### 1 invalidate方法

#### 1.1 代码流程

##### 1.1.1 View

public void invalidate() {

invalidate(true);}void invalidate(boolean invalidateCache) {

invalidateInternal(0, 0, mRight - mLeft, mBottom - mTop, invalidateCache, true);}void invalidateInternal(int l, int t, int r, int b, boolean invalidateCache, boolean fullInvalidate) {

if (mGhostView != null) {

mGhostView.invalidate(true);

return;

}

//这里判断该子View是否可见或者是否处于动画中

if (skipInvalidate()) {

return;

}

//根据View的标记位来判断该子View是否需要重绘，假如View没有任何变化，那么就不需要重绘

if ((mPrivateFlags & (PFLAG\_DRAWN | PFLAG\_HAS\_BOUNDS)) == (PFLAG\_DRAWN | PFLAG\_HAS\_BOUNDS)

|| (invalidateCache && (mPrivateFlags & PFLAG\_DRAWING\_CACHE\_VALID) == PFLAG\_DRAWING\_CACHE\_VALID)

|| (mPrivateFlags & PFLAG\_INVALIDATED) != PFLAG\_INVALIDATED

|| (fullInvalidate && isOpaque() != mLastIsOpaque)) {

if (fullInvalidate) {

mLastIsOpaque = isOpaque();

mPrivateFlags &= ~PFLAG\_DRAWN;

}

//设置PFLAG\_DIRTY标记位

mPrivateFlags |= PFLAG\_DIRTY;

if (invalidateCache) {

mPrivateFlags |= PFLAG\_INVALIDATED;

mPrivateFlags &= ~PFLAG\_DRAWING\_CACHE\_VALID;

}

// Propagate the damage rectangle to the parent view.

//把需要重绘的区域传递给父容器

final AttachInfo ai = mAttachInfo;

final ViewParent p = mParent;

if (p != null && ai != null && l < r && t < b) {

final Rect damage = ai.mTmpInvalRect;

damage.set(l, t, r, b);

//调用父容器的方法，向上传递事件

p.invalidateChild(this, damage);

}

...

}}

可以看出，invalidate有多个重载方法，但最终都会调用invalidateInternal方法，在这个方法内部，进行了一系列的判断，判断View是否需要重绘，接着为该View设置标记位，然后把需要重绘的区域传递给父容器，即调用父容器的invalidateChild方法。

##### 1.1.2 ViewGroup

**ViewGroup.invalidateChild()**

public final void invalidateChild(View child, final Rect dirty) {

//设置 parent 等于自身

ViewParent parent = this;

final AttachInfo attachInfo = mAttachInfo;

if (attachInfo != null) {

......

//储存子View的mLeft和mTop值

final int[] location = attachInfo.mInvalidateChildLocation;

location[CHILD\_LEFT\_INDEX] = child.mLeft;

location[CHILD\_TOP\_INDEX] = child.mTop;

......

do {

View view = null;

if (parent instanceof View) {

view = (View) parent;

}

if (drawAnimation) {

if (view != null) {

view.mPrivateFlags |= PFLAG\_DRAW\_ANIMATION;

} else if (parent instanceof ViewRootImpl) {

((ViewRootImpl) parent).mIsAnimating = true;

}

}

// If the parent is dirty opaque or not dirty, mark it dirty with the opaque

// flag coming from the child that initiated the invalidate

if (view != null) {

if ((view.mViewFlags & FADING\_EDGE\_MASK) != 0 &&

view.getSolidColor() == 0) {

opaqueFlag = PFLAG\_DIRTY;

}

if ((view.mPrivateFlags & PFLAG\_DIRTY\_MASK) != PFLAG\_DIRTY) {

//对当前View的标记位进行设置

view.mPrivateFlags = (view.mPrivateFlags & ~PFLAG\_DIRTY\_MASK) | opaqueFlag;

}

}

//调用ViewGrup的invalidateChildInParent，如果已经达到最顶层view,则调用ViewRootImpl的invalidateChildInParent。

parent = parent.invalidateChildInParent(location, dirty);

if (view != null) {

// Account for transform on current parent

Matrix m = view.getMatrix();

if (!m.isIdentity()) {

RectF boundingRect = attachInfo.mTmpTransformRect;

boundingRect.set(dirty);

m.mapRect(boundingRect);

dirty.set((int) (boundingRect.left - 0.5f),

(int) (boundingRect.top - 0.5f),

(int) (boundingRect.right + 0.5f),

(int) (boundingRect.bottom + 0.5f));

}

}

} while (parent != null);

}}

可以看到，在该方法内部，先设置当前视图的标记位，接着有一个do…while…循环，该循环的作用主要是不断向上回溯父容器，求得父容器和子View需要重绘的区域的并集(dirty)。当父容器不是ViewRootImpl的时候，调用的是ViewGroup的invalidateChildInParent方法。

**ViewGroup.invalidateChildInParent()**

public ViewParent invalidateChildInParent(final int[] location, final Rect dirty) {

if ((mPrivateFlags & PFLAG\_DRAWN) == PFLAG\_DRAWN ||

(mPrivateFlags & PFLAG\_DRAWING\_CACHE\_VALID) == PFLAG\_DRAWING\_CACHE\_VALID) {

if ((mGroupFlags & (FLAG\_OPTIMIZE\_INVALIDATE | FLAG\_ANIMATION\_DONE)) !=

FLAG\_OPTIMIZE\_INVALIDATE) {

//将dirty中的坐标转化为父容器中的坐标，考虑mScrollX和mScrollY的影响

dirty.offset(location[CHILD\_LEFT\_INDEX] - mScrollX,

location[CHILD\_TOP\_INDEX] - mScrollY);

if ((mGroupFlags & FLAG\_CLIP\_CHILDREN) == 0) {

//求并集，结果是把子视图的dirty区域转化为父容器的dirty区域

dirty.union(0, 0, mRight - mLeft, mBottom - mTop);

}

final int left = mLeft;

final int top = mTop;

if ((mGroupFlags & FLAG\_CLIP\_CHILDREN) == FLAG\_CLIP\_CHILDREN) {

if (!dirty.intersect(0, 0, mRight - left, mBottom - top)) {

dirty.setEmpty();

}

}

mPrivateFlags &= ~PFLAG\_DRAWING\_CACHE\_VALID;

//记录当前视图的mLeft和mTop值，在下一次循环中会把当前值再向父容器的坐标转化

location[CHILD\_LEFT\_INDEX] = left;

location[CHILD\_TOP\_INDEX] = top;

if (mLayerType != LAYER\_TYPE\_NONE) {

mPrivateFlags |= PFLAG\_INVALIDATED;

}

//返回当前视图的父容器

return mParent;

}

...

}

return null;}

可以看出，这个方法做的工作主要有：调用offset方法，把当前dirty区域的坐标转化为父容器中的坐标，接着调用union方法，把子dirty区域与父容器的区域求并集，换句话说，dirty区域变成父容器区域。最后返回当前视图的父容器，以便进行下一次循环。

##### 1.1.3 ViewRootImpl

@Overridepublic ViewParent invalidateChildInParent(int[] location, Rect dirty) {

checkThread();

if (DEBUG\_DRAW) Log.v(TAG, "Invalidate child: " + dirty);

if (dirty == null) {

invalidate();

return null;

} else if (dirty.isEmpty() && !mIsAnimating) {

return null;

}

if (mCurScrollY != 0 || mTranslator != null) {

mTempRect.set(dirty);

dirty = mTempRect;

if (mCurScrollY != 0) {

dirty.offset(0, -mCurScrollY);

}

if (mTranslator != null) {

mTranslator.translateRectInAppWindowToScreen(dirty);

}

if (mAttachInfo.mScalingRequired) {

dirty.inset(-1, -1);

}

}

final Rect localDirty = mDirty;

if (!localDirty.isEmpty() && !localDirty.contains(dirty)) {

mAttachInfo.mSetIgnoreDirtyState = true;

mAttachInfo.mIgnoreDirtyState = true;

}

// Add the new dirty rect to the current one

localDirty.union(dirty.left, dirty.top, dirty.right, dirty.bottom);

// Intersect with the bounds of the window to skip

// updates that lie outside of the visible region

final float appScale = mAttachInfo.mApplicationScale;

final boolean intersected = localDirty.intersect(0, 0,

(int) (mWidth \* appScale + 0.5f), (int) (mHeight \* appScale + 0.5f));

if (!intersected) {

localDirty.setEmpty();

}

if (!mWillDrawSoon && (intersected || mIsAnimating)) {

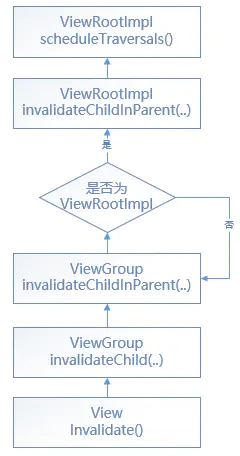
scheduleTraversals();

}

return null;}

可以看出，该方法进行了offset操作对坐标的调整，然后把dirty区域的信息保存在mDirty中，最后调用了scheduleTraversals方法，触发View的工作流程，由于没有添加measure和layout的标记位，因此measure、layout流程不会执行，而是直接从draw流程开始。

#### 1.2 流程总结



当子View调用了invalidate方法后，会为该View添加一个标记位，同时不断向父容器请求刷新，父容器通过计算得出自身需要重绘的区域，直到传递到ViewRootImpl中，最终触发scheduleTraversals方法，进行View树重绘流程(只绘制需要重绘的视图)。  
注：关于scheduleTraversals参看【View:刷新机制】

### 2 postInvalidate方法

这个方法与invalidate方法的作用是一样的，都是使View树重绘，但两者的使用条件不同，postInvalidate是在非UI线程中调用，invalidate则是在UI线程中调用。

#### 2.1 View

public void postInvalidate() {

postInvalidateDelayed(0);}

public void postInvalidateDelayed(long delayMilliseconds) {

// We try only with the AttachInfo because there's no point in invalidating

// if we are not attached to our window

final AttachInfo attachInfo = mAttachInfo;

if (attachInfo != null) {

attachInfo.mViewRootImpl.dispatchInvalidateDelayed(this, delayMilliseconds);

}}

由以上代码可以看出，只有attachInfo不为null的时候才会继续执行，即只有确保视图被添加到窗口的时候才会通知view树重绘，因为这是一个异步方法，如果在视图还未被添加到窗口就通知重绘的话会出现错误，所以这样要做一下判断。接着调用了ViewRootImpl#dispatchInvalidateDelayed方法。

#### 2.2 ViewRootImpl

public void dispatchInvalidateDelayed(View view, long delayMilliseconds) {

Message msg = mHandler.obtainMessage(MSG\_INVALIDATE, view);

mHandler.sendMessageDelayed(msg, delayMilliseconds);}

这里用了Handler，发送了一个异步消息到主线程，显然这里发送的是MSG\_INVALIDATE，即通知主线程刷新视图，具体的实现逻辑我们可以看看该mHandler的实现：

final ViewRootHandler mHandler = new ViewRootHandler();

final class ViewRootHandler extends Handler {

@Override

public String getMessageName(Message message) {

....

}

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_INVALIDATE:

((View) msg.obj).invalidate();

break;

...

}

}}

可以看出，参数message传递过来的正是View视图的实例，然后直接调用了invalidate方法，然后继续invalidate流程。

### 3 requestLayout方法

#### 3.1 View

public void requestLayout() {

if (mMeasureCache != null) mMeasureCache.clear();

if (mAttachInfo != null && mAttachInfo.mViewRequestingLayout == null) {

// Only trigger request-during-layout logic if this is the view requesting it,

// not the views in its parent hierarchy

ViewRootImpl viewRoot = getViewRootImpl();

if (viewRoot != null && viewRoot.isInLayout()) {

if (!viewRoot.requestLayoutDuringLayout(this)) {

return;

}

}

mAttachInfo.mViewRequestingLayout = this;

}

//为当前view设置标记位 PFLAG\_FORCE\_LAYOUT

mPrivateFlags |= PFLAG\_FORCE\_LAYOUT;

mPrivateFlags |= PFLAG\_INVALIDATED;

if (mParent != null && !mParent.isLayoutRequested()) {

//向父容器请求布局

mParent.requestLayout();

}

if (mAttachInfo != null && mAttachInfo.mViewRequestingLayout == this) {

mAttachInfo.mViewRequestingLayout = null;

}}

首先先判断当前View树是否正在布局流程，接着为当前子View设置标记位，该标记位的作用就是标记了当前的View是需要进行重新布局的，接着调用mParent.requestLayout方法。由于ViewGroup没有重写requestLayout方法，所以会一直循环遍历直到ViewRoot。

#### 3.2 ViewRootImpl

@Overridepublic void requestLayout() {

if (!mHandlingLayoutInLayoutRequest) {

checkThread();

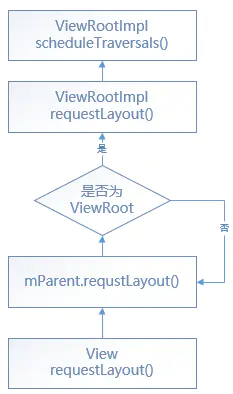
mLayoutRequested = true;

scheduleTraversals();

}}

检查主线程，调用scheduleTraversals()方法，触发View的重绘、布局操作。

#### 3.3 requestLayout流程



### 4 invalidate、requestLayout比较

| **方法** | **过程** | **作用** |
| --- | --- | --- |
| invalidate | 从调用处层层向上调用，计算脏区域，直到ViewRootImpl的invalidateChildInParent(..)，在其中调用scheduleTraversals() | 不会执行测量和布局流程，对View树进行重绘； |
| requestLayout | 从调用处层层向上调用，标记PFLAG\_FORCE\_LAYOUT标识，直到ViewRootImpl的requestLayout()，在其中调用scheduleTraversals() | 会进行测量、布局流程，不一定会触发绘制流程？？（存疑，知道的告知下） |

* 若仅View的显示内容发生改变且新显示内容不影响View的大小、位置，则只需调用invalidate方法；
* 若View宽高、位置发生改变且显示内容不变，只需调用requestLayout方法；