4D V11 SQL IN DEPTH

データベースの内部設計とパフォーマンスに関する考察

4D デベロッパカンファレンス 2010 配付資料

目次

第一部・クライアントサーバーモデルの解析

双子（ときには三つ子）のプロセス達

クライアントのグローバル（つまりローカルではない）プロセスは，必ずサーバーサイドのツインプロセス（この文脈における「プロセス」とは，OS のプロセスではなく 4D のプロセスのこと）と通信しています。双子の一方はプリエンプティブスレッドであり，純粋な DB4D リクエストを司っています。具体的には，クライアントサイドで発行される QUERY，ORDER BY，CREATE RECORD などのコマンドがこれに該当します。双子の他方はコオペラティブスレッドであり，アプリケーションリクエストを担当しています。具体的には Current date(*)，GET PROCESS VARIABLE(-1;…)などのコマンド，および トリガすべてがこれに該当します。

コオペラティブスレッドは，どれだけ数が多くても，すべてひとつの CPU コアによって処理されなければなりません。言い換えるならば，単一のコオペラティブスレッドが CPU を完全に占有することも可能であり，ひとつのコオペラティブスレッドが他のコオペラティブスレッド全部をブロックできるということを意味します。

プリエンプティブスレッドは，複数の CPU コアに対して処理を分散することができます。プリエンプティブスレッドがどのような仕方で同時に複数の CPU を使用するのかは OS の判断に委ねられています。そのようなわけで，プリエンプティブスレッドが他のプリエンプティブスレッドを完全にブロックしてしまうことは非常に稀です。

クライアントのグローバルプロセスは，ときとして第三のスレッドを使用することがあります。このスレッドもまた，プリエンプティブなものです。このスレッドは，Begin SQL の開始とともに作成されます。これら三個のスレッドは，それぞれことなるネットワークポートを使用します。

ひとつのグローバルプロセスにつき，最大で三個のスレッドが使用されるということは，とりわけクライアント数が多いシステムにおいて，リソースの使用量という形で影響が現れます。アプリケーションを存続させるだけでも大量のメモリが必要になるのです。そのような極限状態では，それ自体はコオペラティブスレッドであるクライアントコネクションハンドラーがボトルネックとなり，新規接続が困難になる可能性も考えなければなりません。

コオペラティブスレッドのストレス緩和が先決

トリガ，「サーバーで実行」プロパティが設定されたメソッド，ストアドプロシージャー，Web サーバーおよび SOAP サーバー，サーバーサイドのユーザーインタフェース（管理画面）など，コオペラティブなプロセスを使用するものを必要最低限にとどめることはたいせつです。また，どのコマンドがプリエンプティブなのかを意識することも有益でしょう。SQL サーバーおよび DB4D アクセスは，プリエンプティブスレッドなので大丈夫ですが，QUERY WITH ARRAY など，中には例外的なものもあるので注意が必要です。

ネットワークトラフィックの最適化

レコードの構造がネットワークトラフィックに与える影響については，これまで多くの論考が重ねられてきました。その代表的な例が，「BLOB の別テーブル保存」です。4D では，テキスト，ピクチャ，BLOB をまとめて「おおきなオブジェクト（BLOB）」と呼称しています。4D v11 SQL は，テキストを「レコードの外に」保存できるようになり，このオプションは，とりわけ同じテーブルに対してシーケンシャルクエリを実行する際のパフォーマンスを向上するために効果的です。このプロパティを活用すれば，BLOB を別テーブルに保存しなくても，ネットワークトラフィックを最適化することができます。しかしながら，ピクチャおよび BLOB は，引き続きレコードと一緒に保存されるため，そのような内部最適化は働いていません。ピクチャおよび BLOB を含む，すべての BLOB が「レコードの外に」保存できるようになるのは 4D v12 です。なお，4D v13 では，BLOB が外部ドキュメントとしてほんとうの意味でレコードが「データベースの外に」保存できるになる予定です。

第二部・SQL vs QUERY

何でも SQL で書き換えればそれだけで速くなるというのは本当か

実際に書き換えてみると 4D ランゲージよりも SQL のほうが遅いケースが結構あります。それはなぜでしょうか。理由のひとつは，両者がことなる言語体系であることが多分に関係しています。4D のコマンドは，おおきな特徴として細分不可（アトミック）であることが挙げられます。QUERY を例に挙げるならば，検索，セレクションの作成，レコードのロードまでがひとつのコマンドであり，特定のアクションに対してコマンドが極めて最適化されています。反対に SQL は極めて汎用的（ジェネリック）です。基本的に SELECT というコマンドがあるだけであり，データベースアクセスに関するすべてのアクションがこのコマンドひとつに集約されています。セレ CT コマンドは内部的にあらゆるケースを想定している必要があり，JOIN があるのかないのか，テーブルは単一なのか複数なのか，並び替えのするのか，グループはしないのかなど，その用途は非常に多岐にわたります。汎用的であるということは，それだけコマンドが最適化しづらいということを意味します。

単純な例で比較してみましょう。すべてのデータをとりだして配列にコピーするというアクションです。4D ランゲージであれば，ALL RECORDS，SELECTION TO ARRAY，これだけで十分であり，処理はほとんど瞬時です。コマンドがこのアクションに対して最適化された専用のものであることに加え，データベースエンジンに対する直接的なアクセス能力を有していることがその背景にあります。4D ランゲージを使用すれば，メモリ上で配列を特別な構造体にラップし，DB4D にそのまま渡すことができます。結果的に前述の処理は，メモリコピーとさほど変わらないレベルにまで最適されているのです。これに対し，SQL はもっと複雑です。まず，ステートメントを解析するところから始めなければなりません。それぞれのオブジェクト（テーブルと配列）が確かに存在することをチェックした後，型の互換性を検証しなければならないという意味です。そのチェックをパスしてはじめて，メインの処理であるメモリコピーに進むことができるのです。

おおまかにいって，原状ではステートメントの解析に 7%，ステートメントの検証に 15%，ODBC または SQL パススルー用の内部セッション管理に 7%，ランゲージバインディング（ODBC，内部，外部データベースの識別）に 8%，データベースエンジンとの言語障壁を超えることに 9%のオーバーヘッドがあります。言語障壁についてですが，これは SQL には DB4D エンジンに対するダイレクトアクセスがないことを指しています。DB4D のキャッシュは保護されており，データはレコード単位で取り出さなければなりません。どんなに最適化を進めても，v11 の場合，前述のクエリは SQL のほうが宿命的に 5 倍から 10 倍は遅いということが判明しています。

この点で v12 は相当な改善が施されました。ローカルデータベースの場合，データの取り出しは 2 倍から 3 倍高速になり，リモートデータベースでは 5 倍から 20 倍高速になりました。とはいえ，依然，v12 でも SQL は 4D ランゲージよりも 1.5 倍から 2 倍は遅いというのが原状です。それでも，技術的には 4D と SQL の差は 1-2%というレベルにまで縮められると考えられています。

ここまでの主要な論点は，4D ランゲージは優れたデータベース言語であり，なんでも SQL のほうが勝っているということはなく，既存のクエリを SQL で書き換えれば必ず速くなるというほど単純な話ではないということです。

SQL が勝っているのはどのような場合か

SQL が 4D よりも優れているケースはたくさん挙げることができます。これまでの例は，はっきりいって公平な比較ではありませんでした。式の評価などは SQL に軍配が上がる良い例です。たとえば，テーブルに貸出金額および借り入れ金額というフィールドがあったとしましょう。「SELECT（貸出-借り入れ）FROM 顧客 INTO 収支」みたいなステートメントは，4D に匹敵するレベル，あるいはそれ以上のパフォーマンスがみられるはずです。V12 であれば，数値式の評価は 23 倍高速になりました。文字列の比較は，より複雑な要素が関係しているので，そこまで劇的な違いはありませんが，それでもずっと高速です。

また，SQL サーバーはプリエンプティブであるということも見逃せない要素です。4D ランゲージは，構造上，複数の CPU コアを同時に使用することができません。（一部の機能は例外的に複数のコアを使用することがあり

ます。）SQL は，式の評価を同時に複数のコアで実行することができるので，それだけ速く計算を終えることができるのです。もっとも，メモリの速度やディスクアクセスなど，CPU コア以外の部分が最終的なパフォーマンスを左右することを忘れてはなりません。

また，単一のテーブルにアクセスしているのか，複数のテーブルにアクセスしているのか，という点も重要です。同一のテーブルに対する集中的なアクセスはマルチコアパフォーマンスを低下させます。こうしたことをすべて考慮し，最悪のクエリ，つまり同一のテーブルに対する集中的なアクセスで試した場合でも，SQL は 8 コアマシンの CPU を 4 コアないし 5 コアまで同時に使用することが確認できました。複数のテーブルが関係していれば，これが 7 コアにまで増えました。（残る一個のコアは，明らかに 4D のメインスレッドが占有しているものです。）

SQL は，式の評価を要する計算処理が，とりわけマルチコアマシンにおいて優れているということです。

もうひとつ，初代の 4D SQL（つまり v11）は，クライアントサーバーのパフォーマンスについて，まだ改善できる余地がありました。4D v12 では，ネットワークコミュニケーションで使用される内部バッファリングが完全に書き直され，新しくなりました。その結果，v12 では，データの量が多ければ多いほど，そして一度にアクセスする列（フィールド）の数が多ければ多いほど，データのダウンロードが速くなります。具体的には，異なるフィールドタイプ，たくさんのフィールド数，大量のデータであれば，最大で 17 から 18 倍，高速です。その上，新しいアルゴリズムは，これまでよりも 5-7%少ないメモリをクライアントとサーバーのそれぞれで消費します。実のところ，SQL サーバーは全体的にメモリ管理がずっと洗練されたものになり，とりわけ大きな対象から比較的小さな結果を取り出すような処理においてはその違いが顕著です。4D v12 は，ローカルアクセスでさえ以前よりも軽快なものになり，リモートアクセスでは一層パフォーマンスが良くなりました。

もうひとつ，v12 では，SQL のステートメントキャッシングを始めました。SQL は，常にテキストベースであり，またコンパイルされていません。しかしながら，ほとんどの場合，リモート SQL はパラメータ化されているとはいえ，完全にダイナミックなものではないのが現実です。そこで v12 では，一度処理した SQL の内部形式とそのスタンプを記憶するようにしました。SQL アナライザーが新しい SQL ステートメントを受け取ると，はじめに再利用できるオブジェクトがないかどうかをチェックします。ステートメントの種類にもよりますが，これで 5%ほどのパフォーマンス向上が見込めるということが分かったからです。そのような訳で，SQL はこれからも進化と改良を重ね，特にネットワーク越しの使用，そしておおきなデータを扱う場合において向上してゆく見込みです。

ところで，しばしば問い合わせがある点なのですが，4D SQL は SQL-92 を元にしているということに留意してください。4D に寄せられる SQL 関連の不具合報告中，かなり部分を占めるのが「SQL の仕様と違う」という主旨のものですが，その多くは，Oracle（あるいは他の SQL 製品）の仕様との違いを指摘したものでした。今後 SQL を改善してゆく中で，同一のキーワードに複数のエイリアスを設けるなど，さまざまな仕方で SQL を拡張し，より互換性を高めるということにも積極的に取り組んでゆく予定です。

第三部・キャッシュの謎

キャッシュの中には何かあるのか

キャッシュは，4D が独自のメモリマネージャーを使用して管理しているおおきなメモリ領域です。しばしばデータキャッシュなどとも呼ばれますが，キャッシュで管理されているものはデータオブジェクトだけではありません。キャッシュはさまざまなデータエンジンオブジェクトのために幅広く使用されています。

キャッシュに置かれるものとしては，テーブル・フィールド・リレーションなどのストラクチャ定義，インデックス，現在開かれているデータベースに関する種々の情報（ファイルパスやファイルプロパティ），データファイルのアロケーションビットテーブル，レコード・インデックス・BLOB などのアドレステーブル，インデックスページ，レコード，（キャッシュに収まらないときはカーネルメモリに置かれる場合もある）BLOB，シーケンシャルナンバー，トランザクション，セレクション，セット，並び替のために使用される一時的バッファ，ディスク書き込みや読み出しのバッファなどが挙げられます。もっとも，このリストは決定的なものではなく，将来，変更される可能性があります。

キャッシュの使用量が増減する理由

アプリケーションを長時間実行していると，キャッシュの占有スペースは次第におおきくなってゆき，ある時点で下がることがあります。これは，新たなオブジェクトが入る場所を確保するために，一部のオブジェクトがキャッシュから除去されているためです。

キャッシュのサイズ

4D は起動時に環境設定で指定されているだけのバーチャルメモリを OS に対して要求します。4D が推奨するのは絶対値によるキャッシュサイズの設定です。11.6 以降，キャッシュサイズの最大値は 2.5GB にハードコーディングされました。たとえ OS が 64 ビットであったとしても，それ以上のサイズを設定することはできません。キャッシュサイズの理論的な上限がなくなるのは 4D v12 64 ビット版サーバーからです。

キャッシュの管理

4D は，キャッシュのサイズが足りなくなると，キャッシュの 10%をパージ（解放）しようと試みます。はじめにバッファをフラッシュし，その後，キャッシュの 10%をパージするのです。（フラッシュとパージは違います。フラッシュはメモリに置かれたレコードをディスクに書き込むことですが，その後もレコードはメモリに残っています。これに対し，パージされたオブジェクトはディスクに書き込まれるだけでなく，メモリからも解放されています。）それでも十分のスペースが確保できない場合，4D はフラッシュできないオブジェクトの一部をディスクに移し，それらをメモリからパージします。これらパージされたオブジェクトは，データファイルと同じ階層（デフォルトの位置）に現れる Temporary Files フォルダに一時保存されています。

では，一体どのオブジェクトからフラッシュされてゆくのでしょうか。はじめに，4D はフラッシュできるオブジェクトがあれば，キャッシュをジャンプしてそのオブジェクトに到達し，ディスクにフラッシュします。ただし，いつも同じオブジェクトから順番にフラッシュされてゆくとは限りません。フラッシュの順番は，オブジェクトの配置と密接な関連があります。ディスク上で密集しているオブジェクトほど素早く大量にフラッシュできるのであり，そのために定期的な圧縮が（現実的であれば）推奨されているのです。

通常，最初にパージされるのは，比較的「小さな」オブジェクトです。キャッシュ内でオブジェクトはそのタイプごとにグループ化されているからです。とはいえ，1000 万レコード分のアドレステーブルは，それぞれ 64KB 以下の細かいアドレステーブルに分割されていますし，セットもまた圧縮・分解されて小さなオブジェクトに分かれているので，それ自体は（レコード数が多いからといって）おおきなオブジェクトにはなりません。おおきなオブジェ

クトとは，ピクチャ，BLOB，（レコードと一緒に保存された）テキストのことであり，これらは必ずしも頻繁に使用されないオブジェクトであったとしても，最後までキャッシュに残る傾向があります。

4D v11 と 2004 のキャッシュの違い

4D 2004 のキャッシュは，Mac OS のハンドル，つまりポインタに対するポインタと連結リストを使用していました。これは，メモリ内でオブジェクトが動き得たことを意味し，4D は目的のオブジェクトに到達するまでリンクを順番にたどってゆく必要がありました。4D v11 の新しいキャッシュメモリマネージャーは，ハンドルではなく，ポインタ，および一種のアドレステーブルを使用しています。オブジェクトがメモリ内で移動することはなく，最大でも3回のジャンプで必ず目的のオブジェクトに到達することができます。

4D 2004 では，一度のパージでキャッシュの三分の一が解放されていたので，キャッシュのサイズが大きければ，それだけ大々的なパージが行なわれていました。4D v11 は，前述したように 10%なので，もっと柔軟です。そのことも理由のひとつですが，これまでのバージョンでは，キャッシュサイズをむやみにおおきく設定しないように推奨されていました。キャッシュがおおきければ，それだけ目的のオブジェクトに到達するまでの連結リストが長くなり，かえってパフォーマンスが低下してしまったからです。新しい v11 のキャッシュマネージャーでは，そのような問題はありません。

第四部・トリガを正しく理解する

違いを知ることは重要

4D Server のトリガは，サーバーサイドのツインプロセス，それもコオペラティブスレッドのほうで実行されるコードです。このプロセスは，「サーバーで実行」メソッドプロパティが有効にされたメソッドと同じコンテキストで実行されるのが特徴ですが，ひとつだけ違いがあります。トリガのスコープは，そのトリガが所属するテーブルのカレントレコードだけであるという重要な違いです。

4D Desktop つまりローカルデータベースでは，トリガのスコープがカレントセレクション，カレントレコード，テーブルの READ WRITE 状態，レコードのロック，トランザクションの状態，プロセスセットなど，プロセスコンテキスト全般をそのトリガを起動したプロセスから継承します。4D 2004 も同様でした。

4D Server の場合，トリガプロセスがそのトリガを起動したプロセスからカレントセレクションや他のテーブルのカレントレコードを継承することはありません。実際のところ，カレントセレクションや他のテーブルのカレントレコードに関していえば，トリガプロセス（つまり「サーバーで実行」メソッドのプロセス）は，独自のコンテキストを持っています。

もうひとつ，4D 2004 と違うのは，4D v11 SQL 以降，複数のトリガ（当然，それぞれの別々のテーブルのもの）が同時に起動するようになったということです。

トリガの使用は避けるべきなのか

そのような極端な見方をする必要はありませんが，トリガの使用に関しては慎重に判断をするべきです。トリガには制約があります。トリガはコオペラティブスレッドで実行される以上，できるだけ素早くメソッドを完了することがたいせつです。そのためには，汎用的なトリガなどではなく，極めて特化されたメソッドをテーブルごとに記述したいと思うことでしょう。トリガの処理に関して事前にできることがあれば，トリガに突入にしてからではなく，「サーバーで実行」メソッドを活用してあらかじめ準備しておくことにより，さらに時間を節約することができます。またコマンドでシーケシャルナンバーを代入するのではなく，自動インクリメントプロパティを使用してデータベースに番号を振らせることも有効です。なお，v12 では自動 UUID を使用するという選択肢も用意されています。

第五部・スケジューラーとうまく付き合う

スケジューラーの役目

4D のデータベースエンジン DB4D は，マルチスレッド用のものとして完全に書き直されました。しかしながら，4D ランゲージはスレッドセーフなプログラミング言語ではありません。メソッドの 4D コードは，引き続き単一のコオペラティブスレッドですべて実行される必要があります。このコオペラティブプロセスの実行サイクルを管理しているのがスケジューラーです。

アプリケーションを実行している間，スケジューラーは定期的にシステムをコールしており，4D に関係があるイベントがないかどうかを絶えずチェックしています。システムコールの間隔は，4D がアイドルのときには若干長めの 8 ティック，ビジーのときには最短の 0 ティックに設定されています。（環境設定またはデータベースパラメーターで変更することもできます。）スケジューラーは，イベント監視中，4D のプロセスに関係があるイベントを拾ったときには，それを待機しているプロセスのために取り置き，イベントが何も返されないか，前述のタイムアウト（0 ないし 8 ティック）が経過するまでこのことを続けます。

スケジューラーと CPU 使用率

スケジューラーは，イベント監視サイクルを抜けると，遅延あるいは停止されていないアクティブなプロセスそれぞれに対し，最低 1 ティックずつの実行時間を与えてゆきます。これにより，4D は擬似的なマルチタスクを実現しているのです。注目に値するのは，アクティブではないプロセスやセマフォ待ちのプロセスには実行時間が与えられないこと，またプロセスに与えられる時間はたとえそのプロセスが実行時間を必要としていなくても一律最低 1 ティックであるということです。結果的に，4D はこのサイクルを完了するまでにアクティブなプロセス分のティック数，60 プロセスなら 1 秒を要することになります。すべてのプロセスに時間を配分した後，所定の時間（これも環境設定あるいはデータベースパラメーターで変更できます）が経過していなければ，もう一度それぞれのプロセスに 1 ティックずつ実行時間が与えられてゆきます。システムに制御が返されない以上，この状態の 4D は何もしていなくても CPU 使用率が非常に高いアプリケーションになります。

スケジューラーの設定をいじれば，確かに CPU 使用率が増減しますが，CPU 使用率だけでアプリケーションのパフォーマンスを判断することは避けるべきです。第一に，CPU 使用率は一定期間内における CPU 使用時間の平均であり，スナップショットに過ぎません。第二に，システムが計測しているのは，スケジューラーの CPU 使用率であって，個々の 4D プロセスがそれを有効活用できているかどうかはまた別の問題です。パフォーマンスを判断するのであれば，特定の処理を終えるまでに要する時間そのものを調べることがたいせつです。

スケジューラーに制御が返らないとき

スケジューラーに制御を返さないプロセスは，他のプロセスすべてに迷惑をかけます。これにはユーザーインタフェースも含まれます。インタプリタモードであれば，メソッドを一行実行するたびに制御がスケジューラーに戻りますが，コンパイルモードではそのような機会がありません。ループ文が実行されている場合など，スケジューラーが完全に停止する可能性さえあるので，IDLE（プラグインならば PA_Yield）コマンド，DELAY PROCESS 0（プラグインなら PA_YieldAbsolute）コマンドを適宜コールする必要があります。

データベースパラメーターを変更してパフォーマンスを改善できるのは，極めて特殊で限定なケースだけであり，むしろメソッドを分析して不要なプロセスに必要のない実行時間が配分されないようにすることが肝要です。逆に，アプリケーションが恒常的に異常な CPU 使用率を示すときは，スケジューラーの設定を疑ってください。

理論的には，プリエンプティブスレッドとコオペラティブスレッドが同一の CPU コアで実行されることもありますが，通常はすでにコオペラティブスレッドで忙しいコアにそれ以上の負荷が載せられることはないので，何よりも 4D のプロセス連携を最適化し，コオペラティブスレッドを極力最適化することが効果的です。

第六部・スタックの話

クライアントプロセスのプリエンプティブスレッド

4D Server は，クライアントプロセスごとにプリエンプティブスレッドを作成し，そのスレッドにメモリスタックをアロケートします。このスレッドは，おもにデータベースアクション（ORDER BY など）を処理するために使用されるものです。プリエンプティブスレッドのスタックサイズは，システムによって若干の差異はありますが，基本的に 1MB です。

その他のプリエンプティブスレッド

4D のプリエンプティブスレッドは，クライアントプロセスのツインプロセスだけではありません。カーネルにもプリエンプティブスレッドがあります。DB4D サーバー，SQL ネットセッションマネージャー，SQL ネットコネクションマネージャー，および予備 SQL スレッドなどはいずれもプリエンプティブスレッドです。予備 SQL スレッドのスタックサイズは，データベースパラメーター53 番で取得あるいは設定することができます。フラッシュマネージャーおよびインデックスビルダーもプリエンプティブスレッドですが，これらのスタックサイズはハードコーディングされており，512MB です。

クライアントプロセスのコオペラティブスレッド

4D Server は，クライアントプロセスごとにコオペラティブスレッドも作成し，そのスレッドにもメモリスタックをアロケートします。このスレッドは，おもに 4D ランゲージを実行するために使用されるものです。ランタイムエクスプローラーなどのローカルプロセス，4D Remote の管理画面，On Exit データベースメソッドなどはいずれもコオペラティブスレッドです。特殊なケースとして，Web プロセスはそれぞれグローバルプロセスですが，Web サーバー自体はローカルプロセスです。たとえば，4D Remote で Web サーバーを起動した場合，Web プロセスごとにツインプロセスが 4D Server に作成されることになります。また，SQL サーバーはプリエンプティブスレッドですが，SQL の FN キーワードでメソッドをコールしたり，データベースアクションの結果としてトリガを実行したりするようなことがあれば，4D ランゲージを実行するためにコオペラティブスレッドが作成されることになります。重要な点として，そのようなプロセスのコオペラティブスレッドは，呼び出し元が明示的にログアウトするまで停止された状態で保持され残されています。

その他のコオペラティブスレッド

一部のコオペラティブスレッドは，スタックサイズが‘4STK’リソースに保存されています。その他のものは，スタックサイズがハードコーディングされています。いずれにしても，値はベースとなるサイズであり，実際にアロケートされるスタックサイズはこれに一定の係数をかけたものになります。Windows の場合，実際に使用されるスタックサイズは，値に 40KB を足したものです。Mac の場合，実際に使用されるスタックサイズは，値に値の 50%を足し，さらに 1KB を足したものです。New Process および Execute on server コマンドに渡されたスタックサイズも，同じように調整を受けます。たとえば，スタックサイズに 0KB を要求した場合，ベースの値は 512KB となり，前述の係数をかけた結果，Windows では 552KB，Mac では 896KB となります。1KB 以上 16KB 以下を要求した場合，ベースの値は 16KB となり，係数によって Windows では 56KB，Mac では 152KB になります。17KB 以上を要求した場合，要求値がそのままベースになり，係数によって実際のサイズが決まります。負の値は，符号付き整数の計算が働いて 4GB のスタックを要求することになりますから，絶対にそのようなことはしないでください。

‘4STK’リソースの 1 番から 4 番には On Event Call，On Serial Port Call，デザインモードで実行したメソッドやマクロから実行したメソッドの各プロセスおよびアプリケーションメインプロセス，メニューから実行したメソッドのプロセススタックサイズのベース値が保存されており，デフォルトは 512KB です。5 番および 8 番はそれぞれサーバーのツインコオペラティブスレッドのプロセスと Web/SOAP プロセスのスタックサイズのベース値であり，

256KB となっています。6 番と 7 番はバックアップと復元に関する古い設定です。9 番はサーバーのイベントループ, 4D Server のユーザーインタフェース, クライアントマネージャー（起動時に開始し, コオペラティブ接続を管理するプロセス）, ランタイムエクスプローラー, および EXECUTE ON CLIENT 用に登録されたクライアントのプロセス（これもグローバル）のスタックサイズのベース値であり, やはり 512KB です。

その他のコオペラティブスレッド, つまり内部タイマープロセス, デザインプロセス, コンパイラプロセス, 管理画面, On Exit データベースメソッドのプロセス, Web サーバーのメインプロセス, SQL が自動するメソッド実行用プロセスのスタックサイズは, いずれもベース値が 512KB にハードコーディングされています。

プロセスごとに一定のコオペラティブスレッド, そしてそのためのスタックサイズが必要であるということは, 多数のクライアントが接続するようなシステムでは, スタックサイズの管理が必要な場合もあるということです。

第七部・パラダイムシフト

その役目を終え, もはや必須ではなくなったもの

MODIFY SELECTION および DISPLAY SELECTION, ラベルエディタ, プラグインなどがこれに相当します。4D が提供する古いデフォルトユーザーインタフェースは, ユーザーの期待には沿わないかもしれません。リストフォームなどはその典型的な例です。サブフォームやリストボックスは, ずっと良いユーザーエクスペリアンスを容易に提供することができるので, v12 以降, 特に積極的な使用が推奨されています。リスト表示をデザインする場合, まずリストボックスを考慮してください。リストボックスは多くの機能が内蔵されたオブジェクトであり, 標準的なルックアンドフィールを有しています。リストボックス(配列またはセレクション)で用が足りるのであれば, 迷わずに選択してください。リストボックスでは役不足に思える場合, サブフォームを検討してください。

ラベルエディタは, 4D でもっとも古い部類の属するエディタのひとつです。すぐに使用できる既成のエディタは便利であり, 間違いなく必要ですが, 一方で誰もがユーザーインテフェースをカスタマイズしたがるというのもまた事実です。4D v12 では, そのような定番のエディタをコンポーネントとして提供します。これらはプラグインエリアではなく, 普通の 4D コードだけで作られています。またオープンソースです。今後, ラベルエディタやプラグインエリアなどは, そのようなコンポーネント型サブフォームに取って代わられてゆくことでしょう。

プラグインについても同様です。4F の機能を拡張する場合, まずはコンポーネントとして拡張機能を設計することを考慮してください。4D v12 では, PHP インタプリタが組み込まれました。その目的は, よくある処理を 4D に実装したい場合, C++でプラグインを自作するよりは, むしろ PHP で既存のソリューションを流用したほうがはるかに生産的であるからです。ビジネスアプリケーションで何らかの問題に取り組まなければならない場合, その問題はすでに PHP コミュニティでは解決されている場合が少なくありません。4D に機能を追加する上で, プラグインの作成は最後の手段であるべきです。

公式に廃止予定が発表されており, 使用を控えるべきもの

4D は 20 年以上の歴史があるアプリケーションです。この間, 情報技術の世界は飛躍的な進歩を遂げただけでなく, 根本的な考え方にも変化が起きました。とりわけ Mac の変化はめまぐるしいものでした。4D は, 新しい技術に対応し, よりおおきな画面, 新しいプロセッサ, 新しい OS にも合わせていかなければなりません。68K, PPC, x86, OS 9, OS X とこれまで対応を重ねてきたように, 今後は Carbon から Cocoa, そして 64 ビットの対応を予定しています。4D のコードはその変化から遮断し, できるだけ互換性を維持することようにいつでも心がけていますが, デベロッパ様の協力がどうしても欠かせない場合があるというのもまた事実です。

その代表的な例が 4D Draw です。4D Draw は QuickDraw を使用しており, Windows では Altura によってエミュレーションされています。最新の API で QuickDraw のエミュレーターを開発するという選択肢はなかったのでしょうか。ひとことでいえば, それはあまり追求する価値のない目標です。QuickDraw は, 座標系に整数を使用しており, フォントを番号で管理し, 塗りパターンを模様で表現しています。いずれも今日のコンピューターグラフィックでは廃れた考え方です。その上, 4D Draw のコーディングは非常に煩雑なものになります。多大な努力を傾けてエレガントではないものをあえて再現することにどれだけの意義があるのでしょうか。その上, そのようなやり方では, 4D Draw の制限が少しも改善されません。4D Draw は最新のピクチャフォーマットが開けない, 4D Draw ドキュメントを一般の画像処理ソフトで編集できない, などといった制限のことです。4D が SVG を選択した理由はそこにあります。SVG は, 現在もっとも一般的なベクトル画像フォーマットです。4D の SVG レイヤーは, Mac の CoreGraphics, Windows の GDI+を使用しています。4D v12 では, CoreText も使用しています。SVG は標準テキスト形式なので, どんなソフトでも開いて編集することができます。それでいて, 透明度, 重ね合わせ, 回転, 変形, 組み込みなど, 洗練されたグレフィックエデフェクトに対応しており, 4D のピクチャコマンドで操作することもできるのです。たとえば, SVG と PNG を合成したり, ピクチャコマンドで SVG を変形させたりすることもできます。4D v12 では, 簡単なエディタを MiniDraw コンポーネントとして提供する予定です。

特筆すべき点として，2010 年の 1 月，Internet Explorer の開発チームが SVG ワークグループに参加を表明しました。IE9 では SVG がサポートされるのかもしれません。こうした事実もまた，SVG という選択が正しかったことを証明しています。

関連するトピックですが，PICT フォーマットも使用を中止するべきです。PICT は QuickDraw の独自フォーマットであり，QuickDraw とともに消滅する運命にあります。4D は，Altura あるいは QuickTime の力を借りて Windows で PICT を処理しています。4D データベースを変換した場合，変換の過程で既存の PICT データが何らかの処理を受けることはありません。CONVERT PICTURE コマンドを使用し，適切なフォーマットにこれを変換する作業が必要です。ベクトル画像であれば，PDF や EMF，ビットマップであれば PNG あるは JPG が妥当ですが，PICT には合成画像も含まれるため，一概にフォーマットの対応関係を挙げることは困難です。

そのようなわけで，4D Draw や 4D Chart の代わりには SVG が推奨されますが，グラフやチャートの処理についていえば，優れた Flash/JavaScript のライブラリが多数存在するので，それらを Web エリアに表示するという選択肢もたいへん有効です。

もうひとつ廃止予定が発表されているものに 4D Open が挙げられます。4D Open プラグインは v11 でも動作しますが，インタプリタモードに限定されます。4D Open は，かつてデータベースの同期と再現を可能にするものとして紹介され，またそのような用途で使用されてきました。しかしながら，4D v11 ではデータベースエンジンが新しくなり，4D Open が使用する API とは互換性がなくなりました。4D Open は Unicode ではなく，Altura に依存しており，古いデータタイプを使用しています。それでは，新しいデータベースエンジンと Unicode と新しいデータタイプに対応した 4D Open を Altura に依存しないで新作するという選択肢はなかったのでしょうか。4D Draw と同様の回答になりますが，4D Open は非標準的な API であり，コードがエレガントではありません。4D にはいまや優れた通信方法がいくつも用意されており，4D Open のような方法を使用しなくても 4D 同士を対話させることができます。DOM Parse XML Source を使用すれば，リモートサーバーから XML を受信することができますし，Web サービスも非常に強力です。SQL パススルーという方法もあります。4D v12 では SQL コマンド一発で同期と復元ができるようになります。SQL，RPC，SOAP，XML はいずれも標準のプロトコルであり，高い互換性があります。4D Open と同等のことがより標準的でエレガントな方法でできる以上，独自のプロトコルで雑然とした 4D Open をあえて開発するというのは道理に合いません。むしろ，前述した既存の方法を一層洗練されたものにし，発展させてゆくことのほうがはるかに建設的であると判断しました。

最後に，4D 2004 で廃止予定が発表された古いアピアランスも注意を要する問題です。Windows 98，Windows NT，Mac OS 7，Mac OS 9 などのアピアランスは，内部的にはカスタムコードと QuickDraw および Altura で描画されています。こうしたオブジェクトを平然と表示しているアプリケーションは，古くて醜い印象を与えます。業務用のアプリケーションは，Microsoft および Apple が公開しているガイドラインをしっかり守り，スマートなルックアンドフィールを提供して然るべきでしょう。