JDK 1.1 開発ガイド (Solaris 編)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650–960–1300

Part Number 806-3702–10
2000 年 3 月

原典: *JDK 1.1 for Solaris Developer's Guide*

Part No: 806-3461-10

Revision A

目次

はじめに **5**

1. **Java** プログラミング環境の概要 **9**

Java プログラミング環境と Java 実行時環境 (JRE) 9

Java プログラミング環境とは 9

JRE の構成要素 11

Java 仮想マシン (JVM) 11

Sun の JIT コンパイラ 13

2. マルチスレッド **17**

マルチスレッドの定義 * 17

Solaris 環境における従来の Java スレッド * 17

マルチスレッドの概念 * 18

マルチスレッド化の利点 * 19

マルチスレッドモデル 20

「複数対単一」のモデル (グリーンスレッド) 20

「単一対単一」のモデル 21

「複数対複数」のモデル (Solaris 上の Java - ネイティブスレッド) 22

マルチスレッドカーネル 23

Solaris 上の Java マルチスレッドの利点 24

スレッドのグループ化 28

Java スレッドに関する注意事項　28

Java 全般の注意事項　28

Solaris に固有の問題　28

3.　Java プログラミング環境　31

Java プログラム　32

アプリケーションの例　33

アプレットの例　33

javald と再配置可能なアプリケーション　34

演算を並列的に処理する Java アプリケーションの作成　35

`thr_setconcurrency(3T)` の使用例　35

API の対応関係　38

スレッドグループメソッド　40

Java 開発ツール　41

Java WorkShop (JWS)　41

4.　推奨されないメソッド　43

「推奨されない」とは *　43

推奨されないスレッドメソッド　48

5.　アプリケーションのパフォーマンスチューニング　53

チューニングに関するヒント　53

システムインタフェース　53

はじめに

このマニュアルでは、Java™ 開発者向けに Solaris™ 2.6、Solaris 7、および Solaris 8 環境における Java の使用方法について説明します。具体的には、Solaris 上で Java を使用する場合の重要な構成要素の概要と説明、開発者にとっての利点、Solaris 上で最大のアプリケーションパフォーマンスを得られるようにする Java の使用方法が含まれます。また、このマニュアルでは、互換性についても説明しています。

対象読者

このマニュアルは、主に次の開発者を読者対象としています。

- Solaris 上で Java を初めて使用する開発者
- Java を初めて使用する開発者。このような開発者向けの情報には、アスタリスク (*) が付けられています。

内容の紹介

第 1 章「Java プログラミング環境の概要」では、このマニュアルの内容の概要を示します。

第 2 章「マルチスレッド」では、マルチスレッドの基礎と、Solaris 上でネイティブスレッド化された Java Virtual Machine (JVM) を使用する利点について説明します。

第 3 章「Java プログラミング環境」では、マルチスレッド化された Solaris 上で Java を使用する際のプログラミング環境について説明します。

第 4 章「推奨されないメソッド」では、JDK™ 1.1 では推奨されないメソッドを紹介しています。

第 5 章「アプリケーションのパフォーマンスチューニング」では、アプリケーションのパフォーマンスを向上させる方法について説明します。

関連マニュアル

Solaris オペレーティング環境上で使用する Java について、以下の URL を参照してください。

- `http://www.sun.com/solaris/java/`
- `http://dp-websvr.eng.sun.com/products/jpt/`

Sun のマニュアルの注文方法

専門書を扱うインターネットの書店 Fatbrain.com から、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、`http://www1.fatbrain.com/documentation/sun` の Sun Documentation Center をご覧ください。

Sun のオンラインマニュアル

`http://docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索をおこなうこともできます。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

- スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が、適宜併記されています。
- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、Cyrix が提供する互換マイクロプロセッサチップが含まれます。

第 1 章

Java プログラミング環境の概要

このマニュアルでは、Java の特徴を紹介するとともに、Solaris 2.6、Solaris 7、および Solaris 8 環境における Java アプリケーション開発に関する情報を提供します。

注 - Solaris 上で使用する Java の最新情報については、`www.sun.com/solaris/java/` を参照してください。

Java プログラミング環境と Java 実行時環境 (JRE)

この節では、Java と JRE (Java Runtime Environment) の基礎を紹介します。

Java プログラミング環境とは

Java は、クラスを基本単位とするオブジェクト指向の、新しく開発された並列型のプログラミング環境および実行環境です。Java は、次の要素から構成されています。

- プログラミング言語
- API 仕様
- 仮想マシン仕様

Java には、次の特徴があります。

- オブジェクト指向 – Java は、C++ の基本オブジェクト技術に多少の変更 (機能の拡張および削除) を加えたオブジェクト技術を提供します。
- アーキテクチャに依存しない – Java ソースコードをコンパイルすると、アーキテクチャに依存しないオブジェクトコードが生成されます。このオブジェクトコードは、ターゲットアーキテクチャ上の Java Virtual Machine (JVM) によって解釈されます。
- 移植性 – Java では、移植性の面での規格が強化されています。たとえば、int は常に 2 の補数の 32 ビット整数です。ユーザーインタフェースは、Solaris および他のオペレーティングシステムですぐに実装可能な抽象ウィンドウシステム (AWT、Abstruct Window Toolkit) を使用して構築されます。
- 分散型 – Java には、包括的な TCP/IP ネットワーク機能が実装されています。ライブラリルーチンは、HTTP (ハイパーテキスト転送プロトコル、HyperText Transfer Protocol) や FTP (ファイル転送プロトコル、File Transfer Protocol) などのプロトコルをサポートしています。
- 厳密な検査 – Java コンパイラおよび Java インタプリタは、包括的なエラー検査機能を提供します。すべての動的メモリーを管理し、配列境界などの例外を検査します。
- 安全性 – Java 言語には、C や C++ でしばしば見られるような不正なメモリーアクセスを引き起こす機能は実装されていません。Java インタプリタは、コンパイル後のコードをテストして、不正なコードがないかどうかを検査します。これらのテストによって、コンパイル後のコードがオペランドスタックのオーバーフローやアンダーフローを起こしたり、不正なデータ変換を行なったり、無効なオブジェクトフィールドにアクセスしたりすることがなくなり、すべての opcode のパラメータタイプが有効であることが検証されます。
- 高いパフォーマンス – アーキテクチャに依存しない、機械語に似た言語にプログラムをコンパイルするため、インタプリタでの処理サイズが小さく、効率的に実行されます。また、Java 環境において Java バイトコードは実行時にネイティブのマシンコードにコンパイルされます。
- マルチスレッド化 – Java 言語には、マルチスレッド機能が組み込まれています。マルチスレッド処理では、ユーザー操作の処理を継続しながら、イメージの読み込みなどの処理を行うことができるので、対話式処理のパフォーマンスを向上させることができます。
- 動的 – Java は、呼び出されたモジュールを実行時までリンクしません。
- 簡潔 – Java は C++ に似ていますが、C あるいは C++ の複雑な機能は取り除かれています。

Java には、以下の機能は提供されていません。

- プログラミングで制御する動的メモリー
- ポインタ演算
- `struct` (構造体)
- `typedef`
- `#define`

JRE の構成要素

JRE (Java Runtime Environment) は、一般的な JVM 実装用にコンパイルされたプログラムを実行できるソフトウェア環境です。実行時システムは、次のものから構成されます。

- Java プログラムの実行、ネイティブメソッドの動的リンク、メモリーの管理、例外処理に必要なコード
- JVM 実装

次の図は、JRE 全体と、一般的な JVM 実装の各種モジュールから構成される JRE の各要素とクラスライブラリとの関係を表しています。

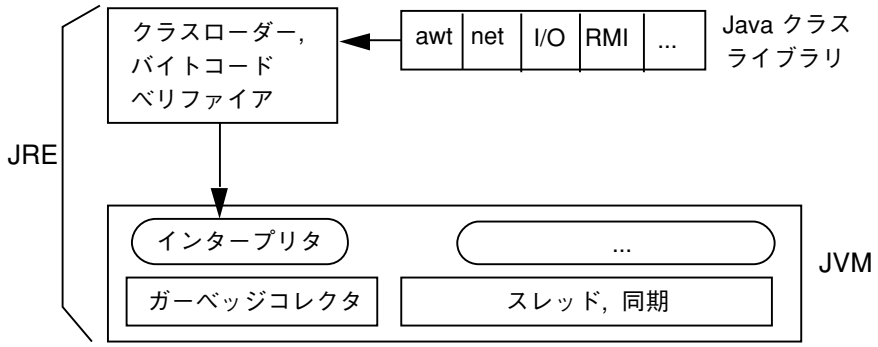

図 1–1　一般的な JVM 実装と JRE およびクラスライブラリとの機能関係

Java 仮想マシン (JVM)

JVM (Java Virtual Machine) は、1 つの命令セットを持ち、メモリーを使用する抽象的な演算マシンです。仮想マシンは、しばしばプログラミング言語の実装に使用されます。JVM は、Java プログラミング言語の基礎となるものです。異なるプラット

フォーム間で Java プログラムの移植が可能なこと、コンパイル後のコードサイズが小さいということは、JVM によって実現されます。

Solaris JVM は、Java アプリケーションの実行に使用されます。Java コンパイラ `javac` はバイトコードを生成し、そのコードを `.class` ファイルに書き込みます。JVM はこれらのバイトコードを解釈し、そのバイトコードはどの JVM 実装でも実行することができます。このようにして、異なるプラットフォーム間での Java の移植性が実現されます。図 1–2 および図 1–3 はそれぞれ、従来のコンパイル時環境と、Java が提供する移植可能な新しいコンパイル時環境を表しています。

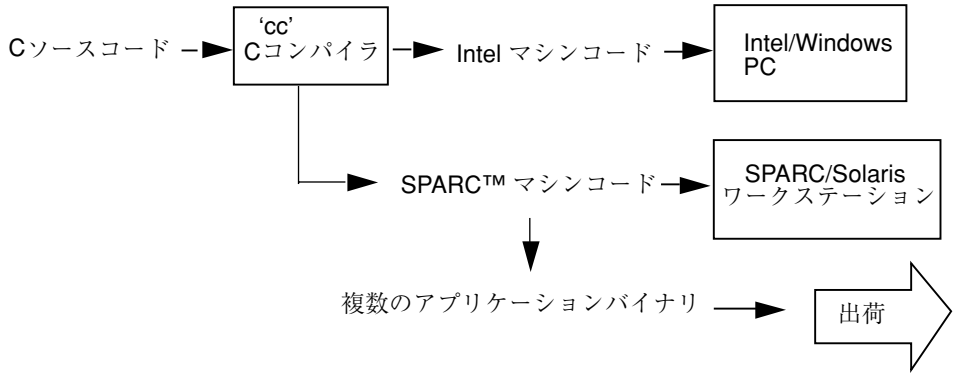

図 1–2　従来のコンパイル時環境

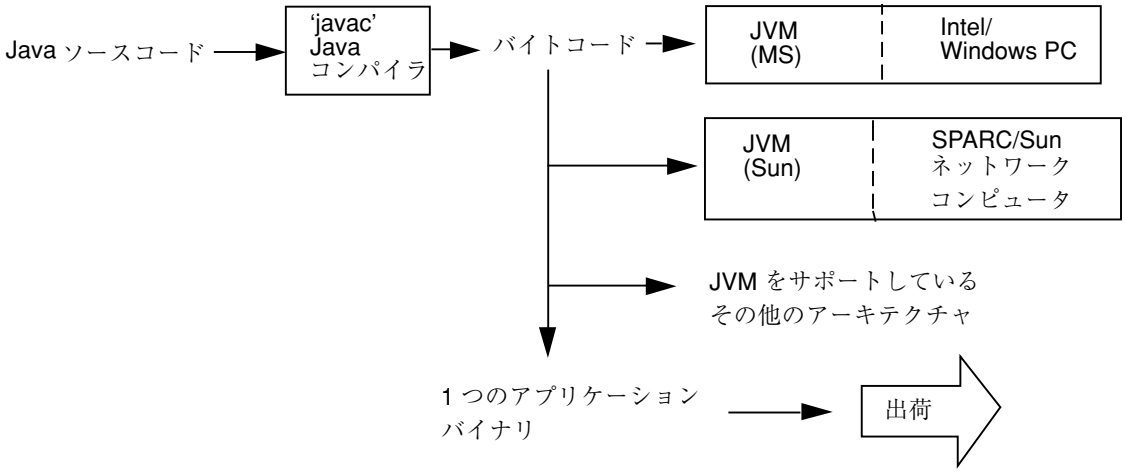

図 1–3　移植可能な新しい Java コンパイル時環境

マルチスレッド JVM

Java プログラミング言語では、マルチスレッドプログラムがサポートされる必要があります (第 2 章 を参照)。Java インタプリタはすべてマルチスレッドプログラミング環境を提供しますが、Java インタプリタの多くは、シングルプロセッサによるマルチスレッド処理しかサポートしないため、一度に実行される Java プログラムスレッド数は 1 つだけです。

Solaris JVM インタプリタは、Solaris のマルチスレッド機能を使用することによって、マルチプロセッサシステムを最大限に活用します。マルチプロセッサシステムでは、1 つのプロセスの複数のスレッドが複数の CPU で同時に実行されるようにスケジューリングできます。Solaris の JVM 上でマルチスレッド Java プログラムを実行すると、他のプラットフォームの JVM で同じプログラムを実行したときに比べて、大幅に並列性が向上します。

Sun の JIT コンパイラ

Solaris JVM に組み込まれている Sun の Java JIT (Just-In-Time) コンパイラは、従来より大幅に高速に動作し、特に長時間連続して演算中心の処理を実行するプログラムでは、大幅にパフォーマンスが向上します。

JIT コンパイラのコンパイルプロセス

JIT コンパイラ環境変数が有効になっている (デフォルト) の場合、JVM は `.class` ファイルを読み取って解釈し、JIT コンパイラに渡します。JIT コンパイラは、バイトコードを、ターゲットのプラットフォーム (プログラムを実行するプラットフォーム) にネイティブなコードにコンパイルします。次の図は、JIT コンパイラのコンパイルプロセスを表しています。

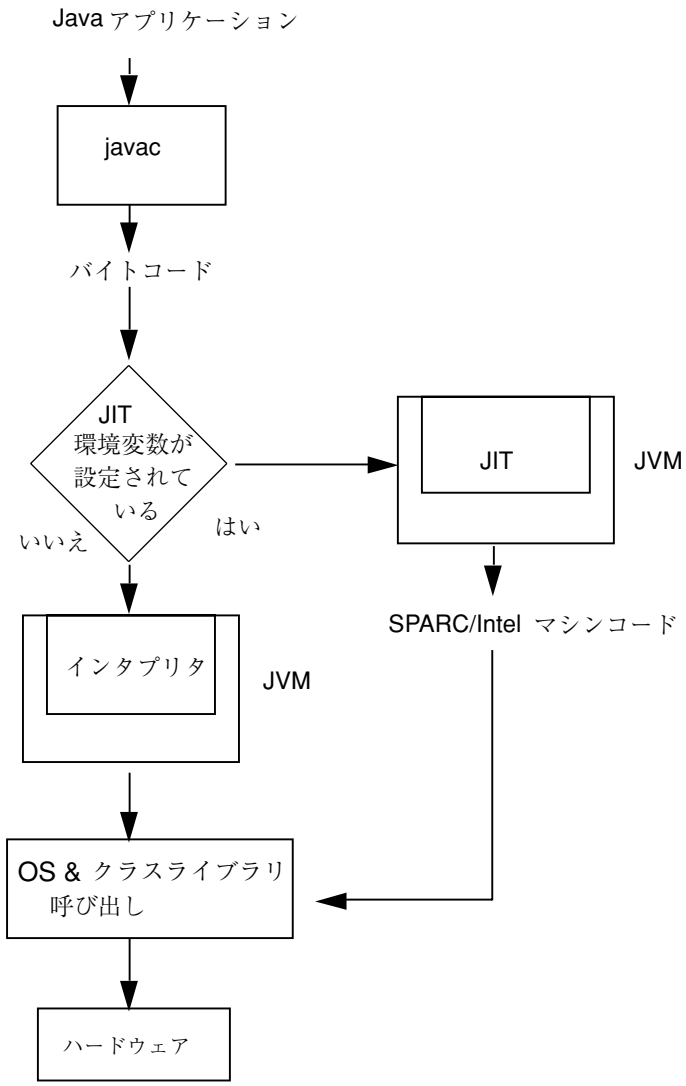

図 1–4　JIT コンパイラのコンパイルプロセス

次の図は、JIT コンパイラと Solaris JVM および JRE との機能関係を表しています。

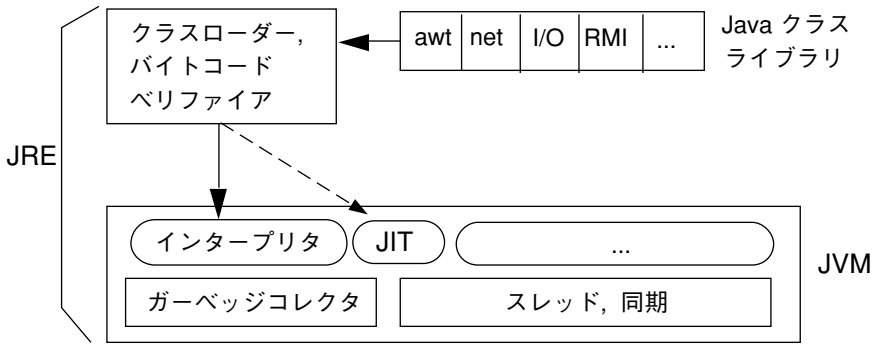

図 1–5　Solaris JVM と JIT との機能関係

第 2 章

マルチスレッド

この章では、マルチスレッドの概要、および Solaris 上の Java に固有のマルチスレッド、ネイティブスレッド JVM について説明します。

Java を初めて使用する開発者のための情報には、タイトルにアスタリスク (*) が付いています。

マルチスレッドの定義 *

スレッドは、プロセス内の制御シーケンス (一連の制御の流れ) です。単一スレッドのプロセスは、1 つの制御シーケンスに従って動作します。マルチスレッドのプロセスは、複数の制御シーケンスを持ち、独立した複数の動作を同時に扱うことができます。複数のプロセッサが使用できる場合、このような独立したそれぞれの動作を実際に並列に実行することができます。

Solaris 環境における従来の Java スレッド *

Soiaris 2.6 以前の従来の Java 実行時環境では、Java 実行時スレッドおよびシステムサポート層の一部である、「グリーンスレッド」と呼ばれるスレッドライブラリが使用されていました。このグリーンスレッドライブラリはユーザーレベルのライブラリであり、Solaris システムは一度に 1 つのグリーンスレッドしか処理できません。このため Solaris は「複数対単一」のスレッド方式で Java 実行時環境を処理す

るので (20ページの「「複数対単一」のモデル (グリーンスレッド)」を参照)、次のような問題がありました。

- Java アプリケーションと Solaris 環境の既存のマルチスレッドアプリケーションとを相互運用できない。
- Java スレッドはマルチプロセッサ上であっても並列動作できない。
- マルチスレッド Java アプリケーションが OS の並列処理機能を利用できないため、シングルプロセッサ、マルチプロセッサのどちらのシステムでも、アプリケーションの高速化を実現できない。

アプリケーションのパフォーマンスを大幅に向上させるため、Solaris 2.6 プラットフォームの Java では、グリーンスレッドライブラリが Solaris のネイティブスレッドに置き換えられています。これは、Solaris 7 および Solaris 8 でも同様です。

マルチスレッドの概念 *

マルチスレッドプログラミングを行うことにより、ソフトウェア開発者は、アプリケーションを高速化したり、ハードウェアを並列的に動作させたりオブジェクトを有効活用することができます。Solaris のマルチスレッドの実装を利用するので、Java は、効率的で信頼性が高く、標準に準拠し、開発者および一般ユーザーの両方に大きな利点をもたらします。マルチスレッドアプリケーションの開発において、Solaris オペレーティング環境は、最高のパフォーマンスとツール、サポート、柔軟性を提供します。Solaris オペレーティング環境には、次のような最新のマルチスレッド機能が採用されています。

- Solaris マルチスレッドカーネル – マルチスレッドアーキテクチャの完全な実装で基本となる構成要素
- 2 レベルのスレッドモデル – プロセスが使用するスレッド数の制限をなくして、最高のパフォーマンスを得られるようにする Solaris システムのマルチスレッド実装モデル
- POSIX の `pthreads` 規格 – IEEE POSIX 1003.1c 仕様に定義されているマルチスレッドインタフェースの実装
- Java スレッド API (Java API の 1 つ) – マルチスレッドアプリケーションを作成するための標準インタフェースになりつつある API

マルチスレッド化の利点 *

マルチスレッド化の大きな利点の 1 つとして、並列動作によってアプリケーションの実行が高速になることが挙げられます。

マルチスレッド化によって、ハードウェアの並列性を最大限に活用し、さらにサブシステムであるマルチプロセッサを効果的に使用できます。マルチスレッドは、対称型マルチプロセッサを十分に活用するために欠かせないものです。また、マルチスレッド化によって、演算や入出力などの処理の重複が少なくなるので、シングルプロセッサのシステムでもパフォーマンスが向上します。

マルチスレッド化の大きな利点として、次のような点を挙げることができます。

- スループットの向上。1 つのプロセス内で複数の演算と入出力要求を並列処理できます。
- 複数のプロセッサを同時に対称的に使用して、演算および入出力の処理を行うことができます。
- アプリケーションの応答性の向上。各要求ごとに別々のスレッドを使用できるので、アプリケーションが応答しなくなったり、マウスポインタの形が砂時計のまま元に戻らなくなることがなくなります。アプリケーション全体の処理が停止されたり、別の要求が完了するのを待たされたりすることがなくなります。
- サーバーの応答性の向上。大規模あるいは複雑な要求、あるいは低速のクライアントによって、他のサービス要求が待たされることがなくなります。サーバーの全体のスループットが大幅に向上します。
- 最小限のシステム資源の使用。スレッドがシステム資源に及ぼす負荷を最低限に抑えられます。スレッドでは、従来のプロセスに比べて、生成、保持、管理するために必要なオーバーヘッドが軽減されます。
- プログラム構造の簡略化。スレッドを使用すると、サーバークラスのアプリケーションやマルチメディアアプリケーションなどの複雑なアプリケーションの構造を簡略化できます。1 つ 1 つの処理を簡単なルーチンで作成できるので、複雑なプログラムの設計およびコーディングが簡単になり、より広範囲のユーザー要求に対して柔軟に対応できます。
- 通信の強化。スレッド同期機能を使用して、プロセス間の通信を強化できます。また、同じアドレス空間内でそれぞれ独立して動作するスレッド間で大量のデータを共有することにより、アプリケーション内のタスクどうしが高い帯域幅と短い応答時間で通信を行うことができます。

マルチスレッドモデル

大部分のマルチスレッドモデルは、以下のいずれかの形態で実装されています。

- 複数対単一
- 単一対単一
- 複数対複数

「複数対単一」のモデル (グリーンスレッド)

「複数対単一」のモデル (1 つのカーネルスレッドに対してユーザースレッドが複数のモデル) では、アプリケーションは、並列に動作できるスレッドをいくつでも作成できます。スレッドのすべての動作はユーザー空間のみに限定されます。カーネルにアクセスできるスレッドは一度に 1 つだけなので、スケジューリング可能な 1 つのエンティティ (実体) だけがオペレーティングシステムに認識されます。このため、「複数対単一」のマルチスレッドモデルでの並列処理は限定されたものになり、マルチプロセッサが有効に利用されることはありません。Solaris システムにおける Java スレッドの初期の実装は、次の図に示すような「複数対単一」のモデルでした。

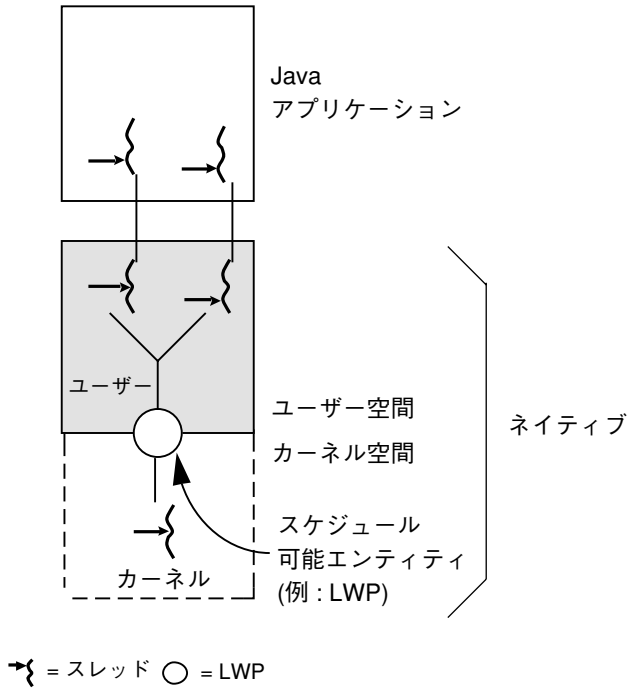

図 2–1　「複数対単一」のマルチスレッドモデル

「単一対単一」のモデル

「単一対単一」のモデル (1 つのカーネルスレッドに対してユーザースレッドが 1 つのモデル) は、正しいマルチスレッドの初期の実装の 1 つです。「単一対単一」のモデルでは、アプリケーションによって作成されたユーザーレベルのスレッドはそれぞれカーネルによって認識され、すべてのスレッドが同時にカーネルにアクセスできます。「単一対単一」モデルの最大の問題は、スレッドが増えるほどプロセスが重くなるため、開発者はそのことを考慮しながらスレッドをあまり多く使用しないようにする必要があることです。このため、Windows NT や OS/2 のスレッドパッケージなどの「単一対単一」モデルのスレッド実装の多くは、システムでサポートされるスレッド数を制限しています。

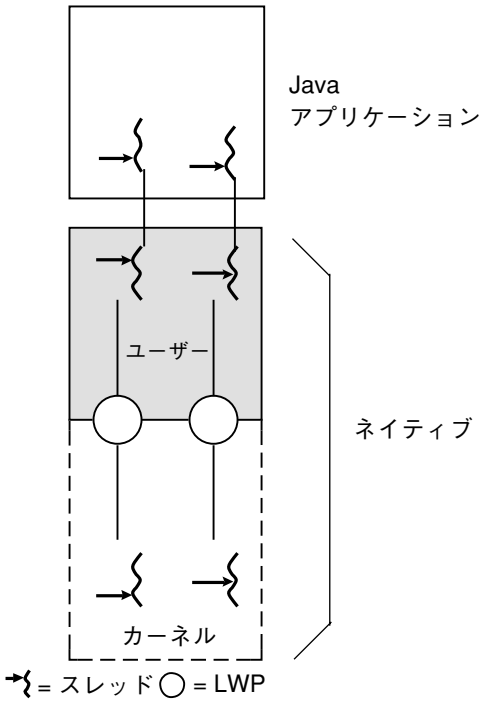

図 2–2　「単一対単一」のマルチスレッドモデル

「複数対複数」のモデル (Solaris 上の Java - ネイティブスレッド)

「複数対複数」のモデル (複数のカーネルスレッドに対してユーザースレッドが複数のモデル) は、「単一対単一」のモデルが持つ制限の多くを解消し、マルチスレッドの応用範囲を広げます。2 レベルモデルとも呼ばれる「複数対複数」のモデルは、各スレッドの負荷を軽減し、プログラミング作業も簡潔になります。

「複数対複数」のモデルでは、プログラムは、プロセスを重くしすぎることなく適切な個数のスレッドを持つことができます。ユーザーレベルのスレッドライブラリによって、カーネルスレッドの上位でユーザーレベルのスレッドをスケジューリングすることが可能になります。カーネルが管理する必要があるのは、アクティブになっているスレッドだけです。ユーザーレベルで「複数対複数」のモデルが実装されることにより、アプリケーションで効果的に使用できるスレッド数の制限がなくなるため、プログラミング作業が軽減されます。

つまり、標準のインタフェースを持つより簡単なプログラミングモデルが提供され、すべてのプロセスについて最高のパフォーマンスが得られるようになります。

Solaris 上の Java オペレーティング環境は、市販製品で初めてマルチスレッドオペレーティングシステムに「複数対複数」モデルの Java が実装された環境です。

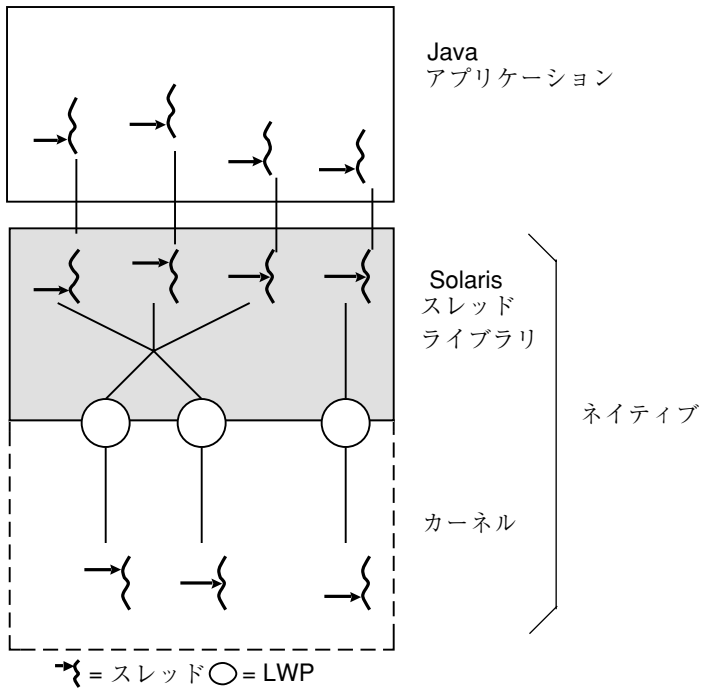

図 2–3　「複数対複数」のマルチスレッドモデル

マルチスレッドカーネル

マルチスレッドカーネルは、マルチスレッドの完全な実装の基礎となるものです。Solaris オペレーティングシステムで使用されているようなマルチスレッドカーネルでは、各カーネルスレッドはカーネルのアドレス空間内の 1 つの制御の流れです。カーネルスレッドは完全にプリエンプティブであり、システムで使用できるリアルタイムクラスなど任意のスケジューリングクラスを使用して、スケジューリングすることができます。あらゆる実行エンティティは、カーネルスレッドを使用して作成されます。カーネルスレッドは、カーネル内で完全にプリエンプティブな、リアルタイムの「核」とみなすことができます。

カーネルスレッドは、スレッドやプロセスが別の処理が完了するのを待っていたために優先順位どおりに実行されなくなるのを防ぐためのプロトコルをサポートす

る、同期プリミティブを使用します。これにより、アプリケーションは意図したとおりに動作するようになります。カーネルスレッドはまた、NFS デーモン、ページアウトデーモン、割り込み、などのカーネルレベルのタスクが非同期に動作することを可能にして、並列処理性および全体のスループットを向上させます。

マルチスレッドカーネルは、一般的な JVM 実装などのマルチスレッドアプリケーションアーキテクチャの作成に欠かせないものです。マルチスレッドカーネルには、次の特徴があります。

- マルチプロセッサの最高のパフォーマンスが得られるように、完全に対称型です。
- 複数のカーネルスレッドを使用して、マルチプロセッサマシン上での並列処理を可能にします。演算、ネットワーキング、表示、入出力などの処理を並列に実行することによって、ハードウェアサブシステムがより有効に活用されるようになります。
- 従来のアプリケーション (単一スレッドアプリケーション) もそのままマルチスレッドカーネル上で実行できます。
- 完全にプリエンプティブであり、リアルタイムの応答を実現します。

Solaris 上の Java マルチスレッドの利点

Solaris のマルチスレッドカーネルは、Solaris オペレーティングシステムの特に重要な構成要素の 1 つです。このマルチスレッドカーネルによって Solaris オペレーティングシステムは、効率的な並列処理を実現し、高機能を備えている、唯一の標準的なオペレーティングシステムになっています。

Solaris 上の Java は、カーネルのマルチスレッド機能を利用しています。開発者は、単純なプログラミングインタフェースを使用して、マルチプロセッサあるいはシングルプロセッサシステム用の、数千のユーザーレベルのスレッドを持つ Java アプリケーションを作成することができます。

Solaris 上の Java 環境は、「複数対複数」のスレッドモデルをサポートします。図 2–4 に示すように、Solaris の 2 レベルアーキテクチャは、軽量プロセス (LWP) と呼ばれる中間層を提供することによって、プログラミングインタフェースと実装とを分離します。LWP が提供されることにより、アプリケーション開発者は、汎用のアプリケーションレベルのインターフェースを使用して、高速で負荷が軽いスレッドを短期間に作成できます。スレッドを利用するようにアプリケーションを作成し

ておけば、実行時環境側でスレッドライブラリの実装に基づいて実行可能なスレッドを実行時資源 (LWP) に多重化し、スケジューリングしてくれます。

各 LWP は、コードやシステムコールを実行する仮想の CPU のような動作をします。LWP は、スケジューリングクラスと優先順位に従って、カーネルによって個々にディスパッチされる (振り分けられる) ため、独立にシステムコールを行なったり、独立にページフォルトを発生させたり、複数のプロセッサ上で並列的に動作したりすることができます。スレッドライブラリは、システムのスケジューラとは別のユーザーレベルのスケジューを提供します。ユーザーレベルのスレッドは、カーネルのスケジューリングが可能な LWP によってカーネル内でサポートされます。カーネルの LWP プール上に、多数のユーザースレッドが多重化されます。

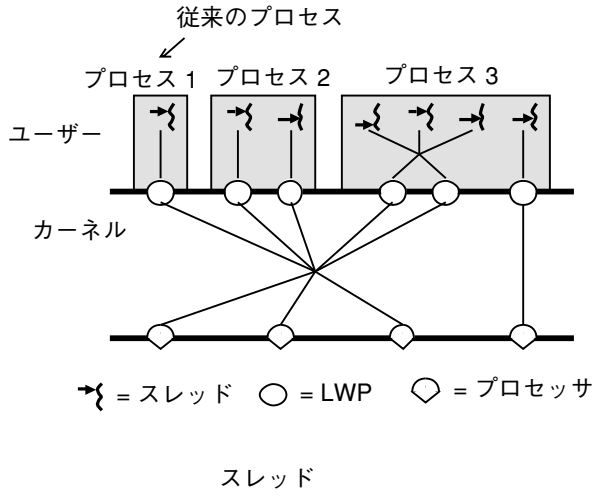

図 2–4　Solaris の 2 レベルアーキテクチャ

Solaris スレッドでは、ユーザーレベルのスレッドを LWP に結合するかあるいは LWP に結合しないままにするかを、アプリケーションが選択することができます。ユーザーレベルのスレッドと LWP とは、排他的に結合されます。スレッドの結合は、リアルタイムの応答を必要とするアプリケーションなど、自身の多重性 (並列処理) を厳密に制御する必要があるアプリケーションにとっては便利です。スレッド結合を行う Java API はありません。大部分の Java アプリケーションでは、スレッド結合は必要ないと考えられるためです。結合が必要な場合は、Solaris のネイティブメソッド呼び出しを行なって結合することができます。

このため、すべての Java スレッドはデフォルトでは非結合状態になっています。ユーザーレベルの非結合スレッドの並列処理は、スレッドライブラリが制御し

ます。スレッドライブラリは、アプリケーションが必要とする非結合スレッドに合わせて LWP プールを大きくしたり小さくしたりします。

Solaris 環境のすべての Java アプリケーションで、Solaris 上の Java スレッドに固有の次の機能をデフォルトで使用できます。

- Java スレッドは、Solaris の結合されていないスレッドと基本的には同じなので、結合されていないスレッドの利点を持っています。
- 複数のユーザーレベルのスレッドで 1 つの LWP を共有する機能
- 結合されていないスレッドに対する並列処理を自動的に制御し、アプリケーションの要求に合わせて、スレッドライブラリが LWP プールを大きくしたり小さくしたりします。詳細は35ページの「演算を並列的に処理する Java アプリケーションの作成」を参照してください。
- 極端に多くのシステム資源を消費したりシステムパフォーマンを低下させずに、きわめて軽量のユーザースレッドを大量に生成、使用、廃棄することができます。
- 同期プリミティブは、カーネルの関与を必要としないので、システム資源をいっさい消費しません。
- Solaris に固有のマルチスレッド機能。ネイティブメソッドを使用して利用できます。

注 - 一般的に、ネイティブスレッドを使用する Solaris のネイティブ機能を、Java アプリケーションが使用しないようにしてください。Java アプリケーションが Solaris プラットフォーム用に限定されてしまい、他のプラットフォームで動作しなくなる可能性があります。また、100% Pure Java™ にも準拠しなくなります。

Java アプリケーションから Solaris 固有の機能にアクセスすることは推奨できませんが、次に Solaris のマルチスレッドアーキテクチャの豊富な機能の例を紹介します。

- アプリケーションの並列処理をユーザーレベルあるいはシステムレベルで制御するための、結合または非結合スレッドを定義することができます。ネイティブスレッドを使用してのみ、結合 Java スレッドを作成できることに注意してください。
- アプリケーションは、プログラミングインタフェースを介して、アプリケーションの並列処理を制御できます。詳細は、35ページの「演算を並列的に処理する Java アプリケーションの作成」を参照してください。

- 1 つのプロセッサで実行し続けることができる LWP にユーザーレベルのスレッドを結合することができます (ネイティブメソッドを使用)。この機能は、マルチプロセッサシステムで動作するリアルタイムアプリケーションに便利です。
- プロセス間で有効な同期プリミティブ
- ファイルに書き込むことによって作成元のスレッドが終了した後にも残存する同期プリミティブ
- Java デーモンスレッドを直接ネイティブにサポートすることができます。デーモンスレッドは、バックグラウンドで動作し、独自に終了条件と動作を持つスレッドです。独自に終了条件と動作を持っているので、スレッドを使用するプロセスから独立して終了できます。デーモンスレッドは、アプリケーションとは無関係にスレッドを作成する必要があるライブラリに便利です。Solaris の JVM は、Java の直接ネイティブにサポートされているデーモンスレッドを活用するわけではありませんが、最終的には利用する場合もあります。

Solaris の 2 レベルモデルでは、多数の異なるプログラミング要件を満たすことができるよう、従来では見られないような高い柔軟性が提供されます。ウィンドウプログラムのように (少なくとも) 見かけ上は多くの要求を並列に処理する必要があるプログラもあれば、行列の乗算を行うプログラムのように並列演算を実際に使用可能な数のプロセッサに割り当てる必要のあるプログラムもあります。2 レベルモデルであることによって、カーネルは、システムサービスに対するスレッドからのアクセスをブロックしたり制限したりせずに、あらゆる種類のプログラムの並列処理要求に応えることができます。

Solaris 上の Java は、システム資源を効率的に使用するように設計されています。スレッドを使用することによるオーバーヘッドを最低限に抑えながら、アプリケーションに数千のスレッドを持たせることができます。スレッドはそれぞれ独立して動作し、同じプロセス中の他のスレッドとプロセス命令を共有したり、データを透過的に共有したりします。またスレッドは、プロセスのオペレーティングシステム状態の多くを共有するので、あるスレッドで開いたファイルを他のスレッドから読み込むこともできます。また程度の差はありますが、別々のプロセス間の同期をとることもできます。

Solaris のスレッドモデルに基づく Java は、速度、多重性、機能、カーネル資源の利用の面で、最良の組み合わせを提供します。

スレッドのグループ化

Java スレッドそれぞれは、スレッドグループを構成するメンバーです。スレッドグループは、複数のスレッドを別々に操作するのではなく、複数のスレッドを 1 つのオブジェクトに集めることによって、それらのスレッドをすべて一度にまとめて操作するためのものです。

たとえば、1 回のスレッド呼び出しによって、1 つのグループ内のすべてのスレッドを開始したり一時停止したりすることができます。Java スレッドグループは、`java.lang` パッケージ中の `ThreadGroup` クラスによって実装されています。実行時システムは、スレッドの構築中にそのスレッドをスレッドグループに追加します。スレッドを作成するときに、その新しいスレッドを適切なデフォルトのスレッドグループに追加するか、または明示的に新しいスレッドグループを指定することもできます。スレッドの作成時にいったんスレッドをスレッドグループに追加したら、そのスレッドを別のスレッドグループへ移動することはできません。

Java スレッドに関する注意事項

Solaris 用の Java アプリケーションを作成する場合の、Java 全般および Solaris に固有の Java スレッドに関する注意事項を説明します。

Java 全般の注意事項

JDK 1.1 で推奨されないメソッドについては、表 4–1を参照してください。

Solaris に固有の問題

以降で説明するように、いくつかの問題は Solaris に固有です。

マルチスレッドで安全でないライブラリの使用

注意 - 他に回避方法がない場合のみ、以下の方法を使用してください。正規の方法ではないので、十分に注意してプログラミングを行わないと、デッドロックが発生する可能性があります。

`-D_REENTRANT` を有効にせずにコンパイルされた既存のライブラリを使用する Java アプリケーションを実行しようとすると、以下のような問題が発生します。

JDK 1.1 のようなネイティブスレッドの JVM の場合、`libc` は、システムコールのエラーコードをスレッド固有の `errno` に書き込みます。`–D REENTRANT` フラグを有効にしてコンパイルされていないため、マルチスレッドで安全でないライブラリは、`errno` を参照するときにグローバル変数の `errno` を参照します。このため、ライブラリはスレッド固有の `errno` にアクセスすることができず、失敗したシステムコールに対応する正しい応答を返すことができません。

この問題を根本的に解決するには、ネイティブメソッドによってネイティブコードを使用するマルチスレッドの Java アプリケーションが、マルチスレッドで安全 (または少なくとも `errno` が安全) なライブラリとリンクされるようにするようにします。

`errno` が安全でないライブラリをどうしても参照する必要がある場合は、次のような回避方法があります。Java アプリケーションをメインスレッドで開始し、すべての安全でないライブラリに対する呼び出しがメインスレッド経由で行われるようにします。たとえばスレッドが JNI を呼び出す場合、JVM を使って、メインスレッドによって処理される 1 つの待ち行列に、すべての JNI 引数を整列化して追加することができます。このようにして、Java ネイティブインタフェース (JNI、Java Native Interface) を呼び出すスレッドは、自分の代わりにメインスレッドが呼び出しを実行してその結果を返してくれるのを待ちます。

安全でないライブラリに対する呼び出しがメインスレッドからのみ実行される時には、ライブラリ中でも単一スレッドによって処理が行われるので、ロックによって排他処理を行う必要はありません。メインスレッドである程度の並列処理を実現するために、ブロックされない (入出力処理の完了を待たずに復帰する) 呼び出しを実行する場合もありますが、メインスレッドから参照されるのはグローバル変数の `errno` なので、`libc` とマルチスレッドで安全でないライブラリの両方も同じ `errno` を参照することになります。

interrupt() メソッド

このメソッドには現在のところ特別に便利な機能はないので、一般的には推奨していません。Java 言語仕様 (JLS、Java Language Specification) では、対象のスレッドが wait() メソッドを呼び出しているときだけその対象スレッドに割り込む、と定義されています。

Solaris プラットフォームでは、対象のスレッドが入出力の呼び出しを行なっている時にも割り込むように、このメソッドの動作が拡張されていますが、interrupt() メソッドのこの動作に依存しないようにしてください。この拡張された動作は将来サポートされなくなる可能性があり、また異なる JVM 間でコードの互換性がなくなるためです。

スレッドの優先順位

ネイティブスレッド化された JVM での Java スレッドでは、スレッドに優先順位を付けることができますが、スケジューラはこの値を単なるヒントとして扱います。特に、演算処理が中心のスレッドがある場合には、スレッドがその優先順位どおりに実行されない可能性があります。通常、1 つのプロセスについて利用できる複数のプロセッサは動的で予測できないため、スレッドに優先順位を付けることによって、マルチタスクのマルチプロセッサシステム上での処理をスケジューリング しようとしても、あまりうまくいきません。

第 3 章

Java プログラミング環境

Solaris JVM では、任意のテキストエディタと make(1S)、さらに次の構成要素を使用して、Java プログラミングを行うことができます。

表 3–1 Java プログラミング環境の構成要素

構成要素	説明
javac	Java コンパイラ。Java ソースコードのファイル (*name*.java) を、Java インタプリタ java(1) が処理できるバイトコードのファイル (*name*.class) に変換します。Java アプリケーションと Java アプレットの両方がコンパイルされます。
javald	ラッパージェネレータ。Java アプリケーションのコンパイルと実行に必要な環境情報を収集するラッパーを作成します。ラッパーが呼び出されるまで、指定されたパスが結合されないため、JAVA_HOME と CLASSPATH の再配置が可能です。
java	Java インタプリタ。コマンドとして呼び出され Java アプリケーションを実行したり、HTML コードを介してブラウザから呼び出されてアプレットを実行したりすることができます。
appletviewer	Java アプレットビューア。指定されたドキュメントや資源を表示したり、ドキュメントによって参照されるアプレットを実行したりします。
javap	Java クラスファイル逆アセンブラ。javac によってコンパイルされたバイトコードのクラスファイルを逆アセンブルし、その結果を stdout に出力します。

make(1S) については、『プログラミングユーティリティ』の「make ユーティリティ」の章を参照してください。

一般的な Java 環境変数は次のとおりです。

表 3–2 Java 環境変数

変数	説明
JAVA_HOME	Java ソフトウェアのベースディレクトリのパス。たとえば、$JAVA_HOME/bin には、javac、 java、 appletviewer、 javap、 javah がすべて格納されています。Solaris JVM を使用するために設定する必要はありません。
CLASSPATH	コンパイルによって生成された *.class ファイルを含むディレクトリへのパスをコロン (:) で区切ったリスト。javac、 java、 javap、 javah によって使用されます。この環境変数が設定されていない場合、すべての Solaris JVM 実行ファイルは /usr/java/lib/classes.zip をデフォルトのクラスパスとみなします。Solaris JVM を使用するために設定する必要はありません。
PATH	通常の実行ファイル検索パスのリスト。$JAVA_HOME/bin を含めることができます。

注 - JVM ツールは /usr/java/bin に格納され、各実行ファイルへのシンボリックリンクは /usr/bin に格納されます。このため、新たにインストールした JVM パッケージを使用するために、ユーザーの PATH 変数にパスを追加する必要はありません。また、すべての Solaris JVM 実行ファイルは、標準の Java クラスライブラリを検索するデフォルトのパスを、/usr/java/lib/classes.zip とみなします。

基本 Java プログラミング環境には、デバッガはありません。Java プログラム用のデバッガは、Solaris には同梱されていない Java WorkShop™ に含まれています。

Java プログラム

Java プログラムは、アプリケーションとアプレットの 2 つの形態に分類されます。

Java アプリケーションは、コマンド行から Java インタプリタを起動し、コンパイル済みのアプリケーションを含むファイルを指定することによって実行されます。

Java アプレットは、ブラウザから実行されます。ブラウザが解釈する HTML コードには、コンパイル済みのアプレットを含むファイル名を指定します。ブラウザはそのアプレットを読み込み実行するために、Java インタプリタを呼び出します。

アプリケーションの例

例 3–1 は、stdout に「Hello World」と表示するアプリケーションのソースです。メソッドは、呼び出し時に引数を受け取りますが、引数に対して処理は行いません。

例 3–1 Java アプリケーションの例

```
//
// HelloWorld アプリケーション
//
class HelloWorldApp{
    public static void main (String args[]) {
        System.out.println ("Hello World");
    }
}
```

C と同様に、Java では最初に呼び出すメソッドまたは関数が main と認識されることに注意してください。キーワード public は、そのメソッドが誰でも実行できるようにします。static は、main が HelloWorldApp クラスを参照するようにします。つまり、static はこのクラスのインスタンスに 1 つだけ存在します。void は、main が何も返さないことを示します。args[] は、String 型の配列を宣言しています。

このアプリケーションをコンパイルするには、次のようにコマンドを実行します。

```
$ javac HelloWorldApp.java
```

アプリケーション実行するには、次のようにコマンドを実行します。

```
$ java HelloWorldApp arg1 arg2 ...
```

アプレットの例

例 3–2 は、例 3–1 のアプリケーションと同じように動作するアプレットのソース例です。

例 **3–2**　Java アプレットの例

```
//
// HelloWorld アプレット
//
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorld extends Applet {
    public void paint (Graphics g) {
        g.drawString ("Hello World", 25, 25);
    }
}
```

アプレットでは、参照先クラスを必ず明示的にインポートする必要があります。キーワード public と void は、アプリケーションの場合とそれぞれ同じ意味を持ちます。extends は、HelloWorld クラスが Applet クラスからの継承であることを示します。

このアプレットをコンパイルするには、次のようにコマンドを実行します。

```
$ javac HelloWorld.java
```

アプレットは、HTML コードを使用してブラウザ中に呼び出されます。アプレットを実行するのに最小限必要な HTML ページの構成は次のとおりです。

```
<title>Test</title>
<hr>
<applet code="HelloWorld.class" width=100 height=50>
</applet>
<hr>
```

javaId と再配置可能なアプリケーション

多くの場合、Java アプリケーションが正しく実行されるかどうかは、JAVA_HOME、CLASSPATH、LD_LIBRARY_PATH 環境変数の設定に影響されます。これらの環境変数は、ユーザーが任意のパスを設定できるため、適切でないディレクトリがパス中に設定されている可能性もあります。一般的に、アプリケーションは CLASSPATH 変数に固有の値を必要とします。

javald(1) は、Java アプリケーションのラッパーを生成するコマンドです。ラッパーは、JAVA_HOME、CLASSPATH、LD_LIBRARY_PATH 環境変数のいずれかあるいはすべてに、正しいパスを設定できます。ただし、この設定がこれらの環境変数のユーザー設定値に影響することはありません。Java アプリケーションの実行中は、ラッパーによる設定が、ユーザーの設定よりも優先されます。ラッパーは、Java アプリケーションが実際に実行されるまで指定されたパスを割り当てないので、アプリケーションを再配置しやすくなります。

演算を並列的に処理する Java アプリケーションの作成

注意 - 通常は、ネイティブメソッドを使用して、thr_setconcurrency(3T) などの Solaris 固有の機能にアクセスしないようにしてください。アプリケーションが Solaris 用のみに限定され、100% Pure Java に準拠しなくなってしまうためです。

注 - Java アプリケーションで thr_setconcurrency (3T)が必要になる場合はほとんどありません。必要になる場合としては、たとえば、ダミースレッドを使用して実行を繰り返すデモアプリケーション、あるいは行列の乗算や並列化されたグラフィックス計算などの演算アプリケーションなどが考えられます。

thr_setconcurrency(3T) の使用例

並列化された行列の乗算などの演算アプリケーションでは、ネイティブメソッドを使用して、thr_setconcurrency(3T) を呼び出す必要があります。これは、Java アプリケーションから同時に使用可能な資源を十分に使用して、複数のプロセッサを活用できるようにするためです。この方法は、大部分の Java アプリケーションやアプレットには必要ありません。次に、ネイティブメソッドを使用して thr_setconcurrency(3T) を呼び出す例を示します。

1 つ目の要素は、MPtest_NativeTSetconc() を使用する Java アプリケーション、MPtest.java です。このアプリケーションは、1 行ずつ識別情報を表示し、10,000,000 回ループして演算処理をシミュレートするスレッドを 10 個作成します。

例 **3–3** MPtest.java

```
import java.applet.*;
import java.io.PrintStream;
import java.io.*;
import java.net.*;

class MPtest {
   static native void NativeTSetconc();
   static public int THREAD_COUNT = 10;
   public static void main (String args[]) {
      int i;

// sysconf (_SC_NPROCESSORS_ONLN) に
// Solaris での並列処理を設定
      NativeTSetconc();
// スレッドを開始
      client_thread clients[] = new client_thread[ THREAD_COUNT ];
      for ( i = 0; i < THREAD_COUNT; ++i ){
         clients[i] = new client_thread(i, System.out);
         clients[i].start();
      }
   }

   static { System.loadLibrary("NativeThreads");
}
class client_thread extends Thread {
   PrintStream out;
   public int LOOP_COUNT = 10000000;
   client_thread(int num, PrintStream out){
      super( "Client Thread" + Integer.toString( num ) );
      this.out = out;
      out.println("Thread " + num);
   }
   public void run () {
      for( int i = 0; i < this.LOOP_COUNT  ; ++i ) {;
    }
  }
}
```

2 つ目の要素は、javah(1) ユーティリティを使用して、MPtest.java から作成した C スタブファイルの MPtest.c です。このファイルを作成するには、次のようにコマンドを実行します。

```
% javac MPtest.java
% javah -stubs MPtest
```

3 つ目の要素は、同じく javah(1) ユーティリティを使用して、MPtest.java から作成した C ヘッダーファイルの MPtest.h です。

```
% javah MPtest
```

4 つ目の要素は、C ライブラリインタフェースへの呼び出しを行う C 関数の NativeThreads.c です。

```
#include <thread.h>
#include <unistd.h>
#include <jni.h>
JNIEXPORT void JNICALL Java_MPtest_NativeTSetconc(JNIEnv *env, jclass obj) {
  thr_setconcurrency(sysconf(_SC_NPROCESSORS_ONLN));
}
```

次のようなメークファイルを使用すると、4 つのファイルから、MPtest.class という Java アプリケーションを簡単に作成することができます。

例 3–4 MPtest.class を生成する makefile

```
# make は次の処理を行います。
# 1. make MPtest を実行
# 2. thr_setconcurrency(_SC_NPROCESSORS_ONLN) へのネイティブ呼び出しを
#    取り込むために NativeThreads.c を作成
# 3. make lib を実行
#
# これで、LD_LIBRARY_PATH および CLASSPATH を . に設定して
#、java MPtest を実行できるようになる

JAVA_HOME=/usr/java
JH_INC1=${JAVA_HOME}/include
JH_INC2=${JAVA_HOME}/include/solaris

MPtest:
  ${JAVA_HOME}/bin/javac MPtest.java
  (CLASSPATH=. ; export CLASSPATH ; ${JAVA_HOME}/bin/javah -stubs MPtest)
  (CLASSPATH=. ; export CLASSPATH ; ${JAVA_HOME}/bin/javah -jni MPtest)

lib:
  cc -G -I${JH_INC1} -I${JH_INC2} NativeThreads.c -lthread \
  -o libNativeThreads.so

clean:
  rm -rf *.class libNativeThreads.so *.h MPtest.c
```

API の対応関係

表 3–3 は、Java スレッド API と Solaris および POSIX スレッド API をできるかぎり対応づけたものです。Java スレッド API と Solaris スレッド API および POSIX スレッド API との完全な対応を示すものではありません。また、この表を参考にすれば、Solaris または POSIX スレッドプログラムを簡単に Java スレッドプログラムに、あるいは Java スレッドプログラムを Solaris スレッドプログラムまたは POSIX スレッドプログラムに簡単に変換できるというわけでもありません。表 3-3 は、単に API 間のおおよその対応を示すだけです。3 つの API のうちのいずれかに関する知識があり、各 API 間の対応を確認するときのガイドラインとして利用してください。C の手続き型および階層プログラミングで Solaris API を使用することと、Java のオブジェクト指向のプログラミング手法で Solaris API を使用することの間には、概念上の違いがあります。

次に、Java スレッド API と Solaris スレッド API とは完全には対応していない理由を示す例を紹介します。

- 表 3–3 では、Java スレッドの破壊スレッド (`Destroy()`) は、POSIX の `pthread_cancel()` に対応しています。ただし、POSIX の `pthread_cancel()` は、取り消し点の概念が存在し、かつ `pthread_cleanup_push()` と `pthread_cleanup_pop()` を使用して、取り消し点のあたりでクリーンアップハンドラを作成することによって初めて完全になります。Java スレッド API には、スレッドの破壊について同様の概念はありません。この意味で、これら 2 つの破壊手法はまったく異なります。

注 - JDK 1.1 では、`Destroy()` メソッドは推奨されていません。

- 表 3–3 では、Java スレッドの `interrupt()` メソッドは POSIX の `pthread_kill()` に対応していますが、両者はまったく異なります。Java には安全な割り込み点の概念がありますが (たとえば `wait()`)、POSIX にはそのような概念はありません。

注 - Java には、Solaris の読み取り/書き込みロックインタフェースや POSIX の属性に相当するインタフェースはありません。

表 3–3　Java API と Solaris および POSIX API の対応

Java スレッド **API**	**Solaris** スレッド **API**	**POSIX** スレッド **API**
	thr_create()	pthread_create()
activeCount()		
checkAccess()		
countStackFrames()		
currentThread()	thr_self()	pthread_self()
destroy()		pthread_cancel()
dumpStack()		
enumerate()		
getName()		
getPriority()	thr_getprio()	pthread_ getschedparam()
getThreadGroup()		
interrupt()	thr_kill()	pthread_kill()
interrupted()		
isAlive()		
isDaemon()		
isInterrupted()		
join()	thr_join()	pthread_join()
resume()	thr_continue()	
run()		
setDaemon()	THR_DAEMON フラグ	
setName()		
setPriority()	thr_setprio()	pthread_ setschedparam()
sleep()	sleep()	sleep()
start()		
stop()		
suspend()		
同期メソッド		
wait()	cond_wait()	pthread_cond_wait()

表 3–3　Java API と Solaris および POSIX API の対応　続く

Java スレッド **API**	**Solaris** スレッド **API**	**POSIX** スレッド **API**
`notify()`	`cond_signal()`	`pthread_cond_signal()`
synchronized メソッド synchronized 文	mutex_*name*()	pthread_mutexe_*name*()

スレッドグループメソッド

次に示すメソッドは、スレッドグループに作用します。Java には、スレッドグループ機能がありますが、Solaris や POSIX にはありません。

- `activeCount()`
- `activeGroupCount()`
- `allowThreadSuspension()`
- `checkAccess()`
- `getMaxPriority()`
- `getParent()`
- `getName()`
- `isDaemon()`
- `list()`
- `parentOf()`
- `resume()`
- `setDaemon()`
- `stop()`
- `suspend()`
- `toString()`
- `uncaughtException()`

Java 開発ツール

以降の節では、Java 開発ツールについて説明します。

Java WorkShop (JWS)

Java WorkShop (JWS) は、Java 開発者向けの強力なビジュアル開発ツールです。Java アプレットやアプリケーションを短時間に簡単に作成するための使いやすいツール群で構成されています。

JWS では専用の Java インタプリタが使用されます。JWS は、表 3–4 に示すアプリケーションで構成されています。

表 3–4 Java WorkShop アプリケーション

アプリケーション	説明
ポートフォリオマネージャ	Java プロジェクトのポートフォリオを作成したりカスタマイズしたりします。新しいアプレットやアプリケーションを作成する元となるオブジェクトやアプレットを管理します。
プロジェクトマネージャ	プロジェクトのディレクトリなどの各種設定を行なったり、保存したりします。これによって、開発者はプロジェクの各構成要素へのパスを覚えておかなくても済みます。
ソースエディタ	ソースコードを作成し編集するためのポイント&クリック方式のツールです。作成、コンパイル、デバッグ中に、Java WorkShop の他のアプリケーションと連動して呼び出されることもあります。
構築ツール	Java ソースコードを Java バイトコードにコンパイルし、ソースコード内のエラーを検出します。構築ツールが検出したエラーとリンクさせてソースエディタを起動するので、開発者はすばやくエラーを修正して再コンパイルすることができます。
ソースブラウザ	プロジェクト中のすべてのオブジェクトのクラス継承情報を示すツリー図を表示します。また、プロジェクト中のすべてのコンストラクタと一般的なメソッドを一覧表示し、文字列やシンボルを検索することができます。ソースエディタにリンクしてソースコードを表示することができます。

表 3–4　Java WorkShop アプリケーション　続く

アプリケーション	説明
デバッガ	デバッグプロセスを制御し管理します。作成したアプリケーションまたはアプレットをコントロールパネルを使用して実行することによって、スレッドの一時停止や実行再開、ブレークポイントの設定、例外のトラップ、スレッドのアルファベット順表示、メッセージの確認、などを行うことができます。
プロジェクトテスタ	アプレットビューアと同様に、アプレットを実行して動作をテストすることができます。構築ツールを使用してアプレットをコンパイルしてから、プロジェクトテスタでアプレットを実行してください。
GUI ビルダー	ポイント&クリック式のユーザーインタフェースを使用して Java の GUI アプリケーションを作成することができる、統合 GUI ビルダーです。カスタマイズ可能な GUI ウィジェットのパレットが提供されています。
オンラインヘルプ	プロジェクトおよびポートフォリオ、ソースコードの編集、プロジェクトの構築やデバッグ、ブレークポイント、GUI ビルダー、などについての説明のほかに、Java WorkShop の基本的な使い方をひととおり学習できるチュートリアルが含まれています。目次や索引もあります。

JWS についての詳細は、`http://www.sun.com/workshop/java/` または `http://www.sun.co.jp/workshop/jws/` を参照してください。

第 4 章

推奨されないメソッド

この章では、JDK 1.1 で推奨されないメソッドについて、およびその一覧を示します。

「推奨されない」とは *

あるメソッドが重要とみなされなくなったためにクラスから削除される可能性があるので使用すべきではないとき、そのメソッドは推奨されません。こうしたことが起きるのは、クラスに変更が加えられていくと、メソッド名が変更されたり、新しいメソッドが追加されたり、属性が変更されるなど、そのクラスの API が変更されるためです。新しい API への移行を促すため、推奨されないクラスおよびメソッドのマニュアル (API ドキュメント) には、コメントとして「@deprecated」という印が付けられています。次の表に、推奨されないメソッドの一覧を示します。

表 4–1 推奨されないメソッド

クラス	メソッド	推奨するメソッドまたは代替法
java.awt.BorderLayout	addLayoutComponent()	addLayoutComponent(component,object)
java.awt.CardLayout	addLayoutComponent()	addLayoutComponent(component,object)
java.awt.CheckboxGroup	getCurrent()	getSelectedCheckbox()
	setCurrent()	setSelectedCheckbox()
java.awt.Choice	countItems()	getItemCount()
java.awt.Component	getPeer()	

表 4–1　推奨されないメソッド　続く

	enable()	setEnabled(true)
	disable()	setEnabled(false)
	show()	setVisible(true)
	hide()	setVisible(false)
	location()	getLocation()
	move()	setLocation()
	size()	getSize()
	resize()	setSize()
	bounds()	getBounds()
	reshape()	setBounds()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
	layout()	doLayout()
	inside()	contains()
	locate()	getComponentAt()
	deliverEvent()	dispatchEvent()
	postEvent()	dispatchEvent()
	handleEvent()	processEvent()
	mouseDown()	processMouseEvent()
	mouseDrag()	processMouseMotionEvent()
	mouseUp()	processMouseEvent(MouseEvent)
	mouseMove()	processMouseMotionEvent()
	mouseEnter()	processMouseEvent()
	mouseExit()	processMouseEvent()
	keyDown()	processKeyEvent()
	keyUp()	processKeyEvent()
	action()	コンポーネント破棄アクションイベントに対する `ActionListener` として登録してください。
	gotFocus()	processFocusEvent()
	lostFocus()	processFocusEvent()
	nextFocus()	transferFocus()

表 4–1　推奨されないメソッド　続く

java.awt.Container	countComponents()	getComponentCount()
	insets()	getInsets()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
	deliverEvent()	dispatchEvent()
	locate()	getComponentAt()

表 4–2　推奨されないメソッド

クラス	メソッド	推奨するメソッドまたは代替法
java.awt.FontMetrics	getMaxDescent()	getMaxDescent()
java.awt.Frame	setCursor()	Component 中の setCursor() メソッド
	getCursorType()	Component 中の getCursor() メソッド
java.awt.Graphics	getClipRect()	getClipBounds()
java.awt.List	countItems()	getItemCount()
	clear()	removeAll()
	isSelected()	isIndexSelected()
	allowsMultipleSelections()	isMultipleMode()
	setMultipleSelections()	setMultipleMode()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
	delItems()	`public` としては使用しないでください。`package private` 用に残されています。
java.awt.Menu	countItems()	getItemCount()
java.awt.MenuBar	countMenus()	getMenuCount()
java.awt.MenuComponents	getPeer()	
	postEvent()	dispatchEvent()
java.awt.MenuContainer	postEvent()	dispatchEvent()
java.awt.MenuItem	enable()	setEnabled(true)
	disable()	setEnabled(false)
java.awt.Polygon	getBoundingBox()	getBounds()

表 4–2　推奨されないメソッド　続く

	inside()	contains()
java.awt.Rectangle	reshape()	setBounds()
	move()	setLocation()
	resize()	setSize()
	inside()	contains()
java.awt.ScrollPane	layout()	doLayout()
java.awt.Scrollbar	getVisible()	getVisibleAmount()
	setLineIncrement()	setUnitIncrement()
	getLineIncrement()	getUnitIncrement()
	setPageIncrement()	setBlockIncrement()
	getPageIncrement()	getBlockIncrement()
java.awt.TextArea	insertText()	insert()
	appendText()	append()
	replaceText()	replaceRange()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
java.awt.TextField	setEchoCharacter()	setEchoChar()
	preferredSize()	getPreferredSize()
	minimumSize()	getMinimumSize()
java.awt.Window	postEvent()	dispatchEvent()
java.io.ByteArrayOutputStream	toString()	toString(String enc) または toString()。これらは、プラットフォームのデフォルトの文字コードを使用します。
java.io.DataInputStream	readLine()	BufferedReader.readLine()
java.io.PrintStream	printStream()	PrintWriter class
java.io.StreamTokenizer	streamTokenizer()	入力ストリームを文字ストリームに変換してください。
java.lang.Character	isJavaLetter()	isJavaIdentifierStart(char)
	isJavaLetterOrDigit()	isJavaIdentifierPart(char)
	isSpace()	isWhitespace(char)
java.lang.ClassLoader	defineClass()	defineClass(java.lang.String,byte[],int,int)

表 4–2　推奨されないメソッド　続く

java.lang.Runtime	getLocalizedInputStream()	InputStreamReader および BufferedReader クラス
	getLocalizedOutputStream()	OutputStreamWriter、BufferedWriter、PrintWriter クラスを使用してください

表 4–3　推奨されないメソッド

クラス	メソッド	推奨するメソッドまたは代替法
java.lang.String	string()	文字コード名を取得するかまたはデフォルトの文字コードを使用する、`String` コンストラクタを使用してください。
	getBytes()	getBytes(String enc)または getBytes()
java.lang.System	getenv()	java.lang.System.getProperty メソッドのシステムプロパティおよび Boolean, Integer, または Long プリミティブ型の対応する get TypeName メソッドを使用してください。
java.lang.Thread	resume()	48ページの「推奨されないスレッドメソッド」 を参照。
java.lang.Thread	stop()	48ページの「推奨されないスレッドメソッド」 を参照。
java.lang.Thread	suspend()	48ページの「推奨されないスレッドメソッド」 を参照。
java.util.Date	getYear()	Calendar.get(Calendar.YEAR)-1900
	setYear()	Calendar.set(Calendar.YEAR+1900)
	getMonth()	Calendar.get(Calendar.MONTH)
	setMonth()	Calendar.set(Calendar.MONTH,int month)
	getDate()	Calendar.get(Calendar.DAY_OF_MONTH)
	setDate()	Calendar.set(Calendar.DAY_OF_MONTH, int date)
	getDay()	Calendar.get(Calendar.DAY_OF_WEEK)
	getHours()	Calendar.get(Calendar.HOUR_OF_DAY)
	setHours()	Calendar.set(Calendar.HOUR_OF_DAY,int hours)
	getMinutes()	Calendar.get(Calendar.MINUTE)

表 4–3　推奨されないメソッド　続く

	setMinutes()	Calendar.set(Calendar.MINUTE,int minutes)
	getSeconds()	Calendar.get(Calendar.SECOND)
	setSeconds()	Calendar.set(Calendar.SECOND,int seconds)
	parse()	DateFormat.parse(String s)
	getTimezoneOffset()	Calendar.get(Calendar.ZONE_OFFSET)+ Calendar.get(Calendar.DST_OFFSET)
	toLocaleString()	DateFormat.format(Date date)
	toGMTString()	DateFormat.format(Date date) (GMT タイムゾーンを使用)
	UTC()	Calendar.set(year+1900,month,date,hrs, min,sec) または GregorianCalendar(year+1900,month, date,hrs,min,sec) (UTC タイムゾーンを使用)、その後に Calendar.getTime().getTime()

推奨されないスレッドメソッド

Thread.stop、Thread.suspend、Thread.resume メソッドは、JDK 1.1 では推奨されません。Thread.stop は、本質的に安全でないためです。スレッドを停止すると、スレッドがロックしたモニターをすべてロック解除してしまいます。ThreadDeath 例外がスタックを上に移動した場合に、モニターのロックは解除されます。これらのモニターによって以前に保護されたオブジェクトが不整合の状態になっている場合、その他のスレッドはこれらのオブジェクトが不整合の状態にあるとみなします。そのようなオブジェクトは破損しているとみなされます。

破損しているオブジェクトに対して動作するスレッドは、明示的または非明示的に不特定に動作します。検査が行われない他の例外とは異なり、ThreadDeath はこのようなスレッドを警告メッセージを表示せずに停止します。このように、プログラムが破損しているということを示す警告が、ユーザーには何も通知されません。その後予期しない時に、プログラムが破損していることがわかります。

Thread.stop を、より安全にスレッドを終了させるコードに置き換えてください。ほとんどの stop() は、対象となるスレッドの実行を停止するかどうかを判

定する変数を変更するコードに置き換えることができ、また置き換える必要があります。そのスレッドは、定期的にその変数を検査するようにする必要があります。スレッドを停止するようにその変数が示している場合は、スレッドは通常どおりに run() メソッドから返るようにします。たとえば、次のような start()、stop()、run() メソッドがアプレットに含まれているとします。

```
public void start() {
  blinker = new Thread(this);
  blinker.start();
}
public void stop() {
  blinker.stop();
// UNSAFE!
 }
public void run() {
  Thread thisThread = Thread.currentThread();
  while (true) {
   try {
    Thread.sleep(interval);
   }
   catch (InterruptedException e){
   }
   repaint();
  }
 }
```

アプレットの stop() および run() メソッドを以下のように変更することによって、Thread.stopを使用しなくて済みます。

```
public void stop() {
  blinker = null;
}
public void run() {
  Thread thisThread = Thread.currentThread();
  while (blinker == thisThread) {
   try {
    Thread.sleep(interval);
   }
   catch (InterruptedException e){
   }
   repaint();
  }
}
```

また、Thread.suspend は、本質的にデッドロックを引き起こす可能性があるため、推奨されません。この Thread.resume も、他のコードに書き換える必要があります。スレッドが中断された時に、そのスレッドが重要なシステム資源を保護するモニターをロックしていると、ターゲットスレッドの実行が再開されるまで、他

のスレッドはこのシステム資源にアクセスすることはできません。ターゲットの実行を再開するスレッドが Thread.resume を呼び出す前にモニターをロックしようとすると、デッドロックが発生します。

このようなデッドロックが発生すると通常はプロセスがフリーズして応答しなくなるので、デッドロックが発生したことがわかります。前述の Thread.stop の場合と同様に、適切なスレッドの状態 (active または suspended) を示す変数をスレッドに持たせる必要があります。スレッドが中断されたときには、スレッドは Object.wait を使用して待機します。スレッドが再開されたときには、Object.notify によってスレッドが再開されたことがターゲットスレッドに通知されます。たとえば、以下のように blinker というスレッドの状態を切り替えるイベントハンドラ mousePressed が、アプレットに含まれているとします。

```
public void mousePressed(MouseEvent e) {
  e.consume();
  if (threadSuspended)
   blinker.resume();
  else
   blinker.suspend();
// DEADLOCK-PRONE!
  threadSuspended = !threadSuspended;
}
```

上記のイベントハンドラを以下のように変更することによって、Thread.suspend および Thread.resume を使用せずに済みます。

```
public synchronized void mousePressed(MouseEvent e) {
  e.consume();
  threadSuspended = !threadSuspended;
  if (!threadSuspended)
   notify();
}
```

ループを実行するために、次のコードを追加します。

```
synchronized(this) {
  while (threadSuspended)
   wait();
}
```

wait() メソッドは InterruptedException をスローするので、try ... catch 節の中に置く必要があります。sleep と同じ節の中に wait() メソッドを置くこともできます。スレッドが再開された時にウィンドウが

ただちに再描画 (repaint)されるように、check は sleep よりも後に置いてください。たとえば次のように run() を記述します。

```
public void run() {
  while (true) {
   try {
    Thread.sleep(interval);
    synchronized(this) {
     while (threadSuspended)
      wait();
    }
   }
   catch (InterruptedException e){
   }
   repaint();
  }
}
```

notify() が mousePressed() メソッドの中にあり、run() メソッドの中の wait() が synchronized ブロック中にあることに注意してください。これは Java 言語で必要とされていることで、wait() と notify() が適切にシリアライズされるようにします。つまり、中断されたスレッドが notify() を認識できずに中断したままになる可能性がある競合状態が発生するのを防ぎます。

第 5 章

アプリケーションのパフォーマンスチューニング

この章では、Solaris 8 環境において Java アプリケーションのパフォーマンスを向上させる方法について説明します。アプリケーションのパフォーマンスとは、そのアプリケーションの資源の使用量と定義することができます。パフォーマンスチューニングとは、資源の使用量を最低限に抑えることです。

注意 - ここで紹介するパフォーマンスチューニングに関するヒントの多くは、Solaris 2.6、Solaris 7、および Solaris 8 で使用する Java に固有です。将来のリリースでは、パフォーマンス特性が変わることが予想され、ここで紹介しているヒントが適切でなくなる可能性があります。

チューニングに関するヒント

以降で説明するように、チューニングはいくつかのレベルで行うことができます。

システムインタフェース

チューニングによって大幅なパフォーマンスの向上が見込める Java システムのインタフェースには、次のものがあります。

- 入出力
- 文字列

- 配列
- ベクトル
- 塗りつぶしと描画
- ハッシュテーブル
- イメージ
- メモリー使用
- スレッド

コンパイラによる最適化

以下のコンパイラによる最適化が可能です。

- Java コンパイラ
- JIT コンパイラ

コードのチューニング

パフォーマンス向上のためには、以下の部分のコードをチューニングします。

- ループ
- 真偽式をテーブルルックアップに変換する
- キャッシュ
- 事前に計算を行う
- 評価の引き延ばし
- クラス — オブジェクトの初期化

入出力

一般的に、Java アプリケーションで最も一般的で大きなパフォーマンス上の問題は、非効率的な入出力です。このため、一般的に入出力の問題は、Java アプリケーションのパフォーマンスチューニングで最初に検討すべき問題になります。入出力の問題を解決することによって、その他のすべての最適化を行なった場合よりも大幅にパフォーマンスが向上することがあります。効率的な入出力手法を用いることによって、10 倍以上の速度の向上が得られることも珍しくありません。

アプリケーションが大量の入出力を行う場合は、入出力のパフォーマンスチューニングを行なってみてください。チューニング結果は、アプリケーションをプロファイルすることによって確認できます。Java アプリケーションのプロファイルするには、Sun の Java WorkShop™ 製品を使用することができます。Java WorkShop は、以下の URL から入手することができます。

- `http://www.sun.com/workshop`
- `http://www.sun.co.jp/workshop/jws`

Java WorkShop のオンラインヘルプで、プロファイラまたはプロファイルについての説明を参照してください。以下の例は、4 つの異なるメソッドを使用して、150,000 行からなるファイルを読み取るベンチマークテストの結果です。

1. `DataInputStream.readLine()` のみ (バッファーなし)
2. `DataInputStream.readLine()` と `BufferedInputStream` (2048 バイトのバッファーあり)
3. `BufferedReader.readline()` (8192 バイトのバッファーあり)
4. `BufferedFileReader(fileName)`

結果は次のとおりです (単位: 秒)。

DataInputStream:	178.740
DataInputStream(BufferedInputStream):	21.559
BufferedReader	11.150
BufferedFileReader	6.991

メソッド 1 と 2 では Unicode の文字が正しく処理されませんが、メソッド 3 と 4 では Unicode 文字が正しく処理されることに注意してください。つまりほとんどの製品では、メソッド 1 と 2 は使用できないことになります。JDK 1.1 では、`DataInputStream.readLine()` も推奨されません。Java WorkShop とその他のプログラムでは、メソッド 1 が使用されています。

Solaris の入出力処理の問題を見つけるためのもう 1 つの方法として、`truss(1)` を使用して、`read(1)` と `write(1)` システムコールを検索するという方法もあります。

文字列

文字列に関して忘れてならない最重要事項は、ループで文字を処理するときには、`String` や `StringBuffer` クラスではなく必ず char 配列を使用することで

す。配列要素へアクセスする方が、charAt() メソッドを使用して文字列内の文字にアクセスするよりもはるかに高速です。また、文字列定数 ("...") はすでに文字列オブジェクトであることを忘れないでください。

```
//DON'T

String s = new String("hello");

//DO

 String s = "hello";
```

- String クラス

  ループ内の可変文字列や文字処理、charAt() メソッドで String クラスを使用しないでください。

- StringBuffer クラス

  StringBuffer クラスは、文字列が可変で、複数のスレッドによって並列にアクセスされ、文字処理が行われない場合にのみ使用してください。ループ内の非可変の文字列や文字処理、charAt() メソッド、setCharAt() メソッドには使用しないでください。デフォルトの文字列サイズは 16 文字です。このクラスは、文字列を連結するときにコンパイラによって自動的に使用されます。最大の文字列サイズがわかっている場合は、初期バッファーサイズとしてそのサイズを設定してください。

- StringTokenizer クラス

  StringTokenizer クラスは、簡単な解析や読み取り走査に役立ちますが、非常に非効率的です。このクラスは、String ではなく文字配列に文字列や区切り文字を格納するか、最上位の区切り文字を格納して、より短時間に検査が行われるようにすることによって最適化できます。区切り文字リストや処理文字列によって異なりますが、このような最適化によって、1.6 倍から 10 倍 (通常は 2.4 倍程度) にパフォーマンスが向上します。

配列

配列は境界が検査されるので、その分パフォーマンスが低下します。ただし、配列へのアクセスは、ベクトルや String、StringBuffer にアクセスするよりははるかに高速です。より高いパフォーマンスを得るには、System.arraycopy() を使用してください。これはネイティブメソッドであり、手動の配列処理よりもかなり高速です。

ベクトル

Vector は便利ですが非効率的です。最高のパフォーマンスを得るには、構造体のサイズが不明で効率性がそれほど重要でない場合にのみ使用してください。Vector を使用する場合は、パフォーマンスが低下するのでループ内で elementAt() を使用しないでください。Vector は、次の特徴を持つ配列に対してのみ使用してください。

- 複数のスレッドによって同時にアクセスされる
- サイズが動的に変化する

ハッシュ

HashTable には、以下のチューニング可能なパラメータがあります。

- initialCapacity (容量、通常は素数): 十分な大きさに設定されていないと、衝突が発生し、ハッシュ処理が停止して、その後線形リスト処理が実行されます。
- loadFactor (負荷率、0.0 から 0.1 の範囲): 容量を超えてテーブルが拡張される割合です。HashTable は hashCode() を呼び出します。これらのクラスには、あらかじめ定義されている hashCode() メソッドがあります。
- Color、Font、Point
- File
- Boolean、Byte、Character、 Double、 Float、 Integer、 Long、 Short、 String
- URL
- BitSet、Date、 GregorianCalendar、 Locale、SimpleTimeZone

  長さによっては、String.hasCode() が必ずしもすべての文字をサンプリングするわけではないことに注意してください。
- 1〜15 文字の長さ: すべての文字
- 16〜23 文字の長さ: 1 文字おき
- 24〜31 文字の長さ: 2 文字おき

イメージ

イメージについては、次のような方法があります。

塗りつぶしと描画

塗りつぶしと描画のパフォーマンスを向上させるには、次の方法を使用してください。

- ダブルバッファリング (たとえば、アニメーションではオフスクリーンにイメージを描画して全体を一度に読み込みます)
- `update()` 関数によるデフォルト値以外の使用 (オーバーライド)

```
public void update(Graphics g) {
    paint(g);
   }
```

- カスタマイズした独自のレイアウトマネージャ の使用。独自の動作が必要な場合は、そのためのコードを作成することによって、最高の GUI パフォーマンスを得ることができます。
- イベントの使用。JDK 1.1 には、1.0 に比べて効率的なイベントモデルが用意されています。
- 損傷を受けた部分だけの再描画 (`ClipRect` を使用)。

非同期の読み込み

非同期の読み込みパフォーマンスを向上させるには、独自の `imageUpdate()` メソッドを使用して `imageUpdate()` をオーバーライドします。`imageUpdate()` は、必要以上に再描画を行うことがあります。

```
//wait for the width information to be loaded
  while (image.getWidth(null) == -1 {
     try {
       Thread.sleep(200);
   }
   catch(InterruptedException e) {
     }
  }
  if (!haveWidth) {
   synchronized (im) {
       if (im.getWidth(this) == -1) {
     try {
              im.wait();
           }
           catch (InterruptedException) {
           }
       }
     }
```

(続く)

```
    //If we got this far, the width is loaded, we will never go thru
    // all that checking again.
    haveWidth = true;
   }
...
public boolean imageUpdate(Image img, int flags, int x, int y, int width, int height) {
   boolean moreUpdatesNeeded = true;
   if ((flags&ImageObserver.WIDTH)!= 0 {
      synchronized (img) {
        img.notifyAll();
     moreUpdatesNeeded = false;
    }
   }
   return
   moreUpdatesNeeded;
  }
```

事前のデコード

イメージのデコードは、読み込みより長い時間がかかります。`PixelGrabber` と `MemoryImageSource` を使用して事前にデコードすることによって、複数のイメージを 1 つのファイルにまとめ、最高の速度が得ることができます。この方法は、ポーリングを行うよりも効率的です。

メモリー使用

アプリケーションのパフォーマンスは、実行中のガーベッジコレクション量を少なくすることによって大幅に向上させることができます。また、次の方法によってもパフォーマンスを向上できます。

- 次のコマンド使用して、初期ヒープサイズをデフォルトの 1 M バイトより大きくする。

  `java -ms number. java -mx number.`

  デフォルトのヒープサイズは最大で 16M バイトです。

- 次のコマンドを使用して、メモリーを多く使いすぎる部分を見つける。

  `java -verbosegc`

- 配列を割り当てるときにサイズを考慮する (たとえば short で十分ならば、int の代わりに short を使用する)。
- ループ内でのオブジェクトの割り当て (`readLine()` など) を避ける。

スレッド

17ページの「Solaris 環境における従来の Java スレッド *」で説明したように、アリケーションのパフォーマンスは、ネイティブメソッドを使用することによって大幅に向上します。グリーンスレッドがタイムスライスされることはないため、実行状態を示すには、ループ内での `Thread.yield()` の呼び出しが必要になり、実行速度が低します。その他、次の方法は使用しないでください。

- 同期の過度の使用。コーディングエラーが原因のデッドロックや、ロック競合が原因の遅延が発生する可能性が大きくなります。また、頻繁な同期によって得られる利点よりも、オーバーヘッドの方が大きくなる可能性もあります。このような場合には同期を最小限にした方がパフォーマンスが向上します。
- ポーリング。外部のイベントを待つときに、副次的なスレッド (メインスレッド以外のスレッド) 中で実行される場合にのみ、ポーリングを行うことができます。ポーリングの代わりに `wait()` および `notify()` を使用してください。

コンパイラによる最適化

Java コンパイラと JIT コンパイラは、次のような最適化を自動的に行います。

Java コンパイラ

- インライン化
- 一定の折り返し処理

JIT コンパイラ

- 配列境界検査を行わない
- ブロック内で共通の部分式を省略
- 空のメソッドを省略
- ローカルのレジスタ割り当ての一部を省略
- フロー分析を行わない
- インライン化を限定する

コード最適化

ループ

パフォーマンスを向上させるには、次のことを守ってください。

- ループ不定正規は、ループの外に移動します。
- テストは、できるかぎり単純にします。
- ループ内では、ローカル変数だけ使用します。ループに入る前にローカル変数にクラスフィールドを割り当ててください。
- 値が定数の条件式はループの外に移動します。
- 同様のループは結合します。
- ループが交換可能な場合は、最も頻繁に実行されるループを入れ子にします。
- 最後の手段として、ループを展開します。

真偽式のテーブルルックアップへの変換

値がある範囲の小さな整数である 1 つの式に基づいて値が選択される場合は、テーブルルックアップに変換してください。条件分岐があると、コンパイラによる最適化の多くが行われなくなります。

キャッシュ

キャッシュによってメモリーの使用量は増えますが、パフォーマンス向上に利用することができます。フェッチや計算に重い負荷がかかる値は、キャッシュを利用してください。

結果の事前計算

コンパイル時にわかっている値を事前に計算しておくと、パフォーマンスが向上します。

評価の引き延ばし

必要になるまで結果の計算を遅らせると、起動時間が短縮されます。

クラスとオブジェクトの初期化

1 回だけ行われる初期化をすべて 1 つのクラスイニシャライザでまとめて行うようにすると、パフォーマンスが向上します。