Java 2 SDK 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 806-2733–10
2000 年 3 月

原典: *Java 2 SDK for Solaris Developer's Guide*

Part No: 806-1367-10

Revision A

Adobe PostScript

目次

はじめに

『*Java 2 SDK* 開発ガイド *(Solaris* 編*)*』では、Solaris™ 8 オペレーティング環境用の Java™ 2 SDK の製品バージョンの新機能と強化機能を紹介し、概要を述べます。

対象読者

このガイドは、Java Developer's Kit (JDK™)を使用するアプリケーション開発者向けに作成されています。Java 2 SDK ソフトウェアは、エンタープライズ環境でサーバ側の Java テクノロジアプリケーションが優れた性能とスケーラビリティを発揮するように最適化されています。Java 2 SDK の拡張された Java Virtual Machine (JVM) には、最適化を行う Just In Time (JIT) コンパイラが含まれています。Java 2 SDK では、マルチプロセッサシステムで大量の Java スレッドを実行する大きなサーバ側アプリケーションの性能が大幅に向上します。サーバ側アプリケーションには、主に次のような特徴があります。

- 使用期間が長い
- 高スレッド型 (マルチプロセッサ対応)
- ネットワーク集約型
- メモリ集約型

JVM の高速化により、クライアント側アプリケーションの処理能力も向上します。

内容の紹介

第 1 章では、新しい機能と強化された機能について説明します。

第 2 章では、Java 2 SDK プログラムと JDK 1.1 プログラムでの互換性の問題を列挙し、説明します。

第 3 章では、Native Method Interface (NMI) から Java Native Interface (JNI) へ移行するときにユーザが考慮すべき事項について説明します。

第 4 章では、Java 2 SDK のリファレンスプラットフォームと製品プラットフォームでサポートされるオプションについて詳しく説明します。

第 5 章では、新しいデバッグ方法について説明します。

付録 A では、Java 2 SDK がメモリをどのように割り当てるかを説明します。

付録 B では、ガベージコレクションの問題を解決する場合、`–verbosegc` をどのように使用するかを説明します。

付録 C では、新しい `Poller` クラスの使い方を説明します。

付録 D では、システムのデフォルト JDK を JDK 1.1 から Java 2 SDK に変更する方法について説明します。

関連マニュアル

このリリースに関する情報は次のドキュメントにも記載されています。

- 『リリースノート*: Java 2 SDK 1.2.1_04 Solaris* 版』
- 『*Java 2 SDK v.1.2.1_04 Troubleshooting*』

これらのドキュメントは次のサイトからダウンロードできます。

`http://www.sun.com/solaris/java`

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。

表 P–1 表記上の規則 続く

字体または記号	意味	例
『 』	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

新しい機能と強化された機能

この章では、Java 2 SDK Solaris 版の新しい機能と強化された機能について説明します。このリリースを使ってアプリケーションを開発すると、以下のような Java 2 プラットフォームの高度な機能を使用することができます。

- 機能強化されたセキュリティモデル
- 向上した相互運用性 (データベースアプリケーションや CORBA ベースのアプリケーションにおいて)
- コレクション、国際化、GUI 開発用の API が組み込まれた、完全な開発プラットフォーム

さらに、Java 2 SDK Solaris 版では、Java Virtual Machine (VM) で次の点が強化されたためケーラビリティと性能が大幅に向上しました。

- 最適化 Just-In-Time (JIT) コンパイラ
- 高速スレッド同期化
- 最新のメモリシステム

Java 2 SDK Solaris 版の製品リリースでは、アプリケーションの導入を支援する次の機能が提供されます。

- Java アプリケーションを実行するための安全な環境
- 大規模なサーバ側アプリケーション用の最先端のスケーラビリティと性能
- 既存の Solaris オペレーティング環境サポート契約に組み込まれたミッションクリティカルなサポート

高性能メモリシステム

Java 2 SDK Solaris 版には高度に最適化されたメモリシステムが組み込まれているため、メモリの割り当てやガベージコレクションが以前よりも効率的になります。このメモリシステムでは直接ポインタが使用されます。このメモリシステムには次の特徴があります。

- 革新的である
- メモリの使用効率が高まる
- 世代管理方式である

このような機能により、プログラムの性能が向上し、ガベージコレクションのために対話プログラムが中断されることが少なくなります。

高性能メモリシステムの一部で次の機能が拡張されたため、システムの性能が大幅に向上します。

正確なガベージコレクション

Java 2 SDK Solaris 版では正確なガベージコレクションが行われます。このガベージコレクションでは、従来のシステムよりも広範囲なガベージコレクション技法が使われるため性能が向上します。

Java 2 SDK Solaris 版では、多くのアプリケーションに対して大幅に性能が向上する効率的な世代管理方式のガベージコレクタが使用されています。世代管理方式のガベージコレクションでは、割り当て済みオブジェクトのサブセットに対して、再利用が可能かどうかが検査されます。このサブセットの選択はオブジェクトの古さに基づいて行われ、そこでは新しい「世代」ほど頻繁に検査が行われます。このサブセット化を行うのは、Java プログラムでは、新たに作成されたオブジェクトほど、古いオブジェクトと比べて回収の対象である可能性が高いためです。通常、新しい世代の回収はすべてを回収する場合よりかなり短い時間で済むため、この手法には、ガベージコレクションのための一時停止時間が短縮されるという利点もあります。

この性能の向上を実現するには、ユーザが作成したネイティブコードを含めて、システム全体が正確なガベージコレクションをサポートしている必要があります。JNI (Java Native Interface) の実装では、正確なガベージコレクションに対応することができます。Java 2 SDK Solaris 版では、そうした実装が可能なため、JNI を使用して

作成したネイティブコードは Java 2 SDK Solaris 版 VM に対してそのまま使用できます。

直接ポインタ

Java 2 SDK Solaris 版の正確なガベージコレクションでは、オブジェクトに対しハンドルの代わりに直接ポインタが使用されます。直接ポインタではオブジェクトをアクセスするための間接レベルが 1 つ除かれるため、メモリ使用量が減少し、割り当て速度が増し、システム性能が向上します。

ヒープにおけるダブルワード (long と double) の 8 バイト境界への位置合わせ

ヒープではダブルワード値は 8 バイト境界に位置合わせされます。そのため、SPARC™ システムで、変動しやすいダブルワード値を正確に保持しながら、ネイティブコードおよび JIT でコンパイルされた Java コードの性能が両方とも向上します。ただし、アプリケーションで小さいオブジェクトをたくさん割り当てて保持する場合は、ヒープサイズをいくらか増やす必要があるかもしれません。これらのオブジェクトは 8 バイトの倍数の境界に割り当てられるため、メモリ使用量が増加するからです。

マルチスレッド機能

同期処理におけるオーバーヘッドの軽減

次の各機能によりマルチスレッドアプリケーションの同期オーバーヘッドが減少するため、性能が向上します。

高速同期モニターロック/高速スレッド同期

Java 2 SDK Solaris 版では新しい高速ロックアルゴリズムが内部的に使用されるため、メソッドの同期化が効率的になります。この VM では、Java プラットフォームの同期化プリミティブの実装が大幅に拡張されました。この実装では、並行プログラムの実行が効率的になり、同期化プリミティブが単一スレッドのアプリケーション性能に与える影響が減少します。

ロック競合の抑制

Java 2 SDK Solaris 版は、スレッドに対してローカルなデータ構造を使用し、よりきめの細かいロックを行うことによって、VM においてできる限りロック競合が発生しないようにします。

VM のコアロックアーキテクチャは、よりきめの細かい VM ロックを行うことができます。このロックによって、競合するスレッド実行パスの数が最低限に押さえられます。競合するパスが少なくなるということは、あるスレッドの実行が他のスレッドを妨害する可能性が低くなることを意味します。これにより、マルチプロセッサ (MP) のスケーラビリティが改善されます。また、きめの細かいロックにより VM が大量の同時処理を行うことが可能になります。

スレッド固有データのアクセス

Java 2 SDK Solaris 版では、新しい内部キャッシュ機構を使用することにより、スレッド固有データにアクセスする時間を短縮しました。これは、VM と各プラットフォームに固有の JIT から実行環境データにアクセスする際に使われる一般的なパフォーマンス拡張機能です。

ツール

ヒープ検査ツール

この診断ツールは、`SIGQUIT` ハンドラメニューから終了されたプログラムに対して対話的に使用できます。このツールでは、プログラムのメモリリークを見つけることができます。メモリリークは、プログラムが不注意でオブジェクトを保持しているためにガベージコレクターがメモリを再利用できないときに起ります。ヒープ検査をすると、ヒープにあるオブジェクトがクラスごとに合計メモリ使用量の多い順に示されます。オブジェクトへの参照チェーンを調べると、オブジェクトがなぜ終了しないのかがわかります。

デバッグユーティリティ

Java 2 SDK Solaris 版では新しいデバッグプロセスが提供されます。このプロセスでは、フォアグラウンドで動作する Java プロセスに `SIGQUIT` シグナルを送信しま

す。このシグナルを使えば、そのプロセスに対し、スレッドやモニターの状態ダンプ、デッドロック検出などのデバッグタスクを行うことができます。詳細は、第 5 章を参照してください。

最適化 JIT コンパイラ

最適化 JIT コンパイラは、Java アプリケーションの起動時に最大の実行速度が与えられるように Java Virtual Machine に指示します。JIT コンパイラの新しい最適化機能には、仮想および非仮想メソッドのインライン化、拡張基本ブロック内の CSE 処理、ループ解析 (配列境界の検査が不要になる)、高速の型検査など、SPARC プラットフォームと Intel (IA) プラットフォームの両方で最適化が実現されます。

インライン化

Java 2 SDK Solaris 版のインライン化を使うと、インライン化を手動で行う必要がなくなります (したがって、プログラムのモジュール性を損うこともありません)。自動インライン化は、フロー制御を含まない比較的小さな非同期メソッドに限られます。

混在モード実行

Java 2 SDK Solaris 版には、性能の大幅な向上を可能にする混在モード実行機能が新たに追加されています。混在モード実行では、VM は性能が重要となるメソッドだけをコンパイルして、残りのメソッドについてはインタプリタとして動作します。この機能はコンパイル時のオーバーヘッドを軽減し、プログラムの起動を高速化します。また、本当に重要な部分のコンパイルと最適化に多くの時間を割いて、性能の向上を図れるようにします。

混在モード実行は、デフォルトのモードです。混在モードでは、VM はメソッドを次の 2 つに分類します。

- ループを含む可能性があるメソッド
- ループを含まないメソッド

このそれぞれの種類に対して、JIT コンパイラがコンパイルのタイミングを決定します。ループを含む可能性があるメソッドは最初の実行時にコンパイルされます。ループを含まないメソッドの場合は、15 回目の呼び出し時にコンパイルが行われます。

スケーラビリティの向上

Java 2 SDK Solaris 版では、並行プリミティブや並行メソッドの処理が向上し、マルチスレッド (MT) プログラムの性能が向上し、多数のスレッドを使用するプログラムのガベージコレクションによる停止時間が大幅に減少します。

テキスト描画性能の向上

グラフィックの最適化により、ダイレクトグラフィックアクセス (DGA) サポートのない Solaris プラットフォームで、Java 2 SDK のテキスト描画性能が向上しました。ここでいう Solaris プラットフォームとは、Ultra™ 5、Ultra 10、Intel で実行する Solaris (IA)、およびすべてのリモートディスプレイシステムです。

`poller` クラスのデモ

新しい `poller` クラスを使うと、Java アプリケーションから C の `poll(2)` ルーチンの関数に効率よくアクセスできます。Java 2 SDK Solaris 版のデモコードは、サンプル用サーバと一緒に以下のディレクトリにあります。

```
${JAVA_HOME}/demo/jni/Poller
```

デモでは、複数の入出力オブジェクトの入出力状態を 1 つの呼び出しで判定する方法が示されています。

`poller` クラスをサポートする JNI C コードは、Solaris 7 の `poll(2)` カーネルキャッシュ機能の利点を最大限に利用するように最適化されています。このコードは、高速カーネルポーリング機構 `/dev/poll` を利用するようにコンパイルできます。この機構は Solaris 7 system Software update 2 (5/99) 版とそれ以降のバージョンで使用できます。

`poller` クラスの詳細は、付録 C を参照してください。

第 2 章

Java 2 SDK Solaris 版での下位互換性

この章の内容は次のとおりです。

Java 言語仕様は「JLS」で表します。

バイナリレベルの互換性

Java 2 SDK Solaris 版は、以降に示す非互換の場合を除き、JDK ソフトウェアバージョン 1.0 および 1.1 とバイナリレベルで上位互換性があります。したがって、バージョン 1.0 か 1.1 のコンパイラで構築したクラスファイルは、以降に示す非互換の場合を除き、Java 2 SDK Solaris 版で正しく動作します。

`javac` コンパイラの `–target 1.2` コマンド行オプションを使用しなければ、一般にはバイナリレベルでの下位互換性がサポートされます。ただし、保証はされません。したがって、Java 2 SDK Solaris 版のコンパイラで構築したクラスファイルは、

Java プラットフォームのバージョン 1.0 か 1.1 で定義されている API だけを使用している限り、一般には Java Virtual Machine の 1.0 や 1.1 バージョンで正しく動作します。ただし、この下位互換性は完全にテストされているわけではないので、保証はされません。クラスファイルが新しい Java 2 SDK Solaris 版の API を使用している場合、これらのファイルは以前のプラットフォームでは当然ながら動作しません。

互換性に関する一般的な方針は次のとおりです。

- ファミリ (JDK 1.1 ベースのリリース) 内の保守リリース (JDK 1.1.1、1.1.2 など) では、相互の上位と下位への互換性がバイナリレベルで維持されます。
- ファミリ (JDK 1. ベースのリリース) 内の機能リリース (JDK 1.1、1.2 など) では、上位互換性はバイナリレベルで維持されますが、バイナリレベルの下位互換性は必ずしも維持されません。早期の Java バイトコードオブファスケータの中には、Java Virtual Machine 仕様 (『*Java Virtual Machine Specification, Second Edition*』) で指定されたクラスファイル形式に合わないクラスファイルを生成するものがありました。このようなクラスファイルは、以前のバージョンの VM で動作していたとしても、Java 2 SDK Solaris 版の VM では動作しません。これを解決するには、新しいオブファスケータを使って正しい形式のクラスファイルを再生成する必要があります。

ソースレベルの互換性

Java 2 SDK Solaris 版は、以降に示す非互換の場合を除き、JDK ソフトウェアバージョン 1.0 および 1.1 とソースレベルで上位互換性があります。したがって、バージョン 1.0 や 1.1 に定義された言語機能や API を使用するソースファイルは、以降に示す非互換の場合を除き、Java 2 SDK Solaris 版で正しくコンパイルして実行することができます。

ソースレベルでの下位互換性はサポートされません。ソースファイルに新しい言語機能や Java 2 SDK Solaris 版の API が含まれている場合は、このファイルを以前のバージョンの Java プラットフォームで使用することはできません。

互換性に関する一般的な方針は次のとおりです。

- 保守リリースでは新しい言語機能や API は導入されないため、両方向へのソースレベルの互換性が維持されます。
- 機能リリースと主要リリースではソースレベルの上位互換性は維持されますが、下位へのソースレベルの互換性は維持されません。

推奨されない API とは、下位互換の目的でのみサポートされるメソッドやクラスです。推奨されない API を検出すると、コンパイラは警告メッセージを生成します (–nowarn コマンド行オプションが使用されなかった場合)。推奨されないメソッドやクラスを API から削除する時期や実際に削除するかどうかは、まだ決定されていませんが、推奨されないメソッドやクラスを使用しないでください。

Java 2 SDK Solaris 版の互換性の問題

次の各項目は、Java プラットフォームのバージョン 1.0 や 1.1 で動作するプログラムが Java 2 SDK Solaris 版で動作しない場合を網羅したものです。このほとんどは一般的な状況で起るものではないので、既存のほとんどのプログラムにはおそらく影響ありません。

言語の非互換性の問題

JDK 1.0 および 1.1 のコンパイラは、数種類の不正なコードを警告メッセージやエラーメッセージを表示せずにコンパイルしてしまいます。コードを JLS の仕様に準拠させるという点において、Java 2 SDK Solaris 版のコンパイラは厳密さが増しています。次は、JDK 1.0 または 1.1 のコンパイラではコンパイルされるが、Java 2 SDK Solaris 版のコンパイラではコンパイルされない不正なコードの種類を示します。

1. 以前のコンパイラでは、`long` 型に対する `int` 型初期設定が許可されることがありました。Java 2 SDK Solaris 版ではつねにエラーになります。たとえば、次のコードの *i* および *j* の初期設定は無効です。

```
public class foo {
 int i = 3000000000;
 int j = 6000000000;
}
```

   この場合、以前のコンパイラは *i* の不正な初期化を報告するだけであり、*j* の初期化はメッセージを表示せずに単にオーバーフローしていました (バグ ID 4035346)。

2. 以前のコンパイラでは、8 ビットに収まる文字リテラルに対する `char` から `byte` および `short` への暗黙の代入変換が許可されていました。Java 2 SDK

Solaris 版のコンパイラは、そうした暗黙の代入変換を許可しません。たとえば、次のコードはコンパイラを通りません。

```
byte b = 'b';
```

こうした変換には、次のような明示的な型変換を使用してください (バグ ID 4030496)。

```
byte b = (byte)'b';
```

3. Java 2 SDK Solaris 版のコンパイラは *0xL* を許可しません (正当な 16 進リテラルではない)。以前のコンパイラは、これをゼロとして解析します (バグ ID 4049982)。
4. Java 2 SDK Solaris 版のコンパイラは ''' (つまり '*u0027*') を許可しません (正当な `char` リテラルではない)。代わりに '\" を使用してください (バグ ID 1265387)。
5. Java 2 SDK Solaris 版のコンパイラは "*u000D*" を許可しません (`string` および `char` リテラルでは不正)。CR および LF 文字 (*u000A* と *u000D*) は、コメントで使用されている場合でも行を終了します (バグ ID 4086919)。このため、次のコードは不正です。

```
//This comment about \u000D is not legal; it is really two lines.
```

代わりに *r* を使用してください (バグ ID 4063417)。

6. Java 2 SDK Solaris 版のコンパイラは、`void[]` 型を許可しません。不正です (バグ ID 4034979)。
7. `abstract` メソッド修飾子を `private`、`final`、`native`、または `synchronized` 修飾子と組み合わせないでください。不正です (バグ ID 1266571)。
8. 以前のコンパイラでは、状況によっては、`final` 変数に対する二重代入が許可されることがあります。たとえば、次の 2 つの例には `final` 変数に対する二重代入が含まれていますが、1.1 のコンパイラでは許可されます。Java 2 SDK Solaris 版のコンパイラは、こうした代入を許可しません (バグ ID 4066275 および 4056774)。

```
public class Example1 {
  public static void main(String[] argv) {
   int k=0;
   for (final int a;;) {
    k++;
    a=k;
```

(続く)

```
    System.out.println("a="+a);
    if (k>3)
     return;
   }
  }
}

public class Example2 {
  final int k;
  Example2() {
   k = 1;
  }
  Example2(Object whatever) {
   this();
   k = 2;
  }
}
static public void main(String[] args) {
  Example2 t = new Example2(null);
  System.out.println("k is "+ t.k);
}
```

9. static フィールド式であることが明らかな Classname.fieldname の形式の式で、static ではないフィールドを参照することはできません。Java 2 SDK Solaris 版より前のバージョンでは、javac は this.fieldname と書かれているかのようにそのまま許可していました (バグ ID 4087127)。
10. JLS のセクション 5.5 は、2 つのインタフェースが存在していて、それらのインタフェースにシグニチャーは同じでも、戻り値の型が異なるメソッドが含まれている場合、そのインタフェース間のキャストはコンパイル時エラーになると規定しています。Java 2 SDK Solaris 版より前のコンパイラでは、このコンパイル時エラーが生成されませんでした (バグ ID 4028359)。たとえば、次のコードは現在ではコンパイル時エラーになります。

```
interface Noisy {
  int method();
}
interface Quiet {
  void method();
}
public class InterfaceCast {
  public static void main(String[] args) {
   Noisy one = null;
   Quiet two = (Quiet) one;
```

(続く)

```
  }
}
```

11. Java 2 SDK Solaris 版は、条件文の 3 つ目の部分式に代入式を受け付けません。たとえば、Java 2 SDK Solaris 版のコンパイラは次の文でエラーをスローします。

```
myVal = condition ? x = 7 : x = 3;
```

既存のコードでこのエラーが発生する場合は、以前の JDK のときのように、原因になっている代入式を括弧で囲って、コンパイルしてください。

```
myVal = condition ? x = 7 : (x = 3);
```

12. 前のバージョンの `javac` では、フィールドにデフォルト値が初期設定されていた場合に、コンパイラが誤って初期設定を省略していました (バグ ID 1227855)。このため、次のようなプログラムは異なる意味をもつ場合があります。

```
abstract class Parent {
  Parent() {
   setI(100);
  }
}
 abstract public void setI(int value);
}
public class InitTest extends Parent {
  public int i = 0;
  public void setI(int value) {
   i = value;
  }
}
 public static void main(String[] args) {
  InitTest test = new InitTest();
  System.out.println(test.i);
}
```

前のバージョンの `javac` でコンパイルした場合、このプログラムは不正な出力 (*100*) を生成します。Java 2 SDK Solaris 版の `javac` は、正しい出力 (*0*) を生成します。

1 つのクラスの例を次のように作成することもできます。

```
public class InitTest2 {
  public int j = method();
  public int i = 0;
  public int method() {
   i = 100;
   return 200;
  }
}
public static void main(String[] args) {
  InitTest2 test = new InitTest2();
  System.out.println(test.i);
}
```

以前、この出力は *100* でした。現在は *0* になります。同じ現象が、参照型と `null` を使用したコードで発生することがあります。

13. JLS 6.6.1 には、修飾名におけるアクセスに関して次の規定があります。

> “参照型 (`class`、`interface`、または `array`) のメンバ (フィールドまたはメソッド) または `class` 型のコンストラクタは、その型がアクセス可能で、そのメンバまたはコンストラクタのアクセス許可が宣言されている場合にのみアクセスすることができます。”

Java 2 SDK Solaris 版より前のコンパイラは、この規則を正しく適用していませんでした。メンバまたはコンストラクタが属する型がアクセスできるかどうかに関係なく、アクセス許可が宣言されている場合はアクセスを許可していました。次に不正なプログラムの例を示します。

```
import pack1.P1;
public class CMain {
  public static void main(String[] args) {
   P1 p1 = new P1();
// 以下のフィールド i へのアクセスは
// 不正。p2 の型、すなわち class 型 pack1.P2 に
// はアクセスできない
   p1.p2.i = 3;
   System.out.println(p1.p2.i);
  }
}
package pack1;
public class P1 {
  public P2 p2 = new P2();
}
// パッケージ pack1 からのみしか P2 をアクセス可能
// にするアクセス修飾子がないことに注目
public class P2 {
  public int i = 0;
```

(続く)

```
}
```

Java 2 SDK Solaris 版における内部クラスの導入によって、クラスまたはインタフェースのメンバはフィールドやメソッドだけでなく、別のクラスまたはインタフェースにすることができます。Java 2 SDK Solaris 版では、内部クラスに対しても上記のアクセス規則が適用されます。

14. Java 2 SDK Solaris 版より前のコンパイラは、あるクラスが `abstract` クラスであることを検出できないことがありました。これは、サブクラスが別のパッケージのスーパークラスに定義されている `package-private` の `abstract` メソッドと同じ名前のメソッドを宣言した場合に発生します。メソッドが同じ名前であっても、名前が無効にされることはありません。たとえば、次のファイルをコンパイルすると、

```
package one;
public abstract class Parent {
  abstract void method();
}

package two;
public class Child extends one.Parent {
 void method() {}
}
```

次のエラーメッセージが返されます。

```
two/Child.java:3: class two.Child is not able to provide an
implementation for the method void method() declared in class
one. Parent because it is private to another package. class
two. Child must be declared abstract.
 public class Child extends one.Parent {
   ^
```

15. Java 2 SDK Solaris 版のコンパイラは、入れ子内のラベルの重複を正しく検出します。JLS の仕様では、入れ子内のラベルの重複を禁止しています。このため、次の文は不正です。

```
sameName:
while (condition) {
 sameName:
 while (condition2) {
  break sameName;
 }
}
```

16. Java 2 SDK Solaris 版のコンパイラは、ラベル付きの宣言を正しく検出します。JLS の仕様では、ラベル付きの宣言が禁止されています。これはバグ ID 4039843 の修正です。

17. Java 2 SDK Solaris 版のコンパイラは新しいキーワードの `strictfp` を認識します。このため、プログラムで `strictfp` を識別子として使用することはできません。Java 2 SDK Solaris 版のコンパイラは、この新しいキーワードを使用して、メソッドデータ構造体の修飾子ビットを設定します。Java 2 SDK Solaris 版より前の Java プラットフォームの指定では、このビットをゼロにしている必要がありました。この `strictfp` キーワードを使用しているコードは厳しい浮動小数点モード (Java プラットフォームに定義されているデフォルトのモード) で動作します。この新しいキーワードを使用していないコードは、デフォルト浮動小数点モードで動作し、いくつかのプロセッサを巧みに利用して、性能の向上を図れるようにします。

    `strictfp` キーワードのマークが付いていない一部の数値コードの動作が、Java 2 SDK Solaris 版と前のバージョンとでは異なることがあります。そうしたコードの動作は、Java プラットフォームの実装状態によっても異なることがあります。状況によっては、オーバーフローまたはアンダーフローが発生し、多少異なる結果が生成されることがあります。大部分の数値コードは、こうした違いの影響を受けるとは考えられません。しかし、浮動小数点演算が重要な意味を持つコードは影響を受けることがあります。

18. JDK 1.1 では式構文が拡張され、次の例に示すように `this` キーワードを使用して、現在のインスタンスに対する参照をクラス名で修飾できるようになりました。

```
PrimaryNoNewArray: ...
                ClassName . this
```

    こうした式の値は、必ず存在する必要がある囲むクラス (`ClassName`) の現在のインスタンスに対する参照になります。

Java 2 SDK Solaris 版より前の javac コンパイラでは、こうした式の取り扱いに誤りがありました。ClassName で指定された型のサブタイプであるもっとも内側で囲むクラスの現在のインスタンスに対する参照を生成していました。現在は、そうした式も正しく実装されます。

19. JDK 1.1 では、式構文が拡張され、次の例に示すように、this キーワードを使用して、現在のインスタンスに対する参照をクラス名で修飾できるようになりました。

```
PrimaryNoNewArray: ...
                ClassName . this
```

この構文では、次のような形式でメンバにアクセスすることができます。

```
ClassName.this.fieldname
ClassName.this.methodname( ... )
```

この拡張が必要なのは、内部クラスが存在していて、このコードの任意の地点で現在のインスタンスが複数存在する可能性があるためです。次の例に示すように、内部クラス仕様では、super キーワードに対して同様の拡張が含まれていませんでした。

```
FieldAccess: ...
                ClassName.super.Identifier
MethodInvocation:...
                ClassName.super.Identifier(ArgumentList_opt)
```

第 2 版の JLS に含まれることを予想して、Java 2 SDK Solaris 版には、これらの構文が実装されています。

これらのどの場合も、現在のインスタンスは、存在する必要がある囲むクラス (ClassName) の現在のインスタンスです。

次は、1 つの例として、修飾された super の指定が実際に必要になる場合を示しています。

```
class C {
  void f() { ... }
}
class D extends C {
  void f() {
            // f() をオーバーライドして新しいスレッドで実行
   new Thread(new Runnable() {
    public void run() {
     D.super.f();
    }
   }).start();
  }
}
```

実装に関する注: アクセスメソッドを使用する必要がある場合は、スーパークラスではなく ClassName で指定しているクラスに置く必要があります。スーパークラスを同じコンパイルユニットで定義する必要はありません。事前にコンパイルしておいてもかまいません。

20. JDK 1.1 では、式構文が拡張され、次の例に示すように、super キーワードを使用したコンストラクタの呼び出しを、外部インスタンスに対する参照で修飾できるようになりました。

```
ExplicitConstructorInvocation: ...
 Primary.super(ArgumentList_opt);
```

this を使用したコンストラクタの呼び出しが不注意にも省略されていました。

```
ExplicitConstructorInvocation: ...
 Primary.this(ArgumentList_opt);
```

第 2 版の JLS に含まれることを予想して、Java 2 SDK Solaris 版には、これらの構文が実装されています。

21. 内部クラスの仕様では、内部クラスでメンバインタフェースを宣言することはできません。Java 2 SDK Solaris 版では、この規則が適用されますが、JDK 1.1 リリースでは、そうした宣言は黙認されます。 たとえば、次のコードは、JDK 1.1 リリースの javac では誤って容認され、Java 2 SDK Solaris 版の javac では拒否されます。

```
class InnerInterface {
  class Inner {
   interface A { }
  }
}
```

静的なメンバクラスは内部クラスになりません。静的なメンバクラスは最上位のクラスです (内部クラスの仕様で定義に従う)。JDK 1.1 リリースおよび Java 2 SDK Solaris 版の `javac` はともに、正しいコードとして次のコードを受け入れます。

```
class NestedInterface {
  static class Inner {
   interface A { }
  }
}
```

ローカルのクラスが最上位のクラスになることはありません。このため、次の例は不正であり、1.1 リリースおよび Java 2 SDK Solaris 版のどちらの `javac` でも、構文エラーが返されます。

```
class LocalNestedInterface {
 void foo()
    static class Inner {
     interface A { }
   }
 }
}
```

22. Java 2 SDK Solaris 版のコンパイラは、パッケージが同じ名前の型とサブパッケージを含むことはできないという制限事項を強制的に適用します。この変更のため、`classpath` にパッケージの完全修飾名と同じ完全修飾名を持つクラスをコンパイルすることは誤りです。また、既存のクラスと同じ名前を持つことになるパッケージ (または正しい接頭辞) を持つクラスをコンパイルすることも誤りです。これは、バグ ID 4101529 に対する修正です。次に、現在はコンパイルできないクラスの例をいくつか示します。

```
\\例 1
package java.lang.String;
class Illegal {
}

\\例 2
package java;
class util {
}
```

実行時の非互換性の問題

1. JDK 1.0 および 1.1 の実行システムは、いくぶん積極的にオブジェクトをファイナライズします。すなわち、`finalize` メソッドを呼び出します。つまり、適格ではあるが、`unfinalize` されていない状態のオブジェクトは、ほぼすべてのガベージコレクションサイクルの最後にファイナライズされることがあります。これらの実行システム用に作成されたコードは、正しい動作のために、この即時 GC 駆動ファイナライズを無意識に前提としている可能性があります。これは複雑なバグやデッドロックを発生させる可能性があります。

   Java 2 SDK Solaris 版では、ガベージコレクタによって直接にファイナライズが行われることはありません。その代わりに優先度の高いスレッドによってオブジェクトがファイナライズされます。このため、使用中のプログラムでは、オブジェクトが適格になってからファイナライズされるまでの時間が、以前のバージョンの実行システムより長くなることがあります。

   Java 2 SDK Solaris 版の実行システムでは、JLS のファイナライズの定義が正しく実現されているため、厳密に言うと、この違いが互換性の問題になることはありません。ただし、即時ファイナライズに依存している場合、この変更によってプログラムが誤作動することがあります。多くのプログラムは、finalize メソッドではなく、参照オブジェクトを使用することによって修正することができます (参照オブジェクトは、`java.lang.ref.Reference` クラスとそのサブクラスで実装する)。 また、あまり望ましい回避策ではありませんが、定期的に `System.runFinalization` メソッドを呼び出す方法もあります。

2. Java 2 SDK Solaris 版より前のバージョンの Virtual Machine は、JLS に従えば拒否されるべき一部のクラスファイルを受け付けます。一般的に、そうしたクラスファイルには、次の問題が少なくとも 1 つ存在します。

   a. ファイルの末尾に余分なバイトがある

   b. ファイルに、英字以外の文字で始まるメソッドまたはフィールド名が含まれている

c. クラスが別のクラスの private メンバへのアクセスを試みる

d. ファイルに、不正な定数プールインデックス、不正な UTF-8 文字列などの書式エラーが含まれている

Java 2 SDK Solaris 版 VM は、これらのすべてに関してさらに厳密になるように、仕様により忠実に実装されています。

Java 2 SDK Solaris 版の RC2 リリースになるまで、デフォルトではこの厳密な検査になっていました。しかし、最終的な製品テストでは、こうした厳密な検査のために、既存の多数の Java アプリケーションが実行できないことが明らかになっています。具体的には、理解しにくいコードでは a および b の問題が頻繁に発生し、以前のコンパイラで生成された内部クラスのコードは c の問題を抱えています。Java 2 SDK Solaris 版の最終版では、開発者や一般ユーザができる限り少ない労力で Java 2 SDK Solaris 版にアップグレードできるよう、これらの検査の一部が緩和されています。

Java 2 SDK Solaris 版の `–Xfuture` オプションは、クラスファイルのできる限り厳密な形式検査、アクセス検査、検査ポリシーを有効にします。開発者は、できる限り速やかに新たに始まる開発プロジェクトでこのオプションを使用してください。これにより、新しい Java アプレットおよびアプリケーションが、再びデフォルトになる厳密な動作に移行できるようになります。

Java Plug-in は、あたかも `–Xfuture` が設定されているかのようにつねにクラスファイルの厳密な検査を行います。appletviewer は `–Xfuture` フラグを無視し、厳密さでは劣るデフォルトの一群の検査を行います。

3. Java 2 SDK Solaris 版では、`abstract` メソッドまたは `interface` メソッドが実装されていないと、実行時にそのメソッドが呼び出されたときに `AbstractMethodError` が発生します。以前のバージョンでは、このエラーはリンク時に発生していました。

4. Java 2 SDK Solaris 版より前では `CLASSPATH` にコードを書き込むと、特権レベルが高くなります。このために、たとえば、Java 2 SDK Solaris 版より前の VM では、信頼されているクラスファイルの検査が行われないことがあります。検査は、インストールされているセキュリティマネージャに依存します。たとえば、`CLASSPATH` に指定されたクラスに次のメッセージがあり、アプレットによって呼び出された場合、そのメソッドは `user.name` プロパティを読み取ることができました。

```
String getUser() {
 return System.getProperty("user.name");
}
```

Java 2 SDK Solaris 版では、この種のコードは、次の例に示すように、Privileged を呼び出す必要があります。

```
String getUser(){
  return(String)
  java.security.AccessController.doPrivileged
                 (new PrivilegedAction(){
   public Object run() {
    return System.getProperty(''username'');
   }
  })
}
```

- Java 2 SDK Solaris 版のセキュリティモデルでは、資源アクセス方法が変更されています。セキュリティマネージャが有効な場合、資源は URL ポリシーファイルによって付与された適切なセキュリティアクセス権を持つ URL に存在する必要があります。デフォルトのポリシーファイル (java.policy) は、lib/ext ディレクトリ (拡張機能が格納されているディレクトリ) に存在する資源にすべてのセキュリティアクセス権を付与します。デフォルトのポリシーファイルは、/lib/security/java.policy にあります。

  さらに、システム資源にアクセスするための呼び出し (たとえば、ClassLoader.getResource による) は、AccessController.doPrivileged の呼び出しで囲む必要があります。doPrivileged 文を使用せずにシステム資源にアクセスしようとすると失敗します。これは、ポリシーファイルによってシステム資源にセキュリティアクセス権が付与されていた場合でも同様です。

5. JDK 1.1.6 より前では、デフォルトの ISO 8859-1 文字エンコーディングの名前は 8859_1 でした。JDK 1.1.6 では、この名前は ISO8859_1 に変更されています。API において引数としてメソッドに渡した場合、古い名前の 8859_1 は機能しますが、font.properties ファイルで使用された場合は、機能しません。
6. java.util.zip パッケージの次の各クラスには、バッファサイズを指定する *int* パラメータを受け付けるコンストラクタが追加されています。
   - DeflaterOutputStream
   - InflaterInputStream
   - GZIPInputStream
   - GZIPOutputStream

バッファサイズの入力パラメータ値が 0 以下の場合、Java 2 SDK Solaris 版では、`IllegalArgumentException` がスローされます。JDK 1.1 プラットフォームでは、サイズパラメータが 0 以下であっても、これらのコンストラクタは `IllegalArgumentException` をスローしません。

7. Java 2 SDK Solaris 版の VM は、クラスファイル形式が不正なクラスファイルのロードが試みられると、`java.lang.ClassFormatError` をスローします。また、サポートされているメジャーまたはマイナーバージョンではないクラスファイルのロードが試みられると `java.lang.UnsupportedClassVersionError` をスローします。以前のバージョンの Virtual Machine は、クラスファイルにこうした問題があると、`java.lang.NoClassDefFoundError` をスローしていました。

8. Java 2 SDK Solaris 版では、アプリケーションクラスは実際の `ClassLoader` インスタンスによってロードされます。したがって、アプリケーションクラスは、インストールされた拡張機能を使用することができ、アプリケーションのクラスパスとブートストラップのクラスパスを分離できます。アプリケーションのクラスパスはユーザが指定しますが、ブートストラップのクラスパスは一定であり、ユーザが修正すべきものではありません。必要な場合は、`–Xbootclasspath` オプションを使ってブートストラップのクラスパスを無効にできます。

   しかし、このことは、Java 2 SDK Solaris 版では、アプリケーションクラスがデフォルトですべてのアクセス権を持つことはなくなることを意味します。この代わりに、アプリケーションクラスには、システムの構成されたセキュリティポリシーに基づいてアクセス権が与えられます。そのため、1.0/1.1 の元のセキュリティモデルに基づいて作成された独自のセキュリティコードを含むアプリケーションは例外をスローし、Java 2 SDK Solaris 版で起動しないことがあります。この問題を回避するには、このアプリケーションを `oldjava` アプリケーション起動ツールを使って実行します。この説明は、java アプリケーション起動ツールのリファレンスページに記載されています。`oldjava` ユーティリティの詳しい使用方法については、第 4 章を参照してください。

   新しい拡張機能機構と、クラスのロードに対するその効果について、詳しくは `http://java.sun.com/products/jdk/1.2/ja/docs/ja/guide/extensions/spec.html` の「拡張機能機構の仕様」を参照してください。このドキュメントでは、新しいクラスローダ委託モデルとクラスローダの API の変更に関係する情報も提供します。

Java 2 SDK Solaris 版のセキュリティモデルについては、http://java.sun.com/products/jdk/1.2/ja/docs/ja/guide/security/index.html の JDK セキュリティドキュメントを参照してください。

9. Java 2 SDK Solaris 版では、パッケージ java.io の一部のクラスに、入力パラメータが無効でないかどうかを検査するコンストラクタが追加されています。このような検査は、このプラットフォームの以前のバージョンでは行われていませんでした。

   - *out* パラメータが null である場合、 PrintStream(OutputStream out) および PrintStream(OutputStream out, boolean autoFlush) コンストラクタは NullPointerException をスローします。
   - *in* パラメータが null である場合、InputStreamReader(InputStream in) および InputStreamReader(InputSteam in、String enc) コンストラクタは NullPointerException をスローします。
   - Java 2 SDK Solaris 版では、*lock* パラメータが null である場合、Reader(Object lock) および Writer(Object lock) コンストラクタは NullPointerException をスローします。

10. JDK 1.1 ソフトウェアでは、Thread.stop は、Object.wait または Thread.sleep でブロックされたスレッドの実行を中断することができます。ただし、Solaris 8 オペレーティング環境でネイティブスレッドを持つ Java 2 SDK Solaris 版では、Thread.stop がブロックされたスレッドの実行を中断することはできません。ただし、他のオペレーティングシステムや Solaris 2.6 オペレーティングプラットフォームでは、Java 2 SDK Solaris 版は Thread.stop に関して JDK 1.1 リリースと同じ動作をします。

11. Java 2 SDK Solaris 版では、クラスローディング機構が改定されています。新しいクラスローダでは、jar ファイルのパッケージに属するいずれかのクラスファイルが署名されている場合は、同じパッケージに属するすべてのクラスファイルが同じ署名者によって署名されていなければなりません。したがって、パッケージのあるクラスが署名されているのに、他のクラスが署名されていないか、または別の署名者によって署名されている jar ファイルは使用できなくなります。jar ファイルには署名されていないパッケージが含まれていてもかまいませんが、パッケージのいずれかのクラスが署名されている場合は、そのパッケージのすべてのクラスファイルが同じ署名者によって署名されていなければなりません。既存の jar ファイルがこの規則に従っていない場合は、これを Java 2 SDK Solaris 版または Runtime Environment で使用することはできません。

12. ネイティブ構成要素のフォアグラウンドとバックグラウンドの色は、SetForeground() と SetBackground() メソッドを使って明示的に設定

できます。これらの色を明示的に指定しないと、デフォルトの色は次のように設定されます。

- Java 2 プラットフォームでは、ネイティブ構成要素のデフォルト色は、使用中のオペレーティングシステムによって定義されている色になります。
- Java 2 SDK Solaris 版より前のバージョンでは、デフォルト色は Java プラットフォーム自体によって事前に定義されていました。

デフォルト色に依存するコードが JDK 1.1 でコンパイルされている場合には、これを Java 2 SDK Solaris 版で実行すると、構成要素の色がデフォルト色とは異なった色になり、ときには不適切な色になることがあります。たとえば、JDK 1.1 のコードで構成要素ラベルのフォアグラウンド色を白に明示的に設定し、デフォルトのバックグラウンド色を使用するとします。これを Java 2 SDK Solaris 版で実行すると、オペレーティングシステムのデフォルトのバックグラウンド色が白の場合、ラベルは見えません。

API の非互換性の問題

1. Java 2 SDK Solaris 版では、`ActiveEvent` クラスは `java.awt` パッケージにあります。以前は、`java.awt.peer` にありました。
2. Swing および Accessibility 関連のパッケージは、`com.sun.java.*` 名前空間から `javax.*` 名前空間に移動しました。これらのパッケージの新しい名前は以下のとおりです。
   - `javax.swing`
   - `javax.swing.border`
   - `javax.swing.colorchooser`
   - `javax.swing.event`
   - `javax.swing.filechooser`
   - `javax.swing.plaf`
   - `javax.swing.plaf.basic`
   - `javax.swing.plaf.metal`
   - `javax.swing.plaf.multi`
   - `javax.swing.table`
   - `javax.swing.text`
   - `javax.swing.text.html`

- javax.swing.tree
- javax.swing.undo
- javax.accessibility

  これらの Swing 1.0 のパッケージに対して古い com.sun.java.swing* 形式のパッケージ名を使用しているアプリケーションは、Java 2 SDK Solaris 版プラットフォームでは動作しません。新しい java.swing 形式のパッケージ名を使用するようにアプリケーションを更新してください。以上の変換に必要な PackageRenamer ツールは http://java.sun.com/products/jfc/PackageRenamer で入手できます。

注 - com.sun.java.swing.plaf.windows および com.sun.java.plaf.motif パッケージの名前は変更されていません。

古いパッケージ名を使用しているアプリケーションを強制的に Java 2 SDK Solaris 版プラットフォームで動作するようにすることができます。このためには、起動クラスパスの先頭に Swing 1.0 jar ファイルのパスを設定します。

```
java -Xbootclasspath:<path to 1.0 swingall.jar>:<path to Java 2 SDK
rt.jar> ...
```

注 - Java 2 SDK Solaris 版の Swing パッケージでは、Java 2 SDK Solaris 版のセキュリティ対応するための変更が加えられている点に注意してください。このため、セキュリティマネージャが存在する環境で、この方法を使用して、Java 2 SDK Solaris 版で Swing 1.0 クラス (すなわち、ブラウザ内のアプレットとして) を実行した場合は、プログラムが正しく動作しないことがあります。

---

3. Java 2 SDK Solaris 版では、java.awt.datransfer クラスの次のフィールドが final になっています (stringFlavor と plainTextFlavor フィールド)。

```
public static final DataFlavor stringFlavor
public static final DataFlavor plainTextFlavor
```

4. Java 2 SDK Solaris 版には、java.util.List インタフェースが含まれています。このため、既存のソースコードを変更しないと、java.awt.List と java.util.List の間で名前空間の競合が発生する可能性があります。

次の例に示すように、Java 2 SDK Solaris 版でワイルドカード付きの import 文をまとめて使用し、コードに無修飾名の `List` が含まれていると、コンパイルエラーになります。

```
import java.awt.*;
import java.util.*;
```

この問題を回避するには、次のような `import` 文を追加して、ファイル全体で衝突を解決するか、

```
import java.awt.List;
```

あるいはクラス名を使用するたびに適切なパッケージでクラス名を完全修飾します。

5. Java 2 SDK Solaris 版では、`java.awt.event.KeyEvent` クラスの `CHAR_UNDEFINED` フィールドの値は 0x0ffff です。JDK 1.1 リリースでは、このフィールドの値は 0x0 でした。この変更は、0 が有効な Unicode 文字で、`CHAR_UNDEFINED` の定義に使用できないためです。
6. Java 2 SDK Solaris 版では、`java.io.StringReader.ready` メソッドのシグニチャーが変更され、`StringReader` が閉じている場合に `IOException` をスローできるようになっています。`StringReader` クラスは、`abstract` クラスの拡張対象の `java.io.Reader` に記述されている一般規約を正しく実装しています。
7. Java 2 SDK Solaris 版の `Integer.decode()` および `Short.decode()` に関する仕様には、負数の正しい表現が明記されています。負数はかならず負符号 (-) で始めます。16 進数または 8 進数の場合は、基数指示子 (0x、#、0) は負符号の後ろに付けます。

   Java 2 SDK Solaris 版では、基数指示子の前または後ろに負符号を付けるように明確に規定していませんでした。実装状態では、基数指示子は負符号の前につくと想定されていました。

   たとえば、Java 2 SDK Solaris 版で *-0x5* をデコードすると負の 5 (*-5*) が返され、*0x-5* をデコードすると `NumberFormatException` がスローされます。JDK 1.1 では、結果は逆になり、*-0x5* では `NumberFormatException` がスローされ、*0x-5* では負の 5 (*-5*) が返されます。

   JDK 1.1 で使用されていたデコード規則は一般的なものではなく、文書化もされていませんでしたが、プログラムの中にはこの動作に依存しているものがあります。

8. Java 2 SDK Solaris 版では、クラス `java.util.Vector` と `java.util.Hashtable` が改良され、関連するインタフェースが新しい Collections Framework (それぞれ `java.util.List` と `java.util.Map`) に実装されました。その結果、equals メソッドと `hashCode` メソッドのセマンティクスが変更され、これらのメソッドでは `List.equals` と `Map.equals` に規定される一般的な規約に従って、参照の等価性の代わりに値の等価性が提供されます。

   これにより互換性の問題がいくつか生じます。

   - 「自己参照」 (`Vector` または `Hashtable` をそれ自身に挿入する) は、状況によってはスタックオーバーフローを引き起こすことがあります。

     a. `Hashtable` をキーとしてそれ自身に挿入すると `Hashtable` が壊されるため、その後の操作によってスタックオーバーフローが起ることがあります。(`Hashtable` のキーとして使われるオブジェクトを変更することは今までも許されていませんでした。オブジェクトを `Hashtable` に挿入すると等価の比較に影響があるため、これは変更とみなされます。)

     b. `Hashtable` にそれ自身が値 (または要素ともいう) として含まれていると、equals と `hashCode` が規約によって定義されていないため、`hashCode` メソッドや equals メソッドをこのような「自己参照的な」`Hashtable` に対して呼び出したときに、スタックオーバーフローになることがあります。

     c. `Vector` にそれ自身が要素として含まれていると、equals と `hashCode` が規約によって定義されていないため、スタックオーバーフローが起ることがあります。

   - equals と `hashCode` は `Vector` や `Hashtable` の内容に依存するようになったため、それらは同じコレクションに対する他の操作に対し同期化されます。Java 2 SDK Solaris 版より前は、これらのメソッドは同期化されませんでした。したがって、equals や `hashCode` が同期化されないことをクライアントが前提にしていると、この新しい同期化によって (プロセスの) 活動性の問題が生ずることがあります。

   - `Vector` や `Hashtable` に対する equals の「参照の等価性」に明示的に依存しているクライアントは、正しく動作しません。たとえば、プログラムに保持されている `Hashtable` のキーがすべて何らかのシステムの `Vector` (または `Hashtable`) だとします。今まではそれぞれの `Vector` (または `Hashtable`) は内容と関係なく個別のキーでした。しかし、現在は 2 つの `Vector` (または `Hashtable`) の内容が同じであれば、それらは同じものとみなされます。さらに、`Hashtable` 内でキーとして使用されている `Vector` や `Hashtable` を変更することはできなくなります。

- equals と hashCode は Vector や hashCode の内容全体を調べるため、大きなコレクションでは、従来よりも時間がかかることがあります。

9. 新しいバージョンの File クラスは、当初意図されていた動作と現在の一般的な用途の両方に対応しています。しかし、動作のわずかな違いによって、一部プログラムの実行が失敗することがあります。

   - 新しいバージョンの File クラスはまた、冗長な区切り文字 (パス名文字列の最後にある区切り文字など) を削除します。このため new File ("foo//bar/").getPath() 式の評価は、UNIX システムでは foo/bar になります。

10. Java 2 SDK Solaris 版では、java.lang.Thread の checkAccess メソッドがファイナルです。JDK 1.1 プラットフォームでは、 checkAccess はファイナルではありませんでした (バグ ID 4151102)。

11. Java 2 SDK Solaris 版では、配列を表すクラスで呼び出されたとき、class java.lang.Class の getInterfaces メソッドは Cloneable および Serializable クラスオブジェクトを含む配列を返します。JDK 1.1 では、getInterfaces は空の配列を返していました。

12. JDK 1.1 のときと異なり、Java 2 SDK Solaris 版では、abstract クラスの java.text.BreakIterator、java.text.SimpleTextBoundary、java.text.Collator (Collator のサブクラスの java.text.RuleBasedCollator も含めて) は、Serializable インタフェースを実装しません (バグ ID 4152965)。

13. Java 2 SDK Solaris 版では、PersonalJava™ プラットフォーム用の Input Method Framework の 1 機能として、java.awt.Graphics クラスに abstract メソッドの drawString(AttributedCharacterIterator,int,int) が追加されています。普通、1.1 プラットフォーム用に作成されたアプリケーションに Graphics のサブクラスが存在することはめったにありません。しかし、1.2 プラットフォームでも使用するには、そうしたサブクラスでこの新しい abstract メソッドを実装する必要があります (バグ ID 4128205)。

14. JDK 1.1 に実装されている String ハッシュ関数は、初版の JLS に規定されている関数と異なっていました。実際には、規定されている関数は、入力文字列の外部の文字を扱うという点において実装不可能です。また、実装されている関数は、URL などの一部文字列クラスでうまく動作しませんでした。

    実装を仕様に合わせて、動作上の問題を解決するために、仕様と実装の両方が修正されています。Java 2 SDK Solaris 版の String hash() 関数は、次のように規定されています。

```
s[0] * 31^(n-1) + s[1] * 31^(n-2) + ... + s[n-1]
```

s[i] は、文字列 *s* の *i* 番目の文字です。

通常、ほとんどのアプリケーションはこの変更の影響を受けません。実際の `String` ハッシュ値に依存する固定データがある場合、理論上、アプリケーションが影響を受ける可能性はありますが、`Hashtable` の直列化表現が、`Hashtable` に含まれるキーの実際のハッシュ値に依存することはありません。つまり、固定データの格納に関して直列化に依存するアプリケーションが影響を受けることはありません (バグ ID 4045622)。

15. Java 2 SDK Solaris 版は、NMI (Native Method Interface) をサポートしていません。

    JDK 1.0 がサポートする NMI を使用して作成した非 JNI ネイティブメソッドがあるか、JNI Invocation Interface を使用して実行環境を組み込むようにしている場合、Java 2 SDK Solaris 版で使用するには、ネイティブライブラリを再リンクする必要があります。Solaris 環境では、リンクコマンド行の `-ljava` を `-ljvm` に置き換える必要があります。また、JNI に合わせて書き直す必要があります。

注 - この変更は、JNI プログラミングにおけるネイティブメソッドの実装には影響ありません。

    Java 2 SDK Solaris 版では、Sun は JNI だけをサポートします。JNI は、ネイティブライブラリと Java プログラミング言語を連携させる標準の手段です。JNI は、VM から独立したネイティブコードを作成することを可能にします。メソッドで NMI を使用している場合は、第 3 章のネイティブメソッドの JNI への移行に関する説明を参照してください。

16. `Thread.suspend()` メソッドを使用すると、デッドロック状態になることがあります。このメソッドは本質的に安全ではないため、その使用は推奨されていません。これは、`Thread.stop()` や `Thread.resume()` などの、その他の非同期の thread メソッドについても同じです (バグ ID 4203325)。

    `Thread.suspend()` は安全ではありません。これは、たとえば、メソッドの同期中に、`Thread.suspend()` がスレッドの実行を中断させることがあるためです。この中断によって、他のスレッドがロックアウトされる可能性があり、その場合、デッドロック状態になることがあります (詳細については、`http://java.sun.com/products/jdk/1.2/ja/docs/ja/guide/misc/threadPrimitiveDeprecation.html` のリファレンスプラットフォームのマニュアルを参照してください)。

17. JDK 1.1 では、`Thread.stop` は、`Object.wait` または `Thread.sleep` でブロックされたスレッドの実行を中断することができます。ただし、ネイティブス

レッドを持つ Java 2 SDK Solaris 版では、`Thread.stop` がブロックされたスレッドの実行を中断することはできません。ただし、他のオペレーティングシステムまたはグリーン (非ネイティブ) スレッドを持つ Solaris 2.6 では、Java 2 SDK Solaris 版は `Thread.stop` に関して JDK 1.1 リリースと同じ動作をします。

18. ここでは、`java.sql` パッケージにインタフェースを実装する開発者 (主として JDBC ドライバを実装する開発者) に関係する互換性の問題を説明します。Java 2 SDK Solaris 版では、次のインタフェースに新しいメソッドが追加されています。

    - `java.sql.Connection`
    - `java.sql.DatabaseMetaData`
    - `java.sql.ResultSetMetaData`
    - `java.sql.ResultSet`
    - `java.sql.CallableStatement`
    - `java.sql.PreparedStatement`
    - `java.sql.Statement`

    これらインタフェースの JDK 1.1 版を実装するソースコードは、Java 2 SDK Solaris 版では正しくコンパイルされません。正しく動作するには、新しいメソッドを実装するように変更する必要があります。

    また、Java 2 SDK Solaris 版の次のインタフェースは新しい型をサポートします。

    - `java.sql.ResultSet`
    - `java.sql.CallableStatement`
    - `java.sql.PreparedStatement`

    これらのインタフェースを実装するソースコードや Java 2 SDK Solaris 版で正しくコンパイルされたコードは、JDK 1.1 では新しい型が存在しないため、正しくコンパイルされません。

19. public フィールド `serialVersionUID` が Java 2 SDK Solaris 版 Standard Edition v1.2 で `java.io.Serializable` インタフェースに導入されています。このフィールドはこのインタフェースに導入すべきではありませんでした。この意味は指定されていませんし、このフィールドは `java.io.Serializable` の仕様に反するものです。

    `java.io.Serializable` の API 仕様では次のように記述されています。

    “直列化インタフェースにはメソッドもフィールドもありません。

さらに、直列化仕様によると、クラスの直列バージョン UID を取得するには `ObjectStreamClass.lookup(className).getSerialVersionUID()` メソッドを使用する必要がありました。しかし、存在するかどうか不明なフィールドを使ってクラスの直列バージョン UID を取得しようとすることは意味がありません。

直列バージョン UID は計算されることもあれば、クラスに明示的に定義されることもありますが、スーパークラスやインタフェースから継承されることはありません。インタフェースのインスタンスを直接生成することはできないため、直列化システムにとってインタフェースの `serialVersionUID` フィールドは役に立ちません。実際、直列化システムがこのフィールドを使用することはありません。

このような理由から Java 2 SDK Solaris 版のバージョン 1.2.2 では、public `serialVersionUID` フィールドは除去されています。この変更が既存のアプリケーションに影響を与えることはないはずです。

ツールの互換性の問題

1. Java 2 SDK Solaris 版では、`javac –O` オプションは JDK 1.1 とは異なる意味を持ち、生成されたコードに対して性能面で異なる影響を与えることがあります。Java 2 SDK Solaris 版では、`javac –O` は、高速のコードを生成するようコンパイラに指示します。クラスにまたがってメソッドをインライン化したり、暗黙的に `–depend` をオンにしたり、暗黙的に `–g` をオフにしたりすることはありません。暗黙的に `–depend` をオンにすることはないため、`–depend` をオンにする必要がある場合は、コマンド行に `–depend` を追加する必要があります。
2. Java 2 SDK Solaris 版のクラスファイルを作成する場合は、JDK 1.1.4 以降に発表されたバージョンの `javac` コンパイラだけを使用してください。JDK 1.1 版の `javac` を使用すると、不正な内部クラス属性が生成されることがあります (Java 2 SDK Solaris 版の `javac` は正しい属性を生成します)。JDK 1.1.4 およびそれ以前の `javac` コンパイラは、正しい書式を検出したときにクラッシュする可能性があります。JDK 1.1.5 以降、`javac` は、旧版と新版両方の正しい属性に対応しています。これは、コンパイル時だけの問題です。Java 2 SDK Solaris 版のコンパイラは、古い VM で動作するクラスファイルを生成します。
3. Java 2 SDK Solaris 版では、`javakey` ツールの代わりに `keytool`、`PolicyTool`、`jarsigner` ツールが追加されています。これらの新しいツールについては、`http://java.sun.com/products/jdk/1.2/ja/docs/ja/guide/security/index.html` で、Java 2 SDK Solaris 版のセキュリティに関するドキュメントを参照してください。

4. JDK 1.1 では、javap –verify は、クラスファイルの部分検査を行なっていました。Java 2 SDK Solaris 版には、このオプションはありません。検査のほんの一部を実行するだけであるため、誤解を生みやすいオプションでした。
5. 1.1.4 より前の Java インタプリタでは、java c コマンドを使用して、/a/b/c.class 位置のクラスファイルを /a/b ディレクトリの中から呼び出すことができました (c クラスが a.b.* パッケージに含まれている場合も可能)。この場合、JDK 1.1.4 および Java 2 SDK Solaris 版では、完全修飾クラス名を指定する必要があります。たとえば、/a/b/c.class 位置にある a.b.c を呼び出すには、/a ディレクトリの親ディレクトリから java a.b.c コマンドを発行します。
6. JDK 1.1 ベースのリリースにバグがあるため、JDK 1.1 の javakey ツールを使って署名されたコードが Java 2 SDK Solaris 版では署名されていないものとみなされます。また、Java 2 SDK Solaris 版を使って署名されたコードが JDK 1.1 ベースのリリースでは署名されていないものとみなされます。
7. Java 2 SDK Solaris 版より前の javac では、矛盾するあるいは冗長なコマンド行オプションの組み合わせが見過ごされていました。たとえば、–classpath を複数回指定することができ、最後に指定されたオプションだけが有効になっていました。Java 2 SDK Solaris 版では、こうした動作はしません。
8. Java 2 SDK Solaris 版より前の Java インタプリタの –classpath オプションは、VM がシステムクラスをロードするときに使用する検索パスを設定していました。VM は、このパスに合わせて java.class.path プロパティを設定します。一般的に、アプリケーションクラスは、対応するクラスローダがなくても、システムクラスパスから直接呼び出されていました。

   インストールされている拡張機能と新しいセキュリティモデルの両方を活用するため、Java 2 SDK Solaris 版では、アプリケーションクラスローダからアプリケーションを起動します。このため、–classpath オプションは、アプリケーションクラスローダがクラスと資源をロードするときに使用するクラスパスを設定します。同様に、java.class.path プロパティには、このパスが反映されます。VM が内部的に使用するシステムクラスパスは、新しい –Xbootclasspath オプションを使用して無効にすることができます。ほとんどの場合、システムクラスパスを変更する必要はありません。

   通常、大部分のアプリケーションはこの変更の影響を受けません。ただし、java.class.path プロパティには、システムクラスをロードするときに使用されるディレクトリや JAR ファイルが含まれないことを忘れないでください。詳細は、新しい sun.boot.class.path プロパティを参照してください。独自のセキュリティマネージャをインストールするアプリケーションはこ

れによって悪影響を受けるかもしれません。このようなアプリケーションは Java 2 SDK Solaris 版用に書き換える必要がありますが、当面は、下位互換性のために –Xbootchasspath スイッチが提供されます。

9. Java 2 SDK Solaris 版の javadoc ツールは、パッケージレベルの API 出力に対して次の形式のファイル名を生成します。

   package-<*package name*>.html

   以前は、次の形式のファイル名が生成されていました。

   Package-<*package name*>.html

   たとえば、java.io パッケージに対するデフォルトのパッケージレベルの出力のファイル名は次のようになっていました。

   ```
   Package-java.io.html
   ```

   Java 2 SDK Solaris 版以降、このファイル名は次のようになります。

   ```
   package-java.io.html
   ```

10. NMI をサポートしなくなったため、Java 2 SDK Solaris 版の javah は異なるものになっています。–nmi フラグはありません。詳細は、49ページの「javah」を参照してください。

直列化の非互換性の問題

Externalizable オブジェクトの形式に関してまれに発生する問題に対処するため、Java 2 SDK Solaris 版では、互換性のない変更を行う必要がありました。JDK 1.1.5 (またはそれ以前) のプログラムは、Java 2 SDK Solaris 版の形式で生成された Externalizable オブジェクトを読み込もうとするときに、StreamCorruptedException をスローします。JDK ソフトウェアバージョン 1.1.6 以降のプログラムにはこの問題はありません。

ただし、Java 2 SDK Solaris 版の新しい API は、下位互換性をサポートしています。古い形式のストリームを作成するには、その前に次を呼び出します。

```
ObjectOutputStream.useProtocolVersion(PROTOCOL_VERSION_1)
```

第 3 章

Java Native Interface (JNI)

NMI から JNI への移行

Java 2 SDK Solaris 版は NMI をサポートしません。インストールによっては、NMI は JNI 1.0、JNI は JNI 1.1 と呼ばれます。

注 - ネイティブメソッドを含むアプリケーションはプラットフォームに依存し、「Write Once Run Anywhere™」ではありません。このことは、NMI または JNI のどちらが使用されているかに関係ありません。ネイティブメソッドのサポートが変更されても、アプレットには影響しません。

移植

NMI を使用するアプリケーションを JNI インタフェースを使用するように変更するのは簡単です。たとえば、不慣れな技術者でも 2 日ほどで 6000 行の C コードを移植しました。この例では、NMI は少し使われているだけでしたが、使用箇所は広く分散していました。

javah

`javah` コマンドは、ネイティブメソッドの実装に必要な C ヘッダとソースファイルを生成します。C プログラムは生成されたヘッダとソースファイルを使用して、ネイティブソースコードのオブジェクトのインスタンス変数を参照します。JNI に

は、ヘッダ情報やスタブファイルは必要ありません。引き続き javah コマンドで jni オプションを使用して、JNI 形式のネイティブメソッドに必要なネイティブメソッド関数プロトタイプを生成することができます。結果は、.h ファイルに書き込まれます。

JDK 1.1 のデフォルトのモードでは、javah コマンドは NMI 出力を生成します。

NMI インタフェースは Java 2 SDK Solaris 版の実装とまったく互換性がないため、Java 2 SDK Solaris 版で –old フラグを使用して、NMI 形式の出力を生成することはできません。Java 2 SDK Solaris 版では、–old フラグは解析され、見つかった場合は、-old not supported メッセージが生成され、javah は終了します。

JNI の一般的な問題

コンパイラに関する制限事項

ネイティブメソッドをコンパイルするときにコンパイラが行うことがある最適化によって、VM を実行できないことがあります。VM はスレッドのスタックにある関数のスタックフレーム検査機能に依存します。このため、ネイティブ関数のコードは、非リーフ関数に対するシステム呼び出し規則の規定にしたがってつねにスタックフレームを作成する必要があります。またフレームポインタレジスタは、つねに有効なスタックフレームを指す必要があります。

- Intel (IA) プラットフォーム上で Sun Workshop™ C コンパイラを使用すると、最適化レベルの -xO4 および -xO5 で上記の条件に違反するコードが生成されることがあります。このため、-xO3 より高い最適化レベルは使用しないでください。
- SPARC プラットフォーム上では、すべての最適化レベルで問題はありません。別のコンパイラを使用する場合は、そのコンパイラのドキュメントで上記の条件を満たすコードが生成されるかどうか確認してください。

注 - 上記の制限事項に違反しているネイティブメソッドは、VM を異常終了させることがあります。

ネイティブの Solaris アプリケーションのリンク

ネイティブの Solaris アプリケーションは、–lthread を使用してリンクしてください。そのようにしないと、不正な動作をすることがあります。

JNI を使用するネイティブアプリケーションをリンクするときは、libc.so ライブラリ (–lc オプション) の前に libthread.so ライブラリ (–lthread オプション) を指定する必要があります。Sun C コンパイラの –mt オプションは、–lthread オプションを自動的に追加します。一般的に –lc オプションは指定しません (デフォルトでリストの末尾になります)。このため、Sun のコンパイラやリンカを使用するときは、かならず –mt または –lthread オプションを指定しなければなりません。

JNI における配列への高速アクセス

Java 2 SDK Solaris 版では、Get*ArrayElements() 呼び出しの代わりに jni_GetPrimitiveArrayCritical() および jni_ReleasePrimitiveArrayCritical() を使用することによって JNI 配列へのアクセスを高速にすることができます。

- これらの *Critical() オペレーションを使用するコードは、次の制限事項を守っている必要があります。
- 配列の要素の取得後に速やかにそれらの要素を解放しなければならない
- 配列の要素を保持した状態で Java に戻ることはできない
- 他の JNI 操作を呼び出すことはできない

これらの制限事項があるため、データ固定やコピーを行わずに配列の要素にアクセスすることはできません。詳細は次のサイトを参照してください。

http://java.sun.com/products/jdk/1.2/ja/docs/ja/guide/jni/jni-12.html

共有ライブラリの格納場所

Solaris 2.5.1 または 2.6 ソフトウェアを使用していて、デフォルトの /usr/java1.2 以外の場所に JDK をインストールした場合、JNI のネイティブアプリケーションが JDK 共有ライブラリを検出できないことがあります。この問題は、LD_LIBRARY_PATH 環境変数に JDK のインストール先の jre/lib/sparc または jre/lig/i386 ディレクトリを含めることによって回避することができます。この問題は Solaris 7、Solaris 8 ソフトウェアでは発生しません。

シグナルの処理状態

JNI を使用するネイティブコードがシグナルの処理状態を変更してはなりません。VM はシグナルを使用します。シグナル処理に変更を加えると、VM で障害が発生することがあります。

第 4 章

Java 2 SDK と JDK 1.1 のコマンド行の相違点

VM 固有 (非標準) のオプション

Java 2 SDK のリファレンス実装では、オプションは 2 つのグループに分類されます。1 つのグループは特定の VM に固有のオプション、もう 1 つのグループはあらゆる VM に適用されるオプションです。各グループにはそれぞれ専用のオプション構文があります。特定の VM に固有のオプションはすべて、–Xdebug (デバッグを有効にするオプション) というように –X で始まります。–X だけを指定すると、その実装が受け付ける VM 固有の全オプションを示すヘルプメッセージが生成されます。

Java 2 SDK Solaris 版は、次に示す –X オプションをサポートしています。

表 4–1 現在使用できる –X オプション

-X	使用できるオプションを表示します。
-Xbootclasspath[/a|/p]:<*path*>	ブートクラスパスを設定します。現在の設定の先頭または末尾に追加することもできます。
-Xdebug	リモートデバッグを可能にします。
-Xmaxjitcodesize<*size*>	JIT コード領域の最大サイズをバイト単位で設定します。
-Xms<*size*>	Java ヒープの初期サイズを設定します。

表 4–1　現在使用できる –X オプション　続く

-Xmx<*size*>	Java ヒープの最大サイズを設定します。
-Xnoclassgc	クラスのガベージコレクションを無効にします。
-Xoptimize	(SPARC のみ) 試験的な目的でのみ使用してください。 JIT におけるメソッドの最適化により多くの時間をかけます。このオプションは長時間 CPU を使用するアプリケーションに効果があるので、アプリケーションの性能が向上する可能性があります。
-Xoss<*size*>	スレッドに対する Java スタックの最大サイズを設定します。
-Xrs	OS シグナルの使用量を抑制します。
-Xrunhprof[:file=<*file*>,depth=<*n*>]	java.hprof.txt または <*file*> にヒーププロファイルを出力します。
-Xsqnopause	SIGQUIT に対しユーザの対話のための一時停止を行いません。
-Xss<*size*>	スレッドに対するネイティブスタックの最大サイズを設定します。
-Xt	命令の追跡を有効にします。
-Xtm	メソッドの追跡を有効にします。

注 - –X オプションは予告なしに変更されることがあります。

これらのオプションの多くは、JDK 1.1.1 ～ 1.1.6 のオプションに対応しています。詳細は、次の節を参照してください。

–X オプションは VM に固有のオプションですが、その多くは一般的であり、通常の VM では用意されています。また、Java ソフトウェアがサポートしていて、Java 2 SDK Solaris 版がサポートしていないリファレンスプラットフォームの –X オプションには、次のものがあります。

-Xnoasyncgc	非同期ガベージコレクションを使用不可にします。

オプションの互換性

JDK 1.1 の次の互換性オプションは、Java 2 SDK Solaris 版でも使用することができます。

- –verbosegc
- –t
- –tm
- –debug
- –noasyncgc
- –noclassgc
- –verify
- –verifyremote
- –noverify
- –prof[:<*file*–>]
- –cs
- –checksource
- –ss
- –oss
- –ms
- –mx
- –l

oldjava ユーティリティ

oldjava ユーティリティは、JDK 1.1 ベースの java ユーティリティとの高レベルの互換性を提供します。oldjava を呼び出すと、–classpath コマンド行オプションおよび CLASSPATH 環境変数が JDK 1.1 リリースのときのように処理されます。Java 2 SDK Solaris 版のいくつかの新しいオプション (特に –jar オプション) は無効になります。

第 5 章

SIGQUIT によるデバッグ

Java 2 SDK Solaris 版には、新しいデバッグプロセスが採用されています。このプロセスでは、フォアグラウンドで動作する Java プロセスに SIGQUIT シグナルが送信されます。このシグナルを受けたプロセスは、次のメニューを表示して、一時停止し、ユーザ入力を待ちます。

```
1: Terminate program
2: Find & print one deadlock
3: Find & print all deadlocks
4: Print thread stacks
5: Print lock registry
6: Continue program
```

- Java レベルのデッドロック (2 つ以上のスレッドがモニタに対して周期的待ち状態になる) が発生しているかどうかを調べるには、2 番目と 3 番目のオプションを使用します。マルチスレッドの Java プログラムでは、同期メソッドを注意して使用しないと、この種のデッドロックが発生することがあります。
- 4 番目のオプションでは、システム上のアクティブなスレッドの一覧が提供され、それらスレッドの Java スタックトレースを選択して取得することができます。
- 5 番目のオプションでは、VM の内部ロックのダンプが生成されます。この情報は、VM に関連するバグなどの問題のレポートを作成するときに役立てることができます。

SIGQUIT シグナルは、kill(1) を使用するか、フォアグラウンドプロセスで Ctrl-バックスラッシュを入力することによって送信することができます。

バックグラウンドで動作する Java プロセスに SIGQUIT シグナルを送信した場合は、単に上記オプションの 3、4、5 の出力が標準エラーデバイスにダンプされるだけです。プログラムは、ユーザに入力を求めることなく動作を続行します。

Java プロセスが SIGQUIT を受信した後の一般的なユーザの対話としては以下が考えられます。

```
...
^\SIGQUIT
SIGQUIT が受信されると、ユーザは以下のどれかの操作を選択することができます。
1) terminate program
2) check & print one deadlock
3) check & print all deadlocks
4) dump thread stacks
5) dump lock registry
6) continue program
Select Action: 2
Found 0 deadlock

---------
行う処理を選択します。
  1) terminate program
  2) check & print one deadlock
  3) check & print all deadlocks
  4) dump thread stacks
  5) dump lock registry
  6) continue program
Select Action: 4

Java スレッドの一覧:
--------------------
[Thread# 1]     t@9:    (0xef715af4) GC 式のスレッドまたは破壊されたスレッド
[Thread# 2]     t@8:    "Thread-4"
[Thread# 3]     t@7:    "SoftReference sweeper"
[Thread# 4]     t@6:    "Finalizer"
[Thread# 5]     t@5:    "Reference handler"
[Thread# 6]     t@4:    "Signal dispatcher"
[Thread# 7]     t@1:    "main"
スレッドスタックをダンプするには、インデックス (1 ～ 7) を選択します。
7 以上の値を入力すると、前のメニューに戻る:   7

"main" (TID:0x36adc, sys_thread_t:0x36a50, state:R, thread_t: t@1, threadID:0x20f68, stac
k_bottom:0xf0000000, stack_size:0x20000) prio=5

[1] java.lang.Thread.yield(Thread.java)
[2] PingPoll.run(Compiled Code @ 0x117300)
[3] PingPoll.main(Compiled Code @ 0x1172c8)

--------------------
スレッドスタックをダンプするには、インデックス (1 ～ 7) を選択します。
7 以上の値を入力すると、前のメニューに戻る:   8
---------

行う処理を選択します。
  1) terminate program
```

(続く)

```
  2) check & print one deadlock
  3) check & print all deadlocks
  4) dump thread stacks
  5) dump lock registry
  6) continue program
Select Action: 6



---------
 Continuing Program
---------
...
```

付録 **A**

メモリの割り当てと制約

この付録では、Solaris 版 Java SDK のメモリ割り当て方法を説明します。メモリに関する制約としては、Solaris の 32 ビットのアドレス指定空間の制限とマシンのスワップ空間の制限があるだけです。

VM のサイズ

VM の 1 つのインスタンスのサイズは、次のようにして求めます。

- `long` および `double` フィールド = 8 バイト
- その他の型のフィールド = 4 バイト
- インスタンスサイズ = すべての非 `static` フィールド (継承フィールドを含む) 8 バイトヘッダのサイズ。配置およびその他の暗黙のコストなし
- `array` のサイズは、ヘッダ 12 バイト + その要素用のサイズ (4 バイトの倍数に切り上げ) になります。要素 1 つのサイズは次のようになります。

  `byte[]`, `boolean[]` = 1 バイト

  `short[]`, `char[]` = 2 バイト

  `long[]`, `double[]` = 8 バイト

  その他のすべての `array` = 4 バイト

  配列が、切り上げによって得られる 4 バイトを超えて整列されることはありません。

付録 **B**

–verbosegc の出力の説明

ガベージコレクションの問題の解決

–verbosegc オプションを使用して、アプリケーションによって新しい空間だけ、または新しい空間と古い空間の両方 (完全) のどちらのガベージコレクションが行われたかを調べることができます。このオプションは次のように使用します。

```
% java -verbosegc
```

この入力を行うと、次のような出力が生成されます。

```
GC[1] in 305 ms: (6144kb, 6% free) -> (14Mb, 61% free)
```

この例の GC[1] は、完全なガベージコレクションが行われたことを示します。GC[1] であれば、新しい空間だけのガベージコレクションになります。この例では、回収に 305ms の時間がかかっています。回収の開始時のヒープサイズは 6144K バイトで、未使用領域は 6% でした。回収中にヒープは大きくなり、回収の終了時のヒープサイズは 14M バイトで、未使用領域は 61% になっています。

世代別ヒープサイズ

ヒープは、新しい世代と古い世代に分類されます。それぞれの世代のサイズを調べるには、次のコマンドを使用します。

```
% java –verbosegc –verbosegc
```

出力例

次に、–verbosegc –verbosegc の出力例を検討します。

```
Gen0(semi-spaces): size=4096kb, free=0kb, maxAlloc=0kb
  From space: size=524288 words, used=524286 words, free=2
  To space:   size=524288 words, used=1 words, free=524287
Gen0(semi-spaces)-GC #4 tenure-thresh=0 61ms 0%->28% free
Gen0(semi-spaces): size=4096kb, free=571kb, maxAlloc=571kb
  From space: size=524288 words, used=378157 words, free=146131
  To space:   size=524288 words, used=1 words, free=524287
Gen1(mark-compact): size=4096kb, free=0kb, maxAlloc=0kb
Gen1(mark-compact)-GC #1 Gen1: 850kb dense
 262ms 0%->2% free
Gen1(mark-compact): size=4096kb, free=80kb, maxAlloc=80kb
```

この出力からは、それぞれの世代について以下の情報を得ることができます。

Gen0 は、ガベージコレクタをコピーする、4096K バイトから始まる semi-space です。この新しい空間のガベージコレクション以前には、未使用メモリのサイズは 0K バイトでした。ガベージコレクション以後には、378157 ワード分が未使用になっています (28%)。Gen1 は mark-compact のガベージコレクタです。回収中に 2% が解放されています。

詳しい情報が得られる、–verbosegc –verbosegc –verbosegc モードもあります。

付録 **C**

poller クラスの使い方

poller クラス

poller クラスのデモコードでは、C の poll(2) API の機能にアクセスする方法を提供します。このコードでは、正常な性能を維持しながら、できるだけの範囲で C の poll(2) API をミラー化します。Java とカーネルの間で入出力オブジェクトの大きな配列をやりとりすることによる影響を受けないように、ポーリングされるファイルやソケットの管理が JNI C コード (または /dev/poll デバイスドライバがある場合はそれ自体) に移されました。 poller クラスデモディレクトリにある README.txt ファイルに、このデモコードの詳しい使い方が記載されています。

poller クラスの基本的な使い方

```
Poller Mux = new Poller();

int serverFd = Mux.add(serverSocket, Poller.POLLIN);
int fd1 = Mux.add(socket1, Poller.POLLIN);
...
int fdN = Mux.add(socketN, Poller.POLLIN);
long timeout = 1000; // 1 秒

int numEvents = Mux.waitMultiple(100, fds, revents, timeout);

for (int i = 0; i < numEvents; i++) {
   /*
```

(続く)

```
        * より洗練させる必要がある
        */
        if (fds[i] == serverFd) {
            System.out.println("Got new connection.");
            newSocket = serverSocket.accept();
            newSocketFd = Mux.add(newSocket,Poller.POLLIN);
        } else if (fds[i] == fd1) {
            System.out.println("Got data on socket1");
            socket1.getInputStream().read(byteArray);
            // fd1 接続の状態に基づいて処理を行う
        }
        ...
    }
```

付録 **D**

Java 2 SDK と JDK 1.1 をともに実行する

/user/java シンボリックリンクは、複数の Java 環境がインストールされている場合の Solaris システムにおけるデフォルトの Java 環境の定義に使用します。現在、JDK 1.1 は /usr/java1.1 にインストールされ、Java 2 SDK Solaris 版は /usr/java1.2 にインストールされます。

Solaris 8 より前のリリースの場合、JDK 1.1 と Java 2 SDK Solaris 版をインストールするとき、/usr/java シンボリックリンクは /usr/java1.1 を参照しました。Solaris 8 リリース以降、JDK 1.1 と Java 2 SDK Solaris 版をインストールする場合、/usr/java シンボリックリンクは、デフォルトで /usr/java1.2 を参照します。

/usr/bin (/bin でも可) には、/usr/java を使用するシンボリックリンクがあるので (たとえば /usr/bin/java は /usr/java/bin/java を表します)、この /usr/java リンクでは、ほとんどのユーザに表示されるデフォルトの「java」を変更できます。Java アプリケーションの多くが Java 2 SDK Solaris 版または JDK 1.1 のどちらかで実行しますが、ユーザやアプリケーション側がどの「java」を使用するかを選択したい場合もあります。

JDK 1.1 を使用したい Java ユーザの場合、PATH 設定で /usr/bin の前に /usr/java1.1/bin を追加します。Java 2 SDK Solaris 版を使用したい Java ユーザの場合、PATH 設定で /usr/bin の前に /usr/java1.2/bin を追加します。場合によっては、CLASSPATH、LD_LIBRARY_PATH、または JAVA_HOME などの他の環境変数を変更することもできます。ただしこれらの環境変数はどれも必須ではありません。

JDK 1.1 が必要な Java アプリケーションでは、/usr/java1.1 を参照し、Java 2 SDK Solaris 版が必要なアプリケーションでは /usr/java1.2 を参照します。

シンボリックリンク /usr/java は変更しないでください。シンボリックリンクを変更すると、Solaris 8 より前の Solaris リリース対応の Java アプリケーションや、JDK 1.1 の使用を前提とした Java アプリケーションでは問題が生じることがあります。

索引

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S