特集1

Androidによる組込み開発

Googleが発表したAndroidは、一般的に携帯電話向けのプラットフォームとして認知されています。組込み系のエンジニアにとっては、携帯電話以外への応用も早くから注目していますが、VMのパフォーマンスなど初期リリースにありがちな問題も指摘されています。 この特集では、Androidの組込み開発現場で役に立つ開発テクニックや高速化の手法など、実践的な情報を集めてみました。

第1章

Androidが変える組込みシステム

株式会社イーフロー
統括事業本部 第1事業部
金山 二郎
Kanayama Jiro

Androidの衝撃

2009年5月13日から3日間、東京ビックサイトにて、「第12回組込みシステム開発技術展（通称ESEC）」が催されました。昨年から始まった世界不況の影響で、最近の展示会は出展取り止めのために空地が多かったり来場者も少なかったりと、さびしい内容でした。しかし、ESECはこれまで同様に活気があり、組込み大国ニッポンの力強さを感じました。

その中でも、Androidは別格で、Androidの展示周辺には、不況をまったく感じさせない雰囲気がありました。弊社でも「高速化Android環境」と題して展示を行いましたが、Androidの展示エリアにはつねに人だかりができ、3日間で700枚用意したパンフレットが初日でなくなり、会場にてインクジェットプリンタで増刷するはめになりました（写真1）。

また、定員50名の無料セミナーを催したところ、セミナー会場がかなりわかりにくい場所であったにもかかわらず、ほぼ満席になりました（写真2）。1日1回、協力会社のブースの一角で開催したプライベートセミナーでも、毎回、立ち見が出るほどの賑わいを見せました。かつてこれだけ大きな話題を提供したプラットフォームがあったでしょうか。しかも、Androidの発表は2007年11月、すでに1年半を経過してこの盛況はただ事ではなく、あらためて、Androidに対する関心の高さをうかがわせました。

写真1　Androidの展示にはつねに人だかりができた

写真2　満席状態となったAndroidセミナー

Androidまでの道のり

さて、Androidはかつてないプラットフォームであるわけですが、その素晴らしさを知るためには、これまでのプラットフォームについて知ることが近道です。携帯電話に代表される高機能組込み機器向けプラットフォームの歴史を振り返ってみたいと思います。

本誌の読者に説明は不要かもしれませんが、OSすら搭載しない組込み機器を除けば、ほんの10年ちょっと前の組込み機器はいわゆるRTOSが搭載されており、普及のただ中にあった携帯電話なども例外ではありませんでした。OSやドライバはもちろん高額な製品を購入する必要があり、さらにそれらのインテグレーションに何億円もの投資を必要としました。しかし、契約者数が伸び続けたこのころが、携帯電話開発の一番華やかなりし時。まさに、高度成長期であったことも事実でした。そして、携帯電話はついにJavaを搭載するに至ります。

しかし、高機能化に突き進む一方、不具合による回収が相次ぐようになります。また時を同じくして、契約者数の頭打ちが囁かれ、セット機ベンダが揃って売り上げの下方修正を行いました。

このような背景から、安定性／高機能化／低コスト化への直接的な要求が高まり、これを満たすプラットフォームに対する期待が高まりました。その中で、Windows Mobile、Symbian OS、Linux、さらにBREWがそれぞれの特徴を生かした形で携帯電話に採用され、普及していきました。日本国内では、さらにFOMA開発環境であるMOAPやauの携帯電話向け開発プラットフォームであるKCP+などが登場し、プラットフォームという言葉自体も広く認知されるようになりました。

しかし、高機能プラットフォームは順風満帆というわけにはいきませんでした。Windows MobileはWindowsCEのころから粘り強く成長を続けていましたが、Win32APIをそのままサポートするという理想はあまりにも高く、Microsoftの力をもってしても製品を満足できるレベルに引き上げるのに苦慮していました。また、とくに3Gに移行して間もないころの日本では、Windows Mobileは携帯電話としてはまだまだ異端の扱いでした。

Symbian OS、Linuxも大きな開発を必要とするプラットフォームでしたが、当時は60%に迫る圧倒的なシェアを誇っていたNTTドコモの主導により、携帯電話向けプラットフォームとしての体をなすことができました。

BREWもまた特別なプラットフォームで、これはクアルコムがチップセットと一緒に販売したために普及しましたが、JavaなどのVM形式と違い、ネイティブベースのアプリケーションプラットフォームであったため、アプリケーションが環境に影響を及ぼさないかを検証する手間がかかるという問題を抱えています。

MOAPやKCP+は、特定携帯電話事業者向けプラットフォームとして成長し続けましたが、一携帯電話事業者がどこまでプラットフォームをけん引していけるのか、3Gが浸透するにつれ、ガラパゴスという言葉もやや否定的に用いられるようになり、その動向は注目されていました。

Androidの受難

Androidはそんな喧騒のさなかに登場しました。IT業界では、Microsoftに対抗できるということがひとつのステータスになったりしますが、当時、Googleは間違いなくその有力な一社でした。そのGoogleが旗振り役となってOHAを立ち上げ、世界中の主たる携帯電話関連企業30社以上と手を組んでプラットフォーム

を構築するというのですから、話題にならないはずがありません。有力紙もこぞってこのセンセーショナルなニュースを取り上げました。

テクニカルな面でも、Androidは実にセンセーショナルでした。Java技術に基づくアーキテクチャは、実は誰もが一度は夢見たアイデアでしたが、かつて、これほどの規模で実現に向けて動き出した会社はありませんでした。

そして発表から1年を経て、ついにAndroid搭載携帯電話『T-Mobile G1』が発売されました。独Deutsche Telekomが今年4月に行った決算発表によると、アメリカで発売されたこの端末は半年ほどで100万台を売り上げ、iPhone, BlackBerry Curve／Pearlに次ぐシェアを確保しました。

さて、確かに好調を維持しているAndroidですが、さすがに良いことばかりではありません。それも100年に一度の大不況がやってきて、Androidを直撃したのです。Androidのような大規模プラットフォームには大きな初期投資を必要とします。まだバグも多く、他の章で触れるような性能の問題の改善／解決が必須とされました。

それが、かつてない大規模な不況のために、どの企業も研究開発や商品開発などの投資を凍結し、経済の様子を窺う姿勢に移行しました。また、新聞やテレビでご存じの通り、大規模なリストラが敢行され、そこには景気が少しくらい悪くとも比較的元気であったIT関連企業も名を連ねていました。Android携帯電話の開発計画はほとんど公表されていませんでしたが、水面下で進んでいた計画の多くが、2008年末から2009年初頭にかけて中断の憂き目にあったことは間違いありません。

不幸中の幸いは、T-Mobile G1の出荷が始まっていたことかもしれません。これが間に合わなければ、Androidそのものがまだ世に出ていなかったかもしれないのです。

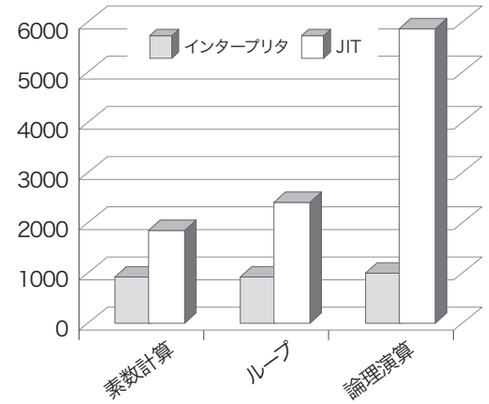


図1　Atom搭載MID端末におけるインタプリタとJITの実行速度比較。論理演算などのJITコンパイルは何倍もの性能改善を示す

Androidの飛翔1 ～高速化～

日系企業の多くが2009年3月末を期末と定めており、それはすなわち、少なくともそのころまで国内のAndroid開発が進まないことも意味しました。また、実質的な決算があきらかになる時期はもう少しあとになりますので、4月もAndroidに関する動きは鈍かったと言えます。

決算がおおむねあきらかになり、2009年度の計画もなんとなく決まってきたゴールデンウィーク前後から、Android周辺の市場も再び動き出し、冒頭のESECへとつながっていったのです。来場者は皆熱心で、突っ込んだ質問をする人も多く、そういったことからもAndroidの可能性の大きさを実感しました。

写真3　Atom搭載MID端末におけるJITコンパイル実行のデモ

展示内容としては高速化Android環境という展示を行いました。Androidの基幹コンポーネントであるDalvikVMには性能上の大きな問題が残っており、パフォーマンスを要求される部分はすべてネイティブ実装に置き換えられています。これがAndroid環境自身のポータビリティを下げ、Java言語で記述された秀逸なソフトウェア／サービスの速やかな導入を妨げる元凶となっています。この大問題に取り組むため、弊社ではDalvikVMにJITコンパイラをサポートし、Android環境自体の高速化の可能性を示しました（図1）。

Androidの飛翔2 ～デバイス対応～

また、携帯電話以外のデバイスへの適用は、もうひとつの大きな課題です。期待と言いかえてもいいかもしれません。PNDやいわゆるモバイル情報端末はその典型です（図2）。JavaはWrite Once Run Anywhereというスローガンを掲げましたが、Androidも、そのプラットフォーム自身としてはRun Anywhereを自負していると言えるでしょう。前述のESECにおいて、多様なデバイスへの応用が目につきました（写真5）。

写真4　ご存知Armadillo 500FX。ARMのJITコンパイル実行のデモを行った

AndroidはLinuxをデフォルトのOSとしており、とりあえずLinuxが動作するデバイスへの展開の可能性を秘めています。CPU対応としては、携帯電話用ということでまずARMがサポートされていますが、ARMと今後の組込み機器向けCPU勢力を2分するといわれるAtomベースのデバイスにおいても、実用化ベースのAndroidの対応が行われています。さらに、MIPSテクノロジー社は自らMIPS向けのAndroidの対応を発表し、国産CPUとしてはSH向けのポーティングも始まっています。

製品化ということについて言えば、Androidはいまだ難産のさなかにあり、現時点ではT-Mobile G1とその後続機種が発売されているのみです。一般の目から見れば、1機種にしか映らないのではないかと思います。しかし、おもに中国本土において、2009年末には2桁の製品が登場すると言われており、実際に、大小

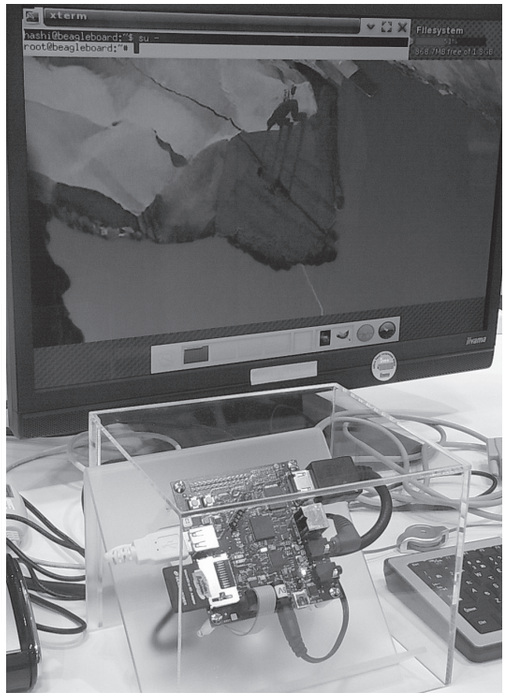


写真5　Androidの動画コーデックを商品のコーデックに置き換えて高速化したデモ

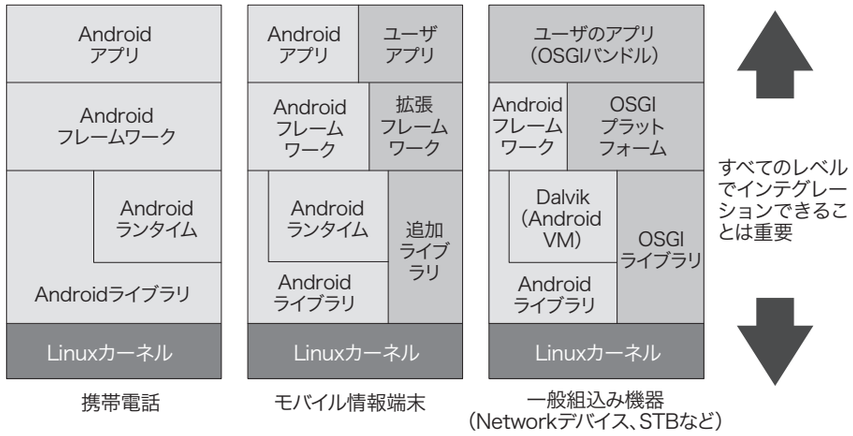


図2
Androidの適用可能性を示した図。特にESECのような会場では、Androidの携帯電話以外のデバイスへの適用に関する突っ込んだ質問が相次いだ

さまざまな事業者／ベンダの動きが活発になってきています。今年後半のAndroidの動向は、まさに目が離せない状況となるでしょう。

AndroidとJavaの関係性

最後にもう一点、JavaとAndroidの競合関係について述べたいと思います。2つのプラットフォームの争いは、クラウドの覇権争いにもつながるものと言えます。裁判準備のために、Androidのソースコードには絶対にアクセスしないようにと、Sun社内でアナウンスがなされたという噂もまことしやかに囁かれました。元はSunのCTOまで勤め上げたEric SchmidtがSunに叛旗をひるがえした、SunよりJavaを知っている彼には勝てないといった言い方もされました。

そのEricが次世代のコンピューティング形態として掲げたのがクラウドコンピューティングです。クラウドについてはいろいろな見方がなされていますが、

現代のテクノロジーに基づくクライアント・サーバの発展形

という言い方ができると思います。あるいは、より端的に、

NCの現実解

とも言ってしまっても面白いでしょう（NCが目指したネットワークセントリックな世の中になりつつあるということです。NC：Network Computerとは、1990年代半ばにOracleが中心となって策定した、現代で言うシンクライアント端末のことですが、普及はしませんでした）。

そして、Ericがクラウドコンピューティングという言葉をまだ表明していなかったころ、次世代のクライアントプラットフォームの最右翼と目されたのがJavaでした。そして、それを真向うから覆したのが、Java技術をふんだんに盛り込んだAndroidだったのです。

次世代のITビジネスはクラウド・コンピューティングの覇権争いになるという見解はもはやあらためて申し上げるまでもないことです。Googleがその大事業のためにAndroidをプロデュースしたことは明白な事実です。

では、Javaはどうなったのでしょうか？これも、本誌の読者ならずとも周知の事実ですが、OracleがSunを買収したことにより、JavaはOracleの技術となりました。すると、クラウドコンピューティングの支配権を賭けたGoogleとOracleの大決戦という図式に、AndroidとJavaは見事に当てはまるのです（IBMが、数億ドルの譲歩ができずJavaの権利を逃したらしいというショートストーリーも残されました）。

これらを骨肉の争いと称する人も少なからずいます。Android、そしてその実質的基盤であり、Androidの当面最大の敵であるJavaの両方はもともと同根であり、まさに骨肉、その争いは兄弟喧嘩の様相を呈しています。ではこの戦いは、どのような展開を見せるのでしょうか。

この結論にはやや勇気を要しますが、誤解を恐れず、家電業界の事例にあてはめると、

HD DVD vs Blu-Ray

の争いに通じるものがあります。あるいは、

ベータマックス vs VHS

と聞いて、なるほどと膝を打った方は、いわゆるアラフォー以上の世代になりますでしょうか。

この結論は多分に私見を含みますので、その点をご了承いただきたいのですが、筆者としては、GoogleとOracle、また他のクラウド事業者が、ベータマックスとVHSのように長きに渡った争いを展開してほしいと考えています。そのような争いは早く決着した方が消費者のためだという声も確かにあります。

しかし、HD DVDとBlu-Rayの戦いがあまりにも早く決着してしまったため、「過当競争のタイミングも早まり、両者ともビジネス的に意味のある規格ではなくなってしまった」と漏らす関係者が少なくないこともまた事実なのです。どんな技術も、一時の盛り上がりを見せたあとは一般化し、成長が鈍る局面を迎えます。究極的には“空気のように浸透した”という成果だけが残り、それ以上の発展はありません。一方、競い合っている間は、相手をやりこめ、自分たちが躍進するために懸命になり、それが技術を活性化させる触媒となります。

ポリティカルな理由で握りつぶしてしまうには、JavaもAndroidも、どちらも惜しい存在です。両者とも、もうひと働きもふた働きもしてもらわなければと強く願います。そして、Android特集ということで申し上げますと、今JavaとAndroidを同じ土俵に並べるためには、速度性能を第一とするAndroidの品質の向上が急務でしょう。その詳細は他の章に譲り、Android端末およびJava端末、そして後続のさまざまな端末の群れがクラウド世界を気持ちよく駆け回る姿を夢見て、本章を結びたいと思います。

column ■Javaは遅いか、Androidは速いか

本章全体に渡って、Androidの遅さをその短所として挙げていますが、速度についてはJavaとAndroidは見事に明暗が分かれました。ご存じの通り、Javaは「動くホームページ」という、今にして思えば実に時代がかったうたい文句で登場しました。そして、当時のMicrosoftと戦う勇敢な企業の一社であったSunの切り札として、Microsoft製品を駆逐するための対抗馬としての役割を負ったのです。

しかし、そのような場合によくあるパターンとして、機が熟しておりませんでした。当時のJavaはJITコンパイラなどの技術を搭載しない純粋なインタープリタでしたが、にもかかわらずデスクトップPCでユーザインターフェースの記述に使われ、また、組込み向けということで携帯電話に採用されていきました。そこで、その遅さがあきらかになり、Javaすなわち遅いというイメージを定着させてしまいました。その印象は今もって残り、Javaが世に出てから10有余年を経た今でも、Java製品の売り込みの際に「でもJavaって遅いんですよね」という、懸命な弁解を要する素朴な疑問を投げかけられることは珍しくないのです。

Androidはどうでしょうか。実際には、現代のハードウェアのパフォーマンスを10%も引き出せないようなレベルのプラットフォームであるにもかかわらず、ごく好意的に迎えられています。これは、現代の組込み機器の性能が十分に上がったことと、T-Mobile G1向けの開発の段階で、十分にネイティブ化がなされた結果によります。そのおかげで、Androidはすぐにも搭載～製品化が可能という誤解を与えています。

JavaとAndroidの対決に際し、突貫で製品化にこぎつけたAndroidは、第一印象こそ良かったものの、これから試練を迎えることになります。

第2章

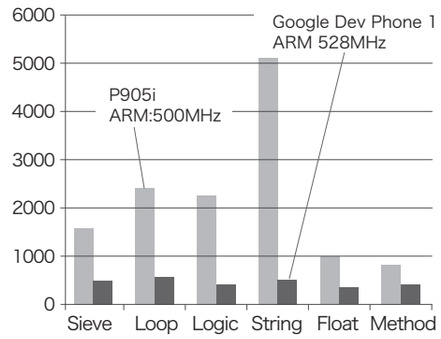
株式会社イーフロー
事業統括本部 第1事業部
中川 輪土
Nakagawa Rindo

Android高速化テクニック

Androidの現状

Android上で動くAndroidアプリケーションはJava言語を使って作成することができます。高機能なAndroid Frameworkがすでに用意されているため高い生産性を持っています。一方、Androidは同様のハードウェア構成であっても携帯Javaに比べて処理が遅いのが現状です。図1は携帯JavaとAndroidの性能を組込みJavaベンチマークEmbedded CaffeineMarkで測定した結果です。スコアの高さは性能の高さに比例しています。Androidは携帯Javaに比べて約2〜10倍性能が低いことがわかります。携帯Javaで問題なく動作しているアプリケーションを、Androidに移植した際にこの処置の遅さが一番問題となります。

図1 Embedded CaffeineMark 結果



Androidが遅い理由

Androidアプリケーションは Java言語を使って作成されます。Javaコードは図2のようにJavaのバイトコードである.classファイルに変換され、さらにAndroid独自のDEXバイトコードに変換されます。

変換されたDEXバイトコードは、図3にあるDalvikVMと呼ばれる仮想マシン上で実行されます。DEXバイトコードはバイトコードであるためCPUに依存しません。

DalvikVMはバイトコードインタープリタ方式でAndroidアプリケーションを実行します。インタープリタ方式で実行されるためC言語などで作成されたネイティブアプリケーションと比べて格段に遅くなります。Javaの場合も同じバイトコード形式のclassファイルをJava仮想マシン上で実行しますが、現在のJava仮想マシンはさまざまな高速化手法が適用されており、ネイティブアプリケーションとさほど変わらない速度で実行することができます。Androidは1990年代後半のJavaに似ています。AndroidとJavaの間には10年分の差があることになります。本章ではこの10年間で培ったJava高速化手法を元にしたAndroidの高速化テクニックについて解説します。

図2 Javaコードの変換

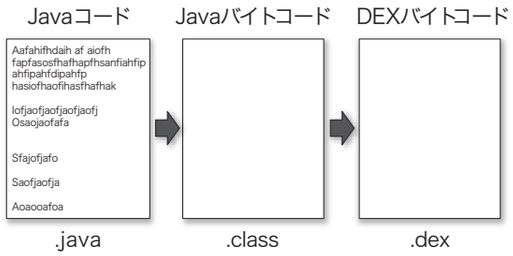
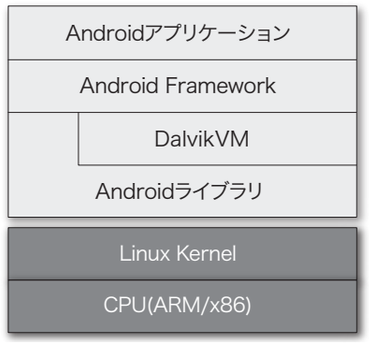


図3 Android構成図



Androidアプリケーション高速化テクニック

まずはAndroidアプリケーション高速化のテクニックについて解説します。Androidアプリケーションは Java言語で作成するため、Javaアプリケーションの高速化テクニックがそのまま使えます。

●インスタンス生成の抑制

インスタンスは以下の手順を経て生成されます。

1.クラスのロード、初期化
2.インスタンスのヒープへの確保
3.インスタンスの初期化

インスタンスを生成するコードを書くと、上記1~3が毎回実行されることになります（ただし、1については初回時のみ）。とくに2では、ヒープに空きがない場合、不要になったオブジェクトを回収してヒープの空きを作るためにGarbage Collection（以下、GC）が発生します。GC実行中はすべてのスレッドが停止してしまうため、結果的に実行速度が低下します。生成したインスタンスを破棄せず、使いまわすことでインスタンスの生成を抑えることができます。この手法はオブジェクトプールと呼ばれます。生成したオブジェクトをプーリングし、必要な時にプール（pool）からオブジェクトを取り出し、不要になったオブジェクトはプールに戻してインスタンスを初期状態に戻します。とくにスレッドの場合はスレッドプールと呼ばれ、Webアプリケーションの分野で活用されています。

●ローカル変数へのキャッシュ

リスト1のようにforループの終了条件でメソッドを呼び出すと、終了条件判定ごとにメソッドが呼ばれます。この場合、リスト2のようにgetSizeメソッドの結果をローカル変数に入れることで、毎回getSizeメソッドを呼び出す必要はなくなります。

リスト1

```
for（int i=0; i < inst.getSize（）; i++）
 inst.getIndex（i）;
```

リスト2

```
int size = inst.getSize（）;
for（int i=0; i <size; i++）
 inst.getIndex（i）;
```

●インスタンス変数のキャッシュ

リスト3のcall01メソッドのようにインスタンス変数を頻繁に使用すると、毎回インスタンス取得、値の格納命令がDalvikVM上で実行されます。この場合リスト4のcall02ようにローカル変数にインスタンス変数をキャッシュして、結果をインスタンスに格納することでインスタンス変数の取得、値の格納を減らすことができます。

リスト3

```
private int n = 0;
public void call01 () {
n = 1 + n;
  n = 2 + n;
  n = 3 + n;
  n = 4 + n;
  n = 5 + n;
}
```

リスト4

```
private int n = 0;
public void call02 () {
int tmp = n;
tmp = 1 + tmp;
tmp = 2 + tmp;
tmp = 3 + tmp;
tmp = 4 + tmp;
tmp = 5 + tmp;
n = tmp;
 }
```

リスト5がcall1メソッド、リスト6がcall2メソッドのバイトコードをAndroid SDKのdexdumpツールで逆コンパイルしたものです。iget命令がインスタンス変数取得、iput命令がインスタンス変数格納の命令です。call01メソッドで頻繁にiget/iput命令が実行されています。iput/iget命令ではインスタンスがnullでないかのチェックなどが毎回実行されます。命令としては実行負荷が高いため、この処理を省くことでアプリケーションの高速化が可能になります。

リスト5

```
XXXX.call01: () V
iget v0, v1, LXXXX;.n:I
add-int/lit8 v0, v0, #int 1
iput v0, v1, LXXXX;.n:I
iget v0, v1, LXXXX;.n:I
add-int/lit8 v0, v0, #int 2
iput v0, v1, LXXXX;.n:I
iget v0, v1, LXXXX;.n:I
add-int/lit8 v0, v0, #int 3
iput v0, v1, LXXXX;.n:I
iget v0, v1, LXXXX;.n:I
add-int/lit8 v0, v0, #int 4
iput v0, v1, LXXXX;.n:I
iget v0, v1, LXXXX;.n:I
add-int/lit8 v0, v0, #int 5
iput v0, v1, LXXXX;.n:I
return-void
```

リスト6

```
XXXX.call02: () V
iget v0, v1, LXXXX;.n:I
add-int/lit8 v0, v0, #int 1 // #01
add-int/lit8 v0, v0, #int 2 // #02
add-int/lit8 v0, v0, #int 3 // #03
add-int/lit8 v0, v0, #int 4 // #04
add-int/lit8 v0, v0, #int 5 // #05
iput v0, v1, LXXXX;.n:I
return-void
```

●スタティックメソッドの活用

インスタンスメソッドを呼び出すためには、インスタンスを生成する必要があります。スタティックメソッドの場合、インスタンス生成のコストがかかりません。また、インスタンスメソッドを呼び出す場合、DalvikVMはinvoke-virtual命令を実行してメソッド呼び出しを行います。スタティックメソッドの場合は、invoke-static命令を実行します。invoke-static命令の方がinvoke-virtual命令よりも実行負荷が小さい実装のため、スタティックメソッドの呼び出しの方が速くなります。

●浮動小数点演算は極力使用しない

浮動小数点演算は整数演算に比べて演算速度が遅いため、浮動小数点演算はなるべく使用しないようにします。

●数はstatic finalで定義する

定数をstatic finalでフィールドに定義すると、C言語のマクロ定義のようにフィールドを使用している部分がDalvikVM上ですべて定数として扱われます。static finalを付けずにフィールドを定義した場合、スタティックフィールドのアクセスの度にスタティックフィールドの取得が必要になるため、その分速度が遅くなります。

●ポリモーフィズムの使用を控える

ポリモーフィズムを使用することで抽象化を行い、機能と実装を切り離すことができますが、抽象化される分だけ実装のコード呼び出しの際オーバーヘッドが発生します。このオーバーヘッドの発生を減らすため、できるだけポリモーフィズムの使用は控えるようにします。

●オブジェクト指向プログラミングを止める

オブジェクト指向プログラミングは、生産性が高い半面、その分処理が全体的に重くなります。オブジェクト指向プログラミングを止め、インスタンスのアクセスを減らし、メソッドをインライン展開します。生産性は落ちますが、その分余計なオーバーヘッドが減るため速度は向上します。

●Androidアプリケーション描画の高速化

Androidアプリケーションで描画を行うためには、まずアクティビティ(Activity)を生成して描画する領域を設定します。描画する領域にはCanvasとSurfaceViewの2種類があります。Canvasの描画はアプリケーション側のスレッドで行う必要があります。描画に時間がかかるとその分だけアプリケーションの処理が止まることになります。また、Canvasはソフトウェア上で描画処理を行うため描画速度も遅くなります。

一方SurfaceViewの場合は、ハードウェアアクセラレータを使った描画を指定できます。Androidアプリケーションは描画スレッドを意識する必要がありません。2D描画の場合はフレームバッファに描画されますが、3D描画の場合はOpenGL ESが使用されるのでハードウェアアクセラレータを使うことで描画を高速化できます。ゲームなど描画時に大きな負荷がかかるAndroidアプリケーションを作成する場合には、SurfaceViewの使用が望ましいでしょう。

●Android Native Development Kit (NDK)の活用

Android1.5からJNI (Java Native Interface)が公式でサポートされました。そのためのツールがAndroid Native Development Kit (以下、NDK)です。JNIはJavaからCやC++で書いたネイティブコードを呼び出すためのインターフェースです。JNIを使うとJavaのコード実行処理に時間がかかっている部分をC/C++言語のネイティブコードに書き換えることが可能になります。ネイティブコードで実行されることになるので処理速度は向上しますが、Javaコードに比べてポータビリティが悪くなります。また、JNI呼び出しのオーバーヘッドも通常のJavaメソッドの呼び出しに比べて大きいのが難点です。

JNIを使用するメリット、デメリットに注意が必要ですが、Androidの一番のボトルネックであるインタープリタ実行がなくなることで、高速化の効果はかなり大きくなります。NDKの詳細については別項を参照願います。

●Designing for Performance

最後に、Android開発ガイドのDesigning for PerformanceにもAndroidアプリケーションの高速化の話が書かれていますので紹介します。

DalvikVM高速化テクニック

Androidはオープンソースで、一部のコードを除いてほとんどが公開されています。DalvikVMのソースコードはApache License V2.0で公開されています。DalvikVMはおもにC言語で書かれており、インタープリタの部分はARM v4／v5、x86の場合はアセンブラ言語で書かれています。DalvikVMの実装を変更することで高速化する手法について解説します。Androidシステム開発者向けの解説になります。

JavaコードのJNI化

Androidフレームワークのクラスと、システムクラス(java.lang等)はJavaコードで実装されています。

このため、実行時にはインタープリタで実行されることになります。システムクラスのメソッドをJNI化することで「Android Native Development Kit（NDK）の活用」と同じ効果を得ることができます。たとえば頻繁に使われるクラスのひとつにjava.lang.Stringが挙げられますが、これらのクラスのメソッドをJNI化するだけでシステム全体の性能が上がる場合もあります。Stringクラスのメソッドはシステム全体に影響するので、時にはJNIの中でアセンブリ言語を使用してさらなる高速化を行う場合もあります。

Just-in-time（JIT）コンパイラ

Just-in-time（以下、JIT）コンパイラは、Androidアプリケーション実行中にDEXバイトコードをネイティブコードに変換します。AndroidアプリケーションのDEXバイトコードをJITコンパイラでネイティブコードにコンパイルすることにより、AndroidアプリケーションをDalvikVM上でインタープリタとして実行するのではなく、ネイティブコードとして実行できます。DEXバイトコードのフォーマットについては、Androidのソースコードのdalvik／docs以下に仕様があります。DEXバイトコードは16ビットのコードユニットで区切られており、命令によってコードユニットの数が定義されています。

たとえばint型の加算命令add-int命令の場合は、リスト7のように2コードユニットの4バイトで定義されています。add-int命令はレジスタBとレジスタCに加算する2つのint型整数を加算し、結果をレジスタAに入れる命令です。このレジスタはCPUのレジスタではなく、DalvikVMの仮想マシンのレジスタを指しているので、具体的にはメモリのどこかがレジスタとして扱われているだけになります。

リスト7

```
add-int vAA, vBB, vCC
opcode (8bit)
A: destination register or pair (8 bits)
B: first source register or pair (8 bits)
C: second source register or pair (8 bits)
```

JITコンパイラは、図4のようにDEXバイトコードとDalvikVMのレジスタのアドレスを元に、ネイティブコードを生成します。生成するネイティブコードは、add-int命令の場合以下になります。

1. DalvikVMのBレジスタ（実体はメモリ）の内容をCPUレジスタに格納する。
2. DalvikVMのCレジスタ（実体はメモリ）の内容をCPUレジスタに格納する。
3. 1、2のレジスタの内容を加算する。
4. 加算した結果をDalvikVMのAレジスタ（実体はメモリ）に格納する。

1〜4を実行するネイティブコードを生成し、動的確保したJITコンパイルコードを格納するメモリにコピーします。そのメモリ領域に実行権限をつけることで、コピーしたネイティブコードを実行できるようにします。そのあとに、JITコンパイルしたコードの先頭番地にジャンプさせて、JITコンパイルコードの実行が終わったらインタープリタに戻るようにします。

Intel Core2 Duo上のUbuntu／x86環境で、DalvikVMのオリジナル（インタープリタ版）とイーフローで開発したJITコンパイラ版をEmbedded CaffeineMarkで測定した結果が図5になります。Sieve、Loop、Logicのみの測定結果となりますが、JITコンパイラ版がDalvikVMのオリジナル（インタープリタ版）に比べて2〜6倍速くなっていることがわかります。

JITコンパイラは、現在のJavaでは当たり前のように搭載されている機能です。JITコンパイラは大きな高速化を見込めますが、JITコンパイルしたコードの管理の実装、消費メモリの増加、JITコンパイルのオーバーヘッドなどデメリットもあります。

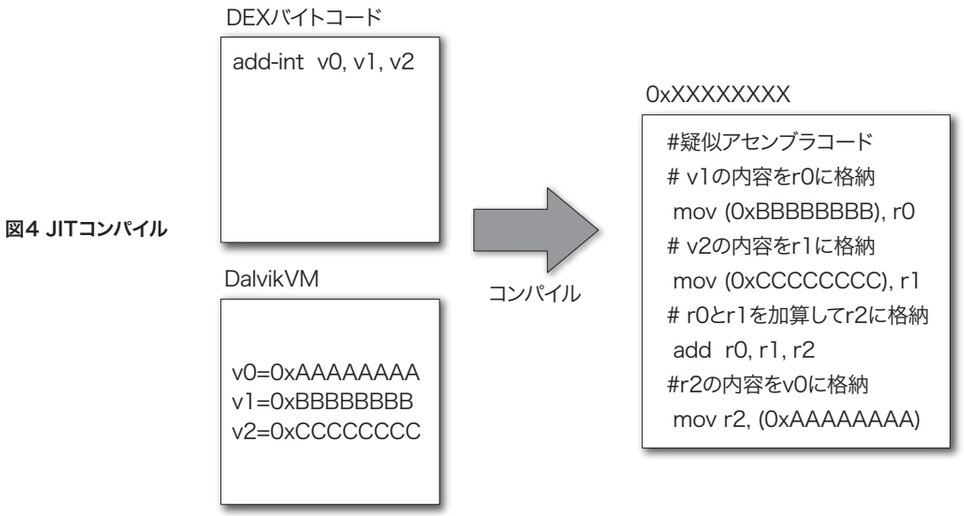
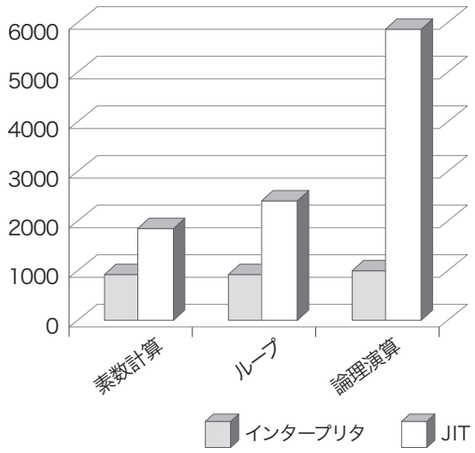


図4 JITコンパイル

図5 Embedded CaffeineMark 結果

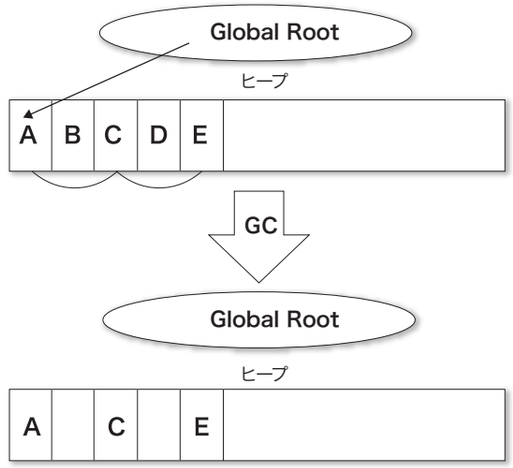


GCの最適化

DalvikVMには未使用のオブジェクトを回収するGCが実装されています。DalvikVMのGCはMark & Sweep GCが採用されています。DalvikVMのGCは、まずGCが開始される前にすべてのスレッドを停止します。図6のようにGlobal Rootから参照されるオブジェクトAが存在し、オブジェクトAがオブジェクトCを、オブジェクトCがオブジェクトEを参照しているとします。

まず、参照をたどれるオブジェクトをマークします。オブジェクトA、C、Eがマークされるオブジェクトになります。次にマークされていないオブジェクトを回収します。このときマークされていないオブジェクトはB、Dなので、オブジェクトB、Dが回収されます。オブジェクトの回収が終わったらスレッドを再開させます。DalvikVMのGCのメリットは、実装が容易であることです。一方デメリットとしては、GCの停止時間はヒープサイズが大きくなると増加していくことが挙げられます。また、DalvikVMのGCではオブジェクト回収後に図6のようにコンパクションを行わないので、オブジェクトAとオブジェクトCの間のように歯抜けの状態になります。

図6 Mark & Sweep GC



実際Android Emulator環境でGCが発生すると200~500msの間処理が停止します。Google Dev Phone 1の場合約100msの間処理が停止します。100msの間すべてのスレッドは停止するため、結果的にAndroidアプリケーションの処理が遅いように見えてしまいます。GCにはいくつかのアルゴリズムがあります。DalvikVMのGCをそれらに置き換えることでAndroidアプリケーションのGCの影響を軽減させることができます。

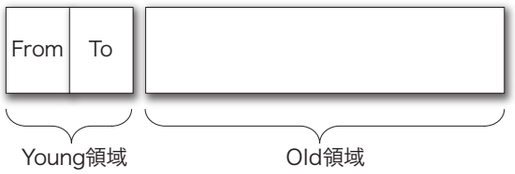
世代別GC

世代別GCは図7のようにヒープが大きく、Young領域とOld領域に分かれています。生成したばかりのインスタンスはYoung領域のFrom領域に作成されます。From領域がいっぱいになったら、参照されているオブジェクトをTo領域にコピーします。コピーしたあと、ToとFromのラベルを入れ替えます。次にオブジェクトを作成するとFrom領域の参照されているオブジェクトのあとからオブジェクトを確保していきます。このGCをマイナーGCといいます。

マイナーGCの領域は小さいため、GCの停止時間は短くなります。何回かマイナーGCを実行しYoung領域に残ったオブジェクトはOld領域に移動します。Old領域がいっぱいになった場合は、Old領域でMark & Compact GCを実行します。Mark & Compact GCはMark & Sweep　GCを実行後、コンパクションを実行してヒープの歯抜けを防ぎます。このGCをメジャーGCもしくはフルGCといいます。フルGCはMark & Sweep GCと同様にGC停止時間が長いです。

オブジェクト指向プログラミングではオブジェクトの寿命が短く、サイズが小さい大量のオブジェクトが生成されます。世代別GCでは積極的にYoung領域でオブジェクトを回収し、長く使うオブジェクトをOld領域に残すことでフルGCが発生するのを軽減させます。結果的にすぐ終わるGCが頻繁に発生しますが、長時間停止するフルGCを軽減することで、全体的に性能を上げることが可能になります。

図7 世代別GC



Incremental GC

Incremental GCはオープンソースのBoehmGCで使われているGCです。GCとしてはMark & Sweep GCと同じオブジェクト管理方式なのですが、オブジェクトがいっぱいになってからGCを開始するのではなく、ユーザープログラムと交互に短い間隔でGCを少しずつ実行します。GCの1回あたりの停止時間は短くなりますが、トータルのGC時間はMark & Sweep GCと同じになります。長時間停止すると不都合の生じるアプリケーションが多い場合は、このGCが適しています。

リアルタイムGC

Mark & Sweep GCや世代別GCなどはGC実行中にすべてのスレッドが停止してしまいます。それに対してリアルタイムGCはGC実行中でもスレッドが止まらない方式、またはGCの最長応答時間を保証するGCを指します。現在、リアルタイムGCについては実用化までこぎつけたケースは少ないですが、リアルタイムGCの実用化に向けた研究開発は着実に進められており、より一層の高速化実現など今後の成果が期待されます。組

第3章

Androidネイティブコードデバッグ手法

株式会社イーフロー
事業戦略本部 R&D
久納 孝治
Hisano Koji

はじめに

昨年Androidのソースコードが公開されました。ユーザが日々使うAndroidアプリから、下支えのLinux OS／ドライバの部分まで、ほとんどすべてのソースコードを読めるようになりました。読者の中には実際にソースコードを読み進めている方も大勢いらっしゃると思います。

他の記事や書籍で通常のAndroidアプリの開発手法は多々解説されていますので、本章ではAndroidアプリより下の層に取り組む際の開発手法について解説したいと思います。

通常のAndroidアプリの開発にあたっては、Googleより提供されているAndroid Development Toolkit（以下、ADT）を使って、Eclipse上で効率的にコードを書くことができます。さらに先日公開されたNative Development Kit（以下NDK）を用いることによって、Androidアプリ内で使うネイティブライブラリの開発も効率的に行えるようになりました。

ですが、Googleから公開されているADTやNDKは、通常のAndroidアプリの開発向けです。より下位層のネイティブアプリやシステムライブラリなどの開発には用いることができません。そのため、これらを開発する際には、自分自身で開発環境を整備する必要があります。

では、実際に"Hello, World!"と表示するシンプルなネイティブアプリ（コマンドラインアプリ、以下helloアプリ）を題材に開発環境を徐々に整えていきましょう。

makeコマンド＋エディタによる開発

一番シンプルな開発環境は、makeコマンド + エディタによる開発です。公開されているAndroidソースコードは、すべてmakeを用いてビルドされます。したがって、新規にディレクトリを追加して適切にファイル構成を行えば、短時間でネイティブアプリを開発できる環境を整えられます（開発環境の整備が短いのであって、開発時間が短いわけではありません）。

helloアプリをAndroidの流儀に従って作成してみましょう。まずはチェックアウトしたAndroidソースコードのトップディレクトリ直下（以下MYDROID環境変数にディレクトリパスが登録されているものとします）に、helloディレクトリを作成して下記のファイル群を作成します。

$MYDROID/hello/Main.cの内容

```
#include <stdio.h>

int main（int argc, char *argv[]）
{
    printf（"Hello, World!\n"）;
    return 0;
}
```

$MYDROID/hello/Android.mkの内容

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

LOCAL_SRC_FILES:= \
        Main.c

LOCAL_MODULE:= hello

include $（BUILD_EXECUTABLE）
```

その後、下記のようにコマンドを入力してビルドを行います。

コマンド入力

```
$ cd $MYDOROID
$ make
```

ビルドが完了したら、1.5から導入されたAndroid Virtual Diskを使わずにビルドしたイメージファイルを直接指定してエミュレータを起動します。

コマンド入力

```
$ emulator -kernel $MYDROID/prebuilt/android-arm/
kernel/kernel-qemu -sysdir $MYDROID/out/target/
product/generic -data $MYDROID/out/target/product/
generic/userdata.img &
```

しばらく待ってホームスクリーンが表示されたら、実際に実行してみましょう。

コマンド入力

```
$ adb shell hello
Hello, World!
```

念のため、/system/binフォルダ以下にhelloアプリがインストールされていることを確認しましょう。

コマンド入力

```
$ adb shell ls -l /system/bin/hello
-rwxr-xr-x root     shell        5288 2009-06-26 07:11 hello
```

ここまで実行するのにどれぐらい時間がかかったでしょうか？ PC環境にもよりますが、ビルド + エミュレータ起動+アプリの起動+結果確認とひと通り流すまでに、数十分かかったかと思います。コードを変更するたびに上記の時間が必要になると、開発が極めて非効率ですので、順に改善していきます。

アプリをファイル転送して時間短縮

Androidエミュレータが起動するまでには、長い時間待たなければいけません。まずは、ビルドしたあとにAndroidエミュレータを毎回起動する必要がなく試せるようにします。

ビルドしたアプリは、$MYDROID/out/target/product/generic/system/bin/helloとして生成されます。エミュレータを起動している状態で、このファイルを/dataフォルダ以下に転送して実行します。/systemディレクトリ以下はデフォルトで書き込み禁止になっており、remoutコマンドを実行して書き込むことも一応可能ではありますが、ひと手間かかるので/dataディレクトリを使っていきます。

コマンド入力

```
$ adb push $MYDROID/out/target/product/generic/
system/bin/hello /data
56 KB/s (5288 bytes in 0.091s)
$ adb shell /data/hello
Hello, World!
```

これによりエミュレータを起動した状態で、アプリを転送して試せるようになったので、時間が短縮されたはずです。

ディレクトリ単位でビルドして時間短縮

トップディレクトリでmakeすると、helloアプリとは関係のないフォルダもチェックしてビルドするので、長い時間がかかります。次はhelloアプリ単体でビルドできるようにして時間を短縮します。

幸いなことに、Androidソースコードにはディレクトリ単体でビルドするためのシェルスクリプト（$MYDROID/build/envsetup.sh）が含まれています。このシェルスクリプトは/bin/shがbashであることを前提に書かれているので、最近のUbuntuのように/bin/shがdashになっている場合には、下記をsudoコマンドで実行して変更する必要があります。

コマンド入力

```
$ ln -sf /bin/bash /bin/sh
```

/bin/shがbashであることを確認して準備が整ったら、sourceコマンドで現在の実行コンテキストにコマンド群を登録します。

コマンド入力

```
$ cd $MYDROID
$ source build/envsetup.sh
```

コマンド群が登録されたので、指定ディレクトリ以下をビルドするmmmコマンドを用いて、helloモジュール単体でビルドして実行してみましょう。

コマンド入力

```
$ mmm hello
（途中の出力は略）
$ adb push $MYDROID/out/target/product/generic/
system/bin/hello /data
64 KB/s (5288 bytes in 0.080s)
$ adb shell /data/hello
Hello, World!
```

上記作業の入力を別のシェルスクリプトとして書いておくと、より短時間でビルド + 実行が行えるようになります。

envsetup.shには、説明以外にもさまざまな便利なコマンドがあります。ひと通り内容をチェックされることをお勧めします。

デバッグ環境を整備して時間短縮

helloアプリは非常にシンプルなプログラムでしたが、複雑なプログラムになると、デバッグのためにデバッガを使いたいところです。ここでは、フリーで使えるgdbを用いてデバッグ環境を整えます。前項のenvsetup.shをひと通りチェックされた方は、gdbclientというコマンドが入っていることに気づかれたと思いますが、このコマンドを使ってデバッグします。まずは、デバッグを行いやすいようにhelloアプリにデバッグ情報を付加 + 最適化をオフにしてコード順序が変わらないように変更します。Android.mkの中央に、以下の行を追加してください。

追加行

```
LOCAL_CFLAGS += -O0 -g
```

次に、Androidエミュレータ内でgdbserverを立ちあげてデバッグできるように準備を行います。helloアプリからの出力をすぐに確認できると便利なので、新しくシェルを立ちあげて下記コマンドを入力してください。

コマンド入力

```
$ adb shell gdbserver :5039 /data/hello
Process /data/hello created; pid = 732
Listening on port 5039
```

その後、前のシェルに戻って下記コマンドを入力してデバッグを開始します。

コマンド入力

```
$ export PATH= $PATH: $MYDROID/prebuilt/linux-x86/
toolchain/arm-eabi-4.3.1/bin
$ gdbclient hello :5039 /data/hello
```

gdbserverを用いたリモートデバッグですが、gdbclientコマンドを実行してgdbのプロンプトが表示されたあとは、通常のgdbコマンドでデバッグできるようになります。main文で止めてargc、argvの値を表示してみましょう。

コマンド入力

```
(gdb) b main
Breakpoint 1 at 0x8370: file hello/Main.c, line 5.
(gdb) c
...
Breakpoint 1, main (argc=1, argv=0xbef52d94) at hello/
Main.c:5
5               printf ("Hello, World!\n") ;
Current language:  auto; currently c
(gdb) p argc
$1 = 1
(gdb) p argv[0]
$2 = 0xbef52e5e "/data/hello"
```

実行を再開するともうひとつのシェルに文字列が表示されます。

コマンド入力

```
(gdb) c
```

デバッグが終わったのでgdbを終了します。

コマンド入力

```
(gdb) q
```

上記の手順でgdbデバッグを行った際には、いくつか警告が表示されます。これは、gdbがデバッグ用に存在すると仮定している関数がAndroidにはなかったり、Androidのネイティブコードがprelinkされているため表示されます。しかし実用上大きな問題はないため無視してもかまいません。

統合開発環境で時間短縮

ここまでは、エディタ＋gdbを用いて開発を行ってきました。ですが、開発をさらに効率化させるために、統合開発環境でコードの編集＋デバッグを行えるようにしたいところです。

そこで、Eclipse 3.5のC/C++開発環境のCDTをセットアップします。Eclipse 3.4＋Device Software Development PlatformプロジェクトのDevice Debuggingプラグインでも一部可能ではありますが、バグの修正や新機能によりデバッグしやすくなっているのでEclipse 3.5の利用を強くお勧めします。

まずは、先日リリースされたばかりのEclipse 3.5のEclipse IDE for C/C++ Developersパッケージをダウンロードして適当な場所に解凍します。

ダウンロードURL

```
http://www.eclipse.org/downloads/download.php?file=/
technology/epp/downloads/release/galileo/R/eclipse-cpp-
galileo-linux-gtk.tar.gz
```

[File - New - C Project]メニューから、新規にCプロジェクトを作成します。図1のように入力を行い、[Next]ボタンで次のページに進み、[Finish]ボタンを押してプロジェクトを作成します。

プロジェクトが作成できたら、下記ファイルをプロジェクトルートに作成します。

$MYDROID/hello/build.shの内容

```
#!/bin/sh

cd ..
source build/envsetup.sh
mmm hello
```

作成後、このファイルに実行権限をつけて実行してみましょう。

コマンド入力

```
$ cd $MYDROID/hello
$ chmod u+x build.sh
$ ./build.sh
```

無事に実行できたら、プロジェクトの自動ビルド設定を行います。Project Explorerビューのhelloプロジェクト上で右クリックメニューを開き、Properties

図1　プロジェクトの作成

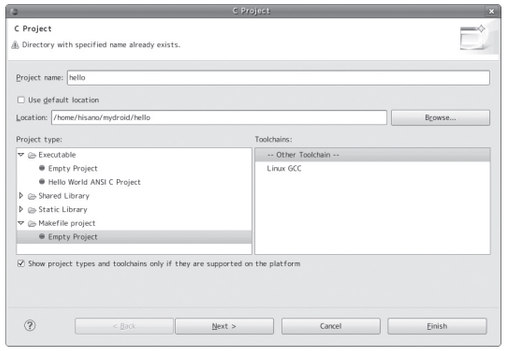
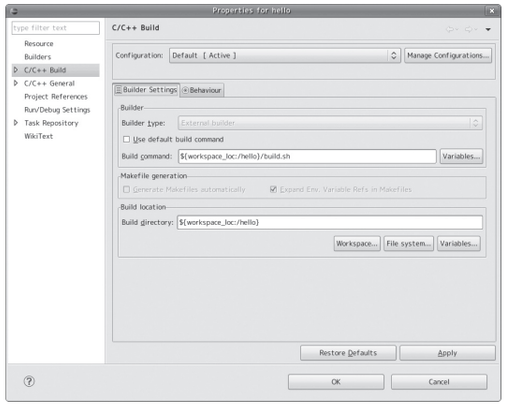


図2　Builder Settingsタブの設定



メニューからプロジェクトプロパティウィンドウを表示します。[C/C++ Build]ページを開いて、Builder Settingsタブを図2のように、Behaviourタブを図3のように設定します。

それでは、自動ビルドが行われるかどうか、Main.cを以下のように変更して保存してください。

$MYDROID/hello/Main.cの内容

```
#include <stdio.h>

int main  (int argc, char *argv[])
{
   printf  ("Hello, World!\n") ;
   error;
   return 0
}
```

図3　Behaviourタブの設定

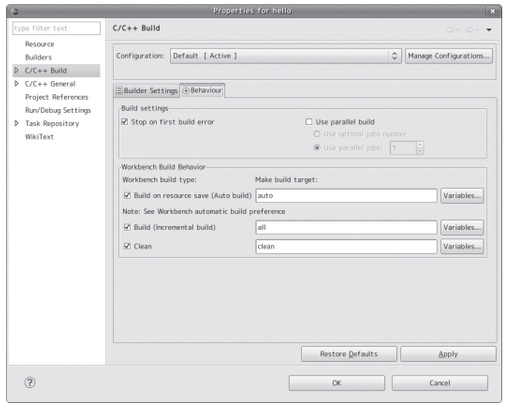


図4　ビルド結果

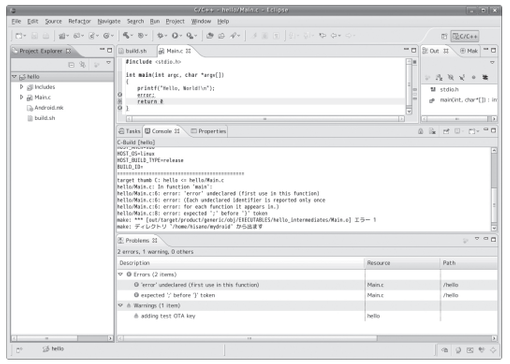


図4のように、Consoleビューにビルド結果メッセージが表示されて、エディタ上にエラーマーカー＋Problemビュー上にエラーが表示されたら成功です。成功したら、ソースコードを元に戻して保存します。

Builder Settingsタブで、保存時のビルド／自動ビルド／クリーン時それぞれ別のコマンドラインオプションが渡されるように設定しました。それぞれ別々の処理を行うようにbuild.shを以下のように書き直します。

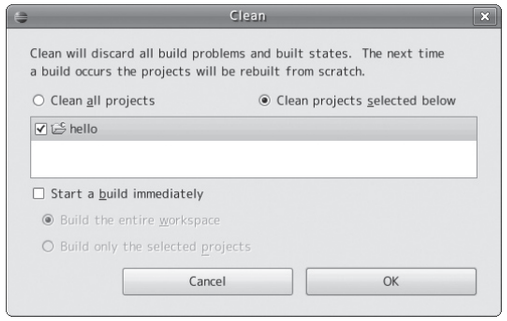
$MYDROID/hello/build.shの内容

```
#!/bin/sh

case $1 in
    auto | all )
        echo "building..."
        cd ..
        source build/envsetup.sh
        mmm hello
        ;;
    clean )
        echo "cleaning..."
        ;;
esac
```

では、実際にクリーンを行って試してみます。[Project - Build Automatically]メニューのチェックを外して[Build - Clean...]メニューを選択し、次のウィンドウで図5のように入力してクリーンを行います。

図5　クリーン処理



Consoleビューに"cleaning..."と表示されたら成功です。保存時にビルドが行われる方が便利なので、[Project - Build Automatically]メニューのチェックをつけて元に戻します。

自動ビルドができるようになったので、次はinclude文／関数などの入力時にコードアシストが働くようにしたり、F3キーを押すことにより関数の実装コードを参照できるように設定しましょう。$MYDROIDディレクトリは開発者ごとに違うため、Eclipseのビルド変数／パス変数を用いて、プロジェクトとは別に設定する形にします。こうすることにより、プロジェクトファイル内には$MYDROIDの絶対パスが含まれないようになり、プロジェクトのファイル群をソースコード管理システムに入れたとしても、チェックアウト時にプロジェクト設定を修正する手間を省けるようになります。

[Window - Preferences]メニューからPreferencesウィンドウを開いて、[C/C++ - Build Variables]ページを開いてください。[Add...]ボタンを押して、図6のようにビルド変数を追加します。ビルド変数のValue値には$MYDROIDディレクトリの絶対パスを指定してください。

図6　ビルド変数の追加

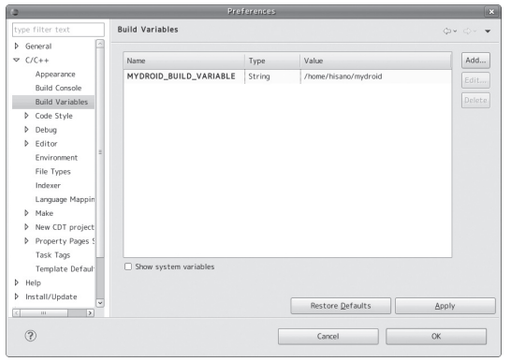
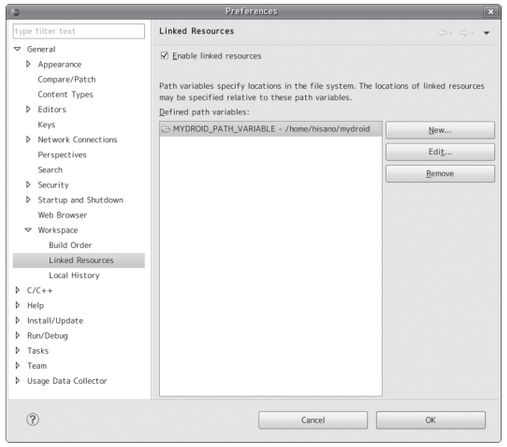


図7　パス変数の追加



次に[Window - Preferences]メニューからPreferencesウィンドウを開いて、[General - Workspace - Linked Resources]ページを開いてください。[New...]ボタンを押して、図7のようにパス変数を追加します。パス変数のLocation値には$MYDROIDディレクトリの絶対パスを指定してください。

その後、プロジェクトプロパティウィンドウを開いて、[C/C++ General - Paths and Symbols]ページの[Includes - Languages - GNU C]項目を図8のように設定してください。[Includes - Languages - Assembly]項目と[Includes - Languages - GNU C++]項目の内容は空にします。

最後に[File - New - Folder]メニューから、図9のように入力してリンクフォルダを作成します。

図8　Includes項目の設定

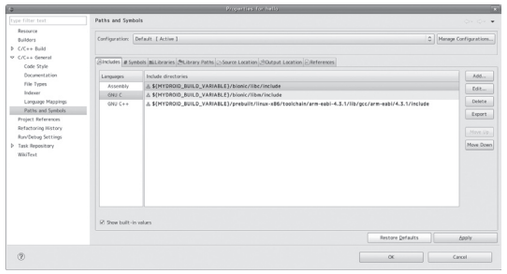
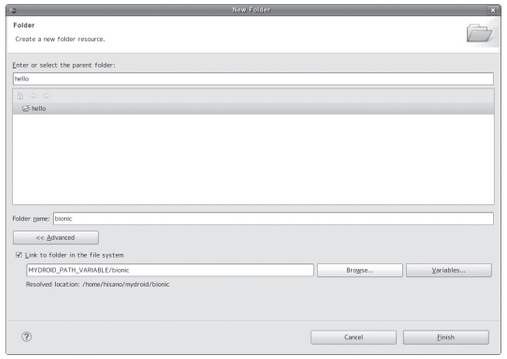


図9　リンクフォルダの作成



これで、コード編集中にコードアシストを使えたり、ソースコード中の関数をたどっていくことができるようになりました。

Eclipseから、実行／デバッグもできるようにしてみましょう。gdbserverを用いたリモートデバッグとなるため、残念ながら実行とデバッグは別々に設定する必要があります。

まずは実行設定からです。新規に下記のファイルを追加します。

$MYDROID/hello/launch.shの内容

```
#!/bin/sh

cd ..
adb push out/target/product/generic/system/bin/hello /
data >&/dev/null
adb shell /data/hello " $@"
```

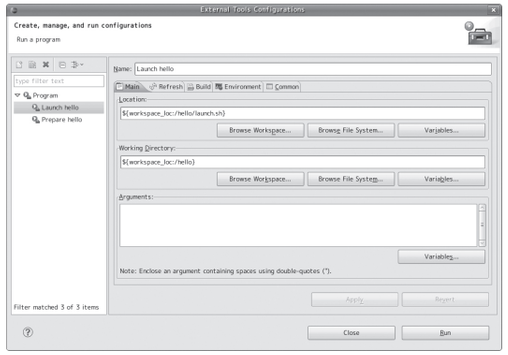
作成後、このファイルに実行権限をつけて実行してみましょう。

コマンド入力

```
$ cd $MYDROID/hello
$ chmod u+x launch.sh
$ ./launch.sh
```

無事に実行できたら、このシェルスクリプトを簡単に実行できるように登録します。[Run – External Tools – External Tools Configurations…]メニューを選択して、図10のように実行設定を行います。

図10　実行設定

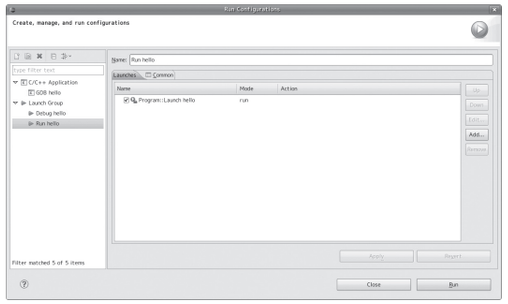


通常の実行設定（[Run – Run Configurations…]メニュー）ではなく、外部ツールの実行設定にしていることに注意してください。CDTの通常の実行設定では、生成したバイナリでなければ実行することができないため、外部ツールとしてシェルスクリプトを登録しています。

その後、[Build]タブに移って[Build before launch]チェックボックスのチェックを外します。
設定が完了したら、[Run]ボタンを押して試してみてください。

無事に実行できることを確認できたら、外部ツールではなく、通常の実行メニューに表示されるように設定します。[Run – Run Configurations…]メニューを選択して、Launch Groupツリーに図11のように通常の実行設定として追加してください。

図11　実行メニューにも表示させる



続けて、デバッグ設定も追加しましょう。新規に下記のファイルを追加します。

$MYDROID/hello/prepare.shの内容

```
#!/bin/sh

cd ..
adb push out/target/product/generic/system/
bin/hello /data >&/dev/null
adb forward tcp:5039 tcp:5039
adb shell gdbserver :5039 /data/hello " $@"
```

作成後、このファイルに実行権限をつけます。

コマンド入力

```
$ cd $MYDROID/hello
$ chmod u+x prepare.sh
$ ./prepare.sh
```

次に、このシェルスクリプトを [Run – External Tools – External Tools Configurations…]メニューから、図12のように外部ツールに追加します。

その後、[Build]タブに移って[Build before launch]チェックボックスのチェックを外します。これで

図12　外部ツールに追加

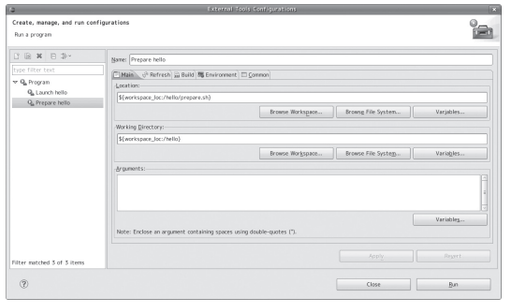


図13　gdb側の設定(1)

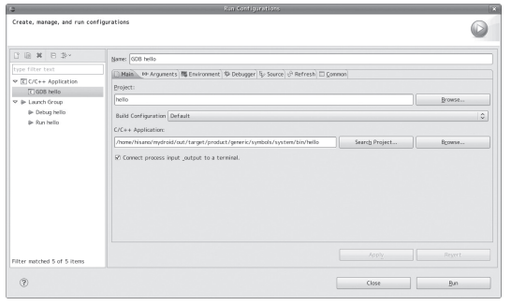
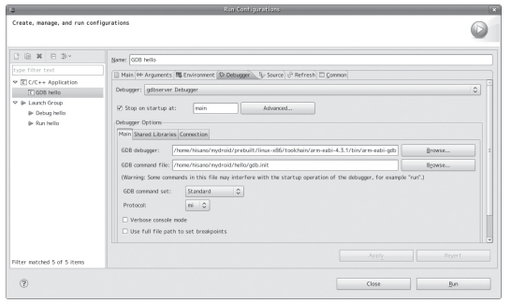


図14　gdb側の設定(2)



gdbserver側の実行設定は完了です。

gdb側の実行設定は、[Run – Run Configurations…]メニューから、図13–図16のように設定して追加します。

gdbserver側／gdb側両方の実行設定が整ったので、図17のようにワンクリックで起動できるようにLaunch Groupに登録します。「GDB hello」を追加する際に、Launch Modeをdebugに設定することを忘れないでください。

図15　gdb側の設定(3)

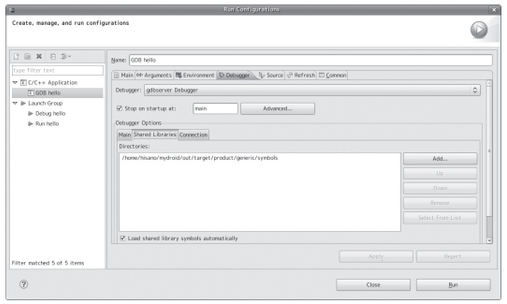


図16　gdb側の設定(4)

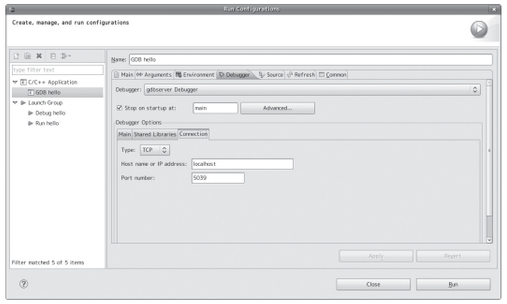
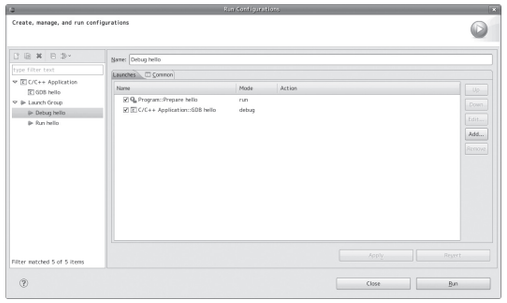


図17　Launch Groupへの登録



実際に、[Debug hello]デバッグ設定を使ってデバッグ起動してみてください（[Run-Run]メニューではなく[Run-Debug]メニューから起動）。main関数のところで止まってデバッグできるようになったはずです。

おわりに

非常に駆け足ではありましたが、C/C++の開発環境を整備する方法を解説しました。Androidのソフトウェアスタックには共有ライブラリが多く含まれていますが、静的ライブラリとして別途ビルドし、テスト用コマンドラインアプリを書いて静的リンクする形にすれば、どのようなコンポーネントであろうと効率的に開発を行えるようになります。本章の手順をベースに、より良い開発環境を探ってみてください。

column　■Android SDKに含まれているツール群

Androidの開発者ページからダウンロードできるSDKには、Androidアプリケーションの開発に役に立つツールがいくつか含まれています。以下に、それらをリストします。各ールの詳細は、http://developer.android.com/index.htmlを参照してください。

- Android Development Tools Plugin(Eclipse用)
- Android Emulator
- Android Virtual Device(AVD)
- Hierarchy Viewer
- Draw 9-patch
- Dalvik Debug Monitor Service (ddms)
- Android Debug Bridge(adb)
- Android Asset Packaging Tool(aapt)
- Android Interface Description Language(aidl)
- sqlite3
- Traceview
- mksdcard
- dx
- UI/Application Exerciser Monkey
- android

第4章

Android NDKによるJNI開発手法

株式会社イーフロー
事業統括本部 海外開発部
緒方 聡
Ogata Satoshi

2009年6月26日 に、Android 1.5 Native Development Kit Release 1（以下、Android NDK）がリリースされました。ここではAndroid NDKを使用してJNI開発を行う手法について解説します。

Android NDKとは

Android NDKはAndroidアプリ開発者向けのCおよびC++のビルド環境です。Androidはもともと JNIをサポートしていますが、Android NDKリリース以前は、共有ライブラリを作成する方法が明確ではありませんでした。今後はAndroid NDKを使用することで、以下のメリットが享受できます。

・ライブラリの作成が確実／簡単になる
・作成したライブラリの上位互換が保てる
・AndroidのC/C++ APIが使用できる

JNIを用いるメリット、デメリット

AndroidでJNIを利用した共有ライブラリを使用する最大のメリットは「実行処理の向上」と「既存のライブラリの有効活用」です。第2章の「Android高速化テクニック」で紹介されている高速化テクニックの最後の手段としてJavaコードをネイティブコードに置き換えるのは、実行処理の向上という側面では有効です。ただし、ネイティブコードにしてしまうと特定のプラットフォームに依存してしまうため、たとえばARM向けに作成したネイティブコードはx86では動作しないので、結果として作成したアプリはARM専用ということになってしまいます。

メリット	デメリット
実行速度の向上	プラットフォームに依存
既存のライブラリの活用	

メリットとデメリットを比較し、JNIを使うか使わないかをよく考えてください。もしJNIを使うなら、Android NDKを使用することで開発が圧倒的に容易になります。

ダウンロード

Android NDKは以下のURLからダウンロードが可能です。

http://developer.android.com/sdk/ndk/1.5_r1/index.html

現在はAndroid 1.5向けのRelease 1がリリースされています。Android NDKはAndroid SDKと同じくWindows、Mac OS X（Intel）、Linux 32/64ビット（x86）のプラットフォームをサポートしています。それぞれの好みの環境のAndroid NDKをダウンロードしてください。なお、Windowsの場合、別途Cygwin環境が必要になります。今回の例ではUbuntu 8.04 LTS上に環境を構築することにします。

事前準備

Android NDKを使用するだけなら、初期状態のUbuntuに必要なコマンドはすべて入っています。makeのバージョンは3.81を期待しているので、ターミナルから以下のコマンドでバージョンをチェックしてみてください。

```
make -v
```

GNU Make 3.81と出力されれば問題ありません。ソースリポジトリなどを変更していなければ基本的に問題ないはずです。

インストール

ダウンロードしたアーカイブを適当なディレクトリに展開します。ここではユーザーホームディレクトリ/home/ogata/android-ndk-1.5_r1（以下、NDKホームディレクトリ）に展開したものとして説明します。インストールはNDKホームディレクトリでターミナルから以下のコマンドを入力します。

```
bash ./build/host-setup.sh
```

docs/INSTALL.TXTではbashは必要ないように記載されていますが、使用するシェルはbashを期待しています。最近のUbuntuでは/bin/shはbashではなくdashにリンクされているので、明示的にbashを指定してシェルスクリプトを実行します。

host-setup.shを実行すると、out/host/config.mkが作成されます。筆者の環境で作成されたconfig.mkは以下のとおりです。

```
HOST_OS       := linux
HOST_ARCH     := x86
HOST_TAG      := linux-x86
HOST_CC       := gcc
HOST_CFLAGS   :=
HOST_CXX      := g++
HOST_CXXFLAGS :=
HOST_LD       := gcc
HOST_LDFLAGS  :=
HOST_AR       := ar
HOST_ARFLAGS  :=
```

作成された直後の状態では、コンパイルオプションなどが空です。ここで設定するとすべてのプロジェクトのデフォルトになります。コンパイルオプションはプロジェクトごとに個別に指定することも可能で、その方法は後述します。

サンプルのビルド

Android NDKにはあらかじめふたつのサンプルが付属しています。このうちのひとつ「hello-jni」をビルドしてみましょう。ビルドの方法はNDKホームディレクトリでターミナルから以下のように入力します。

```
make APP=hello-jni
```

APPには、ビルドしたいアプリのプロジェクト名を指

定します。プロジェクトはapps配下にプロジェクト名でディレクトリが存在していなければなりません。現在のリリースでは、以下の引数がサポートされています。

APP=<name>	<name>にプロジェクト名を指定（必須）
V=1	ビルド時に詳細な出力を行う
-B	必ずリビルドを行う

ビルドが正常に行えたら、プロジェクトディレクトリのlibsフォルダに作成した共有ライブラリがコピーされます。あとはADTでパッケージングすれば共有ライブラリは.apkに含まれ配布可能になります。

新規プロジェクトの作成

サンプルがビルドできて、Android NDKのインストールが正常に行えていることが確認できたら、さっそく自分でプロジェクトを作成してみましょう。今回作るアプリはネイティブから文字列を返して、それをJava側で受け取って画面に表示する、という簡単なものです。まずはEclipseで以下のようなプロジェクトを作成します。

表1　コマンド一覧

Project name	HelloNDK
Build Target	Android 1.5
Application name	Hello NDK
Package name	com.example.android.ndk.hello
Create Activity	HelloNDK

以下は全ソースコードです。

```
package com.example.android.ndk.hello;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
public class HelloNDK extends Activity {
    static {
        System.loadLibrary ("HelloNDK") ;
    }
    private native String hello () ;
    @Override
    public void onCreate (Bundle savedInstanceState) {
        super.onCreate (savedInstanceState) ;
        TextView textView = new TextView (this) ;
        textView.setText (hello ()) ;
        setContentView (textView) ;
    }
}
```

このソースコード以外、AndroidManifest.xmlやres/配下のXMLなどはとくに変更する必要はありません。

ソースコードで特筆すべき点は2個所です。ひとつはnativeメソッドの宣言、もうひとつはSystem.loadLibrary（String）メソッドによるライブラリのロードです。Android 1.5からは、JNIから呼び出す共有ライブラリは.apkの作成時に自動的にパッケージングされ、インストール時に自動的に/data/data/<package>/lib に展開され、アプリ起動時に自動的に上記パスでライブラリの動的ロードが行えるようになるため、Android 1.1以前のように、/system/libに配置したり、.apkから共有ライブラリを書き出し読み込むような自前のローダー機能を準備したりしなくてもよくなりました。また、自動的に展開された共有ライブラリはアプリのアンインストール時に自動的に削除されます。

ネイティブコードの作成

作成した新規Androidプロジェクトのbinディレクトリで、以下のコマンドをターミナルから実行します。

```
javah com.example.android.ndk.hello.HelloNDK
```

カレントディレクトリにcom_example_android_ndk_hello_HelloNDK.hが作成されたと思います。このヘッダファイルをベースにして、HelloNDK.cというソースコードを作成します。

```
#include <jni.h>
JNIEXPORT jstring JNICALL
Java_com_example_android_ndk_hello_HelloNDK_hello
(JNIEnv *env, jobject thiz) {
    return (* env) ->NewStringUTF (env, " こんにちは ,
NDK") ;
}
```

作成したソースコードはどこにあっても問題ないのですが、管理がしやすいようにNDKホームディレクトリのsources/HelloNDK/HelloNDK.cに保存することにします。この場所に保存する理由は後述します。ソースコードはUTF-8で保存することを忘れないでください。

Application.mkの設定

ここから先はAndroid NDKの設定になります。Android NDKでは新規プロジェクトのためにApplication.mkとAndroid.mkというふたつの設定ファイルを作成する必要があります。

ひとつはNDKホームディレクトリ配下に以下の内容でapps/HelloNDK/Application.mkというファイルを作成します。

```
APP_PROJECT_PATH := /home/ogata/workspace/HelloNDK
APP_MODULES      := HelloNDK
```

APP_PROJECT_PATHは相対パスでも記述できます。今回の例は絶対パスですが、相対パスの記述方法も後述します。上記ふたつは必ず定義しなければなりません。Application.mkに使用可能な定義はこれらを含め、以下が用意されています。

APP_PROJECT_PATH	プロジェクトパスを指定（必須）
APP_MODULES	モジュール名を指定（必須）
APP_OPTIM	debugかreleaseを指定（releaseがデフォルト）
APP_CFLAGS	Cソースコンパイルフラグを指定
APP_CXXFLAGS	C++ソースコンパイルフラグを指定
APP_CPPFLAGS	CとC++で共通のコンパイルフラグを指定

これらは、このアプリ共通の定義です。ひとつのアプリで複数のライブラリを使用する場合などで、ライブラリごとにコンパイルフラグを変更したい場合は後述のAndroid.mkに設定します。

Android.mkの設定

ソースファイルは、NDKホームディレクトリ配下にsources/HelloNDKというディレクトリを作成して、そこに保存しました。同じ場所に以下の内容でAndroid.mkを作成します。

```
LOCAL_PATH := $（call my-dir）
include $（CLEAR_VARS）
LOCAL_MODULE    := HelloNDK
LOCAL_SRC_FILES := HelloNDK.c
include $（BUILD_SHARED_LIBRARY）
```

LOCAL_PATHは必ず最初に定義しなければなりません。ここではAndroid.mkのあるディレクトリをmy-dirというマクロを使用してLOCAL_PATHに設定しています。次の“include $(CLEAR_VARS)”は、LOCAL_PATHを除くLOCAL_xxxの定義をクリーンアップします。複数のライブラリを使用する場合などに、先に読み込んだAndroid.mkのLOCAL_xxxの値が不本意に使用されてしまう、という障害を防ぐた

めにも必ず定義しておきましょう。

LOCAL_MODULEはモジュール名を指定します。この名前はユニークでなければならず、スペースを含んではいけません。ビルドシステムはここで与えられた名前にプレフィックスとサフィックスを自動的に付与し、今回の場合ではlibHelloNDK.soという共有ライブラリを生成します（もしここでlibHelloNDKという名前を指定した場合、生成される共有ライブラリはliblibHelloNDK.soではなく、例外的にlibHelloNDK.soとなることに注意してください）。

LOCAL_SRC_FILESにはCまたはC++のソースリストを指定します。ヘッダは含めません。C++ソースファイルの拡張子は.cppがデフォルトです。“include $(BUILD_SHARED_LIBRARY)”は共有ライブラリを作成する際に指定します。

上記がAndroid.mkの最小セットで、これだけ定義すれば共有ライブラリのビルドは可能です。もっと詳しく知りたい方のために、上記を含めたAndroid.mkで使用可能なすべての定義、マクロ、機能を以下で説明します。

●Android NDKが提供する定義

・CLEAR_VARS

“LOCAL_”で始まる定義を未定義にします。新しいモジュールの記述前にインクルードすべきです。

・BUILD_SHARED_LIBRARY

“LOCAL_”で始まる定義をすべて記述したあとでこの定義をインクルードします。少なくともLOCAL_MODULEとLOCAL_SRC_FILESが事前に定義されていなければなりません。

・BUILD_STATIC_LIBRARY

静的ライブラリを作成する際に指定します。静的ライブラリはアプリのプロジェクトにコピーされませんが、他の共有ライブラリから使用できるようになっています。なお、作成される静的ライブラリのファイル名は“lib$(LOCAL_MODULE).a”となります。

・TARGET_ARCH

ターゲットCPUのアーキテクチャが定義されています。ARM9もARM11も“arm”であることに注意が必要です。

・TARGET_PLATFORM

ターゲットプラットフォームが定義されています。現時点では“android-1.5”だけがサポートされています。

・TARGET_ARCH_ABI

Android.mkが構文解析されるターゲットのCPUとABIの名前の定義です。現在は“arm”だけがサポートされています。これはARMv5TE以上のsoft-float版浮動小数点をサポートしたCPUを意味します。将来的には“arm”以外のABIもサポートされます。

・TARGET_ABI

“$(TARGET_PLATFORM)-$(TARGET_ARCH_ABI)”という定義を持ちます。デフォルトは“android-1.5-arm”です。

●Android NDKが提供するマクロ

マクロは“$(call マクロ)”のように使用します。

・my-dir

NDKビルドシステムに、このAndroid.mkが置かれているディレクトリのパスを返します。

・all-subdir-makefiles

このAndroid.mkのディレクトリのサブディレクトリにあるAndroid.mkを探します。たとえばディレクトリとAndroid.mkが以下の構成になっていたとします。

```
sources/foo/Android.mk
sources/foo/lib1/Android.mk
sources/foo/lib2/Android.mk
```

この際、sources/foo/Android.mkに以下の行が含まれていると、sources/foo/lib1/Android.mkとsources/foo/lib2/Android.mkも自動的にビルドに含まれます。

```
include $（call all-subdir-makefiles）
```

この定義はサブディレクトリのサブディレクトリまでは探さないことに注意してください。

・this-makefile

現在のMakefileのパスを返します。

・parent-makefile

呼び出し元のMakefileのパスを返します。

・grand-parent-makefile

呼び出し元の呼び出し元を返します。

●Android NDKが提供するモジュール記述変数

これらは“include $(CLEAR_VARS)”と“include $(BUILD_XXXXX)”の間で定義すべきです。

・LOCAL_PATH

このAndroid.mkに記載されるすべてのパスのベースとなるパスを指定します。必ず先頭で定義しなければなりません。

・LOCAL_MODULE

モジュール名を指定します。すべてのモジュールでユニークな名前で、かつスペースを含まないようにし、必ず定義しなければなりません。NDKビルドシステムによって作成されるライブラリ名は、このモジュール名にプレフィックスとサフィックスが付与されますが、モジュール間での参照の際には、プレフィックスとサフィックスを指定する必要がないことに注意してください。

・LOCAL_SRC_FILES

モジュールを構成するソースファイルのパスリストを指定します。ソースファイルのパスはLOCAL_PATHからの相対パスで指定します。以下は複数指定の例です。

```
LOCAL_SRC_FILES := foo.c \
                   toto/bar.c
```

・LOCAL_CPP_EXTENSION

C++のソースファイルの拡張子を指定します。デフォルトの拡張子は.cppです。拡張子を.cxxに変更したい場合は、以下のように指定します。

```
LOCAL_CPP_EXTENSION := .cxx
```

・LOCAL_CFLAGS

Cソースファイルのコンパイルフラグを指定します。このフラグはC++ソースファイルのコンパイルには適用されません。

・LOCAL_CXXFLAGS

C++ソースファイルのコンパイルフラグを指定します。このフラグは、Cソースファイルのコンパイルには適用されません。

・LOCAL_CPPFLAGS

CソースファイルとC++ソースファイルの両方に対してコンパイルフラグを指定します。

・LOCAL_STATIC_LIBRARIES

BUILD_STATIC_LIBRARYで指定したモジュールでリンクしたいものをリストで指定します。この指定は、共有ライブラリを作成する場合でのみ指定可能です。

・LOCAL_SHARED_LIBRARIES

このモジュールが実行時に参照する共有ライブラリを指定します。

・LOCAL_LDLIBS

ライブラリのビルド時に必要な追加のリンクフラグを指定します。“-l<library>”でリンカに通知する方法と同じです。

・LOCAL_ALLOW_UNDEFINED_SYMBOLS

デフォルトの動作として、未定義の何かを参照している場合、共有ライブラリ作成時に“undefined symbol”エラーが発生します。何らかの理由でエラーを発生させたくない場合は、この定義をtrueにします。

上記以外にAndroid.mkで独自定義を行いたい場合、その定義名に“MY_”というプレフィックスを用いることが推奨されています。以下のプレフィックスはNDK

ビルドシステムが予約しています。

- **“LOCAL_”で始まる定義**
- **“PRIVATE_”、“NDK_”、“APP_”で始まる定義**
- **小文字の定義**

その他有用な情報

Android NDKを使用する際に知っておくべき情報、CやC++のサポート範囲や制限事項、Android NDKが提供する関数などをまとめます。

●Cのサポート

Cライブラリはlibcではなく、BionicというAndroid独自のものなので、ANSI C準拠ではありません。ビルド時にANSI C標準のヘッダファイルがなかったり、ヘッダファイルはあるけれど定義されているはずの関数がなかったりした場合、それらはBionicには実装されていないので、自分で代替関数を用意するか、あるいは別の方法を考えなければなりません。

NDKビルドシステムはCライブラリを自動的にリンクするので、LOCAL_LDLIBSに含める必要はありません。<math.h>も利用可能で、“-lm”をLOCAL_LDLIBSに記述する必要はありません。NDKビルドシステムはpthreadを標準でサポートするので、“LOCAL_LIBS := -lpthread”を記述する必要はありません。

ただしpthread_cancelはサポートされていません。またリアルタイム拡張も同様なので、“-lrt”も必要ありません。

ワイドキャラクタはサポートされていません。

ヘッダファイル<linux/*.h>および<asm/*.h>は将来的に変更される可能性があるため、これらヘッダファイルを直接インクルードすることは推奨されません。

●C++のサポート

極めて小さなC++のAPIがサポートされています。Android 1.5向けとして、現在は以下のヘッダだけが利用可能です。

<cstddef>
<new>
<utility>
<stl_pair.h>

また、これらは標準として必要なすべての定義を含んでいません。とくに、C++の例外とRTTIはAndroid 1.5システムイメージで使用することができません。

NDKビルドシステムはC++ライブラリを自動的にリンクするので、“-lstdc++”をLOCAL_LDLIBSに含める必要はありません。

●Android特有のログサポート

<android/log.h>は、ネイティブコードからカーネルにログメッセージを送るために使用できる定義を含んでいます。以下にそれらをリストします。

関数名	説明
__android_log_write	文字列出力
__android_log_print	文字列出力（printf 相当）
__android_log_vprint	文字列出力（va_list 版）
__android_log_assert	アサート

JavaのLogクラスと同じく、プライオリティによる出力制御、タグの指定が行えます。これら関数の引数の詳細は以下のヘッダファイルを参照してください。

build/platforms/android-1.5/common/include/android/log.h 組

第5章

プロファイリング手法

株式会社イーフロー
事業統括本部 第1事業部
山口 祐治
Yamaguchi Yuji

はじめに

作成したプログラムの性能が悪い場合、ただやみくもにプログラムを修正するのではなく、まずパフォーマンスのボトルネックになっている個所を見つけることが必要です。

本章では、Androidプログラムのボトルネックを見つけるために効果的な以下の2つのプロファイリング手法を説明します。

・Traceviewを用いたプロファイリング

Android SDKに同梱されるツールTraceviewを使います。Javaのメソッドごとに実行時間や呼び出し回数などを計測できます。

・OProfileを用いたプロファイリング

Androidのkernelに手を加えて、Javaだけではなくネイティブのモジュールレベルでプロファイリングを行います。Androidのシステム全体を開発する開発者向けの手法です。

Traceviewを用いたプロファイリング

TraceviewはJava言語のレベルでプロファイリングを行うための方法です。本節ではその方法を順を追って説明します。参考までに、本節で説明した内容は以下の環境で確認しています。

・Android SDK

OS: Windows XP Profession SP3
SDK: Android 1.5 SDK, Release 2

●プロファイリングのコードを追加

はじめに、プロファイリングを行いたい部分の開始と終了に一行ずつコードを書く必要があります。開始部分ではDebug.startMethodTracingを、終了部分ではDebug.stopMethodTracingを呼び出すようにコードを書きます。

```
Debug.startMethodTracing（"foo"）; // プロファイリング開始
…
（プロファイリングルを行いたい処理）
Debug.stopMethodTracing（）; // プロファイリング終了
…
```

プログラムを実行すると、プロファイリング結果ファイルがSDカードに書き出されます。

●プロファイリング結果の取得

プロファイリング結果ファイルをPCにコピーします。この処理には、adbコマンドのpullアクションを使います。

```
adb pull /sdcard/foo.trace
```

●プロファイリング結果の表示

PCにコピーしたプロファイリング結果ファイルをTraceviewツールで開いて分析します。先ほどコピーした結果ファイルを引数に指定してtraceviewコマンドを起動します。

```
traceview foo.trace
```

Traceviewの画面は、上下2つのパネルからなります。上部はTimeline Panel 、下部はProfile Panelといいます。

Timeline Panel（図1）は、実行しているJavaのスレッドごとに時間軸の上で実行しているメソッドを図示するものです。この線上では、各々のメソッドは違う色で区別されます。このパネルの上でカーソルを当てると、カーソルを当てた位置で実行されているメソッドが上部に表示されます。

Profile Panel（図2）は、Javaのメソッド単位で実行時間や呼び出し回数などの集計が見られる表です。

各項目の意味は以下の通りです。

・Inclusive、Incl %

メソッドの実行時間（そのメソッドから呼び出された他のメソッドの呼び出し時間含む）、および全体から見た比率を表します。

・Exclusive、Excl %

メソッドの実行時間（そのメソッドから呼び出された他のメソッドの呼び出し時を間含まない）、および全体から見た比率を表します。

・Calls+RecurCalls/Total

メソッド呼び出し回数＋再帰メソッド呼び出し回数を表します。

・Time/Call

メソッド1回呼び出しあたりの実行時間をミリセカンドで表します。

各メソッド行をクリックすると、メソッドの呼び出し関係が表示されます。Parentは呼び出し元を表し、Childrenは呼び出し先を表します。

図1 Timeline Panel

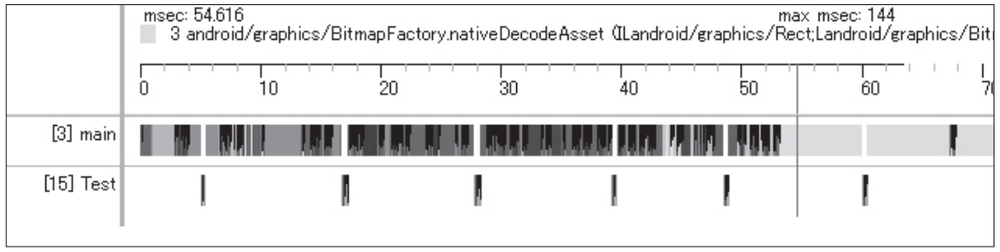


図2 Profile Panel

Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur...	Time/Call
0 (toplevel)	100.1%	142.663	7.9%	11.193	2+0	71.332
1 com/example/android/a	66.2%	94.331	2.8%	3.959	1+0	94.331
2 android/graphics/Bitmaj	41.1%	58.526	0.5%	0.730	4+0	14.632
Parents						
1 com/example/ar	69.7%	40.820			2/4	
11 android/graphic	19.6%	11.496			1/4	
16 android/graphic	10.6%	6.210			1/4	
Children						
self	1.2%	0.730				
3 android/graphics	97.1%	56.838			4/4	
48 android/content	1.2%	0.716			4/4	
83 android/content	0.2%	0.129			4/4	
87 android/content	0.2%	0.113			4/4	
3 android/graphics/Bitmaj	39.9%	56.838	39.8%	56.660	4+0	14.210

OProfileを用いたプロファイリング

OProfileはOSやCのライブラリなどのネイティブも含めたプロファイリングを行うためのものです。本節では、Google Android Dev Phone 1（以下ADP1）向けにOProfileを実行した例を説明します。参考までに、本節で説明した内容は以下の環境で確認しています。

・Android SDK

OS: Windows XP Profession SP3

SDK: Android 1.5 SDK, Release 2

・Android ビルド環境

OS: Windows XP Profession SP3

VMware: VMware Server 2.0.0.2073

Linux: Ubuntu 9.04

Source: http://source.android.com（cupcake）

●Androidファームウェアの更新

OProfileを有効にするためには、AndroidのkernelをOProfile対応にする必要があります。

■ADP1用ソースセットアップ

http://source.android.com/documentation/building-for-dream に従ってADP1のソースをセットアップします。ただし、ソースはcupcakeブランチで取得します。

```
repo init -u git://android.git.kernel.org/platform/manifest.git -b cupcake
```

また、local_manifest.xmlは以下のように設定します。

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
 <remove-project name="kernel/common"/>
 <project path="kernel" name="kernel/msm" revision="refs/heads/android-msm-2.6.27"/>
 <project path="vendor/htc/dream" name="platform/vendor/htc/dream" revision="capcake"/>
 <project path="hardware/msm7k" name="platform/hardware/msm7k" revision="capcake"/>
</manifest>
```

上記設定をしたのち、repo syncコマンドを使ってソースをチェックアウトします。以後、チェックアウトしたAndroidソースコードのトップディレクトリをMYDROID環境変数として参照します。

■kernelのコンフィグレーション取得

下記コマンドを実行して$MYDROID/kernel/.configにQualcomm MSMチップ用の設定をコピーします。

```
$ cd $MYDROID/kernel
$ make ARCH=arm msm_defconfig
```

■kernelのコンフィグレーション編集

エディタで$MYDROID/kernel/.configを開いて以下のように追加／編集します。

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
CONFIG_OPROFILE_ARMV6=y
CONFIG_LOCALVERSION="-00392-g8312baf"
# CONFIG_LOCALVERSION_AUTO is not set
```

最後の2行はADP1で無線LANを有効にするためのワークアラウンドです。Androidソースツリー上に同梱されている無線LANのモジュールwlan.koが想定しているカーネルバージョンに合わせています。

■kernelのビルド

```
$ cd $MYDROID/kernel
$ make ARCH=arm CROSS_COMPILE=../prebuilt/linux-x86/
toolchain/arm-eabi-4.2.1/bin/arm-eabi-
```

makeが正常に終了した場合、カーネルイメージが$MYDROID/kernel/arch/arm/boot/zImageに生成されます。

■Androidビルド 設定

$MYDROID/build/buildspec.mk.defaultを$MYDROID/buildspec.mkにコピーし、以下を追加します。

```
TARGET_PRODUCT:=htc_dream
TARGET_PREBUILT_KERNEL=./kernel/arch/arm/boot/
zImage
BOARD_WLAN_TI_STA_DK_ROOT=./system/wlan/ti/sta_
dk_4_0_4_32/
BOARD_HAVE_BLUETOOTH:=true
```

■Androidのビルド

```
$ cd $MYDROID
$ make
```

makeが正常に終了した場合、$MYDROID/out/target/product/dream ディレクトリにファームウェアイメージが生成されます。

■ファームウェアの書き換え

ADP1の電源をオフにしたあと、PCとUSB接続し、Back Spaceキーを押しながら電源キーを押してFASTBOOTモードで起動します。

起動したら、以下のコマンドを入力してファームウェアを書き換えます。

```
$ cd $MYDROID
$ source build/envsetup.sh
$ setpaths
$ fastboot -w flashall
```

この方法では、ユーザデータや設定も含めすべてのデータがクリアされることにご注意下さい。

●プロファイリング実行

OProfileによるプロファイルデータの収集は以下の流れに沿って行って下さい。

```
$ prebuilt/linux-x86/oprofile/bin/opreport --session-
dir=oprofile
```

■OProfile初期化

まず、OProfileの初期化を行います。以下のコマンドを適宜使用して下さい。

・プロファイラを初期化

```
# opcontrol --setup
```

・プロファイラをリセット

```
# opcontrol --reset
```

・プロファイラを設定

```
# opcontrol --event=CPU_CYCLES:150000
```

これはCPUサイクルが150000ごとに計測してカウントすることを表します。

■OProfile開始／終了

・Oprofile開始

```
# opcontrol --start
```

・Oprofile終了

```
# opcontrol –stop
```

●プロファイリング結果の閲覧

■OProfile初期化

external/oprofile/opimport_pullコマンドでデータを取得します。

```
$ cd $MYDROID
$ source build/envsetup.sh
$ setpaths
$ external/oprofile/opimport_pull oprofile
```

上記最後のoprofileオプションはoprofileフォルダに出力することを意味しています。変更する場合は、次節のsession-dir指定も変更してください。

■opreportコマンドで分析

モジュールごとの実行時間を見るためには、以下のコマンドを入力して分析します。

```
$ prebuilt/linux-x86/oprofile/bin/opreport --session-
dir=oprofile
```

出力結果は下記のようになります。

```
samples|      % |
------------------
    3768 31.5499 libwebcore.so
    2278 19.0739 libdvm.so
    1781 14.9125 libc.so
    1528 12.7941 libsgl.so
    1276 10.6841 no-vmlinux
     254  2.1268 libcorecg.so
...
```

これはブラウザでwww.google.comにアクセスして表示完了するまでをプロファイリングしたものです。Dalvik VM（libdvm.so）よりもWebkit（libwebcore.so）に時間がかかっていることがわかります。また、kernel（no-vmlinux）の実行比率もかなり高いことがわかります。

また、関数ごとの実行時間を見るためには以下のコマンドを実行します。

```
$ prebuilt/linux-x86/oprofile/bin/opreport --session-
dir=oprofile --image-path out/target/product/dream/
symbols -l
```

出力結果は下記のようになります。

```
samples  %        app name                 symbol name
1276     10.6841  no-vmlinux               /no-vmlinux
616      5.1578   libwebcore.so            jscyyparse（void*）
552      4.6220   libdvm.so                dalvik_inst
410      3.4330   libwebcore.so            JSC::Lexer::lex（void*,
void*）
...
```

kernel（no-vmlinux）については、本章を書いた段階では関数単位に分割することができず、すべての関数がひとつのモジュールとして扱われています。

おわりに

Androidのプロファイリング手法について説明しました。組込み機器のハードウェアの性能は日進月歩で向上しています。今回プロファイリングで実際に使用したADP1のCPUクロックは500MHzであり、これはひと昔前のPCに相当するレベルの性能です。しかし、組込み分野においてもソフトウェアは複雑化、巨大化し続ける傾向にあります。

ハードウェア性能の進歩とともに、ソフトウェアへの要求も高まり、扱うデータも巨大になってきています。結局のところ、パフォーマンスの問題はこれからも組込み開発の重要なファクターであり続けるでしょう。

今回の内容が、Android上でのプログラム開発の際の一助になれば幸いです。

第6章

Androidポーティング事例

株式会社 アックス
渡邊 昌之
Wtanabe Masayuki

AndroidはLinuxをベースにしているので、Linuxが動作するデバイスであれば、さまざまな機器にポーティングすることができます。

ここでは、特定のデバイス（組込み機器）にAndroidをポーティングする際に必要な最小限の作業と、具体的な事例として無線通信デバイス（Willcom W-SIM）を使用した通信を行うためのRIL（Radio Interface Layer）の扱いについて説明します。

CPUは、ARM系として必要最小限の動作のみに限定するので、マルチメディアデバイスや特定の周辺機器への対応は含んでいません。

Androidポーティングに必要なこと

まず、Androidそのものをポーティングするためには、SDKなどのアプリケーション開発環境はなくてもかまいません。開発には、LinuxやMacOS XなどのUnix系のOSが必要です。

検証されているLinuxディストリビューションは、Ubuntu（32ビット）ですので、新しい環境が用意できる場合は、はじめからUbuntuを入れると細かな問題につまずかなくて済みます。

手順の概要は以下の通りです。

・Ubuntuのインストール

http://www.ubuntu.com/getubuntu/download

からUbuntu 8.04 LTS Desktopをダウンロードしインストールします。

・ツール類のインストール

以下が必要となるので、これらをapt-getでインストールします。

```
git-core
gnupg,sun-java5-jdk
flex
bison
gperf
libsdl-dev
libesd0-dev
libwxgtk2.6-dev
build-essential
zip
curl
libncurses5-dev
zlib1g-dev
```

・repoのインストール

Androidのプロジェクトは、複数のGitプロジェクトをrepoというツールでまとめて管理しています。ホームディレクトリ下にbinディレクトリがなければbinを作成し、PATHを通して以下のコマンドでrepo自体をダウンロードします。

```
$ curl http://android.git.kernel.org/repo >~/bin/repo
```

~/bin/repoに実行属性をつけます。

・ソースのダウンロード

まず、適当なディレクトリを作成し、

```
$ mkdir mydroid
$ cd mydroid
```

以下のコマンドで、repoの環境を構築します。

```
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

repoを使ってソースツリーを取得します。

```
$repo sync
```

このコマンドは、ソースツリーの更新に使用するので、適宜必要に応じて実行することで手元のソースを最新に保つことができます。

途中、ネットワークのエラーやGitサーバのエラーで中断することがあるので、正しく終了していないようだったら、repo syncを再実行して最後まで実行されるのを確認します。

あとは、このディレクトリでmakeを実行すれば、Android一式がビルドされますが、Linuxカーネルやコンパイラも含んでいるので、初回のビルドは環境にもよりますが、数時間かかることもあります。outというディレクトリが作成され、Androidのイメージが作成されています。

しかし、この環境ではカーネルが目的としているデバイスに対応していませんので、このままでは目的とするデバイスで動作しません。次項から最小限のデバイスについて対応を行います。

最小限のドライバ

Androidで要求されるデバイスの要件は、以下の通りです。

・128MB以上のRAM、256MB 以上のFlash
・ファイルストレージ（SDカードなど）
・QVGA、16ビットカラー以上のディスプレイ（タッチパネルつきが理想）
・最小限のキー5個（application、power、camera、volume+、volume-）
・ネットワークデバイス
・無線通信デバイス
・デバッグ用のシリアルデバイス

これらの周辺デバイスは各機器によって異なるので、機器ごとのデバイスドライバを用意する必要があります。

最近のLinuxカーネルでは一般的な周辺デバイスのドライバはカーネルソースの中に組み込まれているので、コンフィグレーションするだけで使えることがほとんどです。

以上から、最小限必要なデバイスドライバは、

・キーボード（キーマップ）
・ディスプレイドライバ
・パワーマネージメントドライバ

となります。

●キーボード

フルキーボードとは別に最小限5個のキーを用意しなければなりません。Linuxの/dev/event0デバイスからの入力を使用してこれらのキーを認識します。そのため、デバイスについている特殊キーの押し下げ、解

放などのイベントをこのデバイス経由でアプリケーションに送らなければなりません。

それ以外に、フルキーボードもサポートしますが、こちらは接続すべきキーボードのレイアウトに合わせたレイアウトデータを作成して、通常のキーボードデバイスとして認識されるようにしておけば問題ありません。

●ディスプレイ

/dev/fb0か/dev/graphics/fb0というデバイスが、フレームバッファデバイスと想定されます。

そのためこのデバイスを正しく動作させてやる必要があります。

Androidでは、以下のようなフレームバッファを想定しています。

・RGB 5,6,5ビットの16ビットカラー
・連続したメモリで直接書き込みできる

Androidではダブルバッファで画面を描画するので、グラフィックメモリは1画面分のメモリの倍以上が必要です。これらの条件がクリアできていれば、ディスプレイドライバはいくつかのioctlを実装するだけで、比較的簡単に対応することができます。

●パワーマネージメント

Android上のアプリケーションの動作に合わせて電源を制御するために、オリジナルのパワーマネージメントドライバが用意されています。このドライバがポーティングすべき機器で動作するように調整する必要があります。

「試しに動かしてみるだけ」という目的なら、ioctlやread/writeがエラーを返さない程度に動作していれば、この操作で実際の電源操作ができなくても問題はありません。

●シリアルドライバ

これは、Androidを動作させるために必要なわけではありませんが、デバッグの際には必ず必要になります。そのため、最小限のシリアルドライバは動作するようにしておくことが必要です。

●無線通信デバイス

ここでは、Wi-Fiなどの無線LANデバイスではなく、GSM/CDMA/W-CDMA/W-SIMなどの無線通信（携帯電話）デバイスのことを指しています。元々Androidは無線端末（携帯電話など）のOSとして開発されているため、これらのデバイスを使用するための仕組みやアプリケーションが組み込まれています。

無線通信デバイスは、CPUとシリアルで通信し、コマンドのやりとりを行う仕様になっていることが多いので、ここではシリアルデバイスとして接続できるものとします（昨今はUSB経由での接続も増えてきたので、その場合はUSBドライバも必要になります）。

ここまでの説明では、カーネルはAndroidのツリーに含まれているものを使うことを前提に話を進めてきましたが、実際には、カーネルは2.6.25以上ならそれほど問題なく動作します。したがって、カーネルとユーザランドは評価ボードメーカの用意したものをそのまま持ってきて、最小限のデバイスドライバだけを組み込む方が簡単です。

以上のドライバの中で、キーボードやフレームバッファは一般によく使用され、特別なことは何もありません。ここでは無線通信デバイスを使用する事例について説明します。

RIL（Radio Interface Layer）の構造

無線デバイスを制御するためにRadio Interface Layer（RIL）という仕組みが使われています。RILは名前の通り、無線通信デバイスの制御を行うためのインタフェース層です。

上位のアプリケーションからは、アプリケーションフレームワーク経由で、rildを使って回線制御を行います。rildは、ベンダごとに異なるvendor-rilという共有ライブラリを使って、無線通信デバイスの制御を行います。RILではこのような階層構造で、各層での処理を抽象化しています。

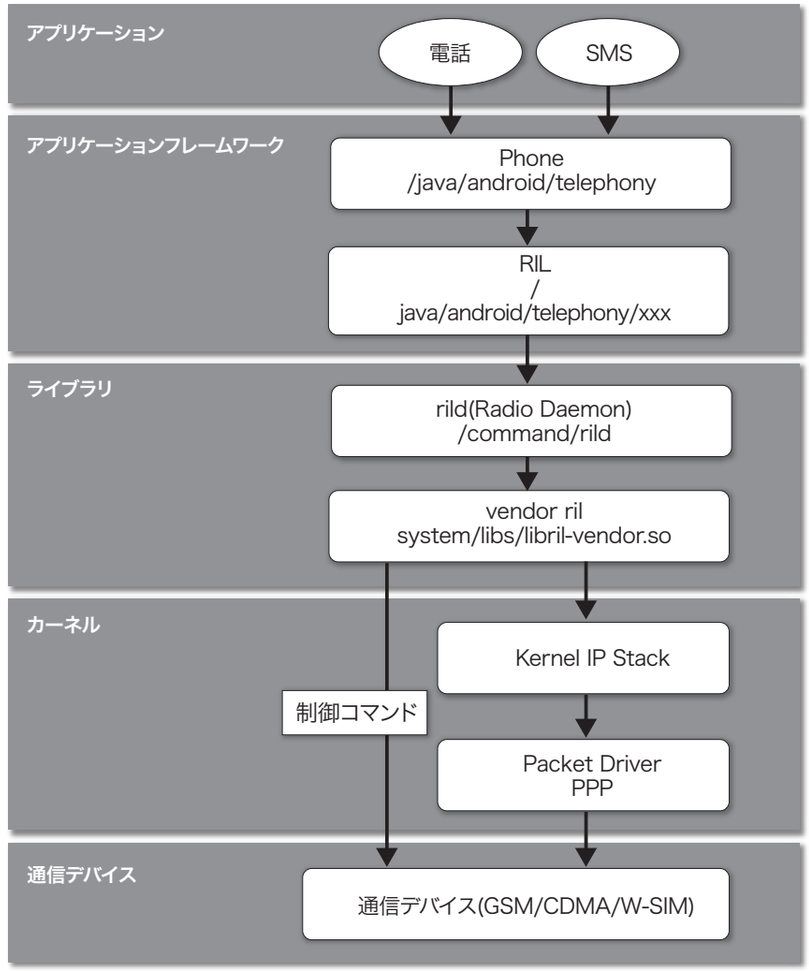
当然その目的は、RILで物理層の違いを吸収し、アプリケーションからはデバイスの差を気にしないで操作できることだったはずです。しかし、実際には、RILの構造がそのまま物理層（この場合は無線通信デバイス）の構造に依存してしまい、アプリケーション層にまでその影響が及んでいます。そのため、最初にAndroidが搭載された端末はGSM方式だったので、RILの構造（コマンド）自体がGSMの構造に強く依存しています。

このままでは、ほかの無線通信デバイスへのポーティングは簡単ではありません。現在のソースツリーにはcdma-importというブランチができていて、そちらの中には、GSMと並んでCDMAへの対応が始まっています。

すでにNTTドコモがHTCの端末が発売されたので、正式にCDMAに対応したソースも公開されるのではないでしょうか。

また、それと同時にアプリケーションやライブラリレベルで無線通信デバイスの方式に依存した部分が解消されると、他の通信デバイスへのポーティングも楽になると思います。

図1 RIL構造



rildとvendor-RIL関係

rildはデーモンプロセスとして動作し、アプリケーションからの制御を受けて必要な処理を行います。具体的には、rildが起動時に作成したUnix Domain Socketをアプリケーションで開き、プロセス間通信でコマンドを送信して処理を行います。

rildの仕事は、アプリケーション側からの要求をvendor-ril.soへ渡すことです。rild自体が何かの処理をするわけではなく、ソケットから受け取ったコマンドを解釈し、適当なvendor-rilの関数を呼び出します。

また、無線通信デバイスの制御は、CPUから見ると、多くの時間を要するのが普通なので、各リクエストはとりあえず最小限の処理だけを行って即座にリターンし、処理が完了した時点でonRequestCompleteを呼び出し、処理の完了をアプリケーションに通知するという仕組みになっています。この仕組みはとくに変更する必要もないので、ここではvendor-rilの内容について説明します。

ソースツリーにはvendor-ril.soのリファレンスとして、reference-rilが用意されているので、その内容について見てみます。

各コマンドはRequest codeで区別されます。現在用意されているRequest codeは、

RIL_REQUEST_GET_SIM_STATUS (1)

から始まって、

RIL_REQUEST_SET_LOCATION_UPDATES (76)

までの76個ですが、cdma-import ブランチを見ると、CDMA用に、

RIL_REQUEST_CDMA_SET_SUBSCRIPTION (77)

から

RIL_REQUEST_DEVICE_IDENTITY (102)

までが追加されています。

また、アプリケーションから要求していないレスポンス(UnsolicitedResponse)も存在します。

当然ですが、電話なので電話がかかってくることもあります。それはアプリケーション側から要求したことではなく、通信デバイス側から突然レスポンスが発生するので、それを受け止める仕組みが必要です。

それにもrequest codeが割り当てられていて、

RIL_UNSOL_RESPONSE_RADIO_STATE_CHANGED (1000)

から

RIL_UNSOL_CALL_RING (1018)

までの19個あります。加えて、

RIL_UNSOL_RESPONSE_SIM_STATUS_CHANGED (1019)

から

RIL_UNSOL_RESTRICTED_STATE_CHANGED (1023)

までCDM用に用意されているのですが、cdma-importでは別の名前で定義されている部分があるので、マージの際には注意が必要です。以前も、開発ブランチ(cupcake)をmasterにマージする際に大きな混乱が生じました。また同じようなことが起こらないことを祈るばかりです。

こうしてみると、かなり多数のコマンドを処理してやらなければならないように見えますが、電話の発着信、SMS(ショートメッセージ)の送受信だけなら、それほど多くのコマンドに対応してやる必要はありません。

最小限対応しなければならないコマンドは、

・RIL_REQUEST_GET_SIM_STATUS
・RIL_REQUEST_GET_CURRENT_CALLS
・RIL_REQUEST_DIAL
・RIL_REQUEST_HANGUP
・RIL_REQUEST_SIGNAL_STRENGTH
・RIL_REQUEST_OPERATOR
・RIL_REQUEST_SEND_SMS
・RIL_REQUEST_SMS_ACKNOWLEDGE
・RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED
・RIL_UNSOL_RESPONSE_NEW_SMS

になります。

もちろん、実際の製品化にはすべてのコマンドに正しく対応することが必要となりますが、その部分は前述のコマンドへの対応と、基本的には同様な作業の繰り返しなので、ここでは割愛します。

vendor-RIL(libril.so)作成の実際

vendor-rilは共有ライブラリとして作成され、rildから呼び出されます。前項で説明した各コマンドの実際の処理内容を記述することになります。

本来ならば、新たなライブラリを作成し、build/target/board/generic/system.prop内のrild.libpathにそのPATHを設定すべきなのですが、そのためにはコンパイル環境を整えたり、ファイルの追加をしたりとやらなければならないことがいろいろあります。

ここでは、すでに用意されているreference-rilのソースを編集してしまいます。hardware/ril/reference-rilにreference-ril.cというファイルがあり、rildはデフォルトでlibreference-ril.soを読み込むように設定されています。

例として、WillcomのW-SIMというPHSの通信モジュールを使った通信への対応について説明します。ただし、具体的な仕様は公開されていないので、詳細は記載できません。詳細な資料が必要な方は、Willcom Core Module Forum(WCMF:http://www.wcmf.jp)に加入して資料を入手してください。

各コマンドでは、以下のような処理を行う必要があります。

・RIL_REQUEST_GET_SIM_STATUS

現在設定されているSIMカードの状態を返すコマンドです。本来は、SIMスロットの状態を確認してその結果を返すべきなのですが、テストなら必ず「正常」と返してかまいません。問答無用でRIL_SIM_READYを返してしまいましょう。

・RIL_REQUEST_GET_CURRENT_CALLS

これは現在接続している回線の情報を返すコールです。GSMでは、複数の回線を接続したまま切り替えるということができるので、こういうコールが用意されているようです。

このコマンドは、現在通話中かどうかの判定にも使われているので、つねに0を返す訳にはいきません。現在通話しているかどうかを内部的に保持し、その値を返すようにします。

・RIL_REQUEST_DIAL

単純に電話をかける処理を行います。現在通話中の回線がなければ、無線通信デバイスに発信のコマンドを発行するだけです。

一般的なATコマンドが実装されていれば、単純にATDコマンドを実行するだけです。ATコマンドを送信するライブラリはすでに用意されており、GSMでも発信はATDなのでとくに変更することはありません。正しく接続できれば、内部で管理している通話中の回線数を増やします。

・RIL_REQUEST_HANGUP

GSMでは、HANGUPでいったん保留する処理を行うのですが、W-SIMでは保留ができないので、ATHコマンドで回線を切断し、内部で管理している通話中の回線数を0にします。

・RIL_REQUEST_SIGNAL_STRENGTH

無線通信デバイスは、信号強度を取得する必要があります。これは画面上アンテナアイコンで表示されるものです。無線通信デバイスから簡単に信号強度を取得できるのであれば、それを返せばいいのですが、ここでは必ず同じ値を返してしまいます。

・RIL_REQUEST_OPERATOR

現在使用している携帯電話キャリアの名称を返します。本来は、無線通信デバイスにアクセスし現在選択しているキャリアの名称を返す必要がありますが、複数のキャリアに対応したデバイスでなければ固定値でかまわないので、固定値を返します。

・RIL_REQUEST_SEND_SMS

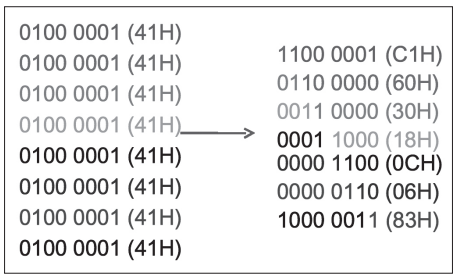
ショートメッセージを送信します。GSMでは、ショートメッセージは半角英数のみにしか対応していません。そのため、ショートメッセージの内容は1文字7ビットで構成し、それを8ビットに詰め込む形で送られてきます。

たとえば、AAAAAAAAの8文字を送った場合Aの文字コードはASCIIで0x41なので、16進数で、41 41 41 41 41 41 41 41の8バイトになります。

しかし、実際にreference-rilに送られてくるデータは16進数でC1 60 30 18 0C 06 83の7バイトとなっていました。

2進数で説明すると図のようになります。

図2　コード変換



この変換はフレームワークで行われていました。

rild以下で通信方式の違いを吸収するのであれば、rildには変換していない文字列を送り、通信方式に依存したコードの変換はreference-rilで行うべきだと思います。しかし、SMS以外にも通信方式の違いがアプリケーションやフレームワークに波及してしまうのは避けられないので、フレームワークで行っているのでしょう。

変更点を減らすためには、reference-rilでGSM方式の文字列をW-SIM方式の文字列に変換するように実装します。

・RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED

回線状態が変化したときに発生します。無線通信デバイス側から、RING、BUSY、NO CARRIERなどが帰ってきたとき、つまりダイアルしたけれど接続できなかったときや電話がかかってきたときに発生します。

これがないと電話の着信ができないので、サポートしなければなりませんが、RINGという文字列はほとんどのデバイスで同じですので、着信を認識することだけに関してはとくに変更することはありません。

着信があると、アプリケーションは前述のRIL_REQUEST_GET_CURRENT_CALLSを使用して待ち状態の回線があることを認識するので、内部の回線状態を正しく更新しておきます。アプリケーションはRIL_REQUEST_ANSWERで電話を取ります。

・RIL_UNSOL_RESPONSE_NEW_SMS

SMSのメッセージが到着した時実行されます。SMSの内容は前述の文字コード変換の逆変換を行ってアプリケーションに渡します。

WilcomのW-SIMへの対応について書いてきましたが、これらの対応は他のキャリアでも同様に必要となると思います。とくに、W-SIMのようにGSMとは回線管理の方式自体が異なるようなキャリアに対応するためには、上位のアプリケーション、フレームワークを含めた対応が必要になります。

たとえば、W-SIMでは無線強度が6段階あるのですが、GSM方式では4段階しかありません。したがって、アンテナのアイコンが違ったり、通話中に保留ボタンが表示されていても、W-SIMでは保留して別の電話に出ることができないなど、とくにSMSで日本語を処理するためには、フレームワークの変更が必要になります。

今後もっと多くのデバイスに実装され、いろいろなデバイスで簡単に使えるようになれば、これまでGUIの貧弱さゆえに敬遠されてきたUnix系のフリーOSが、WindowsCEなどの市場に進出していくことができるかもしれません。組