ネイティブ・オブジェクトからのイベントをJavaコードで処理する

オブザーバー・パターンとプロキシー・パターンで実装を容易に

Sachin Agrawal
Senior Software Engineer
IBM Software Labs India

2006年 1月 31日

Vijil Chenthamarakshan (vijil.e.c@in.ibm.com)
Staff Software Engineer
IBM Software Labs, India

他の言語で書かれたオブジェクトを使う場合、ネイティブのイベント・ソースとJava™リスナーとの間の通信は厄介なものですが、特にマルチスレッド環境では非常に面倒です。この記事では、ネイティブ・コードからJVMへのイベント通信を透明な形で扱う設計パターンを利用することによって、伝統的なネイティブ・ライブラリーを効果的に扱う方法について解説します。

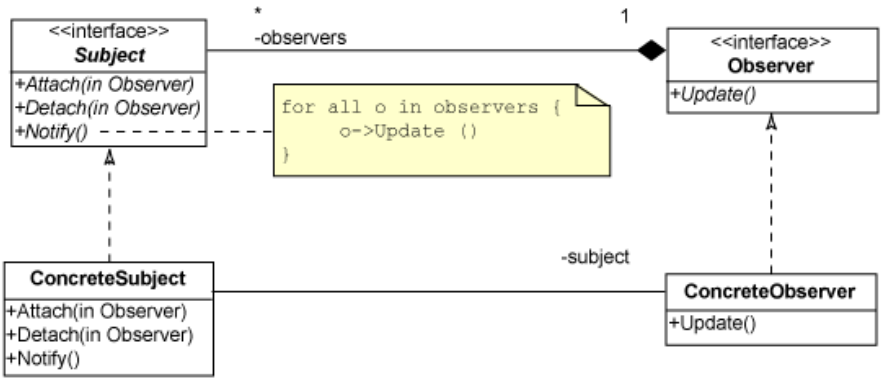
オブジェクト指向のシステムでは、あるオブジェクトが一連のイベントを起動することができます。Javaプログラミング言語では、オブザーバー設計パターンに基づいてイベント・リスナーを定義できるようになっていますが、他の言語で書かれたオブジェクトを使う必要がある場合には、それだけでは不十分です。ネイティブのイベント・ソースとJavaリスナーとの間の通信にJNI（JavaNative Interface）を使うのは厄介なものですが、マルチスレッド環境では特に面倒です。この記事では、ネイティブ・コードからJVMへのイベント通信を透明な形で扱うための設計パターンについて説明します。この設計パターンを使うと、レガシーのネイティブ・アプリケーションにJavaインターフェースを付加したり、あるいはJavaリスナーを念頭に置いてネイティブ・アプリケーションを構築したりできるのです。

オブザーバー設計パターン

オブザーバー設計パターンでは、イベント・リスナーとイベント・クリエーターとの間に、多対一の依存関係を定義します。イベント・クリエーターがイベントを起動すると、そのイベントのリスナーすべてに対してイベントが通知されます。イベント・クリエーターとリスナーは分離されているため、それぞれを独立に使用し、変更することができます。この設計パターンはイベント駆動型プログラミングの中心として、SwingやSWTなどのGUIフレームワークで広く使用されています。

オブザーバー設計パターンの実装は、アプリケーション全体がJava言語で書かれている場合には、ごく単純です。図1のクラス図は、その一例です。

図1. オブザーバー設計パターン

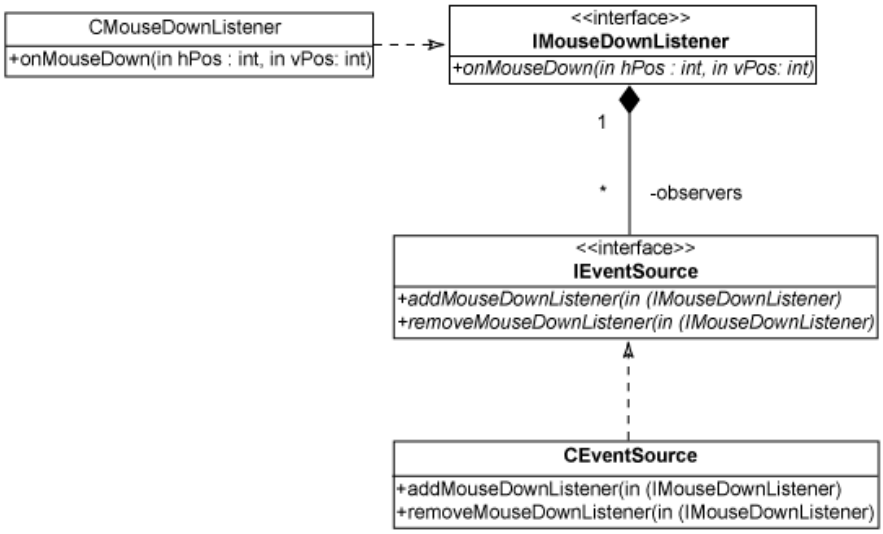


ネイティブ・アプリケーション用のJavaリスナー

しかし、場合によると、ネイティブ・アプリケーションに対してJavaリスナーをサポートしたい場合があります。そうした場合には、アプリケーションの大部分（アプリケーションのエントリー・ポイントを含めて）はネイティブ・コードで書かれているかも知れず、またユーザー・インターフェースとの対話方法として、アプリケーションがイベントを生成するかも知れません。そうした場合にJavaベースのUIをサポートするために最善の方法は、Javaクラスが自分を、アプリケーションが生成する様々なイベントのリスナーとして登録してしまうことです。簡単に言うと、Java言語で書かれたリスナーをサポートすることによって、ネイティブ・アプリケーションを『Javaイネーブル』にしたいというわけです。

図2に示すクラス図は、サンプル・シナリオを示しています。CEventSourceクラスはC++で書かれています。このクラスは、addMouseDownListener()メソッドと removeMouseDownListener() メソッドを使って、このクラスのリスナーを、「マウス下げ(mousedown)」イベントに対して登録、登録解除します。このクラスは、全てのリスナーがaddMouseDownListener()インターフェースを実装していることを想定しています。

図2. サンプル・シナリオのクラス図



IMouseDownListenerはC++の抽象クラスであることに注意してください。どのようにすれば、Javaリスナーと CEventSourceクラスとの間にコンパイル時バインディングを必要とせずに、Javaクラスをイベントに登録できるのでしょう。またそのJavaクラスをイベントに登録した後、CEventSourceクラスは、どのようにしてJavaクラスの中にあるメソッドを呼び出すのでしょう。ここで、Java呼び出しAPI（JavaInvocation API）が登場するのです。

呼び出しAPI

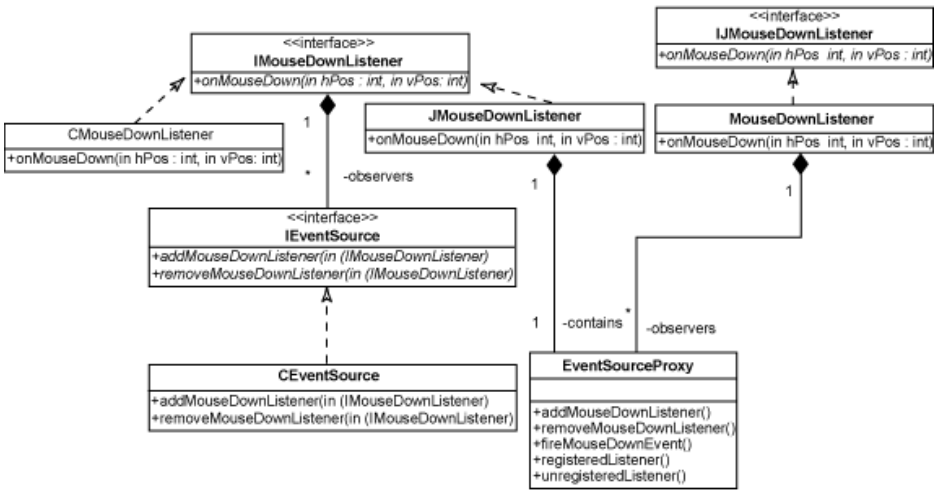
この呼び出しAPIを利用すると、（明示的にJVMソースとリンクすることなく）ネイティブ・アプリケーションの中にJVMをロードすることができます（参考文献を見てください）。jvm.dllの中のファンクションを呼び出すことによってJVMを作成するのです。また、これによって、カレントのネイティブ・スレッドがJVMに付加されます。そうすると、JVMの中にあるすべてのJavaメソッドを、ネイティブ・スレッドから呼べるようになります。

ただし、呼び出しAPIによって問題を完全に解決できるわけではありません。つまり、CEventSourceクラスが、Javaリスナーとコンパイル時依存関係を持つのは望ましくありません。また、リスナーをCEventSourceに登録するためにJNIを使わせるという責任を、Javaリスナーに負わせるべきではありません。

プロキシー設計パターン

こうした問題は、プロキシー設計パターンを使うことによって避けることができます。一般的な形でのプロキシーは、別のオブジェクトに対するプレースホルダーです。クライアント・オブジェクトはプロキシー・オブジェクトを相手に動作し、一方プロキシーは、ネイティブ・メソッドを使うための詳細すべてをカプセル化するのです。プロキシー・パターンの詳細を図3に示します。

図3. プロキシー設計パターンの例



EventSourceProxyはCEventSourceクラスのプロキシーです。クライアントが自分をリスナーとして登録するためには、IJMouseDownListenerインターフェースを実装する必要があります。このインターフェースはIMouseDownListenerに似ていますが、Javaコードで書かれています。最初のクライアントが自分のaddMouseDownListener()メソッドを呼び出すと、そのクライアントはregisterListener() ネイティブ・メソッドを使って、自分をCEventSourceのリスナーとして登録します。registerListener()メソッドはC++で実装されており、JMouseDownListenerオブジェクトを作成し、このオブジェクトをCEventSourceのリスナーとして登録します。JMouseDownListenerのonMouseDownListener()メソッドは、そのonMouseDownイベントが起動すると、呼び出しAPIを使ってEventSourceProxyに通知します。

またEventSourceProxyは、自分に登録された一連のリスナーも保持しています。EventSourceProxyのonMouseDownが起動すると、そうしたリスナーすべてに対して通知されます。ただし、このイベントに対して複数のJavaリスナーが存在したとしても、このプロキシーの1つのインスタンスしかCEventSourceには登録されないことに注意してください。プロキシーは、onMouseDownイベントを、全リスナーに代行させるのです。これによって、ネイティブ・コードとJavaコード間での不必要なコンテキスト・スイッチを防止することとができます。

マルチスレッドの問題

ネイティブ・メソッドは、JNIインターフェース・ポインターを引数として受け取ります。しかし、イベントを、自分に関連付けられたJavaプロキシーに逆に代行させようとするネイティブ・リスナーは、既製のJNIインターフェース・ポインターを持っていません。そのためJNIインターフェース・ポインターをいったん取得したら、後で使えるようにメモリーの中に保存する必要があります。

JNIインターフェース・ポインターは、カレント・スレッドの中でのみ有効です。JNIを実装するJVMは、スレッド・ローカルなデータを、JNIインターフェース・ポインターが指す領域に割り当て、保存します。これはつまり、JNIインターフェース・ポインターも、スレッド・ローカルなデータの中に保持する必要がある、ということです。

JNIインターフェース・ポインターは、次の2つの方法で取得することができます。

- いったんスレッドがJNI_CreateJavaVMを使ってJVMを作成すると、JNIは自分の2番目のパラメーターが指定する位置にインターフェース・ポインターの値を置きます。これによって、この値はスレッド・ローカルな領域に保存されることになります。
- プロセス中の他のスレッドによってJVMが既に作成されている場合には、カレント・スレッドはAttachCurrentThreadを呼びます。JNIは、自分の最初のパラメーターが指定する位置にインターフェース・ポインターの値を置きます。

しかし、これですべてが終わったわけではありません。プログラミングはC/C++で行われていることを思い出してください。Java言語で行われるような自動的なガーベジ・コレクションは利用できないのです。JNIコールによってスレッドが完了したら、そのスレッドはDetachCurrentThreadを呼ぶことによって、インターフェース・ポインターを解放する必要があります。このコールが行われず、スレッドが存在していると、そのプロセスはきれいに終了しません。逆に、今や存在しなくなったスレッドがDestroyJavaVMコールの中でJVMから分離されるまで、そのスレッドを永遠に待ち続けるのです。

サンプル・コードの中では、これらのすべては、様々な場所にあるCJvmクラスの中にカプセル化されています。また、プライマリー・スレッドとは別のワーキング・スレッドからイベントを起動することによって、マルチスレッドがデモされています。

環境設定とサンプル・コード

これでサンプル・コードをビルドし、実行する準備が整いました。

共通インターフェース

IMouseDownListenerインターフェースとIEventSourceインターフェースは、common.hの中で定義されています。IMouseDownListenerには、onMouseDown()という1つのメソッドしかありません。このメソッドは、画面上でマウスがクリックされた位置を受け取ります。IEventSourceインターフェースには、リスナーを登録、登録解除するための、addMouseDownListener()メソッドとremoveMouseDownListener() メソッドがあります。

Java呼び出しAPI用のヘルパー・ルーチン

Java呼び出しAPIを単純化して使うために頻繁に要求される7つのユーティリティーが、Jvm.hの中に定義されており、Jvm.cppで実装されています。

- CreateJavaObject() は、クラス名と、ネイティブIEventSourceへのハンドルが与えられると、Javaオブジェクトを作成します。作成されたJavaオブジェクトは、このネイティブ・ハンドルからイベントをリスンしようとします。このハンドルは、そのコンストラクターを通してオブジェクトに渡されます。
- ReleaseJObject() は、Javaオブジェクトのrelease() メソッドを呼び出します。このメソッドが、オブジェクトのリスナーをEventSourceProxyから登録解除することになっています。
- DetachThread() は、（カレント・スレッドがJVMに付加されている場合には）カレント・スレッドをJVMから分離します。このコールは、（スレッドが付加されている間にJNIのために割り当てられていた）スレッド特有のリソースを解放するために必要です。

その他は自明でしょう。

- CreateJVM() 英語のまま

- DestroyJVM()
- GetJVM()
- GetJNIEnv()

CJvmクラスは、スレッド特有な位置でJNI環境ポインターも保持します。JNI環境ポインターはスレッド依存であるため、これが必要です。

ネイティブ・イベント・ソース

CEventSourceは、EventSource.hとEventSource.cppの中にあるIEventSourceインターフェースの、単純で素直な実装です。

ネイティブ・イベント・リスナー

CMouseDownListenerは、MouseDownListener.hとMouseDownListener.cppの中にあるIMouseDownListenerインターフェースの実装です。このネイティブ・リスナーは、単に説明用に書かれたものです。

エントリー・ポイント

main.cppには、main() とThreadMain() が含まれています。main() は、ネイティブEventSourceとネイティブ・リスナー、そしてJavaリスナーを作成します。そしてスレッドを作成し、数秒間スリープすることによって、そのスレッドを実行させます。最後に、Javaリスナーを解放し、JVMを破棄します。

ThreadMain() は単純にイベントを起動してから、自分をJVMから分離します。

Javaモジュール

IJMouseDownListener.javaの中のIJMouseDownListenerは、Javaプラットフォームに対するネイティブ・インターフェースの単純なクローンです。

MouseDownListenerはJavaでのサンプル・リスナーであり、MouseDownListener.javaの中で実装されています。これは、そのコンストラクターの中で、ネイティブのEventSourceハンドルを受け取ります。また、そのリスナーをEventSourceProxyから登録解除するrelease()メソッドを定義します。

EventSourceProxyは、ネイティブ・モジュールからのEventSourceに対するプレースホルダー、あるいは代理（surrogate）であり、EventSourceProxy.javaの中で実装されています。また、プロキシーを実際のEventSourceにマッピングするための静的ハッシュ・テーブルを保持します。

addMouseDownListener() あるいはremoveMouseDownListener() を使うと、Javaリスナーの集合を保持することができます。1つのネイティブEventSourceは1つ以上のJavaリスナーを持つことができますが、プロキシーは、必要な場合しかネイティブEventSourceに対する登録/登録解除を行いません。

ネイティブEventSourceからイベントを転送する間、EventSourceProxyのネイティブ実装は、fireMouseDownEvent()を呼び出します。このメソッドはJavaリスナーのハッシュ・セットに従って繰り返しを行い、それらのJavaリスナーに対して通知を行います。

EventSourceProxyのネイティブ部分は、自分自身へのグローバル参照も保持します。これは、後でfireMouseDownEvent()を起動するために必要です。

サンプル・コードをビルドし、実行する

サンプル・コードの中のJavaクラスは、どれも通常の手続きを使ってビルドされており、特別なステップは必要ありません。EventSourceProxyをネイティブ実装するためには、下記のようにjavahを使ってヘッダー・ファイルを生成する必要があります。

```
javah -classpath .\java\bin events.EventSourceProxy
```

Win32プラットフォーム用のC++モジュールをビルドするために、Microsoft DeveloperStudioプロジェクト・ファイルとcpp.dswワークスペースを提供してあります。このワークスペースを開き、単純にmainプロジェクトをビルドします。ワークスペース内の全プロジェクトは、適当な依存関係に関連付けられています。DeveloperStudioがJNIヘッダーとコンパイル時JNIライブラリーを発見するのを確認してください。そのためには、Tools > Options > Directories というメニュー項目を選択します。

無事ビルドできたら、このサンプル・プログラムを実行する前に、幾つかのステップを実行する必要があります。

まず、これまではJDKがJavaクラスをビルドしており、JNIヘッダーも含んでいること、またライブラリーがそのJava呼び出しAPI用のランタイム・コンポーネント（例えばjvm.dll）を持っている場合もあるため、これを設定する必要があります。最も単純な手法は、PATH変数を更新することです。

次に、mainプログラムは、コマンドライン引数をとりますが、これは単なるJVM引数です。JVMには、次のように少なくとも2つの引数を渡す必要があります。

```
main.exe "-Djava.class.path=.\\java\\bin"
"-Djava.library.path=.\\cpp\\listener\\Debug"
```

その結果、コンソール出力は次のようなものになります。

```
In CMouseDownListener::onMouseDown
        X = 50
        Y = 100
In MouseDownListener.onMouseDown
        X = 50
        Y = 100
```

コンソール出力から分かる通り、Javaリスナーは、（説明用にビルドされた）ネイティブ・リスナーと同じ結果を生成します。

まとめ

この記事は、ネイティブ・アプリケーションが生成するイベントへのリスナーとしてJavaクラスを登録する方法について説明しました。オブザーバー設計パターンを使うことによって、イベント・ソースとイベント・リスナーとの間の結合を減少させることができます。また、プロキシー

設計パターンを使えば、イベント・ソースの実装の詳細をJavaリスナーから隠すこともできます。この2つの設計パターンを組み合わせることによって、既存のネイティブ・アプリケーションにJavaのUIを追加できるのです。

著者について

Sachin Agrawal

Sachin Agrawalは、数年来、C++に積極的に取り組んできました。そして、様々なコンパイラーのC++オブジェクト・モデルの研究に、継続して取り組んでいます。彼はインドのIBM Software Labsに勤務しています。

Vijil Chenthamarakshan

Vijil ChenthamarakshanはIBM Software Labsに勤務しています。関心を持っている領域としては、開発者がソフトウェアの複雑さに対して上位レベルで作業できるようにするための、非構造化情報用の管理システムや仕組みなどです。