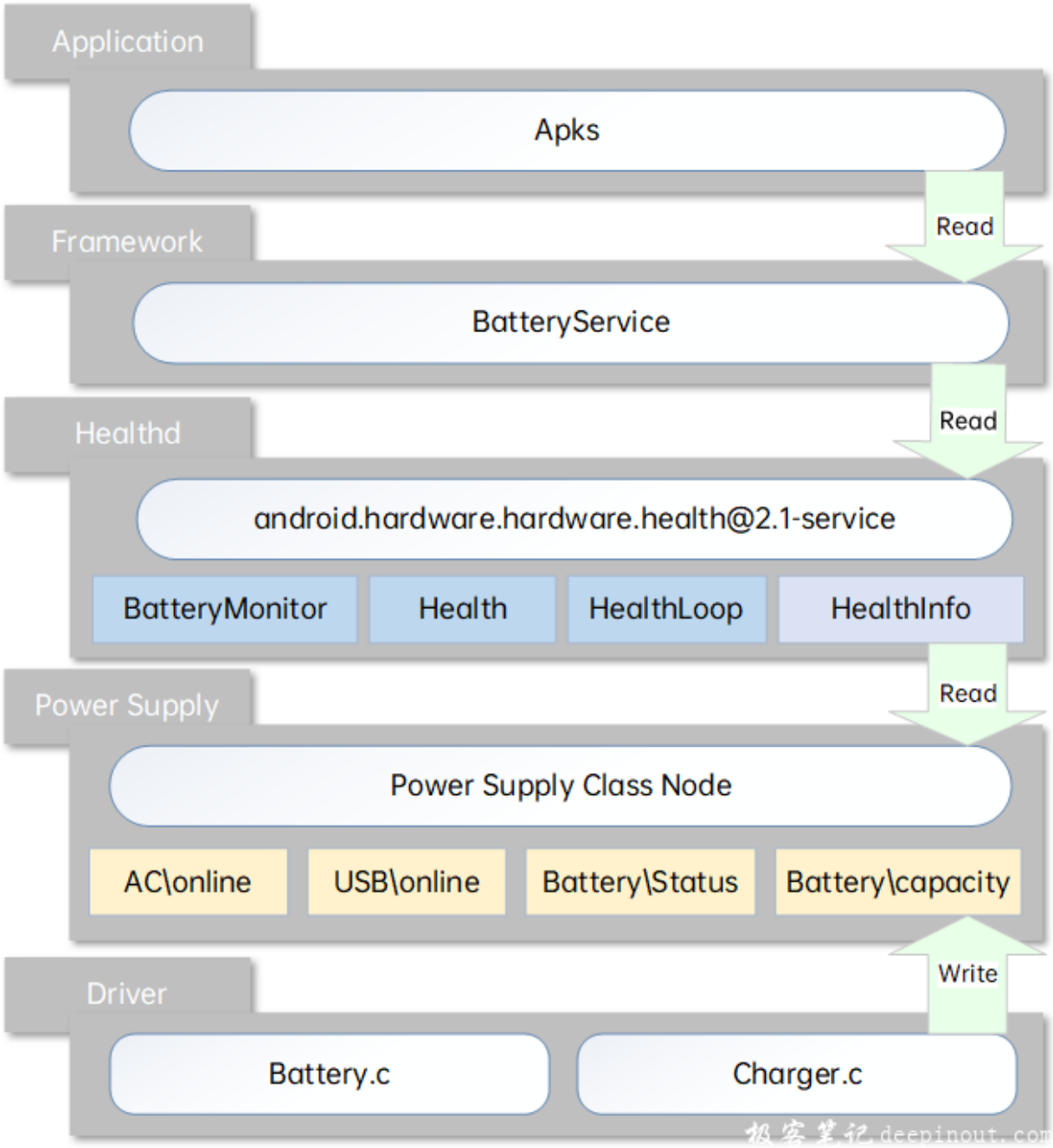
深入浅出理解Power Supply

作为一个内核初学者，经常容易进入“知其然但不知其所以然”的状态，在power supply子系统中就是这样，知道如何去添加一个属性prop，知道psy可以创建一堆文件节点，也知道上层是通过读取这些节点来获取供电信息的，但对于其中的细节，便知之甚少。最近深究其中，才逐步发现内核的奥妙所在。

Android供电系统框架



power supply(以下简称psy)是Linux中从供电驱动抽象出来的子系统，是Linux电源管理的重要组成部分。psy是一个中间层，在kernel中是属于设备驱动的一部分，psy的作用主要是向用户空间汇总各类供电的状态信息。抽象出来的各类信息称为property，比如供电设备是否连接就对应着POWER_SUPPLY_PROP_ONLINE。在驱动层，主要是两大模块，与电池相关的驱动和与充放电管理相关的驱动（对应图中的battery.c和charger.c）,这两大模块主要处理硬件相关的逻辑，在硬件状态发生变化

时，会触发相关的中断，驱动层会调用相应的中断函数，并更新修改相应的psy节点值。驱动负责更新psy节点的状态，HAL层会去读这些节点，驱动在检测硬件、传感器信息变化会去更新节点值，而HAL层什么时候会去读取这些节点值呢？以及其中调用的流程是怎样的？今天就简单介绍下。

power supply子系统简介

概述

psy子系统的基础是建立在设备驱动模型之上的，主要运用了其中的class、device、kobject、sysfs、uevent相关知识，也是驱动设备模型的一个具体应用。psy子系统中power_supply_class对应着系统中供电设备类，是一个抽象化的集合，对应着/sys/class/power_supply/目录，供电设备都在该目录之下，比如battery设备就对应该目录下一个子目录battery，而battery设备的一个属性则对应battery的一个文件节点，也对应着一个kobject。

相关结构体

psy相关的定义在/include/linux/power_supply.h。

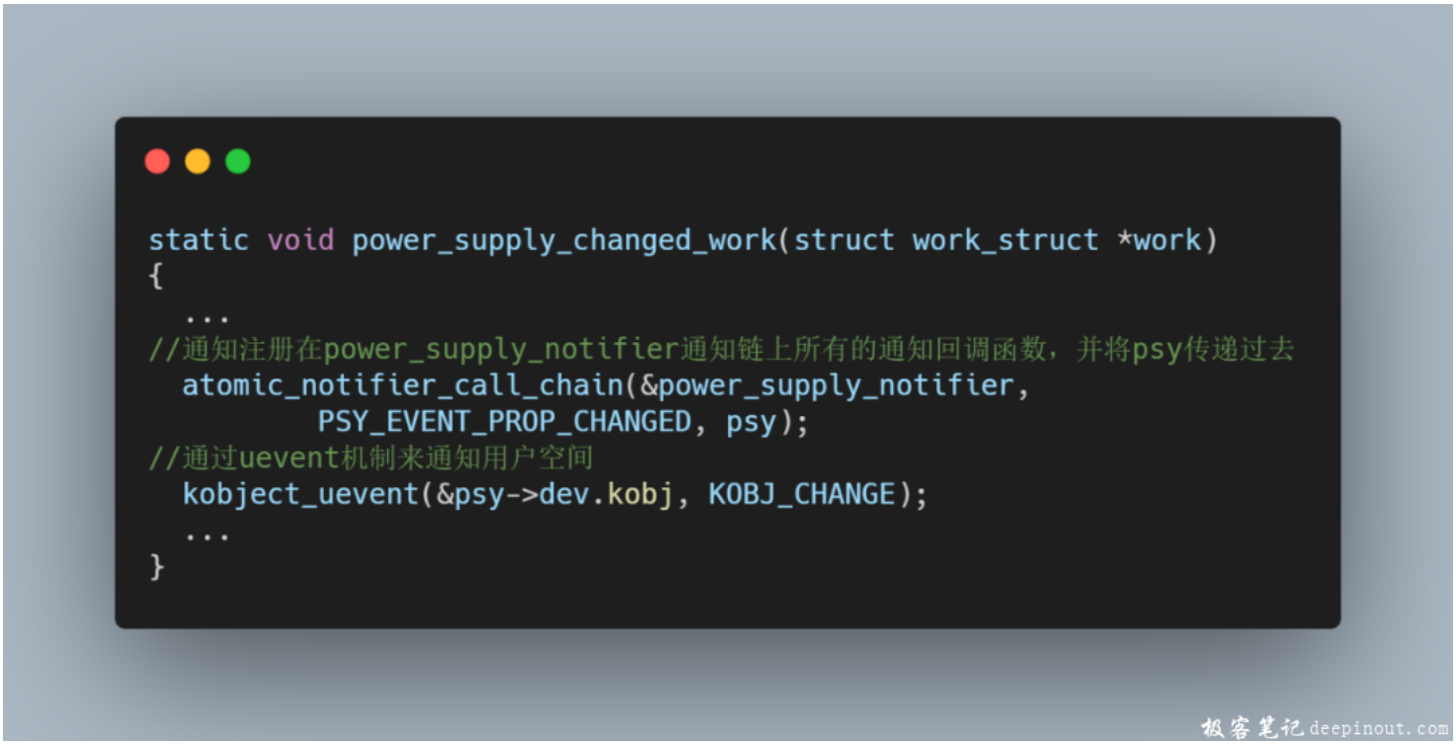
psy_desc是psy子系统中最重要的结构体，该描述符定义了psy的属性property，以及相关的set/get/is_writeable接口，is_writeable即文件节点的“w”权限，所有节点默认是可读的。从get函数可以看到调用该函数需要指定某个psy和psp（属性），结果保存在val中，值得一提的是，val是个union类型，可以传递int或char *。

struct power_supply表示一个psy供电设备，比如电池、AC、USB，一般可通过devm_power_supply_register函数注册成一个psy设备，在注册设备之前需要定义好该设备的psy_desc。

相关接口函数

相关的函数主要在psy_core.c和psy_sysfs.c中，core主要负责设备状态变化逻辑，sysfs主要负责文件节点相关逻辑。

最重要的是power_supply_changed，在驱动中检测到硬件状态发生变化，会通过该函数调度起psy中的changed_work。该工作队列负责发送notifier（内核内不同模块之间）和通过uevent进行change上报。



__power_supply_register负责注册一个psy设备：

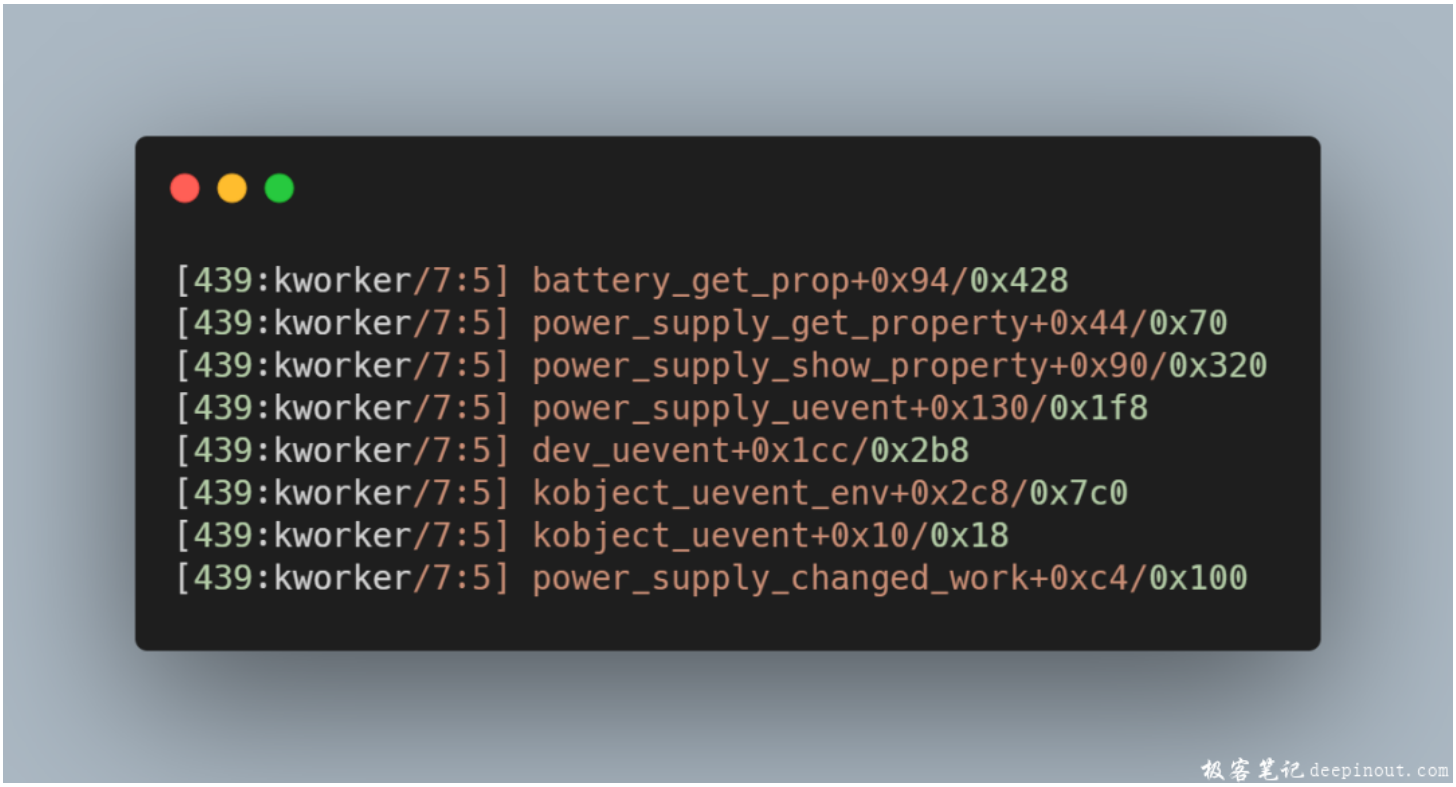
```c
//该函数用户注册psy，一般在设备驱动的probe流程中调用
//需要提前构建好desc、属性
static struct power_supply *__must_check
__power_supply_register(struct device *parent,
                const struct power_supply_desc *desc,
                const struct power_supply_config *cfg,
                bool ws)
{
//该函数主要是初始化dev和psy
  struct device *dev;
  struct power_supply *psy;

  psy = kzalloc(sizeof(*psy), GFP_KERNEL);
  if (!psy)
      return ERR_PTR(-ENOMEM);
  ...
  dev = &psy->dev;
//根据传入的参数初始化dev，其parent为传入的dev
  device_initialize(dev);
  dev->class = power_supply_class;
  dev->type = &power_supply_dev_type;
  dev->parent = parent;
  dev->release = power_supply_dev_release;
  dev_set_drvdata(dev, psy);
//注册desc
  psy->desc = desc;
//将设备名称设置为desc->name
  rc = dev_set_name(dev, "%s", desc->name);
  if (rc)
      goto dev_set_name_failed;
//初始化两个work，其中changed_work是重点
  INIT_WORK(&psy->changed_work, power_supply_changed_work);
  INIT_DELAYED_WORK(&psy->deferred_register_work,
            power_supply_deferred_register_work);
  ...
  spin_lock_init(&psy->changed_lock);
//在总线上添加初始化好的dev
  rc = device_add(dev);
//该delaywork用于等待父设备probe完成之后，调度一次changed_work
  queue_delayed_work(system_power_efficient_wq,
             &psy->deferred_register_work,
             POWER_SUPPLY_DEFERRED_REGISTER_TIME);
  ...
```

在sysfs中有个power_supply_uevent，该函数在psy class初始化时注册为设备节点的dev_uevent，在每一个psy目录下都有一个uevent节点，读取该节点即调用power_supply_uevent函数。该函数遍历当前设备下的所有属性并将结果保存在kobj_uevent_env中，结果以键值对的形式进行保存。


```
//用于保存键值对字符串
struct kobj_uevent_env {
  char *argv[3];
  char *envp[UEVENT_NUM_ENVP];//用于保存env的key
  int envp_idx;//用户访问envp
  char buf[UEVENT_BUFFER_SIZE];//保存env的value
  int buflen;//用户访问buf，指向buf下一个可用地址
}
```


调用流程

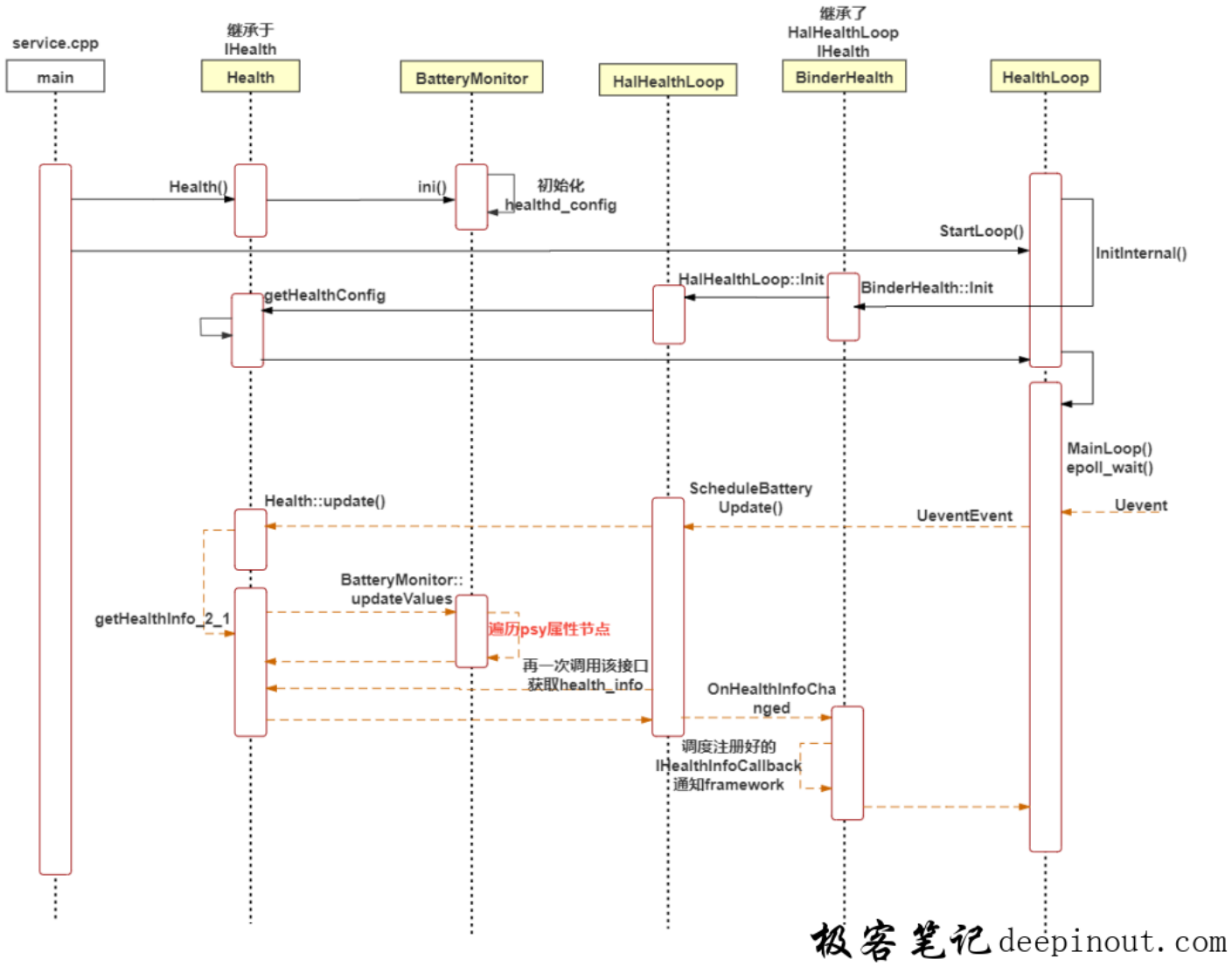
psy子系统主要调度的逻辑都在power_supply_changed_work中。跟踪这一调用流程可以在驱动中实现的get_property函数增加调用栈打印：



可以看到，在kobject_uevent_env函数中调用对应kset的uevent，会去遍历每一个属性节点（dev_uevent），之后，会通过netlink_broadcast函数进行广播（使用netlink机制），其中广播的字符串保存在sk_buff->data中，这一字符串以“action_string@devpath”进行拼接，其中action为kobject_action，而devpath则为该psy设备的设备路径。值得注意的是，使用uevent-netlink机制传递的字符串并不会包含psy属性节点的kobject_uevent_env键值对状态。

healthd简介

由于uevent机制仅将一个简单的字符串传递给了用户空间，而安卓系统建立在kernel之上，需要思考如何将设备属性的变化值及时更新到用户空间，于是就有了healthd服务，healthd目前已经更新到了2.1版本，其主要工作通过epoll_wait来监听kernel中的uevent事件。具体的函数调用流程图如下：



相关的代码路径主要是在：

/hardware/interfaces/health/；

/system/core/healthd/；

从service.cpp开始理一下调用流程，可以整理出上述的调用流程，黑色线条为初始化流程，红色线条为当psy-uevent上报后触发epoll之后的调用流程。与底层节点交互的逻辑都在BatteryMonitor中，在初始化过程中会初始化healthd_config结构体，用于保存psy属性节点的路径。

在监听循环MainLoop函数中，一个while(1)循环，调用epoll_wait()函数来监听uevent，收到事件之后会调用初始化时注册好的func（UeventEvent），该函数会通过uevent_kernel_multicast_recv接口去读取netlink发送的sk_buff->data，通过查找其中的字符串来判断事件是否为psy子系统发送的，如果不是的

话，不会进行处理。进一步的处理流程主要是调用到BatteryMonitor中的updateValues，在该函数中会遍历读取psy属性节点，存储在HealthInfo结构体中，之后通过BinderHealth中注册好的回调函数IHealthInfoCallback通知BatterySerice，具体的通知函数为BinderHealth：OnHealthInfoChanged。

Healthd是一个根植于powersupply子系统，并采用了epoll监听底层节点的uevent事件，之后轮询底层属性节点的守护进程。在安卓R版本中，Healthd相关代码重构为libhealthloop和libhealth2impl，但为了保证向后兼容，可以看到在ScheduleBatteryUpdate()函数中调用了两次updateValues，这样会遍历两次底层节点造成了冗余。另外在psy-uevent机制中，也有一次属性节点的遍历，一共三次遍历，这就要求底层驱动在更新属性值时，不能加入耗时的IO操作，否则会影响系统性能。

总结

power supply架构的精髓是极大化的发挥了uevent和sysfs的作用，简单高效地抽象出了与硬件无关的关键信息，通过notify机制使得其他内核模块可以及时获取相关事件；Healthd通过epoll监听psy创建的节点uevent，之后再去遍历读取结果，这样是为了避免与kernel的耦合。psy和Healthd比较适合新手学习，能提供一个由外向内的视角去解读kernel，也能加深对设备驱动模型的理解。熟悉之后可以进行相关的逻辑扩展，也可以进一步学习usb模块与psy子系统的关系，也可以进一步探究涉及的notify、netlink、epoll等机制。