in black, otherwise
11. // we just skip this update.
13. // figure out if there is something below us
14. Region under;
15. const SurfaceFlinger::LayerVector& drawingLayers(mFlinger->mDrawingState.layersSortedByZ);
16. const **size\_t** count = drawingLayers.size();
17. for (**size\_t** i=0 ; i<count ; ++i) {
18. const sp<LayerBase>& layer(drawingLayers[i]);
19. if (layer.get() == static\_cast<LayerBase const\*>(this))
20. break;
21. under.orSelf(layer->visibleRegionScreen);
22. }
23. // if not everything below us is covered, we plug the holes!
24. Region holes(clip.subtract(under));
25. if (!holes.isEmpty()) {
26. clearWithOpenGL(holes, 0, 0, 0, 1);
27. }
28. return;
29. }

void Layer::onDraw(const Region& clip) const

{

Texture tex(mBufferManager.getActiveTexture());

if (tex.name == -1LU) {

// the texture has not been created yet, this Layer has

// in fact never been drawn into. This happens frequently with

// SurfaceView because the WindowManager can't know when the client

// has drawn the first time.

// If there is nothing under us, we paint the screen in black, otherwise

// we just skip this update.

// figure out if there is something below us

Region under;

const SurfaceFlinger::LayerVector& drawingLayers(mFlinger->mDrawingState.layersSortedByZ);

const size\_t count = drawingLayers.size();

for (size\_t i=0 ; i<count ; ++i) {

const sp<LayerBase>& layer(drawingLayers[i]);

if (layer.get() == static\_cast<LayerBase const\*>(this))

break;

under.orSelf(layer->visibleRegionScreen);

}

// if not everything below us is covered, we plug the holes!

Region holes(clip.subtract(under));

if (!holes.isEmpty()) {

clearWithOpenGL(holes, 0, 0, 0, 1);

}

return;

}

        这段代码主要是用来处应用程序窗口的纹理尚未创建好的情况。

        在纹理未创建好的情况下，一个应用程序窗口是不应该被渲染的。这时候函数首先将位于当前正在处理的应用程序窗口下面的所有其它应用程序窗口的可见区域组合起来，并且保存在变量under所描述的区域中。由于这时候当前正在处理的应用程序窗口不会被绘制，因此，如果变量under所描述的区域小于参数clip所描述的区域，即变量holes所描述的区域不为空，那么SurfaceFlinger服务所要求缓制的区域clip就会留下一个洞。这个洞会被绘制成黑色，这是通过调用函数clearWithOpenGL来实现的。绘制完成之后，函数就可以直接返回了。

        在前面第2部分的Step 4中提到，Layer类的成员变量mBufferManager描述的是一个BufferManager对象，通过调用它的成员函数initEglImage或者loadTexture就可以根据一个应用程序窗口的当前需要渲染的图形缓冲区来创建出一个纹理对象。这个纹理对象就保存在BufferManager类的成员变量mFailoverTexture或者另外一个成员变量mBufferData所描述的一个BufferData数组中，取决于系统是否支持在硬件上直接创建纹理对象。这个纹理对象是使用一个Texture对象来描述的，并且可以通过调用Layer类的成员变量mBufferManager描述的是一个BufferManager对象的成员函数getActiveTexture来获得。如果获得的Texture对象的名称name等于-1，那么就说明当前正在处理的应用程序窗口尚未创建好需要渲染的纹理。

        我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. #ifdef USE\_COMPOSITION\_BYPASS
2. sp<GraphicBuffer> buffer(mBufferManager.getActiveBuffer());
3. if ((buffer != NULL) && (buffer->transform)) {
4. // Here we have a "bypass" buffer, but we need to composite it
5. // most likely because it's not fullscreen anymore.
6. // Since the buffer may have a transformation applied by the client
7. // we need to inverse this transformation here.
9. // calculate the inverse of the buffer transform
10. const uint32\_t mask = HAL\_TRANSFORM\_FLIP\_V | HAL\_TRANSFORM\_FLIP\_H;
11. const uint32\_t bufferTransformInverse = buffer->transform ^ mask;
13. // To accomplish the inverse transform, we use "mBufferTransform"
14. // which is not used by Layer.cpp
15. const\_cast<Layer\*>(this)->mBufferTransform = bufferTransformInverse;
16. drawWithOpenGL(clip, tex);
17. // reset to "no transfrom"
18. const\_cast<Layer\*>(this)->mBufferTransform = 0;
19. return;
20. }
21. #endif

#ifdef USE\_COMPOSITION\_BYPASS

sp<GraphicBuffer> buffer(mBufferManager.getActiveBuffer());

if ((buffer != NULL) && (buffer->transform)) {

// Here we have a "bypass" buffer, but we need to composite it

// most likely because it's not fullscreen anymore.

// Since the buffer may have a transformation applied by the client

// we need to inverse this transformation here.

// calculate the inverse of the buffer transform

const uint32\_t mask = HAL\_TRANSFORM\_FLIP\_V | HAL\_TRANSFORM\_FLIP\_H;

const uint32\_t bufferTransformInverse = buffer->transform ^ mask;

// To accomplish the inverse transform, we use "mBufferTransform"

// which is not used by Layer.cpp

const\_cast<Layer\*>(this)->mBufferTransform = bufferTransformInverse;

drawWithOpenGL(clip, tex);

// reset to "no transfrom"

const\_cast<Layer\*>(this)->mBufferTransform = 0;

return;

}

#endif

       这段代码用来检查当前正在处理的应用程序窗口的图形缓冲区是否是一个可以跳过合成阶段的图形缓冲区。本来这种图形缓冲区是可以直接渲染到硬件帧缓冲区中去的，但是由于它不是全屏显示的，因此就需要与其它应用程序窗口的图形缓冲区进行合并操作。如果这个图形缓冲区之前曾经被旋转过，例如，被水平翻转或者垂直翻转过，那么在对它进行合并之前，还需要将它的旋转方向恢复回来。

       当用来描述一个图形缓冲区的一个GraphicBuffer对象的成员变量transform的值不等于0时，那么就说明这个图形缓冲区是被旋转过的，这时候函数就会对这个成员变量的值的HAL\_TRANSFORM\_FLIP\_V位或者HAL\_TRANSFORM\_FLIP\_H位进行取反，目的就是为了恢复它之前的旋转方向。反转后得到的方向就保存在Layer类的成员变量mBufferTransform中。Layer类的成员变量mBufferTransform是从父类LayerBase继承下来的，接下来在调用从父类LayerBase继承下来的成员函数drawWithOpenGL来绘制当前正在处理的应用程序窗口时，就需要使用到它来设置纹理的旋转方向。在后面的Step 5中，我们再详细分析LayerBase类的成员函数drawWithOpenGL的实现。

       我们继续往下阅读最后一行代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. drawWithOpenGL(clip, tex);

drawWithOpenGL(clip, tex);

}

        如果当前正在处理的应用程序窗口的图形缓冲区没有被旋转过，或者这个图形缓冲区本来就需要进行合并的，那么Layer类的成员函数onDraw最后就会调用从父类LayerBase继承下来的成员函数drawWithOpenGL来将这个图形缓冲区的内容绘制在系统的主显示屏的指定区域上来。这个图形缓冲区的内容是使用纹理象tex来描述的，而指定的主显示屏区域是由参数clip来描述的。

        接下来，我们就继续分析LayerBase类的成员函数drawWithOpenGL的实现，以便可以了解一个应用程序窗口的绘制过程，即它的图形缓冲区被合成到系统主显示屏的过程。

        Step 5. LayerBase.drawWithOpenGL

        这个函数定义在文件frameworks/base/services/surfaceflinger/LayerBase.cpp中，它通过OpenGL提供的接口来绘制一个应用程序窗口的指定区域，我们分段来阅读：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void LayerBase::drawWithOpenGL(const Region& clip, const Texture& texture) const
2. {
3. const DisplayHardware& hw(graphicPlane(0).displayHardware());
4. const uint32\_t fbHeight = hw.getHeight();
5. const State& s(drawingState());
7. // bind our texture
8. TextureManager::activateTexture(texture, needsFiltering());
9. uint32\_t width  = texture.width;
10. uint32\_t height = texture.height;
12. GLenum src = mPremultipliedAlpha ? GL\_ONE : GL\_SRC\_ALPHA;
13. if (UNLIKELY(s.alpha < 0xFF)) {
14. const GLfloat alpha = s.alpha \* (1.0f/255.0f);
15. if (mPremultipliedAlpha) {
16. glColor4f(alpha, alpha, alpha, alpha);
17. } else {
18. glColor4f(1, 1, 1, alpha);
19. }
20. glEnable(GL\_BLEND);
21. glBlendFunc(src, GL\_ONE\_MINUS\_SRC\_ALPHA);
22. glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);
23. } else {
24. glColor4f(1, 1, 1, 1);
25. glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);
26. if (needsBlending()) {
27. glEnable(GL\_BLEND);
28. glBlendFunc(src, GL\_ONE\_MINUS\_SRC\_ALPHA);
29. } else {
30. glDisable(GL\_BLEND);
31. }
32. }

void LayerBase::drawWithOpenGL(const Region& clip, const Texture& texture) const

{

const DisplayHardware& hw(graphicPlane(0).displayHardware());

const uint32\_t fbHeight = hw.getHeight();

const State& s(drawingState());

// bind our texture

TextureManager::activateTexture(texture, needsFiltering());

uint32\_t width = texture.width;

uint32\_t height = texture.height;

GLenum src = mPremultipliedAlpha ? GL\_ONE : GL\_SRC\_ALPHA;

if (UNLIKELY(s.alpha < 0xFF)) {

const GLfloat alpha = s.alpha \* (1.0f/255.0f);

if (mPremultipliedAlpha) {

glColor4f(alpha, alpha, alpha, alpha);

} else {

glColor4f(1, 1, 1, alpha);

}

glEnable(GL\_BLEND);

glBlendFunc(src, GL\_ONE\_MINUS\_SRC\_ALPHA);

glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

} else {

glColor4f(1, 1, 1, 1);

glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);

if (needsBlending()) {

glEnable(GL\_BLEND);

glBlendFunc(src, GL\_ONE\_MINUS\_SRC\_ALPHA);

} else {

glDisable(GL\_BLEND);

}

}

        参数clip用来描述要绘制的区域，而参数texture用来描述要绘制的纹理。

        这段代码首先得到主显示屏的高度fbHeight、要绘制的纹理texture的宽度width和高度height，以及用来描述当前正在处理的应用程序窗口状态的一个State对象s，接下来就是调用函数再设置纹理texture的绘制模式，即是否需要以混合模式来绘制，这是通过调用函数glEnable(GL\_BLEND)来实现的。在需要混合模式来绘制纹理texture的情况下，还需要调用函数glBlendFunc来指定混合函数。

        在以下两种情况下，纹理texture需要以混合模式来绘制：

        1. 当前正在处理的应用程序窗口的Alpha通道的值小于0xFF，即State对象s的成员变量alpha的值小于0xFF，这表明该窗口的背景是半透明的。

        2. 当前正在处理的应用程序窗口的像素格式是半透明的，这是通过调用LayerBase类的成员函数needsBlending来判断的。Layer类重写了父类LayerBase的成员函数needsBlending。当一个Layer对象所描述的应用程序窗口的像素格式是半透明的时候，它就会将它的成员变量mNeedsBlending的值设置为true。这样，我们就可以调用Layer类的成员函数needsBlending来检查它的成员变量mNeedsBlending是否为true来判断一个应用程序窗口的像素格式是否是半透明的。这一点可以参考前面[Android应用程序请求SurfaceFlinger服务创建Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7884628)一文的Step 6。

       我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. /\*
2. \*  compute texture coordinates
3. \*  here, we handle NPOT, cropping and buffer transformations
4. \*/
6. GLfloat cl, ct, cr, cb;
7. if (!mBufferCrop.isEmpty()) {
8. // source is cropped
9. const GLfloat us = (texture.NPOTAdjust ? texture.wScale : 1.0f) / width;
10. const GLfloat vs = (texture.NPOTAdjust ? texture.hScale : 1.0f) / height;
11. cl = mBufferCrop.left   \* us;
12. ct = mBufferCrop.top    \* vs;
13. cr = mBufferCrop.right  \* us;
14. cb = mBufferCrop.bottom \* vs;
15. } else {
16. cl = 0;
17. ct = 0;
18. cr = (texture.NPOTAdjust ? texture.wScale : 1.0f);
19. cb = (texture.NPOTAdjust ? texture.hScale : 1.0f);
20. }

/\*

\* compute texture coordinates

\* here, we handle NPOT, cropping and buffer transformations

\*/

GLfloat cl, ct, cr, cb;

if (!mBufferCrop.isEmpty()) {

// source is cropped

const GLfloat us = (texture.NPOTAdjust ? texture.wScale : 1.0f) / width;

const GLfloat vs = (texture.NPOTAdjust ? texture.hScale : 1.0f) / height;

cl = mBufferCrop.left \* us;

ct = mBufferCrop.top \* vs;

cr = mBufferCrop.right \* us;

cb = mBufferCrop.bottom \* vs;

} else {

cl = 0;

ct = 0;

cr = (texture.NPOTAdjust ? texture.wScale : 1.0f);

cb = (texture.NPOTAdjust ? texture.hScale : 1.0f);

}

        这段代码用来计算纹理坐标，并且保存在变量cl、ct、cr和cb中，用来描述纹理texture的四个角的坐标。

        从前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文可以知道，如果Android应用程序设置了一个窗口的纹理坐标，那么SurfaceFlinger服务就会通过调用LayerBase类的成员函数setBufferCrop来保存在成员变量mBufferCrop中。另一方面，如果Android应用程序没有指定一个窗口的纹理坐标，那么这个窗口的纹理坐标的默认值就使用要绘制的纹理的四个角的坐标来描述。注意，在计算纹理坐标的时候，还要考虑纹理的大小，以及纹理本身所设置的缩放因子，以便可以正确地将纹理绘制在应用程序窗口中。

        我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. /\*
2. \* For the buffer transformation, we apply the rotation last.
3. \* Since we're transforming the texture-coordinates, we need
4. \* to apply the inverse of the buffer transformation:
5. \*   inverse( FLIP\_V -> FLIP\_H -> ROT\_90 )
6. \*   <=> inverse( ROT\_90 \* FLIP\_H \* FLIP\_V )
7. \*    =  inverse(FLIP\_V) \* inverse(FLIP\_H) \* inverse(ROT\_90)
8. \*    =  FLIP\_V \* FLIP\_H \* ROT\_270
9. \*   <=> ROT\_270 -> FLIP\_H -> FLIP\_V
10. \*
11. \* The rotation is performed first, in the texture coordinate space.
12. \*
13. \*/
15. struct TexCoords {
16. GLfloat u;
17. GLfloat v;
18. };
20. enum {
21. // name of the corners in the texture map
22. LB = 0, // left-bottom
23. LT = 1, // left-top
24. RT = 2, // right-top
25. RB = 3  // right-bottom
26. };
28. // vertices in screen space
29. **int** vLT = LB;
30. **int** vLB = LT;
31. **int** vRB = RT;
32. **int** vRT = RB;
34. // the texture's source is rotated
35. uint32\_t transform = mBufferTransform;
36. if (transform & HAL\_TRANSFORM\_ROT\_90) {
37. vLT = RB;
38. vLB = LB;
39. vRB = LT;
40. vRT = RT;
41. }
42. if (transform & HAL\_TRANSFORM\_FLIP\_V) {
43. swap(vLT, vLB);
44. swap(vRT, vRB);
45. }
46. if (transform & HAL\_TRANSFORM\_FLIP\_H) {
47. swap(vLT, vRT);
48. swap(vLB, vRB);
49. }
51. TexCoords texCoords[4];
52. texCoords[vLT].u = cl;
53. texCoords[vLT].v = ct;
54. texCoords[vLB].u = cl;
55. texCoords[vLB].v = cb;
56. texCoords[vRB].u = cr;
57. texCoords[vRB].v = cb;
58. texCoords[vRT].u = cr;
59. texCoords[vRT].v = ct;

/\*

\* For the buffer transformation, we apply the rotation last.

\* Since we're transforming the texture-coordinates, we need

\* to apply the inverse of the buffer transformation:

\* inverse( FLIP\_V -> FLIP\_H -> ROT\_90 )

\* <=> inverse( ROT\_90 \* FLIP\_H \* FLIP\_V )

\* = inverse(FLIP\_V) \* inverse(FLIP\_H) \* inverse(ROT\_90)

\* = FLIP\_V \* FLIP\_H \* ROT\_270

\* <=> ROT\_270 -> FLIP\_H -> FLIP\_V

\*

\* The rotation is performed first, in the texture coordinate space.

\*

\*/

struct TexCoords {

GLfloat u;

GLfloat v;

};

enum {

// name of the corners in the texture map

LB = 0, // left-bottom

LT = 1, // left-top

RT = 2, // right-top

RB = 3 // right-bottom

};

// vertices in screen space

int vLT = LB;

int vLB = LT;

int vRB = RT;

int vRT = RB;

// the texture's source is rotated

uint32\_t transform = mBufferTransform;

if (transform & HAL\_TRANSFORM\_ROT\_90) {

vLT = RB;

vLB = LB;

vRB = LT;

vRT = RT;

}

if (transform & HAL\_TRANSFORM\_FLIP\_V) {

swap(vLT, vLB);

swap(vRT, vRB);

}

if (transform & HAL\_TRANSFORM\_FLIP\_H) {

swap(vLT, vRT);

swap(vLB, vRB);

}

TexCoords texCoords[4];

texCoords[vLT].u = cl;

texCoords[vLT].v = ct;

texCoords[vLB].u = cl;

texCoords[vLB].v = cb;

texCoords[vRB].u = cr;

texCoords[vRB].v = cb;

texCoords[vRT].u = cr;

texCoords[vRT].v = ct;

        这段代码主要根据当前正在处理的应用程序窗口的旋转方向来调整前面所计算得到的纹理坐标。

        从前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文可以知道，如果Android应用程序设置了一个窗口的旋转方向时，那么SurfaceFlinger服务就会通过调用LayerBase类的成员函数setBufferTransform来保存在成员变量mBufferTransform中，因此，这段代码就可以根据这个成员变量的值来相应地调用前面所计算得到的纹理坐标。

        我们继续往下阅读代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. if (needsDithering()) {
2. glEnable(GL\_DITHER);
3. } else {
4. glDisable(GL\_DITHER);
5. }

if (needsDithering()) {

glEnable(GL\_DITHER);

} else {

glDisable(GL\_DITHER);

}

        这段代码用来检查是否需要以抖动的方式来绘制纹理，如果需要的话，就调用函数glEnable(GL\_DITHER)来启动抖动功能，否则的话，就调用函数glDisable(GL\_DITHER)来关闭抖动功能。

        从前面[Android应用程序请求SurfaceFlinger服务渲染Surface的过程分析](http://blog.csdn.net/luoshengyang/article/details/7932268)一文可以知道，如果一个Android应用程序窗口的像素格式的红色通道的位数大于系统主显示屏的像素格式的红色通道的位数时，SurfaceFlinger服务就会将用来描述该Android应用程序窗口的一个Layer对象的成员变量mNeedsDithering的值设置为true。Layer类重写了父类LayerBase的成员函数needsDithering，它通过检查其成员变量mNeedsDithering的值是否等于true来告诉这段代码用来检查是否需要以抖动的方式来绘制纹理。

        我们继续往下阅读最后一段代码：

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);
2. glVertexPointer(2, GL\_FLOAT, 0, mVertices);
3. glTexCoordPointer(2, GL\_FLOAT, 0, texCoords);
5. Region::const\_iterator it = clip.begin();
6. Region::const\_iterator const end = clip.end();
7. while (it != end) {
8. const Rect& r = \*it++;
9. const GLint sy = fbHeight - (r.top + r.height());
10. glScissor(r.left, sy, r.width(), r.height());
11. glDrawArrays(GL\_TRIANGLE\_FAN, 0, 4);
12. }
13. glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);
14. }

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, mVertices);

glTexCoordPointer(2, GL\_FLOAT, 0, texCoords);

Region::const\_iterator it = clip.begin();

Region::const\_iterator const end = clip.end();

while (it != end) {

const Rect& r = \*it++;

const GLint sy = fbHeight - (r.top + r.height());

glScissor(r.left, sy, r.width(), r.height());

glDrawArrays(GL\_TRIANGLE\_FAN, 0, 4);

}

glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);

}

       这段代码首先调用函数glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY)来指定使用顶点数组的方式绘制当前正在处理的应用程序窗口，接着又分别调用函数glVertexPointer和glTexCoordPointer来指定要绘制的顶点数组以及纹理坐标。设置好要绘制的顶点数组以及纹理坐标之后，最后就要设置裁剪区域，以便可以调用函数glDrawArrays来绘制前面指定的顶点数组以及纹理。

        前面提到，参数clip用来描述要绘制的区域，而要绘制的区域即为当前需要指定的裁剪区域。由于参数clip所描述的区域是可以由一系列的矩形区域来组成的，因此，这段代码就通过一个while循环来遍历里面的每一个矩形区域，并且调用函数glScissor来分别将它们设置为裁剪区域。

        至此，我们就分析完成系统中各个应用程序窗口的图形缓冲区的合成过程了，这个过程其实就是分别根据各个应用程序窗口的图形缓冲区来创建一个纹理对象，并且结合各个应用程序窗口的UI元数据来将该纹理对象绘制在系统主显示屏的指定区域中，而这个指定的区域即为系统的脏区域。

        接下来，我们就继续分析系统中各个应用程序窗口将自己的图形缓冲区合成到系统的主显示屏之后，SurfaceFlinger服务是如何将系统的主显示屏渲染到硬件帧缓冲区中去的，即分析SurfaceFlinger类的成员函数postFramebuffer的实现。

        5. postFramebuffer

        SurfaceFlinger类的成员函数postFramebuffer用来将系统的主显示屏的内容渲染到硬件帧缓冲区中去，它的执行过程如图11所示：

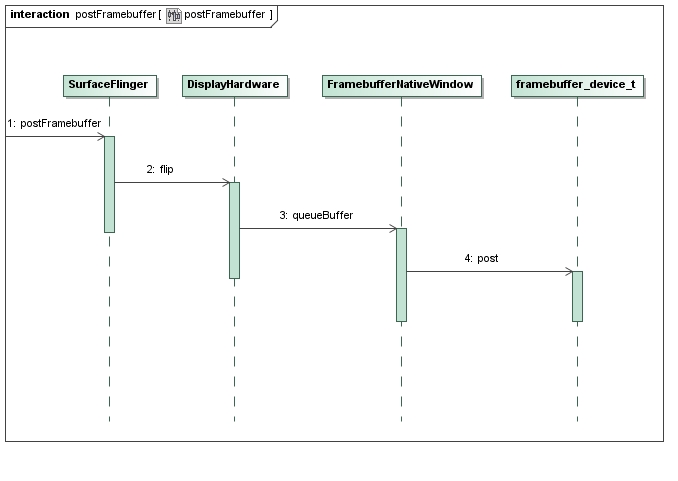


图11 SurfaceFlinger服务渲染系统主显示屏的内容到硬件帧缓冲区的过程

        这个过程可以划分为4步骤，接下来我们就详细分析每一个步骤。

        Step 1. SurfaceFlinger.postFramebuffer

**[cpp]**[view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/8079456)

1. void SurfaceFlinger::postFramebuffer()
2. {
3. if (!mInvalidRegion.isEmpty()) {
4. const DisplayHardware& hw(graphicPlane(0).displayHardware());
5. const nsecs\_t now = systemTime();
6. mDebugInSwapBuffers = now;
7. hw.flip(mInvalidRegion);
8. mLastSwapBufferTime = systemTime() - now;
9. mDebugInSwapBuffers = 0;
10. mInvalidRegion.clear();
11. }
12. }

void SurfaceFlinger::postFramebuffer()

{

if (!mInvalidRegion.isEmpty()) {

const DisplayHardware& hw(graphicPlane(0).displayHardware());

const nsecs\_t now = systemTime();

mDebugInSwapBuffers = now;

hw.flip(mInvalidRegion);

mLastSwapBufferTime = systemTime() - now;

mDebugInSwapBuffers = 0;

mInvalidRegion.clear();

}

}

        这个函数定义在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中。

        在前面第4部分内容的Step 1中提到，SurfaceFlinger类的成员变量mInvalidRegion用来描述系统主显示屏的脏区域，即SurfaceFlinger服务当前需要渲染的区域。函数首先得到用来描述系统主显示屏的一个DisplayHardware对象hw，接着再调用这个DisplayHardware对象hw的成员函数flip来渲染这个脏区域。

        接下来，我们就继续分析DisplayHardware类的成员函数flip的实现。

        Step 2.  DisplayHardware.flip

        这个函数定义在文件frameworks/base/services/surfaceflinger/DisplayHardware/DisplayHardware.cpp中，在前面[Android系统Surface机制的SurfaceFlinger服务对帧缓冲区（Frame Buffer）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8046659)一文中，我们已经分析过它的实现了。这个函数会检查系统是否支持部分更新功能。如果支持的话，那么就先设置要更新的区域，否则的话，就直接调用函数eglSwapBuffers来将前面已经合成好的了图形缓冲区渲染到硬件帧缓冲区去。

        从前面[Android系统Surface机制的SurfaceFlinger服务对帧缓冲区（Frame Buffer）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8046659)一文可以知道，调用函数eglSwapBuffers在渲染图形缓冲区的时候，会导致FramebufferNativeWindow类的成员函数queueBuffer被调用，后者会通过HAL层的Gralloc模块来执行渲染硬件帧缓冲区的操作。

       Step 3. FramebufferNativeWindow.queueBuffer

       这个函数定义在文件frameworks/base/libs/ui/FramebufferNativeWindow.cpp中，同样，在前面[Android系统Surface机制的SurfaceFlinger服务对帧缓冲区（Frame Buffer）的管理分析](http://blog.csdn.net/luoshengyang/article/details/8046659)一文中，我们已经分析过它的实现了。这个函数主要就是通过HAL层的Gralloc模块中的framebuffer\_device\_t设备的成员函数post来执行渲染硬件帧缓冲区的操作。

       Step 4. framebuffer\_device\_t.post

       这个函数指向定义在HAL层的Gralloc模块中的函数fb\_post，后者定义在文件hardware/libhardware/modules/gralloc/framebuffer.cpp，在前面[Android帧缓冲区（Frame Buffer）硬件抽象层（HAL）模块Gralloc的实现原理分析](http://blog.csdn.net/luoshengyang/article/details/7747932)一文中，我们已经分析过这个函数的实现了。由于要渲染的图形缓冲区是用于渲染系统主显示屏的，因此，它是直接在硬件帧缓冲区上分配的，这时候函数fb\_post就会通过IO控制命令FBIOPUT\_VSCREENINFO来通知位内核空间的fb驱动来将系统主显示屏的UI绘制出来。

       至此，SurfaceFlinger服务渲染系统主显示屏的内容到硬件帧缓冲区的过程就分析完成了，整个SurfaceFlinger服务渲染应用程序UI的过程也分析完成了。

       这样，我们就通过[Android应用程序与SurfaceFlinger服务的关系概述和学习计划](http://blog.csdn.net/luoshengyang/article/details/7846923)和[Android系统Surface机制的SurfaceFlinger服务简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8010977)这两个系列的文章系统地分析了Android系统的SurfaceFlinger服务的实现，为后面我们后面进一步分析Android系统的UI架构打下坚实的基础！