Adaptive Server Enterprise における Java

Adaptive Server® Enterprise

15.7

ドキュメント ID：DC31656-01-1570-01
改訂：2011 年 9 月

Copyright © 2011 by Sybase, Inc. All rights reserved.

このマニュアルは Sybase ソフトウェアの付属マニュアルであり、新しいマニュアルまたはテクニカル・ノートで特に示されないかぎりは、後続のリリースにも付属します。このマニュアルの内容は予告なしに変更されることがあります。このマニュアルに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、無断で使用することはできません。

このマニュアルの内容を弊社の書面による事前許可を得ずに、電子的、機械的、手作業、光学的、またはその他のいかなる手段によっても、複製、転載、翻訳することを禁じます。

ybase の商標は、Sybase trademarks ページ (http://www.sybase.com/detail?id=1011207) で確認できます。Sybase およびこのリストに掲載されている商標は、米国法人 Sybase, Inc. の商標です。® は、米国における登録商標であることを示します。

このマニュアルに記載されている SAP、その他の SAP 製品、サービス、および関連するロゴは、ドイツおよびその他の国における SAP AG の商標または登録商標です。

Java および Java 関連の商標は、米国およびその他の国における Sun Microsystems, Inc. の商標または登録商標です。

Unicode と Unicode のロゴは、Unicode, Inc. の登録商標です。

IBM および Tivoli は、International Business Machines Corporation の米国およびその他の国における登録商標です。

このマニュアルに記載されている上記以外の社名および製品名は、当該各社の商標または登録商標の場合があります。

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

第 1 章　データベースにおける Java の概要 1
データベースにおける Java の利点 1
データベースにおける Java の機能 2
データベース内の Java メソッドの呼び出し 2
Java クラスをデータ型として格納 3
データベースへの XML の格納とデータベース内の XML へのクエリ 3
Java コンポーネント 3
Adaptive Server 15.0.3 およびそれ以降の機能変更 4
クラスの配布方式の変更 4
ヘッドレス・モードでの PCA/JVM の実行 5
メモリ管理における変更 5
ClassLoader の動作の変更 5
規格 6
データベースにおける Java に関する Q/A 6
主な機能 6
データベースに Java 命令を格納する方法 7
Java をデータベース内で実行する方法 7
サポートされている Java 仮想マシン (JVM) 7
ヘッドレス・モードについて 7
JDBC について 7
Java と SQL を一緒に使用する方法 8
Java API とは 8
Java API でサポートされる Java クラス 8
データベースへのユーザ定義クラスのインストール 8
Java を使用したデータへのアクセス 9
クライアントとサーバでの同じクラスの使用 9
SQL で Java クラスを使用する方法 9
データベースでの Java についての詳細 10
データベースで Java を使用してできないこと 10

第 2 章 **Java 環境の管理 11**

Java 環境のコンポーネント 11
- JVM プラグ可能コンポーネント 12
- Pluggable Component Adapter JVM (PCA/JVM) 13
- Pluggable Component Interface (PCI) および PCI Bridge 14
- PCI メモリ・プール 14
- sybpcidb データベース 15
- sybpcidb で設定値を構成する方法 16

設定値を変更する状況 17
- サーバ・レベルのオプション 17
- PCI Bridge の設定オプション 17
- PCA/JVM の設定オプション 17

実行中のサーバでの設定値の変更 18
- Adaptive Server の再起動による設定値の変更 19
- JVM の初期化前の設定値の変更 19
- JVM の初期化後の設定値の変更 20

sybpcidb へのデフォルト設定値の復元 20

モニタリング・テーブルを使用した PCI Bridge に関する情報の表示 21

第 3 章 **データベースでの Java の準備と管理 23**

Java Runtime Environment 23
- データベース内の Java クラス 23
- JDBC ドライバ 24
- JVM 24

Java の有効化 25

データベースへの Java クラスのインストール 25
- installjava の使用 25
- 他の Java-SQL クラスの参照 28

インストールされたクラスと JAR についての情報の表示 28

インストールされたクラスと JAR のダウンロード 29

クラスと JAR の削除 29
- クラスの保持 30

第 4 章 **SQL での Java クラスの使用 31**

一般考慮事項 31
- Java について 32
- Java-SQL の名前 32

データ型としての Java クラスの使用 33
- Java-SQL カラム付きのテーブルの作成と変更 33
- Java オブジェクトの選択、挿入、更新、削除 35

SQL での Java メソッドの呼び出し 37
- サンプル・メソッド 38
- Java-SQL メソッドの例外 38

Java インスタンスの表現 39

Java-SQL データ項目の割り当てのプロパティ 40

Java フィールドと SQL フィールドの間でのデータ型のマッピング 42
データと識別子の文字セット 43
Java-SQL データの部分型.. 43
範囲を広げる変換 43
範囲を狭める変換 44
ランタイム・データ型とコンパイル時データ型.................................. 44
Java-SQL データでの null の扱い方 45
null インスタンスのフィールドとメソッドに対する参照...................... 45
Java-SQL メソッドに対する引数としての null 値.................................... 46
SQL convert 関数を使用する場合の null 値 47
Java-SQL の文字列データ 48
長さがゼロの文字列.. 49
type メソッドと void メソッド 49
Java の void インスタンス・メソッド 50
Java の void 静的メソッド 51
等号 (=) 演算子と順序演算子 52
評価順と Java メソッド呼び出し 52
カラム 53
変数とパラメータ 53
式の中の deterministic な Java 関数 54
Java-SQL クラスの静的変数 55
Adaptive Server 15.0.3 およびそれ以降での静的変数の変更 56
Cluster Edition での静的変数の変更 57
複数のデータベース内の Java クラス... 57
スコープ.. 57
データベース間での相互参照 58
クラス間転送.. 58
クラス間引数の引き渡し.. 59
テンポラリ・データベースとワーク・データベース 60
Java クラス... 60

第 5 章 JDBC を使用したデータ・アクセス 65
概要.. 65
JDBC の概念と用語 66
クライアント側 JDBC とサーバ側 JDBC の違い 66
パーミッション.. 67
JDBC を使用したデータへのアクセス 67
JDBCExamples クラスの概要 68
main( ) メソッドと serverMain( ) メソッド 68
JDBC 接続の取得：Connecter( ) メソッド.. 70
その他のメソッドへの動作の送信：doAction( ) メソッド..................... 71
命令型 SQL オペレーションの実行：doSQL( ) メソッド...................... 71
update 文の実行：updater( ) メソッド 71
select 文の実行：selecter( ) メソッド 72
SQL ストアド・プロシージャの呼び出し：caller( ) メソッド 73
ネイティブ JDBC ドライバでのエラー処理 74

JDBCExamples クラス 76
main( ) メソッド 76
serverMain( ) メソッド 77
connecter( ) メソッド 77
doAction( ) メソッド 78
doSQL( ) メソッド 79
updater( ) メソッド 79
selecter( ) メソッド 80
caller( ) メソッド 80

第 6 章 SQLJ 関数およびストアド・プロシージャ 83
概要 83
SQLJ Part 1 仕様に対する準拠 84
一般的な問題 84
セキュリティとパーミッション 84
SQLJ の例 85
Java メソッドの Adaptive Server での起動 86
Sybase Central を使用した SQLJ 関数およびプロシージャの管理 88
SQLJ ユーザ定義関数 89
引数の null 値の処理 92
SQLJ 関数名の削除 93
SQLJ ストアド・プロシージャ 94
SQL データの変更 96
入出力パラメータの使用 97
結果セットのリターン 100
SQLJ 関数およびプロシージャに関する情報の表示 104
詳細情報 104
Java と SQL データ型のマッピング 104
コマンド main メソッドの使用 108
SQLJ と Sybase の実装上の相違点 109
SQLJExamples クラス 111

第 7 章 データベースでの Java のデバッグ 115
サポートされている Java デバッガ 115
Java のデバッグの設定 116
デバッグをサポートするサーバの設定 116
リモート・デバッガの JVM デバッグ・エージェントへの付加 117

第 8 章 Java を使用したファイルおよびネットワークへのアクセス 119

java.io を使用するファイル・アクセス 119
ユーザ ID とパーミッション 119
ファイル I/O でのディレクトリの指定：UNIX プラットフォーム 121
ファイル I/O でのディレクトリの指定：Windows プラットフォーム ... 123
ファイル I/O の変更 124
既存のファイルを開くための規則 124
ファイルを開く操作でファイルを作成するための規則........................ 126
ファイルの最終確認.. 126
java.net を使用するファイル・アクセス 126
例.. 127

第 9 章 補足トピック 131

サーバにおける Java クラスの JDK 使用条件.. 131
割り当て.. 131
コンパイル時の割り当てのルール 132
実行時の割り当てのルール 132
可能な変換 133
クライアントへの Java-SQL オブジェクトの転送.. 133
パフォーマンス向上のためのアドバイス 134
SQL から JVM への呼び出し数の最小化.. 134
java.lang.Thread クラスの慎重な使用 135
PCA/JVM 内で実行されているかどうかの判断.. 136
マルチエンジン環境での SQL ループの回避.. 136
PCA/JVM でのネイティブ・メソッドへのアクセスの制御........................ 137
サポートされていない Java API パッケージ、クラス、メソッド 138
制限付き Java パッケージ、クラス、メソッド.. 138
サポートされていない java.sql メソッドとインタフェース................ 140
Java からの SQL の呼び出し 141
特別な考慮事項.. 141
Java メソッドからの Transact-SQL コマンド 142
Java と SQL の間のデータ型のマッピング 146
Java-SQL 識別子.. 147
Java-SQL のクラス名およびパッケージ名 148
Java-SQL カラムの宣言 149
Java-SQL 変数の宣言.. 149
Java-SQL カラムの参照 150
Java-SQL メンバ参照.. 151
Java-SQL メソッド呼び出し 152

用語解説.. 155

索引.. 161

第 1 章　データベースにおける Java の概要

この章では、Adaptive Server® Enterprise における Java の概要について説明します。

トピック名	ページ
データベースにおける Java の利点	1
データベースにおける Java の機能	2
Java コンポーネント	3
Adaptive Server 15.0.3 およびそれ以降の機能変更	4
規格	6
データベースにおける Java に関する Q/A	6

データベースにおける Java の利点

Adaptive Server は Java 用のランタイム環境を提供します。つまり、Java コードをサーバで実行できます。Java 用のランタイム環境をデータベース・サーバに構築することによって、データとロジックの両方を効果的に管理したり格納したりできます。

- Java プログラミング言語を Transact-SQL と統合して使用可能。
- Java コードを、クライアント、中間層、サーバなどアプリケーションの異なる層で再利用したり、どこでも必要な場所で使用することが可能。
- Adaptive Server における Java は、データベースにロジックを構築する場合、ストアド・プロシージャより強力な言語になる。
- Java クラスは充実したユーザ定義データ型。
- Java クラスのメソッドは、SQL からアクセス可能な新しい関数を提供。
- Java をデータベースで使用しても、データベースの整合性、セキュリティ、堅牢性は損われない。Java を使用しても、既存の SQL 文の動作や他の Java 以外のリレーショナル・データベースの動作は変更されない。

データベースにおける Java の機能

Adaptive Server における Java は、以下の機能を提供します。

- データベース内の Java メソッドを呼び出す。
- Java クラスをデータ型として格納する。
- データベースへの XML の格納とデータベース内の XML へのクエリ。

データベース内の Java メソッドの呼び出し

Java クラスを Adaptive Server にインストールし､それらのクラスの静的メソッドを次の 2 つの方法で呼び出すことができます。

- SQL で Java メソッドを直接呼び出す。
- メソッドを SQL 名でラップして､標準の Transact-SQL ストアド・プロシージャの場合と同様にそれらを呼び出す。

SQL での Java メソッドの直接呼び出し

オブジェクト指向言語におけるメソッドは、手続き型言語の関数に相当します。データベース内に保存されたメソッドは、参照することによって呼び出すことができます。データベースに保存されたメソッドは名前を指定することで、インスタンスのメソッドや静的メソッドを実行できます。メソッドは、Transact-SQL の select リストと where 句などで直接呼び出すことができます。

また、静的メソッドは呼び出し元に値を返すユーザ定義関数 (UDF) と同様に使用できます。

Java メソッドをこのように使用する場合、次のような制限が適用されます。

- Java メソッドで JDBC を介してデータベースにアクセスする場合、結果セットの値を使用できるのは Java メソッドのみで、クライアント・アプリケーションでは使用できない。
- 出力パラメータはサポートされない。メソッドでは JDBC 接続から受け取るデータを操作できるが、メソッドの呼び出し元に返される値は、メソッドの定義の一部として宣言された単一の戻り値だけである。

Java メソッドの SQLJ ストアド・プロシージャおよび関数としての呼び出し

Java 静的メソッドを SQL ラッパに含めると､Transact-SQL ストアド・プロシージャまたは組み込み関数とまったく同様に使用することができます｡この機能により、次のようなことが実現できます。

- Java メソッドで、呼び出し元の環境に出力パラメータと結果セットを返す。
- 従来の SQL 構文、メタデータ、パーミッション機能を利用する。
- データベース間で SQLJ 関数を呼び出す。
- 既存の Java メソッドをサーバ上、クライアント上、SQLJ に準拠するサードパーティのデータベース上の SQLJ プロシージャや関数として使用する。
- 標準仕様の Part 1 に準拠する。「規格」(6 ページ) を参照してください。

Java クラスをデータ型として格納

データベースで Java を使用すると、SQL システムに pure Java クラスをインストールして、このクラスを普通に SQL データベースのデータ型として使用できます。この機能では、よく知られたモデルや簡単で使用頻度の高い言語を使用して、SQL に完全オブジェクト指向のデータ型の拡張機能が付加されます。この機能で作成、格納したオブジェクトは、他の SQL システムやスタンドアロン Java 環境など、Java が実行可能なあらゆる環境に簡単に転送できます。

データベースで Java クラスを使用するこの機能には、互いに補完的な異なる 2 つの使い方があります。

- SQL に対してデータ型拡張メカニズムを提供する。このメカニズムは、SQL で作成／処理されたデータに使用できます。
- Java 用に永続的なデータ機能を提供する。主に Java で作成／処理されたデータを SQL で格納するのに使用できます。Adaptive Server における Java の優位性は、従来の SQL の機能に比較すると歴然です。Java オブジェクトをスカラ SQL データ型にマップしたり、型のないバイナリ文字列として保存したりする必要がありません。

データベースへの XML の格納とデータベース内の XML へのクエリ

ハイパーテキスト・マークアップ言語 (HTML) と同様、拡張マークアップ言語 (XML) では、使用するアプリケーション専用のマークアップ・タグを定義することができるため、特にデータ交換に適しています。

『Adaptive Server Enterprise における XML Services』－ データベースに XML 機能を導入する、Sybase® ネイティブの XML プロセッサと Sybase Java ベースの XML のサポートについて、また XML サービスに準拠したクエリとマッピング用の関数について説明しています。

Java コンポーネント

Adaptive Server を使用することで、市販されている既成の Java Runtime Environment (JRE) と Java 仮想マシン (JVM) のコンポーネントをプラグインできます。Java に対応するように Adaptive Server を設定すると、Java 6 以降をサポートする任意の標準 JVM を含めることができます。このインフラストラクチャによって、15.0.3 およびそれ以降のバージョンの Adaptive Server を使用して作成されたアプリケーションだけでなく、15.0.3 より前のバージョンの Adaptive Server の Java ソリューションに対して設定されている Java アプリケーションも実行できるようになります。

Adaptive Server に対する Java インタフェースには、対応する次のような市販の JVM および Sybase のコンポーネントも含まれます。

- Pluggable Component Adapter/ JVM (PCA/JVM)
- Pluggable Component Interface (PCI) および PCI Bridge (Adaptive Server 内部にある)

「第 2 章 Java 環境の管理」を参照してください。

Adaptive Server 15.0.3 およびそれ以降の機能変更

Adaptive Server バージョン 15.0.3 では、市販の JVM (Sun Java 2 Platform, Standard Edition (J2SE) など) をサポートできるようになりました。Adaptive Server バージョン 15.0.2 およびそれ以前では、内部 JVM が提供されていました。

Adaptive Server PCA/JVM によって、15.0.3 より前のバージョンで作成された Java アプリケーションも、Adaptive Server バージョン 15.0.3 およびそれ以降で作成した Java アプリケーションと共に、問題なく実行できます。

この項での説明のほかに、変更点については以下を参照してください。

- 「Adaptive Server 15.0.3 およびそれ以降での静的変数の変更」(56 ページ)
- 「Cluster Edition での静的変数の変更」(57 ページ)

クラスの配布方式の変更

Adaptive Server 15.0.2 およびそれ以前のバージョンで配布されていた Java ランタイム・クラスは、Java 1.2 リリースの限定的なサブセットでした。Adaptive Server では、ランタイム・クラスを配布しなくなりました。代わりに、JVM は市販の JRE の一部として配布されているランタイム・クラスを使用します。

一般的に、最近のバージョンの Java クラスには以前のバージョンに対する下位互換性があると考えられます。しかし、以前のバージョンで "deprecated" とマークされている一部のメソッドまたはクラスには、最近のバージョンで互換性がないものもあります。アプリケーションで使用されている "deprecated" のクラスまたはメソッドが、最近のバージョンの Java でもサポートされていることを確認してください。

Adaptive Server バージョン 15.0.2 およびそれ以前では、*$SYBASE/$SYBASE_ASE/lib* ディレクトリに *runtime.zip* ファイルが含まれています。このファイルには Adaptive Server 固有のクラス、ドライバのサポートに必要な JDBC クラス、および標準 Java クラスのサブセットが含まれています。

Adaptive Server 15.0.3 では *runtime.zip* ファイルの代わりに *sybasert.jar* (PCA/JVM に必要な Sybase Java クラスを含む) が提供され、標準 Java クラス・セットを提供するには *rt.jar* を使用します。*sybasert.jar* は *$SYBASE/ASE-15_0/lib/pca* にあり、*rt.jar* は Java の配布に含まれる *$SYBASE/shared/<jre_directory>/lib* (*jre_directory* は使用するプラットフォームの固有の名前) にあります。

ヘッドレス・モードでの PCA/JVM の実行

ユーザとの対話が必要なクラスとメソッドは、Adaptive Server 15.0.2 およびそれ以前の Java の配布には含まれませんでした。PCA/JVM では標準クラス配布を使用するため、そのようなクラスも使用できるようになりました。ユーザの対話が必要なメソッドを呼び出さないようにするには、PCA/JVM を必ずヘッドレス・モードで実行します。

メモリ管理における変更

Adaptive Server 15.0.2 およびそれ以前のバージョンは、グローバル固定ヒープ、共有クラス・ヒープ、プロセス・オブジェクト・ヒープの 3 つの独立したヒープからなるメモリ管理システムを使用しました。Adaptive Server 15.0.3 およびそれ以降のバージョンでは、単一の PCI メモリ・プールを使用します。15.0.2 およびそれ以前のバージョンによる既存の設定値は、Adaptive Server 15.0.3 では無視されます。pci memory size 設定パラメータを使用して PCI サブシステムに対する合計メモリを指定する必要があります。「第 2 章 Java 環境の管理」を参照してください。

Adaptive Server バージョン 15.0.2 およびそれ以前からの移行では、PCI メモリ・プールのデフォルト・サイズを変更する必要もあります。市販の JVM が使用するクラスおよびガーベジ・コレクション・アルゴリズムのライフ・サイクルは、Sybase の内部 JVM に含まれるものとははっきりと異なります。PCI メモリ・プールのサイズを適切に設定すると、動作に違いが見られなくなります。

ClassLoader の動作の変更

Adaptive Server バージョン 15.0.3 およびそれ以降では、ロード中のクラス検証に対する ClassLoader の動作は JVM 仕様に準拠します。

Adaptive Server バージョン 15.0.2 およびそれ以前では、クラス内に追加でロードされたクラスへの参照は確認されますが、完全には解決されません。たとえば、クラス A があるメソッド内のクラス B を参照する場合、ClassLoader はクラス B が実際に使用可能であるかどうかを確認しませんでした。したがって、あるクラスのすべての依存関係が満たされなくても正常にロードされることがありました。例外が発生するのは、満たされない依存関係が必要なメソッドが発生した場合だけでした。

すべての商用実装の JVM に対する ClassLoader は、最初にクラスがロードされるときに完全なクラスの検証を実行します。その結果、依存関係が満たされないクラスはロードされず、Unhandled Java Exception (未処理の Java 例外) が発生し、Java スタック・トレースに "java.lang.NoClassDefFoundError" というメッセージが記されます。

これは、まれなケースではあるものの、Adaptive Server 15.0.2 およびそれ以前のバージョンでは正常にロードされていたクラスが、ユーザと Java のクラスの完全なセットを提供し、すべての依存関係が満たされない場合に限り、Adaptive Server 15.0.3 およびそれ以降ではロードされないことを意味します。

規格

SQL で Java 機能を使用するための SQL 拡張機能は、ANSI SQL 規格で指定されています。Java-SQL 仕様は、SQL 標準の「Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)」に含まれています。この規格は、略式で "SQLJ" と呼ばれます。

Sybase では、Java ルーチンについては SQLJ 仕様をサポートし、Java 型については同等の機能を提供しています。Sybase では、この標準をさらに拡張しています。たとえば、Adaptive Server では SQL 内で Java メソッドやクラスを直接参照できます。

データベースにおける Java に関する Q/A

このマニュアルでは、Java に関する基本的な知識があることを前提にしていますが、データベースにおける Java については覚えなければならない事項がたくさんあります。Sybase は、Java によってデータベースの機能を拡張し、データベースによって Java の機能を拡張します。

Java に慣れていないユーザ、慣れているユーザを問わず、この項に目を通してください。この項では、Adaptive Server における Java の基本について、質問／答えの形式で学んでいきます。

主な機能

次の各項目については、このあとの項で詳しく説明します。Adaptive Server で Java を使用すると、以下のことができます。

- Java 6 およびそれ以降をサポートする市販の JVM を使用して Java を実行する。
- Java の関数 (メソッド) を SQL 文から直接呼び出す。
- Java メソッドを SQL エイリアスでラップし、それらを標準 SQL ストアド・プロシージャや組み込み関数として呼び出す。
- 内部 JDBC ドライバを使用して Java から SQL データにアクセスする。
- Java クラスを SQL データ型として使用する。
- Java クラスのインスタンスをテーブルに保存する。
- Adaptive Server データベースに格納されたロー・データを元に XML フォーマットのドキュメントを生成する。または、XML ドキュメントおよび XML ドキュメントから抽出したデータを Adaptive Server データベースに格納する。
- データベース内で実行している Java クラスをデバッグする。

データベースに Java 命令を格納する方法

Java はオブジェクト指向言語です。その命令はクラスの形式を取ります。Java 命令をデータベースの外部で記述およびコンパイルし、Java 命令を保持するバイナリ・ファイルであるコンパイル済みクラス (バイト・コード) にします。

次に、コンパイル済みクラスをデータベースにインストールし、このデータベースからクラスをデータベース・サーバで実行します。

Adaptive Server は、Java クラス用のランタイム環境を提供します。Java を記述したり、コンパイルしたりするには、PowerJ™ または Sun Microsystems Java Development Kit (JDK) などの Java 開発環境が必要です。

Java をデータベース内で実行する方法

SQL 文の実行中に Java 文があった場合、Adaptive Server はその文を実行するために JVM を呼び出します。JVM がすでに実行中だった場合は、そこに Java 呼び出しが転送されます。最初に Java を要求したときに、JVM が自動的に起動します。JVM はその Java 文で指定されているクラスを検索してロードし、そのバイト・コードを実行します。

サポートされている Java 仮想マシン (JVM)

Adaptive Server の Java フレームワークは、Java 6 およびそれ以降をサポートするすべての標準 JVM で動作するように設計されています。Adaptive Server バージョン 15.0.3 は、*$SYBASE/shared* ディレクトリに含まれている Java 6 バージョンに対して動作確認されています。それより前の Java のバージョンでコンパイルされたクラスも、以降の Java のバージョンで正常に動作します。

ヘッドレス・モードについて

Adaptive Server では Java はヘッドレス・モードで実行します。つまり、表示デバイス、キーボード、マウスは使用されません。ユーザは標準 Java 配布のすべてのクラスを使用できますが、ユーザによる入力や出力デバイスを想定する特定のメソッドはサポートされません。

JDBC について

JDBC は、Java で SQL を実行するための業界標準 API です。

Adaptive Server はネイティブの JDBC ドライバを提供しています。このドライバをサーバ上で実行するとネットワークでの通信が不要になるので、最高のパフォーマンスが発揮されます。このドライバによって、データベースにインストールされた Java クラスは、SQL 文を実行する JDBC クラスを使用できます。

Java と SQL を一緒に使用する方法

データベースでの Java を設計する主な目的は、既存の SQL 機能を自然に、しかも幅広く拡張することです。

- *Java* オペレーションを *SQL* から呼び出す － Sybase では、Java オブジェクトのフィールドやメソッドを含むことができるように SQL 式の範囲を拡大したので、Java オペレーションを SQL 文に含むことができます。
- *SQLJ* ストアド・プロシージャや関数としての *Java* メソッド － Java 静的メソッドの SQLJ エイリアスを作成し、それらを標準 SQL ストアド・プロシージャやユーザ定義の関数 (UDF) として呼び出すことができます。
- *Java* クラスはユーザ定義のデータ型となる － 従来の SQL データ型に使用されているのと同じ SQL 文を使用して Java クラスのインスタンスを格納します。

Java API の一部であるクラスと、Java 開発者が作成およびコンパイルしたクラスを使用できます。

Java API とは

Java の API (Application Programming Interface) は、Sun Microsystems によって定義された基本的な機能のセットです。Java 開発者はこれを使用し、拡張できます。これは、Java を使用して「実行できること」の中核です。

Java API は独自の権限で大量の機能を提供し、個々のユーザ・アプリケーションのために作成されるすべてのユーザ定義のクラスの基礎にもなります。

Java API でサポートされる Java クラス

Adaptive Server はデータベースでのすべての標準 Java クラスをサポートします。データベース内の Java はヘッドレス・モードで実行されるため (「ヘッドレス・モードについて」(7 ページ) を参照)、ユーザによる入力や出力デバイスを想定する特定のメソッドでは Java 例外が発生します。

データベースへのユーザ定義クラスのインストール

たとえば、ユーザが作成した Employee クラスや、開発者が設計、作成し、Java コンパイラでコンパイルした Inventory クラスを、データベースに独自の Java クラスとしてインストールできます。

ユーザ定義の Java クラスには、データとそれを操作するメソッドの両方を含むことができます。Adaptive Server では、クラスをデータベースにインストールすると、そのクラスをデータベースのすべての部分や演算で使用でき、(クラスまたはインスタンス・メソッドの形式で) それらの機能を実行できるようになります。

Java を使用したデータへのアクセス

JDBC インタフェースは、データベース・システムへのアクセス用に設計された業界標準です。JDBC クラスは、データベースへの接続、SQL 文を使用したデータの要求、クライアント・アプリケーションで処理可能な結果の返送を行うように設計されています。

Adaptive Server は内部 JDBC ドライバを提供しているため、データベースにインストールされた Java クラスでも、SQL 文を実行する JDBC クラスを使用できます。

クライアントとサーバでの同じクラスの使用

エンタープライズ・アプリケーションのさまざまなレベルで使用できる Java クラスを作成できます。また、同じ Java クラスをクライアント・アプリケーション、中間層、またはデータベースに統合できます。

異なる層で使用されるクラス、または同じ層でも長期にわたって使用されるクラスが互換性を保持するか、または意図的に非互換にされるかに注意し、アプリケーションが異なっても動作が一貫するようにします。java.io.Serializable クラスの serialVersionUID に関する Java のマニュアルを参照してください。

SQL で Java クラスを使用する方法

ユーザが定義した Java クラスを使用するには、次の手順 1 〜 3 に従います。

1. SQL データ型または静的メソッドの SQL エイリアスとして使用する Java クラス・セットを作成または入手します。
2. 作成した Java クラスを Adaptive Server データベースにインストールします。

   注意　Java の配布に含まれるクラスは常に使用可能で、使用前にデータベースにインストールする必要はありません。

3. インストールした Java クラスを SQL コードで使用します。
   - UDF として静的メソッドを直接呼び出します。
   - Java クラスを SQL のカラム、変数、パラメータのデータ型として宣言します。このマニュアルでは、これらを Java-SQL のカラム、変数、パラメータと呼びます。
   - Java-SQL のカラム、変数、パラメータのフィールドまたはメソッドを参照します。
   - 静的メソッドを SQL エイリアスでラップし、それらをストアド・プロシージャまたは関数として使用します。

データベースでの Java についての詳細

Java や、データベースにおける Java を扱った書籍は多数あります。最新の Java 言語仕様は、Sun の Web サイトにあります。

データベースで Java を使用してできないこと

Adaptive Server は、Java 開発環境ではなく、Java クラスのランタイム環境です。

データベースで実行できないアクションを次に示します。

- クラス・ソース・ファイル (*.*java* ファイル) の編集。
- Java クラス・ソース・ファイル (*.*java* ファイル) のコンパイル。
- アプレットやビジュアル・クラスなどサポートされていない Java API の実行。
- Java ネイティブ・インタフェース (JNI) の使用。
- リモート・プロシージャ・コールとの間で送受信するパラメータとしての Java オブジェクトの使用。これらは正確に変換されません。

Java-SQL 関数、SQLJ 関数、SQLJ ストアド・プロシージャによって参照されるメソッド内では、final 以外の静的変数を使用しないことをおすすめします。これらの変数に対して返される値は信頼できない可能性があります。これは、静的変数のスコープが実装に依存しているためです。

第 2 章

Java 環境の管理

トピック名	ページ
Java 環境のコンポーネント	11
設定値を変更する状況	17
実行中のサーバでの設定値の変更	18
sybpcidb へのデフォルト設定値の復元	20
モニタリング・テーブルを使用した PCI Bridge に関する情報の表示	21

J2SE など、既製の標準 Java JRE および JVM コンポーネントを Adaptive Server にプラグインできます。この章では、Java をサポートする Sybase コンポーネントについて、およびデフォルト設定値を変更する方法について説明します。

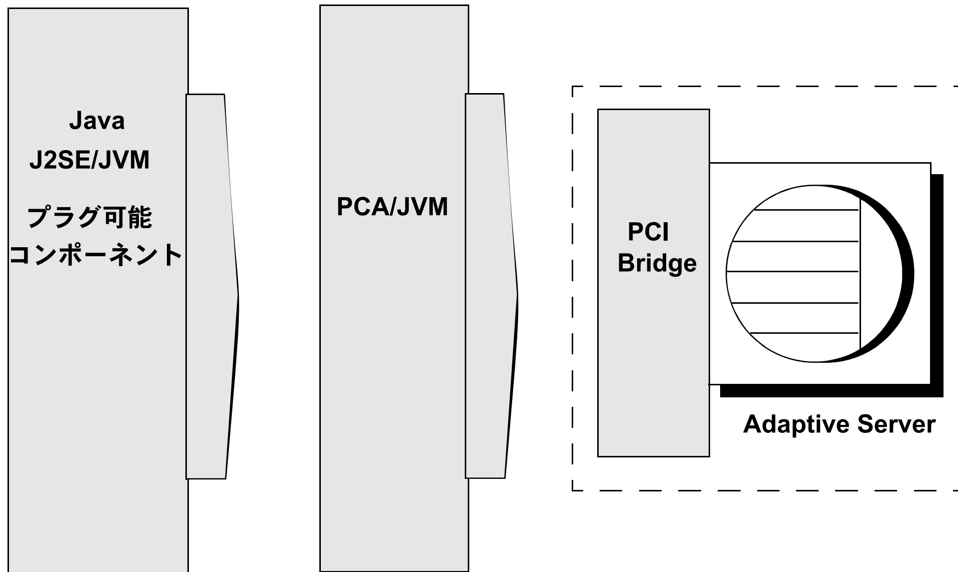
Adaptive Server の Java フレームワークは、Java 6 およびそれ以降をサポートするすべての標準 JVM で動作するように設計されています。ASE 15.5 は、*$SYBASE/shared* ディレクトリに含まれている Java 6 バージョンに対して動作確認されています。それより前の Java のバージョンでコンパイルされたクラスも、以降の Java のバージョンで正常に動作します。

JVM は、Adaptive Server から独立しています。Java の新機能が利用可能になったら、Java アプリケーションを変更またはアップグレードして新機能を活用できます。

Java 環境のコンポーネント

図 2-1 に、Adaptive Server Java 環境を構成するコンポーネントを示します。

図 2-1: Java コンポーネント



JVM プラグ可能コンポーネント

JVM プラグインは、Adaptive Server から独立したプラットフォームでエンジニアリング、サポート、およびインストールされる、動的にロードされるモジュールです。Adaptive Server に対しては、プラグインは「ブラック・ボックス」アプリケーションであり、Sybase がサポートする技術ではありません。JVM プラグインは、PCI Bridge により変換される Java 結果セットを発行し、その後変換された結果セットを Adaptive Server に送信します。

JVM プラグインは PCA/JVM によって制御されるため、Adaptive Server に間接的に接続されます。JVM プラグインは、Adaptive Server から独立してインストール、アップグレード、および起動できます。

通常、Java の配布には 1 つ以上の JVM 実装が含まれます。これにより、ユーザは個々のアプリケーションのパフォーマンス要件に最も適した VM を選択できます。

- クライアント・アプリケーション － 通常はクライアント・アプリケーションで使用されるプラットフォームでは、起動時間とメモリ消費量が減るようにチューニングされた VM が JRE に含まれています。

- サーバ・アプリケーション － すべてのプラットフォームで、プログラムの実行速度が最大になるように設計された JVM のバージョンが JRE に含まれています。

多くの Java の配布がありますが、Java テクノロジのこれらの機能はクライアントおよびサーバの両方の VM バージョンに共通です。

- Adaptive コンパイラ － Java プラグインは、標準のインタプリタを使用してアプリケーションを起動しますが、実行時にコードを分析し、パフォーマンスのボトルネック (つまり「ホット・スポット」) を検出します。
- 高速なメモリの割り付けとガーベジ・コレクション － Java テクノロジでは、オブジェクトの高速なメモリの割り付けが行われ、高速かつ効率的な最新のガーベジ・コレクタを選択できます。
- スレッド同期 － Java プログラミング言語を使用すると、プログラム実行の複数の同時パス (「スレッド」と呼ばれます) を使用できます。Java テクノロジには、大規模な共有メモリ・マルチプロセッサ・サーバで使用するために簡単に拡張できるスレッド処理機能が用意されています。

注意　子スレッドを生成するメソッドを使用するときは注意してください。Java メソッド内で起動された java.lang.Thread オブジェクトは、Adaptive Server スケジューラによってではなく実行時にスケジュールされます。これらのスレッドがプロセッサを集中利用したり、多数のスレッドを生成したりする場合、プロセッサ時間の競合によりサーバのパフォーマンスが低下することがあります。

PCA/JVM プラグインは、クライアント JVM とサーバ JVM のどちらでも使用できますが、Sybase では Java メソッドのパフォーマンスを最大限に高めるため、デフォルトでサーバ・バージョンを使用することをおすすめします。サーバのバージョンは、インストール・プロセスで使用されます。

各企業に適したクライアント・バージョンについては、クライアント・バージョンのマニュアルを参照してください。

Pluggable Component Adapter JVM (PCA/JVM)

PCA/JVM はブローカとして機能し、Adaptive Server と JVM の間のサービス要求を管理します。PCA/JVM は Adaptive Server から JVM、および JVM から Adaptive Server の両方向の要求を転送および制御します。

Pluggable Component Interface (PCI) および PCI Bridge

PCI は、Adaptive Server 内部の汎用インタフェースです。Adaptive Server をインストールまたはアップグレードすると、デフォルトでインストールされます。PCI Bridge は、PCI 内部のコンポーネントで、Adaptive Server と JVM プラグインの間の実際の処理を実行します。

PCI Bridge には、次の機能が備わっています。

- ネイティブ・スレッド (プロセス) 管理
- メモリ管理
- 同期 (ロック、条件、およびイベント) 管理
- データ・アクセス・サービスのサポート
- 設定管理
- 自動プラグイン・ロードによるオンデマンド機能ディスパッチ
- シグナルおよび例外処理
- プラットフォーム・ランタイムのサポート
- 動的なインストゥルメンテーション機能
- Adaptive Server のエラー・ログへのエラー・メッセージ・チャンネル

ほとんどのシナリオには、PCI Bridge のデフォルト設定が適しており、変更の必要はありません。変更の必要な場合、および Sybase 製品の保守契約を結んでいるサポート・センタのアドバイスがあった場合は、sp_pciconfig システム・ストアド・プロシージャを使用して、PCI 設定を変更できます。sp_pciconfig には、sybpcidb のディレクティブおよび引数のリスト、レポート、有効化、または無効化が可能なパラメータが含まれています。「実行中のサーバでの設定値の変更」(18 ページ) を参照してください。

PCI メモリ・プール

PCI メモリ・プールは、PCI Bridge の初期化時にすべて一度に割り付けられます。その後大きくなることはありません。Adaptive Server により制御され、他のメモリ・プールと同じ制約が適用されます。たとえば、1 つの割り付けが 1MB を超えることはできません。PCI メモリ・プールのデフォルト・サイズは 32,768 KB です。

Java 用にサーバを設定するときに PCI メモリ・ツールを有効にするには、enable pci 設定パラメータを使用します。詳細については、プラットフォームのインストール・ガイドを参照してください。

PCI メモリ・プールのサイズの変更

PCI メモリ・プールのデフォルト・サイズは、ほとんどのノンクラスタード・インストールに適しています。メモリ・プールのサイズを大きくするには、pci memory size 設定パラメータをリセットします。

たとえば、pci memory size を 13800 ページ (各ページは 2KB) に設定するには、次のように入力します。

```
sp_configure "pci memory size", 13800
```

pci memory size は動的な設定パラメータであり、変更内容がすぐに反映されるため、Adaptive Server を再起動する必要はありません。

Adaptive Server にメモリ・プールを割り付ける十分なメモリがない場合、この設定変更は無視され、PCI Bridge は起動しません。

pci memory size の詳細については、『システム管理ガイド 第 1 巻』を参照してください。

マルチエンジン Adaptive Server における Java VM のメモリ消費

マルチエンジン環境では、複数の Adaptive Server タスクが同時に Java VM を使用することがあります。このため、マルチエンジン環境ではシングルエンジン環境よりも多くのメモリが Java VM に必要です。したがって、実行するアプリケーションの種類と、同時に Java を実行するユーザの数に基づいて、PCI メモリ・プールのサイズを大きくする必要がある場合があります。

Adaptive Server がヒープ・サイズを計算できるようにしたり、sp_jreconfig を使用して PCA_JVM_JAVA_OPTIONS ディレクティブの -Xmx 引数および -Xms 引数を設定することでヒープ・サイズを自分で設定したりすることができます。

Adaptive Server がヒープ・サイズを設定できるようにするには、計算されるヒープ・サイズが 4MB より大きい必要があります。-Xmx 引数および -Xms 引数を設定することはできません (Adaptive Server は sybpcidb に格納された値を使用します)。

Adaptive Server がヒープ・サイズを設定すると、次のようになります。

- PCA_JVM_JAVA_OPTIONS ディレクティブの -Xmx 引数は、Java ヒープ・サイズが PCI メモリ・プール・サイズの 65% になるように設定されます。
- -Xms 引数は、-Xmx と同じ値に設定されます。
- Java ヒープ・サイズの 20% は、初期ヒープ生成用に設定されます (エデン・スペースとも呼ばれます)。

sybpcidb データベース

sybpcidb データベースには、PCI Bridge および PCA/JVM プラグインの設定情報が格納されます。Java 用にサーバを設定するときは、sybpcidb を作成して、そのテーブルをインストールし、そのシステム・ストアド・プロシージャを作成します。詳細については、プラットフォームのインストール・ガイドを参照してください。

sybpcidb システム・ストアド・プロシージャは次のとおりです。

- sp_pciconfig － PCI Bridge プロパティを設定します。
- sp_jreconfig － PCA/JVM プラグイン・プロパティを設定します。

sybpcidb テーブル

sybpcidb データベースには、これらのテーブルが格納されます。

ユーザ・テーブル	内容
pci_directives	PCI Bridge のディレクティブ設定情報。
pci_arguments	PCI Bridge の引数設定情報。
pci_slotinfo	関連するディレクティブおよび引数のテーブル名など、各スロットの情報。
pci_slot_syscalls	PCI Bridge により使用される実行時ディスパッチ・モデルの実行時システム呼び出し設定情報。
pca_jre_directives	PCA/JVM プラグインに固有のディレクティブ情報。
pca_jre_arguments	PCA/JVM プラグインに固有の引数情報。

sybpcidb の詳細については、『リファレンス・マニュアル：テーブル』を参照してください。sp_pciconfig および sp_jreconfig の詳細については、『リファレンス・マニュアル：プロシージャ』を参照してください。

sybpcidb で設定値を構成する方法

PCI Bridge および PCA/JVM の構成値は、sybpcidb に格納され、ディレクティブと引数の階層で構成されます。各ディレクティブには 1 つ以上の引数が含められ、各引数には設定値が保持されます。引数の型は次のとおりです。

- “switch” 引数 － 有効化または無効化のみが可能なプロパティを記述します。switch 引数にデータは含まれません (PCI Bridge および PCA/JVM)。
- “number” 引数 － 数値プロパティ値が含まれます (PCI Bridge および PCA/JVM)。
- “string” 引数 － 文字列プロパティ値が含まれます (PCA/JVM のみ)。
- “array” 引数 － 1 つ以上のプロパティ値のコレクションです (PCA/JVM のみ)。

各ディレクティブとその各引数を有効化または無効化できます。ディレクティブの状態は、その引数の状態より優先されます。たとえば、ディレクティブが 3 つの引数 (“arg1” が有効、“arg2” が無効、“arg3” が無効) を持つ場合を想定しましょう。

- ディレクティブが有効な場合、各引数にはそのベースの状態が保持されます。つまり、“arg1” が有効、“arg2” が無効、“arg3” が無効です。
- ディレクティブが無効な場合、“arg1”、“arg2”、および “arg3” がすべて無効になるように、ディレクトリの無効の状態が引数のベースの状態より優先されます。
- ただし、ディレクティブが再度有効になった場合、各引数はそのベースの状態 (“arg1” が有効、“arg2” が無効、“arg3” が無効) に戻ります。この配置により、1 つのコマンドですべての引数を無効にしたり、すべての引数を元の状態に戻したりすることができます。

設定値を変更する状況

ほとんどのインストールでは、サーバのデフォルト設定オプションと、PCI Bridge および PCA/JVM のデフォルト設定オプションを変更する必要はありません。いくつかの設定オプションは、安全に変更および管理できますが、ほとんどの設定オプションは Sybase 製品の保守契約を結んでいるサポート・センタの指示がない限り変更しないでください。

次の設定オプションを構成できます。

- サーバ・レベル
- PCI Bridge
- PCA/JVM

サーバ・レベルのオプション

次のサーバ・レベルの設定パラメータを変更および管理するには、sp_configure を使用します。

- enable pci － PCI Bridge を有効にします。
- enable java － データベース内の Java を有効にします。
- pci memory size － PCI メモリ・プールの最大サイズを設定します。

注意　PCA/JVM を使用するには、Java および PCI Bridge の両方を有効にする必要があります。

プラットフォームのインストール・ガイドと『システム管理ガイド 第 1 巻』を参照してください。

PCI Bridge の設定オプション

製品の保守契約を結んでいるサポート・センタから指示があった場合を除き、PCI Bridge の設定オプション (ディレクティブや引数) は変更しないでください。

PCA/JVM の設定オプション

PCA/JVM の次の引数は、安全に変更できます。

- pca_jvm_module_path － このプロパティは、インストール時に用意された JRE を使用している場合のみ変更してください。使用している場合、このプロパティが PCA/JVM で使用される JRE を指すようにします。

- pca_jvm_work_dir － 必要に応じて、特定のパーミッション・マスクを使用して設定可能な各作業ディレクトリ (信頼されたディレクトリ) のこの引数配列に 1 つのエントリを追加します。「第 8 章 Java を使用したファイルおよびネットワークへのアクセス」を参照してください。
- pca_jvm_netio － ネットワーク I/O を有効にするには、この引数を有効にします。ネットワーク I/O を無効にするには、この引数を無効にします。
- pca_jvm_dbg_agent_port － この引数を有効にし、数値を JVM がデバッグ・エージェントに使用するポート番号に設定します。Java デバッガで同じポートを受信する必要があります。
- pca_jvm_java_dbg_agent_suspend － デバッグ・エージェントを中断状態で起動するには、この引数を有効にします。この引数を有効にすると、実行プロセスに付加された後、Java デバッガでブレークポイントと他のオプションを設定できます。『リファレンス・マニュアル：コマンド』を参照してください。

注意 pca_jvm_java_dbg_agent_suspend の使用には注意が必要です。pca_jvm_java_dbg_agent_suspend を有効にすると、JVM は中断されて、デバッガを付加して JVM に続行を指示するまで、すべての Adaptive Server の Java タスクが待機します。pca_jvm_java_dbg_agent_suspend を有効にする代わりに、JVM を起動して単純な Java コマンドを実行して、デバッガを付加することをおすすめします。Java コマンドを使用すると、JVM の起動が可能になり、デバッグ対象のクラスを実行する前にデバッガを付加できるようになります。

Sybase 製品の保守契約を結んでいるサポート・センタの指示がない限り、PCA/JVM の他のどのディレクティブまたは引数も変更しないでください。

実行中のサーバでの設定値の変更

Sybase 製品の保守契約を結んでいるサポート・センタからのアドバイスによりデフォルト設定値を変更する場合は、sp_jreconfig および sp_pciconfig システム・ストアド・プロシージャを使用できます。「設定値を変更する状況」(17 ページ) を参照してください。この項では、変更した設定値を実行中のサーバのメモリにロードする方法について説明します。

Adaptive Server が起動すると、サーバが Java 用に設定されている場合は JVM が自動的にロードされます。ただし、JVM は最初の Java 要求を受け取るまで初期化されません。これは、Java の使用頻度によって異なります。初期化前に設定値を変更する方法は、比較的簡単です。初期化後に、設定情報がメモリ内のデータ構造に読み込まれてから設定値を変更するほうが難解です。

次の場合に設定情報を更新できます。

- Adaptive Server の再起動
- JVM が初期化される前 (PCA/JVM プラグインの場合のみ)
- JVM が初期化された後 (PCA/JVM プラグインの場合のみ)

Adaptive Server の再起動による設定値の変更

これは、設定情報を変更する最も簡単な方法で、いつでも実行できます。

注意　sp_pciconfig を使用して PCI Bridge の設定値を変更する場合は、この方法を使用する必要があります。

---

1. sp_jreconfig または sp_pciconfig を使用して、設定値を変更します。
2. Adaptive Server を再起動します。

JVM の初期化前の設定値の変更

Adaptive Server が実行中で JVM が初期化されていない場合、PCA/JVM プラグインの設定値を変更するには、この方法を使用します。

1. sp_jreconfig を使用して、設定値を変更します。
2. 設定パラメータをメモリにロードします。たとえば、次のように入力します。

```
sp_jreconfig "reload_config"
```

新しい設定値を有効にするために Adaptive Server を再起動する必要はありません。

注意　sp_jreconfig "reload_config" を使用して加えた変更内容は、JVM をまだ初期化していない場合にのみ有効になります。sp_jreconfig を使用すると、sybpcidb 内のテーブル値のみ変更され、Adaptive Server の起動時にメモリにロードされたメモリ内の現在のデータ構造は影響を受けません。

この方法は、JVM が初期化されているかどうかにかかわらず、安全に実行できます。JVM が初期化されている場合、reload_config コマンドは失敗し、エラー・メッセージが表示されます。悪影響はありません。

JVM の初期化後の設定値の変更

Adaptive Server が実行中であり、JVM が初期化されている場合、設定パラメータはメモリ内にあり、PCA/JVM プラグイン設定パラメータは変更できます。

JVM 設定値を変更する手順は、Adaptive Server がスレッド・モードに設定されているかプロセス・モードに設定されているかによって変わります。

- スレッド・モード：

  a　sp_jreconfig を使用して、設定値を変更します。

  b　Adaptive Server を再起動する。

  Adaptive Server が再起動した後、JVM は最初の Java 要求を受け取るまで初期化されていない状態になります。

- プロセス・モード：

  a　sp_jreconfig を使用して、設定値を変更します。

  b　JVM を実行しているエンジンをオフラインにします (この例では、エンジン番号 3)。

  ```
  sp_engine "offline", 3
  ```

  c　JVM を実行しているエンジンをオンラインに戻します。

  ```
  sp_engine "online", 3
  ```

  一連の手順を実行している間、Adaptive Server は実行され続けますが、JVM を実行しているエンジンがオンラインになるまで Java は使用できません。エンジンをオンラインにした後、JVM は最初の Java 要求を受け取るまで再び初期化されていない状態になります。

sybpcidb へのデフォルト設定値の復元

JVM の初期化後にデフォルト設定値を sybpcidb 設定値に復元する手順は、Adaptive Server を再起動できるかどうかと、単一エンジンと複数エンジンのどちらの Adaptive Server を使用しているかによって異なります。

単一エンジンの Adaptive Server を使用している場合：

1. *installpcidb* インストール・スクリプトを再インストールして、sybpcidb 設定テーブルの値を出荷時設定にリセットします。次に例を示します。

   ```
   isql -Usa -Psa_password -Sserver_name
   -i $SYBASE_ASE/scripts/installpcidb
   ```

2. Adaptive Server を再起動します。デフォルト設定値は、最初の Java 要求に反応して JVM が初期化されたときに有効になります。

複数エンジンの Adaptive Server を使用している場合：

1 *installpcidb* を再インストールして、sybpcidb 設定テーブルの値を出荷時設定にリセットします。次に例を示します。

```
isql -Usa -Psa_password -Sserver_name
-i $SYBASE_ASE/scripts/installpcidb
```

2 JVM を実行しているエンジンをオフラインにします。次に例を示します。

```
sp_engine "offline", 3
```

この例では、JVM はエンジン “3” で実行されています。

3 JVM を実行しているエンジンをオンラインに戻します。次に例を示します。

```
sp_engine "online", 3
```

新しい設定値を有効にするために Adaptive Server を再起動する必要はありません。

モニタリング・テーブルを使用した PCI Bridge に関する情報の表示

PCI Bridge に関する情報は、次の 3 つのモニタリング・テーブルを使用して表示できます。

- monPCIBridge － PCI Bridge に関する全般的な情報が表示されます。

  次に例を示します。

```
select * from monPCIBridge

Status   ConfiguredSlots  ActiveSlots  ConfiguredPCIMemoryKB  UsedPCIMemoryKB
-----------------------------------------------------------------------------
ACTIVE                 1            1                  65668             1613
```

- monPCISlots － 各スロットにバインドされたプラグインに関する情報が表示されます。次に例を示します。

```
select * from monPCISlots

Slot     Status     Modulename    Engine
-----------------------------------------
   1     IN USE        PCA/JVM         0
```

- monPCIEngine − PCI Bridge とそのプラグインのエンジン情報が表示されます。次に例を示します。

```
select * from monPCIEngine

Engine  Status  PLBStatus  NumberofActiveThreads PLBRequest  PLBWakeupRequests
------------------------------------------------------------------------------
     0 PCA ACTIVE  ACTIVE                    10          4                  4
     1 PCA ACTIVE  ACTIVE                     4          0                  0
```

詳細については、『リファレンス・マニュアル：テーブル』を参照してください。

第 3 章　データベースでの Java の準備と管理

この章では、Java Runtime Environment、サーバ上で Java を有効にする方法、データベース内の Java クラスのインストールと管理の方法について説明します。

トピック名	ページ
Java Runtime Environment	23
Java の有効化	25
データベースへの Java クラスのインストール	25
インストールされたクラスと JAR についての情報の表示	28
インストールされたクラスと JAR のダウンロード	29
クラスと JAR の削除	29

Java Runtime Environment

Adaptive Server の Java Runtime Environment には、データベース・サーバの一部として使用できるサード・パーティ JVM の Sybase PCI と Sybase ランタイム Java クラス、または Java API が必要です。クライアント上で Java アプリケーションを実行する場合、クライアント上に jConnect の Sybase JDBC ドライバも必要です。

データベース内の Java クラス

Java クラスに次のソースを使用できます。

- *rt.jar* にある標準 Java 配布と、Java インストール・ディレクトリの下の “ext” ディレクトリにインストールされたクラス。

  注意 ext ディレクトリの内容は、Java ベンダによって異なる場合があります。これらのクラスの詳細については、各ベンダのマニュアルを参照してください。

- ユーザ定義クラス

Sybase ランタイム Java クラス

Sybase ランタイム Java クラスは低レベルのクラスで、データベースで Java を有効にするためにインストールされます。このクラスは、Adaptive Server のインストール時に自動的にダウンロードされ、インストール後は *$SYBASE/$SYBASE_ASE/lib/sybasert.jar* (UNIX)、または *%SYBASE%¥%SYBASE_ASE%¥lib¥sybasert.jar* (Windows) から使用できます。Adaptive Server は、JVM の起動時に CLASSPATH 環境を設定します。

注意 オペレーティング・システム環境で CLASSPATH が設定されている場合、Adaptive Server は内部 JVM の起動時にその値を無視します。

ユーザ定義 Java クラス

installjava ユーティリティを使用して、データベースにユーザ定義クラスをインストールします。インストールしたクラスは、データベース内の他のクラスから使用でき、SQL からもユーザ定義データ型として使用できます。

JDBC ドライバ

Adaptive Server に同梱されている Sybase ネイティブの JDBC ドライバは、JDBC バージョン 1.1 と 1.2 をサポートしており、JDBC バージョン 2.0 のクラスやメソッドに準拠しています。サポートする／サポートしないクラスとメソッドのすべてのリストについては、「第 9 章 補足トピック」を参照してください。

ご使用のシステムでクライアント上に JDBC ドライバが必要な場合、JDBC バージョン 2.0 をサポートする jConnect バージョン 6.x 以降を使用してください。

JVM

Adaptive Server の Java フレームワークは、Java 6 およびそれ以降をサポートするすべての標準 JVM で動作するように設計されています。Adaptive Server バージョン 15.0.3 は、*$SYBASE/shared* ディレクトリに含まれている Java 6 バージョンに対して動作確認されています。それより前の Java のバージョンでコンパイルされたクラスも、以降の Java のバージョンで正常に動作します。

Java の有効化

注意 PCI および Java を有効にする前に、プラットフォームのインストール・ガイドで説明されているように sybpcidb を設定します。

サーバとそのデータベースの Java を有効にするには、isql からのコマンドを入力します。

```
sp_configure "enable pci", 1
sp_configure "enable java", 1
```

次に、サーバを停止し再起動します。Adaptive Server 15.0.3 以降では、Java を有効にする前提条件として PCI を有効にする必要があります。

デフォルトでは、Adaptive Server の Java は有効になっていません。サーバでの Java の使用を有効にしなければ、Java クラスをインストールしたり Java の操作を実行したりすることはできません。

データベースへの Java クラスのインストール

クライアントのオペレーティング・システムから Java クラスをインストールするには、installjava (UNIX) または instjava (Windows) ユーティリティをコマンド・ラインから使用します。

これらのユーティリティの詳細については、『ASE ユーティリティ・ガイド』を参照してください。どちらのユーティリティも実行するタスクは同じです。このマニュアルでは簡略化のために UNIX の例を使用します。

installjava の使用

installjava によって Adaptive Server システムに非圧縮 JAR ファイルがコピーされ、この JAR ファイルに格納されている Java クラスが現在のデータベースで使用できるようになります。構文は次のとおりです。

installjava
-f *file_name*
[-new | -update]
[-j *jar_name*]
[-S *server_name*]
[-U *user_name*]
[-P *password*]
[-D *database_name*]
[-I *interfaces_file*]
[-a *display_charset*]
[-J *client_charset*]
[-z *language*]
[-t *timeout*]

たとえば、*addr.jar* ファイル内のクラスをインストールするには、次を入力します。

```
installjava -f "/home/usera/jars/addr.jar"
```

-f パラメータを使用して、JAR が格納されているオペレーティング・システムのファイルを指定します。JAR には完全なパス名を使用します。

この項では、保持した JAR ファイル (-j を使用) と、インストールされた JAR とクラスの更新 (new と update を使用) について説明します。これらのオプションを含み、installjava で使用可能なオプションの詳細については、『ユーティリティ・ガイド』を参照してください。

注意 JAR ファイルのインストール時に、Application Server は対象ファイルをテンポラリ・テーブルにコピーし、次にそのファイルをインストールします。サイズの大きい JAR ファイルをインストールする場合、alter database コマンドを使用して tempdb のサイズを増やす必要があることがあります。

非圧縮 JAR のインストール

installjava ツールおよび instjava ツールには、非圧縮 jar ファイルが必要です。

Java クラスをデータベースにインストールするには、クラスまたはパッケージを非圧縮形式で JAR ファイルに保存します。Java クラスが格納されている非圧縮の JAR ファイルを作成するには、Java の jar cf0 (ゼロ) コマンドを使用します。

次の UNIX の例では、jar コマンドで非圧縮の JAR ファイルを作成します。この JAR ファイルには、jcsPackage ディレクトリ内の *.class* ファイルがすべて格納されます。

```
jar cf0 jcsPackage.jar jcsPackage/*.class
```

JAR ファイルの保持

JAR をデータベースにインストールする場合、サーバは JAR を分解してクラスを抽出し、別々に格納します。installjava で -j を指定しないかぎり、JAR はデータベースに格納されません。

-j を使用することで、installjava で指定した JAR、または JAR から抽出されたクラスを Adaptive Server システムへ保持するかを決定します。

- -j パラメータを指定した場合、Adaptive Server は JAR に格納されているクラスを通常の方法でインストールします。次に、JAR および JAR とインストールされたクラスとの対応を保持します。
- -j パラメータを指定しない場合、Adaptive Server は JAR とクラスの対応を保持しません。これは、デフォルトのオプションです。

JAR の名前を指定して、インストールしたクラスを管理しやすくすることをおすすめします。JAR ファイルを保持した場合、次を実行できます。

- remove java 文を使用して、JAR ファイルと対応するすべてのクラスを一度に削除できます。JAR ファイルを保持しなかった場合、クラスまたはクラスのパッケージを 1 つずつ削除する必要があります。
- extractjava を使用して、オペレーティング・システムのファイルに JAR をダウンロードできます。「インストールされたクラスと JAR のダウンロード」(29 ページ) を参照してください。

インストールしたクラスの更新

installjava の new 句と update 句によって、現在インストールされているクラスを新しいクラスで置き換えるかを示します。

- new を指定した場合、既存のクラスと同じ名前のクラスをインストールできません。
- update を指定した場合、既存のクラスと同じ名前のクラスをインストールでき、新しくインストールしたクラスによって既存のクラスは置き換えられます。

警告！ 修正したバージョンのクラスを再インストールしてカラムのデータ型として使用しているクラスを変更する場合、そのクラスをデータ型として使用しているテーブルの既存のオブジェクト (ロー) を、既存のクラスが読んだり使用したりできることを確認します。これができない場合、オリジナルのクラスを再インストールしないかぎり既存のオブジェクトにアクセスできない場合があります。

インストールされているクラスが新しいクラスに置き換えられるかは、インストールするクラスまたはすでにインストールされているクラスが JAR と対応しているかによっても異なります。次の制限が適用されます。

- JAR を更新すると、既存の JAR 内のすべてのクラスが削除され、新しい JAR 内のクラスで置き換えられます。
- 1 つのクラスと対応させることができるのは、単一の JAR だけです。同じ名前のクラスがすでにインストールされていて別の JAR と対応している場合、JAR にそのクラスをインストールできません。同様に、クラスが現在インストールされていて JAR と対応している場合、JAR と対応していないクラスはインストールできません。

  ただし、保持した JAR に対しては、JAR と対応していないインストール済みのクラスとして、同じ名前でクラスをインストールできます。この場合、JAR と対応していないクラスは削除され、同じ名前の新しいクラスが新しい JAR と対応します。

新しい JAR にインストールしたクラスを再編成する場合は、影響を受けるクラスと JAR の対応を最初に解除する方が簡単である場合があります。詳細については、「クラスの保持」(30 ページ) を参照してください。

他の Java-SQL クラスの参照

インストールしたクラスは、同じ JAR ファイル内の他のクラスと同じデータベースに以前にインストールされているクラスを参照できますが、他のデータベース内のクラスは参照できません。

次のように JAR ファイル内のクラスが未定義のクラスを参照すると、エラーが発生する場合があります。

- 未定義のクラスが SQL 内で直接参照された場合、undefined class の構文エラーが発生します。
- 未定義のクラスが呼び出された Java メソッド内で参照された場合、呼び出された Java メソッドで Java 例外が発生するか、または 「Java-SQL メソッドの例外」(38 ページ) で説明されている一般 SQL 例外が発生します。

サポートされていないクラスやメソッドがアクティブに参照または呼び出されないかぎり、クラスの定義にそのクラスやメソッドへの参照を含ませることができます。同様に、クラスがインスタンス化または参照されないかぎり、同じデータベースにインストールされていないユーザ定義クラスへの参照を、インストールされたクラスに含めることができます。

インストールされたクラスと JAR についての情報の表示

データベースにインストールされたクラスと JAR についての情報を表示するには、sp_helpjava を使用します。構文は次のとおりです。

sp_helpjava [‘class’ [, *name* [, 'detail' | , 'depends']] |
    ‘jar’ [, *name* [, 'depends']]]

たとえば、Address クラスの詳細情報を表示するには、isql にログインして次のように入力します。

```
sp_helpjava ’class’, Address, detail
```

詳細については、『リファレンス・マニュアル』の「sp_helpjava」の項を参照してください。

インストールされたクラスと JAR のダウンロード

データベースにインストールされている Java クラスのコピーをダウンロードして、他のデータベースまたはアプリケーションで使用できます。

extractjava システム・ユーティリティを使用して、クライアントのオペレーティング・システムのファイルに JAR ファイルとそのクラスをダウンロードします。たとえば、*addr.jar* を *~/home/usera/jars/addrcopy.jar* にダウンロードするには、次のように入力します。

```
extractjava -j 'addr.jar' -f
    '~/home/usera/jars/addrcopy.jar'
```

詳細については、『ユーティリティ・ガイド』を参照してください。

クラスと JAR の削除

Transact-SQL の remove java 文を使用して、データベースから 1 つまたは複数の Java-SQL クラスをアンインストールします。remove java には、1 つまたは複数の Java クラス名、Java パッケージ名、保持した JAR 名を指定できます。たとえば、isql から utilityClasses のパッケージをアンインストールするには、次を入力します。

```
remove java package "utilityClasses"
```

注意　カラムとパラメータのデータ型として使用されているクラス、または SQLJ 関数またはストアド・プロシージャによって参照されているクラスは削除できません。他のクラスは使用状況を確認できず、ストアド・プロシージャで参照されている間でも削除できます。変数または UDF の戻り値の型として使用されるサブクラスまたはクラスを削除しないようにします。

---

remove java package コマンドは、指定したパッケージ内のクラスとサブパッケージをすべて削除します。

remove java の詳細については、『リファレンス・マニュアル』を参照してください。

クラスの保持

データベースから JAR ファイルを削除して、そのクラスを JAR と対応しないクラスとして保持できます。たとえば、保持した複数の JAR の内容を再配列する場合、retain classes オプションとともに remove java を使用します。

たとえば、isql から次のように入力します。

```
remove java jar 'utilityClasses' retain classes
```

一度 JAR との対応を解除されたクラスは、installjava に new キーワードを使用して新しい JAR と対応させることができます。

第 4 章 SQL での Java クラスの使用

この章では、Adaptive Server 環境で Java クラスを使用する方法について説明します。始めの項では概要を説明し、その後の項で詳細を説明します。

トピック名	ページ
一般考慮事項	31
データ型としての Java クラスの使用	33
SQL での Java メソッドの呼び出し	37
Java インスタンスの表現	39
Java-SQL データ項目の割り当てのプロパティ	40
Java フィールドと SQL フィールドの間でのデータ型のマッピング	42
データと識別子の文字セット	43
Java-SQL データの部分型	43
Java-SQL データでの null の扱い方	45
Java-SQL の文字列データ	48
type メソッドと void メソッド	49
等号 (=) 演算子と順序演算子	52
評価順と Java メソッド呼び出し	52
Java-SQL クラスの静的変数	55
複数のデータベース内の Java クラス	57
Java クラス	60

このマニュアルでは、データ型が Java-SQL クラスである SQL カラムと SQL 変数を、Java-SQL カラムと Java-SQL 変数、または Java-SQL データ項目と呼びます。

一般考慮事項

この項では、Java と Java-SQL 識別子の概要について説明します。

Java について

Adaptive Server データベースで Java を使用する前に、一般考慮事項について説明します。

- Java クラスには以下のものがあります。
  - Java のデータ型を宣言するフィールド。
  - パラメータおよび結果が Java のデータ型を宣言するメソッド。
  - 対応する SQL データ型がある Java データ型。「Java と SQL の間のデータ型のマッピング」(146 ページ) に定義されています。
- Java クラスに含まれるクラス、フィールド、メソッドは、private、protected、public です。

  public であるクラス、フィールド、メソッドは、SQL で参照できます。private または protected であるクラス、フィールド、メソッドは、SQL では参照できませんが、Java では参照でき、通常の Java の規則に従います。
- Java のクラス、フィールド、メソッドには、さまざまな構文のプロパティがあります。
  - クラス － フィールドの数とその名前。
  - フィールド － データ型。
  - メソッド － パラメータの数とそのデータ型、および結果のデータ型。

  SQL システムでは、Java Reflection API を使用して Java-SQL クラス自体から構文のプロパティを決定します。

Java-SQL の名前

Java-SQL のクラス名 (識別子) の長さは 255 バイトまでに制限されています。Java-SQL のフィールド名とメソッド名の長さに制限はありませんが、Transact-SQL で使用する場合は 255 バイト以下にする必要があります。Transact-SQL 文で使用する場合は、すべての Java-SQL の名前は Transact-SQL 識別子のルールに従っている必要があります。

30 バイト以上のクラス名、フィールド名、メソッド名は、二重引用符で囲んでください。

名前の最初の文字は、アルファベット (大文字または小文字) かアンダースコア (_) にします。その後の文字は、アルファベット文字、数字、ドル記号 ($)、アンダースコア (_) のどれでもかまいません。

SQL システムが大文字と小文字を区別するように指定されているかどうかにかかわらず、Java-SQL の名前では常に大文字と小文字が区別されます。

識別子の詳細については、「Java-SQL 識別子」(147 ページ) を参照してください。

データ型としての Java クラスの使用

Java クラスのセットをインストールすると、SQL で Java クラスをデータ型として参照できます。カラムのデータ型として使用するには、Java-SQL クラスを public として定義し、java.io.Serializable または java.io.Externalizable を実装する必要があります。

Java-SQL クラスは、次のものとして指定できます。

- SQL カラムのデータ型
- Transact-SQL 変数のデータ型、および Transact-SQL ストアド・プロシージャへのパラメータ
- SQL カラムのデフォルト値

テーブルを作成する場合、Java-SQL クラスを SQL カラムのデータ型として指定できます。

```
create table emps (
        name varchar(30),
        home_addr Address,
        mailing Address2Line null )
```

name カラムは、通常の SQL 文字列です。home_addr カラムと mailing_addr カラムには、Java オブジェクトが入ります。Address と Address2Line は、データベースにインストールされている Java-SQL クラスです。

Java-SQL クラスを Transact-SQL 変数のデータ型として指定できます。

```
declare @A Address
declare @A2 Address2Line
```

また、Java-SQL カラムのデフォルト値を指定できます。指定したデフォルトは定数式である必要があるという通常の制約を受けます。この式では、通常は定数の引数が指定された new 演算子を使用してコンストラクタを呼び出します。たとえば、次のようになります。

```
create table emps (
        name varchar(30),
        home_addr Address default new Address
            ('Not known', ''),
        mailing_addr Address2Line
)
```

Java-SQL カラム付きのテーブルの作成と変更

Java-SQL カラム付きのテーブルを作成したり変更したりする場合、インストールされているどの Java クラスでもカラムのデータ型として指定できます。また、カラム内の情報の保管方法も指定できます。どの保管オプションを選択するかによって、これらのカラム内での Adaptive Server によるフィールドの参照および更新の速度が異なります。

ローのカラム値は、通常は「ロー内」で格納されます。つまり、テーブルに割り付けられたデータ・ページ上に連続して格納されます。ただし、text データ項目や image データ項目を格納するのと同様に、Java-SQL カラムを「ロー外」で別の場所に格納することもできます。Java-SQL カラムのデフォルト値は、ロー外で格納されます。

Java-SQL カラムがロー内で格納されている場合は、次のようになります。

- ロー内で格納されているオブジェクトは、ロー外で格納されているオブジェクトよりも速く処理されます。
- ロー内で格納されているオブジェクトは、データベース・サーバのページ・サイズと他の変数に応じて、約 16K バイトまでを使用できます。これは、オブジェクトのフィールドの値だけではなく、オブジェクトの直列化全体が対象になります。ランタイム表現が 16 KB を超える Java オブジェクトでは例外が発生し、コマンドの実行がアボートします。

ロー外で格納されている Java-SQL カラムは、text カラムと image カラムに適用される次の制限を受けます。

- ロー外で格納されているオブジェクトは、ロー内で格納されているオブジェクトよりも遅く処理されます。
- ロー外で格納されているオブジェクトは、text カラムと image カラムの通常の制限内で任意のサイズを使用できる。
- ロー外のカラムは検査制約で参照できない。

  同様に、ロー外のカラムを含むテーブルを検査制約で参照しません。Adaptive Server では、テーブルの作成時や変更時に検査制約を含めることができますが、コンパイル時に警告メッセージが発生し、制約は実行時に無視されます。
- ロー外のカラムを、select distinct を使用して選択クエリのカラム・リストに含めることはできない。
- ロー外のカラムを、比較演算子、述部、group by 句で指定できない。

in row/off row オプションを使用した create table の構文の一部は次のようになります。

```
create table...column_name datatype
      [default {constant_expression | user | null}]
    {[{identity | null | not null}]
      [off row | [ in row [ ( size_in_bytes ) ] ] ]...
```

size_in_bytes は、ロー内のカラムの最大サイズを指定します。値は 16KB まで指定できます。デフォルト値は 255 バイトです。

create table に入力するロー内のカラムの最大サイズには、フィールドの値だけでなく、カラムの直列化全体とオーバヘッドの最小値を含める必要があります。

直列化全体の値とオーバヘッドの値を含む適切なカラム・サイズを決定するには、datalength システム関数を使用します。datalength を使用すると、カラムに格納する見本オブジェクトの実サイズを決定できます。

次に例を示します。

```
select datalength (new class_name(...))
```

class_name は、インストールされている Java-SQL クラスです。

同様に、alter table の構文の一部は次のようになります。

```
alter table...{add column_name datatype
    [default {constant_expression | user | null}]
    {identity | null} [ off row | [ in row ] ]...
```

注意　このリリースの Adaptive Server では、alter column を使用してロー内のカラムのサイズを変更することはできません。

分割テーブルの変更

Java カラムを含むテーブルが分割されている場合は、分割を削除しないとテーブルを変更できません。テーブル・スキーマを変更するには、次の手順に従います。

1　分割を削除します。

2　alter table コマンドを使用します。

3　テーブルを再分割します。

Java オブジェクトの選択、挿入、更新、削除

Java-SQL カラムを指定した後は、これらのデータ項目に割り当てる値は Java インスタンスである必要があります。このようなインスタンスは、new 演算子を使用した Java コンストラクタの呼び出しによって最初に生成されます。カラムと変数の両方に対する Java インスタンスを生成できます。

コンストラクタ・メソッドは、擬似インスタンス・メソッドであり、インスタンスを作成します。このメソッドは、クラスと同じ名前を持ち、宣言されたデータ型を持ちません。クラス定義にコンストラクタ・メソッドを含めない場合は、Java ベースのクラス・オブジェクトからデフォルトのメソッドが提供されます。各クラスに、数とタイプが異なる引数を持つ複数のコンストラクタを指定できます。コンストラクタを呼び出すと、適切な数とタイプの引数を持つコンストラクタが使用されます。

次の例では、Java インスタンスがカラムと変数の両方に対して生成されます。

```
declare @A Address, @AA Address, @A2 Address2Line,
        @AA2 Address2Line

 select @A = new Address( )
 select @AA = new Address('123 Main Street', '99123')
 select @A2 = new Address2Line( )
 select @AA2 = new Address2Line('987 Front Street',
        'Unit 2', '99543')

 insert into emps values('John Doe', new Address( ),
        new Address2Line( ))
 insert into emps values('Bob Smith',
        new Address('432 ElmStreet', ë99654í),
        new Address2Line('PO Box 99', 'attn: Bob Smith', '99678') )
```

これで、Java-SQL カラムと Java-SQL 変数に割り当てられた値を、他の Java-SQL カラムと Java-SQL 変数に割り当てることができます。次に例を示します。

```
declare @A Address, @AA Address, @A2 Address2Line,
        @AA2 Address2Line

 select @A = home_addr, @A2 = mailing_addr from emps
        where name = 'John Doe'
 insert into emps values ('George Baker', @A, @A2)

 select @AA2 = @A2
 update emps
        set home_addr = new Address('456 Shoreline Drive', '99321'),
         mailing_addr =  @AA2
         where name = 'Bob Smith'
```

また、Java-SQL カラムの値を別のテーブルにコピーすることもできます。次に例を示します。

```
create table trainees (
        name char(30),
        home_addr Address,
        mailing_addr Address2Line null
 )
insert into trainees
select * from emps
        where name in ('Don Green', 'Bob Smith',
        'George Baker')
```

Java-SQL カラムと Java-SQL 変数のフィールドを、通常の SQL 修飾を使用して参照したり更新したりできます。SQL でドットを使用して名前を修飾する場合に曖昧さをなくすには、SQL で参照するときに二重右向き山カッコ (>>) を使用して Java フィールド名と Java メソッド名を修飾してください。

```
declare @name varchar(100), @street varchar(100),
        @streetLine2 varchar(100), @zip char(10), @A Address
```

```
select @A = new Address()
select @A>>street = '789 Oak Lane'
select @street = @A>>street

select @street = home_add>>street, @zip = home_add>>zip from emps
       where name = 'Bob Smith'
select @name = name from emps
       where home_addr>>street= '456 Shoreline Drive'

update emps
       set home_addr>>street = '457 Shoreline Drive',
           home_addr>>zip = '99323'
       where home_addr>>street = '456 Shoreline Drive'
```

SQL での Java メソッドの呼び出し

Java のメソッドは SQL の中で呼び出すことができ、名前を指定することで、インスタンスのメソッドや静的メソッドを実行できます。

一般に、インスタンス・メソッドは、クラスの特定のインスタンスにカプセル化されたデータと密接に結合しています。静的 (クラス) メソッドは、クラスの特定のインスタンスではなくクラス全体に影響します。静的メソッドは、多くの場合さまざまなクラスのオブジェクトと値に適用されます。

静的メソッドは、インストールしたらすぐに使用できます。関数として使用する静的メソッドを含むクラスは、public にする必要がありますが、直列可能にする必要はありません。

Java を Adaptive Server で使用すると、値を呼び出し元に返す静的メソッドを UDF (ユーザ定義関数) として使用できるという大きな利点があります。

Java 静的メソッドは、ストアド・プロシージャ、トリガ、where 句で、または SQL の組み込み関数を使用できる場合に、UDF として使用できます。

SQL で UDF として直接呼び出された Java メソッドは、以下の制限を受けます。

- Java メソッドで JDBC を介してデータベースにアクセスする場合、結果セットの値を使用できるのは Java メソッドのみで、クライアント・アプリケーションでは使用できない。
- 出力パラメータはサポートされない。メソッドでは JDBC 接続から受け取るデータを操作できるが、メソッドの呼び出し元に返される値は、メソッドの定義の一部として宣言された単一の戻り値だけである。
- 静的メソッドのデータベース間呼び出しは、クラス・インスタンスをカラム値として使用する場合にのみサポートされる。

UDF を実行するパーミッションは、public に対して暗黙的に付与されます。UDF が JDBC を介して SQL クエリを実行する場合、データにアクセスするパーミッションは UDF の呼び出し元に対してチェックされます。したがって、ユーザ A がテーブル t1 にアクセスする UDF を呼び出す場合、ユーザ A には t1 に対する select パーミッションが必要です。パーミッションがない場合は、クエリに失敗します。Java メソッドの呼び出しにおけるセキュリティ・モデルの詳細については、「セキュリティとパーミッション」(84 ページ) を参照してください。

Java 静的メソッドを使用して結果セットと出力パラメータを返すには、メソッドを SQL ラッパで囲んで SQLJ ストアド・プロシージャまたは SQLJ 関数として呼び出します。Adaptive Server における Java メソッドの呼び出し方法の比較については、「Java メソッドの Adaptive Server での起動」(86 ページ) を参照してください。

サンプル・メソッド

サンプルの Address クラスと Address2Line クラスには、インスタンス・メソッドの toString( ) があります。また、サンプルの Misc クラスには、静的メソッドの stripLeadingBlanks( )、getNumber( )、getStreet( ) があります。値の式では値のメソッドを関数として呼び出すことができます。

```
declare @name varchar(100)
declare @street varchar(100)
declare @streetnum int
declare @A2 Address2Line

select @name = Misc.stripLeadingBlanks(name),
           @street = Misc.stripLeadingBlanks(home_addr>>street),
           @streetnum = Misc.getNumber(home_addr>>street),
           @A2 = mailing_addr
        from emps
        where home_addr>>toString( ) like '%Shoreline%'
```

void メソッド (戻り値のないメソッド) の詳細については、「type メソッドと void メソッド」(49 ページ) を参照してください。

Java-SQL メソッドの例外

Java-SQL メソッドの呼び出しが完了しても処理できない例外がある場合は、SQL 例外が発生し、次のエラー・メッセージが表示されます。

```
Unhandled Java method exception
```

例外のメッセージには、例外を引き起こした Java クラスの名前が表示されます。また、Java 例外の発生時に提供された文字列がある場合は名前の後に表示されます。

Java インスタンスの表現

isql などの非 Java クライアントは、サーバから直列 Java オブジェクトを受け取ることができません。オブジェクトを表示したり使用したりするには、Adaptive Server でオブジェクトを表示可能な表現に変換する必要があります。

実際の文字列値を使用するには、Adaptive Server でオブジェクトを char 値または varchar 値に変換するメソッドを呼び出す必要があります。Address クラスの toString() メソッドは、このようなメソッドの例です。toString() メソッドの独自のバージョンを作成して、オブジェクトの表示可能な表現を操作してください。

注意　Java API の toString() メソッドでは、オブジェクトは表示可能な表現に変換されません。ユーザが作成した toString() メソッドによって、Java API の toString() メソッドは上書きされます。

---

toString( ) メソッドを使用する場合、Adaptive Server では返されるバイト数に制限があります。Adaptive Server では、オブジェクトの出力可能な表現を @@stringsize グローバル変数にトランケートします。@@stringsize のデフォルト値は 50 です。この値は、set stringsize コマンドを使用して変更できます。次に例を示します。

```
set stringsize 300
```

コンピュータのディスプレイ・ソフトウェアによっては、データ項目をさらにトランケートし、ラップしないで画面に収まるようにすることがあります。

toString( ) またはこれに類したメソッドを各クラスに含める場合、オブジェクトの toString( ) メソッドの値を次の 2 つの方法のどちらかで返すことができます。

- Java-SQL カラムの特定のフィールドを選択する。これによって、toString( ) が自動的に呼び出されます。

  ```
  select home__addr>>street from emps
  ```

- カラムと toString( ) メソッドを選択する。これによって、カラムのフィールド値すべてが 1 つの文字列にリストされます。

  ```
  select home_addr>>toString() from emps
  ```

Java-SQL データ項目の割り当てのプロパティ

Java-SQL データ項目に割り当てられた値は、Java VM 内で Java-SQL メソッドによって構築された値から最終的に抽出されます。しかし、Java-SQL の変数、パラメータ、結果の論理的表現は、Java-SQL カラムのそれとは異なります。

- Java-SQL カラム。これは、永続的なもので、含まれているテーブルのローに格納される Java の直列化されたストリームです。格納された値であり、Java インスタンスの表現を含みます。
- Java-SQL の変数、パラメータ、結果関数。これらは、一時的なものです。Java-SQL インスタンスを実際には含まず、Java VM にある Java インスタンスへの参照を含みます。

これらの表現の違いによって、次の例に示すように割り当てのプロパティが異なります。

- new 演算子付きの Address コンストラクタ・メソッドは、Java VM で評価される。このメソッドは、Address インスタンスを構築して、そのインスタンスへの参照を返します。この参照は、Java-SQL 変数 *@A* として割り当てられます。

```
declare @A Address, @AA Address, @A2 Address2Line,
        @AA2 Address2Line
select @A = new Address('432 Post Lane', '99444')
```

- 変数 *@A* には Java VM の Java インスタンスへの参照が入る。この参照は、変数 *@AA* にコピーされます。これで、変数 *@A* と *@AA* は同じインスタンスを参照することになります。

```
select @AA=@A
```

- この割り当てによって、*@A* によって参照される Address の zip フィールドが修正される。これは、*@AA* によって参照されるものと同じ Address インスタンスです。したがって、*@A.zip* の値と *@AA.zip* の値は、ともに '99222' になります。

```
select @A>>zip='99222'
```

- new 演算子付きの Address コンストラクタ・メソッドは、Address インスタンスを構築して、そのインスタンスへの参照を返す。しかし、このターゲットは Java-SQL カラムなので、SQL システムでは、その参照によって示される Address インスタンスを直列化します。そして、その直列化した値を emps テーブルの新しいローにコピーします。

```
insert into emps
        values ('Don Green', new Address('234 Stone
        Road', '99777'), new Address2Line( ) )
```

Address2Line コンストラクタ・メソッドは Address メソッドと同様に動作します。ただし、このメソッドは、指定されたパラメータ値の付いたインスタンスではなく、デフォルトのインスタンスを返します。それ以外では、Address インスタンスの場合と同じように動作します。SQL システムでは、デフォルトの Address2Line インスタンスを直列化して、直列化した値を emps テーブルの新しいローに保管します。

- insert 文は、mailing_addr カラムに値を指定しない。このため、このカラムは、insert で値が指定されていないカラムと同じ方法で null に設定されます。この null 値は SQL 全体で生成されます。mailing_addr カラムを初期化しても、Java VM はまったく呼び出されません。

```
insert into emps (name, home_addr)  values ('Frank Lee', @A)
```

insert 文は、home_addr カラムの値を Java-SQL の変数 *@A* から取得するように指定します。この変数には Java VM の Address インスタンスへの参照が入ります。このターゲットは Java-SQL カラムなので、SQL システムでは *@A* によって示される Address インスタンスを直列化します。そして、その直列化した値を emps テーブルの新しいローにコピーします。

- 次の文によって、"Bob Brown" 用の新しい emps ローが挿入される。home_addr カラムの値は、SQL 変数 *@A* から取得します。これも、*@A* によって参照される Java インスタンスの直列化の 1 つです。

```
insert into emps (name, home_addr)  values ('Bob Brown', @A)
```

- 次の update 文は、"Frank Lee" ローの home_addr カラムの zip フィールドを "99777" に設定する。この設定を行っても、"Bob Brown" ローの zip フィールドは影響されず、"99444" のまま変わりません。

```
update emps
        set home_add>>zip = '99777'
        where name = 'Frank Lee'
```

- Java-SQL カラム home_addr には、Address インスタンスの値の直列化された表現が入る。SQL システムでは、Java VM を呼び出してその表現を Java VM の Java インスタンスとして非直列化し、非直列化したコピーへの参照を返します。この参照は、*@AA* に割り当てられます。*@AA* によって参照される非直列化された Address インスタンスは、カラム値からも *@A* によって参照されるインスタンスからも完全に独立しています。

```
select @AA = home_addr from emps where name = 'Frank Lee'
```

- この割り当てによって、*@A* によって参照される Address インスタンスの zip フィールドが修正される。このインスタンスは、"Frank Lee" ローの home_addr カラムのコピーですが、そのカラム値からは独立しています。したがって、この割り当てでは、"Frank Lee" ローの home_addr カラムの zip フィールドは修正されません。

```
select @A>>zip = '95678'
```

Java フィールドと SQL フィールドの間でのデータ型のマッピング

Java VM と Adaptive Server の間でデータを転送するときは、データ項目のデータ型がシステムによって異なることに注意してください。Adaptive Server では、「Java と SQL の間のデータ型のマッピング」(146 ページ) の対応表どおりに、SQL 項目を Java 項目に、Java 項目を SQL 項目に自動的にマップします。

したがって、たとえば、SQL の型 char は Java の型 String に変換され、SQL の型 binary は Java の型 byte[ ] に変換されます。

- SQL から Java へのデータ型の対応の場合は、どのような長さの char 型、varchar 型、varbinary 型でも Java の String 型または byte[ ] 型に適宜対応する。
- Java から SQL へのデータ型の対応の場合は、次のようになる。
  - Java の String データ型と byte[ ] データ型は、SQL の varchar データ型と varbinary データ型に対応する。最大長の値 16KB は、Adaptive Server によって定義される。
  - Java の BigDecimal データ型は、SQL の numeric(precision,scale) に対応する。精度と位取りは、ユーザによって定義される。

emps テーブルでは、Address クラス、Address2Line クラス、street フィールド、zip フィールド、line2 フィールドの最大値は、225 バイト (デフォルト値) です。これらのクラスの Java でのデータ型は java.String です。このデータ型は、SQL では varchar(255) として扱われます。

Java オブジェクトのデータ型を持つ式は、SQL のコンテキストで使用される場合のみ、対応する SQL のデータ型に変換されます。たとえば、従業員 "Smith" のフィールド home_addr>>street が 260 文字で、"6789 Main Street ..." で始まる場合、次のようになります。

```
select Misc.getStreet(home_addr>>street) from emps where name='Smith'
```

select リストの式は、home_addr>>street フィールドの 260 文字の値を getStreet( ) メソッドに渡します (この 260 文字の値は、ここでは 255 文字にトランケートされません)。次に、getStreet( ) メソッドは、"Main Street ..." で始まる 255 文字の文字列を返します。この 255 文字の文字列は、ここでは SQL の select リストの要素です。したがって、この文字列は SQL のデータ型に変換され、必要に応じて 255 文字にトランケートされます。

データと識別子の文字セット

Java ソース・コードと Java String データの文字セットは、両方とも Unicode です。Java-SQL クラスのフィールドには、Unicode データを入れることができます。

注意 参照できるクラスの完全に修飾された名前や参照できるメンバの名前で使用される Java 識別子には、ラテン文字とアラビア数字のみを使用できます。

Java-SQL データの部分型

クラスの部分型を使うと、Java の特性である部分型の置換とメソッドの上書きを使用できます。あるクラスからそのスーパークラスの 1 つへの変換は範囲を広げる変換になり、あるクラスからそのサブクラスの 1 つへの変換は範囲を狭める変換になります。

- 範囲を広げる変換は、通常の割り当てと比較によって暗黙的に実行される。すべてのサブクラス・インスタンスがスーパークラスのインスタンスでもあるので、この変換は常に成功します。
- 範囲を狭める変換を指定するには、明示的な convert 式を使用する。範囲を狭める変換が成功するのは、スーパークラス・インスタンスがサブクラスのインスタンスであるか、サブクラスのサブクラスである場合のみです。それ以外の場合は、例外が発生します。

範囲を広げる変換

範囲を広げる変換を指定する場合は、convert 関数を使用する必要はありません。たとえば、Address2Line クラスは Address クラスのサブクラスなので、Address2Line 値は Address データ項目に割り当てることができます。emps テーブルでは、home_addr カラムは Address データ型になり、mailing_addr カラムは Address2Line データ型になります。

```
update emps
    set home_addr = mailing_addr
    where home_addr is null
```

where 句を満たすローの場合、home_addr カラムには Address2Line が入ります。しかし、home_addr の宣言された型は Address です。

このような割り当てでは、クラスのインスタンスはそのクラスのスーパークラスのインスタンスとして暗黙的に扱われます。サブクラスのランタイム・インスタンスは、そのサブクラスのデータ型と関連データを保持します。

範囲を狭める変換

あるクラスのインスタンスをそのクラスのサブクラスのインスタンスに変換するには、convert 関数を使用します。次に例を示します。

```
update emps
        set mailing_addr = convert(Address2Line, home_addr)
        where mailing_addr is null
```

update 文で範囲を狭める変換を、Address2Line ではない Address インスタンスがある home_addr カラムに適用すると、例外が発生します。where 句に条件を組み込むと、このような例外を回避できます。

```
update emps
        set mailing_addr = convert(Address2Line, home_addr)
        where mailing_addr is null
        and home_addr>>getClass( )>>toString( ) = 'Address2Line'
```

"home_addr>>getClass( )>>toString( )" という式では、Java の Object クラスの getClass( ) メソッドと toString( ) メソッドを呼び出します。Object クラスは、暗黙的にすべてのクラスのスーパークラスになるので、このクラスに対して定義されたメソッドはすべてのクラスで使用できます。

また、case 式も使用できます。

```
update emps
        set mailing_addr =
            case
                when home_addr>>getClass( )>>toString( )
                  ='Address2Line'
                then convert(Address2Line, home_addr)
                else null
            end
        where mailing_addr is null
```

ランタイム・データ型とコンパイル時データ型

範囲を広げる変換と範囲を狭める変換のどちらの場合も、実際のインスタンス値またはそのランタイム・データ型は修正されません。コンパイル時の型に使用するクラスが指定されるだけです。したがって、Address2Line の値を mailing_addr カラムから home_address カラムに格納する場合、その値のランタイムの型は Address2Line のままです。

たとえば、Address クラスと Address2Line サブクラスには両方ともメソッド toString( ) があります。このメソッドによって、完全なアドレス・データの String フォームが返されます。

```
select name, home_addr>>toString( )  from emps
        where home_addr>>toString( ) not like '%Line2=[ ]'
```

emps の各ローでは、home_addr カラムの宣言された型は Address です。しかし、home_addr 値のランタイムの型は、前の update 文の結果により Address または Address2Line になります。home_addr カラムのランタイム値が Address であるローでは、Address クラスの toString( ) メソッドが呼び出されます。また、home_addr カラムのランタイム値が Address2Line であるローでは、Address2Line サブクラスの toString( ) メソッドが呼び出されます。

範囲を広げる変換と範囲を狭める変換での null 値の詳細については、「SQL convert 関数を使用する場合の null 値」(47 ページ) を参照してください。

Java-SQL データでの null の扱い方

この項では、Java-SQL データ項目での null の使用方法について説明します。

null インスタンスのフィールドとメソッドに対する参照

フィールド参照で指定されたインスタンスの値が null の場合、そのフィールド参照は null になります。同じように、インスタンス・メソッド呼び出しで指定されたインスタンスの値が null の場合、その呼び出しの結果は null になります。

Java では、null インスタンスのフィールド参照やメソッド参照の影響に対してさまざまなルールがあります。Java で null インスタンスのフィールドを参照しようとすると、例外が発生します。

たとえば、emps テーブルに次のローがあるとします。

```
insert into emps (name, home_addr)
    values ("Al Adams",
    new Address("123 Main", "95321"))

insert into emps (name, home_addr)
    values ("Bob Baker",
    new Address("456 Side", "95123"))

 insert into emps (name, home_addr)
    values ("Carl Carter", null)
```

次の select について考えてみます。

```
select name, home_addr>>zip  from emps
where home_addr>>zip in ('95123', '95125', '95128')
```

Java ルールが "home_addr>>zip" への参照に使用された場合、この参照によって "home_addr" カラムが null である "Carl Carter" ローに対して例外が発生します。こうした例外を回避するには、select を次のように記述する必要があります。

```
select name,
```

```
        case when home_addr is not null then home_addr>>zip
        else null end

    from emps
        where case when home_addr is not null
        then home_addr>>zip
    else
        null end
    in ('95123', '95125', '95128')
```

SQL の規約は、このため、null インスタンスのフィールドおよびメソッド参照に使用します。インスタンスが null の場合、フィールドまたはメソッド参照は null です。この SQL ルールの影響によって、上記の case 文は暗黙的になります。

ただし、null インスタンスを使用したフィールド参照に関するこの SQL ルールは、ソース・コンテキスト (右側) でのフィールド参照のみに適用され、割り当てまたは set 句のターゲット (左側) であるフィールド参照には適用されません。次に例を示します。

```
    update emps
        set home_addr>>zip D '99123'
        where name D 'Charles Green'
```

この where 句は "Charles Green" ローでは明らかに true なので、update 文が set 句を実行しようとします。この場合は、例外が発生します。これは、値を割り当てられるフィールドが null インスタンスになく、null インスタンスのフィールドに値を割り当てられないためです。したがって、null インスタンスのフィールドに対するフィールド参照が有効なので、右側のコンテキストに null 値が返され、左側のコンテキストに例外が発生します。

null インスタンスのメソッドの呼び出しにも同じことがあてはまり、同じルールが適用されます。たとえば、前の例を修正して、home_addr カラムの toString( ) メソッドを呼び出します。

```
    select name, home_addr>>toString( )from emps
        where home_addr>>toString( ) D
        'StreetD234 Stone Road ZIPD 99777'
```

インスタンス・メソッド呼び出しで指定されたインスタンスの値が null の場合、その呼び出し結果は null になります。このため、select 文はここでは有効ですが、Java では例外が発生します。

Java-SQL メソッドに対する引数としての null 値

パラメータとして null を渡す結果は、その値を引数とするメソッドのアクションからは独立していますが、null 値を配信するリターン・データ型の機能に依存しています。

Java のスカラ型メソッドに対して null 値をパラメータとして渡すことはできません。Java のスカラ型では、null は認められていません。しかし、Java のオブジェクト型では、null 値は認められています。

次の Java-SQL クラスを例として示します。

```
public class General implements java.io.Serializable {
        public static int identity1(int I) {return I;}
        public static java.lang.Integer identity2
          (java.lang.Integer I) {return I;}
        public static Address identity3 (Address A) {return A;}
   }
```

次の呼び出しを考慮します。

```
declare @I int
declare @A Address;

select @I = General.identity1(@I)
select @I = General.identity2(new java.lang.Integer(@I))
select @A = General.identity3(@A)
```

変数 *@I* と変数 *@A* には値が割り当てられていないので、どちらの値も null になります。

- identity1( ) メソッドを呼び出すと、例外が発生する。identity1( ) のパラメータ *@I* のデータ型は、Java の int 型です。この型はスカラで、null 状態になりません。identity1( ) に対して値が null の引数を渡そうとすると、例外が発生します。
- identity2( ) メソッドを呼び出すと、成功する。identity2( ) のパラメータのデータ型は、Java クラスの java.lang.Integer です。new 式では、変数 *@I* の値に設定される java.lang.Integer のインスタンスが作成されます。
- identity3( ) メソッドを呼び出すと、成功する。

identity1( ) の呼び出しが成功した場合に、null の結果が返されることはありません。この戻り型は null 状態にならないからです。パラメータの型情報がないとメソッドの解析に失敗するため、null は直接渡せません。

identity2( ) の呼び出しと identity3( ) の呼び出しが成功した場合には、null の結果を返すことがあります。

SQL *convert* 関数を使用する場合の null 値

あるクラスの Java オブジェクトをそのクラスのスーパークラスまたはサブクラスの Java オブジェクトに変換するには、convert 関数を使用します。

「Java-SQL データの部分型」(43 ページ) で説明したように、emps テーブルの home_addr カラムには Address と Address2Line の両方のクラスの値を入れることができます。次に例を示します。

```
select name, home_addr>>street, convert(Address2Line, home_addr)>>line2,
        home_addr>>zip from emps
```

"convert(Address2Line, home_addr)" という式では、データ型 (Address2Line) と式 (home_addr) を指定します。コンパイル時には、式 (home_addr) はクラス (Address2Line) の部分型またはスーパー型である必要があります。実行時には、式の値のランタイム型がクラス、サブクラス、またはスーパークラスのどれであるかによって、この convert 呼び出しのアクションが異なります。

- 式 (home_addr) のランタイム値が指定のクラス (Address2Line) またはそのサブクラスである場合は、その式の値が指定されたデータ型 (Address2Line) で返される。
- 式 (home_addr) のランタイム値が指定のクラス (Address) のスーパークラスである場合は、null が返される。

Adaptive Server では、結果のローごとに select 文を評価します。各ローは次のように処理されます。

- home_addr カラムの値が Address2Line である場合は、convert によってその値が返され、フィールド参照によって line2 フィールドが抽出される。convert が null を返す場合は、フィールド参照自体が null である。
- convert が null を返す場合は、フィールド参照自体が null と評価される。

このため、select の結果は、home_addr カラムが Address2Line のローでは line2 値になり、home_addr カラムが Address のローでは null になります。「Java-SQL データでの null の扱い方」(45 ページ) で説明したように、select は home_addr カラムが null のローでは null の line2 値も示します。

Java-SQL の文字列データ

Java-SQL カラムでは、String 型のフィールドは Unicode として保管されます。

Java-SQL の String フィールドが、char 型、varchar 型、nchar 型、nvarchar 型、または text 型の SQL データ項目に割り当てられる場合、Unicode データは SQL システムの文字セットに変換されます。変換エラーは、set char_convert オプションを使用して指定します。

char 型、varchar 型、nchar 型、または text 型の SQL データ項目が、Unicode として保管される Java-SQL String フィールドに割り当てられる場合、文字データは Unicode に変換されます。このようなデータに未定義のコードポイントがあると、変換エラーが発生します。

長さがゼロの文字列

Transact-SQL では、長さがゼロの文字列は null 値として扱われ、空の文字列 ( ) はシングル・スペースとして扱われます。

Transact-SQL との一貫性を保つため、長さがゼロの Java-SQL String 値が、char 型、varchar 型、nchar 型、nvarchar 型、または text 型の SQL データ項目に割り当てられる場合は、Java-SQL String 値はシングル・スペースに置き換えられます。

次に例を示します。

```
1> declare @s varchar(20)
2> select @s = new  java.lang.String()
3> select @s, char_length(@s)
4> go

(1 row affected)

-----------------   -----------------
                         1
```

上記以外の場合、長さがゼロの値は SQL では SQL null として扱われます。また、Java-SQL String に割り当てられた場合、Java-SQL String は Java null になります。

type メソッドと void メソッド

Java メソッドは、インスタンス・メソッドと静的メソッドのどちらも type メソッドまたは void メソッドのいずれかになります。一般に、type メソッドは結果の型の値を返します。void メソッドはアクションを実行しますが、何も返しません。

たとえば、Address クラスでは次のようになります。

- toString( ) メソッドは、String 型の *type* メソッドである。
- removeLeadingBlanks( ) メソッドは、*void* メソッドである。
- Address コンストラクタ・メソッドは、型が Address クラスの *type* メソッドである。

コンストラクタ・メソッドを呼び出す場合は、type メソッドを関数として呼び出し、new キーワードを使用します。

```
insert into emps
        values ('Don Green', new Address('234 Stone Road', '99777'),
            new Address2Line( ) )

select name, home_addr>>toString( ) from emps
        where home_addr>>toString( )  like ë%Baker%í
```

Address クラスの removeLeadingBlanks( ) メソッドは、指定されたインスタンスの street フィールドと zip フィールドを修正する void インスタンス・メソッドです。emps テーブルのそれぞれのローの home_addr カラムに対して removeLeadingBlanks( ) を呼び出すことができます。次に例を示します。

```
update emps
    set home_addr =
        home_addr>>removeLeadingBlanks( )
```

removeLeadingBlanks( ) は、home_addr カラムの street フィールドと zip フィールドから先行ブランクを削除します。Transact-SQL の update 文は、このようなアクションに対してフレームワークや構文を提供しません。カラム値を置き換えるだけです。

Java の void インスタンス・メソッド

SQL システムで Java の void インスタンス・メソッドの「置き換え更新」アクションを使用するために、Adaptive Server の Java は Java の void インスタンス・メソッドの呼び出しを次のように扱います。

クラス C のインスタンス CI の void インスタンス・メソッド M( ) の場合、"CI.M(...)" と記述され、次のようになります。

- SQL では、これは type メソッド呼び出しとして扱われる。結果の型は暗黙的にクラス C になり、結果の値は CI への参照になります。void インスタンス・メソッドの呼び出し後、この参照ではインスタンス CI のコピーが示されます。
- Java では、これは void メソッド呼び出しである。この呼び出しは、アクションを実行しますが値を返しません。

たとえば、emps テーブルの選択したローにある home_addr カラムの removeLeadingBlanks( ) メソッドを、次のように呼び出すことができます。

```
update emps
        set home_addr = home_addr>>removeLeadingBlanks( )
        where home_addr>>removeLeadingBlanks( )>>street like ì123%î
```

1. where 句で、“home_addr>>removeLeadingBlanks( )” は emps テーブルのローの home_addr カラムに対して、removeLeadingBlanks( ) メソッドを呼び出します。removeLeadingBlanks( ) は、そのカラムのコピーの street フィールドと zip フィールドから先行ブランクを取り除きます。その後、SQL システムによって、home_addr カラムの修正済みのコピーに対する参照が返されます。後続のフィールド参照は次のようになります。

   ```
   home_addr>>removeLeadingBlanks( )>>street
   ```

   このフィールド参照は、先行ブランクが削除された street フィールドを返します。where 句の home_addr に対する参照は、そのカラムのコピーに対して機能します。where 句がここで評価されても、home_addr カラムは修正されません。

2 update 文は、where 句が true である emps テーブルのローごとに set 句を実行します。

3 set 句の右側では、“home_addr>>removeLeadingBlanks( )” の呼び出しは、where 句に対するもののように実行されます。removeLeadingBlank( ) は、そのコピーの street フィールドと zip フィールドから先行ブランクを取り除きます。その後、SQL システムによって、home_addr カラムの修正済みのコピーに対する参照が返されます。

4 set 句の右側にある結果によって示される Address インスタンスは、直列化されて、set 句の左側に指定されたカラム内にコピーされます。set 句の右側にある式の結果は home_addr カラムがコピーされたものです。このコピーでは、street フィールドと zip フィールドから先行ブランクが削除されています。修正済みのコピーは、その後、home_addr カラムの新しい値としてそのカラムに再度割り当てられます。

set 句の左右の式は独立しています。これは、update 文では普通です。

次の update 文は、左側にある home_address カラムに割り当てられている set 句の右側にある mailing_addr カラムの void インスタンス・メソッド呼び出しを示します。

```
update emps
        set home_addr = mailing_addr>>removeLeadingBlanks( )
        where ...
```

この set 句では、mailing_addr カラムの void メソッド removeLeadingBlanks( ) が、mailing_addr カラムにある Address2Line インスタンスの修正済みのコピーに対する参照を解放します。この参照によって示されるインスタンスは、直列化され、home_addr カラムに割り当てられます。このアクションによって、home_addr カラムは更新されますが、mailing_addr カラムは影響を受けません。

Java の void 静的メソッド

簡単な SQL の execute コマンドを使用して void 静的メソッドを呼び出すことはできません。void 静的メソッドの呼び出しを、select 文に置いてください。

たとえば、Java クラス C に void 静的メソッド M(...) があり、M( ) は SQL で呼び出すアクションを実行するものとします。M( ) は JDBC 呼び出しを使用して、create や drop など、戻り値のない一連の SQL 文を実行できます。これらは void メソッドに適しています。

select コマンド内では void 静的メソッドを次のように呼び出します。

```
select C.M(...)
```

select を使用して void 静的メソッドを呼び出すことができるようにするために、SQL では、null 値を持つデータ型 int の値を返すものとして void 静的メソッドを扱います。

等号 (=) 演算子と順序演算子

データベースで Java を使用する場合、等号 (=) 演算子と順序演算子を使用できます。次の処理はできません。

- 順序演算子での Java-SQL データ項目の参照。
- 等号 (=) 演算子での Java-SQL データ項目の参照 (その項目がロー外のカラムに格納されている場合)。
- order by 句の使用。この句を使用するにはソート順を決めておく必要があります。
- ">"、"<"、"<="、または ">=" 演算子を使用した直接比較。

このような等号 (=) 演算子はロー内のカラムで使用できます。

- distinct キーワードの使用。このキーワードは、Java-SQL カラムを含む、ローの等号の項で定義されます。
- "=" 演算子と "!=" 演算子を使用した直接比較。
- union 演算子の使用 (union all 演算子ではない)。この演算子は、重複を削除するもので、distinct 句と同じ種類の比較が必要です。
- group by 句の使用。この句は、グループ化カラムの値が等しいセットにローを分割します。

評価順と Java メソッド呼び出し

Adaptive Server では、比較と他の演算のオペランドを評価するための順序が定義されていません。その代わり、Adaptive Server では、それぞれのクエリを評価して、最も実行速度が速いものに基づいて評価順を選択します。

この項では、カラムまたは変数、およびパラメータを引数として渡す場合に評価順を変えると、結果がどのようになるかについて説明します。この項の例では、次のように Java-SQL クラスを使用します。

```
public class Utility implements java.io.Serializable {
            public static int F (Address A) {
                if (A.zip.length( ) > 5) return 0;
                else {A.zip = A.zip + "-1234"; return 1;}
            }
            public static int G (Address A) {
                if (A.zip.length( ) > 5) return 0;
                else {A.zip = A.zip + "-1234"; return 1;}
            }
}
```

カラム

通常は、同じ Java-SQL オブジェクト上で同じ SQL 文の複数のメソッドを呼び出すのは避けます。少なくともそれらのメソッドの 1 つがオブジェクトを修正する場合、評価順により結果が異なる可能性があります。

次に例を示します。

```
select *  from emp E
       where Utility.F(E.home_addr) > Utility.F(E.home_addr)
```

where 句は、2 つの異なるメソッド呼び出しに同じ home_addr カラムを渡します。home_addr カラムに "95123" のような 5 文字の zip があるローに対する where 句の評価について考えてみます。

Adaptive Server では、比較の左側または右側のどちらかを最初に評価します。最初の評価が完了すると、もう一方を評価します。実行速度が高速になるので、Adaptive Server では最初の呼び出しによる引数の変更を 2 番目の呼び出しで参照できます。

この例では、Adaptive Server によって選択された最初の呼び出しから 1 が返され、2 番目の呼び出しから 0 が返されます。左のオペランドが最初に評価される場合、比較は 1>0 になり、where 句は true になります。右のオペランドが最初に評価される場合、比較は 0>1 になり、where 句は false になります。

変数とパラメータ

同じように、変数とパラメータを引数として渡す場合も、評価順によって結果が変わります。

次の文を考えてみます。

```
declare @A Address
declare @Order  varchar(20)

select @A = new Address('95444', '123 Port Avenue')
select case when Utility.F(@A)>Utility.G(@A)
           then ‘Left’ else ‘Right’ end
select @Order = case when utility.F(@A) > utility.G(@A)
           then 'Left'  else 'Right' end
```

新しい Address には 5 文字の zip コード・フィールドがあります。case 式が評価される場合、左右の比較のオペランドのどちらが先に評価されるかによって、比較は 1>0 または 0>1 のどちらかになります。また、*@Order* 変数は、その比較に応じて "Left" または "Right" に設定されます。

カラムの引数の場合と同様、式の値も評価順により異なります。左右の比較のオペランドのどちらが先に評価されるかによって、*@A* によって参照される Address インスタンスの zip フィールドの結果は、"95444-4321" または "95444-1234" のどちらかの値になります。

式の中の deterministic な Java 関数

同じ入力値のセットを使用して評価されている場合、deterministic 式および関数は常に同じ結果を返します。Adaptive Server のすべての Java 関数は deterministic です。その結果、Java 関数にかかわる式のパラメータと入力値が変化しない場合、Adaptive Server はその式全体を deterministic として処理します。

Adaptive Server が式の中で Java 関数を検出すると、その式を即時に計算し、その計算を一度だけ実行して各ローに対して繰り返さないようにします。これによってパフォーマンスが改善されますが、予期しない動作が発生することもあります。

次に例を示します。

```
1> create table CaseTest
2> (TestValue varchar(50))
3> go
1> insert into CaseTest values(’07’)
2> go
(1 row affected)

1> declare @IntArray sybase.cpp.value.client/common.IntArray
2> select @IntArray = new sybase.cpp.value.client.common.IntArray()
3> SELECT CASE
4> WHEN CT.TestValue = ’07’
5> THEN @IntArray >> setInt(new java.lang.Integer(10))
6> ELSE @IntArray >> setInt(new java.lang.Integer(11))
7> END
8> FROM CaseTest CT
9> select @IntArray >> getInt(0) as GetObjAfter0
10> select @IntArray >> getInt(1) as GetObjAfter1
11> select @IntArray >> getArraySize() as NumObjectsOnArray
12> go

----------------------------------------------
sybase.cpp.value.client.common.IntArray@22cc0f30

(1 row affected)
GetObjAfter0
----------------------------------------------
11

(1 row affected)
NumObjectsOnArray
---------------
2

(1 row affected)
```

case 文の 1 つの分岐で true と評価され、整数の配列には 1 つの値 (10) だけが挿入されると想定しますが、式 `setInt(new java.lang.Integer(10))` および `setInt(new java.lang.Integer(11))` は deterministic であるため、Adaptive Server はその結果を「あらかじめ計算」し、その両方の値を配列に設定します。

カラムへの参照を追加すると、繰り返し実行した式の計算結果が同じであると Adaptive Server が判断できなくなるため、式を nondeterministic にできます。たとえば、この例の Transact-SQL を次のように変更します。

```
1> declare @IntArray Sybase.cpp.value.client.common.IntArray
2> select @IntArray = new sybase.cpp.value.client.common.IntArray()
3> SELECT CASE
4> WHEN CT.TestValue = '07'
5> THEN @IntArray >> setInt(new java.lang.Integer(10 + convert(int,CT.TestValue) -
convert(int,CT.TestValue)))
6> ELSE @IntArray >> setInt(new java.lang.Integer(11 + convert(int,CT.TestValue) -
convert(int,CT.TestValue)))
7> END
8> FROM CaseTest CT
9> select @IntArray >> getInt(0) as GetObjAfter0
10> select @IntArray >> getInt(1) as GetObjAfter1
11> select @IntArray >> getArraySize() as NumObjectsOnArray
12> go
```

case 文の THEN および ELSE の部分にカラム参照を含めると、オプティマイザはその文を定数として処理できず、Java の insert 文をあらかじめ計算できなくなります。

Java-SQL クラスの静的変数

静的と宣言される Java の変数は、クラスのそれぞれのインスタンスではなく、Java クラスと対応しています。変数は、クラス全体に対して一度割り付けられます。

たとえば、Street フィールドの長さの推奨制限値を指定する Address クラス内に、静的変数を含めることができます。

```
public class Address implements java.io.Serializable {

    public static int recommendedLimit;
    public String street;
    public String zip;
// ...
}
```

静的変数を final と指定できます。これは、その変数は更新不可能であることを示します。

```
public static final int recommendedLimit;
```

それ以外の場合は、変数を更新できます。

静的変数をクラスのインスタンスで修飾して、SQL の Java クラスの静的変数を参照します。次に例を示します。

```
declare @a Address
select @a>>recommendedLimit
```

クラスのインスタンスがない場合は、次の方法を使用します。

```
select convert(Address, null)>>recommendedLimit
```

式 "(convert(null, Address))" は、null 値を Address 型に変換します。つまり、null Address インスタンスを生成します。このインスタンスは、静的変数名で修飾できます。SQL の Java クラスの静的変数は、静的変数をクラス名で修飾しても参照できません。たとえば、次の文はいずれも正しくありません。

```
select Address.recommendedLimit
```

```
select Address>>recommendedLimit
```

final 以外の静的変数に割り当てられた値にアクセスできるのは、現在のセッション内のみです。

Adaptive Server 15.0.3 およびそれ以降での静的変数の変更

Adaptive Server 15.0.2 およびそれ以前では、それぞれのタスクに専用の内部 JVM が割り当てられていました。各 JVM は ClassLoader の一意のセットと関連付けられました。その結果、クラス変数は Adaptive Server の 1 つのタスクでのみ使用可能でした。

Adaptive Server 15.0.3 およびそれ以降では、PCA/JVM が導入され、Adaptive Server タスクのそれぞれに対して同じ JVM 内の個別の JVM スレッドが使用されます。すべてのユーザ・クラスは、特定の Java メソッドを実行する固有の Adaptive Server タスクだけと関連付けられた ClassLoader によってロードされます。ユーザ・クラスと関連付けられた ClassLoader は Adaptive Server タスク間で共有されないため、ユーザ・クラスは同じであるとは見なされません。したがって、ユーザ・クラスのクラス変数は、別の Adaptive Server タスクからは不可視です。

ただし、すべてのユーザ ClassLoaders はシステム ClassLoader を親として共有するため、システム ClassLoader によってロードされたクラスのクラス変数は、すべての Adaptive Server タスクから見られます。これは、すべての標準 JVM について当てはまります。このようなクラスのクラス変数は、複数のタスクにまたがって使用されるときでも機能やセキュリティを危険にさらしません。

Cluster Edition での静的変数の変更

Cluster Edition では、「Adaptive Server 15.0.3 およびそれ以降での静的変数の変更」(56 ページ) で説明されているとおり、ユーザ・クラスのクラス変数とシステム ClassLoader によってロードされたクラスのクラス変数を Adaptive Server が処理します。ただし、それぞれのノードでは独立した、互いに関連しない PCA/JVM インスタンスが実行されています。あるクラス変数を 1 つのノードで設定した場合、クラスタ内のほかのすべてのノードではその値は自動的には変更されません。Adaptive Server タスクは複数のノードにまたがって実行できるため、ユーザ・クラスがクラス変数に依存する場合は、すべてのノードで同じクラス変数を明示的に設定する必要があります。

複数のデータベース内の Java クラス

同じ Adaptive Server システムの異なるデータベース内に同じ名前の Java クラスを保管できます。この項では、このようなクラスの使用方法について説明します。

スコープ

1 つの Java クラスまたはクラスのセットは、現在のデータベースにインストールします。データベースをダンプしたり、ロードしたりするときには、現在そのデータベースにインストールされている Java-SQL クラスを常に含めます。同じ名前のクラスが Adaptive Server システムの他のデータベースに存在している場合でも同様です。

異なるデータベース内に同じ名前を持つ Java クラスをインストールできます。同義クラスは次のいずれかになります。

- 異なるデータベースにインストールされている同一のクラス。
- 相互に互換性があるように意図された異なるクラス。したがって、いずれかのクラスで生成された直列化された値を、他のクラスでも受け入れられます。
- 「上位」互換性があるように意図された異なるクラス。つまり、1 つのクラスで生成された直列化された値を、他のクラスで受け入れられますが、その逆はできません。
- 相互に互換性がないように意図された異なるクラス。たとえば、紙の業者用に設計された Sheet という名前のクラスと、リネン業者用に設計された Sheet という名前のクラスがあります。

データベース間での相互参照

あるデータベース内にあるオブジェクトを、別のデータベースから参照できます。

たとえば、次のように設定されていると仮定してください。

- Address クラスが、db1 と db2 にインストールされる。
- 所有者が Smith である db1 と所有者が Jones である db2 の両方に、emps テーブルが作成されている。

この例では、db1 が現在のデータベースです。データベース間でジョインまたはメソッドを呼び出すことができます。次に例を示します。

- データベース間での join は次のようになります。

```
declare @count  int
select @count(*)
    from db2.Jones.emps, db1.Smith.emps
    where db2.Jones.emps.home_addr>>zip =
        db1.Smith.emps.home_addr>>zip
```

- データベース間でのメソッド呼び出しは次のようになります。

```
select db2.Jones.emps.home_addr>>toString( )
        from db2.Jones.emps
        where db2.Jones.emps.name = 'John Stone'
```

この例では、インスタンス値は転送されません。db2 に入っているインスタンスのフィールドとメソッドが、db1 内でルーチンによって参照されるだけです。したがって、データベース間のジョインとメソッド呼び出しでは、次のようになります。

- db1 に Address クラスが含まれている必要はありません。
- db1 に Address クラスが含まれている場合は、そのプロパティを db2 の Address クラスのプロパティとはまったく別のものにできます。

クラス間転送

あるデータベース内のクラスのインスタンスを、別のデータベース内の同じ名前のクラスのインスタンスに割り当てられます。ソース・データベース内のクラスによって作成されたインスタンスは、宣言された型が現在の (ターゲットの) データベース内のクラスであるカラムまたは変数に転送されます。

あるデータベース内のテーブルから別のデータベース内のテーブルに対して挿入や更新ができます。次に例を示します。

```
insert into db1.Smith.emps select * from
        db2.Jones.emps

update db1.Smith.emps
```

```
set home_addr = (select db2.Jones.emps.home_addr
    from db2.Jones.emps
where db2.Jones.emps.name =
    db1.Smith.emps.name)
```

あるデータベース内の変数から別のデータベースに対して挿入や更新ができます (次のフラグメントは、db2 のストアド・プロシージャ内にあります)。次に例を示します。

```
declare @home_addr Address
select @home_addr = new Address('94608', '222 Baker
        Street')
insert into db1.Janes.emps(name, home_addr)
        values ('Jone Stone', @home_addr)
```

この例では、データベース間でインスタンス値が転送されます。次の操作ができます。

- 2 つのローカル・データベース間でのインスタンスの転送
- ローカル・データベースとリモート・データベースの間でのインスタンスの転送
- SQL クライアントと Adaptive Server の間でのインスタンスの転送
- install 文と update 文または remove 文と update 文を使用したクラスの置換

クラス間転送では、Java の直列化はソースからターゲットに転送されます。ソース・データベースのクラスとターゲット・データベースのクラスとの互換性がない場合、Java 例外 InvalidClassException が発生します。

クラス間引数の引き渡し

異なるデータベース内の同じ名前のクラス間で引数を渡すことができます。クラス間引数を渡す場合、次のようになります。

- Java-SQL カラムは、そのカラムを含むデータベース内の指定された Java クラスのバージョンと対応している。
- (Transact-SQL の) Java-SQL 変数は、現在のデータベース内の指定された Java クラスのバージョンと対応している。
- クラス C の Java-SQL 中間結果は、その結果を返した Java メソッドと同じデータベース内のクラス C のバージョンと対応している。
- Java インスタンス値 *JI* がターゲットの変数またはカラムに割り当てられるか Java メソッドに渡される場合、*JI* は、それに対応するクラスから受信側ターゲットまたはメソッドに対応するクラスに変換される。

テンポラリ・データベースとワーク・データベース

Java クラスとデータベースに関するルールは、すべてテンポラリ・データベースと model データベースにも適用されます。

- テンポラリ・テーブルの Java-SQL カラムには、Java インスタンスのバイト文字列の直列化が入る。
- Java-SQL カラムは、テンポラリ・データベース内の指定されたクラスのバージョンと対応している。

Java クラスはテンポラリ・データベースにインストールできますが、インストールした Java クラスが存続するのはそのテンポラリ・データベースが存続する間だけです。

参照用の Java クラスをテンポラリ・データベースに提供する場合、Java クラスを model データベースにインストールするのが最も簡単な方法です。インストールした Java クラスは、その model データベースから抽出されるあらゆるテンポラリ・データベースに存在します。

Java クラス

この項では、この章で Adaptive Server の Java の説明に使用する簡単な Java クラスを示します。

以下に、Address クラスを示します。

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// All Rights Reserved
//
/**
* A simple class for address data, to illustrate using a Java class
* as a SQL datatype.
*/

public class  Address implements java.io.Serializable  {

/**
* The street data for the address.
* @serial A simple String value.
*/
        public String street;

/**
* The zipcode data for the address.
* @serial A simple String value.
*/
```

```
        String zip;

/** A default constructor.
*/
        public Address ( ) {
            street = "Unknown";
            zip = "None";
        }
/**
* A constructor with parameters
* @param S         a string with the street information
* @param Z         a string with the zipcode information
*/
        public Address (String S, String Z) {
            street = S;
            zip = Z;
        }
/**
* A method to return a display of the address data.
* @returns a string with a display version of the address data.
*/
        public String toString( ) {
            return  "Street= " + street +  "  ZIP= " + zip;
        }
/**
* A void method to remove leading blanks.
* This method uses the static method
* <code>Misc.stripLeadingBlanks</code>.
*/
        public void removeLeadingBlanks( ) {
        street = Misc.stripLeadingBlanks(street);
        zip = Misc.stripLeadingBlanks(street);
        }
}
```

以下に、Address2Line クラスを示します。このクラスは Address クラスのサブクラスです。

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// All Rights Reserved
//
/**
* A subclass of the Address class that adds a seond line of address data,
* <p>This is a simple subclass to illustrate using a Java subclass
* as a SQL datatype.
*/
public class Address2Line extends Address implements java.io.Serializable  {

/**
```

```
* The second line of street data for the address.
* @serial a simple String value
*/
        String line2;
/**
* A default constructor
*/
        public Address2Line ( ) {
            street = "Unknown";
            line2 = " ";
            zip = "None";
        }
/**
* A constructor with parameters.
* @param S a string with the street information
* @param L2 a string with the second line of address data
* @param Z a string with the zipcode information
*/
public Address2Line (String S, String L2, String Z) {
        street = S;
        line2 = L2;
        zip = Z;
}

/**
* A method to return a display of the address data
* @returns a string with a display version of the address data
*/

public String toString( ) {
        return "Street= " + street + " Line2= " + line2 +  " ZIP= " + zip;
}

/**
* A void method to remove leading blanks.
* This method uses the static method
* <code>Misc.stripLeadingBlanks</code>.
*/

        public void removeLeadingBlanks( ) {
            line2  = Misc.stripLeadingBlanks(line2);
            super.removeLeadingBlanks( );
        }
}
```

Misc クラスにはその他のルーチンのセットが入っています。

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// All Rights Reserved
```

```
//
/**
* A non-instantiable class with miscellaneous static methods
* that illustrate the use of Java methods in SQL.
*/

public class Misc{

/**
* The Misc class contains only static methods and cannot be instantiated.
*/

private Misc( ) { }

/**
* Removes leading blanks from a String
*/
        public static String stripLeadingBlanks(String s) {
            if (s == null) return null;
            for (int scan=0; scan<s.length( ); scan++)
                if (!java.lang.Character.isWhitespace(s.charAt(scan) ))
                    break;
                } else if (scan == s.length( )){
                    return  "";
                } else return s.substring(scan);
                }
            }
        }
        return  "";
}
/**
* Extracts the street number from an address line.
* e.g., Misc.getNumber("   123 Main Street") == 123
*        Misc.getNumber("   Main Street")  == 0
*        Misc.getNumber("") == 0
*        Misc.getNumber("   123   ") ==  123
*        Misc.getNumber("   Main 123 ")  == 0
* @param s a string assumed to have address data
* @return a string with the extracted street number
*/

        public static int getNumber (String s) {
            String stripped = stripLeadingBlanks(s);
            if (s==null) return -1;
            for(int right=0; right < stripped.length( ); right++){
                if (!java.lang.Character.isDigit(stripped.charAt(right))) {
                    break;
                } else if (right==0){
                    return 0;
                } else {
                    return java.lang.Integer.parseInt
```

```
                        (stripped.substring(0, right), 10);
                }
                }
                return -1;
            }

/**
* Extract the "street" from an address line.
* e.g.,  Misc.getStreet("   123 Main Street") ==  "Main Street"
*       Misc.getStreet("   Main Street")  ==   "Main Street"
*       Misc.getStreet("") ==  ""
*       Misc.getStreet("   123   ") ==  ""
*       Misc.getStreet("   Main 123 ")  == "Main 123"
* @param s a string assumed to have address data
* @return a string with the extracted street name
*/
        public static String getStreet(String s) {
            int left;
            if (s==null) return null;
            for (left=0; left<s.length( ); left++){
                if(java.lang.Character.isLetter(s.charAt(left))) {
                    break;
                } else if (left == s.length( )) {
                    return "";
                } else {
                    return s.substring(left);
                }
            }
            return "";
        }
}
```

第 5 章 JDBC を使用したデータ・アクセス

この章では、JDBC (Java Database Connectivity) を使用したデータへのアクセス・メソッドについて説明します。

トピック名	ページ
概要	65
JDBC の概念と用語	66
クライアント側 JDBC とサーバ側 JDBC の違い	66
パーミッション	67
JDBC を使用したデータへのアクセス	67
ネイティブ JDBC ドライバでのエラー処理	74
JDBCExamples クラス	76

概要

JDBC は Java アプリケーションに対して SQL インタフェースを提供します。Java からリレーショナル・データにアクセスする場合、JDBC 呼び出しを使用します。

Adaptive Server SQL インタフェースでは、次のどちらかの方法で JDBC を使用できます。

- クライアント側で JDBC を使用する － Java クライアント・アプリケーションは、Adaptive Server に対して JDBC 呼び出しが可能です。Sybase jConnect JDBC ドライバを使用します。
- サーバ側で JDBC を使用する － データベースにインストールされた Java クラスは、データベースに対して JDBC 呼び出しが可能です。Adaptive Server のネイティブ JDBC ドライバを使用します。

どちらの場合も、SQL オペレーションの実行に JDBC 呼び出しを使用するので、基本的に同じです。

この章では、サンプルのクラスとメソッドを紹介しながら、JDBC を使用した SQL オペレーションの実行方法について説明します。これらのクラスとメソッドは、一般的なガイドラインとして紹介するもので、テンプレートとして使用することを目的としているわけではありません。

JDBC の概念と用語

JDBC は、Java API および標準の Java クラス・ライブラリの一部で、Java アプリケーション開発用の基本機能を制御します。JDBC が提供する SQL 機能は、ODBC や動的 SQL の機能に似ています。

JDBC アプリケーションでの一般的なイベントの流れを次に示します。

1. *Connection* オブジェクトを生成する － DriverManager クラスの getConnection( ) 静的メソッドを呼び出し、*Connection* オブジェクトを生成します。これによりデータベース接続が確立されます。
2. *Statement* オブジェクトを生成する － *Connection* オブジェクトを使用して、*Statement* オブジェクトを生成します。
3. SQL 文を *Statement* オブジェクトに渡す － この文がクエリである場合、*ResultSet* オブジェクトが返されます。

   ResultSet オブジェクトには SQL 文から返されたデータが格納されていますが、一度に 1 つのローしか公開されません (カーソルの動きと同じです)。
4. 結果セットのローをループする － *ResultSet* オブジェクトの next( ) メソッドを呼び出して、次のことを行います。
   - 現在のロー (*ResultSet* オブジェクトによって公開されている結果セット内のロー) を、ローの 1 つ前に送ります。
   - ブール値 (true/false) が返され、前に送るローが存在するかどうかが示されます。
5. それぞれのローに対し、*ResultSet* オブジェクト内のカラムの値を検索する － getInt( )、getString( )、または類似のメソッドを使用して、カラムの名前か位置を指定します。

クライアント側 JDBC とサーバ側 JDBC の違い

クライアント側の JDBC とデータベース・サーバ側の JDBC の違いは、データベース環境との接続の確立方法にあります。

クライアント側またはサーバ側 JDBC を使用する場合、Drivermanager.getConnection() メソッドを呼び出してサーバへの接続を確立します。

- クライアント側 JDBC では、Sybase jConnect JDBC ドライバを使用し、サーバの ID を指定して Drivermanager.getConnection() メソッドを呼び出します。これにより、指定したサーバへの接続が確立されます。
- サーバ側 JDBC では、Adaptive Server のネイティブ JDBC ドライバを使用し、以下の値の 1 つを指定して Drivermanager.getConnection() メソッドを呼び出します。

- jdbc:default:connection
- jdbc:sybase:ase
- jdbc:default
- 空の文字列

これにより、現在のサーバへの接続が確立されます。getConnection() メソッドへの最初の呼び出しのみが、現在のサーバへの新しい接続を作成します。後続の呼び出しは、接続プロパティをまったく変更せずに、その接続のラッパを返します。

条件文を使用して URL を設定すると、クライアントとサーバの両方で実行できる JDBC クラスを記述できます。

パーミッション

- Java 実行パーミッション － JDBC 文を含んでいるクラスは、データベース内のすべての Java クラスと同じように、すべてのユーザがアクセスできます。Java メソッドにはプロシージャを実行するパーミッションを付与する grant execute 文に相当するものはなく、クラス名をその所有者名で修飾する必要はありません。
- SQL 実行パーミッション － Java クラスはそれらを実行する接続のパーミッションによって実行されます。この動作は、データベース所有者が付与したパーミッションによって実行されるストアド・プロシージャの動作とは異なります。

JDBC を使用したデータへのアクセス

この項では、JDBC を使用して SQL アプリケーションの代表的なオペレーションを実行する方法について説明します。サンプルは、JDBCExamples クラスから抽出されます。サンプルに関する説明は、「JDBCExamples クラス」(76 ページ) にあります。

JDBCExamples では、ユーザ・インタフェースの基本事項、および SQL オペレーションの内部コーディングの手法について紹介します。

JDBCExamples クラスの概要

これらの例をマシン上で実行するには、サーバに Address クラスをインストールし、それを jConnect クライアントの Java CLASSPATH に含めます。

JDBCExamples のメソッドは、jConnect クライアントまたは Adaptive Server のどちらからでも呼び出し可能です。

注意 ストアド・プロシージャの作成や削除は jConnect クライアントから行ってください。Adaptive Server のネイティブ・ドライバは、create procedure 文と drop procedure 文をサポートしていません。

JDBCExamples 静的メソッドは、次の SQL オペレーションを実行します。

- テーブル例 xmp の作成と削除。

```
create table xmp (id int, name varchar(50), home Address)
```

- ストアド・プロシージャのサンプル inoutproc の作成と削除。

```
create procedure inoutproc @id int, @newname varchar(50),
        @newhome Address, @oldname varchar(50) output, @oldhome
        Address output as

select @oldname = name, @oldhome = home from xmp
        where id=@id
update xmp set name=@newname, home = @newhome
        where id=@id
```

- テーブル xmp にローを挿入。
- テーブル xmp からローを選択。
- xmp のローを更新。
- ストアド・プロシージャ inoutproc の呼び出し。このプロシージャには入力パラメータと出力パラメータの両方があり、データ型は java.lang.String と Address です。

JDBCExamples は、xmp テーブル上、および inoutproc プロシージャ上でのみ動作します。

main( ) メソッドと serverMain( ) メソッド

JDBCExamples には 2 つの主要なメソッドがあります。

- main( ) − jConnect クライアントのコマンド･ラインから呼び出されます。
- serverMain( ) − main( ) と動作は同じですが、Adaptive Server から呼び出されます。

JDBCExamples クラスのすべての動作は、パラメータを使用し、これらのメソッドのどれか 1 つを呼び出して行われます。パラメータは実行される動作を示します。

main( ) の使用

次のようにして、jConnect コマンド・ラインから main( ) メソッドを呼び出すことができます。

```
java JDBCExamples
        ìserver-name:port-number?user=user-name&password=passwordî action
```

ユーザは、dsedit ツールを使用して、自分の interfaces ファイルから *server-name* と *port-number* を調べることができます。*user-name* はユーザー名、*password* はパスワードです。&password=*password* を省略すると、デフォルトは空のパスワードになります。以下に 2 つの例を示します。

```
"antibes:4000?user=smith&password=1x2x3"
"antibes:4000?user=sa"
```

パラメータは必ず引用符で囲みます。

action パラメータは、create table、create procedure、insert、select、update、call にすることができます。大文字と小文字は区別しません。

jConnect コマンド・ラインから JDBCExamples を呼び出して、テーブル xmp や、ストアド・プロシージャ inoutproc を以下のように作成することができます。

```
java JDBCExamples “antibes:4000?user=sa” CreateTable
java JDBCExamples “antibes:4000?user=sa” CreateProc
```

insert、select、update、call の動作に対して、以下のように JDBCExamples を呼び出すことができます。

```
java JDBCExamples “antibes:4000?user=sa” insert
java JDBCExamples “antibes:4000?user=sa” update
java JDBCExamples “antibes:4000?user=sa” call
java JDBCExamples “antibes:4000?user=sa” select
```

これらの呼び出しでは、"Action performed" というメッセージが表示されます。

テーブル xmp やストアド・プロシージャ inoutproc を削除するには、次のように入力します。

```
java JDBCExamples “antibes:4000?user=sa” droptable
java JDBCExamples “antibes:4000?user=sa” dropproc
```

serverMain( ) の使用

注意 サーバ側 JDBC ドライバは create procedure または drop procedure をサポートしていないので、これらの例を実行する前に、クライアント側で main( ) メソッドを呼び出して、テーブル xmp とストアド・プロシージャの例 inoutproc を作成します。「JDBCExamples クラスの概要」(68 ページ) を参照してください。

xmp と inoutproc を作成した後、serverMain( ) メソッドを以下のようにして呼び出すことができます。

```
select JDBCExamples.serverMain('insert')
go
select JDBCExamples.serverMain('select')
go
select JDBCExamples.serverMain('update')
go
select JDBCExamples.serverMain('call')
go
```

注意 サーバ側からの serverMain( ) の呼び出しでは、*server-name:port-number* パラメータは必要ありません。Adaptive Server が Adaptive Server 自体に接続するだけです。

JDBC 接続の取得：*Connecter( )* メソッド

main( ) と serverMain( ) はいずれも、JDBC *Connection* オブジェクトを返す connecter( ) メソッドを呼び出します。*Connection* オブジェクトは、後に続くすべての SQL オペレーションのベースになります。

main( ) と serverMain( ) が呼び出す connecter( ) にはパラメータがあり、そのパラメータでサーバ側環境またはクライアント側環境用の JDBC ドライバを指定します。返された *Connection* オブジェクトは、JDBCExamples クラスの他のメソッドに引数で渡されます。connecter( ) メソッドは他の接続動作から切り離されているので、JDBCExamples の他のメソッドはサーバ側環境またはクライアント側環境に依存しません。

その他のメソッドへの動作の送信：*doAction( )* メソッド

doAction( ) メソッドは、別のメソッドの 1 つに対して呼び出しを送信します。送信は、*action* パラメータに基づいて行われます。

doAction( ) メソッドには *Connection* というパラメータがあります。これは、ターゲットとなるメソッドに中継するだけのパラメータです。また、*locale* というパラメータもあります。これは、呼び出しがサーバ側かクライアント側かを示します。create procedure または drop procedure がサーバ側環境内で呼び出される場合、*Connection* は例外を引き起こします。

命令型 SQL オペレーションの実行：*doSQL( )* メソッド

doSQL( ) メソッドは、create table、create procedure、drop table、drop procedure などのように入力または出力のパラメータを必要としない SQL アクションを実行します。

doSQL( ) にはパラメータが 2 つあります。*Connection* オブジェクトと、実行する SQL 文です。doSQL( ) は、JDBC の *Statement* オブジェクトを作成し、それを使用して指定した SQL 文を実行します。

update 文の実行：*updater( )* メソッド

updater( ) メソッドは Transact-SQL update 文を実行します。update の動作は以下のとおりです。

```
String sql = "update xmp set name = ?, home = ? where id = ?";
```

これにより、指定した *id* 値を持つすべてのローの name カラムと home カラムが更新されます。

name カラムと home カラムの update 値、および id 値はパラメータ・マーカ (?) で指定されます。updater( ) は、文を準備して実行する前にこれらのパラメータ・マーカの値を提供します。値は、JDBC setString( )、setObject( )、および setInt( ) メソッドで以下のパラメータを使用して指定されます。

- 置き換えられる通常のパラメータ・マーカ
- 置き換えられる値

次に例を示します。

```
pstmt.setString(1, name);
pstmt.setObject(2, home);
pstmt.setInt(3, id);
```

これらの置き換えを行ったあとで、updater( ) は update 文を実行します。

updater( ) を簡単にするために、例の中では置き換える値を固定しています。通常は、アプリケーションが置き換える値を計算するか、パラメータとして値を取得します。

select 文の実行：*selecter( )* メソッド

selecter( ) メソッドは Transact-SQL select 文を実行します。

```
String sql = "select name, home from xmp where id=?";
```

where 句には、選択されるローのためのパラメータ・マーカ (?) があります。JDBC の setInt( ) メソッドを使用すると、SQL 文が準備された後に selecter( ) はパラメータ・マーカのための値を提供します。

```
PreparedStatement pstmt =
    con.prepareStatement(sql);
pstmt.setInt(1, id);
```

その後 selecter( ) が select 文を実行します。

```
ResultSet rs = pstmt.executeQuery();
```

注意 結果を返さない SQL 文に対しては、doSQL( ) や updater( ) を使用します。この 2 つのメソッドは SQL 文を executeUpdate( ) メソッドで実行します。

結果を返す SQL 文に対しては、executeQuery( ) メソッドを使用します。このメソッドは JDBC *ResultSet* オブジェクトを返します。

ResultSet オブジェクトは SQL カーソルに似ています。初期設定では、結果の最初のローの前に配置されています。next( ) メソッドが呼び出されるたびに *ResultSet* オブジェクトは次のローに進みます。これは、ローがなくなるまで繰り返されます。

selecter() は、*ResultSet* が正確に 1 つのローを持っていることを要求します。selecter( ) メソッドは次のメソッドを呼び出して、そこで *ResultSet* にローがなかったり、複数のローがあったりしないかをチェックします。

```
if (rs.next()) {
   name = rs.getString(1);
   home = (Address)rs.getObject(2);
   if (rs.next()) {
      throw new Exception("Error:  Select returned multiple rows");
   } else { // No action
   }
} else { throw new Exception("Error:  Select returned no rows");
}
```

上のコードでは、getString( ) メソッドと getObject( ) メソッドの呼び出しで、結果セットの最初のローのカラムを 2 つ取得します。式 "(Address)rs.getObject(2)" は 2 番目のカラムを Java オブジェクトとして取得し、それを Address クラスに強制的に送ります。返されたオブジェクトが Address でない場合、例外が発生します。

selecter( ) は単一ローを取得し、ローがないかどうか、または複数あるかどうかをチェックします。複数のローの *ResultSet* を処理するアプリケーションであれば、next( ) メソッドの呼び出しでループし、それぞれのローを単一ローとして処理します。

バッチ・モードでの実行

SQL 文のバッチを実行する場合は、必ず execute( ) メソッドを使用します。バッチ・モードで executeQuery( ) を使用すると、以下のようになります。

- バッチ・オペレーションが結果セットを返さない場合 (select 文を含まない場合)、バッチはエラーなしで実行される。
- バッチ・オペレーションが 1 つの結果セットを返す場合、結果を返す文に後続する文はすべて無視される。getXXX( ) が呼び出されて出力パラメータを取得する場合、残りの文が実行されて現在の結果セットが閉じます。
- バッチ・オペレーションが複数の結果セットを返す場合、例外が発生してオペレーションはアボートされる。

execute( ) を使用すると、あらゆる場合にバッチが完全に実行されます。

SQL ストアド・プロシージャの呼び出し：*caller( )* メソッド

caller( ) メソッドは、ストアド・プロシージャ inoutproc を呼び出します。

```
create proc inoutproc @id int, @newname varchar(50), @newhome Address,
            @oldname varchar(50) output,  @oldhome Address output as

        select @oldname = name, @oldhome = home from xmp where id=@id
        update xmp set name=@newname, home = @newhome where id=@id
```

このプロシージャには、3 つの入力パラメータ (*@id*、*@newname*、*@newhome*) と、2 つの出力パラメータ (*@oldname*、*@oldhome*) があります。caller( ) は、テーブル xmp の ID 値が *@id* のローの name カラムおよび home カラムを、値 *@newname* と値 *@newhome* にそれぞれ設定します。そして、出力パラメータ *@oldname* および *@oldhome* にこれらのカラムの前の値を返します。

inoutproc プロシージャは、JDBC 呼び出しの中の入力パラメータ、出力パラメータの提供方法を示します。

caller( ) は次の call 文を実行します。call 文は次のように準備します。

```
CallableStatement cs = con.prepareCall("{call inoutproc (?, ?, ?, ?, ?)}");
```

呼び出しのパラメータはすべてパラメータ・マーカ (?) で指定します。

caller( ) は、JDBC setInt( ) メソッド、setString( ) メソッド、setObject( ) メソッドを使って、入力パラメータに値を提供します。これらのメソッドは doSQL( ) メソッド、updatAction( ) メソッド、selecter( ) メソッドの中で使用されます。

```
cs.setInt(1, id);
cs.setString(2, newName);
cs.setObject(3, newHome);
```

これらの set メソッドは、出力パラメータには適しません。JDBC registerOutParameter( ) メソッドを使用して、出力パラメータの予測データ型を caller( ) が指定してから、call 文を実行します。

```
cs.registerOutParameter(4, java.sql.Types.VARCHAR);
cs.registerOutParameter(5, java.sql.Types.JAVA_OBJECT);
```

caller( ) は call 文を実行して、selecter( ) メソッドが使用したのと同じ getString( ) メソッドと getObject( ) メソッドを使用して出力値を取得します。

```
int res = cs.executeUpdate();
String oldName = cs.getString(4);
Address oldHome = (Address)cs.getObject(5);
```

ネイティブ JDBC ドライバでのエラー処理

Sybase では、java.sql.SQLException クラスと java.sql.SQLWarning クラスのすべてのメソッドをサポートし、実装しています。SQLException は、データベース・アクセス・エラーに関する情報を提供します。SQLWarning は、SQLException を拡張し、データベース・アクセス警告に関する情報を提供します。

Adaptive Server で発生したエラーには、重大度に応じて番号が付けられます。番号が小さいほど重大度が低く、番号が大きいほど重大度が高くなります。エラーは重大度に応じて、次のようにグループ化されます。

- 警告 (EX_INFO：重大度 10) － SQLWarnings に変換
- 例外 (重大度 11 ～ 18) － SQLExceptions に変換
- 致命的なエラー (重大度 19 ～ 24) － 致命的な SQLExceptions に変換

SQLExceptions は、JDBC、Adaptive Server、ネイティブ JDBC ドライバで発生する可能性があります。SqlException が発生すると、エラーの原因となった JDBC クエリがアボートされます。その後のシステムの動作は、エラーが検出された場所によって次のように異なります。

- エラーが Java 内で検出された場合 － "try" ブロックと後続の "catch" ブロックでエラーが処理されます。

  Adaptive Server では、JDBC ドライバ固有の拡張された SQLException エラー・メッセージがいくつか用意されています。これらはすべて、EX_USER (重大度 16) で、検出される可能性は常にあります。ドライバ固有の SQLWarning メッセージはありません。

- エラーが Java 内で検出されない場合 － Java VM が Adaptive Server に制御を返して Adaptive Server がエラーを検出し、未処理の SQLException エラーが発生します。

通常、raiserror コマンドをストアド・プロシージャで使用し、エラーを発生させてユーザ定義のエラー・メッセージを出力します。raiserror コマンドを呼び出すストアド・プロシージャが JDBC から実行されると、エラーは重大度 EX_USER の内部エラーとして扱われ、致命的でない SQLException が発生します。

注意 raiserror コマンドを使用して、拡張エラー・データにアクセスすることはできません。これは、with errordata 句が SQLException に実装されていないためです。

エラーによってトランザクションがアボートされた場合、結果は、Java メソッドが呼び出されたトランザクションのコンテキストによって次のように異なります。

- トランザザクションが複数の文を含む － トランザクションがアボートされ、制御がサーバに返されるため、トランザクション全体がロールバックされます。制御がサーバから戻るまで、JDBC ドライバはクエリの処理を停止します。
- トランザクションが単一の文を含む － トランザクションがアボートされ、それに含まれる SQL 文がロールバックされます。JDBC ドライバはクエリの処理を続行します。

以下の例では、さまざまな結果について説明します。次の文を含む Java メソッド jdbcTests.Errorexample() を考えてみます。

```
stmt.executeUpdate("delete from parts where partno = 0"); Q2
stmt.executeQuery("select 1/0"); Q3
stmt.executeUpdate("delete from parts where partno = 10"); Q4
```

複数の文を含むトランザクションには、次のような SQL コマンドが含まれます。

```
begin transaction
delete from parts where partno = 8 Q1
select JDBCTests.Errorexample()
```

この場合、アボートされたトランザクションによって次のようなアクションが発生します。

- ゼロ除算の例外が Q3 で発生する
- Q1 と Q2 からの変更がロールバックされる
- トランザクション全体がアボートされる

単一の文を含むトランザクションには、次のような SQL コマンドが含まれます。

```
set chained off
delete from parts where partno = 8 Q1
select JDBCTests.Errorexample()
```

この場合、次のようになります。

- ゼロ除算の例外が Q3 で発生する
- Q1 と Q2 からの変更がロールバックされない
- 例外は JDBCTests.Errorexample の "catch" ブロックと "try" ブロックで検出される
- Q4 で指定した削除は、Q3 と同じ "try" ブロックと "catch" ブロック内で処理されるため、実行されない
- 現在の "try" ブロックと "catch" ブロック以外の JDBC クエリは実行できる

JDBCExamples クラス

```
// An example class illustrating the use of JDBC facilities
// with the Java in Adaptive Server feature.
//
// The methods of this class perform a range of SQL operations.
// These methods can be invoked either from a Java client,
// using the main method, or from the SQL server, using
// the serverMain method.
//
import java.sql.*;             // JDBC
public class JDBCExamples {
{
```

main( ) メソッド

```
// The main method, to be called from a client-side command line
//
    public static void main(String args[]) {
        if (args.length!=2) {
            System.out.println("\n Usage:    "
                 + "java ExternalConnect server-name:port-number
                   action ");
            System.out.println("  The action is connect, createtable,
               " + "createproc, drop, "
               + "insert, select, update, or call \n" );
            return;
        }
        try{
           String server = args[0];
           String action = args[1].toLowerCase();
           Connection con = connecter(server);
           String workString = doAction( action, con, client);
           System.out.println("\n" + workString + "\n");
```

```
        }  catch (Exception e) {
           System.out.println("\n Exception: ");
           e.printStackTrace();
        }
    }
```

serverMain( ) メソッド

```
// A JDBCExamples method equivalent to 'main',
// to be called from SQL or Java in the server

    public static String serverMain(String action) {
        try {
           Connection con = connecter("default");
           String workString = doAction(action, con, server);
           return workString;
        }  catch ( Exception e ) {
             if (e.getMessage().equals(null)) {
               return "Exc: " + e.toString();
             } else {
               return "Exc - " + e.getMessage();
             }
        }
    }
```

connecter( ) メソッド

```
// A JDBCExamples method to get a connection.
// It can be called from the server with argument 'default',
// or from a client, with an argument that is the server name.

public static  Connection connecter(String server)
           throws Exception, SQLException, ClassNotFoundException  {

       String forName="";
       String url="";

       if (server=="default") {  // server connection to current server
           forName = "sybase.asejdbc.ASEDriver";
           url = "jdbc:default:connection";
       } else if (server!="default") {  //client connection to server
           forName= "com.sybase.jdbc.SybDriver";
           url = "jdbc:sybase:Tds:"+ server;
       }

       String user = "sa";
       String password = "";
```

```
        // Load the driver
        Class.forName(forName);
        // Get a connection
        Connection con = DriverManager.getConnection(url,
            user, password);
        return con;
    }
```

doAction( ) メソッド

```
// A JDBCExamples method to route to the 'action' to be performed

    public static String doAction(String action, Connection con,
                String locale)
            throws Exception {

        String createProcScript =
             " create proc inoutproc @id int, @newname varchar(50),
               @newhome Address, "
           + "   @oldname varchar(50) output,  @oldhome Address
               output as "
           + " select @oldname = name, @oldhome = home from xmp
               where id=@id "
           + " update xmp set name=@newname, home = @newhome
               where id=@id ";
        String createTableScript =
           " create  table xmp (id int, name varchar(50),
               home Address)" ;

        String dropTableScript = "drop   table xmp  ";
        String dropProcScript = "drop  proc inoutproc ";
           String insertScript = "insert  into xmp "
           + "values (1, 'Joe Smith', new Address('987 Shore',
           '12345'))";

        String workString = "Action (" + action + ) ;
        if (action.equals("connect")) {
             workString += "performed";
        } else if (action.equals("createtable")) {
             workString += doSQL(con, createTableScript );
        } else if (action.equals("createproc")) {
             if (locale.equals(server))  {
                 throw new exception (CreateProc cannot be performed
                   in the server);
             }  else {
                 workString += doSQL(con, createProcScript );
             }
        } else if (action.equals("droptable")) {
             workString += doSQL(con, dropTableScript );
        } else if (action.equals("dropproc")) {
```

```
            if (locale.equals(server))  {
                throw new exception (CreateProc cannot be performed
                  in the server);
            }  else {
                workString += doSQL(con, dropProcScript );
            }
        } else if (action.equals("insert")) {
            workString += doSQL(con, insertScript );
        } else if (action.equals("update")) {
            workString += updater(con);
        } else if (action.equals("select")) {
            workString += selecter(con);
        } else if (action.equals("call")) {
            workString += caller(con);
        } else { return "Invalid action:  " + action ;
        }
        return workString;
    }
```

doSQL( ) メソッド

```
// A JDBCExamples method to execute an SQL statement.

    public static String doSQL (Connection con, String action)
           throws Exception {

        Statement stmt = con.createStatement();
        int res = stmt.executeUpdate(action);
        return "performed";
 }
```

updater( ) メソッド

```
// A method that updates a certain row of the 'xmp' table.
// This method illustrates prepared statements and parameter markers.

    public static String updater(Connection con)
           throws Exception {

        String sql = "update xmp set name = ?, home = ? where id = ?";
        int id=1;
        Address home = new Address("123 Main", "98765");
        String name = "Sam Brown";
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setString(1, name);
        pstmt.setObject(2, home);
        pstmt.setInt(3, id);
        int res = pstmt.executeUpdate();
```

```
        return "performed";
    }
```

selecter( ) メソッド

```
//  A JDBCExamples method to retrieve a certain row
//  of the 'xmp' table.
//  This method illustrates prepared statements, parameter markers,
//  and result sets.

    public static String selecter(Connection con)
           throws Exception {

        String sql = "select name, home from xmp where id=?";
        int id=1;
        Address home = null;
        String name = "";
        String street = "";
        String zip = "";
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
           name = rs.getString(1);
           home = (Address)rs.getObject(2);
           if (rs.next()) {
              throw new Exception("Error:  Select returned
                    multiple rows");
           } else { // No action
           }
        } else { throw new Exception("Error:  Select returned no rows");
        }
        return "- Row with id=1:  name("+ name + )
                + " street(" + home.street + ) zip("+ home.zip + );
```

caller( ) メソッド

```
// A JDBCExamples method to call a stored procedure,
// passing input and output parameters of datatype String
 // and Address.
 // This method illustrates callable statements, parameter markers,
 // and result sets.

    public static String caller(Connection con)
           throws Exception {
        CallableStatement cs  = con.prepareCall("{call inoutproc
           (?, ?, ?, ?, ?)}");
        int id = 1;
```

```
        String newName = "Frank Farr";
        Address newHome = new Address("123 Farr Lane", "87654");
        cs.setInt(1, id);
        cs.setString(2, newName);
        cs.setObject(3, newHome);
        cs.registerOutParameter(4, java.sql.Types.VARCHAR);
        cs.registerOutParameter(5, java.sql.Types.JAVA_OBJECT);
        int res = cs.executeUpdate();
        String oldName = cs.getString(4);
        Address oldHome = (Address)cs.getObject(5);
        return "- Old values of row with id=1: name("+oldName+ )
           street(" + oldHome.street + ")  zip("+ oldHome.zip + );
    }
}
```

第 6 章

SQLJ 関数およびストアド・プロシージャ

この章では、Java メソッドを SQL 名を使用してラッピングする方法、そしてラッピングされた Java メソッドを Adaptive Server の関数やストアド・プロシージャとして使用する方法について説明します。

名前	ページ
概要	83
Java メソッドの Adaptive Server での起動	86
Sybase Central を使用した SQLJ 関数およびプロシージャの管理	88
SQLJ ユーザ定義関数	89
SQLJ ストアド・プロシージャ	94
SQLJ 関数およびプロシージャに関する情報の表示	104
詳細情報	104
SQLJ と Sybase の実装上の相違点	109
SQLJExamples クラス	111

概要

Java 静的メソッドを SQL ラッパで囲むと、Transact-SQL ストアド・プロシージャまたは組み込み関数とまったく同様に使用できます。この機能により、次のようなことが実現できます。

- Java メソッドで、呼び出し元の環境に出力パラメータと結果セットを返す。
- ANSI SQLJ 標準仕様 Part 1 に準拠する。
- 従来の SQL 構文、メタデータ、パーミッション機能を利用する。
- 既存の Java メソッドをサーバ上、クライアント上、SQLJ に準拠するサードパーティのデータベース上の SQLJ プロシージャや関数として使用する。

❖ SQLJ ストアド・プロシージャまたは関数を作成する

SQLJ ストアド・プロシージャまたは関数を作成して実行するには、次の手順に従います。

1　Java メソッドを作成し、コンパイルします。installjava ユーティリティを使用して、作成したメソッドのクラスをデータベースにインストールします。

　Java メソッドの Adaptive Server での作成、コンパイル、インストールの詳細については、「第 3 章 データベースでの Java の準備と管理」を参照してください。

2　SQLJ の create procedure 文または create function 文を使用して、そのメソッドに使用する SQL 名を定義します。

3　作成したプロシージャまたは関数を実行します。この章で説明する例では JDBC メソッド呼び出しまたは isql を使用します。また、Embedded SQL や ODBC を使用して実行することもできます。

SQLJ Part 1 仕様に対する準拠

Adaptive Server SQLJ のストアド・プロシージャと関数は標準仕様の SQLJ Part 1 に準拠しており、Java で SQL を使用できます。SQLJ 標準の詳細については、「規格」(6 ページ) を参照してください。

Adaptive Server は SQLJ Part 1 仕様で規定している機能をほとんどサポートしていますが、いくつか相違点があります。サポートしていない機能は表 6-3 (110 ページ) に、一部サポートしている機能は表 6-4 (110 ページ) に、それぞれ示してあります。Sybase 定義の機能 (標準規格の実装依存部分) は表 6-5 (110 ページ) に示してあります。

Sybase 独自の実装が SQLJ 仕様と異なる場合は、SQLJ 規格をサポートします。たとえば、非 Java Sybase SQL ストアド・プロシージャは in と inout の 2 つのパラメータ・モードをサポートしています。一方、SQLJ 標準は in、out、inout の 3 つのパラメータ・モードをサポートします。この場合、Sybase の SQLJ ストアド・プロシージャ作成構文は、この 3 つのパラメータ・モードをすべてサポートします。

一般的な問題

この項では、SQLJ 関数やストアド・プロシージャに見られる一般的な問題や制約事項について説明します。

セキュリティとパーミッション

SQLJ ストアド・プロシージャと SQLJ 関数では、それぞれ異なるセキュリティ・モデルが提供されています。

SQLJ 関数とユーザ定義関数 (UDF) (「SQL での Java メソッドの呼び出し」(37 ページ) を参照) では、同じセキュリティ・モデルを使用します。public に対しては、任意の UDF または SQLJ 関数を実行するパーミッションが暗黙に与えられています。JDBC を通して SQL クエリを実行する関数を呼び出すと、関数の呼び出し元がデータのアクセス・パーミッションに関するチェックを受けます。したがって、ユーザ A がテーブル t1 にアクセスする関数を呼び出した場合、ユーザ A に t1 に対する select パーミッションがないと、クエリは失敗します。

SQLJ ストアド・プロシージャは、Transact-SQL ストアド・プロシージャと同じセキュリティ・モデルを使用します。SQLJ または Transact-SQL ストアド・プロシージャを実行する場合、ユーザに明示的なパーミッションが与えられている必要があります。SQLJ プロシージャが JDBC をとおして SQL クエリを実行する場合、暗黙のパーミッションが適用されます。このセキュリティ・モデルにより、ストアド・プロシージャの所有者が、そのプロシージャが参照するすべての SQL オブジェクトの所有者である場合、その所有者はそのプロシージャの実行パーミッションを他のユーザに与えることができます。実行パーミッションを持つユーザは、オブジェクトにアクセスするパーミッションがなくても、そのストアド・プロシージャのあらゆる SQL クエリを実行できます。

通常、JVM の設定および実行後は、データベースから Java クラスにアクセスできるどのユーザでも JVM を実行できます。ただし、次の操作は制限されます。

- 作成とジョインに必要な操作を除くスレッド操作
- exit( ) や abort( ) など、サーバに影響を及ぼすシステム操作
- クラス・ローダ階層の変更
- インストールされた SecurityManager の上書き

ストアド・プロシージャのセキュリティの詳細については、『セキュリティ管理ガイド』を参照してください。

SQLJ の例

この章で使用する例では、次のようなカラムがある SQL テーブル sales_emps を使用します。

- name － 従業員名
- id － 従業員の ID 番号
- state － 従業員の所在地域 (州)
- sales － 従業員の売上高
- jobcode － 従業員の職責コード

このテーブルの定義は次のとおりです。

```
create table sales_emps
    (name varchar(50), id char(5),
    state char(20), sales decimal (6,2),
    jobcode integer null)
```

例として使用するクラス名は SQLJExamples です。次のメソッドが用意されています。

- region() － 地域番号としてアメリカの州コードをマップします。このメソッドでは SQL は使いません。
- correctStates() － SQL update コマンドを実行して、state コード綴りの誤りを訂正します。入力パラメータとして古い綴りと新しい綴りを指定します。
- bestTwoEmps() － sales の記録から売上高上位 2 名を算出して、出力パラメータにその値を返します。
- SQLJExamplesorderedEmps() － 指定された従業員のローを sales カラムの値で並べ替えた SQL 結果セットを生成して、それをクライアントに返します。
- job() － 職責コードの整数値に対応する文字列値を返します。

各メソッドのソース・コードについては、「SQLJExamples クラス」(111 ページ) を参照してください。

Java メソッドの Adaptive Server での起動

Java メソッドの Adaptive Server での起動方法には、次の 2 とおりがあります。

- Java メソッドを SQL から直接起動する。この方法によるメソッドの起動方法の詳細については、「第 4 章 SQL での Java クラスの使用」を参照してください。
- メソッド名に Transact-SQL エイリアスを持つ SQLJ ストアド・プロシージャや関数を使用して、Java メソッドを間接的に起動する。この章では、この方法による Java メソッドの起動方法について説明します。

いずれの方法を使用する場合も、まず Java メソッドを作成し、installjava ユーティリティを使用して Adaptive Server データベースにインストールする必要があります。詳細については、「第 3 章 データベースでの Java の準備と管理」を参照してください。

Java 名による Java メソッドの直接起動

SQL に含まれている Java メソッドは、完全修飾されている Java 名を使用して参照することで起動できます。これにより、インスタンス・メソッドに対してはインスタンスが、静的メソッドに対してはインスタンスまたはクラスが参照されます。

静的メソッドは、呼び出し元の環境に対して値を返すユーザ定義関数 (UDF) として使用できます。Java 静的メソッドは、ストアド・プロシージャ、トリガ、where 句、select 文など、組み込み SQL 関数を使用できるあらゆる場面で UDF として使用できます。

Java メソッドをその名前を使用して呼び出す場合、呼び出し元の環境に出力パラメータまたは結果セットを返すメソッドは使用できません。メソッドによって、JDBC 接続経由で受信したデータを操作することはできますが、呼び出し元の環境に返すことができるのは定義されている 1 つの戻り値だけです。

UDF 関数のデータベース間起動は使用できません。

Java メソッドのこのような使用法の詳細については、「第 4 章 SQL での Java クラスの使用」 を参照してください。

Java メソッドの SQLJ を使用した間接起動

Java メソッドを SQLJ 関数またはストアド・プロシージャとして起動できます。Java メソッドを SQL ラッパ内にラッピングすると、次のことが可能になります。

- SQLJ ストアド・プロシージャを使用して、呼び出し元の環境に対して結果セットや出力パラメータを返すことができます。
- SQL　メタデータ機能を利用できます。たとえば、データベースに格納されているすべてのストアド・プロシージャと関数をリスト表示できます。
- SQLJ ではメソッドに SQL 名を与えることができ、これにより標準の SQL パーミッションを使用してメソッドの起動を保護できます。
- Sybase SQLJ は業界標準である SQLJ Part 1 仕様に準拠しており、Sybase SQLJ プロシージャおよび関数は仕様に準拠している非 Sybase 環境でも使用できます。
- SQLJ 関数や SQLJ ストアド・プロシージャはデータベース間で起動できます。
- Adaptive Server は SQLJ ルーチンが作成されるとデータ型のマッピングをチェックするので、ルーチンの実行時にデータ型のマッピングに注意する必要はありません。

SQLJ ルーチンでは静的メソッドを参照してください。インスタンス・メソッドは参照できません。

この章では、Java メソッドを SQLJ ストアド・プロシージャおよび関数として使用する方法について説明します。

Sybase Central を使用した SQLJ 関数およびプロシージャの管理

SQLJ 関数およびプロシージャは、isql を使用してコマンド・ラインから管理したり、Sybase Central にアクセスする Adaptive Server プラグインから管理したりできます。Adaptive Server プラグインからは次の処理が可能です。

- SQLJ 関数またはプロシージャの作成
- SQLJ 関数またはプロシージャの実行
- SQLJ 関数またはプロシージャのプロパティの表示および変更
- SQLJ 関数またはプロシージャの削除
- SQLJ 関数またはプロシージャの依存性の表示
- SQLJ プロシージャのパーミッションの作成

次に SQLJ ルーチンのプロパティの作成と表示の手順について説明します。ルーチンのプロパティ・シートを使用して、依存性の表示やパーミッションの作成および表示が可能です。

❖ SQLJ 関数／プロシージャを作成する

まず、Java メソッドを作成し、コンパイルします。installjava を使用して、作成したメソッドのクラスをデータベースにインストールします。その後で、次の手順に従います。

1. Adaptive Server プラグインを起動し、Adaptive Server に接続します。
2. ルーチンの作成先になるデータベースをダブルクリックします。
3. [SQLJ プロシージャ] または [SQLJ 関数] フォルダを開きます。
4. [新しい Java ストアド・プロシージャ] または [新しい Java 関数の追加] アイコンをダブルクリックします。
5. [新しい Java ストアド・プロシージャ] または [新しい Java 関数の追加] ウィザードを使用して、SQLJ プロシージャまたは関数を作成します。

   ウィザード操作を終了すると、Adaptive Server プラグインは作成した SQLJ ルーチンを編集画面に表示します。ルーチンはここで編集または実行します。

❖ SQLJ 関数またはプロシージャのプロパティを表示する

1. Adaptive Server プラグインを起動し、Adaptive Server に接続します。
2. ルーチンが保存されているデータベースをダブルクリックします。
3. [SQLJ プロシージャ] または [SQLJ 関数] フォルダを開きます。
4. 関数アイコンまたはプロシージャ・アイコンを強調表示します。
5. [ファイル] − [プロパティ] を選択します。

SQLJ ユーザ定義関数

create function コマンドは、Java メソッドの SQLJ 関数名とシグニチャを指定するのに使用します。SQLJ 関数を使用して、SQL を読み出したり、SQL を修正したり、参照されているメソッドに記述されている値を返したりします。

create function の SQLJ 構文は次のとおりです。

```
create function [owner].sql_function_name
        ([sql_parameter_name sql_datatype
            [( length)| (precision[, scale])]
        [, sql_parameter_name sql_datatype
            [( length )    | ( precision[, scale]) ]]
        ...])
    returns sql_datatype
        [( length)| (precision[, scale])]
    [modifies sql data]
    [returns null on null input |
        called on null input]
    [deterministic | not deterministic]
    [exportable]
    language java
    parameter style java
    external name 'java_method_name
        [([java_datatype[ {, java_datatype }
        ...]])]'
```

SQLJ 関数を作成する場合、次の点に注意します。

- 「**SQL** 関数のシグニチャ」とは、各関数のパラメータの SQL データ型 *sql_datatype* のことです。
- ANSI 標準に準拠するには、パラメータ名の前にアットマーク (@) を使用しないでください。

  アットマーク (@) は、パラメータ・バインド用に内部的に使用しています。sp_help を使用して SQLJ ストアド・プロシージャに関する情報を印刷すると、このように使用されているアットマーク (@) を見ることができます。
- SQLJ 関数を作成する場合、*sql_parameter_name* 情報と *sql_datatype* 情報を囲むカッコは必ず使用してください。これらの情報を使用しない場合も必ずカッコを付けます。

  次に例を示します。

  ```
  create function sqlj_fc()
      language java
      parameter style java
  external name 'SQLJExamples.method'
  ```

- modifies sql data 句は、そのメソッドが SQL 処理の起動、SQL データの読み出し、SQL データの修正を行うことを示します。これはデフォルトの値です。SQLJ Part 1 標準との構文的な互換性を維持する必要がない場合、この句を使用する必要はありません。
- returns null on null input と called on null input は、関数呼び出しの引数が null だった場合の Adaptive Server による処理を規定します。returns null on null input を指定すると、ランタイム時に null の引数が 1 つでもあった場合、関数の戻り値は null に設定され、関数本体は起動されません。デフォルトは called on null input です。この場合、引数に null があっても関数は起動されます。

  関数の呼び出しと引数の値が null の場合の詳細については、「関数呼び出し時の null への対処」(93 ページ) を参照してください。
- キーワード deterministic または not deterministic は構文上は使用できますが、Adaptive Server はこれらを使用しません。これらのキーワードは、SQLJ Part 1 標準との構文上の互換性を維持するためにサポートされています。
- exportable キーワード句は、Sybase OmniConnect™ 機能を使用してリモート・サーバ上で実行する関数であることを示すのに使用します。元になる関数やメソッドはリモート・サーバ上にインストールされている必要があります。
- language java 句と parameter style java 句は、参照されているメソッドが Java で記述されており、パラメータが Java パラメータであることを示します。SQLJ 関数を作成する場合は、これらのフレーズを必ず使用してください。
- external name 句は、そのルーチンが SQL では記述されていないことを示すと同時に、Java メソッド、クラス、パッケージ名 (該当する場合) を示します。
- Java メソッドのシグニチャは、各メソッドのパラメータの Java データ型 *java_datatype* を示します。Java メソッドのシグニチャはオプションです。これが指定されていない場合、Adaptive Server は Java メソッドのシグニチャを SQL 関数のシグニチャから導出します。

  この規則はあらゆるデータ型変換に適用されるので、このメソッド・シグニチャを使用することをおすすめします。「Java と SQL データ型のマッピング」(104 ページ) を参照してください。
- 1 つの Java メソッドを create function を使用して異なる SQL 名で定義しつつ、同じ使い方をすることができます。

Java メソッドの記述

SQLJ 関数を作成する前に、それが参照する Java メソッドを記述し、そのメソッドのクラスをコンパイルし、データベースにインストールしておく必要があります。

次の例に示されている SQLJExamples.region() では、州コードを地域番号にマップし、その番号を返しています。

```
public static int region(String s)
        throws SQLException {
    s = s.trim();
    if (s.equals "MN") || s.equals("VT") ||
        s.equals("NH") ) return 1;
    if (s.equals("FL") || s.equals("GA") ||
        s.equals("AL") ) return 2;
    if (s.equals("CA") || s.equals("AZ") ||
        s.equals("NV") ) return 3;
    else throw new SQLException
        ("Invalid state code", "X2001");

}
```

SQLJ 関数の作成

メソッドの記述とインストールが完了したら、SQLJ 関数を作成します。次に例を示します。

```
create function region_of(state char(20))
        returns integer
language java parameter style java
external name
        'SQLJExamples.region(java.lang.String)'
```

SQLJ の create function 文は、入力パラメータが `(state char(20))` で、戻り値が integer であることを示しています。この SQL 関数のシグニチャは char(20) です。Java メソッドのシグニチャは java.lang.String です。

関数の呼び出し

SQLJ 関数は、組み込み関数と同様に直接呼び出すことができます。次に例を示します。

```
select name, region_of(state) as region
    from sales_emps
where region_of(state)=3
```

注意　Adaptive Server における関数の検索順は次のとおりです。

1　組み込み関数

2　SQLJ 関数

3　直接呼び出しが可能な Java-SQL 関数

引数の null 値の処理

Java クラス・データ型と Java プリミティブ・データ型には、引数に null 値があった場合の処理方法が異なります。

- クラスである **Java** オブジェクト･データ型 － たとえば java.lang.Integer、java.lang.String、java.lang.byte[]、java.sql.Timestamp は実体のある値も null 参照値も保持できます。
- **Java** プリミティブ・データ型 － たとえば boolean、byte、short、int は null 値を表すことはできません。保持できるのは非 null 値だけです。

Java メソッドが起動されたとき、引数として SQL null 値が渡された Java パラメータのデータ型が Java クラスだった場合、これは Java null 参照値として渡されます。一方、SQL null 値が渡された Java パラメータが Java プリミティブ・データ型だった場合、Java プリミティブ・データ型は null 値を扱わないため例外が発生します。

一般的に、Java メソッドのパラメータのデータ型はクラスにしておきます。こうすれば、例外を発生させることなく null を処理できます。null 値に対処できない Java パラメータを使用する Java 関数を記述する場合は、次のいずれかの方法を採用します。

- SQLJ 関数を作成する場合に returns null on null input 句を使用する。
- SQLJ 関数の起動に case などの条件文を使用して null 値の有無を確認し、null 値がない場合にのみ SQLJ 関数を起動する。

null 値の対処は SQLJ 関数の作成時でも呼び出し時でも可能です。次の 2 つの項で、この両方のシナリオについて説明します。そこでは、次のメソッドを参照します。

```
public static String job(int jc)
        throws SQLException {
if (jc==1) return "Admin";
else if (jc==2) return "Sales";
else if (jc==3) return "Clerk";
else return "unknown jobcode";
}
```

関数作成時の null への対処

null 値があり得ることがあらかじめ分かっている場合、returns null on null input 句を使用して関数を作成します。次に例を示します。

```
create function job_of(jc integer)
    returns varchar(20)
returns null on null input
language java parameter style java
external name 'SQLJExamples.job(int)'
```

これで、job_of を次のように呼び出すことができます。

```
select name, job_of(jobcode)
    from sales_emp
where job_of(jobcode) <> "Admin"
```

SQL システムが sales_emps ローに対する関数呼び出し job_of(jobcode) を評価する場合、このローの jobcode カラムの値が null であると、この関数の値は null に設定され、Java メソッド SQLJExamples.job が実際に呼び出されることはありません。jobcode カラムの値が null でないローに対しては、関数は通常どおり起動されます。

このように、returns null on null input 句を使用して作成された SQLJ 関数の引数に null が含まれていた場合、関数呼び出しの結果は null になり、関数は実際には起動されません。

注意 returns null on null input 句を使用して SQLJ 関数を作成した場合、returns null on null input 句は null に対処できるパラメータを含むすべての関数パラメータに適用されます。

---

called on null input 句を使用した場合 (デフォルト)、null に対処できないパラメータに引数として null が渡ると例外が発生します。

関数呼び出し時の null への対処

null を扱えないパラメータに null 値が渡った場合の対処として、関数呼び出し時に条件文を使用する方法があります。次の例では case 式を使用しています。

```
select name,
    case when jobcode is not null
       then job_of(jobcode)
       else null end
from sales_emps where
    case when jobcode is not null
       then job_of(jobcode)
       else null end <> "Admin"
```

この例では、関数 job_of でデフォルト句 called on null input を使用していると仮定しています。

SQLJ 関数名の削除

Java メソッドの SQLJ 関数名を削除するには、drop function コマンドを使用します。たとえば、次のように入力します。

```
drop function region_of
```

これにより、region_of 関数名とそこからの SQLJExamples.region メソッドへの参照が削除されます。drop function は参照先の Java メソッドまたはクラスには影響を与えません。

構文と使用法の詳細については、『リファレンス・マニュアル：ビルディング・ブロック』を参照してください。

SQLJ ストアド・プロシージャ

Java-SQL 機能を使用して、Java クラスをデータベースにインストールし、メソッドをクライアントや SQL システムで起動できます。また Java 静的 (クラス) メソッドを別の方法、すなわち SQLJ ストアド・プロシージャとしても起動できます。

SQLJ ストアド・プロシージャ：

- 結果セットか出力パラメータまたはその両方をクライアントに返すことができます。
- Transact-SQL ストアド・プロシージャとまったく同様に実行できます。
- ODBC、isql、または JDBC を使用しているクライアントから呼び出すことができます。
- サーバ上の他のストアド・プロシージャまたはネイティブな Adaptive Server JDBC から呼び出すことができます。

エンド ユーザはプロシージャの呼び出し元が SQLJ ストアド・プロシージャなのか Transact-SQL ストアド・プロシージャなのかを意識する必要はありません。両方とも同様に起動されます。

create procedure の SQLJ 構文は次のとおりです。

```
create procedure [owner.]sql_procedure_name
    ([[ in | out | inout ] sql_parameter_name
        sql_datatype [( length) |
        (precision[, scale])]
    [, [ in | out | inout ] sql_parameter_name
        sql_datatype [( length) |
        (precision[, scale]) ]]
    ...])
    [modifies sql data]
    [dynamic result sets integer] [deterministic | not
deterministic]
    language java
    parameter style java
    external name 'java_method_name
        [([java_datatype[, java_datatype
    ...]])]'
```

注意　ANSI 標準に準拠するため、SQLJ create procedure コマンド構文は Sybase Transact-SQL ストアド・プロシージャの作成に使用する構文とは異なります。

このコマンドの各キーワードやオプションの詳細については、『リファレンス・マニュアル：コマンド』を参照してください。

SQLJ ストアド・プロシージャを作成する場合、次の点に注意します。

- 「**SQL** プロシージャ・シグニチャ」とは、各プロシージャのパラメータの SQL データ型 *sql_datatype* です。
- SQLJ ストアド・プロシージャを作成する場合、アットマーク (@) をパラメータ名の前に使用しないでください。この規則は ANSI 標準に準拠しています。

  アットマーク (@) は、パラメータ・バインド用に内部的に使用しています。sp_help を使用して SQLJ ストアド・プロシージャに関する情報を印刷すると、このように使用されているアットマーク (@) を見ることができます。

- SQLJ ストアド・プロシージャを作成する場合、*sql_parameter_name* 情報と *sql_datatype* 情報を囲むカッコは必ず使用してください。これらの情報を使用しない場合も必ずカッコを付けます。

  次に例を示します。

  ```
  create procedure sqlj_sproc ()
      language java
      parameter style java
  external name "SQLJExamples.method1"
  ```

- キーワード modifies sql data は、そのメソッドが SQL 処理の起動、SQL データの読み出し、SQL データの修正を行うことを示すのに使用します。これはデフォルトの値です。
- 結果セットを呼び出し元の環境に返す場合、dynamic result sets *integer* オプションを使用してください。この *integer* 変数を使用して、予期される結果セットの最大数を指定します。
- キーワード deterministic または not deterministic を使用して、SQLJ 標準との互換性を確保できます。ただし、Adaptive Server はこのオプションを使用しません。
- キーワード language java parameter と style java を使用してください。これにより、この外部ルーチンが Java で記述されており、この外部ルーチンに渡される引数のランタイム時の規則に Java の規則が適用されることを Adaptive Server に通知します。
- external name 句は、その外部ルーチンが Java で記述されていることを示します。また、Java メソッド、クラス、パッケージ名 (該当する場合) を示します。

- Java メソッドのシグニチャは、各メソッドのパラメータの Java データ型 *java_datatype* を示します。Java メソッドのシグニチャはオプションです。これが指定されていない場合、Adaptive Server は Java メソッドのシグニチャを SQL プロシージャのシグニチャから導出します。

  この規則はあらゆるデータ型変換に適用されるので、このメソッド・シグニチャを使用することをおすすめします。詳細については、「Java と SQL データ型のマッピング」(104 ページ) を参照してください。

- 1 つの Java メソッドを create procedure を使用して異なる SQL 名で定義して、同じように使用できます。

SQL データの変更

SQLJ ストアド・プロシージャを使用して、データベースの情報を変更できます。SQLJ プロシージャが参照するメソッドは、次のいずれかである必要があります。

- void 型のメソッド、または
- 戻り値のデータ型が int のメソッド (戻り値のデータ型としての int は SQLJ 標準に対する Sybase の拡張です)。

Java メソッドの記述

メソッド SQLJExamples.correctStates() は、SQL update 文を実行して州コードの綴りを修正します。入力パラメータで、古い綴りと新しい綴りを指定します。correctStates() は void メソッドで、呼び出し側に値を返しません。

```
public static void correctStates(String oldSpelling, String
newSpelling) throws SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
        Class.forName("sybase.asejdbc.ASEDriver");
        conn = DriverManager.getConnection
            ("jdbc:default:connection");
    }
    catch (Exception e) {
        System.err.println(e.getMessage() +
            “:error in connection”);
    }
    try {
        pstmt = conn.prepareStatement
            ("UPDATE sales_emps SET state = ?
            WHERE state = ?");
        pstmt.set.String(1, newSpelling);
        pstmt.set.String(2, oldSpelling);
        pstmt.executeUpdate();
    }
    catch (SQLException e) {
```

```
            System.err.println(“SQLException:"+
            e.getErrorCode() + e.getMessage());
        }
        return;
    }
```

ストアド・プロシージャの作成

SQL 名を使用して Java メソッドを呼び出すことができるようにするには、SQLJ create procedure コマンドを使用して、そのメソッドの SQL 名を作成する必要があります。modifies sql data 句はオプションです。

```
create procedure correct_states(old char(20),
        not_old char(20))
    modifies sql data
    language java parameter style java
    external name
        'SQLJExamples.correctStates
        (java.lang.String, java.lang.String)'
```

correct_states プロシージャの SQL プロシージャ・シグニチャは char(20), char(20) です。Java メソッド・シグニチャは、java.lang.String, java.lang.String です。

ストアド・プロシージャの呼び出し

SQLJ プロシージャは、Transact-SQL プロシージャとまったく同様に実行できます。次の例では、isql からプロシージャを実行します。

```
execute correct_states 'GEO', 'GA'
```

入出力パラメータの使用

Java メソッドは、出力パラメータをサポートしません。ただし、Java メソッドを SQL でラッピングした場合は、Sybase SQLJ 機能を利用して、SQLJ ストアド・プロシージャの入力、出力、入出力のパラメータを使用できます。

SQLJ プロシージャを作成する場合は、各パラメータのモードとして in、out、または inout を指定します。

- 入力パラメータには、in キーワードを使用します。in はデフォルトです。Adaptive Server は、パラメータ・モードを指定していない場合は、入力パラメータと見なされます。
- 出力パラメータには、out キーワードを使用します。
- 参照先の Java メソッドとの間で値の受け渡しが可能なパラメータには、inout キーワードを使用します。

注意　Transact-SQL ストアド・プロシージャを作成する場合に使用するのは、in キーワードと out キーワードだけです。out キーワードは、SQLJ の inout キーワードに対応します。詳細については、『リファレンス・マニュアル：コマンド』の create procedure のリファレンス・ページを参照してください。

出力パラメータを使用する SQLJ ストアド・プロシージャを作成する場合、次の点に注意してください。

- SQLJ ストアド・プロシージャを作成するときは、out オプションか、inout オプションを使用して出力パラメータを定義します。
- これらのパラメータを、Java メソッド内で Java 配列として宣言します。SQLJ は、配列をメソッドの出力パラメータ値のコンテナとして使用します。

  たとえば、関数の呼び出し側に Integer のパラメータから値を返すようにする場合は、メソッド内でパラメータのデータ型を Integer[ ] (Integer の配列) として指定する必要があります。

  out パラメータや inout パラメータ用の配列オブジェクトは、システムが暗黙で作成します。この配列の大きさは 1 です。入力値があれば、配列の最初の要素 (唯一の要素) に格納され、その後で Java メソッドが呼び出されます。Java メソッドが戻ると、最初の要素は除去されて出力変数に割り当てられます。通常、この要素には、呼び出されたメソッドにより新しい値が割り当てられます。

次の例は、Java メソッド bestTwoEmps() と、このメソッドを参照するストアド・プロシージャ best2 を使用する出力パラメータの使用例です。

Java メソッドの記述

SQLJExamples.bestTwoEmps() メソッドは、営業成績の上位 2 名の 名前、ID、地域、売上高を返します。最初の 8 つのパラメータは出力パラメータで、格納用の配列が必要です。9 番目のパラメータは入力パラメータで、配列は必要ありません。

```
public static void bestTwoEmps(String[] n1,
    String[] id1, int[] r1,
    BigDecimal[] s1, String[] n2,
    String[] id2, int[] r2, BigDecimal[] s2,
    int regionParm) throws SQLException {

n1[0] = "****";
id1[0] = "";
r1[0] = 0;
s1[0] = new BigDecimal(0):
n2[0] = "****",
id2[0] = "";
r2[0] = 0;
s2[0] = new BigDecimal(0);

try {
    Connection conn = DriverManager.getConnection
        ("jdbc:default:connection");
    java.sql.PreparedStatement stmt =
        conn.prepareStatement("SELECT name, id,"
        + "region_of(state) as region, sales FROM"
        + "sales_emps WHERE"
        + "region_of(state)>? AND"
```

```
            + "sales IS NOT NULL ORDER BY sales DESC");
        stmt.setInteger(1, regionParm);
        ResultSet r = stmt.executeQuery();

        if(r.next()) {
            n1[0] = r.getString("name");
            id1[0] = r.getString("id");
            r1[0] = r.getInt("region");
            s1[0] = r.getBigDecimaL("sales");
        }
        else return;

        if(r.next()) {
            n2[0] = r.getString("name");
            id2[0] = r.getString("id");
            r2[0] = r.getInt("region");
            s2[0] = r.getBigDecimal("sales");
        }
        else return;
    }
    catch (SQLException e) {
        System.err.println("SQLException: "+
            e.getErrorCode() + e.getMessage());
        }
    }
```

SQLJ プロシージャの作成

bestTwoEmps メソッドの SQL 名を生成します。最初の 8 つは出力パラメータで、9 番目は入力パラメータです。

```
create procedure best2
    (out n1 varchar(50), out id1 varchar(5),
    out s1 decimal(6,2), out r1 integer,
    out n2 varchar(50), out id2 varchar(50),
    out r2 integer, out s2 decimal(6,2),
    in region integer)
    language java
    parameter style java
    external name
        'SQLJExamples.bestTwoEmps (java.lang.String,
        java.lang.String, int, java.math.BigDecimal,
        java.lang.String, java.lang.String, int,
        java.math.BigDecimal, int)'
```

best2 の SQL プロシージャ・シグニチャは varchar(20), varchar(5), decimal (6,2) ... です。Java メソッド・シグニチャは、String, String, int, BigDecimal, ... です。

プロシージャの呼び出し

メソッドをデータベースにインストールし、そのメソッドを参照する SQLJ プロシージャを作成した後、その SQLJ プロシージャを呼び出すことができます。

SQL システムは、実行時に次のように動作します。

1 SQLJ プロシージャが呼び出されると、out パラメータと inout パラメータに必要な配列を生成します。

2 SQLJ プロシージャから戻るときに、パラメータ配列の内容をターゲットとなる out 変数と inout 変数にコピーします。

 次の例では、best2 プロシージャを isql から呼び出します。region 入力パラメータの値は、地域番号です。

```
declare @n1 varchar(50), @id1 varchar(5),
@s1 decimal (6,2), @r1 integer, @n2 varchar(50),
@id2 varchar(50), @r2 integer, @s2 decimal(6,2),
@region integer
select @region = 3
execute best2 @n1 out, @id1 out, @s1 out, @r1 out,
@n2 out, @id2 out, @r2 out, @s2 out, @region
```

注意 Adaptive Server は、SQLJ ストアド・プロシージャを Transact-SQL ストアド・プロシージャとまったく同様に呼び出します。このため、isql などの Java 以外のクライアントを使用する場合は、パラメータ名の先頭にアットマーク (@) を付ける必要があります。

結果セットのリターン

SQL 結果セットは、呼び出し側の環境に受け渡される一連の SQL ローです。

Transact-SQL ストアド・プロシージャが 1 つまたは複数の結果セットを返す場合、これらの結果セットはプロシージャ・コールからの暗黙の出力です。つまり、これらはパラメータや戻り値として明示的に宣言されません。

Java メソッドは Java 結果セットを返すことができますが、この場合は明示的に宣言されたメソッド値として返します。

SQL スタイルの結果セットを Java メソッドから返すには、まず Java メソッドを SQLJ ストアド・プロシージャでラッピングする必要があります。Java メソッドを SQLJ ストアド・プロシージャとして呼び出すと、その結果セットは、Java メソッドから Java 結果セットとして返され、サーバによって SQL 結果セットに変換されます。

SQL スタイルの結果セットを返す SQLJ プロシージャとして呼び出される Java メソッドを記述する場合は、メソッドが返すことのできる結果セットごとに、メソッドの追加パラメータを指定する必要があります。このパラメータは、要素が 1 つしかない Java ResultSet クラス配列です。

この項では、メソッドの記述、SQLJ ストアド・プロシージャの作成、メソッドの呼び出しに関する基本的な手順について説明します。結果セットのリターンの詳細については、「Java メソッド・シグニチャの明示的な指定と暗黙の指定」(106 ページ) を参照してください。

Java メソッドの記述

次に示すメソッド SQLJExamples.orderedEmps は SQL を呼び出し、ResultSet パラメータを取り込み、JDBC 呼び出しを使用して接続のセキュリティを確保しながら文を開きます。

```
public static void orderedEmps
        (int regionParm, ResultSet[] rs) throws
        SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;

    try {
        Class.forName
            ("sybase.asejdbc.ASEDriver");
        Connection conn =
            DriverManager.getConnection
            ("jdbc:default:connection");
    }
    catch (Exception e) {
        System.err.println(e.getMessage()
            + ":error in connection");
    }

    try {
        java.sql.PreparedStatement
            stmt = conn.prepareStatement
            ("SELECT name, region_of(state)"
            "as region, sales FROM sales_emps"
            "WHERE region_of(state) > ? AND"
            "sales IS NOT NULL"
            "ORDER BY sales DESC");
        stmt.setInt(1, regionParm);
        rs[0] = stmt.executeQuery();
        return;
    }
    catch (SQLException e)
        System.err.println("SQLException:"
        + e.getErrorCode() + e.getMessage());
    }
    return;
}
```

orderedEmps は結果セットを 1 つ返します。また、複数の結果セットを返すメソッドを記述することもできます。この場合、返される各結果セットで以下の処理が必要になります。

- メソッド・シグニチャに個別の ResultSet 配列パラメータを含める。
- 各結果セットに対し Statement オブジェクトを作成する。
- 各結果セットに ResultSet 配列の第 1 要素を割り当てる。

Adaptive Server は常に、各 Statement オブジェクトで現在オープンしている ResultSet オブジェクトを返します。結果セットを返す Java メソッドを作成する場合は、次の点に注意してください。

- クライアントに返される各結果セット用に Statement オブジェクトを作成する。
- ResultSet オブジェクトと Statement オブジェクトは明示的にクローズしない。Adaptive Server が自動的に閉じます。

注意 Adaptive Server では、ResultSet オブジェクトと Statement オブジェクトは、それらの影響を受ける結果セットの処理が完了してクライアントに返されるまでは、ガーベジ・コレクションによって閉じられません。

- 結果セットのローに Java next() メソッドの呼び出しによってフェッチされたものがあると、クライアントに返される結果セットのローはその残りだけになる。

SQLJ ストアド・プロシージャの作成

結果セットを返す SQLJ ストアド・プロシージャを作成する場合、返される結果セットの最大数を指定する必要があります。次の例では、ranked_emps プロシージャは結果セットを 1 つだけ返します。

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps(int,
       ResultSet[]'
```

ranked_emps が create procedure で指定されている数以上の結果セットを生成すると、警告が表示され、プロシージャは指定されている数の結果セットだけを返します。記述のとおり、ranked_emps SQLJ ストアド・プロシージャと一致する Java メソッドは 1 つだけです。

注意 結果セットに関連するメソッド・シグニチャを導出するときには、メソッドのオーバロードに一定の制限が適用されます。詳細については、「Java と SQL データ型のマッピング」(104 ページ) を参照してください。

プロシージャの呼び出し

メソッドのクラスをデータベースにインストールし、そのメソッドを参照する SQLJ ストアド・プロシージャを作成すると、そのプロシージャを呼び出すことができるようになります。この呼び出しは、SQL 結果セットを処理するどのメカニズムを使用しても記述できます。

たとえば、ranked_emps プロシージャを JDBC で呼び出す場合は、次のように記述します。

```
java.sql.CallableStatement stmt =
    conn.prepareCall("{call ranked_emps(?)}");
stmt.setInt(1,3);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    String name = rs.getString(1);
    int.region = rs.getInt(2);
    BigDecimal sales = rs.get.BigDecimal(3);
    System.out.print("Name = " + name);
    System.out.print("Region = "+ region);
    System.out.print("Sales = "+ sales);
    System.out.printIn():
}
```

ranked_emps プロシージャは、create procedure 文で宣言されているパラメータだけを提供します。SQL システムは、ResultSet パラメータの空の配列を提供して Java メソッドを呼び出し、呼び出された Java メソッドがこの配列パラメータに出力結果セットを割り当てます。呼び出された Java メソッドが終了すると、SQL システムは出力配列要素に格納されている結果セットを SQL 結果セットとして返します。

注意　テンポラリ・テーブルの結果セットを返すことができるのは、jConnect などの外部 JDBC ドライバを使用している場合のみです。Adaptive Server ネイティブの JDBC ドライバは、この場合には使用できません。

SQLJ ストアド・プロシージャ名の削除

Java メソッドの SQLJ ストアド・プロシージャ名を削除するには、drop procedure コマンドを使用します。たとえば、次のように入力します。

```
drop procedure correct_states
```

これにより、correct_states プロシージャ名とそこからの SQLJExamples.correctStates メソッドへの参照が削除されます。drop procedure は参照先の Java メソッドまたはクラスには影響を与えません。

SQLJ 関数およびプロシージャに関する情報の表示

SQLJ ルーチンに関する情報を提供するシステム・ストアド・プロシージャには、次のものがあります。

- sp_depends は、SQLJ ルーチンが参照するデータベース・オブジェクトとデータベース・オブジェクトが参照する SQLJ ルーチンをリストします。
- sp_help は SQLJ ルーチンの各パラメータの名前、タイプ、長さ、精度、位取り、パラメータ順、パラメータ・モード、戻り値のタイプをリストします。
- sp_helpjava は、データベースにインストールされている Java クラスと JAR の情報をリストします。depends パラメータには、SQLJ create function または SQLJ create procedure 文の external name 句で指定されているクラスの依存関係がリストされます。
- sp_helprotect は、SQLJ ストアド・プロシージャや SQLJ 関数のパーミッションをレポートします。

ここにあげた各システム・プロシージャの構文や使用法の詳細については、『リファレンス・マニュアル：プロシージャ』を参照してください。

詳細情報

この項では、SQLJ の上級者向けのトピックについて詳しく説明します。

Java と SQL データ型のマッピング

Java メソッドを参照するストアド・プロシージャまたは関数を作成する場合、値が SQL 環境向けから Java 環境向けに、またその逆に変換されるときに、入出力パラメータや結果セットのデータ型の不一致が起こらないよう注意する必要があります。ここで適用されるマッピング規則は、JDBC 標準を実装した場合と同じです。次に、表 6-1 (106 ページ) と合わせて説明します。

各 SQL パラメータと対応する Java パラメータはマッピング可能である必要があります。SQL データ型と Java データ型は次の 3 つの方法でマッピングできます。

- SQL データ型とプリミティブ Java データ型は、表 6-1 に該当する記述がある場合、直接マッピング可能です。
- SQL データ型と非プリミティブ Java データ型は、表 6-1 に該当する記述がある場合、オブジェクト・レベルでマッピング可能です。

- SQL 抽象データ型 (ADT) と非プリミティブ Java データ型は、クラスまたはインタフェースが一致している場合、ADT レベルでマッピング可能です。
- SQL データ型と Java データ型は、Java データ型が配列で、SQL データ型が Java データ型に対して直接マッピング可能、オブジェクト・レベルでマッピング可能、または ADT レベルでマッピング可能な場合は、出力レベルでマッピング可能です。たとえば、character と String[ ] は出力レベルでマッピング可能です。
- Java データ型は、それが結果セット指向クラス java.sql.ResultSet の配列である場合は、結果セット・レベルでマッピング可能です。

一般に、Java メソッドが SQL にマッピング可能なのは、Java メソッドの各パラメータが SQL にマッピング可能で、結果セット・パラメータが結果セット・レベルでマッピング可能で、戻り値の型がマッピング可能 (関数の場合) または void か int (プロシージャの場合) の場合です。

SQLJ ストアド・プロシージャにおける戻り値の型としての int のサポートは、SQLJ Part 1 標準を拡張したものです。

表 6-1: 直接およびオブジェクト・レベルでマッピング可能な SQL と Java データ型

SQL データ型	対応する Java データ型	
	直接マッピング可能	オブジェクト・レベルでマッピング可能
char/unichar		java.lang.String
nchar		java.lang.String
varchar/univarchar		java.lang.String
nvarchar		java.lang.String
text		java.lang.String
numeric		java.math.BigDecimal
decimal		java.math.BigDecimal
money		java.math.BigDecimal
smallmoney		java.math.BigDecimal
bit	boolean	Boolean
tinyint	byte	Integer
smallint	short	Integer
integer	int	Integer
bigint	long	java.math.BigInteger
unsigned smallint	int	Integer
unsigned int	long	Integer
unsigned bigint		java.math.BigInteger
real	float	Float
float	double	Double
double precision	double	Double
binary		byte[ ]
varbinary		byte[ ]
datetime		java.sql.Timestamp
smalldatetime		java.sql.Timestamp
date		java.sql.Date
time		java.sql.Time

Java メソッド・シグニチャの明示的な指定と暗黙の指定

SQLJ 関数やストアド・プロシージャを作成する場合、通常 Java メソッド・シグニチャを指定します。また、表 6-1 とともに先に説明した標準の JDBC データ型対応規則に基づいて、Adaptive Server にルーチンの SQL シグニチャから Java メソッド・シグニチャを導出させることもできます。

Java メソッド・シグニチャを使用すると、データ型変換が常に指定されたとおりに処理されます。Java メソッド・シグニチャの使用をおすすめします。

Adaptive Server にメソッド・シグニチャを導出させることができるデータ型は次のとおりです。

- 直接マッピング可能
- ADT レベルでマッピング可能

- 出力レベルでマッピング可能
- 結果セット・レベルでマッピング可能

たとえば、Adaptive Server で correct_states のメソッド・シグニチャを導出させる場合、create procedure 文は次のように記述します。

```
create procedure correct_states(old char(20),
        not_old char(20))
    modifies sql data
    language java parameter style java
    external name 'SQLJExamples.correctStates'
```

Adaptive Server は java.lang.String と java.lang.String の Java メソッド・シグニチャを導出します。明示的に Java メソッド・シグニチャを追加する場合、create procedure 文は次のようになります。

```
create procedure correct_states(old char(20),
        not_old char(20))
    modifies sql data
    language java parameter style java
    external name 'SQLJExamples.correctStates
        (java.lang.String, java.lang.String)'
```

オブジェクト・レベルでマッピング可能なデータ型に対しては、Java メソッド・シグニチャを必ず明示的に指定してください。そうしないと、Adaptive Server は直接マッピング可能なプリミティブなデータ型を導出します。

たとえば、SQLJExamples.job メソッドには、int 型のパラメータがあります (「引数の null 値の処理」(92 ページ) を参照)。このメソッドを参照する関数を作成するときは、Adaptive Server が int の Java シグニチャを導出するので、指定する必要はありません。

ここで、SQLJExamples.job のパラメータが Java Integer だった場合を考えてみます。これは、オブジェクト・レベルでマッピング可能なデータ型です。次に例を示します。

```
public class SQLJExamples {
    public static String job(Integer jc)
           throws SQLException ...
```

これを参照する関数を作成する場合は、次のように Java メソッド・シグニチャを指定する必要があります。

```
create function job_of(jc integer)
...
external name
    'SQLJExamples.job(java.lang.Integer)'
```

結果セットのリターンとメソッドのオーバロード

結果セットを返す SQLJ ストアド・プロシージャを作成する場合は、返すことができる結果セットの最大数を指定する必要があります。

Java メソッド・シグニチャを指定すると、Adaptive Server は、そのメソッド名とシグニチャに一致するメソッドを 1 つ探します。次に例を示します。

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps
        (int, java.sql.ResultSet[])'
```

この場合、Adaptive Server は通常の Java オーバロード規則に従ってパラメータ型を解決します。

ここで、次のように Java メソッド・シグニチャを指定しなかった場合を考えてみます。

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps'
```

メソッドが 2 つ存在し、片方のシグニチャが int, RS[ ]、もう片方のシグニチャが int, RS[ ], RS[ ] だった場合、Application Server はこの 2 つを区別できず、プロシージャは失敗します。結果セットを返すときに Adaptive Server に Java メソッド・シグニチャを導出させる場合、導出に使用される条件に適合するメソッドが 1 つしかないことを確認してください。

注意 指定されている動的結果セット数は、返すことができる結果セットの最大数にだけ影響します。メソッドのオーバロードには影響しません。

シグニチャの妥当性の保証

インストール済みのクラスが変更された場合、Adaptive Server はそのクラスを参照する SQLJ プロシージャまたは関数が起動されたときに、変更があったメソッドのシグニチャの妥当性をチェックします。変更されたメソッドのシグニチャが依然として有効な場合、SQLJ ルーチンは正しく実行されます。

コマンド main メソッドの使用

Java クライアントでは一般的に、クラスのコマンド main メソッド上で Java 仮想マシン (Java VM) を実行することで Java アプリケーションを起動します。たとえば、JDBCExamples クラスには、main メソッドがあります。コマンド・ラインからクラスを次のように実行した場合、実行されるのはコマンド main メソッドです。

```
java JDBCExamples
```

注意 SQLJ create function 文のなかで Java main コマンドを参照することはできません。

Java main メソッドを SQLJ create procedure 文のなかで参照する場合、コマンド main メソッドには次のように Java メソッド・シグニチャ `String[]` が必要です。

```
public static void main(java.lang.String[]) {
...
}
```

Java メソッド・シグニチャを create procedure 文に指定する場合、(java.lang.String[]) のように記述してください。Java メソッド・シグニチャが指定されていない場合は、(java.lang.String[]) であると想定されます。

SQL プロシージャ・シグニチャでパラメータを使用する場合、パラメータの型は char、unichar、varchar、または univarchar にする必要があります。これらは、実行時に java.lang.String の Java 配列として渡されます。

SQLJ プロシージャに渡す各引数は、char、unichar、varchar、univarchar、またはリテラル文字列にする必要があります。これは、main メソッドを java.lang.String 配列の要素として渡す必要があるからです。dynamic result sets 句は、main プロシージャの作成には使用できません。

SQLJ と Sybase の実装上の相違点

この項では、SQLJ Part 1 標準仕様と、Sybase 独自の SQLJ ストアド・プロシージャおよび関数の実装との相違点について説明します。

表 6-2 に、Adaptive Server で拡張した SQLJ 実装上のポイントを示します。

表 6-2: Sybase での拡張

カテゴリ	SQLJ 標準	Sybase での実装
create procedure コマンド	値を返さない Java メソッドのみサポート。メソッドの戻り値の型は必ず void。	整数値を返す Java メソッドをサポート。create procedure から参照されるメソッドの戻り値の型としては void または int が可能。
create procedure コマンドと create function コマンド	create procedure または create function のパラメータ・リストの SQL データ型のみをサポート。	SQL データ型と非プリミティブ Java データ型を抽象データ型 (ADT) としてサポート。
SQLJ 関数と SQLJ プロシージャの呼び出し	SQL の SQLJ データ型への暗黙の変換はサポートしない。	SQL の SQLJ データ型への暗黙の変換をサポート。
SQLJ 関数	SQLJ 関数のリモート・サーバ上での実行を許可しない。	SQLJ 関数の Sybase OmniConnect 機能を利用したリモート・サーバ上での実行を許可。
drop procedure コマンドと drop function コマンド	コマンド名 drop procedure または drop function をフルスペルで入力。	フルスペル、または省略形 drop proc および drop func での入力が可能。

表 6-3 に、Sybase 実装でサポートされていない SQLJ 標準機能を示します。

表 6-3: サポートされていない SQLJ 機能

SQLJ カテゴリー	SQLJ 標準	Sybase での実装
create function コマンド	複数の SQLJ 関数で同じ SQL 名を指定することが可能。	ストアド・プロシージャと関数では、いずれも固有の名前が必要。
ユーティリティ	sqlj.install_jar、sqlj.replace_jar、sqlj.remove_jar など、JAR ファイルのインストール、置換、削除を行うユーティリティをサポート。	同じ機能を実行する installjava ユーティリティと remove java Transact-SQL コマンドをサポート。

表 6-4 に、Sybase での実装で部分的にサポートされている SQLJ 標準機能を示します。

表 6-4: 一部サポートされている SQLJ 機能

SQLJ カテゴリー	SQLJ 標準	Sybase での実装
create procedure コマンドと create function コマンド	名前が同じクラスでも、異なる JAR ファイルに存在する場合、クラスを同一のデータベースにインストール可能。	同一データベース内に存在するためには固有の名前が必要。
create procedure コマンドと create function コマンド	Java メソッドで実行できる SQL 操作を指定するためのキーワード no *sql*、contains *sql*、reads *sql* data、modifies *sql* data をサポート。	modifies sql data のみサポート。
create procedure コマンド	java.*sql*.ResultSet と SQL/OLB 反復子の宣言をサポート。	java.sql.ResultSet のみサポート。
drop procedure コマンドと drop function コマンド	キーワード restrict をサポート。このキーワードは、プロシージャや関数を削除する前に、それらを呼び出すすべての SQL オブジェクト (テーブル、ビュー、ルーチン) を削除する場合に必要。	restrict キーワードとその機能はサポートしない。

表 6-5 に、Sybase 実装における SQLJ 実装依存機能を示します。

表 *6-5:* 実装依存の *SQLJ* 機能

SQLJ カテゴリー	SQLJ 標準	Sybase での実装
create procedure コマンドと create function コマンド	deterministic | not deterministic キーワードをサポート。これらのキーワードで、プロシージャや関数が out パラメータ、inout パラメータ、関数の結果で常に同じ値を返すようにするかどうかを指定。	deterministic | not deterministic は、構文としてはサポートするが、機能はサポートしない。
create procedure コマンドと create function コマンド	SQL シグニチャと Java メソッド・シグニチャのマッピングの検証は、create コマンドの実行時、またはプロシージャや関数の起動時に実行。検証の実行タイミングは実装依存。	参照先クラスに変更があった場合、すべての検証は create コマンドの実行時に実行。これにより実行速度を高速化。

SQLJ カテゴリー	SQLJ 標準	Sybase での実装
create procedure コマンドと create function コマンド	create procedure コマンドまたは create function コマンドは、配備記述子ファイル内で、または SQL DDL 文として指定可能。どちら (または両方) をサポートするかは実装依存。	create procedure と create function は、配備記述子外の SQL DDL 文としてサポート。
SQLJ ルーチンの起動	Java メソッドで SQL 文を実行する場合、例外はいずれも Java メソッド内で Exception.sqlException サブクラスの Java 例外として発生。例外による影響は実装依存。	Adaptive Server JDBC 規則に従う。
SQLJ ルーチンの起動	SQL 名を使用して呼び出された Java メソッドが、そのプロシージャまたは関数を作成したユーザに与えられている権限で実行されるか、またはそのプロシージャまたは関数を起動したユーザに与えられている権限で実行されるかは、実装依存。	SQLJ プロシージャと関数は、SQL ストアド・プロシージャと Java-SQL 関数のセキュリティ機能をそれぞれ継承。
drop procedure コマンドと drop function コマンド	drop procedure コマンドまたは drop function コマンドは、配備記述子ファイル内で、または SQL DDL 文として指定可能。どちら (または両方) をサポートするかは実装依存。	create procedure と create function は、配備記述子外の SQL DDL 文としてサポート。

SQLJExamples クラス

SQLJ ストアド・プロシージャと関数の説明に使用した SQLJExamples クラスを次に示します。

```
import java.lang.*;
import java.sql.*;
import java.math.*;

static String _url = “jdbc:default:connection”;

public class SQLExamples {

    public static int region(String s)
          throws SQLException {
        s = s.trim();
        if (s.equals("MN") || s.equals("VT") ||
            s.equals("NH") ) return 1;
        if (s.equals("FL") || s.equals("GA") ||
            s.equals("AL") ) return 2;
        if (s.equals("CA") || s.equals("AZ") ||
            s.equals("NV") ) return 3;
        else throw new SQLException
```

```
        ("Invalid state code", "X2001");
}
public static void correctStates
    (String oldSpelling, String newSpelling)
    throws SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
        Class.forName
            ("sybase.asejdbc.ASEDriver");
        conn = DriverManager.getConnection(_url);
    }
    catch (Exception e) {
        System.err.println(e.getMessage() +
            ":error in connection");
    }
    try {
        pstmt = conn.prepareStatement
            ("UPDATE sales_emps SET state = ?
            WHERE state = ?");
        pstmt.setString(1, newSpelling);
        pstmt.setString(2, oldSpelling);
        pstmt.executeUpdate();
    }
    catch (SQLException e) {
        System.err.println("SQLException: "+
        e.getErrorCode() + e.getMessage());
    }

}
public static String job(int jc)
        throws SQLException {
    if (jc==1) return "Admin";
    else if (jc==2) return "Sales";
    else if (jc==3) return "Clerk";
    else return "unknown jobcode";
}
public static String job(int jc)
        throws SQLException {
if (jc==1) return "Admin";
else if (jc==2) return "Sales";
else if (jc==3) return "Clerk";
else return "unknown jobcode";
}

public static void bestTwoEmps(String[] n1,
    String[] id1, int[] r1,
    BigDecimal[] s1, String[] n2,
    String[] id2, int[] r2, BigDecimal[] s2,
    int regionParm) throws SQLException {
```

```
n1[0] = "****";
id1[0] = "";
r1[0] = 0;
s1[0] = new BigDecimal(0):
n2[0] = "****";
id2[0] = "";
r2[0] = 0;
s2[0] = new BigDecimal(0);

try {
    Connection conn = DriverManager.getConnection
        ("jdbc:default:connection");
    java.sql.PreparedStatement stmt =
        conn.prepareStatement("SELECT name, id,"
        + "region_of(state) as region, sales FROM"
        + "sales_emps WHERE"
        + "region_of(state)>? AND"
        + "sales IS NOT NULL ORDER BY sales DESC");
    stmt.setInteger(1, regionParm);
    ResultSet r = stmt.executeQuery();

    if(r.next()) {
        n1[0] = r.getString("name");
        id1[0] = r.getString("id");
        r1[0] = r.getInt("region");
        s1[0] = r.getBigDecimal("sales");
    }
    else return;

    if(r.next()) {
        n2[0] = r.getString("name");
        id2[0] = r.getString("id");
        r2[0] = r.getInt("region");
        s2[0] = r.getBigDecimal("sales");
    }
    else return;
}
catch (SQLException e) {
    System.err.println("SQLException: "+
        e.getErrorCode() + e.getMessage());
    }
}

public static void orderedEmps
        (int regionParm, ResultSet[] rs) throws
        SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;
```

```
            try {
                Class.forName
                    ("sybase.asejdbc.ASEDriver");
                Connection conn =
                    DriverManager.getConnection
                    ("jdbc:default:connection");
            }
            catch (Exception e) {
                System.err.println(e.getMessage()
                    + ":error in connection");
            }

            try {
                java.sql.PreparedStatement
                    stmt = conn.prepareStatement
                    ("SELECT name, region_of(state)"
                    "as region, sales FROM sales_emps"
                    "WHERE region_of(state) > ? AND"
                    "sales IS NOT NULL"
                    "ORDER BY sales DESC");
                stmt.setInt(1, regionParm);
                rs[0] = stmt.executeQuery();
                return;
            }
            catch (SQLException e) {
                System.err.println("SQLException:"
                + e.getErrorCode() + e.getMessage());
            }
            return;
        }   return;
        }
}
```

第 7 章 データベースでの Java のデバッグ

トピック名	ページ
サポートされている Java デバッガ	115
Java のデバッグの設定	116

すべての PCA/JVM には、Java Platform Debugger Architecture (JPDA) に対するサポートが組み込まれています。JPDA を使用すると、Adaptive Server で実行する Java コードをデバッグできます。JPDA は次のもので構成されます。

- デバッグを制御するユーザ・インタフェースであるデバッガ
- デバッグ対象のクラスを実行する JVM、およびその JVM へのアクセスを提供するデバッグ・エージェント
- デバッグ・エージェントとデバッガの間の通信チャネル

JPDA を使用すると、コマンド・ラインからデバッガ・アプリケーション内の JVM を起動して、またはリモートから実行中の JVM のデバッグ・エージェントにデバッガを付加して、Java クラスをデバッグできます。ユーザはサーバの JVM コマンド・ラインにはアクセスできないため、Adaptive Server データベース内のすべての Java のデバッガはリモートで実行されます。

サポートされている Java デバッガ

すべての JDK には、開発ツール・パッケージ内に基本的なコマンド・ライン・デバッガ "jdb" の実装があります。また、Java の開発とデバッグに統合開発環境 (IDE) を使用することもできます。たとえば、Sun Java Studio、IBM WebSphere Studio、JBuilder、Eclipse などです。さらに、JSwat など、スタンドアロンの JPDA デバッガもあります。

IDE またはスタンドアロンのデバッグ・ツールを使用する場合、固有の JDK の要件についてはベンダから提供されたマニュアルを確認してください。

注意 jdb デバッガは JRE の配布には含まれていません。jdb を使用するには、JDK をインストールして、jdb デバッガにアクセスできるようにする必要があります。

Java のデバッグの設定

IDE、スタンドアロン・デバッガ、jdb デバッガのいずれを使用する場合でも、次の操作を実行する必要があります。

1. デバッグをサポートするサーバを設定する。
2. リモート・デバッガを JVM デバッグ・エージェントに付加する。

デバッグをサポートするサーバの設定

ユーザ提供、またはデフォルトのポート番号を使用して、JVM に対するデバッグ・エージェントを起動します。デバッグの有効化、ポート番号の選択、および JVM を即時に中断するかどうかの指定には、次の設定パラメータと共に sp_jreconfig を使用します。

- pca_jvm_java_dbg_agent_port － デバッグを有効化または無効化し、および JVM のデバッグ・エージェントが受信するポート番号を設定します。このパラメータを有効にすると、JVM は起動時に、リモート・デバッガを付加できるようにデバッグ・エージェントを実行します。デフォルトでは、デバッグ・エージェントはポート 8000 を受信します。デバッグ・エージェントを有効化してデフォルト・ポートを使用してデバッグするようにするには、次のように入力します。

```
sp_jreconfig "enable", "pca_jvm_java_dbg_agent_port"
```

  別のポートを使用するには、JVM を起動する前にポート番号を変更します。一度 JVM が起動してデバッグ・エージェントを実行すると、JVM がシャットダウンするまでデバッグ・エージェントはそのポートで受信します。デバッグを有効化してデバッグ・エージェントを受信するポートを変更するには、次のように入力します。

```
sp_jreconfig "update", "pca_jvm_java_dbg_agent_port", new_port_number
```

- pca_jvm_java_dbg_agent_suspend － デバッグ・エージェントを実行しているとき、JVM が起動時に中断するかどうかを制御します。デフォルトでは、pca_jvm_java_dbg_agent_suspend は無効です。

  pca_jvm_java_dbg_agent_suspend が有効な場合、デバッガが付加されて JVM が再起動されるまで Java メソッドは実行できません。JVM を中断すると、クラスがロードされる前の JVM の初期化を検査できます。通常は、ユーザ・クラスのデバッグには JVM の中断は必要ありません。

  pca_jvm_java_dbg_agent_suspend を有効にするには、次のように入力します。

```
sp_jreconfig "enable", "pca_jvm_java_dbg_agent_suspend"
```

注意 pca_jvm_java_dbg_agent_suspend の使用には注意が必要です。pca_jvm_java_dbg_agent_suspend を有効にすると、JVM は中断されて、デバッガを付加してそのデバッガを通じて JVM に続行を指示するまで、すべての Adaptive Server の Java タスクが待機します。pca_jvm_java_dbg_agent_suspend を有効にするかわりに、JVM を起動して単純な Java コマンドを実行して、デバッガを付加できるようにすることをお勧めします。これによって JVM のブートが可能になり、デバッグ対象のクラスを実行する前にデバッガを付加できるようになります。

JVM でデバッグ・エージェントを有効にする設定値を設定すると、その次に JVM を起動したときにデバッグ・エージェントを使用できます。デバッグ・エージェントを無効にするには、設定パラメータを無効にして JVM を再起動します (JVM を起動してエージェントを実行した場合はそのエージェントをオフにできません)。

注意 デフォルトではデバッグ･エージェントを実行しないでください。デバッグ･エージェントを実行すると、ホストに対してネットワークでアクセスするすべてのデバッグ・アプリケーションが JVM に接続し、オブジェクトの内部データにアクセスする可能性があります。

リモート・デバッガの JVM デバッグ・エージェントへの付加

Adaptive Server で実行しているデバッグ・エージェントにリモート・デバッガが付加されると、デバッグ・セッションが開始します。sp_jreconfig で指定した接続情報に加えて、デバッグ対象クラスのソース・ファイルの場所を入力する必要があります。

IDE またはスタンドアロン・デバッガを使用する場合、デバッグ・エージェントにリモート・デバッガを付加する方法についてはベンダから提供されたマニュアルを参照してください。

次の例では、jdb コマンド・ライン・デバッガを使用していると想定しています。ポート 8000 でマシン "myhost" 上のデバッグ･エージェントに接続し、ホーム・ディレクトリの JAR アーカイブ *mysource.jar* にある Java ソース・ファイルを指定します。

```
jdb -attach myhost:8000 -source .:${HOME}/mysource.jar
```

デバッガ・ツールによって構文は異なりますが、接続情報とソース・ファイルの場所は、常に指定する必要があります。

第 8 章

Java を使用したファイルおよびネットワークへのアクセス

この章では、Java を使用したファイルおよびネットワークへのアクセスの例を示します。

トピック名	ページ
java.io を使用するファイル・アクセス	119
java.net を使用するファイル・アクセス	126

Adaptive Server は、java.io、java.net、java.nio の各パッケージを使用したファイルおよびネットワークの I/O 機能をサポートします。

注意 ファイルおよびネットワークの I/O がサーバの内外で大きなテキスト・ドキュメントの入出力ストリーミングを行う場合、JVM で使用できるメモリ容量を増やす必要がある場合があります。大きなドキュメントを処理する場合は、pci memory size 設定パラメータの値を増やして、メモリ要件が大きくなる状況に対応する必要もあります。「PCI メモリ・プール」(14 ページ) を参照してください。

java.io を使用するファイル・アクセス

PCA/JVM は、java.io と java.nio のパッケージによってファイルの直接 I/O をサポートします。これらのパッケージによって、ファイル・システムとの間で、ファイルの読み込みと書き込みができるようになります。

オペレーティング・システムによって使用されるユーザ ID と Adaptive Server によって使用されるユーザ ID を明確に区別する必要があります。

ユーザ ID とパーミッション

Adaptive Server が起動するとき、サーバのプロセスは、プロセスを開始したシステム・ユーザ ID を使用して実行します。たとえば、システム・ユーザ ID "sybase" によって Adaptive Server が起動される場合を考えます。

```
% ps -Usybase -o user,pid,command

USER      PID   CMD
sybase   20405   /sybase/ASE-15-0/bin/dataserver ...
```

このように、Adaptive Server のプロセスとオペレーティング・システムの間のすべての対話は、 Adaptive Server を起動したシステム・ユーザ ID と関連付けられます。

ただし、サーバ内では状況が異なります。各ユーザがサーバにログインするとき、ユーザは Adaptive Server サーバで定義されたユーザ ID を使用してログインします。このユーザ ID は、ホスト・マシンで定義されたユーザ ID とは異なります。あるユーザ ID が Adaptive Server とオペレーティング・システムの両方で同じユーザを表すと想定される場合でも、両者は区別されます。

データベース内では、ユーザは割り当てられた役割に基づいてさまざまなアクションを実行します。多くの場合、Adaptive Server にログインしたユーザには、ホスト・マシンのユーザ・アカウントがありません。したがって、サーバを起動したユーザ・アカウントは、複数のデータベース・ユーザのプロキシになる可能性があります。たとえば、Adaptive Server のユーザによって読み取られるファイルが 2 つあるとします (そのユーザのファイル・パーミッションは厳密に読み込み専用です)。

```
-r----------1 sybase sybuser    1263 Aug 19 18:54 myfile1.dat
-r----------1 jdoe   sybuser     952 Aug  7  9:02 myfile2.dat
```

これらのファイルを読み取ろうとする Java メソッドを実行するためにユーザが Adaptive Server にログインした場合、その Java のファイル I/O は結果的にホスト・インタフェースによって管理される関数に帰結します。

```
isql -Usa -P...
isql -Ujdoe -P...
isql -Ujanedoe -P...
```

基になる read() ランタイム関数の動作は、各ユーザに対して同じです。すべてのユーザは *myfile1.dat* を読み取ることができます。このファイルの所有者はシステム・ユーザ ID の "sybase" ですが、それは、そのユーザがそのサーバを所有していると、オペレーティング・システムによって識別されているためです。ただし、どのユーザも (見かけ上このファイルを所有しているデータベース・ユーザでも) *myfile2.dat* を読み取ることはできません。すべてのデータベース・ユーザの ID は単一のオペレーティング・システム ID の "sybase" に集約され、プロセスの所有者に関連付けられるためです。そのため、ファイル・アクセスは拒否されます。

ファイル I/O でのディレクトリの指定：UNIX プラットフォーム

従来の UNIX 表記を使用して、パスに対して追加のパーミッション制限を任意で指定できます。たとえば、"u+rw" は、あるユーザに読み込み / 書き込みアクセス、グループに読み込み専用アクセスを許可し、それ以外のユーザに対してアクセスを拒否します。これらの制限はオペレーティング・システムのパーミッションには影響しません。設定の文で読み込み / 書き込みアクセスを許可されたユーザは、オペレーティング・システムの読み込み専用パーミッションを持つディレクトリに対して書き込みアクセスをすることはできません。

マスクが指定されない場合、ファイル作成を含むすべての書き込み操作で、ディレクトリに対してデフォルト・マスク 0666 が使用されます。マスクは読み込み専用操作では使用されません。

マスクが指定されると、デフォルト・マスクはすべて 0 として想定されます。これによって、(u+rw) として指定されたマスクは 0600 のマスクになります。

マスクの構文

work_dir (信頼されたディレクトリ) のパーミッション・マスクは、次のようになります。

- パスの直後に、間にスペースを入れずに指定する。
- [u]：ユーザ、[g]：グループ、[o]：その他、[a]：すべて、を最初の文字として、次に +、-、=、r、w、x を指定して、マスクを定義する。

次に例を示します。

- (u=rw,go=r) は 0644 と同等です。
- (ugo+r,u+w) は 0644 と同等です。
- (ugo+r,u+wx) は 0755 と同等です。
- (ugo=rwx,go–wx) は 0755 と同等です。

マスクを定義する方法は数多くありますが、どの方法でも左から右に評価されます。たとえば、マスクが当初 0777 (ugo=rwx) と定義されているとします。g(グループ) と o(その他) に対する w(書き込み) と x(実行) を後で削除すると、8 進数の表現形は 0744 になり、そのマスクは (ugo=rwx,go–wx) になります。

マスクが指定されない場合 (マスクの指定が任意のとき)、そのディレクトリはデフォルトの書き込みマスク 0666 を使用します。

有効な構文は次のとおりです。

u ... ユーザ (または所有者)
g ... グループ
o ... その他 (または world)
a ... すべて (u、g、o を設定)。たとえば、(a+rw) は、u、g、o に対する
　読み込み / 書き込みをオンにします。
+ ... ビットをオンにする
– ... ビットをオフにする
= ... ビットを置き換える。たとえば、(u=rw) はユーザを置き換えます。
r ... 読み込みビット
w ... 書き込みビット
x ... 実行ビット

例

- pca_jvm_work_dir 配列に新しい作業ディレクトリ・パスを追加するには、次のように入力します。

```
sp_jreconfig "add", "work_dir", "/some/path(u+rw)
```

  または

```
sp_jreconfig "add", "work_dir", "/some/path(u=rw)
```

- pca_jvm_work_dir 配列の既存の作業ディレクトリ・パスを削除するには、次のように入力します。

```
sp_jreconfig "delete", "work_dir", "/some/path"
```

  work_dir の配列要素またはパス・エントリを削除または更新するとき、指定した文字列のパスの部分のみが必要です。

- pca_jvm_work_dir 配列の既存の作業ディレクトリ・パスを変更するには、次のように入力します。

```
sp_jreconfig "update", "work_dir", "/old", "/new"
```

- パスを変更してパーミッションを更新するには、次のように入力します。

```
sp_jreconfig "update", "work_dir", "/some/path(u+rw)", "/some/path(u+w)"
```

- pca_jvm_work_dir　配列の既存の作業ディレクトリ・パスを無効にするには、次のように入力します。

```
sp_jreconfig "disable", "work_dir", "/some/path"
```

  最後の引数は、work_dir 配列の個別の要素を指定する完全な (または部分的な) 文字列値で、その配列に要素が 1 つしかない場合でも指定する必要があります。

- pca_jvm_work_dir 配列の作業ディレクトリ・パスのセット全体をクリアするには、次のように入力します。

```
sp_jreconfig "array_clear", "work_dir"
```

- 配列全体を有効にするには、次のように入力します。

```
sp_jreconfig "array_enable", "work_dir"
```

- 配列全体を無効にするには、次のように入力します。

```
sp_jreconfig "array_disable", "work_dir"
```

ファイル I/O でのディレクトリの指定：Windows プラットフォーム

次の表記を使用して、パスに対して追加のパーミッション制限を、任意で指定できます。

マスクの構文

Windows 環境では、作業ディレクトリ定義の最後に次の構文を追加するとパーミッション・マスクを定義できます。

- /RW - 読み込み / 書き込みパーミッションを定義
- /RO - 読み込み専用パーミッションを定義
- /NA － アクセスなしを定義

例

- *D:¥my_work_dir* を信頼された完全なアクセスを許可して定義するには、次のように入力します。

```
sp_jreconfig "add", "work_dir", "C:¥my_work_dir/RW"
```

- *D:¥my_read_only* を信頼済みの読み込み専用として定義するには、次のように入力します。

```
sp_jreconfig "add", "work_dir","D:¥my_read_only_dir/RO"
```

- *E:¥general* を信頼された完全なアクセスを許可して定義し、ただし *TOP_SECRET* という *E:¥general* のサブディレクトリに対するアクセスを許可しないように定義するには、次のように入力します。

```
sp_jreconfig "add", "work_dir","E:¥general/RW;E:¥general¥TOP_SECRET/NA"
```

個々のディレクトリはセミコロンで区切ります。

ファイル I/O の変更

JVM のファイル I/O は、第一にファイルを開く操作によって制御されます。ファイルが正常に開いたら、そのファイルに対するその後の I/O 操作は通常の場合は許可されます。セキュリティ上の理由から、すべてのファイルを開く要求では物理ファイルへの絶対パスが必要です。ソフト・リンクはサポートされません。相対パスは、ファイル I/O 操作が試行される前に絶対パスに変換されます。このため、*$SYBASE* ディレクトリをソフト・リンクとして設定することはできません。そのようにすると、*$SYBASE/shared* のファイルを開くことができなくなるため、JVM は初期化できなくなります。

ファイルを開く操作が特定の規則に従わない場合、そのファイルは開きません。ファイルを開く規則の基本は、次のとおりです。

- そのファイルが存在しているかどうか
- そのファイルを読み込み専用で開くか、読み込み / 書き込み可能で開くか
- 開くファイルはどこにあるか

既存のファイルを開くための規則

ここでは、UNIX プラットフォームと Windows プラットフォームでファイルを開くための規則と確認事項について説明します。

注意 いずれかの確認事項が満たされない場合、ファイルを開く要求は拒否され、呼び出し側にはエラーが報告されます。

UNIX プラットフォームの場合

サーバと関連付けられたユーザ ID にそのファイルにアクセスするパーミッションがある場合、そのファイルが *$SYBASE/shared* ディレクトリにあるときは読み込み専用で開くことができます。$SYBASE 以外のディレクトリに対しては読み込みアクセスが許可されません。

注意 ファイルの作成を含む書き込みアクセスは、すべての $SYBASE ディレクトリに対して許可されません。

書き込みアクセスに対して開いたファイルには、そのファイルを開く要求が許可される前に追加の確認事項があります。Adaptive Server は次のことを確認します。

- ファイルを開く要求を発行したユーザは、そのファイルの所有者であること。
- ハード・リンクの数が 1 を超えないこと。複数の場合、要求は失敗します。

- 開くファイルが有効なディレクトリにあること。そのファイルが $SYBASE ディレクトリにある場合や、作業ディレクトリとして設定されたディレクトリにない場合、要求は失敗します。
- 作業ディレクトリには、ファイルに対する書き込みアクセスを許可するアクセス・マスクが設定されています。デフォルト・マスクは 0666 です。デフォルト以外のマスクを必要としない限り、マスクは必要ありません。

Windows プラットフォーム

サーバに関連付けられたユーザ ID がファイルに対するアクセスを許可されている場合、次のときにはアクセスが許可されます。

- %SYBASE% ディレクトリ構造にそのファイルが存在し、読み込み専用アクセスが許可され、書き込み用に開く要求が ERROR_ACCESS_DENIED エラーを受信する
- Windows %TEMP% ディレクトリにファイルが存在するか作成されていて、読み込み / 書き込みアクセスが許可されている
- 設定された作業ディレクトリ (信頼されたディレクトリ) にファイルが存在するか作成されていて、許可されたアクセスは作業ディレクトリに対して定義されている
- 信頼されたディレクトリの任意のサブディレクトリにファイルが存在するか作成されていて、許可されたアクセスは親ディレクトリに対して定義されている
- 1 つの信頼されたディレクトリが他のディレクトリにネストされている場合、システムはターゲット・ファイル・パスのそれぞれの信頼された親へのアクセスを確認し、最も制限されたアクセスを適用する。したがって、読み込み / 書き込みアクセスを信頼されたディレクトリ・ツリーに許可し、ただしその下の特定のディレクトリには読み込み専用またはアクセス拒否を指定できます。これは、Windows がファイルに ACL を適用するときの動作に似ています。

ファイルを開く操作でファイルを作成するための規則

存在しないファイルを開く要求は、実質的にはファイルを作成する操作で、既存のファイルに対する操作とは当然異なります。存在するファイルを書き込み用に開くときに適用されるものと同じ場所の制約が、新しく作成されるファイルにも適用されます。新しく作成されるファイルが *$SYBASE* ディレクトリ構造に含まれるか、または設定された作業ディレクトリに含まれない場合、要求は失敗します。さらに、そのディレクトリに対するアクセス・マスクが、サーバ・プロセスに関連付けられたユーザ ID に対して、ターゲット・ディレクトリへの書き込みを許可している必要があります。

注意 ファイルの作成を含む書き込みアクセスは、*/tmp* ディレクトリに対しては常に許可されます。

UNIX プラットフォームの場合、ファイルを開く要求で作成されたファイルには、書き込みアクセスを指定する必要があり、ファイルを開くフラグ (O-CREAT | O-EXCL | O-RDWR) とアクセス・マスク (0600) を使用して開かれます。セキュリティ上の理由から、これらのファイルを開くフラグとアクセス・マスクは、ファイルを開く要求で指定されるかどうかにかかわらず、常に使用されます。ファイルを作成するときに、ファイルを開くフラグで読み込み専用アクセスを指定することはできません。ファイルのサイズを制限したりディスク使用クォータを設定したりするには、オペレーティング・システム・レベルで指定する必要があります。

ファイルの最終確認

ファイルを開く要求がすべてのファイル確認事項をパスして開くことが許可されたら、最終確認によって、開いたファイルが元の要求を満たすようにします。これによって、確認事項を迂回することで、許可されていない他のファイルが開かれることが避けられます。ファイルを開く要求が失敗した場合、監査トレースには注釈が付加され、呼び出し側メソッドでは java.lang.IOException が発生します。IOException のメソッド固有の処理によって、その例外をユーザに表示するか、Java コードの代替メカニズムによって処理するかが決まります。

java.net を使用するファイル・アクセス

Adaptive Server では java.net および java.nio をサポートしているので、クライアント側の Java ネットワーク・アプリケーションをサーバ上で作成できます。任意のサーバに接続するネットワーク Java クライアント・アプリケーションを作成できるので、Adaptive Server を外部サーバへのクライアントとして有効に機能させることができます。

java.net と java.nio を使用すると、次のことができます。

- インターネット上の任意の URL からドキュメントをダウンロードする。
- サーバ内部から電子メール・メッセージを送信する。
- 外部サーバへ接続してドキュメントの保存、編集などのファイル機能を実行する。
- XML を使用してドキュメントにアクセスする。

注意　java.net を使用するには、以下の点を注意する必要があります。

- java.net に関連付けられたオブジェクトのほとんどは直列化できません。テーブルにも挿入できません。
- I/O 関連のほとんどのメソッドはバッファ付きの I/O を使用し、自動的にはフラッシュされません。そのようなメソッド (PrintWriter など) は、明示的にフラッシュする必要があります。

例

この項では、Socket クラスと URL クラスを使用した例について説明します。次の処理ができます。

- URL クラスと XML 問い合わせ言語 (XQL) を使用した、外部ドキュメントへのアクセス
- MailTo クラスを使用した、ドキュメントのメールによる送信

Socket クラスの使用

Java Socket クラスを使用すると、URL クラスより高度なネットワーク転送を実行できます。Socket クラスを使用することで、任意のネットワーク・ホストの指定したポートに接続でき、InputStream クラスと OutputStream クラスを使用したデータの読み込みや書き込みが可能です。

URL クラスの使用

URL クラスは、次の目的で使用できます。

- 電子メール・メッセージの送信。
- Web サーバからの HTTP ドキュメントのダウンロード。HTTP ドキュメントは、静的ファイルでも Web サーバが動的に生成したファイルでもダウンロードできる。
- XQL を使用した外部ドキュメントへのアクセス。

- mailto:URL クラスを使用した、ドキュメントのメールによる送信

たとえば、URL クラスを使用してドキュメントをメール送信できます。クライアントをメール・サーバに接続して、システム・プロパティが参照するマシン (この場合は salsa.sybase.com) が sendmail などのメール・サーバを実行している状態にします。

この例では、次の手順を実行します。

1 URL オブジェクトを作成します。

2 URLConnection オブジェクトを設定します。

3 URL オブジェクトから OutputStream オブジェクトを作成します。

4 メールを作成します。次に例を示します。

```
import java.io.*;
import java.net.*;
public class MailTo {
    public static void sendIt()
        throws Exception{
        System.getProperty("mail.host", "salsa.sybase.com");
        URL url = new URL("mailto:name@sybase.com");
        URLConnection conn = url.openConnection();
        PrintStream out = new PrintStream(conn.getOutputStream(),true);
        out.println ("From janedoes@sybase.com");
        out.println ("Subject: Works Great!");
        out.println ("Thanks for the example - it works great!");
        out.close();
        System.out.println("Message Sent");
    }
}
```

5 データベース内から電子メール・メッセージを送信するために、mailto:URL をインストールします。

```
select MailTo.sendIt()
```

また、URL クラスを使用すると、HTTP URL からドキュメントをダウンロードできます。クライアントの起動時に Web サーバに接続します。手順は次のとおりです。

1 URL オブジェクトを作成します。

2 URL オブジェクトから InputStream オブジェクトを作成します。

3 InputStream オブジェクトに対して read を使用してドキュメントを読み込みます。

次のコード例は、ドキュメント全体を Adaptive Server のメモリに読み込み、そのメモリ上のドキュメントに新しい InputStream を作成します。

```
import java.io.*;
import java.net.*;
public class URLprovess {
            public static InputStream readURL()
                throws Exception {
                URL u = newURL("http://www.xxxx.con");
                InputStream in = u.openStream");
                //This is the same as creating URLConnection, then calling
                //getInputStream(). In Adaptive Server, you must read the entire
                //document into memory, and then create an InputStream on the
                //in-memory copy.
                int n = 0;
                int off = 0;
                byte b() = new byte(50000);
                for)off = 0; (off<b.length512) &&
                    ((n = in.read(b.off,512)! = 1);off+=n) {}
                System.out.prinln("Number of bytes read :" + off);
                in.close();
                ByteArrayInputStream test = new ByteArrayInputStream(b,-,off);
                return (InputStream) test;
            }
}
```

新しい InputStream クラスを作成したら、このクラスをインストールしてテキスト・ファイルをデータベースに読み込むことができます。次の例では、テーブル mytable にデータを挿入します。

```
create table mytable (cl text)
go
insert into mytable values (URLprocess.readURL())
go
Number of bytes read :40867
select datalength(cl) from mytable
go

---------------------
40867
```

第 9 章 補足トピック

この章では、リファレンス・トピックについて説明します。

トピック名	ページ
サーバにおける Java クラスの JDK 使用条件	131
割り当て	131
可能な変換	133
クライアントへの Java-SQL オブジェクトの転送	133
パフォーマンス向上のためのアドバイス	134
PCA/JVM でのネイティブ・メソッドへのアクセスの制御	137
サポートされていない Java API パッケージ、クラス、メソッド	138
Java からの SQL の呼び出し	141
Java メソッドからの Transact-SQL コマンド	142
Java と SQL の間のデータ型のマッピング	146
Java-SQL 識別子	147
Java-SQL のクラス名およびパッケージ名	148
Java-SQL カラムの宣言	149
Java-SQL 変数の宣言	149
Java-SQL カラムの参照	150
Java-SQL メンバ参照	151
Java-SQL メソッド呼び出し	152

サーバにおける Java クラスの JDK 使用条件

サーバにインストールして使用する Java クラスは、PCA/JVM を通して Adaptive Server にプラグインされた JVM のバージョンより低い必要があります。PCA/JVM は、Java 6 以降をサポートします。

割り当て

この項では、データ型が Java-SQL クラスである SQL データ項目間での割り当てのルールについて定義します。

それぞれの割り当てにより、ソース・インスタンスがターゲットとなるデータ項目に転送されます。

- Java-SQL カラムを持つテーブルを指定する insert 文は、ターゲットとなるデータ項目として Java-SQL カラムを参照し、ソース・インスタンスとして挿入値を参照する。
- Java-SQL カラムを更新する update 文は、ターゲットとなるデータ項目として Java-SQL カラムを参照し、ソース・インスタンスとして更新値を参照する。
- 変数やパラメータに対して割り当てる select 文または fetch 文は、ターゲットとなるデータ項目として変数またはパラメータを参照し、ソース・インスタンスとして取得値を参照する。

注意 ソースが変数またはパラメータの場合、それが Java VM 内のオブジェクトに対する参照になります。ソースがカラムの参照である場合、直列化しているので、カラムの参照に対するルール (「Java-SQL カラムの参照」(150 ページ) を参照) は、Java VM 内のオブジェクトに対する参照になります。このように、ソースは Java VM 内のオブジェクトに対する参照になります。

コンパイル時の割り当てのルール

1. SC と TC を、ソースおよびターゲットのコンパイル時クラス名として定義します。ターゲットに関連付けられたデータベース内で、SC_T および TC_T を SC および DT という名前のクラスとして定義します。同様に、ソースに関連付けられたデータベース内で、SC_S および TC_S を SC および DT という名前のクラスとして定義します。
2. SC_T は、TC_T または TC_T のサブクラスと同じにします。

実行時の割り当てのルール

DT_SC が、DT_TC またはそのサブクラスと同じであると仮定します。

- RSC を、ソースの値のランタイム・クラス名として定義します。ソースに関連付けられたデータベース内で、RSC_S を RSC という名前のクラスとして定義します。RSC_T を、RSC_T クラスの名前として定義します。RSC_T クラスは、ターゲットに関連付けられたデータベースにインストールされています。RSC_T クラスがない場合、例外が発生します。RSC_T が、TC_T または TC_T のサブクラスのどちらとも異なる場合、例外が発生します。
- ソースとターゲットに関連付けられたデータベースが異なる場合、ソース・オブジェクトは現在のクラス RSC_S によって直列化され、クラス RSC_T によって非直列化されます。これによりターゲットに対応したデータベース内で関連付けられます。

- ターゲットが SQL 変数またはパラメータの場合、ソースはターゲットへの参照によってコピーされます。
- ターゲットが Java-SQL カラムの場合、ソースは直列化され、その直列化がクライアントに deep コピーされます。

可能な変換

convert を使用してデータ型の式を次のように変換できます。

- Java データ型が Java オブジェクト型である Java 型を、「Java と SQL の間のデータ型のマッピング」(146 ページ) に示した SQL データ型に変換。convert 関数は、Java-SQL マッピングにより暗示されたマッピングを行います。
- SQL データ型を「Java と SQL の間のデータ型のマッピング」(146 ページ) に示した Java 型に変換。convert 関数は、SQL-Java マッピングにより暗示されたマッピングを行います。
- 式 (ソース・クラス) のコンパイル時データ型のソースがターゲット・クラスのサブクラスまたはスーパークラスである場合、SQL システムにインストールされている Java-SQL クラスを SQL システムにインストールされている別の Java-SQL クラスに変換。これ以外の場合は、例外が発生します。

変換の結果は、現在のデータベースと関連付けられます。

Java 部分型に対する convert 関数の使い方については、「Java-SQL データの部分型」を参照してください。

クライアントへの Java-SQL オブジェクトの転送

データ型が Java-SQL オブジェクト型の値は、Adaptive Server からクライアントに転送されます。データの変換はクライアント側の型に依存します。

- isql クライアントの場合、オブジェクトの toString( ) メソッドまたはこれに類似したメソッドが呼び出され、結果が varchar にトランケートされてクライアントに転送されます。

注意　クライアントに転送されるバイト数は、@@stringsize グローバル変数の値によって異なります。デフォルト値は 50 バイトです。詳細については、「Java インスタンスの表現」(39 ページ) を参照してください。

- jConnect 4.0 以降を使用している Java クライアントの場合、サーバはオブジェクトの直列化をクライアントに転送します。直列化は、jConnect によって連続的に非直列化され、オブジェクトのコピーを生成します。
- bcp クライアントの場合
  - オブジェクトが in row として宣言されたカラムである場合、カラムのなかにある直列化した値は、カラム・サイズで指定した長さの varbinary 値としてクライアントに転送されます。
  - それ以外の場合、オブジェクトの直列化した値 (オブジェクトの writeObject メソッドの結果) は、image 値としてクライアントに転送されます。

パフォーマンス向上のためのアドバイス

この項では、Adaptive Server で Java を使用するときにパフォーマンスを向上させるためのガイドラインを示します。

SQL から JVM への呼び出し数の最小化

既製の JVM (つまり PCA/JVM) は、JVM の機能と大幅な最適化の利点を活用しているため、Adaptive Server 15.0.2 以降の内部 JVM に比べて非常に高速です。ただし、Java への SQL 呼び出しを伝達すると、PCA/JVM で顕著なボトルネックが発生することがあります。

PCA/JVM の速度を活用するには、SQL から JVM への呼び出し数を最小限に抑えます。

簡単な Address クラスについて考えてみます。

```
public class Address implements java.io.Serializable {
    private int state;
    private String street;
    private String zip;

    // ...

    public Address()
    {
        // ...
    }

    public Address(String street, String zip, int state)
    {
        this();
        this.setStreet(street);
```

```
        this.setZip(zip);
        this.setState(state);
    }

    // ,,,

    public void setStreet(String street)
    (
        // ..
    }

    public void setZip(String zip)
    {
        // ...
    }
}
```

JVM への呼び出しに関連するオーバヘッドのため、引数のないコンストラクタの後にデータ・メンバの set メソッドを使用するよりも、SQL から 3 つの引数を持つコンストラクタを使用する方がかなり早くなります。次のような文があるとします。

```
1> declare @a Address
2> select @a=new Address("123 Elm Street", "12345", 10)
```

この文は、次の文より効率的です。

```
1> declare @a Adress
2> select @a = new Address()
3> select @a >> setStreet("123 Elm Street")
4> select @a >> setZip("12345")
5> select @a >> setState(10)
```

SQL-Java インタフェースの交差の繰り返しを求めずに Java に処理を要求すると、オーバヘッドが減少し、JVM の改善された機能をより十分に活用できます。

java.lang.Thread クラスの慎重な使用

PCA/JVM は、Adaptive Server でマルチスレッド・メソッドを使用するクラスを作成可能な java.lang.Thread クラスをサポートします。Java メソッド内で作成されたスレッドは、CPU および他のリソースに関して Adaptive Server と競合します。スレッドの数が多い場合や、スレッドが大量のリソースを必要とする場合は、サーバ全体のパフォーマンスに影響を与える可能性があります。

PCA/JVM 内で実行されているかどうかの判断

通常、クラスが PCA/JVM とスタンドアロン JVM で実行されていてもほとんど違いがありません。ブール論理を使用すると、クラスが Sybase ContextClassLoader を通してロードされているかどうかを確認できます。次に例を示します。

```
boolean running_in_ase = false;

running_in_ase =
this.getClass().getClassLoader().getName().equals
("sybase.aseutils.ContextClassLoader");

if (running_in_ase)
{
    //in ASE
    ...
}
else
{
    //in a standalone JVM
    ...
}
```

マルチエンジン環境での SQL ループの回避

マルチエンジン環境では、特定の Java/SQL コマンドがパフォーマンスに悪影響を与えることがあります。これは、通常同じ Java メソッドが SQL ループ内で複数回実行されたときに起こります。これを回避するには、メソッドとループが VM コンテキストで実行されるように Java/SQL コマンドを記述します。

1 Java でループを記述します。

2 Java コードのループからメソッドを呼び出します。

PCA/JVM でのネイティブ・メソッドへのアクセスの制御

Java 言語では、Java 以外の言語に実装された言語を、ネイティブ・メソッドを使用して Java ネイティブ・インタフェース (JNI) から使用できます。ネイティブ・メソッドを使用するクラスは、java.lang.System クラスと java.langRuntime クラスの両方に記述されているとおりに、load(String *filename*) メソッドまたは loadLibrary(String *libname*) メソッドを使用してネイティブ・ライブラリを明示的にロードする必要があります。これらのライブラリは、制御されるオブジェクトとしてデータベースに格納されるわけではないため、ユーザによっては安全でないと見なす可能性があります。

ネイティブ・ライブラリに予期しないアクセスが行われないようにするため、ネイティブ・ライブラリのロードを制御するシステム・プロパティ sybase.allow.native.lib が PCA/JVM に導入されました。

多くの Java プロパティは、コマンド・ラインで、または java.lang.System setProperty(String *key*, String *value*) メソッドを使用してアプリケーション内から設定できます。ただし、これはユーザがシステム・ポリシーを上書きできないようにするため、SecurityManager により禁止されています。デフォルトでは、ユーザはネイティブ・ライブラリをロードできません。ネイティブ・ライブラリのロードまたは既存のプロパティ設定の変更が試みられた場合、SecurityException が発生し、ロードの試行に失敗します。

たとえば、sybase.allow.native.lib プロパティを設定せずに java.net.ServerSocket クラスをロードしようとした場合、初期化にはソケット・ライブラリをロードする必要があるため、エラーが発生します。実際の Java スタックは異なりますが、次のような Java スタックまたはクライアント・メッセージが表示されます。

```
java.lang.SecurityException: Cannot load native libraries from
within a user Task!
```

これは、必要なネイティブ・ライブラリをロードできなかったことを示しています。

ネイティブ・ライブラリをロードできるようにするには、JVM の起動前に sybpcidb データベースでこのプロパティを設定します。

```
1>  sp_jreconfig "add","pca_jvm_java_option",
"-Dsybase.allow.native.lib=true"

2>  go
```

sybase.allow.native.lib が true に設定されると、JVM の起動時にコマンド・ラインで追加のプロパティが JVM に渡されます。このプロパティは、JVM の実行中は変更できません。ライブラリをロードする必要がなくなった場合は、sp_jreconfig を使用して pca_jvm_java_option を削除または無効にします。

サポートされていない Java API パッケージ、クラス、メソッド

Adaptive Server は、Java API のほとんどのクラスやメソッドをサポートしていますが、サポートしていないものもあります。また、Adaptive Server によって、セキュリティ上の制限や実装に関する制限事項が加えられる場合もあります。たとえば、Adaptive Server は java.lang.Thread のスレッド操作のすべての機能をサポートしているわけではありません。

警告！ 子スレッドを生成するメソッドを使用するときは注意してください。Java メソッド内で起動された java.lang.Thread オブジェクトは、Adaptive Server スケジューラによってではなく実行時にスケジュールされます。これらのスレッドがプロセッサを集中利用したり、多数のスレッドを生成したりする場合、多量に消費するユーザ・スレッドによるプロセッサ時間の競合のためサーバのパフォーマンスが低下することがあります。

PCA/JVM は標準の Java プラグインを使用するため、完全なクラスの配布を使用できます。通常、メソッドの使用がサーバや他の Java タスクの操作に干渉するリスクがなければ、それらのメソッドはサポートされます。

Adaptive Server の Java は、Java ネイティブ・インタフェース (JNI) を通して呼び出されるネイティブ・メソッドをサポートしません。

この項では次のリストを示します。

- サポートされていない Java メソッド
- サポートされていない java.sql メソッド

制限付き Java パッケージ、クラス、メソッド

- JVM はヘッドレス・モードで実行されるため、ユーザによる入力や出力が必要な Java メソッドは無効になります。
- サーバや他の JVM タスクの操作に干渉する可能性がある操作は許可されません。
- 次の java.lang.Thread メソッドは許可されません。
    - interrupt()
    - setPriority ()
    - setName()
    - enumerate()
    - setDaemon()
    - checkAccess()
    - getContextClassLoader()

    - setDefaultExceptionHandler()
    - setContextClassLoader()
    - getStackTrace()
    - getAllStarkTraces()
    - setDefaultUncaughtExceptionHandler()
    - stop()
    - destroy()
    - suspend()
    - resume()
    - 古い関数は許可されますが、安全ではない可能性があります。
        - countStackFrames()
- 次の java.lang.ThreadGroup メソッドは許可されません。
    - getParent()
    - setDaemon()
    - setMaxPriority()
    - checkAccess()
    - enumerate()
    - interrupt()
    - stop()
    - destroy()
    - suspend()
    - resume()
    - 古い関数は許可されますが、安全ではない可能性があります。
        - allowThreadSuspension()
- セキュリティの問題
    - 既存の SecurityManager を上書きしたり、他のクラス・ローダをインスタンス化したりすることはできません。
    - java.lang.System および java.lang.Runtime の exit() メソッドは許可されません。

サポートされていない *java.sql* メソッドとインタフェース

Java 6 クラスの配布では、java.sql パッケージは JDBC 4.x 仕様に準拠しています。ただし、基本となる Sybase 実装は JDBC 2.0 レベルです。JDBC 2.0 仕様はサポートされていないため、すべての JDBC メソッドが含まれます。また、JDBC 2.0 で定められた次のメソッドはサポートされていません。

- Connection.commit( )
- Connection.getMetaData( )
- Connection.nativeSQL( )
- Connection.rollback( )
- Connection.setAutoCommit( )
- Connection.setCatalog( )
- Connection.setReadOnly( )
- Connection.setTransactionIsolation( )
- DatabaseMetaData.* − DatabaseMetaData は以下のメソッド以外ではサポートされています。
  - deletesAreDetected()
  - getUDTs()
  - insertsAreDetected()
  - updatesAreDetected()
  - othersDeletesAreVisible()
  - othersInsertsAreVisible()
  - othersUpdatesAreVisible()
  - ownDeletesAreVisible()
  - ownInsertsAreVisible()
  - ownUpdatesAreVisible()
- PreparedStatement.setAsciiStream( )
- PreparedStatement.setUnicodeStream( )
- PreparedStatement.setBinaryStream( )
- ResultSetMetaData.getCatalogName( )
- ResultSetMetaData.getSchemaName( )
- ResultSetMetaData.getTableName( )
- ResultSetMetaData.isCaseSensitive( )

- ResultSetMetaData.isReadOnly( )
- ResultSetMetaData.isSearchable( )
- ResultSetMetaData.isWritable( )
- Statement.getMaxFieldSize( )
- Statement.setMaxFieldSize( )
- Statement.setCursorName( )
- Statement.setEscapeProcessing( )
- Statement.getQueryTimeout( )
- Statement.setQueryTimeoutt()

Java からの SQL の呼び出し

Adaptive Server は、JDBC 1.1 および 1.2 仕様を実装し、2.0 に準拠したネイティブ JDBC ドライバ java.sql を提供します。java.sql により、Adaptive Server で実行中の Java メソッドが SQL オペレーションを実行できるようになります。

特別な考慮事項

java.sql.DriverManager.getConnection( ) は次の URL を受け付けます。

- null
- “” (null 文字列)
- jdbc:default:connection

SQL を Java から呼び出すときは、いくつかの制限が適用されます。

- 更新作業を実行している SQL クエリ (update、insert、または delete) は、java.sql の機能を使用して他の SQL オペレーションを呼び出すことができない (他の SQL オペレーションも更新作業を実行しているため)。
- SQL で java.sql 機能を使用して起動したトリガは、結果セットを生成できない。
- java.sql を使用して、拡張ストアド・プロシージャまたはリモート・ストアド・プロシージャを実行することはできない。

Java メソッドからの Transact-SQL コマンド

SQL システムのなかで呼び出された Java メソッドの特定の Transact-SQL コマンドを使うことができます。表 9-1 は、Transact-SQL コマンドのリストと、それが Java メソッドで使用できるかどうかを示しています。これらのコマンドの詳細については、『リファレンス・マニュアル：コマンド』を参照してください。

表 9-1: Transact-SQL コマンドのサポート状況

コマンド	ステータス
alter database	サポートなし
alter role	サポートなし
alter table	サポートしている
begin ... end	サポートしている
begin transaction	サポートなし
break	サポートしている
case	サポートしている
checkpoint	サポートなし
commit	サポートなし
compute	サポートなし
connect - disconnect	サポートなし
continue	サポートしている
create database	サポートなし
create default	サポートなし
create existing table	サポートなし
create function	サポートしている
create index	サポートなし
create procedure	サポートなし
create role	サポートなし
create rule	サポートなし
create schema	サポートなし
create table	サポートしている
create trigger	サポートなし
create view	サポートなし
cursors	サポートなし "server cursors" のみサポートしている。server cursors はストアド・プロシージャ内で宣言および使用される cursors である。
dbcc	サポートなし
declare	サポートしている
disk init	サポートなし

コマンド	ステータス
disk mirror	サポートなし
disk refit	サポートなし
disk reinit	サポートなし
disk remirror	サポートなし
disk unmirror	サポートなし
drop database	サポートなし
drop default	サポートなし
drop function	サポートしている
drop index	サポートなし
drop procedure	サポートなし
drop role	サポートなし
drop rule	サポートなし
drop table	サポートしている
drop trigger	サポートなし
drop view	サポートなし
dump database	サポートなし
dump transaction	サポートなし
execute	サポートしている
goto	サポートしている
grant	サポートなし
group by and having clauses	サポートしている
if...else	サポートしている
insert table	サポートしている
kill	サポートなし
load database	サポートなし
load transaction	サポートなし
online database	サポートなし
order by Clause	サポートしている
prepare transaction	サポートなし
print	サポートなし
raiserror	サポートしている
readtext	サポートなし
return	サポートしている
revoke	サポートなし
rollback trigger	サポートなし
ÉçÅ[ÉãÉoÉbÉN	サポートなし
save transaction	サポートなし
set	set コマンドのオプションは 表 9-2 を参照。
setuser	サポートなし

コマンド	ステータス
shutdown	サポートなし
truncate table	サポートしている
union Operator	サポートしている
update statistics	サポートなし
update	サポートしている
use	サポートなし
waitfor	サポートしている
where Clause	サポートしている
while	サポートしている
writetext	サポートなし

表 9-2 は、set コマンドのオプションとそれが Java メソッドで使用できるかどうかを示します。

表 9-2: set コマンドのオプションのサポート状況

set コマンドのオプション	ステータス
ansinull	サポートしている
ansi_permissions	サポートしている
arithabort	サポートしている
arithignore	サポートしている
chained	サポートなし(注意 1 を参照)
char_convert	サポートなし
cis_rpc_handling	サポートされていない
close on endtran	サポートされていない
cursor rows	サポートされていない
datefirst	サポートしている
dateformat	サポートしている
fipsflagger	サポートされていない
flushmessage	サポートされていない
forceplan	サポートしている
identity_insert	サポートしている
language	サポートされていない
lock	サポートしている
nocount	サポートしている
noexec	サポートされていない
offsets	サポートされていない
or_strategy	サポートしている
parallel_degree	サポートしている (注意 2 を参照)
parseonly	サポートされていない
prefetch	サポートしている

set コマンドのオプション	ステータス
process_limit_action	サポートしている (注意 2 を参照)
procid	サポートされていない
proxy	サポートされていない
quoted_identifier	サポートしている
replication	サポートされていない
role	サポートされていない
rowcount	サポートしている
scan_parallel_degree	サポートしている (注意 2 を参照)
self_recursion	サポートしている
session_authorization	サポートされていない
showplan	サポートしている
sort_resources	サポートされていない
statistics io	サポートされていない
statistics subquerycache	サポートされていない
statistics time	サポートされていない
string_rtruncation	サポートしている
stringsize	サポートしている
table count	サポートしている
textsize	サポートされていない
transaction iso level	サポートなし(注意 1 を参照)
transactional_rpc	サポートされていない

注意　(1) chained オプションまたは transaction isolation level オプションを指定した set コマンドは、指定した設定がすでに有効になっているときだけ許可されます。つまり、この種の set コマンドは、まったく効果がないときに許可されます。これは、ストアド・プロシージャ内の共通のコーディング手法をサポートするために行われます。

注意　(2) 並列度に関連する set コマンドは許可されますが、効果はありません。これは、他のコンテキストの並列度を設定するストアド・プロシージャの使用をサポートします。

Java と SQL の間のデータ型のマッピング

Adaptive Server では、SQL データ型を Java 型にマッピングし (SQL-Java データ型マッピング)、Java スカラ型を SQL データ型にマッピングします (Java-SQL データ型マッピング)。表 9-3 は、SQL-Java データ型マッピングを示します。

表 9-3: SQL データ型から Java 型へのマッピング

SQL 型	Java 型
char	String
varchar	String
nchar	String
nvarchar	String
unichar	String
univarchar	String
unitext	String
text	String
numeric	java.math.BigDecimal
decimal	java.math.BigDecimal
money	java.math.BigDecimal
smallmoney	java.math.BigDecimal
bit	boolean
tinyint	byte
smallint	short
integer	int
bigint	long
unsigned smallint	int
unsigned int	long
unsigned bigint	java.math.BigInteger
bigint	java.math.BigInteger
real	float
float	double
double precision	double
binary	byte[ ]
varbinary	byte[ ]
image	java.io.InputStream
datetime	java.sql.Timestamp
smalldatetime	java.sql.Timestamp
bigdatetime	java.sql.Timestamp
bigtime	java.sql.Time
date	java.sql.Date
time	java.sql.Time

注意　unsigned bigint から double へのマッピングは近似であり、正確な値は提供されません。正確な値を得るには、Java メソッドに渡すときに unsigned bigint 値を string 値に変換してください。

表 9-4 は、Java-SQL データ型マッピングを示します。

表 9-4: Java スカラ型から SQL データ型へのマッピング

Java スカラ型	SQL 型
boolean	bit
byte	tinyint
short	smallint
int	integer
long	bigint
float	real
double	double

Java-SQL 識別子

説明

Java-SQL 識別子は、SQL 内で参照できる Java 識別子のサブセットです。

構文

```
java_sql_identifier ::= アルファベット文字 | アンダースコア (_) 記号
          [ アルファベット文字 | アラビア数字 | アンダースコア (_) 記号 |
          ドル ($) 記号 ]
```

使用法

- Java-SQL 識別子の長さは、引用符で囲まれていれば最大 255 バイトまで使用可能です。それ以外の場合は 30 バイト以下にします。
- 識別子の最初の文字にはアルファベット (大文字または小文字) あるいはアンダースコア記号 (_) を使用します。その次の文字には、アルファベット (大文字または小文字)、数字、ドル記号 ($)、またはアンダースコア (_) 記号を使用できます。
- Java-SQL 識別子では、常に大文字と小文字が区別されます。

区切り識別子

- 区切り識別子は、二重引用符で囲まれたオブジェクト名です。Java-SQL 識別子に区切り識別子を使用すると、Java-SQL 識別子の命名に対する一部の制限を取り除くことができます。

注意　set quoted_identifier オプションが on の場合も off の場合も、Java-SQL 識別子に二重引用符を使うことができます。

- 区切り識別子では、パッケージ、クラス、メソッドなどに SQL 予約語を使用できます。文の中で区切り識別子を使うたびに二重引用符で囲む必要があります。次に例を示します。

```
create table t1
        (c1 char(12)
        c2 p1.”select”.p2.”jar”)
```

- 二重引用符はそれぞれの Java-SQL 識別子だけを囲み、完全に修飾された名前は囲みません。

参照

識別子の追加情報については、『リファレンス・マニュアル：ビルディング・ブロック』の「第 4 章 式、識別子、およびワイルドカード文字」を参照してください。

Java-SQL のクラス名およびパッケージ名

説明

Java-SQL クラスまたはパッケージを参照するには、次の構文を使います。

構文

java_sql_class_name ::= [*java_sql_package_name*.]*java_sql_identifier*

java_sql_package_name ::=
[*java_sql_package_name*.]*java_sql_identifier*

パラメータ

java_sql_class_name
現在のデータベース内にある Java-SQL クラスの完全に修飾された名前です。

java_sql_package_name
現在のデータベース内にある Java-SQL パッケージの完全に修飾された名前です。

java_sql_identifier
「Java-SQL 識別子」を参照してください。

使用法

Java-SQL クラス名には次のことが当てはまります。

- クラス名の参照は常に現在のデータベース内のクラスを参照します。
- パッケージ名を参照せずに Java-SQL クラス名を指定する場合、現在のデータベース内にその名前の Java-SQL クラスが 1 つだけしかなく、そのパッケージがデフォルト (匿名) のパッケージである必要があります。
- SQL のユーザ定義データ型と Java-SQL クラスの識別子のシーケンスが同じ場合、Adaptive Server は、そのユーザ定義 SQL データ型名を使用し、Java-SQL クラス名は無視します。

Java-SQL パッケージ名には次のことが当てはまります。

- Java-SQL サブパッケージ名を指定する場合、そのパッケージ名とともにサブパッケージ名を参照する必要があります。

  java_sql_package_name.java_sql_subpackage_name

- Java-SQL パッケージ名はクラス名またはサブパッケージ名の修飾子としてのみ使用し、remove java コマンドを使用してパッケージを削除します。

Java-SQL カラムの宣言

説明

テーブルを作成または変更した場合に Java-SQL カラムを宣言するには、次の構文を使用します。

構文

```
java_sql_column ::= column_name  java_sql_class_name
```

パラメータ

java_sql_column
Java-SQL カラムを宣言する構文を指定します。

column_name
Java-SQL カラムの名前です。

java_sql_class_name
現在のデータベース内にある Java-SQL クラスの名前です。これがカラムの「宣言されたクラス」です。

使用法

- 宣言されたクラスは、**直列化**または**外部化**インタフェースを実装する必要があります。
- Java-SQL カラムは、常に現在のデータベースに対応しています。
- Java-SQL カラムは次のように指定することはできません。
  - not null
  - unique
  - プライマリ・キー

参照

Java-SQL カラムの宣言はテーブルを作成または変更する場合にだけ使用します。『リファレンス・マニュアル：コマンド』の create table および alter table の説明を参照してください。

Java-SQL 変数の宣言

説明

Java-SQL 変数の宣言を使用して、データ型が Java-SQL クラスである変数とストアド・プロシージャのパラメータを宣言します。

構文

```
java_sql_variable  ::=  @variable_name java_sql_class_name
java_sql_parameter  ::=  @parameter_name java_sql_class_name
```

パラメータ

java_sql_variable
SQL ストアド・プロシージャ内で Java-SQL 変数の構文を指定します。

java_sql_parameter
SQL ストアド・プロシージャ内で Java-SQL パラメータの構文を指定します。

java_sql_class_name
現在のデータベース内にある Java-SQL クラスの名前です。

使用法

java_sql_variable または *java_sql_parameter* は、常にストアド・プロシージャを含むデータベースに関連付けられます。

参照

変数宣言の詳細については、『リファレンス・マニュアル』を参照してください。

Java-SQL カラムの参照

説明

Java-SQL カラムを参照するには、次の構文を使用します。

構文

column_reference ::=
[[[*database_name*.]*owner*.]*table_name*.]*column_name*
| *database_name*..*table_name*.*column_name*

パラメータ

column_reference
データ型が Java-SQL クラスであるカラムへの参照です。

使用法

- カラムの値が null の場合、カラムの参照も null になります。
- カラムの値が Java の直列化 S で、そのクラスの名前が CS の場合、次のようになります。
  - クラス CS が現在のデータベースに存在しない場合、または CS が直列化に関連付けられたデータベース内にあるクラスの名前でない場合、例外が発生します。

    注意 直列化に関連付けられたデータベースは、通常はカラムを含むデータベースです。ただし、"insert into #tempdb" で作成される作業テーブルとテンポラリ・テーブルに含まれる直列化は、最初に直列化が格納されていたデータベースと関連付けられます。

  - カラム参照の値は次のようになります。

    ```
    CSC.readObject(S)
    ```

    CSC はカラム参照です。この式により、取得されていない Java の例外を引き起こす場合、例外が発生します。

    式は Java VM 内のオブジェクトに対する参照になります。これは、直列化に関連付けられたデータベースに関連付けられています。

Java-SQL メンバ参照

説明

クラスまたはクラス・インスタンスのフィールドあるいはメソッドを参照します。

構文

member_reference ::= *class_member_reference* |
　　instance_member_reference

class_member_reference ::= *java_sql_class_name.method_name*

instance_member_reference ::= *instance_expression>>member_name*

instance_expression ::= *column_reference* | *variable_name*
　　| *parameter_name* | *method_call* | *member_reference*

member_name ::= *field_name* | *method_name*

パラメータ

member_reference
クラスまたはオブジェクトのフィールドまたはメソッドを記述する式です。

class_member_reference
Java-SQL クラスの静的メソッドを記述する式です。

instance_member_reference
Java-SQL クラス・インスタンスの静的メソッド、動的メソッド、またはフィールドを記述する式です。

java_sql_class_name
現在のデータベース内の完全に修飾された Java-SQL クラス名です。

instance_expression
データ型が Java-SQL クラスの式です。

member_name
クラスまたはクラス・インスタンスのフィールド名あるいはメソッド名です。

使用法

- メンバがクラス・インスタンスのフィールドを参照する場合、そのインスタンスは null 値を持ち、Java-SQL メンバ参照は fetch 文、select 文、または update 文のターゲットになるので、例外が発生します。

  それ以外の場合、Java-SQL メンバ参照は null 値を持ちます。

- 二重山カッコ (>>) とドット修飾 (.) は、加法演算子 (+) や等号演算子 (=) などの演算子より優先されます。

  ```
  X>>A1>>B1 + X>>A1>>B2
  ```

  この式では、メンバが参照されたあと加算処理が実行されます。

- メンバ参照によって指定されたフィールドまたはメソッドは、Java-SQL クラスのデータベースと同じデータベース、または Java-SQL クラスのインスタンスと関連付けられます。

  メンバ参照の Java 型が Java スカラ型の 1 つである場合 (boolean、byte など)、対応する参照の SQL データ型は Java 型をその同等の SQL 型にマッピングすることによって取得されます。

メンバ参照の Java 型がオブジェクト型である場合、SQL データ型は同じ Java オブジェクト型またはクラスです。

Java-SQL メソッド呼び出し

説明

単一の値を返す Java-SQL メソッドを呼び出すには、次の構文を使用します。

構文

method_call ::= *member_reference* ([*parameters*])
　　| new *java_sql_class_name* ([*parameters*])

parameters ::= *parameter* [(, *parameter*)...]

parameter ::= *expression*

パラメータ

method_call
静的メソッド、インスタンス・メソッド、またはクラス・コンストラクタの呼び出しです。メソッド呼び出しは、メソッドのデータ型を定数以外にする必要がある式で使用できます。

member_reference
メソッドを示すメンバ参照です。

parameters
メソッドに渡されるパラメータのリストです。パラメータがない場合は、空のカッコを入れます。

使用法

メソッドのオーバロード

- 同じクラスまたはインスタンス内に同じ名前のメソッドがある場合、Java メソッドのオーバロードの規則に従って解決されます。

メソッド呼び出しのデータ型

- メソッド呼び出しのデータ型は次のように決定されます。
  - メソッド呼び出しが new を指定する場合、そのデータ型は Java-SQL クラスのデータ型になります。
  - メソッド呼び出しが型の値を返すメソッドを示すメンバ参照を指定する場合、メソッド呼び出しのデータ型はその型になります。
  - メソッド呼び出しが void 静的メソッドを示すメンバ参照を指定するとき、メソッド呼び出しのデータ型は SQL integer 型になります。
  - メソッド呼び出しがクラスの void インスタンス・メソッドを示すメンバ参照を指定するとき、メソッド呼び出しのデータ型はそのクラスのデータ型になります。
- パラメータが別のデータベースに対応する Java-SQL インスタンスであり、そのパラメータをメンバ参照に含める場合は、Java-SQL インスタンスに対応するクラス名を両方のデータベースに追加する必要があります。これ以外の場合は、例外が発生します。

ランタイムの結果

- メソッド呼び出しのランタイム結果は次のようになります。
    - メソッド呼び出しがランタイム値 null のメンバ参照 (null インスタンスのメンバへの参照) を指定すると、結果は null になります。
    - メソッド呼び出しが型の値を返すメソッドを示すメンバ参照を指定すると、結果はそのメソッドからの戻り値になります。
    - メソッド呼び出しが void 静的メソッドを示すメンバ参照を指定すると、結果は null 値になります。
    - メソッド呼び出しがクラスのインスタンスの void インスタンス・メソッドを示すメンバ参照を指定すると、結果はそのインスタンスへの参照になります。
    - メソッド呼び出しとメソッド呼び出しの結果は、同じデータベースに対応しています。
    - Adaptive Server は、Java 型がスカラであるメソッドに対して null 値をパラメータ値として渡しません。

用語解説

この用語解説では、このマニュアルで使用されている Java と Java-SQL の用語について説明します。Adaptive Server と SQL の用語については、『ASE 用語解説』を参照してください。

Java Development Kit (JDK)

Sun Microsystems が提供するツールキット。オペレーティング・システムからの Java プログラムの書き込みとテストを可能にします。

Java-SQL 変数 (Java-SQL variable)

データ型が Java-SQL クラスの SQL 変数。

Java-SQL カラム (Java-SQL column)

データ型が Java-SQL クラスの SQL カラム。

Java-SQL クラス (Java-SQL class)

public の Java クラス。Adaptive Server システムにインストールされています。一連の変数定義とメソッドで構成されています。

クラス・インスタンスは、クラスの各フィールドのインスタンスから構成されています。クラス・インスタンスは、クラス名によって強力に型付けされます。

サブクラスは、(最大でも) 1 つの他のクラスに拡張するために宣言されるクラスです。この他のクラスは、サブクラスの直接スーパークラスと呼ばれています。サブクラスには、直接および間接のスーパークラスのすべての変数とメソッドがあり、その変数とメソッドを区別なく使用できます。

Java-SQL データ型のマッピング (Java-SQL datatype mapping)

Java と SQL の間のデータ型変換。「Java と SQL の間のデータ型のマッピング」(146 ページ) 参照。

Java アーカイブ (Java archive: JAR)

単一ファイル内にクラスを収集する、プラットフォームに依存しないフォーマット。

Java オブジェクト (Java object)

Java クラスのインスタンス。Java VM の格納領域に含まれています。SQL で参照される Java インスタンスは、Java カラムの値または Java オブジェクトのどちらかです。

Java 仮想マシン (JVM)

サーバで Java を処理する、Java のインタプリタ。SQL 実装 によって呼び出されます。

Java データベース・コネクティビティ (Java Database Connectivity: JDBC)

Java-SQL API。標準の Java クラス・ライブラリの一部で、Java アプリケーションの開発を管理します。JDBC は、ODBC と同様の機能を提供します。

Java データ型 (Java datatypes)
ユーザ定義または JavaSoft API からの Java クラス、またはプリミティブ Java データ型。たとえば、boolean、byte、short、int などです。

Java ファイル (Java file)
"java" タイプのファイル。たとえば、*myfile.java* などです。このファイルは Java のソース・コードを含みます。「クラス・ファイル (class file)」と「Java アーカイブ (Java archive: JAR)」参照。

Java メソッドのシグニチャ (Java method signature)
Java メソッドの各パラメータの Java データ型。

Pluggable Component Adaptor/ JVM
Adaptive Server と JVM の間の要求を管理する Sybase のコンポーネント。

Pluggable Component Interface (PCI)
Adaptive Server の Java フレームワーク。PCA/JVM とともに使用することで、Adaptive Server を商用の JVM とともに使用できるようになります。

Pluggable Component Interface (PCI) Bridge
Adaptive Server コンポーネントで PCI の一部。JVM プラグインと Adaptive Server の間の対話を可能にします。

public
Public フィールドと Public メソッド (Java で定義)。

SQL-Java データ型のマッピング (SQL-Java datatype mapping)
Java と SQL の間のデータ型変換。「Java と SQL の間のデータ型のマッピング」(146 ページ) 参照。

SQL 関数シグニチャ (SQL function signature)
SQLJ 関数の各パラメータの SQL データ型。

SQL プロシージャ・シグニチャ (SQL procedure signature)
SQLJ プロシージャの各パラメータの SQL データ型。

Unicode
多くの言語をサポートする、ISO 10646 で定義された 16 ビットの文字セット。

インスタンス・メソッド (instance method)
呼び出されたメソッドで、クラスの特定のインスタンスを参照するもの。

インストールされたクラス (installed classes)
installjava ユーティリティによって Adaptive Server システムに保存された Java クラスと Java メソッド。

インタフェース (interface)
メソッド宣言の集まりを指す名称。クラスがインタフェース内で宣言されたすべてのメソッドを定義する場合、クラスはそのインタフェースを実装できます。

可視 (visible)

SQL システムにインストールされた Java クラスは、public で宣言された場合に SQL 内で可視になり、Java インスタンスのフィールドとメソッドは、public で宣言され、かつ、マッピング可能な場合に、SQL 内で可視になります。可視のクラス、フィールド、メソッドは SQL で参照できます。その他の、private、protected、friendly のクラスを含むクラス、フィールド、メソッドは SQL では参照できません。また、private、protected、friendly、または、マッピング可能でないフィールドとメソッドも SQL で参照できません。

関連付けられている JAR (associated JAR)

-jar オプションを指定して installjava で class/jar をインストールすると、JAR はデータベース内に保持され、データベース内で、クラスは関連付けられている JAR とリンクされます。「保持された JAR (retained JAR)」参照。

外部化 (externalization)

Java インスタンスの外部化とは、クラスがインスタンスを再構築できる十分な情報を含んだバイト・ストリームです。外部化可能なインタフェースで定義されます。すべての Java-SQL クラスは、外部化可能か直列化可能のどちらかに設定してください。「直列化 (serialization)」参照。

クラス (class)

Java プログラムの基本的な要素で、一連のフィールド宣言とメソッドから構成されています。クラスはマスタ・コピーであり、各クラスの個々のインスタンスの動作と属性を決定します。クラス定義とはアクティブなデータ型の定義です。一連の有効な値を指定し、それらの値を処理する一連のメソッドを定義します。「クラス・インスタンス (class instance)」参照。

クラス・インスタンス (class instance)

クラスのデータ型の値。これにはそのクラスの各フィールドの値が入り、そのクラスのすべてのメソッドを受け入れます。

クラス・ファイル (class file)

「クラス」タイプのファイル。たとえば、*myclass.class* などです。このファイルには、Java クラスのバイトコードをコンパイルしたものが含まれています。「Java ファイル (Java file)」と「Java アーカイブ (Java archive: JAR)」参照。

クラス・メソッド (class method)

「静的メソッド (static method)」参照。

サブクラス (subclass)

ある階層内の別のクラスの下位にあるクラス。上位にあるクラスから属性と動作を継承します。サブクラスは、スーパークラスと区別なく使用できます。サブクラスの上位クラスは、直接スーパークラスです。「スーパークラス (superclass)」、「範囲を狭める変換 (narrowing conversion)」、「範囲を広げる変換 (widening conversion)」参照。

スーパークラス (superclass)

ある階層内で、1 つまたは複数のクラスの上位にあるクラス。下位のクラスに属性と動作を渡します。サブクラスと区別なく使用することはできません。「サブクラス (subclass)」、「範囲を狭める変換 (narrowing conversion)」、「範囲を広げる変換 (widening conversion)」参照。

静的メソッド (static method)

オブジェクトを参照せずに呼び出されたメソッド。静的メソッドは、クラスのインスタンスではなく、クラス全体に影響します。「クラス・メソッド」とも呼びます。

宣言されたクラス (declared class)
Java-SQL データ項目の宣言されたデータ型。ランタイム値のデータ型、またはスーパー型です。

正しい形式のドキュメント (well-formed document)
XML の場合、正しい形式のドキュメントには、開始タグと終了タグの両方がそろったすべての要素、引用符で囲まれた属性値、正しくネストされたすべての要素が必要です。

直列化 (serialization)
Java インスタンスの直列化とは、クラスを識別してインスタンスを再構築するための十分な情報を含んだバイト・ストリームです。すべての Java-SQL クラスは、外部化可能か直列化可能のどちらかに設定してください。「外部化 (externalization)」参照。

データ型のマッピング (datatype mapping)
Java と SQL の間のデータ型変換。

同義クラス (synonymous classes)
同じ完全修飾名を持ち、別のデータベースにインストールされている Java-SQL クラス。

範囲を狭める変換 (narrowing conversion)
クラス・インスタンスへの参照を、そのクラスのサブクラスのインスタンスへの参照へ変換する Java オペレーション。このオペレーションは、convert 関数を使用して SQL で記述されます。「範囲を広げる変換 (widening conversion)」参照。

範囲を広げる変換 (widening conversion)
クラス・インスタンスへの参照を、そのクラスのスーパークラスのインスタンスへの参照に変換する Java オペレーション。このオペレーションは、convert 関数を使用して SQL で記述されます。「範囲を狭める変換 (narrowing conversion)」参照。

バイトコード (bytecode)
Java のソース・コードをコンパイルしたもの。Java VM で実行されます。

パッケージ (package)
関連するクラスのセット。クラスは、パッケージを指定したり、匿名のデフォルト・パッケージの一部となります。クラスは、Java の import 文を使用して、クラスを参照できる他のパッケージを指定できます。

プロシージャ (procedure)
SQL のストアド・プロシージャ、または、*void* 結果タイプの Java メソッド。

変数 (variable)
Java では、変数はクラス、クラスのインスタンス、またはメソッドに対してローカルです。静的に宣言された変数は、クラスに対してローカルです。クラスで宣言されたそれ以外の変数は、クラスのインスタンスに対してのローカルです。このような変数は、クラスのフィールドと呼ばれます。メソッドで宣言された変数は、メソッドに対してローカルです。

保持された JAR (retained JAR)
「関連付けられている JAR (associated JAR)」参照。

マッピング可能 (mappable)

Java のデータ型は、次のどちらかの場合にマッピング可能となります。

- 表 9-3 (146 ページ) の最初の列にリストされているデータ型
- Adaptive Server システムにインストールされている public の Java-SQL クラス

SQL のデータ型は、次のどちらかの場合にマッピング可能となります。

- 表 9-4 (147 ページ) の最初の列にリストされているデータ型
- Adaptive Server システムに組み込まれているか、インストールされている public の Java-SQL クラス

Java のメソッドは、パラメータと結果のデータ型がすべてマッピング可能な場合に、マッピング可能になります。

メソッド (method)

一連の命令。Java クラスに含まれ、タスクを実行します。メソッドは静的に宣言でき、その場合はクラス・メソッドと呼ばれます。そうでない場合は、インスタンス・メソッドと呼ばれます。クラス・メソッドは、クラス名かクラスのインスタンスの名前でメソッド名を修飾して、参照することができます。インスタンス・メソッドは、クラスのインスタンスの名前でメソッド名を修飾して、参照されます。インスタンス・メソッドのメソッド本体は、そのインスタンスに対してローカルな変数を参照できます。

割り当て (assignment)

データ転送の一般的な用語。select、fetch、insert、update の Transact-SQL コマンドで指定します。割り当てとは、ターゲットとなるデータ項目にソースの値を設定することです。

索引

記号

>> (二重山カッコ)
　修飾、Java フィールドと Java メソッド　151

A

Adaptive Server
　プラグイン　35, 88
Adaptive Server 15.0.3 およびそれ以降の変更　4
ADT レベルでマッピング可能なデータ型　105
alter table
　構文　35
　コマンド　35
ANSI 規格　6
array 引数　16

C

called on null input パラメータ　90
case 式　44, 93
ClassLoader の動作　5
convert 関数　43, 133
create procedure (SQLJ) コマンド　94, 97
create table コマンド、構文　33, 34

D

deterministic パラメータ　90, 95
distinct キーワード　52
drop function コマンド　93
dynamic result sets パラメータ　95

E

external name パラメータ　95
extractjava ユーティリティ　29

G

group by 句　52

I

in パラメータ　97
inout パラメータ　97
installjava ユーティリティ　24, 25
　-f オプション　26
　-j オプション　26
　-new オプション　27
　update オプション　27
　構文　25

J

JAR
　非圧縮、インストール　26
JAR の削除　29
JAR ファイル
　作成　26
　保持　26
Java API　8
　Sybase サポート　8
　サポートされているパッケージ　138–141
Java Development Kit　7
Java Runtime Environment　23
Java インスタンス、表現　39
Java オブジェクト　35
Java オペレーション、SQL からの呼び出し　8
Java 仮想マシン
　サポート　7, 24
Java 環境
　JVM プラグ可能コンポーネント　12
　Pluggable Component Adapter (PCA/JVM)　13
　Pluggable Component Interface (PCI)　14
　Pluggable Component Interface (PCI) Bridge　14
　コンポーネント　11

Java クラス
SQLJ の例　86
インストール　25–28
更新　27
サポートされる　8
データ型　3, 33
部分型　43
他のクラスの参照　28
保持　30
ユーザ定義　8, 24
ランタイム　24
Java クラス・データ型　92
Java クラスの配布方式　4
Java データ型
ADT レベルでマッピング可能　105
オブジェクト・レベルでマッピング可能　104
結果セット・レベルでマッピング可能　105
出力レベルでマッピング可能　105
直接マッピング可能　104
Java と SQL データ型のマッピング　104
Java のデバッグ　115
Java の有効化　25
Java 配列　98
Java プリミティブ・データ型　92
Java メソッド
type　49
void　49
インスタンス　50
型　48
コマンド main　108
参照による呼び出し　38, 52
静的　51
呼び出し　37, 86
例外　38
Java メソッドのシグニチャ　90, 96
java.lang.Thread クラス、慎重な使用　135
java.net、ネットワーク・アクセス　119
java.sql　141
java.sql メソッド、サポートされていない　140
Java-SQL
オブジェクトの転送　133
オブジェクトの転送、クライアント　133
カラム　40, 53
カラムの参照　150
カラムの宣言　149
関数結果　40
クラス名　148
作成、テーブル　33
識別子　147
静的変数　55
名前　32
パッケージ名　148
パラメータ　40, 53
変数　40, 53
変数の宣言　149
メソッド呼び出し　152
メソッド、サポートされていない　140
メンバ参照　151
Java-SQL カラム
格納オプション　34
Java-SQL クラス
インストール　25–28
複数のデータベース　55
Java、SQL、一緒に使用　8
jdb デバッガ　117
JDBC　65–81
JDBCExamples クラス　68
インタフェース　9
概念　66
クライアント側　66
コネクション　70
サーバ側　66
接続デフォルト　67
接続の取得　70
データのアクセス　67
バージョン・サポート　24
パーミッション　67
用語　66
JDBC ドライバ　24, 141
クライアント側　66
サーバ側　66
JDBC 標準のデータ型マッピング　104
JDBCExamples クラス　76–81
概要　68
メソッド　68–74

L

language java パラメータ　95

M

modifies sql data パラメータ　90, 95

N

null 値
 case 文 93
 SQLJ 関数内 92
null、Java-SQL 45–48
 使用、convert 関数 47
 引数、メソッドに対する 46
number 引数 16

O

order by 句 52
out パラメータ 97

P

parameter style java パラメータ 95
PCA/JVM 13
PCI Bridge 14
PCI メモリ・プール 14
 サイズの変更 14
 マルチエンジン環境 15

Q

Q/A 6

R

remove java コマンド 29, 149
ResultSet
 マッピング可能なデータ型 105
returns null on null input パラメータ、Java 句 90

S

set コマンド
 Java メソッド内で許可された 144
 更新 51
sp_depends システム・プロシージャ 104
sp_help システム・プロシージャ 104
sp_helpjava
 utilitysp_helpjava 28
 構文 28
sp_helpjava システム・プロシージャ 104
sp_helprotect システム・プロシージャ 104
SQL
 関数のシグニチャ 89
 式、Java オブジェクトを含める 8
 プロシージャ・シグニチャ 95
 ラッパ 83, 87
SQL の規格 6
SQL ループ、回避 136
SQLJ create procedure コマンド 94
SQLJ 関数 89–94
 削除 93
 情報の表示 104
SQLJ ストアド・プロシージャ 94–96, 103
 SQL データの変更 96
 機能 94
 削除 103
 情報の表示 104
 入出力パラメータの使用 97
SQLJ の実装
 SQLJ と Sybase の相違点 109
 Sybase による定義 110
 一部サポートされている機能 110
 サポートされていない機能 110
SQLJ 標準 84
SQLJExamples クラス 111
SQLJExamples.bestTwoEmps() メソッド 86
SQLJExamples.correctStates() メソッド 86, 96
SQLJExamples.job() メソッド 86
SQLJExamples.region() メソッド 86, 91
string 引数 16
style java キーワード 95
switch 引数 16
Sybase Central
 SQLJ 関数またはプロシージャの作成 88
 SQLJ プロシージャおよび関数の管理 88
 SQLJ ルーチンのプロパティの表示 88
sybpcidb データベース 15
 値の変更 17
 システム・テーブル 16
 設定値 16
 デフォルト値の復元 20

T

Transact-SQL
 コマンド、Java メソッド内 142

U

Unicode 48
union 演算子 52

V

void メソッド 96

W

where 句 43, 50, 53

あ

アットマーク (@) 89
暗黙の Java メソッド・シグニチャ 106

い

一時的なデータ項目 40
インスタンス・メソッド 50
インストール
  Java クラス 25, 28
  非圧縮 JAR 26
インストールされたクラスの再配列 30

え

永続的なデータ項目 40

お

オブジェクト・レベルでマッピング可能なデータ型 104
オプション
  external name 90
  language java 90
  parameter style java 90

か

外部化 149
格納オプション
  ロー内 34
カラム
  参照 150
  宣言 149
カラムの参照 150
カラムの宣言 149
カラムのデータ型、要件 33

き

規格の仕様 6

く

区切り識別子 147
クライアント
  bcp 133, 134
  isql 133
クラス間引数 59
クラスの削除 29
クラスの部分型 43–45
クラス名 148
クラス。「Java クラス」参照

け

結果セット 107
検索順序
  関数の種類 91

こ

更新、Java オブジェクト 35
コマンド
  create procedure SQLJ 97
  create table 33, 34
  drop function 93
  SQLJ create procedure 94
  SQLJ create 関数 89
コマンド main メソッド 108
コンストラクタ 35, 49
コンストラクタ・メソッド 35
コンパイル時データ型 44

さ

削除　35, 103
- Java オブジェクト　35

作成
- テーブル　33

参照
- fields　36

サンプル・クラス　60–62
- address　60
- Address2Line　61
- JDBCExamples　68–81
- Misc　62

し

識別子　147
- 区切り　147

システム・プロシージャ
- helpjava　28
- sp_depends　104
- sp_help　104
- sp_helpjava　104
- sp_helprotect　104

出力レベルでマッピング可能なデータ型　105

順序演算子　52

使用
- Java クラス　31, 60
- Java と SQL を一緒に使用　8

情報の表示
- インストールされた JAR について　28
- インストールされたクラスについて　28

す

スーパー型　43

せ

静的変数　55

静的メソッド　51, 87, 94

セキュリティ
- SQLJ ルーチン　84

接続 (複数) の取得　70

設定オプション
- PCA/JVM　17
- PCI Bridge　17
- 実行中のサーバでの値の変更　18
- デフォルト値の復元　20

選択、Java オブジェクト　35

そ

挿入
- Java オブジェクト　35

た

ダウンロード
- インストールされた JAR　29
- インストールされたクラス　29

ち

直接マッピング可能なデータ型　104

直列化　149, 150

つ

追加情報
- Java 関連　10

て

データ型
- Java クラス　3
- コンパイル時　44
- 変換　133
- メソッド呼び出し　152
- ランタイム　44

データ型のマッピング　42, 104, 146–147

データ型変換　133

データベースで Java を使用するときの制限　10

データベースにおける Java
Q/A 6
機能 2
主要機能 6
準備 23–30
メリット 1
テーブルの定義 86
デバッガ
設定 116
付加 117
デバッグ
Java 115–117
テンポラリ・データベース 60

と

等号 (=) 演算子 52

な

長さがゼロの文字列 49
名前、Java-SQL 内 32
大文字と小文字 32
長さ 32

に

二重山カッコ
Java フィールドと Java メソッドの修飾 151
修飾、Java フィールドと Java メソッド 36

ね

ネイティブ・メソッド、PCA/JVM 137
ネットワーク・アクセス、java.net 119

は

パーミッション
Java 32
JDBC 67
SQLJ ルーチン 84
パッケージ名 148
パフォーマンス、向上 134
パラメータ
deterministic 95
external name 95
inout 97
input 97
language java 95
modifies sql data 95
not deterministic 95
output 97
parameter style java 95
範囲を狭める変換 44

ふ

ファイル・アクセス
java.io の使用 119
java.net の使用 126
ディレクトリの指定 121
ファイルを作成するための規則 126
ファイルを開くための規則 124
ユーザ ID とパーミッション 119
複数のデータベース 57
部分型 43
プロシージャ
SQLJ ルーチンの作成 83

へ

ヘッドレス・モード 5, 7
変換 133
範囲を狭める 44
範囲を広げる 43
変数 149
静的 55
データ型 33
割り当てられた値 36
変数の宣言 149

ま

マッピング、データ型 146–147

め

明示的な Java メソッド・シグニチャ　106
メソッド
　SQLJExamples.bestTwoEmps()　86
　SQLJExamples.correctStates()　86, 96
　SQLJExamples.job()　86
　SQLJExamples.region()　86
　「XQL メソッド」参照
　ランタイムの結果　153
　例外　38
メソッドのオーバロード　107, 152
メソッド呼び出し　152
　データ型　152
メンバ参照　151

も

文字セット
　Adaptive Server プラグイン　88
　Unicode　35, 43, 88
文字列データ　48
　長さがゼロ　49

よ

呼び出し
　Java メソッド　37, 86
　Java メソッド、SQLJ 使用　86, 87
　Java メソッド、直接起動　86
　SQL、Java からの　141, 145

ら

ランタイム
　データ型　44
ランタイム Java クラス　24
　ロケーション　24
ランタイム環境　23

れ

例
　SQLJ ルーチン　85
例外　38

わ

ワーク・データベース　60
割り当て　131
割り当てのプロパティ
　Java-SQL データ項目　40