

BUILDING MY OWN DEBUGGING TOOL ON OVERLAY

About me

- ▶ 横幕圭真 (Keishin Yokomaku)



Drivemode, Inc. / エンジニア



Keith Yokoma: [GitHub](#) / [Twitter](#) / [キータ](#) / [Stack Overflow](#)

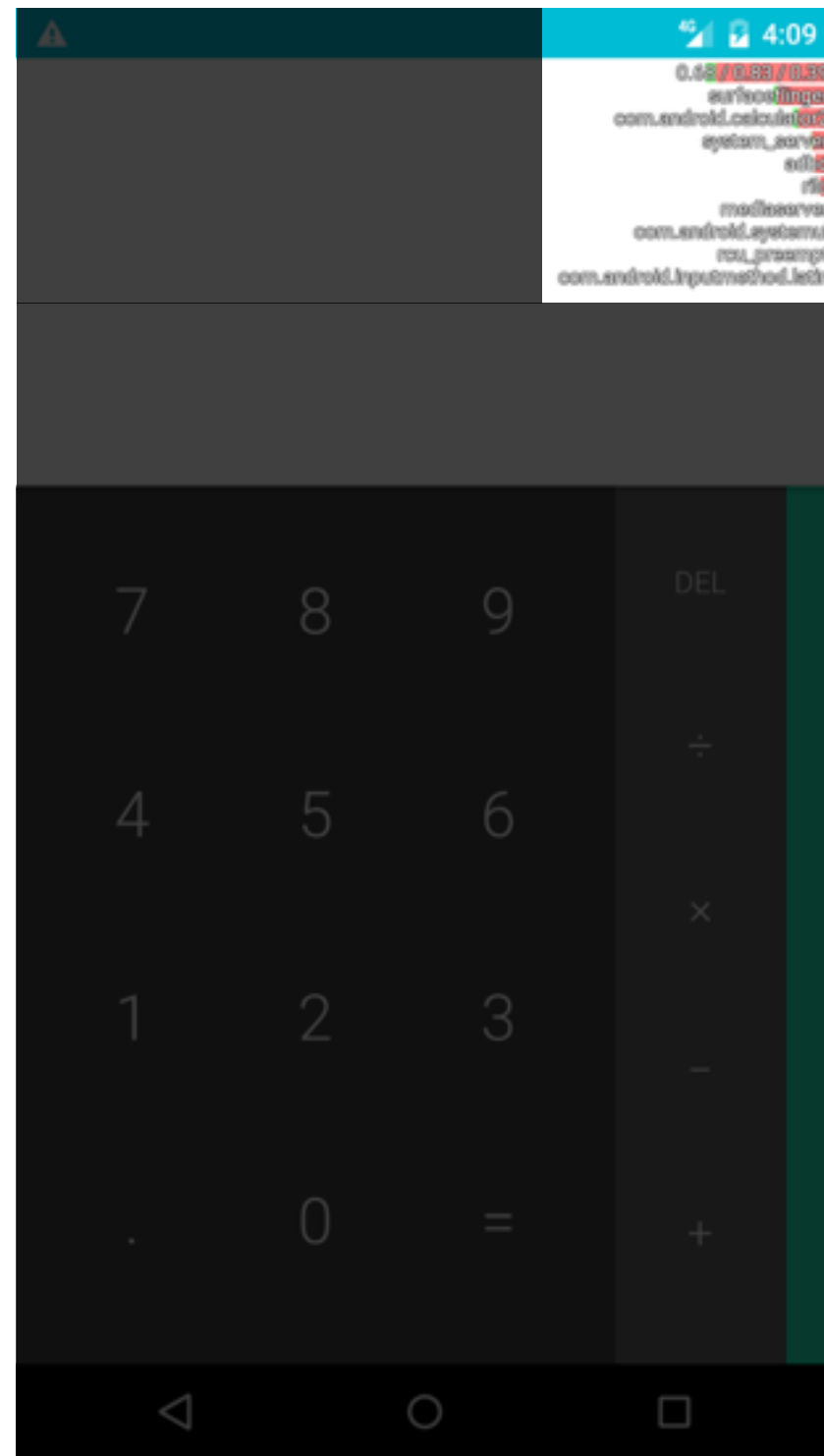
- ▶ Books: [Android アカデミア](#) / [AZ 異本](#) / [なないろ Android](#)
- ▶ Fun: 器械体操 / 自転車 / 写真 / 筋トレ / 🍺 / 🍶
- ▶ Today's Quote: "Happy DroidKaigi!"

BUILDING MY OWN DEBUGGING TOOL ON OVERLAY



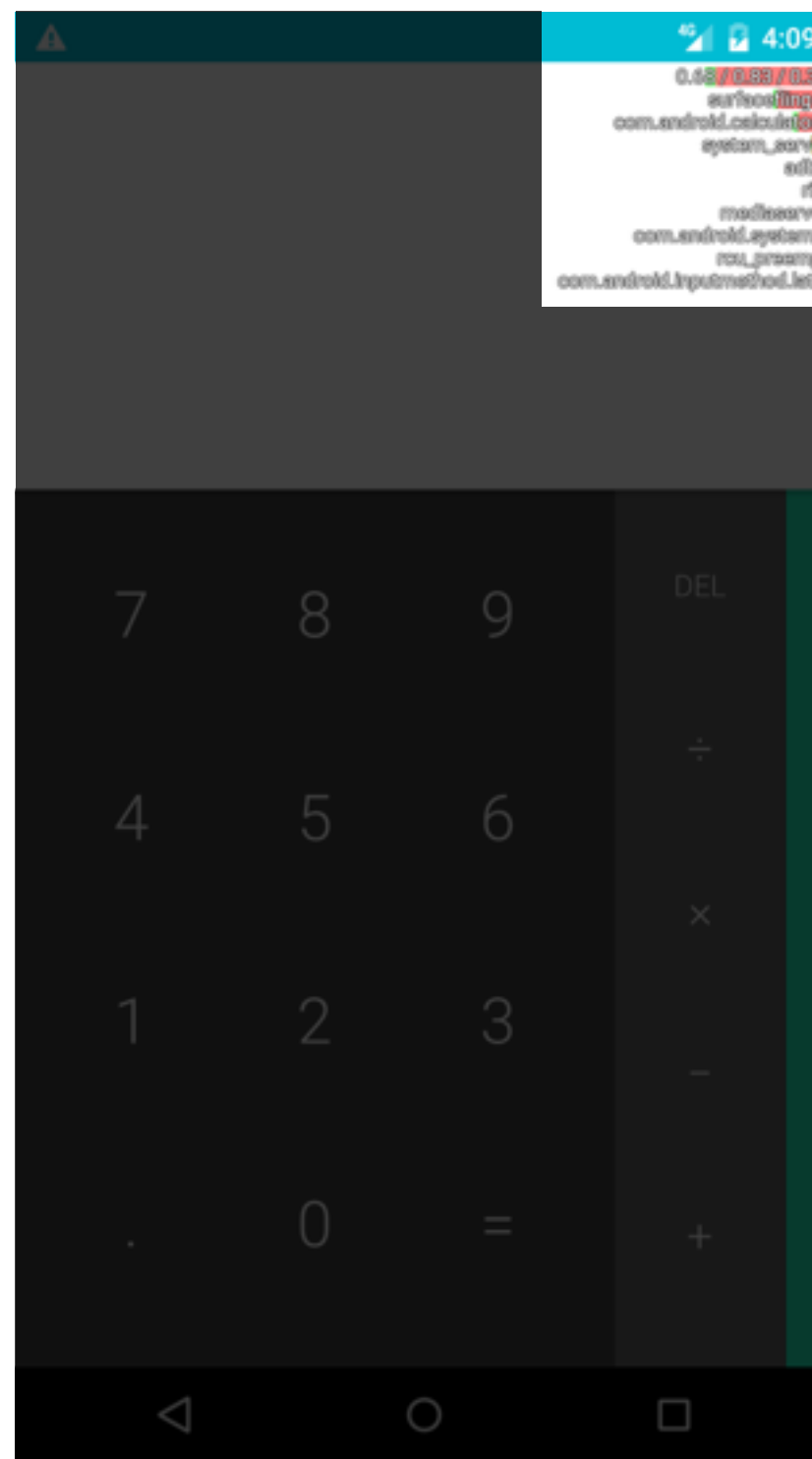
さいしょに

- ▶ このトークには若干の黒魔術的内容が含まれています
- ▶ 応用する場合、用法用量を守って楽しくお使いください



これは何？

- ▶ 常に手前に表示され続ける View
 - ▶ 開発者オプション
 - ▶ どのアプリを開いていても見える
 - ▶ 定期的に情報が更新され続ける

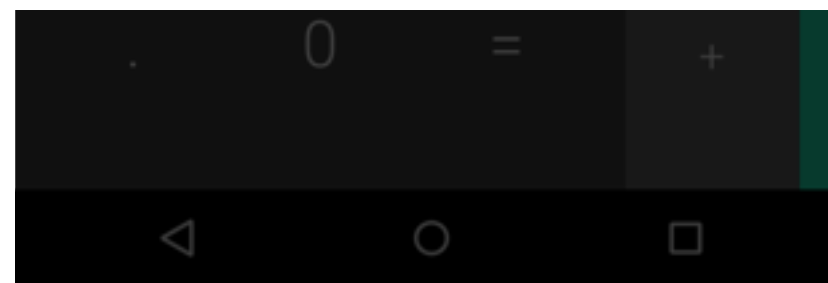


これは何？

- ▶ 常に手前に表示され続ける View



自分で作りたい 😊



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ **Activity などとつなぎこむ**



本題に入る前に

- ▶ DroidKaigi アプリにサンプル実装を組み込みました
 - ▶ リポジトリ: <http://bit.ly/2IGA6Aj>
 - ▶ 設定 -> Developer Menu -> Debug Overlay View
 - ▶ 言語設定を変更したり、画面回転すると...

特別なレイヤに
VIEW を表示する



登場するクラス

- ▶ [android.view.WindowManager](#)
 - ▶ システムサービス
 - ▶ View を追加・削除・更新するメソッドをもつ
- ▶ [android.view.WindowManager.LayoutParams](#)
 - ▶ WindowManager で描画する View のためのパラメータ

登場するクラス

- ▶ **android.view.WindowManager**
 - ▶ システムサービス
 - ▶ View を追加・削除・更新するメソッドをもつ
- ▶ android.view.WindowManager.LayoutParams
 - ▶ WindowManager で描画する View のためのパラメータ

登場するクラス

- ▶ android.view.WindowManager
 - ▶ システムサービス
 - ▶ View を追加・削除・更新するメソッドをもつ
- ▶ **android.view.WindowManager.LayoutParams**
 - ▶ WindowManager で描画する View のためのパラメータ

WindowManager とパーミッション

- ▶ 特別なパーミッション
 - ▶ “他のアプリに重ねて表示”
 - ▶ `<uses-permission>`
 - ▶ `android.permission.SYSTEM_ALERT_WINDOW`
- ▶ インストール時に自動で許可される
- ▶ あとで許可を取り下げることができる

WindowManager とパーミッション

- ▶ 設定画面を開く Intent のアクション
 - ▶ [android.settings.action.MANAGE_OVERLAY_PERMISSION](#)
- ▶ 許可があるかどうかチェックするメソッド
 - ▶ [Settings.canDrawOverlays\(Context\)](#)
- ▶ パーミッションを取ったアプリのプロセスが生きていると、他のアプリのインストールボタンが押せなくなることに注意

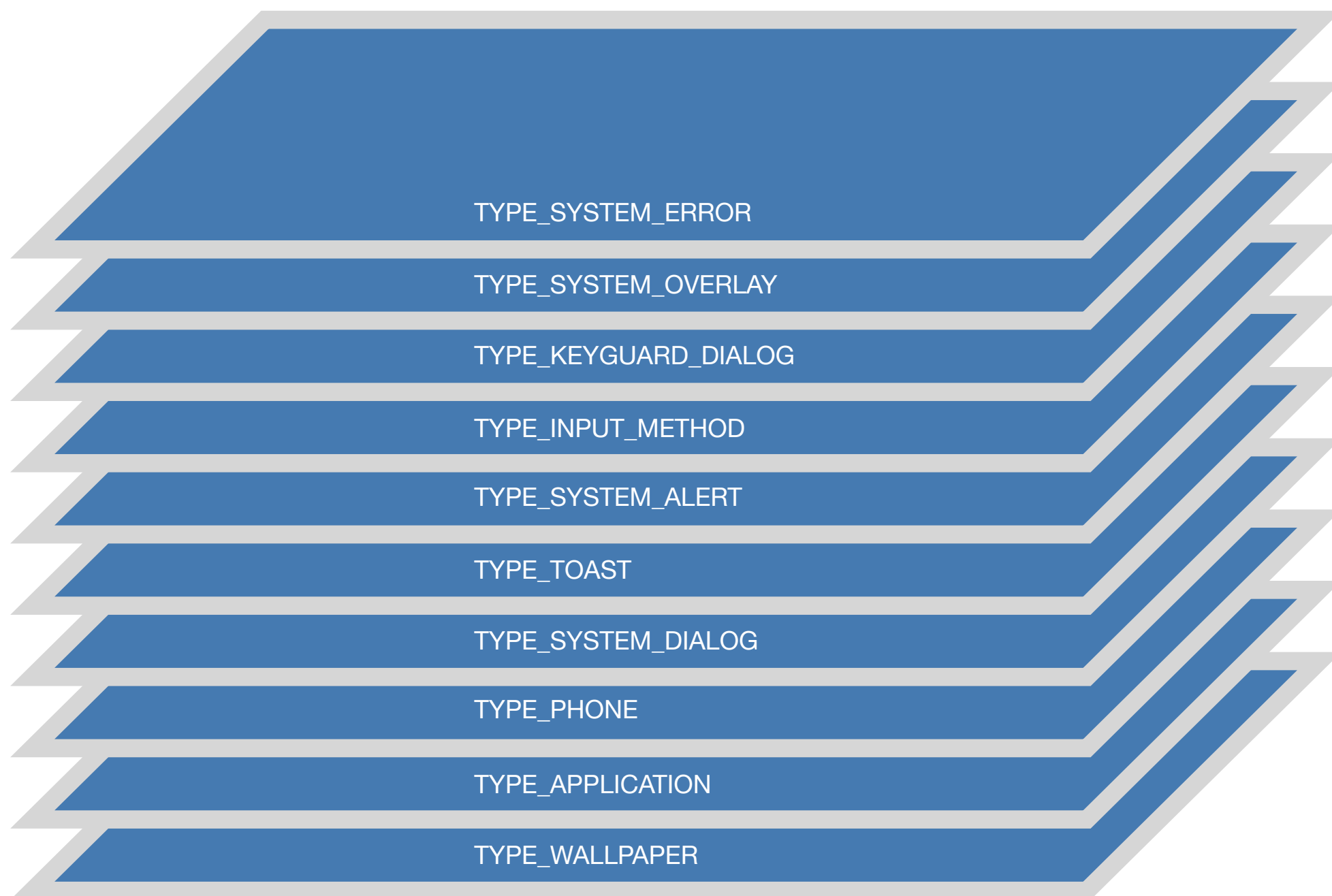
WindowManager のメソッド

- ▶ [WindowManager#addView\(View, LayoutParams\)](#)
 - ▶ View の追加
- ▶ [WindowManager#removeView\(View\)](#)
 - ▶ View の削除
- ▶ [WindowManager#updateViewLayout\(View, LayoutParams\)](#)
 - ▶ View のパラメータ更新

WindowManager で使うパラメータ

- ▶ WindowManager.LayoutParams
 - ▶ 縦・横のサイズ
 - ▶ レイヤ
 - ▶ フラグ
 - ▶ ピクセルフォーマット

WindowManager のレイヤ



WindowManager のレイヤ

多すぎる 😂😂

TYPE_NONE

TYPE_APPLICATION

TYPE_WALLPAPER

(多分)よく使うWindowManager のレイヤ



TYPE_SYSTEM_OVERLAY

TYPE_SYSTEM_ALERT

TYPE_SYSTEM_DIALOG

TYPE_APPLICATION

(多分)よく使うWindowManager のレイヤ

▶ TYPE SYSTEM OVERLAY

- ▶ ロック画面の上。タッチイベントを拾ってはダメ。

▶ TYPE SYSTEM ALERT

- ▶ Toast よりも上。

▶ TYPE SYSTEM DIALOG

- ▶ アプリケーションよりも上。

WindowManager のフラグ

- ▶ WindowManager に追加する View の振る舞いを決める
 - ▶ タッチイベントやフォーカスの取得をするかどうか
 - ▶ 領域外への描画の可否

タッチイベントやフォーカスのフラグ

▶ FLAG_NOT_FOCUSABLE

- ▶ フォーカスを奪わない
- ▶ FLAG_NOT_TOUCH_MODAL も自動で追加される

▶ FLAG_NOT_TOUCH_MODAL

- ▶ WindowManager に追加した View の領域外のタッチイベントを背後のアプリに流す

タッチイベントやフォーカスのフラグ

▶ FLAG_NOT_TOUCHABLE

- ▶ タッチイベントをすべて背後のアプリに流す

領域外への描画の可否を決めるフラグ

▶ FLAG_LAYOUT_IN_SCREEN

- ▶ 画面のサイズが View の大きさの限界値になる

▶ FLAG_LAYOUT_NO_LIMITS

- ▶ 画面サイズを超えて View の大きさを設定できる
- ▶ 左上を原点として、マイナス方向にも View を配置できる

ピクセルフォーマット

- ▶ 描画する View のピクセルがもつアルファ値
 - ▶ [android.graphics.PixelFormat](#)
 - ▶ [OPAQUE](#): アルファ値なし・不透明
 - ▶ [TRANSLUCENT](#): 複数ビットでアルファ値を表現
 - ▶ [TRANSPARENT](#): 最低1ビットでアルファ値を表現

View の追加コード

```
public class MainActivity extends Activity{
    private View mOverlayView;
    private WindowManager mWindowManager;

    @Override
    protected void onCreate(Bundle icicle) {
        // .....
        WindowManager.LayoutParams p = new WM.LP(
            WRAP_CONTENT, WRAP_CONTENT, TYPE_SYSTEM_DIALOG,
            FLAG_NOT_FOCUSABLE, PixelFormat.TRANSLUCENT
        );
        mWindowManager.addView(mOverlayView, p);
    }
}
```

View の追加コード

```
public class MainActivity extends Activity{  
    private View mOverlayView;  
    private WindowManager mWindowManager;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        // .....  
        WindowManager.LayoutParams p = new WindowManager.LayoutParams(  
            WRAP_CONTENT, WRAP_CONTENT, TYPE_SYSTEM_DIALOG,  
            FLAG_NOT_FOCUSABLE, PixelFormat.TRANSLUCENT  
        );  
        mWindowManager.addView(mOverlayView, p);  
    }  
}
```


View の追加コード

```
public class MainActivity extends Activity{
    private View mOverlayView;
    private WindowManager mWindowManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // .....
        WindowManager.LayoutParams p = new WM.LayoutParams(
            WRAP_CONTENT, WRAP_CONTENT, TYPE_SYSTEM_DIALOG,
            FLAG_NOT_FOCUSABLE, PixelFormat.TRANSLUCENT
        );
        mWindowManager.addView(mOverlayView, p);
    }
}
```

View の追加コード

```
public class MainActivity extends Activity{  
    private View mOverlayView;  
    private WindowManager mWindowManager;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        // .....  
        WindowManager.LayoutParams p = new WM.LayoutParams(  
            WRAP_CONTENT, WRAP_CONTENT, TYPE_SYSTEM_DIALOG,  
            FLAG_NOT_FOCUSABLE, PixelFormat.TRANSLUCENT  
        );  
        mWindowManager.addView(mOverlayView, p);  
    }  
}
```

View の追加コード

```
public class MainActivity extends Activity{
    private View mOverlayView;
    private WindowManager mWindowManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // .....
        WindowManager.LayoutParams p = new WM.LayoutParams(
            WRAP_CONTENT, WRAP_CONTENT, TYPE_SYSTEM_DIALOG,
            FLAG_NOT_FOCUSABLE, PixelFormat.TRANSLUCENT
        );
        mWindowManager.addView(mOverlayView, p);
    }
}
```

View の追加コード

```
public class MainActivity extends Activity{
    private View mOverlayView;
    private WindowManager mWindowManager;

    @Override
    protected void onCreate(Bundle icicle) {
        // .....
        WindowManager.LayoutParams p = new WM.LP(
            WRAP_CONTENT, WRAP_CONTENT, TYPE_SYSTEM_DIALOG,
            FLAG_NOT_FOCUSABLE, PixelFormat.TRANSLUCENT
        );
        mWindowManager.addView(mOverlayView, p);
    }
}
```

View の削除コード

```
public class MainActivity extends Activity{
    private View mOverlayView;
    private WindowManager mWindowManager;

    @Override
    protected void onDestroy() {
        // .....
        mWindowManager.removeView(mOverlayView) ;
    }
}
```

View の削除コード

```
public class MainActivity extends Activity{  
    private View mOverlayView;  
    private WindowManager mWindowManager;  
  
    @Override  
    protected void onDestroy() {  
        // .....  
        mWindowManager.removeView(mOverlayView) ;  
    }  
}
```

描画する View の管理

- ▶ WindowManager は View の追加・削除・更新しかしない
 - ▶ どのタイミングで追加・削除・更新するかは自分で管理
 - ▶ WindowManager に findViewById のようなものはない
 - ▶ View のインスタンスをメンバーに持っておかないと削除できなくなる
 - ▶ View のプロパティを変更するためにも必要

全ては View である



他のアプリにも View を重ねたい？

- ▶ Activity で管理する場合は自分のアプリにしか重ならない
 - ▶ ライフサイクルを超えた View の管理をする必要性
 - ▶ View 自体は Activity とは独立している
 - ▶ ライフサイクルを超えられる View の管理方法？
 - ▶ Service

他のアプリにも View を重ねたい？

- ▶ Activity で管理する場合は自分のアプリにしか重ならない
 - ▶ ライフサイクルを超えた View の管理をする必要性
 - ▶ View 自体は Activity とは独立している
- ▶ ライフサイクルを超えられる View の管理方法？
 - ▶ **Service**

裏で VIEW を
更新し続ける



バックグラウンドで View を表示し続けるには

- ✓ View は Service でも生成できる
 - ▶ Service はメインスレッドで動作する
- ✓ Activity とはことなるライフサイクルで動かす
 - ▶ バックグラウンドで常に動き続けられる
- ➡ Service で View を WindowManager に渡せば、他のアプリに重ねて表示し続けられる

Service で View を描画するときの設計

- ▶ 描画する View のライフサイクル管理
 - ▶ **Activity がしてくれることを Service で再実装する**
 - ◎ ライフサイクルの開始: Service#onCreate など
 - ◎ ライフサイクルの終了: Service#onDestroy
 - ◎ ConfigurationChange: BroadcastReceiver

Sort of ...

```
<manifest package="">
  <application>
    <activity
      android:name=".MyActivity"
      android:configChanges="keyboard|
        keyboardHidden|screenLayout|screenSize|
        orientation|density|fontScale|layoutDirection|
        locale|mcc|mnc|navigation|smallestScreenSize|
        touchScreen|uiMode" />
    </application>
  </manifest>
```

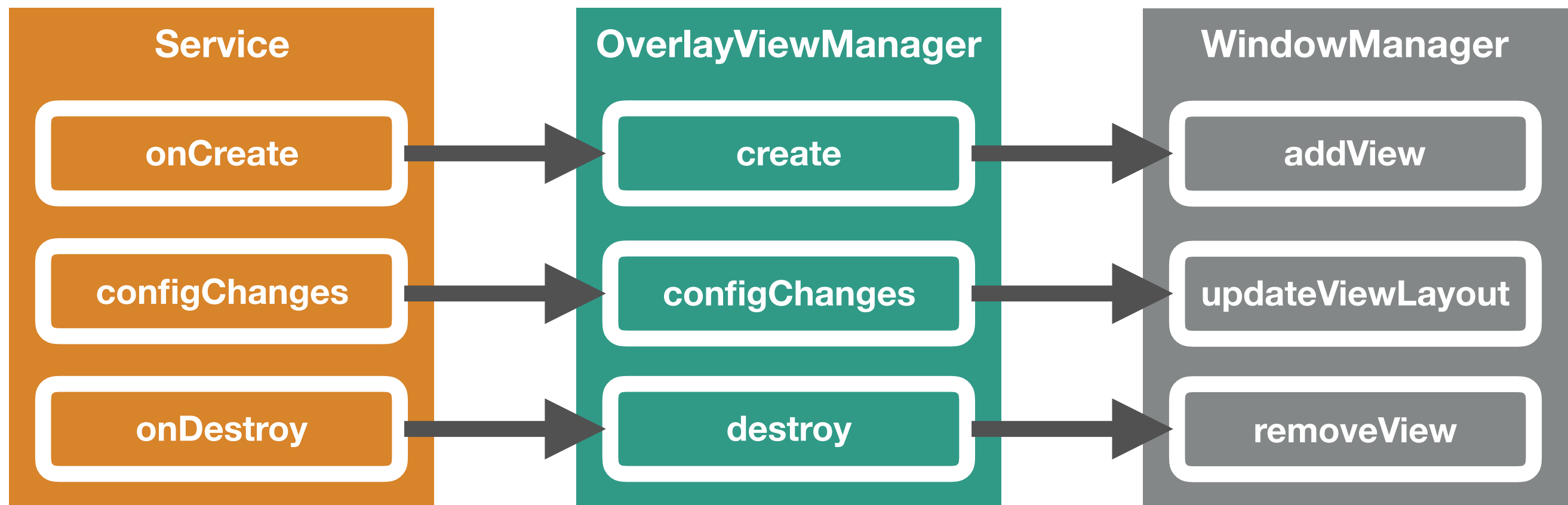
Sort of ...

```
<manifest package="">  
  <application>  
    <activity  
      android:name=".MyActivity"  
      android:configChanges="keyboard|  
keyboardHidden|screenLayout|screenSize|  
orientation|density|fontScale|layoutDirection|  
locale|mcc|mnc|navigation|smallestScreenSize|  
touchScreen|uiMode" />  
    </application>  
  </manifest>
```

Service で View を描画するときの設計

- ▶ DroidKaigi アプリでの実装
 - ▶ [OverlayViewManager](#)
 - ▶ Service 内での View のライフサイクルを握っている
 - ▶ 画面遷移等は考慮していない
- ▶ 画面遷移等を含めた View ベースの画面構築フレームワーク
 - ▶ e.g. [square/flow](#) and [square/mortar](#)

Service で View を描画するときの設計



OverlayViewManger ですること

- ▶ [OverlayViewManger#create](#)
 - ▶ WindowManager#addView を呼ぶ
- ▶ [OverlayViewManger#changeConfiguration](#)
 - ▶ WindowManager#updateViewLayout を呼ぶ
- ▶ [OverlayViewManger#destroy](#)
 - ▶ WindowManager#removeView を呼ぶ

OverlayViewManager の実装

```
public class OverlayViewManager {
    private final Context mContext; // Service
    private final WindowManager mWindowManager;
    private final WindowManager.LayoutParams mParams;
    private View mRootView;

    public void create() {
        mRootView =
            LayoutInflater.from(mContext).inflate(
                R.layout.view_root_overlay, null, false);
        mWindowManager.addView(mRootView, mParams)
    }
}
```

OverlayViewManager の実装

```
public class OverlayViewManager {  
    private final Context mContext; // Service  
    private final WindowManager mWindowManager;  
    private final WindowManager.LayoutParams mParams;  
    private View mRootView;  
  
    public void destroy() {  
        if (mRootView == null)  
            return;  
        mWindowManager.removeView(mRootView)  
    }  
}
```

Service で ConfigurationChange を扱う

- ▶ BroadcastReceiver で ConfigurationChange を検知
 - ▶ BroadcastReceiver#onReceive で View を再描画
 - ▶ ref. WindowManager#updateViewLayout

Service で ConfigurationChange を扱う

```
public class OverlayViewService extends Service {  
    private final BroadcastReceiver mReceiver =  
        new BroadcastReceiver() {  
        @Override  
        public void onReceive(Context c, Intent i) {  
            mOverlayViewManager.changeConfiguration();  
        }  
    }  
}
```

Service で ConfigurationChange を扱う

```
public class OverlayViewService extends Service {  
    private final BroadcastReceiver mReceiver = //...  
  
    @Override  
    public void onCreate() {  
        //.....  
        IntentFilter filter = new IntentFilter(  
            Intent.ACTION_CONFIGURATION_CHANGE);  
        registerReceiver(mReceiver, filter);  
    }  
}
```

Service で ConfigurationChange を扱う

```
public class OverlayViewService extends Service {  
    private final BroadcastReceiver mReceiver = //...  
  
    @Override  
    public void onCreate() {  
        //.....  
        IntentFilter filter = new IntentFilter(  
            Intent.ACTION_CONFIGURATION_CHANGE);  
        registerReceiver(mReceiver, filter);  
    }  
}
```


Service で ConfigurationChange を扱う

```
public class OverlayViewService extends Service {  
    private final BroadcastReceiver mReceiver = //...  
  
    @Override  
    public void onDestroy() {  
        //.....  
        unregisterReceiver(mReceiver) ;  
    }  
}
```

LayoutInflater.from(Context) の引数

- ▶ Context から LayoutInflater を取り出す
 - ▶ LayoutInflater から生成される View は常に同じではない
 - ▶ Context によって振る舞いがちがう
 - ▶ 適切な Context を使わないと見た目が変わってしまう

Activity と Service の違い

- ▶ Activity

- ▶ [ContextThemeWrapper](#) の子クラス
- ▶ テーマで設定した属性値を持っている

- ▶ Service

- ▶ [ContextWrapper](#) の子クラス
- ▶ テーマは存在しない

Context のラップ

- ▶ Service を ContextThemeWrapper でラップしよう
 - ▶ Composite パターン
- ▶ ラップした Context を使えば Service 内でもテーマが適用できる

再び OverlayViewManager の実装

```
public class OverlayViewManager {  
    private final Context mContext; // Service  
    private Context mThemedContext;  
    private View mRootView;  
  
    public void create() {  
        mThemedContext = new  
            OverlayViewContext(mContext);  
        mRootView =  
            LayoutInflater.from(mThemedContext).inflate(  
                R.layout.view_root_overlay, null, false);  
    }  
}
```

再び OverlayViewManager の実装

```
public class OverlayViewManager {  
    private final Context mContext; // Service  
    private Context mThemedContext;  
    private View mRootView;  
  
    public void create() {  
        mThemedContext = new  
            OverlayViewContext(mContext) ;  
        mRootView =  
            LayoutInflater.from(mThemedContext).inflate(  
                R.layout.view_root_overlay, null, false);  
    }  
}
```

再び OverlayViewManager の実装

```
/* package */ class OverlayViewContext  
    extends ContextThemeWrapper {  
  
    public OverlayViewContext(Context base) {  
        super(context, R.style.AppTheme);  
    }  
}
```

再び OverlayViewManager の実装

```
/* package */ class OverlayViewContext  
    extends ContextThemeWrapper {  
  
    public OverlayViewContext(Context base) {  
        super(context, R.style.AppTheme);  
    }  
}
```


再び OverlayViewManager の実装

```
/* package */ class OverlayViewContext
    extends ContextThemeWrapper {
    private LayoutInflater mInflater;

    @Override
    public Object getSystemService(String name) {
        // .....
    }
}
```

再び OverlayViewManager の実装

```
private LayoutInflater mInflater;
```

```
@Override
```

```
public Object getSystemService(String name) {  
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {  
        if (mInflater == null) {  
            mInflater = LayoutInflater.from(  
                getBaseContext()).cloneInContext(this);  
        }  
        return mInflater;  
    }  
    return super.getSystemService(name);  
}
```

再び OverlayViewManager の実装

```
private LayoutInflater mInflater;
```

```
@Override
```

```
public Object getSystemService(String name) {  
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {  
        if (mInflater == null) {  
            mInflater = LayoutInflater.from(  
                getBaseContext()).cloneInContext(this);  
        }  
        return mInflater;  
    }  
    return super.getSystemService(name);  
}
```

再び OverlayViewManager の実装

```
private LayoutInflater mInflater;
```

```
@Override
```

```
public Object getSystemService(String name) {  
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {  
        if (mInflater == null) {  
            mInflater = LayoutInflater.from(  
                getBaseContext()).cloneInContext(this);  
        }  
        return mInflater;  
    }  
    return super.getSystemService(name);  
}
```

再び OverlayViewManager の実装

```
private LayoutInflater mInflater;
```

```
@Override
```

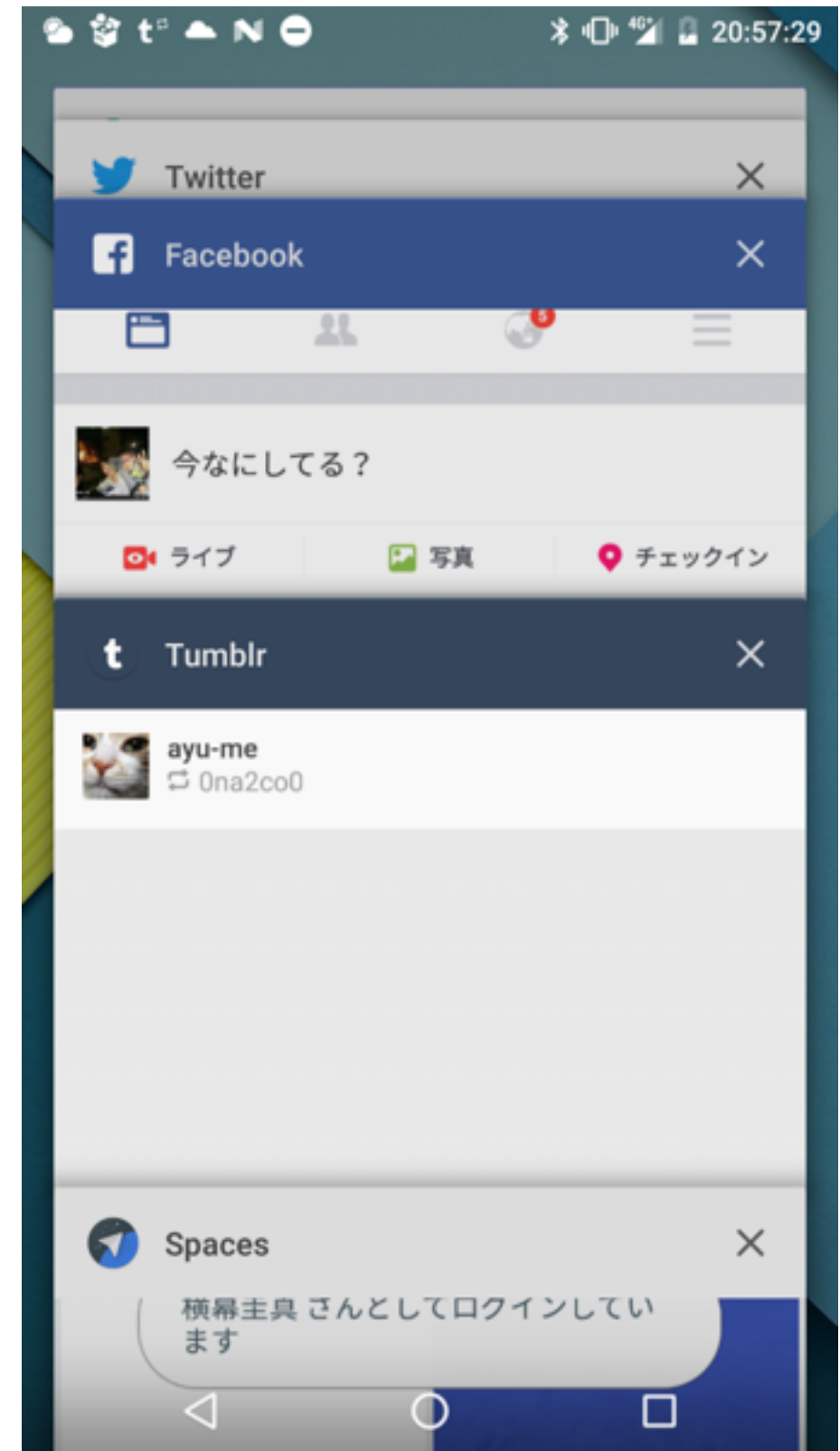
```
public Object getSystemService(String name) {  
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {  
        if (mInflater == null) {  
            mInflater = LayoutInflater.from(  
                getBaseContext()).cloneInContext(this);  
        }  
        return mInflater;  
    }  
    return super.getSystemService(name);  
}
```

Service の終了とプロセスの終了

- ▶ Service が動き続ける限りプロセスは生きている
 - ▶ Service を止める方法
 - ▶ Context#stopService(Intent)
 - ▶ Service#stopSelf()
- ▶ プロセスを止めるためにユーザが取る行動
 - ▶ タスクリストからアプリを消す

Service の終了とプロセスの終了

- ▶ タスクリストからアプリを終了
 - ▶ [Service#onTaskRemoved](#)
- ▶ ここで Service を止めないとプロセスは生きてしまう
- ▶ なんか無限に生きているアプリがいる => ☆1



Service の終了とプロセスの終了

- ▶ システムが強制的にプロセスを終了する場合もある
 - ▶ 表示していた View が急にいなくなる
 - ▶ ユーザのインタラクションがあるときは致命的...
 - ▶ 出来る限りプロセスを長生きさせたい

Service の終了とプロセスの終了

- ▶ システムが強制的にプロセスを終了する場合もある
 - ▶ 表示していた View が急にいなくなる
 - ▶ ユーザのインタラクションがあるときは致命的...
- ▶ 出来る限りプロセスを長生きさせたい

プロセスを
動かかし続ける



Android におけるプロセスの優先度

- ▶ 4 種類の優先度
 - ▶ Foreground Process
 - ▶ Visible Process
 - ▶ Service Process
 - ▶ Cached Process
- ▶ 下のものほど優先度も低いので、kill の対象になる

Android におけるプロセスの優先度

- ▶ Service が関係するプロセスの優先度
 - ▶ Visible Process
 - ▶ Service Process
- ▶ なんとか Visible Process に昇格すれば長生きできる

Android におけるプロセスの優先度

- ▶ Service が関係するプロセスの優先度
 - ▶ Visible Process
 - ▶ **Service Process**
- ▶ なんとか Visible Process に昇格すれば長生きできる

Service Process

- ▶ Service#startService で常駐する Service をもつプロセス
 - ▶ メモリが逼迫してくると kill される
 - ▶ 長生きしているプロセスは kill の対象になる
 - ▶ もしメモリリーク等で Service が正常終了できていない場合にメモリを専有し続けてしまうため

Android におけるプロセスの優先度

- ▶ Service が関係するプロセスの優先度
 - ▶ **Visible Process**
 - ▶ Service Process
- ▶ なんとか Visible Process に昇格すれば長生きできる

Visible Process

- ▶ ユーザにとって重要な情報を表示しているプロセス
 - ▶ e.g. ダイアログの裏にいる Activity を持っている
 - ▶ e.g. Service#startForeground で通知を出している
- ▶ WindowManager に View を出しているても直接は Visible Process にならない

Visible Process

- ▶ ユーザにとって重要な情報を表示しているプロセス
 - ▶ e.g. ダイアログの裏にいる Activity を持っている
 - ▶ e.g. **Service#startForeground** で通知を出している
- ▶ WindowManager に View を出しているも直接は Visible Process にならない

メモリに優しい Service

- ▶ Visible Service でもメモリが厳しいと kill される
 - ▶ メモリに優しい Service を作ろう
 - ▶ メモリリークはご法度
 - ▶ Service にもライフサイクルがあることに注意する
 - ▶ 画面遷移を作る場合、都度必要ない View を remove する

重ねた View に情報を表示する

- ▶ 構築した View にデータを渡したい
 - ▶ デバッグ用の情報を出す
 - ▶ アプリのいろいろな部分からデータを渡せる設計が必要



重ねた View に情報を表示する

- ▶ 構築した View にデータを渡したい
- ▶ デバッグ用の情報を出す
- ▶ アプリのいろいろな部分からデータを渡せる設計が必要



Activity など
とつなぎこむ



Service につなぎこむ 3 つの方法

- ▶ Service にデータをわたす 3 つの方法
 - ▶ startService の Intent にデータをつめる
 - ▶ EventBus を活用してデータをわたす
 - ▶ Dagger 等の DI でスコープを切る & Rx を使う

startService の Intent にデータをつめる

- ▶ Intent#putExtra() を活用
 - ▶ Service#onStartCommand() で受け取る
- ▶ pros & cons
 - ▶ Android フレームワークの仕組みで完結する
 - ▶ データ型に Parcelable の実装が必要
 - ▶ View を管理する抽象レイヤに具体的な処理が入り込む

startService の Intent にデータをつめる

```
public class DebugOverlayService extends Service {
    public static final String EXTRA_DATA = // .....
    private OverlayViewManager mManager;

    @Override
    public int onStartCommand(Intent i, int flags,
        int startId) {
        // .....
        Data data = i.getParcelableExtra(EXTRA_DATA);
        mManager.onReceiveData(data);
    }
}
```


startService の Intent にデータをつめる

```
public class DebugOverlayService extends Service {  
    public static final String EXTRA_DATA = // .....  
    private OverlayViewManager mManager;  
  
    @Override  
    public int onStartCommand(Intent i, int flags,  
        int startId) {  
        // .....  
        Data data = i.getParcelableExtra(EXTRA_DATA);  
        mManager.onReceiveData(data);  
    }  
}
```

startService の Intent にデータをつめる

```
public class DebugOverlayService extends Service {  
    public static final String EXTRA_DATA = // .....  
    private OverlayViewManager mManager;  
  
    @Override  
    public int onStartCommand(Intent i, int flags,  
        int startId) {  
        // .....  
        Data data = i.getParcelableExtra(EXTRA_DATA);  
        mManager.onReceiveData(data);  
    }  
}
```

EventBus を活用してデータをわたす

- ▶ [greenrobot/EventBus](#), [square/otto](#), [reactivex/RxJava](#) など
 - ▶ シンプルな publisher/subscriber パターン
- ▶ pros & cons
 - ▶ 使い方を覚えれば簡単に適用できる
 - ▶ Parcelable の実装がいらない
 - ▶ ライブラリによっては Deprecated になっている

EventBus を活用してデータをわたす

```
public class DebugOverlayView
    extends RelativeLayout {
    private EventBus;

    @Subscribe
    public void onDataReceived(Data data) {
        // draw something from data
    }
}
```

EventBus を活用してデータをわたす

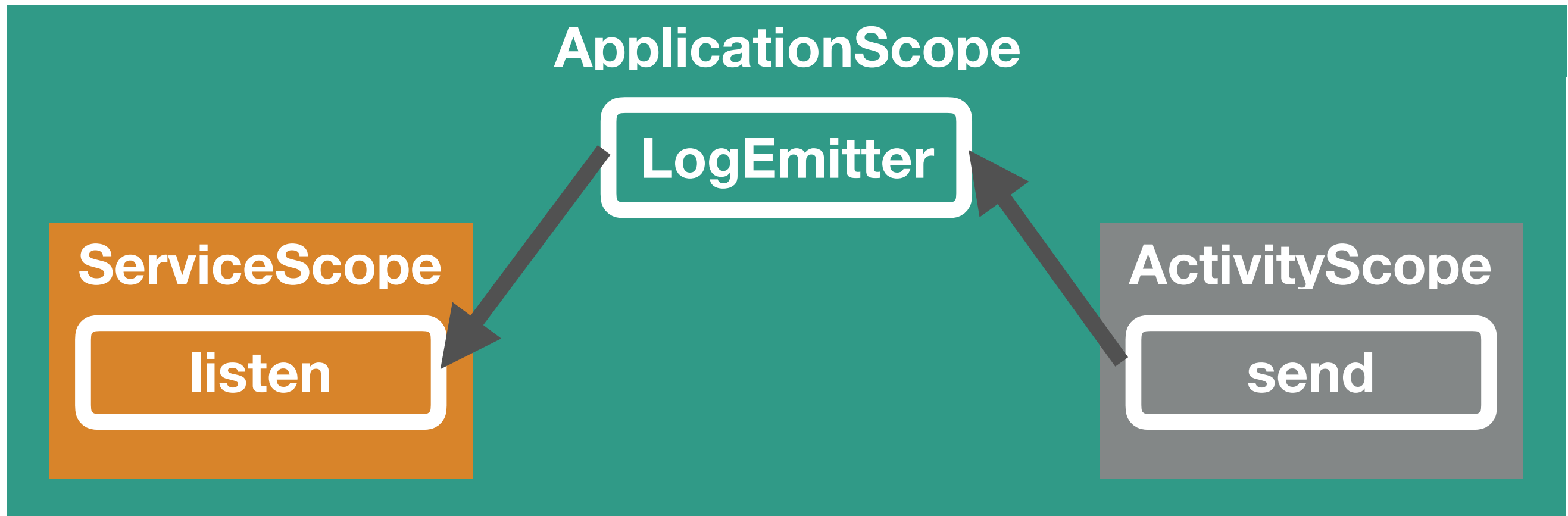
```
public class DebugOverlayView
    extends RelativeLayout {
    private EventBus;

    @Subscribe
    public void onDataReceived(Data data) {
        // draw something from data
    }
}
```

Dagger 等の DI でスコープを切る & Rx を使う

- ▶ スコープの切り方
 - ▶ Application のライフサイクルに合わせるスコープ
 - ▶ Activity のライフサイクルに合わせるスコープ
 - ▶ Service のライフサイクルに合わせるスコープ
- ▶ DroidKaigi アプリ
 - ▶ Application のスコープにいるものがルーティングを担当

Dagger 等の DI でスコープを切る & Rx を使う



Dagger 等の DI でスコープを切る & Rx を使う

- ▶ pros & cons

- ▶ オブジェクトのライフサイクルが明確になる
- ▶ EventBus に類似のパターンが Rx で完結する
- ▶ 自分で EventBus の仕組みを作る必要がある

まとめ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ Activity などとつなぎこむ



どう実現するのか

- ▶ やること
 - ▶ 特別なレイヤに View を表示する
 - ▶ 裏で View を更新し続ける
- ▶ 考慮すべきこと
 - ▶ プロセスを動かし続ける
 - ▶ **Activity などとつなぎこむ**



さいごに

- ▶ このトークには若干の黒魔術的内容が含まれています
- ▶ 応用する場合、用法用量を守って楽しくお使いください

BUILDING MY OWN DEBUGGING TOOL ON OVERLAY

Appendix

▶ 參考資料

- ▶ [WindowManager - Android Developers](#)
- ▶ [WindowManager.LayoutParams - Android Developers](#)
- ▶ [Whoa, Views can do that?](#)
[WindowManager ideas and tricks! - @eric cochran](#)
- ▶ [DroidKaigi/conference-app-2017 - GitHub](#)

We are hiring!



What we do

