

My Book

published by ReVIEW

別冊 詳解 Binder

有野和真 著

2017-04-10 版 発行

第 1 章

{intro} Binder とシステムサービス

別のプロセスに処理を依頼する場合には、何かしらのプロセス間通信 (IPC: Inter Process Communication) の仕組みが必要です。Unix ではパイプやシグナル、Unix ドメインソケットなどが良く使われますが、Android ではこれにプラスして Binder という独自の IPC も使われています。

本章では Binder について詳細に扱っていきます。Binder は通常のプロセス間通信の仕組みよりも用途が限定されています。システムプログラミングで使う、というターゲットの元に設計されています。その為プロセスを超えて送る事が出来る物も通常の IPC よりも多く、単純な値以外にも、ファイルディスクリプタやある種のオブジェクトなどを送る事が出来ます。Android はこの機能をフル活用していて、例えば 7 章の内容の裏側では、ActivityThread の内部クラスである ApplicaitonThread のプロキシを Binder で送ったり、Bundle や ActivityRecord なども Binder で送ったりされていました。そういう意味では Binder というのは Android にとって重要な技術と言えます。

一方で本章は他の章よりも読者に要求する前提知識が多く、本書の中では少し特殊な章となっています。そこで最初に 8.1 で、あまり前提知識が無い読者を想定して、本書の他の章を理解するのに必要な最低限の事、ひいては Android という物を理解するのに必要な Binder の最低限の所を説明する事から始める事にしました。

あまりこの分野に馴染みのない人がどこまで本章を読むのかは、8.1 を読んで、自身で判断してもらいたい、と思っています。8.1 の内容さえ理解しておけば、他の章を読むのには差し支えありません。本章が難しいと思ったら 8.1 だけ読んだ上で次の章に進んでもらえたら、と思います。

ですが、もし分散オブジェクトにある程度の理解があるなら、本章を最後まで読めば Binder についての全てを理解する事が出来るでしょう。

1.1 8.1 最低限の Binder 基礎知識 - 分散オブジェクトを知らない人向け

Binder は、使うのは簡単だけれど実装が複雑、という物です。使い方覚えるのは比較的簡単で、それだけ分かっていれば他の章を理解するのには十分です。

そこで本節では、難解な実現方法は気にせずに、利用者の視点に限定して必要最低限な話をしていきたいと思います。使う側に関して言えば、以下の三つを抑えておけば十分だと思います。

- Binder の特徴
- Binder で送れるもの
- IBinder とは何か、Stub.Proxy と Stub.asInterface

そこで本節では、なるべく専門用語などを出さずに、上記の事を説明してみたいと思います。逆に分散オブジェクトに詳しくて Binder の詳細に興味がある人は、本節の内容はあとでより詳細に扱うので、流し読みで十分です。8.2 から真面目に読み始めていただけたら、と思います。

8.1.1 Binder の特徴

Binder とはプロセス間通信の仕組みと、その上に構築された分散オブジェクトの仕組みです。この時点で「プロセス間通信」とか「分散オブジェクト」と言った専門用語が出てきてしまいますが、この節ではなるべくそういう言葉を使わないで説明してみます。

■コラム: 分散オブジェクトシステムと IPC、RPC

分散オブジェクトとは、別のプロセスのオブジェクトを自分のプロセスのオブジェクトのように呼べる技術の事です。これに関連のある言葉として、IPC(Inter Process Communication、プロセス間通信)、RPC (Remote Procedure Call) があります。

IPC はプロセスの間で何かしら情報をやり取りするもの全般を指します。この時点ではオブジェクトや関数と言った概念を前提とはしません。Unix のパイプなども IPC の仕組みと言えるでしょう。

RPC は単なる IPC を越えてプロシージャ、つまり関数を呼び出す、という構造を持っています。関数を呼び出す場合、相手の関数を識別する方法があって、それを通常の関数呼び出しのように呼べる、という事を意味します。さらに RPC の場合は関数の引数をシリアル化して送信し、受け取る側でデシrializeする仕組みが含まれるのが普通です。多くの場合 RPC はプロキシの関数を呼び出すと、相手側の実体の関数が呼ばれる、という振る舞いをします。そして間の部分は何らかの方法の自動生成になる事が多いと思います。古くは IDL という言語でインターフェースを書いて、そこからコードを自動生成していました。最近はインターフェースなどの型からリフレクションを用いて実行時に生成する物も多くなっています。

RPC と分散オブジェクトは昨今ではほとんど同じ意味に使われる事が多いですが、分散オブジェクトの場合はオブジェクトのインスタンスが存在する前提となります。オブジェクトのインスタンスがあると、一意性の識別と寿命の管理という問題が発生し、マシンをまたがる前提だといろいろと難しい問題が発生します。乱暴に言ってしまえば、分散オブジェクトでは RPC では必要無かった、規約に従ったリファレンスカウント処理が必要となる、という事です。

また、オブジェクトが複数存在する結果として、オブジェクトを探す、というのも重要な機能となる事が多いと思います。マシンを超えてネットワークからオブジェクトを探す、となると、なかなか複雑な仕組みとなります。

Binder の構成要素に照らし合わせると、Binder で IPC の仕組みをつかさどるのが binder ドライバとなります。その上の BpBinder と BBinder の仕組みでだいたい分散オブジェクトの仕組みは構成されている、と言ってしまって良いと思いますが、普通は IInterface や AIDL のレイヤまでを含めて分散オブジェクトシステム、と言うと思います。binder ドライバは最初から上に載る分散オブジェクトの為に作られているため、IPC の時点で寿命管理の仕組み（リファレンスカウント）や相手を識別するための仕組み（binder_node）が入っているのが特徴と言えます。またオブジェクトを探す仕組みとして servicemanager があります。

Binder は別のプロセスのオブジェクトを、自分のプロセス内の通常のオブジェクトと同じように見せかける技術です。同じような技術はたくさんあるのですが、Binder が少し変わっているのは、最初からシステムプログラミングの用途だけを念頭に作られている事です。これは一番下の、よそのプロセスとの通信の部分から、一番上、ユーザーが使うようなライブラリまで一貫しています。通信部分だけを別の用途で使えるようにもしよう、という配慮も無いですし、上のライブラリを通信部分を差し替えて使えるようにもしよう、とも考えていません。スタックを上から下まで抱え持つて全体を最適化する、というのは Google らしいですね。

Binder はシステムプログラミングを前提にして、それ以外には必要のない抽象化は一切行わず、なるべくシンプルな実装となっています。それを踏まえた Binder のカタログ的な特徴としては、以下のような物になるでしょう。

- 同一マシン内だけしか使えない
 - シンプルなインターフェース
 - コンパクトな実装
 - 高パフォーマンスで省メモリ
- ドライバとしてカーネルにアクセスする事前提
 - 通信のレイヤでスレッドを認識出来る
 - 相手のタスク構造体を直接操作出来る
 - * ファイルディスクリプタが送れる

カーネルのドライバでの実装を前提とするので、相手のタスク構造体^{*1}を直接触れる、というのは珍しい部分に思います。

個々の項目の妥当性などを詳細に検証していくと複雑な実装に踏み込まなくてはいけませんが、この位の印象をそのまま持っていたら、そう大きく実態とは乖離していないと思います。

1.2 8.1.2 Binder の通常の使われ方 - getSystemService() × ソッド

Binder を一般のユーザーが使うのは、9割くらいはシステムサービスを使う時だと思います。

^{*1} Linux カーネルの用語。プロセスを表すカーネルのデータ構造。

第1章 {intro} Binder とシステムサービスBinder の通常の使われ方 - getSystemService() メソッド

0 章で見たように、Android はシステムサービスから構成されたシステムです。システムサービスはアプリのプロセスとは別のプロセスで動いているので、システムサービスの呼び出しは全て Binder を使った呼び出しどなります。

システムサービスを使う具体例を見てみましょう。例えば現在 Activity が表示されているディスプレイのサイズを取りたい場合、Activity や Context に対して

リスト 1.1: WindowManager サービスの取得

```
WindowManager wm = (WindowManager) getSystemService(WINDOW_SERVICE);
Display disp = wm.getDefaultDisplay();
...
```

などというコードを書くとディスプレイのサイズが取得出来ます。ディスプレイのサイズに限らず、アカウントマネージャでもオーディオでも、別のプロセスで実現されていそうな機能は、この getSystemService() というメソッドで何かを取得して、それをキャストして作業するのが Android の基本となっています。

上記のコードのうち、wm.getDefaultDisplay() の呼び出しが Binder によるメソッド呼び出しどなります。見て分かる通り、通常のオブジェクトに対するメソッド呼び出しとコードの上では区別できません。

getSystemService() を使って取得したオブジェクトに対してメソッドを呼び出すのが Binder を使っている例なんだな、とだけ覚えておいてもらえば十分だと思います。

■コラム: 分散オブジェクトに関わる物達

分散オブジェクトはなかなか複雑なシステムでありながら 90 年代末期には流行っていた為、割と多くの中年プログラマは良く学んでいます。一方で web の時代が来た後にはあまり流行らなくなり、最近のエンジニアだとちょっと使われている事はあるけれど大して知らない、という人も多くなってきた印象です。

90 年代の終わりには複合ドキュメント、というのがこれからは流行る、と言われていました。ようするに Word ファイルの中に Excel のグラフを貼って、ワープロソフトの中で表計算ソフトを動かす事でグラフも編集出来るようにする、という奴です。結局はあまり使いやすく無い上にワープロに貼りたい物は限られているので、アプリケーションごとに対応してしまう方が良い、という結論になったように思います。

Windows で複合ドキュメントを実現するために使われていた分散オブジェクト技術が COM です。COM は Word や Excel のアプリケーション、Internet Explorer などのブラウザを Ruby や JScript といったスクリプト言語から触る為に良く使われている技術に思います。巨大なアプリケーションを外部のスクリプト言語から操作する、というのは分散オブジェクトの使われ方のうち、現在に至るまで最も成功している使われ方に思います。

また、分散オブジェクトはサーバーサイドの開発でアプリケーションのロジックを作る単位として使う、という使い方が提案された事もありました。アプリケーションのロジックをマシンを分散させる事で負荷分散をしたり出来るんじゃないかな、と。でも結局はコンポーネント単

位では無くでアプリケーションサーバー全体を複製しておく方がてつとりばやく、効率も良いという結論になり、この目的で分散オブジェクトが使われる事もほとんど無くなりました。

1.3 8.1.3 Binder で送信できるもの

Android のソースを読んでいて、Binder に付きあたった時に周辺のコードを理解する上で重要なのは、Binder で何を送信出来るか、という事です。送信出来るものさえ把握しておけば、それが内部でどのように送信されているかはあまり知らなくても、ソースを読む上では問題ありません。

Binder で送信できる主な物は以下の 3 つです。

1. 数値、文字列などの値
2. ファイルディスクリプタ
3. サービスオブジェクト (BBinder のサブクラス)^{*2}

1 はそのままなので良いでしょう。

2 は Binder の一つの特徴となっています。Linux では、ファイルを open すると、カーネル内にそのファイルを表すファイルオブジェクトが出来て、各ユーザープロセスにはこのファイルオブジェクトを表すテーブルが作られます。この各テーブルに対応するファイルオブジェクトが入り、テーブルのインデックスがファイルディスクリプタと呼ばれる物になります。

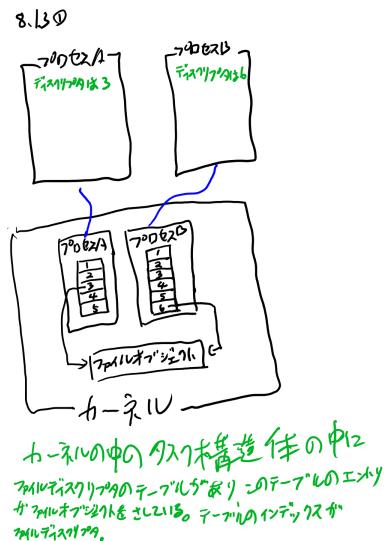
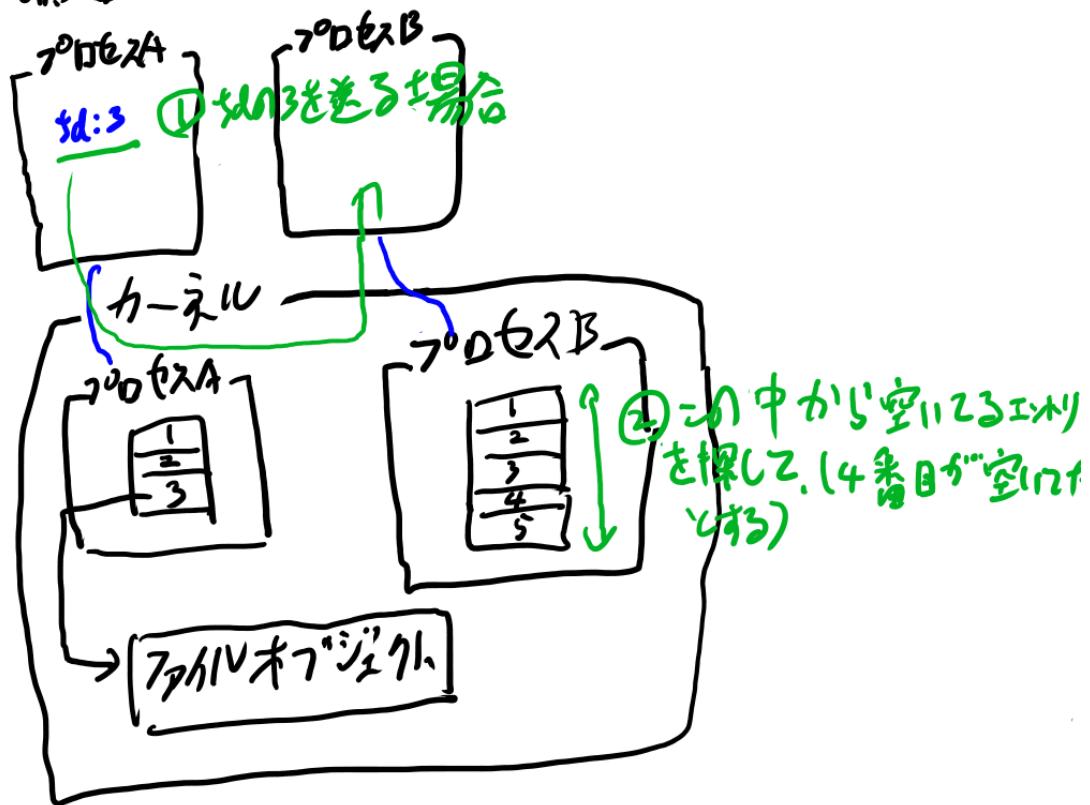


図 1.1 ファイルディスクリプタとファイルディスクリプタテーブル

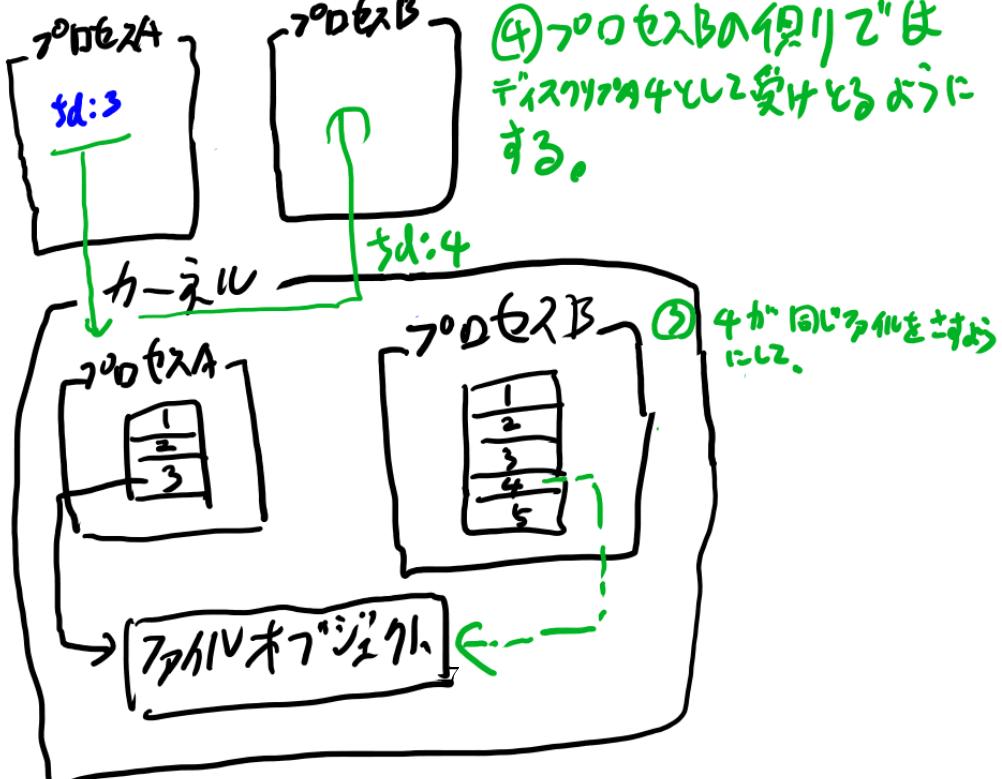
^{*2} 実際はサービスプロキシも送る事が出来ます。つまり IBinder のサブクラスを送る事が出来ます。

binder ドライバはプロセス A からプロセス B にファイルディスクリプタを送信している事に気づいたら、プロセス A のファイルディスクリプタテーブルを見て、その参照先のファイルオブジェクトを探し出し、それをプロセス B のファイルディスクリプタテーブルに追加し、そのテーブルのインデックスに変換します。これは双方のタスク構造体というプロセスを表すデータ構造を直接触れるデバイスドライバだからこそ出来る芸当です。

8.1.3 ②



④ $sd:3$ の奥で "は
ディスク DMA で受け取るよう
にする。



3のサービスオブジェクトというのがBinderの中心となる物です。技術的にはC++のBBinderのサブクラスという事になります。BBinderのサブクラスをプロセスをまたいで送受信出来る、というのがBinderという物の中心的な機能です。6章のActivityManagerServiceも、7.2.3で登場したApplicationThreadも、BBinderのサブクラスです。

サービスオブジェクトを送信して、相手のプロセスからこのサービスオブジェクトのメソッドを呼ぶ事が出来る、これがBinderです。

■コラム: 分散オブジェクトいろいろ

世の中には様々な分散オブジェクトの仕組みがあります。私は私が関わった物以外はあまり知らないので、ここで全ての一覧を提示する事は出来ませんが、幾つか名前を挙げてみましょう。

まず本書のコラムでもたまに言及される、分散オブジェクトの代表としてはCOMが挙げられます。COMはWindowsで使われている分散オブジェクトのシステムで、トップクラスに使われている物の一つと言えます。IEとOfficeがCOMから操れる、というのがこの技術がここまで使われた直接の理由だと思います。このシステムではIDLの方言であるMIDLというインターフェース定義言語をコンパイルして、間のコードを自動生成していました。COMは使う側から見るとシンプルなため、互換なシステムも良く作られました。モバイルプラットフォームだとBREWやEMPといったシステムでは、インターフェースはCOMとなっています。

同じMicrosoftでも、より最近の.NETでは、WCFという分散オブジェクト技術があります。これは分散オブジェクト技術を含んだより幅広い物で、オブジェクト呼び出し以外のweb APIのような呼び出しもオブジェクトの呼び出しのように呼び出す事が出来ます。これはインターフェースから実行時にコードを自動生成します。型情報がネイティブよりも動的コード生成が容易な仮想マシンではこのスタイルが主流となっていると思います。

Microsoft以外で一番有名な分散オブジェクトシステムと言えばCORBAでしょう。昔はWindows以外で分散オブジェクトと言えばCORBAだった、と言っても過言では無いくらい、CORBA一色でした。オープンな規格で言語非依存でその他様々な物に非依存な分散オブジェクトシステムです。最近見かける所では、Linuxなどで良く使われるGnomeで採用されました。

JavaのRMIなども典型的な分散オブジェクト技術です。またEJBなどはトランザクションなど多くの付加機能がついていますが、これも分散オブジェクト技術の一つと言えます。ORBというJavaのCORBA実装（少し厳密な言い方では無いですが）も有名です。Javaはその他にもDynamic Proxyを用いたより軽量な分散オブジェクトのシステムをたまに見かける気がしますが、EJB以降は分散オブジェクト自体があまり流行っていない印象を受けます。

OS XなどにもNSConnectionという分散オブジェクトの仕組みがあります。Macもかつては複合ドキュメントを推していた時代があったと思いますが、最近ではあまり聞かなくなりました。

こうしてみると分かるように、だいたい分散オブジェクト自身は流行りが終わった技術、という印象を受けます。一方でシステムを構築するには何かしらこの手の技術が無いと不便でもあり、Androidがいまさら独自に必要最低限の物を作った、というのは、長くこの業界に居

た人間としてはなかなか感慨深いものがあり、何がサポートされていないかを見ると「ああ、分散オブジェクト」というと複雑になりがちだけど、結局必要なのはこの辺だったんだなあ」と10年越しに答えを見せてもらったような気持ちになります。

1.4 8.1.4 サービスを実行する側のスレッド

Binder はよそのプロセスのオブジェクトを通常のオブジェクトのように呼ぶ事が出来る、という仕組みなので、普段は深くいろいろ考えずに、よそのプロセスのオブジェクトのメソッドを呼んでいる、と考えるだけで良いようになっています。

ですが、少し細かい事を考え出すと事はそう簡単ではありません。まず、スレッドがどうなっているのか、という問題があります。プロセス A がプロセス B のメソッドを呼んでいるとしましょう。プロセス B のメソッドを実行する時にはプロセス B の中にこのメソッドを実行しているスレッドが無くてはいけません。

通常スレッドはプロセス内で作られます。外部から勝手にスレッドを作って走らせる、という事は、普通はしません。ですから、プロセス B でメソッドを実行しているスレッドは、プロセス B が作ったものであるはずです。そこで通常、プロセス B は、あらかじめ Binder のメソッド実行専用のスレッドを作つておいて、ずっと外部のプロセスからの呼び出しを待ち構えておきます。この待ち構えているスレッドが、メソッドを実行するのです。これはメインスレッドとは異なるスレッドです。

Android では Java のプロセスが起動した時に、このスレッドを開始して待ち続けます。詳細は zzz で扱います。

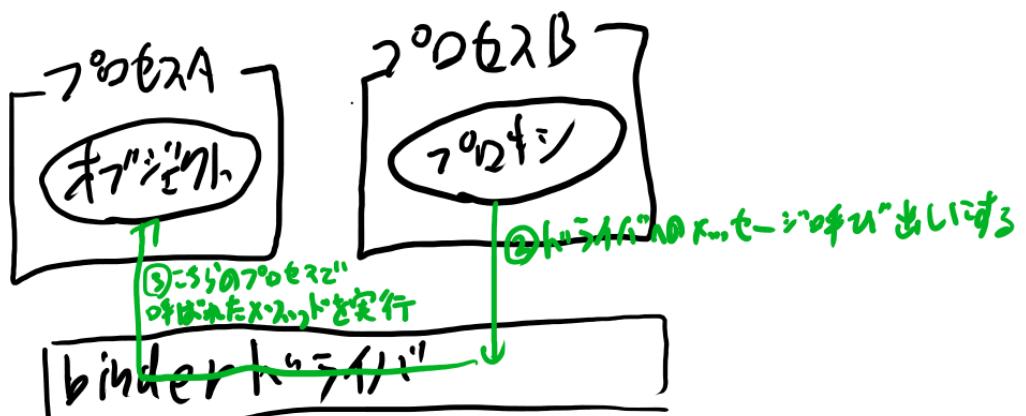
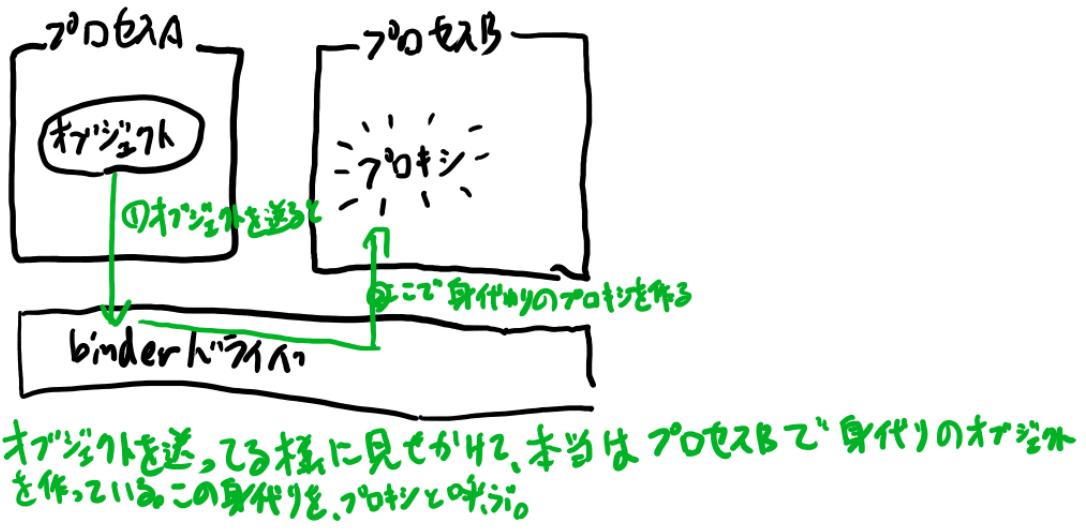
重要なポイントとしては、この Binder 越しに呼び出された側のスレッドは、いつも GUI スレッドとは別の Binder 処理用のスレッドだ、という事です。他の章を読む時には、ここは意識しておく必要があります。

1.5 8.1.5 サービス実装とサービスプロキシ

Binder でサービスオブジェクトを送る事が出来る、という話をしました。

実際には送った先はプロキシオブジェクトという物に変換されています。これはメソッド呼び出しを、そのメソッド呼び出しを表すメッセージに変換して、サービス実装側のプロセスに送信する、というオブジェクトです。

9.1.5①



プロキシは、リストの中から引数などの呼び出された条件をメッセージにして送信。実際の実行はプロキシが行う。

図 1.3 プロキシのメソッドを呼ぶとメッセージが送信される

多くの場所ではその事を意識せずにただよそのプロセスのオブジェクトのメソッドを呼んでいる、と思っておけば良いのですが、たまにその幻想が綻びていて、実際はプロキシである、という事を意識しないといけない事があります。

そこで実装している側と、それを呼ぶプロキシ側に分かれている、という事は知っておく必要があります。その現実が一番深刻に表れてしますのが次の IBinder です。

1.6 8.1.6 IBinder と XX.Stub.asInterface と XX.Stub.Proxy

アプリを開発していると、Binder という事を意識しないといけない事はほとんどないのですが、たまに出てくるのがこの IBinder という奴です。ソースコードを調べていても出てくると思います。同じ名前の C++ のクラスもありますが、ここでは Java の IBinder の話です。

この IBinder とは何なのか？ とソースコードを読んでもなんだか何もしてないように見える。そしてそれがなんだかごちゃごちゃしたコードの XX.Stub.asInterface() とかの引数に渡されている。なんだか良く分からないので、見様見真似でコピペで済ます、というのが良くある IBinder を前にした光景に思います。

これを全部ちゃんと理解しようとすると本章の内容をちゃんと追っていく必要が出てきてしまいますが、概念的な事はそこまで深く理解してなくても理解は出来るので、ここで簡単に説明しておきます（それでもやや難しいですが…）。

IBinder とは、サービスの実装とプロキシの両方を区別なく扱うためのクラスです。イメージとしては C 言語の union のようなもので、実態としてはサービス実装とサービスプロキシのどちらかが入っています。このどちらかが入っているがどちらが入っているかが分からない、というのがコードを読みにくくしています。

IBinder はサービスのオブジェクトがそのまま入っている場合はただキャストすれば良い訳です。少し読みにくいのがプロキシ側だった場合です。

プロキシのクラスは、通信用のオブジェクトをラップして作られています。そして IBinder はプロキシの時は通信用のオブジェクトが入っています。だからプロキシオブジェクトを得るために、この通信用オブジェクトをプロキシのクラスのコンストラクタに渡してやる必要があります。このラップするプロキシオブジェクトのクラス名は「インターフェース名.Stub.Proxy」という名前にする事になっています。例えば IHelloWorld というインターフェースなら、プロキシクラスは IHelloWorld.Stub.Proxy となります。おかしな名前ですが、これは Java の言語の制約上から来ていて、間の Stub に大した意味はありません。

ですから、IBinder がプロキシの場合には、これをコンストラクタに渡してプロキシクラスを作るのです。

リスト 1.2: IBinder からプロキシクラスを作る例

```
// もしこの引数の token がプロキシなら
void someMethod(IBinder token) {

    // プロキシクラスのコンストラクタに渡して、ラップするプロキシオブジェクトを作る
    IHelloWorld proxy = new IHelloWorld.Stub.Proxy(token);
```

```
// 以下この proxy のメソッドを呼び出す  
}
```

この、

1. サービス実装のオブジェクトならただキャストするだけ
2. プロキシの通信用オブジェクトならラップしたオブジェクトを返す

の二つをやってくれるのが XX.Stub.asInterface() です。IHelloWorld というインターフェースなら、IHelloWorld.Stub.asInterface(token) と呼ぶと、とにかくこの IHelloWorld のインターフェースとして使えるオブジェクトが返ってくるので、実装者はこの IBinder がどちらだったのか、という事を気にする必要は無い訳です。

まとめると以下のようになります。

1. IBinder にはサービス実装のオブジェクトがそのまま入っている場合と、プロキシに使う通信オブジェクトが入っている場合がある
2. 「インターフェース名.Stub.Proxy」という名前がプロキシクラスで、これは通信オブジェクトをラップして機能する
3. 「インターフェース名.Stub.asInterface()」というメソッドが IBinder のそれぞれの場合を適切に処理してくれる

この三つが、Binder になるべく関わらないようにしている人でもたまに必要となる知識です。

1.7 8.1.6 Binder の基礎知識、まとめ

本節では、難しい話題を理解していくなくともこれだけ知っていれば他の部分のソースを読む時に困らない、という事をまとめてみました。

1. Binder のカタログスペック的な特徴を知っている
2. getSystemService() を使ったオブジェクトに対する操作は Binder を使っている事を知っている
3. 何が送信できるか知っている
4. サービスを実行しているスレッドが GUI スレッドで無い事を知っている
5. IBinder と「インターフェース名.Stub.Proxy」と「インターフェース名.Stub.asInterface()」を知っている

くらいを抑えておけば、内部の詳細に立ち入らなくても他の部分を理解する事は出来ると思います。内部の詳細を理解せずにこれらの事を理解しようとすると、どうしてもぼやっとした部分が残ってしまうと思いますし、たまにここでは扱っていない事もちょっとはあると思いますが、それはそういうものだ、と飲み込んでしまって、詳細を学ぶ時間を他の事に充てるのも一つの選択でしょう。

ここより先は、もっと細かい事を知りたい、という人の為の内容となります。

第 2 章

{main-intro} Binder の本格的な入門

ここからはある程度この分野に慣れている人を対象に、Binder とその主たる応用例となるシステムサービスの詳細をしっかりと理解していくという内容となります。本節では以降の節全体をまとめた全体像の提示や、以後の節を読んで行く為の導入として、Binder の詳細を理解する意義や想定する前提知識の話をていきます。

なお、本書ではドライバ名としては小文字の binder を、Binder という仕組み全体を表す時は大文字始まりの Binder を使っていきます。

2.1 8.2.1 Binder とシステムサービスを詳細に理解する意義

8.1 の内容で、他の章を読んだりソースコードを読んでいく分には十分と言いました。それでは以後の長い本章の内容は何故あるのか？ という話を最初にしたいと思います。

Binder を詳細に理解していく意義としては、以下のようない物があると思っています。

1. ソースコードを読んでいて Binder が登場しても、全てを正確に理解出来る
2. システムサービスのメソッド呼び出しをカーネルのコンテキストスイッチのレベルで理解出来る
3. 全てのコードが公開されている、世界中で日々使われている分散オブジェクトのシステムを学習出来る
4. Android の Android らしさを知る手がかりとなる

(1) 1 が一番大きな所だと思います。Android というシステムのソースコードを調べていく時に、「ここから先は分からぬ…」と毎回壁となってしまうのが、この Binder 周辺だと思います。それは本章がこれだけ長い事からも分かる通り、やろうとしている事が単純な割にはソースコードで知らないくてはいけない事が多いからです。調べたい本題が別にある状態でこの Binder のソースの深い森に遭遇してしまうと、どうしてもそれを最後まで読み切るのは難しく、いつもこの森の前までソース読みを終える事になってしまいます。

実際のプロダクションの現場において、この森の手前までの調査で困る事はおそらくほとんど無いと思います。ですから仕事で飯を食べるだけなら、多くのプログラマはこの手前までも十分でしょう。

でも Android がこれまでの携帯電話のシステムと比較して最も新しかった点は、携帯電話のシス

テムとしては信じられないくらいオープンだった事です。せっかく全てのソースコードが公開されているのですから、Android を楽しみ尽くす為には、ソースは全て理解出来る方が絶対に良いと思います。

全てを理解しよう、と思った時に大きな壁として立ちはだかる Binder 周辺の全構造がちゃんと解説されている本章は、Android を楽しもう、と思う読者には、大きな助けとなると思います。

(2) システムサービスの呼び出しなどに代表される Binder 越しの呼び出しで、実際に何が起こっているのかをカーネルのコンテキストスイッチのレベルで理解出来る、というのは、システムを深く理解する立場からすると重要になります。コンテキストスイッチのレベルから分かる事で、汚されるキャッシュや TLB エントリなど、既にベテランのデベロッパが知っている多くの周辺知識が使えるようになります。

最終的なパフォーマンスは計測の必要がありますが、Android のシステム構成がどの位重そうか、という感覚が分かるようになります。

(3) この 3だけはちょっと変わった視点で、分散オブジェクトのシステムの勉強としての視点となります。複雑になりがちな分散オブジェクトシステムですが、Binder は他のシステムと比べるとローカルだけを前提としていて、使われる環境も Androidだけをターゲットにしているので非常にコンパクトなシステムです。そうでありながらおもちゃの分散オブジェクトシステムとは違いプロダクションで最も良く使われている分散オブジェクトシステムの一つと言えるくらい使われていて、しかも現在でも使われ続けている現代のシステムとして、十分な実績があります。

実績のある分散オブジェクトシステムを学ぼうとすると間の抽象化レイヤが多すぎてなんだかぼやっとした理解にとどまってしまう事がが多いと思いますが、Binder は抽象化を極限までしない実装となっているため、一人の人間が隅から隅まで理解出来るシステムとなっています。分散オブジェクトを本格的に学ぶ最初のシステムとしては、最も良い教材だと思います。

(4) 最後は上三つと比べると少し抽象的な話となりますが、Binder は Android というシステムのプロセス構成という、モバイル OS としての立ち位置を考える時にキーとなる部分をつかさどる技術となっています。

モバイルの OS としては全て単一のプロセス内で構築する原始的な RTOS のような物から、全てプロセスを分けるマイクロカーネル的な物までいろいろと考えられます。そして多くのシステムで全てを別プロセスにするのは重すぎるため、アプリは dll にしたりと言った工夫が見られます。

Android では Binder という物を導入する事でモバイル OS としては旧来と同程度に貧弱なリソース（か少しリッチな程度）のデバイスでも動きつつ、将来的にはプロセスを分けていきよりリッチなデバイスで高パフォーマンスなシステムとしていく、という進化の道筋を最初につけておいたのが特徴的です。

Binder を詳細に理解しておくと、この辺のバランス感覚を知る事が出来て、ひいては Android の Android らしさ、のような物を知る事が出来ます。こうした哲学や「らしさ」という要素は、一段深い Android という物の理解を生むと同時に、何故いろいろある中で Android だったのか？ と言ったより難しい問への答えを自分の中で探す時の手掛かりとなります。

■コラム：分散オブジェクトとマイクロカーネルの夢

私的な話となりますますが、著者の私は COM などの分散オブジェクト技術が好きな方です。

COM には若いころ結構な時間をかけて学んだ思い出があります。

カーネルを小さく保って最初からオブジェクト指向ベースの OS を作る、というのは結構多くの人が夢見た所で、私も若いころは MSR の MMLite などのような同コンセプトのシステム資料を見ては胸躍らせた時代があります。

Binder の起源となった BeOS については、私はあまり詳しくありませんが、同様にマイクロカーネルでオブジェクト指向ベースなシステムだったと理解しています。その BeOS の Binder を元に Linux というマイクロカーネルと正反対のカーネルが組み合わさって、分散オブジェクトベースのシステムでありながら現実の数々の問題を解決している、というバランス感覚は、夢見る青二才とは格の違う真の OS アーキテクトの凄みを感じさせる部分だと思っています。

Honeycomb の頃などにその設計思想を活かして現代的な GUI システムへと変貌していくさまをリアルタイムで見ていた私は、久しぶりに若いころの興奮を思い出しました。

====[/column]

2.2 8.2.2 Binder を詳細に理解するのに必要な前提知識

本章の以降の内容では、分散オブジェクトシステムの開発の経験を前提とさせてもらいます。具体的には

1. IDL を書いてコンパイルし、スタブの実装を書いてプロキシを使った事がある（またはその意味が分かる）
2. スタブ、プロキシという言葉を理解している
3. IPC、RPC、分散オブジェクト、シリализацияといった用語が分かる
4. C 言語などの低レベルの通信層のコードに慣れている（オブジェクトのシリализация等）

という位をイメージしています。昔書いた事はあるが忘れかかっている、という人や、完全に上記条件は満たさないが似たようなシステムを使っている（例：インターフェースからの動的コード生成系のフレームワーク利用者など）人などに向けて、多少は説明も行いますが、まったく知らない人が上記の事を学ぶには本書は適切では無いと思います。

上記の事を学びたい人は、分散オブジェクトの入門書などを学ぶのが良いと思います^{*1}

2.3 8.2.3 Binder を構成するレイヤ

分散オブジェクトのシステムは一般に複数のレイヤで実現されていて、各レイヤでプロキシと実装側のクラスが出てくる事から、毎回同じようなクラスが出てきて同じような説明をする事になります。

そんな説明を読み続けて行くと、そこで言っている事は理解する事は出来ても、そもそもなんで今こんな話をしているのか、という事が良く分からなくなっていく事になります。

^{*1} 今なら gRPC や Thrift などでしょうか。私の頃は COM で学びました。

そこでまずは全体の構成を見てみたいと思います。以下の各節で何の話か分からなくなった時にはここに戻ってきてみてください。

8.2.3 ① Binderを構成するレイヤ。

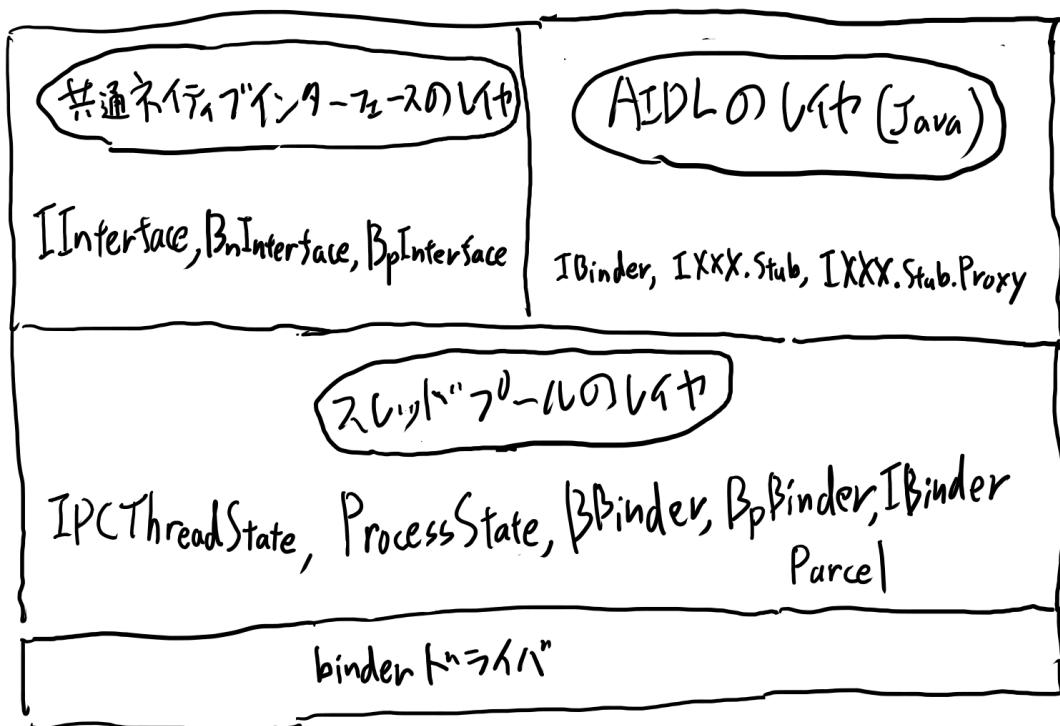


図 2.1 Binder を構成するレイヤ達

まず一番下から見ていくと、一番下は binder ドライバとなります。binder ドライバが二つのプロセスの間の通信を担当します。

その上にはスレッドプールを実現するレイヤがあります。^{*2} クラスとしては ProcessState、IPC-ThreadState、BBinder、BpBinder、IBinder です。

このレイヤで単純な ioctl によるプロセス間通信が、スレッドプールと C++ のオブジェクトに対するメッセージ送信、という形へと発展します。スレッドプールは、その実現で BBinder というオ

^{*2} なお、このレイヤ分けの図も名前も私がソースを読んで決めたもので、公式から発表されている物ではありません。そもそもこの周辺はあまり公式の情報はありません。

プロジェクトの基底クラスと BpBinder というプロキシの基底クラスがやりとりされる、という前提に基づいる為、これらは合わせて使う必要があります。この基底クラスで、全てのサービスで共通となるメッセージの処理を行います。

ここまでオブジェクトという対象をやりとりするのですが、呼び出し側はサービスが自分のプロセスに居るのか別のプロセスに居るのかに応じてコードを変えなくてはいけません。

このスレッドプールのレイヤの上には、それぞれ並行して二つのレイヤが存在しています。ネイティブと Java で、呼び出し側コードをサービスがどこのプロセスに居るかを意識せずに使えるようになる為のレイヤです。共通ネイティブインターフェースのレイヤと AIDL のレイヤはお互い依存していない、独立したレイヤです。どちらも下のスレッドプールのレイヤを使用しています。

共通ネイティブインターフェースのレイヤは、クラスとしては IInterface、BnInterface、BpInterface の三つのクラスとなります。

AIDL のレイヤは Java によるサービス実装のレイヤとなります。AIDL からコードを自動生成する仕組みを用いるレイヤで、主なクラスは生成するインターフェースに応じた名前となる為、図では IXXX と表記しました。通常は IAudioService などのような名前となります。

ネイティブでサービスを実装する時は IInterface の方を使い、Java でサービスを実装する場合は AIDL を使う事になります。

この二つのレイヤが実際にサービスを実装する人が直接用いるレイヤの為、なるべく実装者が不要なコードを書かなくても済むように自動生成やマクロなどのいろいろなトリックが使われています。また、実装にはいろいろと慣例があり、その慣例に何も考えずに従って実装すると、下の方の仕組みがどうなっているか、という事はあまり意識せずに、分散オブジェクトとしてのシステムサービスが実装出来るようになっています。開発者があまり良く理解していないなくてもとりあえず見様見真似で動くコードが書ける、という訳です。

仕組みとして重要なのはドライバと、その上のスレッドプールのレイヤ、つまり IPCThreadState や BBinder のレイヤとなります。そこより上の層は開発者が実際に目にする事は多くとも、実装的には複雑な事はありません。Binder の全体像をしっかりと把握するためには binder ドライバとスレッドプールのレイヤをしっかり学ぶ事がポイントとなります。

2.4 8.2.4 本章の以後の構成

分散オブジェクトの解説をする時に、上からするか下からするかは難しい選択です。

上から解説すると各コードの使われ方が分かるので目的は理解しやすい反面、毎回依存する下のレイヤーはまだ解説されていない状態で現在のレイヤーのコードを理解しないといけない為、読み手のコードリーディングの慣れを要求します。一方下から解説すると依存する物は既に説明されたものだけなので説明は全て理解出来る反面、現在説明している事が何に使われるかが分からぬまま説明が続く事になります。

本書では下から順番に解説する事を選びました。その為説明はそこまでに全て出てきた要素で閉じるため読みやすい反面、そのコードがどう使われるかは先を読まないと分からない、という構成になっています。読んでいて何のために今説明しているコードがあるのか、という事が良く分からなくなったら、少し先を読んでみてから戻ってくると言っている事が分かる、という事があると思いますので、そういう読み方をしてみて下さい。

まず 8.3、8.4、8.5 の三つの節で binder ドライバのレイヤの話をします。その後 8.6 でスレッドプールのレイヤを、8.7 で共通ネイティブインターフェースのレイヤを扱い、8.8 で AIDL のレイヤ、つまり Java によるシステムサービス実装を扱います。

最後に 8.9 で Binder の仕組みを実際に使っている側に視点を移して、システムサービス関連のプロセスがどのように動いているか、他の章で Binder が絡む所を Binder 側から見るとどうなるか、などの他のコンポーネントから見た Binder という位置づけを見ていきます。

なお、本章ではかなり低レベルなコードがたくさん出てきます。その為、少し他の章よりはプログラムを読みなれている人向けになっています。コードは説明の為に使っているのであって、動く事は意図していません。たとえば実際には 0 クリアをしないといけない重要なフィールドを無視したりしている為、完全に動くコードではありません。

それでは順番に見ていきましょう。まずは binder ドライバです。

第3章

{syscall} binder ドライバを扱う 三つのシステムコール - open, mmap, ioctl

Binder の仕組みのうち、一番下のレイヤとなるのが binder ドライバです。binder ドライバ自身はとても単純な物で、提供している機能もかなり原始的です。

この節から続く三つの節で、binder ドライバについて解説していきます。この節では binder ドライバを扱う基本的なシステムコールである、open(), mmap(), ioctl() について、使い方の例を見ていきます。次の 8.4 ではこれらのシステムコールを使ってどのようにメッセージを送受信するか、その内容と使い方についてみていきます。8.5 ではこれらのメッセージで送信できるものについて、ドライバの内部実装と合わせて見ていきます。

なお、本章でサービスと言った場合はシステムサービスの事だとします。

3.1 8.3.1 binder ドライバの特徴

binder ドライバはプロセス間通信の仕組みです。サービスというプログラミングモデルでシステムを組む為に、それ専用のプロセス間通信の仕組みを作った、それが binder ドライバです。

以下のような特徴があります。

1. ローカルに特化している
2. ドライバとして実装されている為、Linux カーネルの内部データ構造を用いる事が出来る
3. サービスというプログラミングモデルを前提としていて、スレッドプールやリファレンスカウントのサポートが最初から入っている
4. 呼び出し元のプロセスの uid を正確に把握している
5. ファイルディスクリプタを送信出来る

以上を 8.1.1 と比較すると、Binder の特徴の多くはそのドライバで実現されている事に気づきます。

世の中にはたくさんのプロセス間通信の仕組みやその上の分散オブジェクトの仕組みがありますが、分散オブジェクトシステムの為だけに作られたプロセス間通信と、そのプロセス間通信だけで

動く分散オブジェクトシステム、というのは珍しいのではないでしょうか。少なくとも私は Binder しか知りません。

binder ドライバはプロセス間通信の仕組みとしては、TCP/IP や UDP のような本格的なプロトコルスタックを持つ物と比較すると、かなりシンプルな物と言えます。一方で相手のスレッドやリファレンスカウントといったものをサポートしているという点で、パイプなどの原始的な仕組みに比べるとやや複雑と言えます。多くの分散オブジェクトでは IPC のレベルではスレッドやリファレンスカウントの概念を持たない物が多いので、その上の分散オブジェクトのシステムでその仕組みを持つ事になります。一方 Binder は IPC のレベルでそのサポートを持つので、上のオブジェクトシステムが複雑になるのを防いでいます。

また、デバイスドライバとして直接カーネルモードで動くように実装されているのも特徴的です。相手のスレッドを起こすのも、カーネルのタスク構造体に直接アクセスして、まるで通常のブロック型デバイスのファイル read と同じように起こす為、単純明快です。カーネルのデータ構造に直接アクセスできるため、相手の uid を直接参照したり、ファイルディスクリプターテーブルを書き換えたり、と言った事も簡単に出来ます。

uid を正確に把握している、というのは、地味ですがリモートにも送る事が出来るメッセージング機構だと素直には作りづらい所があります。uid というのはシステムローカルな物だからです。ですがシステム内でしか使われない事を前提としている Binder では変に抽象化せずに直接 uid を使う事が出来ます。

4 章でも触れた通り、Android はアプリのプロセスを別々の uid に設定する事でセキュリティを確保している為、uid を調べる事は頻繁に発生します。カーネルモジュールは current という参照を通じて呼び元のプロセスの情報にアクセス出来るので、カーネルモジュールのドライバとしてプロセス間通信を実装するなら、呼び元のプロセスの uid を調べる、という機能は、ストレートに実装出来ます。

3.2 8.3.2 binder ドライバの使い方

binder ドライバの実装の細かい話に入る前に、実際に binder ドライバを使用するコードの全体像がどうなるかを見てみましょう。binder ドライバは /dev/binder というファイルとして存在していて、binder ドライバを使うプロセスはこれを open したり ioctl したりします。

binder ドライバはプロセス間通信の仕組みです。何かしらのデータをプロセスを越えて送る物、と言えます。送る物の詳細は後に回して、ここでは data というデータで長さが len の物を送る、という前提でのコードを見てみましょう。データ送信の単純化したコードの全体像を示すと以下のようになります。

リスト 3.1: binder ドライバを用いた呼び出し全体

```
// 1. binder ドライバをオープン
fd = open("/dev/binder", O_RDWR);

// 2. binder ドライバをメモリ領域に mmap。サイズは 128K Bytes
mmap(NULL, 128*1024, PROT_READ, MAP_PRIVATE, fd, 0);
```

```
// 3. read と write に使う引数の初期化。read と write は同時に使える
struct binder_write_read bwr;

// write 関連初期化
bwr.write_size = len;
bwr.write_consumed = 0;
bwr.write_buffer = (uintptr_t) data;

// read 関連初期化、今回は read はしないので 0 を入れておく。
bwr.read_size = 0;
bwr.read_consumed = 0;
bwr.read_buffer = 0;

// 4. binder ドライバの ioctl 呼び出し
res = ioctl(fd, BINDER_WRITE_READ, &bwr);
```

上記のコードはデータを送信するのに必要な全コードを最初から最後まで書いています。個々のブロックの意味についてはこれから説明していきますが、まずは全体がこんな感じになる、という事を見てください。

上記のコードにもあるように、binder ドライバを使用する典型的な手続きとしては以下のようになります。

1. open("/dev/binder", O_RDWR) を呼び出す
2. 1 で得られた fd を mmap する
3. binder_write_read という構造体に送りたいデータを指定して BINDER_WRITE_READ で ioctl

実用の場面では、open と mmap はプロセスの初期化で一回行い、以後は ioctl の所だけを繰り返し呼び出して通信を行います。上記コードのコメントで言う所の 1 と 2 はプロセスの初期化の所で一回行うだけ、3 と 4 はメッセージの呼び出しの都度行うという事です。

ioctl は送信と通信をどちらも担当します。この ioctl を呼び出したあとに、bwr の write_consumed や read_consumed の値を見て、書き込み、読み込みのどちらが処理されたかを判断します。

以下では上記のそれぞれのコードについての詳細を説明していきたいと思います。

3.3 8.3.3 binder ドライバの open と mmap

zzz このパラグラフはボツ

ユーザープロセスのメモリ空間はそれぞれ異なる為、通常の方法で他人のプロセスのデータを触ったりコードを呼んだりは出来ません。そこでカーネルの力を借りる必要が出てきます。プロセスが切り替わってもカーネルのメモリ空間はそのままの為、カーネル空間にデータを書けば、それを別のプロセス空間でもアクセスする事が出来ます。ですがユーザープロセスは直接はカーネルのメモリはアクセス出来ない為、何らかのシステムコールが必要となります。

binder ドライバを使うプロセスは、まずデバイスファイルである /dev/binder ファイルを open します。

リスト 3.2: binder ドライバのオープン

```
// 1. binder ドライバをオープン  
fd = open("/dev/binder", O_RDWR);
```

open はファイルディスクリプタを返します。以後の操作はこの open で返ってくるファイルディスクリプタに対して行います。

binder ドライバを使う時には、open した後にこのデバイスファイルを mmap する事になります。

リスト 3.3: binder ドライバを mmap

```
// 2. binder ドライバをメモリ領域に mmap  
mmap(NULL, 128*1024, PROT_READ, MAP_PRIVATE, fd, 0);
```

mmap はフラグが多いので全てを説明はしませんが、ポイントとなる一番目と二番目の引数だけ簡単に説明しておきます。

先頭の引数はユーザーのアドレス空間のどこにマップするかを指定します。NULL を指定するとカーネルが勝手に選びます。

二番目の引数はマップするサイズです。ここでは 128K Bytes の範囲を mmap するように指示しています。

普通 mmap はファイルの中身をメモリ空間にマップする為の API です。ですが mmap の呼び出しは内部ではドライバに処理が委譲されていて、ドライバごとに違う処理を行う事も出来ます。実際 binder ドライバはただファイルの中身をメモリにマップしている、というのとは、だいぶ異なる挙動をしています。

binder ドライバファイルを mmap すると、内部でドライバはカーネル空間に指定されたサイズくらいの送受信用のバッファを確保します。そしてそのカーネル空間に割り当てたメモリと同じ物理メモリを、ユーザー空間にもマップします。

そしてこのカーネル空間にマップされたメモリは、以後ドライバ側で送受信に使われます。ユーザー空間にマップされている領域は、ioctl の戻りなどで使われていますが、直接コードの中でそのアドレスをあらわに参照する事はありません。

何故こういう事が必要か、というのは、少しカーネルのコンテキストスイッチに慣れていないと想像しにくいくかもしれません。仮想メモリから実メモリへの参照、というのは、基本的にはハードウェアで自動的に行われています。カーネルはハードウェアに仮想メモリと実メモリの対応表をセットしたり、といった操作はしますが、実際にカーネルなどのソフトウェアが仮想メモリのアドレスから実メモリをたどって値を読んだりはしません。

ドライバにとっても、アクセス出来るメモリは現在のプロセスのメモリ空間とカーネルのメモリ空間だけなのです。そのどちらでも無い、例えば送信先のプロセスのメモリを参照する方法はありません。

そこで送信先プロセスにどのようにデータを渡すか、というと、送信先プロセスが「現在のプロ

第3章 {syscall} binder ドライバを扱う三つの3.3.3 binderopen, mmapとmmap

セス」の時に、送信先プロセスのメモリ空間とカーネルのメモリ空間で同じ物理メモリを指すようにマップするのです。この時は送信先プロセスがまだ現在のプロセスなので、こういう操作がドライバから可能です。

そして送信元のプロセスにコンテキストスイッチしてしまっても、このカーネル空間のメモリに書きこんでおけば、後で送信先のプロセスにコンテキストスイッチした時も、ユーザー空間にコピーする事無くそのまま参照出来ます。コンテキストスイッチした時にマップされる場所をカーネル側にも置いておく事で、コンテキストスイッチせずにコンテキストスイッチした後のメモリ空間にデータを置いて置ける訳です。

8.3.3 ①

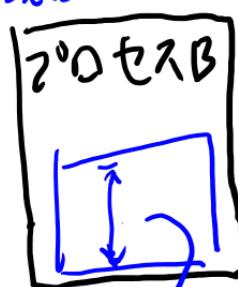
現在のプロセス



カーネル

現在のプロセスのメモリとカーネルのメモリに触る事は出来ない。
プロセスAがプロセスBにデータを送る時、この状態で、プロセスBのメモリには触れない。

現在のプロセス

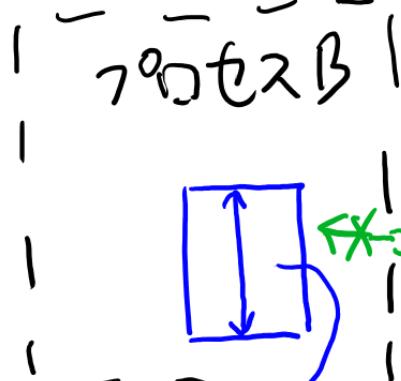


ここで「現在のプロセスがBの時に、同じ物理Xモリのブロックをカーネル空間とプロセスBのメモリにマップしておきます。(この時はプロセスBのメモリに触れる)

物理Xモリ

カーネル

現在のプロセス



左側のメモリは触れないが、

²⁴
物理Xモリ

カーネル

左側から書き

3.4 8.3.4 binder ドライバの ioctl と読み書き

Binder を用いたメッセージの送受信には、ioctl システムコールを使います。ioctl については「 [ioctl システムコール](#)」を参照ください。

メッセージの送受信に使うリクエスト ID は BINDER_WRITE_READ です。呼び出しは以下のようないコードとなります。

リスト 3.4: binder ドライバの ioctl

```
// 4. binder ドライバの ioctl 呼び出し
res = ioctl(fd, BINDER_WRITE_READ, &bwr);
```

三番目の引数の**&bwr** というのは、送受信に使う構造体、binder_write_read のポインタです。BINDER_WRITE_READ は送信と受信を一度に出来る API となっている為、

binder_write_read には読み取り関連の設定と書き込み関連の設定を行う事になっています。書き込み関連は以下のようにバッファと長さ、そして現在どこまで書いたかを表す consumed を初期化します。

リスト 3.5: binder_write_read の write 側の初期化

```
struct binder_write_read bwr;

// write 関連初期化
bwr.write_size = len;
bwr.write_consumed = 0;
bwr.write_buffer = (uintptr_t) data;
```

ここで data はドライバに送りたいデータの入ったポインタ、len はそのデータの長さです。読み込み関連は読み込みに使うバッファとそのサイズですが、読み込みをしない場合は read_size に 0 を入れておきます。

リスト 3.6: binder_write_read の read 側初期化

```
// read 関連初期化、今回は read はしないので 0 を入れておく。
bwr.read_size = 0;
bwr.read_consumed = 0;
bwr.read_buffer = 0;
```

このように初期化した bwr を ioctl に渡します。再掲すると以下のコードです。

リスト 3.7: binder ドライバの ioctl 再掲

```
// 4. binder ドライバの ioctl 呼び出し  
res = ioctl(fd, BINDER_WRITE_READ, &bwr);
```

このように ioctl を呼ぶ事で、ドライバにデータを送ったり、逆にドライバからデータを受け取ったり出来ます。

以上で 8.3.1 で示した、binder ドライバを使った API 呼び出しの標準的なコードについて、一通りの説明を行いました。

しかし、ここまで説明では、bwr の read_buffer の中身や write_buffer の中身、つまりドライバにどういう種類のデータを書き込んで、ドライバからはどういう種類のデータが読み取れるのか、という話は一切していません。

binder ドライバと送受信するデータの中身は、bwr の write_buffer や read_buffer の中でさらに決まりがあります。今回の例で言うと bwr.write_buffer に渡しているデータの中で、さらに決まりがあるのです。

以後ではこの送受信のデータの詳細から、binder ドライバという物の動きを具体的に見ていきましょう。

第 4 章

{driver_message} 8.4 binder ドライバによるメッセージの送受信 - servicemanager とサービス

前節では、binder ドライバを扱う基本的なシステムコールとなる、open, mmap, ioctl について簡単に見ました。このうち、ioctl については実際に何を送受信するのか、という所の話はしていません。本節ではこの ioctl でやり取りするメッセージの内容について見てきます。

メッセージを見ていく時には、そのメッセージのやり取りの相手、という物が登場します。この相手がサービスです。

そこで本節ではメッセージの内容とサービスの呼び出しについて見ていきます。その過程で重要で特別なサービスである、servicemanager についても扱います。

4.1 8.4.1 ドライバに書きこむデータの入れ子構造

ドライバに書きこむデータは binder_write_read 構造体のデータだ、と言いました。この構造体の中にさらに BC_TRANSACTION の時には binder_transaction_data 型のデータが入り、その中にさらに flat_binder_object が入ります。このようにデータは入れ子になっていて、しかも構造は全て似ています。そのデータの種類があって、さらにバッファのポインタとその長さ、というのがパターンです。

通信にまつわる説明ではありがちな構造ですが、この手の説明を読むのに慣れてないと、毎回同じような事を言っている別の型が乱立していて、何の話をしているのか良く分からなくなる、という問題が発生します。

そこで個々のデータ型の説明をする前に、まずは全体の入れ子構造をここでしめしておきたいと思います。binder_write_read、binder_transaction_data、flat_binder_object は、以下のような関係にあります。

8.4.1 ①

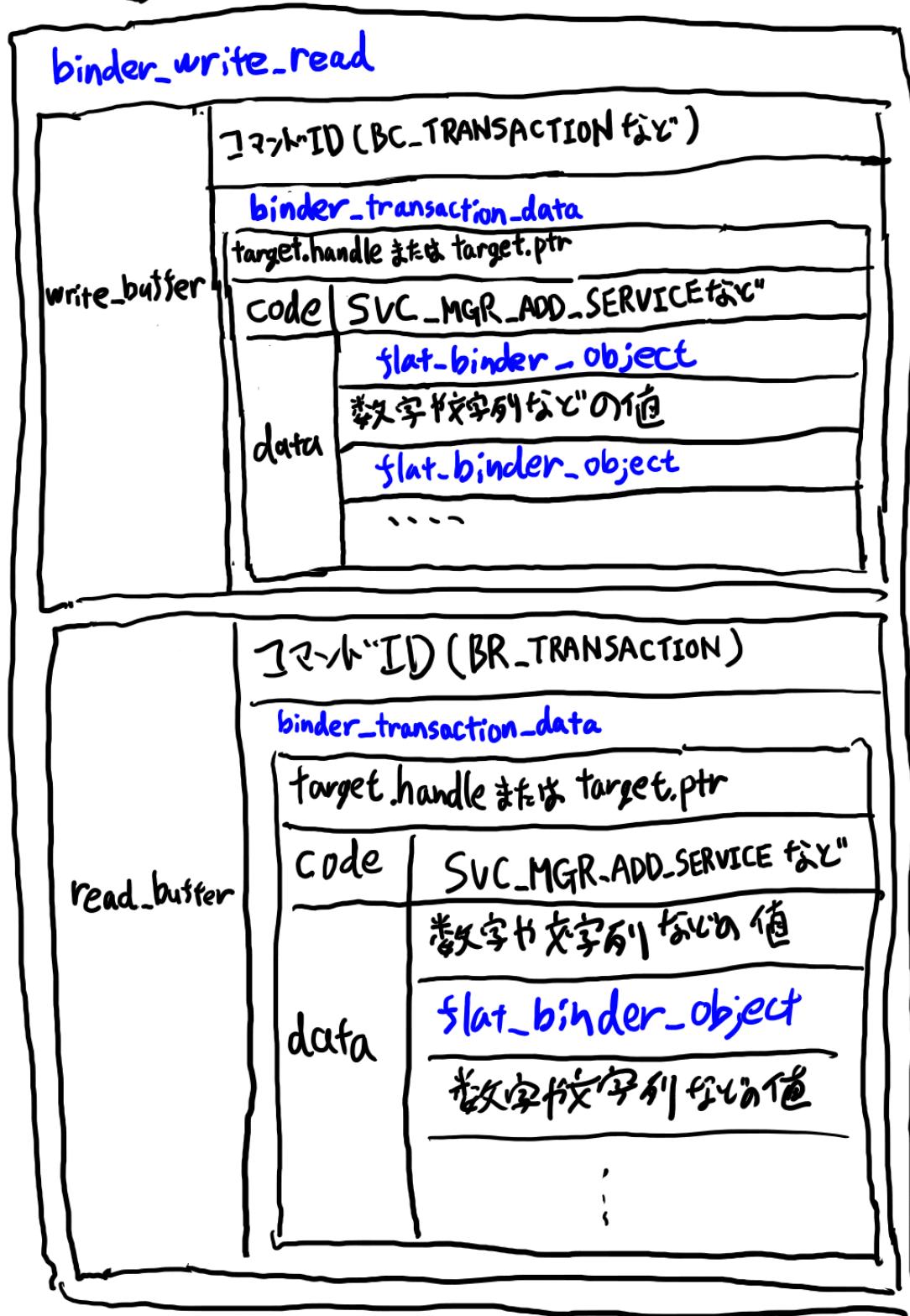


図 4.1 binder_write_read、binder_transaction_data、flat_binder_object の包含関係

以下の節でそれぞれの関係が良く分からなくなってきた時には、この図に戻ってきてみてください。

4.2 8.4.2 ドライバに書き込むデータのフォーマットとコマンド ID

ioctl を使って読み書きするデータは binder_write_read 構造体だと言いました。(8.3.4) 例えは以下のようなコードで初期化していました。

リスト 4.1: binder_write_read の write 側初期化、再掲

```
struct binder_write_read bwr;

// write 関連初期化
bwr.write_size = len;
bwr.write_consumed = 0;
// /* 1 */
bwr.write_buffer = (uintptr_t) data;
```

ここで /* 1 */ で data という変数のデータを詰めています。このデータの中身についての話をします。

このデータの先頭は、コマンド ID となっています。そして各コマンド ID に応じてそのあとに続くデータが決まります。

良く使うコマンド ID には以下の物があります。

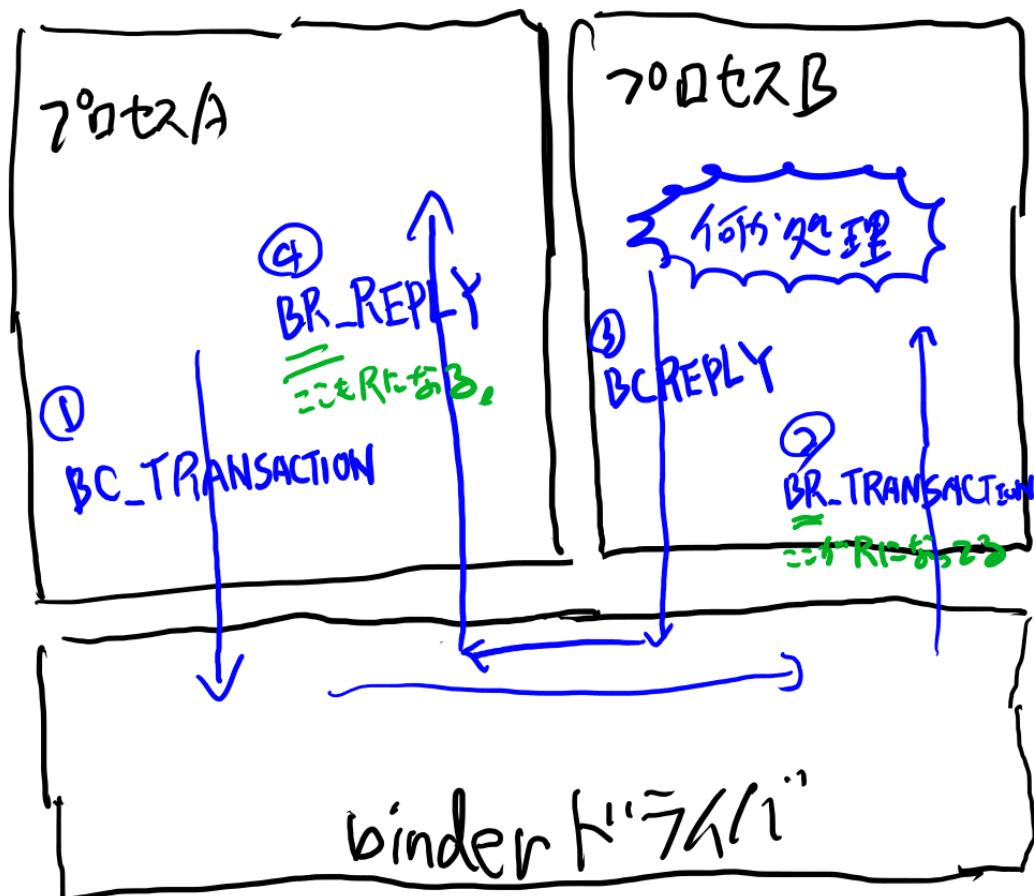
1. BC_TRANSACTION
2. BC_REPLY
3. BC_ACQUIRE
4. BC_RELEASE

一番大切なのが 1 の BC_TRANSACTION です。これはよそのプロセスのメソッド呼び出しを行う時に使用するコマンドです。2 は上記のメソッド呼び出しに対する返信のコマンドです。

また、書き込む時と、書き込んだデータを読み込む時でコマンド ID の接頭辞が変わります。具体的には BC_TRANSACTION を書き込むと、サービス側で読みだす時には BR_TRANSACTION という ID になりますし、BC_REPLY は BR_REPLY に、BC_ACQUIRE は BR_ACQUIRE になります。

BC_TRANSACTION して BC_REPLY が返る様子を図にすると、以下のようになります。これが一つのメソッド呼び出しに対応します。

8.4.2①



トライバに送る時はいつも「BC-」で
始まる。トライバから読む時はいつも「BR-」で
始まる。またBC_TRANSACTIONに対する返答はBC_REPLY

図 4.2 BC が BR になり、TRANSACTION に REPLY が返る

コマンドの下二つに話を戻すと、BC_ACQUIRE と BC_RELEASE はハンドルのリファレンスカウントを上げたり下げたりする時に使います。ハンドルとは、メソッドを呼び出す相手や呼び手を表す物です。サービスのインスタンスの ID と言えます。ハンドルさえあれば、binder ドライバは対応するサービスのポインタを探す事が出来ます。

Binder の特徴の一つとして、最初から C++ やオブジェクト指向を前提としている、という物があります。そこでリファレンスカウントによるオーナーシップやメソッドという物が最初からある程度下のレイヤにも組み込まれています。具体的に言うとハンドルというインスタンスを指し示すものがあったり、そのリファレンスカウントが、binder ドライバのレベルで存在している、という事です。

それでは以下、BC_TRANSACTION について詳しく見ていきましょう。

4.3 8.4.3. BC_TRANSACTION コマンドと binder_transaction_data

BINDER_WRITE_READ で ioctl を呼び出す時に書き込むデータのうち、先頭が BC_TRANSACTION コマンド ID の場合、

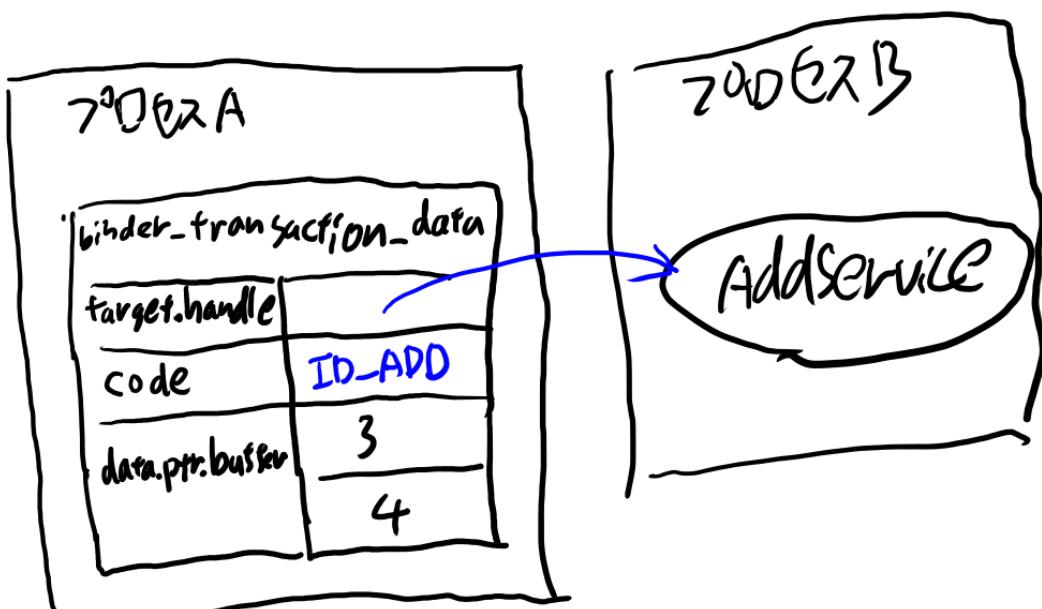
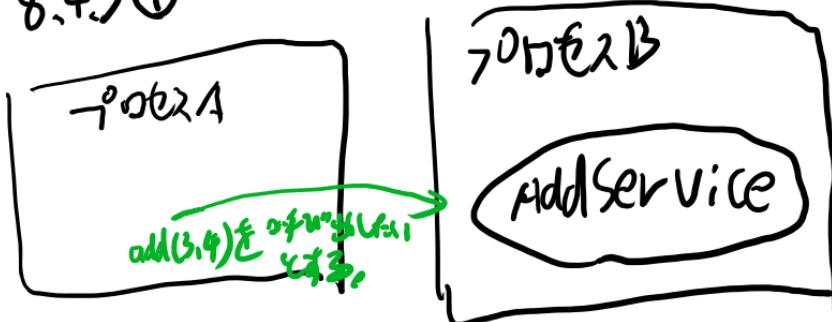
送るデータの中身は、コマンド ID の後に binder_transaction_data が続く事になっています。長さは sizeof(struct binder_transaction_data) です。

binder_transaction_data はメンバの多い構造体で全部説明するのは大変ですが主要な物だけ抜き出すと以下になります。

表 4.1 binder_transaction_data のフィールド

target.handle	やりとりをする相手を表すハンドル
code	どのメソッドかを表す ID
data.ptr.buffer	メソッドの引数のデータ領域をさすポインタ
data_size	メソッドの引数のデータの長さ

8.4.3 ①



binder_transaction_data はこうなる。

target.handle : 呼び出し元サービスのハンドル

code : 呼び出すドリ、トを表す int 値。サービスが決まる。

data.ptr.buffer : 引数が入る。add(3,4)を呼び出したいなら 3 と 4 がバイト配列に入る。

図 4.3 binder_transaction_data とメソッド呼び出し

4.3 8.4.3. BC_TRANSACTION コマンドと binder_transaction_data

メソッドを呼び出す相手、どのメソッドかを表す ID、そして引数のデータ (data.ptr.buffer と data_size はセットでデータ)、という事で、これだけあると相手のメソッドを呼ぶ事が出来るのが分かると思います。

8.4.3 ②

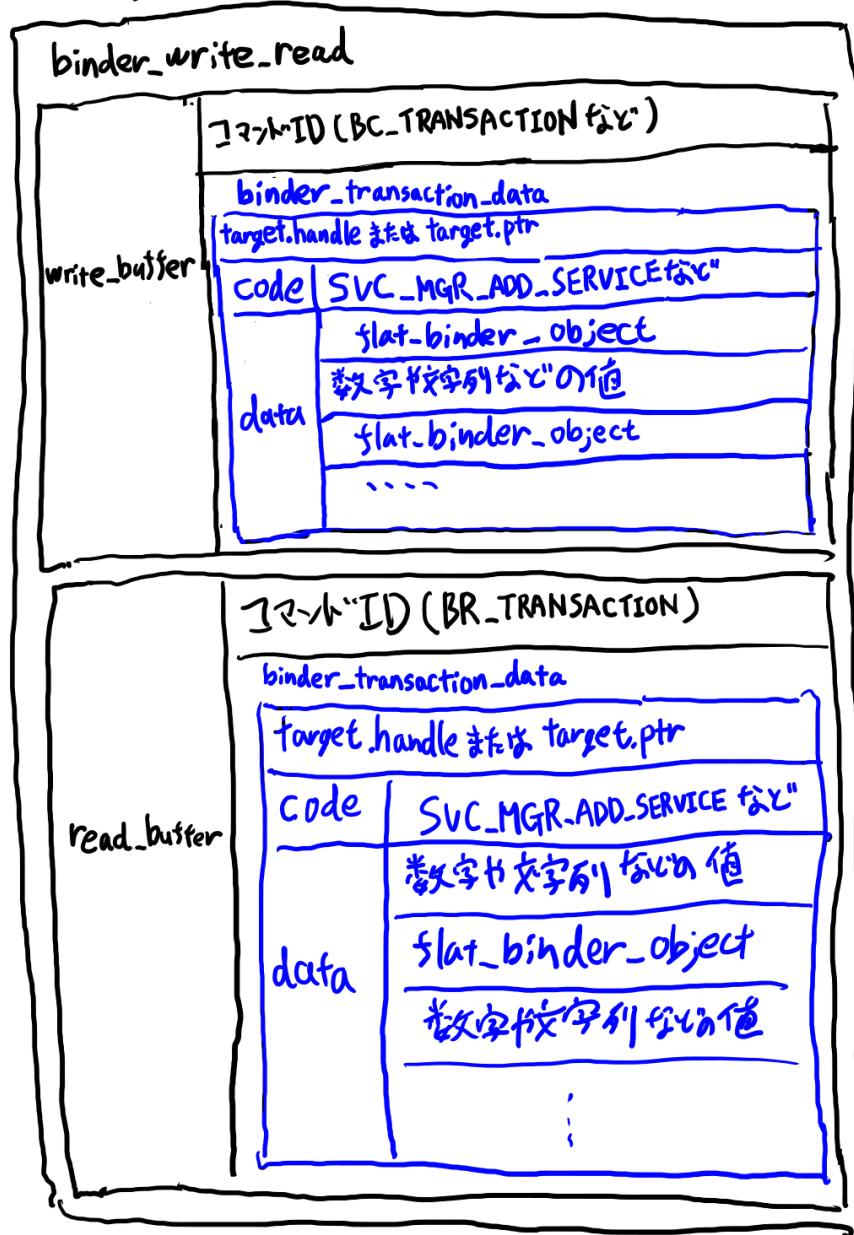


図 4.4 binder_write_read と binder_transaction_data の関係

引数のデータは先頭からベタにバイナリ値が書かれています。

書いた方と読んだ方で対応がとれていれば正しい値が取れる、という形式です。Parcel という utility クラスがシリализとデシリализに使えますが、ただ生のデータを読み書きするだけ

です。

code はメソッドを表す ID で、各サービスが勝手に決めます。

さて、あと表 4.1「binder_transaction_data のフィールド」の中で残っているのは target.handle だけです。この target.handle さえ分かればメソッドを呼び出す事が出来ます。そこで登場するのが servicemanager です。

4.4 8.4.4 servicemanager によるサービスハンドルの取得

分散オブジェクトのシステムではオブジェクトを表す何らかの名前から、そのリファレンスを取る所が重要となります。一般にはそれをネーミングサービスと言いますが、Android の場合そのネーミングサービスに相当するのが servicemanager です。

servicemanager は特別なサービスです。サービスなので、上記の BC_TRANSACT を送る事によって servicemanager のメソッドを呼び出す事が出来ます。ここまででは通常のサービスと変わりません。servicemanager が特別なのは、ハンドルが 0 番である事が最初から決まっている事です。

ですから、全てのクライアントは、最初から servicemanager のハンドルは知っている事になります。target.handle に 0 を代入すれば、それは servicemanager へのメソッド呼び出しだ、と解釈されます。こうして、どこからかハンドルを取得しなくとも、servicemanager へだけは BC_TRANSACT を送る事が出来ます。

servicemanager のメソッドのうち、良く呼び出す物のメソッド ID の一覧を以下に載せます。

1. SVC_MGR_ADD_SERVICE
2. SVC_MGR_CHECK_SERVICE

1 がサービスとして登録するメッセージです。詳細は後述します。

2 の SVC_MGR_CHECK_SERVICE で、サービスを名前で検索出来ます。SVC_MGR_GET_SERVICE というメッセージもあって同じ処理をしていますが、SVC_MGR_CHECK_SERVICE を使っているようです。

SVC_MGR_CHECK_SERVICE の引数としては、"android.os.IServiceManager" という文字列と検索したいサービス名の二つです。

例えば SurfaceFlinger サービスのハンドルを検索したい場合の binder_transaction_data の作り方は以下のようになります。(重要な所だけ抜き出しています) 簡単のため Parcel というシリアルライザを使いますが、特に説明しなくともコードから何をやってるかは想像出来るでしょう(詳細は 8.6.2 でも扱います)。

リスト 4.2: Parcel を用いた binder_transaction_data の作り方

```
// binder_transaction_data のうち引数の所のデータを作る。  
// 作りたいのはベタのデータを書き込んだバイナリ配列だが、Parcel ユーティリティクラスを使って作る。  
Parcel writeData;  
  
// servicemanager のインターフェース名。ハードコードされた文字列  
writeData.writeString16(String16("android.os.IServiceManager"));  
  
// 探したいサービスの名前
```

```
writeData.writeString16(String16("SurfaceFlinger"));

// writeData が完成したので、次は binder_transaction_data を作る。
// BC_TRANSACTION で送るデータ。
struct binder_transaction_data tr;

// servicemanager のハンドルは 0 にハードコード
tr.target.handle = 0;

// 呼び出すメソッドの ID。
tr.code = SVC_MGR_CHECK_SERVICE;

// 引数には上で作った writeData を設定
tr.data_size = writeData.dataSize();
tr.data.ptr.buffer = writeData.data();
```

この binder_transaction_data 構造体のデータを binder ドライバに送りつけて、結果を取得すると、SurfaceFlinger サービスのハンドルが得られます。

このように、servicemanager という特別なサービスはハンドルが 0 と固定の値になっているので、クライアントのコードは最初から servicemanager に対してだけはメソッドを呼び出せます。そしてその servicemanager が全サービスの一覧を持っていて、その servicemanager にハンドルの検索を頼む訳です。

4.5 8.4.5 SVC_MGR_CHECK_SERVICE を例に、ioctl呼び出しを復習する

以上で一通り servicemanager のメソッド呼び出しの解説を終えたのですが、復習も兼ねてこの binder_transaction_data を実際に送信するまでのコードも見てみましょう。内容としては 8.3.4 と同じ内容となります。以下、重要なコードだけを抜粋していきます。

まずは binder_transaction_data は BC_TRANSACTION コマンドで送信するでした。BC_TRANSACTION コマンドの送受信には binder_write_read 構造体を使い、これを ioctl に渡すでした。

そこで binder_write_read 構造体を用意します。まずは送信用のデータから。これはコマンド ID BC_TRANSACTION と、先ほどの binder_transaction_data をバイト列に書き込んだ物になります。

リスト 4.3: binder_transaction_data を binder_write_read にセットする

```
// 送信用のデータのバッファ
byte writebuf[1024];
*((int*)writebuf) = BC_TRANSACTION
memcpy(&writebuf[4], &tr, sizeof(struct binder_transaction_data));
```

8.4.5 ①

writebuf

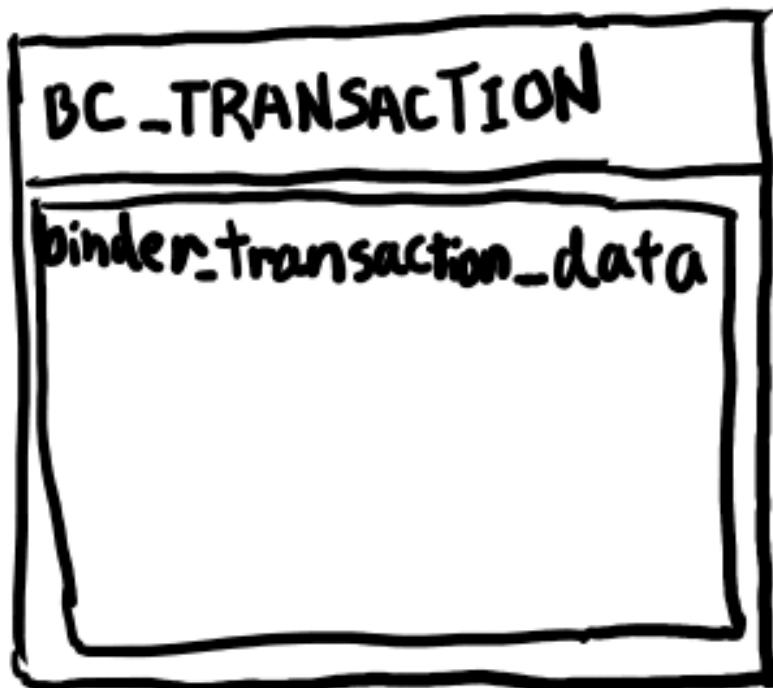


図 4.5 writebuf の中身

binder_transaction_data にはユーザー領域のデータへのポインタ、data.ptr.buffer が含まれるのですが、これはドライバ内で allocate してコピーしてくれます。

あとはこの送信用のデータを binder_write_read に設定して、受信用にはデータを受け取るバッファを設定し、ioctl を呼びます。

リスト 4.4: ioctl で結果を受け取る

```
// 結果受け取りのバッファ
byte readbuf[1024];

struct binder_write_read bwr;

// 送信用データ。長さはコマンド ID のサイズ +binder_transaction_data のサイズ
bwr.write_size = writebuf;
bwr.write_buffer = sizeof(int)+sizeof(struct binder_transaction_data);

// 受信用バッファ
bwr.read_size = 1024;
bwr.read_buffer = readbuf;

// ioctl で servicemanager の SVC_MGR_CHECK_SERVICE を呼び出す
res = ioctl(fd, BINDER_WRITE_READ, &bwr);
```

8.4.5②

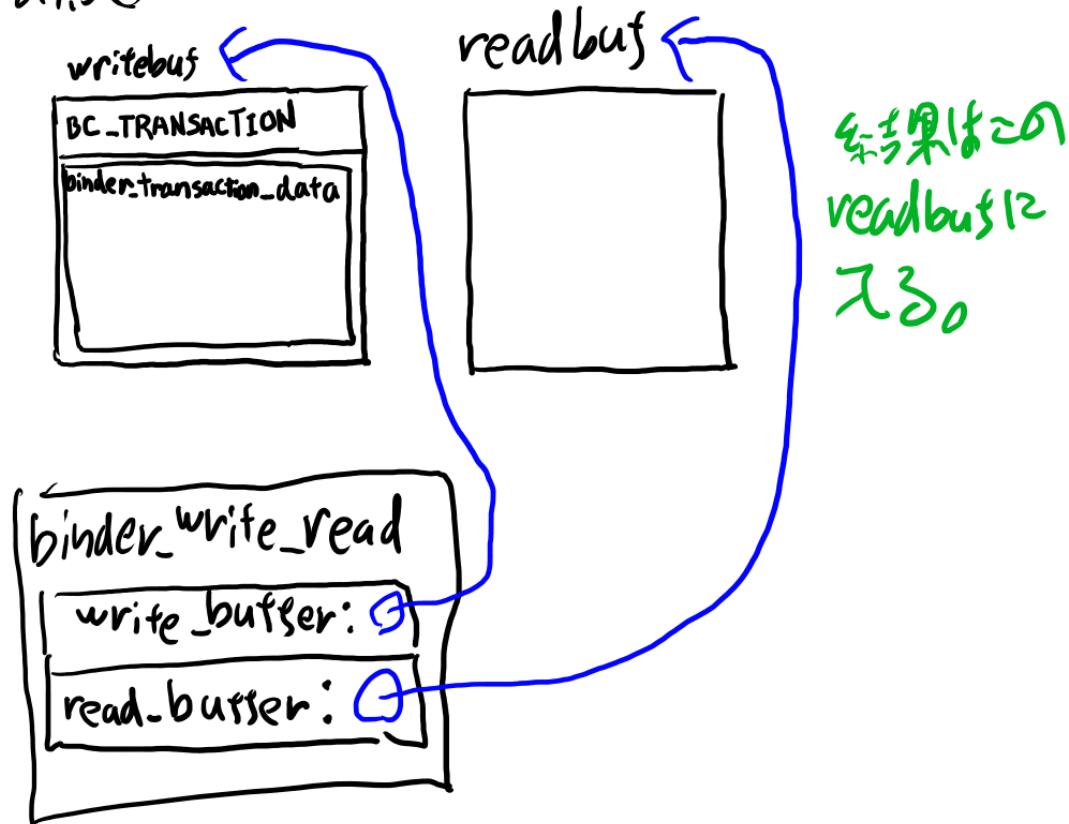


図 4.6 `binder_write_read` の `writebuf` と `readbuf`

最初の `writeData` が `binder_transaction_data` に入り、それが `writebuf` に入って

buffer_write_read 構造体に入る、という 4 段階の入れ子になっているのでややこしいですが、一つ一つの処理はかなり単純です。

このコードを実行すると、binder ドライバはこのスレッドを一旦止めて、servicemanager のスレッドを起こしてこの binder_transaction_data を渡して処理させ、その結果を受け取ってから元の呼び出しのスレッドを起こして ioctl から返ります。

結果は bwr.read_buffer に入ります。

では次にこの結果がどういう物か、典型的な処理を見る事で見ていきましょう。

4.6 8.4.6 サービスハンドルの取得の結果 - メッセージ受信

ioctl を用いたメソッド呼び出しの結果は、binder_write_read 構造体の read_buffer に書かれます。書かれたデータの長さは bwr.read_consumed に入ります。

read_buffer には先頭の 4 バイトに戻りのコマンド ID が、それ以後にそのコマンドの付随データが入ります。送信の側と同じですね。

BC_TRANSACTION の結果が正常に返る場合のコマンドは、BR_REPLY と決まっています。その後に付随するデータは binder_transaction_data で、送信時と同じです。

そしてその binder_transaction_data の data.ptr.buffer の中には flat_binder_object 構造体というのが入っています。

この構造体はサービスのハンドルやサービスのプロセスの場合はサービス自身のポインタ、そしてファイルディスクリプタなどを保持できるオブジェクトです。

今回のケースでは、この構造体の中にハンドルが入っています。

8.4.6 ①

readbuf

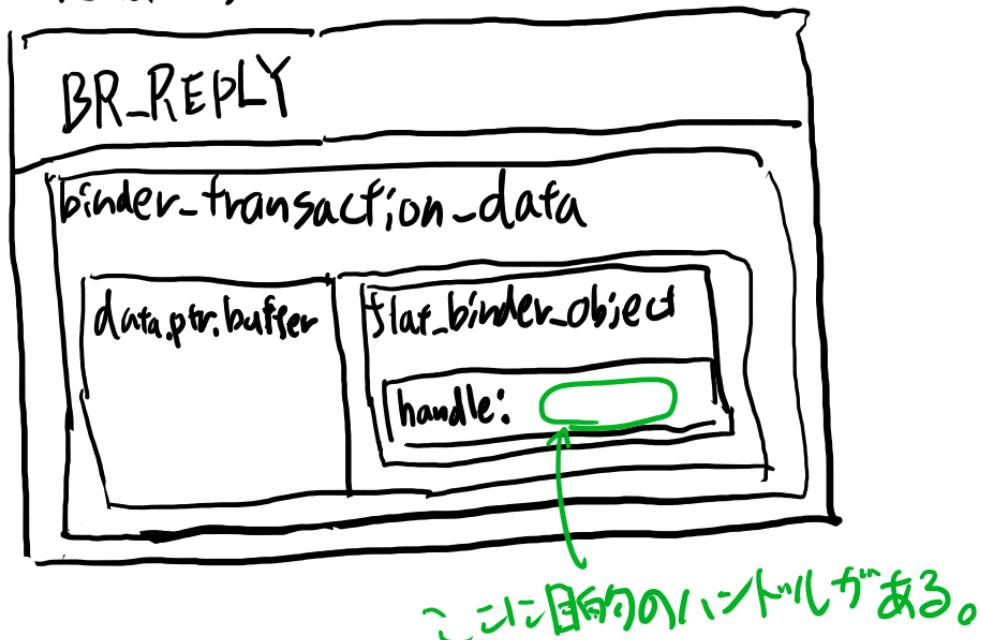


図 4.7 読み出したバッファの構造

8.4.5 のコードの続きとしては以下のようなコードでこのハンドルが取れます。

リスト 4.5: 読みだしたバッファから binder_transaction_data を取り出す

```
// /* 1 */ 先頭 4 バイトはコマンド ID
int cmd = *((int*)readbuf);

// BC_TRANSACTION のリプライは正常時は BR_REPLY
assert(cmd == BR_REPLY);

// /* 2 */ BR_REPLY の後続データも binder_transaction_data 型
struct binder_transaction_data tr_res;
memcpy(&tr_res, &reabuf[4], sizeof(struct binder_transaction_data));

// /* 3 */ binder_transaction_data 型には flat_binder_object のデータが入っている
struct flat_binder_object *obj;
obj = (struct flat_binder_object*)tr_res.data.ptr.buffer;

// /* 4 */ handle は flat_binder_object の中の handle フィールドに入っている
int handle = obj->handle;
```

少し細かいコードになりますが、このように

1. コマンドを取る（今回のケースでは使いませんが）
2. binder_transaction_data を取り出す
3. その中から flat_binder_object を取り出す
4. その中の handle フィールドに目的のサービスハンドルが入っている

という手順になります。なお、/* 3 */で flat_binder_object という物が登場しましたが、これについては次節で詳細に扱います。

こうして目的のサービスのハンドルを取得したら、以後はこのようにして得たハンドルに対して 8.4.4 で説明したのと同様なコードで、指定したサービスのメソッドが呼び出せます。

ここで解説したコードはサービスのハンドルの取得時の受信した後のコードですが、メッセージ受信全般でほとんど同じ構造のコードとなります。

ioctl は binder_write_read 構造体の write_size を 0 にして呼び出すと、呼び出し時点でブロックしてメッセージが来るのを待ちます。

サービスの実装側では、この受信としての ioctl 呼び出しでブロックしてメッセージが来るのを待ち、メッセージがやってくるとこの ioctl から帰ってくるので binder_write_read の read_buffer から、先ほど解説したサービスハンドルの取得と同じような手順でコマンド ID を読み出し、コマンド ID に応じた処理を行います。

さて、ioctl を使うコードとしては以上でだいたいの説明が終わりです。以下では実装側に目をうつし、ioctl の内で何が起こっているのかをもう少し詳しく見ていきましょう。

第 5 章

{flat_binderobj} 8.5 binder ドライバ の内側とオブジェクトの送信 - `flat_binder_object`

前節では、servicemanager でサービスを検索する例を通じて、binder ドライバを利用して引数が文字列のみの簡単なメソッドを呼び出す手順を見てきました。

さて、サービスを検索するには、サービスが既に登録されていないといけません。そこで次の話題としては当然、このサービスをどう登録するか、という事に移る訳ですが、サービスを登録するのは、サービスを検索するよりも、一つだけ難しい所があります。それは引数にサービスというオブジェクトが含まれてしまう、という事です。

サービスというオブジェクトを含んだ引数をどのように扱うか、というのが本節の主要なテーマとなります。その為には binder ドライバ内部で様々なデータをどう管理しているのか、という事を知る必要があります。

binder ドライバ内部のデータ管理は、カーネルのメモリ空間で行われるカーネルモードでの話となります。

5.1 8.5.1 オブジェクトを送信すると何が起こるか？

詳細に入る前に、オブジェクトを送信すると何が起こるのか？ という事の概要から始めたいと思います。

オブジェクトというのはメソッドという処理が含まれるので、転送する事が出来ません（少なくとも Binder では転送されません）。ここで例示の為に、オブジェクトの存在しているプロセスをプロセス A と呼び、それを送りつける先をプロセス B とします。

プロセス A からオブジェクトをドライバに渡すと、ドライバがこのオブジェクトのポインタを覚えておきます。そしてプロセス B には、このポインタを表すハンドルを渡します。B にはオブジェクトを転送される訳では無くハンドルが渡されるのですが、ここでプロセス A 内のポインタがドライバに保持される、というのが工夫です。

第5章 {flat_binderobj} 8.5 binder ドライバの内部とオブジェクトを送信するときが起こる問題

ポインタという物は概念的にはそのプロセスのアドレス空間内のアドレスです^{*1}ですからプロセス A のポインタはプロセス A のアドレス空間上でないと有効ではありません。ですがアドレスをドライバが記憶する事は出来ます（そのアドレスの中身を参照するのはちょっと大変ですが）。

そしてプロセス B がハンドルに対してメソッド呼び出しをしたら、それをプロセス A に転送するのですが、この時は記憶したポインタのアドレスも戻してやる訳です。プロセス A の ioctl から戻った時というのは、アドレス空間はプロセス A の物となっているので、このポインタはそのまま有効で、以前送信の時に引数で渡したポインタとなっている訳です。

^{*1} ARM の場合厳密にはそれを参照するためのディスクリプタですが、この場合の議論は等しく有効です。

8.5.1 ①

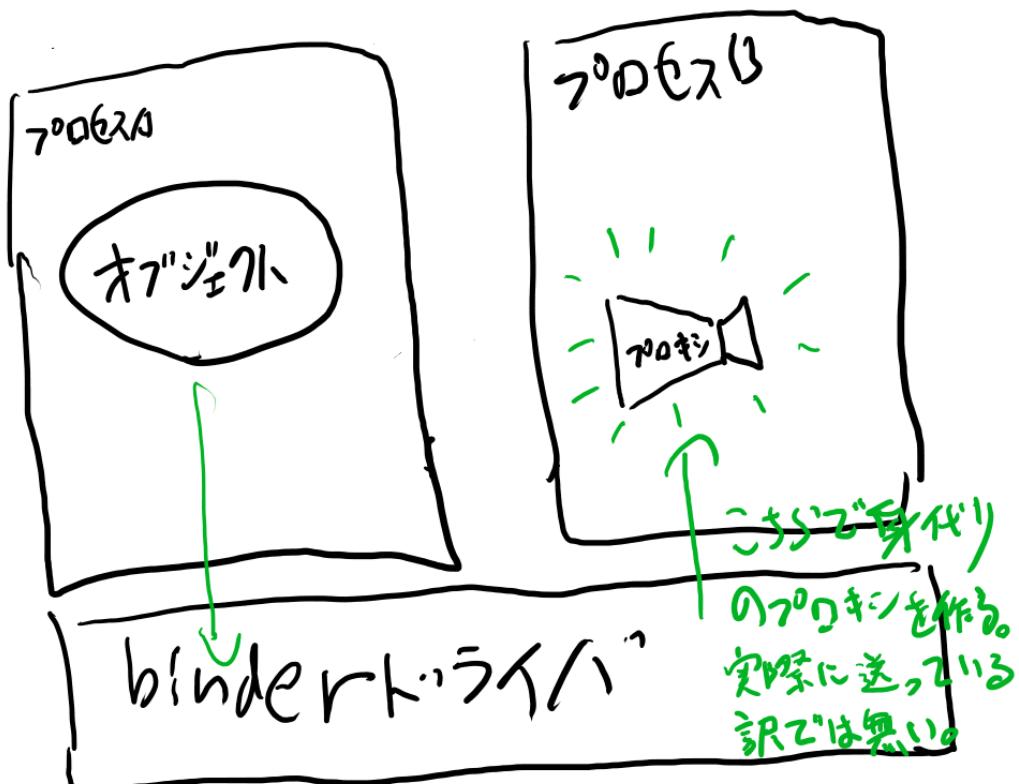
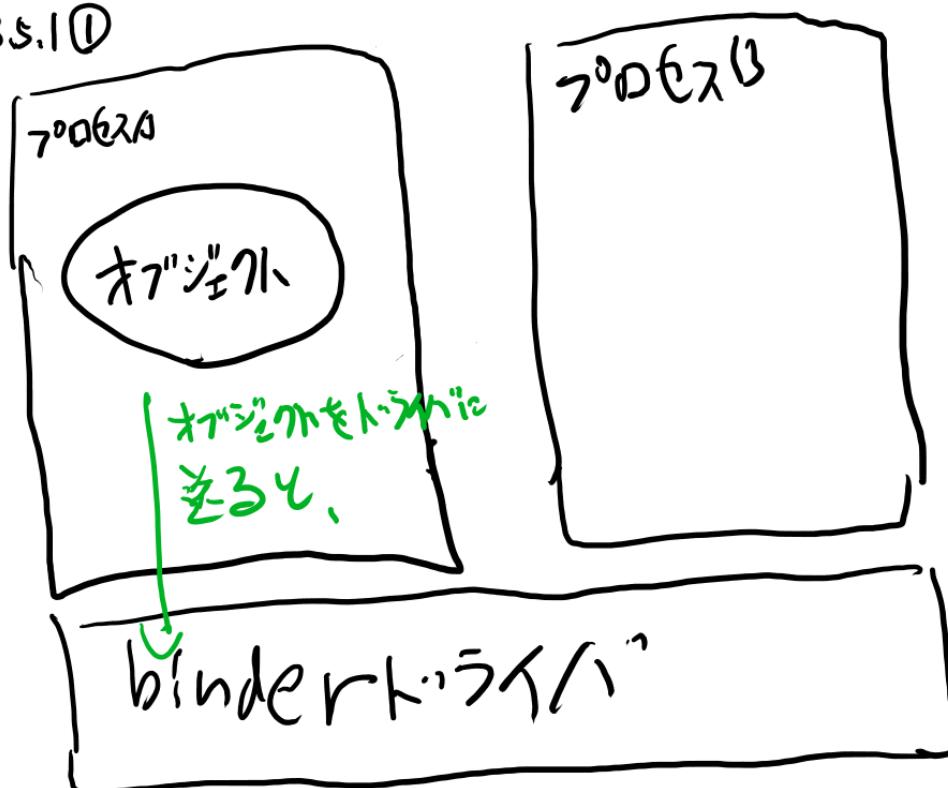


図 5.1 オブジェクトを送ると、プロキシが作られる

8.5.1②

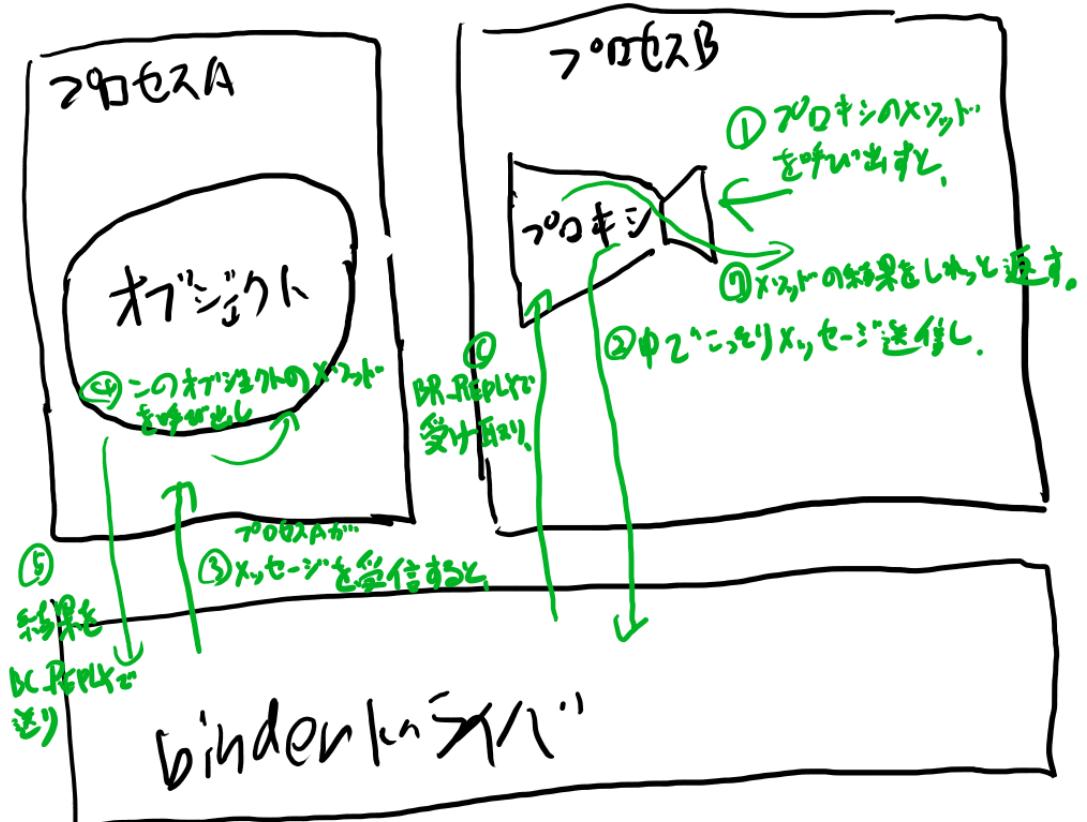


図 5.2 プロキシのメソッドを呼ぶとオブジェクトが呼ばれるメカニズム

つまり、オブジェクトは転送されません。プロセス A からオブジェクトを転送しようとしてもドライバの所で記憶されるだけで、実体はプロセス B には行かないのです。でも身代わりとしてハンドルというのが代わりに届く事になります。

これがオブジェクトを送信する時に起こる事のイメージです。

5.2 8.5.2 スレッドとプロセスのデータ構造 - binder_proc と binder_thread

binder ドライバは、binder ドライバを使用するプロセスに関する情報を、binder_proc というデータ構造で管理しています。これはドライバを open した時にカーネルによって作られる file 構造体のフィールドに格納されます。ユーザープロセスの側から見れば、binder ドライバを open した時に返るファイルディスクリプタに格納されています。ファイルディスクリプタは ioctl 呼び出しの

第5章 {flat_binder} と binder プロセスの内側とデータ構造 - binder の送信 profit と binder_object

第一引数で渡し続けるので、この binder_proc もプロセス内で同じインスタンスが毎回ドライバに渡されます。@#@ TODO: 図と説明でファイルオブジェクトが open の都度作られる前提で書いているが後で本当にそうか確認。

また、スレッドを表すデータ構造もあります。binder はメソッド呼び出しを前提としたプロセス間通信です。メソッド呼び出しが単なるメッセージングと違うのは、結果が返る、という所です。

メソッド呼び出しを成立させるためには、BC_TRANSACTION と BC_REPLY の二つの ioctl 呼び出しが必要です。そして BC_REPLY を送る時には対応する BC_TRANSACTION を送ってきたスレッドに送信しないと、結果が呼び出したスレッドに返りません。つまり、BC_TRANSACTION を受け取る側が処理をしている間、送信元のスレッドを覚えておく必要があります。^{*2}

そこで binder ドライバは、ioctl を呼び出される都度、呼び出しスレッドに対応するデータ構造を作成して管理します。このスレッドを表す構造体は binder_thread という名前です。binder_thread は binder_proc にツリーとして保持されます。^{*3}

^{*2} BC_REPLY の時には target の指定は必要ありません。binder ドライバが自動的に探してくれます。

^{*3} Linux カーネルが提供している赤黒木で保持されます。

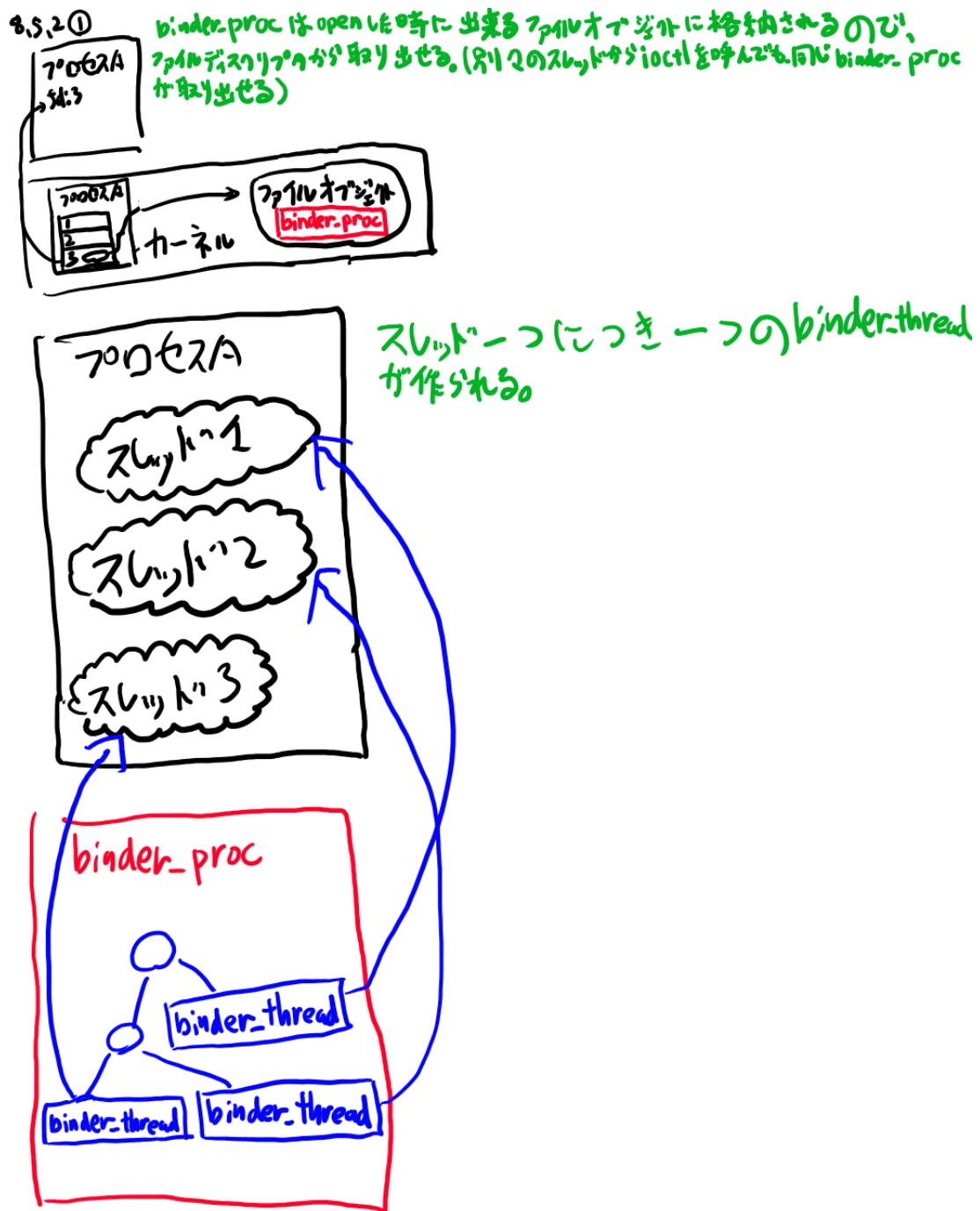


図 5.3 binder_proc と binder_thread

ioctl が呼ばれる都度、呼び出し元のスレッド ID を見て、そのスレッド ID に対応するスレッド構造体が、プロセス構造体に既に入っているかを検索します。無ければ新たに生成してツリーに追加します。このように、スレッド構造体は ioctl を呼び出す都度 lazy に、しかも暗黙に作られます。

こうして、ioctl を呼び出している各スレッドをドライバが管理しているので、

BR_TRANSACTION を呼び出されている側のユーザープロセスが処理している間は、呼び出し元のスレッド情報をドライバが覚えておいてくれるので、その後に BC_REPLY をドライバに送ると、自動的に呼び出し元のスレッドを探し出してそのスレッドに返してくれます。

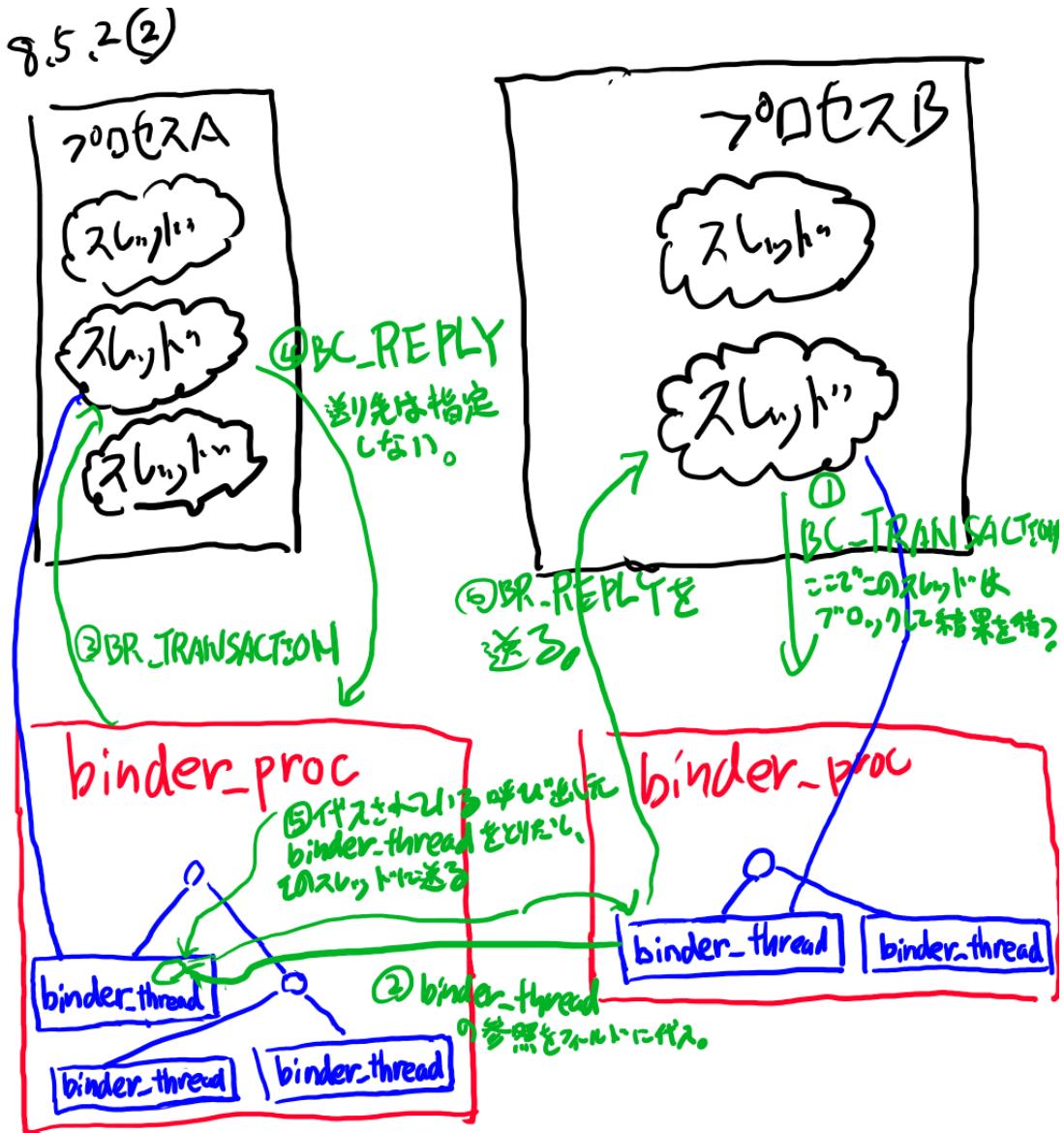


図 5.4 binder_thread に REPLY 先を記憶する

■コラム: プロセス構造体という呼び名と binder_proc

Linux カーネルでは、プロセス構造体と言う物があります。この本以外の本では、プロセス構造体と単に言ったら、この Linux カーネルのプロセス構造体を指すのが普通です。そ

して binder ドライバが持つプロセスのデータを表す構造体は、区別する為に構造体名である binder_proc と呼ぶのが普通でしょう。ですが、本章ではプロセス構造体と言ったら binder_proc を指します。これは、この binder ドライバの周辺にはプロセス構造体の他にも大量の構造体名が出てくる為、覚えてなくてはならない構造体名を一つでも減らす為の工夫です。私も初めてこの周辺を読んだ時は構造体名が多すぎて、すぐにどの構造体名が何だったのかを忘れてしまい苦労しました。幸い、本章では Linux カーネルのプロセス構造体に言及する必要は無いので、どちらか分からなくなる事は無いはずです。もし Linux カーネルに詳しいが為に覚えてこの工夫がややこしい、という方が居たら、心の中でこの章のプロセス構造体、という言葉を全部 binder_proc に置き換えて読んでください。====[/column]

5.3 8.5.3 オブジェクトの送信と flat_binder_object その 1 - ユーザープロセス側

メソッド呼び出しの引数としてオブジェクトを渡す場合の話に入ります。

オブジェクトというのは生のポインタです。そのプロセス内のメモリ空間でだけ意味があります。この生のポインタを binder ドライバに渡すと相手側ではハンドルとして渡ってきます。

まずは生のポインタが存在しているプロセス側で、送信する時のコードを見てみましょう。
binder ドライバに生のポインタを渡す場合は、flat_binder_object 構造体に入れて渡します。
サービスの取得の所、つまり受け取る側でも、結果は flat_binder_object 構造体として返ってくる、という事を説明しました。(8.4.6) 送る側もこの flat_binder_object という構造体を使います。
flat_binder_object には型を表すフィールドがあり、そこには大きく三つの値が入ります。

1. BINDER_TYPE_BINDER
2. BINDER_TYPE_HANDLE
3. BINDER_TYPE_FD

1 がポインタ、2 がハンドル、3 がファイルディスクリプタです。

たとえば handle という変数に入ったハンドルの値を保持する flat_binder_object は以下のように作れます。

リスト 5.1: flat_binder_object にハンドルを入れる場合

```
flat_binder_object obj;
obj.type = BINDER_TYPE_HANDLE;
obj.handle = handle;
```

ptr というポインタ変数に入ったポインタを保持する flat_binder_object なら以下のようになります。

リスト 5.2: flat_binder_object にポインタを入れる場合

```
flat_binder_object obj;
obj.type = BINDER_TYPE_BINDER;
obj.binder = ptr;
```

8.5.3 ①

flat_binder_object	
type	BINDER_TYPE_BINDER
handle → binder	ポインタ

flat_binder_object	
type	BINDER_TYPE_HANDLE
handle → binder	サービスのハンドル

flat_binder_object	
type	BINDER_TYPE_FD
handle → binder	ファイルディスクアダプタ

flat_binder_object で送れるのは、①ポインタ、②ハンドル③ファイルディスクアダプタの 3つ

図 5.5 flat_binder_object で送れる物三つ

基本的にはこの flat_binder_object を引数を表すバッファに入れて少し補助的な設定をした上で ioctl を呼べば良い、という事になります。引数というのは binder_transaction_data の ptr.buffer に入る、という話を 8.4.3 で行いましたが、いろいろな場所に説明が飛ぶと読む方も大変だと思うので、一部繰り返しになりますがもう一度ここで全体像を見てみましょう。

サービスの servicemanager への登録を例として、オブジェクトの送信を見ていきまます。MyService というクラスをサービスとして登録する場合を見ていきます。サービス名は "com.example.MyService" とします。

servicemanager はハンドルが 0 という固定値でした。この 0 というハンドルに BC_TRANSACTION で SVC_MGR_ADD_SERVICE というメソッドを呼び出すことで、サービスの登録を行えます。(8.4.4)

SVC_MGR_ADD_SERVICE は、引数として以下の三つを受け取ります

- String16 型の "android.os.IServiceManager" という固定文字列
- String16 型のサービス名、この場合は "com.example.MyService"
- サービスのポインタ

1 番目と 2 番目は適当なバイト配列に memcpy してやれば良い訳です。3 は flat_binder_object にオブジェクトのポインタを詰めて、それを memcpy してやれば良いのですが、flat_binder_object の時にはもう一つ、オフセットの指示という作業が追加で必要となります。というのは、それ以外の値は全てドライバとしてはバイト列をコピーするだけで良いので中を知っている必要は無いのですが、flat_binder_object だけはドライバが変換するので、中身を知っている必要があるのです。

まずはバイト配列を生成し、二つの固定文字列をバイト配列にコピーします。詳細は省略します。

リスト 5.3: バイト配列に二つの文字列をコピー

```
byte writedata[1024];
// "android.os.IServiceManager" の String16 を writedata に memcpy する。省略
...
```

```
// "com.example.MyService" の String16 を writedata に memcpy する。省略
```

次に flat_binder_object を memcpy します。まずは flat_binder_object の作成から。

リスト 5.4: バッファにコピーする flat_binder_object を生成

```
// MyService のインスタンスを生成
MyService *service = new MyService;

flat_binder_object obj;
// ポインタの時はタイプは BINDER_TYPE_BINDER
obj.type = BINDER_TYPE_BINDER;
// obj.binder にポインタを入れる
obj.binder = service;
```

このように flat_binder_object という物を作って、writedata に memcpy します。なお、文字列二つを memcpy した時に used バイトまで既に使ったとします。^{*4}

リスト 5.5: 生成した flat_binder_object をバイト配列にコピー

```
// flat_binder_object を writedata の used バイトより先に書き込む。
memcpy(&writedata[used], &obj, sizeof(flat_binder_object));

// 後で使うので used を進めておく。
used += sizeof(flat_binder_object);
```

こうして出来た writedata を 8.4.4 と同様に binder_transaction_data に入れて、それを 8.4.5 と同様に binder_write_readに入れれば良い訳ですが、flat_binder_object を渡す時は先ほども述べた通り、さらにオフセットの指定という事もやらなくてはいけません。

^{*4} used の実際の値は $4 + 2 * \text{sizeof}(\text{"android.os.IServiceManager"}) + 4 + 2 * \text{sizeof}(\text{"com.exmaple.MyService"})$ ですが、詳細を気にする必要は無いでしょう。

図5.6

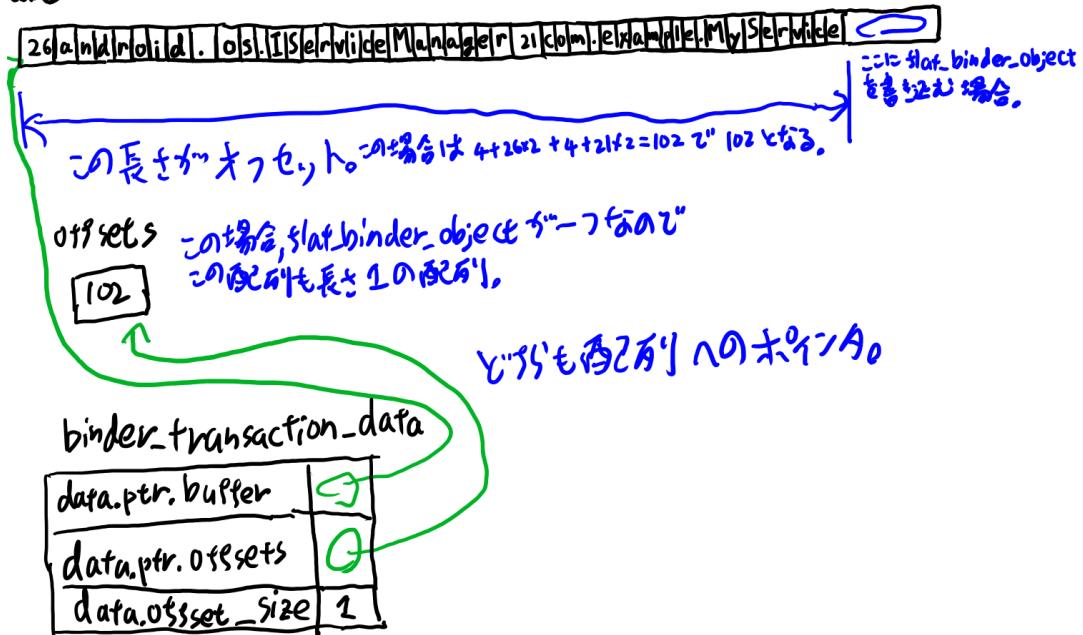


図 5.6 flat_binder_object とオフセットの書き方

一気に三つの事が説明に出てきたので、一つずつ見ていきましょう。

まずは `binder_transaction_data` に上で作った `writedata` やターゲットとなるハンドル等を設定します。

リスト 5.6: `binder_transaction_data` に送り先のハンドルを指定し、ここまで作ったバッファをセット

```
// binder_transaction_data の設定
struct binder_transaction_data tr;

// servicemanager のハンドルは 0 にハードコード
tr.target.handle = 0;

// 呼び出すメソッドの ID。今回はサービスの登録なので ADD_SERVICE。
tr.code = SVC_MGR_ADD_SERVICE;

// 引数には上で作った writedata を設定
tr.data_size = used;
tr.data.ptr.buffer = writedata;
```

送り先が 0、メソッド ID が `SVC_MGR_ADD_SERVICE`、引数が `writedata`、という訳です。さらにこの引数データの中で、どこに `flat_binder_object` があるか、という情報も追加してやります。

リスト 5.7: `binder_transaction_data` の引数の中のうち、どこが `flat_binder_object` かを表す offset を指定

```

// offsets として、今回は送り出すデータの中には flat_binder_object は一つなのでサイズは 1
size_t offsets[1];
// flat_binder_object をどこに書いたか。最後の引数なので末尾から sizeof(flat_binder_object) だけ戻った所にあるはず。
offsets[0] = used - sizeof(flat_binder_object);

// offsets は、今回は長さ 1 の配列
tr.data.offset_size = 1;

// 上で設定した offsets を代入
tr.data.ptr.offsets = offsets;

```

binder ドライバにこの ptr.buffer の中のこことここに変換の必要のある flat_binder_object が入っているよ、と伝える為に、offsets というメンバと offset_size というメンバを設定します。offsets は size_t の配列で、各要素が flat_binder_object が ptr.buffer の先頭からのオフセットに対応します。offset_size は offsets 配列の長さです。

このようにして設定した binder_transaction_data を buffer_write_read に詰めて BC_TRANSACTION として ioctl を呼び出します。

以後は前に説明したサービス取得のコードと全く同じコードになりますが、再掲しておきます。

リスト 5.8: 再掲: サービス取得のコード

```

// 送信用のデータのバッファ。コマンド ID と先ほど作った binder_transaction_data を詰める。
byte writebuf[1024];
*((int*)writebuf) = BC_TRANSACTION
memcpy(&writebuf[4], &tr, sizeof(struct binder_transaction_data));

// 結果受け取りの為の受信用バッファ
byte readbuf[1024];

// binder_write_read に送信用と受信用のバッファを設定
struct binder_write_read bwr;

// 送信用データ。長さはコマンド ID のサイズ +binder_transaction_data のサイズ
bwr.write_size = writebuf;
bwr.write_buffer = sizeof(int)+sizeof(struct binder_transaction_data);

// 受信用バッファ
bwr.read_size = 1024;
bwr.read_buffer = readbuf;

// ioctl で servicemanager の SVC_MGR_ADD_SERVICE を呼び出す
res = ioctl(fd, BINDER_WRITE_READ, &bwr);

```

このようにすると、MyService という生のポインタを引数にして、servicemanager にサービスを登録する、というメソッドを呼び出した事になります。それではこの生のポインタが binder ドライバではどう扱われて servicemanager 側に渡るのか？ という部分を見ていきましょう。

5.4 8.5.4 オブジェクトの送信と flat_binder_object その 2 - binder ドライバと受信側

ioctl 呼び出しの時に flat_binder_object を渡すと、ドライバは内部で flat_binder_object の中身の種類に合わせて適切に変換し、送り先に送ります。

BINDER_TYPE_BINDER のポインタは呼び出し元のメモリ空間でしか有効で無いので、このポインタを渡されても別のプロセスは困ってしまいます。

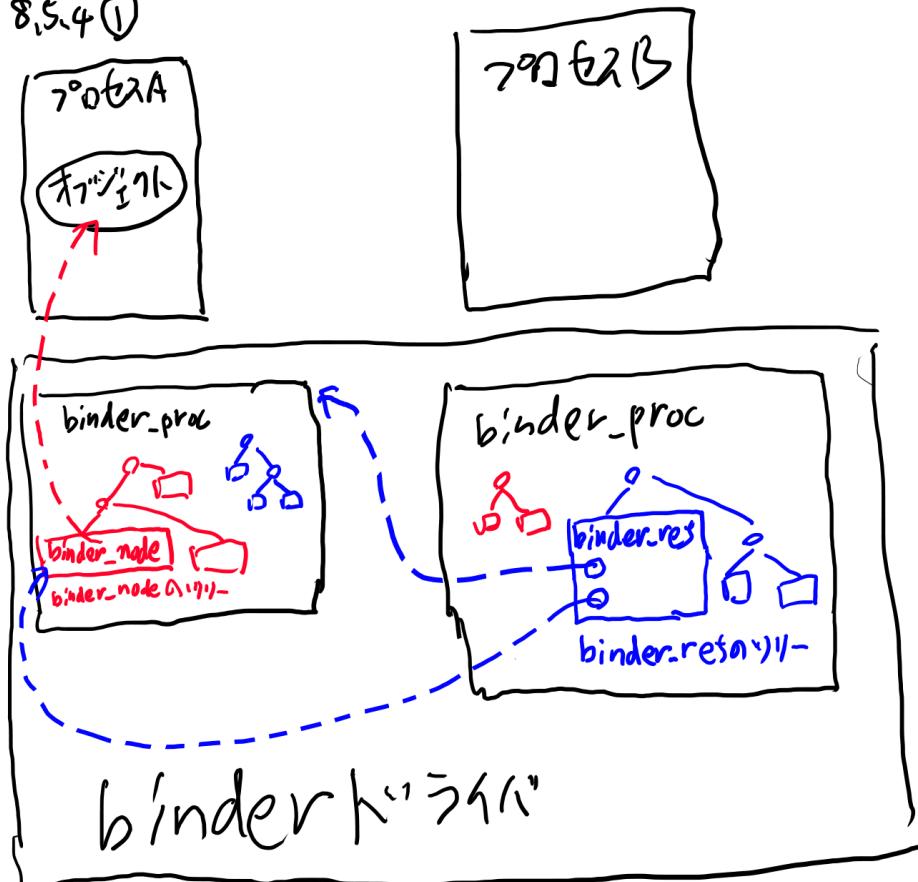
8.5.2 で解説したように、binder ドライバを open した時のファイルディスクリプタには、そのプロセスの構造体が格納されています。

そしてこのプロセス構造体には、そのプロセスが保持する BINDER_TYPE_BINDER のポインタを格納するツリーがあります。このツリーのノードを、binder_node と呼んでいます。ツリーになっているのは高速に検索する為です。^{*5}

あるプロセスが保持しているサービスのポインタの一覧がツリーで管理されていると言いました。さらに、そのプロセスが参照している外部のサービスの一覧もツリーで管理されています。これは binder_ref というノードの表すツリーです。このツリーのノードで、参照しているサービスが表されます。binder_ref のフィールドには、参照しているサービスが所属しているプロセス構造体と、そのプロセス構造体にある BINDER_TYPE_BINDER のポインタを参照する binder_node が格納されます。

^{*5} これも Linux カーネルの提供する赤黒木となっています。

8.5.4 ①



binder_nodeは自身のプロセスのポインタに対する。
binder_resは参照している別のプロセスのポインタに対応。

図 5.7 binder_ref が参照している物

別の角度から同じ事を説明してみましょう。別のプロセスに BINDER_TYPE_BINDER のポインタを渡す時を考えます。ポインタが所属しているプロセスを A、送り先のプロセスを B とします。

8.5.4(2)

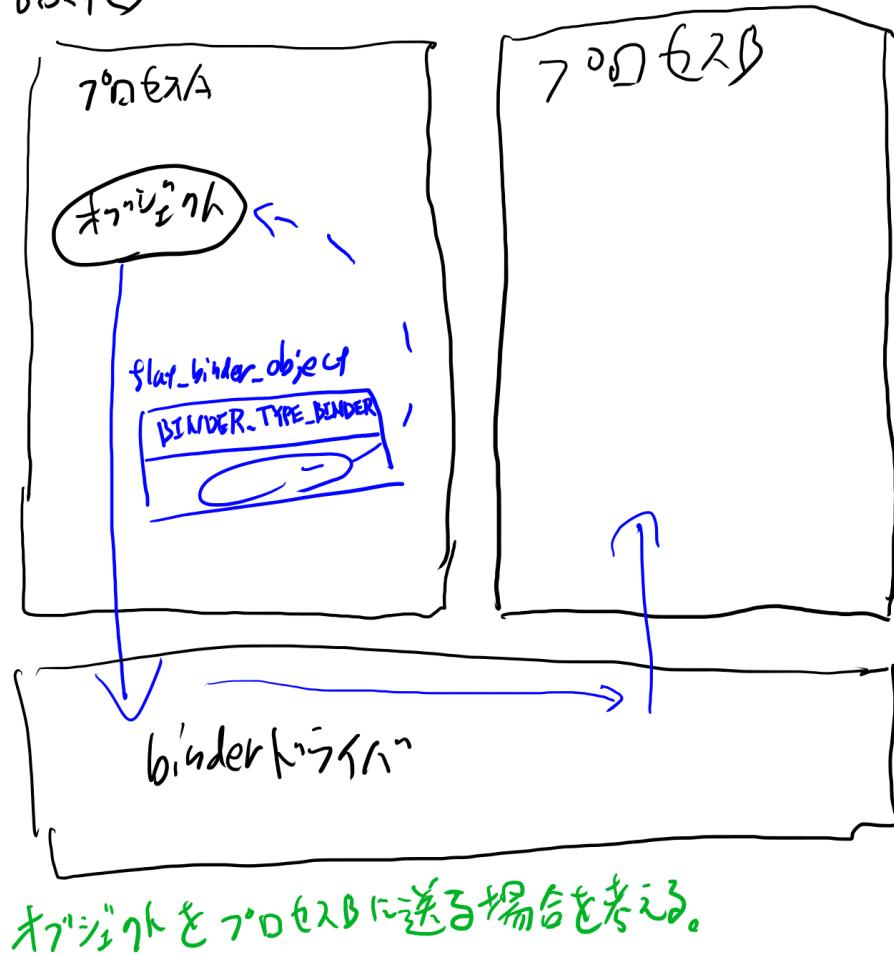


図 5.8 問題設定。オブジェクトを B に送信したい。

その時には、以下の事が起こります

1. A のプロセス構造体の binder_node ツリーに、このポインタを表すノードを追加してポインタを格納
2. B のプロセス構造体にこのポインタへの参照を表す binder_ref のノードを追加し、1 のノードと A のプロセス構造体を追加
3. B の binder_ref のツリーのノードに一意の int の ID を振って、その ID をハンドルとしてプロセス B に渡す

この 3 のハンドルこそが、サービスの取得の時に得られた BINDER_TYPE_HANDLE の flat_binder_object に格納されていた、そして binder_transaction_data の target.handle に格納する、そして servicemanager は 0 とハードコードされているハンドルです。

binder_ref とハンドルの関係はちょっとわかりにくいですが、binder_ref のノードに順番に 1 から自然数の ID を振っていて、その ID がハンドルだ、というとだいたい正しい説明となります。「だいたい正しい」というのは、途中でノードを削除したりすると抜け番が出来て、新しくノードを作る時にはそれを再利用する処理がある為です。とにかく、binder_ref のノードを一意に識別する int 値がハンドルです。

ハンドルは binder_ref のノードを引くキーと言えます。内部実装を忘れれば、ハッシュのような物に格納されていて int のキーで lookup 出来ると思っておいて問題ありません。そしてそのキーがハンドルという訳です。^{*6}

B のプロセス内でのこのハンドルは、このプロセス B でのみ有効な int 値です。この値があれば、B のプロセス構造体から素早く binder_ref を検索できます。そして binder_ref の中を見ると、このサービスを所持しているプロセス構造体と、このサービスのポインタを所持している binder_node が得られます。

以上が A から B を呼び出した場合です。

これを逆の順序で見ていくと、B からこのハンドルに対して BC_TRANSACTION した時に何が起こるのかが分かります。B から A を呼び出す事を考えましょう。

^{*6} 厳密には内部ではポインタによる赤黒木とハンドル値による赤黒木の二本の木を持つ事でこの構造を実現しています。

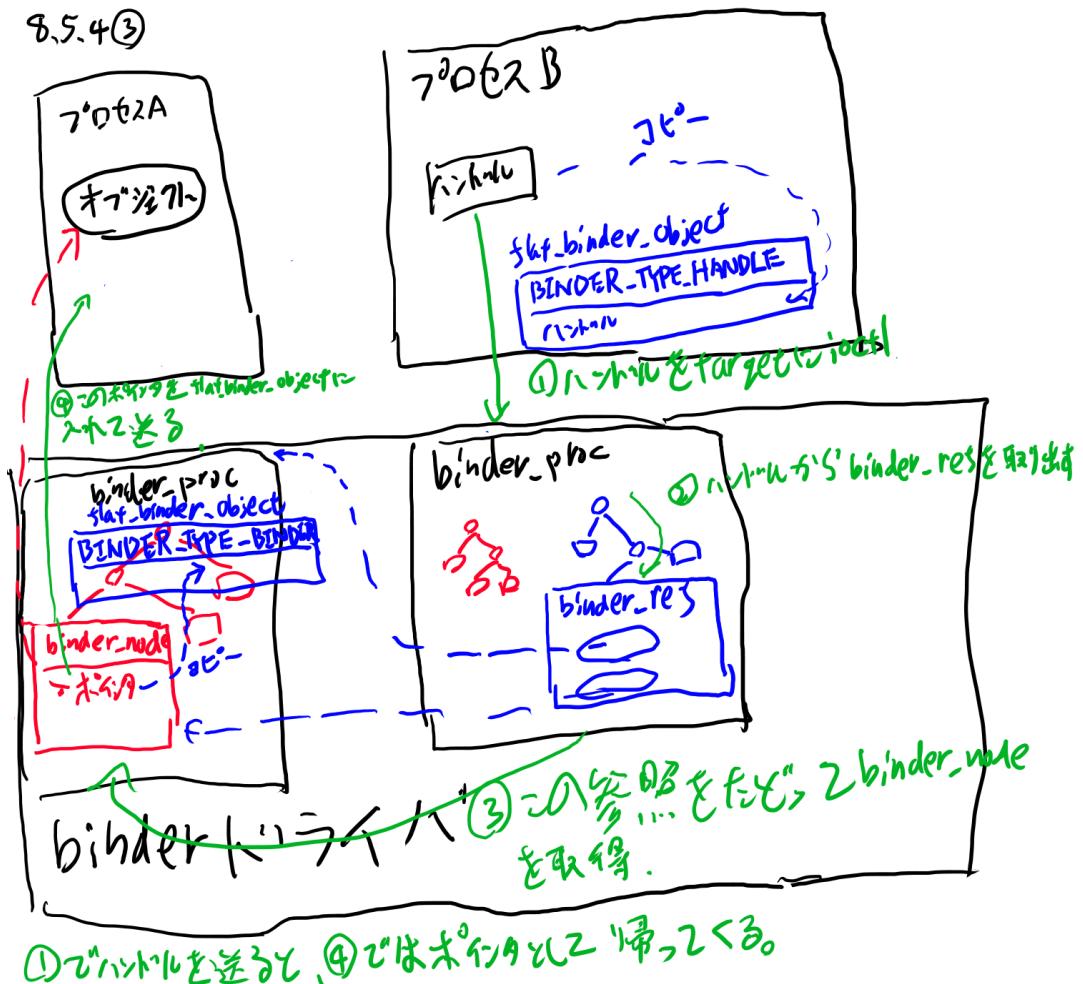


図 5.9 B からハンドルをターゲットに A のオブジェクトを呼び出す

1. プロセス B がハンドルを target に ioctl を呼び出す
2. ドライバがプロセス B のプロセス構造体の binder_ref ツリーからハンドルの表すノードを高速に検索して取り出す
3. binder_ref のノードの中にある送り先のプロセス構造体と binder_node を取り出す
4. 送り先のプロセスのメモリ空間で有効なサービスへのポインタを binder_node から取り出す
5. binder_transaction_data の target.ptr にこのポインタを詰める
6. プロセス A の ioctl から戻る

という手順になります。1で渡したハンドルが、binder_ref を介して B のポインタになって B に渡る訳です。

target 以外の所で、引数などに BINDER_TYPE_HANDLE の flat_binder_object がある場合にも、ほぼ同じ作業が行われます。唯一の違いは、flat_binder_object は送り先ではタイプが

BINDER_TYPE_BINDER に変更される事です。

ターゲットの場合は送り先のプロセスに必ずオブジェクトが存在するので型を表すフィールドは必要ないのですが、引数の場合はそのプロセスには存在しない場合も存在するのでハンドルかポインタかを表すフィールドが必要になる訳です。

8.5.4④

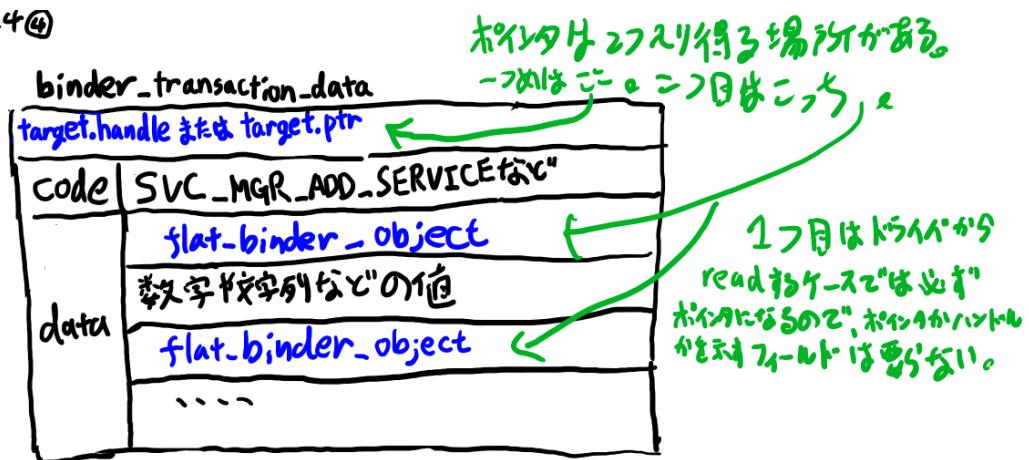


図 5.10 ポインタが入る二つの場所

ハンドルはプロセス B の中でしか有効ではありません。例えばプロセス C がまたプロセス A の同じサービスに対して呼び出しを行う時には、プロセス C にはプロセス B とは別の `binder_ref` リストがあるので、その辺りのインデックスも別物となります。

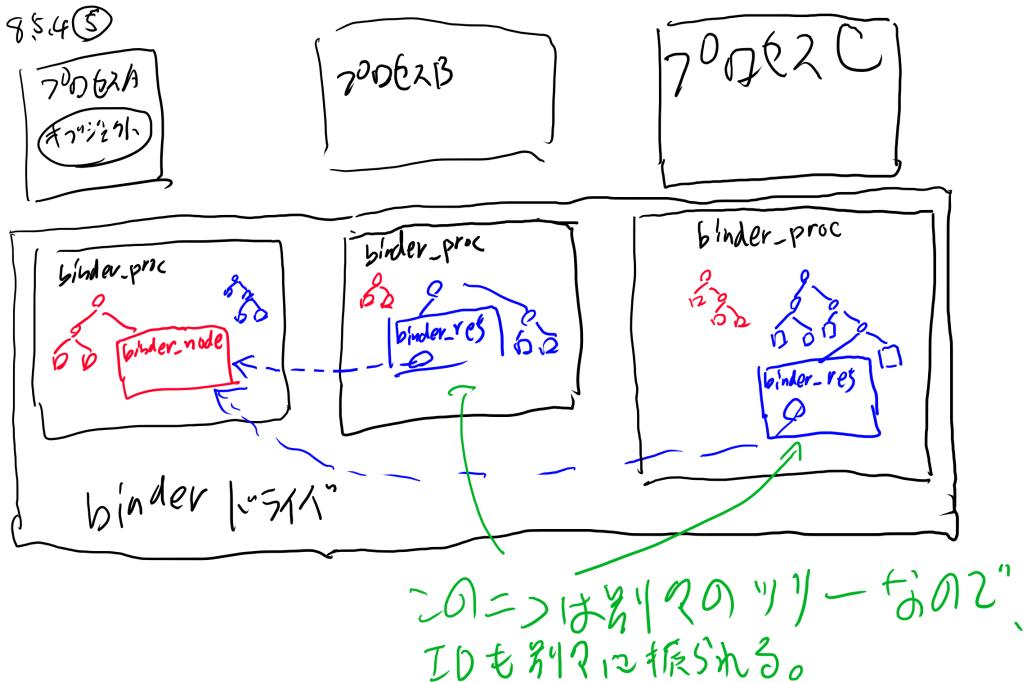


図 5.11 B と C では別のハンドルの値となる

こうして、ポインタを送るとそれを管理する binder_node のツリーと、それを参照する binder_ref のツリーが双方のプロセスに作られて、binder_ref のノードを表すハンドルが ioctl からは返る事になります。そしてこのハンドルをターゲットに ioctl を呼び出すと、受け取る側では binder_node からポインタを引いて、ポインタに戻って ioctl から返ってくる事になります。

このようにして、binder ドライバはまるでオブジェクトを送っているかのように見せかけています。

以上でオブジェクトの送信のメカニズムの説明が終わったので、任意のメソッドを呼び出す事が出来るようになります。これで binder ドライバの主要なところは一通り解説した事になりますが、おまけとしてファイルディスクリプタの送信の話もしましょう。

5.5 8.5.5 ファイルディスクリプタの送信と flat_binder_object - BINDER_TYPE_FD の場合

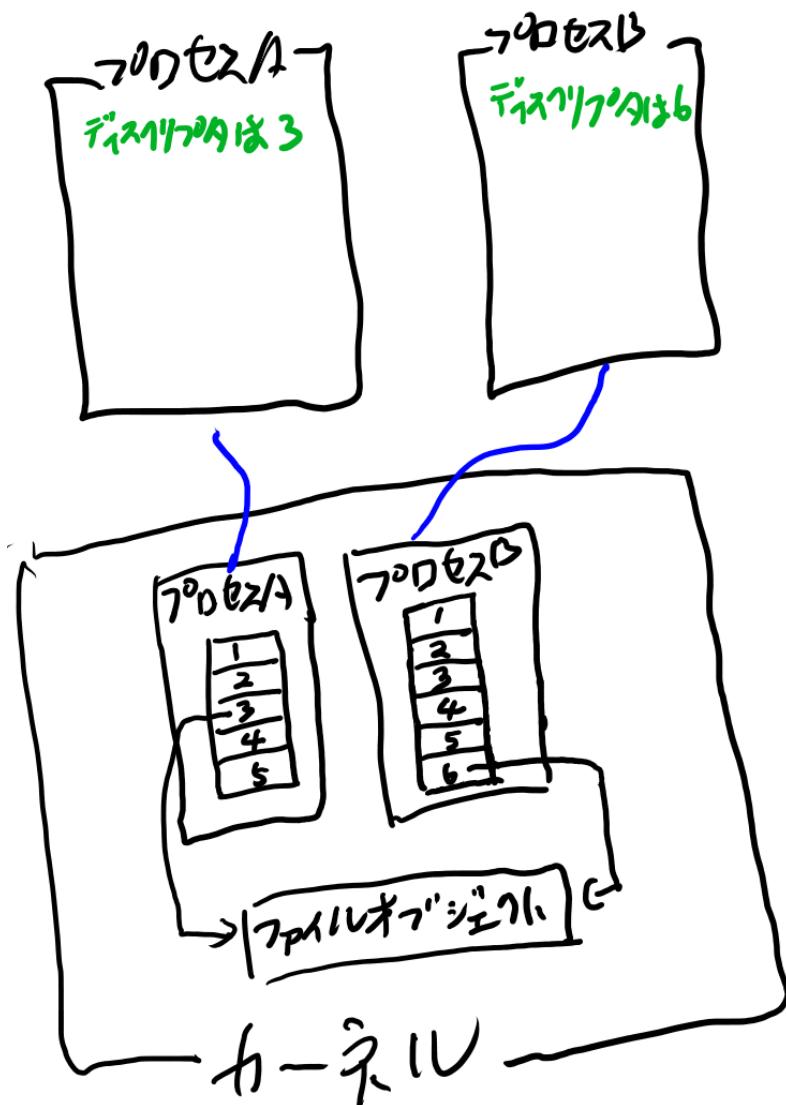
binder の特徴でファイルディスクリプタを別のプロセスに送れる、という物があります。ここまで解説してしまうと大した話でも無いのですが、有名な話でもあるのでここでメカニズムを確認しておきます。

ファイルディスクリプタの話をする為には Linux のファイルの話を少しする必要があります。Linux の各プロセスには、自身のオープンしているファイルの一覧を保持するテーブルがあります。このテーブルをファイルディスクリプタテーブルと言います。このファイルディスクリプタテーブルの各エントリが、カーネルの管理するオープンしているファイルオブジェクトを指しています。ファイルディスクリプタというのは、通常はこのプロセスごとのファイルディスクリプタテーブル

第 55章 `Plain Binder` による自動的な送信の内部 `binder::object` の選択と `Plain Binder` の場合

の先頭からのインデックスです。

8.13①



カーネルの中の タスク構造体の中には
ファイルディスクリプタのテーブルがあり、このテーブルのエントリ
がファイルオブジェクトをさしている。テーブルのインデックスが
ファイルディスクリプタ。

図 5.12 ファイルディスクリプタとファイルディスクリプタテーブル、再掲

第 55 章 flat_binder_object を用いたため送信の内側 binder_object の送信と flat_binder_object の場合

さて、プロセス A でオープンしているファイル、fd1 があったとします。これをプロセス B に送る場合を考えます。

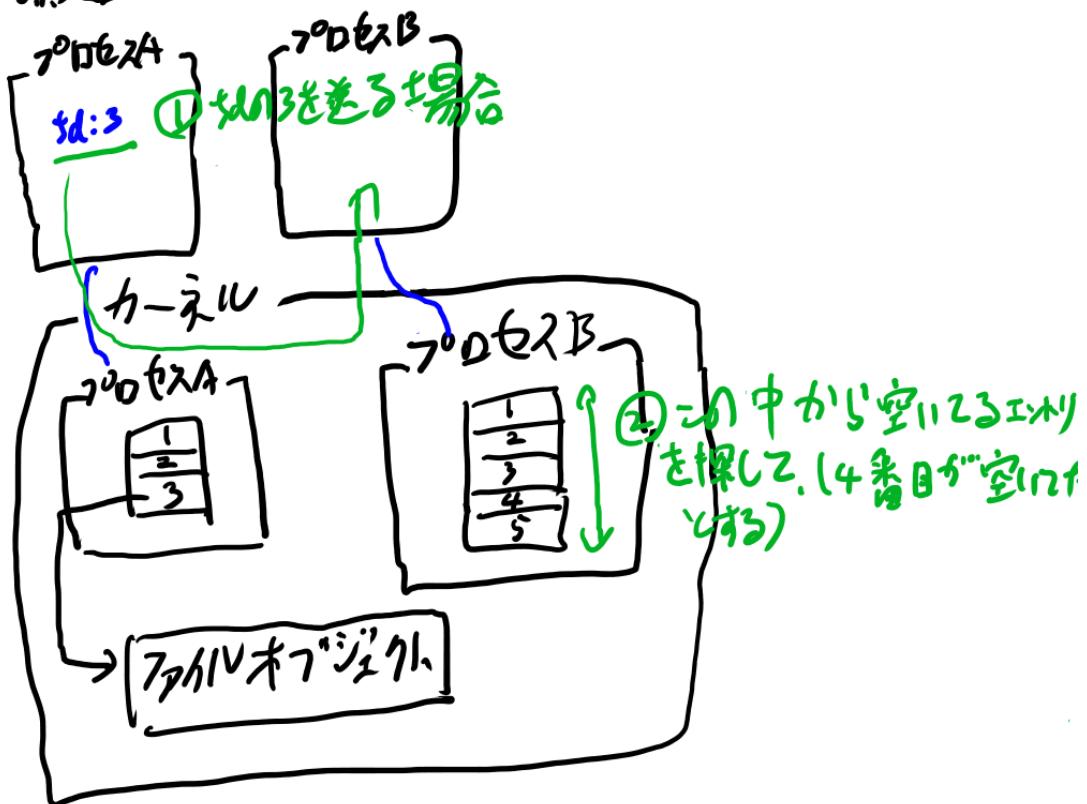
ファイルディスクリプタを送る場合も、BINDER_TYPE_BINDER でサービスを送る場合と同様に、flat_binder_object を使います。

リスト 5.9: ファイルディスクリプタも flat_binder_object で送る

```
flat_binder_object obj;  
  
// ファイルディスクリプタの時はタイプは BINDER_TYPE_FD  
obj.type = BINDER_TYPE_FD;  
  
// obj.handle にファイルディスクリプタを入れる  
obj.handle = fd1;
```

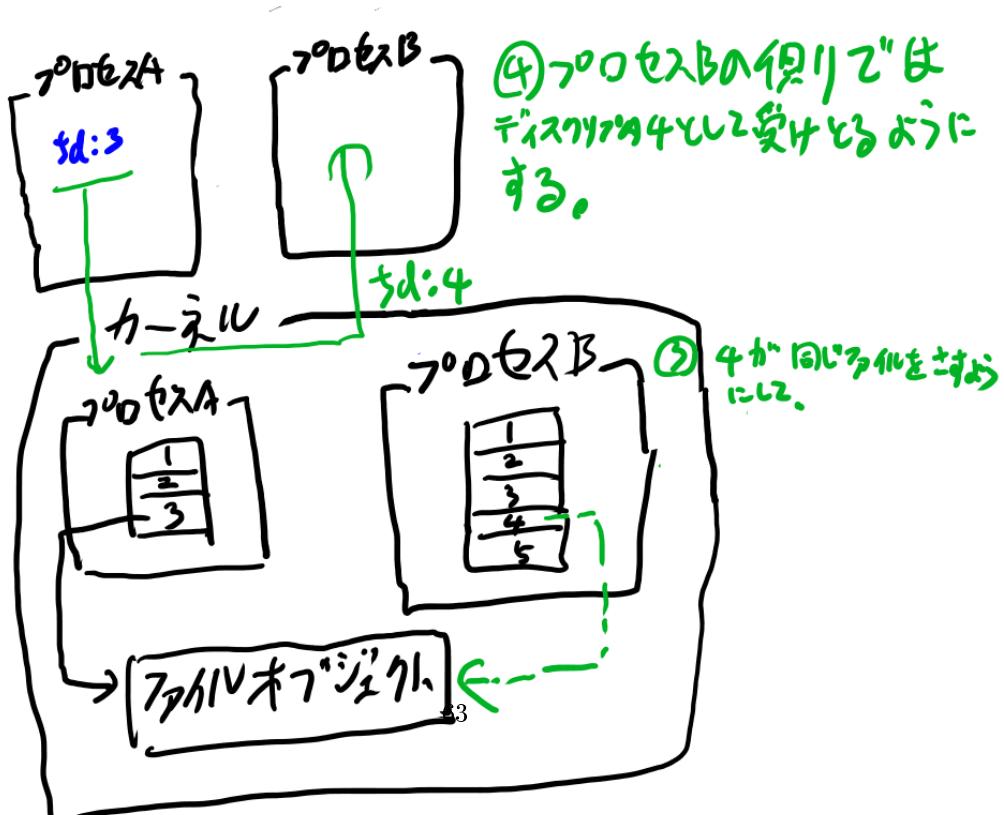
さて、こんな flat_binder_object を含んだデータを ioctl に渡すと、binder ドライバは送り先のプロセスのファイルディスクリプタテーブルから空きを探して、このファイルディスクリプタテーブルが指しているのと同じファイルのエントリを指すようにして、この送り先のディスクリプタテーブルに handle の値を書き換えます。

8.1.3 ②



④ 700セ入Bの奥で"は
ディスク DMA で受け取るように
する。

⑤ 4が同じアドレスをもつ
ように。



第 55 章 flat_binder_object を用いたため送信の内部 binder_object の処理と flat_file_fd の場合

そこのコードを抜き出すと以下のようになっています。

リスト 5.10: binder ドライバ内のファイルディスクリプタの処理

```
int target_fd;
struct file *file;

// pobj には flat_binder_object のポインタが入っているとする。
// カーネル API。オープンしているファイルオブジェクトを取ってリファレンスカウントを +1
file = fget(pobj->handle);

// target_proc で表しているプロセスのファイルディスクリプターテーブルから空いているディスクリプタを
// 取ってくる
target_fd = task_get_unused_fd_flags(target_proc, 0_CLOEXEC);

// target_proc のプロセスのファイルディスクリプターテーブルにカーネルのファイルオブジェクトを代入
task_fd_install(target_proc, target_fd, file);

// 送り先のハンドルに変換
pobj->handle = target_fd
```

task_get_unused_fd_flags() 関数と task_fd_install() 関数は binder ドライバで実装されている関数ですが、名前の通りの事をしているだけなのでここではこれ以上は踏み込みません。

ポイントとしては、flat_binder_object に BINDER_TYPE_FD として開いているファイルのファイルディスクリプタを入れて binder 経由でサービスに送ると、サービス側ではそのサービスの動くプロセス上のファイルディスクリプターテーブル上の同じファイルを指すエントリに自動的に変換してくれる、という事です。

この辺のコードはカーネルの内部構造をいろいろ触る割にはシンプルで読みやすいので、カーネルの勉強をしたい人などは読んでみると面白いと思います。

以上で binder ドライバの説明は全て終わりました。

サービスは全て、binder ドライバさえあれば実装出来ます。ですが皆が全部をこの ioctl で実装していくのは大変だし無駄なので、この binder ドライバを使うライブラリがこの上に提供されています。

以下ではこの binder ドライバの上に作られているライブラリについて話をしていきます。

第 6 章

{threadpool-layer} 8.6 スレッドプールのレイヤ - BBinder と IPCThreadState

前節までで解説した binder ドライバさえあれば、サービスの実現に必要な事は全て出来ます。ですが、これはあまりにも低レベルな為、これだけで分散オブジェクトのシステムを実装すると、各サービスの開発者皆が同じようなコードを書く事になってしまいます。

そこで ioctl を直接呼び出して決まった処理を行う部分をスレッドプールで行うレイヤがあります。このレイヤはデータをシリализ-デシリализする Parcel、最終的に処理を受け取る BBinder、BBinder に対応するハンドルを保持してメッセージを送信する BpBinder、としてスレッドプールである IPCThreadState と ProcessState で構成されています。

//image[6_1][スレッドプールのレイヤの位置づけ]

また、ネイティブで実装されたシステムサービスは、それをを実行するプロセスの main 関数が多くの場合このレイヤのクラスを走らせて、自身のサービスを servicemanager に登録するだけ、というコードになっています。そこでこの節でシステムサービスの典型的な main 関数についても見ていきます。

なお、システムサービスは SDK のレイヤでは提供出来ません。システムイメージに含める必要があります。このレイヤを直接使うのはカスタム ROM 開発者やメーカーの人など、かなり限られた人しか作る事は出来ないと思います。

独自のハードウェアをサービスとして提供したいメーカーの方などは本節の内容はとても貴重な資料となると自負していますが、そうで無い人でも、スレッドモデルの周辺を理解しているとカーネルのレベルで何が起こるのかを正確にイメージ出来るようになるので、私は Android を深く理解するなら必須の内容と思っています。

6.1 8.6.1 スレッドプールのレイヤの構成要素

スレッドプールのレイヤを構成する中心となるクラスは IPCThreadState です。このクラスは各スレッドごとに一つインスタンスが出来るようなスレッドローカルなシングルトンで、IPCThreadState::self() と呼ぶと、TLS にインスタンスが無ければ生成されます。(TLS については 3.2.2 のコ

ラム参照) zzz 参照の仕方相談

ioctl でデータを送受信するには、バイト配列に値をシリアル化したり、バイト配列から値をデシリアル化する必要があります。また flat_binder_object のオフセットを指定したりする必要があります。そういう ioctl に渡す引数処理を行うためのユーティリティが Parcel です。前節で memcpy で行っていたような処理を代わりに行ってくれます。

IPCThreadState は Parcel を二つメンバに持ります。mIn と mOut です。mOut に書いておいたものが、binder_write_read の write_buffer の方に、mIn は read_buffer の方に使われます。binder_write_read については 8.3.4 などで扱いました。

IPCThreadState の joinThreadPool() というメソッドが、ioctl を呼び出して、結果を処理する、というループを回します。この時に上記の mIn と mOut を設定した binder_write_read を引数とします。

IPCThreadState は、servicemanager に登録するオブジェクトは BBinder である、という前提を設ける事で、ioctl の結果の処理のうち、多くの共通部分を処理してくれます。これがサービス実装の基底クラスとなります。

そしてサービスの実装が BBinder であるなら、サービスのプロキシに対応するクラスもあります。それが BpBinder です。この BpBinder は IPCThreadState を用いて、ターゲットとなるハンドルに対して ioctl 呼び出しを行います。また、共通の基底クラスとして IBinder が存在します。IBinder の役割はこの時点では述べるのは難しいので、zzz で説明します。

最後に大した事はしないクラスですが良く登場するものに ProcessState という物があります。これは一プロセス一インスタンスなシングルトンオブジェクトで、IPCThreadState を立ち上げたりといった処理を行うユーティリティクラスです。

IPCThreadState, Parcel, IBinder と BBinder と BpBinder、そしておまけの ProcessState が、スレッドプールのレイヤを構成しているクラスです。

6.2 8.6.2 Parcel とシリアル化

Parcel はシリアル化やデシリアル化の機能を備えたバッファです。つまり内部にバイト配列を持っていて、このバイト配列に値をコピーしたり、このバイト配列から値を復元したりします。8.4.4 でちょっと登場しました。

大した事をするクラスでは無いのですが、今後良く登場するのでここで少し詳細に見ておきます。まず、String16("android.os.IServiceManager") をバッファにコピーする事を考えます。バイト配列にコピーするなら以下のようないいコードとなります。

リスト 6.1: バイト配列に文字列をコピーする場合

```
String16 s = String16("android.os.IServiceManager");

byte buf[1024];
memcpy(buf, s.string(), s.size());
```

これをパーセルに書く場合は以下のようになります。

第6章 {threadpool-layer} 8.6 スレッドプールのレイヤ - BBinder & IPCHandlerState アライズ

リスト 6.2: Parcel に文字列をコピーする場合

```
String16 s = String16("android.os.IServiceManager");

Parcel buf;
buf.writeString16(s);
```

writeString16 の他に writeInt32 や、バイト配列を書きこむ write などもあります。書きこんだ配列を取得するのは data() メソッドです。

リスト 6.3: Parcel からバッファのポインタを取得

```
byte *ptr = buf.data();
int size = buf.dataSize();
```

また、Parcel には flat_binder_object のサポートがあります。writeStrongBinder というメソッドに BBinder のポインタを渡すと、内部で flat_binder_object にラップして書き込み、書き込んだ場所を覚えておいてくれて、ipcObjects() というメソッドでオフセットの配列を取得出来ます。

8.5.3 で見たサービスの登録の時に用意するバッファと同じバッファは、以下のように用意出来ます。

リスト 6.4: サービス登録の三つの引数の書き込み

```
Parcel buf;

// 第一、第二引数の文字列を書く
buf.writeString16(String16("android.os.IServiceManager"));
buf.writeString16(String16("com.example.MyService"));

// MyService のインスタンスを生成
MyService *service = new MyService;

// 第三引数の flat_binder_object を生成して書きこみ
buf.writeStrongBinder(service);
```

Parcel に write するメソッドを呼び出していくだけで、簡単です。こうして出来たバッファを binder_transaction_data にセットするのでした。

8.4.1 ①

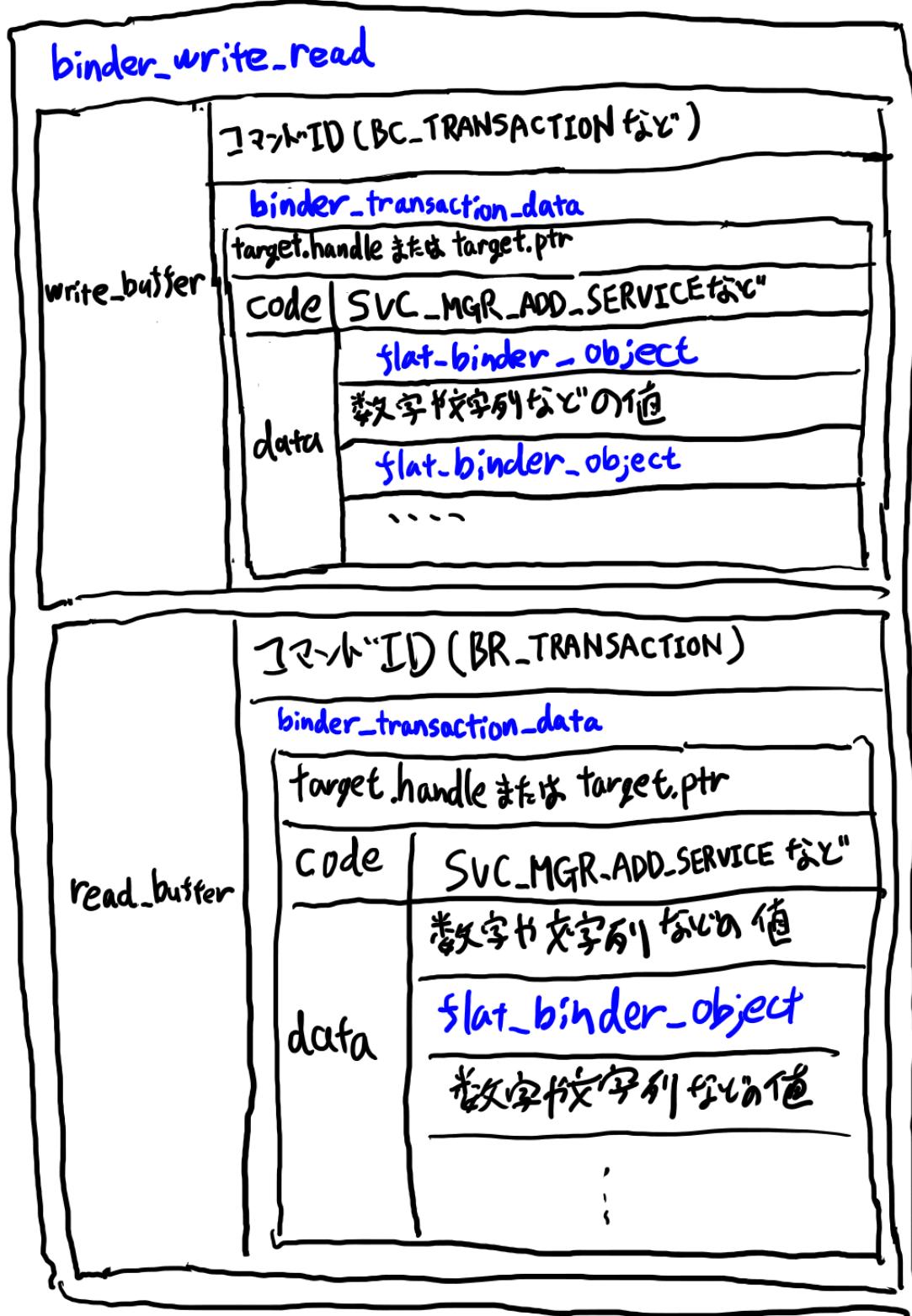


図 6.1 binder_write_read、binder_transaction_data、flat_binder_object の包含関係、再掲

そのコードは以下のようになります。

リスト 6.5: 引数の書きこまれたバッファを binder_transaction_data に設定

```
struct binder_transaction_data tr;

// servicemanager のハンドルは 0 にハードコード
tr.target.handle = 0;

// 呼び出すメソッドの ID。今回はサービスの登録なので ADD_SERVICE。
tr.code = SVC_MGR_ADD_SERVICE;

// 引数には上で作った writedata を設定
tr.data_size = buf.dataSize();
tr.data.ptr.buffer = buf.data();

// /* 1 */ offsets 配列と配列の長さの設定。
tr.data.ptr.offsets = buf.ipcObjects();
tr.data.offsets_size = buf.ipcObjectsCount();
```

8.5.3 のコードに比べると、バイト配列の扱いをほとんど気にする必要が無くなっているのが分かると思います。また、/* 1 */にあるように offsets の配列を作ってくれる所が Binder 専用のシリアルライザっぽいですね。

最後になりましたが、Parcel は binder_transaction_data のバッファを作る時にも、binder_write_read のバッファを作る時にも使えます。

6.3 8.6.3 IPCThreadState 概要

IPCThreadState は各スレッドごとに一つインスタンスが出来るような単位のオブジェクトです。そしてそのスレッドで、「ioctl を呼んで受信した結果を処理する」という処理をループするオブジェクトです。GUI のメッセージループに似ていますね。ioctl 周辺の処理を全て受け持つクラス、と言えます。

IPCThreadState は Parcel 型の mIn と mOut というオブジェクトを保持します。そしてスレッドプールのループで、mOut に書かれている事を write_buffer にセットし、そして read_buffer に mIn をセットして ioctl を呼び出します。つまり mOut を送信し、結果を mIn で受け取る訳です。

このようなループが走る事で、コードの他の部分では mOut に送りたい物を詰めておけばやがて勝手に送信される、という風に出来ます。

IPCThreadState は名前の通り、スレッドローカルなオブジェクトです。さらにスレッドローカルなシングルトンオブジェクトもあります。IPCThreadState::self() と呼び出すと、同一スレッド内ならどこでも同じインスタンスが返ります。<fn>初回呼び出しでインスタンスが作られて、7 章で紹介した TLS(スレッドローカルストレージ) にインスタンスが入ります。</fn>

だから次の ioctl ループで何か送り出してほしい、と思う事があったら、IPCThreadState とは全然関係無いクラスの中でも、IPCThreadState()::self() と現在のスレッドの IPCThreadState インスタンスを取り出して、そのインスタンスにリクエストなどを依頼する事が出来ます。

具体例は 8.6.7 の BpBinder で登場します。

IPCThreadState は、メッセージの受信対象が BBinder のサブクラスである、という前提で処理を行います。ここで少しメッセージの受信対象について補足しておきます。ioctl のメッセージ送受信には、必ず送る相手、受け取る相手がいます。この「相手」はハンドルで指定します。ハンドルが存在するためには、どこかしらで flat_binder_object か target にポインタを入れて binder ドライバに渡してある必要があります。(8.5.4 参照) 通常のケースではハンドルは servicemanager から取得する物ですが、幾つかのケースでは引数に渡されたオブジェクトがハンドルとして相手側に渡る事もあります。<fn>7.2.3 の ApplicationThread などがこのケースです。</fn>

どちらにせよ、何かしらの手段で binder ドライバに渡したポインタだけがメッセージを受け取る事になります。このプロセスにメッセージがやってきた、という事は、このプロセスのそれ以前の場所でポインタを binder ドライバに渡していて、そのポインタに対してメッセージがやってきている訳です。

この、現在のプロセスで以前に binder ドライバに渡したポインタ、それが BBinder のサブクラスでなくてはいけない、と IPCThreadState は要求している訳です。実際 Android で binder ドライバに渡される全ポインタが BBinder のサブクラスとなっています。

話を戻します。`ioctl`呼び出しのループを実際に行うメソッドが`joinThreadPool()`です。次項でこの`IPCThreadState`の本体とも言える、`joinThreadPool()`メソッドを見ていきます。

6.4 8.6.4 IPCThreadState の ioctl() 呼び出しループ - joinThreadPool() メソッドと BBinder

IPCThreadState の joinThreadPool() は、ioctl を呼び出して結果を処理する、というループを行なうメソッドです。メソッドの名前は、このメソッドを呼ぶと以後このスレッドはスレッドプールの一員としてループを処理し続けます、というような意味合いでしょう。ループのメソッドでは普通ですが、このメソッドも一度呼び出すと終了メッセージまで戻ってきません。

ioctlで受信したメッセージの先頭はコマンドIDになっていました。(8.4.2参照)

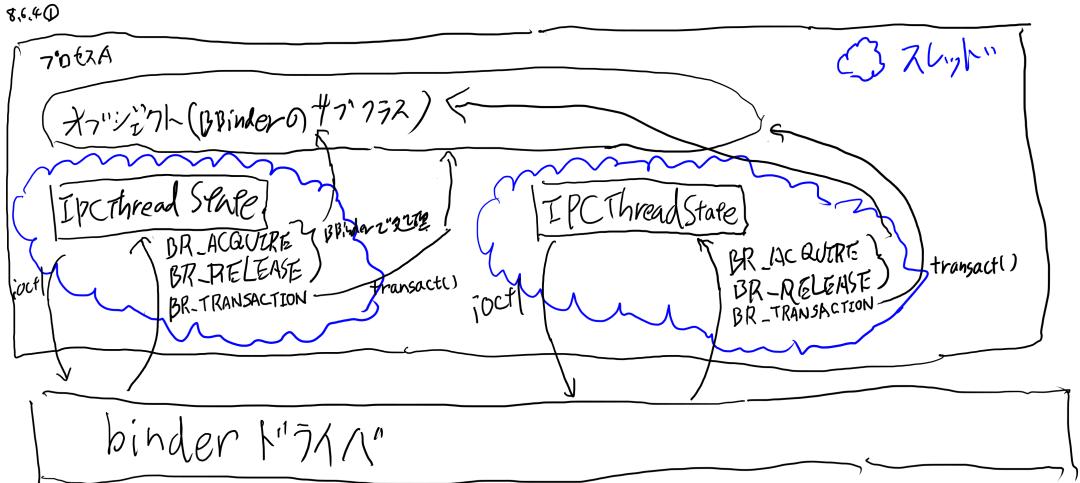
リスト 6.6: コマンド ID の取り出し

```
// ioctl呼び出し後。結果はmInに入っている  
  
int32_t cmd;  
cmd = mIn.readInt32();
```

コマンド ID で、重要なのは以下の物が挙げられます。

1. BR_TRANSACTION
 2. BR_ACQUIRE
 3. BR_RELEASE

受信側なので BR で始まっています。joinThreadPool() メソッドは、ioctl を呼び出してはこれらのコマンド ID に応じた処理を行うメソッドです。

図 6.2 `joinThreadPool()` メソッドの処理概要

先頭の `BR_TRANSACTION` 以外はリファレンスカウントなどの寿命管理関連です。これらの処理は、サービスの基底クラスとなる BBinder だけで全ての処理が実行出来るので、`joinThreadPool()` メソッド内で処理が完結し、BBinder を継承するサービスの実装者は別段何もコードを書かなくても必要な処理が行われます。

さて、一番重要なのは残った `BR_TRANSACTION` コマンドです。`BR_TRANSACTION` はサービスのメソッドの処理を行う所です。これはサービス実装者にしか何をやりたいのかは分かりません。だから `IPCThreadState` は皆に共通と思われる事だけをやってくれて、後はサービスの実装者に実装を任せます。

`IPCThreadState` がやってくれる、皆に共通と思われる事は大ざっぱには以下の事となります。

1. メッセージ受信対象オブジェクトと引数のデータを取り出す
2. メッセージ受信対象オブジェクトの `transact()` メソッドに引数データを渡して呼び出す
3. 結果を `BC_REPLY` として送り返す

ようするにバイト配列からオブジェクトを取り出してメソッドを呼び、またバイト配列にして送り返す、という事をやります。`transact()` メソッドにするのに必要な事を全部やってくれる、と思っておくのが良いですね。

もう少し厳密に書くと、以下のような手続きとなります。

1. `binder_write_read` の `read_buffer` から、`binder_transaction_data` を取り出す
2. `binder_transaction_data` からメソッドを表す ID を取り出す (`code` と呼ぶ)
3. `binder_transaction_data` からメソッドの引数に相当するデータを `Parcel` に詰める (`buffer` と呼ぶ)
4. 呼び出し元の `uid` や `pid` を取り出してフィールドに保存
5. `tr.target.ptr` を `BBinder*` にキャストする
 <fn>厳密には `tr.target.cookie` だが概念的には同じ</fn>

6. 5 でキャストした BBinder の transact を呼び出す。引数は 2 のメソッドを表す ID(code)、3 の Parcel、あとは結果を入れる空の Parcel(reply と呼ぶ)
7. 6 を呼び出した結果の reply を、BC_REPLY コマンドとして呼び出し元に返信する

それほど複雑な処理でもないのでソースコードを読んでみても良いのですが、上記の説明とそれ程変わらないコードなのでここには載せません。興味のある方は IPCThreadState.cpp の getAndExecuteCommand() メソッドの周辺を読んでみてください。

とにかく、IPCThreadState の joinThreadPool() メソッドによるループは、コマンド ID に応じて行える処理は全て行って、BR_TRANSACTION の時には必要なデシリアライズや結果の返信は引き受けた上で、transact() メソッドを呼び出して後はサービス実装者に任せる、という振る舞いをします。

そこでサービスを実装する側としては、BBinder の transact() を実装すればいい訳です。そしてそのメソッドの処理の結果は reply という引数の Parcel に書き込んでやると、IPCThreadState は勝手に BC_REPLY として結果を返信してくれます。

このように、transact() メソッド以外の部分は BBinder の基底クラスと IPCThreadState で勝手に処理してくれます。そこで次には、BBinder の transact() とはどういうメソッドか？ という話になります。

6.5 8.6.4 BBinder の transact() と onTransact()

BBinder はリファレンスカウントに応じた寿命処理と、transact() メソッドを持ったクラスです。
<fn>寿命管理周辺は別段難しい事も無いコードとなっているので本書では扱いません。</fn>

IPCThreadState による ioctl のメッセージループからは、BBinder の transact() メソッドが呼ばれる、と言いました。

BBinder の transact() メソッドはサービスの transact() の共通の処理を行います。そしてサービス固有の処理に関しては、BBinder 自身の onTransact() を呼びます。onTransact() はいわゆるテンプレートメソッドのデザインパターンです。

BBinder の onTransact() メソッドの型を見てみましょう。

リスト 6.7: onTransact() メソッドの型

```
status_t BBinder::onTransact(  
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
```

先頭の引数、code は binder_transaction_data の code に入っている、メソッドを表す ID です。これはサービスが独自に決めます。servicemanager なら SVC_MGR_ADD_SERVICE や SVC_MGR_CHECK_SERVICE などでした (8.4.4 参照)。

二番目の引数、Parcel の data はメソッドの引数のデータが入った Parcel です。

三番目の引数の reply は、結果を書き込む Parcel です。onTransact の中にメソッドの結果を書き込みます。これは return の値は正常に成功したかどうかのステータスコードを返す、という決まりになっているから、通常のメソッド呼び出しのように return で結果を返すという手段が使えない

第6章 [threadpool-layer] 8.6 スレッドプールの実装と BBinder の onTransact() の実装

です。

四番目の flags は呼び出しが oneway、つまり結果を受け取らなくて良い前提で呼び出されたか、それとも通常の結果が返るメソッド呼び出しかを表します。

onTransact の実装者としては、第一引数の code を見て switch し、その code に応じた引数を data から取り出して、結果を reply に write していければ良い訳です。

典型的なコードとして、int a と int b の結果を足した物を返す MYSERVICE_ADD と、引いた結果を返す MYSERVICE_SUB を持ったサービス、MyService1 を実装すると以下のようになります。

リスト 6.8: add と sub を持つ MyService1 の実装例

```
// BBinder を継承
class MyService1 : public BBinder {
    enum {
        // メソッド ID。IBinder::FIRST_CALL_TRANSACTION より大きい値を使う約束になっている
        MYSERVICE_ADD = IBinder::FIRST_CALL_TRANSACTION,
        MYSERVICE_SUB
    }
    ...

    // onTransact をオーバーライド
    virtual status_t onTransact(
        uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0) {
        switch(code) {
            case MYSERVICE_ADD: {
                // data から引数を取り出し。これはサービスごとに決める。
                // 今回は呼び出し側で int32 を二つ並べて書き込んでいる、と仮定している。
                int a = data.readInt32();
                int b = data.readInt32();

                // a+b を計算して reply に書く
                reply->writeInt32(a+b);
                return NO_ERROR;
            }
            case MYSERVICE_SUB: {
                // MYSERVICE_ADD と全く同様。
                int a = data.readInt32();
                int b = data.readInt32();

                // a-b を計算して reply に書く
                reply->writeInt32(a-b);
                return NO_ERROR;
            }
        }
        return BBinder::onTransact(code, data, reply, flags);
    }
};
```

このように作った MyService1 を new して servicemanager に登録すれば、クライアントはこのサービスを呼び出す事が出来るようになります。

引数を data から readInt32() したりする、というのは少しまだ低レベルな要素が残っています

が、この MyService1 の実装くらいまで来ると、だいぶ通信回りのコードは無くなって、提供する機能に集中出来るコードとなっていませんか？

6.6 8.6.5 サービスの呼び出し側のコードとプロキシの必要性

さて、上記のように作ったサービスを呼び出そう、と思ったとします。まずサービスのハンドルは 8.4.4 で説明した手順で取れます。より簡単な取得の方法についてはプロキシを扱った後に触れます（8.6.9 参照）。

プロキシを使わないサービス呼び出しの例

さて、この handle に対して binder_transaction_data を用意し、binder_write_read に詰めて ioctl をすれば、メソッド呼び出しが出来るのでした。（8.4.5 など参照）でもそれはとても長いコードになるので、毎回サービスを呼び出す都度やるのは大変です。例えばエラー処理などを省いても、以下のようなコードになってしまいます。

リスト 6.9: プロキシ無しのサービス呼び出し

```
int handle;

// 8.4.4 にあるようなコードで MyService1 のハンドルを取得したとする。
// コードは省略。

// 3+4 を計算させる。
int a = 3;
int b = 4;

Parcel trbuf;
// /* 1 */ 引数 a, b の書き込み
trbuf.writeInt32(a);
trbuf.writeInt32(b);

// ハンドルとメソッド ID を指定
struct binder_transaction_data tr;
tr.target.handle = handle;
// /* 2 */ 呼び出したいメソッドのメソッド ID
tr.code = MYSERVICE_ADD;

// 引数を設定
tr.data_size = trbuf.dataSize();
tr.data.ptr.buffer = trbuf.data();

// binder_write_read の初期化開始。
// binder_write_read に使う送信用バッファとして bwrbuf を初期化。
Parcel bwrbuf;

bwrbuf.writeInt32(BC_TRANSACTION);
bwrbuf.write(&tr; sizeof(tr));

// binder_write_read の受信に使うバッファ。
Parcel readbuf;
```

```

readbuf.setDataCapacity(256);

// binder_write_read に上記送信用、受信用バッファをセット。
struct binder_write_read bwr;

bwr.write_size = bwrbuf.dataSize();
bwr.write_consumed = 0;
bwr.write_buffer = bwrbuf.data();

bwr.read_size = readbuf.dataCapacity();
bwr.read_consumed = 0;
bwr.read_buffer = readbuf.data();

// ioctl呼び出し
res = ioctl(fd, BINDER_WRITE_READ, &bwr);

// 結果は readbuf 内のバイト配列内に入っているが、長さを教えてやる必要がある。
readbuf.setDataSize(bwr.read_consumed);
readbuf.setDataPosition(0);

int32_t cmd = readbuf.readInt32();
assert(cmd == BR_REPLY);

// binder_write_read から binder_transaction_data を取り出し、その中のバイト配列を resultbuf に
// セットする。
binder_transaction_data tr2;
readbuf.read(&tr2, sizeof(tr2));

Parcel resultbuf;
resultbuf.ipcSetDataReference(tr.data.ptr.buffer, tr.data_size, NULL, 0);

// /* 3 */ a+b の結果の取り出し。つまり 7 が入ってる
int result = resbuf.readInt32();

```

こんなコードになってしまいます。binder ドライバの復習として全体のコードを見てみたい、という時ならいざ知らず、ただ $3+4$ を計算させる為に毎回こんなコードを書くのは大変ですよね。

そこでサービスの提供者は上記と同じ事をするコードを、サービスの実装と一緒に提供する事になっています。それがサービスプロキシです。

プロキシを使ったサービス呼び出しの例

サービスのプロキシのインターフェースは、直接呼びたいメソッドの形で定義します。上記の MyService1 のプロキシなら、以下のようになります。

リスト 6.10: プロキシのインターフェース

```

class 何かのクラス {
    int add(int a, int b);
};

```

このインスタンスを作って、このメソッドをただ呼べば良いように作ります。名前は、普通はサービスの名前の前に Bp をつけた物にする約束です。

リスト 6.11: サービスプロキシの名前は、Bp から始めるコンベンション

```
class BpMyService1 {
    int add(int a, int b);
};
```

サービスを呼び出す為にはハンドルが必要です。そこで、コンストラクタでハンドルを渡すようになる事になっています。

リスト 6.12: サービスプロキシのコンストラクタを追加

```
class BpMyService1 {
public:
    BpMyService1(int32_t handle);
    int add(int a, int b);
};
```

実装はおいといて、使う側としては、以下のように使えます。

リスト 6.13: サービスプロキシを用いてサービスを呼び出す例

```
int handle;
// 今回も handle は 8.4.4 のようなコードで習得済みとする。

sp<BpMyService1> myservice = new BpMyService1(handle);

// MYSERVICE_ADD を呼び出す
int result = myservice->add(3, 4);
```

こんな風に使える、BpMyService1 をサービスと一緒に提供します。この ByMyService1 をプロキシクラス、と呼びます。

ByMyService1 を実装するのは、原理的には先ほどのプロキシ無しのコードと同じ事をすれば良い訳です。binder_transaction_data と binder_write_read を適切に初期化して ioctl を呼び、結果を返します。

ただ、これも毎回全部書くのは大変です。これでは大変なのがサービスを呼ぶ人からサービスを実装する人に移っただけです。そこでプロキシを実装するのを支援してくれるクラスが提供されています。これが BpBinder です。

そこで以下では、この BpBinder について見てきましょう。

6.7 8.6.6 サービスのプロキシと BpBinder の使い方

前項のメソッドの呼び出しの長いコード「プロキシ無しのサービス呼び出し」を見ていくと、呼び出すメソッド特有な処理は以下の三つだけである事に気づきます。

1. 引数を trbuf に詰める所
2. メソッドごとのメソッド ID
3. 結果を取り出す所

これ以外の処理は、基本的にはどのサービスのメソッド呼び出しても同じです。そこで上記の作業だけ自分でやって、それ以外は BpBinder::transact() を呼ぶ、というのが、BpBinder の使い方です。

例えば BpBinder を使うプロキシの最小限な物は、以下のようになります。

リスト 6.14: BpBinder を用いたプロキシの実装例

```
class MyService1Proxy {
public:
    MyService1Proxy(int32_t handle) : mRemote(handle) {}
    BpBinder mRemote;

    int add(int a, int b) {
        Parcel data, reply;

        // 1. 引数を data に詰める
        data.writeInt32(a);
        data.writeInt32(b);

        // 2. BpBinder の transact をメソッド ID をつけて呼び出す
        mRemote.transact(MYSERVICE_ADD, data, &reply);

        // 3. 結果の取り出し
        return reply.readInt32();
    }
};
```

add の実装はこれだけです。前項のコードに比べるとずっと短くなりましたね。そしてメソッドに特有の三つの部分だけの実装となっている事が分かります。

このようなプロキシクラスさえ提供されていれば、MyService1 を使うのは簡単です。MyService1 サービスを使う人は、handle をコンストラクタで渡して、このプロキシの add を呼びだせば良いのです。

リスト 6.15: MyService1Proxy を使うコード例

```
int handle;
// handle は 8.4.4 のコードで取り出してあるとする。
```

```
// プロキシクラスのコンストラクタに handle を渡す
MyService1Proxy myservice(handle);

// プロキシのメソッド呼び出し
int result = myservice.add(3, 4);
```

サービスの使用もだいぶ簡単になりました。このように、BpBinder を用いる事で、サービスのプロキシを簡単に実装出来る事が分かりました。

6.8 8.6.7 BpBinder の実装 - transact() メソッドと IPCThreadState

BpBinder はハンドルを渡して初期化し、transact() メソッドを呼んでメッセージを送信する、という話をしました。

実際の実装は、実は BpBinder 自身は ioctl を呼び出しません。その代わり、現在のスレッドの TLS に入っている IPCThreadState に処理を任せます。

エラー処理を省くと以下のようなコードになっています。

リスト 6.16: BpBinder::transact() メソッド

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    status_t status = IPCThreadState::self()->transact(
        mHandle, code, data, reply, flags);
    return status;
}
```

BpBinder は IPCThreadState の参照を受け取ったりしていないのですが、IPCThreadState はスレッドローカルなシングルトンなので、いつでもこのように IPCThreadState::self() で取得することができます。

IPCThreadState の transact は mOut に引数の処理を書いた後に joinThreadPool() メソッドとほぼ同じ処理を一回だけ行う、という振る舞いをします。

こうして、プロキシの実装者から見ると、BpBinder は transact を呼ぶと binder_write_read や binder_transaction_data を自分で設定して呼び出して結果を取り出す、という事をやってくれます。BpBinder を使えば実際の ioctl 周辺の処理を自分で書く事無く、全てこのスレッドプールのレイヤが処理してくれます。

こうしてサービスの実装者も使用者も、細かいプロセス間通信のコードを書く事無く、サービスの実装とプロキシの実装を提供出来るようになりました。

6.9 8.6.8 servicemanager のプロキシ - IServiceManager

BpBinder とプロキシの説明を終えたので、servicemanager のプロキシについて見ておく事にします。servicemanager のプロキシは最初から Android のフレームワークの中に含まれています。

servicemanager はハンドル 0 番に固定されているので、プロキシとして最初からメソッド呼び出しが出来ます。具体的には 8.4.4 で行っているコードと同じ事をすれば良いのですが、これをプロキシ化したクラスに、BpServiceManager というクラスがあります。通常はその基底クラスである IServiceManager を使う事になっています。

ハンドルを取得せずに使う事が出来るので、IServiceManager のインスタンスはグローバル関数で簡単に取得出来るようになっています。そのグローバル関数の名は、defaultServiceManager() です。

リスト 6.17: defaultServiceManager の宣言

```
namespace android {
    sp<IServiceManager> defaultServiceManager();
}
```

sp は StrongPointer の略で、スマートポインタです。寿命管理をしてくれる以外は普通のポインタとして使えます。コラム「リファレンスカウントとスマートポインタ - RefBase と Weak Reference と sp」

■コラム: リファレンスカウントとスマートポインタ - RefBase と Weak Reference と sp

リファレンスの寿命管理として、Android では RefBase というユーティリティクラスが提供されています。これは寿命管理では良く出てくる、Weak Reference と通常のリファレンスを管理する基底クラスです。Android では Weak Reference じゃない通常の所有を、Weak Reference の反対として Strong Reference と呼びます。RefBase には、incStrong(), decStrong() のようなリファレンスカウントの上げ下げを行うメソッドと、incWeak(), decWeak() という Weak Reference のリファレンスカウントを上げ下げするメソッドがあります。

そしてこれらのリファレンスカウントのメソッドを使って寿命管理をするスマートポインタの一つが sp です。sp は Strong Pointer の略で、Weak でないリファレンス、つまりカウントが 0 になるとそのオブジェクトを削除するオーナーシップを表します。

RefBase 自身は Binder とは関係無く使える汎用のユーティリティクラスですが、Binder 関連のクラスは RefBase を継承している物がほとんどなので、Binder 関連のコードではこの sp は良く出てきます。====[/column]

こうして取得した IServiceManager で、良く使うメソッドは以下の二つです。

リスト 6.18: IServiceManager の addService() と checkService() メソッドの宣言

```

class IServiceManager {
public:
    // サービス登録に使うメソッド
    virtual status_t addService( const String16& name,
                                const sp<IBinder>& service,
                                bool allowIsolated = false) = 0;

    // サービス取得に使うメソッド
    virtual sp<IBinder> checkService( const String16& name) const = 0;
};


```

addService() と checkService() の二つのメソッドです。これらは、8.4.4 で挙げた servicemanager の二つのメソッド ID、つまり SVC_MGR_ADD_SERVICE と SVC_MGR_CHECK_SERVICE に対応したプロキシメソッドです。

checkService() の結果は、ハンドルを返すのではなく、それを BpBinder にラップした物を返します。型はそのスーパークラスの IBinder を返す事になっています。何故 BpBinder でなく IBinder なのか、については 8.6.10 で扱います。

なお、何回か自動でリトライする getService() というラッパも存在します。こちらの方が便利なので通常はこちらを使いますが、本質的には checkService メソッドと同じ事を行います。

6.10 8.6.9 IServiceManager を用いてサービスのハンドルを簡単に取得する

比較の為、8.4.4 で述べた servicemanager 呼び出しでサービスのハンドルを取得するのと同じコードを、IServiceManager でも書いてみましょう。

defaultServiceManager() でプロキシを取得し、checkService() か getService() で取り出せば良い、という事になります。8.4.4 と同様に "SurfaceFlinger" サービスを取得する場合のコードは以下のようになります。

リスト 6.19: getService() の使用例

```

sp<IBinder> binder = defaultServiceManager()->getService(String16("SurfaceFlinger"));
int handle = binder->remoteBinder()->handle();

```

IBinder や BpBinder の詳細はここでは重要では無いので、すぐにハンドルを取り出してしまいます。普段は BpBinder を取得すれば十分なのでわざわざ handle() まで取得する事はありませんが、必要であれば上記 2 行で簡単に handle も取得出来ます。

6.11 8.6.10 IBinder とは何か？ - SVC_MGR_CHECK_SERVICE でハンドルが返ってこない 場合

getService() メソッドの結果は BpBinder では無くて IBinder といいう基底クラスだと
言いました。いつも BpBinder であれば最初から BpBinder を返せば良いのですが、実は
SVC_MGR_CHECK_SERVICE では、ポインタが返ってくるケースがあります。それは検索
するサービスが自分のプロセスのサービスの場合です。この場合は BBinder がそのまま返ります。

例を挙げましょう。例えば以下のコードでは、addService() で渡したポインタがそのまま帰って
きます。

リスト 6.20: addService() したのと同じ場所で checkService() する例

```
// /* 1 */ 登録するサービスのポインタ
MyService1* service1 = new MyService1();

sp<IServiceManager> svcmgr = defualtServiceManager();
svcmgr->addService(String16("com.example.MyService1"), service1);

// /* 2 */ checkService() で取得したオブジェクト
sp<IBinder> result = svcmgr->checkService("com.example.MyService1");
```

この場合、/* 1 */ の service1 と /* 2 */ の result は同じインスタンスを指します。

これくらい単純なケースだと、いちいち IServiceManager に問い合わせたりせず、最初から
service1 を使えばいいじゃないか、という気がしてしまいますが、現実にはもっとずっと複雑なケー
スもありうるのです。

本質的にはこれは checkService() に限らず、binder ドライバに BINDER_TYPE_HANDLE の
flat_binder_object を送る場合にいつでも起こり得ます。BINDER_TYPE_HANDLE を送信し
た先が、そのハンドルの元となるポインタの存在するプロセスの場合、BINDER_TYPE_BINDER
に変換されて送信されます。

この事をもう少し詳しく見てていきましょう。プロセス A にサービスのポインタが存在し、プロセ
ス B とやり取りする場合を考えましょう。

プロセス A から生のポインタをプロセス B に送る時は、flat_binder_object に
BINDER_TYPE_BINDER を入れるのでした。(8.5.3, 8.5.4) すると、プロセス B の側では
BINDER_TYPE_HANDLE として渡ってきます。

8.6.10 ①

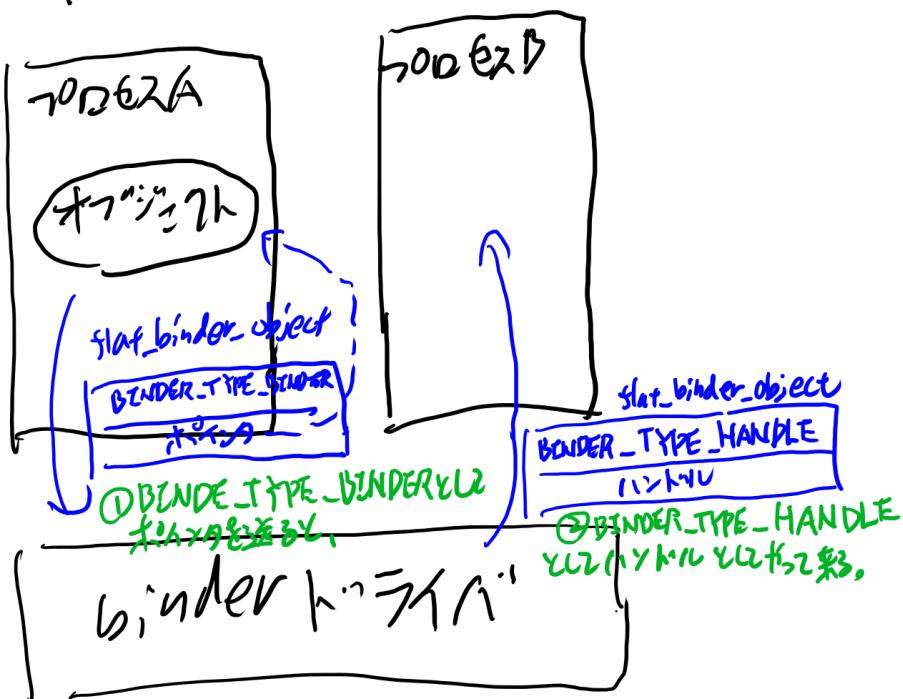


図 6.3 BINDER_TYPE_BINDER を送ると BINDER_TYPE_HANDLE として出てくる

このハンドルを今度は逆にプロセス B からプロセス A に送る場合を考えます。この場合は、逆に BINDER_TYPE_HANDLE が、BINDER_TYPE_BINDER に変換されてプロセス A の側に出てきます。

8.6.10 ②

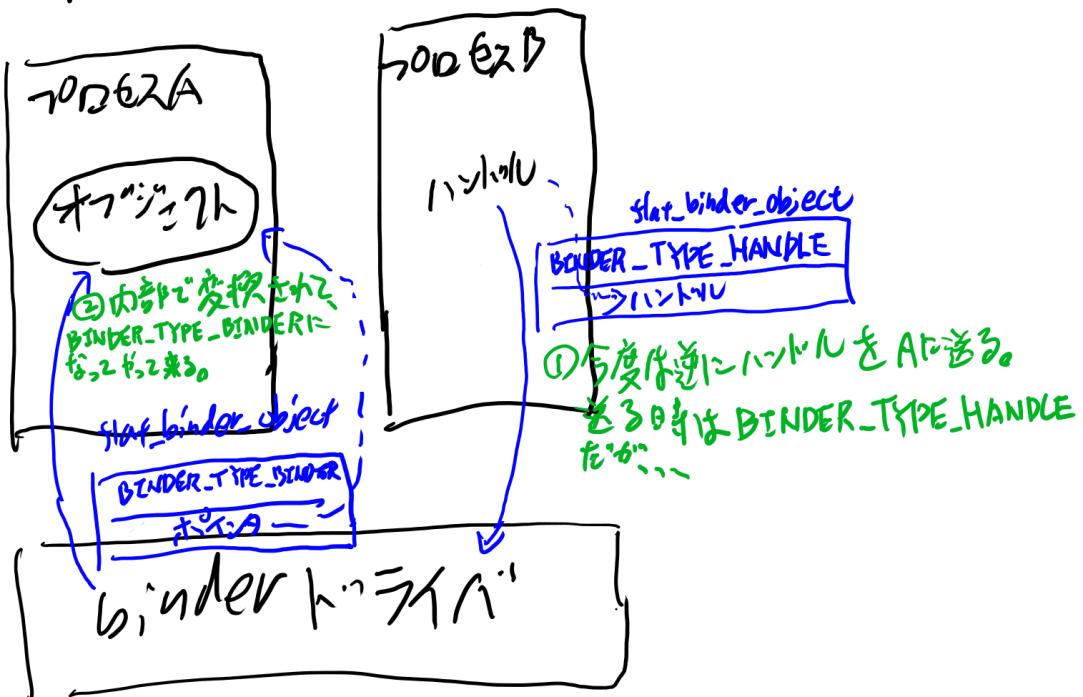


図 6.4 BINDER_TYPE_HANDLE を送ると BINDER_TYPE_BINDER として出てくる

二つのプロセスの場合では自明にも思えるかもしれないですが、プロセス C を足してみます。
プロセス A が B に送ります。すると BINDER_TYPE_BINDER が BINDER_TYPE_HANDLE となります。

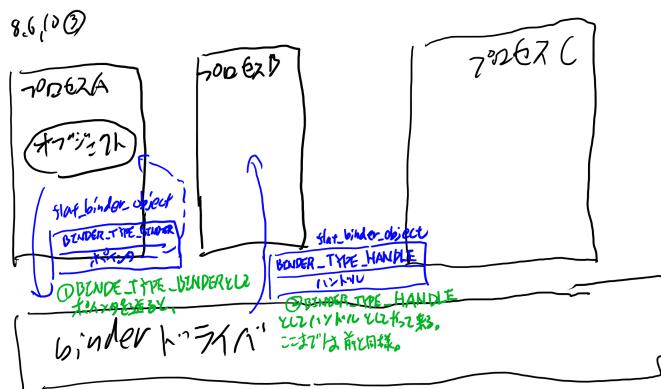


図 6.5 BINDER_TYPE_BINDER を送ると前回同様、BINDER_TYPE_HANDLE として出てくる

プロセス B がプロセス C に BINDER_TYPE_HANDLE を送ります。するとプロセス C でもこれは BINDER_TYPE_HANDLE のままでです。

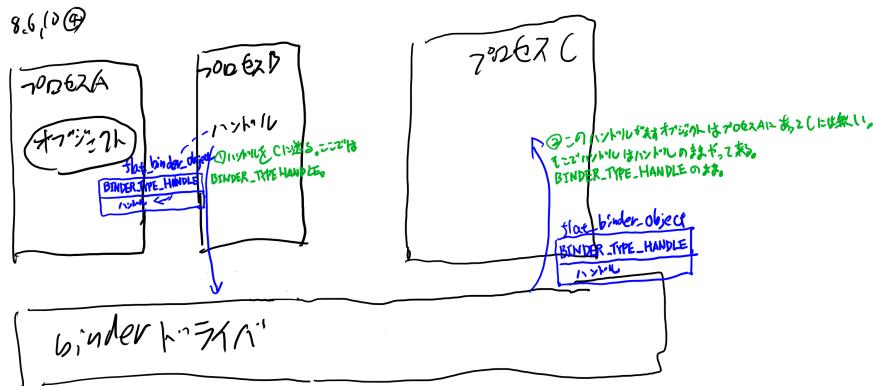


図 6.6 BINDER_TYPE_HANDLE を送ると、今回は BINDER_TYPE_HANDLE のまま出てくる

図 6.7「BINDER_TYPE_HANDLE を送ると、今回は BINDER_TYPE_HANDLE のまま出てくる」と図 6.5「BINDER_TYPE_HANDLE を送ると BINDER_TYPE_BINDER として出てくる」の違いに注目してください。どちらも BINDER_TYPE_HANDLE を送信していますが、受け取る側は図 6.5 が BINDER_TYPE_BINDER に変換されるのに対し、図 6.7 は

BINDER_TYPE_HANDLE のままでです。^{*1}

さて、ここからプロセス C からプロセス A にこのハンドルを送るとどうなるか？ というと、この場合はこのハンドルの表す生のポインタが所属するプロセスに戻ってきたという事なので、BINDER_TYPE_BINDER として名前のポインタが返ります。

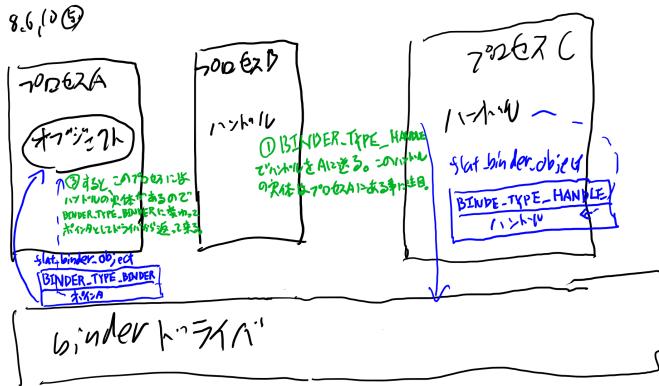


図 6.7 BINDER_TYPE_HANDLE を送ると、今度は BINDER_TYPE_BINDER として出てくる

このように、ハンドルを送信した結果がハンドルなのか BINDER_TYPE_BINDER に変換されるかは、相手のプロセスによります。

そこでクライアントとしてはサービスの参照に生のポインタでもハンドルでもどちらの場合でも共通に扱えるインターフェースを使う事になります。これが IBinder です。

IBinder は BBinder と BpBinder の共通の基底クラスです。

^{*1} なお、ハンドルの値はプロセスごとに異なります。図 5.11 を参照

8.6.10 ⑥

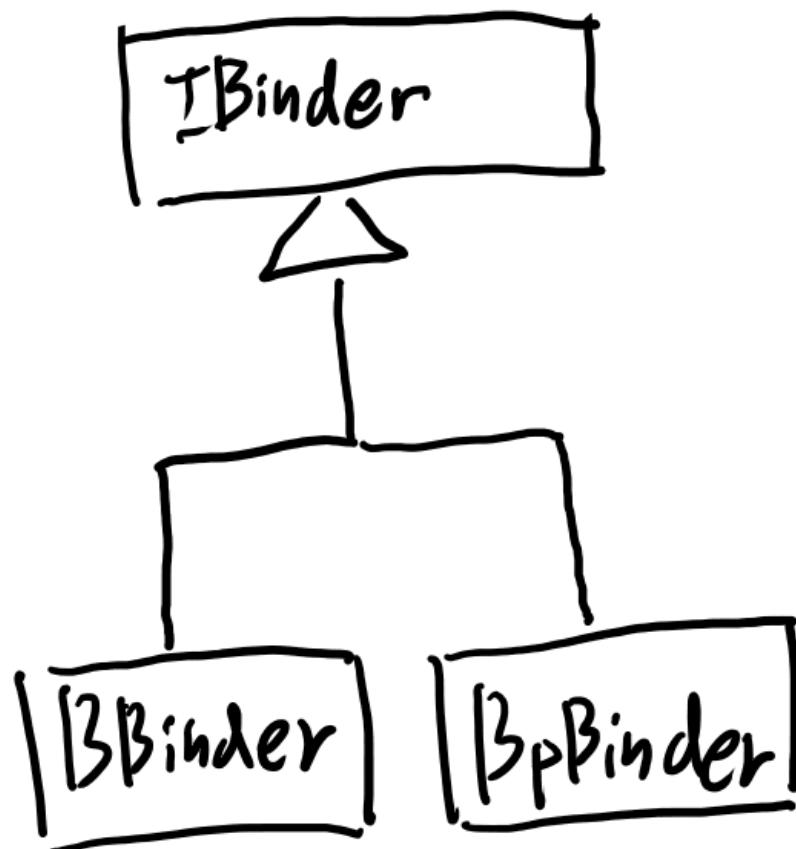


図 6.8 IBinder、BBinder、BpBinder のクラス図

IBinder が BBinder なのか BpBinder なのかを問い合わせるために、localBinder() というメソッドと remoteBinder() というメソッドが存在します。

リスト 6.21: localBinder() と remoteBinder() の型

```
class IBinder : public virtual RefBase
{
public:
    virtual BBinder*      localBinder();
    virtual BpBinder*     remoteBinder();
};
```

BBinder は localBinder で this を返し、remoteBinder で NULL を返します。BpBinder は localBinder で NULL を返し、remoteBinder で this を返します。

リスト 6.22: localBinder() と remoteBinder() の実装それぞれ

```
class BBinder : public IBinder
{
public:
    virtual BBinder*      localBinder() { return this; }
    virtual BpBinder*     remoteBinder() { return NULL; }
};

class BpBinder : public IBinder
{
public:
    virtual BBinder*      localBinder() { return NULL; }
    virtual BpBinder*     remoteBinder() { return this; }
};
```

これらのメソッドを用いる事で、どちらのインスタンスかをしる事が出来ます。

IServicemanager の checkService() などは、返ってきた flat_binder_object が BINDER_TYPE_BINDER か BINDER_TYPE_HANDLE かに応じて、BBinder のポインタを返すか BpBinder のインスタンスを生成して返すかを判断しています。

つまり以下のようないいコードになっている訳です。

リスト 6.23: IServiceManager の checkService() の概要

```
flat_binder_object *obj;
// obj は binder_transaction_data から取り出す。詳細省略

sp<IBinder> out;
switch(obj->type) {
    case BINDER_TYPE_BINDER:
        // obj->ptr と obj->cookie は概念的には同じ物が入っている
        out = reinterpret_cast<IBinder*>(obj->cookie);
        return out;
    case BINDER_TYPE_HANDLE:
        out = new BpBinder(obj->handle);
        return out;
}
```

上記コードで obj->ptr と obj->cookie が出てきますが、概念的には同じインスタンスが入っています。実際には obj->ptr には Weak Reference が、obj->cookie には実体のポインタが入っています。

6.12 8.6.11 ProcessState とスレッドプール

ここまで IPCThreadState の joinThreadPool() メソッドが ioctl を呼び出すループの処理を行っている、という話をしました。ですがこのクラスのどちらへんがスレッドプールなのか、という話はしていません。肝心のスレッドを開始するのが ProcessState クラスとなります。

ProcessState はシングルトンオブジェクトで、一プロセスにつき一インスタンスが対応し、プロセス全体のスレッドプールに関する情報を保持します。

ProcessState を取得するには、ProcessState::self() を呼び出します。するとまだインスタンスが出来ていなければ作成し、既にインスタンスがあればそのインスタンスを返します。

ProcessState クラスは、startThreadPool() というメソッドを持ちます。これが新しいスレッドを開始し、そのスレッドの中で IPCThreadState の joinThreadPool() を呼び出します。

リスト 6.24: startThreadPool() の実装

```
void ProcessState::startThreadPool()
{
    // 新しいスレッドを作り実行。スレッドのクラスは PoolThread。
    sp<Thread> t = new PoolThread(isMain);
    t->run();
}

// PoolThread は、新しいスレッドの中で IPCThreadState の joinThreadPool を呼ぶだけのスレッド。
class PoolThread : public Thread
{
protected:
    virtual bool threadLoop()
    {
        IPCThreadState::self()->joinThreadPool();
        return false;
    }
};
```

こうして、ProcessState の startThreadPool() メソッドを呼び出すと、新しいスレッドが作られて、そのスレッドの中では IPCThreadState の joinThreadPool() メソッドが呼ばれます。この joinThreadPool() は ioctl を呼び出して処理するループでした。(8.6.4)

ProcessState の startThreadPool() を何回か呼ぶと、呼んだ回数分だけスレッドが作られて、皆が ioctl で待ち状態になります。そして binder ドライバからメッセージがやってきたら処理を行つて、また ioctl で待ち状態に入る訳です。これはまさにスレッドプールです。

このように、実装のほとんどは IPCThreadState の joinThreadPool() メソッドではありますが、そのループをスレッドプールとして管理するのが ProcessState です。

6.13 8.6.12 システムサービスの main 関数と ProcessState - 独自のシステムサービスを提供する時のコード例

ProcessState にはスレッドプールの管理の他に、もう一つ役割があります。それは binder ドライバの open と mmap です。

サービスを提供するプロセスは、まず binder ドライバを open して mmap しなくてはいけないのでした。(8.3) これら open と mmap の処理は、ProcessState のコンストラクタで行います。

リスト 6.25: ProcessState のコンストラクタで binder ドライバの open と mmap が行われる

```
// open_driver() で open() が行われる。
ProcessState::ProcessState()
    : mDriverFD(open_driver())
...
{
    if (mDriverFD >= 0) {
        // open に成功していたら mmap を行う。
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE, mDriverFD, 0);
        ...
    }
}
```

このように、ProcessState を作成すれば、binder ドライバを使うのに必要な初期化は自動的に行われます。

ProcessState はシングルトンオブジェクトで、最初に static メソッドである ProcessState::self() が呼ばれた時にインスタンスが作成されます。以後の self() メソッド呼び出しはその作成した同じインスタンスが返されます。

こうして open したファイルディスクリプタはどこのスレッドからも参照出来る為、IPCThreadState から ioctl を呼ぶ時にも ProcessState から取り出して第一引数に渡す事が出来ます。具体的には ProcessState::self()->mDriverFD で参照出来ます。

この ProcessState のコンストラクタによる binder ドライバの open と mmap 処理も踏まえると、典型的な Service、例えば MyService を実装する C++ のコードを走らせる main() のコードは、簡易的に書くと以下のように書けます。

リスト 6.26: サービスを提供する exe の典型的な main() の内容

```
void main() {
    // (1) ProcessState のコンストラクタ呼び出し。
    ProcessState* ps = ProcessState::self();

    // (2) 別スレッドを立ち上げて ioctl のメッセージループ開始
    ps->startThreadPool();

    // (3) servicemanager にサービスを登録する、後述
    defaultServiceManager()->addService(String16("MyServiceName"), new MyService);
```

```
// (4) この main のスレッドも ioctl メッセージループを回し続ける事でスレッドプールのースレッドとして振る舞う。
IPCThreadState::self()->joinThreadPool();
}
```

(1) まずは先頭の ProcessState::self() の呼び出しで、ProcessState のコンストラクタが呼ばれます。ここで /dev/binder の open と mmap を行います。

(2) startThreadPool() 呼び出しで、新しいスレッドを立ち上げて、ここで ioctl をメッセージ受信の為に呼び出して、このスレッドはブロックします。

(3) その次にやるべき事は、MyService のインスタンスを new で作り、そのポインタをバインダドライバ経由で servicemanager に SVC_MGR_ADD_SERVICE メソッド ID で送る事で、このサービスを servicemanager に登録する事でした。

ここでは 8.6.8 で出てきた IServiceManager を使っています。この過程で MyService ポインタに対応した binder_node が作られ、別のプロセスからはこの binder_node を参照する事でポインタを識別できます。

(4) サービスのポインタを servicemanager に登録したので、このメインスレッドもやる事は無くなりました。そこでこのスレッドも有効利用すべく、ioctl 呼び出しを行うループを実行し、スレッドプールのースレッドとして振る舞います。

このコードでは (2) の startThreadPool() と (4) を合わせて、メッセージループは二つのスレッドとなりました。

これでサービスを提供するプロセスのやるべき事は終わりです。実際に記の main 関数とほとんど同じ内容の main 関数のプロセスは、Android のシステムサービスのプロセスでは良く見かけます。

説明の為にあえて分離しましたが、実際は (1) と (2) は 1 行で書く事が出来たため、main 関数はたったの 3 行となります。

6.14 8.6.13 サービスの仕組みとシステムの発展 - サービスの実装とプロセスの分離

ここまでで、Binder という仕組みのうち、スレッドプールのレイヤの解説が終わりました。先に進む前に、この時点で実現されているサービス、という物について、どういう特徴があるかを少し考えてみたいと思います。

Android ではハードウェアや新たなシステムの機能を追加する時は、システムサービス、という形で追加する事を推奨しています。システムサービスとは BBinder のサブクラスで、servicemanager に登録して使うものです。

サービス自身は何かのプロセス上で動きますが、一対一の関係では無く、一つのプロセスで複数のサービスを提供する事は可能ですし、また良く行われる事もあります。

システムサービスの実装は BBinder のサブクラスとして onTransact を実装し、さらにそれに対応したプロキシを BpBinder を用いて実装するだけです。このコードには main 関数で作った何かを

参照する必要は一切ありません。main 関数でこのサービスへの依存が発生するのは、addService() の引数だけです。(8.6.12 の (3) に対応)。

システムサービスがそれぞれ別のプロセスに分かれている方がロバストで安全なシステムにしやすいですが、一方でプロセスはメモリや CPU などのハードウェア資源を消費する物でもあり、組み込みのシステムであまり多くのプロセスを立ち上げるのは、重くなりすぎて使いものなりません。また、プロセス間通信もプロセス内のメソッド呼び出しそれぞれ重くなります。システムサービス相互のやりとりをなるべくプロセス内のメソッド呼び出しで済ますには、同じプロセスに多くのサービスが存在する方が良いという事になります。

■コラム: サーバープロセスとサービス

システムサービスを提供しているプロセスはサーバープロセスと呼ばれます、サーバーという用語はいろいろな場所で使われていてややこしいので、特に必要が無ければ本書では「サービスを提供しているプロセス」と呼ぶ事にしています。ですが、このサーバーという呼称を知っていると、ソースを読む時に便利な事もあります。例えば SystemServer プロセスは、Androidにおいてサービスを提供している重要なプロセスですが、本コラムで述べたようなサーバーという名前の使われ方を知っていれば、SystemServer という名前を見ただけでサービスを提供しているプロセスである事が推測出来ます。====[/column]

Android のシステムサービスは、どれくらいプロセスを分けるのか、という決断を、あまり実装に影響を与える事が出来る設計となっています。次の節で扱いますが、システムサービスの仕組みは、呼び出し側はサービスがどこのプロセスに属しているかを意識せずに使えるように作る事が出来ます。プロセスを分けていても他のパートのコードには影響を与えません。

まだ多くの端末でリソースが限られている時代の場合には一つのプロセスにたくさんのサービスを動かす事により、分散で無い通常のモノリシックなシステムのようにふるまい、少ないリソースでも動くように出来ます。

そして時代が進みハードウェアが発展てきて、メモリや CPU 資源が潤沢になっていくに応じて、重要なサービスを別のプロセスに分けてハードウェアスレッドを割り当てたり、障害に対してロバストにしていったりできます。

実際、Android はバージョンを重ねるごとにサービスのプロセスを分けていった歴史があり、その歴史は現在でも進行中です。

■コラム: surfaceflinger サービスに見る、システムの発展

surfaceflinger というシステムサービスがあります。詳細は 12 章で扱います。このサービスは、初期の頃はその他のシステムサービスと同様、SystemServer というプロセスに存在していました。初期の頃はほとんどのサービスがSystemService プロセス一つで実行されていました。少なくとも Android 2.3 の GB までは SystemServer プロセスにありました。

ですが、ハードウェアの進歩に伴い、おそらく Honeycomb の頃から^{*2} surfaceflinger は別プロセスに分かれる事になりました。ハードウェアスレッドの少ない端末ではパフォーマンスの低下がみられましたが、十分なハードウェアスレッドの存在する機種では、なめらかで引っかかるないアニメーションが実現されるようになりました。

このように、ハードウェアの進歩に合わせて柔軟にシステムのプロセス構成を変更していく、というのは、Android というシステムの大きな特徴と言えます。年々劇的な進歩を遂げてきた携帯電話というハードウェアの上で、時代の激変になんとか対応し続けて来られたのも、Android の基盤とも言えるシステムサービスの設計の段階で、このようなハードウェアの進歩に応じたシステムの発展がデザインされていたおかげである、と言えるでしょう。

また、1巻のコラムで触れる Stagefright バグとその結果の Media Framework の改善も、時代に合わせたプロセス構成の発展のまさに現在行われている例と言えると思います。

また、スレッドプールにどれだけのスレッドを用意するかもサービスの実装とは独立に決定出来ます。main 関数でたくさん startThreadPool() を呼べばスレッドプールに割り当てられるスレッドは多くなる訳です。サービスの実装側ではスレッドプールにスレッドが幾つあるかは、一切気にする必要はありません。

このように、サービスの実装は一切いじらずにプロセスやスレッド数といったリソースやシステム構成の設計を行えるのは、Android のシステムサービスという仕組みの重要な特徴と言えます。

第7章

8.7 共通ネイティブインターフェースの レイヤ - `IInterface` と `interface_cast`

前節までで、IPCThreadState と ProcessState を利用し、BBinder を継承したクラスを用いる事でサービスを提供できる、という話をしました。また、その提供されたサービスを利用する為のプロキシクラスを実装するのも、BpBinder を用いる事で簡単に出来る、という話をしました。

この節では、IBinder が BBinder なのか BpBinder なのかを気にせずにクライアントが同じコードを使えるようにするための、共通ネイティブインターフェースのレイヤについて説明します。^{*1} クラスとしては IInterface、BnInterface、BpInterface の三つです。

//footnote[mydef][なお、スレッドプールのレイヤも共通ネイティブインターフェースのレイヤも、レイヤの名前は私が勝手につけました。この周辺は公式のドキュメントが無いので、公式の呼び名が存在しません。]

共通ネイティブインターフェースのレイヤはネイティブ実装の時にだけ使われるレイヤで、Java で AIDL を使う場合はこのレイヤは使用しません。AIDL とは同じ階層の、兄弟の関係にあるレイヤと言えます。また、内容自体もこれまでのレイヤに比べると随分とシンプルで、大したことではありません。

大した事はないレイヤなのでそれほど重要でもないのですが、この説明を Java より先に持ってきたのにも訳があります。

まず、システムサービスは性質上、ネイティブで実装される物が多くあります。そしてネイティブの実装は、ほとんどがこの共通ネイティブインターフェースのレイヤを用いて実装されている為、システムサービスの実装ではこのレイヤしか出てきません。(main 関数で ProcessState と IPCThreadState が出てくるくらいです)。実装をするにせよソースを読むにせよ、この周辺を理解しておくと、分からぬ事は無くなります。逆に BBinder や BpBinder などは、概念としては重要度は高くともソースコード上の登場回数はむしろ少ないかもしれません。

実際にシステムサービスを実装する立場の人は比較的少数だとは思いますが、その人にとっては、本節は現存する中ではもっともしっかり書かれた文書だと自負しています。

^{*1} なお、スレッドプールのレイヤも共通ネイティブインターフェースのレイヤも、レイヤの名前は私が勝手につけました。この周辺は公式のドキュメントが無いので、公式の呼び名が存在しません。

第7章 8.7 共通ネイティブAPI呼び出しコードが共通である意義 - 共通のInterface Castの必要性

もう一つ、後の Java によるサービスの実装は、本節の内容をかなり忠実に Java にマップしています。そこで本節の理解は Java の側の理解の大きな助けとなります。Java のサービスは言語境界をまたぐ都合で、本節の共通ネイティブインターフェースのソースに比べて理解するのが困難だと思います。そこで一段簡単な本節で基本的な所を理解しておいて、次の AIDL の方のコードの準備としたい、という狙いもあります。

7.1 8.7.1 呼び出しコードが共通である意義 - 共通のインターフェースの必要性

8.6.13 でも触れた通り、サービスという仕組みは、時代と共にホストしているプロセスを分離していく、という事が想定されています。

その為には、呼び出し元のコードが呼び出し先のサービスが別のプロセスに居るか同じプロセスに居るかで、違いが無いようにしたい。

もともと携帯電話の OS においては、サービスごとに別々のプロセスとするのはパフォーマンス的に厳しい物がありました。だからサービスはなるべく一つのプロセスに入れたい、という思惑がありました。全てを一つのプロセスに入れて、必要なだけスレッドを割り当てれば、かつてのガラケーなどのITRON などにみられる RTOS と似たような構成となります。

一方で計算資源が許せば重要なサービスはプロセスが分かれている方が都合が良い事も多いものです。プロセスが分かれていれば、特定のサービスだけ優先度を上げたい時に、そのプロセスの優先度を上げれば良くなりますし、ハードウェアスレッドを特定のサービスには多く割り当てられるようにチューニングしたりも出来ます。プロセッサが複数ある時にキャッシュを汚しにくくする配置も可能です。また、セキュリティ的にもプロセスが分かれている方が望ましい事は多くあります。外部の入力を良く扱う所はバッファオーバーフローなどのセキュリティホールをつく事で、良く悪意のあるコードを挟みこまれがちです。そういういったサービスはプロセスを分けて低い権限で動かす方が、破られた時の被害がずっと少なくて済みます。

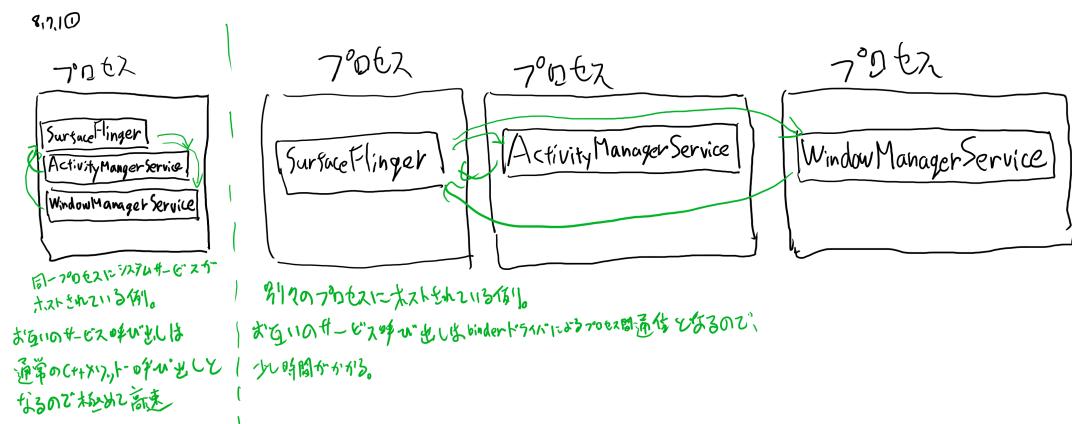


図 7.1 サービスが同一プロセスにある場合と別々のプロセスにある場合

携帯電話の計算資源が年々増えていくであろう事は以前から誰でも予想出来ました。一方で

第7章 8.7 共通ネイティブインターフェースのプロセスが同じか別かで変わる所 - IBinder の役割

Android の初期のバージョンが出た頃のスマホは、バイトコードを動かすだけで精一杯で、サービスをたくさんのプロセスで構成出来るほどの余裕はありませんでした。

そこで初期の Android は、サービスという物を一つのプロセスにたくさん集める、という方針でいきます。そしてハードウェア資源が潤沢になってきたら、状況に応じてサービスを別のプロセスに分けていく、と思ったのだと思います。

サービスは別のサービスを使って実装されている事も多くあります。むしろ機能ごとに別のサービスに分けて、オブジェクト指向プログラミングに則ったシステム設計にする事を推奨しています。様々なサービスが連携して目的を達成するのは、望ましい事なのです。

この時、自分のサービスが依存しているサービスが同じプロセスなのか、別のプロセスなのか、という事を区別してコードを書かなくてはいけない、となると、計算資源が潤沢になってきたらサービスを別のプロセスに分けていく、というのが大変になります。

そこで、共通ネイティブインターフェースのレイヤでは、サービスの使用者が、サービスが自分のプロセスに居るのか、別のプロセスに居るのかを区別せずにコードを書けるようになっています。

その為の仕組みが IInterface 関連のクラスです。

7.2 8.7.2 プロセスが同じか別かで変わる所 - IBinder の役割

サービスを呼び出す側のコードが、サービスが同じプロセスが別のプロセスかに関わらず同じになるようにしたい。その為には何が必要か、という事を考える為に、呼び出すサービスのプロセスが同じか別かで何が変わるか、という所を見ていきたいと思います。

8.6.10 でも触れた通り、binder ドライバからサービスを受け取る時に、サービスが同じプロセスだと BINDER_TYPE_BINDER として生のポインタが、そして別のプロセスだと BINDER_TYPE_HANDLE としてハンドルが返ってくるのでした。

ポインタの時にはただのオブジェクトなのだから、キャストして普通に使えば良い訳です。ハンドルの時にはプロキシオブジェクトを生成して、これを呼ぶのでした。(8.6.6)

具体例を見ましょう。まずサービスの実装として 8.6.4 の MyService1 を用いて説明します。サービスの取得は、IServiceManager の getService() を使えば良いのでした。

リスト 7.1: IServiceManager の getService() を使ってサービスを取得

```
sp<IBinder> service = defaultServiceManager()->getService(String16("com.example.MyService1"));
```

この service が BBinder 由来の物か BpBinder 由来の物か、つまり BINDER_TYPE_BINDER か BINDER_TYPE_HANDLE かで処理を分けます。

説明の都合で、先にプロキシから見ていきます。MyService1 のプロキシは 8.6.6 にありました。(都合によりコンストラクタの引数を変更してます)。

リスト 7.2: MyService1Proxy の実装、再掲

```
class MyService1Proxy {  
public:
```

```

MyService1Proxy(BpBinder* remote) : mRemote(remote) {}
BpBinder* mRemote;

int add(int a, int b) {
    Parcel data, reply;

    data.writeInt32(a);
    data.writeInt32(b);

    mRemote->transact(MYSERVICE_ADD, data, &reply);

    return reply.readInt32();
}
};

```

service が BpBinder ならこちらを使えば良い、という事になります。IBinder には localBinder と remoteBinder というメソッドがあって、この IBinder の実体が BBinder か BpBinder のどちらかが問い合わせ出来ます。(8.6.10)

リスト 7.3: IBinder の localBinder() と remoteBinder()

```

class IBinder {
    virtual BBinder*      localBinder();
    virtual BpBinder*     remoteBinder();
};

```

これを用いると、以下のように書けます。

リスト 7.4: BpBinder の時のプロキシ呼び出し

```

// 引数の例
int a = 3; int b = 4;

// 説明の為、結果を入れる変数を作つておく。
int result;

BpBinder* bp = service->remoteBinder();
if (bp != NULL) {
    sp<MyService1Proxy> proxy = new MyService1Proxy(bp);
    result = proxy->add(a, b);
}

```

さて、問題は bp が NULL の場合です。その時はこの IBinder は MyService1 のポインタでした。

リスト 7.5: IBinder に生のポインタが入っている場合、ポインタの取得まで

```
BpBinder* bp = service->remoteBinder();
if (bp != NULL) {
    // 既に述べたプロキシの処理、省略
} else {
    MyService1* ptr = (MyService1*)service->localBinder();
    // ...
}
```

さて、オブジェクトは無事得られました。このオブジェクトをどう呼んだら良いのでしょうか？そこで MyService1 のクラスを見ると、8.6.4 に実装がありました。以下のコードです。

リスト 7.6: MyService1 の実装、再掲

```
class MyService1 : public BBinder {
    enum {
        // メソッド ID。IBinder::FIRST_CALL_TRANSACTION より大きい値を使う約束になっている
        MYSERVICE_ADD = IBinder::FIRST_CALL_TRANSACTION,
        MYSERVICE_SUB
    }
    ...

    // onTransact をオーバーライド
    virtual status_t onTransact(
        uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0) {
        switch(code) {
            case MYSERVICE_ADD: {
                // data から引数を取り出し。これはサービスごとに決める。
                // 今回は呼び出し側で int32 を二つ並べて書き込んでいる、と仮定している。
                int a = data.readInt32();
                int b = data.readInt32();

                // a+b を計算して reply に書く
                reply->writeInt32(a+b);
                return NO_ERROR;
            }
            case MYSERVICE_SUB: {
                // MYSERVICE_ADD と全く同様。
                int a = data.readInt32();
                int b = data.readInt32();

                // a-b を計算して reply に書く
                reply->writeInt32(a-b);
                return NO_ERROR;
            }
        }
        return BBinder::onTransact(code, data, reply, flags);
    }
};
```

理論上は Parcel を使って transact を呼び出せば呼び出す事は出来ます。

リスト 7.7: transact を使ってローカルのロジックを呼び出す例

```
Parcel arg, reply;  
  
arg.writeInt32(a);  
arg.writeInt32(b);  
  
ptr->transact(MyService1::myservice_ADD, arg, reply);  
  
result = reply.readInt32();
```

ですが、生のポインタを持っていて型まで分かるのにこの呼び出し方は少し迂遠です。効率も良くありません。何より、これではプロキシ側と呼び出しコードがあまりにも違っていて共通化出来ません。

7.3 8.7.3 サービスをローカルからも使えるようにする

そこで普通に考えれば、add のロジックをリファクタリングして別メソッドにし、ローカル呼び出しの場合はこちらを呼び出すようにするのが普通です。

リスト 7.8: 変更前コード

```
case MYSERVICE_ADD: {  
    int a = data.readInt32();  
    int b = data.readInt32();  
  
    // /* 1 */ a+b を計算して reply に書く  
    reply->writeInt32(a+b);  
  
    return NO_ERROR;  
}
```

リスト 7.9: 変更後コード (add の実装は省略)

```
case MYSERVICE_ADD: {  
    int a = data.readInt32();  
    int b = data.readInt32();  
  
    // /* 2 */ a+b を計算して reply に書く、add 呼び出しに変更  
    int result = add(a, b);  
  
    // reply に結果を書く  
    reply->writeInt32(result);  
  
    return NO_ERROR;  
}
```

第7章 8.7 共通ネイティブインターフェースの8.7.3 オブジェクトをひとつの方法で使うようにする

/* 1 */ のコードを /* 2 */ に変える訳です。こうして add というメソッドを作つておいて、これを public にしておけば、MyService1 のポインタを持ったら、素直に add を呼ぶ事が出来ます。

リスト 7.10: ptr を使ってローカルのロジックを呼び出す例

```
result = ptr->add(a, b);
```

コードの説明ばかり続くと何をやっているのか分からなくなってくるので、ここでポインタのケースで何をやっていたのかを見直しましょう。

まず、BBinder は通常、IPCThreadState の ioctl 呼び出しループから呼び出されて、メッセージの処理をしていたのでした。これが onTransact メソッドでした。

一方でこの IPCThreadState からでは無く直接ポインタを取得出来る場合があるので、その場合の呼び出しをどうしよう？ という話で、その場合用にも public メソッドを作つておいて、onTransact からも直接呼び出しからもそのメソッドを使おう、という話です。

8.7.3①

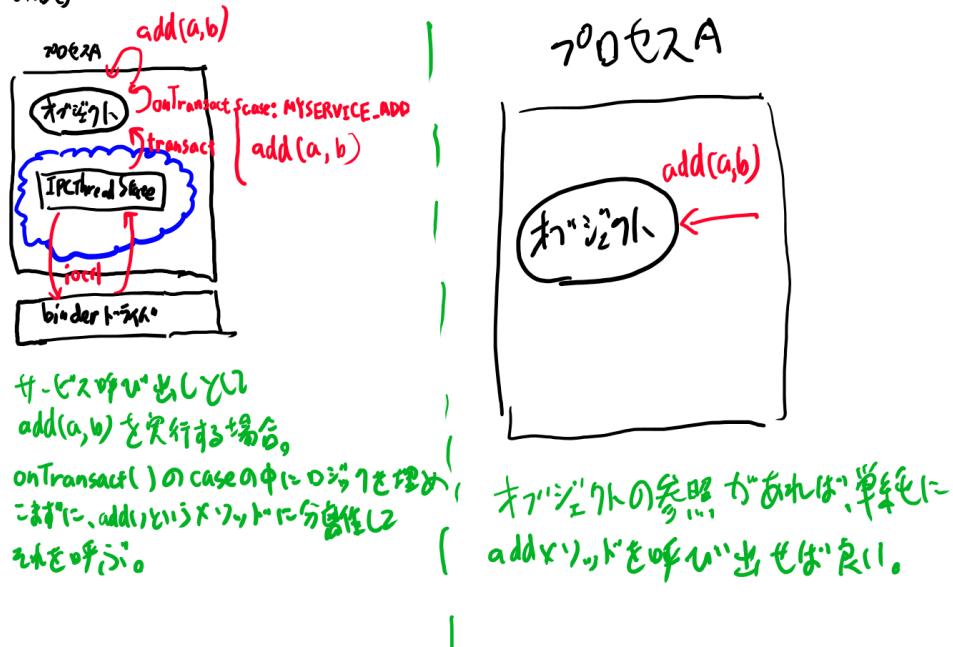


図 7.2 IPCThreadState から呼ばれるケースと直接呼ばれるケースを両方処理

プロキシと生ポインタ呼び出しのケースのコードを並べると以下のようになります。

リスト 7.11: IBinder がどちらかによって処理を分けるコード

```
BpBinder* bp = service->remoteBinder();
if (bp != NULL) {
```

```

sp<MyService1Proxy> proxy = new MyService1Proxy(bp);
// /* 1 */ プロキシの add 呼び出し
result = proxy->add(a, b);
} else {
    MyService1* ptr = (MyService1*)service->localBinder();
// /* 2 */ 生ポインタの add 呼び出し
result = ptr->add(a, b);
}

```

この if 文を除去するのは、通常のリファクタリングの話となります。

1 と 2 は完全にインターフェースが同じとなっています。そこで MyService1Proxy と MyService1 に共通のインターフェースがあれば、生成時だけどちらかを見て、以後はその共通のインターフェースにキャストして使えば良い、という事になります。

ここで全ての実装を書いてもいいのですが、それを理解出来る人はたぶん上記の説明だけでも理解出来るし、上記の説明だけで理解出来ない人はコードを見ても分からぬと思うので、説明だけにしておきます。

呼び出す側のコードを共通化するには、以下の条件が必要そうです。

1. プロキシも生ポインタも、同一のインターフェースを継承
2. 生ポインタは onTransact の switch 文からこの継承したインターフェースの呼び出しをするようにしておく
3. IBinder を取得した時に localBinder メソッドなどを用いて、プロキシか生ポインタかを選びインターフェースにキャスト

以上の事は別段ライブラリが無くても行う事が出来ますが、こうした事は紳士協定にしておくのでは無くて型システムで強制する方が全体としては一貫したシステムになります。

この手の強制の仕方は C++ の得意分野ですね。テンプレート引数を使ってこの形で書かないとコンパイル出来ないように設計する訳です。それが共通ネイティブインターフェースのレイヤの仕事で、具体的には IInterface、BnInterface、BpInterface の三つのクラスがやる事です。

7.4 8.7.4 IInterface 関連の三つのクラス - IInterface, BpInterface, BnInterface

ここからは呼び出し側のコードをローカルカリモートかによらずに同一にする為のレイヤである、共通ネイティブインターフェースのレイヤについて説明していきます。

このレイヤは IInterface, BnInterface, BpInterface の三つのクラスで構成されています。先頭の I, Bn, Bp で役割を表していると思われます。I はインターフェース、Bp は Binder のプロキシ、つまりプロキシです。Bn の n はなんだかわかりませんが、たぶんインスタンスか何かの n だと思います。とにかくサービスの実装を表します。

これまで出てきた IBinder、BBinder、BpBinder と役割分担は似ていますが、レイヤが一つ上になります。

第7章 8.7 共通な41874Interface関連の3つのクラスIBinderIIInterfaceBpInterfaceBnInterface

表 7.1 三つのクラスの対応関係

役割	IBinder 群	IIInterface 群	本節で実装するサンプルのクラス
インターフェース	IBinder	IIInterface	IMyService1
サービス実装の基底クラス	BBinder	BnInterface	BnMyService1
サービスプロキシのクラス	BpBinder	BpInterface	BpMyService1

IIInterface、BnInterface、BpInterface は、C++ のテンプレートを用いる事で、サービスの実装を共通インターフェースを通じて実装する事を半強制します。そうする事でローカルで生ポインタを直接使う場合にプロキシと同じインターフェースになるように型システムのレベルで強要します。

これまで作っていた add をするだけの MyService1 をこの IIInterface 関連クラスで再実装してみましょう。このクラス群を用いる人は、IIInterface、BnInterface、BpInterface をそれぞれ継承したクラスを作ります。ここではそれぞれ IMyService1、BnMyService1、BpMyService1 とする事にします。

8.7.4①

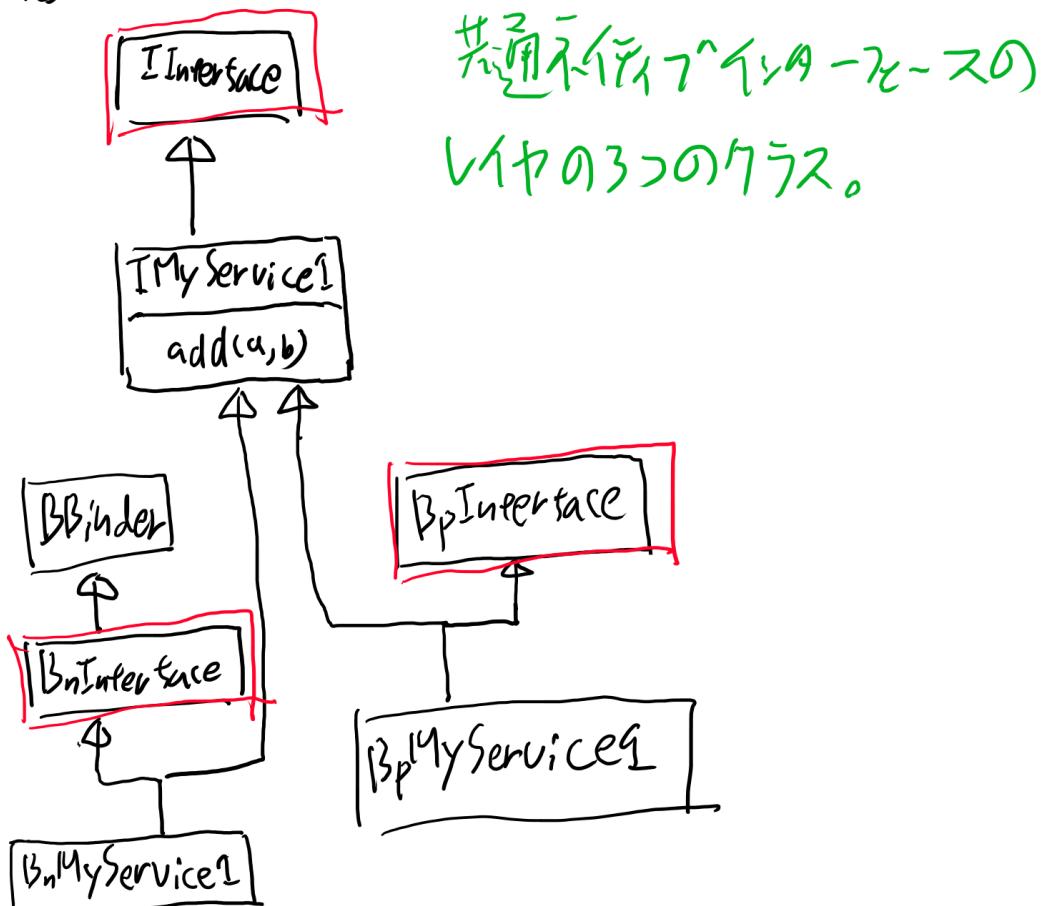


図 7.3 IInterface、BnInterface、BpInterface 関連のクラス図

これを実装する所でどのように IInterface 関連クラスが共通インターフェースを強制するのかを見ていきます。

7.5 8.7.5 テンプレートによる継承の強制

`BnInterface` も `BpInterface` もテンプレート引数を取ります。このテンプレート引数はインターフェースのクラスにする約束となっています。

宣言を見ると以下のようになっています。

リスト 7.12: BnInterface と BpInterface の定義

```

template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface> queryLocalInterface(const String16& _descriptor);
    virtual const String16& getInterfaceDescriptor() const;

protected:
    virtual IBinder* onAsBinder();
};

template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
    BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder* onAsBinder();
};

```

見ての通り、どちらも INTERFACE は継承に使っているだけで、中で参照している場所はありません。BnInterface と BpInterface を使う人に、インターフェースを実装する事を強制するためにこのテンプレート引数は存在しています。

MyService1 を実装する時に作るクラスの定義の例を見ると、以下のようになります。

リスト 7.13: MyService1 関連クラスの定義

```

// インターフェースのクラスは IInterface を継承しないといけない
class IMyService1 : public IInterface {
    ...

    // サービスとして実装するメソッド。サービスの実装者が好きに決める
    virtual int add(int arg1, int arg2) = 0;
};

// BnInterface を継承する時はインターフェースのクラスを渡す。そうするとこのクラスを継承した事になる。
class BnMyService1 : public BnInterface<IMyService1> {
    ...
};

// BpInterface を継承する時もインターフェースのクラスを渡す。
class BpMyService1: public BpInterface<IMyService1> {
    ...
};

```

このようにテンプレート引数を用意する事で、ここにインターフェースのクラスを渡さないと、このクラス群を使えないようにしています。意図的にこのデザインを迂回する為にダミーのクラス

第7章 8.7 共通ネイティブ API の IInterface と asInterface メソッド - IMyService1 クラス

を渡したりする事も可能ではありますが、普通に既存実装を参考に自分も実装しよう、と考えれば、自然と共通のインターフェースを両者が継承して実装する事になります。

これはドキュメントなどに書いて紳士協定としておくよりも、ずっと良い強制方法です。

このようなスタイルで実装をしていくと、自然と 8.7.3 のようなスタイルでコードを書く事になります。あとは、プロキシを生成するかキャストするかを関数で隠して、以後は IMyService1 だけ使えば良くなります。

リスト 7.14: IMyService1 へ変換する関数例

```
sp<IMyService1> asMyService1Interface(IBinder service) {
    BpBinder* bp = service->remoteBinder();
    if (bp != NULL) {
        return new BpMyService1(bp);
    } else {
        return (BnMyService1*)service->localBinder();
    }
}
```

この実装はどのサービスでもクラス名以外は全く一緒なので、マクロやテンプレートで自動生成出来ます。そこで最初から提供されています。それが IMyService1 の asInterface メソッドとなります。

そこで IMyService1 インターフェースクラスについて見ていきましょう

7.6 8.7.6 IInterface と asInterface メソッド - IMyService1 クラス

インターフェースのクラスである IMyService1 の実装を見てみます。

このクラスを定義する時には、幾つか決まりがあります。

1. IInterface を継承しないといけない
2. 宣言に DECLARE_META_INTERFACE(インターフェース名); という行を置かないといけない
3. 実装に IMPLEMENT_META_INTERFACE(インターフェース名, インターフェースディスクリプタ); という行を置かないといけない

これはそういう決まりになっている、という事で、特に中を理解していなくても問題は無いように設計されています。

例えばヘッダファイルは以下のようになります。

リスト 7.15: IMyService1 のヘッダ例

```
// /* 1 */ IMyService1 は IInterface を継承しないといけない
class IMyService1 : public IInterface {
    // /* 2 */ そしてヘッダには DECLARE_META_INTERFACE という行が必要。後述
```

第7章 8.7 共通ネイティブAPI: 8.7.6 InterfaceとasInterface メソッド MyService1 クラス

```
DECLARE_META_INTERFACE(MyService1);

// サービスとして実装するメソッド。サービスの実装者が好きに決める
virtual int add(int arg1, int arg2);
};
```

add の所は実装したいメソッドです。好きな名前で複数書く事が出来ます。サービスもそのプロキシも、この IMyService1 を（間接的に）継承する約束になっています。

ヘッダの中で、DECLARE_META_INTERFACE というマクロを置かないといけない決まりとなっています。引数はサービス名です。このマクロの展開結果は getInterfaceDescriptor() というメソッドと asInterface() というメソッドの宣言となっています。

また、この DECLARE_META_INTERFACE に対応するマクロとして IMPLEMENT_META_INTERFACE というマクロもあり、この IMPLEMENT_META_INTERFACE マクロは.cpp の方に含めなくてはいけません。interface といいつつ.cpp が必要なのです。

具体的には以下のような行を含めた.cpp ファイルをリンクする必要があります。

リスト 7.16: .cpp ファイル内の IMPLEMENT_META_INTERFACE マクロの例

```
IMPLEMENT_META_INTERFACE(MyService1, "com.example.IMyService1")
```

この IMPLEMENT_META_INTERFACE マクロは、getInterfaceDescriptor() と asInterface() の実装を吐くマクロです。

ここで第二引数の"com.example.IMyService1"はインターフェースディスクリプタと呼ばれる、このインターフェースを表す文字列です。この Android のマシン内で一意である事が期待されます。

このマクロが生成する asInterface() は、このサービスを表す IBinder を受け取って、IBinder が BINDED_TYPE_BINDER に対応する方か BINDER_TYPE_HANDLE に対応するかに応じて、

1. サービスと同じプロセスなら BnMyService1 を返す
2. サービスと異なるプロセスなら BpMyService1 を返す

というふるまいをするメソッドです。どちらも IMyService1 にキャストして返すので、外からは区別がつきません。

IMyService1::asInterface() メソッドは、その内部で BnMyService1 と BpMyService1 と名前を決め打ちしたクラスを参照します。そこでこのメソッドから参照出来る場所にこの二つのクラスが必要ですし、別の名前で定義する事は出来ません。MyService1、というサービスの名前を決めたら、三つのクラスは

1. IMyService1
2. BnMyService1
3. BpMyService1

第7章 8.7 共通ネイティブ API の実装と asInterface メソッド MyService1 クラス

に固定されてしまい、これに従って実装しないとコンパイルエラーとなります。

IMPLEMENT_META_INTERFACE は、具体的には以下のようなコードに展開されます。(簡略化しています)

リスト 7.17: IMPLEMENT_META_INTERFACE マクロの展開例

```
// このサービスを表す名前。インターフェースディスクリプタと呼ばれる。
const android::String16 IMyService::descriptor("com.example.IMyService1");

// asInterface の実装
android::sp<IMyService> IMyService1::asInterface(
    const android::sp<android::IBinder>& obj)
{
    android::sp<IMyService> intr;
    intr = static_cast<IMyService1*>(
        // obj の queryLocalInterface を呼び出す。詳細は後述だが、基本的には localBinder() と同じ。
        obj->queryLocalInterface(
            IMyService1::descriptor).get()
    );

    // intr が NULL という事は IBinder の localBinder() が NULL だったという事なので、この obj は BpBinder。
    // そこでプロキシオブジェクトを作つて返す
    if (intr == NULL) {
        intr = new BpMyService1(obj);
    }
    return intr;
}
```

asInterface はちょっとややこしいコードですが、要約すると以下のようになります。

1. IBinder が BBinder だったらただ IMyService1* にキャストして返す
2. IBinder が BpBinder だったら BpMyService1 でラップして返す

インターフェースディスクリプタというのは ioctl で送るコマンドの先頭に書く決まりになっていて、どのサービス実装者も書いている物です。実際 8.4.4 でもハードコードした文字列、として書いていました。servicemanager はこの文字列をチェックして、無かつたらエラーとする為です。

ですが、Android でこれを有効に使っている例を私は見つけられませんでした。そこで本書では、インターフェースディスクリプタについて、多くは解説しません。こういう決まりになっていて、みんな書いていて、みんなチェックしている。それ以上の事は私にもわかりません。

queryLocalInterface() は BnInterface ではインターフェースディスクリプタをチェックした上で this を返し、BpInterface では NULL を返します。IBinder の localBinder() とほぼ同じ実装となっています。つまり queryLocalInterface() が NULL じゃなければ、BnInterface のサブクラス、ひいては flat_binder_object が BINDER_TYPE_BINDER だったケースなので、これはサービスのポインタという事です。つまり BnMyService のポインタです。そこで IMyService にただキャストするだけでインスタンスを直接呼び出せる訳です。

■コラム: `queryLocalInterface()` メソッドとインターフェースディスクリプタの著者の勝手な憶測

本文でも解説している通り、`queryLocalInterface()` メソッドは現状では完全に `localBinder()` メソッドと同じ機能となっています。引数の文字列により各インターフェースを保持しているかを問い合わせるケースでは振る舞いが変わるのでですが、そういうユースケースは存在しません。普通の分散オブジェクトのシステムだと、この手の仕組みが必要なのは

1. 型の無い言語で動的に問い合わせる必要がある場合
2. インターフェースのバージョニング
3. 特定のインターフェースを実装しているオブジェクトにだけアスペクトを注入するようなコンテナライブラリの存在

というのが良くあるパターンです。バージョニングに関しては通常は他のサービスが動き続ける中で自分だけアップデート、みたいな状況でないと、この仕組みが必要とは思えず、その場合にわざわざ備えておく、というのは、わざわざローカルに特化した分散オブジェクトシステムを作り直した Android らしくないと思います。

そこで自分の予想では、これはスクリプト言語対応の為に入れた、というもの。何故現存しないスクリプト言語対応の為のコードが残っているのかは謎ですが、Binder 周辺の歴史のどこかでスクリプト言語を使っていた環境があって、結構なコードがその環境で書かれた後に、そのコード遺産をそのまま持ってきた結果、ここを修正するのが大変になりそのまま残っているんじゃないでしょうか。全て私の勝手な憶測なので本当の所は分かりませんが。===[/column]

NULL だった場合は `BpBinder` です。つまりハンドルが中に入っているだけで、サービスのクラスのポインタでは無く、ただの int 値です。そこでプロキシクラスである `BpMyService1` を作って `IMyService1` にキャストして返します。こうする事で、`ByMyService1` で 8.6.6 のような事をするように実装すれば、サービス呼び出しのコードが実現出来ます。

このメソッドのポイントとしては、どちらにせよ `IMyService1*` が返るので、利用者はこの `IBinder` が `BpBinder` なのか `BBinder` なのかを気にせずに、同じコードでサービスを呼べます。そこで、あるバージョンの Android からそのサービスが別のプロセスに変更になっても、全くコードを変更する必要はありません。

なお、この `asInterface()` メソッドですが、これを呼ぶフリーフローイング関数として `interface_cast` という関数が用意されています。

リスト 7.18: `interface_cast()` インライン関数

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

見る人が見ればこれだけで使い方は想像がつくと思いますが、詳細は zzz で扱います。

このように、IMyService1 という物を定義して、決まり通りマクロを置くと、asInterface() メソッドが使えるようになります。

7.7 8.7.7 サービスの実装とプロキシの実装 - BnInterface と BpInterface

基本的には asInterface() メソッドの説明で、この共通ネイティブインターフェースのレイヤの説明は終わりなのですが、ここは実際にサービスを実装する人が見るクラスでもあるので、簡単に他のクラスの実装も追って全体像を提示しておきましょう。

サービスの実装は BnInterface のサブクラスとして行います。プロキシの実装は BpInterface のサブクラスとして行います。実装内容は BBinder と BpBinder の時とほとんど変わりません。つまり、ほとんど 8.6.4 と 8.6.6 の二つの節の内容の繰り返しとなります。

サービスの実装 - BnInterface のサブクラスの実装

まずはサービスの実装側、つまり BnInterface を継承した BnMyService1 の実装を見てみます。サービスを実装する人は BnInterface を継承し、onTransact を実装しなくてはいけません。そしてメソッドの ID も定義します。

コードとしては以下のようなコードになります。

リスト 7.19: MyService1 の実装側の宣言（つまり BnMyService1 の宣言）

```
// /* 1 */ テンプレート引数で IMyService1 を渡す
class BnMyService1 : public BnInterface<IMyService1> {

    enum {
        // メソッド ID。IBinder::FIRST_CALL_TRANSACTION より大きい値を使う約束になっている
        MYSERVICE_ADD = IBinder::FIRST_CALL_TRANSACTION
    }

    virtual status_t onTransact( uint32_t code,
                                const Parcel& data,
                                Parcel* reply,
                                uint32_t flags = 0);
};


```

この onTransact についての説明は 8.6.4 の内容と完全に同じです。ただ、BBinder と大きく違う所としては、BnInterface には/* 1 */にあるように、インターフェースクラスを受け取るテンプレート引数がある事です。この結果、BnMyService1 は IMyService1 も継承するように展開されます。

コードとしては以下の BBinder のコードとほとんど変わりません。

第7章 8.7 共通ネイティブクラスとサービスの実装とBpInterfaceを用いたBnInterface

リスト 7.20: BnMyService1 の簡略化した定義

```
class BnMyService1 : public IMyService1, public BBinder {  
...  
};
```

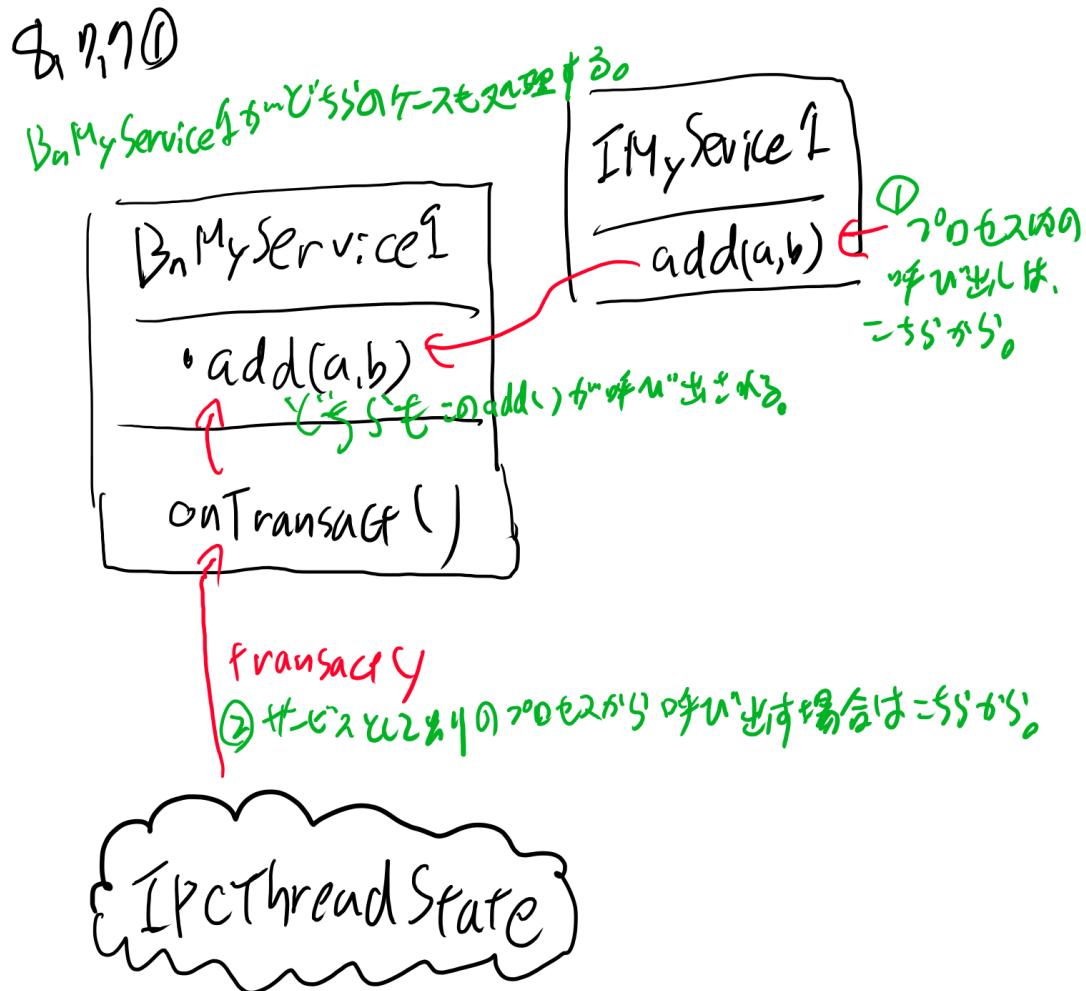
IMyService を継承した為、add も実装しないといけなくなりました。

リスト 7.21: add の実装

```
int BnMyService1::add(int arg1, int arg2) {  
    return arg1+arg2;  
}
```

同じプロセスからこのサービスが呼ばれる場合には、このメソッドが直接呼ばれます。

このように IMyService1 を BnMyService1 も継承する事で、同じプロセスなら直接 add を呼び、別のプロセスから来た場合は IPCThreadState の joinThreadPool から呼ばれる transact 呼び出しの仕組みを使いまわす、という事が一つのクラスで出来ています。

図 7.4 `IMyService1` から呼ばれる場合と `transact` から呼ばれる場合が同じメソッドに行き着く

BBinder 同様、別のプロセスから呼ばれる場合は `onTransact()` が呼ばれます。BBinder の時とあまり変わりませんが、一応完全を期す為に `onTransact` の実装例を書いておきます。

リスト 7.22: `onTransact` の実装

```
// onTransact をオーバーライド
status_t BnMyService1::onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0) {
    switch(code) {
        case MYSERVICE_ADD: {
            // data から引数を取り出し。
    }
}
```

```

// /* 1 */ IInterface を使う場合、先頭二つは strict policy とインターフェースディ
スクリプタ。後述。
int strictPolicy = data.readInt32();
String16 interfaceDescName(data.readString16());

// 残りはサービスごとに決める。
//呼び出し側で int32 を二つ並べて書き込んでいる、と仮定している。
int a = data.readInt32();
int b = data.readInt32();

// add を呼び出す。
int result = add(a+b);

reply->writeInt32(result);
return NO_ERROR;
}
}

return BnInterface::onTransact(code, data, reply, flags);
}

```

BBinder の時と違うのは、case 文で引数を取り出す時の /* 1 */ の所にある最初の二行くらいでしょう。一応そこだけ軽く説明しておきます。

/* 1 */ の所だけのコードを抜き出すと以下のようになっています。

リスト 7.23: 1 の所だけ抜粋

```

// /* 1 */ IInterface を使う場合、先頭二つは StrictMode とインターフェースディスクリプタ。後
述。
int strictPolicy = data.readInt32();
String16 interfaceDescName(data.readString16());

```

一行目の strictPolicy という変数で読みだしているのは、StrictMode という仕組みで使う値です。基本的には 0x100 が書かれます。これは thread が何か意図しないディスクアクセスなどをしないかチェックする為の機構ですが、本書ではこれ以上は扱いません。以下のリンクを参照ください。

<https://developer.android.com/reference/android/os/StrictMode.html>

インターフェースディスクリプタは IMPLEMENT_META_INTERFACE で渡した "com.example.IMyService1" が書いてあります。

この二つは別段使い道は無いので、ただスキップして捨てれば十分です。ですが、これらの値がちゃんと書いてある事を確認するマクロ、CHECK_INTERFACE という物があるので、自分で読み飛ばすよりは、これらを使う方が望ましいでしょう。

以下のような行を書けば、チェックしつつ読み飛ばした事になります。

リスト 7.24: CHECK_INTERFACE マクロの使用例

```
CHECK_INTERFACE(IHelloService, data, reply);
```

このマクロを呼び出すと、StrictMode を設定し、インターフェースディスクリプタが正しい事を確認してくれます。

それ以外は BBinder の時と変わりません。以上で BnInterface の説明は終わりです。

サービスプロキシの実装 - BpInterface のサブクラスの実装

サービスプロキシは、BpInterface を継承して実装します。内容は BpBinder の時の 8.6.6 とほとんど同じです。

まずは BpInterface を継承した BpMyService1 の宣言から見てみましょう。

リスト 7.25: MyService1 のプロキシ実装の宣言（つまり BpMyService1 の宣言）

```
class BpMyService1: public BpInterface<IMyService1>
{
...
};
```

BnInterface と同様に BpInterface もテンプレート引数を通じて IMyService1 を継承する事になっています。

BpInterface にはコンストラクタで IBinder が渡されます。IBinder は前にも言った通り実体は大きく BpBinder と BBinder のケースがあるのですが、BpInterface に渡されるケースは BpBinder の方です。BBinder の時は BnInterface 側のクラスにキャストされるだけなので、こちらのクラスには渡ってきません。

BpInterface クラスは、コンストラクタで渡された BpBinder を remote() というアクセサで参照できます。この BpBinder の transact() を用いてプロキシを作るのは、8.6.6 で解説した手順とほとんど同じです。

試しに add() を実装してみましょう。

リスト 7.26: MyService1 のプロキシの実装

```
// /* 1 */ テンプレート引数に IMyService1 を渡す
class BpMyService1 : public BpInterface<IMyService1>
{
public:
    // /* 2 */ コンストラクタは IBinder を受け取る物を用意する必要あり。実装はスーパークラスを呼び出すのみ。
    BpMyService1(const sp<IBinder> &remote) : BpInterface<IMyService1>(remote) {}

    // IMyService1 から継承した add の実装。remote()->transact() を呼ぶ。
    virtual int add(int arg1, int arg2) {

        Parcel data, reply;
        // まずは引数を準備。
```

第7章 8.7 共通ネイティブプロトコルのサービスの実装とBnInterface casts BpInterface

```
// /* 3 */ 最初は StrictMode とインターフェースディスクリプタを書く決まりになっている。  
以下の文で書かれる。  
    data.writeInterfaceToken(IMyService::getInterfaceDescriptor());  
  
    // あとは引数を書く。 int を二つ。  
    data.writeInt32(arg1);  
    data.writeInt32(arg2);  
  
    // メソッド ID MYSERVICE_ADD で引数は上で準備した物で呼び出し。  
    remote()->transact(MYSERVICE_ADD, data, &reply);  
  
    // 結果を取り出して返す。結果は int。  
    return reply.readInt32();  
}  
};
```

実装としては 8.6.6 とほとんど同じ内容となっています。違うのはテンプレート引数、コンストラクタ、writeInterfaceToken くらいでしょうか。この 3 つを簡単に解説します。

`/* 1 */ BnInterface` と同様、`BpInterface` もテンプレート引数で `IMyService1` を渡します。こうする事で、`IMyService1` を間接的に継承します。

`/* 2 */` コンストラクタとしては、`sp<IBinder>`を受け取るコンストラクタが必要です。というのは `IMPLEMENT_META_INTERFACE` マクロで生成される `asInterface()` のコードで、それを決め打ちで呼ぶからです。一般的には `const` リファレンスにします。そして実装はだいたいは基底クラスのコンストラクタを呼ぶだけです。

つまり、以下のコードになります。

リスト 7.27: プロキシ実装が要求されるコンストラクタ

```
BpMyService(const sp<IBinder> &remote) : BpInterface<IMyService>(remote) {}
```

`/* 3 */ add()` の中は、`writeInterfaceToken()` の所だけ 8.6.6 の実装（リスト 6.14）と異なっていますね。以下のコードです。

リスト 7.28: 前の例（リスト 6.14）との差分

```
data.writeInterfaceToken(IMyService::getInterfaceDescriptor());
```

`IMyService::getInterfaceDescriptor()` は、`IMPLEMENT_META_INTERFACE` マクロに渡した文字列、つまり "com.example.IMyService1" が返ります。

`data`、つまり `Parcel` の `writeInterfaceToken()` にインターフェースディスクリプタを渡して最初に呼ぶ、というのは、決まりとなっている、と言って良いでしょう。これを呼ぶと `StrictMode` とインターフェースディスクリプタという名の文字列が書かれて、前述のとおり `BnInterface` 側で `CHECK_INTERFACE` マクロを使うとこれらをチェックしてくれます。

実際に writeInterfaceToken() で何が書かれていて、CHECK_INTERFACE マクロでなにがチェックされているかはあまり気にする必要は無いと思います。

プロキシで大切なのは、前項の BnMyService1 の add で引数を読み出す処理と、今回の transact の実装で data に引数を書き込んでいる所で、順番や型などを一致させる、という事です。この二つの対応がとれていれば、中はどうでも良いのです。

全てのクラスが揃った時の使い方

以上で IInterface、BnInterface、BpInterface の 3 つのサブクラスが揃いました。これで asInterface() を使う事が出来ます。

リスト 7.29: 三つのサブクラスが揃っている場合の使い方

```
sp<IMyService1> intr = IMyService1::asInterface(defaultServiceManager()->getService(String16("MyService1"));

int result = intr->add(3, 4);
```

こうすると asInterface() メソッドは、getService() の結果が BpBinder 由来の IBinder だったらプロキシを作り、ポインタ由来だったら BnMyService1 にキャストして結果を返します。

呼ぶ側としてはこのインターフェースが同じプロセスにある生のポインタなのか、別のプロセスにあるサービスのプロキシなのかを気にする事無く、ただ add のメソッドを呼べば良くなります。

そして BBinder 由来の場合は、完全にローカルな BnMyService のポインタに対してメソッド呼び出しをするだけなので、分散オブジェクトのオーバーヘッドは一切無い、ただの C++ の仮想関数呼び出しで済みます。

さらに上の asInterface() 呼び出しと全く同じ事ですが、interface_cast という名前の inline 関数も定義されています。

リスト 7.30: 再掲: interface_cast() インライン関数

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

これを使うと、以下のように、キャストのように書けます。

リスト 7.31: interface_cast() バージョン

```
sp<IMyService1> intr = interface_cast<IMyService1>(defaultServiceManager()->getService(String16("MyService1"));
```

普通はこの interface_cast を使う方が読みやすく意図も分かりやすいので良いと思います。です

が asInterface() も Java の方でも同名のメソッドがあって役割は同じなので、知っておくと良いでしよう。

7.8 8.7.8 共通ネイティブインターフェースまとめ

以上で、共通ネイティブインターフェースのレイヤの説明を終わりました。

まず、IInterface を継承してインターフェースを作り、決まり事となっているマクロを置きます。その後 BnInterface と BpInterface を継承したクラスを用意すると、asInterface() メソッドと interface_cast 関数が使えるようになります。こうして呼ぶ側はサービスが同じプロセスに居るのか別のプロセスに居るのかを気にせずサービスを使う事が出来ます。

そしてサービスがローカルにあると通常の C++ のメソッド呼び出しとなるので、これは極めて高速に行われます。サービスに分けるコストというのはプロセスさえ同じならほとんどないと言つて良いでしょう。

共通ネイティブインターフェースの説明が終わったので、これで C++ でのシステムサービスの実装にかかるクラスライブラリの説明は全て終わりです。ここまで内容を理解した読者の皆様は、mediaserver などのネイティブのシステムサービスを読んだり、自分の端末のファームウェア用に専用のシステムサービスを追加するのに十分な知識を備えた事になります。

次の節からは、Java でシステムサービスを作る場合の手順に進みます。ほとんどが本章で解説した共通ネイティブインターフェースのレイヤと同じ構成なのですが、Java のシステムサービスの実装では、一つだけ大きく異なる所があります。

それが AIDL (Android Interface Definition Language) からの自動生成です。これを次に見ていきましょう。

TODO: 以下は一部ボツ。あとで一部（必要なら）サルベージ

サービスの実装と呼び出しは、理論的には binder ドライバに対して binder_write_read に適切なデータを入れて ioctl を呼べば良い、という事になります。ですが、それはあまりにも低レベルな為、Android ではその上に RPC(Remote Procedure Call) のレイヤがあります。

一般的に RPC は通常クライアント側のプロキシと、サーバー側のコードに分けられます。クライアントは普通引数をシリализして自身のメソッドを表すメソッド ID をメッセージに詰めて送ります。サーバー側はこのメソッド ID に応じて switch して、それぞれのメソッドの引数に応じて引数をデシリализしてメソッドを呼び出します。RPC のフレームワークは通常プロキシのコードを自動生成してくれて、また上記の switch する部分のコードも自動生成してくれます。サーバーを実装する人は、呼び出されるメソッドだけ実装すれば良い、という形になっています。

Android においてもそれらを自動生成してくれる仕組みがあります。aidl という IDL でインターフェースを定義すると、プロキシとサービスのディスパッチの部分を行う基底クラスを自動生成してくれます。

ここではこれらの自動生成の仕組みを理解する為に、自動生成される基底クラスに相当する物がどういう事をしているのかを見ていきます。

第 8 章

8.8 AIDL からの自動生成 - Java によるシステムサービスの実装

ここまで C++ によるネイティブのシステムサービスの実装を見てきました。ここからは、Java によるシステムサービスの実装を見ていきます。

サービスのコードは前節で見たように、BpMyService1 の transact() の呼び出しと BnMyService1 の onTransact() の呼び出しが対応しています。SurfaceFlinger サービスの BnSurfaceComposer などのネイティブサービスは上記の手法で自分でサービスを実装しています。

しかし、BnMyService1 も BpMyService1 も、かなりの部分がインターフェースからほぼ機械的に決まります。プロキシである BpMyService1 は IMyService1 のメソッドの定義通りに引数を Parcel に write して送信するコードを書くだけなので、インターフェースから全部のコードが完全に決ります。実装側である BnMyService1 も、Parcel から引数を read してメソッドを呼び出す部分まではインターフェースから機械的に決まり、呼び出されるメソッドだけがインターフェースからは自動では決まらない部分となります。

インターフェースとプロキシ、そして実装のデシリアルライズ部分の対応は機械的なので、人間がやるのは無駄であり、さらには対応が取れてないとバグになる為、インターフェースを変更すると直し漏れなどでバグが生じやすい所でもあります。そこで、人間はインターフェースを定義してその実装だけを書いて、自動的に決まる部分はなんらかの方法で自動生成する方が望ましいと言えます。

普通分散オブジェクトのシステムにはこれらのコードを自動生成する仕組みがあります。多くのシステムは、IDL から生成するか動的にコードを生成するかのどちらかです。

Android も例外ではなく、AIDL という IDL の方言を使ってインターフェースを記述しておくと、これをコンパイルして BnMyService と BpMyService に相当する部分の Java コードを自動生成してくれます。何故か知りませんが、C++ のコード生成は無く、Java しかサポートされていません。

ここからは、Java によるシステムサービスの実装を見ていきます。

余談になりますが、Java によるシステムサービスもネイティブによるシステムサービスと同様、ServiceManager が SDK に公開されていない為、通常のアプリ開発者は行う事が出来ません。あくまでシステムイメージを作るメーカー やカスタム ROM を作る人が行う作業で

す。ですが Java の場合、通常の SDK のサービスの時にも同様の手順で aidl を用いて作業をするので、かなりの部分はアプリ開発者でも行う事が出来ます。(詳細は 8.8.2 を参照)

8.1 8.8.1 Java のシステムサービスを支える基本的なクラス

Java のクラスでも、だいたいはネイティブのこれまで解説したクラスに対応したクラスがあります。

表 8.1 Java とネイティブのクラスの対応

java のクラス名	C++ のクラス名	役割
android.os.IInterface	IInterface	インターフェースの基底クラス
android.os.IBinder	IBinder	サービスのポインタとハンドルの双方の共通基底クラス
android.os.Binder	BBinder	サービスのポインタ
android.os.BinderProxy	BpBinder	サービスのハンドル

android.os.Binder や android.os.BinderProxy の実装は非常に分かりにくくて、ネイティブのシステムサービスの知識が無いとかなり難解ですが、ネイティブ側を知った上で読めばそのままをただ Java でラップしているだけ、という事が分かります。

8.2 8.8.2 AIDL とコンパイルとサービスの実装手順

IDL とは全般に、引数などの型だけ、つまりインターフェースだけを記述する言語です。普通はペースが簡単で、普通の言語よりも少しインターフェースに関する情報が多い傾向にあります。

AIDL の詳細については <https://developer.android.com/guide/components/aidl.html> を参照してください。

Java のシステムサービスの作り方は以下の手順になります。

1. AIDL でインターフェースを記述
2. aidl.exe でコンパイル
3. 生成されたコードを継承して残りを実装
4. ServiceManager に登録

なお、3 まではシステムサービスで無い通常の開発者が開発するサービス、いわゆる ActivityThread が管理する SDK のサービスでも同じ手順で作る事が出来ます。(ただしこの手順でサービスを作るのはスレッドモデルが複雑になり過ぎる為、通常は勧められていません <https://developer.android.com/guide/components/aidl.html> の冒頭の注意を参考の事)

AIDL は、以下のように書きます。

リスト 8.1: AIDL による MyService の定義

```
package com.example.myservice;
```

```
interface IMyService {
    int add(int arg1, int arg2);
}
```

これをSDKに含まれる aidl.exe に渡すと、IMyService というJavaのインターフェースと、その内部クラスとして IMyService.Stub というクラス、およびそのさらに内部クラスの IMyService.Stub.Proxy というクラスが実装されます。なお、AndroidStudioは拡張子 aidl のファイルがあると自動でこのファイルを生成します。

生成されるコードの詳細は後に回すとして、まずは実装と使い方だけ見てみましょう。

サービス実装者は、aidl をコンパイルしたら、「インターフェース名.Stub」というクラスを継承してメソッドを実装しなくてはいけません。今回の例ではインターフェース名は IMyService なので、以下のようになります。

リスト 8.2: AIDL をコンパイルした後は Stub を継承して実装

```
class MyService extends IMyService.Stub {
    public int add(int arg1, int arg2) {
        return arg1+arg2;
    }
}
```

このクラスをServiceManagerに登録すれば、サービスとして使う事が出来ます。ServiceManagerはネイティブの IServiceManager と同じ役割のクラスです。

IServiceManagerと同様、ServiceManagerはメーカーなどのシステム提供者が使う事が前提のクラスで、一般的の開発者が使う物ではありません。ですから、SDKには含まれません。ROM開発など、Androidのシステムをビルドする時には使えます。

こうして実装したサービスを以下のようにServiceManagerに登録してやると、クライアントから使う事が出来ます。

リスト 8.3: 実装したサービスを ServiceManager に登録

```
ServiceManager.addService("myservice", new MyService(context));
```

サービス側が以上の事をしていると、クライアントはこのサービスを使う事が出来ます。

サービスを使うには、ネイティブの場合と同様に/* 1 */ServiceManagerのgetService()を使ってIBinderを取得し、/* 2 */それをネイティブの場合のasInterfaceとほぼ同じ役割をする「サービス名.Stub.asInterface()」を使ってインターフェースとして呼び出します。

具体的には以下のようないい處になります。

リスト 8.4: Javaにおけるサービスの使用

```
// /* 1 */ getService で servicemanager からハンドルを取得
IBinder binder = ServiceManager.getService("myservice");

// /* 2 */ Stub.asInterface で BpBinder か BBinder かに応じて適切なクラスでラップして
IMyService にキャスト
IMyService myservice = IMyService.Stub.asInterface(binder);

// サービス呼び出し
int result = myservice.add(3, 4);
```

このように、aidl をコンパイルすると、binder を使用した通信回りのコードは全て自動生成されて、実際のインターフェースの実装をするだけで済みます。使う側も、取得の所を適当なファクトリーメソッドでラップてしまえば、中を何も理解せずに使う事が出来ます。

ですが、まったく自分で作っていない「インターフェース名.Stub.asInterface()」というのを呼び出さなくてはいけなくて、このメソッドは本章で解説したような背景知識が無いと何をしているのか分からないので、自分で実装してみるのは簡単だけど、自分が実装したコードが何をやっているかは分からない、という事になります。しかし本章をここまで読み進めた読者なら、実装を見なくてもこれが何をしているかは想像できる事でしょう。そして実際にその想像通りの事しかしていません。

以下では、軽く自動生成される Java のコードを見ていきましょう。基本的にはネイティブと同じなので自分で読んでも分かると思うので、不要と思う読者はスキップしていただいて構いません。

8.3 8.8.3 AIDL のコンパイルで生成されるクラス - Stub と Stub.Proxy と Stub.asInterface()

IMyService.aidl を aidl.exe や AndroidStudio でコンパイルすると、Java のソースが生成されます。以下に生成されるクラスの入れ子関係と asInterface() の宣言だけを並べてみましょう。

リスト 8.5: AIDL で生成されるクラスの入れ子関係と asInterface()

```
package com.example.myservice;

// IMyService は Java の interface となる
public interface IMyService extends IInterface {

    // BBinder 相当のサービスの実装の基底クラスを Stub インナークラスで定義。
    // interface では実装は含めないので IMyService が実装すべき物もここに押し込んである
    (例: asInterface())
    public static abstract class Stub extends Binder implements IMyService
    {
        public static IMyService asInterface(IBinder obj)...

        // Proxy クラスは Stub クラスの中にある。asInterface() からしか参照しないので、private
        で。
        private static class Proxy implements IMyService {
```

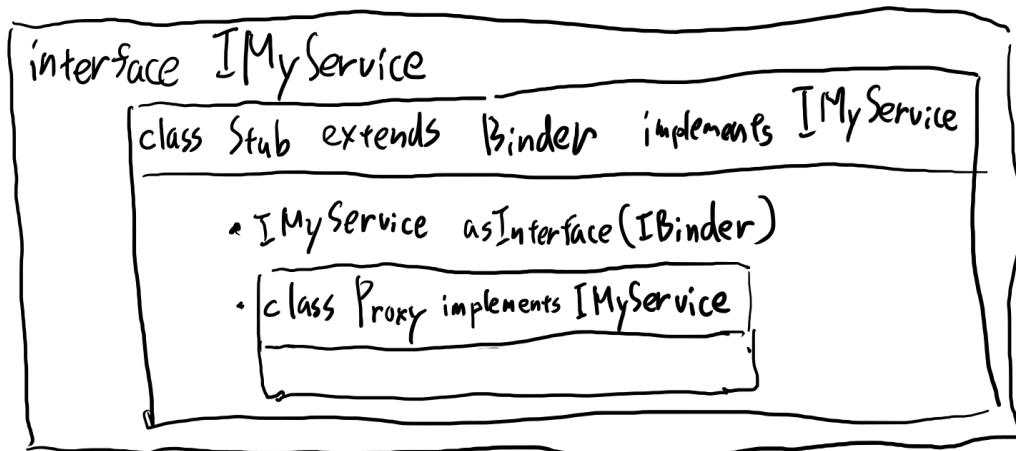
```

    ...
}
}
}

```

クラスとインターフェースを並べると以下の親子関係になります。

8.8.3 ①



`IMyService.aidl` から自動生成されるクラス群の包含関係、

図 8.1 IMyService、Stub、Proxy の包含関係

IMyService.Stub はネイティブで言う所の BnMyService に相当するコードを生成します。ただしサービスのメソッドの実装部分は abstract のままです。Android 以外の分散オブジェクトで Stub と言われる物と同様の物です。

IMyService.Stub はいわゆる TemplateMethod パターンで、add() がサブクラスで実装される、という前提で onTransact() の実装を生成します。この onTransact はやってきたデータから引数をデシリアライズしてこの add() を呼び出します。なおこの onTransact は、最終的にはネイティブの時と同様、IPCThreadState の joinThreadPool から BBinder の transact() を通して呼ばれます。

asInterface() の実装もこの Stub という内部クラスに入ります。本来的には Stub 側では無く IMyService に含めるべき物な気もしますが、Java は interface に実装が含まれないのでここに入るのでしょう。

IMyService.Stub.Proxy はネイティブの所で解説した BpMyService と全く同じ事をするプロキシのクラスです。BinderProxy 由来の IBinder をコンストラクタで受け取り、この BinderProxy に対してメソッド呼び出しに対応する引数とメソッド ID を用意して transact() を呼び出します。

他の分散オブジェクトシステムに詳しい方の為に補足しておくと、名前は IMyService.Stub.Proxy ですが、いわゆる分散オブジェクトの Stub では無くプロキシです。IMyService に asInterface() の

実装が置けないのでこうなっているに過ぎません。Proxy 自身はファクトリメソッドの中でだけ触れれば十分で、そこより上はインターフェースを返すので、asInterface() からだけアクセス出来る Stub 内部クラスの中で private 宣言されています。

IMyService.Stub.Proxy とか IMyService.Stub.asInterface() とか間に Stub が入るので難しく見えますが、ほとんどは単にインターフェースには実装がおけないという Java の制約を回避する為に Stub という内部クラスに関係無い実装まで置いているからに過ぎません。対応するネイティブの概念を本書で見ながら見ていくべき、見た目よりはずっと簡単な物である事が分かります。

以上で aidl をコンパイルした場合のケースも解説が終わりました。基本的にはサービスの実装者は IMyService.Stub を継承して実装し、呼ぶ側は IBinder を IMyService.Stub.asInterface() でインターフェースに変換して使えば良いだけで、解説する事もありません。

ネイティブの IIInterface 関連を使った実装と比べると、実装者としては onTransact の switch-case やプロキシクラスを書かなくても良い分楽になります。一方で実装の楽さに比べると登場する物の多さと言語境界をまたいだ分の複雑さから、なかなか Java のレイヤーで全容を理解するのが難しくなっています。

ですが、本章をここまで読み進める事が出来た読者の方は、端から端まで全てが理解出来るのではないかでしょうか。

8.4 8.8.4 実際の ACCOUNT_SERVICE 呼び出しにみる、システムサービス呼び出しの例

システムサービスの仕組みについてはここまで話で全て終わっています。

そこで終わっても良いのですが、せっかくなので、実際に普段皆さまがアプリを書く際に書いているコードを見て、それが実際どういう事だったのか、という事を、これまでの説明に照らし合わせて見ていきましょう。

例として、ACCOUNT_SERVICE からそのユーザーの google アカウントの一覧を取り出す例を考えます。それは、例えば Activity などで以下のように書きます。

リスト 8.6: AccountManager サービスを使う例 (*1)

```
AccountManager accountManager = (AccountManager)getSystemService(ACCOUNT_SERVICE);
Account[] accounts = accountManager.getAccounts();

for (Account account : accounts) {
    if (account.type.equals("com.google")) {
        ...
    }
}
```

このように AccountManager というオブジェクトを取り出して、まるで自身のプロセスのクラスであるかのように getAccounts() を呼び出す事が出来ます。

*1 一行目の getSystemService を直接呼ぶ方法は AccountManager に関しては現在では推奨されていません、実際には AccountManager.get(context) というメソッドがあってそちらを使う事になっています。ですがやっている事は getSystemService() を呼ぶ事と同じです。

これらのありふれたコードと、これまで説明したシステムサービスの仕組みはどう関係しているのでしょうか？

Activity の getSystemService() は、実際は ContextImpl で実装されています。この getSystemService() から呼び出されるコードは以下のようにになっています。

リスト 8.7: getSystemService() で最終的にサービスを取得している所

```
IBinder b = ServiceManager.getService(ACCOUNT_SERVICE);
IAccountManager service = IAccountManager.Stub.asInterface(b);
return new AccountManager(ctx, service);
```

AccountManager は IAccountManager のラッパとなっています。このように既知のサービスは ContextImpl 内で Stub.asInterface() がハードコードされているので、アプリ開発者はこの Binder という仕組みを理解していくなくても使う事が出来ます。

ServiceManager.getService() でサービスの binder_node の binder_ref を表すハンドルを取得し、IAccountManager.Stub.asInterface(b) で、このハンドルに対する IAccountManager のプロキシオブジェクトが生成されます。このプロキシオブジェクトのメソッドを呼び出すと、サービスの機能を使う事が出来ます。

ですがここでの実装では、直接この IAccountManager をユーザーに返さずに、このプロキシをさらに AccountManager というラッパクラスにくるんで、それを返しています。このクラスは大した事はしません。 RemoteException を RuntimeException にしたり、という程度です。

AccountManager の getAccounts() の呼び出しは、結局は IAccountManager の getAccounts(null) を呼び出します。これは ServiceManager から取得した binder_ref のハンドルに対して、getAccounts の method ID と引数をシリализして transact を呼び出します。すると、AccountManager のサービスの方の IPCThreadState::joinThreadPool の中から BBinder の onTransact 経由で AccountMnaager のサービスの onTransact が呼ばれて、適切な動作が行われて、reply としてアカウントの一覧を返します。

このように、Android は多くのシステムの機能をシステムサービスという形で実装していて、ユーザーはシステムサービスが独自のプロセスで実装されているのか、それともライブラリとしてアプリのプロセスで動くのかを気にせず使う事が出来ます。

サービスの仕組みは同一マシンを前提としていて使い方を決め打ちしている都合で、かなりライトウェイトでシンプルな仕組みとなっています。これがスマホなどのリソースの制約されたシステムにおいても、かなり低レベルな所まで分散オブジェクトをベースとしたシステムを、リーズナブルなパフォーマンスで構築する事を可能にしている鍵と言えます。

このおかげで、複数のサービスをプロセスのオーバーヘッドを避ける為一つのプロセスにまとめたり、端末のリソースが増えてきたら別のプロセスに移動したり、はたまたクライアント側のプロセス内でサービスで無くライブラリとして実装したり、と言った風な Android のバージョンに応じたシステムの発展を、アプリのコードを変更せずに実現しているのです。

第9章

8.9 様々なプロセスやシステムサービスの Binder 関連の初期化を理解する

前節までで、システムサービスの実装の仕方は全て説明しました。そしてなぜ Binder を用いた呼び出しが動くのかも、ネイティブはちゃんと説明してありますし、Java のプロセスでもだいたいは想像ができます。Binder の章としてはここで終わりにしても良いのですが、その為には他の各章との間をある程度想像力で補う必要がある事でしょう。そこで、本書ではそうした想像力に期待して終えるのではなく、ここで復習も兼ねて他の章との境目となりそうな話をしておきたいと思います。

境目として着目するのは、各システムサービスがどこのプロセスに所属しているか、というプロセス的な側面と、それぞれのプロセスの初期化の部分です。

システムサービスは、どのプロセスに属しているか、という事を意識せずに使えるシステムです。ですが、逆にそうであるからこそ、個々のシステムサービスがどこのプロセスに属しているか、という事に着目すると、Android というシステムが現在どの程度はハードウェアリソースがあると考えているか、という判断を理解する事が出来て、面白い視点を提供してくれます。

また、各サービスでなぜ Binder を使った呼び出しが動くのか、という事を確認する為に、Java のプロセスで ProcessState や IPCThreadState 関連のメソッドをどこで呼んでいるか、ネイティブのサービスでは実際はどうなっているのか、という事も見ておきます。

9.1 8.9.1 SystemServer とそれ以外のサービス - Zygote から起動されるサービスと init.rc から起動されるサービス

TODO: 参照更新 (0 章は一巻か二巻に編入されたはず)。この章は本編への参照が多いので、いちいち TODO は足さずにまとめて直す。

0 章で解説した通り、Android の init からは、app_process というプロセスが Zygote モードで立ち上がります。Zygote モードで立ち上がる app_process は Java 関連の初期化を行い必要なシステムのクラスライブラリをロードした後、「Zygote になる直前」に、SystemServer と呼ばれるプロセスを fork します。

8.9.1 ①

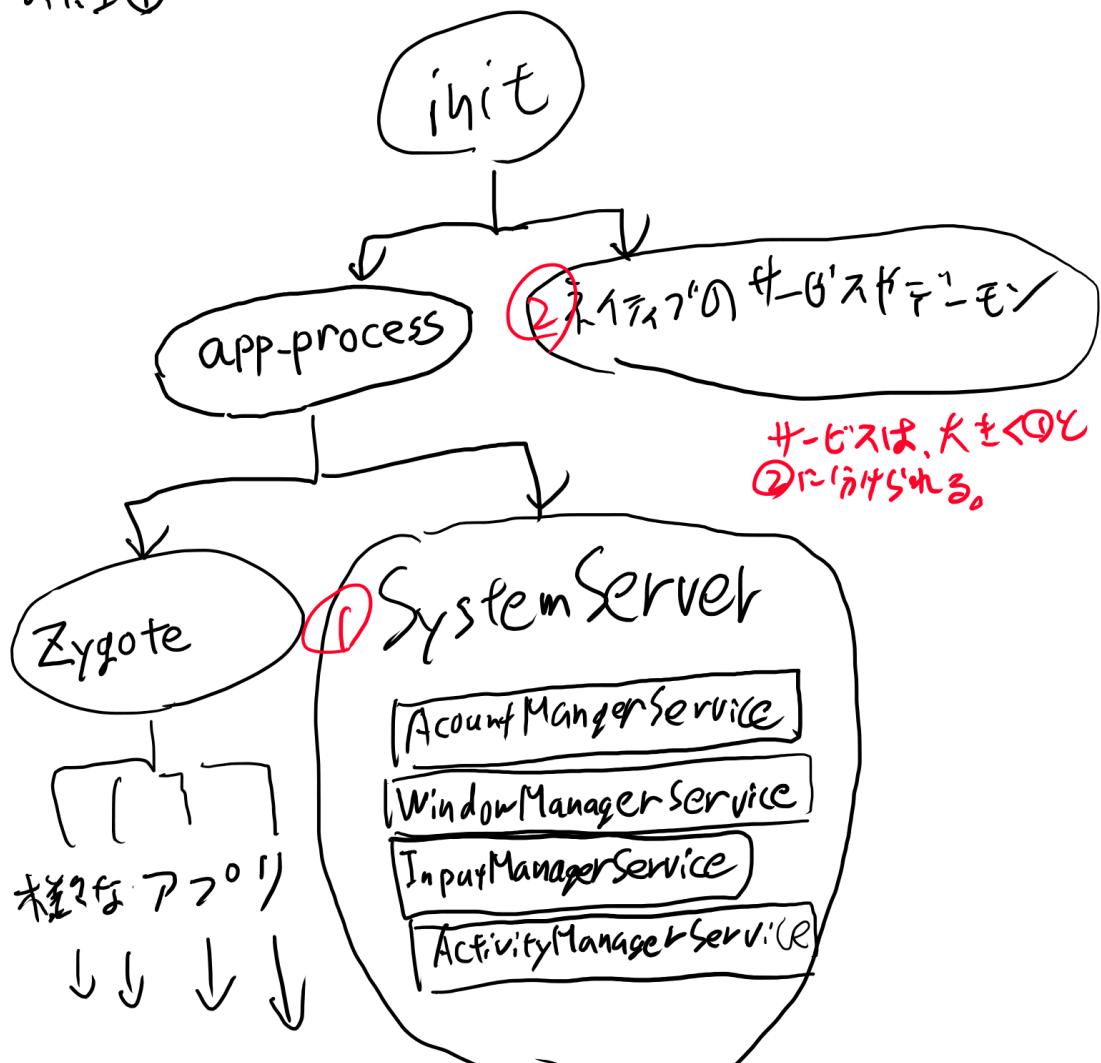


図 9.1 init から fork されるサービス達

SystemServer は Java で書かれたプログラムが動く app_process でありながら、Zygote になった後の app_process から fork する訳では無い、少しだけ特別なプロセスです。ps では system_server と見えて、Java のクラスとしては SystemServer クラスとなります。

様々なサービスがどのプロセスにホストしているか、という事を調べる場合、最初に知っておくべき基本となる事としては、ほとんど全てのシステムサービスは、この SystemServer プロセスにホストされている、という事実です。SystemServer に何がホストされているか、と考えるよりも、何が SystemServer 以外かを調べた方がずっと早いのが現状です。

大まかには、以下のように分かれています。

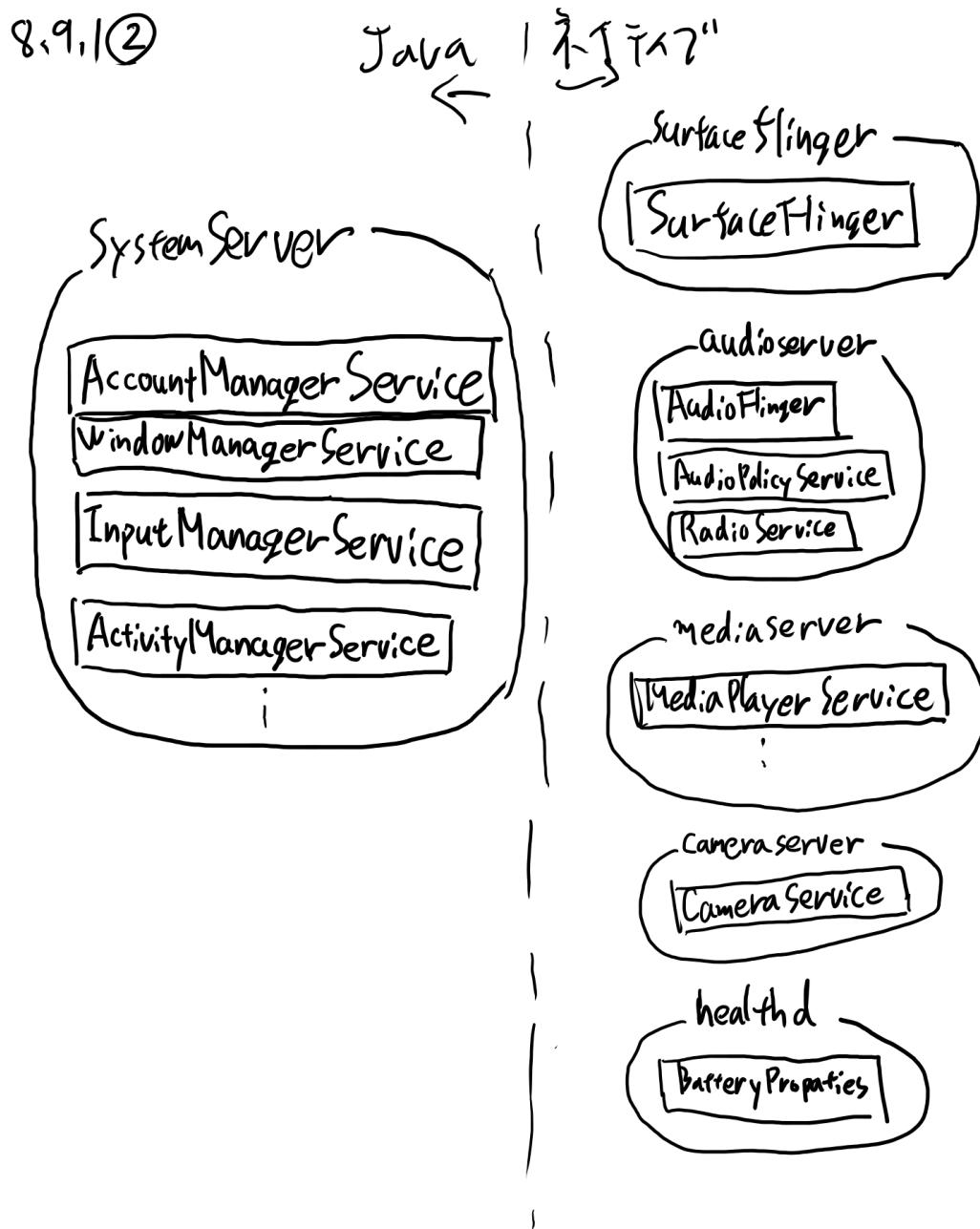


図 9.2 システムサービスがどこにホストされているか

では、まずは SystemServer 以外のプロセスにホストされているサービスから見てていきましょう。

SystemServer 以外のプロセスにホストされているサービス

Android 7.0 現在では、SystemServer 以外にホストされているサービスというと、例えば以下の
ような者たちになります。

1. surfaceflinger サービス
2. audioflinger サービス
3. camera サービス
4. mediaplayer サービス
5. batteryproperties サービス

surfaceflinger サービスは surfaceflinger という名前の単独のプロセスにホストされています。実行ファイルのパスは /system/bin/surfaceflinger にあります。

audioflinger と camera と mediaplayer などは、7.0 で大きく刷新された所です。本章を書き始めた時には、これらのサービスは全部 mediaserver という名前のプロセスにホストされていました。ところがセキュリティ的に大きな問題となった stagefright バグなどの影響で、現在はそれぞれ個別のプロセスに分けられて、audioserver, cameraserver, mediaserver という別個の実行ファイルから起動されます。MediaFramework の刷新と stagefright バグについては 12.6.2 のコラムも参照ください。

batteryproperties サービスは healthd というデーモンにホストされていて、実行ファイルは /sbin/healthd にあります。どれも init.rc 関連のファイルから起動されています。

全体的に、高い権限を必要とするカリアルタイム性を要求される物が多い傾向にあります。音声は途切れると目立ちますし、画面をスクロールした時の引っかかりなどは小さな事ですが大変目立つので、Android ではバージョンを重ねるごとに画面に割くハードウェアリソースを増しています。

SystemServer にホストされているサービス達

一方で SystemServer にホストされるサービスは多すぎて列挙しても見きれない程です。この本でも良く出てくる見慣れた物を幾つか挙げると以下のようになります。

1. AccountManagerService
2. WindowManagerService
3. InputManagerService
4. ActivityManagerService

その他、ほとんどのサービスがこの SystemServer にホストされています。

現在は別のプロセスに分けられたサービスも、かつてはこの SystemServer プロセスがホストしていました。Android はバージョンを重ねるごとに、標準的な端末の CPU のコア数やメモリ等が増えていくに応じて、SystemServer にいるサービスを、別のプロセスを作ってそこに移動していく、という事を繰り返しています。

例えば surfaceflinger は <ref>surfaceflinger サービスにみる、システムの発展</ref> で見たように、Android 2.3 の頃は SystemServer 内で行っていた処理が、Android の 3.0 の頃に現在のように別の surfaceflinger プロセスになったと思われます。

また、Android 7.0 で行われた MediaFramework の刷新で mediaserver プロセス一つにホストされていた様々なサービスが個別のプロセスに分けられたのも、システムの発展に伴いプロセスを分けていく例の一つと言えます。(詳細は 12.6.2 のコラムも参照ください) MediaFramework の刷新は、サービスという仕組みを根底に据える事でシステムの発展に伴いプロセスを分けていく事を可能に

している、という事が、まさに本書執筆時点でも有効に活用されている例と言えます。Binder を用いたシステムサービスの仕組みは、stagefright バグに代表されるより高度なセキュリティ対策の必要性を、初期の Android の頃から見越して作られている仕組み、と言えます。この辺の出来事と対応をリアルタイムで見ていると、システムの最初に織り込まれた哲学という物の凄みを感じます。

将来もコアの数などが増えるに従い、もっと多くのシステムサービスが、だんだんと別のプロセスに分かれしていく事でしょう。

9.2 8.9.2 surfaceflinger の main を見る - SystemServer 以外のシステムサービスの main 関数

まずは SystemServer 以外のサービスの、立ち上がる所から見てみましょう。一番簡単で本書での出番も多い surfaceflinger が説明にも都合が良いので、surfaceflinger の main 関数を見てみます。コメントだけ和訳して、まずは全コードを載せてみます。

リスト 9.1: SurfaceFlinger の main()

```
int main(int, char**) {
    // surfaceflinger のスレッドの最大数は 4 に制限しておく
    ProcessState::self()->setThreadPoolMaxThreadCount(4);

    // スレッドプール開始
    sp<ProcessState> ps(ProcessState::self());
    ps->startThreadPool();

    // surfaceflinger をインスタンシエート
    sp<SurfaceFlinger> flinger = new SurfaceFlinger();

#ifndef HAVE_PTHREADS
    setpriority(PRI_O_PROCESS, 0, PRIORITY_URGENT_DISPLAY);
#endif
    set_sched_policy(0, SP_FOREGROUND);

    // クライアントコードが接続してくる前の初期化
    flinger->init();

    // surface flinger を servicemanager に登録して公開
    sp<IServiceManager> sm(defaultServiceManager());
    sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, false);

    // このスレッドで run する。
    flinger->run();

    return 0;
}
```

いろいろとしますね。この章に関連する所だけ順番に見ていきましょう。全体的に、8.6.12 で説明したのと同じ構造となっているので、そちらの説明も参考にしてください。

まずは ProcessState の self() を呼んでますね。

リスト 9.2: ProcessState の self() 呼び出し

```
sp<ProcessState> ps(ProcessState::self());
```

この ProcessState::self() の呼び出しで Binder ドライバが open されて mmap されるのでした。次にスレッドプールを開始しています。(8.6.11 参照)

リスト 9.3: スレッドプールの開始

```
ps->startThreadPool();
```

この呼び出しでは、新しいスレッドが作られて、そこで IPCThreadState::self() の joinThreadPool() が呼ばれているのでした。この joinThreadPool() の中は ioctl を呼んでそれを処理するメッセージループとなっています。(8.6.4 参照)

次に SurfaceFlinger のインスタンスを new して初期化しています。

リスト 9.4: SurfaceFlinger をインスタンシエートして初期化

```
sp<SurfaceFlinger> flinger = new SurfaceFlinger();
flinger->init();
```

ここで SurfaceFlinger の型を見てみましょう。

リスト 9.5: SurfaceFlinger の型定義

```
class SurfaceFlinger : public BnSurfaceComposer,
    private IBinder::DeathRecipient,
    private HWComposer::EventHandler
```

いろいろな物を継承していますが、BnSurfaceComposer は名前から BnInterface を継承したクラスと予想出来ます。つまりこの transact が IPCThreadState の joinThreadPool から呼ばれて、その中から呼ばれる onTransact をオーバーライドして実装しているのでしょう。(8.7.7 参照)

一応 BnSurfaceComposer の宣言を見てみます。

リスト 9.6: BnSurfaceComposer の定義

```
class BnSurfaceComposer: public BnInterface<ISurfaceComposer> {
```

予想通り BnInterface を継承していて、ISurfaceComposer をテンプレートパラメータ経由で間接的に継承しています。

第 9 章 8.9 様式と並んで最初のプロセスの開始の Bindes 関連の初期化を理解する: onZygoteInit()

つまりサービスとしては BnSurfaceComposer が普通のサービス実装のようですね。SurfaceFlinger はこれを継承して、さらに機能を足している事が読み取れます。

main のコードに戻って、続きを読んでいきましょう。次は SurfaceFlinger を defaultServiceManager() の addService() を呼び出して、servicemanager に登録するコードです。

リスト 9.7: SurfaceFlinger を servicemanager に登録して公開

```
sp<IServiceManager> sm(defaultServiceManager());
sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, false);
```

これも 8.6.10 とほとんど同様で、唯一の違いは最後に false を渡している事です。このフラグは allowIsolated という変数名で、これは制限の強いサンドボックスからのアクセスは弾く、というフラグです。

こうして IServiceManager に登録すれば、このスレッドの役割は終わりなので、通常のサービスの main 関数のパターンでは IPCThreadState の joinThreadPool() を呼び出して、このスレッドもスレッドプールに加えてしまうのですが、surfaceflinger はちょっとここが違います。

この main 関数のスレッドでも、スレッドプールによる ioctl のループとは別のループを回すようです。

リスト 9.8: SurfaceFlinger のループ開始

```
// このスレッドで run する。
flinger->run();
```

この run() の中身の詳細はここでは説明しませんが、SurfaceFlinger は通常のサービスのイベントループとは別に、Looper を使ったループを持っていて、別のスレッドからメッセージベースでアクセスする事を可能にしています。

以上で surfaceflinger サービスの main 関数を見ました。大筋は 8.6.12 で解説したのと同じ処理をしている事が分かります。

9.3 8.9.3 Java のプロセスの開始と ProcessState - AppRuntime::onZygoteInit()

Java のプロセスの開始の詳細については 9.6 の Zygote の所で詳細に扱いますが、ここでは Binder 関連の初期化の部分だけ見ておきましょう。

Java のプロセスは、基本的には Zygote というシステムサービスが生成するか、または SystemServer プロセスかの、二種類しかありません。そして Zygote が生成する場合も SystemServer プロセスも、初期化時に AppRuntime の onZygoteInit() というメソッドを呼ぶ事は同様です。つまり、Java のプロセスならこの AppRuntime の onZygoteInit() がいつも呼ばれている、という事になります。

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？ 第 9 章 8.9 様々なプロセスや ApplicationThread にみる初期化を理解するクラスの呼び出し

その AppRuntime の onZygoteInit() メソッドは以下のようになっています。

リスト 9.9: AppRuntime の onZygoteInit() メソッド

```
virtual void onZygoteInit()
{
    ...
    // いつもの ProcessState のコンストラクタ呼び出し。binder ドライバが open されて mmap される
    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    // 新しいスレッドを作り、中で IPCThreadState::self()->joinThreadPool() が呼ばれる
    proc->startThreadPool();
}
```

ProcessEvent の self() と startThreadPool() が呼ばれています。これで、新しいスレッドが ioctl をメッセージ受信の為に呼び出して、その結果が BBinder による呼び出しだったらこのポインタの transact() を呼ぶ、という ioctl のループが始まります。

このように、Java のプロセスの場合、つまり SystemServer でも Zygote により生成されたプロセスでも、最終的にはネイティブのシステムサービスで出てきた ProcessState の self() で binder ドライバが初期化され、startThreadPool() で ioctl 呼び出しと、その結果が戻ってきたら BBinder の transact 呼び出しを行っていく訳です。こうして、Java のプロセスでも、外部からのサービス呼び出しが処理されます。

大分マニアックな話ではありますが、SystemServer のサービスの呼び出しのスレッドがどうなっているか、を正確に理解する事は、Android を一段深く理解する上ではなかなか重要な所です。9.6 と合わせてここを理解しておくと、OS のスレッドのレベルでサービス呼び出しが理解出来るようになります。

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？ - ApplicationThread にみるシステムサービスでないクラスの 呼び出し

Zygote のプロセスで ProcessState::self()->startThreadPool() 呼び出しが行われている、という話を見たので、これまで保留にしてきた、ApplicationThread 呼び出しの所を最後に見てみます。

6 章と 7 章において、ActivityManagerService は ActivityThread と協調して Activity を開始する様子を見ました。その時に、例えば 7.3 の attach() メソッドなどで、ActivityManagerService から、ActivityThread の scheduleXXX 群のメソッドを呼ぶ、という話をしました。これは Binder を用いて行われているのですが、その仕組みについて、最後に見ていきたいと思います。最後という事なので、復習も兼ねて細かい話を再度、通してみていきます。

この例をわざわざ見るのは、少し特殊な事があるからでもあります。その特殊な事とは、ActivityThread は別にサービスでは無い、という事です。実際、ActivityThread は servicemanager にも

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？

第9章 8.9 様々なプロセスやActivityThreadとBinderの関連の初期化を理解するクラスの呼び出し

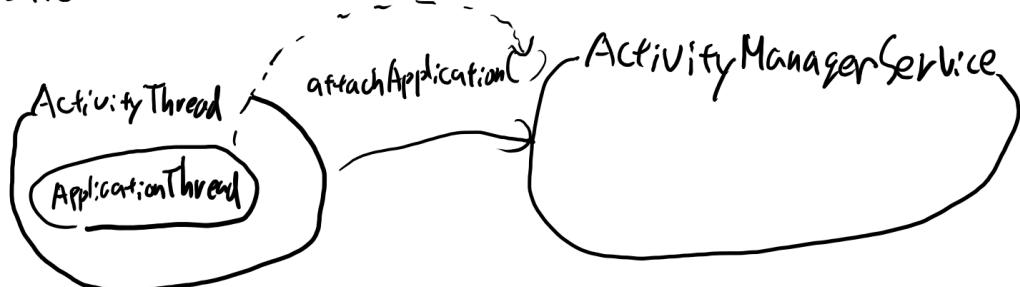
登録されていません。それが、一体どこで ioctl を待ち BBinder の transact() を呼んでいるのか、この BBinder は誰か、という事を確認していきます。

ActivityThread, ApplicationThread, ActivityManagerService の確認

7.2.3 で扱ったように、ActivityThread には ApplicationThread というインナークラスがあります。この ApplicationThread は、間にいろいろ挟まりますが基本的には Java の Binder クラスを継承しています。復習しておくと、Java の Binder クラスはネイティブの BBinder に対応しているクラスでした。

ActivityManagerService 自身は 8.9.1 で確認した通り、SystemServer にホストされているサービスです。ですから、通常のサービス呼び出しとして別のプロセスからメソッドを呼び出す事が出来ます。そこで ActivityThread は、attach() の呼び出しの所で、この ApplicationThread インスタンスを引数に ActivityManagerService の attachApplication() というメソッドを呼んでいます。

8.9.4①



ApplicationThread を引数に attachApplication() メソッドを呼び出している。

図 9.3 attachApplication() で ApplicationThread のインスタンスを渡す

ActivityManagerService はサービスの実装クラスであり、ActivityThread はそれとは別のプロセスからこのサービスを呼ぶクライアントなので、普通に考えるとこの呼び出しは ActivityManagerService のプロキシクラスを使って行います。実際コードを確認してもそうなります。

そのサービスプロキシの呼び出しの引数に Java の Binder を継承したクラスを渡している訳です。するとどうなるか？

ApplicationThread は、flat_binder_object で送られる

サービスプロキシのメソッド呼び出しは、最終的には BpBinder の transact を呼び出すのでした。この BpBinder の transact に、Java の Binder を継承したオブジェクトが、flat_binder_object に詰

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？

第 9 章 8.9 様々なプロセスや ApplicationThread が binder 連絡の初期化を理解するクラスの呼び出し

められて渡されます。^{*1} そして ActivityManagerService のあるプロセスにこの flat_binder_object が送られる。

8.9.4②

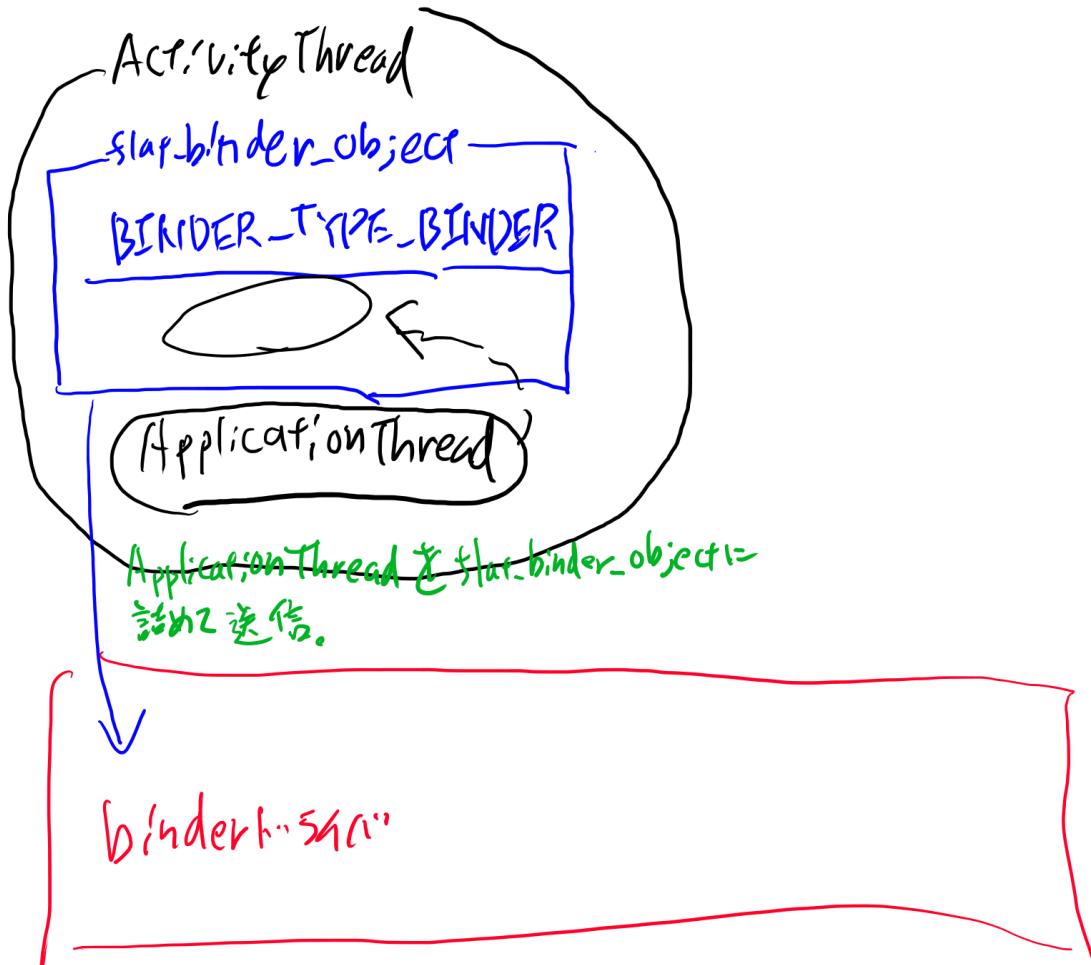


図 9.4 ApplicationThread を flat_binder_object に詰めて送信する

すると 8.5.4 で扱った flat_binder_object の変換のメカニズムで、ActivityThread のあるプロセスには binder_node というツリーのノードがドライバ内に生成されて、ActivityManagerService のあるスレッド側にはこの binder_node を参照する binder_ref のツリーにノードが追加され、このインデックスがハンドルとして ActivityManagerService の attachApplication の引数には入る訳です。このハンドルを ActivityRecord の持つプロセス構造体に紐づけるのです。

^{*1} 厳密に言うと、Java の Binder オブジェクトは、その Java のオブジェクトと一対一に対応したネイティブの BBinder のポインタを保持しているのですが、その BBinder のポインタが flat_binder_object に入ります。

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか?
第9章 8.9 様々なプロセスや ApplicationThread と Binder 連絡の初期化を理解するクラスの呼び出し

8.9.4③

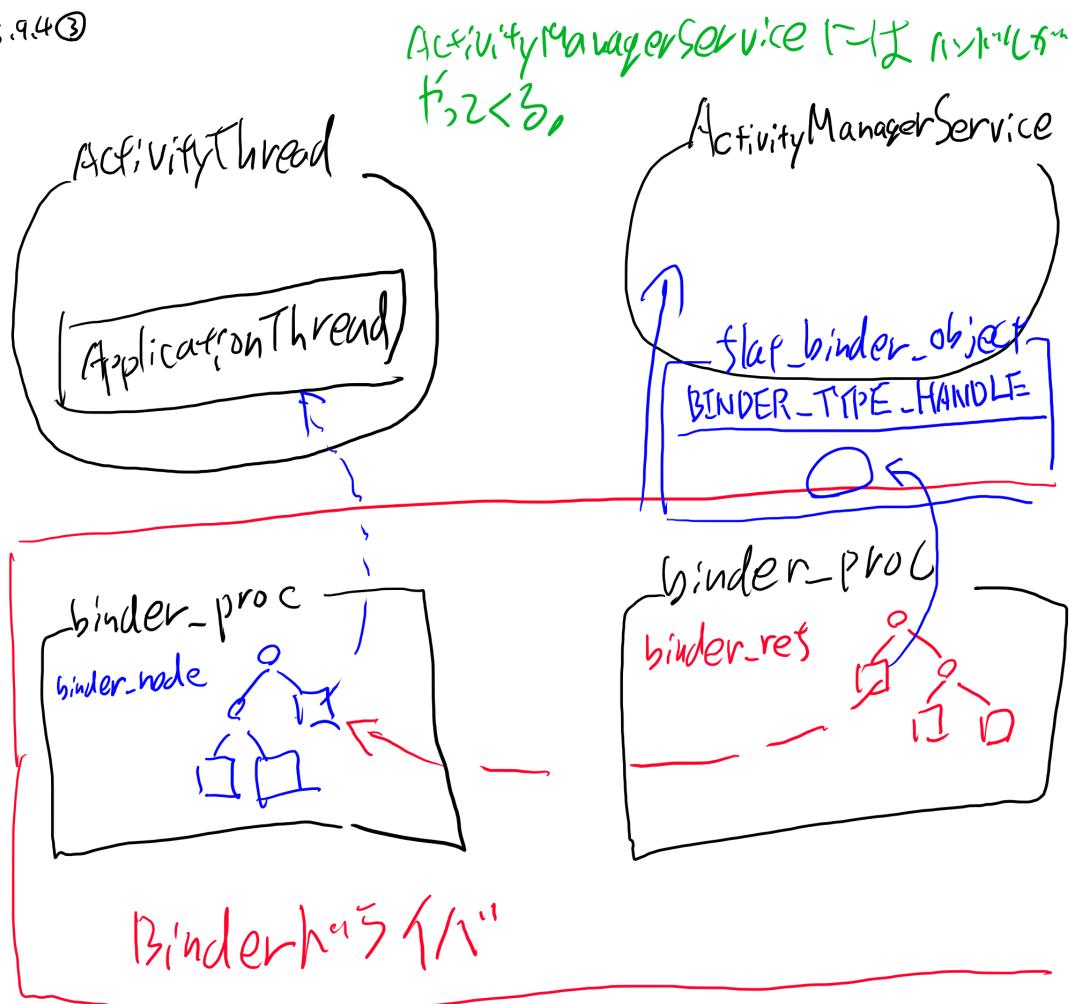


図 9.5 ActivityManagerService 側にはハンドルがやってくる

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか?
第 9 章 8.9 様々なプロセスや ApplicationThread が関わる初期化を理解するクラスの呼び出し

8.9.4④

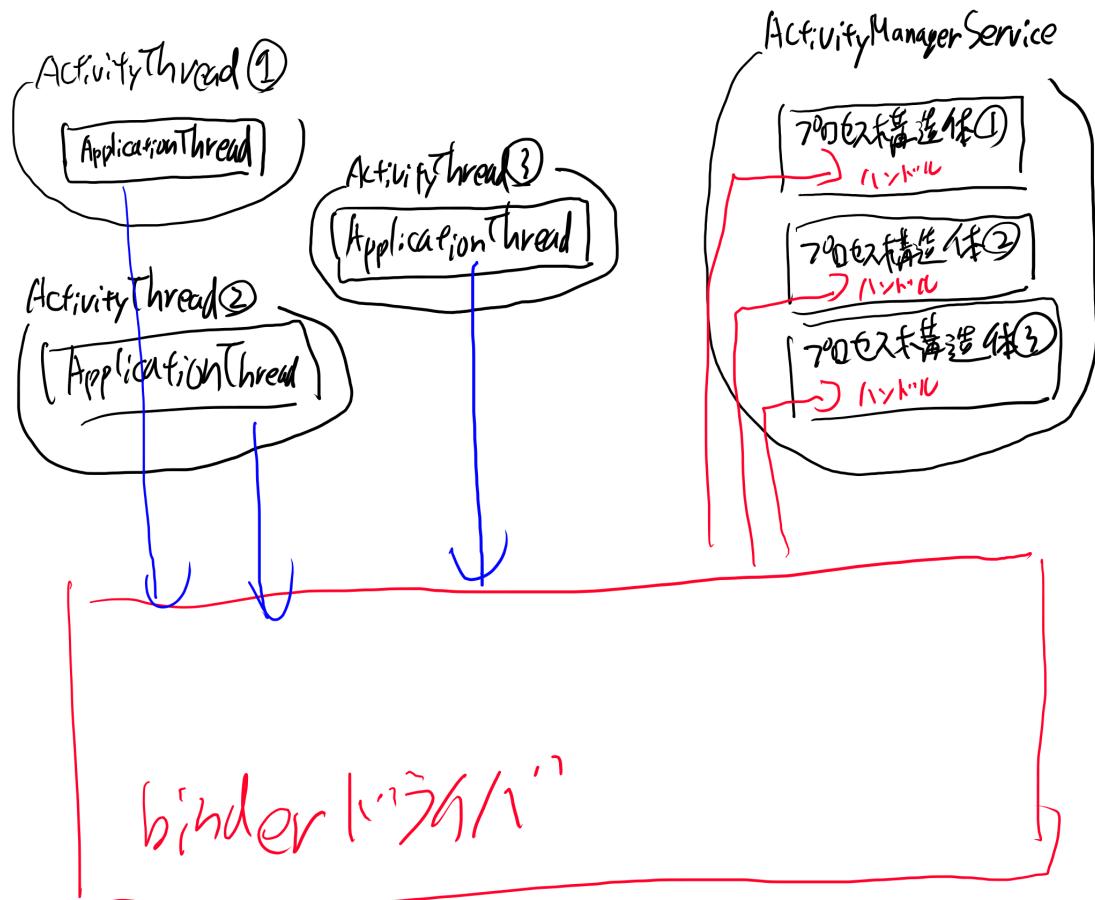


図 9.6 実際は複数のアプリがある

このように、ActivityThread は servicemanager に登録していないのですが、それでもシステムサービスと同様に、flat _ binder _ object にポインタをセットして渡せば、相手にはハンドルとして渡ります。

アプリのプロセスがメソッド呼び出しを受け付けられる理由 - onZygoteInit() × ソッドの処理

一見するとこれはどうという事のない話のように見えますが、これを実現させる為には、二つほど追加の条件が必要です。

1. ActivityThread 側も、binder ドライバを open して mmap してある
2. ActivityThread 側も、ioctl 呼び出しでブロックして待っている

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？

第9章 8.9 様々なプロセスや内部ActivityThreadとBpBinder間連の初期化を理解するクラスの呼び出し

8.5.4で解説した通り、binder_nodeはbinderドライバのプロセス構造体が保持しています。このプロセス構造体は、binderドライバファイルをopenした時のファイルディスクリプタに対応して作られる物でした。つまり、binder_nodeが生成される為には、ActivityThreadのプロセス、今まさにZygoteからforkして、アプリのapkからクラスをロードするそのプロセスも、binderドライバをどこかでopenしてmmapしてある必要があります。これが(1)に相当します。

また、メッセージを受け取るのにも準備が必要でした。

サービスのメソッドを呼び出す事、つまりActivityManagerServiceのメソッドを呼び出す側は、メッセージの送信だけで済みます。ハンドルに対するメッセージの送信は、特に難しい事はありません。ハンドルに対するメッセージ送信はBpBinderというラッパのクラスで行うでした。(JavaではBinderProxyクラス)中身としてもbinder_write_readに適切に初期化したデータを入れてioctlを呼べば良いでした。

ですが、メソッド呼び出しを受け付ける側、つまり呼び出される側となるには、もう少し難しい処理が必要です。メッセージを受け取る以上は、どこかで受信の為に誰かがioctlを呼び出して、ブロック状態になっていないといけません。そうでないと、メッセージを受け取るスレッドが居ない事になってしまいます。メッセージを受け付ける為にioctl呼び出しでブロック状態になるのは、IPCThreadStateのjoinThreadPoolの中で行われました。IPCThreadStateのjoinThreadPoolを、Zygoteでforkした後、誰かがどこかで呼び出しているといけません。これが(2)に相当します。

この(1)と(2)を行っているのが、前項の8.9.3で扱ったAppRuntimeのonZygoteInit()メソッドだった、という訳です。

AppRuntimeのonZygoteInit()メソッド呼び出しは、Zygoteのループで、ソケットからのコマンドを待って子プロセスをforkしている所で行っています。詳細は9.6.4で扱いますが、ここでは新しく生成されたプロセスの初期化の所で前項で説明したAppRuntime.onZygoteInit()が呼ばれる、という事が重要です。このAppRuntimeのonZygoteInit()メソッドからProcessState::self()->startThreadPool()が呼ばれて、あらなたスレッドが作られてそのスレッドがスレッドプールの一員として振る舞います。これらの初期化は、ActivityThreadのmainを呼び出す前に行われます。

つまり、アプリのプロセスは、システムサービスでは無い通常のアプリであるにも関わらず、必ずProsessState::self()->startThreadPool()が呼ばれているのです。言い換えると全てのアプリのプロセスでは、サービスを呼び出そうと呼び出さなかろうと、必ずioctlによるループを回しているスレッドが一つは存在しています。^{*2}

このように、Zygoteをforkしたプロセスは必ず最初にProcessState::self()を呼び出し、そこでbinderドライバファイルをopenしてmmapし、スレッドを一つ作り、その中でioctlのループを回しています。このスレッドで、ioctlからBC_TRANSACTIONが来たら、binder_nodeで管理されているポインタのtransact()を呼び出す訳です。このポインタはApplicationThreadと紐づいたネイティブのポインタという事になります。それは巡り巡ってApplicationThreadのscheduleLaunchActivity()などの、scheduleXXX系列のメソッドを呼び出す事になります。

ですから、Zygoteをforkしたプロセスでは、BBinderのサブクラスをflat_binder_objectに詰めて別のプロセスに送り付ければ、何もしなくても別のプロセスからのメソッド呼び出しを受け取

^{*2} 「サービスを呼び出そうと呼び出さなかろう」と言いましたが、アプリが起動するにはActivityManagerService呼び出しが必須なので、実際にはサービスを呼び出さないアプリのプロセスという物は存在しません。

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？

第9章 8.9 様々なプロセスやActivityThreadのある関連の初期化を理解するクラスの呼び出し

る事が出来る訳です。

メソッド呼び出しを受け付けるスレッドは GUI スレッドでは無い

少し細かい話となります、この BBinder が呼び出されるスレッドは、AppRuntime の onZygoteInit() メソッドで作られたスレッドプールのスレッドであって、いわゆる Looper.loop() などを実行している UI スレッドではありません。ですから、Binder 越しのメソッド呼び出しは GUI スレッドでは無いスレッドで実行される訳です。

そこで、GUI スレッドで処理を行わせる為に、7 章で説明した Handler の postMessage() の仕組みを使って、UI スレッドで Activity のコンストラクタや、ライフサイクルにかかる start() や stop() などのメソッドを呼び出していくのです。

TODO: 図解

このように ApplicationThread のポインタが binder ドライバによって管理されて、そのハンドルが ActivityManagerService にわたる。ActivityManagerService は渡されたハンドルに対してメッセージを送り、すると binder ドライバがこの flat_binder_object を BINDER_TYPE_BINDER のポインタ、つまり ApplicationThread のポインタに置き換えて、ActivityThread の存在しているプロセスの ProcessState の startThreadPool() で作られたスレッドが ioctl でブロックして待っているのを起こし、起こされたスレッドは BC_TRANSACTION なのを見て BBinder にキャストして transact() を呼ぶ訳です。

TODO: 図解

最後はずいぶんと長く複雑な話になりましたが、このように binder ドライバ、およびその上に構築されたシステムサービスという仕組みは、Android の根幹を支えている技術と言えます。多くのハードウェア的な機能やシステムの機能は、システムサービスという形で提供されていて、呼ぶ側はサービスがどこのプロセスに居るのか、という事を気にせずに呼ぶことが出来ます。そして同じプロセスの呼び出しは通常の C++ のメソッド呼び出しまで最適化される為、比較的低いコストで複数のサービスに分割してシステムを構築する事が出来ます。そして、ActivityManagerService と ActivityThread という Android の中でもとても重要な部分でも、この binder とその上のシステムサービスの仕組みはふんだんに使われています。

■コラム: サービスという言葉の定義のむずかしさ

2.3.4 のコラムでも書いた通り、Android には様々な所でサービスという言葉が使われます。そして、システムサービスと ActivityThread の管理するサービス、という二つの区別は、なかなか難しい物があります。本書では前者をシステムサービスと呼び、後者を SDK のサービス、と呼んでいますが、ここまで説明を理解すると、そこまで単純には分けられない事も分かってきます。たとえば、ApplicationThread はシステムサービスなのでしょうか？ servicemanager に登録されないので違う気がします。一方で Binder、BinderProxy、IInterface といった仕組みは全て使われていて、システムサービスとの違いが技術的に何なのか、というのは極めてあいまいです。また、ActivityThread に管理される SDK のサービスも、Binder を継承する事が出来て、binder ドライバを通してアプリと通信する事が出来ます。Activity から

9.4 8.9.4 ActivityManagerService はどうやって ActivityThread のメソッドを呼び出すのか？

第9章 8.9 様々なプロセスやActivityThreadとIBinderの関連の初期化を理解するクラスの呼び出し

bindService を呼ぶと、このサービスの IBinder が取得出来ます。このサービスが別のプロセスならば、この章で述べたのとほぼ同じ仕組みで使う事になります（ただし AIDL で作成した別のプロセスのサービスに bindService() を呼び出すのは、Android では推奨されていません <https://developer.android.com/guide/components/bound-services.html>）。システムサービスと SDK のサービスは、初めて見た時は良く区別が分からなくて、少し理解してみると全然違う物に見えて、さらに深く理解するとまた区別が分からなくなる、そんな関係にあります。最初の頃は区別して仕組みを学んでいき、一定以上理解が進んだら両者の区別はあまり気にしなくて良いんじゃないかな、と私は思っています。

第 10 章

8.10 この章のまとめ

本章では Binder について扱いました。8.2 から 8.5 までで、binder ドライバと flat_binder_object について扱いました。8.2 では bindar ドライバの概要、8.3 では open, mmap, ioctl と言ったシステムコールを、8.4 では ioctl によるメッセージの送受信の構造を扱い、8.5 ではその ioctl を用いて flat_binder_object を送信する時に何が起こるのかについて詳細に扱いました。

8.6 ではその binder ドライバを利用したスレッドプールのレイヤの実現について、スレッドプールの実装である IPCThreadState や ProcessState について説明し、そのスレッドプールが想定するサービス実装である BBinder 基底クラスとその使い方について扱いました。また、プロキシオブジェクトとは何か、という事やその実装に使う BpBinder についても説明しました。

8.7 ではそのスレッドプールのレイヤのクラス達を用いて、サービスが同じプロセスに居るか別のプロセスに居るかを吸収する為の共通インターフェースのレイヤとして、IInterface や interface_cast について扱いました。

8.8 では Java でシステムサービスを実装する為の、AIDL によるインターフェース記述とその生成されるクラスについて、ネイティブの共通インターフェースとの類似点を元に説明しました。

8.9 では以上の binder の仕組みが実際のサービスやアプリのプロセスでどのように使われているかについて、SurfaceFlinger の main を見たり ActivityThread と ActivityManagerService のやり取りを見直す事で調べました。

8 章はなかなか膨大な量になってしましましたが、Binder について必要な物は全て説明出来たと自負しています。Binder については公式のドキュメントがあまり無いので、全体像を解説した本章は、Binder を知りたい人にとってはなかなか貴重な文書だと思います。

別冊 詳解 Binder

2017年4月10日 全文章が入った最初のバージョン

著者 有野和真
