

より速い,より効率のいいプログラムとは?

プログラムの 高速化/最適化技法の 検証

松浦健一郎

速いプログラムはいいものです。機能が同じで速いプログラムと遅いプログラムがあったら、誰でも速いプログラムのほうを使うでしょう。最近是非常にマシンが高速になりましたが、それでもなお動作が重いプログラムがたくさんあります。複雑な機能を持つプログラムはそれなりに遅くなってしまふものですが、それを差し引いても遅すぎるプログラムが見受けられます。

プログラムはプログラマの努力しだいでもかなり速くなります。なにより、プログラムの高速化はプログラミングの大きな楽しみのも1つでもあります。これは初心者にも上級者にも共通です。もしもプログラミン

グ言語を学んだばかりでも、思ったとおりの動作をするプログラムがそこそこ書けるようになったら、次は速いプログラムを書くことに挑戦する段階です。熟練したプログラマの中には、高速化が3度の飯より好きという人も少なくありません。

では、プログラムを速くするにはどんなコーディングをするべきなのでしょう。プログラミングの本には高速化について述べたものがありますし、ある程度経験を積んだプログラマに尋ねればいろいろとノウハウを教えてください。

しかし、ここに難しい問題があります。まず、高速化テクニックの中には最近のマ

シンには通用しない「古いテクニック」が混じっています。また、プログラマの「思い込み」や「勘違い」も多いのです。こうコーディングすれば速くなる「はず」という方法はいろいろと知られていますが、実際に速くなるのかどうかをきちんと確かめたいうえで実践していることはまれです。

そこで本稿では「プログラムの高速化」に焦点を絞り、速いプログラムを書くためのテクニックについて考察します。「わずかなコーディングの違いが意外な結果につながる」という題材についてじっくり検証するので、初心者から上級者までお楽しみいただければ幸いです。

part-1 高速化の基本

PART1ではプログラムを高速化するための基本的な考え方を解説します。重要なのは「思い込み」や「勘違い」をしないようにきちんと手法の検証をすることと、コンパイラの最適化機能をうまく活用することです。

速いプログラムを書こう

高速化はプログラミングの醍醐味の1つです。単にプログラムが「動けばいいや」と

いうだけではなく、常に「より速く、より美しいプログラムを書くこと」を念頭に置いてコーディングするのが、プログラマのあるべき姿でしょう。

速いプログラムにはいろいろなメリット

があります。主な利点は次の2つです。

- ・レスポンスが機敏なので、快適に操作できる
- ・遅いプログラムに比べて、同じ時間により多くのデータを処理することができる

最近のマシンは昔に比べると非常に高速になりましたが、相変わらず遅いプログラムもあります。ツール系プログラムの場合、プログラムが速いことは仕事をスムーズに進めるうえで重要です。またゲーム系プログラムの場合、速いプログラムは遅いプログラムよりも高い画質や音質が得られたり、滑らかな動きが実現できたりします。

プログラムが速いことは、そのプログラ

ムの価値を大きく上げます。機能が多くて遅いプログラムよりも、機能が単純で速いプログラムのほうが快適に使えることもしばしばです。とくに携帯性を重視した小型PCや携帯端末あるいはゲーム機などは、通常のデスクトップPCに比べて性能が低めです。こういったマシンで動かすプログラムを作るときには、動作を軽くすることに特別な注意を払わなければなりません。

「思い込み高速化テクニック」の罠

速いプログラムを書くためのテクニックはいろいろとあります。たとえば、

- ・メモリアクセスをなるべく少なくする
- ・マクロやインライン関数を使う
- ・ループ中で変化しない変数をループ外に追い出す

といったものです。こういったテクニックは本やWebでも紹介されていますし、学校や職場で教わることもあります。

しかしこれらのテクニックをうのみにして、「こう書けばプログラムは速くなるはず」と信じ込むのは危険です。テクニック

の中には、昔のマシンでは通用しても今の高速なマシンでは通用しないものがあります。また、高速化になると思っていたのに実際にはあまり速くならなかったり、悪いときには遅くなってしまふような書き方もあります。

「こう書けば速くなるはず」という情報は多いのですが、「このテクニックを使ったら実際にこれだけ効果がありました」という情報は案外手に入らないものです。また残念ながら、思い込みや勘違いのテクニックも少なくありません。

したがって、見聞きした情報をそのまま使うのではなく、やはり一度はきちんと自分の手で有効性を確かめるべきでしょう。少々めんどうではありますが、本当に速くなるのかどうかわからないテクニックを使うよりもずっと効果的です。

生産性と高速性のバランスをとる

プログラムを書くときには、生産性と高速性のバランスも大事です。生産性と高速性とは相反することがあるので、プログラ

マは最良のバランスを見つける必要があります。

たとえば、C++のクラスはプログラムの生産性を向上しますが、速度の点ではただの構造体を使ったほうが有利なことも多いのです。ではC++はいっさい使わずにCを使うべきかといえば、それは状況によります。「生産性がよいC++を使いつつ、C++のできる範囲で高速化を図る」のも1つの選択です。たとえクラスを使っても、クラス構造をシンプルにしてメンバ関数のインライン化などに気を使えば、著しく速度が低下することはありません。

「正しい高速化テクニック」を見極めることは、生産性と高速性のバランスを良好に保つうえでも重要です。高速化するつもりで念入りに手間をかけても、効果がなければ生産性が落ちるぶんだけ損だからです。本当に必要などころにだけ手間をかけるべきでしょう。

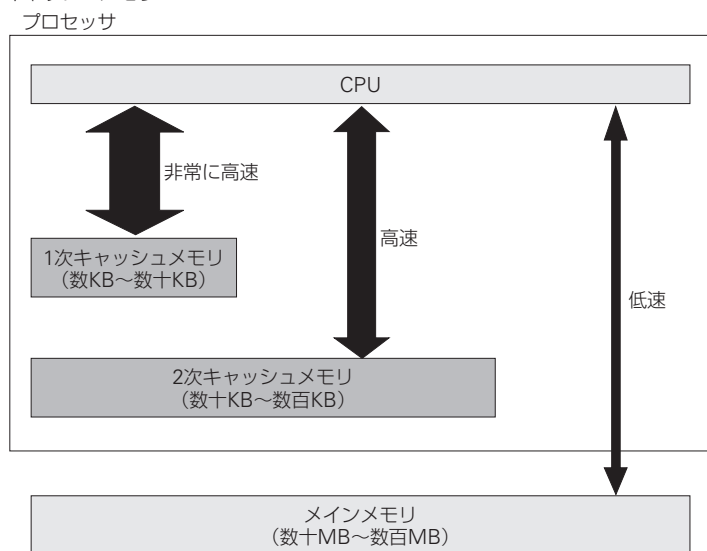
またプログラミングを始める以前に、生産性と高速性のバランスがとれたプログラミング言語を選ぶことも大切です。たとえば速度の点ではC/C++のほうがJavaよりも優れていますが、Webサーバ上で運用するにはJavaが便利です。目的に合わせて、生産性と高速性の両方を達成できる言語を選択する必要があります。

高速化の基本は実行時間の測定

プログラムの高速化を行うときには、実際にプログラムが速くなったかどうかを確かめることが肝心です。いちばん効果的かつ直接的な確認方法は、プログラムの実行時間を計ることです。

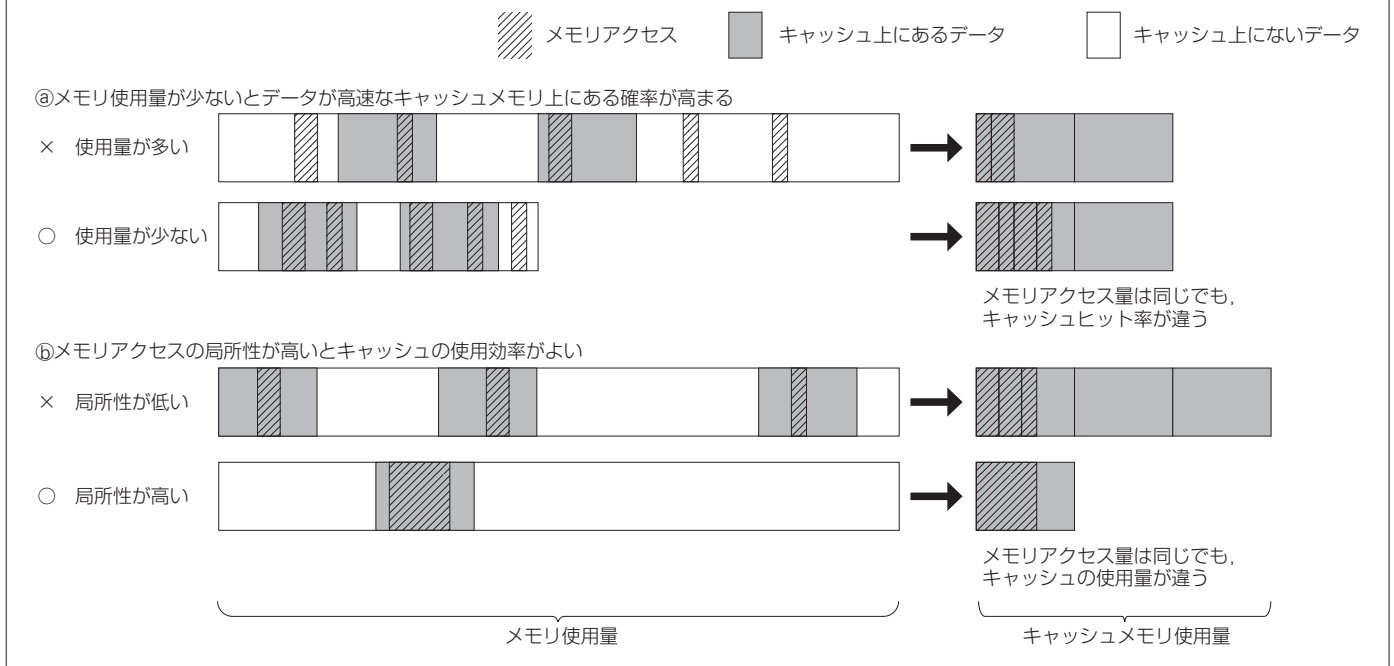
今のマシンはプロセッサやメモリが昔よりもずっと複雑になったため、「マシン語(アセンブリ)でこう書けば絶対に速い」とはいえなくなりました。同じ系列のプロセッサでも製品によって実行方式が異なるため、製品ごとに得意分野と不得意分野があります。もちろん、OSの違いによってもプ

Fig. 1 キャッシュメモリ



※キャッシュメモリの構造やメインメモリとの接続形態はプロセッサによって異なる。この図は2次キャッシュまでがプロセッサに内蔵されている場合。2次キャッシュがプロセッサの外にあるシステムや3次キャッシュを持つシステムもある

Fig. 2 キャッシュメモリが有効に働く状況



プログラムの速度は大幅に変わります。

また、最近のマシンのほとんどはキャッシュメモリ^[註1]を利用しているので、キャッシュが有効に働くかどうか(キャッシュがきくかどうか)によって発揮する性能がまったく違います(Fig. 1)。しかしキャッシュの挙動は複雑なので、実際にプログラムを動かしてみないとどの程度の性能が出るのかわかりません。しかもプロセッサによってキャッシュの容量や方式が微妙に違います。

したがって重要なのは、実際の運用環境に近い環境で実行時間を測定することです。実行時間を計るための具体的な方法(時間計測用の関数)については、PART2で解説します。

[註1]キャッシュメモリ(キャッシュ)低速なメインメモリへのアクセスを減らすために設けられた、高速で小容量のメモリ。使用頻度の高いデータを蓄積して、すばやくアクセスできるようにする。

メモリ使用量にも注意する

メモリ使用量が少ないことは、プログラムの高速化につながります。速いプログラムを書くには、実行時間と同様にメモリ使用量にも気を配ることが大切です。メモリ使用量が高速化につながる理由は、キャッシュとページング^[註2]です。

キャッシュはメモリ使用量が少ないほど有効に働き、プログラムの実行速度を引き上げます。これは、データ容量が小さいほど、キャッシュメモリ上にデータがある確率(キャッシュヒット率)が高いからです(Fig. 2-③)。

また全体としてはメモリ使用量が多くて、一度に使うデータが狭い範囲に集まっていれば、やはりキャッシュは有効に働きます。この状態を「メモリアクセスの局所性が高い」といいます(Fig. 2-④)。キャッシュは近い位置にあるデータをまとめて記録するため、アクセスの局所性が高いほどキャッシュの使用効率がよくなるのです。

Fig. 3 メモリリーク

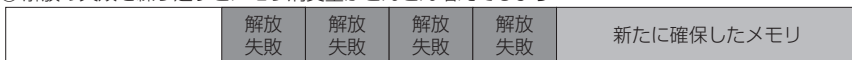
①データファイルを読み込むためにメモリを確保する



②データファイルを閉じたので不要なメモリを解放するが、一部の解放に失敗してしまう



③解放の失敗を繰り返すとメモリ消費量がどんどん増えてしまう



一方、メモリ使用量が極端に多いプログラムではページングの問題が発生します。ページングはディスクアクセスを伴うため、頻繁に起こるとプログラムの実行速度を著しく落とします。幸い最近のマシンでは、特別大きな画像データやムービーデータを扱わないかぎりページングに悩まされることはありませんが、ムダなメモリ使用は極力避けるべきです。

もう1つメモリに関して注意が必要なのは、使用済みメモリの解放です。たとえば多くのツール系プログラムでは、データファイルを読み込むときにメモリを確保します。この確保したメモリが不要になったとき、適切に解放するか再利用するかしないと、マシンのメモリ資源が徐々に減少してしまいます。これを「メモリリーク」と呼びます (Fig. 3)。

Fig. 4 タスクマネージャ



いちばん危ないのは、「メモリを解放したつもりだけでも実は解放漏れがあった」というパターンです。とくにJavaでは暗黙

にメモリが確保/解放されるうえにメモリ使用も激しいので、プログラミングの際には特別な注意が必要です。

プログラムのメモリ使用量を計測するにはOSの機能を使うのが簡単です。たとえばWindows NT系 (NT4/2000/XP) の場合、Fig. 4のようなタスクマネージャが利用できます。タスクマネージャの起動は **Ctrl+Shift+Esc** キーです。タスクマネージャは、プログラム (プロセス) ごとの使用メモリ量に加えてCPU占有時間なども表示します。実行時間の大きな計測にも便利です。

[注2] ページング

ハードディスク上にメインメモリ上のデータの一部を退避 (スワップ) させ、必要に応じてメインメモリ上に書き戻すことによって、メインメモリの容量があたかも増えたようにするOSの機能。この「あたかも増えたようなメモリ領域」のことを「仮想メモリ」という。

遅い部分を重点的に高速化する

プログラムを速くするにはそれなりに手間がかかります。プログラム全体を高速化できればということはありませんが、仕事でも余暇でもプログラミングに使える時間は限られているのが普通です。そこで、プロ

コラム 1

速いマシンはプログラマをダメにする!?

これは半分冗談ですが、半分は本当です。プログラムの開発作業自体は、当然ながらマシンが速いほうが早く進みます。しかし速いマシンにはデメリットもあります。

まず速いマシンでは、自分の作ったプログラムの速さ (遅さ) がよくわかりません。多少遅いプログラムでもまともな速度で動いてしまうので、高速化しようという気にならないのです。その点遅いマシンを使っていると、いやでもプログラムの遅さが目につきます。普通のプログラマならば、自分の書いたプログラムが遅かったら放っておけないはずです。

それから、コンパイルがあまり速いのも考えものです。「ちょっといじってコンパイル、ちょっといじってコンパイル……」というのは便利なのですが、コンパイルを繰り返すばかりで実際の作業はあまり進まない、ということにもなりかねません。ときには手を休めて、プログラムの構造や動作をよく頭の中で考えることも大事です。

ということで、もしも遅いマシンが手元にあれば、それを有効活用するのがお勧め

です。わざわざ遅いマシンで開発を行うのはさすがに疲れますが、完成間近のプログラムを遅いマシンで動かしてみるのはいいことです。速いマシンでは気づかなかったプログラムの欠点が見えてきます。

遅いマシンに関する話を1つ。友人が速いマシンでプログラミングをしていましたが、ある日そのマシンが故障してしまいました。彼は修理期間中にプログラミングできないのが耐えられず、手元にあった10倍くらい遅い古いマシンで作業を再開したそうです。コーヒーが入れられるほどコンパイルに時間がかかったようですが、「遅いマシンはいいね。自分のプログラムを本気で高速化しないと耐えられない。俺は速いマシンに浸かってダメになってたよ (笑)」といいながら楽しんでいました。

彼は「MSX (昔の8ビット機) のCコンパイラは平気で一晩かかったよ。それに比べれば天国」という猛者なので、同じ状況を誰もが楽しめるかどうかは微妙ですが、遅いマシンを使うと何かしら発見があるものです。ぜひ一度、暇なときにお試しを。