1. Ext4文件系统检查及修复
   1. 分区挂载前，判断分区状态是否正常，若不正常，则使用fdisk+mkfs.重新格式化分区；
   2. 挂载前，使用fsck检查文件系统状态，异常则修复；
   3. 挂载失败后，使用mkfs格式化文件系统，然后重新挂载；
2. eMMC异常检测及复位机制
   1. 起用一个线程，定时检测eMMC在位状态及各分区的在位状态和读写权限；
   2. 因eMMC不支持热插拔，故将修复操作放至系统重启后进行；
   3. 异常处理方案：
      1. eMMC分区丢失或分区权限异常：
         1. 重启系统；
         2. 若分区异常，则会自动触发挂载前的检查修复机制；
      2. eMMC设备异常：
         1. 设置环境变量记录SD卡异常（uboot环境变量，保存在flash内）；
         2. 重启系统；
         3. 在uboot初始化阶段获取对应环境变量值，若需要复位，则在此处进行eMMC的软复位（复位eMMC控制器和device）+硬复位（复位device）；
3. 配置分区实时备份及异常恢复
   1. 备份机制： inotifywait+rsync
      1. 使用inotifywait监听配置分区的文件读写；
      2. 当检测到对应目录存在写入或删除类操作时，调用rsync工具，将其修改同步至备份目录；
   2. 异常恢复机制：
      1. 挂载配置分区时，使用fsck检测分区状态，存在问题则直接修复；
      2. 挂载分区失败时，使用mkfs格式化分区，并从备份目录恢复配置；
      3. 恢复配置时，设置标志位，避免出现恢复未完成，被中断的情况出现；
4. 客户现场问题处理
   1. 问题：

客户现场出现eMMC异常，且系统重启、断电无法恢复的问题，与eMMC厂商协同定位，确定是eMMC固件缺陷，eMMC误开启了一个工厂测试配置，该配置需要向eMMC发送一个vendor command关闭；

* 1. 处理难点：
     1. 从用户态根据eMMC协议模拟发送命令存在时序问题，功能无法调通，需要从内核态发送cmd；
     2. eMMC bus和mmc\_card句柄为局部变量，在external module中，无法直接获取相应句柄进行操作；
     3. 客户不同意升级固件，无法通过修改内核的eMMC初始化流程实现；
  2. 解决方案：
     1. 在external module中，注册一个fake eMMC device到eMMC bus，获取到eMMC bus句柄
     2. 调用bus\_find\_device\_by\_name，根据实际的eMMC设备文件名，查找到对应的mmc\_card句柄
     3. 调用mmc\_wait\_for\_cmd，发送指定的vendor command；

1. eMMC寿命预测及告警功能
   1. eMMC协议中的寿命统计是以10%为最小颗粒度，不够精确；
   2. 在mmc block层增加写入数据统计，统计eMMC累计写入数量，并将该数据定期写入flash中记录，避免掉电丢失
   3. 统计各个不同型号的在用的eMMC，每个寿命阶段，累计写入数据量
   4. 根据累计写入数据量，预估出eMMC剩余寿命
2. 指定目录的文件读写监测
   1. 为延长flash和eMMC使用寿命，需要监测并限制一些文件的高频读写操作；
   2. 在vfs层的文件读写接口（vfs\_write和generic\_file\_open）内，增加文件路径打印操作；
   3. 设置全局开关，控制文件路径打印接口打印；
   4. 使用位图的形式，不同bit对应不同路径的打印使能；
   5. 通过sysrq，设置全局开关各个bit的使能；
   6. 功能示例：
      1. 执行一次echo Z > /proc/sysrq-trigger，开启/etc目录的write监测
      2. 再次执行echo Z > /proc/sysrq-trigger，开启/etc目录的open监测
      3. 第三次执行echo Z > /proc/sysrq-trigger，关闭/etc目录的所有监测
3. 增加KDB魔术键功能，依次按下ESC、K、D、B四个键后，触发KDB
   1. 在串口挂死但未触发kernel panic之类的场景中，需要进入kdb定位问题；
   2. 系统内置的break键无响应；
   3. 在串口驱动的serial8250\_read\_char接口内，加入gdb按键检查操作，当按下ESC键（无特殊含义，因该按键使用较少，所以选用该按键）时，记录该次按键事件；
   4. 当按下ESC之后，若后续依次按下K、D、B三个键（每个键间隔不超过10s，超过则忽略该次ESC事件），则设置port->sysrq\_ch值为SYSRQ\_KDB\_OPT\_KEY\_G，触发串口驱动内自带的sysrq-kdb功能；
4. 从内核中剥离BIOS Flash驱动滞后注册流程
   1. BIOS的Flash默认连接到主机侧，使用CPLD可以将其硬件链路切换至BMC侧；
   2. 在BIOS FLash连接至BMC侧时，BMC将其作为一个普通的spi Flash，向其写入BIOS的固件，由此实现BIOS的带外固件升级；
   3. 由于BMC的Flash驱动在启动时已经初始化完成，而硬件链路切换是在BMC业务内实现的，因此需要将BIOS Flash的初始化流程滞后处理，仅在需要时主动触发BIOS Flash的初始化；
   4. 原来的实现是在spi驱动内创建一个字符设备，将spi device的注册接口添加到字符设备的文件回调函数中，驱动初始化时，只创建字符设备而不调用注册接口，检测对应的设备节点时记录节点地址并跳过，当实际硬件链路切换完成后，调用字符设备的对应接口，完成Flash的初始化；
   5. 该方案可以实现Flash的滞后注册，但是当存在多个版本内核时，需要在每个内核中都重新实现一遍，尤其版本差别较大时，一些驱动实现也存在差异，难以管理维护；
   6. 解决方案：
      1. 将该流程从内核中抽离出来，作为一个独立的external module维护；
      2. Dts中，将BIOS FLash对应的设别节点状态设置为disabled，取消其在内核启动时的初始化操作；
      3. 在external module中，调用of\_find\_node\_by\_name查找到对应的dts设备节点，调用spi\_busnum\_to\_master接口，获取实际的spi控制器句柄，调用of\_register\_spi\_device，将其注册到spi控制器，即实现设备的滞后初始化；
5. 代码多平台适配
   1. 公司新引入多款国产芯片，原有代码无法直接兼容（存在大量业务代码直接读写寄存器的神奇操作），需要进行多平台的适配；
   2. 原计划新增OSA、HAL、CONFIG以及BSP四个模块，将代码框架整体优化，后因人力原因，未拆分这几个模块，仅搭建了个简易版的平台适配层框架；
   3. 框架介绍：
      1. 用户态以动态库的形式，对外提供统一API接口；
      2. 内核态按功能划分platform、product、interrupt三个大模块，依次对应：
         1. 平台差异性功能的实现，如主备切换、CPLD读写等功能，主要是原业务代码中读写寄存器的操作，因过于分散，未对其细分模块，统一放至platform中；
         2. 产品初始化及定制化配置；
         3. GPIO中断模块的初始化及配置；
      3. Platform模块中，各个功能接口以回调函数的形式定义，在ioctl中根据传入的命令字，调用对应的回调函数；
      4. 每个平台有一份自己的源码，实现对应功能的回调，并定义一个初始化接口中将其注册到platform模块，编译时根据传入参数，仅编译指定平台的源文件；
      5. 初始化接口以弱引用的形式声明，实际调用参与编译的平台的初始化接口；