



1008711

2017年04月21日 阅读 901

Android性能优化——代码优化（一）

注：根据Android官方的建议，编写高效代码的三个基本准则如下：

- 不要做冗余的工作
- 尽量避免次数过多的内存分配操作
- 深入理解所有语言特性和系统平台的API，具体到Android开发，熟练掌握java语言，并对SDK的API熟悉，了如指掌。

一、数据结构的选择

正确的选择合适的数据结构很重要，对java中常见的数据结构例如ArrayList和LinkedList、HashMap和HashSet等，需要做到对它们的联系与区别有教深入的理解，这样在编写代码中面临选择时才能作出正确的选择，下面我们以android开发中使用SparseArray代替HashMap为例进行说明。

SparseArray是Android平台特有的稀疏数组的实现，它是Integer到Object的一个映射，在特定场合可用于代替HashMap<Integer,<E>>，提高性能。核心实现是二分法查找算法。

SparseArray家族目前有以下四类：

```
HashMap<Integer, Boolean> booleanHashMap = new HashMap<>();
SparseBooleanArray booleanArray = new SparseBooleanArray();
```

```
HashMap<Integer,Integer> integerHashMap = new HashMap<>();  
SparseIntArray intArray = new SparseIntArray();  
  
HashMap<Integer,Long> longHashMap = new HashMap<>();  
SparseLongArray sparseLongArray = new SparseLongArray();  
  
HashMap<Integer,String> stringHashMap = new HashMap<>();  
SparseArray<String> sparseArray = new SparseArray<>();
```

需要注意的几点如下：

- SparseArray不是线程安全。
- 由于要进行二分法查找，因此，SparseArray会对插入的数据按照Key值大小顺序插入。
- SparseArray对删除操作做了优化，它并不会立即删除这个元素，而是通过设置标识位（DELETED）的方式，后面尝试重用。

在Android工程中运行Lint进行静态代码块分析，会有一个名为AndroidLintUseSparseArrays的检查项，如果违规，它会提示：

HashMap can be replaced with SparseArray

这样可以很轻松地找到工程中优化的地方。

二、Handler和内部类的正确用法

Android代码中涉及线程间通信的地方经常会使用Handler，典型的代码结构如下：

```
public class HandlerActivity extends Activity {  
  
    private final Handler mLeakyHandler = new Handler() {  
        @Override  
        public void handleMessage(Message msg) {  
            super.handleMessage(msg);  
        }  
    };  
  
}
```

使用AndroidLint分析这段代码，会违反检查项AndroidLintHandlerLeak,得到如下提示：

This Handler class should be static or leaks might occur

那么产生内存泄漏的原因可能是什么？Handler和Looper以及MessageQueue一起工作的，在Android中，一个应用启动后，系统默认会创建一个为主线程服务的Looper对象，该Looper对象用于处理主线程的所有Message对象，它的生命周期贯穿于整个应用的生命周期。在主线程中使用的Handler都会默认绑定到这个Looper对象。在主线程中创建Handler对象时，它会立即关联主线程Looper对象的MessageQueue，这时发送到MessageQueue中的Message对象都会持有这个Handler对象的引用，这样Looper处理消息时才能回调到Handler的handlerMessage方法。因此，如果Message还没有被处理完成，那么Handler对象也就不会被垃圾回收。

在上面的代码中，将Handler的实例声明为HandlerActivity类的内部类。而在Java语言中，非静态内部匿名类会持有外部类的一个隐式的引用，这样就可能会导致外部类无法被垃圾回收。因此，最终由于MessageQueue中的Message还没有处理完成，就会持有Handler对象的引用，而非静态的Handler对象会持有外部类HandlerActivity的引用，这样Activity无法被垃圾回收，从而导致内存泄漏。

一个明显的会引入内存泄漏的例子如下：

```
/**
 * =====
 * User:xijiufu
 * Email:xjfsml@163.com
 * Version:1.0
 * Time:2017/4/20--2:07
 * Function:
 * ModifyHistory:
 * =====
 */
public class HandlerActivity extends Activity {

    private final Handler mLeakyHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //延迟10分钟发送消息

        mLeakyHandler.postAtTime(new Runnable() {
            @Override
            public void run() {
                /**/
            }
        }, 1000 * 60 * 10);

    }
}
```

由于消息延迟10分钟发送，因此，当用户进入这个Activity并退出后，在消息发送并处理完成之前，这个Activity是会被系统回收（系统内存确实不够使用的情况例外）

如何解决呢？两个方案：

- 在子线程中使用Handler，这是需要开发者自己创建一个Looper对象，这个Looper对象的生命周期同一般的Java对象，因此这种用法没有问题。

- 将Handler声明为静态的内部类，前面说过，静态内部类不会持有外部类的引用，因此，也不会引用内存泄漏，经典用法的代码如下。
-

```
/**
 * =====
 * User:xijiufu
 * Email:xjfsml@163.com
 * Version:1.0
 * Time:2017/4/20--2:07
 * Function:
 * ModifyHistory:
 * =====
 */
public class HandlerActivity extends Activity {

    /**
     * 声明一个静态的Handler内部类，并持有外部类的弱引用
     */
    private static class InnerHandler extends Handler {

        private final WeakReference<HandlerActivity> mActivity;

        public InnerHandler(HandlerActivity activity) {
            this.mActivity = new WeakReference<HandlerActivity>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            HandlerActivity activity = mActivity.get();
            if (activity != null) {
                //..
            }
        }
    }

    private final InnerHandler mHandler = new InnerHandler(this);

    /**
     * 静态的匿名内部类不会持有外部类的引用
     */
    private static final Runnable sRunnable = new Runnable() {
        @Override
        public void run() {
```

```
        /***/  
    }  
};  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    //延迟10分钟发送消息  
  
    mHandler.postAtTime(sRunnable, 1000 * 60 * 10);  
}  
  
}
```

三、正确地使用Context

Context应该是每个入门Android开发的程序员第一个接触到的概念，它代表当前的上下文环境，可以用来实现很多功能的调用，语句如下：

```
//获取资源管理器对象，进而可以访问到例如string，color等资源  
Resources resources = context.getResources();  
  
//启动指定的Activity  
context.startActivity(new Intent(this, MainActivity.class));  
  
//获取各种系统服务  
TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService(Context.TELEPHONY_S  
  
//获取系统文件目录  
File internalDir = context.getCacheDir();  
File externalDir = context.getExternalCacheDir();  
  
//更多。。。
```

可见，正确理解Context的概念是很重要的，虽然应用开发中随处可见Context的使用，但并不是所有

的Context实例都具备相同的功能，在使用上需要区别对待，否则很可能会引入问题。我们首先来总结下Context的种类。

(1)、Context的种类

根据Context依托的组件以及用途不同，我们可以将Context分为如下几种。

- Application：Android应用中的默认单例类，在Activity或者Service中通过getApplication（）可以获取到这个实例，通过context.getApplicationContext() 可以获取到应用全局唯一的Context实例。
- Activity/Service：这两个类都是ContextWrapper的子类，在这两个类中可以通过getBaseContext（）获取到它们的Context实例，不同的Activity或者Service实例，它们的Context都是独立的，不会复用。
- BroadcastReceiver：和Activity以及Service不用，BroadcastReceiver本身并不是Context的子类，而是在回调函数onReceive（）中由Android框架传入一个Context的实例。系统传入的这个Context实例是经过功能裁剪的，它并不能调用registerReceiver（）以及bindService（）这两个函数。
- ContextProvider：同样的，ContentProvider也不是Context的子类，但在创建时系统会传入一个Context实例，这样在ContentProvider中可以通过调用getContext（）函数获取。如果ContentProvider和调用者处于相同的应用进程中，那么getContext（）将返回应用全局唯一的Context的实例。如果是其他进程调用的ContentProvider，那么ContentProvider将持有自身所在进程的Context实例。

(2)、错误使用Context导致的内存泄漏

错误地使用Context可能会导致内存泄漏，典型的例子是在实现单例模式时使用Context，如下代码是可能会导致内存泄漏的单例实现。

```
/**
 * =====
 * User:xijiufu
 * Email:xjfsml@163.com
```

```

* Version:1.0
* Time:2017/4/20--23:57
* Function:
* ModifyHistory:
* =====
*/
public class SingleInstance {

    private Context mContext;
    private static SingleInstance sInstance;

    private SingleInstance(Context context) {
        mContext = context;
    }

    public static SingleInstance getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new SingleInstance(context);
        }
        return sInstance;
    }

}

```

如果使用者调用getInstance时传入的Context是一个Activity或者Service的实例，那么在应用退出之前，由于单例一直存在，会导致对应的Activity或者Service被单例引用，从而不会被垃圾回收，Activity或者Service中关联的其他View或者数据结构对象也不会被释放，从而导致内存泄漏。正确的做法是使用Application Context，因为它是应用唯一的，而且声明周期是跟着应用一致的，正确的单例实现如下：

```

/**
* =====
* User:xijiufu
* Email:xjfsml@163.com
* Version:1.0
* Time:2017/4/20--23:57
* Function:
* ModifyHistory:
* =====

```



```
*/
public class SingleInstance {

    private Context mContext;
    private static SingleInstance sInstance;

    private SingleInstance(Context context) {
        mContext = context;
    }

    public static SingleInstance getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new SingleInstance(context.getApplicationContext()); //这一句关键
        }
        return sInstance;
    }

}
```

(3)、不同Context的对比

不同组件中的Context能提供的功能不尽相同，总结起来，如下表：

功能	Application	Activity	Service	BroadcastReceiver
显示Dialog	NO	YES	NO	NO
启动Activity	NO[1]	YES	NO[1]	NO[1]
实现LayoutInflation	NO[2]	YES	NO[2]	NO[2]
启动Service	YES	YES	YES	YES
绑定Service	YES	YES	YES	YES
发送Broadcast	YES	YES	YES	YES
注册Broadcast	YES	YES	YES	YES
加载资源Resource	YES	YES	YES	YES

其中NO[1]标记表示对应的组件并不是真的不可以启动Activity，而是建议不要这么做，因为这些组件会在新的Task中创建Activity，而不是在原来的Task中。

NO[2]标记也是表示不建议这么做，因为在非Activity中进行Layout Inflation，会使用系统默认的主题，而不是应用中设置的主题。

NO[3]标记表示在Android4.2及以上的系统上，如果注册的BroadcastReceiver是null时是可以的，用来获取sticky广播的当前值。

四、掌握java的四种引用方式

掌握java的四种引用类型对于写出内存使用良好的应用是很关键的。

- 强引用：Java里面最广泛使用的一种，也是对象默认的引用类型。如果一个对象具有强引用，那么垃圾回收期是不会对它进行回收操作的，当内存空间不足时，Java虚拟机将会抛出 OutOfMemoryError 错误，这时应用将会终止运行。一句话总结，只要引用存在，垃圾回收器永远

不会回收。Object obj = new Object(); 可以直接通过obj取得对应的对象，只有obj这个引用被释放之后，对象才会被释放掉。

- 软引用：一个对象如果只有软引用，那么当内存空间不足时，垃圾回收器不会对它进行回收操作，只有当内存空间不足时，这个对象才会被回收。软引用可以用来实现内存敏感的高速缓存，如果配合引用队列（ReferenceQueue）使用，当软引用指向的对象被垃圾回收器回收后，Java虚拟机将会把这个软引用加入到与之关联的引用队列中。一句话总结，非必须引用，内存溢出之前进行回收，

```
Object object = new Object();  
SoftReference<Object> sf = new SoftReference<Object>(object);
```

- 软引用：弱引用是比软引用更弱的一种引用类型，只有弱引用指向的对象的生命周期更短，当垃圾回收器扫描到只具有弱引用的对象时，不论当前内存空间是否不足，都会对弱引用对象进行回收。弱引用也可以和一个引用队列配合使用，当弱引用指向的对象被回收后，Java虚拟机会将这个弱引用加入到与之关联的引用队列中。一句话总结，

```
Object object = new Object();  
WeakReference<Object> reference = new WeakReference<Object>(object);
```

- 虚引用：和软引用和弱引用不同，虚引用并不会对所指向的对象生命周期产生任何影响，也就是对象还是会按照它原来的方式被垃圾回收器回收，虚引用本质上只是一个标记作用，主要用来跟踪对象被垃圾回收的活动，虚引用必须和引用队列配合使用，当对象被垃圾回收时，如果存在虚引用，那么Java虚拟机会将这个虚引用加入与之关联的引用队列中。

五、其他代码微优化

(1)、避免创建非必要的对象

对象的创建需要内存分配，对象的销毁需要垃圾回收，这些都会一定程度上影响到应用的性能。因此一般来水，最好是重用对象而不是在每次需要的时候去创建一个功能相同的新对象，特别是注意不要在循环中重复创建相同的对象。

(2)、对常量使用static final 修饰

对于基本数据类型和String类型的常量，建议使用static final 修饰，因为final类型的常量会在会进入静态dex文件的域初始化部分，这是对基本数据类型和String类型常量的调用不会涉及类的初始化，而是直接调用字面量。

(3)、避免内部的Getters/Setters

在面向对象编程中，Getters/Setters的作用主要是对外屏蔽具体的变量定义，从而达到更好的封装性。但如果是在类内部还使用Getters/Setters函数访问变量的话，会降低访问的速度。根据Android官方文档，在没有JIT（Just In Time）编译器时，直接访问变量的速度是调用Getter方法的3倍；在JIT编译时，直接访问变量的速度是调用Getters方法的7倍。当然，如果你的应用中使用了ProGuard（混淆代码）的话，那么ProGuard会对Getters/Setters进行内联操作，从而达到直接访问的效果。

(4)、代码的重构

代码的重构是一项长期的持之以恒的工作，需要依靠团队中每一个成员来维护代码库的高质量，要会去享受高质量代码带来的快感，如何有效的进行代码重构，除了需要你所在项目有较深入的理解之外，你还需要一定的方法指导。重构代码可以使用不同的设计模式来达到高质量的代码，这儿可以关注我的设计模式系列博客：[Android设计模式之——单例模式（一）](#) [Android设计模式之——Builder模式（二）](#)

关注下面的标签，发现更多相似文章

Android

安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

评论

输入评论...

相关推荐

专栏 HongJay · 11小时前 · Android

孔乙己的疑问：单例模式有几种写法

 11  4

专栏 Yuloran · 10小时前 · Android

Android LowMemoryKiller 简介

 9 

专栏 hankinghu · 12小时前 · Android

自己动手实现Android中的三级缓存框架

 10 

专栏 尚妆产品技术刊读 · 40分钟前 · Android / 微信

SocialSdk-快速接入登录分享

 1 

专栏 hankinghu · 3天前 · Android


Android自定义View之实现简单炫酷的球体进度球


 14  12

热 · 张风捷特烈 · 2天前 · Android

专栏

2018年终总结（兼个人详历）

 188

 47

[专栏](#)腾讯云加社区 · 3天前 · Android

自己动手写事件总线(EventBus)

 10



[专栏](#)Yuloran · 2天前 · Android

Android 平台的垃圾回收机制

 4



[专栏](#)jsonchao · 3天前 · Android

我的2018年终总结（菜鸟的进阶之路）

 36

 5

[专栏](#)Yuloran · 1天前 · Android

Android 内存泄露详解

 13

