BEA WebLogic Server

WebLogic Event ユーザーズ ガイド（非推奨）

WebLogic Server バージョン 6.1
マニュアル第 6.1 版
2001 年 11 月 30 日

著作権

Copyright © 2001 BEA Systems, Inc. All Rights Reserved.

限定的権利条項

本ソフトウェアおよびマニュアルは、BEA Systems, Inc. 又は日本ビー・イー・エー・システムズ株式会社（以下、「BEA」といいます）の使用許諾契約に基づいて提供され、その内容に同意する場合にのみ使用することができ、同契約の条項通りにのみ使用またはコピーすることができます。同契約で明示的に許可されている以外の方法で同ソフトウェアをコピーすることは法律に違反します。このマニュアルの一部または全部を、BEA からの書面による事前の同意なしに、複写、複製、翻訳、あるいはいかなる電子媒体または機械可読形式への変換も行うことはできません。

米国政府による使用、複製もしくは開示は、BEA の使用許諾契約、および FAR 52.227-19 の「Commercial Computer Software-Restricted Rights」条項のサブパラグラフ (c)(1)、DFARS 252.227-7013 の「Rights in Technical Data and Computer Software」条項のサブパラグラフ (c)(1)(ii) 、NASA FAR 補遺 16-52.227-86 の「Commercial Computer Software--Licensing」条項のサブパラグラフ (d)、もしくはそれらと同等の条項で定める制限の対象となります。

このマニュアルに記載されている内容は予告なく変更されることがあり、また BEA による責務を意味するものではありません。本ソフトウェアおよびマニュアルは「現状のまま」提供され、商品性や特定用途への適合性を始めとする（ただし、これらには限定されない）いかなる種類の保証も与えません。さらに、BEA は、正当性、正確さ、信頼性などについて、本ソフトウェアまたはマニュアルの使用もしくは使用結果に関していかなる確約、保証、あるいは表明も行いません。

商標または登録商標

BEA、WebLogic、Tuxedo、および Jolt は BEA Systems, Inc. の登録商標です。How Business Becomes E-Business、BEA WebLogic E-Business Platform、BEA Builder、BEA Manager、BEA eLink、BEA WebLogic Commerce Server、BEA WebLogic Personalization Server、BEA WebLogic Process Integrator、BEA WebLogic Collaborate、BEA WebLogic Enterprise、および BEA WebLogic Server は、BEA Systems, Inc. の商標です。

その他の製品名はすべて、関係各社の商標である場合があります。

WebLogic Event ユーザーズ ガイド

マニュアルの版数	日付	ソフトウェアのバージョン
6.1	2001 年 11 月 30 日	BEA WebLogic Server 6.1

目次

1. WebLogic Events の概要

WebLogic Event のアーキテクチャ 1-2
トピック ツリー 1-2
トピック ツリーの構造 1-2
構造化ツリーの例 1-3
イベントへの関心の登録 1-4
トピック ツリーの作成方法 1-4
クライアントがイベント トピックへの関心を登録する方法 1-4
クライアントがイベント トピックへの関心を登録解除する方法 1-5
イベントの処理 1-5
トピック ツリーの走査方法 1-5
各 EventRegistration の処理方法 1-6
EventRegistration によるイベントの評価方法 1-7
アクション プロセスの動作の仕組み 1-8
パラメータの詳細 1-8

2. WebLogic Event オブジェクトとそのクラス

Evaluate オブジェクトと Action オブジェクト 2-2
EvaluateDef インタフェースと ActionDef インタフェース 2-3
実装対象のメソッド 2-3
EventTopic オブジェクト 2-5
EventRegistration オブジェクト 2-6
EventMessage オブジェクト 2-9
ParamSet オブジェクトと ParamValue オブジェクト 2-10
ParamSet の効率的な使い方 2-12
WebLogic Event を使った実装 2-14
Evaluate クラスの作成 2-15
手順 1. パッケージのインポート 2-15

手順 2. registerInit() メソッド........2-16
手順 3. evaluate() メソッド........2-17
EvaluateStocks（エバリュエータ）クラスのコード........2-17
Action クラスの作成........2-19
手順 1. パッケージのインポート........2-19
手順 2. registerInit() メソッド........2-19
手順 3. action() メソッド........2-20
MailStockInfo（アクション）クラスのコード........2-21
イベントへの関心の登録........2-22
手順 1. パッケージのインポート........2-22
手順 2. コマンドライン引数のチェック........2-22
手順 3. コマンドライン引数の処理........2-23
手順 4. EventServices ファクトリの取得........2-23
手順 5. 登録の作成と提出........2-24
Register クラスのコード........2-26
WebLogic Server へのイベント送信........2-28
手順 1. パッケージのインポート........2-28
手順 2. コマンドライン引数のチェック........2-28
手順 3. コマンドライン引数の処理........2-28
手順 4. イベントの提出........2-29
SendEvents クラスのコード........2-30
クライアントサイド通知の使い方........2-32
WebLogic レルム内での WebLogic Event 用 ACL のセットアップ........2-33

1 WebLogic Events の概要

WebLogic Event API は、パブリッシュ / サブスクライブ パラダイムを使用した軽量のイベント管理システムを提供します。たとえば、WebLogic/JDBC クライアントは、WebLogic Server にイベントを送信（パブリッシュ）できます。WebLogic Server の他のクライアントは、これらのイベントへの関心を登録（つまりイベントをサブスクライブ）できます。新しいイベントが発生すると、WebLogic Server はそれをサブスクライバに通知します。

クライアントは、エバリュエータと呼ばれる条件を指定することができ、その条件が満たされないと、イベントはクライアントに送られません。エバリュエータは不必要なネットワーク トラフィックを防止できます。エバリュエータは WebLogic Server 上で実行されます。

クライアントは、イベントが発生したときのアクションも指定します。イベントの結果から発生するアクションは、サーバ サイドにもクライアント サイドにも実装できます。このマニュアルの「イベントへの関心の登録」を参照してください。

サービスの 1 つである WebLogic Event は、JDBC、RMI、ロギング、インスツルメンテーション、ワークスペースなど、WebLogic の他のすべてのサービスを利用することができます。これらのサービスはすべて、WebLogic に統合されています。これらの API は共通の側面を多く共有しているので、複雑なネットワーク アプリケーションの構築が容易になります。アプリケーションは複数のサービスを利用でき、それらはすべてオブジェクトとクライアント リソースへのアクセスを共有することができます。

どの WebLogic Server も他のサーバ上のイベントに対してパブリッシュとサブスクライブを同時に行えるので、複数の WebLogic Server が WebLogic クラスタとして連動して、通知と登録を管理することができます。

WebLogic Server は、JavaSoft の Java Messaging Service（JMS）仕様を実装しています。WebLogic Event を使用できるどのアプリケーションでも、WebLogic JMS を使用できます。WebLogic JMS は、WebLogic Event に備わっていない、メソッド永続性、ポイントツーポイント メッセージング、保証付きメッセージ配信シーケンスなどの機能を提供します。WebLogic JMS は業界標準インタフェースなので、WebLogic JMS を使用して新しいイベントベース アプリケー

ションを実装することをお勧めします。もちろん、JMS が提供する高度な機能を必要としないアプリケーションでは、WebLogic Event を使用してもかまいません。WebLogic Event サービスは小規模で高速に動作しますが、JMS に比べると機能が限定されています。WebLogic JMS の詳細については、『WebLogic JMS プログラマーズ ガイド』を参照してください。

WebLogic Event のアーキテクチャ

トピック ツリー

トピック ツリーは、WebLogic Event のアーキテクチャ上の主要機能です。トピック ツリーは WebLogic Server 上に存在し、その中には、クライアントがサブスクライブしたすべてのイベント トピックが入っています。これは、WebLogic クライアントが WebLogic Event のサブスクライブおよびパブリッシュを行うときに、それらのイベントを記憶し処理するのに用いられるデータ構造です。

トピック ツリーの構造

このツリー構造によって、イベント タイプをカテゴリへ、さらにサブカテゴリへグループ化できるようになり、ツリー内の分岐は、その分岐元のイベントのサブカテゴリを表します。適切に構成されたトピック ツリーでは、ルート ノードからリーフ ノードへ進むにつれて、イベント トピックはより具体的なものになります。

ツリー内のイベントを記述するための表記は、ドメイン アドレスのドット表記に似ています。それぞれの単語が、ツリー内の特定の分岐でのイベントを表します。たとえば、*comms.devices.telephone.ring* または *comms.devices.telephone.page* です。これにより、クライアントは、完全イベント修飾子を使って特定のイベント トピックをサブスクライブできるようになります。また、このモデルによって、クライアントは分岐レベルへの関心を指定するだけで、イベント トピックの一般カテゴリをサブスクライブできるようにもなります。たとえば、*comms.devices.telephone* の場合、電話に関係するあらゆるイベントをリスンすることになります。

ただし、ツリーの編成は、WebLogic のフレームワークを構成するクライアント アプリケーションが処理する必要があります。開発者は、イベントを適切に組織化して、この構造を最大限に利用するようにシステムをプログラミングする必要があります。

構造化ツリーの例

どのトピック ツリーでも頂点（つまりルート）には、本質的に「あらゆる種類のイベント」を意味するワイルドカード トピックがあり、これはアスタリスク (*) で表記されます。それ以外のトピックはすべて、ルート トピックより具体的なものとみなされます。ルート トピックへの関心を登録するアプリケーションは、WebLogic Server 上で発生する、トピック ツリー内のすべてのイベントを評価することができます。

図 1-1 WebLogic のトピック ツリー

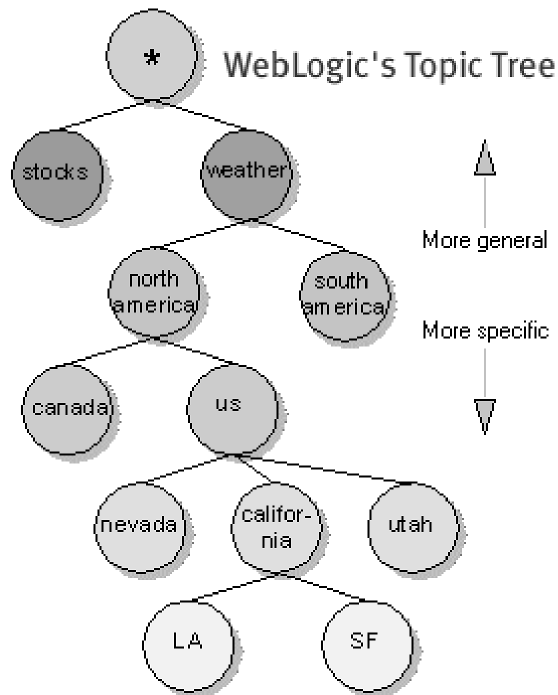

図 1-1 に示された例には、ルートから派生するトピックの大きな分枝が 2 つあります。stocks（株式）と weather（天気）です。ここで作成したのは、カリフォルニア州の 2 つの都市、ロサンゼルスとサンフランシスコの天気への関心を登録するための典型的なトピック ツリーです。これらのトピックは、以下のように表記されます。

weather.northamerica.us.california.la

および

weather.northamerica.us.california.sf

イベントへの関心の登録

トピック ツリーの作成方法

トピック ツリーは、クライアントがイベント トピックをサブスクライブするときに、WebLogic Server 内部に動的に作成されます。クライアントがトピック ツリー内に存在しないイベント トピックをサブスクライブすると、新しいノードと、そのノードに到達するのに必要な新しい分枝が自動的に作成されます。これで、サブスクライブ側のクライアントは、新しいイベントが発行されるたびにその通知を受け取ることになります。

クライアントがイベント トピックへの関心を登録する方法

WebLogic クライアントは、イベントが発行されたときにそれらを評価し、それらに基づいて動作するために、トピックへの関心を WebLogic Server に登録しなければなりません。ネットワーク上のどの WebLogic クライアント アプリケーションも、WebLogic EventRegistration サービスを通じて、任意の数のイベント トピックへの関心を登録することができます。

登録は、通常は以下の情報と一緒に WebLogic Server に提出されます。

どのイベントをサブスクライブするか。これは登録パラメータで記述されます。

- イベントが発生したときに、それをどのように評価するか。これは、`EvaluateDef` オブジェクトを通じて指定されます。

- イベントの評価結果が true の場合には、何が起こるか。`ActionDef` オブジェクトを通じて指定されます。

この点については、後でコード例を使って詳しく説明します。「イベントへの関心の登録」を参照してください。

クライアントがイベント トピックへの関心を登録解除する方法

クライアント アプリケーションは、以下の 2 つの方法のいずれかで関心を登録解除することができます。

count プロパティを使用して、関心をいつ登録解除するかを制御します。イベント登録期間を制御する方法は複数あります。

- `EventRegistration.unregister()` メソッドを呼び出します。

注意： `evaluate()` メソッドと `action()` メソッドが両方とも WebLogic Server に存在する場合のイベント登録については、クライアントが関心の登録を解除する必要があります。一方、`action()` メソッドがクライアントに存在する場合には、WebLogic クライアントが接続を解除すると、登録解除が自動的に行われます。

イベントの処理

この節では、イベントの伝播の仕組みについて説明します。これを理解すれば、ネットワーク アプリケーション内での WebLogic Event の使い方を理解する上で役に立ちます。

トピック ツリーの走査方法

どのようなアプリケーションでも、WebLogic Server にイベントを提出することができます。イベントは、そのスコープを限定する一連のイベント パラメータを付けて提出されます。イベントが提出されると、WebLogic Server は、トピック ツリー内の特定のイベントに完全に一致するものを見つけようとします。それが見つかった場合、その EventTopic の EventRegistration が処理されます（次に説明します）。完全に一致するものが見つからない場合、またはその

EventTopic への関心を登録したクライアントがない場合、そのイベントは、この時点で、送信されないものとみなされます。次いで、具体性の一段低い EventTopic までトピック ツリーをさかのぼり、そのノードの EventRegistration が処理されます。この処理がトピック ツリーの頂点に達するまで繰り返されます。

各 EventRegistration の処理方法

特定の EventTopic に関心がある各クライアントは、そのトピックに対する EventRegistration を登録済みでなければなりません。そのため、トピック ツリー内の各 EventTopic は EventRegistration のリストを備えており、このリストには各クライアントがその EventTopic にどのように関心を持っているかが記述されています。ある EventTopic がある Event に一致すると、そのイベント トピックは各 EventRegistration を以下のように処理します。

- EventTopic が Event に完全に一致する場合、その Event は EventRegistration によって評価されます（つまり、以下の条件は省かれます)。
- ただし、これが実際にパブリッシュされたイベントよりも具体性の低いイベント トピックである場合、つまり、それがトピック ツリーの上位にあるイベント トピックである場合には、以下の点が考慮されます。
  - そのイベントがより具体的なイベント トピックに送信されなかった場合には、EventRegistration によって評価されます。
  - そのイベントがより具体的なイベント トピックに送信された場合には、EventRegistration の `sink` フラグが考慮されます。

`sink`

登録には *sink* というフラグが付けられることがあります。これは、関心が登録されたすべてのイベントだけでなく、トピック ツリー内でその登録より下位にあるもっと具体的なトピックに関連付けられているすべてののイベントも評価する機会が必ず与えられることを意味します（sink フラグを true に設定してルート トピック（*）を登録すると、WebLogic Server に提出されるすべてのイベントを評価する機会が保証されます。このような登録の evaluate メソッドが true を返すだけであれば、WebLogic Server に提出されるすべてのイベントに基づいて効率的に動作します)。

登録の sink フラグを `false`（デフォルト）に設定した場合、クライアントは、正確に一致するイベントが発生したときにのみ通知を受け取り、

その分岐より下位のもっと具体的なイベントが発生したときには通知を受け取りません。ただし、この規則には以下のような例外があります。

`sink` フラグが `false` に設定されていても、そのイベントがより具体的なイベント トピックに正しく送信されなかった場合には、EventRegistration はより具体的なイベントを評価します。こうしたことは、その時点であるより具体的なイベント トピックに関心を登録しているクライアントがない場合に起こります。トピックは、クライアントがそのトピックへの関心を登録して初めてツリー内に存在するので、イベントを評価する際には注意しなければなりません。`sink` を `false` に設定しても、クライアントがそのトピックに関連するイベントだけを受け取るとはかぎりません。

イベントはこのようにして評価されるので、受信登録したクライアントがないイベントを捕捉する EventRegistration をセットアップすることができます。

EventRegistration によるイベントの評価方法

クライアントは、EventRegistration を通じてイベントへの関心を登録するときには、その EventRegistration に関連付けられる Evaluate オブジェクトを指定しなければなりません。一致する登録にイベントが到達すると、WebLogic Server はその Evaluate オブジェクトの `evaluate()` メソッドを呼び出します。Evaluate クラスはインタフェース `weblogic.event.evaluators.EvaluateDef` を実装するもので、必ずこのメソッドを実装します。通常はユーザが作成したクラスか、または Weblogic のデフォルト エバリュエータの 1 つです。Evaluate クラスは、サーバ上にインストールされなければならず、またサーバの `CLASSPATH` 内に含まれていなければなりません。

`evaluate()` メソッドには、イベントに付随するパラメータが渡されます。カスタム メソッドでは、これらのイベント パラメータを解析して、true か false のいずれかを返すことができます。true の場合には、`phase` が false に設定されていないかぎり、WebLogic Server は Action オブジェクトの `action()` メソッドを呼び出します。

`phase`

クライアントがトピックへの関心を登録する場合、`phase` を設定することがあります。これは、Action を呼び出すロジックを反転させるものです。たとえば、あるアプリケーションが、いつサンフランシスコの天気が晴れになるかに関心がある場合には、登録は以下のようになります。

- トピックは *weather.northamerica.california.sf* です。
- Evaluate パラメータは、`SKYINDICATOR="fogginess"`、`INDICATORLEVEL="over"`、および `INDICATORVALUE="40%"` です。
- phase は false です。

  これでこのロジックが反転されるので、このクライアントには、サンフランシスコで霧が出ていないときに通知が行われます。クライアントではこれが晴れを意味すると見込んでいます。

アクション プロセスの動作の仕組み

評価プロセスが成功すると、その登録のアクション クラスが呼び出されます。

アクション クラスはユーザが作成するクラスで、インタフェース `weblogic.event.actions.ActionDef` を実装しています。アクション クラスは、Java で記述可能なアクションのすべてを実行できます。アクション クラスの例は `ActionEmail`、`ActionUDP`、および `ActionNull` で、これらは `weblogic.event.actions` パッケージに入っています。

Action クラスは、関心の登録を発行した WebLogic クライアントに、エバリュエータが true を返したことを通知できます。クライアントサイドの通知の例については、以下を参照してください。

パラメータの詳細

パラメータには以下のものがあり、WebLogic Server 内のオブジェクトによって使われます。

- 登録管理。管理パラメータは登録管理に使われるもので、エバリュエータが呼び出される回数の制限やその他の管理上の詳細を設定します。
- 関心の登録。登録パラメータは名前 = 値の組の集合で、それらが全体としてイベントへの関心のスコープを定義します。これらのパラメータを使って、WebLogic Server はイベントをフィルタ処理して、イベントを評価すべきかどうかをさらに限定できます。登録は、常に一群の登録パラメータ（それ自体がその登録の管理パラメータのサブセット）を付けて、WebLogic Server に提出されます。

- イベント パラメータ。イベント パラメータは、登録パラメータと同様に、イベントをさらに限定する一群の名前 = 値の組です。イベントは、常に一群のイベント パラメータを付けて WebLogic Server に提出されます。

パラメータは ParamSet オブジェクトとして作成され、それ自体が ParamSet の配列である場合もあります。ParamSet の各パラメータに関連付けられる値は ParamValue オブジェクトで、それ自体が ParamValue の配列である場合があります。

2 WebLogic Event オブジェクトとそのクラス

WebLogic Event API には、以下のパッケージがあります。

パッケージ weblogic.event.actions
パッケージ weblogic.event.common
パッケージ weblogic.event.evaluators

WebLogic Event には、5 つの基本的なタイプのオブジェクトがあります。

- Evaluate オブジェクトと Action オブジェクト。登録時に WebLogic Server 内に作成されます。これらは、イベントがどのように評価され、それに基づいてどのように動作すべきかについての情報を格納します。これらのオブジェクトは、登録の引数になります。
- EventTopic オブジェクト。EventTopic は、トピック ツリー内の 1 つのノードを表すオブジェクトです。これは、トピックにイベントを提出するメソッドと、トピックへの関心を登録するメソッドを備えています。また、このオブジェクトを使用すると、ユーザはトピックの有効期間を制御できるようになります。
- EventRegistration オブジェクト。登録に関する情報を格納します。登録される情報の ID、登録の期間、および Evaluate オブジェクトと Action オブジェクトに関するクラス情報を含んでおり、登録時に WebLogic Server に提出されます。
- EventMessage オブジェクト。トピック ツリーによるフィルタ処理時にイベントをカプセル化するもので、EventTopic ごとに現在の EventRegistration に基づいて評価され、アクションが行われます。
- パラメータ オブジェクト。イベント、エバリュエータ、およびアクションのスコープに関する詳細を格納します。

Evaluate オブジェクトと Action オブジェクト

weblogic.event.evaluators.* および weblogic.event.actions.* のパッケージには、以下の用途のクラスとインタフェースが入っています。

- ユーザ定義のエバリュエータ クラスとアクション クラスを作成する（EvaluateDef インタフェースと ActionDef インタフェースの実装）
- 登録時に、これらのクラスをインスタンス化するオブジェクトを登録パラメータおよびその他の設定と一緒に WebLogic Server に提出する（Evaluate クラスと Action クラス）

イベントへの関心を登録する場合には、weblogic.common.EventServices.getEventRegistration() メソッドの 2 つの引数として、エバリュエータ オブジェクトとアクション オブジェクトのクラス名も提出しなければなりません。

作成するエバリュエータ クラスは、インタフェース EvaluateDef を実装しなければなりません。また、作成するアクション クラスは、インタフェース ActionDef を実装しなければなりません。なお、両方のインタフェースを実装する単一のクラスを作成することができます。

これらのオブジェクトのコンストラクタは、ユーザが作成したクラスの完全パッケージ名とパラメータ群（ParamSet）を引数に取ります。エバリュエータ クラスとアクション クラスは、登録時に WebLogic Server 内にインスタンス化されます。Java クラス ローダでは、動的にロードされるクラスのコンストラクタに引数を渡すことが許可されないので、これらのクラスのコンストラクタは、デフォルト コンストラクタ、つまり引数を持たないコンストラクタでなければなりません。このため、registerInit() メソッドを使って、新たに作成されたエバリュエータ オブジェクトやアクション オブジェクトに登録パラメータを指定します。これによって、これらのオブジェクトは、登録の際に提出された評価パラメータとアクション パラメータを調べ、それに従って動作することができます。

EvaluateDef インタフェースと ActionDef インタフェース

`weblogic.event.evaluators.EvaluateDef`

`weblogic.event.actions.ActionDef`

これらのパッケージにはそれぞれ、インタフェース `EvaluateDef` と `ActionDef` が入っています。`EvaluateTrue` や `ActionEmail` といった、これらのパッケージに含まれる他のクラスは、EvaluateDef インタフェースと ActionDef インタフェースを実装したものです。独自の Evaluate クラスと Action クラスを作成する場合には、これらを方法のサンプルとして調べてください。

イベントに応じたアクションの評価用のパラメータを設定するには、ParamSet オブジェクトを使用します。これらのパラメータは、関心を持つすべての当事者が知っておかなければなりません。イベント、登録、エバリュエータ、およびアクションのパラメータ間には一定の関係はありませんが、アプリケーションによっては、開発者が関係を構築することもできます。

以下の例は、パラメータ間の関係を構築する方法をわかりやすく示しています。この例では、`evaluate()` メソッドのパラメータは、提出されたイベントのパラメータに一致しなければならず、また、`action()` メソッドのパラメータは、登録のパラメータに一致しなければなりません。天気の関心のあるトピックはサンフランシスコで、`evaluate()` メソッドが true を返すには、評価とイベントのパラメータが一致しなければなりません。同様に、この例では、霧の程度を示すファクターが一定の最低値になったときに取るべきアクションは、電子メールを送信することです。したがって、登録パラメータは、電子メールを送信するアクション クラスに必要な情報をすべて提供しなければなりません。この特定なイベント、登録、評価、アクションに適した ParamSet を設定します。

EvaluateDef インタフェースと ActionDef インタフェースは、evaluate() メソッドと action() メソッドの両方を含む単一のクラスによって実装することができます。単一のクラスを使用すると、両方のメソッドが同じ変数にアクセスできるという利点があります。

実装対象のメソッド

```
public boolean evaluate(EventMessageDef eventMsg)
```

```
                          throws ParamSetException;

public void action(EventMessageDef eventMsg);
```

これらのメソッドのそれぞれは、EventMessageDef インタフェースを実装する Object を渡されます。Object をインタフェースで参照すると、このオブジェクトの実装上の詳細に注意を払う必要がなくなります（基底のオブジェクトは、クライアントサイドまたはサーバサイドでの実装になる可能性があります)。EventMessageDef オブジェクトには、イベントとイベント パラメータに関する情報が入っています。このインタフェースに定義されたメソッドを通じて、これらにアクセスすることができます。

```
registerInit()
```

Java クラス ローダでは、動的にロードされるクラスのコンストラクタに引数を渡すことが許可されないので、インタフェースを実装するあらゆるユーザ作成クラスのコンストラクタは、デフォルト コンストラクタ、つまり引数を取らないコンストラクタでなければなりません。このため、registerInit() メソッドを使って、新たに作成されたエバリュエータ オブジェクトやアクション オブジェクトに登録パラメータ（ParamSet オブジェクト）を指定します。これによって、これらのオブジェクトは登録パラメータを調べ、それに従って動作することができます。

```
isLongRunning()
```

このメソッドは、バージョン 2.5 より非推奨になりました。エバリュエータとアクションのインタフェースを実装するユーザは、このメソッドを指定する必要がなくなりました。現在、evaluate メソッドと action メソッドは、WebLogic Server 内のスレッド プールから選択された別個のスレッド内でデフォルトで動作するので、操作はより高速かつ効率的に実行されます。

EventTopic オブジェクト

`weblogic.event.common.EventServicesDef`

`weblogic.event.common.EventTopicDef`

リリース 3.0 現在、WebLogic Event は、イベント メッセージを送受信したいアプリケーションで使用するためのもっとも重要なオブジェクトとして、EventTopic をサポートしています。これは、イベントベースの簡単なプログラミング手法を提供します。EventTopic オブジェクトを使用すると、WebLogic クライアント アプリケーションは、下位トピックの取得、EventMessage の送信、またはイベントへの関心の登録を行うことができます。

`EventServicesDef.getEventTopic()` メソッドを呼び出すことによって、EventServices ファクトリに EventTopic を要求します。下位トピックを作成するには、`EventTopicDef.getEventTopic()` メソッドを使用します。次に例を示します。

```
EventTopicDef topic =
    t3services.events().getEventTopic("WEATHER.CA.SF");
```

t3services は、JNDI ルックアップで取得したリモート インタフェースです。

また、EventTopic の有効期間を制御することもできますが、それには、`EventTopicDef.getEventTopic()` メソッドの呼び出しで、`EventTopicDef.EPHEMERAL` または `EventTopicDef.DURABLE` に設定します。EventServices ファクトリに EventTopic「ルート」を要求し、`DURABLE` な下位トピックを作成することで、トピック ツリーのサイズと形状をより詳細に管理できます。次に例を示します。

```
EventTopicDef topic = t3services.events().
                      getEventTopic("WEATHER.CA.SF",
                                    EventTopicDef.DURABLE);
```

EventTopic オブジェクトを使用して、トピック ツリー内の下位トピックを取得または作成することができます。下位トピックは、トピック ツリー内の単一のノード以上のものを表すことができます。以下に示すように、EventTopic 自身に対して `getEventTopic()` メソッドを呼び出すだけでかまいません。

```
  EventTopicDef topic =
    t3services.events().getEventTopic("WEATHER");
  EventTopicDef weatherCA   = topic.getEventTopic("CA");
```

```
EventTopicDef weatherCASF = topic.getEventTopic("SF");
EventTopicDef weatherNYNY = topic.getEventTopic("NY.NY");
```

EventTopic を作成したら、そのトピックに EventMessage または EventRegistration を提出することができます。詳細については、以下の「WebLogic Event を使った実装」で説明します。以下に短い例を 2 つ示します。最初の例は、天気イベントへの関心を登録するものです。

```
EventTopicDef topic =
  t3services.events().getEventTopic("WEATHER.CA.SF");
Evaluate eval =
  new Evaluate("weblogic.event.evaluators.EvaluateTrue");
Action action = new Action(this);
EventRegistrationDef er = topic.register(eval, action);
```

2 番目の例は、トピック ツリー内の同じトピックの EventMessage を提出するものです。

```
EventTopicDef topic =
  t3services.events().getEventTopic("WEATHER.CA.SF");
ParamSet ps = new ParamSet();
ps.setParam("TEMPERATURE", 23);
topic.submit(ps);
```

また、EventTopic にアクセス制御リスト（ACL）を関連付けて、どのユーザがイベントを提出したり受け取ったりできるかを制御することもできます。ACL の詳細については、「WebLogic レルム内での WebLogic Event 用 ACL のセットアップ」を参照してください。

EventRegistration オブジェクト

`weblogic.event.common.EventServicesDef`

`weblogic.event.common.EventRegistrationDef`

クライアントがイベントへの関心を登録しておくと、そのイベントが発生したときにクライアントに通知されます。イベントを評価し、それに従って動作できるためには、イベントへの関心を登録しなければなりません。

`EventTopic.register()` メソッドを使用して（Evaluate オブジェクトと Action オブジェクトを引数として渡す)、EventRegistration を取得することができます。これは、イベントへの関心を登録する一番簡単な方法です。

また、メソッド `getEventRegistration()` を使用して、EventServices ファクトリから EventRegistration オブジェクトへのインタフェースを取得することができます。次いで、以下のようにして、イベントへの関心を登録します。

```
EventRegistrationDef erDef=
  t3services.events().
    getEventRegistration(String topicName,
                         Evaluate evaluator,
                         Action action,
                         boolean sink,
                         boolean phase,
                         int count);
```

t3services は、JNDI ルックアップで取得したリモート サービス ファクトリであり、

また、上記のパラメータは以下のとおりです。

`EventRegistrationDef` *erDef*

このメソッドは、`EventRegistrationDef` インタフェース オブジェクトを返します。ここでも、このインタフェースは、サーバ上に存在する可能性のある実際の EventRegistration オブジェクト内のすべてのメソッドへのアクセスをクライアントに提供します。

`String` *topicName*

topicName は、関心のある EventTopic を、解析可能なドット表記フォーマットの文字列（たとえば、「weather.northamerica.us.california」）として指定します。また、トピックは、文字列の配列として指定することもできます。この場合、配列内の各要素は下位トピック（たとえば、「weather」、「northamerica」、「us」、「california」）に対応します。各トピックは、新しい登録が受信されたときに、WebLogic Server 内のトピック ツリーに動的に追加されます。もちろん、イベントへの関心を登録しようとするアプリケーションは、アプリケーションが提出するイベントのトピックを知っている、またはその逆のことが必要です。

`Evaluate` *evaluator*

ユーザが作成したエバリュエータ クラスをインスタンス化して WebLogic Server 上で実行するのに使われる `Evaluate` オブジェクト。Evaluate オブジェクトを作成するときは、EvaluateDef クラスの完全パッケージ名と、関心のあるトピックを限定する評価パラメータ群（ParamSet）を指定します。

`Action` *action*

ユーザが作成したアクション クラスをインスタンス化するのに使われる Action オブジェクトで、イベントの評価結果が true の場合に呼び出されるもの。以下のいずれかを指定することによって、Action オブジェクトを作成します。

- サーバ上でインスタンス化され実行される ActionDef クラスの完全パッケージ名
- クライアント上でローカルに呼び出される ActionDef オブジェクト自体のローカル インスタンス

また、アクションがどのように実行されるかを限定するパラメータ群（ParamSet）も指定します。

`boolean` *sink*

`sink` が true の場合、登録は、関心を登録したすべてのイベントの通知と、トピック ツリー内で登録されたトピックより下位にあるすべてのイベントの通知を受け取ります。たとえば、トピック *weather.northamerica.us.california* の登録について、*sink* を true に設定すると、この登録が *weather.northamerica.us.california* だけでなく、*weather.northamerica.us.california.la* や *weather.northamerica.us.california.sf* のトピックのイベントも評価することになります。sink のデフォルト値は true です。

`sink` が `false` の場合、イベント メッセージが正しく送信されないときでも、登録は、より具体的なトピックに向けられた任意のイベント メッセージを受け取ります。

`boolean` *phase*

phase が `false` に設定された場合、評価のロジックが反転します。デフォルト値は `true` です。たとえば、「fogginess」パラメータが一定の値を超えたと報告されたときに weather トピックのエバリュエータが true を返す場合には、phase を `false` に設定して、同じエバリュエータを使用すると、「fogginess」パラメータが一定の値以下になった場合に true を返すようにすることができます。

`int` *count*

count は、登録がイベントを評価できる回数を指定します。その回数だけ評価した後は、登録は自動的にキャンセルされます。回数が未設定の場合、デフォルトは、`EventRegistrationDef.UNCOUNTED` です。リ

リース 3.0 で追加された別のオプションは EventRegistrationDef.ON_DISCONNECT で、クライアントが接続解除するとイベント登録を自動的にキャンセルするものです。

EventRegistrationDef オブジェクトへのインタフェースを取得したら、その register() メソッドを使って、それを WebLogic Server に登録しなければなりません。これは、register() メソッドの成否にかかわらず、インスタンス化時の一意な識別番号を返します。register() メソッドが成功すると、EventRegistrationDef.isRegistered 変数は true に設定されます。

EventRegistration クラスには、(getEvaluator() のように)、EventRegistration オブジェクトが要求されたときに提供された引数を返すアクセサがあります。

EventRegistrationDef オブジェクトに対して unregister() メソッドを呼び出すことによって、登録を解除することができます。EventRegistration オブジェクトにアクセスできない場合には、EventServicesDef インタフェースの unregister() メソッドを以下のようにして使用できます。

```
t3client.event.services().unregister(int regID);
```

ここで、t3client は T3Client オブジェクトで、regId は EventRegistrationDef オブジェクトが登録されたときに返される一意な識別子です。

登録が成功すると、アクション パラメータと評価パラメータに使える内部パラメータがあります。それらは以下のとおりです（パッケージによって異なります)。

- EVENT_SERVER_REGISTRATION_TIME
- EVENT_SERVER_REGISTRATION_THREAD
- EVENT_CLIENT_REGISTER_TIME
- EVENT_CLIENT_REGISTER_THREAD
- EVENT_CLIENT_REGISTER_HOST

EventMessage オブジェクト

イベントは、EventMessage オブジェクトとして、WebLogic Server に提出されます。EventMessage を提出するもっとも簡単な方法は、EventServicesDef.getEventTopic() メソッドを使って、EventServices ファク

トリに EventTopic を要求することです。次いで、ParamSet を作成し、それを引数として渡して `EventTopic.submit()` メソッドを呼び出すことによって、EventMessage を提出します。

`EventServicesDef.getEventMessage()` メソッドを使って、（オブジェクトを作成する代わりに）EventServices ファクトリに EventMessage オブジェクトを要求することもできます。EventMessage は、インタフェース `EventMessageDef` を実装します。

どのようなアプリケーションでも WebLogic Server にイベントを提出できますが、ここでの説明は、Java オブジェクトを使える Java アプリケーションに限定します。

`getEventMessage()` ファクトリ メソッドは、2 つの引数を取ります。トピックと、イベントを限定するパラメータ群（ParamSet）です。WebLogic Server にイベントを提出するには、EventServices ファクトリにイベントを要求した後、そのオブジェクトに対して `submit()` メソッドを呼び出します。このクラス内の他のメソッドを使うと、イベント パラメータにアクセスしたり、イベントに関する詳細を表示したりすることができるようになります。EventMessage オブジェクトは、WebLogic Server によって、`evaluate()` メソッドに渡されます。これによって、エバリュエータが比較のためにイベント パラメータにアクセスできるようになります。

ParamSet オブジェクトと ParamValue オブジェクト

イベント、登録、評価およびアクションはすべて、パラメータを使ってスコープを限定します。パラメータは、WebLogic Event では weblogic.common.ParamSet オブジェクトによって扱われます。このオブジェクトには、weblogic.common.ParamValues が入っています。WebLogic は、ParamSet と ParamValue を使って、クライアントとサーバの間でデータを受け渡します。

ParamSet パラメータは、SKYINDICATOR="fogginess" のように、名前 = 値の組です。パラメータの名前はそのキー名で、ParamSet のすべての内容はキー名でアクセスできます。ParamSet 内のキー名ごとに、対応する ParamValue を設定します（ParamType 内の mode、desc、type、name 用の変数は、イベントには使われません)。

名前 = 値の組だけの ParamSet を作成することは単純な操作ですが、必要であれば ParamSet と ParamValue の間に複雑な関係を設定できるほど強力です。たとえば、以下の例では、名前 = 値の組を 3 つ作成して、サンフランシスコの天気への関心を登録するための評価基準を設定する方法を示しています。

```
ParamSet evalRegParams = new ParamSet();
evalRegParams.setParam("SKYINDICATOR",      "fogginess");
evalRegParams.setParam("INDICATORLEVEL",    "over");
evalRegParams.setParam("INDICATORVALUE",    "40");
```

これらのパラメータは、Evaluate クラスのコンストラクタとして使われ、このクラス自体は EventRegistration の引数として使われます。たとえば、サンフランシスコの天気の状態に関して WebLogic Server にイベントを提出するときも、以下のように同様のパラメータを設定します。

```
ParamSet eventParams = new ParamSet();
eventParams.setParam("SKYINDICATOR",        "fogginess");
eventParams.setParam("INDICATORLEVEL",      "equals");
eventParams.setParam("INDICATORVALUE",      "35");
```

イベント パラメータは、getEventMessage() メソッドの引数として使われます。イベントが発生すると、イベント サーバは、そのイベントを Evaluate メソッドに渡し、それによって、イベント パラメータは Evaluate クラスから利用できるようになります。登録を取り消すには、weblogic.event.evaluators.EvaluateDef.registerInit() メソッドを使います。

```
public void registerInit(ParamSet params) {
  weatherSymbol = params.getValue("SKYINDICATOR").asString();
  weatherLevel  = params.getValue("INDICATORVALUE").asInt();
}
```

次いで、以下のようにイベント パラメータと登録パラメータを比較することができます。

```
public boolean evaluate(EventMessage ev) {
  ParamSet eventParams = ev.getParameters();
  if (eventParams.getValue("SKYINDICATOR").asString()
              .equalsIgnoreCase(weatherSymbol))
  {
    int eventLevel =
```

```
        eventParams.getValue("INDICATORVALUE").asInt();
    if (eventLevel == weatherLevel)
    return true;
  }
  return false;
}
```

ParamSet の設定と取得の方法を示すこの簡単な例を見れば、イベント登録、イベント提出、および評価プロセスがどのように相互作用するかについての基本的な概要も把握できます。

ParamSet の効率的な使い方

ParamSet とそれによって限定されるオブジェクトを使う際に効率に関していくつかの考慮事項があります。特定のトピックについてのイベントを提出するたびに、新しい EventMessage とそれに関連付けられる ParamSet を作成することは、必要でもなければ、望ましくもありません。

このコード例では、提出のたびに新しい ParamSet と EventMessage を作成していますが、ここでは、ParamSet が 100 個、ParamValue が約 300 個 (EventMessage.submit() によって 2 個の ParamValue が自動的に追加されるので)、そしてイベントが 100 個それぞれ生成され、ParamSet 内の ParamValue のルックアップが 100 回行われます。

```
  for (int i = 0; i < 100; i++) {
    ps = new ParamSet();
    EventMessageDef em = t3.services.events()
                           .getEventMessage(topic, ps);
    ps.setParam("number", i);
    em.submit();
  }
```

ParamSet と EventMessage をクラス内のインスタンス変数として作成した後、それらを修正し、必要に応じて提出し直す方が効率的です。以下の例では、ParamSet を 1 つ、ParamValue を 3 つ、Event を 1 つそれぞれ生成し、ParamSet 内の ParamValue のルックアップを 100 回行います。

```
  ps = new ParamSet();
  EventMessageDef em = t3.services.events()
                         .getEventMessage(topic, ps);

  for (int i = 0; i < 100; i++) {
    ps.setParam("number", i);
    String status = em.submit();
  }
```

もっとも効率的な方法は、基底の ParamValue への参照を作成し、それを繰り返し設定することです。以下の例では、この方法をどのように使えば、ParamValue を何度もルックアップしてカウンター「number」を取得しなくても済むかを示しています。

```
ps = new ParamSet();
ParamValue num = ps.getParam("number");
EventMessageDef em = t3.services.events()
                        .getEventMessage(topic, ps);

for (int i = 0; i < 100; i++) {
  num.set(i);
  String status = em.submit();
}
```

この最後のコード例では、ParamSet を 1 つ、ParamValue を 3 つ、Event を 1 つそれぞれ生成し、ParamSet 内の ParamValue のルックアップを 1 回行います。

Event と ParamSet は、逐次再利用可能ですが、スレッドセーフではありません。つまり、再利用できますが、複数のスレッドで同時に使うことはできません。同じコード断片をマルチスレッド セーフにするには、たとえば、以下のように Event の提出を synchronized ブロック内にラッピングします。

```
ps = new ParamSet();
ParamValue num = ps.getParam("number");
EventMessageDef em = t3.services.events()
                        .getEventMessage(topic, ps);

for (int i = 0; i < 100; i++) {
  synchronized (em) {
    num.set(i);
    em.submit();
  }
}
```

新しい ParamSet を作成してから新しい EventMessage を要求しなければなりませんが（ParamSet オブジェクトは `getEventMessage()` メソッドで使われるので)、`Event.submit()` メソッド（または Evaluate と Action のコンストラクタの場合には `register()` メソッド）が呼び出される直前まで、`ParamSet.setValue()` メソッドを呼び出す必要はありません。ParamSet が実際に調べられるのは、`submit()` または `register()` が呼び出されるときだけです。

WebLogic Event を使った実装

WebLogic Event の主な実装は 2 通りあります。1 つは、イベントへの関心を登録できる WebLogic Event アプリケーションを構築することで、これには、`evaluate()` および `action()` メソッドの作成と ParamSet の構築が必要になります。もう 1 つは、他のアプリケーションにイベント生成を組み込むことです。これらの例では、以下の 4 つのクラスを使ってこのプロセスを説明します。

1. イベントを評価するクラス
2. 適切なイベントに従って動作するクラス
3. イベントに関心を登録するクラス
4. WebLogic Server にイベントを送るクラス

以下の例では、アプリケーションを使うと、コマンド ラインから株式への関心を登録し、購入希望価格を設定できるようになります。次いで、株式を入札に出しているイベント サーバに一連のイベントを送ることができます。買い注文と一致する付け値が WebLogic Server 内で評価されると、アクション（つまり、電子メールによる通知を送ること）が呼び出されます。

`evaluate()` メソッドと `action()` メソッドを両方とも備えている単一のクラスを使って、EvaluateDef インタフェースと ActionDef インタフェースを実装できます。

- Evaluate クラスの作成
  - 手順 1. パッケージのインポート
  - 手順 2. registerInit() メソッド
  - 手順 3. evaluate() メソッド
  - EvaluateStocks（エバリュエータ）クラスのコード
- Action クラスの作成
  - 手順 1. パッケージのインポート
  - 手順 2. registerInit() メソッド
  - 手順 3. action() メソッド
  - MailStockInfo（アクション）クラスのコード

- イベントへの関心の登録
  - 手順 1. パッケージのインポート
  - 手順 2. コマンドライン引数のチェック
  - 手順 3. コマンドライン引数の処理
  - 手順 4. EventServices ファクトリの取得
  - 手順 5. 登録の作成と提出
- WebLogic Server へのイベント送信
  - 手順 1. パッケージのインポート
  - 手順 2. コマンドライン引数のチェック
  - 手順 3. コマンドライン引数の処理
  - 手順 4. イベントの提出
  - SendEvents クラスのコード

株式の例の次は、クライアントサイド通知を示す例です。クライアントサイド通知を使用すると、Action メソッドを WebLogic Server ではなく T3Client 上で実行できるようになります。

- クライアントサイド通知の使い方

Evaluate クラスの作成

サンプル アプリケーションでは、イベント（つまり、誰かが一定の株式を特定の価格で売却するイベントを提出する）を、特定価格での一定株式の購入に対する関心の登録に照らし合わせて評価します。ここで作成する Evaluate クラスは、インタフェース `weblogic.event.evaluators.EvaluateDef` を実装します。

手順 1. パッケージのインポート

すべての WebLogic Event クラス用の以下のパッケージをインポートします。

- `weblogic.common.*;` ParamSet 、ParamValue へのアクセス用

- `weblogic.event.common.*;` 共通の WebLogic Event オブジェクトへのアクセス用

Evaluate クラスについては、このクラスが実装するインタフェースである `weblogic.event.evaluators.EvaluateDef` もインポートします。

このクラスでは、アプリケーションが EventServices オブジェクト ファクトリにアクセスするのに使う WebLogic Server サービスを定義するクラス変数「services」も作成します。`setServices()` メソッドは、実行時にエバリュエータが実行されると呼び出されます。

手順 2. registerInit() メソッド

動的にロードされるクラス（Evaluate クラスと Action クラスは、どちらも登録時に WebLogic Server に動的にロードされます）は、コンストラクタに引数を渡せないので、`registerInit()` メソッドを使って、新たに作成された Evaluate オブジェクトに登録パラメータを渡します。WebLogic Server は、登録処理時に Evaluate クラス用に作成された ParamSet *params* を Evaluate クラスに渡します。

この場合には、関心の登録に伴う「SYMBOL」パラメータと「TRIGGERVALUE」パラメータに注目します。`evaluate()` メソッドでは、これらのパラメータと提出されたイベントのパラメータを比較します。

```
public void registerInit(ParamSet params)
     throws ParamSetException
{
  regSymbol       = params.getValue("SYMBOL").asString();
  regTriggerValue = params.getValue("TRIGGERVALUE").asInt();
  System.out.println("Symbol/Trigger Value = " +
     regSymbol + "/" +
     regTriggerValue);
}
```

見つかった登録パラメータを確認するには、標準出力に 1 行出力します。

手順 3. evaluate() メソッド

evaluate() メソッドは、簡単に言えば、イベントへの関心の登録によって設定されたパラメータと、イベント自体のパラメータを比較します。これが true を返す場合には、WebLogic Server は、action() メソッドを呼び出し、そのイベントに対してアクションを実行します。

この例では、関心のある株式 SYMBOL と、イベントとして渡された株式 SYMBOL を比較します。イベントの SYMBOL と、この登録が関心を持っているものが一致した場合、イベントによって提出された BID のチェックを行い、それが、関心のあるものとして登録された TRIGGERVALUE と一致するかどうかを調べます。

```
public boolean evaluate(EventMessageDef ev)
     throws ParamSetException
{
  // イベント パラメータを取得する
  ParamSet eventParams = ev.getParameters();

  // イベントの「SYMBOL」パラメータの値と登録時に
  // 「SYMBOL」に設定された値を比較する
  if (eventParams.getValue("SYMBOL").asString()
                   .equalsIgnoreCase(regSymbol)) {

    int eventValue = eventParams.getValue("BID").asInt();

    // 次いで、そのイベント値と、登録時に設定された
    // トリガ値が等しいかどうかを判定する
    if (eventValue == regTriggerValue)
return true;
  }
  return false;
}
```

これで Evaluate クラスは完成です。完全なコード例は以下のとおりです。

EvaluateStocks（エバリュエータ）クラスのコード

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.common.*;
import weblogic.event.evaluators.EvaluateDef;
```

```
public class EvaluateStocks implements EvaluateDef {

  String  regSymbol;
  int     regTriggerValue;
  private boolean verbose = false;

  T3ServicesDef services=null;

  // サービス オブジェクトを保存する
  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  // イベントを評価するのに使う
  // 登録パラメータを取得する
  public void registerInit(ParamSet params)
       throws ParamSetException
  {
    regSymbol       = params.getValue("SYMBOL").asString();
    regTriggerValue = params.getValue("TRIGGERVALUE").asInt();
    System.out.println("Symbol/Trigger Value = " +
       regSymbol + "/" +
       regTriggerValue);
  }

  public boolean evaluate(EventMessageDef ev)
       throws ParamSetException
  {
    // イベント パラメータを取得する
    ParamSet eventParams = ev.getParameters();

    // イベントの「SYMBOL」パラメータの値と登録時に
    // 「SYMBOL」に設定された値を比較する
    if (eventParams.getValue("SYMBOL").asString()
                       .equalsIgnoreCase(regSymbol)) {

      int eventValue = eventParams.getValue("BID").asInt();

      // 次いで、そのイベント値と、登録時に設定された
      // トリガ値が等しいかどうかを判定する。
      if (eventValue == regTriggerValue)
      return true;
    }
    return false;
  }
}
```

Action クラスの作成

evaluate() メソッドが true を返した場合に実行するアクションは、イベントへの関心を登録する際に指定したアドレスに電子メールを送信することです。Action クラスは、インタフェース weblogic.event.actions.ActionDef を実装します。

手順 1. パッケージのインポート

weblogic.common.* と weblogic.event.common.* に加えて、実装対象のインタフェース weblogic.event.actions.ActionDef をインポートします。

このクラスでは、アプリケーションが EventServices オブジェクト ファクトリにアクセスするのに使う WebLogic Server サービスを定義するクラス変数「services」も作成します。setServices() メソッドは、アクションが実行されると呼び出されます。

手順 2. registerInit() メソッド

Evaluate クラスと同様、Action クラスは、WebLogic Server 内に動的にロードされるので、コンストラクタに引数を渡してオブジェクトを作成することはできません。したがって、registerInit() メソッドを使って、新たに作成された Action オブジェクトに Action 登録パラメータを渡します。WebLogic Server は、このメソッドを使って、登録 ParamSet *params* を Action クラスに渡します。このメソッドでは、次の手順で作成する action() メソッドに関係のあるパラメータにアクセスできます。

この例では、イベントへの関心を登録した人に電子メールを送信する方法に関する情報に注目します。action() メソッドで電子メールを送るのに必要なパラメータ、つまり送信先と SMTP ホスト名だけを取得します。これらのパラメータはどちらも、関心の登録に必要だったものです。

```
public void registerInit(ParamSet params) {
  smtphost = params.getValue("SMTPhost").toString();
  to       = params.getValue("Addressee").toString();
}
```

手順 3. action() メソッド

この例のクラスでは、エバリュエータが true を返した場合に実行するアクションは、株式の購入に対する関心を登録した人に、登録された株式が関心を持っていた価格で売りに出されたことを通知することです。イベント パラメータにアクセスすることができ、それらを電子メール メッセージに含めることができます。この例ではさらに、アクションが実行されていることを WebLogic Server 内の標準出力に 1 行出力し、その中に送信先と関心のある付け値も含めます。

ここでは、sendMail() メソッドを使いますが、これは、SMTP ホスト名、送信元の電子メール アドレス、メッセージの宛先の電子メール アドレス、件名、およびメッセージの本文という 5 つの引数を取ります。イベント自体に対して dump() メソッドを呼び出し、電子メールに記入するために関心のあるイベントの表示を作成します。

```
public void action(EventMessageDef ev) {
  try {
    ParamSet eventParams = ev.getParameters();
    int eventValue = eventParams.getValue("BID").asInt();

    System.out.println("*** Mailing stock event to " + to +
                       " at price: " + eventValue);
    Utilities.sendMail(smtphost,
                       "events@weblogic.com",
                       to,
                       "Stock Event triggered!",
                       ev.dump());
  }
  catch (ParamSetException e) {
    System.out.println("No BID price in ParamSet");
  }
  catch (java.io.IOException ioe) {
    System.out.println("Failed to connect: [" + ioe + "]");
  }
}
```

最後に、try ブロックが失敗した場合には、ParamSetException を調べます。さらに、電子メールの送信に問題がある場合には、IO 例外を取得します。

これで Action クラスは完成です。完全なコード例は以下のとおりです。

MailStockInfo（アクション）クラスのコード

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.actions.ActionDef;
import weblogic.event.common.*;

public class MailStockInfo implements ActionDef {

  String smtphost = "";
  String to       = "";

  T3ServicesDef services = null;

  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  public void registerInit(ParamSet params) {
    smtphost = params.getValue("SMTPhost").toString();
    to       = params.getValue("Addressee").toString();
  }

  public void action(EventMessageDef ev) {
    try {
      ParamSet eventParams = ev.getParameters();
      int eventValue = eventParams.getValue("BID").asInt();

      System.out.println("*** Mailing stock event to " + to +
                         " at price: " + eventValue);
      Utilities.sendMail(smtphost,
                         "errors@weblogic.com",
                         to,
                         "Stock Event triggered!",
                         ev.dump());
    }
    catch (ParamSetException e) {
      System.out.println("No BID price in ParamSet");
    }
    catch (java.io.IOException ioe) {
      System.out.println("Failed to connect: [" + ioe + "]");
    }
  }
}
```

イベントへの関心の登録

関心の登録を行うためのクラスは、登録パラメータ群を作成するのに使う引数をコマンド ラインから受け取ります。次いで、これらのパラメータを使って、EventRegistration オブジェクトを作成すると同時に、これまでに作成した Evaluate クラスと Action クラスをインスタンス化する Evaluate オブジェクトと Action オブジェクトも作成します。最後に、登録を提出します。

手順 1. パッケージのインポート

すべての WebLogic Event アプリケーション用にインポートされるパッケージ weblogic.common.* と weblogic.event.common.* の他に、登録クラス用の以下のパッケージもインポートします。

- weblogic.event.actions.* この登録のコンストラクタとして使われる Action オブジェクト用
- weblogic.event.evaluators.* この登録のコンストラクタとして使われる Evaluate オブジェクト用

手順 2. コマンドライン引数のチェック

単一のコマンドラインを通じてこの登録を WebLogic Server に渡し、後で使うための引数を取得します。最初の手順は、正しい数の引数があるかどうかをチェックし、そうでない場合には使い方に関する情報を出力することです。

```
if (argv.length !=5> {
  System.out.println("Usage: "
           + "java tutorial.event.stocks.Register "
           + "WebLogicURL STOCKSYMBOL PRICE SMTPHOST EMAIL");
  System.out.println("Example: "
           + "java tutorial.event.stocks.Register "
           + "t3://localhost:7001 SUNW 75 "
           + "smtp.foo.com demos@foo.com");
  return;
}
```

手順 3. コマンドライン引数の処理

最初のコマンドライン引数（WebLogic Server の URL）を使って、T3Client クライアントを作成し、接続します。

```
T3Client t3 = null;
try {
  t3 = new T3Client(argv[0]);
  t3.connect();
```

2 番目と 3 番目のコマンドライン引数を使って、Evaluate クラスに登録パラメータを提供するのに使う ParamSet オブジェクトを作成します。これらのパラメータは、WebLogic Server に提出されるイベントの類似パラメータと比較されます。

```
ParamSet evRegParams = new ParamSet();
evRegParams.setParam("SYMBOL",          argv[1]);
evRegParams.setParam("TRIGGERVALUE",    argv[2]);
```

最後に、最後の 2 つのコマンドライン引数を使って、Action クラスに登録パラメータを提供するのに使う 2 つ目の ParamSet オブジェクトを作成します。この場合には、電子メールの送信に関する情報を提供します。

```
ParamSet acRegParams = new ParamSet();
acRegParams.setParam("SMTPhost",  argv[3]);
acRegParams.setParam("Addressee", argv[4]);
```

手順 4. EventServices ファクトリの取得

すべてのイベント登録は、`EventServicesDef` インタフェース（別名 WebLogic EventServices ファクトリ）を通じて達成されます。EventServices ファクトリへのリモート インタフェースは、`T3ServicesDef` インタフェース（別名 WebLogic T3Services ファクトリ）を通じて取得します。以下のようなコードを使って、WebLogic JNDI ツリー内の T3Services ファクトリをルックアップします。

```
  T3ServicesDef t3services;
  Hashtable env = new Hashtable();
  env.put(Context.PROVIDER_URL, weblogic_url);
  env.put(Context.INITIAL_CONTEXT_FACTORY,
          weblogic.jndi.WLInitialContextFactory.class.getName());
  Context ctx = new InitialContext(env);
  t3services = (T3ServicesDef)
ctx.lookup("weblogic.common.T3Services");
```

```
ctx.close();
```

ここで、`weblogic_url` は、WebLogic Server の URL です。EventServices ファクトリには、以下のように、`T3Services` インタフェースを通じてアクセスします。

```
EventServicesDef eventServices = t3services.event();
```

アプリケーションでは、`EventServicesDef` API を使って、WebLogic Server 上のイベント機能を利用します。

手順 5. 登録の作成と提出

```
weblogic.event.common.EventTopicDef
```

```
weblogic.event.common.EventRegistrationDef
```

```
weblogic.event.actions.ActionDef
```

```
weblogic.event.evaluators.EvaluateDef
```

登録するには、以下に示しようにして、まず EventServices ファクトリから EventTopic（登録したい関心の対象）を取得します。

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
```

次いで、`EventTopicDef.register()` を呼び出すことで、EventTopic を使って登録します。このメソッドは、以下のような少なくとも 2 つの引数を取ります（`egister()` メソッドのそれ以外の引数については、以下を参照してください）。

- Evaluate オブジェクト
- Action オブジェクト

`register()` メソッドに渡す Evaluate オブジェクトと Action オブジェクトは、それぞれ 2 つの引数、上記で作成したクラスの名前と、このクラスにおいてコマンドライン引数を使って作成した ParamSet を使って作成しなければなりません。

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
Evaluate eval =
  new Evaluate("tutorial.event.stocks.EvaluateStocks",
                evRegParams);
Action action =
  new Action("tutorial.event.stocks.MailStockInfo",
```

```
      acRegParams);
  EventRegistrationDef er = topic.register(eval, action);
```

クラスの名前ではなく、Object を引数として使って、新しい Action オブジェクトを作成することもできます。これによって、クライアントサイド プログラムは、Action のローカル コピーを使うことができるようになります。その結果、Evaluate メソッドが true を返すと Action クラスがクライアント上で実行されることになり、それによってクライアントサイド通知またはコールバックが可能になります。次に例を示します。ただし、この例はこの説明で使用しているクラスとは関係ありません。

```
  EventTopicDef topic =
    t3.services.events().getEventTopic("STOCKS");
  Evaluate eval =
    new Evaluate("tutorial.event.stocks.EvaluateStocks",
                 evRegParams);
  Action action = new Action(this);
  EventRegistrationDef er = topic.register(eval, action);
```

なお、Evaluate コンストラクタの引数として Object を使用することはできません。Evaluate オブジェクトは常にサーバ上で実行されます。

`register()` メソッドには、各登録に必要な Evaluate オブジェクトと Action オブジェクト以外にも、以下の引数を指定することができます。

- トピックが sink かどうかを示す Boolean（デフォルトは false）
- トピックの phase、つまり「true」と「false」のどちらを返す Evaluate メソッドを評価すべきかどうかを示す Boolean（デフォルトは true）。
- count、つまりこの登録が自身を自動的に登録解除するまでに受け取るべきイベントの最大数を示す定数（デフォルトは `EventRegistrationDef.UNCOUNTED`）。リリース 3.0 で新たに加わったのは、オプション `EventRegistrationDef.ON_DISCONNECT` で、これは、関心を登録したクライアントが接続解除されたときに、登録をキャンセルしなければならないことを示しています（Action にクライアントサイド オブジェクトを使っている場合には、この処理は自動的に行われます。これは、Action オブジェクトが WebLogic Server 上にある場合のイベント登録に適用され、クライアントは、関心が完了したときに登録を解除する必要があります）。

以下の例は、イベント登録に対する sink、phase、および count の設定を示します。

```
  EventTopicDef topic =
    t3.services.events().getEventTopic("STOCKS");
  Evaluate eval =
```

```
    new Evaluate("tutorial.event.stocks.EvaluateStocks",
                 evRegParams);
  Action action =
    new Action("tutorial.event.stocks.MailStockInfo",
       acRegParams);
  EventRegistrationDef er =
        topic.register(eval, action, true, false,
                       EventRegistrationDef.ON_DISCONNECT);
```

WebLogic Server にこの登録を提出した後、`finally` ブロック内で接続を切断します。

```
      int regid = er.getID();
      System.out.println("Registration ID is " + regid);
    }
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
```

これで Register クラスは完成です。完全なコード例は以下のとおりです。

Register クラスのコード

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.actions.*;
import weblogic.event.common.*;
import weblogic.event.evaluators.*;

public class Register {

  public static void main(String argv[]) throws Exception {

    // 登録パラメータの設定に使われる 5 つの
    // コマンドライン引数を取得する
    if (argv.length != 5)
    {
      System.out.println("Usage: "
                + "java tutorial.event.stocks.Register "
                + "WebLogicURL STOCKSYMBOL PRICE SMTPHOST EMAIL");
      System.out.println("Example: "
                + "java tutorial.event.stocks.Register "
                + "t3://localhost:7001 SUNW 75 smtp.best.com "
                + "demos@foo.com");
      return;
    }

    // 最初のコマンドライン引数として指定された URL を使って、
    // WebLogic Server に接続する
    T3Client t3 = null;
```

```
    try {
      t3 = new T3Client(argv[0]);
      t3.connect();

      // 各イベントを受信したときに、Action メソッドを呼び出すか
      // どうかを決めるために Evaluate メソッドによって使われる
      // ParamSet を作成する。
      // 2 番目と 3 番目のコマンドライン引数を値とみなす
      ParamSet evRegParams = new ParamSet();
      evRegParams.setParam("SYMBOL",           argv[1]);
      evRegParams.setParam("TRIGGERVALUE",     argv[2]);

      // 電子メールの送信先を指定するために Action メソッドによって
      // 使われる別の ParamSet を作成する。最後の 2 つの
      // コマンドライン引数を値とみなす
      ParamSet acRegParams = new ParamSet();
      acRegParams.setParam("SMTPhost",  argv[3]);
      acRegParams.setParam("Addressee", argv[4]);

      // トピック「STOCKS」用の EventTopicDef を作成し、
      // エバリュエータ クラス EvaluateStocks と
     // アクション クラス ActionEmail に、そのトピックに対する関心を登録する
      EventTopicDef topic =
        t3.services.events().getEventTopic("STOCKS");
      Evaluate eval =
          new Evaluate("tutorial.event.stocks.EvaluateStocks",
                               evRegParams);
      Action action =
          new Action("tutorial.event.stocks.MailStockInfo",
                             acRegParams);

      // EventRegistration を WebLogic Server に提出する
      EventRegistrationDef er = topic.register(eval, action);
      int regid = er.getID();
      System.out.println("Registration ID is " + regid);
    }
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
  }
}
```

WebLogic Server へのイベント送信

イベントへの関心を登録した後は、さらに、評価のために WebLogic Server にイベントを提出するクラスが 1 つ必要になります。この例では、一連のコマンドライン引数を取り、それらを使って WebLogic Server へイベントを提出するためのパラメータを設定する簡単なクラス（Register クラスのようなもの）を示します。

手順 1. パッケージのインポート

このクラスでは、パッケージ `weblogic.common.*` と `weblogic.event.common.*` をインポートします。

手順 2. コマンドライン引数のチェック

この例では、イベントを限定するパラメータを指定するようユーザに要求します。ここでコマンドライン引数の数をチェックし、数が合わない場合には、使い方の例を示します。

```
if (argv.length != 4) {
  System.out.println("Usage: "
            + "java tutorial.event.stocks.SendEvents "
    + "WebLogicURL STOCKSSYMBOL STARTPRICE ENDPRICE");
  System.out.println("Example: "
            + "java tutorial.event.stocks.SendEvents "
    + "t3://localhost:7001 SUNW 75 95");
  return;
  }
```

手順 3. コマンドライン引数の処理

ユーザが指定した最初の引数、つまり WebLogic Server の URL を使って、T3Client を作成します。

```
T3Client t3 = null;
try {
```

```
t3 = new T3Client(argv[0]);
t3.connect();
```

EventMessage のコンストラクタで使う ParamSet の値として、残りのコマンドライン引数を使います。このイベントの株式シンボルを売ろうとする価格の上限と下限を指定した後、その範囲内の各整数を別個のイベントとして WebLogic Server に提出します。イベントごとに新しい EventMessage を要求し、新しい ParamSet オブジェクトを作成するのではなく、ループ内で、同じオブジェクトを再利用し、提出のたびにパラメータをリセットします。WebLogic Event コードの効率向上の詳細については、上記を参照してください。

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
ParamSet eventParameters = new ParamSet();
eventParameters.setParam("SYMBOL", argv[1]);
int open  = Integer.parseInt(argv[2]);
int close = Integer.parseInt(argv[3]);
```

手順 4. イベントの提出

ループ内で、価格の幅に対応する一連のイベントを提出します。ループでは、価格の範囲に渡る繰り返し、パラメータのリセット、および EventTopic へのイベントの提出以外の処理は行いません。

```
      for (int bid = open; bid < close; bid++) {
        eventParameters.setParam("BID", bid);
        System.out.println("Injecting price event with BID = " +
bid);
        String status = topic.submit(eventParameters);
      }
    }
```

最後に、WebLogic Server から接続解除します。

```
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
  }
```

これで、WebLogic Server にイベントを提出するためのクラスは完成です。完全なコード例は以下のとおりです。

SendEvents クラスのコード

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.common.*;

public class SendEvents {

  public static void main(String argv[]) throws Exception {

    // コマンドライン引数の数をチェックする
    if (argv.length != 4) {
      System.out.println("Usage: "
               + "java tutorial.event.stocks.SendEvents "
               + "WebLogicURL STOCKSSYMBOL STARTPRICE ENDPRICE");
      System.out.println("Example: "
               + "java tutorial.event.stocks.SendEvents "
               + "t3://localhost:7001 SUNW 75 95");
      return;
    }

    // 最初のコマンドライン引数として指定された URL を使って、
    // WebLogic Server に接続する
    T3Client t3 = null;
    try {
      t3 = new T3Client(argv[0]);
      t3.connect();

     // Evaluate メソッドが true を返して Action メソッドを呼び出す時点まで
      // 株の付け値を上げる。イベント パラメータを変更するのに、
      // 新しいイベントを作成する必要はなく、新しい ParamSet も作成する
      // 必要はない。値を設定して、
      // イベントを提出するだけでよい。また、イベントを提出するときは、
      // このイベントへの関心の登録時と同じトピック「STOCKS」と
      // 同じパラメータ名「SYMBOL」を
      // 使う必要がある
      ParamSet eventParameters = new ParamSet();
      EventTopicDef topic =
        t3.services.events().getEventTopic("STOCKS");

      // 2 番目のコマンドライン引数を
      // 「STOCKS」パラメータの値に使う
      eventParameters.setParam("SYMBOL", argv[1]);

      // 最後の 2 つのコマンドライン引数を
      // イベントの始値と終値に使う
      int open  = Integer.parseInt(argv[2]);
      int close = Integer.parseInt(argv[3]);

      for (int bid = open; bid < close; bid++) {
        eventParameters.setParam("BID", bid);
```

```
        System.out.println("Injecting price event with BID = " +
bid);
        String status = topic.submit(eventParameters);
      }
    }
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
  }
}
```

以下に示すのは、この例を実行したときに受信される電子メール メッセージのコピーです。

```
Topic: STOCKS
Registration:
Topic : STOCKS
ID      :11
Flags   :+Sink+Phase:true
Evaluate:tutorial.event.stocks.EvaluateStocks
Evaluate Params:
EVENT_CLIENT_REGISTER_TIME = Tue Sep 03 20:09:07  1996
SYMBOL = SUNW
TRIGGERVALUE = 75
EVENT_CLIENT_REGISTER_HOST = bigbox/107.4.192.255
EVENT_CLIENT_REGISTER_THREAD = main
EVENT_SERVER_REGISTRATION_THREAD = ExecuteThread
EVENT_SERVER_REGISTRATION_TIME = Tue Sep 03 20:09:10  1996

Action  :tutorial.event.stocks.MailStockInfo
Action Params:
EVENT_CLIENT_REGISTER_TIME = Tue Sep 03 20:09:07  1996
SMTPhost = smtp.myhost.com
Addressee = abc@myhost.com
EVENT_CLIENT_REGISTER_HOST = bigbox/107.4.192.255
EVENT_CLIENT_REGISTER_THREAD = main
EVENT_SERVER_REGISTRATION_THREAD = ExecuteThread
EVENT_SERVER_REGISTRATION_TIME = Tue Sep 03 20:09:10  1996

Count   :UNCOUNTED
EventMessage Parameters:
SYMBOL = SUNW
BID = 75
EVENT_SERVER_SUBMIT_THREAD = ExecuteThread
EVENT_SERVER_SUBMIT_TIME = Tue Sep 03 20:09:28  1996
--------------------------------------------
```

クライアントサイド通知の使い方

WebLogic Server ではなくクライアントで Action を実行することもできます。クライアントサイド通知を使用すると、T3Client はイベントへの関心を登録する際に、ローカルの JVM で動作する Action オブジェクトを登録に対して指定できます。WebLogic Server 上のクラスの完全パッケージ名に相当する String を指定して Action オブジェクトを作成するのではなく、weblogic.event.actions.ActionDef を実装する（ローカル）オブジェクトへの参照を指定して Action オブジェクトを作成します。

以下は、T3Client がイベントへの関心を登録する例で、クライアントサイド通知を使えるように Action オブジェクトを作成する方法を示します。この登録用の Action オブジェクトはオブジェクト「clientSideNotify」（weblogic.event.actions.ActionDef を実装）への参照であり、このオブジェクトはクライアント内でインスタンス化され、その action() メソッドは、Evaluate クラスの evaluate() メソッド（常に WebLogic Server 内で実行される）が成功するたびに呼び出されます。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();

Action action  = new Action(new clientSideNotify());
Evaluate eval  =
     new Evaluate("weblogic.event.evaluators.EvaluateTrue");

try {
  EventTopicDef topic =
    t3.services.events().getEventTopic("STOCKS");
  EventRegistrationDef er =
     topic.register(eval, action,
                    true,  // sink
                    true,  // phase
                    EventRegistrationDef.UNCOUNTED);

  int localregID = er.getID();
}
```

別のオブジェクトを指定する必要はありません。通知を受け取るオブジェクトとして、「this」を指定することができます。

tutorial\event\clientside\client1.java に、クライアントサイド通知の簡単な例があります。

WebLogic レルム内での WebLogic Event 用 ACL のセットアップ

WebLogic では、イベントなどの内部リソースへのアクセスは、WebLogic レルム内にセットアップされた ACL によって制御されます。WebLogic レルム内の ACL のエントリは、`weblogic.properties` ファイルにプロパティとして記述されています。

プロパティ ファイルにプロパティを入力することで、「submit」と「receive」というパーミッションをイベントに設定できます。ACL は登録の制御と下位トピックからのイベントのフィルタ処理も行うので、receive パーミッションには二重の目的があります。

ACL 名「weblogic.event」は、すべてのイベント サービスへのアクセスを制御します。ACL 名「weblogic.event」のパーミッション「submit」および「receive」を「everyone」に設定すれば、より特化したパーミッションが設定されていないかぎり、誰でもイベントを提出したり受け取ったりすることができます。

複数のパーミッション（この場合は「submit」と「receive」）を持つ特定のオブジェクト用の ACL を作成する場合は、パーミッションごとに ACL を作成しなければなりません。より一般的な ACL の場合でも、パーミッションは提供しません。

たとえば、一般的な ACL を作成して、高レベルのトピック「weather.northamerica」についてのイベント受信にパーミッションを設定し、誰もがそのトピックに関するイベントを受信できるようにした後、joe と bill だけがトピック「weather.northamerica.us」についてのイベントを提出できるような ACL を作成した場合には、より一般的なトピックについてはイベント通知を受け取るパーミッションを全員に与える ACL を作成したにもかかわらず、誰もそのトピックについてのイベントを受け取ることができなくなります。受け取るには、そのための ACL を別途作成しなければなりません。トピック「weather.northamerica.us」についての任意のアクションに対するパーミッション用の ACL を作成する場合には、そのトピックについてのあらゆるパーミッションをユーザに指定しなければなりません。

この ACL が設定されない場合には、全員がイベントを提出したり、受け取ったりすることができます。

例 :

```
weblogic.allow.receive.weblogic.event.weather.us=everyoneweblogic
.allow.submit.weblogic.event.weather.us=weatherWireweblogic.allow
.receive.weblogic.event.weather.us.ca.sf=billc,sam,donweblogic.al
low.submit.weblogic.event.weather.us.ca.sf=weatherWire
```

この場合には、両方の「submit」パーミッションが必要になります。下位トピック「weather.us.ca.sf」についてのイベント通知を 3 ユーザにしか許可しないように具体的なパーミッションが設定されているので、そのトピックの「submit」についての具体的なパーミッションも設定する必要があります。そうしないと、誰もその下位トピックについてのイベントを提出できなくなります。