Connected, Limited Device Configuration

仕様 バージョン 1.0

Java 2 Platform Micro Edition

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 U.S.A.

CLDC 1.0
2000 年 7 月 31 日

目次

図目次

はじめに

この仕様書『Connected, Limited Device Configuration』では、Java™2 Platform Micro Edition (J2ME™) の Connected, Limited Device Configuration (CLDC) を定義します。

J2ME の構成で規定するのは、サポートされる Java プログラミング言語機能のサブセット、その構成の Java Virtual Machine の機能のサブセット、ネットワーク、セキュリティ、インストール、およびその他のサポートされるコアプラットフォームなどです。これらはすべて、組み込みコンシューマ製品の特定のグループをサポートするために規定されます。

Connected, Limited Device Configuration は、1 つまたは複数のプロファイルの基礎となります。J2ME のプロファイルは、特定の垂直市場、デバイスカテゴリまたはデバイス業界のための API および機能の付加的なセットを定義します。構成およびプロファイルは、関連マニュアルの『Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)』(Sun Microsystems, Inc.) でより明確に定義されています。

対象読者

この仕様書が対象とする読者は次のとおりです。この構成を定義する Java Community Process (JCP) のエキスパートグループ (JSR-30)、小規模で、Java 対応デバイスを構築しようとするデバイスメーカー、および小規模で、リソースに制約のある接続デバイスのための Java アプリケーションを作成するコンテンツ開発者などです。

内容の紹介

この仕様書は、次のように構成されています。

第 1 章「概要と背景」では、CLDC 仕様書の内容を紹介し、この仕様書の制定の仕事に携わっている企業名のリストを掲載します。

第 2 章「目的、要件、および適用範囲」では、この仕様書の目的、特別要件、および適用範囲を定義します。

第 3 章「アーキテクチャとセキュリティの概要」では、CLDC アーキテクチャの概要を定義し、そのセキュリティ機能について説明します。

第 4 章「Java 言語仕様への準拠」では、CLDC 構成が必要とする標準 Java プログラミング言語からの変更点を定義します。

第 5 章「Java Virtual Machine 仕様への準拠」では、CLDC 構成が必要とする標準 Java Virtual Machine からの変更点を定義します。

第 6 章「CLDC ライブラリ」では、CLDC 構成が必要とする固有の Java API を定義します。

付録では、Java 2 Micro Edition と KVM を紹介しています。

関連マニュアル

『The Java™ Language Specification』、James Gosling、Bill Joy、Guy L. Steele 著、Addison-Wesley 発行、(1996 年、ISBN 0-201-63451-1)

『The Java™ Virtual Machine Specification (Java Series) 第 2 版』、Tim Lindholm、Frank Yellin 著、Addison-Wesley 発行、(1999 年、ISBN 0-201-43294-3)

『Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)』、Sun Microsystems, Inc. 発行

『Java™ 2 Platform: Micro Edition, A White Paper』、Sun Microsystems, Inc. 発行

『The K Virtual Machine (KVM), A White Paper』、Sun Microsystems, Inc. 発行

第1章

概要と背景

この文書は、Java 2 Platform、Micro Edition (J2ME) の Connected, Limited Device Configuration (CLDC) に関する仕様書です。この文書の目的は、最小サイズの標準 Java プラットフォームを、小型でリソースに制約のある次のような特性を持つ接続デバイス用に定義することです。

- Java プラットフォーム用に 160K バイトから 512K バイトの合計メモリ (4 ページの「ハードウェア必要条件」を参照)
- 16 ビットまたは 32 ビットプロセッサ
- 低電力消費、電池でも動作する
- 特定のネットワークへの接続性。無線、断続的な接続、帯域幅の制限がある (多くの場合 9600 bps 以下)

この仕様でサポートされるデバイスの例としては、携帯電話、双方向ポケットベル、携帯情報端末 (PDA)、オーガナイザ、家電機器、POS 端末などがあります。

この J2ME 構成仕様書は、小型の接続デバイス用の Java 技術構成要素およびライブラリに必要最小限な補助機能を定義します。この仕様書が取り扱う主要な項目は、Java 言語および Virtual Machine の機能、コアライブラリ、入出力、ネットワーク、およびセキュリティについてです。

多数の業界パートナーで構成される Java Community Process (JCP) エキスパートグループ JSR-30 の活動の成果として、この仕様書が作成されました。CLDC の定義の設定に際し以下の企業にご協力いただきました (アルファベット順)。

- America Online
- Bull
- Ericsson
- Fujitsu
- Matsushita

- Mitsubishi
- Motorola
- Nokia
- NTT DoCoMo
- Oracle
- Palm Computing
- RIM
- Samsung
- Sharp
- Siemens
- Sony
- Sun Microsystems
- Symbian

CLDC は、1 つまたは複数のプロファイルの基礎として利用されるコア技術です。J2ME プロファイルは、特定の垂直市場、デバイスカテゴリーまたはデバイス業界のための、より包括的で集約的な Java プラットフォームを定義します。

J2ME 構成およびプロファイルに関する正確な情報については、『Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)』(Sun Microsystems, Inc.) を参照してください。Java 2 Micro Edition の構成およびプロファイルに関する一部の情報は、付録 A にあります。この付録には、Connected, Limited Device Configuration を実行するために利用できる Java Virtual Machine である KVM に関する情報が掲載されています。

第2章

目的、要件、および適用範囲

目的

Connected, Limited Device Configuration (CLDC) の目的は、小型でリソースに制約のある接続デバイスのための、最小サイズの Java プラットフォームを定義することです。CLDC の対象デバイスには、一般的に次のような特性があります。

- Java プラットフォーム用に 160K バイトから 512K バイトの合計メモリ (4 ページの「ハードウェア必要条件」を参照)
- 16 ビットまたは 32 ビットプロセッサ
- 低電力消費、電池でも動作する
- 特定のネットワークへの接続性。無線、断続的な接続、帯域幅の制限がある (多くの場合 9600 bps 以下)

通常これらのデバイスは、大量に (数十万あるいは数百万台) 製造されます。つまり、メーカーはきわめてコストに敏感で、1 台あたりの製造コストをできるだけ抑えたいと考えています。

Java アプリケーションとコンテンツの動的配信

CLDC エキスパートグループからのフィードバックに基づいた、小型デバイス空間における Java 技術の最大の長所の 1 つ は、対話的コンテンツとアプリケーションを異種ネットワークを通して小型のクライアントデバイスに動的、かつ安全に配信することです。ハードコードされた機能セットが携帯電話やポケットベルのような小規模デバイスに付属していた従来の方法とは異なり、デバイスメーカーは、サードパーティのコンテンツプロバイダや開発者により開発された内容豊富で動的かつ対話的なコン

テンツをサポートするための拡張可能なデバイスを構築できるソリューションを求めています。最近のインターネット対応の携帯電話、通信装置、ポケットベルなどの導入により、このような移行はすでに進行中です。この CLDC 仕様書の主要目的の 1 つは、これらの拡張可能な次世代デバイス用に、動的なコンテンツを安全に配信するための標準プラットフォームとして Java プログラミング言語を利用できるようにすることにより、この移行をさらに推進することです。

サードパーティアプリケーション開発者を対象とする

この仕様書で動的に配信される Java アプリケーションに焦点をおいているのは、ハードウェアメーカーとそのシステムプログラマだけでなく、サードパーティアプリケーションの開発者も対象としているためです。実際、ひとたび小型の Java 対応デバイスが一般的になれば、これらのデバイスのアプリケーション開発者の大多数として、デバイスメーカー自身もサードパーティ開発者となるであろうということがこの仕様書の前提となっています。

この方向性は、この仕様書に含まれるべき Java プラットフォームの機能および API にとってある意味を持ちます。第一に、この仕様書は、サードパーティアプリケーション開発者が十分にプログラム開発機能を利用できる高度なライブラリのみを含む必要があります。たとえば、CLDC に含まれるべきネットワーク API は、プログラマに特定のネットワーク伝送プロトコルの詳細について知ることを要求するのではなく、たとえばファイル、アプリケーション、または Web ページ全体を一度に転送する機能などの、意味のある、高度な抽象的機能をプログラマに提供する必要があります。第二に、一般性と移植性の重要さが強調されています。CLDC 仕様書は、特定のデバイスカテゴリーまたは垂直市場にのみ焦点を合わせるべきではありません。

必要条件

ハードウェア必要条件

CLDC は、広範囲にわたる小型デバイス上で実行されるように設計されています。そのようなデバイスには、携帯電話、双方向ポケットベルなどの無線通信デバイスから、電子手帳、POS 端末、さらに家電機器までも含まれます。これらのデバイスの実際のハードウェア機能は著しく多岐にわたるため、CLDC 仕様書はメモリ要件以外には特定のハードウェア条件を課しません。

この仕様書では、Virtual Machine、構成ライブラリ、プロファイルライブラリ、およびアプリケーションは、合計で 160K から 512K バイトのメモリ内に収まるように規定しています。詳細は、次のとおりです。

- 128K バイトの不揮発性メモリ[1] が Java Virtual Machine および CLDC ライブラリ用に利用できること
- 最低でも 32K バイトの揮発性メモリ[2] が Java ランタイムおよびオブジェクトメモリに利用できること

メモリ全体における揮発性メモリと不揮発性メモリの割合は、対象となるデバイスおよびデバイス内の Jave プラットフォームの役割りに大きく依存します。Java プラットフォームがデバイス内に構築されたシステムアプリケーションを実行するためにだけ使用されるのなら、アプリケーションはプリリンクが可能で、非常に限られた量の揮発性メモリだけが必要になります。Java プラットフォームが動的にダウンロードされたコンテンツの実行に使用される場合は、デバイスに必要な揮発性メモリの割合は高くなります。

ソフトウェアの必要条件

ハードウェア機能と同様に、CLDC デバイスのシステムソフトウェアには多様なものがあります。たとえば、一部のデバイスでは、複数の並列オペレーティングシステムプロセスと階層型ファイルシステムをサポートしている本格的なオペレーティングシステムを装備している場合があります。その他の多数のデバイスは、ファイルシステムの概念を持たない、非常に限られたシステムソフトウェアを備えています。このような多様性に対して、CLDC には CLDC デバイスが利用可能なシステムソフトウェアに関して最小限の前提条件があります。

この仕様書では、小型のホストオペレーティングシステム (9 ページの「Virtual Machine 環境」を参照) またはカーネルが、構成するハードウェアを管理するために利用可能である、ということを仮定しています。このホストオペレーティングシステムは、Jave Virtual Machine を実行するために、少なくとも 1 つのスケジュール可能な項目を提供しなければなりません。ホストオペレーティングシステムは、異なるアドレス空間やプロセスをサポートする必要はなく、リアルタイムのスケジューリングや遅延動作を保証する必要もありません。

1. 不揮発性という用語は、ユーザがデバイスの電源をいったん切っても、メモリには記憶内容が保持されるということを示すために使用されます。この仕様書では、不揮発性メモリは通常読み取りモードでアクセスされ、かつそれに書き込むためには特別のセットアップが必要になる、ということが仮定されています。不揮発性メモリの例としては、ROM、フラッシュメモリおよびバッテリバックの SDRAM があります。実際のメモリ技術については、この仕様書の範囲外です。
2. 揮発性という用語は、ユーザがデバイスの電源を一度切ると、メモリには記憶内容は保持されないということを示すために使用されます。この仕様書では、揮発性メモリは特別のセットアップを必要とせず、直接に読み書きでる、ということが仮定されています。最も一般的な揮発性メモリとしては、DRAM があります。

J2ME の必要条件

CLDC は、Java 2 Micro Edition (J2ME) の構成として定義されています。これは、CLDC 仕様書にとっては重要な意味があります。

- J2ME の構成は、最小限の補助機能、つまり Java 技術の共通機能のみを定義しています。1 つの構成に含まれるすべての機能は、一般に多様なデバイスに適用可能でなければなりません。特定の垂直市場、デバイスカテゴリーまたは業界に固有の機能は、CLDC にではなく、プロファイルに定義するべきです。つまり、CLDC の扱う範囲には制限があり、一般的にはプロファイルによって補う必要があるということです。
- 構成の目的は、種々のリソースに制約のあるデバイス間での移植性や相互運用性を保証することにあるので、特定のオプションの機能を定義すべきではありません。この制限により、構成に含むことができるものと含むべきでないものが明確に区別されます。ドメイン固有の機能については CLDC ではなくプロファイルに定義されなければなりません。

J2ME 構成およびプロファイルを定義するための規則およびガイドラインの詳細については、『Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)』(Sun Microsystems, Inc. 発行) を参照してください。

CLDC にオプションの機能がなくても、様々な実装レベルで最適な条件で利用できるということに注意してください。たとえば、実装レベルでは、識別可能なユーザレベルでの実装の意味が CLDC 仕様書が定義するものと同じであるかぎり、代りの実行手法 (たとえば、JIT コンパイル) またはクラス表現手法が利用できます。

適用範囲

JSR-30 エキスパートグループの決定に基づき、この CLDC 仕様書では、次の領域を取り扱います。

- Java 言語および Virtual Machine の機能
- コア Java ライブラリ (`java.lang.*`、`java.util.*`)
- 入出力
- ネットワーク
- セキュリティ
- 国際化

この CLDC 仕様書では次の機能は扱いません。

- アプリケーションの有効期間の管理 (アプリケーションのインストール、起動、および削除)
- ユーザインタフェースの機能性
- イベント処理
- 高レベルアプリケーションモデル (ユーザとアプリケーション間の対話)

これらの機能は、CLDC 上に実装されたプロファイルによって扱うことができます。

CLDC エキスパートグループは、CLDC が扱う領域の数を意図的に少なくしようとしています。メモリ制限の範囲内で処理するための、あるいは特定のデバイスカテゴリーを除外するためには、CLDC の扱う範囲を限定する必要があります。この CLDC 仕様書の将来のバージョンでは新しい領域を扱う可能性はあります。

この仕様書の残りの部分は、次のような構成になっています。最初は、典型的な CLDC 環境のアーキテクチャについて説明します。その後、Java 言語および CLDC をサポートしている Java Virtual Machine (JVM) の Virtual Machine 機能と従来の Java 環境とを比較します。最後に CLDC に含まれる Java ライブラリの詳細について説明します。

第3章

アーキテクチャとセキュリティの概要

この章では、典型的な CLDC 環境のアーキテクチャについて説明します。ここでの説明は、後半の章で取り上げるより詳細な仕様説明の導入部分となります

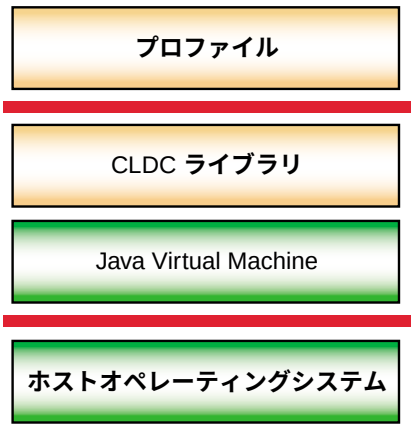

図 3-1　アーキテクチャの概要

Virtual Machine 環境

典型的な CLDC デバイスのアーキテクチャの概要を図 3−1 に示します。CLDC 実装の中心にあるのは Java Virtual Machine で、この仕様書の後半部で定義されている特定の相違点を除けば、Java Virtual Machine 仕様および Java 言語仕様に準拠しています。Virtual Machine は、一般に CLDC 仕様の範囲外であるホストオペレーティングシステム上で動作します。

Virtual Machine の上位には、Java ライブラリがあります。これらのライブラリは、2 つのカテゴリーに分類されます。

1. Connected, Limited, Device Configuration(CLDC) により定義されたライブラリ
2. プロファイルにより定義されたライブラリ

Java アプリケーションの概念

この仕様書では「Java アプリケーション」という用語は、アプリケーションの起動位置を特定する、1 つの固有のメソッド `main` を含む Java クラスファイルの集合を指します。『Java Virtual Machine Specification (JVMS)』のセクション 5.2 と 2.17.1 に規定されているように、メソッド `main` は次に示すように `public`、`static` および `void` を宣言しなければなりません。

```
public static void main(String[] args)
```

CLDC をサポートする JVM は、メソッド `main` を呼出すことにより Java アプリケーションの実行を開始します。

アプリケーション管理

小型でリソースに制約のあるデバイスには、ファイルシステムや、その他の動的にダウンロードされた情報をデバイス上に格納するための標準機構を持っていないものが多くあります。したがって、CLDC の実装は、外部ソースからダウンロードされた Java クラスがデバイス上に恒久的に格納されることを要求するこはできません。むしろ、Virtual Machine ではただ単にクラスファイルをロードし、その後それらを廃棄することがあります。

ただし、多くの CLDC デバイスにおいては、アプリケーションを何度もダウンロードしなくても、同じ Java アプリケーションを数度実行できれば有益です。これは、アプリケーションが無線ネットワーク経由でダウンロードされている場合は特に重要です。その場合ユーザには高いダウンロードの費用がかかることになるからです。

ここでは、CLDC を実装するデバイスは、そのデバイスに格納されている Java アプリケーションを管理する機能を持っていると仮定します。アプリケーション管理とは、一般的には次の機能を意味します。

- デバイス上に格納されている既存の Java アプリケーションを検査すること
- Java アプリケーションを選択、起動すること
- 既存の Java アプリケーションを (可能ならば) 削除すること

CLDC デバイス間の重要な違いや機能上の相違のために、アプリケーション管理の詳細は、デバイスに固有で、実装により異なります。したがって、アプリケーション管理機能は C プログラミング言語、またはその他のホストオペレーティングシステムに固有の低レベルプログラミング言語によって書かれていることが多くあります。実際のアプリケーション管理の詳細は CLDC 仕様書では扱っていません。

セキュリティ

Java プラットフォームで今後に期待されている機能は、対話的コンテンツおよびアプリケーションを、多種多様なネットワークを通してクライアントデバイスに安全な方法で動的に配信する機能です。

残念ながら、Java 2 Standard Edition のセキュリティに使用されたコードの合計量は、CLDC をサポートする Java Virtual Machine に用意されたメモリを遥かに超えています。したがって、CLDC のセキュリティモデルを定義するときには、ある程度の妥協が必要です。CLDC のセキュリティモデルを定義するための一般的なガイドラインとしては、すべてを単純化して、この CLDC 仕様書の後のバージョンで、より包括的なセキュリティソリューションを考慮することが必要です。

この仕様書が重要視しているのは、次の領域です。

1. 低レベル Virtual Machine のセキュリティ
2. アプリケーションレベルのセキュリティ

低レベル Virtual Machine のセキュリティ

この仕様書では、低レベル Virtual Machine のセキュリティとは、Virtual Machine が実行する Java アプリケーションは、実行しているデバイスに悪影響を与えてはならない、ということを意味します。標準 Java Virtual Machine の実装では、この制約は Java クラスファイルベリファイアによって保証されます。この Java クラスファイルベリファイアは、Java クラスファイルに格納されている Java バイトコードとその他の項目が不正なメモリ位置や Java オブジェクトメモリ (Java ヒープ) 外にあるメモリ

領域への参照を持つことができないことを保証します。クラスファイルベリファイアの役割は、Virtual Machine にロードされたクラスファイルは 『Java Virtual Machine 仕様』で許可されていない方法では実行されない、ということを保証することです。

21 ページの「クラスファイルの検証」の節で詳細に説明しますが、この CLDC 仕様書では、CLDC をサポートする JVM は不正なクラスファイルの処理をしないということを規定しています。これは、21 ページの「デバイス外の事前検証とスタックマップによる実行時検証」で定義している検証技術により保証することができます。

アプリケーションレベルのセキュリティ

標準の Java 環境においても、クラスファイルのベリファイアが提供するセキュリティには制限があります。このベリファイアは、指定されたプログラムが有効な Java プログラムであることを保証できるだけであり、それ以上のことはできません。他にもベリファイアが発見することのできないセキュリティを脅かす可能性のある要因があります。たとえば、ファイルシステム、プリンタ、赤外線デバイス、またはネットワークなどのような外部リソースへのアクセスはベリファイアでは検証できません。

Java アプリケーションから外部リソースへの規制されたアクセスを可能にするために、J2SE つまり J2EE 環境はセキュリティマネージャの概念を提供しています。Java アプリケーションまたは Java ランタイムシステムのさまざまな部分が保護されたリソースにアクセスする必要が起きたときはいつでも、セキュリティマネージャが呼び出されます。J2SE は、アクセス権、アクセスコントローラ、およびセキュリティポリシーの正式な概念で構成される包括的なモデルを提供します。

残念ながら J2SE のセキュリティモデルは、合計でわずか数 100 Kバイトのメモリしか使用できない CLDC デバイスに採用するにはメモリを消費しすぎます。したがって、もっと単純なソリューションが必要になります。

サンドボックスモデル

CLDC をサポートする JVM は単純な「サンドボックス」セキュリティモデルを提供します。サンドボックスが意味するところは、Java アプリケーションは閉じられた環境で実行されなければならない、ということです。その閉じられた環境では、アプリケーションは、デバイスがサポートする構成、プロファイル、および使用権取得者のオープンクラスにより定義された API のみにアクセスできます。

具体的には、サンドボックスには、次の意味があります。

- Java クラスファイルが正しく検証され、有効な Java アプリケーションであることが保証されている。(21 ページの「クラスファイルの検証」を参照)
- CLDC、プロファイル、および使用権取得者のオープンクラスで定義されているように、Java API が制限され、事前に定義されたセットのみをアプリケーションプログラマが利用可能
- デバイス上の Java アプリケーションのダウンロードおよび管理は、Virtual Machine 内で、ネイティブコードレベルで実行され、ユーザが定義可能なクラスローダは提供されない。このため、プログラマが Virtual Machine 標準のクラスローディング機構をオーバーライドすることを防ぐことができる
- Virtual Machine がアクセス可能なネイティブ機能のセットは閉じている。つまり、アプリケーションプログラマは、ネイティブな機能を含む新しいライブラリをダウンロードしたり、あるいは CLDC、プロファイル、および使用権取得者のオープンクラスが提供する Java ライブラリの一部ではないネイティブな機能にアクセスできない

J2ME プロファイルがさらにセキュリティソリューションを提供する場合があります。プロファイルは、アプリケーションプログラマが他のどのような API を利用できるかも定義します。

システムクラスの保護

CLDC の中心となる必要条件の 1 つは、Java アプリケーションの Virtual Machine への動的なダウンロードをサポートする機能です。ダウンロードされたアプリケーションが、パッケージの `java.*`、`javax.microedition.*`、あるいは他のプロファイルまたはシステムに固有のパッケージが提供するシステムクラスをオーバーライドする可能性がある場合には、Java Virtual Machine のセキュリティホールが明確になります。CLDC の実装は、プログラマがこのような保護されたシステムパッケージ内のクラスを、どのような方法にしろオーバーライドできないことを保証しなければなりません。このような実装レベルでは、実装がシステムクラスのプリロード / プリリンクをサポートするか否かによって、異った方法で保証されます (28 ページの「クラスファイルの形式とクラスのロード」を参照)。1 つのソリューションとしては、クラスファイルの検索を実行するときは、常にシステムクラスを最初に検索するようにすることです。またセキュリティ上の理由から、アプリケーションプログラマがクラスファイルの検索順序をいかなる方法でも操作できないようにすることが必要です。クラスファイルの検索順序は、29 ページの「クラスファイルの検索順序」の節で詳細に説明しています。

同時に実行される複数の Java アプリケーションのサポート

デバイス上で利用できるリソースにより異なりますが、1 つの CLDC システムでは複数の Java アプリケーションを並行して実行できます。または 1 度に 1 つの Java アプリケーションの実行しかできないようにシステムを制御できます。実際のホストオペレーティングシステムのマルチタスク機能 (使用できる場合) を利用して、あるいは、Java アプリケーションを並列的に実行するために複数の論理 Virtual Machine をインスタンス化することによって、複数の Java アプリケーションの実行をサポートするかどうかは、各 CLDC 実装に依存します。

その他のセキュリティ領域

CLDC をサポートするデバイスは、通常、無線ネットワークまたは支払端末ネットワークのような終端間のソリューションの一部になっています。これらのネットワークは、サーバマシンとクライアントデバイス間のデータおよびコードの安全な配信を保証するために、多数の拡張セキュリティソリューションを必要とします。これら終端間のセキュリティソリューションは、すべて実装に依存し、この CLDC 仕様書では扱いません。

第4章

Java 言語仕様への準拠

CLDC をサポートする JVM の一般的な目的は、第 2 章で定義している使用可能なメモリの制限内で Java 言語仕様に準拠することです。この章では、CLDC をサポートする JVM と J2SE Java Virtual Machine の間の相違について簡単に説明します。ここに指摘していない相違については、CLDC をサポートする JVM では、『The Java™ Language Specification』James Gosling、Bill Joy、および Guy L. Steele 著、Addison-Wesley 1996 年発行、(ISBN 0-201-63451-1) の第 1 章から第 17 章までに記述されている規定との互換性が必要です。

浮動小数点をサポートしない

完全な Java 言語仕様書とこの CLDC 仕様書間の言語レベルの主な相違は、CLDC をサポートする JVM では浮動小数点をサポートしていない、ということです。大部分の CLDC が対象とするデバイスでは浮動小数点計算を行うためのハードウェアを搭載していません。また、ソフトウェアによる浮動小数点のサポートにはコストがかかりすぎると考えられるため、浮動小数点はサポートされなくなりました。

注 – この仕様書の以降の記述では、Java 言語仕様書を JLS とします。Java 言語仕様書のセクション番号は、記号の § を使用して表記します。たとえば (JLS § 4.2.4) のように表記します。

具体的には、CLDC をサポートする JVM は、浮動小数点のリテラル (JLS §3.10.2)、浮動小数点の型および値 (JLS §4.2.3)、および浮動小数点操作 (JLS §4.2.4) を使用すべきではない、ということです。詳細については、この CLDC 仕様書の 17 ページの「浮動小数点をサポートしない」を参照してください。

ファイナライズをサポートしない

CLDC ライブラリには `Object.finalize()` メソッドが存在しないので、CLDC をサポートする JVM は、クラスインスタンスのファイナライズ (JLS §12.6) をサポートすべきではありません。CLDC をサポートする JVM 上に作成されるアプリケーションには、ファイナライズの利用を要求すべきではありません。

エラー取り扱い上の制限

一般に、CLDC をサポートする JVM は、JSL の第 11 章で定義されている例外処理をサポートする必要があります。ただし、CLDC ライブラリに含まれるエラークラスのセットは制限されているため、CLDC のエラー処理機能にも制限があります。これには次の 2 つの理由があります。

1) 組み込みシステムにおけるエラー状態からの復旧は、通常はデバイスに固有の方法で行われます。重大なエラーから復旧しようと試みる組み込みデバイスもありますが、多くの組み込みデバイスではエラーに遭遇した際、単にそれ自体にソフトリセットをかけるだけです。アプリケーションプログラマがデバイス固有のエラー処理機構や規約に関わることはありません。

2) JLS §11.5.2 に規定してあるように、`java.lang.Error` クラスおよびそのサブクラスは、通常のプログラムが普通の方法で復旧することが期待できないような例外クラスです。Java 言語仕様書に完全に準拠したエラー処理を実装することは、むしろ費用が高くなります。エラークラスの存在をすべて把握し、さらにそれらの処理をすると、CLDC 対象のデバイスにメモリ制約がある実装に対して著しいオーバヘッドを課すことになります。

CLDC をサポートする JVM は、J2SE から継承された、34 ページの「J2SE から継承されたクラス」に定義されているエラークラスの限定されたセットをサポートしなければなりません。その他のエラーに遭遇した場合、実装はそのエラーをデバイスに適当な方法で処理する必要があります。

第5章

Java Virtual Machine 仕様への準拠

CLDC をサポートする JVM の一般的な目的は、第 2 章で定義している使用可能なメモリ制限内で可能なかぎり Java Virtual Machine 仕様書に準拠することです。この章では、CLDC をサポートする JVM と J2SE Java Virtual Machine の間に存在する相違について簡単に説明します。ここに指摘している相違以外については、CLDC をサポートする JVM では、『Java™ Virtual Machine Specification (Java Series)』第 2 版、Tim Lindholm および Frank YellinJames 著、Addison-Wesley 1999 年発行、(ISBN 0-201-43294-3) に記述されている Java Virtual Machine との互換性が必要です。

注 – この仕様書の以降の記述では、Java Virtual Machine 仕様書を JVMS とします。Java Virtual Machine 仕様書のセクション番号は、記号の § を使用して表記します。たとえば (JVMS §2.4.3) のように表記します。

浮動小数点をサポートしない

CLDC をサポートする JVM は、浮動小数点をサポートしていません。大部分の CLDC が対象とするデバイスでは浮動小数点計算を行うためのハードウェアを搭載していません。また、ソフトウェアによる浮動小数点のサポートにコストがかかりすぎると考えられるため、浮動小数点はサポートされなくなりました。そのため、CLDC をサポートする JVM では、次のバイトコードをサポートすべきではありません。

定数

`fconst_0`、`fconst_1`、`fconst_2`、`dconst_0`、`dconst_1`

ロード

`fload`、`fload_x`、`dload`、`dload_x`

ストア

`fstore`、`fstore_x`、`dstore`、`dstore_x`

配列

`faload`、`daload`、`fastore`、`dastore`、`newarray T_DOUBLE`、`newarray T_FLOAT`

算術

`fadd`、`dadd`、`fsub`、`dsub`、`fmul`、`dmul`、`fdiv`、`ddiv`、`frem`、`drem`、`fneg`、`dneg`、`fcmpl`、`fcmpg`、`dcmpl`、`dcmpg`

変換

`i2f`、`f2i`、`i2d`、`d2i`、`l2f`、`l2d`、`f2l`、`d2l`、`f2d`、`d2f`

戻り値

`freturn`、`dreturn`

CLDC をサポートする JVM 上で動作するユーザ独自のクラスとメソッドはすべて、次の制限を満たさなければなりません。

- いかなるメソッドも上記の禁止されたバイトコードを使用しない
- いかなるフィールドも型として `float`、`double`、またはそれらの型の配列を持たない
- いかなるメソッドも引数または戻り値として、型が `float` または `double`、あるいは要素の型が `float` または `double` である配列を持たない
- いかなる定数プール項目も `CONSTANT_Float` 型または `CONSTANT_Double` 型ではない
- いかなる定数プール項目もクラスの型が `float` または `double` の配列である `CONSTANT_Class` 型ではない

浮動小数点がサポートされていないために、次に示す『Java Virtual Machine Specification (JVMS)』のセクションおよびサブセクションである§2.4.3、§2.4.4、§2.18、§3.3.2 および §3.8 は、CLDC をサポートする JVM に適用できません。さらに、浮動小数点のデータ型 (`float` または `double`)、あるいは操作を参照する JVMS のその他の部分は、この CLDC 仕様書では扱いません。

ライブラリの変更またはセキュリティ上の理由により削除された機能

多数の機能が CLDC をサポートする JVM から次に示す理由で削除されています。その理由は、CLDC に含まれている Java ライブラリは正規の J2SE ライブラリよりもかなり制限されていること、また、たとえ機能があったとしても、完全な Java セキュリティモデルが存在しない状況ではセキュリティ上の問題を引き起す可能性があるためです (11 ページの「セキュリティ」を参照)。削除された機能には次のものがあります。

- Java Native Interface (JNI)
- ユーザ定義のクラスローダ
- リフレクション
- スレッドグループとデーモンスレッド
- ファイナライズ
- 弱参照

CLDC で書かれたアプリケーションは、上記の機能のいずれにも依存すべきではありません。次に、これらの各機能の詳細を述べます。

Java Native Interface (JNI)

CLDC をサポートする JVM は、Java Native Interface (JNI) を実装していません。Virtual Machine がネイティブな機能を呼び出す方法は、実装により異なります。JNI は、主に次の 2 つの理由によりサポートされなくなりました。

1) CLDC が提供する限定されたセキュリティモデルは、ネイティブ機能のセットが閉じていることを前提とする (12 ページの「サンドボックスモデル」を参照)

2) JNI を完全に実装するには、CLDC が対象とするデバイスでのメモリの使用量が多すぎる (4 ページの「ハードウェア必要条件」を参照)

ユーザ定義のクラスローダ

CLDC をサポートする JVM は、ユーザ定義の Java レベルクラスローダをサポートしていません (JVMS の §5.3, §2.17.2 を参照)。CLDC をサポートする JVM は、ユーザがオーバーライド、置換、および再構成できない内蔵クラスローダを持っていなければなりません。実際のクラスローダの実装とクラスロード中に起こり得るエラーは、実装により異なります。ユーザ定義のクラスローダの削除は、12 ページの「サンドボックスモデル」で説明しているセキュリティ制約の一部です。

リフレクション

CLDC をサポートする JVM は、リフレクション機能、つまり Java プログラムが Virtual Machine 内のクラス、オブジェクト、メソッド、フィールド、スレッド、実行スタック、その他の実行時構造体の数および内容を検査することを可能にする機能を持っていません。

CLDC をサポートする JVM 上に作成された Java アプリケーションは、リフレクションを必要とする機能に左右されるべきではありません。そのため、CLDC をサポートする JVM は、RMI、オブジェクトの直列化、JVMDI (デバッグインタフェース)、JVMPI (プロファイラインタフェース)、およびその他のリフレクション機能に依存する J2SE の拡張機能をサポートしていません。

スレッドグループとデーモンスレッド

CLDC をサポートする JVM はマルチスレッド機能を実装していますが、スレッドグループとデーモンスレッドはサポートすべきではありません (JVMS §2.19, §8.12-14 を参照)。スレッドを起動および停止させるなどのスレッド操作は、個々のスレッドオブジェクトにのみ適用可能です。アプリケーションプログラマがスレッドグループでスレッド操作をしたい場合は、スレッドオブジェクトを格納するために明示的なコレクションオブジェクトをアプリケーションレベルで使用しなければなりません。

ファイナライズ

CLDC ライブラリには、`Object.finalize()` メソッドが存在しないので、CLDC をサポートする JVM は、クラスインスタンスのファイナライズをサポートしていません。(JVMS §2.17.7 を参照)。 CLDC をサポートする JVM 上に作成されたアプリケーションでは、ファイナライズ機能を利用することはできません。

弱参照

CLDC をサポートする JVM は、弱参照をサポートしていません。CLDC をサポートする JVM 上に作成されたアプリケーションでは、弱参照機能を利用することはできません。

エラー

16 ページの「エラー取り扱い上の制限」で前述したように、CLDC をサポートする JVM のエラー処理機能には制限があります。34 ページの「J2SE から継承されたクラス」に定義してあるエラークラスのサポートとは別に、CLDC をサポートする JVM のエラー処理機能が、対象デバイスに適切な方法で定義されていると考えられます。

クラスファイルの検証

標準の J2SE Java Virtual Machine と同様に CLDC をサポートする JVM は不正なクラスファイルの処理をすべきではありません。ただし、典型的な CLDC が対象とするデバイスには、標準 Java クラスファイルベリファイアが占有する静的および動的なメモリ領域は大きすぎるので、もっと必要メモリの少ない、効率的な検証方法が指定されています。その検証方法を次に説明します。

デバイス外の事前検証とスタックマップによる実行時検証

『Java™ Virtual Machine Specification JVMS』§4.9 に定義されている既存の J2SE クラスファイルベリファイアは、小型でリソースに制約のあるデバイスにとっては不向きです。J2SE が採用しているベリファイアは、少なくとも 50K バイトのバイナリコード領域と、30 〜 100K バイトの動的 RAM を実行時に必要とします。さらに、従来のベリファイアで対話的データフローアルゴリズムを実行するには、かなりの CPU パワーが必要になる可能性があります。

この節で説明している検証方法は、リソースに制約のあるデバイスでは、既存の J2SE ベリファイアよりも著しく小さく、かつ効率的です。典型的なクラスファイルに対して Sun の KVM で新しいベリファイアが実行されると、約 10K バイトの Intel x86 バイナリコードと、100 バイト未満の動的 RAM を必要とします。ベリファイアは、ただバイトコードを先頭から最後まで 1 回走査するだけで、多くのメモリを消費する対話的データフローアルゴリズムを必要としません。

新しいベリファイアに、特別な属性を追加するには、Java クラスファイルが必要です。新しいベリファイアには、この属性を正常なクラスファイルに挿入するプリベリファイアツールが含まれています。変換されたクラスファイルも有効な J2SE クラスファイルで、実行時に効率的な検証を可能にする付加的な属性を持っています (24 ページの「スタックマップ属性の定義」および 28 ページの「Java アプリケーションとリソースの公開」を参照)。これらの属性は、従来のクラスファイルベリファイアでは自動的に無視され、そのためこのソリューションは J2SE Virtual Machine に対して完全に上位互換性を持っています。余分な属性を含んでいる、前処理されたクラスファイルは、元の変更されないクラスファイルに比べて約 5% 大きくなっています。

実行時のバイトコードの検証では、型の安全性を保証します。たとえばベリファイアでパスするクラスは、Java Virtual Machine の型システムに違反せず、メモリも破壊しません。コード署名方式に基づく手法と異なり、このような保証は、検証された属性が確かに信頼できるものであるかどうかは問題ではありません。検証属性がなかったり、不正確、あるいは破壊されていた場合、クラスはベリファイアにより拒否されます。

検証プロセス

新しいクラスファイルベリファイアは、2 つのフェーズ、つまり 1) 事前検証、および 2) デバイス内検証で動作します。事前検証は、一般にデバイス外、つまり Java アプリケーションをダウンロードするサーバ上または、新しいアプリケーションを開発している開発ワークステーション上で行われます。デバイス内検証は、Virtual Machine を内蔵しているデバイス内で実行されます。デバイス内ベリファイアは、プリベリファイアツールが生成する情報を利用します。

実際の検証プロセスは、次のように定義されます。

フェーズ 1： 事前検証 (デバイス外)

新しいベリファイアが提供するプリベリファイアツールが次の 2 つの動作を実行します。

- すべてのサブルーチンをインライン化して、`jsr` (JVMS p. 304、`jsr_w` (JVMS p. 305)、`ret` (JVMS p. 352)、および `wide ret` (JVMS p. 360) のすべてのバイトコードをクラスファイルから削除します。これらの命令を含むメソッドのそれぞれが `jsr`、`jsr_w`、`ret` および `wide ret` バイトコードを含まない意味的に等価なバイトコードで置き換えられます。

- 実行時検証を簡単にするために、特別なスタックマップ属性をクラスファイルに追加します。スタックマップ属性の形式および意味は 24 ページの「スタックマップ属性の定義」で定義してあります。

フェーズ 2: **デバイス内検証**

デバイス内検証アルゴリズムは、次の手順で構成されます。

- 最初に、ベリファイアは、指定されたメソッドのすべての局所変数およびオペランドスタック項目の型を格納するのに十分なメモリを割り当てます。メモリサイズは、`Code` 属性に指定されている局所変数の最大数および最大スタック深さにより決まります。このメモリ領域は、ベリファイアがバイトコードを最初から順にたどるときに派生した型を格納するために使われます。これは、ベリファイアが割り当てる唯一のメモリです。

- 第二に、ベリファイアは派生型を、インスタンスメソッド、引数型、および空のオペランドスタック用の `this` ポインタの型に初期設定します。

- 第三に、ベリファイアは各命令を最初から順に繰り返します。各命令に対して次のことが起こります。
  - 前の命令が、無条件ジャンプ (たとえば `goto`) または復帰 (たとえば `ireturn`)、`athrow`、`tableswitch`、あるいは `lookupswitch` の場合、前の命令から直接的な制御フローはありません。ベリファイアは現在の派生型を無視し、現在の命令用に記録されているスタックマップエントリに応じて派生型を設定します。現在の命令がスタックマップエントリを持っていない場合、ベリファイアはエラーを報告します。
  - 前の命令が、無条件ジャンプ (たとえば `goto`) または復帰 (たとえば `ireturn`)、`athrow`、`tableswitch`、あるいは `lookupswitch` でない場合は、前の命令からの直接的な制御フローがあります。ベリファイアは、現在の命令用にスタックマップエントリが記録されているか検証します。スタックマップエントリがあれば、ベリファイアは派生型を記録されたスタックマップエントリと突き合わせます。記録されている型が派生型よりも一般的でない場合、派生型は記録されている型に設定されます。派生型が記録されている型よりも一般的でない場合は、ベリファイアは型エラーを報告します。
  - 現在の命令が例外ハンドラの適応範囲内にある場合、派生型は、例外ハンドラの開始バイトコードオフセットに対応するスタックマップエントリと突き合わされます。例外ハンドラの開始バイトコードオフセットに対応するスタックマップエントリがない場合、ベリファイアはエラーを報告します。

- その後、ベリファイアは派生型を、命令が期待する派生型と突き合わせます。たとえば `iadd` 命令は、先頭の 2 つのオペランドスタック項目は整数であるとしています。派生型は、その後命令の動作に応じて変更されます。たとえば `iadd` 命令は、2 つの整数をオペランドスタックからポップし、1 つの整数型の結果をオペランドスタックにプッシュします。
- 最後にベリファイアは、現在の命令の直後にない後続の命令用に記録されているスタックマップエントリがあれば、それらと派生型を突き合わせます。

最後に、ベリファイアはメソッド内の最後の命令が無条件ジャンプ (たとえば `goto`) または復帰 (たとえば `ireturn`)、`athrow`、`tableswitch`、あるいは `lookupswitch` であることを確認します。そうでなければ、検証エラーとして「制御フローがメソッドの終りを通り越して落ちる。」と報告します。

上記の手順に加え、デバイス内ベリファイアは次の検証を実行します。つまり、ベリファイアは新しく割り当てられたオブジェクトとコンストラクタが呼び出されたオブジェクトを区別します。コンストラクタが、新しい命令によって指定されたバイトコードオフセットに割り当られたオブジェクト上で一度だけ呼び出されること、新しく割り当てられたオブジェクト上で許可される唯一の操作はそのコンストラクタを呼び出すことであること、および後方分岐が起こる可能性がある場合は局所変数またはオペランドスタックには新しく割り当てられたオブジェクトがないことなどを、ベリファイアは確認する必要があります。

スタックマップ属性の定義

前述したプリベリファイアツールは、特別な属性を追加するためにクラスファイルを修正します。これらの属性は、すべてインタプリタのスタック上に常駐し、局所変数およびオペランドスタック項目の型を記述しているので、スタックマップと呼ばれています。

各スタックマップ属性には複数のエントリがあり、各エントリは指定されたバイトコードオフセットにある局所変数とオペランドスタック項目を記録しています。

スタックマップ属性は、JVMS §4.7.3 に定義されている `Code` 属性の副属性です。`Code` 属性およびスタックマップ属性の `Code` 属性内での位置付けの詳細については、『The Java™ Virtual Machine Specification (Java Series), Second Edition』Tim Lindholm および Frank Yellin 著、Addison-Wesley 1999 年発行 (ISBN 0-201-43294-3) を参照してください。

スタックマップ属性の形式は、次のとおりです。

```
StackMap_attribute {
```

```
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_entries;
    {    u2 byte_code_offset;
         u2 number_of_locals;
         ty types_of_locals[number_of_locals];
         u2 number_of_stack_items;
         ty types_of_stack_items[number_of_stack_items];
    } entries [number_of_entries];
}
```

StackMap_attribute 構造体のデータ項目は次のとおりです。

attribute_name_index

attribute_name_index 項目の値は、constant_pool テーブルへの有効なインデックスでなければなりません。そのインデックスにおける constant_pool エントリは、文字列 StackMap を含む CONSTANT_Utf8_info 構造体でなければなりません。

attribute_length

attribute_length 項目の値は、最初の 6 バイトを除いた、属性の長さを示します。

number_of_entries

number_of_entries 項目の値は、エントリ配列内のエントリの数を示します。

entries[]

各エントリは、指定されたバイトコードオフセットにある局所変数とスタック項目の型を記録しています。かならずしもすべてのバイトコードオフセットが局所変数およびスタック項目の型を記録しなければならないわけではありません。バイトコードオフセットに対する局所変数およびスタック項目の型は、そのオフセットがバイトコード命令の始まりを示し、次に示す条件の 1 つ以上がその命令を満たす場合に、そのバイトコードオフセットのために記録されます。

1. 命令が、条件ジャンプ (たとえば ifeq)、無条件ジャンプ (たとえば goto)、tableswitch、 または alookupswitch 命令の対象である
2. 命令が、Code 属性の exception_table 内にある handler_pc によってマークされている

3. 命令が、無条件ジャンプ (たとえば goto)、tableswitch、lookupswitch、athrow、 または orreturn 命令の直後に続く。この命令が無効なコードでないかぎり、1 または 2 の条件に当てはまらなければならない。

各エントリには、次の項目が含まれます。

byte_code_offset

現在のエントリに対応するバイトコード命令のオフセット。このオフセットの命令は、上記の 3 つの条件の 1 つ以上を満たす必要があります。

number_of_locals

型が記録されている局所変数の数

types_of_locals

局所変数の型。type_of_locals[n] は、局所変数 n (0 <= n < number_of_locals に対して) の型を表わします。型 (ty) は 1 バイトまたは 3 バイトのエントリです。これらは次のように符号化されます。

名前	コード	説明
ITEM_Bogus	0	未知の、あるいは初期化されていない値
ITEM_Integer	1	32 ビット整数
ITEM_Float	2	(CLDC 実装のベリファイアでは使用されていない)
ITEM_Double	3	(CLDC 実装のベリファイアでは使用されていない)
ITEM_Long	4	64 ビット整数
ITEM_Null	5	null の型
ITEM_InitObject	6	詳細は下記参照
ITEM_Object	7	詳細は下記参照
ITEM_NewObject	8	詳細は下記参照

最初の 7 つの型は 1 バイトに符号化され、最後の 2 つの型は 3 バイトに符号化されることに注意してください。

上記の型である ITEM_InitObject、ITEM_Object、および ITEM_NewObject の意味は次のとおりです。

ITEM_InitObject

java.lang.Object 以外のクラスに対するコンストラクタ (<init> メソッド) が、そのスーパークラスの内の 1 つのコンストラクタ(<init> メソッド) を呼出す前は、this ポインタは型 ITEM_InitObject を持っています。(注：ベリファイアはこの型を使用して、コンストラクタが this ポインタに対して他の操作を実行する前に、最初にスーパークラスのコンストラクタを呼び出します。)

ITEM_Object

クラスのインスタンス。1 バイトの型コード (7) の後に 2 バイトの type_name_index (a u2) が続きます。type_name_index の値は、constant_pool テーブルの有効なエントリでなければなりません。そのインデックスにおける constant_pool エントリは、CONSTANT_Class_info 構造体でなければなりません。

ITEM_NewObject

初期化されていないクラスインスタンス。クラスインスタンスは new 命令で生成されてはいるが、コンストラクタ (<init> メソッド) がまだそれに対して呼び出されていない。型コードの 8 の次に 2 バイトの new_instruction_index (a u2) が続きます。new_instruction_index はバイトコード命令の有効なオフセットでなければなりません。バイトコード命令の OP コードは、new でなければなりません。(注：初期化されていないオブジェクトはこの new 命令で生成されます。ベリファイアは、この型を使用して、インスタンスは完全にコンストラクトされるまで使用できないようにします。)

number_of_stack_items

スタック上の項目の数

types_of_stack_items

スタック上の項目の型。types_of_stack_items[0] は、オペランドスタックの底を表わし、types_of_stack_items[n] (0 < n < number_of_stack_items に対して) は、types_of_stack_items[n-1] の上のスタック項目を表わします。

クラスファイルの形式とクラスのロード

Connected, Limited Device Configuration(CLDC) の必須の要件は、Java アプリケーションおよびサードパーティのコンテンツの動的なダウンロードをサポートすることです。Java プラットフォームの動的クラスローディング機構が、ダウンロードする際に中心的役割を果たしています。この節では、CLDC をサポートする JVM に要求されるアプリケーション表現の形式およびクラスローディングの実行方法について説明します。

サポートしているファイル形式

CLDC の実装では、標準 Java クラスファイル (JVMS の第 4 章に定義、事前検証に関する変更は、「Java アプリケーションとリソースの公開」に定義) を読み込むことができるものと仮定されています。さらに、CLDC 実装は、圧縮された Java Archive (JAR) ファイルをサポートしなければなりません。この条件が追加された理由は、大規模な Java 環境および既存の Java ツールに対して上位互換性を維持し、かつ通常のクラスファイルよりも占める領域を少なくするためです。JAR 形式についての詳細な情報が http://java.sun.com/products/jdk/1.2/docs/guide/jar にあります。

一般にネットワーク帯域幅を節約することは、今日の低帯域幅無線ネットワークでは重要なことです。圧縮 JAR 形式は、通常の Java クラスファイルを 30% から 50% 圧縮しながらも、記号情報も損失せず、また既存の Java システムとの互換性でも問題を起しません。

Java アプリケーションとリソースの公開

Java アプリケーションは、そのシステムが公開されて、そのアクセスを可能にするトランスポート層およびプロトコルが公開された標準である場合、「公開されている」あるいは「公的に配信されている」と考えられています。対照的に、デバイスは、ベンダーがすべての通信を制御する唯一の閉じたネットワークシステムの一部となる可能性があります。この場合、アプリケーションは、一度でも閉じたネットワークシステムに入れられるか、あるいはそのシステム経由で配信されれば、もはや公開されません。

CLDC デバイス用に設計された Java アプリケーションが公開された場合は常に、圧縮 JAR ファイル表示形式を使用しなければなりせん。JAR ファイルは、次の制限および追加の要件を持つ正規の Java クラスファイルを含まなければなりません。

- スタックマップ属性 (24 ページの「スタックマップ属性の定義」) が クラスファイル内に含まれること
- クラスファイルは、次の Java バイトコードのどれも含んではならない。`jsr` (JVMS p. 304、`jsr_w` (JVMS p. 305)、`ret` (JVMS p. 352) および `wide ret` (JVMS p. 360)

Sun の CLDC リファレンス実装には、上記の変更を Java クラスファイルに反映するためのプリベリファイアツールが含まれています。スタックマップ属性は JVMS §4.9 に記述されている従来のクラスファイルベリファイアによって自動的に無視されることに注意してください。つまり、ここに規定されている形式は、J2SE または J2EE のような大規模な Java 環境と完全に上位互換性があります。

さらに、JAR ファイルは、メソッド `Class.getResourceAsStream(String name)` (詳細はライブラリのマニュアルを参照) を呼び出すことにより Virtual Machine にロードされる可能性のある、アプリケーション固有のリソースファイルを含んでいる場合があります。

クラスファイルの検索順序

Java 言語仕様書および Java Virtual Machine 仕様書は、新しいクラスファイルが Virtual Machine にロードされるときにクラスファイルが検索される順序についてはなにも規定していません。実装レベルでは、典型的な Java Virtual Machine の実装は、特別な環境変数 `classpath` を使用して検索順序を定義します。

この CLDC 仕様書では、クラスファイルの検索順序は実装によって決まり、次の節で説明する制約があるものと仮定してます。検索方法は、一般的にはアプリケーション管理実装の一部として定義されます (10 ページの「アプリケーション管理」を参照)。CLDC をサポートする JVM は、`classpath` の概念をサポートすることを要求されていませんが、実装レベルでは要求される場合があります。

クラスファイルの検索順序に 2 つの制約が適用されます。第一に、13 ページの「システムクラスの保護」の節で説明しているように、アプリケーションプログラマは、どのような方法ででもシステムクラス (CLDC またはサポートされているプロファイルに属するクラス) をオーバーライドできない、ということを CLDC をサポートする JVM は保証しなければなりません。第二に、アプリケーションプログラマは、いかなる方法によってもクラスファイルの検索順序を操作できない、ということが要求されます。これらの制約の両方ともセキュリティ上の理由から重要です。

実装の最適化

この CLDC 仕様書は、公開および配信された Java アプリケーション用の圧縮 JAR ファイルの使用について規定しています。ただし、閉じたネットワーク環境 (28 ページの「Java アプリケーションとリソースの公開」を参照) および実行時の Virtual Machine の内部では、代りの形式を使用することができます。たとえば、ネットワークトランスポートレベルが低帯域幅の無線ネットワークでは、ネットワーク帯域幅を節約するために、より小さいトランスポート形式を利用するのが便利な場合があります。同様に、ダウンロードしたアプリケーションを CLDC デバイスに格納する場合、ユーザレベルで使用されるアプリケーションが元の表現形式の意味と同じであれば、より小さな表現形式を使用することができます。より小さいクラスファイルの表現形式の定義は実装により異なり、この CLDC 仕様書の対象範囲外であるとみなされます。

プリロード/プリリンク (ROM 化)

CLDC をサポートする JVM が、いくつかのクラスをプリロード / プリリンクすることを選択する場合があります。この方法は、非公式には ROM 化と呼ばれています。一般的に言って、小規模な Virtual Machine の実装では、すべてのシステムクラス (特定の構成またはプロファイルに属するクラス) をプリロードし、アプリケーションのロードを動的に実行します。プリロードの実際のメカニズムは実装に依存し、この CLDC 仕様書の対象範囲外です。あらゆる場合に、プリロード / プリリンクの実行時の効果や意味は、実際のクラスがその時点でロードされた場合と同じでなければなりません。プリロードでユーザが認識できるのは、アプリケーションの起動が速くなることだけです。特に、クラスがシステムにプリロードされていない場合には、そのクラスが最初にロードされる時に明確にクラスの初期化が実行されなければなりません。

将来のクラスファイル形式

通常の Java クラスファイルは、帯域幅の限られている環境のネットワークトランスポート用には最適化されていません。これは、各 Java クラスファイルが、自分自身の定数プール (シンボルテーブル)、メソッド、フィールド、例外テーブル、バイトコード、およびその他の情報を含む、独立した単位だからです。クラスファイルの自己完結的内容は、Java 技術の長所であり、アプリケーションが必ずしも同じ場所に常駐する必要のない複数の要素から構成されることを可能にし、実行時にアプリケーションを動的に拡張することを可能にします。ただし、この柔軟性には対価を伴います。Java アプリケーションが閉じられた単位として扱われれるようなことがあれば、複数

の定数プールとその他の構造体の中の無駄を省くことにより、多くの領域が節約できるはずです。特に完全な記号情報を除外する場合には、かなりの領域が節約されます。また、アプリケーショントランスポート形式の望ましい機能の 1 つとして、パワーの制限された計算環境というのがあります。これは、静的表現と実行時表現の間に特別なロードプロセスまたは変換プロセスを置かずに、その場でアプリケーションを実行する機能です。標準の Java クラスファイルはそのような実行用には設計されていません。

CLDC の将来のリリースでは、Sun Microsystems, Inc. 発行の『Java Card™ 2.1 Virtual Machine Specification, Revision 1.1』に規定されている「cap」ファイルのような、「分割 VM」でプリリンクされたクラスファイルの変形を使用する可能性があります。

第6章

CLDC ライブラリ

概要

Java 2 Platform, Enterprise Edition (J2EE) および Java 2 Platform, Standard Edition (J2SE) は、デスクトップおよびサーバマシン用のエンタープライズアプリケーションのために豊富なライブラリセットを提供します。残念なことに、これらのライブラリは、数メガバイトの実行メモリを必要とし、したがって小規模で資源に制約のあるデバイスには不向きです。

一般的な目的

CLDC 用の Java ライブラリの一般的な目的は、多様な小規模デバイス用の実際的なアプリケーション開発およびプロファイル定義のために最小限の有用なライブラリのセットを提供することです。現在の小型デバイスには厳しいメモリ制約があり、また多様な機能を持っているため、すべてを満たすライブラリのセットを作成するのはほとんど不可能です。機能を包含するための境界線をどこに置いても、境界線はあるデバイスやユーザにとっては低すぎて、他の多くのデバイスやユーザには高すぎることになるのは避けられません。

ライブラリの範囲を定義する際に、元の 『Java Specification Request JSR-30』をガイドラインとして採用し、接続性に重点を置きました。これが意味するのは、CLDC に含まれるライブラリは、基本システムクラスおよびデータ型クラスに加えて、現在および未来の小規模接続デバイスのための、ネットワーク機能の拡張可能なセットを提供するべきである、ということです。

互換性

CLDC に含まれるライブラリの大部分は、アプリケーションの上位互換性および移植性を保証するために、より大きな Java エディション (J2SE および J2EE) となっています。上位互換性は非常に望ましい目標ですが、J2SE および J2EE ライブラリは、セキュリティ、入出力、ユーザインタフェース定義、ネットワーキング、それにストレージのような重要な領域で、それらのサブセットの作成を難しくするような強い内部依存性を持っています。これらの依存性は、長年にわたる Java ライブラリの開発において設計が進化し再利用されたことによる当然の結果です。このような理由で、ライブラリのいくつか、特にネットワーキングおよび入出力に関して再設計を行いました。

この CLDC 仕様書で提供される CLDC ライブラリは、2 つのカテゴリーに分類できます。

- 標準 J2SE ライブラリのサブセットであるクラス類
- CLDC に固有で (ただし J2SE にマップ可能) なクラス類

前者のカテゴリーに属するクラスは、パッケージ `java.lang.*`、`java.util.*`、および `java.io.*` にあります。これらのクラスは Java 2 Standard Edition バージョン 1.3 から派生したものです。これらのクラスの詳細なリストは、34 ページの「J2SE から継承されたクラス」にあります。

後者のカテゴリーに属するクラスは、パッケージ `javax.microedition.*` にあります。これらのクラスは40 ページの「CLDC 固有のクラス」で説明しています。

J2SE から継承されたクラス

CLDC は J2SE から継承されたクラスを多数提供しています。J2ME 構成の規則によれば、J2SE クラスと同じ名前とパッケージ名をもつクラスは、対応する J2SE クラスと同じか、あるいはそのサブセットでなければならない、となっています。そのサブセットに含まれるクラスとそれらのメソッドの意味は変更することはできません。クラスには、対応する J2SE クラスにない public または protected メソッドあるいはフィールドを追加しないでください。これらの規則に関する公式情報については、Sun Microsystems, Inc. 発行『Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)』を参照してください。

システムクラス

J2SE クラスライブラリには、Java Virtual Machine と緊密に結合されているクラスがいくつか含まれます。同様に、いくつかの標準 Java ツールでは、特定のクラスがシステムにあることを仮定しています。たとえば、J2SE Java コンパイラ (`javac`) は、クラス `String` および `StringBuffer` のある関数が利用可能であることを要求するようなコードを生成します。次に示すシステムクラスが含まれます。

```
java.lang.Object
java.lang.Class
java.lang.Runtime
java.lang.System
java.lang.Thread
java.lang.Runnable (interface)
java.lang.String
java.lang.StringBuffer
java.lang.Throwable
```

データ型クラス

パッケージ `java.lang.*` から派生した次の基本データ型クラスがサポートされています。各クラスが、対応する J2SE のクラスのサブセットになっています。

```
java.lang.Boolean
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Character
```

コレクションクラス

パッケージ `java.util.*` から派生した次のコレクションクラスがサポートされています。

```
java.util.Vector
```

```
java.util.Stack
java.util.Hashtable
java.util.Enumeration (interface)
```

入出力クラス

パッケージ `java.io.*` から派生した次のクラスがサポートされています。クラスの `Reader`、`Writer`、`InputStreamReader` および `InputStreamWriter` は、国際化 (38 ページの「国際化」を参照) をサポートするために必要です。

```
java.io.InputStream
java.io.OutputStream
java.io.ByteArrayInputStream
java.io.ByteArrayOutputStream
java.io.DataInput (interface)
java.io.DataOutput (interface)
java.io.DataInputStream
java.io.DataOutputStream
java.io.Reader
java.io.Writer
java.io.InputStreamReader
java.io.OutputStreamWriter
java.io.PrintStream
```

カレンダーおよびタイムクラス

CLDC には、標準 J2SE クラスの `java.util.Calendar`、`java.util.Date`、および `java.util.TimeZone` の小さなサブセットが含まれます。デフォルトでは、1 つのタイムゾーンのみがサポートされています。追加のタイムゾーンは、実装により提供されます。

```
java.util.Calendar
java.util.Date
java.util.TimeZone
```

追加のユーティリティクラス

2 つのユーティリティクラスが追加されています。クラス `java.util.Random` は、ゲームのようなアプリケーションを実装する際に有用な、簡単な擬似乱数発生関数を提供します。クラス `java.lang.Math` は、他の `Java` ライブラリクラスが頻繁に利用するメソッド `min`、`max`、および `abs` (データ型 `int` および `long` 用) を提供します。

```
java.util.Random
java.lang.Math
```

例外とエラークラス

CLDC に含まれるライブラリは、一般に J2SE ライブラリと高い互換性をもつように意図されているため、CLDC に含まれるライブラリクラスは、通常の J2SE クラスとまったく同じ例外をスローしなければなりません。したがって、かなり包括的なセットの例外クラスが含まれています。

対照的に、16 ページの「エラー取り扱い上の制限」で説明したように、CLDC のエラー処理機能には限界があります。デフォルトでは、CLDC をサポートする JVM は、次の節 38 ページの「エラークラス」に記述されているエラークラスのみをサポートすることが要求されています。

例外クラス

```
java.lang.Exception
java.lang.ClassNotFoundException
java.lang.IllegalAccessException
java.lang.InstantiationException
java.lang.InterruptedException
java.lang.RuntimeException
java.lang.ArithmeticException
java.lang.ArrayStoreException
java.lang.ClassCastException
java.lang.IllegalArgumentException
java.lang.IllegalThreadStateException
java.lang.NumberFormatException
```

```
java.lang.IllegalMonitorStateException
java.lang.IndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException
java.lang.StringIndexOutOfBoundsException
java.lang.NegativeArraySizeException
java.lang.NullPointerException
java.lang.SecurityException

java.util.EmptyStackException
java.util.NoSuchElementException

java.io.EOFException
java.io.IOException
java.io.InterruptedIOException
java.io.UnsupportedEncodingException
java.io.UTFDataFormatException
```

エラークラス

```
java.lang.Error
java.lang.VirtualMachineError
java.lang.OutOfMemoryError
```

国際化

CLDC には、Unicode 文字とバイトシーケンス間の変換をするための限られたサポートが含まれています。J2SE では、これは *Readers* および *Writers* と呼ばれるオブジェクトを使用して行われていますが、ここでは同じメカニズムが、同じコンストラクタを持った `InputStreamReader` および `OutputStreamWriter` クラスを使用して行われています。

```
new InputStreamReader(InputStream is);
new InputStreamReader(InputStream is, String name);
new OutputStreamWriter(OutputStream os);
new OutputStreamWriter(OutputStream os, String name);
```

文字列パラメータがある場合は、それは使用すべきエンコーディングの名前です。文字パラメータがない場合は、デフォルトのエンコーディング (システムプロパティの `microedition.encoding` により定義されている) が使われます。特別な実装では別のコンバータが追加される場合があります。あるエンコーディング用のコンバータがない場合は、`UnsupportedEncodingException` がスローされます。J2SE における文字エンコーディングに関する明確な情報については、http://java.sun.com/products/jdk/1.3/docs/guide/intl/encoding.doc.html を参照してください。

CLDC は地域に対応した機能を提供しないことに注意してください。これは、日付、時間、通貨などの書式化に関するすべてのソリューションは、CLDC の扱う範囲外であるとみなされるからです。

プロパティサポート

CLDC をサポートする JVM は、J2SE の一部であるクラス `java.util.Properties` を実装していません。ただし、次の表で説明しているように、限定されたプロパティのセットがあります。これらのプロパティには、メソッド `System.getProperty(String key)` を呼出すことによりアクセスできます。

プロパティ `microedition.encoding` は、デフォルトの文字エンコーディング名を記述します。

表 6-1　標準システムプロパティ

キー	説明	値
`microedition.platform`	ホストプラットフォームまたはデバイス名	デフォルトは null
`microedition.encoding`	デフォルトの文字エンコーディング	デフォルトは ISO8859_1
`microedition.configuration`	サポートしている構成の名前とバージョン	デフォルトは CLDC-1.0
`microedition.profiles`	サポートしているプロファイル名	デフォルトは null

この情報は、システムが国際化をサポートする際に、デフォルトの文字エンコーディング用の正しいクラスを探すために使用されます。プロパティ `microedition.platform` は、ホストプラットフォームまたはデバイスの性質を記

述します。プロパティ `microedition.configuration` は、現在の J2ME 構成およびバージョンを記述し、プロパティ `microedition.profiles` は、空白文字で区切られたサポートプロファイル名を定義します。

プロファイルが、上記の表 6-1 に含まれない追加のプロパティを定義する場合があります。

CLDC 固有のクラス

この節では、入出力およびネットワーキングを一般化した、拡張可能な方法でサポートするための Generic Connection フレームワークについて説明します。Generic Connection フレームワークは、資源に制約のある環境でデータにアクセスし、編成するための一貫した手段を提供します。

背景と動機

J2SE および J2EE ライブラリは、ストレージおよびネットワーキングシステムへの入出力を取り扱うための機能の豊富なセットを提供します。J2SE のパッケージ `java.io.*` は、約 60 のクラスおよびインタフェース、それに 16 以上の 例外クラスを含んでいます。J2SE のパッケージ `java.net.*` は、約 20 の 通常クラスおよび 10 の例外クラスから構成されています。これらのクラスファイルの静的なサイズの合計は、約 200 キロバイトです。せいぜい数百キロバイトのメモリしかない小規模デバイスに、この機能を持たせるのは困難です。さらに、標準入出力およびネットワーキング機能のかなりの部分は、赤外線または Bluetooth のような特殊な種類の接続をしばしばサポートする必要のある、あるいは TCP/IP のサポートを提供しない、今日の小規模デバイスには直接適用できません。

一般的に言って、ネットワーキングおよびストレージライブラリに対する要求は、資源に制約のあるデバイス間でかなり異なります。パケット交換ネットワークを取り扱っているこれらのデバイスのメーカーは、データグラムに基づいた通信メカニズムを要求するのが普通ですが、回線交換ネットワークを取り扱っているメーカーストリームに基づいた接続を要求します。伝統的なファイルシステムを持つデバイスもありますが、他の多くのデバイスは非常にデバイス固有のメカニズムを持っています。厳しいメモリ制限のために、ある種の入出力、ネットワーキングおよびストレージ機能をサポートしているメーカーは、他のメカニズムをサポートしたくないのが普通です。このため、CLDC 用にこれらの機能を設計することは非常に努力を要することに

なります。特に、J2ME 構成はオプションの機能を定義することが許されていないのでなおさらです。また、複数のネットワーキングメカニズムおよびプロトコルがあることは、アプリケーションプログラマにとっては潜在的に紛らわしく、プログラマが低レベルのプロトコル問題に対処しなければならない場合は、特にわずらわしいものとなります。

Generic Connection フレームワーク

メモリ領域の小さい J2ME システムを持つことが要求された結果、J2SE ネットワークおよび入出力クラスを一般化することになりました。この新しいシステムの一般的目的は、J2SE クラスの正確な機能的サブセットを実現することで一般的低レベルハードウェアまたは J2SE 実装に簡単にマップ可能で、かつ新しいデバイスおよびプロトコルをサポートする際により良い拡張性、柔軟性、および整合性を発揮することです。

一般的な概念を次に示します。異なる形態の通信に対してまったく異る種類の抽象化を使用するかわりに、関連する抽象化のセットがアプリケーションプログラミングレベルで使用されます。

一般形式

すべての接続は、`Connector` と呼ばれるシステムクラスの唯一の静的メソッドを使用して作成されます。成功した場合、このメソッドは汎用接続インタフェースの 1 つを実装するオブジェクトを返します。`Connection` インタフェースをルートにした階層を形成する、これらのインタフェースが多数存在します。このメソッドは、次の形式の文字パラメータを 1 つ取ります。

```
Connector.open("<プロトコル>:<アドレス>;<パラメータ群>");
```

これらの文字列の構文は、一般に IETF 標準の RFC2396 (http://www.ietf.org/rfc/rfc2396.txt) に定義されている Uniform Resource Indicator (URI) 構文に従うべきです。

注 – これらの例は、説明のためにのみ掲載しています。 CLDC 自身には、プロトコルの実装の、いかなるものも含まれていません (42 ページの「構成レベルで提供されない実装」を参照)。特別な J2ME プロファイルがこれらの接続のすべてをサポートするということは期待されていません。また、J2ME プロファイルが、次に示されていないプロトコルをサポートしている場合もあります。

HTTP

```
Connector.open("http://www.foo.com");
```

ソケット

```
Connector.open("socket://129.144.111.222:9000");
```

通信ポート

```
Connector.open("comm:0;baudrate=9600");
```

データグラム

```
Connector.open("datagram://129.144.111.333");
```

ファイル

```
Connector.open("file:/foo.dat");
```

この設計の重要な目的は、プロトコルのセットアップ間の相違をできるだけ分離して、接続の種類を特徴づける文字列にすることです。この文字列は、`Connector.open()` へのパラメータとなります。この手法の大きな長所は、使用されている接続の種類にかかわらず、多量のアプリケーションコードが同じであるということです。これは、アプリケーション内で使用されている抽象実装が、通信形態を変更したときに劇的に変化することがたびたびある従来型の実装とは異なります。

プロトコルの J2ME プログラムとのバインディングは実行時に行われます。実装レベルでは、`Connector.open()` へのパラメータとして与えられる (: が最初に現れるまでの) 文字列が、システムに対して、すべてのプロトコル実装が格納されている場所から所定のプロトコル実装を取得するように指示します。これは遅延バインディングといい、プログラムが実行時に異なるプロトコルに動的に適合します。これは概念的には、PC またはワークステーションコンピュータ上のアプリケーションプログラムとデバイスドライバ間の関係と同じです。

構成レベルで提供されない実装

CLDC に含まれている Generic Connection フレームワークは、実際にサポートされているプロトコルを指定せず、また、特定のプロトコルの実装も持っていません。実際の実装およびサポートされているプロトコルに関する決定は、プロファイルレベルで行われなければなりません。

Generic Connection フレームワークの設計

異なる種類のデバイスに接続するには異なる種類の動作が必要になります。たとえば、ファイル名は変更できますが、TCP/IP ソケットの場合は同様な操作はありません (時間を通して送信されるデータは、空間を通して送信されるデータとはまったく異なる方法で管理されます)。Generic Connection フレームワークはこれらの異なる機能を反映して、論理的に同じ操作は同じ API を共有することを保証します。

新しいフレームワークが、同じ意味を持つプロトコルのクラスをグループにまとめる `Connection` インタフェースの階層を利用して実装されています。この階層は 7 つのインタフェースから構成されます。更に、`Connector` クラス、例外クラスが 1 つ、その他のインタフェースが 1 つ、および多数のデータを読み書きするためのデータストリームクラスがあります。実装のレベルでは、各サポートされたプロトコルを実現するために少なくとも 1 つのクラスが必要です。各プロトコル実装クラスには、単に基礎となるホストオペレーティングシステムのネイティブな機能を呼び出すラッパー機能が多数含まれていることがあります。

Generic Connection フレームワークにより経路指定される 6 つの基本インタフェース型があります。

- 基本シリアル入力デバイス
- 基本シリアル出力デバイス
- データグラム指向の通信デバイス
- 回線指向の通信デバイス (TCP など)
- サーバにクライアント-サーバ接続を知らせるための通知メカニズム
- 基本 Web サーバ接続

`Connection` インタフェースのコレクションは、ルート `Connection` インタフェースから派生するにつれて、徐々に機能を増していく階層を形成しています。この階層のおかげで、アプリケーションプログラマは、作成中のコードのためのクロスプロトコルの移植性について最適なレベルを選択できます。

Connection インタフェース階層を図 6-1 に示します。

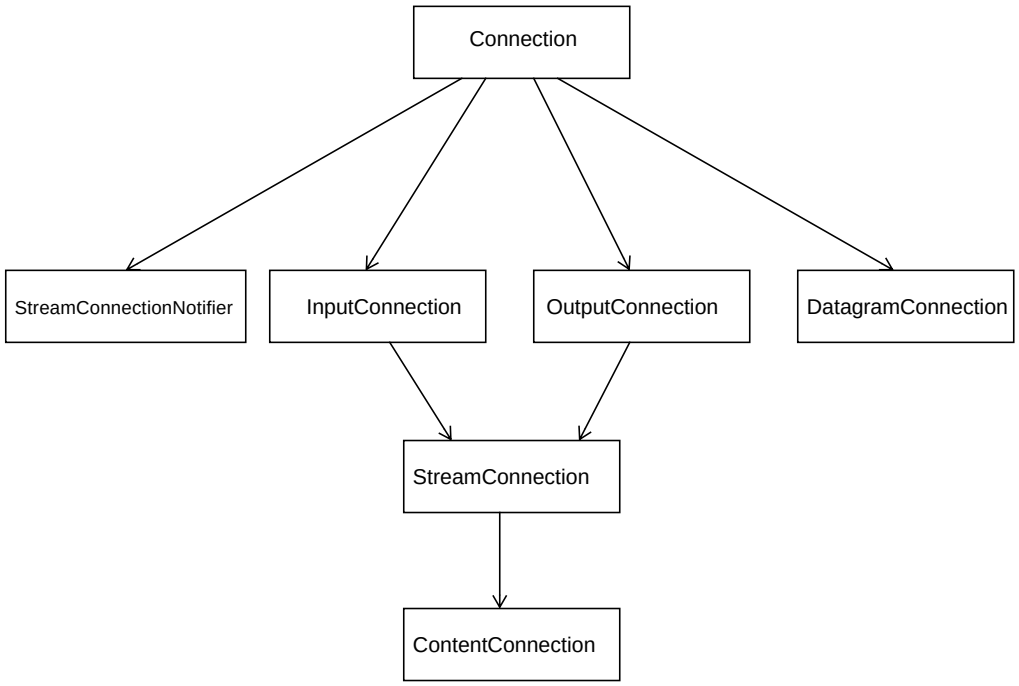

図 6-1　Connection インタフェース階層

インタフェース Connection

これは、開いたり閉じたりできるだけの最も基本的な接続型です。open メソッドは、Connector クラスの静的な open() メソッド経由で常に呼び出されるので、public ではありません。

メソッド:

```
public void close() throws IOException;
```

インタフェース InputConnection

この接続型は、データを読み出すことができるデバイスを表わします。このインタフェースの openInputStream メソッドは、接続のための入力ストリームを返します。

メソッド:

```
public InputStream openInputStream() throws IOException;
public DataInputStream openDataInputStream() throws IOException;
```

インタフェース OutputConnection

この接続型はデータを書き込むことができるデバイスを表わします。このインタフェースの `openInputStream` メソッドは、接続のための出力ストリームを返します。

メソッド:

```
public OutputStream openOutputStream() throws IOException;
public DataOutputStream openDataOutputStream() throws IOException;
```

インタフェース StreamConnection

これは単に、`InputConnection` と `OutputConnection` インタフェースを結合しただけのインタフェースです。通信インタフェースを実装するクラスの論理的な出発点を形成します。

インタフェース ContentConnection

これは、`StreamConnection` のサブインタフェースで、HTTP 接続により提供される最も基本的なメタデータのある部分に対するアクセスを提供します。

メソッド:

```
public String getType();
public String getEncoding();
public long getLength();
```

インタフェース StreamConnectionNotifier

この接続型は、接続が確立するまで待機するために使用します。このクラスの `acceptAndOpen` メソッドは、クライアントプログラムが接続するまでブロックされます。このメソッドは、通信リンクが確立した `StreamConnection` を返します。すべての接続と同様に、返されたストリーム接続は、不要になったときには閉じられなければなりません。

メソッド:

```
public StreamConnection acceptAndOpen() throws IOException;
```

インタフェース DatagramConnection

このインタフェースは、データグラムの端点を表わします。J2SE データグラムインタフェースと同様に、接続を開くために使用されるアドレスは、データグラムが受信される端点です。スローすべきデータグラムの行き先は、データグラムオブジェクト自身の中に置かれます。この API にはアドレスオブジェクトはありません。その代わりに、Connection インタフェースの設計と同様に、アドレスの特定を抽象化させることができる文字列が使用されます。

メソッド:

```
public String getAddress();
public int getMaximumLength();
public int getNominalLength();
public void setTimeout(int time);
public void send(Datagram datagram);
public void receive(Datagram datagram);
public Datagram newDatagram(int size);
public Datagram newDatagram(byte[] buf, int size);
public Datagram newDatagram(byte[] buf, int size, String addr);
```

このクラスは、データバッファおよび関連するアドレスを格納するために使用される Datagram と呼ばれるデータ型を必要とします。Datagram インタフェースは、データをデータグラムバッファに出し入れできるアクセスメソッドの有用なセットを持っています。これらのアクセスメソッドは、DataInput および DataOutput インタフェースに準拠します。つまり、データグラムオブジェクトもまたストリームのように振舞います。現在位置は、データグラム内に保存されます。これは、読み出し動作が実行された後に、自動的にデータグラムバッファの始めにリセットされます。

注記

データを汎用の接続から読み出したり書き込んだりするためには、多くの入出力ストリームクラスが必要になります。CLDC がサポートするストリームクラスは、36 ページの「入出力クラス」に記述されています。プロファイルが必要に応じて追加ストリームクラスを提供する場合もあります。

例

Generic Connection フレームワーク使用例が別の資料にあります。

付録A

Java 2 Micro Edition および KVM の紹介

Java 2 Platform Micro Edition の紹介

1 つのサイズですべてに適合させることはできないことを認識して、Sun は最近になって Java 技術を 3 つのエディション、すなわち Java 2 Platform Enterprise Edition (J2EE)、Java 2 Platform Standard Edition (J2SE)、および Java 2 Platform Micro Edition (J2ME) に再編成しました。各エディションは特定の市場に照準を当てています。

- Java 2 Enterprise Edition。顧客、サプライア、および従業員に、堅実で完全な、拡張可能なインターネットビジネスサーバソリューションを提供する必要のあるエンタープライズ用
- Java 2 Standard Edition。定着したデスクトップ市場用
- Java 2 Micro Edition。多様な情報デバイスを構築する顧客および組み込みデバイスメーカー、それらのデバイス経由でコンテンツを顧客に配信したいサービスプロバイダ、および小規模で資源に制約のあるデバイス用に説得力のあるコンテンツを製作するコンテンツ作成者用

各エディションは、特定の製品に使用できるツールおよび提供物のセットを定義しています。

- 広範囲のコンピュータデバイスに適合する Java Virtual Machine
- コンピュータデバイスの個々の種類に特化したライブラリおよび API
- 配備およびデバイス構成のためのツール

図 1-1 に各 Java エディションの対象市場を示します。

Java 2 Micro Edition (J2ME または Java 2 ME) は、ポケットベルなどの小さな商品から、セットトップボックス、すなわちデスクトップコンピュータ程度の機能のある機器までの広い範囲をカバーしている、特に大規模で、急速に成長しているコンシューマ製品群を取り扱っています。より大きな Java エディションのように、Java 2 Micro Edition は、Java 技術が知られるきっかけになった、次のような品質を維持することをめざしています。

- 「どこでも、いつでも、どんなデバイス上でも実行」という意味で製品間で共通に内蔵される整合性
- 大規模な開発者ベースを持つ高レベルオブジェクト指向言語のパワー
- コードの移植性
- 安全なネットワーク配信
- J2SE および J2EE への上位拡張性

J2ME により Sun は、コンシューマおよび組み込み市場用に動的で拡張可能なネットワーク製品およびアプリケーションを提供することを目指しています。J2ME により、デバイスメーカー、サービスプロバイダ、およびコンテンツ作成者は、人を引き付けるような新しいアプリケーションおよびサービスを開発し、世界中の顧客に展開することにより、競争上の優位性と、新しい開発資本の手段を獲得できます。

図 A-1　Java 2 エディションおよびそれらの対象市場

全体としては、J2ME は 2 つの明確なグループの製品を対象にしています (図 1-1 を参照)。

- パーソナルでモバイル接続される情報デバイス。携帯電話、ポケットベル、およびオーガナイザは、このクラスの装置の最適な例です。これらのデバイスは、デスクトップコンピュータシステムと比較して非常に単純なユーザインタフェースを持ち、合計メモリ容量は、128K から 512K バイトの範囲にあり、低帯域幅で、ネットワークには断続的に接続されます。このグループの製品におけるネットワーク通信は、必ずしも TCP/IP プロトコル群をベースにしていません。
- 共有された、固定接続された情報デバイス。このカテゴリーのデバイスの典型的な例としては、セットトップボックス、インターネット TV、インターネット対応のスクリーンフォン、ハイエンドコミュニケーター、およびカーエンターテインメント/ナビゲーションシステムがあります。これらのデバイスは、より広範囲のユーザインタフェース機能を持ち、合計メモリ容量は、2M から 16M バイトの範囲にあり、かつネットワークに対しては永続的で高帯域幅の TCP/IP 接続ができます。

これらの製品グループの境界線は流動的であることに注目する必要があります。コンピュータ、電気通信、コンシューマエレクトロニクス、および娯楽産業が 1 つに収束しつつあるため、より多くの汎用コンピュータ、パーソナル通信デバイス、コンシューマエレクトロニクスデバイス、および娯楽デバイス間の区別が曖昧になっています。また、将来はデバイスが、伝統的な固定ネットワークの代りに無線接続をますます利用するようになると予想されます。実際 2 つのグループ間の境界線は、特定の機能性またはある種の接続性よりも、合計メモリ容量やデバイスの物理的スクリーンサイズによって定義されるようになります。

J2ME 構成およびプロファイル

携帯電話、ポケットベル、電子手帳、およびセットトップボックスなどの接続コンシューマデバイスには共通のものがありますが、それらは、形態、機能、および特徴の点で多種多様です。情報機器は、専用で限定された機能のデバイスになる傾向があります。この多様性に対処するための、J2ME に課せられた重要な要件としては、小型というだけではなく、モジュール性とカスタマイズ可能性もあります。

Java 2 ME アーキテクチャは、顧客および組み込み市場が要求しているある種の柔軟な展開をサポートできるように、モジュール式で拡張可能になっています。これを可能にするために、Java 2 ME は、広範囲の Virtual Machine 技術を提供していますが、個々の技術はコンシューマおよび組み込み市場でよく見られる、異るプロセッサとメモリ占有領域用に最適化されています。

ローエンドの資源に制約のある製品に対して、Java 2 ME は、さまざまなデバイスに対し必要不可欠な機能を持つ最小限の構成の Java Virtual Machine および Java API をサポートしています。デバイスメーカーが、デバイスに新しい機能を開発するにつれて、あるいはサービスプロバイダが新しくてエキサイティングなアプリケーションを開発するにつれて、これらの最小の構成は、API を追加しより豊富な Java Virtual Machine の機能の補強をすることにより拡張することができます。この種のカスタマイズ可能性および拡張性をサポートするために、次の必要不可欠なコンセプトが定義されています。

- 構成。J2ME の構成は、合計メモリ容量および処理能力に関して同様な条件が課せられている、デバイスの「水平的な」クラスまたはファミリ用の最小プラットフォームです、1 つの構成は、デバイスメーカーまたはコンテンツプロバイダが、同じクラスのすべてのデバイスで利用可能であると期待できる、Java 言語、Virtual Machine 機能、および 最小限のライブラリを定義します。
- プロファイル。J2ME のプロファイルは、ある種の「垂直的」市場セグメントまたはデバイスカテゴリの特定の需要を取り扱います。プロファイルの主要な目的は、ある垂直的デバイスカテゴリーまたはドメインにおける相互運用性を、その市場用に標準 Java プラットフォームを定義することにより保証することです。プロファイルには、構成内に提供されているライブラリよりもはるかにデバイスカテゴリに特有なライブラリが含まれています。プロファイルは、1 つの構成上に実装されています。

構成およびプロファイルは、Java Community Process (JCP) により定義されます。構成およびプロファイルの詳細を次に説明します。

構成

J2ME は多数の構成に展開できます。1 つの構成が、合計メモリ容量その他のハードウェア機能に関して同様な必要条件を持っているデバイスの「水平的な」クラスまたはファミリ用の Java プラットフォームを定義します。具体的には、1 つの構成は次のことを行います。

- サポートされている Java プログラミング言語機能を指定する

- サポートされている Java Virtual Machine 機能を指定する
- サポートされている Java ライブラリおよび API を指定する

各構成は、ユーザおよびコンテンツプロバイダが、工場から出荷された状態のすべてのデバイスで利用可能であると安心して考えられる Java Virtual Machine 機能および API のセットを指定します。アプリケーション開発者およびコンテンツプロバイダは、その構成が指定した Java Virtual Machine 機能および API の境界内に収まるようにコードを設計しなければなりません。

フラグメンテーションを回避するために、非常に限られた数の J2ME 構成があります。現在のところ、その目的は、2 つの標準 J2ME 構成を定義することです (ページ 2 の 図 1-1 を参照)。

- **Connected, Limited Device Configuration (CLDC)**。パーソナルでモバイル接続されたデバイスで構成される市場は、CLDC が取り扱います。この構成には、J2SE API から得られたものではありませんが、特に小さい容量のデバイスの需要を満たすように設計された、新しいクラスがいくつか含まれます。
- **Connected Device Configuration (CDC)**。共有された固定の接続情報デバイスは、Connected Device Configuration (CDC) が取り扱います。構成間の上位互換性を保証するために、CDC は CLDC のスーパーセットになっていなければなりません。

この仕様書は Connected, Limited Device Configuration (CLDC) に焦点を合わせています。

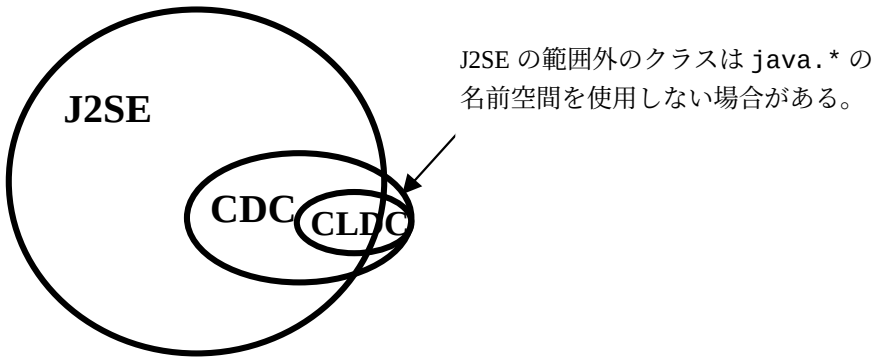

図 A-2　J2ME の構成と J2SE の関係

図 A-2 に CLDC、CDC および Java 2 Standard Edition (J2SE) の間の関係を示します。図に示されているように、CLDC および CDC の機能の大部分が J2SE から継承されています。J2SE から継承された各クラスは、Java 2 Standard Edition の対応するクラスとまったく同じか、あるいはそれのサブセットになっていなければなりません。さらに、CLDC および CDC は、J2SE API から得られたものではありません。特に小さい容量のデバイスの需要を満たすように設計された、多数の機能を導入する場合があります。詳細については、Sun Microsystems, Inc. 発行の『Configurations and Profiles Architecture Specification, Java™ 2 Platform Micro Edition (J2ME)』を参照してください。

プロファイル

アプリケーションの移植性は、デスクトップおよびエンタープライズサーバ市場の重要な長所になっています。移植性は、またコンシューマデバイス用 J2ME の価値の重要な要素です。ただし、コンシューマ製品群におけるアプリケーションの移植性の必要条件は、一般に、デスクトップおよびサーバ市場が要求する移植性の必要条件とはまったく異なっています。ほとんどの場合コンシューマデバイスには、メモリサイズ、ネットワーキング、およびユーザインタフェースの機能の点で相当な違いがあり、すべてのデバイスを 1 つのソリューションだけでサポートするのは非常に困難です。

一般にコンシューマデバイスの市場は、エンドユーザが普遍的なアプリケーションの移植性を期待し、あるいは要求する程同質ではありません。むしろ、コンシューマ製品群においてはアプリケーションが同種のデバイス間で完全な移植性を持っているのが理想的です。さらに、ある種のアプリケーション、たとえば支払および銀行用のアプリケーションでは、多種類のデバイス間に高い移植性があることが期待されています。

特定の垂直市場用に Java プラットフォームを定義できるようにするために、J2ME のフレームワークがプロファイルの概念を提供しています。プロファイルは、特定の垂直市場セグメントまたはデバイスカテゴリのための Java プラットフォームを定義します。プロファイルは、2 つの明確な移植性要件を満たすことができます。

- 携帯電話、ポケットベルまたはセットトップボックスのような特別な種類のデバイス用のアプリケーションを実現するための完全なツールキットを提供すること
- プロファイルは、数種類のデバイス上で処理される可能性のある、重要で整合性のあるグループのアプリケーション (たとえばパーソナル情報管理) をサポートするために作成される場合がある

実装のレベルでは、プロファイルは、特定の市場セグメントのデバイス用のドメイン固有の機能を提供するために構成の上位に位置する Java API およびクラスライブラリの単なるコレクションとして定義されます。プロファイルおよび J2ME プロファイルを定義するための特別な規則が、別の仕様書で詳しく記述されています。

KVM の紹介

現在のところ CLDC は Sun の K Virtual Machine (KVM) 上でのみ動作します。ただし、この仕様書では、別の Virtual Machine 上で動作する可能性を認めています。KVM は、小規模で資源に制約のあるデバイス用に最初から特別に設計された、小型で携帯可能な Virtual Machine です。KVM の全体的な設計目標は、Java プログラミング言語の中心的な側面のすべてを維持するが、数 100k バイトの合計メモリ容量しかなくて、資源に制約のあるデバイスで動作する、できるだけ小さい「完全な」Java Virtual Machine を作成することでした。具体的には、KVM の設計意図は次のとおりです。

- 小さいこと、つまり、Virtual Machine のコアの静的な占有メモリ領域は、 40K バイトから 80K バイトの範囲にあること (コンパイルオプションおよび対象プラットフォームに依存する)
- クリーンで高度な移植性があること
- モジュール式でカスタマイズ可能であること
- 他の設計目標を損なうことなく安全で速いこと

KVM の K は「キロ」を表します。KVM のサイズが数 10K バイト単位で測定されているためそのように名付けられました。KVM は、合計メモリ容量が数 100K バイト (あるいは 128K バイト未満) の 16/32 ビットの RISC/CISC マイクロプロセッサに適しています。これは、典型的にはディジタル携帯電話、ポケットベル、電子手帳、および小型の小売り支払システムに適用されます。

KVM の実装が要求する最小合計メモリ容量は、約 128K バイトで、そのなかには、Virtual Machine、最小限の Java ライブラリ、およびある程度の Java アプリケーションを実行するためのヒープ領域を含みます。より一般的な実装では 256K バイトの合計メモリ容量が必要で、そのうち半分はアプリケーションのためのヒープ領域として利用されます。また、40K から 80K バイトが Virtual Machine 自体のために必要とされ、残りはクラスライブラリのために確保されています。合計メモリ容量内の揮発性メモリ (たとえば DRAM) と非揮発性メモリ (たとえば ROM またはフラッシュ) の割

合は、実装により著しく変ります。システムクラスのプリリンクサポートのない単純な KVM 実装では、システムクラスがデバイスにプリロードされた KVM 実装よりも多くの揮発性メモリを必要とします。

対象デバイス内の KVM の実際の役割は、著しく変る可能性があります。ある実装では、KVM は既存のソフトウェアスタックの先頭で使用されて、デバイスに対し動的で、対話的、かつ安全な Java コンテンツをダウンロードし実行する機能を与えます。別の実装では、KVM はより低いレベルで使用され、システムソフトウェアおよびデバイスのアプリケーションを Java プログラミング言語で実現しています。代りの使用モデルもいくつか可能です。

KVM に関する詳細については、KVM の web サイトの
http://java.sun.com/products/kvm を参照してください

KVM は、もともとは Sun Microsystems Laboratories で開発された Spotless と呼ばれる研究システムから生まれたものです。Spotless に関する詳しい情報は、Sun Labs テクニカルレポートの 『The Spotless System: Implementing a Java system for the Palm Connected Organizer』(Sun Labs Technical Report SMLI TR-99-73) にあります。