java层实现动态切换系统字体

笔记本:	里克分享		
创建时间:	2017/2/28 8:46	**更新时间:**	2017/2/28 8:47
作者:	里克Rick_tan		
URL:	http://km.oa.com/group/22595/articles/show/229242?kmref=search&from_page=1&no=1		

java层实现动态切换系统字体

在前面的《Android下Color Emoji 的实现》有说：android字体由android2D图形引擎skia实现，并在Zygote的Preloading classes中对系统字体进行load...，实现类：frameworks/base/graphics/java/android/graphics/Typeface.java

在Typeface里面默认随系统启动就加载好了系统字体：

```
 public static final Typeface DEFAULT;      //默认字体
public static final Typeface DEFAULT_BOLD;  //默认的粗体字体，不过这个不一定是粗体，和就用的字体有关，如果你用的字体不是粗体，这个就不是粗体。
public static final Typeface SANS_SERIF;    //正常样式的无衬线字体
public static final Typeface SERIF;         //正常样式的衬线字体
public static final Typeface MONOSPACE;     //正常样式的等宽字体。
//4种样式
public static final int NORMAL = 0;
public static final int BOLD = 1;
public static final int ITALIC = 2;
public static final int BOLD_ITALIC = 3;
```

分别是正常、粗体、斜体、粗斜体。默认系统里面是有7个字体文件

这些都是在static初始化的

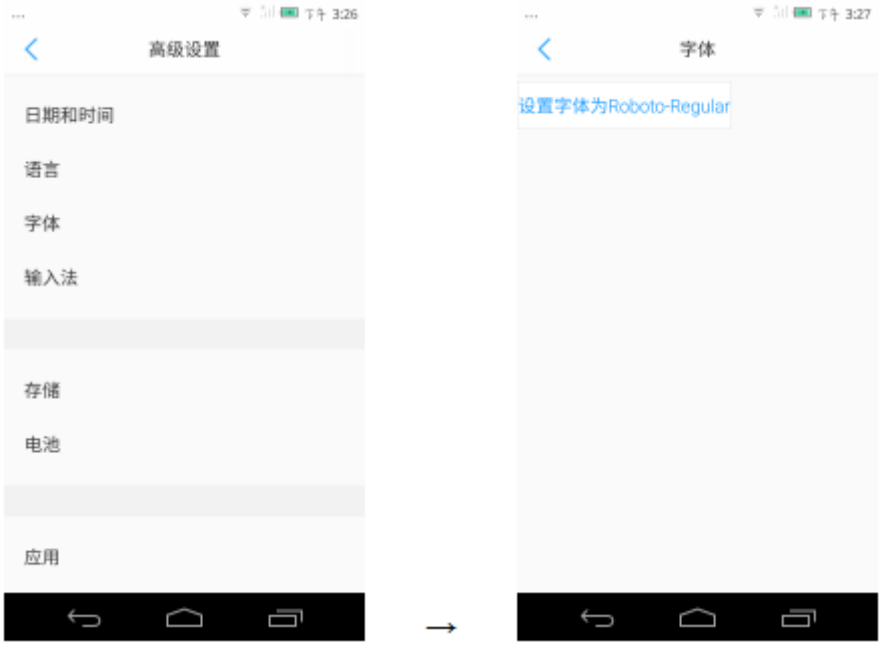
```
static {
    DEFAULT        = create ((String) null, 0);
    DEFAULT_BOLD    = create((String) null, Typeface.BOLD);
    SANS_SERIF      = create( "sans-serif", 0);
    SERIF          = create ("serif" , 0);
    MONOSPACE       = create( "monospace", 0);

    sDefaults = new Typeface[] {
        DEFAULT,
        DEFAULT_BOLD,
```

```
        create((String) null, Typeface.ITALIC),
        create((String) null, Typeface.BOLD_ITALIC),
    };
}
```

要动态切换字体一般至少要覆盖sans_serif 的 Regular, Bold, Italic, BoldItalic 。我暂只从字体管家里面撸了一个中文字体做预研。
实现的基本思路：

在系统里面弄一个界面(Settings)，读取指定路径下的字体文件。然后替换掉系统的默认字体，最后让应用更新一下。
一、界面

截图：



这里的实现很简单，就是点击界面按钮就通过SystemProperties来增加一条记录字体路径，然后通知Typeface调用setCustomFont进行字体替换。

二、Typeface改造

把final属性去掉，增加更改字体处理逻辑：

```
private static final String TAG_R = "rick_Print:";
 static final Typeface DEFAULT_INTERNAL;
 static final Typeface DEFAULT_BOLD_INTERNAL;
 static final Typeface SANS_SERIF_INTERNAL;
 static final Typeface SERIF_INTERNAL;
 static final Typeface MONOSPACE_INTERNAL;
 static final String QROM_THEME_FONTS_FILE_SETTINS_NAME = "persist.sys.fonts" ;
 static final String QROM_THEME_FONTS_DEFAULT_NAME = "default" ;
 static public void setCustomFont() {
    Log.d( TAG_R, "setCustomFont" );
```

```
        boolean useDefault = false;
        Typeface font = null ;
        String fontPath =
SystemProperties.get(QROM_THEME_FONTS_FILE_SETTINS_NAME , QROM_THEME_FONTS_DEF
AULT_NAME );
        Log. d( TAG_R, "Typeface fontPath=" +fontPath);
        if (QROM_THEME_FONTS_DEFAULT_NAME .equals(fontPath)) {
            Log. d( TAG_R, "use the default font !!" );
            font = DEFAULT_INTERNAL ;
            useDefault = true ;
        } else {
            try {
                font = createFromFile(fontPath);
            } catch (Exception e) {
                e.printStackTrace();
                Log. e( TAG_R, "load custom font: " + fontPath + " failed !!" );
                font = null ;
            }
        }


        if (!useDefault && null != font) {
            Log. i( TAG_R, "system: use the custom font: " + fontPath + " as system font. old-
default=" + DEFAULT + ", new=" + font);
            // 预研吗，所以处理得很简单
            DEFAULT = font;
            DEFAULT_BOLD = font;
            SANS_SERIF = font;
            SERIF = font;
            MONOSPACE = font;
            sDefaults [0] = DEFAULT ;
            sDefaults [1] = DEFAULT_BOLD ;
            sDefaults [2] = DEFAULT ;
            sDefaults [3] = DEFAULT ;
        } else {
            Log. d( TAG_R, "setCustomFont useDefault" );
            DEFAULT = DEFAULT_INTERNAL ;
```

```
            DEFAULT_BOLD = DEFAULT_BOLD_INTERNAL;
            SANS_SERIF = SANS_SERIF_INTERNAL;
            SERIF = SERIF_INTERNAL;
            MONOSPACE = MONOSPACE_INTERNAL;
            sDefaults[0] = DEFAULT;
            sDefaults[1] = DEFAULT_BOLD;
            sDefaults[2] = T_ITALIC;
            sDefaults[3] = T_BOLD_ITALIC;
        }
    }
```

调整静态代码：

```
    static {
        // DEFAULT = create((String) null, 0);
        // DEFAULT_BOLD = create((String) null, Typeface.BOLD);
        // SANS_SERIF = create("sans-serif", 0);
        // SERIF = create("serif", 0);
        // MONOSPACE = create("monospace", 0);
        DEFAULT = DEFAULT_INTERNAL = create((String) null , 0);
        DEFAULT_BOLD = DEFAULT_BOLD_INTERNAL = create((String) null, Typeface. BOLD );
        SANS_SERIF = SANS_SERIF_INTERNAL = create( "sans-serif", 0);
        SERIF = SERIF_INTERNAL = create( "serif" , 0);
        MONOSPACE = MONOSPACE_INTERNAL = create( "monospace", 0);

        T_ITALIC = create((String) null , Typeface.ITALIC );
        T_BOLD_ITALIC = create((String) null , Typeface.BOLD_ITALIC );
        sDefaults = new Typeface[] { DEFAULT, DEFAULT_BOLD , T_ITALIC , T_BOLD_ITALIC, };

        setCustomFont();
    }
```

编译push进行测试，发现不管是已运行的应用还是新打开的应用都没改变。。。o(ˊ□ˋ)o按当初的设想新开的应用应该生效的，事实上没有，重启手机后发现字体被切换过来了，这个坑.....就是当初android优化造成的。。。继续往下！！！

Typeface 这几个 static 变量是在 static 语句的初始化的，就是每个进程这个 Typeface 类第一次实例化的时候就触发。

android 用的是 dalvik（从 4.4 开始加入 art，L 开始 art 变成默认的虚拟机了）虚拟机。启动一个 java 程序是很慢的，没感觉的去 PC 启动个 java 程序看看就知道了。不光要启动进程，还要启动 dalvik 虚拟机，而且 android 上层 java 接口还有 N 个 class，系统还有一堆系统资源。如果重头开始加载的话，冷

启动一个 android 程序是非常慢的（这里为什么说冷启动呢，因为 android 一旦启动了一个应用，为其开启了一个进程，退出应用后，这个进程不会马上销毁的，这样进程加载了的运行环境都还在，下次启动直接从应用的业务逻辑开始加载就新了，这个是不是有点像电脑的快速休眠、唤醒，这种我觉得可以叫热启动。那么相应的如果事先不存在这个应用的运行环境，重头开始启动就叫冷启动，相当于电脑开机）。
于是 android首先弄了一个叫 Zygote 的东西，这个是个 natvie 程序，开机由 init.rc 启动。然后这个会调用 AppRuntime 的 start 方法，去启动 dalvik 虚拟机，然后去加载 java 中 zygote 相关的代码，然后这个 start 就会调用 startVM 去启动 dalvik 虚拟机加载 java 运行环境，然后调用 com.android.internal.os.ZygoteInit。在ZygoteInit的main就进行了一系列的initialization，包括：

```
static void preload() {
    preloadClasses();
    preloadResources();
    preloadOpenGL();
}
```

由于启动虚拟机很慢，而且还要 android 应用经常要使用系统提供的那一堆 java class 的接口和资源，这些东西第一加载也慢，第二其实每个应用这些东西都是一样的，如果每个应用一份会浪费内存。

于是 android引入了一个源自 linux（unix） 的 fork 机制。fork 是一个 linux 系统调用，能够在当前进程创建出一个子进程，子进程完全共享父进程的共有环境变量。好，有了这个基础就可以做一些优化了：

1. 弄一个所有应用的共有父进程，专门用来 fork 子进程的，这个就是 zygote（孵化器这个名字取得真形象）。
2. 由于子进程可以共享（继承）父进程的运行环境，所以可以把一些共用的东西在 zygote 中加载好，这样子进程一 fork 出来就有完整的运行环境，不需要重新加载。
3. 由于子进程可以共享（继承）父进程的运行环境，基于 fork 的 `copy-on-write` 原则，只要这些变量不改变，那么子进程都不需要复制 zygote 进程的环境变量，共同一份，内存开销大大降低。

所以这个 zygote 在初始化的时候会有一个 preload 的预加载处理，其实这些资源，zygote 根本不用，是给它的子进程用的，这些子进程就是 android 的那些上层应用。这里我看看 preloadClasses 就是预先加载 sdk 中的那些 class。

上面的内容就解释了为什么在Settings里面进行设置字体，改变了Typeface里面的静态变量会没起作用。
因为通过 Settings 里面调用设置了 SystemProperties，Typeface拿到这个路径加载字体将静态变量替换掉，就算是另外一个应用重新启动，但是由于这些应用都是通过 zygote fork 出来的，zygote 里面预先加载了 Typeface，跑了它的 static 代码块，它的那些 static 变量都已经设置好了。所以它 fork 出来的子进程，就算是新 fork 出来的，也不会再跑 Typeface 的那些 static 语句块，因为它继承了 zygote 预先加载好的 Typeface 的这些变量。这应该也是为什么Typeface里面的那几个静态变量是final的原因...

三、zygote 改造

代码在手上，办法肯定是有的:
fork 的子进程会继承 zygote 的环境变量，那么只要把 zygote 里面的预先加载好的 Typeface 的环境变量更新一下就可以了。简单点就是在 zygote 的进程里面调用下在 Typeface 中新增的那个

setCustomFont()就可以了。
android 中 IPC 使用的 binder，但zygote 用的是 socket （还有 vold）,看一下协议：当有一个连接请求过来的话，这个处理被封装成 ZygoteConnection 了，然后进入里面的 runOnce 函数。分析下Zygote的
parseArgs(frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java)确定发送命令：

```
    public void sendArgsToZygoteForChangeSysFont() {
        synchronized (this ) {
          // update Zygote preload class cache.
          Log. d( TAG_R, "sendArgsToZygoteForChangeSysFont" );
           try {
              ArrayList<String> argsForZygote = new ArrayList<String>();
              argsForZygote.add( "-classpath" );
              argsForZygote.add( ZYGOTE_CHANGE_FONT_COMMAND );
              argsForZygote.add( "--nice-name=" + ZYGOTE_COMMAND_UPDATE_FONT );
              Process.zygoteSendArgsAndGetResult_tos(argsForZygote);
          } catch (Exception e) {
              e.printStackTrace();
          }
       }
    }
```

Process.java:

```
    public static void zygoteSendArgsAndGetResult_tos(ArrayList<String> args){
      try {
          zygoteSendArgsAndGetResult(args);
      } catch (ZygoteStartFailedEx ex) {
          Log. e( "rick_Print:", "Starting VM process through Zygote failed");
          throw new RuntimeException( "Starting VM process through Zygote failed", ex);
      }
   }
```

这里的发送命令是放在AMS::updateConfiguration里面的，因为后面要处理已有的进程更新，会在UI层调用updateConfiguration。
然后就是在ZygoteConnection 处理发送过来的命令：

```
    boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
        ...
      //add by yongchen
      boolean isChangeFontCommand = false;
```

```
        try {
            parsedArgs = new Arguments(args);
              //add by yongchen
              if (ZYGOTE_CHANGE_FONT_COMMAND.equals(parsedArgs. classpath)) {
                  Log. d( TAG_R, "ZYGOTE_CHANGE_FONT_COMMAND.equals(parsedArgs.classpath)
is true");
                  isChangeFontCommand = true ;
              }
             ...
              //add by yongchen
              if (isChangeFontCommand) {
                  pid = 9999;
                   execChangeFontCommand(parsedArgs);
              }
               else {
                pid = Zygote.forkAndSpecialize(parsedArgs.uid , parsedArgs. gid, parsedArgs.gids ,
                        parsedArgs. debugFlags , rlimits, parsedArgs.mountExternal ,
parsedArgs.seInfo ,
                        parsedArgs. niceName);
              }
        } catch (IOException ex) {
            logAndPrintError(newStderr, "Exception creating pipe", ex);
        } catch (ErrnoException ex) {
            logAndPrintError(newStderr, "Exception creating pipe", ex);
        } catch (IllegalArgumentException ex) {
            logAndPrintError(newStderr, "Invalid zygote arguments", ex);
        } catch (ZygoteSecurityException ex) {
            logAndPrintError(newStderr,
                    "Zygote security policy prevents request: ", ex);
        }

        try {
            if (pid == 0) {
                // in child
                IoUtils.closeQuietly(serverPipeFd);
```

```
            serverPipeFd = null ;
            handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);

            // should never get here, the child is expected to either
            // throw ZygoteInit.MethodAndArgsCaller or exec().
            return true ;
        } else {
            // in parent...pid of < 0 means failure
            IoUtils.closeQuietly(childPipeFd);
            childPipeFd = null ;
               //add by yongchen
               if (isChangeFontCommand) {
                   return handleParentProcInChangeFontCommand(pid, descriptors, parsedArgs);
               } else {
                   return handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);
               }
        }
    } finally {
        IoUtils.closeQuietly(childPipeFd);
        IoUtils.closeQuietly(serverPipeFd);
    }
}
  //add by yongchen
  private void execChangeFontCommand(Arguments parsedArgs) {
      Log. d( TAG_R, "execMagicCommand" );
      String cmds = parsedArgs. niceName;
       if (null == cmds) {
          Log. i( TAG_R, "the changefont command is null, ignore it !!");
           return ;
      }
       if (ZYGOTE_COMMAND_UPDATE_FONT .equals(cmds)) {
          Log. i( TAG_R, "exec zygote changefont command: " + cmds);
          Typeface.updateDefaultFont();
      } else {
          Log. i( TAG_R, "the changefont command: " + cmds + " is unknow, ignore it !!" );
```

```
        }
    }
     //add by yongchen
     private boolean handleParentProcInChangeFontCommand(int pid,
            FileDescriptor[] descriptors, Arguments parsedArgs) {
        Log. d( TAG_R, "handleParentProcInChangeFontCommand" );
         if (descriptors != null) {
             for (FileDescriptor fd : descriptors) {
                 IoUtils.closeQuietly(fd);
             }
        }
         boolean usingWrapper = false;
         try {
             mSocketOutStream .writeInt(pid);
              mSocketOutStream .writeBoolean(usingWrapper);
        } catch (IOException ex) {
            Log. e( TAG_R, "Error reading from command socket", ex);
             return true ;
        }

         return false ;
    }
```

zygote 就改造就完成了，编译push测试一下果然有效果。

但是现在只是新 fork 的子进程有效，原来 fork 出来的还是没改变。

四、更新已启动的应用

在Configuration里面添加一个用于字体切换的字段mFlipFont,在UI层调用updateConfiguration

```
     private void updateFonts() {
         Log. d( "rick_Print:", "Settings to call updateFonts");
          try {
             IActivityManager am = ActivityManagerNative.getDefault();
             Configuration config = am.getConfiguration();
             config. mQromConfiguration .mFlipFont += 1;
             am.updateConfiguration(config);
         } catch (RemoteException e) {
```

```
            e.printStackTrace();
        }
    }
```

字段的处理参考Android语言切换逻辑[《android源码分析(一) - 语言切换机制》]

到这里编译push测试一下发现原来fork 出来的还是没改变，还是前面的那个坑，这样只需要在每一个应用的运行环境里调用一下在 Typeface 中新增的那个setCustomFont()就可以。

```
if (isFontChanged(config)) {
    Typeface.setCustomFont();
}
```

附录：

老罗的[《 Android系统进程Zygote启动过程的源代码分析》]

[《android源码分析(一) - 语言切换机制》]

备注：

切换字体存在比较多的小问题，比如：Android4.4在textview里面设置字体（`setTypefaceFromAttrs)` 的时候如果是传入了`familyName`字体切换后效果就有问题等