



Sybase® jConnect for JDBC
プログラマーズ・リファレンス

jConnect for JDBC
バージョン 4.2 および 5.2

ドキュメント ID: 38164-01-0520-01

改訂 : 1999 年 10 月

Copyright © 1989-1999 by Sybase, Inc. All rights reserved.

このマニュアルの内容は、予告なく変更されることがありますが、Sybase, Inc. およびその関連会社では内容の変更に関して一切の責任を負いません。

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。

マニュアルの注文

マニュアルの注文を承ります。ご希望の方は、サイバース株式会社営業部または代理店までご連絡ください。マニュアルの変更は、弊社の定期的なソフトウェア・リリース時にのみ提供されます。

Sybase の商標

Sybase、SYBASE のロゴ、AnswerBase、Embedded SQL、ENTERPRISECONNECT、Gain Momentum、Navigation Server、ObjectConnect、ObjectCycle、OMNICONNECT、Open Client、Open ClientConnect、Power Dynamo、PowerBuilder、Powersoft、Replication Agent、Replication Driver、Replication Server、SQL Central、SQL Remote、Sybase IQ、Sybase SQL Anywhere、Sybase System 11、Sybase WAREHOUSEWORKS、Sybase どこでも SQL、SyBooks は、米国法人 Sybase, Inc. の登録商標です。Backup Server、Client Library、DBLibrary、e-Anywhere、EIP、Enterprise Information Portal、Enterprise Messaging Services、Fastbuild、Financial Server、Information Anywhere、Mainframe Connect、MASS DEPLOYMENT、media.splash、Net Gateway、Net Library、NetImpact、Open SERVERCONNECT、QuickStart DataMart、SADG、SQL Debug、SQL Server、SQL Servermanager、SQL Servermonitor、Support Plus、Support Plus Lite、The Architecture for Change、web.sql は、米国法人 Sybase, Inc. の商標です。

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、各社の商標または登録商標場合があります。

権利について

米国政府による使用、複写、開示は、国防総省の契約に関して DFARS 52.227-7013 の項目 (c) (1) (ii) に明記されている制約条項、およびその他の政府機関の契約に関しては FAR 52.227-19 (a)-(d) に明記されている制約条項に従います。

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608

目次

はじめに.....	vii
第 1 章	概要..... 1
	JDBC の概要..... 2
	jConnect の概要..... 4
第 2 章	プログラミング情報..... 5
	jConnect の設定..... 6
	jConnect バージョンの設定..... 6
	jConnect ドライバの呼び出し..... 10
	接続の確立..... 12
	接続プロパティの設定..... 12
	Adaptive Server Enterprise への接続..... 18
	Adaptive Server Anywhere への接続..... 19
	JNDI を使用してサーバに接続する方法..... 21
	カスタム・ソケット・プラグインの実装..... 26
	SYB SOCKET_FACTORY 接続プロパティ..... 27
	カスタム・ソケットの作成と設定..... 28
	国際化とローカライゼーションの処理..... 31
	jConnect 文字セット・コンバータ..... 31
	データベースの作業..... 37
	高可用性フェールオーバー・サポートの実装..... 37
	サーバ間のリモート・プロシージャ・コールの実行..... 42
	データベース・メタデータへのアクセス..... 44
	結果セットでのカーソルの使用方法..... 45
	バッチ更新のサポート..... 56
	ストアド・プロシージャの結果セットによるデータベースの更新..... 58
	データ型の作業..... 60
	高度な機能の実装..... 65
	イベント通知の使用方法..... 65
	エラー・メッセージの処理..... 68

	テーブル内のカラム・データとしての Java オブジェクトの格納	
	73	
	動的クラスのロード	78
	JDBC 2.0 Optional Package の拡張機能のサポート	82
	処理上の制約、制限、JDBC 規格外の実装	93
	マルチスレッドに対する調整	93
	ResultSet.setCursorName() の使用	93
	大きなパラメータ値での setLong() の使用	94
	COMPUTE 文の使用	94
	ストアド・プロシージャの実行	94
第 3 章	トラブルシューティング	97
	jConnect でのデバッグ	98
	Debug クラスのインスタンスの取得	98
	アプリケーションのデバッグをオンにする方法	99
	アプリケーションのデバッグをオフにする方法	99
	デバッグ用に CLASSPATH を設定する方法	99
	Debug メソッドの使用	100
	TDS 通信の取得	102
	PROTOCOL_CAPTURE 接続プロパティ	102
	Capture クラスでの pause() メソッドと resume() メソッド	103
	接続不成功エラー	104
	ゲートウェイ接続が拒否される	104
	SQL Server 4.9.2 に接続できない	105
	jConnect アプリケーションでのメモリ使用率	106
	ストアド・プロシージャのエラー	107
	RPC が登録数よりも少ない出力パラメータを返す	107
	ストアド・プロシージャが出力パラメータを返す場合のフェッ チとステータスのエラー	107
	ストアド・プロシージャを非連鎖トランザクション・モードで しか実行できない	107
	カスタム・ソケット実装エラー	109
第 4 章	パフォーマンスおよびチューニング	111
	jConnect パフォーマンスの改善	112
	BigDecimal の再位取り	113
	REPEAT_READ 接続プロパティ	113
	文字セット変換	114
	動的 SQL の準備文のパフォーマンス・チューニング	115
	準備文かストアド・プロシージャかの選択	116
	移植可能なアプリケーションでの準備文	116
	jConnect 拡張機能を持つアプリケーション内の準備文	117
	Connection.prepareStatement()	119

	DYNAMIC_PREPARE 接続プロパティ	119
	SybConnection.prepareStatement()	121
	カーソルのパフォーマンス	122
	LANGUAGE_CURSOR 接続プロパティ	122
第 5 章	jConnect アプリケーションへのマイグレート	125
	jConnect アプリケーションへのマイグレート	126
	jConnect 4.1 へのアプリケーションのマイグレート	126
	jConnect 5.x へのアプリケーションのマイグレート	126
	jConnect 4.2 および 5.2 へのアプリケーションのマイグレート . 127	
	Sybase 拡張機能の変更	129
	変更の例 :	129
	変更されたメソッド名	130
	Debug クラス	130
第 6 章	Web サーバ・ゲートウェイ	133
	Web サーバ・ゲートウェイの概要	134
	TDS トンネリング	134
	jConnect およびゲートウェイの設定	135
	カスケード・ゲートウェイの使用	137
	カスケード・ゲートウェイのインストール	138
	カスケード・ゲートウェイの起動方法	138
	カスケード・ゲートウェイのテスト	139
	index.html ファイルの読み込み	140
	サンプル Isql アプレットの実行	141
	カスケード・ゲートウェイへの接続の定義	142
	TDS トンネリング・サーブレットの使用	143
	TDS トンネリング・サーブレットのシステム稼働条件	144
	サーブレットのインストール	145
	サーブレットの呼び出し	146
	アクティブな TDS セッションのトラッキング	146
	TDS セッションの再開	147
	TDS トンネリングと Netscape Enterprise Server 3.5.1 on Solaris 147	
付録 A	SQL の例外メッセージと警告メッセージ	149
付録 B	jConnect のサンプル・プログラム	165
	IsqlApp の実行	166
	jConnect のサンプル・プログラムとサンプル・コードの実行 ...	169
	サンプル・アプリケーション	169

	サンプル・コード.....	170
索引	173

はじめに

『Sybase jConnect for JDBC プログラマーズ・リファレンス』では、jConnect for JDBC 製品について説明し、この製品を使用してリレーショナル・データベース管理システムに格納されているデータにアクセスする方法について説明します。

対象読者

このマニュアルは、Java プログラミング言語、JDBC、および Sybase 版の SQL である Transact-SQL についての知識を持っているデータベース・アプリケーション・プログラマの方を対象としています。

関連マニュアル

次のマニュアルも参照してください。

- 『jConnect for JDBC インストール・ガイド』
- 『jConnect for JDBC リリース・ノート』
- JDBC への jConnect 拡張機能用の javadoc マニュアル。
JavaSoft の Java Development Kit (JDK) には、ソース・コード・ファイルからコメントを抽出する javadoc スクリプトが含まれています。このスクリプトは、ソース・ファイルから jConnect パッケージのマニュアル部分、クラス、メソッドを抽出するのに使用されています。フル・インストールまたは javadoc オプションを使用して jConnect をインストールするとき、javadoc の情報は *javadocs* ディレクトリに置かれます。

Installation_directory/docs/en/javadocs

表記規則

このマニュアルでは次の表記および構文規則に従います。

- クラス、インタフェース、メソッド、およびパッケージは文中では太字 (**bold Helvetica**) で表示します。次に例を示します。

SybConnection クラス

SybEventHandler インタフェース

setBinaryStream() メソッド

com.sybase.jdbcx パッケージ

- オブジェクト、インスタンス、パラメータは斜体で表記します。次に例を示します。

次の例では *ctx* は **DirContext** オブジェクトです。

eventHdlr は、実装する **SybEventHandler** クラスのインスタンスです。

classes パラメータはデバッグするクラスをリストする文字列です。

- コード例は等幅フォントで表記します。コード例の変数 (ユーザが入力する値を示す語句) は斜体で表記します。次に例を示します。

```
Connection con = DriverManager.getConnection("jdbc:
sybase:Tds:host:port", props);
```

不明な点があるときは

Sybase ソフトウェアがインストールされているサイトには、Sybase 製品の保守契約を結んでいるサポート・センタとの連絡担当の方 (コンタクト・パーソン) を決めてあります。マニュアルだけでは解決できない問題があった場合には、担当の方を通して Sybase のサポート・センタまでご連絡ください。

第 1 章

概要

この章では、jConnect for JDBC を紹介し、その概念とコンポーネントについて説明します。

この項では、次の項目について説明します。

項名	ページ
JDBC の概要	2
jConnect の概要	4

JDBC の概要

Sun Microsystems 社の Java Software Division の JDBC (Java Database Connectivity) は、API (アプリケーション・プログラム・インタフェース) に対応する仕様になっているので、Java アプリケーションは、SQL (Structured Query Language) を使用し、複数のデータベース管理システムにアクセスできます。JDBC ドライバのマネージャは、異なるデータベースに接続する複数のドライバを処理します。

標準の JDBC API には、インタフェースのセットが含まれているので、データベースへの接続のオープン、SQL コマンドの実行、結果の処理が可能です。表 1-1 では、インタフェースについて説明します。

表 1-1: JDBC のインタフェース

インタフェース	説明
java.sql.Driver	データベースの URL 用のドライバを見つける。
java.sql.Connection	特定のデータベースへ接続する。
java.sql.Statement	SQL 文を実行する。
java.sql.PreparedStatement	パラメータを使用する SQL 文を処理する。
java.sql.CallableStatement	データベースのストアド・プロシージャ・コールを処理する。
java.sql.ResultSet	SQL 文の結果を取得する。
java.sql.DatabaseMetaData	接続の DBMS およびデータベースについてのさまざまな情報にアクセスする場合に使用する。
java.sql.ResultSetMetaData	結果セットの属性を示すさまざまな情報にアクセスする場合に使用する。

それぞれのリレーショナル・データベース管理システムには、これらのインタフェースを実装するためのドライバが必要です。すべての JDBC 呼び出しは JDBC ドライバ・マネージャに送信され、指定されたドライバに渡されます。

JDBC ドライバには、次の 4 種類があります。

- タイプ 1 JDBC-ODBC ブリッジ - JDBC 呼び出しを ODBC 呼び出しに変換し、ODBC ドライバに渡します。ODBC ソフトウェアには、クライアント・マシンに常駐する必要があるものがあります。クライアント・データベースのコードも、クライアント・マシンに常駐する場合があります。

- タイプ 2 ネイティブ *API* / 部分 *Java* ドライバ - **JDBC** の呼び出しをデータベース固有の呼び出しに変換します。このドライバは、データベース・サーバと直接通信しますが、クライアント・マシンにバイナリ・コードが必要です。
- タイプ 3 ネット・プロトコル / 全部 *Java* ドライバ - **DBMS** に依存しないネット・プロトコルを使用して、中間層サーバと通信します。中間層のゲートウェイが、要求をベンダ固有のプロトコルに変換します。
- タイプ 4 ネイティブ・プロトコル / 全部 *Java* ドライバ - **JDBC** 呼び出しをベンダ固有の **DBMS** プロトコルに変換し、クライアント・アプリケーションはデータベース・サーバと直接通信できます。

jConnect の概要

jConnect は Sybase のパフォーマンスの高い JDBC ドライバです。
jConnect は次のようなドライバです。

- 3 層環境でのネット・プロトコル／全部 Java ドライバ
- 2 層環境でのネイティブ・プロトコル／全部 Java ドライバ

jConnect が使用するプロトコルは、Adaptive Server および Sybase Open Server アプリケーションのネイティブのプロトコルである、TDS 5.0 (Tabular Data Stream バージョン 5) です。jConnect は、Sybase 製品ファミリーへの最適なコネクティビティを提供する JDBC 標準を実装しています。これによって、次の製品を含む 25 のエンタープライズ・システムと旧世代のシステムにアクセスできます。

- Adaptive Server Enterprise
- Adaptive Server Anywhere
- Adaptive Server IQ (以前は Sybase IQ)
- Replication Server
- OmniConnect

注意 製品の名前を Sybase SQL Server から Adaptive Server Enterprise に変更しましたが、サポートされているバージョンの Sybase SQL Server と Adaptive Server Enterprise を集合的に参照するのに Adaptive Server と Adaptive Server Enterprise という名前を使用する場合があります。

さらに、jConnect for JDBC は DirectConnect を使用して Oracle、AS/400、およびその他のデータ・ソースにもアクセスできます。

また、jConnect は JDBC 1.x もしくは 2.x の仕様外の実装が存在します。詳細については、「[Sybase 固有の処理上の制約、制限及び JDBC 規格外の実装](#)」(93 ページ) をご参照ください。

第 2 章

プログラミング情報

この章では jConnect for JDBC を構成する、基本的なコンポーネントおよびプログラミング要件について説明します。jConnect ドライバの呼び出し、接続プロパティの設定、データベース・サーバへの接続の方法を説明します。また、jConnect の機能の使い方についても説明しています。

注意 JDBC プログラミングの情報については、下記の URL をアクセスしてください。

<http://java.sun.com/jdbc>

JDBC 1.0 用の『JDBC Guide: Getting Started』マニュアルは、下記の URL をアクセスしてください。

<http://java.sun.com/products/jdk/1.1/docs/guide/jdbc>

JDBC 2.0 用の『JDBC Guide: Getting Started』マニュアルは、下記の URL をアクセスしてください。

<http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/>

この章では、次の項目について説明します。

項名	ページ
jConnect の設定	6
接続の確立	12
カスタム・ソケット・プラグインの実装	26
国際化とローカライゼーションの処理	31
データベースの作業	37
高度な機能の実装	65
Sybase 固有の処理上の制約、制限及び JDBC 規格外の実装	93

jConnect の設定

この項では jConnect の使用を開始する前に必要な作業について説明します。

jConnect バージョンの設定

jConnect にはいくつかのバージョンがあります。バージョン設定によって次の項目を決定します。

- LANGUAGE 接続プロパティのデフォルト値
- 使用可能なバージョン固有の機能
- CHARSET 接続プロパティによって文字セットが指定されていない場合のデフォルト文字セット
- CHARSET_CONVERTER 接続プロパティのデフォルト値
- CANCEL_ALL 接続プロパティのデフォルト値。これを使用して **Statement.cancel()** の動作を設定する。デフォルトでは **Statement.cancel()** は、呼び出された対象のオブジェクト、および実行を開始して結果を待っているその他の **Statement** オブジェクトをキャンセルする。

表 2-1 に、使用できるバージョン設定とその機能をリストします。

表 2-1: jConnect バージョン設定とその機能

バージョン定数	機能	コメント
VERSION_5	<ul style="list-style-type: none"> • LANGUAGE 接続プロパティのデフォルト値は null。 • CHARSET 接続プロパティに文字セットの指定がない場合、jConnect はデータベースのデフォルトの文字セットを使用する。CHARSET_CONVERTER のデフォルト値は PureConverter クラス。 • デフォルトでは、Statement.cancel() は、呼び出された対象の Statement オブジェクトだけをキャンセルする。 • JDBC 2.0 メソッドは Java オブジェクトをカラム・データとして格納および検索する。 	<p>jConnect version 5.x では、デフォルトは VERSION_5。</p> <p>詳細については、VERSION_4 のコメントを参照。</p>

バージョン定数	機能	コメント
VERSION_4	<ul style="list-style-type: none"> LANGUAGE 接続プロパティのデフォルト値は null。 CHARSET 接続プロパティに文字セットの指定がない場合、jConnect はデータベースのデフォルトの文字セットを使用する。CHARSET_CONVERTER のデフォルト値は PureConverter クラス。 デフォルトでは、Statement.cancel() は、呼び出された対象の Statement オブジェクトだけをキャンセルする。 JDBC 2.0 メソッドは Java オブジェクトをカラム・データとして格納および検索する。 	<p>jConnect version 4.x 以前では、デフォルトは VERSION_2。</p> <p>サーバからのメッセージは、ローカル環境変数内の言語設定に従ってローカライズされる。サポートされている言語は次のとおり。中国語、英語、フランス語、ドイツ語、日本語、韓国語、ポルトガル語、スペイン語</p> <p>Statement.cancel() のデフォルト動作は JDBC に準拠している。</p> <p>CANCEL_ALL 接続プロパティを使用して Statement.cancel() の動作を設定する。詳細については、「CANCEL_ALL 接続プロパティ」(9 ページ)を参照。</p> <p>カラム・データとしての Java オブジェクトの詳細については、「テーブル内のカラム・データとしての Java オブジェクトの格納」(73 ページ)を参照。</p>
VERSION_3	<ul style="list-style-type: none"> LANGUAGE 接続プロパティのデフォルト値は us_english。 CHARSET 接続プロパティに文字セットの指定がない場合は、jConnect はデータベースのデフォルト文字セットを使用する。 CHARSET_CONVERTER のデフォルト値は PureConverter クラス。 デフォルトでは、Statement.cancel() は呼び出された対象のオブジェクト、および実行を開始して結果を待っているその他の Statement オブジェクトをキャンセルする。 	<p>デフォルトは VERSION_2。</p> <p>VERSION_2 のコメントを参照。</p>

バージョン定数	機能	コメント
VERSION_2	<ul style="list-style-type: none"> LANGUAGE 接続プロパティのデフォルト値は us_english。 CHARSET 接続プロパティに文字セットの指定がない場合のデフォルト文字セットは iso_1 である。 CHARSET 接続プロパティにマルチバイトまたは 8 ビット文字セットが指定されていない場合の CHARSET_CONVERTER のデフォルト値は TruncationConverter クラスであり、指定されている場合のデフォルト CHARSET_CONVERTER は PureConverter クラスである。 デフォルトでは、Statement.cancel() は呼び出された対象のオブジェクト、および実行を開始して結果を待っているその他の Statement オブジェクトをキャンセルする。 	<p>jConnect version 2.x のデフォルトのバージョン設定は VERSION_2。</p> <hr/> <p>注意 jConnect version 5.x のデフォルトのバージョン設定は VERSION_5。</p> <hr/> <p>LANGUAGE 接続プロパティは、jConnect とサーバからのメッセージを表示する言語を決定する。</p> <p>CHARSET 接続クラスと CHARSET_CONVERTER 接続クラスについては、「jConnect 文字セット・コンバータ」(31 ページ) を参照。</p> <p>Statement.cancel() の VERSION_2 のデフォルト動作は JDBC に準拠していない。CANCEL_ALL 接続プロパティを使用して Statement.cancel() の動作を設定する。詳細については、「CANCEL_ALL 接続プロパティ」(9 ページ) を参照。</p>

バージョン値は、**SybDriver** クラスからの定数値です。バージョン定数を参照するときには、次の構文を使用します。

```
com.sybase.jdbcx.SybDriver.VERSION_5
```

jConnect のバージョンを設定するには **SybDriver.setVersion()** を使用します。次のコード例は、jConnect ドライバをロードしてバージョンを設定する方法を示します。

jConnect 4.x の場合：

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
    Class.forName ("com.sybase.jdbc.SybDriver").newInstance();
sybDriver.setVersion
    (com.sybase.jdbcx.SybDriver.VERSION_4);
DriverManager.registerDriver(sybDriver);
```

jConnect 5.x の場合：

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
    Class.forName
    ("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
sybDriver.setVersion
    (com.sybase.jdbcx.SybDriver.VERSION_5);
DriverManager.registerDriver(sybDriver);
```

setVersion() は、バージョン設定を変更するために何度も呼び出すことができます。新しい接続は、接続が行われるときのバージョン設定に対応する動作を継承します。セッション中にバージョン設定を変更しても、現在の接続には影響しません。

次の項で説明するように、**JCONNECT_VERSION** は **SybDriver** バージョン設定を上書きして、特定の接続に対して異なるバージョン設定を指定するために使用できます。

JCONNECT_VERSION 接続プロパティ

JCONNECT_VERSION によって、特定のセッションに使用される **jConnect** バージョン設定を指定できます。**JCONNECT_VERSION** は、必要な特性に応じて整数値 "2"、"3"、"4"、または "5" に設定できます (表 2-1 を参照してください)。

CANCEL_ALL 接続プロパティ

CANCEL_ALL は、**Statement.cancel()** メソッドの動作を指定するためのブール値の接続プロパティです。

注意 **jConnect 4.0** 以前のバージョンでは、**CANCEL_ALL** のデフォルト設定は "true" です。**jConnect 4.1** では、**JDBC** 仕様に従うために、接続プロパティ **JCONNECT_VERSION** を "4" 以上に設定した場合の **CANCEL_ALL** のデフォルト設定は "false" になります。

CANCEL_ALL の設定は、**Statement.cancel()** に対して次のような効果があります。

- **CANCEL_ALL** が "false" の場合は、**Statement.cancel()** は、呼び出された対象の **Statement** オブジェクトだけをキャンセルします。したがって、**stmtA** が **Statement** オブジェクトである場合は、**stmtA.cancel()** はデータベース内の **stmtA** に含まれる **SQL** 文の実行をキャンセルしますが、ほかの文への影響はありません。**stmtA** は、キャッシュ内で実行を待っているか、実行を開始して結果を待っている場合にキャンセルされます。
- **CANCEL_ALL** が "true" の場合、**Statement.cancel()** は、呼び出された対象のオブジェクトだけでなく、すでに実行を開始していて結果を待っている、同じ接続上にあるその他の **Statement** オブジェクトもキャンセルします。

次の例では、**CANCEL_ALL** を "false" に設定します。この例では、*props* は接続プロパティを指定するための **Properties** オブジェクトです。

```
...
props.put("CANCEL_ALL", "false");
```

注意 サーバ上で実行を開始しているかどうかに関係なく、接続されているすべての **Statement** オブジェクトの実行をキャンセルする場合は、拡張メソッド **SybConnection.cancel()** を使用します。

jConnect ドライバの呼び出し

Sybase jConnect ドライバの登録と呼び出しには、次のいずれかの方法を使用します。

方法 1

jConnect 4.x の場合 :

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance();
```

jConnect 5.x の場合 :

```
Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
```

方法 2

jConnect ドライバを **jdbc.drivers** システム・プロパティに追加する方法。**DriverManager** クラスは、初期化時に、**jdbc.drivers** にリストされているドライバをロードします。先に示した方法の方が効率的です。このプロパティには、複数のドライバをコロン(:)で区切ってリストできます。次のコード例は、プログラム内で **jdbc.drivers** へドライバを追加する方法を示します。

jConnect 4.x の場合 :

```
Properties sysProps = System.getProperties();
String drivers = "com.sybase.jdbc.SybDriver";
String oldDrivers =
sysProps.getProperty("jdbc.drivers");
if (oldDrivers != null)
    drivers += ":" + oldDrivers;
sysProps.put("jdbc.drivers", drivers.toString());
```

jConnect 5.x の場合 :

```
Properties sysProps = System.getProperties();
String drivers = "com.sybase.jdbc2.jdbc.SybDriver";
String oldDrivers =
sysProps.getProperty("jdbc.drivers");
```

```
if (oldDrivers != null)
    drivers += ":" + oldDrivers;
sysProps.put("jdbc.drivers", drivers.toString());
```

注意 `System.getProperties()` は Java アプレットには使用できません。
`Class.forName()` メソッドを使用してください。

接続の確立

この項では、jConnect を使用して Adaptive Server Enterprise または Adaptive Server Anywhere データベースへの接続を確立する方法について説明します。

接続プロパティの設定

表 2-2 に jConnect の接続プロパティをリストし、そのデフォルト値を示します。接続を作成する前に接続プロパティを設定する必要があります。

ドライバ接続プロパティの設定には、2 つの方法があります。

- アプリケーションで **DriverManager.getConnection()** メソッドを使用する
- URL を定義するときに設定する

注意 URL に設定されたドライバ接続プロパティは、**DriverManager.getConnection()** メソッドを使用してアプリケーションに設定された、対応するドライバ接続プロパティを上書きしません。

任意のドライバについて現在のプロパティのリストを取得するには、**Driver.getDriverPropertyInfo(String url, Properties props)** を使用してください。これは **DriverPropertyInfo** オブジェクトの配列を返します。配列には次のものがリストされます。

- ドライバ・プロパティ
- ドライバ・プロパティが基づいている現在の設定
- 渡された URL および **props**

ドライバ接続プロパティ名は大文字と小文字を区別しません (jConnect は **String.equalsIgnoreCase(String)** メソッドを使用してプロパティ名を比較します)。

表 2-2: 接続プロパティ

プロパティ	説明	デフォルト値
APPLICATIONNAME	ユーザ定義プロパティ。このプロパティに指定された値を解釈するよう、サーバ側をプログラムすることができる。	Null

プロパティ	説明	デフォルト値
CANCEL_ALL	Statement.cancel() メソッドの動作を決定する。詳細については、「 CANCEL_ALL 接続プロパティ 」(9 ページ) を参照。	バージョン設定によって異なる。詳細については、「 jConnect バージョンの設定 」(6 ページ) を参照。
CHARSET	TDS に渡された文字列用の文字セットを指定します。指定する CHARSET は、syscharsets にリストされている CHARSET と一致している必要がある。 null の場合、jConnect はサーバのデフォルトの CHARSET を使用する。	Null
CHARSET_CONVERTER_CLASS	このプロパティを使用して、jConnect で使用する文字セット・コンバータ・クラスを指定する。jConnect は SybDriver.setVersion() からのバージョン設定を使用して、デフォルト文字セット・コンバータを決定する。詳細については、「 文字セット・コンバータの選択 」(32 ページ) を参照。	バージョンに依存。
CONNECTION_FAILOVER	JNDI (Java Naming and Directory Interface) と共に使用する。詳細については、「 CONNECTION_FAILOVER 接続プロパティ 」(23 ページ) を参照。	true
DYNAMIC_PREPARE	動的 SQL の準備文がデータベースでプリコンパイルされているかどうかを判別する。詳細については、「 DYNAMIC_PREPARE 接続プロパティ 」(119 ページ) を参照。	false
EXPIRESTRING	ライセンスの有効期限を示す、読み込み専用のプロパティ。評価版の jConnect 以外は、有効期限は "never"。	Never
HOSTNAME	現在のホストの名前。	なし
HOSTPROC	ホスト・マシン上のアプリケーションのプロセスを識別する。	なし
IGNORE_DONE_IN_PROC	"true" に設定されている場合、更新の中間結果 (ストアド・プロシージャ内での) は返されず、最終結果セットだけが返される。	false
JCONNECT_VERSION	バージョン固有の特性の設定に使用する。詳細については、「 JCONNECT_VERSION 接続プロパティ 」(9 ページ) を参照。	5

プロパティ	説明	デフォルト値
LANGUAGE	サーバから返されるエラー・メッセージおよび jConnect メッセージに対して設定する。syslanguages 内の言語と対応する必要がある。	バージョンに依存。 詳細については、 「 jConnect バージョンの設定 」(6 ページ) を参照。
LANGUAGE_CURSOR	jConnect で、"protocol cursors." ではなく "language cursors" を使用する場合に設定する。詳細については、「 カーソルのパフォーマンス 」(122 ページ) を参照。	false
LITERAL_PARAMS	準備文をリテラルとして送信する必要がある Adaptive Server Anywhere とだけ使用する。それ以外の Sybase データベースには、このプロパティを "false" に設定できる。 "true" に設定すると、PreparedStatement インタフェースで setXXX メソッドによって設定されたパラメータは、SQL 文が実行されるときに SQL 文にリテラルに挿入される。 "false" に設定すると、パラメータ・マークは SQL 文内に残り、パラメータ値が別々にサーバに送信される。	false
PACKETSIZE	ネットワーク・パケット・サイズ	512
PASSWORD	ログイン・パスワード getConnection(String, String, String) メソッドを使用している場合は自動的に設定される。getConnection(String, Props) を使用している場合は明示的に設定する。	なし
PROTOCOL_CAPTURE	PROTOCOL_CAPTURE 接続プロパティは、アプリケーションと Adaptive Server 間の TDS 通信を取得するためのファイルの指定に使用する。	Null
PROXY	ゲートウェイ・アドレス。HTTP プロトコルの場合、URL は http://host:port。 暗号化をサポートする HTTPS プロトコルを使用する場合、URL は https://host:port/servlet_alias。	なし
REMOTEPWD	サーバ間のリモート・プロシージャ・コールを介したアクセスに対するリモート・サーバのパスワード。詳細については、「 サーバ間のリモート・プロシージャ・コールの実行 」(42 ページ) を参照。	なし

プロパティ	説明	デフォルト値
REPEAT_READ	<p>カラムを順序不同で読み込んだり、繰り返し読みこんだりできるよう、ドライバがカラムおよび出力パラメータのコピーを保持するかどうかを決定する。詳細については、「REPEAT_READ 接続プロパティ」(113 ページ)を参照。</p>	true
REQUEST_HA_SESSION	<p>このプロパティは、HA Failover が実行できるように設定されているバージョン 12 以降の Adaptive Server で、接続クライアントが HA Failover セッションを開始するかどうかを示す。</p> <p>このプロパティを "true" に設定すると、jConnect は HA Failover のログインを試みる。この接続プロパティを設定しないと、HA Failover を実行できるようにサーバが設定されている場合でも HA Failover セッションは開始しない。</p> <p>接続の確立後にプロパティをリセットすることはできない。</p> <p>HA Failover セッションをより柔軟に要求できるようにするには、実行時に REQUEST_HA_SESSION を設定するようにクライアント・アプリケーションをコーディングする。</p>	false
SELECT_OPENS_CURSOR	<p>"true" に設定されている場合、Statement.executeQuery() への呼び出しは、クエリに "FOR UPDATE" 句が含まれていると自動的にカーソルを生成する。</p> <p>以前に同じ文上で Statement.setFetchSize() または Statement.setCursorName() を呼び出したことがある場合、SELECT_OPENS_CURSOR に対して "true" を設定しても効果はない。</p> <p>注意 SELECT_OPENS_CURSOR を "true" に設定すると、パフォーマンスが若干低下することがある。</p> <p>jConnect でのカーソルの使用については「結果セットでのカーソルの使用方法」(45 ページ)を参照。</p>	false

プロパティ	説明	デフォルト値
SERIALIZE_REQUESTS	"true" に設定した場合、jConnect はサーバの応答を待ってから追加の要求を送信する。	false
SERVICENAME	DirectConnect ゲートウェイがサービスするバックエンド・データベース・サーバの名前。Adaptive Server Anywhere が接続するデータベースを示すのにも使用される。	None
SESSION_ID	このプロパティが設定されていると、jConnect は、TDS トンネリング・ゲートウェイによって開かれている既存の TDS セッション上でアプリケーションが通信を再開しようとしていると想定する。 jConnect はログイン・ネゴシエーションをスキップし、アプリケーションからの要求をすべて特定のセッション ID に転送する。	Null
SESSION_TIMEOUT	jConnect TDS トンネリング・サーブレットを使って作成した HTTP-tunnelled セッションが、アイドル時にクローズされるまでの時間を秒単位で指定する。指定時間後、接続は自動的にクローズされる。TDS トンネリング・サーブレットの詳細については、 143 ページ を参照。	Null
SQLINITSTRING	コマンドのセットを、バックエンド・データベース・サーバに渡されるように定義する。コマンドは、Statement.executeUpdate() メソッドを使用して実行できる SQL コマンドである必要がある。	Null
SYB SOCKET_FACTORY	jConnect がカスタム・ソケット実装を使用できるようにする。 SYB SOCKET_FACTORY を次のいずれかに設定する。 <ul style="list-style-type: none"> com.sybase.jdbcx.SybSocketFactory を実装するクラスの名前。 "DEFAULT"。これによって新しい java.net.Socket() をインスタンス化する。 詳細については、「 カスタム・ソケット・プラグインの実装 」(26 ページ)を参照。	Null
STREAM_CACHE_SIZE	文の応答ストリームのキャッシュに使用する最大サイズ。	Null (キャッシュ・サイズの制限なし)

プロパティ	説明	デフォルト値
USE_METADATA	<p>"true" に設定されている場合、DatabaseMetaData 接続を確立するとオブジェクトが作成、初期化される。DatabaseMetaData オブジェクトは指定のデータベースに接続する必要がある。</p> <p>jConnect は、Distributed Transaction Management Support (JTA/JTS : Java Transaction API / Java Transaction Services の略) や Dynamic Class Loading (DCL) などのいくつかの機能で DatabaseMetaData を使用する。</p> <p>アプリケーションがメタデータを必要とすることを示すエラー 010SJ が表示された場合は、jConnect のメタデータを返すストア・プロシージャをインストールする (『jConnect for JDBC インストール・ガイド』の第 3 章の「ストア・プロシージャのインストール」を参照)。</p>	true
USER	<p>ログイン ID</p> <p>getConnection(String, String, String) メソッドを使用している場合は自動的に設定する。getConnection(String, Props) を使用している場合は明示的に設定する。</p>	None
VERSIONSTRING	JDBC ドライバについての読み込み専用のバージョン情報。	jConnect ドライバのバージョン

次のコード例は接続プロパティの設定の方法を示します。jConnect で提供されるサンプル・プログラムにも、これらのプロパティの設定例が含まれています。

```

Properties props = new Properties();
props.put("user", "userid");
props.put("password", "user_password");
/*
 * If the program is an applet that wants to access
 * a server that is not on the same host as the
 * web server, then it uses a proxy gateway.
 */
props.put("proxy", "localhost:port");
/*
 * Make sure you set connection properties before
 * attempting to make a connection. You can also
 * set the properties in the URL.
 */
Connection con = DriverManager.getConnection

```

```
("jdbc:sybase:Tds:host:port", props);
```

Adaptive Server Enterprise への接続

Java アプリケーションで、jConnect ドライバを使用して Adaptive Server に接続する URL を定義します。URL の基本的なフォーマットは次のとおりです。

```
jdbc:sybase:Tds:host:port
```

ここで、

jdbc:sybase - ドライバを識別します。

Tds - Adaptive Server の Sybase 通信プロトコルです。

host:port - Adaptive Server のホスト名と受信ポートです。データベースや Open Server アプリケーションが使用するエントリについては、*\$SYBASE/interfaces* (UNIX) または *%SYBASE%\ini\sql.ini* (Windows) を参照してください。"query" エントリから *host:port* を取得します。

次のフォーマットを使用すると、特定のデータベースに接続できます。

```
jdbc:sybase:Tds:host:port/database
```

注意 Adaptive Server Anywhere 6.x または DirectConnect を使用している特定のデータベースに接続するには、[接続プロパティ](#) 接続プロパティを使用して、"/database" ではなくデータベース名を指定してください。

例

次のコードはポート 3697 で受信しているホスト "myserver" に、Adaptive Server への接続を作成します。

```
SysProps.put("user", "userid");  
SysProps.put("password", "user_password");  
String url = "jdbc:sybase:Tds:myserver:3697";  
Connection_con =  
    DriverManager.getConnection(url, SysProps);
```

URL 接続プロパティ・パラメータ

URL を定義するときに、jConnect ドライバ接続プロパティの値を指定できます。

注意 URL に設定されたドライバ接続プロパティは、**DriverManager.getConnection()** メソッドを使用してアプリケーションに設定された、対応するドライバ接続プロパティを上書きしません。

URL で接続プロパティを設定するには、プロパティ名とその値を URL 定義に追加します。次の構文を使用します。

```
jdbc:sybase:Tds:host:port/database?  
property_name=value
```

複数の接続プロパティを設定するには、それぞれの接続プロパティと値の前に "&" をつけて追加します。次に例を示します。

```
jdbc:sybase:Tds:myserver:1234/mydatabase?  
LITERAL_PARAMS=true&PACKETSIZE=512&HOSTNAME=myhost
```

いずれかの接続プロパティの値に "&" が含まれている場合は、接続プロパティ値の "&" の前にバックスラッシュ (\) を追加してください。たとえばホスト名が "a&bhost" の場合は、次の構文を使用します。

```
jdbc:sybase:Tds:myserver:1234/mydatabase?  
LITERAL_PARAMS=true&PACKETSIZE=512&HOSTNAME=  
a\&bhost
```

接続プロパティ値には、文字列であっても引用符を使用しないでください。たとえば、次を使用します。

```
HOSTNAME=myhost
```

次は使用しないでください。

```
HOSTNAME="myhost "
```

Adaptive Server Anywhere への接続

jConnect を Adaptive Server Anywhere と共に使用するには、Adaptive Server Anywhere バージョン 6.x にアップグレードする必要があります。

Adaptive Server Anywhere 5.x.x への接続

jConnect を介して Adaptive Server Anywhere バージョン 5.x.x に接続する必要がある場合は、Adaptive Server Anywhere と一緒に配布される Adaptive Server Anywhere Open Server Gateway dbos50 を実行してください。

注意 Powersoft Web サイトから入手できる Adaptive Server Anywhere の無料ダウンロード版には、この Open Server Gateway は含まれていません。Open Server Gateway および必要な Open Server DLL を含む CD を入手するには、Powersoft ((800) 265-4555) に連絡してください。送料および手数料がかかります。

- 1 Open Server Gateway 5.5.x3 以降、および Open Server DLL をインストールします。Open Server DLL バージョン 11.1 を使用してください。
- 2 たとえば **sqledit** などを使用して、`%SYBASE%\ini\sql.ini` ファイルにゲートウェイのエントリを追加します。
- 3 次のように入力して、ゲートウェイを開始します。

```
start dbos50 gateway-demo
```

gateway-demo は、手順 2 で定義したゲートウェイの名前です。

- 4 Open Server Gateway が稼働されると、次のように接続を定義できます。

```
jdbc:sybase:Tds:host:port
```

host は、Adaptive Server Anywhere および Open Server Gateway が稼働されているホストの名前です。*port* は *sql.ini* で定義されているポート番号です。

注意 複数の Adaptive Server Anywhere データベースをサポートするには、**sqledit** を使用して各データベースに対して異なるポートでエントリを追加します。その後、各データベースに対して Open Server Gateway を稼働します。

JNDI を使用してサーバに接続する方法

jConnect 4.0 以降では、JNDI (Java Naming and Directory Interface) を使用して、接続情報を提供できます。これには、次のようなものがあります。

- サーバに接続するためのホスト名およびポートを指定できる、集中化されたロケーション。アプリケーション内に特定のホストとポート番号をハードコードする必要はありません。
- すべてのアプリケーションで使用できるように接続プロパティとデフォルト・データベースを指定できる、集中化されたロケーション。
- 失敗した接続要求を処理するための jConnect CONNECTION_FAILOVER プロパティ。
CONNECTION_FAILOVER を "true" に設定すると、jConnect は接続に成功するまで JNDI ネーム・スペース内の一連のホストとポート・サーバ・アドレスに接続しようとします。

JNDI を使って jConnect を使用するには、JNDI がアクセスするどのディレクトリ・サービスでも一定の情報が使用でき、必要な情報が **javax.naming.Context** クラスに設定されることを確認する必要があります。この項では次の項目について説明します。

- [JNDI を使用するための接続 URL](#)
- [必要なディレクトリ・サービス情報](#)
- [CONNECTION_FAILOVER 接続プロパティ](#)
- [JNDI コンテキスト情報の提供](#)

JNDI を使用するための接続 URL

jConnect が接続情報の取得に JNDI を使用するよう指定するには、URL のサブプロトコルとして "sybase" の後ろに "jndi" を追加します。

```
jdbc:sybase:jndi:protocol-information-for-use-with-JNDI
```

URL 内で "jndi" に続くものはすべて JNDI を介して処理されます。たとえば、Lightweight Directory Access Protocol (LDAP) で JNDI を使用するには、次のように入力します。

```
jdbc:sybase:jndi:ldap://LDAP_hostname:port_number/servername=
Sybase11,o=MyCompany,c=US
```

この URL は LDAP サーバから情報を取得するよう JNDI に通知し、使用する LDAP サーバのホスト名とポート番号を渡して、LDAP 固有のフォームでデータベース・サーバの名前を提供します。

必要なディレクトリ・サービス情報

jConnect で JNDI を使用するときは、JNDI はターゲットとなるデータベース・サーバについて次のような情報を返す必要があります。

- 接続先のホスト名とポート番号
- 使用するデータベースの名前
- 個々のアプリケーションが独自に設定することができない接続プロパティ

この情報は、接続情報の提供に使用されるディレクトリ・サービス内に、固定のフォーマットに従って格納される必要があります。必要とされるフォーマットは、数字で表されるオブジェクト識別子 (OID) で構成されています。この識別子は、たとえば送信先データベースなど、提供される情報のタイプを識別し、その後ろにフォーマットされた情報が続きます。表 2-3 に必要なフォーマットを示します。

表 2-3: JNDI に必要なディレクトリ・サービス情報

情報のタイプ	オブジェクト識別子 (OID)	フォーマット	コメント
ホストおよびポート	1.3.6.1.4.1.897.4.2.5	TCP#1#hostname portnumber	複数のホストとポートを別々のエントリとして指定でき、 CONNECTION_FAILOVER を使用できる。
接続プロパティ	1.3.6.1.4.1.897.4.2.10	Prop1=value&Prop2=value&Prop3=value&...	各プロパティに対して別々のエントリを使用するか、または複数のプロパティをアンパサンド (&) で区切ってひとつのエントリに挿入して、複数の接続プロパティを指定できる。
データベース	1.3.6.1.4.1.897.4.2.11	databasename	接続先のデータベースの名前。このプロパティは、JDBC URL での "/database" のように機能する。
接続プロトコル	1.3.6.1.4.1.897.4.2.9	Tds	オプションだが、接続プロトコルを使用する場合は必ず "Tds" でなければならない。

次の例では、LDAP ディレクトリ・サービス下のデータベース・サーバ SYBASE11 に対して入力された接続情報を示します。

```

dn: servername=SYBASE11,o=MyCompany,c=US
servername: SYBASE11
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1266
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1337
1.3.6.1.4.1.897.4.2.5:TCP#1#standby1 4444
1.3.6.1.4.1.897.4.2.10:REPEAT_READ=false&PACKETSIZE=1024
1.3.6.1.4.1.897.4.2.10:CONNECTION_FAILOVER=true
1.3.6.1.4.1.897.4.2.11:pubs2
1.3.6.1.4.1.897.4.2.9:Tds

```

この例では、SYBASE11 はホスト "giotto" のポート 1266 か 1337 のいずれかを介してアクセスでき、ホスト "standby1" のポート 4444 を介してもアクセスできます。REPEAT_READ と PACKETSIZE の 2 つの接続プロパティは、1 つのエントリで設定されています。

CONNECTION_FAILOVER 接続プロパティは別のエントリで設定されています。SYBASE11 に接続するアプリケーションは、最初は *pubs2* データベースと接続されます。接続プロトコルを指定する必要はありませんが、指定する場合は、属性を "Tds" ("TDS" ではなく) として入力する必要があります。

CONNECTION_FAILOVER 接続プロパティ

CONNECTION_FAILOVER は、jConnect が JNDI を使用して接続情報を取得する場合に使用できるブール値の接続プロパティです。

CONNECTION_FAILOVER が "true" に設定されている場合、jConnect はサーバへの接続を複数回試みます。サーバに関連付けられたホストとポート番号への接続が 1 つ失敗すると、jConnect は JNDI を使用してそのサーバに関連付けられた次のホストとポート番号を取得し、それを介して接続を試みます。接続の試みは、サーバに関連付けられたすべてのホストとポートを順に介して続けられます。

たとえば、CONNECTION_FAILOVER が "true" に設定されていて、データベース・サーバは、前述の LDAP の例で示したように、次のホストとポート番号に関連付けられているとします。

```

1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1266
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1337
1.3.6.1.4.1.897.4.2.5:TCP#1#standby1 4444

```

サーバへの接続を取得するには、jConnect はポート 1266 でホスト "giotto" に接続しようとします。これに失敗すると、jConnect は "giotto" の 1337 で接続します。これに失敗した場合、jConnect はポート 4444 を介してホスト "standby1" に接続します。

CONNECTION_FAILOVER のデフォルトは "true" です。

CONNECTION_FAILOVER が "false" に設定されている場合、jConnect は最初のホストとポート番号への接続を試みます。失敗した場合、jConnect は SQL 例外を返し、再試行は行いません。

JNDI コンテキスト情報の提供

JNDI を使って jConnect を使用するには、開発者は Sun Microsystems からの JNDI 仕様に精通している必要があります。これは次の URL から入手できます。

<http://java.sun.com/products/jndi>

開発者は特に、JNDI と jConnect を一緒に使用するときに必要な初期化プロパティが **javax.naming.directory.DirContext** に設定されていることを確認する必要があります。これらのプロパティはシステム・レベルとランタイムのいずれかで設定できます。

2 つのキー・プロパティを次に示します。

- **Context.INITIAL_CONTEXT_FACTORY**

このプロパティは、使用する JNDI の最初のコンテキスト・ファクトリの完全に修飾されたクラス名をとります。

Context.PROVIDER_URL プロパティに指定された URL で使用する JNDI ドライバを決定します。

- **Context.PROVIDER_URL**

このプロパティは、LDAP ドライバなどのドライバがアクセスするディレクトリ・サービスの URL をとります。URL は、`"ldap://ldaphost:427"` といった文字列でなければなりません。

次の例では、ランタイムでのコンテキスト・プロパティの設定方法と、JNDI および LDAP を使用した接続の取得方法を示します。この例では、**INITIAL_CONTEXT_FACTORY** コンテキスト・プロパティは Sun Microsystems の LDAP サービス・プロバイダの実装を呼び出すように設定されます。**PROVIDER_URL** コンテキスト・プロパティは、ホスト `"ldap_server1"` のポート 983 にある LDAP ディレクトリ・サービスの URL に設定されます。

```
Properties props = new Properties();
```

```
/* We want to use LDAP, so INITIAL_CONTEXT_FACTORY is set to the
 * class name of an LDAP context factory. In this case, the
 * context factory is provided by Sun's implementation of a
 * driver for LDAP directory service.
 */
```

```

props.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");

/* Now, we set PROVIDER_URL to the URL of the LDAP server that
 * is to provide directory information for the connection.
 */
props.put(Context.PROVIDER_URL, "ldap://ldap_server1:983");

/* Set up additional context properties, as needed. */
props.put("user", "xyz");
props.put("password", "123");

/* get the connection */
Connection con = DriverManager.getConnection
    ("jdbc:sybase:jndi:ldap://ldap_server1:983" +
    "/servername=Sybase11,o=MyCompany,c=US",props);

```

getConnection() に渡される接続文字列に、LDAP 固有の情報が含まれることに注意してください。これは開発者が提供してください。

前述の例で示したように JNDI プロパティがランタイムで設定されると、jConnect は、サーバの初期化に使用されるよう、これを JNDI に渡します。次に例を示します。

```

javax.naming.directory.DirContext ctx =
    new javax.naming.directory.InitialDirContext(props);

```

次に jConnect は、次の例に示すように **DirContext.getAttributes()** を呼び出して JNDI から必要な接続情報を取得します。ctx は **DirContext** オブジェクトです。

```

javax.naming.directory.Attributes attrs =
    ctx.getAttributes(ldap://ldap_server1:983/servername=
    Sybase11, SYBASE_SERVER_ATTRIBUTES);

```

SYBASE_SERVER_ATTRIBUTES は、jConnect 内で定義された文字列の配列です。配列の値は、表 2-3 にリストされている必要なディレクトリ情報の OID です。

カスタム・ソケット・プラグインの実装

この項では、カスタム・ソケットの実装をアプリケーションにプラグインして、クライアントとサーバの間の通信をカスタマイズする方法について説明します。**javax.net.ssl.SSLSocket** は、暗号化を有効にするようカスタマイズできるソケットの例です。

com.sybase.jdbcx.SybSocketFactory は、**java.net.Socket** を返す **createSocket(String, int, Properties)** メソッドを含む Sybase 拡張インタフェースです。jConnect バージョン 4.1 以降のドライバでカスタム・ソケットをロードするには、アプリケーションは次のことを行う必要があります。

- このインタフェースの実装
- **createSocket(..)** メソッドの定義

jConnect は、以降の入出力オペレーションに新しいソケットを使用します。**SybSocketFactory** を実装するクラスはソケットを作成して、パブリックなソケットレベル機能の追加用に一般的な枠組みを提供します。

```
/**
 * Returns a socket connected to a ServerSocket on the named host,
 * at the given port.
 * @param host    the server host
 * @param port    the server port
 * @param props   Properties passed in through the connection
 * @returns Socket
 * @exception IOException, UnknownHostException
 */
public java.net.Socket createSocket(String host, int port, Properties props)
throws IOException, UnknownHostException;
```

プロパティ内で渡すことによって、**SybSocketFactory** のインスタンスは接続プロパティを使用してインテリジェント・ソケットを実装できます。

SybSocketFactory を実装してソケットを生成すると、ソケットを作成する異なる種類のファクトリ、または仮想ファクトリをアプリケーションに渡すことによって、同じアプリケーション・コードが異なるソケットを使用できます。ファクトリはソケットの構成に使用されているパラメータでカスタマイズできます。たとえば、異なるネットワーク・タイムアウトやすでに設定されているセキュリティ・パラメータでソケットを返すよう、ファクトリをカスタマイズできます。アプリケーションに返されたソケットは、圧縮、セキュリティ、レコード・マーキング、統計収集、ファイアウォール・トンネリング (**javax.net.SocketFactory**) などの機能に対して新しい API を直接公開する **java.net.Socket** のサブクラスとなることができます。

注意 **SybSocketFactory** は **javax.net.SocketFactory** をより簡略化することを意図しており、必要に応じてアプリケーションによる **java.net.*** から **javax.net.*** までのブリッジを可能にします。

jConnect でカスタム・ソケットを使用するには、次の手順に従います。

- 1 **com.sybase.jdbcx.SybSocketFactory** を実装する Java クラスを提供します。詳細については、「[カスタム・ソケットの作成と設定](#)」(28 ページ) を参照してください。
- 2 使用している実装を使って jConnect がソケットを取得できるよう、SYB SOCKET_FACTORY 接続プロパティを設定します。

SYB SOCKET_FACTORY 接続プロパティ

jConnect でカスタム・ソケットを使用するには、SYB SOCKET_FACTORY 接続プロパティを、次のいずれかの文字列に設定してください。

- **com.sybase.jdbcx.SybSocketFactory** を実装するクラスの名前
または
- "DEFAULT"。これによって新しい **java.net.Socket()** をインスタンス化する。

SYB SOCKET_FACTORY の設定方法については、「[接続プロパティの設定](#)」(12 ページ) を参照してください。

カスタム・ソケットの作成と設定

カスタム・ソケットを取得すると、jConnect はソケットを使用してサーバに接続します。ソケットの設定は、jConnect が取得する前に完了している必要があります。

この項では、**javax.net.ssl.SSLSocket** などの SSL ソケットの実装を jConnect でプラグインする方法について説明します。

注意 現在のところ、SSL をサポートしている Sybase サーバはありません。

次の例は、SSL の実装がどのように **SSLSocket** のインスタンスを作成、設定し、それを返すかを示します。例では、**MySSLSocketFactory** クラスが **SybSocketFactory** を実装し、**javax.net.ssl.SSLSocketFactory** を拡張して SSL を実装します。ここでは 2 つの **createSocket** メソッドが含まれています。1 つは **SSLSocketFactory** に、もう 1 つは **SybSocketFactory** に対するもので、次のことを実現します。

- SSL ソケットを作成します。
- **SSLSocket.setEnabledCipherSuites()** を呼び出して暗号化に使用する cipher suites を指定します。
- jConnect が使用するソケットを返します。

例

```
public class MySSLSocketFactory extends SSLSocketFactory
    implements SybSocketFactory
{
    /**
     * Create a socket, set the cipher suites it can use, return
     * the socket.
     * Demonstrates how cipher suites could be hard-coded into the
     * implementation.
     *
     * See javax.net.SSLSocketFactory#createSocket
     */
    public Socket createSocket(String host, int port)
        throws IOException, UnknownHostException
    {
        // Prepare an array containing the cipher suites that are to
        // be enabled.
        String enableThese[] =
```

```

    {
        "SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA",
        "SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5",
        "SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA"
    }
    ;
    Socket s =
        SSLSocketFactory.getDefault().createSocket(host, port);
    ((SSLSocket)s).setEnabledCipherSuites(enableThese);
    return s;
}
/**
 * Return an SSLSocket.
 * Demonstrates how to set cipher suites based on connection
 * properties like:
 * Properties _props = new Properties();
 * Set other url, password, etc. properties.
 * _props.put("CIPHER_SUITES_1",
 *     "SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA");
 * _props.put("CIPHER_SUITES_2",
 *     "SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5");
 * _props.put("CIPHER_SUITES_3",
 *     "SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA");
 * _conn = _driver.getConnection(url, _props);
 *
 * See com.sybase.jdbcx.SybSocketFactory#createSocket
 */
public Socket createSocket(String host, int port,
    Properties props)
    throws IOException, UnknownHostException
{
    // check to see if cipher suites are set in the connection
    // properites
    Vector cipherSuites = new Vector();
    String cipherSuiteVal = null;
    int cipherIndex = 1;
    do
    {
        if((cipherSuiteVal = props.getProperty("CIPHER_SUITES_"
            + cipherIndex++)) == null)
        {
            if(cipherIndex <= 2)
            {
                // No cipher suites available
                // return what the object considers its default
                // SSLSocket, with cipher suites enabled.

```

```

        return createSocket(host, port);
    }
    else
    {
        // we have at least one cipher suite to enable
        // per request on the connection
        break;
    }
    else
    {
        // add to the cipher suit Vector, so that
        // we may enable them together
        cipherSuites.addElement(cipherSuiteVal);
    }
}
while(true);
// lets you create a String[] out of the created vector
String enableThese[] = new String[cipherSuites.size()];
cipherSuites.copyInto(enableThese);
// enable the cipher suites
Socket s =
    SSLSocketFactory.getDefault().createSocket
        (host, port);
((SSLSocket)s).setEnabledCipherSuites(enableThese);
// return the SSLSocket
return s;
}
// other methods
}

```

jConnect はソケットの種類に関する情報を必要としないので、ソケットを返す前に設定を完了しておいてください。

詳細については、次を参照してください。

- *Encrypt.java* - jConnect ディレクトリの *sample* (jConnect 4) および *sample2* (jConnect 5) サブディレクトリにあります。このサンプルは jConnect アプリケーションでの **SybSocketFactory** インタフェースの使い方を示します。
- *MySSLSocketFactory.java* - これも jConnect ディレクトリの *sample* (jConnect 4) および *sample2* (jConnect 5) サブディレクトリにあります。これは **SybSocketFactory** インタフェースの実装のサンプルで、アプリケーションにプラグインして使用できます。

国際化とローカライゼーションの処理

この項では jConnect に関する国際化とローカライゼーションについて説明します。

jConnect 文字セット・コンバータ

jConnect は、すべての文字セット変換に対して特別なクラスを使用します。文字セット・コンバータ・クラスを選択することによって、シングルスバイトおよびマルチバイトの文字セット変換と、文字セット変換がアプリケーションに与えるパフォーマンスの影響を jConnect がどのように処理するかを指定します。

文字セット変換クラスは 2 つあります。jConnect が使用する変換クラスは、バージョン設定 (たとえば、`VERSION_4`)、`CHARSET` および `CHARSET_CONVERTER_CLASS` 接続プロパティに基づきます。

- **TruncationConverter** クラスは、`iso_1` や `cp850` などの ASCII 文字を使用するシングルスバイト文字セットでのみ動作します。マルチバイト文字セットや非 ASCII 文字を使用するシングルスバイト文字セットでは動作しません。

TruncationConverter クラスを使用して、jConnect 5.x は文字セットを jConnect バージョン 2.2 と同じように処理します。

TruncationConverter クラスは、バージョン設定が `VERSION_2` の場合のデフォルト・コンバータです。

- **PureConverter** クラスは、pure Java のマルチバイト文字セット・コンバータです。バージョン設定が `VERSION_4` 以上の場合、jConnect はこのコンバータ・クラスを使用します。

TruncationConverter クラスと互換性のない `CHARSET` 接続プロパティに指定された文字セットを検出した場合は、jConnect は `VERSION_2` でもこのコンバータを使用します。

これは、マルチバイト文字セット変換を有効にしますが、

PureConverter クラスは jConnect ドライバのパフォーマンスに悪影響を与えることがあります。ドライバのパフォーマンスが問題になる場合は、「[文字セット変換パフォーマンスの改善](#)」(33 ページ)を参照してください。

文字セット・コンバータの選択

jConnect は **SybDriver.setVersion()** からのバージョン設定を使用して、使用するデフォルト文字セット・コンバータ・クラスを決定します。VERSION_2 の場合、デフォルトは **TruncationConverter** です。VERSION_4 以上の場合、デフォルトは **PureConverter** です。

CHARSET_CONVERTER_CLASS 接続プロパティを設定して、jConnect でどの文字セット・コンバータを使用するかを指定できます。これは jConnect バージョンのデフォルト以外の文字セット・コンバータを使用する場合に便利です。

たとえば jConnect を VERSION_4 以上に設定したが、マルチバイトの **PureConverter** クラスではなく **TruncationConverter** クラスを使用する場合は、CHARSET_CONVERTER_CLASS を次のように設定します。

jConnect 4.1 の場合：

```
...
props.put("CHARSET_CONVERTER_CLASS",
    "com.sybase.utils.TruncationConverter")
```

jConnect 5.x の場合：

```
...
props.put("CHARSET_CONVERTER_CLASS",
    "com.sybase.jdbc2.utils.TruncationConverter")
```

CHARSET 接続プロパティの設定

CHARSET ドライバ・プロパティを設定することによって、アプリケーションで使用する文字セットを指定できます。CHARSET プロパティを設定していない場合は、次のようになります。

- VERSION_2 の場合、jConnect は iso_1 をデフォルト文字セットとして使用します。
- VERSION_3、VERSION_4、および VERSION_5 の場合、jConnect はデータベースのデフォルト文字セットを使用し、クライアント側に必要な変換を行うよう自動的に調整します。

IsqlApp アプリケーションに対して **-J charset** コマンドライン・オプションを使用して文字セットを指定することもできます。

使用している Adaptive Server にどの文字セットがインストールされているかを調べるには、サーバで次の SQL クエリを発行してください。

```
select name from syscharsets
go
```

PureConverter クラスの場合、指定の CHARSET がクライアントの Java 仮想マシン (VM) で使用できないと、接続は **SQLException** で失敗し、Adaptive Server とクライアントの両方にサポートされている文字セットに CHARSET を設定するよう示されます。

TruncationConverter クラスを使用している場合は、指定された CHARSET が 7-bit ASCII であるかどうかに関係なく、文字トランケーションが適用されます。

文字セット変換パフォーマンスの改善

マルチバイト文字セットを使用していてドライバ・パフォーマンスを改善する必要がある場合は、jConnect サンプルで提供される

SunloConverter クラスを使用できます。詳細については、「[文字セット変換](#)」(114 ページ) を参照してください。

サポートされている文字セット

表 2-4 に、このリリースの jConnect でサポートされている Sybase 文字セットをリストします。また、サポートされている文字セットのそれぞれについて、対応する JDK バイト・コンバータもリストします。

jConnect は Unicode の 1 つである UCS-2 をサポートしていますが、現在のところ Sybase データベースまたは Open Server は Unicode の UTF8 をサポートします。

Sybase *sjis* 文字セットは IBM 拡張、Microsoft 拡張を含みます。JDK SJIS バイト・コンバータには含まれないため、一部のコードが "?" (0x3f) として扱われる場合があります。この現象は Windows Platform 固有のもので後述の **SunloConverter** の入出力を直接 "MS932" と指定することで回避できます。なお、JDK 1.1.8 以降で動作します。

表 2-4 に、現在 Sybase がサポートしている文字セットをリストします。

表 2-4: サポートされている Sybase 文字セット

SybCharset 名	JDK バイト・コンバータ
ascii_7	8859_1
big5	Big5
cp037	Cp037
cp437	Cp437
cp500	Cp500
cp850	Cp850
cp852	Cp852
cp855	Cp855
cp857	Cp857
cp860	Cp860
cp863	Cp863
cp864	Cp864
cp866	Cp866
cp869	Cp869
cp874	Cp874
cp932	Cp932
cp936	Cp936
cp950	Cp950
cp1250	Cp1250
cp1251	Cp1251
cp1252	Cp1252
cp1253	Cp1253
cp1254	Cp1254
cp1255	Cp1255
cp1256	Cp1256
cp1257	Cp1257
cp1258	Cp1258
deckanji	EUCJIS
eucgb	GB2312
eucjis	EUCJIS
eucksc	Cp949
ibm420	Cp420
ibm918	Cp918
iso_1	8859_1
iso88592	8859-2
is088595	8859_5

SybCharset 名	JDK バイト・コンバータ
iso88596	8859_6
iso88597	8859_7
iso88598	8859_8
iso88599	8859_9
iso885915	8859_15
koi8	KOI8_R
mac	Macroman
mac_cyr	MacCyrillic
mac_ee	MacCentralEurope
macgreek	MacGreek
macturk	MacTurkish
sjis (注記を参照)	SJIS
tis620	MS874
utf8	UTF8

ヨーロッパ通貨記号のサポート

jConnect バージョン 4.1 以降では、ヨーロッパの新通貨の記号「ユーロ」の使用、および UCS-2 Unicode との間の変換をサポートしています。

ユーロは、次の Sybase 文字セットに追加されています。cp1250、cp1251、cp1252、cp1253、cp1254、cp1255、cp1256、cp1257、cp1258、cp874、iso885915、および utf8。

文字セット cp1257、cp1258、iso885915 は新規です。

ユーロ記号を使用するには、次のようにしてください。

- **PureConverter** クラス、pure Java、マルチバイト文字セット・コンバータを使用してください。詳細については、[「jConnect 文字セット・コンバータ」\(31 ページ\)](#) を参照してください。
- 新しい文字セットがサーバにインストールされていることを確認してください。

現在のところ、ユーロ記号は Adaptive Server Enterprise バージョン 11.9.2 以降でしかサポートされていません。Adaptive Server Anywhere はユーロ記号をサポートしません。

- クライアントに適切な文字セットを選択してください。詳細については、[「CHARSET 接続プロパティの設定」\(32 ページ\)](#) を参照してください。

- JDK 1.1.7 または Java 2 Platform へアップグレードしてください。
Windows 環境のみでご利用の方は、前述 (33 ページ) の文字変換にご注意の上、JDK 1.1.8 以降のリリースをご使用ください。

サポートされていない文字セット

以下の JDK バイト・コンバータは Sybase 文字セットに対応しないため、次の Sybase 文字セットは jConnect 5.x ではサポートされていません。

- cp1047
- euccns
- greek8
- roman8
- turkish8

アプリケーションがこれらの文字の 7-bit ASCII サブセットだけを使用している場合、**TruncationConverter** クラスでこれらの文字セットを使用できます。

データベースの作業

この項では、jConnect に関する次の項目について説明します。

- ・ 高可用性フェールオーバー・サポートの実装
- ・ サーバ間のリモート・プロシージャ・コールの実行
- ・ データベース・メタデータへのアクセス
- ・ 結果セットでのカーソルの使用方法
- ・ バッチ更新のサポート
- ・ ストアド・プロシージャの結果セットによるデータベースの更新
- ・ データ型の作業

高可用性フェールオーバー・サポートの実装

jConnect バージョン 4.2 および 5.2 は、Adaptive Server Enterprise バージョン 12.0 で使用できる Sybase Failover 機能をサポートしています。

注意 高可用性システムでの Sybase Failover は、「接続フェールオーバー」とは異なる機能です。両方とも使用する場合は、必ずこの項をお読みください。

概要

Sybase Failover を使用すると、2 つの Adaptive Servers バージョン 12.0 をコンパニオンとして設定できます。プライマリ・コンパニオンで障害が発生した場合、そのサーバのデバイス、データベース、および接続は、セカンダリ・コンパニオンに移行します。

高可用性システムは、非対称型と対称型のいずれかに設定することができます。

非対称型 の設定では、2つの **Adaptive Server** が異なるマシンに物理的に配置されていて、一方のサーバが停止した場合、もう一方のサーバが停止したサーバの負荷を引き受けるように接続されています。セカンダリ **Adaptive Server** は、「ホット・スタンバイ」として機能し、フェールオーバーが発生するまで処理を実行しません。対称型 の設定も、異なるマシンで動作している2つの **Adaptive Server** で構成されます。ただし、この設定では、フェールオーバーが発生すると、いずれかの **Adaptive Server** がもう一方の **Adaptive Server** のプライマリあるいはセカンダリ・コンパニオンとして動作します。この設定の場合、各 **Adaptive Server** は、各自のシステム・デバイス、システム・データベース、ユーザ・データベースおよびユーザ・ログインを使って完全に動作します。

どちらの設定でも、2つのマシンはデュアル・アクセス用に設定されているため、ディスクを参照したり両方のマシンにアクセスすることができます。

jConnect で Sybase Failover を有効にすることで、クライアント・アプリケーションをフェールオーバー用に設定されている **Adaptive Server** に接続できます。プライマリ・サーバがセカンダリ・サーバにフェールオーバーした場合、クライアント・アプリケーションもセカンダリ・サーバに自動的に切り替わり、ネットワーク接続を再確立します。

注意 Sybase Failover の詳細については、『高可用性システムにおける Sybase フェールオーバーの使用』を参照してください。

稼働条件、依存性、制限

- 2つの **Adaptive Servers** バージョン 12.0 をフェールオーバー用に設定する必要があります。
- jConnect 4.2 または jConnect 5.2 を使用する必要があります。以前のバージョンのドライバは、この機能をサポートしていません。
- クライアントがフェールオーバーした場合、フェールオーバーが発生する前にデータベースにコミットされた変更だけが保持されます。
- クライアント・アプリケーション接続は、JNDI を使用して確立する必要があります。詳細については、「[JNDI を使用してサーバに接続する方法](#)」(21 ページ) を参照してください。

- フェールオーバーが発生した場合、jConnect のイベント通知機能は動作しません。詳細については、「[イベント通知の使用方法](#)」(65 ページ) を参照してください。
- 使用していない Statement インスタンスはすべて閉じてください。jConnect は、フェールオーバーできるように Statement インスタンスに関する情報を格納します。Statement インスタンスをクローズしないと、メモリ・リークが発生します。

jConnect でのフェールオーバーの実装

jConnect でフェールオーバー・サポートを実装するには、次の手順に従います。

- 1 フェールオーバー用のプライマリ Adaptive Servers とセカンダリ Adaptive Servers を設定します。
- 2 JNDI が必要とするディレクトリ・サービス情報ファイルに、プライマリ・サーバ用のエントリと、セカンダリ・サーバ用のエントリを個別に追加します。プライマリ・サーバのエントリには、セカンダリ・サーバのエントリを参照する属性 (HA OID) を入れます。

LDAP を JNDI のサービス・プロバイダとして使用している場合は、この HA 属性に次の 3 つのフォームを使用できます。

- *Relative Distinguished Name (RDN)* - このフォームは、検索ベース (通常、`java.naming.provider.url` 属性によって提供される) とこの属性値の組み合わせにより、セカンダリ・サーバを識別できると想定しています。たとえば、プライマリ・サーバがホスト名 :4200 にあり、セカンダリ・サーバがホスト名 :4202 にある場合は、次のようになります。

```
dn: servername=happrimary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: servername=hasecondary
objectclass: sybaseServer
```

```
dn: servername=hasecondary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

- *Distinguished Name (DN)* - このフォームは、HA 属性の値がセカンダリ・サーバをユニークに識別すると想定していて、検索ベース内に重複した値が見つかる場合と見つからない場合があります。次に例を示します。

```
dn: servername=happrimary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: servername=hasecondary,
    o=Sybase, c=US ou=Accounting
objectclass: sybaseServer
```

```
dn: servername=hasecondary, o=Sybase, c=US, ou=Accounting
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

hasecondary がツリーの異なるブランチにあることに注目してください (追加の **ou=Accounting** 修飾子を参照してください)。

- *Full LDAP URL* - このフォームは、検索ベースに関して何も想定していません。HA 属性は、セカンダリ・サーバ (異なる LDAP サーバを指すことも可能) を識別するための完全に修飾された LDAP URL と考えられます。次に例を示します。

```
dn: servername=hafailover, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: ldap://ldapserver:386/servername=secondary,
    o=Sybase, c=US ou=Accounting
objectclass: sybaseServer
```

```
dn: servername=secondary, o=Sybase, c=US, ou=Accounting
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

- 3 JNDI が要求するディレクトリ・サービス情報ファイル内に **REQUEST_HA_SESSION** 接続プロパティを設定して、接続を確立するたびにフェールオーバー・セッションを有効にします。

新しい **REQUEST_HA_SESSION** 接続プロパティは、接続クライアントがフェールオーバー用に設定された **Adaptive Server** バージョン 12.0 とフェールオーバー・セッションを開始することを示すのに使用します。プロパティを "true" に設定すると、**jConnect** はフェールオーバー・ログインを試行します。この接続プロパティを設定しないと、サーバが正しく設定されていてもフェールオーバー・セッションは開始しません。**REQUEST_HA_SESSION** のデフォルト値は、"false" です。

接続プロパティを他の接続プロパティと同様に設定します。接続の確立後にプロパティをリセットすることはできません。

フェールオーバー・セッションをより柔軟に要求できるようにするには、実行時に **REQUEST_HA_SESSION** を設定するようにクライアント・アプリケーションをコーディングしてください。

次の例では、LDAP ディレクトリ・サービスにあるデータベース・サーバ SYBASE11 用に入力された接続情報を示します。

```
dn: servername=SYBASE11,o=MyCompany,c=US
1.3.6.1.4.1.897.4.2.5:TCP#1#tahiti 3456
1.3.6.1.4.1.897.4.2.10:REPEAT_READ=false&PACKETSIZE=1024
1.3.6.1.4.1.897.4.2.10:CONNECTION_FAILOVER=false
1.3.6.1.4.1.897.4.2.11:pubs2
1.3.6.1.4.1.897.4.2.9:Tds
1.3.6.1.4.1.897.4.2.15:servername=SECONDARY
1.3.6.1.4.1.897.4.2.10:REQUEST_HA_SESSION=true

dn:servername=SECONDARY, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1#moorea 6000
```

ここで、"tahiti" はプライマリ・サーバで、"moorea" はセカンダリ・コンパニオン・サーバです。

4 JNDI および LDAP を使用して接続を要求します。

- 次の場合、jConnect は、LDAP サーバのディレクトリ・サーバを使用して、プライマリ・サーバとセカンダリ・サーバの名前とロケーションを判断します。

```
/* get the connection */
Connection con = DriverManager.getConnection
("jdbc:sybase:jndi:ldap://ldap_server1:983" +
"/servername=Sybasell,o=MyCompany,c=US",props);
```

または

- 次に示すように、検索ベースを指定します。

```
props.put(Context.PROVIDER_URL,
"ldap://ldap_server1:983/ o=MyCompany, c=US");

Connection con=DriverManager.getConnection
("jdbc:sybase:jndi:servername=Sybasell", props);
```

プライマリ・サーバへのログイン

Adaptive Server がフェールオーバー用に設定されていない場合、または何らかの理由でフェールオーバー・セッションを付与できない場合、クライアントはログインすることができず、次の警告が表示されます。

```
'The server denied your request to use the high-
availability feature.
```

```
Please reconfigure your database, or do not request a
high-availability session.'
```

セカンダリ・サーバへのフェールオーバー

フェールオーバーが発生すると、例外が発生し、クライアントは JNDI を使用してセカンダリ・データベースに自動的に再接続します。

次のことに注意してください。

- クライアントが接続されていたデータベースの ID とすべてのコミット済みのトランザクションが保持されます。
- 部分的に読み込まれた結果セット、カーソル、およびストアド・プロシージャ・コールは失われます。
- フェールオーバーが発生した場合、アプリケーションは、プロシージャを再起動するか、最後に完了したトランザクションまたはアクティビティに戻る必要があります。

プライマリ・サーバへのフェールバック

ある時点で、クライアントはセカンダリ・サーバからプライマリ・サーバにフェールバックします。いつフェールバックするかは、セカンダリ・サーバに **sp_failback** を発行するシステム管理者によって決定されます。その後、クライアントは、「[セカンダリ・サーバへのフェールオーバー](#)」(42 ページ) で説明しているのと同じ動作と結果をプライマリ・サーバ上で得ることができます。

サーバ間のリモート・プロシージャ・コールの実行

あるサーバで稼働している Transact-SQL 言語のコマンドやストアド・プロシージャは、別のサーバにあるストアド・プロシージャを実行できます。アプリケーションが接続されているサーバは、リモート・サーバにログインしてサーバ間のリモート・プロシージャ・コールを実行します。

アプリケーションはサーバ間の通信に "universal" パスワードを指定できます。このパスワードはすべてのサーバ間の通信に使用されます。接続が開かれると、サーバはどのリモート・サーバにログインするにも、このパスワードを使用します。

デフォルトでは、jConnect は現在の接続のパスワードをサーバ間通信のデフォルト・パスワードとして使用します。

ただし、2 つのサーバで同一ユーザに対するパスワードが異なり、そのユーザがサーバ間のリモート・プロシージャ・コールを実行している場合、アプリケーションはそれぞれのサーバで使用するパスワードを明示的に定義する必要があります。

jConnect バージョン 4.1 以降では、ユニバーサルな "remote" パスワード、またはいくつかのサーバでの異なるパスワードを指定できるプロパティが含まれています。jConnect では、**SybDriver** クラスの **setRemotePassword()** メソッドを使用してプロパティを設定できます。

```
Properties connectionProps = new Properties();
```

```
public final void setRemotePassword(String serverName,
    String password, Properties connectionProps)
```

このメソッドを使用するには、アプリケーションで **SybDriver** クラスをインポートしてからメソッドを呼び出してください。

jConnect 4.x の場合：

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
    Class.forName("com.sybase.jdbc.SybDriver").newInstance();
sybDriver.setRemotePassword
    (serverName, password, connectionProps);
```

jConnect 5.x の場合：

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
    Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
sybDriver.setRemotePassword
    (serverName, password, connectionProps);
```

注意 複数のサーバに異なるリモート・パスワードを設定するには、使用している jConnect のバージョンに従って、各サーバに対して前述の呼び出しを繰り返します。

この呼び出しによって、指定されたサーバ名とパスワードの組が、指定された **Properties** オブジェクトに追加されます。これはアプリケーションによって **DriverManager.getConnection(server_url, props)** の **DriverManager** に渡されます。

serverName が NULL の場合、**setRemotePassword()** への直前の呼び出しによって具体的に定義されたサーバを除くすべてのサーバへの以降の接続に対して、ユニバーサル・パスワードが **password** に設定されます。

アプリケーションが **REMOTEPWD** プロパティを設定すると、jConnect はデフォルト・ユニバーサル・パスワードを設定しなくなります。

データベース・メタデータへのアクセス

JDBC の **DatabaseMetaData** メソッドをサポートするために、Sybase では jConnect がデータベースについてのメタデータと呼び出せる一連のストアド・プロシージャを提供しています。これらのストアド・プロシージャは、JDBC メタデータ・メソッドが動作するようにサーバにインストールされる必要があります。

メタデータを提供するためのストアド・プロシージャが Sybase サーバにまだインストールされていない場合は、jConnect で提供されるストアド・プロシージャ・スクリプトを使用してそれらをインストールすることができます。

- *sql_server.sql* は、12.0 より前のバージョンの Adaptive Server データベースにストアド・プロシージャをインストールします。
- *sql_server12.sql* は、Adaptive Server バージョン 12.0 データベースにストアド・プロシージャをインストールします。
- *sql_anywhere.sql* は、Adaptive Server Anywhere データベースにストアド・プロシージャをインストールします。

注意 これらのスクリプトの最新バージョンは jConnect のすべてのバージョンと互換性があります。

ストアド・プロシージャをインストールする方法については、『jConnect for JDBC インストール・ガイド』を参照してください。

さらに、メタデータ・メソッドを使用するには、接続を確立するときに `USE_METADATA` 接続プロパティを "true" (デフォルト値) に設定する必要があります。

データベース内のテンポラリ・テーブルに関するメタデータを取得することはできません。

注意 `DatabaseMetaData.getPrimaryKeys()` メソッドは、テーブル定義 (CREATE TABLE) または代替テーブル (ALTER TABLE ADD CONSTRAINT) で宣言されたプライマリ・キーを探します。`sp_primarykey` を使用して定義されたキーは探しません。

サーバ側メタデータのインストール

メタデータ・サポートは、クライアント (ODBC、JDBC) かデータ・ソース (サーバ・ストアド・プロシージャ) のいずれかに実装できます。jConnect ではメタデータ・サポートをサーバに提供しているので、次のような利点があります。

- jConnect を小さなサイズで管理し、インターネットからドライバをダウンロードするときの時間を短縮する。
- データ・ソースに事前にロードされたストアド・プロシージャからランタイムの効率を取得する。
- 柔軟性が高まる - jConnect は、さまざまなデータベースに接続可能。

結果セットでのカーソルの使用方法

jConnect 5.x は多くの JDBC 2.0 カーソルと更新メソッドを実装しています。これらのメソッドは、カーソルの使用と結果セット内の値に基づくテーブル内のローの更新を簡単にします。

注意 JDBC 2.0 の完全なサポートを得るには、jConnect バージョン 5.x 以降を使用してください。jConnect バージョン 4.x では、Sybase 拡張機能、および jConnect ディレクトリ下の *sample* サブディレクトリにある **ScrollableResultSet.java** サンプルを介して、JDBC 2.0 の機能のいくつかを提供しています。**com.sybase.jdbcx** およびこれらのメソッドの javadocs の **sample** パッケージを参照してください。

JDBC 2.0 では、**ResultSets** はそのタイプと同時性が特長になっています。タイプと同時性の値は **java.sql.ResultSet** インタフェースの一部で、javadocs に記述されています。

表 2-5 に、jConnect 5.x で使用できる **java.sql.ResultSet** の特長を示します。

表 2-5: jConnect 5.x で使用できる **java.sql.ResultSet** オプション

	タイプ		
	TYPE_FORWARD_ONLY	TYPE_SCROLL_INSENSITIVE	TYPE_SCROLL_SENSITIVE
同時性			
<i>CONCUR_READ_ONLY</i>	5.x でサポート。	5.x でサポート。	5.x では使用できない。
<i>CONCUR_UPDATABLE</i>	5.x でサポート。	5.x では使用できない。	5.x では使用できない。

この項では、次の項目について説明します。

- [カーソルの作成](#)
- [JDBC 1.x メソッドを使用した位置付け更新と削除](#)
- [準備文でのカーソルの使用](#)
- [jConnect の SCROLL_INSENSITIVE 結果セットのサポート](#)

カーソルの作成

jConnect 4.x を使用してカーソルを作成するには、**SybStatement.setCursorName()** または **SybStatement.setFetchSize()** を使用してください。**SybStatement.setCursorName()** を使用するとき、カーソルに明示的に名前を割り当てます。

SybStatement.setCursorName() のシグニチャを次に示します。

```
void setCursorName(String name) throws SQLException;
```

SybStatement.setFetchSize() を使用してカーソルを作成し、それぞれのフェッチでデータベースから返されるローの数を指定します。

SybStatement.setFetchSize() のシグニチャを次に示します。

```
void setFetchSize(int rows) throws SQLException;
```

setFetchSize() を使用してカーソルを作成すると、jConnect ドライバがカーソルに名前をつけます。カーソル名を取得するには、**ResultSet.setCursorName()** を使用してください。

jConnect バージョン 5.x にはバージョン 4.x の場合と同様にしてカーソルを作成できますが、バージョン 5.x は JDBC 2.0 をサポートしているので、別の方法でも作成できます。次の JDBC 2.0 メソッドを接続に使用して、文に返される **ResultSet** の種類を指定できます。

```
Statement createStatement(int resultSetType, int  
resultSetConcurrency) throws SQL Exception
```

タイプと同時性は、[表 2-5](#) にリストされている **ResultSet** インタフェースのタイプと同時性に対応します。サポートされていない **ResultSet** を要求すると、SQL 警告が接続に関連付けられます。返された **Statement** が実行されると、要求したものに最も近い **ResultSet** の種類を受け取ります。メソッドの動作の詳細については、JDBC 2.0 の仕様を参照してください。

createStatement() を使用しない場合、または jConnect バージョン 4.x を使用している場合、**ResultSet** のデフォルト・タイプは次のようになります。

- **Statement.executeQuery()** だけ呼び出すと、返される **ResultSet** は **SybResultSet** で、**TYPE_FORWARD_ONLY** と **CONCUR_READ_ONLY** になります。
- **setFetchSize()** または **setCursorName()** を呼び出すと、**executeQuery()** から返される **ResultSet** は **SybCursorResultSet** で、**TYPE_FORWARD_ONLY** と **CONCUR_UPDATABLE** になります。

ResultSet オブジェクトの種類が意図したものであることを確認するために、**ResultSet** の JDBC 2.0 API では2つのメソッドが追加されました。

```
int getConcurrency() throws SQLException;
int getType() throws SQLException;
```

カーソルの作成と使用の基本的な手順を次に示します。

- 1 **Statement.setCursorName()** または **SybStatement.setFetchSize()** を使用してカーソルを作成します。
- 2 **Statement.executeQuery()** を呼び出して文に対してカーソルを開き、カーソル結果セットを返します。
- 3 **ResultSet.next()** を呼び出してローをフェッチし、結果セットにカーソルを位置付けます。

次の例ではカーソルを作成して結果セットを返すのに、2つのメソッドを1つずつ使用します。またここでは、**SybStatement.setFetchSize()** によって作成されたカーソルの名前を取得するのに、**ResultSet.setCursorName()** も使用します。

```
// With conn as a Connection object, create a
// Statement object and assign it a cursor using
// Statement.setCursorName().
Statement stmt = conn.createStatement();
stmt.setCursorName("author_cursor");

// Use the statement to execute a query and return
// a cursor result set.
ResultSet rs = stmt.executeQuery("SELECT au_id,
    au_lname, au_fname FROM authors
    WHERE city = 'Oakland'");
while(rs.next())
{
    ...
}

// Create a second statement object and use
// SybStatement.setFetchSize() to create a cursor
```

```
// that returns 10 rows at a time.
SybStatement syb_stmt = conn.createStatement();
syb_stmt.setFetchSize(10);

// Use the syb_stmt to execute a query and return
// a cursor result set.
SybCursorResultSet rs2 =
    (SybCursorResultSet)syb_stmt.executeQuery
        ("SELECT au_id, au_lname, au_fname FROM authors
         WHERE city = 'Pinole'");
while(rs2.next())
{
    ...
}

// Get the name of the cursor created through the
// setFetchSize() method.
String cursor_name = rs2.getCursorName();
...
// For jConnect 5.x, create a third statement
// object using the new method on Connection,
// and obtain a SCROLL_INSENSITIVE ResultSet.
// Note: you no longer have to downcast the
// Statement or the ResultSet.
Statement stmt = conn.createStatement(

    ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
ResultSet rs3 = stmt.executeQuery
    ("SELECT ... [whatever]");
// Execute any of the JDBC 2.0 methods that
// are valid for read only ResultSets.
rs3.next();
rs3.previous();
rs3.relative(3);
rs3.afterLast();

...
```

JDBC 1.x メソッドを使用した位置付け更新と削除

次の例は、JDBC 1.x でメソッドを使って位置付け更新を実行する方法について説明します。この例では2つの **Statement** オブジェクトを作成します。1つはカーソル結果セットにローを挿入するためのもので、もう1つは結果セットのローからデータベースを更新するためのものです。

```
// Create two statement objects and create a cursor
// for the result set returned by the first
// statement, stmt1. Use stmt1 to execute a query
// and return a cursor result set.
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
stmt1.setCursorName("author_cursor");
ResultSet rs = stmt1.executeQuery("SELECT
    au_id, au_lname, au_fname
FROM authors WHERE city = 'Oakland'
FOR UPDATE OF au_lname");

// Get the name of the cursor created for stmt1 so
// that it can be used with stmt2.
String cursor = rs.getCursorName();

// Use stmt2 to update the database from the
// result set returned by stmt1.
String last_name = new String("Smith");
while(rs.next())
{
    if (rs.getString(1).equals("274-80-9391"))
    {
        stmt2.executeUpdate("UPDATE authors "+
            "SET au_lname = "+last_name +
            "WHERE CURRENT OF " + cursor);
    }
}
```

結果セット内での削除

次の例では前述のコード例にある **Statement** オブジェクト *stmt2* を使用して、位置付け削除を実行します。

```
stmt2.executeUpdate("DELETE FROM authors
    WHERE CURRENT OF " + cursor);
```

JDBC 2.0 メソッドを使用した位置付け更新と削除

この項では、JDBC 2.0 を使用した、現在のカーソル・ローにあるカラムの更新と、結果セット内の現在のカーソル・ローからのデータベースの更新について説明します。それぞれの例を示します。

結果セット内でのカラムの更新

JDBC 2.0 は、メモリ内の結果セットからカラム値を更新するためのメソッドを、クライアントにいくつか指定します。更新された値は、基本となるデータベースで更新、挿入、削除オペレーションを実行するのに使用されます。これらのメソッドはすべて **SybCursorResultSet** クラスに実装されます。

jConnect で使用できる JDBC 2.0 更新メソッドのいくつかの例を示します。

```
void updateAsciiStream(String columnName, java.io.InputStream x,
    int length) throws SQLException;
void updateBoolean(int columnIndex, boolean x) throws
    SQLException;
void updateFloat(int columnIndex, float x) throws SQLException;
void updateInt(String columnName, int x) throws SQLException;
void updateInt(int columnIndex, int x) throws SQLException;

void updateObject(String columnName, Object x) throws
    SQLException;
```

結果セットからデータベースを更新するメソッド

JDBC 2.0 は、結果セットの現在の値に基づいて、データベースのローを更新、または削除するためのメソッドを新たに 2 つ指定します。これらのメソッドは JDBC 1.x の **Statement.executeUpdate()** よりも簡潔なフォームで、カーソル名を必要としません。これは **SybCursorResultSet** に実装されます。

```
void updateRow() throws SQLException;
void deleteRow() throws SQLException;
```

注意 結果セットの同時性は **CONCUR_UPDATABLE** にしてください。そうしないと前述のメソッドは例外を引き起こします。**insertRow()** には、null 以外のエントリを必要とするすべてのテーブル・カラムを指定してください。

DatabaseMetaData で提供されるメソッドは、これらの変更がいつ参照できるかを指示します。

例

次の例ではカーソル結果セットを返すのに使用される1つの **Statement** オブジェクトを作成します。結果セットの各ローについて、カラム値はメモリ内で更新され、データベースがローの新しいカラム値で更新されます。

```
// Create a Statement object and set fetch size to
// 25. This creates a cursor for the Statement
// object Use the statement to return a cursor
// result set.
SybStatement syb_stmt =
(SybStatement)conn.createStatement();
syb_stmt.setFetchSize(25);
SybCursorResultSet syb_rs =
(SybCursorResultSet)syb_stmt.executeQuery(
    "SELECT * from T1 WHERE ...")

// Update each row in the result set according to
// code in the following while loop. jConnect
// fetches 25 rows at a time, until fewer than 25
// rows are left. Its last fetch takes any
// remaining rows.
while(syb_rs.next())
{
    // Update columns 2 and 3 of each row, where
    // column 2 is a varchar in the database and
    // column 3 is an integer.
    syb_rs.updateString(2, "xyz");
    syb_rs.updateInt(3,100);
    //Now, update the row in the database.
    syb_rs.updateRow();
}

// Create a Statement object using the
// JDBC 2.0 method implemented in jConnect 5.x
Statement stmt = conn.createStatement
(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
// Use the Statement to return an updatable ResultSet
ResultSet rs = stmt.executeQuery("SELECT * FROM T1 WHERE...");
// In jConnect 5.x, downcasting to SybCursorResultSet is not
// necessary. Update each row in the ResultSet in the same
// manner as above
while (rs.next())
{
    rs.updateString(2, "xyz");
    rs.updateInt(3,100);
    rs.updateRow();
}
```

結果セットからのローの削除

カーソル結果セットからローを削除するには、**SybCursorResultSet.deleteRow()** を次のように使用します。

```
while(syb_rs.next())
{
    int col3 = getInt(3);
    if (col3 >100)
    {
        syb_rs.deleteRow();
    }
}
```

結果セットへのローの挿入

次の例は JDBC 2.0 API を使用して挿入を行う方法を示します。これは **jdbcConnect 5.x** でのみ実行できます。**SybCursorResultSet** にダウンキャストする必要はありません。

```
// prepare to insert
rs.moveToInsertRow();
// populate new row with column values
rs.updateString(1, "New entry for col 1");
rs.updateInt(2, 42);
// insert new row into db
rs.insertRow();
// return to current row in result set
rs.moveToCurrentRow();
```

準備文でのカーソルの使用

PreparedStatement を一度作成すれば、入力パラメータに同じ値や異なる値を指定して、何度も使用できます。**PreparedStatement** オブジェクトが指定されたカーソルを使用する場合は、カーソルを使用が終わるたびに閉じて、次に使用するときに再度開く必要があります。カーソルは結果セットを閉じると閉じられます (**ResultSet.close()**)。そしてその準備文を実行するときに開かれます (**PreparedStatement.executeQuery()**)。

次の例は **PreparedStatement** オブジェクトを作成し、それにカーソルを割り当て、**PreparedStatement** オブジェクトを二度実行してカーソルを閉じて再度開く方法を説明します。

```
// Create a prepared statement object with a
// parameterized query.
```

```
PreparedStatement prep_stmt =
conn.prepareStatement(
"SELECT au_id, au_lname, au_fname "+
"FROM authors WHERE city = ? "+
"FOR UPDATE OF au_lname");

//Create a cursor for the statement.
prep_stmt.setCursorName("author_cursor");

// Assign the parameter in the query a value.
// Execute the prepared statement to return a
// result set.
prep_stmt.setString(1, "Oakland");
ResultSet rs = prep_stmt.executeQuery();

//Do some processing on the result set.
while(rs.next())
{
    ...
}

// Close the result, which also closes the cursor.
rs.close();

// Execute the prepared statement again with a new
// parameter value.
prep_stmt.setString(1,"San Francisco");
rs = prep_stmt.executeQuery();
// reopens cursor
```

jConnect の SCROLL_INSENSITIVE 結果セットのサポート

jConnect バージョン 5.x は TYPE_SCROLL_INSENSITIVE 結果セットだけをサポートします。

jConnect は、Sybase 固有のプロトコルである Tabular Data Stream (TDS) を使用して Sybase データベース・サーバと通信します。jConnect 5.x では、TDS はカーソルのスクロールをサポートしません。カーソルのスクロールをサポートするために、jConnect は **ResultSet.next()** を呼び出すたびに、必要に応じてロー・データをクライアント上にキャッシュします。しかし、結果セットの終了に到達すると、結果セット全体がクライアントのメモリに格納されます。これはパフォーマンス上の問題を発生させるので、結果セットが比較的小さい場合のみ **TYPE_SCROLL_INSENSITIVE** 結果セットを使用することをおすすめします。

注意 **TYPE_SCROLL_INSENSITIVE ResultSet** を jConnect 5.x で使用するときは、**ResultSet** の最後のローに到達してからでないと **isLast()** メソッドを呼び出せません。最後のローに到達する前に **isLast()** を呼び出すと、**UnimplementedOperationException** が返されます。

jConnect バージョン 4.x には、JDBC 1.0 インタフェースを使用した制限付きの **TYPE_SCROLL_INSENSITIVE ResultSet** を提供するサンプルが追加されています。

この実装は標準の JDBC 1.0 メソッドを使用して、スクロールの影響を受けない読み込み専用の結果セット、つまり、結果セットが開かれている間に行われた変更の影響を受けない基本のデータの静的ビューを生成します。**ExtendedResultSet** はクライアントの **ResultSet** ローをすべてキャッシュします。このクラスを大規模な結果セットで使用する場合は注意してください。

sample.ScrollableResultSet について次に説明します。

- JDBC 1.0 **java.sql.ResultSet** の拡張機能です。
- JDBC 2.0 **java.sql.ResultSet** と同じシグニチャを持つ追加のメソッドを定義します。
- JDBC 2.0 メソッドのすべてを含むわけではありません。含まれないメソッドは **ResultSet** の修正を処理します。

JDBC 2.0 API からの定義されているメソッドを次に示します。

```
boolean previous() throws SQLException;
boolean absolute(int row) throws SQLException;
boolean relative(int rows) throws SQLException;
boolean first() throws SQLException;
boolean last() throws SQLException;
void beforeFirst() throws SQLException;
void afterLast() throws SQLException;
```

```
boolean isFirst() throws SQLException;
boolean isLast() throws SQLException;
boolean isBeforeFirst() throws SQLException;
boolean isAfterLast() throws SQLException;
int getFetchSize() throws SQLException;
void setFetchSize(int rows) throws SQLException;
int getFetchDirection() throws SQLException;
void setFetchDirection(int direction) throws SQLException;
int getType() throws SQLException;
int getConcurrency() throws SQLException;
int getRow() throws SQLException;
```

新しいサンプル・クラスを使用するには、任意の JDBC 1.0 **java.sql.ResultSet** を使用して **ExtendedResultSet** を作成します。関連コードを次に示します (Java 1.1 環境を想定しています)。

```
// import the sample files
import sample.*;
//import the JDBC 1.0 classes
import java.sql.*;
// connect to some db using some driver;
// create a statement and a query;
// Get a reference to a JDBC 1.0 ResultSet
ResultSet rs = stmt.executeQuery(_query);
// Create a ScrollableResultSet with it
ScrollableResultSet srs = new ExtendedResultSet(rs);
// invoke methods from the JDBC 2.0 API
srs.beforeFirst();
// or invoke methods from the JDBC 1.0 API
if (srs.next())
    String column1 = srs.getString(1);
```


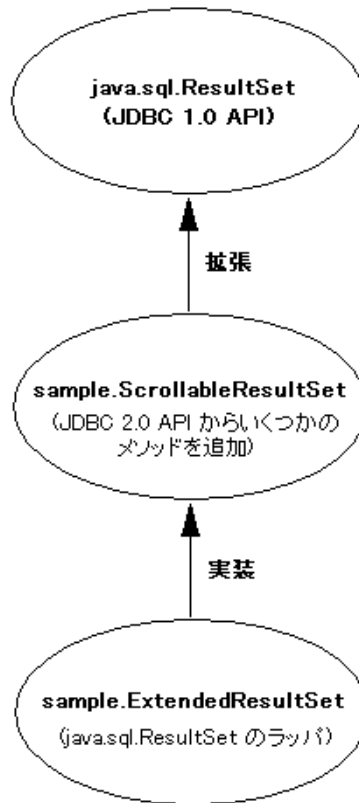
 2-1 は新しいサンプル・クラスと JDBC API の関係を示します。

図 2-1: クラス図



詳細については、<http://java.sun.com/products/jdbc/jdbcse2.html> で JDBC 2.0 API を参照してください。

バッチ更新のサポート

バッチ更新を使用すると **Statement** オブジェクトは複数の更新コマンドをひとつの単位 (バッチ) として基本のデータベースに送信し、一度に処理することができます。

注意 バッチ更新を使用するには、jConnect インストール・ディレクトリ内の `sp` ディレクトリにある SQL スクリプトを更新する必要があります。

Statement、**PreparedStatement**、**CallableStatement** でバッチ更新を使用する場合の例については、*sample* (jConnect 4.x) および *sample2* (jConnect 5.x) サブディレクトリにある *BatchUpdates.java* を参照してください。

jConnect はバッチでの動的 **PreparedStatement**s もサポートしています。

実装上の注意

jConnect は JDBC 2.0 API で指定されているようにバッチ更新を実装しますが、次のような例外があります。

- BatchUpdateException.getUpdateCounts()** を実装するための JDBC 2.0 基準が今後修正されたり緩和された場合、jConnect は **BatchUpdateException.getUpdateCounts()** に長さ $M < N$ の `int[]` を返させることによって、オリジナルの基準を実装し続けます。これはバッチの最初の M 文が成功し、 $M+1$ 文が失敗し、 $M+2..N$ 文 ("N" は、バッチ内にある文の合計数) が実行されなかったことを示します。
- ストアド・プロシージャのバッチ更新
 バッチ (非連鎖) モードでストアド・プロシージャを呼び出すには、非連鎖モードでストアド・プロシージャを作成する必要があります。詳細については、「[ストアド・プロシージャを非連鎖トランザクション・モードでしか実行できない](#)」(107 ページ) を参照してください。
- Adaptive Server Enterprise バージョン 11.5.x 以降
BatchUpdateException.getUpdateCounts() は長さ 0 の `int[]` を返します。エラーが検出された場合はトランザクション全体がロールバックされ、成功したローの数は 0 になります。
- Adaptive Server Enterprise バージョン 11.0.1
 ストアド・プロシージャに対して影響を受けたローの数 0 を返します。
- SQL Anywhere バージョン 5.5.x
 - SQL Anywhere バージョン 5.5.x では、挿入を含むストアド・プロシージャから、挿入されたロー・カウントを取得できません。次に例を示します。

```
create proc sp_A as insert tableA values (1,
'hello A')
```

```
create proc sp_B
as
insert tableA values (1, 'hello A')
update tableA set coll=2
create proc sp_C
as
update tableA set coll=2
delete tableA
```

前述のストアド・プロシージャで **executeBatch** を実行すると、それぞれ次のような結果になります。

```
0 Rows Affected
1 Rows Affected
2 Rows Affected
```

- バッチでの動的 **PreparedStatements** はサポートされていません。
- **SQL Anywhere 5.5.x** は **JDBC 2.0** 仕様に従ったバッチ更新をネイティブでサポートしていないので、バッチ更新は **executeUpdate** ループで実行されます。
- バッチ更新をサポートしていないデータベースでのバッチ更新
jConnect は、データベースがバッチ更新をサポートしていなくても、バッチ更新を **executeUpdate** ループで実行します。これによって、どのデータベースを示しているかに関係なく、同じバッチ・コードを使用できます。

バッチ更新の詳細については、『**Sun Microsystems, Inc. JDBC 2.0 API**』を参照してください。

ストアド・プロシージャの結果セットによるデータベースの更新

jConnect には、ストアド・プロシージャによって返される結果セットに対するカーソルを取得できる更新と削除のメソッドがあります。カーソルの位置を使用することによって、結果セットを提供する基本となるテーブル内のローの更新や削除を行うことができます。このメソッドは **SybCursorResultSet** にあります。

```
void updateRow(String tableName) throws SQLException;
```

```
void deleteRow(String tableName) throws SQLException;
```

tableName パラメータは、結果セットを提供するデータベース・テーブルを示します。

ストアド・プロシージャによって返される結果セットに対するカーソルを取得するには、プロシージャを含んでいる呼び出し可能な文を実行する前に、**SybCallableStatement.setCursorName()** または **SybCallableStatement.setFetchSize()** を使用する必要があります。次の例は、ストアド・プロシージャの結果セットに対してカーソルを作成し、結果セット内の値を更新してから、**SybCursorResultSet.update()** メソッドを使用して基本となるテーブルを更新する方法を示します。

```
// Create a CallableStatement object for executing the stored
// procedure.
CallableStatement sproc_stmt =
    conn.prepareStatement("{call update_titles}");

// Set the number of rows to be returned from the database with
// each fetch. This creates a cursor on the result set.
(SybCallableStatement)sproc_stmt.setFetchSize(10);

//Execute the stored procedure and get a result set from it.
SybCursorResultSet sproc_result = (SybCursorResultSet)
    sproc_stmt.executeQuery();

// Move through the result set row by row, updating values in the
// cursor's current row and updating the underlying titles table
// with the modified row values.
while(sproc_result.next())
{
    sproc_result.updateString(...);
    sproc_result.updateInt(...);
    ...
    sproc_result.updateRow(titles);
}
```

データ型の作業

image データの送信

jConnect は、Adaptive Server Enterprise または Adaptive Server Anywhere データベース内の *image* カラムを更新するための **sendData()** メソッドを持つ **TextPointer** クラスを採用しています。以前のバージョンの jConnect では、イメージ・データは、**java.sql.PreparedStatement** の **setBinaryStream()** メソッドを使用して送信する必要がありました。**TextPointer.sendData()** メソッドは、**java.io.InputStream** を使用して、イメージ・データが Adaptive Server データベースに送信される場合にパフォーマンスを大幅に改善します。

TextPointer クラスのインスタンスを取得するには、次のように **SybResultSet** 内の 2 つの **getTextPtr()** メソッドのいずれかを使用できます。

```
public TextPointer getTextPtr(String columnName)
public TextPointer getTextPtr(int columnIndex)
```

TextPointer クラスのパブリック・メソッド

com.sybase.jdbc パッケージには **TextPointer** クラスが含まれています。このパブリック・メソッド・インタフェースを次に示します。

```
public void sendData(InputStream is, boolean log)
    throws SQLException

public void sendData(InputStream is, int length,
    boolean log) throws SQLException

public void sendData(InputStream is, int offset,
    int length, boolean log) throws SQLException

public void sendData(byte[] byteInput, int offset,
    int length, boolean log) throws SQLException
```

sendData(InputStream is, boolean log) - 指定された入カストリーム内のデータで *image* カラムを更新します。

sendData(InputStream is, int length, boolean log) - 指定された入カストリーム内のデータで *image* カラムを更新します。*length* は送信されるバイト数です。

sendData(InputStream is, int offset, int length, boolean log) - 指定された入カストリーム内のデータで *image* カラムを更新します。*offset* パラメータに指定されたバイト・オフセットから開始して、*length* パラメータに指定されたバイト数まで続行します。

sendData(byte[] byteInput, int offset, int length, boolean log) - byteInput パラメータに指定されたバイト配列に含まれるイメージ・データでカラムを更新します。更新は *offset* パラメータに指定されたバイト・オフセットで開始され、*length* パラメータに指定されたバイト数まで続行されます。

各メソッドには *log* パラメータがあります。*log* パラメータは *image* データがデータベース・トランザクション・ログに完全にログを取られるかどうかを指定します。*log* パラメータが "true" に設定されている場合、バイナリ・イメージ全体がトランザクション・ログに書き込まれます。*log* パラメータが "false" に設定されている場合、更新のログは取られますが、イメージそのものはログに含まれません。

❖ **TextPointer.sendData()** での *image* カラムの更新

イメージ・データでカラムを更新するには、次の手順に従います。

- 1 更新するローとカラムに対する **TextPointer** を取得します。
- 2 **TextPointer.sendData()** を使用して更新を実行します。

次の 2 つの項で例を示します。この例では、*pubs2* データベースにある *au_pix* テーブルの *pic* カラムを更新するために、ファイル *Anne_Ringer.gif* から *image* データが送信されます。更新は作家 ID 899-46-2035 のローに対して行われます。

TextPointer オブジェクトの取得

text および *image* カラムには、タイムスタンプと、このカラムのテキストおよびイメージ・データとは別のページ位置情報が含まれています。データが *text* または *image* カラムから選択されると、この情報は結果セットの一部として隠されます。

image カラムを更新するための **TextPointer** オブジェクトは隠された情報を必要としますが、カラム・データのイメージ部分は必要としません。この情報を取得するには、**ResultSet** オブジェクトにカラムを選択してから、**SybResultSet.getTextPtr()** を使用する必要があります (次のパラグラフの後の例を参照してください)。

SybResultSet.getTextPtr() はテキスト・ポインタ情報を要求し、イメージ・データを無視して **TextPointer** オブジェクトを作成します。

カラムに大量のイメージ・データが含まれている場合、1 つ以上のローに対してカラムを選択してすべてのデータの取得を待つのは、データが使用されていないため、効率的ではありません。**set textsize** コマンドを使用して、パケットで返されるデータの量を少なくすることで、この処理を短縮できます。次の、**TextPointer** オブジェクトを取得するコード例には、これを目的とした **set textsize** の使用方法が含まれています。

```
/*
 * Define a string for selecting pic column data for author ID
 * 899-46-2035.
 */
String getColumnData = "select pic from au_pix where au_id = '899-46-2035'";

/*
 * Use set textsize to return only a single byte of column data
 * to a Statement object. The packet with the column data will
 * contain the "hidden" information necessary for creating a
 * TextPointer object.
 */
Statement stmt= connection.createStatement();
stmt.executeUpdate("set textsize 1");

/*
 * Select the column data into a ResultSet object--cast the
 * ResultSet to SybResultSet because the getTextPtr method is
 * in SybResultSet, which extends ResultSet.
 */
SybResultSet rs = (SybResultSet)stmt.executeQuery(getColumnData);

/*
 * Position the result set cursor on the returned column data
 * and create the desired TextPointer object.
 */
rs.next();
TextPointer tp = rs.getTextPtr("pic");

/*
 * Now, assuming we are only updating one row, and won't need
 * the minimum textsize set for the next return from the server,
 * we reset textsize to its default value.
 */
stmt.executeUpdate("set textsize 0");
```

TextPointer.sendData 次のコード例では、前述の項の **TextPointer** オブジェクトを使用して、
での更新の実行 ファイル *Anne_Ringer.gif* 内のイメージ・データで *pic* カラムを更新します。

```
/*
 * First, define an input stream for the file.
 */
FileInputStream in = new FileInputStream("Anne_Ringer.gif");

/*
 * Prepare to send the input stream without logging the image data
```

```
* in the transaction log.  
*/  
boolean log = false;  
  
/*  
* Send the image data in Anne_Ringer.gif to update the pic  
* column for author ID 899-46-2035.  
*/  
tp.sendData(in, log);
```

詳細については、jConnect インストール・ディレクトリの *sample* (jConnect 4.x) および *sample2* (jConnect 5.x) サブディレクトリ内にある *TextPointers.java* サンプルを参照してください。

Date データ型と Time データ型の使用方法

JDBC は、時間に関するデータ型を 3 つ使用します。それらは、Time、Date、Timestamp です。Adaptive Server は、時間に関するデータ型、datetime をひとつだけ使用します。これは、JDBC の Timestamp データ型に相当します。サーバの datetime データ型は、1/300 秒の精度をサポートします。

3 つの JDBC データ型はすべて、サーバ側では datetime データ型として扱われます。JDBC の Timestamp は、基本的にサーバの datetime と同じです。したがって、変換の必要はありません。ただし、JDBC の Time または Date データ型とサーバの datetime データ型との交換は変換を必要とします。

- Time を datetime に変換するには、1970 年 1 月 1 日の日付を加算します。
- Date を datetime に変換するには、"00:00:00" を追加します。
- datetime を Date 変数または Time 変数に変換するには、使用されない情報を削除します。

実装上の注意

- JDBC の Timestamp データ型と Adaptive Server の timestamp データ型は同じではありません。Adaptive Server の timestamp データ型は、「オブティミスティック同時実行性」方式を使用して更新を行うときに使用するユニークな varbinary 値です。

- 値を **Time** データ型として挿入する場合は、日付部分は本質的に無意味になります。したがって、この値は **Time** データ型だけを使用して取り出す必要があり、**Date** や **Timestamp** データ型を使用して取り出さないでください。
- **Adaptive Server Anywhere** の *date* または *time column* で **getObject()** を使用すると、値は **JDBC Timestamp** データ型として返されます。

Char / Varchar / Text データ型および getObject()

データが 16 進数、8 進数、または 10 進数以外の場合は、**char**、**varchar**、または **text** フィールドで **rs.getBytes()** を使用しないでください。

高度な機能の実装

この項では、jConnect の高度な機能を使用する方法について説明します。次の項目について説明します。

- イベント通知の使用方法
- エラー・メッセージの処理
- テーブル内のカラム・データとしての Java オブジェクトの格納
- 動的クラスのロード
- JDBC 2.0 Optional Package の拡張機能のサポート

イベント通知の使用方法

jConnect のイベント通知機能は、Open Server プロシージャが実行される時にアプリケーションが通知されるようにするために使用できます。

この機能を使用するには、**Connection** インタフェースを拡張した **SybConnection** クラスを使用する必要があります。**SybConnection** は、イベント通知をオンにするための **regWatch()** メソッドと、イベント通知をオフにするための **regNoWatch()** メソッドを含んでいます。

アプリケーションには、**SybEventHandler** インタフェースも実装する必要があります。このインタフェースには、1 つのパブリック・メソッド **void event(String proc_name, ResultSet params)** が含まれています。これは、指定されたイベントが発生すると呼び出されます。イベントのパラメータは **event()** に渡されて、アプリケーションに応答方法を通知します。

アプリケーションでイベント通知を使用するには、**SybConnection.regWatch()** を呼び出して、アプリケーションをレジスタード・プロシージャの通知リストに登録します。次の構文を使用します。

```
SybConnection.regWatch(proc_name,eventHdlr,option)
```

- *proc_name* は、通知を生成するレジスタード・プロシージャの名前である String です。
- *eventHdlr* は、実装する **SybEventHandler** クラスのインスタンスです。

- *option* は、NOTIFY_ONCE または NOTIFY_ALWAYS のいずれかで
す。NOTIFY_ONCE は、プロシージャがはじめて実行されるとき
にだけアプリケーションに通知させたい場合に使用します。
NOTIFY_ALWAYS は、プロシージャが実行されるたびにアプリ
ケーションに通知させたい場合に使用します。

指定された *proc_name* のイベントが Open Server で発生するたびに、
jConnect は別のスレッドから **eventHdlr.event()** を呼び出します。イベ
ント・パラメータは、実行時に **eventHdlr.event()** に渡されます。これ
は別のスレッドなので、イベント通知がアプリケーションの実行をブ
ロックすることはありません。

proc_name がレジスタード・プロシージャでないか、Open Server がク
ライアントを通知リストに追加できない場合は、**regWatch()** への呼び
出しで SQL 例外が発生します。

イベント通知をオフにするには、次の呼び出しを使用します。

```
SybConnection.regNoWatch(proc_name)
```

注意 Sybase イベント通知拡張機能を使用する場合、アプリケーション
は接続上で **close()** メソッドを呼び出して、**regWatch()** への最初の呼
び出しによって作成された子スレッドを削除する必要があります。こ
れを実行しないと、アプリケーションを終了するときに仮想マシンが
ハングすることがあります。

イベント通知の例

次の例はイベント・ハンドラを実装し、接続後にイベント・ハンドラ
のインスタンスでイベントを登録する方法について説明します。

```
public class MyEventHandler implements SybEventHandler
{
    // Declare fields and constructors, as needed.
    ...
    public MyEventHandler(String eventname)
    {
        ...
    }

    // Implement SybEventHandler.event.
    public void event(String eventName, ResultSet params)
    {
        try
        {
```

```

        // Check for error messages received prior to event
        // notification.
        SQLWarning sqlw = params.getWarnings();
        if (sqlw != null)
        {
            // process errors, if any
            ...
        }
        // process params as you would any result set with
        // one row.
        ResultSetMetaData rsmd = params.getMetaData();
        int numColumns = rsmd.getColumnCount();
        while (params.next())           // optional
        {
            for (int i = 1; i <= numColumns; i++)
            {
                System.out.println(rsmd.getColumnName(i) + " = " +
                                   params.getString(i));
            }
            // Take appropriate action on the event. For example,
            // perhaps notify application thread.
            ...
        }
    }
    catch (SQLException sqe)
    {
        // process errors, if any
        ...
    }
}

public class MyProgram
{
    ...
    // Get a connection and register an event with an instance
    // of MyEventHandler.
    Connection conn = DriverManager.getConnection(...);
    MyEventHandler myHdlr = new MyEventHandler("MY_EVENT");

    // Register your event handler.
    ((SybConnection)conn).regWatch("MY_EVENT", myHdlr,
        SybEventHandler.NOTIFY_ALWAYS);
    ...
    conn.regNoWatch("MY_EVENT");
    conn.close();
}

```

```
}
```

エラー・メッセージの処理

jConnect は、Sybase 固有のエラー情報を返すための **SybSQLException** と **SybSQLWarning** という 2 つのクラス、および jConnect がサーバから受信したエラー・メッセージを処理する方法をカスタマイズできる **SybMessageHandler** インタフェースを提供します。

Sybase 固有のエラー情報の検索

jConnect は Sybase 固有のエラー情報を取得するためのメソッドを指定する **EedInfo** インタフェースを提供します。**EedInfo** インタフェースは **SQLException** および **SQLWarning** クラスを拡張する **SybSQLException** と **SybSQLWarning** に実装されています。

SybSQLException と **SybSQLWarning** は次のメソッドを含みます。

- **public ResultSet getEedParams();**
エラー・メッセージに付随するパラメータ値を含む、1 ローの結果セットを返します。
- **public int getStatus();**
メッセージにパラメータ値がある場合は "1" を、ない場合は "0" を返します。
- **public int getLineNumber();**
エラー・メッセージの原因となったストアド・プロシージャまたはクエリの行番号を返します。
- **public String getProcedureName();**
エラー・メッセージの原因となったプロシージャの名前を返します。
- **public String getServerName();**
エラー・メッセージを生成したサーバの名前を返します。
- **public int getSeverity();**
エラー・メッセージの重大度を返します。
- **public int getState();**

サーバ内のエラー・メッセージの内部ソースに関する情報を返します。これは Sybase 製品の保守契約を結んでいるサポート・センタでのみ使用されます。

- **public int getTranState();**

次のいずれかのトランザクション・ステータスを返します。

- 0 接続は現在拡張トランザクションにあります。
- 1 直前のトランザクションは正常にコミットされました。
- 3 直前のトランザクションはアボートされました。

SybSQLException または **SybSQLWarning** とはならず、**SQLException** や **SQLWarning** メッセージとなるメッセージがあることに注意してください。アプリケーションが、処理している例外のタイプを確認してから **SybSQLException** または **SybSQLWarning** にダウンキャストするようにしてください。

エラー・メッセージ処理のカスタマイズ

SybMessageHandler インタフェースを使用して、サーバに生成されたエラー・メッセージを **jConnect** が処理する方法をカスタマイズできます。エラー・メッセージを処理するために自分のクラスに **SybMessageHandler** を実装すると、次のような利点があります。

- "ユニバーサルな" エラー処理
エラー処理論理を、アプリケーションの間中繰り返すのではなく、使用しているエラー・メッセージ・ハンドラに置くことができます。
- "ユニバーサルな" エラー・ロギング
エラー・メッセージ・ハンドラには、すべてのエラー・ロギングを処理するための論理を含むことができます。
- アプリケーション稼働条件に基づいた、エラー・メッセージ重大度の再マップ

エラー・メッセージ・ハンドラには、特定のエラー・メッセージを認識して、その重大度をサーバの重大度レベルではなく、アプリケーションの評価に基づいてダウングレードしたりアップグレードするための論理を含むことができます。たとえば、古いローを削除するクリーンアップ・オペレーションを行っているあいだにローが存在しないメッセージの重大度をダウングレードしたい場合、他の環境での重大度をアップグレードする場合などです。

注意 **SybMessageHandler** インタフェースを実装するエラー・メッセージ・ハンドラはサーバが生成したメッセージだけを受け取ります。**jConnect** によって生成されたメッセージは処理しません。

jConnect はエラー・メッセージを処理するときに、メッセージを処理するために **SybMessageHandler** クラスが追加されたかどうかを確認します。追加されていた場合、**jConnect** は **messageHandler()** メソッドを呼び出します。**messageHandler()** メソッドは SQL 例外を引数として受け入れ、**jConnect** は **messageHandler()** からどの値が返されたかに基づいてメッセージを処理します。エラー・メッセージ・ハンドラは次のことを行います。

- SQL 例外をそのまま返します。
- **null** を返します。結果として、**jConnect** はメッセージを無視します。
- SQL 例外から SQL 警告を作成し、返します。これによって警告メッセージ・チェーンに警告が追加されます。
- 最初のメッセージが SQL 警告の場合、**messageHandler()** は SQL 警告を緊急だと考え、制御がいったん **jConnect** に返されると返される SQL 例外を作成して返します。

エラー・メッセージ・ハンドラのインストール

SybDriver、**SybConnection**、または **SybStatement** から **setMessageHandler()** メソッドを呼び出すことによって、**SybMessageHandler** を実装するエラー・メッセージ・ハンドラをインストールできます。**SybDriver** からエラー・メッセージ・ハンドラをインストールした場合は、後続のすべての **SybConnection** オブジェクトに継承されます。**SybConnection** オブジェクトからエラー・メッセージ・ハンドラをインストールした場合は、その **SybConnection** によって作成されたすべての **SybStatement** オブジェクトに継承されます。

この階層はエラー・メッセージ・ハンドラ・オブジェクトがインストールされた時点からだけ適用されます。たとえば、**SybConnection** オブジェクト、*myConnection* を作成してから **SybDriver.setMessageHandler()** を呼び出してエラー・メッセージ・ハンドラ・オブジェクトをインストールすると、*myConnection* はこのオブジェクトを使用できません。

現在のエラー・メッセージ・ハンドラ・オブジェクトを返すには、**getMessageHandler()** を使用してください。

エラー・メッセージ・ハンドラの例

次の例では jConnect バージョン 4.1 を使用します。

```
import java.io.*;
import java.sql.*;
import com.sybase.jdbcx.SybMessageHandler;
import com.sybase.jdbcx.SybConnection;
import com.sybase.jdbcx.SybStatement;
import java.util.*;

public class MyApp
{
    static SybConnection conn = null;
    static SybStatement stmt = null
    static ResultSet rs = null;
    static String user = "guest";
    static String password = "sybase";
    static String server = "jdbc:sybase:Tds:192.138.151.39:4444";
    static final int AVOID_SQLLE = 20001;

    public MyApp()
    {
        try
        {
```

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance();
Properties props = new Properties();
props.put("user", user);
props.put("password", password);
conn = (SybConnection)
DriverManager.getConnection(server, props);
conn.setMessageHandler(new NoResultSetHandler());
stmt =(SybStatement) conn.createStatement();
stmt.executeUpdate("raiserror 20001 'your error'");

for (SQLWarning sqw = _stmt.getWarnings();
    sqw != null;
    sqw = sqw.getNextWarning());
{
    if (sqw.getErrorCode() == AVOID_SQLLE);
    {
        System.out.println("Error" +sqw.getErrorCode()+
            " was found in the Statement's warning list.");
        break;
    }
}
stmt.close();
conn.close();
}
catch(Exception e)
{
    System.out.println(e.getMessage());
    e.printStackTrace();
}
}

class NoResultSetHandler implements SybMessageHandler
{

    public SQLException messageHandler(SQLException sqe)
    {
        int code = sqe.getErrorCode();
        if (code == AVOID_SQLLE)
        {
            System.out.println("User " + _user + " downgrading " +
                AVOID_SQLLE + " to a warning");
            sqe = new SQLWarning(sqe.getMessage(),
                sqe.getSQLState(),sqe.getErrorCode());
        }
        return sqe;
    }
}
```

```
}

public static void main(String args[])
{
    new MyApp();
}
```

テーブル内のカラム・データとしての Java オブジェクトの格納

データベース製品には、Java オブジェクトをデータベース内のカラム・データとして直接格納できるものもあります。このようなデータベースでは、Java クラスはデータ型として扱われ、Java クラスをそのデータ型として持つカラムを宣言できます。

jConnect では、**PreparedStatement** インタフェースに定義された **setObject()** メソッドと、**CallableStatement** および **ResultSet** インタフェースに定義された **getObject()** メソッドを実装することによって、Java オブジェクトをデータベースに格納することができます。これによって、ネイティブの JDBC クラスおよびメソッドを使用して Java オブジェクトをカラム・データとして直接格納したり取り出すアプリケーションで jConnect を使用することができます。

注意 **getObject()** および **setObject()** メソッドを使用するには、jConnect バージョンを **VERSION_4** 以上に設定してください。詳細については、[「jConnect バージョンの設定」\(6 ページ\)](#) を参照してください。

このあとの項では、オブジェクトをテーブルに格納し、jConnect で JDBC を使用して取り出すための条件と手順について説明します。

- [Java オブジェクトをカラム・データとして格納するための前提条件](#)
- [Java オブジェクトをデータベースに送信する方法](#)
- [Java オブジェクトをデータベースから受信する方法](#)

注意 Adaptive Server Enterprise バージョン 12.0 および Adaptive Server Anywhere バージョン 6.0.x 以降は、Java オブジェクトをテーブルに格納できますが、いくつかの制限があります。詳細については『[jConnect for JDBC リリース・ノート](#)』を参照してください。

Java オブジェクトをカラム・データとして格納するための前提条件

ユーザ定義の Java クラスに属している Java オブジェクトをカラム内に格納するには、次の 3 つの条件を満たす必要があります。

- クラスは **java.io.Serializable** インタフェースを実装している必要があります。これは jConnect がネイティブの Java 直列化／直列化解除を使用してデータベースとの間のオブジェクトの送受信を行うためです。
- クラス定義は送信先データベースにインストールされる必要があります。または、**DynamicClassLoader (DCL)** を使用して Adaptive Server Anywhere または Adaptive Server Enterprise サーバからクラスを直接ロードし、ローカルの **CLASSPATH** に存在しているかのようにそれを使用する必要があります。詳細については、「[動的クラスのロード](#)」(78 ページ) を参照してください。
- クライアント・システムは、ローカルの **CLASSPATH** 環境変数を経由してアクセスできる **.class** ファイルにクラス定義を持っている必要があります。

Java オブジェクトをデータベースに送信する方法

ユーザ定義クラスのインスタンスをカラム・データとして送信するには、**PreparedStatement** インタフェースに指定されているように、次のいずれかの **setObject()** メソッドを使用します。

```
void setObject(int parameterIndex, Object x, int targetSqlType,  
    int scale) throws SQLException;  
void setObject(int parameterIndex, Object x, int targetSqlType)  
    throws SQLException;  
void setObject(int parameterIndex, Object x) throws SQLException;
```

次の例では **Address** クラスを定義し、データ型が **Address** クラスの **Address** カラムを持つ **Friends** テーブルの定義を示し、テーブルにローを挿入します。

```
public class Address implements Serializable  
{  
    public String streetNumber;  
    public String street;  
    public String apartmentNumber;  
    public String city;  
    public int zipCode;  
    //Methods  
    ...  
}
```

```
/* This code assumes a table with the following structure
** Create table Friends:
** (firstname varchar(30),
**  lastname varchar(30),
**  address Address,
**  phone varchar(15))
*/

// Connect to the database containing the Friends table.
Connection conn =
    DriverManager.getConnection("jdbc:sybase:Tds:localhost:5000",
        "username", "password");

// Create a Prepared Statement object with an insert statement
//for updating the Friends table.
PreparedStatement ps = conn.prepareStatement("INSERT INTO
    Friends values (?, ?, ?, ?)");

// Now, set the values in the prepared statement object, ps.
// set firstname to "Joan."
ps.setString(1, "Joan");

// Set last name to "Smith."
ps.setString(2, "Smith");

// Assuming that we already have "Joan_address" as an instance
// of Address, use setObject(int parameterIndex, Object x) to
// set the address column to "Joan_address."
ps.setObject(3, Joan_address);

// Set the phone column to Joan's phone number.
ps.setString(4, "123-456-7890");

// Perform the insert.
ps.executeUpdate();
```

Java オブジェクトをデータベースから受信する方法

クライアント JDBC アプリケーションは結果セット内のデータベースから、またはストアド・プロシージャから返される出力パラメータの値として Java オブジェクトを受け取ることができます。

- Java オブジェクトがカラム・データとして結果セットに含まれている場合は、**ResultSet** インタフェース内の次のいずれかの **getObject()** メソッドを使用してオブジェクトを取り出します。

```
Object getObject(int columnIndex) throws SQLException;
Object getObject(String columnName) throws SQLException;
```

- Java オブジェクトがストアド・プロシージャからの出力パラメータに含まれている場合は、**CallableStatement** インタフェース内の次の **getObject()** メソッドを使用してオブジェクトを取り出します。

```
Object getObject(int parameterIndex) throws SQLException;
```

次の例では **ResultSet.getObject(int parameterIndex)** を使用して、結果セット内で受け取ったオブジェクトをクラス変数に割り当てます。この例では前述の項で使用した **Address** クラスと *Friends* テーブルを使用しており、封筒に名前と住所を印刷する簡単なアプリケーションを示しています。

```
/*
** This application takes a first and last name, gets the
** specified person's address from the Friends table in the
** database, and addresses an envelope using the name and
** retrieved address.
*/
public class Envelope
{
    Connection conn = null;
    String firstName = null;
    String lastName = null;
    String street = null;
    String city = null;
    String zip = null;

    public static void main(String[] args)
    {
        if (args.length < 2)
        {
            System.out.println("Usage: Envelope <firstName>
                                <lastName>");
            System.exit(1);
        }
        // create a 4" x 10" envelope
        Envelope e = new Envelope(4, 10);
        try
        {
            // connect to the database with the Friends table.
            conn = DriverManager.getConnection(
                "jdbc:sybase:Tds:localhost:5000", "username",
                "password");
```

```

        // look up the address of the specified person
        firstName = args[0];
        lastName = args[1];
        PreparedStatement ps = conn.prepareStatement(
            "SELECT address FROM friends WHERE " +
            "firstname = ? AND lastname = ?");
        ps.setString(1, firstName);
        ps.setString(2, lastName);
        ResultSet rs = ps.executeQuery();
        if (rs.next())
        {
            Address a = (Address) rs.getObject(1);
            // set the destination address on the envelope
            e.setAddress(firstName, lastName, a);
        }
        conn.close();
    }
    catch (SQLException sqe)
    {
        sqe.printStackTrace();
        System.exit(2);
    }
    // if everything was successful, print the envelope
    e.print();
}

private void setAddress(String fname, String lname, Address a)
{
    street = a.streetNumber + " " + a.street + " " +
        a.apartmentNumber;
    city = a.city;
    zip = "" + a.zipCode;
}

private void print()
{
    // Print the name and address on the envelope.
    ...
}
}

```

詳細については、jConnect ディレクトリの *sample* (jConnect 4.x) および *sample2* (jConnect 5.x) サブディレクトリ内にある **HandleObject.java** の例を参照してください。

動的クラスのロード

Adaptive Server Anywhere バージョン 6.0 および Adaptive Server Enterprise バージョン 12.0 では、SQL (JCS) で Java クラスが提供されており、これによって Java クラスを次のように指定できます。

- SQL カラムのデータ型
- Transact-SQL 変数のデータ型
- SQL カラムのデフォルト値

以前は、jConnect の CLASSPATH に表示されるクラスだけにしかアクセスできず、jConnect アプリケーションがローカルの CLASSPATH がないクラスのインスタンスにアクセスしようとするとき **java.lang.ClassNotFound** 例外が発生しました。

jConnect バージョン 5.2 は、**DynamicClassLoader** (DCL) を実装して Adaptive Server Anywhere または Adaptive Server Enterprise サーバからクラスを直接ロードし、ローカルの CLASSPATH に存在しているかのようにそれを使用します。

スーパークラスに存在するすべてのセキュリティ機能は継承されます。Java 2 に実装されているローダ委任モデルに従います。つまり、jConnect は、要求されたクラスを CLASSPATH からロードしようとし、それが失敗した場合は **DynamicClassLoader** を試します。

JCS と Adaptive Server の詳細については、『Adaptive Server Enterprise Version 12.0 Feature Overview』を参照してください。

DynamicClassLoader の使用

DCL 機能を使用するには、次の手順に従います。

- 1 クラス・ローダを作成して設定します。jConnect アプリケーションのコードは、次のようになります。

```
Properties props = new Properties();

// URL of the server where the classes live.
String classesUrl = "jdbc:sybase:Tds:myase:1200";

// Connection properties for connecting to above server.
props.put("user", "grinch");
props.put("password", "meanone");
...

// Ask the SybDriver for a new class loader.
DynamicClassLoader loader = driver.getClassLoader(classesUrl, props);
```

- 2 **CLASS_LOADER** 接続プロパティを使用して、クエリを実行する文が新しいクラス・ローダを使用できるようにします。クラス・ローダを作成したら、次のコード例(手順1のコード例の続き)で示しているように、それを後続の接続に渡します。

```
// Stash the class loader so that other connection(s)
// can know about it.
props.put("CLASS_LOADER", loader);

// Additional connection properties
props.put("user", "joeuser");
props.put("password", "joespassword");

// URL of the server we now want to connect to.
String url = "jdbc:sybase:Tds:jdbc.sybase.com:4446";

// Make a connection and go.
Connection conn = DriverManager.getConnection(url, props);
```

次の Java クラス定義を想定しています。

```
class Addr {
    String street;
    String city;
    String state;
}
```

また、次の SQL テーブル定義を想定しています。

```
create table employee (char(100) name, int empid, Addr address)
```

- 3 クライアント・アプリケーションの **CLASSPATH** に **Addr** クラスが存在しない場合は、次のクライアント側コードを使用します。

```
Statement stmt = conn.createStatement();
// Retrieve some rows from the table that has a Java class
// as one of its fields.
ResultSet rs = stmt.executeQuery(
    "select * from employee where empid = '19'");
if (rs.next()) {
    // Even though the class is not in our class path,
    // we should be able to access its instance.
    Object obj = rs.getObject("address");
    // The class has been loaded from the server,
    // so let's take a look.
    Class c = obj.getClass();
    // Some Java Reflection can be done here
    // to access the fields of obj.
    ...
}
```

CLASS_LOADER 接続プロパティには、1つのクラス・ローダを複数の接続で共有するための便利な機能があります。

1つのクラス・ローダを複数の接続で共有した場合に、クラスの競合が発生しないことを確認する必要があります。たとえば、2つの異なるデータベースに、クラス **org.foo.Bar** の2つ互換性のないインスタンスが存在する場合、同じローダを使用して両方のクラスにアクセスすると問題が発生します。最初のクラスは、最初の接続から返された結果セットを検査するときにロードされます。このため、2番目の接続から返された結果セットを検査するときには、すでにクラスはロードされています。この結果、2番目のクラスはロードされず、jConnectはこの状況を直接検出できません。

ただし、Javaには、クラスのバージョンが非直列化オブジェクト内のバージョン情報と一致していることを保証するメカニズムが組み込まれています。このため、前述の状況は、少なくともJavaによって検出され、報告されます。

クラスとそのインスタンスは、同一のデータベースやサーバに常駐している必要はありません。ただし、ローダと後続の接続は同一のデータベースやサーバを参照できます。

直列化解除

次の例は、オブジェクトをローカル・ファイルから直列化解除する方法を示しています。直列化オブジェクトは、サーバ上に常駐するクラスのインスタンスで、**CLASSPATH**には存在しません。

SybResultSet.getObject() は、デフォルトのシステム ("boot") クラス・ローダではなく、**DynamicClassLoader** からクラス定義をロードする **ObjectInputStream** のサブクラスである **DynamicObjectInputStream** を使用します。

```
// Make a stream on the file containing the
//serialized object.
FileInputStream fileStream = new FileInputStream("serFile");
// Make a "deserializer" on it. Notice that, apart
//from the additional parameter, this is the same
//as ObjectInputStreamDynamicObjectInputStream
stream = new DynamicObjectInputStream(fileStream, loader);
// As the object is deserialized, its class is
//retrieved via the loader from our server.
Object obj = stream.readObject();stream.close();
```

JARS のプリロード

jConnect バージョン 5.2 には、**PRELOAD_JARS** という新しい接続プロパティがあります。JAR ファイル名のカンマで区切られたリストとして定義した場合、**JARS** ファイルはそれらのエンティティにロードされます。ここでは、"JAR" は、サーバが使用する「保持された JARname」を表します。この JAR 名は、次に示すような Java のインストール・プログラムで指定します。

```
install java new jar 'myJarName' from file '/tmp/mystuff.jar'
```

PRELOAD_JARS を設定すると、**JARS** はクラス・ローダに関連付けられるため、接続ごとにそれらをプリロードする必要がなくなります。**PRELOAD_JARS** は、1 つの接続でだけ指定します。同じ **JARS** を繰り返しプリロードしようとする、JAR データがサーバから不必要に取り出されるため、パフォーマンスの問題が生じることがあります。

注意 Adaptive Server Anywhere 6.x 以降は、JAR ファイルを 1 つのエンティティとして返せないため、jConnect は各クラスを順に取り出します。ただし、Adaptive Server 12.x 以降は、JAR 全体を取り出し、それに含まれている各クラスをロードします。

高度な機能

DynamicClassLoader にはさまざまなパブリック・メソッドがあります。詳細については、次の javadoc 情報を参照してください。

JDBC_HOME/docs/en/javadocs

高度な機能には、一連のクラスのロードが予想される場合に、ローダのデータベース接続を確立したままにしておく機能や、名前を指定して単一のクラスを明示的にロードする機能があります。

また、**java.lang.ClassLoader** から継承したパブリック・メソッドを使用することもできます。クラスのロードを処理する **java.lang.Class** 内のメソッドも使用できますが、これらのメソッドのいくつかは使用するクラス・ローダを想定するため、注意して使用する必要があります。特に、**Class.forName()** の 3 引数バージョンを使用する必要があります。これを使用しないと、システム ("boot") クラス・ローダが使用されます。詳細については、「[エラー・メッセージの処理](#)」(68 ページ)を参照してください。

JDBC 2.0 Optional Package の拡張機能のサポート

『JDBC 2.0 Optional Package』(以前の『JDBC 2.0 Standard Extension API』)は、JDBC 2.0 ドライバによって実装されるいくつかの新しい機能を定義しています。jConnect バージョン 5.2 は、次のオプション・パッケージの拡張機能を実装しています。

- [ネーミング・データベース用の JNDI](#)
(jConnect がサポートしているあらゆる Sybase DBMS で動作)
- [Connection Pooling](#)
(jConnect がサポートしているあらゆる Sybase DBMS で動作)
- [分散トランザクション管理サポート](#)
(Adaptive Server Enterprise バージョン 12.0、または XA-Server を使用しているバージョン 11.x でのみ動作)

上記の機能は、標準の Java 2 の配布には含まれていないクラスまたはインタフェース、あるいはその両方を必要とします。ネーミング・データベース用の JNDI と接続プーリングを実装するには、**javax.sql.*** と **javax.naming.*** をダウンロードする必要があります。また、分散トランザクション管理サポートを実装するには、**javax.transaction.xa.*** をダウンロードする必要があります。

注意 Java 1.1.6 以降と互換性がある JNDI 1.2 を使用することをおすすめします。

ネーミング・データベース用の JNDI

リファレンス

『JDBC 2.0 Optional Package』(以前の『JDBC 2.0 Standard Extension API』)、第 5 章の「JNDI and the JDBC API」。

関連インタフェース

- **javax.sql.DataSource**
- **javax.naming.Referenceable**
- **javax.naming.spi.ObjectFactory**

この機能により、JDBC クライアントは標準以外の手順でデータベース接続を行えるようになります。クライアントは、**Class.forName** ("com.sybase.jdbc2.jdbc.SybDriver") を呼び出して、**DriverManager** の **getConnection()** メソッドに JDBC URL を渡す代わりに、論理名を使用して JNDI ネーム・サーバにアクセスして、**javax.sql.DataSource** オブジェクトを取得できます。このオブジェクトは、ドライバをロードし、それが表している物理データベースへの接続を確立します。**DataSource** オブジェクト内にベンダ固有の情報が入っているため、クライアント・コードはより簡潔になり、再使用可能になります。

Sybase の **DataSource** オブジェクトの実装は、**com.sybase.jdbcx.SybDataSource** です (詳細については、javadoc を参照してください)。この実装は、JavaBean コンポーネントのデザイン・パターンを使用している次の標準プロパティをサポートしています。

- **databaseName**
- **dataSourceName**
- **description**
- **networkProtocol**
- **password**
- **portNumber**
- **serverName**
- **user**

roleName はサポートされていません。

jConnect は、**DataSource** オブジェクトをネーム・サーバ・エントリの属性から構成できるように **javax.naming.spi.ObjectFactory** インタフェースの実装を提供しています。**javax.naming.Reference**、または **javax.naming.Name** および **javax.naming.DirContext** を指定した場合、このファクトリは **com.sybase.jdbcx.SybDataSource** オブジェクトを構成できます。このファクトリを使用するには、**com.sybase.jdbc2.SybObjectFactory** を含むように **java.naming.object.factory** システム・プロパティを設定します。

使用方法

DataSource は、さまざまな方法およびアプリケーションで使用できます。すべてのオプションの説明といくつかのコード例を以降に示します。詳細については『JDBC 2.0 Optional Package』(以前の『JDBC 2.0 Standard Extension API』)および Sun Microsystems 社の Web サイトにある JNDI のマニュアルを参照してください。

1a. 管理者による設定 ::LDAP

jConnect は、バージョン 4.0 以降で LDAP 接続をサポートしています。このため、LDIF フォーマットを使用して **DataSourcees** を LDAP のエントリとして設定することをおすすめします。これには、カスタム・ソフトウェアは必要ありません。次に例を示します。

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
```

1b. クライアントによる アクセス

これは一般的な JDBC クライアント・アプリケーションです。唯一異なる点は、**DriverManager** にアクセスして JDBC URL を提供する代わりに、ネーム・サーバにアクセスして **DataSource** オブジェクトへの参照を取得することです。接続が確立されたら、クライアント・コードは他の JDBC クライアント・コードと同様に使用できます。コードは非常に汎用的で、環境の一部として設定できるオブジェクト・ファクトリ・プロパティを設定する場合だけ Sybase を参照します。.

jConnect インストール環境には、**DataSource** の使用方法を説明しているサンプル・プログラム *sample2/SimpleDataSource.java* が含まれています。このサンプルは、リファレンスとして提供されています。つまり、独自の環境を設定して、適切に編集しないとこのサンプルは実行できません。*SimpleDataSource.java* には、次の重要なコードが含まれています。

```
import javax.naming.*;
import javax.sql.*;
import java.sql.*;

// set necessary JNDI properties for your environment (same as above)
Properties jndiProps = new Properties();

// used by JNDI to build the SybDataSource
jndiProps.put(Context.OBJECT_FACTORIES,
    "com.sybase.jdbc2.jdbc.SybObjectFactory");

// nameserver that JNDI should talk to
jndiProps.put(Context.PROVIDER_URL,
    "ldap://some_ldap_server:238/o=MyCompany,c=Us");
```

```
// used by JNDI to establish the naming context
jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");

// obtain a connection to your name server
Context ctx = new InitialContext(jndiProps);
DataSource ds = (DataSource) ctx.lookup("servername=myASE");

// obtains a connection to the server as configured earlier.
// in this case, the default username and password will be used
Connection conn = ds.getConnection();

// do standard JDBC methods
...
```

プロパティが仮想マシン上ですでに定義されている場合は、**Properties** を **InitialContext** コンストラクタに明示的に渡す必要はありません。この場合は、Java が起動されたときに次のコードによって渡されます。

```
java -Djava.naming.object.factory=com.sybase.jdbc2.jdbc.SybObjectFactory
```

または、ブラウザ・プロパティの一部として設定されます。

環境プロパティの詳細については、Java VM のマニュアルを参照してください。

2a. システム管理者による設定: custom

このフェーズは、通常、データベース・システムの管理またはアプリケーションの統合を担当している方が実行します。データ・ソースを定義して、それをネーム・サーバの論理名の下に展開することを目的としています。サーバを再設定する必要がある場合(たとえば、他のマシン、ポートなどに移動した場合)、管理者はこの設定ユーティリティ(概要については下記を参照)を実行して、論理名を新しいデータ・ソース設定に再割り当てします。クライアント・コードには論理名しか含まれていないため、クライアント・コードは変更されません。

```
import javax.sql.*;
import com.sybase.jdbcx.*;
.....

// create a SybDataSource, and configure it
SybDataSource ds = new com.sybase.jdbc2.jdbc.SybDataSource();
ds.setUser("my_username");
ds.setPassword("my_password");
ds.setDatabaseName("my_favorite_db");
ds.setServerName("db_machine");
```

```
ds.setPortNumber(4000);
ds.setDescription("This DataSource represents the Adaptive Server
    Enterprise server running on db_machine at port 2638. The default
    username and password have been set to 'me' and 'mine' respectively.
    Upon connection, the user will access the my_favorite_db database on
    this server.");
Properties props = new Properties();
props.put("REPEAT_READ", "false");
props.put("REQUEST_HA_SESSION", "true");
ds.setConnectionProperties(props);
// store the DataSource object. Typically this is
// done by setting JNDI properties specific to the
// type of JNDI service provider you are using.
// Then, initialize the context and bind the object.
Context ctx = new InitialContext();
ctx.bind("jcbc/myASE", ds);
```

DataSource を設定したら、情報の格納場所と格納方法を決定する必要があります。利便性を考慮して、**SybDataSource** は、**java.io.Serializable** および **javax.naming.Referenceable** の両方になっていますが、管理者は、JNDI に使用しているサービス・プロバイダに従ってデータの格納方法を決定できます。

2b. クライアントによるアクセス

クライアントは、**DataSource** を展開したのと同じ方法で JNDI プロパティを設定して、**DataSource** オブジェクトを取得します。クライアントは、オブジェクトを Java オブジェクトに格納するときに変換（たとえば、直列化）することができるオブジェクト・ファクトリを持っている必要があります。

```
Context ctx = new InitialContext();
DataSource ds = (DataSource ctx.lookup("jcbc/myASE"));
```

Connection Pooling

リファレンス

『JDBC 2.0 Optional Package』（以前の『JDBC 2.0 Standard Extension API』）、第 6 章の「接続プーリング」。

関連インタフェース

- **javax.sql.ConnectionPoolDataSource**
- **javax.sql.PooledConnection**

概要

従来のデータベース・アプリケーションは、データベースに対して 1 つの接続を確立し、アプリケーションの各セッションでこれを使用します。これに対して、Web ベースのデータベース・アプリケーションは、アプリケーションの使用中に新しい接続を何回もオープンしたりクローズする必要があります。Web ベースのデータベース接続を効率的に処理するには、接続プーリングを使用します。接続プーリングは、オープンしているデータベース接続を管理したり、異なるユーザ要求間で共有している接続を管理することで、パフォーマンスを維持し、アイドル状態になっている接続の数を削減します。各接続要求で、接続プールは、まずプール内にアイドル状態の接続があるかどうかを判断します。アイドル状態の接続がある場合、接続プールは、データベースへの新しい接続を確立する代わりにその接続を返します。

接続プーリング機能は、**ConnectionPoolDataSource** によって提供されます。このインタフェースを使用した場合は、接続をプールできません。**DataSource** インタフェースを使用した場合は、接続をプールできません。

ConnectionPoolDataSource を使用した場合、プールの実装は **PooledConnection** を受信します。ユーザが接続をクローズしたとき、または接続を破壊するエラーがユーザに存在する場合、実装にこれが通知されます。この時点で、プールの実装は **PooledConnection** をどのように処理するかを決定します。

接続プーリングを使用しない場合、トランザクションは次のことを行います。

- 1 データベースへの接続を作成します。
- 2 データベースにクエリを送信します。
- 3 結果セットを受け取ります。
- 4 結果セットを表示します。
- 5 接続を削除します。

接続プーリングを使用する場合、トランザクションは次のことを行います。

- 1 未使用の接続が接続の「プール」に存在しているかどうか確認します。
- 2 存在している場合は、それを使用し、存在しない場合は新しい接続を確立します。

- 3 データベースにクエリを送信します。
- 4 結果セットを受け取ります。
- 5 結果セットを表示します。
- 6 接続を「プール」に返します。
(ユーザは "**close()**" を呼び出しますが、接続はオープンしたままになり、プールにはクローズ要求が通知されます)。

クライアントがデータベースへの接続を確立する必要がある場合に、毎回新しい接続を確立するよりも接続を再使用したほうがコストが少なく済みます。

サード・パーティが接続プールを実装できるように、jConnect の実装には、**ConnectionPoolDataSource** インタフェースが生成する **PooledConnections** があります。これは、**DataSource** インタフェースが **Connections** を生成するのと類似した方法で生成されます。

プールの実装は、**ConnectionPoolDataSource** の **getPooledConnection()** メソッドを使用して、実際のデータベース接続を作成します。そして、プールの実装は、それ自体を **PooledConnection** へのリスナとして登録します。

現在は、クライアントが接続を要求すると、プールの実装は使用可能な **PooledConnection** で **getConnection()** を開始します。クライアントが接続を使い終わって **close()** を呼び出すと、プールの実装は、接続が解放されていて再使用可能であることを **ConnectionEventListener** インタフェースから通知されます。

また、クライアントが何らかの方法でデータベース接続を破壊し、プールの実装がその接続をプールから削除できる場合、プールの実装は **ConnectionEventListener** インタフェースからも通知されます。

詳細については、『JDBC 2.0 Optional Package』(以前の『JDBC 2.0 Standard Extension API』)を参照してください。

管理者による設定:
LDAP

この方法は、LDIF エントリに行を追加することを除けば **ネーミング・データベース用の JNDI** で説明している「**1a. 管理者による設定: :LDAP**」と同じです。次の例では、追加したコード行を太字で示しています。

```
dn:servername=myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.18:ConnectionPoolDataSource
```

中間層クライアントによるアクセス

このプロシージャは、3つのプロパティ (84 ページに示している INITIAL_CONTEXT_FACTORY、PROVIDER_URL、および OBJECT_FACTORIES) を初期化し、**ConnectionPoolDataSource** オブジェクトを取り出します。完全なコード例については、*sample2/SimpleConnectionPool.java* を参照してください。基本的に異なる部分は、次のとおりです。

```
...
ConnectionPoolDatabase cpds = (ConnectionPoolDataSource)
    ctx.lookup("servername=myASE");
PooledConnection pconn = cpds.getPooledConnection();
```

分散トランザクション管理サポート

この機能は、Adaptive Server Enterprise バージョン 12.x または XA-Server を使用しているバージョン 11.x で分散トランザクションを実行するための標準の Java API を提供します。

注意 この機能は、大規模な多層環境で使用するよう設計されています。

リファレンス

『JDBC 2.0 Optional Package』(以前の『JDBC 2.0 Standard Extension API』) の第 7 章の「分散トランザクション」を参照してください。

関連インタフェース

- `javax.sql.XADataSource`
- `javax.sql.XAConnection`
- `javax.transaction.xa.XAResource`

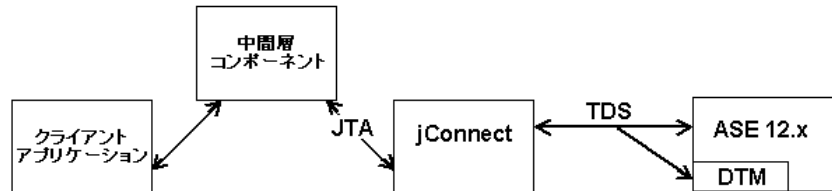
バックグラウンド情報とシステム稼働条件

Adaptive Server Enterprise 12.0 の場合

- jConnect は Sybase Adaptive Server Enterprise バージョン 12.0 内のリソース・マネージャと直接通信するため、インストール環境に分散トランザクション管理サポートがインストールされている必要があります。
- 分散トランザクションに参加したいユーザに、"dtm_tm_role" が付与されている必要があります。付与されていない場合、トランザクションは失敗します。

- 分散トランザクションを使用するには、`/sp` ディレクトリにストアド・プロシージャをインストールする必要があります。『jConnect for JDBC インストール・ガイド』の第1章の「ストアド・プロシージャのインストール」を参照してください。

図 2-2: バージョン 12.x での分散トランザクション管理サポート

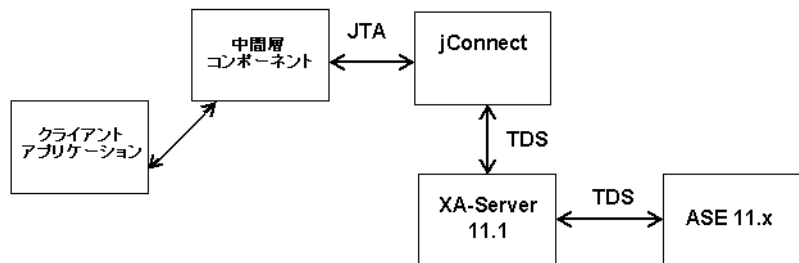


Adaptive Server Enterprise 11.x の場合

jConnect は、データベース・サーバに Adaptive Server Enterprise バージョン 11.x を使用している場合に分散トランザクションを実行するための標準の Java API も提供しています。

- この実装は、Sybase Adaptive Server Enterprise バージョン 11.x および XA-Server 11.1 でだけ動作します。

図 2-3: バージョン 11.x での分散トランザクション管理サポート



- 選択したログインに、デフォルトのログイン・データベースとして *master*、*model*、または *sybssystemdb* を設定してはいけません。これは、ユーザの作業が分散トランザクションに関連付けられている場合だけ XA-Server が接続されると、これらのデータベース上で分散トランザクションが許可されていないためです。
- メタデータへはアクセスできません。これはクライアントに対する制限ではありますが、通常これは分散トランザクションで使用する API の機能ではありません。

Adaptive Server Enterprise 12.x の使用

管理者による設定 : この方法は、LDIF エントリに行を追加することを除けば「ネーミング・データベース用の JNDI」(82 ページ) で説明している「1a. 管理者による設定 : :LDAP」と同じです。次の例では、追加したコード行を太字で示しています。

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.18:XADataSource
```

中間層クライアントによるアクセス このプロシージャは、3つのプロパティ (INITIAL_CONTEXT_FACTORY、PROVIDER_URL、および OBJECT_FACTORIES) を初期化し、**XADataSource** オブジェクトを取り出します。次に例を示します。

```
...
XADataSource xads = (XADataSource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection();
```

または、次に示すように、指定したユーザ名とパスワードのデフォルト設定を上書きします。

```
...
XADataSource xads = (XADataSource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection("my_username", "my_password");
```

Adaptive Server Enterprise 11.x の使用

管理者による設定 : この方法は、LDIF エントリに 3 行追加することを除けば「ネーミング・データベース用の JNDI」(82 ページ) で説明している「1a. 管理者による設定 : :LDAP」と同じです。

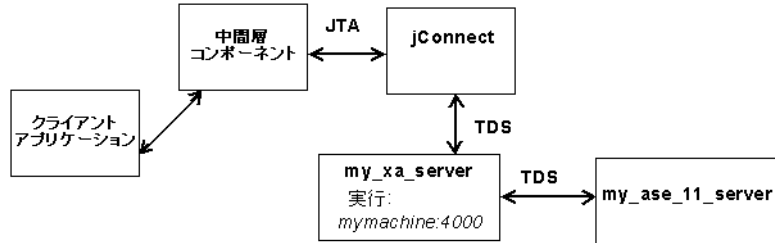
次の例では、追加したコード行を太字で示しています。

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.16:userconnection
1.3.6.1.4.1.897.4.2.17:1
1.3.6.1.4.1.897.4.2.18:XADataSource
```

ここで、...**4.2.17:1** は、jConnect が XA-Server に接続し、Logical Resource Manager (LRM) に対応する **userconnection** を使用することを示しています。XA-Server には、次のエントリを含んでいる *xa_config* ファイルがあります。

```
[xa]
lrm=userconnection
server=my_ase_11_server
XAserver=my_xa_server
```

図 2-4: 分散トランザクション管理サポートの設定例



xa_config ファイルへの書き込み方法については、XA-Server のマニュアルを参照してください。

中間層クライアントによるアクセス

このプロシージャは、3つのプロパティ (INITIAL_CONTEXT_FACTORY、PROVIDER_URL、および OBJECT_FACTORIES) を初期化し、**XADataSource** オブジェクトを取り出します。次に例を示します。

```
...
XADataSource xads = (XADataSource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection();
```

Adaptive Server Enterprise 11.x では、デフォルトのユーザ名とパスワードは上書きできません。つまり、次のコードは呼び出せません。

```
xads.getXAConnection("my_username", "my_password");
```

これは、*lrm* が特定のユーザ名とパスワードに関連付けられているためです。

Sybase 固有の処理上の制約、制限及び JDBC 規格外の実装

この項では、Sybase の jConnect がどのようにして JDBC 1.x および 2.0 規格外の実装を含み、jConnect に適用される制約と制限について説明します。次の項目について説明します。

- マルチスレッドに対する調整
- `ResultSet.setCursorName()` の使用
- 大きなパラメータ値での `setLong()` の使用
- `COMPUTE` 文の使用
- ストアド・プロシージャの実行

マルチスレッドに対する調整

おすすめしませんが、同一の **Statement**、**CallableStatement**、または **PreparedStatement** でいくつかのスレッドが同時にメソッドを呼び出す場合、**Statement** 上でのメソッドへの呼び出しを手動で同期化させる必要があります。jConnect はこれを自動的には行いません。

たとえば、同一の **Statement** 上で 2 つのスレッドを操作していて、1 つのスレッドがクエリの送信を、もう 1 つが警告の処理を実行している場合、**Statement** 上のこのメソッドへの呼び出しを同期化させないと、矛盾が発生する可能性があります。

ResultSet.setCursorName() の使用

JDBC ドライバのなかには、常に文字列が返されるよう、SQL クエリにカーソル名を生成するものがあります。ただし、次のいずれかを行っていないと、jConnect は `ResultSet.setCursorName()` が呼び出されるときに名前を返しません。

- 対応する **Statement** 上で `setFetchSize()` または `setCursorName()` を呼び出した。
- `SELECT_OPENS_CURSOR` 接続プロパティを "true" に設定し、クエリは次に示すような `SELECT...FOR UPDATE` のフォームを使用した。

```
select au_id from authors for update
```

対応する Statement で **setFetchSize()** または **setCursorName()** を呼び出さない場合、または **SELECT_OPEN_CURSOR** 接続プロパティを "true" に設定しない場合は、null が返されます。

JDBC 2.0 API (第 11 章「Clarifications」) によると、他のすべての SQL 文はカーソルを開いて名前を返す必要はありません。

jConnect でのカーソルの使用方法については、「[結果セットでのカーソルの使用方法](#)」(45 ページ) を参照してください。

大きなパラメータ値での **setLong()** の使用

PreparedStatement.setLong() メソッドの実装は、パラメータ値を SQL BIGINT データ型に設定します。Adaptive Server データベースのほとんどは 8 バイト BIGINT データ型を持っていません。パラメータ値に 4 バイトを超える BIGINT が必要な場合、**setLong()** を使用するとオーバフロー例外を引き起こす可能性があります。

COMPUTE 文の使用

jConnect は計算ローをサポートしていません。クエリに計算ローが含まれていると、結果は自動的にキャンセルされます。たとえば、次の文は拒否されます。

```
SELECT name FROM sysobjects
WHERE type="S" COMPUTE COUNT(name)
```

この問題を回避するには、次のコードに置き換えます。

```
SELECT name from sysobjects WHERE type="S"
SELECT COUNT(name) from sysobjects WHERE type="S"
```

ストアド・プロシージャの実行

- **CallableStatement** オブジェクト内でパラメータ値を疑問符として表現するストアド・プロシージャを実行すると、パラメータに疑問符とリテラルの両方を使用した場合よりもよいパフォーマンスが得られます。さらに、リテラルと疑問符を混合した場合はストアド・プロシージャに出力パラメータを使用できません。

次の例はストアド・プロシージャ **MyProc** を実行するための **CallableStatement** オブジェクトとして *sp_stmt* を作成します。

```
CallableStatement sp_stmt = conn.prepareCall(
    "{call MyProc(?,?)}" );
```

MyProc 内の2つのパラメータは疑問符として表記されています。**CallableStatement** インタフェースの **registerOutParameter()** メソッドを使用して、これらのいずれか、または両方を出力パラメータとして登録できます。

次の例では、*sp_stmt2* がストアド・プロシージャ **MyProc2** を実行するための **CallableStatement** オブジェクトです。

```
CallableStatement sp_stmt2 = conn.prepareCall(
    "{call MyProc2(?, 'javelin')}" );
```

sp_stmt2 では、1つのパラメータがリテラルとして指定され、もう1つが疑問符として指定されています。いずれのパラメータも、出力パラメータとして登録することはできません。

- パラメータのネーム・バインドを使用した **RPC** コマンドでのストアド・プロシージャを実行するには、次のいずれかのプロシージャを使用します。
- **PreparedStatement** クラスを使用する Java 変数から直接入力パラメータを渡す言語コマンドを使用します。次のコード例にこれを示します。

```
// Prepare the statement
System.out.println("Preparing the statement...");
String stmtString = "exec " + procname + " @p3=?, @p1=?";
PreparedStatement pstmt = con.prepareStatement(stmtString);

// Set the values
pstmt.setString(1, "xyz");
pstmt.setInt(2, 123);

// Send the query
System.out.println("Executing the query...");
ResultSet rs = pstmt.executeQuery();
```

- **jConnect** バージョン 5.2 では、次の例で示している **com.sybase.jdbcx.SybCallableStatement** インタフェースを使用します。

```
import com.sybase.jdbcx.*;
....
// prepare the call for the stored procedure to execute as an RPC
String execRPC = "{call " + procName + " (?, ?)}";
SybCallableStatement scs = (SybCallableStatement)
con.prepareCall(execRPC);
```

```
// set the values and name the parameters
// also (optional) register for any output parameters
scs.setString(1, "xyz");
scs.setParameterName(1, "@p3");
scs.setInt(2, 123);
scs.setParameterName(2, "@p1");

// execute the RPC
// may also process the results using getResultSet()
// and getMoreResults()

// see the samples for more information on processing results
ResultSet rs = scs.executeQuery();
```

第 3 章

トラブルシューティング

この章では、jConnect を使用しているときに発生することがある問題の解決法と対処方法について説明します。

この章では次の項目について説明します。

項名	ページ
jConnect でのデバッグ	98
TDS 通信の取得	102
接続不成功エラー	104
jConnect アプリケーションでのメモリ使用率	106
ストアド・プロシージャのエラー	107
カスタム・ソケット実装エラー	109

jConnect でのデバッグ

jConnect には、一連のデバッグ機能を含んでいる **Debug** クラスがあります。**Debug** メソッドには、さまざまな `assert`、`trace`、`timer` 関数が含まれています。これらの関数を使用して、デバッグ処理の適用範囲と、デバッグ結果の出力の送信先を定義できます。

jConnect のインストール環境には、デバッグに使用できる一連の詳細なクラスが含まれます。これらのクラスは、jConnect インストール・ディレクトリの下に *devclasses* サブディレクトリに置かれます。デバッグ時には、jConnect 標準の *classes* ディレクトリではなく、デバッグ・モード・ランタイム・クラス (jConnect 4.x の場合は *devclasses*、jConnect 5.x の場合は *devclasses/jconn2d.jar*) を参照するように、`CLASSPATH` 環境変数をリダイレクトする必要があります。これは、Java プログラムを実行するときに `java` コマンドに `-classpath` 引数を明示的に使用することによっても同様に行えます。

Debug クラスのインスタンスの取得

jConnect のデバッグ機能を使用するには、アプリケーションは **Debug** インタフェースをインポートし、**SybDriver** クラスの `getDebug()` メソッドを呼び出すことによって、**Debug** クラスのインスタンスを取得する必要があります。

jConnect 4.x の場合：

```
import com.sybase.jdbcx.Debug
import com.sybase.jdbcx.SybDebug
//
...
SybDriver sybDriver = (SybDriver)
    Class.forName("com.sybase.jdbc.SybDriver").newInstance();
Debug sybdebug = sybDriver.getDebug();
...
```

jConnect 5.x の場合：

```
import com.sybase.jdbcx.Debug
import com.sybase.jdbcx.SybDebug
//
...
SybDriver sybDriver = (SybDriver)
    Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
Debug sybdebug = sybDriver.getDebug();
...
```

アプリケーションのデバッグをオンにする方法

Debug オブジェクト上で **debug()** メソッドを使用してアプリケーション内でデバッグをオンにするには、次の呼び出しを追加してください。

```
sybdebug.debug(true, [classes], [printstream]);
```

classes パラメータは、デバッグしたい特定のクラスをコロンで区切ってリストする文字列です。次に例を示します。

```
sybdebug.debug(true, "MyClass")
```

および

```
sybdebug.debug(true, "MyClass:YourClass")
```

クラス文字列内の "STATIC" は、指定したクラスのほかに jConnect 内のすべての **static** メソッドに対するデバッグをオンにします。次に例を示します。

```
sybdebug.debug(true, "STATIC:MyClass")
```

"ALL" を指定すると、すべてのクラスに対するデバッグをオンにできます。次に例を示します。

```
sybdebug.debug(true, "ALL");
```

printstream パラメータはオプションです。printstream を指定しない場合は、デバッグ出力は **DriverManager.setLogStream()** で指定された出力ファイルに対して行われます。

アプリケーションのデバッグをオフにする方法

デバッグをオフにするには、次の呼び出しを追加します。

```
sybdebug.debug(false);
```

デバッグ用に CLASSPATH を設定する方法

デバッグが有効なアプリケーションを実行する前に、jConnect インストール・ディレクトリの */devclasses* サブディレクトリを参照するように CLASSPATH 環境変数を再定義します。

jConnect 4.x の場合：

- UNIX の場合は、`$JDBC_HOME/classes` ではなく `$JDBC_HOME/devclasses` を使用します。
- Windows の場合は、`%JDBC_HOME%\classes` ではなく `%JDBC_HOME%\devclasses` を使用します。

jConnect 5.x の場合：

- UNIX の場合は、`$JDBC_HOME/classes/jconn2.jar` ではなく `$JDBC_HOME/devclasses/jconn2.jar` を使用します。
- Windows の場合は、`%JDBC_HOME%\classes\jconn2.jar` ではなく `%JDBC_HOME%\devclasses\jconn2.jar` を使用します。

Debug メソッドの使用

デバッグ処理をカスタマイズするために、ほかの **Debug** メソッドの呼び出しを追加することもできます。

次のメソッドでは、呼び出し元のオブジェクトを指定するために、最初の (オブジェクト) パラメータとして通常は *this* を指定します。これらのメソッドの 1 つでも **static** の場合には、オブジェクト・パラメータに *null* を使用してください。

- **println()**

このメソッドは、デバッグが有効で、デバッグするクラスのリストにオブジェクトが含まれている場合に、出力ログに出力するメッセージを定義するために使用します。デバッグ出力は、**sybdebug.debug()** で指定されたファイルに対して行われます。

構文は次のとおりです。

```
sybdebug.println(object,message string);
```

次に例を示します。

```
sybdebug.println(this,"Query: "+ query);
```

このメッセージに似たメッセージを出力ログに生成します。

```
myApp(thread[x,y,z]): Query: select * from authors
```

- **assert()**

このメソッドは、条件を指定して、その条件に合わないときにランタイム例外を発生させるために使用します。条件に合わない場合に出力ログに出力するメッセージを定義することもできます。構文は次のとおりです。

```
sybdebug.assert(object,boolean condition,message
string);
```

次に例を示します。

```
sybdebug.assert(this,amount<=buf.length,amount+"
too big!");
```

この例は、"amount" が *buf.length* の値を超えると、出力ログに次のようなメッセージを出力します。

```
java.lang.RuntimeException:myApp(thread[x,y,z]):
Assertion failed: 513 too big!
at jdbc.sybase.utils.sybdebug.assert(
sybdebug.java:338)
at myApp.myCall(myApp.java:xxx)
at .... more stack:
```

- **startTimer()**
stopTimer()

これらのメソッドは、イベント中に経過する時間をミリ秒単位で計測するタイマを開始したり停止したりするために使用します。このメソッドは、各オブジェクトごとに1つと、すべての静的メソッド用に1つのタイマを保持します。次に、タイマを開始させる構文を示します。

```
sybdebug.startTimer(object);
```

次に、タイマを停止させる構文を示します。

```
sybdebug.stopTimer(object,message string);
```

次に例を示します。

```
sybdebug.startTimer(this);
stmt.executeQuery(query);
sybdebug.stopTimer(this,"executeQuery");
```

このメッセージに似たメッセージを出力ログに生成します。

```
myApp(thread[x,y,z]):executeQuery elapsed time =
25ms
```

TDS 通信の取得

Tabular Data Stream (TDS) は、クライアント・アプリケーションと Adaptive Server 間の通信を処理する Sybase 固有のプロトコルです。jConnect には、ロー TDS パケットをファイルに取得できる `PROTOCOL_CAPTURE` 接続プロパティが含まれています。

アプリケーションで問題が発生して、アプリケーションでもサーバでも解決できない場合、`PROTOCOL_CAPTURE` を使用してクライアントとサーバ間の通信をファイルに取得できます。直接解釈できないバイナリ・データを含んだファイルを解析するために、Sybase 製品の保守契約を結んでいるサポート・センタにそのファイルを送ることもできます。

注意 `Ribo` ユーティリティを使用して、クライアントとサーバの間を通信しているプロトコル・ストリームを取得、表示、および変換することもできます。`Ribo` の取得方法と使用方法については、jConnect ユーティリティ Web ページを参照してください。

<http://www.sybase.com/products/internet/jconnect/utilities/>

PROTOCOL_CAPTURE 接続プロパティ

`PROTOCOL_CAPTURE` 接続プロパティは、アプリケーションと Adaptive Server 間で交換された TDS パケットを受信するのに使います。`PROTOCOL_CAPTURE` は、動作をすぐに開始するので、接続の確立中に交換された TDS パケットも指定したファイルに書き込まれます。`Capture.pause()` が実行されるかまたはセッションが閉じられるまで、すべてのパケットがファイルに書き込まれます。

次の例では、`PROTOCOL_CAPTURE` を使用して TDS データを `tds_data` ファイルに送ります。

```
...
props.put("PROTOCOL_CAPTURE", "tds_data")
Connection conn = DriverManager.getConnection(url, props);
```

`url` は接続 URL、また `props` は指定している接続プロパティの **Properties** オブジェクトです。

Capture クラスでの `pause()` メソッドと `resume()` メソッド

Capture クラスは、**com.sybase.jdbcx** パッケージにあります。これには2つのパブリック・メソッドが含まれています。

- **public void pause()**
- **public void resume()**

Capture.pause() は、ロー TDS パケットのファイルへの取得を停止します。**Capture.resume()** は取得を再開します。

セッション全体の TDS 取得ファイルは、非常に大きくなることがあります。アプリケーション内で TDS データを取得する場所がわかっていて、取得ファイルのサイズを制限したい場合、次のことを行うことができます。

- 1 接続を確立した直後に、接続用の **Capture** オブジェクトを入手し、**pause()** メソッドを使用して TDS データの取得を停止します。

```
Capture cap = ((SybConnection)conn).getCapture();  
cap.pause();
```

- 2 TDS データの取得を開始する場所に **cap.resume()** を置いてください。
- 3 TDS データの取得を停止する場所に **cap.pause()** を置いてください。

接続不成功エラー

この項では、接続を確立するときやゲートウェイを起動するときが発生する問題について説明します。

ゲートウェイ接続が拒否される

```
Gateway connection refused:  
HTTP/1.0 502 Bad Gateway|Restart Connection
```

このエラー・メッセージは、Adaptive Server に接続するために使用された *hostname* または *port#* に関してなにか問題があることを示します。\$SYBASE/interfaces (UNIX の場合) または %SYBASE%\ini\sql.ini (Windows の場合) の [query] エントリを調べてください。

hostname と *port#* を確認したあとも引き続き問題が発生する場合は、'verbose' システム・プロパティを使用して HTTP サーバを起動することによってさらに情報を得ることができます。

Windows の場合、DOS プロンプトで次のように入力します。

```
httpd -Dverbose=1 > filename
```

UNIX の場合、次のように入力します。

```
sh httpd.sh -Dverbose=1 > filename &
```

filename は、デバッグ・メッセージの出力ファイルです。

Cascade HTTP ゲートウェイを使用しないで接続している場合は、Web サーバが Connect メソッドをサポートしていない可能性があります。データベース・サーバにプロキシとしてパスを提供する Cascade HTTP ゲートウェイを使用しない場合、アプレットはそれがダウンロードされたホストに対してだけ接続できます。

Cascade HTTP ゲートウェイと Web サーバは同じホストで稼働する必要があります。この場合、アプレットは、要求を適切なデータベースにルート指定する Cascade HTTP ゲートウェイによって制御されるポートを使用して同じマシンやホストに接続できます。

これがどのように行われるかを調べるには、jConnect インストール・ディレクトリの *sample* (jConnect 4.x) または *sample2* (jConnect 5.x) サブディレクトリ内の *Isql.java* と *gateway.html* のソースを確認してください。"proxy" を検索します。

SQL Server 4.9.2 に接続できない

jConnect は、TDS 5.0 (Sybase 転送プロトコル) を使用します。SQL Server 4.9.x は TDS 4.6 を使用しますが、これは TDS 5.0 とは互換性がありません。

jConnect で使用するには、SQL Server 10.0.2 以降の SQL Server が必要です。

jConnect アプリケーションでのメモリ使用率

次に示す状況とその解決法を覚えておくと、jConnect アプリケーション内でメモリ使用率が増加したときに役立ちます。

- jConnect アプリケーションでは、文がメモリ内で累積しないように、すべての **Statement** オブジェクトとサブディレクトリ（たとえば **PreparedStatement**、**CallableStatement**）を、最後に使用したあとで明示的に閉じるようにしてください。**ResultSet** を閉じるのでは十分ではありません。

次に例を示します。

```
ResultSet rs = _conn.prepareCall(_query).execute();
...
rs.close();
```

これは問題を発生します。代わりに次のようにします。

```
PreparedStatement ps = _conn.prepareCall(_query);
ResultSet rs = ps.execute();
...
ps.close();
rs.close();
```

- jConnect は、Sybase 固有のプロトコルである Tabular Data Stream (TDS) を使用して Sybase データベース・サーバと通信します。jConnect 5.0 では、TDS はカーソルのスクロールをサポートしません。カーソルのスクロールをサポートするために、jConnect は **ResultSet.next()** を呼び出すたびに、必要に応じてロー・データをクライアント上にキャッシュします。しかし、結果セットの終了に到達すると、結果セット全体がクライアントのメモリに格納されます。これはパフォーマンス上の問題を発生させるので、結果セットが比較的小さい場合のみ **TYPE_SCROLL_INSENSITIVE** 結果セットを使用することをおすすめします。

ストアド・プロシージャのエラー

この項では、jConnect とストアド・プロシージャを使用するときに発生する問題について説明します。

RPC が登録数よりも少ない出力パラメータを返す

```
SQLState: JZ0SG - An RPC did not return as many output parameters as the application had registered for it.
```

このエラーは、ストアド・プロシージャ内の "OUTPUT" パラメータとして宣言したよりも多いパラメータに対して

CallableStatement.registerOutParam() を呼び出すと発生します。

"OUTPUT" として適切なパラメータがすべて宣言されていることを確認してください。読み込んだコードの行を確認してください。

```
create procedure yourproc (@p1 int OUTPUT, ...
```

注意 Adaptive Server Anywhere (以前の SQL Anywhere) を使用しているときにこのエラーが発生した場合は、Adaptive Server Anywhere バージョン 5.5.04 以降にアップグレードしてください。

ストアド・プロシージャが出力パラメータを返す場合のフェッチとステータスのエラー

クエリがロー・データを返さない場合は、**executeQuery()** メソッドではなく **CallableStatement.executeUpdate()** メソッドまたは **execute()** メソッドを使用してください。

JDBC 標準で必要とされるように、**executeQuery()** に結果セットがない場合、jConnect は SQL 例外を発生します。

ストアド・プロシージャを非連鎖トランザクション・モードでしか実行できない

```
Sybase Error 7713 - Stored Procedure can only be executed in unchained transaction mode.
```

JDBC は **autocommit(true)** モードで接続しようとします。アプリケーションは、**Connection.setAutoCommit(false)** または **"set chained on"** 言語コマンドを使用することによって接続を連鎖モードに変更できます。このエラーは、ストアド・プロシージャが互換モードで作成されていない場合に発生します。

この問題を修正するには、次のシステム・プロシージャを使用します。

```
sp_procxmode procedure_name, "anymode"
```

カスタム・ソケット実装エラー

sun.security.ssl.SSLSocketImpl.setEnabledCipherSuites を呼び出しているときに SSL ソケットを設定しようとしている間に次のような例外を受け取ったとします。

```
java.lang.IllegalArgumentException:  
SSL_SH_anon_EXPORT_WITH_RC4_40_MDS
```

この場合は SSL ライブラリがシステム・ライブラリ・パスにあるかどうか確認してください。

第 4 章

パフォーマンスおよびチューニング

この章では `jConnect` で作業するときのパフォーマンスの微調整または改善の方法について説明します。

次の項目について説明します。

項名	ページ
jConnect パフォーマンスの改善	112
動的 SQL の準備文のパフォーマンス・チューニング	115
カーソルのパフォーマンス	122

jConnect パフォーマンスの改善

jConnect を使用するアプリケーションのパフォーマンスを最適化する方法は数多くあります。次にいくつかのガイドラインを示します。

- テキストおよびイメージ・データを Adaptive Server データベースに送信するには、**TextPointer.sendData()** メソッドを使用します。詳細については、「[image データの送信](#)」(60 ページ) を参照してください。
- セッション中に何度も使用される動的 SQL 文については、プリコンパイルされた **PreparedStatement** オブジェクトを作成します。詳細については、「[動的 SQL の準備文のパフォーマンス・チューニング](#)」(115 ページ) を参照してください。
- バッチ更新は、ネットワーク・トラフィックを軽減することによってパフォーマンスを改善します。具体的には、すべてのクエリが 1 つのグループとしてサーバに送信され、クライアントに返されるすべての応答が 1 つのグループとして送信されます。詳細については、「[バッチ更新のサポート](#)」(56 ページ) を参照してください。
- イメージ・データ、大量のロー・セット、長いテキスト・データを転送する可能性のあるセッションの場合は、**PACKETSIZE** 接続プロパティを使用して可能な最大の packetsize を設定します。
- TDS-tunneled HTTP の場合は、最大 TDS packetsize を設定して Web サーバが HTTP1.1 KeepAlive 機能をサポートするように設定します。*SkipDoneProc* サブレットの引数を "true" に設定してください (145 ページ参照)。
- **LANGUAGE_CURSOR** 接続プロパティのデフォルト設定であるプロトコル・カーソルを使用します。詳細については、「[LANGUAGE_CURSOR 接続プロパティ](#)」(122 ページ) を参照してください。
- **TYPE_SCROLL_INSENSITIVE** 結果セットを使用する場合は、結果セットが適度に小さい場合だけにしてください。詳細については、「[jConnect の SCROLL_INSENSITIVE 結果セットのサポート](#)」(53 ページ) を参照してください。

パフォーマンスを改善するには、次に説明するようにさらにガイドラインがあります。

BigDecimal の再位取り

JDBC 1.0 仕様では、`getBigDecimal()` の位取り要素が必要です。この場合 **BigDecimal** オブジェクトは、サーバから返されると `getBigDecimal()` で使用したオリジナルの位取り要素を使用して、再度位取りされなければなりません。

再度の位取りに必要な時間を短縮するには、JDBC 2.0 の `getBigDecimal()` メソッドを使用します。これは `jConnect` が **SybResultSet** クラスに実装するもので、*scale* 値を必要としません。

```
public BigDecimal getBigDecimal(int columnIndex)
    throws SQLException
```

次に例を示します。

```
SybResultSet rs =
    (SybResultSet)stmt.executeQuery("SELECT
    numeric_column from T1");
while (rs.next())
{
    BigDecimal bd rs.getBigDecimal(
        "numeric_column");
    ...
}
```

REPEAT_READ 接続プロパティ

REPEAT_READ 接続プロパティを "false" に設定している場合は、データベースから結果セットを取り出すときのパフォーマンスを改善できます。ただし、REPEAT_READ が "false" の場合、次のような条件があります。

- カラム・インデックスに従って、カラム値を順番どおりに読み込まなければなりません。カラム番号ではなく名前でカラムにアクセスする場合、この方法は困難です。
- 1つのカラム値を、1つのロー内で複数回読み込むことはできません。

文字セット変換

マルチバイト文字セットを使用していてドライバ・パフォーマンスを改善する必要がある場合は、jConnect サンプルで提供される **SunloConverter** クラスを使用できます。このコンバータは Sun Microsystems, Inc の Java Software Division が提供している **sun.io** クラスに基づいています。

SunloConverter クラスは文字セット・コンバータ機能の pure Java 実装ではないので、標準の jConnect 製品には組み込まれていません。ただし、このコンバータ・クラスは参考として提供されており、jConnect ドライバで使用して文字セット変換のパフォーマンスを改善することはできます。

注意 Sybase でのテストでは、テスト対象のすべての VM のパフォーマンスが **SunloConverter** クラスにより改善されました。ただし、Sun Microsystems, Inc の Java Software Division は JDK の今後のリリースで **sun.io** クラスを削除または変更する権利を持っているので、この **SunloConverter** クラスは以降の JDK リリースとは互換性がなくなる場合があります。

SunloConverter クラスを使用するには、jConnect サンプル・アプリケーションをインストールしてください。サンプル・アプリケーションを含む、jConnect とコンポーネントをインストールする方法については、『jConnect for JDBC インストール・ガイド』を参照してください。サンプルをインストールすると、CHARSET_CONVERTER_CLASS 接続プロパティを設定して、jConnect インストール・ディレクトリの *sample* (jConnect 4.x) または *sample2* (jConnect 5.x) サブディレクトリにある **SunloConverter** クラスを参照できます。

"MS932" 指定で拡張文字が使用可能になります。本ドキュメントの [33](#) ページをご参照ください。

動的 SQL の準備文のパフォーマンス・チューニング

Embedded SQL では、動的 SQL とは、静的ではなく実行時にコンパイルされる必要のある SQL 文です。通常、動的 SQL には入力パラメータが含まれていますが、これは必須ではありません。SQL では、**prepare** コマンドを使用して、セッション中に再コンパイルすることなく繰り返し実行できるよう、動的 SQL をプリコンパイルして保存します。

ある SQL がセッション中に何度も使用される場合、それをプリコンパイルすると、使うたびにデータベースに送信してコンパイルするよりもパフォーマンスが改善されます。文が複雑であればあるほど、パフォーマンスの利点は大きくなります。

SQL が数回しか使用されないと思われる場合は、プリコンパイルして保存し、あとでデータベース内で割り付けを解除するのにオーバーヘッドを伴うため、プリコンパイルは効率的ではないことがあります。

実行する動的 SQL 文をプリコンパイルしてメモリ内に保存するには、時間とリソースが要求されます。文がセッション中に何度も使用されるのでないと、データベースに **prepare** を実行するコストの方が、得られる利点を上回ってしまいます。また動的 SQL 文は、いったんデータベースで準備されると、ストアド・プロシージャと同様と考えられます。場合によっては、アプリケーションで準備文を定義するのではなく、ストアド・プロシージャを作成してサーバに常駐させた方が良い場合があります。詳細については、「[準備文かストアド・プロシージャかの選択](#)」(116 ページ)を参照してください。

次のように、jConnect を使用して Sybase データベースで動的 SQL 文のパフォーマンスを最適化できます。

- 文がセッション中に何度か実行されると思われる場合には、プリコンパイルされた文を含む **PreparedStatement** オブジェクトを作成します。
- 文をセッション中に実行する頻度が非常に低い場合は、コンパイルされていない SQL 文を含む **PreparedStatement** オブジェクトを作成します。

以降の項で説明するように、DYNAMIC_PREPARE 接続プロパティを設定して **PreparedStatement** オブジェクトを作成する最適な方法は、アプリケーションが JDBC ドライバを介して移植可能である必要があるかどうか、または JDBC への jConnect 固有の拡張機能を使用できるアプリケーションを作成しているかどうかによって依存します。

jConnect 4.1 以降では、動的 SQL 文に対するパフォーマンス・チューニング機能を提供します。

準備文かストアド・プロシージャかの選択

プリコンパイルされた動的 SQL 文を含む **PreparedStatement** オブジェクトを作成した場合、文は、いったんデータベースでコンパイルされると事実上ストアド・プロシージャとなり、メモリ内に保持されてセッションに関連するデータ構造体に追加されます。データベースにストアド・プロシージャを保持するか、アプリケーションにコンパイルされた SQL 文を含む **PreparedStatement** オブジェクトを作成するか of 決定には、リソース要件およびデータベースとアプリケーションの管理が重要な考慮事項になります。

- ストアド・プロシージャは、コンパイルされると、すべての接続にわたってグローバルに使用できます。これとは対照的に、**PreparedStatement** オブジェクト内の動的 SQL 文は、この文を使用するセッションごとにコンパイルして割り付けを解除する必要があります。
- アプリケーションが複数のデータベースにアクセスする場合、ストアド・プロシージャを使用するという事は、すべてのターゲット・データベースで同じストアド・プロシージャを使用できる必要があるということです。これはデータベース管理上の問題となることがあります。動的 SQL 文に対して **PreparedStatement** オブジェクトを使用すると、この問題を回避できます。
- アプリケーションがストアド・プロシージャを呼び出すための **CallableStatement** オブジェクトを作成すると、SQL コードとテーブルのリファレンスをストアド・プロシージャにカプセル化できます。この場合、アプリケーションを変更することなく、基本のデータベースや SQL コードを修正できます。

移植可能なアプリケーションでの準備文

アプリケーションを複数の異なるベンダからの複数のデータベースで実行するときに、**PreparedStatement** オブジェクトのいくつかにプリコンパイルされた文を含ませ、その他にはコンパイルされていない文を含ませたい場合は、次の手順に従ってください。

- Sybase データベースにアクセスするときに、**DYNAMIC_PREPARE** 接続プロパティが "true" に設定されていることを確認してください。
- プリコンパイルされた文を含む **PreparedStatement** オブジェクトを返すには、通常どおりに **Connection.prepareStatement()** を使用します。

```
PreparedStatement ps_precomp =
    Connection.prepareStatement(sql_string);
```

- コンパイルされていない文を含む **PreparedStatement** オブジェクトを返すには、**Connection.prepareStatement()** を使用します。

Connection.prepareStatement() は **CallableStatement** オブジェクトを返しますが、**CallableStatement** は **PreparedStatement** のサブクラスであるため、**CallableStatement** オブジェクトを **PreparedStatement** オブジェクトにアップキャストすることができます。次に例を示します。

```
PreparedStatement ps_uncomp =
    Connection.prepareStatement(sql_string);
```

プリコンパイルされた文を含む **PreparedStatement** オブジェクトを返すように実装されているのは **Connection.prepareStatement()** だけなので、**PreparedStatement** オブジェクトの *ps_uncomp* は、コンパイルされていない文を含むことが保証されます。

jConnect 拡張機能を持つアプリケーション内の準備文

ドライバ間の移植性に問題がなければ、**SybConnection.prepareStatement()** を使用するコードを作成して、**PreparedStatement** オブジェクトがプリコンパイルされた文を含むか、それともコンパイルされていない文を含むかを指定できます。この場合、どのように準備文をコーディングするかは、アプリケーション内の動的文の大半がセッション中に何度も実行されるのか、それとも数回しか実行されないのかによって異なります。

動的文の大半が数回しか実行されない場合

動的 SQL 文がセッション中で 1、2 度しか実行されないアプリケーションの場合は、次の手順に従ってください。

- **DYNAMIC_PREPARE** 接続プロパティを "false" に設定してください。
- コンパイルされていない文を含む **PreparedStatement** オブジェクトを返すには、通常どおりに **Connection.prepareStatement()** を使用します。

```
PreparedStatement ps_uncomp =
    Connection.prepareStatement(sql_string);
```

- プリコンパイルされた文を含む **PreparedStatement** オブジェクトを返すには、*dynamic* を "true" に設定して **SybConnection.prepareStatement()** を使用します。

```
PreparedStatement ps_precomp =  
    (SybConnection)conn.prepareStatement(sql_string, true);
```

動的文の大半がセッション中に何度も実行される場合

アプリケーション内の動的文がセッション中に何度も実行される場合は、次の手順に従ってください。

- DYNAMIC_PREPARE 接続プロパティを "true" に設定してください。
- プリコンパイルされた文を含む **PreparedStatement** オブジェクトを返すには、通常どおりに **Connection.prepareStatement()** を使用します。

```
PreparedStatement ps_precomp =  
    Connection.prepareStatement(sql_string);
```

- コンパイルされていない文を含む **PreparedStatement** オブジェクトを返すには、**Connection.prepareCall()** ([移植可能なアプリケーションでの準備文](#)の3つめの項目を参照)を使用するか、*dynamic* を "false" に設定して **SybConnection.prepareStatement()** を使用します。

```
PreparedStatement ps_uncomp =  
    (SybConnection)conn.prepareStatement(sql_string, false);
```

```
PreparedStatement ps_uncomp =  
    Connection.prepareCall(sql_string);
```

Connection.prepareStatement()

jConnect は **Connection.prepareStatement()** を実装しているので、**PreparedStatement** オブジェクトにプリコンパイルされた SQL 文とコンパイルされていない SQL 文のどちらを返すようにも設定できます。**PreparedStatement** オブジェクトにプリコンパイルされた SQL 文を返すように **Connection.prepareStatement()** を設定すると、**prepare** コマンドを直接実行したときとまったく同じようにプリコンパイルされて保存される動的 SQL 文がデータベースに送信されます。コンパイルされていない SQL 文を返すように **Connection.prepareStatement()** を設定すると、文はデータベースに送信されずに **PreparedStatement** オブジェクトに返されます。

Connection.prepareStatement() が返す SQL 文のタイプは接続プロパティ **DYNAMIC_PREPARE** によって判別され、そのセッション全体に適用されます。

Sybase 固有のアプリケーションの場合、jConnect 5.0 では jConnect **SybConnection** クラスの下に **prepareStatement()** メソッドを提供しています。**SybConnection.prepareStatement()** を使用すると、**DYNAMIC_PREPARE** 接続プロパティのセッション・レベル設定に関係なく、個々の動的 SQL 文がプリコンパイルされるかどうかを指定できます。

DYNAMIC_PREPARE 接続プロパティ

DYNAMIC_PREPARE は、動的 SQL 準備文を有効にするためのブール値の接続プロパティです。

- **DYNAMIC_PREPARE** が "true" に設定されていると、セッション中の **Connection.prepareStatement()** の呼び出しはどれも **PreparedStatement** オブジェクトにプリコンパイルされた文を返そうとします。

この場合、**PreparedStatement** が実行されると、これに含まれる文はすでにデータベースでプリコンパイルされて動的に割り当てられた値に対するプレースホルダを持っているので、文は実行されるだけです。

- **DYNAMIC_PREPARE** が接続に対して "false" に設定されていると、**Connection.prepareStatement()** によって返される **PreparedStatement** オブジェクトにはプリコンパイルされた文は含まれません。

この場合、**PreparedStatement** が実行されるたびに、これに含まれる SQL 文はデータベースに送信され、コンパイルされて実行される必要があります。

DYNAMIC_PREPARE のデフォルト値は "false" です。

次の例では、動的 SQL 文のプリコンパイルを有効にするよう、DYNAMIC_PREPARE が "true" に設定されています。この例では、**props** は接続プロパティを指定するための **Properties** オブジェクトです。

```
...
props.put("DYNAMIC_PREPARE", "true")
Connection conn = DriverManager.getConnection(url, props);
```

DYNAMIC_PREPARE が "true" に設定されているときは、次のことに注意してください。

- すべての動的文を **prepare** コマンド下でプリコンパイルできるわけではありません。SQL-92 標準では、**prepare** コマンドで使用できる文にいくつかの制約を設けています。また個々のデータベース・ベンダが独自の制約を設けている場合もあります。
- データベースが、**Connection.prepareStatement()** を介して送信された文をプリコンパイルして保存することができないためにエラーを生成する場合、**jdbc** はエラーをトラップして、コンパイルされていない SQL 文を含む **PreparedStatement** オブジェクトを返します。**PreparedStatement** オブジェクトが実行されるたびに、文はデータベースに再送信され、コンパイルされて実行されます。
- プリコンパイルされた文はデータベースのメモリ内に常駐して、セッションの終わりか、または **PreparedStatement** オブジェクトが明示的に閉じられるまで存続します。**PreparedStatement** オブジェクトでのガーベジ・コレクションは、データベースから準備文を削除しません。

セッション中にサーバ・メモリに準備文が累積してパフォーマンスを低下させるのを回避するには、一般的には、**PreparedStatement** オブジェクトを使用したあとでひとつひとつ明示的に閉じます。

SybConnection.prepareStatement()

アプリケーションが JDBC への jConnect 固有の拡張機能を使用できる場合は、**SybConnection.prepareStatement()** 拡張メソッドを使用して **PreparedStatement** オブジェクトに動的 SQL 文を返すことができます。

```
PreparedStatement SybConnection.prepareStatement(String sql_stmt,  
boolean dynamic) throws SQLException
```

SybConnection.prepareStatement() は、プリコンパイルされた SQL 文かコンパイルされていない SQL 文を含む **PreparedStatement** オブジェクトを返します。どちらを返すかは *dynamic* パラメータの設定によって異なります。*dynamic* が "true" に設定されている場合、**SybConnection.prepareStatement()** はプリコンパイルされた SQL 文を含む **PreparedStatement** オブジェクトを返します。*dynamic* が "false" に設定されている場合は、コンパイルされていない SQL 文を含む **PreparedStatement** オブジェクトを返します。

次の例では、

SybConnection.prepareStatement() を使用して、プリコンパイルされた文を含む **PreparedStatement** オブジェクトを返します。

```
PreparedStatement precomp_stmt =  
((SybConnection) conn).prepareStatement( "SELECT * FROM  
authors WHERE au_fname LIKE ?", true);
```

この例では、**SybConnection.prepareStatement()** の使用を可能にするために、接続オブジェクト *conn* が **SybConnection** オブジェクトにダウンキャストされています。**SybConnection.prepareStatement()** に渡された SQL 文字列は、DYNAMIC_PREPARE 接続プロパティが "false" に設定されていてもデータベースでプリコンパイルされることに注意してください。

データベースが、**SybConnection.prepareStatement()** を介して送信された文をプリコンパイルして保存することができないためにエラーを生成する場合、jConnect は **SQLException** を返し、呼び出しは **PreparedStatement** オブジェクトを返すことができません。これは、エラーの場合に SQL エラーをトラップして、コンパイルされていない文を含む **PreparedStatement** オブジェクトを返す **Connection.prepareStatement()** とは異なります。

カーソルのパフォーマンス

SybCursorResultSet クラスで **Statement.setCursorName()** メソッドまたは **setFetchSize()** メソッドを使用すると、**jConnect** はデータベース内にカーソルを作成します。ほかのメソッドの場合は、**jConnect** はカーソルをオープンし、フェッチして、更新します。

4.0 より前のバージョンの **jConnect** では、明示的にカーソル・コマンドを使用して、解析とコンパイルを行うために **SQL** 文をデータベースに送信することによってだけ、カーソルを作成して操作できます。

jConnect バージョン 4.0 以降は、**SQL** 文をデータベースに送信するか、カーソル・コマンドを **TDS (Tabular Data Stream)** 通信プロトコル内のトークンとしてコード化することによって、カーソルを作成し操作します。最初のタイプのカーソルは「言語カーソル」であり、2 番目のタイプのカーソルは「プロトコル・カーソル」です。

プロトコル・カーソルは、言語カーソルよりもすぐれたパフォーマンスを提供します。さらに、必ずしもすべてのデータベースが言語カーソルをサポートしているわけではありません。たとえば、**Adaptive Server Anywhere** データベースは言語カーソルをサポートしていません。

jConnect では、デフォルト条件はすべてのカーソルについてプロトコル・カーソルになります。ただし、**LANGUAGE_CURSOR** 接続プロパティではデータベース内で言語コマンドを使用してカーソルを作成し操作できるオプションがあります。

LANGUAGE_CURSOR 接続プロパティ

LANGUAGE_CURSOR は、カーソルがプロトコル・カーソルと言語カーソルのどちらとして作成されるかを判別できるようにするための、**jConnect** のブール値の接続プロパティです。

- **LANGUAGE_CURSOR** が "false" に設定されている場合、セッション中に作成されたカーソルはすべてプロトコル・カーソルとなり、より優れたパフォーマンスを提供します。**jConnect** はカーソル・コマンドを **TDS** プロトコル内のトークンとして送信することによって、カーソルを作成、操作します。

デフォルトでは、**LANGUAGE_CURSOR** は "false" に設定されています。

- `LANGUAGE_CURSOR` が "true" に設定されている場合、セッション中に作成されたカーソルはすべて言語カーソルとなります。`jConnect` は SQL 文を解析、コンパイルするためにデータベースに送信することによって、カーソルを作成、操作します。

`LANGUAGE_CURSOR` を "true" に設定することによる明確な利点はありませんが、`LANGUAGE_CURSOR` を "false" に設定したときにアプリケーションが予期しない動作をした場合に備えて、このオプションが提供されています。

第 5 章

jConnect アプリケーションへのマイグレート

この章では、Sybase 拡張機能を使用するアプリケーションを jConnect バージョン 4.0 以前から jConnect バージョン 4.1 以降にマイグレート (移行) する方法について説明します。

次の項目について説明します。

項名	ページ
jConnect 4.1 へのアプリケーションのマイグレート	126
jConnect 5.x へのアプリケーションのマイグレート	126
jConnect 4.2 および 5.2 へのアプリケーションのマイグレート	126
Sybase 拡張機能の変更	129

jConnect アプリケーションへのマイグレート

jConnect 4.1 へのアプリケーションのマイグレート

jConnect 4.1 は、jConnect の以前のバージョンとの下位互換性があります。既存のアプリケーションは、再コンパイルしなくても、そのまま使用できます。

Sybase 拡張機能を使用して新しいアプリケーションを開発する場合、**com.sybase.jdbcx** にあるインタフェースを使用してください。

この共通インタフェースによって、最少の変更でアプリケーションを jConnect 4.1 以降にアップグレードできます。Sybase 拡張機能の変更については、「[Sybase 拡張機能の変更](#)」(129 ページ) を参照してください。

jConnect 5.x へのアプリケーションのマイグレート

Sybase jConnect 5.x ドライバ用のパッケージとクラスは、*com.sybase.jdbc2.jdbc.SybDriver* です。これは、jConnect の以前のバージョンとは異なります。アプリケーションに対しては、SybDriver をロードするソース・コードの変更と、Java 2 プラットフォームによる再コンパイルが必要です。

jConnect 5.0 でドライバをロードする例を示します。

```
Driver d =  
(Driver)Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();  
DriverManager.registerDriver(d);
```

アプリケーションで JDBC API への Sybase 拡張機能を使用している場合、インポートを **com.sybase.jdbc** か **com.sybase.utils** またはその両方の使用から **com.sybase.jdbcx** に変更してください。Sybase の拡張機能の変更点については、「[Sybase 拡張機能の変更](#)」(129 ページ) を参照してください。

Sybase の拡張機能の使用法の例については、jConnect で提供されるサンプルを参照してください。

jConnect 4.2 および 5.2 へのアプリケーションのマイグレート

jConnect を以前のバージョンから 4.2 または 5.2 にアップグレードする場合は、次の表に示すアップグレード・パスに従って、ソース・コードを変更および再コンパイルします。

記号：

A `com.sybase.jdbcx` パッケージに変更する（推奨）

B 新しいインストール構造に応じて `CLASSPATH` を変更する

C 新しい jConnect 5.x ドライバを使用するように再コンパイルする

詳細については、この後の説明を参照してください。

アップグレード元	アップグレード先			
jConnect のバージョン	4.1	4.2	5.0	5.2
4.0 およびそれ以前のバージョン	A	AB	AC	ABC
4.1	-	AB	AC	ABC
4.2	-	-	サポート外のパス	AC
5.0	-	-	-	B

❖ A. 新しい Sybase 拡張機能を使用する

1 パッケージのインポートを次のように変更します。

```
import com.sybase.jdbc.*
```

上記を下記に変更します。

```
import com.sybase.jdbcx.*;
```

2 新しい Sybase 拡張機能 API を使用します。詳細については、「[Sybase 拡張機能の変更](#)」(129 ページ) を参照してください。

❖ B. 新しいインストール構造に応じて `CLASSPATH` を変更する

インストールした jConnect ドライバの上位ディレクトリを `JDBC_HOME` の値として指定します。

次に例を示します。

- jConnect 4.2 の場合

```
JDBC_HOME=jConnect-4_2
```

- jConnect 5.0 の場合

```
JDBC_HOME=<jConnect installation directory>
```

JDBC_HOME の設定方法の詳細については、『jConnect for JDBC インストール・ガイド』の第 1 章の「環境変数の設定」を参照してください。

バージョンの変更		CLASSPATH の設定値
変更前	4.1	<i>JDBC_HOME/classes</i>
変更後	5.2	<i>JDBC_HOME/jconn2.jar</i>
変更前	4.1	<i>JDBC_HOME/classes</i>
変更後	4.2	<i>JDBC_HOME/classes</i>
変更前	5.0	<i>JDBC_HOME/classes/jconn2.jar</i>
変更後	5.2	<i>JDBC_HOME/classes/jconn2.jar</i>

❖ **C. 新しい jConnect 5.x ドライバを使用するように再コンパイルする**

- ドライバをロードするソース・コードを次のように変更します。

```
Class.forName("com.sybase.jdbc.SybDriver");
```

上記を下記に変更します。

```
Class.forName("com.sybase.jdbc2.jdbc.SybDriver");
```

Sybase 拡張機能の変更

com.sybase.jdbcx は、jConnect バージョン 4.1、4.2、および 5.x に新たに追加されたパッケージです。このパッケージには、JDBC への Sybase 拡張機能がすべて含まれています。jConnect の以前のバージョンでは、これらの拡張機能は **com.sybase.jdbc** と **com.sybase.utils** パッケージで使用できました。

com.sybase.jdbcx は、jConnect のバージョンの違いにかかわらず一貫性のあるインタフェースを提供します。Sybase のすべて拡張機能は、Java インタフェースとして定義されているので、これらのインタフェースを使用して構築されたアプリケーションに何も影響を与えずに基本となる実装が変更できます。

Sybase 拡張機能を使用する新しいアプリケーションを開発する場合には、**com.sybase.jdbcx** を使用してください。このパッケージのインタフェースでは、アプリケーションに必要な最小限の変更を加えるだけで、バージョン 4.0 より新しいバージョンの jConnect にアップグレードできます。

注意 アプリケーション **com.sybase.jdbc** および **com.sybase.utils** で使用可能な、JDBC API への Sybase 拡張機能を使用して以前に構築されたアプリケーションは、jConnect 4.x でも引き続き機能しますが、**com.sybase.jdbc** および **com.sybase.utils** 内のすべての Sybase 拡張機能は推奨ではないことがマーク付けされます。

Sybase 拡張機能の一部は、新しい **com.sybase.jdbcx** インタフェースを取り入れるために変更されました。

変更の例：

たとえば、アプリケーションで **SybMessageHandler** を使用する場合のコード変更は次のとおりです。

- jConnect 4.0 のコード：

```
import com.sybase.jdbc.SybConnection;
import com.sybase.jdbc.SybMessageHandler;
.
.
Connection con = DriverManager.getConnection(url, props);
SybConnection sybCon = (SybConnection) con;
sybCon.setMessageHandler(new ConnectionMsgHandler());
```

- **jConnect 4.1** 以降のコード:

```
import com.sybase.jdbcx.SybConnection;
import com.sybase.jdbcx.SybMessageHandler;
.
.
Connection con = DriverManager.getConnection(url, props);
SybConnection sybCon = (SybConnection) con;
sybCon.setSybMessageHandler(new ConnectionMsgHandler());
```

Sybase の拡張機能の使用方法的例については、**jConnect** で提供されるサンプルを参照してください。

変更されたメソッド名

次の表は、新しいインタフェースで変更されたメソッド名を示します。

クラス	古い名前	新しい名前
SybConnection	getCapture()	createCapture()
SybConnection	setMessageHandler()	setSybMessageHandler()
SybConnection	getMessageHandler()	getSybMessageHandler()
SybStatement	setMessageHandler()	setSybMessageHandler()
SybStatement	getMessageHandler()	getSybMessageHandler()

Debug クラス

Debug クラスへの直接の静的参照はサポートされなくなりました。しかし、**com.sybase.utils** パッケージには推奨しない形ですがまだ残っています。**jConnect** のデバッグ機能を使用するには、**SybDriver** クラスの **getDebug()** メソッドを使用して **Debug** クラスを参照してください。次に例を示します。

```
import com.sybase.jdbcx.SybDriver;
import com.sybase.jdbcx.Debug;
.
.
.
SybDriver sybDriver =
    SybDriver.Class.forName
        ("com.sybase.jdbc2.jdbc.SybDriver") newInstance();
```

```
Debug sybDebug = sybDriver.getDebug();  
sybDebug.debug(true, "ALL", System.out);
```

Sybase 拡張機能のリストは、jConnect インストール・ディレクトリの *docs/* ディレクトリにある jConnect javadoc のマニュアルで説明しています。

第 6 章

Web サーバ・ゲートウェイ

この章では Web サーバ・ゲートウェイについて説明し、さらに jConnect での使い方について説明します。

この章では、次の項目について説明します。

項名	ページ
Web サーバ・ゲートウェイの概要	134
カスケード・ゲートウェイの使用	137
TDS トンネリング・サブレットの使用方法	143

Web サーバ・ゲートウェイの概要

データベース・サーバが、Web サーバ以外のホストで稼働している場合、またはファイアウォールを通してセキュア・データベース・サーバに接続する必要があるインターネット・アプリケーションを開発している場合には、データベース・サーバへのパスを提供し、プロキシとして動作するゲートウェイが必要になります。

サーバに Secure Sockets Layer (SSL) プロトコルを使用して接続するために、jConnect は **javax.servlet** インタフェースをサポートするどの Web サーバにもインストールできる Java サブレットを提供します。このサブレットにより、jConnect は Web サーバをゲートウェイとして使用した暗号化のサポートが可能になります。

注意 jConnect は、クライアント・システムでの SSL のサポートを含みます。jConnect のクライアント側の SSL サポートの詳細については、「[カスタム・ソケット・プラグインの実装](#)」(26 ページ) を参照してください。

jConnect には、カスケード Web サーバを若干修正したバージョンの、カスケード・ゲートウェイが含まれています。カスケード・ゲートウェイは CONNECT HTTP メソッドをサポートします。これによって Tabular Data Stream (TDS) データの HTTP トンネリング機能を使用できます。カスケード・ゲートウェイを使用すると、Web サーバ以外のホストで稼働する、ファイアウォールを経由して渡す必要のないデータベース・サーバに接続できます。詳細については、「[カスケード・ゲートウェイの使用](#)」(137 ページ) を参照してください。

TDS トンネリング

jConnect は、TDS を使用して、データベース・サーバと通信します。HTTP-tunnelled TDS は、要求を転送する場合に便利です。ゲートウェイを通過するクライアントからバックエンド・サーバへの要求は、要求の本体に TDS を含みます。要求ヘッダは、要求パケットに含まれる TDS の長さを示します。

TDS は、HTTP とは異なり接続指向型プロトコルです。インターネット・アプリケーションでの暗号化のようなセキュリティ機能をサポートするために、jConnect は TDS トンネリング・サブレットを使用し、HTTP 要求間での論理コネクションを管理します。サブレットは、最初のログイン要求の間にセッション ID を生成します。セッション ID は、次の要求のヘッダに含まれます。セッション ID を使用することによって、アクティブなセッションを識別できます。また、サブレットが特定のセッション ID を使用してオープンしている接続を持っている限り、セッションを再開できます。

TDS トンネリング・サブレットが提供する論理コネクションによって、jConnect は 2 つのシステムの間での暗号化された通信をサポートできるようになります。たとえば、TDS トンネリング・サブレットを実行している Web サーバに接続する、CONNECT_PROTOCOL 接続プロパティを "https" に設定した jConnect クライアントなどです。

jConnect およびゲートウェイの設定

Web サーバと Adaptive Server の設定にはいくつかのオプションがあります。一般的な設定を次に示します。これらの例は、jConnect のドライバがどこにインストールされるか、カスケード・ゲートウェイまたは TDS トンネリング・サブレットを使用したゲートウェイがいつ使用されるかを示します。

単一ホスト上の Web サーバと Adaptive Server

この 2 層の設定では、Web サーバと Adaptive Server は両方とも同じホストにインストールされます。

- Web サーバ・ホストでの jConnect のインストール
- ゲートウェイは必要なし

単一ホスト上の専用 JDBC Web サーバと Adaptive Server

この設定では、メインの Web サーバとは別のホストを使用します。2 番目のホストは、Web サーバによって、特に Adaptive Server アクセスと Adaptive Server によって、共有されます。メイン・サーバからのリンクは、専用 Web サーバへの SQL アクセスを必要とする要求にダイレクトされます。

- 2 番目の (Adaptive Server) ホストへの jConnect のインストール

- ゲートウェイは必要なし

別のホストでの Web サーバと Adaptive Server

この 3 層の設定では、Adaptive Server は、Web サーバとは別のホスト上にあります。jConnect は、Adaptive Server のプロキシとして機能するにはゲートウェイが必要です。

- Web サーバ・ホストでの jConnect のインストール
- TDS トンネリング・サブレットまたはカスケード・ゲートウェイが必要

ファイアウォールを介したサーバへの接続

ファイアウォールで保護されているサーバに接続するには、データベース要求の応答をインターネット上で転送することをサポートする、TDS トンネリング・サブレットを持った Web サーバを使用する必要があります。

- Web サーバ・ホストでの jConnect のインストール
- `javax.servlet` インタフェースをサポートする Web サーバが必要

カスケード・ゲートウェイの使用

データベース・サーバが、Web サーバではなく別のホストで稼働している場合、ファイアウォールがない場合でも、プロキシとして機能するには、データベース・サーバへのパスを提供するゲートウェイが必要です。

jConnect には、カスケード・ゲートウェイが含まれています。これは、Java で記述された Cascade Web サーバを David Wilkerson が少し修正したバージョンです (email: davidw@cascade.org.uk、Web サイト: <http://www.cascade.org.uk/>)。

カスケード・ゲートウェイは、パケットを受信して転送します。また、要求を Adaptive Server に送信する場合は HTTP から TDS、結果を返すときは TDS から HTTP に切り替えます。

注意 カスケード・ゲートウェイは、暗号化をサポートしません。そのため、ファイアウォールを経由してバックエンド・サーバに接続するインターネット・アプリケーションには適しません。

使用上の条件

- jConnect をデフォルトのインストール・ディレクトリにインストールしなかった場合は、*www.dos* (DOS) または *www.template* (UNIX) を編集して、jConnect をインストールしたリテラル・パスを指すよう、すべてのデフォルト・インストール・ディレクトリ参照部分を変更してください。
- カスケード・ゲートウェイと Web サーバは同じホストで稼働する必要があります。こうすると、アプレットは Web サーバと同じホストに接続しますが、カスケード・ゲートウェイが制御するポートに接続します。ゲートウェイは適切なデータベースに要求をルート指定します。これがどのように行われるかを調べるには、jConnect インストール・ディレクトリの *sample* (jConnect 4.x) または *sample2* (jConnect 5.x) サブディレクトリ内の *Isql.java* と *gateway.html* のコードを確認してください。"proxy" を検索します。

データベースが Web サーバと同じホストで稼働している場合は、カスケード・ゲートウェイは必要ありません。

カスケード・ゲートウェイのインストール

カスケード・ゲートウェイは、jConnect インストーラまたは *install.bat* か *install.sh* ファイルのいずれかを使用して jConnect をフル・インストールするとインストールされます。jConnect インストーラを使用して、必要に応じてカスケード・ゲートウェイだけをインストールすることもできます。『jConnect for JDBC インストール・ガイド』を参照してください。

カスケード・ゲートウェイの起動方法

カスケード・ゲートウェイをテストするには、次に示すプラットフォーム固有の指示に従ってください。

Windows NT および Windows 95

- 1 DOS プロンプトで jConnect インストール・ディレクトリに移動する。

JDBC_HOME 環境変数をこのディレクトリに設定してください。

- 2 次のように入力して、カスケード・ゲートウェイを開始します。

```
httpd
```

コマンドが成功すると、*httpd.bat* からの出力が表示されます。これは次の文字列で終了しています。

```
HTTPDServer www.dos
```

UNIX

jConnect をインストールしたディレクトリ (JDBC_HOME ディレクトリ) に移動します。次のコマンドを入力します。

```
sh httpd.sh &
```

トラブルシューティング

- **httpd** コマンドを入力してもメッセージが表示されない場合は、サーバが稼働していません。冗長モードでコマンドをリトライしてください。

Windows の場合、DOS プロンプトで次のように入力します。

```
httpd -Dverbose=1 > filename
```

UNIX の場合、次のように入力します。

```
sh httpd.sh -Dverbose=1 > filename &
```

これらのコマンドでは、*filename* はデバッグ・メッセージの出力ファイルです。

- 次のエラー・メッセージが表示された場合：

```
HTTPServer: IOException: getRequest() Address  
already in use
```

これは JDBC_HOME にある *www.dos* (DOS) または *www.template* (UNIX) ファイルに指定されているポートで、別のプロセスが稼働していることを意味します。このエラーはゲートウェイを起動したときに発生します。

次の処理ができます。

- 指定されているポートで現在稼働しているプロセスを停止します。プロセスが終了したことを確認してから、ゲートウェイを起動します。

または

- *www.dos* または *www.template* ファイル内のポート番号を変更してから、JDBC_HOME の *sample* (jConnect 4) または *sample2* (jConnect 5) サブディレクトリにある *gateway.html* ファイルを変更します。これは *proxy* パラメータを "*localhost:new_port*" に変更して行います。

ホストが "*localhost*" ではない場合 (Cascade HTTP サーバとブラウザが異なるホスト上にある場合) は、*proxy* パラメータに "*localhost*" ではなくリモート・ホスト名を使用してください。

カスケード・ゲートウェイのテスト

設定を確認し、カスケード・ゲートウェイをテストするには、Sybase デモ・データベースに接続する検証プログラムを実行します。

注意 検証プログラムは "*localhost:8000*" を使用してゲートウェイをテストします。

Windows NT か Windows 95 の DOS プロンプトから、または UNIX プロンプトで、JDBC_HOME ディレクトリに移動します。

jConnect 4.x の場合は、次のように入力します。

```
java sample.SybSample Validate
```

jConnect 5.x の場合は、次のように入力します。

```
java sample2.SybSample Validate
```

検証が成功すると、jConnect バージョン番号と「Connected successfully」というメッセージがサンプル出力ウィンドウに表示されます。

トラブルシューティング

「Bad command or file name」エラー (Windows 95) または「Name specified is not recognized as an internal or external command」エラー (Windows NT) が表示された場合は、JDK ホーム・ディレクトリの `%bin` サブディレクトリがパスに含まれているか確認してください。

index.html ファイルの読み込み

Web ブラウザを使用して jConnect インストール・ディレクトリにある *index.html* ファイルを参照してください。*index.html* は jConnect のマニュアルとサンプル・コードへのリンクを提供しています。

注意 jConnect をインストールしたマシンで Netscape を使用する場合は、ブラウザが CLASSPATH 環境変数へのアクセスを持っていないことを確認してください。『jConnect for JDBC インストール・ガイド』の第 1 章の「Netscape を使用する場合の CLASSPATH 設定に関する制限」を参照してください。

- 1 Web ブラウザを開きます。
- 2 設定にあった URL を入力します。たとえば、ブラウザとカスケード・ゲートウェイが同じホスト上で稼働している場合は、次のように入力します。

```
http://localhost:8000/index.html
```

ブラウザとカスケード・ゲートウェイが異なるホスト上で稼働している場合は、次のように入力します。

`http://host:port/index.html`

host はカスケード・ゲートウェイが稼働しているホストの名前で、*port* は受信ポートです。

トラブルシューティング

正しいホスト、ポートおよびファイル情報を入力したがブラウザがこのリンクを開くことができない場合は、カスケード・ゲートウェイが稼働していません。詳細については、「[カスケード・ゲートウェイの起動方法](#)」(138 ページ) を参照してください。

サンプル Isql アプレットの実行

index.html ファイルをブラウザにロードしたら、次を実行してください。

- 1 [Run Sample JDBC Applets] をクリックします。
これによって jConnect Sample Programs ページに移動します。
- 2 Sample Programs ページの下の方へ移動して、"Executable Samples" の下の表を探します。
- 3 表の中の "Isql.java" を見つけて、ローの終わりで [Run] をクリックします。

サンプル **Isql.java** アプレットがサンプル・データベース上で簡単なクエリを入力するよう要求し、結果を表示します。このアプレットはデフォルトの Adaptive Server ホスト名、ポート番号、ユーザ名 (*guest*)、パスワード (*sybase*)、データベース、およびクエリを表示します。アプレットはデフォルト値を使用して Sybsae デモ・データベースに接続します。[Go] をクリックすると結果が返されます。

トラブルシューティング

UNIX では、アプレットが予期したとおりに表示されない場合、アプレットの画面の大きさを変更することができます。

- 1 テキスト・エディタを使用して次の編集を行います。

jConnect 4.x の場合：

`$JDBC_HOME/sample/gateway.html`

jConnect 5.x の場合：

\$JDBC_HOME/sample2/gateway.html

- 2 7行目の高さを指定するパラメータを 650 に変更します。違う高さの設定を試すことができます。
- 3 ブラウザの Web ページを再ロードします。

カスケード・ゲートウェイへの接続の定義

カスケード・ゲートウェイを使用するアプリケーション内の接続を定義するには、カスケード・ゲートウェイが稼働しているホストの名前を URL に入力します。

host:port

host はカスケード・ゲートウェイが稼働しているホストの名前で、*port* は受信ポートです。

TDS トンネリング・サブレットの使用法

TDS トンネリング・サブレットを使用するためには、Sun Microsystems の Java Web サーバなどの **javax.servlet** インタフェースをサポートする Web サーバが必要です。Web サーバをインストールする場合、jConnect TDS トンネリング・サブレットをアクティブ・サブレットのリストに含めてください。サブレットのパラメータを設定して、接続タイムアウトと最大パケット・サイズを定義できます。

トンネリング・サブレットを使用した場合、ゲートウェイを通過するクライアントからバックエンド・サーバへの要求は、GET または POST コマンド、TDS セッション ID (最初の要求のあと)、バックエンド・アドレス、要求のステータスを含んでいます。

TDS は、要求の本体内で接続されています。ヘッダ・フィールド 2 つは、TDS ストリームの長さ、およびゲートウェイによって割り当てられたセッション ID を示します。

クライアントが要求を送信する場合、Content-Length ヘッダ・フィールドは TDS の内容のサイズ、および要求コマンドが POST であることを示します。クライアントがサーバからの応答データの次の部分を検索しているか、または接続をクローズしているために、要求内に TDS データがない場合、要求コマンドは GET です。

次の例は、TDS-tunneled HTTPS プロトコルを使用して、クライアントと HTTPS ゲートウェイの間をどのように情報が渡されるかを示します。この例では、ポート番号 "1234" の DBSERVER という名前のバックエンド・サーバへの接続を示します。

表 6-1: クライアントからゲートウェイに対するログイン要求。セッション ID なし。

クエリ	POST/tds?ServerHost=dbserver&ServerPort=1234&Operation=more HTTP/1.0
ヘッダ	Content-Length:605
内容 (TDS)	ログイン要求

表 6-2: ゲートウェイからクライアントへの場合。ヘッダには TDS サーブレットによって割り当てられたセッション ID が含まれる。

クエリ	200 SUCCESS HTTP/1.0
ヘッダ	Content-Length:210 TDS-Session:TDS00245817298274292
内容 (TDS)	ログイン受信確認 EED

表 6-3: クライアントからゲートウェイへの場合。後続のすべての要求のヘッダには、セッション ID が含まれる。

クエリ	POST/tds?TDS-Session=TDS00245817298274292&Operation=more HTTP/1.0
ヘッダ	Content-Length:32
内容 (TDS)	クエリ "SELECT * from authors"

表 6-4: ゲートウェイからクライアントへの場合。後続のすべての応答のヘッダには、セッション ID が含まれる。

クエリ	200 SUCCESS HTTP/1.0
ヘッダ	Content-Length:2048 TDS-Session:TDS00245817298274292
コンテンツ (TDS)	ローのフォーマットおよびクエリ応答からのいくつかのロー

TDS トンネリング・サーブレットのシステム稼働条件

TDS-tunneled HTTP に jConnect サーブレットを使用するには、次のものがが必要です。

- **javax.servlet** インタフェースをサポートする Web サーバ。サーバをインストールするには、サーバが提供する指示に従ってください。
- JDK 1.1 をサポートする Web サーバ。Netscape 4.0、Internet Explorer 4.0、HotJava など。

サーブレットのインストール

jConnect インストール環境では、*classes* ディレクトリの下に *gateway* サブディレクトリ (jConnect 4.x) または *gateway2* サブディレクトリ (jConnect 5.x) が含まれています。このサブディレクトリには TDS トンネリング・サーブレットに必要なファイルが入っています。

jConnect **gateway** パッケージを、Web サーバのサーブレット・ディレクトリの下に *gateway* サブディレクトリ (jConnect 4.x) または *gateway2* サブディレクトリ (jConnect 5.x) にコピーしてください。サーブレットをコピーしたら、Web サーバの指示に従ってアクティブにします。

サーブレットの引数の設定

サーブレットを Web サーバに追加すると、オプションの引数を指定してパフォーマンスをカスタマイズできます。

- *SkipDoneProc* [true/false] - Sybase データベースは、クエリの実行中に中間処理手順が行われているときにロー・カウント情報を返すことがよくあります。通常、クライアント・アプリケーションはこのデータを無視します。*SkipDoneProc* を "true" に設定すると、サーブレットはこの余計な情報を応答から直ちに切り除きます。これによってクライアントのネットワーク使用率と処理要件を軽減します。これは、不要なデータを無視する前に暗号化したり暗号化を解除したりしないため、HTTPS/SSL を使用しているときは特に便利です。
- *TdsResponseSize* - Tunneled HTTPS の最大 TDS パケット・サイズを設定します。データ量の多い、少数のユーザの場合は、*TdsResponseSize* を大きな値に指定すると効果的です。小さなトランザクションを行うユーザが多数いる場合は、*TdsResponseSize* の値を小さくします。
- *TdsSessionIdleTimeout* - サーバ接続が自動的に閉じられるまでに接続がアイドル状態で保持される時間をミリ秒単位で定義します。デフォルトの *TdsSessionIdleTimeout* は 600,000 (10 分) です。
長い時間アイドル状態になる可能性のある対話型クライアント・プログラムがある場合、接続を切断したくないときは、*TdsSessionIdleTimeout* の値を大きくします。

SESSION_TIMEOUT 接続プロパティを使用して、jConnect クライアントからの接続タイムアウト値を設定することもできます。これは、長い時間アイドル状態になる可能性のあるアプリケーションが具体的にはない場合に便利です。この場合は、サーブレットに設定するのではなく、SESSION_TIMEOUT 接続プロパティを使用して接続に対して長いタイムアウトを設定してください。

- *Debug* - デバッグをオンにします。「[jConnect でのデバッグ](#)」(98 ページ)を参照してください。

サーブレット引数は、カンマで区切った文字列で入力してください。次に例を示します。

```
TdsResponseSize=[size],TdsSessionIdleTimeout=[timeout],Debug=true
```

サーブレット引数を入力する方法については、Web サーバのマニュアルを参照してください。

サーブレットの呼び出し

jConnect は TDS-tunnelling サーブレットが *proxy* 接続プロパティにのパス拡張子に基づいてインストールされているゲートウェイをいつ使用するかを決定します。jConnect は *proxy* へのサーブレットのパス拡張子を認識して、指定のゲートウェイでサーブレットを呼び出します。

次のフォーマットを使用して接続 URL を定義してください。

```
http://host:port/TDS-servlet-path
```

jConnect は Web サーバ上で TDS トンネリング・サーブレットを呼び出して、HTTP を介して TDS をトンネリングします。サーブレットのパスは、Web サーバのサーブレット・エイリアス・リストにユーザが定義したものでなければなりません。

アクティブな TDS セッションのトラッキング

各セッションに対するサーバ接続を含む、アクティブな TDS セッションに関する情報を参照できます。使用している Web ブラウザで管理 URL を開きます。

```
http://host:port/TDS-servlet-path?Operation=list
```

たとえば、使用しているサーバが MYSERVER で TDS サーブレットのパスが */tds* の場合は、次のように入力します。

```
http://myserver:8080/tds?Operation=list
```

これによってアクティブな TDS セッションのリストが表示されます。セッションをクリックして、サーバ接続などの詳細を見ることができます。

TDS セッションの終了

前述の URL を使用して、アクティブな TDS セッションを終了することができます。最初のページにあるセッションのリストからアクティブなセッションをクリックし、[Terminate This Session] をクリックします。

TDS セッションの再開

開いている既存の接続を必要に応じて再開できるよう、SESSION_ID 接続プロパティを設定できます。SESSION_ID を指定すると、jConnect はプロトコルのログイン・フェーズをスキップし、指定されたセッション ID を使用するゲートウェイとの接続を再開します。指定したセッション ID がサーブレットに存在しない場合、ユーザがその接続を使用しようとする、jConnect は SQL 例外を返します。

TDS トンネリングと Netscape Enterprise Server 3.5.1 on Solaris

Netscape Enterprise Server 3.5.1 は、
javax.servlet.ServletConfig.getInitParameters() と
javax.servlet.ServletConfig.getInitParameterNames() メソッドをサポートしていません。必要なパラメータ値を提供するには、
getInitParameter() および **getInitParameterNames()** への呼び出しを *TDSTunnelServlet.java* 内のハードコードされたパラメータ値で置き換える必要があります。

TDSTunnelServlet.java 内の必須パラメータを入力し、Netscape Enterprise Server 3.5.1 on Solaris で TDS トンネリングを使用するには、次の手順に従ってください。

- 1 *TDSTunnelServlet.java* のパラメータ値をハードコードします。
- 2 *TDSTunnelServlet.java* 内のクラス宣言から .class ファイルを作成します。

これによって次のファイルが作成されます。

- *TDSTunnelServlet.class*
- *TdsSession.class*
- *TdsSessionManager.class*

- 3 次に示すように、Netscape Enterprise Server 3.5.1 (NSE_3.5.1) インストール・ディレクトリの下に *.class* ファイルを作成します。

```
mkdir NSE_3.5.1_install_dir/plugins/java/servlets/gateway
```

- 4 *TDSTunnelServlet.java* から派生した *.class* ファイルを、今作成したディレクトリにコピーします。
- 5 *\$JDBC_HOME/classes/com/sybase* の下にあるクラスを *NSE_3.5.1_install_dir/docs/com/sybase* にコピーします。

次に示すように、*\$JDBC_HOME/classes* の下層にあるすべてのファイルを *NSE_3.5.1_install_dir/docs* に再帰的にコピーすると、これを簡単に行うことができます。

```
cp -r $JDBC_HOME/classes NSE_3.5.1_install_dir/docs
```

これによって *\$JDBC_HOME/classes/com/sybase* の下でない多数のファイルやディレクトリがコピーされます。余分なファイルやディレクトリによる悪影響はありませんが、ディスク領域が消費されます。これらを削除してディスク領域を再利用することができます。

- 6 *proxy URL* を TDS トンネリング・サーブレットに設定します。
たとえば、*\$JDBC_HOME/sample/gateway.html* では、*proxy* パラメータを次のように編集します。

```
<param name=proxy value="http://hostname/servlet/  
gateway_name.TDSTunnel_Servlet_name">
```

付録 A

SQL の例外メッセージと警告 メッセージ

次の表では、jConnect の使用中に表示される可能性のある SQL の例外メッセージと警告メッセージをリストします。

SQLState	メッセージ / 説明 / 対処方法
010DP	<p>Duplicate connection property _____ ignored.</p> <p>説明：接続プロパティが 2 度定義されました。ドライバ接続プロパティ内で大文字と小文字で 2 度定義されている可能性があります (例: "password" と "PASSWORD" など)。接続プロパティ名では大文字と小文字を区別しないため、jConnect はこのような 2 つのプロパティ名を大文字と小文字であっても同じ名前としてみなします。</p> <p>または、接続プロパティ・リスト内と URL 内の両方で定義されている可能性があります。このような場合には、接続プロパティ・リストの定義が優先されます。</p> <p>対処方法：アプリケーションが、接続プロパティを 1 度だけ定義するようにしてください。アプリケーションによっては、URL よりプロパティ・リストで定義された接続プロパティを優先されるという利点を利用したい場合があります。このような場合には、この警告を無視してください。</p>
010HA	<p>The server denied your request to use the high-availability feature. Please reconfigure your database, or do not request a high-availability session.</p> <p>説明：REQUEST_HA_SESSION 接続プロパティが "true" に設定されておらず、jConnect から接続しようとしたサーバが接続を許可しませんでした。</p> <p>対処方法：高可用性フェールオーバをサポートするようにサーバを再設定してください。または、REQUEST_HA_SESSION を "true" に設定しないでください。</p>
010HD	<p>Sybase high-availability failover is not supported by this type of database server.</p> <p>説明：jConnect から接続しようとしたサーバが高可用性フェールオーバをサポートしていません。</p> <p>対処方法：高可用性フェールオーバをサポートしているデータベース・サーバにのみ接続してください。</p>

SQLState	メッセージ / 説明 / 対処方法
010MX	<p>Metadata accessor information was not found on this database.Please install the required tables as mentioned in the jConnect documentation.Error encountered while attempting to retrieve metadata information:_____</p> <p>説明：サーバが、メタデータ情報を返すのに必要なストアド・プロシージャを持っていない可能性があります。</p> <p>対処方法：メタデータを提供するストアド・プロシージャがサーバにインストールされていることを確認してください。『jConnect for JDBC インストール・ガイド』の第1章の「ストアド・プロシージャのインストール」を参照してください。</p>
010P4	<p>An output parameter was received and ignored.</p> <p>説明：実行したクエリから出力パラメータが返されましたが、アプリケーションの結果処理コードによってフェッチされなかったため、無視されました。</p> <p>対処方法：使用しているアプリケーションで出力パラメータのデータが必要であれば、取得できるようにアプリケーションを修正してください。この修正には、CallableStatement によるクエリの実行と、registerOutputParameter() および getXXX() の呼び出しの追加が必要になることがあります。</p>
010PF	<p>One or more jars specified in the PRELOAD_JARS connection property could not be loaded.</p> <p>説明：これは、PRELOAD_JARS 接続プロパティに JAR 名のカンマ区切りリストを指定して DynamicClassLoader を使用した場合に発生します。DynamicClassLoader は、クラスのロード元となるサーバへの接続をオープンするときに、この接続プロパティに指定されているすべての JAR を "プリロード" しようとします。サーバ上に存在しない JAR 名が指定されていると、上記のエラーメッセージが出力されます。</p> <p>対処方法：アプリケーションの PRELOAD_JARS 接続プロパティに指定した JAR ファイルがすべてサーバ上に存在し、アクセス可能であることを確認してください。</p>
010RC	<p>The requested ResultSet type and concurrency is not supported.They have been converted.</p> <p>説明：サポートされていない ResultSet のタイプと同時実行性の組み合わせを要求しました。要求した値は変換する必要があります。</p> <p>対処方法：サポートされている ResultSet のタイプと同時実行性の組み合わせを要求してください。</p>
010SJ	<p>Metadata accessor information was not found on this database.Please install the required tables as mentioned in the jConnect documentation.</p> <p>説明：メタデータ情報がサーバに設定されていません。</p> <p>対処方法：アプリケーションがメタデータを必要とする場合、jConnect のメタデータを返すストアド・プロシージャをインストールしてください。『jConnect for JDBC インストール・ガイド』の第1章の「ストアド・プロシージャのインストール」を参照してください。メタデータを必要としない場合、USE_METADATA プロパティを "false" に設定してください。</p>

SQLState	メッセージ / 説明 / 対処方法
010SK	<p>Database cannot set connection option _____. 説明：アプリケーションは、接続されているデータベースがサポートしていないオペレーションを行いました。 対処方法：使用しているデータベースをアップグレードする必要がある可能性があります。または、メタデータ情報の最新バージョンがインストールされていることを確認してください。</p>
010SN	<p>Permission to write to file was denied.File:_____. Error message:_____</p> <p>説明：VM でのセキュリティ違反のため、PROTOCOL_CAPTURE 接続プロパティで指定されているファイルへの書き込みパーミッションが拒否されました。これは、指定されたファイルにアプレットが書き込もうとすると発生することがあります。 対処方法：アプレットからファイルに書き込む場合は、アプレットから対象ファイル・システムへのアクセスが可能であることを確認してください。</p>
010SP	<p>File could not be opened for writing.File:_____. Error message:_____</p> <p>対処方法：ファイル名が正しく、そのファイルへの書き込みが可能であることを確認してください。</p>
010TP	<p>The connection's initial character set,_____, could not be converted by the server.The server's proposed character set,_____, will be used, with conversions performed by jConnect.</p> <p>説明：サーバは jConnect によって最初に要求された文字セットを使用できず、異なる文字セットで応答しました。jConnect は変更を受け入れて、必要な文字セット変換を実行します。 このメッセージは情報メッセージです。これ以上の結果はありません。 対処方法：このメッセージを表示しないようにするには、CHARSET 接続プロパティを、サーバがサポートする文字セットに設定します。</p>
010UF	<p>Attempt to execute use database command failed.Error message:_____</p> <p>説明：jConnect が、接続 URL で指定されているデータベースへの接続ができませんでした。可能性のある原因は 2 つあります。</p> <ul style="list-style-type: none"> • URL 内の名前が正しくありません。 • USE_METADATA は "true" (デフォルトの状態) ですが、メタデータを返すストア・プロシージャがインストールされていません。結果として、jConnect は URL 内のデータベースを使用して <i>use database</i> コマンドを実行しようとしたましたが、失敗しました。SQL Anywhere データベースにアクセスしようとした可能性があります。SQL Anywhere データベースは、<i>usedatabase</i> コマンドをサポートしていません。 <p>対処方法：URL 内のデータベース名が正しいことを確認してください。メタデータを返すストア・プロシージャがサーバにインストールされていることを確認してください。『JConnect for JDBC インストール・ガイド』の第 1 章の「ストア・プロシージャのインストール」を参照してください。SQL Anywhere データベースにアクセスしている場合、URL 内でデータベース名を指定しないでください。または、USE_METADATA を "false" に設定してください。</p>

SQLState	メッセージ / 説明 / 対処方法
010UP	<p>Unrecognized connection property _____ ignored.</p> <p>説明：jConnect が現在認識できない URL 内の接続プロパティを設定しようとした。jConnect は認識できないプロパティを無視します。</p> <p>対処方法：アプリケーション内の URL 定義をチェックし、正しい jConnect ドライバ接続プロパティが参照されていることを確認してください。</p>
0100V	<p>The version of TDS protocol being used is too old. Version:_____</p> <p>説明：必要なバージョンの TDS プロトコルをサーバがサポートしていません。jConnect にはバージョン 5.0 以降の TDS プロトコルが必要です。</p> <p>対処方法：必要なバージョンの TDS をサポートしているサーバを使用してください。詳細については、『jConnect for JDBC インストール・ガイド』のシステム移動条件の項を参照してください。</p>
JW0IO	<p>I/O layer:thread operation failed.</p> <p>説明：時間制限された I/O ストリームで内部エラーが発生しました。</p> <p>対処方法：接続を閉じて、再び開いてください。</p>
JZ001	<p>User name property '_____' too long. Maximum length is 30.</p> <p>対処方法：30 バイト以下にしてください。</p>
JZ002	<p>Password property '_____' too long. Maximum length is 30.</p> <p>対処方法：30 バイト以下にしてください。</p>
JZ003	<p>Incorrect URL format.URL:_____</p> <p>対処方法：URL の形式を確認してください。詳細については、「URL 接続プロパティ・パラメータ」(19 ページ)を参照してください。</p> <p>PROXY 接続プロパティを使用している場合、PROXY プロパティのフォーマットが正しくないと接続中に JZ003 例外が発生することがあります。</p> <p>カスケード・プロキシの PROXY フォーマット：</p> <p><i>ip_address:port_number</i></p> <p>TDS トンネリング・サーブレットの PROXY フォーマット：</p> <p><i>http[s]://host:port/tunneling_servlet_alias</i></p>
JZ004	<p>User name property missing in DriverManager.getConnection(..., Properties)</p> <p>対処方法：必要なユーザプロパティを指定してください。</p>
JZ006	<p>Caught IOException:_____</p> <p>説明：予期しない I/O エラーが低いレイヤから検出されました。このような I/O 例外が発生した場合、ERR_IO_EXCEPTION JZ006 sqlstate を使用して、SQL 例外として再度発生します。多くの場合は、ネットワークの通信障害がこのエラーの原因になります。</p> <p>対処方法：文のキャッシュ・サイズを増やしてください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ008	<p>Invalid column index value ____.</p> <p>説明：要求したカラム・インデックス値が 1 よりも小さいか、または有効範囲の上限を超えています。</p> <p>対処方法：getXXX() メソッドの呼び出しとオリジナルのクエリのテキストを確認してください。また、rs.next() を呼び出していることを確認してください。</p>
JZ009	<p>Error encountered in conversion. Error message: ____</p> <p>説明：可能性のある原因を次に示します。</p> <ul style="list-style-type: none"> • 互換性のない 2 つの型の間でデータを変換しようとした。たとえば、date と int は互換性がありません。 • 数字以外の文字を含んだ文字列を数値型に変換しようとした。 • 誤ってフォーマットされた time/date 文字列などのフォーマット・エラーがあります。 <p>対処方法：実行しようとした型変換が JDBC 仕様でサポートされていることを確認してください。文字列が正しくフォーマットされていることを確認してください。数字以外の文字を含む文字列は、数値型に変換できません。</p>
JZ00B	<p>Numeric overflow.</p> <p>説明：BigInteger を TDS 数値として送信しようとしたが、値が大きすぎました。または Java の long を int として送信しようとしたが、値が大きすぎました。</p> <p>対処方法：これらの値は、Sybase に格納できません。long については、Sybase の数値を使用してください。Bignum については、解決方法がありません。</p>
JZ00E	<p>Attempt to call execute() or executeUpdate() for a statement where setCursorName() has been called.</p> <p>対処方法：カーソル名のセットを持つ文で execute または executeUpdate を呼び出さないでください。カーソルの削除および更新には、別々の文を使用してください。詳細については、「結果セットでのカーソルの使用方法」(45 ページ) を参照してください。</p>
JZ00F	<p>Cursor name has already been set by setCursorName().</p> <p>対処方法：1 つの文に対してカーソル名を 2 度設定しないでください。現在のカーソル文の結果セットをクローズしてください。</p>
JZ00G	<p>No column values were set for this row update.</p> <p>説明：カラム値が変更されていないローを更新しようとした。</p> <p>対処方法：ローのカラム値を変更するには、updateRow() を呼び出す前に updateXX() メソッドを呼び出してください。</p>
JZ00H	<p>The result set is not updatable. Use Statement.setResultSetConcurrencyType().</p> <p>対処方法：結果セットを読み込み専用から更新可能に変更するには、Statement.setResultSetConcurrencyType() メソッドを使用するか、または for update 句を SQL select 文に追加してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ00L	Login failed.Examine the SQLWarnings chained to this exception for the reason(s). 対処方法：メッセージ・テキストを参照し、ログインが失敗した理由に応じて対処してください。
JZ010	Unable to deserialize an Object value.Error text: _____ 対処方法：データベースの Java オブジェクトが Serializable インタフェースを実装しており、ローカル CLASSPATH 変数に指定されていることを確認してください。
JZ011	Number format exception encountered while parsing numeric connection property _____. 説明：数値接続プロパティに整数以外の値が指定されました。 対処方法：接続プロパティに整数値を指定してください。
JZ012	Internal Error.Please report it to Sybase technical support.Wrong access type for connection property _____. 対処方法：Sybase 製品の保守契約を結んでいるサポート・センタに連絡してください。
JZ013	Error obtaining JNDI entry:_____ 対処方法：JNDI URL を訂正するか、またはディレクトリ・サービス内に新しいエントリを作成してください。
JZ0BD	Out of range or invalid value used for method parameter. 対処方法：メソッドのパラメータ値が正しいことを確認してください。
JZ0BE	BatchUpdateException:Error occurred while executing batch statement:_____.
JZ0BP	Output parameters are not allowed in Batch Update Statements. 対処方法：
JZ0BR	The cursor is not positioned on a row that supports the _____ method. 説明：現在のロー位置では無効になる ResultSet メソッドを呼び出そうとしました。 たとえば、カーソルが挿入ローに置かれていないときに insertRow() を呼び出すと、このエラーが発生します。 対処方法：現在のロー位置で無効な ResultSet メソッドを呼び出さないでください。
JZ0BS	Batch Statements not supported.
JZ0BT	The _____ method is not supported for ResultSets of type _____. 説明： ResultSet のタイプに対して無効な ResultSet メソッドを呼び出そうとしました。 対処方法： ResultSet のタイプに対して無効な ResultSet メソッドを呼び出さないでください。

SQLState	メッセージ / 説明 / 対処方法
JZ0C0	<p>Connection is already closed.</p> <p>説明：この接続オブジェクトは、アプリケーションからすでに <code>Connection.close()</code> が呼び出されているため、ほかの用途に使用できません。</p> <p>対処方法：接続がクローズされたときに接続オブジェクトの参照が <code>null</code> になるように、コードを修正してください。</p>
JZ0D0	<p>This jConnect installation has not been registered yet.You need to install the appropriate SybDriverKey classes.</p> <p>対処方法：Connect Web サイトで jConnect ソフトウェアを登録してください。 http://www.sybase.com/products/internet/jconnect/ 登録すれば、jConnect ドライバをアクティブにするために必要な SybDriverKey クラスをダウンロードできます。</p>
JZ0D2	<p>Your Sybase JDBC license expired on _____.Please obtain a new license.</p> <p>対処方法：Sybase に連絡して、jConnect ドライバの新しいライセンスを取得してください。</p>
JZ0D3	<p>Your Sybase JDBC license will expire soon.Please obtain a new license.It will expire on _____. </p> <p>対処方法：Sybase に連絡して、jConnect ドライバの新しいライセンスを取得してください。</p>
JZ0D4	<p>Unrecognized protocol in Sybase JDBC URL:_____.</p> <p>説明：TDS 以外のプロトコルを使用して接続の URL を指定しました。現在 jConnect でサポートされているプロトコルは TDS だけです。</p> <p>対処方法：URL 定義を確認してください。URL が TDS をサブプロトコルとして指定する場合、エントリは次のフォーマットを使用し、大文字小文字を区別していることを確認してください。</p> <p><code>jdbc:sybase:Tds:host:port</code> URL JNDI をサブプロトコルとして指定する場合、先頭が次のようになっていることを確認してください。</p> <p><code>jdbc:sybase:jndi:</code></p>
JZ0D5	<p>Error loading protocol _____.</p> <p>対処方法：CLASSPATH システム変数の設定を確認してください。</p>
JZ0D6	<p>Unrecognized version number _____ specified in setVersion.Choose one of the SybDriver.VERSION_* values, and make sure that the version of jConnect that you are using is at or beyond the version you specify.</p> <p>対処方法：メッセージ・テキストを参照してください。</p>
JZ0D7	<p>Error loading url provider _____.Error message:_____</p> <p>対処方法：JNDI URL が正しいことを確認してください。</p>
JZ0D8	<p>Error initializing url provider:_____</p> <p>対処方法：JNDI URL が正しいことを確認してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0EM	End of data. 対処方法：Sybase 製品の保守契約を結んでいるサポート・センタに連絡してください。
JZ0H0	Unable to start thread for event handler; event name = _____. 対処方法：Sybase 製品の保守契約を結んでいるサポート・センタに連絡してください。
JZ0H1	An event notification was received but the event handler was not found; event name = _____. 対処方法：Sybase 製品の保守契約を結んでいるサポート・センタに連絡してください。
JZ0HC	Illegal character '_____' encountered while parsing hexadecimal number. 説明：バイナリ値を表す文字列に含まれている文字が 16 進数の範囲 (0-9, a-f) を超えています。 対処方法：文字列内の文字値を確認して、必要な範囲内に入っているかを確認してください。
JZ0I1	I/O Layer:Error reading stream. 説明：接続で、要求された量を読み込めませんでした。このエラーは、文のタイムアウト時間が超過し、接続がタイムアウトになった場合に発生することがよくあります。 対処方法：文のタイムアウト値を増やしてください。
JZ0I2	I/O layer:Error writing stream. 説明：接続で、要求された出力を書き込めませんでした。このエラーは、文のタイムアウト時間が超過し、接続がタイムアウトになった場合に発生することがよくあります。 対処方法：文のタイムアウト値を増やしてください。
JZ0I3	Unknown property.This message indicates an internal product problem.Report this error to Sybase Technical support. 対処方法：製品内部の問題が発生したことを示します。Sybase 製品の保守契約を結んでいるサポート・センタに連絡してください。
JZ0I5	An unrecognized CHARSET property was specified:_____. 説明：CHARSET 接続プロパティに対して、サポートされていない文字セット・コードを指定しました。 対処方法：接続プロパティに対して有効な文字セット・コードを入力してください。詳細については、「 jConnect 文字セット・コンバータ 」(31 ページ)を参照してください。

SQLState	メッセージ / 説明 / 対処方法
JZ0I6	<p>An error occurred converting UNICODE to the charset used by the server. Error message: _____</p> <p>対処方法：サーバへの送信に必要なすべての文字をサポートできる jConnect クライアントで、CHARSET 接続プロパティに対して別の charset コードを選択してください。異なる文字セットをサーバにインストールする必要がある場合があります。</p>
JZ0I7	<p>No response from proxy gateway.</p> <p>説明：カスケードまたはセキュリティ・ゲートウェイが応答しません。</p> <p>対処方法：ゲートウェイが適切にインストールおよび実行されていることを確認してください。</p>
JZ0I8	<p>Proxy gateway connection refused. Gateway response: _____</p> <p>説明：PROXY 接続プロパティによって示された web サーバ/ゲートウェイが、接続要求を拒否しました。</p> <p>対処方法：proxy のアクセスとエラー・ログを確認して、接続が拒否された原因を調べてください。proxy が JDBC ゲートウェイであることを確認してください。</p>
JZ0I9	<p>This InputStream was closed.</p> <p>説明：InputStream がすでにクローズされているときに getAsciiStream()、getUnicodeStream()、または getBinaryStream() から取得した InputStream を読み込もうとしました。別のカラムに移動したまたは結果セットをキャンセルしたために、リソースが不足してデータをキャッシュできないので、このストリームはクローズされている可能性があります。</p> <p>対処方法：キャッシュ・サイズを増やすか、カラムを順番に読み込んでください。</p>
JZ0IA	<p>Truncation error trying to send _____.</p> <p>説明：文字列の送信前の文字セットの変換中にトランケーション・エラーが発生しました。変換後の文字列の長さが、割り当てられたサイズを超えています。</p> <p>対処方法：サーバへの送信に必要なすべての文字をサポートできる jConnect クライアントで、CHARSET 接続プロパティに対して別の charset コードを選択してください。また、必要に応じて、異なる文字セットをサーバにインストールしてください。</p>
JZ0IS	<p>(getXXXStream may not be called on a column after it has been updated in the result set.)</p> <p>説明：結果セット内でカラムを更新したあと、getAsciiStream()、getUnicodeStream()、getBinaryStream() のいずれかの SybResultSet メソッドを使用して、更新したカラム値を読み込もうとしました。jConnect では、このような使用方法をサポートしていません。</p> <p>対処方法：更新対象のカラムから入力ストリームをフェッチしないでください。</p>
JZ0JO	<p>Offset and/or length values exceed the actual text/image length.</p> <p>対処方法：使用したオフセットか長さの値またはその両方が正しいことを確認してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0NC	<p>カラムを取得するための呼び出しを事前に行わずに <code>wasNull</code> が呼び出されました。</p> <p>説明 : <code>getInt()</code> や <code>getBinaryStream()</code> などのカラムを取得する呼び出しのあとのみ <code>wasNull()</code> を呼び出すことができます。</p> <p>対処方法 : コードを変更して <code>wasNull()</code> の呼び出しを移動してください。</p>
JZ0NE	<p><code>Incorrect URL format.URL:_____. Error message:_____</code></p> <p>対処方法 : URL のフォーマットを確認してください。ポート番号には、数値文字だけが使用されていることを確認してください。</p>
JZ0NF	<p><code>Unable to load SybSocketFactory.Make sure that you have spelled the class name correctly, that the package is fully specified, that the class is available in your class path, and that it has a public zero-argument constructor.</code></p> <p>対処方法 : メッセージ・テキストを参照してください。</p>
JZ0P1	<p><code>Unexpected result type.</code></p> <p>説明 : データベースが結果を返したが、文がアプリケーションに返すことのできない結果でした。または、アプリケーションがこの時点で予期していない結果でした。一般的に、アプリケーションが JDBC を不適切に使用しているために、クエリまたはストアド・プロシージャを実行できないことを示します。JDBC アプリケーションが Open Server アプリケーションに接続されている場合、Open Server アプリケーション内のエラーのために、Open Server が予期しない結果のシーケンスを送信している可能性があります。</p> <p>対処方法 : デバッグ・ツール <code>com.sybase.utils.Debug(true, "ALL")</code> を使用して、どのような予期しない結果が表示されるかを把握し、原因を明らかにしてください。</p>
JZ0P4	<p><code>Protocol error.This message indicates an internal product problem.Report this error to Sybase technical support.</code></p> <p>対処方法 : メッセージ・テキストを参照してください。</p>
JZ0P7	<p><code>Column is not cached; use RE-READABLE_COLUMNS property.</code></p> <p>説明 : <code>REPEAT_READ</code> 接続プロパティを <code>"false"</code> に設定して、カラムを再読み込みしようとしたか、カラムを誤った順序で読み込もうとしました。</p> <p><code>REPEAT_READ</code> が <code>"false"</code> の場合、ローのカラム値を一度だけ読み込むことができ、昇順カラム／インデックス順でのみカラムを読み込むことができます。たとえば、ローのカラム 3 を読み込んだあとには、再度その値を読み込むことはできず、ローのカラム 2 を読み込むことはできません。</p> <p>対処方法 : <code>REPEAT_READ</code> を <code>"true"</code> に設定するか、またはカラム値を再度読み込まないようにしてください。また、カラムの読み込みは、昇順カラム／インデックス順であることを確認してください。</p>
JZ0P8	<p><code>The RSMData Column Type Name you requested is unknown.</code></p> <p>説明 : <code>jConnect</code> が <code>ResultSetMetaData.getColumnTypeName()</code> メソッドのカラム・タイプ名を判別できませんでした。</p> <p>対処方法 : データベースが、最新のメタデータ用のストアド・プロシージャを持っていることを確認してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0P9	<p>A COMPUTE BY query has been detected.That type of result is unsupported and has been cancelled.</p> <p>説明：jConnect でサポートされていない COMPUTE 結果がクエリから返されました。</p> <p>対処方法：COMPUTE BY を使用しないようにクエリまたはストアド・プロシージャを変更してください。</p>
JZ0PA	<p>The query has been cancelled and the same response discarded.</p> <p>説明：この接続上の他の文がキャンセルの原因となっている可能性があります。</p> <p>対処方法：この文とその他の文の一連の SQL 例外および警告を確認して、原因を調べてください。</p>
JZ0PB	<p>The server does not support a requested operation.</p> <p>説明：jConnect は、サーバへの接続を作成するときに、サポートされている必要がある機能をサーバに通知します。サーバは、実際にサポートしている機能を jConnect に通知します。アプリケーションが元の機能ネゴシエーション内で拒否された操作を要求すると、このエラー・メッセージが送信されます。</p> <p>たとえば、動的 SQL 文のプリコンパイルをサポートしていないデータベースに対し、dynamic を "true" に設定した状態でコードから SybConnection.prepareStatement(sql_stmt, dynamic) を呼び出すと、このメッセージが出力されます。</p> <p>対処方法：プログラムを修正して、サポートされていない機能を要求しないようにしてください。</p>
JZ0R0	<p>Result set has already been closed.</p> <p>説明：ResultSet.close() メソッドがすでに結果セット・オブジェクトに対して呼び出されています。この結果セットは、ほかの用途に使用できません。</p> <p>対処方法：接続がクローズされたときに ResultSet オブジェクトの参照が null になるように、コードを修正してください。</p>
JZ0R1	<p>Result set is IDLE as you are not currently accessing a row.</p> <p>説明：アプリケーションが ResultSet.getXXX カラム・データの検索メソッドの 1 つを呼び出しましたが、現在のローがありません。アプリケーションは ResultSet.next() を呼び出していません。または、データがないことを示す false が ResultSet.next() から返されました。</p> <p>対処方法：rs.next() が "true" に設定されていることを確認してから、rs.getXXX を呼び出してください。</p>
JZ0R2	<p>No result set for this query.</p> <p>説明：Statement.executeQuery() を使用しましたが、文がローを返しませんでした。</p> <p>対処方法：ローを返さない文に対しては、executeUpdate を使用してください。</p>
JZ0R3	<p>Column is DEAD.This is an internal error.Please report it to Sybase technical support.</p> <p>対処方法：メッセージ・テキストを参照してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZOR4	<p>Column does not have a text pointer.It is not a text/image column or the column is NULL.</p> <p>説明：text/image カラムは、NULL の場合には更新できません。NULL text/image カラムにテキスト・ポインタがありません。</p> <p>対処方法：text/image データをサポートしていないカラムへのテキスト・ポインタを更新または取得しようとしていないことを確認してください。NULL になっている text/image カラムを更新しようとしていないことを確認してください。データを先に挿入してから更新してください。</p>
JZORM	<p>refreshRow may not be called after updateRow or deleteRow.</p> <p>説明：データベース内のローを、<code>SybCursorResult.updateRow()</code> を使用して更新、または <code>SybCursorResult.deleteRow()</code> を使用して削除したあと、データベースからローをリフレッシュするために <code>SybCursorResult.refreshRow()</code> を使用しました。</p> <p>対処方法：データベースのローを更新またはデータベースから削除したあとは、ローをリフレッシュしないでください。</p>
JZOS0	<p>Statement state machine:Statement is BUSY.</p> <p>説明：このエラーは、アプリケーションがカーソル名を設定するときに、文がすでに使用されていて読み込む必要のあるカーソル以外の結果がある場合に、<code>Statement.setCursorname()</code> メソッドから生成されます。</p> <p>対処方法：文にカーソル名を設定してから、クエリを実行してください。またはカーソル名を設定する前に <code>Statement.cancel()</code> を呼び出して、文がビジーでないことを確認してください。</p>
JZOS1	<p>Statement state machine:Trying to FETCH on IDLE statement.</p> <p>説明：文で、内部エラーが発生しました。</p> <p>対処方法：文をクローズして、別の文をオープンしてください。</p>
JZOS2	<p>Statement object has already been closed.</p> <p>説明：文オブジェクトに対して、<code>Statement.close()</code> メソッドが既に呼び出されています。このオブジェクトは、ほかの用途に使用できません。</p> <p>対処方法：文がクローズされたときに <code>Statement</code> オブジェクトの参照が null になるように、アプリケーションを修正してください。</p>
JZOS3	<p>The inherited method _____ cannot be used in this subclass.</p> <p>説明：PreparedStatement は、<code>executeQuery(String)</code>、<code>executeUpdate(String)</code>、および <code>execute(String)</code> をサポートしていません。</p> <p>対処方法：クエリ文字列を渡す場合は、PreparedStatement ではなく Statement を使用してください。</p>
JZOS4	<p>Cannot execute an empty (zero-length) query.</p> <p>対処方法：空のクエリ ("") を実行しないでください。</p>
JZOS8	<p>An escape sequence in a SQL Query was malformed: '_____'. </p> <p>説明：このエラーは、エスケープ構文に誤りがあることを示します。</p> <p>対処方法：正しい構文については、JDBC のマニュアルを参照してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0S9	<p>Cannot execute an empty (zero-length) query.</p> <p>対処方法：空のクエリ ("") を実行しないでください。</p>
JZ0SA	<p>Prepared Statement:Input parameter not set, index:_____.</p> <p>対処方法：各入力パラメータに値が設定されていることを確認してください。</p>
JZ0SB	<p>Parameter index out of range:_____.</p> <p>説明：パラメータを取得、設定、または登録したときに、パラメータの最大数を超えました。</p> <p>対処方法：クエリのパラメータ数を確認してください。</p>
JZ0SC	<p>Callable Statement:attempt to set the return status as an InParameter.</p> <p>説明：ステータスを返すストアド・プロシージャの呼び出しを準備しましたが、リターン・ステータスであるパラメータ 1 を設定しています。</p> <p>対処方法：このタイプの呼び出しでは、2 以降のパラメータを設定できます。</p>
JZ0SD	<p>No registered parameter found for output parameter.</p> <p>説明：これは、アプリケーションの論理エラーを示します。パラメータで <code>getXXX()</code> または <code>wasNull()</code> を呼び出しましたが、パラメータを読み込んでいないか、または出力パラメータがありませんでした。</p> <p>対処方法：アプリケーションに呼び出し可能な文の出力パラメータが登録されていることを確認してください。また、文が実行され、出力パラメータが読み込まれたことを確認してください。</p>
JZ0SE	<p>Invalid object type specified for setObject().</p> <p>説明：不正なデータ型の引数が、<code>PreparedStatement.setObject</code> に渡されました。</p> <p>対処方法：JDBC のマニュアルを確認してください。引数は、<code>java.sql.Types</code> の定数でなければなりません。</p>
JZ0SF	<p>No Parameters expected.Has query been sent?</p> <p>説明：パラメータをとらない文に対してパラメータを設定しようとしてしました。</p> <p>対処方法：クエリが送信されたことを確認してから、パラメータを設定してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0SG	<p>An RPC did not return as many output parameters as the application had registered for it.</p> <p>説明：このエラーは、ストアド・プロシージャ内の "OUTPUT" パラメータとして宣言したよりも多いパラメータに対して <code>CallableStatement.registerOutParam()</code> を呼び出すと発生します。詳細については、「RPC が登録数よりも少ない出力パラメータを返す」(107 ページ) を参照してください。</p> <p>対処方法：使用しているストアド・プロシージャおよび <code>registerOutParameter</code> 呼び出しを確認してください。"OUTPUT" として適切なパラメータがすべて宣言されていることを確認してください。読み込んだコードの行を確認してください。</p> <pre>create procedure yourproc (@p1 int OUTPUT, ...</pre> <p>注意 Adaptive Server Anywhere (以前の SQL Anywhere) を使用しているときにこのエラーが発生した場合は、Adaptive Server Anywhere バージョン 5.5.04 にアップグレードしてください。</p>
JZ0SH	<p>A static function escape was used, but the metadata accessor information was not found on this server.</p> <p>対処方法：メタデータ・アクセッサ情報をインストールしてから、静的関数のエスケープを使用してください。</p>
JZ0SI	<p>A static function escape _____ was used which is not supported by this server.</p> <p>対処方法：このエスケープを使用しないでください。</p>
JZ0SJ	<p>Metadata accessor information was not found on this database.</p> <p>対処方法：メタデータ情報をインストールしてから、メタデータを呼び出してください。</p>
JZ0SM	<p>Unsupported SQL type _____.</p> <p>対処方法：Types.NULL、Types.OTHER、または <code>PreparedStatement.setObject(null)</code> を使用しないでください。</p>
JZ0SN	<p>setMaxFieldSize: フィールド・サイズを負の数にすることはできません。</p> <p>対処方法：<code>setMaxFieldSize</code> を呼び出すときは、正の値か 0 (無制限) を使用してください。</p>
JZ0T2	<p>Listener thread read error.</p> <p>対処方法：ネットワークの通信を確認してください。</p>
JZ0T3	<p>Read operation timed out.</p> <p>説明：クエリに対する応答の読み込みに割り当てられた時間を超過しました。</p> <p>対処方法：<code>Statement.setQueryTimeout()</code> を呼び出して、タイムアウト時間を増やしてください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0T4	<p>Write operation timed out.Timeout in milliseconds:_____.</p> <p>説明：要求の送信に割り当てた時間を超過しました。</p> <p>対処方法：Statement.setQueryTimeout() を呼び出して、タイムアウト時間を増やしてください。</p>
JZ0T5	<p>Cache used to store responses is full.</p> <p>対処方法：STREAM_CACHE_SIZE 接続プロパティにデフォルト値または現在の値よりも大きい値を使用してください。</p>
JZ0T6	<p>Error reading tunneled TDS URL.</p> <p>説明：URL のヘッダの読み込み中に、Tunnelled プロトコルが失敗しました。</p> <p>対処方法：接続に対して定義した URL を確認してください。</p>
JZ0T7	<p>Listener thread read error -- caught ThreadDeath.Check network connection.</p> <p>対処方法：ネットワーク接続を確認して、アプリケーションを再度実行してください。スレッドが引き続きアボートされる場合には、Sybase 製品の保守契約を結んでいるサポート・センタに問い合わせてください。</p>
JZ0T9	<p>Request to send not synchronized.Please report this error to Sybase Technical Support.</p> <p>対処方法：メッセージ・テキストを参照してください。</p>
JZ0TC	<p>Attempted conversion between an illegal pair of types.</p> <p>説明：Java の型と SQL の型との間の変換に失敗しました。</p> <p>対処方法：要求された型の変換が JDBC 仕様でサポートされていることを確認してください。</p>
JZ0TE	<p>Attempted conversion between an illegal pair of types.Valid database types are:'_____.'</p> <p>説明：データベースのカラム・データ型と ResultSet.getXXX() 呼び出しで要求されたデータ型の間に暗黙的な互換性はありません。</p> <p>対処方法：エラー・メッセージに示された有効なデータ型のいずれかを使用してください。</p>
JZ0US	<p>The SybSocketFactory connection property was set, and the PROXY connection property was set to the URL of a servlet.The jConnect driver does not support this combination.If you want to send secure HTTP from an applet running within a browser, use a proxy URL beginning with "https://".</p> <p>対処方法：メッセージ・テキストを参照してください。</p>

SQLState	メッセージ / 説明 / 対処方法
JZ0CX	<p>_____ is an unrecognized transaction coordinator type.</p> <p>説明：メタデータ情報には、サーバで分散トランザクションがサポートされていることが示されていますが、jConnect でサポートされていないプロトコルが使用されています。</p> <p>対処方法：最新のメタデータ・スクリプトがインストールされていることを確認してください。エラーが再発する場合は、Sybase 製品の保守契約を結んでいるサポート・センタにエラーを報告してください。</p>
JZ0XS	<p>サーバは、XA スタイルのトランザクションをサポートしていません。トランザクション機能が有効になっており、このサーバでライセンスされているかどうかを確認してください。</p> <p>説明：jConnect から接続しようとしたサーバが分散トランザクションをサポートしていません。</p> <p>対処方法：このサーバに対しては、XADataSource を使用しないでください。または、分散トランザクションを使用できるように、このサーバをアップグレードするか、再設定してください。</p>
JZ0XU	<p>Current user does not have permission to do XA-style transactions. Be sure user has _____ role.</p> <p>説明：データベースに接続したユーザには、分散トランザクションを実行する権限が与えられていません。ユーザの役割が適切に設定されていない(ブランクになっている)可能性があります。</p> <p>対処方法：エラー・メッセージに示されている役割をユーザに付与してください。または、適切な役割がすでに付与されているユーザにトランザクションを実行させてください。</p>
S0022	<p>Invalid column name '_____'. 説明：カラムを名前参照しようとしたが、その名前のカラムが存在しません。 対処方法：カラム名の綴りを確認してください。</p>
ZZ00A	<p>The method _____ has not been completed and should not be called.</p> <p>説明：実装されていないメソッドを使用しようとした。</p> <p>対処方法：詳細については、jConnect の使用しているバージョンに添付されていたリリース・ノートを確認してください。また、jConnect の最新バージョンがそのメソッドを実装しているかどうかを参照するには、jConnect web ページ http://www.sybase.com を参照してください。記載されていない場合、そのメソッドを使用しないでください。</p>

付録 B

jConnect のサンプル・プログラム

この付録では、Sybase jConnect のサンプル・プログラムについて説明します。

ここでは、次の項目について説明します。

項名	ページ
IsqlApp の実行	166
jConnect のサンプル・プログラムとサンプル・コードの実行	169

IsqlApp の実行

IsqlApp を使用すると、コマンド・ラインから **isql** コマンドを発行して、jConnect のサンプル・プログラムを実行することができます。

IsqlApp の構文は次のとおりです。

```
IsqlApp [-U username] [-P password]
        [-S servername]
        [-G gateway]
        [-p {http|https}]
        [-D debug-class-list]
        [-v]
        [-I input-command-file]
        [-c command_terminator]
        [-C charset] [-L language]
        [-T sessionID]
        [-V <version {2,3,4,5}>]
```

パラメータ	説明
-U	サーバに接続するログイン ID。
-P	指定したログイン ID のパスワード。
-S	接続先のサーバの名前。
-G	ゲートウェイ・アドレス。HTTP プロトコルの場合、URL は <code>http://host:port</code> 。暗号化をサポートする HTTPS プロトコルを使用する場合、URL は <code>https://host:port/servlet_alias</code> 。
-p	HTTP プロトコルを使用するか、暗号化をサポートする HTTPS プロトコルを使用するかを指定する。
-D	すべてのクラスについて、またはカンマで区切って指定したものだけについて、デバッグをオンにする。次に例を示す。 -D ALL すべてのクラスについてデバッグを表示する。 -D SybConnection, Tds SybConnection と Tds についてだけデバッグを表示する。
-v	表示または印刷の冗長出力をオンにする。
-I	IsqlApp が、キーボードからではなく、ファイルからコマンドを取得するようにする。 パラメータの後に、 IsqlApp 入力に使用するファイルの名前を指定する。ファイルにはコマンド・ターミネータが含まれていなければならない（デフォルトでは "go")。

パラメータ	説明
-c	行に単独で入力されたときにコマンドを終了させるキーワード（たとえば、"go"）を指定できる。これによって、ターミネータ・キーワードを使用するまで複数行にわたってコマンドを入力できる。コマンド・ターミネータを指定しない場合は、コマンドは復帰改行ごとに終了する。
-C	TDS に渡された文字列用の文字セットを指定する。 文字セットを指定しないと、IsqlApp はサーバのデフォルト文字セットを使用する。
-L	サーバから返されるエラー・メッセージおよび jConnect メッセージを表示する言語。
-T	このパラメータが設定されていると、jConnect は、TDS トンネリング・ゲートウェイによって開かれている既存の TDS セッション上でアプリケーションが通信を再開しようとしていると想定する。jConnect はログイン・ネゴシエーションをスキップし、アプリケーションからの要求をすべて特定のセッション ID に転送する。
-V	バージョン固有の特性を使用できるようにする。詳細については、 「JCONNECT_VERSION 接続プロパティ」(9 ページ) を参照。

注意 各オプション・フラグのあとにはスペースを 1 つ入力する必要があります。

コマンド・ライン・オプションの詳細な説明を表示するには、次のように入力します。

```
java IsqlApp -help
```

次の例は、ポート "3756" を使用して "myserver" というホスト上のデータベースに接続して、"myscript" という **isql** スクリプトを実行する方法を示します。

```
java IsqlApp -U sa -P sapassword
-S jdbc:sybase:Tds:myserver:3756
```

```
-I $JDBC_HOME/sp/myscript -c run
```

注意 各バージョンの jConnect には、GUI から **isql** コマンドにアクセスするためのアプレットが用意されています。

jConnect 4.x の場合 :

\$JDBC_HOME/sample/gateway.html (UNIX)

%JDBC_HOME%\sample\gateway.html (Windows)

jConnect 5.x の場合 :

\$JDBC_HOME/sample2/gateway.html (UNIX)

%JDBC_HOME%\sample2\gateway.html (Windows)

jConnect のサンプル・プログラムとサンプル・コードの実行

jConnect で提供されているいくつかのサンプル・プログラムは、この章で説明している多くの項目について例を示し、さまざまな JDBC クラスとメソッドでの jConnect の動作方法の理解に役立ちます。この項には、参照用のサンプル・コードも含まれています。

サンプル・アプリケーション

jConnect をインストールするときに、サンプル・プログラムもインストールできます。これらのサンプルにはソース・コードが含まれており、jConnect がどのようにしてさまざまな JDBC クラスとメソッドを実装するかを見ることができます。サンプル・プログラムをインストールする方法については、『jConnect for JDBC インストール・ガイド』を参照してください。

注意 jConnect サンプル・プログラムはデモ用としてのみ提供されています。

サンプル・プログラムは、jConnect インストール・ディレクトリの *sample* サブディレクトリ (jConnect 4.x) または *sample2* サブディレクトリ (jConnect 5.x) にインストールされます。*sample* または *sample2* サブディレクトリにある *index.html* ファイルには、使用できるサンプルのリストと、各サンプルの説明が含まれています。また、*index.html* を使用してサンプル・プログラムをアプレットとして参照、実行することができます。

サンプル・アプレットの実行

使用している Web ブラウザで、サンプル・プログラムのいくつかをアプレットとして実行できます。これによって、出力結果を表示しながらソース・コードを参照することができます。

サンプルをアプレットとして実行するには、カスケード・ゲートウェイを起動する必要があります。詳細については、「[カスケード・ゲートウェイの使用](#)」(137 ページ) を参照。

使用している Web ブラウザで *index.html* を開きます。

jConnect 4.x の場合は、次のように入力します。

http://localhost:8000/sample/index.html

jConnect 5.x の場合は、次のように入力します。

http://localhost:8000/sample2/index.html

Adaptive Server Anywhere でのサンプル・プログラムの実行

サンプル・プログラムはすべて Adaptive Server と互換性がありますが、Adaptive Server Anywhere と互換性のあるものは限られています。現在 Adaptive Server Anywhere と互換性のあるサンプル・プログラムのリストについては、*sample* または *sample2* サブディレクトリにある *index.html* を参照してください。

Adaptive Server Anywhere に使用できるサンプル・プログラムを実行するには、Adaptive Server Anywhere サーバに *pubs2_any.sql* スクリプトをインストールする必要があります。スクリプトは *sample* サブディレクトリ (jConnect 4.1) または *sample2* サブディレクトリ (jConnect 5.0) にあります。

Windows の場合、DOS コマンド・ウィンドウで次のように入力します。

```
java IsqlApp -U dba -P password
-S jdbc:sybase:Tds:[hostname]:[port]
-I %JDBC_HOME%\sample\pubs2_any.sql -c go
```

UNIX の場合、次のように入力します。

```
java IsqlApp -U dba -P password
-S jdbc:sybase:Tds:[hostname]:[port]
-I $JDBC_HOME/sample/pubs2_any.sql -c go
```

サンプル・コード

次のサンプル・コードは、どのように jConnect ドライバを呼び出し、接続を行い、SQL 文を発行して結果を処理するかを示します。

```
import java.io.*;
import java.sql.*;

public class SampleCode
{
    public static void main(String args[])
    {
        try
```

```
{
/*
 * Open the connection. May throw a SQLException.
 */
    Connection con = DriverManager.getConnection(
        "jdbc:sybase:Tds:myserver:3767", "sa", "");
/*
 * Create a statement object, the container for the SQL
 * statement. May throw a SQLException.
 */
    Statement stmt = con.createStatement();
/*
 * Create a result set object by executing the query.
 * May throw a SQLException.
 */
    ResultSet rs = stmt.executeQuery("Select 1");
/*
 * Process the result set.
 */

    if (rs.next())
    {
        int value = rs.getInt(1);
        System.out.println("Fetched value " + value);
    }
}
/*
 * Exception handling.
 */
catch (SQLException sqe)
{
    System.out.println("Unexpected exception : " +
        sqe.toString() + ", sqlstate = " +
        sqe.getSQLState());

    System.exit(1);
}
System.exit(0);
}
```


索引

A

Adaptive Server
 接続 17
 接続例 18
Adaptive Server Anywhere 16
 Java オブジェクトの格納と検索 73
 SERVICENAME 接続プロパティ 18
 イメージ・データの送信 60
 接続 19
 メタデータのアクセス 44
 ユーロ記号 35
APPLICATIONNAME 接続プロパティ 12

C

CANCEL_ALL 接続プロパティ 6, 9, 13
CHARSET 接続プロパティ 6, 13
 設定 32
CHARSET_CONVERTER 接続プロパティ 6
CHARSET_CONVERTER_CLASS 接続プロパティ 13, 32
CLASSPATH
 デバッグ用の設定 99
Compute 文 94
CONNECTION_FAILOVER 接続プロパティ 13, 21

D

Debug クラス 98
DYNAMIC_PREPARE 接続プロパティ 13

E

EXPIRESTRING 接続プロパティ 13

H

HOSTNAME 接続プロパティ 13
HOSTPROC 接続プロパティ 13
HTTP 134

I

IGNORE_DONE_IN_PROC 接続プロパティ 13
isql アプレット
 サンプルの実行 141
IsqlApp ユーティリティ 166

J

JARS
 プリロード 81
JARS のプリロード 81
Java オブジェクト
 ASA 6.0 での格納と検索 73
 カラム・データの格納 73
 テーブル内のカラム・データとしての格納 73
jConnect
 jConnect 4.1 へのアプリケーションのマイグレート 126
 jConnect 4.2 および 5.2 へのアプリケーションのマイグレート 127
 jConnect 5.x へのアプリケーションのマイグレート 126
 アプリケーションでのメモリの問題 106
 カーソルの使用 45
 カスケード・ゲートウェイ 137
 ゲートウェイ 134
 サンプル・プログラム 169
 接続プロパティの設定 12
 設定 6

定義 4
デバッグ 98
パフォーマンスの改善 112
呼び出し 10
jConnect 4.x
 SCROLL_INSENSITIVE 結果セット 53
jConnect 4.x での SCROLL_INSENSITIVE 結果セ
 ット 53
jConnect アプリケーションでのメモリの問題 106
jConnect の呼び出し 10
JCONNECT_VERSION 接続プロパティ 9, 13
JDBC
 インタフェース 2
 制約、制限、逸脱 93
 定義 2
 ドライバ・タイプ 2
JDBC 2.0
 Optional Package の拡張機能のサポート 82
 標準拡張機能 82
JDBC 規格からの逸脱 93
JDBC ドライバ
 JDBC-ODBC ブリッジ 2
 JDBC-ODBC ブリッジ・ドライバ 2
 ネイティブ API / 部分 Java 3
 ネイティブ・プロトコル / 全部 Java 3
 ネット・プロトコル / 全部 Java 3
jdbc.drivers 10
JNDI
 コンテキスト情報 24
 使用 21

L

LANGUAGE 接続プロパティ 6, 14
LANGUAGE_CURSOR 122
LANGUAGE_CURSOR 接続プロパティ 14
Lightweight Directory Access Protocol (LDAP) 21
LITERAL_PARAMS 接続プロパティ 14

O

Open Server Gateway 20

P

PACKETSIZE 接続プロパティ 14
PreparedStatement
 カーソルの使用 52
PROTOCOL_CAPTURE 接続プロパティ 14
PROXY 接続プロパティ 14
PureConverter クラス 31

R

REMOTEPWD 接続プロパティ 14
REPEAT_READ 113
REPEAT_READ 接続プロパティ 15
REQUEST_HA_SESSION 15
rs.getBytes() 64

S

SELECT_OPENS_CURSOR 接続プロパティ 15
SERIALIZE_REQUESTS 接続プロパティ 16
SERVICENAME 接続プロパティ 16
SESSION_ID 接続プロパティ 16
SESSION_TIMEOUT 接続プロパティ 16
setRemotePassword() 43
SQL の例外メッセージと警告メッセージ 149
SQLNITSTRING 接続プロパティ 16
Statement.cancel() 9
STREAM_CACHE_SIZE 接続プロパティ 16
Sybase 拡張機能の変更 129
SybEventHandler 65
SybMessageHandler 69
SYB SOCKET_FACTORY 接続プロパティ 16

T

TDS 4
 サブレット 134
 サブレット・システムの稼働条件 144
 サブレットのインストール 145
 サブレットの引数の設定 145
 セッションの再開 147
 セッションのトラッキング 146

通信の取得 102
トンネリング 134
TDS セッションのトラッキング 146
TDS 通信の取得 102
Technical Library x
Time、Date、Timestamp データ型 63
TruncationConverter クラス 31, 36
TYPE_SCROLL_INSENSITIVE 制限 54

U

URL
構文 18
接続プロパティ・パラメータ 19
USE_METADATA 接続プロパティ 17
USER 17

V

VERSIONSTRING 接続プロパティ 17

W

Web サーバ・ゲートウェイ 134

X

XAServer 89

あ

アプリケーション
jConnect 4.1 へのマイグレート 126
jConnect 4.2 および 5.2 へのマイグレート 127
jConnect 5.x へのマイグレート 126
デバッグ機能をオフにする 99
デバッグ機能をオンにする 99
アプリケーションでのデバッグ機能のオフ 99
アプリケーションでのデバッグ機能のオン 99
アプレット 137, 138

い

位置付け更新と削除
JDBC 1.x メソッドの使用 48
JDBC 2.0 メソッドの使用 49
イベント通知 65
例 66
イメージ・データ
TextPointer オブジェクトの取得 61
TextPointer クラスでのパブリック・メソッド 60
TextPointer.sendData を使用した更新の実行 61
TextPointer.sendData() を使用したカラムの更新 61
送信 60
イメージ・データの送信 60
インストール
TDS サブレット 145
エラー・メッセージ・ハンドラ 71
カスケード・ゲートウェイ 138
インタフェース、JDBC 2

え

エラー
ストアド・プロシージャ 107
接続 104, 105
エラー・メッセージ
SQL 例外と警告 149
Sybase 固有 68
エラー・メッセージ・ハンドラのインストーラ 71
エラー・メッセージ・ハンドラの例 71
処理 68
処理のカスタマイズ 69

か

カーソル 45
PreparedStatement の使用 52
作成 46
カーソル結果セット

- JDBC 1.x メソッドを使用した位置付け更新と削除 48
- JDBC 2.0 メソッドを使用した位置付け更新と削除 49
- 位置付け更新 49
- カラムの更新 50
- 削除 49
- データベースを更新するためのメソッド 50
- ローの削除 52
- ローの挿入 52
- カーソルの作成 46
- カーソルのパフォーマンス
 - LANGUAGE_CURSOR 接続プロパティ 122
- 拡張機能の変更、Sybase 129
- カスケード・ゲートウェイ 137
 - インストール 138
 - 起動 138
 - 接続の定義 142
 - テスト 139
- カラム
 - カーソル結果セットでの更新 50
 - カーソル結果セットでの削除 49
- 関連マニュアル ix

け

- ゲートウェイ 134
 - Open Server 20
 - 接続の拒否 104
 - 設定 135

こ

- 高可用性サポート 37
- 更新
 - ストアド・プロシージャの結果セットによるデータベースの更新 58
- 高度な機能 65
- 構文の表記規則 xii
- 国際化 31

さ

- サーバ間リモート・プロシージャ・コール 42
- サーブレット 134
 - TDS 134
- サーブレットの引数
 - Debug 145
 - SkipDoneProc 145
 - TdsResponseSize 145
 - TdsSessionIdleTimeout 145
- 再開
 - TDS セッション 147
- サンプル・プログラム 169

し

- システム・プロパティ
 - jdbc.drivers 10
- 処理
 - エラー・メッセージ 68

す

- ストアド・プロシージャ
 - エラー 107
 - 結果セットからのデータベースの更新 58
 - 実行 94

せ

- 接続
 - Adaptive Server 17
 - Adaptive Server Anywhere 19
 - JNDI を使用したサーバ 21
 - エラー 104, 106
 - カスケード・ゲートウェイへの接続の定義 142
 - ゲートウェイ接続の拒否 104
 - プーリング 86
- 接続プロパティ
 - APPLICATIONNAME 12
 - CANCEL_ALL 6, 9, 13
 - CHARSET 6, 13

CHARSET_CONVERTER 6
CHARSET_CONVERTER_CLASS 13, 32
CONNECTION_FAILOVER 13, 21
DYNAMIC_PREPARE 13
EXPIRESTRING 13
HOSTNAME 13
HOSTPROC 13
IGNORE_DONE_IN_PROC 13
JCONNECT_VERSION 9, 13
LANGUAGE 6, 14
LANGUAGE_CURSOR 14, 122
LANGUAGE_CURSOR とカーソルのパフォーマンス 122
LITERAL_PARAMS 14
PACKETSIZE 14
PROTOCOL_CAPTURE 14
PROXY 14
REMOTEPWD 14
REPEAT_READ 15, 113
REQUEST_HA_SESSION 15
SELECT_OPENS_CURSOR 15
SERIALIZE_REQUESTS 16
SERVICENAME 16
SESSION_ID 16
SESSION_TIMEOUT 16
SQLNSTRING 16
STREAM_CACHE_SIZE 16
SYB SOCKET_FACTORY 16
URL での設定 19
USE_METADATA 17
USER 17
VERSIONSTRING 17
設定 12
パスワード 14

設定

jConnect 接続プロパティ 12
TDS サブレットの引数 145

た

対象読者 ix

ち

直列化解除 80

つ

通貨記号、ユーロ 35

て

データ型

Time、Date、Timestamp 63

データベース

テーブル内のカラム・データとしての Java オブジェクトの格納 73

ネーミング・データベース用の JNDI 82

テーブル内のカラム・データとしての Java オブジェクトの格納 73

前提条件 74

データベースからの Java オブジェクトの受信 75

データベースへの Java オブジェクトの送信 74

デバッグ 98

CLASSPATH の設定 99

Debug クラスのインスタンスの取得 98

アプリケーションでオフにする 99

アプリケーションでオンにする 99

メソッド 100

と

動的クラスのロード 78

ドライバ

JDBC タイプ 2

プロパティ 12

トラブルシューティング 98

トンネリング

TDS 134

ね

ネイティブ API / 部分 Java ドライバ 3
ネイティブ・プロトコル / 全部 Java ドライバ 3
ネーミング・データベース用の JNDI 82
ネット・プロトコル / 全部 Java ドライバ 3

は

パスワード 14
バッチ更新 58
 ストアド・プロシージャ 57
パフォーマンスのチューニング 112

ひ

表記規則 xi

ふ

ブーリング接続 86
フォント規則 xi
プロパティ
 ドライバ 12
分散トランザクションのサポート 89

ま

マルチスレッド
 調整 93
マルチバイト文字セット
 コンバータ・クラス 31
 サポート 33

め

メタデータ
 USE_METADATA 17
 アクセス 44
 サーバ側の実装 45

も

文字セット
 コンバータ・クラス 31
 サポート 33
 設定 32
文字セットのコンバータ・クラス
 PureConverter 31
 TruncationConverter 31
 選択 32
文字セットのコンバータ・クラスの選択 32
文字セット変換
 ドライバ・パフォーマンスの改善 114
 パフォーマンスの改善 33

ゆ

ユーティリティ
 IsqlApp 166
ユーロ通貨記号 35

り

リモート・プロシージャ・コール (RPC)
 サーバカン 42

ろ

ロー
 カーソル結果セットからの削除 51
 カーソル結果セットでの挿入 52
ローカライゼーション 31