设备 DP 数据解析

文档版本: 20240608

查看在线版本

目录

设备功能点（Data Point，简称 DP）是设备信息中描述一个设备功能的最小单元，每个 DP 都分为产品信息中的 DP 配置和设备信息中的 DP 值两部分。不同的 DP 类型可能还有拓展属性。在涂鸦体系中将 DP 在功能上划分为三种：**快捷开关**、**展示型 DP** 和 **操作型 DP**。本文介绍如何通过设备、群组模型，快速获取功能点，方便定制 App UI 展示与操作。

1 操作步骤

1. 打开涂鸦 Developer Platform **产品开发** 页面。
2. 选择一个产品，单击 **操作** 栏的 **继续开发**。
3. 选择第四步 **产品配置** > **快捷开关配置**，并单击 **设置**。

您可以配置 **快捷开关** 和 **常用功能** 等功能点，并在 App 设备、群组卡片上体现。

2 获取 DP 解析模块

```
1 val plugin = PluginManager.service(IAppDpParserPlugin::class.java)
```

2.1 接口说明

解析模块内部不会去监听设备（群组）的属性变化。如果需要使用该模块，请至少调用一次 `update` 后才能通过 API 去获取缓存的 DP 模型。DP 模型 `IDeviceDpParser` 对象包含当前设备（群组）下所有 DP 的信息。

推荐在以下时机调用 `update` 方法进行缓存更新：

- 请求到设备列表时。
- 监听到 DP 数据变化时。

```
1  /**
2  * 更新群组信息
3  * @param groupBean 用于更新的群组模型数据
4  * @return 解析之后的群组 DP 模型
5  */
6  fun update(groupBean: GroupBean): IDeviceDpParser
7  /**
8  * 更新设备信息
9  * @param deviceBean 用于更新的设备模型数据
10 * @return 解析之后的设备 DP 模型
11 */
12 fun update(deviceBean: DeviceBean): IDeviceDpParser
```

调用 `update` 方法后，插件内部会缓存该 DP 模型的信息，以便其他业务取用。

```
1 /**
2 * 获取已解析的 DP 模型数据
3 * @param id 设备模型的 [DeviceBean.getDevId] 或者群组模型的 [GroupBean.
    getId]
4 * @return 存在缓存则返回缓存数据，否则返回 null
5 */
6 fun getParser(id: String): IDeviceDpParser?
```

业务上不再需要设备或者群组时，要手动调用删除方法，来清理已有的 DP 模型缓存数据。

```kotlin
/**
 * 移除 ID 对应的缓存
 * @param id 设备模型的 [DeviceBean.getDevId] 或者群组模型的 [GroupBean.
    getId]
 */
fun remove(id: String)
/**
 * 移除列表中 ID 对应的所有缓存
 * @param ids 所有要移除的缓存 ID
 */
fun removeAll(ids: List<String>)
```

3 设备 DP 模型

```
interface IDeviceDpParser {
    /**
     * 展示型 DP
     * @return 返回当前设备（群组）解析后的所有展示型 DP，如果产品未配置展示型 DP 则返回空列表
     */
    fun getDisplayDp(): List<IDpParser<Any>>
    /**
     * 操作型 DP
     * @return 返回当前设备（群组）解析后的所有操作型 DP，如果产品未配置操作型 DP 则返回空列表
     */
    fun getOperableDp(): List<IDpParser<Any>>
    /**
     * 当前设备的所有 DP ，只支持部分方法
     * 支持调用的方法：[IDpParser.getDpId], [IDpParser.getSchema], [IDpParser.getStandardSchema], [IDpParser.getValue]
     *
     * 如果当前设备（群组）中不存在此 DP 的 value，[IDpParser.getValue] 方法会返回 [NoValue]
     *
     * 调用上面未列出的接口方法会抛出异常 [UnsupportedOperationException]
     * @return 返回当前设备的产品信息中配置的所有 DP
     */
    fun getAllDp(): List<IDpParser<Any>>
    /**
     * 快捷开关
     * @return 当前设备的快捷开关，如果没有配置则返回 null
     */
    fun getSwitchDp(): ISwitch?
}
```

3.1 普通 DP：IDpParser

普通类型的 DP 指的是一个个单一的功能，例如：开关、温度、亮度等。在涂鸦 Developer Platform 上，可以配置能否控制、能否展示。大多数的 DP 值并不能直接被展示出来，需要拼接单位或者将值转换成为对应的可展示内容。

```kotlin
interface IDpParser<T> {
    /**
     * 当前 DP 的 iconfont 值，可能为空
     */
    fun getIconFont(): String?
    /**
     * 当前 DP 的名称
     */
    fun getDisplayTitle(): String
    /**
     * 当前 DP 在快捷操作区域显示的内容（操作型）
     */
    fun getDisplayStatusForQuickOp(): String
    /**
     * 当前 DP 在状态区域显示的内容（展示型）
     */
    fun getDisplayStatus(): String
    /**
     * DP 的 ID
     */
    fun getDpId(): String
    /**
     * UI 展示样式：percent、 percent1、countdown、countdown1 等
     */
    fun getDpShowType(): String
    /**
     * DP 数据类型：bool、enum、num、string
     */
    fun getType(): String
    /**
     * DP 原始配置信息
     */
    fun getSchema(): SchemaBean
    /**
     * DP 原始配置信息
     */
    fun getStandardSchema(): FunctionSchemaBean?
    /**
     * 生成用于下发给设备的指令
     * @param status 想要发送给设备的值，必须是当前 DP 类型的值
     */
    fun getCommands(status: T): String
    /**
     * 当前的 DP 值
     */
    fun getValue(): T
}
```

根据不同的类型，普通 DP 有如下派生接口：

- IBoolDp
- IEnumDp
- ILightDp
- INumDp
- IStringDp

下面分别介绍这几种类型的特有能力。

3.1.1 布尔类型：IBoolDp

开关类型，状态只有 `true` 和 `false`。

```
interface IBoolDp : IDpParser<Boolean> {
    /**
     * 返回对应状态的显示文案，该文案是配置在产品信息中的。常见的例如中文环境下：true => "开", false => "关"
     */
    fun getDisplayMessageForStatus(value: Boolean): String
}
```

3.1.2 枚举类型：IEnumDp

下发给设备的状态是 `getRange()` 方法返回的选项之一。

```
interface IEnumDp : IDpParser<String> {
    /**
     * DP 的取值范围
     */
    fun getRange(): List<String>
    /**
     * DP 值对应的显示文案
     */
    fun getNames(): List<String>
}
```

3.1.3 数值类型：INumDp

在平台上配置数值类型 DP 时，可以设置不同的取值范围、步进值和倍率。

1. 取值范围：当前 DP 允许的取值区间，下发给设备的值必须在该范围之内。

2. 步进：DP 变化的最小值。
3. 倍率：UI 展示时使用倍率对 DP 值进行缩放，展示用值是 DP 值除以 10 的指数倍。例如，设置一个温度 DP，取值范围是 [-100,100]。如果倍率是 1，那么展示的值范围就是 [-10.0,10.0]。如果倍率是 2，那么展示值范围就是 [-1.0,1.0]。

展示文案会根据 DP 的 **展示类型** 来决定，不同类型计算规则不一样。

```
interface INumDp : IDpParser<Int> {
    /**
     * 最小值
     */
    fun getMin(): Int
    /**
     * 最大值
     */
    fun getMax(): Int
    /**
     * 步进
     */
    fun getStep(): Int
    /**
     * 缩放倍率
     */
    fun getScale(): Int
    /**
     * 单位
     */
    fun getUnit(): String
}
```

3.1.4 光源类型：ILightDp

```
interface ILightDp : IDpParser<String> {
    companion object {
        //老彩灯 dpCode，14 位 HSV 颜色值，标准 dpCode 为 colour_data
        const val STRING_STANDARD_CODE_COLOUR_DATA_V1 = "colour_data
"
        //新彩灯 dpCode，12 位 HSV 颜色值，标准 dpCode 为
            colour_data_v2
        const val STRING_STANDARD_CODE_COLOUR_DATA_V2 = "colour_data
_v2"
    }
    //灯光颜色，从最小值到最大值，包含之间的所有数据
    fun getLightColorRange(): IntArray
    //当前颜色数据对应的描述，例如：'红色'、'绿色'、'蓝色'等等。如果无
        法获取描述，对应多语言会返回对应的 key 值，所有
key 值均是以 'color_' 开头，例如 'color_red'、'color_indigo'、'
    color_cyan_green
'。
    fun getLightColorDesc(context: Context): String
    //最小值：[0] 饱和度，[1] 亮度
    fun getColorMin(): IntArray
    //最大值：[0] 饱和度，[1] 亮度
    fun getColorMax(): IntArray
    //当前彩灯数据转换后的值：[0] 颜色值，[1] 饱和度，[2] 亮度
    fun getColorCurrent(): FloatArray
    //是否是 v2 类型数据，不同版本数据解析规则不一样
    fun isNewColorData(): Boolean
    //将颜色转换为字符串用于下发
    fun getStringColorHSV(hsvPoint: FloatArray, hsvValue: IntArray):
 String
}
```

3.1.5 字符串类型：IStringDp

字符串类型 DP 没有额外属性。

```
interface IStringDp : IDpParser<String>
```

3.2 快捷开关：ISwitch

快捷开关是由一个或多个 DP 组成，通过一个开关同时控制多个 `bool` 型 DP。

DP 配置时可以选择 **只上报、只下发、上报下发** 三种类型。

- **只上报**：不接收控制命令，但是 DP 值变化时会上报该状态。

- **只下发**：接受控制指令并做出相应的改变，但是不会上报对应的状态，涂鸦无从得知该 DP 的实际状态。
- **上报下发**：接受控制指令并做出相应的改变，设备收到后变更对应的状态，然后上报。

快捷开关的作用是控制设备，不接收设备控制的 **只上报** 类型就不在考虑范围内。快捷开关就分为两种：

- **有状态**：即普通类型，可以下发开关状态给设备，同时设备有状态变化也会上报。
- **无状态**：即只下发，设备并不会上报状态，无从得知开关的真实状态，因此每次下发给设备的状态是对当前状态取反。

> 快捷开关由多个 DP 组成时，只要其中有 1 个 DP 为 **只下发** 类型，那么该开关就会被解析成 **无状态** 类型。

```
interface ISwitch {
    companion object {
        const val SWITCH_TYPE_NORMAL = -1
        const val SWITCH_TYPE_WRITE_ONLY = -2
        const val SWITCH_STATUS_ON = 1
        const val SWITCH_STATUS_OFF = 2
    }
    /**
     * 当前快捷开关的所有 DP ID
     */
    fun getSwitchDpIds(): List<String>
    /**
     * 快捷开关此时的状态，如果快捷开关是由多个 DP 组成，只要其中任意一个为 true，那么返回 true。只有全部 DP 值
都是 false，才会返回 false。
     */
    fun getValue(dpId: String): Boolean
    /**
     * 快捷开关类型：普通类型 [SWITCH_TYPE_NORMAL]
     *                          只下发 [SWITCH_TYPE_WRITE_ONLY]
     */
    fun getSwitchType(): Int
    /**
     * 快捷开关状态：[SWITCH_STATUS_ON]、[SWITCH_STATUS_OFF]
     */
    fun getSwitchStatus(): Int
    /**
     * 生成用于下发给设备的指令
     * @param obj 参数类型为 bool，只下发类型不会使用该参数
     */
    fun getCommands(obj: Any): String
}
```