都是 Push 惹的祸

阅读 1376 收藏 60 2016-12-08
原文链接：solart.cc

作者：imilk

> 版权声明：本文图文为博主原创，转载请注明出处。

这是一篇 KPI 考核背景下产出的文章，这一切都起源于我司要求提升 App 推送送达率，以节省在短信推广上花费的开销。这里记录了在整个技术调研的关键点。

1、概述

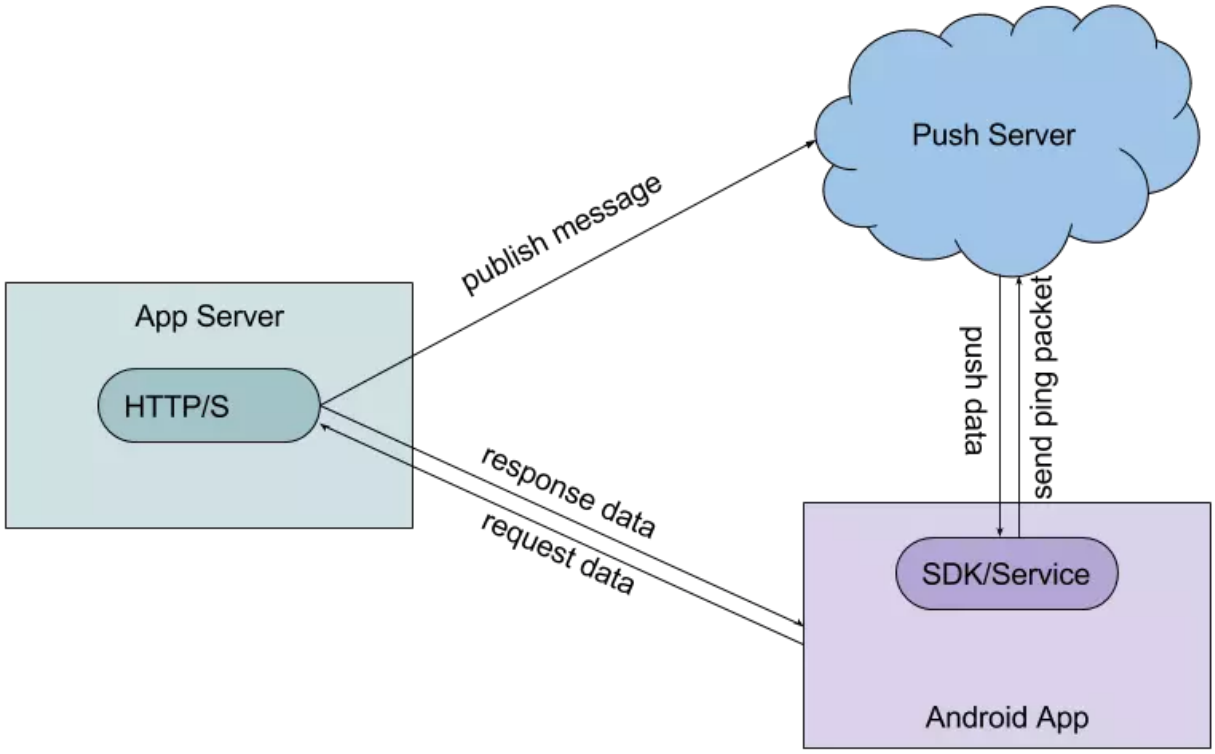
iOS 和 Android 均在系统级集成了推送服务，来说说原生 Android 的推送服务，最在 Android 2.2 时， C2DM 作为系统级服务集成进了 Android 系统，而 GCM（Google Clould Messaging）在 2013 Google IO 大会发布后就正式取代了 C2DM ，然后 Google 并没有止步，在 2014 年收购了 Firebase ，经过近两年的整合，在 2016 年 Google IO 大会上隆重发布了 Firebase 服务，一个全新的移动和 Web 开发的完整后端解决方案，其中就包括了 FCM（Firebase Cloud Messaging） 。如果就这么简单，我们就可以在 Android 平台上像 iOS 平台一样使用系统级共享的推送服务了，然而一股神秘的东方力量打破了原本简单的事情...

众所周知 Google 退出中国市场后，各家厂商在 Android 底层的定制修改，使得国内 Android

以上说了这么多，并不能解决我们的问题，既然存在于这个大环境之中，我们就只能想些办法去适应它。

2、推送及保活

下面就简单先说说推送服务的实现机制，简单点，看图说话：



这就是一个非常简单的推送模型，服务器向 Push 服务器发布推送消息，推送服务器经过处理按照要求将推送消息通过长链接通道将消息推送至 App 。

2.1 皮之不存，毛将焉附

我们知道 Push 服务器想要成功的将消息推送至客户端真正的关键就是这个长链接的稳定性，各家服务也都对这个长链接做了很多优化，比如通道共享、透传等。对于 `Android` 来说通常的做法是通过 `Service` 维持一个长链接，定时发送心跳包以保证实时在线

那么我们首先解决进程存活的问题，在这之前，简单的说明一下进程及资源的冲突问题，大家应该知道，每个进程的启动都意味着消耗掉一部分手机资源，比如 CPU ，内存等。而像内存这样的资源在手机中其实是相当宝贵的，系统为了保证自身运行的稳定性以及前台 App 的性能，在必要时会触发资源回收的机制，在 Android 中这种机制被称之为 `LowMemoryKiller` 。

2.2 进程杀手 LMK

`LowMemoryKiller` 是一种根据 `OOM_ADJ` 阈值级别触发相应力度的内存回收的机制。什么是 `OOM_ADJ` 呢？ OOM 想必大家都知道就不在解释了， `ADJ` 何解？ `Adjustment` ，调整，即内存溢出调整的阈值。 `LMK` 回收内存时会根据进程的级别优先杀死 `OOM_ADJ` 比较大的进程，对于优先级相同的进程则进一步受到进程所占内存和进程存活时间的影响。关于这部分内容大家可以参看 Android LowMemoryKiller 原理分析这篇文章，今天并不主要分析源码，我们开看一个关于 OOM_ADJ 的对应表（参照源码 `com.android.server.am.ProcessList` 中定义的取值整理）。

CACHED_APP_MAX_ADJ	15	不可见进程的adj最大值
CACHED_APP_MIN_ADJ	9	不可见进程的adj最小值
SERVICE_B_ADJ	8	B List中的Service
PREVIOUS_APP_ADJ	7	上一个App的进程
HOME_APP_ADJ	6	Home进程
SERVICE_ADJ	5	服务进程
HEAVY_WEIGHT_APP_ADJ	4	后台的重量级进程
BACKUP_APP_ADJ	3	备份进程
PERCEPTIBLE_APP_ADJ	2	可感知进程，比如后台音乐播放
VISIBLE_APP_ADJ	1	可见进程
FOREGROUND_APP_ADJ	0	前台进程
PERSISTENT_SERVICE_ADJ	-11	关联着系统或presistent进程
PERSISTENT_PROC_ADJ	-12	系统presistent进程
SYSTEM_ADJ	-16	系统进程
NATIVE_ADJ	-17	Native进程（不被系统管理）

其中红色部分代表比较容易被杀死的 Android 进程（ OOM_ADJ >= 4 ）,绿色部分表示不容易被杀死的 Android 进程，其他表示非 Android 进程（纯 Linux 进程）。

2.3 进程保活

根据以上的信息，我们就要想方设法降低我们应用的 OOM_ADJ 的值，那么怎么降低呢？这时候就依赖翻看一些源码了，在 `ActivityManagerService` 中有一个 `updateOomAdjLocked` 函数，在许多地方被调用去更新应用的 OOM_ADJ 值，在这个函数的调用链中我们找到一个 `computeOomAdjLocked` 函数用于计算 App 的 OOM_ADJ 的值，以下我们截取部分代码做简单分析：

```
public final class ActivityManagerService extends ActivityManagerNative
        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback{
```

```
// Determine the importance of the process, starting with most
// important to least, and assign an appropriate OOM adjustment.
int adj;
int schedGroup;
int procState;
boolean foregroundActivities = false;
BroadcastQueue queue;
if (app == TOP_APP) {
    // The last app on the list is the foreground app.
    // 列表中的最后一个应用程序是前台应用程序,这取决于用户的操作，我们无法干预。
    adj = ProcessList.FOREGROUND_APP_ADJ;
    schedGroup = Process.THREAD_GROUP_DEFAULT;
    app.adjType = "top-activity";
    foregroundActivities = true;
    procState = ActivityManager.PROCESS_STATE_TOP;
} else if (app.instrumentationClass != null) {
    // Don't want to kill running instrumentation.
    // 这个case可以尝试，可将 app adj 值降低至 0
    adj = ProcessList.FOREGROUND_APP_ADJ;
    schedGroup = Process.THREAD_GROUP_DEFAULT;
    app.adjType = "instrumentation";
    procState = ActivityManager.PROCESS_STATE_IMPORTANT_FOREGROUND;
} else if ((queue = isReceivingBroadcast(app)) != null) {
    // An app that is currently receiving a broadcast also
    // counts as being in the foreground for OOM killer purposes.
    // It's placed in a sched group based on the nature of the
    // broadcast as reflected by which queue it's active in.
    // 这个 case 要求 app 正在或将要接收广播,可以尝试但操作难度较高
    adj = ProcessList.FOREGROUND_APP_ADJ;
    schedGroup = (queue == mFgBroadcastQueue)
            ? Process.THREAD_GROUP_DEFAULT : Process.THREAD_GROUP_BG_NONI
    app.adjType = "broadcast";
    procState = ActivityManager.PROCESS_STATE_RECEIVER;
} else if (app.executingServices.size() > 0) {
    // An app that is currently executing a service callback also
```

```
            app.adjType = "exec-service";
            procState = ActivityManager.PROCESS_STATE_SERVICE;
            //Slog.i(TAG, "EXEC " + (app.execServicesFg ? "FG" : "BG") + ": " + a
        } else {
            // As far as we know the process is empty.  We may change our mind la
            schedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
            // At this point we don't actually know the adjustment.  Use the cach
            // value that the caller wants us to.
            adj = cachedAdj;
            procState = ActivityManager.PROCESS_STATE_CACHED_EMPTY;
            app.cached = true;
            app.empty = true;
            app.adjType = "cch-empty";
        }

        // Examine all activities if not already foreground.
        if (!foregroundActivities && activitiesSize > 0) {
            for (int j = 0; j < activitiesSize; j++) {
                final ActivityRecord r = app.activities.get(j);
                if (r.app != app) {
                    Slog.w(TAG, "Wtf, activity " + r + " in proc activity list no
                            + app + "?!?");
                    continue;
                }
                if (r.visible) {
                    // App has a visible activity; only upgrade adjustment.
                    // 维护一个用户不可见的 Activity 可以将应用的 adj 降低至 1
                    if (adj > ProcessList.VISIBLE_APP_ADJ) {
                        adj = ProcessList.VISIBLE_APP_ADJ;
                        app.adjType = "visible";
                    }
                    if (procState > ActivityManager.PROCESS_STATE_TOP) {
                        procState = ActivityManager.PROCESS_STATE_TOP;
                    }
                    schedGroup = Process.THREAD_GROUP_DEFAULT;
```

```
                }
                ...
            }
        }

        if (adj > ProcessList.PERCEPTIBLE_APP_ADJ) {
            // 维护一个前台 Service 可将 app adj值降低至 2.
            if (app.foregroundServices) {
                // The user is aware of this app, so make it visible.
                adj = ProcessList.PERCEPTIBLE_APP_ADJ;
                procState = ActivityManager.PROCESS_STATE_IMPORTANT_FOREGROUND;
                app.cached = false;
                app.adjType = "fg-service";
                schedGroup = Process.THREAD_GROUP_DEFAULT;
            } else if (app.forcingToForeground != null) {
                // The user is aware of this app, so make it visible.
                adj = ProcessList.PERCEPTIBLE_APP_ADJ;
                procState = ActivityManager.PROCESS_STATE_IMPORTANT_FOREGROUND;
                app.cached = false;
                app.adjType = "force-fg";
                app.adjSource = app.forcingToForeground;
                schedGroup = Process.THREAD_GROUP_DEFAULT;
            }
        }
        ...
    }
}
```

经过我们的分析，我们可以从以下几个方面尝试：

- 尝试在 Manifest 中注册一个 Instrumentation
- 尝试维护一个用户无法感知的 Activity
- 尝试维持一个前台 Service

其他还可以考虑动机式的 Service 及不停的收发广播这两种变态的方案，在实践中考虑到对性能

我们启动 App 通过 `ps` 命令查看进程对应的 `pid`：

```
➜  ~ ps | grep solart
```

输出为：

```
u0_a250 3855 162 931684 43932 ffffffff 00000000 S cc.solart.nuts
```

我们再通过 `cat` 命令查看 oom_adj 的值：

```
➜ ~ cat /proc/3855/oom_adj
```

输出为：

```
0
```

通过按 Home 键，在查看 oom_adj 的值，输出为：

```
7
```

通过按 Back 键盘退出页面，oom_adj 的值输出为

```
9
```

大家可以看到我们什么都没做的时候，App 进入后台和页面退出后 OOM_ADJ 的值其实已经非常高
如果此时系统内存紧张，则有比较大的概率会被干掉。

尝试维护一个用户无法感知的 Activity：这个方案确实有效的降低了adj的值，但在实际测试过程中发现，在某些机器上因为广播延迟的原因 Activity 无法及时销毁，导致亮屏幕后有一瞬间用户是无法操作的，这个方案被 Pass。

尝试维持一个前台 Service： 这是一个行之有效的方案，在实际过程中发现发送空 Notification 在某些机型上会发生异常，后又改为有内容的通知，但这会在通知栏上闪一下，但通过测试发现确实能够有效让 App 的 adj 值维持在 2 的水平。

```
public class KeepAliveService extends Service {

    static final int NOTIFICATION_ID = 1001;

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {

        try{
            Log.d("nuts", "KeepAliveService onStartCommand invoke");
            startForeground(NOTIFICATION_ID, getNotification(this));

            Intent innerIntent = new Intent(this, InnerService.class);
            startService(innerIntent);

        }catch (Exception e){
            //nothing to do
        }
        // 利用系统自启
        return START_STICKY;
    }
```

```
        public IBinder onBind(Intent intent) {
            return null;
        }

        @Override
        public int onStartCommand(Intent intent, int flags, int startId) {

            try{
                Log.d("nuts", "InnerService onStartCommand invoke");
                startForeground(NOTIFICATION_ID, getNotification(this));
                stopSelf();
            }catch (Exception e){
                //nothing to do
            }
            return super.onStartCommand(intent, flags, startId);
        }
    }

    private static Notification getNotification(Context ctx){
        Notification notify = new NotificationCompat.Builder(ctx)
                .setSmallIcon(R.drawable.ic_launcher)
                .setContentTitle("")
                .setContentText("")
                .setAutoCancel(true)
                .build();
        return notify;
    }
}
```

并在 Manifest 中生命这两个 Service 组件

大家对比我们不做处理时的 oom_adj 值，可以明显看到差别。

3、进程唤醒

通过以上保活的策略确实让 App 避免过快的被系统干掉，但用户经常手动清理，也是一个比较头疼的问题，那如何能够在 App 被杀死后将其唤起，则成为了第二阶段的目标。

3.1 大树底下好乘凉

市面上推送 SDK 选择非常多，我们在集成推送 SDK 时又不想每个都集成一遍，这个时候我们就需要耍些花招了，大家知道，用户活跃度高的 App 大多是 BAT 系的，我们不妨就从这里入手，自己体系的 App 大都会集成自家的推送服务（你可以反编译几个看看，反编译微信并没有发现其用到什么推送，想想本身微信也是即时通讯类的 App 也就明白了）。这里我们就拿百度推送 SDK 开刀了，首先官网下载 SDK 包，其中找到开发文档，阅读 SDK 集成方法，从中找到蛛丝马迹，具体过程就不在赘述了，大家自己去看一下，就能够明白。

在百度推送 SDK 中，我们发现在 `CommandService` 中有这么一段被混淆的代码，但仍然不妨碍我们去大致猜测其实现：

```
public class CommandService extends Service {
    ...
    private void a(Intent var1) {
        String var2 = t.c(this, this.getPackageName(), var1.getAction());
        t.b("CommandService#onStartCommand#reflectReceiver#recevier = " + var2, t
        if(TextUtils.isEmpty(var2)) {
            a.b("CommandService", " reflectReceiver error: receiver for: " + var1
            var1.setPackage(this.getPackageName());
            this.sendBroadcast(var1);
        } else {
            try {
                Class var3 = Class.forName(var2);
                Constructor var4 = var3.getConstructor(new Class[0]);
```

```
                Object[] var9 = new Object[]{this.getApplicationContext(), var1};
                var8.invoke(var5, var9);
            } catch (Exception var11) {
                a.a("CommandService", var11);
            }

        }
    }
    ...
}
```

其实我们发现百度通过反射 `BroadcastReceiver` 中 `onReceive` 函数调用来唤醒集成了百度推送的 App，那既然如此我们干脆测试一下，将百度的 `RegistrationReceiver` 做壳，在其中启动自己的推送服务，接下来就简单的做壳及在 Manifest 中注册。

```
public class RegistrationReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("nuts", "baidu onReceiver invoke");
        Intent daemonIntent = new Intent(context, KeepAliveService.class);
        context.startService(daemonIntent);
    }
}
```

手动杀死自己的 App 模拟被系统杀死，通过 logcat 过滤 log 并启动百度地图或其他百度系 App 输出如下：

```
➜  ~ adb logcat -s nuts
--------- beginning of /dev/log/system
--------- beginning of /dev/log/main
D/nuts    (12804): baidu onReceiver invoke
D/nuts    (20650): KeepAliveService onStartCommand invoke
D/nuts    (20650): InnerService onStartCommand invoke
```

在实际测试中发现，部分机型 ROM 阻断了应用间唤醒，这个招式并不能覆盖到全部机型。

3.2 打铁还需自身硬

俗话说打铁还需自身硬，求人不如靠自己。在 Android 5.0 以后系统收紧了进程的管理，以节约用户电量消耗，但仍然提供以一个新的 `JobScheduler` API，允许应用在某个时间或者指定的条件下（例如，设备充电时）异步执行任务。官方推出这个 API 的出发点是好的，但也让我们有空子可钻。

我们先来通过继承 `JobService` 来实现一个 `WakeUpService` ：

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
public class WakeUpService extends JobService{
    @Override
    public boolean onStartJob(JobParameters jobParameters) {
        Log.d("nuts", "WakeUpService onStartJob");
        // to start push service or something else
        Intent daemonIntent = new Intent(this, KeepAliveService.class);
        startService(daemonIntent);
        return false;
    }

    @Override
    public boolean onStopJob(JobParameters jobParameters) {
        return false;
    }

    public static void startJobScheduler(Context ctx) {
        try {
            int jobId = 1002;
            JobInfo.Builder builder = new JobInfo.Builder(jobId, new Compon      n
            // 调整启动 service 间隔时间，这里为了测试随意写了 5 秒
            builder.setPeriodic(1000 * 5);
            builder.setPersisted(true);
```

```
        }
    }
}
```

同样的在 Manifest 中注册这个服务:

再次杀掉进程观察 log 输出:

```
➜  ~ adb logcat -s nuts
- waiting for device -
--------- beginning of /dev/log/system
--------- beginning of /dev/log/main
11-28 16:34:35.821  5675  5675 D nuts    : WakeUpService onStartJob
11-28 16:34:35.821  5675  5675 D nuts    : KeepAliveService onStartCommand invoke
11-28 16:34:35.841  5675  5675 D nuts    : InnerService onStartCommand invoke
11-28 16:34:40.871  5675  5675 D nuts    : WakeUpService onStartJob
11-28 16:34:40.881  5675  5675 D nuts    : KeepAliveService onStartCommand invoke
11-28 16:34:40.901  5675  5675 D nuts    : InnerService onStartCommand invoke
11-28 16:34:45.931  5675  5675 D nuts    : WakeUpService onStartJob
11-28 16:34:45.931  5675  5675 D nuts    : KeepAliveService onStartCommand invoke
11-28 16:34:45.951  5675  5675 D nuts    : InnerService onStartCommand invoke
```

我们看到 App 被杀死后，还是被唤起了。即使通过一键清理仍然能够唤醒，目前经过测试在 5.0 之后的机型还暂没有遇到不能唤醒的 case 。当然如果厂商将你的 App 加入了黑名单那是无论如何也起不来了。

4、总结

当然除了这些手段，还针对通知权限做了一些研究，在 Android 4.3/4.4 上可以强行打开用户通知

多厂商联合起来推出一个 Push 服务...算了不在 YY 了。

最后，以上所有手段请慎重使用！

Android

相关热门文章

2018年终总结（兼个人详历）

张风捷特烈 105 29

Android 内存泄露详解

Yuloran 7

我的2018年终总结（菜鸟的进阶之路）

jsonchao 23

利用RecyclerView实现无限轮播广告条

wizardev 9 1

燃烧吧!我的并发之魂--synchronized

张风捷特烈 66 4

wekin

但是还是为了我们小白长见识了

2年前 回复

wekin

这些其实市面上的推送都有做过吧

2年前 回复