布局加载源码分析：

先从Activity.setContentView开始分析布局是如何被加载的：

Activity.setContentView(@LayoutRes int layoutResID)

public void setContentView(@LayoutRes int layoutResID) {

getWindow().setContentView(layoutResID);

initWindowDecorActionBar();

}

PhoneWIndow.setContentView(int layoutResID)

@Override

public void setContentView(int layoutResID) {

if (mContentParent == null) {

//初始化DecorView和mContentParent

installDecor();

}

...

//加载资源文件，创建view树装载到mContentParent

mLayoutInflater.inflate(layoutResID, mContentParent);

...

}

LayoutInflate.inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean attachToRoot)

public View inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean attachToRoot) {

final Resources res = getContext().getResources();

//1.加载解析xml文件

final XmlResourceParser parser = res.getLayout(resource);

try {

//2.填充View树

return inflate(parser, root, attachToRoot);

} finally {

parser.close();

}

}

可以看出布局加载流程主要分为加载解析xml文件和填充View树两部分

加载解析xml文件：

Resources.getLayout(@LayoutRes int id)

public XmlResourceParser getLayout(@LayoutRes int id) throws NotFoundException {

return loadXmlResourceParser(id, "layout");

}

Resources.loadXmlResourceParser(@NonNull String file, @AnyRes int id, int assetCookie,@NonNul l String type)

XmlResourceParser loadXmlResourceParser(@NonNull String file, @AnyRes int id, int assetCookie,@NonNul l String type) throws NotFoundException {

if (id != 0) {

try {

synchronized (mCachedXmlBlocks) {

final int[] cachedXmlBlockCookies = mCachedXmlBlockCookies;

final String[] cachedXmlBlockFiles = mCachedXmlBlockFiles;

final XmlBlock[] cachedXmlBlocks = mCachedXmlBlocks;

// First see if this block is in our cache.

final int num = cachedXmlBlockFiles.length;

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

if (cachedXmlBlockCookies[i] == assetCookie && cachedXmlBlockFiles[i] != null

&& cachedXmlBlockFiles[i].equals(file)) {

return cachedXmlBlocks[i].newParser();

}

}

// Not in the cache, create a new block and put it at

// the next slot in the cache.

final XmlBlock block = mAssets.openXmlBlockAsset(assetCookie, file);

if (block != null) {

final int pos = (mLastCachedXmlBlockIndex + 1) % num;

mLastCachedXmlBlockIndex = pos;

final XmlBlock oldBlock = cachedXmlBlocks[pos];

if (oldBlock != null) {

oldBlock.close();

}

cachedXmlBlockCookies[pos] = assetCookie;

cachedXmlBlockFiles[pos] = file;

cachedXmlBlocks[pos] = block;

return block.newParser();

}

}

} catch (Exception e) {

}

}

}

我们不用非常深入这个方法的具体实现细节，我们只需要知道，这个方法的作用就是将我们写的xml文件读取到内存中，并进行一些数据解析和封装。所以这个方法本质上就是一个IO操作，我们知道，IO操作往往是比较耗费性能的

填充View树：

LayoutInflate.inflate(XmlPullParser parser, @Nullable ViewGroup root, boolean attachToRoot)

public View inflate(XmlPullParser parser, @Nullable ViewGroup root, boolean attachToRoot) {

synchronized (mConstructorArgs) {

final Context inflaterContext = mContext;

final AttributeSet attrs = Xml.asAttributeSet(parser);

Context lastContext = (Context) mConstructorArgs[0];

mConstructorArgs[0] = inflaterContext;

View result = root;

int type;

final String name = parser.getName();

if (TAG\_MERGE.equals(name)) {

if (root == null || !attachToRoot) {

throw new InflateException("<merge /> can be used only with a valid "+ "ViewGroup root and attachToRoot=true");

}

rInflate(parser, root, inflaterContext, attrs, false);

} else {

// Temp is the root view that was found in the xml

final View temp = createViewFromTag(root, name, inflaterContext, attrs);

ViewGroup.LayoutParams params = null;

if (root != null) {

params = root.generateLayoutParams(attrs);

if (!attachToRoot) {

temp.setLayoutParams(params);

}

}

rInflateChildren(parser, temp, attrs, true);

if (root != null && attachToRoot) {

root.addView(temp, params);

}

if (root == null || !attachToRoot) {

result = temp;

}

}

return result;

}

}

上面这个方法中我们最主要关注createViewFromTag(View parent, String name, Context context, AttributeSet attrs)

View createViewFromTag(View parent, String name, Context context, AttributeSet attrs,

boolean ignoreThemeAttr) {

//解析view标签

if (name.equals("view")) {

name = attrs.getAttributeValue(null, "class");

}

//如果需要该标签与主题相关，需要对context进行包装，将主题信息加入context包装类ContextWrapper

if (!ignoreThemeAttr) {

final TypedArray ta = context.obtainStyledAttributes(attrs, ATTRS\_THEME);

final int themeResId = ta.getResourceId(0, 0);

if (themeResId != 0) {

context = new ContextThemeWrapper(context, themeResId);

}

ta.recycle();

}

if (name.equals(TAG\_1995)) {

//BlinkLayout是一种闪烁的FrameLayout，它包裹的内容会一直闪烁，类似QQ提示消息那种。

return new BlinkLayout(context, attrs);

}

//设置Factory，来对View做额外的拓展，这块属于可定制的内容

View view;

if (mFactory2 != null) {

view = mFactory2.onCreateView(parent, name, context, attrs);

} else if (mFactory != null) {

view = mFactory.onCreateView(name, context, attrs);

} else {

view = null;

}

if (view == null && mPrivateFactory != null) {

view = mPrivateFactory.onCreateView(parent, name, context, attrs);

}

//如果此时不存在Factory，不管Factory还是Factory2，还是mPrivateFactory都不存在，那么会直接对name直接进行解析

if (view == null) {

final Object lastContext = mConstructorArgs[0];

mConstructorArgs[0] = context;

try {

//如果name中包含"."即为自定义View，否则为原生的View控件

if (-1 == name.indexOf('.')) {

view = onCreateView(parent, name, attrs);

} else {

view = createView(name, null, attrs);

}

} finally {

mConstructorArgs[0] = lastContext;

}

}

return view;

}

根据源码可以将createViewFromTag分为三个流程：

对一些特殊标签，做分别处理，例如：view，TAG\_1995(blink)

进行对Factory、Factory2的设置判断，如果设置那么就会通过设置Factory、Factory2进行生成View

如果没有设置Factory或Factory2，那么就会使用LayoutInflater默认的生成方式，进行View的生成

createViewFromTag过程分析：

处理view标签

如果标签的名称是view，注意是小写的view，这个标签一般大家不太常用

<view

class="RelativeLayout"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"></view>

在使用时，相当于所有控件标签的父类一样，可以设置class属性，这个属性会决定view这个节点会变成什么控件

如果该节点与主题相关，则需要特殊处理

如果该节点与主题（Theme）相关，需要将context与theme信息包装至ContextWrapper类

处理TAG\_1995标签

这就有意思了，TAG\_1995指的是blink这个标签，这个标签感觉使用的很少，以至于大家根本不知道。

这个标签最后会被解析成BlinkLayout，BlinkLayout其实就是一个FrameLayout，这个控件最后会将包裹内容一直闪烁(就和电脑版QQ消息提示一样）

<blink

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="这个标签会一直闪烁"/>

</blink>

判断其是否存在Factory或者Factory2

在这里先对Factory进行判空，这里不管Factory还是Factory2（mPrivateFactory 就是Factory2），本质上都是一种扩展操作，提前解析name，然后直接将解析后的View返回

public interface Factory {

public View onCreateView(String name, Context context, AttributeSet attrs);

}

Factory2

public interface Factory2 extends Factory {

public View onCreateView(View parent, String name, Context context, AttributeSet attrs);

}

从这里可以看出，Factory2和Factory都是一个接口，需要自己实现，而Factory2和Factory的区别是Factory2继承Factory，从而扩展出一个参数，就是增加了该节点的父View。设置Factory和Factory2需要通过setFactory()或者setFactory2()来实现

setFactory()

public void setFactory(Factory factory) {

//如果已经设置Factory，不可以继续设置Factory

if (mFactorySet) {

throw new IllegalStateException("A factory has already been set on this LayoutInflater");

}

if (factory == null) {

throw new NullPointerException("Given factory can not be null");

}

//设置Factory会添加一个标记

mFactorySet = true;

if (mFactory == null) {

mFactory = factory;

} else {

mFactory = new FactoryMerger(factory, null, mFactory, mFactory2);

}

}

setFactory2()

public void setFactory2(Factory2 factory) {

if (mFactorySet) {

throw new IllegalStateException("A factory has already been set on this LayoutInflater");

}

if (factory == null) {

throw new NullPointerException("Given factory can not be null");

}

//注意设置Factory和Factory2的标记是共用的

mFactorySet = true;

if (mFactory == null) {

mFactory = mFactory2 = factory;

} else {

mFactory = mFactory2 = new FactoryMerger(factory, factory, mFactory, mFactory2);

}

}

通过上面代码可以看出，Factory和Factory2只能够设置一次，并且Factory和Factory2二者互斥，只能存在一个。所以一般setFactory()或者setFactory2()，一般在cloneInContext()之后设置，这样生成一个新的LayoutInflater，标记默认是false，才能够设置

createView(String name, String prefix, AttributeSet attrs)

public final View createView(String name, String prefix, AttributeSet attrs)

throws ClassNotFoundException, InflateException {

//判断构造器是否存在

Constructor<? extends View> constructor = sConstructorMap.get(name);

if (constructor != null && !verifyClassLoader(constructor)) {

constructor = null;

sConstructorMap.remove(name);

}

Class<? extends View> clazz = null;

try {

//如果构造器不存在，这个就相当于Class之前是否被加载过，sConstructorMap就是缓存这些Class的Map

if (constructor == null) {

//通过前缀+name的方式去加载

clazz = mContext.getClassLoader().loadClass(

prefix != null ? (prefix + name) : name).asSubclass(View.class);

//通过过滤去设置一些不需要加载的对象

if (mFilter != null && clazz != null) {

boolean allowed = mFilter.onLoadClass(clazz);

if (!allowed) {

failNotAllowed(name, prefix, attrs);

}

}

constructor = clazz.getConstructor(mConstructorSignature);

constructor.setAccessible(true);

//缓存Class

sConstructorMap.put(name, constructor);

} else {

//如果Class存在，并且加载Class的ClassLoader合法

//这里先判断该Class是否应该被过滤

if (mFilter != null) {

//过滤器也有缓存之前的Class是否被允许加载，判断这个Class的过滤状态

Boolean allowedState = mFilterMap.get(name);

if (allowedState == null) {

//加载Class对象操作

clazz = mContext.getClassLoader().loadClass(

prefix != null ? (prefix + name) : name).asSubclass(View.class);

//判断Class是否可被加载

boolean allowed = clazz != null && mFilter.onLoadClass(clazz);

mFilterMap.put(name, allowed);

if (!allowed) {

failNotAllowed(name, prefix, attrs);

}

} else if (allowedState.equals(Boolean.FALSE)) {

failNotAllowed(name, prefix, attrs);

}

}

}

Object[] args = mConstructorArgs;

args[1] = attrs;

//如果过滤器不存在，直接实例化该View

final View view = constructor.newInstance(args);

//如果View属于ViewStub那么需要给ViewStub设置一个克隆过的LayoutInflater

if (view instanceof ViewStub) {

final ViewStub viewStub = (ViewStub) view;

viewStub.setLayoutInflater(cloneInContext((Context) args[0]));

}

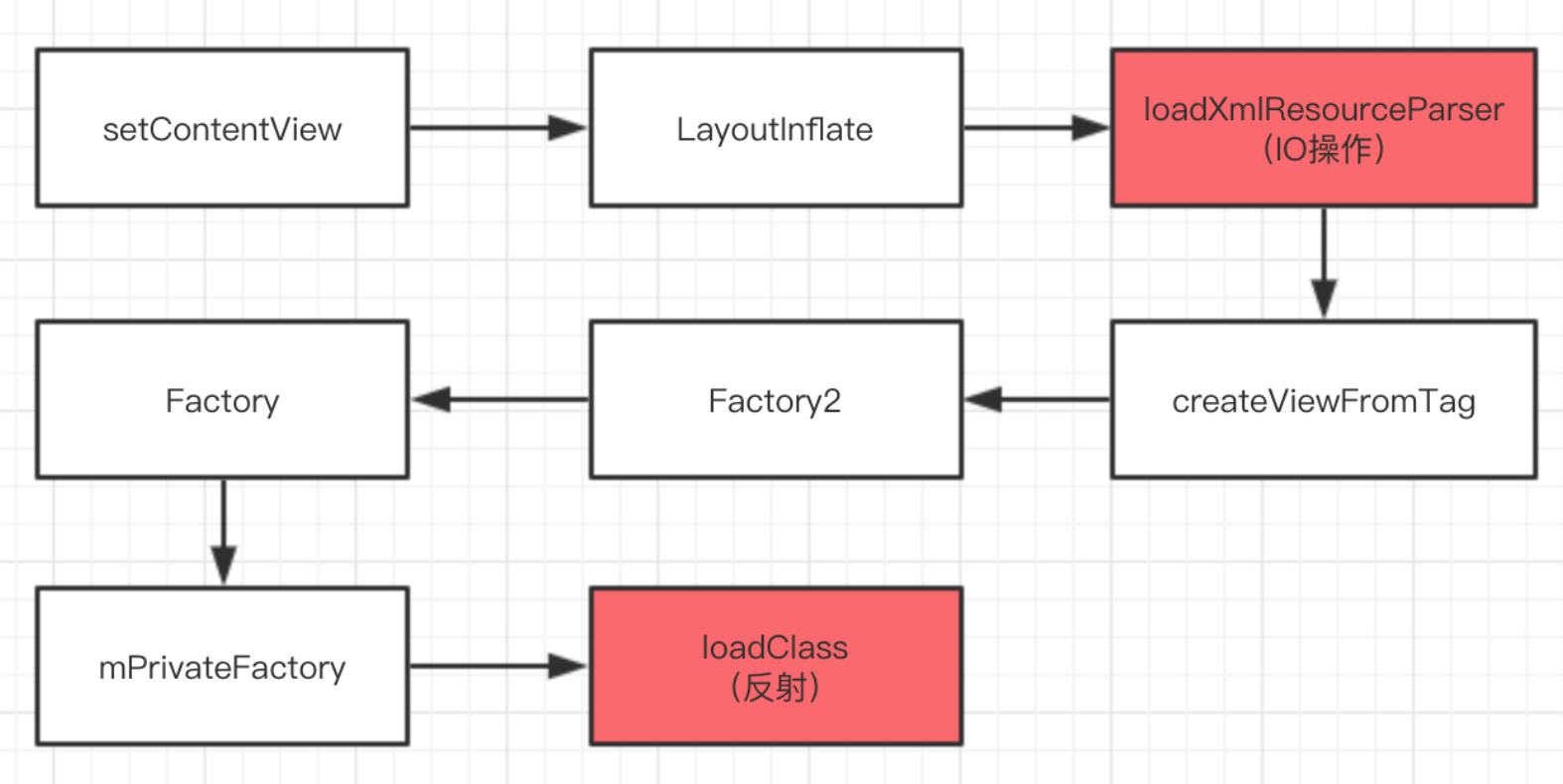
return view

通过反射的方式去创建View实例的。

经过对布局加载原理的分析，我们可以看出布局加载的主要性能瓶颈主要在两个方面

加载xml文件是一个IO过程，如果xml文件过大，就会比较耗时

View实例是通过反射进行创建的，通过反射创建对象相对会更耗费性能



获取界面布局耗时：

可以通过AOP的方式更优雅、侵入性更低的统计页面布局耗时，如下方法能够Hook所有的Activity的setContentView方法

@Aspect

public class setContentViewAspect {

@Around("execution(\* android.app.Activity.setContentView(..))")

public void hookSetContentView(ProceedingJoinPoint joinPoint) {

Signature signature = joinPoint.getSignature();

//开始处打点

String name = signature.toShortString();

long startTime = System.currentTimeMillis();

try {

joinPoint.proceed();

} catch (Throwable throwable) {

throwable.printStackTrace();

}

//结束打点，并计算耗时。

long costTime = System.currentTimeMillis() - startTime;

Log.i("Geekholt", "method " + name + " cost:" + costTime);

}

}

有时候我们还希望监测一些自定义控件的加载耗时，那么我们要怎么做呢？

Factory，Factory2继承自Factory接口，在API 11(HONEYCOMB)中引入的。Factory2的onCreateView(View parent, String name, Context context, AttributeSet attrs)比Factory多了一个parent参数，用来存放构建出的View

系统通过Factory提供了一种hook的方法，方便开发者拦截LayoutInflater创建View的过程。应用场景包括：

在XML布局中自定义标签名称；

全局替换系统控件为自定义View；

替换app中字体；

全局换肤；

获取控件加载耗时等

在API Level 26及以后,LayoutInflaterCompat.setFactory被标记为Deprecated，故我们这里使用的是LayoutInflaterCompat.setFactory2。我们可以在项目基类BaseActivity的super.onCreate()前调用setFactory2，还可以将控件加载耗时上传到我们自己的数据后台用于控件性能检测

@Override

protected void onCreate(Bundle savedInstanceState) {

setFactory2();

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

private void setFactory2() {

LayoutInflaterCompat.setFactory2(getLayoutInflater(), new LayoutInflater.Factory2() {

@Override

public View onCreateView(View parent, String name, Context context, AttributeSet attrs) {

long startTime = System.currentTimeMillis();

View view = getDelegate().createView(parent, name, context, attrs);

long endTime = System.currentTimeMillis();

long cost = endTime - startTime;

Log.i("geekholt", name + "cost " + cost + "ms");

return view;

}

@Override

public View onCreateView(String name, Context context, AttributeSet attrs) {

return null;

}

});

}

为什么setFactory2一定要在super.onCreate()前调用呢？

//AppCompatActivity.java

@Override

protected void onCreate(@Nullable Bundle savedInstanceState) {

final AppCompatDelegate delegate = getDelegate();

//这里主要关注installViewFactory方法

delegate.installViewFactory();

delegate.onCreate(savedInstanceState);

...

super.onCreate(savedInstanceState);

}

官方提供的AppCompatDelegate子类实现，如AppCompatDelegateImplN。帮助我们实现了AppCompat风格组件的向下兼容，利用AppCompatDelegateImplN提供的Factory2将TextView等组件替换为AppCompatTextView，这样就可以使用一些新的属性，如autoSizeMinTextSize

//AppCompatDelegateImplV9.java

@Override

public void installViewFactory() {

LayoutInflater layoutInflater = LayoutInflater.from(mContext);

if (layoutInflater.getFactory() == null) {

LayoutInflaterCompat.setFactory2(layoutInflater, this);

} else {

if (!(layoutInflater.getFactory2() instanceof AppCompatDelegateImplV9)) {

Log.i(TAG, "The Activity's LayoutInflater already has a Factory installed"

+ " so we can not install AppCompat's");

}

}

}

/\*\*

\* From {@link LayoutInflater.Factory2}.

\*/

@Override

public final View onCreateView(View parent, String name, Context context, AttributeSet attrs) {

// First let the Activity's Factory try and inflate the view

final View view = callActivityOnCreateView(parent, name, context, attrs);

if (view != null) {

return view;

}

// If the Factory didn't handle it, let our createView() method try

return createView(parent, name, context, attrs);

}

上述代码可知，如果AppCompatActivity未在onCreate之前设置LayoutInflater的Factory，则AppCompatActivity会尝试设置一个Factory2，其中Factory2在AppCompatDelegate的具体子类代码中实现

//LayoutInflate.java

public void setFactory2(Factory2 factory) {

if (mFactorySet) {

throw new IllegalStateException("A factory has already been set on this LayoutInflater");

}

if (factory == null) {

throw new NullPointerException("Given factory can not be null");

}

mFactorySet = true;

if (mFactory == null) {

mFactory = mFactory2 = factory;

} else {

mFactory = mFactory2 = new FactoryMerger(factory, factory, mFactory, mFactory2);

}

}

在同一个Actiivty内，setFactory2不可以重复调用，所以，如果在super.onCreate之后设置就会抛出异常



这里有一个问题不知道读者有没有注意到，既然setFactory2不能重复调用，那如果我们主动在onCreate()之前调用了setFactory2，会不会使系统的AppCompatTextView等失效呢？答案其实是不会的，我们只需要在执行自己的定制化操作之前调用 getDelegate().createView(parent, name, context, attrs)就不会有问题了。

如现在有一个全局替换Textview字体需求，我们可以这么做。

LayoutInflaterCompat.setFactory2(LayoutInflater.from(this), new LayoutInflater.Factory2() {

@Override

public View onCreateView(View parent, String name, Context context, AttributeSet attrs) {

//调用兼容类的createView

View view = getDelegate().createView(parent, name, context, attrs);

if (view != null && (view instanceof TextView)) {

//你可以在这里直接new自定义View

//你可以在这里将系统类替换为自定义View

//替换字体示例

((TextView) view).setTypeface(typeface);

}

return view;

}

});

AsyncLayoutInflater异步加载布局：

我们知道，当主线程进行一些耗时操作可能就会导致页面卡顿，更严重的可能会产生ANR，所以我们能如何来进行布局加载优化呢？解决这个问题有两种思路，直接解决和侧面缓解。直接解决就是不使用IO和反射等技术（这个我们会在下一节进行介绍）。侧面缓解的就是既然耗时操作难以避免，那我们能不能把耗时操作放在子线程中，等到inflate操作完成后再将结果回调到主线程呢？答案当然是可以的，Android为我们提供了AsyncLayoutInflater类来进行异步布局加载

AsyncLayoutInflater的使用非常简单，就是把setContentView和一些view的初始化操作都放到了onInflateFinished回调中

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

new AsyncLayoutInflater(this).inflate(R.layout.activity\_main,null, new AsyncLayoutInflater.OnInflateFinishedListener(){

@Override

public void onInflateFinished(View view, int resid, ViewGroup parent) {

setContentView(view);

rv = findViewById(R.id.tv\_right);

rv.setLayoutManager(new V7LinearLayoutManager(MainActivity.this));

rv.setAdapter(new RightRvAdapter(MainActivity.this));

}

});

}

AsyncLayoutInflater构造方法和初始化

构造方法中做了三件事件

创建BasicInflater

创建Handler

创建InflateThread

inflate方法创建一个InflateRequest对象，并将resid、parent、callback等变量存储到这个对象中，并调用enqueue方法向队列中添加一个请求

public final class AsyncLayoutInflater {

private static final String TAG = "AsyncLayoutInflater";

LayoutInflater mInflater;

Handler mHandler;

InflateThread mInflateThread;

public AsyncLayoutInflater(@NonNull Context context) {

mInflater = new BasicInflater(context);

mHandler = new Handler(mHandlerCallback);

mInflateThread = InflateThread.getInstance();

}

@UiThread

public void inflate(@LayoutRes int resid, @Nullable ViewGroup parent,

@NonNull OnInflateFinishedListener callback) {

if (callback == null) {

throw new NullPointerException("callback argument may not be null!");

}

InflateRequest request = mInflateThread.obtainRequest();

request.inflater = this;

request.resid = resid;

request.parent = parent;

request.callback = callback;

mInflateThread.enqueue(request);

}

....

}

InflateThread

这个类的主要作用就是创建一个子线程，将inflate请求添加到阻塞队列中，并按顺序执行BasicInflater.inflate操作（BasicInflater实际上就是LayoutInflater的子类）。不管infalte成功或失败后，都会将request消息发送给主线程做处理

private static class InflateThread extends Thread {

private static final InflateThread sInstance;

static {

sInstance = new InflateThread();

sInstance.start();

}

public static InflateThread getInstance() {

return sInstance;

}

//生产者-消费者模型，阻塞队列

private ArrayBlockingQueue<InflateRequest> mQueue = new ArrayBlockingQueue<>(10);

//使用了对象池来缓存InflateThread对象，减少对象重复多次创建，避免内存抖动

private SynchronizedPool<InflateRequest> mRequestPool = new SynchronizedPool<>(10);

public void runInner() {

InflateRequest request;

try {

//从队列中取出一条请求，如果没有则阻塞

request = mQueue.take();

} catch (InterruptedException ex) {

// Odd, just continue

Log.w(TAG, ex);

return;

}

try {

//inflate操作（通过调用BasicInflater类）

request.view = request.inflater.mInflater.inflate(

request.resid, request.parent, false);

} catch (RuntimeException ex) {

// 回退机制：如果inflate失败，回到主线程去inflate

Log.w(TAG, "Failed to inflate resource in the background! Retrying on the UI"

+ " thread", ex);

}

//inflate成功或失败，都将request发送到主线程去处理

Message.obtain(request.inflater.mHandler, 0, request)

.sendToTarget();

}

@Override

public void run() {

//死循环（实际不会一直执行，内部是会阻塞等待的）

while (true) {

runInner();

}

}

//从对象池缓存中取出一个InflateThread对象

public InflateRequest obtainRequest() {

InflateRequest obj = mRequestPool.acquire();

if (obj == null) {

obj = new InflateRequest();

}

return obj;

}

//对象池缓存中的对象的数据清空，便于对象复用

public void releaseRequest(InflateRequest obj) {

obj.callback = null;

obj.inflater = null;

obj.parent = null;

obj.resid = 0;

obj.view = null;

mRequestPool.release(obj);

}

//将inflate请求添加到ArrayBlockingQueue（阻塞队列）中

public void enqueue(InflateRequest request) {

try {

mQueue.put(request);

} catch (InterruptedException e) {

throw new RuntimeException(

"Failed to enqueue async inflate request", e);

}

}

}

InflateRequest

InflateRequest其实就可以理解为主线程和子线程之间传递的数据模型，类似Message的作用

private static class InflateRequest {

AsyncLayoutInflater inflater;

ViewGroup parent;

int resid;

View view;

OnInflateFinishedListener callback;

InflateRequest() {

}

}

BasicInflater

BasicInflater 继承自 LayoutInflater，只是覆写了 onCreateView：优先加载这三个前缀的 Layout，然后才按照默认的流程去加载，因为大多数情况下我们 Layout 中使用的View都在这三个 package 下

private static class BasicInflater extends LayoutInflater {

private static final String[] sClassPrefixList = {

"android.widget.",

"android.webkit.",

"android.app."

};

BasicInflater(Context context) {

super(context);

}

@Override

public LayoutInflater cloneInContext(Context newContext) {

return new BasicInflater(newContext);

}

@Override

protected View onCreateView(String name, AttributeSet attrs) throws ClassNotFoundException {

for (String prefix : sClassPrefixList) {

try {

//优先加载"android.widget.”、 "android.webkit."、"android.app."

View view = createView(name, prefix, attrs);

if (view != null) {

return view;

}

} catch (ClassNotFoundException e) {

}

}

return super.onCreateView(name, attrs);

}

}

mHandlerCallback

这里就是在主线程中handleMessage的操作，这里有一个回退机制，就是当子线程中inflate失败后，会继续再主线程中进行inflate操作，最终通过OnInflateFinishedListener接口将view回调到主线程

private Callback mHandlerCallback = new Callback() {

@Override

public boolean handleMessage(Message msg) {

InflateRequest request = (InflateRequest) msg.obj;

if (request.view == null) {

//view == null说明inflate失败

//继续再主线程中进行inflate操作

request.view = mInflater.inflate(

request.resid, request.parent, false);

}

//回调到主线程

request.callback.onInflateFinished(

request.view, request.resid, request.parent);

mInflateThread.releaseRequest(request);

return true;

}

};

OnInflateFinishedListener

布局加载完成后，通过OnInflateFinishedListener将加载完成后的view回调出来

public interface OnInflateFinishedListener {

void onInflateFinished(View view, int resid, ViewGroup parent);

}

AsyncLayoutInflater的局限性及改进

使用AsyncLayoutInflate主要有如下几个局限性：

所有构建的View中必须不能直接使用 Handler 或者是调用 Looper.myLooper()，因为异步线程默认没有调用 Looper.prepare ()

异步转换出来的 View 并没有被加到 parent view中，AsyncLayoutInflater 是调用了 LayoutInflater.inflate(int, ViewGroup, false)，因此如果需要加到 parent view 中，就需要我们自己手动添加；

AsyncLayoutInflater 不支持设置 LayoutInflater.Factory 或者 LayoutInflater.Factory2

同时缓存队列默认 10 的大小限制如果超过了10个则会导致主线程的等待

使用单线程来做全部的 inflate 工作，如果一个界面中 layout 很多不一定能满足需求

那我们如何来解决这些问题呢？AsyncLayoutInflate类修饰为 final ，所以不能通过继承重写父类来实现。庆幸的是AsyncLayoutInflate的代码非常短而且相对简单，所以我们可以直接把AsyncLayoutInflate的代码复制出来一份，然后在这基础之上进行改进优化

引入线程池，减少单线程等待

手动设置setFactory2

public class AsyncLayoutInflatePlus {

private static final String TAG = "AsyncLayoutInflatePlus";

private Pools.SynchronizedPool<InflateRequest> mRequestPool = new Pools.SynchronizedPool<>(10);

LayoutInflater mInflater;

Handler mHandler;

Dispather mDispatcher;

public AsyncLayoutInflatePlus(@NonNull Context context) {

mInflater = new BasicInflater(context);

mHandler = new Handler(mHandlerCallback);

mDispatcher = new Dispather();

}

@UiThread

public void inflate(@LayoutRes int resid, @Nullable ViewGroup parent,

@NonNull OnInflateFinishedListener callback) {

if (callback == null) {

throw new NullPointerException("callback argument may not be null!");

}

InflateRequest request = obtainRequest();

request.inflater = this;

request.resid = resid;

request.parent = parent;

request.callback = callback;

mDispatcher.enqueue(request);

}

private Handler.Callback mHandlerCallback = new Handler.Callback() {

@Override

public boolean handleMessage(Message msg) {

InflateRequest request = (InflateRequest) msg.obj;

if (request.view == null) {

request.view = mInflater.inflate(

request.resid, request.parent, false);

}

request.callback.onInflateFinished(

request.view, request.resid, request.parent);

releaseRequest(request);

return true;

}

};

public interface OnInflateFinishedListener {

void onInflateFinished(@NonNull View view, @LayoutRes int resid,

@Nullable ViewGroup parent);

}

private static class InflateRequest {

AsyncLayoutInflatePlus inflater;

ViewGroup parent;

int resid;

View view;

OnInflateFinishedListener callback;

InflateRequest() {

}

}

private static class Dispather {

//获得当前CPU的核心数

private static final int CPU\_COUNT = Runtime.getRuntime().availableProcessors();

//设置线程池的核心线程数2-4之间,但是取决于CPU核数

private static final int CORE\_POOL\_SIZE = Math.max(2, Math.min(CPU\_COUNT - 1, 4));

//设置线程池的最大线程数为 CPU核数 \* 2 + 1

private static final int MAXIMUM\_POOL\_SIZE = CPU\_COUNT \* 2 + 1;

//设置线程池空闲线程存活时间30s

private static final int KEEP\_ALIVE\_SECONDS = 30;

private static final ThreadFactory sThreadFactory = new ThreadFactory() {

private final AtomicInteger mCount = new AtomicInteger(1);

public Thread newThread(Runnable r) {

return new Thread(r, "AsyncLayoutInflatePlus #" + mCount.getAndIncrement());

}

};

//LinkedBlockingQueue 默认构造器，队列容量是Integer.MAX\_VALUE

private static final BlockingQueue<Runnable> sPoolWorkQueue =

new LinkedBlockingQueue<Runnable>();

/\*\*

\* An {@link Executor} that can be used to execute tasks in parallel.

\*/

public static final ThreadPoolExecutor THREAD\_POOL\_EXECUTOR;

static {

Log.i(TAG, "static initializer: " + " CPU\_COUNT = " + CPU\_COUNT + " CORE\_POOL\_SIZE = " + CORE\_POOL\_SIZE + " MAXIMUM\_POOL\_SIZE = " + MAXIMUM\_POOL\_SIZE);

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(

CORE\_POOL\_SIZE, MAXIMUM\_POOL\_SIZE, KEEP\_ALIVE\_SECONDS, TimeUnit.SECONDS,

sPoolWorkQueue, sThreadFactory);

threadPoolExecutor.allowCoreThreadTimeOut(true);

THREAD\_POOL\_EXECUTOR = threadPoolExecutor;

}

public void enqueue(InflateRequest request) {

THREAD\_POOL\_EXECUTOR.execute((new InflateRunnable(request)));

}

}

private static class BasicInflater extends LayoutInflater {

private static final String[] sClassPrefixList = {

"android.widget.",

"android.webkit.",

"android.app."

};

BasicInflater(Context context) {

super(context);

if (context instanceof AppCompatActivity) {

// 手动setFactory2，兼容AppCompatTextView等控件

AppCompatDelegate appCompatDelegate = ((AppCompatActivity) context).getDelegate();

if (appCompatDelegate instanceof LayoutInflater.Factory2) {

LayoutInflaterCompat.setFactory2(this, (LayoutInflater.Factory2) appCompatDelegate);

}

}

}

@Override

public LayoutInflater cloneInContext(Context newContext) {

return new BasicInflater(newContext);

}

@Override

protected View onCreateView(String name, AttributeSet attrs) throws ClassNotFoundException {

for (String prefix : sClassPrefixList) {

try {

View view = createView(name, prefix, attrs);

if (view != null) {

return view;

}

} catch (ClassNotFoundException e) {

// In this case we want to let the base class take a crack

// at it.

}

}

return super.onCreateView(name, attrs);

}

}

private static class InflateRunnable implements Runnable {

private InflateRequest request;

private boolean isRunning;

public InflateRunnable(InflateRequest request) {

this.request = request;

}

@Override

public void run() {

isRunning = true;

try {

request.view = request.inflater.mInflater.inflate(

request.resid, request.parent, false);

} catch (RuntimeException ex) {

// Probably a Looper failure, retry on the UI thread

Log.w(TAG, "Failed to inflate resource in the background! Retrying on the UI"

+ " thread", ex);

}

Message.obtain(request.inflater.mHandler, 0, request)

.sendToTarget();

}

public boolean isRunning() {

return isRunning;

}

}

public InflateRequest obtainRequest() {

InflateRequest obj = mRequestPool.acquire();

if (obj == null) {

obj = new InflateRequest();

}

return obj;

}

public void releaseRequest(InflateRequest obj) {

obj.callback = null;

obj.inflater = null;

obj.parent = null;

obj.resid = 0;

obj.view = null;

mRequestPool.release(obj);

}

public void cancel() {

mHandler.removeCallbacksAndMessages(null);

mHandlerCallback = null;

}

}

X2C — 提升布局加载速度200%：

一般大家在写页面时都是通过xml写布局，通过setContentView、或LayoutInflater.from(context).inflate方法将xml布局加载到内存中

优点

可维护性好

支持即时预览

代码结构清晰

缺点

读取xml很耗时

递归解析xml较耗时

反射生成对象的耗时是new的3倍以上

为了即保留xml的优点，又解决它带来的性能问题，我们开发了X2C方案。即在编译生成APK期间，将需要翻译的layout翻译生成对应的java文件，这样对于开发人员来说写布局还是写原来的xml，但对于程序来说，运行时加载的是对应的java文件。     我们采用APT（Annotation Processor Tool）+ JavaPoet技术来完成编译期间【注解】->【解注解】->【翻译xml】->【生成java】整个流程的操作。

集成使用

1.导入依赖

在module的build.gradle文件添加依赖

annotationProcessor 'com.zhangyue.we:x2c-apt:1.1.2'

implementation 'com.zhangyue.we:x2c-lib:1.0.6'

2.添加注解

在使用布局的任意java类或方法添加注解即可

@Xml(layouts = "activity\_main")

3.配置自定义属性(没有可不配)

在module下建立X2C\_CONFIG.xml文件，里面配置定义属性和方法的映射关系,如果接收者是view，则写view.否则填params.

<x2c-config>

<attr name="app:mixColor" toFunc="view.setMixColor(int)" />

<attr name="android:layout\_marginTop" toFunc="params.topMargin=int" />

</x2c-config>

4.通过X2C加载布局

在原先使用setContentView或inflate的地方替换，如下：

this.setContentView(R.layout.activity\_main); --> X2C.setContentView(this, R.layout.activity\_main);

LayoutInflater.from(this).inflate(R.layout.activity\_main,null); --> X2C.inflate(this,R.layout.activity\_main,null);

过程文件：

原始的xml

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

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

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:paddingLeft="10dp">

<include

android:id="@+id/head"

layout="@layout/head"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_centerHorizontal="true" />

<ImageView

android:id="@+id/ccc"

style="@style/bb"

android:layout\_below="@id/head" />

</RelativeLayout>

生成的java文件

/\*\*

\* WARN!!! dont edit this file

\* translate from {@link com.zhangyue.we.x2c.demo.R.layout.activity\_main}

\* autho chengwei

\* email chengwei@zhangyue.com

\*/

public class X2C\_2131296281\_Activity\_Main implements IViewCreator {

@Override

public View createView(Context ctx, int layoutId) {

Resources res = ctx.getResources();

RelativeLayout relativeLayout0 = new RelativeLayout(ctx);

relativeLayout0.setPadding((int)(TypedValue.applyDimension(TypedValue.COMPLEX\_UNIT\_DIP,10,res.getDisplayMetrics())),0,0,0);

View view1 =(View) new X2C\_2131296283\_Head().createView(ctx,0);

RelativeLayout.LayoutParams layoutParam1 = new RelativeLayout.LayoutParams(ViewGroup.LayoutParams.WRAP\_CONTENT,ViewGroup.LayoutParams.WRAP\_CONTENT);

view1.setLayoutParams(layoutParam1);

relativeLayout0.addView(view1);

view1.setId(R.id.head);

layoutParam1.addRule(RelativeLayout.CENTER\_HORIZONTAL,RelativeLayout.TRUE);

ImageView imageView2 = new ImageView(ctx);

RelativeLayout.LayoutParams layoutParam2 = new RelativeLayout.LayoutParams(ViewGroup.LayoutParams.MATCH\_PARENT,(int)(TypedValue.applyDimension(TypedValue.COMPLEX\_UNIT\_DIP,1,res.getDisplayMetrics())));

imageView2.setLayoutParams(layoutParam2);

relativeLayout0.addView(imageView2);

imageView2.setId(R.id.ccc);

layoutParam2.addRule(RelativeLayout.BELOW,R.id.head);

return relativeLayout0;

}

}

生成的映射文件

/\*\*

\* WARN!!! don't edit this file

\*

\* author chengwei

\* email chengwei@zhangyue.com

\*/

public class X2C127\_activity implements IViewCreator {

@Override

public View createView(Context context) {

View view = null ;

int sdk = Build.VERSION.SDK\_INT;

int orientation = context.getResources().getConfiguration().orientation;

boolean isLandscape = orientation == Configuration.ORIENTATION\_LANDSCAPE;

if (isLandscape) {

view = new com.zhangyue.we.x2c.layouts.land.X2C127\_Activity().createView(context);

} else if (sdk >= 27) {

view = new com.zhangyue.we.x2c.layouts.v27.X2C127\_Activity().createView(context);

} else if (sdk >= 21) {

view = new com.zhangyue.we.x2c.layouts.v21.X2C127\_Activity().createView(context);

} else {

view = new com.zhangyue.we.x2c.layouts.X2C127\_Activity().createView(context);

}

return view;

}

}

不支持

merge标签 ,在编译期间无法确定xml的parent，所以无法支持

系统style,在编译期间只能查到应用的style列表，无法查询系统style，所以只支持应用内style

支持

兼容ButterKnifer

兼容DataBinding

各种系统控件、自定义控件

include标签

viewStub标签

fragment标签

应用style

自定义属性

系统属性

如何监控应用渲染速度：

点击设置—>开发人员选项—>监控—>GPU呈现模式分析，然后选择 在屏幕上显示为条形图 即可以看到一个图表。



1.沿水平轴的每个竖条都代表一个帧，每个竖条的高度表示渲染该帧所花的时间（单位：毫秒）

2.水平绿线表示 16 毫秒。 要实现每秒 60 帧，代表每个帧的竖条需要保持在此线以下。 当竖条超出此线时，可能会使动画出现暂停

再来看下每个竖条的颜色代表什么意思：



分析从哪些方向进行绘制优化：

从GPU呈现模式分析可以看出来，我们能够进行优化的点主要就是测量、布局、绘制、动画和输入处理

测量、布局、绘制过程都会存在自顶而下遍历过程，所以如果布局的层级过多，这会占用额外的CPU资源

当屏幕上的某个像素在同一帧的时间内被绘制了多次（Overdraw），这会浪费大量的CPU以及GPU资源

在绘制过程，也就是onDraw()方法内，我们应该尽量避免局部对象的创建，因为onDraw()方法在绘制过程中会多次调用，大量的局部变量可能会造成内存抖动

合理使用动画，这个本章不做讨论，有兴趣的可以自己了解动画的相关知识

不应该在Event响应的回调中做耗时操作

总结下来视图绘制优化主要要解决的问题就是：

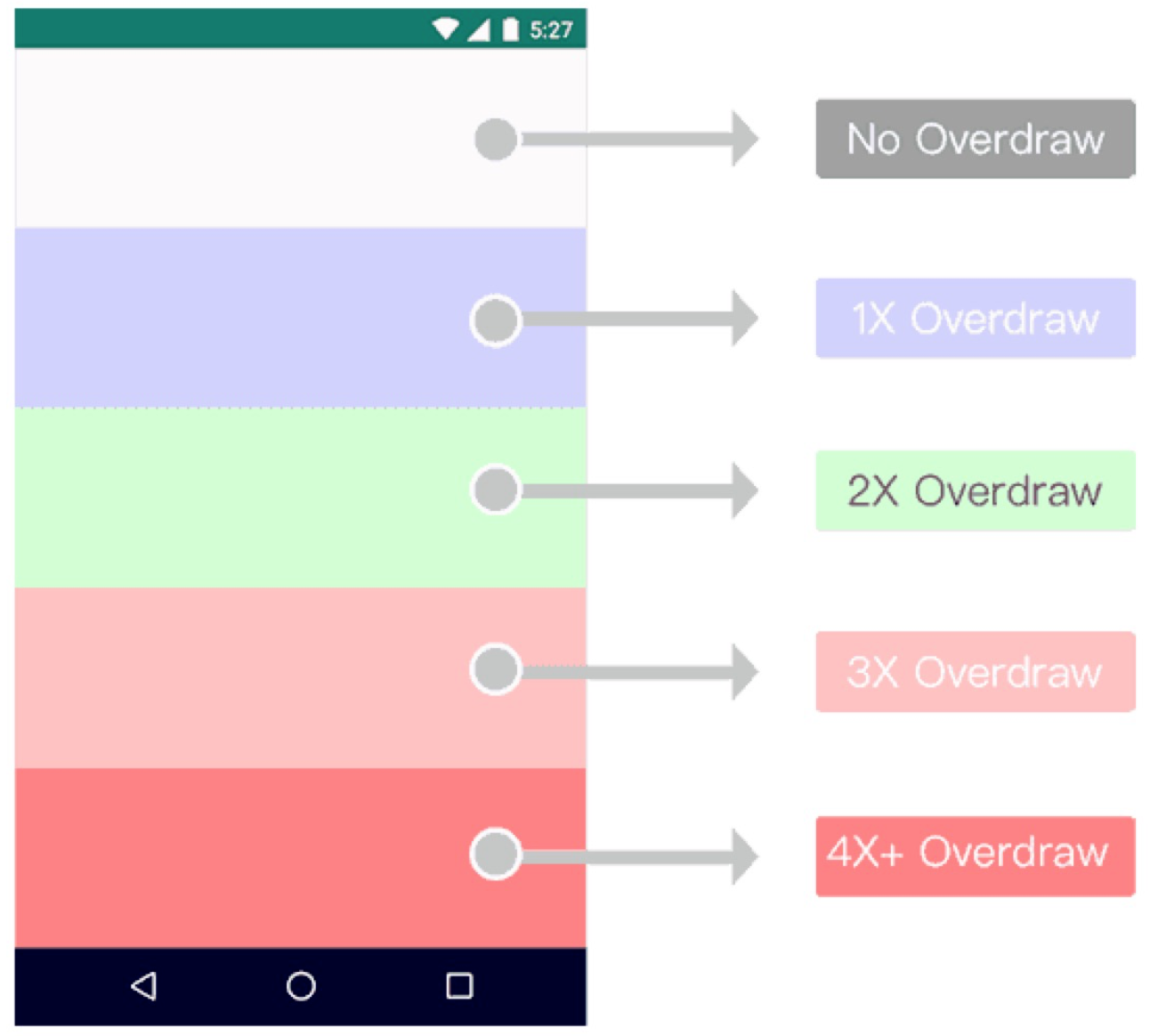
减少view树层级，要宽而浅，避免窄而深

如何检测过度绘制：

点击设置—>开发人员选项—>硬件—>调试GPU过度绘制，然后选择 显示过度绘制区域 即可以看到一个图表，如下图所示



再来看下每种颜色代表什么意思：



有些过度绘制是无法避免的。但是在优化界面时，应该尽量让大部分的界面显示为原色（即无过度绘制）或者为蓝色（仅有 1 次过度绘制）。如果出现粉色或者红色，应该查看代码看看能否尽量避免

如何避免过度绘制：

移除window的背景

一般情况下我们的AppTheme都默认带会有windowBackground

<style name="AppTheme" parent="Theme.AppCompat.Light">

...

<item name="android:windowBackground">@color/background\_material\_light</item>

...

</style>

但是这个windowBackground大部分清洁下都是没有什么意义的，因为我们往往都会在布局文件中设置我们当前view的背景颜色。如果我们同时设置了windowBackground和布局文件中的background，那就会出现两次绘制，这显然是没有什么意义的，因为最终用户看到的颜色还是以background为准

我们可以通过下面两个方法来解决这个问题

在xml中设置

<item name="android:windowBackground">@null</item>

通过代码设置

getWindow().setBackgroundDrawable(null);

移除控件中不需要的背景

例子：

列表页（RecyclerView） 与 其内子控件（Item）的背景相同，故可移除子控件（Item）布局中的背景

对于1个ViewPager+多个 Fragment 组成的首页界面，若每个Fragment 都设有背景色，即 ViewPager 则无必要设置，可移除

所以对于控件背景颜色的设置基本可以归纳为以下两个原则：

对于子控件，如果其背景颜色跟父布局一致，那么就不用再给子控件添加背景了

如果子控件背景五颜六色，且能够完全覆盖父布局，那么父布局就可以不用添加背景了

减少透明度的使用

对于不透明的view，只需要渲染一次即可把它显示出来。但是如果这个view设置了alpha值，则至少需要渲染两次。这是因为使用了alpha的view需要先知道混合view的下一层元素是什么，然后再结合上层的view进行Blend混色处理。透明动画、淡入淡出和阴影等效果都涉及到某种透明度，这就会造成了过度绘制。可以通过减少渲染这些透明对象来改善过度绘制。比如：在TextView上设置带透明度alpha值的黑色文本可以实现灰色的效果。但是，直接通过设置灰色的话能够获得更好的性能

使用ConstraintLayout减少布局层级

ConstraintLayout，可以翻译为约束布局，在2016年Google I/O 大会上发布。ConstraintLayout相比RelativeLayout，其性能更好，也更容易使用。连官方的hello world都用ConstraintLayout来写了。所以极力推荐使用ConstraintLayout来编写布局

使用merge标签减少布局层级

我们通过两个例子来认识merge标签

自定义view时使用merge标签

比如我们现在要写一个自定义viewGroup继承自RelativeLayout

public class MyViewGroup extends RelativeLayout {

public MyView(Context context) {

this(context, null, 0);

}

public MyView(Context context, AttributeSet attrs) {

this(context, attrs, 0);

}

public MyView(Context context, AttributeSet attrs, int defStyleAttr) {

super(context, attrs, defStyleAttr);

initView();

}

public void initView() {

LayoutInflater.from(getContext()).inflate(R.layout.layout\_my\_view, this, true);

}

}

我们通过LayoutInflater将XML加载出view并添加到这个自定义view的根布局中，这时候我们的XML文件就可以这么写。我们在根布局中使用了merge标签，就代表这个xml文件的根布局就是其parent，也就是我们上面的MyViewGroup，这样相比在根布局中使用RelativeLayout就减少了一个布局层级

这里有一个细节需要注意：当我们使用merge标签时，如果我们希望在Design窗口中实时预览布局效果，我们需要使用 tools:parentTag="android.widget.RelativeLayout"来告诉AndroidStudio你的父布局是什么

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

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

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

tools:parentTag="android.widget.RelativeLayout">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_centerInParent="true"

android:text="hello world" />

</merge>

有时候我们会通 过include标签来提高布局的复用性，如果layout\_include\_xx.xml的布局和其父布局使用的是同一个布局类型，如线性布局等。这时候就可以在layout\_include\_xx.xml中使用merge标签来减少布局层级

使用ViewStub标签延迟加载

ViewStub是一个不可见的View类，用于在运行时按需懒加载资源，只有在代码中调用了viewStub.inflate()或者viewStub.setVisible(View.visible)方法时才内容才变得可见。这里需要注意的一点是，当ViewStub被inflate到parent时，ViewStub就被remove掉了，即当前view hierarchy中不再存在ViewStub，而是使用对应的layout视图代替

<ViewStub

android:id="@+id/view\_stub"

android:layout="@layout/layout\_error\_view"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

通常用于不常使用的控件，如

网络请求失败的提示

列表为空的提示

新内容、新功能的引导，因为引导基本上只显示一次

又或者我们写了一个通用的自定义 View，但其中部分子 View 只在部分情况下才显示

ViewStub标签使用注意点：

ViewStub标签不支持merge标签。因此这有可能导致加载出来的布局存在着多余的嵌套结构，具体如何去取舍就要根据各自的实际情况来决定了

ViewStub的inflate只能被调用一次,第二次调用会抛出异常

虽然ViewStub是不占用任何空间的，但是每个布局都必须要指定layout\_width和layout\_height属性，否则运行就会报错

减少自定义View的过度绘制,使用clipRect()

下面我们自定义一个View用来显示多张重叠的图片，效果图如下：



其onDraw()方法也很简单，就是遍历所有图片，然后绘制出来：

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

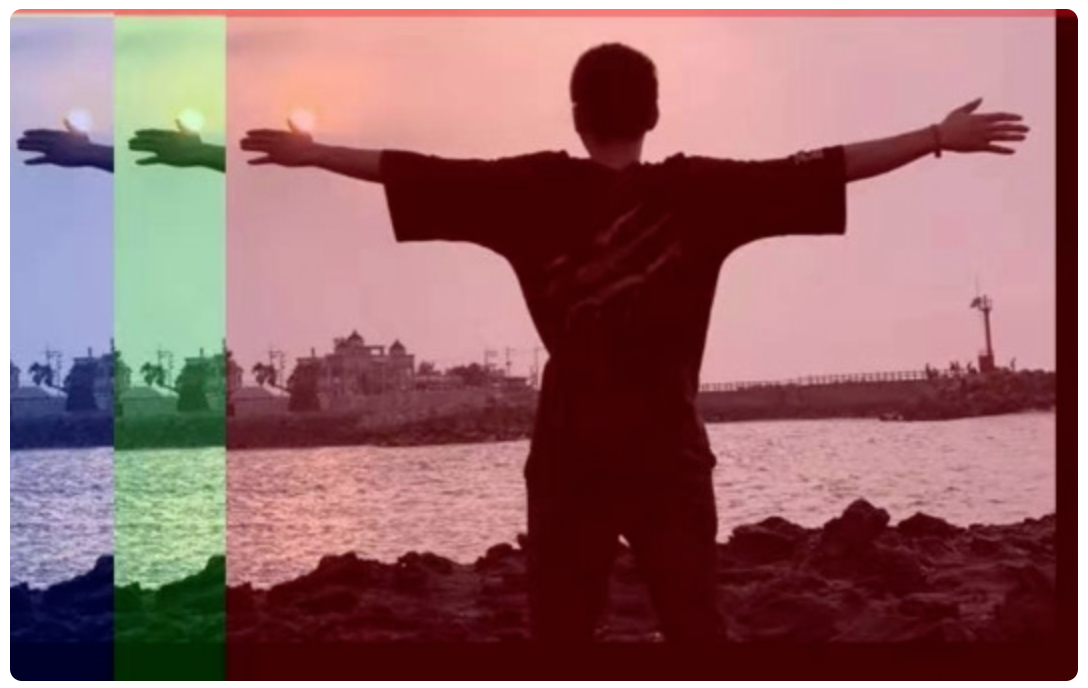
for (int i = 0; i < imgs.length; i++) {

canvas.drawBitmap(imgs[i], i \* 100, 0, mPaint);

}

}

显示过度绘制区域：



过度绘制比较严重，那么如何解决？

我们先来分析一下为什么会出现过度绘制：以第一张图为例，上面的代码会把整张图都绘制出来了，第二张在第一张上面继续绘制，这就造成了过度绘制

那么，解决办法也很简单，对于前面的n-1张图，我们只需要绘制一部分即可，对于最后一张才绘制完整的。

Canvas中的clipRect()方法能够设置一个裁剪矩形，只在这个矩形区域内的内容才能够绘制出来

优化后的代码如下：

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

for (int i = 0; i < imgs.size(); i++) {

canvas.save();

if (i < imgs.size() - 1) {

//前面的n-1张图，只裁剪一部分

canvas.clipRect(i \* 100, 0, (i + 1) \* 100, imgs.get(i).getHeight());

} else if (i == imgs.size() - 1) {

//最后一张，完整的

canvas.clipRect(i \* 100, 0, i \* 100 + imgs.get(i).getWidth(), imgs.get(i).getHeight());

}

canvas.drawBitmap(imgs.get(i), i \* 100, 0, mPaint);

canvas.restore();

}

}

优化后的效果图如下：



所有区域都是蓝色的，即只有1次过度绘制。

Canvas除了clipRect()方法外，还有clipPath()等方法，优化时选择合理的方法去裁剪即可