

Enterprise COBOL for z/OS
6.4

パフォーマンス・チューニング・ガイド



注

本書および本書で紹介する製品をご使用になる前に、[75 ページの『特記事項』](#)に記載されている情報をお読みください。

第 3 版 (2022 年 5 月 27 日更新)

この版は、新しい版で別途指示されるまで、z/OS® の Language Environment®・コンポーネントのバージョン 2.1 で稼働する IBM® Enterprise COBOL (プログラム番号 5655-EC6) のバージョン 6.4 とそれ以降のすべてのリリースおよび修正に適用されます。

ソフトコピー資料は、Enterprise COBOL for z/OS ライブラリーから無料で参照またはダウンロードできます。Enterprise COBOL for z/OS は継続的デリバリー (CD) モデルをサポートしており、資料は CD モデルで配布されるフィーチャーを記述するために更新されるので、2 カ月ごとに更新の有無を確認することをお勧めします。

このリリースに関する製品資料は、注文番号を更新せずに定期的に更新する予定です。製品資料の版を個別に参照する必要がある場合は、更新日付で注文番号を参照してください。

© Copyright International Business Machines Corporation 1993, 2022.

目次

表.....	vii
前書き.....	ix
この情報について.....	ix
パフォーマンス測定.....	ix
変更の要約.....	ix
Enterprise COBOL for z/OS 6.4.....	x
ご意見の送付方法.....	x
第 1 章 Enterprise COBOL 6 で再コンパイルする理由.....	1
アーキテクチャーの利用.....	1
高度な最適化.....	3
機能の拡張.....	4
第 2 章 COBOL 6 への移行のためのアプリケーションの優先順位付け.....	5
COMPUTE.....	5
INSPECT.....	8
MOVE.....	10
SEARCH.....	10
テーブル.....	11
条件式.....	11
第 3 章 COBOL 6 を最大限に活用するためのコンパイラー・オプションのチューニング方法.....	13
AFP.....	13
ARCH.....	14
ARITH.....	15
AWO.....	16
BLOCK0.....	16
DATA(24) および DATA(31).....	17
DYNAM.....	17
FASTSRT.....	18
HGPR.....	18
INLINE.....	19
INVDATA.....	19
MAXPCF.....	21
NUMCHECK.....	22
NUMPROC.....	22
OPTIMIZE.....	23
PARMCHECK.....	23
SSRANGE.....	24
STGOPT.....	24
TEST.....	25
THREAD.....	26
TRUNC.....	26
TUNE.....	27
プログラムの常駐とストレージの考慮事項.....	28
第 4 章 ランタイム・パフォーマンスに影響するランタイム・オプション.....	31

AIXBLD.....	31
ALL31.....	31
CBLPSHPOP.....	32
CHECK.....	33
DEBUG.....	33
INTERRUPT.....	33
RPTOPTS.....	34
RPTSTG.....	34
RTEREUS.....	35
STORAGE.....	35
TEST.....	37
TRAP.....	37
VCTRSAVE.....	38
第 5 章ランタイム・パフォーマンスに影響する COBOL および LE の機能.....	39
ストレージ管理チューニング.....	39
ストレージ・チューニング・ユーザー出口.....	40
CEEENTRY マクロおよび CEETERM マクロの使用.....	40
事前初期設定サービス (CEEPIPI) の使用.....	41
ライブラリー・ルーチン保存機能 (LRR) の使用.....	41
LPA/ELPA 内のライブラリー.....	42
CALL の使用.....	42
PROGRAM-ID ステートメントまたは INITIAL コンパイラー・オプションでの IS INITIAL の使用.....	43
PROGRAM-ID ステートメントでの IS RECURSIVE の使用.....	43
第 6 章ランタイム・パフォーマンスに影響するその他の製品関連要因.....	45
ILC アプリケーションにおける 10 進オーバーフローへの影響.....	45
最初のプログラムが LE に非準拠.....	46
CICS.....	46
Db2.....	48
DFSORT.....	48
IMS.....	48
LLA.....	49
第 7 章 COBOL 6 を最大限に活用するためのコーディング技法.....	51
BINARY (COMP または COMP-4).....	51
DISPLAY.....	53
PACKED-DECIMAL (COMP-3).....	53
固定小数点と浮動小数点.....	54
一括表示表現.....	54
シンボリック定数.....	55
Occurs Depending On テーブルのパフォーマンス・チューニング考慮事項.....	55
PERFORM の使用.....	55
QSAM ファイルの使用.....	57
可変長ファイルの使用.....	57
HFS ファイルの使用.....	58
VSAM ファイルの使用.....	58
第 8 章プログラム・オブジェクト・サイズと PDSE 要件.....	61
COBOL 4 と COBOL 6 間のロード・モジュール・サイズの変更.....	61
ディスク上のプログラムのオブジェクト・サイズに対する SMARTBIN オプションの影響.....	61
プログラム・オブジェクト・サイズに対する TEST サブオプションの影響.....	62
COBOL 6 で実行可能ファイルに PDSE を使用するのなぜですか？.....	64
付録 A Automatic Binary Optimizer を使用して COBOL アプリケーションのパフォーマンスを改善する.....	67

付録 B 組み込み関数の実装に関する考慮事項.....	69
付録 C Enterprise COBOL for z/OS のアクセシビリティ機能.....	73
特記事項.....	75
商標.....	77
特記事項.....	77
用語集.....	79
リソース・リスト.....	123
Enterprise COBOL for z/OS.....	123
関連資料.....	123

表

1. ARCH レベル別の平均向上率.....	14
2. ARCH の設定とハードウェア・モデル.....	15
3. 以前の COBOL バージョンからマイグレーションする場合の INVDATA オプションと NUMPROC オプションの設定.....	20
4. NOTEST に対する TEST(NOEJPD) または TEST(EJPD) のパフォーマンス低下.....	25
5. TRUNC(STD) 指定時の 4 つのテスト・ケースのパフォーマンス差の結果.....	51
6. TRUNC(BIN) 指定時の 4 つのテスト・ケースのパフォーマンス差の結果.....	52
7. TRUNC(OPT) 指定時の 4 つのテスト・ケースのパフォーマンス差の結果.....	52
8. 異なるアクセス・モードでの CPU 時間、経過時間、および EXCP カウント.....	58
9. SMARTBIN と比較した場合の NOSMARTBIN のサイズ縮小率.....	62
10. NOTEST(NODWARF) と比較した NOTEST(DWARF) のサイズ増加率 (%).....	63
11. NOTEST と比較した TEST のサイズ増加率 (%).....	63
12. TEST(NOSOURCE) と比較した TEST(SOURCE) のサイズ増加率 (%).....	63
13. TEST(NOEJPD) と比較した TEST(EJPD) のサイズ増加率 (%).....	63
14. 組み込み関数の実装.....	69

前書き

この情報について

本書では、Enterprise COBOL 6.4 を使用する際の重要なパフォーマンス上の利点とチューニングに関する考慮事項について説明します。

本書では、「COBOL」または「Enterprise COBOL」とは「IBM Enterprise COBOL for z/OS」または「IBM Enterprise COBOL Value Unit Edition for z/OS」を指します。

最初に COBOL 6 のコンパイラーの主なパフォーマンス・フィーチャーとオプションの概要を示し、続いていくつかの特定 COBOL ステートメントのパフォーマンス改善について説明します。次に、COBOL アプリケーションのパフォーマンスに影響する数多くのコンパイラー・オプションおよびランタイム・オプションに関するチューニング考慮事項を取り上げます。さらに、COBOL 6 使用時の変更されたコーディング推奨事項に注目しながら、最良のパフォーマンスを得るためのコーディング技法について検討します。

最後のセクションでは、プログラム・オブジェクトのサイズが増加したいくつかの原因について考察し、さまざまな新しい TEST サブオプションがオブジェクト・サイズに与える影響を検討し、COBOL 6 を使用してコンパイルされたプログラムで PDSE が必要な理由に関連した問題について説明します。

COBOL 5 のコンパイラーのパフォーマンス特性は、COBOL 6 に類似しています。特に注釈がある場合を除き、本書の情報および推奨事項は COBOL 5 にも該当します。COBOL 5 から COBOL 6 への移行で必要だったパフォーマンス・チューニングの変更は少なかったため、本書の推奨事項および比較は、読者が COBOL 4 のコンパイラーから移行することを前提としています。

ハードウェアの進化に伴い、COBOL 6 生成コードは常に向上しています。COBOL 4 生成コードも向上することがあり、場合によっては COBOL 6 よりも大きく向上します。このため、COBOL 4 生成コードと比べた COBOL 6 生成コードの高速化は、ハードウェアの新世代ごとに単純増加しているわけではありません。

パフォーマンス測定

本書のパフォーマンス測定は、特に記載がない限り、z16 システムで COBOL 6.4 を使用して行われたものです。使用されたプログラムはバッチ・タイプ (非対話式) のアプリケーションです。特に記載がない限り、本書で行われたパフォーマンス比較はすべて、経過時間のパフォーマンスではなく、CPU 時間のパフォーマンスを指しています。

注: 特に明記されている場合を除き、本書におけるパフォーマンス比較は、マイクロベンチマークを使用して測定されています。各マイクロベンチマークは、特定領域でのコンパイラーのパフォーマンス向上を強調するように設計されています。マイクロベンチマークは、特定の COBOL ステートメントに対して生成される命令のパフォーマンスを強調して、コンパイラー・チームが特定の COBOL ステートメントに最適な命令セットを選択する際に役立つように作成されています。これらは、現実世界のアプリケーションの一般的なパフォーマンスを示すものではなく、現実世界の COBOL アプリケーションの一般的なパフォーマンスの指標として使用してはなりません。実際のプログラムでは一般に、機能が混用されています。プログラムがその時間の大部分を、コンパイラーの向上によるパフォーマンス向上の影響を受けるコードの実行に費やす場合、プログラムで確認できるのは、そのような向上のみになります。

注: カスタマー・アプリケーションのパフォーマンス結果は、ソース・コード、コンパイラー・オプションの指定、およびその他の要因によって異なります。

変更の要約

このセクションには、Enterprise COBOL for z/OS 6.4 で行われた本書の主な変更点を記載しています。本書で解説されている変更には、読者の便宜のため、参照ページが記載されています。最新の技術的な変更には、HTML バージョンでは >| と |< のマークが付いています。PDF バージョンでは左マージンに縦線 (|) がマークされています。

Enterprise COBOL for z/OS 6.4

アーキテクチャーの利用

- 新しい上位レベルの ARCH(14) および TUNE(14) が受け入れられます。新しいハードウェア機能である、新しい Vector Packed Decimal Enhancement Facility 2 が導入されました。詳しくは、[1 ページの『アーキテクチャーの利用』](#)を参照してください。

機能の拡張

「Enterprise COBOL for z/OS 移行ガイド」の『IBM Enterprise COBOL for z/OS 6.4 の変更点』には、Enterprise COBOL 6.4 の新機能と変更された機能の全リストが含まれています。以下のようなハイライトがあります。

- Java™/COBOL の相互運用性の向上により、COBOL アプリケーションの機能を Java で簡単に拡張できるようになりました。
- AMODE 31 (31 ビット) と AMODE 64 (64 ビット) COBOL プログラム間のインターオペラビリティ
- 独自の関数を作成し、組み込み関数のように呼び出すことができるようにするためのユーザー定義関数のサポート。これは、2002 COBOL Standard の一部です。
- IBM Automatic Binary Optimizer for z/OS (ABO) との統合が改善され、現在コンパイルしているモジュール (再コンパイルは不要) で将来の IBM Z® ハードウェアの機能拡張を利用できるようにする、将来に向けた投資を行えるようになりました。

COBOL 6.4 は、COBOL 6.3、6.2、6.1、および COBOL 5 で導入されたすべての新機能を引き続きサポートします。

ご意見の送付方法

本書または Enterprise COBOL の他のマニュアルについてご意見がありましたら、IBM 発行のマニュアルに関する情報の Web ページ (<http://www.ibm.com/jp/manuals/>) よりお送りください。今後の参考にさせていただきます。(URL は、変更になる場合があります) この情報またはその他の Enterprise COBOL の資料についてのコメントは、コメントを compinfo@cn.ibm.com に送信していただく。

マニュアルの名前、資料番号、Enterprise COBOL のバージョン、および必要な場合は ご意見のあるテキストの具体的な場所 (ページ番号やセクション見出しなど) を必ずご連絡ください。

お客様が IBM に情報を送る場合、お客様は、IBM がお客様に一切の義務を負わせることなく適切と信ずる方法で情報を使用もしくは配布することができる包括的権利を IBM に付与するものとします。

第 1 章 Enterprise COBOL 6 で再コンパイルする理由

Enterprise COBOL 6 には、Enterprise COBOL 5 および 4 に比べて多数の改善点があります。アプリケーションを再コンパイルすると、Enterprise COBOL 6 の高度な最適化と IBM z/Architecture® の活用機能が利用され、IBM Z での COBOL のパフォーマンスが向上します。COBOL 5 と比較すると、COBOL 6 コンパイラーの内部機能が拡張され、大規模なプログラムのコンパイルと最適化が可能になりました。COBOL 6 を使用することで、コード生成プログラムで作成された COBOL プログラムなどの、より大きなプログラムをコンパイルできるようになりました。

パフォーマンスの向上は、以下によってもたらされます。

- [1 ページの『アーキテクチャーの利用』](#)
- [3 ページの『高度な最適化』](#)
- [4 ページの『機能の拡張』](#)

アーキテクチャーの利用

COBOL 6 は、COBOL 5 で導入された ARCH オプション (アーキテクチャーを表す略語) を引き続きサポートします。このオプションを使用すると、新しいハードウェア命令を最大限に活用することができます。保守が適用された COBOL 6.3 以降、どのアーキテクチャー用に実行可能プログラムを最適化するかを指定できる TUNE オプションが導入されました。ハードウェアへの投資を最大限に活用するには、両方のオプションを使用します。

ARCH は、使用可能な命令の中からどの命令を選択するかをコンパイラーに指示します。例えば、ARCH(10) は、zEC12 または zBC12 にある命令しか使用してはいけないことを指定します。z13®、z13s®、またはそれ以降のハードウェアに追加された命令は使用されません。これにより、zEC12 または zBC12 に存在しない命令がプログラムに含まれないようにします。

ARCH によって、コンパイラーで選択できる命令を指定することはできますが、特定の COBOL ステートメントにどの命令を使用するかは、やはりコンパイラーが決定しなければなりません。このような選択は TUNE によって管理されます。これは、ARCH によって指定された使用可能な命令の制限の下で、特定のハードウェア・モデルにとって最適な判断を行うようにコンパイラーに指示します。

ARCH のデフォルト設定は 10 です。TUNE のデフォルト設定は ARCH と同じ値になります。サポートされるその他の値は、11、12、13、および 14 です。TUNE レベルは常に ARCH レベル以上でなければなりません。アプリケーションが実行される最も古いマシン (災害復旧 (DR) マシンを含む) のアーキテクチャーに一致するように ARCH 値を設定します。アプリケーションが最も頻繁に実行されるアーキテクチャーに一致するように TUNE 値を設定します。

各レベルで使用可能な機能や、これらの ARCH レベルと具体的なハードウェア・モデルの対応関係について詳しくは、「Enterprise COBOL for z/OS プログラミング・ガイド」の『ARCH』を参照してください。

TUNE レベルから特定のハードウェア・モデルへのマッピングの詳細については、「Enterprise COBOL for z/OS プログラミング・ガイド」の『TUNE』を参照してください。

一連の ARCH レベルはそれぞれ、コンパイラーがハードウェアのより多くの機能を利用することを可能にし、パフォーマンス向上の潜在能力を引き出します。COBOL アプリケーションの観点からの利点を説明するために、それぞれの ARCH レベルと TUNE レベルについて、以下で詳しく説明します。

ARCH(10)

ハードウェア機能: 10 進浮動小数点 (DFP) のパフォーマンスの向上

これが COBOL のパフォーマンスに重要である理由: 古いハードウェア・モデルでは、パック 10 進数演算および外部 10 進数演算のパフォーマンスを向上させるためにコンパイラーが使用できる DFP 命令が提供されていません。さらなる向上のために、ARCH(10) では、DISPLAY (具体的には、符号なしおよび後置符号付きのオーバーパンチ・ゾーン 10 進数) 型と DFP の間の変換を行う効率的な命令が追加されています。

周囲の状況が最適で、かつ最適化レベルが 0 より大きい場合、これらの ARCH(10) 命令は、ゾーン 10 進数データ項目の算術演算に DFP を使用することによるオーバーヘッドを低減し、コンパイラーで DFP を最大限に活用できるようにしてパフォーマンスを向上させます。

算術演算を実行するためにゾーン 10 進数データ項目をパック 10 進数形式に変換する代わりに、コンパイラーはゾーン 10 進数データを DFP 形式に直接変換し、計算の完了後に再度ゾーン 10 進数形式に戻します。DFP 命令は、多くの場合ハードウェアでより効率的に処理されるレジスター内 (メモリー内に対比) データに対して実行されるため、一般的にパフォーマンスが向上します。

ARCH(10) は、zEnterprise® EC12 および zEnterprise BC12 で最高のパフォーマンスを発揮します。

ARCH(11)

ハードウェア機能: パック 10 進数と 10 進浮動小数点 (DFP) 間の改善された変換

これが COBOL のパフォーマンスに重要である理由: ARCH(10) では、コンパイラーが DISPLAY 型と DFP の間の変換をより効率的に実行できるので、コンパイラーで DFP を活用してパック 10 進数演算および外部 10 進数演算のパフォーマンスを向上させることができます。ARCH(10) にはパック 10 進数と DFP の間の変換を行う命令がありましたが、非効率的で、パック演算を DFP で行う利点より、DFP との間でパック 10 進数値を変換するコストの方が上回っていました。

ARCH(11) には、パック 10 進数と DFP をより効率的に変換する新しい命令があります。これらの命令により、周囲条件が最適なものであって、最適化レベルがゼロより大きい場合、パック 10 進数データ項目に対して DFP 算術演算を使用する際のオーバーヘッドが低減し、コンパイラーは DFP をさらに活用できます。

パック 10 進数データ項目に対して算術演算を実行する代わりに、コンパイラーはパック 10 進数データを DFP 形式に変換し、計算の完了後に再度パック 10 進数形式に戻します。DFP 命令は、多くの場合ハードウェアでより効率的に処理されるレジスター内 (メモリー内に対比) データに対して実行されるため、一般的にパフォーマンスが向上します。変換命令の効率が向上したため、DFP で算術演算を実行することによる利点が、パック算術演算を直接実行する代わりにパック 10 進数と DFP を変換することで生じるコストを上回ります。

ハードウェア機能: ベクトル・レジスター

これが COBOL パフォーマンスに重要な理由: 新しいベクトル機能は、最大 16 バイトのサイズの要素に対して並列に作動できます。ARCH(11) では、COBOL 6 は、新しいベクトル命令を利用して、一度に 16 バイトを処理することにより、いくつかの形式の INSPECT ステートメントを高速化できます。この処理は、一時に 1 バイトずつ操作するよりもはるかに高速です。

ARCH(11) は、z13 および z13s で最高のパフォーマンスを発揮します。

ARCH(12)

ハードウェア機能: ベクトル・パック 10 進数命令

これが COBOL のパフォーマンスに重要である理由: ARCH(11) 以下では、パック 10 進数演算は、メモリー内データを使用するか、データを 10 進浮動小数点 (DFP) に変換しないと実行できません。ARCH(12) では、新しいベクトル・パック 10 進数機能により、コンパイラーはネイティブのパック 10 進数演算をレジスター内のデータに対して実行できるようになりました。これにより、メモリーの代わりにレジスターを使用するというパフォーマンス上の利点が得られ、さらにパック 10 進数と DFP の間でデータを双方向に変換するオーバーヘッドがなくなります。

ARCH(12) は、z14 および z14 ZR1 で最高のパフォーマンスを発揮します。

ARCH(13)

ハードウェア機能: 個々のベクトル・パック 10 進数命令でのハードウェア・オーバーフロー例外を抑止する機能

これが COBOL パフォーマンスにとって重要な理由: パック 10 進数オーバーフローが発生した場合、ハードウェアは何も行わずにオーバーフローを抑止することができます。これはデフォルトの COBOL の動作であるか、例外が発生することがあります。これはアプリケーション単位のハードウェア設定で制御されま

す。COBOL プログラムでの正しい動作はオーバーフロー例外を抑止することであるため、Enterprise COBOL プログラムはこの設定を変更しません。純粋な COBOL アプリケーションでは、すべてのオーバーフローがハードウェア・レベルで抑止されます。混合言語アプリケーションでは、他の言語でこの設定が有効化され、これにより例外が発生します。この設定はアプリケーション単位で行われるので、これにより COBOL プログラムも影響を受けます。例外は LE によって処理されます。その際、COBOL プログラムから生成された例外は選択的に抑止されますが、LE が関与することでパフォーマンスが犠牲になります。さらに、オーバーフローが少ない(または発生しない)プログラムの場合のように COBOL プログラムはこの設定をオン/オフしないので、これによってもパフォーマンスが犠牲になります。

ARCH(13) では、ARCH(12) で導入されたベクトル・パック 10 進数命令で、オーバーフローを抑止するかどうかを命令ごとに指定することができます。これによりハードウェアは、LE の関与なしで、また設定を変更する際のオーバーヘッドなしで、COBOL プログラムのオーバーフローを抑止できます。

ARCH(13) は、z15™ および z15 T02 で最高のパフォーマンスを発揮します。

ARCH(14)

ハードウェア機能: ベクトル・パック 10 進数拡張機能 2

これが COBOL のパフォーマンスに重要である理由: この新しい機能により、以下のタイプのステートメントを 1 つ以上含む COBOL プログラムのパフォーマンスが向上します。

- 指数が 1 桁以上の小数で宣言されているパック 10 進数またはゾーン 10 進数のデータ項目に対する指数演算
- 10 進数データ項目と浮動小数点データ項目が混在する算術ステートメント
- 数字編集データ項目を使用するステートメント

ARCH(14) は、z16 で最高のパフォーマンスを発揮します。

高度な最適化

Enterprise COBOL 6 は、アーキテクチャーを存分に活用することに加え、高度な最適化のスイートを使用することによって、アプリケーションのパフォーマンスを向上させます。COBOL 4 では OPTIMIZE オプションに 3 つの設定値がありましたが、6 で使用可能な最適化の種類と数は大きく異なります。

OPTIMIZE(1) または OPTIMIZE(2) を指定することで、さまざまな、一般および COBOL 固有の最適化が可能になります。

例えば、OPTIMIZE(1) を指定すると、以下のような最適化を行うことができます。

- 以下のような、複雑で負荷の高い演算の強度を下げる
 - 10 進数の 10 の累乗による乗算および除算を、より単純でパフォーマンスの良い 10 進数シフト演算に低減する
 - 2 進数の 2 の累乗による乗算および除算を、より負荷の低いシフト演算に低減する
 - 定数指数を持つ指数演算を一連の乗算に低減する
 - 算術演算をリファクタリングおよび再配分する
- 一般的な副次式を除去して、計算が重複しないようにする
- 非インライン PERFORM ステートメントをインライン化して、分岐オーバーヘッドを小さくし、周囲のコードにその他の最適化の機会を与える
- 定数値の順次ストアを 1 つのより大きいストアに合体し、パス長さを削減する
- 順次ストレージでのロード/ストアを 1 つのより大きい移動操作として合体し、パス長さおよびオブジェクト全体のサイズを削減する
- コードを単純化して不要な計算を除去する
- 到達不能コードを除去する
- 読み取られるが書き込まれないデータ項目に関して、VALUE OF 節リテラルをプログラム全体に伝搬する
- ネストされたプログラムをインラインに移動して、CALL オーバーヘッドを小さくし、周囲のコードにその他の最適化の機会を与える

- 定数式 (全算術演算、データ型変換、分岐など) をコンパイル時に計算する
- レベル 88 変数の条件付き設定に、よりパフォーマンスの良い、分岐のないシーケンスを使用する
- 一部のパック 10 進数およびゾーン 10 進数の計算を、よりパフォーマンスの良い 10 進浮動小数点型を使用するように変換する
- 小さい DISPLAY および COMP-3 項目の比較をメモリー内ではなくレジスター内で実行する
- 数字編集項目に移動するためのより高速なコードを生成する
- よりパフォーマンスの良いシーケンスを DIVIDE GIVING REMAINDER に使用する

OPTIMIZE(2) を指定すると、上記すべての他に、以下の最適化が可能になります。

- プログラム全体にわたって値および値の範囲を伝搬して、定数を公開し、より単純な命令シーケンスを使用できるようにする
- 「符号なし」符号エンコードなどの符号値をプログラム全体にわたって伝搬し、符号修正の重複を除去する
- 索引付きテーブルへのアクセス用のグローバル・レジスターの割り振りと、パス長さを削減するための PERFORM 'N' TIMES ループ構成の制御
- 重複する符号修正操作をグローバルに除去する。例えば、ループ内のデータ項目の符号修正が、同じデータ項目に対して 1 つ外側のループで制御されている場合、ループ内のこれらの符号修正命令は除去されます。

機能の拡張

COBOL 6 では、アーキテクチャーの開発と高度な最適化により、既存のプログラムに提供されるパフォーマンスの向上に加えて、いくつかの領域で拡張機能が提供されています。

Enterprise COBOL 6.4 のハイライト

「Enterprise COBOL for z/OS 移行ガイド」の『IBM Enterprise COBOL for z/OS 6.4 の変更点』には、Enterprise COBOL 6.4 の新機能と変更された機能の全リストが含まれています。以下のようなハイライトがあります。

- Java/COBOL の相互運用性の向上により、COBOL アプリケーションの機能を Java で簡単に拡張できるようになりました。
- AMODE 31 (31 ビット) と AMODE 64 (64 ビット) COBOL プログラム間のインターオペラビリティ
- 独自の関数を作成し、組み込み関数のように呼び出すことができるようになるためのユーザー定義関数のサポート。これは、2002 COBOL Standard の一部です。
- IBM Automatic Binary Optimizer for z/OS (ABO) との統合が改善され、現在コンパイルしているモジュール (再コンパイルは不要) で将来の IBM Z ハードウェアの機能拡張を利用できるようにする、将来に向けた投資を行えるようになりました。

COBOL 6.4 は、COBOL 6.3、6.2、6.1、および COBOL 5 で導入されたすべての新機能を引き続きサポートします。

第 2 章 COBOL 6 への移行のためのアプリケーションの優先順位付け

COBOL 6 への移行の取り組みを優先順位付けするために、このセクションでは、コンパイラの旧リリースと比較して、COBOL 5 および COBOL 6 で一般的にパフォーマンスが向上するいくつかの COBOL ステートメントおよびデータ型宣言について説明します。完全なリストを提供することではなく、COBOL 5 および COBOL 6 のパフォーマンスが確実に向上することがわかっているいくつかの事例を示すことを意図しています。

アプリケーションの正確性を維持するための移行に関する関連情報については、「Enterprise COBOL for z/OS 移行ガイド」の『アプリケーションの優先順位付け』を参照してください。

パフォーマンス測定はすべて、COBOL 4 でコンパイルされた同一のプログラムの、同一マシン・レベルでの実行と比較しています。すべての事例で、COBOL 4 プログラムは、OPTIMIZE(FULL) 指定、また他のオプションはデフォルト設定のままでコンパイルされています。ただし、含まれている桁数が 18 桁を超えるデータで ARITH(EXTEND) が必要であった場合を除きます。

COMPUTE

COBOL 6 では、多数の COMPUTE、ADD、SUBTRACT、MULTIPLY、DIVIDE の各ステートメントでパフォーマンスが向上しています。

下の例の「データ型」では、パフォーマンスについてテストされたバリエーションをイタリックで示していますが、リストされたデータ型はすべて同様のパフォーマンス結果を示しています。

大きい 10 進数の乗算/除算

ステートメント: COMPUTE (* | /)、MULTIPLY、DIVIDE

データ型: COMP-3、DISPLAY、NATIONAL

オプション: OPT(1 | 2)

条件: 中間結果がハードウェアのパック 10 進数命令の使用制限を超えた場合。これは、演算によって異なりますが、15 桁付近で発生します。

COBOL 4 の動作: ランタイム・ルーチン呼び出す

COBOL 6 の動作: パック 10 進数に変換し、ベクトル・パック 10 進数命令を使用した後にインライン化する

ソースの例:

```
1 z14v2 pic s9(14)v9(2)
1 z13v2 pic s9(13)v9(2)

Compute z14v2 = z14v2 / z13v2.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 81% 高速

ゾーン 10 進数 (DISPLAY) 算術演算

ステートメント: COMPUTE (+ | - | * | /)、ADD、SUBTRACT、MULTIPLY、DIVIDE

データ型: DISPLAY

オプション: OPT(1 | 2)、ARCH(10)

条件: すべての事例

COBOL 4 の動作: パック 10 進数命令を使用してインライン化する

COBOL 6 の動作: パック 10 進数に変換し、ベクトル・パック 10 進数命令を使用した後にインライン化する

ソースの例:

```
1 z12v2 pic s9(12)v9(2)
1 z11v2 pic s9(11)v9(2)
Compute z12v2 = z12v2 / z11v2
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 73% 高速

10 の累乗による除算 (10,100,1000,...)

ステートメント: COMPUTE (/)、*DIVIDE*

データ型: *COMP-3*、*DISPLAY*、*NATIONAL*

オプション: デフォルト

条件: 除数が 10 の累乗であること (例えば 10,100,1000,...)

COBOL 4 の動作: パック 10 進数除算 (DP) 命令を使用する

COBOL 6 の動作: 10 進数右シフトとしてモデル化する

ソースの例:

```
1 p8v2a pic s9(8)v9(2) comp-3
1 p8v2b pic s9(8)v9(2) comp-3

Compute p8v2b = p8v2a / 100
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 73% 高速

10 の累乗による乗算 (10,100,1000,...)

ステートメント: COMPUTE (/)、*MULTIPLY*

データ型: *COMP-3*、*DISPLAY*、*NATIONAL*

オプション: デフォルト

条件: 乗数が 10 の累乗であること (例えば 10,100,1000,...)

COBOL 4 の動作: パック 10 進数乗算 (MP) 命令を使用する

COBOL 6 の動作: 10 進数左シフトとしてモデル化する

ソースの例:

```
1 z5v2 pic s9(5)v9(2)
1 z7v2 pic s9(7)v9(2)

Compute z7v2 = z5v2 * 100
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 85% 高速

10 進数指数演算

ステートメント: COMPUTE (**)

データ型: *COMP-3*、*DISPLAY*、*NATIONAL*

オプション: デフォルト

条件: すべての事例

COBOL 4 の動作: ランタイム・ルーチン呼び出す

COBOL 6 の動作: より効率的なランタイム・ルーチン呼び出す

ソースの例:

```
1 R      PIC 9v9(8) value 0.05.  
1 NF     PIC 9(4) value 300.  
1 EXP    PIC 9(23)v9(8).  
  
COMPUTE EXP = (1.0 + R) ** NF.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 96% 高速

10 進数の位取りおよび除算

ステートメント: COMPUTE (/)、DIVIDE

データ型: COMP-3、DISPLAY、NATIONAL

オプション: デフォルト

条件: 除数値と 10 進数の位取りが相殺される場合。次の例では、除算演算は 10 進数の 2 桁左シフトを必要としますが、100 による除算は 10 進数の 2 桁右シフトとしてモデル化されているため、これらの演算は相殺されます。

COBOL 4 の動作: パック 10 進数シフト (SRP) 命令および除算 (DP) 命令を使用する

COBOL 6 の動作: 除算と 10 進数の位取りが相殺されるため、単純な MOVE 操作に相当する命令が生成される

ソースの例:

```
1 p9v0 pic s9(9) comp-3  
1 p10v2 pic s9(10)v9(2) comp-3.  
  
COMPUTE p10v2 = p9v0 / 100
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 86% 高速

TRUNC(STD) 2 進数算術演算

ステートメント: COMPUTE (+|-|*|/), ADD、SUBTRACT、MULTIPLY、DIVIDE

データ型: BINARY、COMP、COMP-4

オプション: TRUNC(STD)

条件: すべての事例

COBOL 4 の動作: 負荷の高い除算演算を使用し、桁数を PIC 指定に合わせて修正する

COBOL 6 の動作: 実際に必要な場合 (オーバーフローの場合) のみ除算を使用する

ソースの例:

```
1 b5v2a pic s9(5)v9(2) comp.  
1 b5v2b pic s9(5)v9(2) comp.  
  
COMPUTE b5v2a = b5v2a + b5v2b
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 75% 高速

大きい 2 進数算術演算

ステートメント: COMPUTE (+|-|*|/), ADD、SUBTRACT、MULTIPLY、DIVIDE

データ型: BINARY、COMP、COMP-4

オプション: TRUNC(STD)

条件: 中間結果が 9 桁を超える

COBOL 4 の動作: 区分的に実行され、パック 10 進数に変換される算術演算

COBOL 6 の動作: 64 ビット・レジスターで実行される算術演算

ソースの例:

```
1 b8v2a pic s9(8)v9(2) comp.  
1 b8v2b pic s9(9)v9(2) comp.  
  
Compute b8v2a = b8v2a + b8v2b.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 97% 高速

10 進数の否定

ステートメント: COMPUTE (-)、SUBTRACT

データ型: COMP-3、DISPLAY、NATIONAL

オプション: デフォルト

条件: すべての事例

COBOL 4 の動作: 他のゼロからの減算と同様に扱う

COBOL 6 の動作: 特殊な事例の否定演算として認識する

ソースの例:

```
1 p7v2a pic s9(7)v9(2) comp-3.  
1 p7v2b pic s9(7)v9(2) comp-3.  
  
Compute p7v2b = - p7v2a.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 74% 高速

DIVIDE GIVING REMAINDER の融合

ステートメント: DIVIDE

データ型: COMP-3、DISPLAY

オプション: OPT(1 | 2)

条件: 除算の剰余と商の両方が使用される

COBOL 4 の動作: 除算と剰余計算を分離する

COBOL 6 の動作: 単一 DP 命令を使用し、剰余と商の両方を回復する

ソースの例:

```
01 A COMP-3 PIC S9(15).  
01 B COMP-3 PIC S9(15).  
01 C COMP-3 PIC S9(15).  
01 D COMP-3 PIC S9(15).  
  
DIVIDE A BY B GIVING C REMAINDER D
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 18% 高速

INSPECT

1 バイト・オペランドでの INSPECT REPLACING ALL

ステートメント: INSPECT REPLACING ALL

データ型: PIC X

オプション: デフォルト

条件: すべての事例

COBOL 4 の動作: すべての事例で一般変換命令を使用する

COBOL 6 の動作: 単純テストおよび移動で短い事例を処理する

ソースの例:

```
1 ITEM PIC X(1)
INSPECT ITEM REPLACING ALL ' ' BY '.'.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 85% 高速

同一データ項目に対する連続 INSPECT

ステートメント: 同一データ項目に対する複数の連続 INSPECT REPLACING ALL

データ型: PIC X

オプション: OPT(1 | 2)

条件: コンパイラーが、INSPECT の規則に従って、最適化によって結果が変更されないことを証明できる場合。

COBOL 4 の動作: INSPECT 操作ごとに別個の操作を生成する

COBOL 6 の動作: 別々の INSPECT を 1 つの INSPECT 操作に合体させる

ソースの例:

```
1 ITEM PIC X(15)
INSPECT ITEM REPLACING ALL QUOTE BY SPACE.
INSPECT ITEM REPLACING ALL LOW-VALUE BY SPACE.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 60% 高速

INSPECT TALLYING ALL / INSPECT REPLACING ALL

ステートメント: INSPECT TALLYING ALL / INSPECT REPLACING ALL

データ型: PIC X

オプション: ARCH(11)

条件: BEFORE、AFTER、FIRST、または LEADING 節がないこと。REPLACING の場合、置き換えられる値の長さは > 1 でなければなりません。

COBOL 4 の動作: 通常の命令またはランタイム呼び出しを使用する

COBOL 6 の動作: ARCH(11) では、COBOL 6 は、z13 で導入されたベクトル命令を使用してコードを生成できます。これらの命令は一度に 16 バイトまで処理できます。

ソースの例:

```
01 STR PIC X(255).
01 C PIC 9(5) COMP-5 VALUE 0.
INSPECT STR TALLYING C FOR ALL ' '
```

パフォーマンス: COBOL 6 ARCH(11) は COBOL 4 よりも 99% 高速

ソースの例:

```
01 STR PIC X(255).
INSPECT STR REPLACING ALL 'AB' BY 'CD'
```

パフォーマンス: COBOL 6 ARCH(11) は COBOL 4 よりも 83% 高速

MOVE

VALUE 節およびグループの初期化

ステートメント: MOVE および VALUE IS

データ型: すべての型

オプション: OPT(1 | 2)

条件: リテラルでのデータ項目の初期化

COBOL 4 の動作: 一連の個別および順次の移動命令

COBOL 6 の動作: リテラルを合体し、生成する移動命令を少なくする

ソースの例:

```
01 WS-GROUP.  
  05 WS-COMP3      COMP-3 PIC S9(13)V9(2).  
  05 WS-COMP        COMP   PIC S9(9)V9(2).  
  05 WS-COMP5       COMP-5 PIC S9(5)V9(2).  
  05 WS-COMP1       COMP-1.  
  05 WS-ALPHANUM    PIC X(11).  
  05 WS-DISPLAY     PIC 9(13) DISPLAY.  
  05 WS-COMP2       COMP-2.  
  
Move +0 to WS-COMP5  
          WS-COMP3  
          WS-COMP  
          WS-DISPLAY  
          WS-COMP1  
          WS-COMP2  
          WS-ALPHANUM.
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 20% 高速

数字編集データ項目への移動

ステートメント: MOVE

データ型: 受け取り側は数字編集

オプション: OPT(1 | 2)

条件: なし

COBOL 4 の動作: すべての事例で ED 命令または EDMK 命令を使用する

COBOL 6 の動作: 数字編集を処理する ED 命令と EDMK 命令は極度に低速になる。COBOL 6 では、これらの特殊な命令の使用が、一連の他の命令に変換されます。これは、COBOL 6 の新しい最適化手法です。

ソースの例:

```
01 PRINCIPAL PIC 9(8)V9999 VALUE 1234.1234.  
01 AMT-PRINCIPAL PIC $,,$$,,$$9.99.  
  
Move PRINCIPAL to AMT-PRINCIPAL.
```

パフォーマンス: COBOL 6 は 73% 高速化

SEARCH

SEARCH ALL

ステートメント: SEARCH ALL

オプション: デフォルト

条件: すべての事例

COBOL 4 の動作: ランタイム・ルーチン呼び出す

COBOL 6 の動作: より効率的なランタイム・ルーチン呼び出す

ソースの例:

```
SEARCH ALL table
  AT END
    statements
  WHEN conditions
    statements
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 50% 高速

テーブル

索引付きテーブル

ステートメント: 索引付きテーブル内のデータ項目へのアクセス

オプション: OPT(2)

条件: すべての事例

COBOL 6 の動作: テーブルの先頭までのオフセットを毎回再ロードするのではなく、グローバルに使用可能なレジスターでこれをキャッシングすることにより、効率的なシーケンスを使用して索引付きテーブル・エレメントにアクセスします。

ソースの例:

```
1  TAB.
   5 TABENTS OCCURS 40 TIMES INDEXED BY TABIDX.
     10 TABENT1    PIC X(4) VALUE SPACES.
     10 TABENT2    PIC X(4) VALUE SPACES.

   IF TABENT1 (TABIDX) NOT = TABENT2 (TABIDX)
     statements
   END-IF
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 17% 高速

条件式

小さいデータ項目と定数の比較

ステートメント: 条件式

データ型: DISPLAY、COMP-3

オプション: OPT(1 | 2)

条件: データ項目がゾーンの場合は 8 桁以下、パックの場合は 15 桁以下。

COBOL 4 の動作: メモリー内の命令を使用し、データ項目の符号コードを既知の値に変更後、定数と比較する。

COBOL 6 の動作: データ項目の値をレジスターにロード後、符号コードを変更し、レジスター内で比較を実行する。これは、COBOL 6 での新しい最適化です。

ソースの例:

```
01 A PIC 9(4).

If A = 0 THEN
...
```

パフォーマンス: COBOL 6 は COBOL 4 よりも 85% 高速です。

第3章 COBOL 6 を最大限に活用するためのコンパイラー・オプションのチューニング方法

Enterprise COBOL 6 には、パフォーマンスに影響する可能性のある、大幅に変更された多数の新しいコンパイラー・オプションが用意されています。このセクションでは、これらのオプションに焦点を当て、アプリケーションで可能な最良のパフォーマンスを達成するための最適な設定に関する推奨事項を示します。

最良のパフォーマンスを得るための推奨されるコンパイラー・オプションは、OPT(2)、ARCH(x)、TUNE(y)です。

これらのオプションは、以下によってパフォーマンスを向上させます。

- 最大レベルの最適化 - OPT(2)
- アーキテクチャーの高度な活用:
 - ARCH(x) (x は 10 | 11 | 12 | 13 | 14)。アプリケーションが実行される最も古いマシン (災害復旧 (DR) システムを含む) のアーキテクチャーに一致するように値を設定します。
 - TUNE(y) (y は 10 | 11 | 12 | 13 | 14)。アプリケーションを最も頻繁に実行する予定のマシンのアーキテクチャーと一致するように値を設定してください。TUNE レベルは常に ARCH レベル以上でなければなりません。

本書の推奨事項および「Enterprise COBOL for z/OS プログラミング・ガイド」を参照してください。

一部のユーザーに該当する最大パフォーマンスのための追加設定は、STGOPT、AFP(NOVOLATILE)、HGPR(NOPRESERVE) です。

これらのオプションは、以下によってパフォーマンスを向上させます。

- 参照されないデータ項目の除去 - STGOPT
- 浮動小数点レジスターと高位ワード・レジスターの保存/復元の省略 - AFP および HGPR

注：これらの追加オプションを使用するには、後述のように、また「Enterprise COBOL for z/OS プログラミング・ガイド」の『コンパイラー・オプション』で説明されているように、重要な前提条件がいくつかあります。使用する前に、これらのオプション設定についての説明を読み、完全に理解しておいてください。

手短かに言えば、以下のような制約事項があります。

- STGOPT - プログラムが以下のいずれかのデータ項目に依存している場合は、STGOPT を使用できません。
 - 未参照 LOCAL-STORAGE および非外部 WORKING-STORAGE レベル 77 およびレベル 01 基本データ項目
 - 非外部レベル 01 グループ項目 (どの従属項目も参照されない場合)
 - 未参照特殊レジスター
- 高位ワード・レジスターの保存/復元の省略 - HPGR
- HGPR(NOPRESERVE) - 呼び出し元が Enterprise COBOL、Enterprise PL/I、または z/OS XL C/C++ のいずれかのコンパイラー生成コードである場合にのみ設定する必要があります。

次に、これらの、およびその他のパフォーマンス関連コンパイラー・オプションを設定する際の考慮事項について説明します。

関連参照

パフォーマンスに関連するコンパイラー・オプション
(Enterprise COBOL for z/OS プログラミング・ガイド)

Default

AFP(NOVOLATILE)

推奨

AFP(NOVOLATILE)

理由

AFP(VOLATILE) を指定した場合、呼び出し中に値をレジスター FP8 から FP15 に保管することはできません。代わりに、値はメモリーに保存し、後で復元する必要があります。パフォーマンス上の影響は、呼び出し頻度の高い、小さいプログラムで最も大きくなります。

OPT(2) で AFP(NOVOLATILE) を使用すると、AFP(VOLATILE) に比べて、プログラム呼び出しのオーバーヘッドが 7% 削減されます。これは、このオプションのパフォーマンス・コストを強調するために、これ以外は空の COBOL プログラムで測定されたものであり、より大きい呼び出し先プログラムでの全体的な低下量よりは少ない可能性があることに注意してください。

考慮事項

AFP(NOVOLATILE) を指定するには、CICS® Transaction Server 4.1 以降が必要です。

注：LP(64) コンパイラー・オプションが指定されている場合、AFP は影響を及ぼしません。つまり、ユーザー・アプリケーションは COBOL コンパイラーによる浮動小数点レジスターの使用方法を制御できません。コンパイラーは、AFP(NOVOLATILE) が指定されている場合と同じ動作をする可能性があります。LP(64) の場合、すべてのレジスター・クラスでのレジスター使用方法 (何が保持され、何が揮発するか) は XPLINK 指定に従うことに注意してください。この指定を超える保証はサポートされていません。

関連参照

AFP (Enterprise COBOL for z/OS プログラミング・ガイド)

ARCH

デフォルト

ARCH(10)

推奨

ARCH(x)。x は、災害復旧システムなど、アプリケーションの実行時に必要となる最低限のハードウェア・レベルです。

理由

新しいハードウェア・モデルは、より迅速にタスクを実行できる新しいハードウェア命令を提供します。ARCH 設定は、対応するハードウェア上に存在する命令を選択するようにコンパイラーに指示し、プログラムがターゲット・ハードウェア上で実行できるようにします。

注：ご使用のアプリケーションは、ARCH オプションで指定したものよりも低いアーキテクチャー・レベルのプロセッサで実行した場合、異常終了するおそれがあります。

考慮事項

ハードウェア・レベルとのマッチング以外は、特にありません。

ARCH のみを変更し、他のオプションを最適推奨値に維持することにより、IBM 社内での一連のパフォーマンス・ベンチマークを通じて、以下のパフォーマンス向上が測定されました。

表 1. ARCH レベル別の平均向上率	
ARCH レベル	平均向上率 (%)
ARCH(11) vs ARCH(10)	2.7%
ARCH(12) vs ARCH(11)	18.6%
ARCH(13) vs ARCH(12)	1.5%
ARCH(14) と ARCH(13) の比較	3.6%

ARCH(10) から直接 ARCH(14) に移行する場合、これらの同じベンチマーク・セットのパフォーマンス向上率の平均は 25% です。

上記の数字では、ARCH コンパイラー・オプションのみが変更されたことに注意してください。基礎となるハードウェアはすべての事例で IBM z14 マシンでした。このことは、パフォーマンス向上が主に、テスト対象の COBOL アプリケーションの最適化におけるコンパイラー向上に基づくものであることを意味します。

この一連のベンチマークには、計算集中型アプリケーションと入出力集中型アプリケーションが混在しています。最も大きな改善が見られるのは、計算処理が主体のアプリケーションです。例えば、計算処理が主体のベンチマークの 1 つは、ARCH(10) から ARCH(14) に移行することで、実行時間が 84% 短縮されました。他のベンチマークでは、時間の大部分が入出力操作の実行に費やされ、コンパイラー生成コードにおける時間は比較的小さくなります。これらのベンチマークのパフォーマンスは、ARCH オプションによって大きく影響を受けることはありません。

参考のため、ARCH 設定とハードウェア・モデルのマッピングを以下に示します。

表 2. ARCH の設定とハードウェア・モデル	
ARCH	ハードウェア・モデル
ARCH(10)	2827-xxx モデル (IBM zEnterprise EC12) 2828-xxx モデル (IBM zEnterprise BC12)
ARCH(11)	2964-xxx モデル (IBM z13®) 2965-xxx モデル (IBM z13s)
ARCH(12)	3906-xxx モデル (IBM z14) 3907-xxx モデル (IBM z14 ZR1)
ARCH(13)	8561-xxx モデル (IBM z15) 8562-xxx モデル (IBM z15 T02)
ARCH(14)	3931-xxx モデル (IBM z16)

関連参照

ARCH

(Enterprise COBOL for z/OS プログラミング・ガイド)

ARITH

デフォルト

ARITH(COMPAT)

推奨

ARITH(EXTEND) は、このオプションを使用してより大きい最大桁数 (18 の代わりに 31) を有効にする必要がある場合のみ使用してください。それ以外の場合は、状況に応じてより良いパフォーマンスが得られる可能性のある ARITH(COMPAT) を使用してください。

理由

ARITH(EXTEND) では、より大きい変数の宣言が可能になるだけでなく、中間結果として維持できる最大桁数も増加します。これらのより大きい中間結果は、より低速な異なるコードの生成を必要とする場合があります。インライン計算は、より負荷の高いランタイム・ライブラリー・ルーチンへの置き換えが必要になることがあります。

例えば、次のような comp-1 浮動小数点指数演算を考えます。

```
COMPUTE C = A ** B
```

この場合、ARITH(COMPAT) を使用すると、ARITH(EXTEND) に比べて 67% 速くなります。

関連参照

ARITH (Enterprise COBOL for z/OS プログラミング・ガイド)

AWO

デフォルト
NOAWO

推奨

AWO。ただし、書き込まれたレコードをディスク上で可能な限り速やかに更新する必要がない場合に限りです。

理由

複数のレコードを1つのブロックにまとめて書き込むことで、EXCPを大幅に削減できます。その結果、ファイル出力操作が高速化し、CPU使用量が減少します。

AWO コンパイラー・オプションは、プログラムで APPLY WRITE-ONLY 節が指定されていない場合でも、すべての物理順次可変長ブロック化ファイルに対して APPLY WRITE-ONLY 節を有効にします。APPLY WRITE-ONLY が有効になっていると、次のレコード用に十分なスペースがファイル・バッファにない場合に、バッファの内容が出力デバイスに書き込まれます。APPLY WRITE-ONLY がないと、最大サイズのレコード用に十分なスペースがファイル・バッファにない場合に、バッファの内容が出力デバイスに書き込まれます。アプリケーションの書き込むレコード・サイズが大きく変動する場合は、APPLY WRITE-ONLY を使用すると一般的に入出力を処理するデータ管理サービスの呼び出しが減るため、パフォーマンス上の節約になります。

注:

- AWO コンパイラー・オプションを使用する代わりに、APPLY WRITE-ONLY 節をプログラム内の物理順次可変長ブロック化ファイルに対して使用できます。ただし、最大限のパフォーマンス上の利点を得るためには、プログラム内のすべての物理順次可変長ブロック化ファイルに対して APPLY WRITE-ONLY 節を使用する必要があります。この方法を使用した場合、パフォーマンス上の利点は、AWO コンパイラー・オプションを使用した場合と同じになります。
- AWO コンパイラー・オプションは、物理順次可変長ブロック化ファイルを含まないプログラムには効果がありません。

パフォーマンスの例として、可変長ブロック化ファイルを使用する、あるテスト・プログラムで、AWO は NOAWO よりも 90% 高速でした。この処理の高速化は、書き込みを処理する EXCP の使用が 98% 減少した結果です。

関連参照

AWO (Enterprise COBOL for z/OS プログラミング・ガイド)

BLOCKO

デフォルト
NOBLOCKO

推奨

BLOCKO

理由

ブロック入出力によって物理的入出力転送回数が削減され、その結果 EXCP が減少します。BLOCKO コンパイラー・オプションは、(BLOCK CONTAINS 0 節をファイルに指定した場合のように) QSAM ファイルのデフォルトを非ブロック化からブロック化に変更するため、出力ファイルのシステム決定ブロック化の利点を得られます。BLOCKO は、以下のすべての基準を満たすプログラム内の各ファイルに対して、暗黙の BLOCK CONTAINS 0 節を活動化します。

- FILE-CONTROL 段落で ORGANIZATION SEQUENTIAL が指定されているか、ORGANIZATION 節が省略されている。
- FD 項目で RECORDING MODE U が指定されていない。
- FD 項目で BLOCK CONTAINS 節が指定されていない。

パフォーマンスの例として、上記の基準を満たす、BLOCK0 を使用したあるテスト・プログラムは、NOBLOCK0 を使用したプログラムより 90% 高速であり、EXCP の使用が 98% 減少しました。

関連参照

BLOCK0 (Enterprise COBOL for z/OS プログラミング・ガイド)

DATA(24) および DATA(31)

デフォルト

DATA(31)

推奨

DATA(31) - プログラムが AMODE 24 サブプログラムを呼び出してパラメーターを渡す必要がない場合。

理由

RENT プログラムでの DATA(31) の使用は、16 MB 境界より下の仮想記憶域の制約の問題をある程度緩和するために役立ちます。RENT プログラムで DATA(31) を使用すると、大半の QSAM ファイル・バッファを 16 MB 境界より上に割り振ることができます。ランタイム・オプション HEAP(,ANYWHERE) を指定して DATA(31) を使用すると、すべての非 EXTERNAL の WORKING-STORAGE および非 EXTERNAL の FD レコード域を 16 MB 境界より上に割り振ることができます。

DATA(24) では、WORKING-STORAGE および FD レコード域は 16 MB 境界より下に割り振られます。

注:

- NORENT プログラムの場合、RMODE オプションによって非 EXTERNAL のデータの割り振り先が決まります。
- QSAM ファイル・バッファについて詳しくは、[『QSAM バッファ』](#)を参照してください。
- EXTERNAL データの割り振り先については、[『ALL31』](#)を参照してください。
- LOCAL-STORAGE データは DATA オプションの影響を受けません。STACK ランタイム・オプションおよびプログラムの AMODE によって、LOCAL-STORAGE の割り振り先が決まります。

これによってアプリケーションのパフォーマンスが影響を受けることはない想定されますが、プログラムのデータの配置場所は影響を受けることに注意してください。

関連参照

DATA (Enterprise COBOL for z/OS プログラミング・ガイド)

DYNAM

デフォルト

NODYNAM

考慮事項

DYNAM コンパイラー・オプションは、CALL リテラル・ステートメントで呼び出されたすべてのサブプログラムを、実行時に動的にロードすることを指定します。これにより、サブプログラムを変更した場合にアプリケーションの再リンクが不要になるため、共通サブプログラムを複数の異なるアプリケーション間で共有でき、これらのサブプログラムの保守が容易になります。また、DYNAM を使用すると、サブプログラムが不要になった場合に、サブプログラムで使用されていた仮想ストレージを CANCEL ステートメントを使用して解放できるため、仮想ストレージ使用の制御が可能になります。ただし、DYNAM オプションを使用すると、呼び出しはライブラリー・ルーチンを経由しなければならないため、パフォーマンスが犠牲になります。一方、NODYNAM オプションを使用すると、呼び出しは直接サブプログラムに対して行われます。したがって、パス長さは、DYNAM の方が NODYNAM よりも長くなります。

CALL 処理が主体のプログラムで CALL リテラルを使用したパフォーマンスの例として (CALL オーバーヘッドのみを測定)、DYNAM を使用した場合は、CALL に関連するオーバーヘッドのために、NODYNAM よりも約 74% 低速でした。この結果は、同一プログラムの呼び出し回数による影響を受けています。

呼び出し回数が多いと、サブプログラムのロードによるオーバーヘッド・コストがより多く償却される傾向があります。

CALL リテラルと CALL ID の使用に関するその他の考慮事項については、[42 ページの『CALL の使用』](#)を参照してください。

注: このテストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、さほど機能低下はありません。

関連参照

DYNAM (Enterprise COBOL for z/OS プログラミング・ガイド)

FASTSRT

デフォルト

NOFASTSRT

推奨

FASTSRT - COBOL ファイル・エラー処理セマンティクスがソート処理中に不要な場合。

理由

適格ソートの場合、FASTSRT コンパイラー・オプションは、SORT 製品がすべての入出力を処理すること、また COBOL がこれを処理する必要がないことを指定します。これにより、各レコードの読み取り後、または COBOL が SORT に返す各レコードの処理後に、制御を COBOL に返すオーバーヘッドがすべて排除されます。ソート作業ファイルに直接アクセス装置を使用する場合は、FASTSRT の使用をお勧めします。このオプションを使用すると、コンパイラーは、このオプションに適格なソートを決定して、適切なコードを生成します。このオプションにソートが適格でない場合、コンパイラーは、NOFASTSRT オプションが有効になっている場合と同じコードを生成します。FASTSRT オプションを使用するための要件のリストが COBOL プログラミング・ガイドにあります。

パフォーマンスの例として、100,000 件のレコードを処理したあるテスト・プログラムは、FASTSRT 使用時には NOFASTSRT 使用時に比べて 45% 高速であり、EXCP の使用が 4,000 減少しました。

関連参照

FASTSRT (Enterprise COBOL for z/OS プログラミング・ガイド)

HGPR

デフォルト

HGPR(PRESERVE)

推奨

HGPR(NOPRESERVE)

理由

64 ビット GPR の高位半分を入り口で保存し、出口で復元するためのコード生成が不要であるため、より良いパフォーマンスが得られます。推奨される HGPR 設定の使用は特に、頻繁に実行される比較的小さな COBOL プログラムのパフォーマンスを向上させるために重要です。

HGPR(PRESERVE) を指定した場合、コンパイラーは、呼び出し中に保存される汎用レジスター (GPR) の上位半分に依存することはできません。代わりに、これらはメモリーに保存し、後で復元する必要があります。パフォーマンス上の影響は、呼び出し頻度の高い、小さいプログラムで最も小さくなります。

HGPR (NOPRESERVE) を使用すると、HGPR (PRESERVE) と比べて、OPT(2) でのプログラム呼び出しのオーバーヘッドが 15% 削減されます。これは、このオプションのパフォーマンス・コストを強調するために、これ以外は空の COBOL プログラムで測定されたものであり、より大きい呼び出し先プログラムでの全体的な低下量よりは少ない可能性があることに注意してください。

考慮事項

PRESERVE サブオプションは、プログラムの呼び出し元が Enterprise COBOL、Enterprise PL/I、または z/OS XL C/C++ のどのコンパイラ生成コードでもない場合にのみ必要です。

関連参照

HGPR (Enterprise COBOL for z/OS プログラミング・ガイド)

INLINE

Default

INLINE

推奨

INLINE

理由

INLINE オプションを指定すると、コンパイラは段落またはセクションの PERFORM を、その段落またはセクションのコードのコピーに置き換えることを選択する可能性があります。コンパイラは、PERFORM の位置にそのコードを挿入することで、プロシージャに対する分岐ロジックのオーバーヘッドをなくします。また、コンパイラはインライン化によって、インライン化されたコードに対してさらに最適化を実行することができます。例えば、インライン化されたコード内部で使用するデータ項目は、その PERFORM で既知の定数値を持つ可能性があり、コンパイラは式を単純化できます。

NOINLINE を指定すると、コンパイラはインライン化された重複コードを作成しません。

例えば、ループで段落を繰り返し実行する単純なプログラムで INLINE を使用した場合、NOINLINE を使用した場合よりも 45% 速く実行されます。

考慮事項

>>INLINE OFF ディレクティブおよび >>INLINE ON ディレクティブは、インライン化に対してよりきめ細かい制御を行うことができます。これは、プログラム・サイズを縮小し、PERFORM の実行頻度が低い (エラー処理コードなど) 場合に命令キャッシュのパフォーマンスを向上させるために役立ちます。

関連参照

INLINE (Enterprise COBOL for z/OS プログラミング・ガイド)

INVDATA

Default

NOINVDATA

推奨

NOINVDATA

ほとんどのユーザーは、USAGE DISPLAY データ項目および USAGE PACKED-DECIMAL データ項目に有効なデータを持っているため、NOINVDATA を使用してアプリケーションのパフォーマンスを向上させます。プログラムが NUMCHECK コンパイラ・オプションを使用して実行時に無効なデータを処理していることが判明した場合でも、無効なデータを処理しないようにプログラムを変更して、NOINVDATA を使用する必要があります。

注: INVDATA オプションの目的は、無効な数値データの場合に、COBOL 4 以前のバージョンでコンパイルされたプログラムの動作と可能な限り互換性のある動作を提供することです。矛盾が見つかった場合、このオプションは、COBOL 4 以前のバージョンの動作とまったくよく一致した動作になるように更新されます。

INVDATA オプションが有効な場合、ゾーン 10 進数データ項目またはパック 10 進数データ項目に無効な数字または無効な符号コードが含まれている場合、またはゾーン 10 進数データ項目に無効なゾーン・ビットが含まれている場合に、コンパイラは、COBOL 4 以前のバージョンとは異なる結果を生成する可能性がある既知の最適化を実行しません。

次の表は、以前のバージョンの COBOL で使用されていた NUMPROC オプションのデフォルト値、および無効なデータがあるかどうかに応じて、以前のバージョンの COBOL から COBOL 6.2 以降のバージョンに移行する際に INVDATA オプションと NUMPROC オプションを設定する方法についてのクイック・リファレンスです。

表 3. 以前の COBOL バージョンからマイグレーションする場合の INVDATA オプションと NUMPROC オプションの設定			
COBOL のバージョン	無効なデータが存在しますか?	COBOL 6.1 以前のバージョンで使用される NUMPROC/ZONEDATA	COBOL 6.2 以降のバージョンでの INVDATA および NUMPROC の設定
COBOL 5 より前	No	NUMPROC (MIG)	NOINVDATA, NUMPROC (NO PFD)
COBOL 5 より前	No	NUMPROC (NOPFD)	NOINVDATA, NUMPROC (NOPFD)
COBOL 5 より前	No	NUMPROC (PFD)	NOINVDATA, NUMPROC (PFD)
COBOL 5 より前	Yes	NUMPROC (MIG)	INVDATA (FORCENUMCMP, NOCLEANSIGN), NUMPROC (NOPFD)
COBOL 5 より前	Yes	NUMPROC (NOPFD)	INVDATA (NOFORCENUMCMP, CLEAN SIGN)、NUMPROC (NOPFD) または INVDATA、NUMPROC (NOPFD)
COBOL 5 より前	Yes	NUMPROC (PFD)	INVDATA (NOFORCENUMCMP, CLEAN SIGN)、NUMPROC (PFD) または INVDATA、NUMPROC (PFD)
COBOL 5 以降	No	ZONEDATA (PFD)	NOINVDATA
COBOL 5 以降	Yes	ZONEDATA (NOPFD)	INVDATA (NOFORCENUMCMP, CLEAN SIGN) または単に INVDATA
COBOL 5 以降	Yes	ZONEDATA (MIG)	INVDATA (FORCENUMCMP, CLEAN SIGN) ¹
<p>1. INVDATA (FORCENUMCMP, NOCLEANSIGN) は、無効なデータが存在する場合の INVDATA (FORCENUMCMP, CLEAN SIGN) よりも、COBOL 5 より前の NUMPROC (MIG) 動作をより詳しく表したものです。無効なデータが存在する場合の COBOL 5 以降のバージョンでの ZONEDATA (MIG) の動作に満足できない場合は、INVDATA (FORCENUMCMP, NOCLEANSIGN) を使用して、無効なデータが存在する場合の COBOL 5 より前の NUMPROC (MIG) の動作をより厳密に模倣することを検討してください。</p>			

INVDATA オプションについて、および符号コード、数字、ゾーン・ビットが無効な場合のコンパイラーの動作について詳しくは、Enterprise COBOL for z/OS プログラミング・ガイドの INVDATA を参照してください。

理由

- NOINVDATA オプションが有効な場合、コンパイラーは、USAGE DISPLAY および PACKED-DECIMAL データ項目のデータが有効であると想定し、数値比較を行うために可能な限り最も効率的なコードを

生成します。例えば、コンパイラーは数値変換を避けるために、ストリング比較を生成する場合があります。

- INVDATA(FORCENUMCMP) オプションが有効な場合、コンパイラーは、ゾーン 10 進数データ項目内の各桁のゾーン・ビットを無視する数値比較を行うための追加命令を生成する必要があります。例えば、ゾーン 10 進数値は比較前に、PACK 命令によってパック 10 進数に変換される場合があります。
- INVDATA(NOFORCENUMCMP) オプションが有効な場合、COBOL 6 コンパイラーは、COBOL 4 コンパイラーと同じ方法で、無効なゾーン・ビット、無効な符号コード、および無効な数字を処理するシーケンスを(たとえそのシーケンスが、考えられる別のシーケンスよりも非効率的であったとしても)生成する必要があります。次のケースが考慮されます:
 - COBOL 4 (またはそれ以前のバージョン) でゾーン・ビットの対象となっていた場合、コンパイラーは英数字比較を生成し、英数字比較でもゾーン 10 進数データ項目内の各桁のゾーン・ビットが数値比較の対象となります。ゾーン 10 進数値はゾーン 10 進数のままです。
 - COBOL 4 以前のバージョンでゾーン 10 進数データ項目内の各桁のゾーン・ビットが無視されていた場合。ゾーン 10 進数値は比較前に、PACK 命令によってパック 10 進数に変換されます。

ソースの例

```
01 A PIC S9(5)V9(2).  
01 B PIC S9(7)V9(2).  
COMPUTE B = A * 100
```

この例の乗算は、NOINVDATA を使用すると、INVDATA を使用するよりも 63% 速くなります。NOINVDATA では、コンパイラーは、乗算の代わりにシフト命令を使用できます。INVDATA では、コンパイラーは、より負荷の高い乗算を実行しなければなりません。

関連参照

INVDATA (Enterprise COBOL for z/OS プログラミング・ガイド)

MAXPCF

デフォルト

MAXPCF(100000)

推奨

MAXPCF(0)

理由

MAXPCF を指定すると、過度のコンパイル時間またはストレージ所要量を必要とする可能性のある、大きく複雑なプログラムの最適化量を自動的に削減することができます。

MAXPCF オプションは、最適化を犠牲にして、大きなプログラムを正常にコンパイルできるようにすることを目的としています。ただし、可能であれば、大きなアプリケーションを小さな別々のプログラムに再構成することをお勧めします。

いくつかの新しい「グローバル」最適化が、COBOL 5 コンパイラー・リリースに追加されています。これらの最適化は、1つのステートメント内で、またはセクション内の一連のステートメント内で、「ローカル」ではなく、プログラム全体で協同部分を検出してパフォーマンスを改善しようとするため、「グローバル」と呼ばれます。これらのグローバル最適化は、プログラム全体のステートメントとデータ項目を分析する必要があるため、かなりのストレージと時間を要する場合があります。

この理由から、また一般的にソフトウェア保守アクティビティーに有益となるように、大きなプログラムを、静的呼び出しで相互にリンクされた別個の小さなプログラムに分割することを強くお勧めします(動的呼び出しの使用に比べてオーバーヘッドを最小化するため)。このような小さなプログラムは、MAXPCF オプションによる最適化のダウングレードを必要としなくなる可能性が高くなります。また、コンパイル時間が短縮され、ストレージ所要量が少なくなる可能性もあります。さらに、コンパイルおよびリンクされた最終的なアプリケーションは、COBOL 6 コンパイラーで使用可能な一連の最適化の対象となります。

考慮事項

MAXPCF(0) を指定すると、コンパイル時間が増加するおそれがあります。

関連参照

MAXPCF (Enterprise COBOL for z/OS プログラミング・ガイド)

NUMCHECK

デフォルト

NONUMCHECK

推奨

最良のパフォーマンスを得るには、NONUMCHECK をお勧めします。NUMCHECK を指定すると、数値データ項目が送り出し側として使用される場合は必ず、IS NUMERIC クラス・テストが生成されます。

理由

NUMCHECK によって挿入される追加検査により、ゾーン 10 進数データ項目を送り出し側データ項目として使用するプログラムのパフォーマンスが大幅に低下するおそれがあります。プログラムのパフォーマンスが重要な部分を含め、すべてのケースで検査を有効にする NUMCHECK を使用する代わりに、データが読み取られるプログラム内の場所に IS NUMERIC テストを手動で挿入の方がより高速です。

例えば、ゾーン 10 進数で以下の移動を行うとします。

```
01 X1 PIC 9(5).  
01 X2 PIC 9(5).  
MOVE X1 TO X2.
```

NONUMCHECK を使用した場合は、NUMCHECK(MSG) または NUMCHECK(ABD) に比べて 51% 高速です。一連のベンチマーク・プログラムで NONUMCHECK を使用すると、NUMCHECK(ZON,PAC,BIN,MSG) よりも 17% 高速でした。

考慮事項

NUMCHECK の使用は、ランタイム・パフォーマンスへの影響に加え、コンパイル時間が大幅に増加するおそれもあります。

注: NUMCHECK(ZON) は、以前には ZONECHECK と呼ばれていました。

関連参照

NUMCHECK (Enterprise COBOL for z/OS プログラミング・ガイド)

NUMPROC

デフォルト

NUMPROC(NOPFD)

推奨

数値データが、「Enterprise COBOL for z/OS プログラミング・ガイド」の NUMPROC で詳述されている IBM システム標準に完全に準拠している場合、アプリケーションのパフォーマンスを向上させるには、NUMPROC(PFD) を使用します。

理由

NUMPROC(PFD) を使用すると、コンパイラーは入力された符号構成を修正するためのコードを生成する必要がなくなるため、パフォーマンスが向上します。これは特に、アプリケーションに符号なし内部 10 進数データとゾーン 10 進数データが含まれている場合に重要です。この型のデータは、特定の数値演算ステートメント、移動ステートメント、または比較ステートメントの後で修正を必要とするだけでなく、数値演算ステートメントまたは比較ステートメントで使用する前にも修正を必要とするためです。

NUMPROC(PFD) を使用すると、多様な数値演算が含まれるベンチマークは NUMPROC(NOPFD) に比べて 11% 向上します。

関連参照

NUMPROC (Enterprise COBOL for z/OS プログラミング・ガイド)

OPTIMIZE

Default

OPT(0)

推奨

OPT(2)

理由

最大レベルの最適化によって一般的に、最高速で実行されるコードがコンパイラーによって生成されます。

例えば、一連のベンチマーク・プログラムでは、OPT(1)を使用したコンパイルはOPT(0)よりも42%速く、OPT(2)を使用したコンパイルはOPT(1)よりも1.7%速いことが示されています。ベンチマーク・スイートのうち計算処理が主体のプログラムの1つでは、OPT(1)を使用するとOPT(0)よりも84%速く、OPT(2)を使用するとOPT(1)よりも49%速く、OPT(2)を使用するとOPT(0)よりも92%速くなりました。

考慮事項

OPT(2)のコンパイルは一般的に、OPT(1)またはOPT(0)を使用した場合に比べて、より多くのメモリーを使用し、完了までに時間がかかります。

一連のベンチマークから集められたコンパイル時データによると、平均してOPT(1)ではOPT(0)の1.5倍、OPT(2)ではOPT(0)の1.8倍の時間がかかることがわかっています。非常に大規模なテスト・ケースの場合は、コンパイル時トレードオフが平均より悪化する可能性があります。

さらに、コンパイラー最適化および不要コード除去がこの設定ではより高度になっているため、デバッグ容易性が低下することもあります。

OPTIMIZE オプションで可能な設定は、COBOL 4 と COBOL 5 の間で変更されています。COBOL 6 では、新しい COBOL 5 の設定が引き続き使用されます。これらの設定の意味は、COBOL 5 と COBOL 6 の間で変わっていません。

参照されないレベル 01 とレベル 77 の項目は、COBOL 4 で OPT(FULL) を指定した場合と同様、この最高 OPT 設定で削除されなくなったことに注意してください。これは、以前に OPT(FULL) を使用できなかったプログラムが OPT(2) を指定できることを意味します。詳細については、[24 ページの『STGOPT』](#)を参照してください。

COBOL 4 と COBOL 6 はいずれも 3 つのレベルの OPT 指定を提供していますが、名前と (さらに重要なことに) 基礎となる有効な最適化が変更されています。

例えば、COBOL 4 と COBOL 6 の重要な相違は、COBOL 4 での最高設定である OPT(FULL) は、一連の OPT(STD) 最適化と、参照されないデータ項目および対応する (VALUE 節を初期化する) コードの除去を組み合わせたものであったことです。

一方、COBOL 6 での最高設定は OPT(2) であり、これには一連の OPT(1) 最適化と、パフォーマンスを向上させるための追加の最適化 (値および符号状態情報のグローバルな伝搬、索引付きテーブルにアクセスするためのレジスター割り振りの改善など) が含まれています。

「Enterprise COBOL for z/OS プログラミング・ガイド」の OPTIMIZE にある『削除されたオプションから新しいオプションへのマッピング』の表で詳述されているように、COBOL 4 OPT 設定は現在許容されていますが、どの COBOL 4 設定も OPT(2) にマップされていません。例えば、COBOL 6 で指定される OPT(FULL) は、OPT(1) と STGOPT にマップされています。

関連参照

OPTIMIZE (Enterprise COBOL for z/OS プログラミング・ガイド)

PARMCHK

デフォルト

NOPARMHECK

推奨

最良のパフォーマンスを得るには、NOPARMCHECK をお勧めします。PARMCHECK を指定すると、WORKING-STORAGE の最後の項目の後にバッファを生成するようにコンパイラに指示します。そして、そのバッファを使用して、呼び出されたサブプログラムが WORKING-STORAGE の領域外のデータを破壊したかどうかを検査されます。

理由

呼び出し側プログラムが PARMCHECK でコンパイルされている場合、コンパイラは、WORKING-STORAGE セクションの最後のデータ項目の後にバッファを生成します。実行時に、各呼び出しの前にバッファが ALL x'AA' に設定されます。各呼び出しの後、バッファが検査されて、バッファが変更されたかどうかを確認されます。PARMCHECK オプションは、呼び出し元の WORKING-STORAGE SECTION の終わりを越えて呼び出されたプログラムがいつ書き込まれたかを検出することで、COBOL 4 以前のコンパイラから COBOL 6 以降のコンパイラへの移行に役立ちます。

プログラム内のすべての CALL ステートメントの後でバッファを検査すると、オーバーヘッドが加わります。例えば、空のプログラムを呼び出すプログラムで NOPARMCHECK を使用した場合、PARMCHECK を使用するよりも 22.2% 速くなります。

関連参照

PARMCHECK (Enterprise COBOL for z/OS プログラミング・ガイド)

SSRANGE

Default

NOSSRANGE

推奨

最良のパフォーマンスを得るには、NOSSRANGE をお勧めします。SSRANGE を指定すると、範囲外ストレージ参照を検出するための追加コードが生成されます。

理由

SSRANGE で有効になる追加検査は、プログラムのパフォーマンス依存領域において、索引、添え字、参照変更式 (UTF-8 以外のデータ項目および関数値の場合) を使用するプログラムのパフォーマンスを大幅に低下させる可能性があります。プログラムの一部の場所でのみ追加の範囲検査が必要な場合は、すべての事例で検査を有効にする SSRANGE を使用する代わりに、独自の検査をコーディングする方が高速化する可能性があります。

COBOL 6 では、コンパイル範囲検査を無効にするためのランタイム・オプションがなくなっていることに注意してください。そのため、SSRANGE を指定すると、実行時に常に範囲検査コードが使用されることになります。SSRANGE を指定した場合、テーブルに対する添え字付き参照を中程度に使用するベンチマークは、速度が 5% 低下します。

SSRANGE(ZLEN) と SSRANGE(NOZLEN) では、パフォーマンスの差はありません。

考慮事項

SSRANGE の使用は、ランタイム・パフォーマンスへの影響に加え、コンパイル時間が大幅に増加するおそれもあります。

関連参照

SSRANGE (Enterprise COBOL for z/OS プログラミング・ガイド)

STGOPT

デフォルト

NOSTGOPT

推奨

STGOPT

理由

COBOL 5 で導入されたこの新しいオプションは、OPT とは関連がありません。COBOL 4 では、OPT(STD) から OPT(FULL) に変更した場合に、参照されないデータ項目と、VALUE 節を初期設定するための対応するコードを除去する STGOPT の動作が暗黙指定されていました。COBOL 5 以降、この動作は独立して指定されるようになりました。一連のベンチマーク・プログラムでは、STGOPT を使用すると、オブジェクト・ファイルのサイズが、OPT(2) を使用した場合と比較して平均 4.3%、最大 32.6% 削減されました。

考慮事項

COBOL 6 で STGOPT の使用を決定する際には、COBOL 4 での OPT(FULL) の指定に適用されたものと同じ事項を考慮する必要があります。つまり、以下のいずれかのデータ項目に依存している場合は、OPT(FULL) も STGOPT も使用できません。

- 未参照 LOCAL-STORAGE および非外部 WORKING-STORAGE レベル 77 およびレベル 01 基本データ項目
- 非外部レベル 01 グループ項目 (どの従属項目も参照されない場合)
- 未参照特殊レジスター

注: STGOPT オプションは、VOLATILE 節を持つデータ項目では無視されます。

関連参照

STGOPT (Enterprise COBOL for z/OS プログラミング・ガイド)

TEST

パフォーマンスとデバッグ能力のトレードオフの説明など、TEST のオプションとサブオプションについて詳しくは、「Enterprise COBOL for z/OS プログラミング・ガイド」の『TEST』を参照してください。

TEST オプションでコンパイルされたプログラムのパフォーマンス・トレードオフを要約すると、以下のようになります。

- NOTEST は TEST(NOEJPD) よりパフォーマンスが良好です。
- TEST(NOEJPD) は TEST(EJPD) よりパフォーマンスが大幅に良好です。

TEST(EJPD) では JUMPTO コマンドと GOTO コマンドが使用可能であるため、コンパイラーによって実行される最適化量に厳しい制限が課されます。EJPD サブオプションは、コンパイラーにステートメント内最適化の制限を課し、JUMPTO コマンドと GOTO コマンドが適切に動作できるようにします。

注: TEST(NOEJPD) およびゼロ以外の OPTIMIZE レベルを指定した場合: JUMPTO および GOTO コマンドは有効になりませんが、SET WARNING OFF コマンドを使用する場合は JUMPTO および GOTO を使用できます。このシナリオでは、JUMPTO と GOTO の結果は予測できません。

TEST(NOEJPD) も最適化プログラムに制限を課しますが、TEST(EJPD) ほどではありません。NOEJPD サブオプションを使用すると、ステートメント境界のデータ項目を表示できます。また、一部のデッド・コードとデッド・ストアの削除に関して、最適化プログラムに制限を課します。

下表に、一連の IBM 内部パフォーマンス・ベンチマークにおける、平均的な実行時パフォーマンスの数値を示します。

これらの数値は、さまざまな TEST サブオプションを使用して、OPT(1) および OPT(2) で得られたものです。パーセンテージは、NOTEST に対する TEST(NOEJPD) または TEST(EJPD) のパフォーマンス低下を示しています。

表 4. NOTEST に対する TEST(NOEJPD) または TEST(EJPD) のパフォーマンス低下		
OPT レベル	NOTEST に対する TEST(NOEJPD) の低下率 (%)	NOTEST に対する TEST(EJPD) の低下率 (%)
OPT(1)	1.7%	13.6%
OPT(2)	0.94%	13.6%

予期されたとおり、これは TEST(EJPD) のパフォーマンスへの影響が、TEST(NO EJPD) に比べてはるかに大きいことを示しています。

関連参照

TEST (Enterprise COBOL for z/OS プログラミング・ガイド)

THREAD

デフォルト

NOTHREAD

推奨

NOTHREAD

理由

THREAD オプションは生成されたコードおよび COBOL ランタイム・ライブラリーで追加ロックを必要するため、パフォーマンスが影響を受けるおそれがあります。この追加ロックは、プログラムがマルチスレッド環境で実行されていない場合は不要です。

これは、Enterprise COBOL 6 だけではなく、旧 Enterprise COBOL コンパイラーにも該当します。

THREAD オプションは、複数の POSIX スレッドまたは PL/I タスクがある環境で COBOL プログラムを実行可能にする必要があることを指定します。これを行うために、コンパイラーは、生成されたコード内のさまざまな場所にロックを挿入して、実行を保護します。これにより、THREAD でコンパイルされたプログラムのパフォーマンスは、対応する NOTHREAD プログラムに比べて影響を受けるおそれがあります。

プログラムが THREAD オプションを必要としない限り、NOTHREAD オプションを使用することをお勧めします。コンパイラーのデフォルトは NOTHREAD です。

コンパイラーが保護のためにロックを挿入する必要がある例として、入出力があります。THREAD オプションを使用すると、すべての入出力動詞 (OPEN、READ、WRITE、REWRITE、CLOSE など) はロックによって保護されます。測定では、THREAD オプションに起因する 10% のパフォーマンス低下が確認されています。

関連参照

THREAD (Enterprise COBOL for z/OS プログラミング・ガイド)

TRUNC

COBOL 4 の場合と同様に、TRUNC オプションには BIN、STD、および OPT という 3 つの使用可能な設定があります。

最良のパフォーマンスを得るための推奨オプションは、従来どおり TRUNC(OPT) です。このオプションは、コンパイラーが最も効率的にコードを生成できる自由度が最も高いためです。指定する TRUNC オプションを判断する方法については、「Enterprise COBOL for z/OS プログラミング・ガイド」の『TRUNC』を参照してください。

TRUNC(STD) を使用するコストは、COBOL 4 に比べて向上しました。これは、BINARY 受け取り側データ項目の PICTURE 節の桁数に合わせて結果を切り捨てるために使用される除算命令が、COBOL 6 では条件付きでのみ実行されるためです。コンパイラーは、オーバーフローのランタイム・チェックを挿入し、切り捨てが不要な場合は除算を迂回します。

ただし、TRUNC(OPT) の使用時には、ランタイム・オーバーフロー・チェックまたは除算命令が一切不要であるため、さらに良いパフォーマンスが得られます。

TRUNC(BIN) を使用すると、多くの場合パフォーマンスは低下します。このオプションは通常、3 つの TRUNC サブオプションのうちで最も低速です。結果を切り捨てるために除算 (条件付きまたはそれ以外) が不要であっても、2、4、または 8 バイトの完全な値が有効とみなされるため、中間結果はすぐにそれだけ大きくなり、より大きい、または複雑なデータ型への変換が必要になります。

例えば、TRUNC(STD) または TRUNC(OPT) を使用して 2 つの BINARY PIC 9(10) 値を加算した場合、最大結果サイズは 11 桁です。オーバーフローはできません。加算は、2 進算術計算を使用して実行できます。同じ加算を TRUNC(BIN) を使用して実行した場合、各オペランドは最大 18 桁まで使用でき、最大結果サイズは 19 桁です。これはオーバーフローが可能です。そのため、加算を実行する前に、オペランドをパック 10 進数に変換する必要があります。こちらのほうが低速です。

同様に、TRUNC(STD) または TRUNC(BIN) を使用して 2 つの BINARY PIC 9(10) 値を乗算した場合、最大結果サイズは 20 桁です。これは 2 進数演算には大きすぎますが、パック 10 進数演算には大きすぎません。同じ乗算を TRUNC(BIN) を使用して実行した場合、各オペランドは最大 18 桁まで使用でき、最大結果サイズは 36 桁です。パック 10 進数演算のハードウェア・サポートは 31 桁に制限されているため、この乗算は負荷の高いランタイム呼び出しを必要とします。

具体的には、2 つの BINARY PIC 9(10) 項目を加算する場合、TRUNC(OPT) を使用すると、TRUNC(STD) よりも 0.4% 速く、TRUNC(BIN) よりも 69% 速くなります。

大量の 2 進数演算を行うあるプログラムでは、設定 TRUNC(BIN) を使用すると、TRUNC(STD) と比較して 19% 遅くなりました。このパフォーマンスの差は、サイズの大きい中間結果に必要なランタイム・ライブラリー呼び出しが原因です。

BINARY データについて、また TRUNC サブオプションとの相互作用について詳しくは、[51 ページの『BINARY \(COMP または COMP-4\)』](#)を参照してください。

関連参照

TRUNC (Enterprise COBOL for z/OS プログラミング・ガイド)

TUNE

Default

ARCH が指定されたものの場合、過怠の調整段階は ARCH レベルと一致します。ARCH が未指定の場合は、ARCH と調整の両方の過怠が 10 になります。

推奨

TUNE(x)。x はアプリケーションが最も頻繁に実行されるハードウェアのレベルです。ほとんどのユーザーの場合、これは実稼働環境のハードウェア・レベルです。

理由

ARCH は、新しいハードウェアで使用可能な命令を使用せずに、対応するハードウェアで使用可能な命令を使用するようにコンパイラーに指示します。コンパイラーが使用できる命令に制限があるとしても、特定の COBOL ステートメントに対して複数の命令シーケンスの中からコンパイラーを選択しなければならない場合があります。TUNE(x) は、対応するハードウェア・レベルで最適となる命令のシーケンスを選択するようにコンパイラーに指示します。

多くのユーザーは、実稼働環境で新しいマシンを使用し、災害復旧環境では古いマシンを使用しています。例えば、実稼働環境に z16 があり、災害復旧環境に z15 があるとします。アプリケーションが災害復旧環境で実行されるようにするには、ARCH(13) を選択する必要があります。ただし、TUNE(14) も選択する必要があります。これにより、コンパイラーが、選択される命令を z15 マシン上で使用可能な命令に制限しながらも、z16 ハードウェアに最適な選択を行うようになります。その結果、z15 マシン上で機能しつつ、z16 マシンでも可能な限り高いパフォーマンスを発揮するアプリケーションになります。

考慮事項

ハードウェア・レベルとのマッチング以外は、特にありません。

コンパイラーは、TUNE オプションの影響を受ける多くの決定を行います。ほとんどの場合、古い命令より新しい命令の方が望ましいため、[1 ページの『アーキテクチャーの利用』](#)で説明されているように、コンパイラーは新しい命令を使用するシーケンスを選択します。その他の状況では、特定のハードウェア・モデルをターゲットにして、他のハードウェア・モデルをターゲットにする場合とは異なる命令シーケンスを選択することが望ましい場合があります。概して、TUNE の影響を受ける決定は、個別に見ると小さなものですが、累積的に顕著な影響をもたらす可能性があります。場合によっては、TUNE によるパフォーマンスの向上が顕著になることがあります。

例えば、z14 マシンでは、USAGE DISPLAY レシーバーを使用して計算を実行するプログラムは、ベクトル・パック 10 進数レジスターの中間結果が一時的な場所に保管され、その後レシーバーに直接保管

されるのではなくレシーバーの場所に移る場合、状況によってはパフォーマンスが向上します。他のハードウェア・モデルの場合は、最初に中間の一時的な場所に保管することに利点はないので、z15 マシンまたは z16 マシンでこの追加保管を行うと、最終的な場所に直接保管するよりもパフォーマンスが低下します。災害復旧環境には z14 マシンがあり、実稼働環境には z15 または z16 マシンがある場合は、ARCH(12) を使用しなければならない可能性があります。それぞれに TUNE(13) または TUNE(14) を使用すると、追加のストアの生成を回避できます。ループで USAGE DISPLAY 計算を行うプログラムは、ARCH(12) および TUNE(14) を使用した場合、ARCH(12) および TUNE(12) を使用した場合よりも、z16 マシンで 21.5% 高速でした。

ARCH 設定より TUNE 設定を高くすると、ARCH レベルと一致するマシンでプログラム速度が遅くなる可能性があることに注意してください。例えば、実稼働環境に z16 マシンがあり、災害復旧環境に z15 マシンがある場合に、ARCH(13) および TUNE(14) を使用すると、ARCH(13) および TUNE(13) を使用する場合より、z16 マシンではパフォーマンスが向上しますが、z15 マシンでは、パフォーマンスが低下する可能性があります。ただし、プログラムは実稼働環境で最も頻繁に実行されるため、プログラムが実稼働環境で実行される一般的なケースでは、ARCH(13) と TUNE(14) を選択するとパフォーマンスが向上します。そのため、システム全体では、ARCH(13) と TUNE(13) を選択して災害復旧環境でパフォーマンスの優先順位付けを行うよりもパフォーマンスが向上します。

関連参照

14 ページの『ARCH』

TUNE (Enterprise COBOL for z/OS プログラミング・ガイド)

プログラムの常駐とストレージの考慮事項

コンパイラー・オプション

以下のコンパイラー・オプションは、プログラムの常駐場所 (16 MB 境界の上または下)、WORKING-STORAGE セクションの場所、および入出力ファイルのバッファーとレコード域に影響する可能性があります。

RENT または NORENT

RENT コンパイラー・オプションを使用すると、コンパイラーは、プログラムを再入可能にするためのいくつかの追加コードを生成します。再入可能プログラムは、リンク・パック域 (LPA) または拡張リンク・パック域 (ELPA) などの共有ストレージに置くことができます。RENT オプションはまた、プログラムを 16 MB 境界より上で実行できるようにします。再入可能コードを作成すると、実行時間パスの長さがわずかに増える可能性があります。

注: RMODE(ANY) オプションは、NORENT プログラムを 16 MB 境界より上で実行するために使用できません。

RENT を使用した場合のパフォーマンス考慮事項: 平均して、RENT は NORENT と同等でした。

詳しくは、Enterprise COBOL for z/OS プログラミング・ガイド内の RENT を参照してください。

RMODE - AUTO、24、または ANY

RMODE コンパイラー・オプションは、COBOL プログラムの RMODE 設定を決定します。RMODE(AUTO) を使用すると、RMODE 設定は RENT または NORENT の使用に依存します。RENT の場合、プログラムは RMODE ANY になります。NORENT の場合、プログラムは RMODE 24 になります。RMODE(24) を使用すると、プログラムは常に RMODE 24 になります。RMODE(ANY) を使用すると、プログラムは常に RMODE ANY になります。

注: NORENT を使用する場合、WORKING-STORAGE が常駐する場所は RMODE オプションによって制御されます。RMODE(24) では、WORKING-STORAGE は 16 MB 境界より下に置かれます。RMODE(ANY) では、WORKING-STORAGE を 16 MB 境界より上に置くことができます。

これによってアプリケーションのパフォーマンスが影響を受けることはない想定されますが、プログラムと WORKING-STORAGE の配置場所は影響を受ける可能性があります。

詳しくは、Enterprise COBOL for z/OS プログラミング・ガイド内の RMODE を参照してください。

ストレージの場所

WORKING-STORAGE

COBOL の WORKING-STORAGE は、プログラムが RENT オプションでコンパイルされた場合、Language Environment ヒープ・ストレージから割り振られます。

LOCAL-STORAGE

COBOL の LOCAL-STORAGE は、常に Language Environment スタック・ストレージから割り振られます。これは LE STACK ランタイム・オプションの影響を受けます。

外部変数

Enterprise COBOL プログラムの外部変数は、常に Language Environment ヒープ・ストレージから割り振られます。

QSAM バッファ

QSAM バッファは、以下のすべての条件を満たしている場合、16 MB 境界より上に割り振ることができます。

- プログラムが、VS COBOL II 1.3 以降、COBOL/370 1.1 以降、IBM COBOL for MVS™ & VM 1.2 以降、IBM COBOL for OS/390® & VM、または IBM Enterprise COBOL でコンパイルされている。
- プログラムが RENT および DATA(31) でコンパイルされているか、NORENT および RMODE(ANY) でコンパイルされている
- プログラムが AMODE 31 で実行されている
- ランタイム・オプション ALL31(ON) および HEAP(,ANYWHERE) が (EXTERNAL ファイルに) 使用されている
- ファイルが TSO 端末に割り振られていない
- ファイルが外部でスパンされていないか、SAME RECORD 節でスパンされているか、または入出力としてオープンでスパンされ REWRITE で更新されている

詳しくは、「Enterprise COBOL for z/OS プログラミング・ガイド」の『QSAM ファイル用のバッファの割り振り』を参照してください。

VSAM バッファ

プログラムが、VS COBOL II 1.3 以降、COBOL/370 1.1 以降、IBM COBOL for MVS & VM 1.2 以降、IBM COBOL for OS/390 & VM、または IBM Enterprise COBOL でコンパイルされている場合は、VSAM バッファを 16 MB 境界より上に割り振ることができます。

第4章 ランタイム・パフォーマンスに影響するランタイム・オプション

適切なランタイム・オプションを選択すると、COBOL アプリケーションのパフォーマンスに効果があります。

そのため、適切なランタイム・オプションがインストール済み環境用に正しくセットアップされるように、LE 環境のインストールとセットアップを担当するシステム・プログラマーは、アプリケーション・プログラマーと連携することが重要です。また、高速なパフォーマンスを必要とする特定のプログラム、アプリケーション、および領域に適切なオプションを設定できるように、これらのオプションについて理解することも重要です。個々の LE ランタイム・オプションは、個々のオプションを設定するためにサポートされているどの方法を使用しても設定できます。ここでは、個々のアプリケーションだけでなく、LE ランタイム環境全体のパフォーマンスを向上させるために役立ついくつかのオプションについて検討します。

注：以下のオプションの説明では、オプションの設定が CICS と CICS 以外とで異なる場合は、設定が括弧付きテキストで修飾されています。その他の場合は、同じ設定が CICS と CICS 以外の両方に適用されます。

関連参照

ランタイム・オプションの使用 (z/OS Language Environment プログラミング・ガイド)Language Environment ランタイム・オプションの要約

(z/OS Language Environment プログラミング・リファレンス)Language Environment ランタイム・オプションの使用

(z/OS Language Environment プログラミング・リファレンス)Language Environment ランタイム・オプション

(z/OS Language Environment カスタマイズ)

AIXBLD

デフォルト

OFF (CICS 以外)、N/A (CICS)

推奨

OFF (CICS 以外)、N/A (CICS)

考慮事項

AIXBLD オプションを使用すると、実行時に代替索引を作成できます。ただし、これはアプリケーションのランタイム・パフォーマンスに悪影響を与えるおそれがあります。AIXBLD ランタイム・オプションを使用するよりも、COBOL アプリケーションの実行前にアクセス方式サービス・プログラムを使用して代替索引を作成する方が効率的です。RLS モードで VSAM データ・セットにアクセスする場合、AIXBLD はサポートされないことに注意してください。

関連参照

AIXBLD (COBOL のみ) (z/OS Language Environment プログラミング・リファレンス)AIXBLD (COBOL のみ) (z/OS Language Environment カスタマイズ)

ALL31

デフォルト

ON

推奨

ON - アプリケーション内に AMODE(24) ルーチンがない場合

考慮事項

ALL31 オプションを使用すると、LE はアプリケーション内に AMODE(24) ルーチンがないという認識を利用できます。

ALL31(ON) は、アプリケーション全体を AMODE(31) で実行することを指定します。これにより、LE は共通ランタイム・ライブラリー・ルーチンの呼び出し全体でモード切り替え回数を最小化できるため、すべて AMODE(31) のアプリケーションのパフォーマンス向上に役立ちます。また、ALL31(ON) を使用すると、16 MB 境界より下の記憶域の使用量が減少するため、16 MB 境界より下の仮想記憶域の制約の問題をある程度緩和するために役立ちます。

ALL31(ON) の使用時に、HEAP(,ANYWHERE) ランタイム・オプションも使用していて、コンパイラー・オプション DATA(31) と RENT、または RMODE(ANY) と NORENT のどちらかでプログラムをコンパイルしている場合は、すべての EXTERNAL WORKING-STORAGE および EXTERNAL FD レコード域を 16 MB 境界より上に割り振ることができます。ALL31(OFF) を使用する場合は、STACK(,BELOW) も使用する必要があることに注意してください。

注:

- z/OS 1.2 用の LE 以降、ランタイム・デフォルトは ALL31(ON),STACK(,ANY) に変更されました。OS/390 2.10 用以前の LE では、ランタイム・デフォルトは ALL31(OFF),STACK(,BELOW) でした。
- CICS 下で実行されていないすべての OS/VS COBOL プログラム、すべての VS COBOL II NORES プログラム、およびその他すべての AMODE(24) プログラムでは、ALL31(OFF) が必須です。

パフォーマンス例 (CALL オーバーヘッドのみ測定) として、ALL31(ON) を使用したテスト・プログラムの結果は、ALL31(OFF) の場合と同じでした。

注: このテストでは、RENT プログラムの CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、LE 共通ランタイム・ルーチンへの呼び出し回数によっては異なる結果になる場合があります。

関連参照

ALL31 (z/OS Language Environment プログラミング・リファレンス)

ALL31 (z/OS Language Environment カスタマイズ)

CBLPSHPOP

デフォルト

ON

推奨

N/A (CICS 以外)、ON (CICS) - VS COBOL II との互換動作が EXEC CICS 条件処理コマンドで必要な場合。VS COBOL II との互換動作が不要な場合、またはプログラムが EXEC CICS 条件処理コマンドを使用しない場合、推奨設定は OFF です。

考慮事項

CBLPSHPOP オプションは、COBOL サブルーチンの呼び出し時に CICS PUSH HANDLE コマンドおよび CICS POP HANDLE コマンドを発行するかどうかを制御します。

このオプションは CICS 環境にのみ適用されます。CBLPSHPOP オプションは、CICS の CONDITION、AID、ABEND のいずれかの条件処理コマンドを含む COBOL サブルーチンの呼び出し時に、互換性の問題を回避するために使用します。

- CBLPSHPOP が OFF であり、COBOL サブプログラム内でこれらの CICS 条件を処理したい場合は、COBOL サブプログラムを呼び出す前に独自の CICS PUSH HANDLE を発行し、戻り時に CICS POP HANDLE を発行することが必要になります。そうしないと、COBOL サブルーチンが呼び出し元の設定を継承し、戻り時には、呼び出し元がサブプログラムで行われたすべての設定を継承します。この動作は VS COBOL II の動作とは異なります。
- CBLPSHPOP が ON の場合は、CICS 条件処理コマンドを使用すると、VS COBOL II ランタイムと同じ動作が行われます。ただし、呼び出しのパフォーマンスは影響を受けます。

CBLPSHPOP 使用時のパフォーマンス考慮事項については、[46 ページの『CICS』](#)を参照してください。

関連タスク

COBOL プログラムの開発 (CICS の場合)

関連参照

CICS 環境での CBLPSHPOP ランタイム・オプションの使用

(z/OS Language Environment プログラミング・ガイド)CBLPSHPOP (COBOL のみ) (z/OS Language

Environment プログラミング・リファレンス)CBLPSHPOP (COBOL のみ) (z/OS Language Environment カスタマイズ)

CHECK

CHECK ランタイム・オプションは、Enterprise COBOL 6 でコンパイルされたアプリケーションでは無視されます。

コンパイル時オプション SSRANGE が指定されている場合は、範囲検査がコンパイラーによって生成され、実行時に常にその検査が実行されます。コンパイルで組み込まれた範囲検査を無効にすることはできません。

関連参照

CHECK (COBOL のみ) (z/OS Language Environment プログラミング・リファレンス)CHECK (COBOL のみ)

(z/OS Language Environment カスタマイズ)

DEBUG

デフォルト

OFF

推奨

OFF

考慮事項

DEBUG オプションは、USE FOR DEBUGGING 宣言で指定される COBOL のバッチ・デバッグ機能を活性化します。これにより、デバッグ・ステートメントの処理に多少のオーバーヘッドが追加される可能性があります。このオプションは、USE FOR DEBUGGING 宣言のあるプログラムにのみ影響します。

DEBUG を使用した場合のパフォーマンス考慮事項は以下のとおりです。

- USE FOR DEBUGGING 宣言を使用しない場合、平均すると DEBUG は NODEBUG と同等でした。
- USE FOR DEBUGGING 宣言を使用した場合、測定されたテスト・プログラムは、DEBUG 使用時には NODEBUG 使用時に比べて 900% 低速でした。

注：このテストでのプログラムには、SOURCE-COMPUTER 段落に WITH DEBUGGING MODE 節があり、USE FOR DEBUGGING ON という段落名が手続き部に含まれていました。この段落は空であり (EXIT ステートメントのみを含む)、ループ内で何度も実行されます。宣言セクション内の段落も空 (EXIT ステートメントのみ) です。この目的は、USE FOR DEBUGGING 宣言に制御が移ることによるオーバーヘッドを示すことです。

関連参照

DEBUG (COBOL のみ) (z/OS Language Environment プログラミング・リファレンス)DEBUG (COBOL のみ)

(z/OS Language Environment カスタマイズ)

INTERRUPT

デフォルト

OFF (CICS 以外)、N/A (CICS)

推奨

OFF (CICS 以外)、N/A (CICS)

考慮事項

INTERRUPT オプションは、アテンション割り込みを Language Environment に認識させます。割り込みを発生させると、Language Environment はアプリケーションまたは Debug Tool に制御を渡すことができます。

INTERRUPT を使用した場合のパフォーマンス考慮事項: 平均して、INTERRUPT(ON) は INTERRUPT(OFF) よりも 1% 低速です。最大で 20% 低速です。

関連参照

INTERRUPT (z/OS Language Environment プログラミング・リファレンス)

INTERRUPT (z/OS Language Environment カスタマイズ)

RPTOPTS

デフォルト

OFF

推奨

OFF

考慮事項

RPTOPTS オプションを使用すると、アプリケーションの実行中に使用されていたランタイム・オプションのレポートを入手できます。このレポートはアプリケーションの終了後に生成されます。そのため、アプリケーションが異常終了した場合は、レポートが生成されないことがあります。レポートの生成によって、いくらかの追加オーバーヘッドが生じる可能性があります。RPTOPTS(OFF) を指定すると、このオーバーヘッドは排除されます。

RPTOPTS を使用した場合のパフォーマンス考慮事項: 平均して、RPTOPTS(ON) は RPTOPTS(OFF) と同等でした。

注: 単一バッチ・プログラムの平均は、RPTOPTS(ON) に関して同等のパフォーマンスを示していますが、メインプログラムが繰り返し起動されるトランザクション環境 (例えば CICS) では、いくらかパフォーマンスが低下する場合があります。

関連参照

RPTOPTS (z/OS Language Environment プログラミング・リファレンス)

RPTOPTS (z/OS Language Environment カスタマイズ)

RPTSTG

デフォルト

OFF

推奨

OFF

考慮事項

RPTSTG オプションを使用すると、アプリケーションで使用されていたストレージに関するレポートを入手できます。このレポートはアプリケーションの終了後に生成されます。そのため、アプリケーションが異常終了した場合は、レポートが生成されないことがあります。このレポートのデータは、アプリケーションのストレージ・パラメーターを微調整するために役立ち、LE ストレージ・マネージャーがストレージを獲得または解放するためにシステム要求を行わなければならない回数を減らすことができます。

データの収集およびレポートの生成によって、いくらかの追加オーバーヘッドが生じる可能性があります。RPTSTG(OFF) を指定すると、このオーバーヘッドは排除されます。

RPTSTG を使用した場合のパフォーマンス考慮事項: 呼び出し集中型のプログラムでのパフォーマンス低下は 200% を超えると測定されています。

注: プログラムでは、いくつかの空の (つまり、GOBACK ステートメントのみを含む) サブプログラムを繰り返し呼び出す以外には何も行っていません。

関連参照

RPTSTG (z/OS Language Environment プログラミング・リファレンス)

RPTSTG (z/OS Language Environment カスタマイズ)

RTEREUS

デフォルト

OFF (CICS 以外)、N/A (CICS)

推奨

OFF (CICS 以外)、N/A (CICS)

考慮事項

RTEREUS オプションは、最初の COBOL プログラムの起動時に、LE ランタイム環境を再利用のために初期化します。

LE ランタイム環境は初期化状態を維持し (すべての COBOL プログラムとその作業域がストレージ内に保持されます)、またライブラリー・ルーチンも初期化状態を維持し、ストレージに保持されます。これは、以降の COBOL プログラムの呼び出しでは、ほとんどのランタイム環境の初期化が迂回されることを意味します。STOP RUN が実行されない限り、または環境を終了するための明示的な呼び出しが行われない限り、ほとんどのランタイム終了も迂回されます (注: STOP RUN を使用すると、最初の COBOL プログラムを起動したルーチンの呼び出し元に制御が返され、再利用可能ランタイム環境が終了します)。

STOP RUN ステートメントはランタイム環境に作用するため、RTEREUS の利点を活かすには、すべての STOP RUN ステートメントを GOBACK ステートメントに変更する必要があります。最も顕著な影響は、COBOL サブプログラムを繰り返し呼び出している非 COBOL ドライバー (例えば、COBOL アプリケーションを繰り返し呼び出す LE 非標準アセンブラー・ドライバー) のパフォーマンスで認められます。この事例で RTEREUS オプションが役立ちます。

ただし、RTEREUS オプションを使用すると、COBOL アプリケーションのセマンティクスが影響を受けます。各 COBOL プログラムはサブプログラムと見なされ、以降の起動では最後に使用された状態になります (プログラムを初期状態にしたい場合は、PROGRAM-ID ステートメントで INITIAL 節を使用できます)。これは、アプリケーションの実行中に獲得されたストレージが解放されないことを意味します。ストレージが解放されないため、SVC LINK で RTEREUS を使用することはできません。SVC LINK から戻った後、LINK されたプログラムはオペレーティング・システムによって削除されますが、COBOL 制御ブロックは初期化されたままストレージ内に保持されるからです。したがって、すべての環境に RTEREUS を適用できるわけではありません。

RTEREUS を使用した場合のパフォーマンス考慮事項 (CALL オーバーヘッドのみを測定): RTEREUS を使用したあるテスト・ケース (COBOL を呼び出す LE 非標準アセンブラー) では、NORTEREUS を使用した場合より 99% 高速でした。

注: このテストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、異なる結果になる場合があります。

64 ビットの考慮事項

COBOL レガシー・ランタイム再使用オプションは 64 ビット・プログラムではサポートされていません。再使用可能なランタイム環境を確立するには、LE 初期設定済み環境機能を使用してください。

関連トピック アセンブラー・ドライバーを使用するアプリケーションのアップグレード
(Enterprise COBOL for z/OS 移行ガイド)

関連参照

RTEREUS (COBOL のみ) (z/OS Language Environment プログラミング・リファレンス) RTEREUS (COBOL のみ) (z/OS Language Environment カスタマイズ) COBOL と Language Environment のランタイム・オプションの比較
(z/OS Language Environment ランタイム・アプリケーション マイグレーション・ガイド)

STORAGE

デフォルト

NONE,NONE,NONE,OK

推奨

NONE,NONE,NONE,OK

考慮事項

STORAGE オプションは、ヒープ割り振りまたはスタック・ストレージを指定します。

このオプションの最初のパラメーターは、外部データ用のストレージが割り振られると、プログラムが獲得したすべての外部データ・レコードを含め、すべてのヒープ割り振りを指定の値に初期化します。これには、RENT プログラムが最初に呼び出されたとき、または動的呼び出しの場合にはプログラムが取り消されてから再度呼び出されたときに、VALUE 節がデータ項目で使用されていない限り、このプログラムが獲得した WORKING-STORAGE も含まれます (下の注を参照)。いずれの場合も、ストレージは以降のプログラムの呼び出しでは初期化されません。これにより、プログラム内の外部データ・レコード数と WORKING-STORAGE セクションのサイズに応じて、実行時にいくらかのオーバーヘッドが生じる可能性があります。

WORKING-STORAGE は、RENT プログラムの以下のカテゴリーで STORAGE オプションに影響を受けます。

- プログラムが CICS 環境で実行されている場合
- プログラムが Enterprise COBOL 4.2 以前でコンパイルされている場合
- プログラムが Enterprise COBOL 6.1 以降でコンパイルされている場合
- (プログラムが常駐する) プログラム・オブジェクトに、COBOL 5.1.1 以降または COBOL 4.2 以前のコンパイラでコンパイルされたプログラムのみが含まれる (つまり、そのプログラム・オブジェクト内に Language Environment 言語間呼び出しがない) 場合
- プログラム・オブジェクトの 1 次エントリー・ポイントが、Enterprise COBOL 5.1.1 以降でコンパイルされたプログラムである (つまり、プログラム・オブジェクト内に LE 言語間呼び出しを持つことが許可されている) 場合

注: WSCLEAR オプションを VS COBOL II で使用した場合は、STORAGE(00,NONE,NONE) が Language Environment での同等のオプションです。

このオプションの 2 番目のパラメーターは、解放時にすべてのヒープ・ストレージを初期化します。

このオプションの 3 番目のパラメーターは、割り振り時にすべての DSA (スタック) ストレージを初期化します。オーバーヘッド量は、呼び出されるルーチン数 (サブルーチンとライブラリー・ルーチン) および使用される LOCAL-STORAGE データ項目数によって異なります。このオーバーヘッドは、呼び出し集中型アプリケーションの CPU 時間に重大な影響を与えるおそれがあります。アプリケーションの変数を初期化するために、STORAGE(,00) を使用してはなりません。代わりに、特定の変数を初期化するようにアプリケーションを変更する必要があります。STORAGE(,00) をパフォーマンスが重要なアプリケーションで使用してはなりません。

STORAGE を使用した場合のパフォーマンス考慮事項は以下のとおりです。

- 平均して、STORAGE(00,00,00) は STORAGE(NONE,NONE,NONE) よりも 11% 低速であり、最大で 133% 低速でした。RENT サブプログラムを呼び出し、40 MB の WORKING-STORAGE を持つ、ある RENT プログラムは、28% 低速でした。呼び出し集中型アプリケーションの使用時には、200% 以上低速になる可能性があることに注意してください。
- 平均して、STORAGE(00,NONE,NONE) は STORAGE(NONE,NONE,NONE) と同等でした。RENT サブプログラムを呼び出し、40 MB の WORKING-STORAGE を持つ、ある RENT プログラムは、5% 低速でした。
- 平均して、STORAGE(NONE,00,NONE) は STORAGE(NONE,NONE,NONE) と同等でした。RENT サブプログラムを呼び出し、40 MB の WORKING-STORAGE を持つ、ある RENT プログラムは、9% 低速でした。
- 呼び出し集中型プログラムの場合、STORAGE(NONE,NONE,00) を使用すると、呼び出し数によっては低下率が 100% を超える可能性があります。

注: 呼び出し集中型テストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、さほど機能低下はありません。

関連参照

STORAGE (z/OS Language Environment プログラミング・リファレンス)

STORAGE (z/OS Language Environment カスタマイズ)

COBOL と Language Environment のランタイム・オプションの比較

(z/OS Language Environment ランタイム・アプリケーション マイグレーション・ガイド)

TEST

Default

NOTEST(ALL,*,PROMPT,INSPREF)

推奨

NOTEST(ALL,*,PROMPT,INSPREF)

考慮事項

TEST オプションは、ユーザー・アプリケーションが起動された場合に z/OS Debugger でどのような制御を行うかを指定します。

指定すると、z/OS Debugger が初期化されて呼び出される可能性があるため、TEST の使用時には追加のオーバーヘッドが発生する可能性があります。NOTEST を指定すると、このオーバーヘッドは排除されます。

関連参照

TEST | NOTEST (z/OS Language Environment プログラミング・リファレンス)

TEST | NOTEST (z/OS Language Environment カスタマイズ)

TRAP

デフォルト

ON,SPIE

推奨

ON,SPIE

考慮事項

TRAP オプションを使用すると、LE は異常終了 (アベンド) をインターセプトし、異常終了情報を提供して、LE ランタイム環境を終了することができます。

TRAP(ON) はまた、異常終了の発生時にすべてのファイルを確実にクローズします。オーバーフロー条件に関して算術演算ステートメントの ON SIZE ERROR 節を適切に処理するために必要です。さらに、LE は内部的に条件処理を使用するため、TRAP(ON) を必要とします。TRAP(OFF) は、LE が異常終了をインターセプトしないようにします。一般的に、TRAP(ON) を使用しても、COBOL アプリケーションのパフォーマンスに対する重大な影響はありません。

SPIE サブオプションを使用すると、LE は ESPIE を発行して、プログラム割り込みを処理します。

NOSPIE サブオプションを使用すると、LE は ESTAE を介してプログラム割り込みを処理します。

TRAP を使用した場合のパフォーマンス考慮事項: 平均して、TRAP(ON) は TRAP(OFF) と同等でした。

関連タスク

QSAM ファイルのクローズ (Enterprise COBOL for z/OS プログラミング・ガイド)

VSAM ファイルのクローズ (Enterprise COBOL for z/OS プログラミング・ガイド)

行順次ファイルのクローズ (Enterprise COBOL for z/OS プログラミング・ガイド)

算術演算でのエラーの処理

(Enterprise COBOL for z/OS プログラミング・ガイド)

関連参照

Language Environment ランタイム・オプション

(Enterprise COBOL for z/OS 移行ガイド)条件処理プロセスへの TRAP の影響

(z/OS Language Environment プログラミング・ガイド)TRAP ランタイム・オプションおよびユーザー作成条件処理ルーチン

(z/OS Language Environment プログラミング・ガイド)TRAP ランタイム・オプションおよび CEEBXITA

(z/OS Language Environment プログラミング・ガイド)TRAP (z/OS Language Environment プログラミング・リファレンス)
TRAP (z/OS Language Environment カスタマイズ)

VCTRSAVE

デフォルト

OFF (CICS 以外)、N/A (CICS)

推奨

OFF (CICS 以外)、N/A (CICS)

考慮事項

VCTRSAVE オプションは、ユーザー提供の条件ハンドラーの呼び出し時に、アプリケーションの言語でベクトル機構を使用するかどうかを指定します。

CEEHDLR に登録されたユーザー作成条件ハンドラーがないか、ユーザー作成条件ハンドラーが、ARCH(11) のある INSPECT ステートメントで生成されたベクトル機構命令を取得していない場合、このオーバーヘッドを回避するには、VCTRSAVE(OFF) を指定して実行する必要があります。

VCTRSAVE を使用した場合のパフォーマンス考慮事項: 平均して、VCTRSAVE(ON) は VCTRSAVE(OFF) と同等でした。

関連参照

VCTRSAVE (z/OS Language Environment プログラミング・リファレンス)

VCTRSAVE (z/OS Language Environment カスタマイズ)

第5章 ランタイム・パフォーマンスに影響する COBOL および LE の機能

COBOL および Language Environment には、アプリケーションのパフォーマンス向上に役立つ、インストレーションおよび環境チューニング機能がいくつかあります。

ここでは、アプリケーションに関して考慮が必要ないくつかの追加要因について説明します。

ストレージ管理チューニング

ストレージ管理チューニングにより、アプリケーション・プログラム用のストレージの取得および解放に関連するオーバーヘッドを削減することができます。適切なチューニングを行うことで、いくつかの GETMAIN 呼び出しと FREEMAIN 呼び出しを除去できます。

まず第一に、ストレージ管理は、ストレージのブロックを必要な期間だけ保持するよう設計されています。つまり、COBOL プログラムの実行中に、ストレージのいずれかのブロックが使用されなくなった場合、そのブロックは解放されます。この方法は、他のランザクション (またはアプリケーション) でストレージを効率的に使用できるように、できるだけ早くストレージを解放したいランザクション環境で (またはどの環境でも) 役立ちます。

ただし、ライブラリー・ルーチンによるストレージ要求を満たすために十分なフリー・スペースがストレージの最後のブロックにない場合、この方法は有害となるおそれもあります。例えば、ライブラリー・ルーチンが 2K のストレージを必要としていても、ストレージの最後のブロックで使用可能なストレージは 1K のみであるとします。ライブラリー・ルーチンは、ストレージ管理を呼び出して、2K のストレージを要求します。ストレージ管理は、最後のブロックに十分なストレージがないと判断し、このストレージを獲得するために GETMAIN を発行します (この GETMAIN のサイズもチューニングできます)。ライブラリー・ルーチンはこのストレージを使用し、完了すると、ストレージ管理を呼び出して、この 2K のストレージが不要になったことを伝えます。ストレージ管理は、ストレージのこのブロックが使用されなくなったことを確認すると、FREEMAIN を発行し、このストレージを解放してオペレーティング・システムに戻します。

ここで、このライブラリー・ルーチンまたは 1K を超えるストレージを必要とする他のライブラリー・ルーチンが頻繁に呼び出された場合、GETMAIN および FREEMAIN のアクティビティー量が原因で、CPU 時間の多大なパフォーマンス低下が生じるおそれがあります。

幸いにも、LE にはこれを補うためのストレージ管理チューニングと呼ばれる方法があります。特定のアプリケーション・プログラムに使用する値を決定する際に、RPTSTG(ON) ランタイム・オプションが役立ちます。RPTSTG(ON) オプションによって返される値を、HEAP、ANYHEAP、BELOWHEAP、STACK、および LIBSTACK の各ランタイム・オプションの初期ストレージ・ブロックのサイズとして使用します。これにより、VS COBOL II、COBOL/370、COBOL for MVS & VM、COBOL for OS/390 & VM、または Enterprise COBOL のすべてのアプリケーションで、上記の問題の発生を防ぐことができます。ただし、頻繁に呼び出される OS/VS COBOL プログラムもアプリケーションに含まれている場合、RPTSTG(ON) オプションでは、追加ストレージの必要性を示せないことがあります。このような混合環境では、これらの初期値を大きくすることでも、ストレージ管理アクティビティーをある程度削減できます。

バッチ・アプリケーション用の IBM 提供デフォルト・ストレージ・オプションを以下に示します。

```
ANYHEAP(16K,8K,ANYWHERE,FREE)
BELOWHEAP(8K,4K,FREE)
HEAP(32K,32K,ANYWHERE,KEEP,8K,4K)
LIBSTACK(4K,4K,FREE)
STACK(128K,128K,ANYWHERE,KEEP,512K,128K)
THREADHEAP(4K,4K,ANYWHERE,KEEP)
THREADSTACK(OFF,4K,4K,ANYWHERE,KEEP,128K,128K)
```

COBOL アプリケーションのみを実行している場合は、次のようないくつかの追加ストレージ・チューニングを行うことができます。

```
STACK(64K,64K,ANYWHERE,KEEP)
```

CICS アプリケーション用の IBM 提供デフォルト・ストレージ・オプションを以下に示します。

```
ANYHEAP (4K,4080,ANYWHERE,FREE)
BELOWHEAP (4K,4080,FREE)
HEAP (4K,4080,ANYWHERE,KEEP,4K,4080)
LIBSTACK (32,4000,FREE)
STACK (4K,4080,ANYWHERE,KEEP,4K,4080)
```

すべてのアプリケーションが AMODE(31) である場合は、ALL31(ON) および STACK(,ANYWHERE) を使用できます。それ以外の場合は、ALL31(OFF) および STACK(,BELOW) を使用する必要があります。

16 MB 境界より下の全体的なストレージ要件は、デフォルト・ストレージ・オプションを小さくし、16 MB 境界より上のライブラリー・ルーチンの一部を移動することによって削減されました。

注: z/OS リリース 1.2 用の LE 以降、ランタイム・デフォルトは ALL31(ON),STACK(,ANY) に変更されました。OS/390 リリース 2.10 以前の LE では、ランタイム・デフォルトは ALL31(OFF),STACK(,BELOW) でした。

ストレージ・チューニング・ユーザー出口

CICS、IMS、その他のトランザクション処理タイプの環境など、Language Environment が頻繁に初期化および終了されている環境では、ストレージ・オプションをチューニングすることにより、アプリケーションのパフォーマンス全体を向上させることができます。

このチューニングは、GETMAIN および FREEMAIN のアクティビティー削減に役立ちます。Language Environment ストレージ・チューニング・ユーザー出口は、ご使用の環境に最適な値を選択するタスクを管理可能な 1 つの方法です。ストレージ・チューニング・ユーザー出口を使用すると、値をロード・モジュールにリンク・エディットする必要なしに、メインプログラムのストレージ値を設定できます。

関連参照

ストレージ・チューニング・ユーザー出口 (z/OS Language Environment カスタマイズ)

CEEENTRY マクロおよび CEETERM マクロの使用

COBOL を呼び出す LE 非準拠アセンブラーのパフォーマンスを向上させるために、アセンブラー・プログラムを LE 準拠にすることができます。これは、LE で提供されている CEEENTRY マクロおよび CEETERM マクロを使用して行うことができます。

このチューニングは、GETMAIN および FREEMAIN のアクティビティー削減に役立ちます。Language Environment ストレージ・チューニング・ユーザー出口は、ご使用の環境に最適な値を選択するタスクを管理可能な 1 つの方法です。ストレージ・チューニング・ユーザー出口を使用すると、値をロード・モジュールにリンク・エディットする必要なしに、メインプログラムのストレージ値を設定できます。

CEEENTRY マクロおよび CEETERM マクロを使用した場合のパフォーマンス考慮事項は以下のとおりです (CALL オーバーヘッドのみを測定)。

- CEEENTRY マクロおよび CEETERM マクロを使用したあるテスト・ケース (COBOL を呼び出す LE 準拠アセンブラー) では、これらのマクロを使用しない場合よりも 99 % 高速でした。

注: このテストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、異なる結果になる場合があります。

CEEENTRY および CEETERM を使用してその他の環境初期化手法と比較したパフォーマンスの追加の考慮事項については、[46 ページの『最初のプログラムが LE に非準拠』](#)を参照してください。

関連参照

CEEENTRY マクロ (z/OS Language Environment プログラミング・ガイド)

CEETERM マクロ (z/OS Language Environment プログラミング・ガイド)

事前初期設定サービス (CEEPIPI) の使用

LE 事前初期設定サービス (CEEPIPI) を使用して、COBOL を呼び出す LE 非準拠アセンブラーのパフォーマンスを向上させることもできます。

LE 事前初期設定サービスを使用すると、アプリケーションは LE 環境を 1 回初期化し、複数の LE 準拠プログラムを実行してから、明示的に LE 環境を終了することができます。これにより、アプリケーションの各プログラム用に LE 環境を初期化および終了するために必要だったシステム・リソースの使用が、大幅に削減されます。

CEEPIPI を使用して COBOL サブプログラムを呼び出す例については『Call_Sub での CEEPIPI の使用』を、CEEPIPI を使用して COBOL メインプログラムを呼び出す例については『Call_Main での CEEPIPI の使用』を参照してください。

CEEPIPI を使用した場合のパフォーマンス考慮事項は以下のとおりです (CALL オーバーヘッドのみを測定)。

- CEEPIPI を使用して COBOL プログラムをサブプログラムとして呼び出すテスト・ケース (COBOL を呼び出す LE 非準拠アセンブラー) では、CEEPIPI を使用しない場合よりも 99% 高速でした。
- CEEPIPI を使用して COBOL プログラムをメインプログラムとして呼び出す同じプログラムでは、CEEPIPI を使用しない場合よりも 95% 高速でした。

注：このテストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、異なる結果になる場合があります。

CEEPIPI を使用してその他の環境初期化技法と比較したパフォーマンスの追加の考慮事項については、[46 ページの『最初のプログラムが LE に非準拠』](#)を参照してください。

関連参照

事前初期設定サービスの使用 (z/OS Language Environment プログラミング・ガイド)

ライブラリー・ルーチン保存機能 (LRR) の使用

LRR は、以下の属性を持つ MVS 上で実行中のアプリケーションまたはサブシステムのパフォーマンスを向上させる機能です。

- アプリケーションまたはサブシステムが、LE を必要とするプログラムを呼び出す
- アプリケーションまたはサブシステムが LE 準拠ではない (つまり、アプリケーションまたはサブシステムが LE を必要とするプログラムを呼び出すときに、LE がまだ初期化されていない)。
- アプリケーションまたはサブシステムが、同じ MVS タスク下で実行中の LE を必要とするプログラムを繰り返し呼び出す
- アプリケーションまたはサブシステムが、LE 事前初期設定サービスを使用していない

LRR は、LE 準拠言語を繰り返し呼び出す LE 非準拠アセンブラー・ドライバー、および IMS/TM 領域に有益です。LRR は、CICS 下ではサポートされません。IMS 下での LRR の使用については、[48 ページの『IMS』](#)を参照してください。

LRR が初期設定されると、LE は、環境の終了後もそのリソースのサブセットをメモリー内に保持します。その結果、LE を初期設定した同じ MVS タスクでの以降のプログラム呼び出しは、リソースを再獲得も再初期設定もする必要なしに再利用できるため、高速化されます。LE 終了時に LE がメモリー内に保持するリソースには以下があります。

- LE ランタイム・ロード・モジュール
- これらのロード・モジュールに関連付けられたストレージ
- LE 開始制御ブロック用のストレージ

LRR の終了時に、これらのリソースはメモリーから解放されます。

LE 事前初期設定サービスおよび LRR は、同時に使用することができます。ただし、LE 事前初期設定サービスを使用している場合、LRR を使用することによる追加の利点はありません。基本的に、LRR がアクティブであり、LE 非準拠アプリケーションで事前初期設定サービスを使用している場合、LE 準拠プログラム

が繰り返し呼び出されている間、LE は事前初期設定された状態を維持し、終了しません。LE 非標準アプリケーションに戻る際には、事前初期設定サービスを呼び出して LE 環境を終了できます。この場合、LRR は再度有効になります。LRR の使用例については、[41 ページの『ライブラリー・ルーチン保存機能 \(LRR\) の使用』](#)を参照してください。

LRR を使用した場合のパフォーマンス考慮事項は以下のとおりです。

- LRR を使用したあるテスト・ケース (COBOL を呼び出す LE 非標準アセンブラー) では、LRR を使用しない場合よりも 96% 高速でした。

注: このテストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、異なる結果になる場合があります。

LRR を使用してその他の環境初期化技法と比較したパフォーマンスの追加の考慮事項については、[46 ページの『最初のプログラムが LE に非標準』](#)を参照してください。

関連参照

Language Environment ライブラリー・ルーチン保存機能 (LRR)
(z/OS Language Environment プログラミング・ガイド)IMS での言語環境プログラムの使用
(z/OS Language Environment カスタマイズ)

LPA/ELPA 内のライブラリー

COBOL ライブラリー・ルーチンと LE ライブラリー・ルーチンをリンク・パック域 (LPA) または拡張リンク・パック域 (ELPA) に置くことによっても、総合的なシステム・パフォーマンスを向上させることができます。

これにより、COBOL/370、COBOL for MVS & VM、COBOL for OS/390 & VM、Enterprise COBOL、VS COBOL II RES、または OS/VS COBOL RES 用のシステム全体の実ストレージ所要量が削減されます。これは、各アプリケーションがライブラリー・ルーチンのコピーを所有する代わりに、すべてのアプリケーションがライブラリー・ルーチンを共有できることによるものです。LPA/ELPA への配置に適格である COBOL ライブラリー・ルーチンのリストについては、SCEESAMP データ・セットのメンバー CEEWLPA と IGZWMLP4 を参照してください。

ライブラリー・ルーチンを共有域に置くことにより、入出力アクティビティーも削減されます。これは、ライブラリー・ルーチンがアプリケーション・プログラムごとではなく、システム始動時に 1 回のみロードされるためです。

関連参照

リンク・パック域に置くことができるモジュール
(z/OS Language Environment カスタマイズ)Language Environment でのリンクと実行の計画
(z/OS Language Environment ランタイム・アプリケーションマイグレーション・ガイド)

CALL の使用

すべての CALL 集中型アプリケーションのストレージ管理チューニングを検討する必要があります。

静的 CALL (NODYNAM を含む CALL リテラル) では、プログラムはすべて一緒にリンク・エディットされるため、それらのプログラムは、呼び出さない場合でも常にストレージ内にあります。ただし、アプリケーションとリンク・エディットされたブートストラッピング・ライブラリー・ルーチンのコピーは 1 つのみ存在します。動的 CALL (DYNAM または呼び出し ID を含む CALL リテラル) では、各サブプログラムは他のサブプログラムとは別個にリンク・エディットされます。これらのサブプログラムは、必要な場合にのみストレージに取り込まれます。これは、複雑なアプリケーションを管理するための良い方法です。ただし、各サブプログラムには、それ自体にリンク・エディットされたブートストラッピング・ライブラリー・ルーチンの専用コピーがあり、アプリケーションの実行時に、これらのルーチンの複数のコピーがストレージに取り込まれます。

別の側面として、プログラム・ローディングがあります。動的 CALL サブプログラムは最初に必要になったときにストレージに取り込まれるため、開始時に呼び出し元プログラムとともにストレージにロードされるわけではありません。プログラム・ロード処理にはオーバーヘッドが伴います。一般的に、アプリケーションの呼び出し構造が複雑な場合、サブプログラムのサイズが小さくない場合、およびすべてのサブ

プログラムがアプリケーションの特定の実行で呼び出されるわけではない場合は、動的呼び出しを使用すると効果的です。

類似する COBOL バージョンを使用してコンパイルされたプログラムの間の、CALL を使用した場合のパフォーマンス考慮事項は以下のとおりです (CALL オーバーヘッドのみを測定)。

- 静的 CALL リテラルは、動的 CALL リテラルよりも平均で 40% 高速でした。
- 静的 CALL リテラルは、動的 CALL ID よりも平均で 52% 高速でした。
- 動的 CALL リテラルは、動的 CALL ID よりも平均で 20% 高速でした。

注:

- この説明では、以下の COBOL バージョンを類似のものと見なしています。
 - COBOL 4.2 およびそれ以前のリリース
 - COBOL 5.1 およびそれ以降のリリース
- これらの測定は、CALL のオーバーヘッドのみを対象としています (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、異なる結果になる場合があります。

関連タスク

別のプログラムへの制御権移動

(Enterprise COBOL for z/OS プログラミング・ガイド)

PROGRAM-ID ステートメントまたは INITIAL コンパイラー・オプションでの IS INITIAL の使用

PROGRAM-ID ステートメントまたは INITIAL コンパイラー・オプションの IS INITIAL 節は、プログラムの呼び出し時に、そのプログラムと、それに含まれるすべてのプログラムが、初期状態または初回呼び出し時の状態になるようにすることを指定します。

すべての WORKING-STORAGE 変数の VALUE 節による初期化には、オーバーヘッドがあります。パフォーマンスへの影響は、このような変数の数とサイズによって異なります。

PROGRAM-ID ステートメントでの IS RECURSIVE の使用

PROGRAM-ID ステートメントの IS RECURSIVE 節は、前の呼び出しがまだアクティブである間に COBOL プログラムを再帰的に呼び出し可能であることを指定します。

IS RECURSIVE 節は、THREAD コンパイラー・オプションでコンパイルされたすべてのプログラムに必要です。

IS RECURSIVE を PROGRAM-ID ステートメントで使用した場合のパフォーマンス考慮事項は以下のとおりです (CALL オーバーヘッドのみを測定)。

- IS RECURSIVE を使用したあるテスト・ケース (COBOL を繰り返し呼び出す LE 準拠アセンブラー) では、IS RECURSIVE を使用しない場合よりも 15 % 低速でした。

注: このテストでは、CALL のオーバーヘッドのみを測定しました (つまり、サブプログラムは GOBACK のみを実行)。そのため、より多くの作業をサブプログラム内で行うフル・アプリケーションでは、さほど機能低下はありません。

第6章 ランタイム・パフォーマンスに影響するその他の製品関連要因

アプリケーションのパフォーマンスを向上させるには、COBOL と他の製品との相互作用について理解することが重要です。

このセクションでは、アプリケーションに関して考慮が必要ないくつかの製品関連要因について説明します。

ILC アプリケーションにおける 10 進オーバーフローへの影響

言語間通信 (ILC) アプリケーションとは、2 つ以上の高水準言語 (COBOL、PL/I、C など) および多くの場合アセンブラで構築されたアプリケーションのことです。

ILC アプリケーションは単一言語環境の領域外で実行されます。そのため、各言語のデータをロード・モジュール境界を超えてマップする方法、条件の処理方法、各言語でのデータの受け渡し方法といった、特殊な条件が生じます。

LE では ILC アプリケーションがフルサポートされていますが、これらを COBOL アプリケーションとともに使用すると、パフォーマンスに重大な影響を与える可能性があります。注目すべき領域の 1 つに、条件処理があります。

COBOL は通常、10 進オーバーフロー条件を無視するか、または 10 進命令の後で条件コードを検査することによってこれらを処理します。ただし、C や PL/I などの言語が COBOL と同じアプリケーション内に存在する場合、C と PL/I の両方で 10 進オーバーフロー・ビットがプログラム・マスクに設定されるため、これらの 10 進オーバーフロー条件は LE 条件管理によって処理されるようになります。このため、特に CICS では、COBOL プログラムで算術演算中に 10 進数オーバーフローが発生した場合に、COBOL アプリケーションのパフォーマンスに重大な影響が生じる可能性があります。

以下のプログラミング言語または COBOL 機能では、ハードウェア 10 進オーバーフローを有効にする必要があります。COBOL コンパイル単位での算術演算中は、COBOL コンパイル単位でオーバーフローが発生した場合に実行を再開するために、言語環境プログラムと COBOL ランタイムの条件管理が連携して機能します。このため、特に CICS では、COBOL プログラムで算術演算中に多数の 10 進数オーバーフローが発生した場合に、COBOL アプリケーションのパフォーマンスに重大な影響が生じる可能性があります。

- C/C++ および PL/I との言語間通信 (ILC)
- DLL 呼び出し
- XML GENERATE および XML PARSE
- JSON GENERATE
- BPXWDYN 動的割り振り

注記：

- これらの言語または機能は、10 進オーバーフローに影響を与えるために使用する必要はありません。これらの言語または機能が実行可能コード内に存在するだけで、動的呼び出しなどによる実行可能ファイルのロード時に 10 進オーバーフローが有効になります。
- C/C++ または PL/I プログラムを呼び出す必要はありません。ロード・モジュール内に C/C++ または PL/I が存在するだけで、10 進オーバーフロー条件ではこの低下の原因となります。
- DLL の実装では、C/C++ ランタイムを使用します。
- XML GENERATE ステートメント、XML PARSE ステートメント、または JSON GENERATE ステートメントがプログラム内に存在する場合は、C ランタイム・ライブラリーが初期化されます。そのため、C プログラムまたは PL/I プログラムをアプリケーション内で明示的に使用していなくても、プログラム・マスクに 10 進オーバーフロー・ビットが設定されます。これにより、10 進オーバーフロー条件に関しても同じ低下が生じます。JSON PARSE ステートメントには、C ランタイム・ライブラリーは必要ありません。したがって、JSON PARSE は、プログラム・マスク内の 10 進オーバーフロー・ビットには影響しません。

- 動的割り振りへの BPXWDYN インターフェースの場合は、代わりに代替エントリー・ポイント BPXWDY2 を使用してください。BPXWDY2 はプログラム・マスクを保存します。

10 進数オーバーフロー状態を引き起こす 100,000 個の算術ステートメントで COMP-3 (PACKED-DECIMAL) データ型を使用する COBOL プログラムは、C または PL/I との ILC である場合に、COBOL 単独の非 ILC である場合よりも 99.98% 以上低速でした。

z15 では、COBOL プログラムが命令ごとにオーバーフローを抑止する方法が導入されました。これにより、プログラム・マスクの 10 進オーバーフロー・ビットが設定されている場合でも、ハードウェアはオーバーフロー条件を無視できるようになります。一部の計算がオーバーフローした ILC ベンチマークでは、ARCH(13) でコンパイルすると、ARCH(12) を使用した場合よりもパフォーマンスが 27% 向上しました。

関連参照

10 進オーバーフローのパフォーマンス (*Enterprise COBOL for z/OS 移行ガイド*)

最初プログラムが LE に非準拠

アプリケーションの最初のプログラムが LE に準拠しておらず、このプログラムが繰り返し COBOL を呼び出している場合は、COBOL メインプログラムが起動される度に COBOL 環境を初期化して終了しなければならないため、大幅な性能低下が発生するおそれがあります。

このオーバーヘッドは、以下のいずれかを行うことで削減できます (改善性の高い順にリストされています)。

- アプリケーションの最初のプログラムで CEEENTRY マクロと CEETERM マクロを使用して、LE 準拠プログラムにする。
- COBOL スタブ・プログラム (オリジナルの最初のプログラムに対する CALL ステートメントだけを持つプログラム) からアプリケーションの最初のプログラムを呼び出す。
- アプリケーションの最初のプログラムから CEEPIPI サブを呼び出して LE 環境を初期化し、COBOL プログラムを起動して、アプリケーション完了時に LE 環境を終了する。
- ランタイム・オプション RTEREUS を使用してランタイム環境を再利用できるように初期化し、すべての COBOL メインプログラムをサブプログラムにする。
- ライブラリー・ルーチン保存 (LRR) 機能 (VS COBOL II の LIBKEEP ランタイム・オプションで提供される機能に似ています) を使用する。
- アプリケーションの最初のプログラムから CEEPIPI メインを呼び出して LE 環境を初期化し、COBOL プログラムを起動して、アプリケーション完了時に LE 環境を終了する。
- LE ライブラリー・ルーチンを LPA または ELPA に置く。LPA または EPLA に置くルーチンのリストはリリースによって異なります。これらのルーチンは、[IMS プリロード・リストの考慮事項](#)にリストされているものと同じです。

関連参照

アセンブラーに関する考慮事項 (*z/OS Language Environment プログラミング・ガイド*)

CICS

Language Environment は、VS COBOL II よりも多くのトランザクション・ストレージを使用します。LE では実行単位レベルでストレージが管理されるため、複数の実行単位 (エンクレーブ) が使用されている場合は、これが特に顕著になります。このことは、HEAP、STACK、ANYHEAP などが、LE 下で各実行単位に割り振られることを意味します。VS COBOL II では、スタック (SRA) とヒープ・ストレージはトランザクション・レベルで管理されます。さらに、割り振りが必要な LE 制御ブロックがいくつかあります。

CICS 下で LE が使用する 16 MB 境界より下のストレージ量を最小化するには、可能な限り ALL31(ON) および STACK(,ANYWHERE) を指定して実行する必要があります。そのためには、OS/VS COBOL ではない、すべての AMODE(24) COBOL プログラムを特定する必要があります。その後で、必要なコード変更を行ってこれらを AMODE(31) にするか、実行単位の必要に応じて ALL31(OFF) および STACK(,BELOW) を指定して CEEUOPT をリンク・エディットすることができます。特定のトランザクションで使用されているストレージ量は、そのトランザクションの補助トレース・データを参照すればわかります。LE ランタイム・オプションは CICS 下で実行中の OS/VS COBOL には影響しないため、OS/VS COBOL プログラムについての懸念は不要です。また、トランザクションが TASKDATALOC(ANY) として定義され、ALL31(ON) が使用され、

プログラムが DATA(31) でコンパイルされている場合、LE は CICS 下のトランザクションに 16 MB 境界より下のストレージを使用しないため、このストレージをさらに節約できます。

GETMAIN と FREEMAIN のアクティビティー量を削減するために使用可能な 2 つの CICS SIT オプションがあります。これにより応答時間が改善されます。1 つは RUWAPOL SIT オプションです。RUWAPOL を YES に設定すると、GETMAIN と FREEMAIN のアクティビティーを削減できます。もう 1 つは AUTODST SIT オプションです。CICS Transaction Server 1.3 以降を使用している場合は、AUTODST を YES に設定して、Language Environment に CICS 領域用のストレージを自動チューニングさせることもできます。これにより、CICS 領域内の GETMAIN と FREEMAIN の要求数が減少します。さらに、AUTODST=YES の使用時には、ストレージ・チューニング・ユーザー出口 (40 ページの『ストレージ・チューニング・ユーザー出口』を参照) を使用して、この自動ストレージ・チューニングのデフォルト動作を変更することもできます。

詳しくは、「z/OS Language Environment カスタマイズ」の『CICS 環境での Language Environment の使用方法』を参照してください。

CICS 下で実行するアプリケーションには、RENT コンパイラー・オプションが必須です。さらに、CICS 変換プログラムまたはコプロセッサを通じてプログラムを実行する (プログラムに EXEC CICS コマンドが含まれる) 場合、NODYNAM コンパイラー・オプションも使用する必要があります。Enterprise COBOL の場合は、CICS Transaction Server 1.3 以降が必要です。

詳しくは、Enterprise COBOL for z/OS プログラミング・ガイド内の RENT を参照してください。

Enterprise COBOL では、CICS コマンドまたは依存関係を含む Enterprise COBOL および VS COBOL II の (RES オプションが指定された) サブプログラムに対する静的呼び出しおよび動的呼び出しがサポートされています。Enterprise COBOL 4.2 以前のリリースでは、NORES オプションが指定された VS COBOL II の呼び出しもサポートされています。静的呼び出しは CALL リテラル・ステートメントによって行われ、動的呼び出しは CALL ID ステートメントによって行われます。EXEC CICS LINK を COBOL CALL に変換することで、トランザクション応答時間が向上し、仮想記憶域の使用量が減少する可能性があります。Enterprise COBOL では、CICS 環境内の OS/VS COBOL プログラムへの呼び出し、または OS/VS COBOL プログラムからの呼び出しはサポートされていません。この場合は、EXEC CICS LINK を使用する必要があります。

注 : Language Environment 下で EXEC CICS LINK を使用すると、新しい実行単位 (エンクレーブ) が各 EXEC CICS LINK 用に作成されます。これは、LINK 先のプログラムごとに、新しい制御ブロックが割り振られ、その後に解放されることを意味します。この結果、ストレージ要求数が増加します。ストレージ管理チューニングが行われていない場合は、エンクレーブごとにより多くのストレージ要求が発生する可能性があります。EXEC CICS LINK ごとに新しいエンクレーブが作成された結果、VS COBOL II に比較して CPU 時間のパフォーマンスも低下します。アプリケーションで多数の EXEC CICS LINK が使用されている場合は、可能であれば常に COBOL CALL を使用することで、このような追加オーバーヘッドを回避できます。

COBOL CALL ステートメントを使用して、CICS 変換プログラムで変換されたプログラム、または CICS コプロセッサでコンパイルされたプログラムを呼び出す場合は、CALL ステートメントの最初の 2 つのパラメーターとして DFHEIBLK および DFHCOMMAREA を渡す必要があります。ただし、変換されていないプログラムを呼び出す場合、CALL ステートメントで DFHEIBLK および DFHCOMMAREA を渡してはなりません。さらに、呼び出し先サブプログラムで EXEC CICS 条件処理コマンドを使用していない場合は、ランタイム・オプション CBLPSHPOP(OFF) を使用することで、LE ランタイムによる呼び出しごとに行われる EXEC CICS PUSH HANDLE および EXEC CICS POP HANDLE の実行オーバーヘッドを排除できます。CBLPSHPOP 設定は、CLER トランザクションを使用して動的に変更できます。

アプリケーション内のすべての 2 進 (COMP) データ項目の使用法が PICTURE および USAGE の指定に準拠していれば、TRUNC(OPT) を使用してトランザクション応答時間を改善できます。パフォーマンス重視の CICS アプリケーションには、この方法をお勧めします。2 進データ項目の使用法が PICTURE および USAGE の指定に準拠していない場合は、TRUNC(BIN) コンパイラー・オプションを使用する代わりに、COMP-5 データ型を使用するか、PICTURE 節内の精度を高くする方法があります。CICS 変換プログラムは切り捨てを起こすようなコードを生成しないこと、また CICS コプロセッサは切り捨てを起こさない COMP-5 データ型を使用することに注意してください。OS/VS COBOL プログラムで問題なく NOTRUNC を使用していた場合は、IBM Enterprise COBOL での TRUNC(OPT) も同様に動作します。TRUNC コンパイラー・オプションの追加情報については、「Enterprise COBOL for z/OS プログラミング・ガイド」の『TRUNC』を参照してください。

Db2

アプリケーション内のすべての 2 進 (COMP) データ項目の使用法が PICTURE および USAGE の指定に準拠し、2 進データが COBOL プログラムによって作成されていれば、TRUNC(OPT) を使用して Db2® 下でのパフォーマンスを改善できます。

パフォーマンス重視の Db2 アプリケーションには、この方法をお勧めします。2 進データ項目の使用法が PICTURE および USAGE の指定に準拠していない場合は、COMP-5 データ型を使用するか、TRUNC(BIN) コンパイラー・オプションを使用する必要があります。OS/VS COBOL プログラムで NOTRUNC を問題なく使用していた場合は、COBOL for MVS & VM、COBOL for OS/390 & VM、および Enterprise COBOL での TRUNC(OPT) も同様に動作します。TRUNC オプションについて詳しくは、26 ページの『TRUNC』を参照してください。

Db2 ストアード・プロシージャーとして使用される COBOL プログラムには、RENT コンパイラー・オプションを使用する必要があります。

最良のパフォーマンスを得るためには、互換性のあるコード・ページを使用して、不要な変換を避けてください。例えば、Db2 データベースでコード・ページ 037 が使用されていて、コンパイラー・オプション CODEPAGE(1140)、SQL、SQLCCSID を使用すると、CODEPAGE(037)、SQL、SQLCCSID または CODEPAGE(1140)、SQL、NOSQLCCSID のいずれかを使用する場合よりもパフォーマンスが低下することがあります。最初のオプション・セットではコード・ページに適合させるための変換が必要ですが、2 番目と 3 番目のオプション・セットではそのような変換が不要であるためです。

DFSORT

多くのソート操作のパフォーマンスを向上させるには、FASTSORT コンパイラー・オプションを使用します。FASTSORT を使用すると、DFSORT 製品は、以下のいずれか (または両方) で指定された入力ファイルと出力ファイルのいずれか (または両方) に対して入出力を実行します: SORT ... USING または SORT ... GIVING ステートメント。ソート・ファイルに INPUT PROCEDURE 句または OUTPUT PROCEDURE 句がある場合、FASTSORT オプションは、INPUT PROCEDURE または OUTPUT PROCEDURE に影響を与えません。ただし、INPUT PROCEDURE 句と一緒に GIVING 句を指定した場合、または USING 句と一緒に OUTPUT PROCEDURE 句を指定した場合でも、FASTSORT は SORT ステートメントの USING 部分または GIVING 部分に適用されます。要件の完全なリストについては、「Enterprise COBOL for z/OS プログラミング・ガイド」を参照してください。

DFSORT を使用した場合のパフォーマンス考慮事項は以下のとおりです。

- 100,000 件のレコードを処理したあるプログラムは、FASTSORT を使用した場合、NOFASTSORT の使用時に比べて 45% 高速でした。

関連参照

FASTSORT (Enterprise COBOL for z/OS プログラミング・ガイド)

IMS

アプリケーションが IMS 下で実行されている場合は、アプリケーション・プログラムとライブラリー・ルーチンをプリロードすると、ロード/検索オーバーヘッドや入出力アクティビティーを削減するために役立ちます。

これは特にライブラリー・ルーチンに当てはまります。ライブラリー・ルーチンはあらゆる COBOL プログラムで使用されているからです。アプリケーション・プログラムがプリロードされていると、外部ストレージからプログラムを取り出す必要がないため、そのプログラムの以降の要求はより高速に処理されます。プリロードされるアプリケーションには、RENT コンパイラー・オプションが必須です。

ライブラリー・ルーチン保存 (LRR - Library Routine Retention) 機能を使用すると、IMS/TM 下で実行される COBOL トランザクションのパフォーマンスが大幅に向上する可能性があります。LRR は、VS COBOL II LIBKEEP ランタイム・オプションに類似の機能を提供します。この機能は、LE 環境を初期化状態に維持し、ロードされた LE ライブラリー・ルーチン、これらのライブラリー・ルーチンに関連するストレージ、および LE 始動制御ブロック用のストレージをメモリー内に保持します。IMS 従属領域内で LRR を使用するには、以下の手順を実行する必要があります。

1. 始動 JCL またはプロシージャで IMS 従属領域を開始するには、PREINIT=xx パラメーターを指定します。xx は、IMS PROCLIB データ・セットの DFSINTxx メンバーの 2 文字の接尾部です。
2. IMS PROCLIB データ・セットの DFSINTxx メンバーに、CEELRRIN という名前を組み込みます。
3. IMS 従属領域を開始します。

SCEESAMP データ・セットにある CEELRRIN サンプル・ソースを変更して、LRR 機能を初期化する独自のロード・モジュールを作成することもできます。これを行う場合は、上記の CEELRRIN の代わりにモジュール名を使用します。

注意: RTEREUS オプションは IMS には推奨されません。このオプションの使用は避ける必要があります。RTEREUS ランタイム・オプションを使用する場合は、すべてのアプリケーションの最上位 COBOL プログラムをプリロードしておく必要があります。

RTEREUS を使用すると、領域が停止するまで、または COBOL プログラムで STOP RUN が発行されるまで、LE 環境は稼働状態のままになります。これは、あらゆるプログラムとその WORKING-STORAGE が (最初の COBOL プログラムの初期化時から) 領域内に保持されることを意味します。この方法は非常に高速ですが、領域がすぐにいっぱいになってオーバーフローするおそれがあります。これは特に、多数の COBOL プログラムが起動されている場合に顕著です。

注: このオプションに関する制約事項および使用上の注意については、z/OS 言語環境プログラム カスタマイズを参照してください。

RTEREUS と LRR をいずれも使用しない場合は、以下のライブラリー・モジュールをプリロードすることをお勧めします。

- すべての COBOL アプリケーションの場合: CEEBINIT、IGZCPAC、IGZCPCO、CEEEV005、CEEPLPKA、IGZETRM、IGZEINI、IGZCLNK、CEEEV004、IGZDXMR、IGZXD24、IGZXLPIO、IGZXLPKA、IGZXLPKB、IGZXLPKC
 - アプリケーションに VS COBOL II プログラムも含まれている場合: IGZCTCO、IGZEPLF、および IGZEPCL
- プリロードにより、トランザクションごとにこれらのモジュールをロードおよび削除することに関連した入出力アクティビティー量が削減されます。

上記の COBOL ライブラリー・モジュール以外に、必要な 16 MB 境界より下のルーチンもプリロードする必要があります。16 MB 境界より下のルーチンのリストは、「Language Environment カスタマイズ」マニュアルにあります。

さらに、使用頻度の高いアプリケーション・プログラムは、RENT コンパイラー・オプションでコンパイルし、プリロードすると、ロードに関連した入出力アクティビティー量を削減できます。

以下の条件を満たしている場合は、TRUNC(OPT) コンパイラー・オプションを使用できます。

- COBOL 以外のプログラムによって作成されたデータベースを使用していない。
- すべての 2 進データ項目の使用が、データ項目の PICTURE 指定と USAGE 指定に準拠している (例えば、2 進データ型を使用するポインター演算がない)。

それ以外の場合は、TRUNC(BIN) コンパイラー・オプションまたは COMP-5 データ型を使用する必要があります。TRUNC コンパイラー・オプションについて詳しくは、[26 ページの『TRUNC』](#)を参照してください。

関連参照

言語環境プログラム ライブラリー・ルーチン保存機能 (LRR)

(z/OS Language Environment プログラミング・ガイド)

IMS での言語環境プログラムの使用

(z/OS Language Environment カスタマイズ)

言語環境プログラム COBOL コンポーネント・モジュール

(z/OS Language Environment カスタマイズ)

LLA

RMODE 24 属性を持つ CSECT にリンクしている Enterprise COBOL プログラム (COBOL 5 以降のリリース) は、Library Lookaside (LLA) による管理から除外できます。

LLA 機能を使用する際の考慮事項

下記のプログラムには、RMODE 24 属性を持つ CSECT が含まれています。

- RMODE(24) または NORENT のいずれかのコンパイラー・オプションでコンパイルされた Enterprise COBOL プログラム。
- NORENT コンパイラー・オプションでコンパイルされた VS COBOL II プログラム。
- RMODE 24 の CSECT を含むアセンブラー・プログラム。

デフォルトでは、Enterprise COBOL 5 以降のプログラムの RMODE 属性は RMODE ANY です。このようなプログラムが上記のいずれかにリンクされると、バインド・プログラムは RMODE 24 の CSECT をあるセグメントに置き、Enterprise COBOL 5 コードを 2 番目のセグメントに置きます。C-WSA クラス (COBOL 5 から導入) 用の 3 番目のセグメントもあります。2 つより多くのセグメントを持つプログラム・オブジェクトは、Library Lookaside (LLA) 機能で使用できません。この問題は、Enterprise COBOL 5 プログラムに RMODE(24) コンパイラー・オプションを明示的に指定し、DYNAM=NO バインダー・オプション (バインダーのデフォルト) を指定することによって回避できます。これにより、RMODE 属性がプログラム・オブジェクト内で一貫性のあるものになります。あるいは、RMODE(24) オプションと NORENT オプションを使用せず、アセンブラー・プログラムで RMODE 24 の CSECT を使用しないことで、上記の COBOL プログラムのコンパイルを変更することもできます。

第 7 章 COBOL 6 を最大限に活用するためのコーディング技法

このセクションでは、プログラムをチューニングしてパフォーマンスを向上させるためのソース・コードの変更方法に焦点を当てます。コーディング・スタイルは、データ型とともに、アプリケーションのパフォーマンスに大きく影響する可能性があります。

BINARY (COMP または COMP-4)

BINARY データと同義語 COMP および COMP-4 は、COBOL における 2 の補数のデータ表現です。

BINARY データは、内部または外部 10 進数データと同様に PICTURE 節を使用して宣言されますが、基礎となるデータ表現は、ハーフワード (2 バイト)、フルワード (4 バイト)、またはダブルワード (8 バイト) です。

コンパイラー・オプション TRUNC(OPT | STD | BIN) は、宣言された PICTURE 節に合わせてコンパイラーが値を修正するかどうか、その方法、およびデータ項目へのアクセス時に存在する有効データ量を決定します。

COBOL 4 以前のバージョンにおける BINARY データおよび TRUNC オプションのパフォーマンスに関する一般的な考慮事項は COBOL 6 にも引き続き該当しますが、各種 TRUNC サブオプションによるパフォーマンスの相対的差異は一部変化しています (劇的な変換の場合もあります)。このような変化は、コーディングおよびコンパイラー・オプションの選択に影響することがあります。

パフォーマンスの相対的および絶対的な差異を定量化するために、2 進データ項目に対する一連の加算演算が z15 マシン上のループ内で実行されました。以下の 4 種類のテストには同タイプで同数の算術演算が含まれていますが、桁数が異なります。オペランドはすべて符号付きです。

- テスト 1: 8 つの加算 (1 桁から 8 桁が 1 つずつ)
- テスト 2: 8 つの加算 (それぞれが 9 桁)
- テスト 3: 8 つの加算 (10 桁から 17 桁が 1 つずつ)
- テスト 4: 8 つの加算 (それぞれが 18 桁)

次に、これらのテストを TRUNC オプションを変えてコンパイルしました。

最初の実験では、TRUNC(STD) コンパイラー・オプションを指定しています。

TRUNC(STD) はコンパイラーに、常に指定された PICTURE 節に合わせて修正するように指示します。コンパイラーはロードされた値に、PICTURE 節で指定された桁数のみが含まれていると想定することが可能になります。

表 5. TRUNC(STD) 指定時の 4 つのテスト・ケースのパフォーマンス差の結果			
TRUNC(STD)	COBOL 4 vs TEST 1	COBOL 6 vs TEST 1	COBOL 6 vs COBOL 4
テスト 1: 1 桁から 8 桁	100%	100%	17.2%
テスト 2: 9 桁	153.1%	99.7%	11.2%
テスト 3: 10 桁から 17 桁	484.5%	153.5%	5.4%
テスト 4: 18 桁	759.6%	99.7%	2.3%

これらの結果は、以下を実証しています。

- TRUNC(STD) を使用した場合、COBOL 6 はすべての長さで COBOL 4 よりパフォーマンスが優れている。
- COBOL 4 と COBOL 6 ではどちらも、桁数の増加に伴ってパフォーマンスが低下する。ただし、COBOL 6 の使用時には COBOL 4 に比べて低下が緩やかであり、低下の全体量も小さい。

2 番目の実験では、TRUNC(BIN) コンパイラー・オプションを指定しています。このオプションを指定することは、すべての BINARY データに COMP-5 型を使用することと同等です。

TRUNC(BIN) はコンパイラーに、指定された PICTURE 節に合わせて値を修正するのではなく、基礎データ表現 (2 バイト、4 バイト、または 8 バイト) に値を修正し戻すことのみが可能であることを指示します。また、このオプションはコンパイラーに、ロードされた値が 2 バイト、4 バイト、または 8 バイト相当の有効データを含む可能性があることを想定するよう要求します。

表 6. TRUNC(BIN) 指定時の 4 つのテスト・ケースのパフォーマンス差の結果			
TRUNC(BIN)	COBOL 4 vs TEST 1	COBOL 6 vs TEST 1	COBOL 6 vs COBOL 4
テスト 1: 1 桁から 8 桁	100%	100%	36.2%
テスト 2: 9 桁	165.8%	100%	21.8%
テスト 3: 10 桁から 17 桁	3889.4%	2363.8%	22.0%
テスト 4: 18 桁	3889.2%	2363.7%	22.0%

これらの結果は、以下を実証しています。

- TRUNC(BIN) を使用した場合、COBOL 6 はやはりすべての長さで COBOL 4 よりパフォーマンスが優れている。
- 9 桁でのテストの場合、COBOL 6 は低速化を示さない。
- 長さが 9 桁を超える場合、COBOL 4 と COBOL 6 の両方でパフォーマンスが劇的に低下する (ただし、絶対量では COBOL 4 がより顕著)。これは、全整数ハーフ/フル/ダブルワードまでのデータが入力データに含まれる可能性がある (さらに、追加のデータ型変換とライブラリー・ルーチンが必要になる) という TRUNC(BIN) の要件によるものです。

3 番目の実験および最後の実験では、TRUNC(OPT) コンパイラー・オプションを指定しています。TRUNC(OPT) はパフォーマンス・オプションです。コンパイラーは、入力データが PICTURE 節に準拠しているものと想定し、その後、以下のうち最適な方法でデータを自由に操作できます。

- TRUNC(STD) の場合と同様に、PICTURE 節に合わせて修正するか、または
- TRUNC(BIN) の場合と同様に、2 バイト境界、4 バイト境界、または 8 バイト境界に合わせて修正する

表 7. TRUNC(OPT) 指定時の 4 つのテスト・ケースのパフォーマンス差の結果			
TRUNC(OPT)	COBOL 4 vs TEST 1	COBOL 6 vs TEST 1	COBOL 6 vs COBOL 4
テスト 1: 1 桁から 8 桁	100%	100%	87.3%
テスト 2: 9 桁	400.3%	100.9%	22.0%
テスト 3: 10 桁から 17 桁	235.3%	100.6%	37.3%
テスト 4: 18 桁	6099.0%	156.9%	2.3%

これらの結果は、以下を実証しています。

- TRUNC(OPT) を使用した場合、COBOL 6 はやはり COBOL 4 よりパフォーマンスが優れている。
- COBOL 6 は 18 桁までは低速化を示さないが、この最長の長さにおいても、COBOL 4 より大幅にパフォーマンスが優れている。

注: TRUNC(OPT) は、2 進数領域に移動されるデータが PICTURE 節に準拠していることがわかっている場合にのみ使用してください。そうしないと、予測不能な結果が生じるおそれがあります。詳しくは、「Enterprise COBOL for z/OS プログラミング・ガイド」の『TRUNC』を参照してください。

すべての TRUNC オプションとデータ項目長にわたって、上述のように COBOL 6 は COBOL 4 よりパフォーマンスが優れています。このような向上は、以下の理由によるものです。

- 64 ビット「G」形式の命令の使用により、8 桁を超える事例ではるかに効率的なコードが可能

- 非常に大きい TRUNC(BIN) のための、より効率的なライブラリー・ルーチン

『5 ページの『第 2 章 COBOL 6 への移行のためのアプリケーションの優先順位付け』』では、2 進数ダブルワード算術演算 (大きい 2 進数算術演算) の具体例を紹介し、この種の演算におけるパフォーマンス向上を COBOL 4 のコンパイラーと比較して示しています。この例で、COBOL 6 は COBOL 4 以前のコンパイラー・リリースに比べて大幅に高速化しています。

各種 TRUNC オプション間のパフォーマンスの相対的差異は、COBOL 4 に比べて少なくなっています。これは主に、コンパイラーがオーバーフロー用のランタイム・テストを挿入することに起因します。オーバーフローが発生しなければ、負荷の高い「divide」ハードウェア命令が回避されます。

データが PICTURE 節に準拠していることがわかっている場合は、やはり TRUNC(OPT) が選択すべき総合的な最良オプションですが、相対的に言えば、TRUNC(STD) に対する向上は COBOL 4 の場合より小さく、また、COBOL 6 ではいずれのオプションでも総合的な絶対パフォーマンスは優れています。

TRUNC(BIN) は、COMPUTE または MOVE の結果からの格納時にはコードをより効率化することが可能ですが、これらのデータ項目が算術演算ステートメントへの入力として使用される場合、このオプションがパフォーマンスに重大な悪影響を及ぼすことには変わりはありません (コンパイラーが最大 2 バイト、4 バイト、8 バイトのサイズを想定する必要があるため)。COBOL 6 は TRUNC(STD) の修正コードを最適化するため、TRUNC(BIN) によるパフォーマンス上の利点はやや少なくなりました。

TRUNC(BIN) を使用する代わりに、選択したデータ項目にのみ COMP-5 を指定する方が良い場合があります。例えば、特に COMPUTE ステートメント内のデータ項目が COMP-5 で指定されていない場合、一般的にパフォーマンスは向上します。

DISPLAY

「IBM Enterprise COBOL 4.2 Performance Tuning」に、「USAGE DISPLAY データ項目は、計算で (特に計算で頻繁に使用される領域で) 使用しないでください」という記載があります。これは COBOL 6 でも依然ベスト・プラクティスです。ただし、オプション OPT(1 | 2) と ARCH(10 | 11) を使用すると、COBOL 6 コンパイラーは、DISPLAY オペランドを 10 進数浮動小数点 (DFP) に効率的に変換できます。ARCH(12) 以上では、OPT(1 | 2) を使用すると、COBOL 6 コンパイラーは DISPLAY オペランドを PACKED-DECIMAL に変換し、PACKED-DECIMAL 算術演算をベクトル・レジスターで実行できるようになります。どちらの最適化も、計算で DISPLAY データ項目を使用することのオーバーヘッドを削減します。

次の USAGE DISPLAY データを、

```
1 A pic s9(17).  
1 B pic s9(17).  
1 C pic s9(18).
```

次の COMP-3 と比較します。

```
1 A pic s9(17) COMP-3.  
1 B pic s9(17) COMP-3.  
1 C pic s9(18) COMP-3.
```

対象となるステートメントは次のとおりです。

```
ADD A TO B GIVING C.
```

COBOL 4 では COMP-3 を使用すると DISPLAY を使用するよりも 22% 速くなりますが、COBOL 6 では COMP-3 を使用すると DISPLAY を使用するよりも 52% 速くなります。

パフォーマンスの比較はデータのサイズや計算ステートメントの種類によって異なりますが、計算での DISPLAY データ項目のパフォーマンスは、ARCH(10 | 11) と OPT(1 | 2) を使用すると、あるいは ARCH(12) 以上を使用すると COBOL 6 では多くの場合に向上しました。COBOL 4 から COBOL 6 の間で向上はしていますが、計算で使用されるデータ項目には引き続き COMP-3 または BINARY の使用が推奨されています。特に、データ項目をループ・カウンターまたはテーブル索引として使用する場合は、そのデータ項目を DISPLAY から BINARY へ変換すると、パフォーマンスが大幅に向上する可能性があります。

PACKED-DECIMAL (COMP-3)

「IBM Enterprise COBOL 4.2 Performance Tuning」に、「PACKED-DECIMAL (COMP-3) データ項目を計算で使用する場合は、乗算および除算でのライブラリー・ルーチンの使用を避けるために、PICTURE 指定には 15 桁以下を使用してください」という記載があります。

COBOL 6 でオプション ARCH(8 | 9 | 10 | 11) と OPT(1 | 2) を使用すると、コンパイラーは、このような大きな乗算演算および除算演算の一部のインライン 10 進浮動小数点 (DFP) コードを生成できます。この最適化でサポートされる最大の中間結果サイズは 34 桁です。この DFP への変換にはいくつかのオーバーヘッドがありますが、ライブラリー・ルーチンと呼び出さなければならない場合に比べて損失は少なくなります。これは、COMPUTE ステートメント用にコンパイラーによってパック 10 進数に変換される、外部 10 進数 (DISPLAY および NATIONAL) 型にも当てはまります。

COBOL 6.2 で ARCH(12) を使用すると、コンパイラーは代わりにベクトル・パック 10 進数機能を使用します。この機能は、中間結果をメモリーではなく、ベクトル・レジスターに保管することにより、パックおよびゾーンの 10 進数計算を高速化します。これにより、DFP への変換のオーバーヘッドが回避されます。

固定小数点と浮動小数点

「IBM Enterprise COBOL 4.2 Performance Tuning」に、「大きい指数を使用して固定小数点の指数演算を行う場合は、指数演算を浮動小数点で評価することを強制するオペランドを使用することで、計算をより効率的に行うことができます」という記載があります。

COBOL 6 でも、浮動小数点の指数演算の方が固定小数点の指数演算よりもずっと高速ですが、各タイプの指数演算の相対コストが COBOL 4 と COBOL 6 では変化しています。

以下のコード例を考えてみましょう。

```
01 A PIC S9(6)V9(12) COMP-3 VALUE 0.  
01 B PIC S9V9(12) COMP-3 VALUE 1.234567891.  
01 C PIC S9(10) COMP-3 VALUE -9.  
  
COMPUTE A = (1 + B) ** C. (original)  
COMPUTE A = (1.0E0 + B) ** C. (forced to floating-point)
```

元の固定小数点の指数演算は、COBOL 6 では COBOL 4 に比べて 89% 高速です。

浮動小数点の指数演算の強制は、COBOL 6 では、COBOL 4 と比較して 68% 高速です。

ただし、浮動小数点の指数演算は、固定小数点の指数演算よりも数倍も高速であるため、可能であれば、やはり浮動小数点の指数演算を使用することをお勧めします。

一括表示表現

「IBM Enterprise COBOL 4.2 Performance Tuning」に、「演算式を評価する場合、コンパイラーは COBOL の左から右への評価規則に制約を受けます。最適化プログラムに定数計算 (コンパイル時に実行可能) または重複計算 (共通副次式) を認識させるためには、すべての定数式と重複式を式の左端に移動するか、それらを括弧内にグループ化してください。」という記載があります。

COBOL 6 コンパイラーは最適化の一環として式を因数処理するため、COBOL 4 で推奨されていたソース・コード・レベルで行われるこのタイプの因数処理は不要になりました。ただし、これは 1 つの式の中でのみ当てはまります。次の例を考えてみます。ここでは、一連の品目の総額を集計し、値引きを適用する 2 つの方法を示しています。

```
MOVE ZERO TO TOTAL  
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10  
COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT  
END-PERFORM
```

```
MOVE ZERO TO TOTAL  
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10  
COMPUTE TOTAL = TOTAL + ITEM(I)  
END-PERFORM  
COMPUTE TOTAL = TOTAL * DISCOUNT
```

コードの最初のブロックでは 10 回の乗算が実行され、2 番目のブロックでは 1 回のみ実行されています。そのため、コードの 2 番目のブロックのほうが効率的です。COBOL 6 コンパイラーは、このタイプの因数処理をループに対して実行しません。ソース・コード・レベルでのみ実行します。

シンボリック定数

「IBM Enterprise COBOL 4.2 Performance Tuning」に、「最適化プログラムが、プログラム全体でデータ項目を定数として認識するようにしたい場合は、データ項目を VALUE 節で初期化し、プログラム内のどこでもその項目を変更しないようにします」という記載があります。

これが有効かつ重要な推奨事項であることに変わりはありませんが、1 つ相違点があります。COBOL 6 コンパイラーは、データ項目が VALUE 節で指定されたものと同じ値に再初期化されることを許容します。この場合、コンパイラーは、データ項目の値がその初期値から変更されていないと認識し、引き続きこのデータ項目を定数として扱います。

Occurs Depending On テーブルのパフォーマンス・チューニング考慮事項

通常、データ項目宣言を相対的に順序付けしても、パフォーマンスに対する大幅な、または容易に予測可能な影響はありません。

ただし、Occurs Depending On (ODO) テーブルがプログラムに含まれている場合は、ODO テーブルに続くデータ項目の特定のグループ・レイアウトが、特定の他の変数へのアクセス時に大幅なパフォーマンス低下を招くおそれがあります。

ODO テーブルが以下のように宣言されているとします。

```
01 TABLE-1.  
  05 X PIC S9(4) comp.  
  05 Y OCCURS 3 TIMES  
      DEPENDING ON X PIC X.  
  05 Z PIC S9.
```

TABLE-1 の項目 Y のサイズは別のデータ項目に依存しているため、同じレベル 01 レコード内の後続の非従属項目は、可変位置項目 (上の例の項目 Z など) です。

可変位置項目 Z へのロードまたは格納を行うには、X の現行値に基づいて Z の位置を判別するための追加コードをコンパイラーで生成する必要があります。ODO テーブルがネストされている場合は、複数の計算がさらに必要です。

ただし、常に ODO テーブルの後でレコードを終了することによって、テーブルの後で宣言される変数はすべて可変位置変数ではなくなるため、これらの変数へのアクセスはより効率的になります。次の例では、新しいレベル 01 のレコードが、ODO テーブルの後に、また他の変数が宣言される前に追加されています。

```
01 TABLE-1.  
  05 X PIC S9(4) comp.  
  05 Y OCCURS 3 TIMES  
      DEPENDING ON X PIC X.  
01 WS-VARS.  
  05 Z PIC S9.
```

01 レベル・レコードの固定サイズ・テーブルの後に置かれたデータ項目に対して算術演算を実行するベンチマーク・プログラムでは、テーブルに OCCURS DEPENDING ON 節を指定した場合よりも 91% パフォーマンスが向上しました。

PERFORM の使用

Enterprise COBOL では、インライン PERFORM またはライン外 PERFORM という、2 つの基本的な方法で PERFORM 動詞を使用することができます。

パフォーマンスの観点からは、すべての最適化レベルで制御フローが単純であるため、インライン PERFORM が推奨されます。また、COBOL 6 プログラム・オブジェクトでは、Debug Tool はライン外

PERFORM の内容をスキップすることができます。ただし、単にインライン PERFORM を組み込むために、大きな、または複雑なコード・シーケンスを複製することは、一般的には望ましくありません。

一般的に、ライン外 PERFORM の実行コードには以下のステップが含まれます。

1. PERFORM の完了時に制御を戻すプログラム・アドレスを設定し、そのアドレスをコンパイラ生成 LOCAL-STORAGE データ項目に保管します。
2. PERFORM 対象の範囲の先頭に分岐します。
3. PERFORM 対象の範囲を実行します。
4. 最初のステップで言及したコンパイラ生成データ項目を介し、間接的に分岐して戻ります。

また、反復回数の指定や条件のテストなどを行うための句に関連付けられたロジックも実行されます。

OPT(0) より上の最適化レベルでは、コンパイラは、一部のライン外分岐コードの削除を試みます。これを行うために、コンパイラは、必要に応じてコード・シーケンスを複製します。この複製は、PERFORM 範囲の最大サイズおよびプログラム全体の最大合計サイズに制限されています。これらの最大値を制御するための構成オプションはありません。

この「PERFORM インライン化」による最適化は、PERFORM ステートメントごとに行うことができます。ただし、候補とするためには、PERFORM 対象の範囲の性質に、ある特性が含まれていなければなりません。

基本的に、最適化の絶好の機会を得るためには、ライン外 PERFORM ステートメントがプロシージャ呼び出しに類似している必要があります。そして当然ながら、このことは、実行対象範囲がプロシージャに類似している必要があることを暗に示しています。

通常、プロシージャには単一のエントリー・ポイントが含まれ、制御は必ず最終的には呼び出し元に戻ります。したがって、実行対象範囲では、すべての分岐(その範囲自体に含まれるコードが実行する可能性のある追加 PERFORM ステートメントを除く)が範囲内にとどまっている必要があります。同様に、プログラムには、実行対象範囲外から範囲内のステートメントへの分岐(範囲の PERFORM 以外)が含まれていてはなりません。例えば、順次文 A、B、および C がプログラムに順番に含まれているとすると、コンパイラは以下の PERFORM を最適化しません。これは、2 番目の PERFORM が実質的には最初の PERFORM の途中に分岐するためです。

```
PERFORM A THROUGH C  
PERFORM B THROUGH C
```

実行対象範囲のオーバーラップは一般に、PERFORM インライン化による最適化だけでなく、より単純な制御フロー構成で最も効果的な傾向のある、その他のグローバル最適化も妨げるおそれがあります。文 A、B、および C に加えて、(C の直後に) 文 D もプログラムに含まれているとします。以下のステートメントによって結果的に、実行対象範囲のオーバーラップが生じます。

```
PERFORM A THROUGH C  
PERFORM B THROUGH D
```

一般的に、メインプログラム内のコードと宣言セクション内のコード間の分岐(COBOL プログラムの自然なフローの一部として発生する分岐を除く)は、すべて最適化の妨げとなります。そして、このことは PERFORM ステートメントの形式での分岐にも確実に当てはまります。

必要なロジックを最も自然に表す COBOL コードを記述する必要があります。ただし、特にユーティリティー・ルーチンでは、同じことをさまざまな方法で行うことができます場合があります。例えば、次のコードは 1 つの方法でロジックを表しています。

```
LOCAL-STORAGE SECTION.  
01 ACTION PIC 9.  
  
PROCEDURE DIVISION.  
  
    MOVE 1 TO ACTION  
    PERFORM A  
  
    MOVE 2 TO ACTION  
    PERFORM A  
  
    MOVE 1 TO ACTION  
    PERFORM A
```



```
A.      IF ACTION = 1 DISPLAY "X" ELSE DISPLAY "Y".
```

このような事例では、以下のように実行対象範囲を特殊化すると良い場合があります。

```
PERFORM A1  
PERFORM A2  
PERFORM A1  
  
A1. DISPLAY "X".  
A2. DISPLAY "Y".
```

この特定の事例では、最適化プログラムによって同じ効果が得られます。最適化プログラムは、まず最初に、各 PERFORM ステートメントで、A にあるステートメントを複製します。最適化プログラムは次に、各 PERFORM ステートメントでコンパイル・リソースを使用して、コンテキストごとに、ACTION = 1 であるかどうかを明確に識別できるようにする必要があります。さらに、類似するコード・パターンでは、最適化プログラムが推測できない有用性範囲の使用について、プログラマーがある程度把握している場合もあります。

QSAM ファイルの使用

QSAM ファイルを使用する際には、BLOCK CONTAINS 節をファイル定義で使用して、可能な限り大きなブロック・サイズを使用してください (COBOL のデフォルトでは非ブロック化ファイルを使用)。

BLOCK CONTAINS 0 節を作成する新規ファイルすべてに指定し、これらのファイルの JCL で BLKSIZE パラメーターを省略すると、ユーザーに代わってシステムに最適なブロック・サイズを決定させることができます。ファイルの BLOCK CONTAINS 節を省略し、BLOCK0 コンパイラー・オプションを使用して、同じ効果を得ることもできます。これにより、ファイル処理時間が (CPU 時間と経過時間の両方で) 大幅に改善される可能性があります。

JCL で BLOCK CONTAINS 節および BLKSIZE を使用せずに 14,000 レコードを読み取り、28,000 レコードを書き込むプログラムに入出力バッファを使用した場合の、パフォーマンス考慮事項は以下のとおりです。

- BLOCK0 を使用した場合は、NOBLOCK0 よりも 90% 高速で、EXCP の使用が 98% 減少しました。

また、負荷の大きい入出力ジョブの入出力バッファ数を増加すると、ストレージ使用量は増加しますが、CPU 時間と経過時間の両方のパフォーマンスを向上させることができます。これを行うには、DCB パラメーターの BUFNO サブパラメーターを JCL で使用するか、SELECT ステートメントの RESERVE 節を FILE-CONTROL 段落で使用します。BUFNO サブパラメーターまたは RESERVE 節をいずれも使用しない場合は、システム・デフォルトが使用されることに注意してください。

ブロック化を使用せずに 14,000 レコードを読み取り、28,000 レコードを書き込むプログラムに入出力バッファを使用した場合のパフォーマンス考慮事項は以下のとおりです。

- DCB=BUFNO=1 を使用した場合は 0.452 CPU 秒かかりました
- DCB=BUFNO=5 を使用した場合は 0.129 CPU 秒かかりました
- DCB=BUFNO=10 を使用した場合は 0.089 CPU 秒かかりました
- DCB=BUFNO=25 を使用した場合は 0.067 CPU 秒かかりました

QSAM バッファの場所の説明については、第 4.1 章を参照してください。

可変長ファイルの使用

可変長ブロック化順次ファイルに書き込む場合、APPLY WRITE-ONLY 節をファイルに使用するか、AWO コンパイラー・オプションを使用します。これにより、入出力を処理するためにデータ管理サービス呼び出す回数が減少します。APPLY-WRITE-ONLY 節または AWO コンパイラー・オプションを使用した場合のパフォーマンス考慮事項については、[16 ページの『AWO』](#)を参照してください。

HFS ファイルの使用

QSAM を使用して PATH=fully-qualified-pathname オプションと FILEDATA=BINARY オプションを DD ステートメントで指定するか、環境変数を使用してファイルを定義することにより、バイト・ストリーム HFS ファイルを ORGANIZATIONAL SEQUENTIAL ファイルとして処理することができます。

QSAM を使用して PATH=fully-qualified-pathname と FILEDATA=TEXT を DD ステートメントで指定することにより、テキスト HFS ファイルを ORGANIZATION SEQUENTIAL ファイルとして処理するか、PATH=fully-qualified-pathname を DD ステートメントで指定することにより、テキスト HFS ファイルを ORGANIZATION LINE SEQUENTIAL として処理することができます。

VSAM ファイルの使用

VSAM ファイルの使用時には、順次アクセスの場合はデータ・バッファ数 (BUFND) を、ランダム・アクセスの場合は索引バッファ数 (BUFNI) を増加してください。

また、アプリケーションに適した制御インターバル・サイズ (CISZ) を選択してください。CISZ が小さいと、ランダム処理での検索は高速化しますが、挿入が犠牲になります。逆に、CISZ が大きいと、順次処理での効率が向上します。一般的に、CI およびバッファ・スペースを大きくした VSAM パラメーターを使用すると、アプリケーションのパフォーマンス向上に役立つ場合があります。

一般的に、順次アクセスが最も効率的であり、動的アクセスが次に効率的で、ランダム・アクセスが最も非効率的です。ただし、相対レコード VSAM (ORGANIZATION IS RELATIVE) の場合、各レコードをランダム順で読み取る際に ACCESS IS DYNAMIC を使用すると、ACCESS IS RANDOM を使用した場合よりも低速になる可能性があります。これは、VSAM では、ACCESS IS DYNAMIC を使用すると、複数トラックのデータをプリフェッチすることがあるためです。1つのレコードをランダム順に読み取り、次に複数の後続レコードを順次に読み取る場合は、ACCESS IS DYNAMIC が最適です。

ランダム・アクセスでは、VSAM が要求ごとに索引にアクセスしなければならないため、入出力アクティビティが増加します。INDEXED ファイルに対する順次操作に SEQUENTIAL、RANDOM、DYNAMIC の各アクセスを使用した場合の違いを説明するために、ORGANIZATION IS INDEXED ファイルをテスト・システムで使用する COBOL プログラムの実行から得られた測定値を示します。これは、ご使用のシステムで得られる典型的な結果ではない可能性があります。COBOL プログラムで、10,000 回の書き込みと 10,000 回の読み取りを行います。ACCESS IS SEQUENTIAL をベースライン 100% として使用した場合の、CPU 時間、経過時間、および EXCP カウントの比率を示します。

表 8. 異なるアクセス・モードでの CPU 時間、経過時間、および EXCP カウント			
アクセス・モード	CPU 時間 (秒)	経過時間 (秒)	EXCP 数
ACCESS IS SEQUENTIAL	100%	100%	100%
READ NEXT での ACCESS IS DYNAMIC	134%	143%	193%
READ での ACCESS IS DYNAMIC	713%	1095%	7189%
ACCESS IS RANDOM	1405%	3140%	15190%

注：READ での DYNAMIC、および RANDOM の各事例では、次の順次レコードのレコード・キーが READ に先立ってデータ・バッファに移動されました。

代替索引を使用する場合は、AIXBLD ランタイム・オプションを使用するよりも、アクセス方式サービスを使用して代替索引を作成する方が効率的です。可能であれば、複数の代替索引を使用することは避けてください。更新を適用するために基本パスを、反映するために複数の代替パスを使用しなければならないためです。

VSAM バッファの場所については、第 4.1 章を参照してください。

VSAM パフォーマンスを向上させるために、可能であればシステム管理バッファリング (SMB) を使用できます。SMB を使用するには、データ・セットは、システム管理サブシステム (SMS) ストレージを使用し、

拡張フォーマット (データ・クラスの DSNTYPE=xxx。xxx は何らかの形式の拡張フォーマット) であることが必要です。これで、必要なレコード・アクセス・タイプに応じて、以下のいずれかを使用できます。

1. AMP='ACCBias=DO': ランダム・レコード・アクセスに対してのみ最適化
2. AMP='ACCBias=SO': 順次レコード・アクセスに対してのみ最適化
3. AMP='ACCBias=DW': 主としてランダム・レコード・アクセスに対して、また一部の順次アクセスに対して最適化
4. AMP='ACCBias=SW': 主として順次レコード・アクセスに対して、また一部のランダム・アクセスに対して最適化

その他のコーディング技法およびベスト・プラクティスについては、「Enterprise COBOL for z/OS プログラミング・ガイド」の『プログラムのチューニング』を参照してください。

第 8 章 プログラム・オブジェクト・サイズと PDSE 要件

COBOL 4 と COBOL 6 間のロード・モジュール・サイズの変更

COBOL 4 と COBOL 6 の間で実行可能コードが大きくなった最も重要な理由は、COBOL 6 では、(COBOL 5 で導入された) より高度な最適化によって、呼び出し側サイトへの一部のライン外 PERFORM ステートメントがインライン化されたことです。このインライン化には、プログラム・パフォーマンス上の数多くの利点があります。まず、ライン外 PERFORM のディスパッチとリターンに伴うオーバーヘッドが回避されます。次に、実行中のステートメントを公開することで、周囲のステートメントのコンテキストでさらに最適化が行われます。この後者の理由で、最適化プログラムは、OPT(1) であっても、協同部分を活用し、冗長性を排除して、パフォーマンスを向上できる場合がよくあります。パフォーマンス向上の可能性が低い事例については、COBOL 5.1 以降、インライン化されている PERFORM の数を減らすためのチューニングが行われています。

COBOL 6 では、ソース・プログラム内の PERFORM ステートメントによって参照されるプロシーチャー (段落またはセクション) をインライン化するかどうかを制御する、INLINE オプションおよび NOINLINE オプションが導入されました。詳しくは、「Enterprise COBOL for z/OS プログラミング・ガイド」の『*INLINE*』を参照してください。

COBOL 4 と比較して実行可能サイズが大きくなる場合には、他に以下のような理由があります。

- 多くの低位 ARCH 命令での 4 バイトに対し、通常は 6 バイトである、より高位の ARCH 命令を使用しています。例:
 - 1 つのメモリー内移動の代わりに複数の ARCH(8) Move Immediate 命令を使用
 - パック/ゾーン 10 進数の算術演算に 10 進浮動小数点を活用
- COBOL 4 以上に多様な COBOL 6 最適化の結果、生成されるコードは増加しますが、パスの長さは短くなり、パフォーマンスが向上します。例:
 - より高度な INSPECT インライン化
 - 一部の複雑な変換の条件付きインライン化
 - 2 進データの 10 進数精度の条件付き修正
 - 数字編集データ項目および英数字編集データ項目に対する MOVE の高速化
- COBOL 4 では 4 バイト・ロードによってアクセスされる「ベース・ロケーター」ポインターが使用されていたのに対し、COBOL 6 ではベース・ロケーターの使用が減り、代わりに 6 バイト長の変位命令が使用されています。
- COBOL 6 では COBOL 4 に比べて、より負荷の高い MVCL 命令を避けて複数の MVC を大きなコピーに使用することを決定する際のアンロールしきい値が高くなっています。

ディスク上のプログラムのオブジェクト・サイズに対する SMARTBIN オプションの影響

Enterprise COBOL 6.4 以降、SMARTBIN が有効な場合、LP(32) オプションはデフォルトでオンになります。SMARTBIN を使用して、IBM Automatic Binary Optimizer (ABO) for z/OS 2.2 による最適化を可能にする追加のバイナリー・メタデータが入ったモジュールを生成するようにコンパイラーに指示することができます。

SMARTBIN が有効な場合、追加のバイナリー・メタデータが、モジュールの NOLOAD セグメントに配置されます。メタデータを生成するために、コンパイル時間が、ハードウェア圧縮をオンにした IBM z15 マシンの場合は、最大で 21% (OPT(0)) および 2 から 3% (OPT(1) および OPT(2)) 増加する可能性があります。また、ハードウェア圧縮をオフにした場合は、最大で 33% (OPT(0)) および 10% (OPT(1) および OPT(2)) 増加する可能性があります。

追加のメタデータによって、ディスク上のモジュールのサイズも増加するので、必要なロード・ライブラリーも大きくなります。ただし、プログラムの実行時は、ロードされないでメモリー・サイズが増加することはありません。

サイズ比較は、弊社の一連のパフォーマンス検証アプリケーションにおける、さまざまな COBOL テストから収集されたものです。

表 9. SMARTBIN と比較した場合の NOSMARTBIN のサイズ縮小率	
最適化レベル	SMARTBIN と比較した場合の NOSMARTBIN のサイズ縮小率
OPT(0)	37.1%
OPT(1)	33.2%
OPT(2)	62%

このオプションを NOSMARTBIN に変更することはできますが、追加のバイナリー・メタデータが存在しないと、Enterprise COBOL 6.4 で作成された COBOL モジュールは ABO 最適化の対象外になります。IBM Z ハードウェア改良の利点を最大限に活用するためには、将来的にモジュールを再コンパイルしてテストする必要があります。ABO を使用する場合、または将来使用する予定の場合は、SMARTBIN オプションをお勧めします。

ABO の利点について詳しくは、[IBM Automatic Binary Optimizer for z/OS 製品ページ](#)を参照してください。

関連参照

SMARTBIN

(Enterprise COBOL for z/OS プログラミング・ガイド)

プログラム・オブジェクト・サイズに対する TEST サブオプションの影響

「Enterprise COBOL for z/OS プログラミング・ガイド」の『TEST』で詳述されているように、COBOL 6 では、TEST オプションおよび NOTEST オプションにいくつかのサブオプションがあります。サブオプション SOURCE/NOSOURCE、DWARF/NODWARF および SEPARATE/NOSEPARATE は、デバッグに使用される追加の情報をプログラム・オブジェクトに組み込むかどうかを直接制御するため、オブジェクトのサイズが大幅に変わる可能性があります。

単独の TEST オプションおよびサブオプション EJPD もプログラム・オブジェクトのサイズに影響しますが、コンパイラーで実行される最適化の量とタイプがこれらのオプションによって変わる可能性があるため、サイズは大きくなる場合と小さくなる場合があります (そのため、結果として生成されるコードおよびリテラル・データ域も大きくなる場合と小さくなる場合があります)。

デバッグ情報が追加された結果、プログラム・オブジェクトのサイズは影響を受けますが、LOAD されるサイズは影響を受けないことに注意してください。

デバッグ情報は NOLOAD クラス・セグメントに含まれているため、プログラム・オブジェクトのこれらの部分は、Debug Tool、Fault Analyzer、または CEEDUMP の処理によって明示的に要求されない限り、プログラムの実行時にロードされません。そのため、COBOL 4 以前のバージョンとは異なり、デバッグ情報に関連するプログラム・オブジェクトのサイズは、LOAD 回数または実行パフォーマンスに影響しません。

オブジェクト・サイズに影響を与える方法はサブオプションごとに異なるため、各サブオプションを個別に検討してみましょう。事例ごとに OPTIMIZE(0) と OPTIMIZE(1) の結果が示され、他のオプションはすべてデフォルト設定のままになっています。

サイズ比較は、弊社の一連のパフォーマンス検証アプリケーションにおける、さまざまな COBOL テストから収集されたものです。

最初に、NOTEST のサブオプション DWARF/NODWARF を比較してみましょう。DWARF を設定すると、基本的な DWARF 診断情報がオブジェクトに組み込まれます。

表 10. <i>NOTEST(NODWARF)</i> と比較した <i>NOTEST(DWARF)</i> のサイズ増加率 (%)	
平均サイズ	NOTEST(NODWARF) と比較した NOTEST(DWARF) のサイズ増加率 (%)
OPTIMIZE(0)	35%
OPTIMIZE(1)	38%

このように、測定された両方の OPTIMIZE 設定で、すべてのデフォルト NODWARF 設定に比べ、DWARF を指定した場合は全体的なオブジェクト・サイズがおよそ 2 倍になっています。

TEST について、最初にこのオプションと NOTEST を検討してみましょう。この事例でオブジェクト・サイズが異なる理由には、以下のようなものがあります。

- サイズ増加の主な理由の 1 つめは、TEST が常に DWARF のフル・デバッグ情報をオブジェクトに組み込むことです。
- サイズ増加の主な理由の 2 つめは、TEST はデフォルトで SOURCE サブオプションを有効にするため、生成される DWARF デバッグ情報に拡張ソース・コードが組み込まれることです。
- 3 つめの理由は (一般に上記の 2 つの理由ほど重要ではありませんが)、TEST は最適化をわずかに抑制するため、プログラムの特性に応じてオブジェクト・サイズが増減する場合があることです。

表 11. <i>NOTEST</i> と比較した <i>TEST</i> のサイズ増加率 (%)	
平均サイズ	NOTEST と比較した TEST のサイズ増加率 (%)
OPTIMIZE(0)	50%
OPTIMIZE(1)	54%

次に、SOURCE/NOSOURCE サブオプションの影響を検討してみましょう。TEST(SOURCE) が指定されている場合は拡張ソースが DWARF デバッグ情報に組み込まれるため、この増加は、拡張ソース・ファイルのサイズに直接関連しています。

オブジェクト・サイズが増加するにもかかわらず SOURCE を指定する利点は、DWARF 情報に拡張ソースが組み込まれるため、IBM Debug Tool が別個にコンパイラー・リストを必要としないことです。

表 12. <i>TEST(NOSOURCE)</i> と比較した <i>TEST(SOURCE)</i> のサイズ増加率 (%)	
平均サイズ	TEST(NOSOURCE) と比較した TEST(SOURCE) のサイズ増加率 (%)
OPTIMIZE(0)	14%
OPTIMIZE(1)	15%

最後に、TEST のサブオプション EJPD/NOEJPD の切り替えがオブジェクト・サイズに与える影響を見てみましょう。このオプションは、オブジェクトに組み込まれる DWARF デバッグ情報の量やタイプを変更することはありませんが、EJPD のデバッグ要件を満たすために、コンパイラーで実行される最適化の量とタイプには影響します。

表 13. <i>TEST(NEJPD)</i> と比較した <i>TEST(EJPD)</i> のサイズ増加率 (%)	
平均サイズ	TEST(NEJPD) と比較した TEST(EJPD) のサイズ増加率 (%)
OPTIMIZE(0)	0%
OPTIMIZE(1)	0.2%

OPTIMIZE(0) は、EJPD の追加デバッグ要件による制限を受けなくてもすでに十分に低レベルであるため、この最低レベルの最適化での変化が 0% であることは理にかなっています。

OPTIMIZE(1) では、より小さく、実行が高速な実行可能コードを生成するはずだった最適化が、より制約的な EJPD 設定によって一般的に抑制されます。

COBOL 6 で実行可能ファイルに PDSE を使用するのなぜですか？

Enterprise COBOL 5 および 6 の変更で詳しく説明しているように (Enterprise COBOL for z/OS 移行ガイドを参照)、COBOL 6 実行可能ファイルは PDSE に存在する必要があり、PDS には存在できなくなります。

このセクションでは、動作がこのように変更された理由の一部を説明します。

まず、PDS に関する背景情報を示します。PDS の使用時に、お客様から複数の領域で問題が報告されています。

- 頻繁に圧縮する必要がある
- ディレクトリーの上書きによってデータが失われる
- 順次ディレクトリー検索によってパフォーマンスが影響を受ける
- メンバーがディレクトリーの先頭に追加された場合にパフォーマンスが低下する
- PDS が複数エクステントになる場合

また、PDS データ・セットでは、データ・セット全体をエンキューしないと、メンバーに対する更新アクセスを共有できません。さらに重要なことに、メンバー・スペースを再利用するために圧縮を実行したり、浪費スペース (ガスとも呼ばれます) を再利用するためにディレクトリーを再割り振りするには、PDS ライブラリーを停止しなければなりません。

いずれの場合も、実動システムでアプリケーション・ダウン時間が発生するおそれがあるため、非常に望ましくありません。

1990 年に導入された PDSE は、これらの問題を除去または少なくとも軽減するように設計されており、大部分では成功しています。PDSE は初公開時には問題をかかえており、ずっと以前のこれらの問題のため、多くのサイトが今日に至るまで PDSE を避け続けていました。

その一方で、他の多くのサイトは COBOL ロード・ライブラリーを PDSE に移動しており、これを行うプロセスは極めて機械的なものです。例:

- 新規 PDSE データ・セットを新しい名前でも割り振る
- ロード・モジュールを PDSE にコピーする (これらはプログラム・オブジェクトに変換されます)
- PDS を名前変更し、次に PDSE を名前変更する

実際のところ、Enterprise COBOL はプログラム・オブジェクトを必要としました。したがって、長いプログラム名やオブジェクト指向プログラムなどの機能のため、またプリリンカーの代わりにバインダーを使用する DLL のために、2001 年以降の実行可能ファイルには PDSE が必要です。

プログラム・オブジェクトを収容できるのは PDSE (および z/OS USS ファイル) のみです。これにより、プログラム管理バインダーは、長年にわたって存在する問題の一部を、これらのプロジェクト・オブジェクト機能を使用して解決することができます。

例えば、ロード・モジュールの 16 MB テキスト・サイズ制限に達した場合、唯一の解決方法は、サイズを小さくするために、費用がかかるプログラムの再設計またはリファクタリングを行うことでした。プログラム・オブジェクトを使用すれば、テキスト・サイズ制限は 1GB まで拡大されます。

この追加スペースは、COBOL コンパイラーが、(ランタイム・パフォーマンス向上コースの目標として) プログラム・リテラル域を拡大し、最終的にはオブジェクト・サイズを拡大できる、より高度な最適化を行うことも可能にします。プログラム・オブジェクトを使用する COBOL には、他にも利点があります。

- QY-con はプログラム・オブジェクトを必要とします。
- 条件順次 RLD サポートはプログラム・オブジェクトを必要とします (ブートストラップ起動のパフォーマンスの向上につながります)。
- パフォーマンス向上のために、プログラム・オブジェクトを使用して一度に 4K ずつページをマップできます。
- C/C++ を使用した共通の再入可能性モデルには、プログラム・オブジェクトが必要です。

- 将来の XPLINK の可能性を検討するには、プログラム・オブジェクトが必要であり、これは AMODE 64 に使用されます。

関連する問題に、SYSPLEX システム全体にわたるさまざまな共有ルールがあります。PDS ライブラリーとは異なり、PDSE データ・セットは SYSPLEX システム全体で共有することができません。したがって、既存の COBOL 6 PDS ベースより前の COBOL ロード・ライブラリーが共有されている場合は、以下のプロセスを使用して COBOL 6 PDSE ベースのロード・ライブラリーを移動します。

- 1 つの SYSPLEX をメインの PDSE ロード・ライブラリーの作成者/所有者にすることができます (開発 SYSPLEX)。
- PDSE ロード・ライブラリーの更新時に、XMIT または FTP を使用して、新しいコピーを実動 SYSPLEX システムにプッシュします。
- 他の SYSPLEX システムは続いて、更新された PDSE ロード・ライブラリーを RECEIVE します。

付録 A Automatic Binary Optimizer を使用して COBOL アプリケーションのパフォーマンスを改善する

IBM Automatic Binary Optimizer for z/OS (ABO) を使用すると、既にコンパイル済みの IBM COBOL プログラムのパフォーマンスを、再コンパイルせずに改善できます。ABO に対する入力コンパイル済みの COBOL プログラム・モジュールで、ソース・コード、ソース・コードのマイグレーション、パフォーマンス・オプションのチューニングは不要です。

引き続き最新バージョンの Enterprise COBOL を使用して新しい開発、モダナイゼーション、保守を行うことができますが、特定のコンパイル済みプログラムの再コンパイルの計画がない場合や、特定のプログラムのソース・コードを使用できない場合には、ABO を使用してそれらの COBOL モジュールのパフォーマンスを改善できます。

COBOL と ABO の間の関係について詳しくは、*IBM Automatic Binary Optimizer for z/OS ユーザーズ・ガイド* の、[ABO と Enterprise COBOL の連携](#)を参照してください。ABO について詳しくは、[ABO 製品のページ](#)を参照してください。

ABO に関するよくあるご質問 (FAQ) の一部を以下に示します。これ以外の FAQ については、[ABO 製品のページ](#)を参照してください。

ABO はコスト項目ですか？

サポートされている完全ライセンス版 ABO にはライセンス使用料が必要です。価格については、IBM 営業担当員か[オンラインの ABO 営業担当員](#)に問い合わせてください。

ABO は 90 日間のクラウド評価版か、オンプレミス評価版としても入手できます。これらの評価版は両方とも無料です。ABO クラウド評価版はインストールが不要であり、オンプレミス評価版はお客様のサイトに ABO をインストールできます。

ロード・モジュールがあるもののソース・コードがない場合、ABO の使用時にどうなりますか？

ABO に必要なのはロード・モジュールだけで、ソースは検索されません。COBOL コンパイラーにはソースが必要なので、ソースがない場合、そのコンパイラーを使用して再コンパイルすることはできません。しかし、VS COBOL II から Enterprise COBOL 4.2 までの間の COBOL コンパイラー・バージョンを使用してロード・モジュールがコンパイルされていれば、ABO は正常に機能します。

ABO の使用時に元のロード・モジュールを保管する必要がありますか？

万一来て、元のロード・モジュールをバックアップできます。

付録 B 組み込み関数の実装に関する考慮事項

COBOL 組み込み関数は、LE 呼び出し可能サービス、ライブラリー・ルーチン、インライン・コード、またはこれらの組み合わせを使用して実装されます。下表に、それぞれの組み込み関数の実装方法を示します。

表 14. 組み込み関数の実装			
関数名	LE サービス	ライブラリー・ルーチン (COBOL ランタイム)	インライン・コード
ABS			Yes
ACOS	Yes		
ANNUITY			Yes
ASIN	Yes		
ATAN	Yes		
BIT-OF		Yes	
BIT-TO-CHAR		Yes	
BYTE-LENGTH			Yes
CHAR			Yes
COMBINED-DATETIME		Yes	
CONTENT-OF			Yes
COS	Yes		
CURRENT-DATE	Yes		
DATE-OF-INTEGERS	Yes ¹	Yes ²	
DATE-TO-YYYYMMDD	Yes ¹	Yes ²	
DAY-OF-INTEGERS	Yes ¹	Yes ²	Yes
DAY-TO-YYYYMMDD	Yes ¹	Yes ²	
DISPLAY-OF		Yes	
E			Yes
EXP			Yes
EXP10			Yes
FACTORIAL			Yes
FORMATTED-CURRENT-DATE	Yes ¹	Yes ²	
FORMATTED-DATE	Yes ¹	Yes ²	
FORMATTED-DATETIME	Yes ¹	Yes ²	
FORMATTED-TIME		Yes	
HEX-OF		Yes	
HEX-TO-CHAR		Yes	
INTEGER	Yes	Yes	Yes

表 14. 組み込み関数の実装 (続き)

関数名	LE サービス	ライブラリー・ルーチン (COBOL ランタイム)	インライン・コード
INTEGER-OF-DATE	Yes ¹	Yes ²	
INTEGER-OF-DAY	Yes ¹	Yes ²	Yes
INTEGER-OF-FORMATTED-DATE	Yes ¹	Yes ²	
INTEGER-PART			Yes
LENGTH			Yes
LOG	Yes		
LOG10	Yes		
LOWER-CASE			Yes
最大			Yes
MEAN			Yes
MEDIAN		Yes	
MIDRANGE			Yes
最小			Yes
MOD			Yes
NATIONAL-OF		Yes	
NUMVAL		Yes	
NUMVAL-C		Yes	
NUMVAL-F		Yes	
ORD			Yes
ORD-MAX			Yes
ORD-MIN			Yes
PI			Yes
PRESENT-VALUE		Yes	
RANDOM	Yes		Yes
RANGE			Yes
REM (固定小数点)			Yes
REM (浮動小数点)	Yes		
REVERSE	Yes		Yes
SECONDS-FROM-FORMATTED-TIME		Yes	
SECONDS-PAST-MIDNIGHT	Yes ¹	Yes ²	
SIGN			Yes
SIN	Yes		

表 14. 組み込み関数の実装 (続き)

関数名	LE サービス	ライブラリー・ルーチン (COBOL ランタイム)	インライン・コード
SQRT	Yes		
STANDARD-DEVIATION		Yes	Yes
SUM			Yes
TAN	Yes		
TEST-DATE-YYYYMMDD		Yes	
TEST-DAY-YYYYMMDD		Yes	
TEST-FORMATTED-DATETIME		Yes	
TEST-NUMVAL		Yes	
TEST-NUMVAL-C		Yes	
TEST-NUMVAL-F		Yes	
TRIM			Yes
ULENGTH		Yes	
UPOS		Yes	
UPPER-CASE		Yes	
USUBSTR		Yes	
USUPPLEMENTARY		Yes	
UUID4		Yes	
UVALID		Yes	
UWIDTH		Yes	
VARIANCE		Yes	Yes
WHEN-COMPILED			Yes ³
YEAR-TO-YYYY	Yes ¹	Yes ²	

1. LP(32) が有効な場合

2. LP(64) が有効な場合

3. WHEN-COMPILED は、必要なときには必ず使用されるリテラルです。

付録 C Enterprise COBOL for z/OS のアクセシビリティ機能

アクセシビリティ機能は、運動や視覚などに障害を持つユーザーが情報技術製品を快適に使用できるように支援します。z/OS のアクセストビックスは、Enterprise COBOL for z/OS のアクセシビリティ機能を提供します。

アクセシビリティ機能

z/OS は、以下のような主要アクセシビリティ機能を備えています。

- スクリーン・リーダーおよび画面拡大機能ソフトウェアで一般的に使用されるインターフェース
- キーボードのみによるナビゲーション
- 色、コントラスト、フォント・サイズなど表示属性のカスタマイズ機能

z/OS では、最新の W3C 標準 [WAI-ARIA 1.0](http://www.w3.org/TR/wai-aria/) (<http://www.w3.org/TR/wai-aria/>) が、[US Section 508](https://www.access-board.gov/ict/) (<https://www.access-board.gov/ict/>) および [Web Content Accessibility Guidelines \(WCAG\) 2.0](http://www.w3.org/TR/WCAG20/) (<http://www.w3.org/TR/WCAG20/>) に準拠するように使用されています。アクセシビリティ機能を利用するには、最新リリースのスクリーン・リーダーを、この製品でサポートされる最新の Web ブラウザーと併用してください。

キーボード・ナビゲーション

ユーザーは、TSO/E または ISPF を使用して z/OS ユーザー・インターフェースにアクセスできます。

ユーザーはまた、IBM Developer for z/OS を使用して z/OS サービスにアクセスすることもできます。これらのインターフェースへのアクセスに関する情報については、以下の資料を参照してください。

- [z/OS TSO/E Primer](http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120) (<http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120>)
- [z/OS TSO/E User's Guide](http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3) (<http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3>)
- [z/OS ISPF User's Guide Volume I](http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70) (<http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70>)
- [IBM Developer for z/OS 資料](http://www.ibm.com/support/knowledgecenter/SSQ2R2/rdz_welcome.html?lang=en) (http://www.ibm.com/support/knowledgecenter/SSQ2R2/rdz_welcome.html?lang=en)

上記の資料には、キーボード・ショートカットまたはファンクション・キー (PF キー) の使用方法を含む TSO/E および ISPF の使用方法が記載されています。それぞれの資料では、PF キーのデフォルトの設定値とそれらの機能の変更方法についても説明しています。

インターフェースに関する情報

Enterprise COBOL for z/OS のオンライン製品資料は、IBM 資料で提供されており、標準的な Web ブラウザーで表示できます。

PDF ファイルでのアクセシビリティ・サポートは限定的です。PDF 資料では、オプションのフォント拡大機能およびハイコントラスト表示設定を使用でき、キーボードのみでナビゲートできます。

スクリーン・リーダーで、ピリオドやコンマなどの PICTURE 記号を含む構文図、ソース・コード例、およびテキストを正確に読み上げるには、すべての句読点を読み上げるようにスクリーン・リーダーを設定する必要があります。

支援テクノロジー製品は、z/OS のユーザー・インターフェースと連動します。特定のガイダンス情報については、z/OS インターフェースへのアクセスに使用する支援技術製品の資料を参照してください。

関連アクセシビリティ情報

標準の IBM ヘルプ・デスクとサポート Web サイトに加え、IBM は、聴覚が不自由なお客様が営業やサポート・サービスにアクセスするために使用できる TTY 電話サービスを立ち上げました。

TTY サービス
800-IBM-3383 (800-426-3383)
(北米内)

IBM およびアクセシビリティ

IBM のアクセシビリティへの取り組みについて詳しくは、[IBM Accessibility \(www.ibm.com/able\)](http://www.ibm.com/able) を参照してください。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

IBM は、本書に記載の製品、サービス、または機能を日本においては提供していない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態を提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

有理 ソフトウェアの知的財産部門
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性があります。その測定値が、一般に利用可能なシステムのものと同一である保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM はこれらの製品のテストを行っておりません。したがって IBM 以外の製品に関するパフォーマンス、互換性、またはその他のクレームの正確性は確認できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者をお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があります、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。これらのサンプル・プログラムは"現状のまま"提供されるものであり、いかなる保証も提供されません。IBM は、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年).このコードの一部は、IBM Corp. のサンプル・プログラムから派生したものです。© Copyright IBM Corp. 1993, 2020.

プライバシー・ポリシーに関する考慮事項:

Software as a Service ソリューションを含む IBM ソフトウェア製品 ("ソフトウェア・オファリング") は、製品の使用に関する情報の収集、エンド・ユーザー・エクスペリエンスの改善、エンド・ユーザーとの対話の調整、その他の目的のために Cookie または他のテクノロジーを使用できます。多くの場合、ソフトウェア・オファリングにより個人情報が収集されることはありません。一部の「ソフトウェア・オファリング」では、個人情報を収集できるようになっているものがあります。ご使用の「ソフトウェア・オファリング」が、これらの Cookie およびそれに類するテクノロジーを通じてお客様による個人情報の収集を可能にする場合、以下の具体的事項をご確認ください。

この「ソフトウェア・オファリング」は、Cookie もしくはその他のテクノロジーを使用して個人情報を収集することはありません。

この「ソフトウェア・オファリング」が Cookie およびさまざまなテクノロジーを使用してエンド・ユーザーから個人を特定できる情報を収集する機能を提供する場合、お客様は、このような情報を収集するにあたって適用される法律、ガイドライン等を遵守する必要があります。これには、エンドユーザーへの通知や同意の要求も含まれますがそれらには限られません。

これらを目的とした Cookies を含むさまざまなテクノロジーの使用について詳しくは、IBM のプライバシー・ポリシー (<https://www.ibm.com/privacy>) および IBM のオンライン・プライバシー・ステートメント (<http://www.ibm.com/privacy/details>) のセクション「"Cookies、Web Beacons、その他のテクノロジー"」、

および "IBM ソフトウェア製品と Software-as-a-Service のプライバシー・ステートメント" (<http://www.ibm.com/software/info/product-privacy>) を参照してください。

商標

IBM、IBM ロゴ、および [ibm.com](http://www.ibm.com)® は、International Business Machines Corp. の商標または登録商標であり、世界中の多くの法域で登録されています。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、www.ibm.com/legal/copytrade.html をご覧ください。

特記事項

本書に記載されているパフォーマンス考慮事項は、選択された一連のテストを使用して、特定のハードウェア/ソフトウェア構成でサンプル・プログラムを実行することで得られたものであり、実例として提供されています。

パフォーマンスは、構成、プログラム特性、その他のインストレーションおよび環境要因によって異なるため、他の稼働環境で得られる結果は異なる場合があります。お客様のワークロード向けのサンプル・プログラムを作成して、ご使用の環境に適した構成で独自の実験を行うことをお勧めします。

IBM は、本書で報告されている実験結果と同じ結果または類似の結果をユーザーの環境で実現できることを表明も保証もするものではありません。

この用語集に記載されている用語は、COBOL における意味に従って定義されています。これらの用語は、他の言語では同じ意味を持つことも、持たないこともあります。

この用語集には、以下の資料からの用語および定義が記載されています。

- 「ANSI INCITS 23-1985, *Programming languages - COBOL*」 (「ANSI INCITS 23a-1989, *Programming Languages - COBOL - Intrinsic Function Module for COBOL*」および「ANSI INCITS 23b-1993, *Programming Languages - Correction Amendment for COBOL*」で改訂)
- 「ISO 1989:1985, *Programming languages - COBOL*」は「ISO/IEC 1989/AMD1:1992, *Programming languages - COBOL: Intrinsic function module*」および「ISO/IEC 1989/AMD2:1994, *Programming languages - Correction and clarification amendment for COBOL*」に改訂されました。
- ANSI X3.172-2002, *American National Standard Dictionary for Information Systems*
- INCITS/ISO/IEC 1989-2002, *Information technology - Programming languages - COBOL*
- INCITS/ISO/IEC 1989:2014, *Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL*

米国標準規格 (ANS) の定義の前にはアスタリスク (*) を付けています。

A

簡略複合比較条件 (* abbreviated combined relation condition)

連続する一連の比較条件の中で、共通のサブジェクトまたは共通のサブジェクトと共通の比較演算子を明示的に省略したことにより生ずる複合条件。

ABEND

プログラムの異常終了。

2 GB 境界より上

いわゆる 2 GB 境界より上にあるストレージ。このストレージのアドレス指定は、AMODE 64 プログラムでのみ可能です。

16 MB 境界より上 (above the 16 MB line)

いわゆる 16 MB 境界より上かつ 2 GB 境界より下にあるストレージ。このストレージのアドレス指定は、AMODE 24 プログラムでは不可能です。1980 年代に IBM が MVS/XA アーキテクチャーを導入するまで、プログラムの仮想ストレージは 16MB に制限されていました。AMODE 24 としてリンク・エディットされたプログラムは、仮想ストレージ・ラインの下に保持されているかのように、16 MB のスペースしかアドレス指定できません。VS COBOL II 以降、プログラムに AMODE 31 を指定して、16 MB 境界より上にロードできるようになりました。

アクセス・モード (* access mode)

ファイル内のレコードを操作するに当たって使用する方式。

実際の小数点 (* actual decimal point)

10 進小数点文字のピリオド (.) またはコンマ (,) によるデータ項目における小数点位置の物理表現。

実績の文書化エンコード方式

XML 文書のエンコード・カテゴリーで、以下のいずれかとなる。XML パーサーは文書の最初の数バイトを調べて判別する。

- ASCII
- EBCDIC
- UTF-8
- UTF-16 (ビッグ・エンディアンまたはリトル・エンディアンのいずれか)
- これ以外のサポートされないエンコード
- 認識不能なエンコード

英字名 (* alphabet-name)

ENVIRONMENT DIVISION の SPECIAL-NAMES 段落内のユーザー定義語で、特定の文字セットまたは照合シーケンス、あるいはその両方に名前を割り当てる。

英字 (* alphabetic character)

英字またはスペース文字。

英数字キャラクターを置く

文字位置 (*character position*) を参照。

英字データ項目 (alphabetic data item)

記号 A のみを含む PICTURE 文字ストリングを使用して記述されるデータ項目。英字データ項目には USAGE DISPLAY がある。

英数字 (* alphanumeric character)

コンピューターの 1 バイト文字セットの任意の文字。

英数字データ項目 (alphanumeric data item)

USAGE DISPLAY として暗黙的または明示的に記述され、英数字、英数字編集、または数字編集のカテゴリを持つデータ項目への一般参照。

英数字編集データ項目 (alphanumeric-edited data item)

記号 A または X の少なくとも 1 つのインスタンスと、単純挿入記号 B、0、または / の少なくとも 1 つを含む PICTURE 文字ストリングによって記述されるデータ項目。英数字編集データ項目には USAGE DISPLAY がある。

英数字関数 (* alphanumeric function)

コンピューターの英数字セットからの 1 つ以上の文字のストリングで値が構成されている関数。

英数字グループ項目 (alphanumeric group item)

GROUP-USAGE NATIONAL 節なしで定義されるグループ項目。INSPECT、STRING、および UNSTRING などの操作の場合、英数字グループ項目は、グループの実際の内容に関係なく、そのすべての内容が USAGE DISPLAY として記述されているかのように処理される。グループ内の基本項目 (MOVE CORRESPONDING、ADD CORRESPONDING、INITIALIZE など) の処理を必要とする操作の場合、英数字グループ項目はグループ・セマンティクスを使用して処理される。

英数字リテラル (alphanumeric literal)

'、"、X'、X"、Z'、Z" の各セットからの開始区切り文字を持つリテラル。この文字ストリングには、コンピューターの有する文字セットの任意の文字を含めることができる。

代替レコード・キー (* alternate record key)

基本レコード・キー以外のキーで、その内容が索引付きファイルの中のレコードを識別するもの。

米国規格協会 (American National Standards Institute: ANSI)

米国で認定された組織が自発的工業規格を作成して維持する手順を設定する組織であり、製造業者、消費者、および一般の利害関係者で構成される。

ARGUMENT

(1) ID、リテラル、算術式、または関数 ID で、これにより関数の評価に使用する値を指定する。(2) 呼び出されたプログラムまたは呼び出されたメソッドに値を渡すために使用される、CALL または INVOKE ステートメントの USING 句のオペランド。

算術式 (* arithmetic expression)

数値リテラル、数値基本項目 (算術演算子で区切られた ID とリテラルなど) を表す ID、1 つの算術演算子で区切られた 2 つの算術式、または括弧で囲まれた算術式。

算術演算 (* arithmetic operation)

ある算術ステートメントが実行されることにより、またはある算術式が計算されることにより生じるプロセスで、そこで指定された引数に対して数学的に正しい解が求められる。

算術演算子 (* arithmetic operator)

単一のキャラクター、または以下の Secure Electronic Transaction に属する固定 2 文字の結合。

文字	意味
+	追加
-	減算

文字	意味
*	乗算
/	除算
**	指数

算術ステートメント (* arithmetic statement)

算術演算を実行させるステートメント。算術ステートメントは、ADD、COMPUTE、DIVIDE、MULTIPLY、SUBTRACT である。

配列 (array)

データ・オブジェクトで構成される集合体。それぞれのオブジェクトは添え字付けによって一意的に参照できる。配列は、COBOL ではテーブルに類似する。

昇順キー (* ascending key)

データ項目を比較する際の規則に一致するように、最低のキー値から始めて最高のキー値へとデータを順序付けている値に即したキー。

ASCII

情報交換用米国標準コード。基準コードは、7 ビットの符号化キャラクター (パリティ検査を含む 8 ビット) に基づく符号化キャラクターセットを使用します。この基準は、データ処理システム、データコミュニケーションシステム、および関連付けられている装置間の情報交換に使用されます。ASCII セットは、制御文字と図形文字から構成されている。

IBM は、ASCII に対する拡張 (文字 128 から 255) を定義している。

ASCII DBCS

「2 バイト ASCII (*double-byte ASCII*)」を参照。

割り当て名 (assignment-name)

COBOL ファイルの編成を識別する名前、システムがこれを認識する際に使用する。

想定小数点 (* assumed decimal point)

データ項目の中に実際には小数点のための文字が入っていない小数点位置。想定小数点には、論理的な意味があり、物理的には表現されない。

AT END 条件 (condition)

特定の条件下で、READ、RETURN、または SEARCH ステートメントの実行中に発生する条件。

- READ ステートメントは、ファイル内に次の論理レコードが存在しない場合、相対レコード番号の有効数字の数が相対キー・データ項目のサイズより大きい場合、またはオプションの入力ファイルが使用できない場合に、順次アクセス・ファイルに対して実行される。
- RETURN ステートメントは、関連付けられたソート・ファイルまたはマージ・ファイルに次の論理レコードが存在しない場合に実行される。
- SEARCH ステートメントは、関連する WHEN 句のいずれかで指定された条件を満たさずに検索操作が終了すると実行される。

B

基本文字セット (basic character set)

言語のワード、文字ストリング、および区切り文字の作成時に使用される基本的な文字セット。基本文字セットは 1 バイトの EBCDIC でインプリメントされる。拡張文字セットには DBCS 文字が含まれる。これは、コメント、リテラル、およびユーザー定義語で使用できる。

85 COBOL 標準における「COBOL 文字セット (COBOL character set)」と同義。

バッチ・コンパイル

プログラム・シーケンスと同義。

ビッグ・エンディアン (big-endian)

メインフレームおよび AIX® ワークステーションがバイナリー・データおよび UTF-16 文字を保存するときに使用するデフォルト形式。この形式では、バイナリー・データ項目の最下位バイトが最高位アドレスにあり UTF-16 文字の最下位バイトが最高位アドレスにあります。リトル・エンディアン (*little-endian*) と比較。

バイナリー項目 (binary item)

2 進表記 (基数 2 の数体系) で表される数値データ項目。等価の 10 進数は、10 進数字 0 から 9 に演算符号を加えたもので構成される。項目の左端のビットは、演算符号。

二分探索 (binary search)

二分探索の各段階では、データ・エレメント集合が 2 つに分割される。数が奇数の場合は適切なアクションが取られる。

ブロック (* block)

通常は 1 つ以上の論理レコードで構成される物理的データ単位。大容量記憶ファイルの場合、ある論理レコードの一部がブロックに入ることがある。ブロックのサイズは、そのブロックが含まれているファイルのサイズと直接関係はなく、そのブロックに含まれているか、そのブロックにオーバーラップしている論理レコードのサイズとも直接関係はない。「物理レコード (physical record)」と同義。

ブール条件 (boolean condition)

ブール条件は、ブール・リテラルが true であるか false であるかを決定する。ブール条件は定数条件式でのみ使用できる。

ブール・リテラル (boolean literal)

true 値を示す B'1'、または false 値を示す B'0' のどちらか。ブール・リテラルは定数条件式でのみ使用できる。

停止点 (breakpoint)

通常は命令によって指定されるコンピューター・プログラムの場所であり、プログラムの実行は外部からの介入またはモニター・プログラムによって割り込まれる場合がある。

buffer

入力データまたは出力データを一時的に保持するために使用されるストレージの一部分。

組み込み関数 (built-in function)

組み込み関数 (*intrinsic function*) を参照。

ビジネス・メソッド

ビジネス・ロジックまたはアプリケーションのルールをインプリメントする Enterprise Bean のメソッド。(Oracle)

バイト位置

特定の数のビット (通常 8 ビット) から成るストリングであり、1 つの単位として処理され、1 つの文字または制御機能を表す。

バイト・オーダー・マーク (BOM) (byte order mark (BOM))

UTF-16 または UTF-32 テキストの先頭に使用して、後続テキストのバイト・オーダーを示す Unicode 文字。バイト・オーダーには、「ビッグ・エンディアン (big-endian)」または「リトル・エンディアン (little-endian)」がある。

バイトコード (bytecode)

Java コンパイラーによって生成され、Java インタープリターによって実行される、マシンから独立したコード。(Oracle)

C

呼び出し可能サービス (callable services)

言語環境プログラムでは、従来の言語環境プログラム定義の呼び出しインターフェースを使用して COBOL プログラムが呼び出すことができる一連のサービス。言語環境プログラムの規約を共有するすべてのプログラムがこれらのサービスを使用できる。

呼び出し先プログラム (called program)

CALL ステートメントのオブジェクトになるプログラム。呼び出し先プログラムと呼び出し側プログラムが実行時に結合されて、1 つの実行単位が作成される。

呼び出し側プログラム (* calling program)

別のプログラムに対して CALL を実行するプログラム。

標準分解 (canonical decomposition)

複数の Unicode 文字を使用して単一の合成済み Unicode 文字を表す方法。通常、標準分解は、ラテン語文字と発音区別符号が個別に表されるように発音区別符号付きのラテン語文字を分離するために使用される。合成済み Unicode 文字とその標準分解を表す例については、合成済み文字 (*precomposed character*) を参照。

ケース構造 (case structure)

結果として生じた多数のアクションの中から選択を行うために、一連の条件をテストするプログラム処理ロジック。

カタログ式プロシージャ (cataloged procedure)

プロシージャ・ライブラリー (SYS1.PROCLIB) と呼ばれる区分データ・セットに置かれた一連のジョブ制御ステートメント。カタログ式プロシージャを使用すると、JCL をコーディングする時間を節約して、エラーを減らすことができる。

CCSID

コード化文字セット ID (coded character set identifier) を参照。

世紀ウィンドウ (century window)

2 桁年号が固有に決まる 100 年間のこと。COBOL プログラマーが使用できる世紀ウィンドウには、いくつかのタイプがある。

- ウィンドウ操作組み込み関数 DATE-TO-YYYYMMDD、DAY-TO-YYYYDDD、および YEAR-TO-YYYY の場合は、*argument-2* を使用して世紀ウィンドウを指定する。
- 言語環境プログラム呼び出し可能サービスについては、CEEScen で世紀ウィンドウを指定する。

文字 (* character)

言語のそれ以上分割できない基本単位。

文字エンコード・ユニット (character encoding unit)

コード化文字セット内の 1 つのコード・ポイントに相当するデータの単位。1 つ以上の文字エンコード・ユニットを使用して、コード化文字セットの文字が表現される。エンコード・ユニットとも呼ばれる。

USAGE NATIONAL の場合、文字エンコード・ユニットは UTF-16 の 1 つの 2 バイト・コード・ポイントに対応する。

USAGE DISPLAY の場合、文字エンコード・ユニットは 1 バイトに対応する。

USAGE DISPLAY-1 の場合、文字エンコード・ユニットは DBCS 文字セットの 2 バイト・コード・ポイントに対応する。

文字位置 (character position)

1 文字を保持または表示するために必要な物理ストレージまたは表示スペースの量。この用語はどのような文字のクラスにも適用される。文字の特定のクラスについては、以下の用語が適用される。

- 英数字の文字位置 (USAGE DISPLAY で表される文字の場合)
- DBCS 文字位置 (USAGE DISPLAY-1 で表される DBCS 文字の場合)
- 国別文字位置 (USAGE NATIONAL で表される文字の場合)。文字エンコード・ユニット (UTF-16 の場合) と同義。

文字セット (character set)

テキスト情報を表すために使用されるエレメントの集合。ただし、コード化表現は想定されていません。コード化文字セット (coded character set) も参照。

文字列

COBOL ワード、リテラル、PICTURE 文字ストリング、またはコメント記入項目を形成する連続した文字のシーケンス。文字ストリングは区切り文字で区切らなければならない。

チェックポイント (checkpoint)

ジョブ・ステップを後で再始動することができるように、ジョブとシステムの状況に関する情報を記録しておくことができる場所。

クラス (* class)

ゼロ、1 つ、または複数のオブジェクトの共通の動作およびインプリメンテーションを定義するエンティティ。同じ具体化を共有するオブジェクトは、同じクラスのオブジェクトとみなされる。クラスは階層として定義でき、あるクラスを別のクラスから継承することができる。

クラス (オブジェクト指向) (class (object-oriented))

ゼロ、1 つ、または複数のオブジェクトの共通の動作およびインプリメンテーションを定義するエンティティ。同じ具体化を共有するオブジェクトは、同じクラスのオブジェクトとみなされる。

クラス条件 (* class condition)

項目の内容がすべて英字であるか、すべて数字であるか、すべて DBCS であるか、すべて漢字であるか、あるいはクラス名の定義においてリストされた文字だけで構成されるかという命題で、それに関して真の値を判別することができる。

クラス定義 (* class definition)

クラスを定義する COBOL ソース単位。

クラス階層 (class hierarchy)

オブジェクト・クラス間の関係を示すツリーのような構造。最上部に 1 つのクラスが置かれ、その下に 1 つ以上のクラスの層が置かれる。「継承階層 (inheritance hierarchy)」と同義。

クラス識別記入項目 (* class identification entry)

IDENTIFICATION DIVISION の CLASS-ID 段落の項目。この項目には、クラス名を指定し、選択した属性をクラス定義に割り当てる文節が含まれる。

クラス名 (オブジェクト指向) (class-name (object-oriented))

オブジェクト指向 COBOL クラス定義の名前。

クラス名 (データの) (* class-name (of data))

ENVIRONMENT DIVISION の SPECIAL-NAMES 段落で定義されるユーザー定義語。この語は、データ項目の内容がクラス名の定義にリストされている文字のみで構成されている命題 (真の値を定義することができる) に名前を割り当てる。

クラス・オブジェクト (class object)

クラスを表す実行時オブジェクト。

文節 (* clause)

記入項目の属性を指定するという目的で順番に並べられた連続する COBOL 文字ストリング。

クライアント

オブジェクト指向プログラミングにおいて、クラス内の 1 つ以上のメソッドからサービスを要求するプログラムまたはメソッド。

COBOL 文字セット (COBOL character set)

COBOL 構文を作成する際に使用される文字セット。完了した COBOL キャラクター Secure Electronic Transaction は、次の文字で構成されています。

文字	意味
0,1,...,9	数字
A,B,...,Z	英大文字
a,b,...,z	英小文字
	スペース
+	正符号
-	負符号 (-) (ハイフン)
*	アスタリスク
/	斜線 (スラッシュ)
=	等号
\$	通貨符号
,	コンマ
;	セミコロンの
.	ピリオド (小数点、終止符)
"	引用符
'	アポストロフィ
(左括弧

文字	意味
)	右括弧
>	より大きい
<	より小さい
:	コロソ
-	下線

COBOL ワード (* COBOL word)

ワード (word) を参照。

コード・ページ (code page)

すべてのコード・ポイントに図形文字および制御機能の意味を割り当てるもの。例えば、あるコード・ページでは、8 ビット・コードに対して 256 コード・ポイントに文字と意味を割り当て、別のコード・ページでは、7 ビット・コードに対して 128 コード・ポイントに文字と意味を割り当てることができる。例えば、作業場上の英語の IBM 符号化ページの 1 つは イブソ-1252 であり、ホスト上には イブソ-1047 というコード・ページがあります。コード化キャラクター *Secure Electronic Transaction*。

コード・ポイント (code point)

コード化文字セット (コード・ページ) に定義する固有のビット・パターン。コード・ポイントには、グラフィック・シンボルおよび制御文字が割り当てられる。

コード化文字セット (coded character set)

文字セットを設定し、その文字セットの文字とコード化表現との間の関係を設定する明確な規則の集まり。コード化文字セットの例として、ASCII もしくは EBCDIC コード・ページで、または Unicode 対応の UTF-16 エンコード・スキームで表す文字セットがある。

コード化文字セット ID (CCSID) (coded character set identifier (CCSID))

特定のコード・ページを識別する 1 から 65,535 までの IBM 定義番号。

照合シーケンス (* collating sequence)

コンピューターに受け入れ可能な文字のシーケンスで、ソート、マージ、比較、および索引付きファイルの順次処理を目的として順序付けしたもの。

列 (* column)

印刷行または参照形式行におけるバイト位置。列は、行の左端の位置から始めて行の右端の位置まで、1 から 1 ずつ増やして番号が付けられる。列は 1 つの 1 バイト文字を保持する。

複合条件 (* combined condition)

2 つ以上の条件を AND または OR 論理演算子で結合した結果である条件。条件 (condition) および 複合否定条件 (negated combined condition) も参照。

結合文字 (combining characters)

他の前後の Unicode 文字を変更するために使用される Unicode 文字。通常、結合文字は、ラテン語文字を変更するために使用される Unicode 発音区別符号。合成済み文字 (precomposed character) を参照して、ラテン語文字 U+0061 (a) とともに使用される結合文字 U+0308 (¨) の例を確認すること。

コメント記入項目 (* comment-entry)

文書化のために使用され、実行には影響しない IDENTIFICATION DIVISION 内の項目。

コメント行 (comment line)

行の標識域内のアスタリスク (*) または、プログラム・テキスト域 (領域 A と領域 B) の最初の文字ストリングとしての、アスタリスクの後の大なり記号 (>) で表されるソース・プログラム行と、その行の領域 A と領域 B にあるコンピューターの文字セット 後続の 任意の文字。コメント行は、文書化にのみ役立つ。行の標識域に斜線 (/) で表される特殊な形式のコメント行と、その行の領域 A および領域 B にあるコンピューターの文字セットからの文字があると、コメントの出力前にページが排出される。

共通プログラム (* common program)

別のプログラムに直接的に含まれているにもかかわらず、その別のプログラムに直接的または間接的に含まれている任意のプログラムから呼び出すことができるプログラム。

コンパイル・グループ (compilation group)

プログラム・シーケンスと同義。

コンパイル変数 (compilation variable)

特定のリテラル値のシンボル名、または DEFINE ディレクティブまたは DEFINE コンパイラー・オプションによって指定されたコンパイル時の算術式の値。

コンパイル (* compile)

(1) 高水準言語で表現されたプログラムを、中間言語、アセンブリ言語、またはコンピューター言語で表現されたプログラムに変換すること。(2) あるプログラミング言語で書かれたコンピューター・プログラムから、プログラムの全体的なロジック構造を利用することによって、または 1 つの記号ステートメントから複数のコンピューター命令を作り出すことによって、またはアセンブラの機能のようにこれら両方を使用することによって、マシン言語プログラムを生成すること。

コンパイル時 (* compile time)

COBOL コンパイラーによって、COBOL ソース・コードが COBOL オブジェクト・プログラムに変換される時間。

コンパイル時演算式 (compile-time arithmetic expression)

DEFINE ディレクティブおよび EVALUATE ディレクティブ、または定数条件式で指定される算術式のサブセット。コンパイル時演算式における、正規演算式との違い:

- ・ 指数演算子を指定することはできません。
- ・ オペランドはすべて、整数リテラルか、すべてのオペランドが整数リテラルである演算式でなければなりません。
- ・ 式はゼロによる除算が発生しないように指定する必要があります。

コンパイラー

高水準言語で記述されたソース・コードをマシン言語のオブジェクト・コードに変換するプログラム。

コンパイラー指示ステートメント (compiler-directing statement)

コンパイル時にコンパイラーに特定の処置を行わせるステートメント。標準のコンパイラー指示ステートメントは、COPY、REPLACE、USE である。

複合条件 (* complex condition)

1 つ以上の論理演算子が 1 つ以上の条件に基づいて作動する条件。条件 (condition)、単純否定条件 (negated simple condition)、および 複合否定条件 (negated combined condition) も参照。

複合 ODO (complex ODO)

OCCURS DEPENDING ON 節の特定の形式:

- ・ 可変位置項目またはグループ: DEPENDING ON オプションを指定した OCCURS 節によって記述されたデータ項目の後に、非従属データ項目またはグループが続く。グループは英数字グループでも国別グループでも構いません。
- ・ 可変位置テーブル: DEPENDING ON オプションを指定した OCCURS 節によって記述されたデータ項目の後に、OCCURS 節によって記述された非従属データ項目が続く。
- ・ 可変長エレメントを持つテーブル: OCCURS 節によって記述されるデータ項目には、DEPENDING ON オプションを指定した OCCURS 節によって記述される従属データ項目が含まれる。
- ・ 可変長エレメントを持つテーブルの指標名。
- ・ 可変長エレメントを持つテーブルのエレメント。

コンポーネント (component)

(1) 関連ファイルからなる機能グループ化。(2) オブジェクト指向プログラミングでは、特定の機能を実行し、他のコンポーネントやアプリケーションと連携するように設計されている、再使用可能なオブジェクトまたはプログラム。JavaBeans は、コンポーネントを作成するための、Oracle が提供するアーキテクチャーである。

合成形式 (composed form)

標準分解による合成済み Unicode 文字の表記。詳しくは、合成済み文字 (precomposed character) を参照。

コンピューター名 (* computer-name)

プログラムがコンパイルまたは実行されるコンピューターを識別するシステム名。

条件 (例外) (condition (exception))

言語環境プログラムによって使用可能にされる、あるいは認識される例外。したがって、ユーザー条件処理ルーチンと言語条件処理ルーチンの活動化に適している。アプリケーションの通常のプログラミングされたフローを変えるもの。条件は、ハードウェアまたはオペレーティング・システムによって検出され、その結果、割り込みが起こる。このほかにも、条件は言語特定の生成コードまたは言語ライブラリー・コードによっても検出できる。

条件 (式) (condition (expression))

ある真の値が決定される実行時のデータの状況。本書において、"条件" (*condition-1*、*condition-2*) に関連して使用される場合。) 次のいずれかである。オプションとして括弧で囲まれた単純条件からなる条件式、あるいは、単純条件、論理演算子、および括弧の構文的に正しい組み合わせ (真理値を判別できる) からなる複合条件。単純条件 (*simple condition*)、複合条件 (*complex condition*)、単純否定条件 (*negated simple condition*)、複合条件 (*combined condition*)、および 複合否定条件 (*negated combined condition*) も参照。

条件式 (* conditional expression)

EVALUATE、IF、PERFORM、または SEARCH ステートメントで指定された単純条件または複合条件。単純条件 (*simple condition*) および 複合条件 (*complex condition*) も参照。

条件句 (* conditional phrase)

ある条件ステートメントが実行された結果得られる条件の真理値の判別に基づいてとられるべき処置を指定する句。

条件ステートメント (* conditional statement)

条件の真理値を判別することと、オブジェクト・プログラムの次の処理がこの真理値によって決まることを指定するステートメント。

条件変数 (* conditional variable)

1つまたは複数の値を持つデータ項目であり、これらの値が、そのデータ項目に割り当てられた条件名を持つ。

条件名 (* condition-name)

条件変数が想定できる値のサブセットに名前を割り当てるユーザー定義語。または、インプリメントする人が定義したスイッチまたは装置の状況に割り当てられるユーザー定義語。

条件名条件 (* condition-name condition)

真理値を判別できる命題で、かつ、条件変数の値が、その条件変数と関連する条件名に属する一連の値のメンバーである命題。

* CONFIGURATION SECTION

ソース・プログラム、オブジェクト・プログラム、およびクラス定義の全体的な仕様を記述する ENVIRONMENT DIVISION のセクション。

コンソール

オペレーター・コンソールに関連する COBOL 環境名。

定数条件式 (constant conditional expression)

IF ディレクティブまたは EVALUATE ディレクティブの WHEN 句の中で使用できる条件式のサブセット。

定数条件式は、以下の項目のいずれかでなければなりません。

- 両方のオペランドがリテラルであるか、リテラル項のみを含む算術式である比較条件。条件は比較条件の規則に従う必要があり、以下の追加事項があります。
 - オペランドは同じカテゴリーでなければなりません。算術式は数値カテゴリーです。
 - リテラルが指定され、それらが数値リテラルでない場合、関係演算子は "IS EQUAL TO"、"IS NOT EQUAL TO"、"IS ="、"IS NOT ="、または "IS <>" になる。

比較条件 (*relation condition*) も参照。

- 定義済み条件。定義済み条件 (*defined condition*) も参照。
- ブール条件。ブール条件 (*boolean condition*) も参照。

- ・ 前述の単純条件の形式を、AND、OR、および NOT を使用して複合条件に結合することによって形成した複合条件。簡略複合比較条件を指定することはできません。複合条件 (*complex condition*) も参照。

含まれているプログラム (**contained program**)

別の COBOL プログラムにネストされている COBOL プログラム。

連続項目 (*** contiguous items**)

DATA DIVISION 内の連続した項目によって記述され、相互に一定の階層関係を持つ項目。

コピーブック (**copybook**)

COPY ステートメントを使用してコンパイル時にソース・プログラムに組み込まれる一連のコードが入っているファイルまたはライブラリー・メンバー。ファイルはユーザーが作成する場合、COBOL によって提供される場合、または他の製品によって供給される場合とがある。「コピー・ファイル (*copy file*)」と同義。

カウンター (*** counter**)

他の数字を使ってその数字分だけ増減したり、あるいは 0 または任意の正もしくは負の値に変更またはリセットしたりできるようにした、数または数表現を収めるために使用されるデータ項目。

相互参照リスト (**cross-reference listing**)

コンパイラー・リストの一部であり、プログラム内においてファイル、フィールド、および標識が定義、参照、および変更される場所に関する情報が入る。

通貨記号値 (**currency-sign value**)

数字編集項目に保管される通貨単位を識別する文字ストリング。典型的な例としては、\$、USD、EUR などがある。通貨符号値は、CURRENCY コンパイラー・オプション、または ENVIRONMENT DIVISION の SPECIAL-NAMES 段落の CURRENCY SIGN 節のいずれかによって定義できる。CURRENCY SIGN 節が指定されておらず、NOCURRENCY コンパイラー・オプションが有効な場合は、ドル記号 (\$) がデフォルトの通貨符号値として使用される。通貨記号 (*currency symbol*) も参照。

通貨記号 (**currency symbol**)

数字編集項目内の通貨符号値の位置を示すために PICTURE 節で使用される文字。通貨記号は、ENVIRONMENT DIVISION の SPECIAL-NAMES 段落で CURRENCY コンパイラー・オプションまたは CURRENCY SIGN 節のいずれかによって定義できる。CURRENCY SIGN 節が指定されず、NOCURRENCY コンパイラー・オプションが有効な場合、デフォルトの通貨符号値および通貨記号としてドル記号 (\$) が使用される。通貨記号と通貨符号値は複数定義可能。通貨記号値 (*currency sign value*) も参照。

現行レコード (*** current record**)

ファイル処理において、ファイルに関連するレコード域の中で使用可能なレコード。

現行ボリューム・ポインター (*** current volume pointer**)

順次ファイルの現行のボリュームを指している概念上のエンティティー。

D

データ節 (*** data clause**)

COBOL プログラムの DATA DIVISION 内のデータ記述記入項目に現れる文節で、データ項目の特定の属性を記述する情報を提供する。

データ記述項目 (*** data description entry**)

COBOL プログラムの DATA DIVISION 内の記入項目。必要に応じて、レベル番号の後にデータ名を付け、その後に一連のデータ文節を続けることによって構成される。

DATA DIVISION

プログラムまたはメソッドによって処理されるデータを記述する COBOL プログラムまたはメソッドの部。使用されるファイルとその中に含まれるレコード、必要とされる内部 WORKING-STORAGE レコード、COBOL 実行単位内の複数のプログラムで使用可能になるデータ。

データ項目 (*** data item**)

COBOL プログラムにより、または関数評価の規則により、定義されたデータの単位 (リテラルを除く)。

データ・セット

「ファイル (*file*)」の同義語。

データ名 (* data-name)

データ記述項目で記述されたデータ項目に名前を割り当てるユーザー定義語。一般形式で使用された場合、データ名は、その形式の規則で特に許可されていない限り、参照変更、添え字付け、または修飾してはならないワードを表す。

DBCS

2 バイト文字セット (*double-byte character set (DBCS)*) を参照

DBCS 文字 (DBCS character)

IBM 2 バイト文字セット (DBCS) に定義された任意の文字。

DBCS 文字位置 (DBCS character position)

文字位置 (*character position*) を参照。

DBCS データ項目 (DBCS data item)

少なくとも 1 つの記号 G、または NSYMBOL (DBCS) コンパイラー・オプションが有効な場合は少なくとも 1 つの記号 N を含む PICTURE 文字ストリングによって記述されるデータ項目。DBCS データ項目には USAGE DISPLAY-1 がある。

デバッグ行 (* debugging line)

行の標識区域に文字 D がある行のこと。

デバッグ・セクション (* debugging section)

USE FOR DEBUGGING ステートメントを含むセクション。

宣言文 (* declarative sentence)

分離文字ピリオドで終了する単一の USE ステートメントで構成されるコンパイラー指示文。

宣言部分 (* declaratives)

PROCEDURE DIVISION の先頭に書き込まれる特殊な目的を有する 1 つ以上のセクションの集合。最初のセクションの前にはキーワード DECLARATIVE が付き、最後のセクションの後にはキーワード END DECLARATIVES が続く。1 つの宣言は、セクション・ヘッダーと次に続く USE コンパイラー指示文、その後に続く関連する段落から構成される。この段落は、なくてもよいし、1 個でも複数個でもよい。

編集解除 (* de-edit)

項目の編集解除された数値を判別するために、数字編集データ項目からすべての編集文字を論理的に除去すること。

定義済み条件 (defined condition)

コンパイル変数が定義されているかどうかをテストするコンパイル時条件。定義条件は、IF ディレクティブまたは EVALUATE ディレクティブの WHEN 句で指定される。

範囲区切りステートメント (* delimited scope statement)

明示的範囲終了符号を含んでいるステートメント。

区切り文字 (* delimiter)

1 つの文字、または一連の連続する文字であり、文字ストリングの終わりを識別し、その文字ストリングを後続の文字ストリングから区切る。区切り文字は、これを使用して区切られる文字ストリングの一部ではない。

依存地域

IMS において、メッセージ・ドリブン・プログラム、バッチ・プログラム、またはオンライン電力/ガスを含む多重仮想ストレージ 仮想ストレージ地域。

降順キー (* descending key)

データ項目を比較する際の規則に一致するように、最高のキー値から始めて最低のキー値へとデータを順序付けている値に付けられるキー。

digit

0 から 9 までの数値のいずれか。COBOL では、用語はその他のシンボルを参照するために使用されることはありません。

桁位置 (* digit position)

1 つの桁を保管するために必要な物理ストレージの大きさ。この大きさは、データ項目を定義するデータ記述項目に指定された用途によって異なる。

直接アクセス (* direct access)

前回アクセスしたデータへの参照には依存せず、プロセスがデータの位置にのみ依存する方法で、データを記憶装置から取得したり、データを記憶装置に入れる機能。

表示浮動小数点データ項目 (display floating-point data item)

USAGE DISPLAY として暗黙的または明示的に記述され、外部浮動小数点データ項目を記述する PICTURE 文字ストリングを持つデータ項目。

部 (* division)

部の本体と呼ばれる、0 個、1 個、または複数個のセクションまたは段落の集合であり、特定の規則に従って形成および結合されたもの。それぞれの部は、部のヘッダーおよび関連した部の本体で構成される。COBOL プログラムには、見出し部、環境部、データ部、および手続き部の 4 つの部がある。

部の見出し (* division header)

ワードとその後に続く、部の先頭を示す分離文字ピリオドの組み合わせ。部のヘッダーは次のとおり。

```
IDENTIFICATION DIVISION.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
PROCEDURE DIVISION.
```

DLL

ダイナミック・リンク・ライブラリー (DLL) (dynamic link library (DLL)) を参照。

DLL アプリケーション

インポートしたプログラム、関数、または変数を参照するアプリケーション。

ダイナミック・リンク・ライブラリー結合

DLL および NODYNAM オプションを指定してコンパイルされたプログラム内の CALL。CALL は、別のモジュール内のエクスポートされた名前、または別のモジュール内で定義されたメソッドの INVOKE に解決される。

Do 構造 (do construct)

構造化プログラミングでは、DO ステートメントを使用して、プロシージャ内いくつかのステートメントをグループ化する。COBOL では、インライン PERFORM ステートメントが同じように機能する。

do-until

構造化プログラミングにおいて、do-until ループは、少なくとも 1 回は実行され、所定の条件が真になるまで実行される。COBOL では、PERFORM ステートメントで使用される TEST AFTER 句が同じように機能する。

do-while

構造化プログラミングにおいて、do-while ループは、所定の条件が真である場合、および真である間に実行される。COBOL では、PERFORM ステートメントで使用される TEST BEFORE 句が同じように機能する。

文書タイプ宣言 (document type declaration)

あるクラスの文書に対する文法を規定するマークアップ宣言を含む、または指示する XML エレメント。この文法は、文書タイプ定義または DTD とも呼ばれる。

文書化型定義 (文書タイプ定義)

XML 文書のクラスの文法。文書タイプ宣言を参照。

2 バイト ASCII (double-byte ASCII)

DBCS 文字および 1 バイト ASCII 文字を含む IBM の文字セット (「ASCII DBCS」 とも呼ばれる)。

2 バイト EBCDIC (double-byte EBCDIC)

DBCS 文字および 1 バイト EBCDIC 文字を含む IBM の文字セット (「EBCDIC DBCS」 とも呼ばれる)。

2 バイト文字セット (DBCS) (double-byte character set (DBCS))

それぞれの文字が 2 バイトで表現される 1 組の文字。256 個のコード・ポイントで表現される記号より多くの記号を含んでいる言語 (日本語、中国語、および韓国語など) は、2 バイト文字セットを必要とする。各文字に 2 バイトが必要なため、DBCS 文字の入力、表示、および印刷には、DBCS を受け入れ可能なハードウェアおよびサポートされるソフトウェアが必要。

DWARF

DWARF は UNIX International Programming Languages Special Interest Group (SIG) で開発された。言語に依存しないデバッグ情報を提供することにより、さまざまな言語の、統一された方法でのシンボリックなソース・レベル・デバッグのニーズを満たすように設計されている。DWARF ファイルには、さまざまなエレメントに編成されたデバッグ・データが含まれている。詳しくは、「DWARF/ELF エクステンション ライブラリー・リファレンス」の『*DWARF program information*』を参照。

動的アクセス (* dynamic access)

特定の論理レコードを大容量記憶ファイルから順序に関係なく取得したり、ファイル内に配置したりすることができるアクセス・モードで、同じ OPEN ステートメントの範囲内でファイルから論理レコードを順次取得できる。

動的 CALL (dynamic CALL)

DYNAM オプションを指定してコンパイルされているプログラム内の CALL *literal* ステートメント および NODLL オプション、またはプログラム内の CALL *identifier* ステートメント NODLL オプションを指定してコンパイルされたもの。

動的長 (dynamic-length)

実行時に変化する可能性がある論理長を持つ項目を記述する形容詞。

動的長基本項目 (dynamic-length elementary item)

データ宣言項目に DYNAMIC LENGTH 節が含まれる基本データ項目。

動的長グループ (dynamic-length group)

従属する動的長基本項目を含むグループ項目。

ダイナミック・リンク・ライブラリー (DLL) (dynamic link library (DLL))

リンク時ではなく、ロード時または実行時にプログラムにバインドされる実行可能コードおよびデータが入ったファイル。複数のアプリケーションが DLL 内のコードおよびデータを同時に共用することができる。DLL はプログラムの実行可能ファイルの一部ではないが、実行可能ファイルを正しく実行するためには必要となる可能性がある。

動的ストレージ域 (dynamic storage area (DSA))

動的に獲得されるストレージであり、レジスター保管域、および動的ストレージ割り振りに使用可能な区域 (プログラム変数など) から構成される。DSA は、プログラムまたは関数が呼び出されるときに割り振られ、呼び出しインスタンスの継続時間の間持続します。DSA は通常、言語環境プログラムによって管理されるスタック・セグメント内に割り振られる。

E

EBCDIC (拡張 2 進化 10 進コード) (* EBCDIC (Extended Binary-Coded Decimal Interchange Code))

8 ビット・コード化文字をベースとするコード化文字セット。

EBCDIC 文字 (EBCDIC character)

EBCDIC (拡張 2 進化 10 進コード) セットに含まれているいずれかの記号。

EBCDIC DBCS

「2 バイト EBCDIC (*double-byte EBCDIC*)」を参照。

編集データ項目 (edited data item)

0 の抑止または編集文字の挿入、あるいはその両方を行うことによって変更されたデータ項目。

編集用文字 (* editing character)

次に示す集合に属する 1 文字、または 2 文字で構成される固定した組み合わせ。

文字	意味
	スペース
0	0
+	正符号
-	負符号
CR	貸方
DB	借方

文字	意味
Z	ゼロの抑止
*	小切手変造防止
\$	通貨符号
,	コンマ (小数点)
.	ピリオド (小数点)
/	斜線 (スラッシュ)

EGCS

拡張図形文字セット (*extended graphic character set*) (EGCS) を参照。

EJB

Enterprise JavaBeans を参照。

EJB コンテナ

J2EE アーキテクチャーの EJB コンポーネント契約を実装するコンテナ。この契約は、セキュリティ、並行性、ライフ・サイクル管理、トランザクション、デプロイメント、およびその他のサービスを含んでいるエンタープライズ Bean 用のランタイム環境を指定します。EJB コンテナは、EJB サーバーまたは J2EE サーバーによって提供される。(Oracle)

EJB サーバー

EJB コンテナにサービスを提供するソフトウェア。EJB サーバーは、1 つ以上の EJB コンテナをホストできる。(Oracle)

エレメント (テキスト・エレメント) (element (text element))

1 つのデータ項目または動詞の記述などのようなテキスト・ストリングの 1 つの論理単位で、その前にエレメント・タイプを識別する固有のコードが付けられたもの。

基本項目 (* elementary item)

論理的にそれ以上細分できないものとして記述されているデータ項目。

カプセル化

オブジェクト指向プログラミングでは、オブジェクトの固有の詳細を隠すのに使用される技法。オブジェクトは、基礎構造を露出しなくても、データの照会と操作を行うインターフェースを提供する。「情報隠蔽 (*information hiding*)」と同義。

エンクレーブ (enclave)

言語環境プログラムのもとで実行される場合、エンクレーブは実行単位に類似している。で LINK を使用するか、C の system() 関数を使用することにより、あるエンクレーブから他のエンクレーブを作成できる。

エンコード・ユニット (encoding unit)

文字エンコード・ユニット (*character encoding unit*) を参照。

END CLASS マーカー (end class marker)

語の組み合わせに分離文字ピリオドが続いたもので、COBOL クラス定義の終わりを示す。クラス終了マーカーは次のとおり。

```
END CLASS class-name.
```

メソッド終了マーカー (end method marker)

語の組み合わせに分離文字ピリオドが続いたもので、COBOL メソッド定義の終わりを示す。メソッド終了マーカーは次のとおり。

```
END METHOD method-name.
```

PROCEDURE DIVISION の終わり

COBOL ソース・プログラムにおいて、それ以後にはプロシージャが存在しない物理的な位置。

プログラム終了マーカー (* end program marker)

語の組み合わせに分離文字ピリオドが続いたもので、COBOL ソース・プログラムの終わりを示す。プログラム終了マーカーは、次のように記述する。

```
END PROGRAM program-name.
```

企業 Bean

ビジネス・タスクを実装し、EJB コンテナに存在するコンポーネント。(Oracle)

Enterprise JavaBeans

オブジェクト指向の分散エンタープライズ・レベル・アプリケーションの開発とデプロイメントのために、Oracle によって定義されたコンポーネント・アーキテクチャー。

項目 (* entry)

分離文字ピリオドで終了する連続する節の記述セットであり、COBOL プログラムの IDENTIFICATION DIVISION、ENVIRONMENT DIVISION、または DATA DIVISION に書き込まれる。

環境節 (* environment clause)

ENVIRONMENT DIVISION 項目の一部として表示される文節。

ENVIRONMENT DIVISION

COBOL プログラム、クラス定義、またはメソッド定義の 4 つの主コンポーネントの 1 つ。ENVIRONMENT DIVISION では、ソース・プログラムがコンパイルされるコンピューターと、オブジェクト・プログラムが実行されるコンピューターを記述する。この部では、ファイルの論理概念とそのレコードの間のリンケージ、およびファイルが保管される装置の物理的局面を提供する。

環境名 (environment-name)

IBM が指定する名前であり、システム論理装置、プリンターおよびカード穿孔装置の制御文字、報告書コード、またはプログラム・スイッチ、あるいはそれらの組み合わせを識別する。環境名が ENVIRONMENT DIVISION の簡略名と関連付けられている場合は、そのニーモニック名は、その簡略名を、置換が有効な任意の形式で置き換えることができる。

環境変数 (environment variable)

コンピューター環境の一部の局面を定義する多数の変数のいずれかであり、その環境で動作するプログラムからアクセス可能。環境変数は、動作環境に依存するプログラムの動作に影響を与える。

エスケープ・シーケンス

ストリング・文字列およびキャラクターリテラル内の特定の特殊文字を表すために使用される文字の一連。

エスケープするシーケンスは 2 つ以上のキャラクターで構成され、最初の文字は「エスケープ文字」と呼ばれるバックスラッシュ (\) 文字です。残りの文字は、エスケープ・シーケンスの変換処理を決定します。例えば、\n は、改行文字を意味するエスケープ・シーケンスである。

エスケープ・シーケンスは、C、C++、Java、Python などのプログラミング言語で使用する。COBOL には、「エスケープ・シーケンス」または「エスケープ文字」というコンセプトがありません。COBOL リテラル内で特殊文字を扱うには、Enterprise COBOL for z/OS 言語解説書の基本英数字リテラルおよび DBCS リテラルを参照。

実行時 (execution time)

実行時 (run time) を参照。

実行時環境 (execution-time environment)

ランタイム環境 (runtime environment) を参照。

明示的範囲終了符号 (* explicit scope terminator)

特定の PROCEDURE DIVISION ステートメントの有効範囲を終わらせる予約語。

指数 (exponent)

別の数 (底) をべき乗する指数を示す数。正の指数は乗算を示し、負の指数は除算を示し、小数の指数は数量の根を示す。COBOL では、指数式は記号 ** の後に指数を付けて表す。

式 (* expression)

算術式または条件式。

拡張モード (* extend mode)

ファイルに対する EXTEND 句の指定のある OPEN ステートメントが実行されてから、そのファイルに対する REEL 句または UNIT 句の指定のない CLOSE ステートメントが実行される前までの、ファイルの状態。

拡張図形文字セット (extended graphic character set) (EGCS)

それぞれの図形文字を表すために 2 バイトを必要とする図形文字セット (漢字文字セットなど)。現在は、2 バイト文字セット (DBCS) と呼ばれるようになった。

Extensible Markup Language

XML を参照。

拡張 (extensions)

85 COBOL 標準に記述されているものの他に、IBM コンパイラーでサポートされる COBOL 構文およびセマンティクス。

外部コード・ページ (external code page)

XML 文書で、CODEPAGE コンパイラー・オプションで指定された値。

外部データ (* external data)

プログラムの中で外部データ項目および外部ファイル結合子として記述されるデータ。

外部データ項目 (* external data item)

実行単位の 1 つ以上のプログラムにおいて外部レコードの一部として記述されるデータ項目であり、その項目が記述されている任意のプログラムから参照することができる。

外部データ・レコード (* external data record)

実行単位の 1 つ以上のプログラムにおいて記述される論理レコードであり、そのデータ項目は、それらが記述されている任意のプログラムから参照できる。

外部 10 進数データ項目 (external decimal data item)

ゾーン 10 進数データ項目 (zoned decimal data item) および 国別 10 進数データ項目 (national decimal data item) を参照。

外部ファイル結合子 (* external file connector)

実行単位の 1 つ以上のオブジェクト・プログラムにアクセス可能なファイル結合子。

外部浮動小数点データ項目 (external floating-point data item)

表示浮動小数点データ項目 (display floating-point data item) および 国別浮動小数点データ項目 (national floating-point data item) を参照。

外部プログラム (external program)

最外部プログラム。ネストされていないプログラム。

外部スイッチ (* external switch)

インプリメントする人によって定義され、名前が付けられたハードウェア装置またはソフトウェアで、2 つの選択的な状態のうち一方が存在することを示すために使用される。

F

ファクトリー・データ (factory data)

いったんクラスに割り振られ、クラスのすべてのインスタンスに共用されるデータ。ファクトリー・データは、クラス定義の FACTORY 段落の DATA DIVISION の WORKING-STORAGE SECTION で宣言される。Java private 静的データと同等。

ファクトリー・メソッド (factory method)

オブジェクト・インスタンスとは無関係に、クラスによってサポートされるメソッド。ファクトリー・メソッドは、クラス定義の FACTORY 段落に宣言される。Java public 静的メソッドと同等。これらは通常、オブジェクトの作成をカスタマイズするために使用されます。

形象定数 (* figurative constant)

ある予約語を使用することによって参照されるコンパイラー生成の値。

ファイル (* file)

論理レコードの集合。

ファイル属性対立条件 (* file attribute conflict condition)

あるファイルで入出力操作を実行する試みが失敗し、プログラムの中でそのファイルに対して指定したファイル属性が、そのファイルの固定属性と一致していないこと。

ファイル節 (* file clause)

ファイル記述項目 (FD 項目) およびソート・マージ・ファイル記述項目 (SD 項目) のいずれかの一部として現れる DATA DIVISION 文節。

ファイル結合子 (* file connector)

ファイルに関する情報が入っており、ファイル名と物理ファイルの間のリンケージとして、さらにファイル名とその関連レコード域の間のリンケージとして使用されるストレージ域。

ファイル制御 (File-Control)

ソース・プログラムで用いられるデータ・ファイルが宣言されている環境部の段落の名前。

ファイル制御ブロック (FCB) (file control block)

I/O ルーチンのアドレス、それらがどのようにオープンおよびクローズされたかに関する情報、およびファイル情報ブロック (FIB) へのポインターを含むブロック。

ファイル制御項目 (* file control entry)

SELECT 節と、ファイルの関連物理属性を宣言するすべての従属節。

FILE-CONTROL 段落

ENVIRONMENT DIVISION 内の段落であり、この中では、特定のソース単位で使用されるデータ・ファイルが宣言される。

ファイル記述項目 (* file description entry)

DATA DIVISION の FILE SECTION 内の中にある記入項目項目。レベル標識 FD と、それに続くファイル名、および必要に応じて一連のファイル文節から構成される。

ファイル名 (* file-name)

DATA DIVISION の FILE SECTION 内のファイル記述項目またはソート・マージ・ファイル記述項目に記述されているファイル結合子に名前を付けるユーザー定義語。

ファイル編成 (* file organization)

ファイルの作成時に確立される永続的な論理ファイル構造。

ファイル位置標識 (file position indicator)

概念的エンティティであり、索引付きファイルの場合は参照キー内の現行キーの値、順次ファイルの場合は現行レコードのレコード番号、相対ファイルの場合は現行レコードの相対レコード番号が入っている。あるいは、次の論理レコードが存在しないことを示すか、オプションの入力ファイルが使用可能でないことを示すか、AT END 条件が既に存在していることを示すか、もしくは有効な次のレコードが設定されていないことを示す。

* FILE SECTION

DATA DIVISION のセクションであり、ファイル記述項目、ソート・マージ・ファイル記述項目、および関連するレコード記述が入っている。

ファイル・システム

データ・レコードおよびファイル記述プロトコルの特定のセットに準拠するファイルの集合、およびこれらのファイルを管理する一連のプログラム。

固定ファイル属性 (* fixed file attributes)

ファイルに関する情報であり、ファイルの作成時に設定され、それ以降はファイルが存在する限り変更できない。これらの属性には、ファイルの編成 (順次、相対、指標付き)、基本レコード・キー、代替レコード・キー、コード・セット、最小および最大のレコード・サイズ、レコード・タイプ (固定長、可変長)、索引付きファイルのキーの照合シーケンス、ブロック化因数、埋め込み文字、レコード区切り文字がある。

固定長レコード (* fixed-length record)

ファイル記述項目またはソート・マージ記述項目が、すべてのレコードのバイトの個数が同じであるように要求しているファイルに関連付けられたレコード。

固定小数点項目 (fixed-point item)

PICTURE 節で定義される数値データ項目であり、オプションの符号の位置、その中に含まれる桁数、およびオプションの小数点の位置を指定するもの。2 進数、パック 10 進数、または外部 10 進数のいずれかのフォーマットをとることができる。

フローティングコメント標識 (*>)

浮動コメント標識がプログラムのテキスト域 (領域 A プラス領域 B) 内の最初の文字ストリングである場合は、この行がコメント行であることを示します。また、浮動コメント標識がプログラムのテキスト領域内の 1 つ以上の文字ストリングの後にある場合は、インライン・コメントを示します。

FLOATING POINT

実数を 1 対の数表示で表す、数を表記するための形式。浮動小数点表記では、固定小数点部分 (最初の数表示) と、暗黙浮動小数点の底を指数で表される数だけ累乗して得られる値 (2 番目の数表示) との積が、実数になります。例えば、数値 0.0001234 の浮動小数点表記は 0.1234 -3 です (ここで、0.1234 は小数部であり、-3 は指数です)。

浮動小数点データ項目 (floating-point data item)

小数部と指数が入っている数値データ項目。その値は、小数部に、指数で指定されたただけ累乗された数字データ項目の底を乗算することによって得られる。

フォーマット (* format)

データの集合の特定の配列。

関数 (* function)

ステートメントの実行中に参照された時点で決定される値を持つ、一時的なデータ項目。

関数 ID (* function-identifier)

関数を参照する文字ストリングと分離文字の構文的に正しい組み合わせ。関数で表現されるデータ項目は、関数名と引数 (ある場合) によって一意的に識別される。関数 ID は、参照修飾子を含むことができる。英数字関数を参照する関数 ID は、一定の制限に従いつつ ID が指定できる一般フォーマットの中ならばどこにでも指定できる。整数関数または数字関数を参照する関数 ID は、算術式が指定できる一般フォーマットの中ならばどこにおいても指定できる。

機能名 (function-name)

必要な引数を伴って関数の値を決定する呼び出しを行うメカニズムに付けられる名前を表すワード。

関数ポインター・データ項目 (function-pointer data item)

入り口点を指すポインターを保管できるデータ項目。USAGE IS FUNCTION-POINTER 節で定義されるデータ項目に、関数入り口点のアドレスが含まれる。一般的に、C および Java プログラムと通信するために使用される。

g

ガーベッジ・コレクション (garbage collection)

参照されなくなったオブジェクト用のメモリーが Java ランタイム・システムによって自動的に解放されること。

グローバル名 (* global name)

1 つのプログラムにおいてのみ宣言されるが、そのプログラム、またはそのプログラム内に含まれている任意のプログラムから参照できる名前。条件名、データ名、ファイル名、レコード名、報告書名、およびいくつかの特殊レジスターが、グローバル名となり得る。

グローバル参照

メソッドの有効範囲外にあるオブジェクトの参照。

グループ項目 (group item)

(1) 複数の従属データ項目で構成されるデータ項目。英数字グループ項目 (*alphanumeric group item*) および 国別グループ項目 (*national group item*) を参照。(2) 国別グループまたは英数字グループとして明示的に (またはコンテキストで) 限定されていない場合、この用語は一般のグループを指します。

グループ区切り文字 (grouping separator)

読みやすさのために数値を何桁かまとめて区切るのに使用される文字。デフォルトの文字はコンマです。

時間

ヘッダー・ラベル (header label)

(1) 記録メディア・ユニットのデータ・レコードの前にある、データ・セットのラベル。(2) 「ファイル開始ラベル (*beginning-of-file label*)」の同義語。

非表示にする(方法)

親クラスで同じメソッド名により定義されたファクトリーまたは静的メソッドを (サブクラスで) 再定義すること。したがって、サブクラスのメソッドは親クラスのメソッドを隠蔽する。

* 上位の終わり

文字ストリングの左端の文字。

ハイパースペース

z/OS 環境で、プログラムがバッファとして使用できる最大 2 GB までの連続する仮想記憶アドレス範囲。

I

IBM COBOL 拡張部分 (IBM COBOL extension)

85 COBOL 標準に記述されているものの他に、IBM コンパイラーでサポートされる COBOL 構文およびセマンティクス。

IDENTIFICATION DIVISION

COBOL プログラム、クラス定義、またはメソッド定義の 4 つの主コンポーネントの 1 つ。

IDENTIFICATION DIVISION では、プログラム、クラス、メソッドを識別する。IDENTIFICATION DIVISION には、作成者名、インストール、日付を含めることができる。

ID (* identifier)

データ項目に名前を付けるための文字ストリングと分離文字の構文的に正しい組み合わせ。関数ではないデータ項目を参照するときは、ID は、データ名と、修飾子、添え字、または参照修飾子 (一意的に参照するために必要な場合) から構成される。関数であるデータ項目を参照する際には、関数 ID が使われる。

IGZCBSN

COBOL/370 1.1 のブートストラップ・ルーチン。これは、COBOL/370 1.1 プログラムを含むすべてのモジュールとリンク・エディットする必要がある。

IGZCBSO

COBOL for MVS & VM 1.2、COBOL for OS/390 & VM、Enterprise COBOL のブートストラップ・ルーチン。COBOL for MVS & VM 1.2、COBOL for OS/390 & VM、Enterprise COBOL プログラムを含むすべてのモジュールとリンク・エディットする必要がある。

IGZEBST

VS COBOL II のブートストラップ・ルーチン。これは、VS COBOL II リリース 1 プログラムを含んでいないモジュールとリンク・エディットしなければならない。

ILC

言語間通信。言語間通信は、他の高水準言語を呼び出すかまたは他の高水準言語によって呼び出されるプログラムとして定義されています。アセンブラーは高水準言語と見なされません。このため、アセンブラー言語プログラムへの呼び出しおよびアセンブラー言語プログラムからの呼び出しは ILC と見なされません。

命令ステートメント (* imperative statement)

命令の動詞で開始して、行うべき無条件の処置を指定するステートメント。または明示範囲終了符号によって区切られた条件ステートメント (範囲区切りステートメント)。1 つの命令ステートメントは、一連の命令ステートメントから構成することができる。

暗黙の範囲終了符号 (* implicit scope terminator)

終了していないステートメントが前にある場合、その範囲を区切る分離文字ピリオド。または、前にある句の中に含まれるステートメントがある場合、そのステートメントの範囲の終わりをそれが現れることによって示すステートメントの句。

IMS

情報管理システム (Information Management System)。IBM のライセンス製品。IMS は、階層データベース、データ通信 (DC)、変換処理、およびデータベースのバックアウトとリカバリーをサポートする。

索引 (* index)

コンピューターのストレージ域またはレジスター。ここにはテーブル内の個々のエレメントを識別するものを表現する内容が入る。

指標データ項目 (* index data item)

指標名に関連する値を、インプリメントする人が指定した形式で収めることができるデータ項目。

索引付きデータ名 (indexed data-name)

データ名とそれに続く括弧で囲まれた 1 つまたは複数の指標名から構成される ID。

索引付きファイル (* indexed file)

指標付き編成のファイル。

指標付き編成 (* indexed organization)

各レコードがそのレコードの中にある 1 つまたは複数のキー値によって識別される永続的な論理ファイル構造。

指標付け (indexing)

指標名を使用した添え字付け と同義。

索引名 (* index-name)

特定のテーブルに関連付けられた指標に付ける名前を表すユーザー定義語。

継承 (inheritance)

クラスのインプリメンテーションを、別のクラスを基にして使用するメカニズム。定義により、継承するクラスは継承されるクラスに準拠する。Enterprise COBOL は 多重継承 をサポートしない。サブクラスは、必ず 1 つの即時スーパークラスを有する。

継承階層 (inheritance hierarchy)

クラス階層 (class hierarchy) を参照。

初期設定プログラム (* initial program)

実行単位内にプログラムが呼び出されるたびに初期状態に置かれるプログラム。

初期状態 (* initial state)

プログラムが実行単位の中に呼び出された最初期のそのプログラムの状態。

インライン (inline)

ルーチン、サブルーチン、または他のプログラムに分岐せずに、連続的に実行されるプログラム内の命令。

インライン・コメント (inline comments)

インライン・コメントは、プログラムのテキスト域にある、前に 1 つ以上の文字ストリングが付いた浮動コメント標識 (*>) で示され、コンパイル・グループの任意の行に書き込むことができます。浮動コメント標識に続く、領域 B の終わりまでの文字はすべて、コメント・テキストです。

入力ファイル (* input file)

入力モードでオープンされるファイル。

入力モード (* input mode)

ファイルに対する INPUT 句の指定のある OPEN ステートメントが実行されてから、そのファイルに対する REEL 句または UNIT 句の指定のない CLOSE ステートメントが実行される前までの、ファイルの状態。

入出力ファイル (* input-output file)

I-O モードでオープンされるファイル。

*** INPUT-OUTPUT SECTION**

ENVIRONMENT DIVISION のセクションであり、オブジェクト・プログラムまたはメソッドに必要なファイルおよび外部メディアに名前を付け、実行時にデータの伝送および処理に必要な情報を提供する。

入出力ステートメント (* input-output statement)

個々のレコードに対して操作を行うことにより、またはファイルを 1 つの単位として操作することにより、ファイルの処理を行うステートメント。入出力ステートメントには、ACCEPT (ID 句付き)、CLOSE、DELETE、DISPLAY、OPEN、READ、REWRITE、SET (TO ON 句または TO OFF 句付き)、START、WRITE がある。

入力プロシージャ (* input procedure)

ソート対象として指定されたレコードの解放を制御するために、形式 1 SORT ステートメントの実行中に制御が渡されるステートメントのセット。

インスタンス・データ (instance data)

オブジェクトの状態を定義するデータ。クラスによって導入されるインスタンス・データは、クラス定義の OBJECT 段落の DATA DIVISION の WORKING-STORAGE SECTION に定義される。オブジェクトの状態には、クラスが導入した、現行クラスによって継承されているインスタンス変数の状態も含まれる。インスタンス・データの個々のコピーは、各オブジェクト・インスタンスごとに作成される。

整数 (* integer)

(1) 小数点の右側に桁位置がない数値リテラル。(2) DATA DIVISION に定義される数値データ項目であり、小数点の右側に桁位置を含まないもの。(3) 関数の起こりうるすべての評価の戻り値で、小数点の右側の桁がすべてゼロであることが定義されている数字関数。

整数関数 (integer function)

カテゴリーが数字で、その定義が小数点の右側に桁位置を持たない関数。

対話式システム生産性向上機能 (ISPF) (Interactive System Productivity Facility (ISPF))

TSO または VM ユーザーに対してメニュー方式のインターフェースを提供する IBM ソフトウェア・プロダクト。ISPF には、ライブラリー・ユーティリティ、強力なエディター、およびダイアログ管理が組み込まれています。

言語間通信 (ILC) (interlanguage communication (ILC))

異なるプログラム言語で書かれた複数のルーチンが通信できること。ILC サポートにより、各種言語で書かれたコンポーネント・ルーチンからアプリケーションを簡単に構築することができる。

中間結果 (intermediate result)

連続して行われる算術演算の結果を収める中間フィールド。

内部データ (* internal data)

プログラムの中で記述されるデータで、すべての外部データ項目および外部ファイル結合子を除いたもの。プログラムの LINKAGE SECTION で記述された項目は、内部データとして扱われる。

内部データ項目 (* internal data item)

実行単位内の 1 つのプログラムの中で記述されるデータ項目。内部データ項目は、グローバル名を持つことができる。

内部 10 進数データ項目 (internal decimal data item)

USAGE PACKED-DECIMAL または USAGE COMP-3 として記述され、項目を数値として定義する PICTURE 文字ストリング (9、S、P、または V の有効な記号の組み合わせ) を持っている、データ項目。「パック 10 進数データ項目 (packed-decimal data item)」と同義。

内部ファイル結合子 (* internal file connector)

実行単位内にあるただ 1 つのオブジェクト・プログラムのみがアクセスできるファイル結合子。

内部浮動小数点データ項目 (internal floating-point data item)

USAGE COMP-1 または USAGE COMP-2 として記述されているデータ項目。COMP-1 は、単精度浮動小数点データ項目を定義する。COMP-2 は、倍精度浮動小数点データ項目を定義する。内部浮動小数点データ項目に関連した PICTURE 節はない。

レコード内データ構造 (* intrarecord data structure)

連続したデータ記述記入項目のサブセットによって定義される、1 つの論理レコードから得られるグループ・データ項目および基本データ項目の集合全体。これらのデータ記述記入項目には、レコード内データ構造を記述している最初のデータ記述記入項目のレベル番号より大きいレベル番号を持つすべての記入項目が含まれる。

組み込み関数 (intrinsic function)

よく使用される算術関数のような事前定義関数で、組み込み関数参照によって呼び出される。

無効キー条件 (* invalid key condition)

索引付きファイルまたは相対ファイルに関連するキーの特定値が無効であると判別された場合に生じる、実行時の条件。

* I-O-CONTROL

ENVIRONMENT DIVISION 段落の名前。この段落では、再実行開始点についてのオブジェクト・プログラム要件、複数データ・ファイルによる同じ区域の共用、および単一入出力装置上の複数のファイル・ストレージが指定される。

I-O-CONTROL 記入項目 (* entry)

ENVIRONMENT DIVISION の I-O-CONTROL 段落内の記入項目であり、プログラム実行中に指定のファイルへのデータの伝送と処理を行うために必要な情報を提供する節が入っている。

入出力モード (* I-O mode)

ファイルに対する I-O 句の指定のある OPEN ステートメントが実行されてから、そのファイルに対する REEL 句または UNIT 句の指定のない CLOSE ステートメントが実行される前までの、ファイルの状態。

入出力状況 (* I-O status)

入出力操作の結果としての状況を示す 2 文字の値を収める概念上のエンティティ。この値は、そのファイルについてのファイル制御項目で FILE STATUS 節を使用することにより、プログラムで使用可能になる。

is-a

継承階層におけるクラスおよびサブクラスの特徴を表す関係。あるクラスに対して is-a 関係を持つサブクラスは、そのクラスから継承する。

ISPF

対話式システム生産性向上機能 (ISPF) (*Interactive System Productivity Facility (ISPF)*) を参照。

反復構造 (iteration structure)

ある条件が真である間、あるいはある条件が真になるまで、一連のステートメントが繰り返して実行されるプログラムの処理ロジック。

j

J2EE

Java 2 Platform, Enterprise Edition (J2EE) を参照。

Java 2 Platform, Enterprise Edition (J2EE)

エンタープライズ・アプリケーションの開発とデプロイメントのために、Oracle によって定義された環境。J2EE プラットフォームは、多層の Web ベース・アプリケーションを開発するための機能を提供するサービス、アプリケーション・プログラミング・インターフェース (API)、およびプロトコルで構成されている。(Oracle)

Java Batch Launcher and Toolkit for z/OS (JZOS)

従来のバッチ環境で実行されて z/OS システム・サービスにアクセスする z/OS Java アプリケーションの開発を支援するツールのセット。

Java バッチ処理プログラム (JBP) (Java batch-processing program (JBP))

オンライン・データベースおよび出力メッセージ・キューにアクセスできる IMS バッチ処理プログラム。JBP はオンラインで実行されますが、バッチ環境のプログラムと同様に、JCL を使用して、または TSO セッション内で開始されます。

Java バッチ処理領域 (Java batch-processing region)

Java バッチ処理プログラムだけがスケジュールされる IMS 従属領域。

Java Database Connectivity (JDBC)

Java プログラムのデータベースへのアクセスを可能にする API を定義する、Oracle の仕様。

Java メッセージ処理プログラム (JMP) (Java message-processing program (JMP))

トランザクションによって駆動され、オンライン IMS データベースとメッセージ・キューにアクセスできる、Java アプリケーション・プログラム。

Java メッセージ処理領域 (Java message-processing region)

Java メッセージ処理プログラムだけがスケジュールされる IMS 従属領域。

Java Native Interface (JNI)

Java 仮想マシン (JVM) 内で実行される Java コードが、他のプログラム言語で記述されたアプリケーションおよびライブラリーと連携できるようにするプログラミング・インターフェース。

Java 仮想マシン (JVM) (Java virtual machine (JVM))

コンパイル済みの Java プログラムを実行する中央演算処理装置のソフトウェア・インプリメンテーション。

JavaBeans

移植可能で、プラットフォームに依存しない、再使用可能なコンポーネント・モデル。(Oracle)

JBP

Java バッチ処理プログラム (JBP) (Java batch-processing program (JBP)) を参照。

JDBC

Java Database Connectivity (JDBC) を参照。

エビ

Java メッセージ処理プログラム (JMP) (Java message-processing program (JMP)) を参照。

ジョブ制御言語 (JCL) (job control language (JCL))

ジョブをオペレーティング・システムに識別させ、ジョブの要件を記述するために使われる制御言語。

JSON

JSON (JavaScript Object Notation) とは、単純なデータ交換フォーマットである。

JVM

Java 仮想マシン (JVM) (Java virtual machine (JVM)) を参照。

JZOS

Java Batch Launcher と Toolkit for z/OS を参照。

K

K

記憶容量に関連して使用される場合は、2 の 10 乗。10 進表記では 1024。

キー (* key)

レコードの位置を識別するデータ項目、またはデータの順序付けを識別するための一連のデータ項目。

参照キー (* key of reference)

索引付きファイルの中のレコードをアクセスするために現在使用されている基本キーまたは代替キー。

キーワード (* keyword)

コンテキスト・センシティブ語または予約語。その語の表示フォーマットがソース単位で使用されるときは、その語は必須である。

キロバイト (KB) (kilobyte (KB))

1 キロバイトは 1024 バイトに相当する。

L

言語名 (* language-name)

特定のプログラミング言語を指定するシステム名。

言語環境プログラム

z/OS 言語環境プログラムの省略名。C、C++、COBOL、FORTRAN、および PL/I アプリケーションに共通のランタイム環境およびランタイム・サービスを提供する一連のアーキテクチャ構造およびインターフェース。これは、言語環境プログラム準拠のコンパイラーおよびジャワ アプリケーションへ別にコンパイルされたプログラムに必要です。

言語環境プログラム-準拠

言語環境プログラム 規則に準拠するオブジェクト・コードを生成するコンパイラー製品 (Enterprise COBOL、COBOL OS/390 & VM、COBOL for MVS & VM、C/C++ for MVS & VM、PL/I for MVS & VM など) の特性。

最後に使われた状態 (last-used state)

内部値がプログラム終了時と同じままで、初期値にリセットされない、プログラムの状態を言う。

文字 (* letter)

以下の 2 つのセットのいずれかに属する文字。

1. 英大文字: A、B、C、D、E、F、G、H、I、J、K、L、M、N、O、P、Q、R、S、T、U、V、W、X、Y、Z
2. 英小文字: a、b、c、d、e、f、g、h、i、j、k、l、m、n、o、p、q、r、s、t、u、v、w、x、y、z

レベル標識 (* level indicator)

特定のタイプのファイルを識別するか、または階層での位置を識別する 2 つの英字。DATA DIVISION のレベル標識には、CD、FD、SD がある。

レベル番号 (* level-number)

階層構造におけるデータ項目の位置を示すか、またはデータ記述記入項目の特性を示す、2桁の数字で表されたユーザー定義語。1から49までの範囲のレベル番号は、論理レコードの階層構造におけるデータ項目の位置を示す。1から9のレベル番号は、1桁の数字として書き込むことも、0の後に有効数字を書き込むこともできる。レベル番号66、77、および88は、データ記述項目の特性を識別する。

ライブラリー名 (* library-name)

COBOL ライブラリーの名前を表すユーザー定義語。与えられたソース・プログラムをコンパイルするためにコンパイラーが使用するライブラリーを識別する。

ライブラリー・テキスト (* library text)

COBOL ライブラリーの中にある一連のテキスト・ワード、コメント行、インライン・コメント、区切り文字のスペース、または区切り文字の疑似テキスト区切り文字。

リリアン日 (Lilian date)

グレゴリオ暦の開始以降の日数。第1日は1582年10月15日、金曜日。リリアン日フォーマットは、グレゴリオ暦の考案者であるルイジ・リリオにちなんだ名称。

* LINAGE-COUNTER

ページ本体内の現在位置を指す値を収めた特殊レジスター。

リンク (link)

(1) リンク接続 (伝送メディア) と、それぞれがリンク接続の終端にある2つのリンク・ステーションの組み合わせ。1つのリンクは、マルチポイントまたはトークンリング構成において、複数のリンク間で共用できる。(2) データ項目あるいは1つ以上のコンピューター・プログラムの部分を相互接続すること。例えば、リンケージ・エディターによってオブジェクト・プログラムをリンクして実行可能ファイルを作成すること。

LINKAGE SECTION

呼び出し先のプログラムまたはメソッドの DATA DIVISION 内のセクションであり、呼び出し側プログラムまたはメソッドから使用可能なデータ項目が記述される。これらのデータ項目は、呼び出し側プログラムまたはメソッドおよび呼び出し先プログラムまたはメソッドの両方から参照できる。

リンカー (linker)

z/OS バインダー (リンク・エディター)のいずれかを参照する用語。

literal

ストリングを構成するために配列された文字によって、または形象定数を使用することによって、その値が決められる文字ストリング。

リトル・エンディアン (little-endian)

Intel プロセッサが2進データおよび UTF-16 文字を保管するために使用するデフォルト形式。この形式では、2進数データ項目の最上位バイトが最上位のアドレスになり、UTF-16 文字の最上位バイトが最上位のアドレスになる。ビッグ・エンディアン (*big-endian*) と比較。

ローカル参照

メソッドの有効範囲内にあるオブジェクトの参照。

LOCALE

プログラム実行環境の一連の属性であり、文化的に重要な考慮事項を示す。例えば、文字コード・ページ、照合シーケンス、日時形式、通貨表記、数値表記、または言語など。

* LOCAL-STORAGE SECTION

DATA DIVISION のセクションであり、VALUE 節で割り当てられた値に応じて、呼び出し単位で割り振りまたは解放が行われるストレージを定義する。

論理演算子 (* logical operator)

予約語 AND、OR、または NOT のいずれか。条件の形成において、AND または OR、あるいはその両方を論理連結語として使用できる。NOT は論理否定に使用できる。

論理レコード (* logical record)

最も包括的なデータ項目。レコードのレベル番号は01。レコードは、基本項目またはグループ項目のどちらでもよい。「レコード (*record*)」と同義。

下位終了 (* low-order end)

文字ストリングの右端の文字。

m

メインプログラム (main program)

プログラムとサブルーチンからなる階層において、プロセス内でプログラムが実行されたときに最初に制御を受け取るプログラム。

Make ファイル (makefile)

アプリケーションに必要なファイルのリストが収められたテキスト・ファイル。make ユーティリティはこのファイルを使用して、ターゲット・ファイルを最新の変更で更新する。

大容量記憶 (* mass storage)

データを順次と非順次の2つの方法で編成して保管しておくことができるストレージ・メディア。

大容量記憶装置 (* mass storage device)

磁気ディスクなど、大きな記憶容量を持つ装置。

大容量記憶ファイル (* mass storage file)

大容量記憶メディアに格納されたレコードの集合。

メガバイト、MB (* megabyte (MB))

1メガバイトは1,048,576 バイトに相当する。

マージ・ファイル (* merge file)

MERGE ステートメントによってマージされるレコードの集合。マージ・ファイルは、マージ機能により作成され、マージ機能によってのみ使用できる。

メッセージ処理プログラム (MPP) (message-processing program (MPP))

トランザクションによって駆動され、オンラインIMS データベースとメッセージ・キューにアクセスできる、IMS アプリケーション・プログラム。

メッセージキュー

メッセージがアプリケーション・プログラムによって処理されたり端末に送信されたりする前に、キューに入れられるデータ・セット。

メソッド

オブジェクトによってサポートされる操作の1つを定義し、そのオブジェクトに対する INVOKE ステートメントによって実行されるプロシージャ・コード。

メソッド定義 (* method definition)

メソッドを定義する COBOL ソース・コード。

メソッド見出し記入項目 (* method identification entry)

IDENTIFICATION DIVISION の METHOD-ID 段落内の記入項目。この記入項目には、メソッド名を指定する文節が入っている。

メソッドの起動 (method invocation)

あるオブジェクトから別のオブジェクトへの通信で、受信オブジェクトにメソッドを実行するように要求するもの。

メソッド名 (method-name)

オブジェクト指向操作の名前。メソッドを起動するのに使用する場合、名前は、英数字リテラル、国別リテラル、カテゴリー英数字データ項目、またはカテゴリー国別データ項目にすることができます。メソッドを定義する METHOD-ID 段落で使用する場合、名前は英数字リテラルまたは国別リテラルにする必要がある。

メソッド隠蔽 (method hiding)

「隠蔽 (hide)」を参照。

メソッド多重定義 (method overloading)

「多重定義 (overload)」を参照。

メソッドのオーバーライド (method overriding)

「オーバーライド (override)」を参照。

簡略名 (* mnemonic-name)

ENVIRONMENT DIVISION において、指定されたインプリメントする人の名前に関連したユーザー定義語。

モジュール定義ファイル (module definition file)

プログラム・オブジェクト内のコード・セグメントを記述するファイル。

MPP

メッセージ処理プログラム (MPP) (*message-processing program (MPP)*) を参照。

マルチタスキング (multitasking)

2 つ以上のタスクの並行実行またはインターリーブ実行を可能にする操作モード。

マルチスレッド化 (multithreading)

コンピューター内で複数のパスを使用して実行を行う並行操作。「マルチプロセッシング (*multiprocessing*)」と同義。

N

名前 (name)

COBOL オペランドを定義する 30 文字を超えないで構成されたワード。

namespace

XML 名前空間 (*XML namespace*) を参照。

国別文字 (national character)

(1) USAGE NATIONAL データ項目または国別リテラル内の UTF-16 文字。(2) UTF-16 で表される任意の文字。

国別文字データ (national character data)

UTF-16 で表されるデータの一般参照。

国別文字位置 (national character position)

文字位置 (*character position*) を参照。

国別データ (national data)

「国別文字データ (*national character data*)」を参照。

国別データ項目 (national data item)

USAGE NATIONAL のカテゴリ国別、国別編集、数字編集のデータ項目。

国別 10 進数データ項目 (national decimal data item)

暗黙的または明示的に USAGE NATIONAL として記述され、PICTURE の記号 9、S、P、V の有効な組み合わせを含んでいる、外部 10 進数データ項目。

国別編集データ項目 (national-edited data item)

記号 N の少なくとも 1 つのインスタンスと、単純挿入記号 B、0、または / の少なくとも 1 つを含む PICTURE 文字ストリングによって記述されているデータ項目。国別編集データ項目には USAGE NATIONAL がある。

国別浮動小数点データ項目 (national floating-point data item)

暗黙的または明示的に USAGE NATIONAL として記述され、浮動小数点データ項目を記述する PICTURE 文字ストリングを持つ外部浮動小数点データ項目。

国別グループ項目 (national group item)

明示的または暗黙的に GROUP-USAGE NATIONAL 節で記述されたグループ項目。国別グループ項目は、INSPECT、STRING、UNSTRING などの操作で、国別カテゴリの基本データ項目として定義されているかのように処理される。英数字グループ項目内の USAGE NATIONAL データ項目の定義とは対照的に、この処理により、国別文字の埋め込みと切り捨てが確実に正しく行われる。MOVE CORRESPONDING、ADD CORRESPONDING、INITIALIZE など、グループ内の基本項目の処理を必要とする操作の場合、国別グループはグループ・セマンティクスを使用して処理される。

固有文字セット (* native character set)

OBJECT-COMPUTER 段落で指定されたコンピューターに関連した、インプリメントする人が定義した文字セット。

固有照合シーケンス (* native collating sequence)

OBJECT-COMPUTER 段落で指定されたコンピューターに関連した、インプリメントする人が定義した照合シーケンス。

ネイティブ・メソッド

COBOL などの別のプログラム言語で記述されたインプリメンテーションを備える Java メソッド。

複合否定条件 (* negated combined condition)

論理演算子 NOT とその直後に括弧で囲んだ複合条件を続けたもの。条件 (condition) および 複合条件 (combined condition) も参照。

単純否定条件 (* negated simple condition)

論理演算子 NOT とその直後に単純条件を続けたもの。条件 (condition) および 単純条件 (simple condition) も参照。

ネストされたプログラム (nested program)

他のプログラムの中に直接的に含まれているプログラム。

次の実行可能文 (* next executable sentence)

現在のステートメントの実行完了後に制御が移される次の文。

次の実行可能なステートメント (* next executable statement)

現在のステートメントの実行完了後に制御が移される次のステートメント。

次のレコード (* next record)

ファイルの現行レコードに論理的に続くレコード。

独立項目 (* noncontiguous items)

他のデータ項目と階層上の関係を持たない、WORKING-STORAGE SECTION および LINKAGE SECTION 内の基本データ項目。

独立項目 (* noncontiguous items)

別のデータ項目への階層関係がない、WORKING-STORAGE および LINKAGE SECTION の基本データ項目。

非数字項目 (* nonnumeric item)

その内容を、コンピューターの文字セットからの文字の任意の組み合わせで構成して記述することができるデータ項目。特定のカテゴリーの非数字項目は、さらに制限された文字セットから形成することができる。

null

無効なアドレスの値をポインター・データ項目に割り当てるために使用される形象定数。NULLS は、NULL を使用できる場所であればどこでも使用できる。

数字 (* numeric character)

次のような数字に属する文字。0、1、2、3、4、5、6、7、8、9。

数値データ項目 (numeric data item)

(1) 詳細がその中身を、0 から 9 の数字～別に選択された文字で表される量に制限するデータ・アイテム。符号付きの場合、項目には演算符号の +、-、または他の表記の演算符号も含むことができる。(2) カテゴリー数値、内部浮動小数点、または外部浮動小数点のデータ項目。数値データ項目は、USAGE DISPLAY、NATIONAL、PACKED-DECIMAL、BINARY、COMP、COMP-1、COMP-2、COMP-3、COMP-4、COMP-5 を持つことができる。

数字編集データ項目 (numeric-edited data item)

印刷出力の際に使用するのに適したフォーマットの数値データを含むデータ項目。データ項目は、外部 10 進数字の 0 から 9 の数字、小数点、コンマ、通貨符号、符号制御文字、その他の編集記号から構成される。数字編集項目は、USAGE DISPLAY または USAGE NATIONAL のいずれかで表すことができる。

数字関数 (* numeric function)

クラスとカテゴリーは数字だが、考えられる評価のいくつかにおいて整数関数の要件を満たさないような関数。

数値項目 (* numeric item)

その内容の記述が、「0」から「9」までの数字から選択された文字で表される値に制限されるデータ項目。符号付きの場合、その項目には +、-、または他の演算符号の表記を入れることもできる。

数値リテラル (* numeric literal)

1 つ以上の数字から構成されるリテラルで、小数点または代数符号あるいはその両方を含むことができる。小数点は右端の文字であってはならない。代数符号がある場合には、それが左端の文字でなければならない。

o

Object

状態 (そのデータ値) および演算 (そのメソッド) を持つエンティティ。オブジェクトは状態と動作をカプセル化する手段である。クラス内の各オブジェクトは、そのクラスの1つのインスタンスであると言われる。

オブジェクト・コード (object code)

コンパイラまたはアセンブラからの出力。それ自体が実行可能なマシン・コードか、またはその種のコードの作成を目的としての処理に適する。

* OBJECT-COMPUTER

オブジェクト・プログラムが実行されるコンピューター環境が記述されている ENVIRONMENT DIVISION にある段落の名前。

オブジェクト・コンピューター記入項目 (* object computer entry)

ENVIRONMENT DIVISION の OBJECT-COMPUTER 段落内の記入項目。この記入項目には、オブジェクト・プログラムが実行されるコンピューター環境を記述する節が入っている。

オブジェクト・デッキ (object deck)

リンケージ・エディターへの入力として適切なオブジェクト・プログラムの部分。「オブジェクト・モジュール (object module)」および「テキスト・デッキ (text deck)」と同義。

オブジェクト・インスタンス (object instance)

単一の、場合によっては多数のオブジェクト。COBOL クラス定義の Object 段落における指定に基づいてインスタンス生成される。オブジェクト・インスタンスは、そのクラス定義に記述されたデータおよびすべての継承データのコピーを保持する。オブジェクト・インスタンスに関連付けられたメソッドには、そのクラス定義で定義されたメソッドおよびすべての継承メソッドが含まれる。

オブジェクト・インスタンスは Java クラスのインスタンスにすることができる。

オブジェクト・モジュール (object module)

オブジェクト・デッキ (object deck) または テキスト・デッキ (text deck) と同義。

項目のオブジェクト (* object of entry)

COBOL プログラムの DATA DIVISION 記入項目内の一連のオペランドと予約語であり、その記入項目のサブジェクトの直後に続く。

オブジェクト指向プログラミング (object-oriented programming)

カプセル化および継承の概念に基づいたプログラミング・アプローチ。プロシージャ型プログラミング技法とは異なり、オブジェクト指向プログラミングでは、何かが達成される方法ではなく、問題を含むデータ・オブジェクトとその操作方法に重点を置く。

オブジェクト・プログラム (object program)

問題を解決するためにデータと相互に作用することを目的とする実行可能なマシン言語命令とその他の要素の集合またはグループ。このコンテキストでは、オブジェクト・プログラムとは一般に、COBOL コンパイラがソース・プログラムまたはクラス定義を操作した結果得られるマシン言語である。あいまいになる危険がない場合には、オブジェクト・プログラム という用語の代わりに プログラム というワードだけが使用される。

オブジェクト・リファレンス (object reference)

クラスのインスタンスを識別する値。クラスが指定されなかった場合、オブジェクト参照は一般的なものとなり、任意のクラスのインスタンスに適用できる。

オブジェクト時 (*object time)

オブジェクト・プログラムが実行される時。実行時 (run time) と同義。

廃止される言語エレメント (* obsolete element)

2002 COBOL 標準 から削除された 85 COBOL 標準 の COBOL 言語エレメント。

ODO オブジェクト (ODO object)

以下の例では、X は OCCURS DEPENDING ON 節のオブジェクト (ODO オブジェクト) である。

```
WORKING-STORAGE SECTION.  
01 TABLE-1.  
   05 X PIC S9.  
   05 Y OCCURS 3 TIMES  
      DEPENDING ON X PIC X.
```

ODO オブジェクトの値によって、テーブル内の ODO サブジェクトの数が決まる。

ODO 対象 (ODO subject)

上記の例では、Y は OCCURS DEPENDING ON 節の対象 (ODO 対象) である。テーブル内の ODO 対象の数である Y の値は、X の値によって異なる。

オープン・モード (* open mode)

OPEN ステートメントが実行されてから、REEL 句または UNIT 句の指定のない CLOSE ステートメントが実行される前までのファイルの状態。個々のオープン・モードは、OPEN ステートメントの中で INPUT、OUTPUT、I-O、EXTEND のいずれかとして指定する。

オペランド (* operand)

(1) オペランドの一般的な定義は、"操作対象のコンポーネント" である。(2) 本書の目的に沿った言い方をすれば、ステートメントや記入項目の形式中に現れる小文字または日本語で書かれた語 (または語群) はオペランドと見なされ、そのオペランドによって指示されたデータに対して暗黙の参照を行う。

Operation

オブジェクトに関して要求できるサービス。

演算符号 (* operational sign)

値が正であるか負であるかを示すために数字データ項目または数値リテラルに付けられる代数符号。

オプション・ファイル (optional file)

オブジェクト・プログラムが実行されるたびに必ずしも使用可能でなくてもよいものとして宣言されているファイル。

オプション・ワード (* optional word)

言語を読みやすくする目的でのみ特定の形式で含められる予約語。このようなワードが表示されている形式をソース単位内で使用する場合、そのワードの有無はユーザーが選択できる。

出力ファイル (* output file)

出力モードまたは拡張モードのいずれかでオープンされるファイル。

出力モード (* output mode)

OUTPUT 句または EXTEND 句の指定のある OPEN ステートメントが実行されてから、REEL 句または UNIT 句の指定のない CLOSE ステートメントが実行される前までのファイルの状態。

出力プロシージャ (* output procedure)

形式 1 SORT ステートメントの実行中にソート機能が完了した後で制御が渡されるステートメントの集合、または MERGE ステートメントの実行中に、要求があればマージ機能がマージ済みの順序になっているレコードのうち次のレコードを選択できるようになった後で制御が渡されるステートメントの集合。

オーバーフロー条件 (overflow condition)

ある演算結果の一部が意図した記憶単位の容量を超えたときに起こる条件。

多重定義 (overload)

同じクラスで使用可能な別のメソッドと同一の名前を使い (ただし、異なるシグニチャーを使用して)、メソッドを定義すること。シグニチャー (signature) も参照。

指定変更 (override)

サブクラスのインスタンス・メソッド (親クラスから継承された) を再定義すること。

P

パッケージ (package)

関連する Java クラスの集まり。個々に、または全体としてインポートすることができる。

パック 10 進数データ項目 (packed-decimal data item)

内部 10 進数データ項目 (internal decimal data item) を参照。

埋め込み文字 (padding character)

物理レコード内の未使用文字位置を埋めるのに使用される英数字または国別文字。

PAGE

データの物理的分離を表す、出力データの垂直分割。分離は、内部論理要件または出力メディアの外部特性、あるいはその両方に基づいて行われる。

ページ本体 (* page body)

行を記述できる、または行送りすることができる (またはその両方ができる) 論理ページの部分。

段落 (* paragraph)

PROCEDURE DIVISION では、段落名の後に分離文字ピリオドが続き、その後に 0 個以上の文が続く。IDENTIFICATION DIVISION および ENVIRONMENT DIVISION では、段落ヘッダーの後に 0 個以上の記入項目が続く。

段落ヘッダー (* paragraph header)

予約語の後に分離文字ピリオドが付いたもので、IDENTIFICATION DIVISION および ENVIRONMENT DIVISION において段落の始まりを示すもの。IDENTIFICATION DIVISION で使用できる段落ヘッダーは、以下のとおり。

```
PROGRAM-ID. (Program IDENTIFICATION
DIVISION)
CLASS-ID. (Class IDENTIFICATION DIVISION)
METHOD-ID. (Method IDENTIFICATION
DIVISION)
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

ENVIRONMENT DIVISION で使用できる段落ヘッダーは、以下のとおり。

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
REPOSITORY. (Program or Class
CONFIGURATION SECTION)
FILE-CONTROL.
I-O-CONTROL.
```

段落名 (* paragraph-name)

PROCEDURE DIVISION 中の段落を識別し開始するユーザー定義語。

parameter

(1) 呼び出し側プログラムと呼び出し先プログラム間で受け渡されるデータ。(2) メソッド呼び出しの USING 句内のデータ・エレメント。引数によって、呼び出されたメソッドが要求された操作を実行するために使用できる追加情報を与える。

永続的再使用可能 Java VM

トランザクション間で JVM をリセットすることによりトランザクション処理用にシリアルに再利用できる JVM。リセット・フェーズでは、JVM が既知の初期状態に復元される。

句 (* phrase)

連続する 1 つ以上の COBOL 文字ストリングを配列したセットで、COBOL プロシージャ・ステートメントまたは COBOL 節の一部を構成する。

物理レコード (* physical record)

ブロック (block) を参照。

ポインター・データ項目 (pointer data item)

アドレス値を保管できるデータ項目。データ項目は、USAGE IS POINTER 節を使用してポインターとして明示的に定義される。ADDRESS OF 特殊レジスタは、ポインター・データ項目として暗黙的に定義されている。ポインター・データ項目は、他のポインター・データ項目と等しいかどうかを比較したり、他のポインター・データ項目に内容を移動することができる。

port

(1) 異なるプラットフォームで実行できるようにコンピューター・プログラムを変更すること。(2) インターネット・プロトコルでは、Transmission Control Protocol (TCP) プロトコルまたは User Datagram Protocol (UDP) プロトコルと高水準のプロトコルまたはアプリケーションの間の特定の論理結合子。ポートはポート番号によって識別される。

可搬性 (portability)

あるアプリケーション・プラットフォームから別のアプリケーション・プラットフォームに、ソース・プログラムに比較的わずかな変更を加えるだけでアプリケーション・プログラムを移行できる能力。

合成済み文字 (precomposed character)

標準分解により複数の Unicode 文字を使用して表すことができる単一の Unicode 文字。合成済み文字は合成文字形式と同じ物理表記を持たない。例えば、Unicode 文字 U+00E4 (ä) は、Unicode 文字 U+0061 + U+0308 (a) (ラテン語小文字 a + 結合発音区別符号) の組み合わせとして表すことができる合成済み文字。通常、合成済み文字は、発音区別符号を持つラテン語文字や他の結合文字を表すために使用される。

事前初期設定 (preinitialization)

プログラム (特に非 COBOL プログラム) からの複数の呼び出しの準備としての COBOL ランタイム環境の初期設定。この環境は、明示的に終了されるまで終了されない。

基本レコード・キー (* prime record key)

索引付きファイルのレコードを固有なものとして識別する内容を持つキー。

優先順位番号 (* priority-number)

セグメンテーションの目的で PROCEDURE DIVISION 内のセクションを分類するユーザー定義語。部門番号は 0 から 9 の文字だけを含めるものとします。部門数値は、1 桁または 2 桁のいずれかで表すことができます。

PRivate

ファクトリー・データまたはインスタンス・データに適用されるため、そのデータを定義するクラスのメソッドだけがアクセス可能である。

プロシージャ (* procedure)

PROCEDURE DIVISION にある、1 つの段落または論理的に連続する段落のグループ、あるいは 1 つのセクションまたは論理的に連続するセクションのグループ。

プロシージャ・ブランチ・ステートメント (* procedure branching statement)

ソース・コードの中にステートメントが書かれている順番どおりに次の実行可能ステートメントに制御の移動をせず、別のステートメントに明示的に制御の移動を引き起こすステートメント。プロシージャ・ブランチ・ステートメントには、ALTER、CALL、EXIT、EXIT PROGRAM、GO TO、MERGE (OUTPUT PROCEDURE 句付き)、PERFORM、SORT (INPUT PROCEDURE 句または OUTPUT PROCEDURE 句付き)、XML PARSE がある。

PROCEDURE DIVISION

問題を解決するための指示を含む COBOL の割り算。

プロシージャ統合 (procedure integration)

COBOL 最適化プログラムの機能の 1 つであり、実行されるプロシージャまたは含まれているプログラムへの呼び出しを単純化する。

PERFORM プロシージャ統合とは、PERFORM ステートメントが、実行されるプロシージャによって置き換えられるプロセスのこと。含まれているプログラムのプロシージャ統合とは、含まれているプログラムへの呼び出しがプログラム・コードによって置き換えられるプロセスのこと。

プロシージャ名 (* procedure-name)

PROCEDURE DIVISION の中にある段落またはセクションに名前を付けるために使用される ユーザー定義語。プロシージャ名は、段落名 (これは修飾することができる) またはセクション名から構成される。

プロシージャ・ポインター (procedure pointer)

入り口点を指すポインターを保管できるデータ項目。USAGE IS PROCEDURE-POINTER 節を付けて定義したデータ項目が、プロシージャへの入り口点のアドレスを収める。

プロシージャ・ポインター・データ項目 (procedure-pointer data item)

入り口点を指すポインターを保管できるデータ項目。USAGE IS PROCEDURE-POINTER 節で定義されるデータ項目には、プロシージャ入り口点のアドレスが入っている。通常、COBOL および 言語環境プログラムプログラムとの通信に使用されます

処理

プログラムの全部または一部の実行中に発生する一連のイベント。複数のプロセスを並行して実行することができ、1 つのプロセス内で実行されるプログラムはリソースを共用することができる。

Program

(1) コンピューターによる処理に適した一連の命令。処理には、コンパイラーを使用してプログラムの実行準備をすることやランタイム環境を使用してプログラムを実行することが含まれます。(2) 1つ以上の相互に関係のあるモジュールの論理アセンブリー。同じプログラムの複数のコピーを異なるプロセスで実行することができます。

プログラム名

IDENTIFICATION DIVISION とプログラム終了マーカーにおいて、COBOL ソース・プログラムを識別するユーザー定義語または英数字リテラル。

プログラム識別記入項目 (* program identification entry)

IDENTIFICATION DIVISION の PROGRAM-ID 段落内の記入項目であり、プログラム名を指定し、選択されたプログラム属性をプログラムに割り当てる節が入っている。

プログラム名

IDENTIFICATION DIVISION およびプログラム終了マーカーにおいて、COBOL ソース・プログラムを識別するユーザー定義語または英数字リテラル。

プロジェクト (project)

ダイナミック・リンク・ライブラリー (DLL) や他の実行可能ファイル (EXE) などのターゲットを作成するのに必要な、データおよびアクションの完全セット。

疑似テキスト (* pseudo-text)

ソース・プログラムまたは COBOL ライブラリーにおいて、疑似テキスト区切り文字によって区切られた一連のテキスト・ワード、コメント行、インライン・コメント、または区切り文字スペース (疑似テキスト区切り文字を含まない)。

疑似テキスト区切り文字 (* pseudo-text delimiter)

疑似テキストを区切るために使用される隣接した 2 つの等号文字 (==)。

句読文字 (* punctuation character)

以下のセットに属する文字。

文字	意味
,	コンマ
;	セミコロン
:	コロン
.	ピリオド (終止符)
"	引用符
(左括弧
)	右括弧
	スペース
=	等号

q

QSAM (待機順次アクセス方式) (QSAM (Queued Sequential Access Method))

基本順次アクセス方式 (BSAM) の拡張版。この方式を使用する場合、キューは、処理を待機する入力データ・ブロック、または処理が終了して補助ストレージまたは出力装置への転送を待機する出力データ・ブロックで形成される。

修飾されたデータ名 (* qualified data-name)

データ名と、その後に連結語の OF および IN とデータ名修飾子を続けたものが 1 つ以上のセットで続いて構成される ID。

修飾子 (* qualifier)

(1) レベル標識と関連付けられるデータ名または名前であり、参照の際に、別のデータ名 (修飾子に従属する項目の名前) と一緒に、または条件名と一緒に使用される。(2) セクション名。そのセクションの

中で指定されている段落名と共に参照する際に使用される。(3) ライブラリー名。そのライブラリーと関連付けられたテキスト名と共に参照する際に使用される。

R

ランダム・アクセス (* random access)

キー・データ項目のプログラム指定値を使って、相対ファイルまたは索引付きファイルから取り出したり、削除したり、またはそこに入れたりする論理レコードを識別するアクセス・モード。

レコード (* record)

論理レコード (*logical record*) を参照。

レコード域 (* record area)

DATA DIVISION の FILE SECTION 内のレコード記述項目で記述されるレコードを処理する目的で割り振られるストレージ域。FILE SECTION では、レコード域内の現行の文字位置の数は、明示的または暗黙的な RECORD 節によって決められる。

レコード記述 (* record description)

レコード記述項目 (*record description entry*) を参照。

レコード記述項目 (* record description entry)

特定のレコードに関連したデータ記述項目全体。「レコード記述 (*record description*)」と同義。

記録モード (recording mode)

ファイル内の論理レコードの形式。レコード・モードは、F (固定長)、V (可変長)、S (スパン)、または U (不定フォーマット) とすることができる。

レコード・キー (record key)

索引付きファイル内のレコードを識別する内容を持つキー。

レコード名 (* record-name)

COBOL プログラムの DATA DIVISION 内のレコード記述項目で記述されるレコードに名前を付けるユーザー定義語。

レコード番号 (* record number)

編成が順次であるファイル内のレコードの順序数。

レコード・モード (recording mode)

ファイル内の論理レコードの形式。記録モードは、F (固定長)、V (可変長)、S (スパン)、または U (不定形式) とすることができる。

再帰 (recursion)

それ自体を呼び出すプログラム、または、それ自体で呼び出したプログラムのいずれかによって直接あるいは間接に呼び出されるプログラム。

再起可能 (recursively capable)

RECURSIVE 属性が PROGRAM-ID ステートメントで指定されていれば、プログラムは再帰可能である (再帰的に呼び出すことができる)。

リール (reel)

ストレージ・メディアの個別部分。その大きさはインプリメントする人によって決定され、1つのファイルの一部、1つのファイルの全部、または任意の個数のファイルが収容される。「ユニット (*unit*)」および「ボリューム (*volume*)」と同義。

再入可能 (reentrant)

複数のユーザーがプログラム・オブジェクトの単一の部を共用できるようにするプログラムまたはルーチンの属性。

参照形式 (* reference format)

COBOL ソース・プログラムを記述するに際して標準的な方式を提供する形式。

参照変更 (reference modification)

USAGE DISPLAY、DISPLAY-1、または NATIONAL データ項目の左端文字および左端文字位置を基準にした長さを指定して、新しいカテゴリー英数字、カテゴリー DBCS、またはカテゴリー国別データ項目を定義する方式。

参照修飾子 (* reference-modifier)

固有のデータ項目を定義する文字ストリングと分離文字の構文的に正しい組み合わせ。区切り用の左括弧区切り文字、左端の文字位置、区切り文字のコロン、任意指定の長さ、および区切り用の右括弧区切り文字を含む。

関係 (* relation)

関係演算子 (*relational operator*) または 比較条件 (*relation condition*) を参照。

比較文字 (* relation character)

以下のセットに属する文字。

文字	意味
>	より大きい
<	より小さい
=	に等しい

比較条件 (* relation condition)

ある算術式、データ項目、英数字リテラル、または索引名の値が、他の算術式、データ項目、英数字リテラル、または索引名の 値と特定の関係があるという命題 (それに対して真理値を判別する)。関係演算子 (*relational operator*) も参照。

比較演算子 (* relational operator)

比較条件の構造で使用される、予約語、比較文字、連続する予約語のグループ、または 連続する予約語と比較文字のグループ。使用できる演算子とそれらの意味は次のとおり。

文字	意味
IS GREATER THAN	より大きい
IS >	より大きい
IS NOT GREATER THAN	より大きくない
IS NOT >	より大きくない
IS LESS THAN	より小さい
IS <	より小さい
IS NOT LESS THAN	より小さくない
IS NOT <	より小さくない
IS EQUAL TO	に等しい
IS =	に等しい
IS NOT EQUAL TO	に等しくない
IS NOT =	に等しくない
IS GREATER THAN OR EQUAL TO	より大きいか等しい
IS >=	より大きいか等しい
IS LESS THAN OR EQUAL TO	より小か等しい
IS <=	より小か等しい

相対ファイル (* relative file)

相対編成のファイル。

相対キー (* relative key)

相対ファイルの中の論理レコードを識別するための内容を持つキー。

相対編成 (* relative organization)

各レコードが、レコードのファイル内における論理的順序位置を指定する 0 より大きい整数値によって、固有なものとして識別される永続的な論理ファイル構造。

相対レコード番号 (* relative record number)

相対編成ファイル内でのレコードの序数。この数値は、整数の数値リテラルとして扱われる。

予約語 (* reserved word)

COBOL ソース・プログラムの中で使用することができるが、ユーザー定義語またはシステム名としてプログラムの中で使用されてはならないワードのリスト中に挙げられている COBOL ワード。

リソース (* resource)

オペレーティング・システムの制御下に置かれており、実行中のプログラムによって使用できる機能またはサービス。

結果の ID (* resultant identifier)

算術演算の結果が収められるユーザー定義のデータ項目。

再使用可能環境 (reusable environment)

再使用可能環境は、事前初期設定用の古い COBOL インターフェース (RTEREUS ランタイム・オプション) または 言語環境プログラム インターフェース (CEEPIPI) のどちらかを使用してアセンブラー・プログラムをメインプログラムとして設定すると作成される。

ルーチン (routine)

コンピューターに操作または一連の関連操作を実行させる、COBOL プログラム内の一連のステートメント。言語環境プログラムでは、処置、ファンクション、またはサブルーチンのいずれかを参照します。

ルーチン名 (* routine-name)

COBOL 以外の言語で記述されたプロシージャを識別するユーザー定義語。

実行時 (* run time)

オブジェクト・プログラムが実行される時。「オブジェクト時 (object time)」と同義。

ランタイム環境 (runtime environment)

COBOL プログラムが実行される環境。

実行単位 (* run unit)

1つの独立型オブジェクト・プログラム、あるいは COBOL CALL または INVOKE ステートメントによって相互に作用し、実行時に 1つのエンティティとして機能する複数のオブジェクト・プログラム。

実行単位は、言語環境プログラムの用語では別プログラムとも呼ばれます。

S

SBCS

1 バイト文字セット (SBCS) (single-byte character set (SBCS)) を参照。

範囲終了符号 (scope terminator)

PROCEDURE DIVISION の特定のステートメントの終わりを示す COBOL 予約語。これは明示的なもの (END-ADD など) または暗黙的なもの (分離文字ピリオド) であることもある。

セクション (* section)

ゼロ、1つ、または複数の段落またはエンティティ (セクション本体と呼ばれる) と、その最初のものの前にセクション・ヘッダーが付いているもの。各セクションは、セクション・ヘッダーとそれに関連付けられたセクション本体から構成される。

セクション・ヘッダー (* section header)

後ろに分離文字ピリオドが付いたワードの組み合わせであり、ENVIRONMENT、DATA、または PROCEDURE の各部において、セクションの始まりを示すもの。ENVIRONMENT DIVISION および DATA DIVISION では、セクション・ヘッダーは、予約語の後に分離文字ピリオドを続けたものから構成される。ENVIRONMENT DIVISION で許可されているセクション・ヘッダーは次のとおり。

```
CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.
```

DATA DIVISION で許可されているセクション・ヘッダーは次のとおり。

```
FILE SECTION.  
WORKING-STORAGE SECTION.  
LOCAL-STORAGE SECTION.  
LINKAGE SECTION.
```

PROCEDURE DIVISION では、セクション・ヘッダーは、セクション名、その後続く予約語 SECTION、およびその後の分離文字ピリオドから構成される。

セクション名 (* section-name)

PROCEDURE DIVISION の中にあるセクションに名前を付けるユーザー定義語。

セグメント化 (segmentation)

85 COBOL 標準 分割モジュールに基づく Enterprise COBOL の機能。セグメンテーション機能は、セクション・ヘッダーの優先順位番号を使用して、セクションを固定セグメントまたは独立セグメントに割り当てる。セグメント種別は、セグメントに含まれるプロシージャが初期状態の制御を受け取るか、最後に使われた状態の制御を受け取るかに影響を及ぼす。

選択構造 (selection structure)

条件が真であるか偽であるかに応じて、ある一連のステートメントか、または別の一連のステートメントが実行されるというプログラムの処理ロジック。

文 (* sentence)

1 つ以上のステートメントの並びで、その最後のものは、分離文字ピリオドで終了する。

別々にコンパイルされたプログラム (* separately compiled program)

あるプログラムをそこに含まれたプログラムと共に、他のすべてのプログラムとは別個にコンパイルしたときのそのプログラム。

区切り文字 (* separator)

文字ストリングを区切るために使用される、1 文字または連続する 2 文字以上。

区切り文字のコンマ (* separator comma)

文字ストリングを区切るために使われる、後ろに 1 つのスペースが続く 1 つのコンマ (,)。

分離文字ピリオド (* separator period)

文字ストリングを区切るために使われる、後ろに 1 つのスペースが続く 1 つのピリオド (.)。

区切り文字のセミコロン (* separator semicolon)

文字ストリングを区切るために使われる、後ろに 1 つのスペースが続く 1 つのセミコロン (;)。

プログラム・シーケンス (sequence of programs)

コンパイラーに入力できる単一のソース・ファイルに含まれている一連の個別 COBOL プログラム。

プログラム・シーケンスは、バッチ・コンパイル または コンパイル・グループ とも呼ばれます。

順序構造 (sequence structure)

一連のステートメントが、順序どおりに実行されるプログラムの処理ロジック。

順次アクセス (* sequential access)

ファイル内のレコードの並び方によって規定されている、論理レコードの連続した前後関係順に、論理レコードをファイルから取り出したり、ファイルに書き込んだりするアクセス・モード。

順次ファイル (* sequential file)

順次編成のファイル。

順次編成 (* sequential organization)

レコードがファイルに書き込まれるときに確定されたレコードの前後関係によって識別されるような永続的な論理ファイル構造。

逐次探索 (serial search)

最初のメンバーから始めて最後のメンバーで終わるように、ある集合のメンバーが連続的に検査される探査方法。

セッション Bean

EJB において、クライアントによって作成され、通常は 1 つのクライアント/サーバー・セッションの期間だけ存在する Enterprise Bean。(Oracle)

77 レベル記述記入項目 (77-level-description-entry)

レベル番号 77 を持つ不連続データ項目を記述するデータ記述記入項目。

符号条件 (* sign condition)

データ項目や算術式の代数値が、0 より小さいか、大きい、または等しいかという命題で、それに関して真理値が判別できる。

シグニチャー (signature)

(1) ある操作とそのパラメーターの名前。(2) あるメソッドの名前とその仮パラメーターの数と型。

単純条件 (* simple condition)

以下のセットから選択される任意の単一条件。

- 比較条件
- クラス条件
- 条件名条件
- スイッチ状況条件
- 符号条件

条件 (condition) および 単純否定条件 (negated simple condition) も参照。

1 バイト文字セット (single-byte character set (SBCS))

各文字が 1 バイトで表現される文字のセット。ASCII および EBCDIC (拡張 2 進化 10 進コード) (EBCDIC (Extended Binary-Coded Decimal Interchange Code)) も参照。

遊びバイト (レコード内) (slack bytes (within records))

複数の基本データ項目を正しく位置合わせするために、コンパイラーによってデータ項目間に挿入されるバイト。遊びバイトには意味のあるデータは含まれない。正しい位置合わせを行うために遊びバイトが必要なときは、SYNCHRONIZED 節によって、コンパイラーに遊びバイトを挿入させる。

遊びバイト (レコード間) (slack bytes (between records))

複数の基本データ項目を正しく位置合わせするために、プログラマーによってファイルのブロック化論理レコードの間に挿入されるバイト。場合によっては、レコード間に遊びバイトを挿入することによってバッファ内で処理されるレコードのパフォーマンスが改善される。

ソート・ファイル (* sort file)

形式 1 SORT ステートメントによってソートされるレコードの集合。ソート・ファイルは、ソート機能によってのみ作成され使用される。

ソート・マージ・ファイル記述項目 (* sort-merge file description entry)

DATA DIVISION の FILE SECTION 内の中にある記入項目項目。レベル標識 SD と、それに続くファイル名、および必要に応じて一連のファイル文節から構成される。

* SOURCE-COMPUTER

ENVIRONMENT DIVISION にある段落の名前であり、ここではソース・プログラムがコンパイルされるコンピューター環境が記述される。

コンパイル用コンピューター記入項目 (* source computer entry)

ENVIRONMENT DIVISION の SOURCE-COMPUTER 段落内の記入項目であり、ソース・プログラムがコンパイルされるコンピューター環境を記述する節が入っている。

ソース項目 (* source item)

SOURCE 節によって指定される ID で、印刷可能な項目の値を提供する。

ソース・プログラム (source program)

ソース・プログラムは、他の形式や記号を使用して表現することができるが、本書では、構文的に正しい COBOL ステートメントの集合を常に指している。COBOL ソース・プログラムは、IDENTIFICATION DIVISION または COPY ステートメントで始まり、指定された場合はプログラム終了マーカーで終了するか、または追加のソース・プログラム行なしで終了する。

ソース単位 (source unit)

COBOL ソース・コードの 1 単位で、個別にコンパイルできる。プログラムまたはクラス定義。コンパイル単位とも呼ばれる。

特殊文字 (special character)

以下の Secure Electronic Transaction に属するキャラクター。

文字	意味
+	正符号
-	負符号 (-) (ハイフン)
*	アスタリスク
/	斜線 (スラッシュ)
=	等号
\$	通貨符号
,	コンマ
;	セミコロン
.	ピリオド (小数点、終止符)
"	引用符
'	アポストロフィ
(左括弧
)	右括弧
>	より大きい
<	より小さい
:	コロ
_	下線

SPECIAL - NAMES

ENVIRONMENT DIVISION にある段落の名前。この段落では、環境名がユーザー指定の簡略名と関連付けられる。

特殊名記入項目 (* special names entry)

ENVIRONMENT DIVISION の SPECIAL - NAMES 段落内の記入項目。この記入項目は、通貨記号を指定したり、小数点を選択したり、シンボリック文字を指定したり、インプリメントする人の名前をユーザー指定の簡略名と関連付けたり、英字名を文字セットまたは照合シーケンスと関連付けたり、クラス名を一連の文字と関連付けたりするための手段を提供する。

特殊レジスター (* special registers)

コンパイラの生成する特定のストレージ域のことで、その基本的な使用法は、具体的な COBOL 機能を使用したときに作り出される情報を記憶することである。

標準データ・フォーマット (* standard data format)

COBOL データ部でデータの特性を記述するために使用される概念。この概念のもとでは、データの特性は、データが内部的にコンピューターに、または特定の外部メディアに保管される方法に適した形式ではなく、印刷ページ上での無限の長さを持つデータ 外観に適した形式で表現される。

ステートメント (* statement)

COBOL ソース・プログラムに書かれる、動詞を冒頭に置いた、ワード、リテラル、および区切り記号の構文的に正しい組み合わせ。

構造化プログラミング (structured programming)

コンピューター・プログラムを編成してコーディングするための技法であり、この技法では、プログラムはセグメントの階層で構成され、それぞれのセグメントには 1 つの入り口点と 1 つの出口点がある。制御は、構造の下方へと渡され、階層内のより上位レベルへの無条件ブランチは行われない。

サブクラス (* subclass)

別のクラスから継承するクラス。継承関係にある 2 つのクラスをまとめて考える場合、継承する側、つまり継承先のクラスをサブクラスといい、継承される側、つまり継承元のクラスをスーパークラスという。

項目のサブジェクト (* subject of entry)

DATA DIVISION の記入項目内において、レベル標識またはレベル番号の直後に現れるオペランドまたは予約語。

サブプログラム (* subprogram)

呼び出し先プログラム (called program) を参照。

添え字 (* subscript)

整数、(オプションで演算子 + または - 付きの整数が後ろにある) データ名、あるいは (オプションで演算子 + または - 付きの整数が後ろにある) 索引名のいずれかによって表されるオカレンス番号。これによりテーブル内の特定のエレメントを識別する。可変数の引数を認める関数では、添え字付き ID を関数引数として使用する場合は、添え字に ALL を使用できる。

添え字付きデータ名 (* subscripted data-name)

データ名とその後の括弧で囲まれた 1 つ以上の添え字から構成される ID。

置換文字 (substitution character)

ソース・コード・ページからターゲット・コード・ページへの変換の際に、ターゲット・コード・ページで定義されていない文字を表すのに使用される文字。

スーパークラス (* superclass)

別のクラスによって継承されるクラス。サブクラス (subclass) も参照。

サロゲート・ペア (surrogate pair)

UTF-16 形式のユニコードで、共に 1 つのユニコード図形文字を表すエンコード方式ペアの単位。ペアの最初の単位は上位サロゲートと呼ばれ、第 2 の単位は下位サロゲートと呼ばれる。上位サロゲートのコード値の範囲は、X'D800' から X'DBFF' である。下位サロゲートのコード値の範囲は、X'DC00' から X'DFFF' である。サロゲート・ペアは、Unicode 16 ビット・コード化文字セットに適合する文字を 65,536 文字を超えて提供する。

スイッチ状況条件 (switch-status condition)

オンまたはオフに設定可能な UPSI スイッチが、特定の状況に設定されているという命題で、これに関して真理値を判別することができる。

シンボリック文字 (* symbolic-character)

ユーザー定義の形象定数を指定するユーザー定義語。

構文

(1) 意味や解釈および使用の方法に依存しない、文字同士または文字のグループ同士の間の関係。(2) 言語における表現の構造。(3) 言語構造を支配する規則。(4) 記号相互の関係。(5) ステートメントの構築にかかわる規則。

システム名 (* system-name)

オペレーティング環境と連絡し合うために使用される COBOL ワード。

T

テーブル (* table)

OCCURS 節を使用して DATA DIVISION で定義される、論理的に連続するデータ項目の集合。

テーブル・エレメント (* table element)

テーブルを構成する繰り返し項目の集合に属するデータ項目。

テキスト・デッキ (text deck)

オブジェクト・デッキ (object deck) またはオブジェクト・モジュール (object module) と同義。

テキスト名 (* text-name)

ライブラリー・テキストを識別するユーザー定義語。

テキスト・ワード (* text word)

以下のいずれかの文字から成る COBOL ライブラリー、ソース・プログラム、または疑似テキスト内のマージン A およびマージン R の間の、1 文字または連続した文字のシーケンス。

- スペース以外の区切り記号、疑似テキスト区切り文字、英数字リテラルの開始と終了の区切り文字。ライブラリー、ソース・プログラム、または疑似テキスト内のコンテキストに関係なく、右括弧文字と左括弧文字は常にテキスト・ワードと見なされる。
- リテラル。英数字リテラルの場合には、そのリテラルの境界となる開始の引用符と終了の引用符を含むリテラル。

- ・ コメント行および区切り記号によって囲まれたワード COPY を除く、その他の連続する一連の COBOL 文字で、区切り記号でもリテラルでもないもの。

THREAD

プロセスの制御下にあるコンピューター命令のストリーム (プロセス内のアプリケーションによって開始される)。

トークン

COBOL エディターでは、プログラムにおける意味の単位。トークンには、データ、言語キーワード、ID、またはその他の言語構文の一部を含めることができる。

トップダウン設計 (top-down design)

関連付けられた諸機能が、構造の各レベルで実行されるようにする階層構造を使ったコンピューター・プログラムの設計。

トップダウン開発 (top-down development)

構造化プログラミング (*structured programming*) を参照。

トレーラー・ラベル (trailer-label)

(1) 記録メディア・ユニットのデータ・レコードの後にある、データ・セットのラベル。(2) 「ファイル終わりラベル (*end-of-file label*)」の同義語。

トラブルシューティング (troubleshoot)

コンピューター・ソフトウェアの使用中に問題を検出し、突き止め、除去すること。

真の値 (* truth value)

2つの値 (真または偽) のどちらか一方によって、条件評価の結果を表したもの。

型式化オブジェクト・リファレンス (typed object reference)

指定されたクラスまたはそのサブクラスのオブジェクトだけを参照できるデータ名。

U

単項演算子 (* unary operator)

正符号 (+) または負符号 (-)。算術式の変数や算術式の左括弧の前に置き、それぞれ +1 または -1 を式に乗算する。

無制限テーブル (unbounded table)

上限として *integer-2* を指定する代わりに、OCCURS *integer-1* to UNBOUNDED を持つテーブル。

UNICODE

現代世界の各国の言語で記述されるテキストの交換、処理、表示をサポートする汎用文字エンコード標準。UTF-8、UTF-16、UTF-32 など、Unicode を表現する複数のエンコード・スキームがある。Enterprise COBOL では、国別データ・タイプの表記としてビッグ・エンディアン・フォーマットの UTF-16 を使用して Unicode をサポートしている。

URI (Uniform Resource Identifier (URI))

リソースを一意に指す文字のシーケンスのことで、Enterprise COBOL では、名前空間の ID。URI の構文は、[Uniform Resource Identifier \(URI\): Generic Syntax](#) の資料に定義されている。

unit

直接アクセスのモジュールであり、その大きさは IBM によって決められている。

汎用オブジェクト参照 (universal object reference)

どのクラスのオブジェクトでも参照できるデータ名。

無制限ストレージ

AMODE 31 では、無制限ストレージは、2 GB 境界より下にあり、16 MB 境界より上または下に配置可能。

AMODE 64 では、無制限ストレージには、プログラムで使用可能なすべてのストレージ (2 GB 境界より上と下の両方) が含まれる。

不成功の実行 (* unsuccessful execution)

ステートメントの実行が試みられたが、そのステートメントに指定された操作すべてを実行できなかったこと。あるステートメントの実行不成功は、そのステートメントによって参照されるデータには影響を及ぼさないが、状況表示には影響を与える可能性がある。

UPSI スイッチ (UPSI switch)

ハードウェア・スイッチの機能を実行するプログラム・スイッチ。UPSI-0 から UPSI-7 の 8 つのスイッチがある。

URI

URI を参照。

ユーザー定義語 (* user-defined word)

節やステートメントの形式を満たすためにユーザーが提供する必要のある COBOL ワード。

V

変数 (* variable)

オブジェクト・プログラムの実行によって変更を受ける可能性のある値を持つデータ項目。算術式で使われる変数は、数字基本項目でなければならない。

可変長項目 (variable-length item)

OCCURS 節の DEPENDING 句で記述されたテーブルを含んだグループ項目。

可変長レコード (* variable-length record)

ファイル記述項目またはソート・マージ・ファイル記述項目が、文字位置の数が可変であるレコードを許容しているファイルに関連付けられているレコード。

可変オカレンス・データ項目 (* variable-occurrence data item)

可変オカレンス・データ項目とは、反復される回数が可変であるテーブル・エレメントを言う。そのような項目は、そのデータ記述項目内に OCCURS DEPENDING ON 節を含んでいるか、そのような項目に従属している必要がある。

可変位置グループ (* variably located group)

同じレコード内の可変長テーブルに続くグループ項目 (可変長テーブルに従属するわけではない)。グループ項目は、英数字グループでも国別グループでも構いません。

可変位置項目 (* variably located item)

同じレコード内の可変長テーブルに続くデータ項目 (可変長テーブルに従属するわけではない)。

動詞 (* verb)

COBOL コンパイラまたはオブジェクト・プログラムによってとられる処置を表すワード。

VOLUME

外部ストレージのモジュール。テープ装置の場合はリール、直接アクセス装置の場合はユニット。

ボリューム切り替え処理手順 (volume switch procedures)

ファイルの終わりに達する前にユニットまたはリールの終わりに達したとき、自動的に実行されるシステム固有の処理手順。

VSAM ファイル・システム (VSAM file system)

COBOL の順次編成、相対編成、および索引編成をサポートするファイル・システム。

W

Web サービス (web service)

特定のタスクを実行し、HTTP や SOAP といったオープン・プロトコルを介してアクセス可能なモジュラー・アプリケーション。

空白文字 (white space)

文書にスペースを挿入する文字。空白文字には以下のものがある。

- スペース
- 水平タブ
- 復帰
- 改行
- 次の行

Unicode 標準では上記のように呼ばれる。

ワード (* word)

ユーザー定義語、システム名、予約語、または関数名を形成する、30 文字を超えない文字ストリング。

* WORKING-STORAGE SECTION

WORKING-STORAGE データ項目を記述する DATA DIVISION のセクション。独立項目または WORKING-STORAGE レコード、あるいはその両方で構成される。

ワークステーション (workstation)

コンピューターの総称 (パーソナル・コンピューター、3270 端末、インテリジェント・ワークステーション、および UNIX 端末を含む)。ワークステーションはメインフレームまたはネットワークに接続されることがよくある。

ラッパー

オブジェクト指向コードとプロシージャ指向コード間のインターフェースを提供するオブジェクト。ラッパーを使用すると、他のシステムがプログラムを再利用したり、プログラムにアクセスしたりできるようになる。

X

X

PICTURE 節内の記号であり、コンピューターの有する文字セットの任意の文字を含めることができる。

XML

Extensible Markup Language。マークアップ言語を定義するための標準メタ言語。SGML から派生した、SGML のサブセットである。XML では、SGML の複雑で使用頻度の低い部分が省略され、文書タイプを扱うアプリケーションの作成、構造化情報の作成および管理、異種コンピューター・システム間での構造化情報の伝送および共有がはるかに容易になっている。XML を使用するとき、SGML で必要とされるような堅固なアプリケーションや処理は不要である。XML は、World Wide Web Consortium (W3C) の主導で開発された。

XML データ (XML data)

XML エlement を持つ階層構造に編成されたデータ。データ定義は XML エlement ・タイプ宣言で定義される。

XML 宣言 (XML declaration)

使用している XML のバージョンや文書のエンコードなど、XML 文書の特性を指定する XML テキスト。

XML 文書 (XML document)

W3C XML 規格で定義されているとおり正しい形式のデータ・オブジェクト。

XML ネーム・スペース (XML namespace)

W3C XML ネーム・スペース仕様によって定義されたメカニズムで、Element 名および属性名の集まりの有効範囲を制限する。一意的に選択された XML 名前空間によって、複数の XML 文書または XML 文書内の複数のコンテキストで Element 名または属性名が一意的に識別されます。

XML スキーマ (XML schema)

W3C によって定義されたメカニズムで、XML 文書の構造と内容を記述し、制約する。XML スキーマは、それ自体が XML で表され、特定タイプ (購入注文など) の XML 文書のクラスを効率的に定義する。

Z

z/OS UNIX ファイル・システム

階層構造で編成されたファイルとディレクトリーの集合であり、z/OS UNIX を使用してアクセスできる。

ゾーン 10 進数データ項目 (zoned decimal data item)

暗黙的または明示的に USAGE DISPLAY として記述され、PICTURE の記号 9、S、P、V の有効な組み合わせを含んでいる、外部 10 進数データ項目。ゾーン 10 進数データ品目の中身は、0 から 9 の文字で表され、オプションで符号付きで表されます。PICTURE スtring が符号を指定していて、SIGN IS SEPARATE 節が指定されている場合、符号は文字 + または - として表される。SIGN IS SEPARATE が指定されていない場合、符号は、符号位置の最初の 4 ビットをオーバーレイする 1 つの 16 進数字である (先行または末尾)。

#

85 COBOL 標準

以下の標準によって定義された COBOL 言語。

- 「ANSI INCITS 23-1985, Programming languages - COBOL」は「ANSI INCITS 23a-1989, Programming Languages - COBOL - Intrinsic Function Module for COBOL」および「ANSI INCITS 23b-1993, Programming Languages - Correction Amendment for COBOL」に改訂されました。
- 「ISO 1989:1985, Programming languages - COBOL」は「ISO/IEC 1989/AMD1:1992, Programming languages - COBOL: Intrinsic function module」および「ISO/IEC 1989/AMD2:1994, Programming languages - Correction and clarification amendment for COBOL」に改訂されました。

2002 COBOL 標準

以下の標準によって定義された COBOL 言語。

- INCITS/ISO/IEC 1989-2002, Information technology - Programming languages - COBOL

2014 COBOL 標準

以下の標準によって定義された COBOL 言語。

- INCITS/ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL

リソース・リスト

Enterprise COBOL for z/OS

COBOL for z/OS の資料

以下の資料が「[Enterprise COBOL for z/OS ライブラリー](#)」にあります。

- 新機能, SC31-5708-00
- カスタマイズ・ガイド, SC27-8712-03
- 言語リファレンス, SC27-8713-03
- プログラミング・ガイド, SC27-8714-03
- マイグレーション・ガイド, GC27-8715-03
- パフォーマンス・チューニング・ガイド, SC27-9202-02
- メッセージとコード, SC27-4648-02
- プログラム・ディレクトリー, GI13-4526-03
- ライセンス・プログラム仕様, GI13-4532-03

ソフトコピー資料

次のコレクション・キットには、Enterprise COBOL およびその他の製品資料が含まれます。それらは <https://www.ibm.com/resources/publications> にあります。

- *z/OS Software Products Collection*
- *z/OS and Software Products DVD Collection*

サポート

Enterprise COBOL for z/OS のご使用の際に問題がある場合は、サイト: <https://www.ibm.com/support/pages/node/6560933> を参照してください。そこでは最新のサポート情報が提供されています。

関連資料

z/OS ライブラリー資料

以下の資料が「[z/OS ライブラリー](#)」にあります。

ランタイム・ライブラリー拡張機能

- 一般的なデバッグ方式 ライブラリー参照
- 一般的なデバッグ方式 ユーザーズ・ガイド
- DWARF/ELF 拡張ライブラリー参照

z/Architecture

- *z/Architecture* 解説書

z/OS DFSMS

- カタログのためのアクセス方式サービス・プログラム
- *Checkpoint/Restart*
- *Macro Instructions for Data Sets*
- データ・セットの使用法

- *Utilities*

z/OS DFSORT

- アプリケーション・プログラミング・ガイド
- インストールおよびカスタマイズ

z/OS ISPF

- ダイアログ開発者 ガイドとリファレンス
- ユーザーズ・ガイド 第1巻
- ユーザーズ・ガイド 第2巻

z/OS 言語環境プログラム

- 概念
- カスタマイズ
- デバッグのガイド
- *Language Environment Vendor Interfaces*
- プログラミング・ガイド
- プログラミング・リファレンス
- ランタイム・メッセージ
- ランタイム マイグレーション・ガイド
- ILC (言語間通信) アプリケーションの作成

z/OS MVS

- JCL 解説書
- JCL ユーザーズ・ガイド
- プログラミング: 高水準言語向け呼び出し可能サービス
- プログラム管理: ユーザーズ・ガイドおよび解説書
- システム・コマンド
- *z/OS Unicode Services* ユーザーズ・ガイドおよび解説書
- *z/OS XML* システム・サービス ユーザーズ・ガイドおよび解説書

z/OS TSO/E

- コマンド解説書
- 入門
- ユーザーズ・ガイド

z/OS UNIX システム・サービス

- コマンド解説書
- プログラミング: アセンブラー呼び出し可能サービス 解説書
- ユーザーズ・ガイド

z/OS XL C/C++

- プログラミング・ガイド
- ランタイム・ライブラリー・リファレンス

CICS Transaction Server for z/OS

以下の資料が「[CICS ライブラリー](#)」にあります。

- CICS アプリケーションの開発

- API (EXEC CICS) リファレンス
- CICS システム・プログラムの開発
- グローバル・ユーザー出口リファレンス
- XPI Reference
- CICS での EXCI の使用

COBOL 報告書作成プログラム・プリコンパイラー

- *Programmer's Manual*, SC26-4301
- *Installation and Operation*, SC26-4302

Db2 for z/OS

以下の資料が「[Db2 ライブラリー](#)」にあります。

- アプリケーション・プログラミングおよび SQL ガイド
- コマンド解説書
- SQL 解説書

IBM z/OS Debugger (以前の IBM z システムズ およびデバッグ・ツールのデバッグ)

IBM z/OS Debugger については、[IBM z/OS Debugger ライブラリー](#)を参照してください。

IBM Developer for z/OS (以前の IBM z Systems のためのデバッグ)

IBM Developer for z/OS に関する情報は、[IBM Developer for z/OS ライブラリー](#)にあります。

注：IBM Developer for z/OS IBM Developer for z Systems® および Rational® Developer for z Systems に取って代わります。

以下の資料が「[IBM Publications Center](#)」にあり、資料番号で検索できます。

IMS

- *Application Programming API Reference*, SC18-9699
- *Application Programming Guide*, SC18-9698

WebSphere® Application Server for z/OS

- *Applications*, SA22-7959

Softcopy publications for z/OS

以下のコレクション・キットには、z/OS および関連製品資料が含まれます。

- *z/OS CD Collection Kit*, SK3T-4269

java

- IBM ジャワ - ツール資料用の SDK, publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp
- *The Java 2 Enterprise Edition Developer's Guide*, download.oracle.com/javaee/1.2.1/devguide/html/DevGuideTOC.html
- *Java 2 on z/OS*, www.ibm.com/servers/eserver/zseries/software/java/
- *The Java EE 5 Tutorial*, download.oracle.com/javaee/5/tutorial/doc/
- *Java Language Specification, Third Edition* (Gosling ほかに), java.sun.com/docs/books/jls/

- *The Java Native Interface*, download.oracle.com/javase/1.5.0/docs/guide/jni/
- *JDK 5.0 Documentation*, download.oracle.com/javase/1.5.0/docs/

JSON

- JavaScript Object Notation (JSON), www.json.org

Unicode および文字表現

- *Unicode*, www.unicode.org/
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

XML

- *Extensible Markup Language (XML)*, www.w3.org/XML/
- *Namespaces in XML 1.0*, www.w3.org/TR/xml-names/
- *Namespaces in XML 1.1*, www.w3.org/TR/xml-names11/
- *XML specification*, www.w3.org/TR/xml/



プログラム番号: 5655-EC6

SC27-9202-02

