目次

改訂情報

変更年月日	変更内容
2014-12-01	初版

はじめに

本書の目的

本書は intra-mart Accel Platform に含まれているPluginManager API の説明をします。

説明範囲は以下のとおりです。

- PluginManager API の解説

対象読者

本書は次の利用者を対象としています。

- intra-mart Accel Platform 上でアプリケーションを開発する開発者

次の内容を理解していることが必須となります。

- 一般的な HTML、JavaScript、Java 言語の知識

本書の構成

本書は以下のような構成となっています。

- *概要*

  PluginManager の機能、概要について説明します。

- *プラグインの構成*

  プラグインの構成と設定ファイルについて説明します。

- *PluginDescriptor*

  PluginDescriptor の機能、概要についてと設定例を説明します。

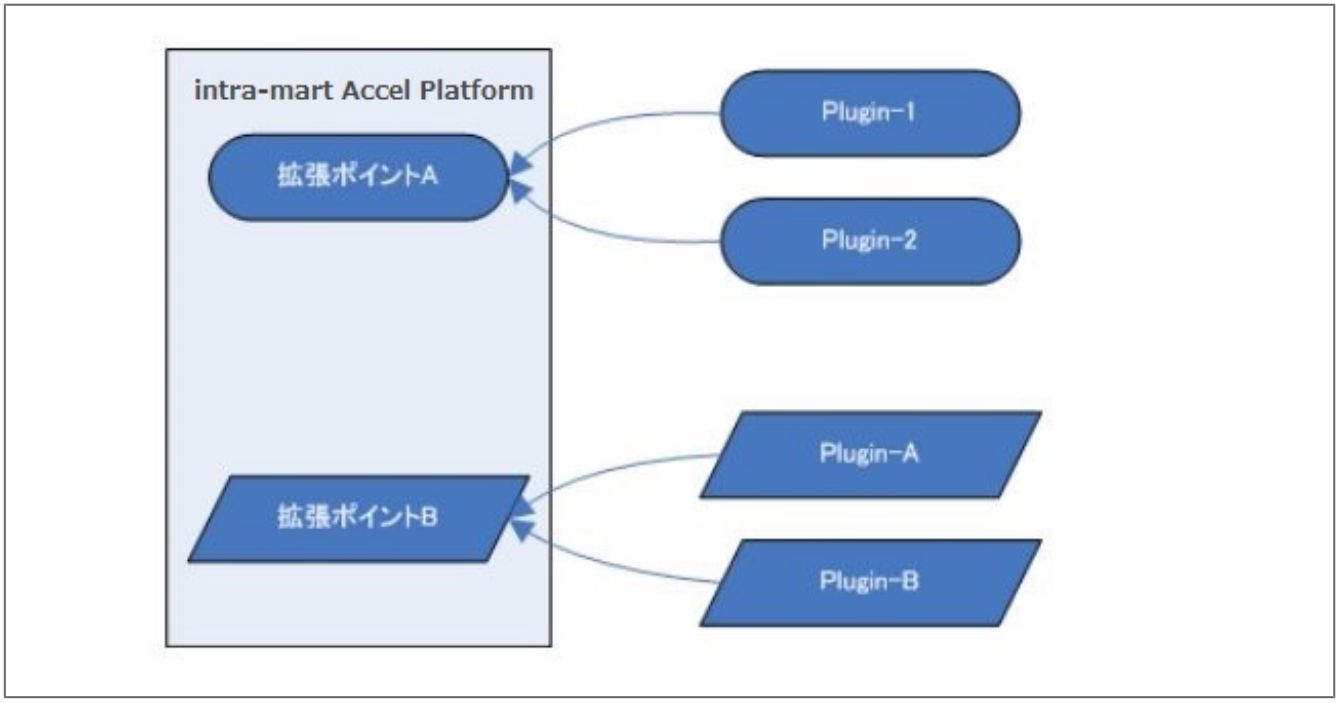
概要

PluginManager では、プラグイン（拡張する設定項目および機能）を拡張ポイント（差込口）に差し込むための仕組みを提供します。
PluginManager を利用することで、各拡張ポイントに差し込まれたプラグインの情報の取得を行うことができます。
アプリケーション開発において独自の拡張ポイントを作成することが可能となります。
PluginManager を利用して各プラグインの読み込み時、提供できる機能は以下のとおりです。

- 各拡張ポイント（差込口）に差し込まれたプラグインの取得
- 各拡張ポイント単位でテナントごとに絞込まれたプラグインの取得
- プラグインのバージョン管理
- プラグインのソート
- プラグインの国際化対応
- プラグインの無効化

拡張ポイントとプラグインのアーキテクチャ

各拡張ポイントに対して作成されたプラグインを読み込む仕組みです。
1つの拡張ポイントに対して、複数のプラグインを設定できます。
各プラグインは PluginManager から拡張ポイントを指定して取得することが可能です。
また、拡張ポイントとプラグイン ID（プラグインの識別 ID）を指定して、取得することも可能です。



プラグインの構成

プラグインを利用するためにユーザモジュールプロジェクトの作成が必要です。
詳しい作成方法については「e Builder での開発の流れ」「モジュール・プロジェクト作成」を参照してください。

プラグインは以下のフォルダにプラグインフォルダ（任意の名前）を作成して、記述します。
プラグインフォルダの名前は、重複しない任意の名前を指定できます。

ユーザモジュールプロジェクト/src/main/plugin

フォルダ作成例 ： ユーザモジュールプロジェクト/src/main/plugin/**jp.co.intra_mart.sample_plugin_1.0**

通常、フォルダの一意性および可視性を保つため、jp.co.intra_mart.plugin.sample_1.0 などのパッケージ名形式 + バージョン番号で記述することをお勧めします。

プラグインフォルダ構成

プラグインフォルダ内のファイル構成は以下の通りです。

```
プラグインフォルダ
│　　重複しない任意の名前を指定できます。（必須）
│
├ plugin.xml
│　　プラグインの設定内容を記述します。記述形式は xml 形式です。（必須）
│
├ plugin.properties
│　　plugin.xml ファイル内で利用する国際化情報です。（オプション）
│　　デフォルトの国際化情報です。
│　　国際化が必要でない場合は、作成する必要はありません。
│　　Javaで用いられるリソースバンドルを用いたプロパティファイル形式です。
│
├ plugin_xx.properties
　　plugin.xmlファイル内で利用する国際化情報です。（オプション)
　　xx で指定したロケールに対応した国際化情報です。
　　国際化が必要でない場合は、作成する必要はありません。
　　Javaで用いられるリソースバンドルを用いたプロパティファイル形式です。
```

plugin.xmlファイル定義

pluginタグ設定

pluginタグ内に必要な情報を定義します。

```
<plugin>
    <extension
        point="｛拡張ポイント ID｝" >
        <プラグインタグ
            id="{プラグイン ID}"
            name="{プラグイン名}"
            version="{バージョン}"
            rank="{ランク}"
            target="{ターゲットプラグイン ID}"
            before="{差し込み方向}"
            groups="{絞り込みテナントID}"
            enable="{有効無効 false|true}">

            // ここに任意のタグを作成して、プラグインの情報を記述します。

        </プラグインタグ>
    </extension>
</plugin>
```

extensionタグ設定

タグ名　extension

【設定項目】

必須項目	○
複数設定	○
設定値・設定する内容	拡張ポイントIDを設定します。
単位・型	なし
省略時のデフォルト値	なし
親タグ	plugin

【属性】

属性名	説明	必須	デフォルト値
point	子ノードに記述するプラグインを差し込む差込口のIDを指定します。 通常、ID の一意性を保つため、 jp.co.intra_mart.plugin.sample などのパッケージ名形式で記述することをお勧めします。	○	なし

プラグインタグ 設定

タグ名	{任意の名前}

【設定項目】

必須項目	○
複数設定	○
設定値・設定する内容	任意のタグを作成して、プラグインの情報を設定します。
単位・型	なし
省略時のデフォルト値	なし
親タグ	extension

【属性】

属性名	説明	必須	デフォルト値
ID	プラグインを識別するIDです。 通常、同じ拡張ポイントID内で一意に識別するプラグインIDです。 同じIDが重複した場合は、version 属性が最も大きいもの 1 つが有効として扱われます。	○	なし
name	プラグインの名称です。	○	なし
version	プラグインのバージョンです。数値をドットつなぎで指定します。（最大 4 個まで） 例 1 / 1.3 / 1.3.12 / 1.3.12.40 etc 同一の拡張ポイントに同じプラグインIDが存在した場合、 バージョン属性がもっとも大きいもの 1 つが有効です。 指定しない場合はもっとも小さいものとして扱われます。	×	なし
rank	ランクを指定することで、このランクの小さい順位にプラグインが読み込まれます。 target 属性が指定された場合は、target 属性が有効です。 数値で指定します。指定しない場合は、ファイルを読み込んだ順序で、逐次最後に追加されます。	×	なし

属性名	説明	必須	デフォルト値
target	同一の拡張ポイント内でこのプラグインの読み込み順を操作するために利用します。 このプラグインを差し込むターゲットプラグイン ID を指定します。 ターゲットプラグインの前後にこのプラグインを差し込みます。 差込む方向は before 属性で指定します。	×	なし
before	ターゲットプラグインの手前に差し込む時：true ターゲットプラグインの後ろに差し込む時：false ターゲットプラグイン ID が指定された場合に有効な属性です。	×	false
enable	このプラグインを無効にする場合は、false を指定します。	×	true
groups	このプラグインが有効なテナントを指定します。複数指定する場合は、カンマ区切りで指定します。 例 tenant1, tenant2 指定しなかった場合、すべてのテナントで有効なプラグインです。	×	なし

プラグインの有効性

同一拡張ポイントに差し込まれたプラグインの enable 属性が false である場合、PluginManagerから取得されるプラグインの対象から外れます。
同一のプラグインIDのプラグインが存在した場合、version属性の値が最も大きいものが有効ですが、有効となったプラグインの enable 属性が false である場合は、同様に取得対象から外れます。

コラム

すでに差し込まれているプラグイン ID と同じプラグインを新しく作成した plugin.xml に作成します。
すでに差し込まれているプラグイン ID の version 属性より大きい値を設定し、enable 属性を false にすることで、元のプラグイン情報を修正することなく、プラグインを無効化することが可能です。

プラグインのソート

同一の拡張ポイントに差し込まれたプラグイン情報の取得順序を制御することが可能です。
プラグインのソートを制御するための属性は、rank, target, before です。

プラグインをソートするルールは以下の通りです。

1. 各プラグインを rank 属性で昇順に並べます。同一の rank が存在した場合は、同一の rank 内で読み込まれた順番に並びます。rank が指定されていないプラグインは、順次最後に追加されます。

2. 1 で並んだプラグインを上から順に target 属性を確認します。target が指定されていた場合は、before 属性に従って、target に指定されたプラグイン ID の前後に移動します。指定された target のプラグイン ID が存在しない場合は、移動は行いません。

プラグインの国際化

plugin.xml 内の情報を国際化することが可能です。
国際化を行うためには、plugin.xml と同一のフォルダに国際化情報(plugin.properties または plugin_xx.properties)を作成する必要があります。
plugin.properties ファイルは、Java のプロパティバンドルの形式で記述します。
key = value 形式です。

plugin.xml 内の任意のタグの属性および値に **%key** で指定することで、国際化情報の key にマッピングされたvalue が設定されます。

例：
プロパティファイル内容
差し替えには以下のファイルをそれぞれ修正します。

```
plugin1.name = テストプラグイン
```

plugin.xml

```
<plugin>
    <extension
        point="jp.co.intra_mart.sample" >
        <test_plugin
            id="plugin1"
            name="%plugin1.name"
        >
            // ここに任意のタグを作成して、プラグインの情報を記述します。
        </test_plugin>
    </extension>
</plugin>
```

PluginManager から取得するときの plugin.xml のイメージ（国際化対応された状態）

```
<plugin>
    <extension
        point="jp.co.intra_mart.sample" >
        <test_plugin
            id="plugin1"
            name="テストプラグイン"
        >
            // ここに任意のタグを作成して、プラグインの情報を記述します。
        </test_plugin>
    </extension>
</plugin>
```

各ロケール国際化されたプラグインの取得

各ロケール国際化されたプラグイン情報を取得する方法は以下の通りです。
まず、plugin.xml と同一のフォルダに plugin_ja.properties ファイルを作成します。

ロケール ja に国際化されたプラグイン情報を取得するには、以下のように実装します。

```
PluginManger mgr = new PluginManager(Locale.Japanese);
Collection<PluginDescriptor> plugins = mgr.getPluginDescriptors("jp.co.intra_mart.sample");
```

PluginManager のコンストラクタでロケール情報を省略した場合は、ログインしているユーザのロケールが自動的に設定されます。

テナント単位で絞り込まれたプラグインの取得

テナント単位で絞り込まれたプラグインを取得することが可能です。

例：
テナント(tenant1)でだけ有効なプラグインを取得する方法は以下の通りです。
まず、テナントで絞り込む設定を plugin.xml に記述します。
プラグインの groups 属性に有効となるテナントID を記述します。
plugin.xml

```
  <plugin>
      <extension
          point="jp.co.intra_mart.sample" >
          <test_plugin
              id="plugin1"
              name="Sample Plugin"
              groups="tenant1"
          >
              // ここに任意のタグを作成して、プラグインの情報を記述します。
          </test_plugin>
      </extension>
  </plugin>
```

テナント（tenant1）でだけ有効なプラグインを取得するには、以下のように実装します。

```
PluginManger mgr = new PluginManager("tenant1");
Collection<PluginDescriptor> plugins = mgr.getPluginDescriptors("jp.co.intra_mart.sample");
```

PluginManager のコンストラクタでテナントを省略した場合は、ログインしているユーザのテナント ID が自動的に設定されます。
また、プラグインの groups 属性を省略した場合、すべてのテナントで有効です。

PluginDescriptor

PluginDescriptor は、PluginManager から取得されるプラグインの定義情報を保持しています。
PluginDescriptor では以下の機能が提供されます。

- プラグインの各属性情報
- プラグイン内の子ノード情報（DOM 形式）
- プラグイン内のタグ属性および値からのインスタンスの生成

プラグイン情報からのインスタンスの生成

プラグイン内のタグ属性および値からクラスまたは サーバサイドJavaScript のインスタンスを生成することができます。

スクリプト開発モデル

plugin.xml を以下のように記述します。
通常、プラグインタグ（この説明では<test_plugin>）の子ノード（ハイライトされた部分）は拡張ポイントによって決定し定義します。定義内容は、xml 形式であれば、その他の制限はありません。

```
<plugin>
    <extension
        point="jp.co.intra_mart.sample.script">
        <test_plugin
            id="plugin1"
            name="sample">
            <test object="test/sample" data="hello">
                <init-param param-name="hoge" param-value ="value"/>
                <init-param param-name="setting" param-value ="1"/>
            </test >
        </test_plugin>
    </extension>
</plugin>
```

インスタンス化するjsファイルを作成します。
スクリプト開発モデル ソースコード配置ディレクトリに配置してください。

```
function getHello() {
    return "Hello World";
}
```

スクリプト開発モデル のファンクションコンテナファイル（js）で、プラグインを取得して、js ファイル（test/sample.js）をインスタンス化し、そのインスタンスのメソッド（**getHello**）を実行します。

```
// PluginManager の生成
var mgr = new PluginManager();

// 拡張ポイント(jp.co.intra_mart.sample.script)の plugin を取得します。
// 以下の処理では、1 つのプラグインが取得されます。(上記の plugin.xml のプラグイン<test_plugin>)
var plugin = mgr.getPluginDescriptor("jp.co.intra_mart.sample.script");

// 取得したプラグインの<test_plugin>の子ノード<test>の object 属性に記述されている
// javascript のファイルをインスタンス化します。
// 取得する場所の指定は、"test/object" のように<test_plugin>(ルート)からのパス形式で指定します。
// このパス(ここでは属性値)に設定されている値"test/sample"すなわち test/sample.js をインスタンス化します。
var obj = plugin.createInstance("test/object");

// インスタンス化したオブジェクトのメソッドを実行します。
var hello = obj.getHello();
```

JavaScript ファイルがインスタンス化される時に、plugin.xml の設定情報を取得することができます。
JavaScript ファイルに initParam および initXml メソッドを追記することで、インスタンス化された後でメソッドが自動的に呼び出されます。

```
function getHello() {
return "Hello World";
}

// このファイルがインスタンス化された時に指定されたタグ（ここでは<test>となります）の子ノードに記述されている
// <init-param>タグの情報がオブジェクト形式で引数に渡されます。
// <init-param>の param-name 属性の値がプロパティ名、param-value 属性の値がそのプロパティの値となります。
function initParam(args) {
    var hoge = args.hoge; // "value"が取得できます。
    var setting = args.setting; // "1"が取得できます。
}

// このファイルがインスタンス化された時に指定されたタグ（ここでは<test>となります）を
// ルートとした DOMNode が引数に渡されます。
function initXml(dom) {
    var data = dom.getArribute("data"); // "hello"が取得できます。
}
```

注意

取得したDOMNodeは、スレッドセーフではありません。
その為、複数のスレッドから同じDOMNodeに対しアクセスするようなコードを記述されている場合正常に動作しなくなる可能性があります。
DOMNodeを利用する際には、初期化処理を用意する等、スレッドセーフとなるようご注意下さい。

plugin.xml を以下のように記述します。
通常、プラグインタグ（この説明では<test_plugin>）の子ノード（ハイライトされた部分）は拡張ポイントによって決定し定義します。定義内容は、xml 形式であれば、その他の制限はありません。

```
<plugin>
    <extension
        point="jp.co.intra_mart.sample.java" >
        <test_plugin
            id="plugin1"
            name="sample">
            <test class="test.Sample" data="hello">
                <init-param param-name="hoge" param-value ="value"/>
                <init-param param-name="setting" param-value ="1"/>
            </test >
        </test_plugin>
    </extension>
</plugin>
```

インスタンス化するクラスを作成します。
作成したクラスはクラスパス上に配置してください。
なお、必ずデフォルトコンストラクタが必要です。

```
package test;
public class Sample {
    // デフォルトコンストラクタ
    public Sample() {}

    public String getHello() {
      return "Hello World";
    }
}
```

JavaEE開発モデル のあるクラスで、プラグインを取得し、クラス（test.Sample）をインスタンス化し、そのインスタンスのメソッド（**getHello**）を実行します。

```
// PluginManager の生成
PluginManager mgr = new PluginManager();

// 拡張ポイント（jp.co.intra_mart.sample.java）の plugin を取得します。
// 以下の処理では、1 つのプラグインが取得されます。（上記の plugin.xml のプラグイン
<test_plugin>）
PluginDescriptor plugin = mgr.getPluginDescriptor("jp.co.intra_mart.sample.java");

// 取得したプラグインの<test_plugin>の子ノード<test>の class 属性に記述されている
// クラスをインスタンス化します。
// 取得する場所の指定は、"test/class"のように<test_plugin>（ルート）からのパス形式で指定します。
// このパス（ここでは属性値）に設定されている値"test.Sample"すなわち test.Sample クラスをインス
タンス化します。
test.Sample obj = (test.Sample)plugin.createInstance("test/class");

// インスタンス化したクラスのメソッドを実行します。
String hello = obj.getHello();
```

クラスがインスタンス化される時に、plugin.xml の設定情報を取得することができます。
インスタンス化されるクラスに以下のインタフェースを実装することで、インスタンス化された後、メソッドが自動的に呼び出されます。

- jp.co.intra_mart.foundation.security.PropertyInitParamable インタフェース
- jp.co.intra_mart.foundation.security.XmlInitParamable インタフェース

必要なインタフェースどちらかまたは両方を指定できます。

```
package test;
public class Sample implements PropertyInitParamable,XmlInitParamable {
    // デフォルトコンストラクタ
    public Sample() {}

    public String getHello() {
        return "Hello World";
    }

    // PropertyInitParamable インタフェースのメソッド
    // このクラスがインスタンス化された時、このメソッドが自動的に呼び出されます。
    // このクラスがインスタンス化された時に指定されたタグ（ここでは<test>となります）の子ノードに記述されている
    // <init-param>タグの情報がMap<String,String>形式で引数に渡されます。
    // <init-param>のparam-name属性の値がMapのkey,param-value属性の値がそのkeyの値となります。
    public void init(Map<String,String> args) {
        String hoge = args.get("hoge"); // "value"が取得できます。
        String setting = args.get("setting"); // "1"が取得できます。
    }

    // XmlInitParamable インタフェースのメソッド
    // このクラスがインスタンス化された時、このメソッドが自動的に呼び出されます。
    // このファイルがインスタンス化された時に指定されたタグ（ここでは<test>となります）を
    // ルートとしたNode(DOM)が引数に渡されます。
    public void init(Node node) {
        String data = new XmlNode(node).getString("data"); // "hello"が取得できます。
    }
}
```

注意

#getNodeで取得したNodeは、スレッドセーフではありません。
その為、複数のスレッドから同じNodeに対しアクセスするようなコードを記述されている場合正常に動作しなくなる可能性があります。
#getNodeを利用する際には、初期化処理を用意する等、スレッドセーフとなるようご注意下さい。