Copyright © 2012 NTT DATA INTRAMART CORPORATION

# 目次


- 改訂情報
- はじめに
  - 本書の目的
  - 対象読者
  - 本書の構成
- 概要
  - ビジネスロジックの範囲を決定
  - ビジネスロジックで必要となるパラメータを決定
  - タスクの実装
  - タスクメッセージを登録するアプリケーションを実装
- タスクの実装
  - 各メソッドまたは関数の実装
  - タスクを実装する時の注意点
  - サンプル
- タスクメッセージ登録処理
  - タスクキューに対するタスクメッセージの登録（共通事項）
  - 並列タスクキューに対するタスクメッセージの登録
  - 直列タスクキューに対するタスクメッセージの登録
- パラメータ
  - Java
  - サーバサイドJavaScript
- タスクキューの管理
  - 並列タスクキューの有効化／無効化
  - 直列タスクキューの有効化／無効化
  - 直列タスクキューの登録
  - 直列タスクキュー の削除

# 改訂情報

| 変更年月日 | 変更内容 |
| --- | --- |
| 2012-10-01 | 初版 |
| 2017-08-01 | 第2版 下記を追加・変更しました<br>■ 「*タスクキューの管理*」に「*直列タスクキューの削除（Java）*」に関するコラムを追加しました。 |
| 2019-04-01 | 第3版 下記を変更しました<br>■ 「*パラメータ*」の「*サーバサイドJavaScript*」に記載されている、サーバサイドJavaScriptでパラメータを渡す場合の制限を変更しました。 |

# はじめに

## 本書の目的

本書では 非同期処理機能 を利用したアプリケーションを開発する場合の基本的な方法や注意点等について説明します。

## 対象読者

次の開発者を対象としてます。

- 非同期処理機能 を利用する
- 非同期処理機能 の 仕様書 を理解している
- 以下のいずれかを理解している
  - Java
  - サーバサイドJavaScript

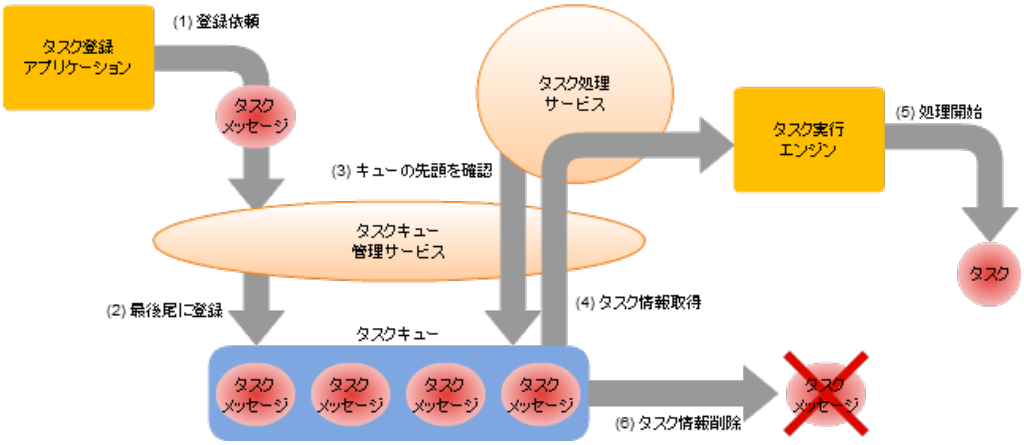
## 本書の構成

- *概要*
  非同期処理機能 を利用したアプリケーションを開発するときの基本的な手順について説明します。
- *タスクの実装*
  タスク の実装方法について説明します。
- *タスクメッセージ登録処理*
  タスク の登録方法について説明します。
- *パラメータ*
  タスク に受け渡すパラメータについて説明します。
- *タスクキューの管理*
  タスクキュー の管理について説明します。

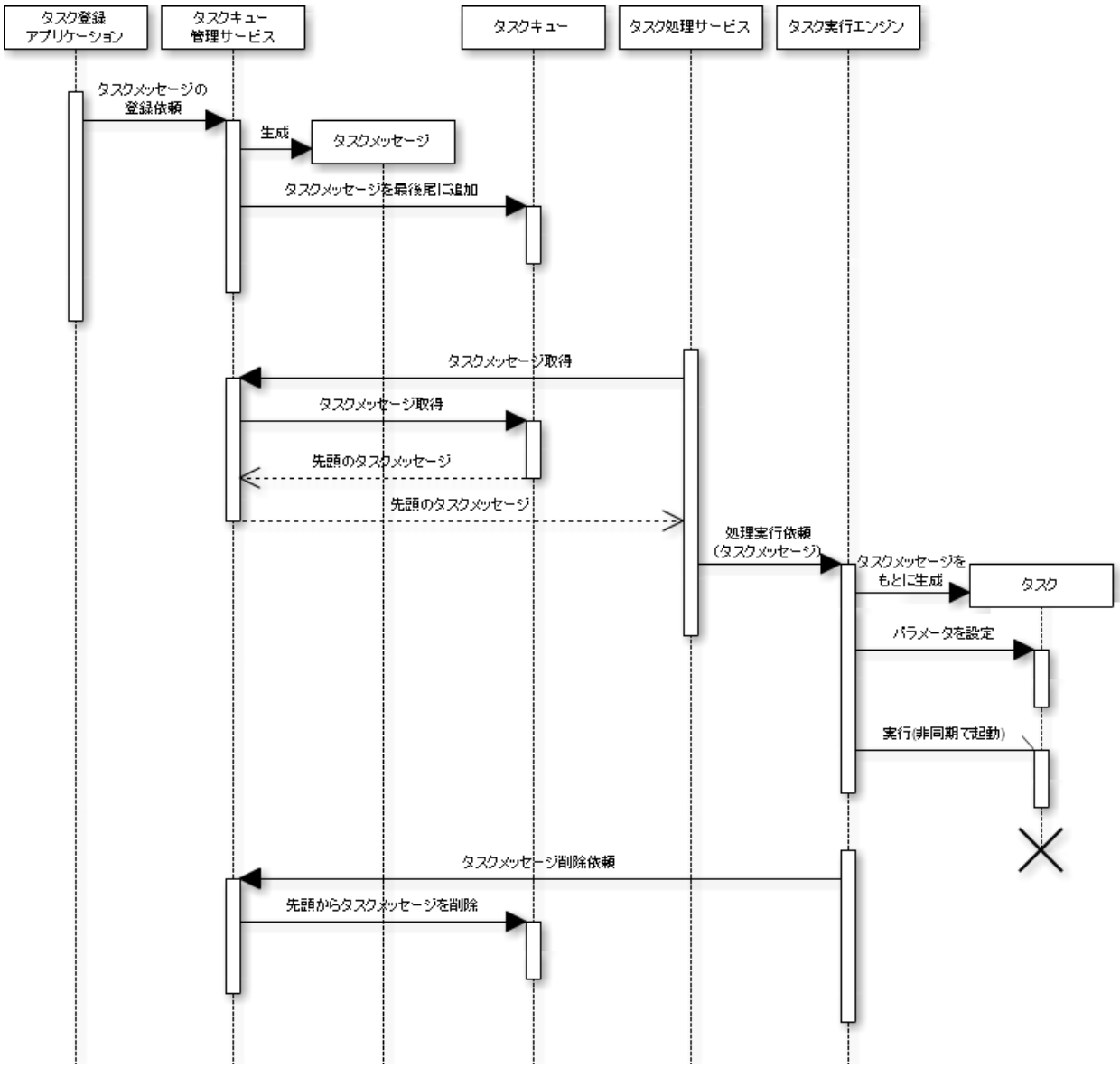
# 概要

非同期処理機能はさまざまな機能やアプリケーションから構成されています。

非同期処理機能の全体的な流れを*非同期処理機能の流れ*に、全体の処理概要のシーケンスを*非同期処理機能のシーケンス*に示します。
非同期処理機能の仕様書も参照してください。




非同期処理機能の流れ




非同期処理機能のシーケンス

非同期処理機能を利用する場合、開発者は以下のものを実装する必要があります。

- タスク
- タスク登録アプリケーション

アプリケーションの開発は以下のような流れです。

1. *ビジネスロジックの範囲を決定*
2. *ビジネスロジックで必要となるパラメータを決定*
3. *タスクの実装*
4. *タスクメッセージを登録するアプリケーションを実装*

## ビジネスロジックの範囲を決定

非同期でビジネスロジックを処理する場合、以下の点について注意する必要があります。

- 非同期では、ビジネスロジックは呼び出し側とは異なるサーバで処理が行われる場合があります。
- タスクメッセージ登録時のプログラム中で何らかの値をメモリ内の共有領域に保存しても、実際のビジネスロジック側ではその値を利用できません。 何らかの情報の受け渡しを行う場合、リモート環境からであっても取得できるような方法を開発者が用意する必要があります。

タスクメッセージの登録時とタスクのビジネスロジック実行時の間で情報を受け渡す方法として、以下のような方法があります。

- パラメータ
  タスクメッセージの登録時に独自のパラメータを設定することができます。
- コンテキスト
  タスクメッセージの登録時のコンテキストは非同期を行うときにも引き継がれます。 そのため、コンテキストを介して情報を受け渡すことができます。

上記以外にも、ストレージサービスやデータベースを使うなどの方法もあります。

## ビジネスロジックで必要となるパラメータを決定

タスクメッセージの登録時に独自のパラメータを設定することができます。

設定できるパラメータの形式には制限があるので、タスクメッセージの登録時およびタスクの実行時にはそれぞれ変換が必要です。

詳細は*パラメータ*を参照してください。

## タスクの実装

タスクは以下のことを注意しながら実装する必要があります。

- タスクは非同期で処理が行われる
- 任意の時点で終了通知が行われる場合がある

詳細は*タスクの実装*を参照してください。

## タスクメッセージを登録するアプリケーションを実装

非同期を利用する場合、タスクメッセージをタスクキューに登録する必要があります。 タスクを直接起動しないでください。

> **コラム**
> タスクメッセージを登録するとき、ビジネスロジックの内部で使用するデータとなるパラメータを渡すことができます。

詳細は*タスクメッセージ登録処理*を参照してください。

# タスクの実装

タスクは非同期処理機能において実際にビジネスロジックを実行する部分です。

本章ではいくつかの断片的なサンプルを交えながらタスクの実装方法と注意点について説明します。

## 各メソッドまたは関数の実装

タスクを実装するためには、さまざまなメソッドや関数を定義する必要があります。

本節ではそれぞれの関数の役割とその実装内容について説明します。

*タスクのメソッド／関数の一覧*もあわせて参照してください。

**タスクのメソッド／関数の一覧**

| 項目 | Java （Taskインタフェース） | Java （AbstractTask抽象クラス） | サーバサイドJavaScript |
|---|---|---|---|
| *パラメータ設定* | setParameter | setParameter<br>getParameter | setParameter |
| *ビジネスロジックの実行* | run | run | run |
| *終了通知* | release | release | — |
| *処理実行可能状態の通知* | taskAccepted | taskAccepted | taskAccepted |
| *処理実行中状態の通知* | taskStarted | taskStarted | taskStarted |
| *ビジネスロジック終了の通知* | taskCompleted | taskCompleted | taskCompleted |
| *タスクの受付拒否* | taskRejected | taskRejected | taskRejected |

### メソッド共通事項／関数共通事項

ここでは各メソッドおよび関数の実装において共通することを記載します。

#### リソース

それぞれのメソッドや関数内で解放が必須となるリソース（ファイルの入出力やデータベースの接続など）を取得した場合、各メソッドおよび関数を終了する時には必ずリソースの解放を行ってください。 その理由のいくつかは以下の通りです。

- 各メソッドや関数は異なるスレッド上で実行される場合があります。スレッドにリソースが関連付けられている場合、リソース取得時のスレッドとは異なるスレッドでリソースを解放しようとすると不具合が発生することが考えられます。
- 各イベントに該当するメソッドや関数は、実行状況によってはすべてが必ず呼び出されるとは限りません。
  （例）taskRejectedが呼び出された場合、runやtaskCompletedは呼び出されません。

> **コラム**
> 各メソッドや関数は異なるスレッド上で実行される場合がありますが、Javaの*終了通知*を除けば、複数のメソッドや関数が同時に実行されることはありません。 *終了通知*のみが、その他のメソッドと同時に実行される場合があります。

#### Java

Javaを利用して、ビジネスロジックとして動作するタスクを実現するためには、jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを実装する必要があります。

Task インタフェースの実装方法として、以下の様な方法があります。

- Taskインタフェースを直接implementsする方法
  jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを実装してタスクを作成する場合、すべてのメソッドを実装する必要があります。
  この方法を採用する場合、ビジネスロジックに不要なメソッドを実装する必要もあるため、特別な理由がない限り、次の方法で実

装することを推奨します。

- AbstractTask抽象クラスをextendsする方法
  jp.co.intra_mart.foundation.asynchronous.AbstractTask抽象クラスを利用すると、必要なメソッドのみをオーバーライドすればタスクを作成することができます。
  この方法を採用した場合、Taskインタフェースを直接実装する方法に比べると以下の様なメリットがあります。
  - 不要なメソッドのオーバーライドをする必要がありません。
  - ビジネスロジック実行時に必要となるパラメータの受け渡しが簡略化できます。

特別な理由がない限り、AbstractTask抽象クラスを拡張してタスクを作成する方法を推奨します。

Javaを利用してタスクを実装するときのメソッドとして*Javaを利用するタスク*も参照してください。

**Javaを利用するタスク**

| メソッド | **Task** インタフェース | **AbstractTask** 抽象クラス | 備考 |
|---|---|---|---|
| setParameter | オーバーライド必須 | オーバーライド任意 | — |
| getParameter | （存在しない） | オーバーライド任意 | setParameter をオーバライドした場合、その内容に応じて修正が必要 |
| run | オーバーライド必須 | オーバーライド任意 | ビジネスロジックを実装する場合はオーバーライド必須 |
| release | オーバーライド必須 | オーバーライド任意 | 外部から安全に終了させる場合はオーバーライド必須 |
| taskAccepted | オーバーライド必須 | オーバーライド任意 | — |
| taskStarted | オーバーライド必須 | オーバーライド任意 | — |
| taskCompleted | オーバーライド必須 | オーバーライド任意 | — |
| taskRejected | オーバーライド必須 | オーバーライド任意 | — |

### サーバサイドJavaScript

サーバサイドJavaScriptを利用して、ビジネスロジックとして動作するタスクを実現するためには、特定の関数群を実装したJSファイルを作成する必要があります。

非同期処理機能から利用される関数は複数ありますが、実際には必要となる関数のみ実装すれば問題ありません。 run関数のみ実装が必須ですが、その他の関数の実装は任意です。

サーバサイドJavaScriptを利用してタスクを実装するときの関数として*サーバサイドJavaScriptを利用するタスク*も参照してください。

**サーバサイドJavaScriptを利用するタスク**

| 関数 | 実装の必須／任意 | 備考 |
|---|---|---|
| setParameter | 任意 | — |
| run | 必須 | — |
| taskAccepted | 任意 | — |
| taskStarted | 任意 | — |
| taskCompleted | 任意 | — |
| taskRejected | 任意 | — |

## パラメータ設定

タスクメッセージ登録時に、ビジネスロジック内で使用する値を「パラメータ」として設定することができます。

タスクが生成されると、実際のビジネスロジックの実行に先立ってパラメータがタスクに受け渡されます。

*パラメータ設定*を参照してください。

パラメータ設定

| 開発モデル | メソッド／関数 | 備考 |
| --- | --- | --- |
| Java | setParameter | AbstractTask抽象クラスでは独自の実装がされています。 |
| Java | getParameter | AbstractTask抽象クラスに追加されています。setParameterで設定されたパラメータを取得できます。<br><br>setParameterメソッドをオーバライドしていない限り、setParameterで設定されたパラメータをこのメソッドで取得することが可能です。<br><br>setParameterメソッドが非同期処理機能から呼び出される前にこのメソッドを呼んだ場合、戻り値としてnullが返されます。 |
| サーバサイドJavaScript | setParameter | — |

### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、setParameterメソッドを独自に開発する必要があります。

setParameterメソッドに渡されたパラメータの使用方法について、特に制限はありません。 一例として、パラメータから取得できる情報をインスタンス変数に保存し、ビジネスロジック実行時（runメソッド実行時）に利用するなどの方法が考えられます。

```
package sample.task;

import java.util.Map;
import jp.co.intra_mart.foundation.asynchronous.Task;

public class MySampleTask implements Task {

  private String firstName = null;
  private String lastName = null;
  private int age = 0;

  @Override
  public void setParameter(Map<String, ?> parameter) {
    firstName = (String) parameter.get("FIRST_NAME");
    lastName = (String) parameter.get("LAST_NAME");
    age = ((Number) parameter.get("AGE")).intValue();
  }

  @Override
  public void run() {
    // firstName, lastName, ageを使って処理を実行
    ...
  }

  ...
}
```

jp.co.intra_mart.foundation.asynchronous.AbstractTask抽象クラスでは、このメソッドは引数に渡されたパラメータを内部で保存します。 この場合、内部に保存されたパラメータは getParameter メソッドで取得することが可能です。

```
package sample.task;

import java.util.Map;
import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class MySampleTask extends AbstractTask {

  @Override
  public void run() {
    Map<String, ?> parameter = getParameter();
    String firstName = (String) parameter.get("FIRST_NAME");
    String lastName = (String) parameter.get("LAST_NAME");
    int age = ((Number) parameter.get("AGE")).intValue();

    // firstName, lastName, ageを使って処理を実行
    ...
  }

  ...
}
```

#### サーバサイドJavaScript

setParameter関数に渡されたパラメータの使用方法について、特に制限はありません。 一例として、パラメータから取得できる情報を変数に保存し、ビジネスロジック実行時（run関数実行時）に利用するなどの方法が考えられます。

```
var firstName;
var lastName;
var age;

function setParameter(parameter) {
  firstName = parameter.firstName;
  lastName = parameter.lastName;
  age = parameter.age;
}

function run() {
  // firstName, lastName, ageを使って処理を実行
  ...
}
```

### ビジネスロジックの実行

実際に非同期で行う処理を記述します。

**ビジネスロジックの実行**

| 開発モデル | メソッド／関数 | 備考 |
|---|---|---|
| Java | run | AbstractTask抽象クラスではこのメソッドは何もしません。 |
| サーバサイドJavaScript | run | — |

runメソッドまたはrun関数の実装例として、*サンプル*も参照してください。

#### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、runメソッドを独自に開発する必要があります。

AbstractTask 抽象クラスでは、このメソッドは何もしません。 必要に応じてこのメソッドをオーバライドし、ビジネスロジックを実装してください。

コラム

releaseメソッドが呼ばれたら、できるだけ速やかにrunメソッドの実行を終了するような実装をしてください。 詳細については非同期処理機能の仕様書を参照してください。

### サーバサイドJavaScript

JSファイル上にrun関数を直接定義します。

ビジネスロジックを実装してください。

コラム

サーバサイドJavaScriptでビジネスロジックを実装する場合、終了通知がされてもJSファイルではそれを検知することができません。 終了通知に依存しない方法でビジネスロジックが確実に終了するように実装してください。

詳細については非同期処理機能の仕様書を参照してください。

## 終了通知

**終了通知**

| 開発モデル | メソッド／関数 | 備考 |
|---|---|---|
| Java | release | AbstractTask抽象クラスではこのメソッドは何もしません。 |
| サーバサイドJavaScript | （該当する関数はありません） | — |

releaseメソッドの実装例として、*releaseメソッドの実装*も参照してください。

### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、releaseメソッドを独自に開発する必要があります。 実行中のビジネスロジックを外部から停止させることがある場合はこのメソッドをオーバーライドしてください。

ビジネスロジックを停止させる一般的な方法はなく、実装方法は開発者に委ねられます。

注意

このメソッドをオーバーライドする場合、上記の点に注意して実装してください。

- このメソッドは他のメソッドとは異なるスレッド上から呼び出される場合があります。
- 非同期処理機能から異なるスレッド上で複数回呼び出される場合もあります。

注意

ビジネスロジック停止時にタスクの再登録やタスクキューの停止を行いたい場合、このメソッドを直接呼び出さないでください。 タスクの再登録やタスクキューの停止はJavaのTaskManager#releaseRunning*XXXX*Taskメソッドまたはサーバサイドの JavaScriptのWorkManager#releaseRunning*XXXX*Task関数で行います。 この時、 TaskManager や WorkManager は、内部で release を呼び出しています。

このメソッドを直接呼び出してrunメソッドが終了された場合、非同期処理機能からは通常の終了と区別がつかず、タスクの再登録やタスクキューの停止等が行われません。 詳細については非同期処理機能の仕様書を参照してください。

### サーバサイドJavaScript

サーバサイドJavaScriptでは終了通知に該当する関数は存在しません。

> **注意**
> サーバサイドJavaScriptで実装されたタスクに対してJavaのTaskManager#release*XXXX*Taskメソッドやサーバサイド JavaScriptのWorkManager#release*XXXX*Task関数を呼び出した場合、該当するタスクは非同期処理機能の管理対象外ですが、ビジネスロジックは停止しません。 そのため、タスクメッセージの再登録を行ったりすると、同一のタスクメッセージから生成された複数のタスクのインスタンスのビジネスロジックが同時に重複して実行される場合があります。

## 処理実行可能状態の通知

タスクが処理実行可能状態になった時に通知されます。

**処理実行可能状態の通知**

| 開発モデル | メソッド／関数 | 備考 |
|---|---|---|
| Java | taskAccepted | AbstractTask抽象クラスではこのメソッドは何もしません。 |
| サーバサイドJavaScript | taskAccepted | — |

### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、taskAcceptedメソッドを独自に開発する必要があります。 タスクが処理実行可能状態になった時に何らかの処理を行いたい場合、このメソッドをオーバーライドしてください。

### サーバサイドJavaScript

JSファイル上でタスクを開発する場合、必要に応じてtaskAccepted関数を独自に開発する必要があります。 タスクが処理実行可能状態になった時に何らかの処理を行いたい場合、この関数を定義してください。

## 処理実行中状態の通知

タスクが処理実行中状態になった時に通知されます。

**処理実行中状態の通知**

| 開発モデル | メソッド／関数 | 備考 |
|---|---|---|
| Java | taskStarted | AbstractTask抽象クラスではこのメソッドは何もしません。 |
| サーバサイドJavaScript | taskStarted | — |

### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、taskStartedメソッドを独自に開発する必要があります。 タスクが処理実行中状態になった時に何らかの処理を行いたい場合、このメソッドをオーバーライドしてください。

### サーバサイドJavaScript

JSファイル上でタスクを開発する場合、必要に応じてtaskStarted関数を独自に開発する必要があります。 タスクが処理実行中状態になった時に何らかの処理を行いたい場合、この関数を定義してください。

## ビジネスロジック終了の通知

タスクのビジネスロジックが終了した時（通常はrunメソッドやrun関数が終了した時）に通知されます。

処理実行中状態の通知

| 開発モデル | メソッド／関数 | 備考 |
| --- | --- | --- |
| Java | taskCompleted | AbstractTask抽象クラスではこのメソッドは何もしません。 |
| サーバサイドJavaScript | taskCompleted | — |

#### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、taskCompletedメソッドを独自に開発する必要があります。 タスクのビジネスロジックが正常に終了した時に何らかの処理を行いたい場合、このメソッドをオーバーライドしてください。

#### サーバサイドJavaScript

JSファイル上でタスクを開発する場合、必要に応じてtaskCompleted関数を独自に開発する必要があります。 タスクのビジネスロジックが正常に終了した時に何らかの処理を行いたい場合、この関数を定義してください。

### タスクの受付拒否

タスクの受付が拒否された時に通知されます。

タスクの受付拒否

| 開発モデル | メソッド／関数 | 備考 |
| --- | --- | --- |
| Java | taskRejected | AbstractTask抽象クラスではこのメソッドは何もしません。 |
| サーバサイドJavaScript | taskRejected | — |

#### Java

jp.co.intra_mart.foundation.asynchronous.Taskインタフェースを直接実装してタスクを開発する場合、taskRejectedメソッドを独自に開発する必要があります。 タスクメッセージが何らかの理由で拒否されて実行されなかった時に何らかの処理を行いたい場合、このメソッドをオーバーライドしてください。

#### サーバサイドJavaScript

JSファイル上でタスクを開発する場合、必要に応じてtaskRejected関数を独自に開発する必要があります。 タスクメッセージが何らかの理由で拒否されて実行されなかった時に何らかの処理を行いたい場合、この関数を定義してください。

## タスクを実装する時の注意点

タスクの非同期処理機能による実行は、アプリケーション内からのメソッド呼び出しとは実行場所やタイミングが異なるため、いくつか注意する点があります。

- *遅延実行*
  タスクは即時に実行されず、一度タスクキューに登録された後に実行されます。
- *非同期による実行*
  タスクは非同期で処理が行われます。
- *リモートによる実行*
  タスクはリモートで処理が行われる場合があります。
- *終了通知（Javaで開発する場合）*
  任意の時点で終了通知が行われる場合があります。 この場合、タスクメッセージがタスクキューの先頭に再登録される場合があります。
- *情報の受け渡し*

- run 以外のメソッドまたは関数では、以下の点について注意してください。
  - Java EE に関連するリソース（データベースへのアクセス、JNDIによる検索等）は取得しないでください。
  - コンテキストは取得しないでください。

この他にもパラメータの受け渡しについて考慮する必要があります。パラメータの受け渡しの詳細については*パラメータ*で説明します。

### 遅延実行

非同期処理機能では直接タスクを実行せず、一度タスクメッセージがタスクキューに登録されます。

非同期処理機能は登録されたタスクメッセージのビジネスロジックを *できるだけ*早く開始しようとしますが、実際に開始される時間については制限が設けられていません。

そのため、タスクがいつ開始されても問題ないような実装をしてください。

### 非同期による実行

非同期処理機能ではタスクメッセージを登録したスレッドとタスクを実行するスレッドは異なります。 そのため、登録時のスレッドに関連付けられた情報は実行時に引き継がれません。

スレッドに関連付けられた情報としては以下のようなものがあります。

- （開発言語としてJavaを利用している場合）java.lang.ThreadLocalに保存された値
  登録時にjava.lang.ThreadLocalを利用して何らかの値を登録しても、実行時には取得できません。
- 登録時のスレッドに関連付けられているトランザクション
  登録時にトランザクションを開始してもタスクにはそのトランザクションは引き継がれないため、登録時と実行時のSQLと同一のトランザクション上で動作させることは出来ません。

タスクメッセージの登録時に何らかの情報を保存し、タスク実行時にそれらの情報を利用したい場合は*情報の受け渡し*を参照してください。

### リモートによる実行

非同期処理機能ではタスクメッセージを登録した時のサーバとタスクを実行するサーバは異なる場合があります。 そのため、登録時のローカル環境に保存された情報は実行時に引き継がれません。

ローカル環境に保存された情報としては以下のようなものがあります。

- （開発言語としてJavaを利用している場合）クラス変数に保存された値
  登録時のサーバと実行時のサーバが異なる（つまり、両者のVirtual Machineが異なる）場合があるため、登録時にクラス変数に何らかの値を登録しても、実行時には取得できない場合があります。
  この現象を確実に回避する方法はありません。
- ローカルのファイルに保存された値
  登録時のサーバと実行時のサーバが異なる場合があるため、 ローカルのファイルで値を受け渡すことはできません。

タスクメッセージの登録時に何らかの情報を保存し、タスク実行時にそれらの情報を利用したい場合は*情報の受け渡し*を参照してください。

## サンプル

### 単純なサンプル

Taskの実装のサンプルを*以下*に示します。

```
package sample;

import java.util.Map;
import java.util.concurrent.TimeUnit;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;
import jp.co.intra_mart.foundation.asynchronous.TaskEvent;
```

```
import jp.co.intra_mart.foundation.async.client.CloudTaskEvent;

public class SampleTask extends AbstractTask {

  private volatile boolean isActive = true;

  private String label;

  @Override
  public void run() {
    // getParameter を利用して設定済みのパラメータを取得
    Map<String, ?> parameter = getParameter();
    String label = (String) parameter.get("label");
    Number countObj = (Number) parameter.get("count");
    int count = countObj.intValue();

    // リソースの取得
    ...

    try {
      int index = 0;
      while (index < count && this.isActive) { // release()が呼ばれている場合は途中でも終了
        // ビジネスロジック
        businessLogic(label, index);
        index++;
      }
    } catch(MyException e) { // 独自の例外が発生する場合がある
      // 例外発生時の処理
      ...

    } finally {
      // リソースの解放
      ...

    }
  }

  private void businessLogic(String label, int index) throws MyException {
    // ビジネスロジック
    ...

  }

  @Override
  public void release() {
    // ビジネスロジックを途中で終了するためのフラグ
    this.isActive = false;
  }

  @Override
  public void taskAccepted(TaskEvent event) {
    // タスク受付時の処理
    ...

  }

  @Override
  public void taskStarted(TaskEvent event) {
    // タスクが開始された時の処理
    ...

  }

  @Override
  public void taskCompleted(TaskEvent event) {
    final Exception exception = event.getException();
    if (exception == null) {
      // ビジネスロジックが問題なく完了した場合の後処理
      ...
```

```
    } else {
      // ビジネスロジック実行時に問題が発生した時の後処理
      ...

    }
  }

  @Override
  public void taskRejected(TaskEvent event) {
    final Exception exception = event.getException();
    if (exception == null) {
      // 例外発生以外の理由によって受付が拒否された場合の処理
      ...

    } else {
      // 例外発生によって受付が拒否された場合の処理
      ...

    }
  }
}
```

*このサンプル* には以下のような特徴があります。

- AbstractTask 抽象クラスを拡張して Task インタフェースを実装している
- 実際のビジネスロジック（businessLogic）では例外（MyException）が発生しているが、 run メソッド内ではthrowしない

コラム

*このサンプル*では、終了通知をした場合releaseメソッドを終了した後でもしばらくrunメソッドが実行されたままである可能性があります。

終了通知を指示する時にタスクメッセージを再登録するよう指定していた場合、同じタスクが同時に動作する可能性があるので注意してください。

## 終了通知（Javaで開発する場合）

Javaでタスクを実装した場合、ビジネスロジック（runメソッド）が実行された後で、その処理が完了する前であっても、終了通知によってその処理を途中でも終了するように通知されることがあります。

タスクの実装者は終了通知に関連して以下のことを考慮する必要があります。

- *releaseメソッドの実装*
- *再登録時の制御*

### releaseメソッドの実装

タスクは、runメソッドの実行時にタスク管理アプリケーションからreleaseメソッドを呼び出されることがあります。 タスクの開発者は、releaseメソッドが呼び出されたら、実行中のrunメソッドをできるだけ速やかに終了させるような実装をしてください。

*releaseの実装例*にreleaseメソッドの実装例を示します。

#### releaseの実装例

```
package sample.mytest;

import java.util.List;
import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class MyTask extends AbstractTask {

  private volatile boolean releaseNotified;

  public MyTask() {
    this.releaseNotified = false;
  }

  @Override
  public void run() {
    // 処理用データの取得
    List<String> list = ...;

    for (String item : list) {
      // itemに対する処理
      doSomething(item);

      // releaseが呼ばれていた場合は途中でも終了
      if (this.releaseNotified) {
        break;
      }
    }
  }

  @Override
  public void release() {
    this.releaseNotified = true;
  }

  private void doSomething(String item) {
    ...
  }
}
```

runメソッドではループ処理を行い、すべてのデータ（item）に対して処理を行うとこのメソッドを終了します。

*releaseの実装例*では、releaseメソッドが呼ばれるとreleaseNotifiedがtrueに設定されます。

runメソッドではreleaseNotifiedを常に観察し、この値がtrueになれば途中でもループから抜けてrunを終了します。

> **注意**
>
> 非同期処理機能はreleaseメソッドの呼び出しが完了すると即座に同タスクを管理対象外とします。
>
> 一方、タスクのrunメソッドはreleaseメソッドが呼び出されても即座に終了するとは限りません。
>
> これはタスクメッセージを再登録する場合に問題になることがあります。 詳細は*再登録時の制御*を参照してください。

### 再登録時の制御

TaskManager#releaseRunning*XXXX*TaskメソッドJava）、またはWorkManager#releaseRunning*XXXX*Task関数（サーバサイドJavaScript）を呼び出すとき、終了通知をしたタスクをタスクキューに再登録するかどうかを第2引数で指定することができます。

タスクメッセージを再登録すると、タイミングによってはタスクのrunが同時に重複して実行される可能性があります。 このケースに該当するシナリオを*releaseの実装例*を実行している状況で考えます。 *再登録時に重複して実行される場合*も参照してください。

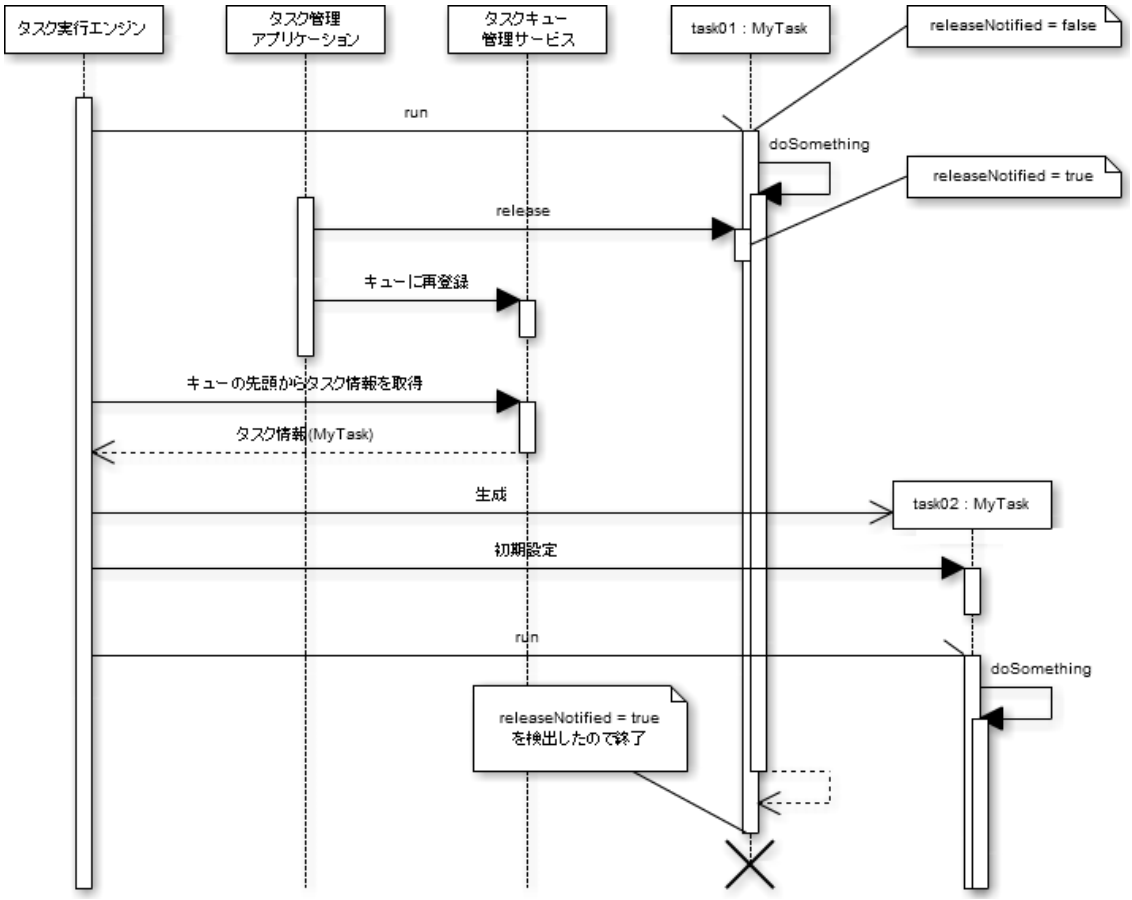
1. *releaseの実装例*のdoSomethingで長時間処理をしている間にタスク管理アプリケーションから終了通知を行うよう要求します。この時、タスクメッセージを再登録するものとします。
2. 該当するタスクのreleaseが呼び出され、releaseNotifiedがtrueに設定されます。
   この時、doSomethingが実行中であるため、runメソッドがまだ完了していないものとします。 つまり、一時的に「releaseメソッドの呼び出しが完了してタスクが管理対象外となったが、runメソッドの実行が終了していない」という状態です。

3. releaseメソッドの呼び出しが完了すると、非同期処理機能は即座に同タスクを管理対象外とします。
   ここでは単に「管理対象外」となるだけであり、runメソッドがまだ完了していないものとします。

4. タスク管理アプリケーションはreleaseの呼び出しが完了するとタスクメッセージをタスクキューの先頭に再登録します。
   この時点でも先に実行中であったタスクのrunメソッドがまだ完了していないものとします。

5. タスク実行エンジンがタスクキューの先頭をチェックし、タスクメッセージを取得し、タスクを生成します。
   この場合のタスクは、現在実行中のタスクとは別のインスタンスです。

6. タスク実行エンジンは新しく生成されたタスクのrunメソッドを非同期で実行開始します。
   この時点で、同じタスクメッセージから生成された2つの異なるタスクのrunメソッドが重複して実行されます。




再登録時に重複して実行される場合

ビジネスロジック（タスクのrunメソッド）が重複して実行されると問題がある場合、上記の点を考慮する必要があります。 この現象を回避する方法としては以下のようなことが考えられます。

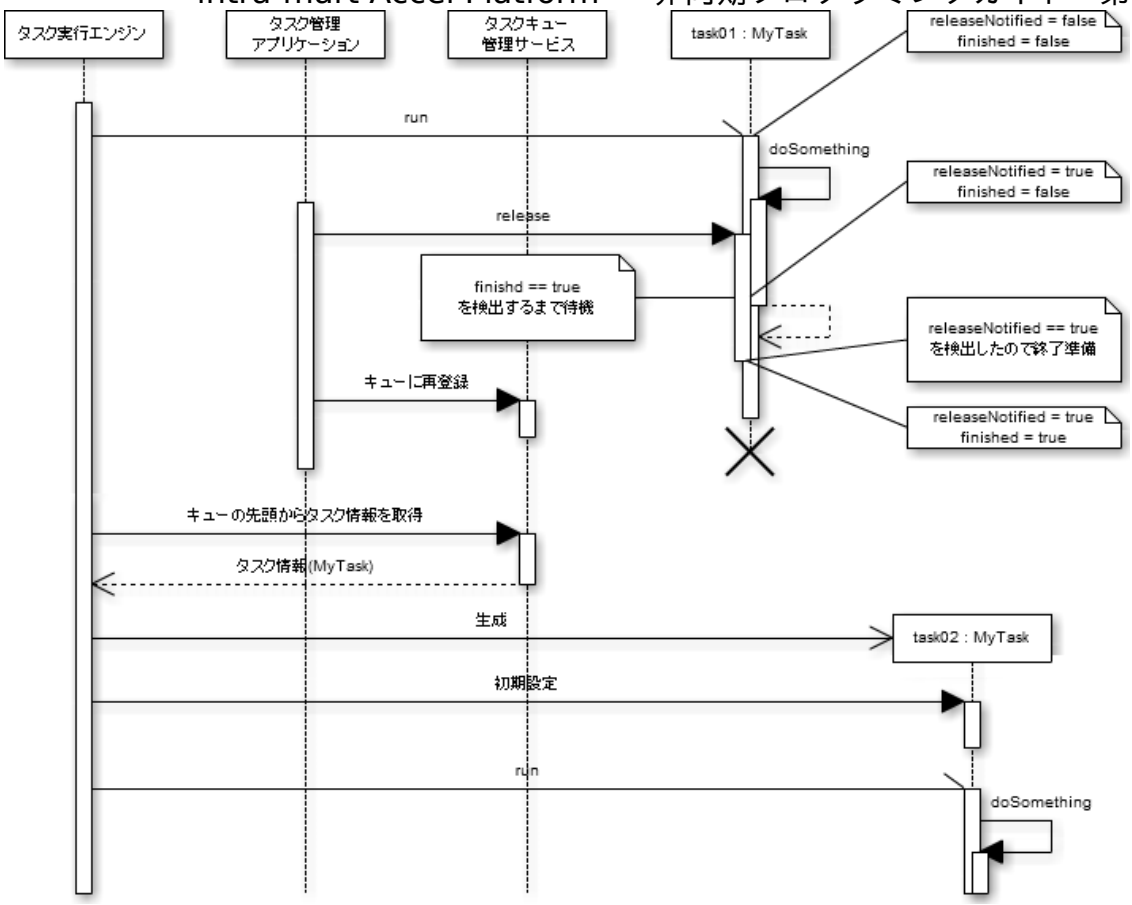
- 終了通知 を行うとき、タスクメッセージの再登録をしない
  タスクキューの状態はTaskManager#releaseRunning*XXXX*Taskメソッド（Java）、またはWorkManager#releaseRunning*XXXX*Task関数（サーバサイドJavaScript）を呼び出すときの第2引数を必ずfalseで呼び出すことが前提です。
- タスクキューを無効状態にする
  タスクキューが無効状態である場合、タスクキューにタスクメッセージが再登録されてもタスク実行エンジンによる実行対象とはなりません。
  タスクキューの状態はTaskManager#releaseRunning*XXXX*Taskメソッド（Java）、またはWorkManager#releaseRunning*XXXX*Task関数（サーバサイドJavaScript）を呼び出すときの第3引数で制御できます。 詳細は非同期処理機能の仕様書を参照してください。
- タスクのreleaseメソッドの実行終了をrunメソッドが終了するまでブロックする
  releaseの内部ではrunメソッドをできるだけ早く終了するような処理を行うだけでなく、実際にrunメソッドが完了するまで待機します。

最後の方法を採用する場合の例を*releaseの実装例2*に示します。 *release時にビジネスロジックが終了準備ができるまで待機する場合*も参照してください。

releaseの実装例2

```java
package sample.mytest;

import java.util.List;
import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class MyTask2 extends AbstractTask {

  private volatile boolean releaseNotified;

  private volatile boolean finished;

  public MyTask2() {
    this.releaseNotified = false;
    this.finished = false;
  }

  @Override
  public void run() {
    try {
      // 処理用データの取得
      List<String> list = ...;

      for (String item : list) {
        // itemに対する処理
        doSomething(item);

        // releaseが呼ばれていた場合は途中でも終了
        if (this.releaseNotified) {
          break;
        }
      }
    } finally {
      this.finished = true;
    }
  }

  @Override
  public void release() {
    this.releaseNotified = true;
    while (!this.finished) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        //
      }
    }
  }

  private void doSomething(String item) {
    ...
  }
}
```

*releaseの実装例2*では、runメソッドの終了時に必ずインスタンス変数finishedをtrueにしています。 releaseメソッドではこの値を1秒ごとに監視し、finishedがtrueになるまで待機します。 finishedがtrueになった場合はrunメソッド内でビジネスロジックが完了したことが確認できたと判断し、releaseメソッドも終了します。




release時にビジネスロジックが終了準備ができるまで待機する場合

> **コラム**
> *releaseの実装例2*ではThread#sleepを使用して待機していますが、実際にはパフォーマンス等を考慮し、java.lang.Object#waitとjava.lang.Object#notifyによるロックや、java.util.concurrent.locksパッケージのロック用ユーティリティ等を利用する方法なども考えられます。

> **コラム**
> 厳密には、*release時にビジネスロジックが終了準備ができるまで待機する場合*の場合であってもtask01とtask02のrunメソッドが同時に実行されている可能性もあります。 しかしながら、task01のビジネスロジックが完了（ここではdoSomethingメソッドがもはや呼び出されない状態）した後にtask02のインスタンスが生成されているので、ビジネスロジックそのものは競合しません。

### 単一実行の強制（直列タスクキュー）

*再登録時の制御*でも述べたように、非同期処理機能はreleaseメソッドの呼び出しが完了すると即座に同タスクを管理対象外とします。この場合、ビジネスロジックの実行状況については無視されるため、タスクメッセージの再登録を指定していない場合は後続のタスクが一時的に同時に実行される場合があります。

タスクメッセージを直列タスクキューに登録している状況でこのような状況を防ぎたい場合は*再登録時の制御*で述べた内容と同様にreleaseでビジネスロジックの完了までブロックするなどの工夫が必要です。

## 情報の受け渡し

非同期処理機能ではタスクメッセージの登録時とタスクの実行時では環境が以下のように異なっています。

- タスク登録アプリケーションはタスクメッセージをタスクキューに登録するが、ビジネスロジックを実行するタスクは直接生成されない（*遅延実行*を参照）
- スレッドが異なる（*非同期による実行*を参照）
- 実行するサーバが異なる（*リモートによる実行*を参照）

そのため、通常のメソッド（または関数）呼び出しとは異なり、呼び出し元から呼び出し先へ直接変数情報を渡すことができません。

登録時に決定される情報をタスクに渡す方法はいくつかあります。 以下に代表的なものを記します。

- *パラメータによる情報の受け渡し*
- *コンテキストによる情報の受け渡し*
- *データベースによる情報の受け渡し*
- *ストレージサービスによる情報の受け渡し*

### パラメータによる情報の受け渡し

タスクメッセージの登録時に、開発者が定義したパラメータを渡すことができます。

設定できるパラメータの形式には制限があるため、登録時と実行時ではそれぞれ変換が必要となる場合があります。

*パラメータを利用するタスクの例*および*パラメータを利用するタスクメッセージ登録の例*にパラメータを利用して情報を受け渡す例を示します。

パラメータの詳細については*パラメータ*を参照してください。

パラメータを利用するタスクの例

```
package sample.mytest;

import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class ParameterTask extends AbstractTask {

  @Override
  public void run() {
    Map<String, ?> parameter = getParameter();
    String name = (String) parameter.get("NAME");
    int age = ((Number) parameter.get("AGE")).intValue();
    businessLogic(name, age);
  }

  private void businessLogic(String name, int age) {
    ...
  }
}
```

パラメータを利用するタスクメッセージ登録の例

```
package sample.mytest;

import java.util.HashMap;
import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;

public class ParameterTaskRegister {
  ...

  private void addTask() {
    Map<String, Object> parameter = new HashMap<String, Object>();
    parameter.put("NAME", "intra-mart");
    parameter.put("AGE", 26);
    try {
      TaskManager.addParallelizedTask("sample.mytest.ParameterTask", parameter);
    } catch (TaskControlException e) {
      ...
    }
  }
  ...
}
```

*パラメータを利用するタスクの例*においてParameterTaskクラスはAbstractTask抽象クラスを継承しています。 そのため、非同期処理機能からsetParameterが呼ばれています。 この状態であれば、getParameterメソッドによって*パラメータを利用するタスクメッセージ登録の例*で登録された値を取得することができます。

> **注意**
> *パラメータを利用するタスクの例*と*パラメータを利用するタスクメッセージ登録の例*で数値型を受け渡していますが、登録時と取得時の方法が異なることに注意してください。 詳細については非同期処理機能の仕様書、または*パラメータ*を確認してください。

### コンテキストによる情報の受け渡し

タスクメッセージの登録時には、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。 保存されたコンテキストはrunメソッド（Java使用時）またはrun関数（サーバサイドJavaScript使用時）に取得および利用することができます。

コンテキストの詳細については関連する仕様書やAPIの説明を参照してください。

*コンテキストを利用するタスクの例*および*コンテキストを利用するタスクメッセージ登録の例*にコンテキストを利用して情報を受け渡す例を示します。

#### コンテキストを利用するタスクの例

```
package sample.mytest;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;
import jp.co.intra_mart.foundation.context.Contexts;
import jp.co.intra_mart.foundation.context.model.AccountContext;
import jp.co.intra_mart.foundation.context.model.UserType;

public class ContextTask extends AbstractTask {

  @Override
  public void run() {
    // 登録時の Context を利用可能
    AccountContext accountContext = Contexts.get(AccountContext.class);
    UserType userType = accountContext.getUserType();
    String userCd = accountContext.getUserCd();
    businessLogic(userCd);
  }

  private void businessLogic(String userCd) {
    ...
  }
}
```

コンテキストを利用するタスクメッセージ登録の例

```
package sample.mytest;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;
import jp.co.intra_mart.foundation.context.Contexts;
import jp.co.intra_mart.foundation.context.model.AccountContext;
import jp.co.intra_mart.foundation.context.model.UserType;

public class ContextTaskRegister {

  private void addTask() {
    // Context の情報を明示的に取得や指定しなくても、
    // 現在の Context が自動的にタスクキューに登録される

    // AccountContext accountContext = Contexts.get(AccountContext.class);
    // UserType userType = accountContext.getUserType();
    // String userCd = accountContext.getUserCd();

    try {
      TaskManager.addParallelizedTask("sample.mytest.ContextTask", null);
    } catch (TaskControlException e) {
      ...
    }
  }
}
```

*コンテキストを利用するタスクメッセージ登録の例*を実行すると、現在のコンテキストも含んだ状態でタスクメッセージを登録します。

そのため、コンテキストはタスクメッセージ登録時とタスク実行時では同じ物が取得できます。

*コンテキストを利用するタスクの例*では`AccountContext#getUserCd`によってユーザコードを、`AccountContext#getUserType`によってユーザタイプを取得しています。 この場合、*コンテキストを利用するタスクメッセージ登録の例*でタスクメッセージを登録した時点の値が反映されています。

### データベースによる情報の受け渡し

非同期処理機能からタスクの`run`メソッドが呼ばれた場合、`run`メソッド内部では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。 （詳細は非同期処理機能の仕様書を参照してください。）

*データベースを利用するタスクの例*および*データベースを利用するタスクメッセージ登録の例*にデータベースを利用して情報を受け渡す例を示します。

このサンプルでは、あらかじめ*テーブル構造*に示す構造を持つテーブルがテナントデータベースに登録されているものとします。

**テーブル構造**

| 項目 | 型 | プライマリキー | 説明 |
|---|---|---|---|
| ID | 整数 | ○ | ID |
| NAME | 文字列型 | — | 名前 |
| AGE | 整数 | — | 年齢 |

データベースを利用するタスクの例

```
package sample.mytest;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;
import jp.co.intra_mart.foundation.database.DBTransaction;
import jp.co.intra_mart.foundation.database.DatabaseUtil;
import jp.co.intra_mart.foundation.database.TenantDatabase;
import jp.co.intra_mart.foundation.database.exception.DatabaseException;

public class DatabaseTask extends AbstractTask {

  @Override
  public void run() {
    DBTransaction transaction = new DBTransaction();
    boolean success = false;
    try {
      transaction.beginTransaction();
      try {
        TenantDatabase database = new TenantDatabase();
        Connection connection = database.getConnection();
        try {
          String sql = "select name, age from my_test";
          PreparedStatement statement = connection.prepareStatement(sql);
          try {
            ResultSet resultSet = statement.executeQuery();
            try {
              while (resultSet.next()) {
                String name = resultSet.getString("name");
                int age = resultSet.getInt("age");
                businessLogic(name, age);
              }
            } finally {
              DatabaseUtil.closeResultSetQuietly(resultSet);
            }
          } finally {
            DatabaseUtil.closeStatementQuietly(statement);
          }
        } finally {
          DatabaseUtil.closeConnectionQuietly(connection);
        }
        transaction.commit();
        success = true;
      } finally {
        if (!success) {
          try {
            transaction.rollback();
          } catch (DatabaseException e) {
            ...
          }
        }
      }
    } catch (DatabaseException e) {
      ...
    } catch (SQLException e) {
      ...
    }
  }

  private void businessLogic(String name, int age) {
    // ...
  }
}
```

データベースを利用するタスクメッセージ登録の例

```
package sample.mytest;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;
import jp.co.intra_mart.foundation.database.DBTransaction;
import jp.co.intra_mart.foundation.database.DatabaseUtil;
import jp.co.intra_mart.foundation.database.TenantDatabase;
import jp.co.intra_mart.foundation.database.exception.DatabaseException;

public class DatabaseTaskRegister {
  ...

  private void addTask() {
    DBTransaction transaction = new DBTransaction();
    boolean success = false;
    try {
      transaction.beginTransaction();
      try {
        TenantDatabase database = new TenantDatabase();
        Connection connection = database.getConnection();
        try {
          for (int index = 0; index < 5; index++) {
            String sql = "insert into my_test (id, name, age) values (?, ?, ?)";
            PreparedStatement statementInsert = connection.prepareStatement(sql);
            try {
              statementInsert.setInt(1, index);
              statementInsert.setString(2, "intra-mart " + index);
              statementInsert.setInt(3, 26 + index);
              statementInsert.execute();
            } finally {
              try {
                statementInsert.close();
              } catch (SQLException e) {
                ...
              }
            }
          }
        } finally {
          DatabaseUtil.closeConnectionQuietly(connection);
        }
        transaction.commit();
        success = true;
      } finally {
        if (!success) {
          try {
            transaction.rollback();
          } catch (DatabaseException e) {
            ...
          }
        }
      }
    } catch (DatabaseException e) {
      ...
    } catch (SQLException e) {
      ...
    }

    try {
      TaskManager.addParallelizedTask("sample.mytest.DatabaseTask", null);
    } catch (TaskControlException e) {
      ...
    }
  }
```

```
  ...
}
```

### ストレージサービスによる情報の受け渡し

非同期処理機能からタスクのrunメソッドが呼ばれた場合、runメソッド内部ではストレージサービスを利用することが可能です。（詳細は非同期処理機能の仕様書を参照してください。）

*ストレージサービスを利用するタスクの例*および*ストレージサービスを利用するタスクメッセージ登録の例*にストレージサービスを利用して情報を受け渡す例を示します。

このサンプルでは、あらかじめ%PUBLIC_STORAGE_PATH%/task/sampleディレクトリが存在しているものとします。

ストレージサービスを利用するタスクの例

```
package sample.mytest;

import java.io.IOException;
import java.nio.charset.Charset;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;
import jp.co.intra_mart.foundation.service.client.file.PublicStorage;

public class StorageTask extends AbstractTask {

  @Override
  public void run() {
    PublicStorage storage = new PublicStorage("/task/sample/test.txt");
    String text = null;
    try {
      text = storage.read(Charset.forName("UTF-8"));
    } catch (IOException e) {
      ...
    }
    businessLogic(text);
  }

  private void businessLogic(String text) {
    ...
  }
}
```

ストレージサービスを利用するタスクメッセージ登録の例

```
package sample.mytest;

import java.io.IOException;
import java.nio.charset.Charset;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;
import jp.co.intra_mart.foundation.service.client.file.PublicStorage;

public class StorageTaskRegister {
  ...

  private void addTask() {
    String content = "name:intra-mart,age:26";
    PublicStorage storage = new PublicStorage("/task/sample/test.txt");

    try {
      storage.write(content, Charset.forName("UTF-8"));
    } catch (IOException e) {
      ...
    }

    try {
      TaskManager.addParallelizedTask("sample.mytest.StorageTask", null);
    } catch (TaskControlException e) {
      ...
    }
  }
  ...
}
```

# タスクメッセージ登録処理

タスクメッセージをタスクキューに登録するとタスクがいずれ実行される状態に変わります。

非同期処理機能においてタスクメッセージを登録可能なタスクキューは以下の2種類があります。

- 並列タスクキュー
  個々のタスクが独立し、互いの処理が実行順序や並列実行に依存しない場合に利用します。
- 直列タスクキュー
  複数のタスク間に実行順序の依存関係がある場合に利用します。

タスクキューの詳細については非同期処理機能の仕様書を参照してください。

## タスクキューに対するタスクメッセージの登録（共通事項）

タスクキューにタスクメッセージを登録する前に、タスクのビジネスロジック実行時に必要な情報がすべて準備されていることを確認してください。

一度タスクキューにタスクメッセージを登録すると、非同期処理機能は可能な限り早い段階でタスクのビジネスロジックの実行を開始します。 そのため、ビジネスロジックの実行時に必要となる情報を、タスクキューにタスクメッセージを登録した後に追加しようとしても、先にビジネスロジックの実行が完了してしまう場合も考えられます。

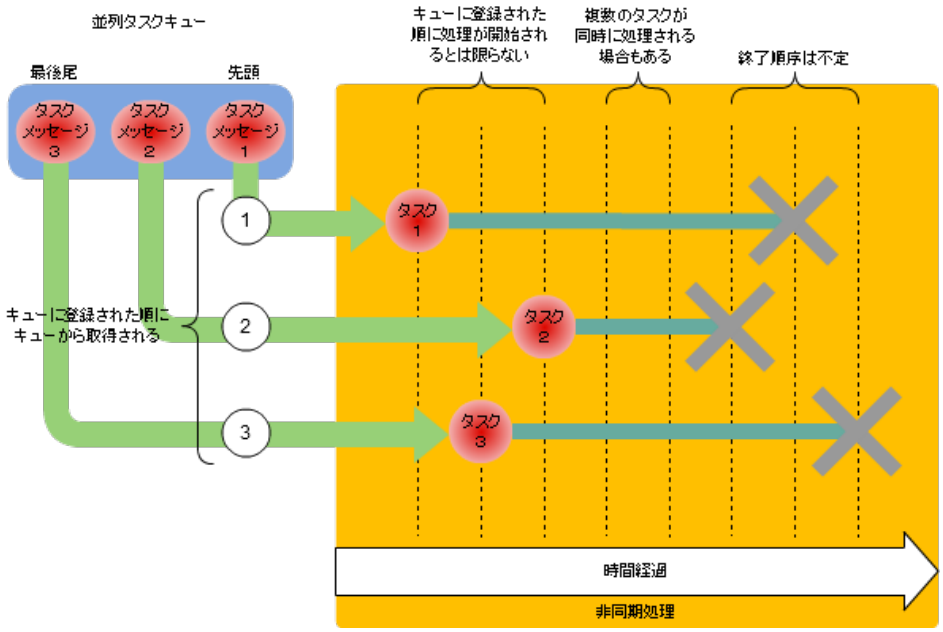
ビジネスロジック実行時に必要な情報はタスクの実装に依存します。 代表的な例として、以下のようなものがあります。

- データベースへの登録
- コンテキストへのデータ保存
- ストレージサービスに対するファイル出力

## 並列タスクキューに対するタスクメッセージの登録

並列タスクキューは、個々のタスクが独立し、互いの処理が実行順序や並列実行に依存しない場合に利用します。

- 並列タスクキューに登録されたタスクメッセージは先頭から順番に取り出されますが、その順番で開始されるとは限りません。
- タスクの終了順序はタスクメッセージの登録順序は異なります。
- 複数のタスクが同時に実行される場合があります。




並列タスクキュー

並列タスクキューに対するタスクメッセージの登録方法は*並列タスクキューに対するタスクメッセージの登録*に示すとおりです。

並列タスクキューに対するタスクメッセージの登録

| 言語 | API |
| --- | --- |
| Java | jp.co.intra_mart.foundation.asynchronous.TaskManager#addParallelizedTask |
| サーバサイドJavaScript | WorkManager#addParallelizedTask |

これらのAPIの詳細については非同期処理機能の仕様書を参照してください。

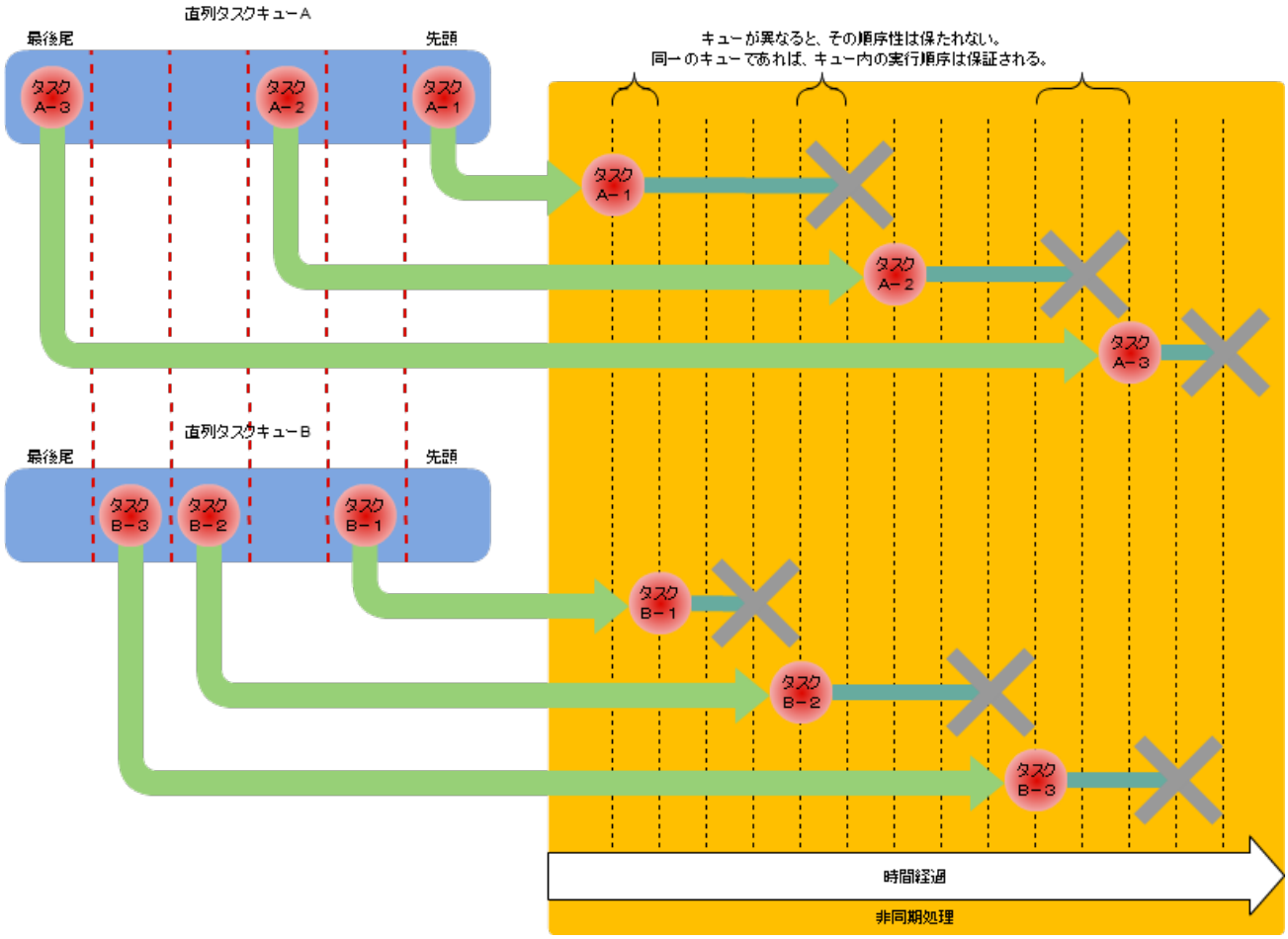
## 直列タスクキューに対するタスクメッセージの登録

直列タスクキューは、複数のタスク間に実行順序の依存関係がある場合に利用します。

- 同一のタスクキュー内のタスクであれば、実行順序は保証されます。
- 直前のタスクが完了するまで、後続のタスクはブロックされます。

> **注意**
>
> 実行中のタスクに対して終了通知をした場合、非同期処理機能は該当するタスクのreleaseメソッドを呼び出した後、直ちに次の処理を行います。 この時、実際のビジネスロジックの実行状態は無視されます。 つまり、同一タスクキューにあるタスクであっても、一時的に連続する2つのタスクが同時に実行される場合があります。
>
> このような状況を避けたい場合、releaseメソッド内でビジネスロジックの完了を待機するような実装をしてください。 詳細は*単一実行の強制（直列タスクキュー）*を参照してください。

- タスクキューが異なるタスクは同時に実行される場合があります。




直列タスクキュー

直列タスクキューに対するタスクメッセージの登録方法は*直列タスクキューに対するタスクメッセージの登録*に示すとおりです。

**直列タスクキューに対するタスクメッセージの登録**

| 言語 | API |
| --- | --- |
| Java | jp.co.intra_mart.foundation.asynchronous.TaskManager#addSerializedTask |
| サーバサイドJavaScript | WorkManager#addSerializedTask |

これらのAPIの詳細については非同期処理機能の仕様書を参照してください。

## 例外発生時

直列タスクキューに登録された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、後続のタスクの処理を行わず、直列タスクキューを停止する必要があると仮定します。

この場合、タスクメッセージの登録時にTaskManager#addSerializedTaskメソッドおよびWorkManager#addSerializedTask関数の第4引数にtrueを指定します。

# パラメータ

ここでは、*タスクの実装*および*タスクメッセージ登録処理*で簡単に説明したパラメータの受け渡しにおける制限ついて説明します。

## Java

### 制限

Javaでパラメータを受け渡す場合、*以下*のような制限があります。 パラメータの制限については非同期処理機能の仕様書を参照してください。

- 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のクラスはこの表で示されているもののいずれかです。 |
| 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。<br><br>value に指定できる値はのクラスはこの表で示されているもののいずれかです。 |

### 数値の制限

パラメータに数値を指定する場合、登録時と取得時ではその設定方法が異なります。

詳細については非同期処理機能の仕様書を参照してください。

登録時は*パラメータとして使用できるインスタンス(Java)*に示す型で登録しますが、取得時はjava.lang.Numberの該当するメソッドで取得する必要があります。

**パラメータとして使用できるインスタンス(Java)**

| 該当するプリミティブ型 | 登録時に使用する型 | 取得時の方法 |
|---|---|---|
| byte | java.lang.Byte | java.lang.Number#byteValue |
| short | java.lang.Short | java.lang.Number#shortValue |
| int | java.lang.Integer | java.lang.Number#intValue |
| long | java.lang.Long | java.lang.Number#longValue |

| 該当するプリミティブ型 | 登録時に使用する型 | 取得時の方法 |
|---|---|---|
| float | java.lang.Float | java.lang.Number#floatValue |
| double | java.lang.Double | java.lang.Number#doubleValue |

パラメータに対して数値を登録する場合の例を*パラメータに数値を登録する場合*に、パラメータに設定されている数値を取得する場合の例を*パラメータに登録されている数値を取得する場合*に示します。

パラメータに数値を登録する場合

```
package sample.task;

import java.util.HashMap;
import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;

public class NumberParameterRegister {
  ...

  public void register() throws TaskControlException {
    Map<String, Object> parameter = new HashMap<String, Object>();
    parameter.put("BYTE", Byte.valueOf((byte) 123));
    parameter.put("SHORT", Short.valueOf((short) 12345));
    parameter.put("INTEGER", Integer.valueOf(123456789));
    parameter.put("LONG", Long.valueOf(1234567890L));
    parameter.put("FLOAT", Float.valueOf(123.45F));
    parameter.put("DOUBLE", Double.valueOf(123.456789));

    TaskManager.addParallelizedTask("sample.task.TestTask", parameter);
  }
  ...

}
```

パラメータに登録されている数値を取得する場合

```
package sample.task;

import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class NumberParameterTask extends AbstractTask {
  ...

  @override
  public void run() {
    Map<String, ?> parameter = getParameter();

    // 直接 java.lang.Byte や java.lang.Integer 等にキャストしないこと
    byte byteVal = ((Number) parameter.get("BYTE")).byteValue();
    short shortVal = ((Number) parameter.get("SHORT")).shortValue();
    int intVal = ((Number) parameter.get("INTEGER")).intValue();
    long longVal = ((Number) parameter.get("LONG")).longValue();
    float floatVal = ((Number) parameter.get("FLOAT")).floatValue();
    double doubleVal = ((Number) parameter.get("DOUBLE")).doubleValue();
  }
  ...

}
```

注意

パラメータから数値型を取得する場合、java.lang.Integerやjava.lang.Doubleなどの具象クラスにキャストしないようにしてください。 数値型をjava.lang.Number以外の型にキャストすると例外が発生する場合があります。

*パラメータに登録されている数値を取得する場合*の代わりに以下のような実装をすると例外が発生する場合があります。

```
Map<String, ?> parameter = getParameter();

// 直接 java.lang.Byte や java.lang.Integer 等にキャストすると例外が発生する可能性がある
byte byteVal = (Byte) parameter.get("BYTE");
short shortVal = (Short) parameter.get("SHORT");
int intVal = (Integer) parameter.get("INTEGER");
long longVal = (Long) parameter.get("LONG");
float floatVal = (Float) parameter.get("FLOAT");
double doubleVal = (Double) parameter.get("DOUBLE");
```

注意

*パラメータに数値を登録する場合*および*パラメータに登録されている数値を取得する場合*では、ソースコードの説明を簡略化するためにnullが設定された場合が考慮されていません。 実際の開発時には、必要に応じてnullが設定された場合にも対応するような実装をしてください。

java.util.Listの制限

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。

詳細については非同期処理機能の仕様書を参照してください。

登録時はjava.util.Listを実装した型（例：java.util.ArrayList、java.util.LinkedList等）で登録しますが、取得時はjava.util.Listにキャストする必要があります。

パラメータに対してjava.util.Listを登録する場合の例を*パラメータにListを登録する場合*に、パラメータに設定されているjava.util.Listを取得する場合の例を*パラメータに登録されているListを取得する場合*に示します。

パラメータにListを登録する場合

```
package sample.task;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;

public class ListParameterRegister {
  ...

  public void register() throws TaskControlException {
    // java.util.List の実装として java.util.ArrayList を使用
    List<Object> list = new ArrayList<Object>();
    list.add(...);
    list.add(...);
    ...
    Map<String, Object> parameter = new HashMap<String, Object>();
    parameter.put("LIST", list);

    TaskManager.addParallelizedTask("sample.mytest.MyTask", parameter);
  }
  ...

}
```

パラメータに登録されているListを取得する場合

```
package sample.task;

import java.util.List;
import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class ListParameterTask extends AbstractTask {
  ...

  @override
  public void run() {
    Map<String, ?> parameter = getParameter();

    // java.util.List にキャストすること
    List<?> list = (List<?>) parameter.get("LIST");
    ...
  }
  ...

}
```

**注意**

パラメータからjava.util.List型のデータを取得する場合、java.util.ArrayListやjava.util.LinkedListなどの具象クラスにキャストしないようにしてください。 たとえ登録時と同じクラスを指定しても、java.util.List以外の型にキャストすると例外が発生する場合があります。

*パラメータに登録されているListを取得する場合*の代わりに以下のような実装をすると例外が発生する場合があります。

```
Map<String, ?> parameter = getParameter();

// java.util.List の実装クラスにキャストすると例外が発生する可能性がある
java.util.ArrayList<Object> list =
  (java.util.ArrayList<Object>) parameter.get("LIST");
```

**注意**

*パラメータにListを登録する場合*および*パラメータに登録されているListを取得する場合*では、ソースコードの説明を簡略化するため、パラメータにnullが設定された場合が考慮されていません。 実際の開発時には、必要に応じてnullが設定された場合にも対応するような実装をしてください。

java.util.Mapの制限

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。

詳細については非同期処理機能の仕様書を参照してください。

登録時はjava.util.Mapを実装した型（例：java.util.HashMap、java.util.TreeMap等）で登録しますが、取得時はjava.util.Mapにキャストする必要があります。

パラメータに対してjava.util.Mapを登録する場合の例を*パラメータにMapを登録する場合*に、パラメータに設定されているjava.util.Mapを取得する場合の例を*パラメータに登録されているMapを取得する場合*に示します。

パラメータにMapを登録する場合

```
package sample.task;

import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;

public class MapParameterRegister {
  ...

  public void register() throws TaskControlException {
    // java.util.Map の実装として java.util.TreeMap を使用
    Map<String, Object> map = new TreeMap<String, Object>();
    map.put("key1", "value1");
    map.put("key2", "value2");
    // ...
    // TreeMap なので key は String#compareTo メソッドに依存した順序となる
    // ex.
    // for (String key : map.keySet()) <- key は "key1", "key2", ... の順に取得される

    Map<String, Object> parameter = new HashMap<String, Object>();
    parameter.put("MAP", map);

    TaskManager.addParallelizedTask("sample.mytest.MyTask", parameter);
  }
  ...

}
```

パラメータに登録されているMapを取得する場合

```
package sample.task;

import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class MapParameterTask extends AbstractTask {
  ...

  @override
  public void run() {
    Map<String, ?> parameter = getParameter();

    // 必ず java.util.Map にキャストすること
    Map<String, ?> map = (Map<String, ?>) parameter.get("MAP");
    String value1 = (String) map.get("key1");
    String value2 = (String) map.get("key2");
    ...

    // 登録時の Map の実装とは異なる場合があるので、 key の順序も異なる場合がある
    // ex.
    // for (String key : map.keySet()) <- key の取得順は不定
    ...
  }
  ...

}
```

注意

パラメータからjava.util.Mapの型を取得する場合、java.util.HashMapやjava.util.TreeMapのようなjava.util.Mapのサブクラスにキャストしないようにしてください。 たとえ登録時と同じクラスを指定しても、java.util.Map以外の型にキャストすると例外が発生する場合があります。

*パラメータに登録されているMapを取得する場合*の代わりに以下のような実装をすると例外が発生する場合があります。

```
Map<String, ?> parameter = getParameter();

// java.util.Map のサブクラスにキャストすると例外が発生する可能性がある
java.util.TreeMap<String, Object> map =
  (java.util.TreeMap<String, Object>) parameter.get("MAP");
```

注意

*パラメータにListを登録する場合*および*パラメータに登録されているListを取得する場合*では、ソースコードの説明を簡略化するため、パラメータにnullが設定された場合が考慮されていません。 実際の開発時には、必要に応じてnullが設定された場合にも対応するような実装をしてください。

ループ構造の禁止

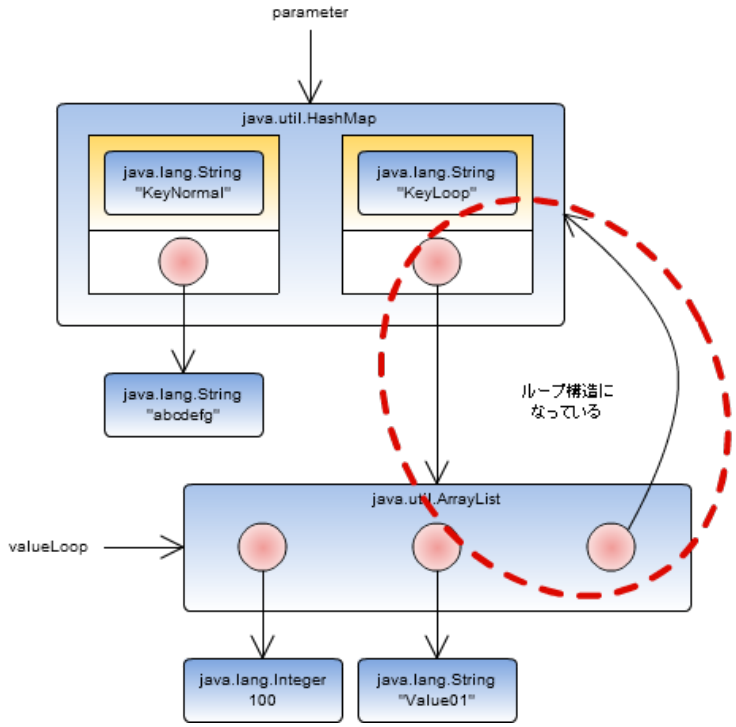
*パラメータとして使用できるインスタンス(Java)*に示す値を使用している場合であっても、ループ構造を含んではいけません。

例えば、*ループ構造を含むパラメータのコード例*に示すようなパラメータはループ構造を含んでいるため、タスクメッセージ登録時に例外が発生します。

ループ構造を含むパラメータのコード例

```
package sample.mytest;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;

public class ParameterExamples {

  private void exampleLoop() {
    Map<String, Object> parameter = new HashMap<String, Object>();
    parameter.put("KeyNormal", "abcdefg");
    List<Object> valueLoop = new ArrayList<Object>();
    valueLoop.add("Value01");
    valueLoop.add(Integer.valueOf(100));
    valueLoop.add(parameter);

    // parameter["KeyLoop"][2] が parameter そのものになる（ループ構造）
    parameter.put("KeyLoop", valueLoop);

    try {
      // parameterにループ構造が含まれるため例外が発生する
      TaskManager.addParallelizedTask("sample.mytest.MyTask", parameter);
    } catch (TaskControlException e) {
      // ...
    }
  }
}
```

*ループ構造を含むパラメータのコード例*に示す構造では、parameterは*ループを含むパラメータ構造*のような構造です。




ループを含むパラメータ構造

### 同値性の保証と同一性

複数の箇所から同一のインスタンスを参照しているようなパラメータの場合、値が同じ別のインスタンスが登録された場合と同様な振る舞いをします。

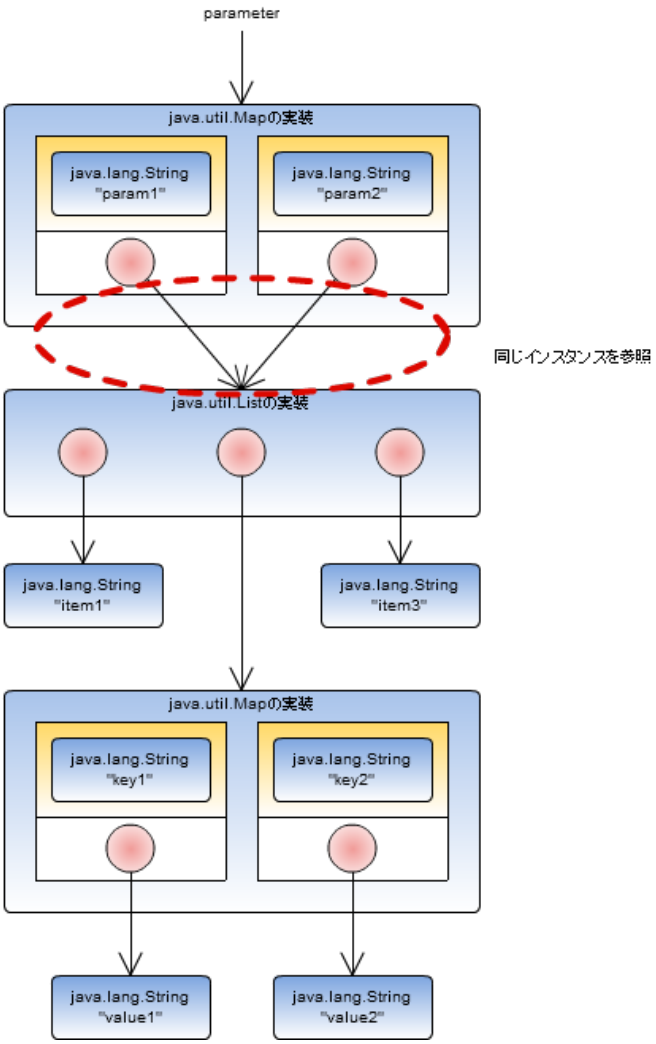
詳細については非同期処理機能の仕様書を参照してください。

*同一のインスタンスへの参照を含むパラメータのコード例（登録時）*に同一のインスタンスへの参照を含むパラメータを登録する例を示します。

同一のインスタンスへの参照を含むパラメータのコード例（登録時）

```
package sample.task;

import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;

public class IdenticalReferenceParameterRegister {
  ...

  public void register() throws TaskControlException {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("key1", "value1");
    map.put("key2", "value2");
    List<Object> list = new ArrayList<Object>();
    list.add("item1");
    list.add(map);
    list.add("item3");
    Map<String, Object> parameter = new HashMap<String, Object>();
    parameter.put("param1", list);
    parameter.put("param2", list);

    TaskManager.addParallelizedTask("sample.mytest.MyTask", parameter);
  }
  ...

}
```

*同一のインスタンスへの参照を含むパラメータのコード例（登録時）*に示したコードでは、*同一インスタンスを参照しているパラメータ（イメージ図）*に示すような構造のパラメータを設定しようとしています。



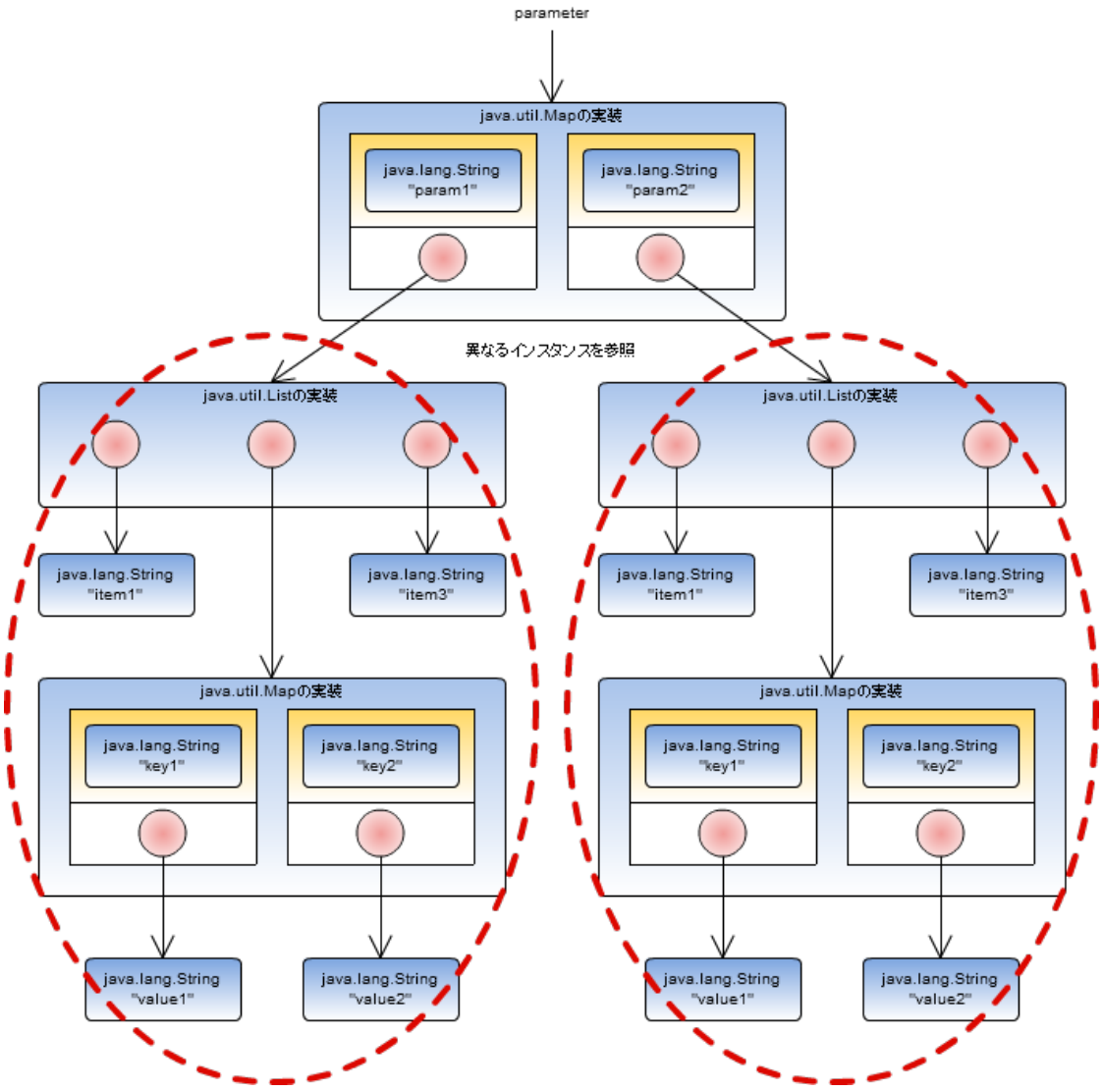
同一インスタンスを参照しているパラメータ（イメージ図）

一方、このパラメータを*同一のインスタンスへの参照を含むパラメータのコード例（取得時）*のように取得することを考えます。

同一のインスタンスへの参照を含むパラメータのコード例（取得時）

```
package sample.task;

import java.util.Map;

import jp.co.intra_mart.foundation.asynchronous.AbstractTask;

public class IdenticalReferenceParameterTask extends AbstractTask {
  ...

  @override
  public void run() {
    Map<String, ?> parameter = getParameter();

    ...
  }
  ...

}
```

*同一のインスタンスへの参照を含むパラメータのコード例（取得時）*に示したタスクのビジネスロジック実行時にパラメータを取得すると、*同一の値を持つインスタンスを参照しているパラメータ（イメージ図）*のような構造のインスタンスです。




同一の値を持つインスタンスを参照しているパラメータ（イメージ図）

*同一インスタンスを参照しているパラメータ（イメージ図）*と*同一の値を持つインスタンスを参照しているパラメータ（イメージ図）*の両者を比べると、以下のことがわかります。

- どちらもparameterから情報をたどった場合、末端に存在する情報の値は等しい
  parameter.get("param1").get(0).equals(parameter.get("param2").get(0))を評価した場合、trueです。
- 参照情報については異なる
  parameter.get("param1").get(0) == parameter.get("param2").get(0)を評価した場合
  - *同一インスタンスを参照しているパラメータ（イメージ図）*の場合はtrueです。つまり、両者は同一のインスタンスを参照しています。
  - *同一の値を持つインスタンスを参照しているパラメータ（イメージ図）*の場合はfalseです。つまり、両者は異なるインスタンスを参照しています。

## サーバサイドJavaScript

### 制限

サーバサイドJavaScriptでパラメータを受け渡す場合、*以下*のような制限があります。 パラメータの制限については非同期処理機能の仕様書を参照してください。

- 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*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。 |
| 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。<br><br>value に指定できる値は*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)*で示されている値のみです。 |

#### 配列の制限

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。

詳細については非同期処理機能の仕様書を参照してください。

パラメータに対して配列を登録する場合の例を*パラメータに配列を登録する場合*に、パラメータに設定されている配列を取得する場合の例を*パラメータに登録されている配列を取得する場合*に示します。

パラメータに配列を登録する場合

```
function register() {
  ...

  let list = [];
  list.push(...);
  list.push(...);

  let parameter = {};
  parameter.LIST = list;

  let workManager = new WorkManager();
  workManager.addParallelizedTask("system/async/test/test_task", parameter);
  ...
}
```

パラメータに登録されている配列を取得する場合

```
let parameter;

function setParameter(request) {
  parameter = request;
}

function run() {
  let list = parameter.LIST;
  ...
}
...
```

### 連想配列の制限

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。

詳細については非同期処理機能の仕様書を参照してください。

パラメータに対して連想配列を登録する場合の例を*パラメータに連想配列を登録する場合*に、パラメータに設定されている連想配列を取得する場合の例を*パラメータに登録されている連想配列を取得する場合*に示します。

パラメータに連想配列を登録する場合

```
function register() {
  ...

  let map = {};
  map.key1 = "value1";
  map.key2 = "value2";
  ...

  let parameter = {};
  parameter.MAP = map;

  let workManager = new WorkManager();
  workManager.addParallelizedTask("system/async/test/test_task", parameter);
  ...
}
```

パラメータに登録されている連想配列を取得する場合

```
let parameter;

function setParameter(request) {
  parameter = request;
}

function run() {
  let map = parameter.MAP;
  let value1 = map.key1;
  let value2 = map.key2;
  ...

  // 登録時の連想配列と key の順序が異なる場合がある
  // ex.
  // for (let key in map) <- key の取得順は不定
  ...
}
...
```

ループ構造の禁止

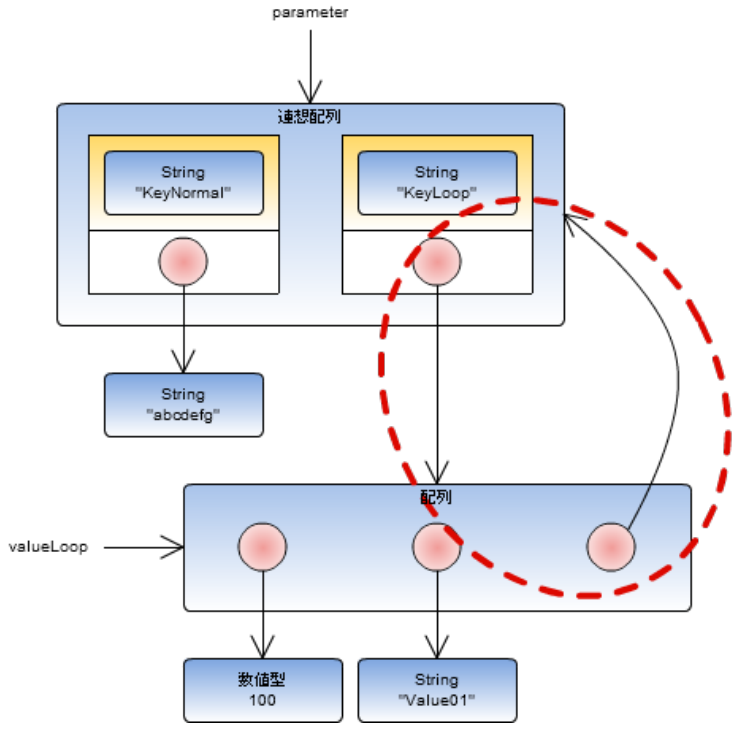
*パラメータとして使用できるインスタンス(サーバサイドJavaScript)*に示す値を使用している場合であっても、ループ構造を含んではいけません。

例えば、*ループ構造を含むパラメータのコード例*に示すようなパラメータはループ構造を含んでいるため、タスクメッセージ登録時に例外が発生します。

ループ構造を含むパラメータのコード例

```
function register() {
  let parameter = {};
  parameter.KeyNormal = "abcdefg";

  let valueLoop = [];
  valueLoop.push("Value01");
  valueLoop.push(100);
  valueLoop.push(parameter);

  // parameter.KeyLoop[2] が parameter そのものになる（ループ構造）
  parameter.KeyLoop = valueLoop;

  let workManager = new WorkManager();
  let result =
    workManager.addParallelizedTask("system/async/test/test_task", parameter);
    // result.error が true となる
}
```

*ループ構造を含むパラメータのコード例*に示す構造では、`parameter`は*ループを含むパラメータ構造*のような構造です。




ループを含むパラメータ構造

### 同値性の保証と同一性

複数の箇所から同一のオブジェクトを参照しているようなパラメータの場合、値が同じ別のオブジェクトが登録された場合と同様な振る舞いをします。

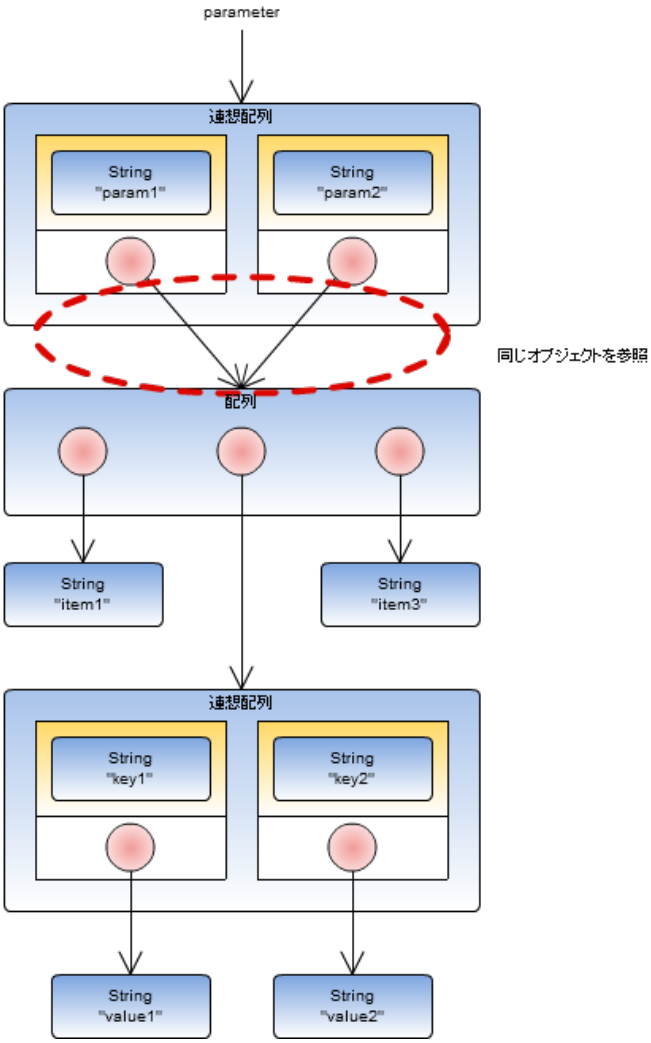
詳細については非同期処理機能の仕様書を参照してください。

*同一のオブジェクトへの参照を含むパラメータのコード例（登録時）*に同一のオブジェクトへの参照を含むパラメータを登録する例を示します。

同一のオブジェクトへの参照を含むパラメータのコード例（登録時）

```
function register() {
  let map = {};
  map.key1 = "value1";
  map.key2 = "value2";

  let list = [];
  list.push("item1");
  list.push(map);
  list.push("item3");

  let parameter = {};
  parameter.param1 = list;
  parameter.param2 = list;

  // parameter.param1 と parameter.param2 は、同一のオブジェクトを参照していることを確認
  parameter.param1[0] = "new_item";
  if (parameter.param1[0] == parameter.param2[0]) {
    // 両者が同一のオブジェクトを参照しているので、上記の条件式は true となる
  } else {
  }

  let workManager = new WorkManager();
  workManager.addParallelizedTask("system/async/test/test_task", parameter);
}
```

*同一のオブジェクトへの参照を含むパラメータのコード例（登録時）*に示したコードでは、*同一インスタンスを参照しているパラメータ（イメージ図）*に示すような構造のパラメータを設定しようとしています。



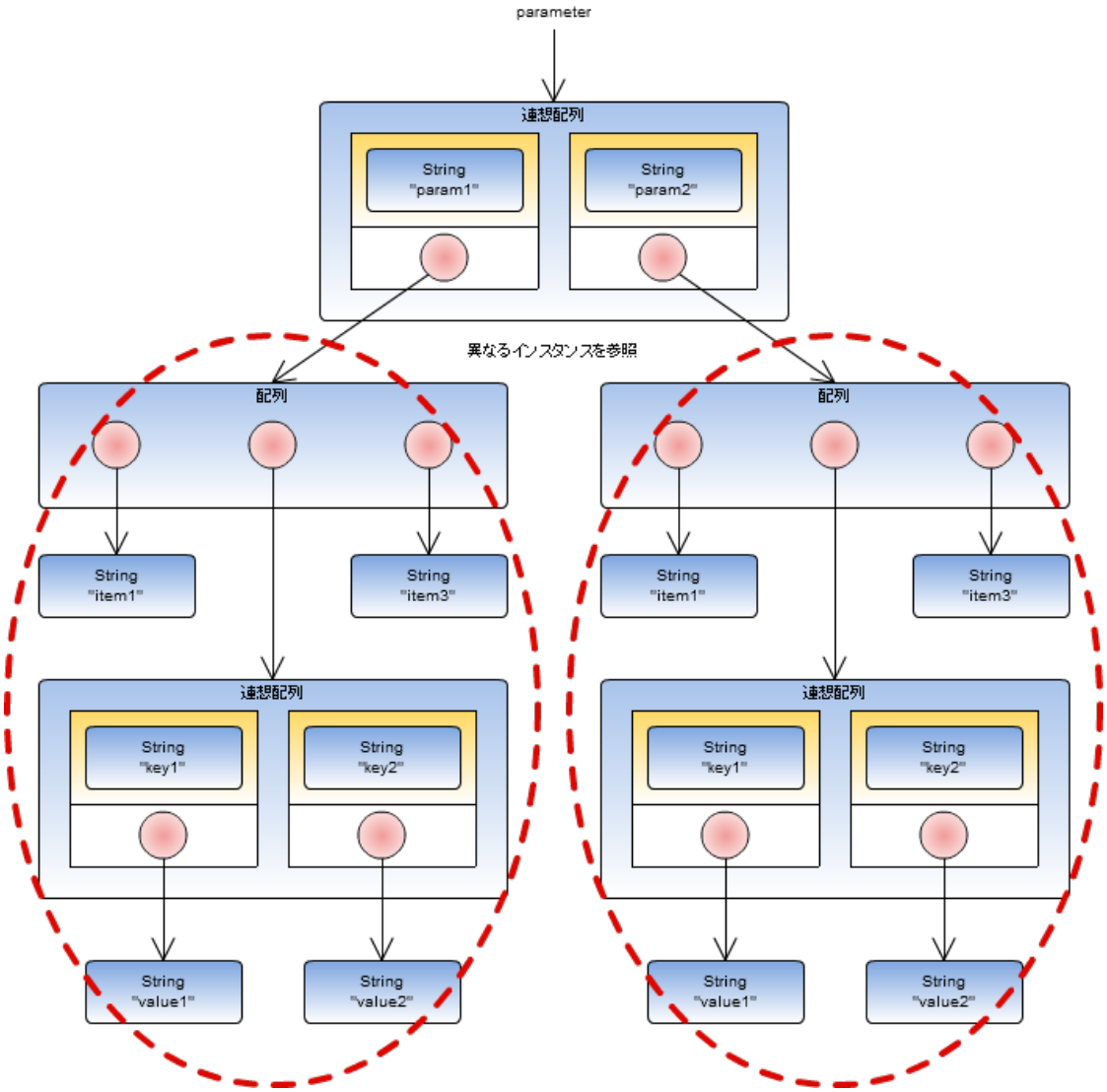

同一インスタンスを参照しているパラメータ（イメージ図）

一方、このパラメータを*同一のオブジェクトへの参照を含むパラメータのコード例（取得時）*のように取得することを考えます。

同一のオブジェクトへの参照を含むパラメータのコード例（取得時）

```
let parameter;

function setParameter(request) {
  parameter = request;
}

function run() {
  // 同値性の検証
  if (parameter.param1[0] == parameter.param2[0]) {
    // 両者は別オブジェクトであるが、値と構造が同じなので、上記の条件式は true となる
  } else {
  }

  // parameter.param1 と parameter.param2 は、異なるオブジェクトを参照していることを確認
  parameter.param1[0] = "modified_item";
  if (parameter.param1[0] == parameter.param2[0]) {
  } else {
    // 両者が異なるオブジェクトを参照しているので、上記の条件式は false となる
  }
  ...
}
...
```

*同一のオブジェクトへの参照を含むパラメータのコード例（取得時）*に示したタスクのビジネスロジック実行時にパラメータを取得すると、*同一の値を持つインスタンスを参照しているパラメータ（イメージ図）*のような構造のインスタンスです。




同一の値を持つインスタンスを参照しているパラメータ（イメージ図）

*同一インスタンスを参照しているパラメータ（イメージ図）*と*同一の値を持つインスタンスを参照しているパラメータ（イメージ図）*の両者を比べると、以下のことがわかります。

- どちらもparameterから情報をたどった場合、末端に存在する情報の値は等しい
  parameter.param1[0] == parameter.param2[0]を評価した場合、trueです。
- 参照情報については異なる
  parameter.param1 == parameter.param2を評価した場合
  - *同一インスタンスを参照しているパラメータ（イメージ図）*の場合はtrueです。つまり、両者は同一のオブジェクトを参照しています。
  - *同一の値を持つインスタンスを参照しているパラメータ（イメージ図）*の場合はfalseです。つまり、両者は異なるオブジェクトを参照しています。

# タスクキューの管理

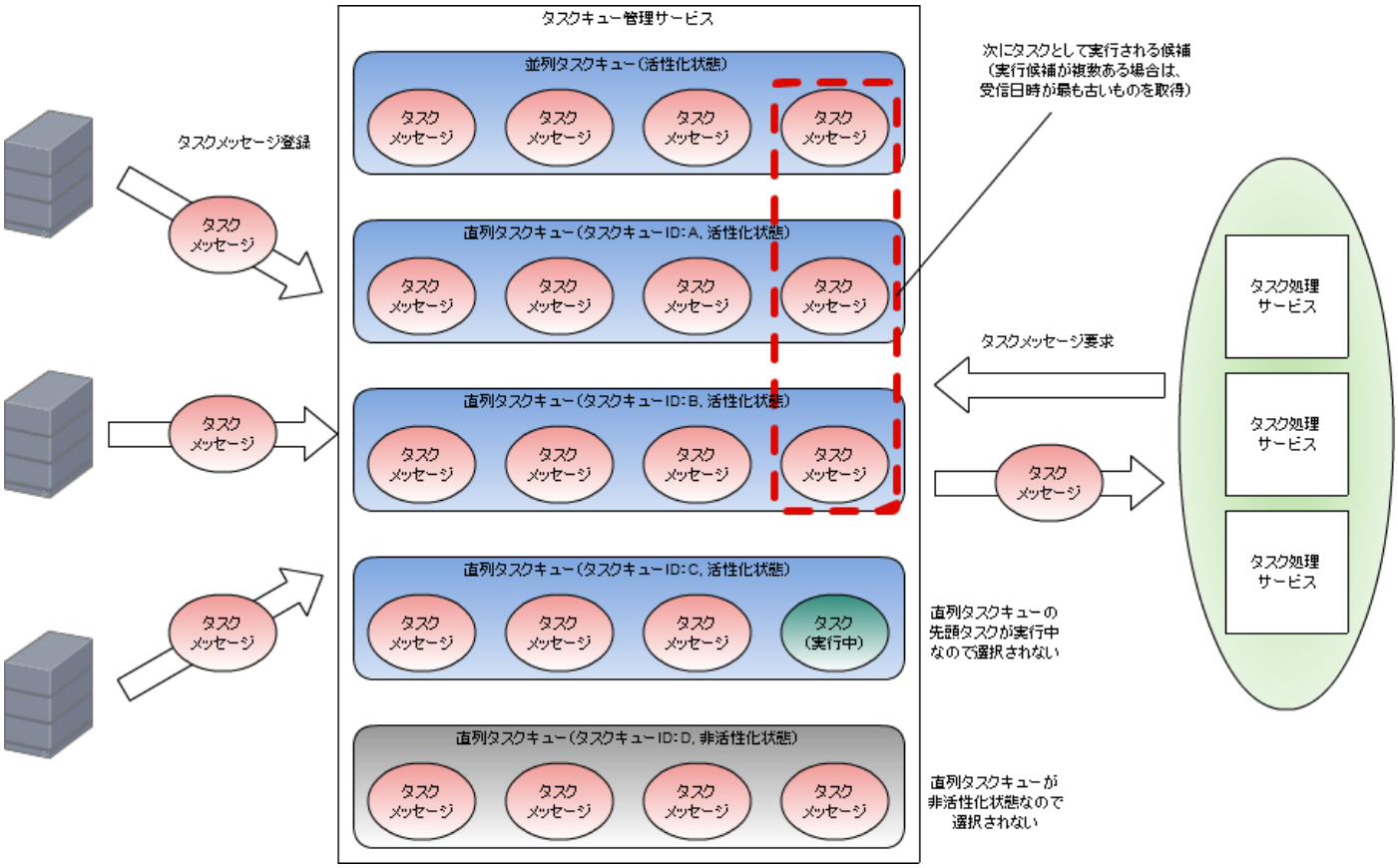
タスクキューは、一時的にタスクメッセージを蓄えておく場所です。

タスクキューに保存されたタスクメッセージは、タスク処理サービスによって、以下の条件に合うものから順に取得されて処理されます。

- 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。

この様子を*タスクの選択*に図示します。




タスクの選択

*タスクメッセージ登録処理*も参照してください。

本章では、タスクキュー自体の管理方法について説明します。

## 並列タスクキューの有効化／無効化

並列タスクキューは任意の時点で以下の状態にすることが可能です。

- *有効状態*
  並列タスクキュー内で待機中のタスクメッセージに対する処理の実行が行われている状態です。
- *無効状態*
  並列タスクキュー内で待機中のタスクメッセージに対する処理の実行を一時停止している状態です。

並列タスクキューの状態を変更するAPIとして*並列タスクキューの状態を変更するAPI*が用意されています。 詳細は非同期処理機能の仕様書を参照してください。

**並列タスクキューの状態を変更するAPI**

| 言語 | API | 備考 |
|---|---|---|
| Java | `TaskManager#setParallelizedTaskQueueActive(boolean active)` | ▪ `active`:<br>▪ `true`: 有効状態にする<br>▪ `false`: 無効状態にする |
| サーバサイドJavaScript | `WorkManager#setParallelizedTaskQueueActive(active)` | ▪ `active`:<br>▪ `true`: 有効状態にする<br>▪ `false`: 無効状態にする |

### 有効状態

有効状態となっている並列タスクキューでは、先頭で待機中のタスクメッセージは常にタスクとして実行される候補となりえます。

### 無効状態

無効状態となっている並列タスクキューでは、待機中のタスクメッセージはタスクとして実行される候補から外されます。 ただし、ビジネスロジックを実行中のタスクについてはそのまま処理が続行されます。

> **注意**
>
> 並列タスクキューを無効状態にする場合は十分注意してください。
>
> 並列タスクキューはテナント毎に1つだけ存在します。 そのため、並列タスクキューを無効状態にすると、並列処理機能として登録されたすべてのタスクメッセージの処理が停止されます。

## 直列タスクキューの有効化／無効化

直列タスクキューは複数存在します。 直列タスクキューの追加または削除は非同期処理機能の管理者によって行われます。

直列タスクキューは任意の時点で以下の状態にすることが可能です。

- *有効状態*
  直列タスクキュー内で待機中のタスクメッセージに対する処理の実行が行われている状態です。
- *無効状態*
  直列タスクキュー内で待機中のタスクメッセージに対する処理の実行を一時停止している状態です。

直列タスクキューの状態を変更するAPIとして*並列タスクキューの状態を変更するAPI*が用意されています。 詳細は非同期処理機能の仕様書を参照してください。

**並列タスクキューの状態を変更するAPI**

| 言語 | API | 備考 |
|---|---|---|

| 言語 | API | 備考 |
| --- | --- | --- |
| Java | TaskManager#setSerializedTaskQueueActive(String queueId, boolean active) | ・queueId: キューID<br>・active:<br>　・true: 有効状態にする<br>　・false: 無効状態にする |
| サーバサイドJavaScript | WorkManager#setSerializedTaskQueueActive(queueId, active) | ・queueId: キューID<br>・active:<br>　・true: 有効状態にする<br>　・false: 無効状態にする |

### 有効状態

有効状態となっている直列タスクキューに実行中のタスクが存在しない場合、その直列タスクキューの先頭で待機中のタスクメッセージは常にタスクとして実行される候補となりえます。

### 無効状態

無効状態となっている直列タスクキューでは、待機中のタスクメッセージはタスクとして実行される候補から外されます。 ただし、ビジネスロジックを実行中のタスクについてはそのまま処理が続行されます。

直列タスクキューを削除するための準備として直列タスクキューを無効状態にするケースがあります。 上記については*直列タスクキュー の削除*で改めて説明します。

> **コラム**
> 直列タスクキューを無効状態にした場合、影響を受ける（処理が停止される）タスクメッセージは該当するキューIDの直列タスクキューのものに限定されます。 他の直列タスクキューに所属するタスクメッセージやタスクには影響ありません。

## 直列タスクキューの登録

直列タスクキューは非同期処理機能の管理者が用意する必要があります。

初期状態の非同期処理機能では、直列タスクキューは用意されていません。

直列タスクキューを登録するAPIとして*直列タスクキューを登録するAPI*が用意されています。 詳細は非同期処理機能の仕様書を参照してください。

**直列タスクキューを登録するAPI**

| 言語 | API | 備考 |
| --- | --- | --- |
| Java | TaskManager#addSerializedTaskQueue(String queueId, boolean active) | ・queueId: キューID<br>・active:<br>　・true: 登録直後の状態を有効状態にする<br>　・false: 登録直後の状態を無効状態にする |

| 言語 | API | 備考 |
| --- | --- | --- |
| サーバサイドJavaScript | WorkManager#addSerializedTaskQueue(String queueId, boolean active) | ▪ queueId: キューID<br>▪ active:<br>　▪ true: 登録直後の状態を有効状態にする<br>　▪ false: 登録直後の状態を無効状態にする |

登録時に該当する直列タスクキューを有効状態または無効状態のいずれかの状態にしておくことが可能です。

- タスクメッセージを処理する環境が用意されていて、直列タスクキューの登録時した時点で処理を開始したい場合は有効状態として登録してください。
- 直列タスクキューの登録だけ先に行い、タスクメッセージの処理そのものはまだ実行させたくない場合は無効状態として登録してください。

## 直列タスクキュー の削除

不要となった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APIとして*直列タスクキューを削除するAPI*が用意されています。 詳細は非同期処理機能の仕様書を参照してください。

**直列タスクキューを削除するAPI**

| 言語 | API | 備考 |
| --- | --- | --- |
| Java | TaskManager#removeSerializedTaskQueue(String queueId) | ▪ queueId: キューID |
| サーバサイドJavaScript | WorkManager#removeSerializedTaskQueue(String queueId) | ▪ queueId: キューID |

直列タスクキューを削除する場合、待機中のタスクメッセージおよびビジネスロジックを実行中のタスクが含まれていないことが前提です。

直列タスクキューの内容を空にする確実な方法はありませんが、以下の様にある程度手順化することは可能です。

1. 該当する直列タスクキューを無効状態にします。
2. 該当する直列タスクキューに新しいタスクメッセージが追加されないようにアプリケーション側で制御します。

> **注意**
> 直列タスクキューが無効状態であってもタスクメッセージは登録できます。

3. 現在処理中のタスクはビジネスロジックが完了するまで待機するか、終了通知を行って管理対象外にします。
4. 待機中のタスクメッセージをすべて削除します。

この手順の実装例を*直列タスクキューの削除（Java）*および*直列タスクキューの削除（サーバサイドJavaScript）*に示します。

### 直列タスクキューの削除（Java）

```
package sample.mytest;

import jp.co.intra_mart.foundation.asynchronous.InvalidTaskException;
import jp.co.intra_mart.foundation.asynchronous.TaskControlException;
import jp.co.intra_mart.foundation.asynchronous.TaskIllegalStateException;
import jp.co.intra_mart.foundation.asynchronous.TaskManager;
import jp.co.intra_mart.foundation.asynchronous.TaskQueueIllegalStateException;
import jp.co.intra_mart.foundation.asynchronous.report.RegisteredInfo;
import jp.co.intra_mart.foundation.asynchronous.report.RegisteredSerializedTaskInfo;
import jp.co.intra_mart.foundation.asynchronous.report.RegisteredSerializedTaskQueueInfo;

public class SerializedTaskQueueSample {
  ...

  private void removeQueue(String queueId)
      throws TaskQueueIllegalStateException, TaskControlException {

    // キューを無効化
    TaskManager.setSerializedTaskQueueActive(queueId, false);

    // タスクメッセージが新規追加されないように制御
    ...

    // すべて直列タスクキュー情報を取得
    RegisteredInfo registeredInfo = TaskManager.getRegisteredInfo();
    RegisteredSerializedTaskQueueInfo queueInfo =
        registeredInfo.getSerializedTaskQueuesInfo().get(queueId);

    // 処理実行中のタスクに対して終了通知
    RegisteredSerializedTaskInfo runningTaskInfo = queueInfo.getRunningTaskInfo();
    if (runningTaskInfo != null) {
      String runningMessageId = runningTaskInfo.getMessageId();
      TaskManager.releaseRunningParallelizedTask(runningMessageId, false, true);
    }

    // 待機中のタスクメッセージをすべて削除
    for (RegisteredSerializedTaskInfo task : queueInfo.getWaitingTasksInfo()) {
      String messageId = task.getMessageId();
      try {
        TaskManager.removeSerializedTask(messageId);
      } catch (TaskIllegalStateException e) {
        ...
      } catch (InvalidTaskException e) {
        ...
      }
    }

    TaskManager.removeSerializedTaskQueue(queueId);
  }
  ...

}
```

コラム

- *直列タスクキューの削除（Java）*ではコードの見やすさを優先しているため、例外処理の詳細については省略しています。
- intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、下記のように、指定のキューのみを取得して直列タスクキューを削除することもできます。

```
// 指定した直列タスクキュー情報を取得
RegisteredSerializedTaskQueueStatus queueInfo =
TaskManager.getSerializedTaskQueuesStatusById(queueId);

if (queueInfo != null) {
  // 処理実行中のタスクに対して終了通知
  RegisteredSerializedTaskInfo runningTaskInfo = queueInfo.getRunningTasksInfo();
  if (runningTaskInfo != null) {
    String runningMessageId = runningTaskInfo.getMessageId();
    TaskManager.releaseRunningParallelizedTask(runningMessageId, false, true);
  }

  // 待機中のタスクメッセージをすべて削除
  SearchRegisteredSerializedTaskInfo waitingQueueInfo = TaskManager.getSerializedTaskInfo(queueId,
null, null, 1, -1);
  for (RegisteredSerializedTaskInfo task : waitingQueueInfo.getTaskInfo()) {
    String messageId = task.getMessageId();
    try {
      TaskManager.removeSerializedTask(messageId);
    } catch (TaskIllegalStateException e) {
      ...
    } catch (InvalidTaskException e) {
      ...
    }
  }
}
```

直列タスクキューの削除（サーバサイドJavaScript）

```
function removeQueue(String queueId) {

  let workManager = new WorkManager();

  // キューを無効化
  workManager.setSerializedTaskQueueActive(queueId, false);

  // タスクメッセージが新規追加されないように制御
  ...

  // すべて直列タスクキュー情報を取得
  let registeredInfo = workManager.getRegisteredInfo().data;
  let queueInfo = registeredInfo.serializedTasksInfo[queueId];

  // 処理実行中のタスクに対して終了通知
  let runningTaskInfo = queueInfo.runningTaskInfo;
  if (runningTaskInfo != null) {
    let runningMessageId = runningTask.messageId;
    workManager.releaseRunningParallelizedTask(runningMessageId, false, true);
  }

  // 待機中のタスクメッセージをすべて削除
  for (let task in queueInfo.waitingTasks) {
    let messageId = task.messageId;
    let result = workManager.removeSerializedTask(messageId);
    if (!result.error) {
      ...
    }
  }

  workManager.removeSerializedTaskQueue(queueId);
}
...
```

コラム

- *直列タスクキューの削除（サーバサイドJavaScript）*ではコードの見やすさを優先しているため、例外処理の詳細については省略しています。
- intra-mart Accel Platform 2017 Summer(Quadra) 以降をご利用の場合、下記のように、指定のキューのみを取得して直列タスクキューを削除することもできます。

```
// 指定した直列タスクキュー情報を取得
let queueStatus = workManager.getSerializedTaskQueuesStatusById(queueId).data;

// 処理実行中のタスクに対して終了通知
if (queueStatus != null) {
  if (!isNull(queueStatus.serializedTaskInfo)) {
    let runningMessageId = queueStatus.serializedTaskInfo.messageId;
    workManager.releaseRunningParallelizedTask(runningMessageId, false, true);
  }

  // 待機中のタスクメッセージをすべて削除
  let waitingQueueInfo = workManager.getSerializedTaskInfo(queueId, null, null, 1, -1).data;
  let waitingTasksInfo = waitingQueueInfo.serializedTaskInfo;
  for (let i = 0; i < waitingTasksInfo.length; i++) {
    let messageId = waitingTaskInfo[i].messageId;
    let result = workManager.removeSerializedTask(messageId);
    if (!result.error) {
      ...
    }
  }

  workManager.removeSerializedTaskQueue(queueId);
}
```