BangIII 机型的接口 SDK 使用简要说明:

公共说明：

该 SDK 重要的工作对象是 Machine，ActuatorProxy, Command，Attention，Response。 大部份的逻辑和流程都是通事件监听和响应来完成的。因此，需要注意在事件监听方法中，不要做特别耗时的动作。 如果确实需要，可以从线程池中申请一个新的线程来执行。

所有命令相关的定义，都是在 CommandDef 类中，所有故障相关的定义，都是在 FaultDef 类中。 请使用这两个类中定义的常数，不要直接在代码中使用 “魔法字符”。

请一定要牢记，业务流程的处理和故障流程的处理是分开的。避免在业务流程中，去处理故障的系统应对流程。

1.初始化。

在使用 BangIII 机型时，需要先针对每一个 IPS 板（一个机器人小车），创建一个 Mechine 对象。然后可以使用该对象操作设备的所有功能和状态的维护。在 APP 启动的时候创建 Machine 对象，并需要一直持有直到整个 APP 的生命周期的结束。

a.创建对象

Machine machine = Machine.create("[设备 ID，10 个字符]", [context 对象], "[串口的端口字符串]");

b.定义监听设备状态变化的事件监听接口（这是故障处理流程）

```
        machine.setStateChangeListener(new
Machine.StateChangeListener() {
        @Override
        public void onFatal(FaultState state) {
          //在这里，设置当设备出现不能正常销售的软硬件故障和状态时，应对措施。
         //主要会是硬件故障、出货口门被阻档无法正常关闭、门被打开，没有正常关闭等情型。
        }


          @Override
          public void onError(FaultState state) {
              //在这里,设置当设备出现不影响正常销售的软硬件故障和状态时，应对措施。
              //主要是一些流程中的错误。如没有正常取到货，找不到货位等。
          }
```

```
        @Override
        public void onWarn(FaultState state) {
            //在这里，设置当设备出现可能会有影响的软硬件状态时，应对措施。
            //主要是系统忙，超时等现象。
        }
        @Override
        public void onRecover() {
            //当系统恢复正常后，会触发此事件。
        }
    });
```

Note:通过监听这些故障和状态的事件，可以及时在屏幕上做出处理，引导消费者操作。

c.定义监听串口通信状态的事件监听接口

```
    machine.setChannelMonitor(new IChannelMonitor() {
      @Override
```

```
      public void onAvailable() {
        //串口的状态从不可用变为可用时触发。
            }
       @Override
            public void onUnavailable() {
                //串口状态从可用变为不可用时触发。并且会以一定的时间隔持续触发，直到状态恢复正常。
                //此时，可以在收到事件，对状态进行持续的跟踪，并及时刷新状态，通知运维人员。
            }

       @Override
       public void onReceivedUnknownData(String errDesc, byte[] data) {
           //当串口接收到奇怪的数据，不合法的数据时会触发。
           //传服务器给开发人员，做进一步的分析。
       }
   });
```

Note: 一般，在这个接口里，并不需要做太多，更多是记录，上传服务器，通知运维人员。由于串口通信引起的故障，可以在心跳的事件时做出处理,就不需要两边同时处理了。

d.定义监听心跳数据的事件监听接口

```
        machine.setHeartBeatListener(new
Machine.HeartBeatListener() {
                @Override
                public void online(final byte[] data) {
                    //每一次的心跳数据被收到，都会触发。参数
data 是心跳数据包中所携带的常规高频监控的状态数据。如温度。
                }
                @Override
                public void offline(long startTime, final
String errMsg) {
                    //当没有收到心跳数据达到两个心跳周期时
（目前定义是一个心跳周期是 3s），会开始触发。
                    //并且会持续触发，直到接收到心跳数据。
                    //参数：startTime，是当前没有心跳数据的
开始时间戳，在持续触发时，这个数据不变。
```

```
                    //参数：errMsg，是没有心跳数据的可能原因：串口故障，IPS 无响应
                    //当没有心跳数据时，应停止销售，直到心跳数据恢复。
            }
        });
```

Note:此时，Machine 对象的初始化动作完成。

2.监听从 IPS 上发的命令或数据。需要定义该命令编号的监听器。（该步骤是可选项）

```
machine.setCommandListener([命令编号：byte], new IDataReceiveListener() {
    @Override
    public void onDataReceived(byte[] buffer) {
        //定义当接收到数据时，应完成的动作。
    } });
```

3.执行一个命令（这是业务流程）

a.创建一个命令，Command 对象。请尽量使用 Command.create 方法。该方法有两种型态。请选择合适的型态。 Command cmd = Command.crate([命令编号：byte], Command.TYPE_ASYNC|Command.TYPE_SYNC);

Command cmd = Command.crate([命令编号：byte], [命令参数，byte[]], Command.TYPE_ASYNC|Command.TYPE_SYNC);

注意，Command 有两种执行方法：Command.TYPE_ASYNC|Command.TYPE_SYNC，异步和同步。

异步，只能通过事件的监听来完成流程。同步则会等命令执行完毕后，才执行下一步。 你无法执行一个在 CommandDef 中没有定义的命令。

a. 定义命令的事件监听

```
cmd.setOnReplyListener(new ICallbackListener() {

    @Override

    public boolean callback(Command cmd) {

        //在命令的执行过程中，会根据不同的命令，将中间过程的状态和结果数据通过 Attention 上发。
```

```
            //如需要向 IPS 发送补弃命令，可以使用
command.setAttention([只需要通知的内容,不需要数据包头的定
义内容]);
            //具体定义的数据格试，请参考文档。
           return true;
       }
 });


cmd.setOnCompleteListener(new ICallbackListener() {
          @Override
          public boolean callback(Command cmd) {
              //在命令正常完成后，触发此事件。
              //你可以通过 Command 对象的 getResult() 方法，
获得最终返回的结果数据。该方法返回的是 Response 对象。
              return true;
          }
      });
```

```
    cmd.setOnErrorListener(new ICallbackListener() {

        @Override

        public boolean callback(Command cmd) {

            //在命令执行结束后，如果在过程中出现过异常，则会触发此事件

            //注意，流程状态和故障状态是分开的。此时，你依然应根据返回的数据结果来判断业务的状态。而不需要在此时处理

            //故障状态。所有的故障状态的处理，都可以通过 Machine 对象来集中维护。

            //你依然可以通过 Command 对象的 getResult() 方法，获得最终返回的结果数据。该方法返回的是 Response 对象。

            //但应该在这里记录和业务相关的异常信息，备查。

            return true;

        }

    });
```

c.设置命令的辅助描述。这个信息，都是在日志记录中，和此命令所有日志标记在一起，方便问题的查找。 cmd.setDesc([辅助信息，例如订单号等]);

d.开始执行命令

machine.executeCommand(cmd);

如果你是采用同步模式执行命令，则此时会阻塞，直到命令完成。所以，不要在主界面流程这么做，会卡住。除非命令执行的时间非常短。 同步执行命令，你既可以通过上述定义事件的方法来推动流程，也可以在执行完成后，再通过 getResult()方法来获得得结果数据，完成流程。

注意，Command 对象会自行处理执行超时的状态。当发生超时时，会触发 OnError 事件。你在命令执行完成后，通过 isError() 方法，getState()方法和 getErrorDescription() 方法来获得命令的执行状态和出错的信息。通过 getState()方法获得的是命令生命周期的状态，不是设备的运行状态。 所有状态的定义在 CommandState 类中。设备的运行状态是通过 getResult()方法获得 Response 对象。再通过 response.getErrorCode() 方法 来获得。

Command 是个有状态的对象。应避免多线程同时访问一个 Command 对象。同时，Command 对象在执行时，是不能重复调用的，必须等到上一次的执行完成。 因此，你应为每一个命令创建一个 Command 对象，而不是创建一个 Command，然后通过修改属性来共享。

4.你可以随时从 ActuatorProxy 对象中，获得新的常规线程或是定时器线程。注意，不能在定时器线程中运行会阻塞的任务或是很费时的任务。 否则会影响后续的定时任务，可能会产生不可预料的结果。

致此，你已基本可以完成工作了。请按照接口定义文档，完成业务流程的处理。

PS:

1. 详细使用情况,可参考项目 BmtVendingMachineDemo

2. 命令定义 CommandDef#IDXXX

```
public static final byte IDDEVICEINIT = 0x01; //设备初始化

public static final byte IDLOCATIONSCAN = 0x02; //扫描货道坐标列表（布局）

public static final byte IDSELLOUT = 0x04; //出货

public static final byte IDACCS_CONTORL = 0x0E; //电源板继电器控制

public static final byte ID_GET_MAC_PARAMETER = 0x23; //查询设备参数

public static final byte ID_GET_LOCATION_DATA = 0x24; //获得货道坐标数据
```

```
public static final byte ID_RW_PICK_OFFSET = 0x29;
//读写取货高度补偿值
public static final byte ID_RW_POP_VAL = 0x2A;          //
读写出货口高度
public static final byte ID_GO_TO_ORIGIN = 0x51;
//找零点位
```