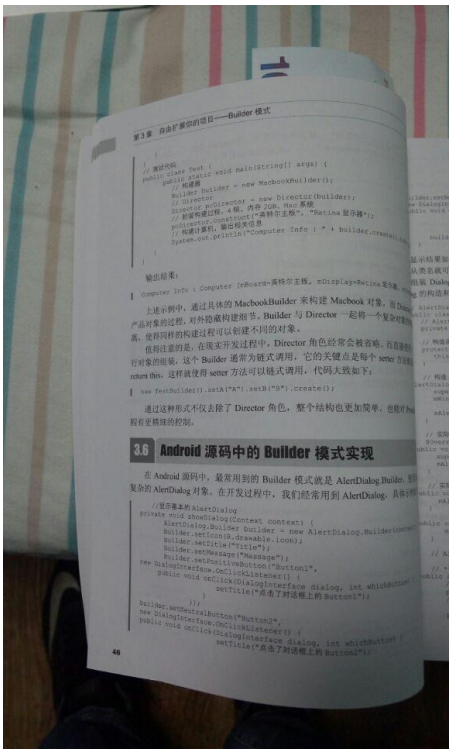
Android 源码设计模式解析与实战 排版勘误

目录

44 页下到 45 页



电脑的组装过程较为复杂，且组装顺序却是不固定不变的。为了示例易于理解，我们把 PC 组装的过程简化为构建主机、设置操作系统、设置显示器三个部分，然后通过 Director 和具体的 Builder 来构建 PC 对象。请看下面的示例。

```
package com.dp.example.builder;

// 电脑抽象类，即 Product 角色
```

```
public abstract class Computer {
	protected String mBoard;
	protected String mDisplay;
	protected String mOS;

	protected Computer() {
	}

	// 设置 CPU 核心数
	public void setBoard(String board) {
		mBoard = core;
	}

	// 设置内存
	public void setDisplay(String display) {
		mDisplay = gb;
	}

	// 设置操作系统
	public abstract void setOS();

	@Override
	public String toString() {
		return "Computer [mBoard=" + mBoard + ", mDisplay=" + mDisplay
				+ ", mOS=" + mOS + "]";
	}

}

// 具体的 Computer 类，Macbook
public class Macbook extends Computer {

	protected Macbook() {
	}

	@Override
	public void setOS() {
		mOS = "Mac OS X 10.10";
	}

}


// 抽象 Builder 类
```

```
public abstract class Builder {
    // 设置主机
    public abstract void buildBoard(String board);

    // 设置显示器
    public abstract void buildDisplay(String    display);

    // 设置操作系统
    public abstract void buildOS();

    // 创建 Computer
    public abstract Computer create();

}

// 具体的 Builder 类，MacbookBuilder
public class MacbookBuilder extends Builder {
    private Computer mComputer = new Macbook();

    @Override
    public void buildBoard(String board) {
        mComputer.setBoard(core);
    }

    @Override
    public void buildDisplay(String display) {
        mComputer.setDisplay(display);
    }

    @Override
    public void buildOS() {
        mComputer.setOs();
    }

    @Override
    public Computer create() {
        return mComputer;
    }

}

// Director 类，负责构造 Computer
public class Director {
    Builder mBuilder = null;
```

```
    /**
     * @param builder
     */
    public Director(Builder builder) {
        mBuilder = builder;
    }

    /**
     * 构建对象
     */
    public void construct(String board, String display) {
        mBuilder.buildBoard(board);
        mBuilder.buildDisplay(display);
        mBuilder.buildOS();
    }
}
```

46 页 下到 47 页上

```
// 显示基本的 AlertDialog
private void showDialog(Context context) {
        AlertDialog.Builder builder = new AlertDialog.Builder(context);
        builder.setIcon(R.drawable.icon);
        builder.setTitle("Title");
        builder.setMessage("Message");
        builder.setPositiveButton("Button1",
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
                                setTitle("点击了对话框上的 Button1");
                            }
                        });
        builder.setNeutralButton("Button2",
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
                                setTitle("点击了对话框上的 Button2");
                            }
                        });
        builder.setNegativeButton("Button3",
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
```

```
                                setTitle("点击了对话框上的 Button3");
                            }
                        });
            builder.create().show();    // 构建 AlertDialog，并且显示
}
```

47 页下到 48 页上

```
// AlertDialog
public class AlertDialog extends Dialog implements DialogInterface {
    // AlertController, 接受 Builder 成员变量 P 中的各个参数
    private AlertController mAlert;

    // 构造函数
    protected AlertDialog(Context context, int theme) {
        this(context, theme, true);
    }

    // 构造 AlertDialog
    AlertDialog(Context context, int theme, boolean createContextWrapper) {
        super(context, resolveDialogTheme(context, theme), createContextWrapper);
        mWindow.alwaysReadCloseOnTouchAttr();
        // 构造 AlertController
        mAlert = new AlertController(getContext(), this, getWindow());
    }

    // 实际上调用的是 mAlert 的 setTitle 方法
    @Override
    public void setTitle(CharSequence title) {
        super.setTitle(title);
        mAlert.setTitle(title);
    }

    // 实际上调用的是 mAlert 的 setCustomTitle 方法
    public void setCustomTitle(View customTitleView) {
        mAlert.setCustomTitle(customTitleView);
    }

    public void setMessage(CharSequence message) {
        mAlert.setMessage(message);
    }
```

```
// AlertDialog 其他的代码省略

// ************    Builder 为 AlertDialog 的内部类     ******************
public static class Builder {
    // 1： 存储 AlertDialog 的各个参数，例如 title, message, icon 等.
     private final AlertController.AlertParams P;
    //  属性省略

     public Builder(Context context) {
          this(context, resolveDialogTheme(context, 0));
    }

     public Builder(Context context, int theme) {
         P = new AlertController.AlertParams(new ContextThemeWrapper(
               context, resolveDialogTheme(context, theme)));
          mTheme = theme;
    }

    // Builder 的其他代码省略 ......

    // 2： 设置各种参数
     public Builder setTitle(CharSequence title) {
           P.mTitle = title;
          return this;
    }
     public Builder setMessage(CharSequence message) {
         P.mMessage = message;
          return this;
    }


     public Builder setView(View view) {
         P.mView = view;
         P.mViewSpacingSpecified = false;
          return this;
    }

      // 2 、构建 AlertDialog，传递参数
     public AlertDialog create() {
         // 3、调用 new AlertDialog 构造对象，并且将参数传递个体 AlertDialog
          final AlertDialog dialog = new AlertDialog(P.mContext, mTheme, false);
         // 4、将 P 中的参数应用的 dialog 中的 mAlert 对象中
          P.apply(dialog.mAlert);
          dialog.setCancelable(P.mCancelable);
```

```
                if (P.mCancelable) {
                    dialog.setCanceledOnTouchOutside(true);
                }
                dialog.setOnCancelListener(P.mOnCancelListener);
                if (P.mOnKeyListener != null) {
                    dialog.setOnKeyListener(P.mOnKeyListener);
                }
                return dialog;
            }
        }
}
```

48 页下到 49 页上

```
public void apply(AlertController dialog) {
        if (mCustomTitleView != null) {
            dialog.setCustomTitle(mCustomTitleView);
         } else {
            if (mTitle != null) {
                dialog.setTitle(mTitle);
             }
            if (mIcon != null) {
                dialog.setIcon(mIcon);
             }
            if (mIconId >= 0) {
                dialog.setIcon(mIconId);
             }
            if (mIconAttrId > 0) {
                dialog.setIcon(dialog.getIconAttributeResId(mIconAttrId));
             }
        }

        if (mMessage != null) {
            dialog.setMessage(mMessage);
        }
        if (mPositiveButtonText != null) {
            dialog.setButton(DialogInterface.BUTTON_POSITIVE, mPositiveButtonText,
            mPositiveButtonListener, null);
        }
        if (mNegativeButtonText != null) {
            dialog.setButton(DialogInterface.BUTTON_NEGATIVE, mNegativeButtonText,
```

```
            mNegativeButtonListener, null);
        }
        if (mNeutralButtonText != null) {
            dialog.setButton(DialogInterface.BUTTON_NEUTRAL, mNeutralButtonText,
            mNeutralButtonListener, null);
         }
        if (mForceInverseBackground) {
            dialog.setInverseBackgroundForced(true);
         }
         // 如果设置了 mItems，则表示是单选或者多选列表，此时创建一个 ListView
        if ((mItems != null) || (mCursor != null) || (mAdapter != null)) {
            createListView(dialog);
        }
        // 将 mView 设置给 Dialog
        if (mView != null) {
            if (mViewSpacingSpecified) {
                dialog.setView(mView, mViewSpacingLeft, mViewSpacingTop, mViewSpacingRight,
                    mViewSpacingBottom);
             } else {
                dialog.setView(mView);
             }
        }
}
```

49 页下

```
        // 显示 Dialog
public void show() {
            // 已经是显示状态，则 return
        if (mShowing) {
            if (mDecor != null) {
                if (mWindow.hasFeature(Window.FEATURE_ACTION_BAR)) {
                    mWindow.invalidatePanelMenu(Window.FEATURE_ACTION_BAR);
                 }
                mDecor.setVisibility(View.VISIBLE);
             }
            return;
         }

        mCanceled = false;
```

```
        // 1、onCreate 调用
        if (!mCreated) {
            dispatchOnCreate(null);
        }

         // 2、onStart
        onStart();
         // 3、获取 DecorView
        mDecor = mWindow.getDecorView();
            // 代码省略
         // 4、获取布局参数
         WindowManager.LayoutParams l = mWindow.getAttributes();
        if ((l.softInputMode
                & WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARD_NAVIGATION) == 0) {
                    WindowManager.LayoutParams nl = new WindowManager.LayoutParams();
                nl.copyFrom(l);
                nl.softInputMode |=
                                WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARD_NAVIGATION;
                l = nl;
        }

        try {
            // 5、将 mDecor 添加到 WindowManager 中
              mWindowManager.addView(mDecor, l);
              mShowing = true;
               // 发送一个显示 Dialog 的消息
              sendShowMessage();
           } finally {
              }
}
```

51 页下到 52 页上

```
private void setupView() {
        // 1、获取并初始化内容区域
        LinearLayout contentPanel = (LinearLayout) mWindow.findViewById(R.id.contentPanel);
        setupContent(contentPanel);

        // 2、初始化按钮
        boolean hasButtons = setupButtons();
```

```
        // 3、获取并初始化 title 区域
        LinearLayout topPanel = (LinearLayout) mWindow.findViewById(R.id.topPanel);
        TypedArray a = mContext.obtainStyledAttributes(
null, com.android.internal.R.styleable.AlertDialog, com.android.internal.R.attr.alertDialogStyle, 0);
        boolean hasTitle = setupTitle(topPanel);

        // 按钮区域的可见性
        View buttonPanel = mWindow.findViewById(R.id.buttonPanel);
        if (!hasButtons) {
            buttonPanel.setVisibility(View.GONE);
            mWindow.setCloseOnTouchOutsideIfNotSet(true);
         }

         // 4、自定义内容视图区域
         FrameLayout customPanel = null;
        // 如果用户设置了内容视图，那么将它显示在 customPanel 的 custom 布局里面
        if (mView != null) {
            customPanel = (FrameLayout) mWindow.findViewById(R.id.customPanel);
            FrameLayout custom = (FrameLayout) mWindow.findViewById(R.id.custom);
            // 显示用户设置的视图
            custom.addView(mView, new LayoutParams(MATCH_PARENT, MATCH_PARENT));
            if (mViewSpacingSpecified) {
                custom.setPadding(mViewSpacingLeft, mViewSpacingTop, mViewSpacingRight,
                        mViewSpacingBottom);
               }
            if (mListView != null) {
                ((LinearLayout.LayoutParams) customPanel.getLayoutParams()).weight = 0;
               }
        } else {
            mWindow.findViewById(R.id.customPanel).setVisibility(View.GONE);
        }

         // 代码省略

         // 设置背景
        setBackground(topPanel, contentPanel, customPanel, hasButtons, a, hasTitle, buttonPanel);
        a.recycle();
}
```

```
registerService(WINDOW_SERVICE, new ServiceFetcher() {
        Display mDefaultDisplay;
        public Object getService(ContextImpl ctx) {
                Display display = ctx.mDisplay;
                if (display == null) {
                        if (mDefaultDisplay == null) {
                                DisplayManager dm = (DisplayManager)ctx.getOuterContext().
                                        getSystemService(Context.DISPLAY_SERVICE);
                                mDefaultDisplay = dm.getDisplay(Display.DEFAULT_DISPLAY);
                          }
                        display = mDefaultDisplay;
                }
                // 构造 WindowManagerImpl 对象
                return new WindowManagerImpl(display);
         }
});
```

53 页下到 54 页上

```
Dialog(Context context, int theme, boolean createContextThemeWrapper) {
        // 设置、包装 Context 相关的代码
        // 1、获取 WindowManager
        mWindowManager = (WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
        Window w = PolicyManager.makeNewWindow(mContext);
        mWindow = w;
        // 设置 Window 的回调
        w.setCallback(this);
        // 设置 Window 的 WindowManager 对象
        w.setWindowManager(mWindowManager, null, null);
        // 代码省略
}
```

可以看到最终是通过 Window 对象的 setWindowManager 函数将 Window 对象与 WindowManager 建立了联系。该函数是在 Window 类中，我们看看这个函数实现。

```
public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
boolean hardwareAccelerated) {
        // 参数赋值等代码
        if (wm == null) {
                wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);
        }
        // 注意这里，调用了 createLocalWindowManager 函数
```

```
        mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);
}
```

最后一句代码很重要，调用的是 WindowManagerImpl 中的 createLocalWindowManage 方法，我们继续跟踪下去。

```
public WindowManagerImpl createLocalWindowManager(Window parentWindow) {
        return new WindowManagerImpl(mDisplay, parentWindow);
}
```

54 页中

```
public final class WindowManagerImpl implements WindowManager {
        private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();
        // Window 对象
        private final Window mParentWindow;

        // 代码省略

        private WindowManagerImpl(Display display, Window parentWindow) {
                mDisplay = display;
                mParentWindow = parentWindow;
        }

        @Override
        public void addView(View view, ViewGroup.LayoutParams params) {
                mGlobal.addView(view, params, mDisplay, mParentWindow);
        }

        @Override
        public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
                mGlobal.updateViewLayout(view, params);
        }

        @Override
        public void removeViewImmediate(View view) {
                mGlobal.removeView(view, true);
        }

        // 代码省略
}
```

54 页下到 55 页上

```
// 将 View 添加到 WindowManager 中，也就是在手机窗口中显示该 View
public void addView(View view, ViewGroup.LayoutParams params,
                    Display display, Window parentWindow) {
        // 检查参数有效性等代码省略
        ViewRootImpl root;
       View panelParentView = null;
       synchronized (mLock) {
              // 代码省略
              // 1、构建 ViewRootImpl
              root = new ViewRootImpl(view.getContext(), display);
              // 2、给 View 设置布局参数
              view.setLayoutParams(wparams);
              // 3、将 View 添加到 View 列表中
              mViews.add(view);
              // 4、将 ViewRootImpl 对象 root 添加到 mRoots 对象中
              mRoots.add(root);
              mParams.add(wparams);
        }

             // do this last because it fires off messages to start doing things
       try {
              // 5、调用 ViewRootImpl 的 setView 方法将 View 显示到手机窗口上
              root.setView(view, wparams, panelParentView);
        } catch (RuntimeException e) {

             }
   }
```

56 页

```
public static IWindowSession getWindowSession() {
       synchronized (WindowManagerGlobal.class) {
              if (sWindowSession == null) {
                     try {
                            InputMethodManager imm = InputMethodManager.getInstance();
```

```
                    // 1、获取 WindowManagerService
                    IWindowManager windowManager = getWindowManagerService();
                    // 2、与 WindowManagerSerice 建立一个 Session
                    sWindowSession = windowManager.openSession(
                            imm.getClient(), imm.getInputContext());
                } catch (RemoteException e) {
                            Log.e(TAG, "Failed to open window session", e);
                }
            }
            return sWindowSession;
        }
}

// 获取 WindowManagerService
public static IWindowManager getWindowManagerService() {
    synchronized (WindowManagerGlobal.class) {
        if (sWindowManagerService == null) {
            sWindowManagerService = IWindowManager.Stub.asInterface(
            ServiceManager.getService("window"));
            }
        return sWindowManagerService;
    }
}
```

在 getWindowSession 函数中，首先 Framework 层首先通过 getWindowManagerService 函数获取到 IWindowManager 对象，该函数中通过 ServiceManager.getService 函数获取 WMS 并且将 WMS 转换为 IWindowManager 类型。我们先看看 ServiceManager.getService 函数代码：

```
public static IBinder getService(String name) {
    try {
        IBinder service = sCache.get(name);
        if (service != null) {
            return service;
        } else {
            return getIServiceManager().getService(name);
        }
      } catch (RemoteException e) {
        Log.e(TAG, "error in getService", e);
      }
    return null;
}
```

57 页

与 WMS 建立 Session 之后就到了调用 ViewRootImpl 的 setView 方法了，该方法会向 WMS 发起显示 Dialog 或者 Activity 中的 DecorView 的请求。代码如下 ：

```
public void setView(View view, WindowManager.LayoutParams attrs,
                View panelParentView) {
    synchronized (this) {
        // 1、请求布局
        requestLayout();
        try {
            // 2、向 WMS 发起请求
            res = sWindowSession.add(mWindow, mWindowAttributes,
            getHostVisibility(), mAttachInfo.mContentInsets);
        }
}
```

setView很复杂，但是我们主要关注两步。

1. requestLayout;
2. 向WMS发起显示当前Window的请求。

我们先看看requestLayout 函数。

```
public void requestLayout() {
    checkThread();
    mLayoutRequested = true;
    scheduleTraversals(); //发送 DO_TRAVERSAL 消息

}

public void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        sendEmptyMessage(DO_TRAVERSAL);
    }
}
```

57 页下到 59 页上

```
private void performDraw() {
        if (!mAttachInfo.mScreenOn && !mReportNextDraw) {
                return;
         }

        final boolean fullRedrawNeeded = mFullRedrawNeeded;
        mFullRedrawNeeded = false;
        // 代码省略
        mIsDrawing = true;
        try {
              // 调用绘制函数
               draw(fullRedrawNeeded);
         } finally {
               mIsDrawing = false;
         }

              // 代码省略
}

private void draw(boolean fullRedrawNeeded) {
       // 1、获取绘制表面
       Surface surface = mSurface;
       if (!surface.isValid()) {
               return;
        }
       // 代码省略

       // 2、绘图表面需要更新
       if (!dirty.isEmpty() || mIsAnimating) {
              // 3、使用 GPU 绘制，也就是硬件加速
              if (attachInfo.mHardwareRenderer != null && attachInfo.mHardwareRenderer.isEnabled()) {
                    // 代码省略
                    // 使用硬件渲染器绘制
                     attachInfo.mHardwareRenderer.draw(mView, attachInfo, this,
                     animating ? null : mCurrentDirty);
                } else {
                     // 代码省略
                     // 4、使用 CPU 绘制图形
                     if (!drawSoftware(surface, attachInfo, yoff, scalingRequired, dirty)) {
                           return;
                      }
              }
```

```
        }

        if (animating) {
            mFullRedrawNeeded = true;
            scheduleTraversals();
        }
    }
```

在 draw 函数中会获取到需要绘制的区域以及是否使用 GPU 进行绘制，通常情况下使用的是 CPU 绘制，也就是调用的是 drawSoftware 函数来绘制。我们看看该函数的实现。

```
// 使用 CPU 绘制图形
private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int yoff,
        boolean scalingRequired, Rect dirty) {
        // Draw with software renderer.
         Canvas canvas;
        try {
            int left = dirty.left;
            int top = dirty.top;
            int right = dirty.right;
            int bottom = dirty.bottom;
            // 1、获取指定区域的 Canvas 对象，用于 Framework 层绘图
            canvas = mSurface.lockCanvas(dirty);
                // 代码省略
        } // 省略 catch

        try {
             // 代码省略
            try {
                // 2、从 DecorView 开始绘制，也就是整个 Window 的根视图，这会引起整棵树的重绘操作。
                 mView.draw(canvas);
                // 代码省略
            } finally {
             }
         } finally {
            try {
                // 3、释放 Canvas 锁，然后通知 SurfaceFlinger 更新这块区域
                surface.unlockCanvasAndPost(canvas);
            } catch (IllegalArgumentException e) {
                 return false;
            }
        }
```

```
        return true;
}
```

60 页到 61 页上

```
public class ImageLoader {
    // 图片缓存
    ImageCache mImageCache = new MemoryCache();
    // 图片加载中显示的图片 id
    int mLoadingImageId ;
    // 加载失败时显示的图片 id
    int mLoadingFailedImageId ;
    // 图片加载策略
    LoaderPolicy mLoaderPolicy ;

    // 线程池,线程数量为 CPU 的数量
    ExecutorService mExecutorService = Executors.newFixedThreadPool(Runtime.getRuntime()
.availableProcessors());

    // 省略单例模式部分代码

    public void displayImage(String imageUrl, ImageView imageView) {
        Bitmap bitmap = mImageCache.get(imageUrl);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
            return;
        }
        // 提交图片加载请求
        submitLoadRequest(imageUrl, imageView);
    }

    public void setImageCache(ImageCache cache) {
        mImageCache = cache;
    }

    public void setLoadingImage(int resId) {
        mLoadingImageId = resId ;
    }

    public void setLoadingFailedImage(int resId) {
```

```
        mLoadingFailedImageId = resId ;
    }

    public void setLoadingPolicy(LoaderPolicy policy) {
        mLoaderPolicy = policy ;
    }

    public void setThreadCount(int count) {
        mExecutorService.shutdown();
        mExecutorService = null;
        // 设置新的线程数量
        mExecutorService = Executors.newFixedThreadPool(count);
    }

    private void submitLoadRequest(final String imageUrl, final ImageView imageView) {
        // 设置加载中的图片
        imageView.setImageResource(mLoadingImageId);

        imageView.setTag(imageUrl);
        mExecutorService.submit(new Runnable() {

            @Override
            public void run() {
                // 加载图片
                Bitmap bitmap = downloadImage(imageUrl);
                if (bitmap == null) {
                    // 设置加载图片失败后显示的图片
                    imageView.setImageResource(mLoadingFailedImageId);
                    return;
                }
                // 显示加载到的图片
            }
        });
    }

    public Bitmap downloadImage(String imageUrl) {
        Bitmap bitmap = null;
        // 省略下载图片过程
        return bitmap;
    }
}
```

61 下到 62 页

```
ImageLoaderConfiguration config = new
     ImageLoaderConfiguration.Builder(context)
    .threadPriority(Thread.NORM_PRIORITY - 2)
    .denyCacheImageMultipleSizesInMemory()
    .discCacheFileNameGenerator(new Md5FileNameGenerator())
    .tasksProcessingOrder(QueueProcessingType.LIFO)
    .writeDebugLogs() // Remove for release app
    .build();
// 通过初始化ImageLoader
ImageLoader.getInstance().init(config);
```

这就是知名图片加载库 Universal-Image-Loader 的初始化配置代码。

莫名，我就是仇恨你！别人写的代码怎么总是那么好！原来让小民眼前一亮的是 Builder 模式，小民想起了前几天刚学习的这个模式。真是“山重水复疑无路，柳暗花明又一村。”小民显然已经找到了答案，“将一个复杂对象的构建与它的表示分离”，然后 Builder 模式来构建一个不可变的配置对象，并且将这个配置对象注入到 ImageLoader 中。也就是说它只能在构建的时候设置各个参数，一旦你调用 build()或者类似方法构建对象之后，它的属性就不再可修改。因为它没有 setter 方法，字段也都是隐藏的。用户只能在初始化通过一次性构造这个配置对象，然后注入给 ImageLoader。ImageLoader 根据配置对象进行初始化。这样，上一个版本中的 setThreadCount、setImageCache 等方法就不需要出现在 ImageLoader 中了，用户可见的函数就会少很多，ImageLoader 的使用成本也随之降低了。

我们看看修改过后的 ImageLoader 吧，代码如下 ：

```
public final class ImageLoader {
    // 图片加载配置对象
    private ImageLoaderConfig mConfig;

    // 省略单例模式的代码

    /**初始化 ImageLoader
     * @param config
     */
    public void init(ImageLoaderConfig config) {
        mConfig = config;
        // 检测配置的合法性,内部会根据配置做一些初始化操作
        checkConfig();
        // 代码省略
    }

    // 加载图片的函数
    public void displayImage(String imageUrl, ImageView imageView) {
        Bitmap bitmap = mImageCache.get(imageUrl);
```

```
            if (bitmap != null) {
                imageView.setImageBitmap(bitmap);
                return;
            }
            // 添加加载请求
            submitLoadRequest(imageUrl, imageView);
        }


        private void submitLoadRequest(final String imageUrl, final ImageView imageView) {
            // 代码省略
        }

        public Bitmap downloadImage(String imageUrl) {
            Bitmap bitmap = null;
            // 代码省略
            return bitmap;
        }

}
```

63 到 64 页

```
public class ImageLoaderConfig {

    // 图片缓存配置对象
    BitmapCache bitmapCache = new MemoryCache();
    // 加载图片时的 loading 和加载失败的图片配置对象
    DisplayConfig displayConfig = new DisplayConfig();
    //加载策略
    LoadPolicy loadPolicy = new SerialPolicy();
    // 线程数量,默认为 CPU 数量+ 1
    int threadCount = Runtime.getRuntime().availableProcessors() + 1;


    private ImageLoaderConfig() {
    }

    /**
     * 配置类的 Builder
     */
```

```
public static class Buider {

    /**
     * 图片缓存配置对象
     */
    BitmapCache bitmapCache = new MemoryCache();

    /**
     * 加载图片时的 loading 和加载失败的图片配置对象
     */
    DisplayConfig displayConfig = new DisplayConfig();
    /**
     * 加载策略
     */
    LoadPolicy loadPolicy = new SerialPolicy();
    // 线程数量
     int threadCount = Runtime.getRuntime().availableProcessors() + 1;

    // 设置线程数量
     public Buider setThreadCount(int count) {
            threadCount = Math.max(1, count);
            return this;
    }

    // 设置缓存
     public Buider setCache(BitmapCache cache) {
            bitmapCache = cache;
            return this;
    }

    // 设置图片加载中显示的图片
     public Buider setLoadingPlaceholder(int resId) {
            displayConfig.loadingResId = resId;
            return this;
    }

    // 设置要加载的图片加载失败时显示的图片
     public Buider setNotFoundPlaceholder(int resId) {
            displayConfig.failedResId = resId;
            return this;
    }

    // 设置加载策略
     public Buider setLoadPolicy(LoadPolicy policy) {
```

```
            if (policy != null) {
                loadPolicy = policy;
            }
            return this;
        }

        void applyConfig(ImageLoaderConfig config) {
            config.bitmapCache = this.bitmapCache;
            config.displayConfig = this.displayConfig;
            config.loadPolicy = this.loadPolicy;
            config.threadCount = this.threadCount;
        }

        /**
         * 根据已经设置好的属性创建配置对象
         *
         * @return ImageLoaderConfig 对象
         */
        public ImageLoaderConfig create() {
            ImageLoaderConfig config = new ImageLoaderConfig();
            // 应用配置
            applyConfig(config);
            return config;
        }
    }
}
```

通过将 ImageLoaderConfig 的构造函数、字段私有优化，使得外部不能访问内部属性。用户唯一能够设置属性的地方就是通过 Builder 对象了。也就是说用户只能通过 Builder 对象构造 ImageLoaderConfig 对象，就是构建和表示相分离嘛！

但是下半句“使得相同的构建过程可以创建不同的表示”又是何解呢？在经典的 Builder 模式中还有一个 Director 和 ConcreteBuilder 角色，不同的 ConcreteBuilder 是可以创建不同的 Product 子类的，因此也就是可以创建不同的表示。我们这里并没有使用经典实现，因此不做过多的描述。

既然小民已经完成了使用 Builder 模式的这个重构，那我们看看用户的使用代码是怎样的吧。代码如下 ：

```
private void initImageLoader() {
    ImageLoaderConfig config = newImageLoaderConfig.Buider()
        .setLoadingPlaceholder(R.drawable.loading)
        .setNotFoundPlaceholder(R.drawable.not_found)
        .setCache(new DoubleCache(this))
        .setThreadCount(4)
```

```
        .setLoadPolicy(new ReversePolicy()).create();
    // 将配置初始化到ImageLoader中
    ImageLoader.getInstance().init(config);
}
```

第七章 118 页到 120 页上

```
public class PriceCalculator {
    // 公交车类型
    private static final in tBUS = 1;
    // 地铁类型
    private static final int SUBWAY = 2;

    public static void main(String[] args) {
        PriceCalculator calculator = new PriceCalculator();
        System.out.println("坐16公里的公交车票价为 : " +
             calculator.calculatePrice(16, BUS));
        System.out.println("坐16公里的地铁票价为 : " +
             calculator.calculatePrice(16, SUBWAY));
    }

    /**
     * 北京公交车,10里之内1块钱,超过十公里之后每加一块钱可以乘5公里
     * @paramkm 公里
     * @return
     */
    private int busPrice(intkm) {
        // 超过10公里的总距离
        int extraTotal = km - 10;
        // 超过的距离是5公里的倍数
        int extraFactor = extraTotal / 5;
        // 超过的距离对5公里的取余
        int fraction = extraTotal % 5;
         // 价格计算
        int price = 1 + extraFactor * 1;
        return fraction > 0 ? ++price : price ;
    }

    /**
     * 6公里(含)内3元；6-12公里(含)4元；12-22公里(含)5元；22-32公里(含)6元；
     * @paramkm 公里
```

```
     * @return
     */
    privateintsubwayPrice(intkm) {
        if (km<= 6) {
           return 3;
        } elseif (km> 6 &&km< 12) {
           return 4;
        } elseif (km> 12 &&km< 22) {
           return 5;
        } else if (km> 22 &&km< 32) {
           return 6;
        }
         // 其他距离我们简化为7块
         return 7;
    }

    Int calculatePrice(intkm, int type) {
       if (type == BUS) {
          return busPrice(km);
        } elseif (type == SUBWAY) {
          return subwayPrice(km);
        }
       return 0;
     }
}
```

PriceCalculator 类很明显的问题就是并不是单一职责，首先它承担了计算公交车和地铁乘坐价格的职责。另一个问题就是通过 if-else 的形式来判断使用哪种计算形式，那么当我们增加一种出行形式时，例如出租车。那么我们就需要在 PriceCalculator 中增加一个方法来计算出租车出行的价格，并且在 calculatePrice(int km, int type)函数中增加一个判断，代码大致添加如下 ：

```
public class PriceCalculator {
    // 公交车类型
    private static final int BUS = 1;
    // 地铁类型
    private static final int SUBWAY = 2;
    // 出租车类型
    private static final int TAXI = 3;



    /**
     * 简单计算为每公里2块钱吧
     *
```

```
     * @param km
     * @return
     */
    private int taxiPrice(int km) {
        return km * 2;
    }

    int calculatePrice(int km, int type) {
        if (type == BUS) {
            return busPrice(km);
        } else if (type == SUBWAY) {
            return subwayPrice(km);
        } else if (type == TAXI) {
            return taxiPrice(km) ;
        }
        return 0 ;
    }
}
```

第八章 146 页到 147 上

```
    /**
     * 电视遥控器，含有开机、关机、下一频道、上一频道、调高音量、调低音量这几个功能
     */
    public class TvController {
        // 开机状态
        private final static int POWER_ON = 1;
        // 关机状态
        private final static int POWER_OFF = 2;
        private int mState = POWER_OFF;

        public void powerOn() {
            mState = POWER_ON;
            if (mState == POWER_OFF) {
                System.out.println("开机啦");
            }
        }

        public void powerOff() {
             mState = POWER_OFF;
            if (mState == POWER_ON) {
```

```
            System.out.println("关机啦");
        }
    }

    public void nextChannel() {
        if (mState == POWER_ON) {
            System.out.println("下一频道");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }

    public void prevChannel() {
        if (mState == POWER_ON) {
            System.out.println("上一频道");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }

    public void turnUp() {
        if (mState == POWER_ON) {
            System.out.println("调高音量");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }

    public void turnDown() {
        if (mState == POWER_ON) {
            System.out.println("调低音量");
        } else {
            System.out.println("两个红灯提示没有开机");
        }
    }
}
```

162 页中

```
/**
 * LoginContext，用户接口和状态管理类
 */
```

```
public class LoginContext {
    // 用户状态，默认为未登录状态
    UserState mState = new LogoutState();
    // 单例
    static LoginContext sLoginContext = new LoginContext();

    privateLoginContext() {
    }

    public static LoginContext getLoginContext() {
        return sLoginContext;
    }

    public void setState(UserState aState) {
        this.mState = aState;
    }

    // 转发
    public void forward(Context context) {
        mState.forward(context);
    }

    public void comment(Context context) {
        mState.comment(context);
    }

}
```

第十三章 248 页到 250 上

```
/**
 * 使命召唤游戏. ( 数据模型不太合理，这里我们只是简单演示 )
 */
public class CallOfDuty {
    Private int mCheckpoint = 1;
    Private int mLifeValue = 100;
    private String mWeapon = "沙漠之鹰";

        // 打游戏
    public void play() {
```

```
        System.out.println("打游戏 : " + String.format("第%d关", mCheckpoint) + " 奋战杀敌中");
        mLifeValue -= 10;
        System.out.println("进度升级啦");
        mCheckpoint++;
        System.out.println("到达 " + String.format("第%d关", mCheckpoint));
    }
    // 退出游戏
    public void quit() {
        System.out.println("--------------");
        System.out.println("退出前的游戏属性 : " + this.toString());
        System.out.println("退出游戏");
        System.out.println("--------------");
    }

        /**
         * 创建备忘录
         */
    public Memoto createMemoto() {
        Memoto memoto = newMemoto();
        memoto.mCheckpoint = mCheckpoint;
        memoto.mLifeValue = mLifeValue;
        memoto.mWeapon = mWeapon;
        return memoto;
    }
        // 恢复游戏
    public void restore(Memoto memoto) {
        this.mCheckpoint = memoto.mCheckpoint;
        this.mLifeValue = memoto.mLifeValue;
        this.mWeapon = memoto.mWeapon;
         System.out.println("恢复后的游戏属性 : " + this.toString());
    }

    // setter 和 getter 省略

        @Override
    public String toString() {
        return "CallOfDuty [mCheckpoint=" + mCheckpoint + ", mLifeValue=" + mLifeValue
                + ", mWeapon=" + mWeapon + "]";
    }
}
```

在 CallOfDuty 游戏类中，我们存储了几个关键字段，关卡、人物的生命值、武器，当调用

play 函数打游戏时，我们对关卡和人物的生命值进行修改。在该类中可以通过 createMemoto 函数来创建该用户的备忘录对象，也就是将自身的状态保存到一个 Memoto 对象中。外部可以通过 restore 函数将 CallOfDuty 对象的状态从备忘录对象中恢复。

我们先来看看备忘录对象，它只是存储 CallOfDuty 对象的字段，代码如下 ：

```
/**
 * 备忘录类
 */
public class Memoto {
    public int mCheckpoint;
    public int mLifeValue;
    public String mWeapon;

    @Override
    public String toString() {
        return "Memoto [mCheckpoint=" + mCheckpoint + ", mLifeValue=" + mLifeValue +
                ", mWeapon=" + mWeapon + "]";
    }

}
```

这是一个无状态、无操作的实体类，只负责用来存储 Originator 角色的一些数据，防止外部直接访问 Originator。

而备忘录的操作者则是 Caretaker 角色，我们看看相关的代码。

```
/**
 * Caretaker,负责管理 Memoto
 */
public class Caretaker {
    Memoto mMemoto;  // 备忘录

    /**
     * 存档
     */
    public void archive(Memoto memoto) {
        this.mMemoto = memoto;
    }

    /**
     * 获取存档
     */
    public Memoto getMemoto() {
        return mMemoto;
    }
}
```

251 到 253 页上

onSaveInstanceState 和 onRestoreInstanceState 都是 Activity 中的函数，我们首先来分析 onSaveInstanceState。

```
protected void onSaveInstanceState(Bundle outState) {
        // 1、存储当前窗口的视图树的状态
        outState.putBundle(WINDOW_HIERARCHY_TAG, mWindow.saveHierarchyState());
        // 2、存储 Fragment 的状态
        Parcelable p = mFragments.saveAllState();
        if (p != null) {
            outState.putParcelable(FRAGMENTS_TAG, p);
        }
        // 3、如果用户还设置了 Activity 的 ActivityLifecycleCallbacks，
        // 那么调用这些 ActivityLifecycleCallbacks 的 onSaveInstanceState 进行存储状态
        getApplication().dispatchActivitySaveInstanceState(this, outState);
 }
```

上述 onSaveInstanceState 函数中，主要分为三步:

1. 存储窗口的视图树的状态；
2. 存储 Fragment 的状态；
3. 调用 Activity 的 ActivityLifecycleCallbacks 的 onSaveInstanceState 函数进行状态存储。

我们首先来看第一步，在这一步中将 Window 对象中的视图树中各个 View 状态存储到 Bundle 中，这样一来，当用户重新进入到该 Activity 中时用户 UI 的结构、状态才会被重新恢复，以此来保证用户界面的一致性。Window 类的具体实现类是 PhoneWindow，我们看看 PhoneWindow 中 saveHierarchyState 函数的实现:

```
    @Override
public Bundle saveHierarchyState() {
     Bundle outState = new Bundle();
    if (mContentParent == null) {
        return outState;
     }

        // 通过 SparseArray 类来存储，这相当于一个 key 为整型的 map
        SparseArray<Parcelable> states = new SparseArray<Parcelable>();
        // 调用 mContentParent 的 saveHierarchyState 方法，这个 mContentParent 就是我们调用 Activity 的
        // setContentView 函数设置的内容视图，它是我们内容试图的根节点，在这里存储整棵视图树的结构
        mContentParent.saveHierarchyState(states);
        // 将视图树结构放到 outState 中
        outState.putSparseParcelableArray(VIEWS_TAG, states);
```

```
        // 保存当前界面中获取了焦点的 View
        View focusedView = mContentParent.findFocus();
        if (focusedView != null) {
            // 持有焦点的 View 必须要设置 id,否则重新进入该界面时不会恢复它的焦点状态
            if (focusedView.getId() != View.NO_ID) {
                outState.putInt(FOCUSED_ID_TAG, focusedView.getId());
            } else {
                // 代码省略
            }
        }

        // 存储整个面板的状态
        SparseArray<Parcelable> panelStates = new SparseArray<Parcelable>();
        savePanelState(panelStates);
        if (panelStates.size() > 0) {
            outState.putSparseParcelableArray(PANELS_TAG, panelStates);
        }
        // 存储 ActionBar 的状态
        if (mActionBar != null) {
            SparseArray<Parcelable> actionBarStates = new SparseArray<Parcelable>();
            mActionBar.saveHierarchyState(actionBarStates);
            outState.putSparseParcelableArray(ACTION_BAR_TAG, actionBarStates);
        }
        return outState;
    }
```

在 saveHierarchyState 函数中，主要是存储了当前 UI、ActionBar 的相关 View 的状态，这里我们拿 mContentParent 来分析。这个 mContentParent 就是我们通过 Activity 的 setContentView 函数设置的内容视图，它是我们整个内容视图的根节点，存储它层级结构中的 View 状态也就存储了用户界面的状态。mContentParent 是一个 ViewGroup 对象，但是 saveHierarchyState 并不是在 ViewGroup 中，而是在 ViewGroup 的父类 View 中，我们看看:

```
    public void saveHierarchyState(SparseArray<Parcelable> container) {
        // 又调用了 dispatchSaveInstanceState 来存储状态
        dispatchSaveInstanceState(container);
    }

    /**
     * 真正存储 View 状态的函数
     */
    protected void dispatchSaveInstanceState(SparseArray<Parcelable> container) {
        // 1、注意这里: 如果 View 没有设置 id，那么这个 View 的状态将不会被存储。
        if (mID != NO_ID && (mViewFlags & SAVE_DISABLED_MASK) == 0) {
```

```
                mPrivateFlags&= ~PFLAG_SAVE_STATE_CALLED;
                // 2、调用 onSaveInstanceState 函数获取自身的状态
                Parcelable state = onSaveInstanceState();
                 if ((mPrivateFlags & PFLAG_SAVE_STATE_CALLED) == 0) {
                        throw new IllegalStateException(
                                "Derived class did not call super.onSaveInstanceState()");
                }
                 if (state != null) {
                     //   3、将自身状态放到 container 中，key 为 id，value 为自身状态
                      container.put(mID, state);
                }
        }

       // View 类默认存储的状态为空
       protected Parcelable onSaveInstanceState() {
            mPrivateFlags |= PFLAG_SAVE_STATE_CALLED;
            return BaseSavedState.EMPTY_STATE;
        }
}
```

在 View 类中的 saveHierarchyState 函数调用了 dispatchSaveInstanceState 函数来存储自身的状态。而 ViewGroup 则覆写了 dispatchSaveInstanceState 函数来存储自身以及子视图的状态。ViewGroup 的 dispatchSaveInstanceState 函数如下:

```
        @Override
        protected void dispatchSaveInstanceState(SparseArray<Parcelable> container) {
            super.dispatchSaveInstanceState(container);
            final int count = mChildrenCount;
            final View[] children = mChildren;
            for (int i = 0; i < count; i++) {
                 View c = children[i];
                 if ((c.mViewFlags & PARENT_SAVE_DISABLED_MASK) != PARENT_SAVE_DISABLED) {
                      c.dispatchSaveInstanceState(container);
                 }
             }
}
```

253 页下到 254 上

```
@Override
public Parcelable onSaveInstanceState() {
```

```
 Parcelable superState = super.onSaveInstanceState();

// Save state if we are forced to
boolean save = mFreezesText;
int start = 0;
int end = 0;

if (mText != null) {
        start = getSelectionStart();
        end = getSelectionEnd();
        if (start >= 0 || end >= 0) {
                save = true;
         }
 }

// 存储 TextView 的 start, end 以及文本内容
if (save) {
       SavedState ss = new SavedState(superState);
       // XXX Should also save the current scroll position!
        ss.selStart = start;
        ss.selEnd = end;

        if (mText instanceof Spanned) {
                Spannable sp = new SpannableString(mText);
                for (ChangeWatcher cw : sp.getSpans(0, sp.length(), ChangeWatcher.class)) {
                        sp.removeSpan(cw);
                }

                if (mEditor != null) {
                        removeMisspelledSpans(sp);
                        sp.removeSpan(mEditor.mSuggestionRangeSpan);
                }

                ss.text = sp;
         } else {
                ss.text = mText.toString();
         }

        if (isFocused() && start >= 0 && end >= 0) {
                ss.frozenWithFocus = true;
         }
        ss.error = getError();
        // 返回状态对象
        return ss;
```

```
        }

        return superState;
    }
```

调用了 View 的 onSaveInstanceState 函数之后我们就得到了 View 要存储的数据，此时执行到注释 3 处，这里以 View 的 id 为 key,以状态为值存储到 container (SparseArray 类型)中。

```
if (state != null) {
        //  3、将自身状态放到 container 中，key 为 id，value 为自身状态
        container.put(mID, state);
    }
```

255 页

```
    final void performStopActivity(IBinder token, boolean saveState) {
        // 获取 ActivityClientRecord
        ActivityClientRecord r = mActivities.get(token);
        // 执行 performStopActivityInner，saveState 就是表示是否要存储状态
        performStopActivityInner(r, null, false, saveState);
     }

    private void performStopActivityInner(ActivityClientRecord r,
            StopInfo info, boolean keepShown, boolean saveState) {
        if (r != null) {
            if (!r.activity.mFinished && saveState) {
                if (r.state == null) {
                    // 执行 ACtivity 的 OnSaveInstanceState 函数
                    callCallActivityOnSaveInstanceState(r);
                }
            }
            if (!keepShown) {
                try {
                    // 执行 Activity 的 onStop 函数
                    r.activity.performStop();
                } catch (Exception e) {
                }
                r.stopped = true;
              }
            r.paused = true;
        }
    }

    private void callCallActivityOnSaveInstanceState(ActivityClientRecord r) {
```

```
            r.state = new Bundle();
            r.state.setAllowFds(false);
            if (r.isPersistable()) {
                r.persistentState = new PersistableBundle();
                mInstrumentation.callActivityOnSaveInstanceState(r.activity, r.state,
                r.persistentState);
             } else {
                // 调用 Instumentation 的 callActivityOnSaveInstanceState 函数，
                // 实际上会调用 Activity 的 onSaveInstanceState 函数
                mInstrumentation.callActivityOnSaveInstanceState(r.activity, r.state);
            }
}
```

256 页

```
    private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
        // 代码省略
        Activity activity = null;
        try {
            java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
            // 1、构建 Activity
            activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
            // 代码省略
         } catch (Exception e) {
         }

        try {
            // 2、创建一个 Application
            Application app = r.packageInfo.makeApplication(false, mInstrumentation);
            if (activity != null) {
                // 创建 COntext,类型为 ContextImpl
                Context appContext = createBaseContextForActivity(r, activity);
                // 3、关联      appContext、Application 对象到 Activity 中
                 activity.attach(appContext, this, getInstrumentation(), r.token,
                        r.ident, app, r.intent, r.activityInfo, title, r.parent,
                        r.embeddedID, r.lastNonConfigurationInstances, config);

                // 4、调用 Activity 的 onCreate 方法
                 mInstrumentation.callActivityOnCreate(activity, r.state);
                 // 5、调用 Activity 的 OnRestoreInstanceState 恢复状态
```

```
                    if (!r.activity.mFinished) {
                        if (r.isPersistable()) {
                            if (r.state != null || r.persistentState != null) {
                                mInstrumentation.callActivityOnRestoreInstanceState(activity,
                                    r.state,r.persistentState);
                            }
                        } else if (r.state != null) {
                            mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
                        }
                    }
                  }
                r.paused = true;
                // 6、将 Activity 的信息记录对象存到 mActivities 中
                mActivities.put(r.token, r);

            } catch (SuperNotCalledException e) {
                    throw e;
            } catch (Exception e) {
            }
        return activity;
}
```

258 到 259

```
Bundle saveFragmentBasicState(Fragment f) {
    Bundle result = null;

    if (mStateBundle == null) {
        mStateBundle = new Bundle();
    }
    f.performSaveInstanceState(mStateBundle);
    if (!mStateBundle.isEmpty()) {
        result = mStateBundle;
        mStateBundle = null;
    }

    if (f.mView != null) {
        saveFragmentViewState(f);
     }
    if (f.mSavedViewState != null) {
        if (result == null) {
            result = new Bundle();
```

```
            }
            result.putSparseParcelableArray(
                        FragmentManagerImpl.VIEW_STATE_TAG, f.mSavedViewState);
        }
        if (!f.mUserVisibleHint) {

            // 此时 result 为空，这里会报异常
            result.putBoolean(FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
                    f.mUserVisibleHint);
        }

        return result;
    }
```

当 saveFragmentBasicState 函数中的最后 if 为真且 result 还未被初始化时就被抛出这个异常，解决的方法就是把最后一个 if 修改如下 ：

```
if (!f.mUserVisibleHint) {
    if (result == null) {
        result = new Bundle();
    }
    // 此时 result 为空，这里会报异常
    result.putBoolean(FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
            f.mUserVisibleHint);
}
```

但这是从源码的角度修复 bug，在开发过程中你通常不会采用这种形式。此时，我们的解决这个问题的方法两个，第一覆写 FragmentActivity 中的 onSaveInstanceState 方法，不调用父类中的 onSaveInstanceState，也就是说不保存该 Activity 中所有 View 的状态。这样也就不会调用到上述奔溃的代码段了。代码如下 ：

```
@Override
  protected void onSaveInstanceState(Bundle outState) {
      //super.onSaveInstanceState(outState);

  }
```

还有一种方法就是覆写 Fragment 中的 onSaveInstanceState 方法，在这个方法中随便存入一个数据，带入如下 ：

```
public class MyFragment extends Fragment {
    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        // 随便存入一个数据
        outState.putString("DO NOT CRASH", "Baby");
    }
```

```
    // 代码省略
}
```

这样保证了原来在 saveFragmentBasicState 中引发奔溃的 result 对象被指向 Fragment 存储状态信息的 Bundle 对象中，源码逻辑如下 ：

```
Bundle saveFragmentBasicState(Fragment f) {
     Bundle result = null;
    if (mStateBundle == null) {
        mStateBundle = new Bundle();
     }
    f.performSaveInstanceState(mStateBundle);
    // Fragment 的 mStateBundle 不为空，此时 result 指向了 mStateBundle
    if (!mStateBundle.isEmpty()) {
        // result 此时指向了 mStateBundle
        result = mStateBundle;
        mStateBundle = null;
     }
    // 代码
    if (!f.mUserVisibleHint) {
        // 此时 result 为空，这里会报异常
        result.putBoolean(FragmentManagerImpl.USER_VISIBLE_HINT_TAG,
            f.mUserVisibleHint);
     }

    return result;
}
```

260 中到 267

布局的 xml 如下 ：

```
<LinearLayout 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:context="${relativePackage}.${activityClass}" >

    <EditText
    android:id="@+id/note_edittext"
    android:layout_width="match_parent"
```

```
    android:layout_height="0dp"
    android:layout_weight="1"
    android:gravity="left"
    android:hint="@string/note_tips" />

    <RelativeLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:paddingLeft="50dp"
    android:paddingRight="50dp" >

        <ImageView
        android:id="@+id/undo_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:src="@drawable/undo" />

        <TextView
        android:id="@+id/save_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="@string/save"
        android:textSize="20sp" />

        <ImageView
        android:id="@+id/redo_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:src="@drawable/redo" />
    </RelativeLayout>

</LinearLayout>
```

这是 MainActivity 中的布局，因此相关的代码写在 MainActivity 中。小民首先初始 View，设置点击事件等，代码如下所示 ：

```
public class MainActivity extends Activity {

    EditText mNodeEditText;  // 编辑器
    TextView mSaveTv;        // 保存按钮
    ImageView mUndoBtn;      // 撤销按钮
```

```
ImageView mRedoBtn;      // 重做按钮

// 最大存储数量
private static final int MAX = 30;
// 存储 30 条记录
List<Memoto> mMemotos = new ArrayList<Memoto>(MAX);

int mIndex = 0;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // 初始化视图
    initViews();
}

private void initViews() {
    mNodeEditText = (EditText) findViewById(R.id.note_edittext);
    mUndoBtn = (ImageView) findViewById(R.id.undo_btn);
    mUndoBtn.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // 返回上一个记录点
            restoreEditText(getPrevMemoto());
            makeToast("撤销 : ");
        }
    });

    mRedoBtn = (ImageView) findViewById(R.id.redo_btn);
    mRedoBtn.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // 恢复状态,恢复到下一个记录点
            restoreEditText(getNextMemoto());
            makeToast("重做 : ");
        }
    });

    mSaveTv = (TextView) findViewById(R.id.save_btn);
    mSaveTv.setOnClickListener(new OnClickListener() {
```

```
            @Override
            public void onClick(View v) {
                saveMemoto(createMemotoForEditText());
                makeToast("保存笔记 : ");
            }
        });
    }
    private void makeToast(String msgPrex) {
        Toast.makeText(this,
                msgPrex + mNodeEditText.getText() + ", 光标位置 : "
                                +                mNodeEditText.getSelectionStart(),
        Toast.LENGTH_LONG).show();
    }
    // 代码省略
}
```

上述代码是初始化布局中的控件，并且为这些控件设置点击事件的处理。点击保存按钮时，会为编辑器创建一个 Memoto 对象用以保存编辑器的文本和光标位置，并且将这个 Memoto 存到一个列表当中。当用户点击撤销按钮时，从 Memoto 列表取出上一个记录，即恢复到上一个状态；用户点击重做按钮时，也就重复上一个撤销了的动作。这部分的代码如下所示 :

```
    public class MainActivity extends Activity {

        // 代码省略
         /**
         * 保存备忘录
         * @param memoto
         */
        public void saveMemoto(Memoto memoto) {
            if (mMemotos.size() > MAX) {
                mMemotos.remove(0);
             }
            mMemotos.add(memoto);
            mIndex = mMemotos.size() - 1;
        }

        // 获取上一个存档，相当于撤销功能
        public Memoto getPrevMemoto() {
            mIndex = mIndex > 0 ? --mIndex : mIndex;
            return mMemotos.get(mIndex);
        }

        // 获取下一个存档，相当于重做功能
        public Memoto getNextMemoto() {
```

```
            mIndex = mIndex < mMemotos.size() - 1 ? ++mIndex : mIndex;
            return mMemotos.get(mIndex);
        }

        // 为编辑器创建 Memoto 对象
        private Memoto createMemotoForEditText() {
             Memoto memoto = new Memoto();
            memoto.text = mNodeEditText.getText().toString();
            memoto.cursor = mNodeEditText.getSelectionStart();
            return memoto;
         }
        // 恢复编辑器状态
        private void restoreEditText(Memoto memoto) {
            mNodeEditText.setText(memoto.text);
            mNodeEditText.setSelection(memoto.cursor);
         }
  }
```

小民是个心细如麻的人，不光要存储编辑器的文本内容，还要把光标的位置也存储起来，这样避免手动移动光标，因此小民建立一个类来存储这些数据，上述的代码也是操作这个类。

```
/**
 * 存储EditText的文本与光标位置
 */
public class Memoto {
    public String text;
    publicint cursor;
}
```

第一版记事本就算差不多啦，作为一名程序员，小民的第一步自然是要输入著名的 Hello World。为了测试功能，在输入了 hello 之后特地保存了一次，然后再输入 World，如图 18-3、18-4 所示。

图 18-3

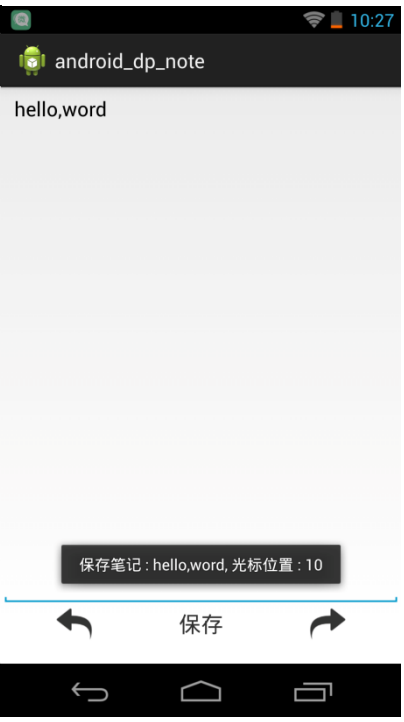


图 18-4

玩玩没想到啊！一向英语不太地道的小民把 world 写成了 word！幸好上一步做了保存，而且有撤销功能，小民点击了左下角的撤销按钮，返回上一状态，如图 18-5 所示。

图 18-5

最近新版本放上线，项目组相对悠闲了些。小民的主管近来也是没啥事，一转头就看到小民在认真地在捣鼓着手机，于是便到小民的工位来一探究竟。

“小民，干啥呢这是？”主管问道。

“老大，最近稍微悠闲点，我自己写个便签软件练练手。要不您给指导一下？”小民答道。

“好吧，反正最近也没啥事!”主管爽快的答应了，说着就拿过小民的鼠标在屏幕上操作起来。

“小民，你这个 MainActivity 的职责太混乱了吧！要负责 View 部分的逻辑，还有负责管理你的便签的记录、修改编辑器的状态，把这些代码耦合在一起会造成类型膨胀，后续难以维护，你的 MainActivity 也会越来越脆弱的。怎么刚学完六大原则就忘了吗？……像你这种需要保存对象状态的情况可以使用备忘录模式，把保存数据的逻辑、职责分离出去，这样耦合更低，每个类的职责也清晰。”把代码扫过一遍之后，主管面色有点凝重的补充到。

“主管，我这也是刚开始嘛，在设计上还没有进行那么多考虑，我待会儿是要重构的……”小民颇有点底气不足的回答到。

“那行，我也只是给你提供点建议嘛，那你等会再好好改改哈，我先忙我自己的了。”主管说完便笑嘻嘻地走开了。

备忘录模式，小民平时倒是看过相关的资料，但是还真没有运用吧，于是在补充了一些基本知识之后便开始对便签项目进行第一轮重构，主要是运用备忘录模式将管理编辑器数据的职责独立出来。首先建立了一个 NoteCaretaker 类来负责管理 Memoto 对象，代码如下 ：

```
// 负责管理 Memoto 对象
public class NoteCaretaker {
    // 最大存储数量
    private static final int MAX = 30;
    // 存储 30 条记录
    List<Memoto> mMemotos = new ArrayList<Memoto>(MAX);

    int mIndex = 0;

    /**
     * 保存备忘录到记录列表中
     * @param memoto
     */
    public void saveMemoto(Memoto memoto) {
        if (mMemotos.size() > MAX) {
            mMemotos.remove(0);
        }
        mMemotos.add(memoto);
        mIndex = mMemotos.size() - 1;
    }

    // 获取上一个存档，相当于撤销功能
    public Memoto getPrevMemoto() {
        mIndex = mIndex > 0 ? --mIndex : mIndex;
        return mMemotos.get(mIndex);
    }
```

```
    // 获取下一个存档，相当于重做功能
    public Memoto getNextMemoto() {
        mIndex = mIndex < mMemotos.size() - 1 ? ++mIndex : mIndex;
        return mMemotos.get(mIndex);
    }
}
```

在 NodeCaretaker 中会维护一个备忘录列表，然后使用 mIndex 标识编辑器当前所在的记录点，通过 getPrevMemoto、getNextMemoto 分别获取上一个、下一个记录点的备忘录，以此来达到撤销、重做的功能。如图 18-6 所示。

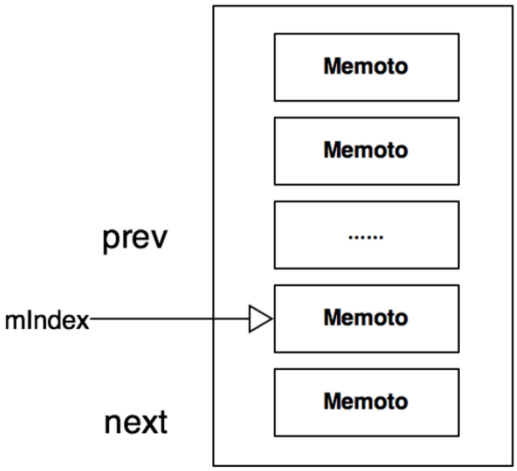


图 18-6

然后自定义了一个 NodeEditText 类，该类继承自 EditText。代码如下 ：

```
// Note 编辑器类
public class NoteEditText extends EditText {
    public NoteEditText(Context context) {
        this(context, null);
    }

    public NoteEditText(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public NoteEditText(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
    // 创建备忘录对象，即存储编辑器的指定数据
    public Memoto createMemoto() {
```

```
        Memoto noteMemoto = newMemoto();
        // 存储文本与光标位置
        noteMemoto.text = getText().toString();
        noteMemoto.cursor = getSelectionStart();
        return noteMemoto;
    }
    // 从备忘录中恢复数据
    public void restore(Memoto memoto) {
        setText(memoto.text);
        // 设置光标位置
        setSelection(memoto.cursor);
    }
}
```

该类就是添加了两个函数，分别是 createMemoto 和 restore 函数。createMemoto 函数是创建一个存储了编辑器文本、光标位置数据的 Memoto 对象，并且返回给客户端。restore 函数是从 Memoto 对象中恢复编辑器的文本和光标位置。通过添加这两个函数，使得 EditText 能够管理自身的状态信息。如果没有 NodeEditText，那么 createMemoto 和 restore 函数的功能就需要放到其他的类型中，而这两个功能放到其他类型又不太合适，例如 MainActivity，最重要的原因是这两个功能操作的都是 EditText 自身的字段，因此它属于这个类型。

最后，我们的 MainActivity 的代码就是如下这样了:

```
public class MainActivity extends Activity {

    NoteEditText mNodeEditText;
    TextView mSaveTv;
    ImageView mUndoBtn;
    ImageView mRedoBtn;
    // note备忘录管理器
    NoteCaretaker mCaretaker = newNoteCaretaker();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 初始化视图
        initViews();
    }

    private void initViews() {
        mNodeEditText = (NoteEditText) findViewById(R.id.note_edittext);
        mUndoBtn = (ImageView) findViewById(R.id.undo_btn);
        mUndoBtn.setOnClickListener(new OnClickListener() {

            @Override
```

```
            public void onClick(View v) {
                // 返回上一个记录点
                 mNodeEditText.restore(mCaretaker.getPrevMemoto());
                makeToast("撤销 : ");
            }
        });

        mRedoBtn = (ImageView) findViewById(R.id.redo_btn);
        mRedoBtn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // 恢复状态,恢复到下一个记录点
                 mNodeEditText.restore(mCaretaker.getNextMemoto());
                makeToast("重做 : ");
            }
        });

        mSaveTv = (TextView) findViewById(R.id.save_btn);
        mSaveTv.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                 mCaretaker.saveMemoto(mNodeEditText.createMemoto());
                makeToast("保存笔记 : ");
            }
        });
    }

    private void makeToast(String msgPrex) {
        Toast.makeText(this,
                msgPrex + mNodeEditText.getText() + ", 光标位置 : "
                                + mNodeEditText.getSelectionStart(),
            Toast.LENGTH_LONG).show();
    }
}
```

第 16 章 303 页到 305

```
// 员工基类
public abstract class Staff {
    public String name;
```

```
    // 员工KPI
    public int kpi;

    public Staff(String aName) {
        this.name = aName;
        kpi = new Random().nextInt(10);
    }
    // 接受Visitor的访问
    public abstract void accept(Visitor visitor);
}
```

Staff 类定义了员工的基本信息以及一个 accept 方法，accept 方法表示接受访问者的访问，由子类具体实现。下面看看工程师和经理的相关代码:

```
// 工程师
public class Engineer extends Staff {
    public Engineer(String aName) {
        super(aName) ;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    // 工程师这一年写的代码行数量
    public int getCodeLines() {
        return new Random().nextInt(10 * 10000);
    }
}

// 经理类型
public class Manager extends Staff {
    private int products;  // 产品数量
    public cManager(String aName) {
        super(aName);
        products = new Random().nextInt(10);
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    // 一年内做的产品数量
    public int getProducts() {
```

```
        return products;
    }
}
```

在工程师类中添加了获取代码行数的函数，而在经理类型中则添加了获取新产品数量的函数，它们的职责是不一样的，也正是由于它们的差异性才使得访问者模式能够发挥它的作用。Staff、Engineer、Manager 三个类型就是我们对象结构，这些类型相对稳定，不太会发生变化。

然后我们将这些员工添加到一个业务报表类中，公司高层可以通过该报表类的 showReport 函数查看所有员工的业绩。代码如下：

```
// 员工业务报表l类
public class BusinessReport {
    List<Staff> mStaffs = new LinkedList<Staff>();

    public BusinessReport() {
        mStaffs.add(new Manager("王经理"));
        mStaffs.add(new Engineer("工程师-Shawn.Xiong"));
        mStaffs.add(new Engineer("工程师-Kael"));
        mStaffs.add(new Engineer("工程师-Chaossss"));
        mStaffs.add(new Engineer("工程师-Tiiime"));
    }



    /**
     * 为访问者展示报表
     * @param visitor公司高层,例如CEO、CTO
     */
    public void showReport(Visitor visitor) {
        for (Staff staff : mStaffs) {
            staff.accept(visitor);
        }
    }
}
```

下面我们看看 Visitor 类型的定义，Visitor 声明了两个 visit 函数，分别是对工程师和经理的访问函数，代码如下：

```
public interface Visitor {
    // 访问工程师类型
    public void visit(Engineer engineer);
    // 访问经理类型
    public void visit(Manager leader);
}
```

首先定义了一个 Visitor 接口，该接口有两个 visit 函数，参数分别为 Engineer、Manager，也就是说对于 Engineer、Manager 的访问会调用两个不同的方法，以此达成区别对待、差异化处理。具体的实现类为 CEOVisitor、CTOVisitor 类，代码如下 ：

```
// CEO访问者,只关注业绩
public class CEOVisitor implements Visitor {
    @Override
    public void visit(Engineer engineer) {
       System.out.println("工程师 : " + engineer.name + ", KPI : " +
       engineer.kpi);
    }

    @Override
    public void visit(Manager mgr) {
        System.out.println("经理 : " + mgr.name + ", KPI : " + mgr.kpi +
             ", 新产品数量 : "
               + mgr.getProducts());
    }
}
```

在 CEO 的访问者中，CEO 只关注 Engineer 员工的 KPI，而对于 Manager 管用除了 KPI 之外还需要关注新产品的数量。两类员工的关注点略有不同，通过两个 visitor 方法分别进行处理。而如果我们不使用 Visitor 模式，只通过一个 visit 函数进行处理，那么我们就需要在这个 visit 函数中对不同的员工类型进行判断，然后分别进行处理，代码大致如下 ：

```
class ReportUitl {
    public void visit(Staff staff) {
        if (staff instanceof Manager) {
            Manager mgr = (Manager) staff;
               System.out.println("经理 : " + mgr.name + ", KPI : " + mgr.kpi
                  + ", 新产品数量 : "
                  + mgr.getProducts());
         } else {
              Engineerengineer = (Engineer) staff;
              System.out.println("工程师 : " + engineer.name + ", KPI : "
                   + engineer.kpi);
          }
     }
}
```

这就导致了 if-else 逻辑的嵌套以及类型的强制转换，难以扩展和维护，当类型较多时，这个 ReportUtil 就会变化混乱不堪。而使用 Visitor 模式，通过同一个函数对不同的元素类型进行相应的处理，使得结构更加清晰、灵活性更高。

例如，CEO 的关注点主要是在业绩上，而 CTO 更关注的是员工在技术层面的贡献。因此，我们添加一个 CTO 的 Visitor 类。

```
public class CTOVisitor implements Visitor {
```

```
    @Override
    public void visit(Engineer engineer) {
        System.out.println("工程师 : " + engineer.name + ", 代码函数 : " +
        engineer.getCodeLines());
    }

    @Override
    public void visit(Manager leader) {
        System.out.println("经理 : " + leader.name + "产品数量 : " +
              leader.getProducts());
    }
}
```

CTO 的 Visitor 中只关注工程师写的代码量，而对于经理只关注了他的新产品数量。不同的 visit 函数会对元素进行不同的操作，而通过注入不同的 Visitor 又可以替换掉访问者的具体实现，使得对元素的操作变得更灵活，可扩展性更高，同时也消除了类型转换、if-else 等”丑陋”的代码。

下面是客户端代码 :

```
    public class Client {
    public static void main(String[] args) {
        // 构建报表
        BusinessReport report = new BusinessReport();
        System.out.println("======== 给CEO看的报表 ====== ");
        // 设置访问者,这里是CEO
        report.showReport(new CEOVisitor());

        System.out.println("\n======== 给CTO看的报表 ====== ");
        // 注入另一个访问者,CTO
        report.showReport(new CTOVisitor());
    }
}
```

310 页

```
public class ExampleActivity extends Activity {
    TextView title;
    ImageViewicon;
    TextView footer;

     @Override public void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_activity);
        // 通过 findViewById 进行视图查找，然后进行类型转换
        title = (TextView) findViewById(R.id.title);
        icon = (ImageView) findViewById(R.id.icon);
        footer = (TextView) findViewById(R.id.footer);
    }
}
```

在 ExampleActivity 函数的 onCreate 函数中，我们通常会对各个子视图进行初始化，这些代码看起来重复性很高，而且丑陋不堪，当一个布局中含有十个以上的 View 时，再加上为某些 View 添加上事件处理等，这部分的代码将占用挺大的篇幅。ButterKnife 就是为了简化这些工作而出现的，让开发人员专注在真正有用的代码上。使用 ButterKnife 之后我们的代码变成了这样:

```
public class ExampleActivity extends Activity {
    @InjectView(R.id.title) TextView title;
    @InjectView(R.id.icon) ImageView icon;
    @InjectView(R.id.footer) TextView footer;

    @Override public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_activity);
        // 将 Activity 注入 ButterKnife
        ButterKnife.inject(this);
  }
}
```

311 页

首先小民建了一个注解类，代码如下 :

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.CLASS)
public @interface ViewInjector {
    int tvalue();
}
```

312 页

在 ViewInjectorProcessor 类的上面我们看到如下注解 @SupportedAnnotationTypes("org.simple.injector.anno.*")，这个注解表明这个类只支持 org.simple.injector.anno 路径下的注解。在该类的 init 函数中我们注册了一个注解处理器，也就是 ViewInjectHandler 类，该类实现了 AnnotationHandler 接口，该接口的声明如下 ：

```
// 注解处理接口
public  interface AnnotationHandler {
    // 关联ProcessingEnvironment
    void  attachProcessingEnv(ProcessingEnvironment processingEnv);
    // 处理注解，将结果存储到Map中
    Map<String, List<VariableElement>> handleAnnotation(RoundEnvironment env);
}
```

第 20 章 381 页

```
public class ArticleAdapter extends Adapter<ViewHolder> {
    List<Article> mArticles;          // 文章数据
    OnItemClickListener<Article> mClickListener; // 每一项的点击事件

    public ArticleAdapter(List<Article> dataSet) {
        mArticles = dataSet;
    }

    // 绑定每一项的数据
    @Override
    public void onBindViewHolder(ViewHolder viewHolder, int position) {
        if (viewHolder instanceof ArticleViewHolder) {
            bindViewForArticle(viewHolder, position);
        }
    }

    protected void bindViewForArticle(ViewHolder viewHolder, int position) {
        ArticleViewHolder articleViewHolder = (ArticleViewHolder) viewHolder;
        final Article article = getItem(position);
        articleViewHolder.titleTv.setText(article.title);
        articleViewHolder.publishTimeTv.setText(article.publishTime);
        articleViewHolder.authorTv.setText(article.author);
```

```
        // 设置点击事件
        articleViewHolder.itemView.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                if (mClickListener != null) {
                    mClickListener.onClick(article);
                }
            }
        });
    }

    @Override
    public int getItemCount() {
        return mArticles.size();
    }
    // 创建ViewHolder
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup viewGroup, int
viewType) {
        return createArticleViewHolder(viewGroup);
    }

    protected ViewHolder createArticleViewHolder(ViewGroup viewGroup) {
        View itemView = LayoutInflater.from(viewGroup.getContext()).inflate(
                R.layout.recyclerview_article_item, viewGroup, false);
        return new ArticleViewHolder(itemView);
    }

    protected Article getItem(int position) {
        return mArticles.get(position);
    }

    public void setOnItemClickListener(OnItemClickListener<Article> mClickListener){
        this.mClickListener = mClickListener;
    }

    // ViewHolder，负责保持Item View
    static class ArticleViewHolder extends RecyclerView.ViewHolder {
        public TextView titleTv;
        public TextView publishTimeTv;
        public TextView authorTv;

        public ArticleViewHolder(View itemView) {
```

```
            super(itemView);

            titleTv = (TextView) itemView.findViewById(R.id.article_title_tv);
            publishTimeTv = (TextView) itemView.findViewById(R.id.article_time_tv);
            authorTv = (TextView) itemView.findViewById(R.id.article_author_tv);
        }
    }
}
```

389 下到 390

```
int fill(RecyclerView.Recycler recycler, LayoutState layoutState,
            RecyclerView.State state, boolean stopOnFocusable) {
    // 存储当前可用空间
    final int start = layoutState.mAvailable;
     // 1、计算 RecyclerView 的可用布局宽或者高
     int remainingSpace = layoutState.mAvailable + layoutState.mExtra;
     LayoutChunkResult layoutChunkResult = new LayoutChunkResult();
     // 2、迭代布局 item view
     while (remainingSpace > 0 && layoutState.hasMore(state)) {
         layoutChunkResult.resetInternal();
         // 3、布局 item View
        layoutChunk(recycler, state, layoutState, layoutChunkResult);
         if (layoutChunkResult.mFinished) {
             break;
         }
         // 4、计算布局偏移量
         layoutState.mOffset        +=        layoutChunkResult.mConsumed        *
layoutState.mLayoutDirection;

        if (!layoutChunkResult.mIgnoreConsumed || mLayoutState.mScrapList != null
            || !state.isPreLayout()) {
            layoutState.mAvailable -= layoutChunkResult.mConsumed;
            // 5、计算剩余的可用空间
            remainingSpace -= layoutChunkResult.mConsumed;
        }
       // 代码省略
    }
```

```
    return start - layoutState.mAvailable;
}
```

在 onLayoutChilden 函数中会调用 fill 函数，在 fill 函数中又会循环地调用 layoutChunk 函数进行布局，每次布局完之后就会计算当前屏幕剩余的可利用空间，并且做出判断是否还需要布局 Item View。因此，我们先看看 layoutChunk 的实现。

```
    void layoutChunk(RecyclerView.Recycler recycler, RecyclerView.State state,
            LayoutState layoutState, LayoutChunkResult result) {
        // 1、获取 Item View
        View view = layoutState.next(recycler);
        // 2、获取 Item View 的布局参数
        RecyclerView.LayoutParams params = (RecyclerView.LayoutParams)
view.getLayoutParams();
        // 3、丈量 Item View
        measureChildWithMargins(view, 0, 0);
        // 4、计算该 Item View 消耗的宽度或者高度
        result.mConsumed = mOrientationHelper.getDecoratedMeasurement(view);
        // item view 的上下左右坐标位置
        int left, top, right, bottom;
        // 5、按照水平或者竖直方向布局计算 Item View 的上下左右坐标
        if (mOrientation == VERTICAL) {
            if (isLayoutRTL()) {
                right = getWidth() - getPaddingRight();
                left = right -
mOrientationHelper.getDecoratedMeasurementInOther(view);
            } else {
                left = getPaddingLeft();
                right = left +
mOrientationHelper.getDecoratedMeasurementInOther(view);
            }
            if (layoutState.mLayoutDirection == LayoutState.LAYOUT_START) {
                bottom = layoutState.mOffset;
                top = layoutState.mOffset - result.mConsumed;
            } else {
                top = layoutState.mOffset;
                bottom = layoutState.mOffset + result.mConsumed;
            }
        } else {
            // 竖直方向布局的计算方式
        }
        // 6、布局 item view
        layoutDecorated(view, left + params.leftMargin, top + params.topMargin,
                right - params.rightMargin, bottom - params.bottomMargin);
```

```
    }
```

398 页

```
public class MainActivity extends Activity {
    private CircleMenuLayout mCircleMenuLayout;
    // 菜单标题
    private String[] mItemTexts = new String  [] {
            "安全中心 ", "特色服务", "投资理财",
            "转账汇款", "我的账户", "信用卡"
    };
    // 菜单图标
    private int[] mItemImgs = newint[] {
            R.drawable.home_mbank_1_normal,
            R.drawable.home_mbank_2_normal, R.drawable.home_mbank_3_normal,
            R.drawable.home_mbank_4_normal, R.drawable.home_mbank_5_normal,
            R.drawable.home_mbank_6_normal
    };

    @Override
    protected voidonCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 初始化圆形菜单
        mCircleMenuLayout = (CircleMenuLayout) findViewById(R.id.id_menulayout);
        // 设置菜单数据项
        mCircleMenuLayout.setMenuItemIconsAndTexts(mItemImgs, mItemTexts);
        // 设置菜单项点击事件
        mCircleMenuLayout.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onClick(View view, int pos) {
                Toast.makeText(MainActivity.this, mItemTexts[pos],
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

400 页

```
static class MenuItem {
    public int imageId;
    public String title;
    public MenuItem(String title, int resId) {
        this.title = title;
         imageId = resId;
     }
 }
```

401 页

```
public class MainActivity extends Activity {
    private ListView mListView;
    List<MenuItem> mMenuItems = new ArrayList<MenuItem>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 模拟数据
        mockMenuItems();

        mListView = (ListView) findViewById(R.id.id_menulayout);
        // 设置适配器
        mListView.setAdapter(new CircleMenuAdapter(mMenuItems));
        // 设置点击事件
        mListView.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int position,
        long id) {
                Toast.makeText(MainActivity.this, mMenuItems.get(position).title,
                        Toast.LENGTH_SHORT).show();
            }
        });
```

```
}
```

第 22 章 421 页到 423 上

首先我们创建一个 Ticket 接口，该接口定义展示车票信息的函数。代码如下 ：

```
public interface Ticket {
    public void showTicketInfo(String bunk);
}
```

它的一个具体的实现类是 TrainTicket 类，代码如下 ：

```
// 火车票
class TrainTicket implements Ticket {
    public String from; // 始发地
    public String to; // 目的地
    public String bunk; // 铺位
    public int price;

    TrainTicket(String from, String to) {
        this.from = from;
        this.to = to;
    }

    @Override
    public void showTicketInfo(String bunk) {
        price = new Random().nextInt(300);
        System.out.println("购买 从 " + from + " 到 " + to + "的 "
                + bunk + " 火车票" + ", 价格 : " + price);
    }
}
```

每张火车票中有出发地、目的地、铺位、价格等字段，在购票用户每次查询时如果我们没有用某种缓存模式，那么我们返回车票数据的接口简单实现如下 ：

```
    public class TicketFactory {
        Public static Ticket getTicket(String from, String to) {
            return new TrainTicket(from, to);
    }
}
```

在 TicketFactory 的 getTicket 函数中每次会 new 一个 TrainTicket 对象，也就是说如果在短时间内有 10000 万请求北京到青岛的车票，那么北京到青岛的车票对象就会被创建 10000 次，当数据返回之后这些对象变得无用了又会被虚拟机回收。此时就会造成大量的重复对象存在内存中，GC 对这些对象的回收也会非常的耗资源。如果用户量很大的请求下可能导致系统变得极其缓慢，甚至可能导致 OOM。

正如上文所说，享元模式通过消息池的形式有效的减少了重复对象的存在。它通过内部

状态标识某个种类的对象，外部程序根据这个不会变化的内部状态从消息池中取出对象。使得同一类对象可以被复用，避免大量重复对象。

使用享元模式很简单，我们只需要简单的改造一下 TicketFactory，代码如下 ：

```
// 车票工厂,以出发地和目的地为 key 缓存车票
public class TicketFactory {
    static Map<String, Ticket> sTicketMap = new ConcurrentHashMap<String, Ticket>();

    public static Ticket getTicket(String from, String to) {
        String key = from + "-" + to;
        if (sTicketMap.containsKey(key)) {
            System.out.println("使用缓存 ==> " + key);
            return sTicketMap.get(key);
        } else {
            System.out.println("创建对象 ==> " + key);
            Ticket ticket = new TrainTicket(from, to);
            sTicketMap.put(key, ticket);
            return ticket;
        }
    }
}
```

我们在 TicketFactory 添加了一个 map 容器，并且以出发地 +“-”+ 目的地为键、以车票对象作为值存储车票对象。这个 map 的键就是我们说的内部状态，在这里就是出发地、横杠、目的地拼接起来的字符串，如果没有缓存则创建一个对象，并且将这个对象缓存到 map 中，下次再有这类请求时则直接从缓存中获取。这样即使有 10000 个请求北京到青岛的车票，那么出发地是北京、目的地是青岛的车票对象只有一个。这样就从这个对象从 10000 减到了 1 个，避免了大量的内存占用以及频繁的 GC 操作。

简单使用代码如下 ：

```
public class Test {
    public static void main(String[] args) {
        Ticket ticket01 = TicketFactory.getTicket("北京", "青岛");
        ticket01.showTicketInfo("上铺");
        Ticket ticket02 = TicketFactory.getTicket("北京", "青岛");
        ticket02.showTicketInfo("下铺");
        Ticket ticket03 = TicketFactory.getTicket("北京", "青岛");
        ticket03.showTicketInfo("坐票");
    }
}
```

423 页

```
private static void testString() {
        String str1 = new String("abc");
        String str2 = "abc";
        String str3 = new String("abc");
        String str4 = "ab" + "c";
        // 使用equals只判定字符值
        System.out.println(str1.equals(str2));
        System.out.println(str1.equals(str3));
        System.out.println(str3.equals(str2));

        // 等号判等，判定两个对象是不是同一个地址
        System.out.println(str1 == str2);
        System.out.println(str1 == str3);
        System.out.println(str3 == str2);
        System.out.println(str4 == str2);
    }
```

424 页

```
public class MainActvity extends Activity {
    Handler mHandler = new Handler(Looper.getMainLooper()) ;

    private void doSomeThing() {
        new Thread(){
            @Override
            public void run() {
                // 耗时操作,得到结果,但不能在这个线程更新UI

                // 可以通过Handler将结果投递到主线程中,并且更新UI
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        // 在这里可以更新UI
                    }
                });
            } // end of run
        };
```

```
    }
}
```

426 页

```
    public static Message obtain() {
        synchronized (sPoolSync) {
            if (sPool != null) {
                Message m = sPool;
                sPool = m.next;
                m.next = null;
                m.flags = 0; // 清空in-use flag
                sPoolSize--;
                return m;
            }
        }
    return new Message();
}
```

第 23 章 441 到 443 章

```
public class MobilePhone {
    private Phone mPhone = new PhoneImpl();
    private Camera mCamera = new SamsungCamera();

    public void dail() {
        mPhone.dail();
    }

    public void videoChat() {
        System.out.println("--> 视频聊天接通中");
        mCamera.open();
        mPhone.dail();
    }

    public void hangup() {
        mPhone.hangup();
    }
```

```
    public void takePicture() {
        mCamera.open();
        mCamera.takePicture();
    }

    public void closeCamera() {
        mCamera.close();
    }
}
```

MobilePhone 类中含有两个子系统，也就是拨号系统和拍照系统，MobilePhone 将这两个系统封装起来，为用户提供一个统一的操作接口，也就是说用户只需要通过 MobilePhone 这个类就可以操作打电话和拍照这两个功能。用户不需要知道有 Phone 这个接口以及它的实现类是 PhoneImpl，同样也不需要知道 Camera 相关的信息，通过 MobilePhone 就可以包揽一切。而在 MobilePhone 中也封装了两个子系统的交互，例如视频电话时需要先打开摄像头，然后再开始拨号，如果没有这一步的封装，每次用户实现视频通话功能时都需要手动打开摄像头、进行拨号，这些逻辑太过细节，增加了用户的使用成本，外观模式使得这些操作更加简单、易用。

我们来看看 Phone 接口和 PhoneImpl:

```
public interface Phone {
    // 拨电话
    public void dail();
    // 挂断
    public void hangup();
}

class PhoneImpl implements Phone {
    @Override
    public void dail() {
        System.out.println("打电话");
    }

    @Override
    public void hangup() {
        System.out.println("挂断");
    }
}
```

代码很简单，就是单纯的抽象与实现。Camera 也是类似的实现，代码如下 ：

```
public interface Camera {
    public void open();
    public void takePicture();
    public void close();
```

```
}

class SamsungCamera implements Camera {

    @Override
    public void open() {
        System.out.println("打开相机");
    }

    @Override
    public void takePicture() {
        System.out.println("拍照");
    }

    @Override
    public void close() {
        System.out.println("关闭相机");
    }
}
    测试代码 :
    Public class Test {
        public static void main(String[] args) {
            MobilePhone nexus6 = new MobilePhone();
             // 拍照
             nexus6.takePicture();
            // 视频聊天
            nexus6.videoChat();
        }
    }
```

461 页到 463

```
public final class SimpleImageLoader {
    // SimpleImageLoader 实例
    private static SimpleImageLoader sInstance;
    // 网络请求队列
    private RequestQueue mImageQueue;
    // 图片加载配置对象
    private ImageLoaderConfig mConfig;
```

```
    public void init(ImageLoaderConfig config) {
        mConfig = config;
        checkConfig();
        // 启动请求队列
        mImageQueue = new RequestQueue(mConfig.threadCount);
        mImageQueue.start();
    }
    // 加载图片
    public void displayImage(final ImageView imageView, final String uri,
            final DisplayConfig config, final ImageListener listener) {
        BitmapRequest request = new BitmapRequest(imageView, uri, config, listener);
        // 加载的配置对象,如果没有设置则使用 ImageLoader 的配置
        request.displayConfig = request.displayConfig != null ? request.displayConfig
                : mConfig.displayConfig;
        // 设置加载策略
        request.setLoadPolicy(mConfig.loadPolicy);
        // 添加对队列中
        mImageQueue.addRequest(request);
    }
}
```

在 ImageLoader 类中对外暴露的函数基本只有 displayImage，也就是加载图片的函数。最简单的情况下用户只需要传递 ImageView 和图片的 uri 即可实现图片加载。在该函数中，首先我们会将传递进来的参数转换为一个 BitmapRequest 对象，然后将该对象传递到请求队列中。请求队列的初始化就在 ImageLoader 的初始化函数中。该请求队列初始化之后就会启动 CPU 数量 + 1 个的请求处理线程，在这些线程的 run 中不断地从请求队列中获取请求、加载请求、最后将图片投递到 UI 线程更新 ImageView。首先我们来看消息队列的初始化。

```
public final class RequestQueue {
    // 请求队列
    private BlockingQueue<BitmapRequest> mRequestQueue = new
PriorityBlockingQueue<BitmapRequest>();

    // 默认的核心数
    public static int DEFAULT_CORE_NUMS = Runtime.getRuntime().availableProcessors() + 1;
    // NetworkExecutor,执行网络请求的线程
    private RequestDispatcher[] mDispatchers = null;

   // 启动 RequestDispatcher
    private final void startDispatchers() {
        mDispatchers = new RequestDispatcher[mDispatcherNums];
        for (int i = 0; i < mDispatcherNums; i++) {
            Log.e("", "### 启动线程 " + i);
            mDispatchers[i] = new RequestDispatcher(mRequestQueue);
            mDispatchers[i].start();
```

```
        }
    }

    public void start() {
        stop();
        startDispatchers();
    }
    // 代码省略
}
```

在 RequestQueue 中会启动指定数量的 RequestDispatcher 线程，每个 RequestDispatcher 本质上是一个线程，在它们的 run 函数中有一个死循环不断地查询请求队列中是否含有请求，并且处理这些请求。

```
public RequestDispatcher extends Thread {
    @Override
    Public void run() {
        try {
            while (!this.isInterrupted()) {
                // 1、读取请求
                final BitmapRequest request = mRequestQueue.take();
                if (request.isCancel) {
                    continue;
                }
                // 2、获取Schema
                final String schema = parseSchema(request.imageUri);
                // 3、获取对应的图片加载器
                Loader imageLoader = LoaderManager.getInstance().getLoader(schema);
                // 4、加载图片，最后会将结果投递给UI线程并且更新UI
                imageLoader.loadImage(request);
            }
        } catch (InterruptedException e) {
            Log.i("", "### 请求分发器退出");
        }
    }
}
```

490 到 491

```
public interface ArticleViewInterface {
    public void showArticles(List<Article> articles); // 展示数据
    public void showLoading();                          // 显示进度条
```

```
    public void hideLoading();                              // 隐藏进度条
}
```

ArticleModelImpl 则是对数据库的存取操作，用于保存网络上加载的数据以及从数据库中加载文章缓存。为了代码简单我们这里的模拟数据库的存取操作，将数据缓存到内存中。代码如下 ：

```
public class ArticleModelImpl implements ArticleModel {
    List<Article> mCachedArticles = new LinkedList<Article>();

    @Override
    public void saveArticles(List<Article> articles) {
        mCachedArticles.addAll(articles);
    }

    @Override
    public void loadArticlesFromCache(DataListener<List<Article>> listener) {
        listener.onComplete(mCachedArticles);
    }
}
```

493 到 494

```
public abstract class BasePresenter<T> {
    protected Reference<T> mViewRef;      // View 接口类型的弱引用

    public void attachView(T view) {
        mViewRef = new WeakReference<T>(view);// 建立关联
    }

    protected T getView() {
        return mViewRef.get();
    }

    public boolean isViewAttached() {
        return mViewRef != null && mViewRef.get() != null;
    }

    public void detachView() {
        if (mViewRef != null) {
            mViewRef.clear();
            mViewRef = null;
        }
    }
```

```
}
```

BasePresenter 有四个方法，分别为与 View 建立关联、解除关联、判断是否与 View 建立了关联、获取 View。View 类型通过 BasePresenter 的泛型类型传递进来，Presenter 对这个 View 持有弱引用。通常情况下这个 View 类型应该是实现了某个特定接口的 Activity 或者 Fragment 等类型。

创建一个 MVPBaseActivity 基类，通过这个基类的声明周期函数来控制它与 Presenter 的关系，相关代码如下 ：

```
public abstract class MVPBaseActivity<V, Textends BasePresenter<V>> extends
        Activity {
    protected T mPresenter;   // Presenter对象

    @SuppressWarnings("unchecked")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
         mPresenter = createPresenter();       // 创建Presenter
        mPresenter.attachView((V) this);       // View与Presenter建立关联
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mPresenter.detachView();
    }

    protected abstract T createPresenter();
}
```

MVPBaseActivity 含有两个泛型参数，第一个是 View 接口类型，第二个是 Presenter 的具体类型。通过泛型参数，使得一些通用的逻辑可以被抽象到 MVPBaseActivity 类中。例如，在 MVPBaseActivity 的 onCreate 函数中，会通过 createPresenter 函数创建一个具体的 Presenter，这个 Presenter 的类型就是 BasePresenter<T>类型的。构建 Presenter 之后调用 attachView 函数与 Activity 建立关联。而在 onDestory 函数中，则会与 Activity 解除关联，从而避免内存泄漏。有的同学可能会问，如果在 onDestory 中解除了对 Activity 的引用，那么就没有必要再用弱引用了。这个问题的答案是并不是在任何情况下 Activity 的 onDestory 都会被调用，一旦这种情况发生，弱引用也能够保证不会造成内存泄漏。而通过 MVPBaseActivity 的封装维护 Presenter 与 View 关联关系的代码，使得子类可以避免重复的代码，例如 ArticlesActivity 继承自 MVPBaseActivity 的实现如下 ：

```
Ppublic class ArticlesActivity extends
        MVPBaseActivity<ArticleViewInterface, ArticlePresenterV2>
           implements ArticleViewInterface {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initViews();
        RequestQueueMgr.init(getApplicationContext());
        mPresenter.fetchArticles();
    }
    // 代码省略
}
```