

Android 的底层库和程序

Android 的底层库和程序

- ❑ 第一部分 底层库和程序的结构
- ❑ 第二部分 标准 C/C++ 库 bionic
- ❑ 第三部分 C 语言底层库 libcutils
- ❑ 第四部分 Init 进程
- ❑ 第五部分 Shell 工具
- ❑ 第六部分 C++ 工具库 libutils
- ❑ 第七部分 Android 的系统进程

第一部分 软件的结构

1.1 本地实现底层的结构

1.2 增加本地程序和库的方法

1.1 本地实现底层的结构

Android 的本地实现层次具有基本的库和程序。这些库和程序是 **Android** 基本系统运行的基础。

主要包含了以下的内容：

- ❑ C 语言底层库 **libcutils**
- ❑ **Init** 进程
- ❑ **Shell** 工具
- ❑ C++ 工具库 **libutils**

1.2 增加本地程序和库的方法

Android 中增加本地的程序或者库，这些程序和库与它们所在的路径没有关系，只和它们的 **Android.mk** 文件有关系。

Android.mk 具有统一的写法，主要包含了一些系统公共的宏。

选项参考以下文件：

[build/core/config.mk](#)

默认的值在以下文件中定义：

[build/core/base_rules.mk](#)

在一个 **Android.mk** 中也可以生成多个可执行程序、动态库或者静态库。

1.2 增加本地程序和库的方法

可执行程序的 **Android.mk** : 

```
# Test Exe

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
    main.c
LOCAL_MODULE:= test_exe

#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_EXECUTABLE)
```

1.2 增加本地程序和库的方法

静态库（归档文件）的 `Android.mk` :

```
# Test Static lib

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
    helloworld.c

LOCAL_MODULE:= libtest_static

#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_STATIC_LIBRARY)
```

1.2 增加本地程序和库的方法

动态库（共享库）的 `Android.mk`：

```
# Test shared lib

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
    helloworld.c

LOCAL_MODULE:= libtest_shared

TARGET_PRELINK_MODULE := false

#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=

include $(BUILD_SHARED_LIBRARY)
```


1.2 增加本地程序和库的方法

可执行程序、动态库和静态库生成的表分别在以下的文件夹中：

[out/target/product/generic/obj/EXECUTABLE](#)

[out/target/product/generic/obj/STATIC_LIBRARY](#)

[out/target/product/generic/obj/SHARED_LIBRARY](#)

其目标的文件夹分别为：

XXX_intermediates

XXX_shared_intermediates


XXX_static_intermediates

对于可执行程序 and 动态库，生成的 **LINK** 子目录中的包含带有符号的库（没有经过 **strip**）。

1.2 增加本地程序和库的方法

编译模板的区别如下所示:

目标的模板: 可执行程序, 动态库, 静态库

```
include $(BUILD_EXECUTABLE)  
include $(BUILD_SHARED_LIBRARY)   
include $(BUILD_STATIC_LIBRARY)
```

如果编译主机的: 可执行程序, 动态库, 静态库

```
include $(BUILD_HOST_EXECUTABLE)   
include $(BUILD_HOST_SHARED_LIBRARY)  
include $(BUILD_HOST_STATIC_LIBRARY)
```

1.2 增加本地程序和库的方法

安装路径的问题

LOCAL_MODULE_PATH 和 LOCAL_UNSTRIPPED_PATH

增加以下可以安装到不同的文件系统:

LOCAL_MODULE_PATH := \$(TARGET_ROOT_OUT)

LOCAL_UNSTRIPPED_PATH := \$(TARGET_ROOT_OUT_UNSTRIPPED)

文件系统的选择:

TARGET_ROOT_OUT :

表示根文件系统 out/target/product/generic/root



TARGET_OUT :

表示 system 文件系统 out/target/product/generic/system

TARGET_OUT_DATA :

表示 data 文件系统 out/target/product/generic/data

1.2 增加本地程序和库的方法

进行安装工作的 **Android.mk** :

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
copy_from := \
A.txt \
B.txt
copy_to := $(addprefix $(TARGET_OUT)/txt/, $(copy_from))
$(copy_to) : PRIVATE_MODULE := txt
$(copy_to) : $(TARGET_OUT)/txt/% : $(LOCAL_PATH)/% | $(ACP)
$(transform-prebuilt-to-target)
ALL_PREBUILT = $(copy_to)
# create some directories
DIRS := $(addprefix $(TARGET_OUT)/, \
txt \

$(DIRS):
@echo Directory: $@
@mkdir -p $@
```

1. 创建路径: [system/txt](#)
2. 在其中安装: **A.txt** 和 **B.txt**

第二部分 标准 C/C++ 库 bionic

bionic 提供 C/C++ 标准库的功能，它是一个专为嵌入式系统设计的轻量级标准库实现。

bionic 的源码和头文件在以下的目录中：

[bionic/](#)

相对传统的标准库实现，如 **glibc**，**bionic** 的体积和内存占用更小。**bionic** 支持标准 C/C++ 库的绝大部分功能，支持数学库，以及 **NPTL** 线程库。它还实现了自己的 **Linker** 以及 **Loader**，用于动态库的创建和加载。

bionic 加入了一些 **Android** 独有的功能，比如 **log** 的底层支持。另外它还实现了一套 **property** 系统，这是整个 **Android** 的全局变量的存储区域，**bionic** 使用共享内存的方式来实现维护 **property** 系统。

第三部分 C 语言底层库 libcutils

C 语言底层库提供了 C 语言中最基本的工具功能。这是 **Android** 本地中最为基础的库，基本上 **Android** 中所有的本地的库和程序都连接了这个库。

头文件的路径：

[system/core/include/cutils](#)

库的路径

[system/core/libcutils](#)

编译的结果是： **libcutils.so**

第三部分 C 语言底层库 libcutils

libcutil 中主要的头文件:

threads.h : 线程

sockets.h : Android 的套接字

properties.h : Android 的属性

log.h : log 信息

array.h : 数组

ashmem.h : 匿名共享内存

atomic.h : 原子操作

mq.h : 消息队列

第四部分 Init 进程

Android 启动后，系统执行的第一个进程是一个名称为 **init** 的可执行程序。提供了以下的功能：

- ❑ 设备管理
- ❑ 解析启动脚本
- ❑ 执行基本的功能
- ❑ 启动各种服务

代码的路径：

[system/core/init](#)

编译的结果是一个可执行文件：**init**

启动脚本的路径：

[system/core/rootdir/init.rc](#)

第四部分 Init 进程

init 的可执行文件是系统运行的第一个用户空间的程序，它以守护进程的方式运行。

```
int main(int argc, char **argv)
{
    /* ..... */
    umask(0);
    /* 创建文件系统的基本目录 */
    open_devnull_stdio();          /* 打开 3 个文件 */
    log_init();                    /* 初始化 log */
    parse_config_file("/init.rc"); /* 处理初始化脚本 */
    /* 获取比内核命令行参数 */
    qemu_init();
    import_kernel_cmdline(0);
    /* 初始化驱动设备，创建文件系统节点 */
    device_fd = device_init();
    /* 属性相关处理和启动 logo */
    /* 初始化 struct pollfd ufds[4]; */
    for(;;) {
        /* 进入循环，处理 ufds[4] 的事件 */
        nr = poll(ufds, fd_count, timeout);
        if (nr <= 0)
            continue;
        /* ..... */
    }
    return 0;
}
```

第四部分 Init 进程

init.rc 是在 init 启动后被执行的启动脚本，其语法包含了

Actions , Triggers , Services , Options , Commands , Properties 等。

```
on init
    export PATH /sbin:/system/sbin:/system/bin:/system/sbin
    mkdir /system
on property:ro.kernel.qemu=1
    start adbd
service vold /system/bin/vold
    socket vold stream 0660 root mount
```

使用方法参考 [system/core/init/readme.txt](#), 关键字参考 [system/core/init/keyword.h](#)。

第五部分 Shell 工具

Android 系统的启动后，提供了基本 shell 界面供开发调试使用。需要启动了一个名称为 console 的服务，实际上执行的程序：

`/system/bin/sh`

sh 代码的路径：

`system/core/sh`

toolbox 代码的路径：

`system/core/toolbox`

生成的文件 `/system/bin/toolbox`，目标文件系统 `/system/bin/` 中的具有一些符号将连接到 toolbox 上。

第六部分 C++ 工具库 libutils

libutils 是 Android 的底层库，这个库以 C++ 实现，它提供的 API 也是 C++ 的。Android 的层次的 C 语言程序和库，大都基于 libutils 开发。

头文件的路径：

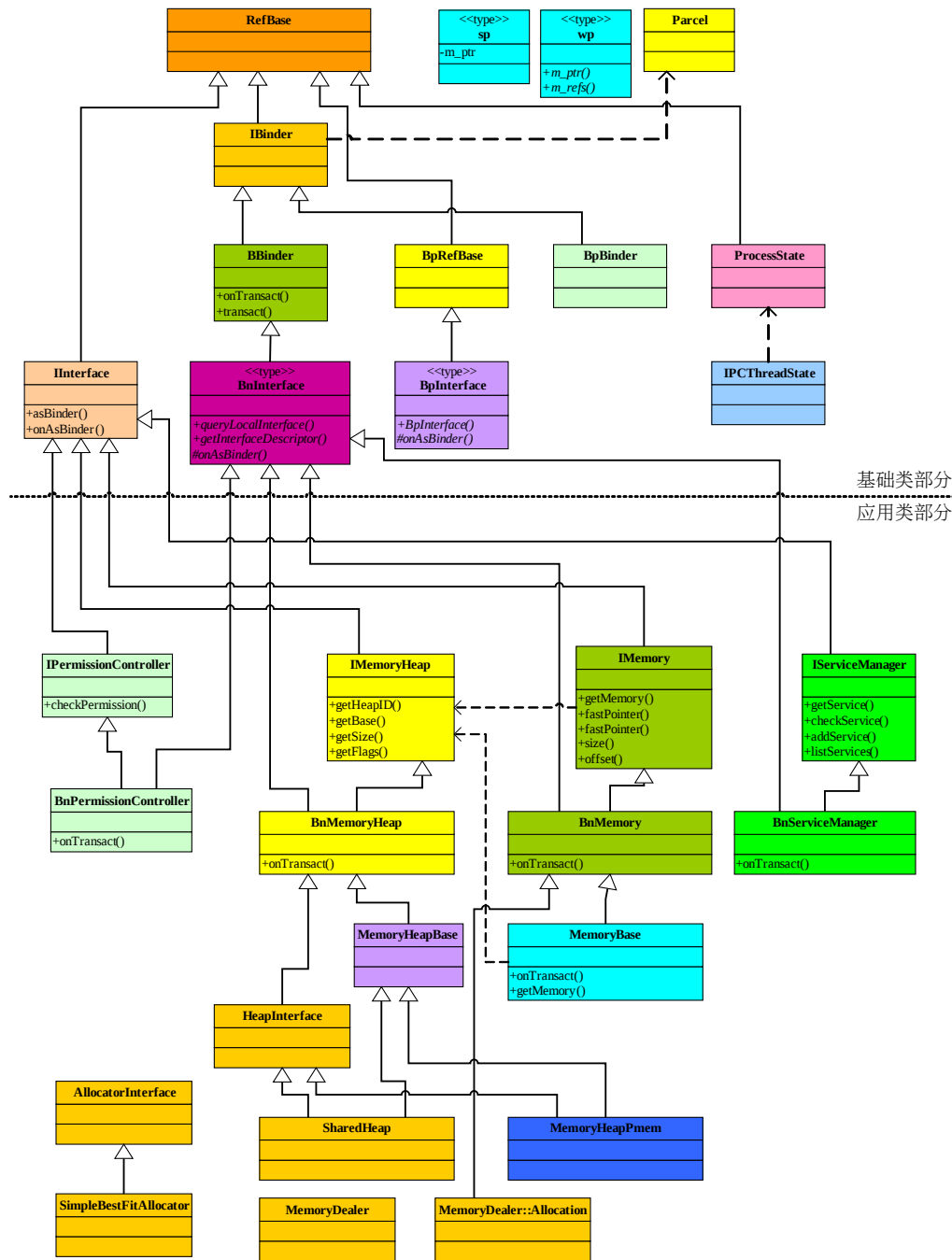
[frameworks/base/include/utils](#)



库的路径

[frameworks/base/libs/utils](#)

编译的结果是： **libutils.so**



第六部分 C++ 工具库 libutils

Errors.h :

定义宏表示错误代码

Endian.h :

定义表示大小端的宏

misc.h :

几个字符串和文件相关的功能函数

TextOutput.h :

定义文本输出的基类 TextOutput

BufferedTextOutput.h :

类 BufferedTextOutput , 它是一个 TextOutput 的实现

Pipe.h :

定义管道类 Pipe

Buffer.h :

定义内存缓冲区域的类 Buffer

List.h :

定义链表的模版类

第六部分 C++ 工具库 libutils

SharedBuffer.h :

定义类 **SharedBuffer** 表示共享内存。

String16.h :

定义表示双字节字符串的类 **String16**

String8.h :

定义表示单字节字符串的类 **String8**，并包含了从 **String16** 转换功能

VectorImpl.h :

定义表示向量的类 **VectorImpl**

Vector.h :

定义继承 **VectorImpl** 的类模版 **Vector**

SortedVector.h :

定义排序向量的模版 **SortedVector**

KeyedVector.h :

定义使用关键字的向量模板 **KeyedVector**

第六部分 C++ 工具库 libutils

threads.h :

定义线程相关的类，包括线程 Thread 、互斥量 Mutex 、条件变量 Condition 、读写锁 ReadWriteLock 等

socket.h :

定义套结字相关的类 Socket

Timers.h :

定义时间相关的函数和定时器类 DurationTimer 。

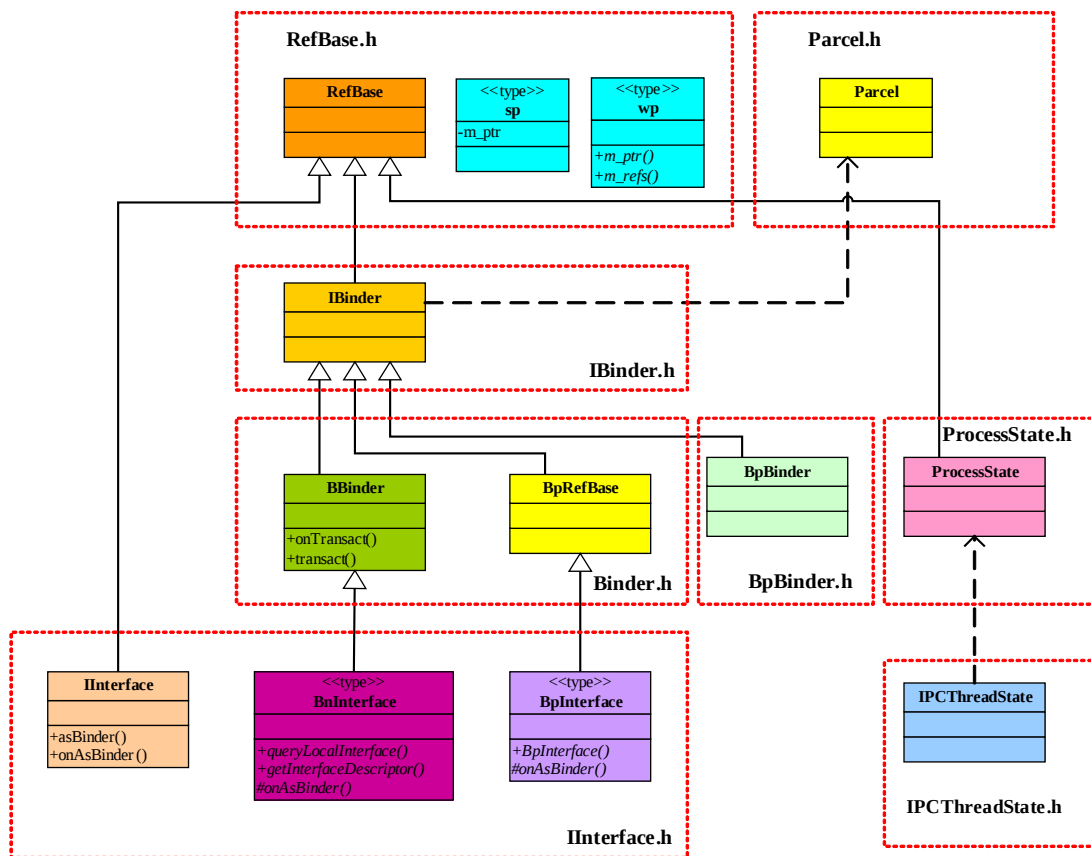
ZipEntry.h 、 ZipFileCRO.h 、 ZipFile.h 、 ZipFileRO.h 、 ZipUtils.h

:

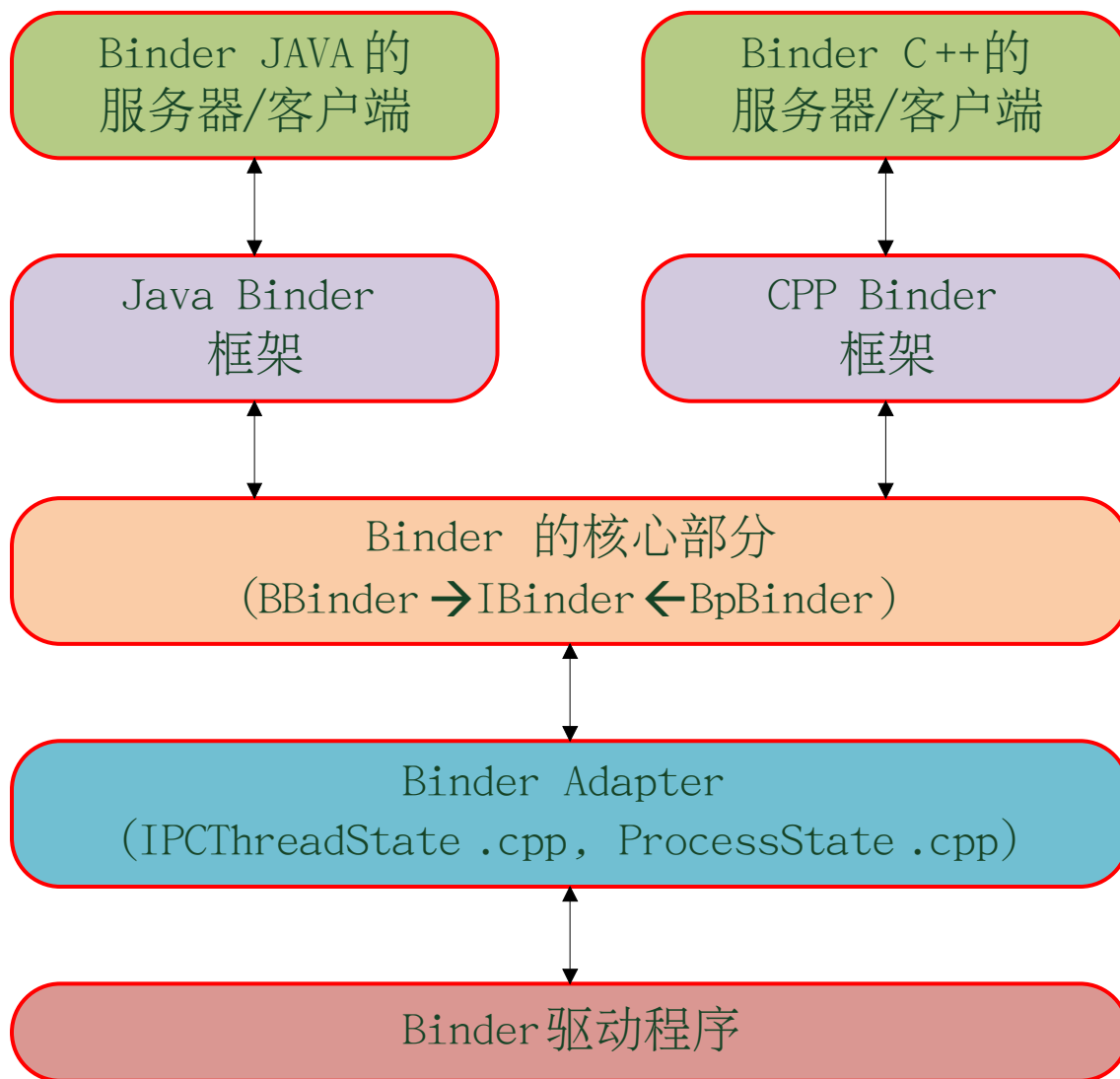
与 zip 功能相关的类。

第六部分 C++ 工具库 libutils

Binder 用于进程间的通讯（IPC），它的实现基础是运行与 kernel 空间的 binder 驱动。



第六部分 C++ 工具库 libutils



第六部分 C++ 工具库 libutils

RefBase.h :

引用计数，定义类 **RefBase** 。

Parcel.h :

为在 IPC 中传输的数据定义容器，定义类 **Parcel**

IBinder.h :

Binder 对象的抽象接口， 定义类 **IBinder**

Binder.h :

Binder 对象的基本功能， 定义类 **Binder** 和 **BpRefBase**

BpBinder.h :

BpBinder 的功能，定义类 **BpBinder**

Interface.h :

为抽象经过 **Binder** 的接口定义通用类，

定义类 **Interface** ， 类模板 **BnInterface** ， 类模板 **BpInterface**

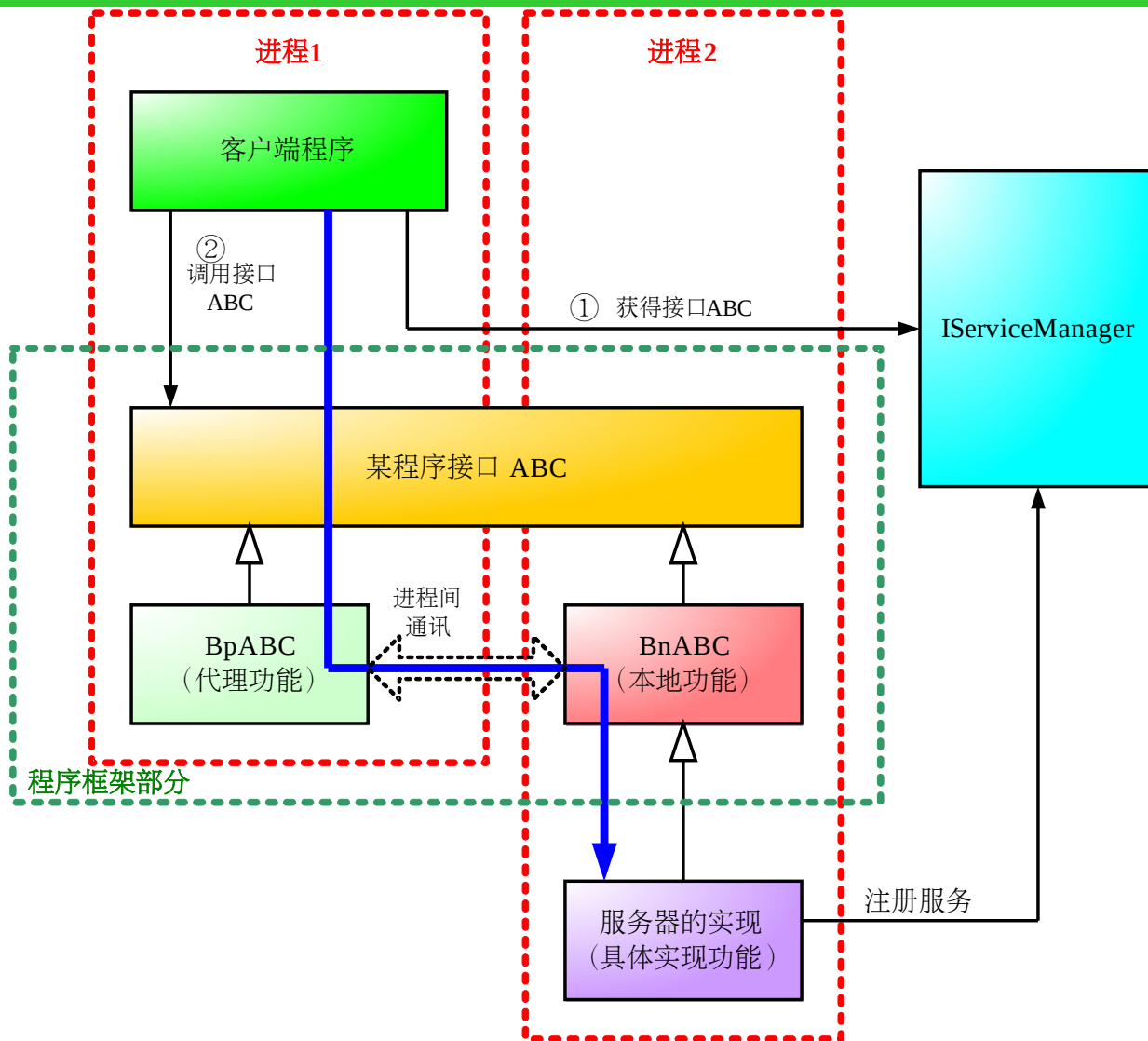
ProcessState.h

表示进程状态的类，定义类 **ProcessState**

IPCThreadState.h

表示 IPC 线程的状态，定义类 **IPCThreadState**

第六部分 C++ 工具库 libutils



第六部分 C++ 工具库 libutils

参考 `PermissionController` 的实现，这个在
`IPermissionController.h`
`IPermissionController.cpp`

框架内定义接口 `IPermissionController`，并且实现了 `BpPermissionController`。

这个类的使用方式是：

- ❑ 实现类继承 `BnPermissionController`
- ❑ 调用者调用类 `IPermissionController`

第六部分 C++ 工具库 libutils

IPermissionController .cpp(1)

[illegible]

第六部分 C++ 工具库 libutils

IPermissionController .cpp(2)

```
status_t BnPermissionController::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CHECK_PERMISSION_TRANSACTION: {
            CHECK_INTERFACE(IPermissionController, data, reply);
            String16 permission = data.readString16();
            int32_t pid = data.readInt32();
            int32_t uid = data.readInt32();
            bool res = checkPermission(permission, pid, uid);
            reply->writeInt32(0);
            reply->writeInt32(res ? 1 : 0);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```


第六部分 C++ 工具库 libutils

IPermissionController .cpp(2)

```
status_t BnPermissionController::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CHECK_PERMISSION_TRANSACTION: {
            CHECK_INTERFACE(IPermissionController, data, reply);
            String16 permission = data.readString16();
            int32_t pid = data.readInt32();
            int32_t uid = data.readInt32();
            bool res = checkPermission(permission, pid, uid);
            reply->writeInt32(0);
            reply->writeInt32(res ? 1 : 0);
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

第七部分 Android 的系统进程

Android 中几个重要系统进程为：

`/init`

`/system/bin/servicemanager` ,

`/system/bin/mediaserver`

`system_server`

`zygote`

前面 `init` 分析章节提到 `init` 通过解析 `init.rc` , 启动对应的服务程序。 `servicemanager` , `zygote` 和 `mediaserver` 都通过这种方式启动。 `system_server` 则是通过 `zygote` 孵化出来。这几个进程是 Android 系统运行的基础



谢谢！