# BEA WebLogic Server

## WebLogic Workspace

## ユーザーズ ガイド

## （非推奨）

WebLogic Server 6.1
マニュアル第 1.0 版
2001 年 11 月 30 日

## 著作権



## 限定的権利条項



## 商標または登録商標



**WebLogic Workspace ユーザーズ ガイド（非推奨）**

| 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 Workspace ユーザーズ ガイド（非推奨）

WebLogic ワークスペースの使い方

はじめに
- ワークスペースの概要

API
- WebLogic ワークスペース API リファレンス
- WebLogic ワークスペース API の概要

ワークスペースを使用した実装
- ワークスペースへのアクセスの取得
- ワークスペースの作成、切断、および再利用
  - ワークスペース ID を使用したワークスペースの作成と再利用
  - ワークスペース名を使用したワークスペースの作成と再利用
- 名前付きワークスペースを使用したオブジェクトの格納と取得
- ワークスペースの仕様変更への対応
- ワークスペース モニタの使い方
  - モニタのタイプ
  - モニタの動作と仕組み
  - モニタの作成
  - ワークスペースへのモニタの追加
  - モニタの例
  - ワークスペースからのモニタの削除
- WebLogic レルム内でのワークスペース用 ACL のセットアップ

その他の関連マニュアル
- WebLogic のインストール（Windows 以外）
- WebLogic のインストール（Windows）
- WebLogic クライアント アプリケーションの作成
- 開発者ガイド
- API リファレンス マニュアル
- 用語集
- コード例

**注意：** WebLogic Server のドキュメントは、このリリースで改訂されています。改訂されたドキュメントは、上に挙げたドキュメントに取って代わります。

# ワークスペース API の非推奨

WebLogic ワークスペース API は、このリリースより非推奨になりました。

一時的な格納場所としてワークスペース以外のものを使用する方法については、『WebLogic JNDI プログラマーズ ガイド』の「「データ キャッシュ」設計パターン」を参照してください。

# はじめに

## ワークスペースの概要

WebLogic は、アプリケーションの作成に役立つさまざまなサービスや機能を提供しています。たとえば、共有ログ、インスツルメンテーション、コンフィグレーション、管理などの機能は、WebLogic フレームワーク内で動作するあらゆるアプリケーションから利用できます。

これらの機能の 1 つがワークスペースです。WebLogic Server は、階層化されたスレッドセーフなワークスペース群のホストとなります。これらのワークスペースは、クライアント、クライアント グループ、および WebLogic Server 自体に割り当てられます。ワークスペースには任意のオブジェクトを格納することができ、名前を付けて保存しておけば、複数のセッションにまたがって使用することもできます。その他、アプリケーションが一定のメソッドを実行してからワークスペースの中身を削除または保存できるように、ワークスペースの中身をモニタすることもできます。

T3Client のリソースのクリーン アップに関する WebLogic Server のコンフィグレーションによっては、T3Client は、システムがその T3Client に割り当てたクライアント ワークスペースの内部に、サブワークスペースを作成したり、同じワークスペースを繰り返し使用したりすることも可能です（T3Client のライフタイムの詳細については「切断タイムアウト」を参照）。

ワークスペースには、クライアントに関するさまざまな情報（T3User オブジェクト、JDBC 接続などに関連したコンテキスト、状態等）が格納されます。

現在、ワークスペースのレベルには、クライアント レベル（WorkspaceDef の final static int `SCOPE_CLIENT` で定義）と、サーバ レベル（WorkspaceDef の final static int `SCOPE_SERVER` で定義）の 2 つがあります。

ワークスペースを使えば、オブジェクトの共有も可能になります。あるクライアントのワークスペースに格納されているオブジェクトに、他のクライアントからアクセスすることもできます。また、サーバのワークスペースに格納されたオブジェクトには、どのクライアントからでもアクセスできます。WebLogic Server 自体は、オブジェクト（データベースの ResultSet など）をサーバ ワークスペースに格納できます。また、その WebLogic Server のクライアントすべてに、結果セットへのアクセス権を与えることができます。

# API

WebLogic ワークスペース API リファレンス

WebLogic Workspace API に関する詳細なリファレンスは、WebLogic Server API リファレンスの以下のパッケージを参照してください。

- `weblogic.common`
- `weblogic.workspace.common`

# WebLogic ワークスペース API の概要

ワークスペースは、WebLogic フレームワークに用意されている T3Client の強力なツールの 1 つです。ワークスペースの API は、T3Client の API と密接に関係し合っており、インタフェースやクラスの多くは、`weblogic.common` パッケージ

に入っています。ここでは、weblogic.common パッケージのワークスペース関連のクラスと weblogic.workspace.common の WorkspaceDef インタフェースがどのように関係し、相互に作用するかを簡単に説明します。

# ワークスペースを使用した実装

ワークスペースへのアクセスの取得

ワークスペースの作成、切断、および再利用
- ワークスペース ID を使用したワークスペースの作成と再利用
- ワークスペース名を使用したワークスペースの作成と再利用

名前付きワークスペースを使用したオブジェクトの格納と取得

ワークスペースの仕様変更への対応

ワークスペース モニタの使い方
- モニタのタイプ
- モニタの動作と仕組み
- モニタの作成
- ワークスペースへのモニタの追加
- モニタの例
- ワークスペースからのモニタの削除

WebLogic レルム内でのワークスペース用 ACL のセットアップ

# ワークスペースへのアクセスの取得

ワークスペースなどの現在使用可能なサービスと機能を利用するには、ファクトリ メソッドを使用します。ファクトリ メソッドは、おおまかに言うと、WebLogic Server 中のリソースの割り当てをきめ細かく設定するための、コンストラクタの代わりとなるメソッドです。

T3Client からワークスペースの機能を利用するには、**services** オブジェクトを使用します。T3Client を作成すると、それに対応するワークスペースが自動的に作成されます。このデフォルトのワークスペースへのアクセスを要求するためのコードは、以下のようになります。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

WorkspaceDef オブジェクトにより、ワークスペースのすべての機能が T3Client から使用できるようになります。T3Client アプリケーションで ワークスペースを利用する前に、デフォルトの T3Client ワークスペースを定義している WorkspaceDef オブジェクトへの参照を取得する必要があります。

# ワークスペースの作成、切断、および再利用

`weblogic.workspace.common.WorkspaceDef`

`weblogic.common.WorkspaceServicesDef.getWorkspace()`

# ワークスペース ID を使用したワークスペースの作成と再利用

クライアント ワークスペースは、T3Client を作成するときにデフォルトで作成されます。デフォルト ワークスペースを含むすべてのワークスペースの識別には、作成時に WebLogic Server によって割り当てられるユニークな ID を使用します。この ID を取得して保存すれば、そのワークスペース ID を利用して同じワークスペースを他の T3Client から利用することもできます。次にその例を示します。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
// クライアントが切断したあとでも、ワークスペースが存続するよう
// ワークスペースのタイムアウトを設定する
t3.setSoftDisconnectTimeoutMins(T3Client.DISCONNECT_TIMEOUT_NEVER
);

// デフォルトの T3Client ワークスペースを取得する
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
String wsid = defaultWS.getID();
// . . . 何らかの処理を行う . .
t3.disconnect();

// 再接続し、ワークスペース ID の wsid を渡す
T3Client newt3 = new T3Client("t3://localhost:7001", wsid);
newt3.connect();
// . . . 処理を完了する . .
// 処理が完了したら、ワークスペースを直ちに
// クリーンアップして切断できる状態にする
```

```
newt3.setSoftDisconnectTimeoutMins(0);
newt3.disconnect();
```

# ワークスペース名を使用したワークスペースの作成と再利用

T3Client の作成時にデフォルトのワークスペースの名前を引数として指定すれば、そのワークスペースの名前を設定できます。また、この名前を使用すると、以下のように同じワークスペースを後で再び利用できます。

```
T3Client t3 = new T3Client("t3://localhost:7001",
                           "MY_CLIENT_WS");
t3.connect();
// クライアントが切断したあとでも、ワークスペースが存続するよう
// ワークスペースのタイムアウトを設定する
t3.setSoftDisconnectTimeoutMins(T3Client.DISCONNECT_TIMEOUT_NEVER
);
// . . . 何らかの処理を行う . .
t3.disconnect();

T3Client newt3 = new T3Client("t3://localhost:7001",
                              "MY_CLIENT_WS");
newt3.connect();
// . . . 処理を完了する . .
// 処理が完了したら、ワークスペースを直ちに
// クリーンアップして切断できる状態にする
newt3.setSoftDisconnectTimeoutMins(0);
newt3.disconnect();
```

setSoftDisconnectTimeout() メソッドに指定した DISCONNECT_TIMEOUT_NEVER は、「WebLogic Server は、クライアントとの接続が終了しても、クライアントが使用していたサーバサイド リソースをクリーンアップしない」という意味です。技術的には、これにより、クライアント ワークスペースのリソースのクリーンアップが永久に（または、WebLogic Server が動作している間は）行われないようにすることも可能ですが、パフォーマンスや効率を考えれば、クライアントにとって不要になった時点でリソースのクリーンアップを行う方がよいでしょう。クライアントの処理が終了したら、setSoftDisconnectTimeout() メソッドには「0」を指定できます。この場合 WebLogic Server は、クライアントとの接続が終了すると、T3Client が使用していたサーバサイドのリソースを即座に解放します。

「あるワークスペースの子（通常は、デフォルト T3Client ワークスペースの子）」という形でサブワークスペースを作成し、名前を付けることも可能です。ユーザが作成したワークスペースは、（厳密には）必ずサブワークスペースとなることに注意してください。たとえば、ワークスペースを SCOPE_SERVER というスコー

プで作成した場合、それはサーバの起動時に作成されたサーバ ワークスペースのサブワークスペースになります。また、スコープに合わないサブワークスペースを作成することはできません。たとえば、SCOPE_CLIENT というスコープを持つサーバ ワークスペースのサブワークスペースは作成できません。

サブワークスペースの機能はワークスペースとまったく同じです。サブワークスペースを使えば、アプリケーション（クライアント）オブジェクトの格納の方法や場所をよりきめ細かに設定できます。WorkspaceDef インタフェースのメソッドはスコープに関係なく、どのワークスペースでも機能しますが、（当然のことながら）ユーザが作成できるのはサブワークスペースだけで、削除できるのは自分の作成したサブワークスペースだけです。このマニュアルでは、厳密には「サブワークスペース」と表記すべきところで「ワークスペース」と表記している場合があります。

次に、サブワークスペースに名前を付け、オブジェクトの格納および抽出を行うためのコード例を示します。名前を付けたサブワークスペースを使用するには、まず次のように、T3Client のデフォルト ワークスペースへのアクセスを取得します。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

次に、「T3Client のデフォルト ワークスペースの子」という形で、サブワークスペースを 1 つ作成します（「子」として作成されたワークスペースは、親が削除されると同時にすべて削除されます。つまり、ここでは、T3Client のデフォルト ワークスペースが削除されれば、その子として作成したワークスペースもすべて削除されるということです）。

```
WorkspaceDef subWS = defaultWS.getWorkspace("DATA_STORE");
```

ワークスペースを作成するときには、その**スコープ**（これによって、ワークスペースがどこに作成されるかが決まりる）を設定できます。サブワークスペースのスコープには以下のレベルがあり、WorkspaceDef インタフェースに final static 宣言された整数値を使って設定します。

- SCOPE_CLIENT は、T3Client ワークスペースのサブワークスペースを作成するためのスコープです。WorkspaceDef を取得するときにスコープを指定しないと、デフォルトによってこの SCOPE_CLIENT が適用されます。
- SCOPE_SERVER は、サブワークスペースを WebLogic Server システムのレベルで作成するためのスコープです。WebLogic Server にアクセスできる T3Client であれば、どれでもこのサブワークスペースに格納されているオブ

ジェクトにアクセスできます。このスコープは、サブワークスペースに格納するオブジェクトに「スタートアップ クラスによって作成され、すべてのクライアントによって使用できる」という特性を持たせたいときに使用します。セキュリティ上の理由から、スコープ サーバのワークスペースのコンテンツは WebLogic Console や Admin サーブレットに表示されないということに注意してください。

ワークスペースを作成するときには、**モード**を指定することもできます。このモードによって、ワークスペースがどのように作成されるかが決まります。ワークスペースを作成または再利用するためのモードには、以下の 3 種類があります。これらのモードは、`WorkspaceDef` インタフェースで final static 宣言された整数値として定義します。

- `WorkspaceDef.CREATE` は、一致する既存のワークスペースが存在しない場合に新規のワークスペースを作成します。
- `WorkspaceDef.ATTACH` は、一致する既存のワークスペースが存在する場合に既存のワークスペースに接続します。
- `WorkspaceDef.OPEN` は、ワークスペースの新規作成と接続の両方を行うためのモードです。一致する名前のワークスペースが存在する場合は接続、存在しない場合は新規作成が行われます。特にモードを指定しない場合は、このモードがデフォルトとして使用されます。

**モード**を使用すると、ワークスペースを作成する条件を非常にきめ細かく設定できます。たとえば、`DATASTORE_SPACE` というシステム スコープのワークスペースを作成してあり、その中にオブジェクトを格納してある場合、そのワークスペースを再利用するときには、`ATTACH` モードを選択します。ただし、存在しないワークスペースに `ATTACH` で接続しようとした場合、例外が送出されるので、それを処理するコードも用意しておきます

同じように、まったく新規にワークスペースを作成する場合は、`CREATE` モードを使用します。ただし、`getWorkspace()` メソッドを `CREATE` モードで呼び出す場合、既存のワークスペースの名前または ID を引数に指定すると例外が送出されます。

以下のコード例は、`WorkspaceDef.CREATE` モードを使用している点を除き、先に紹介したものと同じです。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

```
WorkspaceDef subWS =
  defaultWS.getWorkspace("DATA_STORE",
                         WorkspaceDef.CREATE);
```

サブワークスペースには、さらにサブワークスペースを作成することも可能です。この機能をうまく使えば、オブジェクトを、アプリケーションにとって都合の良い方法で整理することができます。

ワークスペースのすべてのサブワークスペースの名前を列挙するには、`WorkspaceDef.subspaces()` メソッドを使用します。

# 名前付きワークスペースを使用したオブジェクトの格納と取得

ワークスペースは、オブジェクトを格納するのに便利です。格納および取得するオブジェクトを指定するには、名前と値の組み合わせを使用します。任意のオブジェクトを扱うことができますが、以下の制限があります。

- オブジェクトは `java.lang` または `java.util` オブジェクトです。
- または、オブジェクトは `java.io.serializable` です。

ワークスペース中のオブジェクトを管理するのに使用するメソッドは、以下の 3 つです。いずれのメソッドも、String 型の引数 **key** に対象オブジェクトの名前を指定します。

- `WorkspaceDef.store(String key, Object p)`。オブジェクトを格納するために使用します。
- `WorkspaceDef.fetch(String key)`。格納されているオブジェクトをワークスペースから削除せずに取り出すために使用します。
- `WorkspaceDef.remove(String key)`。格納されているオブジェクトを取り出してワークスペースから削除するために使用します。

特定のワークスペースのすべてのキーを取得するには、`WorkspaceDef.keys()` メソッドを使用します。

次に、`DATA_SPACE` というワークスペースを作成し、WebLogic JDBC ResultSet を格納するコード例を示します。`DATA_SPACE` は、システム ワークスペースのサブワークスペースです。

```
T3Client t3 = new T3Client("t3://toyboat.toybox.com:7001");
t3.connect();
// デフォルトの T3Client ワークスペースを取得する

WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
WorkspaceDef dataWS =
  defaultWS.getWorkspace("DATA_WORKSPACE",
                         WorkspaceDef.CREATE,
 WorkspaceDef.SCOPE_SERVER);
// . . . DB に接続して、ResultSetrs rs を取得する . . .
// 次に、作成したシステムのサブワークスペースにそれを格納する
dataWS.store("MyResults", rs);
t3.disconnect();
```

次に、上記とは逆に、ResultSet を取り出すためのコード例を示します。この例では、既存のサブワークスペース DATA_SPACE に接続します。この ResultSet はワークスペース中に（他のクライアントから使用できるように）残しておきたいので、fetch() メソッドを使用して ResultSet を取り出します。取り出したオブジェクトをワークスペースから削除する remove() メソッドは使用しません。

```
T3Client t3 = new T3Client("t3://toyboat.toybox.com:7001");
t3.connect();
// デフォルトの T3Client ワークスペースを取得する
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
// システムの作成済みサブワークスペースに接続する
WorkspaceDef myDataWS =
  defaultWS.getWorkspace("DATA_WORKSPACE",
                         WorkspaceDef.ATTACH,
 WorkspaceDef.SCOPE_SERVER);
// ResultSet を取得してクリーンアップする
ResultSet rs = (ResultSet) myDataWS.fetch("MyResults");
t3.disconnect();
```

# ワークスペースの仕様変更への対応

リリース 2.4 から移行した場合、本リリースではワークスペース機能にアクセスするためのコーディングが大きく変わっていることに気付くはずです。

ワークスペースへのアクセスを取得するメソッドとワークスペースを操作するメソッドは、当初は T3Client クラスに含まれていました。これらのメソッドを使用するには、以下のようなコードを書く必要がありました。

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

しかし新リリースでは、以下に示すように、ワークスペースの機能には T3Client のサービス スタブと weblogic.common.workspace.WorkspaceDef インタフェースを通してアクセスします。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

次に、このインタフェースのメソッドを使用して、WorkspaceDef で処理を行います。getID() メソッドと getName() メソッドは、以前は T3Client クラスに含まれていた同名のメソッドと同じものです。また、ワークスペースにオブジェクトを格納し、ワークスペースからオブジェクトを取り出すためのメソッドは、以前は weblogic.common.WorkspaceServicesDef に含まれていたものと同じです。したがって、これらのメソッドではなく、より強力な WorkspaceDef メソッドを使用するようにしてください。

# ワークスペース モニタの使い方

新しいインタフェースの WorkspaceDef を使用すると、モニタ機能も使用できます。モニタを使用すると、保存や削除といった一定の処理が行われる前後に、ワークスペースの内容に対してユーザが作成したメソッドを呼び出すことができます。モニタは、ビジネス ルールを実装し、ワークフローとデータの検証を一貫した方針のもとで行うのに役立ちます。

モニタに関連するクラスとインタフェースは以下のとおりです。

モニタの設定用

- weblogic.common.SetMonitor
- weblogic.common.GetMonitor
- weblogic.common.DestroyMonitor
- weblogic.common.MonitorException

モニタのアクティブ化用

- weblogic.common.GetMonitor
- weblogic.common.Monitor
- weblogic.common.MonitorDef

モニタは、以下のような処理を行うときに使用します。これらの例は、この節の後半でモニタの作成および使用方法を説明するときに使用します。

- ワークスペース中の値の範囲を限定します。このためには、ワークスペース中の値に対して何らかの処理が行われる前に、値が一定の範囲に納まっているかどうかを確認し、範囲を逸脱したときは MonitorException を発生させるという処理を行う必要があります。
- ワークスペース中の値が削除されたときにイベントを発生させます。アプリケーションは WebLogic フレームワークの中で動作するので、イベントなどの他の WebLogic API にもアクセスできます。このため、イベントとモニタは同時に利用できます。
- ワークスペース中の値を他のワークスペースにミラーリングします。モニタを使用すると、あるワークスペースに加えられた変更を別のワークスペースにミラーリングすることができます。
- ワークスペースの値を設定するときに変更/暗号化します。マスター モニタを使用すると、値を設定しようとするところを捉えて、値を暗号化してから設定することができます。この機能は、データの圧縮や重要な情報の暗号化などに利用できます。

# モニタのタイプ

モニタには、レギュラー モニタとマスター モニタという 2 つのタイプがあります。通常、単に「モニタ」と言えば、レギュラー モニタのことを指します。マスター モニタは対象オブジェクトごとに**1 つだけ**作成できますが、レギュラー モニタはオブジェクトごとに複数作成できます。

レギュラー モニタは、値をモニタして、変更が行われたことを報告したり、値に対する不正な処理を防止したりするためにだけ使用します。レギュラー モニタは、実際に値を変更することはありません。また、1 つの値につき複数のモニタが使用された場合、その動作の優先順位は一定しません。

マスター モニタは、基本的にはレギュラー モニタと同じですが、以下の 2 つの特徴があります。

- マスター モニタは、モニタ対象の処理が実行される前、およびモニタ対象の処理が実行された後に最初に動作します。つまり、マスター モニタの前処理と後処理は、レギュラー モニタの同じ処理に優先します。
- マスター モニタは、対象となるオブジェクトの状態を変更できます。マスター モニタの実行中は、対象オブジェクトのロックが解除され、マスター モニタから変更できるようになります。一方、レギュラー モニタはオブジェ

クトの状態の変更を監視（または防止）するだけです。次のコード例では、マスター モニタは「target」というオブジェクトの値を変更します。この処理は、レギュラー モニタでは行うことができません。

```
public void preSet(Setable target, ParamSet callbackData)
    throws MonitorException {
  target.newValue("ALTERED Value");
}
```

マスター モニタは、レギュラー モニタの setMaster() メソッドを使って作成します。引数には、true を指定します。対象オブジェクトに既にマスター モニタが存在する場合は、MonitorException が発生します。次に、モニタを作成して、それをマスター モニタに設定するためのコード例を示します。

```
Monitor monitor = new Monitor("mycode.MyMonitor", ps);
monitor.setMaster(true);

try {
  workspace.addMonitor(key, monitor);
}
catch (Exception e) {
  inform("addMonitor failed: a Master Monitor " +
         "may already be installed.");
}
```

# モニタの動作と仕組み

モニタが具体的に何をモニタするかは、実装するインタフェースと、対象オブジェクトがサポートしている処理によって決まります。また、オブジェクトがモニタの対象となるためには、Setable、Getable、または Destroyable インタフェースを実装していなければなりません。現在のところ、これらのインタフェースを実装しているオブジェクトはワークスペースだけなので、事実上、対象となるのはワークスペースの値だけです。

ワークスペースの処理のうち、対象となるのは以下の 3 つです。

- 設定（Set）
- 取得（Get）
- 破棄（Destroy）

ワークスペース中のオブジェクトをモニタするには、Destroyable、Setable、Getable のどれかを実装しているワークスペース オブジェクトで *xxx*`Value()` という形式のメソッドのどれかが呼び出されたときに何らかの処理（どのようなものでも良い）を行うクラスを作成します。

モニタを使用すると、特定のワークスペース オブジェクトに対するクエリ、修正または破棄が行われたときに、作成したコードを実行させることができます。

モニタは、以下の 6 つのポイントでワークスペース オブジェクトの処理に介入します。

1. *preSet* と *postSet*。ワークスペース内のオブジェクトの `setValue()`、`newValue()`、`oldValue()` などのメソッドの処理の前後に呼び出されるメソッドです。preSet 処理は、MonitorException の送出によってブロックされます。
2. *preGet* と *postGet*。取得処理の前後に呼び出されるメソッドです。preGet 処理は、MonitorException の送出によってブロックできます。
3. *preDestroy* と *postDestroy*。破棄処理の前後に呼び出されるメソッドです。preDestroy 処理は、MonitorException の送出によってブロックできます。

モニタは、その種類を問わず、ある特定の処理が行われようとしたとき（前処理の段階で）、MonitorException を送出してそれを中断させることができます。ある処理が中断されたら、その旨がすべてのモニタに通知されます。ただし、マスター モニタは、その中断をオーバーライドできます。

設定、取得、または破棄の処理をモニタするためには、モニタは以下のインタフェースを少なくとも 1 つ実装している必要があります。

- `SetMonitor`
- `GetMonitor`
- `DestroyMonitor`

作成するクラスの内部では、特定のワークスペース オブジェクトの値の設定、取得、および破棄処理が呼び出される前と後に呼び出されるタスクを設定できます。これら 3 つのインタフェースをすべて実装するクラスは、開発者によって開発される、設定、取得、破棄という 3 つの処理に対応するメソッドを持ちます。

モニタを使用するために必要な作業は、具体的には以下の 2 つです

- モニタを実際に行うクラスの作成。ワークスペース オブジェクトに対する値の設定、取得、および破棄処理の前後に何らかの処理を行うクラスを作成します。この種のクラスを作成するには、`SetMonitor` インタフェース、`GetMonitor` インタフェース、`DestroyMonitor` インタフェースのうち少なくともいずれか 1 つを実装しなければなりません。

- ワークスペースにモニタを追加するクラスの作成。モニタは WebLogic によってインスタンス化されなければならないが、そのためには、モニタを監視対象のワークスペースに追加する、という処理が必要になります。

まず、上記の各手順のごく簡単な例を紹介します。次に、4 つの異なる例を取り上げてその処理について説明します。

# モニタの作成

モニタは、SetMonitor、GetMonitor、DestroyMonitor という 3 つのインタフェースのうち 少なくともいずれか 1 つと、ワークスペースにモニタを追加してその中の値を監視できるようにするためのクラスを実装している必要があります。

モニタがどのような処理をモニタするかは、監視対象のオブジェクトが実装しているインタフェースによって決まります。以下に、「値を設定する」という処理をモニタする単純なモニタの例を示します。

```
package mycode;

import weblogic.common.*;

public class MyMonitor implements SetMonitor {
  public T3ServicesDef services;
  public void monitorInit(ParamSet params, boolean isMaster) {}

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

  public void preSet(Setable target,
                     ParamSet callbackData)
    throws MonitorException
  {
    System.out.println("preSet called");
  }

  public void postSet(Setable target,
                      ParamSet callbackData,
      Exception e) {
    System.out.println("postSet called");
  }
}
```

# ワークスペースへのモニタの追加

weblogic.workspace.common.WorkspaceDef.addMonitor()

モニタを機能させるためには、モニタをワークスペースに追加して、その中の値をモニタできるように必要があります。そのためには、`WorkspaceDef.addMonitor()` というメソッドを呼び出します。

`addMonitor()` メソッドには、モニタを識別するための名前と Monitor オブジェクトという 2 つのパラメータを指定します。Monitor オブジェクトは、基本的にユーザが作成する MonitorDef のラッパーです。つまり、`SetMonitor`、`GetMonitor`、`DestroyMonitor` を実装するクラスは必ず MonitorDef を実装します。

以下に、`addMonitor()` メソッドに渡す Monitor オブジェクトを作成する例を示します。Monitor オブジェクトのコンストラクタでは、以下のような引数を組み合わせて使うこともあります（モニタが初期化パラメータを必要とするかどうか、またはクライアントとサーバのどちらによって作成されるかで異なる）。

- MonitorDef クラス（つまり、`SetMonitor`、`GetMonitor`、`DestroyMonitor` を実装するクラス）の名前。このクラスには、WebLogic Server の CLASSPATH に存在しなければなりません。WebLogic Server からモニタをワークスペースに追加する場合は、MonitorDef クラスのインスタンスを `addMonitor()` メソッドの引数に指定することもできます。
- 初期化パラメータのセットを格納した ParamSet オブジェクト（オプション）。リモートでクラスをインスタンス化する場合は、引数のないデフォルトコンストラクタが使用されるので、初期化パラメータのセットを送る必要があります。このパラメータは、モニタがインスタンス化されるとすぐに、`monitorInit()` メソッドによって評価されます。
- コールバック パラメータのセットを格納した ParamSet オブジェクト（オプション）。クライアントからモニタを追加する場合、コールバック パラメータをいくつか設定する必要が生じる場合もあります。コールバック パラメータは、モニタの pre メソッド（監視対象のオブジェクトに対する特定の処理が終了する前に呼び出されるメソッド）や post メソッド（監視対象のオブジェクトに対する特定の処理が終了した後に呼び出されるメソッド）に渡されます。

以下に、デフォルト クライアント ワークスペースに Monitor オブジェクトを作成する例を示します。

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

```
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();

initps.setParam("topic", "newBooks");
Monitor myMon =
  new Monitor("mycode.MyMonitor", initps, cbps);
defaultWS.addMonitor("topic", myMon);
```

# モニタの例

ここでは、モニタをワークスペースに組み込むためのコードをより詳しく示すとともに、モニタそのもののコード例も示します。コード例を全部で 4 つあります。いずれも、配布キットの `examples\workspace\monitor` ディレクトリに完全な形で収められています。

**注意：** ワークスペースのコード例は、このリリースには収められていません。

- コード例 1. ワークスペースの値の範囲の制限
- コード例 2. イベントを発生させるモニタ
- コード例 3. ワークスペースの値を別のワークスペースにミラーリングするモニタ
- コード例 4. ワークスペースの値を変更するマスター モニタ

# コード例 1. ワークスペースの値の範囲の制限

`examples.workspace.monitor.RangeMonitor`

これは、ワークスペースの値を一定の範囲に限定するモニタの例です。クライアントがワークスペースの値に変更を加えようとすると、このモニタは新しい値が定められた範囲内にあるかどうかを（新しい値が設定される前に）確認し、範囲を超えている場合は MonitorException を送出します。

*addMonitor()* メソッドを呼び出すためのコード

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
// 前処理と後処理で使う 2 つの ParamSet を作成する
// 実際にはパラメータは不要なので、それらは
// 空のままにしておく
ParamSet initPS = new ParamSet();
ParamSet callbackPS = new ParamSet();
```

```
// ワークスペースを取得する
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
WorkspaceDef workspace = defaultWS;
try {
  Monitor rangeMonitor =
    new Monitor("examples.workspace.monitor.RangeMonitor",
                initPS, callbackPS);

  workspace.addMonitor("key3", rangeMonitor);
  System.out.println("Setting Value within range 0-100");
  workspace.store("key3", new Integer(50));
  try {
    System.out.println("Setting Value outside range 0-100");
    workspace.store("key3", new Integer(150));
  }
  catch (T3Exception ex) {
    System.out.println("Received Exception for " +
                       "out-of-range value");
  }
catch (Exception e) {
  e.printStackTrace();
}
t3.disconnect();
```

モニタそのもののコード

ここに示したのは、preSet() メソッドと postSet() メソッドの実装（処理はすべて preSet() メソッドが行う）の部分だけです。完全なコードは、examples\workspace\monitor\RangeMonitor.java に収められています。このクラスは SetMonitor インタフェースを実装しており、「target」として指定されたワークスペースのオブジェクトをモニタします。

**注意：** ワークスペースのコード例は、このリリースには収められていません。

```
public void preSet(Setable target,
                   ParamSet callbackData)
  throws MonitorException
{
  if (!target.newValue() instanceof Integer) {
    throw MonitorException("Value must be of type Integer");
  }
  integer newValue = ((Integer) target.newValue()).intValue();
  if (newValue < 0 || newValue > 100) {
    throw MonitorException(newValue +
                           " must be between 0 and 100");
}

public void postSet(Setable target,
                    ParamSet callbackData,
    Exception e) {}
```

# コード例 2. イベントを発生させるモニタ

ここでは、ワークスペース中の値が破棄されたときにイベントを発生させるモニタのコード例を紹介します。この場合、モニタを組み込むクラスが `weblogic.event.actions.ActionDef` を実装していなければならないという点に注意してください。

*addMonitor()* メソッドを呼び出すためのコード

ここでは、ワークスペース中の値が削除されたときにイベントを発生させます。イベントを発生させる方法については、開発者ガイドの『WebLogic Events の使い方』を参照してください。

要点を簡潔に示すため、このコード例では T3Client （t3）を作成および接続するコード、初期化パラメータやコールバック パラメータ（initPS と callbackPS）を作成するコード、およびワークスペース（workspace）を取得するためのコードは省略されています。最初のコード例で詳細を確認するか、`examples\workspace\monitor\MonitorDemo.java` に収められている完全なコード例を参照してください。

**注意：** ワークスペースのコード例は、このリリースには収められていません。

ワークスペースの値が削除されると、destructionMonitor がイベントを発生させます。「lock」を使用して、通知が届くまでスレッドを強制的に待機させます。

```
initPS.setParam("topic", "destroyTopic");
Monitor destuctionMonitor =
  new Monitor("examples.workspace.monitor.DestructionMonitor",
              initPS, callbackPS);

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

EventRegistrationDef destroyReg =
  t3.services.events().getEventRegistration("destroyTopic",
                                            eval, action);
destroyReg.register();

workspace.addMonitor("key1", destuctionMonitor);
System.out.println("Storing value under key1");
workspace.store("key1", "testValue");
System.out.println("Removing key1 from workspace " +
                   "(expect message notification)");
synchronized(md.lock) {
  workspace.remove("key1");
  md.lock.wait();
}
```

```
// これは ActionDef を実装する。これは、このクラスが EventRegistration を
// 提出するために行わなければならないアクションを定義している
public void setServices(T3ServicesDef services) {}

public void registerInit(ParamSet ps) {}

public void action(EventMessageDef message) {
  System.out.println("Message Received: " + message.getTopic());
  synchronized(lock) {
    lock.notify();
  }
}
```

モニタそのもののコード

ここに示したのは、実質的な処理を行う postDestroy() メソッドだけです。このクラスは、DestroyMonitor インタフェースを実装しており、postDestroy() メソッドの引数「target」には、Destroyable インタフェースを実装したワークスペース オブジェクトが指定されます。

```
  public void postDestroy(Destroyable target,
                          ParamSet callbackData, Exception e) {
    if (e == null) {
      ParamSet ps = new ParamSet();
      try {
        if (target instanceof WorkspaceValue) {
  ps.setParam("key", ((WorkspaceValue) target).getKey());
}
EventMessageDef em =
  services.events().getEventMessage(topic, ps);
em.submit();
      }
      catch (ParamSetException pse) {
      }
      catch (EventGenerationException ege) {}
    }
  }
```

# コード例 3. ワークスペースの値を別のワークスペースにミラーリングするモニタ

*addMonitor()* メソッドを呼び出すためのコード

要点を簡潔に示すため、このコード例では T3Client （t3）を作成および接続するコード、初期化パラメータやコールバック パラメータ（initPS と callbackPS）を作成するコード、およびワークスペース（workspace）を取得するためのコード

は省略されています。最初のコード例で詳細を確認するか、examples\workspace\monitor\MonitorDemo.java に収められている完全なコード例を参照してください。

**注意：** ワークスペースのコード例は、このリリースには収められていません。

```
try {
  // MirrorMonitor は SetMonitor オブジェクトと DestroyMonitor
  // オブジェクトを使用して
  // 別のワークスペース内のワークスペース値の状態を
  // ミラーリングする
  initPS.setParam("mirror", "mirroredWorkspace");
  Monitor mirrorMonitor =
    new Monitor("examples.workspace.monitor.MirrorMonitor",
                initPS, callbackPS);

  workspace.addMonitor("key2", mirrorMonitor);
  String mirrorVal = "mirror this";
  workspace.store("key2", mirrorVal);
  System.out.println("Set key2 = " + mirrorVal +
                     ", in default workspace");
  WorkspaceDef mirror =
    defaultWS.getWorkspace("mirroredWorkspace",
                           WorkspaceDef.OPEN,
                           WorkspaceDef.SCOPE_SERVER);
  String mirroredValue =
    (String) mirror.fetch("key2");
  System.out.println("Got key2 = " + mirrorVal +
                     ", in workspace " + mirror.getName());

  }
  catch (Exception e) {
    e.printStackTrace();
  }

  // クライアントを切断する。ソフト切断タイムアウトが
  // NEVER なので、WebLogic Server は
  // セッションを保存する
  t3.disconnect();
}
```

モニタそのもののコード

このモニタは、モニタしているワークスペースの値に変更が加えられるたびに、別のワークスペースにそれをミラーリングする、という機能を備えています。

ここでは、値の設定および破棄をモニタするため、SetMonitor インタフェースと DestroyMonitor インタフェースの両方を実装します。監視対象のワークスペース オブジェクト（Setable と Destroyable を実装している）は、「target」として渡されます。

また、Monitor クラスの monitorInit() メソッドと、その引数である ParamSet も活用します。ここでは、ParamSet はミラーリングされたワークスペースの名前を監視対象のクラスに知らせるために使用します。

```
public class MirrorMonitor implements SetMonitor,
                                      DestroyMonitor {
  public T3ServicesDef services;
  WorkspaceDef mirror = null;

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

  public void monitorInit(ParamSet params, boolean isMaster)
    throws ParamSetException
  {
    String mirrorName = params.getParam("mirror").asString();
    mirror =
      services.workspace().getWorkspace(mirrorName,
                           WorkspaceDef.OPEN,
                    WorkspaceDef.SCOPE_SERVER);
  }

  public void preSet(Setable target, ParamSet callbackData)
    throws MonitorException {}

  // これを実装して変更をミラーリングする
  public void postSet(Setable target,
                      ParamSet callbackData,
     Exception e)
  {
    if (e == null && target instanceof WorkspaceValue) {
      try {
        mirror.store(((WorkspaceValue)target).getKey(),
               target.newValue());
      }
      catch (T3Exception t3e) {}
    }
  }

  public void preDestroy(Destroyable target,
                         ParamSet callbackData) {}

  // これを実装して破棄をミラーリングする
  public void postDestroy(Destroyable target,
                          ParamSet callbackData,
  Exception e) {
    try {
      mirror.destroy(((WorkspaceValue)target).getKey());
    }
    catch (T3Exception t3e) {}
  }
}
```

# コード例 4. ワークスペースの値を変更するマスター モニタ

マスター モニタは、監視対象オブジェクト 1 つにつき 1 つしか作成できません。また、マスター モニタは、監視対象の処理が実行される前または後に最初に動作するモニタです。マスター モニタがレギュラー モニタと異なっているのは、マスター モニタの実行中は対象オブジェクトのロックが解除されるという点です。

モニタをマスター モニタにするには、そのモニタ自体の `Monitor.setMaster(true)` メソッドを呼び出します。既にマスター モニタが存在していれば、MonitorException が発生します。

次に、マスター モニタのコード例を示します。この例では、セキュリティ確保のため、ワークスペース オブジェクトに格納される値を暗号化（エンコード）します。

*addMonitor()* を呼び出し、モニタをマスター モニタにするためのコード

```
  try {
    // XXXMonitor は MasterMonitor の例で、これは、
    // モニタ対象の値を変更できる。モニタは
    // 与えられた値を "XXX" に置き換える
    Monitor xxxMonitor =
      new Monitor("examples.workspace.monitor.XXXMonitor",
                  initPS, callbackPS);
    xxxMonitor.setMaster(true);

    String val = "hello world";
    workspace.addMonitor("key4", xxxMonitor);
    workspace.store("key4", val);
    System.out.println("Set Value: "+val);
    val = (String) workspace.fetch("key4");
    System.out.println("Got Value: "+val);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
```

モニタそのもののコード

このコード例では、具体的な値が入るべきところを `XXX` と表記しています。暗号化のフォームを使用するかわりに、`preSet()` メソッドを作成して、値を取り出すときに復号化できるようにすることもできます。

```
public class XXXMonitor implements SetMonitor {
  T3ServicesDef services;
  boolean isMaster;
```

```
  public void monitorInit(ParamSet params, boolean isMaster) {
    this.isMaster = isMaster;
  }

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

  public void preSet(Setable target, ParamSet callbackData)
    throws MonitorException
  {
    if (isMaster) {
      target.newValue("XXX");
    }
  }

  public void postSet(Setable target,
                      ParamSet callbackData,
      Exception e) {}
}
```

## ワークスペースからのモニタの削除

モニタをワークスペースから削除するには、以下のようにワークスペース オブジェクトの `removeMonitor()` メソッドを使用します。

```
workspace.removeMonitor(key, monitor);
```

このメソッドを使用すると、*key* に指定されたワークスペースのモニタが削除されます。このメソッドは、引数を 1 つだけ（つまりモニタ自体を）指定して呼び出すこともできます。

# WebLogic レルム内でのワークスペース用 ACL のセットアップ

`weblogic.workspace`

`weblogic.workspace.`*`namedWorkspace`*

WebLogic では、ワークスペース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。

**注意：** WebLogic Server の本リリースには、すべてのコンフィグレーションおよび管理タスクを実行できる Web ベースの Administration Console が用意されています。このため、weblogic.properties ファイルは使用されません。

プロパティ ファイルにプロパティを記述することで、ワークスペース（ユーザが名前を付けて作成したものを含む）に `read` および `write` パーミッションを設定できます。作成されたすべてのワークスペースへのアクセスは、`weblogic.workspace` という ACL によって制御されます。ユーザの名前付きワークスペースの ACL を設定しないままにしておくと、その ACL はユーザのログイン時に動的に作成されます（`weblogic.workspace` の ACL をコピーして、read および write パーミッションを追加します）。これによって、明示的な ACL がなくても、各ユーザは自分専用のワークスペースを使えるようになりますが、他のワークスペースに対する明示的な読み書きパーミッションもユーザに与えられます。

次に、ACL の例をいくつか示します。最初の ACL （最初の ACL エントリのペア）は、4 人のユーザに、T3UserSales の名前付きワークスペース内のオブジェクトと、接続時に各ユーザに対して作成されるワークスペース内のオブジェクトに対する読み込みおよび書き込みを許可するものです。2 番目の ACL （次の 2 行）は、T3User `sysMonitor` に、あらゆるワークスペースに対する読み込みおよび書き込みアクセス権を与えるものです。ワークスペースが作成されるたびに、この ACL がコピーされて、ログインしたユーザに読み込みおよび書き込みパーミッションがそれに追加されます。

例：

```
weblogic.allow.read.weblogic.workspace.T3UserSales=karl,michael,skip,msmithweblogic.allow.write.weblogic.workspace.T3UserSales=karl,michael,skip,msmithweblogic.allow.read.weblogic.workspace=sysMonitorweblogic.allow.write.weblogic.workspace=sysMonitor
```