オブジェクト指向プログラミング言語 molecule

OBJECT ORIENTED PROGRAMMING LANGUAGE molecule

銭谷　謙吾
Kengo ZENITANI

〒590-0105 大阪府堺市竹城台２丁１番１４－２０２　E-mail: zenitani@mtb.biglobe.ne.jp

ABSTRACT. Services implementations over the Internet become more and more important today. But the methods for those developments are generally limited in client-server style or peer-to-peer style. For this development, I designed an object oriented programming language `molecule' for purposes of network services development in a sense of distributed object oriented method. `molecule' supports serialization of any object with its running status, and any software written in `molecule' can be automatically treated as an Agent without any particular implementation effort. In this context, Agent means software can move across the network boundary with its running status.

１．背景

近年に見るインターネットの発展はソフトウェア製作技術にも多大な影響を及ぼしており、特に Web サービスを中心とした分散オブジェクト指向技術の発達により、ソフトウェアはなんらかの個物としてではなく、様々なサービスの表現手段へと変化してゆくことが見込まれる。ただし、現状の Web サービスで主として想定されているのはあくまでもオブジェクト間通信をリモートにまで拡張することであり、クライアントサーバ型システムのオブジェクト指向による拡張としての性格が強い。

一方、これとは別にここ最近の大きな潮流として見られるのが、P2P(Peer to Peer)型ネットワークによるシステムである。P2P 型のネットワークではシステムの中心となる特定のサーバーを持たず、求められる機能を分散協調により実現する。これはクライアントサーバ型ネットワークの中央集権的なアプローチとは異なる、あるいはこれを補完する方向性であり、ファイル共有アプリケーションを中心にして急速な発展を続けている。

いずれの二者にも共通するのは、一般ユーザーに対しインターネットを通じていかにしてサービスを供給するかという問題意識であり、一面を捉えれば、それを解決する際の比重をサーバー側に置くかクライアント側に置くかの違いが性格を分けていると言える。簡単には、クライアントサーバ型 Web サービスにおいてはサーバーを設置することによるサービス供給の安定性や信頼性の向上が図れる一方、サーバーの維持・運用にまつわるコストやシステム改変の際の労力などが柔軟性を削ぐ傾向にある。これに対し、P2P 型サービスにおいては固定されたシステムへの依存性が低いために柔軟性が高く、サーバーサイド処理の単純化に伴う維持・運用コストの低減が図れる一方、バケツリレー方式の通信に伴う遅延・帯域の無駄遣いやピアの信頼性の低さに伴うサービスの品質維持の困難などがある。サービス開発時のコストを考慮すれば P2P 型システムに優位性があるが、長期的な運用品質を考慮すればクライアントサーバ型に優位性があり、一般ユーザーに向けたサービス供給に際しては、そのサービスの特性を考慮して、これらの技術を取捨選択しなければならないのが現状である。インターネットを経由してサービスを供給するシステムの開発技術としては両者とも改善の余地がある。

これからのサービス開発技術としてのソフトウェア製作技術には、P2P 型と同様の開発の容易さ・柔軟性と、クライアントサーバ型と同様の安定性・信頼性の両方を兼ね備えることが望まれる。本案件はこのような背景に基づき、ネットワークサービスを開発する手段としてのエージェントシステムを開発するものである。

２．目的

前掲のような背景に基づき、本案件ではネットワークサービスを開発する手段としてのエージェントシステムの開発を行う。ここでいうネットワークサービスとは、主として一般ユーザーを対象に供給される、インターネットを通じて利用可能なサービスを指す。既存の具体例としては、ブラウザを経由して利用できるスケジューラやストリーミング放送視聴サービス、掲示板・チャットなどのコミュニケーションサービス、広義にはディスクスペースやメールアカウントの供給などを含む。今後考えられる例としては、オンラインデータベース、大規模放送システム、オンデマンド配信型アプリケーションなどが挙げられる。このようなネットワークサービスの開発を対象として想定するにあたり、目標とするのは次の 4 点である。

（1） サーバーの柔軟な活用

ネットワークサービスの供給品質を安定させるにはなんらかのサーバーを確保する必要がある。しかしサーバーの確保はそれ自体が費用を必要とするものであるのに加え、サーバーサイドにおける現在の開発手段は Web に関連するものに集中している。サーバーサイドにおいては、クライアントサイドアプリケーション開発において RAD(Rapid Application Development)環境の普及がもたら

したほどの開発の効率化は実現されていない。これは同時に、一旦開発したシステムを変更することの困難にも繋がっている。これを解決し、低コストにサーバーを確保し、かつ、それらの上で動作するシステムの開発を易化する。

（２） 高い拡張性の確保

ソフトウェア市場の変化は急速なものであり、現段階で開発した技術がそのままの仕様で長期的に実用性を維持できることは期待できない。これに対処するべく、高い拡張性を確保することにより、今後の状況の変化に追随できるよう配慮する。

（３） ネットワークに係る処理の記述の簡単化

ネットワークサービスを記述するにあたってはネットワークを経由した通信処理の記述が不可欠となる。端的には、種々のプロトコル処理をいかに簡便に記述できるかによって開発の効率は大幅に変化する。HTTP、SMTP、POP3、FTP、SOAP などの主要なプロトコルへの対応を行う。

（４） 文字処理の安全性の保証

インターネットには多種多様な言語圏に属するデータが流通しており、これらの文字データを損失なく効率的に扱うことは重要な課題である。特に、多国間にまたがるコミュニケーションシステムを構築する際には多言語処理が必須となる。もっとも重要なのは文字情報の損失を抑止することであり､UNICODE や JIS に依存しない文字処理への対応を行う。

なお、本案件の開発にあたり、次のことがらは目標としない。

（５） 実行パフォーマンスの追求

ネットワークサービスの開発手段としての生産性を最重視し、実行パフォーマンスの追求は二次的な要求とした。このことは、開発が容易になるような特徴を常に実行パフォーマンスよりも優先させることを意味しないが、今現在標準的なスクリプト言語(Perl,Python,Ruby)の実行パフォーマンスを基本とし、ネイティブアプリケーション水準のパフォーマンスは今後の課題とした。

（６） 過度のプラットフォーム非依存性の追求

本開発にあたっては 8 割の利用者の 8 割の要求に応えることに主眼を置き、ありとあらゆる環境に対応するような幅広いプラットフォーム非依存性を重視しない。これに合わせ、もっぱら PC 環境を利用環境として想定し、組み込みプロセッサなどへの特別な配慮は行わない。

これらの前提の下に、本開発では、ネットワークサービスをエージェントの分散協調によって実装することを狙い、これを実現するために必要なエージェントシステムの開発を行うことを目的とする。

３．オブジェクト指向プログラミング言語 molecule

（１） 概要

端的に述べれば、サーバーの利用を容易にするべく、汎用性を持ったサーバーに対する動的なサービスソフトウェアインストールを可能にするための基盤としてエージェントシステムを用いることが、本開発の骨子である。この実現には、エージェントの実行環境、開発環境、の双方を開発しなければならない。これに先立ち、まずエージェントの必須条件を次の 3 点とする。

1. エージェントは特定環境に依存しない。
2. エージェントは他のエージェントと通信できる。
3. エージェントは実行状態を伴いつつ他の環境に移動できる。

これらを比較的よく満たすソフトウェア開発技術としては Sun 社の Java や Microsoft 社の MSIL などがあるが、いずれも VM 水準での基本仕様の拡張という点では柔軟性を欠いているため本開発では採用を見送った。代替として、本案件では独自のオブジェクト指向プログラミング言語 molecule の処理系を実装する。

molecule は柔軟な文法を持ち、型付けのないダイナミックバインディングを前提として設計されている。molecule で記述されたプログラムは任意の実行時点でその全状態をシリアライズ可能であり、適切な通信ライブラリを組み合わせることで実行状態を伴いつつ他環境へ移動できる。また、molecule ではシリアライズアーキテクチャを中心に仕様を定めることにより、VM などを定めることなく環境非依存を実現している。

（２） 基本モデル

molecule における任意のプログラムは、シリアライズ可能なオブジェクトのみからなる集合体と見なされる。このシリアライズ可能性はプログラムの実行時にも維持されることが要求される。逆に、こうしたシリアライズ可能性と、個々のオブジェクトの動作仕様が正確に守られるならば、その実際の実装がどのようなものであるかは問わない。molecule においてはソースコードもオブジェクトのシリアライズ形態の一種に過ぎず、ソースコードのコンパイルはオブジェクトのシリアライズフォーマットを変換することに他ならない。

現実の molecule 処理系の実装では、オブジェクトはモレキュールと呼ばれる単位で取り扱われる。モレキュールは複数のパラメータ入力に対して自身の参照可能な環境に対し変化を与えるオブジェクトであり、一般的なプログラミング言語における関数に対応する。異なるのは、モレキュールは内部状態を参照可能な環境の一部として含み、そこに他のモレキュールを保持してもよく、またその実体がシリアライズ可能な点である。このようなモレキュールを基本単位としてソフトウェアを構築することができれば、そのソフトウェアは明らかにシリアライズ可能である。

（３） 基本オブジェクトセット

molecule 処理系のリファレンス実装では、プリミティブ(Boolean, Integer, Decimal, Letter, String)とアーキテクチャ関連(Agent, Contexts, Function, Method)のモレキュールを基本オブジェクトセットとして実装している。この中で重要なのはアーキテクチャ関連に分類されるモレキュールである。

アーキテクチャ関連のモレキュールは、実行コード水準のオブジェクトを表している。一般的なプログラミング言語では、プリミティブやコレクションの水準でのシリアライズ可能性は備わっているが、実行コード水準のオブジェクトのシリアライズ可能性は備わっていないのが通例である。molecule では実行コード水準の実体をモレキュールとして構成することにより、プログラム全体

のシリアライズを可能にしている。

（4） 実行フローとアーキテクチャモレキュール

一般的なプログラミング言語を例にとって考えてみると、プリミティブからなる状態変数の値を随時書き換えていく処理の表現がプログラムであると捉えることができる。これを molecule の場合に当てはめれば、書き換える側に位置するのがアーキテクチャモレキュールであり、書き換えられる側にあるのがプリミティブやコレクションのモレキュールに対応する。

既に述べたとおり、モレキュールには内部状態がある。状態変数の値を書き換えるとは、まさにその内部状態を変更することに他ならない。モレキュールの内部状態遷移は、そのモレキュールに対するパラメータ入力と参照可能な環境を基礎として決定される。したがって、molecule におけるプログラムの内容は、状態変数に対応するモレキュールに対して次々に環境とパラメータ入力を与える処理に等しい。アーキテクチャモレキュールは状態変数に相当するモレキュールのテーブルを管理し、また、それらに対するパラメータ入力を次々に与える機能を備えたモレキュールである。

リファレンス実装では Contexts モレキュールが実行フローの中心に置かれる。Contexts は一つのエージェントに含まれる全てのモレキュールへの参照と、実行コンテキストを管理する。Contexts はモレキュールへの参照を辞書の形で管理しているため、適当なキー値を用いれば、キー値に対応するモレキュールを Contexts から取得することができる。これにより、どのモレキュールに対してどのモレキュールを入力パラメータとして与えるか、を一組のキー値の列で表現することができる。Function はこのようなキー値の列を管理するオブジェクトであり、Contexts は実行すべき Function を内部のコンテキストスタック上で管理する。Function に含まれるキー値の列に基づき、Contexts の内部状態は随時遷移を繰り返し、辞書の内容や、辞書から参照されているモレキュールの内部状態も遷移する。

（5） シリアライズと状態保存

molecule 処理系によって実装されたソフトウェアは常にシリアライズ可能である。アーキテクチャモレキュールの一つである Agent モレキュールは一つのエージェントに対応するオブジェクトであり、エージェントを構成する全てのモレキュールを内包している。Contexts にはそれを包括する親モレキュールである Agent モレキュールへの参照が含まれており、ユーザープログラム上から Agent モレキュールを利用することができる。これにより、Agent モレキュールに対してエージェント全体のシリアライズ要求を発行することで、エージェントの実行状態をシリアライズできる。シリアライズは実行状態にあるモレキュールのコピーをある特定のフォーマットに沿って生成するものであり、生成されたシリアライズ形態のモレキュールはその状態ではデータとして外部から加工することができる。これにより、シリアライズ結果を更に加工して、自己複製、自己改変を行いながら移動できるエージェントを実装することも可能である。

（6） 拡張オブジェクトセット

molecule 処理系のリファレンス実装では既述の基本オブジェクトの他、コレクション(Vector,Stack,Queue,Map)、通信(HTTP,SMTP,POP3,SOAP)、正規表現などのモジュールの実装を行う予定である。

リファレンス実装は C++によって記述されており、容易に独自のモレキュールを追加することができる。

（7） エージェント間通信とプロトコル

リファレンス実装では Agent は Contexts 上にメッセージキューにあたるモレキュールを持ち、ユーザープログラムは Contexts 経由でこのモレキュールを参照することができる。エージェント間のメッセージは全てこのキューに格納される。最終的にどのようなプロトコルを使用し、どのようなオブジェクトの交換を可能とするかは molecule 自身の標準仕様の形としては定めない。シリアライズ形態におけるデータ表現が適切に規定されていれば、その転送プロトコルがどのようなものであるかを当必要はない。また、プロトコル処理を専門に行うプロキシーエージェントを直接転送することで、利用プロトコルを動的に変更することも考えられる。むしろ molecule における開発では、単一の標準プロトコルを採用することによってウイルスやワームなどの被害が多方面に拡大する危険性を考慮し、個別のプロトコルを採用した互いに独立なスコープを幾つも作ることが望ましい。

（8） インスタンス管理

リファレンス実装では簡単なガベージコレクタも搭載している。その構造上、Contexts の内部辞書に登録されていないインスタンスはユーザープログラム上からは参照することができない。これに基づき、Contexts の辞書に対するインスタンスの束縛状態を追跡し、全ての束縛が解除された場合に即時インスタンスを解放する形でのガベージコレクションを行っている。加えて、個別のインスタンスの即時解放を行うことも許可している。なお、現行の実装では参照カウント方式を用いており、そのままでは循環参照を含むオブジェクト群を回収することができない。しかし、インスタンスの存在メモリ領域は更に別のヒープ管理モジュールによって管理されており、ユーザープログラム中から参照されなくなったインスタンスも、強制的にマーク＆スイープ方式のガベージコレクタを起動することによって確実に回収することができる。リファレンス実装では循環参照オブジェクトに対する自動的なマーク＆スイープ方式のガベージコレクションはエージェントの最終的な解放時を除いては行われず、必要ならばプログラマは明示的にガベージコレクタを起動しなければならない。インスタンスの強制解放が行えないガベージコレクション環境においては循環参照オブジェクトは比較的容易に発生しうるが、本実装ではプログラマが注意深く強制解放を行えばそのような事態を容易に回避することができるため、ガベージコレクタの自動起動による予測しえないパフォーマンス変動の回避を重視したものである。

（9） 文字処理の安全性

リファレンス実装では内部文字コードとして符合化文字集合の識別符合を内包する独自のコード体系を採用している。これにより、文字列処理に際して符合化文字集合の情報を損なうことなく処理を行える。このことによる副次的な欠点として異なる文字集合に属するが同一として扱うべき文字を同一視できないという問題があるが、これについてはプログラマが明示的に特定符合化文字集合の文字への変換を行うことによって処理するものとした。文字集合の識別符合体系は特定の規格に沿って定められるものであるが、現時点では ISO-10646 や US-ASCII などの符号化文字集合の符号化を試験的に行っている状

況であり、今後の開発においてより詳細で広範な符号化を行う必要がある。

（１０）molecule の拡張性

molecule 処理系においては全てがモレキュールの相互作用によって表現される。操作される側のモレキュールはプリミティブやコレクションに類するモレキュールによって表現され、モレキュールの操作もまた Function モレキュールによって表現され、全体の制御も Contexts モレキュールによって司られる。結果、molecule 処理系において実際にどのような処理が記述可能であるかどうかは、どのようなモレキュールが利用可能であるかに依存する。逆に言えば、任意のモレキュールを自由に追加することさえ可能ならば、molecule 処理系によるソフトウェア製作の範囲は自在に拡張可能である。

リファレンス実装は、既に述べたようにごく標準的な範囲内の C++によって記述されている。リファレンス実装においてモレキュールを追加するにあたっては他のモレキュールと協調動作し特定のインターフェースを持つ関数を実装するだけでよく、特別に定められた API の利用などを必要としない。リファレンス実装では状態を保持した構造体と状態遷移を司る関数の組によってモレキュールを実装しているが、そもそもモレキュールは基本的には状態付き関数でしかなく、リファレンス実装とは異なる形での実装も可能である。仮に molecule 処理系を全く新規に書き起こすとしても、守るべき基本原則は、全てをモレキュールとしての基本モデルに従わせ、かつ、他の処理系と動作に互換性を持つ基本的なプリミティブなどのモレキュールを実装することだけである。現状のリファレンス実装はプリミティブとアーキテクチャモレキュール、ガベージコレクタなどを合わせても 20000 行に満たず、移植は極めて容易である。

（１１）molecule の記法

molecule は組み込みの演算子やステートメントを備えていない。これらは全てプログラマが構文宣言によって実装する。molecule において宣言が許される構文は、前置単項演算子・二項演算子・ステートメントの 3 種類である。演算子には個別に結合優先順位を割り振ることができる。プログラムのソースコードは構文宣言に基づいて構文解析され、解析結果の各処理の呼び出しには構文宣言で対応づけられたモレキュールがマッピングされる。加減乗除や if ステートメントなどの一般的な構文についてはリファレンス実装の一部として標準実装が提供されるが、これらを利用する必要は必ずしもない。必要ならばこれらの実装を変更することも、また、独自の構文を自由に追加することもできる。

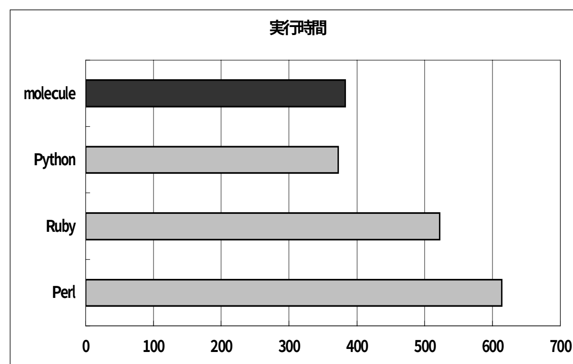
４．動作速度の評価

既存のスクリプト言語との間の基本的な処理フローにおける実行パフォーマンスの差を見るために、フィボナッチ数列を再帰呼び出しによって求めるプログラムを実行し、その実行時間を測定した。使用した処理系は、molecule(リファレンス実装α版)、Perl(ActivePerl-5.6.1.629-MSWin32-x86-multi-thread)、Python(Python-2.2b1)、Ruby(Ruby Entry Package for Win32/1.6.4j)である。OS 環境は Windows2000 であり、使用ハードウェアは PentiumIII/1GHz 搭載の PC/AT 互換機である。それぞれに次の疑似プログラムに同等な処理を行わせ、5 回の実行時間の平均値をとった。

```
fib( n){
    if( n < 2)
        then n
        else fib( n - 1) + fib( n - 2)
}

print fib( 30)
```

結果は次の通りである。



(実行時間の単位は 1/100 秒)

グラフ 1.既存スクリプト言語との動作速度比較

いずれの言語においても実行時間に占める数値計算の処理は微少なものであると考えられ、この測定結果は基本的な処理における各処理系のネイティブバイナリに対するオーバーヘッドの差を示すものと言える。なお、同環境における同等のＣプログラムの実行速度は 0.1 秒程度である。

現実のプログラムの実行パフォーマンスはむしろライブラリ実装の性能に依存するが、基本的な処理の速度はそれらの絶対的な限界速度として影響を及ぼす。測定結果に見られる molecule の速度は他の言語に比べて遜色のないものであり、この意味において、molecule には他のスクリプト言語に比べてパフォーマンス上の大きな問題はないと期待できる。

５．まとめ

molecule は一般的なスクリプト言語と同程度のパフォーマンスを備えつつ、それらの言語にはない動的な運用、特定用途に適した特性を備える。これらの特性を活用することにより、汎用のエージェントサーバーに対してサービス供給エージェントをリアルタイムにインストールする形でのシステム構築を可能にしうる。

６．参加企業及び期間

（財）京都高度技術研究所(http://www.astem.or.jp)
契約件名「プラットフォーム非依存な汎用ネットワークエージェントシステム」