# DBMaster

## SQLストアド・プロシージャ参照編

SYSCOM

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive
Santa Clara, CA 95050, U.S.A.

**Contact Information:**
CASEMaker US Division
E-mail : info@casemaker.com
Europe Division
E-mail : casemaker.europe@casemaker.com
Asia Division
E-mail : casemaker.asia@casemaker.com(Taiwan)
E-mail : info@casemaker.co.jp(Japan)

www.casemaker.com
www.casemaker.com/support


Document No. 645049-234193/DBM52J-M04082010-SLSP
発行日:2010-04-08



本文には記されていない新しい機能についての説明は、CASEMakerのDBMasterをインストールしてからREADME.TXTを読んでください。

## 登録商標



## 注意事項

本書で記述されるソフトウェアは、ソフトウェアと共に提供される使用許諾書に基づきます。

保証については、ご利用の販売店にお問い合わせ下さい。販売店は、特定用途への本コンピュータ製品の商品性や適合性について、代表または保証しません。販売店は、突然の衝撃、過度の熱、冷気、湿度等の外的な要因による本コンピュータ製品へ生じたいかなる損害に対しても責任を負いません。不正な電圧や不適合なハードウェアやソフトウェアによってもたらされた損失や損害も同様です。

本書の記載情報は、その内容について十分精査していますが、その誤りについて責任を負うものではありません。本書は、事前の通知無く変更することがあります。

# コンテント


**1 はじめに ..... 1-1**

**1.1 その他のマニュアル ..... 1-2**

**1.2 技術サポート ..... 1-3**

**1.3 字体の規則 ..... 1-4**

**2 ストアド・プロシージャ概要 ..... 2-1**

**2.1 ストアド・プロシージャとは ..... 2-1**

**2.2 DBMaster対応のストアド・プロシージャ言語 ..... 2-2**

ESQL/C ストアド・プロシージャ ..... 2-2

JAVA ストアド・プロシージャ ..... 2-2

SQLストアド･プロシージャ ..... 2-3

**3 SQLストアド･プロシージャ基礎 ..... 3-1**

**3.1 SQLストアド・プロシージャの利点 ..... 3-2**

SQLストアド・プロシージャはシステムの実行スピードを上げる ..... 3-2

SQLストアド・プロシージャは再利用できる ..... 3-2

SQLストアド・プロシージャは保存される ..... 3-3

SQLストアド・プロシージャは移行できる ..... 3-3

SQLストアド・プロシージャは簡単にアップグレード 3-3

SQLストアド・プロシージャその他の利点 ..... 3-3

**3.2 SQLストアド・プロシージャ・ツールの使用 ..... 3-4**
JDBA ..... 3-4
dmSQL ..... 3-4
**4 SQLストアド・プロシージャの機能 ..... 4-1**
**5 SQLストアド・プロシージャの構文 ..... 5-1**
**5.1 SQLストアド・プロシージャのアーキテクチャ ..... 5-1**
**5.2 SQLストアド・プロシージャの構文 ..... 5-2**
**5.3 SQLストアド・プロシージャの引数 ..... 5-4**
**5.4 SQLストアド・プロシージャにある変数 ..... 5-7**
**5.5 SQLストアド・プロシージャのカーソル ..... 5-9**
SQLストアド・プロシージャのFETCH文 ..... 5-13
SQLストアド・プロシージャのDECLARE CONDITION5-16
SQLストアド・プロシージャのDECLARE HANDLE .... 5-16
**5.6 SQLストアド・プロシージャに代入文 ..... 5-17**
簡単表現 ..... 5-19
複雑表現 ..... 5-21
**5.7 SQLストアド・プロシージャに制御流れ文 ..... 5-22**
変数関連構文 ..... 5-22
条件文 ..... 5-23
ループ文 ..... 5-30
転送制御文 ..... 5-37
ラベルとSQLストアド・プロシージャの複合文 ..... 5-39
共通変数のスコープチェック ..... 5-41
SQLストアド・プロシージャのSQLCODEとSQLSTATE変数 ..... 5-42
**5.8 SQLストアド・プロシージャの戻り結果セット .... 5-44**
**5.9 動的なSQLストアド・プロシージャ ..... 5-46**
EXECUTE IMMEDIATE文 ..... 5-46
PREPARE文 ..... 5-47
EXECUTE 文 ..... 5-47
DEALLOCATE PREPARE 文 ..... 5-48

動的なカーソルの宣言 ..... 5-49
動的なオープン・カーソル ..... 5-49
**5.10 データ・プロセス .....5-51**
空のSQLストアド・プロシージャを作成する ..... 5-51
INSERT文 ..... 5-52
Select文 ..... 5-53
Create文 ..... 5-54
Drop文 ..... 5-55
SQLストアド・プロシージャ実行を辿る ..... 5-56
**6 SQLストアド・プロシージャ .....6-1**
**6.1 SQLストアド・プロシージャの作成.....6-1**
ファイルからSQLストアド・プロシージャを作成 ..... 6-2
JDBAで作成 ..... 6-2
**6.2 SQLストアド・プロシージャ.....6-6**
SQLストアド・プロシージャ構文を実行する ..... 6-6
トリガーでSQLストアド・プロシージャを実行 ..... 6-9
JDBA ツールで使用 ..... 6-9
**6.3 SQLストアド・プロシージャを削除.....6-10**
dmSQLで削除する ..... 6-10
JDBAで削除する ..... 6-11
**6.4 SQLストアド・プロシージャ情報を取る.....6-12**
**6.5 安全管理 .....6-12**
**6.6 SQLストアド・プロシージャの構成設置.....6-13**
**7 SQLストアド・プロシージャの移行.....7-1**
**7.1 アンロード/ロード・プロシージャ.....7-1**
UNLOAD [PROC | PROCEDURE] ..... 7-1
LOAD [PROC | PROCEDURE] ..... 7-1
**8 SQLストアド・プロシージャの制限.....8-1**

# 1 はじめに

*DBMaster SQL*ストアド・プロシージャ参照編をようこそ。DBMasterは、強力で柔軟なSQLデータベース管理システム(DBMS)であり、会話型の構造的問い合わせ言語(SQL)、Microsoftのオープン・データベース結合(ODBC)互換インタフェース、C言語の埋め込みSQL(ESQL/C)をサポートします。ODBCインタフェースは唯一のオープン・アーキテクチャであり、多種多様なプログラミングツールによる顧客アプリケーションの構築、既存のODBC適合アプリケーションによるデータベース問い合わせを可能にします。

DBMasterは、シングル・ユーザーの個人データベースから企業全体に分散するデータベースまで容易にスケール化することができます。DBMasterの先進的なセキュリティ、整合性、信頼性の機能は、どのようなデータベース構成でも、重要データの安全性を確実に保証します。広範なクロス - プラットフォームをサポートし、現在あるハードウェアの効力を高め、需要の増大に応じてより強力なハードウェアに拡大し、グレードアップすることができます。

DBMaster は、優れたマルチメディア処理機能をもち、あらゆるタイプのマルチメディアデータの格納、探索、検索、操作を可能にします。バイナリラージオブジェクト(BLOB)は、DBMaster の先進的セキュリティ・損傷リカバリ機構を全面的に利用して、マルチメディアデータの整合性を確実にします。ファイルオブジェクト(FO)は、元のアプリケーションで各ファイルを編集する機能を維持しつつ、マルチメディアデータを管理します。

本書は*SQLストアド・プロシージャ*の基礎操作とデータベース管理ガイドについて体系的に説明しています。この参照編は開発者と*DBMaster*データベースの管理者を想定しています。*DBMaster*をあまり知らないユーザーでも、リレーショナルデータベースの知識があれば理解できます。そのためには、ユーザーは*Windows/UNIX*環境での知識が多少必要になります。また、本書は、経験を持つユーザーでも参照できます。

本書は*SQL*ストアド・プロシージャでデータベースを保守する多種多様なコマンドとプロシージャを表示します。*Windows NT* と *Windows 98*での*DBMaster*に適合していますし、これは*UNIX*で全部の機能を実行することもできます。データベース・サンプル部分は本書を一貫して現れます。

*SQL*は二重モード言語です。これはつまり、会話型ツールであり、またデータベースにアクセスするために使用されるからです。通常は会話型*SQL*を指しますが、会話型*SQL*はデータベースにアクセスするため応用プログラムを通じて使用するデータベース言語です。

一般的に、殆ど主要なRDBMSは*SQL*を使用するために自らのユーザーインタフェースを提供しています。例えば、*DBMaster*はdmSQL/Cを提供します。ユーザーはこのツールを通じて直接にsyntax文型を入力してデータベースがアクセスできます。

DBMasterは多くのJavaツールを提供します。詳細についての情報 は“その他のマニュアル”をご参照ください。

## 1.1 その他のマニュアル

DBMasterには、本マニュアル以外にも多くのユーザーガイドや参照編があります。特定のテーマについての詳細は、以下のマニュアルをご覧下さい。

- DBMasterの性能と機能性についての概要は、*DBMaster入門編*をご覧下さい。
- DBMasterの設計、管理、保守についての詳細は、データベース管理者参照編をご覧下さい。

- DBMasterの管理についての詳細は、JServer Managerユーザーガイドをご覧下さい。
- DBMasterの環境設定についての詳細は、JConfiguration Tool参照編をご覧下さい。
- DBMasterの機能についての詳細は、JDBA Toolユーザーガイドをご覧下さい。
- DCI COBOLインターフェースについての詳細は、DCIユーザーガイドをご覧下さい。
- dmSQLのSQL言語についての詳細は、SQLコマンドと関数参照編をご覧下さい。
- dmSQLについての詳細は、dmSQLユーザーガイドをご覧下さい。
- ネイティブODBC APIについての詳細は、ODBCプログラマー参照編をご覧下さい。
- エラーと警告メッセージについての詳細は、エラー・メッセージ参照編をご覧下さい。
- ESQLプログラムについての詳細は、ESQL/Cプログラマー参照編をご覧下さい。
- Javaストアドプロシージャについての詳細は、Javaストアドプロシージャガイドをご覧下さい。

## 1.2 技術サポート

CASEMakerでは評価期間内で、30日間の無償でのe-mail、電話によるサポートを提供しております。ソフトウェアが登録された際の追加の30日間のサポートも含まれます。延長により合計でソフトウェアのサポートを60日間受けることができます。しかしながらバグ報告については無償サポート期間終了後も引き続きe-mailでの無償サポートを提供いたします。

60日を超えた追加サポートはほとんどの製品で有効です。製品価格の20%にてサポートを継続して受けることができます。詳細や価格については sales@casemaker.com までお問い合わせください。

郵便、電話、e-mailにてお問い合わせいただけるお近くのCASEMakerサポートをこちらからご確認いただけます。www.casemaker.com/supportCASEMakerサポートスタッフにお問い合わせいただく前に、FAQデータベースをご覧頂くことを推奨いたします。

トラブルシューティングのお電話の際、郵送、e-mailでのお問い合わせの際に以下の情報をお伝えいただくようお願いいたします。

- 製品名とバージョン番号
- レジストレーション番号
- 登録顧客名と住所
- 購入した代理店、購入場所
- コンピュータのプラットフォーム、システム設定
- エラー発生前の操作詳細
- エラーメッセージがある場合は、メッセージとその番号
- その他関連があると思われる情報

## 1.3 字体の規則

本書は、標準の字体規則を使用しているので、簡単かつ明確に読むことができます。手順、例、コマンドライン規則には別の設定があり、インデンテーションにて使用されます。

| 字体 | 解説 |
| --- | --- |
| *斜体* | 斜体は、ユーザー名や表名のような特定の情報を表します。斜体の文字そのものを入力せず、実際に使用する名前をそこに置き換えてください。斜体は、新しく登場した用語や文字を強調する場合にも使用します。 |
| **太字** | 太字は、ファイル名、データベース名、表名、カラム名、関数名やその他同様なケースに使用します。操作の手順においてメニューのコマンドを強調する場合にも、使用します。 |
| キーワード | 文中で使用するSQL言語のキーワードは、すべて英大文字で表現します。 |
| 小さい英大文字 | 小さい英大文字は、キーボードのキーを示します。２つのキー間のプラス記号 (+) は、最初のキーを押したまま次のキーを押すことを示します。キーの間のコンマ(,) は、最初のキーを放してから次のキーを押すことを示します。 |
| ノート | 重要な情報を意味します。 |
| ➲ プロシージャ | 一連の手順や連続的な事項を表します。ほとんどの作業は、この書式で解説されます。ユーザーが行う論理的な処理の順序です。 |
| ➲ 例 | 解説をよりわかりやすくするために与えられる例です。一般的に画面に表示されるテキストと共に表示されます。 |
| コマンドライン | 画面に表示されるテキストを意味します。この書式は、一般的にdmSQLコマンドやdmconfig.iniファイルの内容の入/出力を表示します。 |

*表 1-1 字体の規則*

# 2 ストアド・プロシージャ概要

本章にESQLストアド・プロシージ、Javaストアド・プロシージャ、SQLストアド・プロシージャについて説明します。

## 2.1 ストアド・プロシージャとは

ストアド・プロシージャはデータベース・サーバーに保持、実行されるプログラムです。一度作成すると、実行形式のデータベース・オブジェクトとしてデータベースに保存されます。何度もSQL文をコンパイル、最適化する必要がない為、コマンドを頻繁に繰り返し実行するケースで効率を上げることができます。ストアド・プロシージャは、実行可能な対話型SQL文としてアプリケーション・プログラム、トリガー、他のストアド・プロシージャで使用することができます。

ストアド・プロシージャを使用することで多くの目的を達成できます。例えば、データベースのパフォーマンス改善、アプリケーションの単純化、データベース・アクセスの制限と監視等です。

ストアド・プロシージャはDBMSにて実行されるので、アプリケーションの呼び出し時間が減少できます。多数のSQL文を実行する必要はなく、サーバー側でストアド・プロシージャを実行するだけです。ネットワーク負荷を軽減させることで性能を上げることができます。

# 2.2 DBMaster対応のストアド・プロシージャ言語

DBMasterは三つのストアド・プロシージャ言語を支持します。ESQL/C, Java、SQLの3つのプロシージャです。

DBMaster 4.3以前のバージョンではESQL/Cストアド・プロシージャのみサポートします。バージョン5. 0以降でESQL/C、Javaストアド・プロシージャをサポートします。DBMaster 5.2で新たにSQLストアド・プロシージャがサポートされます。

本書ではSQL言語でストアド・プロシージャを開発するために必要な情報を提供します。

## ESQL/C ストアド・プロシージャ

DBMaster ESQL/CはSQL文をC言語ソースコードに直接埋め込むことで簡単にプロシージャを作成･実行することが可能です。ESQLストアド･プロシージャはESQL/Cプログラム、Cアプリケーションで許可される関数を実行することができ、C関数とシステムの呼び出しもできます。そのためCコンパイラがストアド･プロシージャに必要です。ユーザーはCアプリケーション･プログラムを書くことにより、ESQLコマンドでDBMSをアクセスします。DBMaster ESQL/Cプリプロセッサはコンパイラに渡すためにSQLコマンドを含むアプリケーション･プログラムを準備します。このプリプロセッサはSQLコメントをC文型に変換し、データベース操作を実行します。

ESQL/Cストアド･プロシージャはCREATE PROCEDURE文、宣言セクション（必要な場合）、コードセクションから構成します。プログラムはホスト変数がない場合、宣言セクションを省略することができます。ESQLについての詳細な情報は“ESQLユーザー･ガイド”をご参考ください。

## JAVA ストアド・プロシージャ

昨今のJavaの普及規模を考えると、開発メンバーがESQLよりJavaに精通しているケースも多いと考えられます。DBMasterはJavaストアドプロシージ

ャをサポートしていますので、Javaプログラマーは使い慣れたJAVA言語でコード化できます。経験豊富なESQL開発者もJavaを使うと、Java言語のメリットを生かして、データベースアプリケーションの機能を向上できます。また、既存のコードを再利用して効率を高めることもできます。

Javaストアド･プロシージャはJava方式として実現されます。ユーザーはJavaストアド・プロシージャをコールすると、JDBC接続を通じでプロシージャ名と指定した引数をDBMSに渡し、DBMSはこのプロシージャを実行して結果を返します。

Javaストアド･プロシージャはESQL/Cと比べよりオープンで、データベースから独立しています。Javaストアド･プロシージャはJAVAの強力で豊富な機能、オブジェクトベースの特性を生かすことができます。Javaストアド･プロシージャについての詳細な情報は“Javaストアド･プロシージャ参照編”をご参考ください。

## SQLストアド･プロシージャ

ストアドプロシージャを作成するのに、ESQLとJAVAを使用するのは、かなりおかしなことです。今日、ストアドプロシージャつまりSQLストアドプロシージャを作成するのに直接SQL構文を使用します。

SQLストアド･プロシージャはSQL文のみで論理的構造を埋め込んで実現可能なストアド･プロシージャです。これはサーバーに保存されるSQL文のセットです。一度作成すると、クライアントは独立してSQL文を保持する必要がなく、代わりにSQLストアド･プロシージャと連携することができます。

# 3 SQLストアド･プロシージャ基礎

SQLストアド・プロシージャは特殊なユーザー定義関数です。ストアド・プロシージャは、一度作成すると実行形式のデータベース・オブジェクトとしてデータベースに保存されます。これは高速クエリ変換、データ更新のため簡単なスクリプトを作成して基礎のレポートを生成できます。またアプリケーションの性能を高めてモジュール化、全てのデータベースのデザインを良くする、及びデータベースの安全などSQLストアド･プロシージャを使用して実現できます。SQL文を何度もコンパイルし最適化しなくて済むので、頻繁に繰り返される仕事の効率を上げることができます。ストアド・プロシージャは、会話型SQL文としてを実行することができて、また、アプリケーション・プログラム、トリガー、他のストアド・プロシージャから呼び出すこともできます。

ストアド・プロシージャはオブジェクトとしてデータベースに保存されているので、データベースの全てのアプリケーションで使用できます。複数のアプリケーションが同じなストアド・プロシージャを使用することができるので、アプリケーションの開発期間を短縮します。

SQLストアド・プロシージャの実現を決定する前、ユーザーはまずSQLストアド・プロシージャとは何か、どう実行するかなどのことをわかる必要があります。以下のセクションからSQLストアド・プロシージャを理解してください。それから、ユーザーは自分のデータベース環境によってこれをいつ、どう使用することが決定します。

- SQLストアド・プロシージャの利点
- SQLストアド・プロシージャのツール
- SQLストアド・プロシージャ関数
- SQLストアド・プロシージャ文型
- SQLストアド・プロシージャの使用
- SQLストアド・プロシージャの移行
- SQLストアド・プロシージャの制限

# 3.1 SQLストアド・プロシージャの利点

データベース或いはデータベース・アプリケーション・アーキテクチャには多数のSQLストアド・プロシージャの有効なアプリケーションがあります。ストアド・プロシージャを利用することによって、データベースのパフォーマンス改善、アプリケーションの単純化、データベース・アクセスの制限と監視等の広範囲の目的を達成することができます。

## SQLストアド・プロシージャはシステムの実行スピードを上げる

コンパイルがないので、SQLストアド・プロシージャは外部言語（例えばESQL/C）を書き入れたプログラムと比べて実行スピードが速くありません。スピードを上げるための主な方法はネットワーク通信をできる限り減少することです。処理タスクのチェック、循環の必要があって多文型ですがユーザー会話型複製タスクがないと、サーバー側のストアド・プロシージャを使用してこれを完成します。そうすると、タスクの実行ステプはサーバーとクライアント側の間に情報通信が少なくなります。

## SQLストアド・プロシージャは再利用できる

SQLストアド・プロシージャはクロス - プラットフォームとWin32/64 、Linux32/64をサポートします。

ホスト言語を変更しても SQL ストアド・プロシージャに影響はありません。これはアプリケーションではなくデータベース・ロジックを使用するからです。ストアド・プロシージャは移行できます。SQL 言語を使用してストアド・プロシージャを作成する場合、ユーザーは操作環境を追加、ライセンスを配置、コンピューターに異ったソフトウェア・パッケージを配置する必要がなく、ストアド・プロシージャが DBMaster サポートできるプラントフォームで実行できます。これは java, ESQL/C 外部言語より SQL 文の優勢です。

## SQLストアド・プロシージャは保存される

ユーザーは銀行トランザクション処理で小切手を取り消す操作を表現することようなプロシージャを書くと、小切手を了解する人はこのプログラムを見つかります。これはソース・コードの形式としてデータベースに保存されて、データとデータのプロセスを関連させます。

## SQLストアド・プロシージャは移行できる

DBMasterはSQL 99標準を完全にサポートして良好な移行性があります。ほかのDBMSもこの標準を支持しますので、他のDBMSで書くコードは簡単にDBMasterまで移行できます。

## SQLストアド・プロシージャは簡単にアップグレード

SQLトアド・プロシージャはDBMaster ESQLデータ型の作成、実行、削除の方式と同じです。クライアント・プログラムは変更する必要がない為、ユーザーのアップグレードに貢献します。

## SQLストアド・プロシージャその他の利点

余分なCコンパイルを使用しないので、コンパイルの影響で問題が起こる可能性がありません、例えばパスのインストール、コンパイルのアンロードなどです。

使用が容易、構文が簡単で明確、強力な構造です。この特性については以下のストアド・プロシージャ構文で説明します。

コンパイルする必要がなく、作成スピードが速いです。

## 3.2 SQLストアド・プロシージャ・ツールの使用

開発ツールでSQLストアド・プロシージャの作成と実行は簡単になく、速くなります。以下のGUIとコマンド・ライン・ツールはストアド・プロシージャを作成、実行するため使用されます。

- JDBA
- dmSQL

### JDBA

JDBA Toolは、クロス・プラットフォームでユーザー仕様のグラフィカル・ユーザー・インターフェース(GUI)です。JDBA Toolを使えば、強力で柔軟なSQLデータベース管理システム、DBMasterのデータベース・オブジェクトを簡単に管理することができます。JDBAToolは煩雑なDBMSとクエリ言語を用いずに理解しやすく、容易に扱えるインターフェースです。SQLの知識がなくともデータベースの機能を使用することができて、JDBAToolの監視機能を使うことでユーザは統計データを取得することもできます。JDBA Toolの使い方についての詳細情報は*JDBA Tool参照編*を参考してください。

### dmSQL

dmSQLはコマンド・ライン・ツールです。SQLストアド・プロシージャを開発する場合、これでユーザーはたくさんの操作をすることができます。例えば：問合せ、変更、テーブルデータのロードと提出、XML関数の使用、Javaルーチンの実行などです。dmSQLコマンド・ライン・ツールの使用方法についての詳細な情報は*dmSQL ツールガイド*をご参考ください。

# 4 SQLストアド・プロシージャの機能

SQLストアド・プロシージャは以下の機能があります：

- ストアド・プロシージャ言語文型は伝統の静的、動的SQL文によって制御流れロジックの実行をサポートする。
- 実現しやすいです。高級、簡単、強大な言語を使用するからです。
- 同等な外部プロシージャより強い依頼性が持つ。
- SQL99 ANSI/ISO/IEC SQL言語標準に則する。
- 入力、出力引数の渡すモードと以下のデータ型をサポート：SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, DATE, TIME, TIMESTAMP, BINARY, CHAR, VARCHAR, NCHAR, NVARCHAR。
- 機能強く、簡単な条件とエラー操作モデルをサポート。
- ユーザーは発信者またはクライアント・アプリケーションにシングル結果セットを返すことができる。
- ユーザーはSQLSTATEとSQLCODE値を変数として容易にアクセスできますが、SQLSTATEとSQLCODEに値を割り当てることができません。
- CALL文の呼び出しをサポート。

- 埋め込みSQLストアド・プロシージャはほかの言語で実現のストアド・プロシージャをコールできる。
- 再帰をサポート。
- トリガーにコールされる。
- ストアド・プロシージャに変数が定義できる：cursor, condition, handle。
- カーソル操作をサポート：OPEN, FETCH, CLOSE。
- 変数に値を割り当てるためSET構文をサポート。
- 制御流れ文をサポート：IF, CASE, WHILE, LOOP, FOR, REPEAT
- TRACEをサポート、使用方法はESQLと同じです。

SQLストアド・プロシージャの機能は以上だけではありません。最良方法でストアド・プロシージャを実現する場合、これはデータベース・アーキテクチャ、データベース・アプリケーション・デザインとデータベース・システム性能などの方面に重要な作用があります。

# 5 SQLストアド・プロシージャの構文

SQLストアド・プロシージャ言語(SQL SP)はANSI SQL99言語標準に一致しています。これはSQL文のセットです。SQLコマンドについての詳細な情報は*SQL文と関数参照編*を参考してください。伝統なSQL間合せと操作をめぐって、制御流れロジックを実現するため必要な構成を提供します。

SQLストアド・プロシージャの構文が簡単です。変数支持、条件文、循環文、転換制御文、エラー管理文、結果セット操作文などを含みます。

## 5.1 SQLストアド・プロシージャのアーキテクチャ

SQLストアド・プロシージャは多くのロジック部分とフォーマットから構成するので身につけやすいです。目的はルーチンのデザインと意味を簡略化するものです。

SQLストアド・プロシージャのコアは複合文です。この複合文はキーワードBEGINから始めてENDまで終了します。以下はSQLストアド・プロシージャ構文の構成：

```
BEGIN                          #block headerブロック・ヘッダー
Variable declarations
Condition declarations
```

```
Cursor declarations
Condition handler declarations
Assignment ,flow of control. SQL statements and other compound
statements
END;                    #block endブロック・終了
```

SQLストアド・プロシージャは一つ或いは一つ以上の選択可能な複合文（またはブロック）から構成します。このブロックはシングルなSQLストアド・プロシージャにネストし連続に引用できます。ブロックは選択可能変数、条件、処理宣言でオーダーがあります。SQL制御文、ほかのSQL文及びカーソル宣言のプロシージャロジックを実現する前、この内容は説明しなければなりません。カーソルはSQLストアド・プロシージャ体を含むSQL文セットにどこでも宣言できます。

## 5.2 SQLストアド・プロシージャの構文

以下の命名規則はSQLストアド・プロシージャの引数名と変数名に適用します。

- SQLストアド・プロシージャ名の長さは128文字以下。
- SQLストアド・プロシージャ名には、アンダースコア(_)を含む英数字を使用します。
- 文字の並びに制限はありません。
- SQLストアド・プロシージャ名は大文字と小文字を識別しません。
- SQLストアド・プロシージャ名はユニークです。

➲ 構文：SQLストアド・プロシージャの構文

```
<SQL stored procedure syntax> ::=
CREATE PROCEDURE <procedure_name>
[< procedure_parameters > [ { <comma> < procedure_parameters
> }... ]]
[RETURNS STATUS]
LANGUAGE SQL
```

```
BEGIN
  [stored_procedure_statement]
END;
<procedure_parameters> ::=
[IN | OUT | INPUT | OUTPUT] <parameter_name> <data_type>
```

以上はSQLストアド・プロシージャの構文の整体フレームです。完全なSQLストアド・プロシージャはブロック・ヘッダーとブロック終了があります。ブロック・ヘッダーは名とストアド・プロシージャの引数を説明するため使用されます。ブロック終了は終了サインだけです。例えば：

```
CREATE PROCEDURE project_name.module_name.sp_name [ arg_list ]
LANGUAGE SQL
BEGIN
----------------->Block header
[ declare_list ]
[ statement_list ]
----------------> Block end
END;
```

Arg_listは引数リスト、宣言プロセスの入力引数と出力引数です。NULLなら引数を返しません。

```
<Arg_list> ::=
        [ Variable declarations ]
        [ Condition declarations ]
        [ Cursor declarations ]
        [ Condition handler declarations ]
```

Declare_listは宣言プロセスの変数宣言リストです。

```
<declare_list> ::=
        [ Variable declarations ]
        [ Condition declarations ]
        [ Cursor declarations ]
        [ Condition handler declarations ]
```

Statement_listはプロセス制御文です。これは様々な制御文とSQL文を構成してプロセスにデータベースの操作を完成します。

```
<statement_list> ::=
        [ sp_block ]
        [ procedure_control_statement ]
```

# 5.3 SQLストアド・プロシージャの引数

SQLストアド・プロシージャ支持のSQL引数はSQL値を進\出プロシージャに渡すため使用されます。

実行ロジックは特別な入力または入力スカラー・セットに条件を制限する場合、一つまたは多くの出力スカラー値を返す必要がありますが、結果セットを返したくない場合、SQLストアド・プロシージャの引数は非常に重要です。

SQLストアド・プロシージャをデザイン、作成する時、SQLストアド・プロシージャ中の引数の特性と制限についてよく理解してください。

- DBMasterはSQLストアド・プロシージャ中にたくさんの入力\出力引数が選択可能で使用できます。CREATE PROCEDURE文のルーチンサインの部分キーワードIN/INPUT, OUT/OUTPUTは引数のモードと用途を表現します。IN/INPUT, OUT/OUTPUT引数は値を通じて渡します。
- プロシージャに複数の引数を指定すると、この引数はユニークな名がなければなりません。
- プロシージャの中に宣言変数と引数は同じ名を使用する場合、この変数はプロシージャの中にラベルのブロックネストに宣言しなければなりません。そうしなければDBMasterはほかの不明な名称を使います。
- SQLストアド・プロシージャは支持できるデータ型：SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, DATE, TIME, TIMESTAMP, BINARY, CHAR, VARCHAR, NCHAR, NVARCHAR。

➲ 構文：SQLストアド・プロシージャ引数の構文

```
<procedure_parameters> ::=
```

```
[IN | OUT| INPUT | OUTPUT] <parameter_name> <data_type>
```

この句は選択可能です。SQLストアド・プロシージャは普通の関数と同じように引数を通じてデータを渡します。引数の後に'in/input'を追加してメイン関数はデータがストアド・プロシージャまで渡せます。SQLストアド・プロシージャも引数の後に'out/output'追加してデータをメイン関数まで渡します。

➲ 例 **1**：以下SQLストアド・プロシージャmyparamsはIN/INPUTと OUT/OUTPUT引数モードの用法を説明します：

```
CREATE PROCEDURE myparams (IN p1 INT, OUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
SET p2 = p1 + 1;
SET p3 = 2 * p1;
END;
```

➲ 例 **2**：引数空白のSQLストアド・プロシージャ：

```
CREATE PROCEDURE CRETB
LANGUAGE SQL
BEGIN
        CREATE TABLE TB_1(V1 INT,V2 BIGINT,V3 SMALLINT,V4 CHAR(10),
                          V5 VARCHAR(20),V6 FLOAT,V7 DOUBLE,
                          V8 BINARY,V9 DECIMAL,V10 TIME,V11 DATE,
                          V12 TIMESTAMP);
END;
```

SQLストアド・プロシージャCRETBの引数は空白です。表TB_1を作成するためです。

➲ 例 **3**：入力引数があるSQLストアド・プロシージャ：

```
################################################################################
#
#       Module Name = INS.SP
#       Purpose = Store Procedure testing program
#               1. Test IN parameter store procedure
#               2. Test all type which can use in Store Procedure
```

```
#       Function = 1. Create table INS
#       Use Database: "DBSAMPLE5" "SYSADM"  ""
#            table : INS(V1 int,V2 BIGINT,V3 smallint,V4 INT,
#                       V5 float,V6 DOUBLE, V7 DECIMAL(20,4),
#                        V8 binary(20), V9 CHAR (20), V10 VARCHAR (20),
#                        V11 NCHAR (40),V12 NVARCHAR (40),
#                         V13 DATE, V14 TIME, V15 TIMESTAMP)
##############################################################################
#
CREATE PROCEDURE INS(IN V1 int, IN V2 BIGINT, IN V3 smallint, IN V4 INT,
                     IN V5 FLOAT, IN V6 DOUBLE,IN V7 DECIMAL(20,4),
                     IN V8 BINARY(20), IN V9 CHAR(20),
                     IN V10 VARCHAR(20),IN V11 NCHAR(40),
                     IN V12 NVARCHAR(40), IN V13 DATE,
                     IN V14 TIME, IN V15 TIMESTAMP)
LANGUAGE SQL
BEGIN
        INSERT INTO ins VALUES(V1, V2,V3,V4,V5,V6,V7, V8,
V9,V10,V11,V12,V13,V14,V15);
END;
```

SQLストアド・プロシージャのINS全ての引数タイプはINPUTです。INSに入力引数のデータ型はSQLストアド・プロシージャが支持できる全部のデータ型です。以上の機能は入力データを既存のINS表に挿入します。

**NOTE** *本書に"#"から始めの行はコメントです。*

➲ 例 4 : 入力/出力引数をもつSQLストアド・プロシージャ :

```
##############################################################################
####
#       Module Name = DRPTB.SP
#       Purpose  = Store Procedure testing program
#                  1. Test DROP TABLE in Stored Procedure
#       Function = 1. create table TB_1
#       Use Database : DBSAMPLE5
#       table : TB_1(V1 INT, V2 BIGINT, V3 SMALLINT, V4 CHAR(10),
#                    V5 VARCHAR(20),V6 FLOAT,V7 DOUBLE,V8 BINARY,
```

```
#                    V9 DECIMAL,V10 TIME,V11 DATE,V12 TIMESTAMP,)
################################################################################
###
CREATE PROCEDURE DRPTB(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP TABLE' THEN
                DROP TABLE TB_1;
                SET WORNING ='NORMAL! TABLE TB_1 DROPED';
        ELSEIF V1 = 'CREATE TABLE' THEN
                CALL CRETB;
                SET WORNING ='NORMAL! CREATE A NEW TABLE NAMED TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

以上のSQLストアド・プロシージャDRPTBは入力引数も出力引数もあります。プロセス体に入力引数を判断します。若し入力引数は'DROP TABLE'なら、SQLストアド・プロシージャはDROP TABLEコマンドを実行します、また出力引数を'NORMAL! TABLE TB_1 DROPED'に設置します。そうすると、入力引数によって出力引数を制御することができます。

## 5.4 SQLストアド・プロシージャにある変数

SQLストアド・プロシージャ支持のローカル変数はユーザーがSQL値を配置、復旧することが許せます。このSQL値はSQLストアド・プロシージャロジックから支持されます。

SQLストアド・プロシージャの変数はDECLARE文を通じて定義されます。DECLARE文はルーチンに以下の項目を定義します：ローカル変数、条件とハンドル、カーソル。DECLAREはBEGIN ... END複合文の内部だけに位置できます、他の文の前になります。宣言はオーダーに従わなければなりません：カーソルは宣言ハンドルの前に宣言するべきです、変数と条件は宣言ハンドル或いは宣言カーソルの前に宣言されます。

➲ 構文: SQLストアド・プロシージャDECLARE変数の構文:

```
<sp_declare_main> ::=
DECLARE <variable_name> [ { <comma> <variable_name> }... ] <data_type>
[DEFAULT <default_value>]
```

この文はローカル変数を宣言するため使用されます。初期値（ディフォルト句を含む）を提供するため、この値は表現式として定義できます。ディフォルト句がないと、これを定数としなくてもいいです。初期値はnullです。

ローカル変数の範囲はbegin…endブロックの間に宣言されます。同じ名で宣言変数のブロックを除いて、宣言ブロックにの埋め込みブロックはこの宣言変数が参照できます。

➲ 例 1:ディフォルト値がなくデータを宣言：

```
CREATE PROCEDURE DECALRES
LANGUAGE SQL
BEGIN
        declare v1 int;
        declare v2 bigint;
        declare v3 char(10);
        declare v4 varchar(10);
        declare v5 integer;
        declare v6 time;
        declare v7 date;
        declare v8 timestamp;
        declare v9 nchar(20);
        declare v10 binary(20);
        declare v11 decimal(4,8);
        declare v12 double;
        declare v13 float;
END;
```

➲ 例 2:ディフォルト値でデータを宣言：

```
CREATE PROCEDURE DECLARES
LANGUAGE SQL
```

```
BEGIN
        declare v1 int default 50;
       declare v2 bigint default 7396;
        declare v3 char(10) default 'char';
        declare v4 varchar(10) default 'varchar';
        declare v5 integer default 20;
        declare v6 time default '11:11:11';
        declare v7 date default '2008-08-08';
        declare v8 timestamp default '2008-08-08 11:11:11';
        declare v9 nchar(20) default 'nchar';
        declare v10 binary(20) default 'binary';
        declare v11 decimal(4,8) default 1.123;
        declare v12 double default 123;
        declare v13 float default 6546;
END;
```

## 5.5 SQLストアド・プロシージャのカーソル

SQLストアド・プロシージャのカーソルは結果セット（データ行のセット）を定義して、また基礎行に基づいてシングル行に複雑なロジックを実行します。同じ方法でSQLストアド・プロシージャは結果セットも定義します。これを直接呼出し側に或いはクライアント・アプリケーションに返します。

カーソルは行セットにシングルの行に指すポインターとします。同一タイムだけで一行が参照できますが、必要があると結果セットにほかの行に移動できます。

DECLARE CURSOR文はカーソルを定義します。会話型SQL文は提供するインタフェースが会話型実行のようですが、この文はアプリケーション・プログラムに埋め込むことしかできません。これは実行可能な文ではないので、動的に実行できません。

SQLストアド・プロシージャにカーソルを使用すると、ユーザーは以下のようにしてください:

1. カーソルを宣言して結果セットを定義する。

2. カーソルを開いて結果セットを作成する。
3. 必要によってカーソルからローカルまで変数を取ります、一回は一行です。
4. 終了した後、カーソルを閉じます。

カーソルを使用すると以下のSQL文を使用しなければなりません。

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

➲ 構文: DECLARE CURSOR文の構文：

```
<sp_declare_main> ::=

DECLARE <cursor_name> CURSOR [[NO] SCROLL [WITH RETURN] FOR { CALL
<procedure_name> | <select_statement> }
```

***cursor_name***

ソース・プログラムを実行するとき、カーソル名を指定してください。この名はソース・プログラムに宣言の別のカーソル名と違います。使用する前、このカーソルを必ずスタートしなければなりません。

*[NO] SCROLL*

もしSCROLLが使用されないあるいはNO SCROLL句がDECLARE CURSORステートメントの定義で使用された場合は、FETCH NEXTのみ許可されます。

SCROLL句を使用すると、すべてのFETCH操作が許可されます。デフォルトはnon-scrollableカーソルとなります。

*WITH RETURN*

SQLストアド・プロシージャにWITH RETURN句で宣言のカーソルはSQLストアド・プロシージャを終了するまでまた開いています。これはSQLストアド・プロシージャから定義する結果セットと見なします。SQLストアド・プロシージャを終了するまで、ほかの開くカーソルは全部閉じます。外部ストアド・プロシージャ

(LANGUAGE SQLで定義するのではない)に全てのカーソルの初期値はWITH RETURN TO CALLになります。それゆえ、プロシージャが終了する場合、全部のカーソルは結果セットと考えます。

***Select_statement***

カーソルのSELECT文を識別します。*select-statement*は引数マーカーを含みませんが、ホスト変数の引用を含みます。ホスト変数の宣言はDECLARE CURSOR文の前にあります。

***call***

カーソルを指定して呼出し側に結果セットを返します。例えば、この呼出し側は他のストアドプロシージャなら、結果セットはこのストアドプロシージャまでに返します。呼出し側はクライアント･アプリケーションなら、結果セットはクライアント･アプリケーションまでに返します。

➲ 例 1 : select_statementsコマンドがあるカーソル :

```
CREATE PROCEDURE getbd_sql(IN name char(12), OUT bd DATE)
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR FOR SELECT birthday FROM birthd WHERE NAME =
name;


        OPEN cur;
        FETCH cur INTO bd;
        CLOSE cur;
END;
```

➲ 例 2:結果セットがあるカーソル :

```
CREATE PROCEDURE t42_sql(argn TIMESTAMP)
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select n,ts from t2
where ts < argn;
                OPEN cur;
END;
```

➲ 例 3:呼出しがあるカーソル：

```
CREATE PROCEDURE call_test
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
                OPEN cur;
END;
```

➲ 例 4:call4はcall_testを使用して結果セットを宣言する：

```
CREATE PROCEDURE call4
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR call call_test;
                OPEN cur;
END;
```

➲ 例 5: 下記の例は、SQLストアドプロシージャ内で読取専用カーソルの基本的な使用例です:

```
CREATE PROCEDURE sum_salaries(OUT sum INTEGER)
LANGUAGE SQL
BEGIN
DECLARE p_sum INTEGER;
DECLARE p_sal INTEGER;
DECLARE c CURSOR FOR SELECT SALARY FROM EMPLOYEE;
SET p_sum = 0;
OPEN c;
FETCH FROM c INTO p_sal;
WHILE(SQLCODE = 0)
                DO
                SET p_sum = p_sum + p_sal;
                FETCH FROM c INTO p_sal;
        END WHILE;
CLOSE c1;
SET sum = p_sum;
END;
```

## SQLストアド･プロシージャのFETCH文

FETCH文はカーソルを結果表の次の行に定位できて、またはこの行の値をホスト変数に分配します。会話型SQL文は提供するインタフェースが会話型実行のようですが、この文はアプリケーションプログラムに埋め込むしかできません。これは実行可能な文ではないので、動的に実行できません。

➲ 構文：FETCH文の構文：

```
<FETCH statement main> ::=
FETCH [NEXT|PRIOR|FIRST|LAST] FROM <cursor_name> INTO <fetch_target_arg>
fetch_target_arg ::=
<variable_name> [ { <comma> <variable_name> }... ]
```

***cursor-name***

fetch操作のカーソルを識別します。cursor-nameは宣言のカーソルを識別しなくてはいけません。ソース･プログラムにDECLARE CURSOR文はFETCH文の前に位置しなければなりません。FETCH文を実行する時、カーソルはスタート状態になります。

***NEXT***

結果セットに次の行を返します。NEXTはfetch句のデフォルト･カーソルです。

***LAST***

結果セットに、LASTコマンドはカーソルを最後の行に移動してまたは最後の行に返すため使用されます。

***FIRST***

結果セットに、FIRSTコマンドはカーソルを最初の行に移動してまたは最初の行を返すため使用されます。

***PRIOR***

PRIORコマンドは結果セットに前の行を返します。

*ABSOLUTE n*

結果セットに一番目の行からｎ番目の行まで返します。

*RELATIVE n*

結果セットに既存の行からｎ番目の行までを返します。

*INTO fetch_target_arg*

一つ或いは複数の変数を識別する場合、この変数は宣言変数の規則に則して説明されます。結果行の一番目の値はリストの一番目の変数を指定します、二番目の値は二番目の変数を指定します…それからこのようにします。

➲ 例 1：fetchの基本使用：

```
CREATE PROCEDURE t43_sql(i SMALLINT, OUT ts1 TIMESTAMP)
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR FOR select TS from t2 where n = i;
        OPEN cur;
        FETCH cur INTO ts1;
        CLOSE cur;
END;
```

➲ 例 2：last, next, priorがあるfetch文：

```
Table used in example procedure
dmSQL> SELECT * FROM TB_1;
         C1
====================
FIRST
NEXT
PRIOR
LAST


CREATE PROCEDURE FETCH_TEST(OUT FHTY_1 CHAR(20),OUT FHTY_2 CHAR(20),OUT
FHTY_3 CHAR(20),OUT FHTY_4 CHAR(20))
LANGUAGE SQL
```

```
BEGIN
        DECLARE V1 VARCHAR(20);
        DECLARE V2 VARCHAR(20);
        DECLARE V3 VARCHAR(20);
        DECLARE V4 VARCHAR(20);
        DECLARE CUR CURSOR FOR SELECT * FROM TB_1;
        OPEN CUR;
        FETCH FIRST FROM CUR INTO V1;
                IF V1 = 'FIRST' THEN
                        SET FHTY_1 = V1;
                ELSE
                        SET FHTY_1 = 'NULL';
                END IF;
        FETCH NEXT FROM CUR INTO V2;
                IF V2 = 'NEXT' THEN
                        SET FHTY_2 = V2;
                ELSE
                        SET FHTY_2 = 'NULL';
                END IF;
        FETCH NEXT FROM CUR INTO V3;
        FETCH PRIOR FROM CUR INTO V3;
                IF V3 = 'PRIOR' THEN
                        SET FHTY_3 = V3;
                ELSE
                        SET FHTY_3 = 'NULL';
                END IF;
        FETCH LAST FROM CUR INTO V4;
                IF V4 = 'LAST' THEN
                        SET FHTY_4 = V4;
                ELSE
                        SET FHTY_4 = 'NULL';
                END IF;
        CLOSE CUR;
END;
```

➲ 構文：fetch_testをコールする結果は：

```
dmSQL> CALL FETCH_TEST(?,?,?,?);
FHTY_1                          : FIRST
FHTY_2                          : NEXT
FHTY_3                          : NULL
FHTY_4                          : LAST
```

## SQLストアド・プロシージャのDECLARE CONDITION

ある条件は特別に処理を行う必要があるかもしれません。このような条件はエラーを起こし、しかもルーチンに制御流れも生成します。

➲ 構文：DECLARE CONDITION文の構文：

```
<condition declaration> ::=
DECLARE <condition name> CONDITION  FOR <sqlstate value>
<sqlstate value> ::=
SQLSTATE [ VALUE ] <character string literal>
```

➲ 例：

```
DECLARE con1 CONDITION FOR SQLSTATE '23000';
DECLARE con2 CONDITION FOR SQLSTATE VALUE '23001';
```

## SQLストアド・プロシージャのDECLARE HANDLE

モジュール或いは複合文に異常ハンドルまたは完全条件ハンドルを関連します。

ユーザーはDECLARE HANDLERステートメントの再定義を行うことができます。その場合、以前定義したDECLARE HANDLERステートメントはリセットされることになります。

➲ 構文：DECLARE HANDLE文の構文：

```
<handler declaration> ::=
DECLARE <handler type> HANDLER FOR <condition value list> <handler
action>
<handler type> ::=
```

```
CONTINUE
| EXIT
<handler action> ::= <SQL procedure statement>
<condition value list> ::= <condition value> [ { <comma> <condition
value> }... ]
<condition value> ::=
<sqlstate value>
| <condition name>
| SQLEXCEPTION
| SQLWARNING
| NOT FOUND
```

➲ 例 1:

```
create procedure sphdler
language sql
begin
        declare continue handler for sqlexception;
        drop table tb_1;
        declare exit handler for sqlexception;
        drop tabel tb_1;
end
```

➲ 例 2:

```
DECLARE val INT;
DECLARE con1 CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR con1 SET val = 100;
```

## 5.6 SQLストアド･プロシージャに代入文

代入文はSQL変数またはSQL引数に値を割り当てるため使用されます。変数を宣言する場合、SET文、CURSOR FOR SELECT FROM文を通じて或いは初期値として変数に値を割り当てます。文字、表現式、クエリ結果、特別レジスターの値みんなを変数に割り当てることができます。変数の値はSQLストアド･プロシージャ、SQLストアド･プロシージャにほかの変数に

分配できて、またSQLストアド･プロシージャ文を実行するルーチンに引数として参照されます。

SET変数文はローカル変数、出力引数及び新規なトランザクション変数に値を割り当てます。トランザクションはこれを制御します。SET代入文は簡単な表現式と複雑な表現式を引きうけます。

**NOTE** *変数に値を割り当てると、代入長さは1024未満になります。*

➲ 例 1：以下のは変数値を割り当て、回収するのを表示されます：

```
CREATE PROCEDURE proc_vars()
LANGUAGE SQL
BEGIN
DECLARE v_rcount INTEGER;
DECLARE v_max DECIMAL (9,2);
DECLARE v_adate, v_another DATE;
DECLARE v_total INTEGER DEFAULT 0;            # (1)
DECLARE CUR CURSOR FOR SELECT * FROM TB_1;    # (2)
SET v_total = v_total + 1;                    # (3)
END;
```

変数を宣言する時、ユーザーはDEFAULT句でライン(1)のようにデフォルト値が指定できます。SET文はシングルな変数値を割り当てるのはライン(2)に表します。ライン(3)のようにユーザーはCURSOR FOR SELECT FROMを使用して値が指定できます。

**NOTE** *DECLARE must be in front of the SET definition, or the preprocessor compile error.*
*DECLAREはSET定義の前に表示しなければなりません。そうしないと、プリプロセッサのコンパイルはエラーが起こします。*

➲ 例 2：以下の例は**SET**代入文に変数を割り当て、切り捨てる方法を表示されます：

SET 代入文に、double と integer データ型の範囲を超えた値に割り当てる変数は対応のデータ型なら引きうけます。そうではなければ、double と integer の範囲に基づいて切り捨てます。

```
DECLARE d1, d2 DECIMAL(20, 10);
DECLARE b1, b2 BIGINT;
```

```
SET d1 = 1234.43534534531; #exceeding the double data type range
SET d2 = 1234.43534534532;


SET b1 = 1234454654645645651;  # exceeding the integer data type range
SET b2 = 1234454654645645652;
```

以上のSET等式の比較結果はd1がd2より小さい、b1がb2より小さいです。

以下のようなIF文にd1とd2、b1とb2が同等です。超えた範囲の部分は切り捨てられます。

```
IF 1234.43534534531 = 1234.43534534532 THEN ……..    #as double data type intercepted   doubleデータ型として切り捨てられます
IF 1234454654645645651 = 1234454654645645652 THEN …….#as integer data type intercepted. integerデータ型として切り捨てられます
```

## 簡単表現

簡単な表現は以下のタイプがあります。数字タイプ：INTEGER, BIGINT, SMALLINT, DOUBLE, FLOAT, DECIMAL；文字データ・タイプ：CHAR, NCHAR, VARCHAR, NVARCHAR；BINARYデータ・タイプとtimestampタイプ：DATE, TIME, TIMESTAMP。

簡単な表現は‘+’、‘-‘、‘*’、‘/’、変数、定数、値、ストリングを含みます。簡単表現の実行効率は複雑表現より高いですので、複数循環文に適用して実行スピードを上げます。SQL関数についての詳細な情報は“コマンドと関数参照編”をご参照ください。

➲ 構文：set文の構文：

```
SET <variable_name> ::= <variable_name> [{ <+ | - | * | / | || > <variable_name> }...]
```

➲ 例 1：

```
CREATE PROCEDURE SETTS(out rc1 int,out rc2 int, out rc3 char(10) )
LANGUAGE SQL
BEGIN
        DECLARE d1 INT DEFAULT 2;
        DECLARE d2,d3 INT DEFAULT 2;
```

```
        DECLARE c1,c2 char(10) DEFAULT '12345';
        SET d1 = 1 + d2 * d3*100/10;
        SET d2 = 6;
        SET c2 = c1;
        SET d3 = d1+d2;
        SET rc1 = d1;
        SET rc2 = d2-3;
        SET rc3 = c2;
END;
```

➲ 例 2:

```
CREATE PROCEDURE OUTPUTS(OUTPUT V1 INT,OUTPUT V2 BIGINT,
                         OUTPUT V3 FLOAT,OUTPUT V4 DOUBLE,
                         OUTPUT V5 DECIMAL(8,4),
                         OUTPUT V6 BINARY(20),OUTPUT V7 CHAR(20),
                         OUTPUT V8 VARCHAR(20),OUTPUT V9 NCHAR(40),
                         OUTPUT V10 NVARCHAR(40),OUTPUT V11 DATE,
                         OUTPUT V12 TIME,OUTPUT V13 TIMESTAMP)
LANGUAGE SQL
BEGIN
        SET V1 = 1;
        SET V2 = 7396;
        SET V3 = 2.2;
        SET V4 = 3.3;
        SET V5 = 4.4;
        SET V6 = 'ASSIGNMENT';
        SET V7 = 'CHAR';
        SET V8 = 'VARCHAR';
        SET V9 = 'NCHAR';
        SET V10 = 'NVARCHAR';
        SET V11 = '2008-08-08';
        SET V12 = '11:11:11';
        SET V13 = '2008-08-08 11:11:11';
END;
```

## 複雑表現

複雑表現は簡単表現に全ての代入値を含むだけではなく、SQL関数も含みます。例えば組み込み関数とユーザー定義関数。

組み込み関数は結果セットにカラム或いは制限行のカラムに使用されます。DBMaster組み込み関数についての詳細な情報は*SQL文と関数参照編*の第4章をご参考ください。

DBMasterでは、各ユーザー定義関数(UDF)を作成することができます。一旦UDFを作成したら、新しいDBMasterの組み込み関数のように扱われます。ユーザー定義関数についての詳細情報はデータベース管理者参照編の第13章をご参考ください。

➲ 例：set文の使用：

```
################################################################################
####
#       Module Name = SETTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "SET" in Store Procedure
#       Function = 1. decalare d1 d2 d3 c1 c2 for pass parameter
#       Use Database : DBSAMPLE5
################################################################################
###
CREATE PROCEDURE SETTS(out rc1 int,out rc2 int, out rc3 char(10) )
LANGUAGE SQL
BEGIN
        DECLARE d1 INT DEFAULT 2;
        DECLARE d2,d3 INT DEFAULT 2;
        DECLARE c1,c2 char(10) DEFAULT '12345';

        SET d1 = 1 + d2 * d3*100/10;
        SET d2 = 6;
        SET c2 = c1;
        SET d3 = d1+d2;
        SET rc1 = d1;
```

```
        SET rc2 = d2-3;
        SET rc3 = c2;
END;
```

# 5.7 SQLストアド･プロシージャに制御流れ文

連続実行は基本的なプログラム実行パスです。この方法でプログラムはコードの一番目のラインを実行し始め、それから次へ、実行コードの最後文までです。でもこの方法は簡単なタスクに適しますが、一つ状況だけ取り扱えるので実用性がありません。変更な環境を応対することはプログラムに必要です。制御コードの実行パスを通じて、特別なコードは複数な状況を有効に処理することができます。

SQL制御文から提供の変数支持とフロー制御文は実行順序を制御するため使用されます。IFとCASEのような文は条件付き、SQL制御文のブロックを実行します。WHILEとREPEATのようなほかの文はタスクが終了するまで重複文を実行するため使用されます。

たくさんの制御文のタイプがありますが、以下の種類別にします：

- 変数関連文
- 条件文
- ループ文
- 転送制御文
- ラベルとSQLストアド･プロシージャ複合文

## 変数関連構文

変数関連SQL文は変数を宣言、変数に値を割り当てるため使用されます。ここには変数関連構文の幾つかのタイプがあります：

- SQLストアド･プロシージャにのDECLARE <variable>文
- SQLストアド･プロシージャにのDECLARE <condition>文

- SQLストアド・プロシージャにのDECLARE <condition handler>文
- SQLストアド・プロシージャにのECLARE CURSOR

これらの文は別のタイプのSQL制御文に必要な支持を提供します。またSQL文は変数値に有効です。

## 条件文

条件文は条件状態に基づいてどんなロジックを実行するか決定します。SQLストアド・プロシージャに二つ種類の条件文をサポートします。

- CASE
- IF

これらの文はほぼ同じですが、IF文はCASE文の拡張文です。

### SQLストアド・プロシージャにCASE文

CASE文は適合な条件状態に基づいて条件付きでロジックを入力します。ここには二つのCASE文があります：

- 簡単なcase文：文字値に基づいてロジックを入力します
- 探査case文：表現値に基づいてロジックを入力します

CASE文のWHEN句は制御流れの値を適する時間を指定します。

CASE文はストアド・ルーチンのため複合条件構成が実現できます。search_conditionの値は真なら、対応的なSQL文が実行されます。かなう条件を探査できなかったら、ELSE句のSQL文が実行されます。statement_listは一つ或いは一つ以上の文から構成します。

➲ 構文：CASE文の構文：

```
<case_statement_main> ::=
CASE
<variable_case> | <condition_case>
ELSE
<sp_statement_main>
```

```
END CASE
<variable_case> ::= <variable_name> <variable_case_list>
<variable_case_list> ::= WHEN <var_value> THEN <sp_statement_main>
                         [ { < ; > WHEN <var_value> THEN
                        <sp_statement_main> }... ]
<condition_case> ::= WHEN <condition> THEN <sp_statement_main>
                    [ { < ; > WHEN <condition> THEN
                   <sp_statement_main> }... ]
```

➲ 例 1：以下はwhen句があるCASE文のSQLストアド・プロシージャ:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept char(3))
LANGUAGE SQL
BEGIN
DECLARE v_workdept CHAR(3);
SET v_workdept = p_workdept;
CASE v_workdept
WHEN 'A00' THEN
UPDATE department SET deptname = 'D1';
WHEN 'B01' THEN
UPDATE department SET deptname = 'D2';
ELSE
UPDATE department SET deptname = 'D3';
END CASE
END;
```

➲ 例 2:以下はwhen句がある探査CASE文のSQLストアド・プロシージャ:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept char(3))
LANGUAGE SQL
BEGIN
DECLARE v_workdept CHAR(3);
SET v_workdept = p_workdept;
CASE
WHEN v_workdept = 'A00' THEN
UPDATE department SET deptname = 'D1';
WHEN v_workdept = 'B01' THEN
UPDATE department SET deptname = 'D2';
```

```
ELSE
UPDATE department SET deptname = 'D3';
END CASE
END;
```

以上の例示は論理上に等価です。重要なのはsearched-case-statement-when-clauseがあるCASE文の機能が非常に強大です。全ての支持可能なSQL表現はここで使用できます。これらの表現は関連の変数、引数などを含みます。

➲ 例 3：caseの用法：

```
################################################################################
#
#       Module Name = CASE_TEST.SP
#       Purpose  = Store Procedure testing program
#                       1. Test keyword "CASE" in Store Procedure
#       Use Database : "DBSAMPLE5" "SYSADM"  ""
################################################################################
#
CREATE PROCEDURE CASE_TEST (IN INVAL INT, OUT outval1 INT, OUT outval2
INT)
LANGUAGE SQL
BEGIN
        DECLARE VAL INT;
        SET VAL = INVAL;
        CASE VAL
                WHEN 1 THEN
                        SET OUTVAL1 = 1;
                WHEN 2 THEN
                        SET OUTVAL1 = 2;
                WHEN 3 THEN
                        SET OUTVAL1 = 3;
                ELSE
                        SET OUTVAL1 = 10;
        END CASE;
        CASE
```

```
                WHEN VAL = 1 THEN
                        SET OUTVAL2 = 11;
                WHEN VAL = 2 THEN
                        SET OUTVAL2 = 22;
                WHEN VAL = 3 THEN
                        SET OUTVAL2 = 33;
                ELSE
                        SET OUTVAL2 = 100;
        END CASE;
END;
```

## SQLストアド･プロシージャにIF文

IF文は条件の状態に基づいて条件付きでロジックを入力します。IF文とwhen句があるCASE文は論理上に等価です。

IF文はELSE IF句とディフォルトELSE句を選択して使用できます。END IF句は文の終了を識別します。

IF文は基本条件の構成が実現できます。search_conditionの値は真なら、対応的なSQL文が実行されます。かなう条件を探査できなかったら、ELSE句のSQL文が実行されます。各なstatement_listは一つ或いは一つ以上の文から構成します。

➲ 構文： **IF文の構文**：

```
<IF statement main> ::=
IF <condition_value> THEN <sp_statement_main>
[ELSEIF <condition_value> THEN <sp_statement_main>]
[ELSE <sp_statement_main>]
END IF
```

➲ 例 1：以下の例はIF..CALL文を含むSQLストアド･プロシージャ：

```
#############################################################################
#
#       Module Name = IF_CALL.SP
#       Purpose = Store Procedure testing program
#                   1. Test keyword "IF" and "CALL" in Store Procedure
```

```
#       Function = 1. Store Procedure: CRETB CASE_TEST_2 INS
#       Use Database: "DBSAMPLE5" "SYSADM"  ""
################################################################################
#
CREATE PROCEDURE IF_CALL (INPUT C1 CHAR (20))
LANGUAGE SQL
BEGIN
        DECLARE OBJ CHAR (10);
        IF C1 = 'CRETB' THEN
                CALL CRETB;
        ELSEIF C1 = 'CASE' THEN
                CALL CASE_TEST_2 (OBJ);
        ELSE
                CALL
INS(1,2,3,4,5,6,7,'binary','char','varchar',N'NCHAR',N'NVARCHAR',
'2008-01-01','11:11:11','2008-01-01 11:11:11');
        END IF;
END;
```

➲ 例 2：以下の例はIF…SET文を含むSQLストアド・プロシージャ：

```
################################################################################
#
#       Module Name = SETTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "SET" and "IF" in Store Procedure
#       Function = 1. create table TB_1 and insert data into TB_1
#       Use Database: "DBSAMPLE5" "SYSADM"  ""
#                       table : TB_1(V1 INTEGER)
################################################################################
CREATE PROCEDURE IFTEST_1(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT  V1 FROM TB_1;
```

```
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                IF P_SAL = 2 THEN
                SET P_SUM = 10;
                ELSE
                SET P_SUM = 20;
                END IF;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
```

➲ 例 3：以下の例はIF…ELSEIF…ELSE文を含むSQLストアド･プロシージャ：

```
################################################################################
####
#       Module Name = IF_UPDATE.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "IF" and "UPDATE" in Store Procedure
#       Function = 1. create table TB_1 and insert data into TB_1
#       Use Database : "DBSAMPLE5" "SYSADM"  ""
#       table : TB_1(V1 INT,V2 DOUBLE,V3 CHAR(6))
################################################################################
###
CREATE PROCEDURE IF_UPDATE (IN C1 CHAR(6), IN C2 CHAR(20))
LANGUAGE SQL
BEGIN
```

```
        IF C2 = 'FIRST' THEN
                UPDATE TB_1 SET V2 = V2 * 1.10, V1 = 1000
                WHERE V3 = C1;
        ELSEIF C2 = 'SECOND' THEN
                UPDATE TB_1 SET V2 = V2 * 1.05, V1 = 500
                WHERE V3 = C1;
        ELSE
                UPDATE TB_1 SET V2 = V2 * 1.03, V1 = 0
                WHERE V3 = C1;
        END IF;
END;
```

➲ 例 4：以下の例はIF…ELSEIF文を含むSQLストアド・プロシージャ：

```
################################################################################
#
#       Module Name = ELSEIFS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "ELSEIF" in Store Procedure
#       Use Database: "DBSAMPLE5" "SYSADM"  ""
################################################################################
#
CREATE PROCEDURE ELSEIFS(IN con INT, OUT c1 INT)
LANGUAGE SQL
BEGIN
        IF con = 1 THEN
                SET c1 = 1;
                INSERT INTO TB_1 VALUES(C1);
        ELSEIF con = 2 THEN
                SET c1 = 2;
                INSERT INTO TB_1 VALUES(C1);
        ELSEIF con = 3 THEN
                SET c1 = 3;
                INSERT INTO TB_1 VALUES(C1);
        ELSE
                IF con = 5 THEN
```

```
                        SET c1 = 5;
                        INSERT INTO TB_1 VALUES(C1);
                ELSEIF con = 6 THEN
                        SET c1 = 6;
                        INSERT INTO TB_1 VALUES(C1);
                ELSE
                        SET c1 = 7;
                        INSERT INTO TB_1 VALUES(C1);
                END IF;
        END IF;
END;
```

## ループ文

ループ文はロジックを重複に実行して条件を満たすまでサポートを提供します。SQL制御文に以下のループ文をサポートします：

- FOR
- LOOP
- WHILE
- REPEAT

FOR文はほかの文と違って、これは定義した結果セットの行を反復するため使用されます。しかし、ほかの文は条件に適するまで一連のSQL文を反復します。

### SQLストアド・プロシージャにFOR文

FOR文はループ文の特別なタイプです。これらは定義の読み取り専用結果セットの行に反復できるからです。FOR文を実行した後、FOR-loop循環に自動にカーソルを宣言します。取るなら次の行は結果セットです。ループが続いて結果セットに行がなくなるまでです。

FOR文はカーソルの実行を簡単化になせます、また論理操作に行セットのカラム値を容易に取り戻します。

CURSUR値は０でさえなければ、FOR文の文リストは重複されます。statement_listは一つまたはもっと多くの文から構成します。FOR文が標記されることができます。begin_labelはあるとend_labelがあります。二つもあるなら同じでなければなりません。

➲ 構文：FOR文の構文：

```
FOR [<for loop variable name> AS]
[<cursor name>[<cursor sensitivity>] CURSOR FOR]
<cursor specification>
DO
<sql statement list>
END FOR
```

FOR文の構文は以下の四つの形式をサポートする：

- *FOR* x AS select * from t1
- FOR x AS cur CURSOR FOR select * from t1
- FOR cur CURSOR FOR select * from t1
- FOR select * from t1

**NOTE** *FOR文には必ずselect文が必要です。ユーザーがSQLストアドプロシージャを作成する時点でクエリ表が存在している必要はありません。xは参照変数の範囲を表します。例えば、x.c1, x.c2を使用してクエリ表t1の結果を表示できますが、xがなければ、ユーザーはc1, c2のみ使用でき、参照方式は使用できません。CurはFOR文がcurというカーソルを生成することを表わしています。このカーソルは参照変数としては使用できませんが、カーソル関連の構文には使用できます。例えば、**where current of xxxx, FETCH**などです*

- 例 1: テーブルt1のすべての値の合計を計算し、集計値をテーブルt2に挿入します。C1はテーブルt1のカラム名です。これによりFOR構文で、c1を表現するためにc1およびx.c1の両形式を使用することが可能です：

```
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);

CREATE PROCEDURE test1(OUT res INT)
LANGUAGE SQL
BEGIN
    SET res = 0;
    FOR x AS select * from t1
    DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (x.c1);
    END FOR;
END;
```

- 例 2：この例では**x i**はループ変数名で、変数**old**を参考するため使用されます。**cur**はカーソル名で、カーソル関連の構文に使用されます。例えば、**update**構文などです。

```
CREATE TABLE t1 (c1 INT);

CREATE PROCEDURE test3
LANGUAGE SQL
BEGIN
    FOR x AS cur CURSOR FOR select c1 as old from t1 for update
    DO
        update t1 set c1 = x.old + 100 where current of cur;
    END FOR;
END;
```

- 例 3：この例ではループ変数名がありませんので、ユーザーは直接変数**old**を使用できます、**ur**はカーソル名で、カーソル関連の構文に使用されます。例えば、**update**構文などです。

```
CREATE TABLE t1 (c1 INT);

CREATE PROCEDURE test4
LANGUAGE SQL
BEGIN
    FOR cur CURSOR FOR select c1 as old from t1 for update
    DO
        update t1 set c1 = old + 100 where current of cur;
    END FOR;
END;
```

➲ 例 4：機能は上記の例と同様です。しかし、FORステートメントはカーソルあるいはループ変数名は使用しません。そのためx.c1は表現として使用できません。

```
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);

CREATE PROCEDURE test2(OUT res INT)
LANGUAGE SQL
BEGIN
    SET res = 0;
    FOR select * from t1
    DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (c1);
    END FOR;
END;
```

## SQLストアド･プロシージャにLOOP文

LOOP文はループ文の特別なタイプです。これは終了の条件句がないからです。LOOP文は定義した一連の文が反復に実行して別のロジックと出会うまでです。このロジックは普通は転送制御文で、制御流れをループの外部に跳ぶすることをフォースするため使用されます。

LOOP文は、複雑な処理を繰り返し実行し終了する際に有効です。しかし、無限ループには注意すべきです。

LOOP文は転送制御文がなくなり、単独に使用されると、ループに一連の文を無限に実行されます。そうしないと、条件ハンドルを増加して制御流れの変更をフォースします。または現れる条件に処理できずストアドプロシージャが戻ることをフォースします。

LOOPは簡単な循環構成が実現できて、一つまたは複数の文を構成する文リストを実行できます。ループ文が重複に実行されてループが終了するまでです、これは普通にLEAVE文と一緒に完成します。LOOP文が標記されることができます。begin_labelはあるとend_labelもあります。二つもあるなら同じでなければなりません。

➲ 構文：LOOP文の構文

```
<loop_statement_main> ::=
LOOP <sp_statement_main> END LOOP
```

➲ 例 :以下の例はLOOP文を含むSQLストアド･プロシージャ :

```
CREATE PROCEDURE LOOP_IF (OUTPUT sum INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE p_sum INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT V1 FROM TB_1;
        SET p_sum = 0;
        SET p_sal = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
        LOOP
                SET p_sum = p_sum + P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                IF P_SAL=NULL THEN
                        BREAK;
                END IF;
        END LOOP;
        CLOSE CUR;
        SET sum = p_sum;
END;
```

**NOTE** *luaファイルにNULLは零に転換します。*

## SQLストアド･プロシージャにWHILE文

earch_conditionは真になるとWHILE文は重複されます。statement_listは一つ或いは一つ以上の文から構成します。WHILE文はラベルされることができます。begin_labelはあるとend_labelもあります。二つもあるなら同じでなければなりません。

➲ 構文:WHILE文の構文

```
<while_statement_main> ::=
WHILE <condition_name> DO <sp_statement_main> END WHILE
```

➲ 例 1:以下の例は簡単なWHILEループを含むSQLストアド・プロシージャです：

```
CREATE PROCEDURE WHILE_LOOP(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT V1 FROM TB_1;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                WHILE (P_SAL!= NULL) DO
                        SET P_SUM = P_SUM + P_SAL;
                        FETCH NEXT FROM CUR INTO P_SAL;
                END WHILE;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
```

➲ 例 2：while文の用法

```
################################################################################
#
#       Module Name = SUM_WHILE.SP
#       Purpose  = Store Procedure testing program
#                       1. Test keyword "WHILE" in Store Procedure
#       Function = 1. create test table TB_1 and TB_2
#                          2. insert test data into TB_1 and TB_2
#                          3. select data from TB_1 TB_2 and use crusor
get data
#       Use Database : "DBSAMPLE5" "SYSADM"  ""
#                   table    : TB_1(V1 INTEGER) TB_2(V1 INTEGER)
################################################################################
#
CREATE PROCEDURE SUM_WHILE(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
```

```
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE P_SAL1 INTEGER;
        DECLARE CUR CURSOR FOR SELECT  V1 FROM TB_1;
        DECLARE CUR1 CURSOR FOR SELECT  V1 FROM TB_2;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                WHILE(P_SAL != NULL)DO
                        OPEN CUR1;
                        FETCH FROM CUR1 INTO P_SAL1;
                                WHILE(P_SAL1 != NULL)DO
                                        SET P_SUM = P_SUM + P_SAL1;
                                        FETCH NEXT FROM CUR1 INTO P_SAL1;
                                END WHILE;
                        CLOSE CUR1;
                        FETCH NEXT FROM CUR INTO P_SAL;
                END WHILE;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
The NULL in procedure will be translate as nil in lua file
```

## SQLストアド･プロシージャにREPEAT文

REPEAT文は一組実行の文を定義してREPEATループの端末が真になるまで終了します。repeat-loop-conditionはループの反復が終った後評価されます。

WHILE文の一番目の行while-loop-conditionは偽になると、ループに入れません。この場合、REPEAT文は非常に重要になります；while-loopロジックはREPEAT文として書き直されます。

REPEAT文とUNTIL文はループを作成するため使用されて倫理的な条件を満たすまで終ります。

明らかな条件のせいでループの終了を引き起こすので、REPEATループは容易に維持できます。LEAVE文は簡単なループにどこでもありますが、UNTIL文は常にEND REPEAT句と連合してループの端末にあります。さらに、UNTIL条件は今のループに特定ですので、ユーザーはループにREPEATラベルを指定する必要がありません。読みやすくなるため、REPEATループのラベルを使用したほうがいいです。特にループが埋め込まれる場合です。

REPEATループは少なくとも一回を実行するのを保証します。ループを始めに実行した後UNTIL条件が始めに評価されます。このような条件を満たしないと、ループは一回でも実行できません。

➲ 例：以下の例はREPEAT文を含むSQLストアド・プロシージャです：

```
CREATE PROCEDURE REPEATS
LANGUAGE SQL
BEGIN
        DECLARE V INTEGER DEFAULT 0;
          REPEAT
                INSERT INTO TB_1 VALUES(1,2,3,4);
                SET V = V+1;
        UNTIL V>10
        END REPEAT;
END;
```

## 転送制御文

転送制御文はSQLストアド･プロシージャに制御流れを再構成します。無条件の転移は制御流れを一つのポイントからほかのポイントまで跳ばせて、転送制御文を優先或いは付き従うことになります。SQLストアド･プロシージャに以下の転送文をサポートします：

- ITERATE
- LEAVE

SQLストアド･プロシージャの転送文はどこでも使用できますが、ITERATEとLEAVEは普通にLOOP文或いは他のループ文と一緒に使われます。

## SQLストアドプロシージャにITERATE文

制御流れをLOOPラベル文の始めに返せるため、ITERATE文が使用されます。

➲ 例：以下の例はITERATE文を含むSQLストアド・プロシージャです：

```
CREATE PROCEDURE ITERATOR
LANGUAGE SQL
BEGIN
        DECLARE v_deptno CHAR(3);
        DECLARE v_deptname VARCHAR(29);
        DECLARE c1 CURSOR FOR SELECT deptno, deptname FROM department
ORDER BY deptno;


        OPEN c1;
        LOOP
                FETCH c1 INTO v_deptno, v_deptname;
                IF v_deptno = NULL THEN
                        LEAVE;
                ELSEIF v_deptno = 'D11' THEN
                          INSERT INTO department (deptno, deptname)
VALUES (‘NEW’, v_deptname);
                        ITERATE;
                ELSE
                        ITERATE;
                END IF;
        END LOOP;
        CLOSE c1;
END;
```

この例に、戻り行のカラム値は某値をマッチする場合、ITERATE文はLOOP文が定義した循環まで制御流れを返します。ITERATE文の位置は値がなくdepartment表に挿入できることを確保します。

## SQLストアドプロシージャにLEAVE文

LEAVE文は循環または複合文からプログラム制御を転送出します。この文はSQLストアド･プロシージャ或いは動的複合文に埋め込めます。これは実行可能な文ではないので、動的に準備できません。

➲ 例：以下の例はLEAVE文を含むSQLストアド・プロシージャです：

```
CREATE PROCEDURE ITEA(OUT C1 INT)
LANGUAGE SQL
BEGIN
        DECLARE V1 INT;
        DECLARE CUR CURSOR FOR SELECT * FROM T3;
        OPEN CUR;
        FETCH CUR INTO V1;
        LOOP
        IF V1 = 2 THEN
                SET C1 = 1;
                FETCH NEXT FROM CUR INTO V1;
                ITERATE;
        ELSEIF V1 != NULL THEN
                FETCH NEXT FROM CUR INTO V1;
                ITERATE;
        ELSE
                LEAVE;
        END IF;
        END LOOP;
        CLOSE CUR;
END;
```

## ラベルとSQLストアド･プロシージャの複合文

ラベルはSQLストアド･プロシージャのどんな制御文でも必要による名を付けることができます、複合文とループも含みます。ユーザーは実行流れをフォースして複合文或いはループから跳び出します、また複合文或いはル

ープの始めに跳び入れます。ラベルはITERATEとLEAVE文を通じて参照できます。

ユーザーは複合文のENDに相応するラベルが提供できます。終了ラベルを提供すると、この終了ラベルは開始のレベルと同じではなければなりません。

SQLストアド・プロシージャ体の各なラベルはユニークです。

同じな名を持つ変数は複数な複合文に宣言されると、ラベルは名称の混乱が避けれます。ラベルはSQL変数の名が制限できます。

➲ 構文：以下はコンパウンドステートメント構文です。

```
[<label name> COLON] BEGIN [<any statement list>] END [<label name>]
```

➲ 例：

```
CREATE PROCEDURE test3
LANGUAGE SQL
L1: BEGIN
    ...
    L2: BEGIN
        ...
    END L2;
END L1;
```

**NOTE** 〈ラベル名〉はコンパウンドステートメント内で適宜表現される必要があり、加えてコンパウンドステートメントはネストで使用することもできます。

➲ 例：以下の例は簡単なラベルがあるSQLストアド・プロシージャです：

```
CREATE PROCEDURE LABEL_1(OUT C1 char(20))
LANGUAGE SQL
BEGIN
        DECLARE V1 INT;
        DECLARE CUR CURSOR FOR SELECT * FROM T3;
        OPEN CUR;
        FETCH CUR INTO V1;
        label1: LOOP
        IF V1 = 2 THEN
                SET C1 = 'OK';
```

```
                FETCH NEXT FROM CUR INTO V1;
                ITERATE label1;
        ELSEIF V1 != NULL THEN
                FETCH NEXT FROM CUR INTO V1;
                ITERATE label1;
        ELSE
                LEAVE label1;
        END IF;
        END LOOP label1;
        CLOSE CUR;
END;
```

## 共通変数のスコープチェック

共通変数は、ベースデータタイプによって定義されます。例えばINT、CHAR、FLOAT等のSQLストアドプロシージャでサポートされるデータ型です。すべての変数は独自の利用可能スコープを保持します。前レベルで定義された変数は次レベルで使用することができますが、次レベルで定義された変数は前レベルでの使用はできません。

SQLストアドプロシージャでは、新しいレベルを示すために、BEGINE..ENDステートメントを使用することによりコンパウンドステートメントを囲い込むことができます。

➲ 例 1:

```
CREATE PROCEDURE test4(OUT res1 INT, OUT res2 INT, OUT res3 INT)
LANGUAGE SQL
L1: BEGIN
    DECLARE c1, c2 INT;
    SET c1 = 1; # レベルL1で値をセットします。
    L2: BEGIN
        DECLARE c1 INT;
        SET c1 = 2;
        SET c2 = 3;
        SET res3 = c1; # C1がレベルL2で再定義され、レベルL1の値が遮断されます。res3=2となります。
    END L2;

    SET res1 = c1; # レベルL2で割り当てられたステートメントが終了し、c1はレベルL1での値を
    SET res2 = c2; # C2はL1でのみ定義されているため、L2での変更はレベルL1に影響し、res2=3となります。
END L1;
```

**NOTE** *Forステートメントもまた新しいレベルとなります。*

➲ 例 2 :

```
CREATE PROCEDURE test5(OUT res INT)
LANGUAGE SQL
BEGIN
    DECLARE c1 INT;
    SET c1 = 10;
    SET res = 0;
    FOR select * from t1
    DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (c1);
    END FOR;

    # 影響するスコープはFORステートメント内に限定され、FORステートメントが終了する時もc1の値は依然として10です。
END;
```

## SQLストアド・プロシージャのSQLCODEとSQLSTATE変数

エラー処理を実行またはSQLストアド・プロシージャをデバッグする時、ユーザーはSQLCODEとSQLSTATEの値が非常に有効だということを見つかります。これらの値を返して出力引数とする或いは表に挿入して、基礎なトレース・サポートを提供します。

一つの文を実行さえすれば、DBMasterは自動的にこの変数を設置します。もし一つの文は既存の処理に条件をあげると、ハンドル実行を始める場

合、SQLSTATEとSQLCODEの変数値が利用できます。しかし、一度ハンドルの一番目の文を実行すると、これらの変数は再びに設置されます。だから、一番目の文を実行する場合、普通にはSQLSTATEとSQLCODE値をローカル変数にコピーします。以下の例に、任意の条件を持つCONTINUEハンドルはSQLCODE変数をretcodeという変数にコピーします。retcodeは実行文に使用されてプロシージャ･ロジックを制御します、また出力引数として戻り値をパスします。

構文の実行ステータスのチェックが必要な場合、初めにSQLCODE変数を使用してください。また、SQLSTATEを使用することによって詳細なエラーメッセージを確認することができます。これにより実行される構文の実行ステータスを正確に知ることができます。

```
BEGIN
DECLARE retcode INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND SET
retcode = SQLCODE;
END
```

**SQLストアドプロシージャーでの SQLCODE 変数定義**

SQLストアドプロシージャでのSQLCODE変数の戻り値は以下のように定義されます。

= 0　　成功: 処理が成功したことを示します。

= 1　　警告: 処理結果が不正であることを示します。

= 100　　データが見つからない:処理結果としてデータが見つからないことを示します。

<0　　DBMasterエラーコードを示します。

**SQLストアドプロシージャーでのSQLSTATE変数定義**

SQLストアドプロシージャーに含まれるSQLSTATE変数の情報はSQLのODBC 3.0の標準定義に準拠しております。詳細はDBMasterマニュアル”エラー・メッセージ参照編をご参照ください。

### SQLストアド･プロシージャの条件ハンドル

SQLストアド･プロシージャの条件ハンドル：条件が現れる時、条件ハンドルはユーザーのSQLストアド･プロシージャの操作を決定します。ユーザーはSQLストアド･プロシージャに普通条件、名付け条件、特別なSQLSTATE値に一つ或いは複数の条件ハンドルを宣言することができます。

SQLストアド･プロシージャに一つの文はSQLWARNING或いはNOT FOUND条件をあげると、また条件にハンドルを宣言すると、DBMasterは相応なハンドルに制御権を渡します。この条件にハンドルを宣言しないと、DBMasterはSQLストアド･プロシージャに次の文に制御権を渡します。SQLCODEとSQLSTATE変数はもう宣言された場合、これらは条件ハンドルの相応値があります。

SQLストアド･プロシージャに一つの文はSQLWARNING条件をあげると、またユーザーはSQLSTATE或いはSQLEXCEPTION条件にハンドルを宣言すると、DBMasterはハンドルに制御権を渡します。SQLCODEとSQLSTATE変数はもう宣言された場合、成功に実行した後、値はそれぞれが‘00000’と0です。

SQLストアド･プロシージャに一つの文はQLEXCEPTION条件をあげると、またはユーザーはSQLSTATE或いはSQLEXCEPTION条件にハンドルを宣言しなかったら、DBMasterはSQLストアド･プロシージャを終了してコーラに返します。

## 5.8 SQLストアド・プロシージャの戻り結果セット

SQLストアド・プロシージャに、カーソルは結果セットの行循環より使用範囲がもっと広いです。これは結果セットをコール・プログラムに返すことができます。

SQLストアド・プロシージャから結果セットを返すため、以下の操作を実行してください：

1. WITH RETURN句のDECLAREカーソルを使用する。

2. SQLストアド・プロシージャにカーソルを開く。
3. クライアント・アプリケーションにカーソルのオープンな状態を保持する–閉じないでください。

➲ 例 1:以下の例は簡単な戻りがあるSQLストアド・プロシージャです：

```
CREATE PROCEDURE call_ret
LANGUAGE SQL
BEGIN
DECLARE cur CURSOR WITH RETURN FOR select * from tb;
OPEN cur;
END;
```

➲ 例 2：

```
#############################################################################
####
#       Module Name = RETU.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "RETURN" in Store Procedure
#       Function = 1. decalar r1~r14 for pass parameter
#                                2. create table tb_1 and insert data into
tb_1
#                                3. use cursor get data from tb_1
#       Use Database : DBSAMPLE5
#       table : TB_1(V1 int,V2 smallint,V3 INT,V4 FLOAT,V5 DOUBLE,V6
DECIMAL(20,4),
#                       V7 BINARY(10),V8 CHAR(20),V9 VARCHAR(20),V10
NCHAR(40),
#                       V11 NVARCHAR(40),V12 DATE,V13 TIME,V14 TIMESTAMP)
#############################################################################
###
CREATE PROCEDURE RETU
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR select * from tb_1;
        OPEN CUR ;
END;
```

以下のはRETUをコールして返す結果です：

```
dmSQL> call retu;
V1  V2  V3  V4  V5   V6   V7  V8  V9  V10  V11   V12 V13 V14
=== === === === === ==== ==== === ========== ====== === === ====
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* 汉* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* ba00ba00d* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* ba00ba00d* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
11 rows selected
```

# 5.9 動的なSQLストアド・プロシージャ

## EXECUTE IMMEDIATE文

ユーザーは動的にEXECUTE IMMEDIATE文が準備して実行できます。

➲ 構文：EXECUTE IMMEDIATE文の構文：

```
<execute immediate statement main> ::=
EXECUTE IMMEDIATE <SQL statement variable>
```

<SQL statement variable>宣言のタイプはキャラクタ・ストリングです。

➲ 例：

```
CREATE PROCEDURE dym_test1
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
```

```
        SET str= 'insert into t1 values(5)';
        EXECUTE IMMEDIATE str;
END;
```

## PREPARE文

実行に文を準備します。

➲ 構文:PREPARE文の構文 :

```
<prepare statement main> ::=
PREPARE <SQL statement name> FROM <SQL statement variable>
```

➲ 例:

```
CREATE PROCEDURE dym_test2
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';

        PREPARE stmt FROM str;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END;
```

## EXECUTE 文

入力引数と出力ターゲットを接合してこの文を実行します。

➲ 構文:EXECUTE文の構文 :

```
<execute statement> ::=
EXECUTE <SQL statement name> [ <result using clause> ] [ <parameter
using clause> ]
<result using clause> ::= INTO <into argument> [ { <comma> <into
argument> }... ]
<parameter using clause> ::= USING <using argument> [ { <comma> <using
argument> }... ]
<into argument> ::= SQL SP variable
```

```
<using argument> ::= SQL SP variable
```

➲ 例 1:

```
CREATE PROCEDURE dym_test3(IN val INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(?)';

        PREPARE stmt FROM str;
        EXECUTE stmt USING val;
        DEALLOCATE PREPARE stmt;
END;
```

➲ 例 2:

```
CREATE PROCEDURE dym_test4(IN val INT, OUT co INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'select COUNT(*) from t1 where c1=?';

        PREPARE stmt FROM str;
        EXECUTE stmt INTO co USING val;
        DEALLOCATE PREPARE stmt;
END;
```

## DEALLOCATE PREPARE 文

PREPARE文を通じて準備できたSQL文を分配します。

➲ 構文: DEALLOCATE PREPARE文の構文 :

```
<deallocate prepared statement> ::= DEALLOCATE PREPARE <SQL statement
name>
```

➲ 例 :

```
CREATE PROCEDURE dym_test2
```

```
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';

        PREPARE stmt FROM str;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END;
```

## 動的なカーソルの宣言

文名と関連するカーソルを宣言する場合、この宣言文の名はカーソルと輪番に関連できます。

➲ 構文:動的なカーソル構文：

```
<dynamic declare cursor> ::= DECLARE <dynamic cursor name> [[NO] SCROLL]
CURSOR [WITH RETURN] FOR <prepare statement name>
```

## 動的なオープン・カーソル

カーソルと入力動的な引数を関連してカーソルを開きます。

➲ 構文:動的なオープン・カーソル構文：

```
<dynamic open statement> ::= OPEN <dynamic cursor name> [ <input using
clause> ]
```

➲ 例 1:動的なカーソル：

```
CREATE PROCEDURE dym_test5(OUT sum INT)
LANGUAGE SQL
BEGIN
        DECLARE val INT;
        DECLARE str char(128);
        DECLARE CONTINUE HANDLER FOR NOT FOUND;

       SET str= 'select * from t1';
```

```
        SET sum = 0;

        PREPARE stmt FROM str;
        DECLARE cur CURSOR FOR stmt;

        OPEN cur;
        WHILE SQLCODE = 0 DO
                        SET sum = sum + val;
                        FETCH cur INTO val;
                END WHILE;
        CLOSE cur;

        DEALLOCATE PREPARE stmt;
END;
```

➲ 例 2:動的なカーソル、入力引数を含む：

```
CREATE PROCEDURE dym_test6(IN cd INT, OUT sum INT)
LANGUAGE SQL
BEGIN
        DECLARE val INT;
        DECLARE str char(128);
        DECLARE CONTINUE HANDLER FOR NOT FOUND;

        SET str= 'select * from t1 where c1=?';
        SET sum = 0;

        PREPARE stmt FROM str;
        DECLARE cur CURSOR FOR stmt;

        OPEN cur USING cd;
        WHILE SQLCODE = 0 DO
                        SET sum = sum + val;
                        FETCH cur INTO val;
                        END WHILE;
```

```
        CLOSE cur;

        DEALLOCATE PREPARE stmt;
END;
```

➲ 例 3:動的なカーソルの結果セットを返す：

```
CREATE PROCEDURE dym_test7(IN cd INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'select * from t1 where c1=?';

        PREPARE stmt FROM str;
        DECLARE cur CURSOR WITH RETURN FOR stmt;

        OPEN cur USING cd;
```

# 5.10 データ・プロセス

本章は簡単な例を通じてSQLストアド・プロシージャ・データ・プロセスを説明します。

## 空のSQLストアド・プロシージャを作成する

➲ 例 ：空のSQLストアド・プロシージャを作成する：

```
CREATE PROCEDURE SQLSP
LANGUAGE SQL
BEGIN
END;
```

## INSERT文

以下の例はSQLストアド・プロシージャにinsert文をどう使用することを説明します。

➲ 例：:SQLストアド・プロシージャを作成して、表にデータの挿入操作を終了する：

```
##############################################################################
####
#       Module Name = INPUTS.SP
#       Purpose  = Store Procedure testing program
#              1. Test INPUT parameter store procedure
#              2. Test all type that can use in Store Procedure
#       Function = 1. create table INPUTS
#       Use Database : "DBSAMPLE5" "SYSADM"  ""
#       table : INPUTS(V1 int,V2 BIGINT,V3 smallint,V4 INT,V5 FLOAT,
#                      V6 DOUBLE, V7 DECIMAL(20,4),V8 CHAR(10),
#                       V9 CHAR(20),V10 VARCHAR(20), V11 NCHAR(40),
#                       V12 NVARCHAR(40),V13 DATE,V14 TIME,V15 TIMESTAMP)
##############################################################################
###
C CREATE PROCEDURE INPUTS(INPUT V1 int, INPUT V2 BIGINT,
                        INPUT V3 smallint, INPUT V4 INT,
                         INPUT V5 FLOAT,INPUT V6 DOUBLE,
                         INPUT V7 DECIMAL(20,4),INPUT V8 BINARY(20),
                         INPUT V9 CHAR(20),INPUT V10 VARCHAR(20),
                         INPUT V11 NCHAR(40),INPUT V12 NVARCHAR(40),
                         INPUT V13 DATE,INPUT V14 TIME,
                         INPUT V15 TIMESTAMP)
LANGUAGE SQL
BEGIN
        INSERT INTO INPUTS VALUES(V1,V2,V3,V4,V5,V6,V7,V8,
V9,V10,V11,V12,V13,V14,V15);
END;
```

## Select文

以下の例はSQLストアド・プロシージャにselect文をどう使用することを説明します。

➲ 例：

```
###############################################################################
####
#       Module Name = OUTPUTS.SP
#       Purpose  = Store Procedure testing program
#              1. Test OUTPUT parameter store procedure
#              2. Test all type which can use in Store Procedure
#       Function = 1. create table OUTPUTS and insert data into OUTPUTS
#       2. use cursor get data from OUTPUTS and pass data  to OUTPUT
parameter
#       Use Database : DBSAMPLE5
#       table : OUTPUTS(V1 int, V2 BIGINT, V3 smallint,V4 INT,
#                       V5 double,V6 DOUBLE, V7 DECIMAL(20,4),
#                      V8 BINARY(10),V9 CHAR(20), V10 VARCHAR(20),
#                      V11 NCHAR(40),V12 NVARCHAR(40),
#                       V13 DATE,V14 TIME,V15 TIMESTAMP)
###############################################################################
###

CREATE PROCEDURE OUTPUTS(OUTPUT V1 int, OUTPUT V2 BIGINT,
                         OUTPUT V3 smallint, OUTPUT V4 INT,
                          OUTPUT V5 FLOAT,OUTPUT V6 DOUBLE,
                         OUTPUT V7 DECIMAL(8,4),OUTPUT V8 BINARY(20),
                         OUTPUT V9 CHAR(20),OUTPUT V10 VARCHAR(20),
                         OUTPUT V11 NCHAR(40),
                         OUTPUT V12 NVARCHAR(40),OUTPUT V13 DATE,
                         OUTPUT V14 TIME,OUTPUT V15 TIMESTAMP)
LANGUAGE SQL
BEGIN
        DECLARE CUR CURSOR FOR select * from OUTPUTS;
```

```
        OPEN CUR;
        FETCH FROM CUR INTO
V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15;
        CLOSE CUR;
END;
```

## Create文

以下の例はSQLストアド・プロシージャにcreate文をどう使用することを説明します。

➲ 例 1:SQLストアド・プロシージャを使用して表を作成する：

```
CREATE PROCEDURE CRETB
LANGUAGE SQL
BEGIN

         CREATE TABLE TB_1(V1 int, V2 BIGINT, V3 smallint,V4 INT,
                         V5 FLOAT,V6 DOUBLE,V7 DECIMAL(8,2),V8 CHAR(20),
                          V9 CHAR(20),V10 VARCHAR(20),V11 CHAR(40),
                          V12 VARCHAR(40),V13 DATE,V14 TIME,
                          V15 TIMESTAMP);
END;
```

➲ 例 2:SQLストアド・プロシージャを使用してビューを作成する：

```
CREATE PROCEDURE CREVE
LANGUAGE SQL
BEGIN
        CREATE VIEW VE_1 AS SELECT * FROM TB_1;
END;
```

➲ 例 3:SQLストアド・プロシージャを使用して**user-defined**データ型を作成する：

```
CREATE PROCEDURE CREDM
LANGUAGE SQL
BEGIN
        CREATE DOMAIN TYP_1 VARCHAR(35);
```

```
END;
```

➲ 例 4:SQLストアド・プロシージャを使用して索引を作成する：

```
CREATE PROCEDURE CREIND
LANGUAGE SQL
BEGIN
        CREATE UNIQUE INDEX IND_1 ON TB_1(V1);
END;
```

## Drop文

以下の例はSQLストアド・プロシージャにdrop文をどう使用することを説明します。

➲ 例 1:SQLストアド・プロシージャを使用して表を削除する：

```
CREATE PROCEDURE DRPTB(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP TABLE' THEN
                DROP TABLE TB_1;
                SET WORNING ='NORMAL! TABLE TB_1 DROPED';
        ELSEIF V1 = 'CREATE TABLE' THEN
                CALL CRETB;
                SET WORNING ='NORMAL! CREATE A NEW TABLE NAMED TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

➲ 例 2:SQLストアド・プロシージャを使用してビューを削除する：

```
CREATE PROCEDURE DRPVE(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP VIEW VE_1' THEN
                DROP VIEW VE_1;
```

```
                SET WORNING ='NORMAL! VIEW VE_1 DROPED';
        ELSEIF V1 = 'CREATE VIEW' THEN
                CALL CREVE;
                SET WORNING ='NORMAL! CREATE VIEW AS SELECT V1 FROM
TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

## SQLストアド・プロシージャ実行を辿る

トレース機能性はSQLストアド・プロシージャのデバッグの実行を辿るため使用されます。TRACE機能を使用して辿る変数とプリント情報を確定します。SQLストアド・プロシージャを実行した後、全てのトレース情報はDBMaster bin ディレクトリ下の**_sptrace.log** に置いてます。

➲ 構文:TRACE文の構文：

```
<TRACE statement main> ::=
TRACE [ON <trace_file> | OFF | <trace_detail>]
```

以下のはTRACEの使用形式です：

```
TRACE('V1=', V1);
TRACE('V2=', V2, 'V3=', V3);
```

表現は“ ’ ”を使用しなければなりません。全部のデータ型をサポートします。でも、BINARY, NCHAR, NVARCHARデータ型は16進のフォーマットで表されます。

ユーザーはTRACE機能を使って辿る変数をセットして、結果を出力します。SQLストアド・プロシージャを実行した後、全てのトレース情報は**spusr.log**というファイルに保存されます。このファイルはDBMasterクライアント側のdmconfigファイルに**DB_SPLog**定義ディレクトリにあります。

➲ 例 1:

```
CREATE PROCEDURE INPUTS(INPUT V1 int, INPUT V2 BIGINT,
                        INPUT V3 smallint,INPUT V4 INT,
```

```
                        INPUT V5 FLOAT,INPUT V6 DOUBLE,
                        INPUT V7 DECIMAL(20,4),
                        INPUT V8 BINARY(20),INPUT V9 CHAR(20),
                        INPUT V10 VARCHAR(20),INPUT V11 NCHAR(40),
                        INPUT V12 NVARCHAR(40),INPUT V13 DATE,
                        INPUT V14 TIME, INPUT V15 TIMESTAMP)
LANGUAGE SQL
BEGIN
        TRACE ON;
        TRACE('V1=', V1);
        TRACE('V2=', V2);
        TRACE('V3=', V3);

        TRACE('V4=', V4);
        TRACE('V5=', V5);

        TRACE('V8=', V8);
        TRACE('V9=', V9);

        TRACE('V12=', V12);
        TRACE('V13=', V13);
        TRACE('V14=', V14);
        TRACE('V15=', V15);
        TRACE OFF;
                INSERT INTO INPUTS
VALUES(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15);
END;
```

(1,7396,2,3,4,5,6,'binary','char','varchar','NCHAR','NVARCHAR','2008-01-01','11:11:11','2008-01-01 11:11:11')入力をコールした後、_sptrace.logは以下のログを追加する：

```
INPUTS 47: Begin trace ====>
INPUTS 48: V1=1
INPUTS 49: V2=7396
INPUTS 50: V3=2
```

```
INPUTS 51: V4=3
INPUTS 52: V5=4
INPUTS 53: V8=62696e617279
INPUTS 54: V9=char
INPUTS 55: V12=4e00560041005200430048004100 5200
INPUTS 56: V13=2008-01-01
INPUTS 57: V14=11:11:11
INPUTS 58: V15=2008-01-01 11:11:11.000
```

# 6 SQLストアド・プロシージャ

SQLストアド・プロシージャの開発はほかのストアド・プロシージャの開発とほぼ同じです。このプロセスはデザインから配置まで全てのステップを含みます。

以下のセクションから SQL ストアド・プロシージャの使用について説明します。

- SQLストアド・プロシージャの作成
- SQLストアド・プロシージャの実行
- SQLストアド・プロシージャの削除

SQLストアド・プロシージャの開発について、幾つの例示を提供します。

## 6.1 SQLストアド・プロシージャの作成

SQLストアド・プロシージャを作成するため、ユーザーの要求、SQLストアド・プロシージャの特性、特性の応用、デザインに影響する制限などについて理解する必要があります。

SQLストアド・プロシージャの作成はデータベース・オブジェクトの作成と同じで、SQL文から構成します。SQLストアド・プロシージャは

DBMasterコマンド・ライン（dmSQLツール）で作成でき、また直接グラフィカル・ユーザー・インタフェース・ツール（JDBAツール）を使用しても作成できます。

## ファイルからSQLストアド・プロシージャを作成

まず、SQLストアド・プロシージャを書き入れてファイルに保存します。それから、dmSQLツールを通じてSQLストアド・プロシージャをデータベースに保存します。

**NOTE** *DBMaster SQLストアド・プロシージャは外部ファイル(*.sp )をコールして作成されます、dmSQLコマンド・ライン・ツールに直接にこれを書き入れません。*

➲ 構文:SQLストアド・プロシージャ作成の構文：

```
<CREATE SQL PROCEDURE 構文> ::=
CREATE PROCEDURE FROM <file_name>
```

➲ 例:ファイルからSQLストアド・プロシージャを作成：

```
dmSQL> CREATE PROCEDURE FROM ‘CRETB.SP’;
dmSQL> CREATE PROCEUDRE FROM ‘.\SPDIR\CRETB.SP’;
dmSQL> CREATE PROCEDURE FROM ‘D:\DATABASE\SPDIR\CRETB.SP’;
```

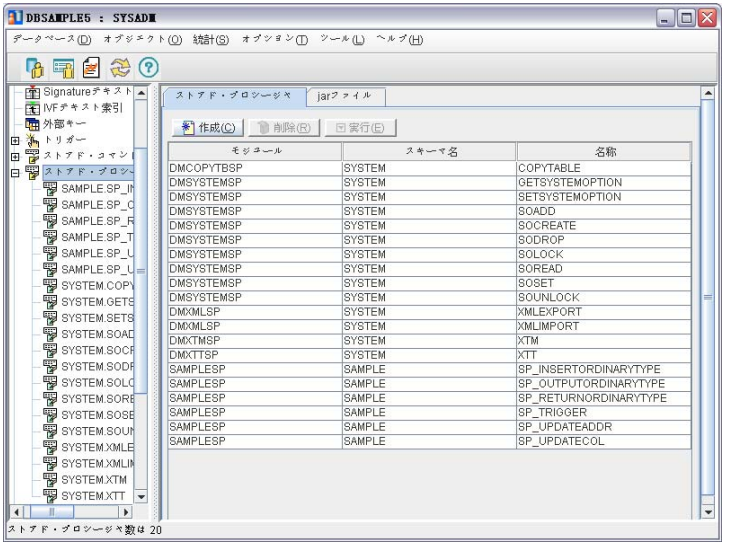
以上の例はdmSQLツールでファイルからSQLストアド・プロシージャをどう作成することを表示されます。

## JDBAで作成

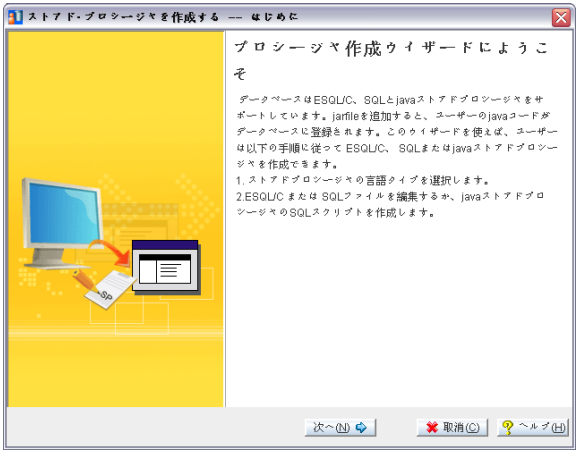
SQLストアド・プロシージャを作成するためDBMasterは三つの言語を提供します：SQL, ESQL/CとJava。以下の図はSQL言語でストアド・プロシージャを作成するのです。

➲ SQLストアド・プロシージャを作成する

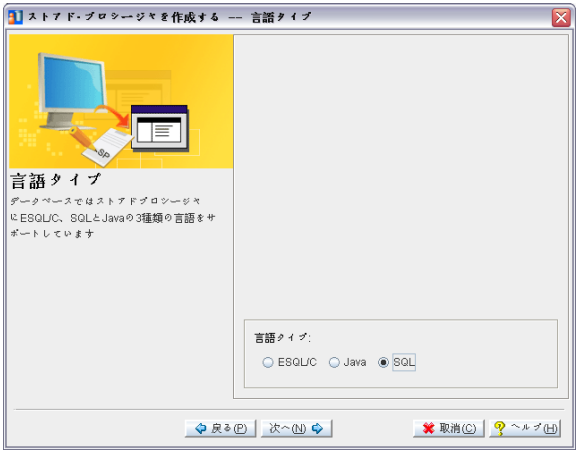
1. ツリーの**ストアド・プロシージャ**をクリックします。**ストアド・プロシージャ**ページが表示されます。




2. **［作成］**ボタンをクリックします。**［ストアド・プロシージャを作成する］**ウィザードが表示されます。

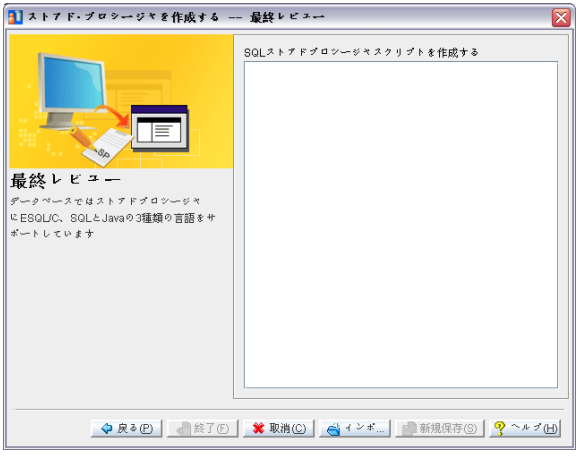



3. **「次へ」**をクリックすると、**「言語の種類」**のウィザードが表示されます。



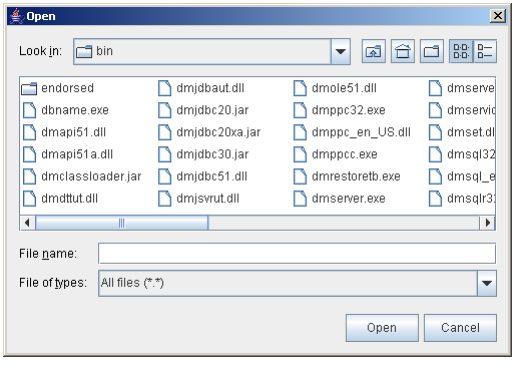

4. SQLラジオ・ボタンをクリックしてストアド・プロシージャを記述するSQL言語を選択します。

5. **「次へ」**ボタンをクリックすると、最終レビュー・ウィンドウが現れます。SQL文を入力してまたは**「インポート・ボタン」**をクリックしてファイルからSQL文をインポートします。



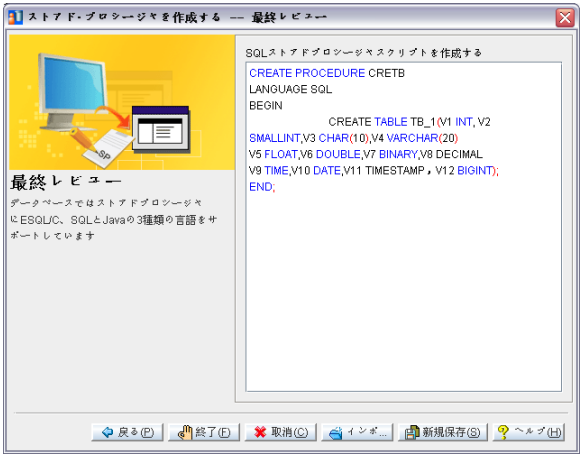

6. **［インポート］**ボタンを選択すると、**［開く］**ウィンドウが開きます。サーバー、或いはネットワーク上のドライブの他のデータベースのSPDIRディレクトリを含むいずれかのソースに、インポートするファイルがあります。**［ファイル名］**でパスを入力、又はディレ

クトリ・ツリーからパスを見つけてインポートするファイルを選択します。




**7.** ［開く］ボタンをクリックします。

**8.** ファイルに正しいフォーマット（ASCII）のテキストがインポートされた場合、またはコードの手動入力を選択した場合、下の例に**「最終レビュー」**のウィンドウが再表示されます。**「名前を付けて保存」**ボタンをクリックしてストアド・プロシージャを他の場所に保存するか、**「完了」**をクリックし、ストアドプロシージャをコンパイルしてデータベースに保存してください。




**9.** SQLストアド・プロシージャが適切にコンパイルされたら、**［情報］**メッセージが表示されます。

10. ［OK］をクリックします。

# 6.2 SQLストアド・プロシージャ

SQLストアド・プロシージャはDBMasterコマンドライン（dmSQLツール）からの文の実行を通じて、または直接にグラフィカル・ユーザー・インタフェース・ツール（JDBAツール）でCREATE PROCEDURE文を使用して実行できます。

## SQLストアド・プロシージャ構文を実行する

CALL文はプロシージャをコールします。この文はアプリケーション・プログラムに埋め込まれて、又は動的なSQL文を通じて実行されます。これは動的に準備される実行可能な文です。

ユーザーはdmSQLツールにストアド・プロシージャが呼び出せます、或いはトリガーを使ってコールします。SQLストアド・プロシージャにCall PROCEDURE文を実行する権限がなければなりません。

➲ 構文: CALL SQLストアド・プロシージャの構文 :

```
<CALL SQL stored procedure> ::=
CALL <procedure_name> [<variable_name> [ { <comma>
<variable_name> }... ]]
```

➲ 例 1:別のSQLストアド・プロシージャをコールする :

```
CREATE PROCEDURE call_test
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
                OPEN cur;
END;

CREATE PROCEDURE call1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
```

```
        CALL CASE_TEST_1(inval, outval1, outval2);
END;
```

➲ 例 2:別のESQLストアド・プロシージャをコールする：

```
EXEC SQL CREATE PROCEDURE ecret(INT ct output) RETURNS STATUS;
{
EXEC SQL BEGIN CODE SECTION;
        EXEC SQL SELECT count(*) FROM call_tb INTO :ct;
        exec sql RETURNS STATUS SQLCODE;
        EXEC SQL END CODE SECTION;
}

CREATE PROCEDURE call2(OUT st INT, OUT i2 INT)
LANGUAGE SQL
BEGIN
                SET st = CALL ecret(i2);
END;
```

➲ 例 3:JAVAストアド・プロシージャをコールする：

```
CREATE PROCEDURE CALLJAR
LANGUAGE SQL
BEGIN
        CALL INSERTS_3;        #  INSERTS_3 is a java sp
END;
```

➲ 例 4:他のストアド・プロシージャをカーソルとしてコールする：

```
CREATE PROCEDURE call3(OUT i1 INT, OUT i2 INT)
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR FOR call call_test;
                OPEN cur;
                FETCH FROM CUR INTO i1, i2;
                CLOSE cur;
END;
```

➲ 例 5:他のストアド・プロシージャを結果セットとしてコールする：

```
CREATE PROCEDURE call4
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR call call_test;
                OPEN cur;
END;
```

➲ 例 6:コマンドラインツールでストアド・プロシージャをコールする：

```
dmSQL> create table tb_1(name char(20),phone char(20));
dmSQL> select * from tb_1;
        NAME                 PHONE
==================== ====================

0 rows selected
dmSQL> CREATE PROCEDURE FROM 'D:\SPDIR\INSERT_1.SP';
dmSQL> CALL INSERT_1;
dmSQL> SELECT * FROM TB_1;

        NAME                 PHONE
====================    ====================
JONTH                   1234567

1 rows selected
```

➲ 例 7:標準ODBCプログラムにSQLストアド・プロシージャをコールする：

```
SQLPrepare(cmdp,(UCHAR*)"call proc1(?, ?)", SQL_NTS);

SQLBindParameter(cmdp, 1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                 20, 0, &n1, 20, NULL);

SQLBindParameter(cmdp, 2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                 20, 0, &n2, 20, NULL);
```

```
SQLBindCol(cmdp, 1, SQL_C_LONG, &i, sizeof(long), NULL);
SQLExecute(cmdp); /* get n2 */

while ((rc=SQLFetch(cmdp))!=SQL_NO_DATA_FOUND) /* fetch result set */
```

## トリガーでSQLストアド・プロシージャを実行

ユーザーはトリガーでSQLストアド・プロシージャをコールすることができます：まず、表を作成してください、例えばtb_2(name char(20),phone char(20))、そして、tb_2にトリガーを作成します。

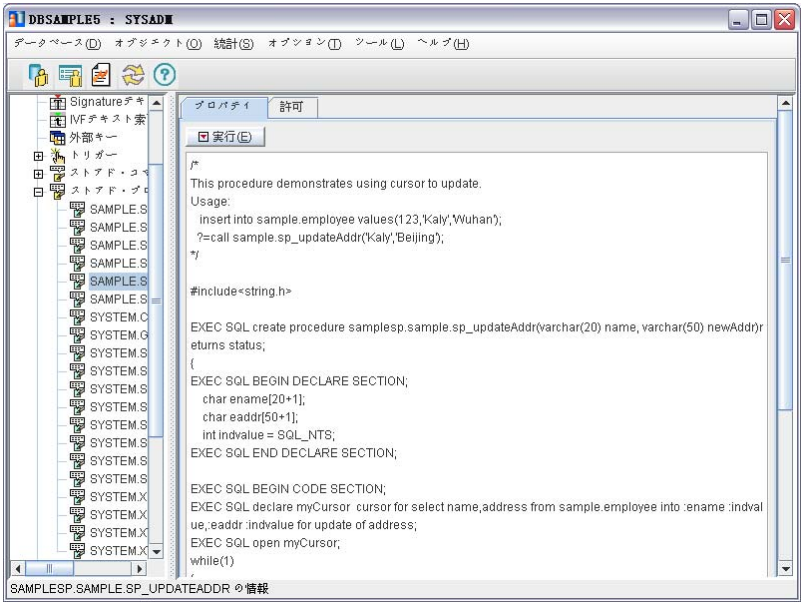
```
CREATE TRIGGER TRG_1 AFTER INSERT ON TB_2 FOR EACH ROW (CALL INSERT_1);
```

## JDBA ツールで使用

SQLストアドプロシージャを作成した後、直接に実行できて又はアプリケーションプログラムにコールできます。

➲ **SQL**ストアドプロシージャを実行する：

1. **［ストアド・プロシージャ］**ノードを開いて、SQLストアド・プロシージャを選択します。SQLストアド・プロシージャの**［プロパティ］**ページが開きます。




**NOTE** *右側のパネルのストアド・プロシージャをダブルクリックしても、同じウィンドウが表示されます。*

2. ［**実行**］ボタンをクリックします。実行されたSQLストアド・プロシージャの結果が表示されます。
3. ［OK］ボタンをクリックします。

# 6.3 SQLストアド・プロシージャを削除

ユーザーはJDBAツール或いはdmSQLコマンド・ライン・ツールでSQLストアド・プロシージャが削除できます。

## dmSQLで削除する

➲ 構文:SQLストアド・プロシージャを削除する構文 :

```
<DROP SQL stored procedure> ::=
DROP PROCEDURE <procedure_name>
```

➲ 例：dmSQLコマンドラインツールでSQLストアド・プロシージャを削除する：

```
dmSQL>  DROP PROCEDURE PROC1;
dmSQL>  DROP PROCEDURE USER1.PROC1;
```

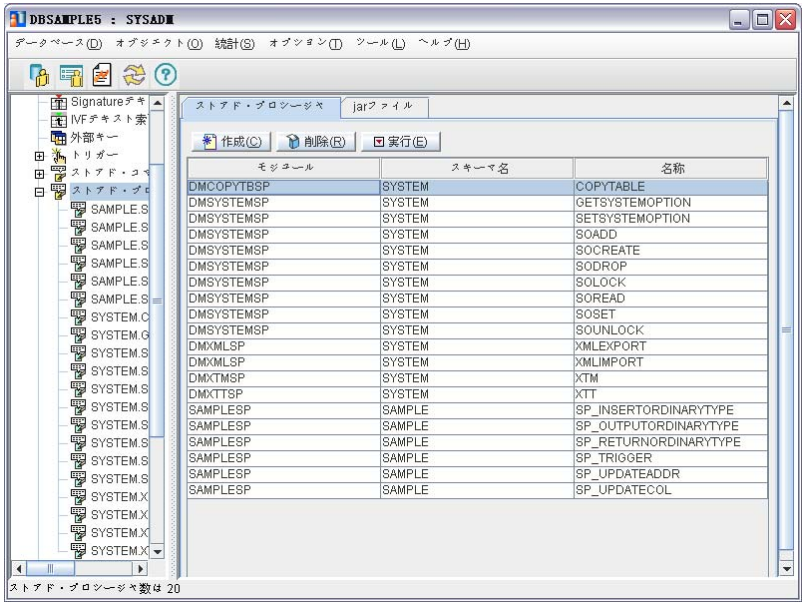
一番目の構文はストアド・プロシージャproc1を削除して、二番目の構文はストアド・プロシージャuser1.proc2を削除します。

## JDBAで削除する

必要の無くなったSQLストアド・プロシージャをJDBAツールで削除することができます。

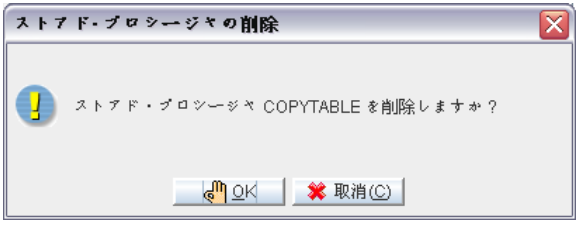
➲ 例：SQLストアドプロシージャを削除する：

**1.** ツリーの〔**ストアド・プロシージャ**〕オブジェクトをクリックします。データベースにある全ストアド・プロシージャが表示されます。




**2.** 削除するSQLストアド・プロシージャを選択します。

3. ［削除］ボタンをクリックします。ストアド・コマンドの削除を確認するため、［ストアド・コマンドの削除］ダイアログボックスが表示されます。




4. ［OK］ボタンをクリックします。データベースに残っているストアド・コマンドが表示されます。

# 6.4 SQLストアド・プロシージャ情報を取る

ユーザーはdmSQLツールでSYSPROCINFOシステム表を間合せることを通じてSQLストアド・プロシージャ情報を取ります。

➲ 例：

```
dmSQL> select * from sysprocinfo
```

# 6.5 安全管理

ストアド・プロシージャの所有者とDBA権限以上のユーザーがストアド・プロシージャを実行できます。他のユーザーまたはユーザー所属グループに実行権限を与えて、プロシージャを実行できます。ストアド・プロシージャの所有者とDBA権限以上のユーザーだけほかのユーザーにEXECUTE PROCEDURE権限を与えます。

➲ 構文：GRANT EXECUTE構文：

```
<GRANT EXECUTE syntax> ::=

GRANT EXECUTE ON <COMMAND | PROCEDURE> <executable_name> TO

< <user_name> | <group_name> | <PUBLIC> > [ { <comma> < <user_name> |
<group_name> | <PUBLIC> > }... ]
```

所有者とDBA権限以上のユーザーは他のユーザーの実行権限を取り消すこともできます。

構文: REVOKE EXECUTE 構文

```
<REVOKE EXECUTE syntax> ::=

REVOKE EXECUTE ON <COMMAND | PROCEDURE> <executable_name> FROM

<<user_name> | <group_name> | <PUBLIC>> [ { <comma> [<user_name> |
<group_name> | <PUBLIC>] }... ]
```

➲ 例 1:

**user1**は**proc1**というプロシージャを作成して、又はdmSQLを使用して**user2**に実行権限を与えます。

```
dmSQL> GRANT EXECUTE ON PROCEDURE proc1 TO user2;
```

➲ 例 2:

**user1**は**proc1**というプロシージャを作成して、又はdmSQLを使用して**PUBLIC**に実行権限を与えます。

```
dmSQL> GRANT EXECUTE ON PROCEDURE proc1 TO PUBLIC;
```

➲ 例 3:

**user1**はdmSQLで**user2**の実行権限を削除する：

```
dmSQL> REVOKE EXECUTE ON PROCEDURE proc1 FROM user2;
```

➲ 例 4:

**user1**はdmSQLで**PUBLIC**の実行権限を削除する：

```
dmSQL> REVOKE EXECUTE ON PROCEDURE proc1 FROM PUBLIC;
```

## 6.6 SQLストアド・プロシージャの構成設置

ストアド・プロシージャを作成する同時に相応な動的リンク・ライブラリも作成されてサーバーに保存されます。初期にライブラリ・ファイルはサーバ・ディレクトリに置いてます。データベース管理者はキーワード**DB_SPDir**を使用してストアド・プロシージャのライブラリ・ファイルに違ったパスを指定できます。

ストアド・プロシージャを作成/実行する場合、クライアント側のユーザーはキーワードDB_SPLogを使用してディレクトリを指定します。このディレクトリはサーバーから戻りエラー・メセッジとログ・ファイルを受け入れるため使用されます。

➲ 例 1:

dmconfig.iniファイルに、ストアド・プロシージャの動的リンク・ライブラリ・ファイルに初期パスを/usr1/dbmaster/data/SPに指定してください。

```
/usr1/DBMaster/data/SP：
DB_SPDIR=/usr1/DBMaster/data/SP
```

➲ 例 2:

dmconfig.iniファイルに、ストアド・プロシージャのログ・ファイル・ディレクトリをc:\usr\jerry\data\SPに指定します。

```
c:\usr\jerry\data\SP：
DB_SPLOG=c:\usr\jerry\data\SP
```

# 7 SQLストアド・プロシージャの移行

ストアド・プロシージャを他の DB に移行するため、ユーザーはアンロード/ロード・コマンドを使用できます。

## 7.1 アンロード/ロード・プロシージャ

```
dmSQL>unload db to dbsample5;
```

### UNLOAD [PROC | PROCEDURE]

```
dmSQL> unload procedure from call to 'd:\spdir\call';
```

以上のコマンドを完成した後、システムは d:¥spdir¥...ディレクトリに call.b0 ファイルと call.so ファイルを生成します。call.b0 はデータを保存して、call.s0 はスクリプトを保存します。

### LOAD [PROC | PROCEDURE]

```
dmSQL> load procedure from 'd:\spdir\call';
```

以上のコマンドを完成した後、システムは外部ファイルからストアドプロシージャをロードします。

# 8 SQLストアド・プロシージャの制限

DBMaster SQLストアド・プロシージャは幾つの制限があります：

1. 簡単表現はNULLがあると、全ての表現はNULLになります。比較表現はNULLがあると、この表現はfalseになります。
2. 0xフォーマットで表示する16進のデータフォーマット（0xはC言語の表現方式です）をサポートしませんが、10進と指数のデータフォーマットをサポートします。

➲ 例1：以下のはサポートしません：

```
SET d1 = 0x1A;
SET d2 = 0x124C;
```

➲ 例2：以下のはサポートします：

```
SET d1 = 12.3;
SET d2 = 1.2345E2;
```

3. SQLストアド・プロシージャの名に*project_name.module_name*のような形をサポートしません、*user.SYSPROCINFO*表にMODULENAMEフィールドはNULLです。

4. SET 文でキャラクタ変数に値を割り当てる場合、長さは変数が定義されたのを超えると、切り捨てられます。

➲ 例： c1の値は12345になると：

```
DECLARE c1 char(4);
SET c1 = '1234';
```

5. SET 文で変数に値を割り当てると、SET 文は変数のタイプにチェックします。違う属性タイプの間に値を割り当てることができません、つまり、キャラクタデータを数字変数に設定できません。逆もまた同じです。数字の間に、小数を整数に値を設定すると、自動的に小数部分を切り捨てます。

6. ストリングはシングルクォーテーションマーク "' " だけで囲みます。以下の規則を従います：

- ストリングに二つの "' " があると、その中の一つはストリングの内容だと表示されます。

➲ 例：

```
SET c1 = '12345''6789';  -- c1 = 12345'6789
```

- ストリングに何のダブルクォーテーションマークを含んでもいいです。これはストリングの部分で、ストリングのデリミタではありません。

➲ 例：

```
SET c1 = '12"34"56"78"9';  -- c1 = 12"34"56"78"9
```

- ストリングにバックスラッシュ"\"が含めます。

➲ 例：

```
SET c1 = '12345\6789';  -- c1 = 12345\6789
```

しかし、バックスラッシュの後のキャラクタはコメントシンボル "#"なら、コメントシンボルを普通なキャラクタと考えられます。

➲ 例:

```
SET c1 = '12345\#6789';  -- c1 = 12345#6789
```

- ストリングにキャリッジリターンを含むことができます。

➲ 例:

```
SET c1 = '12345
6789';
```

それから

```
c1 = 12345
6789
```

- キャリッジリターンの前にバックスラッシュ"\"がある場合、キャリッジリターンを捨てるという意味です。同時に上と下の2行を接続します。

➲ 例:

```
SET c1 = '12345\
6789';
```

それから

```
c1 = 123456789
```

**7.** *SET n1, n2, …. = select c1, c2, … from t1*構文をサポートします。実行中に*select*の一番目のレコードだけを取ります。もし割り当てた変数ナンバーとフィールド数が同等ではなければ、以下の規則に従います：変数ナンバーはｎ個があって、結果セットはm個があると、またn<=m、それから結果セットのｎ個の前を取ります。そうではなければ、全ての結果セットを取って、余分な変数値をNULLにします。同時に結果セットのフィールドタイプは相応な変数タイプとマッチするかをチェックします。マッチしないと、エラーメッセージが現れます。

常用な方法は*count ()*を使用して計算するものです。

➲ 例：

```
SET sum = select count(*) from t1;
```

8. 係数オペレーターはSQL function **MOD ()**を使用して実行します。

9. dmconfig.iniファイルのキーワード**DB_SPLog**はクライアント側だけで有効です。SQLストアド・プロシージャのTRACE情報に無効です。TRACEはディフォルトパスだけを受け入れて、つまりサーバー側でDBMasterの実行ディレクトリです。情報をディフォルトにこのディレクトリの**_sptrace.log**ファイルに書き入れます。

10. SQLストアド・プロシージャのスクリプトは何の拡張子でもいいです。さらに拡張子がなくてもOKです。

11. SQLストアド・プロシージャ文に変数名と同じなテーブル名またはフィールド名がある場合、変数名と区別するため、テーブル名またはフィールド名をダブルクォーテーションマーク“”で引用される必要があります。

➲ 例：

```
DECLARE c1 INT;
Select “c1” from t1 where “c1” > c1;
```

12. 以下のコマンドを除いて、全てのSQLコマンドをサポートします；

ABORT BACKUP

BEGIN BACKUP

BEGIN WORK

END BACKUP

13. SQLストアド・プロシージャを実行中にWARNING と ERRORが発生したら、デフォルトにWARNINGが無視されて実行続いています。最後のWARNING情報をユーザーに返します。ERRORを起す場合、

実行が終了されてエラーメッセージをユーザーに返します。SQLストアド・プロシージャに同時にWARNING と ERRORが起こしたら、デフォルトにERRORを返します。

**14.** 以下の確定できない状況をサポートしません：

➲ 例： intc1, intMax, charc2, charc3 全ては変数です：.

```
INSERT INTO mytb(c1, c2) VALUES(intc1,
     CASE
        WHEN intc1 <= intMax THEN charc2
     ELSE charc3
END);
```

**15.** *set client_char_set 'big5'*のような構文をサポートしません。ユーザーは動的なSQLを使用してこれを実現します。

**16.** (may have intersection).以下のはSQLストアド・プロシージャの保留キーワードです。変数名は以下のキーワードと“*SQLコマンドと関数参照編*”の保留キーワードが設定できません。（交差があるかもしれません）。

ADD, ALTER, AND, AS, ASENSITIVE, BEGIN, BIGINT, BINARY, BREAK, CALL, CASE, CHAR, CLOSE, COMMIT, CONDITION, CONNECT, CONT, CONTINUE, CREATE, CURSOR, DATE, DEALLOCATE, DECIMAL, DECLARE, DEFAULT, DELETE, DISCONNECT, DO, DOUBLE, DROP, DYNAMIC, ELSE, ELSEIF, END, EXECUTE, EXIT, FALSE, FETCH, FIRST, FLOAT, FOR, FOUND, FROM, GO, GOTO, GRANT,HANDLER, HOLD, IF, IMMEDIATE, IN, INOUT, INPUT, INSENSITIVE, INSERT, INT, INTEGER, INTO, IS, ITERATE, LANGUAGE, LAST, LEAVE, LOOP, NCHAR, NCLOB, NEXT, NO, NOT, NULL, NVARCHAR, OFF, ON, OPEN, OR, OUT, UTPUT, PREPARE, PRIOR, PROCEDURE, REPEAT, RESULT, RETURN, RETURNS, ROLLBACK,SCROLL, SELECT, SENSITIVE, SET, SETS, SHORT, SMALLINT, SQL, SQLCODE,

SQLEXCEPTION, SQLSTATE, SQLWARNING, TATISTICS, STOP, THEN, TIME, TIMESTAMP, TO, TRACE, TRUE, UNTIL, UPDATE, USING, VALUE, VARBINARY, VARBPTR, VARCHAR, VARCPTR, WHEN, WHILE, WITH, WITHOUT

17. SQLストアド・プロシージャにバックスラッシュ'\'が使用できます。'\'にキャリッジリターンがあると、'\'を捨てるという意味です。同時に上と下の2行を接続します。

18. INOUT引数をサポートしません。