Java 2 SDK 開発ガイド (Solaris 編)

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–3973–10
2002 年 5 月

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

Part No: 806-7930-10

Revision A

目次

はじめに

このマニュアルでは、Solaris™ オペレーティング環境用 Java™ 2 SDK, Standard Edition バージョン 1.4 の新しい機能と強化された機能を紹介し、その概要を説明します。

対象読者

このマニュアルは、Solaris オペレーティング環境で Java 2 SDK, Standard Edition を使用するアプリケーション開発者を対象としています。Java 2 SDK, Standard Edition ソフトウェアは、企業環境におけるサーバおよびクライアント側の Java テクノロジアプリケーションが優れた性能を発揮できるように最適化されています。

このマニュアルは Java 2 SDK 文書の一部です。Java 2 SDK 文書は http://java.sun.com/j2se/1.4/ja/docs/ja/index.html で入手できます。 最終的なリリース段階では、このオンライン文書が Java 2 SDK, Standard Edition バージョン 1.4 製品の正式な説明になります。

内容の紹介

第 1 章では、製品の特徴と強化された機能を示します。

第 2 章では、互換性の問題について説明します。

第 3 章では、仮想マシンの性能を最大限に活用するためのコマンド行オプションについて説明します。

第 4 章では、Java™ 2 プラットフォームにおける新しいアサーション機能について説明します。

関連項目

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。

- 『*Solaris Java Plug-in* ユーザーズガイド』
- 『*Java 2 SDK, Standard Edition v. 1.4* リリースノート』
  http://java.sun.com/j2se/1.4/ja/relnotes.html にあるオンライン文書
- 『*Java 2 SDK, Standard Edition, v. 1.4* ドキュメント』
  http://java.sun.com/j2se/1.4/ja/docs/ja/index.html にあるオンライン文書
- 『*Java 2 Platform, Standard Edition, v 1.4 API Specification*』
  http://java.sun.com/j2se/1.4/docs/api/index.html にあるオンライン文書

Sun のオンラインマニュアル

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、検索を行うこともできます。URL は、`http://docs.sun.com` です。

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%`

表 **P–1** 表記上の規則　*(続き)*

字体または記号	意味	例
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'`

コード例は次のように表示されます。

- C シェル

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

- C シェルのスーパーユーザー

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

- Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

  ```
  # 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。

第 1 章

新しい機能と強化された機能

Java 2 SDK, Standard Edition (J2SE™) バージョン 1.4 の新しい機能を以下に示します。機能全体の一覧 (Java 2 SDK, Standard Edition の以前のバージョンから繰り越された機能も含む) については、http://java.sun.com/j2se/1.4/ja/docs/ja/index.html を参照してください。

Java 2 Platform

XML 処理

Java 2 Platform には XML 処理用の Java API が追加されました。J2SE 1.4.0 は基本的に XML 文書の処理を API 標準セットでサポートします。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/xml/index.html を参照してください。

New I / O API

新しい入出力 (NIO) API は、バッファの管理、文字セットのサポート、正規表現のマッチング、ファイルの入出力、およびスケーラブルなネットワークの入出力に新しい機能と強化された機能を提供します。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/nio/index.html を参照してください。

セキュリティ

- Java™ Cryptography Extension (JCE)、Java™ Secure Socket Extension (JSSE)、および Java™ Authentication and Authorization Service (JAAS) のセキュリティ機能はオプションパッケージではなく、J2SE v1.4 に統合されました。
- 新しいセキュリティ機能は次の 2 つです。
  - Java™ GSS-APIを使用すると、Kerberos 5 機構を使用して通信するアプリケーション間でメッセージを安全に交換できます。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/security/jgss/tutorials/index.html を参照してください。
  - Java™ Certification Path API の `java.security.cert` パッケージには、認証パス (「認証チェーン」とも呼ぶ) を構築および検証できる新しいクラスとメソッドが含まれています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/security/certpath

/CertPathProgGuide.html を参照してください。

- 輸入管理制限により、J2SE v1.4 に付属する JCE 権限ポリシーファイルは「強力」であるが使用される暗号化を制限します。JCE 権限ポリシーファイルには、「無制限」のバージョン (つまり、暗号化の強度に制限をかけない) もあります。
- このリリースの JSSE 実装には、強力な暗号群が含まれています。しかし、米国の輸出管理制限のため、デフォルトの `SSLSocketFactory` と `SSLServerSocketFactory` を変更することは禁止されています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/security/jsse/JSSERefGuide.html にある『*JSSE* リファレンスガイド』を参照してください。
- JAAS が J2SE に統合されているので、`java.security.Policy` API がプリンシパルベースの照会を処理し、また、デフォルトのポリシー実装がプリンシパルベースの許可エントリをサポートします。したがって、アクセス制御は、どのコードが実行しているかだけではなく、だれがそのコードを実行しているかにより行うことができます。
- 動的ポリシーのサポートが追加されました。バージョン 1.4 より前の J2SE リリースでは、クラスのアクセス権へのバインドは静的に行われており、セキュリティポリシーはクラスのロード中に照会されていました。したがって、バインドの寿命はクラスローダの寿命によって制限されていました。バージョン 1.4 では、このバインドはセキュリティ検査により必要となるまで行われません。つまり、バインドの寿命はセキュリティポリシーの寿命によって制限されます。

J2SE 1.4 におけるセキュリティの詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/security/index.html を参照してください。

Java 2D™ テクノロジ

Java 2D™ 技術には、性能の向上、オフスクリーン画像のハードウェア高速化のサポート、プラグイン可能な画像入出力フレームワーク、新しい印刷サービス API、いくつかの新しいフォント機能など、新しい機能が数多く含まれています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/2d/new_features.html を参照してください。

Image I/O フレームワーク

Java の Image I/O フレームワークが提供するプラグイン可能なアーキテクチャを使用すれば、ファイルに格納されている画像にネットワーク経由でアクセスして処理できます。このフレームワークでは、画像のロードと保存のための API が J2SE 1.4 以前のものよりもはるかに柔軟で強力な機能を提供します。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/imageio/index.html を参照してください。

Java 印刷サービス API

Java 印刷サービスは新しい Java 印刷サービス API であり、クライアントとサーバのアプリケーションは次のことを行います。

- 能力に適した印刷サービスの検出および選択
- 印刷データの形式の指定
- 印刷する文書タイプをサポートするサービスへの、印刷ジョブの実行依頼

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jps/index.html を参照してください。

AWT

Abstract Window Toolkit (AWT) パッケージセンターが変更されて、グラフィカルユーザインタフェースを提供するプログラムの堅牢性、動作、および性能が改善されました。次の点が改善されています。

- 新しいフォーカスアーキテクチャが実装され、プラットフォームの不一致や AWT と Swing の構成要素間の非互換性が原因であったフォーカス関連のバグが数多く修正されました。
- 新しい全画面排他モードでは、ウィンドウシステムを中断し、画面に直接描画することによって、高性能なグラフィックスをサポートします。このモードは、描画が多いアプリケーション (ゲームなど) で効果的です。
- ディスプレイ、キーボード、マウスをグラフィックス環境でサポートできるかどうかを示す新しいグラフィックス環境メソッドによって、ヘッドレスサポートが有効になりました。
- フレームの外観の指定を完全に制御する必要があるアプリケーションのために、ネイティブのフレーム装飾を無効にできるようになりました。この機能を有効にすると、ネイティブの画面構成要素 (タイトルバー、システムメニュー、ボーダーなど) を描画しません。
- 新しい組み込み Java のサポートによって、マウスホイール (マウスのアジャストボタンの代わりに付いているスクロール用ホイール) によるスクロールが有効になりました。また、新しいマウスホイールリスナークラスによって、マウスホイールの動作をカスタマイズできるようになりました。
- AWT パッケージは完全な 64 ビット準拠に変更されて、64 ビットアドレスと 32 ビットアドレスの両方の Solaris マシン上で動作できるようになりました。

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/awt/AWTChanges.html を参照してください。

Swing

Swing には、新しい機能が数多く追加されました。

- スピナーコンポーネントは単一行の入力フィールドであり、このフィールドの小さな上下の矢印ボタンを使用することで、ユーザは一連の値の中から 1 つの番号または値を選択できます。
- 新しい書式付きテキストフィールドコンポーネントを使用すると、10 進数の通貨値だけを受け入れるテキストフィールドなどのように、日付、数字、および文字列を書式化できます。
- 新しいドラッグ&ドロップアーキテクチャによって、コンポーネント間でシームレスなドラッグ&ドロップを行うことができます。また、ユーザがカスタマイズした Swing コンポーネントにも簡単にドラッグ&ドロップを実装できます。一対のメソッドを作成し、自分のデータモデルの詳細を記述するだけで使用できます。

Swing では、いくつかの機能が強化されました。

- 進捗バーコンポーネントは不確定な状態を表示できるように強化されました。不確定進捗バーは完全度を表示するのではなく、絶えず動くアニメーションを使用することで、時間のかかる操作が進行していることをユーザに伝えます。
- タブ区画コンポーネントはスクロール可能なタブをサポートするように強化されました。この機能を有効にすると、すべてのタブが単一のタブ表示区域内に収まらない場合、タブ区画構成要素は複数のタブ (表示区域) を重ねて表示するのではなく、単一のスクロール可能なタブ (表示区域) を表示します。
- Popup クラスと Popup Factory クラスは以前は非公開でしたが、今回は公開されているので、開発者は独自のポップアップをカスタマイズまたは作成できます。
- 新しいフォーカスアーキテクチャが Swing に完全に統合されました。

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/swing/SwingChanges.html を参照してください。

ドラッグ&ドロップ

Swing は、アプリケーション間のデータ転送に新しいサポートを追加しました。ドラッグ&ドロップ操作とは、グラフィカルなポインティングデバイスでのジェスチャーによって特定されるデータ転送要求のことです。コピー / ペーストの場合、データ転送は多くの場合キーボードから行われます。データ転送には、ドラッグ&ドロップによるデータ転送と、カット / コピー / ペーストによるクリップボード経由のデータ転送の 2 つの形式があります。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/swing/1.4/dnd.html を参照してください。

ロギング API

Java の ロギング API は顧客サイトにおけるソフトウェアのサービスおよび保守を簡単にするものであり、エンドユーザ、システム管理者、フィールドサービスエンジニア、およびソフトウェア開発チームによる解析に適したログレポートを生成します。

ロギング API は、アプリケーションまたはプラットフォームにおけるセキュリティの障害、構成のエラー、性能のボトルネック、バグなどの情報を捕捉します。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/util/logging/index.html を参照してください。

Java™ Web Start 製品

Java Web Start 製品は J2SE 1.4.0 にバンドルされている新しいアプリケーション配備テクノロジです。Java Web Start を使用すると、Web ページのリンクをクリックするだけでアプリケーションを起動できます。アプリケーションがユーザのコンピュータ上に存在しない場合、Java Web Start は自動的に必要なファイルをすべてダウンロードします。そして、Java Web Start はダウンロードしたファイルをユーザのコンピュータに格納するので、ユーザはいつでもそのアプリケーションをデスクトップ上のアイコン、または Web ページのリンクから起動することができます。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jws/index.html を参照してください。

JavaBeans™ コンポーネントの長期的な持続性

新しい持続モデルは、JavaBeans のグラフを持続形式にまたは接続形式から変換するときのプロセスを処理するように設計されています。新しい API は JavaBeans コンポーネントのグラフのアーカイブをそのプロパティのテキスト表現として作成するのに適しています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/beans/changes14.html を参照してください。

JDBC™ 3.0 API

JDBC™ 3.0 API には `java.sql` と `javax.sql` のパッケージが含まれており、Java プログラミング言語からの汎用データアクセスを提供します。JDBC 3.0 API を使用すると、リレーショナルデータベースからスプレッドシートや通常のファイルまで、ほとんどどのデータソースにもアクセスできます。さらに、JDBC テクノロジは、ツールや代替インタフェースが構築することができる共通の基礎を提供します。

JDBC 3.0 API を使用すると、次のことを行うことができます。

- トランザクション内のセーブポイントの設定
- トランザクションがコミットされた後でも、結果セットを開いたままにしておく
- 準備済み文の再利用
- 準備済み文へのパラメータについてのメタデータの取得
- 自動的に生成される鍵の取得
- 一度に複数の結果セットを開く

新しい JDBC データタイプには `BOOLEAN` と `DATALINK` の 2 つがあります。`DATALINK` タイプを使用すると、データソースの外からデータを管理できます。このリリースはまた、JDBC Service Provider Interface と Connector アーキテクチャ間に関係を確立します。

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jdbc/index.html を参照してください。

アサーション機能

Java 2 Platform にはアサーション機能が追加されました。アサーションとは、プログラマがコンピュータプログラムの状態について真であると信じていることを示すブール型式のことです。たとえば、一覧をソートした後、プログラマはその一覧が昇順であることを表明できます。実行時にアサーションを評価して妥当性を確認すると、プログラムの動作についてのプログラマの思い違いを素早く見つけることができるので、コードの品質を向上させるためのもっとも強力なツールの 1 つとなります。詳細については、このマニュアルの「アサーション機能」を参照してください。

Preferences API

Preferences API は、ユーザの設定および構成データを管理するための新しい簡単な API です。アプリケーションは異なるユーザ、環境、およびニーズに適合するために設定および構成データを必要とします。アプリケーションはこのような設定および構成データを格納、取得、および変更するための方法を必要とします。このニーズに適合するのが Preferences API です。Preferences API は `java.util.Properties` クラスのもっとも一般的な使用方法を置き換えるものであり、このクラスの軽量さを保持しながら、さまざまな点を訂正しています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/lang/preferences.html を参照してください。

推奨標準優先機構

推奨標準とは、Java Community Process (JCP) 以外の標準化プロセスで定義された Java API のことです。推奨標準は JCP 以外で定義されているので、Java 2 Platform のリリース間に改訂されることがあります。推奨標準優先機構を使用すると、開発者とソフトウェアベンダーは推奨標準の新しいリビジョンの恩恵を受けることができ、Sun からリリースされている Java 2 Platform に含まれている推奨標準よりも新しいバージョンを提供できます。推奨標準優先機構の詳細については、http://java.sun.com/j2se/1.4/docs/guide/standards/ にある Web 上のドキュメントを参照してください。

64 ビットサポート

Java HotSpot™ Server VM を使用する場合、Solaris オペレーティング環境 (SPARC 版) の J2SE 1.4.0 は 64 ビット SPARC V9 プラットフォーム上での 64 ビット動作をサポートします。これによって、4G バイト (32 ビット VM がサポートできる絶対最大値) より大きなヒープをサポートできます。Java HotSpot Server VM では、適切なコマンド行フラグを使用することによって、32 ビットまたは 64 ビットのどちらかの動作をさせることができます。64 ビット VM を使用する場合、ユーザのプログラムが参照変数をアクセスするのに費やす時間によって異なりますが、約 15% から 25% の性能の低下が見られることがあります。64 ビット VM を使用する場合、J2SE 1.4.0 は 32 ビット共有ライブラリをサポートしません。ネイティブ (Java Native Interface) コードは 64 ビットモードでコンパイルし直す必要があります。

Java™ HotSpot 仮想マシン (VM)

このリリースの Java 仮想マシンでは、いくつかの機能が強化されました。

- シグナルチェーン機能
- 64 ビット SPARC V9 プラットフォーム用の 64 ビットサポート
- エラー報告機構
- Java Native Interface (JNI) 関数の追加チェックを実行するための新しいコマンド行オプション
- ガベージコレクションのイベントを記録するための新しい機能

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/vm/index.html を参照してください。

性能

このリリースでは性能も強化されました。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/performance/index.html を参照してください。

ネットワークのサポート (IPv6 を含む)

新しい機能には TCP と UDP に基づくアプリケーションにおける IPv6 のサポートと接続 (バインド) されていないソケットのサポートが含まれており、より柔軟にソケットを作成、バインド、および接続できるようになりました。Java Secure Socket Extension という機構はソケット経由で送信されるデータを暗号化し、新しいクラス `URI` はプロトコルハンドラが存在しなくても URI を構築および構文解析できるようにします。FTP Protocol Handler は現在の標準に準拠するように徹底的に見直されました。デフォルトの文字セットは現在 UTF8 であり、他の文字スキームを有効にするための API が追加されました。

新しいクラス `NetworkInterface` を使用するとインタフェースとアドレスを列挙できるようになり、`InetAddress` の JNDI DNS SP Support を使用すると Pure Java ネームサービスプロバイダを構成できるようになります。TCP 帯域外データは従来のアプリケーションをサポートします。つまり、UDP Connection 機能は宛先アドレスを OS に登録するので、非同期エラーを UDP ソケットに戻すことができます。また、完全な SOCkS V5 と V4 TCP サポートには、どのバージョンを使用するかについてのプロキシとの自動ネゴシエーションが含まれています。さらに、ストリーミング、要求ヘッダーと応答ヘッダーの処理、およびエラー処理において改良が行われました。

詳細について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/net/enhancements14.html を参照してください。

RMI

RMI ランタイム実装では現在、以前のリリースで行われていたクライアント側のスタックトレース情報の保存に加えて、リモート呼び出しからスローされた例外についてのサーバ側のスタックトレース情報も保存します。したがって、このような例外がクライアントコードにアクセス可能になったとき、クライアントのスタックトレースには、クライアント側のトレースデータに続く、オリジナルのサーバ側のトレースデータがすべて含まれています。

J2SE 1.4.0 では、`java.rmi.server.RMIClassLoader` の特定の静的なメソッドが自分の動作を新しいサービスプロバイダインタフェース `java.rmi.server.RMIClassLoaderSpi` に委譲します。このサービスプロバイダオブジェクトを構成すると、特定のアプリケーションに対する RMI の動的クラスロード動作を強化できます。デフォルトでは、サービスプロバイダは `RMIClassLoader` のすべての静的なメソッドについて標準の動作を実装します。

現在では、`java.rmi.server.hostname` プロパティを動的に更新することによって、将来のエクスポートで新しいホスト名を使用することを示すことができます。したがって、新しいホスト名は上記プロパティが更新された後にエクスポートされたオブジェクトのスタブに含まれるようになります。

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/rmi/relnotes.html を参照してください。

直列化

このリリースの直列化 API の機能には、次の変更と強化が行われました。

- データ直列化ストリームで共有解除されることが判明しているオブジェクトの直列化解除のサポート
- クラス定義された readObjectNoData メソッドのサポート
- 重要なバグの修正

詳細について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/serialization/relnotes14.html を参照してください。

Java Naming and Directory Interface™ (JNDI)

J2SE 1.4.0 では、Java Naming and Directory Interface™ (JNDI) に次の機能強化が行われました。

- Internet Domain Naming System (DNS) サービスプロバイダは J2SE 1.4.0 の一部になりました。 この構成要素によって、アプリケーションは DNS に格納されたデータが読めるようになります。
- JNDI Lightweight Directory Access Protocol (LDAP) サービスプロバイダのセキュリティが強化されて、アプリケーションは既存の LDAP 接続上で安全なセッションを確立し、異なる認証プロトコルを使用できるようになりました。
- JNDI CORBA Object Services (COS) ネーミングサービスプロバイダは Interoperable Naming Service (INS) 仕様 (99-12-03) をサポートします。

詳細について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jndi/index.html#14changes を参照してください。

CORBA、Java™ IDL、および RMI-IIOP

Java 2 Platform の一部として出荷される ORB には現在、Portable Object Adapter (POA) 機能が含まれています。ORB を使用すると、クライアントは、同じまたは異なるマシン上で動作しているサーバがサポートしているオブジェクト上で、メソッド呼び出しを行うことができます。POA 機能を使用すると、プログラマは異なる ORB 製品間で移植可能なオブジェクト実装を構築できるので、たとえば、持続性のある識別情報を持つオブジェクトをサポートできます。ほかにも、Portable Interceptors、Interoperable Naming Service、GIOP 1.2 のサポート、Dynamic Management of Any values などの新しい機能や、持続ネーミングサービスなどの機能をサポートする新しいツールが含まれます。

J2SE v.1.3 と J2SE v.1.4 における Java IDL の違いの詳細について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/idl/jidlChanges.html を参照してください。一般的な情報について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/idl/index.html を参照してください。

Java™ Platform Debugger Architecture 製品

機能強化された Java Platform Debugger Architecture を含んだ J2SE 1.4.0 は、次の新しい機能を持つようになりました。

- Java HotSpot™ 仮想マシンは現在「フルスピードデバッグ」を使用します。以前のバージョンの Java HotSpot では、デバッグが有効であると、プログラムはインタプリタを使用した場合にのみ実行できました。現在では、Java HotSpot 技術の性能を完全に活用でき、デバッグを有効にしたままプログラムを実行できるようになりました。性能が改善されたことによって、実行に時間がかかるプログラムでも簡単にデバッグできます。また、テストをフルスピードで行い、デバッガを起動して例外を発生させることもできます。
- HotSwap 機能が追加されて、デバッガの制御下にある間でもクラスを更新できるようになりました。
- `EventRequests` は現在インスタンスフィルタを指定できるようになり、要求によって生成されるイベントを、現在実行しているインスタンスが指定されたオブジェクトであるイベントだけに制限できるようになりました。
- Java Platform Debugger Architecture が強化されて、Java プログラミング言語以外のソースを Java プログラミング言語ソースに変換してデバッグできるようになりました。
- ターゲットの VM 終了通知を制御する要求を発行できるようになって、クリーンシャットダウン同期を行うことができるようになりました。

国際化

J2SE 1.4.0 における文字の処理はバージョン 3.0 の Unicode 標準に基づいています。このような文字の処理は、`java.lang` パッケージ内の `Character` クラスと `String` クラスに影響を与えると同時に、`java.text` パッケージ内の照合と双方向テキスト解析機能にも影響を与えます。

J2SE 1.4.0 は Thai (タイ語) と Hindi (ヒンディー語) をすべての機能においてサポートします。サポートされるロケールと書記法の詳細については、『サポートされているロケール』オンライン文書を参照してください。

クラス `java.util.Currency` が導入されて、ロケールとは無関係に通貨を参照できるようになりました。`java.text.NumberFormat` には、金額を書式化するための通貨を指定する新しいメソッドとそれに関連するクラスが追加されました。

Java™ Plug-in 製品

Java Plug-in 1.4 には、次の新しい機能が追加されました。

- ブラウザではなく、Java Secure Socket Extension (JSSE) で HTTPS をサポートできるようになりました。
- アプレットのキャッシュ方法が強化されて、JAR やクラスファイルに加えて、GIF、JPEG、および AU などのファイルをキャッシュできるようになりました。
- ブラウザのページが変わっても、アプレットは持続するようになりました。

- アプレットの互換性が様々な点で強化されて、ほとんどの JDK 1.1 アプレットをシームレスに Java 2 で実行できるようになりました。

Java Plug-in 1.4 は次の機能を提供します。

- w3c 定義の標準インタフェースで DOM にアクセス可能
- アサーションとロギングのサポート
- アプレットのユーザビリティが強化され、アプレットのロードの進捗バーをカスタマイズ可能
- Java Console のオプションが強化され、アプレットの動作中でも、プロキシの設定やポリシーファイルなどを動的に再構成可能

詳細について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/plugin/index.html を参照してください。

Collections フレームワーク

J2SE 1.4 で強化された Collections フレームワークの機能には、ランダムアクセスを公示するためのマーカーインタフェース、同値性ではなく同一性に基づいた Map、挿入順に保存する Map と Set の実装、および一覧の値を操作および返すためのいくつかの新しいアルゴリズムが含まれます。詳細について
は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/collections/changes4.html を参照してください。

ユーザ補助機能

J2SE 1.4.0 は、次の分野でユーザ補助機能の新しいサポートを提供します。

- JTabbedPane 上ではニーモニックタブで移動可能
- ユーザ補助技術によりテキストを編集可能
- HTML コンポーネントにアクセス可能
- Swing アクションにアクセス可能
- リスト項目の最初の文字を使用してリストを移動可能
- コンポーネントの機能 DATE_EDITOR、FONT_CHOOSER、GROUP_BOX、SPIN_BOX、STATUS_BAR が追加
- 画面拡大機能または画面読取機能の存在を示す属性と、Java 仮想マシンにロードするユーザ補助技術を指定する属性の追加

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/access/index.html を参照してください。

正規表現

新しいパッケージ `java.util.regex` には、正規表現で指定されたパターンに文字シーケンスを一致させるためのクラスが含まれています。詳細については、http://java.sun.com/j2se/1.4/docs/api/java/util/regex/package-summary.html にある `java.util.regex` 用の API 指定を参照してください。

数値演算

呼び出し側が確実性を指定しなくても素数を生成できる新しい効率的なメソッドが `java.math.BigInteger` に追加されました。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/math/enhancements14.html を参照してください。

リフレクション

一部のリフレクション操作、特に、`java.lang.reflect.Field`、`java.lang.reflect.Method.invoke()`、`Java.lang.reflect.Constructor.newInstance()`、および `Class.newInstance()` がより高い性能のために書き直されました。リフレクションの呼び出しとインスタンス化は以前のリリースと比べて何倍も速くなっています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/reflection/index.html を参照してください。

Java™ Native Interface (JNI)

J2SE 1.4 では、Java Native Interface (JNI) が強化されて、`java.nio` パッケージの新しい機能「ダイレクトバッファ」を反映するようになりました。ダイレクトバッファの内容は潜在的に、通常のガベージコレクトされたヒープの外側にあるネイティブメモリ内に格納できます。また、新しい呼び出しインタフェースルーチン `AttachCurrentTreadAsDaemon` を使用すると、ネイティブコードはデーモンスレッドを仮想マシンに接続できます。これは、シャットダウン時に、このスレッドが終了するのを VM が待つ必要がないときに便利です。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jni/jni-14.html にある『*JNI Enhancements*』オンライン文書を参照してください。

ツールとユーティリティ

詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/tooldocs/tools-changes.html にある『ツールの変更点』オンライン文書を参照してください。

- *java* アプリケーション起動ツールは、新しいアサーション機能をサポートするためのコマンド行オプションをサポートします。Solaris™ オペレーティング環境 (SPARC™ 版) では、64 ビットまたは 32 ビットの動作を指定するための新しいコマンド行オプションが利用できます。新しいオプション `-Xloggc:`*file* は各ガベージコレクションイベントを指定されたファイルに記録します。新しいオプション `-Xcheck:jni` は、少し性能が落ちますが、Java Native Interface (JNI) 機能の追加チェックを実行します。
- *javadoc* ツールには、いくつかの新しいタグ、十数個の新しいオプション、doc コメントのより洗練された継承、HTML 出力に対するより多くの制御、ドックレット API の改良、よりわかりやすいエラーメッセージ、および数十個のバグフィックスが含まれており、より実行しやすくなっています。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/tooldocs/javadoc/whatsnew-1.4.html にある『*Javadoc 1.4* の新機能』オンライン文書を参照してください。別の製品としてダウンロードできる MIF ドックレット (http://java.sun.com/j2se/javadoc/mifdoclet) は大幅に更新されて、API 文書が FrameMaker と PDF の形式で作成されています。
- *native2ascii* ツールには、Unicode エンコーディングで符号化されたソースファイルを正しく処理できるバグフィックスが含まれています。
- *idlj* ツールは現在、Portable Servant Inheritance Model に従ってサーバ側バインディングを生成します。この変更によって、デフォルトでは新しい POA バインディングが生成されます。下位互換性のあるサーバ側バインディングを生成するために、新しいコマンド行オプションが追加されました。Portable Servant Inheritance Model の詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/idl/POA.html を参照してください。*idlj* ツールの詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/rmi-iiop/toJavaPortableUG.html を参照してください。
- *orbd* (*Object Request Broker Daemon*) ツールは、一時ネームサービス `tnameserv` に代わる新しいツールです。ORBD には、一時ネームサービスと持続ネームサービスの両方が含まれています。`orbd` ツールを使用すると、クライアントは CORBA 環境内のサーバ上にある持続オブジェクトを透過的に配置して呼び出すことができます。`orbd` ツールはサーバマネージャ、相互運用ネームサービス、およびブートストラップネームサーバの機能を持っています。 `servertool` と一緒に使用すると、クライアントがサーバにアクセスしたい場合、サーバマネージャがサーバを検出、登録、および起動します。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/idl/orbd.html を参照してください。
- `servertool` が J2SE 1.4 で新たに追加されました。 `servertool` は、サーバを登録、登録解除、起動、およびシャットダウンするための使いやすいインタフェースをアプリケーションプログラマに提供します。詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/idl/servertool.html を参照してください。
- `rmic` コンパイラには、Remote Method Invocation をサポートする Portable Object Adapter (POA) を有効にするための新しいオプションが追加されました。POA は、ベンダーの ORB (あるいは、他の使用) 間における移植を可能にします。POA の詳細について

は、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/idl/POA.html を参照してください。`rmic` コンパイラで POA サポートを有効にするには、引数 `rmic -iiop -poa` を使用します。`rmic` コンパイラの詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/tooldocs/solaris/rmic.html を参照してください。

- グラフィカルな *Policy Tool* ユーティリティは強化されて、指定されたアクセス制御権をどのユーザに与えるかを示すプリンシパルフィールドを指定できるようになりました。

第 2 章

以前のリリースとの互換性

このマニュアルでは、次のトピックについて説明します。

Java 2 Platform のバージョン 1.3 と 1.2 の非互換性については、http://java.sun.com/j2se/1.3/ja/compatibility.html にある Java 2 Platform バージョン 1.3 の互換性に関するドキュメントを参照してください。

Java 2 Platform のバージョン 1.2 と 1.1 の非互換性については、http://java.sun.com/j2se/products/jdk/1.2/ja/compatibility.html にある Java 2 Platform バージョン 1.2 の互換性に関するドキュメントを参照してください。

Java 2 Platform のバージョン 1.1 と 1.0 の非互換性については、http://java.sun.com/products/jdk/1.1/compatibility.html にある JDK™ 1.1 ソフトウェアの互換性に関するドキュメントを参照してください。

バイナリ互換性

Java™ 2 SDK, Standard Edition, (J2SE™) バージョン 1.4 は、次に示す非互換性を除いて、J2SE 1.3 との上位バイナリ互換性を持っています。つまり、この非互換性を除けば、バージョン 1.3 コンパイラで構築したクラスファイルは J2SE 1.4 でも正しく動作します。

一般的に、ポリシーは次のとおりです。

- 同じファミリ (たとえば、1.2.x) 内の保守リリース (たとえば、1.2.1 と 1.2.2) は上位と下位のバイナリ互換性を持っています。

- 同じファミリ (たとえば、1.x) 内の機能リリース (たとえば、1.3 と 1.4) は上位バイナリ互換性を持っていますが、必ずしも下位バイナリ互換性を持っているとは限りません。

初期のバイトコードオブファスケータの中には、仮想マシンの仕様とは異なる形式のクラスファイルを生成するものがありました。このような不適切な形式のクラスファイルは J2SE の仮想マシン上では動作しませんが、初期バージョンの仮想マシン上では動作するものもあります。この問題を修復するには、適切な形式のクラスファイルを生成する新しいオブファスケータでクラスファイルを生成し直します。

ソース互換性

J2SE 1.4 は、次に示す非互換性を除いて、初期バージョンとのソースの上位互換性を持っています。この非互換性を除けば、初期リリースで定義された言語機能や API を使用するように書かれたソースファイルは J2SE 1.4 でもコンパイルおよび実行できます。

ソースの下位互換性はサポートされません。新しい言語機能や Java 2 Platform API を使用するソースファイルは、初期バージョンの Java 2 Platform では使用できません。

一般的に、ポリシーは次のとおりです。

- 保守リリースは新しい言語機能や API を導入しないので、上位と下位の両方のソース互換性を持っています。
- 機能リリースとメジャーリリースはソースの上位互換性を持っていますが、ソースの下位互換性は持っていません。

非推奨 API は下位互換性だけをサポートするメソッドやクラスであり、このような API を使用していると、`-nowarn` コマンド行オプションを指定していない限り、コンパイラは警告メッセージを生成します。非推奨メソッドやクラスを使用しないようにプログラムを変更することが推奨されますが、現在のところ、このようなメソッドやクラスをシステムから完全に削除する計画はありません。

`sun.*` パッケージ内のいくつかの API は変更されました。これらの API は開発者向けではありません。 `sun.*` パッケージからインポートする場合、開発者は自分の責任で行うようにしてください。詳細については、http://java.sun.com/products/jdk/faq/faq-sun-packages.html にある『*Why Developers Should Not Write Programs That Call* `sun.*` *Packages*』を参照してください。

Java 2 Platform, Standard Edition, バージョン 1.4 における非互換性

J2SE 1.4 は以前のバージョンの Java 2 Platform との強力な互換性を持っています。既存のプログラムは変更しなくても、ほとんどすべて、J2SE 1.4 上で動作するはずです。しかし、まれな状況や「コーナーケース」で発生するいくつかのマイナーな潜在的な非互換性も存在しているためその状況に対処できるように補足としてここで説明します。

1. Java 2 Platform バージョン 1.4 以降、クラス `javax.swing.tree.DefaultTreeModel` は null のルートノードの設定が可能です。以前のバージョンでは、`TreeModel` の仕様で null のルートが有効であると示されているのにもかかわらず、`DefaultTreeModel` は null のルートを許可しませんでした。`DefaultTreeModel` は現在、null のルート設定が可能であり、コンストラクタ内の null のルート設定も可能です。この変更の一部として、`DefaultTreeModel.setRoot()` の仕様も改訂されました。`DefaultTreeModel.setRoot()` の古い仕様は次のとおりです。

   ルートをルートに設定します。これによって、ルートが null の場合、`IllegalArgumentException` がスローされます。

   `DefaultTreeModel.setRoot()` の新しい仕様は次のとおりです。

   ルートをルートに設定します。ルートが null の場合、ツリーには何も表示されませんが、正当であることを意味します。

2. 直列化可能な内部クラスにそのクラスオブジェクトへの明示的な参照が含まれる場合、そのクラスのシリアルバージョン UID の値は J2SE 1.3 と J2SE 1.4 で異なります。この違いは、J2SE 1.3 と J2SE 1.4 との間で `javac` コンパイラで行われた変更がシリアルバージョン UID の計算に影響を与えているためです。

   この問題を回避するには、明示的なシリアルバージョン UID を直列化可能なクラスに追加することが推奨されます。`serialver` ツールを使用すると、J2SE 1.3 の `javac` コンパイラでコンパイルされたクラスのシリアルバージョン UID を取得できます。

3. J2SE 1.4.0 以降、クラス `javax.swing.text.DefaultHighlighter` の public static フィールド `DefaultPainter` は `final` です。 以前のバージョンの Java 2 Platform, Standard Edition では、このフィールドは `final` ではありませんでした。

4. J2SE 1.4.0 では、HTML フォームが Java 2 Platform の実装に内部的にモデル化される方法が変更されました。 以前のバージョンでは、フォームの属性はすべての子の文字要素の属性セットに格納されていました。J2SE 1.4.0 では、HTML ファイル自身のフォームによりフィットするフォームを表す要素が作成されます。これによって、フォームがより良くモデル化されるようになり、フォームを一貫して書き出せるようになります。この変更は、バグ 4200439 に対処するために行われました。

この変更は、正確に処理されていないフォームを使用している開発者に影響します。たとえば、次のような無効な HTML を想定します。

```
<table>
<form>
</table>
</form>
```

1.4.0 より前の実装では、次のように扱われます。

```
<form>
<table>
</table>
</form>
```

J2SE 1.4.0 では、次のように扱われます。

```
<table>
<form>
</form>
</table>
```

この変更は多くの開発者にはあまり影響しないように見えますが、自分のコードを更新する必要がある可能性はあります。以前、リーフの要素の属性がフォームの属性に含まれていると想定していた場合、1.4.0 以降、このような属性はフォームの要素の属性セットから取得する必要があります。

5. J2SE 1.4.0 以降、クラス `java.awt.event.MouseEvent` の `static final` フィールド `MOUSE_LAST` の値は `507` に変更されました。以前のバージョンの Java 2 Platform では、`MOUSE_LAST` の値は `506` でした。

   コンパイラは `static final` の値をコンパイル時にハードコード化するので、`MOUSE_LAST` を参照する、1.4.0 より前のバージョンの `java.awt.event.MouseEvent` でコンパイルされたコードは古い値のままです。このコードを J2SE 1.4.0 で動作させるには、バージョン 1.4.0 のコンパイラでコンパイルし直す必要があります。

6. J2SE 1.4.0 で出荷される CORBA テクノロジ用の API は OMG ドキュメントで指定されている CORBA 2.3 マッピングに準拠するように変更されています (OMG ドキュメントについては、http://java.sun.com/j2se/1.4/ja/compatibility-CORBA.html にある『*CORBA Compatibility Information*』オンラインドキュメントを参照)。J2SE v1.3 と v1.4.0 の間で行われた CORBA 機能に関連する API の変更については、上記ドキュメントと同様に、J2SE 1.4.0 が準拠しているすべての OMG 仕様の一覧を参照してください。

7. J2SE 1.4.0 以降、 `ObjectOutputStream.putFields` または `ObjectOutputStream.writeUnshared` を無効にするサブクラスによって (直接的または間接的に) 呼び出される場合、`ObjectOutputStream` の引数を 1 つ取る public コンストラクタは「`enableSubclassImplementation`」である `SerializablePermission` を必要とします。

また、J2SE 1.4.0 以降、ObjectInputStream.readFields または ObjectInputStream.readUnshared を無効にするサブクラスによって (直接的または間接的に) 呼び出される場合、ObjectInputStream の引数を 1 つ取る public コンストラクタは SerializablePermission の「enableSubclassImplementation」を必要とします。

この変更はほとんどのアプリケーションにはあまり影響ありません。しかし、putFields または readFields メソッドをオーバーライドするが、直列化インフラストラクチャの残りの部分はオーバーライドしない ObjectInputStream/ObjectOutputStream サブクラスは影響を受けます。

8. J2SE 1.4.0 の Javac バイトコードコンパイラは以前のバージョンよりも厳密に Java 言語仕様に準拠しています。Java 言語仕様に厳密に準拠しない既存のコードは、初期バージョンのコンパイラでコンパイルできていた場合でも、新しいコンパイラではコンパイルできないことがあります。

   次に、J2SE 1.4.0 コンパイラがより厳密である例を示します。

   - 新しいコンパイラは到達不能の空文を検出します (Java 言語仕様で定義されているとおりに)。次に、現在のコンパイラが検出および拒否する一般的な例を 2 つ示します。

     ```
     return 0;/* exit success */;
     ```
     および

     ```
     {
         return f();
     } catch (Whatever e) {
         throw new Whatever2();
     };
     ```

     2 つの例には両方ともに余分なセミコロンが付いていて、新しいコンパイラはこのセミコロンを到達不能の空文であると見なします。さらに、自動生成されるソースコードは到達不能の空文を生成することがあります。

   - 新しいコンパイラは、名前のない名前空間からタイプをインポートする import 文を拒否します。以前のバージョンのコンパイラは、言語仕様では許可されていないにもかかわらず (import 節に現れるタイプ名はスコープ内ではないため)、このような import 宣言を受け入れていました。仕様には、単純名は import 文では使用できず、また、名前のない名前空間からはインポートできないと明確に宣言されています。

     要約すると、

     ```
     import SimpleName;
     ```

     この構文は現在有効ではありません。また、

     ```
     import ClassInUnnamedNamespace.Nested;
     ```

     この構文も現在有効ではなく、名前のない名前空間からは入れ子になったクラスをインポートしません。このような問題を修正するには、すべてのクラスを名前のない名前空間から名前付きの名前空間に移動します。

9. J2SE 1.4.0 では、javac バイトコードコンパイラはデフォルトの動作として、以前の「-target 1.1」ではなく、「-target 1.2」を使用します。これらのオプションの動作については、http://java.sun.com/j2se/1.4/ja/docs/ja/tooldocs/solaris/javac.html にある javac コンパイラのリファレンスページを参照してください。「-target 1.2」に関連する変更の 1 つとして、クラスが実装されていないメソッドをインタフェースから継承する場合、コンパイラは現在、メソッド宣言をクラスファイルに生成および挿入しません。このように挿入されるメソッドは、他のすべての非公開メソッドのように、デフォルトの serialVersionUID 計算に含まれています。結果として、インタフェースを直接実装するが、そのメソッドのうち 1 つでも実装しない抽象的な直列化可能クラスを定義した場合、そのデフォルトの serialVersionUID 値は、J2SE 1.4 の javac または以前の javac のどちらでコンパイルしたかによって異なります。

   このような初期バージョンの javac によって挿入されるメソッドに関する情報については、http://java.sun.com/jdc/bugParade/bugs/4043008.html にあるバグ 4043008 の報告を参照してください。

10. ソース互換性 - JDBC 3.0 API (J2SE 1.4 の一部として含まれる) は 2 つの新しいインタフェースを導入し、いくつかの新しいメソッドを既存のインタフェースに追加しました。初期バージョンの JDBC API を使用するドライバおよびアプリケーションは J2SE 1.4 とバイナリ互換性を持っており、何の問題もなく動作します。しかし、JDBC 3.0 API で行われた変更にはソース互換性がありません。JDBC インタフェースを実装するドライバおよびアプリケーションを正しく構築するには、この変更を反映させるように更新する必要があります。『JDBC 3.0 Specification』の第 6 章には、JDBC 3.0 API に準拠する (つまり、J2SE 1.4 とソース互換性を持つ) ために行う必要があるすべての事項が一覧表示されています。

11. J2SE 1.4 より前では、ファイルタイプが判明しており、応答コードが 400 以上になった場合、FileNotFoundException がスローされます。そうでない場合、例外はスローされません。J2SE 1.4 で実装された正しい動作は、URLConnection.getInputStream に関しては、ファイルタイプにかかわらず、すべての HTTP エラーに対して IOException をスローし、HTTP 応答がリソースが見つからないことを示している場合だけ、IOException のサブクラスである FileNotFoundException をスローします。言い換えると、FileNotFoundException がスローされるのは、応答コードが 404 または 410 の場合だけです。この変更の一部として、現在、HttpURLConnection.getErrorStream を使用すると、サーバから戻されたエラーページを読み取ることができます。J2SE 1.4 より前では、getErrorStream は常に null を戻していました。また、メソッド HttpURLConnection.getResponseCode は J2SE 1.4 で正しく動作します。

12. J2SE 1.4 では、assert キーワードが Java プログラミング言語に追加されました。新しいキーワードのため、「assert」を識別子として使用している既存のプログラムは J2SE 1.4 に準拠しません。しかし、このキーワードが追加されたからと言って、既存のバイナリ (.class ファイル) を使用しても問題は発生しません。(「assert」が正式な識別子であった) J2SE 1.4 より前のリリースから (「assert」が正式な識別子でない) J2SE 1.4 への移行を簡単にするため、J2SE 1.4 の Javac バイトコードコンパイラは 2 つの動作モードをサポートしています。

通常動作モードでは、コンパイラは以前のリリース (J2SE 1.3) の仕様に準拠するプログラムを受け入れます。アサーションが許可されていないので、`assert` キーワードが識別子として使用されている場合、コンパイラは警告を生成します。

代替動作モードでは、コンパイラは J2SE 1.4 の仕様に準拠するプログラムを受け入れます。アサーションが許可されているので、`assert` キーワードが識別子として使用されている場合、コンパイラはエラーメッセージを生成します。

アサーションを有効にするには、`-source 1.4` コマンド行スイッチを使用します。このフラグがない場合、ソース互換性を最大限にするために、デフォルトの動作は「1.3」になります。1.3 とのソース互換性はいずれサポートされなくなる予定です。

13. インタフェース `java.applet.AppletContext` の API 仕様は変更されて、アプレット開発者はブラウザセッション中にデータやオブジェクトをストリームしながら持続的に使用できるようになりました。これによって、開発者は静的なクラスを使用してデータやオブジェクトをキャッシュする必要がなくなりましたが、潜在的なバイナリ非互換性が生じています。`AppletContext` インタフェースを実装するクラスを含む既存のアプリケーションはすべて、新しい `AppletContext` 仕様と互換性がありません。このようなクラスは、改訂された `AppletContext` API を実装するように変更する必要があります。実際には、通常、`AppletContext` を実装しているアプリケーションは、Java™ Plug-in やアプレットビューアなど、アプレットコンテナとして動作するアプリケーションだけです。したがって、この潜在的な非互換性の影響は最小限になると予想されます。

14. J2SE 1.4 が Socket API に行なった重大な変更の 1 つとして、`SocketImpl` 抽象クラスに新しい抽象メソッドが追加されました。新しいメソッドのため、J2SE 1.4 よりも前に `SocketImpl` のサブクラスを作成していた場合、新しいメソッドに提供される実装が存在しないので、1.4 の `javac` ではコンパイルが失敗します。 バイナリ互換性は保たれているので、このサブクラスの既存のクラスファイルは予想どおりに動作します。

    `SocketImpl` のサブクラスを作成しているアプリケーションは少数であり、したがって、この潜在的な非互換性の影響は最小限になると予想されます。`SocketImpl` のサブクラスを作成している開発者がこの変更に対処するには、次の 2 つの方法があります。

    - J2SE 1.3.x (以前) でコンパイルしたクラスファイルを使用する
    - 新しいメソッドに実装を提供する

第 3 章

Java HotSpot VM オプション

この章では、Java HotSpot™ 仮想マシンの性能特性に影響するコマンド行オプションと環境変数について説明します。特に注記しない限り、このドキュメントのすべての情報は Java HotSpot Client VM と Java HotSpot Server VM の両方に適用されます。Java プラットフォームのガベージコレクション (GC)、スレッド化、および性能に関する FAQ の詳細については、http://java.sun.com/docs/hotspot を参照してください。

この章は次の節からなります。

Java HotSpot VM オプションのカテゴリ

Java HotSpot VM が認識する標準オプションについては、Java アプリケーション起動ツール (java ユーティリティ) のマニュアルページや http://java.sun.com/j2se/1.4/ja/docs/ja/tooldocs/solaris/java.html にあるオンラインドキュメントを参照してください。この章では、Java HotSpot VM が認識する非標準オプションについて説明します。

- `-X` から始まるオプションは非標準であり (つまり、必ずしもすべての VM 実装でサポートされることが保証されていない)、Java 2 SDK の今後のリリースで予告なしに変更される可能性があります。
- `-XX` オプションが正しく動作するためには特別なシステム条件があり、システム構成パラメータにアクセスできる特権が必要なため、一般的には使用しないことが推奨されます。このようなオプションも予告なしに変更される可能性があります。

Exact VM オプションと同等な Java HotSpot VM オプション

Java HotSpot VM に関連する事項の詳細については、http://java.sun.com/j2se/1.4/ja/docs/ja/guide/vm/index.html にあるオンラインドキュメントを参照してください。

バージョン 1.3.0 より前では、Java 2 SDK (Solaris オペレーティング環境用) の製品リリースには Exact VM (EVM) という仮想マシン実装が付属していました。バージョン 1.3.0 以降では、Exact VM の代わりに Java HotSpot VM が付属しています。

Exact VM でサポートされていたオプションは、Java HotSpot VM では名前が変わったり、廃止されたりしています。次の表に、Java 2 SDK v 1.4.0 でサポートされている EVM オプションと同等な Java HotSpot VM オプションを示します。

EVM オプション	説明	同等な **Java HotSpot VM** オプション
`-Xt`	命令のトレース	なし (廃止されたオプション)
`-Xtm`	メソッドのトレース	なし (廃止されたオプション)
`-Xoss`	Java スタックの最大サイズ	なし (Java HotSpot VM はネイティブのスタックと Java プログラミング言語のスタックを別々に持っていない。)

`-Xverifyheap`	ヒープの整合性を確認する。	`-XX:+VerifyBeforeGC`
		`-XX:+VerifyAfterGC`
		`-XX:+VerifyTLAB`
		`-XX:+VerifyBeforeScavenge`
		`-XX:+VerifyAfterScavenge` (すべてデバッグのみ)

`-Xmaxjitcodesize`	コンパイルされるコードの最大サイズ	`-Xmaxjitcodesize` (以前は `-Xmaxjitcodesize=32m`、現在は `-Xmaxjitcodesize32m`)
`-Xgenconfig`	ヒープを構成する。	(次の 37 ページの「`-Xgenconfig` オプション」を参照。)
`-Xoptimize`	JIT 最適化コンパイラを使用する。	`-server`
`-Xconcgc`	並行ガベージコレクタ (1.2.2_07+) を使用する。	なし (1.3、1.4 ではまだない。)

Java HotSpot VM は現在、Exact VM ではサポートされていなかった次の -X オプションを認識します。

オプション	説明
-Xincgc	Train GC を使用する。
-Xnoincgc	Train GC を使用しない (デフォルト)
-Xmaxf<*Maximum*>	縮小を避けるための GC 後の空きヒープの最大パーセンテージ (デフォルトは 0.7)
-Xminf<*Minimum*>	拡大を避けるための GC 後の空きヒープの最小パーセンテージ (デフォルトは 0.4)
-Xint	インタープリタのみ
-Xboundthreads	ユーザレベルのスレッドをバインドする (1.4 ではデフォルト、1.3 ではデフォルトでない)
-Xmn<*Size*>	若い世代のサイズを設定する (1.4 のみ)

-Xgenconfig オプション

Java HotSpot VM の若い世代は eden と 2 つのサイズが同じセミスペースから構成されますが、EVM の若い世代は 2 つのサイズが同じセミスペースから構成されます (eden なし)。コマンド -Xgenconfig:32m,64m,semispaces:128m,512m ,markcompact は、それぞれ 32M バイトで始まり、64M バイトまで拡張可能な 64M バイトのセミスペースが 2 つあり、また、128M バイトで始まり、512M バイトまで拡張可能な古い世代があることを示しています。これは、(最大) サイズが 640M バイトのヒープを作成します。Java HotSpot VM では、これと同等なコマンドは「-Xms256m -Xmx640m -XX:NewSize=32m -XX:MaxNewSize=64m」です。見てわかるように、-Xgenconfig を使用するときには各世代のサイズを指定する必要がありましたが、Java HotSpot VM ではまず、ヒープの合計サイズを -Xms/-Xmx で設定し、その後に若い世代をその領域から切り出しています。

_JIT_ARGS 環境変数と同等な Java HotSpot VM オプション

ほとんどの _JIT_ARGS 環境変数は内部的なデバッグ専用のオプションであり、Java HotSpot VM には対応するオプションがありません。問題を突き止めるためのもっとも簡単な方法は、最初に導入したときに不安定の原因となり、内部のテストグループによって使用された可能性がある最適化のフォームをいくつかオフにしてみることです。

_JIT_ARGS 環境	Java HotSpot VM オプション	説明
jit/jbe	-client/- server	jbe は 1.2 ベースのシステムにおける -Xoptimize と同じで、jit はデフォルト。1.2 の -Xoptimize (または jbe) の代わりに -server を使用する。
trace	-XX:+PrintCompilation	コンパイル時にメソッドをトレースする (1.3 ではデバッグのみ、1.4 では利用可能)。
V8/V9	-XX:+UseV8InstrsOnly	これらのフラグを使用して強制的に指定する。両システムで自動的に判別 (Sparc/デバッグのみ)。

_JVM_ARGS 環境変数と同等な Java HotSpot VM オプション

_JVM_ARGS 環境	Java HotSpot VM オプション	説明
bound_threads	-Xboundthreads	このオプションはすべてのスレッドが結合スレッドとして生成されるようにする (1.4 ではデフォルト、1.3 ではデフォルトでない)。
fixed_size_young_gen	- Xmn<size>	若い世代のリサイズを無効にする。Java HotSpot VM でこれを行うには、1.4 の場合、若い世代のサイズを定数に設定する。1.3 の場合、-XX:NewSize=<size> -XX:MaxNewSize=<size> を使用する。
gc_stats	-verbose:gc および/または -XX:+PrintGCDetails	gc 統計収集の様々なフォームを有効にする。
ims_concurrent	なし	
inline_instrs	-XX:MaxInlineSize =<size>	インライン化されるメソッド内のバイトコード命令の最大数を指定する整数。

_JVM_ARGS 環境	Java HotSpot VM オプション	説明
inline_print	-XX:+PrintInlining	インライン化されるメソッドについてのメッセージを出力する (デバッグのみ)。
no_parallel_gc	なし	
sync_final	なし	
yield_interval	-XX:DontYieldALotInterval=<ms>	(デバッグのみ) 生成の間隔 (ミリ秒単位)
monitor_order	なし	

追加の Java HotSpot VM 引数

数字には、キロバイトを表す「k」または「K」、メガバイトを表す「m」または「M」、ギガバイトを表す「g」または「G」、およびテラバイトを表す「t」または「T」を付けることができます (たとえば、32k は 32768 と同じ)。 boolean 型のフラグをオンにするには -XX:+<option> を、オフにするには -XX:-<option> を使用します。

フラグとデフォルト	説明
-XX:-AllowUserSignalHandlers	アプリケーションがシグナルハンドラをインストールしても (エラーまたは警告) メッセージを出さない。
-XX:AltStackSize=16384	代替シグナルスタックのサイズ (K バイト単位)
-XX:+MaxFDLimit	ファイル記述子の数を最大までに上げる。
-XX:MaxHeapFreeRatio=70	縮小を避けるための GC 後の空きヒープの最大パーセンテージ
-XX:MinHeapFreeRatio=40	拡大を避けるための GC 後の空きヒープの最小パーセンテージ
-XX:ReservedCodeCacheSize=32m	コードキャッシュの予約サイズ (バイト単位) — コードキャッシュの最大サイズ
-XX:+UseBoundThreads	ユーザレベルのスレッドを LWP にバインドする (1.4 ではデフォルト)。
-XX:+UseLWPSynchronization	スレッドではなく、LWP に基づく同期を使用する (1.4 ではデフォルト)。
-XX:+UseThreadPriorities	ネイティブのスレッド優先順位を使用する。

フラグとデフォルト	説明
-XXMaxPermSize=64m	恒久的な世代のサイズ
-XX:-CITime	JIT コンパイラが費やした時間 (1.4 のみ)
-XX:-PrintTenuringDistribution	保有期間情報を出力する。
-XX:TargetSurvivorRatio=50	収集後に使用される survivor 空間の希望パーセンテージ
-XX:-DisableExplicitGC	System.gc() への呼び出しを無効にする。それでも VM は必要なときにガベージコレクションを実行する。
-XX:-OverrideDefaultLibthread	Solaris 9 では、このオプションは必要ない。Solaris 8 では、J2SE™ バージョン 1.3.1 _02+ と 1.4+ が代替スレッドライブラリを使用するときにこのオプションを必要とする。このオプションは Solaris 8 より前のオペレーティング環境上では利用できない。

スレッドライブラリの詳細については、http://java.sun.com/docs/hotspot/threads/threads.html にあるスレッドに関するドキュメントを参照してください。

これらのフラグはアーキテクチャまたは OS ごとに異なります。「フラグとデフォルト」は「Sparc/-server」のデフォルトです。

フラグとデフォルト	説明
-XX:CompileThreshold=10000	コンパイル前のメソッド呼び出し/分岐の数 [10,000 — Sparc サーバ、1,000 — Sparc クライアント、1,500 — x86 クライアント]
-XX:MaxNewSize=unlimited	新しい世代の最大サイズ (バイト単位) [32M — Sparc、2.5M — x86 (1.3 の場合)、無制限 (1.4 の場合。これは現在では NewRatio が MaxNewSize を決定するため。)]
-XX:NewRatio=2	新しい世代と古い世代のサイズの比率 [Sparc -server:2、Sparc -client: 4 (1.3 の場合)、8 (1.3.1+ の場合)、x86: 12]
-XX:NewSize=2228224	新しい世代のデフォルトのサイズ (バイト単位) [Sparc 2.125M、x86: 640K]
-XX:SurvivorRatio=64	eden 空間と survivor 空間のサイズの比率 [Solaris: 64]

フラグとデフォルト	説明
`-XX:ThreadStackSize=512`	スレッドスタックのサイズ (K バイト単位) (0 はデフォルトのスタックサイズを使用することを意味する。) [Solaris Sparc 32 ビット: 512、Solaris Sparc 64 ビット: 1024、Solaris x86: 256]
`-XX:+UseTLAB` (`J2SE 1.3` では `XX:+UseTLE`)	スレッドローカルなオブジェクト割り当てを使用する [Sparc -server: 真、その他すべて: 偽]
`-XX:+UseISM`	http://java.sun.com/docs/hotspot/ism.html にある『*Intimate Shared Memory*』オンラインドキュメントを参照。

第 4 章

アサーション機能

アサーションとは、boolean 式を含む文で、その文が実行されるときに true であることをプログラマが検証するために使用します。たとえば、データバッファのすべての引数を非整列化した後、プログラマはバッファに残っているデータのバイト数がゼロであると表明します。アサーションを実行するとき、システムは boolean 式を評価して、偽であると評価した場合にエラーを報告します。boolean 式が真であることを確認することによって、システムはプログラムが正しいことを保証し、これによってエラーのないプログラムになる可能性が高くなります。

性能を優先にするときは、アサーションチェックを無効にします。通常、プログラムを開発およびテストするときにはアサーションチェックを有効にしておき、配備するときに無効にします。

アサーションは無効になっている可能性もあるので、プログラムは「アサーションに含まれている boolean 式が評価される」と仮定すべきではありません。したがって、この式により副作用が生じないようにすべきです。つまり、この式を評価することによって何らかの状態に影響を与える (そして、評価の完了後に判明する) ようなことがあってはなりません。アサーションに含まれる boolean 式が副作用を持つことは不正ではありませんが、アサーションが有効または無効であるかどうかによってプログラムの動作が変わる可能性があるので一般的には適切ではありません。

同様に、アサーションは public メソッドの引数チェックに使用しないでください。通常、引数チェックはメソッドの規約の一部であり、アサーションが有効または無効であるかどうかにかかわらず、この規約は遵守される必要があります。アサーションを引数チェックに使用する場合のもう一つの問題は、引数に問題がある場合に適切なランタイム例外 (`IllegalArgumentException`、 `IndexOutOfBoundsException`、または `NullPointerException`) をスローする必要があるということです。しかし、アサーションが失敗すると適切な例外がスローされません。

詳細については、次の項目を参照してください。

コンパイル

javac バイトコードコンパイラがアサーションを含むコードを受け入れるには、次の例のように、-source 1.4 コマンド行オプションを使用する必要があります。

```
javac -source 1.4 MyClass.java
```

構文

新しいキーワードが Java プログラミング言語に追加されました。assert キーワードは、次のように文法が修正された製品と新しい製品で使用できます。

```
StatementWithoutTrailingSubstatement:
           <All current possibilities, as per JLS,
              Section 14.4> AssertStatement
AssertStatement:
   assert Expression1;
   assert Expression1 : Expression2;
```

どちらの形式の assert 文でも、$Expression_1$ は boolean 型である必要があり、そうでない場合はコンパイル時にエラーが発生します。

セマンティクス

あるクラスでアサーションが無効である場合、そのクラスに含まれる assert 文は何にも影響しません。アサーションが有効である場合、最初の式が評価されます。この式が false であると評価された場合、AssertionError がスローされます。さらに、(最初の式の後にコロンをはさんで) 2 番目の式が存在する場合、この式も評価されて、AssertionError のコンストラクタに渡されます。そうでない場合、パラメータのないコンストラクタが使用されます。最初の式が true であると評価された場合、2 番目の式は評価されません。

どちらかの式の評価中に例外がスローされた場合、assert 文は途中で終了して、当該の例外をスローします。

アサーションの有効化および無効化

デフォルトでは、アサーションは無効になっています。ここで説明する 2 つのコマンド行スイッチを使用すると、アサーションを有効または無効に設定できます。

次のスイッチは、アサーションを様々なレベルで有効にします。

```
java [ -enableassertions | -ea  ] [:<package name>"..." | :<class name> ]
```

引数を指定しない場合、アサーションはデフォルトで有効になります。「...」で終了する引数を 1 つ指定した場合、指定されたパッケージおよびそのサブパッケージにおいて、アサーションはデフォルトで有効になります。引数が「...」だけである場合、現在の作業用ディレクトリにある名前のないパッケージにおいて、アサーションは有効になります。「...」で終了しない引数を 1 つ指定した場合、指定されたクラスにおいて、アサーションは有効になります。

同様に、次のスイッチはアサーションを無効にします。

```
java [ -disableassertions | -da ] [:<package name>"..." | :<class name> ]
```

これらのスイッチの複数のインスタンスを単一のコマンド行に指定した場合、最初にこれらのスイッチが指定された順番どおりに処理されて、その後で任意のクラスがロードされます。たとえば、プログラムを実行するときに、パッケージ `com.wombat.fruitbat` (およびそのサブパッケージ) 内だけでアサーションを有効にするには、次のコマンドを使用します。

```
java -ea:com.wombat.fruitbat...  <Main class>
```

プログラムを実行するときに、パッケージ `com.wombat.fruitbat` ではアサーションを有効にするが、クラス `com.wombat.fruitbat.Brickbat` ではアサーションを無効にするには、次のコマンドを使用します。

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat <class>
```

上記スイッチはすべてのクラスローダおよび (クラスローダを持たない) システムクラスに適用されます。この規則には例外が 1 つあります。引数を指定しない場合、このスイッチはシステムクラスには適用されません。 これによって、システムクラスを除くすべてのクラスでアサーションを有効にすることが簡単になります。すべてのシステムクラスでアサーションを有効にする (つまり、システムクラスのデフォルトのアサーション状態を *true* に設定する) には、次のスイッチを使用します。

```
 java [ -enablesystemassertions | -esa ]
```

逆に、すべてのシステムクラスでアサーションを無効にするには、次のスイッチを使用します。

```
java [ -disablesystemassertions | -dsa ]
```

プログラムによるアサーションの有効化および無効化

ほとんどのプログラマは次の方法を使用する必要はありません。次の方法は、インタプリタまたは他の実行環境を作成するために提供されています。

クラスローダに対するデフォルトのアサーション状態の設定

各クラスローダは、この後でクラスローダが初期化する新しいクラスにおいて、アサーションをデフォルトで有効または無効にするかを決定する「デフォルトのアサーション状態 (boolean 値)」を持っています。デフォルトでは、新たに作成されるクラスローダのアサーション状態は false (無効) です。次のように新しいメソッドをクラス ClassLoader で呼び出すことによって、この状態はいつでも変更できます。

```
public void setDefaultAssertionStatus(boolean enabled)
```

クラスがロードされるとき、そのクラスローダにクラスのパッケージ名またはそのクラス名のアサーション状態に関する特別な命令が (後述する ClassLoader の 2 つの新しいメソッドのどちらかにより) 指定されている場合、このような命令はクラスローダのデフォルトのアサーション状態よりも優先されます。そうでない場合、クラスのアサーションはそのクラスローダのデフォルトのアサーション状態に指定されているとおりに有効または無効になります。

パッケージおよびそのサブパッケージに対するアサーション状態の設定

次の方法を使用すると、パッケージごとにデフォルトのアサーション状態を設定できます。パッケージごとにデフォルトのアサーション状態を設定する場合、実際には、パッケージおよびそのサブパッケージに適用されることに注意してください。

```
public void setPackageAssertionStatus(String packageName, boolean enabled);
```

クラスおよびその入れ子クラスに対するアサーション状態の設定

次の方法を使用すると、クラスごとにアサーション状態を設定できます。

```
public void setClassAssertionStatus(string className, boolean
enabled);
```

クラスローダのデフォルトのアサーション状態へのリセット

次の方法を使用すると、クラスローダに関連するすべてのアサーション状態の設定をクリアできます。

```
public void clearAssertStatus();
```

使用上の注意

この節では、アサーションの仕様ではなく、アサーションを使用するときの注意事項について説明します。標準化コミュニティから見れば、この節で説明することは標準ではありません。

この節では、アサーション要素について適切な使用法と不適切な使用法の例を示します。これらの例はすべてを網羅しているわけではなく、アサーションの本来の使用法を理解してもらうことを目的としています。

内部の不変条件

一般的には、プログラムの動作に関する重要な前提を示す短いアサーションを頻繁に使用することが適切です。

アサーション機能が存在しなければ、多くのプログラマは次のようなコメントを使用します。

```
 if (i%3 == 0) {
        ...
    } else if (i%3 == 1) {
```

```
        ...
    } else { // (i%3 == 2)
        ...
    }
```

このように不変条件を表明している部分がコードに存在する場合、assert に変更するべきです。上記例の場合、assert が `if-else` 文の `else` 節を保護するには次のように変更します。

```
if (i % 3 == 0) {
        ...
    } else if (i%3 == 1) {
        ...
    } else {
        assert i%3 == 2;
        ...
    }
```

`i` が負の場合、`%` 演算子は真の mod 演算子ではないため上記例のアサーションは失敗しますが、残りの計算は続けられ、その結果は負になります。

制御フローの不変条件

アサーションを効果的に使用できるもう一つの例は、デフォルトの case を持たない `switch` 文です。

たとえば:

```
 switch(suit) {
      case Suit.CLUBS:
        ...
        break;

      case Suit.DIAMONDS:
        ...
        break;

      case Suit.HEARTS:
        ...
        break;

      case Suit.SPADES:
        ...
    }
```

プログラマはおそらく、「上記 `switch` 文の 4 つの `case` のうちの 1 つが常に実行される」と仮定しています。この仮定をテストするには、次のようなデフォルトの `case` を追加します。

```
default:
        assert false;
```

通常は、プログラマが「本来到達すべき場所ではない」と仮定している場所ならば、次の文はどこにでも置けるはずです。

```
assert false;
```

たとえば、次のようなメソッドがあると仮定します。

```
void foo() {
       for (...) {
           if (...)
               return;
        }
        // Execution should never reach this point!!!
   }
```

最後のコメントを次のように変更します。

```
assert false;
```

この技法を使用するときには十分に注意してください。ある文が到達不能である場合 (JLS 14.19 を参照)、その文が到達不能であることを表明しようとすると、コンパイル時にエラーが発生します。

事前条件、事後条件、およびクラスの不変条件

assert 構文は、「規約による設計」を完全に適用していない機能ですが、プログラミングにおける非公式の「規約による設計」スタイルをサポートするのに役立ちます。

事前条件

規約上、公開メソッドの事前条件はメソッド内の明示的なチェックによって実施され、結果として、指定された特別な例外が発生します。次に例を示します。

```
/**
     * Sets the refresh rate.
     *
     * @param  rate refresh rate, in frames per second.
     * @throws IllegalArgumentException if rate <= 0 or
     *           rate> MAX_REFRESH_RATE.
     */
     public void setRefreshRate(int rate) {
         // Enforce specified precondition in public method
         if (rate <= 0 || rate> MAX_REFRESH_RATE)
             throw new IllegalArgumentException("Illegal rate: " + rate);

         setRefreshInterval(1000/rate);
     }
```

assert 構文を追加しても、この規約には影響ありません。アサーションが有効であるかどうかにかかわらず、アサーションが入っているメソッドが必ず引数チェックを実施するような事前条件に対しては、アサーションは不適切です。さらに、assert 構文は指定されたタイプの例外をスローしません。

しかし、private メソッドには事前条件があり、かつ、クラスの作成者が「クライアントがそのクラスで何をしてもその事前条件は保たれる」と考えている場合には、アサーションは適切です。次に例を示します。

```
/**
  * Sets the refresh interval (must correspond to a legal frame rate).
  *
  * @param  interval refresh interval in milliseconds.
  */
  private void setRefreshInterval(int interval) {
      // Confirm adherence to precondition in nonpublic method
      assert interval> 0 && interval <= 1000/MAX_REFRESH_RATE;

      ... // Set the refresh interval
  }
```

MAX_REFRESH_RATE が 1000 よりも大きく、ユーザが選択したリフレッシュレートが 1000 よりも大きい場合、上記アサーションは失敗するので注意してください。 つまり、これはライブラリのバグを示しています。

事後条件

public メソッドに指定されているかどうかにかかわらず、事後条件チェックはアサーションで実装するのが最適です。次に例を示します。

```
/**
    * Returns a BigInteger whose value is (this-1 mod m).
    *
    * @param  m the modulus.
    * @return this-1 mod m.
    * @throws ArithmeticException  m <= 0, or this BigInteger
    *         has no multiplicative inverse mod m (that is, this BigInteger
    *         is not relatively prime to m).
    */
   public BigInteger modInverse(BigInteger m) {
       if (m.signum <= 0)
         throw new ArithmeticException("Modulus not positive: " + m);
       if (!this.gcd(m).equals(ONE))
         throw new ArithmeticException(this + " not invertible mod " + m);

       ... // Do the computation

       assert this.multiply(result).mod(m).equals(ONE);
       return result;
   }
```

実際には、2 番目の事前条件(this.gcd(m).equals(ONE)) は冗長であるので、計算を実行する前にチェックしません。この事前条件のチェックは、標準アルゴリズムによるモジュラ乗法逆数計算の副作用として行われます。

場合によっては、計算を実行する前にいくつかのデータを保存しておき、計算が完了した後に事後条件をチェックする必要があります。このような事後条件のチェックを行うには、2 つの assert 文と、計算の後にチェック (または再チェック) できるように 1 つまたは複数の変数の状態を保存するように設計された単純な内部クラスを使用します。たとえば、次のようなコードがあると仮定します。

```
void foo(int[] array) {
        // Manipulate array
        ...

        // At this point, array will contain exactly the ints that it did
        // prior to manipulation, in the same order.
    }
```

次に、上記メソッドを変更して、形だけのアサーションから機能するアサーションに変更する方法を示します。

```
 void foo(final int[] array) {

        class DataCopy {
          private int[] arrayCopy;

          DataCopy() { arrayCopy = (int[])(array.clone()); }

          boolean isConsistent() { return Arrays.equals(array, arrayCopy); }
        }

        DataCopy copy = null;

        // Always succeeds; has side effect of saving a copy of array
        assert (copy = new DataCopy()) != null;

        ... // Manipulate array

        assert copy.isConsistent();
     }
```

このメソッドを簡単に説明すると、複数のデータフィールドを保存して、計算前後の値に関連する複雑なアサーションを任意にテストします。

最初の assert 文 (副作用として単独で実行される) をよりわかりやすくすると、次のようになります。

```
  copy = new DataCopy();
```

しかし、アサーションが有効であるかどうかにかかわらず、この文は配列をコピーするので、「無効である場合、アサーションは何にも影響してはならない」という規則に違反します。

クラスの不変条件

すでに述べたとおり、アサーションは内部不変条件をチェックするのに適切です。アサーション機構自体は表明を行うために特別なスタイルを要求しません。ときには、必要な制約をチェックする数多くの式を単一の内部メソッドに結合して、そのメソッドをアサーションで呼び出すようにする方が便利な場合もあります。たとえば、何かのバランスツリーのデータ構造を実装しようとしていると仮定します。この場合はおそらく、ツリーが実際に (データ構造が示すとおりに) 効率的に構築されているかどうかをチェックする private メソッドを実装する方が適切です。

```
// Returns true if this tree is properly balanced
   private boolean balanced() {
       ...
   }
```

このメソッドはクラス不変条件です。どのメソッドにおいても、クラス不変条件は常に (メソッドが完了する前でも後でも) 真である必要があります。これをチェックするには、次のようにアサーションでチェックします。

```
assert balanced();
```

各 public メソッドとコンストラクタの直前に、assert 行を置きます。データ構造がネイティブメソッドによって実装されている場合を除いて、一般的に、各 public メソッドの先頭に同様なチェックを置く必要はありません。この場合、メソッドの呼び出し間に、メモリー破壊のバグが「ネイティブピア」のデータ構造を破壊する可能性があります。このようなメソッドの先頭にあるアサーションが失敗した場合、このようなメモリー破壊が発生したことを意味します。同様に、ほかのクラスによって状態が変更される可能性があるクラスにおいては、クラス不変条件のチェックをメソッドの先頭に置くことが望まれます。しかし、クラスの状態はほかのクラスから直接見ることができないように設計することが推奨されています。

クラスファイルからのアサーションのすべてのトレースの削除

リソースが制限されているデバイスを開発しているプログラマは、クラスファイルからアサーションを完全に取り去ってしまいたいと思うかもしれません。こうすることによって、フィールドではアサーションを有効にできなくなりますが、クラスファイルのサイズを減らすことができるので、おそらく、クラスをロードする速度を上げることができます。高品質の JIT が存在しない場合、アサーションのトレースを減らし、ランタイム性能を上げることができます。

アサーション機能は、クラスファイルからアサーションのトレースを削除する機能を直接的にはサポートしていません。しかし、次のように「条件付きコンパイル」(JLS 14.19 を参照) で assert 文を使用することができます。

```
static final boolean asserts = ... ; // false to eliminate asserts

    if (asserts)
        assert <expr> ;
```

この方法でアサーションを使用した場合、コンパイラは自由に、自分が生成するクラスファイルからアサーションのトレースをすべて削除できます。リソースが制限されているデバイスのコードを生成するときには、この方法をできるだけ使用することが推奨されます。

アサーションを有効にするための要件

重要なシステムを扱うプログラマにとっては、フィールドでアサーションが無効にならないほうが望ましいかもしれません。次に、アサーションが無効になっている場合、そのクラスをロードしないようにする例を示します。

```
static {
      boolean assertsEnabled = false;
      assert assertsEnabled = true; // Intentional side effect!!!
      if (!assertsEnabled)
          throw new RuntimeException("Asserts must be enabled!!!");
  }
```

ソース互換性

Java プログラミング言語には `assert` キーワードが追加されたので、`assert` を識別子として使用している既存のプログラムは無効になります。しかし、このキーワードが追加されたからと言って、既存のバイナリ (`.class` ファイル) を使用しても問題は発生しません。`assert` が正当な識別子であるプログラムから正当な識別子でないプログラムへの移行を簡単にするために、このリリースではコンパイラは 2 つの動作モードをサポートしています。

- 通常動作モードでは、コンパイラは以前のリリース (J2SE 1.3) の仕様に準拠するプログラムを受け入れます。アサーションは許可されないので、`assert` キーワードが識別子またはラベルとして使用された場合、コンパイラは警告を生成します。
- 代替動作モードでは、コンパイラは J2SE 1.4 の仕様に準拠するプログラムを受け入れます。アサーションが許可されるので、`assert` キーワードが識別子またはラベルとして使用された場合、コンパイラはエラーメッセージを生成します。

アサーションを有効にするには、次のコマンド行スイッチを使用します。

```
-source 1.4
```

このフラグを指定しない場合、ソース互換性を最大限にするために、デフォルトの動作は「1.3」です。1.3 とのソース互換性はいずれサポートされなくなる予定です。

設計に関する FAQ

次に、アサーション機能の設計に関するさまざまな FAQ を示します。

- 一般的な質問
- 互換性
- 構文およびセマンティクス
- AssertionError クラス
- アサーションの有効化と無効化

設計に関する FAQ - 一般的な質問

1. 特別なサポートなしで *Java* プログラミング言語上にアサーションをプログラムできるのにアサーション機能を提供するのはなぜでしょうか。

   アドホックな実装も可能ですが、アサーション機能は必然的に見た目が悪く(アサーションごとに `if` 文が必要)、非効率的です (アサーションが無効の場合でも状態を評価する)。さらに、アドホックな実装はそれぞれ独自のアサーションを有効または無効にする手段を持つことになるので、特に、フィールドでのデバッグにおいてこのような実装の利点を失ってしまいます。これらの欠点のために、アサーションは Java の機能として取り入れられませんでした。アサーションのサポートがプラットフォームに追加されれば、この状況は改善されるでしょう。

2. アサーションの実装にライブラリではなく言語の変更が採用されたのはなぜでしょうか。

   言語の変更は簡単ではなく、たいへんな努力が必要です。ライブラリによるアプローチも考えました。しかし、アサーションを無効にした場合の実行時コストがごくわずかになることが重要であると判断しました。これをライブラリで行おうとすると、プログラマは各アサーションを `if` 文としてハードコードする必要があります。ほとんどのプログラマはこのような作業を行いたくはありません。その場合 `if` 文を省略して性能を落とすか、またはアサーション機能を完全に無視します。実は、James Gosling による Java の最初の仕様にはアサーション機能が含まれていました。しかし、時間の制約のために十分な設計および実装ができなかったため、Oak 仕様から削除されました。

3. *Eiffel* プログラミング言語のように、事前条件、事後条件、およびクラス不変条件に、「規約による設計」機能を本格的に提供しなかったのはなぜでしょうか。

   このような機能を提供することも考えましたが、Java プラットフォームライブラリを大幅に変更せずに、しかも、古いライブラリと新しいライブラリ間の不整合を最小限に抑えながら、このような機能を Java プログラミング言語に実装できるかどうかに確信が持てませんでした。さらに、このような機能が Java の最大の特徴である簡易性を損なわないかどうかにも確信が持てませんでした。すべてを考慮して、単純な boolean 型のアサーション機能がもっとも簡単なソリューションであり、もっともリスクが少ないと結論付けました。ただし、boolean 型のアサーション機能を言語に追加したと言っても、将来、「規約による設計」機能を本格的に追

加する可能性を否定するわけではありません。

単純なアサーション機能は、制限はあるものの、「規約による設計」スタイルのプログラミングを実現します。assert 文は事後条件とクラス不変条件のチェックに適切です。事前条件のチェックは依然、指定された特別な例外 (`IllegalArgumentException` や `IllegalStateException` など) を生成するメソッド内で行う必要があります。

4. *boolean* 型のアサーションとは別に、アサーションが無効になっている場合にコード全体の実行を抑制するような (アサーションに似た) 機能を提供しなかったのはなぜでしょうか。

   このような機能を提供すると、プログラマは、別のメソッドを使用した方がいい場合でも、複雑なアサーションをインライン化してしまう可能性があるためです。

設計に関する FAQ - 互換性

1. 新しいキーワードによって、*assert* を識別子として使用している既存のプログラムに互換性の問題は発生しませんか。

   ソースファイルの場合は発生します。しかし、`assert` を識別子として使用しているクラスのバイナリはそのまま動作します。移行を簡単にするために、どのようにすれば開発者は移行期間中に `assert` を識別子として使用し続けることができるかを説明します。53 ページの「ソース互換性」を参照してください。

設計に関する FAQ - 構文およびセマンティクス

1. *Expression* $_2$ のプリミティブ型を使用できるのはなぜでしょうか。

   このタイプの式を制限する理由はありません。任意のタイプを許可すると、たとえば、アサーションごとに一意の整数コードを関連付けたい開発者には便利になります。さらに、この式をさらに望ましい `System.out.print(...)`, という形にすることが可能になります。

設計に関する FAQ - AssertionError クラス

1. *Expression*$_2$ が存在しない *assert* 文によって *AssertionError* が生成される場合、表明された状態のプログラムテキストが詳細なメッセージとして使用されない (たとえば、「*height* < *maxHeight*」) のはなぜでしょうか。

   こうすることによって、アサーションの「既成概念にとらわれない」便利さを向上させる場合もありますが、このような文字列定数すべてを `.class` ファイルと実行時イメージに追加するということで発生するコストに見合うほどの利点ではありません。

2. *AssertionError* が発生すると、このエラーを生成したオブジェクトにアクセスできなくなるのはなぜでしょうか。同様に、詳細なメッセージの代わりに、任意のオブジェクトをアサーションから *AssertionError* に渡さないのはなぜでしょうか。

このようなオブジェクトへのアクセスを許可すると、プログラマがアサーションの失敗からの復帰を試みようとするため、アサーション機能の目的から逸脱します。

3. コンテキストアクセス用メソッド (*getFile*、*getLine*、*getMethod* など) を *AssertionError* で提供しないのはなぜでしょうか。

   この機能は Throwable で提供するのが最良であるので、AssertionError だけではなく、すべての Throwable で使用できます。アサーション機能が最初に登場するリリースまでには、Throwable を拡張して、この機能を提供できるようにする予定です。

4. *AssertionError* が *RuntimeException* ではなく *Error* のサブクラスなのはなぜでしょうか。

   この問題は大いに議論されました。専門家グループが長時間議論した結果、プログラマがアサーションの失敗を復帰しないようにするには、Error がより適切であるという結論に達しました。一般的には、アサーションが失敗した原因を突き止めることは困難または不可能です。このような失敗は、プログラムが「未知の領域」で動作しており、実行を継続しようとすると危険であることを示しています。さらに、規約によると、メソッドはスローする可能性があるほとんどのランタイム例外を (「@throws」doc コメントにより) 指定することになっています。アサーションの失敗を生成するような条件をメソッドで指定するのは意味がありません。このような情報は実装の詳細である (つまり、実装やリリースごとに変更できる) と考えることができます。

設計 FAQ - アサーションの有効化および無効化

1. オブジェクトファイルからアサーションを完全に削除するコンパイラフラグを提供しないのはなぜでしょうか。

   「フィールドでアサーションを有効にできるようにする」という強い要求があります。開発者がコンパイル時にオブジェクトファイルからアサーションを削除できるようにするという可能性もありました。しかし、アサーションには本来はあってはいけない副作用が生じることもあるため、このようなフラグはプログラムの動作を大幅に変えてしまう可能性があります。有効な Java プログラムごとにセマンティクスが 1 つだけ関連付けられていることが理想的です。また、オブジェクトファイルにアサーションを残しておけばフィールドで有効にできるので、ユーザにはこちらの方が推奨されます。最後に、標準 Java の「条件付きコンパイル」(JLS 14.19 を参照) を使用すると、必要に応じてこの結果を実現することができます。

2. *setPackageAssertionStatus* のセマンティクスが、単純なパッケージ型でなく、パッケージツリー型なのはなぜでしょうか。

   実際には、プログラマはパッケージ階層を使用して自分たちのコードを編成しているので、階層型の制御の方が便利です。たとえば、パッケージツリー型のセマンティクスを使用すると、一度にすべての Swing でアサーションを有効または無効にできます。

3. 呼び出されたときにはすでにアサーション状態を設定するには遅すぎた場合 (つまり、名前付きクラスがすでにロードされている場合)、*setClassAssertionStatus* が例外をスローするのではなく、*boolean* 値を戻すのはなぜでしょうか。

アサーション状態を設定するには遅すぎた場合、警告メッセージなど以外は対処の必要がないか、対処しないでください。 例外のスローは適切ではありません。

4. *setDefaultAssertionStatus* と *setAssertionStatus* の代わりに単一メソッドをオーバーロードしないのはなぜでしょうか。

   メソッドの名前付けにおいては、わかりやすさを優先するためです。

5. アプレットがアサーションを有効または無効にすることを防ぐための *RuntimePermission* が存在しないのはなぜでしょうか。

   アプレットは任意の ClassLoader メソッドを呼び出してアサーション状態を変更する必要はありませんが、これを許可するとマイナスの状況が生じる可能性があります。まだロードされていないクラスのアサーションを有効にすると、最悪の場合、アプレットは弱い DoS 攻撃を受ける可能性があります。さらに、アプレットがアサーション状態を変更できるのは、アプレットがアクセスできるクラスローダによってロードされる予定のクラスだけです。すでに、信頼できないコードがクラスローダへのアクセス権を取得できないようにするための RuntimePermission は存在します (getClassLoader)。

6. 包含クラスのアサーション状態を照会するための構文を提供しないのはなぜでしょうか。

   このような構文を提供すると、プログラマは複雑なアサーションコードをインラインにしようとし、これは望ましくありません。

```
if (assertsEnabled()) {
    ...
}
```

   さらに、必要であれば、現在の API の上のアサーションの状態を照会することは簡単です。

```
boolean assertsEnabled = false;
         assert assertsEnabled = true;  // Intentional side-effect!!!
         // Now assertsEnabled is set to the correct value
```