Andorid提供三种动画，分别为逐帧动画、补间动画、属性动画，下面逐一介绍各个动画的实现流程。

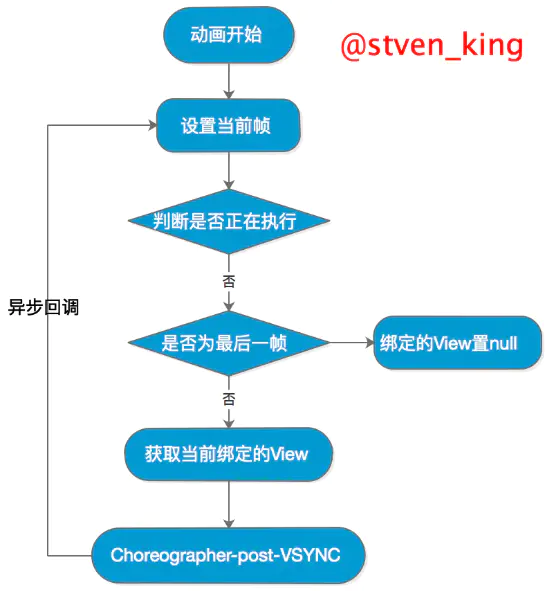
#### 1 逐帧动画 (Frame Animation)

原理：使用了Choreographer机制。

AnimationDrawable animationDrawable = (AnimationDrawable) image.getDrawable();

animationDrawable.start();

启动帧动画，下面观察源码分析其执行过程：



##### 1.1 AnimationDrawable启动动画

public class AnimationDrawable extends DrawableContainer implements Runnable, Animatable {

/\*\*\*代码部分省略\*\*\*/

@Override

public void start() {

mAnimating = true;

if (!isRunning()) {

// Start from 0th frame.

setFrame(0, false, mAnimationState.getChildCount() > 1

|| !mAnimationState.mOneShot);

}

}

//设置当前展示第几帧

private void setFrame(int frame, boolean unschedule, boolean animate) {

if (frame >= mAnimationState.getChildCount()) {

return;

}

mAnimating = animate;

mCurFrame = frame;

selectDrawable(frame);

//如果取消下一帧任务，或者这已经是当前最后一帧，则取消当帧动画任务

if (unschedule || animate) {

unscheduleSelf(this);

}

if (animate) {

// Unscheduling may have clobbered these values; restore them

mCurFrame = frame;

mRunning = true;

scheduleSelf(this, SystemClock.uptimeMillis() + mAnimationState.mDurations[frame]);

}

}

//安排动画绘制任务

public void scheduleSelf(Runnable what, long when) {

//该Callback是当前AnimationDrawable绑定的View

final Callback callback = getCallback();

//判断当前绑定的View是否被销毁

if (callback != null) {

callback.scheduleDrawable(this, what, when);

}

}}

##### **1.2 View.scheduleDrawable(xxx)交付Choreographer**

public class View implements Drawable.Callback, KeyEvent.Callback,

AccessibilityEventSource {

/\*\*\*部分代码省略\*\*\*/

@Override

public void scheduleDrawable(Drawable who, Runnable what, long when) {

if (verifyDrawable(who) && what != null) {

final long delay = when - SystemClock.uptimeMillis();

if (mAttachInfo != null) {

//请求Vsync信号同步

mAttachInfo.mViewRootImpl.mChoreographer.postCallbackDelayed(

Choreographer.CALLBACK\_ANIMATION, what, who,

Choreographer.subtractFrameDelay(delay));

} else {

ViewRootImpl.getRunQueue().postDelayed(what, delay);

}

}

}}

##### **1.3 Choreographer回调AnimationDrawable.run() 方法**

public class AnimationDrawable extends DrawableContainer implements Runnable, Animatable {

/\*\*\*代码部分省略\*\*\*/

//Choreographer的Vsync同步回调

@Override

public void run() {

nextFrame(false);

}

//继续执行下一帧动画

private void nextFrame(boolean unschedule) {

int nextFrame = mCurFrame + 1;

final int numFrames = mAnimationState.getChildCount();

final boolean isLastFrame = mAnimationState.mOneShot && nextFrame >= (numFrames - 1);

// Loop if necessary. One-shot animations should never hit this case.

if (!mAnimationState.mOneShot && nextFrame >= numFrames) {

nextFrame = 0;

}

//新一轮的循环又开始

setFrame(nextFrame, unschedule, !isLastFrame);

}}

#### 2 补间动画 (Tween Animation)

原理：在绘制的过程中，尝试获取动画在当前时刻的变换，然后应用到view的绘制中。

Animation translateAnimation = new TranslateAnimation(0, 100, 0, 0);

translateAnimation.setDuration(500);

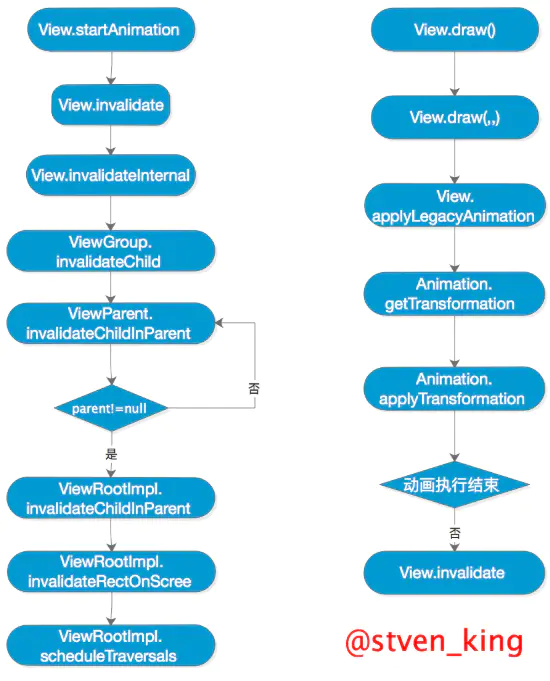
translateAnimation.setInterpolator(new AccelerateInterpolator());

translateAnimation.setFillAfter(true);//设置动画结束后保持当前的位置（即不返回到动画开始前的位置）

imageView.startAnimation(translateAnimation);

上述是一个imageView在x方向移动的动画，可以看到它符合上面所说的四个部分：

* 做什么：沿ｘ方向移动100
* 如何做：直线且不断加速
* 开始时间：startAnimtion调用的那一刻
* 结束时间: 开始时间+500

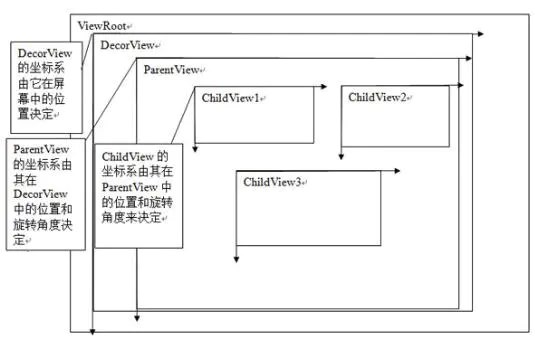


Animation产生的动画数据实际并不是应用在View本身的，而是应用在RenderNode或者Canvas上的，这就是为什么Animation不会改变View的属性的根本所在。另一方面，我们知道Animation仅在View被绘制的时候才能发挥自己的价值，这也是为什么插间动画被放在Android.view包内。

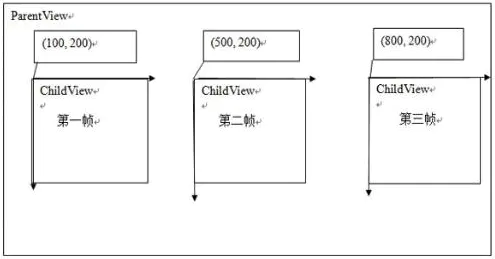
##### **2.1 补间动画框架原理**

###### 2.1.1 原理描述

当一个 ChildView 要重画时，它会调用其成员函数 invalidate() 函数将通知其 ParentView 这个 ChildView 要重画，这个过程一直向上遍历到 ViewRoot，当 ViewRoot 收到这个通知后就会调到提到的 ViewRoot 中的 draw 函数从而完成绘制。View::onDraw() 有一个画布参数 Canvas, 画布顾名思义就是画东西的地方，Android 会为每一个 View 设置好画布，View 就可以调用 Canvas 的方法，比如：drawText, drawBitmap, drawPath 等等去画内容。每一个 ChildView 的画布是由其 ParentView 设置的，ParentView 根据 ChildView 在其内部的布局来调整 Canvas，其中画布的属性之一就是定义和 ChildView 相关的坐标系，默认是横轴为 X 轴，从左至右，值逐渐增大，竖轴为 Y 轴，从上至下，值逐渐增大 , 见下图 :



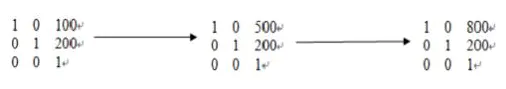
Android 动画就是通过 ParentView 来不断调整 ChildView 的画布坐标系来实现的，下面以平移动画来做示例：假设在动画开始时 ChildView 在 ParentView 中的初始位置在 (100,200) 处，这时 ParentView 会根据这个坐标来设置 ChildView 的画布，在 ParentView 的 dispatchDraw 中它发现 ChildView 有一个平移动画，而且当前的平移位置是 (100, 200)，于是它通过调用画布的函数 traslate(100, 200) 来告诉 ChildView 在这个位置开始画，这就是动画的第一帧。如果 ParentView 发现 ChildView 有动画，就会不断的调用 invalidate() 这个函数，这样就会导致自己会不断的重画，就会不断的调用 dispatchDraw 这个函数，这样就产生了动画的后续帧，当再次进入 dispatchDraw 时，ParentView 根据平移动画产生出第二帧的平移位置 (500, 200)，然后继续执行上述操作，然后产生第三帧，第四帧，直到动画播完。



以上过程涉及两个重要的类型，Animation 和 Transformation：

Animation  
Animation 中主要定义了动画的一些属性比如开始时间、持续时间、是否重复播放等，这个类主要有两个重要的函数：getTransformation 和 applyTransformation，在 getTransformation 中 Animation 会根据动画的属性来产生一系列的差值点，然后将这些差值点传给 applyTransformation，这个函数将根据这些点来生成不同的 Transformation。

Transformation  
Transformation 中包含一个矩阵和 alpha 值，矩阵是用来做平移、旋转和缩放动画的，而 alpha 值是用来做 alpha 动画的（简单理解的话，alpha 动画相当于不断变换透明度或颜色来实现动画）。以上面的平移矩阵为例子，当调用 dispatchDraw 时会调用 getTransformation 来得到当前的 Transformation，这个 Transformation 中的矩阵如下：



###### 2.1.2 补间动画框架意义

Android 的动画框架把动画的播放 / 绘制交给父 View 去处理而不是让子 View 本身去绘制，这种从更高的层次上去控制的方式便于把动画机制做成一个易用的框架。如果用户要在某个 view 中使用动画，只需要在 xml 描述文件或代码中指定就可以了，从而把动画的实现和 View 本身内容的绘制（象 TextView 里面的文字显示）分离开了，起到了减少耦合和提高易用性的效果。

##### 2.2 代码层面分析

###### 2.2.1 startAnimation启动动画

View

public class View implements Drawable.Callback, KeyEvent.Callback,

AccessibilityEventSource {

//部分代码省略

public void startAnimation(Animation animation) {

animation.setStartTime(Animation.START\_ON\_FIRST\_FRAME);

setAnimation(animation);

invalidateParentCaches();

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;

}

if (skipInvalidate()) {

return;

}

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)) {

//部分代码省略

// Propagate the damage rectangle to the parent view.

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);

//执行ViewParent的invalidateChild方法

p.invalidateChild(this, damage);

}

//部分代码省略

}

}}

ViewGroup

public abstract class ViewGroup extends View implements ViewParent, ViewManager {

/\*\*\*部分代码省略\*\*\*/

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

ViewParent parent = this;

final AttachInfo attachInfo = mAttachInfo;

if (attachInfo != null) {

/\*\*\*部分代码省略\*\*\*/

do {

/\*\*\*部分代码省略\*\*\*/

//向顶部的View便利找到根View，即：ViewRootImpl

//执行ViewRootImpl的invalidateChildInParent方法

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));

}

}

/\*\*\*部分代码省略\*\*\*/

} while (parent != null);

}

}}

这里调用了函数invalidateChildInParent()，需要注意的是这里这个函数的实现有两个，一个是ViewGroup中，而另一个是ViewRootImpl。

ViewRootImpl

public final class ViewRootImpl implements ViewParent,

View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {

/\*\*\*部分代码省略\*\*\*/

@Override

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

checkThread();

/\*\*\*部分代码省略\*\*\*/

invalidateRectOnScreen(dirty);

return null;

}

private void invalidateRectOnScreen(Rect dirty) {

/\*\*\*部分代码省略\*\*\*/

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

//开始View的绘制任务

scheduleTraversals();

}

}}

函数scheduleTraversals()的逻辑其实是执行一个Runnable，而这个Runnable其实就是去执行函数doTraversal(),而函数doTraversal()会调用performTraversals(),到这里我们发现它开始重绘了。总体来说就是动画的执行会导致整个View Tree重绘，但是Android内部有一些优化，比如一张图片做移动，我们不需要真正的去重新绘制，Android内部提供缓存机制，不会显示的再调用onDraw(canvas)函数。

###### 2.2.2 draw-视图绘制

public class View implements Drawable.Callback, KeyEvent.Callback,

AccessibilityEventSource {

//部分代码省略

public void draw(Canvas canvas) {

/\*\*\*部分代码省略\*\*\*/

//如果有子 View(DecorView当然有子View)，就会调用dispatchDraw() 将绘制事件通知给子 View。

//ViewGroup 重写了 dispatchDraw()，调用了 drawChild()

//drawChild() 调用了子 View 的 draw(Canvas, ViewGroup, long)

}

boolean draw(Canvas canvas, ViewGroup parent, long drawingTime) {

final boolean hardwareAcceleratedCanvas = canvas.isHardwareAccelerated();

/\*\*\*部分代码省略\*\*\*/

Transformation transformToApply = null;

boolean concatMatrix = false;

final boolean scalingRequired = mAttachInfo != null && mAttachInfo.mScalingRequired;

final Animation a = getAnimation();

if (a != null) {

more = applyLegacyAnimation(parent, drawingTime, a, scalingRequired);

concatMatrix = a.willChangeTransformationMatrix();

if (concatMatrix) {

mPrivateFlags3 |= PFLAG3\_VIEW\_IS\_ANIMATING\_TRANSFORM;

}

transformToApply = parent.getChildTransformation();

} else {

/\*\*\*部分代码省略\*\*\*/

}

/\*\*\*部分代码省略\*\*\*/

// 动画数据应用在RenderNode或者Canvas上的!!!!

if (transformToApply != null) {

if (concatMatrix) {

if (drawingWithRenderNode) {

// 应用动画数据

renderNode.setAnimationMatrix(transformToApply.getMatrix());

} else {

canvas.translate(-transX, -transY);

// 应用动画数据

canvas.concat(transformToApply.getMatrix());

canvas.translate(transX, transY);

}

parent.mGroupFlags |= ViewGroup.FLAG\_CLEAR\_TRANSFORMATION;

}

float transformAlpha = transformToApply.getAlpha();

if (transformAlpha < 1) {

// 应用动画数据

alpha \*= transformAlpha;

parent.mGroupFlags |= ViewGroup.FLAG\_CLEAR\_TRANSFORMATION;

}

}

}

private boolean applyLegacyAnimation(ViewGroup parent, long drawingTime,

Animation a, boolean scalingRequired) {

/\*\*\*部分代码省略\*\*\*/

//绘制动画的当前帧，并获取当前动画的状态（是否继续运行）

boolean more = a.getTransformation(drawingTime, t, 1f);

if (scalingRequired && mAttachInfo.mApplicationScale != 1f) {

if (parent.mInvalidationTransformation == null) {

parent.mInvalidationTransformation = new Transformation();

}

invalidationTransform = parent.mInvalidationTransformation;

a.getTransformation(drawingTime, invalidationTransform, 1f);

} else {

invalidationTransform = t;

}

//如果动画没有结果

if (more) {

if (!a.willChangeBounds()) {

if ((flags & (ViewGroup.FLAG\_OPTIMIZE\_INVALIDATE | ViewGroup.FLAG\_ANIMATION\_DONE)) ==

ViewGroup.FLAG\_OPTIMIZE\_INVALIDATE) {

parent.mGroupFlags |= ViewGroup.FLAG\_INVALIDATE\_REQUIRED;

} else if ((flags & ViewGroup.FLAG\_INVALIDATE\_REQUIRED) == 0) {

// The child need to draw an animation, potentially offscreen, so

// make sure we do not cancel invalidate requests

parent.mPrivateFlags |= PFLAG\_DRAW\_ANIMATION;

//进行绘制

parent.invalidate(mLeft, mTop, mRight, mBottom);

}

} else {

/\*\*\*部分代码省略\*\*\*/

//进行绘制

parent.invalidate(left, top, left + (int) (region.width() + .5f),

top + (int) (region.height() + .5f));

}

}

return more;

}}

View.draw(Canvas)  
—> ViewGroup.dispatchDraw(Canvas)  
—> ViewGroup.drawChild(Canvas, View, long)  
—> View.draw(Canvas, ViewGroup, long)  
—> View.applyLegacyAnimation(ViewGroup, long, Animation, boolean)

###### 2.2.3 计算动画进度，进行矩阵变换

目的：根据当前绘制事件生成Animation中对应帧的动画数据。

public abstract class Animation implements Cloneable {

/\*\*\*部分代码省略\*\*\*/

public boolean getTransformation(long currentTime, Transformation outTransformation) {

/\*\*\*部分代码省略\*\*\*/

//执行时间是否过期

final boolean expired = normalizedTime >= 1.0f;

mMore = !expired;

//动画进度为0.0~1.0之间

if (!mFillEnabled) normalizedTime = Math.max(Math.min(normalizedTime, 1.0f), 0.0f);

if ((normalizedTime >= 0.0f || mFillBefore) && (normalizedTime <= 1.0f || mFillAfter)) {

/\*\*\*部分代码省略\*\*\*/

//插值器计算动画执行进度

final float interpolatedTime = mInterpolator.getInterpolation(normalizedTime);

//真正的动画效果代码执行处(通过矩阵变化)

applyTransformation(interpolatedTime, outTransformation);

}

//如果动画绘制完成

if (expired) {

//判断动画是否需要继续循环

if (mRepeatCount == mRepeated) {

if (!mEnded) {

mEnded = true;

guard.close();

fireAnimationEnd();

}

} else {

if (mRepeatCount > 0) {

mRepeated++;

}

if (mRepeatMode == REVERSE) {

mCycleFlip = !mCycleFlip;

}

mStartTime = -1;

mMore = true;

fireAnimationRepeat();

}

}

if (!mMore && mOneMoreTime) {

mOneMoreTime = false;

return true;

}

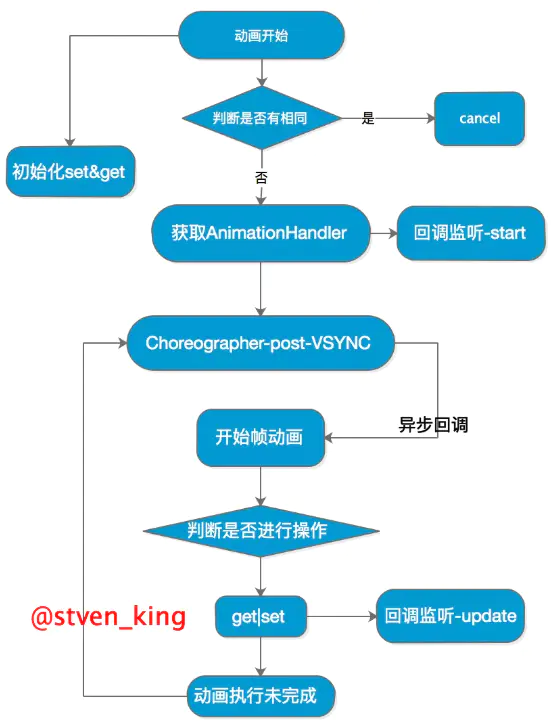
return mMore;

}}

说明：Animation的运行依赖Android本身的机制回调（每帧都得回调getTransformation和applyTransformation）无法自身进行运算计算fraction，并且可参与运算的只有Transformation对象里的alpha和matrix，所以Animation只能实现简单的Alpha，Scale，Translate，Rotate变换效果。

#### 3 属性动画 (Property Animation)

上面分析的Animation受限于Android本身的回调，只能实现Alpha，Scale，Translate，Rotate的变换。而Animator没有此限制，它不依赖于Android本身的机制回调，但是它意依赖于Looper的Thread。其运行流程如下所示：



##### 3.1 启动动画

* ValueAnimator.start  
  AnimationHandler并不是Handler，它是个Runnable。

public class ValueAnimator extends Animator {

/\*\*\*部分代码省略\*\*\*/

protected static ThreadLocal<AnimationHandler> sAnimationHandler =

new ThreadLocal<AnimationHandler>();

//保证每个线程有且只有一个AnimationHandler

private static AnimationHandler getOrCreateAnimationHandler() {

AnimationHandler handler = sAnimationHandler.get();

if (handler == null) {

handler = new AnimationHandler();

sAnimationHandler.set(handler);

}

return handler;

}

@Override

public void start() {

start(false);

}

private void start(boolean playBackwards) {

if (Looper.myLooper() == null) {

throw new AndroidRuntimeException("Animators may only be run on Looper threads");

}

/\*\*\*部分代码省略\*\*\*/

//创建或者获取animationHandler实例

AnimationHandler animationHandler = getOrCreateAnimationHandler();

animationHandler.mPendingAnimations.add(this);

if (mStartDelay == 0) {

// This sets the initial value of the animation, prior to actually starting it running

if (prevPlayingState != SEEKED) {

setCurrentPlayTime(0);

}

mPlayingState = STOPPED;

mRunning = true;

//回调监听器，通知动画开始

notifyStartListeners();

}

//开始动画

animationHandler.start();

}

//回调监听器，通知动画开始

private void notifyStartListeners() {

if (mListeners != null && !mStartListenersCalled) {

ArrayList<AnimatorListener> tmpListeners =

(ArrayList<AnimatorListener>) mListeners.clone();

int numListeners = tmpListeners.size();

for (int i = 0; i < numListeners; ++i) {

tmpListeners.get(i).onAnimationStart(this);

}

}

mStartListenersCalled = true;

}

public void setCurrentPlayTime(long playTime) {

float fraction = mUnscaledDuration > 0 ? (float) playTime / mUnscaledDuration : 1;

setCurrentFraction(fraction);

}

public void setCurrentFraction(float fraction) {

//初始化动画

initAnimation();

if (fraction < 0) {

fraction = 0;

}

/\*\*\*部分代码省略\*\*\*/

}}

* ObjectAnimator.start

public final class ObjectAnimator extends ValueAnimator {

/\*\*\*部分代码省略\*\*\*/

@Override

public void start() {

//首先依次判断了当前动画、等待的动画、延迟的动画中是否有和当前动画相同的动画

//若有就把相同的动画取消掉

// See if any of the current active/pending animators need to be canceled

AnimationHandler handler = sAnimationHandler.get();

if (handler != null) {

int numAnims = handler.mAnimations.size();

for (int i = numAnims - 1; i >= 0; i--) {

if (handler.mAnimations.get(i) instanceof ObjectAnimator) {

ObjectAnimator anim = (ObjectAnimator) handler.mAnimations.get(i);

if (anim.mAutoCancel && hasSameTargetAndProperties(anim)) {

anim.cancel();

}

}

}

/\*\*\*部分代码省略\*\*\*/

}

/\*\*\*部分代码省略\*\*\*/

//然后调用ValueAnimator.start()方法

super.start();

}}

* initAnimation

public final class ObjectAnimator extends ValueAnimator {

/\*\*\*部分代码省略\*\*\*/

@Override

void initAnimation() {

if (!mInitialized) {

// mValueType may change due to setter/getter setup; do this before calling super.init(),

// which uses mValueType to set up the default type evaluator.

final Object target = getTarget();

if (target != null) {

final int numValues = mValues.length;

for (int i = 0; i < numValues; ++i) {

mValues[i].setupSetterAndGetter(target);

}

}

super.initAnimation();

}

}}

* setupSetterAndGetter

public final class ObjectAnimator extends ValueAnimator {

/\*\*\*部分代码省略\*\*\*/

void setupSetterAndGetter(Object target) {

mKeyframes.invalidateCache();

if (mProperty != null) {

/\*\*\*部分代码省略\*\*\*/

}

// We can't just say 'else' here because the catch statement sets mProperty to null.

if (mProperty == null) {

Class targetClass = target.getClass();

if (mSetter == null) {

//初始化mSetter

setupSetter(targetClass);

}

/\*\*\*部分代码省略\*\*\*/

}

}

//初始化mSetter用于以后反射执行get、set操作

void setupSetter(Class targetClass) {

Class<?> propertyType = mConverter == null ? mValueType : mConverter.getTargetType();

mSetter = setupSetterOrGetter(targetClass, sSetterPropertyMap, "set", propertyType);

}}

##### 3.2 AnimationHandler.start

public class ValueAnimator extends Animator {

/\*\*\*部分代码省略\*\*\*/

protected static class AnimationHandler implements Runnable {

/\*\*\*部分代码省略\*\*\*/

//开始动画

public void start() {

scheduleAnimation();

}

//发送VSYNC信号回调请求

private void scheduleAnimation() {

if (!mAnimationScheduled) {

mChoreographer.postCallback(Choreographer.CALLBACK\_ANIMATION, this, null);

mAnimationScheduled = true;

}

}

// Called by the Choreographer.

//Choreographer的VSYNC信号回调

@Override

public void run() {

mAnimationScheduled = false;

doAnimationFrame(mChoreographer.getFrameTime());

}

private void doAnimationFrame(long frameTime) {

/\*\*\*部分代码省略\*\*\*/

// Now process all active animations. The return value from animationFrame()

// tells the handler whether it should now be ended

int numAnims = mAnimations.size();

for (int i = 0; i < numAnims; ++i) {

mTmpAnimations.add(mAnimations.get(i));

}

for (int i = 0; i < numAnims; ++i) {

ValueAnimator anim = mTmpAnimations.get(i);

//执行动画

//doAnimationFrame方法返回ture,则该动画添加在mEndingAnims队列中进行end操作

if (mAnimations.contains(anim) && anim.doAnimationFrame(frameTime)) {

mEndingAnims.add(anim);

}

}

/\*\*\*部分代码省略\*\*\*/

//循环执行，直到endAnimation将mAnimations置空

if (!mAnimations.isEmpty() || !mDelayedAnims.isEmpty()) {

scheduleAnimation();

}

}

}}

##### 3.3 ValueAnimator.doAnimationFrame

public class ValueAnimator extends Animator {

/\*\*\*部分代码省略\*\*\*/

final boolean doAnimationFrame(long frameTime) {

/\*\*\*部分代码省略\*\*\*/

return animationFrame(currentTime);

}

boolean animationFrame(long currentTime) {

boolean done = false;

switch (mPlayingState) {

case RUNNING:

case SEEKED:

/\*\*\*部分代码省略\*\*\*/

if (fraction >= 1f) {

//mCurrentIteration是否等于mRepeatCount

if (mCurrentIteration < mRepeatCount || mRepeatCount == INFINITE) {

// Time to repeat

/\*\*\*部分代码省略\*\*\*/

} else {

//执行完这次，该动画结束

done = true;

fraction = Math.min(fraction, 1.0f);

}

}

if (mPlayingBackwards) {

fraction = 1f - fraction;

}

//设置View的属性值

animateValue(fraction);

break;

}

return done;

}}

##### 3.4 animateValue

* ValueAnimator.animateValue

public class ValueAnimator extends Animator {

/\*\*\*部分代码省略\*\*\*/

void animateValue(float fraction) {

fraction = mInterpolator.getInterpolation(fraction);

mCurrentFraction = fraction;

int numValues = mValues.length;

for (int i = 0; i < numValues; ++i) {

//PropertyValuesHolder.calculateValue就是计算每帧动画所对应的值

mValues[i].calculateValue(fraction);

}

if (mUpdateListeners != null) {

int numListeners = mUpdateListeners.size();

for (int i = 0; i < numListeners; ++i) {

//属性值得改变的回调

mUpdateListeners.get(i).onAnimationUpdate(this);

}

}

}}

* ObjectAnimator.animateValue

public final class ObjectAnimator extends ValueAnimator {

/\*\*\*部分代码省略\*\*\*/

@Override

void animateValue(float fraction) {

final Object target = getTarget();

if (mTarget != null && target == null) {

// We lost the target reference, cancel and clean up.

cancel();

return;

}

//ValueAnimator.animateValue方法

super.animateValue(fraction);

int numValues = mValues.length;

for (int i = 0; i < numValues; ++i) {

//设置target的属性值，进行View的移动，产生动画

mValues[i].setAnimatedValue(target);

}

}}

##### 3.5 内存泄露

ValueAnimator.AnimationHandler.doAnimationFrame 每次执行完动画（如果动画没有结束），都在再一次请求Vsync同步信号回调给自己。Choreographer 的回调都post进入了当前线程的looper队列中。mRepeatCount 无穷大，会导致该循环会一直执行下去，即使关闭当前的页面也不会停止。