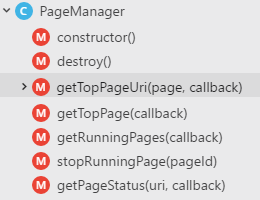
# PageIPC调用源码分析

Alios App获取系统已经安装或者正在运行的Page或者Server可以通过PageManager或者

AppManager对外接口：



以getTopPage(callback)为例，跟踪调用流程

framework\npm\caf2\src\page\PageManager.js

getTopPage(callback) {

this.page.instance.getTopPage(callback);

}

->

framework\npm\pageapi\PageImpl.js

getRunningPages(callback) {

pipc.dpms.getRunningPages(callback);

}

这里pipc是framework\npm\pageapi\internal\PageIPC.js，先看看dpms是什么

get dpms() {

return Impl.getImpl(this).firstDpmsProxy;

}

Impl是PageIPC.js中的内部类；

class Impl {

static getImpl(ipc) {

return pu.getStoredObject(ipc, Impl);

}

……

}

上面代码中pu是framework\npm\pageapi\internal\PageUtil.js

static getStoredObject(obj, ImplClass) {

……

map[realId] = new ImplClass(obj);

……

return map[realId];

}

故实际上Impl. getImpl(ipc)返回的是new Impl(ipc);PageUtils只是将Impl实例缓存起来了（PageUtils 中的gStoredObjects 静态map），如果已经实例化的直接在缓存中取。

下面继续看Impl.getImpl(this).firstDpmsProxy;中的firstDpmsProxy；

get firstDpmsProxy() {

if (this.\_dpmsProxy === undefined) {

this.\_dpmsProxy = new DPMSProxy(this);

}

return this.\_dpmsProxy;

}

返回DPMSProxy实例，DPMSProxy是PageIPC.js中的内部类；下面看DPMSProxy中的getRunningPages；

getRunningPages(callback) {

var msg = this.\_interface.createMethodCallMessage("getRunningPages");

this.\_interface.sendMethodCallMessage(msg, (err, res) => {

var pageList = [];

if (!err && res) {

var size = res.readInt32();

for (var i = 0; i < size; i++) {

let pageInfo = JSON.parse(res.readString());

pageList.push(pageInfo);

}

res.destroy();

}

this.callbackIfValid(callback, err, pageList);

}, this.\_timeout, "m");

}

如上\_interface是ubus通信接口，通过ubus通信调用目标进程getRunningPages方法并回调(err, res)=>{},最终将返回结果解析并回调给callback。

为了了解PageIPC通过ubus和哪个进程在通信，我们继续跟ubus相关代码，下面是DPMSProxy构造方法中\_interface的初始化：

constructor(ipcImpl) {

……

this.\_ubus = new Ubus();

this.\_interface = this.\_ubus.createInterface("dpms.pagemanager.yunos.com",

"/dpms/pagemanager/yunos/com",

"dpms.pagemanager.yunos.com", false, false);

}

这里通过ubus创建一个ubus实例，这个实例通过参数与具体的进程建立连接，通过查找dpms.pagemanager.yunos.com，找到这几个参数定义在framework\nativeservice\pagemgrd\dpms\src\ipc\MessageAdaptor.cpp

const String DYNAMIC\_PAGE\_MANAGER\_DBUS\_SERVICE\_NAME("dpms.pagemanager.yunos.com");

const String DYNAMIC\_PAGE\_MANAGER\_DBUS\_INTERFACE\_NAME("dpms.pagemanager.yunos.com");

const String DYNAMIC\_PAGE\_MANAGER\_DBUS\_OBJECT\_PATH("/dpms/pagemanager/yunos/com");

然后我们看ubus.js中 createInterface方法：

framework\npm\ubus\lib\ubus.js

/\*

\* @param {String} serviceName name of Service.

\* @param {String} path path of ServiceObject.

\* @param {String} interfaceName name of ServiceInterface.

\* @param {Boolean} reverse whether to create a reverse channel after connected to service interface.

\* @param {Boolean} awareLifecycle whether to enable death notificat, defauts to true

\* @return {Interface}

\*/

UBus.prototype.createInterface = function(serviceName, path, interfaceName, reverse,

awareLifecycle, shared) {

……

iface.\_iface = \_ubus.getInterface(serviceName, path, interfaceName, this.type,

reverse, awareLifecycle);

……

}

可以看到最终是调用\_ubus. getInterface，这里的\_ubus初始化如下

try {

\_ubus = require("node\_ubus.node");

} catch (e) {

log.E(TAG, e);

\_ubus = process.binding("ubus");

}

通过查找node\_ubus.node得知其为编译出的二进制文件，而打包的地方在framework\npm\ubus\src\ubus.cc

#if defined(NODE\_ADDON\_BUILTIN)

NODE\_MODULE\_CONTEXT\_AWARE\_BUILTIN(ubus, init);

#else

NODE\_MODULE(node\_ubus, init);

#endif

Ubus.cc中定义了ubus的接口方法，如下是我们这里需要关注的几个方法

NODE\_SET\_METHOD(target, "getInterface", GetInterface);

NODE\_SET\_METHOD(target, "createMethodCallMessage", CreateMethodCallMessage);

NODE\_SET\_METHOD(target, "sendMethodCallMessage", SendMethodCallMessage);

GetInterface方法的具体实现在framework\npm\ubus\src\bus\_proxy.cc

CreateMethodCallMessage方法的具体实现在framework\npm\ubus\src\bus\_message.cc

SendMethodCallMessage方法的具体实现在framework\npm\ubus\src\bus\_proxy.cc

framework\npm\ubus\src\bus\_proxy.cc 中GetInterface创建BusProxy

void GetInterface(const FunctionCallbackInfo<Value>& args) {

……

SharedPtr<BusProxy> bus\_proxy = BusProxy::createProxy(service\_name, object\_path, interface\_name,

strcmp(type, "kdbus") == 0 ? true : false, args[4]->BooleanValue(),

args[5]->BooleanValue(), env);

……

}

framework\npm\ubus\src\bus\_message.cc中CreateMethodCallMessage创建一个DMessage

void CreateMethodCallMessage(const FunctionCallbackInfo<Value>& args) {

……

SharedPtr<DMessage>\* msg =

new SharedPtr<DMessage>((\*proxy)->obtainMethodCallMessage(String(method)));

……

}

framework\npm\ubus\src\bus\_proxy.cc中SendMethodCallMessage通过创建的BusProxy实例发送DMessage。

void SendMethodCallMessage(const FunctionCallbackInfo<Value>& args) {

……

bool ret = (\*proxy)->sendMethodCallMessage(\*msg, callback\_data, timeout);

……

}

Next：

framework\npm\ubus\src\bus\_proxy.cc

bool BusProxy::sendMethodCallMessage(

SharedPtr<DMessage> msg, MethodCallbackData\* callback\_data, int timeout) {

return sendMessageWithReply(msg, ReplyHandler(handleMethodCall, callback\_data), timeout);

}

framework\libs\base\dbus\DProxy.cpp

bool DProxy::sendMessageWithReply(const SharedPtr<DMessage>& msg,

const ReplyHandler& handler, int timeout) {

return mConn->sendMessageWithReply(msg, handler, timeout);

}

Next：

framework\libs\base\dbus\KDBusConnection.cpp

void KDBusConnection::sendMessageWithReply(……){

……

int r = sd\_bus\_call\_async(mConn, NULL, msg,

KDMessagePrivate::methodCallback, call, timeout);

……

}

sd\_bus\_call\_async函数会调到三方ubus源码，alios\_lite具体路径如下

third\_party/libkdbus/libkdbus/sd-bus/sd-bus.c

\_public\_ int sd\_bus\_call\_async(……){

……

\_public\_ int sd\_bus\_call\_async(

……

}

static int bus\_send\_internal(……){

……

r = bus\_write\_message(bus, m, hint\_sync\_call, &idx);

……

}

static int bus\_write\_message(sd\_bus \*bus, sd\_bus\_message \*m, bool hint\_sync\_call, size\_t \*idx) {

……

if (bus->is\_kernel)

r = bus\_kernel\_write\_message(bus, m, hint\_sync\_call);//通过kernel写数据

else

r = bus\_socket\_write\_message(bus, m, idx);//通过socket写数据传递消息

……

}

third\_party/libkdbus/ sd-bus/bus-socket.c

int bus\_socket\_write\_message(sd\_bus \*bus, sd\_bus\_message \*m, size\_t \*idx) {

……

}

Dbus具体如何通信需要后面系统学习后才能弄清楚，大致可以理解这里bus\_socket\_write\_message写消息到KDbus设备，然后由kdbus将信息发到总线，目标进程首先要注册到dbus，然后就可以收到总线的消息，

框架代码跟到这里基本就到了Page客户端通过PageManager调用getTopPage的末端了，getTopPage的消息会通过ubus发送给对应的服务进程，即我们前面说的framework\nativeservice\pagemgrd\dpms\src\ipc\MessageAdaptor.cpp，MessageAdaptor.cpp继承于DAdaptor

class MessageAdaptor : public DAdaptor

系统中的本地服务的IPC通信都是通过实现DAdaptor来接IPC调用消息。比如我们这里调用getRunningPages，ubus会通过调用者设置的serviceName，path，interfaceName参数定位到具体接收消息的Adapter，并回调其handleMethodCall函数，Page相关的IPC调用对应的即为pagemgrd 中的MessageAdaptor.cpp中的handleMethodCall函数。

我们看MessageAdaptor.cpp的handleMethodCall方法

framework\nativeservice\pagemgrd\dpms\src\ipc\MessageAdaptor.cpp

bool MessageAdaptor::handleMethodCall(const SharedPtr<DMessage>& msg) {

……

DBUS\_ON\_MSG\_VOID\_REPLY("getRunningPages", onGetRunningPages)

……

}

getRunningPages 映射到函数onGetRunningPages

void MessageAdaptor::onGetRunningPages(const SharedPtr<DMessage>& reply) {

handler()->getRunningPages(reply);

}

这里的handler()在对应的.h文件中定义MessageAdaptor.h

SharedPtr<DynamicPageManagerService> handler();

故handler()是DynamicPageManagerService实例

framework\nativeservice\pagemgrd\dpms\src\DynamicPageManagerService.cpp

void DynamicPageManagerService::getRunningPages(const SharedPtr<DMessage>& reply) {

std::list<DomainRecord\*> domainList;

dpmsFTY->getDomainManager()->getRunningDomains(&domainList, USER\_CURRENT);

std::list<PageRecord\*> pageList;

for (DomainRecord\* domain : domainList) {

pageList.splice(pageList.end(), domain->getPageList());

}

// skip not ready pages

pageList.remove\_if([](PageRecord\* page){ return !page->isReady(); });

LOG\_D("%s size: %d", \_\_FUNCTION\_\_, pageList.size());

// total size

//这里写数据对应PageIPC中getRunningPages的回调的读数据

reply->writeInt32(pageList.size());

for (PageRecord\* page : pageList) {

page->toMessage(reply);

}

}

上面就是本地服务真正实现getRunningPages的地方，通过代码可以看出其逻辑是先getRunningDomains获取正在运行的Domains（类似Android中的Package），然遍历Domain中的Page，最后将所有这些Page返回。

看到DynamicPageManagerService中的PageRecord和DomainRecord是不是很熟悉的赶脚，自然联想到Android中的ActivityRecord，TaskRecord，ProcessRecord，详细的分析留待后续输出。