Android性能优化

合理管理内存

节制的使用Service

如果应用程序需要使用Service来执行后台任务的话，只有当任务正在执行的时候才应该让Service运行起来。当启动一个Service时，系统会倾向于将这个Service所依赖的进程进行保留，系统可以在LRUcache当中缓存的进程数量也会减少，导致切换程序的时候耗费更多性能。我们可以使用IntentService，当后台任务执行结束后会自动停止，避免了Service的内存泄漏。

当界面不可见时释放内存

当用户打开了另外一个程序，我们的程序界面已经不可见的时候，我们应当将所有和界面相关的资源进行释放。重写Activity的onTrimMemory()方法，然后在这个方法中监听TRIM_MEMORY_UI_HIDDEN这个级别，一旦触发说明用户离开了程序，此时就可以进行资源释放操作了。

当内存紧张时释放内存

onTrimMemory()方法还有很多种其他类型的回调，可以在手机内存降低的时候及时通知我们，我们应该根据回调中传入的级别来去决定如何释放应用程序的资源。

避免在Bitmap上浪费内存

读取一个Bitmap图片的时候，千万不要去加载不需要的分辨率。可以压缩图片等操作。

是有优化过的数据集合

Android提供了一系列优化过后的数据集合工具类，如SparseArray、SparseBooleanArray、LongSparseArray，使用这些API可以让我们的程序更加高效。HashMap工具类会相对比较低效，因为它需要为每一个键值对都提供一个对象入口，而SparseArray就避免掉了基本数据类型转换成对象数据类型的时间。

知晓内存的开支情况

- 使用枚举通常会比使用静态常量消耗两倍以上的内存，尽可能不使用枚举
- 任何一个Java类，包括匿名类、内部类，都要占用大概500字节的内存空间
- 任何一个类的实例要消耗12-16字节的内存开支，因此频繁创建实例也是会在一定程序上影响内存的
- 使用HashMap时，即使你只设置了一个基本数据类型的键，比如说int，但是也会按照对象的大小来分配内存，大概是32字节，而不是4字节，因此最好使用优化后的数据集合

谨慎使用抽象编程

在Android使用抽象编程会带来额外的内存开支，因为抽象的编程方法需要编写额外的代码，虽然这些代码根本执行不到，但是也要映射到内存中，不仅占用了更多的内存，在执行效率上也会有所降低。所以需要合理的使用抽象编程。

尽量避免使用依赖注入框架

使用依赖注入框架貌似看上去把findViewById()这一类的繁琐操作去掉了，但是这些框架为了要搜寻代码中的注解，通常都需要经历较长的初始化过程，并且将一些你用不到的对象也一并加载到内存中。这些用不到的对象会一直站用着内存空间，可能很久之后才会得到释放，所以可能多敲几行代码是更好的选择。

使用多个进程

谨慎使用，多数应用程序不该在多个进程中运行的，一旦使用不当，它甚至会增加额外的内存而不是帮我们节省内存。这个技巧比较适用于哪些需要在后台去完成一项独立的任务，和前台是完全可以区分开的场景。比如音乐播放，关闭软件，已经完全由Service来控制音乐播放了，系统仍然会将许多UI方面的内存进行保留。在这种场景下就非常适合使用两个进程，一个用于UI展示，另一个用于在后台持续的播放音乐。关于实现多进程，只需要在Manifast文件的应用程序组件声明一个android:process属性就可以了。进程名可以

自定义，但是之前要加个冒号，表示该进程是一个当前应用程序的私有进程。

分析内存的使用情况

系统不可能将所有的内存都分配给我们的应用程序，每个程序都会有可使用的内存上限，被称为堆大小。不同的手机堆大小不同，如下代码可以获得堆大小：

```
ActivityManager manager = (ActivityManager)getSystemService(Context.ACTIVITY_SERVICE);
int heapSize = manager.getMemoryClass();
```

结果以MB为单位进行返回，我们开发时应用程序的内存不能超过这个限制，否则会出现OOM。

Android的GC操作

Android系统会在适当的时机触发GC操作，一旦进行GC操作，就会将一些不再使用的对象进行回收。GC操作会从一个叫做Roots的对象开始检查，所有它可以访问到的对象就说明还在使用当中，应该进行保留，而其他的对系那个就表示已经不再被使用了。

Android中内存泄漏

Android中的垃圾回收机制并不能防止内存泄漏的出现导致内存泄漏最主要的原因就是某些长存对象持有了一些其它应该被回收的对象的引用，导致垃圾回收器无法去回收掉这些对象，也就是出现内存泄漏了。比如说像Activity这样的系统组件，它又会包含很多的控件甚至是图片，如果它无法被垃圾回收器回收掉的话，那就算是比较严重的内存泄漏情况了。 举个例子，在MainActivity中定义一个内部类，实例化内部类对象，在内部类新建一个线程执行死循环，会导致内部类资源无法释放，MainActivity的控件和资源无法释放，导致OOM,可借助一系列工具，比如LeakCanary。

高性能编码优化

都是一些微优化，在性能方面看不出有什么显著的提升的。使用合适的算法和数据结构是优化程序性能的最主要手段。

避免创建不必要的对象

不必要的对象我们应该避免创建：

- 如果有需要拼接的字符串，那么可以优先考虑使用StringBuffer或者StringBuilder来进行拼接，而不是加号连接符，因为使用加号连接符会创建多余的对象，拼接的字符串越长，加号连接符的性能越低。
- 在没有特殊原因的情况下，尽量使用基本数据类型来代替封装数据类型，int比Integer要更加有效，其它数据类型也是一样。
- 当一个方法的返回值是String的时候，通常需要去判断一下这个String的作用是什么，如果明确知道调用方会将返回的String再进行拼接操作的话，可以考虑返回一个StringBuffer对象来代替，因为这样可以将一个对象的引用进行返回，而返回String的话就是创建了一个短生命周期的临时对象。
- 基本数据类型的数组也要优于对象数据类型的数组。另外两个平行的数组要比一个封装好的对象数组更加高效，举个例子，Foo[]和Bar[]这样的数组，使用起来要比Custom(Foo,Bar)[]这样的一个数组高效的多。

尽可能地少创建临时对象，越少的对象意味着越少的GC操作。

静态优于抽象

如果你并不需要访问一个对系那个中的某些字段，只是想调用它的某些方法来去完成一项通用的功能，那么可以将这个方法设置成静态方法，调用速度提升15%-20%，同时也不用为了调用这个方法去专门创建对象了，也不用担心调用这个方法后是否会改变对象的状态(静态方法无法访问非静态字段)。

对常量使用static final修饰符

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

编译器会为上面的代码生成一个初始方法，称为方法，该方法会在定义类第一次被使用的时候调用。这个方法会将42的值赋值到intVal当中，从字符串常量表中提取一个引用赋值到strVal上。当赋值完成后，我们就可以通过字段搜寻的方式去访问具体的值了。

final进行优化:

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

这样，定义类就不需要方法了，因为所有的常量都会在dex文件的初始化器当中进行初始化。当我们调用intVal时可以直接指向42的值，而调用strVal会用一种相对轻量级的字符串常量方式，而不是字段搜寻的方式。

这种优化方式只对基本数据类型以及String类型的常量有效，对于其他数据类型的常量是无效的。

使用增强型for循环语法

```
static class Counter {
    int mCount;
}

Counter[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mCount;
    }
}

public void one() {
    int sum = 0;
    Counter[] localArray = mArray;
    int len = localArray.length;
    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mCount;
    }
}

public void two() {
    int sum = 0;
    for (Counter a : mArray) {
        sum += a.mCount;
    }
}
```

zero()最慢，每次都要计算mArray的长度，one()相对快得多，two()fangfa在没有JIT(Just In Time Compiler)的设备上是运行最快的，而在有JIT的设备上运行效率和one()方法不相上下，需要注意这种写法需要JDK1.5之后才支持。

Tips:ArrayList手写的循环比增强型for循环更快，其他的集合没有这种情况。因此默认情况下使用增强型for循环，而遍历ArrayList使用传统的循环方式。

多使用系统封装好的API

系统提供不了的Api完成不了我们需要的功能才应该自己去写，因为使用系统的Api很多时候比我们自己写的代码要快得多，它们的很多功能都是通过底层的汇编模式执行的。 举个例子，实现数组拷贝的功能，使用循环的方式来对数组中的每一个元素一一进行赋值当然可行，但是直接使用系统中提供的System.arraycopy()方法会让执行效率快9倍以上。

避免在内部调用Getters/Setters方法

面向对象中封装的思想是不要把类内部的字段暴露给外部，而是提供特定的方法来允许外部操作相应类的内部字段。但在Android中，字段搜寻比方法调用效率高得多，我们直接访问某个字段可能要比通过getters方法来去访问这个字段快3到7倍。但是编写代码还是要按照面向对象思维的，我们应该在能优化的地方进行优化，比如避免在内部调用getters/setters方法。

布局优化技巧

重用布局文件

标签可以允许在一个布局当中引入另一个布局，那么比如说我们程序的所有界面都有一个公共的部分，这个时候最好的做法就是将这个公共的部分提取到一个独立的布局中，然后每个界面的布局文件当中来引用这个公共的布局。

Tips:如果我们要在标签中覆写layout属性，必须要将layout_width和layout_height这两个属性也进行覆写，否则覆写xiaoguo将不会生效。

标签是作为标签的一种辅助扩展来使用的，它的主要作用是为了防止在引用布局文件时引用文件时产生多余的布局嵌套。布局嵌套越多，解析起来就越耗时，性能就越差。因此编写布局文件时应该让嵌套的层数越少越好。

举例：比如在LinearLayout里边使用一个布局。里边又有一个LinearLayout，那么其实就存在了多余的布局嵌套，使用merge可以解决这个问题。

仅在需要时才加载布局

某个布局当中的元素不是一起显示出来的，普通情况下只显示部分常用的元素，而那些不常用的元素只有在用户进行特定操作时才会显示出来。

举例：填信息时不是需要全部填的，有一个添加更多字段的选项，当用户需要添加其他信息的时候，才将另外的元素显示到界面上。用VISIBLE性能表现一般，可以用ViewStub。ViewStub也是View的一种，但是没有大小，没有绘制功能，也不参与布局，资源消耗非常低，可以认为完全不影响性能。

```
<ViewStub
        android:id="@+id/view_stub"
        android:layout="@layout/profile_extra"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />
```

```
public void onMoreClick() {
    ViewStub viewStub = (ViewStub) findViewById(R.id.view_stub);
    if (viewStub != null) {
        View inflatedView = viewStub.inflate();
        editExtra1 = (EditText) inflatedView.findViewById(R.id.edit_extra1);
        editExtra2 = (EditText) inflatedView.findViewById(R.id.edit_extra2);
        editExtra3 = (EditText) inflatedView.findViewById(R.id.edit_extra3);
    }
}
```

tips：ViewStub所加载的布局是不可以使用标签的，因此这有可能导致加载出来出来的布局存在着多余的嵌套结构。