Development of New Channel Access Library

Hiroshi Ikeda[1A)], Hiroyuki Sako[B)]
[A)] Visible Information Center, Inc.
440 Muramatsu, Tokai, Naka, Ibaraki 319-1112, Japan
[B)] Japan Atomic Energy Agency
2-4 Shirakata-Shirane, Tokai, Naka, Ibaraki 319-1195, Japan

Abstract

The libraries JCA[1] and CAJ[2] are used commonly for channel access in Java applications, but the API of JCA and its implementation contain many problems, so we create new channel access library. In this report we describe problems of previous libraries, some designs of new library, differences of APIs, etc.

新しいチャネルアクセスライブラリの開発

１．既存ライブラリの問題点

既存ライブラリの問題で最も大きいのは、スレッドセーフでないにも関わらずマルチスレッドによる実装を行っていることである。すなわち、これは論理的に正しく動作することが保証されず、何が起こっても不思議ではない。

スレッドセーフが守られていないために起こる障害は再現性が低く、既存ライブラリが組み込まれたアプリケーションは一見動いているように見えるかもしれない。しかし、特に、長時間の動作や高負荷によるJVMの最適化が行われるときなど、致命的なタイミングで障害が発生する可能性がある。

既存のライブラリでこの問題を回避するには、徹底してシングルスレッドを使う設定にし、シングルスレッドでアクセスを行うことである。さらに、サーバからの応答やイベントを処理するため短い間隔でポーリングを行う必要がある。

一方、オブジェクト指向的な観点からも、いくつか問題がある。

オブジェクト指向では、オブジェクトが自分自身の整合性をそれ自身が管理・カプセル化し、それらを積み上げることでプログラムを構成する。ところが、既存ライブラリでは、最も基本となるデータ構造すらその内部フィールドを外部に漏洩し、その責務を全うすることができない。

また、オブジェクト指向的プログラミングは、抽象クラス等の導入により構造的プログラミングでの依存関係の方向性を逆転することが可能であり、ソフトウェアのテーマの一つである Open-Closed Principle に対する一つの解答となっている。しかし、既存ライブラリではそもそも、クラス間・パッケージ間の相互参照性が強く、修正変更の影響が芋づる的にライブラリ全体に及ぶことになる。

他にも、Java言語固有の標準的規則やイデオムに反していたり、受信データの整合性チェックが十分でなかったりするなど、様々な、しかし軽視できない問題が含まれている。

２．設計情報

既存ライブラリJCAは、APIとJNIによる実装の2つのモジュールに分かれているが、このAPIには前述の脆弱な実装が既に含まれているため、これを使わず新しいライブラリJCALを作成した。

主要クラスの構造のイメージを図 1に、ライブラリのAPIとして定義される主要なインタフェースの概要をリスト 1に示す。

2.1 内部スレッド

チャネルの接続やデータの送受信などは、内部で単一のスレッドを用いて行い、外部と内部の境界であるAPIの実装をスレッドセーフにする。内部スレッドを単一にすることで、内部ではマルチスレッドに起因する困難さを避ける（図 1）。

一般にマルチスレッドを正しく使うとレスポンスが向上し、マルチCPUに独立したタスクを割り当てたりI/Oの切り離しなどをうまく行ったりしてCPUの稼働率を上げることでスループットを向上させることができる。しかし、スレッド間の競合で起こるコンテキストスイッチのオーバヘッドは非常に大きく、これが頻繁に発生すると実行性能は著しく劣化する。さらに、送信／受信のような逆向きの制御の流れがある場合、マルチスレッドではデッドロックを回避するのが難しい。

2.2 主API

Context, Channel, Monitorを、JCAのそれとおおよそ同じ役割を持つAPIとして定義する（図 1、リスト 1）。

Contextはライブラリの動作環境を表し、内部スレッドのライフサイクルの管理とチャネルを表す

[1] E-mail: ikeda@vic.co.jp

Channelのインスタンスを生成する責務を負う。Channelは、該当するチャネルに対して能動的に、同期的または非同期的に値の設定・取得を行い、さらに、受動的に値を取得するモニタを表すMonitorのインスタンスを生成する責務を負う。

ここで、同期的という意味は、サーバからの返答があるまで（またはタイムアウトするまで）ブロックすることを意味し、このようなメソッドはチェックされる例外InterruptedExceptionを投げるように宣言される。非同期の場合は要求だけを先に行い、与えられたリスナのコールバックによりその完了を通知させるか、または、返却値として得られたオブジェクトに対し、さらいにブロックするメソッドを（後で）呼ぶことでその完了を認識する。

ChannelおよびMonitorには、上記の目的を含むコールバックを行うためのリスナを与えるメソッドが幾つか存在する。通常、コールバックはその非同期通知という目的から、異なるスレッドによって行われる（この場合、間違いなくリスナはスレッドセーフでなければならない）。実際にどのようにコールバックを行うかは、Contextの実装クラスで決定する。例えば、GUIアプリケーションではコールバックをUIスレッドに委譲することで、スレッドセーフ性を考慮する必要がなくなるだろう。混乱を避けるため、コールバックは同時刻に高々１つのスレッドにて実行されるものとし、ChannelおよびMonitorにおける様々な値の更新は、このコールバックスレッドからのみ行う。これにより、コールバックからはChannelやMonitorの一貫性のある値にアクセス可能となる。逆に、コールバックは速やかに終了させる必要があり、そうでなければ、ChannelやMonitorの値は変化せず、次のイベントのコールバックも行われない。

なお、ChannelおよびMonitorは、コールバックの取りこぼしを防ぐため、メソッドopenにより明示的に動作を開始する。

本ライブラリへのアクセスは、Contextのインスタンスの生成から始まる。Contextの具象クラスは、今後のライブラリの洗練によって内容の変更が予想されるため、これらのAPIとは別パッケージに配置する。

2.3 データ構造

チャネルアクセスによって送受信を行うデータ構造の単位は、インタフェースDbrによって表わされる（リスト 1）。データ構造は、数値や文字列の配列とタイムスタンプ等の属性の組み合わせから構成されるが、その組み合わせ爆発を避けるため、属性はコンポジションとして表す。

データ構造の実装クラスは不変クラスとする。不変クラスのインスタンスへのアクセスはスレッド間で同期する必要がなく、故にスレッドセーフ性やスレッド間の競合による実行性能の劣化を全く心配する必要がない。

2.4 通信

本ライブラリの通信部は、チャネルアクセスのプロトコルの仕様により、同じサーバへのチャネルおよびモニタの通信は多重化され1つのTCP接続にて行われる。このため、やや複雑な構造になる。この他、サーバの捜査やリピータとの送受信を行うUDP接続がそれぞれ1つずつ用いられ、これらの処理を行うクラスは“～Transport”という名称を付けている（図 1）。リピータ自体は、Javaの標準ライブラリではパケットの送信元アドレスを自由に設定できないため、完全には実装できない。またライフサイクルがそもそも異なり、独立したサーバアプリケーションとして動作させるのが望ましい。クライアント側にとっては新規サーバ検出のブロードキャストを多少抑えることができる程度なので、運用段階でなければ無くても問題ない。このため、本ライブラリでは実装しない。

３．既存ライブラリのAPIへのアダプタ

既存ライブラリJCAを用いてすでに作成されているアプリケーションに対し、新しいライブラリJCALを用いて書き換えるのは作業を要する。このため、JCAのAPI経由でJCALを使うためのアダプタJCA-JCALを作成している。

JCAを用いたアプリケーションでは、JCALibraryの静的メソッドcreateContextにより、JCAのContextの実装クラス名を指定してそのインスタンスを生成し、これを基にチャネルアクセスの処理を行う。そこで、指定するクラス名をJCALで実装したものに変更すればよい。

ただし、そもそもJCAのAPIに不適切な実装が含まれているために新しいAPIを作成したのであり、JCAの挙動を完全にトレースするのは困難である。以下にその主な制限・違いを述べる。

(1) Contextのイベントループは JCALでは内部スレッドで自動的に行われるため必要がなく、その実装は、空実装または単に一定時間休止するだけである (2) Context のリスナは使われない (3) コールバックは高々１つのスレッドで呼び出され、その間はチャネルやモニタの値は変化しない。また、コールバックをブロックしてはいけない (4) チャネルやモニタのコールバックは、プロトコルCA_PROTO_ERRORによるサーバからのエラー通知はコールバックされない。また、タイムアウトを含むクライアント側で判断する例外は、コールバックされない。これらは、単にログとして出力する。また、ステータスが“NORMAL”以外のレスポンスは、データ自体は（バイナリ的に）0で埋めた値が返ってくるが、この場合はデータをnullとしたイベント引数でコールバックされる。

４．予定・課題

そもそもの開発の動機は、チャネルアクセスを用いたクライアント側のJavaアプリケーションを作る際にJCA/CAJが安全に使えないことが分かったため

であり、第1の目標としては、そのような簡単なクライアント側のアプリケーションが作成できる程度の機能を有する簡易的なライブラリを作成することであった。

これが一般的な用途にどの程度使用できるか、あるいはどのように拡張修正すれば良いかは、様々なアプリケーションに本ライブラリの適用を試みてフィードバックを得ることが必要となるだろう。

プロトコルに対する処理やコード変換は、クライアント側に必要な一部だけを実装している。これを充実させれば、リピータやテストのための擬似的なサーバの作成が可能になる。ただし、そもそも、プロトコルやデータのコード変換に関する仕様の情報は不十分・不正確なところがあり、実際に稼働している既存のコードを丁寧に調べる必要がある。

本ライブラリは幾つかの外部ライブラリに依存しているが、一部の機能しか使用していない外部ライブラリに関しては、バージョン間の互換性の問題を避けるため、同等の機能を内部で作ってしまうことが考えられる。

本ライブラリの中核はインタフェースで組まれるが、実際に使う場合はその実装が必要になってくる場合がある。もし、その実装をライブラリのクライアント側で作る必要があるならば、似たような実装クラスが方々に出来上がってしまうだろう。このため、APIにあるインタフェースの標準的な実装クラスもAPIにしておくべきと思われるが、このような便利な（あるいは余計な）クラスを多く作成してAPIとしてしまうと、ライブラリのクライアントが実装に依存することになり、ライブラリの修正変更が難しくなってしまうだろう。しかし、APIの境界を明確にしないと、ライブラリの修正変更に対する互換性の定義ができなくなる。現在のところ、APIの範囲は明確には決めていない。

参考文献

[1] http://jca.cosylab.com/
[2] http://caj.cosylab.com/

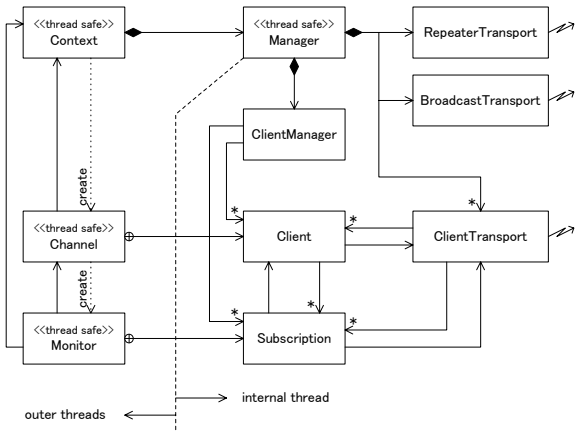

図 1　ライブラリのクラス構造のイメージ

```
public interface Dbr {
  DbrType type();
  int size();
  Attribute attribute();

  byte byteValue() throws ConvertException;
  int intValue() throws ConvertException;
  short shortValue() throws ConvertException;
  float floatValue() throws ConvertException;
  double doubleValue() throws ConvertException;
  String stringValue() throws ConvertException;

  byte byteValue(int index) throws ConvertException;
  int intValue(int index) throws ConvertException;
  ...

  byte[] byteArray() throws ConvertException;
  int[] intArray() throws ConvertException;
  ...
}

public interface Context {
  void start() throws ContextException;
  void shutdown();
  void shutdownNow();
  boolean isTerminated();
  boolean awaitTermination(long timeout) throws InterruptedException;

  Channel createChannel(String name);
  Channel createChannel(String name, short virtualCircuitPriority);
}

public interface Channel {
  String name();
  void open();
  void close();

  boolean isConnected();
  DbrProfile dataProfile();

  void addConnectionListener(ConnectionListener listener);
  void removeConnectionListener(ConnectionListener listener);

  AccessRights accessRights();

  void addAccessRightsListener(AccessRightsListener listener);
  void removeAccessRightsListener(AccessRightsListener listener);

  CaValue get() throws InterruptedException, ExecutionException;
  CaValue get(DbrType type, int count)
      throws InterruptedException, ExecutionException;
  CaValue get(DbrTypeExtender extender, int count)
      throws InterruptedException, ExecutionException;

  void get(GetListener listener);
  void get(DbrType type, int count, GetListener listener);
  void get(DbrTypeExtender extender, int count, GetListener listener);

  CaValueFuture asyncGet();
  CaValueFuture asyncGet(DbrType type, int count);
  CaValueFuture asyncGet(DbrTypeExtender extender, int count);

  void put(Dbr data);
  void put(Dbr data, PutListener listener);

  void syncPut(Dbr data) throws InterruptedException, ExecutionException;
  CaWaiter asyncPut(Dbr data);

  Monitor createMonitor(EnumSet<MonitorMask> mask);
  Monitor createMonitor(
      DbrType type, int count, EnumSet<MonitorMask> mask);
  Monitor createMonitor(
      DbrTypeExtender extender, int count, EnumSet<MonitorMask> mask);

  // put の簡易メソッド群
  void put(byte value);
  void put(short value);
  ...
}

public interface Monitor {
  Channel getChannel();

  void open();
  void close();

  void addMonitorListener(MonitorListener listener);
  void removeMonitorListener(MonitorListener listener);

  CaValue currentValue();
}
```

リスト 1　主要なAPIの概要