8月1日-8月2日

学习内容：

学习了安卓开发中的相对布局、帧布局还有自定义布局

学习收获：

RelativeLayout又称作相对布局，也是一种非常常用的布局。和LinearLayout的排列规则不同，RelativeLayout显得更加随意，它可以通过相对定位的方式让控件出现在布局的任何位置。也正因为如此，RelativeLayout中的属性非常多，不过这些属性都是有规律可循的，其实并不难理解和记忆。

```
android:layout_alignParentBottom="true"
android:layout_alignParentRight="true"
```

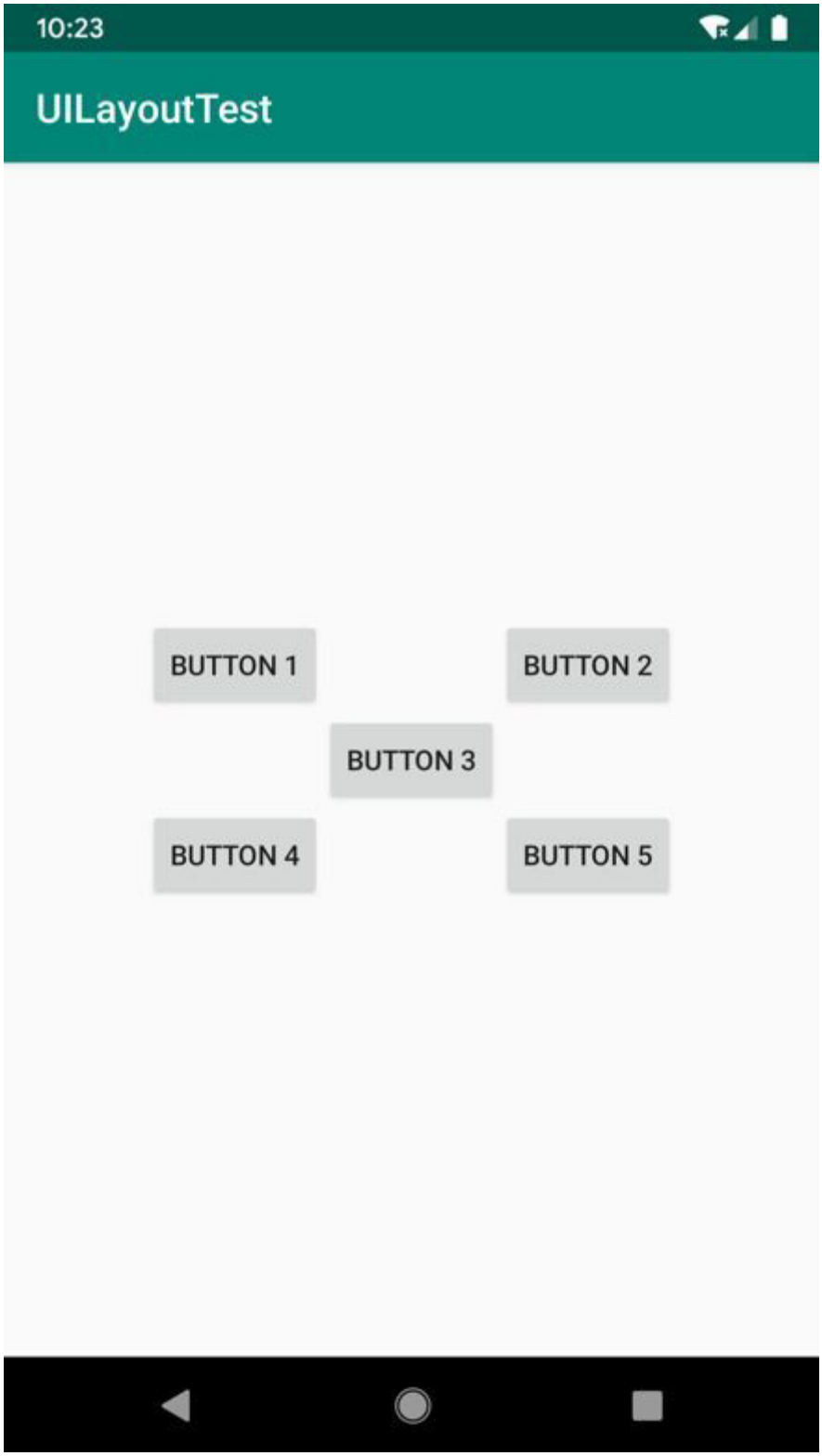
正如上面两行，我们可以很简单的通过翻译属性得知它的含义和用法。

当然，上面这两种都是针对于父布局进行定位的，当然控件也可以相对于控件进行定位，如下。

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent">
<Button
android:id="@+id/button3"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_centerInParent="true"
android:text="Button 3" />
<Button
android:id="@+id/button1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_above="@id/button3"
android:layout_toLeftOf="@id/button3"
android:text="Button 1" />
<Button
android:id="@+id/button2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_above="@id/button3"
android:layout_toRightOf="@id/button3"
android:text="Button 2" />
<Button
android:id="@+id/button4"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_below="@id/button3"
android:layout_toLeftOf="@id/button3"
android:text="Button 4" />
<Button
android:id="@+id/button5"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_below="@id/button3"
```

```
android:layout_toRightOf="@id/button3"
android:text="Button 5" />
</RelativeLayout>
```

效果如图



接下来是帧布局和自定义布局

FrameLayout又称作帧布局，它相比于前面两种布局就简单太多了，因此它的应用场景少了很多。这种布局没有丰富的定位方式，所有的控件都会默认摆放在布局的左上角。当然也可以通过 Android:layout_gravity对摆放位置做调整。这种方法麻烦在于它不会按照某种方式排列，而是会将所有控件堆在一起。

我们所用的所有控件都是直接或间接继承自View的，所用的所有布局都是直接或间接继承自ViewGroup的。View是Android中最基本的一种UI组件，它可以在屏幕上绘制一块矩形区域，并能响应这块区域的各种事件，因此，我们使用的各种控件其实就是在View的基础上又添加了各自特有的功能。而ViewGroup则是一种特殊的View，它可以包含很多子View和子ViewGroup，是一个用于放置控件和布局的容器。

接下来我们试着创建一个一个标题栏来作为我们自创的布局

首先在 layout 目录下创建一个 title.xml ，可以代替系统自带的标题栏

构建好后只需要在对应需要引入布局的 xml 文件中添加以下代码

```
<include layout="@layout/title" />
```

记得在相应的 onCreate 中将系统自带的标题栏隐藏

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second_layout);
    。。。。。。
            //打算在这个活动中使用自定义的布局，需要将系统自带的标题栏隐藏掉
        ActionBar actionBar = getSupportActionBar();
        if (actionBar != null)
            actionBar.hide();
}
```

学习内容

Android 中 lmkd 机制的基本构成

学习收获：

Android 低内存终止守护程序（`lmkd`），全称 Low Memory Killer Daemon，该进程可以监控运行中 Android 系统的内存状态，并通过终止最不必要的进程来应对内存压力大的问题，使系统以可接受的性能水平运行。

主要有几个关键的概念需要关注：

- 内存压力：代表一种系统内存不足的状态
- 压力失速信息（PSI）：是由内核产生，给 lmkd 使用，注意不是给 lmk 驱动程序用。它主要是使用内核压力失速信息监视器来检测内存压力。
- PSI 监视器和 Vmpressure 信号：Vmpressure 信号也是由内核产生，给 lmkd 使用，但是通常包含大量误判，因此需要 lmkd 执行过滤以确定是否真的存在内存压力。使用 PSI 监视器可以更加精确的实现内存压力检测，最大幅度地减少过滤开销。**需要注意一点，如果要使用 psi 监视器，需要配置 `ro.lmk.use_psi` 属性。**

同时，我们需要知道内核的 lmk 驱动程序都有那些缺点：

- 对于低内存设备，必须主动进行调整，而不能自动完成，在处理大活动页面缓存的相关任务时，性能较差，容易发生抖动，但是一般不会终止该进程，导致用户体验差。
- LMK 内核驱动程序一栏与可用内存限制，不会根据内存压力进行扩缩。

在 Android P 之后，内核就没有 lmk 驱动程序了，kill 的操作都交给了用户空间的 lmkd 。

用户空间 `lmkd` 支持基于以下各项的终止策略：

`vmpressure` 事件或 PSI 监视器、其严重性以及交换利用率等其他提示。低内存设备和高性能设备的终止策略有所不同：

- 对于内存不足的设备，一般情况下，系统会选择承受较大的内存压力。
- 对于高性能设备，如果出现内存压力，则会视为异常情况，应及时修复，以免影响整体性能。

用户空间 `lmkd` 还支持一种旧模式，在该模式下，它使用与内核中的 LMK 驱动程序相同的策略（即可用内存和文件缓存阈值）做出终止决策。要启用旧模式，请将 `ro.lmk.use_minfree_levels` 属性设置为 `true`。

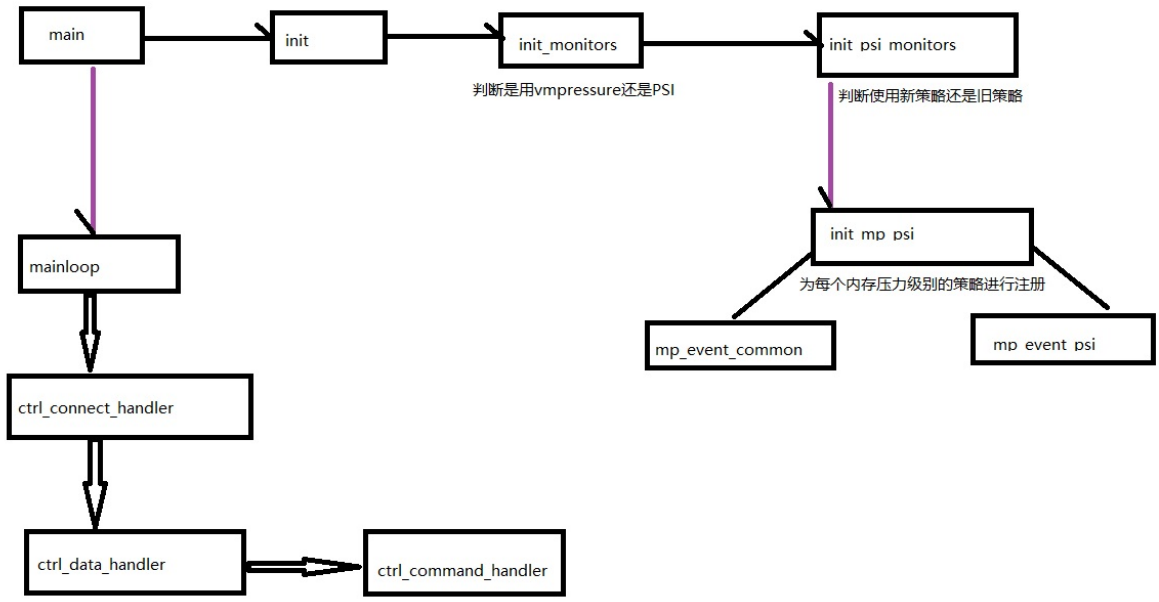
Android 11 通过引入新的终止策略改进了 `lmkd`。该终止策略使用 PSI 机制来执行 Android 10 中引入的内存压力检测。Android 11 中的 `lmkd` 会根据内存资源使用情况和抖动来防止出现内存不足和性能下降。这一终止策略取代了以前的策略，可同时用于高性能设备和低内存 (Android Go) 设备。

8月3日-8月4日

学习内容：

lmkd 的流程分析，同时伴随着一定的源码解读

学习收获：



lmkd是系统一个非常重要的服务，开机是由init进程启动。

服务启动后，入口在system/core/lmkd/lmkd.c文件的main函数中，主要做了如下几件事：

1. 读取配置参数
2. 初始化 epoll 事件监听
3. 锁住内存页
4. 设置进程调度器

5. 循环处理事件

以下的代码解读主要是在代码中添加便于理解的注释，按照调用顺序进行排序。

main 函数解读

主要是看代码中的注释

```
int main(int argc, char **argv) {
    ...
    update_props(); //step 1，进程最初，需要获取所有的lmkd 的prop，为init 做准备

    ctx = create_android_logger(KILLINFO_LOG_TAG);

    if (!init()) { //step 2，init 处理所有核心的初始化工作
        if (!use_inkernel_interface) { //step 3，如果不再使用旧的LMK 驱动程序
            ...
            //step4，给虚拟空间上锁，防止内存交换
            if (mlockall(MCL_CURRENT | MCL_FUTURE | MCL_ONFAULT) && (errno !=
EINVAL)) {
                ALOGW("mlockall failed %s", strerror(errno));
            }

            //step 4，添加调度策略，即先进先出
            struct sched_param param = {
                    .sched_priority = 1,
            };
            if (sched_setscheduler(0, SCHED_FIFO, &param)) {
                ALOGW("set SCHED_FIFO failed %s", strerror(errno));
            }
        }

        mainloop(); //step 5,  进入循环，等待polling
    }

    android_log_destroy(&ctx);

    ALOGI("exiting");
    return 0;
}
```

基本的流程见上面代码的注释部分，**可以看到 lmkd 的核心部分在step 2 和step 5。**

mlockall 锁空间解析

```
 if (mlockall(MCL_CURRENT | MCL_FUTURE | MCL_ONFAULT) && (errno != EINVAL)) {
        ALOGW("mlockall failed %s", strerror(errno));
    }
```

- mlockall 函数将调用进程的全部虚拟地址空间加锁。防止出现内存交换，将该进程的地址空间交换到外存上。
- mlockall 将所有映射到进程地址空间的内存上锁。
- flags可取两个值：MCL_CURRENT, MCL_FUTURE
    - MCL_CURRENT: 表示对所有已经映射到进程地址空间的页上锁
    - MCL_FUTURE: 表示对所有将来映射到进程地空间的页都上锁。

init 函数解读

step 1. 创建epoll

```
    epollfd = epoll_create(MAX_EPOLL_EVENTS);
    if (epollfd == -1) {
        ALOGE("epoll_create failed (errno=%d)", errno);
        return -1;
    }
```

整个lmkd 都是依赖epoll 机制，这里创建了 9 个event：

```
/*
 * 1 ctrl listen socket, 3 ctrl data socket, 3 memory pressure levels,
 * 1 lmk events + 1 fd to wait for process death
 */
#define MAX_EPOLL_EVENTS (1 + MAX_DATA_CONN + VMPRESS_LEVEL_COUNT + 1 + 1)
```

step 2. 初始化socket lmkd

```
    ctrl_sock.sock = android_get_control_socket("lmkd");
    if (ctrl_sock.sock < 0) {
        ALOGE("get lmkd control socket failed");
        return -1;
    }

    ret = listen(ctrl_sock.sock, MAX_DATA_CONN);
    if (ret < 0) {
        ALOGE("lmkd control socket listen failed (errno=%d)", errno);
        return -1;
    }

    epev.events = EPOLLIN;
//当 socket lmkd 有客户链接时，执行下面的回调函数进行数据处理
    ctrl_sock.handler_info.handler = ctrl_connect_handler;
    epev.data.ptr = (void *)&(ctrl_sock.handler_info);
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, ctrl_sock.sock, &epev) == -1) {
        ALOGE("epoll_ctl for lmkd control socket failed (errno=%d)", errno);
        return -1;
    }
    maxevents++;
```

ctrl_sock 主要存储的是socket lmkd 的fd 和handle info，主要注意这里的ctrl_connect_handler()。

step 3. 确定是否用LMK 驱动程序

```
 #define INKERNEL_MINFREE_PATH "/sys/module/lowmemorykiller/parameters/minfree"

   has_inkernel_module = !access(INKERNEL_MINFREE_PATH, W_OK);
   use_inkernel_interface = has_inkernel_module;
//如果还存在就使用，不存在就不用啦
```

通过函数access 确认旧的节点是否还存在，用以确认kernel 是否还在用LMK 驱动程序。

之所以有这样的处理，应该是Android 为了兼容旧版本kernel。

step 4. init_monitors

该函数是init 函数中的核心了，这里用来注册PSI 的监视器策略或者是common 的adj 策略，并将其添加到epoll 中。

```
static bool init_monitors() {
    /* Try to use psi monitor first if kernel has it */
    use_psi_monitors = property_get_bool("ro.lmk.use_psi", true) &&
        init_psi_monitors();
    /* Fall back to vmpressure */
    if (!use_psi_monitors &&
        (!init_mp_common(VMPRESS_LEVEL_LOW) ||
        !init_mp_common(VMPRESS_LEVEL_MEDIUM) ||
        !init_mp_common(VMPRESS_LEVEL_CRITICAL))) {
        ALOGE("Kernel does not support memory pressure events or in-kernel low
memory killer");
        return false;
    }
    ...
    return true;
}
```

变量use_psi_monitors 用以确认是使用 PSI 还是vmpressure

- 如果使用vmpressure，则通过init_mp_common 来初始化kill 策略；
- 如果使用PSI，则通过init_psi_monitors 来初始化kill 策略；

所以lmkd 中如果使用 PSI，要求 **ro.lmk.use_psi** 为 true。

另外，lmkd 支持旧模式的kill 策略，只要 **ro.lmk.use_new_strategy** 设为false，或者将 **ro.lmk.use_minfree_levels** 设为true（针对非低内存设备，即**ro.config.low_ram** 不为true）：

```
static bool init_psi_monitors() {
    bool use_new_strategy =
        property_get_bool("ro.lmk.use_new_strategy", low_ram_device ||
!use_minfree_levels);
```

继续深入分析init_psi_monitors：

```
static bool init_psi_monitors() {
    /*
     * When PSI is used on low-ram devices or on high-end devices without
memfree levels
     * use new kill strategy based on zone watermarks, free swap and thrashing
stats
     */
    //判断是不是新策略
    bool use_new_strategy =
```

```
        property_get_bool("ro.lmk.use_new_strategy", low_ram_device ||
!use_minfree_levels);

    //是新策略就要重设覆盖之前的阈值数组
    /* In default PSI mode override stall amounts using system properties */
    if (use_new_strategy) {
        /* Do not use low pressure level */
        psi_thresholds[VMPRESS_LEVEL_LOW].threshold_ms = 0;
        psi_thresholds[VMPRESS_LEVEL_MEDIUM].threshold_ms =
psi_partial_stall_ms;
        psi_thresholds[VMPRESS_LEVEL_CRITICAL].threshold_ms =
psi_complete_stall_ms;
    }
    //通过init_mp_psi为每个级别的策略进行注册

    if (!init_mp_psi(VMPRESS_LEVEL_LOW, use_new_strategy)) {
        return false;
    }
    if (!init_mp_psi(VMPRESS_LEVEL_MEDIUM, use_new_strategy)) {
        destroy_mp_psi(VMPRESS_LEVEL_LOW);
        return false;
    }
    if (!init_mp_psi(VMPRESS_LEVEL_CRITICAL, use_new_strategy)) {
        destroy_mp_psi(VMPRESS_LEVEL_MEDIUM);
        destroy_mp_psi(VMPRESS_LEVEL_LOW);
        return false;
    }
    return true;
}
```

函数比较简单的，最开始的变量use_new_strategy 用以确认是使用PSI 策略还是vmpressure。如果是使用PSI 策略，psi_thresholds数组中的threshold_ms 需要重新赋值为prop 指定的值（也就是说支持动态配置）。最后通过init_mp_psi 为每个级别的strategy 进行最后的注册，当然对于PSI，只有some 和full 等级，所以与level 中的medium 和 critical 分别对应。

这里的psi_thresholds 数组中threshold_ms 通过prop：

- **ro.lmk.psi_partial_stall_ms low_ram** 默认为200ms，PSI 默认为70ms；
- **ro.lmk.psi_complete_stall_ms** 默认700ms；

接下来看下init_mp_psi 到底做了些什么：

```
static bool init_mp_psi(enum vmpressure_level level, bool use_new_strategy) {
    int fd;

    /* Do not register a handler if threshold_ms is not set */
    if (!psi_thresholds[level].threshold_ms) {
        return true;
    }

    //step1: 传入level值，后期如果超过了阈值就会触发epoll
    fd = init_psi_monitor(psi_thresholds[level].stall_type,
        psi_thresholds[level].threshold_ms * US_PER_MS,
        PSI_WINDOW_SIZE_MS * US_PER_MS);

    if (fd < 0) {
```

```
        return false;
    }
    //step2: 在这里判断是新策略还是之前的旧策略，分别对应 mp_event_psi 和
mp_event_common
    vmpressure_hinfo[level].handler = use_new_strategy ? mp_event_psi :
mp_event_common;
    vmpressure_hinfo[level].data = level;
    //step3: 通过register_psi_monitor 将节点/proc/pressure/memory 添加到epoll 中监听
    if (register_psi_monitor(epollfd, fd, &vmpressure_hinfo[level]) < 0) {
        destroy_psi_monitor(fd);
        return false;
    }
    maxevents++;
    mpevfd[level] = fd;

    return true;
}
```

函数比较简单，主要分三步：

- 通过init_psi_monitor 将不同level 的值写入节点/proc/pressure/memory，后期阈值如果超过了设定就会触发一次epoll；
- 根据use_new_strategy，选择是新策略mp_event_psi，还是旧模式mp_event_common，详细的策略见第**8** 节和第**10** 节；
- 通过register_psi_monitor 将节点/proc/pressure/memory 添加到epoll 中监听；

step 5. 标记进入lmkd 流程

```
property_set("sys.lmk.reportkills", "1");
```

至此，init 基本剖析完成，主要：

- 创建epoll，用以监听 9 个event；
- 初始化socket /dev/socket/lmkd，并将其添加到epoll 中；
- 根据prop ro.lmk.use_psi 确认是否使用PSI 还是vmpressure；
- 根据prop **ro.lmk.use_new_strategy** 或者通过 prop **ro.lmk.use_minfree_levels** 和 prop **ro.config.low_ram** 使用PSI 时的新策略还是旧策略；
- 新、旧策略主要体现在mp_event_psi 和mp_event_common 处理，而本质都是通过节点/proc/pressure/memory 获取内存压力是否达到some/full 指定来确认是否触发event；
- 后期epoll 触发主要的处理函数是mp_event_psi 或 mp_event_common；

mainloop 解读

主要是使用epoll 机制，通过epoll_wait 阻塞等待触发，如下：

```
nevents = epoll_wait(epollfd, events, maxevents, -1);
```

等待唤醒后主要做了两件事情，确认是否有connect 断开，执行handler：

```
static void mainloop(void) {
......
    while (1) {
        nevents = epoll_wait(epollfd, events, maxevents, -1);
        for (i = 0, evt = &events[0]; i < nevents; ++i, evt++) {
```

```
            if ((evt->events & EPOLLHUP) && evt->data.ptr) {
                ALOGI("lmkd data connection dropped");
                handler_info = (struct event_handler_info*)evt->data.ptr;
                ctrl_data_close(handler_info->data);
            }
        }

    /* Second pass to handle all other events */
        for (i = 0, evt = &events[0]; i < nevents; ++i, evt++) {
            if (evt->events & EPOLLERR) {
                ALOGD("EPOLLERR on event #%d", i);
            }
            if (evt->events & EPOLLHUP) {
                /* This case was handled in the first pass */
                continue;
            }
            if (evt->data.ptr) {
                handler_info = (struct event_handler_info*)evt->data.ptr;
                call_handler(handler_info, &poll_params, evt->events);
                //这里会回调ctrl_connect_handler
            }
        }

        ......
    }
}
```

```
static void ctrl_data_handler(int data, uint32_t events) {
    if (events & EPOLLIN) {
        ctrl_command_handler(data);
    }
}
```

```
// lmkd进程的客户端是ActivityManager，通过socket(dev/socket/lmkd)跟 lmkd 进行通信，
// 当有客户连接时，就会回调ctrl_connect_handler函数。
static void ctrl_connect_handler(int data __unused, uint32_t events __unused) {
.........
 // ctrl_sock上调用accept接收客户端的连接
data_sock[free_dscock_idx].sock = accept(ctrl_sock.sock, NULL, NULL);
ALOGI("lmkd data connection established");
......
/* use data to store data connection idx */
data_sock[free_dscock_idx].handler_info.data = free_dscock_idx;
// 客户连接对应的处理函数
data_sock[free_dscock_idx].handler_info.handler = ctrl_data_handler;
......
}
```

从init 中可以知道 epoll 主要监听了 9 个event，不同的event fd 对应不同的handler 处理逻辑。这些 handler 大致分为：

- **一个**socket listener fd 监听，主要是/dev/socket/lmkd，在init() 中添加到epoll；
- **三个**客户端socket data fd 的数据通信，在ctrl_connect_handler() 中添加到epoll，**在step2中有具体的回调函数：ctrl_sock.handler_info.handler = ctrl_connect_handler;**

- **三个**presurre 状态的监听，在init_psi_monitors() -> init_mp_psi() 中添加到epoll；（或者 init_mp_common 的旧策略）**后面会提到。**
- **一个**是LMK event kpoll_fd 监听，在init() 中添加到epoll，目前新的lmkd **不再使用这个监听**；
- **一个**是wait 进程death 的pid fd 监听，在start_wait_for_proc_kill() 中添加到epoll；

ctrl listener fd 的处理流程 ctrl_connect_handler

首先，在init 中得知，socket lmkd 在listen 之后会将fd 添加到epoll 中，用以监听socket 从上一节 mainloop 得知epoll 触发后会调用event 对应的handler 接口，对于 lmkd，如果connect 成功后会触发ctrl_connect_handler。

```
static void ctrl_connect_handler(int data __unused, uint32_t events __unused,
                                 struct polling_params *poll_params __unused) {
    struct epoll_event epev;
    int free_dscock_idx = get_free_dsock();

    if (free_dscock_idx < 0) {
        for (int i = 0; i < MAX_DATA_CONN; i++) {
            //通过这个断开连接
            ctrl_data_close(i);
        }
        free_dscock_idx = 0;
    }
    //通过accept添加连接
    data_sock[free_dscock_idx].sock = accept(ctrl_sock.sock, NULL, NULL);
    if (data_sock[free_dscock_idx].sock < 0) {
        ALOGE("lmkd control socket accept failed; errno=%d", errno);
        return;
    }
//和lmkd的连接建立
    ALOGI("lmkd data connection established");
    /* use data to store data connection idx */
    //存储连接数据的idx
    data_sock[free_dscock_idx].handler_info.data = free_dscock_idx;
    //这个就是data的处理函数
    data_sock[free_dscock_idx].handler_info.handler = ctrl_data_handler;
    data_sock[free_dscock_idx].async_event_mask = 0;
    //加上标记
    epev.events = EPOLLIN;
    epev.data.ptr = (void *)&(data_sock[free_dscock_idx].handler_info);
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, data_sock[free_dscock_idx].sock,
&epev) == -1) {       //通信失败
        ALOGE("epoll_ctl for data connection socket failed; errno=%d", errno);
        ctrl_data_close(free_dscock_idx);
        return;
    }
    maxevents++;
}
```

对于 lmkd 会提供最大 3 个的客户端连接，如果超过3个后要进行ctrl_data_close() 以断开epoll 和 socket。

如果没有超过的话，会通过accept 创建个新的data socket，并将其添加到epoll 中。

主要注意的是data 的交互函数ctrl_data_handler().

ctrl data fd 的处理流程 ctrl_data_handler

```
static void ctrl_data_handler(int data, uint32_t events,
                              struct polling_params *poll_params __unused) {
    if (events & EPOLLIN) {
        ctrl_command_handler(data);
    }
}
```

客户端建立连接后，通过socket给lmkd发送命令，命令的执行操作在函数ctrl_data_handler中处理的。当时添加到epoll 时是以EPOLLIN 添加的，所以这里接着会调用ctrl_command_handler，主要处理从ProcessList.java 中发出的几个 lmk command：

```
enum lmk_cmd {
    LMK_TARGET = 0, // 将minfree与oom_adj_score关联起来

    LMK_PROCPRIO,   //  注册进程并设置oom_adj_score
    LMK_PROCREMOVE, //  注销进程
    LMK_PROCPURGE,  //  清除所有已注册的进程
    LMK_GETKILLCNT, //  获取被杀次数
    LMK_SUBSCRIBE,  /* Subscribe for asynchronous events */
    LMK_PROCKILL,   /* Unsolicited msg to subscribed clients on proc kills */
    LMK_UPDATE_PROPS, /* Reinit properties */
};
```

lmkd 中ctrl_command_handler 函数根据cmd 解析出相应的指令，调用相应的函数

```
/* LMK_TARGET packet payload */
struct lmk_target {
    int minfree;
    int oom_adj_score;
};

/* LMK_PROCPRIO packet payload */
struct lmk_procprio {
    pid_t pid;
    uid_t uid;
    int oomadj;
};

/* LMK_PROCREMOVE packet payload */
struct lmk_procremove {
    pid_t pid;
};

/* LMK_GETKILLCNT packet payload */
struct lmk_getkillcnt {
    int min_oomadj;
    int max_oomadj;
};

static void ctrl_command_handler(int dsock_idx) {
 ......
    switch(cmd) {
        case LMK_TARGET:
```

```
                // 解析socket packet里面传过来的数据，写入lowmem_minfree和lowmem_adj两个
数组中，
                // 用于控制low memory的行为；
                // 设置sys.lmk.minfree_levels，比如属性值：
                // [sys.lmk.minfree_levels]:
[18432:0,23040:100,27648:200,85000:250,191250:900,241920:950]
                 cmd_target(targets, packet);
            case LMK_PROCPRIO:
            // 设置进程的oomadj，把oomadj写入对应的节点(/proc/pid/oom_score_adj)中；
            // 将oomadj保存在一个哈希表中。
            // 哈希表 pidhash 是以 pid 做 key，proc_slot 则是把 struct proc 插入到以
oomadj 为 key 的哈希表 procadjslot_list 里面
                 cmd_procprio(packet);
            case LMK_PROCREMOVE:
            //  解析socket传过来进程的pid,
            // 通过pid_remove 把这个 pid 对应的 struct proc 从 pidhash 和
procadjslot_list 里移除
                  cmd_procremove(packet);
           case LMK_PROCPURGE:
                    cmd_procpurge();
           case LMK_GETKILLCNT:
                    kill_cnt = cmd_getkillcnt(packet);
........
}
```

8月5日

学习内容：

代码练习，完成了 LoginWaitingList 算法题

学习收获：

该题目比较常规，但是在 closeIDs 这个函数上很容易犯错，因为这个函数主要是通过给定前缀来查找满足条件的用户 id ，如果直接按照字符 id 匹配的话，一定会超时，这里我尝试了两种不同的方法：

1. 将每一个用户根据前缀进行分类，每次执行 closeIDs 时就通过 HashMap 定位到对应的前缀。
2. 将用户名进行字典排序，每次查找前缀是模拟一个 id 为前缀的假用户插入到字典排序中，根据 ASCII 码将符合前缀的用户项进行遍历，因为字典排序过了，所以遍历很快，接着将遍历到满足条件的用户进行 close 操作就算完成了。