intra-mart Accel Platform

IM-JavaEE Framework 仕様書

2012/10/01 初版

＜＜　変更履歴　＞＞

変更年月日	変更内容
2012/10/01	初版

<< 目次 >>

1 はじめに

1.1 目的

本書は IM-JavaEE Framework の詳細仕様を説明する。

1.2 本書で扱う範囲

本書はintra-mart AccelPlatform付属する IM-JavaEE Framework の仕様について述べている。各種の制限事項や動作環境等についてはこれらに依存する。

本書は IM-JavaEE Framework のインタフェースや外部仕様について説明している。また、コンポーネントの使われ方やその手順などの内部仕様についても一部提示しているが、内部の実装などについては触れていない。

本書では説明を補強するために複数のサンプルを提示している。これらは各節や章における仕様を理解することを第一の目的としているものであり、プログラムやシステムとしての最適解を示しているものではない。そのため、一部の例外処理などは意図的に実装していないものもある。

本書では読者が上流工程に関係する場合でも有用な情報が含まれているが、プロジェクト管理や業務分析などの詳細な方法については触れていない。

本書では JavaTM Standard Edition (J2SE)や JavaTM Enterprise Edition (JavaEE)の詳細については触れていない。

1.3 本書の対象読者

本書は以下の読者を対象としている。

- 設計者
  設計者に対してはどのようなインタフェース仕様にするべきか、コンポーネントをどのように分割するべきかを提示している。
- 実装者
  実装者に対してはコンポーネントを作成するための情報やその仕様について提示している。

プロジェクト管理者や業務分析者などにとっては、本書は特に明示的な情報を提示していない。具体的な分析、管理を行う場合の参考にとどまる。

また、以下の技術や手法に対してある程度の理解があることが望ましい。

- JavaTM Standard Edition
- JavaTM Enterprise Edition
- UML
- デザインパターン
- XML 1.1[3]

1.4 構成

本書は以下のような構成をしている。

- 「2 アプリケーションの構成」では IM-JavaEE Framework でアプリケーションを作成する場合の構成について示している。
- 「3 サービスフレームワーク」～「7 ログフレームワーク」では IM-JavaEE Framework のサブフレームワークについて詳細な動作やインタフェース、コンポーネントの作成方法などについて説明している。

2 アプリケーションの構成

2.1 概要

IM-JavaEE Framework は中・大規模 Web システムの構築基盤となるものである。IM-JavaEE Framework 上で動作するアプリケーションは、Sun Microsystems が提唱している BluePrints[4]に従った構成となる。

2.2 構成

2.2.1 全体

アプリケーションの構成は「図 2-1 アプリケーションの構成」に示したようなものとなる。

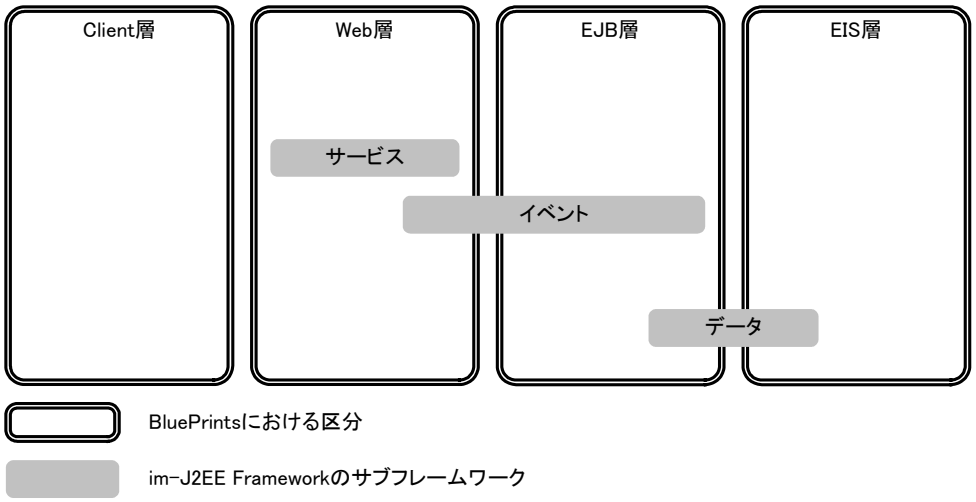


図 2-1 アプリケーションの構成

各サブフレームワークの概要は以下のとおりである。

- サービスフレームワーク(「3 サービスフレームワーク」を参照)
  クライアント(主にブラウザ)からのリクエストを受け付け、画面遷移とその表示をする。
- イベントフレームワーク(「4 イベントフレームワーク」を参照)
  何らかの処理を行う。
- データフレームワーク(「5 データフレームワーク」を参照)
  EIS 層(データベースやレガシーシステムなど)と連携し、データのやり取りを行う。

また、システム全体に関連するサブフレームワークとして以下のものがある。

- メッセージフレームワーク(「2.2.5 メッセージフレームワーク」を参照)
  地域対応したメッセージを取得する。
- ログフレームワーク(「7 ログフレームワーク」を参照)
  ログの出力を行う。

各サブフレームワークは「図 2-2 フレームワーク間の協調関係」のような協調関係にある。

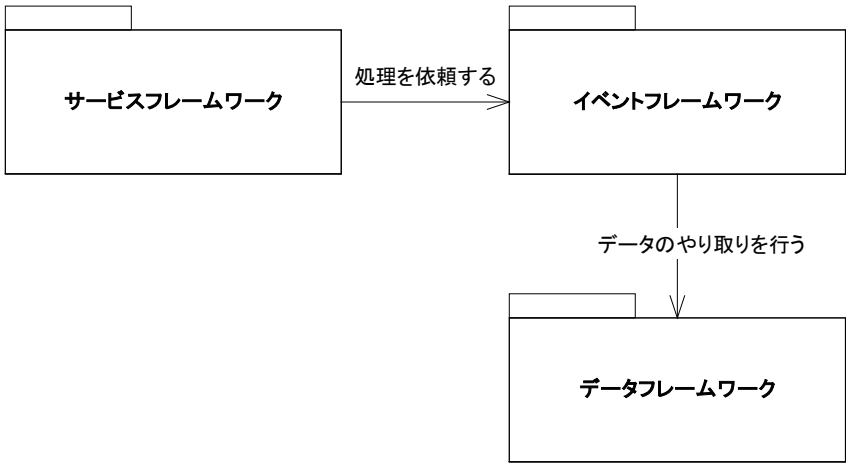


図 2-2 フレームワーク間の協調関係

2.2.2 サービスフレームワーク

サービスフレームワークではクライアント(主にブラウザ)から来たリクエストを Web 層において処理をする。サービスフレームワークは入力(ブラウザからのリクエスト)を受け付け、要求に対応する処理を割り当て、次の画面への遷移を制御する。
要求に対する詳細な処理はここでは記述せず、イベントフレームワークにおいて行うことを推奨する。

2.2.3 イベントフレームワーク

イベントフレームワークでは入力内容に対して何らかの処理を行い、処理結果を返すビジネスロジックを制御する役割を持つ。サービスフレームワークでもビジネスロジックを埋め込むことは可能であるが、イベントフレームワークを利用したほうがより汎用的になる。

2.2.4 データフレームワーク

データフレームワークは EIS 層(データベースやレガシーシステムなど)と連携し、データのやり取りを行う。一般にデータストアに対するアクセス方法はそれぞれ異なるが、データフレームワークでは呼び出し側がそのアクセス方法を意識せずにコーディングができる方法を提供する。

2.2.5 メッセージフレームワーク

メッセージフレームワークは地域対応したメッセージを取得する。国際化されたアプリケーションでは、それぞれの地域のロケールをもとに地域対応した文字列を取得する方法を提供する。

2.2.6 ログフレームワーク

ログフレームワークではログを出力する。ログを出力タイミングと内容、出力先とその書式はさまざまなものがあるが、ログフレームワークではこれらを別々にわけて管理する方法を提供する。

2.2.7 プロパティ

IM-JavaEE Framework ではアプリケーションという概念がある。アプリケーションは複数のコンポーネントをまとめる単位である。アプリケーションとコンポーネントはプロパティの設定によって関連付けられる。プロパティの設定情報の取得は PropertyHandler を通じて行われる。PropertyHandler はそれぞれのサブフレームワークごとに対応したものが存在し、アプリケーション個別のものとすべてのアプリケーションに共通するものがある。

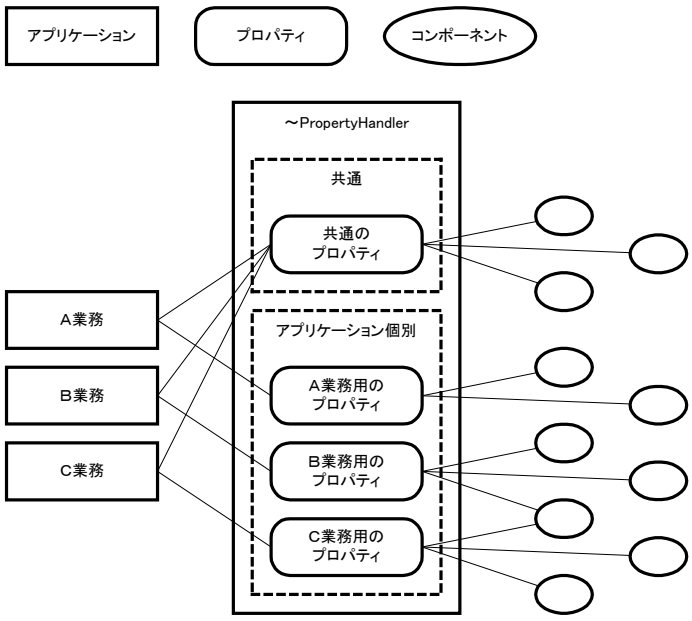


図 2-3 PropertyHandler

2.2.7.1 アプリケーション

アプリケーションは複数のコンポーネントとの関連をまとめる単位である。個々のアプリケーションはアプリケーション ID またはアプリケーション名で識別され、IM-JavaEE Framework の基本単位となる。PropertyHandler からアプリケーション個別のプロパティを取得する場合、アプリケーション ID またはアプリケーション名を指定する必要がある。

2.2.7.2 共通

プロパティの中にはアプリケーションに依存しないものも存在する。これらのプロパティにはアプリケーション ID やアプリケーション名は存在しない。PropertyHandler からアプリケーション共通のプロパティを取得する場合、アプリケーション ID やアプリケーション名を指定する必要はない。

3 サービスフレームワーク

3.1 概要

Web システムを構築する場合、ユーザから見た主な利用形態は 1)ブラウザから情報をサーバに送り、2)サーバ側の処理結果を表示、というのが一般的なものである。

また、入力や表示を複数の地域に対応させるためには、アプリケーションの国際化が必須となる。

サービスフレームワークではこれらの一連の流れで共通的な部分をフレームワーク化している。

3.2 構成

3.2.1 構成要素

IM-JavaEE Framework では Web クライアント（主にブラウザ）から何らかのリクエストがきてそれに対する画面遷移を「サービス」という単位で扱っている。サービスはアプリケーション ID とサービス ID との組み合わせで一意に識別できる。個々のサービスには ServiceController と Transition と呼ばれるコンポーネントをそれぞれ最大1つまで関連付けることができ、さらに複数の遷移先を関連付けることができる。これらの関係を「図 3-1 サービスの構成」に示す。

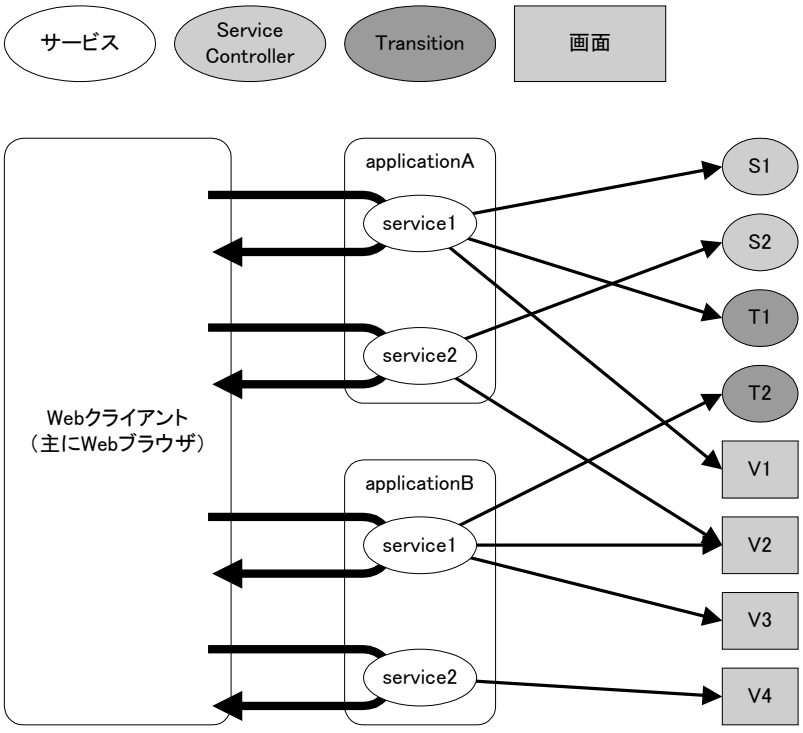


図 3-1 サービスの構成

「図 3-1 サービスの構成」では以下のようなことを示している。

- アプリケーション ID が applicationA、サービス ID が service1 であるサービスは ServiceController として S1、Transition として T1、遷移先画面として V1 と関連付けられている。
- アプリケーション ID が applicationA、サービス ID が service2 であるサービスは ServiceController として S2、遷移先画面として V2 と関連付けられているが、Transition とは関連付けられていない。
- アプリケーション ID が applicationB、サービス ID が service1 であるサービスは Transition として T2、遷移先画面の候補として V2 及び V3 と関連付けられているが ServiceController とは関連付けられていない。
- アプリケーション ID が applicationB、サービス ID が service2 であるサービスは ServiceController とも Transition とも関連付けられてなく、遷移先画面として V4 と関連付けられている。

3.2.1.1 サービス

サービスは IM-JavaEE Framework のサービスフレームワークにおいて最も基本的な単位である。個々のサービスはアプリケーション ID とサービス ID で一意に決定される。

3.2.1.2 ServiceController

ServiceController はリクエストに対する入力チェックと処理を行う役割を持つ。ServiceController の詳細については「3.5.5 処理」で述べている。

3.2.1.3 Transition

Transition は次の画面に遷移するときの準備と実際の遷移を行う役割を持つ。Transition の詳細については「3.5.6 遷移処理」で述べている。

3.2.2 動作

IM-JavaEE Framework のサービスフレームワークではリクエストに対して以下のような順番で各コンポーネントを起動する。

1. リクエストからアプリケーション ID とサービス ID を取得する。
2. アプリケーション ID とサービス ID に該当するサービスを決定する。
3. サービスに ServiceController が関連付けられている場合、その ServiceController の入力処理を起動する。
4. サービスに Transition が関連付けられている場合、その Transition の遷移処理を起動する。

詳細については「3.5 画面遷移」で述べる。

3.3 リクエスト時の前処理

Web クライアント（主に Web ブラウザ）からリクエストが送られたとき、IM-JavaEE Framework ではさまざまな処理に移る前にいくつかの前処理を行う。前処理は Servlet 2.3 仕様のフィルタ機能を利用している。IM-JavaEE Framework の画面に遷移してからの前処理の内容は「図 3-2 リクエスト時の前処理」のようになる。

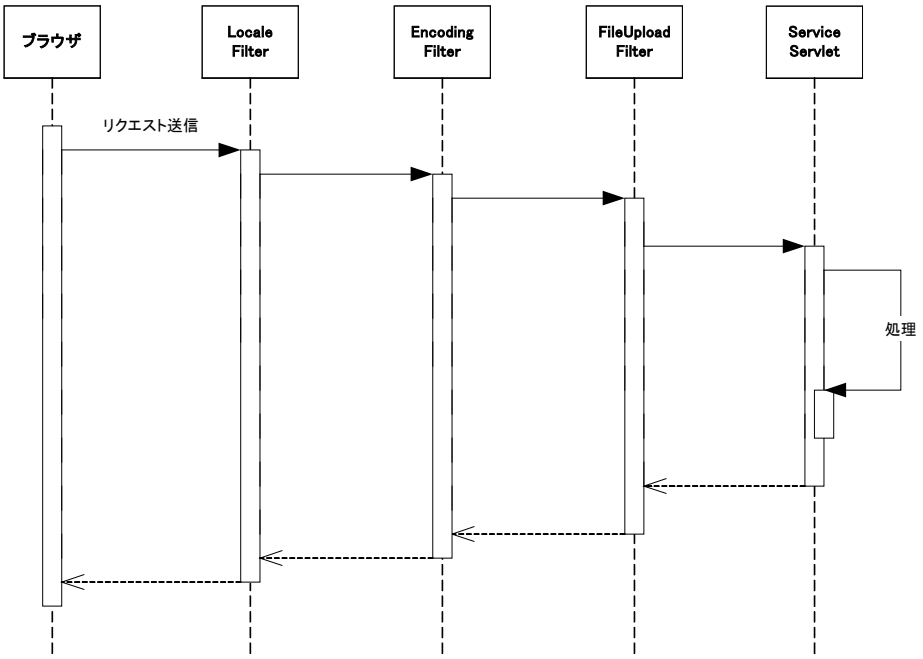


図 3-2 リクエスト時の前処理

3.3.1 ロケール

国際化されたアプリケーションを作成する場合、現在のリクエストがどのロケールに対応するかを決定する必要がある。また、同一ユーザがアプリケーションを使用し続ける場合ロケールは通常変更されることがないため、セッションに登録されていることが便利である場合が多い。

IM-JavaEE Framework ではこれらのことを簡易的に行うためにロケールフィルタを用意している。ロケールフィルタは jp.co.intra_mart.framework.base.service.ServiceManager の getLocale メソッドを通じて現在のロケールを決定および取得し、その内容を javax.servlet.http.HttpSession に登録している。

3.3.1.1 動作

jp.co.intra_mart.framework.base.service.LocaleFilter はリクエストのロケールを決定および取得し、その内容を HttpSession に登録する。このフィルタは「図 3-3 LocaleFilter の処理」に示すような動作をする。

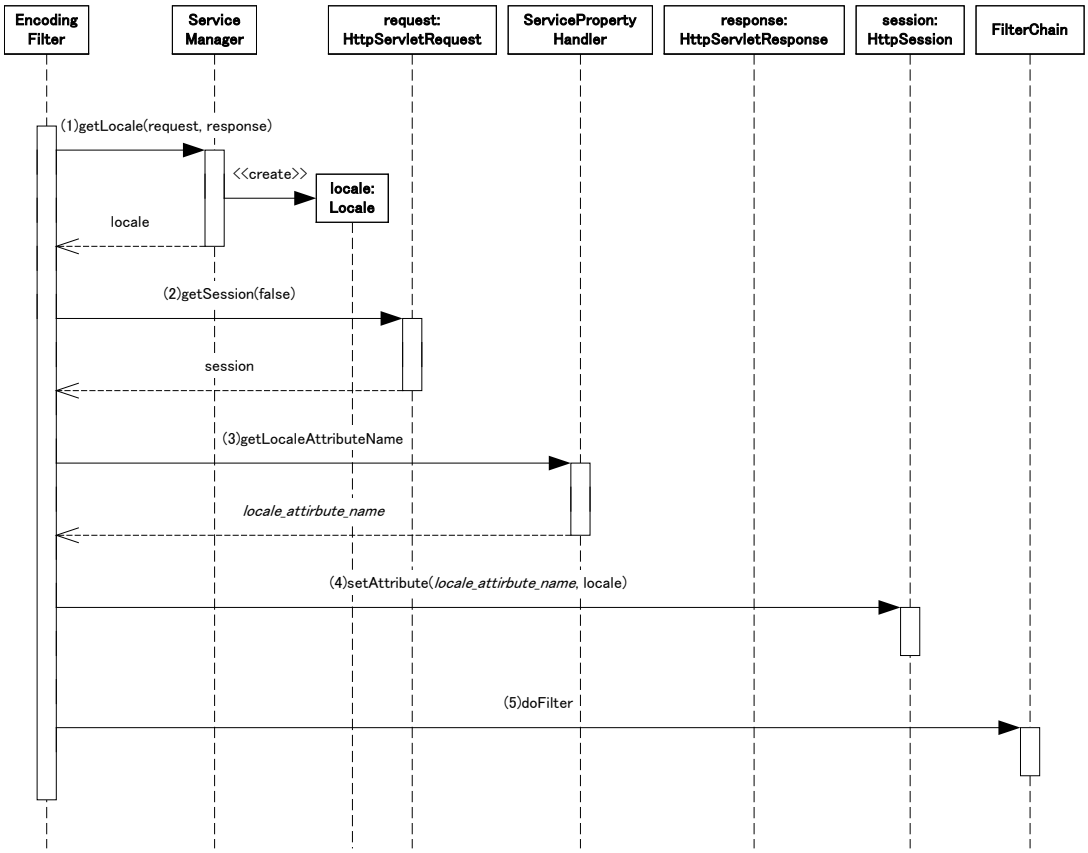


図 3-3 LocaleFilter の処理

ロケールフィルタの処理内容を以下に示す。

1. ServiceManager の getLocale メソッドを利用して現在のリクエストに対するロケールを取得する。
2. HttpSession を取得する。HttpSession が存在しなければ、ロケールに関する処理は以降行わず、次のフィルタに処理を渡す。
3. ServicePropertyHandler を通じて（getLocaleAttributeName メソッドを使用して）クライアントのロケールをセッションに設定するときの属性名を取得する。
4. 取得したロケールを HttpSession に設定する。
5. 次のフィルタに処理を渡す。

3.3.1.2 ロケールの決定方法

ServiceManager の getLocale メソッドでは以下の順番に従ってロケールの取得を試みる。null 以外の値が取得された時点でロケールを決定し、その値を返す。

1. javax.servlet.http.HttpSession に登録されているロケール（「3.3.1.2.1 HttpSession に登録されているロケール」を参照）
2. ServicePropertyHandler から取得されるロケール（「3.3.1.2.2 ServicePropertyHandler から取得されるロケール」を参照）
3. javax.servlet.ServletRequest の getLocale メソッドで取得されるロケール
4. java.util.Locale の getDefault メソッドで取得されるロケール

独自の方法でロケールを決定したい場合、HttpSession にあらかじめロケールを設定すればよい。こうするとロケールフィルタでは(1)の条件に一致するため、以降のアクセスでロケールが決定される。また、アクセスや処理の途中でもこの値を変更すれば、以降の処理では変更されたロケールが使用されることになる。例えばログインユーザごとにロケールを変更したい場合、ログイン処理などで HttpSession にロケールを設定すればよい。また、ログイン後にアクセスしている最中でも、ロケール変更のリクエスト処理などでこの値を変更すれば、以降のアクセスでは新し

いロケールで処理が続行される。

3.3.1.2.1 HttpSession に登録されているロケール

この方法では、HttpSession の属性からロケールの取得を試みる。このときの属性名は ServicePropertyHandler の getLocaleAttributeName メソッド（「3.4.4.1.5 ロケール属性名」参照）で取得されるものである。

3.3.1.2.2 ServicePropertyHandler から取得されるロケール

この方法では、ServicePropertyHandler の getClientLocale メソッド（「3.4.4.1.4 クライアントロケール」参照）を利用してロケールの取得を試みる。

3.3.1.3 ロケールフィルタの適用

ロケールフィルタを利用するためには以下のような設定が必要となる。

- web.xml の設定
- ロケールの設定

3.3.1.3.1 web.xml の設定

ロケールフィルタは Servlet 2.3 の Filter 機能を利用している。このフィルタを有効にするためには web.xml に以下のクラスを Filter として設定する必要がある。

- jp.co.intra_mart.framework.base.service.LocaleFilter

intra-mart をインストールした場合、標準で以下のような設定がされている。

- jp.co.intra_mart.framework.base.service.ServiceServlet にリクエストが渡る前に実行される。

3.3.1.3.2 ロケールの設定

クライアントから送られてくるリクエストのロケールを一意に決定する場合、その設定は「3.4.4.1.4 クライアントロケール」で定義されている項目で行う。設定方法は利用する ServicePropertyHandler に依存するので、関連する資料を参照すること。

3.3.2 エンコーディング

現状の Web ブラウザは送信時のエンコーディングを送らないものが多い。この場合、Servlet ではクライアントからのエンコーディングを ISO-8859-1 として処理するようになっている（「JavaTM Servlet Specification Version 2.3」[5]の「SRV.4.9 Request data encoding」を参照）。そのため javax.servlet.ServletRequest の getParameter メソッド等を利用して送信された内容を取得しようとしてもそのままでは正しい文字列が取得できない場合がある。

IM-JavaEE Framework ではこの問題を解決するためにエンコーディングフィルタを用意している。エンコーディングフィルタは javax.servlet.ServletRequest の setCharacterEncoding メソッドを通じてクライアントのエンコードを設定する。これにより、getParameter メソッド等を使ってもエンコーディングの変換等を行わなくてすむようになる。

3.3.2.1 動作

jp.co.intra_mart.framework.base.service.EncodingFilter はリクエストの内容がどのようなエンコーディングで送信されたかを決定する。このフィルタは「図 3-4 EncodingFilter の処理」に示すような動作をする。

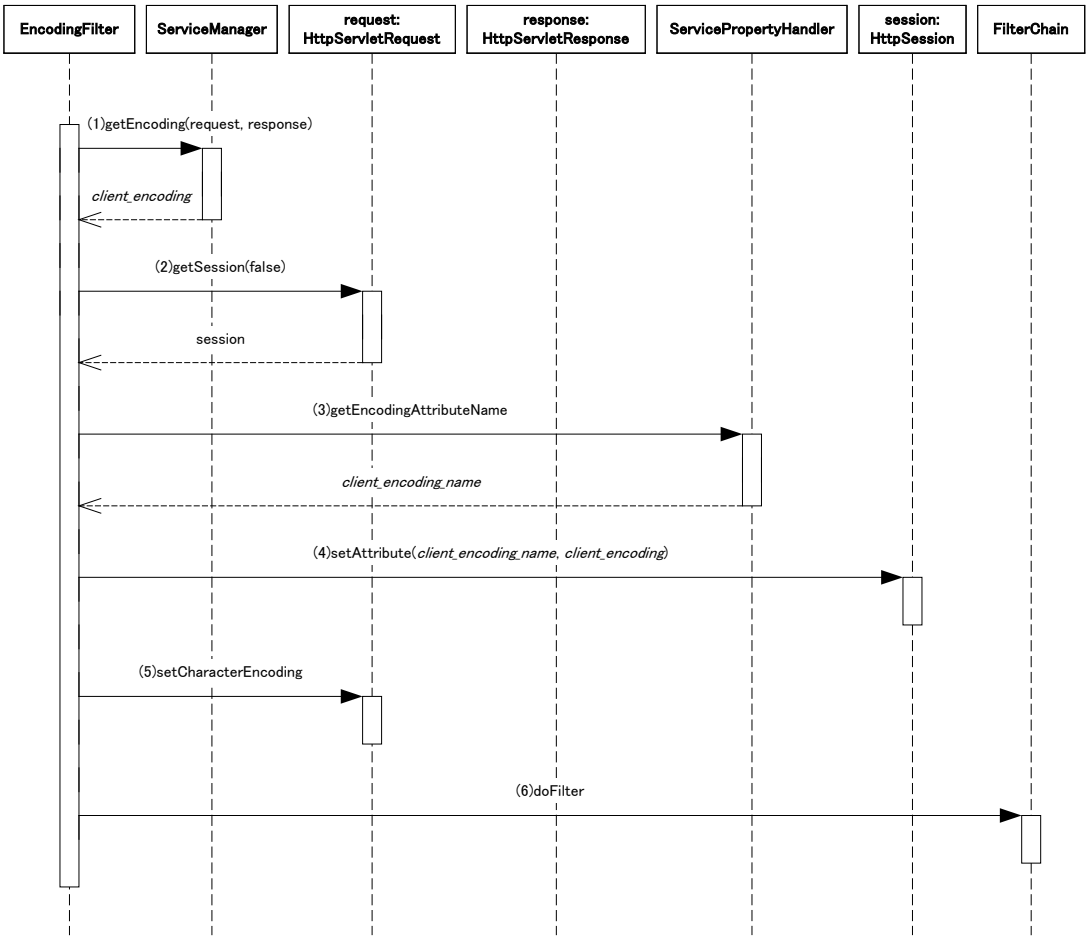


図 3-4 EncodingFilter の処理

エンコーディングフィルタの処理内容を以下に示す。

1. ServiceManager の getEncoding メソッドを利用してリクエストに対するエンコーディングを取得する。
2. HttpSession を取得する。HttpSession が存在しなければ、(3)(4)の処理は行わない。
3. ServicePropertyHandler を通じて（getEncodingAttributeName メソッドを使用して）クライアントのエンコーディングをセッションに設定するときの属性名を取得する。
4. 取得したエンコーディングを HttpSession に設定する。
5. 取得したエンコーディングを ServletRequest に設定する。
6. 次のフィルタに処理を渡す。

3.3.2.2 エンコーディングの決定方法

ServiceManager の getEncoding メソッドでは以下の順番に従ってエンコーディングの取得を試みる。null 以外の値が取得された時点でエンコーディングを決定し、その値を返す。

1. javax.servlet.http.HttpSession に登録されているエンコーディング（「3.3.2.2.1 HttpSession に登録されているエンコーディング」を参照）
2. ServicePropertyHandler から取得されるエンコーディング（「3.3.2.2.2 ServicePropertyHandler から取得されるエンコーディング」を参照）
3. javax.servlet.ServletRequest の getCharacterEncoding メソッドで取得されるロケール

3.3.2.2.1 HttpSession に登録されているエンコーディング

この方法では、HttpSession の属性からエンコーディングの取得を試みる。このときの属性名は ServicePropertyHandler の getEncodingAttributeName メソッド（「3.4.4.1.3 エンコーディング属性名」参照）で取得されるものである。

3.3.2.2.2 ServicePropertyHandler から取得されるエンコーディング

この方法では、ServicePropertyHandler の getClientEncoding メソッド(「3.4.4.1.2 クライアントエンコーディング」参照)を利用してロケールの取得を試みる。

3.3.2.3 エンコーディングフィルタの適用

エンコーディングフィルタを利用するためには以下のような設定が必要となる。

- web.xml の設定
- エンコーディングの設定

3.3.2.3.1 web.xml の設定

エンコーディングフィルタは Servlet 2.3 の Filter 機能を利用している。このフィルタを有効にするためには web.xml に以下のクラスを Filter として設定する必要がある。

- jp.co.intra_mart.framework.base.service.EncodingFilter

intra-mart をインストールした場合、標準で以下のような設定がされている。

- jp.co.intra_mart.framework.base.service.ServiceServlet にリクエストが渡る前に実行される。
- 上記のサーブレットに関連するフィルタの中で最初に実行される [1]。

3.3.2.3.2 エンコーディングの設定

クライアントから送られてくるリクエストのエンコーディングの設定は「3.4.4.1.2 クライアントエンコーディング」で定義されている項目で行う。設定方法は利用する ServicePropertyHandler に依存するので、関連する資料を参照すること。

3.3.3 ファイルアップロード

ブラウザからファイルをアップロードするリクエストが来た場合、ファイルデータを取得する一般的な方法は大体以下のような手順となる:

1. リクエストから入力ストリームを(javax.servlet.ServletRequest の getInputStream 等で)取得する。
2. 取得した入力ストリームからファイルデータを取得する。

しかし、一度リクエストからストリームを取得するとパラメータ取得に関連するメソッド(getParameter 等)が使用できなくなる。入力ストリームの内容を独自に解析してパラメータの情報を取得することは可能だが、それなりの労力を要することになる。

IM-JavaEE Framework ではこの問題を解決するためにファイルアップロードフィルタを用意している。ファイルアップロードフィルタはファイルをアップロードするリクエストが来たとき、通常のリクエストと同様にパラメータを取得するメソッド(getParameter 等)が利用できるのと同時にアップロードされたファイルも取得できる方法を提供する。

3.3.3.1 動作

jp.co.intra_mart.framework.base.service.FileUploadFilter はリクエストがファイルアップロードである場合に jp.co.intra_mart.framework.base.service.ServiceControllerAdapter のサブクラスでパラメータもアップロードされたファイルも容易に取得できる方法を提供する。このフィルタを使うことにより、「図 3-5 FileUploadFilter を利用したときの ServiceControllerAdapter」に示すような方法でアップロードされたファイルの取得を行うことができるようになり、パラメータの取得も通常のリクエストに対する場合と同じ方法(getParameter メソッド等)を利用することができる。

---

[1] ServletRequest の setCharacterEncoding メソッドはリクエストのパラメータや入力内容を読み込む前に呼ばれる必要があるため、できる限り最初のほうに配置している。

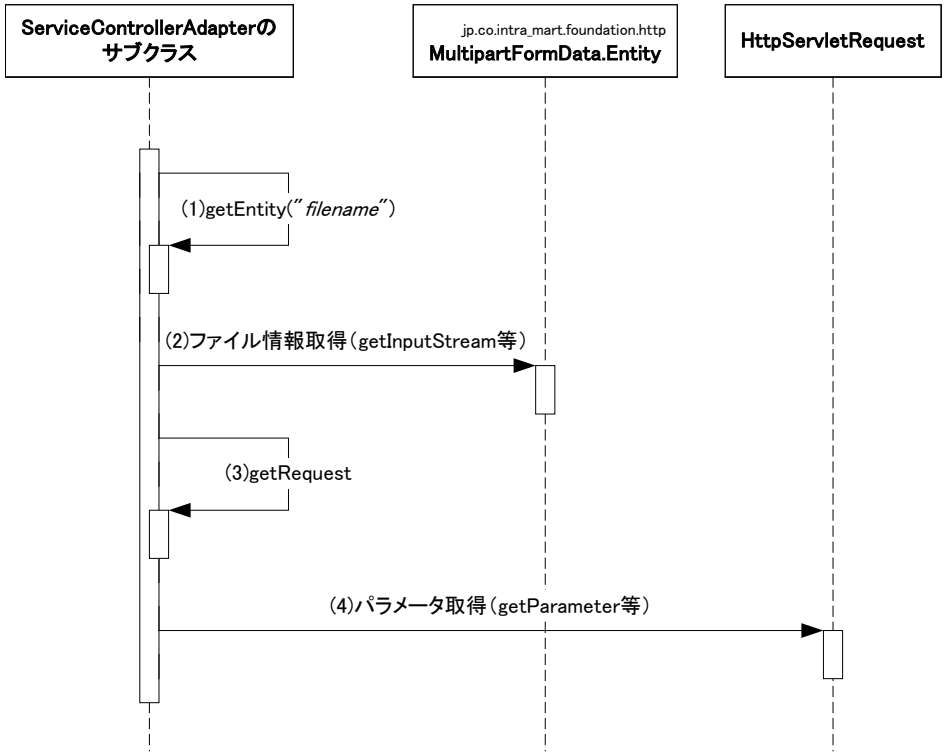


図 3-5 FileUploadFilter を利用したときの ServiceControllerAdapter

ファイルアップロードフィルタを利用した時のファイルの取得ならびにパラメータの取得方法を以下に示す。

1. ServiceControllerAdapter の getEntity メソッドを利用して MultiPartFormData.Entity を取得する。
2. MultiPartFormData.Entity から(getInputStream メソッド等を通じて)ファイルの内容を取得する。
3. ServiceControllerAdapter の getRequest メソッドを利用して HttpServletRequest を取得する。
4. リクエストのパラメータを取得する。

3.3.3.2 ファイルアップロードフィルタの適用

ファイルアップロードフィルタを利用するためには以下のような設定が必要となる。

- ブラウザからのリクエスト
- web.xml の設定

3.3.3.2.1 ブラウザからのリクエスト

ファイルアップロードフィルタは、ブラウザから以下の条件をすべて満たすリクエストに対してのみ適用される。

- リクエストのメソッドが"POST"であること。この場合、大文字、小文字は問わない。
- リクエストの ContentType が"multipart/form-data"であること。

ブラウザから送られてきたリクエストが上記の条件を満たさない場合、ファイルアップロードフィルタは何も行わない。

3.3.3.2.2 web.xml の設定

ファイルアップロードフィルタは Servlet 2.3 の Filter 機能を利用している。このフィルタを有効にするためには web.xml に以下のクラスを Filter として設定する必要がある。

- jp.co.intra_mart.framework.base.service.FileUploadFilter

intra-mart をインストールした場合、標準で以下のような設定がされている。

- jp.co.intra_mart.framework.base.service.ServiceServlet にリクエストが渡る前に実行される。
- エンコーディングフィルタ(「3.3.2 エンコーディング」参照)の次に実行される [2]。

ファイルアップロードフィルタの後でリクエストを入れ替えるようなコーディングは推奨されない。この場合、ServiceControllerAdapter の getEntity メソッドの動作は保障されない。推奨されない例として以下のような場合があげられる。

- リクエスト時に独自の HttpServletRequest を生成して doFilter を実行するフィルタをアップロードフィルタの後で実行されるように追加する。
- アップロードフィルタ実行後に独自の HttpServletRequest を生成して forward する。

3.4 サービスに関連するプロパティ

IM-JavaEE Framework のサービスフレームワークではさまざまなプロパティを外部で設定することが可能である。サービスプロパティの取得は jp.co.intra_mart.framework.base.service.ServicePropertyHandler インタフェースを実装したクラスから取得する。IM-JavaEE Framework ではこのインタフェースを実装した複数の実装クラスを標準で提供している(「図 3-6 ServicePropertyHandler」を参照)。サービスプロパティの設定方法は IM-JavaEE Framework では特に規定してなく、前述のインタフェースを実装したクラスに依存する。

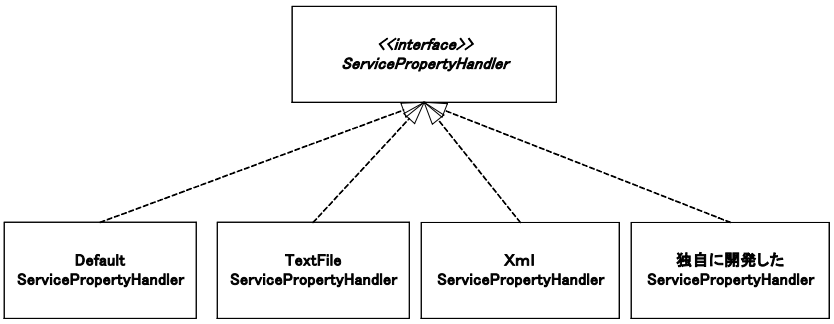


図 3-6 ServicePropertyHandler

3.4.1 サービスに関連するプロパティの取得

サービスに関連するプロパティは ServicePropertyHandler から取得する。ServicePropertyHandler は jp.co.intra_mart.framework.service.ServiceManager の getServicePropertyHandler メソッドで取得することができる。ServicePropertyHandler は必ずこのメソッドを通じて取得されたものである必要があり、開発者が自分でこの ServicePropertyHandler の実装クラスを明示的に生成(new による生成や java.lang.Class の newInstance メソッド、またはリフレクションを利用したインスタンスの生成)をしてはならない。

ServicePropertyHandler の取得とプロパティの取得に関連する手順を「図 3-7 ServicePropertyHandler の取得」に示す。

[2] この位置に配置しない場合、IM-JavaEE Framework の動作は保障されない。

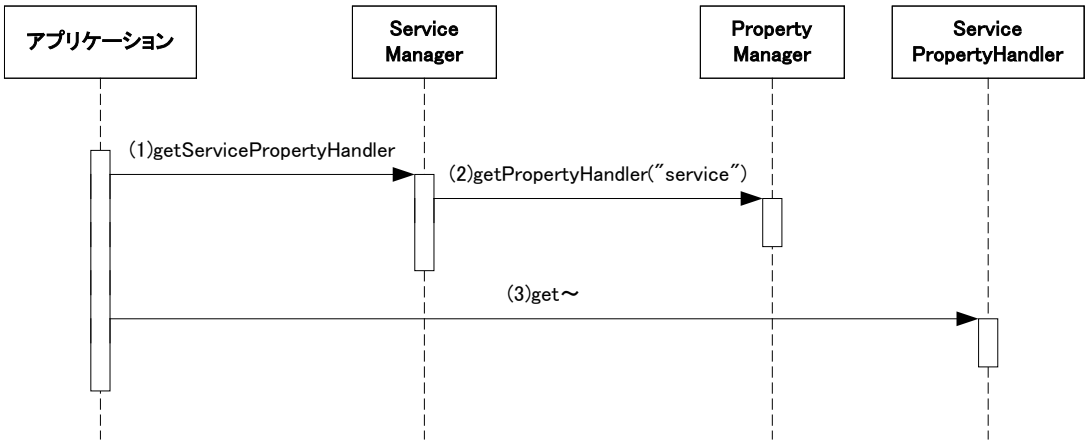


図 3-7 ServicePropertyHandler の取得

1. ServiceManager から ServicePropertyHandler を取得する。
2. ServiceManager の内部では PropertyManager から ServicePropertyHandler を取得し、アプリケーションに返している。この部分は ServiceManager の内部で行っていることであり、開発者は特に意識する必要はない。
3. ServicePropertyHandler を利用して各種のプロパティを取得する。

3.4.2 標準で用意されている ServicePropertyHandler

IM-JavaEE Framework では jp.co.intra_mart.framework.base.service.ServicePropertyHandler を実装したクラスをいくつか提供している。それぞれ設定方法やその特性が違うため、運用者は必要に応じてこれらを切り替えることができる。

3.4.2.1 DefaultServicePropertyHandler

jp.co.intra_mart.framework.base.service.DefaultServicePropertyHandler として提供されている。プロパティの設定はリソースファイルで行う。リソースファイルの内容は「プロパティ名=プロパティの値」という形式で設定する。使用できる文字などは java.util.ResourceBundle に従う。

このリソースファイルは使用するアプリケーションから取得できるクラスパスにおく必要がある。リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

3.4.2.2 TextFileServicePropertyHandler

jp.co.intra_mart.framework.base.service.TextFileServicePropertyHandler として提供されている。DefaultServicePropertyHandler と同じ形式のリソースファイルを利用するが、以下の点が違う。

- クラスパスに通す必要がない。
- アプリケーションから参照できる場所であれば、ファイルシステムの任意の場所に配置できる。
- 設定によってはアプリケーションを停止しないでリソースファイルの再読み込みが可能となる。

リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

3.4.2.3 XmlServicePropertyHandler

jp.co.intra_mart.framework.base.service.XmlServicePropertyHandler として提供されている。プロパティの設定は XML 形式で行う。アプリケーションから取得できるクラスパスにおく必要があり、固有の ID と Java パッケージパスを含めたものをアプリケーション ID として認識する。例えばアプリケーション ID が ”foo.bar.example”の場合、クラスパスに ” foo/bar/service-config-example.xml” として配置する。
また、XmlServicePropertyHandler は動的読み込みに対応している。詳細は API リストを参照。

3.4.3 独自の ServicePropertyHandler

ServicePropertyHandler を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.service.ServicePropertyHandler インタフェースを実装している。
- public なデフォルトコンストラクタ(引数なしのコンストラクタ)が定義されている。
- すべてのメソッドに対して適切な値が返ってくる(「3.4.4 プロパティの内容」参照)。
- isDynamic()メソッドが false を返す場合、プロパティを取得するメソッドはアプリケーションサーバを再起動しない限り値は変わらない。

3.4.4 プロパティの内容

サービスに関連するプロパティの設定方法は運用時に使用する ServicePropertyHandler の種類によって違うが、概念的には同じものである。

また、ほとんどのプロパティは(いくつかの例外を除いて)国際化されており、地域対応させた値を設定することが可能である。

サービスに関連するプロパティの内容は以下のとおりである。

3.4.4.1 共通

3.4.4.1.1 動的読み込み

isDynamic()メソッドで取得可能。地域対応不可。

このメソッドの戻り値が true である場合、このインタフェースで定義される各プロパティ取得メソッド(get～メソッド)は毎回設定情報を読み込みに行くように実装されている必要がある。false である場合、各プロパティ取得メソッドはパフォーマンスを考慮して取得される値を内部でキャッシュしてもよい。

3.4.4.1.2 クライアントエンコーディング

getClientEncoding()メソッドで取得可能。地域対応不可。

ブラウザから送られてくるリクエストの内容のエンコーディングを設定する。省略した場合、リクエストに対する文字コードの変換は行われない。

3.4.4.1.3 エンコーディング属性名

getEncodingAttributeName()メソッドで取得可能。地域対応不可。

ログインユーザが使用するエンコードを HttpSessionn に保存しておくときの属性名を設定する。設定されていない

場合、ServicePropertyHandler.DEFAULT_ENCODING_ATTRIBUTE[3]の値が設定されているものとみなされる。

3.4.4.1.4 クライアントロケール

getClientLocale()メソッドで取得可能。地域対応不可。

クライアントが使用するロケールを設定する。省略した場合、null が設定されたものとみなされる。

3.4.4.1.5 ロケール属性名

getLocaleAttributeName()メソッドで取得可能。地域対応不可。

ログインユーザが使用するロケールを HttpSessionn に保存しておくときの属性名を設定する。設定されていない場合、ServicePropertyHandler.DEFAULT_LOCALE_ATTRIBUTE[4]の値が設定されているものとみなされる。

3.4.4.1.6 コンテキストパス

注意:このプロパティは非推奨であり、設定しないことが望ましい。

getContextPath()メソッドで取得可能。地域対応不可。

IM-JavaEE Framework を動かすときの Web アプリケーションのコンテキストパスを設定する。パスの指定は必ず"/"で開始する必要がある。省略した場合、null が返される。

3.4.4.1.7 例外属性名

getExceptionAttributeName()メソッドで取得可能。地域対応不可。

例外を例外ページに渡すときのリクエストの属性名を設定する。省略した場合、jp.co.intra_mart.framework.base.service.ServicePropertyHandler.DEFAULT_EXCEPTION_ATTRIBUTE の値("exception")を返す。

3.4.4.1.8 デフォルト入力エラーページパス

getInputErrorPagePath()メソッドまたは getInputErrorPagePath(Locale locale)メソッドで取得可能。地域対応可能。

デフォルトの入力エラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、ServicePropertyException が throw される。

3.4.4.1.9 デフォルトサービスエラーページパス

getServiceErrorPagePath()メソッドまたは getServiceErrorPagePath(Locale locale)メソッドで取得可能。地域対応可能。

デフォルトのサービスエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、ServicePropertyException が throw される。

3.4.4.1.10 デフォルトシステムエラーページパス

getSystemErrorPagePath()メソッドまたは getSystemErrorPagePath(Locale locale)メソッドで取得可能。地域対応可能。

デフォルトのシステムエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定

---

[3] IM-JavaEE Framework 5.0 以降ではこの値は"jp.co.intra_mart.framework.base.service.encoding"である。

[4] IM-JavaEE Framework 5.0 以降ではこの値は"jp.co.intra_mart.framework.base.service.locale"である。

は"/"で開始する必要がある。設定されていない場合、ServicePropertyException が throw される。

3.4.4.2 アプリケーション個別

3.4.4.2.1 ServiceController

getServiceControllerName(String, String)メソッドまたは getServiceControllerName(String, String, Loccale)メソッドで取得可能。地域対応可能。

アプリケーション ID とサービス ID に対応する ServiceController のクラス名を設定する。未設定の場合、null を返す。ここで指定するクラスは jp.co.intra_mart.framework.base.service.ServiceController インタフェースを実装している必要がある。

3.4.4.2.2 Transition

getTransitionName(String, String)メソッドまたは getTransitionName(String, String, Locale)メソッドで取得可能。地域対応可能。

アプリケーション ID とサービス ID に対応する Transition のクラス名を設定する。未設定の場合、null を返す。ここで指定するクラスは jp.co.intra_mart.framework.base.service.Transition クラスを拡張している必要がある。

3.4.4.2.3 キー付遷移パス

getNextPagePath(String application, String service, String key)メソッドまたは getNextPagePath(String application, String service, String key, Locale)メソッドで取得可能。地域対応可能。

アプリケーション ID、サービス ID とキーに対応する次の遷移先のパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。遷移先をアプリケーション ID とサービス ID の組み合わせよりも細かく指定したい場合などに使用する。設定されていない場合、ServicePropertyException が throw される。

3.4.4.2.4 遷移パス

getNextPagePath(String application, String service)メソッドまたは getNextPagePath(String application, String service, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID とサービス ID に対応する次の遷移先のパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、ServicePropertyException が throw される。

3.4.4.2.5 キー付入力エラーページパス

getInputErrorPagePath(String application, String service, String key)メソッドまたは getInputErrorPagePath(String application, String service, String key, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID、サービス ID とキーに対応する入力エラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getInputErrorPagePath(String application, String service)を呼んだときと同じ結果となる。入力エラーの遷移先をアプリケーション ID とサービス ID の組み合わせよりも細かく指定したい場合などに使用する。

3.4.4.2.6 入力エラーページパス

getInputErrorPagePath(String application, String service)メソッドまたは getInputErrorPagePath(String application, String service, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID とサービス ID に対応するデフォルトの入力エラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getInputErrorPagePath(String application)を呼んだときと同じ結果となる。

3.4.4.2.7 デフォルト入力エラーページパス

getInputErrorPagePath(String application)メソッドまたは getInputErrorPagePath(String application, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID に対応するデフォルトの入力エラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getInputErrorPagePath()を呼んだときと同じ結果となる。

3.4.4.2.8 キー付サービスエラーページパス

getServiceErrorPagePath(String application, String service, String key)メソッドまたは getServiceErrorPagePath(String application, String service, String key, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID、サービス ID とキーに対応するサービスエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getServiceErrorPagePath(String application, String service)を呼んだときと同じ結果となる。サービスエラーの遷移先をアプリケーション ID とサービス ID の組み合わせよりも細かく指定したい場合などに使用する。

3.4.4.2.9 サービスエラーページパス

getServiceErrorPagePath(String application, String service)メソッドまたは getServiceErrorPagePath(String application, String service, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID とサービス ID に対応するデフォルトのサービスエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getServiceErrorPagePath(String application)を呼んだときと同じ結果となる。

3.4.4.2.10 デフォルトサービスエラーページパス

getServiceErrorPagePath(String application)メソッドまたは getServiceErrorPagePath(String application, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID に対応するデフォルトのサービスエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getServiceErrorPagePath()を呼んだときと同じ結果となる。

3.4.4.2.11 キー付システムエラーページパス

getSystemErrorPagePath(String application, String service, String key)メソッドまたは getSystemErrorPagePath(String application, String service, String key, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID、サービス ID とキーに対応するシステムエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getSystemErrorPagePath(String application, String service)を呼んだときと同じ結果となる。入力エラーの遷移先をアプリケーション ID とサービス ID の組み合わせよりも細かく指定したい場合などに使用する。

3.4.4.2.12 システムエラーページパス

getSystemErrorPagePath(String application, String service)メソッドまたは getSystemErrorPagePath(String application, String service, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID とサービス ID に対応するデフォルトのシステムエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getSystemErrorPagePath(String application)を呼んだときと同じ結果となる。

3.4.4.2.13 デフォルトシステムエラーページパス

getSystemErrorPagePath(String application)メソッドまたは getSystemErrorPagePath(String application, Locale locale)メソッドで取得可能。地域対応可能。

アプリケーション ID に対応するデフォルトのシステムエラーページのパスを設定する。コンテキストルートからの相対パスで指定する。パスの指定は"/"で開始する必要がある。設定されていない場合、getSystemErrorPagePath()を呼んだときと同じ結果となる。

3.5 画面遷移

IM-JavaEE Framework で画面遷移する際の処理概要を「図 3-8 画面遷移の概要」に示す。

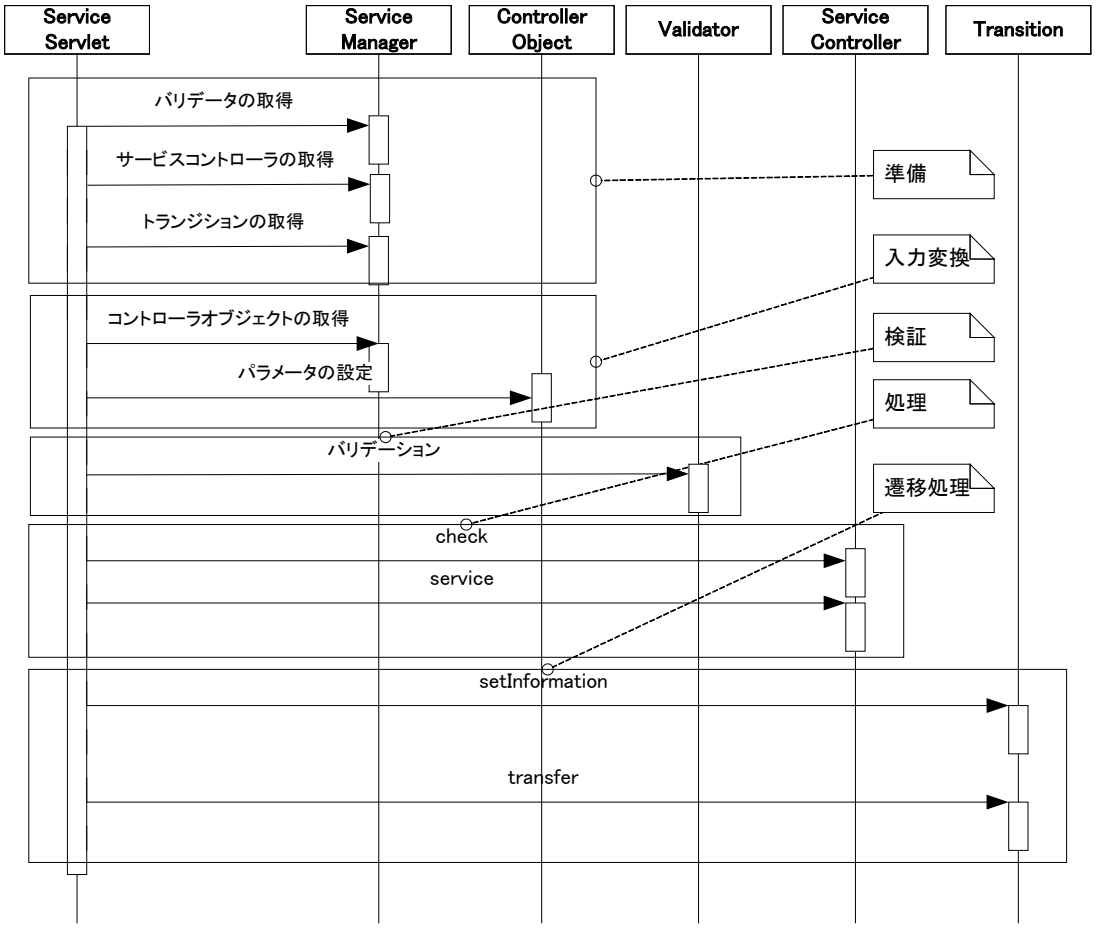


図 3-8 画面遷移の概要

3.5.1 ServiceServlet

「図 3-8 画面遷移の概要」を見ると、リクエストがすべて 1 つのサーブレットで管理されているのがわかる。このサーブレットは jp.co.intra_mart.framework.base.service.ServiceServlet として実装されている。

「準備」の箇所ではリクエストからアプリケーション ID とサービス ID を取得し、該当する ControllerObject、Validator、ServiceController と Transition を取得し、ControllerObject にリクエストパラメータを割り当てる。

「入力処理」ではリクエストの内容を Validator が入力検証を行い、ServiceController の check メソッドでチェックし、service メソッドで具体的な処理を行っている。

「遷移処理」では Transition の setInformation メソッドで次の画面に遷移する準備をし、transfer メソッドで実際に遷移する。

「図 3-8 画面遷移の概要」では画面遷移時の全体の処理概要が示されているが、すべての詳細処理が記述されているわけではない。詳細な内容を以下の順に説明する。

- 準備
- 入力変換
- 検証
- 処理
- 遷移処理

3.5.2 準備

IM-JavaEE Framework では ServiceController で処理を受け付け、Transition で次の画面へ遷移する。ここではリクエストからアプリケーション ID とサービス ID を取得し、該当する ServiceController と Transition を取得している。

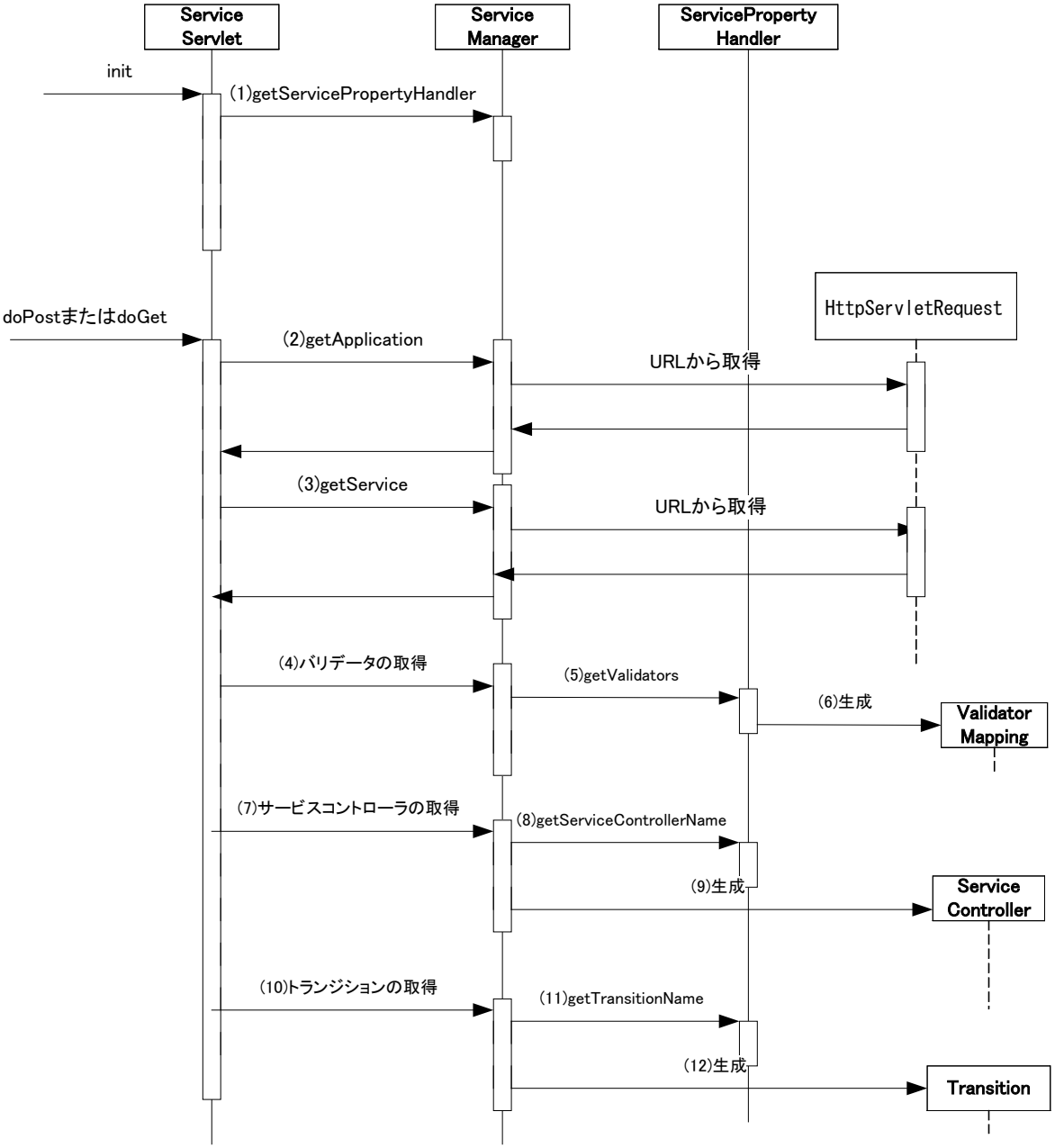


図 3-9 サービスフレームワーク(準備)

3.5.2.1 アプリケーション ID とサービス ID の取得

ServiceServlet は ServiceManager の getApplication メソッドおよび getService メソッドからアプリケーション ID とサービス ID を取得する(「図 3-9 サービスフレームワーク(準備)」の(2)(3))。このとき引数として HttpServletRequest を与える。ServiceManager はリクエストの URL から ID を取得し ServiceServlet に値を返す。

ServiceServlet は HttpServletRequest から直接 ID を取得することが可能だが、ServiceManager がカプセル化をしているためそれを行うべきではない。

現在のバージョンの IM-JavaEE Framework ではアプリケーション ID とサービス ID はリクエストのパラメータとして渡されているが、この方法は今後のバージョンで変更される可能性がある。そのため、開発者はリクエストの中に上記のパラメータ(アプリケーション ID とサービス ID)が存在することを仮定したプログラムを作成するべきではない。

3.5.2.2 Validator の取得

ServiceServlet はアプリケーション ID とサービス ID に該当する Validator を ServiceManager から取得する(「図 3-9 サービスフレームワーク(準備)」の(4))。このとき、ServiceManager の getValidatorMapping メソッドが使われる。このメソッドでは ServicePropertyHandler の getValidators を通じて該当する ValidatorMapping の一覧を取得する(「図 3-9 サービスフレームワーク(準備)」の(6))。

3.5.2.3 ServiceController の取得

ServiceServlet はアプリケーション ID とサービス ID に該当する ServiceController を ServiceManager から取得する(「図 3-9 サービスフレームワーク(準備)」の(7))。このとき、ServiceManager の getServiceController メソッドが使われる。このメソッドでは ServicePropertyHandler の getServiceControllerName を通じて ServiceController のクラス名を取得し(「図 3-9 サービスフレームワーク(準備)」の(12))、該当する ServiceController を新規に生成する(「図 3-9 サービスフレームワーク(準備)」の(9))。

アプリケーション ID とサービス ID に該当する ServiceController のクラス名が指定されていない場合、ServiceManager は null を返す。

3.5.2.4 Transition の取得

ServiceServlet はアプリケーション ID とサービス ID に該当する Transition を ServiceManager から取得する(「図 3-9 サービスフレームワーク(準備)」の(9))。このとき、ServiceManager の getTransition メソッドが使われる。このメソッドでは ServicePropertyHandler の getTransitionName を通じて Transition のクラス名を取得し(「図 3-9 サービスフレームワーク(準備)」の(10))、該当する Transition を新規に生成する(「図 3-9 サービスフレームワーク(準備)」の(15))。

アプリケーション ID とサービス ID に該当する Transition のクラス名が指定されていない場合、ServiceManager は jp.co.intra_mart.framework.base.service.DefaultTransition を新規に生成して返す。

3.5.3 入力変換

Web クライアント(主にブラウザ)からサーバに対してリクエストが送られると、Web コンテナ内のコンポーネント(Servlet、JSP 等)ではその内容が javax.servlet.HttpServletRequest という形で利用することができる。しかしながら、これは開発者にとっては通常は直感的に理解しやすいものではない。これは、HttpServletRequest から情報を得る場合、通常 getParameter メソッドや getParameterNames メソッドなどを使用するが、これらのメソッドを使用するときは次のような点に注意する必要があるためである。

- これらのメソッドはメソッド名から「パラメータを取得する」ことは推測できるが、どのようなパラメータを取得しているのかは引数などで判断する必要がある。
- これらのメソッドは戻り値が java.lang.String 固定であるため、開発者は必要に応じてパラメータを適切なプリミティブ型やオブジェクトに変換する必要がある。
- サービスフレームワークではリクエストの内容を ControllerObject という JavaBeans として表現し、リクエスト

から ControllerObject への変換を ControllerConverter というコンポーネントで行うことができる(「図 3-10 リクエストから ControllerObject への変換」を参照)

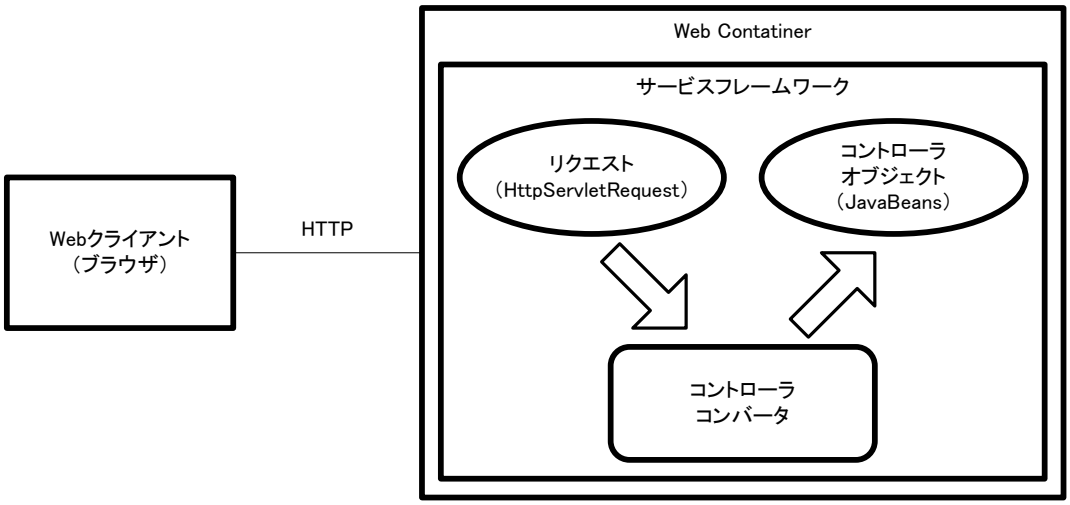


図 3-10 リクエストから ControllerObject への変換

ControllerConverter は 1 つのリクエストに対して 1 つだけ割り当てることが可能である。ControllerConverter の割り当ては必須ではない。

3.5.3.1 ControllerObject

ControllerObject はリクエストの内容を入力情報として変換したものである。ControllerObject は以下の条件を満たしている必要がある。

- JavaBeans としての要件を満たしている。
- jp.co.intra_mart.framework.base.service.controller.ControllerObject インタフェースを実装している。

3.5.3.2 ControllerConverter

ControllerConverter は javax.servlet.HttpServletRequest の内容を ControllerObject に変換するものである。ControllerConverter は以下の条件を満たしている必要がある。

- jp.co.intra_mart.framework.base.service.controller.ControllerConverter インタフェースを実装している。
- public で引数なしのコンストラクタ（デフォルトコンストラクタ）が存在する。
- init メソッド、destroy メソッドが実装されている。
- convert メソッドで適切な ControllerObject が返されるように実装されている。

サービスフレームワークでは ControllerConverter を「図 3-11 ControllerConverter の管理」に示すように管理している。

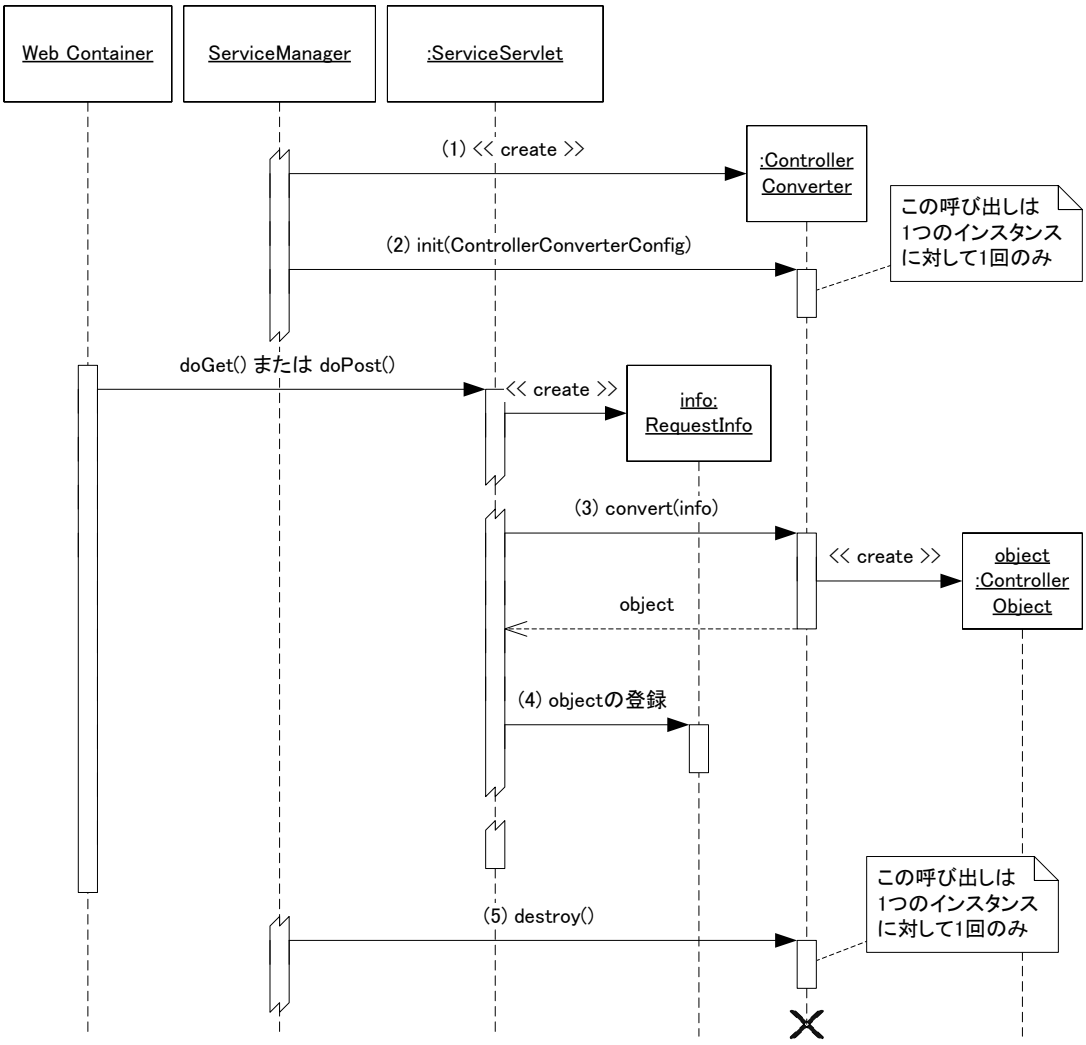


図 3-11 ControllerConverter の管理

1. リクエストが来た時点で ControllerConverter の init メソッドを呼び、ControllerConverter を初期化する。init メソッドは同一インスタンスに対して一度だけ呼ばれる。
2. ControllerConverter の convert メソッドを呼び、ControllerObject を生成する。
3. 生成された ControllerObject を RequestInfo に登録する。ここで登録された ControllerObject は後のフェーズで RequestInfo の getControllerObject メソッドを使用して取得することができる。
4. ControllerObject の生成後、ControllerConverter の destroy メソッドを呼び、ControllerConverter を解放する。destroy メソッドは同一インスタンスに対して一度だけ呼ばれる。

3.5.3.2.1 標準で用意されている ControllerConverter

サービスフレームワークでは基本的な ControllerConverter が標準で用意されている。

3.5.3.2.1.1 SimpleControllerConverter

jp.co.intramart.framework.base.service.controller.SimpleControllerConverter はリクエストのパラメータを指定された ControllerObject に設定する ControllerConverter である。SimpleControllerConverter は以下のようなルールに従ってリクエストに含まれるすべてのパラメータ（javax.serv.et.ServletRequest の getParameterNames メソッドで取得されるパラメータ群）について値を ControllerObject に設定する。

- getParameterValues メソッドの戻り値が大きさ 1 の配列だった場合、ControllerObject の該当するプロパティにそのまま値を設定する。
- getParameterValues メソッドの戻り値が大きさ 2 以上の配列だった場合、ControllerObject の該当するプロパティに配列として値を設定する。配列内の順番は getParameterValues メソッドで取得される配列と同等になる。

例として「リスト 3-1 ControllerObject のデータを含むリクエスト」で示される HTML から送られたリクエストから「リスト 3-2 MyControllerObject.java」で示される ControllerObject を SimpleControllerConverter によって生成する場合を考える。

リスト 3-1 ControllerObject のデータを含むリクエスト

```
・・・
<INPUT type="hidden" name="name" value="foo">
<INPUT type=" hidden" name="password" value="bar">
<INPUT type=" hidden" name="hobby" value="baseball">
<INPUT type=" hidden" name="hobby" value="soccer">
<INPUT type=" hidden" name="hobby" value="reading">
・・・
```

リスト 3-2 MyControllerObject.java

```
・・・
public class MyControllerObject implements ControllerObject {
	private String name = null;
	private String password = null;
	private String[] hobbies = new String[3];

	public String getName() {
		return this.name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getPassword() {
		return this.password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	public String getHobby(int index) {
		return this.hobbies[index];
	}

	public void setHobby(int index, String hobby) {
		this.hobbies[index] = hobby;
	}
}
・・・
```

この場合、リクエストから変換された MyControllerObject の getter メソッドはそれぞれ「表 3-1 設定されている値」に示すような値を返す。

表 3-1 設定されている値

getter メソッド	取得される値
getName()	foo
getPassword()	bar
getHobby(0)	baseball
getHobby(1)	soccer
getHobby(2)	reading

3.5.3.2.2 独自の ControllerConverter

「3.5.3.2.1.1 SimpleControllerConverter」で示した ControllerConverter はリクエストのパラメータ（String）を ControllerObject のプロパティ（String）に設定するだけである。この制約に縛られない ControllerObject（String 以外のオブジェクト、または int などのプリミティブ型をプロパティに持つなど）を生成したい場合、独自に ControllerConverter を作成することもできる。

ControllerConverter が満たすべき条件は「3.5.3.2 ControllerConverter」で示したとおりである。

3.5.3.3 ControllerConverterConfig

ControllerConverter の init メソッドは引数にサービスフレームワークから初期化情報を受け取る。この情報は jp.co.intra_mart.framework.base.service.controller.ControllerConverterConfig インタフェースが実装されたクラスのインスタンスである。ControllerConverterConfig には初期化情報を取得するために次のようなメソッドが用意されて

いる。

- public String getInitParameter(String name)
  指定されたパラメータ名を持つ初期化情報を取得する。
- public Enumeration getInitParameterNames()
  パラメータ名の一覧を取得する。

ControllerConverter の開発者は init メソッド内で ControllerConverterConfig を利用して任意の初期化情報を取得することができる。

3.5.4 検証

Web クライアントからリクエストに対する処理を行う前に、通常はその内容を検証する。Web クライアントとしてブラウザを使用している場合、JavaScript などを使用すればある程度の検証は可能であるが、システム不正データから確実に保護するためにはサーバ側で検証を行うことが必要である。

サービスフレームワークはリクエストの内容を Validator というコンポーネントでリクエストの内容を検証することができる(「図 3-12 リクエストの検証」を参照)。

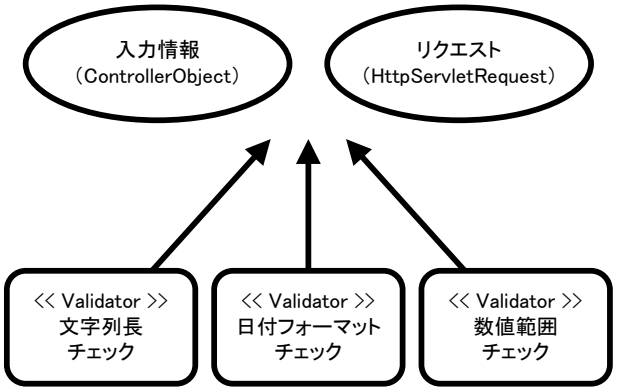


図 3-12 リクエストの検証

Validator は 1 つのリクエストに対して複数割り当てることが可能である。Validator の割り当ては必須ではない。

3.5.4.1 Validator

Validator はリクエストから得られる情報(主に ControllerObject)の内容を検証するものである。Validator は以下の条件を満たしている必要がある。

- jp.co.intra_mart.framework.base.service.validator.Validator インタフェースを実装している。
- public で引数なしのコンストラクタ(デフォルトコンストラクタ)が存在する。
- init メソッド、destroy メソッドが実装されている。
- validate メソッドで適切な ValidationExceptionDetail が返されるように実装されている。validate メソッドは検証結果が正しければ null を、不正であればその詳細情報を含んだ ValidationExceptionDetail を返すように実装されるべきである。

サービスフレームワークデは Validator を「図 3-13 Validator の管理」に示すように管理している。

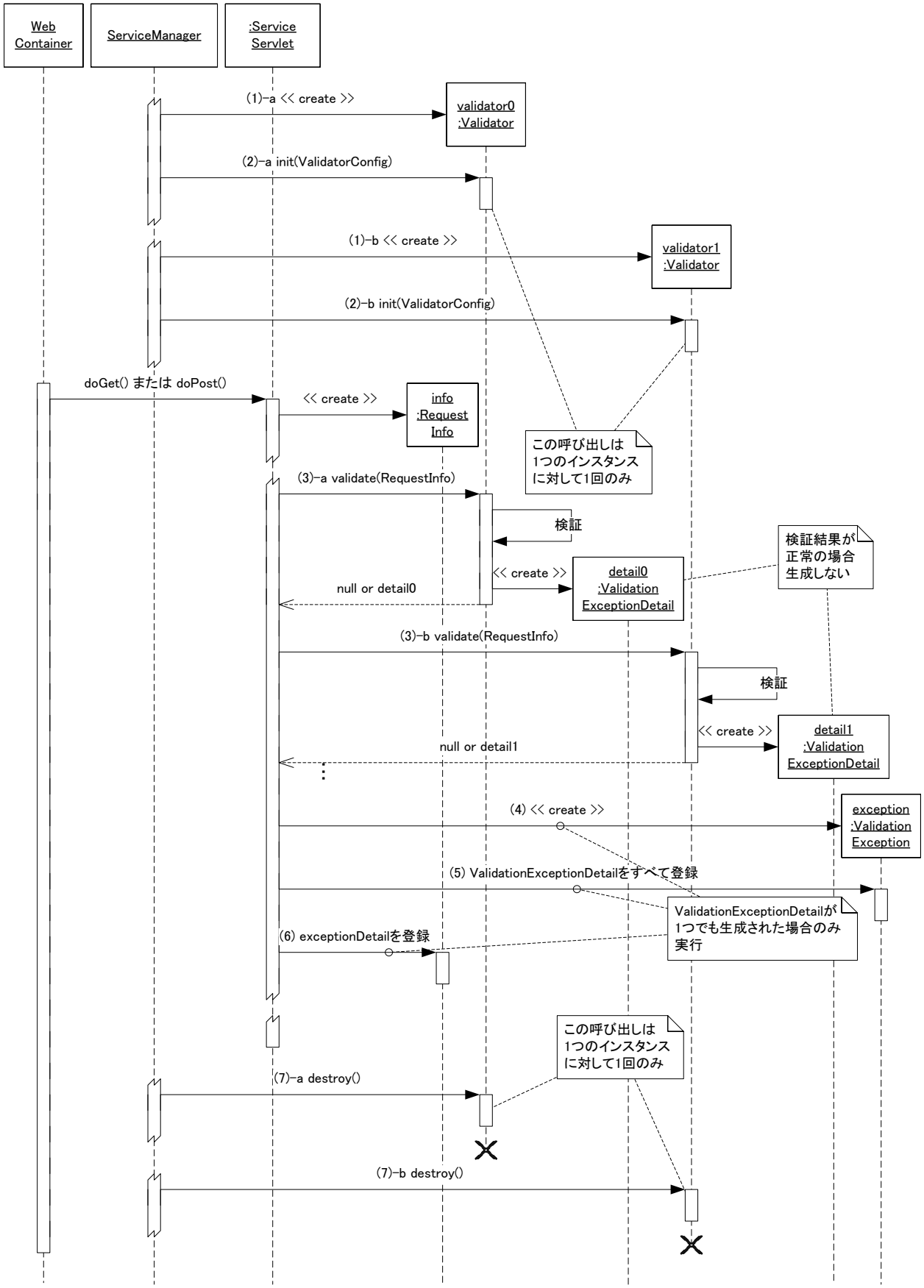


図 3-13 Validator の管理

1. リクエストが来た時点で Validator の init メソッドを呼び、Validator を初期化する。init メソッドは同一インスタンスに対して一度だけ呼ばれる。
2. Validator に対する要求(リクエストの検証)がある場合、該当するすべての Validator の validate メソッドを呼ぶ。1 つのリクエストに対して複数の Validator が該当した場合、設定されている Validator の順番に validate メソッドが呼ばれる。validate メソッドの中ではリクエストの内容が不正であると判断した場合、検証失敗時の詳細情報として ValidationExceptionDetail を生成して返す。リクエストの内容が正常であると判断した場合、null を返す。
3. (2)で該当した Validator のうち 1 つでも validate メソッドが ValidationExceptionDetail を返した場合は ValidationException を生成する。
4. 生成された ValidationException に ValidationExceptionDetail をすべて登録する。
5. (2)で生成され(3)で検証失敗時の詳細情報が設定された ValidationException を RequestInfo に登録する。ここで登録された ValidationException は後のフェーズで RequestInfo の getThrowable メソッドを使用して取得することができる。
6. Validator の destroy メソッドを呼び、Validator を解放する。destroy メソッドは一度だけ呼ばれる。

3.5.4.1.1 標準で用意されている Validator

サービスフレームワークでは基本的な Validator が標準で用意されている。

3.5.4.1.1.1 FormatValidator

jp.co.intra_mart.framework.base.service.validator.FormatValidator は入力された文字列のフォーマットを検証する。このクラスでは ControllerObject を JavaBeans とみなしてプロパティを取得し、正規表現で示されたフォーマットと一致しているかどうか検証する。

設定できるパラメータを「表 3-2 FormatValidator のパラメータ」に示す。

表 3-2 FormatValidator のパラメータ

パラメータ名	意味
property	検証対象となるコントローラオブジェクトのプロパティ名
regex	正規表現によるフォーマット
message	不正文字列である場合のメッセージ

パラメータ property は設定必須のパラメータである。パラメータ property で指定される ControllerObject のプロパティは java.lang.String でなければならない。

パラメータ regex は設定必須のパラメータである。パラメータ regex は java.lang.String の matches メソッドの引数に設定できるものでなければならない。

パラメータ message は設定任意のパラメータである。

例として「リスト 3-3 FormatSampleControllerObject.java」で示される ControllerObject を FormatValidator で評価する場合を考える。

リスト 3-3 FormatSampleControllerObject.java

```
...
public class FormatSampleControllerObject implements ControllerObject {

    public String getLocalPhone() {
        return "01-2345-6789";
    }

    public String getInternationalPhone() {
        return "(+81) 1 2345 6789";
    }
}
```

「リスト 3-3 FormatSampleControllerObject.java」で示される ControllerObject を評価するとき、FormatValidator のパラメータ設定とその検証結果の例を「表 3-3 FormatValidator のパラメータと評価内容」に示す。

表 3-3 FormatValidator のパラメータと評価内容

property	regex	評価内容
localPhone	¥d{2}¥-¥d{4}-¥d{4}	評価:正常 メッセージ:(なし)
	¥d{10}	評価:異常 メッセージ:パラメータ message の値
internationalPhone	¥d{2}¥-¥d{4}-¥d{4}	評価:異常 メッセージ:パラメータ message の値
	¥(¥+¥d{2}¥) ¥d ¥d{4} ¥d{4}	評価:正常 メッセージ:(なし)
(設定なし)	aaaa	初期化時に例外 (パラメータ property は設定必須)
localPhone	(設定なし)	初期化時に例外 (パラメータ regex は設定必須)
other	aaaa	検証時に例外 (FormatSampleControllerObject にプロパティ other は存在しない)

パラメータ message が省略され、かつ文字列がフォーマットに一致しない場合、システムで用意されたメッセージを含む ValidationExceptionDetail が返される。

3.5.4.1.1.2 LengthValidator

jp.co.intra_mart.framework.base.service.validator.LengthValidator は入力された文字列の長さを検証する。このクラスでは ControllerObject を JavaBeans とみなしてプロパティ(文字列)を取得し、その文字列の長さが指定された範囲内であるかどうか検証する。

設定できるパラメータを「表 3-4 LengthValidator のパラメータ」に示す。

表 3-4 LengthValidator のパラメータ

パラメータ名	意味
property	検証対象となるコントローラオブジェクトのプロパティ名
min	文字列の最小長
max	文字列の最大長
message_short	文字列が最小長よりも短い場合のメッセージ
message_long	文字列が最大長よりも長い場合のメッセージ

パラメータ property は設定必須のパラメータである。パラメータ property で指定される ControllerObject のプロパティは java.lang.String でなければならない。

パラメータ min は設定任意のパラメータである。パラメータ min は 0 以上かつ int の最大値（java.lang.Integer.MAX_VALUE）以下でなければならない。パラメータ min が省略された場合、0 が設定されたものとみなされる。

パラメータ max は設定任意のパラメータである。パラメータ max は 0 以上かつ int の最大値（java.lang.Integer.MAX_VALUE）以下でなければならない。パラメータ max が省略された場合、int の最大値が設定されたものとみなされる。

パラメータ message_short は設定任意のパラメータである。

パラメータ message_long は設定任意のパラメータである。

例として「リスト 3-4 LengthSampleControllerObject.java」で示される ControllerObject を LengthValidator で評価する場合を考える。

リスト 3-4 LengthSampleControllerObject.java

```
...
public class LengthSampleControllerObject implements ControllerObject {

    public String getPassword() {
        return "mypassword";
    }
}
```

「リスト 3-3 FormatSampleControllerObject.java」で示される ControllerObject を評価するとき、FormatValidator のパラメータ設定とその検証結果の例を「表 3-5 LengthValidator のパラメータと評価内容」に示す。

表 3-5 LengthValidator のパラメータと評価内容

property	min	max	評価内容
password	8	20	評価:正常 メッセージ:(なし)
	4	8	評価:異常 メッセージ:パラメータ message_long の値
	12	20	評価:異常 メッセージ:パラメータ message_short の値
	10	10	評価:正常 メッセージ:(なし)
	(設定なし)	20	評価:正常 メッセージ:(なし)
	8	(設定なし)	評価:正常 メッセージ:(なし)
	(設定なし)	(設定なし)	評価:正常 メッセージ:(なし)
(設定なし)	8	20	初期化時に例外 (パラメータ property は設定必須)
password	20	8	初期化時に例外 (パラメータ min はパラメータ max 以下でなければならない)
other	8	20	検証時に例外 (LengthSampleControllerObject にプロパティ other は存在しない)

パラメータ message_short が省略され、かつ文字列長が最小値より短い場合、システムで用意されたメッセージを含む ValidationExceptionDetail が返される。パラメータ message_long が省略され、かつ文字列長が最大値より長い場合も同様である。

3.5.4.1.1.3 NumericValidator

jp.co.intra_mart.framework.base.service.validator.NumericValidator は入力された値が整数として妥当であるかどうかを検証する。このクラスでは ControllerObject を JavaBeans とみなしてプロパティを取得し、数値として妥当かどうか検証する。

設定できるパラメータを「表 3-6 NumericValidator のパラメータ」に示す。

表 3-6 NumericValidator のパラメータ

パラメータ名	意味
property	検証対象となるコントローラオブジェクトのプロパティ名
message	整数として妥当ではない場合のメッセージ

パラメータ property は設定必須のパラメータである。

パラメータ message は設定任意のパラメータである。

例として「リスト 3-5 NumericSampleControllerObject.java」で示される ControllerObject を NumericValidator で評価する場合を考える。

リスト 3-5 NumericSampleControllerObject.java

```
・・・
public class NumericSampleControllerObject implements ControllerObject {

    public int getIntNum() {
        return ・・・;
    }

    public long getLongNum() {
        return ・・・;
    }

    public String getPositive () {
        return "123456789";
    }

    public String getNegative() {
        return "-123456789";
    }

    public String getNotNum() {
        return "abcd";
    }
}
```

「リスト 3-5 NumericSampleControllerObject.java」で示される ControllerObject を評価するとき、NumericValidator のパラメータ設定とその検証結果の例を「表 3-7 NumericValidator のパラメータと評価内容」に示す。

表 3-7 NumericValidator のパラメータと評価内容

property	評価内容
intNum	評価:正常 メッセージ:(なし)
longNum	評価:正常 メッセージ:(なし)
positive	評価:正常 メッセージ:(なし)
negative	評価:正常 メッセージ:(なし)
notNum	評価:異常 メッセージ:パラメータ message の値
(設定なし)	初期化時に例外 (パラメータ property は設定必須)
other	検証時に例外 (FormatSampleControllerObject にプロパティ other は存在しない)

パラメータ message が省略され、かつプロパティが整数として妥当ではない場合、システムで用意されたメッセージを含む ValidationExceptionDetail が返される。

3.5.4.1.1.4 NumericRangeValidator

jp.co.intra_mart.framework.base.service.validator.NumericRangeValidator は入力された数値の大きさを検証する。このクラスでは ControllerObject を JavaBeans とみなしてプロパティ(数値)を取得し、その数値が指定された範囲内であるかどうか検証する。

設定できるパラメータを「表 3-8 NumericRangeValidator のパラメータ」に示す。

表 3-8 NumericRangeValidator のパラメータ

パラメータ名	意味
property	検証対象となるコントローラオブジェクトのプロパティ名
min	数値の最小値
max	数値の最大値
message	数値が指定された範囲外である場合のメッセージ

パラメータ property は設定必須のパラメータである。パラメータ property で指定される ControllerObject のプロパティは整数として妥当なものでなければならない。

パラメータ min は設定任意のパラメータである。パラメータ min が省略された場合、下限がないものとみなされる。

パラメータ max は設定任意のパラメータである。パラメータ max が省略された場合、上限がないものとみなされる。

パラメータ message は設定任意のパラメータである。

例として「リスト 3-6 RangeSampleControllerObject.java」で示される ControllerObject を RangeValidator で評価する場合を考える。

リスト 3-6 RangeSampleControllerObject.java

```
...
public class RangeSampleControllerObject implements ControllerObject {

    public int getIntNum() {
        return 500;
    }

    public long getLongNum() {
        return -1234L;
    }

    public String getStringNum() {
        return "123456";
    }

    public String getIllegalNum() {
        return "abcd";
    }
}
```

「リスト 3-6 RangeSampleControllerObject.java」で示される ControllerObject を評価するとき、RangeValidator のパラメータ設定とその検証結果の例を「表 3-9 NumericRangeValidator のパラメータと評価内容」に示す。

表 3-9 NumericRangeValidator のパラメータと評価内容

property	min	max	評価内容
intNum	100	1000	評価:正常 メッセージ:(なし)
	600	(設定なし)	評価:異常 メッセージ:パラメータ message の値
intLong	(設定なし)	-1000	評価:正常 メッセージ:(なし)
	(設定なし)	(設定なし)	評価:正常 メッセージ:(なし)
stringNum	-100000	200000	評価:正常 メッセージ:(なし)
illegalNum	10	100	評価:異常 メッセージ:パラメータ message の値
stringNum	200000	-100000	初期化時に例外 (パラメータ min はパラメータ max 以下でなければならない)
(設定なし)	10	100	初期化時に例外 (パラメータ property は設定必須)
other	10	100	検証時に例外 (LengthSampleControllerObject にプロパティ other は存在しない)

パラメータ message が省略され、かつ数値が指定された範囲外である場合、システムで用意されたメッセージを含む ValidationExceptionDetail が返される。

3.5.4.1.2 独自の Validator

「3.5.4.1.1.1 FormatValidator」から「3.5.4.1.1.4 NumericRangeValidator」で示した Validator は ControllerObject のプロパティを検証するだけである。この制約に縛られない Validator(複数プロパティを同時に検証、または HttpSession の内容と比較など)を生成したい場合、独自に Validator を作成することもできる。

Validator が満たすべき条件は「3.5.4.1 Validator」で示したとおりである。

3.5.4.2 ValidatorConfig

Validator の init メソッドは引数にサービスフレームワークから初期化情報を受け取る。この情報は jp.co.intra_mart.framework.base.service.validator.ValidatorConfig インタフェースが実装されたクラスのインスタンスである。ValidatorConfig には初期化情報を取得するために次のようなメソッドが用意されている。

- public String getInitParameter(String name)
  指定されたパラメータ名を持つ初期化情報を取得する。
- public Enumeration getInitParameterNames()
  パラメータ名の一覧を取得する。

Validator の開発者は init メソッド内で ValidatorConfig を利用して任意の初期化情報を取得することができる。

3.5.5 処理

リクエストに対する処理は IM-JavaEE Framework では ControllerObject、Validator、ServiceController がその役目を担っている。ServiceController の役割は以下のとおりである:

- リクエスト内容のチェック
- リクエストに対する処理

ServiceController のクラス構成は「図 3-14 ServiceController の構成」のようになっている。この中で「～ServiceController」「～ServiceResult」と記述してある箇所は開発者が実装するクラスである。

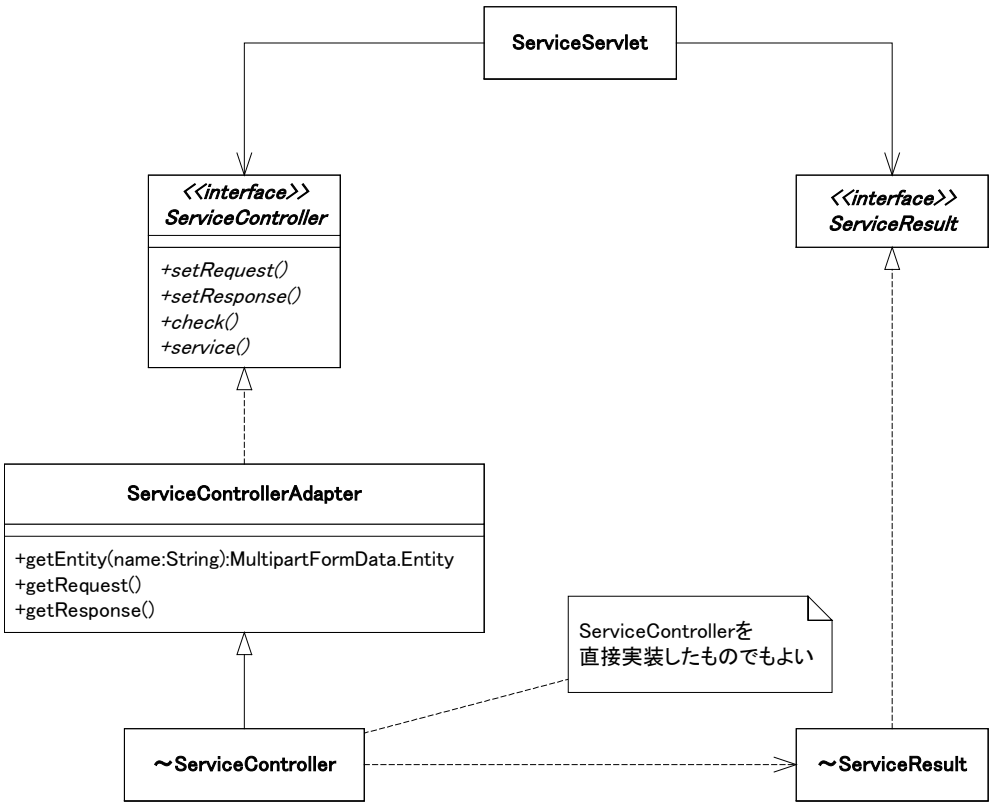


図 3-14 ServiceController の構成

ServiceServlet はリクエストを受け取ると該当する ServiceController が設定されているかどうかチェックする。ServiceController が設定されている場合、その ServiceController のメソッドを「図 3-15 サービスフレームワーク（入力処理）」のように呼び出す。

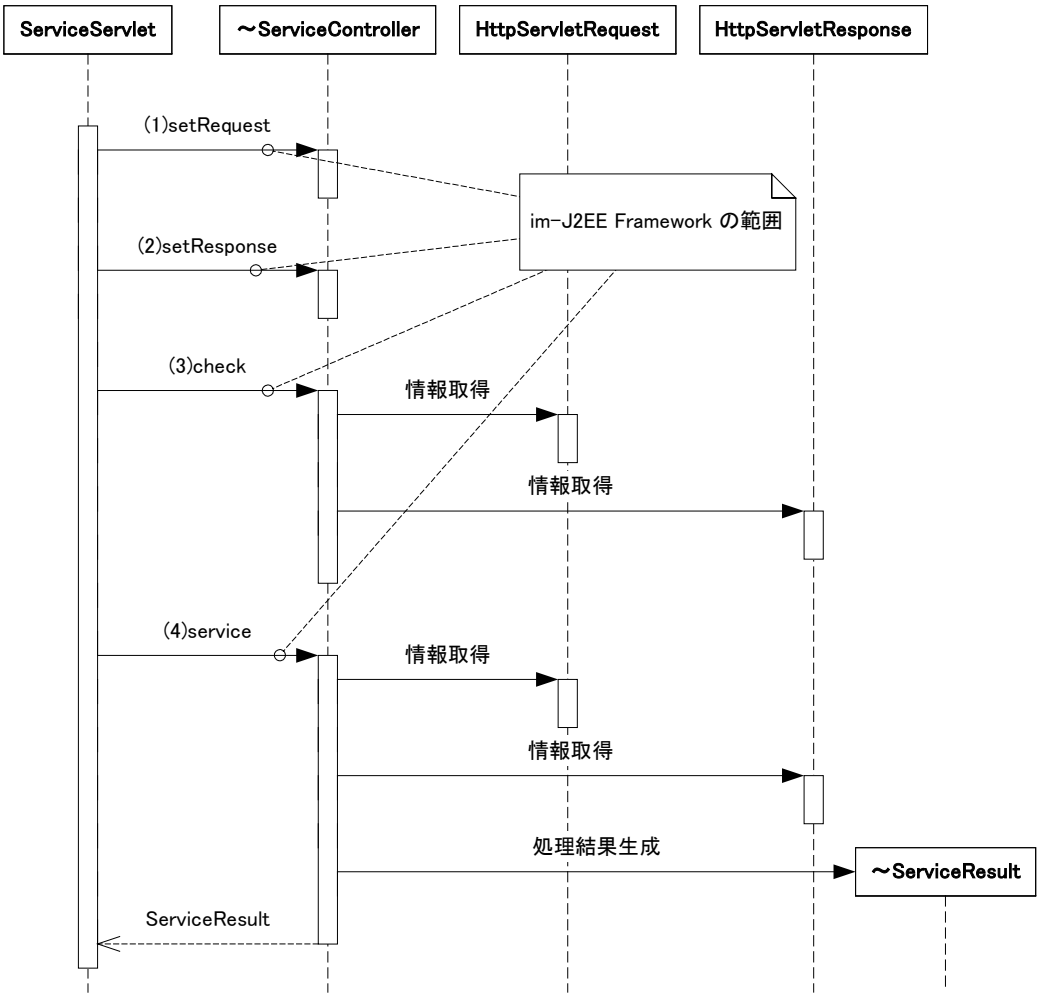


図 3-15 サービスフレームワーク(入力処理)

1. setRequest(javax.servlet.HttpServletRequest request)メソッドを呼び出し、リクエストをServiceControllerに渡す。
2. setResponse(javax.servlet.HttpServletResponse response)メソッドを呼び出し、レスポンスをServiceController に渡す。
3. check()メソッドを呼び出し、入力チェックを行う。このとき、ServiceController 内で先に設定されたHttpServletRequest や HttpServletResponse を取得することが可能であればその内容を利用することができる。
4. service()メソッドを呼び出し、処理結果である ServiceResult を取得する。このとき、ServiceController 内では先に設定された HttpServletRequest や HttpServletResponse を取得することが可能であればその内容を利用することができる。

3.5.5.1 ServiceControllerAdapter

jp.co.intra_mart.framework.base.service.ServiceController はインタフェースであるため、開発者がこのインタフェースを利用して ServiceController を作成する場合はすべてのメソッドを実装しなくてはならない。これは check メソッドや service メソッドでリクエストやレスポンスを利用する場合、setRequest メソッドや setResponse メソッドでリクエストやレスポンスをインスタンス変数などに設定する必要があることを意味し、リクエストやレスポンスが特に必要なかったり、入力チェックを行わない場合でも各種メソッド(setRequest、setResponse および check)を空の状態で実装する必要があることを意味する。

これらの煩わしさを解消するためには、開発者は ServiceController を作成する場合、ServiceController インタフェースではなく、jp.co.intra_mart.framework.base.service.ServiceControllerAdapter クラスを継承する方法を推奨する。このクラスを継承して ServiceController を作成した場合、以下の利点がある。

- 必要以上にメソッドを実装なくてもよい。
- setRequest や setResponse が既に実装されており、設定されたリクエストやレスポンスを取得するメソッド（getRequest、getResponse）が用意されている。
- リクエストがファイルアップロードの場合、その内容を取得するメソッド（getEntity）が用意されている。
- 「4 イベントフレームワーク」で紹介されているイベントフレームワークに関連する各種ユーティリティメソッド（createEvent や dispatchEvent 等）が揃っている。

3.5.5.2 入力チェック

IM-JavaEE Framework では入力チェックとして check メソッドを用意している。このメソッドの実装は開発者に委ねられる。開発者はこのメソッド内でリクエストの内容をチェックすることが望ましい。

入力内容が不正である場合、このメソッドは jp.co.intra_mart.framework.base.service.RequestException またはそのサブクラスを throw するような実装が望ましい。詳細は「3.7.1 入力時の例外処理」を参照。

3.5.5.3 処理

check メソッドでリクエストの検証に成功した場合、IM-JavaEE Framework では引き続いて service メソッドを呼び出す。開発者はこのメソッド内でリクエストの内容を元に処理を依頼するよう実装する。ここで「処理を依頼」と書いたのは、詳細なビジネスロジックはこのメソッドの中で記述するべきではなく、外部（IM-JavaEE Framework のイベントフレームワークなど）に出しておいたほうがメンテナンス性、汎用性が増すからである。service メソッド内では実際に処理を行うのではなく、あくまで窓口として実装することが推奨される。

外部のビジネスロジックを利用する方法として以下のものがあげられる。

- ServiceControllerAdapter のメソッドからイベントフレームワークを利用
- IM-JavaEE Framework のイベントフレームワークを直接利用

3.5.5.3.1 ServiceControllerAdapter のメソッドからイベントフレームワークを利用

ServiceControllerAdapter のメソッドを利用してイベントフレームワークに処理を依頼する方法は最も簡単な実装方法である。開発者は「図 3-16 ServiceControllerAdapter からイベントフレームワークを利用」に示したようにしてイベントフレームワークを利用することができる。

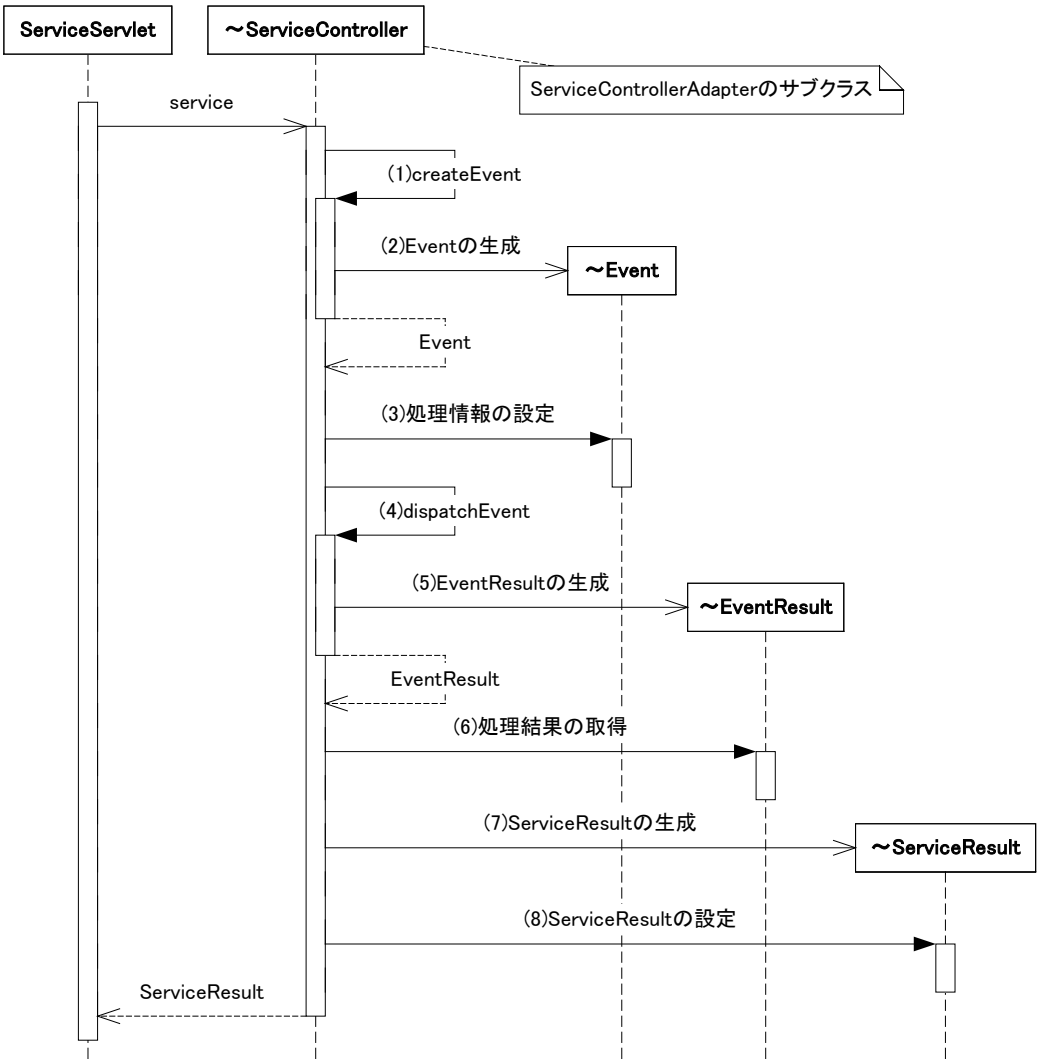


図 3-16 ServiceControllerAdapter からイベントフレームワークを利用

1. createEvent メソッドを呼び出し、Event を取得する。
2. ServiceControllerAdapter はイベントフレームワークを利用して該当する Event を取得する（EventManager.createEvent メソッドを参照）。このとき、セッションからログインユーザ ID とログイングループ ID を取得して Event の生成に使用する。
3. 取得した Event に処理を行うための情報を設定する。
4. dispatchEvent メソッドを呼び出し、処理を行う（EventManager.dispatch メソッドを参照）。
5. ServiceControllerAdapter はイベントフレームワークを利用して処理を実行し、処理結果である EventResult を返す。
6. EventResult から処理結果の情報を取得する。
7. ServiceController の戻り値となる ServiceResult を生成する。
8. ServiceResult に結果を設定する。

これらのうち、「(2)Event の生成」と「(5)EventResult の生成」は既に ServiceControllerAdapter で実装されているため、開発者はこの箇所を実装する必要はない。

3.5.5.3.2 IM-JavaEE Framework のイベントフレームワークを直接利用

ServiceController インタフェースを直接実装する場合、イベントを実行するときは IM-JavaEE Framework のイベントフレームワークを直接利用する必要がある。ここでは「図 3-17 IM-JavaEE Framework のイベントフレームワークを直接利用」にサンプルとして ServiceControllerAdapter の内部でイベントフレームワークを利用している様子を擬似的に示す。実際の ServiceControllerAdapter では、ログインユーザ ID とログイングループ ID の取得が複雑であ

る。

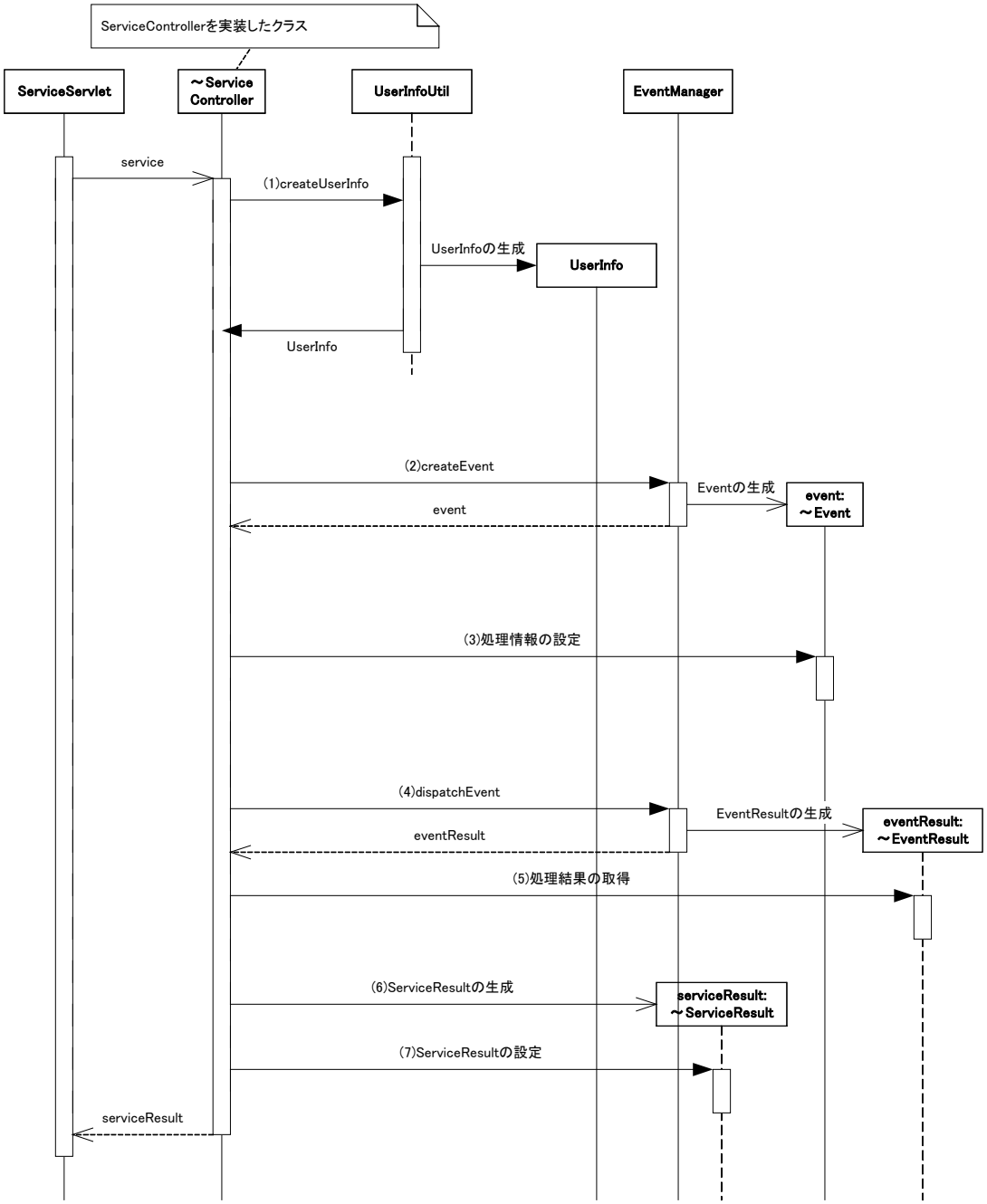


図 3-17 IM-JavaEE Framework のイベントフレームワークを直接利用

1. UserInfoUtil のメソッドから UserInfo のインスタンスを生成する。
2. EventManager.createEvent メソッドを呼び出し、Event を取得する。
3. 取得した Event に処理を行うための情報を設定する。
4. EventManager.dispatch メソッドを呼び出し、処理を行う。
5. EventResult から処理結果の情報を取得する。
6. ServiceController の戻り値となる ServiceResult を生成する。
7. ServiceResult に結果を設定する。

3.5.6 遷移処理

リクエストに対する遷移処理は IM-JavaEE Framework では Transition がその役目を担っている。Transition の役割は以下のとおりである:

- 遷移先に対する準備
- リクエストに対する処理

ServiceController のクラス構成は「図 3-18 Transition の構成」のようになっている。この中で「～Transition」「～ServiceResult」と記述してある箇所は開発者が実装するクラスである。

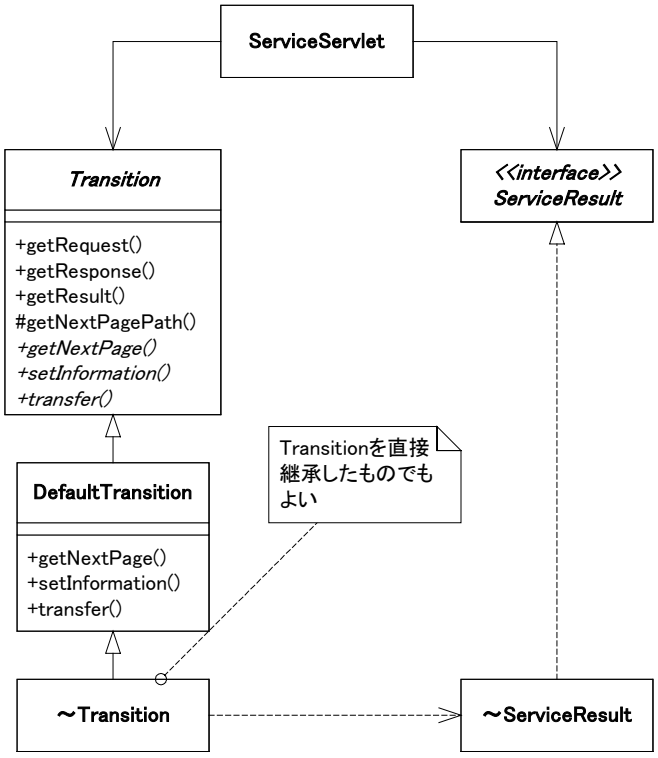


図 3-18 Transition の構成

ServiceServlet はリクエストを受け取ると該当する Transition が設定されているかどうかチェックする。Transition が設定されている場合、その Transition のメソッドを「図 3-19 サービスフレームワーク(画面遷移)」のように呼び出す。

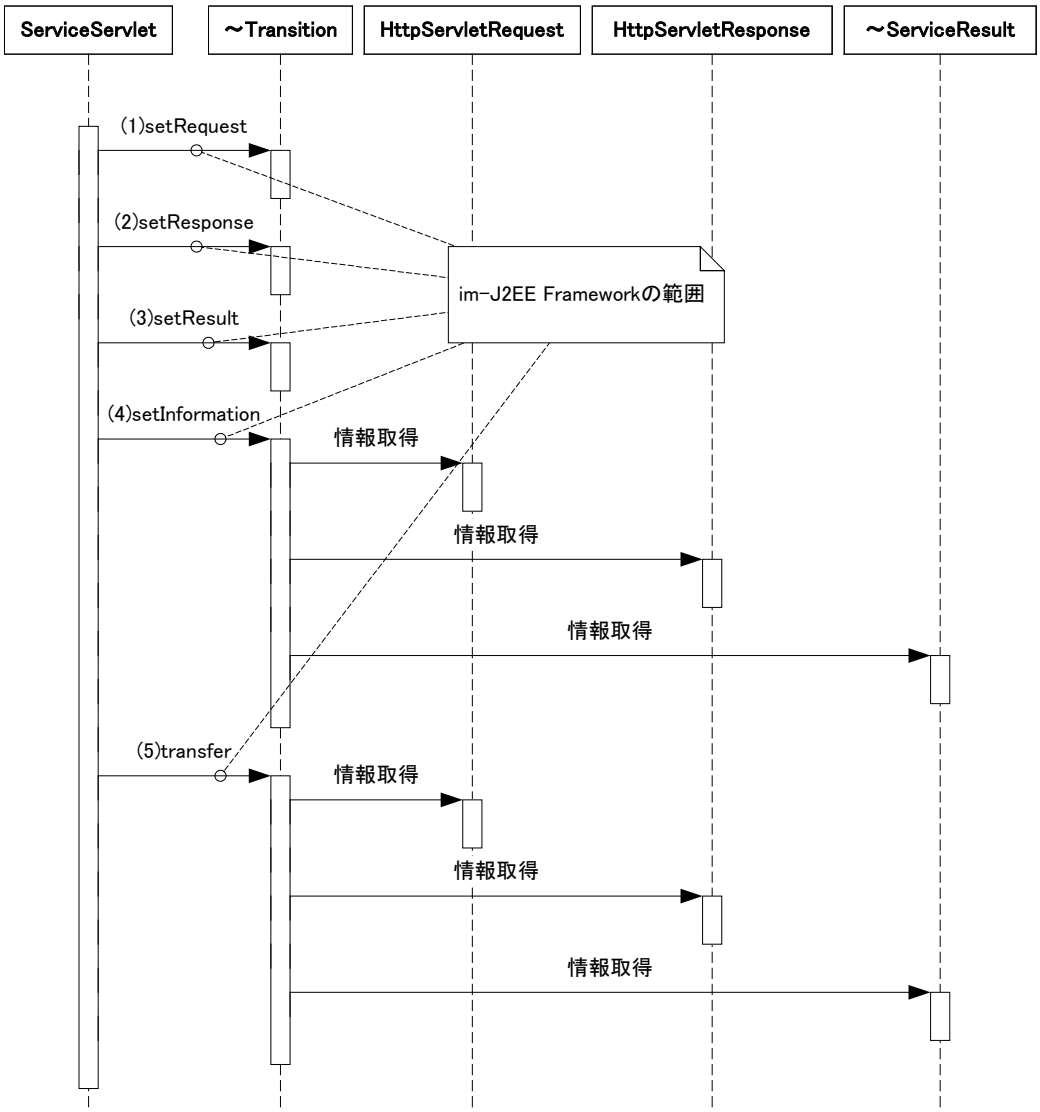


図 3-19 サービスフレームワーク(画面遷移)

1. setRequest(javax.servlet.http.HttpServletRequest request)メソッドを呼び出し、リクエストを Transition に渡す。
2. setResponse(javax.servlet.http.HttpServletResponse response)メソッドを呼び出し、レスポンスを Transition に渡す。
3. setResult(jp.co.intra_mart.framework.base.service.ServiceResult result)メソッドを呼び出し、サービス処理結果を Transition に渡す。ServiceController が指定されていない場合、null が渡される。
4. setInformation()メソッドを呼び出し、次の画面に遷移する準備を行う。このとき、Transition 内で先に設定された HttpServletRequest、HttpServletResponse または ServiceResult を取得することが可能であればその内容を利用することができる。
5. transfer()メソッドを呼び出し、次の画面に遷移する。このとき、Transition 内で先に設定された HttpServletRequest、HttpServletResponse または ServiceResult を取得することが可能であればその内容を利用することができる。

jp.co.intra_mart.framework.base.service.Transition は抽象クラスであるため、開発者がこのクラスを利用して Transition を作成する場合はすべての抽象メソッドを実装しなくてはならない。

IM-JavaEE Framework ではあらかじめ目的に応じた以下のような Transition を用意している:

- jp.co.intra_mart.framework.base.service.DefaultTransition
- jp.co.intra_mart.framework.base.service.IntramartPageBaseTransition

画面遷移に関して特殊な事情がない限りはこれらのクラスを継承することを推奨する。

3.5.6.1 Transition

Transition クラスはすべての Transition のもととなる抽象クラスである。このクラスでは開発者が頻繁に使用すると考えられるメソッドがあらかじめいくつか実装されている。

3.5.6.1.1 getNextPagePath()

Transition の getNextPagePath()メソッドはリクエストで指定されたアプリケーション ID とサービス ID から ServicePropertyHandler の getNextPagePath(String application, String service)メソッドを使用して遷移パス(「3.4.4.2.4 遷移パス」参照)を取得する。この動作を「図 3-20 Transition の getNextPathPage(キーなし)」に示す。

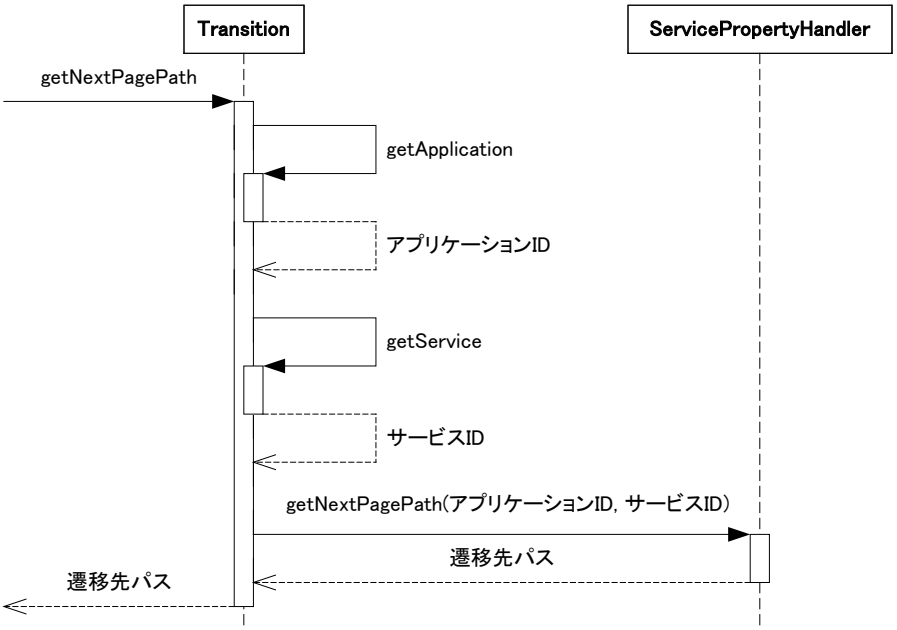


図 3-20 Transition の getNextPathPage(キーなし)

「図 3-20 Transition の getNextPathPage(キーなし)」に示したシーケンス図の中で getApplication()と getService()という 2 つのメソッドがある。これらのメソッドはそれぞれリクエストに対するアプリケーション ID とサービス ID を取得するものである。これらの値は ServiceServlet から Transition に対して設定される。

これらのメソッドをオーバーライドすればアプリケーション ID やサービス ID をリクエストとは無関係なものにすることが可能であるが、画面遷移の保守が難しくなるため推奨されない。

3.5.6.1.2 getNextPagePath(String key)

Transtion の getNextPagePath(String key)メソッドはリクエストで指定されたアプリケーション ID とサービス ID、さらにキーから ServicePropertyHandler の getNextPagePath(String application, String service, String key)メソッドを使用して遷移パス(「3.4.4.2.3 キー付遷移パス」参照)を取得する。この動作を「図 3-21 Transition の getNextPathPage(キー付)」に示す。

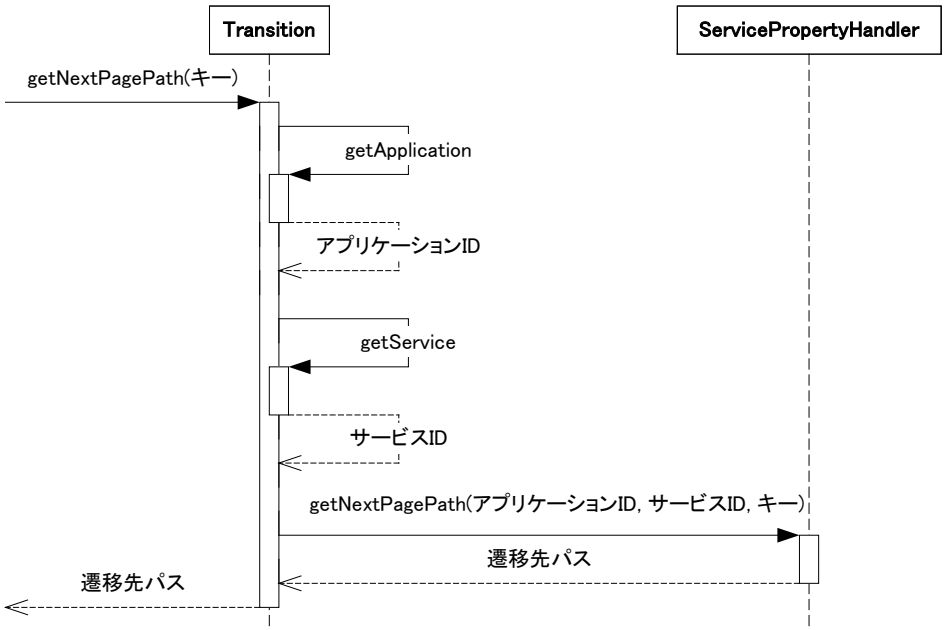


図 3-21 Transition の getNextPathPage（キー付）

「図 3-21 Transition の getNextPathPage（キー付）」に示したシーケンス図の中で getApplication()と getService()については「3.5.6.1.1 getNextPagePath()」で説明した内容と同様である。

3.5.6.2 DefaultTransition

IM-JavaEE Framework を前提として開発を行い、処理も何も行わずに単純に次の画面に forward する場合、開発者は Transition を作成・設定する必要はない。リクエストに対して Transition が設定されていない場合、IM-JavaEE Framework では DefaultTransition が設定されているものとみなされる（リクエストに該当する Transition が設定されていない場合、ServiceManager の getTransition メソッドの戻り値が DefaultTransition となる）。DefaultTransition の setInformation メソッドや transfer メソッドでは特殊な処理は何も行わず、次の画面をプロパティから取得して forward するだけである。DefaultTransition の動作を「図 3-22 DefaultTransition の動作」に示す。

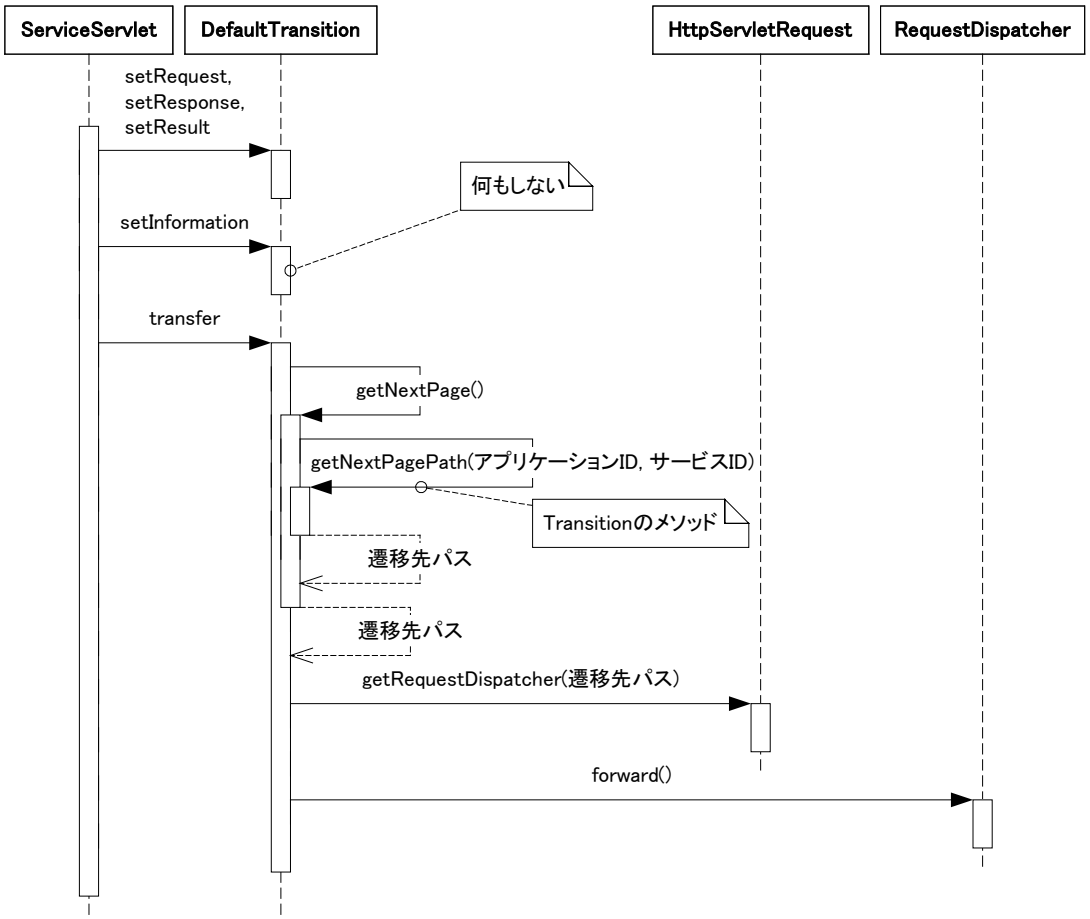


図 3-22 DefaultTransition の動作

DefaultTransition を継承して Transition を作成した場合、以下の利点がある。

- 必要以上にメソッドを実装なくてもよい。
- 簡易的な transfer メソッドが既に実装されており、JSP や Servlet に対する forward であれば遷移先をプロパティに設定するだけでよい。
- リクエストがファイルアップロードの場合、その内容を取得するメソッド（getEntity）が用意されている。

「図 3-22 DefaultTransition の動作」の transfer メソッドでは、結果的に ServicePropertyHandler の getNextPagePath(String application, String service)メソッド（「3.4.4.2.4 遷移パス」を参照）で取得されるパスに forward するだけである。そのため、複雑な遷移を必要としない場合、開発者がするべきことは「3.4.4.2.4 遷移パス」で指定されるプロパティを設定することだけである。

3.5.6.3 IntramartPageBaseTransition

IntramartPageBaseTransition は IM-JavaEE Framework の JSP のページから intra-mart のスクリプト開発モデルで作成された画面へのフォワードを行う Transition である。

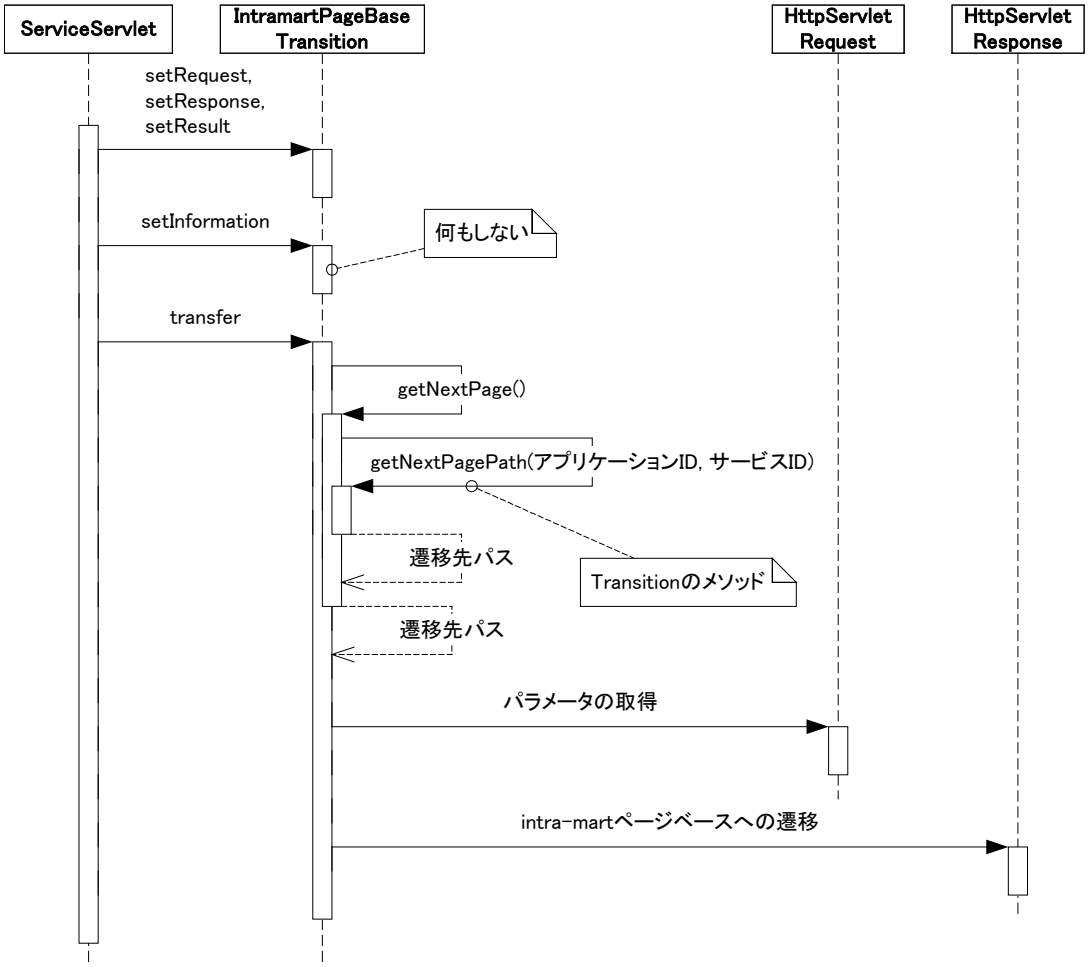


図 3-23 IntramartPageBaseTransition の動作

3.5.6.3.1 単純な遷移

IM-JavaEE Framework のサービスフレームワークを利用して次の画面に遷移する最も単純な方法はプロパティを設定することである。ServicePropertyHandler の getNextPagePath(String application, String service)メソッドで取得される値が遷移先のページパスとなるようにプロパティを設定する。ここで application と service はそれぞれリクエストに指定されたアプリケーション ID とサービス ID である。

この場合、Transition を設定する必要はない。

3.5.6.3.2 次画面に情報を渡す遷移

遷移先の画面で何らかの処理結果を受け取ったり、画面表示用のキーワードなどが必要とされる場合、Transition の setInformation メソッド内でその情報を設定する。最も簡単な実装の例を「リスト 3-7 次画面に情報を渡す Transition」に示す。

リスト 3-7 次画面に情報を渡す Transition

```
...
public class MyTransition extends DefaultTransition {
    public void setInformation() throws TransitionException {
        HttpServletRequest request = getRequest();
        request.setAttribute("search", "keyword");
    }
}
...
```

「リスト 3-7 次画面に情報を渡す Transition」ではリクエストの属性"search"に値"keyword"を設定している。また DefaultTransition を継承しており、setInformation 以外のメソッドがオーバライドされていないので、「3.5.6.3.1 単純な遷移」で示したような単純な遷移が行われるだけである。

3.5.6.3.3 複数の遷移先

遷移先がただ 1 つに固定されている場合、DefaultTransition を使えば IM-JavaEE Framework の画面に遷移することは容易である。しかし、条件によって遷移先を振り分けたい場合、このままでは対応できない。いくつかの遷移先の候補があってそれらを動的に振り分けたい場合、独自の Transition を作成する必要がある。

この場合、具体的には以下のようないくつかの方法があげられる:

- DefaultTransition を継承したクラスを作成し、getNextPage メソッドをオーバーライドする。
- DefaultTransition を継承したクラスを作成し、getNextPagePath メソッドをオーバーライドする。
- Transition(または DefaultTransition)を継承したクラスを作成し、transfer メソッドをオーバーライドする。

遷移先が IM-JavaEE Framework である場合、最初の方法が最も実装が容易である場合が多い。この場合の例を「リスト 3-8 複数の遷移先」に示す。

この Transition では MyResult という独自の ServiceResult を受け取るようになっている。MyResult には処理結果として数値が設定され、それらは getNumber メソッドで取得できるように実装されているとする。「リスト 3-8 複数の遷移先」ではこの数値を元にキーを再定義し(setInformation メソッド)、そのキーに応じて遷移先を取得している(getNextPage メソッド)。

リスト 3-8 複数の遷移先

```
...
import jp.co.intra_mart.framework.base.service.*;

public class ConditionalTransition extends DefaultTransition {

    private String condition = null;

    public void setInformation() {
        MyResult result = (MyResult)getResult();
        if (result.getNumber() == 1) {
            this.condition = "cond1";
        } else if (result.getNumber() == 2) {
            this.condition = "cond2";
        } else if (result.getNumber() == 3) {
            this.condition = "cond3";
        }
    }

    public void getNextPage() throws ServicePropertyException {
        return getNextPagePath(this.condition)
    }
}
```

3.5.6.3.4 その他の遷移先

遷移先が JSP やサーブレット以外である場合、単純な画面遷移では対応できない。例として以下のような場合があげられる。

- PDF ファイルの表示
- ファイルのダウンロード
- 他のコンテンツへのリダイレクト

この場合、一時的に JSP またはサーブレットに遷移してその中で javax.servlet.http.HttpServletResponse の出力ストリームに出力する、またはリダイレクトを行う方法が最も単純である。

ファイルをダウンロードさせる場合の Transition とサーブレットの例をそれぞれ「リスト 3-9 ファイルダウンロード用の Transition」と「リスト 3-10 ファイルダウンロード用のサーブレット」に示す。

この Transition では DownloadResult という独自の ServiceResult を受け取るようになっている。DownloadResult には処理結果としてダウンロードするときのブラウザ側に送信するファイル名とその実際の内容が設定され、それらは getFilename メソッド、getFiledata メソッドで取得できるように実装されているとする。「リスト 3-9 ファイルダウンロード用の Transition」ではこれらの内容を取得し（setInformation メソッド）、「リスト 3-10 ファイルダウンロード用のサーブレット」ではその内容をブラウザ側に送信している（doGet メソッド）[5]。

リスト 3-9 ファイルダウンロード用の Transition

```
...
import jp.co.intra_mart.framework.base.service.*;

public class DownloadTransition extends DefaultTransition {

    public void setInformation() {
        DownloadResult result = (DownloadResult)getResult();
        getRequest().setAttribute("filename", result.getFilename());
        getRequest().setAttribute("filedata", result.getFiledata());
    }
}
```

[5] ここでは HttpServlet の doGet メソッドを使用しているが、実際にはリクエストにあわせて doPost にするなどの修正が必要である。

リスト 3-10 ファイルダウンロード用のサーブレット

```
・・・
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DownloadServlet extends HttpServlet {
・・・

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                  throws IOException, ServletException {

        // 情報の取得
        String filename = (String)request.getAttribute("filename");
        byte[] filedata = (byte[])request.getAttribute("filedata");

        // ヘッダの出力
        response.setContentType("application/octet-stream");
        response.setHeader("Content-Disposition",
                           "attachment; filename=¥"" + filename + "¥"");

        // 出力ストリームへ出力
        OutputStream os = response.getOutputStream();
        os.write(filedata);
        os.flush();
        os.close();
    }
}
```

また、あまり推奨されない方法ではあるが、Transition の transfer メソッドをオーバライドすることによって同様の機能を実現することも可能である。「図 3-19 サービスフレームワーク(画面遷移)」を見るとわかるように、次の画面に遷移する(表示する)方法は transfer メソッドに任されている[6]。そのため、「リスト 3-10 ファイルダウンロード用のサーブレット」で示したメソッドの内容を transfer メソッドで行えばファイルのダウンロードは可能である。ただし、transfer メソッドのオーバライドは特殊な方法であり、あまり汎用性がない。そのためこの方法は推奨されない。

3.6 画面表示

画面表示を行う方法は複数あるが、ここでは IM-JavaEE Framework でもっとも標準的と考えられる方法について述べる。

3.6.1 JSP

IM-JavaEE Framework で開発を行う場合、遷移先として JSP が最も多いと考えられる。単純に表示するだけならば普通の JSP となんら変化はないが、初期表示に必要となる内容を取得したり、次の画面にセッションを続けるためには以下のようなものを利用することができる。

- タグライブラリ
- HelperBean

[6] DefaultTransition や IntramartPageBaseTransition ではこの部分は実装済みであるため、これらのクラスを継承している場合再定義する必要はない。

3.6.1.1 タグライブラリ

IM-JavaEE Framework ではセッションを維持したり画面表示を補助するための以下のような JSP 拡張タグがある。

- Form
- Link
- Frame
- Param
- Submit
- SubmitLink
- HelperBean
- Message

これらの JSP 拡張タグは、IM-JavaEE Framework では標準では「リスト 3-11 IM-JavaEE Framework のタグライブラリの URI」に示す URI で提供されている。

リスト 3-11 IM-JavaEE Framework のタグライブラリの URI

```
http://www.intra-mart.co.jp/taglib/core/framework
```

JSP の中では「リスト 3-11 IM-JavaEE Framework のタグライブラリの URI」に示す URI を指定すれば im-JavaEE のタグライブラリを使用できる。IM-JavaEE Framework のタグライブラリを使用する場合の JSP の記述例を「リスト 3-12 IM-JavaEE Framework のタグライブラリを使用する JSP」に示す。この例では、IM-JavaEE Framework のタグライブラリのプレフィックスとして"imartj2ee"を指定し、Form タグを利用している。

リスト 3-12 IM-JavaEE Framework のタグライブラリを使用する JSP

```
・・・
<%@ taglib prefix="imartj2ee" uri="http://www.intra-mart.co.jp/taglib/core/framework" %>
・・・
<imartj2ee:Form application="sample" service="test" method="POST">
・・・
</imartj2ee:Form>
```

3.6.1.1.1 Form

Form タグの書式は通常の HTML の<FORM>タグとほぼ同じであるが、以下のような違いがある。

- action 属性が存在しない。
- 属性名はすべて小文字である必要がある。

Form タグでは action 属性の代わりに application 属性と service 属性が存在する。これらの属性はそれぞれアプリケーション ID とサービス ID を意味する。

3.6.1.1.2 Link

Link タグの書式は通常の HTML の<A>タグとほぼ同じであるが、以下のような違いがある。

- href 属性が存在しない。
- 属性名はすべて小文字である必要がある。

Link タグでは href 属性の代わりに application 属性と service 属性が存在する。これらの属性はそれぞれアプリケーション ID とサービス ID を意味する。

Link タグでは Param タグを入れ子にして使うことができる。この場合、Param タグの内容が Link タグによって送られるリクエストのパラメータとなる。この場合、リクエストは GET で送られる。

3.6.1.1.3 Frame

Frame タグの書式は通常の HTML の<FRAME>タグとほぼ同じであるが、以下のような違いがある。

- src 属性が存在しない。
- 属性名はすべて小文字である必要がある。

Frame タグでは src 属性の代わりに application 属性と service 属性が存在する。これらの属性はそれぞれアプリケーション ID とサービス ID を意味する。

Frame タグでは Param タグを入れ子にして使うことができる。この場合、Param タグの内容が Frame タグによって送られるリクエストのパラメータとなる。この場合、リクエストは GET で送られる。

3.6.1.1.4 Param

Param タグは Link タグや Frame タグによってブラウザからサーバに送られるときのリクエストのパラメータを設定する。Param タグは常に Link タグまたは Frame タグとともに使われるものであり、単独では存在できない。書式を以下に示す。

```
<prefix:Param name="name_string" value="exp" />
```

この中で prefix は IM-JavaEE Framework のタグライブラリを使うときのプレフィックスを、name_string はリクエストが送られる場合のパラメータ名を、exp はその値を意味する。

上記の書式で Link または Frame タグでリクエストが送られた場合、サーバ側では javax.servlet.http.HttpServletRequest の getParameter メソッドでその値を取得することができる。

3.6.1.1.5 Submit

Submit タグは Form タグで指定されているサービスを変更するものである。Submit タグは Form タグの中でしか使用することができない。

Form タグの中に複数の Submit ボタンを配置し、押されたボタンによって違う処理を行いたい場合が少なからずある。Submit ボタンとして HTML の<INPUT type="submit">タグを使おうとすると、この様な処理は実現できない。この場合、Submit タグによって生成された Submit ボタンであればより柔軟に上記の処理が可能となる。Submit タグは HTML の<INPUT>タグと書式はほぼ同じであるが、以下の点が異なる。

- type 属性が存在しない。
- 属性名はすべて小文字である必要がある。

このタグそのものが Submit ボタンを意味するため、type 属性は存在しない。また application 属性と service 属性が存在する。これらの属性はそれぞれアプリケーション ID とサービス ID を意味する。

Form タグと Submit タグのどちらでもアプリケーション ID とサービス ID の指定が可能である。両方とも指定された場合、実際に押された Submit タグの値が優先される。

3.6.1.1.6 **SubmitLink**

SubmitLink タグは Form タグで指定されたフォームをサブミットするリンクである。SubmitLink タグは Form タグの内部と外部の両方から指定できる。

Form タグの内側または外側から Submit ボタンではなくリンクによってフォームをサブミットしようとする場合、HTML の<A>タグではこのような処理は直接することはできない。この場合、SubmitLink タグによって生成されたリンクであればより柔軟に上記の処理が可能となる。SubmitLink タグは HTML の<A>タグと書式はほぼ同じであるが、以下の点が異なる。

- 以下の属性が存在しない。
  - charset
  - type
  - hreflang
  - rel
  - rev
  - target
- 属性名はすべて小文字である必要がある。

このタグは Submit ボタンの代わりを意味するため、HTML の<FORM>タグで定義されるべき属性については指定できない。また application 属性と service 属性が存在する。これらの属性はそれぞれアプリケーション ID とサービス ID を意味する。また、サブミットするフォームを指定する form 属性がある。

SubmitLink タグにおける application 属性、service 属性および form 属性は必須ではないが、以下のような制限がある。

- Form タグの外部にこのタグを指定する場合、form 属性の指定は必須。この場合、form 属性で指定されたフォームに対してサブミットが行われる。
- Form タグの内部にこのタグを指定する場合、form 属性の指定は不可。この場合、このタグを内包しているフォームに対してサブミットが行われる。
- application 属性を指定する場合、service 属性の指定は必須。
- application 属性を省略する場合、service 属性の指定は不可。
- application 属性を省略して service 属性を指定することは可能。この場合、application 属性は、このタグがサブミットするフォームの application 属性が指定されたものとみなされる。

3.6.1.1.7 **HelperBean**

HelperBean タグは HelperBean を使用するためのタグである。HelperBean については「3.6.1.2 HelperBean」で説明する。HelperBean タグの書式を以下に示す。

```
<prefix:HelperBean id="bean_name" class="class_name" />
```

このタグを使用することにより JSP の中でクラス class_name のインスタンス bean_name が使用できるようになる。

3.6.1.1.8 **Message**

Message タグは地域対応されたメッセージを表示するためのタグである。このタグはメッセージフレームワーク（「2.2.5 メッセージフレームワーク」を参照）で取得できるメッセージを簡易的に表示する。

Message タグでは以下の属性を指定できる。

- application
  アプリケーション ID を指定する。
- key
  メッセージのキーを指定する。
- locale
  メッセージを構築するときのロケールを指定する。省略した場合、サービスフレームワークで決定されるロケール（「3.3.1 ロケール」を参照）が使用される。

また、内部タグとして MessageParam タグを指定することもできる。MessageParam タグはメッセージ内部の変数に置き換えられる値である。

このタグは内部で jp.co.intra_mart.framework.base.message.MessageManager の getMessage(String, String, Object[], Locale)メソッドを使用している。この場合、application 属性、key 属性、MessageParam タグの内容および locale 属性がそれぞれこのメソッドの第 1～第 4 引数に対応している。

Message タグおよび MessageParam タグを「リスト 3-13 Message タグの例」に示すように使用した場合、「リスト 3-14 メッセージの取得」のコードのように MessageManager の getMessage メソッドを呼び出したときの結果と等価な値が表示される。

リスト 3-13 Message タグの例

```
<prefix:Message application="myapp" key="mykey" locale="ja_JP">
    <prefix:MessageParam value="hello" />
    <prefix:MessageParam value="world" />
</prefix:Message>
```

リスト 3-14 メッセージの取得

```
MessageManager manager = MessageManager.getMessageManager();
Object[] parameters = {"hello", "world"};
String message =
    manager.getMessage("myapp", "mykey", parameters, new Locale("ja", "JP"));
```

また、Message タグおよび MessageParam タグを「リスト 3-15 Message タグの例（ロケールを省略）」に示すようにロケールを省略して使用した場合、「リスト 3-16 メッセージの取得（ロケールを省略）」で示すように ServiceManager の getLocale メソッドで決定されるロケールを指定したときの結果と等価な値が表示される。

リスト 3-15 Message タグの例（ロケールを省略）

```
<prefix:Message application="myapp" key="mykey">
    <prefix:MessageParam value="hello" />
    <prefix:MessageParam value="world" />
</prefix:Message>
```

リスト 3-16 メッセージの取得(ロケールを省略)

```
MessageManager manager = MessageManager.getMessageManager();
Object[] parameters = {"hello", "world"};
ServiceManager service = ServiceManager.getServiceManager();
Locale locale = service.getLocale(request, response);
String message =
      manager.getMessage("myapp", "mykey", parameters, locale);
```

3.6.1.1.9 MessageParam

MessageParam タグは Message タグに渡すパラメータを指定するタグである。MessageParam タグは Message タグの内部でのみ使用することが可能である。このタグは value 属性のみを持つ。複数のパラメータを指定する場合、このタグを複数記述する。この場合、記述された順番と同等の配列が指定されたことと同様の意味になる(「3.6.1.1.8 Message」を参照)。

3.6.1.2 HelperBean

初期表示用のデータを取得する場合、画面遷移時にデータを取得して表示用データとして渡す方法と画面表示時にデータを取得する方法が考えられる。後者の場合、HelperBean を利用して初期表示用データを取得するとより便利になる。

HelperBean の構成を「図 3-24 HelperBean の構成」に示す。

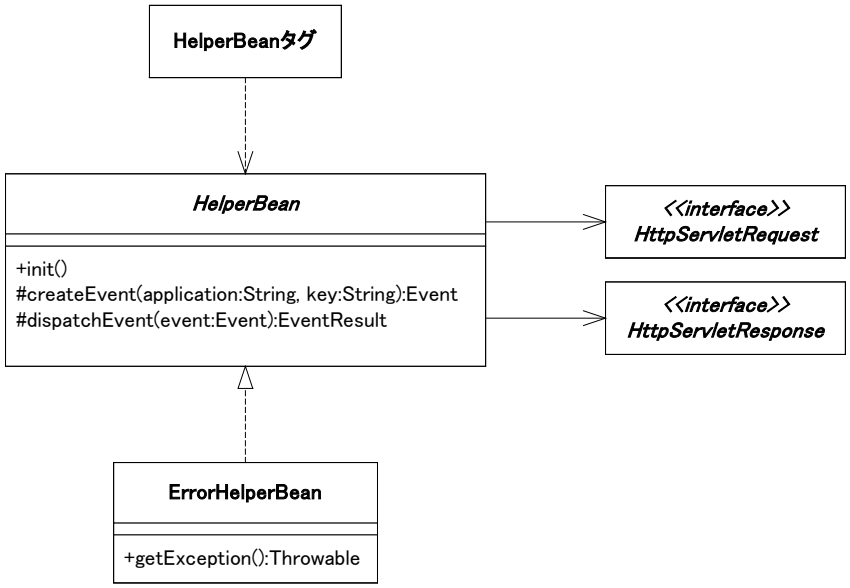


図 3-24 HelperBean の構成

HelperBean は以下のタグライブラリを使うことで使用することが可能となる。

```
<%@ taglib prefix="prefix" uri="http://www.intra-mart.co.jp/taglib/core/framework" %>
・・・
<prefix:HelperBean id="bean_name" class="class_name" />
```

この中で prefix は IM-JavaEE Framework のタグライブラリを使うときのプレフィックスを、bean_name は JSP の中でスクリプト変数として使うときの変数名を、class_name は HelperBean のクラス名を意味する。

class_name に指定できるクラスは以下の条件を満たす必要がある。

- jp.co.intra_mart.framework.base.web.bean.HelperBean のサブクラスである。
- 以下の条件をすべて満たすコンストラクタがあること。
  - public である。
  - 引数がない。
  - throws 節に jp.co.intra_mart.framework.base.web.bean.HelperBeanException またはそのサブクラスが指定されている。

このタグでは class_name で指定されたクラスが新たに生成され、リクエストとレスポンスが設定され初期化メソッド init()が呼び出される。この様子を「図 3-25 HelperBean の初期化」に示す。

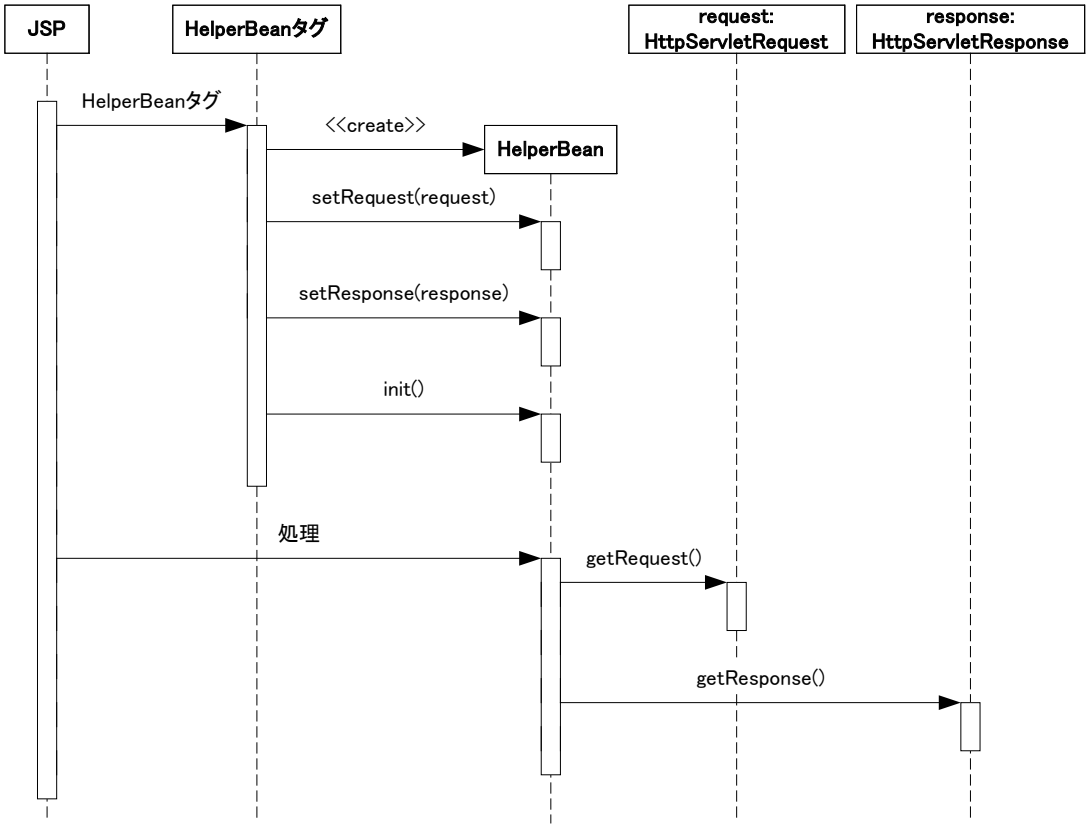


図 3-25 HelperBean の初期化

init メソッド内または HelperBean タグで初期化された後に呼ばれる HelperBean のメソッドからは getRequest メソッドや getResponse メソッドによってリクエストやレスポンスを取得することが可能である。getRequest メソッドや getResponse メソッドはコンストラクタ内で呼んだ場合は何も設定されてないため null が返ってくる。

また、HelperBean にはイベント処理を簡潔に行う createEvent メソッドや dispatchEvent メソッドが用意されている。これらの使用方法は「図 3-16 ServiceControllerAdapter からイベントフレームワークを利用」で示した内容と同様である。HelperBean 内からイベントを呼び出す場合のサンプルコードを「リスト 3-17 HelperBean のサンプル」に示す。

リスト 3-17 HelperBean のサンプル

```
・・・
import java.util.Collection;
import jp.co.intra_mart.framework.base.web.bean.HelperBean;
import jp.co.intra_mart.framework.base.web.bean.HelperBeanException;
・・・

public class TestHelperBean extends HelperBean {

	private String keyword;

	public TestHelperBean() throws HelperBeanException {
		super();
		this.keyword = null;
	}

	public void init() {
		this.keyword = (String)(getRequest().getAttribute("keyword"));
	}

	public Collection getSearchResult() {
		SearchEvent searchEvent = (SearchEvent)createEvent("sample", "search");
		searchEvent.setKeyword(this.keyword);
		SearchEventResult result = (SearchEventResult)dispatchEvent(searchEvent);

		return result.getSearchList();
	}
}
```

「リスト 3-17 HelperBean のサンプル」はリクエストの属性"keyword"で指定されるキーワードをもとに情報を検索してくる。情報の検索は IM-JavaEE Framework のイベントフレームワークを利用して SearchEvent クラスと SearchEventResult クラスで取得される。IM-JavaEE Framework のイベントフレームワークの詳細は「4 イベントフレームワーク」で述べる。

3.6.2 JSP 以外の画面

IM-JavaEE Framework では JSP 以外の画面にも遷移することが可能であるが、その標準的な方法は特に規定していない。この場合、画面表示の内容も遷移先の画面によって独自に実装する必要がある（Servlet に遷移する場合、HttpServletResponse に出力する、まったく別の URL にリダイレクトするなど）。これらの独自の画面では「3.6.1 JSP」で述べたようなタグライブラリや HelperBean は使用できない。

3.7 例外処理

IM-JavaEE Framework のサービスフレームワークでは主に「図 3-26 サービスフレームワークの例外」で示した箇所で例外を検知できるようになっている。

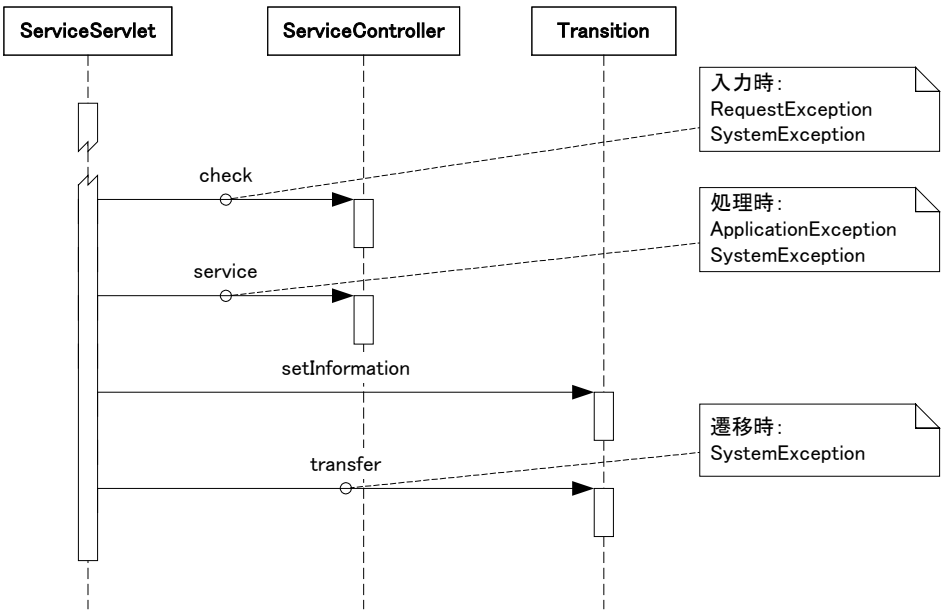


図 3-26 サービスフレームワークの例外

- 入力時(ServiceController の check メソッド)
- 処理時(ServiceController の service メソッド)
- 画面遷移時(Transition の transfer メソッド)

3.7.1 入力時の例外処理

ServiceController の check メソッドでは以下の例外またはそのサブクラスを throw することが可能である。

- jp.co.intra_mart.framework.base.service.RequestException
- jp.co.intra_mart.framework.system.exception.SystemException

開発者は ServiceController の check メソッドを実装するとき、入力内容に誤りや不備がある場合に RequestException 及びそのサブクラスを出すように実装するべきである。それ以外の例外は SystemException 及びそのサブクラスとして出すように実装するべきである。

ServiceController の check メソッドで RequestException が発生した場合の動きを「図 3-27 RequestException 発生時」に示す。

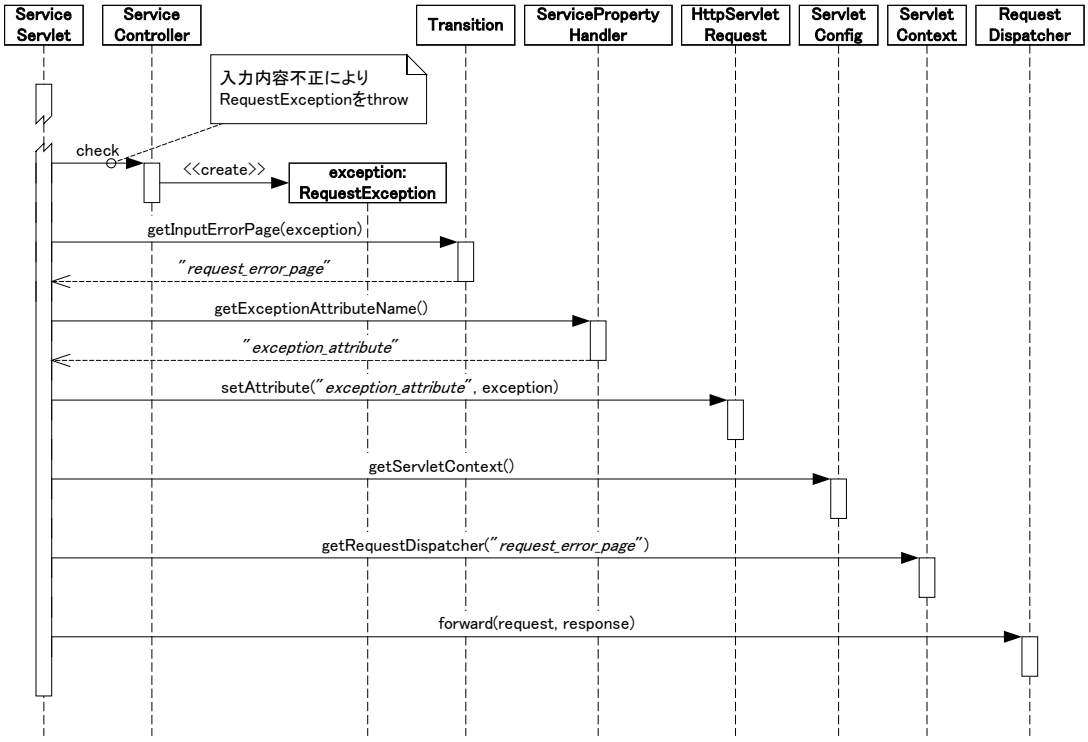


図 3-27 RequestException 発生時

ServiceController の check メソッドで SystemException が発生した場合の動きを「図 3-28 SystemException 発生時（check メソッド）」に示す。

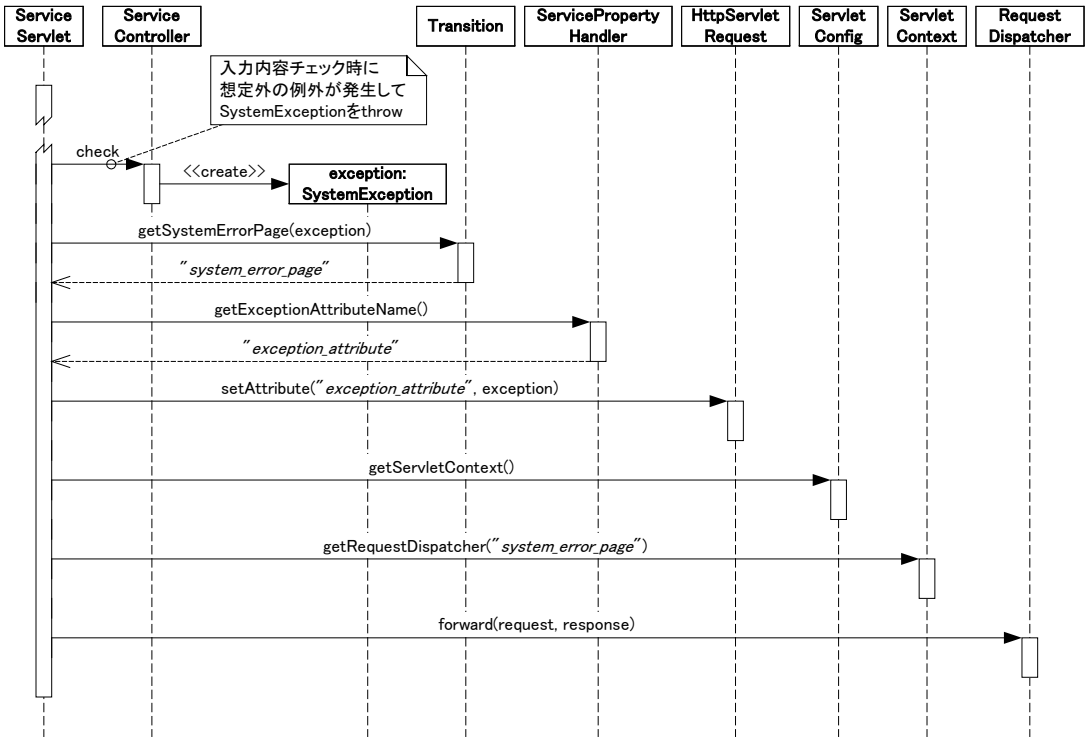


図 3-28 SystemException 発生時（check メソッド）

「図 3-27 RequestException 発生時」と「図 3-28 SystemException 発生時（check メソッド）」の違いは check メソッドが throw した例外の種類による動作のみである。check メソッドが RequestException（またはそのサブクラス）を例外

として throw した場合、ServiceServlet は Transition の getInputErrorPage メソッドを呼んで入力例外ページを取得し、そのページに遷移する。一方、check メソッドが SystemException（またはそのサブクラス）を例外として throw した場合、ServiceServlet は Transition の getSystemErrorPage メソッドを呼んでシステム例外ページを取得し、そのページに forward する。

例外情報（check メソッドから throw された例外）はリクエストの属性に設定される。属性の名前はプロパティによって設定されたものであり、ServicePropertyHandler の getExceptionAttributeName メソッドで取得されるものである。forward された遷移先のページではこの属性を通じて例外情報を取得することができる。

3.7.1.1 ログ出力

ServiceServlet は check メソッドから例外が throw された場合にログを出力する。以下は例外の種類と出力されるログのレベルである。

例外の種類	ログレベル
RequestException	出力しない
SystemException	error

3.7.2 処理時の例外処理

ServiceController の service メソッドでは以下の例外またはそのサブクラスを throw することが可能である。

- jp.co.intra_mart.framework.system.exception.ApplicationException
- jp.co.intra_mart.framework.system.exception.SystemException

開発者は ServiceController の serice メソッドを実装するとき、同一キーのデータの二重登録や削除済みのデータに対するアクセスなどユーザによる操作の範囲内で想定される不具合がある場合に ApplicationException 及びそのサブクラスを出すように実装するべきである。それ以外の例外は SystemException 及びそのサブクラスとして出すように実装するべきである。

ServiceController の service メソッドで ApplicationException が発生した場合の動きを「図 3-29 ApplicationException 発生時」に示す。

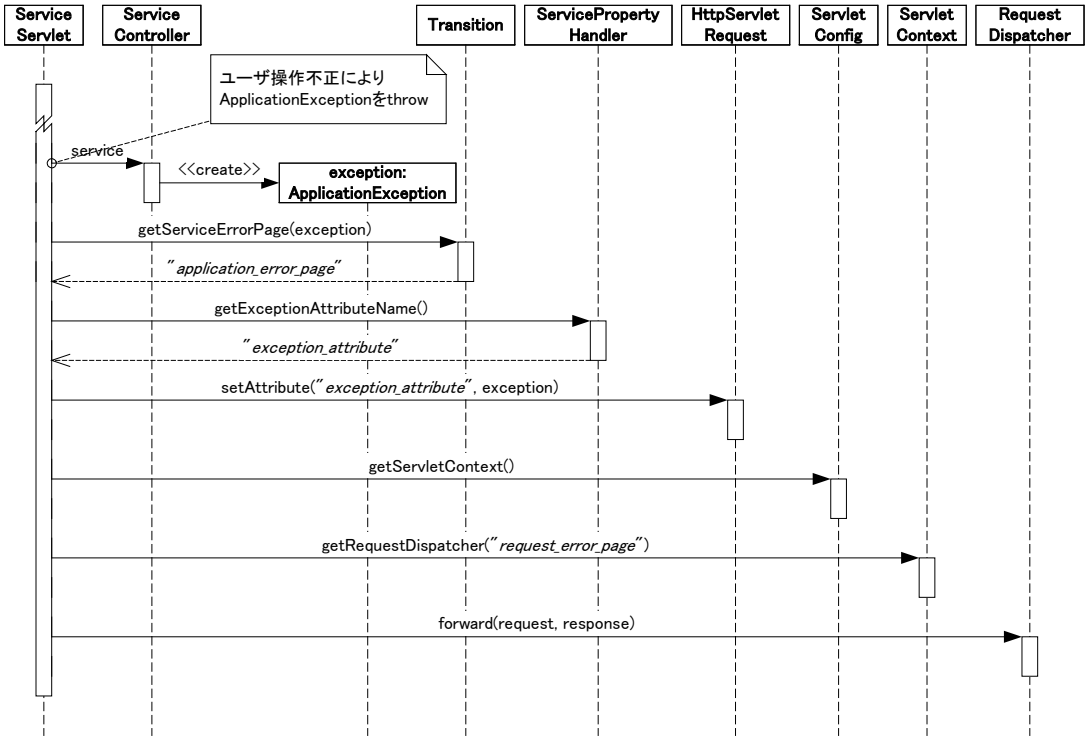


図 3-29 ApplicationException 発生時

ServiceController の service メソッドで SystemException が発生した場合の動きを「図 3-30 SystemException 発生時（service メソッド）」に示す。

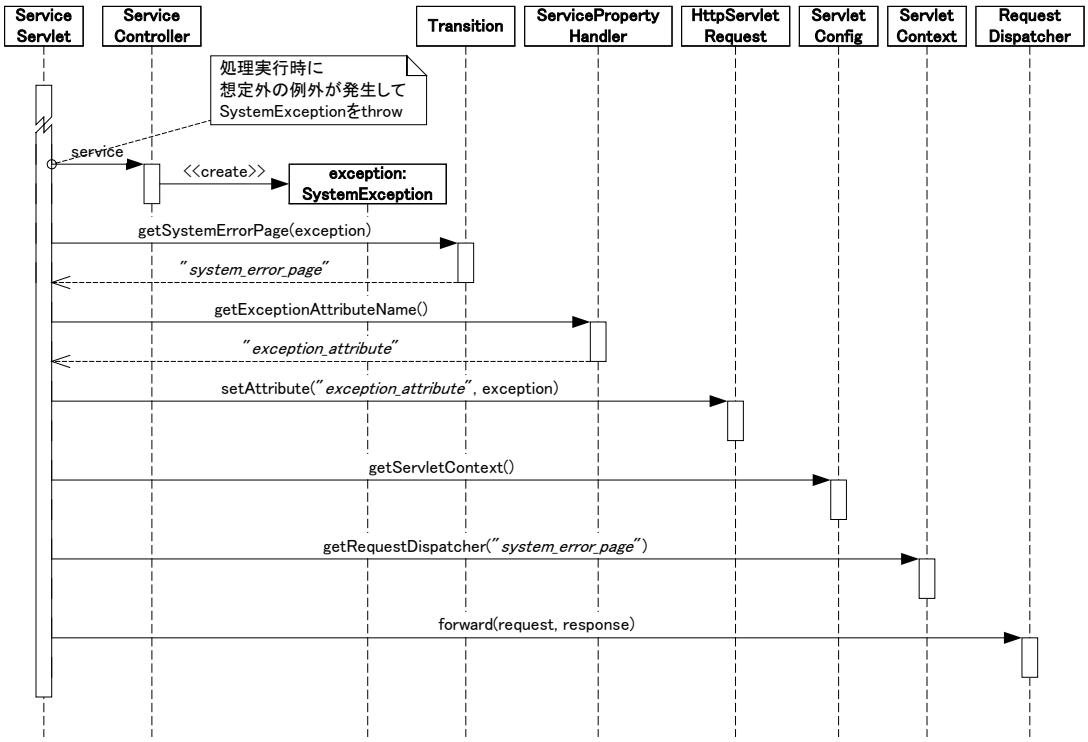


図 3-30 SystemException 発生時（service メソッド）

「図 3-29 ApplicationException 発生時」と「図 3-30 SystemException 発生時（service メソッド）」の違いは service メソッドが throw した例外の種類による動作のみである。service メソッドが ApplicationException（またはそのサブク

ラス)を例外として throw した場合、ServiceServlet は Transition の getServiceErrorPage メソッドを呼んでアプリケーション例外ページを取得し、そのページに遷移する。一方、service メソッドが SystemException(またはそのサブクラス)を例外として throw した場合、ServiceServlet は Transition の getSystemErrorPage メソッドを呼んでシステム例外ページを取得し、そのページに forward する。

例外情報(service メソッドから throw された例外)はリクエストの属性に設定される。属性の名前はプロパティによって設定されたものであり、ServicePropertyHandler の getExceptionAttributeName メソッドで取得されるものである。forward された遷移先のページではこの属性を通じて例外情報を取得することができる。

3.7.2.1 ログ出力

ServiceServlet は service メソッドから例外が throw された場合にログを出力する。以下は例外の種類と出力されるログのレベルである。

例外の種類	ログレベル
ApplicationException	warn
SystemException	error

Version 7.1 以前は ApplicationException が throw された場合に error レベルが出力されていたが、Version 7.2 以降は warn レベルが出力されるように変更された。これは ApplicationException は業務例外として使用されるため、業務例外発生時のログ出力を制御することを目的とした変更である。
ApplicationException 発生時のログ出力は logger の設定を変更することでログを出力しないように変更することが可能である。以下は Logback の設定ファイル使用している場合の設定例である。

```
   ・・・

<logger name="foo.bar.*">
    <level value="warn" />
</logger>

   ・・・
```

intra-mart では以下のディレクトリパスに設定ファイルが格納されている。

- conf/log/im_logger.xml

3.7.3 画面遷移時の例外処理

ServiceController の service メソッドによる処理が正常終了した後、またはリクエストに対して ServiceController が関連付けられていない場合、Transition の transfer メソッドによって次のページに遷移する。画面遷移時に例外が発生した場合、IM-JavaEE Framework では想定外の例外として扱っているため、transfer メソッドでは jp.co.intra_mart.framework.system.exception.SystemException またはそのサブクラスを例外として throw する仕様になっている。

transfer メソッドでシステム例外が発生した場合のシーケンス図を「図 3-31 SystemException 発生時(transfer メソッド)」に示す。

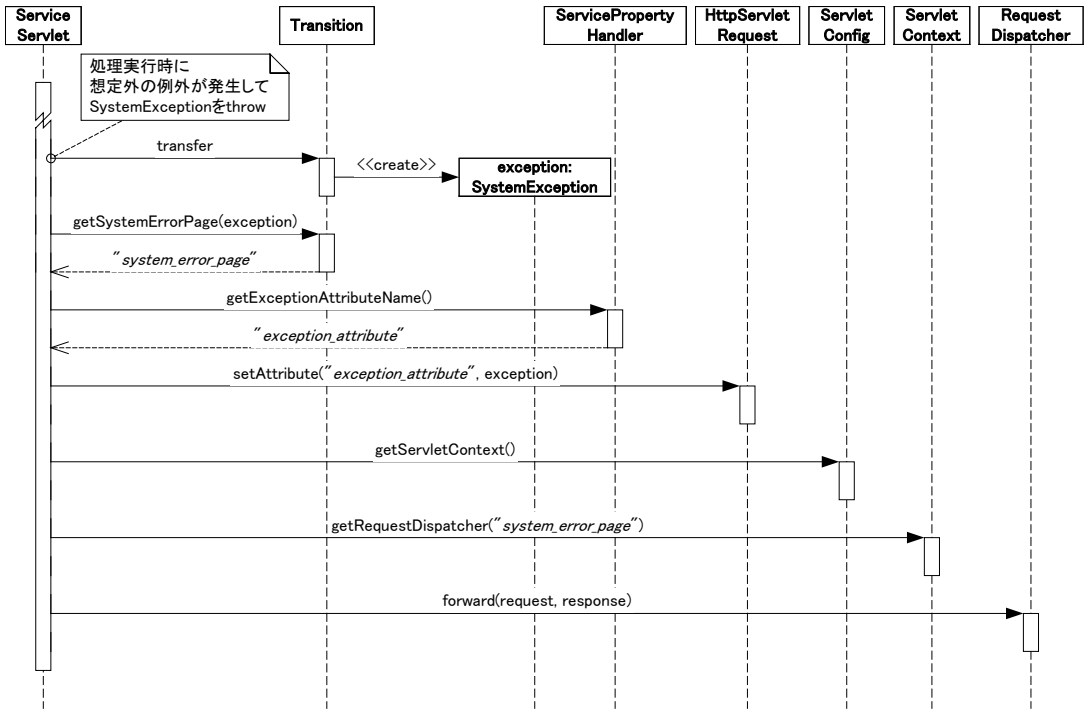


図 3-31 SystemException 発生時（transfer メソッド）

Transition の transfer メソッドが SystemException またはそのサブクラスを throw した後の動作は「3.7.1 入力時の例外処理」や「3.7.2 処理時の例外処理」で示した SystemException の場合と同様である。

3.7.4 画面出力時の例外処理

画面出力時の例外処理に関しては IM-JavaEE Framework では特に規定はしていない。

3.7.5 エラーページの取得

IM-JavaEE Framework ではエラーの場合の遷移先を Transition から取得する。Transition のサブクラスである DefaultTransition でエラーページを取得する様子を「図 3-32 エラーページの取得」に示す。

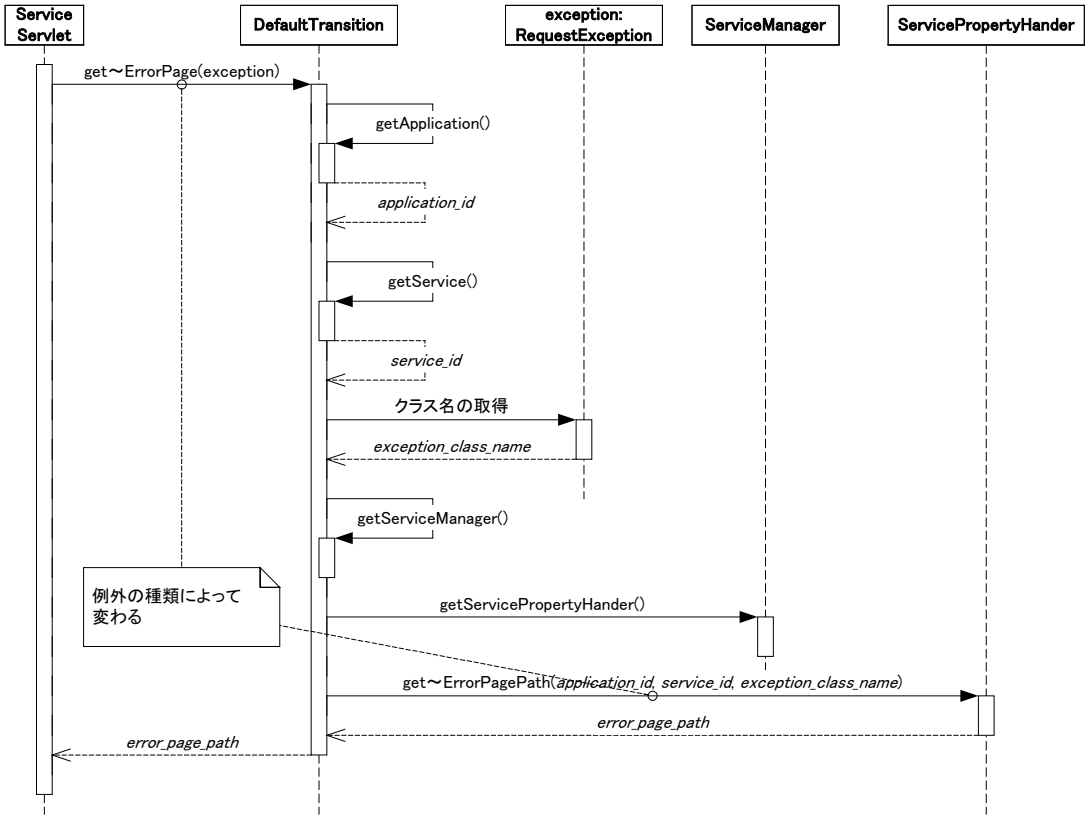


図 3-32 エラーページの取得

「図 3-32 エラーページの取得」において get～ErrorPage または get～ErrorPagePath とあるところは発生する例外の種類とその発生場所によって異なる。「表 3-10 例外発生時のメソッド」に呼ばれるメソッドの一覧を示す。

表 3-10 例外発生時のメソッド

発生する例外の種類	例外が発生する主なメソッド	呼ばれるメソッド
RequestException	ServiceController の check()	getInputErrorPage getInputErrorPagePath
ApplicationExcpetion	ServiceController の service()	getApplicationErrorPage getApplicationErrorPagePath
SystemException	ServiceController の check() ServiceController の service() Transition の transfer()	getSystemErrorPage getSystemErrorPagePath

3.7.6 エラーページの表示

「3.7.1 入力時の例外処理」～「3.7.3 画面遷移時の例外処理」で示した例外処理を行い「3.7.5 エラーページの取得」で取得したエラーページへ遷移し表示を行う。このとき、例外情報（throw された例外）をリクエストの属性から取得することができる。例外情報が含まれるリクエストの属性名は ServicePropertyHandler の getExceptionAttributeName メソッドで取得できる。

エラーページとして JSP を使用する場合、IM-JavaEE Framework の拡張タグである HelperBean タグと jp.co.intra_mart.framework.base.web.bean.ErrorHelperBean を利用して例外情報を取得することもできる。この場合、HelperBean タグの class 属性には ErrorHelperBean またはそのサブクラスのパッケージ名を含んだ完全なクラス名を指定する [7]。これにより ErrorHelperBean が初期化され、getException メソッドによって例外情報を取得する

[7] HelperBean タグの詳細については「3.6.1.2 HelperBean」を参照。

ことができる。

JSP 内で例外情報を取得する例を「リスト 3-18 ErroHelperBean による例外情報の取得」に示す。ここで foo.SampleErrorHelperBean は jp.co.intra_mart.framework.base.web.bean.ErrorHelperBean のサブクラスである。

リスト 3-18 ErroHelperBean による例外情報の取得

```
    ・・・
<%@ taglib prefix="im_j2ee" uri="http://www.intra-mart.co.jp/taglib/core/framework" %>
    ・・・
<im_j2ee:HelperBean id="errorBean" class="foo.SampleErrorHelperBean">
    ・・・
<%
    Throwable e = errorBean.getException();
    ・・・
%>
```

3.8 国際化

IM-JavaEE Framework のサービスフレームワークは国際化を意識して設計されている。国際化の対象としては以下のものがある。

- 表示
- 遷移

3.8.1 表示の国際化

地域対応した表示をする場合、もっとも単純な方法は JSP を国際化することである。この場合、レイアウトはどのロケールでも共通のものを使い、表示する内容だけを Message タグを利用して切り替える(「図 3-33 Message タグによる国際化」を参照)。

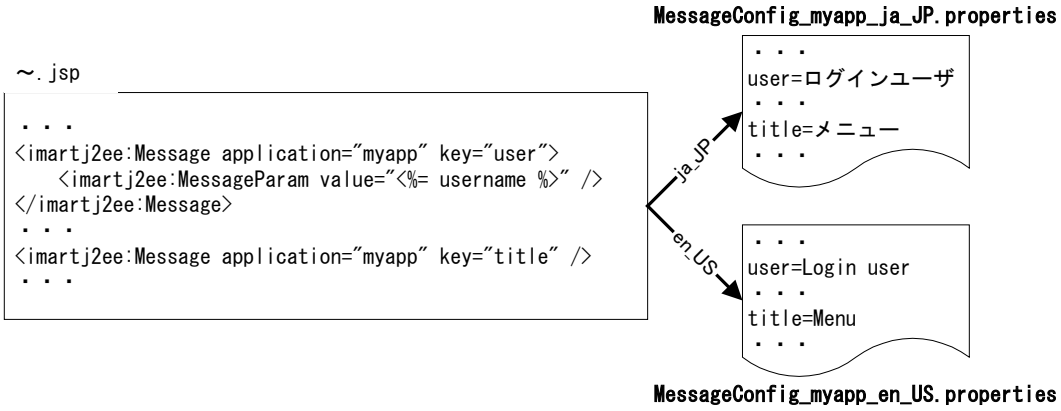


図 3-33 Message タグによる国際化

3.8.2 遷移の国際化

サービスフレームワークでは以下のプロパティに対してロケールごとに地域対応した値を用意することができる。

- ServiceController
- Transition
- 遷移先

これを利用すれば、ロケールごとに違う処理や遷移先を設定することができる。この場合、必要に応じてロケール

ごとにプロパティを用意する(「図 3-34 プロパティ設定による国際化」を参照)。

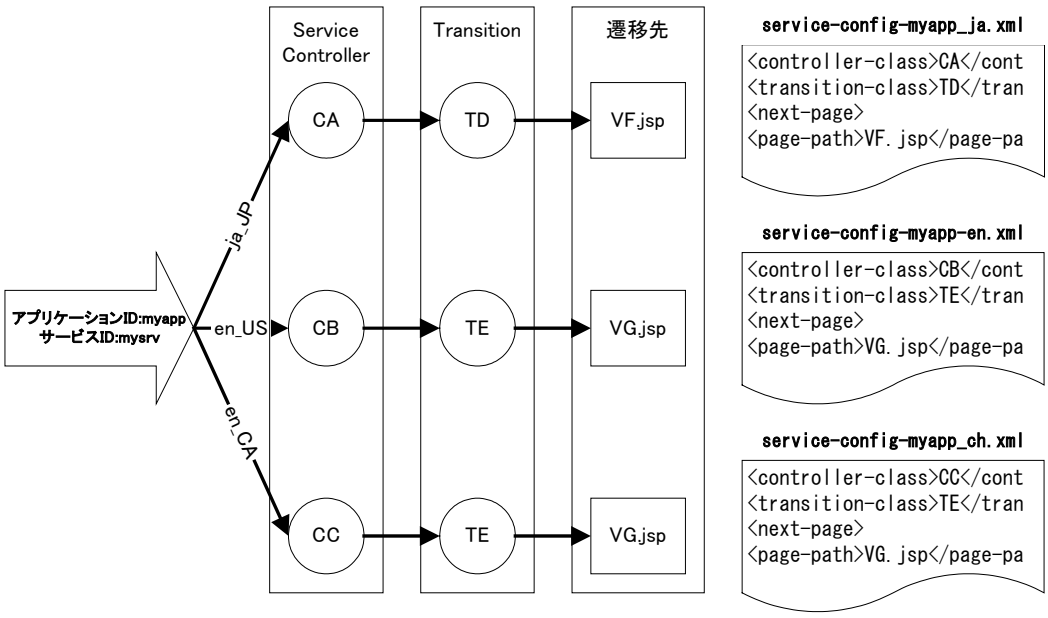


図 3-34 プロパティ設定による国際化

遷移先については「3.8.1 表示の国際化」に示したように同一の JSP に遷移させて文字列だけを地域対応させる方法もあるが、地域ごとにレイアウトを変えたい場合などは「図 3-34 プロパティ設定による国際化」のように遷移させるJSPそのものを変更したほうがよい場合もある。どちらの方法も一長一短があるので、利用するシステムに適した方法を選択することが望ましい。

4 イベントフレームワーク

4.1 概要

ビジネスロジックの処理方法はさまざまな形態があり、その方法を統合化させることは非常に難しい。しかし、その手順は 1)処理に必要な入力情報の生成、2)入力情報に基づく処理、3)処理結果の返却、という点では共通している。

イベントフレームワークではこれらの一連の流れで共通的な部分をフレームワーク化している。

注意:
本章では、EJB を経由して処理を実行することも可能であることを述べているが、この機能を利用する場合 J2EE アプリケーションサーバが EJB に対応している必要がある。

4.2 構成

4.2.1 構成要素

イベントフレームワークは以下のようなものから構成されている。

- Event
- EventListenerFactory
- EventListener
- EventResult
- EventTrigger

これらの関連を「図 4-1 イベントフレームワークのクラス図」に示す。

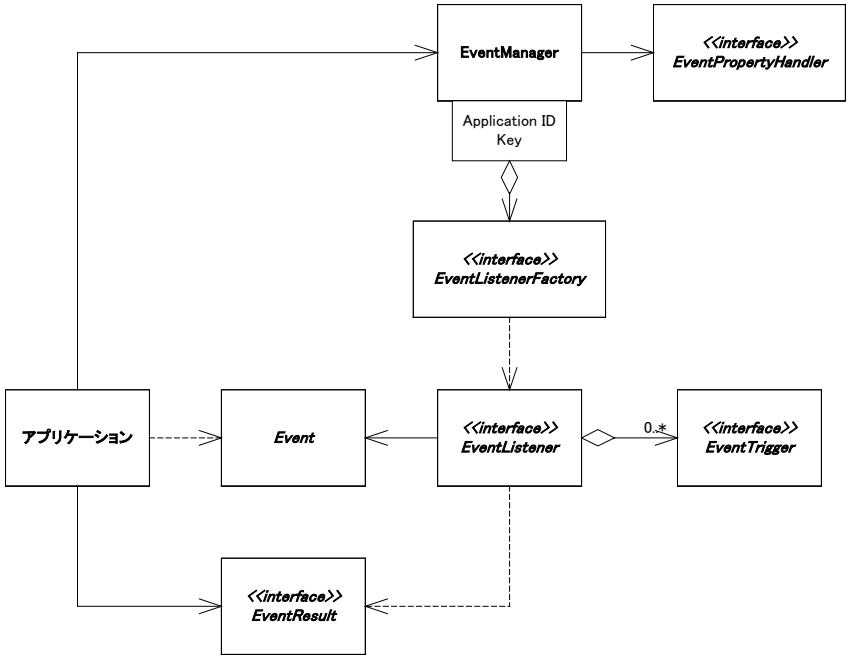


図 4-1 イベントフレームワークのクラス図

4.2.2 イベント処理

IM-JavaEE Framework のイベントフレームワークでビジネスロジックを動かすときの概要を「図 4-2 イベント処理の概要」に示す。

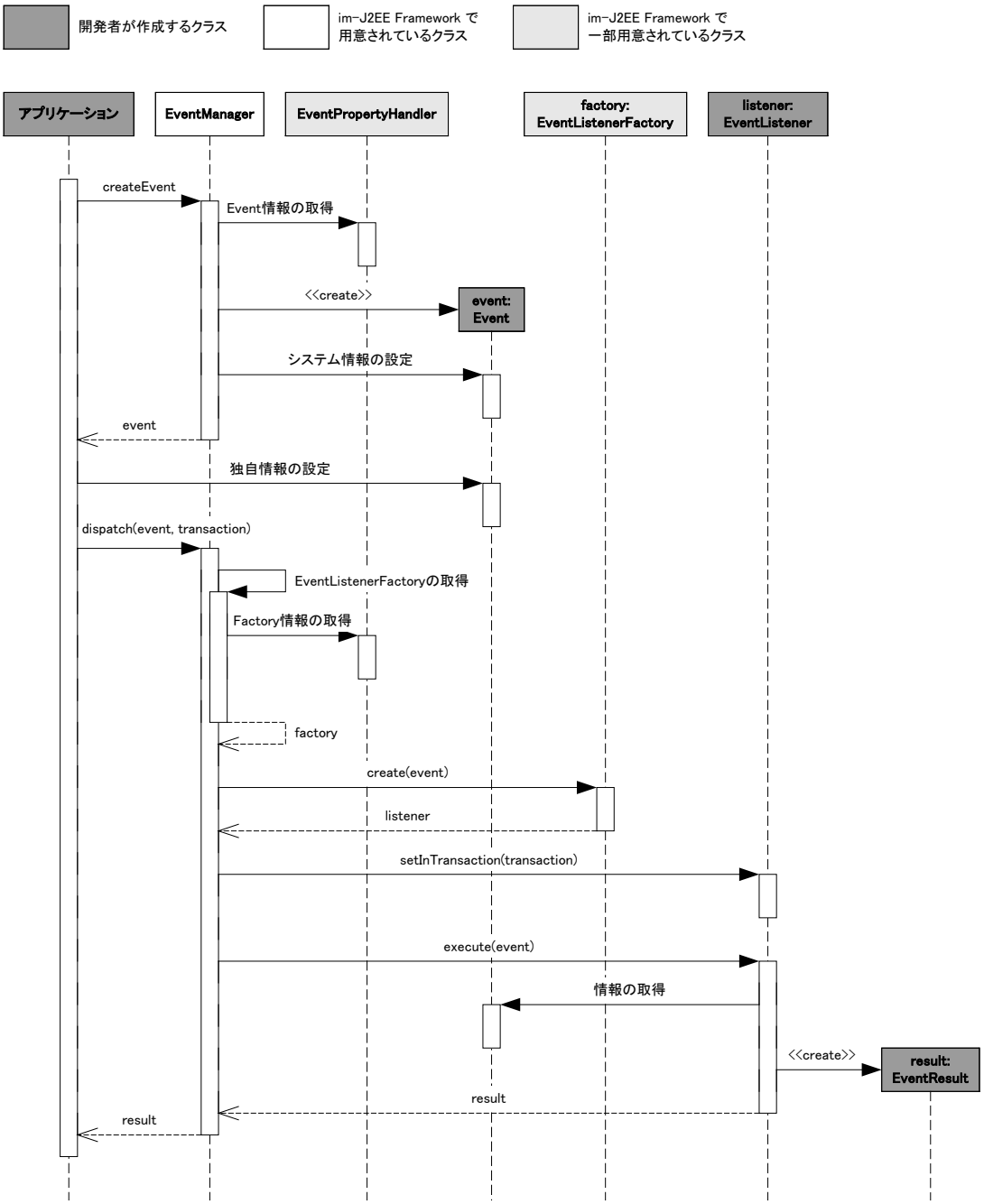


図 4-2 イベント処理の概要

1. アプリケーションは EventManager の createEvent メソッドを使用してアプリケーション ID とイベントキーに対応する Event を生成・取得する。
2. アプリケーションは Event に対してビジネスロジックの処理時に必要となる情報を設定する。
3. アプリケーションは EventManager の dispatch メソッドを使用してビジネスロジックの処理依頼をする。
4. EventManager は EventPropertyHandler を使用してアプリケーション ID とイベントキーに対応する EventListenerFactory を取得する。
5. EventManager は EventListenerFactory の create メソッドを使用して EventListener を取得する。
6. EventManager は EventListener の setInTransaction メソッドを使用してトランザクション内かどうかを判断するフラグを渡す。

7. EventManager は EventListener の execute メソッドを使用して Event を渡し、処理依頼をする。
8. ビジネスロジックである EventListener は処理を行い、結果である EventResult を返す。

4.3 構成要素の詳細

4.3.1 Event

Event はビジネスロジックとなる EventListener の入力情報である。イベント処理の開発者は以下の要件を満たす Event クラスを作成する必要がある。

- jp.co.intra_mart.framework.base.event.Event クラスを継承している。
- public なデフォルトコンストラクタ(引数なしのコンストラクタ)が存在する。
- 以下のフィールド名を持つインスタンス変数が宣言されていない。
  - application
  - key
  - info
- シリアライズ可能であること。EJB などを使う場合、Event はリモート環境に渡される。このとき、シリアライズができないと Event の内容が破壊される恐れがある。

IM-JavaEE Framework のイベントフレームワークを利用してビジネスロジックを実行する場合、Event は EventManager の createEvent メソッドを呼び出して取得する必要がある。開発者は new やリフレクションなどによって Event を直接生成してはならない。EventManager から Event を取得したら、その Event に対してビジネスロジックを起動する際に必要となる入力情報を設定する。これらの流れを「図 4-3 Event の生成」に示す。

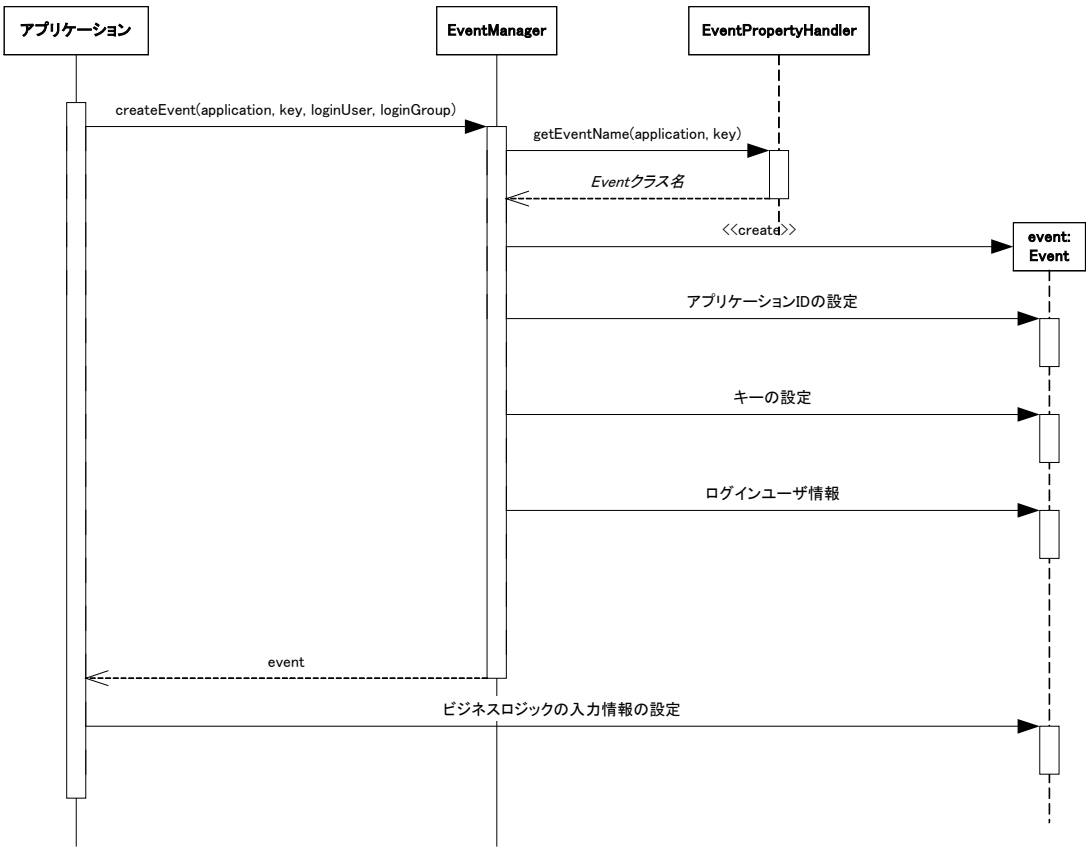


図 4-3 Event の生成

ビジネスロジック(EventListener)がシステムで設定する情報(アプリケーション ID、イベントキー、ログイン情報)以

外の入力情報を必要としない場合、プロパティにはアプリケーション ID とイベントキーに対応する Event について設定する必要はない。この場合、EventPropertyHandler の getEventName メソッドは null を返すように設計されている必要がある(「4.4.4.2.1 イベント」参照)。EventPropertyHandler の getEventName メソッドが null を返した場合、EventManager は jp.co.intra_mart.framework.base.event.EmptyEvent を生成する(「図 4-4 Event の生成(イベントの指定なし)」参照)。

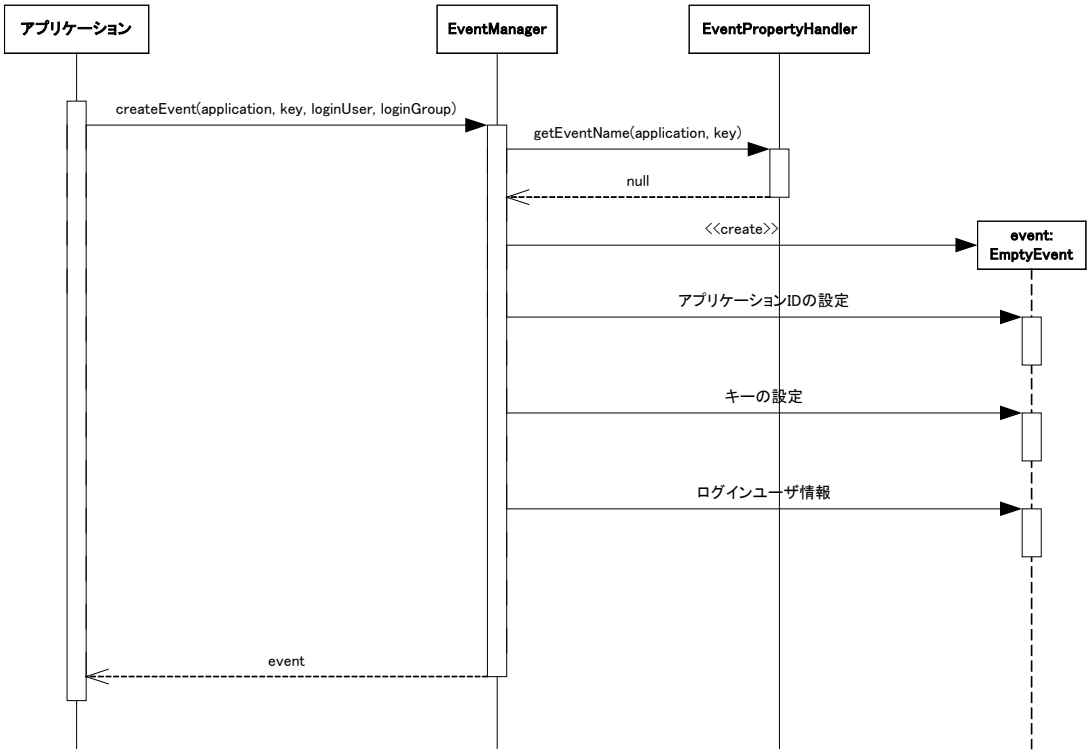


図 4-4 Event の生成(イベントの指定なし)

4.3.2 EventListenerFactory

EventListenerFactory の主な役割は EventListener を生成することである。EventListenerFactory そのものはビジネスロジックの実行とは関係ない。ビジネスロジックの実行は EventListener と密接に関わっている。ここでは EventListenerFactory の存在意義について説明する。

通常、ビジネスロジックを実行する前に何らかの準備をする必要がある。だが、この準備はビジネスロジックの実行形態によって大きく異なる。例えば、ビジネスロジックが同一の Java 仮想マシン(VM)上の Java クラスで実装されている場合、ビジネスロジックの実行前に必要な作業はビジネスロジックのオブジェクトの生成程度であり、後は単純なメソッド呼び出しで行われる。一方、ビジネスロジックが EJB[6]に代表されるようなリモート環境で実行される場合、ビジネスロジックの実行前に必要な作業として Home オブジェクトの取得や Remote オブジェクトの取得などがあり、実行までの手続きは先ほどの Java クラスのメソッドを直接呼び出す場合とは異なる(「図 4-5 ビジネスロジックの実装による違い」参照)。

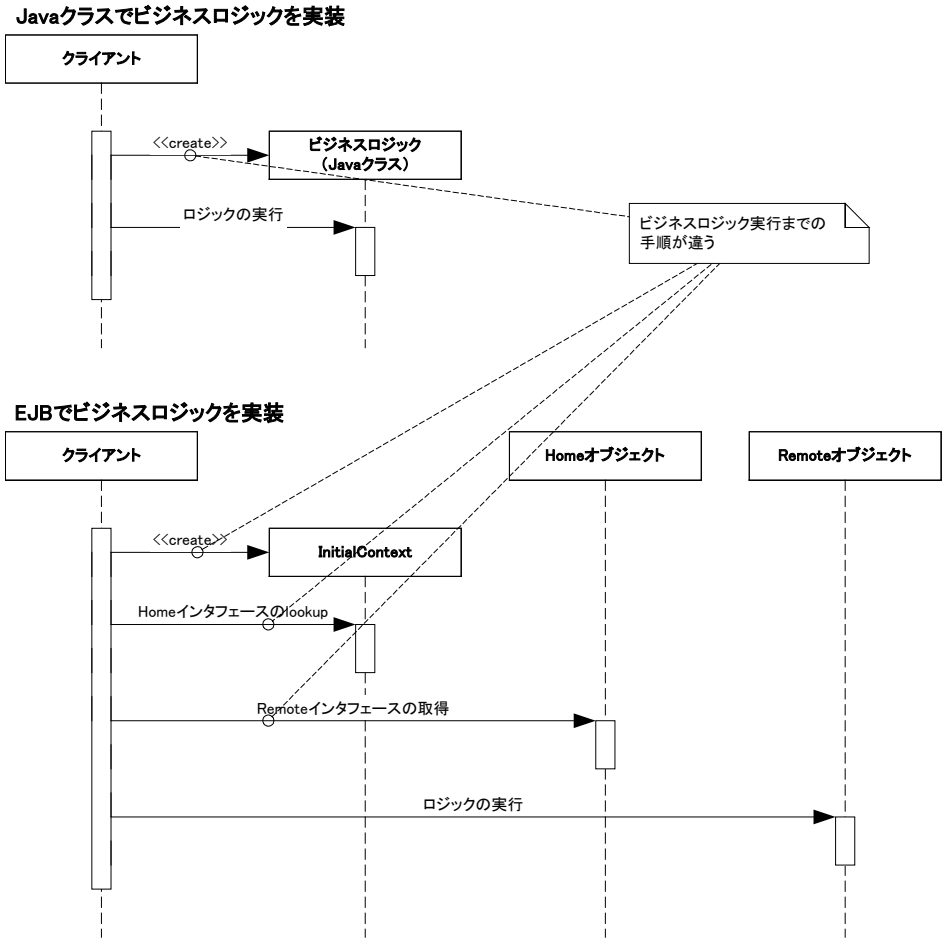


図 4-5 ビジネスロジックの実装による違い

IM-JavaEE Framework では上記の問題を EventListenerFactory と EventListener に分けることで解決している。EventListenerFactory はビジネスロジックへの接続部分の生成と準備を抽象化したものであり、EventListener はビジネスロジックの実行部分である。

EventListenerFactory はアプリケーション ID とイベントキーの組み合わせで 1 つ対応付けられる。EventListenerFactory は動的読み込みが設定されていない場合（EventPropertyHandler の isDynamic メソッドの戻り値が false の場合）内部でキャッシュされる（「図 4-6 EventListenerFactory の生成（キャッシュあり、かつ生成済）」参照）。そうでない場合、EventListenerFactory は毎回生成される（「図 4-7 EventListenerFactory の生成（キャッシュなし、または未生成）」参照）。

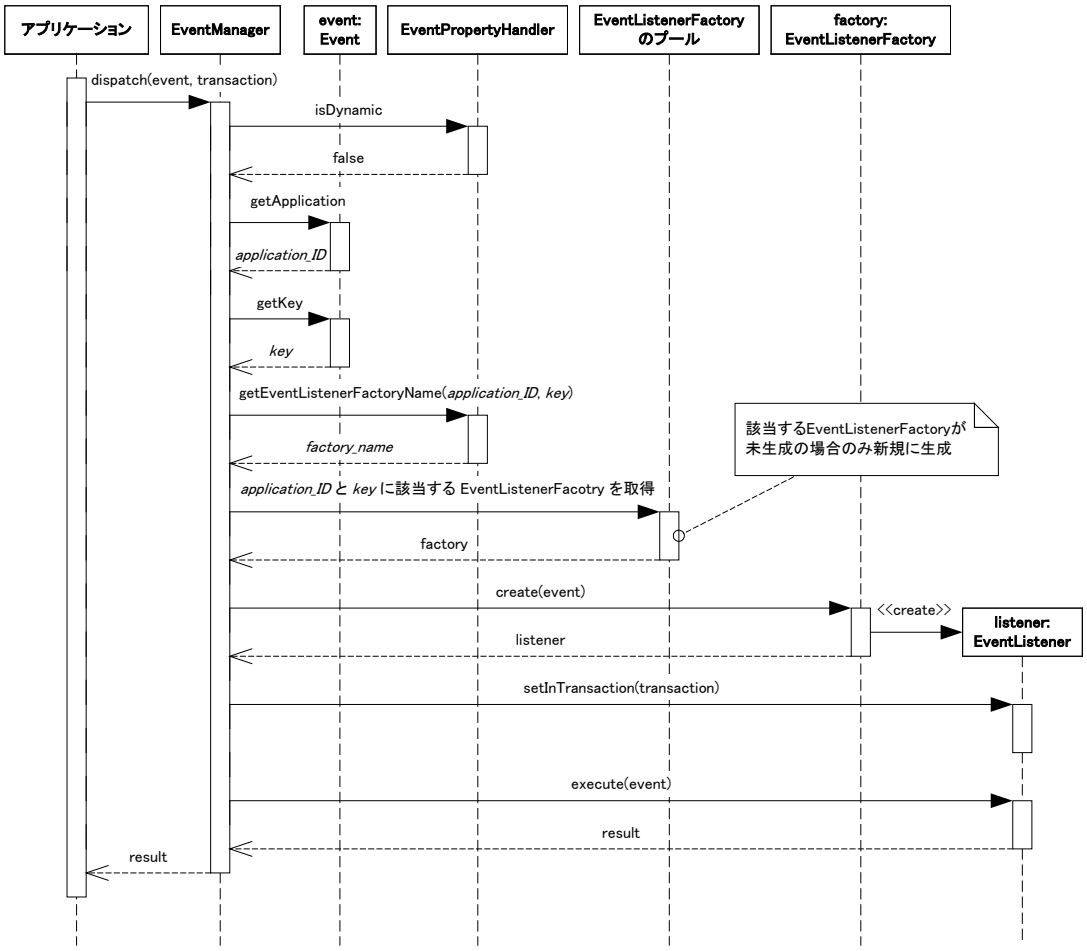


図 4-6 EventListenerFactory の生成（キャッシュあり、かつ生成済）

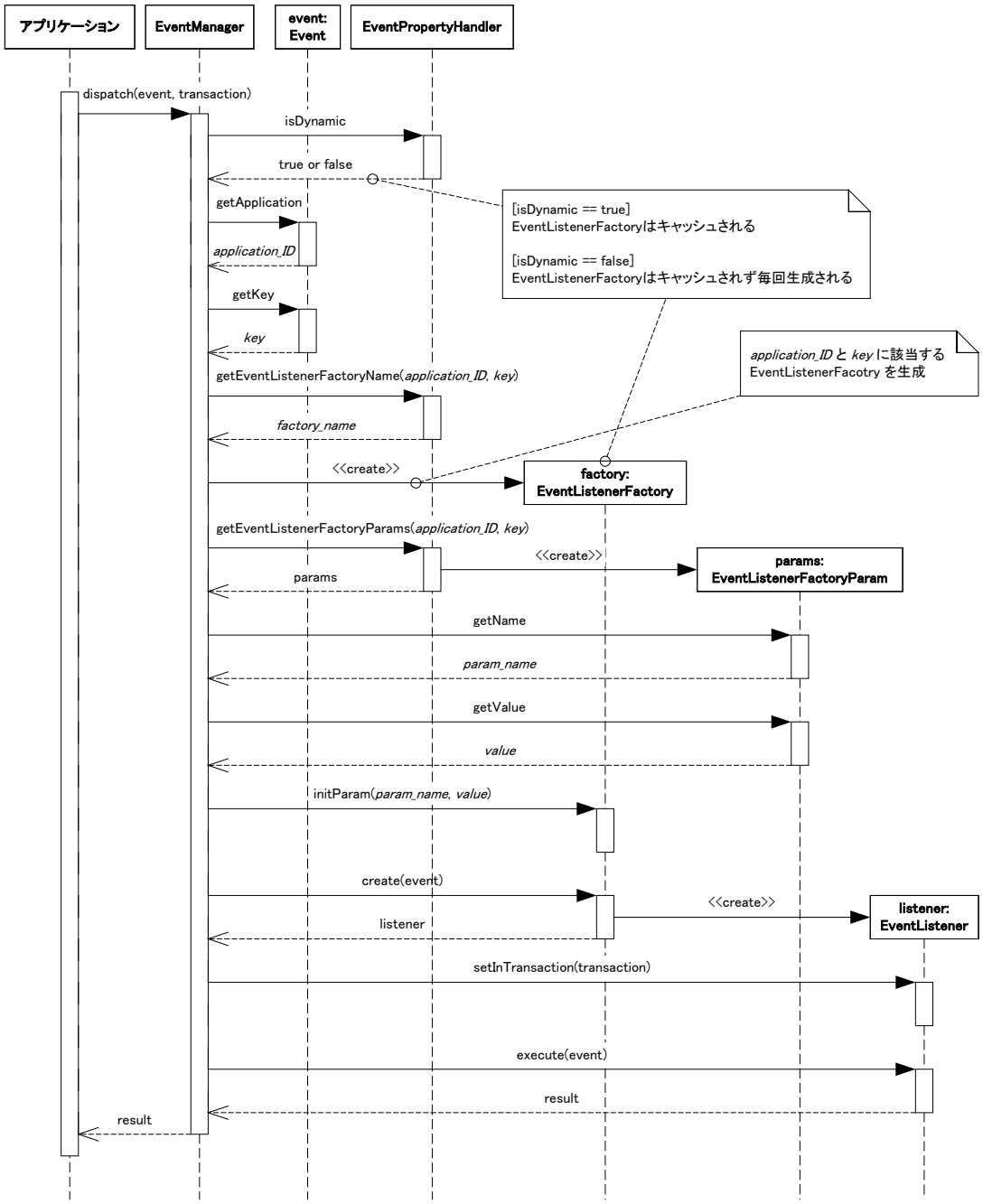


図 4-7 EventListenerFactory の生成（キャッシュなし、または未生成）

4.3.2.1 標準で用意されている EventListenerFactory

EventListenerFactory は開発者が自分で作成することも可能だが、よく使われるものについては最初からいくつか提供されている。最初から提供されている EventListenerFactory は以下のとおりである。

- StandardEventListenerFactory
- GenericEventListenerFactory
- StandardEJBEventListenerFactory
- GenericEJBEventListenerFactory

これらはすべてパッケージ jp.co.intra_mart.framework.base.event に属している。

4.3.2.1.1 StandardEventListenerFactory

StandardEventListenerFactory は StandardEventListener を生成する。StandardEventListenerFactory の create メソッドは呼び出し時に毎回 StandardEventListener のインスタンスを生成する。StandardEventListener のインスタンスはキャッシュされない。「図 4-8 StandardEventListener の生成」を参照。

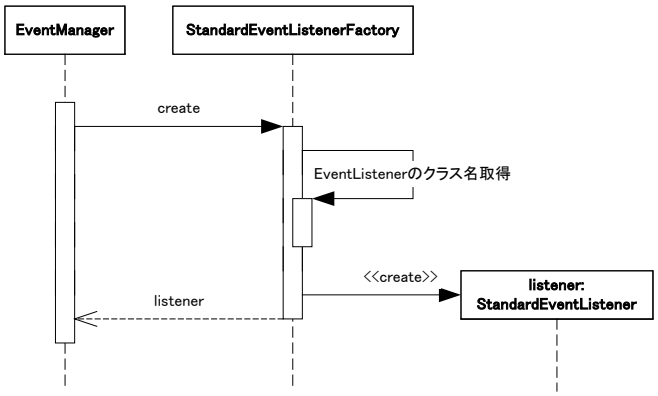


図 4-8 StandardEventListener の生成

このEventListenerFactoryを使用する場合、EventPropertyHandlerのgetEventListenerFactoryParamsメソッドで「表 4-1 StandardEventListenerFactory の初期化パラメータ」に示す初期化パラメータが取得される必要がある。

表 4-1 StandardEventListenerFactory の初期化パラメータ

パラメータ名	パラメータの内容
listener	生成される StandardEventListener のパッケージを含む完全なクラス名。

4.3.2.1.2 GenericEventListenerFactory

GenericEventListenerFactory は GenericEventListener を生成する。GenericEventListenerFactory の create メソッドは呼び出し時に毎回 GenericEventListener のインスタンスを生成する。StandardEventListener のインスタンスはキャッシュされない。「図 4-9 GenericEventListener の生成」を参照。

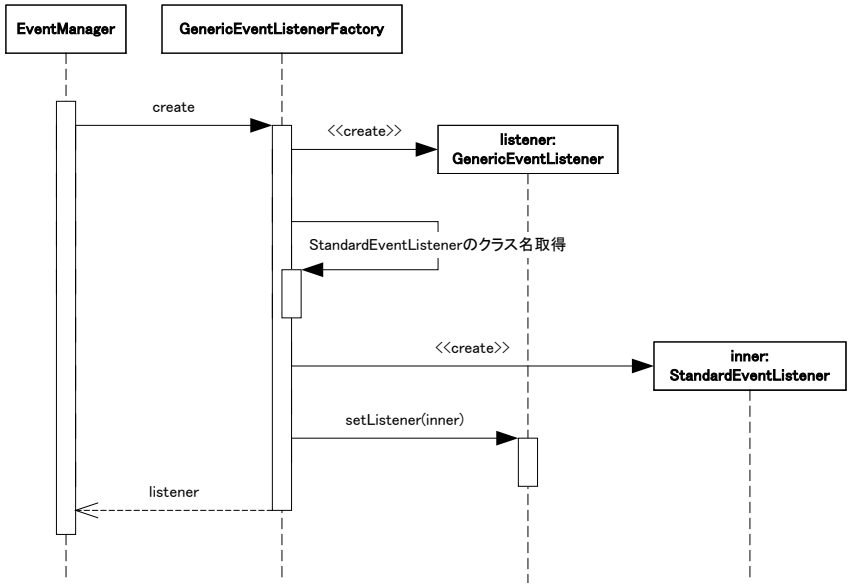


図 4-9 GenericEventListener の生成

このEventListenerFactoryを使用する場合、EventPropertyHandlerのgetEventListenerFactoryParamsメソッドで「表 4-2 GenericEventListenerFactory の初期化パラメータ」に示す初期化パラメータが取得される必要がある。

表 4-2 GenericEventListenerFactory の初期化パラメータ

パラメータ名	パラメータの内容
listener	包含する StandardEventListener のパッケージを含む完全なクラス名。

4.3.2.1.3 StandardEJBEventListenerFactory

StandardEJBEventListenerFactory は StandardEJBEventListener を生成する。一度 create メソッドによって StandardEJBEventListener が生成されると、そのインスタンスは StandardEJBEventListenerFactory のインスタンスにキャッシュされる。「図 4-10 StandardEJBEventListener の生成」を参照。

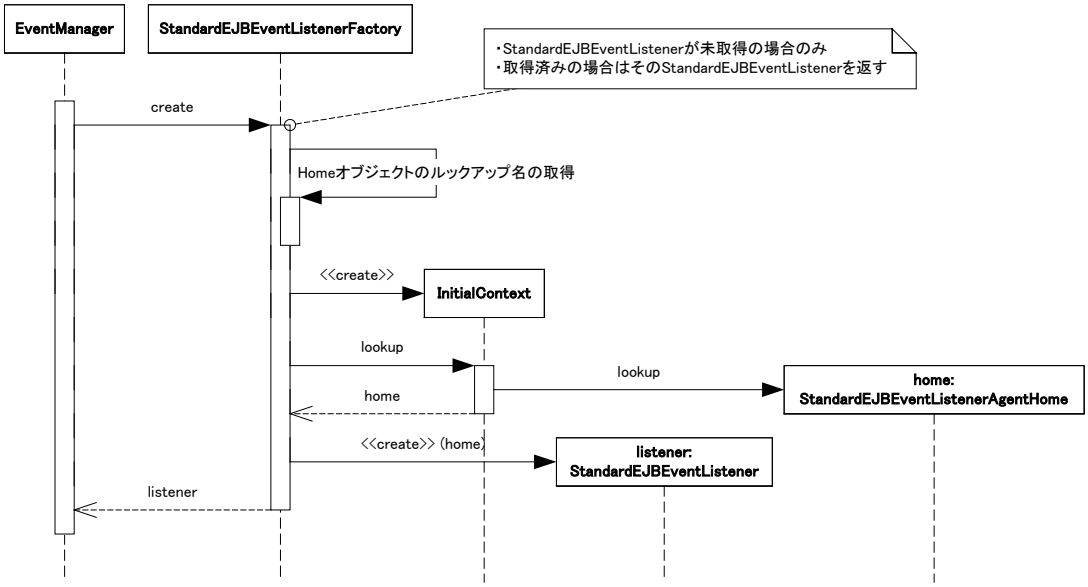


図 4-10 StandardEJBEventListener の生成

「表 4-3 StandardEJBEventListenerFactory の初期化パラメータ」に示す初期化パラメータが EventPropertyHandler の getEventListenerFactoryParams メソッドで取得される必要がある。

表 4-3 StandardEJBEventListenerFactory の初期化パラメータ

パラメータ名	パラメータの内容
home	生成される StandardEJBEventListener で必要とされる EJB の Home オブジェクトをルックアップするときの名前。

4.3.2.1.4 GenericEJBEventListenerFactory

GenericEJBEventListenerFactory は EJB 経由で StandardEventListener を呼び出す際に使用する GenericEJBEventListener を生成する。一度 create メソッドによって GenericEJBEventListener が生成されると、そのインスタンスは GenericEJBEventListenerFactory のインスタンスにキャッシュされる。「図 4-11 GenericEJBEventListener の生成」を参照。

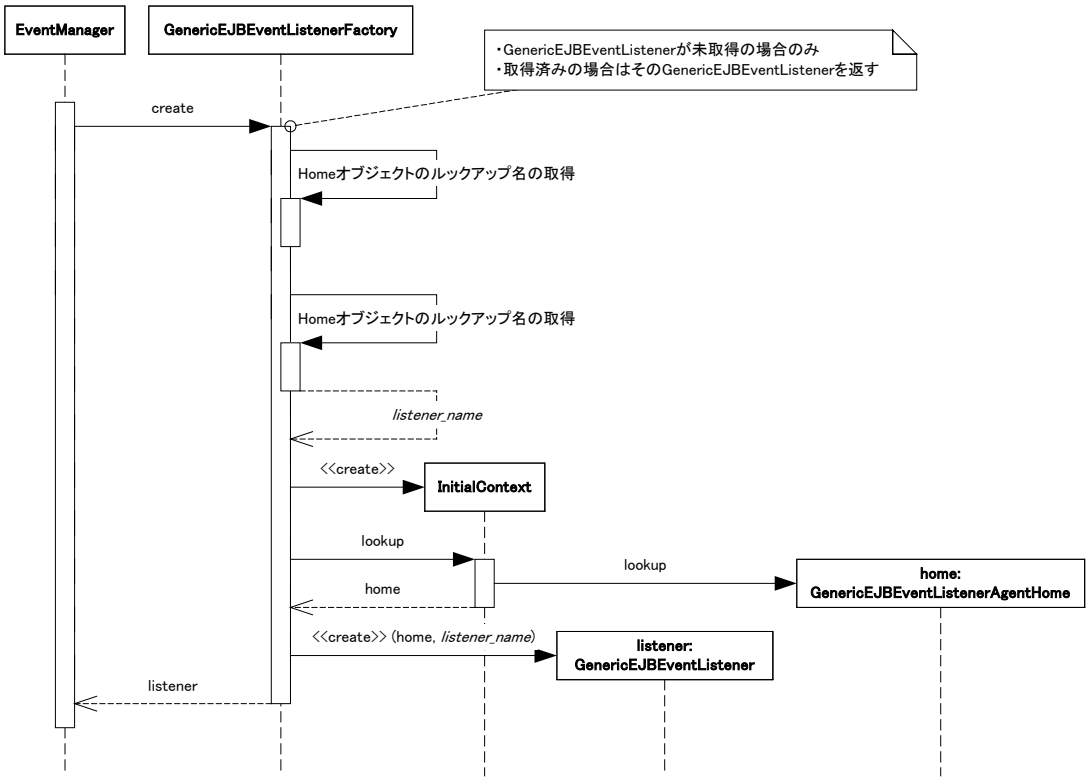


図 4-11 GenericEJBEventListener の生成

「表 4-4 GenericEJBEventListenerFactory の初期化パラメータ」に示すの初期化パラメータが EventPropertyHandler の getEventListenerFactoryParams メソッドで取得される必要がある。

表 4-4 GenericEJBEventListenerFactory の初期化パラメータ

パラメータ名	パラメータの内容
home	生成される GenericEJBEventListener で必要とされる EJB の Home オブジェクトをルックアップするときの名前。
listener	生成される StandardEventListener のパッケージを含む完全なクラス名。

4.3.2.2 独自の EventListenerFactory

EventListenerFactory を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.event.EventListenerFactory インタフェースを実装している。
- public なデフォルトコンストラクタ(引数なしのコンストラクタ)が定義されている。
- create メソッドが EventListener インタフェースを実装した適切なクラスのインスタンスを返す。

独自に開発した EventListenerFactory が上記の条件を満たす場合、IM-JavaEE Framework のイベントフレームワークはその EventListenerFactory を「図 4-6 EventListenerFactory の生成(キャッシュあり、かつ生成済)」や「図 4-7 EventListenerFactory の生成(キャッシュなし、または未生成)」で示したように扱う。

4.3.3 EventListener

EventListener の主な役割はビジネスロジックの実行である。ビジネスロジックの実行は execute メソッドの中で行う。execute メソッドの中では実際のビジネスロジックを実行する前後に EventTrigger(「4.3.4 EventTrigger」参照)を実行するように実装されていることが望ましい。EventListener の処理概要を「図 4-12 EventListener の概要」に示す。

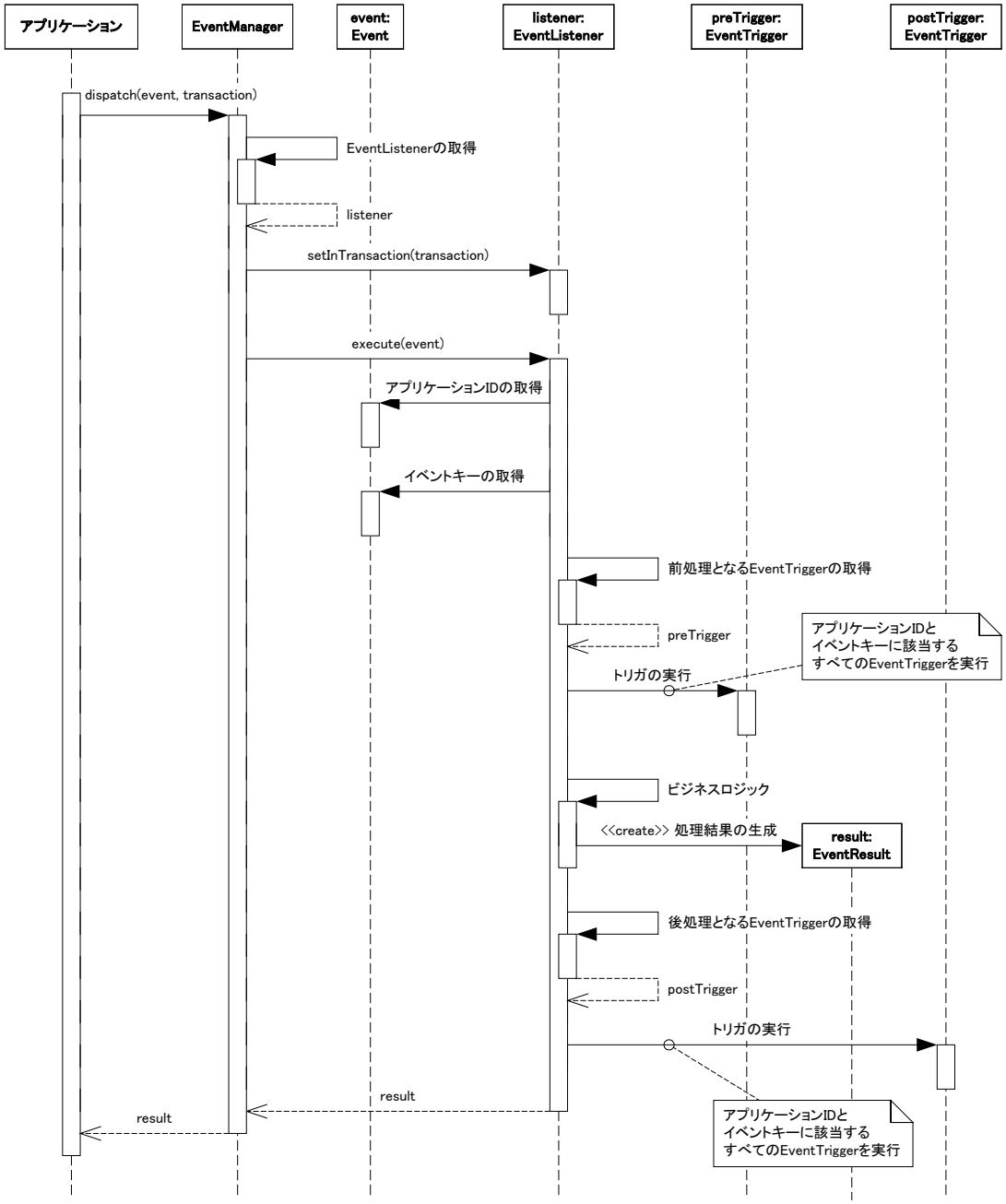


図 4-12 EventListener の概要

4.3.3.1 標準で用意されている抽象 EventListener

EventListener は開発者が自分で作成することも可能だが、よく使われるものについては最初からいくつか抽象クラスとして提供されている。最初から提供されている EventListener は以下のとおりである。

- StandardEventListener
- GenericEventListener
- StandardEJBEventListener
- GenericEJBEventListener

これらはすべてパッケージ jp.co.intra_mart.framework.base.event に属している。

4.3.3.1.1 StandardEventListener

StandardEventListener は最も基本的な機能を持つ EventListener である。開発者はこのクラスを拡張することでビジネスロジックの実装を行うことができる。このクラスは以下のような特徴を持つ。

- EventTrigger の実行に関しては意識する必要がない。
- ビジネスロジックの実装は fire メソッドをオーバライドするだけでよい。
- トランザクションの管理は自動的に行われる。「4.5.1 StandardEventListener」を参照。
- GenericEJBEventListener と組み合わせることにより、EJB を経由してリモート環境で実行することが可能。「4.3.3.1.4 GenericEJBEventListener」を参照。

StandardEventListener の動作を「図 4-13 StandardEventListener の動作」に示す。

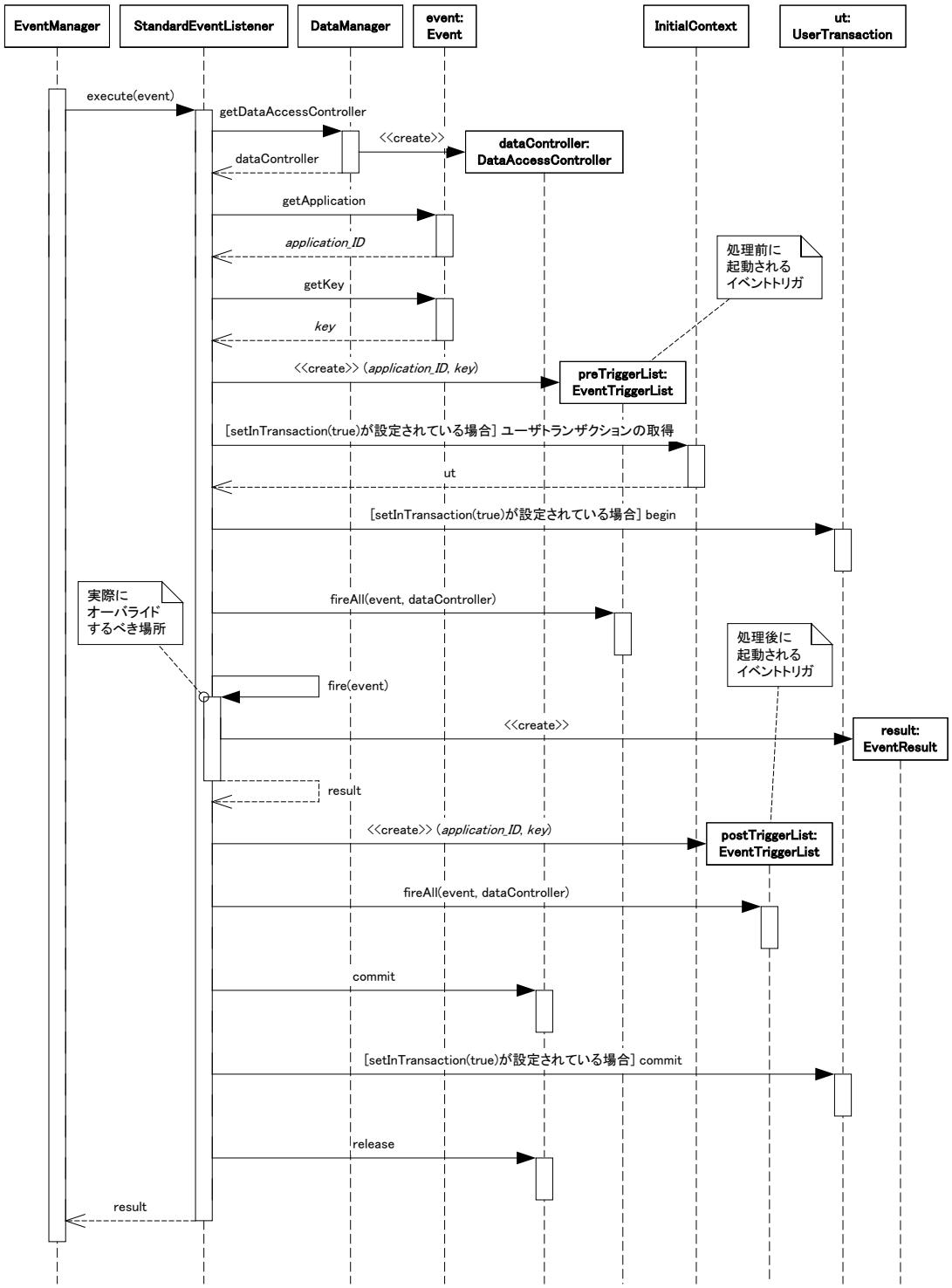


図 4-13 StandardEventListener の動作

4.3.3.1.2 **GenericEventListener**

GenericEventListener は StandardEventListener を単純にラッピングしているだけであり、execute メソッドも StandardEventListener の execute メソッドを呼び出しているだけである。

StandardEventListener の動作を「図 4-14 GenericEventListener の動作」に示す。

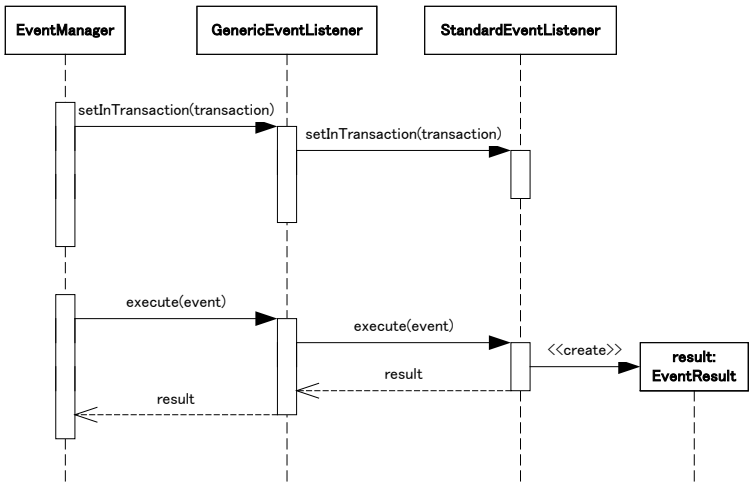


図 4-14 GenericEventListener の動作

4.3.3.1.3 **StandardEJBEventListener**

StandardEJBEventListener は EJB を利用する EventListener である。構造とシーケンス図を「図 4-15 StandardEJBEventListener の構造」と「図 4-16 StandardEJBEventListener の動作(クライアント)」及び「図 4-17 StandardEJBEventListener の動作(EJB サーバ)」に示す。

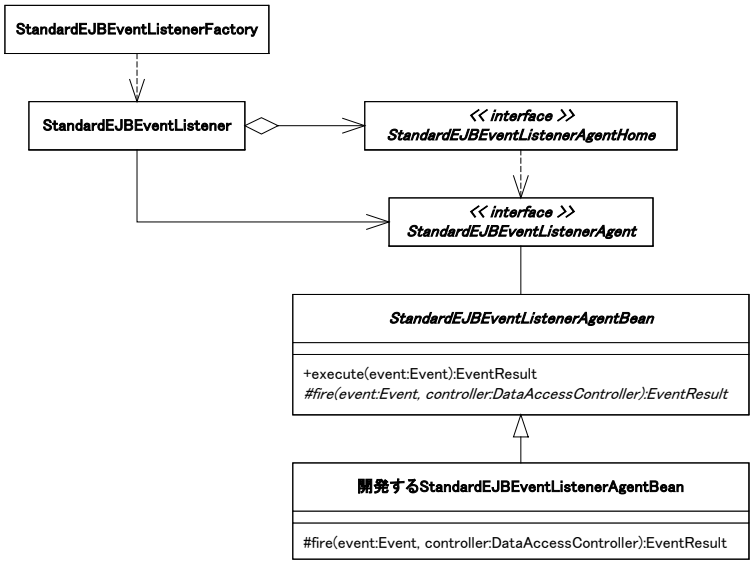


図 4-15 StandardEJBEventListener の構造

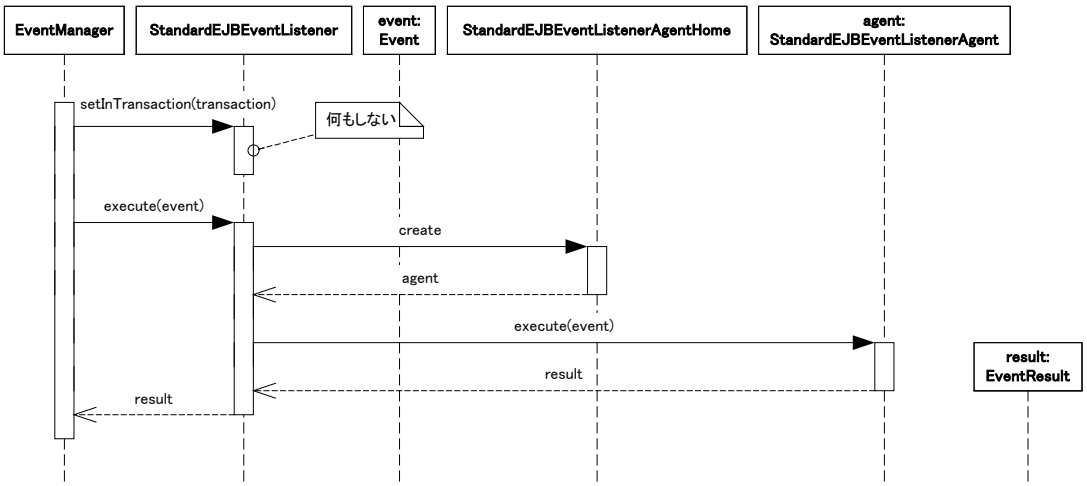


図 4-16 StandardEJBEventListener の動作（クライアント）

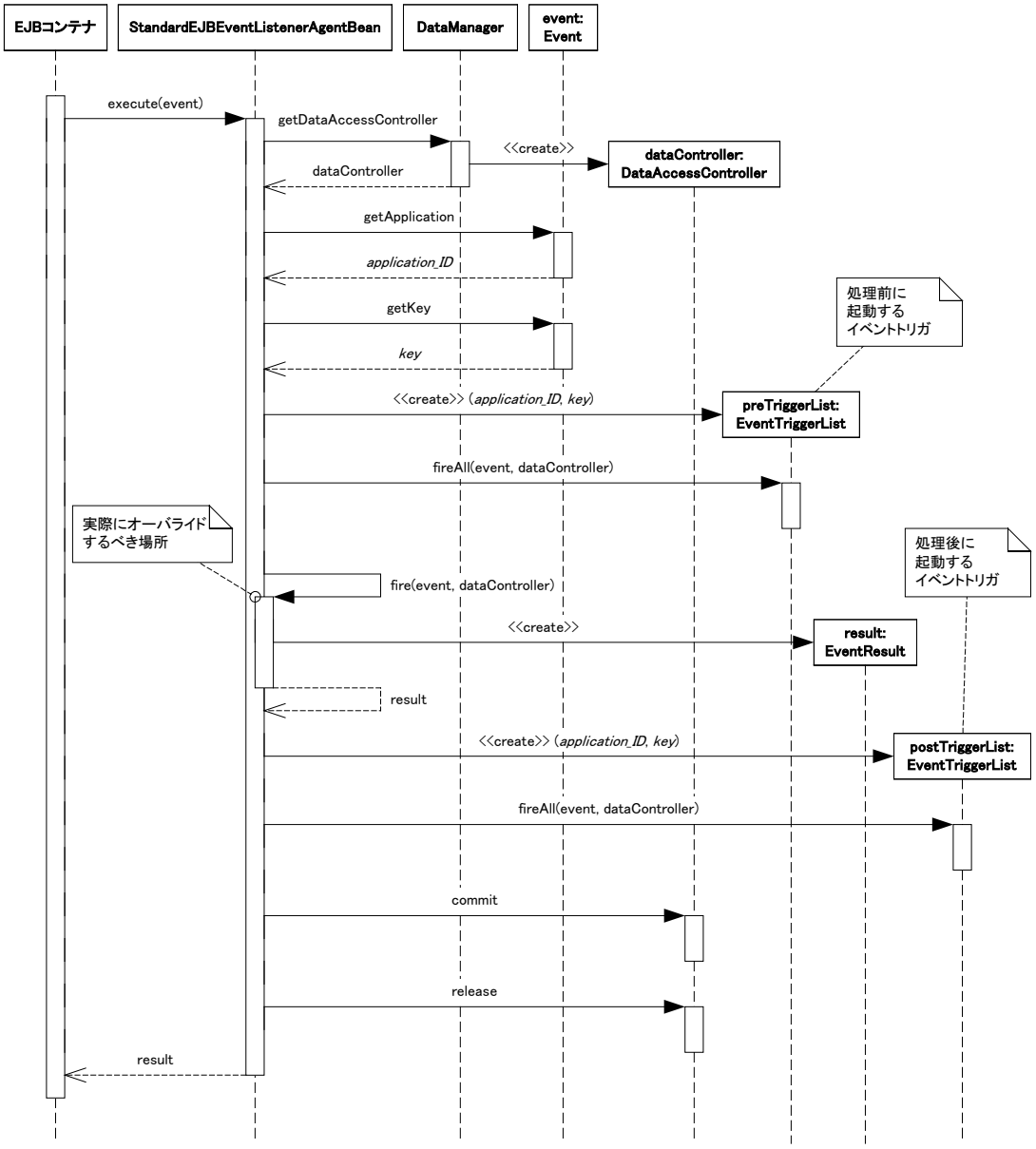


図 4-17 StandardEJBEventListener の動作（EJB サーバ）

4.3.3.1.4 GenericEJBEventListener

GenericEJBEventListener は EJB を経由してリモート環境にある StandardEventListener を実行する EventListener である。構造とシーケンス図を「図 4-18 GenericEJBEventListener の構造」と「図 4-19 GenericEJBEventListener の動作(クライアント)」及び「図 4-20 GenericEJBEventListener の動作(EJB サーバ)」に示す。

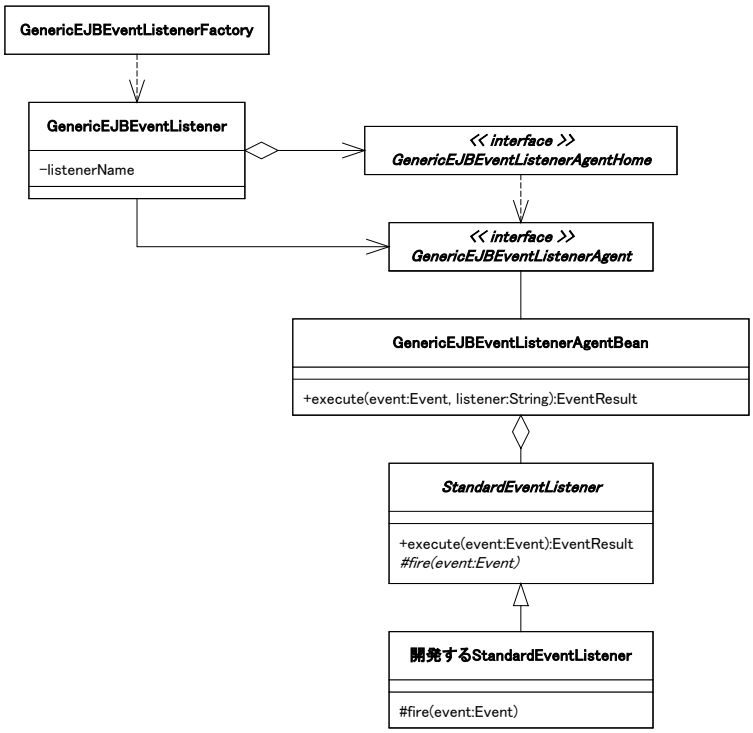


図 4-18 GenericEJBEventListener の構造

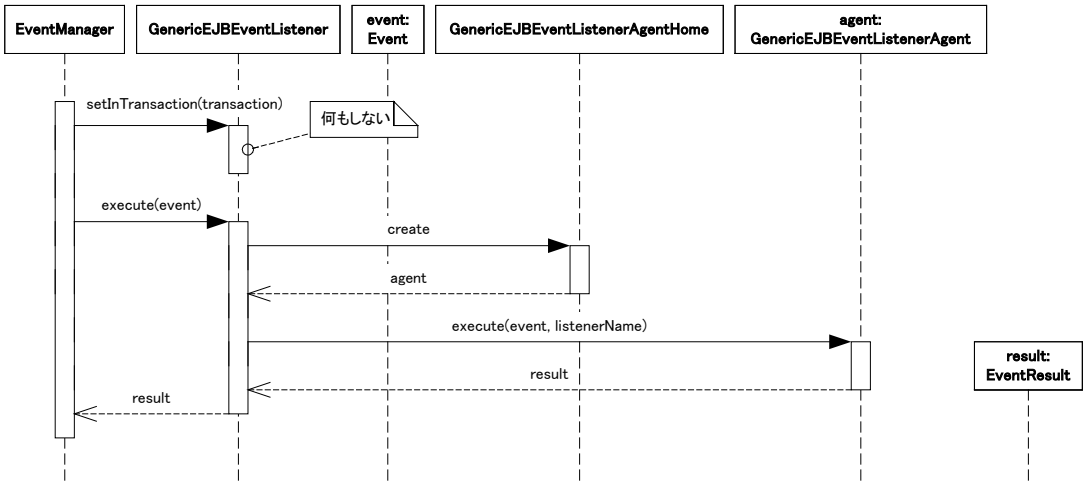


図 4-19 GenericEJBEventListener の動作(クライアント)

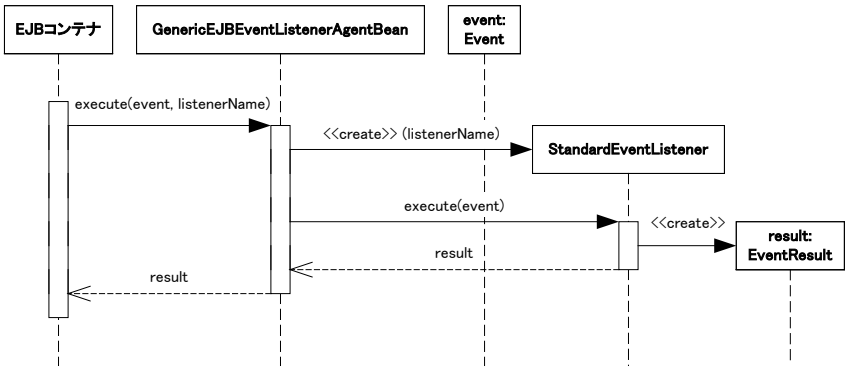


図 4-20 GenericEJBEventListener の動作（EJB サーバ）

4.3.3.2 独自の EventListener

EventListener は受け取ったイベントを処理する。EventListener は以下の要件を満たす必要がある。

- インタフェース jp.co.intra_mart.framework.base.event.EventListener を実装している。
- setInTransaction メソッドの中では、引数の値が true であればトランザクションがすでに開始している場合の準備が、false であればトランザクションがまだ開始されていない場合の準備が必要に応じて実装されている。
- execute メソッドの中では（もし存在すれば）アプリケーション ID とイベントキーに対応するすべての EventTrigger を実処理の前後に行うように実装されている。この場合、以下の要領で EventTrigger の一覧を取得する。
  - アプリケーション ID とイベントキーは引数として渡された event の getApplication メソッドと getKey メソッドで取得される値を使う。
  - 実処理の前に行われる EventTrigger の一覧は EventPropertyHandler の getEventTriggerInfos メソッドで取得する。
  - 実処理の後に行われる EventTrigger の一覧は EventPropertyHandler の getPostEventTriggerInfos メソッドで取得する。
  - 必要に応じてトランザクションに関連する処理（トランザクションの開始、コミット、ロールバック等）を行う。この場合、setInTransaction メソッドで行われた準備を反映しているトランザクション処理が望ましい。

4.3.4 EventTrigger

EventTrigger は EventListener が実行される前後に行われる処理内容である。EventTrigger は以下の点で EventListener とは異なる。

- 同一のアプリケーション ID とイベントキーに対して複数設定できる。
- 処理結果（EventResult）を返さない。

EventTrigger は以下の条件を満たしている必要がある。

- インタフェース jp.co.intra_mart.framework.base.event.EventTrigger を実装している。
- fire メソッドが適切な処理を行うように記述されている。

EventTrigger インタフェースを実装した jp.co.intra_mart.framework.base.event.EventTriggerAdapter 抽象クラスを利用すると、StandardEventListener と同様に fire(Event)メソッドの中で getDAO メソッドを使用することができる。このメソッドで取得される DAO は fire(Event, DataAccessController)メソッドで渡される DataAccessController の getDAO メソッドの戻り値と同じである。

4.3.5 EventResult

EventResult は EventListener の処理結果である。EventResult は以下の条件を満たしている必要がある。

- jp.co.intra_mart.framework.base.event.EventResult インタフェースを実装している。
- シリアライズ可能であること。EJB などを使う場合、EventResult はリモート環境からもとの環境に渡される。このとき、シリアライズができないと EventResult の内容が破壊される恐れがある。

4.4 イベントに関連するプロパティ

IM-JavaEE Framework のイベントフレームワークではさまざまなプロパティを外部で設定することが可能である。イベントプロパティの取得は jp.co.intra_mart.framework.base.event.EventPropertyHandler インタフェースを実装したクラスから取得する。IM-JavaEE Framework ではこのインタフェースを実装した複数の実装クラスを標準で提供している(「図 4-21 EventPropertyHandler」を参照)。イベントプロパティの設定方法は IM-JavaEE Framework では特に規定してなく、前述のインタフェースを実装したクラスに依存する。

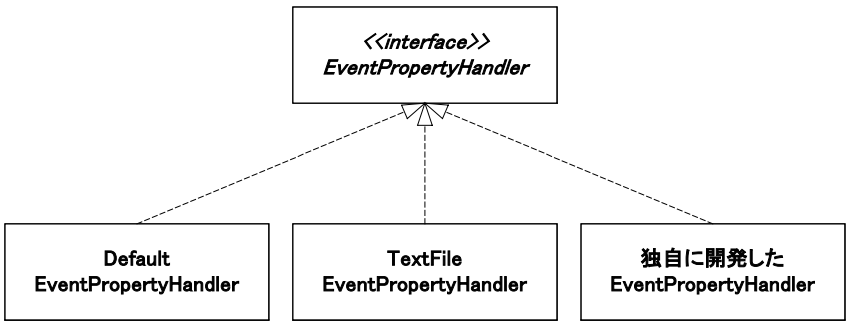


図 4-21 EventPropertyHandler

4.4.1 イベントに関連するプロパティの取得

イベントに関連するプロパティは EventPropertyHandler から取得する。EventPropertyHandler は jp.co.intra_mart.framework.base.event.EventManager の getEventPropertyHandler メソッドで取得することができる。EventPropertyHandler は必ずこのメソッドを通じて取得されたものである必要があり、開発者が自分でこの EventPropertyHandler の実装クラスを明示的に生成(new による生成や java.lang.Class の newInstance メソッド、またはリフレクションを利用したインスタンスの生成)をしてはならない。

EventPropertyHandler の取得とプロパティの取得に関連する手順を「図 4-22 EventPropertyHandler の取得」に示す。

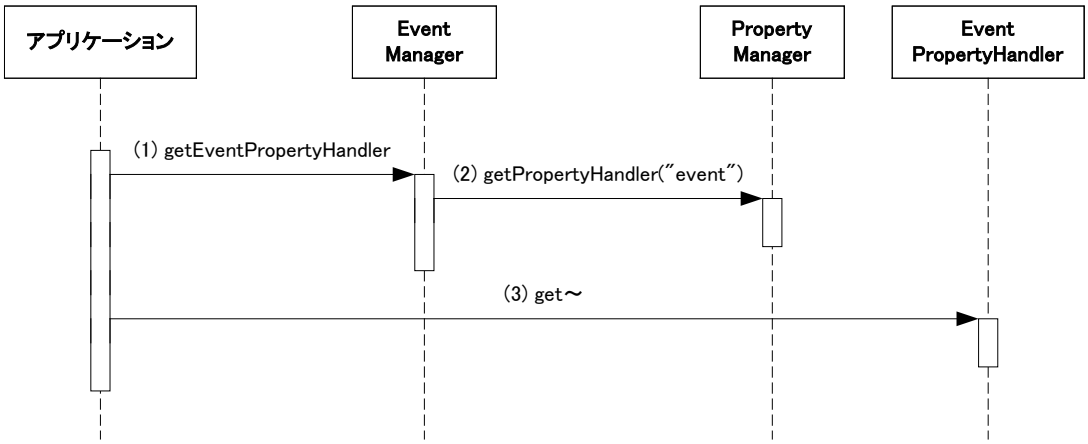


図 4-22 EventPropertyHandler の取得

1. EventManager から EventPropertyHandler を取得する。
2. EventManager の内部では PropertyManager から EventPropertyHandler を取得し、アプリケーションに返している。この部分は EventManager の内部で行っていることであり、開発者は特に意識する必要はない。
3. EventPropertyHandler を利用して各種のプロパティを取得する。

4.4.2 標準で用意されている EventPropertyHandler

IM-JavaEE Framework では jp.co.intra_mart.framework.base.event.EventPropertyHandler を実装したクラスをいくつか提供している。それぞれ設定方法やその特性が違うため、運用者は必要に応じてこれらを切り替えることができる。

4.4.2.1 DefaultEventPropertyHandler

jp.co.intra_mart.framework.base.event.DefaultEventPropertyHandler として提供されている。

プロパティの設定はリソースファイルで行う。リソースファイルの内容は「プロパティ名=プロパティの値」という形式で設定する。使用できる文字などは java.util.ResourceBundle に従う。このリソースファイルは使用するアプリケーションから取得できるクラスパスにおく必要がある。リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

4.4.2.2 TextFileEventPropertyHandler

jp.co.intra_mart.framework.base.event.TextFileEventPropertyHandler として提供されている。

DefaultEventPropertyHandler と同じ形式のリソースファイルを利用するが、以下の点が違う。

- クラスパスに通す必要がない。
- アプリケーションから参照できる場所であれば、ファイルシステムの任意の場所に配置できる。
- 設定によってはアプリケーションを停止しないでリソースファイルの再読み込みが可能となる。

リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

4.4.2.3 XmlEventPropertyHandler

jp.co.intra_mart.framework.base.event.XmlEventPropertyHandler として提供されている。
プロパティの設定は XML 形式で行う。アプリケーションから取得できるクラスパスにおく必要があり、固有の ID と Java パッケージパスを含めたものをアプリケーション ID として認識する。例えばアプリケーション ID が ”foo.bar.example”の場合、クラスパスに ”foo/bar/event-config-example.xml”として配置する。
また、XmlEventPropertyHandler は動的読み込みに対応している。
詳細は API リストを参照。

4.4.3 独自の EventPropertyHandler

EventPropertyHandler を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.event.EventPropertyHandler インタフェースを実装している。
- public なデフォルトコンストラクタ(引数なしのコンストラクタ)が定義されている。
- すべてのメソッドに対して適切な値が返ってくる。(「4.4.4 プロパティの内容」参照)
- isDynamic()メソッドが false を返す場合、プロパティを取得するメソッドはアプリケーションサーバを再起動しない限り値は変わらない。

4.4.4 プロパティの内容

イベントに関連するプロパティの設定方法は運用時に使用する EventPropertyHandler の種類によって違うが、概念的には同じものである。

イベントに関連するプロパティの内容は以下のとおりである。

4.4.4.1 共通

4.4.4.1.1 動的読み込み

isDynamic()メソッドで取得可能。

このメソッドの戻り値が true である場合、このインタフェースで定義される各プロパティ取得メソッド(get～メソッド)は毎回設定情報を読み込みに行くように実装されている必要がある。false である場合、各プロパティ取得メソッドはパフォーマンスを考慮して取得される値を内部でキャッシュしてもよい。

4.4.4.2 アプリケーション個別

4.4.4.2.1 イベント

getEventName(String application, String key)メソッドで取得可能。

アプリケーション ID とイベントキーに対応するイベントの完全なクラス名を設定する。未設定の場合、null を返す。
ここで指定するクラスは jp.co.intra_mart.framework.base.event.Event クラスを拡張している必要がある。

4.4.4.2.2 EventListenerFactory

getEventListenerFactoryName(String application, String key)メソッドで取得可能。

アプリケーション ID とイベントキーに対応する EventListenerFactory の完全なクラス名を設定する。未設定の場合、jp.co.intra_mart.framework.base.event.EventPropertyException が throw される。ここで指定するクラスは jp.co.intra_mart.framework.base.event.EventListenerFactory インタフェースを実装している必要がある。

4.4.4.2.3 EventListenerFactory の初期パラメータ

getEventListenerFactoryParams(String application, String key)メソッドで取得可能。

アプリケーション ID とイベントキーに対応する EventListenerFactory の初期パラメータを設定する。設定されていない場合、サイズが 0 の配列を返す。

4.4.4.2.4 EventTrigger 情報（イベント処理前に起動）

getEventTriggerInfos(String application, String key)メソッドで取得可能。

アプリケーション ID とイベントキーに対応する EventTrigger 情報を設定する。ここで取得される EventTrigger はイベント処理前に起動される。返される情報は以下の条件を満たす必要がある。

- jp.co.intra_mart.framework.base.event.EventTriggerInfo のコレクションである。
- 返されるコレクションの iterator メソッドでは EventTriggerInfo が設定された順番に取得することが可能である。
- EventTriggerInfo は以下のような情報を含んでいる。
  - アプリケーション ID とイベントキーの組み合わせの中で一意となる順番（getNumber メソッドで取得可能）。
  - EventTrigger のパッケージ名を含んだ完全なクラス名（getName メソッドで取得可能）。

設定されていない場合、内容が空の Collection が返される。

4.4.4.2.5 EventTrigger 情報（イベント処理後に起動）

getPostEventTriggerInfos(String application, String key)メソッドで取得可能。

アプリケーション ID とイベントキーに対応する EventTrigger 情報を設定する。ここで取得される EventTrigger はイベント処理後に起動される。返される情報は以下の条件を満たす必要がある。

- jp.co.intra_mart.framework.base.event.EventTriggerInfo のコレクションである。
- 返されるコレクションの iterator メソッドでは EventTriggerInfo が設定された順番に取得することが可能である。
- EventTriggerInfo は以下のような情報を含んでいる。
  - アプリケーション ID とイベントキーの組み合わせの中で一意となる順番（getNumber メソッドで取得可能）。
  - EventTrigger のパッケージ名を含んだ完全なクラス名（getName メソッドで取得可能）。

設定されていない場合、内容が空の Collection が返される。

4.5 トランザクション

IM-JavaEE Framework のイベントフレームワークに標準で用意されている StandardEventListener と StandardEJBEventListener はトランザクション機能を備えている。これらは基本的に execute(Event event)メソッドの単位でトランザクションを開始・終了している。これらを利用して EventListener のコンポーネントを製造する場合、fire メソッドの実装方法を以下のようにすればよい：

- トランザクションをコミットしたい場合は通常処理で終了する。
- トランザクションをロールバックさせたい場合は例外を throw する。

4.5.1 StandardEventListener

StandardEventListener の内部でトランザクションを管理している様子を「図 4-23 StandardEventListener のトランザ

クション（内部で開始）」に示す。

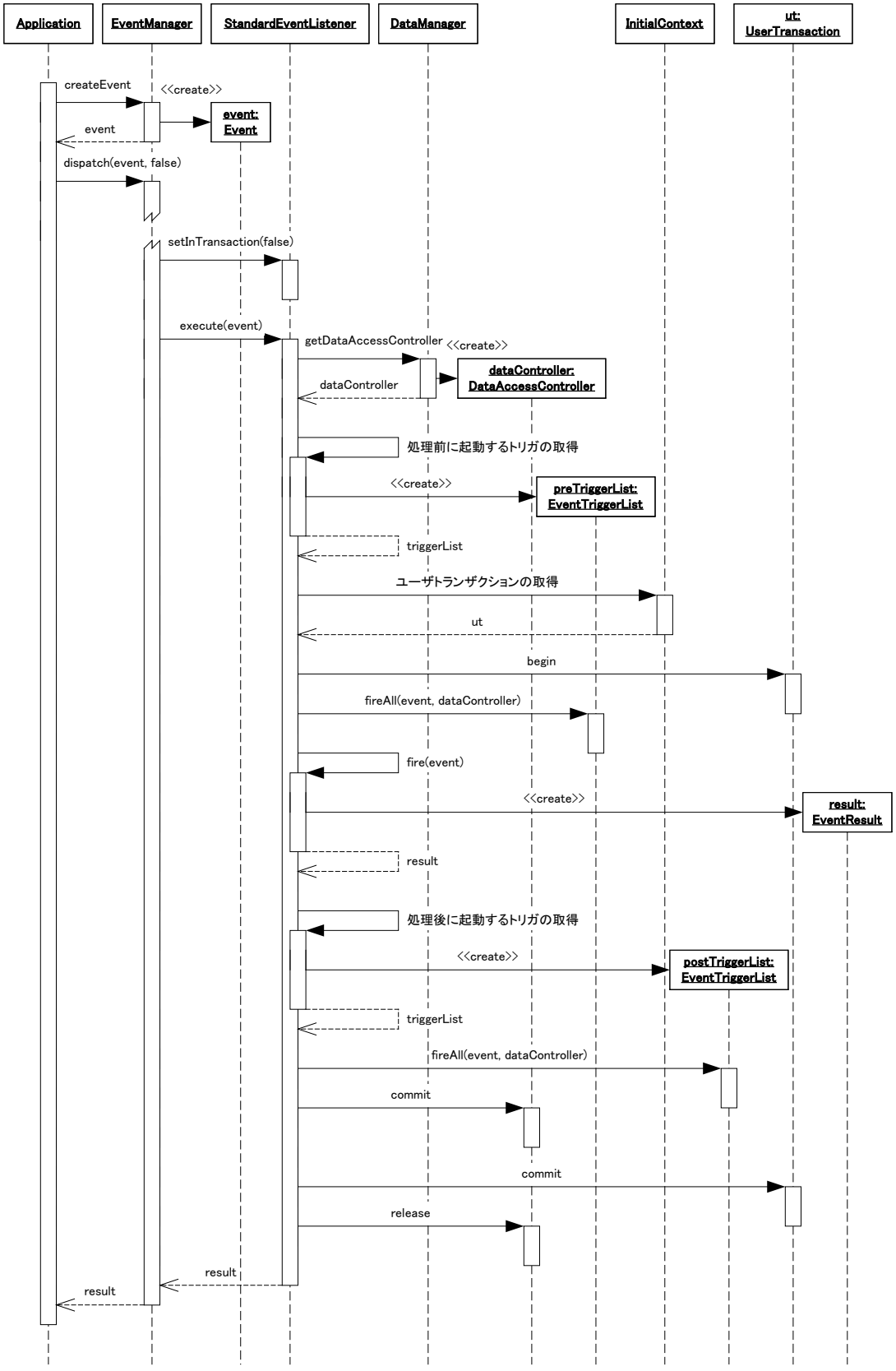


図 4-23 StandardEventListener のトランザクション（内部で開始）

StandardEventListener の外部でトランザクションを管理している様子を「図 4-24 StandardEventListener のトランザ

クション（外部で開始）」に示す。

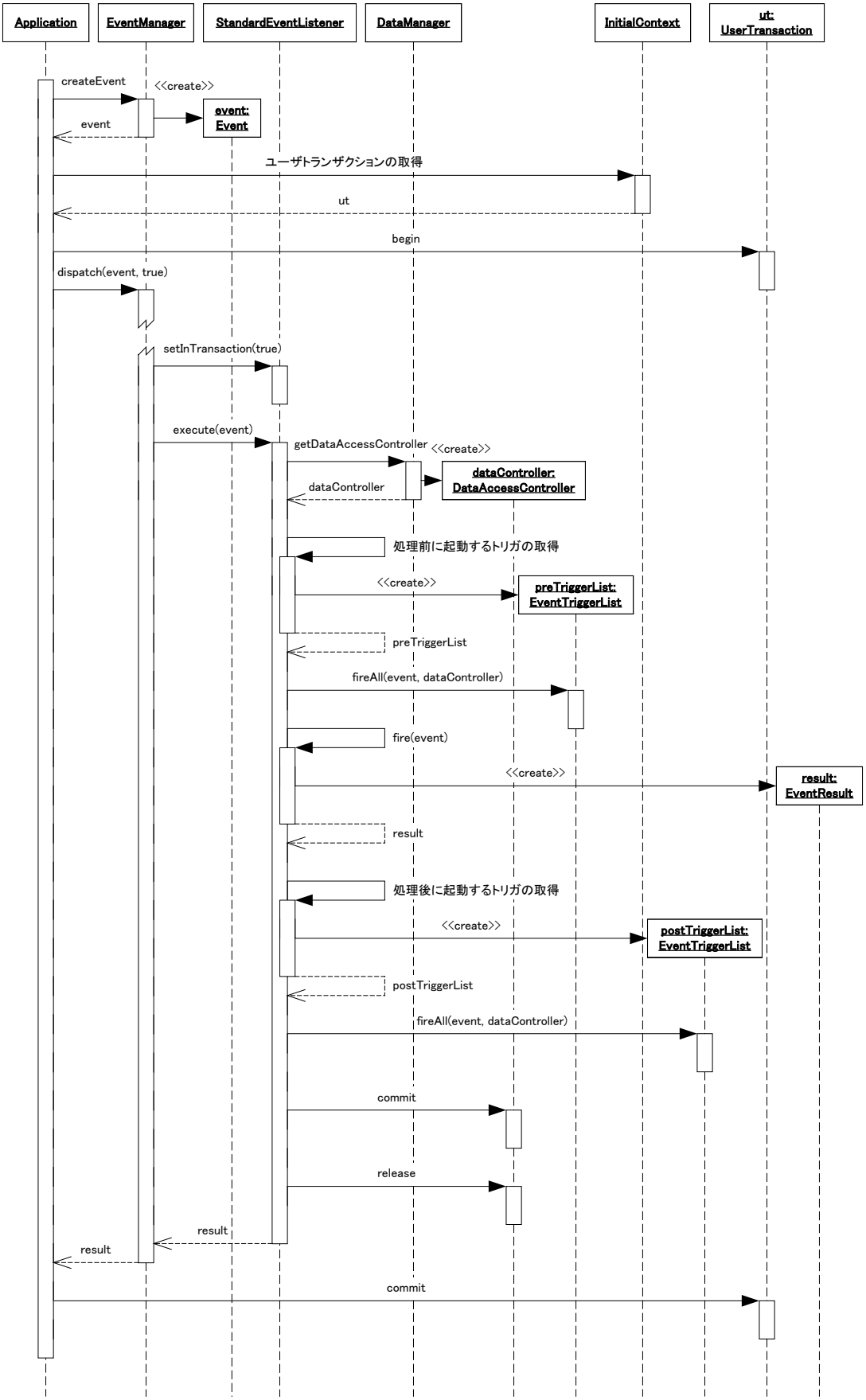


図 4-24 StandardEventListener のトランザクション（外部で開始）

「図 4-23 StandardEventListener のトランザクション（内部で開始）」や「図 4-24 StandardEventListener のトランザクション（外部で開始）」を見ると commit が DataAccessController と UserTransaction の 2 箇所で行われていることがわかる（DataAccessController で行うトランザクション管理については「5.5 トランザクション」を参照）。これは DataAccessController と UserTransaction は管理体系が異なるトランザクションであるため両者は同一のトランザクションにはならないからである。そのため、データフレームワークでデータベースなどを扱う場合、DataAccessController ではなく UserTransaction で扱われる DataSource でアクセスすることを推奨する。

StandardEventListener は fire メソッド内部から他の EventListener をネストして呼び出すことが可能である。この場合、StandardEventListener の dispatchEvent メソッドを使用する。このときの様子を「図 4-25 StandardEventListener から他の EventListener の呼び出し」に示す。

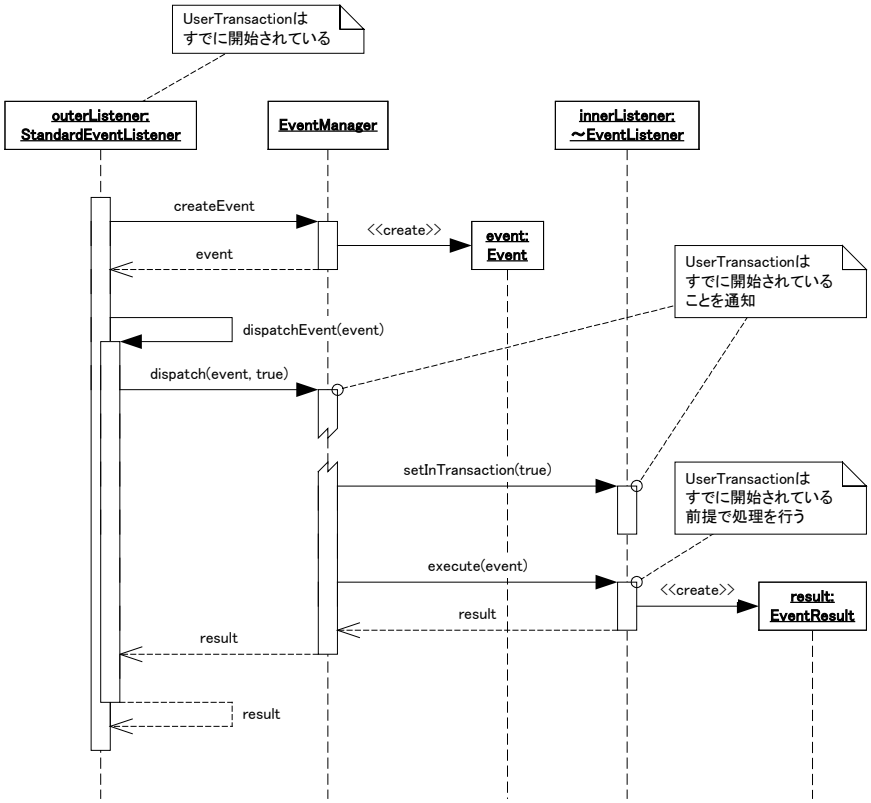


図 4-25 StandardEventListener から他の EventListener の呼び出し

StandardEventListener から StandardEventListener や StandardEJBEventListener を呼び出すとき、1 つ注意すべき点がある。「図 4-23 StandardEventListener のトランザクション（内部で開始）」、「図 4-24 StandardEventListener のトランザクション（外部で開始）」、「図 4-25 StandardEventListener から他の EventListener の呼び出し」を連結した形で見ると、DataAccessController（DataAccessController の詳細は「5.3.2 DataAccessController」を参照）が外部の StandardEventListener と内部の StandardEventListener の両方で生成されることがわかる。これは、DataAccessController で管理されるトランザクションはそれぞれ別々に生成されることを意味する。つまり StandardEventListener をネストした場合、DataAccessController で管理されるデータコネクタ（例えば JDBCConnector など、データコネクタの詳細は「5.3.3 DataConnector」を参照）が使用されている場合、1 つのトランザクションにはまとめられない可能性がある。この問題を避けるためにも、データフレームワークでデータベースなどを扱う場合、DataAccessController ではなく UserTransaction でトランザクションが扱われる DataSource でアクセスすることを推奨する。

4.5.2 GenericEventListener

GenericEventListener では特にトランザクションの管理は行っていない。トランザクションの管理方法は包含している StandardEventListener に委ねられている。

4.5.3 StandardEJBEventListener

StandardEJBEventListener でトランザクションを管理している様子を「図 4-26 StandardEJBEventListener のトランザクション」に示す。

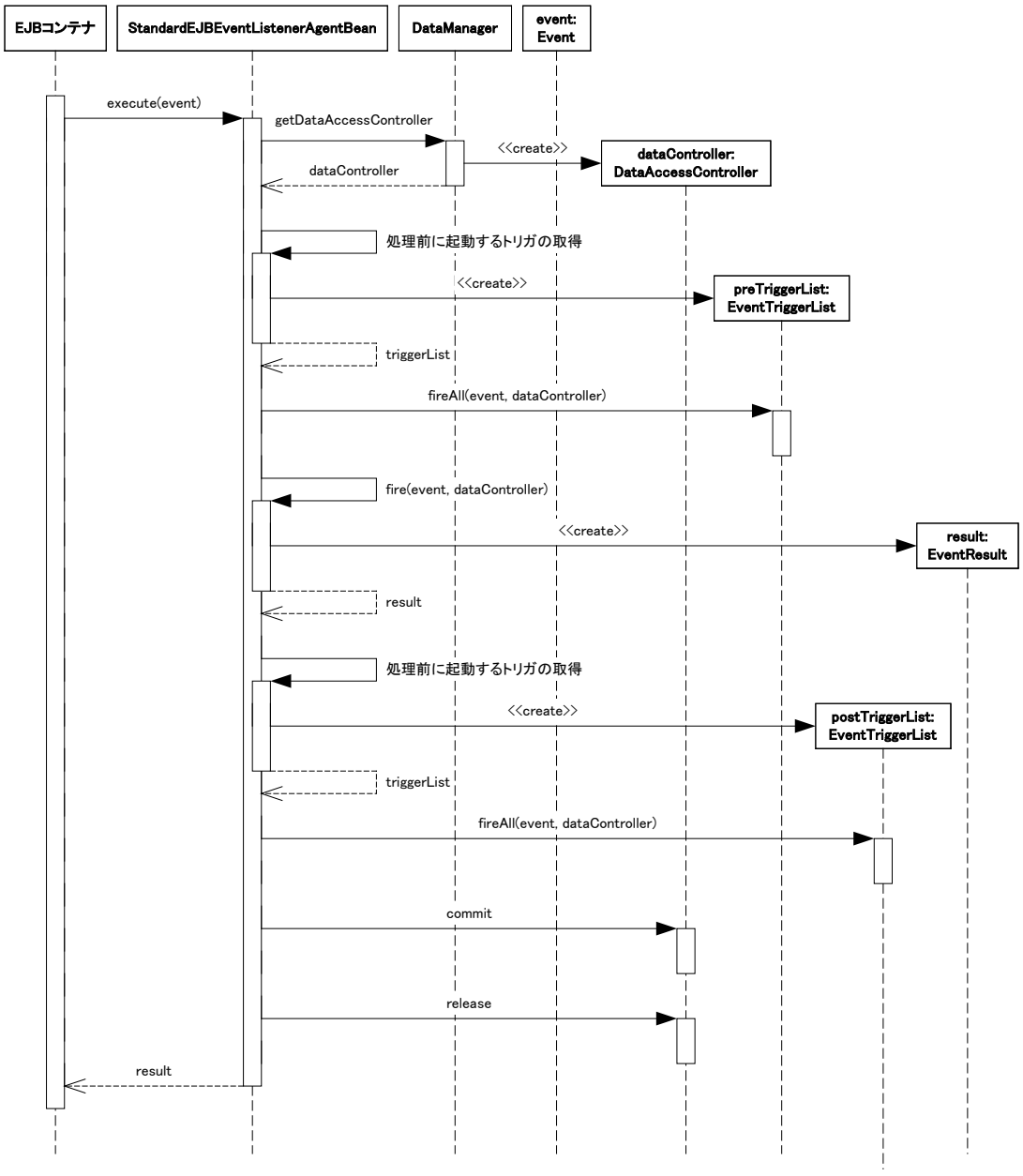


図 4-26 StandardEJBEventListener のトランザクション

「図 4-26 StandardEJBEventListener のトランザクション」を見るとUserTransaction の開始・終了がされていないことがわかる。そのため、StandardEJBEventListener を使う場合、CMT（container-managed transaction：コンテナ管理トランザクション）によってトランザクションの制御をするべきである。また、「4.5.1 StandardEventListener」の説明と同様の理由により、データベースに対するアクセスは DataSource で行うことを推奨する。

StandardEJBEventListener は fire メソッド内部から他の EventListener をネストして呼び出すことが可能である。この場合、StandardEJBEventListener の dispatchEvent メソッドを使用する。このときの様子を「図 4-27 StandardEJBEventListener から他の EventListener の呼び出し」に示す。

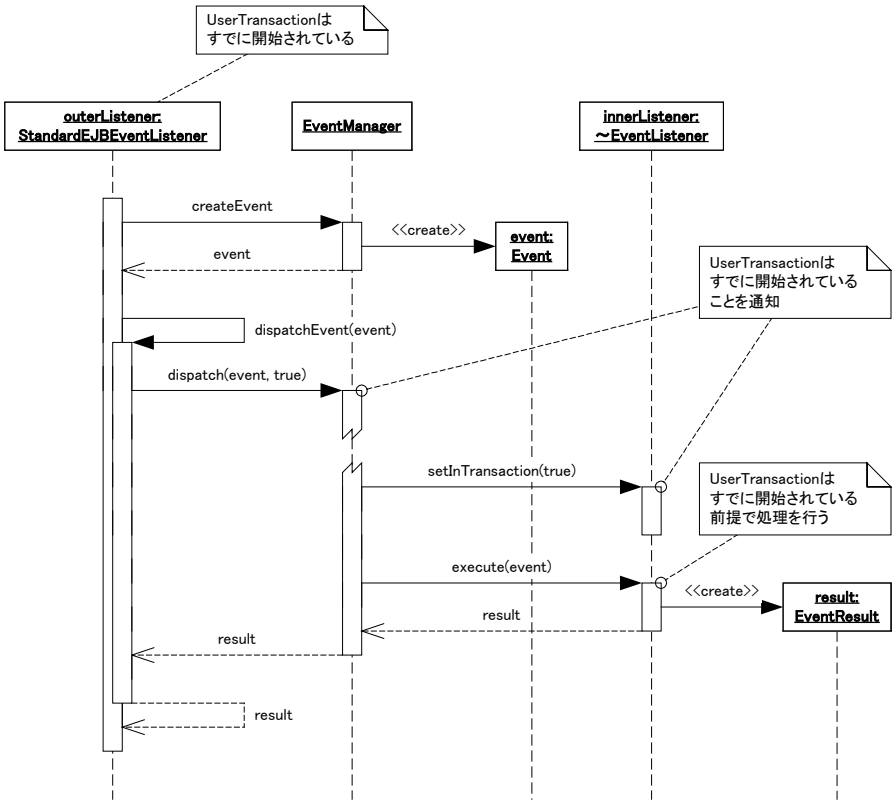


図 4-27 StandardEJBEventListener から他の EventListener の呼び出し

StandardEJBEventListener から StandardEventListener または StandardEJBEventListener を呼び出すとき、データフレームワーク（詳細は「5 データフレームワーク」を参照）に関連する注意点は「4.5.1 StandardEventListener」で説明した注意点と同様である。この問題を避けるためにも、データフレームワークでデータベースなどを扱う場合、DataAccessController ではなく UserTransaction で扱われる DataSource でアクセスすることを推奨する。

4.5.4 GenericEJBEventListener

GenericEJBEventListener でトランザクションを管理している様子を「図 4-28 GenericEJBEventListener のトランザクション」に示す。

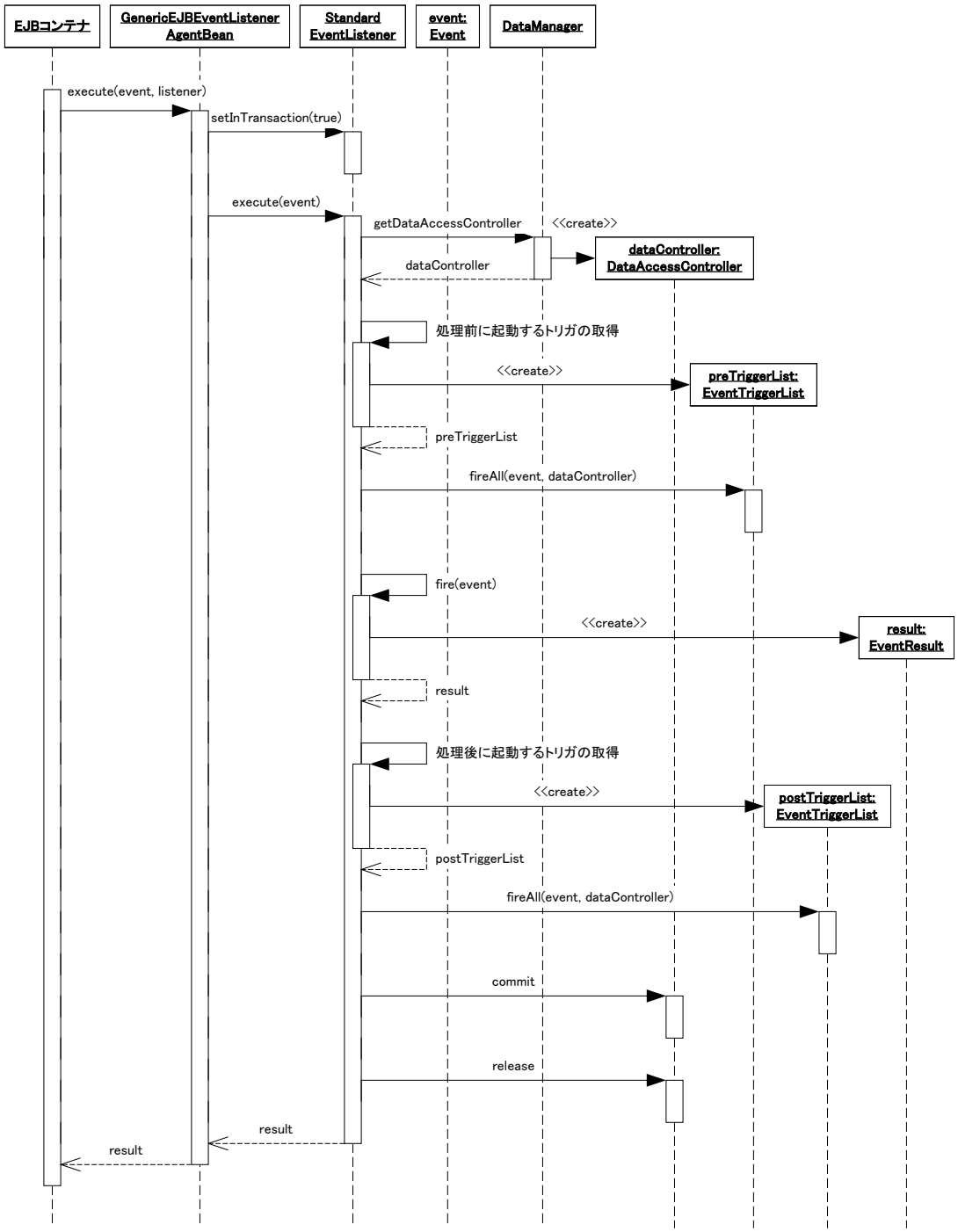


図 4-28 GenericEJBEventListener のトランザクション

「図 4-28 GenericEJBEventListener のトランザクション」を見るとUserTransaction の開始・終了がされていないことがわかる。そのため、GenericEJBEventListener を使う場合、CMT（container-managed transaction：コンテナ管理トランザクション）によってトランザクションの制御をするべきである。また、「4.5.1 StandardEventListener」の説明と同様の理由により、データベースに対するアクセスは DataSource で行うことを推奨する。

GenericEJBEventListener は fire メソッド内部から他の EventListener をネストして呼び出すことが可能である。この場合、GenericEJBEventListener の dispatchEvent メソッドを使用する。このときの様子を「図 4-29 GenericEJBEventListener から他の EventListener の呼び出し」に示す。

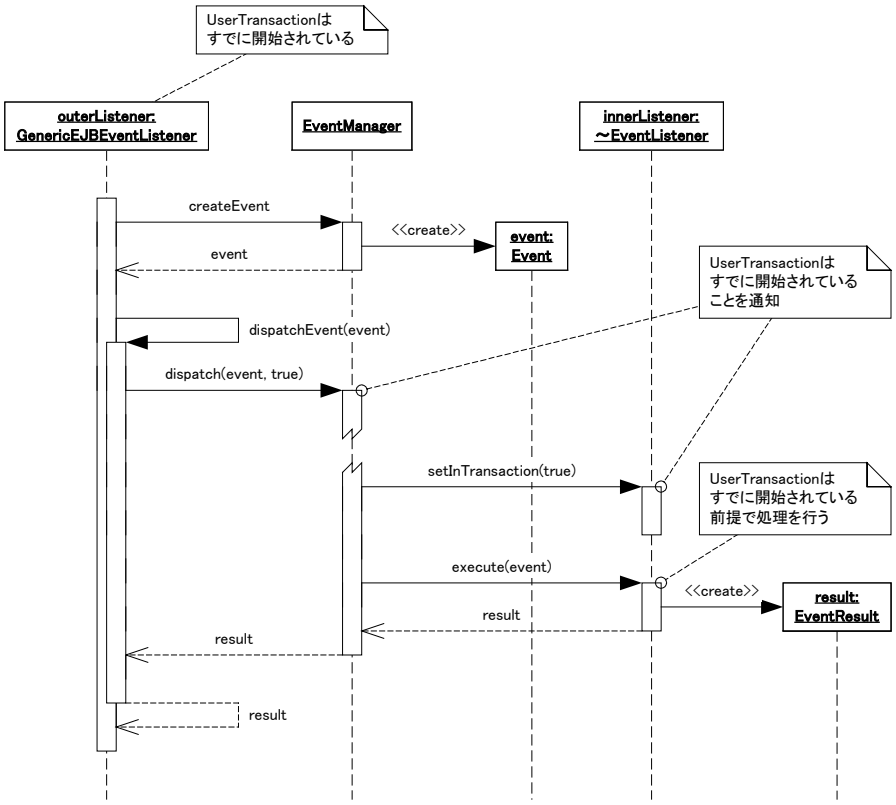


図 4-29 GenericEJBEventListener から他の EventListener の呼び出し

StandardEJBEventListener から StandardEventListener または StandardEJBEventListener を呼び出すとき、データフレームワーク（詳細は「5 データフレームワーク」を参照）に関連する注意点は「4.5.1 StandardEventListener」で説明した注意点と同様である。この問題を避けるためにも、データフレームワークでデータベースなどを扱う場合、DataAccessController ではなく UserTransaction で扱われる DataSource でアクセスすることを推奨する。

5 データフレームワーク

5.1 概要

データの永続化や他システムとの連携などはビジネスロジックの中では最も重要な要素になりうる。データベースや他システムと接続する際には何らかの手続きが必要であるが、これらはほとんどが定型的である。また接続先が変更される場合も考えられるが、この場合に必要な修正は接続先の情報であり、接続の手続きそのものは同じである場合が多い(データベースの種類だけを変更する場合、コネクションの取得やSQLの発行などの修正が必要となるケースはまれである)。

データフレームワークではこれらの定型的な部分をフレームワーク化している。

5.2 構成

5.2.1 構成要素

データフレームワークは以下のようなものから構成されている。

- DAO
- DataConnector
- DataAccessController
- リソース

これらの関連を「図 5-1 データフレームワークの構成」に示す。

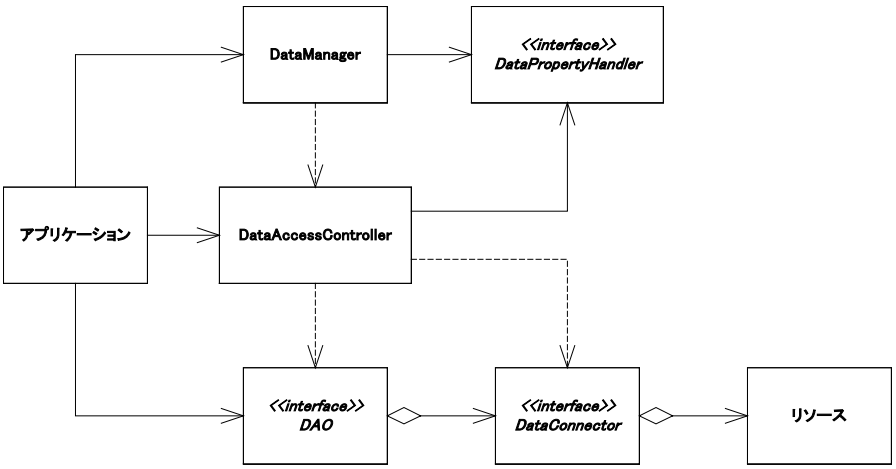


図 5-1 データフレームワークの構成

5.2.1.1 DAO

DAO(データアクセスオブジェクト:Data Access Object)は EIS 層のリソース(データベースや基幹システム等)に対するアクセスを担当する。ビジネスロジックの開発者はデータベースやファイルなどを直接操作せず、DAOを通じてリソースにアクセスすることが望ましい。

DAO は接続先のリソースの種類を知っている必要はあるが、接続先を知る必要はない。DAO は直接リソースにはアクセスせず、DataConnector を経由してアクセスする。

このコンポーネントはリソースに詳しい開発者が実装を担当する。

5.2.1.2 DataConnector

DataConnector は DAO と EIS 層のリソースを接続する役割を担当する。このコンポーネントは標準的なものはいくつか intra-mart から提供されているが、開発者が独自に実装することも可能である。

DataConnector はリソースに特化したものとなる。DataConnector は接続先を明示的に知る必要がある。

5.2.1.3 DataAccessController

DataAccessController は DAO の取得、生成や終了処理、簡易的なトランザクションの管理などを行う。

DAO と DataConnector の関連はプロパティによって定義される。このプロパティ設定内容に従って DataAccessController は適切な DataConnector が設定された DAO を生成し提供する。

このコンポーネントは intra-mart から提供されているため、開発者はその使用方法のみ知っていればよい。

5.2.1.4 リソース

リソースは EIS 層のシステム（データベースや基幹システムなど）の実態またはその接続方法である。リソースの例として intra-mart における Storage サーバ、データベース、基幹システムなどが挙げられる。

リソースそのものは IM-JavaEE Framework では提供されない。

5.2.2 データアクセス

IM-JavaEE Framework のデータフレームワークを利用して EIS 層のリソースにデータアクセスする様子を「図 5-2 データアクセスの概要」に示す。

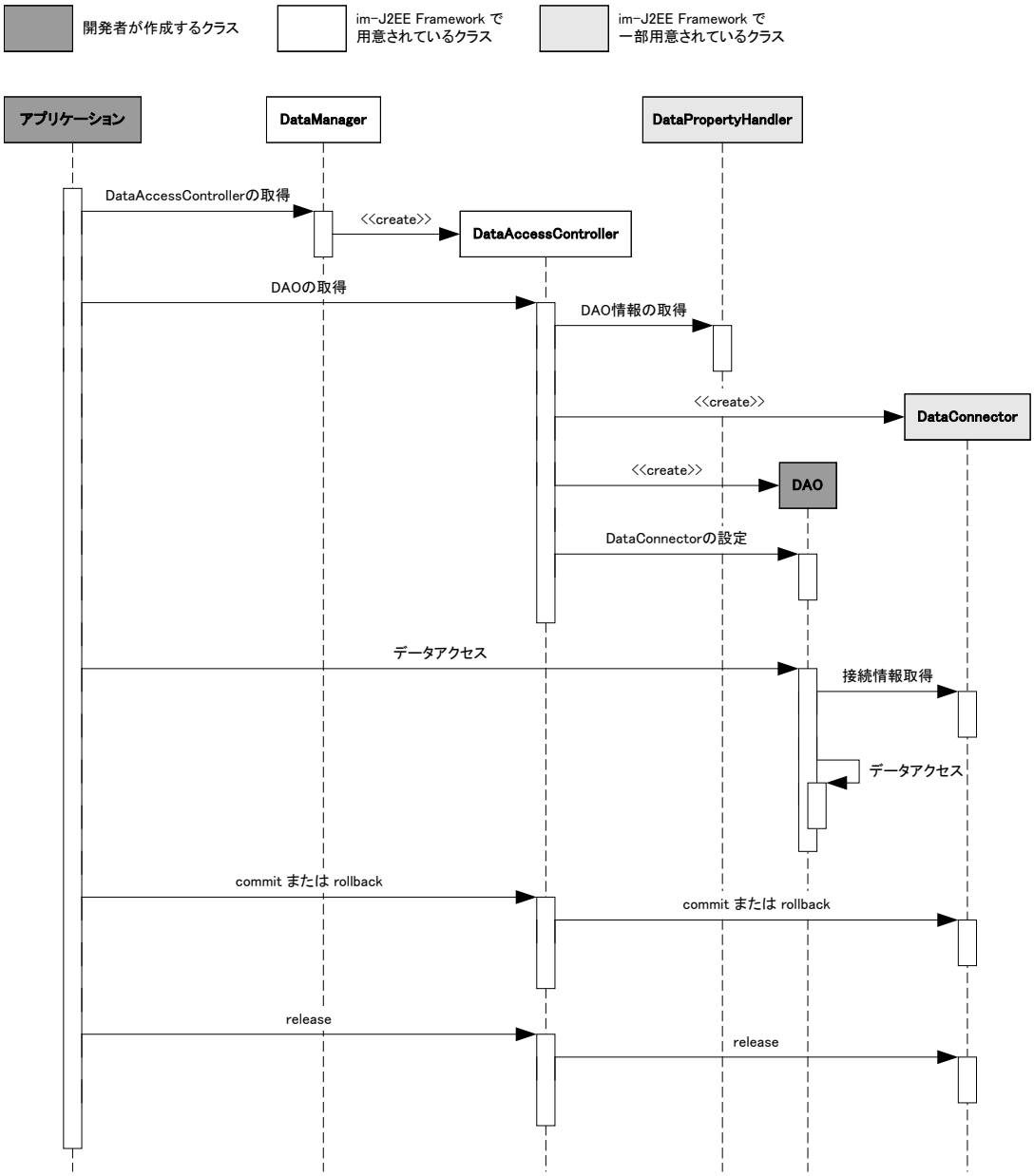


図 5-2 データアクセスの概要

1. DataManager の getDataAccessController メソッドを呼び出し、DataAccessController を取得する。
2. DataAccessController の getDAO メソッドを呼び出し、データにアクセスする DAO を取得する。
3. DAO のメソッドを呼び出しデータにアクセスする。
4. DataAccessController の commit または rollback メソッドを呼び出し、処理内容のコミットまたはロールバックを行う。
5. DataAccessController の release メソッドを呼び出し、リソースへの接続を解放する。

5.3 構成要素の詳細

5.3.1 DAO

DAO（Data Access Object）は EIS 層のリソースに対するアクセスを担当する。DAO は主にビジネスロジックから使用されるためそのインタフェースをビジネスロジックの開発者に対して公開する必要があるが、ビジネスロジックの詳細までは知る必要はない。

DAO ではリソースへのアクセスのみに専念し、接続方法については考える必要はない。リソースへの接続については DataConnector に任せる。

DAO は DataAccessController の getDAO メソッドを使って取得する。DAO を取得する様子を「図 5-3 DAO の取得」に示す。

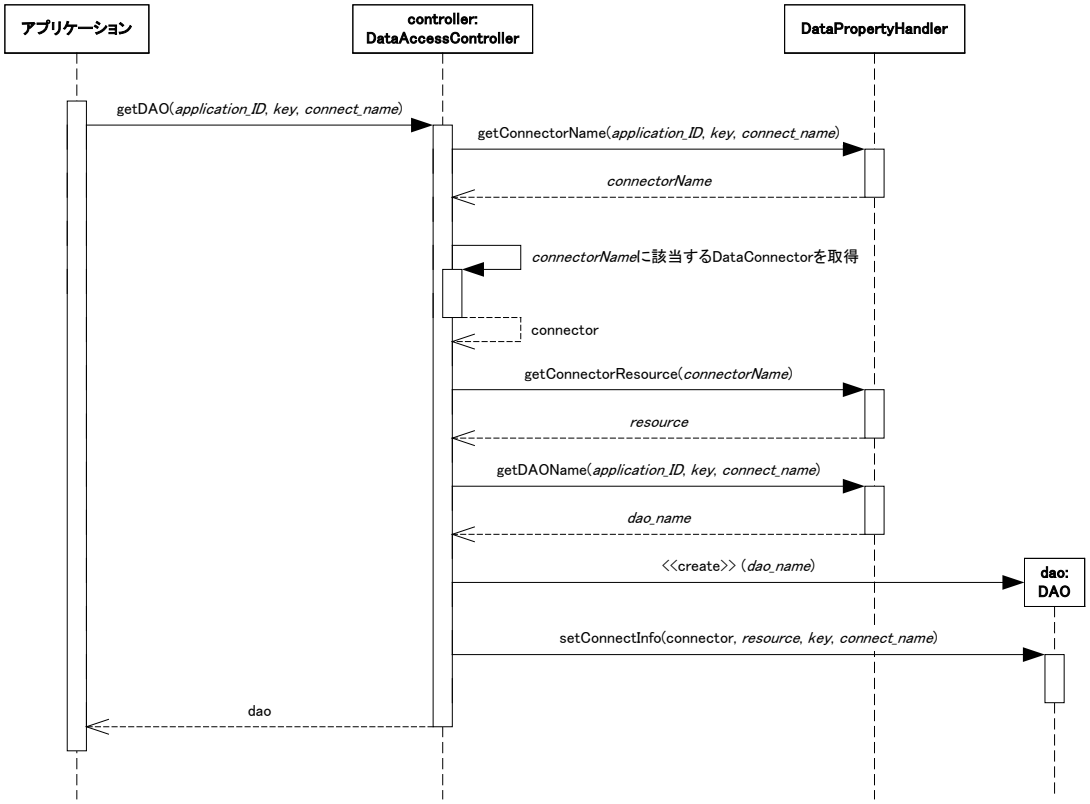


図 5-3 DAO の取得

DAO を取得して直接扱うこともできるが、DAO に対するインタフェース（DAOIF）を用意しそれを実装した DAO を使用するようにアプリケーションを作成することを強く推奨する。こうすることで、リソースの種類が変わった場合でも DAO の設定を変更するだけでよく、DAO を利用するアプリケーションは修正を加える必要はなくなる。詳細は「5.3.1.3 DAOIF」を参照。

5.3.1.1 標準で用意されている DAO

IM-JavaEE Framework では頻繁に利用されると考えられるリソースに特化した DAO がいくつか用意されている。これらの DAO はすべて抽象クラスであるため直接インスタンスを生成することはできないが、開発者はこの DAO のサブクラスを作成することでコーディング量を減らすことが可能となる。

5.3.1.1.1 DBDAO

DBDAO はデータベースに特化した DAO である。DBDAO を利用する場合、DataConnector として jp.co.intra_mart.framework.base.data.DBConnector のサブクラスを指定する必要がある（「図 5-4 DBDAO の構造」参照）。

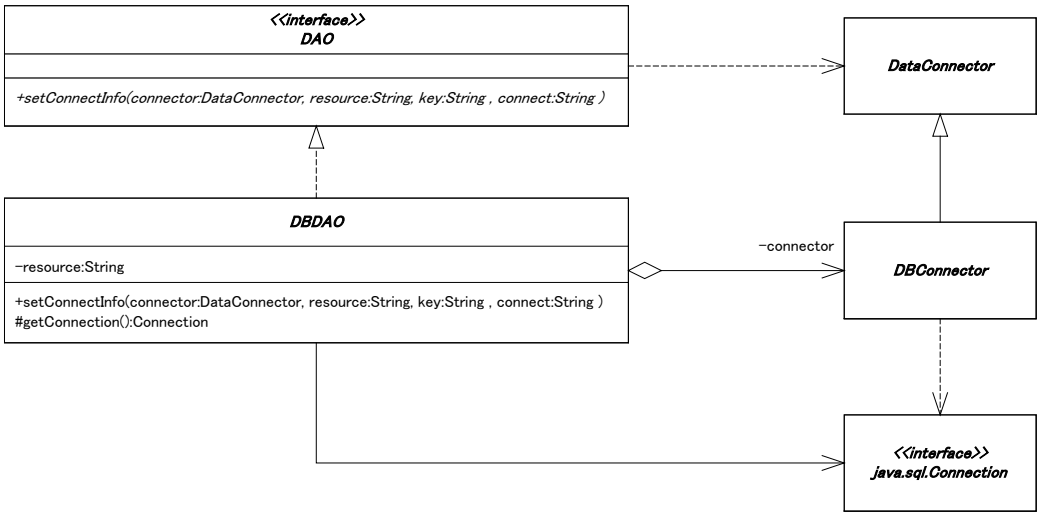


図 5-4 DBDAO の構造

データベースにアクセスするためには java.sql.Connection インタフェースを通じてデータベースにアクセスする必要がある。DBDAO では適切な DataConnector が設定されている場合、getConnection メソッドを呼ぶだけで java.sql.Connection を取得することができる(「図 5-5 Connection の取得(DBDAO)」参照)。

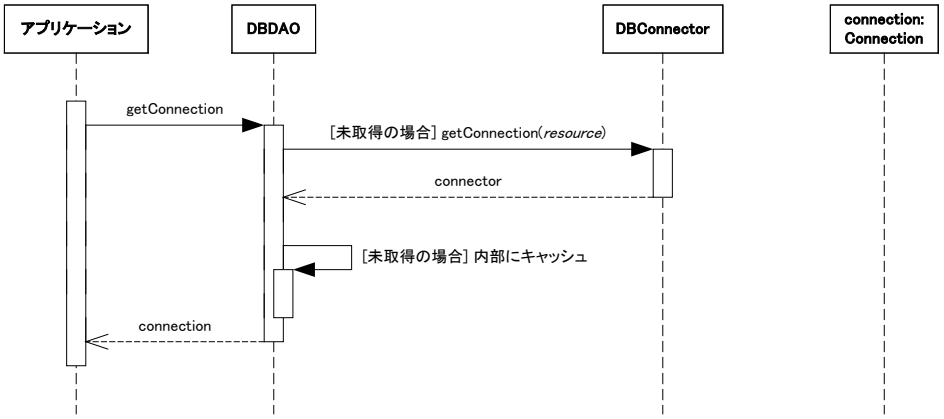


図 5-5 Connection の取得(DBDAO)

5.3.1.1.2 **TenantDBDAO**

TenantDBDAO intra-mart で設定されたテナントデータベース[8]へのアクセスに特化した DAO である。
TenantDBDAO を利用する場合、DataConnector として jp.co.intra_mart.framework.base.data.TenantDBConnector またはそのサブクラスを指定する必要がある(「図 5-6 TenantDBDAO の構造」参照)。

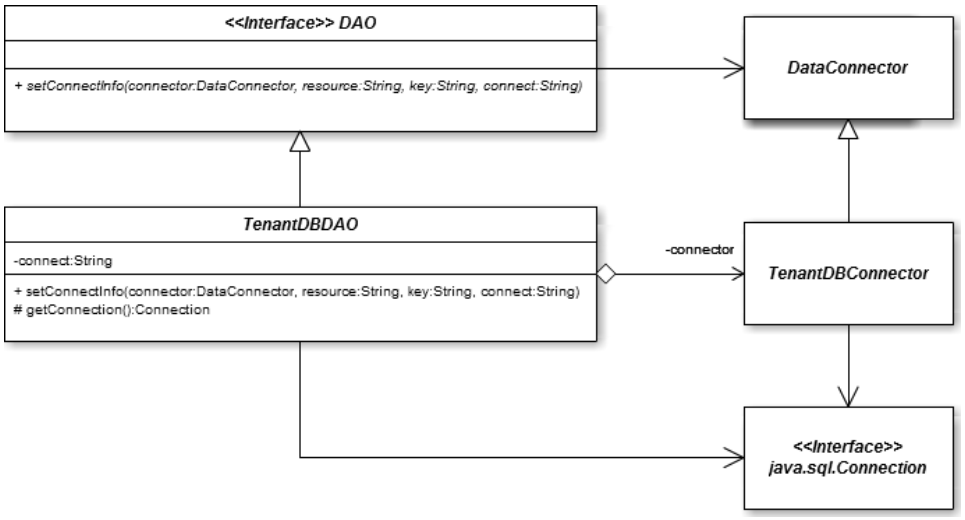


図 5-6 TenantDBDAO の構造

データベースにアクセスするためには java.sql.Connection インタフェースを通じてデータベースにアクセスする必要がある。TenantDBDAO では適切な DataConnector が設定されている場合、getConnection メソッドを呼ぶだけで java.sql.Connection を取得することができる(「図 5-7 Connection の取得(TenantDBDAO)」参照)。

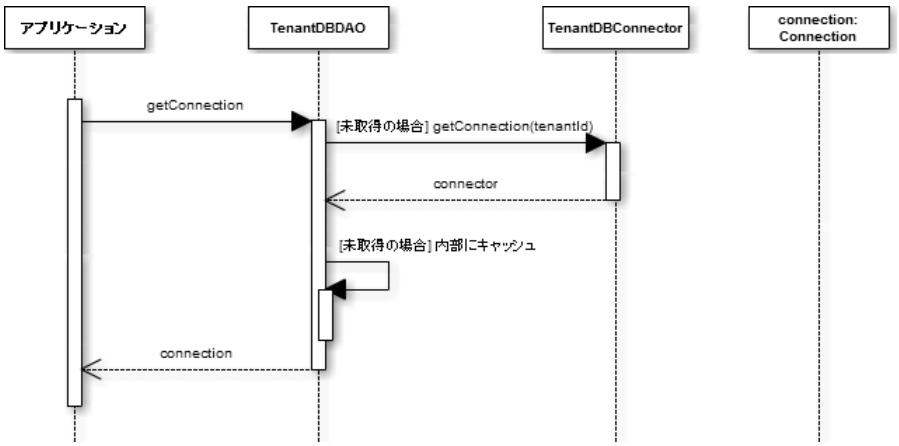


図 5-7 Connection の取得(TenantDBDAO)

5.3.1.1.3 **SharedDBDAO**

SharedDBDAO は intra-mart で設定されたシェアードデータベース[9]へのアクセスに特化した DAO である。
SharedDBDAO を利用する場合、DataConnector として jp.co.intra_mart.framework.base.data. SharedDBConnector またはそのサブクラスを指定する必要がある(「図 5-8 SharedDBDAO の構造」参照)。

8 intra-mart の WEB-INF/conf/data-source-mapping-config.xml で設定されたデータベース。tenant-data-source として定義されたもの。

9 intra-mart の WEB-INF/conf/data-source-mapping-config.xml で設定されたデータベース。shared-data-source として定義されたもの。

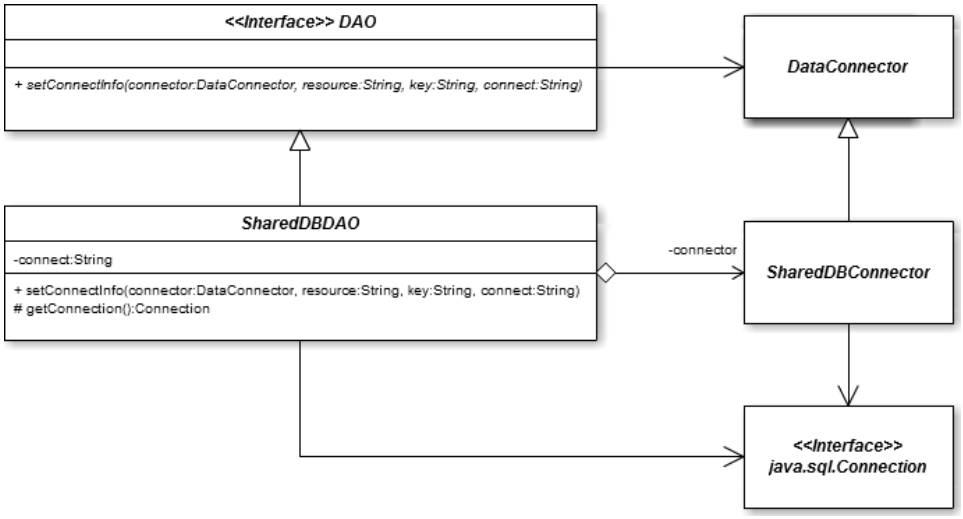


図 5-8 SharedDBDAO の構造

データベースにアクセスするためには java.sql.Connection インタフェースを通じてデータベースにアクセスする必要がある。SharedDBDAO では適切な DataConnector が設定されている場合、getConnection メソッドを呼ぶだけで java.sql.Connection を取得することができる(「図 5-9 Connection の取得(SharedDBDAO)」参照)。

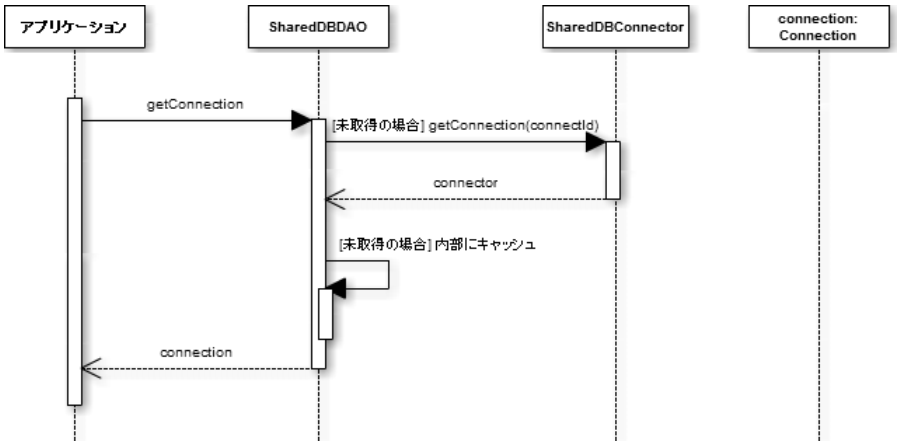


図 5-9 Connection の取得(SharedDBDAO)

5.3.1.1.4 **IntramartDBDAO**

IntramartDBDAO は intra-mart で設定されたデータベース [10] へのアクセスに特化した DAO である。
IntramartDBDAO を利用する場合、
DataConnector として jp.co.intra_mart.framework.base.data.IntramartDBConnector またはそのサブクラスを指定する必要がある(「図 5-10 IntramartDBDAO の構造」参照)。

DbsConnection は非推奨メソッドであり、互換モードとしてインストールしなかれば動作しない。
そのため IntramartDBDAO は互換モジュールをインストールしたときに動作する。
TenantDBDAO または SharedDBDAO を使用することが望ましい。

[10] intra-mart の WEB-INF/conf/data-source-mapping-config.xml で設定されたデータベース。
tenant-data-source と shared-data-source にそれぞれ同じ ID を設定する必要がある。

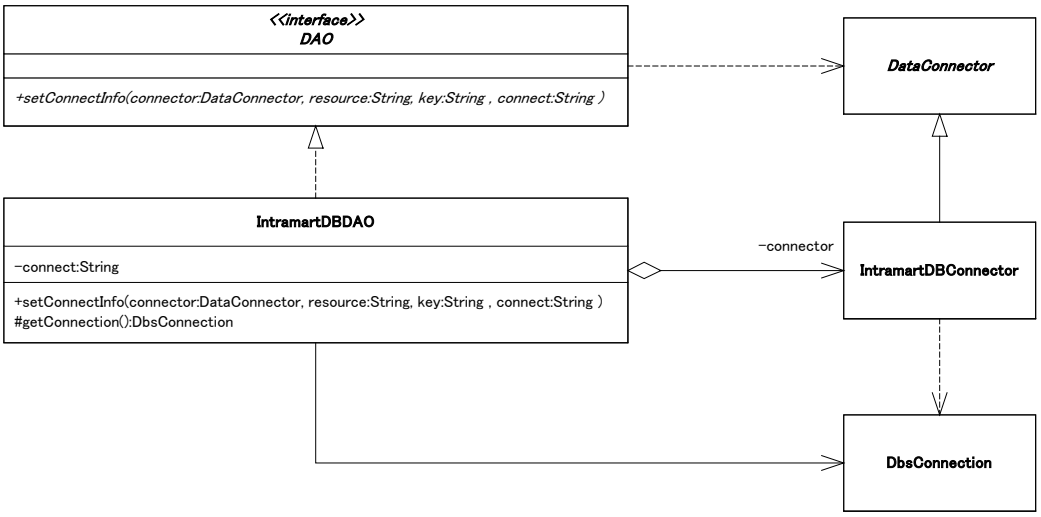


図 5-10 IntramartDBDAO の構造

intra-mart で指定したデータベースにアクセスするためには jp.co.intra_mart.foundation.database.DbsConnection を通じてデータベースにアクセスする必要がある。IntramartDBDAO では、getConnection メソッドを呼ぶだけで DbsConnection を取得することができる(「図 5-11 Connection の取得(IntramartDBDAO)」参照)。

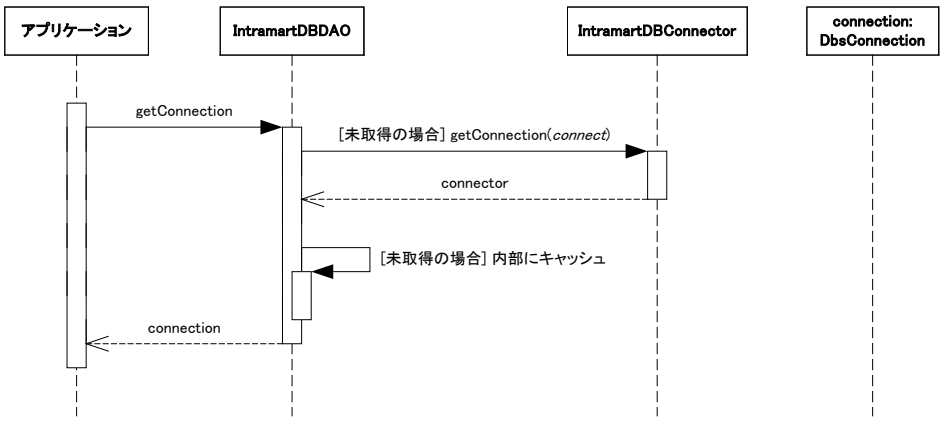


図 5-11 Connection の取得(IntramartDBDAO)

5.3.1.1.5 IntramartStorageDAO

IntramartStorageDAO は intra-mart の Public Storage で運用しているファイルへのアクセスに特化した DAO である。IntramartStorageDAO を利用する場合、
DataConnector として jp.co.intra_mart.framework.base.data.IntramartStorageConnector またはそのサブクラスを指定する必要がある(「図 5-12 IntramartStorageDAO の構造」参照)。

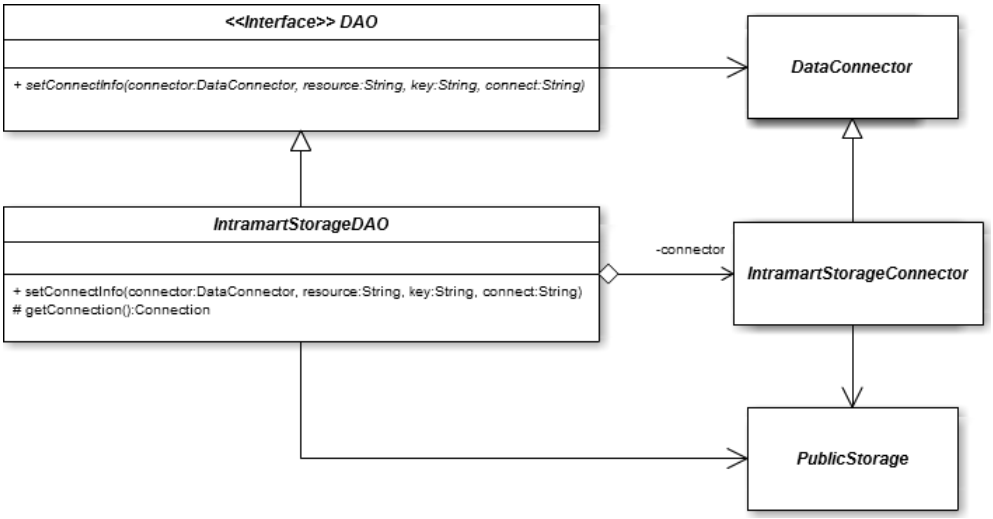


図 5-12 IntramartStorageDAO の構造

Storage Service の利用には jp.co.intra_mart.foundation.service.client.file.PublicStorage を通じてファイルにアクセスする必要がある。IntramartStorageDAO では、getStorageFile メソッドを呼ぶだけで PublicStorage を取得することができる(「図 5-13 Connection の取得(IntramartStorageDAO)」参照)。

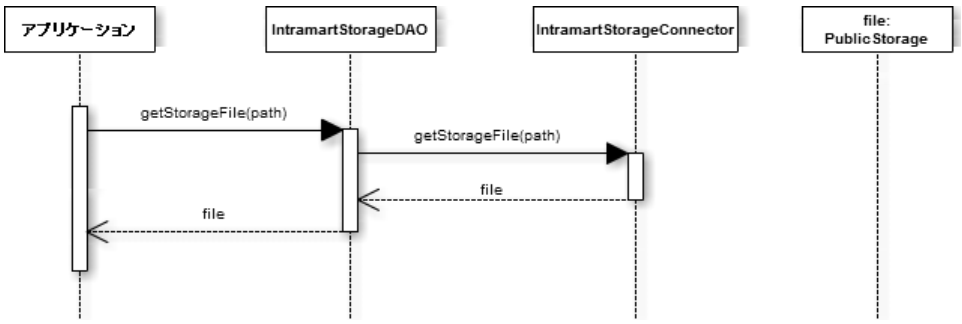


図 5-13 Connection の取得(IntramartStorageDAO)

5.3.1.2 独自の DAO

DAO を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.data.DAO インタフェースを実装している。
- public なデフォルトコンストラクタ(引数なしのコンストラクタ)が定義されている。
- setConnectInfo メソッドにおいて適切な接続情報の設定が行われること。

また、必須ではないが以下のような実装がされていることが望ましい。

- 抽象クラス(abstract class)である。
- 特定の DataConnector を経由して特定のリソースにアクセスする簡易的な方法が実装されている。たとえば、DBDAO は DBConnector のサブクラスを経由してリレーショナルデータベースにアクセスする getConnection メソッドを持っている。

5.3.1.3 DAOIF

アプリケーションは DAO を通じてリソースにアクセスする。DAO は DataAccessController の getDAO メソッドで取得することができる。このメソッドの戻り値の型は java.lang.Object であるため、DAO を使用するアプリケーションは適切なクラスにダウンキャストする必要がある。この場合、使用する DAO のクラスに直接ダウンキャストするのではなく、DAOIF(該当する DAO のメソッドを定義したインタフェース)にダウンキャストすることを推奨する。

DAOIF を用いた場合、リソースの種類が変更された場合でも DAO を取り替えるだけでよく、DAO を利用するアプリケーションは修正をする必要はない。また、DAOIF のメリットを活用するためには、アプリケーションは DAO ではなく DAOIF に対してコーディングをするべきである。

例として「図 5-14 DAOIF の利用」に示すような DAO があるものとする。

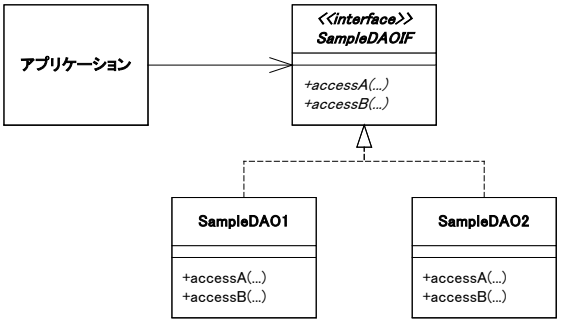


図 5-14 DAOIF の利用

ここで DAOIF を使用せずにリソースにアクセスするアプリケーションのコードは「リスト 5-1 DAOIF を用いない場合」のようになる。このコードでも問題なく動作するが、使用するリソースの種類が変更された場合 SampleDAO1 の実装を変更しない限りアプリケーションを修正する必要がある。

リスト 5-1 DAOIF を用いない場合

```
・・・
// DataAccessController である controller を取得済みであるものとする
SampleDAO1 dao = (SampleDAO1)controller.getDAO("app1", "key1", "con1");
dao.access(...);
・・・
```

これに対して DAOIF を使用してリソースにアクセスするアプリケーションのコードを「リスト 5-2 DAOIF を用いる場合（推奨）」に示す。このコードでは SampleDAOIF に対してコーディングされていることに注意する。このコードでは、使用するリソースの種類が変更された場合プロパティの設定において SampleDAO1 を SampleDAO2 に取り替える必要はあるが、アプリケーションの修正は不要である。

リスト 5-2 DAOIF を用いる場合（推奨）

```
・・・
// DataAccessController である controller を取得済みであるものとする
SampleDAOIF dao = (SampleDAOIF)controller.getDAO("app1", "key1", "con1");
dao.access(...);
・・・
```

5.3.2 DataAccessController

DataAccessController には次の 2 つの役割がある。

- DAO の取得
- トランザクションの管理

5.3.2.1 DAO の取得

DAO は DataAccessController を通じて取得することができる。この様子を「図 5-15 DAO の取得」に示す。

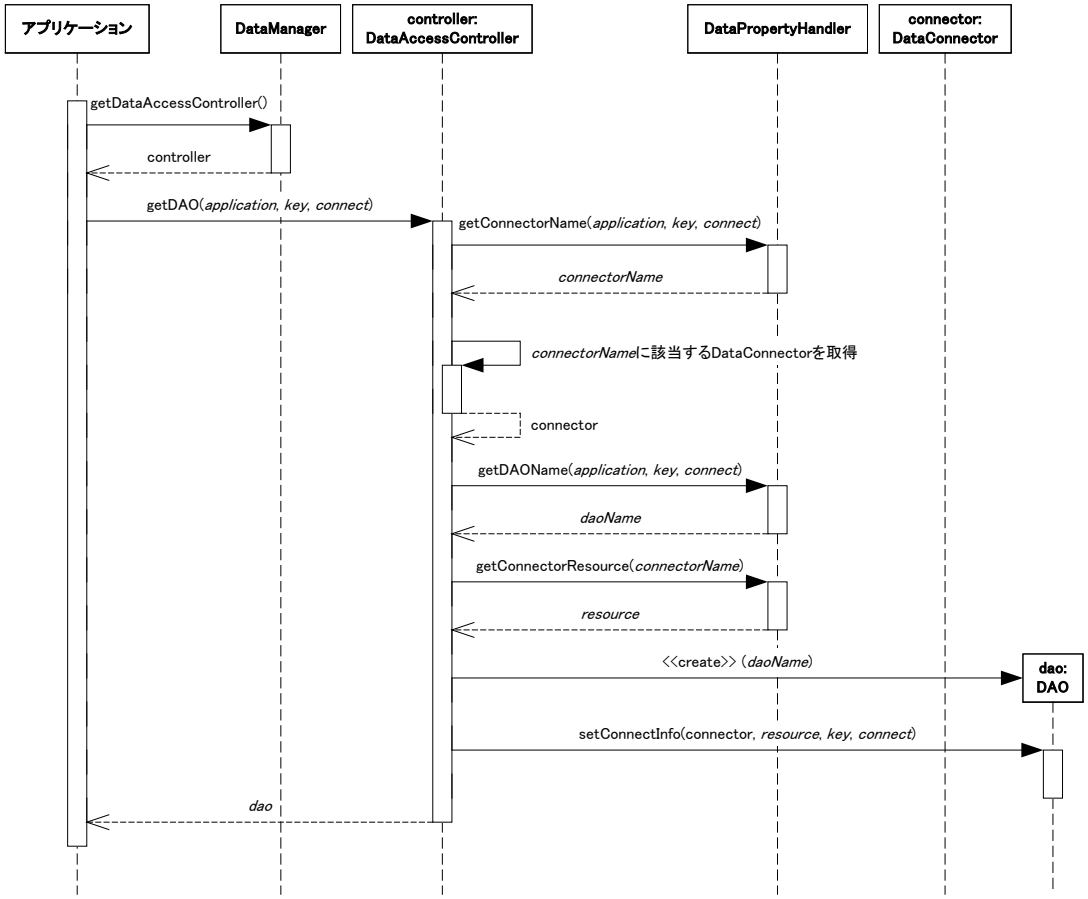


図 5-15 DAO の取得

「図 5-15 DAO の取得」では次のような処理をしている。

1. アプリケーションは DataManager から DataAccessController を取得する（getDataAccessController）。
2. アプリケーションは取得した DataAccessController に対してアプリケーション ID、キー、接続名を渡し、DAO を取得するよう依頼する（getDAO）。
3. DataAccessController はアプリケーション ID、キー、接続名に対応するデータコネクタ名を取得する（getConnectorName）。
4. DataAccessController は取得したデータコネクタ名に該当する DataConnector を取得する。
5. DataAccessController はアプリケーション ID、キー、接続名に対応する DAO のクラス名を取得する（getDAOName）。
6. DataAccessController はアプリケーション ID、キー、接続名に対応するリソース名を取得する（getConnectorResource）。
7. DataAccessController は取得した DAO のクラス名をもとに DAO を生成する。
8. DataAccessController は生成した DAO に DataConnector を設定する（setConnectorInfo）。
9. DataAccessController はアプリケーションに DAO を返す。

DataAccessController はデータコネクタ名をキーとして DataConnector をキャッシュしている。DataAccessController は DataConnector を取得するとき、最初に自身内のキャッシュを検索する。キャッシュ内にない場合、DataPropertyHanler からデータコネクタ名に該当する DataConnector のクラス名を取得し（getConnectorClassName）、新規に DataConnector を生成する。この DataConnector はキャッシュに追加され、同じ DataAccessController 内で再利用される。

DataConnector のキャッシュの様子を「図 5-16 DataConnector の取得（キャッシュされている場合）」および「図 5-17 DataConnector の取得（キャッシュされていない場合）」に示す。

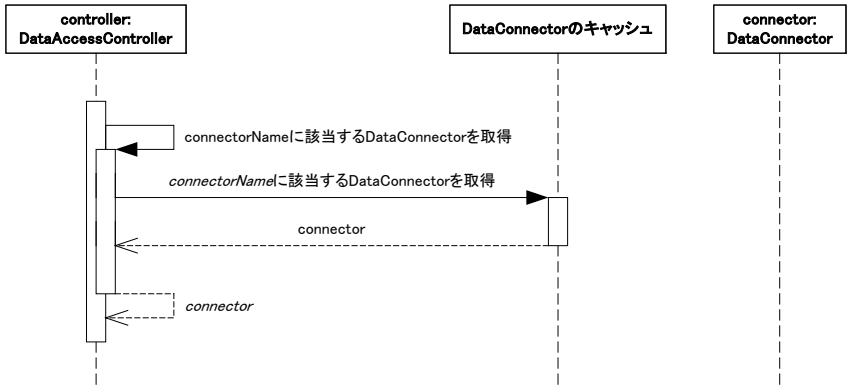


図 5-16 DataConnector の取得（キャッシュされている場合）

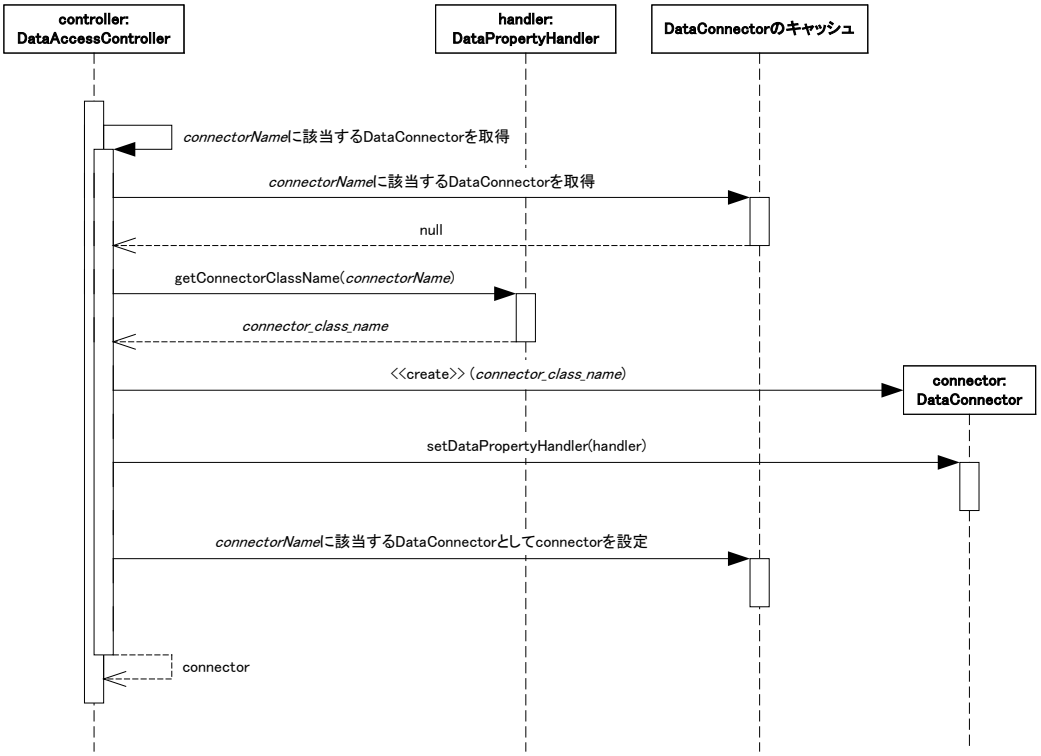


図 5-17 DataConnector の取得（キャッシュされていない場合）

5.3.2.2 トランザクションの管理

DataAccessController を使って簡易的なトランザクションを実現することができる。この方法によるトランザクション管理は推奨されない。詳細については「5.5.1.2 DataAccessController によるトランザクション管理」を参照。

5.3.3 DataConnector

DataConnector は DAO とリソースを接続する役割がある。DAO によっては使用できる DataConnector が限定されるため、どの DataConnector を使用するのかは DAO ごとに考慮する必要がある（「5.3.1 DAO」を参照）。

5.3.3.1 標準で用意されている DataConnector

IM-JavaEE Framework ではよく使えられると考えられる以下のような DataConnector があらかじめ用意されている。

- DBConnector
  リレーショナルデータベースに接続するための DataConnector。これ自体は抽象クラスである。

- JDBCConnector
  JDBC で接続するための DataConnector。DBConnector のサブクラスとして定義されている。
- DataSourceConnector
  データソース経由で接続するための DataConnector。DBConnector のサブクラスとして定義されている。
- IntramartDBConnector
  intra-mart で管理しているデータベースに接続するための DataConnector。
- IntramartStorageConnector
  intra-mart の Public Storage に接続するための DataConnector。

以下にその詳細を記す。

5.3.3.1.1 DBConnector

DBConnector は RDBMS に接続するための java.sql.Connection を取得することを主な役割としている。DBConnector 自体は抽象クラスであり、Connection の取得方法そのものはサブクラスに任されている。このサブクラスのコネクタは DBDAO で使用されることを主な目的としている。

DBConnector の構造を「図 5-18 DBConnector の構造」に示す。

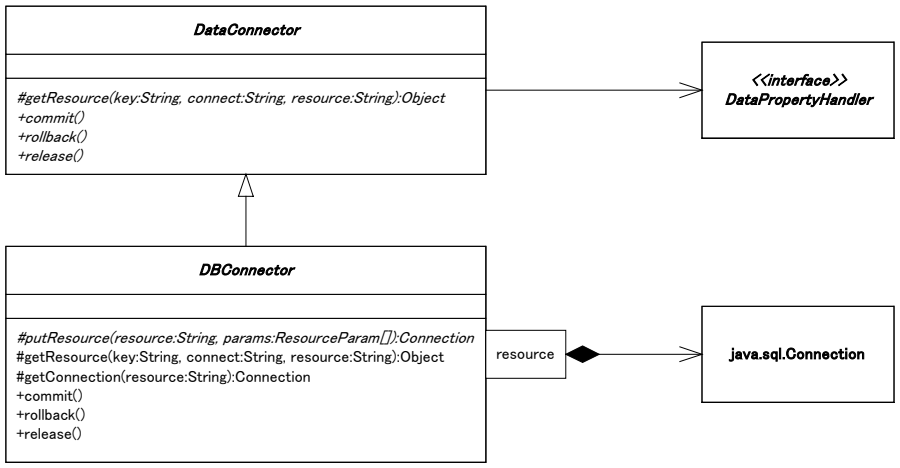


図 5-18 DBConnector の構造

DBConnector は接続要求（getConnection）があった場合、自身で保持している Connection のキャッシュからリソース名に該当する Connection を検索する。Connection が未取得の場合、リソース名に該当するリソース情報を DataPropertyHandler から取得（getResourceParams）して対応する Connection を新規に取得し、取得した Connection をキャッシュに追加する。キャッシュされた Connection は同一の DBConnector 内で再利用される。

Connection が再利用されている様子を「図 5-19 DBConnector から Connection を取得（取得済）」および「図 5-20 DBConnector から Connection を取得（新規）」に示す。

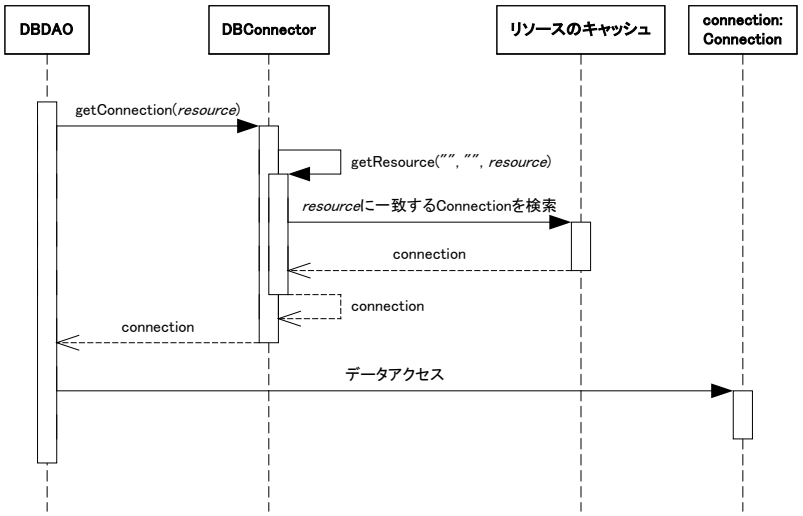


図 5-19 DBConnector から Connection を取得（取得済）

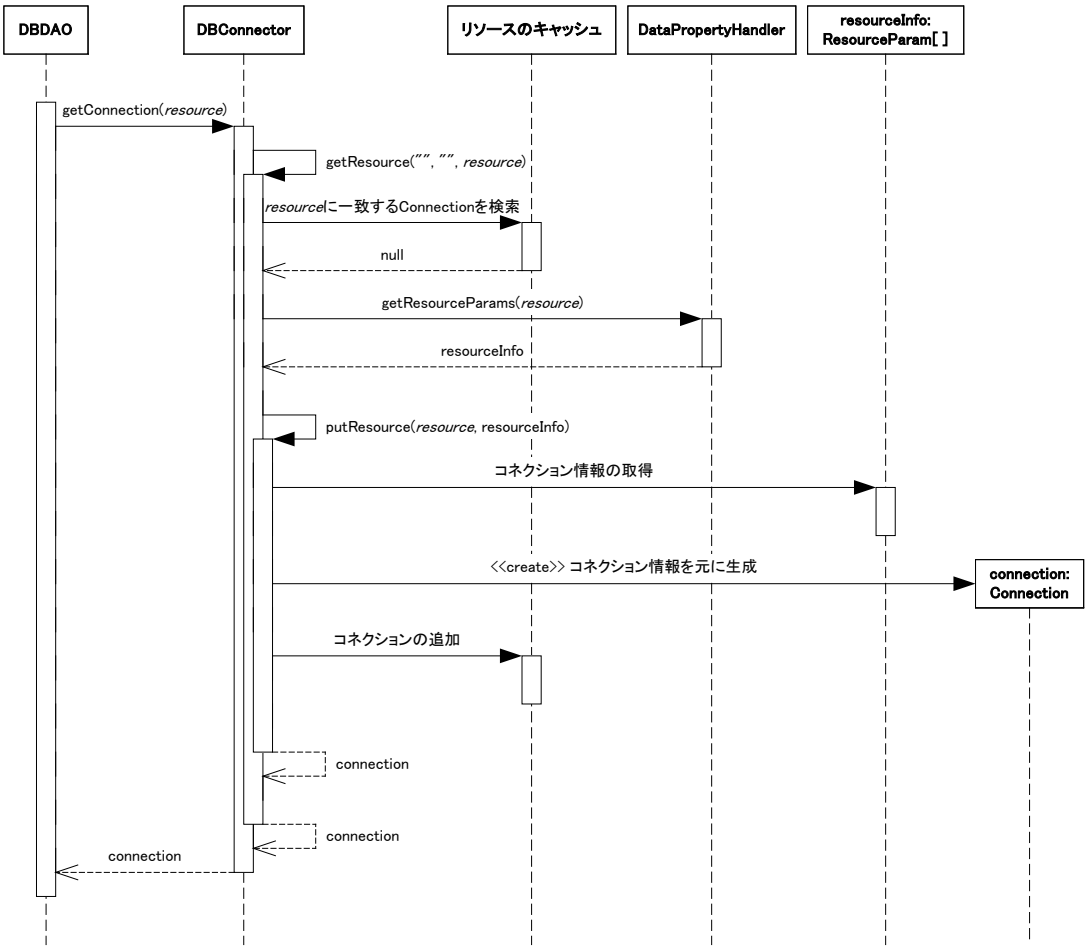


図 5-20 DBConnector から Connection を取得（新規）

5.3.3.1.2 **JDBCConnector**

JDBCConnector の使用は推奨されない。代わりに DataSourceConnector の使用を推奨する。

JDBCConnector は RDBMS に接続する簡易的な機能を提供する。JDBCConnector は主に DBDAO で利用されることを目的としている。

JDBCConnectorを利用する場合、データプロパティに「表 5-1 JDBCConnectorの設定内容」に示す設定が必要となる。

表 5-1 JDBCConnector の設定内容

項目	内容	
DataConnector	jp.co.intra_mart.framework.base.data.JDBCConnector	
データコネクタ名	任意	
リソース名	任意	
リソースパラメータ	driver	データベースに接続するドライバのクラス名
	url	データベースに接続する URL
	username	データベースに接続するユーザ名
	password	データベースに接続するユーザのパスワード

DataPropertyHandlerとしてXmlDataPropertyHandlerを指定している場合、「表 5-1 JDBCConnectorの設定内容」に対応する設定内容の例を「リスト 5-3 data-config.xml の設定例（JDBCConnector）」と「リスト 5-4 data-config-application.xml の設定例（JDBCConnector）」に示す。この例ではアプリケーション ID を application、データコネクタ名を myCon、リソース名を myResource、ドライバのクラス名を sample.jdbc.driber.SampleDriver、データベース接続時の URL を jdbc:sampledb:sample、データベースの接続ユーザを imart、パスワードを imartpass としている。

リスト 5-3 data-config.xml の設定例（JDBCConnector）

```
...
  <connector>
      <connector-name>myCon</connector-name>
      <connector-class>jp.co.intra_mart.framework.base.data.JDBCConnector</connector-class>
      <resource-name>myResource</resource-name>
  </connector>

  <resource>
      <resource-name>myResource</resource-name>
      <init-param>
          <param-name>driver</param-name>
          <param-value>sample.jdbc.driber.SampleDriver</param-value>
      </init-param>
      <init-param>
          <param-name>url</param-name>
          <param-value>jdbc:sampledb:sample</param-value>
      </init-param>
      <init-param>
          <param-name>username</param-name>
          <param-value>imart</param-value>
      </init-param>
      <init-param>
          <param-name>password</param-name>
          <param-value>imartpass</param-value>
      </init-param>
  </resource>
...
```

リスト 5-4 data-config-application.xml の設定例（JDBCConnector）

```
・・・
  <dao-group>
      <dao-key>key</dao-key>
      <dao>
        <dao-class>・・・</dao-class>
        <connector-name>myCon</connector-name>
      </dao>
 </dao-group>
・・・
```

JDBCConnector は以下の理由により推奨されない。

- UserTransaction と同じトランザクションで管理することができない。
- 一般的なアプリケーションサーバでは通常、データソース以外の Connection のプーリング機能がない。

JDBCConnector の構造を「図 5-21 JDBCConnector の構造」に示す。

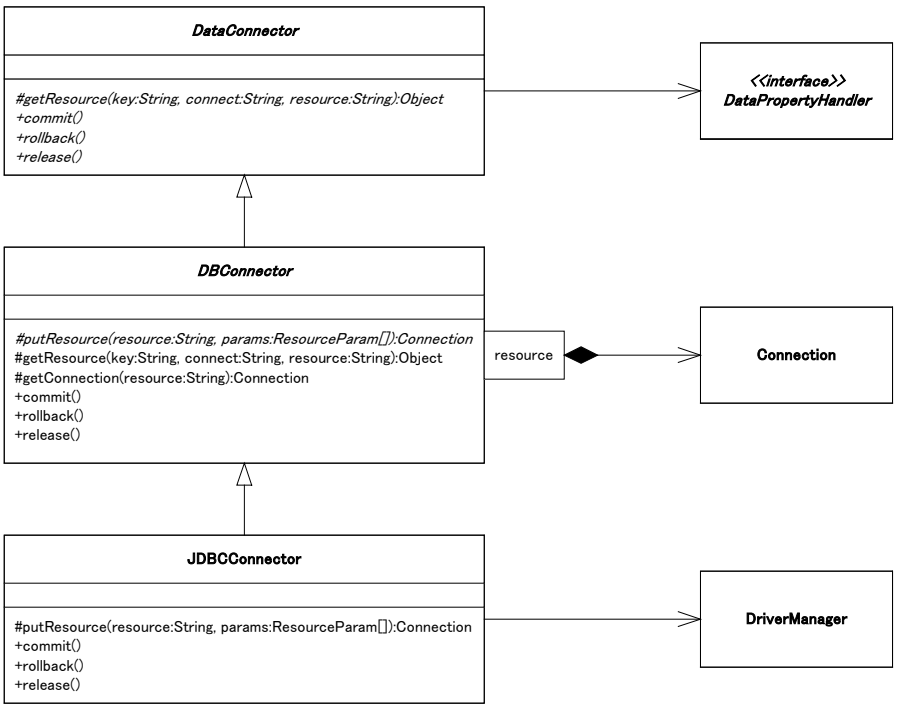


図 5-21 JDBCConnector の構造

JDBCConnector を使って Connection を取得する様子を「図 5-22 Connection の取得（JDBCConnector）」に示す。
DAO を使ってコーディングを行う開発者は、通常はこの動作について深く理解する必要はない。

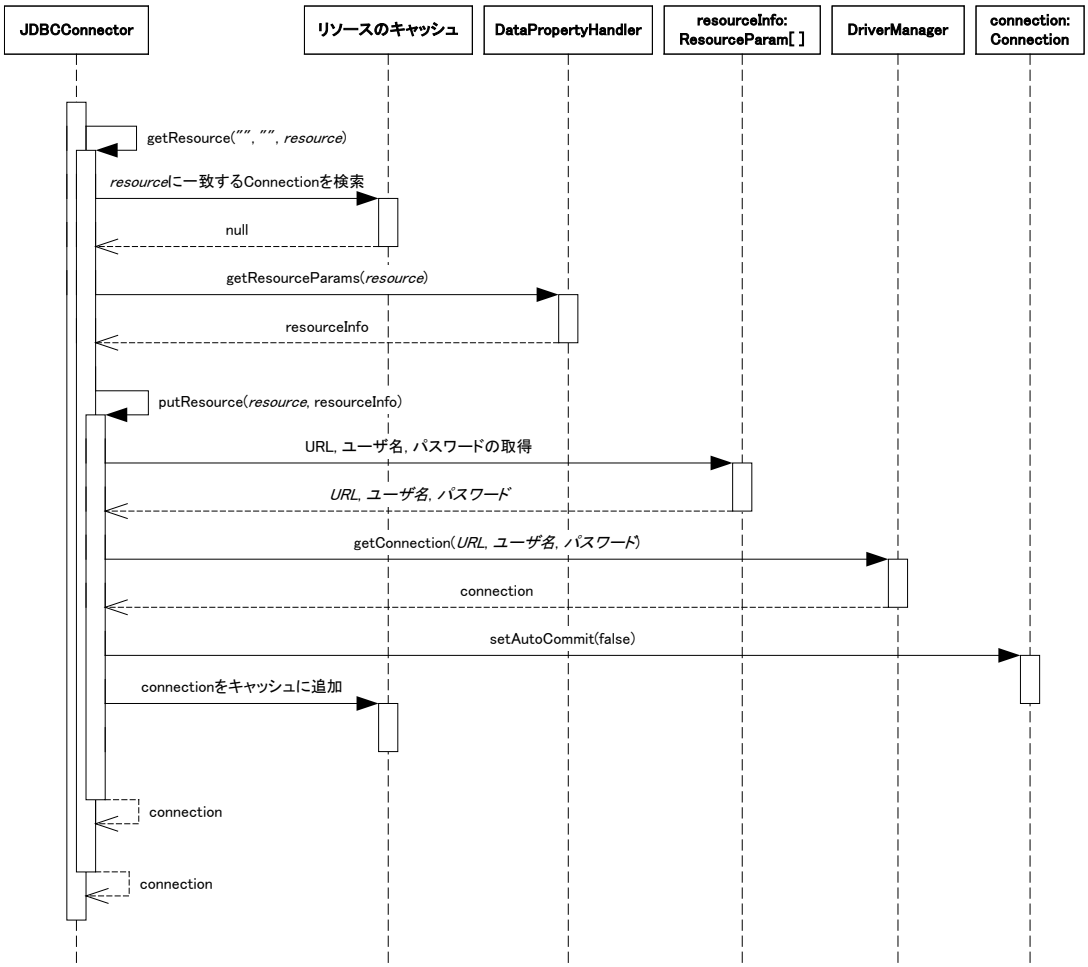


図 5-22 Connection の取得(JDBCConnector)

JDBCConnector に対してコミットまたはロールバックが行われるときの様子を「図 5-23 コミットまたはロールバック(JDBCConnector)」に示す。

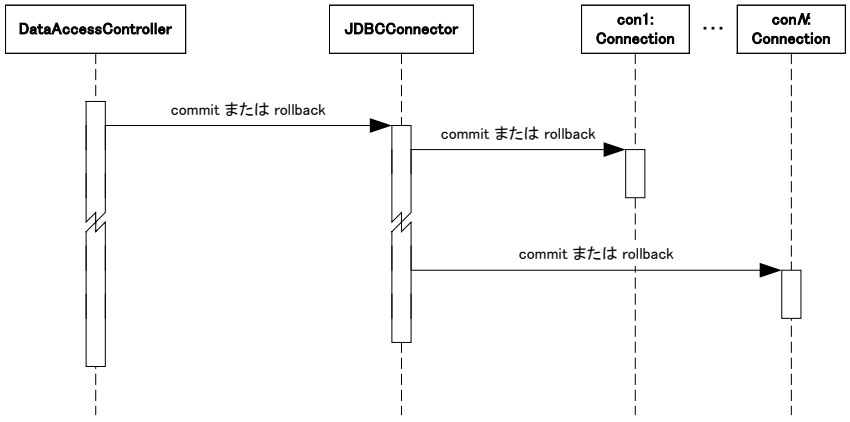


図 5-23 コミットまたはロールバック(JDBCConnector)

「図 5-23 コミットまたはロールバック(JDBCConnector)」を見るとわかるように、JDBCConnector は所有しているすべての Connection それぞれに対してコミットまたはロールバックをしていることがわかる。コミットをしているとき、どれかひとつでもコミットに失敗した場合はそれ以降の Connection に対してはロールバックされるが、既にコミットさ

れた Connection に対しては何も行わない[11]。

5.3.3.1.3 DataSourceConnector

DataSourceConnector は RDBMS に接続する簡易的な機能を提供する。DataSourceConnector は主に DBDAO で利用されることを目的としている。JDBCConnector との違いを以下に示す。

- データベースに対する Connection をアプリケーションサーバに登録された DataSource から取得する。
- UserTransaction と同じトランザクションで管理される。

DataSourceConnector を利用する場合、データプロパティに「表 5-2 DataSourceConnector の設定内容」に示す設定が必要となる。

表 5-2 DataSourceConnector の設定内容

項目	内容	
DataConnector	jp.co.intra_mart.framework.base.data.DataSourceConnector	
データコネクタ名	任意	
リソース名	任意	
リソースパラメータ	jndi	DataSource のルックアップ名

DataPropertyHandler として XmlDataPropertyHandler を指定した場合、「表 5-2 DataSourceConnector の設定内容」に対応する設定内容の例を「リスト 5-5 data-config.xml の設定例（DataDourceConnector）」と「リスト 5-6 data-config-application.xml の設定例（DataDourceConnector）」に示す。この例ではアプリケーション ID を application、データコネクタ名を myCon、リソース名を myResource とし、データソースが "java:comp:env/jdbc/mydb"でルックアップできるよう、アプリケーションサーバに設定されているものとしている。

リスト 5-5 data-config.xml の設定例（DataDourceConnector）

```
・・・
<connector>
        <connector-name>myCon</connector-name>
  <connector-class>jp.co.intra_mart.framework.base.data.DataSourceConnector</connector-class>
  <resource-name>myResource</resource-name>
</connector>
・・・
<resource>
  <resource-name>myResource</resource-name>
  <init-param>
        <param-name>jndi</param-name>
        <param-value>java:comp:env/jdbc/mydb</param-value>
  </init-param>
</resource>
・・・
```

[11] これはトランザクションの原子性（Atomicity：トランザクション内の処理はすべて成功するかすべて失敗するかのいずれか）に反することを意味する。そのため、JDBCConnector の使用は推奨されない。

リスト 5-6 data-config-application.xml の設定例（DataDourceConnector）

```
・・・
<dao-group>
  <dao-key>key</dao-key>
  <dao>
  <dao-class>・・・</dao-class>
    <connector-name>myCon</connector-name>
  </dao>
</dao-group>
・・・
```

DataSourceConnector の構造を「図 5-24 DataSourceConnector の構造」に示す。

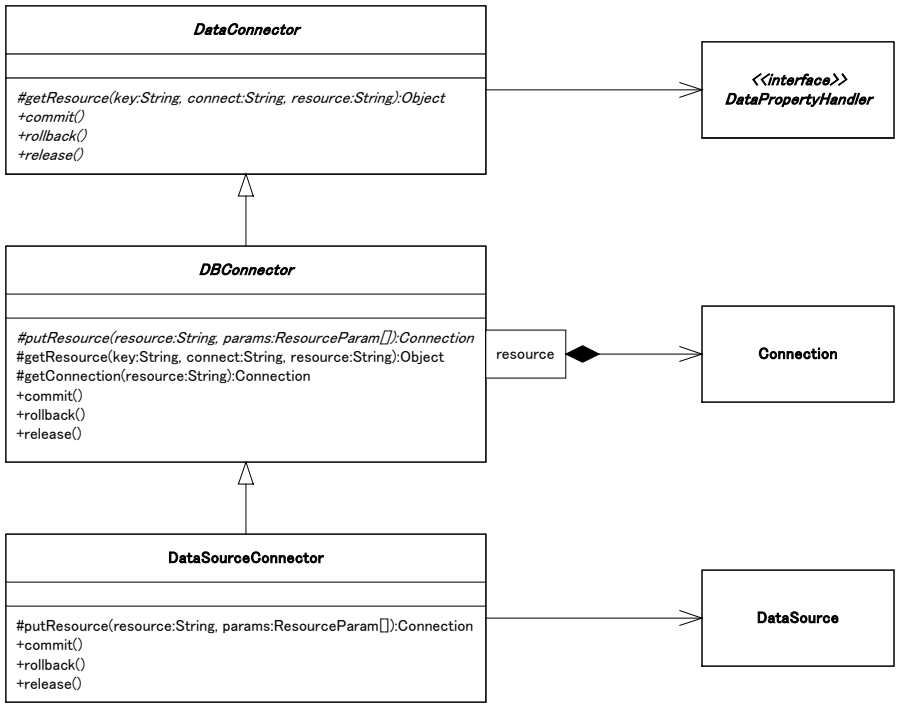


図 5-24 DataSourceConnector の構造

DataSourceConnector を使って Connection を取得する様子を「図 5-25 Connection の取得（DataSourceConnector）」に示す。DAO を使ってコーディングを行う開発者は、通常はこの動作について深く理解する必要はない。

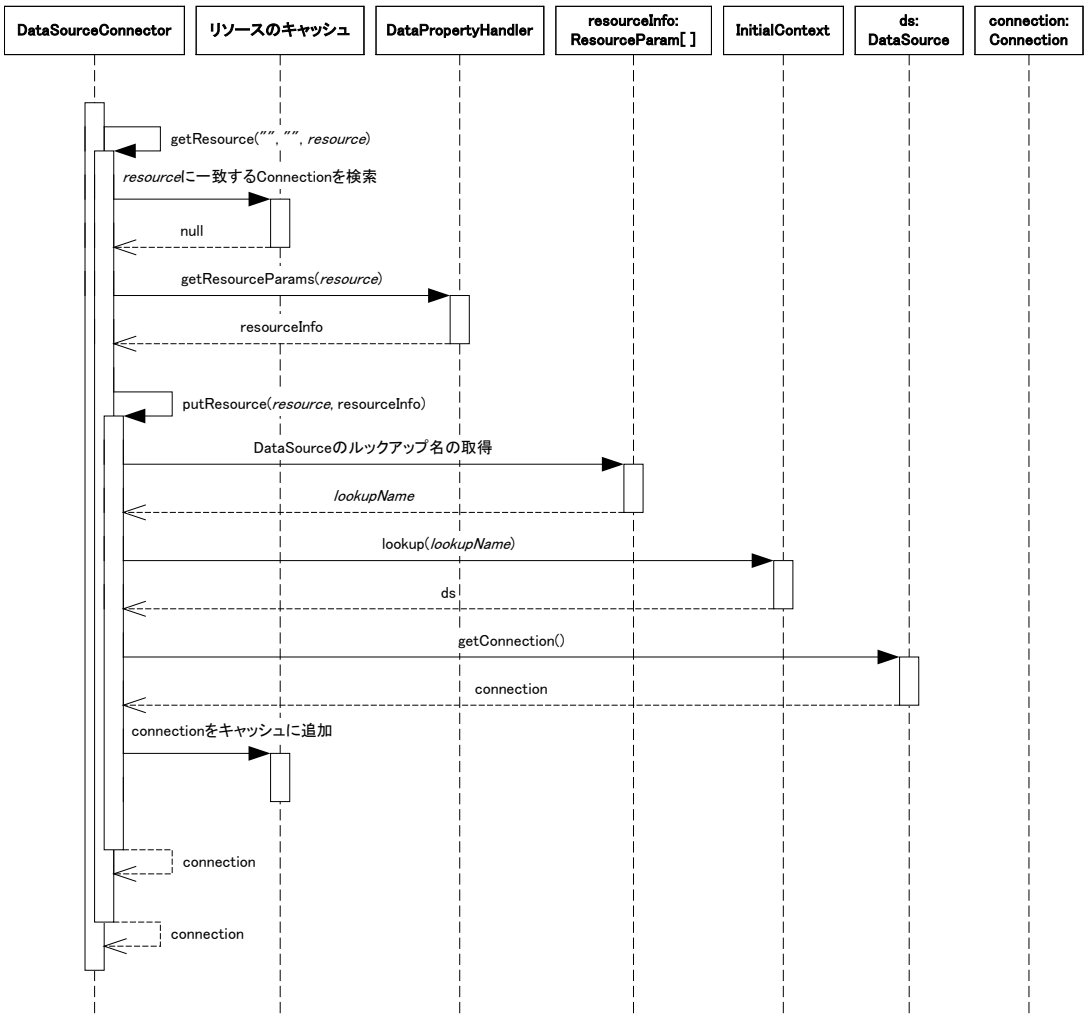


図 5-25 Connection の取得（DataSourceConnector）

DataSourceConnector に対してコミットまたはロールバックが行われるときの様子を「図 5-26 コミットまたはロールバック（DataSourceConnector）」に示す。

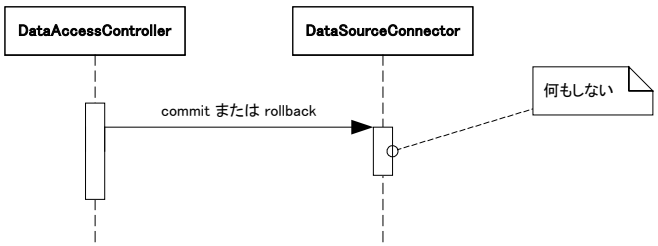


図 5-26 コミットまたはロールバック（DataSourceConnector）

5.3.3.1.4 **TenantDBConnector**

TenantDBConnector は intra-mart のテナントデータベースに設定された RDBMS に接続する機能を提供する。TenantDBConnector は主に TenantDBDAO で利用されることを目的としている。intra-mart で設定される RDBMS は DataSource を使用して接続するので、DataSourceConnector と同様のメリットが得られる。

TenantDBConnector を利用する場合、データプロパティに「表 5-3 TenantDBConnector の設定内容」で示す設定が必要となる。

表 5-3 TenantDBConnector の設定内容

項目	内容
DataConnector	jp.co.intra_mart.framework.base.data.TenantDBConnector
データコネクタ名	任意
リソース名	（なし）

DataPropertyHandler として XmlDataPropertyHandler を指定した場合、「リスト 5-7 data-config.xml の設定例（TenantDBConnector）」に対応する設定内容の例を「リスト 5-8 data-config-application.xml の設定例（TenantDBConnector）」に示す。この例ではアプリケーション ID を application、データコネクタ名を myCon としている。

リスト 5-7 data-config.xml の設定例（TenantDBConnector）

```
・・・
<connector>
    <connector-name>tenant_db</connector-name>
<connector-class>
jp.co.intra_mart.framework.base.data.TenantDBConnector
</connector-class>
  </connector>
・・・
```

リスト 5-8 data-config-application.xml の設定例（TenantDBConnector）

```
・・・
<dao-group>
  <dao-key>key</dao-key>
  <dao>
<dao-class>・・・</dao-class>
    <connector-name>tenant_db</connector-name>
  </dao>
</dao-group>
・・・
```

TenantDBConnector の構造を「図 5-27 TenantDBConnector の構造」に示す。

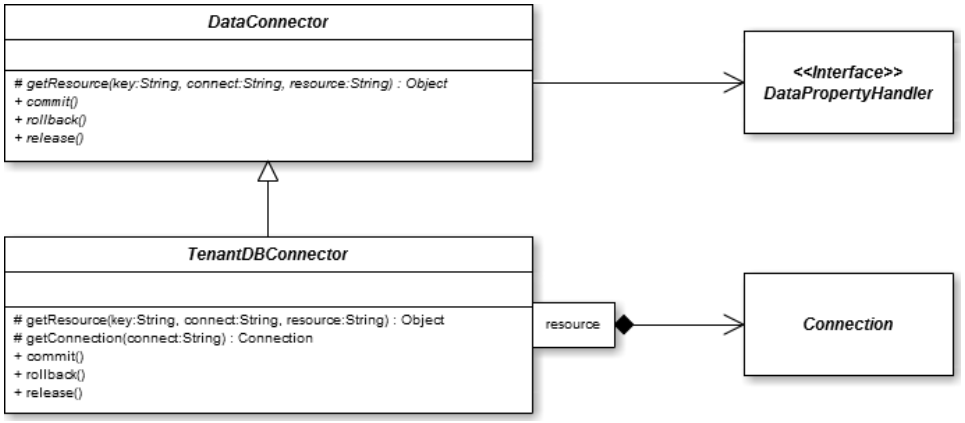


図 5-27 TenantDBConnector の構造

TenantDBConnector を使って Connection を取得する様子を「図 5-28 Connection の取得（TenantDBConnector）」に示す。DAO を使ってコーディングを行う開発者は、通常はこの動作について深く理解する必要はない。

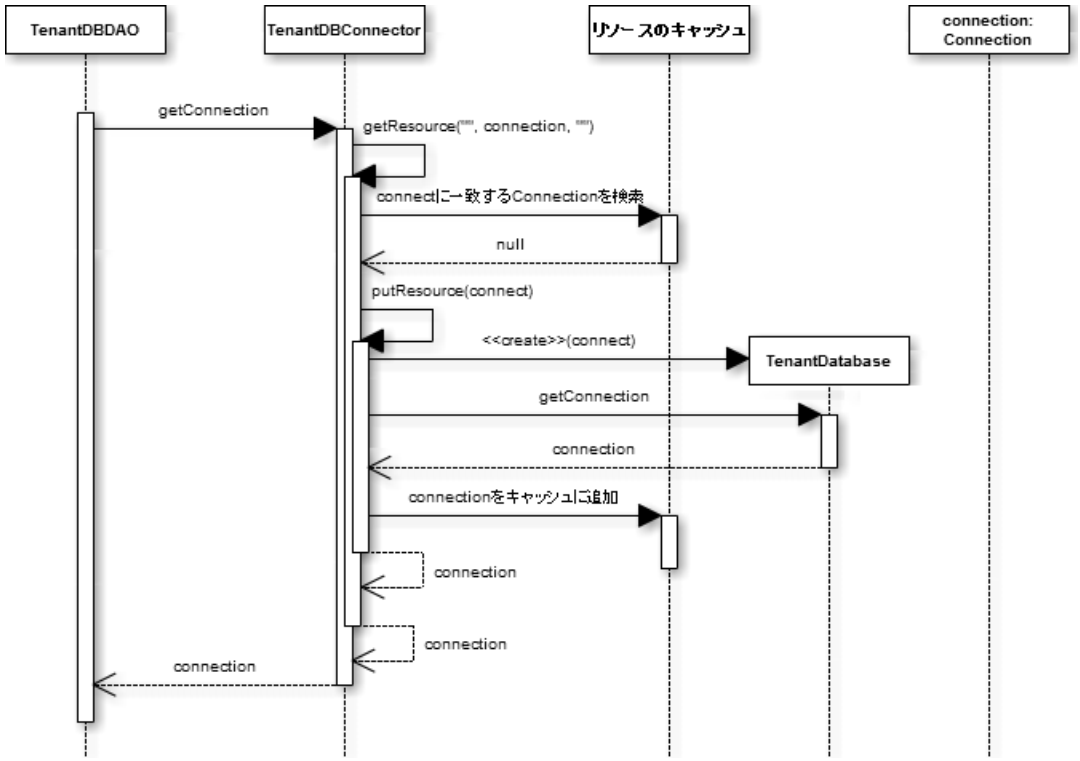


図 5-28 Connection の取得（TenantDBConnector）

TenantDBConnector ではコミットまたはロールバックに対して何も行わない。TenantDBConnector に対してコミットまたはロールバックが行われるときの様子を「図 5-29 コミットまたはロールバック（TenantDBConnector）」に示す。

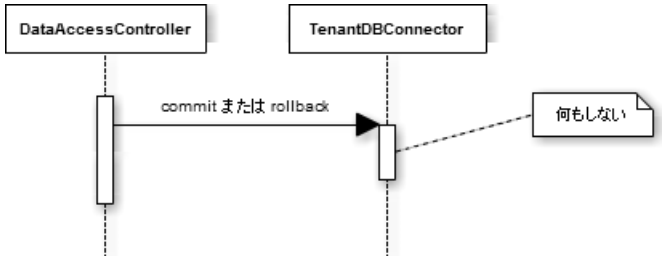


図 5-29コミットまたはロールバック（TenantDBConnector）

5.3.3.1.5 SharedDBConnector

SharedDBConnector は intra-mart のシェアードデータベースに設定された RDBMS に接続する機能を提供する。SharedDBConnector は主に SharedDBDAO で利用されることを目的としている。intra-mart で設定される RDBMS は DataSource を使用して接続するので、DataSourceConnector と同様のメリットが得られる。

SharedDBConnector を利用する場合、データプロパティに「図 5-30 SharedDBConnector の設定内容」で示す設定が必要となる。

図 5-30 SharedDBConnector の設定内容

項目	内容
DataConnector	jp.co.intra_mart.framework.base.data.SharedDBConnector
データコネクタ名	任意
リソース名	（なし）

DataPropertyHandler として XmlDataPropertyHandler を指定した場合、「リスト 5-9 data-config.xml の設定例（SharedDBConnector）」に対応する設定内容の例を「リスト 5-10 data-config-application.xml の設定例（SharedDBConnector）」に示す。この例ではアプリケーション ID を application、データコネクタ名を myCon としている。

リスト 5-9 data-config.xml の設定例（SharedDBConnector）

```
・・・
<connector>
    <connector-name>shared_db</connector-name>
<connector-class>
jp.co.intra_mart.framework.base.data.SharedDBConnector
</connector-class>
  </connector>
・・・
```

リスト 5-10 data-config-application.xml の設定例（SharedDBConnector）

```
・・・
<dao-group>
  <dao-key>key</dao-key>
  <dao>
<dao-class>・・・</dao-class>
    <connector-name>shared_db</connector-name>
  </dao>
</dao-group>
・・・
```

SharedDBConnector の構造を「図 5-31 SharedDBConnector の構造」に示す。

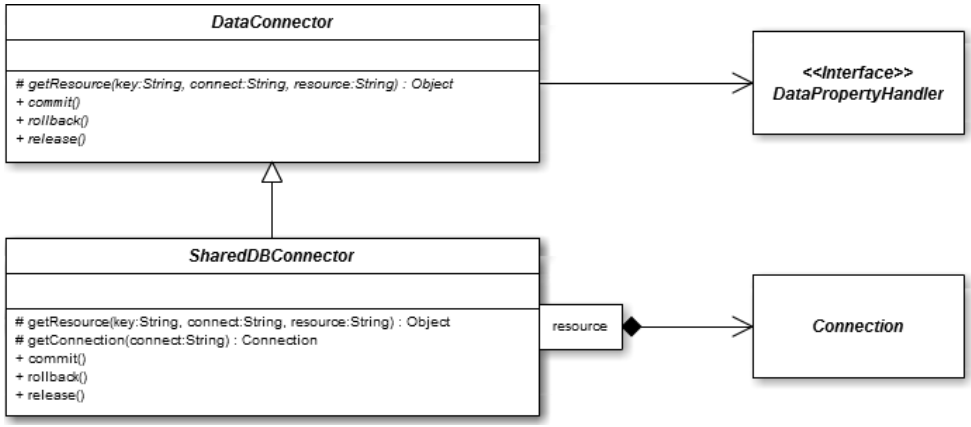


図 5-31 SharedDBConnector の構造

SharedDBConnector を使って Connection を取得する様子を「図 5-32 Connection の取得（SharedDBConnector）」に示す。DAO を使ってコーディングを行う開発者は、通常はこの動作について深く理解する必要はない。

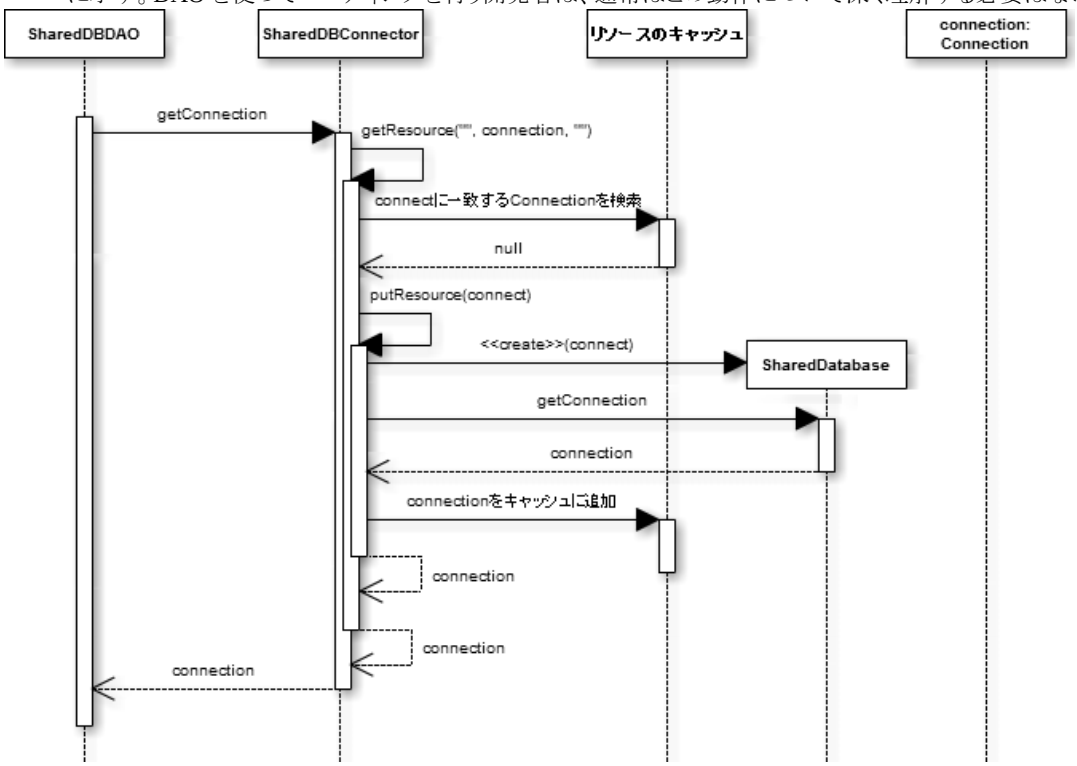


図 5-32 Connection の取得（SharedDBConnector）

SharedDBConnector ではコミットまたはロールバックに対して何も行わない。SharedDBConnector に対してコミットまたはロールバックが行われるときの様子を「図 5-33 コミットまたはロールバック（SharedDBConnector）」に示す。

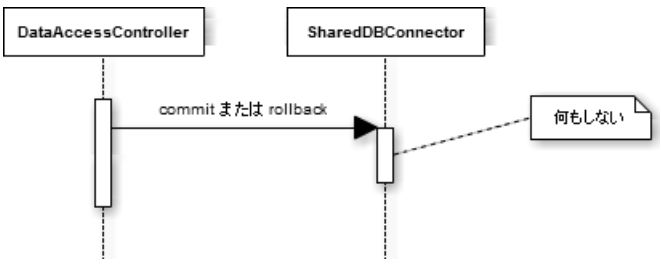


図 5-33 コミットまたはロールバック（SharedDBConnector）

5.3.3.1.6 **IntramartDBConnector**

IntramartDBConnector は intra-mart で設定された RDBMS に接続する機能を提供する。IntramartDBConnector は主に IntramartDBDAO で利用されることを目的としている。intra-mart で設定される RDBMS は DataSource を使用して接続するので、DataSourceConnector と同様のメリットが得られる。

DbsConnection は非推奨メソッドであり、互換モジュールをインストールしなければ動作しない。そのため IntramartDBDAO は互換モジュールをインストールしたときに動作する。

TenantDBDAO または SharedDBDAO を使用するべきである。

IntramarDBConnector を利用する場合、データプロパティに「表 5-4 IntramartDBConnector の設定内容」に示す設定が必要となる。

表 5-4 IntramartDBConnector の設定内容

項目	内容
DataConnector	jp.co.intra_mart.framework.base.data.IntramartDBConnector
データコネクタ名	任意
リソース名	（なし）

DataPropertyHandler として XmlDataPropertyHandler を指定した場合、「表 5-2 DataSourceConnector の設定内容」に対応する設定内容の例を「リスト 5-11 data-config.xml の設定例（IntramartDBConnector）」と「リスト 5-12 data-config-application.xml の設定例（IntramartDBConnector）」に示す。この例ではアプリケーション ID を application、データコネクタ名を myCon としている。

リスト 5-11 data-config.xml の設定例（IntramartDBConnector）

```
・・・
<connector>
   <connector-name>intra_mart_db</connector-name>
<connector-class>
jp.co.intra_mart.framework.base.data.IntramartDBConnector
</connector-class>
  </connector>
・・・
```

リスト 5-12 data-config-application.xml の設定例（IntramartDBConnector）

```
・・・
<dao-group>
  <dao-key>key</dao-key>
  <dao>
<dao-class>・・・</dao-class>
    <connector-name>intra_mart_db</connector-name>
  </dao>
</dao-group>
・・・
```

IntramartDBConnector の構造を「図 5-34 IntramartDBConnector の構造」に示す。

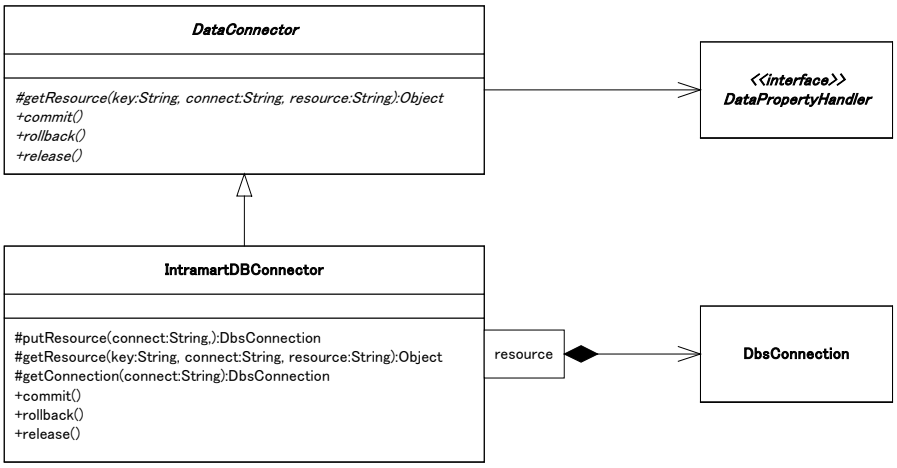


図 5-34 IntramartDBConnector の構造

IntramartDBConnector を使って DbsConnection を取得する様子を「図 5-35 Connection の取得（IntramartDBConnector）」に示す。DAO を使ってコーディングを行う開発者は、通常はこの動作について深く理解する必要はない。

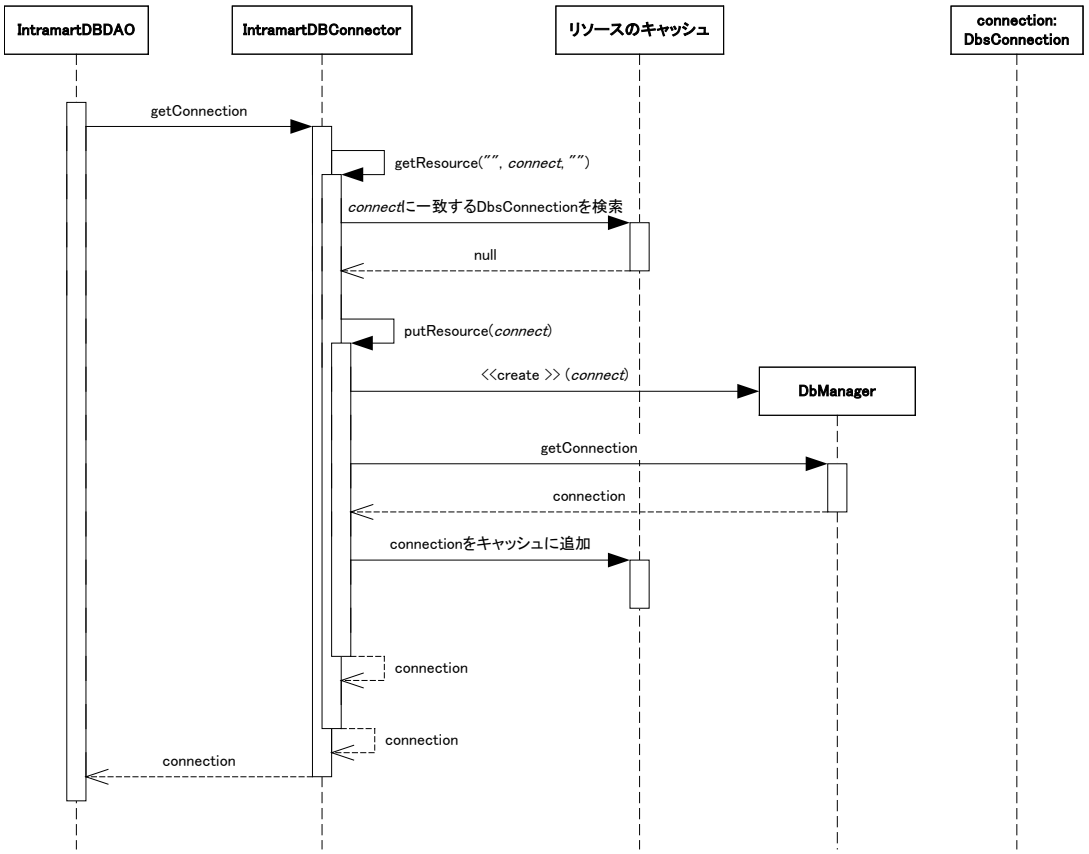


図 5-35 Connection の取得（IntramartDBConnector）

IntramartDBConnector に対してコミットまたはロールバックが行われるときの様子を「図 5-36 コミットまたはロールバック（IntramartDBConnector）」に示す。

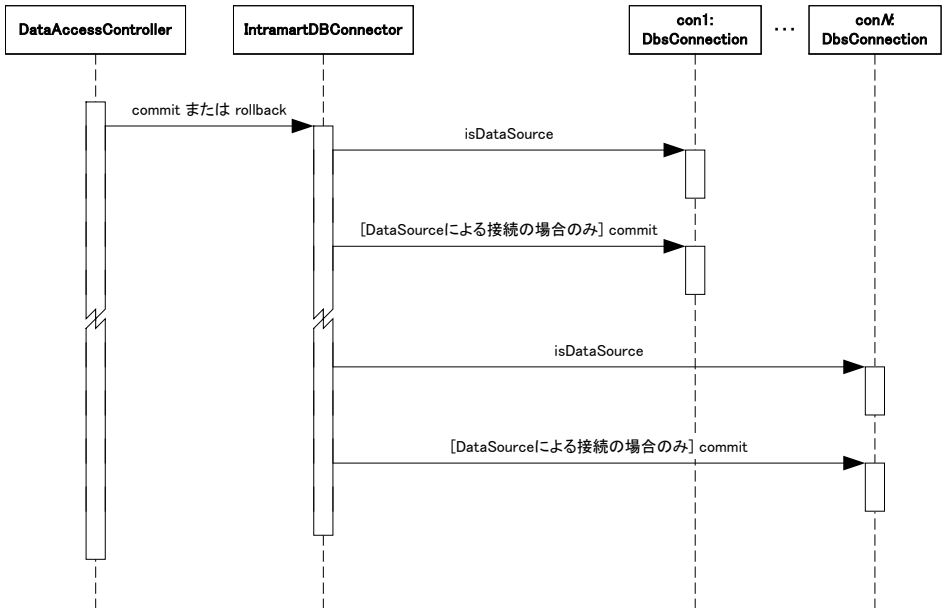


図 5-36 コミットまたはロールバック（IntramartDBConnector）

5.3.3.1.7 **IntramartStorageConnector**

IntramartStorageConnector は intra-mart の Public Storage に接続する機能を提供する。IntramartStorageConnector は主に IntramartStorageDAO で利用されることを目的としている。

IntramartStorageConnector を利用する場合、データプロパティに「表 5-5 IntramartStorageConnector の設定内容」に示す設定が必要となる。

表 5-5 IntramartStorageConnector の設定内容

項目	内容
DataConnector	jp.co.intra_mart.framework.base.data.IntramartStorageConnector
データコネクタ名	任意
リソース名	（なし）

DataPropertyHandler として DefaultDataPropertyHandler を指定した場合、「表 5-2 DataSourceConnector の設定内容」に対応する設定内容の例を「リスト 5-11 data-config.xml の設定例（IntramartDBConnector）」と「リスト 5-12 data-config-application.xml の設定例（IntramartDBConnector）」に示す。ここではアプリケーション ID を application、データコネクタ名を myCon としている。

リスト 5-13 DataConfig.properties の設定例（IntramartStorageConnector）

```
…
<connector>
   <connector-name>intra_mart_storage</connector-name>
<connector-class>
jp.co.intra_mart.framework.base.data. IntramartStorageConnector
</connector-class>
  </connector>
…
```

リスト 5-14 DataConfig_application.properties の設定例（IntramartStorageConnector）

```
・・・
<dao-group>
  <dao-key>key</dao-key>
  <dao>
<dao-class>・・・</dao-class>
    <connector-name>intra_mart_storage</connector-name>
  </dao>
</dao-group>
・・・
```

IntramartFileServerConnector の構造を「図 5-37 IntramartStorageConnector の構造」に示す。

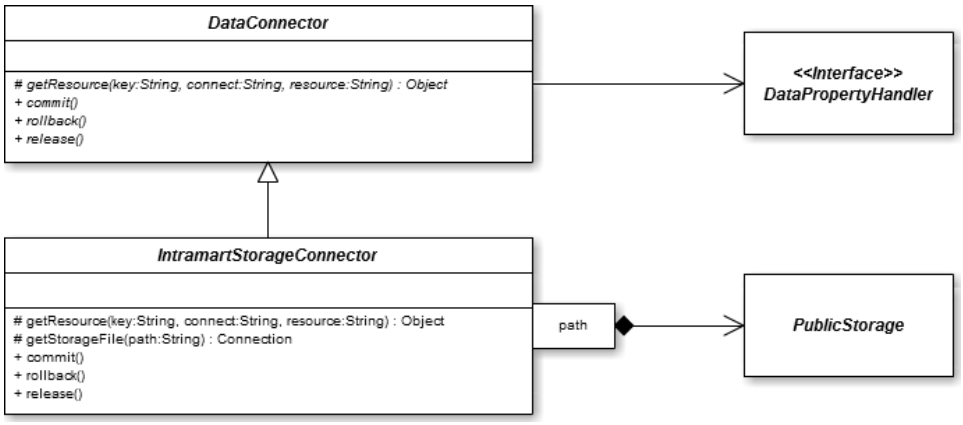


図 5-37 IntramartStorageConnector の構造

IntramartStorageConnector を使って PublicStorage を取得する様子を「図 5-38 PublicStorage の取得」に示す。DAO を使ってコーディングを行う開発者は、通常はこの動作について深く理解する必要はない。

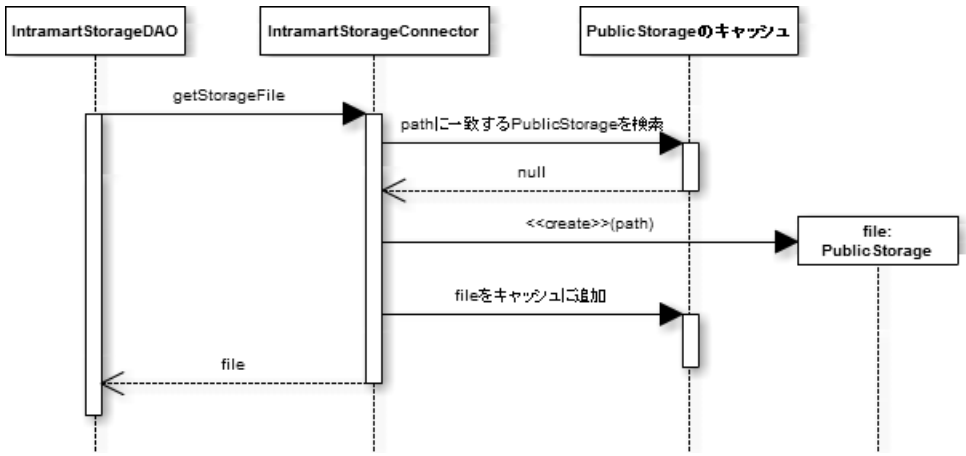


図 5-38 PublicStorage の取得

IntramartStorageConnector ではコミットまたはロールバックに対して何も行わない。コミットまたはロールバックが行われるときの様子を「図 5-39 コミットまたはロールバック（IntramartStorageConnector）」に示す。

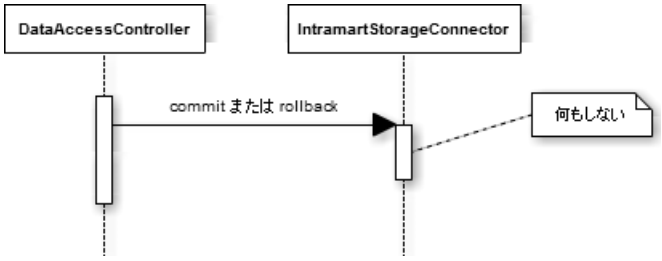


図 5-39 コミットまたはロールバック（IntramartStorageConnector）

5.3.3.2 独自の DataConnector

IM-JavaEE Framework で提供されていないリソースにアクセスする場合、DataConnector を独自に開発する必要がある。DataConnector を独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.data.DataConnector クラスを継承している。
- public なデフォルトコンストラクタ（引数なしのコンストラクタ）が定義されている。
- 以下のメソッドがそれぞれ適切な動作をするよう実装されている。
  - getResource
  - commit
  - rollback
  - release

5.4 データフレームワークに関連するプロパティ

IM-JavaEE Framework のデータフレームワークではさまざまなプロパティを外部で設定することが可能である。データプロパティの取得は jp.co.intra_mart.framework.base.data.DataPropertyHandler インタフェースを実装したクラスから取得する。IM-JavaEE Framework ではこのインタフェースを実装した複数の実装クラスを標準で提供している（「図 5-40 DataPropertyHandler」を参照）。データプロパティの設定方法は IM-JavaEE Framework では特に規定してなく、前述のインタフェースを実装したクラスに依存する。

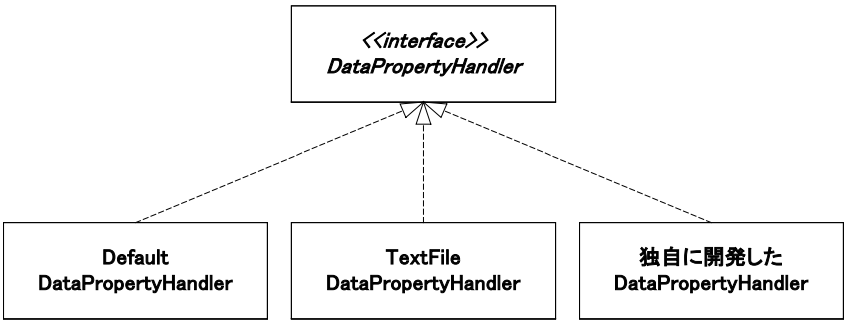


図 5-40 DataPropertyHandler

5.4.1 データフレームワークに関連するプロパティの取得

データフレームワークに関連するプロパティは DataPropertyHandler から取得する。DataPropertyHandler は jp.co.intra_mart.framework.data.DataManager の getDataPropertyHandler メソッドで取得することができる。DataPropertyHandler は必ずこのメソッドを通じて取得されたものである必要があり、開発者が自分でこの DataPropertyHandler の実装クラスを明示的に生成（new による生成や java.lang.Class の newInstance メソッド、またはリフレクションを利用したインスタンスの生成）をしてはならない。

DataPropertyHandler の取得とプロパティの取得に関連する手順を「図 5-41 DataPropertyHandler の取得」に示す。

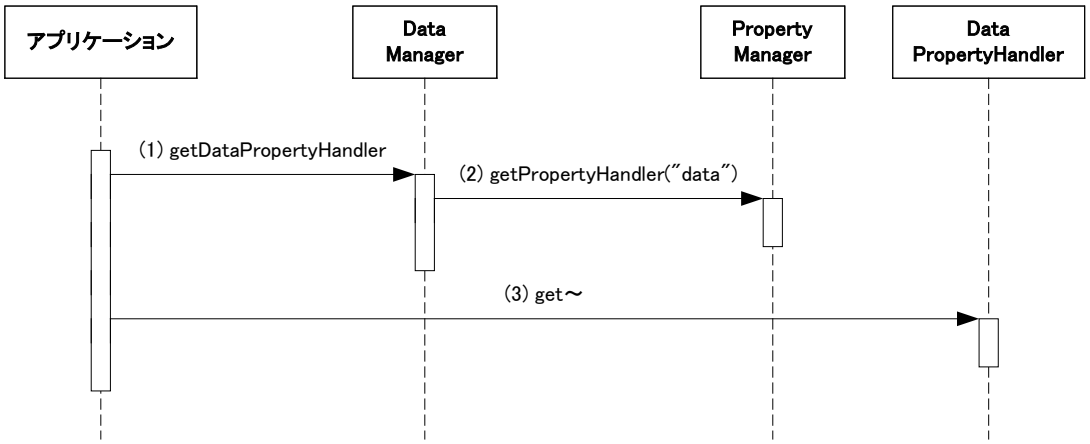


図 5-41 DataPropertyHandler の取得

1. DataManager から DataPropertyHandler を取得する。
2. DataManager の内部では PropertyManager から DataPropertyHandler を取得し、アプリケーションに返している。(この部分は DataManager の内部で行っていることであり、開発者は特に意識する必要はない。)
3. DataPropertyHandler を利用して各種のプロパティを取得する。

5.4.2 標準で用意されている DataPropertyHandler

5.4.2.1 DefautlDataPropertyHandler

jp.co.intra_mart.framework.base.data.DefaultDataPropertyHandler として提供されている。

プロパティの設定はリソースファイルで行う。リソースファイルの内容は「プロパティ名=プロパティの値」という形式で設定する。使用できる文字などは java.util.ResourceBundle に従う。このリソースファイルは使用するアプリケーションから取得できるクラスパスにおく必要がある。リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

5.4.2.2 TextFileDataPropertyHandler

jp.co.intra_mart.framework.base.data.TextFileDataPropertyHandler として提供されている。

DefaultDataPropertyHandler と同じ形式のリソースファイルを利用するが、以下の点が違う。

- クラスパスに通す必要がない。
- アプリケーションから参照できる場所であれば、ファイルシステムの任意の場所に配置できる。
- 設定によってはアプリケーションを停止しないでリソースファイルの再読み込みが可能となる。

リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

5.4.2.3 XmlDataPropertyHandler

jp.co.intra_mart.framework.base.data.XmlDataPropertyHandler として提供されている。
プロパティの設定は XML 形式で行う。アプリケーションから取得できるクラスパスにおく必要があり、固有の ID と Java パッケージパスを含めたものをアプリケーション ID として認識する。例えばアプリケーション ID が ”foo.bar.example”の場合、クラスパスに ” foo/bar/data-config-example.xml”として配置する。
また、XmlDataPropertyHandler は動的読み込みに対応している。
詳細は API リストを参照。

5.4.3 独自の DataPropertyHandler

DataPropertyHandler を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.data.DataPropertyHandler インタフェースを実装している。
- public なデフォルトコンストラクタ（引数なしのコンストラクタ）が定義されている。
- すべてのメソッドに対して適切な値が返ってくる（「5.4.4 プロパティの内容」参照）。
- isDynamic()メソッドが false を返す場合、プロパティを取得するメソッドはアプリケーションサーバを再起動しない限り値は変わらない。

5.4.4 プロパティの内容

データに関連するプロパティの設定方法は運用時に使用する DataPropertyHandler の種類によって違うが、概念的には同じものである。

データに関連するプロパティの内容は以下のとおりである。

5.4.4.1 共通

5.4.4.1.1 動的読み込み

isDynamic()メソッドで取得可能。

このメソッドの戻り値が true である場合、このインタフェースで定義される各プロパティ取得メソッド（get～メソッド）は毎回設定情報を読み込みに行くように実装されている必要がある。false である場合、各プロパティ取得メソッドはパフォーマンスを考慮して取得される値を内部でキャッシュしてもよい。

5.4.4.1.2 DataConnector

getConnectorClassName(String connectorName)メソッドで取得可能。

データコネクタ名に対応する DataConnector のクラス名を設定する。未設定の場合、DataPropertyException を返す。このメソッドの引数には「5.4.4.2.2 データコネクタ名」または「5.4.4.2.3 データコネクタ名（接続名指定）」で取得した DataConnector の論理名を指定する。
ここで指定するクラスは jp.co.intra_mart.framework.base.data.DataConnector インタフェースを実装している必要がある。

5.4.4.1.3 リソース名

getConnectorResource(String connectorName)メソッドで取得可能。

データコネクタ名に対応するリソース名を設定する。対応するリソース名がない場合 null を返す。

5.4.4.1.4 リソースパラメータ

getResourceParams(String name)メソッドで取得可能。

リソース名に対応するリソースのパラメータを設定する。対応するパラメータがない場合サイズが 0 の配列が返される。

5.4.4.2 アプリケーション個別

5.4.4.2.1 DAO

getDAOName(String application, String key, String connect)メソッドで取得可能。

アプリケーション ID、キーおよび接続名に対応する DAO のクラス名を設定する。未設定の場合、接続名が指定されていない場合の DAO の検索結果と同一となる。
ここで指定するクラスは jp.co.intra_mart.framework.base.data.DAO インタフェースを実装している必要がある。

5.4.4.2.2 データコネクタ名

getConnectorName(String application, String key)メソッドで取得可能。

アプリケーション ID およびキーに対応する DataConnector の論理名を設定する。未設定の場合 null を返す。

5.4.4.2.3 データコネクタ名（接続名指定）

getConnectorName(String application, String key, String connect)メソッドで取得可能。

アプリケーション ID、キーおよび接続名に対応する DataConnector の論理名を設定する。未設定の場合「5.4.4.2.2 データコネクタ名」で取得される値を返す。

5.4.5 DataConnector とリソースの関係

パラメータ設定上の DAO、DataConnector およびリソースの関係は「図 5-42 DataConnector とリソースの関係」に示すようなものとなる。

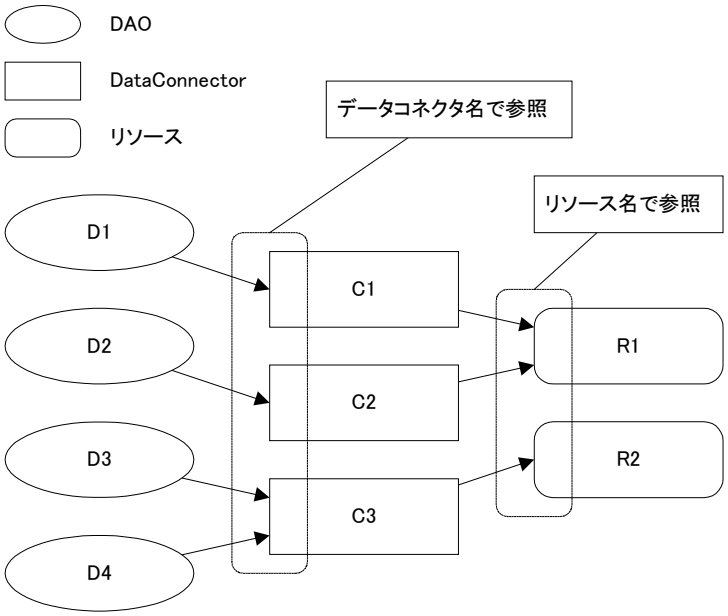


図 5-42 DataConnector とリソースの関係

DAO、DataConnector およびリソースの関係は次のようになっている。

- 1 つの DAO は 1 つの DataConnector を参照する。
- 1 つの DataConnector は複数の DAO から参照される。
- 1 つの DataConnector は 1 つのリソースを参照する。
- 1 つのリソースは複数の DataConnector から参照される。

これらの関係を理解しておけば、DataConnector やリソースの設定は同じものをコピーすることなく最小限で済ませることができる。

5.5 トランザクション

IM-JavaEE Framework のデータフレームワークを使用する場合、トランザクションの管理についていくつか注意する点がある。

5.5.1 トランザクションの種類

EIS 層へのアクセス時にトランザクションを管理する方法として以下のものがある。

- UserTransaction（Java Transaction API：JTA[7]を参照）によるトランザクション管理
- DataAccessController によるトランザクション管理
- 上記のトランザクション管理の併用

トランザクション管理については UserTransaction のみを利用する方法を推奨する。DataAccessController によるトランザクション管理は簡易的なものであり、2 フェーズコミットなどの高度なトランザクション要件を満たしていない。

5.5.1.1 UserTransaction によるトランザクション管理

UserTransaction によるトランザクション管理の様子を「図 5-43 UserTransaction によるトランザクション管理」に示す。この場合、UserTransaction は DataAccesController から取得した DAO に関連する DataConnector に対してコミットもロールバックもしていない点に注意する。

UserTransaction でトランザクション管理を行う場合、DataConnector は UserTransaction のトランザクションに対応するものでなければならない。この場合、DataConnector の以下のメソッドの実装はトランザクション処理に関して何もしないことが望ましい。

- commit
- rollback

IM-JavaEE Framework で用意されている DataConnector では以下のものが上記の条件に該当する。

- jp.co.intra_mart.framework.base.data.DataSourceConnector
- jp.co.intra_mart.framework.base.data.TenantDBConnector
- jp.co.intra_mart.framework.base.data.SharedDBConnector
-

上記の DataConnector を使用する場合、使用されるデータソースはトランザクション処理可能なものでなければならない。

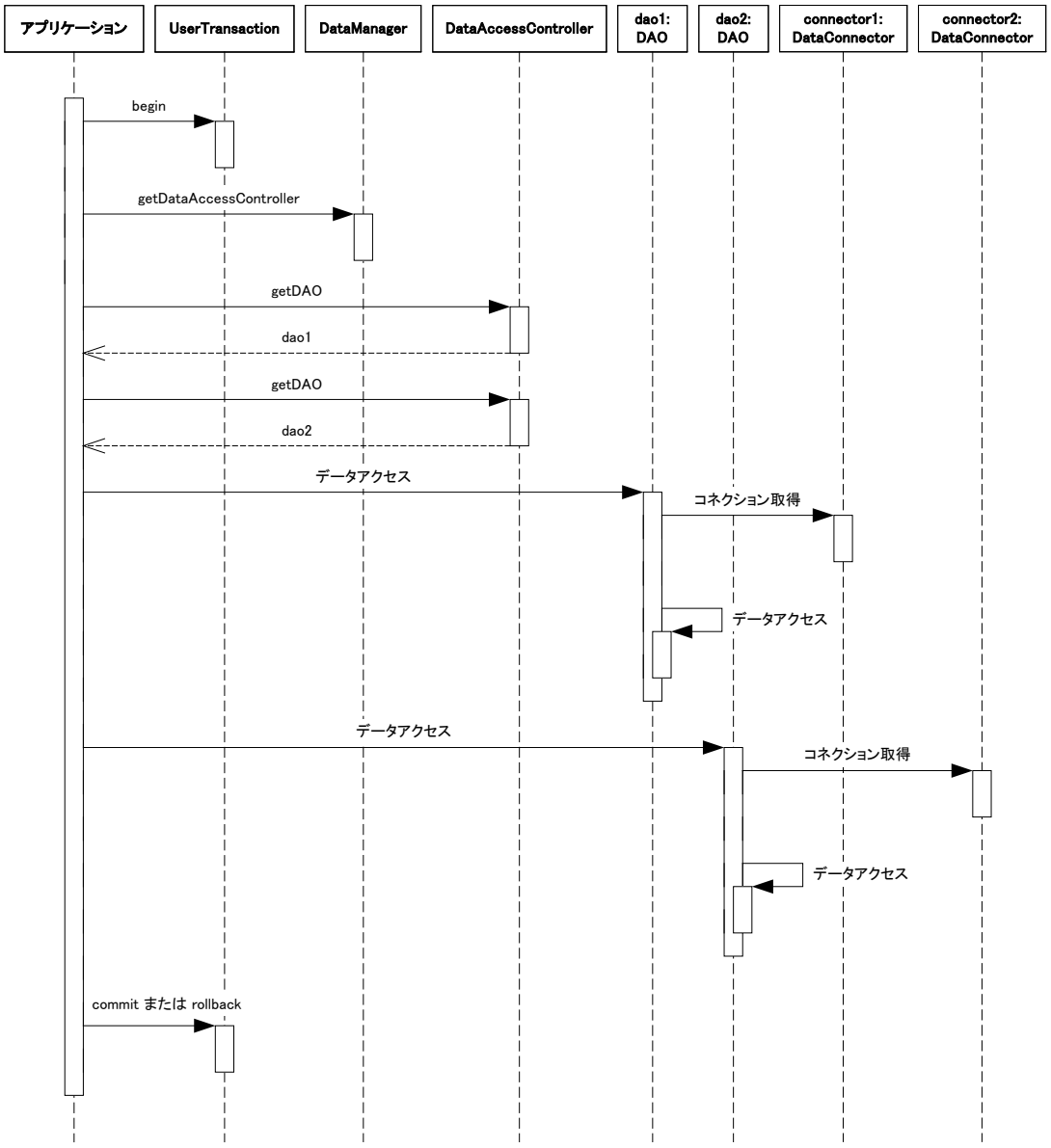


図 5-43 UserTransaction によるトランザクション管理

5.5.1.2 DataAccessController によるトランザクション管理

DataAccessControllerによるトランザクション管理の様子を「図 5-44 DataAccessControllerによるトランザクション管理」に示す。この場合、DataAccesController が DataConnector それぞれに対してコミットやロールバックをしている点に注意する。これは、複数の DataConnector に対してはコミットが成功し、複数の DataConnector に対してはコミットが失敗する可能性があることを意味する。そのためこのトランザクション管理方法は推奨されない。

DataAccessController でトランザクション管理を行う場合、DataConnector は以下のメソッドが正しく実装されている必要がある。

- commit
- rollback

IM-JavaEE Framework で用意されている DataConnector では以下のものが上記の条件に該当する。

- jp.co.intra_mart.framework.base.data.JDBCConnector

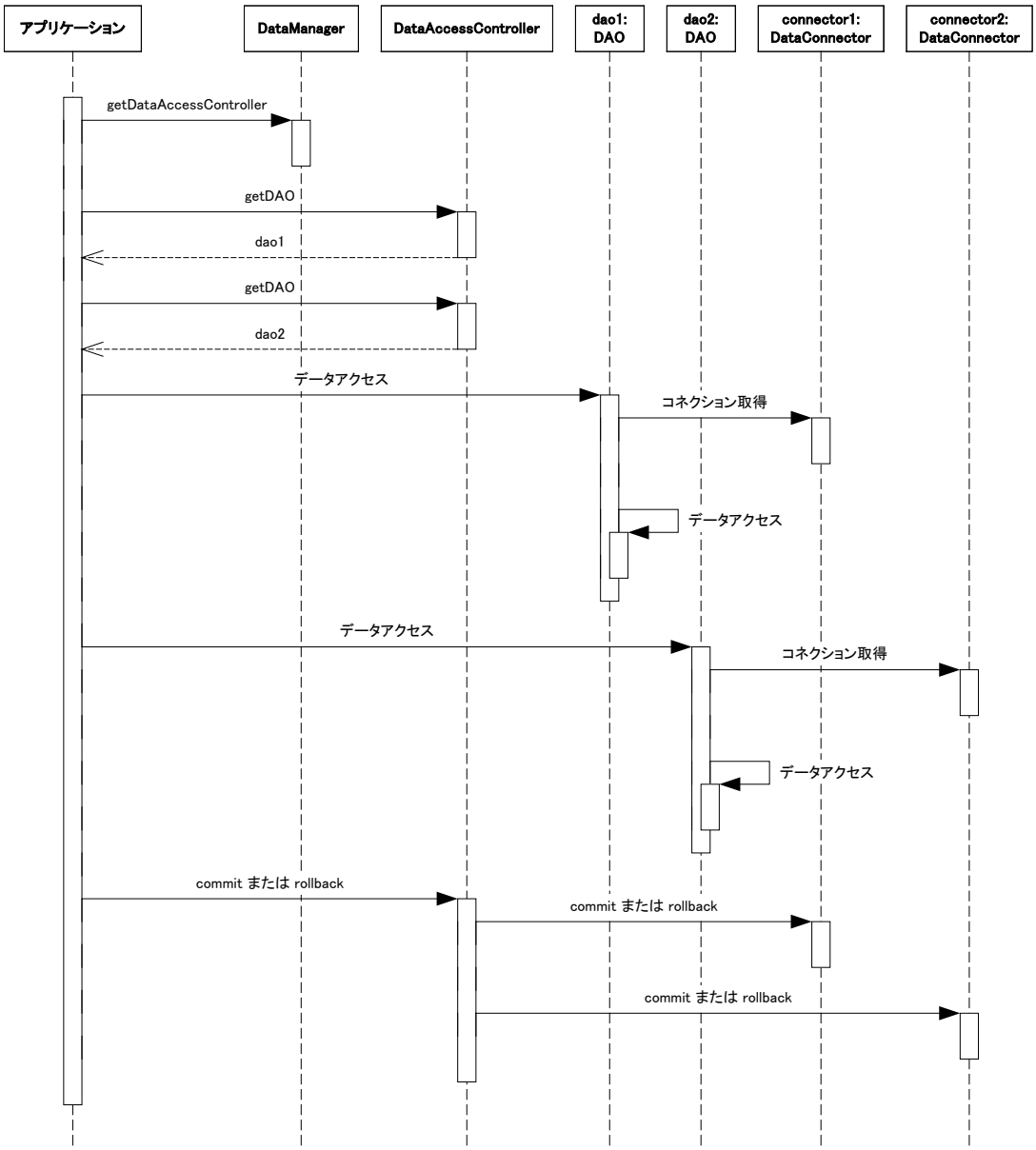


図 5-44 DataAccessController によるトランザクション管理

5.5.1.3 併用によるトランザクション管理

UserTransaction に対応する DataConnector と DataAccessController に対応する DataConnector の両方が混在する場合、両者を併用してトランザクション管理を行う必要がある。この場合の様子を「図 5-45 併用によるトランザクション管理」に示す。

この場合も「5.5.1.2 DataAccessController によるトランザクション管理」で示した場合と同じ欠点があるため、この方法は推奨されない。

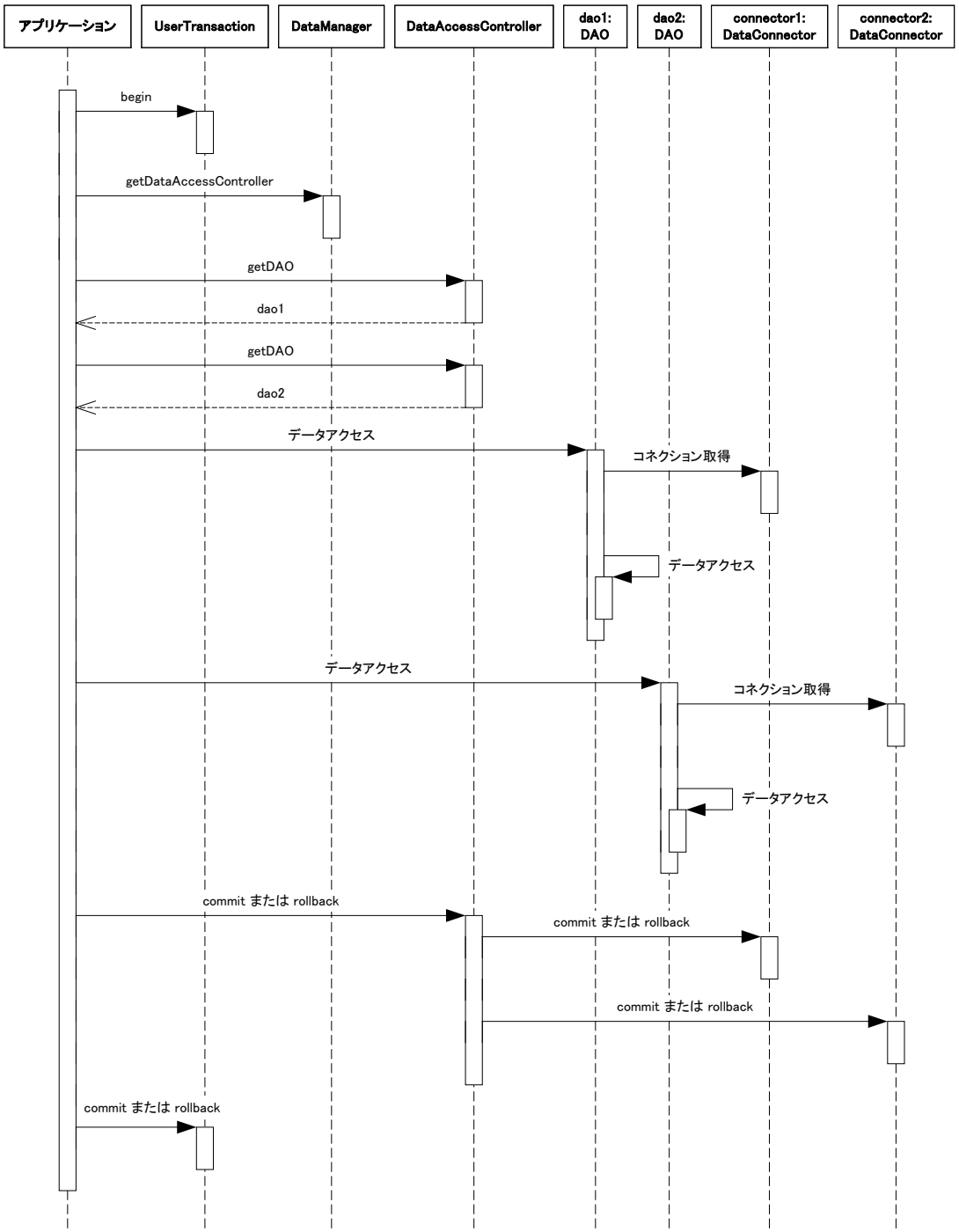


図 5-45 併用によるトランザクション管理

5.5.2 トランザクションの例

ここではトランザクションが管理されている様子の例をいくつか示す。ここではDAOはすべてリレーショナルデータベースにアクセスするものとし、データベースは 2 フェーズコミットに対応しているものと仮定する。

5.5.2.1 単一 DAO + 単一コネクション（JDBCConnector）

JDBCConnector を利用してデータベースにアクセスする場合の処理内容を「図 5-46 単一 DAO、単一コネクション」に示す。

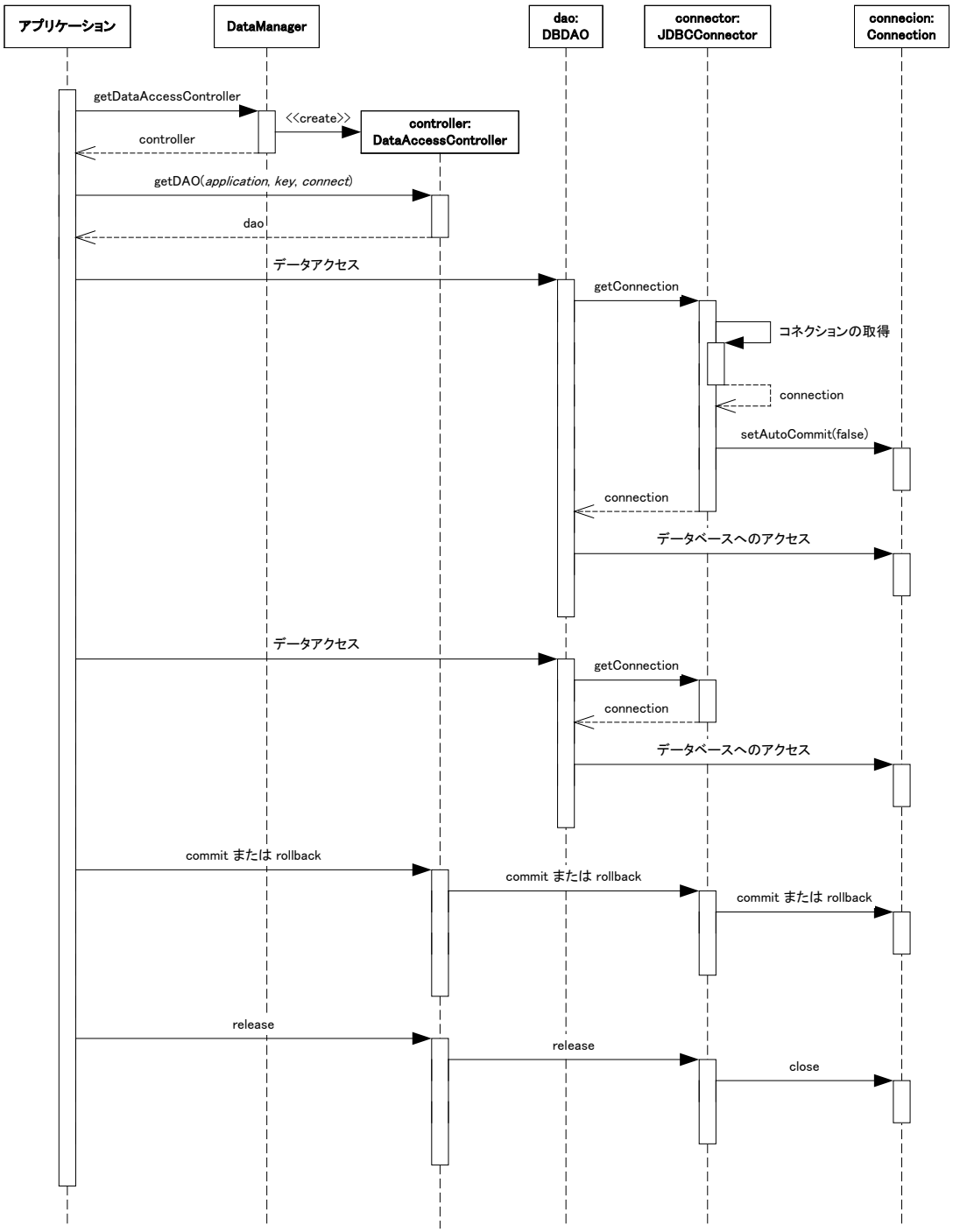


図 5-46 単一 DAO、単一コネクション

「図 5-46 単一 DAO、単一コネクション」に示す内容に該当するサンプルコードは「リスト 5-15 トランザクション処理（単一 DAO ＋ JDBCConnector）」のようになる。「リスト 5-15 トランザクション処理（単一 DAO ＋ JDBCConnector）」ではコードを簡略化するために、いくつかの例外処理については省略している。

リスト 5-15 トランザクション処理（単一 DAO ＋ JDBCConnector）

```
// DataAccessController の取得
DataManager dm = DataManager.getDataManager();
DataAccessController dac = dm.getDataAccessController();

// データアクセス処理
try {

	// dao の取得とデータアクセス
	MyDAOIF dao = (MyDAOIF)dac.getDAO("app", "key", "con");
	dao.access1(...);
	dao.access2(...);

	dac.commit();	// DataAccessController のコミット
} catch (Exception e) {

	try {
		dac.rollback();
	} catch (Exception ex) {
	}

} finally {

	// リソースの解放
	dac.release();

}
```

5.5.2.2 複数 DAO + 単一コネクション（JDBCConnector）

複数の DAO が JDBCConnector を利用してデータベースにアクセスする場合の処理内容を「図 5-47 複数 DAO、単一コネクション」に示す。

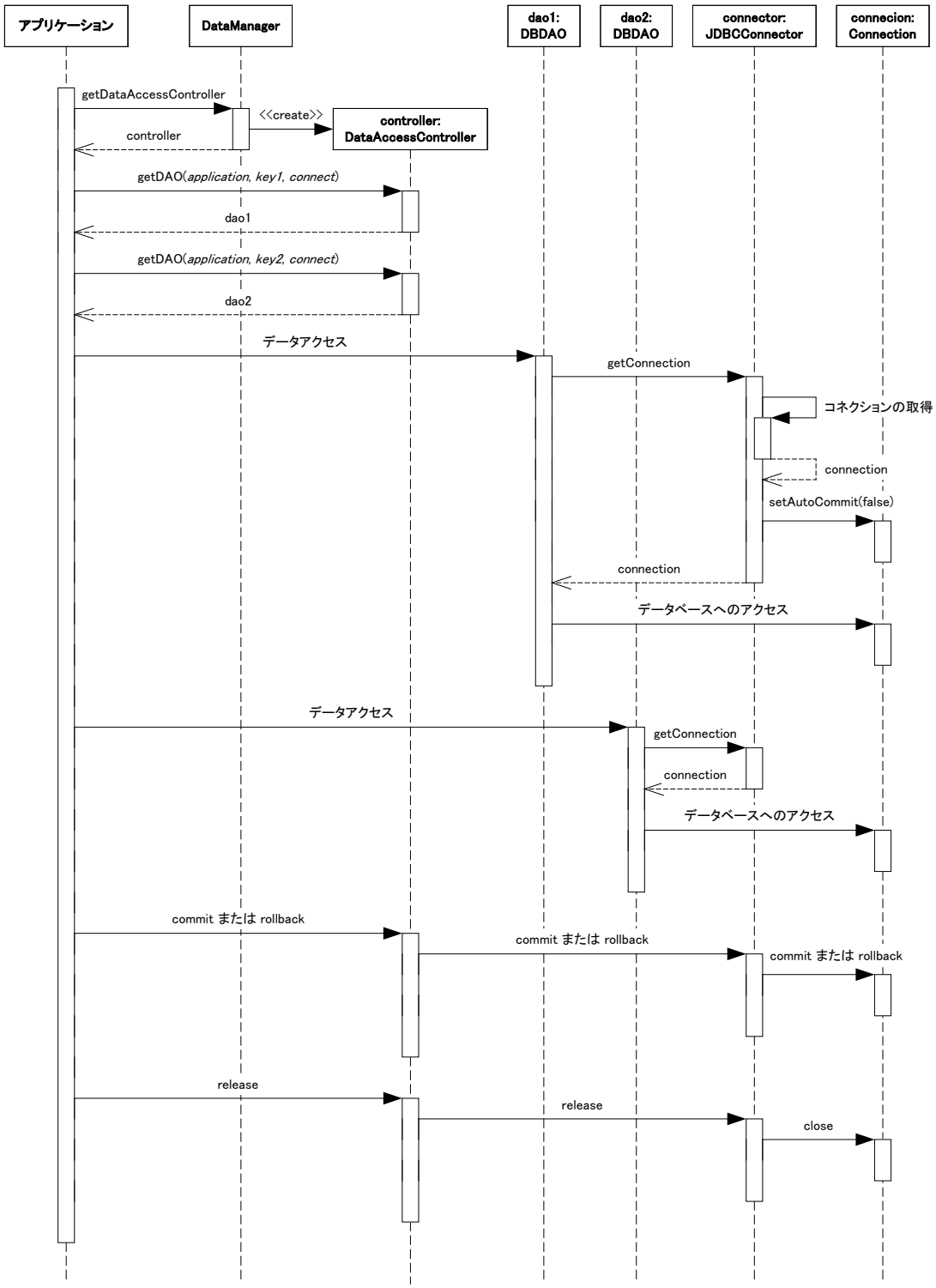


図 5-47 複数 DAO、単一コネクション

「図 5-47 複数 DAO、単一コネクション」に示す内容に該当するサンプルコードは「リスト 5-16 トランザクション処理（複数 DAO ＋ JDBCConnector）」のようになる。「リスト 5-16 トランザクション処理（複数 DAO ＋ JDBCConnector）」ではコードを簡略化するために、いくつかの例外処理については省略している。

リスト 5-16 トランザクション処理（複数 DAO ＋ JDBCConnector）

```
// DataAccessController の取得
DataManager dm = DataManager.getDataManager();
DataAccessController dac = dm.getDataAccessController();

// データアクセス処理
try {

	// dao1 の取得とデータアクセス
	MyDAO1IF dao1 = (MyDAO1IF)dac.getDAO("app", "key1", "con");
	dao1.access(...);

	// dao2 の取得とデータアクセス
	MyDAO2IF dao2 = (MyDAO2IF)dac.getDAO("app", "key2", "con");
	dao2.access(...);

	dac.commit();	// DataAccessController のコミット
} catch (Exception e) {

	try {
		dac.rollback();
	} catch (Exception ex) {
	}

} finally {

	// リソースの解放
	dac.release();

}
```

5.5.2.3 複数 DAO + 複数コネクション（JDBCConnector）

複数の DAO が複数の JDBCConnector を利用してデータベースにアクセスする場合の処理内容を「図 5-48 複数 DAO、複数コネクション」に示す。

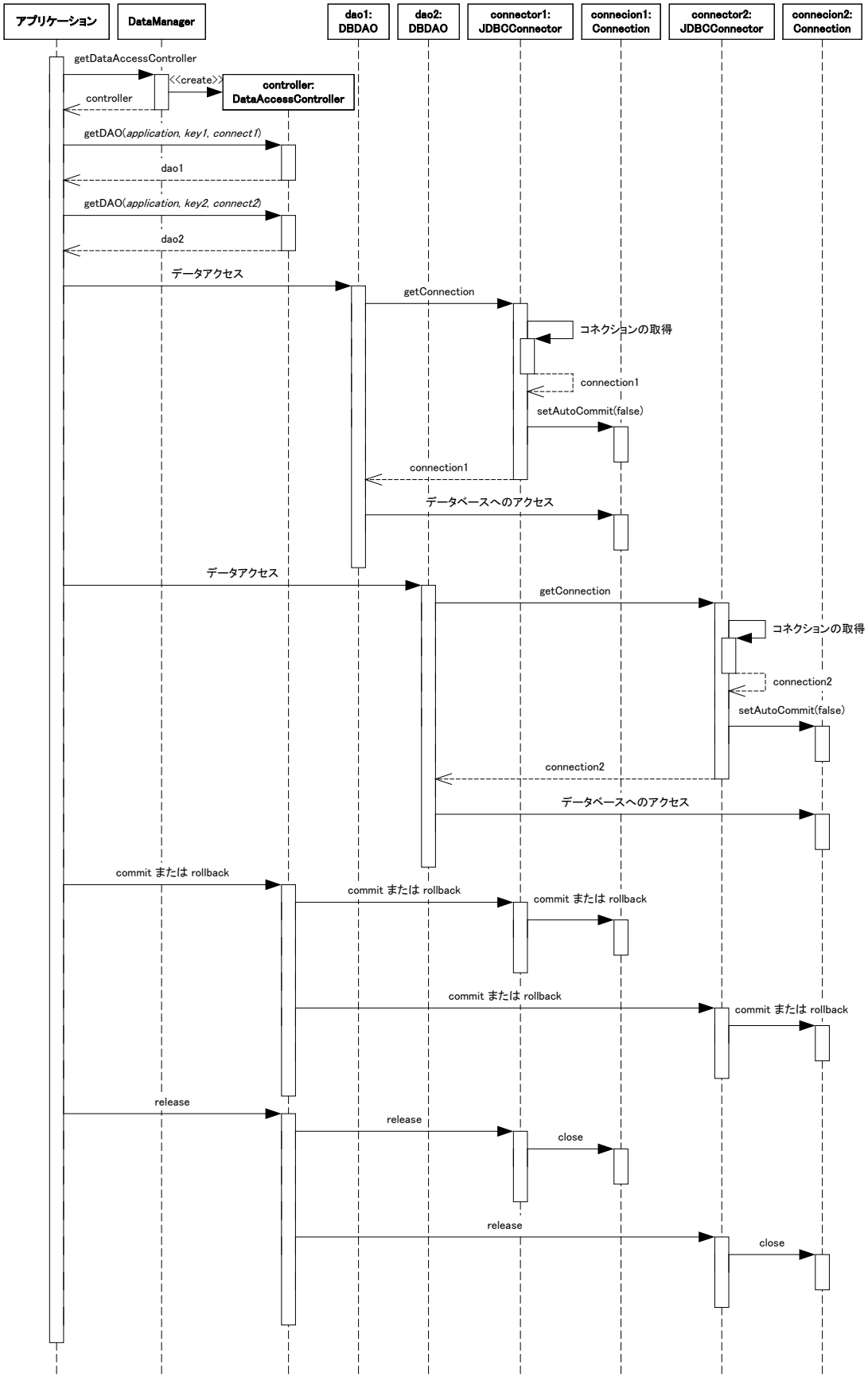


図 5-48 複数 DAO、複数コネクション

「図 5-48 複数 DAO、複数コネクション」に示す内容に該当するサンプルコードは「リスト 5-17 トランザクション処

理（複数 DAO ＋ 複数 JDBCConnector）」のようになる。「リスト 5-17 トランザクション処理（複数 DAO ＋ 複数 JDBCConnector）」ではコードを簡略化するために、いくつかの例外処理については省略している。

リスト 5-17 トランザクション処理（複数 DAO ＋ 複数 JDBCConnector）

```
// DataAccessController の取得
DataManager dm = DataManager.getDataManager();
DataAccessController dac = dm.getDataAccessController();

// データアクセス処理
try {

	// dao1 の取得とデータアクセス
	MyDAO1IF dao1 = (MyDAO1IF)dac.getDAO("app", "key1", "con");
	dao1.access(...);

	// dao2 の取得とデータアクセス
	MyDAO2IF dao2 = (MyDAO2IF)dac.getDAO("app", "key2", "con");
	dao2.access(...);

	dac.commit();	// DataAccessController のコミット
} catch (Exception e) {

	try {
		dac.rollback();
	} catch (Exception ex) {
	}

} finally {

	// リソースの解放
	dac.release();

}
```

「リスト 5-17 トランザクション処理（複数 DAO ＋ 複数 JDBCConnector）」のコードを見ると「リスト 5-16 トランザクション処理（複数 DAO ＋ JDBCConnector）」と一致していることがわかる。これは DAO と DataConnector の関連付けはプロパティ設定によって行われており、コード中に明示的に指定しないためである。つまり、2 つ以上の異なる DAO で同じコネクションを使用するのか、それぞれ別のコネクションを使用するのかはソースコード中ではなくプロパティ設定で外部から設定可能であることを示している。

複数の JDBCConnector を使用するとき、commit する場合注意が必要である。JDBCConnector は 2 フェーズコミットに対応していないため、それぞれのコネクションに対してコミットを行う。このとき、最初のコネクションに対して commit が成功したが次のコネクションの commit に失敗した場合、最初のコネクションに対する commit は元に戻すことができない。

5.5.2.4 複数 DAO + 複数コネクション（DataSource）

複数の DAO が複数の DataSourceConnector を利用してデータベースにアクセスする場合の処理内容を「図 5-49 複数 DAO、複数コネクション（DataSource）」に示す。

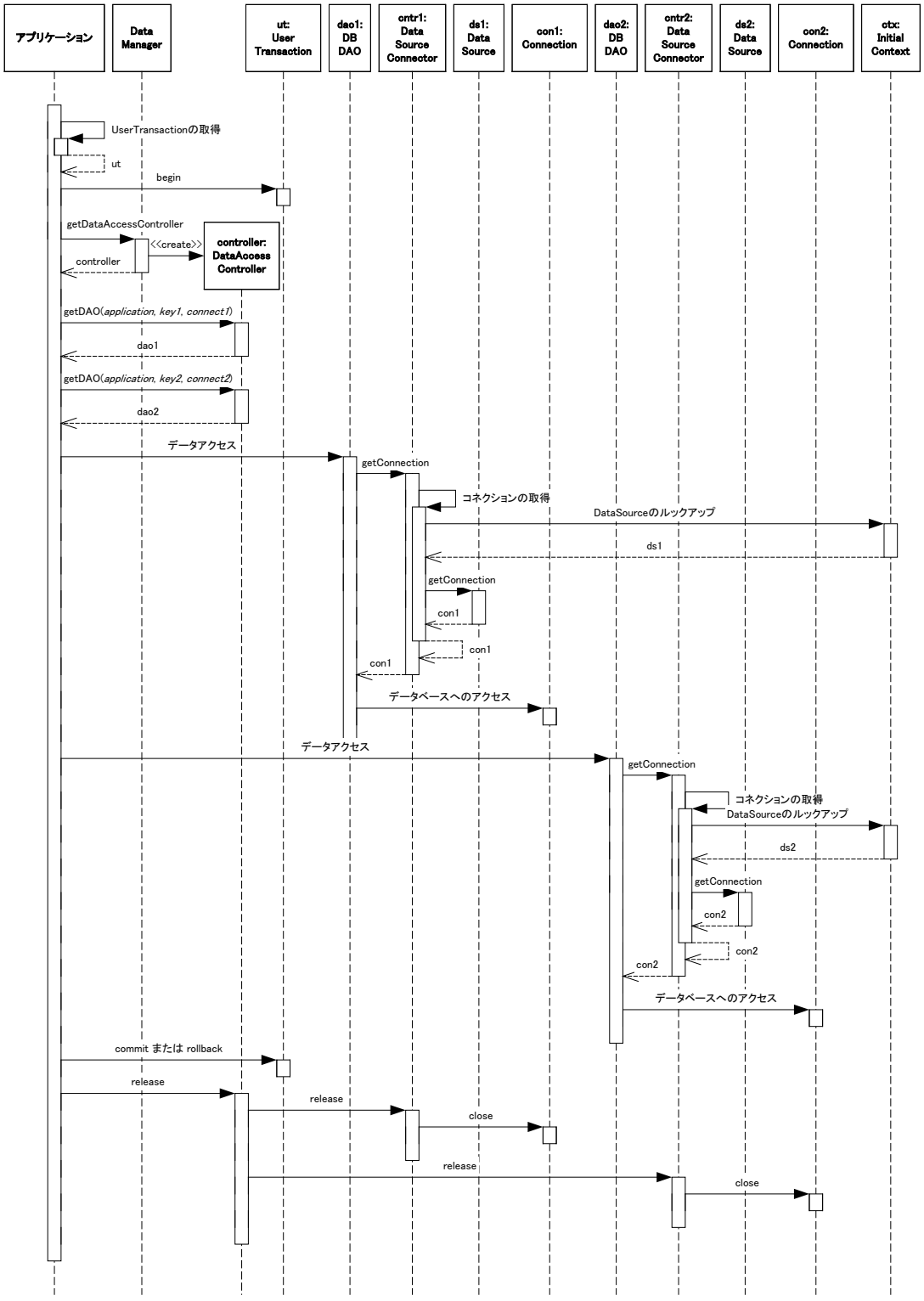


図 5-49 複数 DAO、複数コネクション（DataSource）

「図 5-49 複数 DAO、複数コネクション（DataSource）」に示す内容に該当するサンプルコードは「リスト 5-18 トランザクション処理（DataSourceConnector のみ使用した場合）」のようになる。「リスト 5-18 トランザクション処理（DataSourceConnector のみ使用した場合）」ではコードを簡略化するために、いくつかの例外処理については省略している。

リスト 5-18 トランザクション処理（DataSourceConnector のみ使用した場合）

```
// ユーザトランザクションの取得と開始
InitialContext ctx = new InitiralContext();
UserTransaction ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
ut.begin();

// DataAccessController の取得
DataManager dm = DataManager.getDataManager();
DataAccessController dac = dm.getDataAccessController();

// データアクセス処理
try {

    // dao1 の取得とデータアクセス
    MyDAOIF1 dao1 = (MyDAOIF1)dac.getDAO("app1", "key1", "con1");
    dao1.access(...);

    // dao2 の取得とデータアクセス
    MyDAOIF2 dao2 = (MyDAOIF2)dac.getDAO("app2", "key2", "con2");
    dao2.access(...);

    // コミット
    ut.commit();

} catch (Exception e) {

    // 例外発生時にはロールバック
    ut.rollback();

} finally {

    // リソースの解放
    dac.release();

}
```

「

リスト 5-18 トランザクション処理（DataSourceConnector のみ使用した場合）」のコードを見ると「リスト 5-16 トランザクション処理（複数 DAO ＋ JDBCConnector）」や「リスト 5-17 トランザクション処理（複数 DAO ＋ 複数 JDBCConnector）」とほぼ一致していることがわかる。異なるのはトランザクションの開始と終了である。トランザクションは UserTransaction を取得して begin メソッドを実行しているところから明示的に開始され、commit メソッドまたは rollback メソッドによって終了される。この間に DAO に関連付けられた DataSourceConnector は InitialContext から DataSource をルックアップする。この DataSource から取得された Connection は UserTransaction の管理下におかれる。ここで扱われるすべての DataSource が 2 フェーズコミットに対応している場合、すべての Connection は完全に 1 つのトランザクションとして扱われる（全部 commit が成功するか全部 commit に失敗して rollback されるかのいずれかとなる）。

5.5.2.5 複数 DAO + 複数コネクション（DataSource と JDBC の併用）

複数の DAO が DataSourceConnector と JDBCConnector を併用してデータベースにアクセスする場合の処理内容を「図 5-50 複数 DAO、複数コネクション（DataSource と JDBC の併用）」に示す。

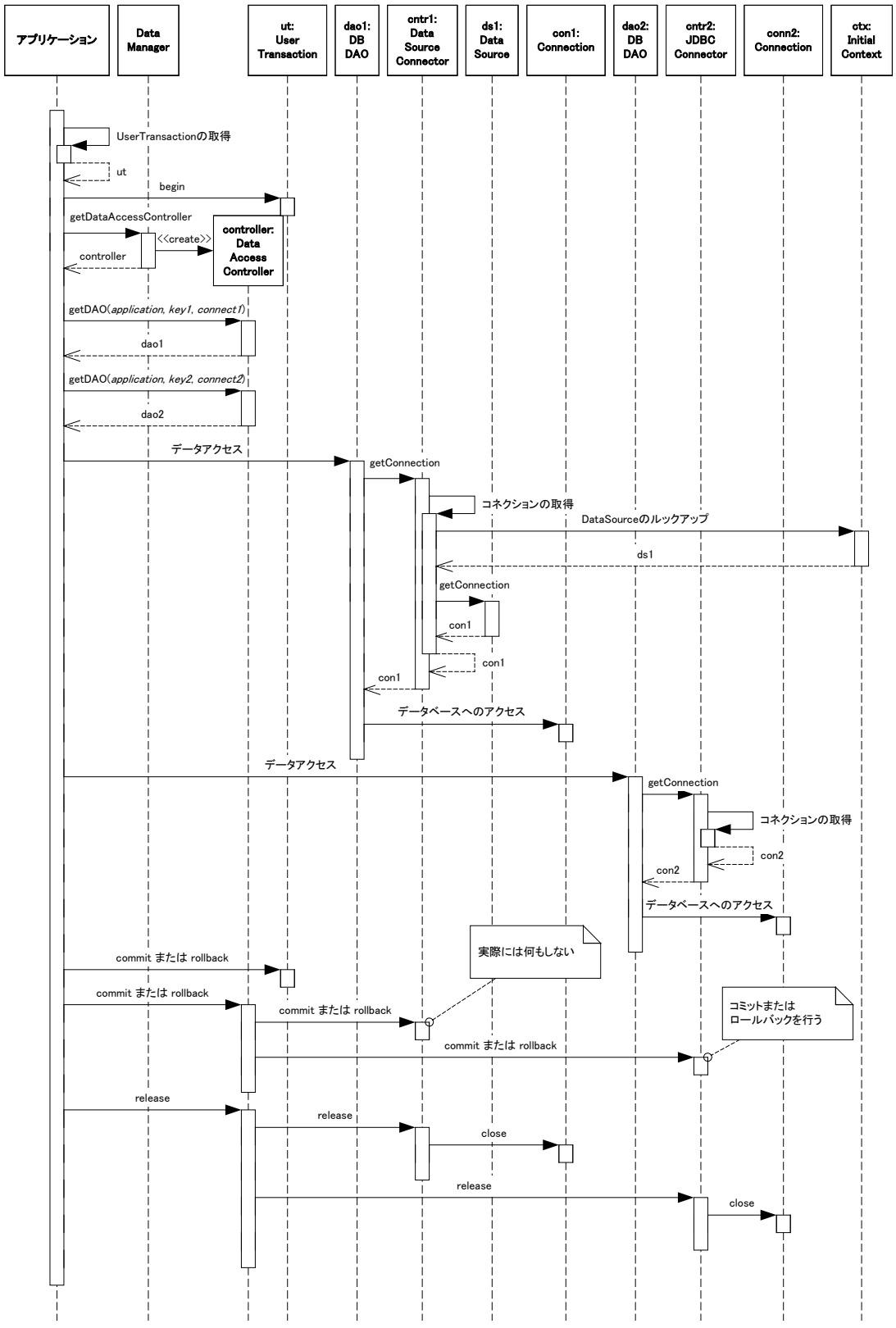


図 5-50 複数 DAO、複数コネクション（DataSource と JDBC の併用）

「図 5-50 複数 DAO、複数コネクション（DataSource と JDBC の併用）」に示す内容に該当するサンプルコードは「リスト 5-19 トランザクション処理（DataSourceConnector と JDBCConnector の併用）」のようになる。「リスト 5-19 トランザクション処理（DataSourceConnector と JDBCConnector の併用）」ではコードを簡略化するために、いくつかの例外処理については省略している。

リスト 5-19 トランザクション処理（DataSourceConnector と JDBCConnector の併用）

```
// ユーザトランザクションの取得と開始
InitialContext ctx = new InitiralContext();
UserTransaction ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
ut.begin();

// DataAccessController の取得
DataManager dm = DataManager.getDataManager();
DataAccessController dac = dm.getDataAccessController();

// データアクセス処理
try {

    try {
        // dao1 の取得とデータアクセス
        MyDAOIF1 dao1 = (MyDAOIF1)dac.getDAO("app1", "key1", "con1");
        dao1.access(...);

        // dao2 の取得とデータアクセス
        MyDAOIF2 dao2 = (MyDAOIF2)dac.getDAO("app2", "key2", "con2");
        dao2.access(...);

        dac.commit();    // DataAccessController のコミット
    } catch (Exception e) {
        try {
            dac.rollback();
        } catch (Exception ex) {
        }
        throw e;
    }
    ut.commit();      // ユーザトランザクションのコミット

} catch (Exception e) {

    // 例外発生時にはロールバック
    ut.rollback();

} finally {

    // リソースの解放
    dac.release();

}
```

「リスト 5-19 トランザクション処理（DataSourceConnector と JDBCConnector の併用）」のコードを見ると「リスト 5-16 トランザクション処理（複数 DAO ＋ JDBCConnector）」、「リスト 5-17 トランザクション処理（複数 DAO ＋ 複数 JDBCConnector）」、さらに「
リスト 5-18 トランザクション処理（DataSourceConnector のみ使用した場合）」とほぼ一致していることがわかる。異なるのはトランザクションの開始と終了である。トランザクションは UserTransaction を取得して begin メソッドを実行しているところから明示的に開始されるとともに、JDBCConnector が関連付けられた DAO が取得されたときにも開始されている。UserTransaction によるトランザクションは commit メソッドまたは rollback メソッドによって終了されており、JDBCConnector のトランザクションは DataAccessController の commit または rollback メソッドによって終了されている。つまり、この場合 2 つの異なるトランザクションが並行して存在していることになる。

2 つの異なるトランザクションが並行して実行されているということは、片方が commit に成功してももう一方が commit に失敗するという事態も考えられる。このような事象を極力避けるためには、UserTransaction で管理される DataConnector のみを使用することが望ましい。

6 メッセージフレームワーク

6.1 概要

国際化された表示をするアプリケーションを作成するためには、それぞれの地域に対応した表示を行う必要がある。メッセージフレームワークはロケールをもとに地域対応した文字列を取得する。

6.2 構成

6.2.1 構成要素

メッセージフレームワークは以下のようなものから構成されている。

- MessageManager
- MessagePropertyHandler

これらの関連を「図 6-1 メッセージフレームワークのクラス図」に示す。

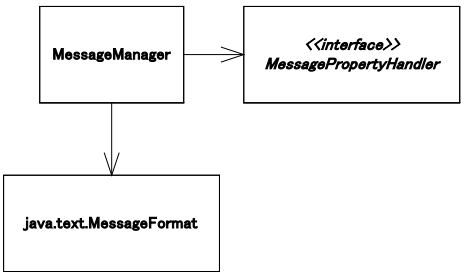


図 6-1 メッセージフレームワークのクラス図

6.2.2 メッセージ取得処理

IM-JavaEE Framework のメッセージフレームワークで地域対応されたメッセージを取得するときの概要を「図 6-2 メッセージ取得の概要」に示す。

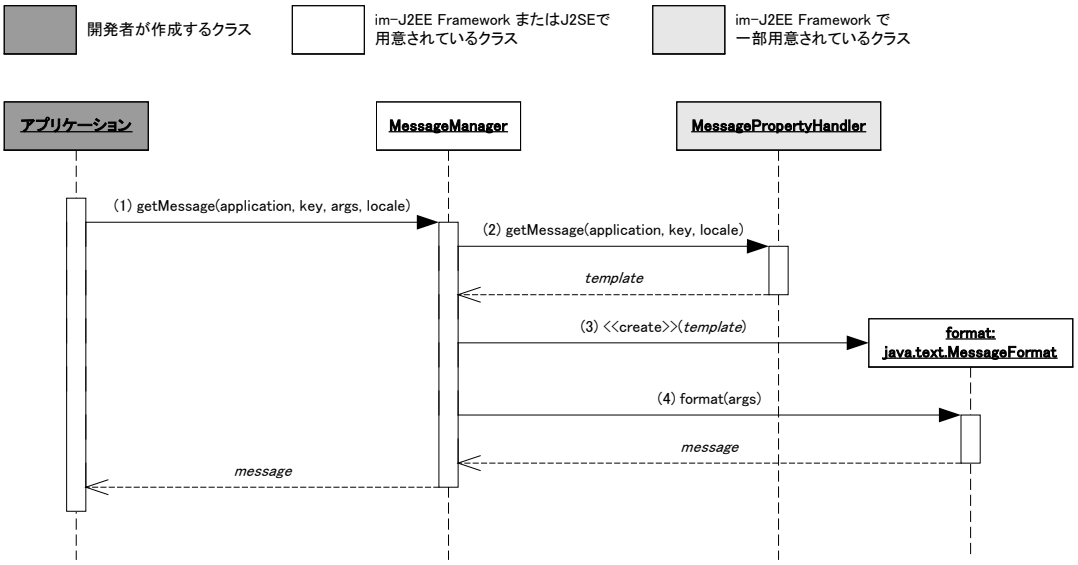


図 6-2 メッセージ取得の概要

1. アプリケーションは MessageManager の getMessage メソッドを使用してアプリケーション ID とメッセージキーに対応するメッセージの取得を依頼する。
2. MessageManager は MessagePropertyHandler を使用してアプリケーション ID とメッセージキーに対応するメッセージの雛形を取得する。
3. MessageManager はメッセージの雛形をもとに java.text.MessageFormat のインスタンスを生成する。
4. MessageManager は生成した MessageFormat のインスタンスの format メソッドを使用し、雛形にパラメータを埋め込んだ形でメッセージを生成する。

6.3 メッセージに関連するプロパティ

IM-JavaEE Framework のメッセージフレームワークではメッセージの雛形を外部で設定することが可能である。メッセージプロパティの取得は jp.co.intra_mart.framework.base.message.MessagePropertyHandler インタフェースを実装したクラスから取得する。IM-JavaEE Framework ではこのインタフェースを実装した複数の実装クラスを標準で提供している(「図 6-3 MessagePropertyHandler」を参照)。メッセージプロパティの設定方法は IM-JavaEE Framework では特に規定してなく、前述のインタフェースを実装したクラスに依存する。

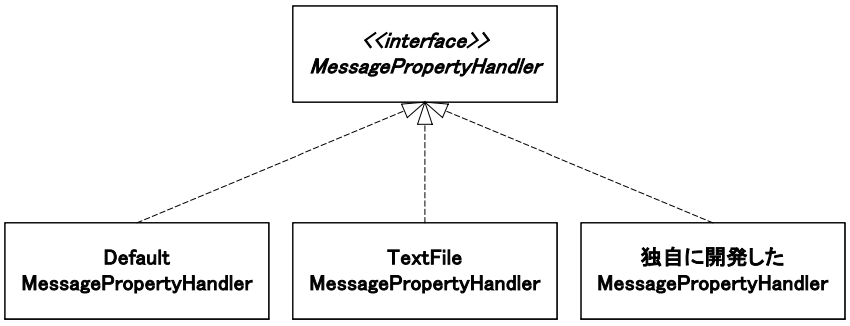


図 6-3 MessagePropertyHandler

ここで取得されるメッセージ(の雛形)は指定された地域に対応したものである。

6.3.1 メッセージに関連するプロパティの取得

メッセージに関連するプロパティは MessagePropertyHandler から取得する。MessagePropertyHandler は jp.co.intra_mart.framework.base.message.MessageManager の getMessagePropertyHandler メソッドで取得すること

ができる。MessagePropertyHandler は必ずこのメソッドを通じて取得されたものである必要があり、開発者が自分でこの MessagePropertyHandler の実装クラスを明示的に生成（new による生成や java.lang.Class の newInstance メソッド、またはリフレクションを利用したインスタンスの生成）をしてはならない。

MessagePropertyHandlerの取得とプロパティの取得に関連する手順を「図 4-22 EventPropertyHandlerの取得」に示す。

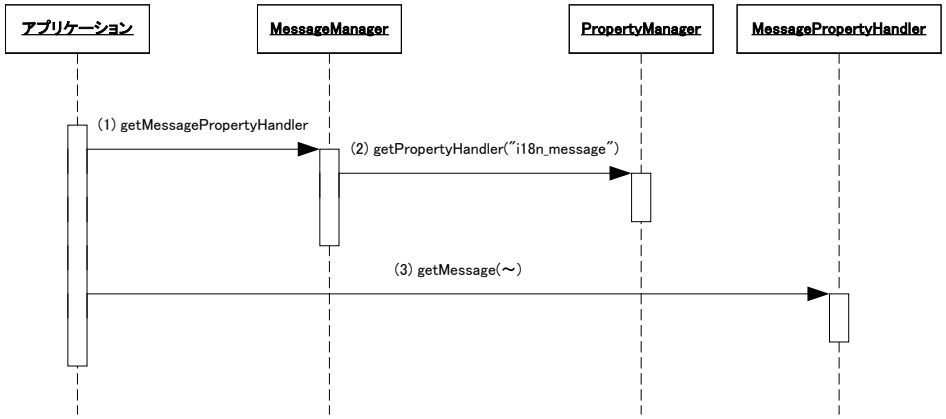


図 6-4 MessagePropertyHandler の取得

1. MessageManager から MessagePropertyHandler を取得する。
2. MessageManager の内部では PropertyManager から MessagePropertyHandler を取得し、アプリケーションに返している。この部分は MessageManager の内部で行っていることであり、開発者は特に意識する必要はない。
3. MessagePropertyHandler を利用して各種のプロパティを取得する。

6.3.2 標準で用意されている MessagePropertyHandler

IM-JavaEE Framework では jp.co.intra_mart.framework.base.message.MessagePropertyHandler を実装したクラスをいくつか提供している。それぞれ設定方法やその特性が違うため、運用者は必要に応じてこれらを切り替えることができる。

6.3.2.1 DefaultMessagePropertyHandler

jp.co.intra_mart.framework.base.message.DefaultMessagePropertyHandler として提供されている。

プロパティの設定はリソースファイルで行う。リソースファイルの内容は「プロパティ名=プロパティの値」という形式で設定する。使用できる文字などは java.util.ResourceBundle に従う。このリソースファイルは使用するアプリケーションから取得できるクラスパスにおく必要がある。リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

リソースファイルからメッセージを取得するとき、国際化が意識されている。詳細については JavaTM 2 SDK, Standard Edition に付随する API リストで java.util.ResourceBundle に関連するドキュメントを参照すること。

6.3.2.2 TextFileMessagePropertyHandler

jp.co.intra_mart.framework.base.message.TextFileMessagePropertyHandler として提供されている。

DefaultMessagePropertyHandlerと同じ形式のリソースファイルを利用する。また、国際化のルールについても同様である。TextFileMessagePropertyHandler と DefaultMessagePropertyHandler で扱うリソースファイルは以下の点が違う。

- クラスパスに通す必要がない。
- アプリケーションから参照できる場所であれば、ファイルシステムの任意の場所に配置できる。
- 設定によってはアプリケーションを停止しないでリソースファイルの再読み込みが可能となる。

リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

6.3.3 独自の MessagePropertyHandler

MessagePropertyHandler を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.base.message.MessagePropertyHandler インタフェースを実装している。
- public なデフォルトコンストラクタ（引数なしのコンストラクタ）が定義されている。
- すべてのメソッドに対して適切な値が返ってくる。（「6.3.4 プロパティの内容」参照）
- isDynamic()メソッドが false を返す場合、プロパティを取得するメソッドはアプリケーションサーバを再起動しない限り値は変わらない。

6.3.4 プロパティの内容

メッセージに関連するプロパティの設定方法は運用時に使用する MessagePropertyHandler の種類によって違うが、概念的には同じものである。

メッセージに関連するプロパティの内容は以下のとおりである。

6.3.4.1 共通

6.3.4.1.1 動的読み込み

isDynamic()メソッドで取得可能。

このメソッドの戻り値が true である場合、このインタフェースで定義される各プロパティ取得メソッド（get～メソッド）は毎回設定情報を読み込みに行くように実装されている必要がある。false である場合、各プロパティ取得メソッドはパフォーマンスを考慮して取得される値を内部でキャッシュしてもよい。

6.3.4.2 アプリケーション個別

6.3.4.2.1 **メッセージ**

getMessage(String application, String key)メソッドまたは getMessage(String application, String key, Locale locale)メソッドで取得可能。

MessagePropertyHandler の getMessage(String application, String key)メソッドは、内部で getMessage(application, key, Locale.getDefault())を呼び出しているのと同等の動きをする。

アプリケーション ID、メッセージキーおよびロケールに対応するメッセージの雛形を設定する。MessagePropertyHandler の getMessage(String application, String key, Locale locale)メソッドが呼ばれた場合、以下の検索順で設定内容を取得し、最初に取得されたものを返す。

1. 指定されたロケールの言語（locale の getLanguage()メソッドで取得されるコード）、国（locale の getCountry()メソッドで取得されるコード）およびバリアント（locale の getVariant()メソッドで取得されるコード）で指定された値
2. 指定されたロケールの言語（locale の getLanguage()メソッドで取得されるコード）および国（locale の getCountry()メソッドで取得されるコード）で指定された値
3. 指定されたロケールの言語（locale の getLanguage()メソッドで取得されるコード）で指定された値
4. ロケールなしで指定された値

上記の順番で検索した値が未設定の場合、MessagePropertyException を throw する。

このプロパティで指定する文字列は java.text.MessageFormat のコンストラクタの引数と同じ形式である必要がある。

7 ログフレームワーク

7.1 概要

開発時または運用時において、ログは内部の情報を知る手がかりの一部となりうる。ログの出力先としてはコンソール、ファイルまたはデータベースなどが上げられる。また、その書式も CSV、XML、データベースのレコードなどさまざまである。これらの事項は開発時に決定できない場合や、さらには運用時に変更したいということも考えられる。

IM-JavaEE Framework のログフレームワークではこれらの問題を解決するため、ログを出力するタイミングと内容、出力先とその書式とわけて管理する方法を提供する。

7.2 構成

7.2.1 構成要素

ログフレームワークは以下のようなものから構成されている。

- LogAgent
- LogManager
- LogPropertyHandler

これらの関連を「図 7-1 ログフレームワークの構成」に示す。

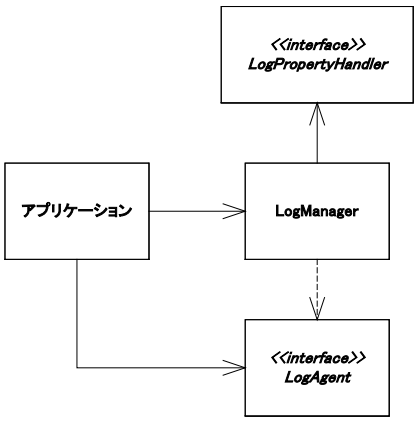


図 7-1 ログフレームワークの構成

7.2.2 ログ出力処理

IM-JavaEE Framework のログフレームワークでログを出力するときの概要を「図 7-2 ログ出力処理の概要」に示す。

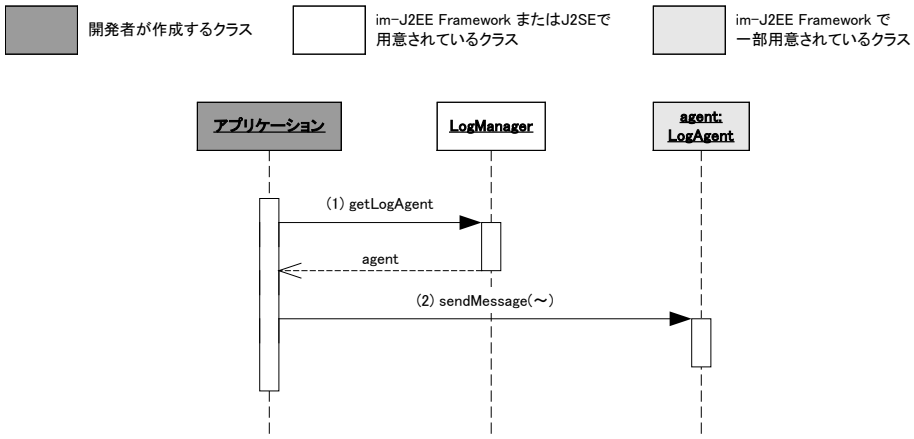


図 7-2 ログ出力処理の概要

1. アプリケーションは LogManager の getLogAgent メソッドを使用して LogAgent のインスタンスを取得する。
2. アプリケーションは取得した LogAgent の sendMessage メソッドを使用してログを出力する。

7.3 LogAgent

jp.co.intra_mart.faramework.system.log.LogAgent インタフェースはログを出力するオブジェクトのインタフェースである。ログの出力先（ファイル、コンソール、データベースなど）やその書式（時系列の羅列、CSV、XML など）はこのインタフェースを実装したクラスに依存する。

7.3.1 LogAgent の機能

LogAgent には次の 2 つのメソッドが用意されている。

- sendMessage(java.lang.String category,
  java.lang.String level,
  java.lang.String message)
- sendMessage(java.lang.String category,
  java.lang.String level,
  java.lang.String message,
  java.lang.Object detail)

引数 category はログを分類するために用いられる。分類の方法はログを扱う開発者に委ねられる。例えば、データベースアクセスや業務ロジックなどのシステム内で実行中の部分に関連するログであったり、購買業務や顧客登録業務などといった業務に関連するログであったりなどである。

引数 level はログの重要度を表す。重要度の区分はログを扱う開発者に委ねられる。主な重要度の区分の例として、デバッグ、情報、警告、例外などがあげられる。jp.co.intra_mart.framework.system.log.LogConstant にはこれらを表現することを意図した定数値がある（それぞれ LEVEL_DEBUG、LEVEL_INFO、LEVEL_WARNNING、LEVEL_ERROR として定義されている）。もちろん、開発者が独自に定義してもよい。

引数 message はログに実際に出力するメッセージを示す。

引数 detail はログに出力するメッセージの詳細を示す。この引数は java.lang.Object であるため、LogAgent の開発者はこの値を適切に扱う必要がある。例えば、java.lang.Throwable が渡されたら printStackTrace メソッドで取得される内容を表示し、その他の場合は toString メソッドで取得される内容を表示する、などである。

7.3.2 LogAgent の準備

LogAgent は LogManager の getLogAgent メソッドを利用して取得する。このとき、LogManager は「図 7-3 LogAgent の初期化」に示す手順で生成された LogAgent を返す。

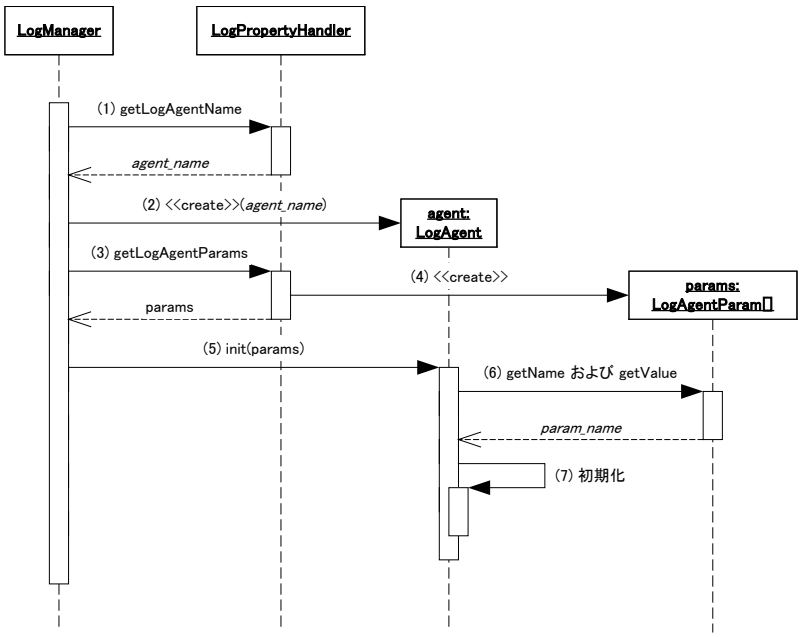


図 7-3 LogAgent の初期化

1. LogManager は LogPropertyHandler の getLogAgentName メソッドを使用して LogAgent のクラス名を取得する。
2. LogManager は取得したクラス名をもとに LogAgent のインスタンスを生成する。
3. LogManager は LogPropertyHandler の getLogAgentParams メソッドを使用して LogAgent の初期化パラメータを取得する。
4. LogPropertyHandler はプロパティ設定内容をもとに LogAgent の初期化パラメータ(LogAgentParam)の配列を返す。
5. LogManager は取得した初期化パラメータをもとに LogAgent の init メソッドを使用して LogAgent を初期化する。
6. LogAgent は LogAgentParam の getName メソッドおよび getValue メソッドを使用して初期化パラメータの値を取得する。
7. LogAgent は取得した初期化パラメータをもとに初期化を行う。

「図 7-3 LogAgent の初期化」のどこかで例外が発生した場合、LogManager の getLogAgent メソッドは LogAgent として DefaultLogAgent(「7.3.3.1 DefaultLogAgent」を参照)を返す。

7.3.3 標準で用意されている LogAgent

IM-JavaEE Framework では基本的な LogAgent がいくつか標準で用意されている。

7.3.3.1 DefaultLogAgent

jp.co.intra_mart.framework.system.log.DefaultLogAgent は標準出力(java.lang.System.out)または標準例外(java.lang.System.err)にログを出力する LogAgent である。出力されるフォーマットは以下に示すとおりとなる。

```
[カテゴリ][レベル]メッセージ
```

ここで、カテゴリ、レベル、メッセージはそれぞれ sendMessage メソッドの第 1 引数、第 2 引数および第 3 引数に対応する。出力先は通常は標準出力であるが、カテゴリが LogConstant.LEVEL_ERROR に等しい場合は標準例外

に出力される。

sendMessage(String, String, String, Object)メソッドも基本的に同じフォーマットで出力される。第 4 引数の値は通常は toString メソッドによって変換された文字列が表示されるが、java.lang.Throwable のサブクラスである場合は出力先にそのスタックトレースが表示される。

7.3.3.2 IntramartLogAgent

IntramartLogAgent は intra-mart で設定されたログへの出力に特化した LogAgent である。

7.3.4 独自の LogAgent

LogAgent を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.system.log.LogAgent インタフェースを実装している。
- public なデフォルトコンストラクタ(引数なしのコンストラクタ)が定義されている。
- init メソッドで適切な初期化が行われること。
- sendMessage メソッドで適切にログが出力されるように実装されていること。

7.3.5 プロパティの内容

ログに関連するプロパティの設定方法は運用時に使用する LogPropertyHandler の種類によって違うが、概念的には同じものである。

ログに関連するプロパティの内容は以下のとおりである。

7.3.5.1 共通

7.3.5.1.1 ログエージェントクラス名

getLogAgentName()メソッドで取得可能。

ログ出力に使用する LogAgent の実装クラスを、パッケージ名を含む完全なクラス名で指定する。何も設定されていない場合、null が返されるように実装されている必要がある。

7.3.5.1.2 ログエージェント初期化パラメータ

getLogAgentParams()メソッドで取得可能。

LogAgent を初期化する時のパラメータを設定する。対応するパラメータがない場合サイズが 0 の配列が返される。

7.3.5.2 アプリケーション個別

ログフレームワークではアプリケーション個別の設定はない。

8 PropertyHandler

8.1 概要

「3 サービスフレームワーク」から「7 ログフレームワーク」まで IM-JavaEE Framework のさまざまなサブフレームワークを紹介してきた。これらの動作はそれぞれのサブフレームワークにおけるプロパティ設定で制御されるようになっている。このプロパティの設定方法は～PropertyHandler という名前のインタフェースを実装したクラスによってそれぞれ異なる。この～PropertyHandler は固定されたものではなく、それぞれのサブフレームワークで使用される～PropertyHandler を実装した任意のクラスと交換することができる。これにより、たとえば頻繁にプロパティ設定を変える必要がある開発時は動的にプロパティを変更できる PropertyHandler を用い、設定を変更することは稀でパフォーマンスが重視される運用時にはプロパティがキャッシュされている PropertyHandler を用いるということが可能となる。

本章では IM-JavaEE Framework における PropertyHandler の設定方法について述べる。

8.2 構成

8.2.1 構成要素

IM-JavaEE Framework におけるプロパティは以下のようなものから構成されている。

- PropertyManager
- PropertyHandler
- ～PropertyHandler

これらの関連を「図 8-1 PropertyHandler の構成」に示す。

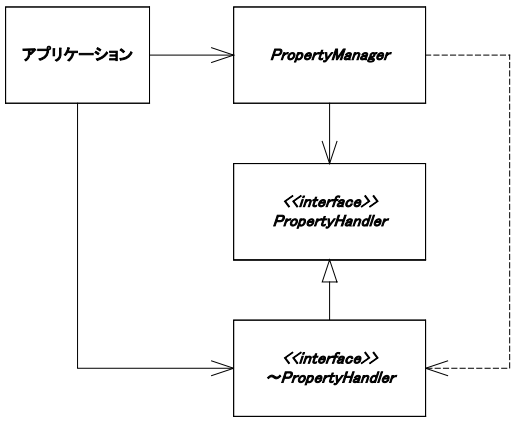


図 8-1 PropertyHandler の構成

8.2.1.1 PropertyManager

jp.co.intra_mart.framework.system.property.PropertyManager は PropertyHandler を生成・取得する役割がある。

8.2.1.2 PropertyHandler

jp.co.intra_mart.framework.system.property.PropertyHandler インタフェースは PropertyManager から生成される PropertyHandler のインタフェースである。IM-JavaEE Framework で使用されるすべての PropertyHandler はこのイ

ンタフェースを実装している必要がある。

8.2.1.3 ～PropertyHandler

これは IM-JavaEE Framework のサービスフレームワークやイベントフレームワークなどそれぞれのサブフレームワークに特化した PropertyHandler のインタフェースである。

8.2.2 PropertyHandler の取得

PropertyHandler を取得するとき、PropertyManager に PropertyHandler の種類を表す「キー」を指定する。PropertyManager はキーに対応する PropertyHandler がキャッシュに存在するか確認し、存在すればその PropertyHandler を返す。PropertyHandler がキャッシュに存在するときの取得の様子を「図 8-2 PropertyHandler の取得（キャッシュに存在する場合）」に示す。

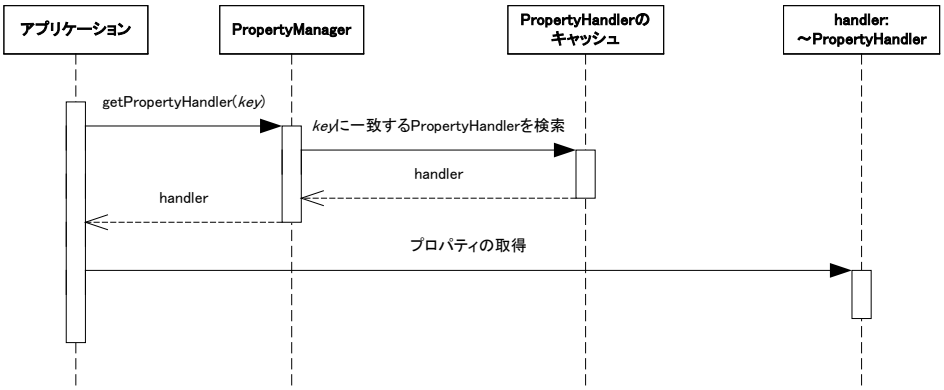


図 8-2 PropertyHandler の取得（キャッシュに存在する場合）

PropertyHandler がキャッシュに存在しない場合、PropertyManager はキーに対応する PropertyHandler を新規に生成する。PropertyHandler に対して初期化する情報があれば PropertyHandler に設定する（PropertyHandler の init メソッド）。最後に PropertyManager は初期化された PropertyHandler をキャッシュに追加して返す。このときの取得の様子を「図 8-3 PropertyHandler の取得（キャッシュに存在しない場合）」に示す。

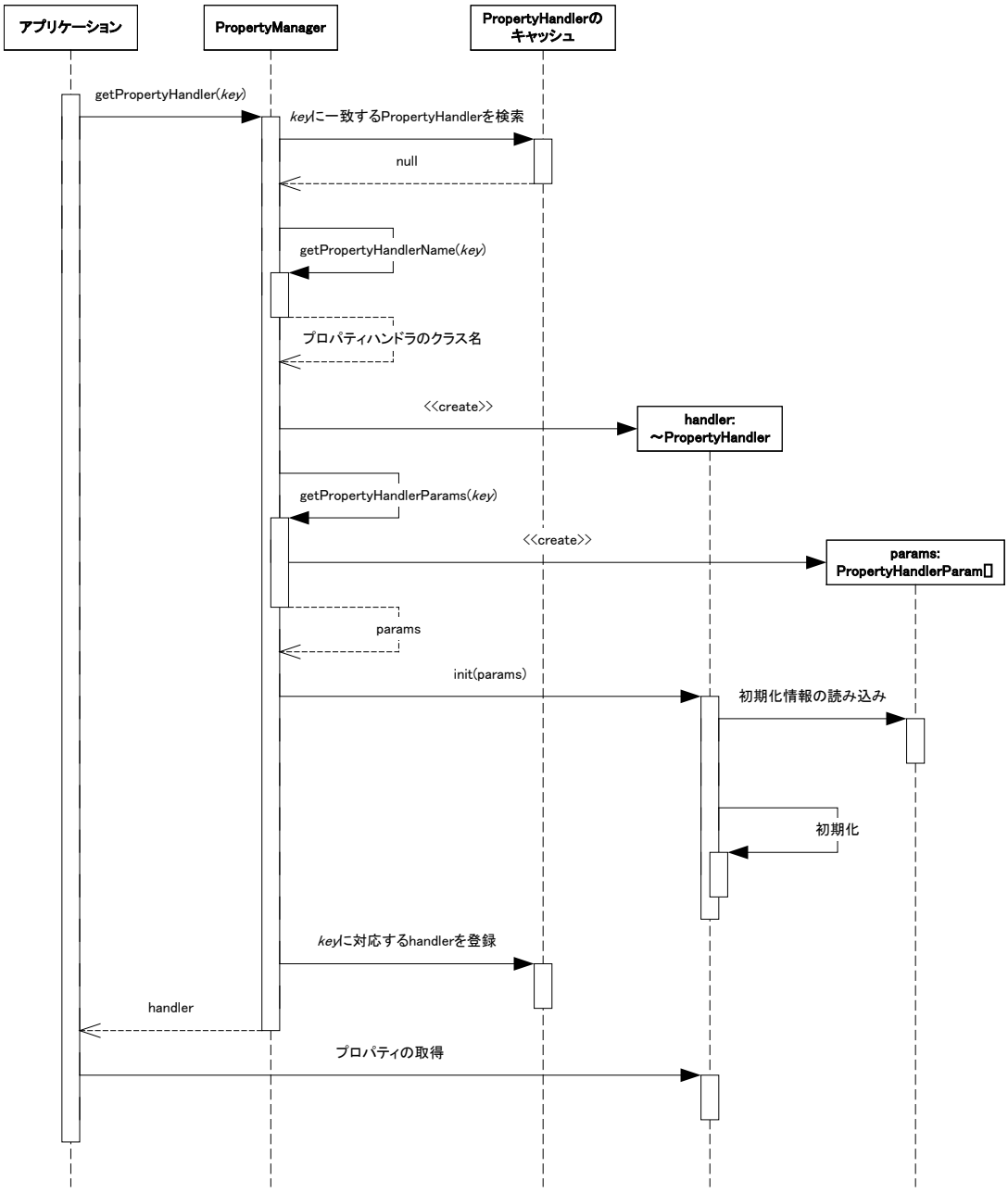


図 8-3 PropertyHandler の取得（キャッシュに存在しない場合）

8.2.3 PropertyManager の取得

jp.co.intra_mart.framework.system.property.PropertyManager はシングルトンパターンの構成をしている。また、PropertyManager 自身は抽象クラスであり、このクラスのコンストラクタは直接使用できない（「図 8-4 PropertyManager の構造」参照）。

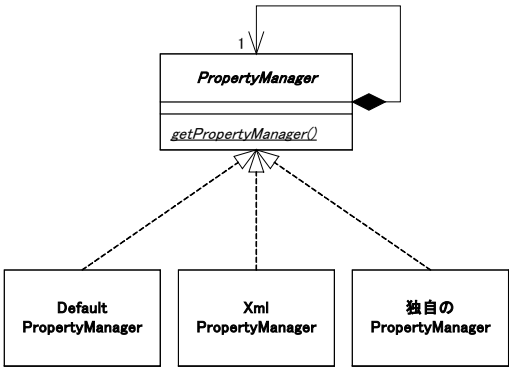


図 8-4 PropertyManager の構造

PropertyManager のインスタンスを取得するためには、PropertyManager の getPropertyManager メソッドを利用する。このメソッドではJRE 起動時のシステムプロパティから実際に使用するPropertyManagerのクラス名を取得し、該当するクラスのインスタンスを生成する。このときのシステムプロパティのキーは PropertyManager.KEY[12]で取得される値である。
このシステムプロパティが設定されていない場合、PropertyManager.DEFAULT_SYSTEM_MANAGER[13]で取得されるクラス名のインスタンスが生成される。

PropertyManager を取得する様子を「図 8-5 PropertyManager の取得（システムプロパティが設定されていない場合）」と「図 8-6 PropertyManager の取得（システムプロパティが設定されている場合）」に示す。

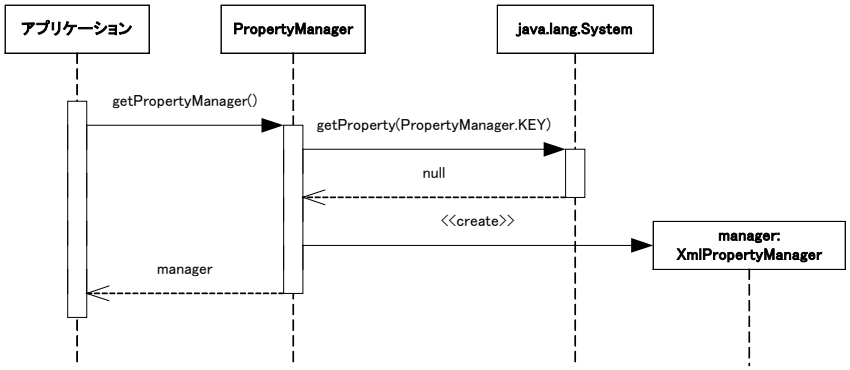
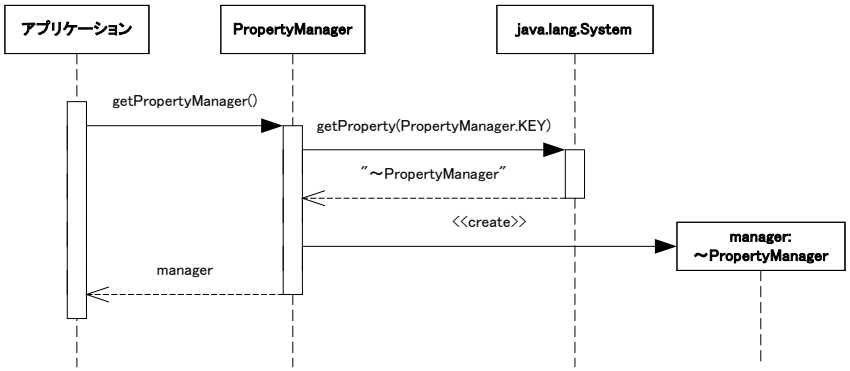


図 8-5 PropertyManager の取得（システムプロパティが設定されていない場合）



12 IM-JavaEE Framework Version 5.0 ではこの値は"jp.co.intra_mart.framework.system.property.PropertyManager"である。

13 IM-JavaEE Framework Version 5.0 ではこの値は"jp.co.intra_mart.framework.system.property.XmlPropertyManager"である。

図 8-6 PropertyManager の取得（システムプロパティが設定されている場合）

8.2.3.1 DefaultPropertyManager

jp.co.intra_mart.framework.system.property.DefaultPropertyManager は IM-JavaEE Framework に標準で用意されている PropertyManager である。

プロパティの設定はリソースファイルで行う。リソースファイルの内容は「プロパティ名=プロパティの値」という形式で設定する。使用できる文字などは java.util.ResourceBundle に従う。このリソースファイルは使用するアプリケーションから取得できるクラスパスにおく必要がある。リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

8.2.3.2 XmlPropertyManager

jp.co.intra_mart.framework.system.property.XmlPropertyManager は IM-JavaEE Framework に標準で用意されている PropertyManager である。
プロパティの設定は XML で行う。この XML ファイルは使用するアプリケーションから取得できるクラスパスにおく必要がある。リソースファイルのファイル名や設定するプロパティ名などの詳細は API リストを参照。

8.2.3.3 独自の PropertyManager

PropertyManager を開発者が独自に作成する場合、以下の要件を満たす必要がある。

- jp.co.intra_mart.framework.system.property.PropertyManager クラスを継承している。
- public なデフォルトコンストラクタ（引数なしのコンストラクタ）が定義されている。
- 以下のメソッドに対して適切な値が返ってくる。
  - getPropertyHandlerName(String key)
    key に対応する PropertyHandler の完全なクラス名を返す。
    このクラスは jp.co.intra_mart.framework.system.property.PropertyHandler インタフェースを実装している必要がある。
  - getPropertyHandlerParams(String key)
    key に対応する PropertyHandler を初期化する再に必要となるパラメータの配列を返す。配列の各要素は jp.co.intra_mart.framework.system.property.PropertyHandlerParam であり、このクラスの getName メソッドでパラメータ名が、getValue メソッドでパラメータの値が取得できるものでなければならない。

9 リソースファイルの移行

9.1 目的

本章は従来のリソースファイルから、Version 5.0 以降で使用可能な XML ファイルへの移行方法について説明する。

9.2 移行可能なリソースファイル

以下のリソースファイルを XML 形式に移行可能である。

- サービスフレームワーク
- イベントフレームワーク
- データフレームワーク

9.3 移行方法

以下のディレクトリに格納されている専用のコンバーターを使用する。

/bin/convert.jar

- 以下のコマンドでヘルプを表示します。

  >java –jar convert.jar

- 使用例

  java -jar convert.jar -app shopping -src c:/imart/doc/imart/WEB-INF/classes -dir c:/xml

10 付録

[1] Java 2 Platform, Standard Edition (J2SE)
http://java.sun.com/j2se/1.4/

[2] Java 2 Platform, Enterprise Edition (J2EE)
http://java.sun.com/j2ee/1.3/docs/

[3] Extensible Markup Language (XML) 1.1
http://www.w3.org/TR/xml11/

[4] JavaTM BluePrints EnterPrise BluePrints
http://java.sun.com/blueprints/enterprise/

[5] JavaTM Servlet Specification Version 2.3
http://java.sun.com/products/servlet/download.html#specs

[6] Enterprise JavaBeansTM Specification, Version 2.0
http://java.sun.com/products/ejb/docs.html

[7] Java Transaction API (JTA) Version 1.0.1
http://java.sun.com/products/jta/index.html

intra-mart Accel Platform
IM-JavaEE Framework 仕様書

2012/10/01 初版

TEL: 03-5549-2821
FAX: 03-5549-2816
E-MAIL: info@intra-mart.jp
URL: http://www.intra-mart.jp/