Nimbusコア仕様書

2016年9月26日

目次

[1. 概要 1](#_Toc463263620)

[2. インタフェース設計 5](#_Toc463263621)

[2.1. Service 5](#_Toc463263622)

[2.2. FactoryService 6](#_Toc463263623)

[2.3. ServiceBaseSupport 7](#_Toc463263624)

[2.4. ServiceLoader 8](#_Toc463263625)

[2.5. ServiceManager 9](#_Toc463263626)

[2.6. RegistrationListener 11](#_Toc463263627)

[2.7. RegistrationBroadcaster 11](#_Toc463263628)

[2.8. ServiceStateListener 12](#_Toc463263629)

[2.9. ServiceStateBroadcaster 12](#_Toc463263630)

[3. 実装設計 13](#_Toc463263631)

[3.1. Service 13](#_Toc463263632)

[3.1.1. jp.ossc.nimbus.core.ServiceBase 13](#_Toc463263633)

[3.2. FactoryService 16](#_Toc463263634)

[3.2.1. jp.ossc.nimbus.core.FactoryServiceBase 16](#_Toc463263635)

[3.3. ServiceLoader 20](#_Toc463263636)

[3.3.1. jp.ossc.nimbus.core.DefaultServiceLoaderService 20](#_Toc463263637)

[3.4. ServiceManager 38](#_Toc463263638)

[3.4.1. jp.ossc.nimbus.core.DefaultServiceManagerService 38](#_Toc463263639)

[3.5. ServiceManagerFactory 40](#_Toc463263640)

# 概要

Nimbusは、システム開発において必要な様々な機能を、最小化してインタフェースとして定義して、それらの実装を提供するライブラリ群である。また、小さな機能単位である各インタフェースの実装を組み合わせる事で、もう一段大きな機能を表現する事が可能になる。このような考え方を、一般的にSOA（**S**ervice **O**riented **A**rchitecture：サービス指向アーキテクチャ）という。

機能 a

機能 A

機能 b

Nimbusでは、この機能のインタフェース化において、１つの機能に対して、３つのインタフェースを要求している。１つ目は、その機能を使うアプリケーションに対するインタフェース。２つ目は、その機能の実装に対して振る舞いを設定する管理者向けインタフェース。３つ目は、この機能をホスティングするNimbusのCore基盤に対するインタフェース。各機能の実装クラスは、この３つのインタフェースを実装する必要がある。

機能Aの実装クラス

Nimbus Core向け

インタフェース

アプリケーション向け

インタフェース

管理者向け

インタフェース

アプリケーション向けインタフェースには、特定の制約はなく、機能を表す任意のインタフェースである。

管理者向けインタフェースは、アプリケーションをモニタ及び管理するためのJavaの規格JMX（**J**ava **M**anagement **E**xtensions）に従い、MBeanインタフェース規約に則って定義する。JMXのMBean規格には、いくつか種類が存在するが、Nimbusでは、基本的にStandard MBeanとして実装する。

Nimbus Core向けインタフェースは、Nimbus Coreからの要求に応えるために、ライフサイクルを持ったサービスというコンポーネントにするServiceインタフェースとして提供される。Nimbus において、機能を実装するクラスは、このServiceインタフェースを実装する必要があるため、サービスと呼ぶ。

**サービス**

機能Aの実装クラス

Service

インタフェース

アプリケーション向け

インタフェース

MBean

インタフェース

Serviceインタフェースには、サービスのライフサイクルを制御するメソッドが宣言されており、サービスの実装クラスは、各ライフサイクルにあった処理を実装する必要がある。ライフサイクルは、生成、開始、停止、破棄の４つで、それぞれcreate()、start()、stop()、destroy()の４つのメソッドで宣言される。ライフサイクルの制御順序も、その順序となる。

生成では、インスタンス変数の初期化などを行う。開始では、このサービスの機能である、アプリケーション向けインタフェースが呼びされても良い状態にするための処理を行う。停止では、開始で行った処理と逆の処理を行う。破棄では、インスタンス変数の解放など、生成で行った処理と逆の処理を行う。

また、サービスをNimbus Core上で管理するために、サービスには一意な名前を付与する。名前の一意性は、２段階で構成しており、サービスをグルーピングして管理する単位であるServiceManagerの名前と、そのグループ内での一意性を示すサービスの名前である。この２つで、サービスの一意性が担保されるため、ServiceManagerの名前と、サービスの名前の組み合わせで、サービス名と呼ぶ。実装上では、サービス名は、”ServiceManager名#サービスの名前”という形で表現される。

つまり、Nimbusにおけるサービスとは、Nmbus Core上でライフサイクル制御及び管理され、アプリケーションに提供され、JMX経由で管理者からモニタ及び操作可能なJavaで実装されたコンポーネントである。

サービス

JMX

管理者

アプリケーション

サービス

Nimbus Core

サービス定義

Nimbus Coreは、このサービスを、メモリ中で管理し、ライフサイクル制御を行い、アプリケーションに提供する、いわゆるコンテナである。サービスに対して、どのサービス実装クラスを用い、どのような設定を行い、どのような名前を付与し、どのような順序でサービスを開始するかを、利用者に「サービス定義」という形で定義させ、その通りにサービスをメモリ中に展開、管理し、アプリケーションの要求に対して提供する。このような機能を持つコンテナを、一般的にDIコンテナ（**D**ependency **I**njection：依存性注入）と呼ぶ。

Nimbus Coreの内部構造は、JavaVM中に静的に１つ存在するServiceManagerFactory、その中に複数存在し、サービスをグルーピングして管理するServiceManager、さらにその中にServiceが複数存在する階層構造となっている。

Java VM

ServiceManagerFactory

ServiceManager

ServiceManager

Service

Service

ServiceLoader

ServiceLoader

サービス定義

サービス定義

アプリケーションは、サービスを利用する場合、ServiceManagerFactoryのstaticなエントリポイントとなるメソッドからサービス定義を読み込むように指示する事で、ServiceLoaderを経由して、ServiceManagerやServiceをメモリ中に展開し、使用可能な状態にする。次に、ServiceManagerFactoryのstaticなメソッドからServiceManagerの名前を指定して、ServiceManagerを取得し、更にServiceManagerからサービスの名前を指定して、サービスの参照を取得し、アプリケーション向けインタフェースを呼び出す事で、その機能を使用する。

JavaDocは、以下を参照。

<http://nimbus.osdn.jp/reports/apidocs/jp/ossc/nimbus/core/package-summary.html>

サンプルコードは、以下のzipファイル内のcoreを参照。

<https://osdn.jp/projects/nimbus/downloads/64236/nimbus-sample-1.2.3.zip>

# インタフェース設計

## Service

ServiceManagerにサービスとして管理及び制御させるために必要な機能を提供するインタフェースである。ライフサイクルの制御、ライフサイクルの状態取得、親となるServiceManagerの名前、その中でのサービスの名前の設定、取得などの機能を持つ。

**Service**

ライフサイクル制御：create()、start()、stop()、destroy()、getState()…

名前付与：setServiceManagerName()、getServiceManagerName()、

setServiceName()、getServiceName()…

ServiceManager

Nimbusに存在する全てのサービスは、このインタフェースを実装している。

また、利用者が、サービスを開発する場合は、このインタフェースを抽象実装した以下のクラスを継承する事を推奨している。

|  |  |  |
| --- | --- | --- |
| # | 実装クラス名 | 概要 |
| 1 | jp.ossc.nimbus.core.ServiceBase | 抽象実装クラスである。  ライフサイクルの状態管理の実装が、行われている。ライフサイクルの処理実装は、createService()、startService()、stopService()、destroyService()の４つのメソッドが、オーバーライド用に空実装されているので、必要に応じてオーバーライドする。  また、ログやメッセージ管理等のサービスも利用できるように実装されている。 |

## FactoryService

サービスをファクトリ化するためのインタフェースである。

**FactoryService**

newInstance()

getServiceObject(name)

ServiceManager

通常のサービスは、ServiceManagerからサービス名を指定してgetServiceObject(name)でサービスを取得すると、毎回同じインスタンスが取得される。このインタフェースを実装したサービスの場合、ServiceManagerはFactoryServiceのnewInstance()メソッドを呼び出し、その戻り値をgetServiceObject(name)の戻り値として返すため、取得する毎に異なるインスタンスが取得される。

FactoryServiceにsetThreadLocal(true)とする事で、newInstance()メソッドの戻り値が、スレッド単位で異なるインスタンスになる。また、FactoryServiceにsetManagement(true)とする事で、newInstance()メソッドで生成されたオブジェクトをサービスとして、ServiceManagerに登録して管理する事ができる。

以下に、FactoryServiceインタフェースの実装クラスの一覧を示す。

|  |  |  |
| --- | --- | --- |
| # | 実装クラス名 | 概要 |
| 1 | jp.ossc.nimbus.core.FactoryServiceBase | 抽象実装クラスである。  抽象メソッドcreateInstance()が宣言されており、ファクトリとしてサービスのインスタンスを生成する機能のみが抽象化されており、他は全て実装されている。 |

## ServiceBaseSupport

Serviceインタフェースの抽象クラスであるServiceBaseを継承できないクラスに、サービスとしてのライフサイクル実装のみを与えるためのインタフェースである。

通常、独自のサービスを実装する場合、Serviceインタフェースの抽象クラスであるServiceBaseを継承し、ライフサイクル実装（createService()、startService()、stopService()、destroyService()）のみオーバーライドする事で、Serviceインタフェースを実装する事が可能である。しかし、Javaでは多重継承が許されないため、独自のサービスクラスが、ServiceBase以外に継承したい親クラスが存在する場合、ServiceBaseを継承できなくなり、Serviceインタフェースを直接実装する必要に迫られる。そのようなサービスクラスにおいて、ライフサイクル実装だけを実装する事ができるようにするインタフェースである。

任意の親クラス

独自のサービスクラス

**ServiceBaseSupport**

Nimbus利用者が、独自サービスを実装する場合に利用するインタフェースであるため、Nimbus内部に、このインタフェースの実装クラスは存在しない。

## ServiceLoader

サービス定義を読み込み、サービスを配置する機能を提供するインタフェースである。

setServiceURL(url)

start()

サービス定義

ServiceMetaData

ServiceMetaData

**ServiceLoader**

registerService()

addServiceLoader()

start()

ServiceManager

registerManager()

ServiceManagerFactory

ServiceLoaderにサービス定義ファイルのURLを設定し、start()する事でサービス定義を読み込む。ServiceManagerを生成し、サービス定義ファイルをパースして得られたサービスの定義情報ServiceMetaDataを登録する。また、自分自身をServiceManagerに登録する。最後に、ServiceManagerを開始する。ServiceManagerは、開始されると、自分自身をServiceManagerFactoryに登録する。

以下に、ServiceLoaderインタフェースの実装クラスの一覧を示す。

|  |  |  |
| --- | --- | --- |
| # | 実装クラス名 | 概要 |
| 1 | jp.ossc.nimbus.core.DefaultServiceLoaderService | XMLファイルで定義したサービス定義を読み込み、サービスを配置するデフォルト実装クラスである。 |

## ServiceManager

サービスを管理し、サービスのライフサイクル制御を行う機能を提供するインタフェースである。

サービス定義

ServiceLoader

addServiceLoader()

start()

ServiceMetaData

ServiceMetaData

registerService()

**ServiceManager**

create()

start()

registerManager()

ServiceManagerFactory

registerService()

Service

Service

ServiceLoaderによってサービス定義が読み込まれ、ServiceManagerにサービス定義情報であるServiceMetaDataが登録される。ServiceLoaderによってstart()が呼び出されると、登録されたServiceMetaDataを元に、各サービスをインスタンス化、create()、属性の注入、start()を呼び出し、サービスを起動していく。この際、サービスの依存関係を考慮して、サービスの起動順の制御を行う。最後に、ServiceManagerが開始すると、自分自身をServiceManagerFactoryに登録する。

ServiceManagerが開始されると、getServiceObject(name)で、サービスの名前を指定してサービスを取得する事ができる。

以下に、ServiceManagerからサービスを取得する例を示す。

import jp.ossc.nimbus.core.ServiceManagerFactory;

import jp.ossc.nimbus.core.ServiceManager;

// ServiceManagerの名前を指定して、ServiceManagerを取得する

ServiceManager manager1 = ServiceManagerFactory.findManager(“Manager1”);

// サービスの名前を指定して、サービスを取得する

Object service1 = manager1.getServiceObject(“Service1”);

また、親インタフェースとして、後述するRegistrationBroadcasterを持ち、サービスの登録状態を通知する機能も持つ。

以下に、ServiceManagerインタフェースの実装クラスの一覧を示す。

|  |  |  |
| --- | --- | --- |
| # | 実装クラス名 | 概要 |
| 1 | jp.ossc.nimbus.core.DefaultServiceManagerService | デフォルト実装クラスである。 |

## RegistrationListener

登録状態のイベント通知を受け取るリスナーインタフェースである。

addRegistrationListener()

RegistrationBroadcaster

**RegistrationListener**

registered(event)

unregistered(event)

RegistrationEvent

登録状態イベント発生させるRegistrationBroadcasterに、自分自身を登録することで、登録及び登録解除イベントの通知を受け取る。

## RegistrationBroadcaster

登録状態のイベント通知を発行する機能を提供するインタフェースである。

登録されたRegistrationListenerに、登録状態イベントを通知する。

以下に、RegistrationBroadcasterインタフェースの実装クラスの一覧を示す。

|  |  |  |
| --- | --- | --- |
| # | 実装クラス名 | 概要 |
| 1 | jp.ossc.nimbus.core.DefaultServiceManagerService | サービスの登録状態を通知する実装クラスである。 |

## ServiceStateListener

サービスのライフサイクル状態のイベント通知を受け取るリスナーインタフェースである。

addServiceStateListener()

ServiceStateBroadcaster

**ServiceStateListener**

stateChanged(event)

ServiceStateChangeEvent

サービスのライフサイクル状態イベント発生させるServiceStateBroadcasterに、自分自身を登録することで、ライフサイクル状態変更イベントの通知を受け取る。

## ServiceStateBroadcaster

サービスのライフサイクル状態のイベント通知を発行する機能を提供するインタフェースである。

登録されたServiceStateListenerに、ライフサイクル状態変更イベントを通知する。

以下に、ServiceStateBroadcasterインタフェースの実装クラスの一覧を示す。

|  |  |  |
| --- | --- | --- |
| # | 実装クラス名 | 概要 |
| 1 | jp.ossc.nimbus.core.ServiceBase | サービスのライフサイクル状態を通知する実装クラスである。 |

# 実装設計

インタフェース設計で示した各インタフェースのうち、機能性のあるインタフェースの実装クラスの機能を説明する。また、Nimbus Coreの中にあり、インタフェースを実装しないクラスについても説明する。

## Service

### jp.ossc.nimbus.core.ServiceBase

Serviceインタフェースの抽象実装クラスである。

Nimbusのサービスとして実装したいクラスを開発する際に、実装すべきServiceインタフェースの基本実装を実装した抽象クラスとして提供している。

ServiceBaseMBean

Service

**ServiceBase**

アプリケーション向け

インタフェース

管理者向けMBean

インタフェース

サービスのクラス

以下に、このクラスを継承したサービスの実装例を示す。

アプリケーション向けインタフェースMessage.java

public interface Message{

public void displayMessage();

}

管理者向けインタフェースMessageServiceMBean.java

import jp.ossc.nimbus.core.ServiceMBean;

public inetrface MessageServiceMBean extends ServiceMBean{

public void setMessage(String message);

public String getMessage();

}

サービスクラスMessageService.java

import jp.ossc.nimbus.core.ServiceBase;

public class MessageService extends ServiceBase

implements Message, MessageServiceMBean{

private String message;

// JMXサーバ向けのMBeanインタフェースを実装

public void setMessage(String message){

this.message = message;

}

public String getMessage(){

return message;

}

// コンテナ向けのServiceインタフェースを実装

// jp.ossc.nimbus.core.ServiceBaseを継承する事で、

// 実装したいライフサイクルだけを実装する事ができる

public void startService() throws Exception{

if(message == null){

throw new Exception("メッセージが設定されていません");

}

}

// アプリケーション向けのインタフェースを実装

public void displayMessage(){

System.out.println(message);

}

}

以下にサービスの属性一覧を示す。

|  |  |  |  |
| --- | --- | --- | --- |
| # | 属性名 | 型 | 説明 |
| 1 | SystemLoggerServiceName | ServiceName | ログ出力に使用するjp.ossc.nimbus.service.log.Loggerサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つLoggerを使用する。 |
| 2 | SystemMessageRecordFactoryServiceName | ServiceName | メッセージ取得に使用するjp.ossc.nimbus.service.message.MessageRecordFactoryサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つMessageRecordFactoryを使用する。 |

## FactoryService

### jp.ossc.nimbus.core.FactoryServiceBase

FactoryServiceインタフェースの抽象実装クラスである。

Nimbusのファクトリサービスとして実装したいクラスを開発する際に、実装すべきFactoryServiceインタフェースの基本実装を実装した抽象クラスとして提供している。

ServiceBaseMBean

ServiceBase

Service

FactoryServiceBaseMBean

FactoryService

**FactroyServiceBase**

管理者向けMBean

インタフェース

ファクトリサービス

のクラス

以下に、このクラスを継承したファクトリサービスの実装例を示す。

管理者向けインタフェースMessageFactoryServiceMBean.java

import jp.ossc.nimbus.core.FactoryServiceBaseMBean;

public inetrface MessageFactoryServiceMBean extends FactoryServiceBaseMBean{

public void setMessage(String message);

public String getMessage();

}

サービスクラスMessageFactoryService.java

import jp.ossc.nimbus.core.FactoryServiceBase;

public class MessageFactoryService extends FactoryServiceBase

implements MessageFactoryServiceMBean{

private String message;

// JMXサーバ向けのMBeanインタフェースを実装

public void setMessage(String message){

this.message = message;

}

public String getMessage(){

return message;

}

// コンテナ向けのServiceインタフェースを実装

// jp.ossc.nimbus.core.FactoryServiceBaseを継承する事で、

// 実装したいライフサイクルだけを実装する事ができる

public void startService() throws Exception{

if(message == null){

throw new Exception("メッセージが設定されていません");

}

}

// jp.ossc.nimbus.core. FactoryServiceBaseの抽象メソッドを実装

protected Object createInstance() throws Exception{

MessageService service = new MessageService();

service.create();

service.setMessage(message);

service.start();

return service;

}

}

また、ServiceBaseは、ServiceStateBroadcasterインタフェースを実装しており、登録されたServiceStateListenerに対して、サービスの状態変更のイベント通知を行う。

以下にサービスの属性一覧を示す。

|  |  |  |  |
| --- | --- | --- | --- |
| # | 属性名 | 型 | 説明 |
| 1 | SystemLoggerServiceName | ServiceName | ログ出力に使用するjp.ossc.nimbus.service.log.Loggerサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つLoggerを使用する。 |
| 2 | SystemMessageRecordFactoryServiceName | ServiceName | メッセージ取得に使用するjp.ossc.nimbus.service.message.MessageRecordFactoryサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つMessageRecordFactoryを使用する。 |
| 3 | CreateTemplateOnStart | boolean | サービスの開始時に、ファクトリするオブジェクトの生成を試みてみるかどうかを設定する。  デフォルトでは、true。 |
| 4 | Management | boolean | このファクトリが生成するオブジェクトを管理するかどうかを設定する。  trueにした場合、その後、このファクトリによって生成されるオブジェクトは、このファクトリの管理化に置かれる。管理されているオブジェクトは、このファクトリの属性の変更を反映される。（どのような属性が管理されるかは、実装に依存する）そのため、このファクトリが参照を保持するので、使う側が参照を捨てても、ガベージされない。このファクトリが、破棄された場合には、管理されているオブジェクトの参照も破棄される。  falseにした場合、その後、このファクトリによって生成されるオブジェクトは、このファクトリの管理化に置かれない。管理されていないオブジェクトは、このファクトリの属性の変更を反映されない。そのため、このファクトリが参照を保持することはないので、使う側が参照を捨てると、ガベージの対象になる。 |
| 5 | ThreadLocal | boolean | このファクトリが生成するオブジェクトをスレッド単位に生成するかどうかを設定する。  デフォルトはfalse。 |

## ServiceLoader

### jp.ossc.nimbus.core.DefaultServiceLoaderService

XMLファイルで記述したサービス定義ファイルを読み込み、サービスを配置するServiceLoaderインタフェースのデフォルト実装クラスである。

サービス定義XMLファイル

<server>

<manager name=”Manager1”>

<service name=”Service1”

code=”sample.Service”/>

：

</manager>

</server>

setServiceURL(url)

**DefaultServiceLoaderService**

ServerMetaData

start()

ManagerMetaData

ServiceMetaData

ManagerMetaData

ServiceMetaData

start()

registerService()

ServiceManager

ServiceManager

ServiceMetaData

Service

Service

サービス定義XMLファイルのURLをsetServiceURL(url)で設定し、サービスを開始すると、サービス定義XMLファイルを読み込んでパースし、その内容をServerMetaData、ManagerMetaData、ServiceMetaDataなどの定義情報として取り込む。次に、ServiceManager の定義情報であるManagerMetaDataの数だけServiceManagerを生成して、サービスの定義情報であるServiceMetaDataをServiceManagerに登録する。最後にServiceManagerを開始する。

以下に、サービス定義ＸＭＬファイルの仕様を定義する。

子要素の”+”は、１個以上。”\*”は、０個以上。”?”は0個か１個、現れる事を意味する。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | 要素名 | 親要素 | 子要素 | 説明 | |
| 1 | server | なし | ref-url\*  manager-repository?  property-editors?  default-log?  log?  message?  server-property\*  manager\*  ifdef\* | ルート要素。  ServiceManagerFactoryの定義に相当する。 | |
| 2 | ref-url | server | なし | 内容に、このファイルが参照するサービス定義XMLファイルのURLを指定する。  この要素を指定した場合には、このファイル内と参照するファイル内とで、サービスの依存関係のチェックを行い、相互に依存している場合や、依存するサービスが存在しない場合などは、例外をthrowする。  この要素を指定しない場合には、相互に依存するサービスは、起動が保証されない。  また、依存するサービスが存在しない場合は、依存するサービスが起動されるまで、該当サービスが起動されない。  内容に指定したURLは、以下の順序で解釈される。  (1)絶対URL  (2)このファイルからの相対パス  (3)ローカルファイルパス  (4)nimbus.jarをロードしたクラスローダのリソース | |
| 3 | manager-repository | server | なし | server要素で構成されるServiceManagerFactoryに、manager要素で構成されるServiceManagerを格納するRepositoryサービスのサービス名を定義する。  サービス名は、manager-name属性でServiceManagerの名前を、内容でサービスの名前を指定する方法と、内容にServiceManagerの名前#サービスの名前で指定する方法がある。 | |
| 属性名 | | 必須/任意 | 説明 |
| manager-name | | 任意 | Repositoryが登録されているServiceManagerの名前を定義する。 |
| 4 | property-editors | server | property-editor+  ifdef\* | attribute要素の内容の文字列を、属性の型に合わせて変換するjava.beans.PropertyEditorを定義するproperty-editor要素の集合を定義する。 | |
| 5 | property-editor | property-editors | なし | attribute要素の内容の文字列を、属性の型に合わせて変換するjava.beans.PropertyEditorを定義する。  type属性で、attribute要素のtype属性に、属性の型に対応するクラス名を完全修飾クラス名で指定する。  また、内容に、その型の文字列を編集するjava.beans.PropertyEditorの実装クラスを完全修飾クラス名で指定する。 | |
| 属性名 | | 必須/任意 | 説明 |
| type | | 必須 | java.beans.PropertyEditorが編集する属性の型を指定する。 |
| 6 | default-log | server | debug?  information?  warning?  error?  fatal? | ServiceManagerFactoryで使用するデフォルトのLoggerで、出力されるシステムログをログレベル毎に出力の設定を定義する。  この要素を指定しない場合は、DEBUGログ以外は出力される。 | |
| 7 | debug | default-log | なし | ServiceManagerFactoryで使用するデフォルトのLoggerで出力されるDEBUGログを出力するかどうかを定義する。  この要素を指定しない場合は、DEBUGログは出力されない。 | |
| 属性名 | | 必須/任意 | 説明 |
| output | | 必須 | このレベルのログを出力する場合、true |
| 8 | information | default-log | なし | ServiceManagerFactoryで使用するデフォルトのLoggerで出力されるINFOログを出力するかどうかを定義する。  この要素を指定しない場合は、INFOログは出力されない。 | |
| 属性名 | | 必須/任意 | 説明 |
| output | | 必須 | このレベルのログを出力する場合、true |
| 9 | warning | default-log | なし | ServiceManagerFactoryで使用するデフォルトのLoggerで出力されるWARNログを出力するかどうかを定義する。  この要素を指定しない場合は、WARNログは出力されない。 | |
| 属性名 | | 必須/任意 | 説明 |
| output | | 必須 | このレベルのログを出力する場合、true |
| 10 | error | default-log | なし | ServiceManagerFactoryで使用するデフォルトのLoggerで出力されるERRORログを出力するかどうかを定義する。  この要素を指定しない場合は、ERRORログは出力されない。 | |
| 属性名 | | 必須/任意 | 説明 |
| output | | 必須 | このレベルのログを出力する場合、true |
| 11 | fatal | default-log | なし | ServiceManagerFactoryで使用するデフォルトのLoggerで出力されるFATALログを出力するかどうかを定義する。  この要素を指定しない場合は、FATALログは出力されない。 | |
| 属性名 | | 必須/任意 | 説明 |
| output | | 必須 | このレベルのログを出力する場合、true |
| 12 | log | server  manager | なし | server要素で構成されるServiceManagerFactoryや、manager要素で構成されるServiceManagerで使用するLogger　のサービス名を定義する。  サービス名は、manager-name属性でServiceManagerの名前を、内容でサービスの名前を指定する方法と、内容にServiceManagerの名前#サービスの名前で指定する方法がある。  指定しない場合には、デフォルトのLoggerが使用される。 | |
| 属性名 | | 必須/任意 | 説明 |
| manager-name | | 任意 | Loggerが登録されているServiceManagerの名前を定義する。 |
| 13 | message | server  manager | なし | server要素で構成されるServiceManagerFactoryや、manager要素で構成されるServiceManagerで使用するMessageRecordFactoryを定義する。  サービス名は、manager-name属性でServiceManagerの名前を、内容でサービスの名前を指定する方法と、内容にServiceManagerの名前#サービスの名前で指定する方法がある。  指定しない場合には、デフォルトのMessageRecordFactoryが使用される。 | |
| 属性名 | | 必須/任意 | 説明 |
| manager-name | | 任意 | MessageRecordFactoryが登録されているServiceManagerの名前を定義する。 |
| 14 | server-property | server | なし | サービス定義内の各所で${プロパティ名}で参照できる環境変数プロパティを定義する。  この環境変数のスコープは、server要素配下すべてである。  name属性でプロパティ名を、内容でプロパティ値を定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 必須 | 環境変数プロパティの名前を定義する。 |
| 15 | manager | server | repository?  log?  message?  manager-property\*  service\*  ifdef\* | ServerManagerを構成する。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 任意 | ServiceManagerの名前を定義する。  ServiceManagerを複数生成する場合には、name属性で一意の名前を指定する必要がある。また、同じファイルまたは、別のファイルに同じname属性を持ったmanager要素を指定する事もできる。  ServiceManagerを複数生成しない場合には、name要素は省略可能である。省略した場合、ServiceManagerの名前は"Nimbus"となり、名前指定なしでServiceManagerFactory#findManager()で取得する事ができる。 |
| shutdown-hook | | 任意 | ServiceManagerに、ShutdownHookを設定するかどうかを定義する。  trueを設定した場合は、JVMの終了と同時に、ServiceManager#destroy()が呼び出されて終了処理が行われる。  指定しない場合は、false。 |
| 16 | repository | manager | なし | manager要素で構成されるServiceManagerに、service要素で構成されるServiceを格納するRepositoryサービスのサービス名を定義する。  サービス名は、manager-name属性でServiceManagerの名前を、内容でサービスの名前を指定する方法と、内容にServiceManagerの名前#サービスの名前で指定する方法がある。 | |
| 属性名 | | 必須/任意 | 説明 |
| manager-name | | 任意 | Repositoryが登録されているServiceManagerの名前を定義する。 |
| 17 | manager -property | manager | なし | サービス定義内の各所で${プロパティ名}で参照できる環境変数プロパティを定義する。  この環境変数のスコープは、manager要素配下すべてである。  name属性でプロパティ名を、内容でプロパティ値を定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 必須 | 環境変数プロパティの名前を定義する。 |
| 18 | service | manager  depends | constructor?  field\*  attribute\*  optional-config?  invoke\*  depends\*  ifdef\* | Serviceを定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| code | | 必須 | Serviceの実装クラス名を完全修飾名で定義する。  ここに定義できるクラスは、基本的に制限はないが、クラスの実装によって、動作的な制限が発生する。以下に、推奨する実装方法の順で、説明する。  (1)jp.ossc.nimbus.core.ServiceBaseのサブクラス  ロード時に、create()、start()が呼び出される。  また、ServiceManager#create()、start()、stop()、destroy()を呼び出すと、ServiceManagerに登録されているServiceの対応するメソッドが呼び出される。  ServiceBaseは、JMXに対応している。  (2)jp.ossc.nimbus.core.FactoryServiceBaseのサブクラス  FactoryServiceBaseは、ServiceBaseのサブクラスであるため、(1)に準拠する。  但し、ServiceManager#getServiceObject(String)を呼び出された場合、このクラスのインスタンスではなく、FactoryService#newInstance()で生成されるオブジェクトを返す。  (3)jp.ossc.nimbus.core.ServiceBaseSupportインタフェース実装クラス  ServiceBaseでラップされ、ServiceManagerに登録される。  ロード時に、createService()、startService()が呼び出される。  また、ServiceManager#create()、start()、stop()、destroy()を呼び出すと、ServiceManagerに登録されているServiceBaseを通して、createService()、startService()、stopService()、destroyService()が呼び出される。  ラッパーであるServiceBaseが、JMXに対応している。  また、FactoryServiceインタフェースも実装すれば、(2)に準拠した動作をする。  (4)jp.ossc.nimbus.core.Serviceインタフェース実装クラス  ロード時に、Service#setServiceName()、create()、start()が呼び出される。  また、ServiceManager#create()、start()、stop()、destroy()を呼び出すと、Serviceの対応するメソッドが呼び出される。  JMXには、対応していない。対応するためには、独自にMBeanを実装する必要がある。  (5)上記以外のクラス  Serviceインタフェースを実装していない場合は、Serviceのプロキシを生成して、ServiceManagerに登録される。  Service#create()、start()、stop()、destroy()と同じシグニチャのメソッドを持っている場合は、(1)と同じ動作をする。  持っていない場合は、ServiceManager#create()を呼び出すと、ServiceManagerへ自分自身を登録する事と、ServiceManager#destroy()を呼び出すと、ServiceManagerから自分自身を削除する。  また、Service#getState()は、CREATED、DESTROYED以外の状態はUNKNOWNとなる。 |
| name | | 必須 | Serviceの名前を定義する。  ここで指定した名前を、ServiceManager#getService(String)の引数に指定する事で、このServiceを取得できる。  また、ServiceManager#getServiceObject(String)を呼ぶと、code属性で指定したクラスのインスタンスが取得できる。但し、FactoryServiceインタフェースを実装している場合は、FactoryService#newInstance()で生成されたインスタンスが取得される。 |
| initState | | 任意 | Serviceのロード直後の初期状態を定義する。  設定可能な値は、Created、Startedで、デフォルトは、Startedである。 |
| instance | | 任意 | Serviceのインスタンス生成種別を定義する。  設定可能な値は、singleton、factory、threadlocal、template。  デフォルト値は、singletonで、インスタンスが１つだけ生成され登録される。  factoryを設定すると、ServiceManager#getServiceObject(String)を呼び出す毎に新しいインスタンスを生成して返す。また、ServiceManager#getService(String)の戻り値は、FactoryServiceになる。  threadlocalを設定すると、ServiceManager#getServiceObject(String)を呼び出すスレッド毎に、インスタンスを生成して返す。また、ServiceManager#getService(String)の戻り値は、FactoryServiceになる。  templateを設定すると、サービス定義のテンプレート定義となり、この定義自体では、サービスは登録されず、templateName属性でこの定義をテンプレートとして取り込む事ができる。 |
| management | | 任意 | instance属性にfactoryまたはthreadlocalを指定した場合に、生成されたインスタンスをServiceManagerに登録するかどうかを定義する。  デフォルト値は、falseで、生成されたインスタンスはServiceManagerに登録されない。  trueを設定すると、生成されたインスタンスはServiceManagerに登録される。  登録されるインスタンスのサービス名は、このservice要素のname属性の値に、通番を付けた名前になる。 |
| createTemplate | | 任意 | instance属性にfactoryまたはthreadlocalを指定した場合に、ファクトリの開始時にファクトリが生成するオブジェクトを生成してみるかどうかを定義する。  デフォルト値は、trueで、ファクトリの開始時にファクトリが生成するオブジェクトの生成を試みる。 |
| template | | 任意 | テンプレートとして参照するservice要素のサービス名を定義する。 |
| 19 | constructor | service  object | argument+  invoke  static-invoke  static-field-ref | serviceやobject要素でオブジェクトを生成する際のコンストラクタを定義する。  デフォルトコンストラクタ以外のコンストラクタを指定したい場合に指定する。 | |
| 20 | argument | constructor  invoke  static-invoke | service-ref  object  static-invoke  static-field-ref | 引数として渡す値を内容または子要素で定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| type | | 任意 | 引数の型を指定する。  子要素を定義する場合で、引数の型が  それらの要素が表すオブジェクトの型と等しい場合は、省略可能。 |
| valueType | | 任意 | 引数に与えるオブジェクト型を指定する。  引数の型と、引数で与えるオブジェクトの型が代入互換の関係にあり、等しい型ではない場合に指定する。 |
| nullValue | | 任意 | 引数の値がnullである事を指定する。  引数にnullを渡したい場合に、trueを指定する。 |
| 21 | field | service  object | service-ref  object  static-invoke  static-field-ref | フィールドに設定する値を内容または子要素で定義する。  内容で指定された値は、java.beans.PropertyEditorによって編集される。  また、対象となるフィールドは、publicである必要がある。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 必須 | フィールド名を指定する。 |
| type | | 任意 | フィールドの型を指定する。  name属性で定義した名前のフィールドの型と、内容の文字列を編集するPropertyEditorが対応する型が同じ場合は、  省略可能。 |
| 22 | attribute | service  object | service-ref  object  static-invoke  static-field-ref | 属性に設定する値を内容または子要素で定義する。  内容で指定された値は、java.beans.PropertyEditorによって編集される。  また対象となる属性を設定するsetterは、publicである必要がある。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 必須 | 属性名を指定する。  ここで指定する属性名は、publicなクラスのpublicなプロパティでなければならない。また、プロパティ名は最初の1文字以外は大文字小文字を区別する。  プロパティの概念は、Java Beansのプロパティの概念より広く、jp.ossc.nimbus.beans.PropertyFactoryの規約に従う。  Java Beansのような単純なプロパティに加え、java.util.Mapなどのマッププロパティやjava.util.List、配列などのインデックスプロパティ、またそれらがネストしたプロパティなどもサポートする。 |
| type | | 任意 | 属性の型を指定する。  name属性で定義した属性名に対するsetterが用意されていない場合で、Mapインタフェースを実装している場合に、String型以外の型で値を設定する必要がある場合は、この属性で型を定義する。  name属性で定義した属性名に対するsetterが用意されている場合でも、そのsetterの引数の型と代入互換の関係にある別の型で値を設定したい場合などにも、この属性で必要な型を定義する。  また、DOMのElementをそのまま設定したい場合は、この属性に、"org.w3c.dom.Element"を指定する。 |
| 23 | optional-config | service | 任意 | ユーザが任意のフォーマットでサービスの構成情報を定義する。  ここで、設定した情報は、ServiceManager#getServiceMetaData(String)で取得できるServiceMetaDataのServiceMetaData#getOptionalConfig()で、org.w3c.dom.Elementが取得できる。 | |
| 24 | invoke | constructor  service  object  target | target?  argument\* | メソッド実行を定義する。  引数ありのメソッドを実行したい場合は、子要素にargument要素を定義する。  この要素がconstructor要素またはtarget要素の子要素として現れる場合のみ、子要素にtarget要素を持つ事ができる。target要素では、呼び出し対象のオブジェクトを指定する。  Serviceやオブジェクトには、該当するpublicなメソッドを実装しておく必要がある。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 必須 | 実行するメソッド名を定義する。 |
| callState | | 任意 | メソッドが実行されるライフサイクル上のタイミングを定義する。  但し、この属性は、service要素の子要素として定義したinvoke要素でのみ有効。  設定可能な値は、以下。  Creating  Created  Starting  Started  Stopping  Stopped  Destorying  Destroyed  デフォルトは、Created。 |
| 25 | target | invoke | object  service-ref  invoke  static-invoke  static-field-ref | invoke要素でメソッド呼び出しを行う対象のオブジェクトを定義する。 | |
| 26 | depends | service | service | service要素で登録するServiceが、依存するServiceを定義する。  depends要素の内容に、依存するServiceのサービス名を指定するか、service要素を定義する。  サービス名は、manager-name属性でServiceManagerの名前を、内容でサービスの名前を指定する方法と、内容にServiceManagerの名前#サービスの名前で指定する方法がある。  この要素が指定されている場合は、依存するServiceの状態に追随して生成、開始される。  また、Serviceの停止及び破棄では、依存するServiceの前に、停止、及び破棄される。 | |
| 属性名 | | 必須/任意 | 説明 |
| manager-name | | 任意 | 依存するServiceが登録されているServiceManagerの名前を定義する。 |
| 27 | static-invoke | constructor  argument  field  attribute  target | argument\* | staticメソッドの呼び出しを定義する。  引数ありのメソッドを実行したい場合は、子要素にargument要素を定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| code | | 必須 | staticメソッドが宣言されているクラス名を完全修飾名で定義する。 |
| name | | 必須 | staticメソッド名を定義する。 |
| 28 | static-field-ref | constructor  argument  field  attribute  target | なし | staticフィールド参照を定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| code | | 必須 | staticフィールドが宣言されているクラス名を完全修飾名で定義する。 |
| name | | 必須 | staticフィールド名を定義する。 |
| 29 | object | argument  field  attribute  target | constructor?  field\*  attribute\*  invoke\*  ifdef\* | 生成する任意のオブジェクトを定義する。  この要素で定義されたオブジェクトは、ServiceManagerには登録されない点が、service要素とは異なる。 | |
| 属性名 | | 必須/任意 | 説明 |
| code | | 必須 | 生成するオブジェクトの実装クラス名を完全修飾名で定義する。 |
| 30 | service-ref | argument  field  attribute  target | なし | 参照するServiceのサービス名を定義する。  サービス名は、manager-name属性でServiceManagerの名前を、内容でサービスの名前を指定する方法と、内容にServiceManagerの名前#サービスの名前で指定する方法がある。 | |
| 属性名 | | 必須/任意 | 説明 |
| manager-name | | 任意 | 参照するServiceが登録されているServiceManagerの名前を定義する。 |
| 31 | ifdef | server  property-editors  manager  service  object | 任意 | 子要素を有効にする条件を定義する。 | |
| 属性名 | | 必須/任意 | 説明 |
| name | | 必須 | 条件となる環境変数名を定義する。  ここで言う環境変数は、JVMのシステムプロパティ、server-property要素で指定した変数、manager-property要素で指定した変数を指す。 |
| value | | 必須 | 子要素を有効にする条件となる環境変数の値を定義する。 |

以下にサービスの属性一覧を示す。

|  |  |  |  |
| --- | --- | --- | --- |
| # | 属性名 | 型 | 説明 |
| 1 | SystemLoggerServiceName | ServiceName | ログ出力に使用するjp.ossc.nimbus.service.log.Loggerサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つLoggerを使用する。 |
| 2 | SystemMessageRecordFactoryServiceName | ServiceName | メッセージ取得に使用するjp.ossc.nimbus.service.message.MessageRecordFactoryサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つMessageRecordFactoryを使用する。 |
| 3 | ServiceURL | URL | サービス定義ファイルのURLを設定する。 |
| 4 | ServicePath | String | サービス定義ファイルのパスを設定する。  ここで指定されたパスは、以下の手順でURLに変更されて、ServiceURL属性として解釈される。   1. 指定されたパスがnull、または空文字の場合、デフォルトURL（後述） 2. 指定されたパスがローカルファイルとして存在する場合、ローカルパスをURLに変換したURL 3. 指定されたパスがこのクラスをロードしたクラスローダのリソースとして存在する場合、そのURL 4. 上記全てに当てはまらない場合、例外をthrowする。   デフォルトURLの決定は、以下の順序で行われる。   1. システムプロパティjp.ossc.nimbus.service.urlで指定された値を、上記のパス→URL変換でURLに変換した値 2. このクラスのクラスファイルがロードされたクラスパス上から、nimbus-service.xmlをClassLoader#getResource(String)でリソースとして取得したURL。このクラスのクラスファイルがJarファイルに格納されている場合は、そのJarファイルと同じパス上のnimbus-service.xmlのURL |
| 5 | ServiceManagerClassName | String | manager要素を表すServiceManagerインタフェースの実装クラス名を設定する。  クラス名は、完全修飾名で設定する。  また、設定しない場合は、デフォルトの実装クラスが使用される。デフォルトは、jp.ossc.nimbus.core.DefaultServiceManagerServiceである。 |
| 6 | Validate | boolean | サービス定義ファイルを評価するかどうかを指定する。  デフォルトはfalseで、評価しない。 |
| 7 | CheckLoadManagerCompleted | boolean | ここまでにロードしたサービスが全て正常に開始できているかをチェックするかどうかを設定する。  この属性をtrueにしておくと、このServiceLoaderの起動完了時に、ServiceManagerFactory#checkLoadManagerCompleted()を呼び出す。 |
| 6 | CheckLoadManagerCompletedBy | String[] | 指定したマネージャのサービスが全て正常に開始できているかをチェックするかどうかを設定する。  この属性を指定すると、このServiceLoaderの起動完了時に、ServiceManagerFactory#checkLoadManagerCompletedBy(Set)を呼び出す。 |

## ServiceManager

### jp.ossc.nimbus.core.DefaultServiceManagerService

ServiceManagerインタフェースのデフォルト実装クラスである。

addServiceLoader(loader)

start()

ServiceLoader

ServiceMetaData

**DefaultServiceManagerService**

create()

start()

ServiceLoader

registerService(data)

ServiceLoader

register(name, service)

registerManager(name, manager)

Repository

Service

Service

ServiceManagerFactory

ServiceLoaderからregisterService()で、ServiceMetaDataを登録されると、その内容に従いサービスを生成して、Repositoryにサービスを登録する。ServiceLoaderは、サービスを登録し終えると、自分自身をServiceManagerに登録して、ServiceManagerを開始する。開始されたServiceManagerは、各サービスを依存関係に従って、生成、開始する。最後に自分自身をServiceManagerFactoryに登録する。

また、DefaultServiceManagerServiceは、RegistrationBroadcasterインタフェースを実装しており、登録されたRegistrationListenerに対して、サービスの登録状態のイベント通知を行う。

以下にサービスの属性一覧を示す。

|  |  |  |  |
| --- | --- | --- | --- |
| # | 属性名 | 型 | 説明 |
| 1 | SystemLoggerServiceName | ServiceName | ログ出力に使用するjp.ossc.nimbus.service.log.Loggerサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つLoggerを使用する。 |
| 2 | SystemMessageRecordFactoryServiceName | ServiceName | メッセージ取得に使用するjp.ossc.nimbus.service.message.MessageRecordFactoryサービスの名前を設定する。  設定しない場合は、ServiceManagerの持つMessageRecordFactoryを使用する。 |

## ServiceManagerFactory

ServiceManagerをJavaVM上で静的に管理するクラスで、Nimbus Coreにアクセスするための静的なエントリポイントとなる。

setServiceURL(url)、start()

loadManager(url)

サービス定義

**ServiceManagerFactory**

ServiceLoader

ServiceMetaData

ServiceMetaData

registerService(data)

start()

ServiceManager

Service

registerManager(manager)

register(service)

findManager(name)

ServiceManager

ServiceManager

Service

Service

Repository

Repository

create()

start()

register(manager)

ServiceManager

ServiceManagerのstaticなメソッドloadManager(url)を呼び出す事で、ServiceLoaderを生成して、サービス定義を読み込み、サービスを開始状態にする事ができる。

ServiceManagerのstaticなメソッドfindManager(name)で、ServiceManagerの名前を指定して、取得する事ができる。取得したServiceManagerのgetServiceObject(name)メソッドで、サービスの名前を指定して、サービスを取得する事ができる。

また、ServiceManagerのstaticなメソッドgetServiceObject(managerName, serviceName)を呼び出す事で、サービスを取得する事ができる。このように、ServiceManagerの機能をショートカットするようなstaticメソッドも持っている。

以下に、ServiceManagerFactoryで、サービス定義を読み込み、サービスを取得する例を示す。

import jp.ossc.nimbus.core.ServiceManagerFactory;

import jp.ossc.nimbus.core.ServiceManager;

// サービス定義ファイルを読み込む

if(!ServiceManagerFactory.loadManager(“nimbus-service.xml”)){

System.out.println(“サービス定義の読み込みに失敗しました。”);

System.exit(-1);

}

// サービスが全て正常に開始できたかチェックする

if(!ServiceManagerFactory.checkLoadManagerCompleted()){

System.out.println(“サービスが正しく開始できません。”);

System.exit(-1);

}

// ServiceManagerの名前を指定して、ServiceManagerを取得する

ServiceManager manager1 = ServiceManagerFactory.findManager(“Manager1”);

// サービスの名前を指定して、サービスを取得する

Object service1 = manager1.getServiceObject(“Service1”);