# 开发规范

## 编码规范

1. 局部变量的命名、全局成员变量的命名
   1. 单词首字母除第一个外，都为大写，其他字母都为小写：pluginContext
   2. 不易清楚识别出该变量类型的变量使用意义描述+类型描述的组合，例如：animationThread、responseStr、backCommand、dogSoundPlayer
2. 常量的命名

包含字母和\_，字母全部大写，单词之间用\_隔开：MIN\_CAPACITY\_INCREMENT

1. 资源文件的命名

命名模式为：（产品名称）\_模块名称\_逻辑名称：kuso\_history\_item.xml，kuso\_strings.xml

1. Activity中view的命名

命名模式为：逻辑名称+view全称或缩写：

1. JAR包的命名

命名模式为：公司名称-模块名称-逻辑名称-版本号

1. 包名的命名
   1. 一律小写，少用缩写和长名
   2. 采用以下规则：基本包.模块名.逻辑名
2. 类和接口的命名
   1. 每个单词首字母大写，除了通用缩写词，一律使用完整单词，尽量简洁而富于描述
   2. Interface命名首字母加大写的I
   3. 根据功能类型结尾加上功能描述字符串
      1. Activity类、Fragment类、Service类、Adapter类
      2. 页面类：Page
      3. 处理类：Handle
      4. 接口实现类：Impl
      5. 动作事件定义类：Action
      6. 网络事件定义类：Net
      7. 数据定义类：Data、Info
      8. 消息处理类：Msg
      9. 资源管理类：Manager
      10. 缓存类：Cache
      11. 参数类：Param
      12. 工具类：Util
      13. 输入输出类：Stream
      14. 代理类：Proxy
      15. 兼容类：Compat
3. 方法命名
   1. 方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写；取值类使用get前缀，设值类使用set前缀，判断类使用is(has)前缀
   2. 初始化方法，命名以init开头，例如：initView
   3. 通过异步加载数据的方法，命名以load开头，例如：loadData
4. 注释
   1. 是程序维护的灵魂，是最好的文档
   2. 已经不推荐使用的类和方法需要注明@Deprecated，并说明替代的类或者方法
   3. 所有的源文件都应该在开头有一个注释，其中列出文件的版权声明、文件名、功能描述
   4. 方法注释：对于含义不是很明显的方法，接口中定义的方法，工具类的方法，采用JavaDoc文档注释，在方法定义之前当对其进行注释，包括方法的描述、输入、输出及返回值说明、抛出异常说明、参考链接等
   5. 方法内注释：比较难理解或关键代码要重点注释，单行使用“//”，多行使用“/\*...\*/”
   6. 代码修改注释：修改bug时使用EasyComment注释插件进行代码注释，写明修改原因和方法
   7. 变量注释：常量、重要变量、数据类的属性变量都要在右侧添加“//”进行注释
5. 编码方式统一用UTF-8
6. 代码格式化：

## 最佳实践

* 1. 不要重复造轮子，如果需要就对已有的类、方法进行完善或扩展，不要创建多个表示同一数据或配置的变量
  2. 控制好变量和方法的作用域，合理使用private、protected、public
  3. 使用layer-list和selector
  4. 对大型图片进行缩放，设置恰当的inSampleSize
  5. 图片尽量分拆成多个可重用的图片
  6. 大图尽量拆分成小图的组合，避免空像素区域造成的内存浪费
  7. 服务端可以实现的，就不要放在客户端
  8. 引用第三方库要慎重，避免引用大容量的第三方库，导致客户端包非常大
  9. 尽量使用.9图片
  10. 使用静态变量要慎重，避免内存泄露或浪费，避免四处引用
  11. Activity中在一个OnClickListener中处理所有的点击事件
  12. 如果多个Activity中包含共同的UI处理，那么可以提炼一个CommonActivity，把通用部分交由它来处理，其它Activity只要继承它即可
  13. 数据一定要效验  
      例如：字符型转数字型，如果转换失败一定要有缺省值；服务端响应数据是否有效要判断；
  14. layout组件化，尽量使用merge及include复用
  15. 自适应屏幕，文字大小的单位统一用sp，元素大小的单位统一用dp，特殊情况除外
  16. 使用animation-list制作动画效果
  17. 方法中一定要加上适当的非空判断，与try catch 语句等程序健壮性的判断
  18. 类型强转前进行类型判断
  19. 字符串统一在strings.xml中定义
  20. 颜色值统一在colors.xml中定义，然后在代码和布局文件中引用，不要在代码和布局文件中引用系统的颜色，除了透明
  21. 注意对隐式Intent的运行时检查保护，预防这个问题的最佳解决方案是在发出这个隐式Intent之前调用resolveActivity做检查
  22. 从Android 4.1开始，Notification可以支持展开显示的模式，这样一来，Notification就演变出了4种不同的风格样式，要选取合适的样式
  23. 不要自己实现HTTP客户端或图片加载缓存机制，要使用Volley、OkHttp、Glide
  24. 尽量在一切可以使用应用ApplicationContext代替Context的地方进行替换
  25. 在使用后台Service时尽量将能够替换为IntentService的地方替换为此，这样可以减轻系统压力、省电、省内存、省CPU占用率
  26. 在Android中实现UI界面默认应该选择Fragments
  27. 使用styles，因为它是让view具有相同外观的常见的方法，定义通用的文本、窗口style等
  28. 使用Lint工具来帮你找到可能的优化点
  29. 使用ViewStub标签，根据需要来加载布局
  30. 使用android:tint属性来改变图片颜色
  31. 避免ANR，不要在UI线程中做耗时操作（网络、数据库、文件、图片等）
  32. 慎用synchronized，尽量减小synchronize的方法，实现同步是要很大的系统开销作为代价的，甚至可能造成死锁
  33. 尽量使用StringBuilder和StringBuffer进行字符串连接
  34. 单线程应尽量使用HashMap、ArrayList
  35. 尽量合理的创建HashMap，指定合适的初始大小
  36. 尽量使用移位来代替’a/b’或’a\*b’的操作
  37. 尽量避免使用二维数组
  38. 尽量避免使用split
  39. 尽量在合适的场合使用单例，单例主要适用于以下三个方面：

1. 控制资源的使用
2. 控制实例的产生
3. 控制数据的共享
   1. 包结构：
4. models包目录存放网络请求API响应经过JSON解析器填充后得到的POJOs对象
5. views包目录存放自定义的views，notifications，action bar views，widgets等等
6. adapters介于data和views间，可以在views包中增加adapters子包
7. managers包存放应用范围的或和系统紧密关联的管理类
8. fragment和activity放在单独的包中
9. utils包放工具处理类
10. network放服务器交互相应类

**性能相关：**

1. 弄清楚当前系统的性能，并进行竞品分析，否则无法衡量你进行尝试所得到的性能提升，然后确定存在的问题，再进行优化
2. 不要执行不必要的代码，不要分配不必要的内存
3. 避免创建短命的临时对象
4. 如果不需要访问某对象的字段，将方法设置为静态，调用会加速15%到20%
5. 访问成员变量比访问本地变量慢得多
6. 永远不要在for的第二个条件中调用任何方法
7. 如果要多次访问一个变量，最好先为它建立一个本地变量，方法的参数与本地变量的效率相同
8. 对常量使用static final修饰符
9. 避免使用浮点数，用乘法代替除法
10. 在需要网络的应用中，执行某些操作前尽量先进行网络状态判断
11. 在网络应用传输中使用高效率的数据格式和解析方法，譬如JSON等
12. 在传输用户反馈或者下载OTA升级包等不是十分紧急的操作时尽量采用压缩数据进行传输且延迟到设备充电和WIFI状态时进行
13. 在有必要的情况下尽量通过PowerManager.WakeLock和JobScheduler来控制一些逻辑操作达到省电优化
14. 对定位要求不太高的场景尽量使用网络定位，而不是GPS定位
15. 对于定时任务尽量使用AlarmManager，而不是sleep或者Timer进行管理
16. 了解并使用类库：String.indexOf()，String.lastIndexOf()，System.arraycopy，Formatter，TextUtils，MemoryFile
17. 合理利用native方法，Java和native之间过渡的关联是有消耗的
18. 复杂算法尽量用C或者C++完成，然后用JNI调用
19. 将不用的对象指向NULL，同时，显示让系统gc回收，例如图片处理时
20. 使用实体类比接口好，HashMap好于Map
21. 布局用Java完成比XML快
22. 访问server端时，建立连接本身比传输需要更多的时间，如非必要，不要将一次交互可以做的事情分成多次交互
23. http用gzip压缩，设置连接超时时间和响应超时时间，连接失败时要有重连机制
24. 相对ContentProvider，使用原始SQL语句执行效率高，比如使用方法rawQuery、execSQL的执行效率比较高
25. 对于需要一次性修改多个数据时，可以考虑使用SQLite的事务方式批量处理
26. 批量插入多行数据使用InsertHelper或者bulkInsert方法
27. 能用文件操作的，尽量采用文件操作，文件操作的速度比数据库的操作要快10倍左右
28. 在DDMS中查看Logcat的VM释放内存情况，大概知道哪里可以缓存数据或改进算法
29. ListView性能优化（推荐使用RecyclerView）
    1. 复用convertView
    2. 异步加载图片并缓存
    3. 快速滑动时不显示图片
30. 使用线程池，分为核心线程池和普通线程池，下载图片等耗时任务放置在普通线程池，避免耗时任务阻塞线程池后，导致所有异步任务都必须等待
31. 尽量避免static成员变量引用资源耗费过多的实例,比如Context
32. 使用WeakReference代替强引用，弱引用可以让您保持对对象的引用，同时允许GC在必要时释放对象，回收内存。对于那些创建便宜但耗费大量内存的对象，即希望保持该对象，又要在应用程序需要时使用，同时希望GC必要时回收，可以考虑使用弱引用
33. Bitmap及时的销毁(Activity的onDestroy时，将bitmap回收)，设置一定的采样率，巧妙的运用软引用，drawable对应resid的资源，bitmap对应其他资源
34. 对SQLiteOpenHelper，SQLiteDatabase，Cursor，文件，I/O操作等都应该记得显示关闭
35. 线程也是造成内存泄露的一个重要的源头。线程产生内存泄露的主要原因在于线程生命周期的不可控（Activity结束时，Activity生命周期内创建的线程并不会随之结束，两者没有关联）
36. 使用Jackson，解析JSON性能更好，但是jar包更大
37. 避免过深或过于复杂的views层级，尽量扁平，复杂布局使用RelativeLayout
38. 小心WebView相关的问题，特别是内存泄露，当WebView持有所在Activity Context引用而不是Application Context引用时，可能导致内存泄漏，webview不再使用时及时销毁
39. 避免使用layout\_weight属性
40. Android提供了一种机制，允许通过xml来绘制简单的图形。因此对于简单的图形图片，如果可能就用xml来绘制它，绘制出来的图形是基于矢量的
41. 可以通过一些工具来定位问题，比如可以使用HierarchyViewer来查找Activity中的布局是否过于复杂，也可以使用手机设置里面的开发者选项，打开Show GPU Overdraw等选项进行观察，必须尽量减少Overdraw的情况发生。还可以使用TraceView来观察CPU的执行情况，更加快捷的找到性能瓶颈；流畅的画面需要60fps，这意味着每一帧你只有16ms=1000/60的时间来处理所有的任务
42. 使用adb shell dumpsys查看各种数据
43. 使用traces.txt文件进行ANR分析优化（/data/anr/traces.txt）
44. 对于有缓存等存在的应用尽量实现onLowMemory()和onTrimMemory()方法
45. HashMap与ArrayMap及SparseArray的选择
46. Android系统中执行垃圾回收（GC）操作时所有线程（包含UI线程）都必须暂停，等垃圾回收操作完成之后其他线程才能继续运行。这些GC垃圾回收一般都会有明显的log打印出回收类型，常见的如下：
    1. GC\_MALLOC——内存分配失败时触发；
    2. GC\_CONCURRENT——当分配的对象大小超过一个限定值（不同系统）时触发；
    3. GC\_EXPLICIT——对垃圾收集的显式调用(System.gc()) ；
    4. GC\_EXTERNAL\_ALLOC——外部内存分配失败时触发；