Drawable+Animator，将优雅进行到底

leobert-lan 郭霖 2022-10-28 08:00 发表于江苏

/ 今日科技快讯 /

近日Facebook母公司Meta发布了截至9月30日的2022年第三季度财报。财报显示，Meta第三季度营收为277亿美元，相比下滑4%；净利润为44亿美元，同比下滑52%；摊薄后每股收益为1.64美元，同比下滑49%。由于第三季度业绩不及市场预期，Meta股价在盘后交易中暴跌近20%。

/ 作者简介 /

明天周六啦，大家好好休息，我们下周再见！

本篇文章来自leobert-lan的投稿，文章主要分享了Drawable的动画原理分析，相信会对大家有所帮助！同时也感谢作者贡献的精彩文章。

leobert-lan的博客地址：

> https://juejin.cn/user/2066737589654327/posts

/ 前言 /

之前发过 **重新认识Drawable** 一文。文中，我们最终以该方案实现了 "自定义一个动画Drawable" ：unscheduleSelf() / scheduleSelf() 机制 停止回调/设置定时回调 + invalidateSelf() 机制进行刷新绘制；方案的本质是 在预设时间点绘制关键帧 。仔细观察后不难发现问题：**效果并不顺滑** 。

彼时，文章的主旨为重新认识Drawable，并未对此展开讨论并进一步优化。本篇文章作为迟来的续集，将会 对问题展开讨论、探索优化方案、追究原理、并进一步拓宽思路。

/ 问题本质 /

上文已经提到，我们通过 unscheduleSelf() / scheduleSelf() 机制 停止回调/设置定时回调，重新绘制关键帧。那么 scheduleSelf() 的本质又是什么？

阅读代码可知，源码中通过接口回调的设计，将功能的实现剥离：

```java
class Drawable {
  public void scheduleSelf(@NonNull Runnable what, long when) {
    final Callback callback = getCallback();
    if (callback != null) {
      callback.scheduleDrawable(this, what, when);
    }
  }

  public final void setCallback(@Nullable Callback cb) {
    mCallback = cb != null ? new WeakReference<>(cb) : null;
  }

  @Nullable
  public Callback getCallback() {
    return mCallback != null ? mCallback.get() : null;
  }

  public interface Callback {
    void invalidateDrawable(@NonNull Drawable who);

    void scheduleDrawable(@NonNull Drawable who, @NonNull Runnable what, long when);

    void unscheduleDrawable(@NonNull Drawable who, @NonNull Runnable what);
  }
}
```

继续寻找 Callback 实现类：**重点关注 scheduleDrawable 即可**

```java
public class View implements Drawable.Callback {
  public void invalidateDrawable(@NonNull Drawable drawable) {
    if (verifyDrawable(drawable)) {
      final Rect dirty = drawable.getDirtyBounds();
      final int scrollX = mScrollX;
      final int scrollY = mScrollY;

      invalidate(dirty.left + scrollX, dirty.top + scrollY,
          dirty.right + scrollX, dirty.bottom + scrollY);
      rebuildOutline();
    }
  }

  //看这里
  public void scheduleDrawable(@NonNull Drawable who, @NonNull Runnable what, long when) {
    if (verifyDrawable(who) && what != null) {
      final long delay = when - SystemClock.uptimeMillis();
```

```
        if (mAttachInfo != null) {
          mAttachInfo.mViewRootImpl.mChoreographer.postCallbackDelayed(
              Choreographer.CALLBACK_ANIMATION, what, who,
              Choreographer.subtractFrameDelay(delay));
        } else {
          // Postpone the runnable until we know
          // on which thread it needs to run.
          getRunQueue().postDelayed(what, delay);
        }
      }
    }

    public void unscheduleDrawable(@NonNull Drawable who, @NonNull Runnable what) {
      if (verifyDrawable(who) && what != null) {
        if (mAttachInfo != null) {
          mAttachInfo.mViewRootImpl.mChoreographer.removeCallbacks(
              Choreographer.CALLBACK_ANIMATION, what, who);
        }
        getRunQueue().removeCallbacks(what);
      }
    }

    public void unscheduleDrawable(Drawable who) {
      if (mAttachInfo != null && who != null) {
        mAttachInfo.mViewRootImpl.mChoreographer.removeCallbacks(
            Choreographer.CALLBACK_ANIMATION, null, who);
      }
    }

  }
```

简单解释程序逻辑如下：如果 "该Drawable作用于自身" 且 "Runnable非空"，计算回调的delay，如果View已经添加到Window，则交给Choreographer，否则丢入缓存队列。

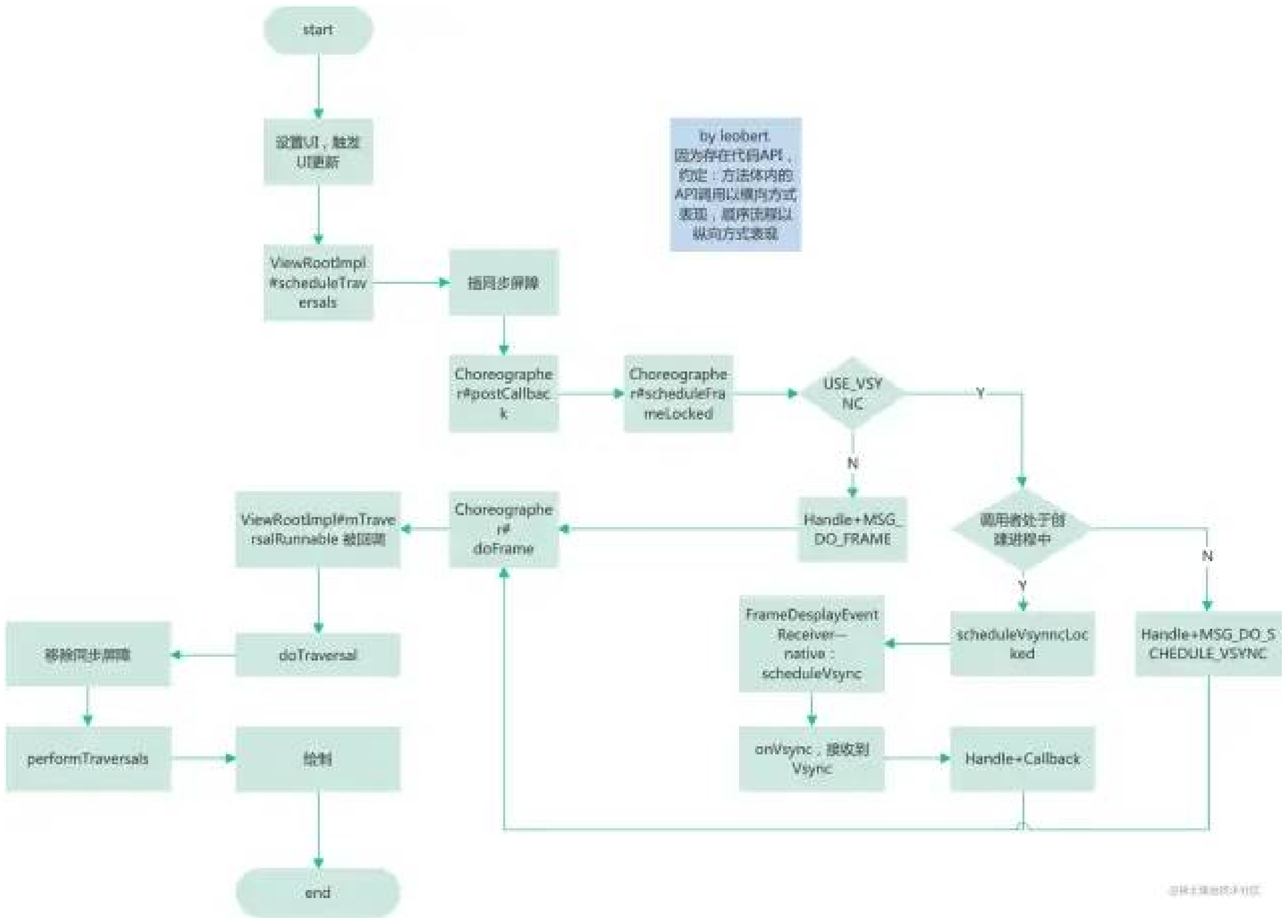
而缓存队列的内容将在View添加到Window时交给 Choreographer

```
  public class View {
    void dispatchAttachedToWindow(AttachInfo info, int visibility) {
      //ignore

      // Transfer all pending runnables.
      if (mRunQueue != null) {
        mRunQueue.executeActions(info.mHandler);
        mRunQueue = null;
      }

      //ignore
    }
  }
```

*读者诸君，如果您熟悉Android的 **屏幕刷新机制** 和 **消息机制** ，一定不会对 Choreographer 感到陌生*

Choreographer 直译为编舞者，暗含了 "编制视图变化效果" 的隐喻，其本质依旧是利用VSync+Handler消息机制。delay Callback的设计存在毫秒级的误差。

作者按：本篇不再展开讨论Android的消息机制，以下仅给出 基于消息机制的界面绘制设计 关键部分流程图：


start
设置UI，触发UI更新
ViewRootImpl#scheduleTraversals
插同步屏障
Choreographer#postCallback
Choreographer#scheduleFrameLocked
USE_VSYNC
Y
N
Handle+MSG_DO_FRAME
调用者处于创建进程中
Y
N
scheduleVsyncLocked
Handle+MSG_DO_SCHEDULE_VSYNC
FrameDesplayEventReceiver—native：scheduleVsync
onVsync，接收到Vsync
Handle+Callback
Choreographer# doFrame
ViewRootImpl#mTraversalRunnable 被回调
doTraversal
移除同步屏障
performTraversals
绘制
end
by leobert
因为存在代码API，约定：方法体内的API调用以横向方式表现，顺序流程以纵向方式表现

结合前面的代码分析，scheduleDrawable 的流程可以参考此图理解。

作者按，虽然仍有差异，但机制一致，可参考理解

验证

在 View 中有一段代码和 scheduleDrawable 高度相似：

```
class View {
  public void postOnAnimationDelayed(Runnable action, long delayMillis) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
      attachInfo.mViewRootImpl.mChoreographer.postCallbackDelayed(
          Choreographer.CALLBACK_ANIMATION, action, null, delayMillis);
    } else {
      // Postpone the runnable until we know
      // on which thread it needs to run.
      getRunQueue().postDelayed(action, delayMillis);
    }
    }
}
```

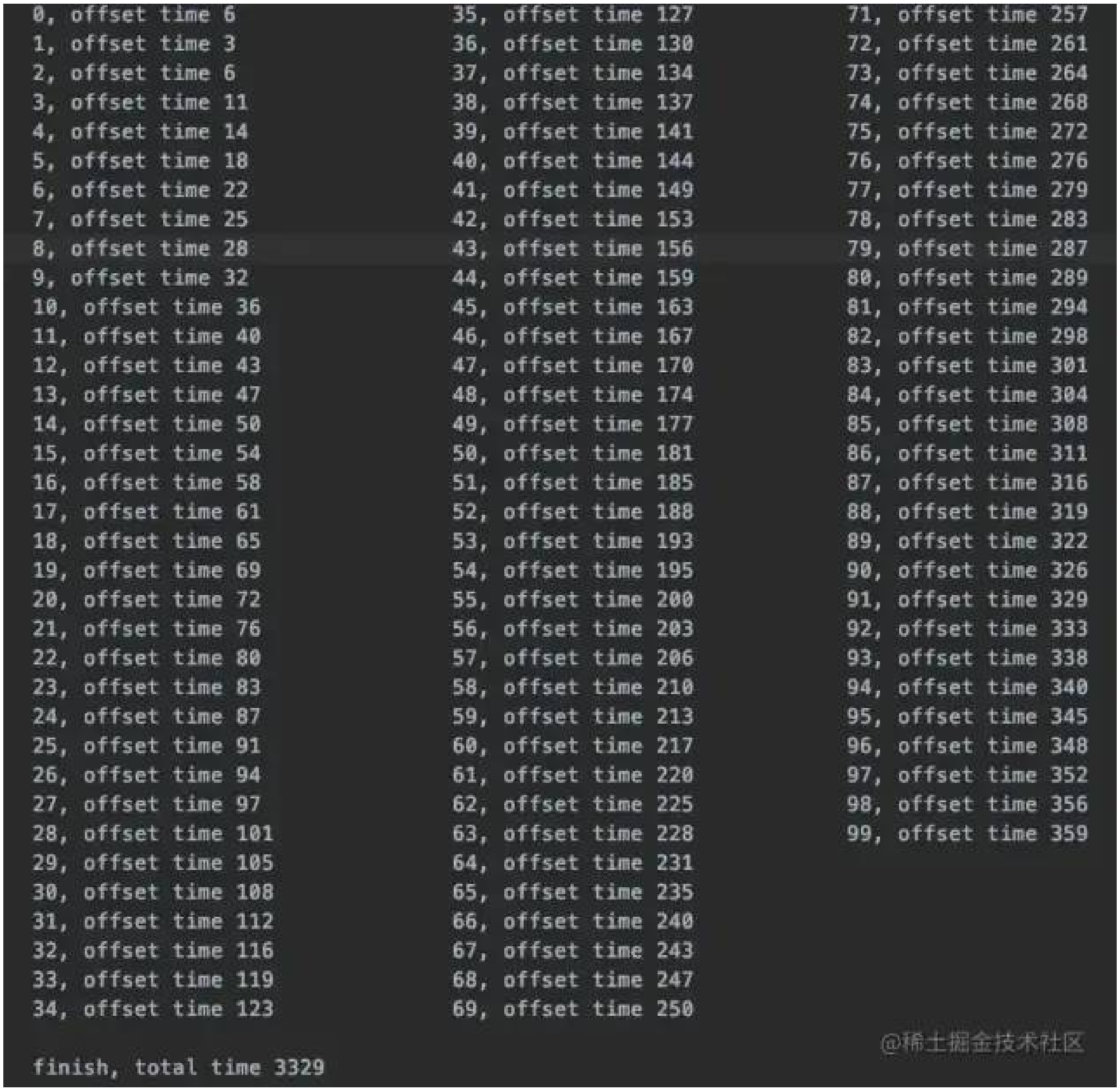
注意：scheduleDrawable 基于执行的目标时间 when，和当前系统时钟计算了delay，又额外调整了delay时间, Choreographer.subtractFrameDelay(delay)，_ 它是隐藏API_

```
public final class Choreographer {
    private static final long DEFAULT_FRAME_DELAY = 10;
    // The number of milliseconds between animation frames.
    private static volatile long sFrameDelay = DEFAULT_FRAME_DELAY;

    public static long subtractFrameDelay(long delayMillis) {
        final long frameDelay = sFrameDelay;
        return delayMillis <= frameDelay ? 0 : delayMillis - frameDelay;
    }
}
```

设计一个简单的验证代码：

```
class Demo {
    //...

    fun test() {
        val btn = findViewById<Button>(R.id.btn)
        var index = 0
        var s = System.currentTimeMillis()

        val action: Runnable = object : Runnable {
            override fun run() {
                Log.e("lmsg", "$index, offset time ${System.currentTimeMillis() - s - index * 30}")
                index++
                if (index < 100) {
                    btn.postOnAnimationDelayed(
                        this,
                        30L - 10L /*hide api:android.view.Choreographer#subtractFrameDelay*/
                    )
                } else {
                    Log.e("lmsg", "finish, total time ${System.currentTimeMillis() - s}")

                }
            }
        }

        btn.setOnClickListener {
            index = 0
            s = System.currentTimeMillis()
            it.postOnAnimationDelayed(action, 0L)
        }
    }
}
```

参考一下结果：注意执行结果不会幂等，但整体表现为超出预期时长



```
0, offset time 6          35, offset time 127       71, offset time 257
1, offset time 3          36, offset time 130       72, offset time 261
2, offset time 6          37, offset time 134       73, offset time 264
3, offset time 11         38, offset time 137       74, offset time 268
4, offset time 14         39, offset time 141       75, offset time 272
5, offset time 18         40, offset time 144       76, offset time 276
6, offset time 22         41, offset time 149       77, offset time 279
7, offset time 25         42, offset time 153       78, offset time 283
8, offset time 28         43, offset time 156       79, offset time 287
9, offset time 32         44, offset time 159       80, offset time 289
10, offset time 36        45, offset time 163       81, offset time 294
11, offset time 40        46, offset time 167       82, offset time 298
12, offset time 43        47, offset time 170       83, offset time 301
13, offset time 47        48, offset time 174       84, offset time 304
14, offset time 50        49, offset time 177       85, offset time 308
15, offset time 54        50, offset time 181       86, offset time 311
16, offset time 58        51, offset time 185       87, offset time 316
17, offset time 61        52, offset time 188       88, offset time 319
18, offset time 65        53, offset time 193       89, offset time 322
19, offset time 69        54, offset time 195       90, offset time 326
20, offset time 72        55, offset time 200       91, offset time 329
21, offset time 76        56, offset time 203       92, offset time 333
22, offset time 80        57, offset time 206       93, offset time 338
23, offset time 83        58, offset time 210       94, offset time 340
24, offset time 87        59, offset time 213       95, offset time 345
25, offset time 91        60, offset time 217       96, offset time 348
26, offset time 94        61, offset time 220       97, offset time 352
27, offset time 97        62, offset time 225       98, offset time 356
28, offset time 101       63, offset time 228       99, offset time 359
29, offset time 105       64, offset time 231
30, offset time 108       65, offset time 235
31, offset time 112       66, offset time 240
32, offset time 116       67, offset time 243
33, offset time 119       68, offset time 247
34, offset time 123       69, offset time 250

finish, total time 3329
```

/ Animator改进 /

Android 在 Android 3.0，API11 中提供了更强大的动画 Animator，借助其中的 ValueAnimator，可以很方便的 编排 动画。

即便尚未分析原理，只要使用过属性动画，也知道它具有非常丝滑的效果

以上还都是推测，接下来进行实测。

实现

刨去一致部分，我们需要完成以下两点:

- 创建 ValueAnimator 实例，并按照动画需求设置 时长、插值器、UpdateListener 等
- 若没有额外需要，可将 Animatable2 弱化为 Animatable，仅保留动画控制API，通过 ValueAnimator 实例委托实现API业务逻辑。

核心代码如下：完整代码可从github获取：**DrawableWorkShop**（https://github.com/leobert-lan/DrawableWorkShop）

```
class AnimLetterDrawable2 : Drawable(), Animatable {
  // 相似部分略去

  private val totalFrames = 30 * 3 //3 second, 30frames per second

  private val valueAnimator = ValueAnimator.ofInt(totalFrames).apply {
    duration = 3000L

    this.interpolator = LinearInterpolator()

    addUpdateListener {
      setFrame(it.animatedValue as Int)
    }
  }

  private var frameIndex = 0


  private fun setFrame(frame: Int) {
    if (frame >= totalFrames) {
      return
    }
    frameIndex = frame
    invalidateSelf()
  }

  override fun start() {
    Log.d(tag, "start called")
    valueAnimator.start()
  }

  override fun stop() {
    valueAnimator.cancel()
    setFrame(0)
  }

  override fun isRunning(): Boolean {
    return valueAnimator.isRunning
  }

}
```

gif的效果太差，可以在 github项目仓库 中获取 webm视频（https://github.com/leobert-lan/DrawableWorkShop/blob/main/DrawableWorkShop/animator%E6%95%88%E6%9E%9C.webm）

关键代码差异：

在原方案中，我们计算了下一帧的播放时间点，借助 scheduleSelf -> View#scheduleDrawable 进行了刷新

```
class AnimLetterDrawable {
  private fun setFrame(frame: Int, unschedule: Boolean, animate: Boolean) {
    if (frame >= totalFrames) {
      return
    }
    mAnimating = animate
    frameIndex = frame

    if (unschedule || animate) {
      unscheduleSelf(this)
    }
    if (animate) {
      // Unscheduling may have clobbered these values; restore them
      frameIndex = frame

      scheduleSelf(this, SystemClock.uptimeMillis() + durationPerFrame)
    }
    invalidateSelf()
  }
}
```

而新方案中，我们借助ValueAnimator的更新回调函数直接刷新，显示预定帧

```
class AnimLetterDrawable2 {
  private val valueAnimator = ValueAnimator.ofInt(totalFrames).apply {
    duration = 3000L

    this.interpolator = LinearInterpolator()

    addUpdateListener {
      setFrame(it.animatedValue as Int)
    }
  }

  private fun setFrame(frame: Int) {
    if (frame >= totalFrames) {
      return
    }
    frameIndex = frame
    invalidateSelf()
  }
}
```

/ Animator原理 /

此时，再来思索一番，为何 Animator 的实现效果明显丝滑呢？

思危：是否和scheduleDrawable相比使用了不一样的底层机制？

源码跟进

单纯阅读文章内的代码会很枯燥，建议读者诸君对文中列出的源码进行泛读，抓住思路后再精读一遍源码。

以下将有6个关键点，可厘清其原理

1. start方法 -- 找到动画被驱动的核心

2. AnimationHandler#addAnimationFrameCallback(AnimationFrameCallback)

3. mAnimationCallbacks 何时移除元素

4. AnimationHandler#doAnimationFrame 方法的逻辑

5. 向前看，何人调用FrameCallback -- 驱动动画的底层逻辑

6. 向后看，ValueAnimator#doAnimationFrame -- 丝滑的原因

1. start方法

```
class ValueAnimator {

  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");
    }
    //略去一部分
    addAnimationCallback(0); //这里是核心

    if (mStartDelay == 0 || mSeekFraction >= 0 || mReversing) {
      startAnimation();
      if (mSeekFraction == -1) {
        setCurrentPlayTime(0);
      } else {
        setCurrentFraction(mSeekFraction);
      }
    }
  }

  private void addAnimationCallback(long delay) {
    //startWithoutPulsing 才会return
    if (!mSelfPulse) {
      return;
    }
    getAnimationHandler().addAnimationFrameCallback(this, delay); //这里是核心
  }
}
```

简单阅读，可以排除掉 startAnimation setCurrentPlayTime setCurrentFraction，他们均不是动画回调的核心，只是在进行必要地初始化和FLAG状态维护。

真正的核心是：getAnimationHandler().addAnimationFrameCallback(this, delay);

注意：AnimationHandler 存在线程单例设计:

```
//使用方:
class ValueAnimator {
  public AnimationHandler getAnimationHandler() {
    return mAnimationHandler != null ? mAnimationHandler : AnimationHandler.getInstance();
  }
}

//ThreadLocal线程单例设计
class AnimationHandler {
  public final static ThreadLocal<AnimationHandler> sAnimatorHandler = new ThreadLocal<>();
  private boolean mListDirty = false;

  public static AnimationHandler getInstance() {
    if (sAnimatorHandler.get() == null) {
      sAnimatorHandler.set(new AnimationHandler());
    }
    return sAnimatorHandler.get();
  }
}
```

2. AnimationHandler#addAnimationFrameCallback(AnimationFrameCallback)

- 方法逻辑中，有两处需要关注:

- 如果无 AnimationFrameCallback 回调实例， 说明没有在运行中的动画， 则挂载 Choreographer.FrameCallback mFrameCallback ， 为更新动画（_ 调用动画的 AnimationFrameCallback回调接口_）做准备。

在动画的 AnimationFrameCallback 回调实例未被注册的情况下，注册该回调实例

看完这一段源码，读者诸君一定会对以下两点产生兴趣，我们在下文展开：

- doAnimationFrame 方法的逻辑
- mAnimationCallbacks 何时移除元素

先看源码：

```
public class AnimationHandler {
  private final Choreographer.FrameCallback mFrameCallback = new Choreographer.FrameCallback() {
    @Override
    public void doFrame(long frameTimeNanos) {
      doAnimationFrame(getProvider().getFrameTime());

      //这不就破案了，只要还有动画的 AnimationFrameCallback，就挂载 mFrameCallback

      if (mAnimationCallbacks.size() > 0) {
        getProvider().postFrameCallback(this);
      }
    }
  };

  private AnimationFrameCallbackProvider getProvider() {
    if (mProvider == null) {
      mProvider = new MyFrameCallbackProvider();
    }
    return mProvider;
  }

  public void addAnimationFrameCallback(final AnimationFrameCallback callback, long delay) {
    if (mAnimationCallbacks.size() == 0) {
      getProvider().postFrameCallback(mFrameCallback);
    }
    if (!mAnimationCallbacks.contains(callback)) {
      mAnimationCallbacks.add(callback);
    }

    //注意，delay为0，阅读时可以忽略这段逻辑
    if (delay > 0) {
      mDelayedCallbackStartTime.put(callback, (SystemClock.uptimeMillis() + delay));
    }
  }
}
```

3. mAnimationCallbacks 何时移除元素

AnimationHandler中 "清理" mAnimationCallbacks 的设计：先设置null，再择机集中清理null，维护链表结构。可以避免循环过程中移除元素带来的潜在bug、以及避免频繁调整链表空间带来的损耗

关键代码为：android.animation.AnimationHandler#removeCallback，它有两处调用点，看完下面这一段源码后再行分析。

```
class AnimationHandler {
  public void removeCallback(AnimationFrameCallback callback) {
    mCommitCallbacks.remove(callback);
    mDelayedCallbackStartTime.remove(callback);
    int id = mAnimationCallbacks.indexOf(callback);
    if (id >= 0) {
      mAnimationCallbacks.set(id, null);
      mListDirty = true;
    }
  }
```

```
    private void cleanUpList() {
      if (mListDirty) {
        for (int i = mAnimationCallbacks.size() - 1; i >= 0; i--) {
          if (mAnimationCallbacks.get(i) == null) {
            mAnimationCallbacks.remove(i);
          }
        }
        mListDirty = false;
      }
    }
  }
```

removeCallback 存在一个直接调用，进而可找到两个间接调用点：

- endAnimation 停止动画时, 主动停止以及计算出动画已结束
- doAnimationFrame 中发现动画已经被暂停

再看一下源码：

```
  class ValueAnimator {
    private void removeAnimationCallback() {
      if (!mSelfPulse) {
        return;
      }
      //直接调用-1
      getAnimationHandler().removeCallback(this);
    }

    private void endAnimation() {
      if (mAnimationEndRequested) {
        return;
      }
      //间接调用-1
      removeAnimationCallback();
      //略去
    }

    public final boolean doAnimationFrame(long frameTime) {
      if (mStartTime < 0) {
        // First frame. If there is start delay, start delay count down will happen *after* this
        // frame.
        mStartTime = mReversing
            ? frameTime
            : frameTime + (long) (mStartDelay * resolveDurationScale());
      }

      // Handle pause/resume
      if (mPaused) {
        mPauseTime = frameTime;
        //间接调用-2
        removeAnimationCallback();
        return false;
      }
      //略
    }
  }
```

4. AnimationHandler#doAnimationFrame 方法的逻辑

一共有三个业务目的：

筛选，调用回调

处理 CommitCallback 情况

清理 mAnimationCallbacks 详见3

```
class AnimationHandler {
  private void doAnimationFrame(long frameTime) {
    long currentTime = SystemClock.uptimeMillis();
    final int size = mAnimationCallbacks.size();
    for (int i = 0; i < size; i++) {
      final AnimationFrameCallback callback = mAnimationCallbacks.get(i);

      // `为何会有null? ` 请看3 `mAnimationCallbacks` 何时移除元素
      if (callback == null) {
        continue;
      }

      //如果是延迟执行的callback，在未到预定时间时为false
      if (isCallbackDue(callback, currentTime)) {

        // 回调，实际逻辑：android.animation.ValueAnimator#doAnimationFrame
        callback.doAnimationFrame(frameTime);

        // 此处值得再写一篇文章
        if (mCommitCallbacks.contains(callback)) {
          getProvider().postCommitCallback(new Runnable() {
            @Override
            public void run() {
              commitAnimationFrame(callback, getProvider().getFrameTime());
            }
          });
        }
      }
    }
    cleanUpList();
  }

  private void commitAnimationFrame(AnimationFrameCallback callback, long frameTime) {
    if (!mDelayedCallbackStartTime.containsKey(callback) &&
        mCommitCallbacks.contains(callback)) {
      callback.commitAnimationFrame(frameTime);
      mCommitCallbacks.remove(callback);
    }
  }
}
```

作者按：值得一提的是，AnimationHandler中定义了所谓的 OneShotCommitCallback ，均添加到mCommitCallbacks中。

ValueAnimator 中曾利用它调整动画起始帧回调

SDK 24 、25 中明确存在，从26直至32均未发现使用。注意，我此次翻阅源码时较为粗略，仍需详查 android.animation.ValueAnimator#addOneShotCommitCallback 方可定论，如有谬误还请读者指出，避免误导。

5. 向前看，何人调用FrameCallback

跟进 getProvider().postFrameCallback(mFrameCallback); 发现是暗度陈仓

```
class AnimationHandler {
  private AnimationFrameCallbackProvider getProvider() {
    if (mProvider == null) {
      mProvider = new MyFrameCallbackProvider();
    }
    return mProvider;
  }

  private class MyFrameCallbackProvider implements AnimationFrameCallbackProvider {

    final Choreographer mChoreographer = Choreographer.getInstance();

    @Override
    public void postFrameCallback(Choreographer.FrameCallback callback) {
      mChoreographer.postFrameCallback(callback);
    }

    @Override
    public void postCommitCallback(Runnable runnable) {
      mChoreographer.postCallback(Choreographer.CALLBACK_COMMIT, runnable, null);
    }

    @Override
    public long getFrameTime() {
      return mChoreographer.getFrameTime();
    }

    @Override
    public long getFrameDelay() {
      return Choreographer.getFrameDelay();
    }

    @Override
    public void setFrameDelay(long delay) {
      Choreographer.setFrameDelay(delay);
    }
  }
}
```

又见 Choreographer ，这回应该不陌生了，跟进代码：

```
class Choreographer {
  public void postFrameCallback(FrameCallback callback) {
    postFrameCallbackDelayed(callback, 0);
  }

  public void postFrameCallbackDelayed(FrameCallback callback, long delayMillis) {
    if (callback == null) {
      throw new IllegalArgumentException("callback must not be null");
    }

    postCallbackDelayedInternal(CALLBACK_ANIMATION,
        callback, FRAME_CALLBACK_TOKEN, delayMillis);
  }
}
```

值得注意的是：此次使用的是：CALLBACK_ANIMATION

Choreographer 中将Callback一共 **分为5类**

- CALLBACK_INPUT = 0;
- CALLBACK_ANIMATION = 1;
- CALLBACK_INSETS_ANIMATION = 2;
- CALLBACK_TRAVERSAL = 3;
- CALLBACK_COMMIT = 4;

回调时的顺序也是如此。

读者诸君可还记得前文给出的 基于消息机制处理UI绘制 的关键流程图？其中多次出现关键字样：TRAVERSAL，对应此处的 CALLBACK_TRAVERSAL，它负责界面布局和绘制相关的业务。

而在上文 View#scheduleDrawable 的分析中，发现它使用的类型为：Choreographer.CALLBACK_ANIMATION，和 Animator 是一致的！

至此，我们悬着的心可以放下，Animator 和 View#scheduleDrawable 相比，使用了同样的底层机制

但是我们的疑问尚未得到答案，再顺着整个流程向后看。

6. 向后看，ValueAnimator#doAnimationFrame

作者按，以API25之后的源码解析，以下源码为API30，注意24之前、24&25，均存在差异，主要体现为首帧的开始。省略部分不重要的源码细节

不难发现，重点部分为：animateBasedOnTime(currentTime)

```
class ValueAnimator {
  public final boolean doAnimationFrame(long frameTime) {
    if (mStartTime < 0) {
      // First frame. If there is start delay, start delay count down will happen *after* this
      // frame.
      mStartTime = mReversing
          ? frameTime
          : frameTime + (long) (mStartDelay * resolveDurationScale());
    }

    // Handle pause/resume
    //省略 暂停、恢复的处理

    if (!mRunning) {
      //省略，判断是否可以开始播放首帧
    }

    if (mLastFrameTime < 0) {
      //省略，处理动画是否seek的情况
    }
    mLastFrameTime = frameTime;

    // The frame time might be before the start time during the first frame of
    // an animation.  The "current time" must always be on or after the start
    // time to avoid animating frames at negative time intervals.  In practice, this
    // is very rare and only happens when seeking backwards.
    final long currentTime = Math.max(frameTime, mStartTime);

    //此处为重点
    boolean finished = animateBasedOnTime(currentTime);

    //完毕的处理
    if (finished) {
      endAnimation();
    }
    return finished;
  }
}
```

继续抓住重点：animateBasedOnTime(currentTime)

```
class ValueAnimator {
  boolean animateBasedOnTime(long currentTime) {
    boolean done = false;
    if (mRunning) {
      //确定lastFraction、fraction
      final long scaledDuration = getScaledDuration();

      //差别在这里
      final float fraction = scaledDuration > 0 ?
          (float) (currentTime - mStartTime) / scaledDuration : 1f;
      final float lastFraction = mOverallFraction;
```

```
            //确定轮播迭代标记
            final boolean newIteration = (int) fraction > (int) lastFraction;
            final boolean lastIterationFinished = (fraction >= mRepeatCount + 1) &&
                (mRepeatCount != INFINITE);

            // 确定 done
            if (scaledDuration == 0) {
              // 0 duration animator, ignore the repeat count and skip to the end
              done = true;
            } else if (newIteration && !lastIterationFinished) {
              // Time to repeat
              if (mListeners != null) {
                int numListeners = mListeners.size();
                for (int i = 0; i < numListeners; ++i) {
                  mListeners.get(i).onAnimationRepeat(this);
                }
              }
            } else if (lastIterationFinished) {
              done = true;
            }

            //确定fraction 重点1
            mOverallFraction = clampFraction(fraction);
            float currentIterationFraction = getCurrentIterationFraction(
                mOverallFraction, mReversing);

            //重点2
            animateValue(currentIterationFraction);
          }
          return done;
        }
      }
```

此处有两处重点：

- 确定 currentIterationFraction
- animateValue 执行动画帧

看重点1：泛读即可，主要理解fraction的设计

```
class ValueAnimator {
  private float clampFraction(float fraction) {
    if (fraction < 0) {
      fraction = 0;
    } else if (mRepeatCount != INFINITE) {
      fraction = Math.min(fraction, mRepeatCount + 1);
    }
    return fraction;
  }

  //重点1 整数部分代表iteration，小数部分代表当前iteration的fraction
  private float getCurrentIterationFraction(float fraction, boolean inReverse) {
    fraction = clampFraction(fraction);
    int iteration = getCurrentIteration(fraction);
    float currentFraction = fraction - iteration;
    return shouldPlayBackward(iteration, inReverse)
        ? 1f - currentFraction
        : currentFraction;
  }

  //依据是fraction和iteration的设计：
  //   Calculates current iteration based on the overall fraction.
  //   The overall fraction will be in the range of [0, mRepeatCount + 1].
  //   Both current iteration and fraction in the current iteration can be derived from it.
  private int getCurrentIteration(float fraction) {
    fraction = clampFraction(fraction);
    // If the overall fraction is a positive integer, we consider the current iteration to be
    // complete. In other words, the fraction for the current iteration would be 1, and the
    // current iteration would be overall fraction - 1.
    double iteration = Math.floor(fraction);
    if (fraction == iteration && fraction > 0) {
      iteration--;
    }
    return (int) iteration;
  }

  //和动画正向、反向播放有关，可先忽略
  private boolean shouldPlayBackward(int iteration, boolean inReverse) {
    if (iteration > 0 && mRepeatMode == REVERSE &&
        (iteration < (mRepeatCount + 1) || mRepeatCount == INFINITE)) {
      // if we were seeked to some other iteration in a reversing animator,
      // figure out the correct direction to start playing based on the iteration
      if (inReverse) {
        return (iteration % 2) == 0;
      } else {
        return (iteration % 2) != 0;
      }
    } else {
      return inReverse;
    }
  }
}
```

看重点2：

```
class ValueAnimator {
  void animateValue(float fraction) {
    //插值器重新计算fraction -- 优雅的设计
    fraction = mInterpolator.getInterpolation(fraction);
    mCurrentFraction = fraction;
    int numValues = mValues.length;

    //PropertyValuesHolder 计算value -- 又是一个优雅的设计
    for (int i = 0; i < numValues; ++i) {
      mValues[i].calculateValue(fraction);
    }

    //回调，onAnimationUpdate 常用到 getAnimatedValue，和 calculateValue 对应
    if (mUpdateListeners != null) {
      int numListeners = mUpdateListeners.size();
      for (int i = 0; i < numListeners; ++i) {
        mUpdateListeners.get(i).onAnimationUpdate(this);
      }
    }
  }
}
```

阶段性小结

源码内容着实很多，经过刚才的源码重点拆解，也已梳理出大致流程。

回归到我们阅读源码前的问题：

Animator 是否和scheduleDrawable相比使用了不一样的底层机制？

否，均使用了 Choreographer [ˌkɔːriˈɑːɡrəfər],记住它的读写 + Vsync + Android 消息机制 ， 且回调类型一致，均为 CALLBACK_ANIMATION

为何更加丝滑？

动画内部调用频次 ≥ 原方案，回调时依据时间计算帧号的算法更加准确合理

ValueAnimator#animateBasedOnTime 中，使用了准确、合理的计算方式： final float fraction = scaledDuration > 0 ? (float) (currentTime - mStartTime) / scaledDuration : 1f;

而先前文章中的代码，并没有依据当前实际时间调整帧。

/ 打开思路 /

至此，动画的核心奥秘已经揭开，似乎一切已尽在不言中，轮子也均已完备，也并不需要再额外实现一套插值器、估值器逻辑。

既然如此，我们不再对第一篇中的例子进行以下改进："依据时间调整帧"，"提升回调频率"。

作者按：如果下次计划写插值器、估值器的文章，可能以逐步完善造轮子的方式进行内容展开

那么本篇的核心内容，除了面试或者给同事科普外，还能带来什么呢？

整体回顾一下，并打开思路：

1. 我们从一个实例出发进行完善，并收获一个经验：可以通过 Drawable+Animator，将动画内容推广到任意View做显示，如果没有必要，可以少做一些自定义View的事情。
2. 分析了Drawable更新内容的底层实现，是否可以将这种动画效果推广到更多地方呢？例如TextView的DrawableStart、ImageSpan，是否都能正确显示动效呢？，如果不能要怎么做？

3. 我们分析动画被驱动的过程中，遇到一个宝藏 Choreographer，是否可以拿来干点有趣的事情？例如：FPS监测
4. 将ValueAnimator的核心机制复刻，在别的平台搞点好玩的东西😂
5. 在视觉呈现内容 与 时间 的函数关系确定时，使用 ValueAnimator 作为核心驱动，将问题变为一个纯数学问题，例如 点迹动效绘制，全景图锚点A到锚点B之间的渐变
6. 融合以上内容，自定义一套数据协议，解析后，所见皆可动起来

文中出现的源码，除去AOSP部分，均收录于仓库中：DrawableWorkShop（https://github.com/leobert-lan/DrawableWorkShop）

推荐阅读：

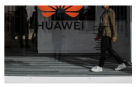
我的新书，《第一行代码 第3版》已出版！

协程简史，一文讲清楚协程的起源、发展和实现

Kotlin Flow响应式编程，基础知识入门

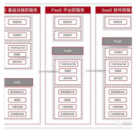
阅读原文

圆锥体



图解架构 | SaaS、PaaS、IaaS

悟空聊架构



PC将死？

价值研究所