摘要

1 前言

为了有利于项目维护、增强代码可读性、提升 Code Review 效率以及规范团队安卓开发，故提出以下安卓开发规范.

2 AS 规范

工欲善其事，必先利其器。

1. 尽量使用最新的稳定版的 IDE 进行开发；
2. 编码格式统一为 **UTF-8**；
3. 编辑完 .java、.xml 等文件后一定要 **格式化，格式化，格式化**（如果团队有公共的样式包，那就遵循它，否则统一使用 AS 默认模板即可）；
4. 删除多余的 import，减少警告出现，可利用 AS 的 Optimize Imports（Settings -> Keymap -> Optimize Imports）快捷键；

3 命名规范

代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。正确的英文拼写和语法可以让阅读者易于理解，避免歧义。

> 注意：即使纯拼音命名方式也要避免采用。但 `alibaba`、`taobao`、`youku`、`hangzhou` 等国际通用的名称，可视同英文。

3.1 包名

包名全部小写，连续的单词只是简单地连接起来，不使用下划线，采用反域名命名规则，全部使用小写字母。一级包名是顶级域名，通常为 `com`、`edu`、`gov`、`net`、`org` 等，二级包名为公司名，三级包名根据应用进行命名，后面就是对包名的划分了，关于包名的划分，推荐采用 PBF（按功能分包 Package By Feature），一开始我们采用的也是 PBL（按层分包 Package By Layer），很坑爹。PBF 可能不是很好区分在哪个功能中，不过也比 PBL 要好找很多，且 PBF 与 PBL 相比较有如下优势：

- package 内高内聚，package 间低耦合

  哪块要添新功能，只改某一个 package 下的东西。

  PBL 降低了代码耦合，但带来了 package 耦合，要添新功能，需要改 model、dbHelper、view、service 等等，需要改动好几个 package 下的代码，改动的地方越多，越容易产生新问题，不是吗？

  PBF 的话 featureA 相关的所有东西都在 featureA 包，feature 内高内聚、高度模块化，不同 feature 之间低耦合，相关的东西都放在一起，还好找。

- package 有私有作用域（package-private scope）

  你负责开发这块功能，这个目录下所有东西都是你的。

  PBL 的方式是把所有工具方法都放在 util 包下，小张开发新功能时候发现需要一个 xxUtil，但它又不是通用的，那应该放在哪里？没办法，按照分层原则，我们还得放在 util 包下，好像不太合适，但放在其它包更不合适，功能越来越多，util 类也越定义越多。后来小李负责开发一块功能时发现需要一个 xxUtil，同样不通用，去 util 包一看，怎么已经有了，而且还没法复用，只好放弃 xx 这个名字，改为 xxxUtil……，因为 PBL 的 package 没有私有作用域，每一个包都是 public（跨包方法调用是很平常的事情，每一个包对其它包来说都是可访问的）；如果是 PBF，小张的 xxUtil 自然放在 featureA 下，小李的 xxUtil 在 featureB 下，如果觉得 util 好像是通用的，就去 util 包看看要不要把工具方法添进 xxUtil, class 命名冲突没有了。

  PBF 的 package 有私有作用域，featureA 不应该访问 featureB 下的任何东西（如果非访问不可，那就说明接口定义有问题）。

- 很容易删除功能

  统计发现新功能没人用，这个版本那块功能得去掉。

  如果是 PBL，得从功能入口到整个业务流程把受到牵连的所有能删的代码和 class 都揪出来删掉，一不小心就完蛋。

  如果是 PBF，好说，先删掉对应包，再删掉功能入口（删掉包后入口肯定报错了），完事。

- 高度抽象

  解决问题的一般方法是从抽象到具体，PBF 包名是对功能模块的抽象，包内的 class 是实现细节，符合从抽象到具体，而 PBL 弄反了。

  PBF 从确定 AppName 开始，根据功能模块划分 package，再考虑每块的具体实现细节，而 PBL 从一开始就要考虑要不要 dao 层，要不要 com 层等等。

- 只通过 class 来分离逻辑代码

  PBL 既分离 class 又分离 package，而 PBF 只通过 class 来分离逻辑代码。

  没有必要通过 package 分离，因为 PBL 中也可能出现尴尬的情况：

  ```
  ├── service ├── MainService.java
  ```

  按照 PBL, service 包下的所有东西都是 Service，应该不需要 Service 后缀，但实际上通常为了方便，直接 import service 包，Service 后缀是为了避免引入的 class 和当前包下的 class 命名冲突，当然，不用后缀也可以，得写清楚包路径，比如 `new com.domain.service.MainService()`，麻烦；而 PBF 就很方便，无需 import，直接 `new MainService()` 即可。

- package 的大小有意义了

  PBL 中包的大小无限增长是合理的，因为功能越添越多，而 PBF 中包太大（包里 class 太多）表示这块需要重构（划分子包）。

如要知道更多好处，可以查看这篇博文：**Package by features, not layers**，当然，我们大谷歌也有相应的 Sample：**todo-mvp**，其结构如下所示，很值得学习。

```
com
  └── example
    └── android
      └── architecture
        └── blueprints
            └── todoapp
                ├── BasePresenter.java
                ├── BaseView.java
                ├── addedittask
                │   ├── AddEditTaskActivity.java
                │   ├── AddEditTaskContract.java
                │   ├── AddEditTaskFragment.java
                │   └── AddEditTaskPresenter.java
                ├── data
                │   ├── Task.java
                │   └── source
                │       ├── TasksDataSource.java
                │       ├── TasksRepository.java
                │       ├── local
                │       │   ├── TasksDbHelper.java
                │       │   ├── TasksLocalDataSource.java
                │       │   └── TasksPersistenceContract.java
                │       └── remote
                │           └── TasksRemoteDataSource.java
                ├── statistics
                │   ├── StatisticsActivity.java
                │   ├── StatisticsContract.java
                │   ├── StatisticsFragment.java
                │   └── StatisticsPresenter.java
                ├── taskdetail
                │   ├── TaskDetailActivity.java
                │   ├── TaskDetailContract.java
                │   ├── TaskDetailFragment.java
                │   └── TaskDetailPresenter.java
                ├── tasks
                │   ├── ScrollChildSwipeRefreshLayout.java
                │   ├── TasksActivity.java
                │   ├── TasksContract.java
                │   ├── TasksFilterType.java
                │   ├── TasksFragment.java
                │   └── TasksPresenter.java
                └── util
                    ├── ActivityUtils.java
                    ├── EspressoIdlingResource.java
                    └── SimpleCountingIdlingResource.java
```

参考以上的代码结构，按功能分包具体可以这样做：

```
com
└── domain
└── app
    ├── App.java 定义 Application 类
    ├── Config.java 定义配置数据（常量）
    ├── base 基础组件
    ├── custom_view 自定义视图
    ├── data 数据处理
    │   ├── DataManager.java 数据管理器，
    │   ├── local 来源于本地的数据，比如 SP，Database，File
    │   ├── model 定义 model（数据结构以及 getter/setter、compareTo、equals 等等，不含复杂操作）
    │   └── remote 来源于远端的数据
    ├── feature 功能
    │   ├── feature0 功能 0
    │   │   ├── feature0Activity.java
    │   │   ├── feature0Fragment.java
    │   │   ├── xxAdapter.java
    │   │   └── ... 其他 class
    │   └── ...其他功能
    ├── injection 依赖注入
    ├── util 工具类
    └── widget 小部件
```

3.2 类名

类名都以 `UpperCamelCase` 风格编写。

类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。现在还没有特定的规则或行之有效的约定来命名注解类型。

名词，采用大驼峰命名法，尽量避免缩写，除非该缩写是众所周知的， 比如 HTML、URL，如果类名称中包含单词缩写，则单词缩写的每个字母均应大写。

类	描述	例如
`Activity` 类	`Activity` 为后缀标识	欢迎页面类 `WelcomeActivity`
`Adapter` 类	`Adapter` 为后缀标识	新闻详情适配器 `NewsDetailAdapter`
解析类	`Parser` 为后缀标识	首页解析类 `HomePosterParser`
工具方法类	`Utils` 或 `Manager` 为后缀标识	线程池管理类：`ThreadPoolManager` 日志工具类： `LogUtils`（`Logger` 也可）打印工具类：`PrinterUtils`
数据库类	以 `DBHelper` 后缀标识	新闻数据库：`NewsDBHelper`
`Service` 类	以 `Service` 为后缀标识	时间服务 `TimeService`
`BroadcastReceiver` 类	以 `Receiver` 为后缀标识	推送接收 `JPushReceiver`
`ContentProvider` 类	以 `Provider` 为后缀标识	`ShareProvider`
自定义的共享基础类	以 `Base` 开头	`BaseActivity`, `BaseFragment`

测试类的命名以它要测试的类的名称开始，以 Test 结束。例如：`HashTest` 或 `HashIntegrationTest`。

接口（interface）：命名规则与类一样采用大驼峰命名法，多以 able 或 ible 结尾，如 `interface Runnable`、`interface Accessible`。

> 注意：如果项目采用 MVP，所有 Model、View、Presenter 的接口都以 I 为前缀，不加后缀，其他的接口采用上述命名规则。

3.3 方法名

方法名都以 `lowerCamelCase` 风格编写。

方法名通常是动词或动词短语。

方法	说明		:----------	----------
`initXX()`	初始化相关方法，使用 init 为前缀标识，如初始化布局 `initView()`
`isXX()`, `checkXX()`	方法返回值为 boolean 型的请使用 is/check 为前缀标识
`getXX()`	返回某个值的方法，使用 get 为前缀标识
`setXX()`	设置某个属性值
`handleXX()`, `processXX()`	对数据进行处理的方法

`displayXX()`, `showXX()`	弹出提示框和提示信息，使用 display/show 为前缀标识
`updateXX()`	更新数据
`saveXX()`, `insertXX()`	保存或插入数据
`resetXX()`	重置数据
`clearXX()`	清除数据
`removeXX()`, `deleteXX()`	移除数据或者视图等，如 `removeView()`
`drawXX()`	绘制数据或效果相关的，使用 draw 前缀标识

3.4 常量名

常量名命名模式为 `CONSTANT_CASE`，全部字母大写，用下划线分隔单词。那到底什么算是一个常量？

每个常量都是一个 `static final` 字段，但不是所有 `static final` 字段都是常量。在决定一个字段是否是一个常量时，得考虑它是否真的感觉像是一个常量。例如，如果观测任何一个该实例的状态是可变的，则它几乎肯定不会是一个常量。只是永远不打算改变的对象一般是不够的，它要真的一直不变才能将它示为常量。

```
// Constants
  static final int NUMBER = 5;
  static final ImmutableListNAMES = ImmutableList.of("Ed", "Ann");
  static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
  static final SomeMutableType[] EMPTY_ARRAY = {};
  enum SomeEnum { ENUM_CONSTANT }

// Not constants
  static String nonFinal = "non-final";
  final String nonStatic = "non-static";
  static final SetmutableCollection = new HashSet();
  static final ImmutableSetmutableElements = ImmutableSet.of(mutable);
  static final Logger logger = Logger.getLogger(MyClass.getName());
  static final String[] nonEmptyArray = {"these", "can", "change"};
```

3.5 非常量字段名

非常量字段名以 `lowerCamelCase` 风格的基础上改造为如下风格：基本结构为 `scope{Type0}VariableName{Type1}`、`type0VariableName{Type1}`、`variableName{Type1}`。

说明：`{}` 中的内容为可选。

> 注意：所有的 VO（值对象）统一采用标准的 lowerCamelCase 风格编写，所有的 DTO（数据传输对象）就按照接口文档中定义的字段名编写。

3.5.1 scope（范围）

非公有，非静态字段命名以 `m` 开头。

静态字段命名以 `s` 开头。

其他字段以小写字母开头。

例如：

```
public class MyClass {
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

使用 1 个字符前缀来表示作用范围，1 个字符的前缀必须小写，前缀后面是由表意性强的一个单词或多个单词组成的名字，而且每个单词的首写字母大写，其它字母小写，这样保证了对变量名能够进行正确的断句。

3.5.2 Type0（控件类型）

考虑到 Android 众多的 UI 控件，为避免控件和普通成员变量混淆以及更好地表达意思，所有用来表示控件的成员变量统一加上控件缩写作为前缀（具体见附录 UI 控件缩写表）。

例如：`mIvAvatar`、`rvBooks`、`flContainer`。

3.5.3 VariableName（变量名）

变量名中可能会出现量词，我们需要创建统一的量词，它们更容易理解，也更容易搜索。

例如：`mFirstBook`、`mPreBook`、`curBook`。

量词列表	量词后缀说明
`First`	一组变量中的第一个
`Last`	一组变量中的最后一个
`Next`	一组变量中的下一个
`Pre`	一组变量中的上一个
`Cur`	一组变量中的当前变量

3.5.4 Type1（数据类型）

对于表示集合或者数组的非常量字段名，我们可以添加后缀来增强字段的可读性，比如：

集合添加如下后缀：List、Map、Set。

数组添加如下后缀：Arr。

例如：`mIvAvatarList`、`userArr`、`firstNameSet`。

> 注意：如果数据类型不确定的话，比如表示的是很多书，那么使用其复数形式来表示也可，例如 `mBooks`。

3.6 参数名

参数名以 `lowerCamelCase` 风格编写，参数应该避免用单个字符命名。

3.7 局部变量名

局部变量名以 `lowerCamelCase` 风格编写，比起其它类型的名称，局部变量名可以有更为宽松的缩写。

虽然缩写更宽松，但还是要避免用单字符进行命名，除了临时变量和循环变量。

即使局部变量是 `final` 和不可改变的，也不应该把它示为常量，自然也不能用常量的规则去命名它。

3.8 临时变量

临时变量通常被取名为 `i`、`j`、`k`、`m` 和 `n`，它们一般用于整型；`c`、`d`、`e`，它们一般用于字符型。 如：`for (int i = 0; i < len; i++)`。

3.9 类型变量名

类型变量可用以下两种风格之一进行命名：

1. 单个的大写字母，后面可以跟一个数字（如：`E`, `T`, `X`, `T2`）。
2. 以类命名方式（参考3.2 类名），后面加个大写的 T（如：`RequestT`, `FooBarT`）。

更多还可参考：**阿里巴巴 Java 开发手册**

4 代码样式规范

4.1 使用标准大括号样式

左大括号不单独占一行，与其前面的代码位于同一行：

```
class MyClass {
    int func() {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
     }
}
```

我们需要在条件语句周围添加大括号。例外情况：如果整个条件语句（条件和主体）适合放在同一行，那么您可以（但不是必须）将其全部放在一行上。例如，我们接受以下样式：

```
if (condition) {
   body();
}
```

同样也接受以下样式：

```
if (condition) body();
```

但不接受以下样式：

```
if (condition)
    body();  // bad!
```

4.2 编写简短方法

在可行的情况下，尽量编写短小精炼的方法。我们了解，有些情况下较长的方法是恰当的，因此对方法的代码长度没有做出硬性限制。如果某个方法的代码超出 40 行，请考虑是否可以在不破坏程序结构的前提下对其拆解。

4.3 类成员的顺序

这并没有唯一的正确解决方案，但如果都使用一致的顺序将会提高代码的可读性，推荐使用如下排序：

1. 常量
2. 字段
3. 构造函数
4. 重写函数和回调
5. 公有函数
6. 私有函数
7. 内部类或接口

例如：

```
public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.getSimpleName();

    private String mTitle;
    private TextView mTextViewTitle;

    @Override
    public void onCreate() {
        ...
    }

    public void setTitle(String title) {
        mTitle = title;
    }

    private void setUpView() {
        ...
    }

    static class AnInnerClass {

    }
}
```

如果类继承于 Android 组件（例如 `Activity` 或 `Fragment`），那么把重写函数按照他们的生命周期进行排序是一个非常好的习惯，例如，`Activity` 实现了 `onCreate()`、`onDestroy()`、`onPause()`、`onResume()`，它的正确排序如下所示：

```
public class MainActivity extends Activity {
    //Order matches Activity lifecycle
    @Override
    public void onCreate() {}

    @Override
    public void onResume() {}

    @Override
    public void onPause() {}

    @Override
    public void onDestroy() {}
}
```

4.4 函数参数的排序

在 Android 开发过程中，`Context` 在函数参数中是再常见不过的了，我们最好把 `Context` 作为其第一个参数。

正相反，我们把回调接口应该作为其最后一个参数。

例如：

```
// Context always goes first
public User loadUser(Context context, int userId);

// Callbacks always go last
public void loadUserAsync(Context context, int userId, UserCallback callback);
```

4.5 字符串常量的命名和值

Android SDK 中的很多类都用到了键值对函数，比如 `SharedPreferences`、`Bundle`、`Intent`，所以，即便是一个小应用，我们最终也不得不编写大量的字符串常量。

当时用到这些类的时候，我们 **必须** 将它们的键定义为 `static final` 字段，并遵循以下指示作为前缀。

类	字段名前缀
SharedPreferences	`PREF_`
Bundle	`BUNDLE_`
Fragment Arguments	`ARGUMENT_`
Intent Extra	`EXTRA_`
Intent Action	`ACTION_`

说明：虽然 `Fragment.getArguments()` 得到的也是 `Bundle` ，但因为这是 `Bundle` 的常用用法，所以特意为此定义一个不同的前缀。

例如：

```
// 注意：字段的值与名称相同以避免重复问题
static final String PREF_EMAIL = "PREF_EMAIL";
static final String BUNDLE_AGE = "BUNDLE_AGE";
static final String ARGUMENT_USER_ID = "ARGUMENT_USER_ID";

// 与意图相关的项使用完整的包名作为值的前缀
static final String EXTRA_SURNAME = "com.myapp.extras.EXTRA_SURNAME";
static final String ACTION_OPEN_USER = "com.myapp.action.ACTION_OPEN_USER";
```

4.6 Activities 和 Fragments 的传参

当 `Activity` 或 `Fragment` 传递数据通过 `Intent` 或 `Bundle` 时，不同值的键须遵循上一条所提及到的。

当 `Activity` 或 `Fragment` 启动需要传递参数时，那么它需要提供一个 `public static` 的函数来帮助启动或创建它。

这方面，AS 已帮你写好了相关的 Live Templates，启动相关 `Activity` 的只需要在其内部输入 `starter` 即可生成它的启动器，如下所示：

```
public static void start(Context context, User user) {
      Intent starter = new Intent(context, MainActivity.class);
      starter.putParcelableExtra(EXTRA_USER, user);
      context.startActivity(starter);
}
```

同理，启动相关 `Fragment` 在其内部输入 `newInstance` 即可，如下所示：

```
public static MainFragment newInstance(User user) {
      Bundle args = new Bundle();
      args.putParcelable(ARGUMENT_USER, user);
      MainFragment fragment = new MainFragment();
      fragment.setArguments(args);
      return fragment;
}
```

> 注意：这些函数需要放在 `onCreate()` 之前的类的顶部；如果我们使用了这种方式，那么 `extras` 和 `arguments` 的键应该是 `private` 的，因为它们不再需要暴露给其他类来使用。

4.7 行长限制

代码中每一行文本的长度都应该不超过 100 个字符。虽然关于此规则存在很多争论，但最终决定仍是以 100 个字符为上限，如果行长超过了 100（AS 窗口右侧的竖线就是设置的行宽末尾 ），我们通常有两种方法来缩减行长。

- 提取一个局部变量或方法（最好）。
- 使用换行符将一行换成多行。

不过存在以下例外情况：

- 如果备注行包含长度超过 100 个字符的示例命令或文字网址，那么为了便于剪切和粘贴，该行可以超过 100 个字符。
- 导入语句行可以超出此限制，因为用户很少会看到它们（这也简化了工具编写流程）。

4.7.1 换行策略

这没有一个准确的解决方案来决定如何换行，通常不同的解决方案都是有效的，但是有一些规则可以应用于常见的情况。

4.7.1.1 操作符的换行

除赋值操作符之外，我们把换行符放在操作符之前，例如：

```
int longName = anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne
        + theFinalOne;
```

赋值操作符的换行我们放在其后，例如：

```
int longName =
        anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne + theFinalOne;
```

4.7.1.2 函数链的换行

当同一行中调用多个函数时（比如使用构建器时），对每个函数的调用应该在新的一行中，我们把换行符插入在 `.` 之前。

例如：

```
Picasso.with(context).load("https://blankj.com/images/avatar.jpg").into(ivAvatar);
```

我们应该使用如下规则：

```
Picasso.with(context)
        .load("https://blankj.com/images/avatar.jpg")
        .into(ivAvatar);
```

4.7.1.3 多参数的换行

当一个方法有很多参数或者参数很长的时候，我们应该在每个 `,` 后面进行换行。

比如：

```
loadPicture(context, "https://blankj.com/images/avatar.jpg", ivAvatar, "Avatar of the user", clickListener);
```

我们应该使用如下规则：

```
loadPicture(context,
        "https://blankj.com/images/avatar.jpg",
        ivAvatar,
        "Avatar of the user",
        clickListener);
```

4.7.1.4 RxJava 链式的换行

RxJava 的每个操作符都需要换新行，并且把换行符插入在 `.` 之前。

例如：

```
public Observable<Location> syncLocations() {
    return mDatabaseHelper.getAllLocations()
            .concatMap(new Func1<Location, Observable<? extends Location>>() {
                @Override
                 public Observable<? extends Location> call(Location location) {
                     return mRetrofitService.getLocation(location.id);
                 }
            })
            .retry(new Func2<Integer, Throwable, Boolean>() {
                 @Override
                 public Boolean call(Integer numRetries, Throwable throwable) {
                     return throwable instanceof RetrofitError;
                 }
            });
}
```

5 资源文件规范

资源文件命名为全部小写，采用下划线命名法。

如果是组件化开发，我们可以在组件和公共模块间创建一个 ui 模块来专门存放资源文件，然后让每个组件都依赖 ui 模块。这样做的好处是如果老项目要实现组件化的话，只需把资源文件都放入 ui 模块即可，如果想对资源文件进行分包，可以参考这篇文章：**Android Studio 下对资源进行分包**；还避免了多个模块间资源不能复用的问题。

如果是三方库开发，其使用到的资源文件及相关的 `name` 都应该使用库名作为前缀，这样做可以避免三方库资源和实际应用资源重名的冲突。

5.1 动画资源文件（anim/ 和 animator/）

安卓主要包含属性动画和视图动画，其视图动画包括补间动画和逐帧动画。属性动画文件需要放在 `res/animator/` 目录下，视图动画文件需放在 `res/anim/` 目录下。

命名规则：`{模块名_}逻辑名称`。

说明：`{}` 中的内容为可选，`逻辑名称` 可由多个单词加下划线组成。

例如：`refresh_progress.xml`、`market_cart_add.xml`、`market_cart_remove.xml`。

如果是普通的补间动画或者属性动画，可采用：`动画类型_方向` 的命名方式。

例如：

名称	说明
`fade_in`	淡入
`fade_out`	淡出
`push_down_in`	从下方推入
`push_down_out`	从下方推出
`push_left`	推向左方
`slide_in_from_top`	从头部滑动进入
`zoom_enter`	变形进入
`slide_in`	滑动进入
`shrink_to_middle`	中间缩小

5.2 颜色资源文件（color/）

专门存放颜色相关的资源文件。

命名规则：`类型{_模块名}_逻辑名称`。

说明：`{}` 中的内容为可选。

例如：`sel_btn_font.xml`。

颜色资源也可以放于 `res/drawable/` 目录，引用时则用 `@drawable` 来引用，但不推荐这么做，最好还是把两者分开。

5.3 图片资源文件（drawable/ 和 mipmap/）

`res/drawable/` 目录下放的是位图文件（.png、.9.png、.jpg、.gif）或编译为可绘制对象资源子类型的 XML 文件，而 `res/mipmap/` 目录下放的是不同密度的启动图标，所以 `res/mipmap/` 只用于存放启动图标，其余图片资源文件都应该放到 `res/drawable/` 目录下。

命名规则：`类型{_模块名}_逻辑名称`、`类型{_模块名}_颜色`。

说明：`{}` 中的内容为可选；`类型` 可以是可绘制对象资源类型，也可以是控件类型（具体见附录UI 控件缩写表）；最后可加后缀 `_small` 表示小图，`_big` 表示大图。

例如：

名称	说明
`btn_main_about.png`	主页关于按键 `类型_模块名_逻辑名称`

`btn_back.png`	返回按键 `类型_逻辑名称`
`divider_maket_white.png`	商城白色分割线 `类型_模块名_颜色`
`ic_edit.png`	编辑图标 `类型_逻辑名称`
`bg_main.png`	主页背景 `类型_逻辑名称`
`btn_red.png`	红色按键 `类型_颜色`
`btn_red_big.png`	红色大按键 `类型_颜色`
`ic_head_small.png`	小头像图标 `类型_逻辑名称`
`bg_input.png`	输入框背景 `类型_逻辑名称`
`divider_white.png`	白色分割线 `类型_颜色`
`bg_main_head.png`	主页头部背景 `类型_模块名_逻辑名称`
`def_search_cell.png`	搜索页面默认单元图片 `类型_模块名_逻辑名称`
`ic_more_help.png`	更多帮助图标 `类型_逻辑名称`
`divider_list_line.png`	列表分割线 `类型_逻辑名称`
`sel_search_ok.xml`	搜索界面确认选择器 `类型_模块名_逻辑名称`
`shape_music_ring.xml`	音乐界面环形形状 `类型_模块名_逻辑名称`

如果有多种形态，如按钮选择器：`sel_btn_xx.xml`，采用如下命名：

名称	说明
`sel_btn_xx`	作用在 `btn_xx` 上的 `selector`
`btn_xx_normal`	默认状态效果
`btn_xx_pressed`	`state_pressed` 点击效果
`btn_xx_focused`	`state_focused` 聚焦效果
`btn_xx_disabled`	`state_enabled` 不可用效果
`btn_xx_checked`	`state_checked` 选中效果
`btn_xx_selected`	`state_selected` 选中效果
`btn_xx_hovered`	`state_hovered` 悬停效果
`btn_xx_checkable`	`state_checkable` 可选效果
`btn_xx_activated`	`state_activated` 激活效果
`btn_xx_window_focused`	`state_window_focused` 窗口聚焦效果

> 注意：使用 Android Studio 的插件 SelectorChapek 可以快速生成 selector，前提是命名要规范。

5.4 布局资源文件（layout/）

命名规则：`类型_模块名`、`类型{_模块名}_逻辑名称`。

说明：`{}` 中的内容为可选。

例如：

名称	说明
`activity_main.xml`	主窗体 `类型_模块名`
`activity_main_head.xml`	主窗体头部 `类型_模块名_逻辑名称`
`fragment_music.xml`	音乐片段 `类型_模块名`
`fragment_music_player.xml`	音乐片段的播放器 `类型_模块名_逻辑名称`
`dialog_loading.xml`	加载对话框 `类型_逻辑名称`
`ppw_info.xml`	信息弹窗（PopupWindow） `类型_逻辑名称`
`item_main_song.xml`	主页歌曲列表项 `类型_模块名_逻辑名称`

5.5 菜单资源文件（menu/）

菜单相关的资源文件应放在该目录下。

命名规则：`{模块名_}逻辑名称`

说明：`{}` 中的内容为可选。

例如：`main_drawer.xml`、`navigation.xml`。

5.6 values 资源文件（ values/ ）

`values/` 资源文件下的文件都以 `s` 结尾，如 `attrs.xml`、`colors.xml`、`dimens.xml`，起作用的不是文件名称，而是 `<resources>` 标签下的各种标签，比如 `<style>` 决定样式，`<color>` 决定颜色，所以，可以把一个大的 `xml` 文件分割成多个小的文件，比如可以有多个 `style` 文件，如 `styles.xml`、`styles_home.xml`、`styles_item_details.xml`、`styles_forms.xml`。

5.6.1 colors.xml

`<color>` 的 `name` 命名使用下划线命名法，在你的 `colors.xml` 文件中应该只是映射颜色的名称一个 ARGB 值，而没有其它的。不要使用它为不同的按钮来定义 ARGB 值。

例如，不要像下面这样做：

```
<resources>
    <color name="button_foreground">#FFFFFF</color>
    <color name="button_background">#2A91BD</color>
    <color name="comment_background_inactive">#5F5F5F</color>
    <color name="comment_background_active">#939393</color>
    <color name="comment_foreground">#FFFFFF</color>
    <color name="comment_foreground_important">#FF9D2F</color>
    ...
    <color name="comment_shadow">#323232</color>
```

使用这种格式，会非常容易重复定义 ARGB 值，而且如果应用要改变基色的话会非常困难。同时，这些定义是跟一些环境关联起来的，如 `button` 或者 `comment`，应该放到一个按钮风格中，而不是在 `colors.xml` 文件中。

相反，应该这样做：

```
<resources>

    <!-- grayscale -->
    <color name="white"     >#FFFFFF</color>
    <color name="gray_light">#DBDBDB</color>
    <color name="gray"      >#939393</color>
    <color name="gray_dark" >#5F5F5F</color>
    <color name="black"     >#323232</color>

    <!-- basic colors -->
    <color name="green">#27D34D</color>
    <color name="blue">#2A91BD</color>
    <color name="orange">#FF9D2F</color>
    <color name="red">#FF432F</color>

</resources>
```

向应用设计者那里要这个调色板，名称不需要跟 `"green"`、`"blue"` 等等相同。`"brand_primary"`、`"brand_secondary"`、`"brand_negative"` 这样的名字也是完全可以接受的。像这样规范的颜色很容易修改或重构，会使应用一共使用了多少种不同的颜色变得非常清晰。通常一个具有审美价值的 UI 来说，减少使用颜色的种类是非常重要的。

> 注意：如果某些颜色和主题有关，那就单独写一个 `colors_theme.xml`。

5.6.2 dimens.xml

像对待 `colors.xml` 一样对待 `dimens.xml` 文件，与定义颜色调色板一样，你同时也应该定义一个空隙间隔和字体大小的“调色板”。 一个好的例子，如下所示：

```
<resources>

    <!-- font sizes -->
    <dimen name="font_22">22sp</dimen>
    <dimen name="font_18">18sp</dimen>
    <dimen name="font_15">15sp</dimen>
    <dimen name="font_12">12sp</dimen>

    <!-- typical spacing between two views -->
    <dimen name="spacing_40">40dp</dimen>
    <dimen name="spacing_24">24dp</dimen>
    <dimen name="spacing_14">14dp</dimen>
    <dimen name="spacing_10">10dp</dimen>
    <dimen name="spacing_4">4dp</dimen>

    <!-- typical sizes of views -->
    <dimen name="button_height_60">60dp</dimen>
    <dimen name="button_height_40">40dp</dimen>
    <dimen name="button_height_32">32dp</dimen>

</resources>
```

布局时在写 `margins` 和 `paddings` 时，你应该使用 `spacing_xx` 尺寸格式来布局，而不是像对待 `string` 字符串一样直接写值，像这样规范的尺寸很容易修改或重构，会使应用所有用到的尺寸一目了然。 这样写会非常有感觉，会使组织和改变风格或布局非常容易。

5.6.3 strings.xml

`<string>` 的 `name` 命名使用下划线命名法，采用以下规则：`{模块名_}逻辑名称`，这样方便同一个界面的所有 `string` 都放到一起，方便查找。

名称	说明
`main_menu_about`	主菜单按键文字
`friend_title`	好友模块标题栏
`friend_dialog_del`	好友删除提示
`login_check_email`	登录验证
`dialog_title`	弹出框标题
`button_ok`	确认键
`loading`	加载文字

5.6.4 styles.xml

`<style>` 的 `name` 命名使用大驼峰命名法，几乎每个项目都需要适当的使用 `styles.xml` 文件，因为对于一个视图来说，有一个重复的外观是很常见的，将所有的外观细节属性（`colors`、`padding`、`font`）放在 `styles.xml` 文件中。 在应用中对于大多数文本内容，最起码你应该有一个通用的 `styles.xml` 文件，例如：

```
<style name="ContentText">
    <item name="android:textSize">@dimen/font_normal</item>
    <item name="android:textColor">@color/basic_black</item>
</style>
```

应用到 `TextView` 中：

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/price"
    style="@style/ContentText"
    />
```

或许你需要为按钮控件做同样的事情，不要停止在那里，将一组相关的和重复 `android:xxxx` 的属性放到一个通用的 `<style>` 中。

5.7 id 命名

命名规则：`view 缩写{_模块名}_逻辑名`，例如： `btn_main_search`、`btn_back`。

如果在项目中有用黄油刀的话，使用 AS 的插件：ButterKnife Zelezny，可以非常方便帮助你生成注解；没用黄油刀的话可以使用 Android Code Generator 插件。

6 版本统一规范

Android 开发存在着众多版本的不同，比如 `compileSdkVersion`、`minSdkVersion`、`targetSdkVersion` 以及项目中依赖第三方库的版本，不同的 module 及不同的开发人员都有不同的版本，所以需要一个统一版本规范的文件。

如果是开发多个系统级别的应用，当多个应用同时用到相同的 `so` 库时，一定要确保 `so` 库的版本一致，否则可能会引发应用崩溃。

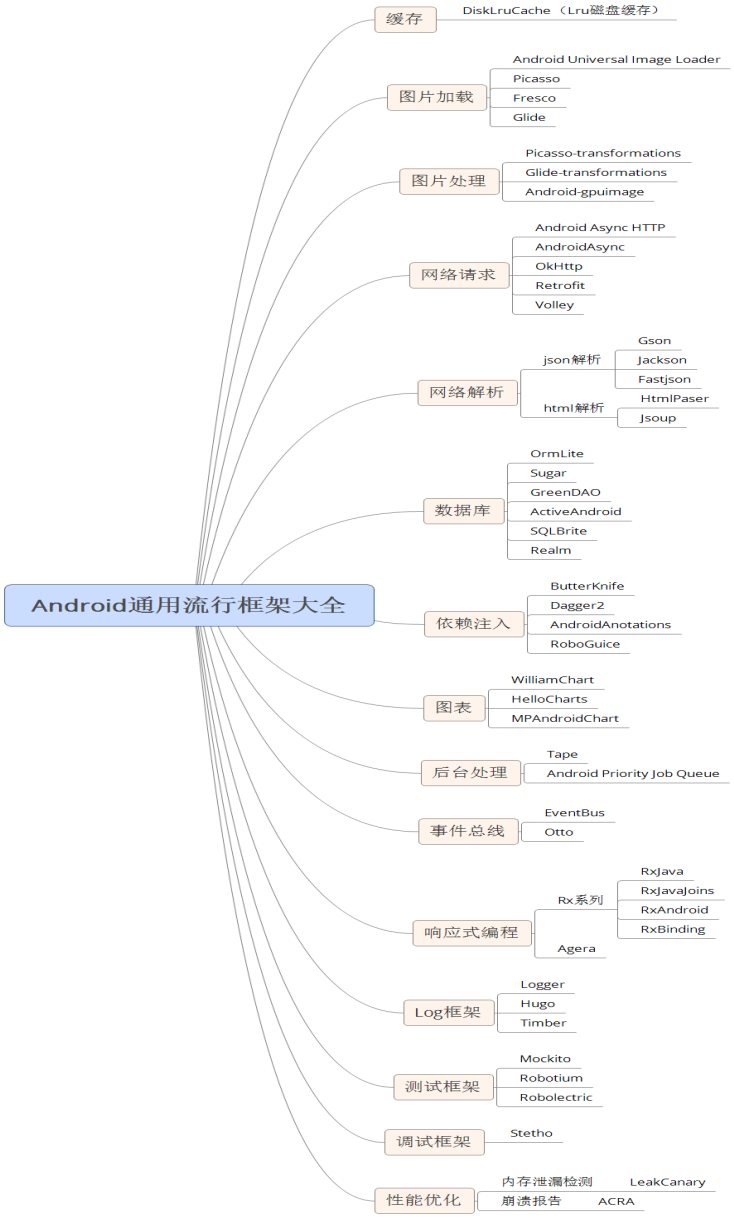
7 第三方库规范

别再闭门造车了，用用最新最火的技术吧，安利一波：**Android 流行框架查速表**，顺便带上自己的干货：**Android 开发人员不得不收集的代码**。

希望 Team 能用时下较新的技术，对开源库的选取，一般都需要选择比较稳定的版本，作者在维护的项目，要考虑作者对 issue 的解决，以及开发者的知名度等各方面。选取之后，一定的封装是必要的。

个人推荐 Team 可使用如下优秀轮子：

- **Retrofit**
- **RxAndroid**
- **OkHttp**
- **Glide**/**Fresco**
- **Gson**/**Fastjson**
- **EventBus**/**AndroidEventBus**
- **GreenDao**
- **Dagger2**（选用）
- **Tinker**（选用）
  附图一张:



8 注释规范

为了减少他人阅读你代码的痛苦值，请在关键地方做好注释。

8.1 类注释

每个类完成后应该有作者姓名和联系方式的注释，对自己的代码负责。

```
/**
 * <pre>
 *     author : Blankj
 *     e-mail : xxx@xx
 *     time   : 2017/03/07
 *     desc   : xxxx 描述
 *     version: 1.0
 * </pre>
 */
public class WelcomeActivity {
    ...
}
```

具体可以在 AS 中自己配制，进入 Settings -> Editor -> File and Code Templates -> Includes -> File Header，输入

```
/**
 * <pre>
 *     author : ${USER}
 *     e-mail : xxx@xx
 *     time   : ${YEAR}/${MONTH}/${DAY}
 *     desc   :
 *     version: 1.0
 * </pre>
 */
```

这样便可在每次新建类的时候自动加上该头注释。

8.2 方法注释

每一个成员方法（包括自定义成员方法、覆盖方法、属性方法）的方法头都必须做方法头注释，在方法前一行输入 `/** + 回车` 或者设置 `Fix doc comment`（Settings -> Keymap -> Fix doc comment）快捷键，AS 便会帮你生成模板，我们只需要补全参数即可，如下所示。

```
/**
 * bitmap 转 byteArr
 *
 * @param bitmap bitmap 对象
 * @param format 格式
 * @return 字节数组
 */
public static byte[] bitmap2Bytes(Bitmap bitmap, CompressFormat format) {
    if (bitmap == null) return null;
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    bitmap.compress(format, 100, baos);
    return baos.toByteArray();
}
```

8.3 块注释

块注释与其周围的代码在同一缩进级别。它们可以是 `/* ... */` 风格，也可以是 `// ...` 风格（**`//` 后最好带一个空格**）。对于多行的 `/* ... */` 注释，后续行必须从 `*` 开始， 并且与前一行的 `*` 对齐。以下示例注释都是 OK 的。

```
/*
 * This is
 * okay.
 */

// And so
// is this.

/* Or you can
 * even do this. */
```

注释不要封闭在由星号或其它字符绘制的框架里。

> Tip：在写多行注释时，如果你希望在必要时能重新换行（即注释像段落风格一样），那么使用 `/* ... */`。

8.4 其他一些注释

AS 已帮你集成了一些注释模板，我们只需要直接使用即可，在代码中输入 `todo`、`fixme` 等这些注释模板，回车后便会出现如下注释。

```
// TODO: 17/3/14 需要实现，但目前还未实现的功能的说明
// FIXME: 17/3/14 需要修正，甚至代码是错误的，不能工作，需要修复的说明
```

9 测试规范

业务开发完成之后，开发人员做单元测试，单元测试完成之后，保证单元测试全部通过，同时单元测试代码覆盖率达到一定程度（这个需要开发和测试约定，理论上越高越好），开发提测。

9.1 单元测试

测试类的名称应该是所测试类的名称加 `Test`，我们创建 `DatabaseHelper` 的测试类，其名应该叫 `DatabaseHelperTest`。

测试函数被 `@Test` 所注解，函数名通常以被测试的方法为前缀，然后跟随是前提条件和预期的结果。

- 模板：`void methodName 前提条件和预期结果()`
- 例子：`void signInWithEmptyEmailFails()`

> 注意：如果函数足够清晰，那么前提条件和预期的结果是可以省略的。

有时一个类可能包含大量的方法，同时需要对每个方法进行几次测试。在这种情况下，建议将测试类分成多个类。例如，如果 `DataManager` 包含很多方法，我们可能要把它分成 `DataManagerSignInTest`、`DataManagerLoadUsersTest` 等等。

9.2 Espresso 测试

每个 Espresso 测试通常是针对 `Activity`，所以其测试名就是其被测的 `Activity` 的名称加 `Test`，比如 `SignInActivityTest`。

10 其他的一些规范

1. 合理布局，有效运用 `<merge>`、`<ViewStub>`、`<include>` 标签；
2. `Activity` 和 `Fragment` 里面有许多重复的操作以及操作步骤，所以我们都需要提供一个 `BaseActivity` 和 `BaseFragment`，让所有的 `Activity` 和 `Fragment` 都继承这个基类。
3. 方法基本上都按照调用的先后顺序在各自区块中排列；
4. 相关功能作为小区块放在一起（或者封装掉）；
5. 当一个类有多个构造函数，或是多个同名函数，这些函数应该按顺序出现在一起，中间不要放进其它函数；
6. 数据提供统一的入口。无论是在 MVP、MVC 还是 MVVM 中，提供一个统一的数据入口，都可以让代码变得更加易于维护。比如可使用一个 `DataManager`，把 `http`、`preference`、`eventpost`、`database` 都放在 `DataManager` 里面进行操作，我们只需要与 `DataManager` 打交道；
7. 多用组合，少用继承；
8. 提取方法，去除重复代码。对于必要的工具类抽取也很重要，这在以后的项目中是可以重用的。
9. 可引入 `Dagger2` 减少模块之间的耦合性。`Dagger2` 是一个依赖注入框架，使用代码自动生成创建依赖关系需要的代码。减少很多模板化的代码，更易于测试，降低耦合，创建可复用可互换的模块；
10. 项目引入 `RxAndroid` 响应式编程，可以极大的减少逻辑代码；
11. 通过引入事件总线，如：`EventBus`、`AndroidEventBus`、`RxBus`，它允许我们在 `DataLayer` 中发送事件，以便 `ViewLayer` 中的多个组件都能够订阅到这些事件，减少回调；
12. 尽可能使用局部变量；
13. 及时关闭流；
14. 尽量减少对变量的重复计算；

    如下面的操作：

    ```
    for (int i = 0; i < list.size(); i++) {
          ...
    }
    ```

    建议替换为：

    ```
    for (int i = 0, len = list.size(); i < len; i++) {
          ...
    }
    ```

15. 尽量采用懒加载的策略，即在需要的时候才创建；

例如：

```
String str = "aaa";
if (i == 1) {
      list.add(str);
}
```

建议替换为：

```
if (i == 1) {
      String str = "aaa";
      list.add(str);
}
```

16. 不要在循环中使用 `try…catch…`，应该把其放在最外层；
17. 使用带缓冲的输入输出流进行 IO 操作；
18. 尽量使用 `HashMap`、`ArrayList`、`StringBuilder`，除非线程安全需要，否则不推荐使用 `HashTable`、`Vector`、`StringBuffer`，后三者由于使用同步机制而导致了性能开销；
19. 尽量在合适的场合使用单例；

    使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

    1. 控制资源的使用，通过线程同步来控制资源的并发访问。
    2. 控制实例的产生，以达到节约资源的目的。
    3. 控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信。
20. 把一个基本数据类型转为字符串，`基本数据类型.toString()` 是最快的方式，`String.valueOf(数据)` 次之，`数据 + ""` 最慢；
21. 使用 AS 自带的 Lint 来优化代码结构（什么，你不会？右键 module、目录或者文件，选择 Analyze -> Inspect Code）；
22. 最后不要忘了内存泄漏的检测；

最后啰嗦几句：

- 好的命名规则能够提高代码质量，使得新人加入项目的时候降低理解代码的难度；
- 规矩终究是死的，适合团队的才是最好的；
- 命名规范需要团队一起齐心协力来维护执行，在团队生活里，谁都不可能独善其身；
- 一开始可能会有些不习惯，持之以恒，总会成功的。

附录

UI 控件缩写表

名称	缩写
Button	btn
CheckBox	cb
EditText	et
FrameLayout	fl
GridView	gv
ImageButton	ib
ImageView	iv
LinearLayout	ll
ListView	lv
ProgressBar	pb
RadioButtion	rb

RecyclerView	rv
RelativeLayout	rl
ScrollView	sv
SeekBar	sb
Spinner	spn
TextView	tv
ToggleButton	tb
VideoView	vv
WebView	wv

常见的英文单词缩写表

名称	缩写
average	avg
background	bg（主要用于布局和子布局的背景）
buffer	buf
control	ctrl
current	cur
default	def
delete	del
document	doc
error	err
escape	esc
icon	ic（主要用在 App 的图标）
increment	inc
information	info
initial	init
image	img
Internationalization	I18N
length	len
library	lib
message	msg
password	pwd
position	pos
previous	pre
selector	sel（主要用于某一 view 多种状态，不仅包括 ListView 中的 selector，还包括按钮的 selector）
server	srv
string	str
temporary	tmp
window	win

程序中使用单词缩写原则：不要用缩写，除非该缩写是约定俗成的。

参考

Android 包命名规范

Android 开发最佳实践

Android 编码规范

阿里巴巴 Java 开发手册

Project and code style guidelines

Google Java 编程风格指南

小细节，大用途，35 个 Java 代码性能优化总结！