引言

不论从事安卓应用开发，还是安卓系统研发，应该都遇到应用无响应（ANR，Application Not Responding）问题，当应用程序一段时间无法及时响应，则会弹出ANR对话框，让用户选择继续等待，还是强制关闭。

绝大多数人对ANR的了解仅停留在主线程耗时或CPU繁忙会导致ANR。面试过无数的候选人，几乎没有人能真正从系统级去梳理清晰ANR的来龙去脉，比如有哪些路径会引发ANR? 有没有可能主线程不耗时也出现ANR？如何更好的调试ANR?

如果没有深入研究过Android Framework的源代码，是难以形成对ANR有一个全面、正确的理解。研究系统源码以及工作实践后提炼而来，以图文并茂的方式跟大家讲解，相信定能帮忙大家加深对ANR的理解。

ANR触发机制

对于知识学习的过程，要知其然知其所以然，才能做到庖丁解牛般游刃有余。要深入理解ANR，就需要从根上去找寻答案，那就是ANR是如何触发的?

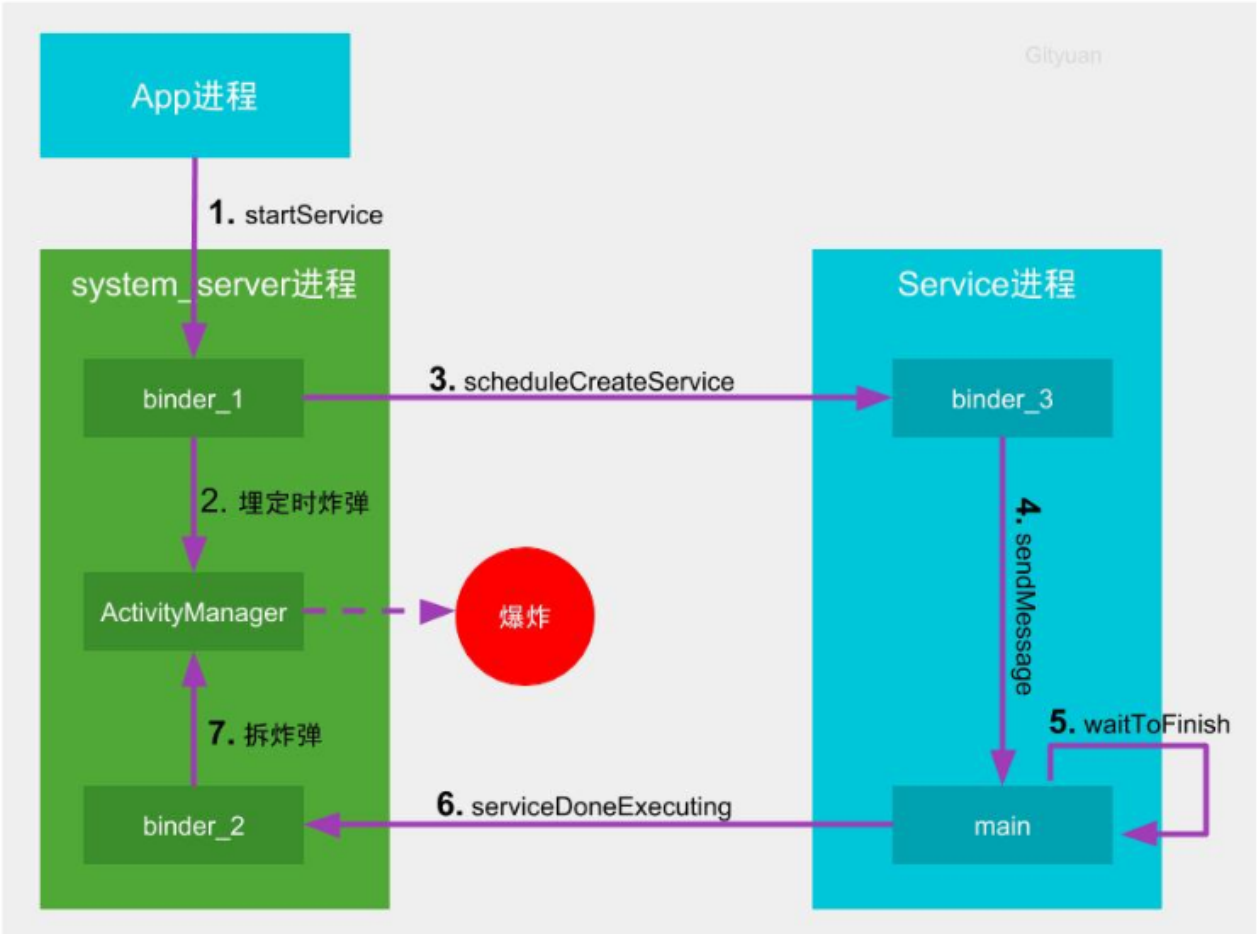
ANR是一套监控Android应用响应是否及时的机制，可以把发生ANR比作是引爆炸弹，那么整个流程包含三部分组成：

1. 埋定时炸弹：中控系统(system_server进程)启动倒计时，在规定时间内如果目标(应用进程)没有干完所有的活，则中控系统会定向炸毁(杀进程)目标。
2. 拆炸弹：在规定的时间内干完工地的所有活，并及时向中控系统报告完成，请求解除定时炸弹，则幸免于难。
3. 引爆炸弹：中控系统立即封装现场，抓取快照，搜集目标执行慢的罪证(traces)，便于后续的案件侦破(调试分析)，最后是炸毁目标。

常见的ANR有service、broadcast、provider以及input，，接下来本文以图文形式分别讲解。

service超时机制

下面来看看埋炸弹与拆炸弹在整个服务启动(startService)过程所处的环节。

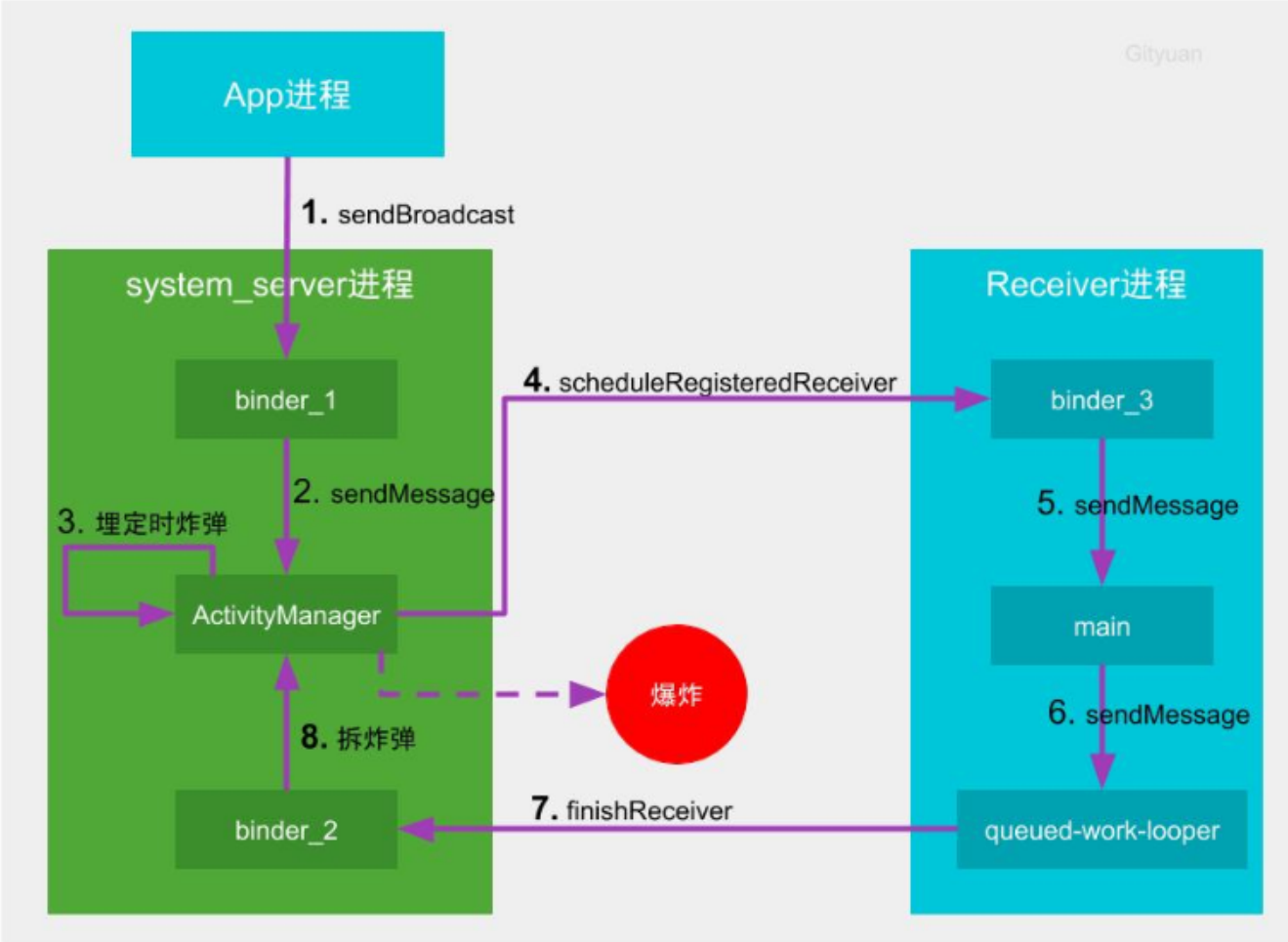


图解1：

1. 客户端(App进程)向中控系统(system_server进程)发起启动服务的请求
2. 中控系统派出一名空闲的通信员(binder_1线程)接收该请求，紧接着向组件管家(ActivityManager线程)发送消息，埋下定时炸弹
3. 通讯员1号(binder_1)通知工地(service所在进程)的通信员准备开始干活
4. 通讯员3号(binder_3)收到任务后转交给包工头(main主线程)，加入包工头的任务队列(MessageQueue)
5. 包工头经过一番努力干完活(完成service启动的生命周期)，然后等待SharedPreferences(简称SP)的持久化；
6. 包工头在SP执行完成后，立刻向中控系统汇报工作已完成
7. 中控系统的通讯员2号(binder_2)收到包工头的完工汇报后，立刻拆除炸弹。如果在炸弹倒计时结束前拆除炸弹则相安无事，否则会引发爆炸(触发ANR)

broadcast超时机制

broadcast跟service超时机制大抵相同，对于静态注册的广播在超时检测过程需要检测SP，如下图所示。

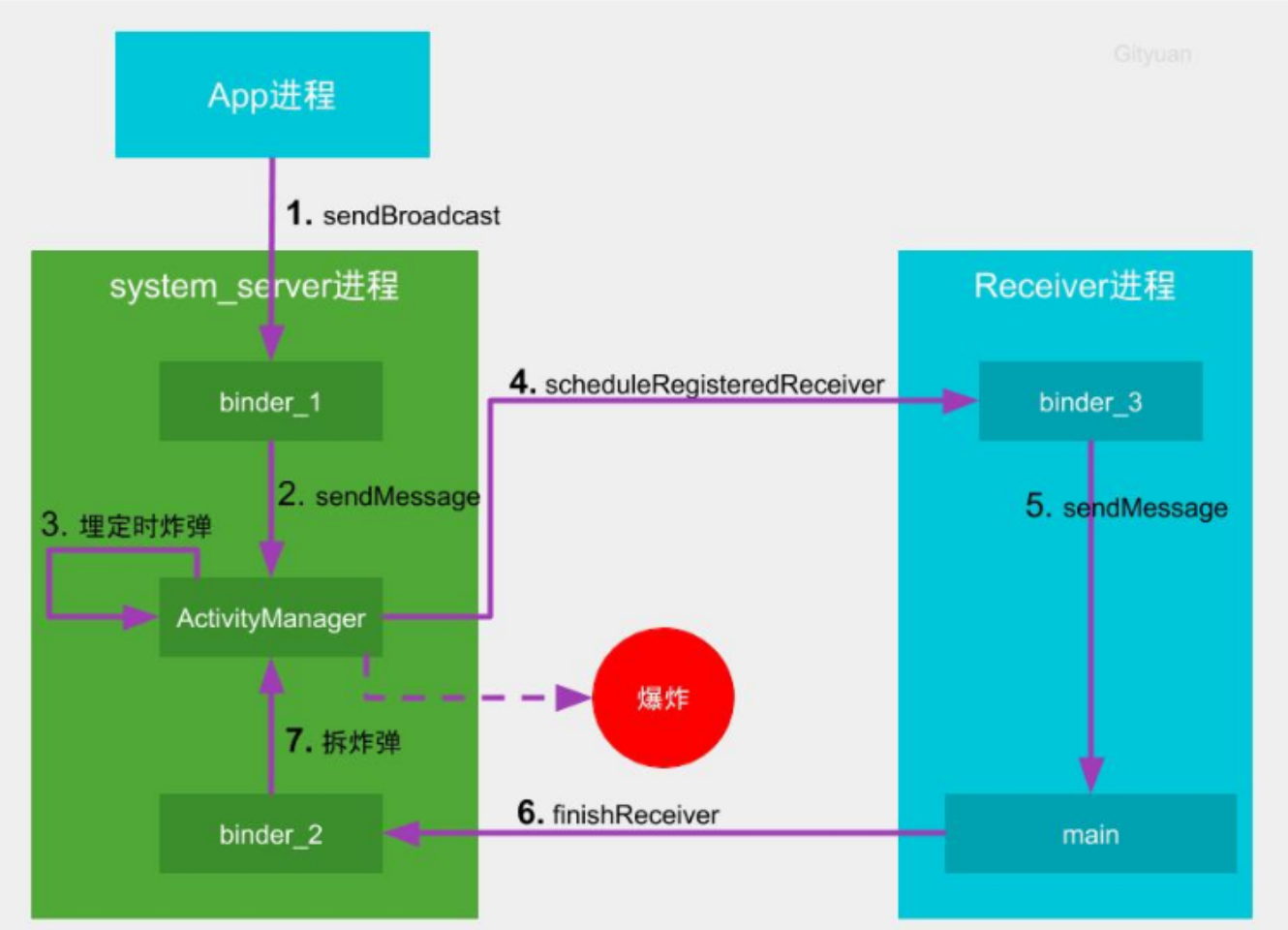


图解2：

1. 客户端(App进程)向中控系统(system_server进程)发起发送广播的请求
2. 中控系统派出一名空闲的通信员(binder_1)接收该请求转交给组件管家(ActivityManager线程)
3. 组件管家执行任务(processNextBroadcast方法)的过程埋下定时炸弹
4. 组件管家通知工地(receiver所在进程)的通信员准备开始干活
5. 通讯员3号(binder_3)收到任务后转交给包工头(main主线程)，加入包工头的任务队列(MessageQueue)
6. 包工头经过一番努力干完活(完成receiver启动的生命周期)，发现当前进程还有SP正在执行写入文件的操作，便将向中控系统汇报的任务交给SP工人(queued-work-looper线程)
7. SP工人历经艰辛终于完成SP数据的持久化工作，便可以向中控系统汇报工作完成
8. 中控系统的通讯员2号(binder_2)收到包工头的完工汇报后，立刻拆除炸弹。如果在倒计时结束前拆除炸弹则相安无事，否则会引发爆炸(触发ANR)

（说明：SP从8.0开始采用名叫“queued-work-looper”的handler线程，在老版本采用newSingleThreadExecutor创建的单线程的线程池）

如果是动态广播，或者静态广播没有正在执行持久化操作的SP任务，则不需要经过“queued-work-looper”线程中转，而是直接向中控系统汇报，流程更为简单，如下图所示：

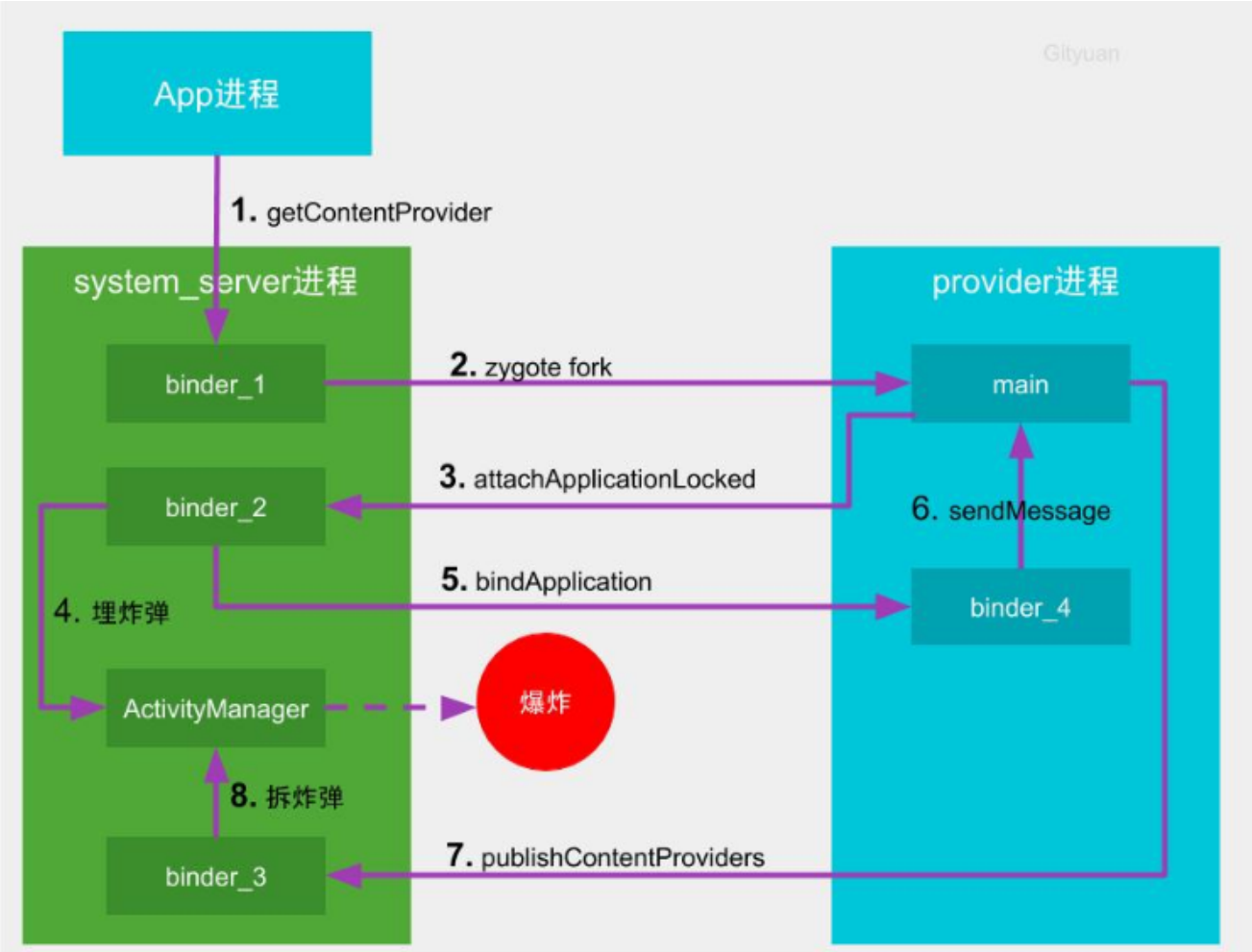


可见，只有XML静态注册的广播超时检测过程会考虑是否有SP尚未完成，动态广播并不受其影响。SP的apply将修改的数据项更新到内存，然后再异步同步数据到磁盘文件，因此很多地方会推荐在主线程调用采用apply方式，避免阻塞主线程，但静态广播超时检测过程需要SP全部持久化到磁盘，如果过度使用apply会增大应用ANR的概率，

Google这样设计的初衷是针对静态广播的场景下，保障进程被杀之前一定能完成SP的数据持久化。因为在向中控系统汇报广播接收者工作执行完成前，该进程的优先级为Foreground级别，高优先级下进程不但不会被杀，而且能分配到更多的CPU时间片，加速完成SP持久化。

provider超时机制

provider的超时是在provider进程首次启动的时候才会检测，当provider进程已启动的场景，再次请求provider并不会触发provider超时。



图解3：

1. 客户端(App进程)向中控系统(system_server进程)发起获取内容提供者的请求
2. 中控系统派出一名空闲的通信员(binder_1)接收该请求，检测到内容提供者尚未启动，则先通过zygote孵化新进程
3. 新孵化的provider进程向中控系统注册自己的存在
4. 中控系统的通信员2号接收到该信息后，向组件管家(ActivityManager线程)发送消息，埋下炸弹
5. 通信员2号通知工地(provider进程)的通信员准备开始干活
6. 通讯员4号(binder_4)收到任务后转交给包工头(main主线程)，加入包工头的任务队列(MessageQueue)
7. 包工头经过一番努力干完活(完成provider的安装工作)后向中控系统汇报工作已完成
8. 中控系统的通讯员3号(binder_3)收到包工头的完工汇报后，立刻拆除炸弹。如果在倒计时结束前拆除炸弹则相安无事，否则会引发爆炸(触发ANR)

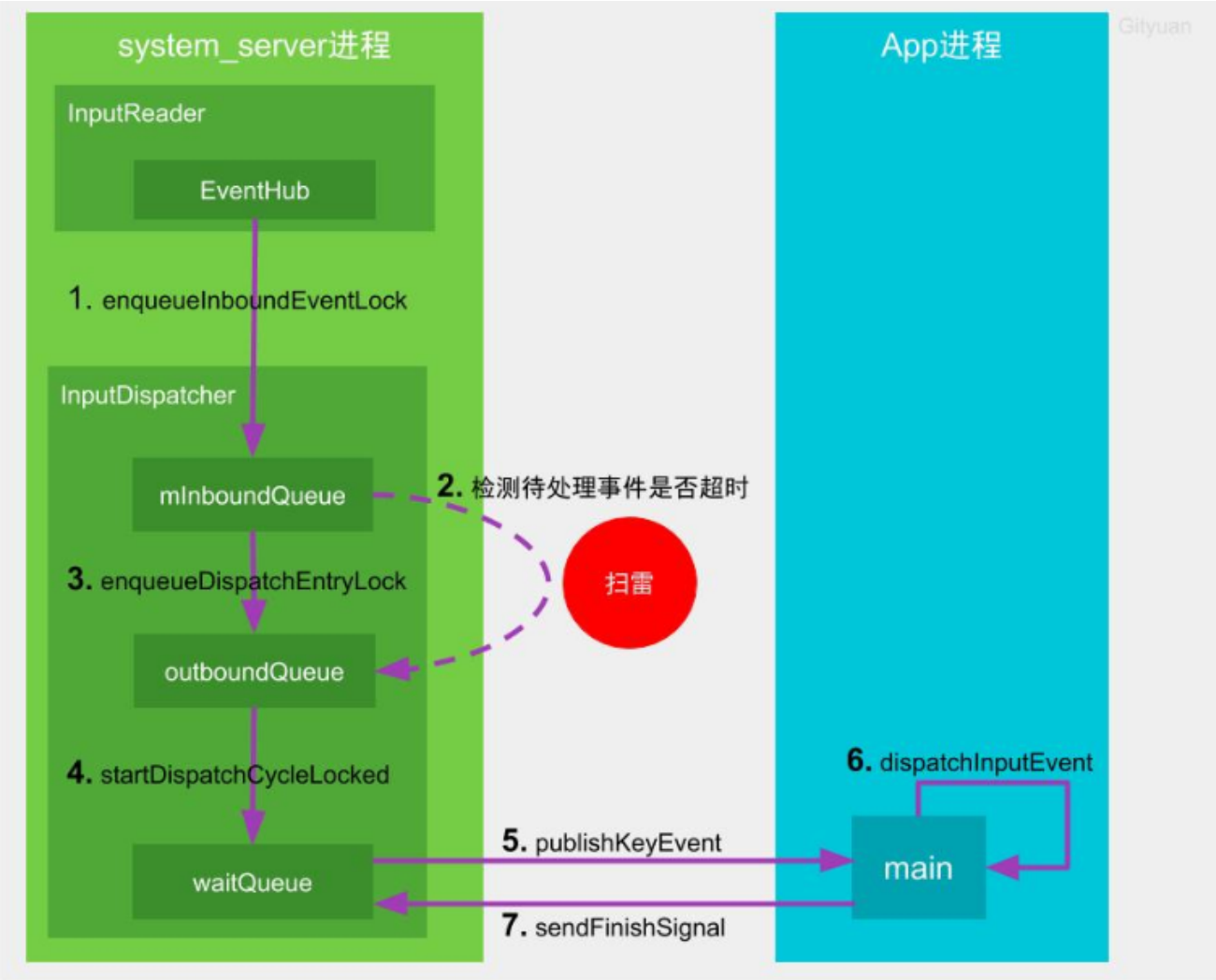
input超时机制

input的超时检测机制跟service、broadcast、provider截然不同，为了更好的理解input过程先来介绍两个重要线程的相关工作：

- InputReader线程负责通过EventHub(监听目录/dev/input)读取输入事件，一旦监听到输入事件则放入到InputDispatcher的mInBoundQueue队列，并通知其处理该事件；
- InputDispatcher线程负责将接收到的输入事件分发给目标应用窗口，分发过程使用到3个事件队列：

- mInBoundQueue用于记录InputReader发送过来的输入事件；
- outBoundQueue用于记录即将分发给目标应用窗口的输入事件；
- waitQueue用于记录已分发给目标应用，且应用尚未处理完成的输入事件；

input的超时机制并非时间到了一定就会爆炸，而是处理后续上报事件的过程才会去检测是否该爆炸，所以更像是扫雷的过程，具体如下图所示。



图解4：

1. InputReader线程通过EventHub监听底层上报的输入事件，一旦收到输入事件则将其放至mInBoundQueue队列，并唤醒InputDispatcher线程
2. InputDispatcher开始分发输入事件，设置埋雷的起点时间。先检测是否有正在处理的事件(mPendingEvent)，如果没有则取出mInBoundQueue队头的事件，并将其赋值给mPendingEvent，且重置ANR的timeout；否则不会从mInBoundQueue中取出事件，也不会重置timeout。然后检查窗口是否就绪(checkWindowReadyForMoreInputLocked)，满足以下任一情况，则会进入扫雷状态(检测前一个正在处理的事件是否超时)，终止本轮事件分发，否则继续执行步骤3。
   - 对于按键类型的输入事件，则outboundQueue或者waitQueue不为空，
   - 对于非按键的输入事件，则waitQueue不为空，且等待队头时间超时500ms
3. 当应用窗口准备就绪，则将mPendingEvent转移到outBoundQueue队列

4. 当outBoundQueue不为空，且应用管道对端连接状态正常，则将数据从outboundQueue中取出事件，放入waitQueue队列
5. InputDispatcher通过socket告知目标应用所在进程可以准备开始干活
6. App在初始化时默认已创建跟中控系统双向通信的socketpair，此时App的包工头(main线程)收到输入事件后，会层层转发到目标窗口来处理
7. 包工头完成工作后，会通过socket向中控系统汇报工作完成，则中控系统会将该事件从waitQueue队列中移除。

input超时机制为什么是扫雷，而非定时爆炸呢？是由于对于input来说即便某次事件执行时间超过timeout时长，只要用户后续在没有再生成输入事件，则不会触发ANR。 这里的扫雷是指当前输入系统中正在处理着某个耗时事件的前提下，后续的每一次input事件都会检测前一个正在处理的事件是否超时（进入扫雷状态），检测当前的时间距离上次输入事件分发时间点是否超过timeout时长。如果前一个输入事件，则会重置ANR的timeout，从而不会爆炸。

ANR超时阈值

不同组件的超时阈值各有不同，关于service、broadcast、contentprovider以及input的超时阈值如下表：

类型	前台	后台
Service	20s	200s
Broadcast	10s	60s
Provider	10s	
Input	5s	

前台与后台服务的区别

系统对前台服务启动的超时为20s，而后台服务超时为200s，那么系统是如何区别前台还是后台服务呢？来看看ActiveServices的核心逻辑：

```
ComponentName startServiceLocked(...) {
    final boolean callerFg;
    if (caller != null) {
        final ProcessRecord callerApp = mAm.getRecordForAppLocked(caller);
        callerFg = callerApp.setSchedGroup != ProcessList.SCHED_GROUP_BACKGROUND;
    } else {
        callerFg = true;
    }
    ...
    ComponentName cmp = startServiceInnerLocked(smap, service, r, callerFg, addToStarting);
    return cmp;
}
```

在startService过程根据发起方进程callerApp所属的进程调度组来决定被启动的服务是属于前台还是后台。当发起方进程不等于ProcessList.SCHED_GROUP_BACKGROUND(后台进程组)则认为是前台服务，否则为后台服务，并标记在ServiceRecord的成员变量createdFromFg。

什么进程属于SCHED_GROUP_BACKGROUND调度组呢？进程调度组大体可分为TOP、前台、后台，进程优先级（Adj）和进程调度组（SCHED_GROUP）算法较为复杂，其对应关系可粗略理解为Adj等于0的进程属于Top进程组，Adj等于100或者200的进程属于前台进程组，Adj大于200的进程属于后台进程组。关于Adj的含义见下表，简单来说就是Adj>200的进程对用户来说基本是无感知，主要是做一些后台工作，故后台服务拥有更长的超时阈值，同时后台服务属于后台进程调度组，相比前台服务属于前台进程调度组，分配更少的CPU时间片。

ADJ级别	取值	含义
NATIVE_ADJ	-1000	native进程
SYSTEM_ADJ	-900	仅指system_server进程
PERSISTENT_PROC_ADJ	-800	系统persistent进程
PERSISTENT_SERVICE_ADJ	-700	关联着系统或persistent进程
FOREGROUND_APP_ADJ	0	前台进程
VISIBLE_APP_ADJ	100	可见进程
PERCEPTIBLE_APP_ADJ	200	可感知进程，比如后台音乐播放
BACKUP_APP_ADJ	300	备份进程
HEAVY_WEIGHT_APP_ADJ	400	重量级进程
SERVICE_ADJ	500	服务进程
HOME_APP_ADJ	600	Home进程
PREVIOUS_APP_ADJ	700	上一个进程
SERVICE_B_ADJ	800	B List中的Service
CACHED_APP_MIN_ADJ	900	不可见进程的adj最小值
CACHED_APP_MAX_ADJ	906	不可见进程的adj最大值

`前台服务准确来说，是指由处于前台进程调度组的进程发起的服务`。这跟常说的fg-service服务有所不同，fg-service是指挂有前台通知的服务。

前台与后台广播超时

前台广播超时为10s，后台广播超时为60s，那么如何区分前台和后台广播呢？来看看AMS的核心逻辑：

```
BroadcastQueue broadcastQueueForIntent(Intent intent) {
    final boolean isFg = (intent.getFlags() & Intent.FLAG_RECEIVER_FOREGROUND) != 0;
    return (isFg) ? mFgBroadcastQueue : mBgBroadcastQueue;
}

mFgBroadcastQueue = new BroadcastQueue(this, mHandler,
        "foreground", BROADCAST_FG_TIMEOUT, false);
mBgBroadcastQueue = new BroadcastQueue(this, mHandler,
        "background", BROADCAST_BG_TIMEOUT, true);
```

根据发送广播sendBroadcast(Intent intent)中的intent的flags是否包含FLAG_RECEIVER_FOREGROUND来决定把该广播是放入前台广播队列或者后台广播队列，前台广播队列的超时为10s，后台广播队列的超时为60s，默认情况下广播是放入后台广播队列，除非指明加上FLAG_RECEIVER_FOREGROUND标识。

后台广播比前台广播拥有更长的超时阈值，同时在广播分发过程遇到后台service的启动(mDelayBehindServices)会延迟分发广播，等待service的完成，因为等待service而导致的广播ANR会被忽略掉；后台广播属于后台进程调度组，而前台广播属于前台进程调度组。简而言之，后台广播更不容易发生ANR，同时执行的速度也会更慢。

另外，只有串行处理的广播才有超时机制，因为接收者是串行处理的，前一个receiver处理慢，会影响后一个receiver；并行广播通过一个循环一次性向所有的receiver分发广播事件，所以不存在彼此影响的问题，则没有广播超时。

`前台广播准确来说，是指位于前台广播队列的广播`。

前台与后台ANR

除了前台服务，前台广播，还有前台ANR可能会让你云里雾里的，来看看其中核心逻辑：

```
final void appNotResponding(...) {
    ...
    synchronized (mService) {
        isSilentANR = !showBackground && !isInterestingForBackgroundTraces(app);
        ...
    }
    ...
    File tracesFile = ActivityManagerService.dumpStackTraces(
            true, firstPids,
            (isSilentANR) ? null : processCpuTracker,
            (isSilentANR) ? null : lastPids,
            nativePids);

    synchronized (mService) {
        if (isSilentANR) {
            app.kill("bg anr", true);
            return;
        }
        ...

        //弹出ANR选择的对话框
        Message msg = Message.obtain();
        msg.what = ActivityManagerService.SHOW_NOT_RESPONDING_UI_MSG;
        msg.obj = new AppNotRespondingDialog.Data(app, activity, aboveSystem);
        mService.mUiHandler.sendMessage(msg);
    }
}
```

决定是前台或者后台ANR取决于该应用发生ANR时对用户是否可感知，比如拥有当前前台可见的activity的进程，或者拥有前台通知的fg-service的进程，这些是用户可感知的场景，发生ANR对用户体验影响比较大，故需要弹框让用户决定是否退出还是等待，如果直接杀掉这类应用会给用户造成莫名其妙的闪退。

后台ANR相比前台ANR，只抓取发生无响应进程的trace，也不会收集CPU信息，并且会在后台直接杀掉该无响应的进程，不会弹框提示用户。

前台ANR准确来说，是指对用户可感知的进程发生的ANR。

ANR爆炸现场

对于service、broadcast、provider、input发生ANR后，中控系统会马上去抓取现场的信息，用于调试分析。收集的信息包括如下：

- 将am_anr信息输出到EventLog，也就是说ANR触发的时间点最接近的就是EventLog中输出的am_anr信息
- 收集以下重要进程的各个线程调用栈trace信息，保存在data/anr/traces.txt文件
    - 当前发生ANR的进程，system_server进程以及所有persistent进程
    - audioserver, cameraserver, mediaserver, surfaceflinger等重要的native进程
    - CPU使用率排名前5的进程
- 将发生ANR的reason以及CPU使用情况信息输出到main log
- 将traces文件和CPU使用情况信息保存到dropbox，即data/system/dropbox目录
- 对用户可感知的进程则弹出ANR对话框告知用户，对用户不可感知的进程发生ANR则直接杀掉

整个ANR信息收集过程比较耗时，其中抓取进程的trace信息，每抓取一个等待200ms，可见persistent越多，等待时间越长。关于抓取trace命令，对于Java进程可通过在adb shell环境下执行kill -3 [pid]可抓取相应pid的调用栈；对于Native进程在adb shell环境下执行debuggerd -b [pid]可抓取相应pid的调用栈。对于ANR问题发生后的蛛丝马迹(trace)在traces.txt和dropbox目录中保存记录。

有了现场信息，可以调试分析，先定位发生ANR时间点，然后查看trace信息，接着分析是否有耗时的message、binder调用，锁的竞争，CPU资源的抢占，以及结合具体场景的上下文来分析，调试手段就需要针对前面说到的message、binder、锁等资源从系统角度细化更多debug信息，这里不再展开，后续再以ANR案例来讲解。

作为应用开发者应让主线程尽量只做UI相关的操作，避免耗时操作，比如过度复杂的UI绘制，网络操作，文件IO操作；避免主线程跟工作线程发生锁的竞争，减少系统耗时binder的调用，谨慎使用sharePreference，注意主线程执行provider query操作。简而言之，尽可能减少主线程的负载，让其空闲待命，以期可随时响应用户的操作。

回答

最后，来回答文章开头的提问，有哪些路径会引发ANR? 答应是从埋下定时炸弹到拆炸弹之间的任何一个或多个路径执行慢都会导致ANR（以service为例），可以是service的生命周期的回调方法(比如onStartCommand)执行慢，可以是主线程的消息队列存在其他耗时消息让service回调方法迟迟得不到执行，可以是SP操作执行慢，可以是system_server进程的binder线程繁忙而导致没有及时收到拆炸弹的指令。另外ActivityManager线程也可能阻塞，出现的现象就是前台服务执行时间有可能超过10s，但并不会出现ANR。

发生ANR时从trace来看主线程却处于空闲状态或者停留在非耗时代码的原因有哪些？可以是抓取trace过于耗时而错过现场，可以是主线程消息队列堆积大量消息而最后抓取快照一刻只是瞬时状态，可以是广播的“queued-work-looper”一直在处理SP操作。