【MicroProfile開発ガイド】
MicroProfile Fault Tolerance

2018/12

日本アイ・ビー・エム システムズ・エンジニアリング株式会社

Disclaimer

- この資料は日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム　システムズ・エンジニアリング株式会社の正式なレビューを受けておりません。
- 当資料は、資料内で説明されている製品の仕様を保証するものではありません。
- 資料の内容には正確を期するよう注意しておりますが、この資料の内容は2018年6月現在の情報であり、製品の新しいリリース、PTFなどによって動作、仕様が変わる可能性があるのでご注意下さい。
- 今後国内で提供されるリリース情報は、対応する発表レターなどでご確認ください。
- IBM、IBMロゴおよびibm.comは、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。他の製品名およびサービス名等は、それぞれIBMまたは各社の商標である場合があります。現時点でのIBMの商標リストについては、www.ibm.com/legal/copytrade.shtmlをご覧ください。
- 当資料をコピー等で複製することは、日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム　システムズ・エンジニアリング株式会社の承諾なしではできません。
- 当資料に記載された製品名または会社名はそれぞれの各社の商標または登録商標です。
- JavaおよびすべてのJava関連の商標およびロゴはOracleやその関連会社の米国およびその他の国における商標または登録商標です。
- Microsoft, WindowsおよびWindowsロゴは、Microsoft Corporationの米国およびその他の国における商標です。
- Linuxは、Linus Torvaldsの米国およびその他の国における登録商標です。
- UNIXはThe Open Groupの米国およびその他の国における登録商標です。

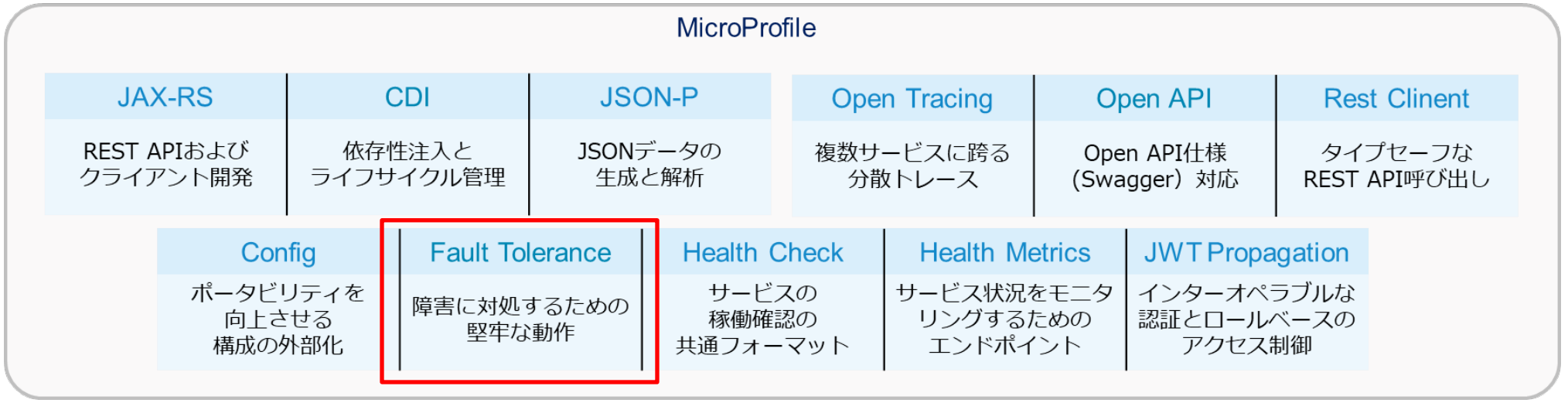
目次

MicroProfile Fault Tolerance概要

MicroProfile Fault Toleranceとは

MicroProfile Fault Toleranceはマイクロサービス環境において1 つのサービスで発生した障害が他のサービスに波及することを食い止め、障害許容性と回復性を実現するための機能を提供します。

MicroProfile Fault Tolerance 1.1はMicroProfile 1.4およびMicroProfile 2.0の一部として2018年6月にリリースされました。
WAS LibertyではMicroProfile Fault Tolerance 1.1 仕様に準拠したmpFaultTolerance-1.1フィーチャー を提供しており、内部実装としてオープン・ソースのFailsafeを使用しています。
当該フィーチャーの各機能はアプリケーションのメソッドやクラスにアノテーションを付与することで利用可能です。

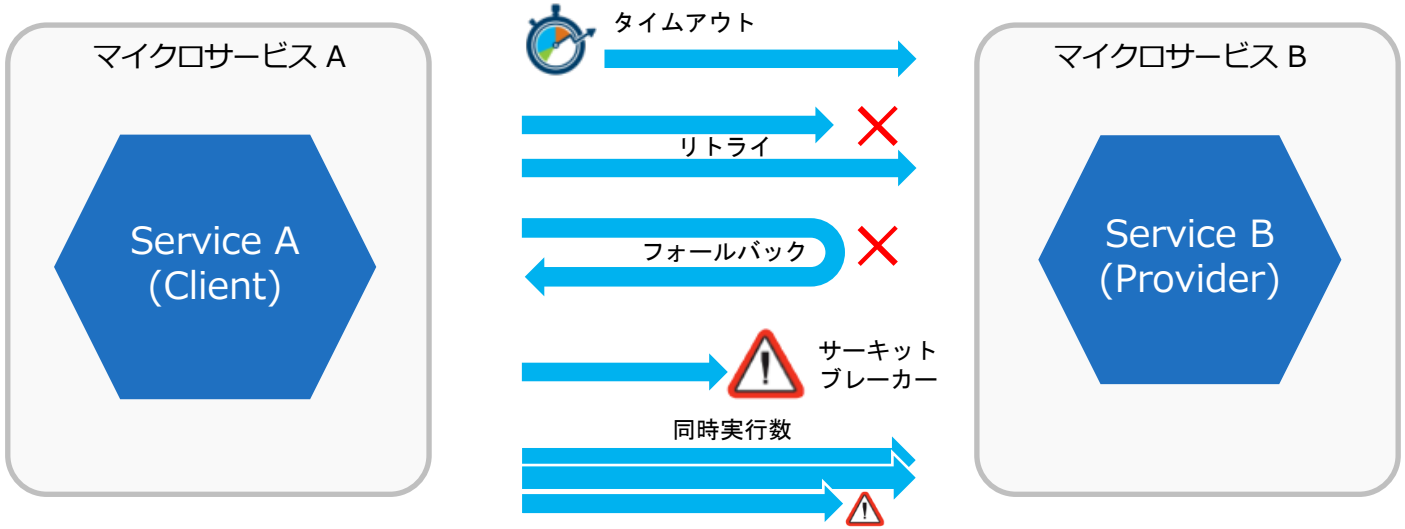


利用シナリオ

マイクロサービスでは「Design for Failure（障害を前提とした設計）」という考え方に基づき、サービスおよびアプリケーション全体の耐障害性（レジリエンシー）を高めることが重要とされています。

MicroProfile Fault ToleranceはCircuitBreakerパターンやBulkheadパターンといったマイクロサービスの耐障害性を向上するためのデザインパターンや、タイムアウトやリトライ、フォールバック等のポリシーを組み込むことにより、サービス呼出の障害に対するアプリケーション全体の耐障害性を高めるために利用します。

サービスの実装クラスにアノテーションを付与することにより、MicroProfile Fault Toleranceの機能がCDIインターセプターとして対象のクラスに組み込まれます。アノテーションにより容易に利用することができるため、サービスに各機能を組み込むための開発負荷を大幅に軽減することができます。



MicroProfile Fault Tolerance 機能一覧

機能	概要
Timeout	サービス呼び出し (メソッド実行) に長時間かかっている場合に、その応答を待つ処理を中断させる機能を提供します。 指定された時間を経過すると、TimeoutExceptionがスローされます。
Retry	サービス呼び出し (メソッド実行) が失敗した場合に、指定した回数や上限時間内で再実行する機能を提供します。
Fallback	サービス呼び出し (メソッド実行) が失敗した場合に、代替サービスの呼び出し (代替メソッドの実行) を行う機能を提供します。
CircuitBreaker	サービス呼び出し (メソッド実行) が指定した割合を超えて失敗すると、サーキット・ブレーカーが開放状態となり、指定された期間はサービス呼び出し (メソッド実行) で即座に例外がスローされるようになります。(詳細な動作は後述) CircuitBreakerを使用すると、障害やスローダウンが発生している後段のサービスを一時的に切り離すことで、前段のサービスへの影響を最小限にすることができます。
Bulkhead	サービス呼び出し (メソッド実行) の同時実行数を制限する機能を提供します。 同時実行数に制約があるサービス (メソッド) が想定以上の多重度で実行されることで、障害やスローダウン等が発生する状況を抑止します。 指定された同時実行数や実行待ち数に達すると、BulkheadExceptionがスローされます。
Asynchronous	この機能は、サービス呼び出し (メソッド実行) を別スレッドで実行する機能を提供し、Bulkheadをスレッド・プール形式で利用する場合に使用されます。 Bulkheadと組み合わせずに利用することも可能で、処理を別スレッドで実行したい場合に重宝する機能です。

フィーチャーの有効化

WAS Liberty上でMicroProfile Fault Toleranceの機能を有効化するためには、該当サーバーのserver.xmlにmpFaultTolerance-1.1フィーチャーを設定します。
当該フィーチャーが含まれるMicroProfile-1.4やMicroProfile-2.0を設定することも可能です。

```
<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>mpFaultTolerance-1.1</feature>
        ・・・・
    </featureManager>
    ・・・・
</server>
```

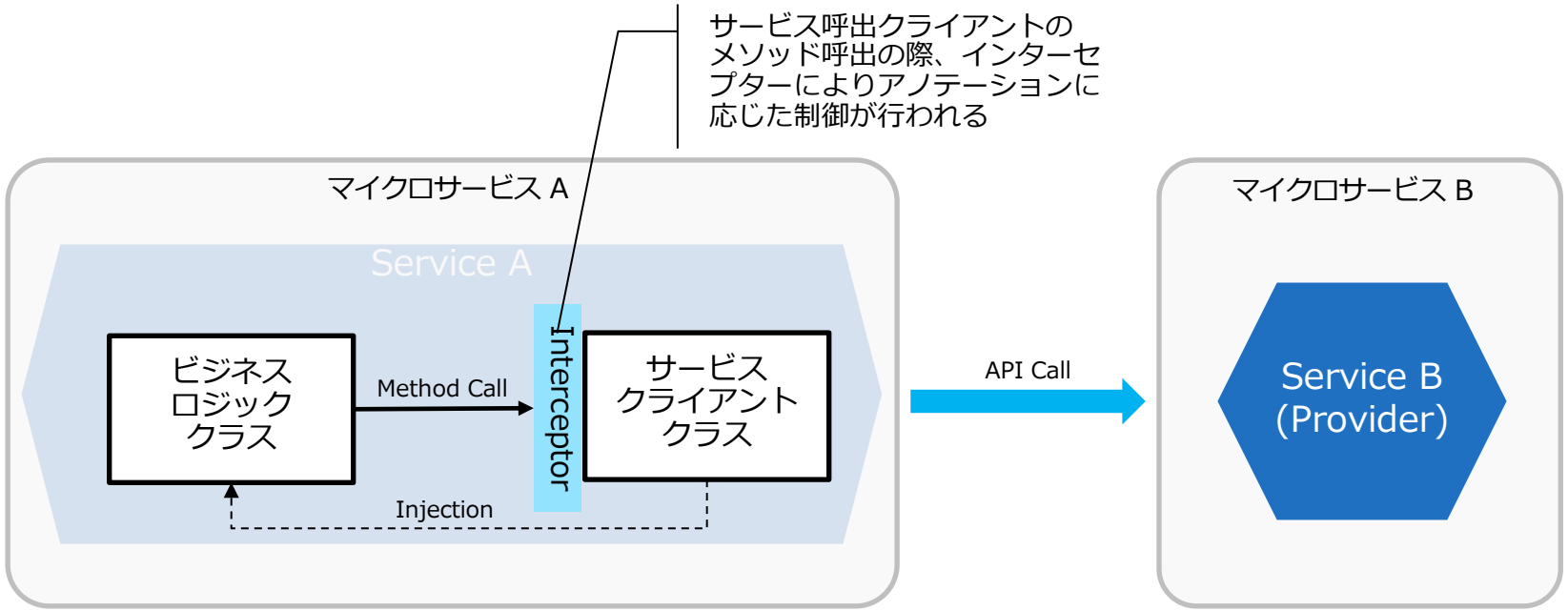
mpFaultTolerance-1.1フィーチャーを有効化することにより、以下のフィーチャーも暗黙的ロードにより自動的に有効化されます。

- mpConfig-1.3
- concurrent-1.0

※mpConfigが有効化される理由については「MicroProfile Fault Toleranceの無効化」(p.43)を参照ください。

インターセプターの組み込み

MicroProfile Fault Toleranceの機能は基本的に呼出元のマイクロサービスに組み込みます。
各機能はCDIインターセプターとして組み込まれるため、サービス呼出処理を行うクラスにMicroProfile Fault Toleranceのアノテーションを付与し、そのクラスを呼出元サービスにCDI管理Beanとしてインジェクションする形になります。



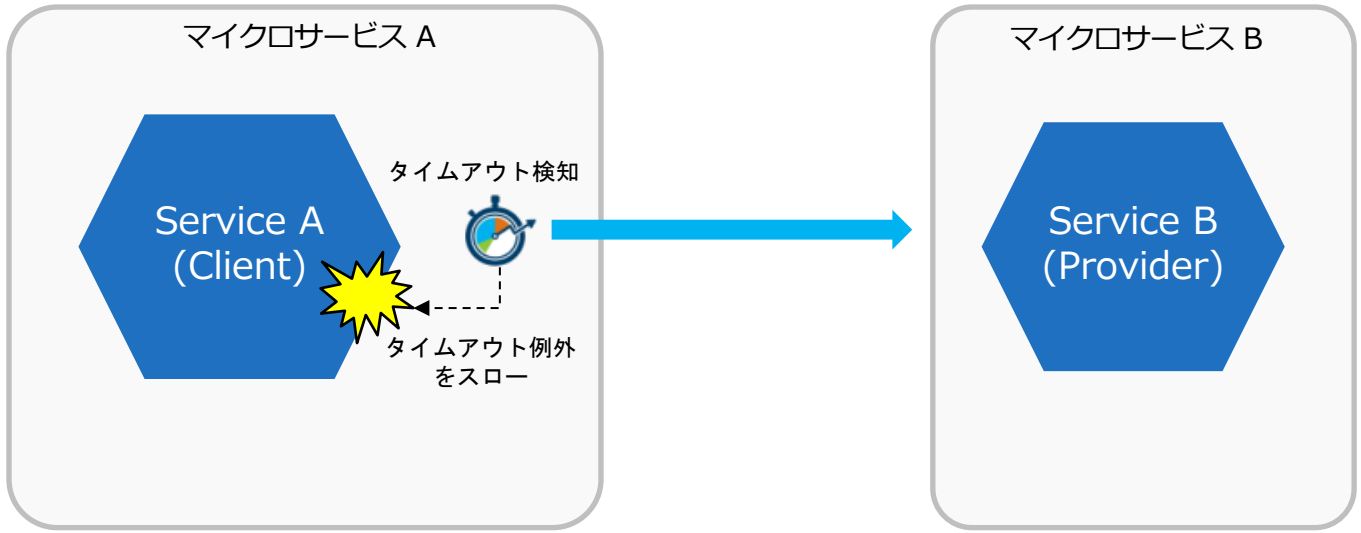
※アノテーションはクラス／メソッドのいずれのレベルでも指定可能です。（クラスレベルの場合全てのメソッドに適用）
※上記の構造が適用可能な箇所であれば、実際には他サービス呼出とは異なる内部処理等にもMicroProfile Fault Toleranceの機能を使用することが可能です。

各機能の使用方法

Timeout

Timeout：サービス呼出のタイムアウト

Timeout機能は、サービス呼び出し (メソッド実行) に長時間かかっている場合に、その応答を待つ処理を中断させる機能を提供します。
指定された時間を経過すると、TimeoutExceptionがスローされます。



Timeoutの使用方法

- アノテーション

アノテーション	クラス
@Timeout	org.eclipse.microprofile.faulttolerance.Timeout

  - パラメータ

パラメータ	説明
value	処理がタイムアウトとなる時間を設定。デフォルトは1000。
unit	valueパラメータに指定する時間の単位（ChronoUnit型）。 デフォルトはChronoUnit.MILLIS（ミリ秒）。

- コード例

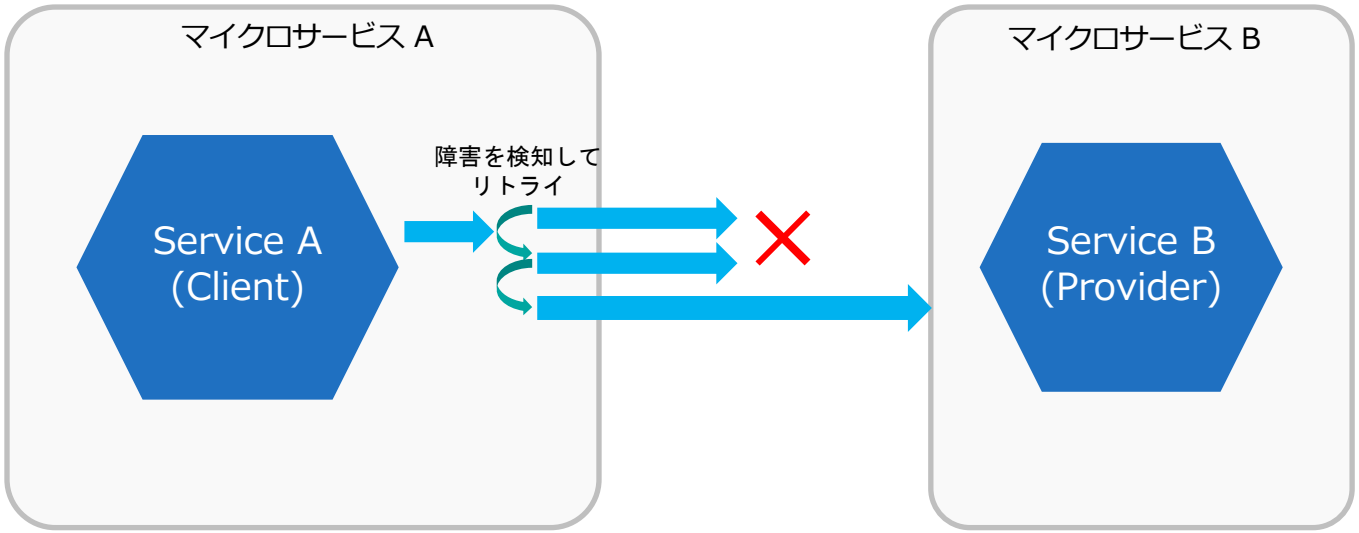
```
@Timeout(400) // タイムアウトを400msに設定
public String serviceA() {

   // サービス呼出
}
```

上記のメソッドを呼び出した際、処理が400ミリ秒で終了しない場合はTimeoutExceptionがスローされます。

Retry

Retry：サービス呼出の自動リトライ

サービス呼び出し（メソッド実行）が失敗した場合に、指定した回数や上限時間内で再実行する機能を提供します。サービス呼び出しにおける一時的な障害に対する影響を低減するために利用できます。



Retryの使用方法

- アノテーション

アノテーション	クラス
@Retry	org.eclipse.microprofile.faulttolerance.Retry

  - パラメータ

パラメータ	説明
maxRetries	リトライ回数の上限。デフォルトは3。
delay	リトライ間隔の遅延時間。デフォルトは0。
delayUnit	リトライ間隔の遅延時間の単位（ChronoUnit型）。デフォルトはChronoUnit.MILLIS（ミリ秒）。
maxDuration	リトライを続行する時間の上限。デフォルトは180000。
durationUnit	リトライを続行する時間の単位（ChronoUnit型）。デフォルトはChronoUnit.MILLIS（ミリ秒）。
jitter	リトライ間隔の遅延時間の揺らぎ。デフォルトは200（-200～+200msecの範囲の揺らぎが発生）。
jitterDelayUnit	リトライ間隔の遅延時間の揺らぎの単位（ChronoUnit型）。 デフォルトはChronoUnit.MILLIS（ミリ秒）。
retryOn	検知した場合にリトライ対象と見做す例外クラス。デフォルトはjava.lang.Exception。
abortOn	検知した場合にリトライを中断する例外クラス。デフォルトはなし。

Retryの使用方法

- コード例

```
@Retry(maxRetries=10, maxDuration=1000) // リトライ回数の上限を10、続行時間の上限を1000msecに設定
public String serviceA() {

   // サービス呼出
}
```

上記のメソッド呼び出しにおいて、1秒間最大10回まで処理をリトライします。10回に満たない場合でも1秒経過すればそれ以上のリトライは行いません。

```
@Retry(delay=400, maxDuration=3200, // リトライ間隔の遅延を400msec、続行時間の上限を3200msecに、
       jitter=400, maxRetries=10,   // 遅延の揺らぎを400msec 、リトライ回数の上限を10に
       retryOn={IOException.class})  // リトライ対象をIOExceptionに設定
public String serviceA() {

   // サービス呼出
}
```

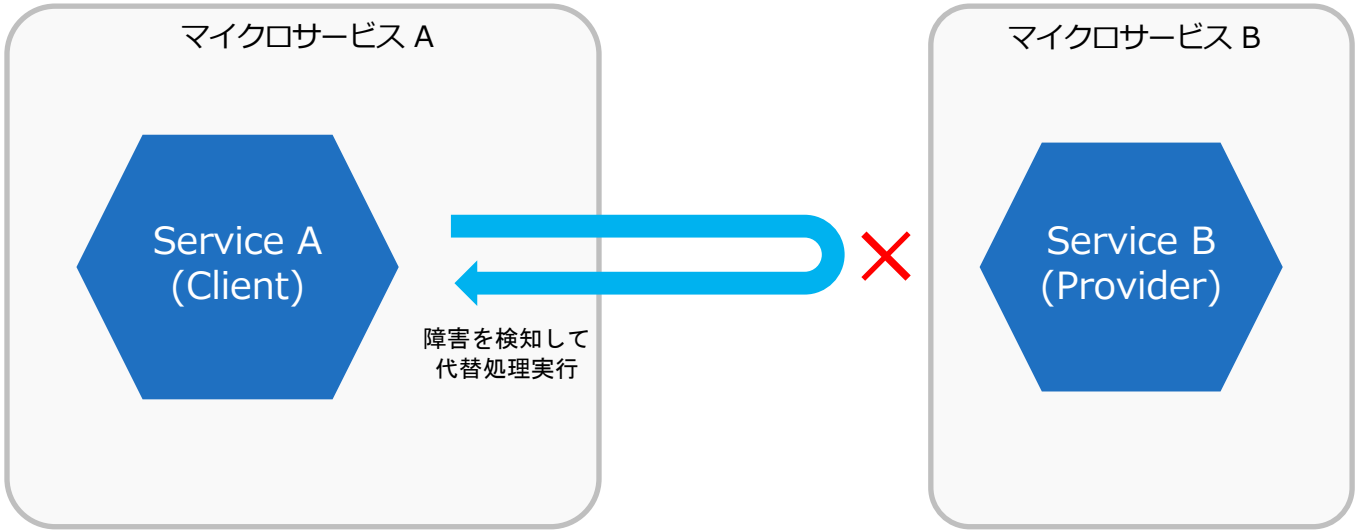
上記のメソッド呼び出しにおいて、IOExceptionが発生した場合、最大3.2秒間10回まで処理をリトライします。リトライ間隔には0～800msecの遅延（-400～400msecの揺らぎ）が挿入されます。

Fallback

Fallback : 代替処理の実行

サービス呼び出し（メソッド実行）が失敗した場合に、代替サービスの呼び出し（代替メソッドの実行）を行う機能を提供します。
代替処理の指定の方法には以下の2つの方法があります。

- フォールバック・メソッド（代替メソッド）の指定
- フォールバック・ハンドラーの指定



Fallbackの使用方法

- アノテーション

アノテーション	クラス
@Fallback	org.eclipse.microprofile.faulttolerance.Fallback

  - パラメータ

パラメータ	説明
value	障害を検知した場合に呼び出すフォールバック・ハンドラーの実装クラス。 フォールバック・ハンドラーは下記のインターフェースを実装し、アノテーションを指定した元のメソッドと同じ型の戻り値を返す代替メソッドを備える必要があります。
fallbackMethod	障害を検知した場合に呼び出す代替メソッド名。valueパラメータと同時に指定できません。 代替メソッドは同一のクラスに存在し、引数の型・数や戻り値の型が同じである必要があります。

- フォールバック・ハンドラー用インターフェース

クラス
org.eclipse.microprofile.faulttolerance.FallbackHandler<T>

  - メソッド

メソッド	説明
T handle(ExecutionContext context)	障害を検知した場合に呼び出す代替メソッド。 代替メソッドは元のメソッドと同じ型の戻り値を返す必要があります。 引数のExecutionContextからは元のメソッド名や引数を取得することができます。

Fallbackの使用方法

- コード例
  - フォールバック・メソッドの使用例

```
@Fallback(fallbackMethod="fallbackForServiceA") // fallbackForServiceA()を代替メソッドに設定
public String serviceA() {

   // サービス呼出
}

public String fallbackForServiceA() {

   // 代替処理
}
```

上記のメソッド（serviceA()）呼び出しにおいて、例外が検知された場合代替メソッド（fallbackForServiceA()）が実行され、代替メソッドの結果が返されます。

Fallbackの使用方法

– フォールバック・ハンドラーの使用例

```
    @Fallback(SampleFallbackHandler.class) // SampleFallbackHandlerクラスをハンドラーに設定
    public String serviceA() {

        // サービス呼出
    }
```

（フォールバック・ハンドラー実装）

```
public class SampleFallbackHandler implements FallbackHandler<String> {

    @Override
    public String handle(ExecutionContext context) {

        // 代替処理
    }
}
```

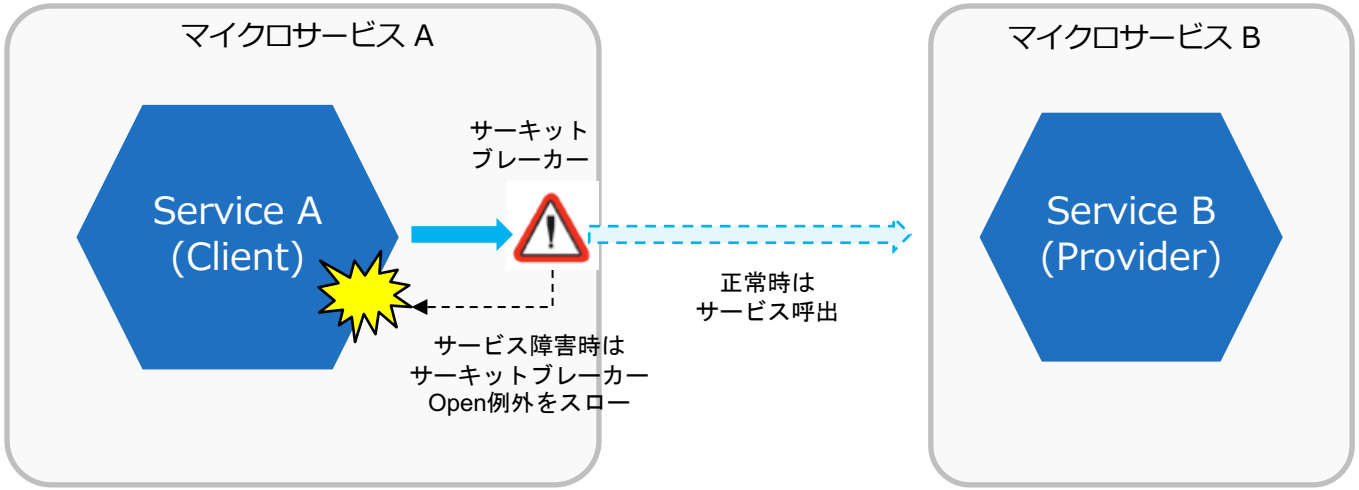
上記のメソッド（serviceA()）呼び出しにおいて、例外が検知された場合フォールバック・ハンドラー（SampleFallbackHandler#handle()）が実行され、代替処理の結果が返されます。

代替処理の中では、引数のExecutionContextオブジェクトから元のメソッド名や引数を取得して使用することが可能です。（https://openliberty.io/javadocs/microprofile-1.3-javadoc/org/eclipse/microprofile/faulttolerance/ExecutionContext.html）

CircuitBreaker

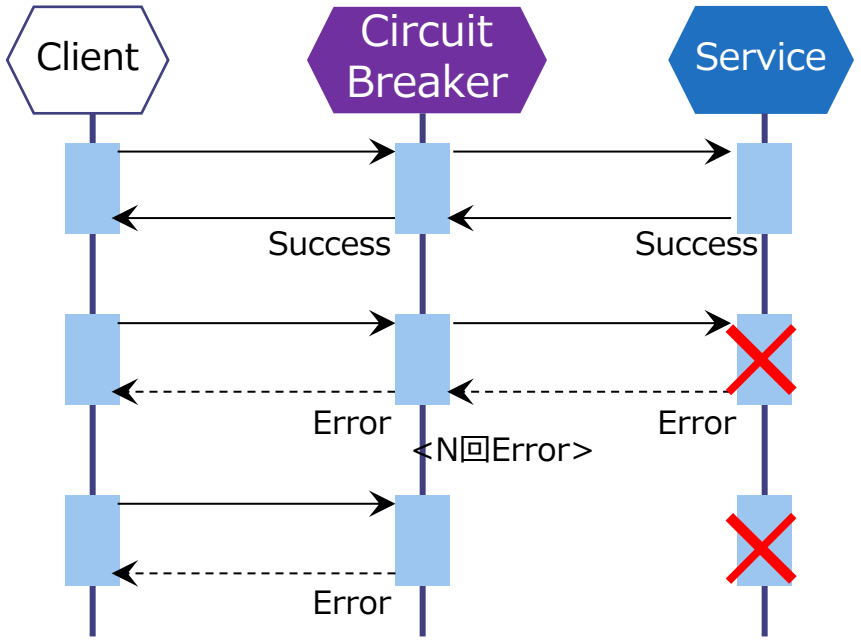
CircuitBreaker：障害サービス呼出の遮断

サービス呼び出し（メソッド実行）が指定した割合を超えて失敗すると、サーキット・ブレーカーが開放状態となり、指定された期間はサービス呼び出し (メソッド実行) で即座に例外（CircuitBreakerOpenException）がスローされるようになります。
CircuitBreaker を使用すると、障害やスローダウンが発生している後段のサービスを一時的に切り離し、前段のサービスへの影響を最小限にすることができます。



CircuitBreakerパターンとは

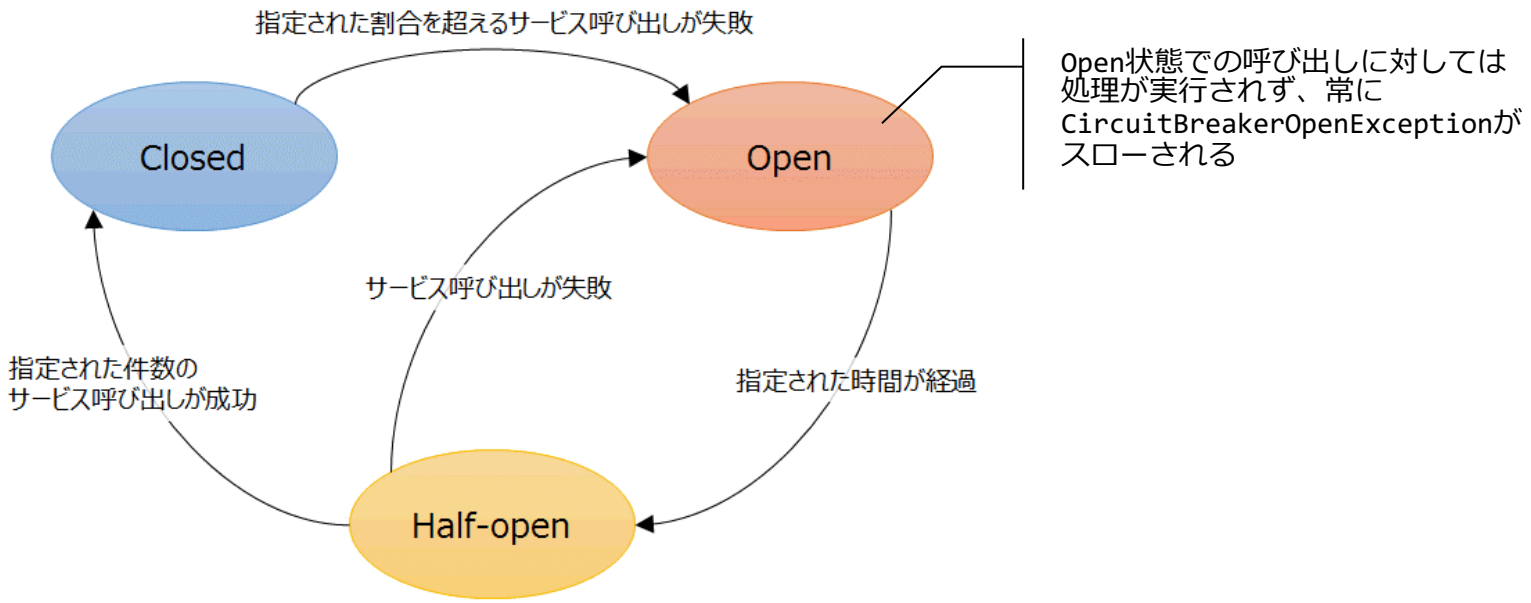
CircuitBreakerパターンはマイクロサービスの耐障害性を向上するためのデザインパターンの一つです。マイクロサービスの呼出パスの中に遮断器（サーキット・ブレーカー）を配置し、障害の発生したサービスに対する呼出を遮断することにより１サービスの障害によるアプリケーション全体の性能低下を防止します。



サーキット・ブレーカーの動作

サーキット・ブレーカーは、Closed／Open／Half-openの3つの状態の遷移により動作を制御します。

状態	説明
Closed（初期状態）	サーキット・ブレーカーが閉じ、サービス呼び出し（メソッド実行）が通常に行える状態です。 Closed状態の時に指定された割合のサービス呼び出しが失敗するとOpen状態に遷移します。
Open	サーキット・ブレーカーが開放され、一切のサービス呼び出しが行えない状態です。 Open状態で一定時間経過すると、Half-openの状態に遷移します。
Half-open	制限されたサービス呼び出しだけが行える状態です。 この状態で指定された件数のサービス呼び出しが成功するとClosed状態に遷移します。



CircuitBreakerの使用方法

- アノテーション

アノテーション	クラス
@CircuitBreaker	org.eclipse.microprofile.faulttolerance.CircuitBreaker

  - パラメータ

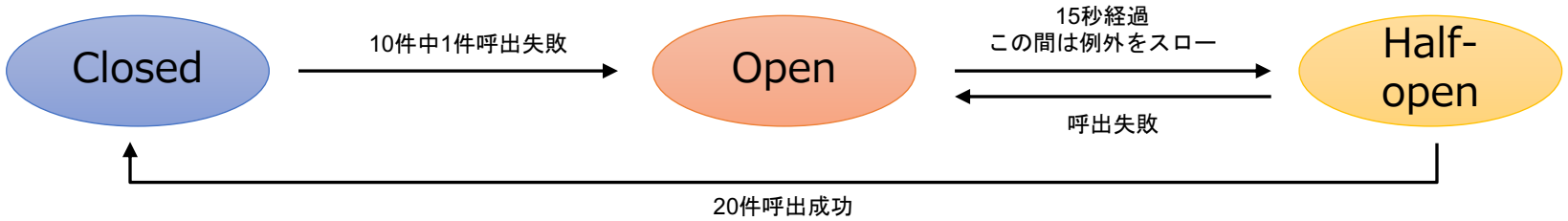
パラメータ	説明
delay	Open状態からHalf-open状態になるまでの経過時間。デフォルトは5000。
delayUnit	Open状態からHalf-open状態になるまでの経過時間の単位（ChronoUnit型）。 デフォルトはChronoUnit.MILLIS（ミリ秒）。
failOn	検知した場合に呼出失敗と見做す例外クラス。デフォルトはjava.lang.Throwable。
failureRatio	Open状態に遷移する直近の呼出失敗の割合の閾値。デフォルトは0.5。
requestVolume Thread	呼出失敗をチェックする直近の呼出件数。デフォルトは20。 指定された件数の直近のサービス呼出がfailureRatioを超える割合で失敗するとOpen状態に 遷移します（呼出件数が指定件数に達するまでは失敗してもそのまま元の例外をスローします）。
successThreshold	Half-open状態からClosed状態へ遷移するまでの呼出成功件数。デフォルトは１。

CircuitBreakerの使用方法

- コード例

```
// 直近10件のメソッド呼出の1/10が失敗するとOpen状態へ遷移
// 15秒経過後Half-open状態へ遷移し、その後20件呼出成功するとClosed状態に戻る
@CircuitBreaker(
      requestVolumeThreshold=10, failureRatio=0.1,
      successThreshold=20,
      delay=15,delayUnit=ChronoUnit.SECONDS)
public String serviceA() {

   // サービス呼出
}
```

上記のメソッド呼び出しにおいて、直近10件中1件以上失敗すればCircuitBreakerOpenExceptionをスローし、その後15秒間は全てCircuitBreakerOpenExceptionをスローし続けます。15秒経過後の呼び出しで20件成功すると、その後は通常に呼び出しを実行できるようになります。



CircuitBreakerの使用方法

- オブジェクト存続期間についての注意点

  CircuitBreakerは前述のように状態を持つものなので、CircuitBreakerが長期間存続し、該当サービスに対する全ての呼び出しで CircuitBreakerが共用されなければなりません。このため、CircuitBreakerのアノテーションを付与したオブジェクトも長期間存続し、共用されなければなりません。

  スコープとして@Dependentが指定され、リクエスト毎に作成・破棄されるオブジェクトになっている場合、CircuitBreakerのアノテーションが付与されていても、CircuitBreakerの状態が保持され共有されるのは1つの要求の範囲になってしまいます。

  この問題を解決するためには、スコープを@ApplicationScoped等に変更し、複数リクエストで共有可能なスコープでCircuitBrekerの状態を保持する必要があります。

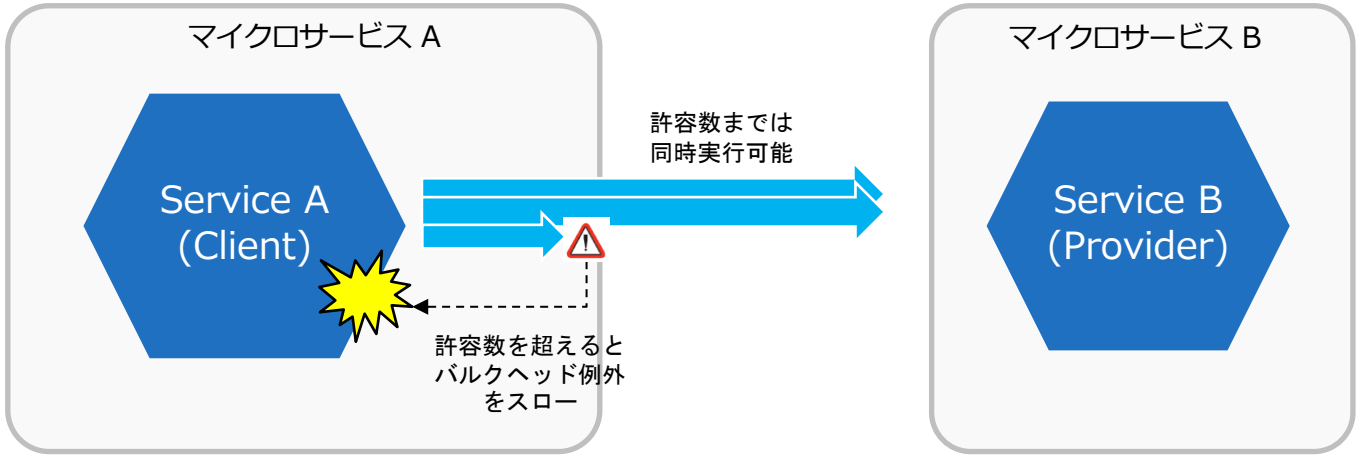
Bulkhead

Bulkhead：同時実行数の制御

Bulkhead機能はサービス呼び出し（メソッド実行）の同時実行数を制限する機能を提供します。
同時実行数に制約があるサービス（メソッド）が想定以上の多重度で実行されることで、障害やスローダウン等が発生する状況を抑止します。

MicroProfile Fault ToleranceのBulkhead機能は以下の２つのアプローチが使用可能です。
指定された同時実行数や実行待ち数に達すると、BulkheadExceptionがスローされます。

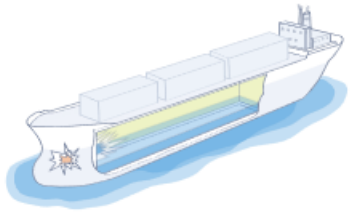
アプローチ	説明
セマフォ分離	設定した同時実行数の上限を超える実行要求に対して即時エラーとする。
スレッドプール分離	該当の処理に専用のスレッドプールと処理待ちキューを割り当て非同期に実行する。設定した同時実行数の上限を超える実行要求はキューに入るが、キューの最大サイズを超える要求はエラーとする。BulkheadとAsynchronousの2機能を組み合わせることにより可能。



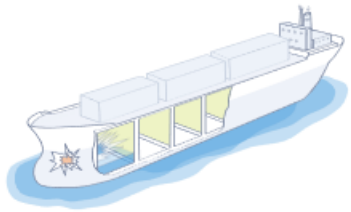
Bulkheadパターン

Bulkheadの元の意味は、船倉を前後方向に区切って荷室の仕切りとすると同時に、船体の強度、剛性を高める隔壁のことです。Bulkheadパターンはサービスのリソースを分割することにより、障害の影響をその発生箇所から隔離してサービスの機能をできる限り維持するためのパターンです。

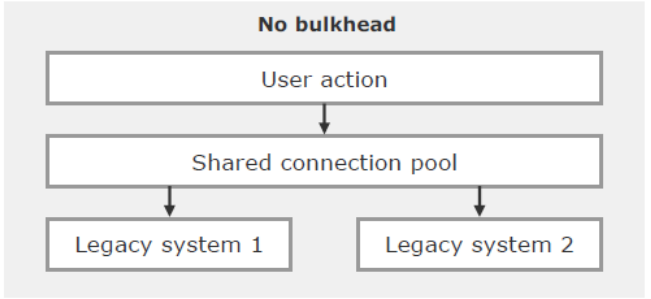
当該パターンにより、複数のサービスの呼び出しに使用されるリソース同士の影響を極小化できます。
例えば、複数のサービスを呼び出すクライアントに各々のサービス用の接続プールを割り当てることができます。 サービスが失敗し始めた場合でもそのサービスに割り当てられている接続プールのみに影響するため、クライアントは他のサービスを引き続き使用できます。

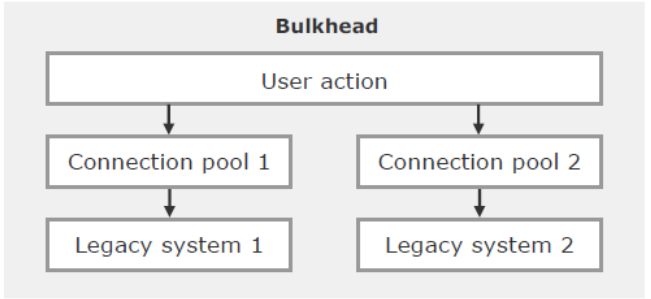


A ship without bulkheads allows water to fill the entire hull.



A ship with bulkheads isolates the water damage.





Bulkheadの使用方法

- アノテーション

アノテーション	クラス
@Bulkhead	org.eclipse.microprofile.faulttolerance.Bulkhead

  - パラメータ

パラメータ	説明
value	同時実行数の上限。デフォルトは10。
waitingTaskQueue	処理待ちキューの上限。デフォルトは10。 @Asynchronousが合わせて指定されている場合のみ有効。@Asynchronousが指定されていない場合は無視されます。

Bulkheadの使用方法

- コード例
  - セマフォ分離形式

```
@Bulkhead(5) // 同時実行数の上限を5に設定
public String serviceA() {

   // サービス呼出
}
```

上記のメソッド呼び出しにおいて、5件を超える同時実行の要求が来るとBulkheadExceptionをスローします。

  - スレッドプール分離形式

```
@Bulkhead(value=5, waitingTaskQueue=5) // 同時実行数の上限を5、処理待ちキューのサイズを5に設定
@Asynchronous // 当該メソッドの処理を非同期実行し、戻り値をFuture型で返す
public Future<String> serviceA() {

   // サービス呼出の上、結果をFutureに格納して返却
   return CompletableFuture.completedFuture(product);
 }
```

上記のメソッド呼び出しにおいて、処理が非同期実行され、5件を超える同時実行の要求はキューイングされます。
キューイング可能な上限は5件で、それを超えるとBulkheadExceptionをスローします。

Bulkheadの使用方法

- オブジェクト存続期間についての注意点

  Bulkheadが管理するセマフォやスレッドプール は状態を持つものなので、該当サービスに対する全ての呼び出しで共用されなければなりません。このため、Bulkhead のアノテーションを付与したオブジェクトも長期間存続し、共用されなければなりません。

  スコープとして@Dependentが指定され、リクエスト毎に作成・破棄されるオブジェクトになっている場合、Bulkheadのアノテーションが付与されていても要求ごとにセマフォやスレッドプールが割り当てられるため有効に機能しません。

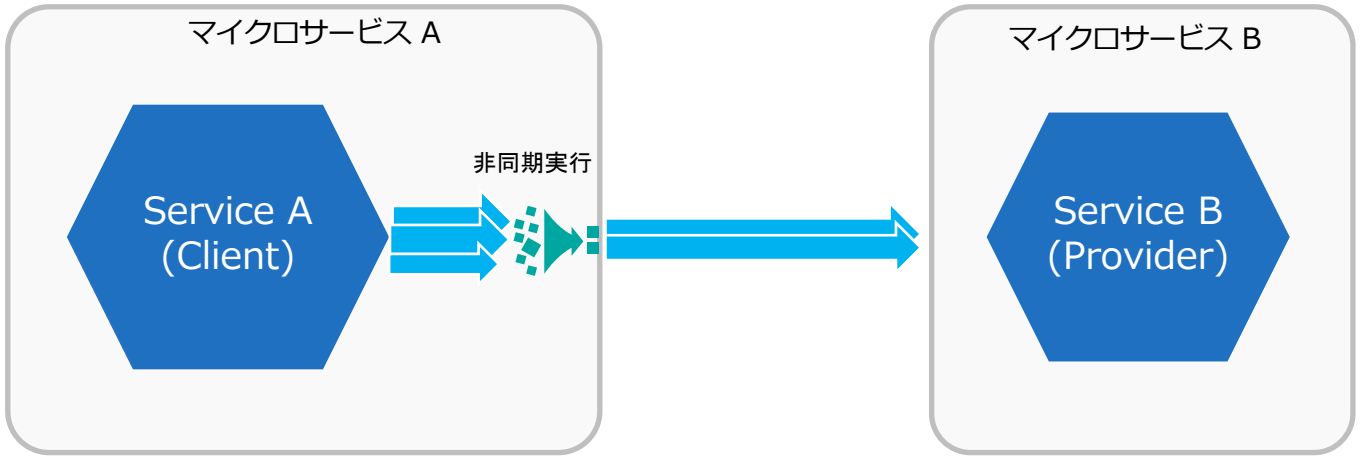
  この問題を解決するためには、スコープを@ApplicationScoped等に変更し、複数リクエストで共有可能なスコープでセマフォやスレッドプールの状態を保持する必要があります。

Asynchronous

Asynchronous：サービス呼出の非同期実行

Asynchronous機能はサービス呼び出し (メソッド実行) を別スレッドで実行する機能を提供し、前述のBulkhead機能をスレッドプール・アプローチで利用する場合に主に使用されます。
Bulkheadと組み合わせずに利用することも可能で、時間のかかるような処理を別スレッドで非同期に実行したい場合に有用な機能です。

Asynchronous機能を使用して非同期処理を行うメソッドは結果をjava.util.concurrent.Future型で返す必要があります。異なる型を戻り値にした場合はアプリケーション自体の起動に失敗します。



Asynchronousの使用方法

- アノテーション

アノテーション	クラス
@Asynchronous	org.eclipse.microprofile.faulttolerance.Asynchronous

  - パラメータ

    なし

Asynchronousの使用方法

- コード例

```
@Asynchronous // 非同期処理に設定
public Future<String> serviceA() {

   // 時間のかかる処理。結果は以下のように返す
   return CompletableFuture.completedFuture(result);
}
```

上記のメソッドを呼び出した際、処理は呼出元とは別スレッドで非同期に実行されます。
呼出元では返されるFutureに対してget()メソッドを実行して非同期に実行された処理の結果を取得可能です。
このget()メソッドは非同期で実行されている処理が終わるまで呼出元のスレッドをブロックして待ちます。

- 呼出元のコード例

```
Future<String> future = svcClient.serviceA(); // 非同期処理の呼び出し

// この間の処理は呼出先と並列に実行される

String result = future.get(); // 非同期処理の終了を待って結果を取得
```

複数機能の同時利用

複数機能の同時利用

MicroProfile Fault Toleranceが提供するアノテーションを同時に指定することにより、対象クラス／メソッドに対して各機能を組み合わせて利用することが可能です。

@Retryや@Fallback、@CircuitBreakerのように例外の発生をトリガーとする機能を同時指定した場合、MicroProfile Fault Toleranceの機能により発生する例外（TimeoutExceptionやBulkheadExceptionなど）もデフォルトでトリガーの対象となります。

@Timeoutと@Retryを同時に使用した例

```
@Timeout(400)          // タイムアウトを400msに設定
@Retry(maxRetries=2) // リトライ回数を2回に設定
public String serviceA() {

   // サービス呼出
}
```

上記のメソッド呼び出しにおいて、処理が400ミリ秒で終了しない場合も他の例外発生と共に2回までリトライの対象となります。2回目のリトライ時にタイムアウトとなった場合はTimeoutExceptionがスローされます。

設定の外部化

MicroProfile Fault Toleranceの無効化

MicroProfile Fault Toleranceを使用したマイクロサービスをサービス・メッシュ（例：Istio）上にデプロイする場合、サービス・メッシュ側の機能を使用してアプリケーション全体の耐障害性を確保したいケースが出てくることが想定されます。このようなケースを想定して、MicroProfile Fault ToleranceではFallback以外の機能をまとめて無効化する手段を提供しています。（Fallbackはサービス・メッシュでは実現できないため対象外となっています）

- 以下のプロパティを環境変数として設定することにより、コンテナー化されたサービスに対しても外部からFallback以外の機能の無効化が可能です。

プロパティ	説明
MP_Fault_Tolerance_NonFallback_Enabled	falseを設定するとFallback以外の機能は無効化される。 true若しくはプロパティが存在しない場合は有効のまま。

  - 上記プロパティの読み込みにはMicroProfile Configの機能を使用しています。
    従ってJVMシステムプロパティに指定することも可能ですが、デフォルトでは環境変数よりJVMシステムプロパティが優先されますので、コンテナー内でJVMシステムプロパティに明示的に指定すると外部から環境変数でオーバーライドすることが難しくなります。

※MicroProfile Configについては「MicroProfile開発ガイド　MicroProfile Config」を参照ください。

パラメータのオーバーライド

前頁の無効化と同様、各アノテーションのパラメータについてもMicroProfile Configの機能を使用して外部からオーバーライドすることが可能です。オーバーライドは以下の形式で環境変数やJVMシステムプロパティにオーバーライドする値を指定することにより適用され、指定の仕方（①-③）により適用範囲を制御できます。

①メソッドレベルで指定したアノテーションに対してピンポイントでパラメータをオーバーライド

```
<クラス名>/<メソッド名>/<アノテーション名>/<パラメータ名>=<オーバーライドする値>
```

②クラスレベルで指定したアノテーションに対してパラメータをオーバーライド

```
<クラス名>/<アノテーション名>/<パラメータ名>=<オーバーライドする値>
```

③（クラス／メソッド問わず）指定したアノテーション全てに対してパラメータをオーバーライド

```
<アノテーション名>/<パラメータ名>=<オーバーライドする値>
```

※同時に指定された複数のプロパティが同一箇所に該当した場合、上記の①－③の（適用範囲が狭い）順に優先的に適用されます。

※クラスレベルでアノテーションが指定されている場合に①の形式で指定しても無視されます。逆の場合も同様です。

パラメータのオーバーライド

- 設定例
  - ①の例

```
com.acme.test.MyClient/serviceA/Retry/maxDuration=3000
```

上記のように指定して実行した場合、MyClientクラスのserviceA()メソッドに指定した@RetryアノテーションのmaxDurationパラメータの値だけが3000にオーバーライドされます。

  - ②の例

```
com.acme.test.MyClient/Retry/maxRetries=100
```

上記のように指定して実行した場合、MyClientクラスに指定した@RetryアノテーションのmaxRetriesパラメータの値が100にオーバーライドされます。

  - ③の例

```
Retry/maxRetries=30
```

上記のように指定して実行した場合、全ての@RetryアノテーションのmaxRetriesパラメータの値が30にオーバーライドされます。

MicroProfile Metricsの併用

概要

MicroProfile Fault Tolerance と MicroProfile Metricsを併用すると、Fault Toleranceのアノテーションを付与したメソッドのメトリックが自動的に提供されるようになります。提供されるメトリックはアノテーションの種類により異なりますので、以降のページで説明します。メトリックスの取得方法については当ガイドの「MicroProfile Metrics」を参照してください。

この機能を有効化するためには、該当サーバーのserver.xmlにmpFaultTolerance-1.1フィーチャーとmpMetrics-1.1を設定します。当該フィーチャーが含まれるMicroProfile-1.4やMicroProfile-2.0を設定することも可能です。

```
<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>mpFaultTolerance-1.1</feature>
        <feature>mpMetrics-1.1</feature>
        ・・・・
    </featureManager>
    ・・・・
</server>
```

Retry, Timeout, CircuitBreaker, Bulkhead, Fallbackで共通のメトリック

- applicationスコープで規定されたメトリック（WAS Libertyで取得可能な項目）
  これらのメトリックはMicroProfile Metrics REST APIからデフォルトで取得可能です。

メトリック	説明	タイプ
ft_<メソッド名>_invocations_total	このメソッドが呼ばれた回数を表示します。	カウンター
ft_<メソッド名>_invocations_failed_total	このメソッドが呼ばれ、Fault Toleranceによる処理が行われ、Throwable例外がスローされた回数を表示します。	カウンター

Retryのメトリック

- applicationスコープで規定されたメトリック（WAS Libertyで取得可能な項目）
  これらのメトリックはMicroProfile Metrics REST APIからデフォルトで取得可能です。

メトリック	説明	タイプ
ft_<メソッド名>_retry_calls_succeeded_not_retried_total	このメソッドが呼ばれてリトライなしに成功した回数を表示します。	カウンター
ft_<メソッド名>_retry_calls_succeeded_retried_total	このメソッドが呼ばれ、1回以上のリトライの後に成功した回数を表示します。	カウンター
ft_<メソッド名>_retry_calls_failed_total	このメソッドが呼ばれ、リトライをしたが、最終的に成功しなかった回数を表示します。	カウンター
ft_<メソッド名>_retry_retries_total	このメソッドがリトライされた合計の回数を表示します。	カウンター

Timeoutのメトリック

- applicationスコープで規定されたメトリック（WAS Libertyで取得可能な項目）
  これらのメトリックはMicroProfile Metrics REST APIからデフォルトで取得可能です。

メトリック	説明	タイプ
ft_<メソッド名>_timeout_execution_duration_seconds	このメソッドの実行に要した時間（※）のヒストグラムを表示します。	ヒストグラム
ft_<メソッド名>_timeout_calls_timed_out_total	このメソッドがタイムアウトした回数を表示します。	カウンター
ft_<メソッド名>_timeout_calls_not_timed_out_total	このメソッドがタイムアウトせずに完了した回数を表示します。	カウンター

（※）単位については、Prometheusフォーマットの場合は秒、JSONフォーマットの場合はナノ秒となり、メトリック名にも"_seconds"は付きません。

CircuitBreakerのメトリック

- applicationスコープで規定されたメトリック（WAS Libertyで取得可能な項目）
  これらのメトリックはMicroProfile Metrics REST APIからデフォルトで取得可能です。

メトリック	説明	タイプ
ft_<メソッド名>_circuitbreaker_calls_succeeded_total	このメソッドの呼び出しがサーキット・ブレーカーにより許可され、成功した回数を表示します。	カウンター
ft_<メソッド名>_circuitbreaker_calls_failed_total	このメソッドの呼び出しがサーキット・ブレーカーにより許可され、失敗した回数を表示します。	カウンター
ft_<メソッド名>_circuitbreaker_calls_prevented_total	このメソッドの呼び出しがサーキット・ブレーカーがオープンのために行われなかった回数を表示します。	カウンター
ft_<メソッド名>_circuitbreaker_calls_open_total_seconds	このメソッドのサーキット・ブレーカーがOpen状態にあった合計時間（※）を表示します。	ゲージ
ft_<メソッド名>_circuitbreaker_calls_half_open_total_seconds	このメソッドのサーキット・ブレーカーがHalf-open状態にあった合計時間（※）を表示します。	ゲージ
ft_<メソッド名>_circuitbreaker_calls_close_total_seconds	このメソッドのサーキット・ブレーカーがClosed状態にあった合計時間（※）を表示します。	ゲージ
ft_<メソッド名>_circuitbreaker_calls_opened_total	このメソッドのサーキット・ブレーカーがClosed状態からOpen状態に遷移した合計回数を表示します。	カウンター

（※）単位については、Prometheusフォーマットの場合は秒、JSONフォーマットの場合はナノ秒となり、メトリック名にも"_seconds"は付きません。

Bulkheadのメトリック

- applicationスコープで規定されたメトリック（WAS Libertyで取得可能な項目）
  これらのメトリックはMicroProfile Metrics REST APIからデフォルトで取得可能です。

メトリック	説明	タイプ
ft_<メソッド名>_bulkhead_concurrent_executions	このメソッドの現在同時実行されている数を表示します。	ゲージ
ft_<メソッド名>_bulkhead_calls_accepted_total	このメソッドの呼び出しがバルクヘッドにより受理された合計回数を表示します。	カウンター
ft_<メソッド名>_bulkhead_calls_rejected_total	このメソッドの呼び出しがバルクヘッドにより却下された合計回数を表示します。	カウンター
ft_<メソッド名>_bulkhead_execution_duration_seconds	このメソッドの実行時間（※）のヒストグラムを表示します。これはバルクヘッドのキューで待ち状態だった時間を含みません。	ヒストグラム
ft_<メソッド名>_bulkhead_waiting_queue_population （※A）	このメソッドのバルクヘッドのキューで待ち状態にある数を表示します。	ゲージ
ft_<メソッド名>_bulkhead_waiting_duration_seconds（※A）	このメソッドのバルクヘッドのキューで待ち状態にあった時間（※）のヒストグラムを表示します。	ヒストグラム

（※）単位については、Prometheusフォーマットの場合は秒、JSONフォーマットの場合はナノ秒となり、メトリック名にも"_seconds"は付きません。
（※A）メソッドに@Asynchronous でアノテートされた場合だけ出力されます。

Fallbackのメトリック

- applicationスコープで規定されたメトリック（WAS Libertyで取得可能な項目）
  これらのメトリックはMicroProfile Metrics REST APIからデフォルトで取得可能です。

メトリック	説明	タイプ
ft_<メソッド名>_fallback_calls_total	このメソッドのフォールバック・ハンドラーまたはフォールバック・メソッドが呼ばれた合計回数を表示します。	カウンター

参考リンク

- MicroProfile Fault Tolerance 1.0
  https://github.com/eclipse/microprofile-fault-tolerance/releases/download/1.0/microprofile-fault-tolerance-spec-1.0.pdf
- MicroProfile Fault Tolerance 1.1 **18.0.0.3+**
  https://github.com/eclipse/microprofile-fault-tolerance/releases/download/1.1/microprofile-fault-tolerance-spec-1.1.pdf
- IBM Knowledge Center - Libertyでのマイクロサービスの回復力の向上
  https://www.ibm.com/support/knowledgecenter/ja/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/twlp_microprofile_fault_tolerance.html
- 連載「Microservice Builder」第 3 回 MicroProfile Fault Toleranceで障害許容性を向上させる
  https://www.ibm.com/developerworks/jp/websphere/library/icp/msb_introduction/3.html
- Building fault-tolerant microservices with the @Fallback annotation
  https://openliberty.io/guides/microprofile-fallback.html
- Preventing repeated failed calls to microservices
  https://openliberty.io/guides/circuit-breaker.html
- Limiting the number of concurrent requests to microservices
  https://openliberty.io/guides/bulkhead.html

End of File