Android面试题之RemoteView，包括RemoteView在通知栏和桌面小部件上的使用、RemoteViews的内部原理、RemoteViews的意义。

> 本文是我一点点归纳总结的干货，但是难免有疏忽和遗漏，希望不吝赐教。
> 转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81463442

有帮助的话请点个赞！万分感谢！

Android面试题-RemoteView

版本：2018/8/6-1(深夜)

原文参考链接：
http://blog.csdn.net/feather_wch/article/details/79175518

1、RemoteViews是什么?

> 1. 是一种远程View，表示一个View结构
> 2. 可以在其他进程中显示，也提供了相应的基础操作
> 3. Android中有两种使用场景: 通知栏、桌面小部件

2、RemoteViews在应用中的要点

> 1. 通知栏开发中主要是通过 `NotificationManager` 的 `notify` 方法来实现，也支持自定义布局
> 2. 桌面小部件主要是通过 `AppWidgetProvider` 来实现的， `AppWidgetProvider` 本质上是一个广播
> 3. 两个使用场景都是运行在系统的 `SystemServer` 进程中，为了跨进程更新界面， `RemoteViews` 提供一些列 `set` 方法
> 4. `RemoteViews` 中支持的View类型是有限的

RemoteViews在通知栏上的应用

3、系统默认样式通知栏是如何实现的?

> 1. 创建PendingIntent，用于点击通知栏跳转
> 2. 实例化Notification (By 建造者模式)
> 3. 获取notificationManager实例
> 4. 通过notificationManager发送通知

4、RemoteViews实现自定义通知:

> 1. 通过自定义布局创建RemoteViews
> 2. 设置远程的View控件
> 3. 设置远程的点击事件(通过PendingIntent实现)
> 4. 创建Notification并且设置RemoteViews和PendingIntent(点击效果)
> 5. 获取notificationManager实例
> 6. 通过notificationManager发送通知

RemoteViews在桌面小部件上的应用

5、AppWidgetProvider是什么?

> 1. 实现桌面小部件的类
> 2. 直接继承自 `BroadcastReceiver`
> 3. 实际开发中直接当成 `BroadcastReceiver`

6、AppWidgetProvider实现步骤

> 1. 自定义 `Widget` 的布局(样式)

2. 定义 `Widget` 的规格大小
3. 实现 `Widget` 继承自 `AppWidgetProvider` (本质是广播)
4. `AndroidManifest` 中进行注册

7、Wdiget小部件的minWidth/minHeight的尺寸

单元格个数(行或列)	对应设置大小(dp)(minWidth或minHeight)
1	40dp
2	110dp
3	180dp
4	250dp
...	...
n	70 * n - 30

PendingIntent

8、PendingIntent是什么?

1. 表示一种pending状态的意图 `Intent` -也就是指待定、等待、即将发生的意图
2. PendingIntent就在将来的某个不确定的时刻发生
3. Intent是立即发生
4. PengdingIntent典型场景就是给RemoteViews添加单击事件(无法通过setOnClickListener的方式添加)
5. PengdingIntent有三种待定意图：启动Activity、启动Service和发送广播

9、PengdingIntent的三种待定意图

意图	方法	备注
Activity	getActivity(Context context, int requestCode, Intent intent, int flags)	该意图发生时，等效于 `Context.startActivity(Intent)`
Service	getService(Context context, int requestCode, Intent intent, int flags)	该意图发生时，等效于 `Context.startService(Intent)`
Broadcast	getBroadcast(Context context, int requestCode, Intent intent, int flags)	该意图发生时，等效于 `Context.startBroadcast(Intent)`

10、PendingIntent的匹配规则(何时两个PendingIntent相同)

1. 匹配规则：当两个PendingIntent内部的 `Intent` 相同，并且 `requestCode` (发送方请求码)相同，则两者为同一个 `PendingIntent` .
2. `requestCode` 相同就是单纯的数值相同
3. `Intent` 的相同必须满足：两个Intent的 `ComponentName` 和 `intent-filter` 相同，则两者相同。因为 `extras` 不参与匹配过程，因此 `extras` 可以不同

11、Intent匹配规则中 `ComponentName` 是什么?

1. `ComponentName` 是指组件名，该类可以用于定义组件-如四大组件
2. 通过给 `Intent` 设置 `ComponentName` ，可以进行如启动一个Activity等操作

```
// 通过ComponentName启动Activity
ComponentName componentName=new ComponentName("com.hao.myapplication", "com.hao.myapplication.Main2Activity");
Intent intent = new Intent();
intent.setComponent(componentName);
startActivity(intent);
```

12、PendingIntent的flags参数的具体含义

Flags	含义	备注
FLAG_ONE_SHOT	当前的PengdingIntent只能被使用一次， 然后被自动cancel，如果有后续相同PendingIntent， 它们的 `send` 都会调用失败	若通知栏消息被这样标记， 则同类通知中只要有某一条消息被单击打开， 其他以及后续的通知都将无法再单击打开
FLAG_NO_CREATE	当前PendingIntent不会主动创建： 若当前PendingIntent之前不存在， 则 三大方法 会直接返回null	极少见，一般没什么用
FLAG_CANCEL_CURRENT	当前PendingIntent如果已经存在，则都 `cancel` 掉， 然后会创建新的PendingIntent	通知栏中只有最新的通知可以打开， 之前弹出的通知均无法打开

Flags	含义	备注
FLAG_UPDATE_CURRENT	当前PendingIntent如果已经存在， 则更新它们 `Intent` 中的 `extras`	之前弹出的通知的PendingIntent都被更新， 且都跟最新一条完全一致(包括extras)， 所有消息都可以打开

> 补充：通知栏消息的 `manager.notify(1, notification)` 中，如果ID一致，后续的通知都会直接替换前面的通知；如果ID不一致，且PendingIntent不匹配，则所有消息互不干扰；若PendingIntent匹配，则会根据上面的4种情况产生不同的结果。

RemoteViews的内部机制

13、RemoteViews支持的所有布局和View有哪些?

类型	详情
View(禁止任何自定义View)	Button/ImageButton/ImageView/TextView/ListView/GridView/ProgressBar/AnalogClock/StackView/AdapterViewFlipper/
Layout	FrameLayout/LinearLayout/RelativeLayout/GridLayout

14、RemoteViews内部原理分析

> 1. `RemoteViews` 会通过 `Binder` 传递到 `SystemServer` 进程(RemoteViews实现了 `Parcelable` 接口)
> 2. 系统会根据 `RemoteViews` 中的包名等信息，去获取该应用的资源
> 3. 会通过 `LayoutInflater` 去加载 `RemoteViews` 中的布局文件
> 4. 对于 `SystemServer` 进程，加载好布局后，就是一个普通的View。然而对于我们的进程是一个 `RemoteViews`
> 5. 接着系统会对View进行UI刷新(提交的一系列set方法)
> 6. `set方法` 的操作不会立即执行，RemoteViews内部会记录所有的UI更新操作，在 `RemoteViews` 被加载之后才会执行
> 7. 最终RemoteViews完成了在 `SystemServer` 进程中的显示。后续的UI更新，就是通过 `NotificationManager` 和 `AppWidgetManager` 提交到 `SystemServer` 进程中完成相应操作

15、RemoteViews的set等操作内部细节

> 1. 系统并没有通过 `Binder` 去支持 `View` 的跨进程访问
> 2. `RemoteViews` 提供了一种 `Action` 的概念， `Action` 实现了 `Parcelable` 接口
> 3. 系统将 `RemoteViews` 的一系列操作封装到 `Action` 对象中，并将 `Action` 跨进程传输到 `SystemServer` 进程，最后在远程进程中执行 `Action` 对象中的所有操作。
> 4. 每调用一次 `set` 方法， `RemoteViews` 中就会添加对应的 `Action` 对象，最终会传到远程进程中。
> 5. 远程进程通过 `RemoteViews` 的apply方法进行View的更新操作(遍历所有Action对象，并调用其apply方法)

16、RemoteViews内部机制的优点

> 1. 不需要定义大量的 `Binder` 接口
> 2. 通过在远程进程中的批量操作，避免了大量的 `IPC` 操作，提高性能

RemoteViews源码分析

17、RemoteViews的set方法源码要点

```
public void setTextViewText(int viewId, CharSequence text) {
    setCharSequence(viewId, // view的id
                "setText", // 方法名
                    text);
}
public void setCharSequence(int viewId, String methodName, CharSequence value) {
    //1. 添加‘反射类型Action’对象
    addAction(new ReflectionAction(viewId, methodName, ReflectionAction.CHAR_SEQUENCE, value));
}
private void addAction(Action a) {
    //...
    //1. 添加Action到Action链表中
    if (mActions == null) {
        mActions = new ArrayList<Action>();
    }
    mActions.add(a); //添加action
    a.updateMemoryUsageEstimate(mMemoryUsageCounter); //更新内存使用状态
}
```

18、RemoteViews的apply方法源码要点

```
    public View apply(Context context, ViewGroup parent, OnClickHandler handler) {
        RemoteViews rvToApply = getRemoteViewsToApply(context);
        //1. 去加载RemoteViews中的布局文件
        View result = inflateView(context, rvToApply, parent);
        loadTransitionOverride(context, handler);
        //2. 通过performApply执行一些更新操作
        rvToApply.performApply(result, parent, handler);
        return result;
    }
    private void performApply(View v, ViewGroup parent, OnClickHandler handler) {
        if (mActions != null) {
            handler = handler == null ? DEFAULT_ON_CLICK_HANDLER : handler;
            final int count = mActions.size();
            //1. 遍历Actions链表，并执行Action的apply方法(真正操作View)
            for (int i = 0; i < count; i++) {
                Action a = mActions.get(i);
                a.apply(v, parent, handler);
            }
        }
    }
```

19、RemoteViews的apply和reapply的区别?

> 1. 例如 `AppWidgetManager` 的 `updateAppWidget` 的内部实现中，是通过 `RemoteViews` 的 `apply`和`reApply` 加载或更新界面
> 2. apply: 加载布局并且更新界面
> 3. reApply：只会更新界面
> 4. 通知栏和桌面小部件初始化时会调用 `apply` ， 后续更新都调用 `reapply`

20、ReflectionAction源码解析:

```
private final class ReflectionAction extends Action {
        //省略成员变量和parcelable相关方法
        //1. 构造方法
        ReflectionAction(int viewId, String methodName, int type, Object value) {
            this.viewId = viewId;
            this.methodName = methodName;
            this.type = type;
            this.value = value;
        }
        @Override
        //1. 反射动作，对View的操作，会以反射的方式调用
        public void apply(View root, ViewGroup rootParent, OnClickHandler handler) {
            final View view = root.findViewById(viewId);
            if (view == null) return;
            //2. 获取参数的类型(如String.class等等)
            Class<?> param = getParameterType();
            if (param == null) {
                throw new RemoteViews.ActionException("bad type: " + this.type);
            }
            //3. getMethod根据方法名来得到反射所需的Method对象
            try {
                //4. invoke给View设置上值value
                getMethod(view, this.methodName, param).invoke(view, wrapArg(this.value));
            } catch (RemoteViews.ActionException e) {
                throw e;
            } catch (Exception ex) {
                throw new RemoteViews.ActionException(ex);
            }
        }
        //...省略...
}
```

> 1. ReflectionAction使用反射实现
> 2. TextViewSizeAction就比较简单，没有使用反射来实现

20、RemoteViews中的PendingIntent的点击事件

分类	作用
setOnClickPendingIntent	给普通View设置点击事件，禁止给(ListView、StackView)等集合中的View设置点击事件(开销太大)
setPendingIntentTemplate/setOnClickFillIntent	组合使用-用于给ListView等View中的item添加点击事件