性能問題リスクへの早期対処
～アスペクト指向を適用した処理性能分析の実践～

こまつ　せいき
小松　　清希

株式会社エクサ　技術推進本部　技術推進室
シニアＩＴアーキテクト

原稿量
本文　　１３，８００字
要約　　１，２００字
図表　　１１枚

＜要　約＞

ウォーターフォール型情報システム開発において、システムテスト局面で明らかとなる性能問題は大きな手戻りとなるリスク要因である。性能問題をより早い局面で検証し、タイムリーに対処できるようにするべきである。

一般にシステムテスト局面では、本番環境に近いシステム構成でシステムテスト用の大量のデータを用いてスループットを計測することができる。しかし結合テスト局面では業務トランザクションを1回起動する程度のことしか行えない。0.5秒や1.0秒といった短期間のライフタイムの中でどの箇所に問題があるかを知り、改善してゆくことは容易ではない。この課題を考察した。

まず、プログラム中の状況を正確に把握するために適切なトレースログを出力したいと考えた。しかし、大量のログ出力コードを適切な場所に、間違えずに埋め込む作業はコストを要する。また、詳細なログを出力する大量のコードをソースコードに直接記述するとプログラムの見通しを悪くし、保守性を低下させる。トレースログの出力コードはソースコードに対して直接記述せず、必要な場合のみ付加し、調査の完了後はきれいに削除できることが望ましい。

そこで筆者はアスペクト指向プログラミング(Aspect Oriented Programming: AOP)に着目した。AOPではプログラム中の幾つもの箇所にかかわる共通の処理(AOP の用語で「横断的関心事」と呼ぶ)を本体とは分けて準備しておき、条件に対応して横断的関心事を自動で適用させる制御が可能となる。トレースログの出力はまさに横断的関心事に該当する。AOP はアプリケーションフレームワークやEclipse 等の開発ツールの中で既に利用されている技術である。例えば宣言型のトランザクション制御やエラー制御が該当する。すなわち、我々は間接的に AOP の恩恵を受けていると言える。しかし情報システム開発において AOP を直接利用した事例は他にあまり目にしない。

本稿では結合テスト局面相当の状況で実際に AOP プロダクトである AspectJ を用いて業務トランザクションの性能を動的に分析する手法を明らかにする。その手法を試行して分析した結果、次の性能問題を発見し改善した事例を紹介する。

・COBOL の知見で最適化したテーブル構造が、Java では性能劣化を導いたこと

上記の試行において、AspectJ に依存した固有の問題も経験した。それらの中から次の事項をピックアップし、それぞれどのように対処したかもあわせて示す。

1. AspectJ がクラスへ追加するフィールドの影響で、フレームワークのリフレクション処理で障害が発生
2. 所要時間算出の精度の問題
3. AOP でトレースログ出力を織り込む対象の選択において、考慮するべき事項
4. ガベージコレクションタイミングの調整
5. クラスローダーキャッシュ、リフレクションキャッシュに対する配慮

目　次

1. はじめに

業務ロジックにフレームワークと業務共通部品を結合して動作させた場合の処理性能上の問題を早期段階で動的に分析するために、アスペクト指向プログラミングツールを活用できそうだ。本稿ではこの手法を実際に試行し、判明した業務処理の構造上の問題例を紹介する。また、アスペクト指向プログラミングツールに起因して生じた実施上の問題と対策をあわせて示す。

本稿の構成は次の通りである。2 章で序論として情報システムの性能問題に取り組むべき時期と検討対象とするシステムの概要を述べる。3 章では性能問題へアプローチするソリューション候補としてアスペクト指向プログラミングについて述べる。4 章では性能問題の分析におけるアスペクト指向プログラミングの適用とそれから得た知見を述べる。最後の 5 章では今回の成果をまとめる。

2. 序論

本章では、情報システム開発における性能問題の取り組み時期にふれ、以降の章で述べる検証の前提を説明する。

2.1 情報システムの性能問題にいつ取り組むか

情報システム開発プロジェクトにおいて、性能テストの実施時期はシステムテスト局面に計画することが一般的である。しかしウォーターフォール型情報システム開発において、下流局面で発見される問題の対応は大きな手戻りとなるリスクがある。

情報システムの性能上のさまざまな問題を性能問題と呼ぶこととする。性能問題を可能な範囲で事前に確認しつつ開発を実施することができれば、システムテスト局面ではじめてそのような問題を発見し局面をさかのぼって対応する事態を回避、または軽減することができるはずである。

2.2 想定する開発案件の状況

全体で 1000 人月規模程度の架空の情報システム開発プロジェクトにおいて、エクサ(以下、当社とする)はバックエンド業務サブシステムの開発を請け負うことを想定する。そのプロジェクト全体には 10 数社が参画し、Java 言語を用いてサブシステムごとに開発を担当するものとする。サブシステムというのは大きな業務ごとの Web 画面とフロント業務のサブシステム、バックエンド業務のサブシステム、フレームワーク、業務共通部品といった粒度のシステム構成単位である。これを図 1 に示す。当社が担当するバックエンド業務サブシステムが今回の性能分析の対象である。

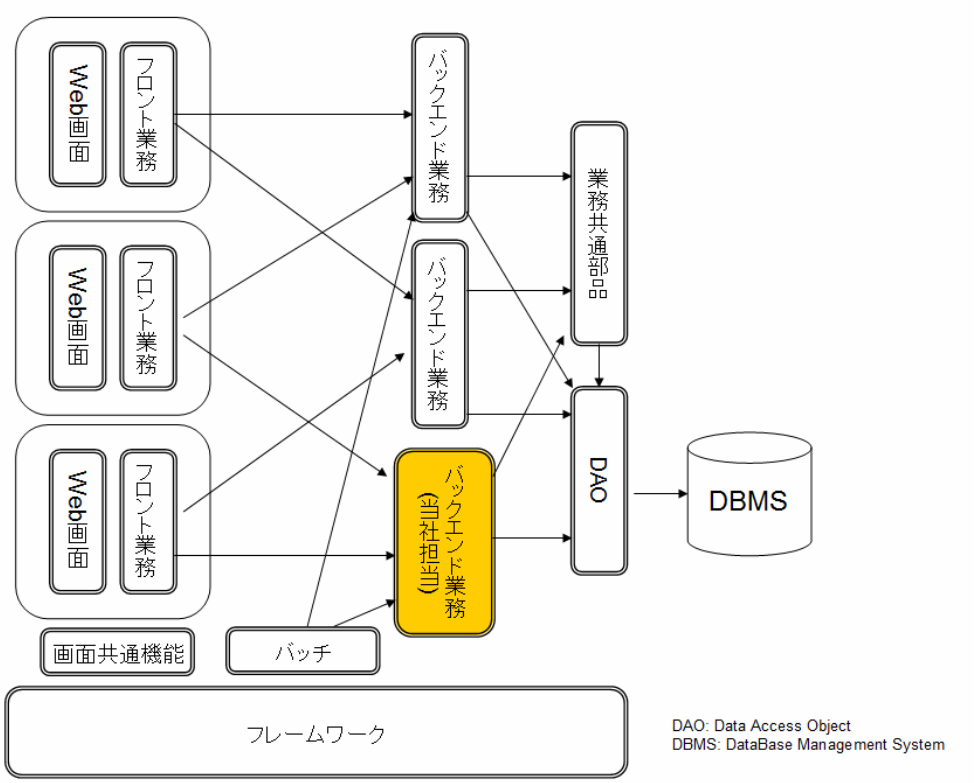

図 1. 想定する担当サブシステム

時期としては、サブシステムごとの単体プログラム製造と単体テスト局面が完了し、サブシステムごとに順次結合し確認していく結合テスト局面とする。一般にこの局面を経過すると、山を越えプロジェクトのメンバーを順次減員してゆくが、システムテスト局面で性能問題が発生した場合に対応可能なスキルの高い要員をリリースすることに心配があり、バックエンド業務サブシステムの中に性能問題が存在するかどうかを結合テスト局面で確認したい状況を想定する。

2.3 分析するトランザクション

分析する業務トランザクションでは、図 2 のようにバックエンド業務から、フレームワークのデータアクセスオブジェクト(以下、DAO と称する)を用いてデータベースへアクセスする。バックエンド業務から業務共通部品を呼び出すこともあり、業務共通部品もその内部から DAO を呼ぶことがあるものとする。

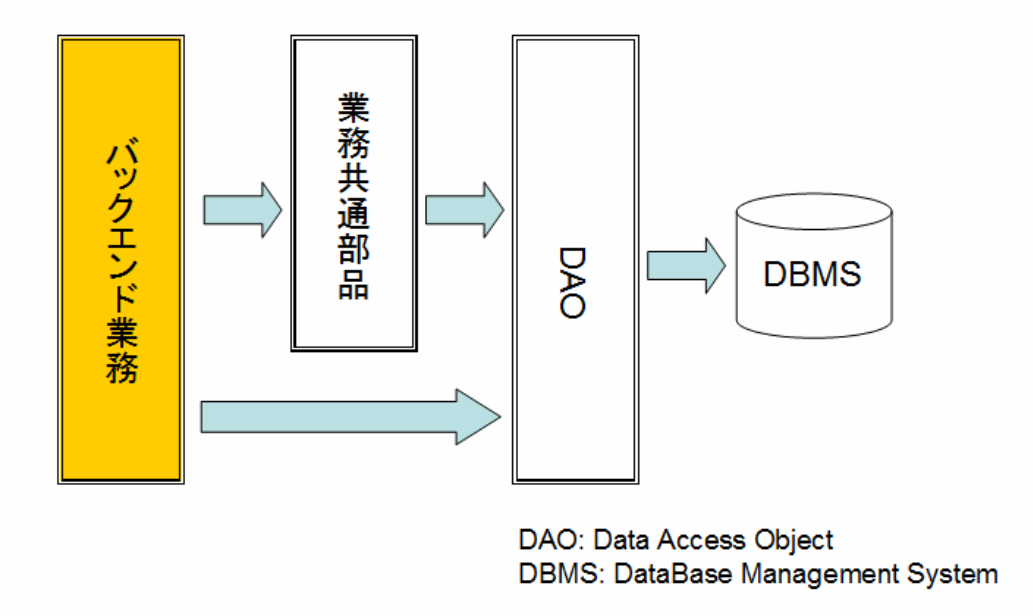

図 2. バックエンド業務からのデータベーストランザクション

2.4 目標とする処理性能

定量的に論ずることができるように、「性能問題がない」という状態は、情報システムが目標とする時間を超えずに処理を完了できる状態と定義する。当社の開発業務における対象領域では、1 回のバックエンドトランザクションで約 300 回の SQL を実行することがある。処理速度の要求がシビアなものでは、時間内にバッチを完了するために、1 トランザクションあたり 700 ミリ秒以内で処理を終える性能を求めていた事例がある。この数値を処理性能の目標とした。

2.5 取り組みの対象範囲

結合テスト局面であらゆる性能問題を分析し対処することは不可能である。取り組む対象範囲は次のように明確化した。

取り組む対象

- プログラム構造に起因する問題
- データベースのテーブル定義構造に起因する問題
- 結合テスト用の限られた量のデータで再現可能な問題
- バックエンド業務処理部分のみで再現可能な問題
- シングルスレッドで再現可能な問題
- 結合テストサーバー環境で再現可能な問題

上記の問題について、原因がバックエンド業務サブシステムでない場合は、原因サブシステムの切り分けまでは行いたい。

対象範囲外(システムテスト局面で実施するべきもの)

- 大量データを並列に処理する場合のみ顕著となる問題
- フロント業務処理を組み合わせたシステム全体ではじめて再現する問題
- 本番サーバー環境、システムテストサーバー環境のみで再現する問題
- OS やミドルウェア(アプリケーションサーバーやデータベースサーバーなど)のチューニングによって解決するべき問題

2.6 性能問題分析の要件

結合テスト局面の時点で性能問題を分析するにあたっては次のような要件があるものとする。

- 一連のトランザクション処理の中で、モジュールごとの所要時間を計測し、ボトルネックの有無と、ネックがあればその場所を把握できること
- 当社が参照・修正可能なソースコードはバックエンド業務処理サブシステムのみであり、その他のサブシステムのソースコードを当社は参照できない
- 詳細な性能問題分析ができるようにコールグラフ(Java メソッドごとの呼び出し順序)を作成し、参照できること
- 実行されている SQL と実行回数を把握できること
- 計測で用いるのは本番環境スペックのシステムではなく、結合テスト環境のシステムである
- 性能問題分析用途で専用のログ出力が必要な場合は、性能分析時のみログ出力コードを結合し、分析後は容易に分離できること

2.7 結合テスト局面の状況

性能問題分析を試行する対象として、ある実在するシステムを用いた。このシステムでは、バックエンド処理結合テスト起動ドライバーが存在し、結合テストを実施するために利用することができる。これを以降は IT ドライバーと称する。

この IT ドライバーを用いると、バックエンド業務以降の一連の業務トランザクションを、Web 画面やバッチ処理本体から切り離し、単体で直接起動することができる。最も重いトランザクションを IT ドライバーで起動した場合の最大所要時間は、当初 1,530 ミリ秒であり、本番サーバー環境と開発用 Windows 環境の違いはあるが、目標性能である 700 ミリ秒を大きく超えていた。性能問題が存在すると言える状況である。

3. 性能問題を早期分析するためのソリューション

本章では性能問題を早期分析するために、適切なソリューションの検討を行い、その結果選択したアスペクト指向プログラミングについて概要を述べ、今回の検証に対する適用イメージを示す。

3.1 Java SE が備えるプロファイラーの限界

Java SE v1.6 以降にはプロファイラーとして Java VisualVM(以降、VisualVM とする) が付属する。VisualVM には次のような機能がある。

- メソッドごとの CPU 利用時間とメソッド実行回数の計測
- クラスごとのメモリー利用量解析
- 各クラスのインスタンス数、生成された数

VisualVM 実行中の画面サンプルを図 3 に示す。

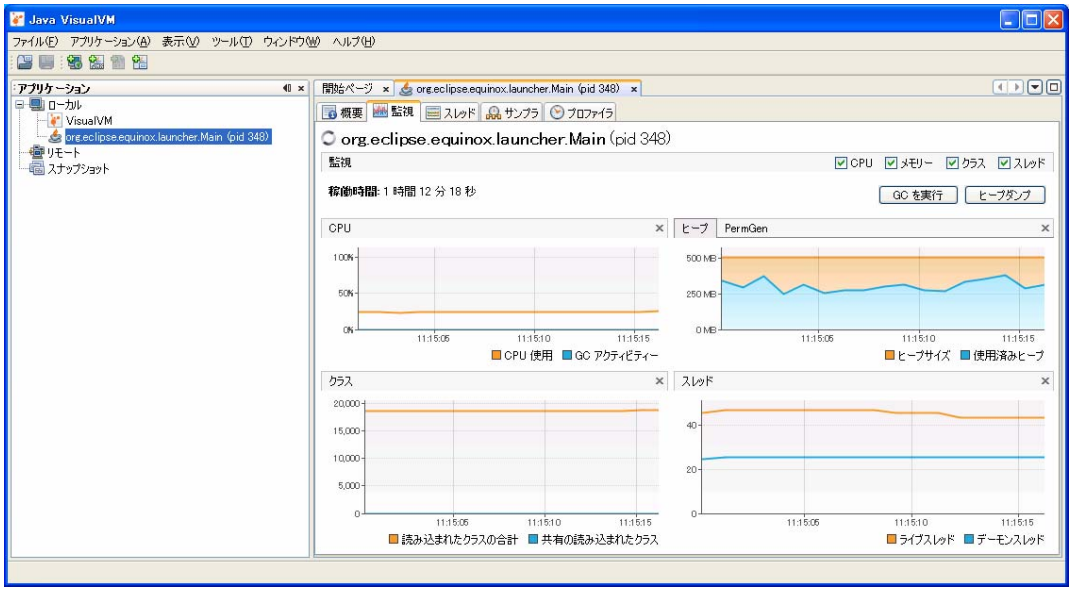

図 3. Java VisualVM の実行例

VisualVM ではある時点のコールグラフを作成することはできるが、実行されたすべてのメソッドに対するコールグラフは生成できない。また、どのメソッドの中でどのような SQL が実行されたかを確認することもできない。更に VisualVM では起動した Java アプリケーションにリンクする操作が必要となる。2～3 秒で完了する IT ドライバーに適切なタイミングでリンクさせ、IT ドライバー全体ではなく、業務ロジック実行部分のみを詳細にプロファイリングする用途には不適切である。

3.2 アスペクト指向プログラミングの適用

前節のように、VisualVM のような既にセットアップされたツールは、結合テスト局面における性能問題分析の要件を満たさないことがわかった。

したがって、アプリケーションのメソッドの開始と終了時点にトレースログ出力を追加し、このログをもとにトランザクション実行中の詳細な情報を分析する仕組みが必要となる。ここで分析対象の Java クラスは約 500 本あり、それらに平均で 15 個のメソッドがある。ログを埋め込む必要のある箇所は 500x15x2 で約 15,000 箇所にも及ぶ。メソッドの終了箇所は 1 箇所でない場合もあり、例外をスローする箇所も考えると実際はより多くの埋め込み箇所があるものと推測できる。このような多数のトレースログ出力の埋め込み作業で漏れや埋め込みミスがあってはならない。

また、最終的にプログラムのソースコードから不要なトレースログ出力のコードは削除したいが、その削除作業でも漏れやミスがあってはならない。以上を満足するための技術的アプローチとしてアスペクト指向プログラミング(以下、AOP とする)に注目した。

AOP では、プログラム中の幾つもの箇所にかかわる共通の処理(AOP の用語で「横断的関心事」と言う)を本体とは分けて準備しておき、条件に対応して横断的関心事を自動で適用させる制御が可能となる。AOP の一般的な適用例を示す。

- 業務ロジックが開始された時に自動でトランザクションを開始する
- 業務ロジックが正常に完了した場合に自動でトランザクションをコミットする
- 画面へ英語ラベルを出力する際に、変換テーブルを参照し自動で日本語のラベルへ置換する

上にあげた例を補足する。業務ロジックが正常に完了した場合に必ずトランザクションをコミットする要件について、それを開発者が手で実装する場合、該当する多数の箇所のどこかで実装漏れが発生するおそれがある。これは自動化を検討するべき課題であったが、構造化プログラミングやオブジェクト指向プログラミングではスマートに解決できなかった。AOP では業務ロジックの正常終了箇所を自動で判別し、コミット処理で行うべきことを 1 度だけ定義することで、すべての箇所に自動で実装することができる。該当箇所が多数存在しても、1 度だけの定義で漏れがないことを保障できる。

Spring Framework や Seasar2 といった著名な Java フレームワークの内部で宣言型トランザクション制御や例外処理の実現のために AOP が用いられている。つまり Java 言語によるアプリケーション開発において我々は既に AOP の恩恵を間接的に受けていると言える。しかし業務アプリケーション開発者が AOP を直接利用することは事例が少ないのではないだろうか。

AOP では、横断的関心事をプログラムで表現する「アドバイス」の記述、横断的関心事を実行する場所である「ポイントカット」の定義、アドバイスをポイントカットに基づいてプログラムに適用する「織り込み」の実行、以上 3 つの作業を行う。「アドバイス」「ポイントカット」「織り込み」は AOP で一般的に用いられている用語である。これらの用語の関連を図 4 に示す。本稿でもこれらの用語を利用する。

今回は、トランザクション開始後に実行される各メソッドの開始と終了の時点を「ポイントカット」として、ログファイルへ情報を出力する「アドバイス」をコンパイル時に「織り込む」形で AOP を利用することとした。

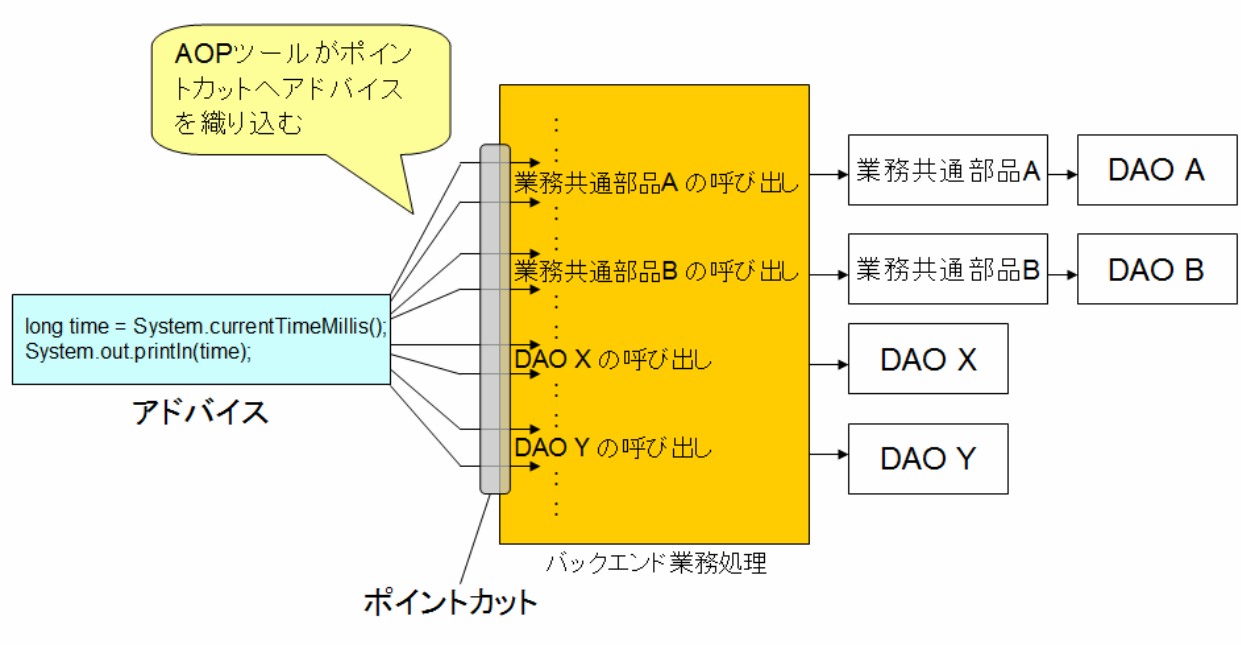

図 4. AOP の利用イメージ

AOP を利用するための具体的なプロダクトとしてはSeasar2 のようなDI コンテナベースのものと、AspectJ のような単体の AOP プロダクトに大別することができる。DI コンテナベースのプロダクトでは、コンテナが管理するオブジェクトに対してのみアドバイスを埋め込むことができる。最初からそのような DI コンテナを採用し開発している場合であれば、AOP が必要となった場合に簡単な設定追加で利用することができる。しかし一般的な開発案件では案件ごとの開発標準がさまざまなフレームワークの利用を指定しており、常に AOP 機能を備えたフレームワークを利用できるわけではない。参考文献(2)に示す論文では、AOP プロダクトとフレームワークとの相性が考察されており、フレームワークへの依存が少ないものとして、オープンソースのプロダクトである AspectJ が良好であることが示されている。任意のフレームワークをベースとして実装済みのコードベースに修正を加えず効果を得るため、単体のツールである AspectJ を外部に追加して性能分析を実施することにした。

3.3 AspectJとその適用イメージ

AspectJ は Java 言語で AOP を利用可能とするオープンソースのツールキットであり、現在 Eclipse プロジェクトが開発し Eclipse Lisence version 1.0 ライセンスでプロダクトを無償で公開している。

AspectJ を追加するということは、既存の業務アプリケーションのソースコードを修正することにはならない。性能分析用の使い捨ての Java クラスをアドバイスとして作成し、コンパイルのタイミングで既存の業務アプリケーションへ織り込むことが可能である。

性能分析が終了すれば AspectJ を用いずに再コンパイルすることで性能分析用のログ出力のない元のクラスモジュールへ戻すことが容易に可能である。手作業と比較し、ログ出力コードの組み込み漏れによる分析精度の低下や、取り外し漏れによる納品物の品質低下の心配がない。

4. AspectJによる性能問題分析

本章では実際に AspectJ を用いた性能問題分析の検証について述べ、それから得た知見を示す。

4.1 性能問題分析の仕様

AspectJ で織り込むアドバイスは次の詳細仕様とした

- 次の 3 種類のメソッドに対する呼び出し時点と復帰時点をポイントカットとする
  1) DAO のメソッド
  2) 業務共通部品のメソッド
  3) バックエンド業務処理内の子クラスのメソッド
- DAO や業務共通部品のソースコードをコンパイル時に参照できなくてもトレースできるように、execute ではなく call 形式でポイントカットを定義する（※1）
- ログには呼び出す対象のクラス名とメソッド名を出力する
- Java API で提供されている System クラスの currentTimeMillis()メソッドを用いて取得した時刻情報をログへ付与する
- メソッド呼び出し時点のログにはマーカーとして「calling」文字列を付与する
- メソッドからの復帰時点のログにはマーカーとして「finish」文字列を付与する
- 一般のログと区別するために、AspectJ で出力するログでは行頭にマーカーとして「__ASPECT__」文字列を出力する

（※1）execute と call について
execute 形式は、呼び出される側のメソッドの先頭と末尾にアドバイスが織り込まれる。そのため、呼び出される側のソースコードが必要となり、それがない場合 execute 形式は利用できない。一方、call 形式は呼び出す側のソースコードを用いて、対象メソッドを呼ぶ直前と直後にアドバイスを織り込む。call 形式では呼び出される側のソースコードは不要となる。

また、AspectJ を用いて出力したログと、もともと出すようにしていたアプリケーションのログをつきあわせて分析し、レポートを集計出力するログアナライザーを Java アプリケーションとして新規に作成した。このログアナライザーは次のような項目をレポートする

- トランザクション全体の処理時間
- メソッド 1 回ごとの処理時間とコールグラフ
- 各メソッドの呼び出し回数と、処理時間の合計
- 実行された SQL
- SQL の種類ごとの統計情報

- ガベージコレクションの発生回数と、所要時間
- モジュール(業務ロジック、DAO、共通部品)それぞれの処理時間合計

ここで SQL のログは対象アプリケーションが出力機能を保有していたため、その機能を利用する。ガベージコレクションのログは Java VM のオプション引数「-verbose:gc」「-XX:+TraceGen0Time」「-XX:+TraceGen1Time」「-XX:+PrintGCDetails」「-XX:+PrintGCTimeStamps」を指定すると得ることができる。

ログアナライザーは CSV 形式のプレインテキストファイルへ結果を出力し、マイクロソフト EXCEL を用いてこれを閲覧するようにした。また、ログ分析の負荷がオーバーヘッドとならないように、ログアナライザーはトランザクションの実行が完全に終了してから実行する。

ログアナライザーは 500 行近い量のレポートを出力するが、その中から統計出力部分の箇所を抜粋して図 5 に例示する。

report_1.xls

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
423															
424		【合計時間の集計】(msec)													
425		合計時間	Main total:	1531											
426		合計時間	DAO	994											
427		合計時間	PARTS	279											
428		合計時間	SORT	1											
429															
430		【総呼び出し回数の集計】													
431		総呼び出し回数	EXA	78	回										
432		総呼び出し回数	DAO	29	回										
433		総呼び出し回数	PARTS	75	回										
434		総呼び出し回数	SORT	39	回										
435															
436		【SQL実行回数の集計】													
437		SQL回数	SELECT	94	回										
438		SQL回数	INSERT	6	回										
439		SQL回数	UPDATE	1	回										
440		SQL回数	DELETE	0	回										
441															
442		【Main 実行期間中のガベージコレクションの集計】(ITドライバ処理中は含みません)													
443		通常GC 合計	0	回	0	μ秒									
444		Full GC 合計	0	回	0	μ秒									
445															
446		【主要モジュール毎の集計】	総処理時間	自己時間	DAO時間	部品時間	sort時間	EXA子回数	DAO回数	部品回数	sort回数	SQL回数	GC回数	FullGC回数	
447		入力チェック	99	59	12	28	0	9	3	15	0	29	0	0	
448		モデル構築	851	41	722	88	0	6	19	15	0	35	0	0	
449		データ解析	46	27	0	19	0	6	0	3	4	1	0	0	
450		データ制御	127	61	0	66	0	39	0	37	2	25	0	0	
451		データ集計	24	6	0	18	0	4	0	5	0	4	0	0	
452		DB更新	337	22	260	54	1	4	7	0	33	7	0	0	
453		出力処理	47	41	0	6	0	3	0	0	0	0	0	0	
454		合計	1531	257	994	279	1	71	29	75	39	101	0	0	
455															

result

図 5. ログアナライザーで出力したレポート

4.2 性能問題分析のトライアル実施

性能問題分析を行う前に、作成した仕組みの動作を確認するためトライアルを行った。本節ではトライアルで経験した 3 つの問題とその対処方法を示す。

4.2.1 AspectJ が自動追加する「ajc$」フィールドによる実行エラーの発生

AspectJ を用いてアドバイスを織り込む前は正常に動作していたが、織り込み後は業務ロジックの

実行で Java の例外が発生するようになった。Java 例外のスタックトレースによると例外の発生箇所はフレームワーク内部の DAO モジュールの中であった。

アプリケーションで利用しているフレームワークでは、DAO が Entity オブジェクト内のフィールドをリフレクション(※2)でスキャンし、それに対応するゲッターやセッターのメソッド名を組み立てて呼び出す処理がある。この時に AspectJ がコンパイル時に各クラスへ内部制御用に自動付加する「ajc$」ではじまる名称のフィールドも検出してしまい、ゲッターやセッターメソッドの名称を組み立てて呼び出そうとする。しかしこの内部制御用フィールドには対応するゲッターやセッターは存在しないため、リフレクションを用いたメソッドの起動時点で NoSuchMethodException 例外が発生していることが判明した。この様子を図 6 に示す。

(※2) リフレクションについて
リフレクションとはプログラムの実行過程でプログラム自身の構造を読み取ったり、メソッドを文字列の名前で呼び出したりする技術のことである。Java もリフレクション機能を備えている。

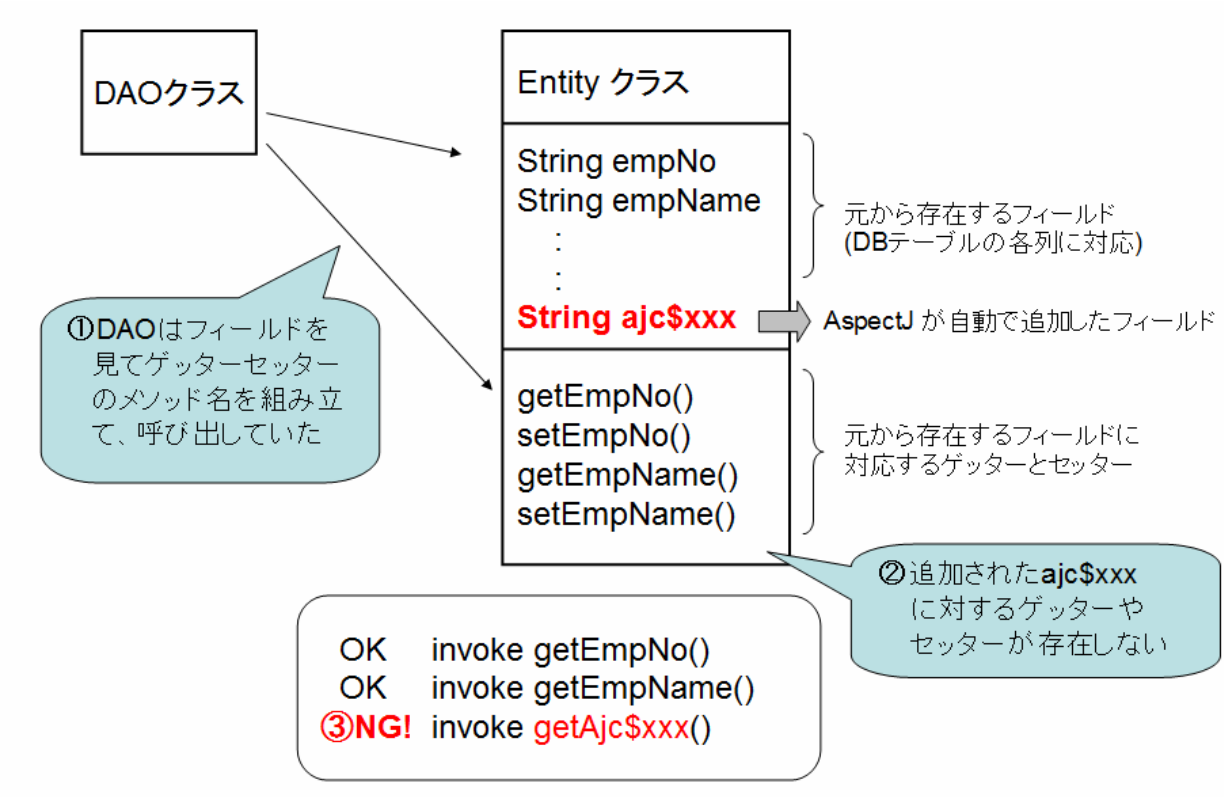

図 6. AspectJ が追加するフィールドによるエラー

フレームワークも含めて当社で開発する場合はフレームワーク自体を改善することも可能であるが、業務アプリケーション部分のみの開発を請け負うケースも多くある。Entity オブジェクトに対するゲッターやセッターは単に変数の値にアクセスするためだけのメソッドであり、これは意味のあ

る業務ロジックではなく、分析の粒度を考えるとこれらの詳細な所要時間を知る必要はない。AspectJ を用いて AOP を実施する場合は対象とする Java ソースコードを AspectJ が提供する専用のラッパーコンパイラでコンパイルするが、Entity クラスを格納するパッケージはその対象外とした。すなわちフレームワークは修正せず、測定対象の調整で実行エラーを回避することとした。この対策により、AspectJ でアドバイスを織り込んだ後でも業務ロジックが正常に動作するようになった。

4.2.2 所要時間算出の精度の問題

処理終了時刻から処理開始時刻を減算することでメソッドごとに処理所要時間を出力したところ、0 ミリ秒 のメソッドが多くみられた。0 ミリ秒のメソッドは 300 回実行した場合でも積み上げた総所要時間の集計は 0 ミリ秒となってしまい他のメソッドのそれと正確に比較することができない。

java.lang.System クラスの currentTimeMillis() メソッドではミリ秒単位の時刻情報を取得できるが、実際に確認してみるとその分解能は 1 ミリ秒ではなく、5 ミリ秒から 10 ミリ秒程度のようである。したがって、1 ミリ秒や 2 ミリ秒を要する処理では 0 ミリ秒と計測されてしまうことがある。時刻情報の取得に currentTimeMillis() メソッドを用いたことが問題の原因である。

JavaSE 1.5 以降では java.lang.System クラスに追加された nanoTime() メソッドを利用することができる。このメソッドでは CPU のクロック等を活用して精密な時刻を取得しており、実際に試してみるとナノ秒とまではいかないが、マイクロ秒程度の分解能を得ることができるようである。currentTimeMillis()より nanoTime() の方が所要時間算出の精度が良好となる。

処理開始前後の時刻を nanoTime() を用いてログ出力するようにアドバイスを修正することで、メソッドの所要時間が 0 ミリ秒となることはなくなった。

4.2.3 アドバイスのオーバーヘッドの問題

当初 1,530 ミリ秒かかっていたトランザクションは AspectJ を適用するとそのオーバーヘッドのために 2,410 ミリ秒かかるようになった。AspectJ によるオーバーヘッドが 58%あることになる。

このオーバーヘッドの手掛かりを得るために、出力したログを確認した。トライアルでは当初トランザクション 1 回あたり、2.4 秒の実行期間に 24,500 行のログを出力していた。この中には、バックエンド業務処理がもともと出力していたログや、Java VM のガベージコレクション情報のログも含んでおり、AspectJ がすべてを出力したのではない。

しかし、出力したログを精査していると、モジュール間でデータを引き渡す JavaBeans のセッターやゲッターの開始と終了のログを AspectJ から大量に出力していることがわかった。

上記 4.2.1 項の Entity と同様にこれらのメソッドも業務ロジックではない。すなわち分析の対象から除外しても差し支えない。ところが該当する JavaBeans は多くの Java パッケージ中に散在し、Entity のように AspectJ のコンパイル対象外に設定することは困難であった。幸いこのような

JavaBeans は命名規則としてクラス名の末尾を必ず DTO としていたため、DTO で終端するクラス名の場合はログを出力しないようにアドバイスの実装を修正した。更に定数を保持するクラスの enum 値周辺に対するアクセスに起因した大量のログもアドバイス織り込み対象外とした。

これらの改善によりログの出力量が 30% 減少し 16,800 行となり、所要時間は 1,750 ミリ秒となった。必要とする業務処理メソッドの処理時間情報は残しつつ、トレースログ出力のオーバーヘッドを上に述べた 58%から 14%へ軽減することができた。

4.3 性能問題分析の実施

前節の改善を適用し、改めてトランザクション性能を分析した。前節の通り、初回の計測ではトランザクション 1 回あたり、平均 1,750 ミリ秒を要した。目標としている 700 ミリ秒を満足しない結果である。本節では、ログアナライザーのレポートを分析して抽出した問題点と、それに対して実施した対策を示す。

4.3.1 データベーステーブル構造に起因する性能劣化

業務処理の前半でデータベースから必要なデータ群を取得するが、この部分で 520 ミリ秒の時間を要していた。コールグラフをたどって処理内容を確認すると、正規化をあえてくずして横に 800 列の構造となったマスター系テーブルを参照しており、その部分が遅いことが判明した。このテーブルは 100 行程度のデータを保有しているだけであり、SQL の実行計画をみても不審な点はなかった。このテーブル設計の背景を知る者に尋ねたところ、高速化のために COBOL の経験をいかしてテーブル構造をこのように工夫しているということであった。

コンパイルの時点で事前に SQL をパースし、静的にデータベースへアクセスする COBOL では、SQL で抽出する列のリストがどんなに長くても、実行時に SQL をパースする時間はゼロとなる。したがって、検索結果の行数を少なくする工夫が高速化に直結する。しかし、図 7 のように、プログラム実行時に毎回 SQL を組み立てパースして実行する Java では SQL が長くなるとデータベースが SQL を解釈するために時間を要する可能性がある。（ただし、PreparedStatement クラスなどを適切に用いることで、SQL パース結果のキャッシュ効果により、2 回目以降のスループットを改善することは可能である。）

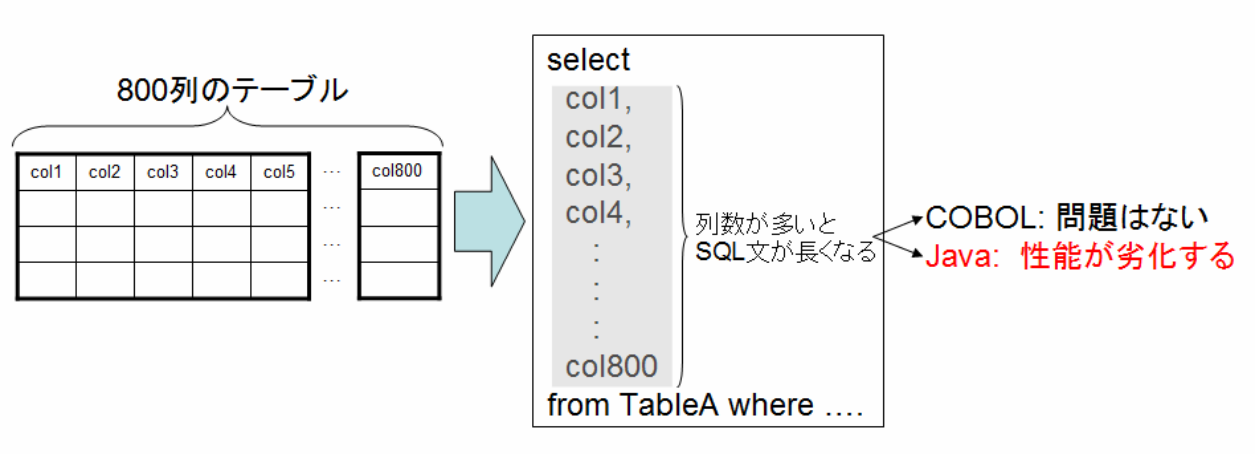

図 7. 長い SQL による Java の性能劣化

また、4.2.1 でも述べたように今回利用している DAO は検索結果を Entity オブジェクトへ格納するが、その仕組みはリフレクションでセッターを動的に実行する汎用的な形式としている。図 8 のように、テーブルの列数が多い場合は対応する Entity オブジェクトのフィールドが多くなる。プログラム実行時にプログラム自身の構造を分析するリフレクションは重い処理であるから多く実行すると処理性能が劣化する。

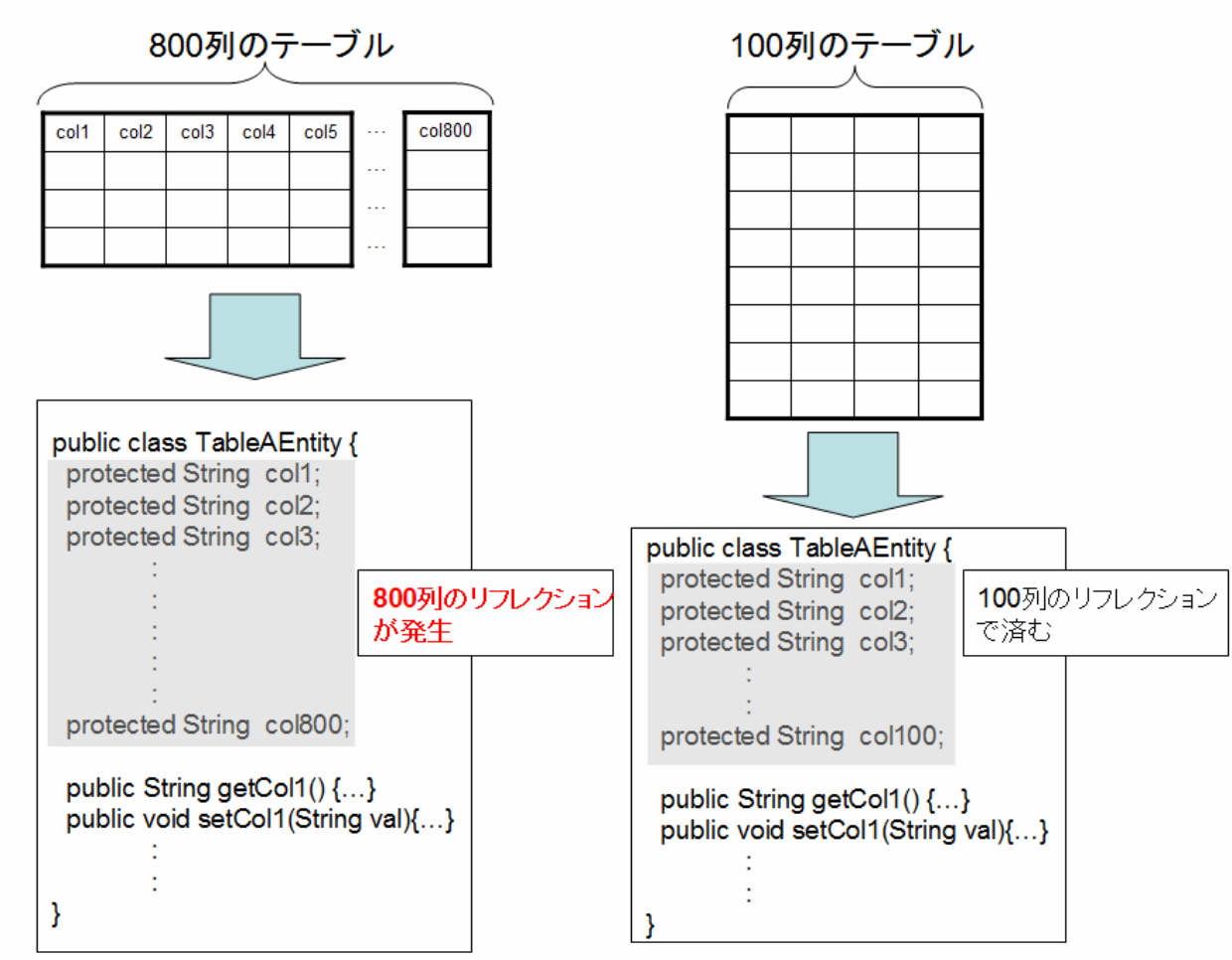

図 8. テーブル構造によるリフレクション対象数の変化

すなわち、テーブルの正規化により検索結果の件数が多くなっても、列の数が少なく短い SQL の方が Java では高速となる可能性がある。この仮説を検証するため、縦の行数は多くなるが、横 100 列の構造に正規化して実際に測定したところ、元の 520 ミリ秒から 230 ミリ秒へ改善することができた。この部分で約 2.3 倍の高速化である。トランザクション全体でみると、1,750 ミリ秒が 1,460 ミリ秒となった。以上のように、COBOL において有効であったデータベース構造の最適化が Java では必ずしも有効でないことを認識した。

このようなデータベース構造の変更による性能改善をシステムテスト局面で実施するとさまざまな影響を伴い大きな手戻りとなる。結合テスト局面で問題を検出し、修正できれば少ないコストで済む。

4.3.2 ガベージコレクションの影響を回避

トランザクション実行部分の測定中に、不要となったメモリーを自動回収する機能である、Java のガベージコレクション(以下、GC とする)が発生していた。1 時間といった、ある時間範囲の中でシステムのスループットを測定する場合は GC の所要時間も織り込んで測定する必要があるが、今回のよ

うに業務ロジック中のボトルネックをトランザクション１回の測定で分析したいシーンではGC の影響は除外したい。そこで IT ドライバーから業務トランザクションを開始する直前で Java API で提供されている System クラスの gc() メソッドを実行し、GC を事前に発生させるようにした。この様子を図 9 に示す。

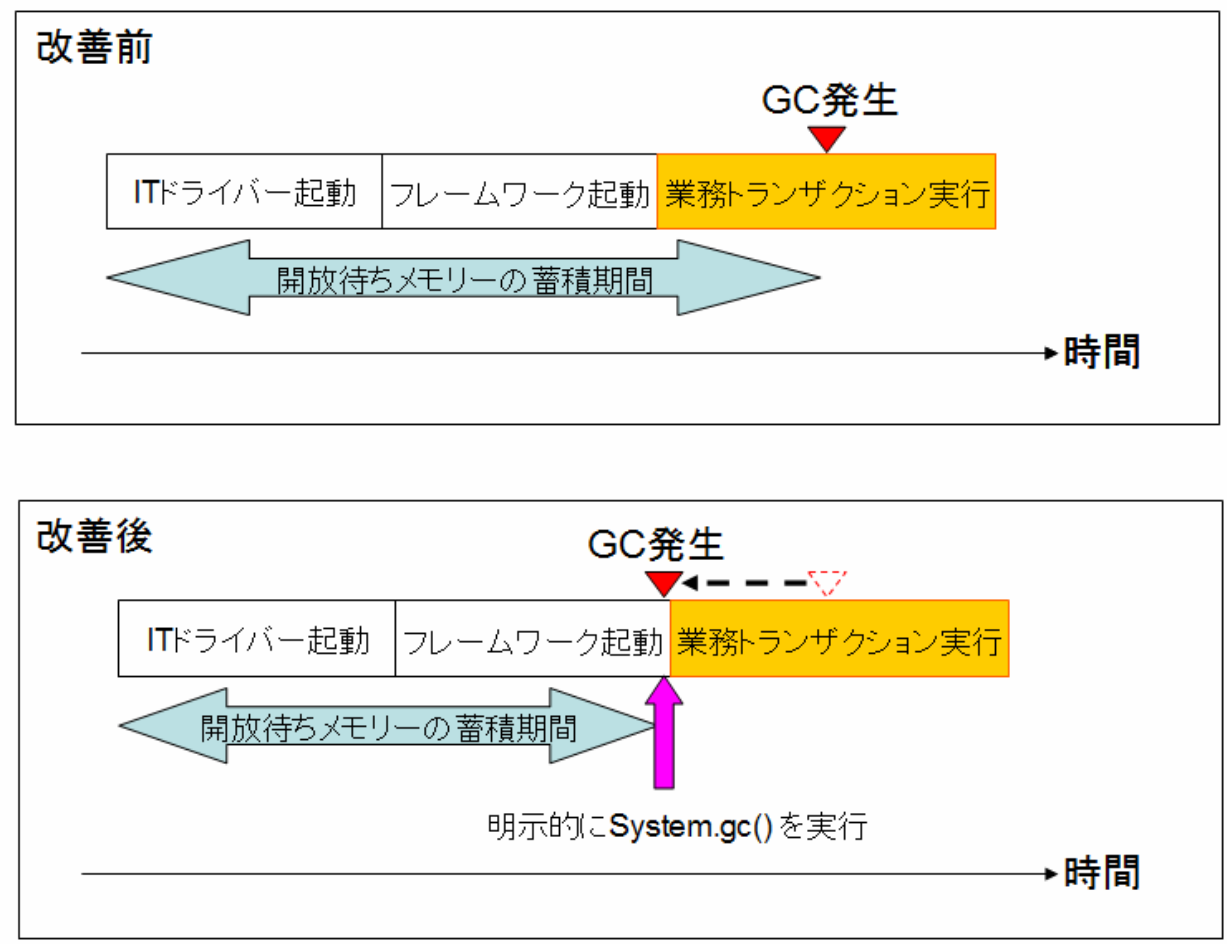

図 9. ガベージコレクションの影響を回避

ヒープメモリーがクリーンな状況で業務トランザクションを開始し､所要時間計測の期間中は GC の発生を回避することができた。所要時間は 1,320 ミリ秒に改善した。

4.3.3 トランザクション起動方法の改善

前項までの改善を行っても、所要時間を目標の 700 ミリ秒以下とすることはできなかった。ログアナライザーによるレポートを精査し、業務アプリケーション側に新たな改善箇所を求めたが、問題となりそうな特定の箇所はなく、よいアイデアは浮かばなかった。そこで視点を変え、IT ドライバーによる起動、それに起因する制約について踏み込んで考察した。

IT ドライバーは Java アプリケーションとして実装している。すなわちアプリケーションサーバー

を用いて起動済みのプロセスがメモリーに常駐する形式ではない。

業務トランザクションを IT ドライバーで実行すると Java Virtual Machine が起動し、IT ドライバーとフレームワークをロードして起動し、更にその上で業務アプリケーションのクラスをロードして実行している。Java Virtual Machine のクラスローダーはロード済みのクラスをキャッシュする機能があるが、IT ドライバーによるトランザクションの実行ではクラスローダーキャッシュが空の状態から実行することとなる。

また、フレームワークは汎用的な処理を実現するために Java の機能であるリフレクションを多用しているが、これは負荷の重い処理であり、リフレクション結果を独自にキャッシュすることで高速化を図っている。システム本来の稼働状況ではアプリケーションサーバーによりプロセスが常駐し、クラスローダーキャッシュが利用され、リフレクションキャッシュも活用される状況であるから、IT ドライバーによる実行とは状況が異なると考えた。

アプリケーションサーバーによる実行に近い状況で本来の処理速度を計測できるように、2 種類のダミートランザクションを流して最後に計測対象のトランザクションを実行するよう、IT ドライバーを工夫した。修正した IT ドライバーで実行する処理の順序を図 10 に示す。この修正の効果は大きかった。業務トランザクション全体の所要時間は 410 ミリ秒となった。

システムテスト局面で利用できるシステムに近い実行速度で、業務プログラムを結合テスト局面で実行するために、この工夫は有効であると感じた。

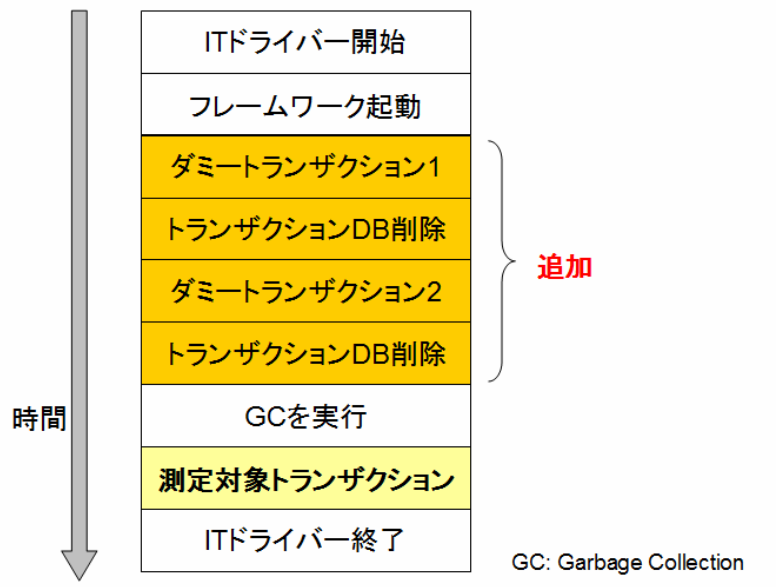

図 10. 改善した IT ドライバーの起動順序

4.4 改善後の測定と結果の評価

前節の改善を行い、業務トランザクション実行の総所要時間を測定すると、前項で述べたように 410 ミリ秒という結果を得た。すなわち、700 ミリ秒以内という目標を 41%オーバー達成することができ

た。

改善にしたがって向上した所要時間の推移を図 11 に示す。

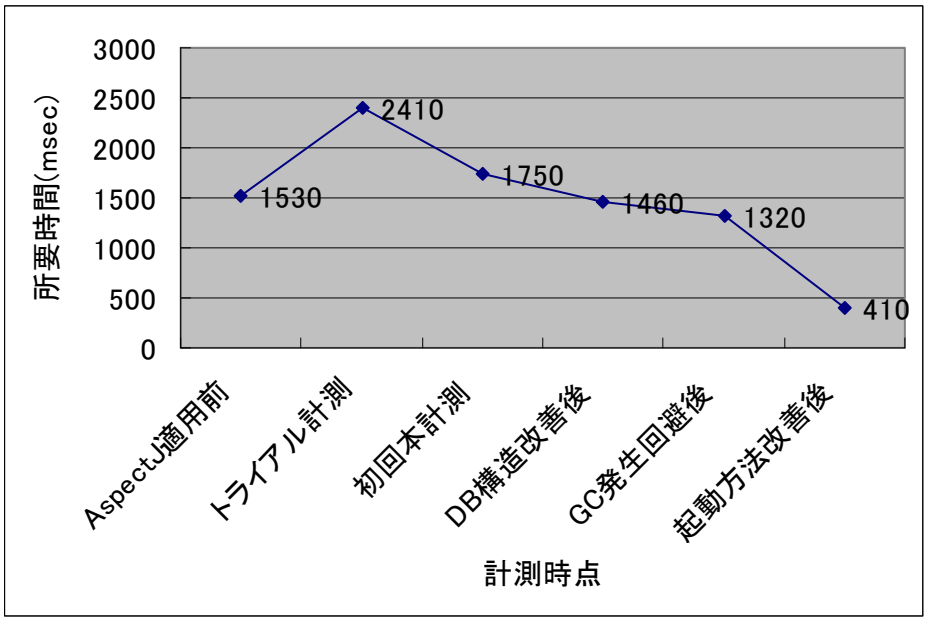

図 11. 改善による所要時間の推移

架空プロジェクトにおける試行でプログラムやデータベースの構造が目標性能を満たすように改善でき、性能問題を解決できることを確認できた。また、プロジェクト管理の視点では後続のシステムテスト局面において担当のサブシステム単体に対し性能問題に関する手戻りは発生しないことを確信できる効果がある。同じ手順を実プロジェクトでも適用することは可能である。

なお、今回の実作業のために要した時間は AspectJ の学習のために約 9 時間、AspectJ の適用のために約 2 時間、トレースログの書式の最適化とログアナライザーの作成に約 4 時間、トライアルと本測定及び改善の試行錯誤のためには約 12 時間を要した。ただし、全体方針の検討時間や調査の時間、実システム周辺の準備時間は含まない。

5. おわりに

システムテスト局面では、本番環境に近いシステム構成で、システムテスト用の大量のデータを用いてスループットを計測することができる。しかし結合テスト局面では業務トランザクションを 1 回起動する程度のことしか行えない。0.5 秒や 1.0 秒といった短期間のライフタイムの中でどの箇所に問題があるかを知り、改善してゆくことは容易ではない。

本稿では AOP を活用し、1 トランザクションの中で発生する詳細な処理状況の分析を実現し、まだインパクトの小さい結合テスト局面で性能問題を把握し、早期に解決できることを明らかにした。ま

た、AOP プロダクトとして AspectJ を実際に利用し経験した問題点と解決方法をあわせて示した。これらの知見が情報システム開発にたずさわる諸氏の案件でも活用され、プロジェクトが予定の期間とコストで進捗するよう寄与できれば幸いである。

参考文献

1. 株式会社エクサ 技術部 友野晶夫、アスペクト指向開発適用記、exa review No.2 2003、
http://www.exa-corp.co.jp/techinfo/review02/exareviewvol2.pdf

2. 株式会社エクサ 技術部 広戸裕介、 アスペクト指向プログラミング適用による Web アプリケーション開発の簡易化、平成 17 年度 JGS 委員会 研究プロジェクト チーム論文 2006 WN-04

3. The AspectJ Project、http://eclipse.org/aspectj/

4. Spring Framework、http://projects.spring.io/spring-framework/

5. The Seasar Project、http://www.seasar.org/
