目次

改訂情報

変更年月日	変更内容
2012-10-01	初版
2015-04-01	第2版 下記を追加・変更しました ■ 「*タスクキュー* 」の内容を変更しました。 ■ 「*タスク処理サービス* 」の内容を変更しました。 ■ 「*設定* 」から「task-queue-storage-config.xml」の説明を削除しました。
2016-08-01	第3版 下記を追加・変更しました ■ 「*設定* 」の「*task-runner-config.xml* 」に注意事項を追加しました。
2017-08-01	第4版 下記を追加・変更しました ■ 「*状態管理* 」に非同期処理機能の状況取得に関するコラムを追加しました。

はじめに

本書の目的

本書では非同期処理機能の仕組の詳細について説明します。

説明範囲は以下のとおりです。

- 非同期処理機能の構成
- 動作原理
- 非同期処理機能を利用した開発をする上で必要となるAPIの詳細

対象読者

本書では次の利用者を対象としています。

- intra-mart Accel Platformを理解している
- 非同期処理機能の管理者
- 以下のいずれかの条件を満たす、非同期処理機能を利用する開発者
  - Java を理解している開発者
  - サーバサイドJavaScript を理解している開発者

本書の構成

本書は以下のような構成です。

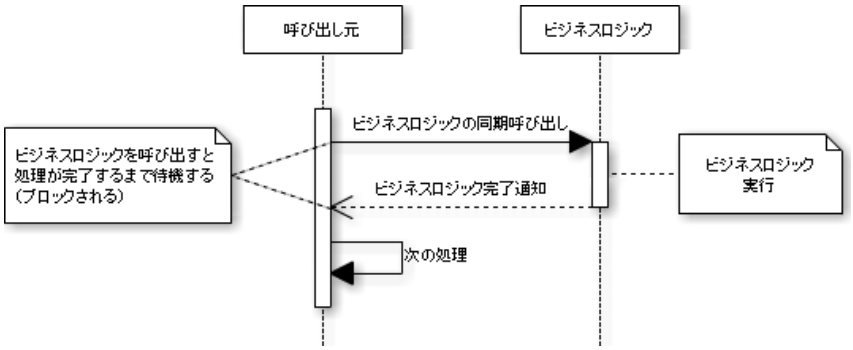
- *概要*
  この章では非同期処理機能の概要について説明します。
  - 非同期処理機能の概要
  - 全体的な流れ
  - 構成要素
- *タスク*
  この章ではタスクを実装する上で必要な事項について説明します。
  タスクはビジネスロジックを実行するコンポーネントです。
- *並列処理機能*
  この章では複数のタスクを同時に実行可能とする並列処理機能の動作原理と利用方法について説明します。
- *直列処理機能*
  この章では複数のタスク間に厳密な順序関係がある場合に使用する直列処理機能の動作原理と利用方法について説明します。
- *タスク処理*
  この章ではタスクの登録方法から実際にビジネスロジックが実行されるまでをタスクの観点から説明します。
- *キュー管理*
  この章ではタスクキューの管理方法について説明します。
  ビジネスロジックの情報であるタスクメッセージは一旦タスクキューに登録され、後に実行されます。
- *終了通知と強制停止*
  この章ではビジネスロジックが実行中であるタスクを途中で終了するための方法について説明します。
  何らかの理由によってタスクのビジネスロジックの終了を待たないで停止させたい場合に使用します。
- *ライフサイクル*
  この章では非同期処理機能における様々なイベントや状態について説明します。
- *状態管理*
  この章では非同期処理機能の現在の状態を監視する方法について説明します。
- *設定*
  この章では非同期処理機能を利用するために必要な環境の設定情報等について説明します。

概要

非同期処理とは？

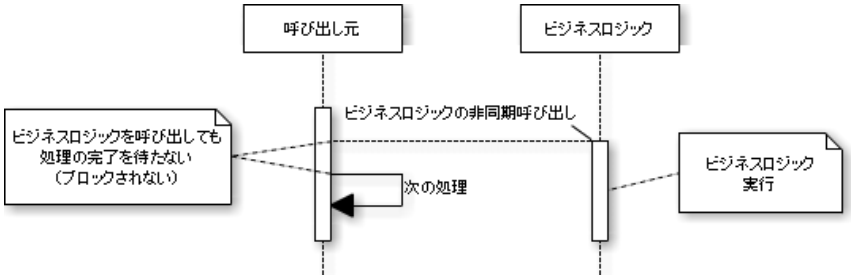
非同期はビジネスロジックを呼び出して処理を行うための一つの方法です。

通常のビジネスロジックの呼び出しは同期的です。 ここで言う「同期的」とは、ビジネスロジックの処理が完了するまで呼び出し側の処理が待機状態になる呼び出し方法を意味します。 *同期的呼び出し*を参照してください。



同期的呼び出し

「同期的な呼び出し」とは対照的に「非同期的な呼び出し」では、呼び出し側はビジネスロジックの終了を待たずに次の処理を続行します。 呼び出されたビジネスロジックは呼び出し元の処理とは別スレッドで開始されます。 *非同期的呼び出し*を参照してください。



非同期的呼び出し

ビジネスロジックの実行結果の取得が重要ではない場合、非同期処理機能を利用することによって、全体的な応答を早めることが可能です。

以下のような条件を満たす処理を行う場合、処理の呼び出し側とは別のスレッドで処理を行うとユーザインタフェースの応答を早くできる場合があります。

- 処理時間そのものは比較的短いが、ユーザインタフェースの観点からすると応答時間が長い。（数秒～数十秒程）
- 処理の実行はリアルタイムである必要はないが、処理要求後にできるだけ早く実行したい。
- 処理の呼び出し側では、処理の完了については特に気にする必要はない。

背景

非同期的な呼び出しによって処理を行う最も単純な方法は、スレッドを生成することです。

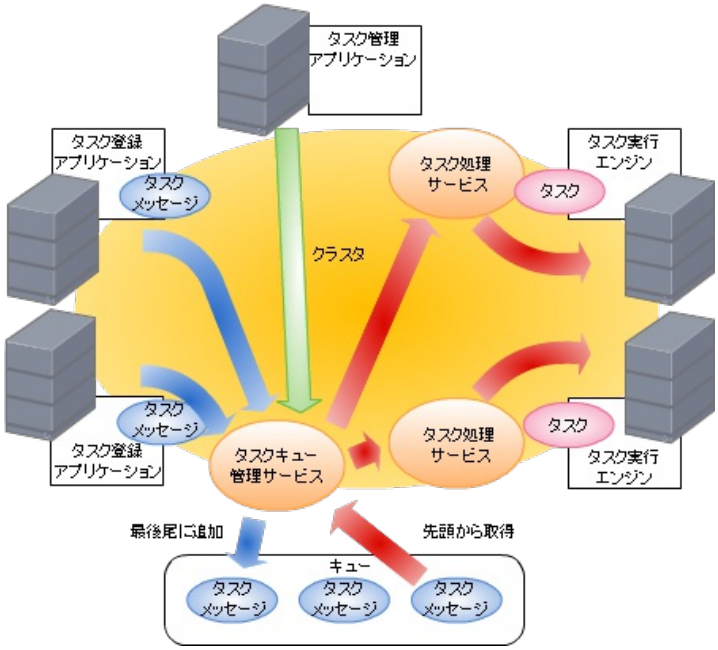
しかしながら、Java EE の仕様では、開発者が独自にスレッドを生成することを原則的に非推奨、または、禁止としています。

非同期処理機能を利用することによって、開発者は明示的にスレッドを作成することなく処理を非同期に行うことが可能です。

> **コラム**
>
> 非同期処理機能では個々の処理時間が比較的短いものを対象としています。
> 処理時間が極端に長い場合（数十分～数時間程）は、非同期処理機能ではなく、バッチやスケジューラ等で実行することを考慮してください。

全体像

*非同期処理機能の全体像（概要）*に非同期処理機能の全体像の概要を示します。



非同期処理機能の全体像（概要）

用語

非同期処理機能には以下のような構成要素があります。

タスク登録アプリケーション

*タスク登録アプリケーション*は*タスクメッセージ*を生成し、*タスクキュー管理サービス*に登録するアプリケーションです。
*タスク登録アプリケーション*は開発者によって提供される必要があります。

タスクキュー管理サービス

*タスクキュー管理サービス*は*タスクメッセージ*および*タスクキュー*の登録および実行状況を管理します。

*タスクキュー管理サービス*を利用することで以下のようなことが可能です。

- *タスクキュー*に対して*タスクメッセージ*を登録または削除します。
- *タスクキュー*の処理を活性化または非活性化します。
- *直列タスクキュー*を追加または削除します。
- *タスクキュー*および*タスクキュー*に現在登録されている*タスクメッセージ*の情報を取得します。

*タスクキュー管理サービス*はintra-mart Accel Platformの非同期処理機能から提供されます。

タスクメッセージ

*タスクメッセージ*は非同期処理機能を利用してビジネスロジックを実行するために必要な情報です。 *タスクメッセージ*は開発者によって提供される必要があります。

個々の*タスクメッセージ*は*メッセージID*によって一意に識別されます。

詳細は*タスク*を参照してください。

メッセージID

*タスクメッセージ*を*タスクキュー*に登録した時に、非同期処理機能から割り当てられる識別子です。

同じ情報を持つ*タスクメッセージ*を複数登録した場合でも、すべて異なる*メッセージID*が割り当てられます。

タスク

*タスクメッセージ*を元にし、実際にビジネスロジックを実行できる形に変換されたものが*タスク*です。

*タスクメッセージ*から*タスク*へ変換する機能は非同期処理機能から提供されます。

*タスク*のクラスの実装は開発者によって提供される必要があります。

詳細は*タスク*を参照してください。

タスクキュー

*タスクキュー*は*タスクメッセージ*を保存します。 *タスクキュー*はその形態によってさらに以下の2種類に分類されます。

- *並列タスクキュー*
- *直列タスクキュー*

並列タスクキュー

*並列タスクキュー*は、同時に処理を行うことが可能な*タスクメッセージ*のみで構成される*タスクキュー*です。 並列処理機能で利用されます。

並列タスクキュー は 非同期処理機能 から テナント 内に一つのみ提供されます。

並列処理機能および*並列タスクキュー*の詳細については*並列処理機能*を参照してください。

直列タスクキュー

*直列タスクキュー*は、逐次的に処理を行うことが必要な*タスクメッセージ*のみで構成される*タスクキュー*です。 直列処理機能 で利用されます。

*直列タスクキュー*は複数存在することが可能です。 *直列タスクキュー*はそれぞれ異なるキューIDを割り当てる必要があります。

*直列タスクキュー*はAPIを通じて自由に登録または削除することが可能です。

*直列タスクキュー*は開発者および管理者によって管理される必要があります。

複数の*タスク*が存在し、以下の条件のいずれかに該当する場合は「直列処理が必要」とみなされます。

- *タスク*間に明確な順序関係が存在する
- 直前の*タスク*が完了した後でなければ、次の*タスク*を開始してはいけない

なお、*直列タスクキュー*が異なれば、複数の*タスク*が同時に実行される場合もあります。

タスク処理サービス

*タスク処理サービス*は*タスクキュー*に登録された*タスクメッセージ*を先頭から取得し、*タスク実行エンジン*に渡します。 *タスクメッセージ*は*タスクキュー*に登録された順に処理されます。

*タスク処理サービス*はintra-mart Accel Platformのサービスの一つであり、非同期処理機能から提供されます。

タスク実行エンジン

*タスク実行エンジン*は*タスク処理サービス*から渡された*タスクメッセージ*を元に*タスク*を生成し、ビジネスロジックを非同期で実行します。

*タスク実行エンジン*は非同期処理機能から提供されます。

*タスク実行エンジン*は1つの*タスク処理サービス*に対して1つのみ起動されます。

タスク管理アプリケーション

*タスク管理アプリケーション*は*タスク処理サービス*を通じて*タスクキュー*や、そこに登録されている*タスクキュー*を管理または操作するアプリケーションです。

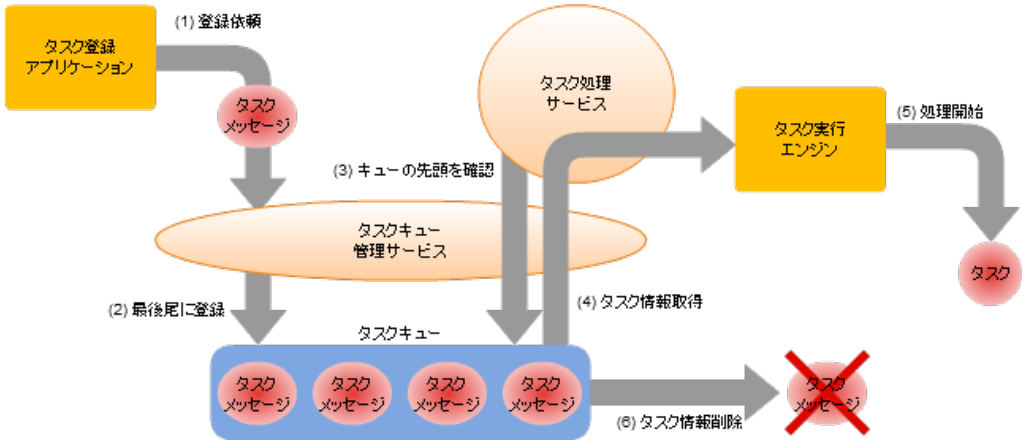
*タスク管理アプリケーション*は非同期処理機能からも提供されますが、開発者が独自に作成することも可能です。

全体の流れと非同期の種類

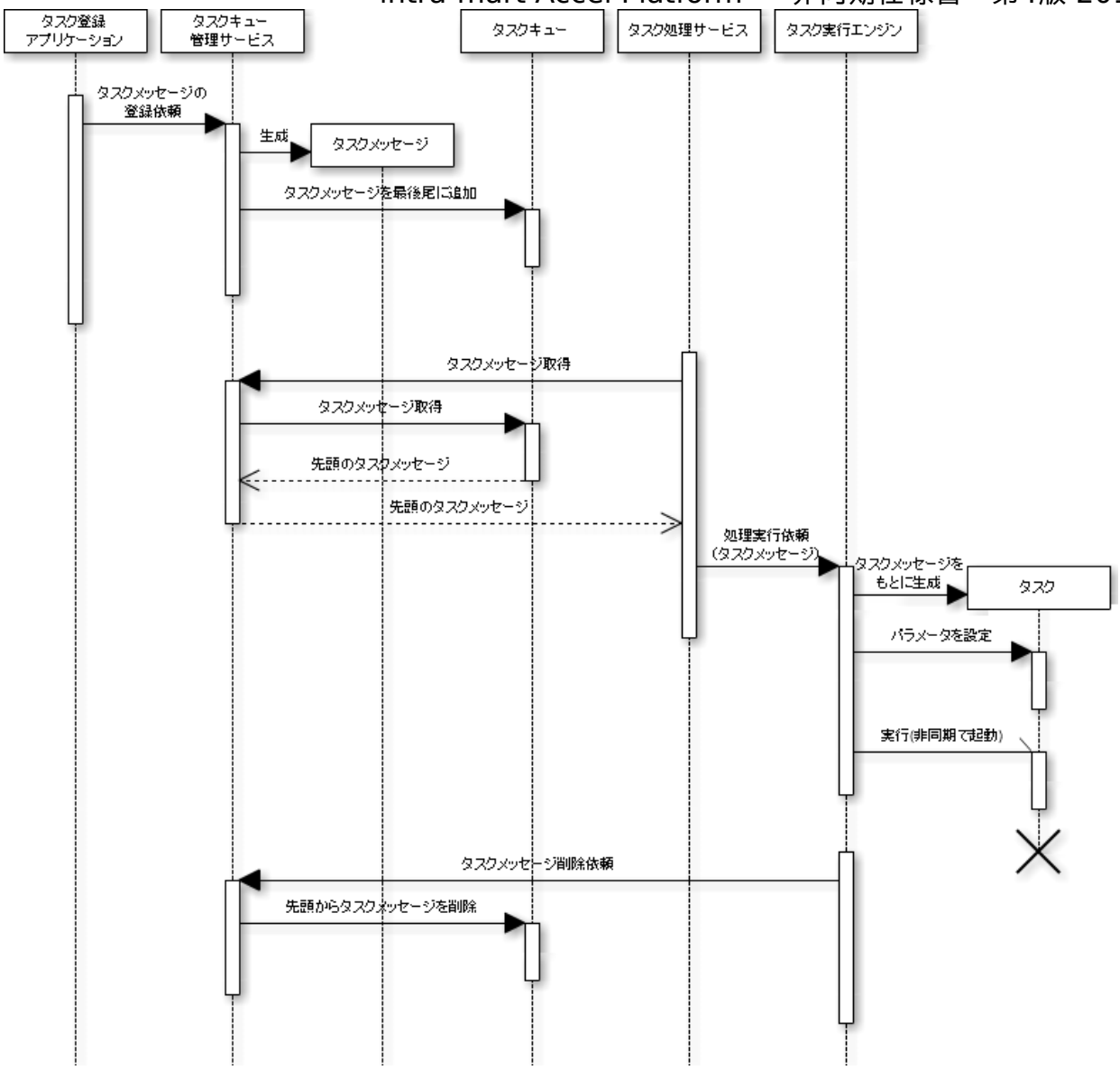
全体の流れ

非同期処理機能を利用する場合、概ね以下のような流れです。*非同期処理機能の流れ*および*非同期処理機能のシーケンス*も参照してください。

1. *タスク登録アプリケーション*から*タスクキュー管理サービス*に対して*タスクメッセージ*の登録を依頼します。
2. *タスクキュー管理サービス*は*タスクメッセージ*を指定された*タスクキュー*の最後尾に登録します。
3. *タスク処理サービス*は*タスクキュー管理サービス*を通じて定期的に*タスクキュー*を観察します。
4. *タスク処理サービス*は実行可能な *タスクメッセージ*が登録されていることを確認したらその*タスクメッセージ*を取得し、*タスク実行エンジン*に渡します。
5. *タスク実行エンジン*は*タスクメッセージ*を元にして*タスク*を生成し、実行します。
6. *タスク実行エンジン*は*タスク*の実行が完了したら*タスクメッセージ*を*タスクキュー*の先頭から削除します。



非同期処理機能の流れ



非同期処理機能のシーケンス

非同期処理の種類

非同期処理機能の処理方法は以下の2種類に分類されます。

- 並列処理機能
  複数のタスクを同時に実行します。
  詳細については*並列処理機能*を参照してください。
- 直列処理機能
  複数のタスクの間に明確な順序関係がある場合、先行するタスクの処理が完了するまでは後続のタスクの処理をブロックします。
  詳細については*直列処理機能*を参照してください。

タスク

タスクはビジネスロジックを実行する部分です。 非同期処理機能を利用する場合、開発者はタスクを提供する必要があります。

タスクを実装するにあたり、開発言語として以下のものを利用することができます。

- Java
- サーバサイドJavaScript

タスクの実装

Java

Javaでタスクを作成する場合、ビジネスロジックを実行するクラスは以下の条件をすべて満たす必要があります。

- jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを実装したクラスであること
- 以下のいずれのクラスでもないこと。
  - 内部クラス(inner class)
  - ローカルクラス(local class)
  - 無名クラス(anonymous class)
- クラスの修飾子が以下の条件をすべて満たしていること
  - publicであること
  - abstractではないこと
- publicで引数なしのコンストラクタを持っていること
- いずれのメソッドもsynchronizedとして宣言されていないこと
  （メソッド内部でsynchronizedブロックを使用することは可能）

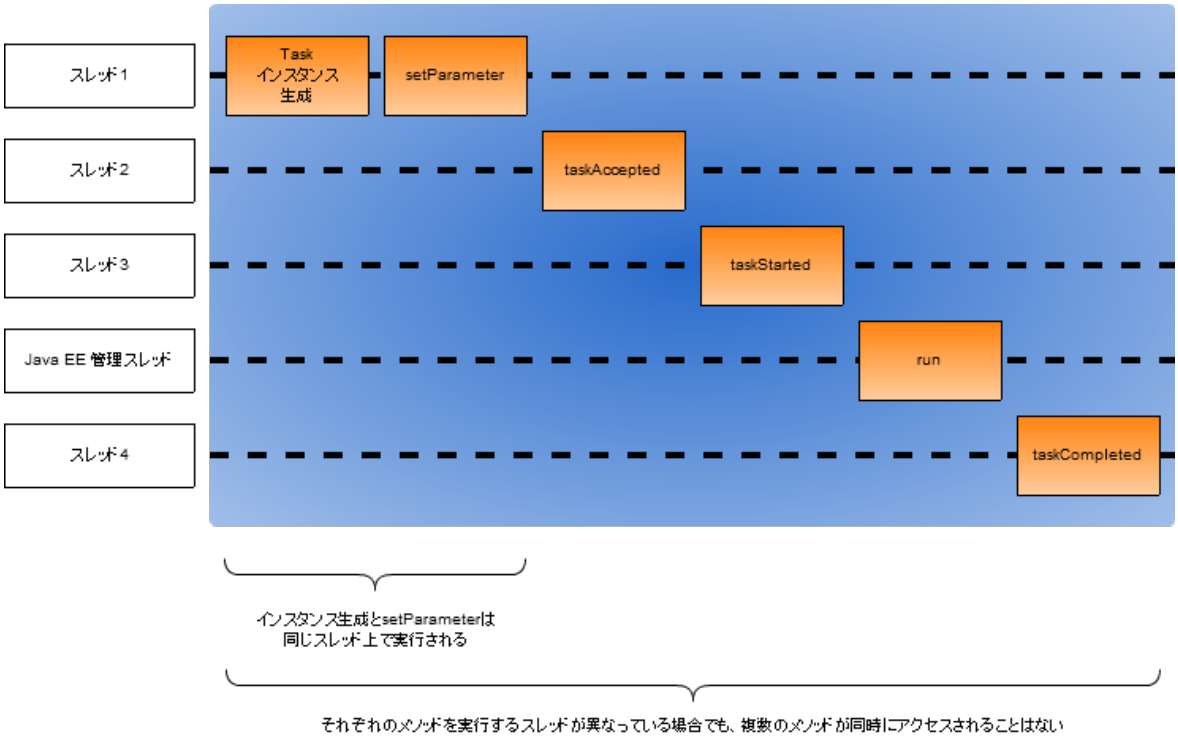
また、Javaでタスクを作成する場合、以下の点について注意してください。

- jp.co.intra_mart.foundation.asynchronous.Taskインタフェースで定義されている各メソッドは異なるスレッドから呼び出される場合があります。
  同一のスレッドであることを前提とする実装をしないでください。

  > **コラム**
  >
  > スレッドに関連付けられているリソース等が存在する場合、リソースの取得と解放を異なるメソッド上で行おうとすると
  > 不具合が発生する場合があります。

  非同期処理機能では、少なくとも以下の異なるスレッドが存在します。
  - タスクのコンストラクタおよび*Task#setParameter*メソッドを呼び出すスレッド
  - タスクの*Task#run*メソッドを呼び出すスレッド
  - タスクの*Task#release*メソッドを呼び出すスレッド

  その他のイベント通知に関連するメソッド
  (*Task#taskAccepted*、*Task#taskStarted*、*Task#taskCompleted*、*Task#taskRejected*)を起動するスレッドは不定です。
  runメソッドと同じスレッドから呼び出される場合もあれば、上記のいずれからも異なるスレッドから呼び出される場合もあります。 そのため、特定のスレッドに依存するような実装はしないようにしてください。
  なお、*Task#run*メソッドおよび*Task#release*メソッドの組み合わせ以外は、同一のタスクにおいて各メソッドが異なるスレッド上から起動された場合であっても、複数のメソッドが同時に実行されないように制御されています。 *異なるスレッド上からメソッドが呼び出される時のイメージ図*も参照してください。
- *Task#run*以外のメソッドではJava EEのリソースやintra-mart Accel Platformから提供されるリソースの取得は行わないでください。



異なるスレッド上からメソッドが呼び出される時のイメージ図

Taskインタフェース

`Task`インタフェース

```
package jp.co.intra_mart.foundation.asynchronous;

import java.util.Map;

public interface Task extends Runnable {

    void release();

    void setParameter(Map<String, ?> parameter);

    void taskAccepted(TaskEvent taskEvent);

    void taskStarted(TaskEvent taskEvent);

    void taskCompleted(TaskEvent taskEvent);

    void taskRejected(TaskEvent taskEvent);
}
```

コンストラクタ

非同期処理機能を利用するタスクを作成する場合、`public`で引数なしのコンストラクタを持っている必要があります。

> **注意**
> 非同期処理機能からコンストラクタによってインスタンスが生成された場合、このコンストラクタの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

Task#run

> **コラム**
> `run()`メソッドは、実際には`Task`インタフェースのスーパーインタフェース`java.lang.Runnable`で宣言されています。

```
public void run();
```

非同期として実行したいビジネスロジックを記述します。

開発者は以下の点に注意してこのメソッドを実装する必要があります。

- 無限ループにならないこと
- *Task#release*が呼ばれた時、速やかに終了すること

非同期処理機能からrunメソッドが呼ばれた場合、このメソッドの内部では以下のリソースを使用することができます。

- コンテキストに保持された値
  タスクメッセージ登録時には、その時点におけるコンテキストの情報も同時に保存されます。
  非同期処理機能からrun()メソッドが実行されている場合、内部でこのコンテキストの情報を取得することができます。
  コンテキストの詳細についてはjp.co.intra_mart.foundation.contextパッケージのAPIリスト等を参照してください。
- JNDIによる参照
  intra-mart Accel Platformが搭載されたWebモジュールから参照可能なJNDI名前空間において、"java:comp"から始まる参照名を利用することが可能です。
- データベースに保持された値
  非同期処理機能からrun()メソッドが実行されている場合、内部でintra-mart Accel Platformで扱うデータベースにアクセスすることが可能です。
  intra-mart Accel Platformで扱うデータベースの詳細についてはjp.co.intra_mart.foundation.databaseパッケージのAPIリスト等を参照してください。
- ストレージサービスに保持された値
  非同期処理機能からrun()メソッドが実行されている場合、内部でストレージサービスにアクセスすることが可能です。
  ストレージサービスの詳細についてはjp.co.intra_mart.foundation.service.client.fileパッケージのAPIリスト等を参照してください。

Task#release

```
public void release();
```

このメソッドは*Task#run*で実行中のビジネスロジックを中断する場合、非同期処理機能から呼び出されます。

このメソッドが呼ばれた場合、速やかにビジネスロジックを終了するようにこのメソッドおよび*Task#run*を実装してください。

このメソッドは*Task#run*を実行しているスレッドとは別のスレッドから呼び出される場合があります。

> **注意**
> 非同期処理機能からreleaseメソッドが呼ばれた場合、このメソッドの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

Task#setParameter

```
public void setParameter(
  java.util.Map<java.lang.String, ?> parameter
);
```

非同期では*TaskManager#addParallelizedTask*を通じてタスクメッセージを登録する際に、独自のパラメータを設定することが可能です。 タスク実行エンジンはビジネスロジックを開始する前にこのメソッドを呼び出し、登録時に設定されたパラメータを渡します。

ここで渡されたパラメータをインスタンス変数等に登録しておけば、Taskインタフェースで定義されているsetParameter以外のすべてのメソッドでその値を参照することができます。

パラメータとして登録できる値の制限については*Javaを利用するときのパラメータの制限*を参照してください。

> **注意**
> 非同期処理機能からsetParameterメソッドが呼ばれた場合、このメソッドの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

Task#taskAccepted

```
public void taskAccepted(
  jp.co.intra_mart.foundation.asynchronous.TaskEvent taskEvent
);
```

タスクキューの中で*選択待機状態*だったタスクメッセージがタスクに変換され、*処理実行可能状態*となった時に呼び出されます。

このメソッドで例外が発生しても非同期処理機能は後続の処理を続行します。 このメソッド内で例外が発生しないような実装を推奨します。

このメソッドが呼び出されるタイミングの詳細については*タスク受付*を参照してください。

> **注意**
> 非同期処理機能からtaskAcceptedメソッドが呼ばれた場合、このメソッドの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

Task#taskStarted

```
public void taskStarted(
  jp.co.intra_mart.foundation.asynchronous.TaskEvent taskEvent
);
```

*処理実行可能状態*となっていたタスクのビジネスロジックが開始される直前に呼び出されます。

このメソッドで例外が発生しても非同期処理機能は後続の処理を続行します。 このメソッド内で例外が発生しないような実装を推奨します。

このメソッドが呼び出されるタイミングの詳細については*タスク処理開始*を参照してください。

> **注意**
> 非同期処理機能からtaskStartedメソッドが呼ばれた場合、このメソッドの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

Task#taskCompleted

```
public void taskCompleted(
  jp.co.intra_mart.foundation.asynchronous.TaskEvent taskEvent
);
```

実行中のタスクのビジネスロジックが完了された直後に呼び出されます。

このメソッドで例外が発生しても非同期処理機能は後続の処理を続行します。 このメソッド内で例外が発生しないような実装を推奨します。

このメソッドが呼び出されるタイミングの詳細については*タスク処理完了*を参照してください。

> **注意**
> 非同期処理機能からtaskCompletedメソッドが呼ばれた場合、このメソッドの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

Task#taskRejected

```
public void taskRejected(
  jp.co.intra_mart.foundation.asynchronous.TaskEvent taskEvent
);
```

何らかの理由でタスクメッセージの受け付けが拒否され、*受付待機状態*にできなかった時に呼び出されます。

このメソッドで例外が発生しても非同期処理機能は後続の処理を続行します。 このメソッド内で例外が発生しないような実装を推奨します。

このメソッドが呼び出されるタイミングの詳細については*タスク受付拒否*を参照してください。

> **注意**
> 非同期処理機能からtaskRejectedメソッドが呼ばれた場合、このメソッドの内部では*Task#run*で説明したリソースを取得できない場合があります。 これらのリソースに依存しないように実装してください。

TaskEventインタフェース

TaskEventインタフェース

```
package jp.co.intra_mart.foundation.asynchronous;

public interface TaskEvent {

  TaskException getException();

  TaskEventType getType();

  Task getTask();
}
```

タスクの各種イベント発生時に呼び出されるメソッドの引数として渡されます。

TaskEvent#getException

```
public jp.co.intra_mart.foundation.asynchronous.TaskException getException();
```

イベント発生時の例外情報を取得します。

例外が発生していなかった場合はnullが返されます。

TaskEvent#getType

```
public jp.co.intra_mart.foundation.asynchronous.TaskEventType getType();
```

発生したイベントの種類を取得します。

種類	内容
TaskEventType.TASK_ACCEPTED	*Task#taskAccepted*が呼び出されたことを表します。
TaskEventType.TASK_STARTED	*Task#taskStarted*が呼び出されたことを表します。
TaskEventType.TASK_COMPLETED	*Task#taskCompleted*が呼び出されたことを表します。
TaskEventType.TASK_REJECTED	*Task#taskRejected*が呼び出されたことを表します。

TaskEvent#getTask

```
public jp.co.intra_mart.foundation.asynchronous.Task getTask();
```

イベント発生元のタスクのインスタンスが返されます。

AbstractTask抽象クラス

タスクの実装方法としてjp.co.intra_mart.foundation.asynchronous.AbstractTaskクラスのサブクラスを作成する方法もあります。このクラスは*jp.co.intra_mart.foundation.asynchronous.Task*インタフェースのほとんどのメソッドにおいて何もしないように実装しているので、開発者は必要なメソッドをオーバライドするだけでタスクを実装できます。

また、jp.co.intra_mart.foundation.asynchronous.AbstractTaskには以下のメソッドが追加されています。

AbstractTask抽象クラスに追加されているメソッド

```
public java.util.Map<java.lang.String, ?> getParameter()
```

上記のメソッドを呼び出すことで*Task#setParameter*メソッドで設定されたパラメータを取得することができます。

サーバサイドJavaScript

サーバサイドJavaScriptでタスクを作成する場合、 以下の関数が定義されたJSファイルを用意する必要があります。

- *run()*（必須）
- *setParameter(parameter)*（任意）
- *taskAccepted(event)*（任意）
- *taskStarted(event)*（任意）
- *taskCompleted(event)*（任意）
- *taskRejected(event)*（任意）

> **コラム**
> それぞれの関数は*Taskインタフェース*で定義している内容とほぼ一致しますが、*Task#release*に該当するものはサーバサイドJavaScriptでは用意されていません。

JSファイルは*WorkManager#addParallelizedTask*の引数で指定されたものです。

JSファイル

run()

この関数の定義は必須です。

非同期として実行したいビジネスロジックを記述します。

非同期処理機能からこのメソッドが呼ばれた場合、このメソッドの内部では以下の情報を取得することができます。

- コンテキストに保持された値
  タスクメッセージ登録時には、その時点におけるコンテキストの情報も同時に保存されます。
  非同期処理機能からrun()関数が実行されている場合、内部でこのコンテキストの情報を取得することができます。
  コンテキストの詳細についてはサーバサイドJavaScriptのAPIリスト等を参照してください。
- データベースに保持された値
  非同期処理機能からrun()関数が実行されている場合、内部でintra-mart Accel Platformで扱うデータベースにアクセスすることが可能です。
  intra-mart Accel Platformで扱うデータベースの詳細についてはサーバサイドJavaScriptのAPIリスト等を参照してください。
- ストレージサービスに保持された値
  非同期処理機能からrun()関数が実行されている場合、内部でストレージサービスにアクセスすることが可能です。
  ストレージサービスの詳細についてはサーバサイドJavaScriptのAPIリスト等を参照してください。

開発者は以下の点に注意してこのメソッドを実装する必要があります。

- 例外を出力しないこと
- 無限ループにならないこと

setParameter(parameter)

この関数の定義は任意です。

非同期では*TaskManager#addParallelizedTask*を通じてタスクメッセージを登録する際に、独自のパラメータを設定することが可能です。 タスク実行エンジンはビジネスロジックを開始する前にこのメソッドを呼び出し、登録時に設定されたパラメータを渡します。

ここで渡されたパラメータをインスタンス変数等に登録しておけば、Taskインタフェースで定義されているsetParameter以外のすべてのメソッドでその値を参照することができます。

パラメータとして登録できる値の制限については*サーバサイドJavaScriptを利用するときのパラメータの制限*を参照してください。

taskAccepted(event)

この関数の定義は任意です。

タスクキューの中で待機中だったタスクメッセージがタスクに変換され、実行可能状態となった時に呼び出されます。

この関数で例外が発生しても非同期処理機能は後続の処理を続行します。 この関数内で例外が発生しないような実装を推奨します。

この関数が呼び出されるタイミングの詳細については*タスク受付*を参照してください。

taskStarted(event)

この関数の定義は任意です。

実行可能状態となっていたタスクのビジネスロジックが開始される時に呼び出されます。

この関数で例外が発生しても非同期処理機能は後続の処理を続行します。 この関数内で例外が発生しないような実装を推奨します。

この関数が呼び出されるタイミングの詳細については*タスク処理開始*を参照してください。

taskCompleted(event)

この関数の定義は任意です。

実行中のタスクのビジネスロジックが完了された時に呼び出されます。

この関数で例外が発生しても非同期処理機能は後続の処理を続行します。 この関数内で例外が発生しないような実装を推奨します。

この関数が呼び出されるタイミングの詳細については*タスク処理完了*を参照してください。

taskRejected(event)

この関数の定義は任意です。

何らかの理由でタスクメッセージの受け付けが拒否され、実行可能状態にできなかった時に呼び出されます。

この関数で例外が発生しても非同期処理機能は後続の処理を続行します。 この関数内で例外が発生しないような実装を推奨します。

この関数が呼び出されるタイミングの詳細については*タスク受付拒否*を参照してください。

イベント

各種イベント発生時に呼び出される関数にはイベント情報が引数として渡されます。

イベント情報は以下の属性を持つオブジェクトです。

属性	内容
type	発生したイベントの種類
error	エラー発生の判定フラグ

type

発生したイベントの種類を取得します。

種類	内容
TASK_ACCEPTED	*taskAccepted(event)*が呼び出されたことを表します。
TASK_STARTED	*taskStarted(event)*が呼び出されたことを表します。
TASK_COMPLETED	*taskCompleted(event)*が呼び出されたことを表します。
TASK_REJECTED	*taskRejected(event)*が呼び出されたことを表します。

error

イベント発生時に例外が発生していたかどうかを取得します。

- 例外が発生していた場合はtrueが返されます。
- 例外が発生していなかった場合はfalseが返されます。

パラメータの制限

タスクメッセージをタスクキューに登録する時、ビジネスロジックの実行に必要となる独自のパラメータを渡し、タスクの実行時にそのパラメータを取得して利用することが可能です。 パラメータとして使用できる値には制限があります。

Javaを利用するときのパラメータの制限

登録時

Javaを利用してタスクメッセージをタスクキューに登録する場合、受け渡すパラメータには以下のような制限があります。

- パラメータはnullか、java.util.Mapを実装したクラスのインスタンスのみ指定できます。
- パラメータとして指定するマップのkey部はjava.lang.Stringのみです。
- パラメータとして指定するマップのkey部には必ず値が設定される必要があります。nullを指定することはできません。
- パラメータとして指定するマップのvalue部にはnullか、*パラメータとして使用できるインスタンス(Java)*に示す値のいずれかのみ指定できます。

パラメータとして使用できるインスタンス(Java)

クラス	備考
java.lang.Boolean	
java.lang.Byte	
java.lang.Short	
java.lang.Integer	
java.lang.Long	
java.lang.Float	
java.lang.Double	
java.lang.String	
java.util.Listの実装クラス	リストの要素として指定できる値は*パラメータとして使用できるインスタンス(Java)*で示されているもののいずれかです。 *TaskManager#addParallelizedTask*または*TaskManager#addSerializedTask*を通じてタスクメッセージを登録した時の値およびその順序のみが保存されます。 それ以外の情報（例：登録時に使用したjava.util.Listの実装クラス、等）は保証されません。
java.util.Mapの実装クラス	key に指定できる値はjava.lang.Stringのインスタンスのみです。nullは指定できません。 value に指定できる値はnullまたは*パラメータとして使用できるインスタンス(Java)*で示されているもののいずれかです。*TaskManager#addParallelizedTask*または*TaskManager#addSerializedTask*を通じてタスクメッセージを登録した時の key-value のマッピング情報のみが保存されます。 それ以外の情報（例：登録時に使用したjava.util.Mapの実装クラス、登録内容の順序、等）は保証されません。

> **注意**
> *パラメータとして使用できるインスタンス(Java)*に示す値を使用している場合であっても、ループ構造を含んではいけません。

取得時

登録時に設定したパラメータは*Task#setParameter*の引数で取得できますが、以下のような制限があります。

数値型のパラメータ

*数値型のパラメータ*に示す値はすべてjava.lang.Numberとして扱います。 設定した時の値を取得する場合、一度java.lang.Numberにダウンキャストし、該当するメソッド(*xxx*Value())を使用してください。

数値型のパラメータ

登録時のクラス	実際の値取得時に使用するメソッド
java.lang.Byte	java.lang.Number#byteValue()
java.lang.Short	java.lang.Number#shortValue()
java.lang.Integer	java.lang.Number#intValue()
java.lang.Long	java.lang.Number#longValue()
java.lang.Float	java.lang.Number#floatValue()
java.lang.Double	java.lang.Number#doubleValue()

java.util.List型のパラメータ

登録時にjava.util.List型を利用している場合、取得時にはjava.util.Listの実装クラスの情報は保持されません。

ダウンキャストする時は必ずjava.util.Listインタフェースとしてください。

java.util.Map型のパラメータ

登録時にjava.util.Map型を利用している場合、取得時にはjava.util.Mapの実装クラスの情報は保持されません。

ダウンキャストする時は必ずjava.util.Mapインタフェースとしてください。

なお、キーは登録時の制限と同様にjava.lang.Stringです。

パラメータの構造

登録時のパラメータの中に、あるインスタンスを複数の箇所から参照しているような構造があっても、取得時には異なるインスタンスを参照しているような構造になっている場合があります。

パラメータの参照先の値は保持されますが、参照先のインスタンスの同一性は保たれません。

サーバサイドJavaScriptを利用するときのパラメータの制限

登録時

サーバサイドJavaScriptを利用してタスクメッセージをタスクキューに登録する場合、受け渡すパラメータには以下のような制限があります。

- パラメータは連想配列のみ指定できます。
- パラメータとして指定する連想配列のkey部は文字列および整数のみ指定できます。 整数を指定した場合、文字列として扱われます。
- パラメータとして指定する連想配列のkey部には必ず値が設定される必要があります。nullを指定することはできません。
- パラメータとして指定する連想配列のvalue部にはnullか、*パラメータとして使用できるインスタンス(サーバサイドJavaScript)*に示す値のいずれかのみ指定できます。

パラメータとして使用できるインスタンス(サーバサイドJavaScript)

クラス	備考
Boolean	
数値型	
文字列	
配列	配列の要素として指定できる値は*パラメータとして使用できるインスタンス(サーバサイドJavaScript)*で示されているもののいずれかです。 *WorkManager#addParallelizedTask*または*WorkManager#addSerializedTask*を通じてタスクメッセージを登録した時の値およびその順序のみが保存されます。
連想配列	key に指定できる値は文字列および整数のみです。nullは指定できません。 value に指定できる値はnullまたは*パラメータとして使用できるインスタンス(サーバサイドJavaScript)*で示されているもののいずれかです。 *WorkManager#addParallelizedTask*または*WorkManager#addSerializedTask*を通じてタスクメッセージを登録した時の key-value のマッピング情報のみが保存されます。 それ以外の情報（例：登録内容の順序、等）は保証されません。

注意

*パラメータとして使用できるインスタンス(サーバサイドJavaScript)*に示す値を使用している場合であっても、ループ構造を含んではいけません。

取得時

登録時に設定したパラメータは*setParameter(parameter)*の引数で取得できますが、以下のような制限があります。

パラメータの構造

登録時のパラメータの中に、あるインスタンスを複数の箇所から参照しているような構造があっても、取得時には異なるインスタンスを参照しているような構造になっている場合があります。

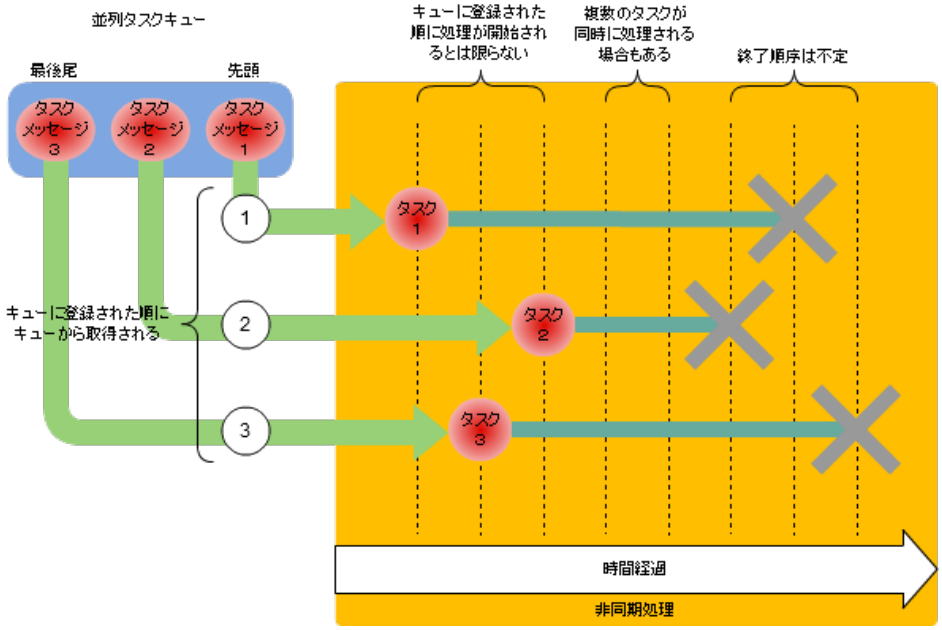
パラメータの参照先の値は保持されますが、参照先のインスタンスの同一性は保たれません。

並列処理機能

並列処理機能は複数の タスクを同時に処理することを可能とする機能です。

タスク間に関連がなく、個々のタスク内でビジネスロジックが完結する場合、並列処理機能を利用することで全体のレスポンスを向上させる可能性があります。

並列処理機能を利用する場合、タスク登録アプリケーションはタスクメッセージを並列タスクキューに登録する必要があります。



並列処理機能の概要

タスク

並列処理機能を利用する場合、タスク間には関連性がなく、複数同時実行が可能であることが必要です。 以下の条件をすべて満たす場合は並列処理機能で処理を行うことが可能です。

- 複数の任意のタスクが同時に実行されても実行結果に影響がない。
- タスク間の実行順序が予期できないものであっても実行結果に影響がない。

並列タスクキュー

並列タスクキューは並列処理機能を利用する場合にタスクメッセージを登録するタスクキューです。

並列タスクキューの詳細については*並列タスクキュー*を参照してください。

並列処理機能におけるタスクメッセージの登録

並列処理機能を利用する場合、タスク登録アプリケーションは非同期処理機能から提供されているAPIを通じてタスクメッセージを並列タスクキューに登録する必要があります。 APIはタスクメッセージを非同期処理機能に登録後、ビジネスロジックの実行を待たずに復帰します。

並列タスクキューに登録されたタスクメッセージは後にタスクに変換され、ビジネスロジック処理が開始されます。

> **コラム**
> タスクメッセージが並列処理機能に登録された時点ではまだ処理が開始されていない可能性があります。

タスクメッセージを登録すると、登録時のコンテキストの情報も保存されます。 タスクメッセージ登録時のコンテキスト情報は、タスクのビジネスロジック実行時に取得して利用することが可能です。

Java

Javaでタスクメッセージを登録する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのaddParallelizedTaskメソッドを呼び出します。 このメソッドを呼び出すことによってタスク登録アプリケーションはJavaを通じて並列タスクキューにタスクメッセージを追加します。

TaskManager#addParallelizedTask

```
public static jp.co.intra_mart.foundation.asynchronous.ParallelizedTaskMessage
  addParallelizedTask(
    java.lang.String taskClassName,
    java.util.Map<java.lang.String, ?> parameter
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

taskClassNameにはビジネスロジックを実行するタスクのクラス名を指定します。

parameterにはビジネスロジック実行時に受け渡したいパラメータを指定します。 パラメータとして指定できる値は*Javaを利用するときのパラメータの制限*の内容に準じます。 パラメータに無効な値が指定された場合、java.lang.IllegalArgumentExceptionが返されます。

このメソッドは戻り値としてjp.co.intra_mart.foundation.asynchronous.ParallelizedTaskMessageを返します。

サーバサイドJavaScript

サーバサイドJavaScriptでタスクメッセージを登録する場合、WorkManagerオブジェクトのaddParallelizedTask関数を呼び出します。 この関数を呼び出すことによってタスク登録アプリケーションはサーバサイドJavaScriptを通じて並列タスクキューにタスクメッセージを追加します。

WorkManager#addParallelizedTask

```
function addParallelizedTask(jsPath, parameter)
```

jsPathにはビジネスロジックとなるrun関数が定義されているJSソースファイルのパスを指定します。

parameterにはビジネスロジック実行時に受け渡したいパラメータを指定します。 パラメータとして指定できる値は*サーバサイドJavaScriptを利用するときのパラメータの制限*の内容に準じます。

このメソッドは戻り値としてメッセージIDを返します。

並列処理機能におけるタスクメッセージの削除

並列タスクキューにタスクメッセージが登録された後であっても、タスクメッセージが実際にタスクとして実行される前（*選択待機状態*を参照）であれば任意の時点でタスクメッセージを削除することができます。 タスクメッセージはAPIまたは非同期処理管理画面を通じて削除することが可能です。

Java

Javaでタスクメッセージを削除する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのremoveParallelizedTaskメソッドを呼び出します。 このメソッドを呼び出すことによってタスク管理アプリケーションはJavaを通じて並列タスクキューからビジネスロジック実行前のタスクメッセージを削除します。

TaskManager#removeParallelizedTask

```
public static boolean removeParallelizedTask(
    final String messageId
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskIllegalStateException,
    jp.co.intra_mart.foundation.asynchronous.InvalidTaskException,
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

messageIdには*TaskManager#addParallelizedTask*または*WorkManager#addParallelizedTask*の呼び出し時に取得されたメッセージIDを指定します。

messageIdが並列タスクキューに登録されたものでない場合、jp.co.intra_mart.foundation.asynchronous.InvalidTaskExceptionが返されます。

messageIdに該当するタスクメッセージが（実行中などの理由により）削除できなかった場合、jp.co.intra_mart.foundation.asynchronous.TaskIllegalStateExceptionが返されます。

Javascript API

サーバサイドJavaScriptでタスクメッセージを削除する場合、WorkManagerクラスのremoveParallelizedTaskメソッドを呼び出します。 このメソッドを呼び出すことによってタスク管理アプリケーションはサーバサイドJavaScriptを通じて並列タスクキューからビジネスロジック実行前のタスクメッセージを削除します。

WorkManager#removeParallelizedTask

```
function removeParallelizedTask(messageId)
```

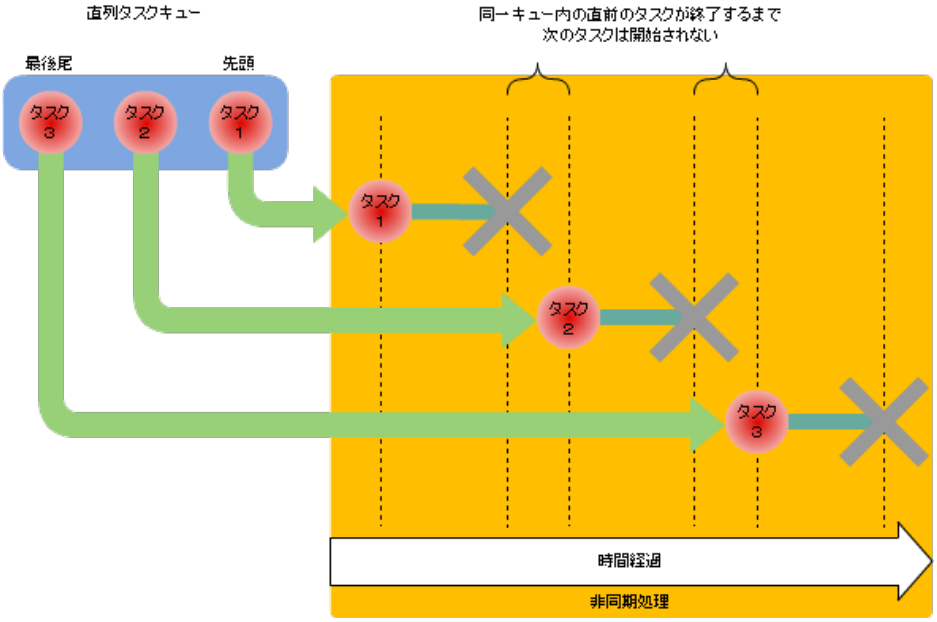
messageIdには*TaskManager#addParallelizedTask*または*WorkManager#addParallelizedTask*の呼び出し時に取得されたメッセージIDを指定します。

直列処理機能

直列処理機能は複数のタスクを逐次的に処理する機能です。

タスク厳密な順序関係があり、先行するタスクのビジネスロジックが完了するまで後続のタスクをブロックする必要がある場合、直列処理機能を利用することで処理順序を保証する必要があります。

直列処理機能を利用する場合、タスク登録アプリケーションはタスクメッセージを直列タスクキューに登録する必要があります。 *直列処理機能の概要*を参照してください。

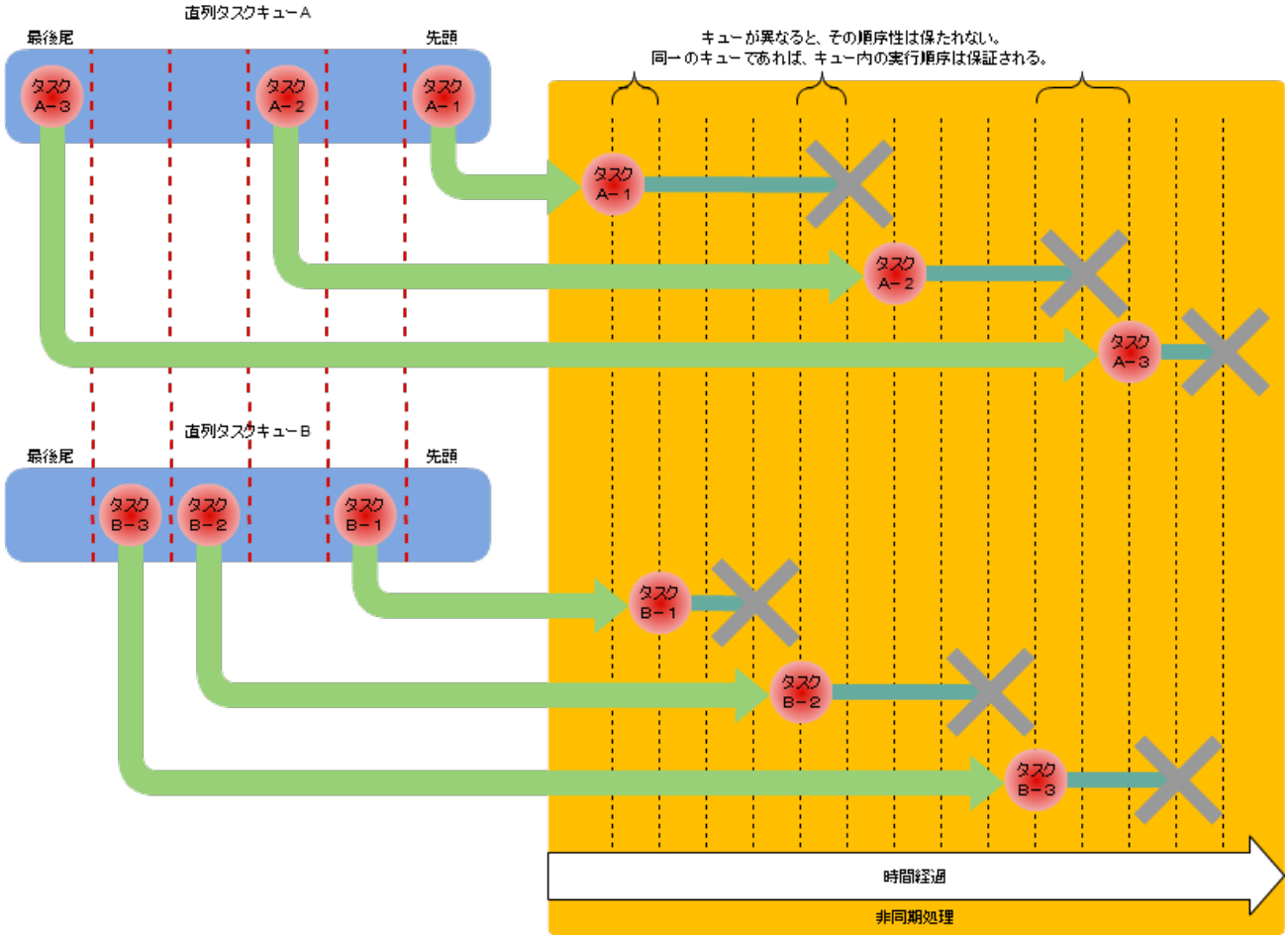


直列処理機能の概要

直列タスクキューは複数登録することができます。

タスクメッセージ間の実行順序の関係は同一の直列タスクキューにおいてのみ保たれます。

直列タスクキューが異なれば、タスクメッセージ間の実行順序は不定となります。 *複数の直列タスクキュー*を参照してください。



複数の直列タスクキュー

タスク

複数のタスク間の実行順序を正確にしたい場合、直列処理機能を利用する必要があります。

直列処理機能でビジネスロジック実行中のタスクに例外が発生した場合、後続のタスクの扱いを以下のいずれかに決定する必要があります。

- 該当するタスクメッセージが所属する直列タスクキューの逐次処理状態を現状のままとする。逐次処理可能状態であれば、後続のタスクを処理する。（*有効状態*または*無効状態*を参照）
- キューの処理を停止し、後続のタスクを待機状態にする。（*無効状態*を参照）

これらの決定はタスクメッセージの登録時に設定します。 登録するタスクメッセージ毎に直列タスクキューの逐次処理の継続または停止を設定することができます。

複数のタスクが存在し、以下の条件のいずれかが満たされる場合は直列処理機能で処理を行う必要があります。

- 複数のタスクが同時に実行されると実行結果に影響がある。
- 複数のタスクの実行順序が変更されると期待される実行結果にならない。

直列タスクキュー

直列タスクキューは直列処理機能を利用する場合にタスクメッセージを登録するタスクキューです。

直列タスクキューの詳細については*直列タスクキュー*を参照してください。

直列処理機能におけるタスクメッセージの登録

直列処理機能を利用する場合、タスク登録アプリケーションは非同期処理機能から提供されているAPIを通じてタスクメッセージを直列タスクキューに登録する必要があります。 APIはタスクメッセージを非同期処理機能に登録後、ビジネスロジックの実行を待たずに復帰

します。

直列処理機能に登録されたタスクメッセージは後でタスクに変換されてビジネスロジック処理が開始されます。

> **コラム**
> タスクメッセージが直列処理機能に登録された時点ではまだ処理が開始されていない可能性があります。

タスクメッセージを登録すると、登録時のコンテキストの情報も保存されます。 タスクメッセージ登録時のコンテキスト情報は、タスクのビジネスロジック実行時に取得して利用することが可能です。

Java

Javaでタスクメッセージを登録する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのaddSerializedTaskメソッドを呼び出します。 このメソッドを呼び出すことによってタスク登録アプリケーションはJavaを通じて直列タスクキューにタスクメッセージを追加します。

TaskManager#addSerializedTask

```
public static jp.co.intra_mart.foundation.asynchronous.SerializedTaskMessage
  addSerializedTask(
    java.lang.String queueId,
    java.lang.String taskClassName,
    java.util.Map<java.lang.String, ?> parameter,
    boolean stopProgressOnError
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

queueIdにはタスクメッセージの登録先となる直列タスクキューのキューIDを指定します。

taskClassNameにはビジネスロジックを実行するタスクのクラス名を指定します。

parameterにはビジネスロジック実行時に受け渡したいパラメータを指定します。 パラメータとして指定できる値は*Javaを利用するときのパラメータの制限*の内容に準じます。 パラメータに無効な値が指定された場合、java.lang.IllegalArgumentExceptionが返されます。

stopProgressOnErrorにはtaskClassNameで指定したタスクの実行時に例外が発生した場合、対象となる直列タスクキューを停止するかどうかを指定します。

- trueが設定された場合、例外発生時には直列タスクキューを現在の状態に関わらず停止状態にし、後続のタスクの処理を停止します。
- falseが設定された場合、例外発生時であっても直列タスクキューの状態を変更しません。

このメソッドは戻り値としてjp.co.intra_mart.foundation.asynchronous.SerializedTaskMessageを返します。

サーバサイドJavaScript

サーバサイドJavaScriptでタスクメッセージを登録する場合、WorkManagerオブジェクトのaddSerializedTask関数を呼び出します。この関数を呼び出すことによってタスク登録アプリケーションはサーバサイドJavaScriptを通じて直列タスクキューにタスクメッセージを追加します。

WorkManager#addSerializedTask

```
function addSerializedTask(queueId, jsPath, parameter, stopQueueProcessingIfError)
```

queueIdにはタスクメッセージの登録先となる直列タスクキューのキューIDを指定します。

jsPathにはビジネスロジックとなるrun関数が定義されているJSソースファイルのパスを指定します。

parameterにはビジネスロジック実行時に受け渡したいパラメータを指定します。 パラメータとして指定できる値は*サーバサイドJavaScriptを利用するときのパラメータの制限*の内容に準じます。

stopProgressOnErrorにはjsPath で指定したタスクの実行時に例外が発生した場合、対象となる直列タスクキューを停止するかどうか

を指定します。

- trueが設定された場合、例外発生時には直列タスクキューを現在の状態に関わらず停止状態にし、後続のタスクの処理を停止します。
- falseが設定された場合、例外発生時であっても直列タスクキューの状態を変更しません。

このメソッドは戻り値としてメッセージIDを返します。

直列処理機能におけるタスクメッセージの削除

直列タスクキューにタスクメッセージが登録された後であっても、タスクメッセージが実際にタスクとして実行される前であれば任意の時点でタスクメッセージを削除することができます。 タスクメッセージはAPIまたは非同期処理管理画面を通じて削除することが可能です。 直列処理機能を利用する場合、タスク登録アプリケーションは非同期処理機能から提供されているAPIを通じてタスクメッセージを直列タスクキューに登録する必要があります。 APIはタスクメッセージを非同期処理機能に登録後、ビジネスロジックの実行を待たずに復帰します。

Java

Javaでタスクメッセージを削除する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのremoveSerializedTaskメソッドを呼び出します。 このメソッドを呼び出すことによってタスク管理アプリケーションはJavaを通じて直列タスクキューからビジネスロジック実行前のタスクメッセージを削除します。

TaskManager#removeSerializedTask

```
public static boolean removeSerializedTask(
    java.lang.String messageId
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskIllegalStateException,
    jp.co.intra_mart.foundation.asynchronous.InvalidTaskException,
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

messageIdには*TaskManager#addSerializedTask*または*WorkManager#addSerializedTask*の呼び出し時に取得されたメッセージIDを指定します。

messageIdが直列タスクキューに登録されたものでない場合、jp.co.intra_mart.foundation.asynchronous.InvalidTaskExceptionが返されます。

messageIdに該当するタスクメッセージが（実行中などの理由により）削除できなかった場合、jp.co.intra_mart.foundation.asynchronous.TaskIllegalStateExceptionが返されます。

サーバサイドJavaScript

サーバサイドJavaScriptでタスクメッセージを削除する場合、WorkManagerクラスのremoveSerializedTaskメソッドを呼び出します。 このメソッドを呼び出すことによってタスク管理アプリケーションはサーバサイドJavaScriptを通じて直列タスクキューからビジネスロジック実行前のタスクメッセージを削除します。

WorkManager#removeSerializedTask

```
function removeSerializedTask(messageId)
```

messageIdには*TaskManager#addSerializedTask*または*WorkManager#addSerializedTask*の呼び出し時に取得されたメッセージIDを指定します。

タスク処理

*並列処理機能*または *直列処理機能*で登録されたタスクメッセージは、いずれタスクに変換されてビジネスロジックが実行されます。

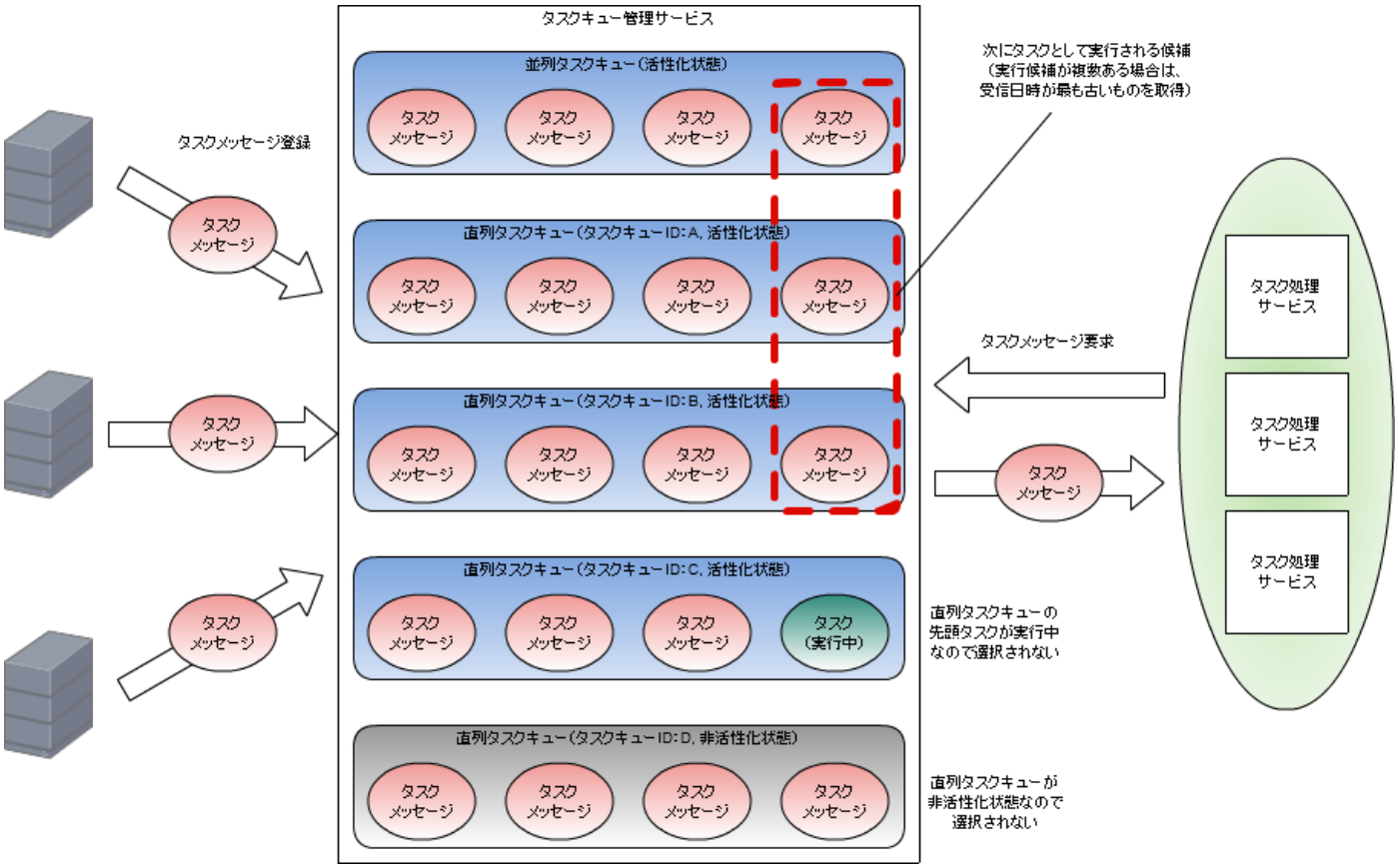
実行可能タスク取得

タスク実行エンジンはタスク処理サービスを通じて定期的に並列タスクキューおよび直列タスクキューの状態を監視しています。 以下の状態をすべて満たす場合、タスク実行エンジンはタスクメッセージを実行可能と判断し、タスクキューの先頭から取得します。

- タスク実行エンジンでは新しいタスクメッセージを受け付けてタスクを実行する余裕がある。
- タスクキューに以下のいずれかのタスクメッセージが存在する。
  - 並列タスクキューの先頭に*選択待機状態*となっているタスクメッセージが登録されている。
    （ただし、並列タスクキューが有効状態であることが前提）
  - 直列タスクキューの先頭に*選択待機状態*となっているタスクメッセージが登録されていて、同タスクキュー内で以下の状態に該当するタスクが存在しない。
    （ただし、直列タスクキューが有効状態であることが前提）
    - *初期状態*
    - *受付待機状態*
    - *処理実行可能状態*
    - *処理実行中状態*

上記の条件に一致するタスクメッセージが複数ある場合、タスクメッセージの受信日時が一番古いものが優先されます。 受信日時が同一のタスクメッセージが複数存在する場合、優先されるタスクメッセージは不定です。

この内容を*実行可能タスク取得*にも示します。



実行可能タスク取得

実行可能として判断され、タスク実行エンジンから取得されたタスクメッセージは*処理実行中状態*として扱われます。

タスク受付

タスク実行エンジンはタスクメッセージを取得すると、自身が「受け付けた」という情報（実行ノード）を該当タスクメッセージに更新します。 この時、受付日時もタスクメッセージに対して更新されます。

受付状態になった場合、イベントが発生します。 詳細は*タスク受付*を参照してください。

タスク処理開始

タスク実行エンジンは受け付けたタスクメッセージをタスクに変換しビジネスロジックを非同期で開始します。

- *Taskインタフェース*を実装したタスクを登録している場合、run()メソッドが非同期で起動されます。
- サーバサイドJavaScriptを利用してタスクメッセージを登録した場合、該当するJSファイルのrun()関数が呼ばれます。

ビジネスロジック内では、タスクメッセージ登録時のコンテキストを取得することが可能です。

タスクが開始される直前にイベントが発生します。 詳細は*タスク処理開始*を参照してください。

> **コラム**
>
> 複数のタスクが同時に存在する場合、各タスク間の開始順序は受付順とは異なる場合があります。
>
> 開始順序を明確にしたい場合は、直列タスクキューを利用してください。

タスク処理完了

以下のいずれかの状態になった場合、タスク実行エンジンはタスクのビジネスロジックが完了したと判断します。

- *Taskインタフェース*を実装したクラスのインスタンスのrun()メソッドが終了した場合（例外発生時も含む）
- JSファイルのrun(parameter)関数が終了した場合（例外発生時も含む）

ただし、いずれの場合も*タスク終了通知*および*タスク強制中断*によって終了した場合は上記の理由に含まれません。

ビジネスロジックが完了した場合、タスク実行エンジンは以下のような事後処理を行います。

1. タスクキューから該当するタスクメッセージを削除します。
2. タスクに対してイベントを通知します。 詳細は*タスク処理完了*を参照してください。

また、タスクメッセージが直列タスクキューに登録されたものであり、以下の条件をすべて満たす場合、該当する直列タスクキューの逐次処理は現在の状態に関わらず停止されます。

- 登録時に、例外発生時は直列タスクキューの逐次処理を停止するように指定した。
- タスク実行時に例外が発生した。

タスク受付拒否

*タスク受付*でタスクメッセージを受け付けようとした時、何らかの理由でタスク実行エンジンは受付を拒否する場合があります。

この場合、タスク実行エンジンは以下のような処理を行います。

- 該当するタスクメッセージを未処理の状態に戻し、登録されていたタスクキューの先頭に再追加します。
- タスクに対してイベントを通知します。 詳細は*タスク受付拒否*を参照してください。

なお、上記の処理は順不定です。

タスク終了通知

タスクのビジネスロジック続行中に、何らかの理由でタスクを終了するよう通知がされることがあります。

詳細は*タスク終了通知*を参照してください。

タスク強制中断

タスクのビジネスロジック続行中に、何らかの理由でタスクを強制停止するために、ビジネスロジックを実行しているスレッドに対してThread.stop()を呼び出す場合があります。

詳細は*タスク強制中断*を参照してください。

キュー管理

タスクキューはタスクメッセージの集合であり、先入れ先出し方式(FIFO: First In, First Out)で管理されています。

- 登録時、タスクメッセージはタスクキューの最後尾に追加されます。
- タスクキューに登録されたタスクメッセージは先頭のものから順次取得されます。

タスク管理アプリケーションは各種APIを通じてタスクキューを管理することができます。 タスクキューの管理には以下のような内容が含まれます。

- タスクキューの停止または起動
- タスクキューの追加または削除（直列タスクキューの場合のみ）

並列タスクキュー

並列タスクキューは並列処理機能を利用する場合にタスクメッセージを登録するタスクキューです。

並列タスクキューの数

並列タスクキューは非同期処理機能からテナント内に一つのみ提供されます。

並列タスクキューを追加したり削除することは出来ません。

タスクメッセージが処理実行可能となる条件

タスク処理サービスは並列タスクキューの処理状態が*有効状態*である場合のみ、先頭に存在するタスクメッセージを任意のタイミングで取得することが可能です。 並列タスクキューの処理状態については*並列タスクキューの処理状態*を参照してください。

並列タスクキューの処理状態

並列タスクキューは以下のいずれかの状態にあります。

- *有効状態*
  この状態にある場合、並列タスクキューに関連するすべてのAPIが利用可能です。
- *無効状態*
  この状態にある場合、タスク処理サービスは、並列タスクキューから実行可能な状態にあるタスクメッセージを取り出しません。

タスク登録アプリケーションまたはタスク管理アプリケーションは、任意の時点でこの状態を変更することが可能です。

Java

Javaで並列タスクキューの状態を変更する場合、`jp.co.intra_mart.foundation.asynchronous.TaskManager`クラスの`setParallelizedTaskQueueActive`メソッドを呼び出します。 このメソッドを呼び出すことによってタスク登録アプリケーションはJavaを通じて並列タスクキューの状態を変更します。

TaskManager#setParallelizedTaskQueueActive

```
public static void setParallelizedTaskQueueActive(
    boolean active
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

`active`には並列タスクキューの状態を指定します。

- `true`
  並列タスクキューを*有効状態*にします。
- `false`
  並列タスクキューを*無効状態*にします。

サーバサイドJavaScript

サーバサイドJavaScriptで並列タスクキューの状態を変更する場合、WorkManagerオブジェクトのsetParallelizedTaskQueueActive関数を呼び出します。 この関数を呼び出すことによってタスク登録アプリケーションはサーバサイドJavaScriptを通じて並列タスクキューの状態を変更します。

WorkManager#setParallelizedTaskQueueActive

```
function setParallelizedTaskQueueActive(active)
```

activeには並列タスクキューの状態を指定します。

- true
  並列タスクキューを*有効状態*にします。
- false
  並列タスクキューを*無効状態*にします。

直列タスクキュー

直列タスクキューは直列処理機能を利用する場合にタスクメッセージを登録するタスクキューです。

直列タスクキューの追加

直列タスクキューは初期状態では存在しません。 APIを通じて明示的に登録する必要があります。

直列タスクキューは複数登録することが可能であり、キューIDによって識別されます。 キューIDの命名規約は特にありませんが、Javaのパッケージ名と同様にドメイン名を逆順にしたものからはじめることを推奨します。

Java

Javaで直列タスクキューを追加する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのsetParallelizedTaskQueueActiveメソッドを呼び出します。 このメソッドを呼び出すことによってタスク登録アプリケーションはJavaを通じて並列タスクキューを追加します。

TaskManager#addSerializedTaskQueue

```
public static boolean addSerializedTaskQueue(
    java.lang.String queueId, boolean active
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

queueIdには直列タスクキューを識別するためのキューIDを指定します。

- キューIDが重複する直列タスクキューを登録することはできません。
- 既に登録済みのキューIDを指定すると、戻り値としてfalseが返されます。 この場合、既存の直列タスクキューには何も反映されません。

activeには直列タスクキューの初期状態を指定します。

- true
  直列タスクキューを*有効状態*で初期登録します。
- false
  直列タスクキューを*無効状態*で初期登録します。

このメソッドは戻り値として直列タスクキューの追加が成功したかどうかをboolean型で返します。

- true
  新しい直列タスクキューが追加された場合、この値が返されます。
- false
  システム上問題がないにも関わらず、新しい直列タスクキューが登録できなかった場合（例：キューIDが重複していた場合な

ど）、戻り値としてこの値が返されます。

サーバサイドJavaScript

サーバサイドJavaScriptで直列タスクキューを追加する場合、WorkManagerクラスのaddSerializedTaskQueueメソッドを呼び出します。 この関数を呼び出すことによってタスク登録アプリケーションはJavaを通じて並列タスクキューを追加します。

WorkManager#addSerializedTaskQueue

```
function addSerializedTaskQueue(queueId, active)
```

queueIdには直列タスクキューを識別するためのキューIDを指定します。

- キューIDが重複する直列タスクキューを登録することはできません。
- 既に登録済みのキューIDを指定すると、戻り値としてfalseが返されます。 この場合、既存の直列タスクキューには何も反映されません。

activeには直列タスクキューの初期状態を指定します。

- true
  直列タスクキューを*有効状態*で初期登録します。
- false
  直列タスクキューを*無効状態*で初期登録します。

このメソッドは戻り値として直列タスクキューの追加が成功したかどうかをboolean型で返します。

- true
  新しい直列タスクキューが追加された場合、この値が返されます。
- false
  システム上問題がないにも関わらず、新しい直列タスクキューが登録できなかった場合（例：キューIDが重複していた場合など）、戻り値としてこの値が返されます。

直列タスクキューの削除

登録済みの直列タスクキューは削除することが可能です。

直列タスクキューを削除するためには、該当する直列タスクキューが以下の条件をすべて満たしているいる必要があります。

- 現在処理中のタスクが存在しない
- タスクメッセージが存在しない

Java

Javaで直列タスクキューを削除する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのremoveSerializedTaskQueueメソッドを呼び出します。 このメソッドを呼び出すことによってタスク登録アプリケーションはJavaを通じて並列タスクキューを削除します。

TaskManager#removeSerializedTaskQueue

```
public static boolean removeSerializedTaskQueue(
    java.lang.String queueId
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskQueueIllegalStateException,
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

queueIdには直列タスクキューを識別するためのキューIDを指定します。 存在しないキューIDを指定すると、戻り値としてfalseが返されます。

このメソッドは戻り値として直列タスクキューの削除が成功したかどうかをboolean型で返します。

- true

指定された直列タスクキューが削除された場合、この値が返されます。

- false
  システム上問題がないにも関わらず、指定された直列タスクキューが削除できなかった場合（例：キューIDで指定した直列タスクキューが存在しない場合など）、戻り値としてこの値が返されます。

キューIDで指定した直列タスクキューに現在処理中のタスクやタスクメッセージが存在する場合、このメソッドはjp.co.intra_mart.foundation.asynchronous.TaskQueueIllegalStateExceptionを返却します。

サーバサイドJavaScript

サーバサイドJavaScriptで直列タスクキューを削除する場合、WorkManagerクラスのremoveSerializedTaskQueueメソッドを呼び出します。 この関数を呼び出すことによってタスク登録アプリケーションはJavaを通じて並列タスクキューを削除します。

WorkManager#removeSerializedTaskQueue

```
function removeSerializedTaskQueue(queueId)
```

queueIdには直列タスクキューを識別するためのキューIDを指定します。 存在しないキューIDを指定すると、戻り値としてfalseが返されます。

このメソッドは戻り値として直列タスクキューの削除が成功したかどうかをboolean型で返します。

- true
  指定された直列タスクキューが削除された場合、この値が返されます。
- false
  システム上問題がないにも関わらず、指定された直列タスクキューが削除できなかった場合（例：キューIDで指定した直列タスクキューが存在しない場合など）、戻り値としてこの値が返されます。

キューIDで指定した直列タスクキューに現在処理中のタスクやタスクメッセージが存在する場合、この関数の戻り値のerror属性にfalseが設定されます。

タスクメッセージが処理実行可能となる条件

タスク処理サービスは直列タスクキューの先頭に存在するタスクメッセージが処理中でない場合のみ取得することが可能です。 タスク処理サービスは直列タスクキューが以下の条件をすべて満たす場合のみ先頭に存在するタスクメッセージを取得することが可能です。

- 直列タスクキューの処理状態が*有効状態*であること
- 直列タスクキューの先頭に存在するタスクメッセージがどのタスク実行エンジンからも実行されていないこと

直列タスクキューの処理状態については*直列タスクキューの処理状態*を参照してください。

直列タスクキューの処理状態

直列タスクキューの処理状態は以下のいずれかです。

- *有効状態*
  この状態にある場合、該当する直列タスクキューに関連するすべてのAPIが利用可能です。
- *無効状態*
  この状態にある場合、タスク処理サービスは、該当する直列タスクキューから実行可能な状態にあるタスクメッセージを取り出しません。

タスク登録アプリケーションまたはタスク管理アプリケーションは、任意の時点でこの状態を変更することが可能です。

Java

Javaで直列タスクキューの状態を変更する場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスのsetSerializedTaskQueueActiveメソッドを呼び出します。 このメソッドを呼び出すことによってタスク登録アプリケーションはJavaを通じて直列タスクキューの状態を変更します。

TaskManager#setSerializedTaskQueueActive

```
public static void setSerializedTaskQueueActive(
    java.lang.String queueId,
    boolean active
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

queueIdには直列タスクキューを識別するためのキューIDを指定します。

activeには直列タスクキューの状態を指定します。

- true
  直列タスクキューを*有効状態*にします。
- false
  直列タスクキューを*無効状態*にします。

サーバサイドJavaScript

サーバサイドJavaScriptで直列タスクキューの状態を変更する場合、WorkManagerオブジェクトのsetSerializedTaskQueueActive関数を呼び出します。 この関数を呼び出すことによってタスク登録アプリケーションはサーバサイドJavaScriptを通じて直列タスクキューの状態を変更します。

WorkManager#setSerializedTaskQueueActive

```
function setSerializedTaskQueueActive(queueId, active)
```

queueIdには直列タスクキューを識別するためのキューIDを指定します。

activeには直列タスクキューの状態を指定します。

- true
  直列タスクキューを*有効状態*にします。
- false
  直列タスクキューを*無効状態*にします。

終了通知と強制停止

一度タスクメッセージがタスクに変換されビジネスロジックが開始されると、通常は非同期処理機能はそのビジネスロジックが完了するまではタスクに対して何も干渉しません。

しかしながら、何らかの理由で実行中のタスクのビジネスロジックを中断したい場合、以下の方法があります。

- 終了通知
- 強制停止（**非推奨**）

> **コラム**
> 他にもタスクの開発者が独自の終了方法を実装することも考えられますが、本書では説明の範囲外であるため割愛します。

終了通知と強制停止の違い

終了通知と強制停止は、いずれもタスクのビジネスロジックの実行途中で処理を中断させることを意図したものですが、*終了通知と強制停止の違い*に示すような違いがあります。

終了通知と強制停止の違い

比較項目	終了通知	強制停止
挙動	該当するタスクの Taskインタフェースのrelease()メソッドが非同期処理機能から呼び出されます。	該当するタスクのビジネスロジックを実行しているスレッドに対してjava.lang.Thread#stop()が非同期処理機能から呼び出されます。
開発者の責務	該当するタスクの開発者は Taskインタフェースのrelease()メソッドを適切に実装する必要があります。	該当するタスクの開発者は、ビジネスロジック中のいかなる時点でjava.lang.Thread#stop()が呼び出されても問題がないような実装をする必要があります。

終了通知

終了通知ではタスク処理サービスから該当するタスクのrelease()メソッドを呼び出します。

実際に実行中のタスクを停止するのは開発者の責務です。 開発者はrelease()メソッドが呼び出されたら、run()メソッドで実行中のビジネスロジックを安全かつ速やかに停止するような実装をしてください。

また、タスク処理サービスからrelease()メソッドが呼ばれたタスクは非同期処理機能の管理対象外です。

release()については*Task#release*を参照してください。

> **注意**
> タスクのビジネスロジックが終了しない状態でrelease()メソッドからタスク処理サービスに処理が戻された場合、後続のタスクの処理が開始されてしまう場合があります。 つまり、先に実行中のタスクのビジネスロジックと後続のタスクの処理が同時に実行されてしまう可能性があります。
>
> このような状況を避けたい場合、後続のタスクが実行されても問題がない状態になるまで、先に実行中のタスクのrelease()メソッドの中で待機する、などの工夫が必要です。

> **注意**
> 該当するタスクのrelease()メソッドに対して空の実装をする、または呼び出してもビジネスロジックの実行を終了しなくても終了通知を行うことは可能です。 ただし、この場合であっても、タスク処理サービスからrelease()メソッドが呼ばれたタスクは非同期処理機能の管理対象外です。
>
> この状態で該当するタスクメッセージを再登録するような指定がされていた場合、同一の情報を持ちながら、最初に作成されたタスクとは異なるインスタンスが再作成され、ビジネスロジックが重複して実行される可能性があります。

Java

Javaで並列処理機能のタスクに対して終了通知をする場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスの以下のメソッドを呼び出します。

- releaseRunningParallelizedTask
- releaseRunningSerializedTask

これらのメソッドを呼び出すことによってタスク処理サービスはJavaを通じてタスクのrelease()メソッドを呼び出します。

TaskManager#releaseRunning*XXXX*Task

```
public static jp.co.intra_mart.foundation.asynchronous.ParallelizedTaskMessage
  releaseRunningParallelizedTask(
    java.lang.String messageId,
    boolean reentry,
    boolean stop
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException

public static jp.co.intra_mart.foundation.asynchronous.SerializedTaskMessage
  releaseRunningSerializedTask(
    java.lang.String messageId,
    boolean reentry,
    boolean stop
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

- releaseRunningParallelizedTaskは、対象となるタスクが並列タスクキューに登録されている場合に使用します。
- releaseRunningSerializedTaskは、対象となるタスクが直列タスクキューに登録されている場合に使用します。

messageIdには終了通知の対象となるタスクのメッセージIDを指定します。 メッセージIDはタスクメッセージの登録時（*並列処理機能におけるタスクメッセージの登録*または*並列処理機能におけるタスクメッセージの登録*を参照してください）に取得した値を使用してください。

reentryには終了通知が完了した後にタスクメッセージを再登録するかどうかを指定します。

- true：タスクメッセージを再登録します。
- false：タスクメッセージを再登録しません。

タスクメッセージの再登録の詳細については*タスクメッセージの再登録*を参照してください。

stopには終了通知が完了した後に並列タスクキューを停止するかどうかを指定します。

- true：並列タスクキューを現在の処理状態に関わらず停止します。
- false：並列タスクキューの現在の処理状態を変更しません。

並列タスクキューの停止の詳細については*タスクキューの停止*を参照してください。

このメソッドは戻り値としてjp.co.intra_mart.foundation.asynchronous.ParallelizedTaskMessageを返します。

サーバサイドJavaScript

サーバサイドJavaScriptで並列処理機能のタスクに対して終了通知をする場合、 WorkManagerオブジェクトのreleaseRunningParallelizedTask関数を呼び出します。 この関数を呼び出すことによってタスク処理サービスはサーバサイドJavaScriptを通じてタスクのrelease()メソッドを呼び出します。

WorkManager#releaseRunningParallelizedTask

```
function releaseRunningParallelizedTask(messageId, reentry, stop)
```

messageIdには終了通知の対象となるタスクのメッセージIDを指定します。 メッセージIDはタスクメッセージの登録時（*並列処理機能におけるタスクメッセージの登録*または*並列処理機能におけるタスクメッセージの登録*を参照してください）に取得した値を使用してください。

reentryには終了通知が完了した後にタスクメッセージを再登録するかどうかを指定します。

- true：タスクメッセージを再登録します。
- false：タスクメッセージを再登録しません。

タスクメッセージの再登録の詳細については*タスクメッセージの再登録*を参照してください。

stopには終了通知が完了した後に並列タスクキューを停止するかどうかを指定します。

- true：並列タスクキューを現在の処理状態に関わらず停止します。
- false：並列タスクキューの現在の処理状態を変更しません。

並列タスクキューの停止の詳細については*タスクキューの停止*を参照してください。

このメソッドは戻り値としてメッセージIDを含む情報を返します。

強制停止

> 危険
>
> この方法でタスクを停止することは非推奨です。 java.lang.Thread#stop()はビジネスロジック中のいかなる場所でも処理を中断して抜ける可能性があり、リソースやロックの解放等が想定通りに行われない可能性があります。
>
> いかなる理由であっても、この方法を利用してタスクを停止し、何らかの被害が出た場合でも補償いたしかねますので注意してください。
>
> タスクを中断する場合は*終了通知*に示す方法を推奨します。

強制停止ではタスク処理サービスから該当するタスクのビジネスロジックを実行しているスレッドに対してjava.lang.Thread#stop()を呼び出します。

また、タスク処理サービスを通じて強制停止されたタスクは非同期処理機能の管理対象外です。

Java

Javaで並列処理機能のタスクに対して強制停止をする場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerクラスの以下のメソッドを呼び出します。

- stopRunningParallelizedTask
- stopRunningSerializedTask

これらのメソッドを呼び出すことによってタスク処理サービスはJavaを通じてタスクのビジネスロジックを実行しているスレッドに対してjava.lang.Thread#stop()を呼び出します。

TaskManager#stopRunning*XXXX*Task

```
public static jp.co.intra_mart.foundation.asynchronous.ParallelizedTaskMessage
  stopRunningParallelizedTask(
    java.lang.String messageId,
    boolean reentry,
    boolean stop
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException

public static SerializedTaskMessage stopRunningSerializedTask(
    java.lang.String messageId,
    boolean reentry,
    boolean stop
  ) throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

- stopRunningParallelizedTaskは、対象となるタスクが並列タスクキューに登録されている場合に使用します。

- stopRunningSerializedTaskは、対象となるタスクが直列タスクキューに登録されている場合に使用します。

messageIdには強制停止の対象となるタスクのメッセージIDを指定します。 メッセージIDはタスクメッセージの登録時（*並列処理機能におけるタスクメッセージの登録*または*並列処理機能におけるタスクメッセージの登録*を参照してください）に取得した値を使用してください。

reentryには強制停止が完了した後にタスクメッセージを再登録するかどうかを指定します。

- true：タスクメッセージを再登録します。
- false：タスクメッセージを再登録しません。

タスクメッセージの再登録の詳細については*タスクメッセージの再登録*を参照してください。

stopには強制停止が完了した後に並列タスクキューを停止するかどうかを指定します。

- true：並列タスクキューを現在の処理状態に関わらず停止します。
- false：並列タスクキューの現在の処理状態を変更しません。

並列タスクキューの停止の詳細については*タスクキューの停止*を参照してください。

このメソッドは戻り値としてjp.co.intra_mart.foundation.asynchronous.ParallelizedTaskMessageを返します。

サーバサイドJavaScript

サーバサイドJavaScriptで並列処理機能のタスクに対して\ 強制停止をする場合、 WorkManagerオブジェクトのreleaseRunningParallelizedTask関数を呼び出します。 この関数を呼び出すことによってタスク処理サービスはサーバサイドJavaScriptを通じてタスクのrelease()メソッドを呼び出します。

WorkManager#releaseRunningParallelizedTask

```
function releaseRunningParallelizedTask(messageId, reentry, stop)
```

messageIdには強制停止の対象となるタスクのメッセージIDを指定します。 メッセージIDはタスクメッセージの登録時（*並列処理機能におけるタスクメッセージの登録*または*並列処理機能におけるタスクメッセージの登録*を参照してください）に取得した値を使用してください。

reentryには強制停止が完了した後にタスクメッセージを再登録するかどうかを指定します。

- true：タスクメッセージを再登録します。
- false：タスクメッセージを再登録しません。

タスクメッセージの再登録の詳細については*タスクメッセージの再登録*を参照してください。

stopには強制停止が完了した後に並列タスクキューを停止するかどうかを指定します。

- true：並列タスクキューを現在の処理状態に関わらず停止します。
- false：並列タスクキューの現在の処理状態を変更しません。

並列タスクキューの停止の詳細については*タスクキューの停止*を参照してください。

このメソッドは戻り値としてメッセージIDを含む情報を返します。

タスクメッセージの再登録

終了通知または強制停止をされたタスクメッセージはタスクキューに再登録することが可能です。 再登録をする場合、タスクキューの先頭にタスクメッセージが追加されます。

再登録された場合でも、即時にビジネスロジックが開始されるわけではありません。 他のタスクメッセージと同様のものとして扱われます。 *タスク処理*を参照してください。

終了通知または強制停止をされたタスクメッセージが再登録された場合、以下の情報はそのまま引き継がれます。

- メッセージID
- 送信日時

- 受信日時
- パラメータ

タスクキューの停止

タスクが終了通知または強制停止をされた場合、対応するタスクキューの処理状態は以下のいずれかにすることができます。

- 現在の処理状態を継続する
- 現在の処理状態に関わらず、停止する

タスクキューの処理状態の詳細については*並列タスクキューの処理状態*または*直列タスクキューの処理状態*を参照してください。

コラム

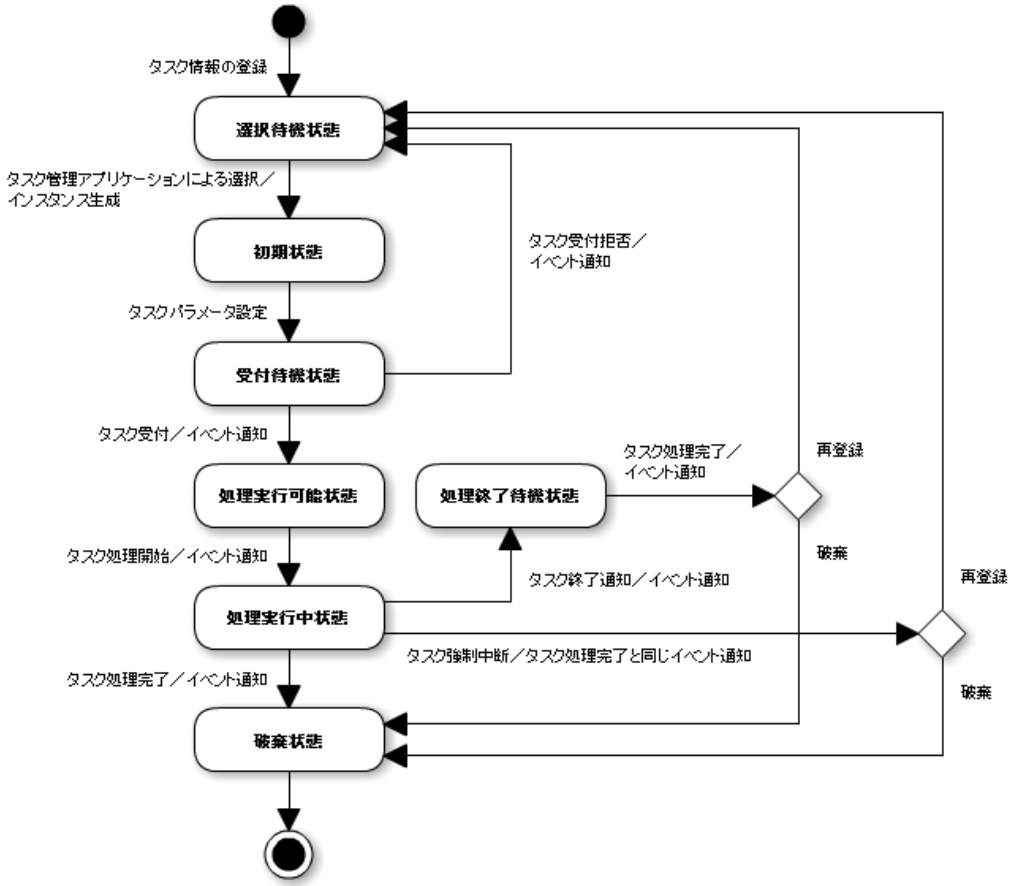
- 並列タスクの場合、並列タスクキュー全体が停止します。
- 直列タスクの場合、対応する直列タスクキューのみにこのルールが適用されます。他の直列タスクキューには影響ありません。

ライフサイクル

非同期処理機能には様々な状態やイベントが存在します。

本章では非同期処理機能における生成や破棄を含んだ様々なイベントとその発生タイミングを説明します。

タスク



タスクの状態遷移図

状態

選択待機状態

タスクキューの中でタスク管理アプリケーションによる選択を待機している状態です。

この時点ではタスクのビジネスロジックは実行されていません。

初期状態

タスク管理アプリケーションによってタスクキューに登録されているタスクメッセージが選択され、タスクに変換された状態です。

この時点ではタスクのビジネスロジックは実行されていません。

受付待機状態

タスクに実行時のパラメータが設定され、タスク実行エンジンによって受け付けられることを待機中である状態です。

この時点ではタスクのビジネスロジックは実行されていません。

処理実行可能状態

タスクがタスク実行エンジンによって受け付けられ、ビジネスロジックの開始を待機している状態です。

この時点ではタスクのビジネスロジックは実行されていません。

処理実行中状態

タスクのビジネスロジックが実行されている状態です。

処理終了待機状態

タスクのビジネスロジックの完了を待機している状態です。

破棄状態

タスクメッセージがタスクキューに登録されていない、または削除されている状態です。

イベント

タスクメッセージの登録

タスク登録アプリケーションによってタスクキューにタスクメッセージが取得されると、該当するタスクは*選択待機状態*です。

> **コラム**
>
> タスクメッセージの登録方法については、以下も参照してください。
>
> - Java
>   - *TaskManager#addParallelizedTask*
>   - *TaskManager#addSerializedTask*
> - サーバサイドJavaScript
>   - *WorkManager#addParallelizedTask*
>   - *WorkManager#addSerializedTask*

タスク管理アプリケーションによる選択

タスク管理アプリケーションによってタスクキューからタスクメッセージが取得されると、タスクのインスタンスが生成されます。

インスタンスが生成された直後のタスクは*初期状態*です。

> **コラム**
>
> Javaで*Taskインタフェース*を実装したタスクのコンストラクタがこのタイミングで呼び出されます。

タスクパラメータ設定

初期状態において、タスクメッセージ登録時に指定したパラメータが生成されたタスクに対して設定されます。

パラメータが設定されたタスクは*受付待機状態*です。

> **コラム**
>
> Javaで*Taskインタフェース*を実装したタスクを登録した場合、*Task#setParameter*メソッドが呼び出されます。
>
> サーバサイドJavaScriptでタスクを登録した場合、指定されたJSファイルの*setParameter(parameter)*関数が呼び出されます（同関数が定義されている場合）。

タスク受付

受付待機状態となっているタスクが非同期処理機能によって受け付けられ、処理を実行する準備ができると、該当するイベントが通知されます。

受け付けられたタスクは*処理実行可能状態*です。

> **コラム**
>
> Java*Taskインタフェース*を実装したタスクを登録した場合、*Task#taskAccepted*メソッドが呼び出されます。
>
> サーバサイドJavaScriptでタスクを登録した場合、指定されたJSファイルの*taskAccepted(event)*関数が呼び出されます（同関数が定義されている場合）。

> **コラム**
>
> 「受け付け」とはタスクがいずれ開始されることが確約された状態です。 この時点ではビジネスロジックはまだ開始されていません。

タスク処理開始

タスクの処理を実行する直前になると、該当するイベントが通知されます。

このイベント通知が問題なく終了すると、タスクのビジネスロジックは直ちに開始され、*処理実行中状態*です。

> **コラム**
>
> Java*Taskインタフェース*を実装したタスクを登録した場合、*Task#taskStarted*メソッドが呼び出されます。
>
> サーバサイドJavaScriptでタスクを登録した場合、指定されたJSファイルの*taskStarted(event)*関数が呼び出されます（同関数が定義されている場合）。

タスク処理完了

タスクのビジネスロジックが完了すると、該当するイベントが通知されます。 このイベントは、ビジネスロジックの内部で例外が発生した場合も通知されます。

このイベント通知が問題なく終了（例外発生による終了も含む）すると、タスクは以下のいずれかの状態に変わります。

- *処理実行中状態*でこのイベントが発生した場合、*破棄状態*に変わります。
- *処理終了待機状態*でこのイベントが発生した場合、*選択待機状態*か*破棄状態*のいずれかに変わります。 詳細は*処理終了待機状態*を参照してください。

> **コラム**
>
> Java*Taskインタフェース*を実装したタスクを登録した場合、*Task#taskCompleted*メソッドが呼び出されます。
>
> サーバサイドJavaScriptでタスクを登録した場合、指定されたJSファイルの*taskCompleted(event)*関数が呼び出されます（同関数が定義されている場合）。

タスク受付拒否

*受付待機状態*となっているタスクが何らかの理由で非同期処理機能によって受け付けされなかった場合、該当するイベントが通知されます。

この場合タスクは*選択待機状態*に戻ります。

> **コラム**
>
> Java*Taskインタフェース*を実装したタスクを登録した場合、*Task#taskRejected*メソッドが呼び出されます。
>
> サーバサイドJavaScriptでタスクを登録した場合、指定されたJSファイルの*taskRejected(event)*関数が呼び出されます（同関数が定義されている場合）。

タスク終了通知

*処理実行中状態*であるタスクに対して*終了通知* がされる場合、該当するメソッドが呼び出されます。

このイベント通知が問題なく終了すると、*処理終了待機状態*に変わります。

この状態になると、たとえビジネスロジックが完了していなくても、タスクは非同期処理機能からは管理対象外の（ビジネスロジックを実行していない）タスクとして扱われます。

> **コラム**
> Javaで*Taskインタフェース*を実装したタスクを登録した場合、*Task#release*メソッドが呼び出されます。

> **注意**
> サーバサイドJavaScriptには該当するメソッドが存在しません。 しかしながら、*終了通知*がされたタスクは、たとえビジネスロジックが実行中であっても非同期処理機能からは管理対象外のタスクとして扱われます。

> **コラム**
> タスク終了通知については、*終了通知*も参照してください。

タスク強制中断

*処理実行中状態*であるタスクに対して*強制停止* がされるとJavaのjava.lang.Thread#stop()が呼び出されます。

メソッドが呼び出された後、*タスク処理完了*で示されたイベントが通知されます。

強制停止された時の指定内容によって、タスクは以下のいずれかの状態に変わります。

- タスクキューの先頭に再登録する場合：
  選択待機状態
- 再登録しない場合：
  破棄状態

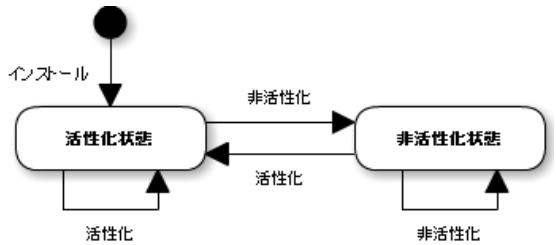
> **コラム**
> タスク強制中断については、*強制停止*も参照してください。

タスクメッセージの削除

選択待機状態にあるタスクは任意の時点でタスクキューから削除することが可能です。

この場合、タスクは*破棄状態*に変わります。

並列タスクキュー



並列タスクキューの状態遷移図

並列タスクキューはテナントで1つだけ用意されています。

並列タスクキューは追加または削除することができません。

状態

有効状態

並列タスクキューの先頭からタスクメッセージを取得し、タスクに変換してビジネスロジックを実行できる状態です。

無効状態

並列タスクキューの先頭からタスクメッセージを取得できない状態です。

タスクメッセージの登録は可能です。

イベント

インストール時

並列タスクキューは非同期処理機能がインストールされた時点では*有効状態*です。

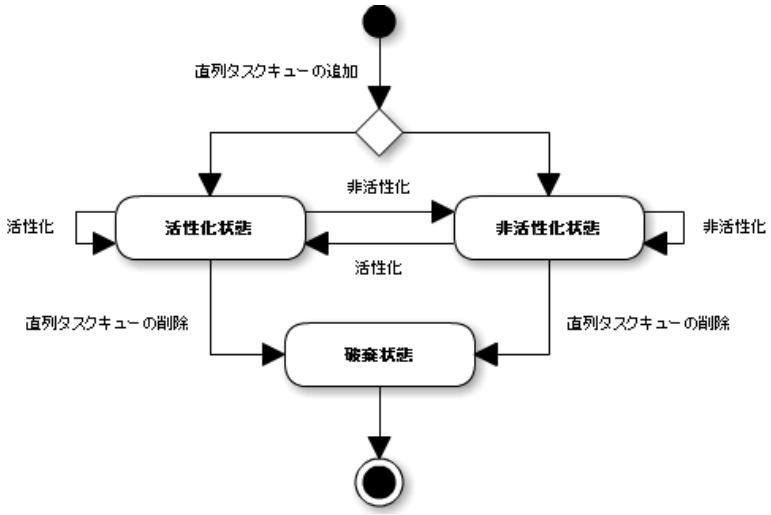
並列タスクキューの活性化

並列タスクキューは任意の時点で*有効状態*にすることが可能です。

並列タスクキューの非活性化

並列タスクキューは任意の時点で*無効状態*にすることが可能です。

直列タスクキュー



直列タスクキューの状態遷移図

直列タスクキューは非同期処理機能がインストールされた時点では何も登録されていない状態です。

状態

破棄状態

直列タスクキューが非同期処理機能から削除された状態です。

この状態ではタスクメッセージの登録や削除は出来ません。

有効状態

タスク実行エンジンが直列タスクキューの先頭からタスクメッセージを取得し、タスクに変換してビジネスロジックを実行できる状態です。

ただし、直列タスクキュー内に現在以下の状態のタスクが存在する場合、タスク実行エンジンはタスクメッセージを取得できません。

- 初期状態
- 受付待機状態
- 処理実行可能状態
- 処理実行中状態

無効状態

直列タスクキューの先頭からタスクメッセージを取得できない状態です。

タスクメッセージの登録は可能です。

イベント

直列タスクキューの追加

直列タスクキューは任意の時点で追加することが可能です。

この時、初期状態を以下のいずれかにして追加することができます。

- *有効状態*
- *無効状態*

直列タスクキューの削除

直列タスクキューは任意の時点で削除することが可能です。

ただし、タスクメッセージが1つでも登録されている直列タスクキューは削除できません。

直列タスクキューの活性化

直列タスクキューは任意の時点で*有効状態*にすることが可能です。

直列タスクキューの非活性化

直列タスクキューは任意の時点で*無効状態*にすることが可能です。

状態管理

非同期処理機能には、以下の状況を確認する機能があります。

- 並列処理機能
  - 並列タスクキューの活性化状態
  - 並列タスクキューで待機中のタスクメッセージ
  - 並列タスクキューで処理実行中のタスク
- 直列処理機能
  - 直列タスクキューの一覧
  - 直列タスクキューの活性化状態
  - 直列タスクキューで待機中のタスクメッセージ
  - 直列タスクキューで処理実行中のタスク

Java

Javaを利用して非同期処理機能の現在の状況を取得するためには、`jp.co.intra_mart.foundation.asynchronous.TaskManager`の`getRegisteredInfo`メソッドを利用します。

```
public static jp.co.intra_mart.foundation.asynchronous.report.RegisteredInfo
  getRegisteredInfo()
  throws
    jp.co.intra_mart.foundation.asynchronous.TaskControlException
```

戻り値の`RegisteredInfo`には、現在の非同期処理機能の状況のスナップショットが戻されます。

RegisteredInfo

`jp.co.intra_mart.foundation.asynchronous.report.RegisteredInfo`インタフェースには以下のようなメソッドが用意されています。

```
public abstract RegisteredParallelizedTaskQueueInfo getParallelizedTaskQueueInfo();

java.util.Map<java.lang.String, RegisteredSerializedTaskQueueInfo>
  getSerializedTaskQueuesInfo();
```

- `getParallelizedTaskQueueInfo()`メソッドは並列タスクキューの情報を取得することができます。
- `getSerializedTaskQueuesInfo`メソッドは、登録されているすべての直列タスクキューの情報を取得することができます。このメソッドの戻り値はキューIDをキーに、キューIDに該当する直列タスクキューを値とする`java.util.Map`です。

RegisteredParallelizedTaskQueueInfo

`jp.co.intra_mart.foundation.asynchronous.report.RegisteredParallelizedTaskQueueInfo`インタフェースには以下のようなメソッドが用意されています。

RegisteredParallelizedTaskQueueInfo#isActive

```
boolean isActive();
```

並列タスクキューの現在の状態（*有効状態*または*無効状態*）を取得します。

- *有効状態*の場合`true`です。
- *無効状態*の場合`false`です。

> **コラム**
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetParallelizedTaskQueuesStatus()メソッドを利用しても、取得できます。

RegisteredParallelizedTaskQueueInfo#getWaitingTasksInfo

```
java.util.List<RegisteredParallelizedTaskInfo> getWaitingTasksInfo();
```

並列タスクキューに登録後、現在処理待機中となっているタスクの一覧情報を取得することができます。

並列タスクキューに登録されているタスクの順序が反映されます。

> **コラム**
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、現在処理待機中となっているタスクの件数は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetParallelizedTaskQueuesStatus()メソッドを利用して取得できます。

RegisteredParallelizedTaskQueueInfo#getRunningTasksInfo

```
java.util.Set<RegisteredParallelizedTaskInfo> getRunningTasksInfo();
```

並列タスクキューに登録後、現在*処理実行中状態*となっているタスクの一覧情報を取得することができます。 タスクの順序は不定です。

> **コラム**
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、現在*処理実行中状態*となっているタスクの件数は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetParallelizedTaskQueuesStatus()メソッドを利用して取得できます。

RegisteredParallelizedTaskInfo

jp.co.intra_mart.foundation.asynchronous.report.RegisteredParallelizedTaskInfoインタフェースには以下のようなメソッドが用意されています。

RegisteredParallelizedTaskInfo#getMessageId

```
java.lang.String getMessageId();
```

登録されているタスクメッセージまたはタスクのメッセージIDを取得します。

メッセージIDはタスクメッセージ登録時に非同期処理機能から返される一意の値です。

RegisteredParallelizedTaskInfo#getSentTimeInMillis / getSentTime

```
long getSentTimeInMillis();

java.util.Calendar getSentTime();
```

送信日時を取得します。

送信日時とは、タスクメッセージを登録するためのAPIを呼び出した日時です。

RegisteredParallelizedTaskInfo#getReceivedTimeInMillis / getReceivedTime

```
long getReceivedTimeInMillis();

java.util.Calendar getReceivedTime();
```

受信日時を取得します。

受信日時とは、非同期処理機能が実際に並列タスクキューにタスクメッセージを登録した日時であり、送信日時とは異なる場合があります。

RegisteredParallelizedTaskInfo#getTaskClassName

```
java.lang.String getTaskClassName();
```

タスクのクラス名を取得します。

登録時にサーバサイドJavaScriptのAPIを使用した場合、この値は不定です。

RegisteredParallelizedTaskInfo#getNode

```
java.lang.String getNode();
```

タスクのビジネスロジックが現在実行されているノード名を取得します。

タスクが実行中でない場合、このメソッドは`null`を返します。

RegisteredParallelizedTaskInfo#getAcceptTimeInMillis / getAcceptTime

```
long getAcceptTimeInMillis();

java.util.Calendar getAcceptTime();
```

タスクが受け付けられ*処理実行可能状態*になった日時を取得します。

タスクがまだ受け付けられていない場合、`getAcceptTimeInMillis()`メソッドは`RegisteredTaskInfo.UNALLOCATED_TIMESTAMP`を、`getAcceptTime()`メソッドは`null`を返します。

RegisteredParallelizedTaskInfo#getStartTimeInMillis / getStartTime

```
long getStartTimeInMillis();

java.util.Calendar getStartTime();
```

タスクのビジネスロジックが開始され*処理実行中状態*になった日時を取得します。

タスクがまだ受け付けられていない場合、`getStartTimeInMillis()`メソッドは`RegisteredTaskInfo.UNALLOCATED_TIMESTAMP`を、`getStartTime()`メソッドは`null`を返します。

RegisteredSerializedTaskQueueInfo

`jp.co.intra_mart.foundation.asynchronous.report.RegisteredSerializedTaskQueueInfo`インタフェースには以下のようなメソッドが用意されています。

RegisteredSerializedTaskQueueInfo#isActive

```
boolean isActive();
```

直列タスクキューの現在の状態（*有効状態*または*無効状態*）を取得します。

- *有効状態*の場合`true`です。

- *無効状態*の場合falseです。

> **コラム**
>
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetSerializedTaskQueuesStatusById(java.lang.String)メソッドを利用しても、取得できます。
>
> すべての直列タスクキューの状態情報は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetAllSerializedTaskQueuesStatus()メソッドを利用して取得できます。

RegisteredSerializedTaskQueueInfo#getQueueId

```
String getQueueId();
```

直列タスクキューのキューIDを取得します。

> **コラム**
>
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetAllSerializedTaskQueuesStatus()メソッドを利用しても、取得できます。

RegisteredSerializedTaskQueueInfo#getWaitingTasksInfo

```
java.util.List<RegisteredSerializedTaskInfo> getWaitingTasksInfo();
```

直列タスクキューに登録後、現在処理待機中となっているタスクの一覧情報を取得することができます。

直列タスクキューに登録されているタスクの順序が反映されます。

> **コラム**
>
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、現在処理待機中となっているタスクの件数は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetSerializedTaskQueuesStatusById(java.lang.String)メソッドを利用して取得できます。
>
> すべての直列タスクキューの処理待機中となっているタスクの件数は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetAllSerializedTaskQueuesStatus()メソッドを利用して取得できます。

RegisteredSerializedTaskQueueInfo#getRunningTasksInfo

```
RegisteredSerializedTaskInfo getRunningTaskInfo();
```

直列タスクキューに登録後、現在*処理実行中状態*となっているタスクを取得することができます。

直列タスクキューに*処理実行中状態*となっているタスクが存在しない場合、nullが返されます。

> **コラム**
>
> intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、現在*処理実行中状態*となっているタスクの件数は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetSerializedTaskQueuesStatusById(java.lang.String)メソッドを利用して取得できます。
>
> すべての直列タスクキューの*処理実行中状態*となっているタスクの件数は、jp.co.intra_mart.foundation.asynchronous.TaskManagerのgetAllSerializedTaskQueuesStatus()メソッドを利用して取得できます。

RegisteredSerializedTaskInfo

jp.co.intra_mart.foundation.asynchronous.report.RegisteredSerializedTaskInfoインタフェースには以下のようなメソッドが用意されています。

RegisteredSerializedTaskInfo#getQueueId

```
java.lang.String getQueueId();
```

直列タスクキューのキューIDを取得します。

RegisteredSerializedTaskInfo#isStoppingProgressOnError

```
boolean isStoppingProgressOnError();
```

実行したタスクのビジネスロジックで例外が発生した場合、直列タスクキューを停止するかどうかを取得します。

- true
  例外発生時には直列タスクキューを現在の状態に関わらず*無効状態*にし、後続のタスクの処理を停止します。
- false
  例外発生時であっても直列タスクキューの状態を変更しません。

RegisteredSerializedTaskInfo#getMessageId

```
java.lang.String getMessageId();
```

登録されているタスクメッセージまたはタスクのメッセージIDを取得します。

メッセージIDはタスクメッセージ登録時に非同期処理機能から返される一意の値です。

RegisteredSerializedTaskInfo#getSentTimeInMillis / getSentTime

```
long getSentTimeInMillis();

java.util.Calendar getSentTime();
```

送信日時を取得します。

送信日時とは、タスクメッセージを登録するためのAPIを呼び出した日時です。

RegisteredSerializedTaskInfo#getReceivedTimeInMillis / getReceivedTime

```
long getReceivedTimeInMillis();

java.util.Calendar getReceivedTime();
```

受信日時を取得します。

受信日時とは、非同期処理機能が実際に直列タスクキューにタスクメッセージを登録した日時であり、送信日時とは異なる場合があります。

RegisteredSerializedTaskInfo#getTaskClassName

```
java.lang.String getTaskClassName();
```

タスクのクラス名を取得します。

登録時にサーバサイドJavaScriptのAPIを使用した場合、この値は不定です。

RegisteredSerializedTaskInfo#getNode

```
java.lang.String getNode();
```

タスクのビジネスロジックが現在実行されているノード名を取得します。

タスクが実行中でない場合、このメソッドはnullを返します。

RegisteredSerializedTaskInfo#getAcceptTimeInMillis / getAcceptTime

```
long getAcceptTimeInMillis();

java.util.Calendar getAcceptTime();
```

タスクが受け付けられ*処理実行可能状態*になった日時を取得します。

タスクがまだ受け付けられていない場合、getAcceptTimeInMillis()メソッドはRegisteredTaskInfo.UNALLOCATED_TIMESTAMPを、getAcceptTime()メソッドはnullを返します。

RegisteredSerializedTaskInfo#getStartTimeInMillis / getStartTime

```
long getStartTimeInMillis();

java.util.Calendar getStartTime();
```

タスクのビジネスロジックが開始され*処理実行中状態*になった日時を取得します。

タスクがまだ受け付けられていない場合、getStartTimeInMillis()メソッドはRegisteredTaskInfo.UNALLOCATED_TIMESTAMPを、getStartTime()メソッドはnullを返します。

サーバサイドJavaScript

サーバサイドJavaScriptを利用して非同期処理機能の現在の状況を取得するためには、WorkManagerオブジェクトのgetRegisteredInfo関数を利用します。

```
function getRegisteredInfo()
```

この関数の呼び出しが成功した場合、以下のような構造のオブジェクトが戻り値として返されます。

- error - false
- data - 非同期処理機能の現在の状況のスナップショット
  - parallelizedTasksInfo - 並列タスクキュー
    - isActive - 並列タスクキューの処理状況です。

値	説明
true	*有効状態*
false	*無効状態*

    - waitingTasks - 並列タスクキューで*受付待機状態*のタスクメッセージの配列です。順序は並列タスクキューに登録されている内容と同じです。
      配列内の各要素は、以下の構成を持つオブジェクトです。
      - messageId - メッセージID
      - sentTime - 送信日時
      - receivedTime - 受信日時
      - taskClassName - タスク実行クラス名
    - runningTasks - 並列タスクキューで*受付待機状態*以外のタスクメッセージの配列です。順序は不定です。

    - messageId - メッセージID
    - sentTime - 送信日時
    - receivedTime - 受信日時
    - taskClassName - タスク実行クラス名
    - node - タスクが実行されているノードのノードID
    - acceptTime - タスクが受け付けられ、*処理実行可能状態*となった時の日時
    - startTime - タスクが開始され、*処理実行中状態*となった時の日時（まだ開始されていない場合はnull）
- serializedTasksInfo - 直列タスクキューの処理状況です。キューIDをkey、直列タスクキューの情報をvalueとする連想配列です。
  直列タスクキューの情報は以下の情報から構成されています。
    - queueId - キューID
    - isActive - 直列タスクキューの処理状況です。

値	説明
true	*有効状態*
false	*無効状態*

    - waitingTasks - 直列タスクキューで*受付待機状態*のタスクメッセージの配列です。順序は直列タスクキューに登録されている内容と同じです。
      配列内の各要素は以下の構成を持つオブジェクトです。
        - queueId - キューID
        - isStoppingProgressOnError - タスクのビジネスロジック実行時に例外が発生した場合、直列タスクキューを*無効状態*にするかどうかを示すフラグです。

値	説明
true	例外発生時には直列タスクキューを停止（*無効状態*）します。
false	例外発生時であっても直列タスクキューの状態を変更しません。

        - messageId - メッセージID
        - sentTime - 送信日時
        - receivedTime - 受信日時
        - taskClassName - タスク実行クラス名
    - runningTask - 直列タスクキューで*受付待機状態*以外のタスクメッセージの配列です。順序は不定です。
        - queueId - キューID
        - isStoppingProgressOnError - タスクのビジネスロジック実行時に例外が発生した場合、直列タスクキューを*無効状態*にするかどうかを示すフラグです。

値	説明
true	例外発生時には直列タスクキューを停止（*無効状態*）します。
false	例外発生時であっても直列タスクキューの状態を変更しません。

        - messageId - メッセージID
        - sentTime - 送信日時
        - receivedTime - 受信日時
        - taskClassName - タスク実行クラス名
        - node - タスクが実行されているノードのノードID

- acceptTime - タスクが受け付けられ、*処理実行可能状態*となった時の日時
- startTime - タスクが開始され、*処理実行中状態*となった時の日時（まだ開始されていない場合はnull）

コラム

非同期処理管理画面を利用することによって現在の状態を確認することも可能です。

詳細についてはシステム管理者 操作ガイドの非同期-タスクキュー一覧を参照してください。

設定

保存場所

非同期処理機能で使用するデータはシステムデータベースの以下のテーブルに保存されます。

- im_async_queue_info
  *タスクキュー*の情報を保持します。
- im_async_task_info
  *タスクメッセージ*を保持します。
- im_async_context_info
  *タスク処理開始*時に使用するコンテキストの情報を保持します。

この内容は、たとえintra-mart Accel Platformが稼働していない状態であっても編集しないようにしてください。

> **注意**
>
> intra-mart Accel Platform停止中にシステムデータベースの非同期処理機能で使用するデータを削除した場合、
> 登録されたタスクが実行されなくなってしまいます。

実行エンジン（共通）

task-runner-config.xml

- `task-runner-config`
  - `max-threads`
    同時実行可能なタスクの数を指定します。
  - `round-interval`
    タスクキュー管理サービスに対して実行可能なタスクメッセージを問い合わせる間隔を秒単位で指定します。

> **注意**
>
> max-threads は、タスク処理サービスが実行されているサーバに対する設定値です。
> システム全体に対する設定値ではありません。
> システム全体で同時に実行できるタスク数は、以下のように求めることができます。
>
> - max-threads の設定値 × タスク処理サービスが実行されているサーバ台数

実行エンジン（実装依存）

Resin

Resinでは非同期処理のタスク実行エンジンとしてJCA(Java EE Connector Architecture)を利用しています。

task-executor-jca-config.xml

- `task-executor-jca-config`
  - `work-manager-resource-info`
    - `jndi-name`
      タスク実行エンジンで利用するリソースアダプタのJNDI参照名を指定します。

resin-web.xml

`<web-app>/ <resource>` に以下の内容を指定します。

```
<web-app xmlns="http://caucho.com/ns/resin">
  <resource jndi-name="jca/work"
        type="jp.co.intra_mart.system.asynchronous.impl.executor.work.
            resin.ResinResourceAdapter" /> <!-- 実際には1行 -->
</web-app>
```

- jndi-name
  *task-executor-jca-config.xml*で指定したjndi-nameと同一にします。
- type
  jp.co.intra_mart.system.asynchronous.impl.executor.work.resin. ResinResourceAdapter固定です。

Commonj (Weblogic)

Weblogicでは非同期処理のタスク実行エンジンとしてCommonjを利用しています。

非同期処理機能ではJNDIを通じてCommonjのWorkManagerを取得しています。

非同期処理機能ではタスクだけではなくタスク実行エンジンもWorkManager上で動作しますので、 WorkManagerで同時に実行可能な最大スレッド数は、最低でも*task-runner-config.xml*の<max-threads>で設定した値**+ 1**にしてください。

task-executor-commonj-config.xml

以下の値を設定してください。

- task-executor-commonj-config
  - work-manager-info
    - jndi-name
      CommonjのWorkManagerを参照するJNDI名。web.xmlにもこれに準じたリソース参照名を設定する必要があります。

web.xmlの例

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0" metadata-complete="false">

  <resource-ref>
    <res-ref-name>wm/WorkManager</res-ref-name>
    <res-type>commonj.work.WorkManager</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>

</web-app>
```

<web-app>/ <resource-ref>/ <res-ref-name> には*task-executor-commonj-config.xml*の<jndi-name>で設定したものと同じ値を設定してください。

WorkManagerの設定

> **コラム**
> Weblogic上でWorkManagerを設定する場合の詳細については、Weblogicに関連するマニュアル等を御覧ください。

ここではweblogic.xmlでWorkManagerを設定する例を説明します。

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-web-app
    http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">

  <work-manager>
    <name>wm/WorkManager</name>
    <min-threads-constraint>
      <name>MinThreadsCountFour</name>
      <count>4</count>
    </min-threads-constraint>
    <ignore-stuck-threads>true</ignore-stuck-threads>
  </work-manager>
</weblogic-web-app>
```

- <weblogic-web-app>/ <work-manager>/ <name>には*task-executor-commonj-config.xml*の<jndi-name>で設定したものと同じ値を設定してください。
- <weblogic-web-app>/ <work-manager>/ <min-threads-constraint>/ <count>には最低でも *task-runner-config.xml*の<max-threads>で設定した値**+ 1**を設定ください。

Commonj (WebSphere)

WebSphereでは非同期処理のタスク実行エンジンとしてCommonjを利用しています。

非同期処理機能ではJNDIを通じてCommonjのWorkManagerを取得しています。

非同期処理機能ではタスクだけではなくタスク実行エンジンもWorkManager上で動作しますので、 WorkManagerで同時に実行可能な最大スレッド数は、最低でも*task-runner-config.xml*の<max-threads>で設定した値**+ 1**にしてください。

task-executor-commonj-config.xml

以下の値を設定してください。

- task-executor-commonj-config
  - work-manager-info
    - jndi-name
      CommonjのWorkManagerを参照するJNDI名。web.xmlにもこれに準じたリソース参照名を設定する必要があります。

web.xmlの例

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0" metadata-complete="false">

  <resource-ref>
    <res-ref-name>wm/WorkManager</res-ref-name>
    <res-type>commonj.work.WorkManager</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>

</web-app>
```

<web-app> <resource-ref> <res-ref-name>には*task-executor-commonj-config.xml*の<jndi-name>で設定したものと同じ値を設定してください。

WorkManagerの設定

コラム

WebSphere上でWorkManagerを設定する場合の詳細については、WebSphereに関連するマニュアル等を御覧ください。

コンソール画面から[リソース]-[非同期 Bean]-[作業マネージャー]でWorkManagerを設定することが可能です。

デプロイ時には、ここで設定したWorkManagerを*task-executor-commonj-config.xml*で設定したJNDI参照名で参照できるようにしてください。