CM-03 スレッド制御機能

■ 概要

本機能は、「CL-03 イベント処理実行機能」の基盤となる共通機能であり、非同期実行やマルチスレッド制御に関する機能を提供し、開発者がスレッドの実行処理を安全かつ簡単に記述することができる。通常、開発者は「CL-03 イベント処理実行機能」を利用すれば本機能を直接利用する必要はないが、開発プロジェクト独自のマルチスレッド処理が必要な場合にも利用できる。

- タスクの非同期実行
  - サーバとのファイル送受信処理などの長時間処理をバックグランドで実行させたいといった要求や、画面上でのボタンクリック後もバックグラウンドでロジックを実行させることで画面操作を継続させたいといった要求がある。このようなケースでは、別スレッドを作成し非同期処理させる必要がある。
  - 本機能では、マルチスレッド制御にまつわる複雑さを開発者から隠蔽し、簡単かつ安全に実装することを可能にするヘルパークラス(InvocationHelper クラス)を提供している。

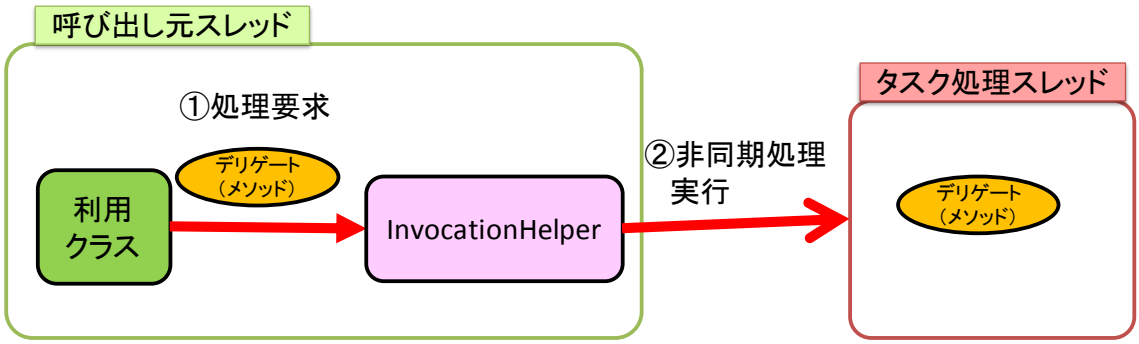

図 1 非同期処理実行処理の動作概念図

- コンテキストとスコープによるマルチスレッド制御
  - InvocationHelper クラスが扱う非同期処理では、「コンテキスト」と呼ばれるスレッド間で安全なデータのやり取りを行うためのオブジェクトが、複雑なマルチスレッド制御処理を隠蔽し、マルチスレッド制御の実装における開発者の負担を軽減している。
  - 「コンテキスト」には、以下の2種類が存在する。
    - スレッドコンテキスト(InvocationContext)
      - マルチスレッド制御に必要な情報を格納するクラス。タイムアウト時間など非同期実行時に必要な制御情報が含まれている。
      - 後述の「透過的な非同期実行」を利用する場合、開発者は「スレッドコンテキスト」のみにアクセスすればよい。「スレッドコンテキスト」は「スレッド実行時コンテキスト」(後述)のインスタンスを生成／管理しており、例えば、開発者によりキャンセル指示があると、管理している「スレッド実行時コンテキスト」へキャンセル指示を伝搬させる。

- ✧ スレッド実行時コンテキスト（InvocationPerCallContext）
  - 呼び出し元スレッドからタスク処理用のスレッド（タスクスレッド）が起動されるたびに、「スレッドコンテキスト」により生成される実行時用のコンテキスト。
  - 処理結果取得の待機、タイムアウト、キャンセルによる待機状態からの解放、タスクスレッドの処理結果取得など、タスクの実行状態を管理する。
  - 後述の「透過的な非同期実行」を利用すると、フレームワークにより「スレッド実行時コンテキスト」インスタンスが生成され、必要な処理が実行されるので、開発者が意識することは原則ない。

- ➢ 「コンテキスト」により、タスクの非同期実行における複雑なマルチスレッド制御を簡単に実施することができる。
  - ✧ タスクスレッドの処理結果の取得
    - .NET の標準機能を使用して非同期で実施したタスクの処理結果を取得するには、以下のような複雑なマルチスレッド制御の実装が必要であり、誤った実装方法により誤動作を起こしたり、その原因の特定にも時間がかかったりするなど開発者への負担が大きい。
      - ➢ 呼び出し元スレッドはタスクスレッドの処理結果を取得するまで待機する。
      - ➢ タスクスレッドは、処理が完了したら待機中の呼び出し元スレッドに通知する。呼び出し元スレッドは、待機状態から復帰後に処理結果を取得する。
    - 本機能では、図 2 のように、呼び出し元スレッドが「スレッド実行時コンテキスト」の中で待機し、タスクスレッドが「スレッド実行時コンテキスト」に対して処理完了を通知すると、待機状態から復帰し処理結果を取得し、処理を継続可能とする。

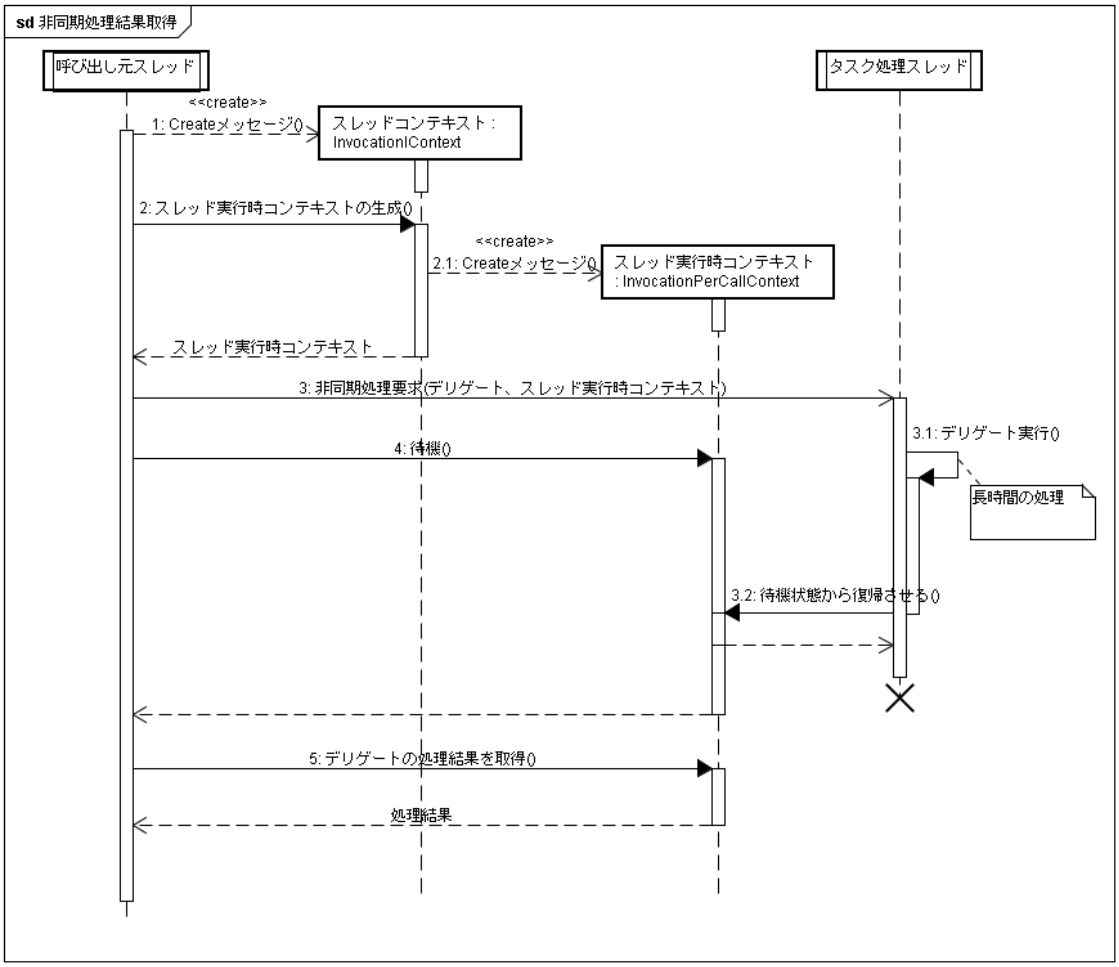

図 2 非同期処理結果取得時のマルチスレッド制御

✧ 長時間処理のタイムアウト

- 呼び出した処理が何らかの原因により終了せず想定以上に長時間にわたり制御が戻らない場合などに、タイムアウトしたいケースがある。
  図 3 のように、呼び出し元スレッドは、処理結果を取得できるまで実行時コンテキストの中で待機するが、タイムアウト用の「スレッド実行時コンテキスト」(TimeoutInvocationPerCallContext)に設定された時間だけ経過すると自動的に待機状態から復帰し処理を継続することができる。

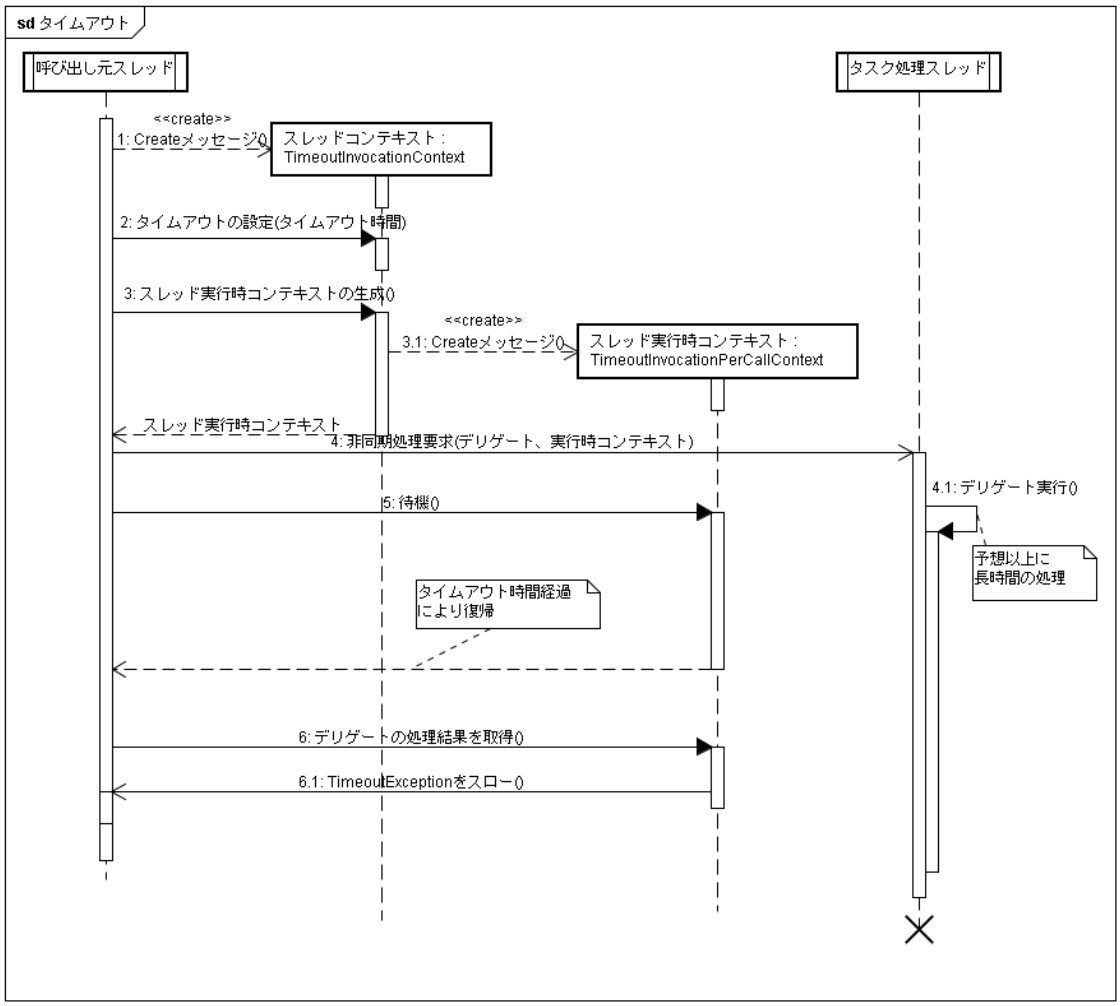

図 3 タイムアウト時のマルチスレッド制御

- タイムアウト後もタスクスレッドは実行を継続している。通常、たいていの処理であればリソースをそれほど消費することがないため、タスク処理スレッドの処理が自然に終了するまで放っておいても問題にならない。一方、無限ループを扱う処理や負荷が異常に高い処理、処理結果が反映されないように制御が必要な処理など、タイムアウト時に即座に処理を中止しないといけないケースでは、タイムアウトフラグをチェックし、処理を中断するよう実装する必要がある。

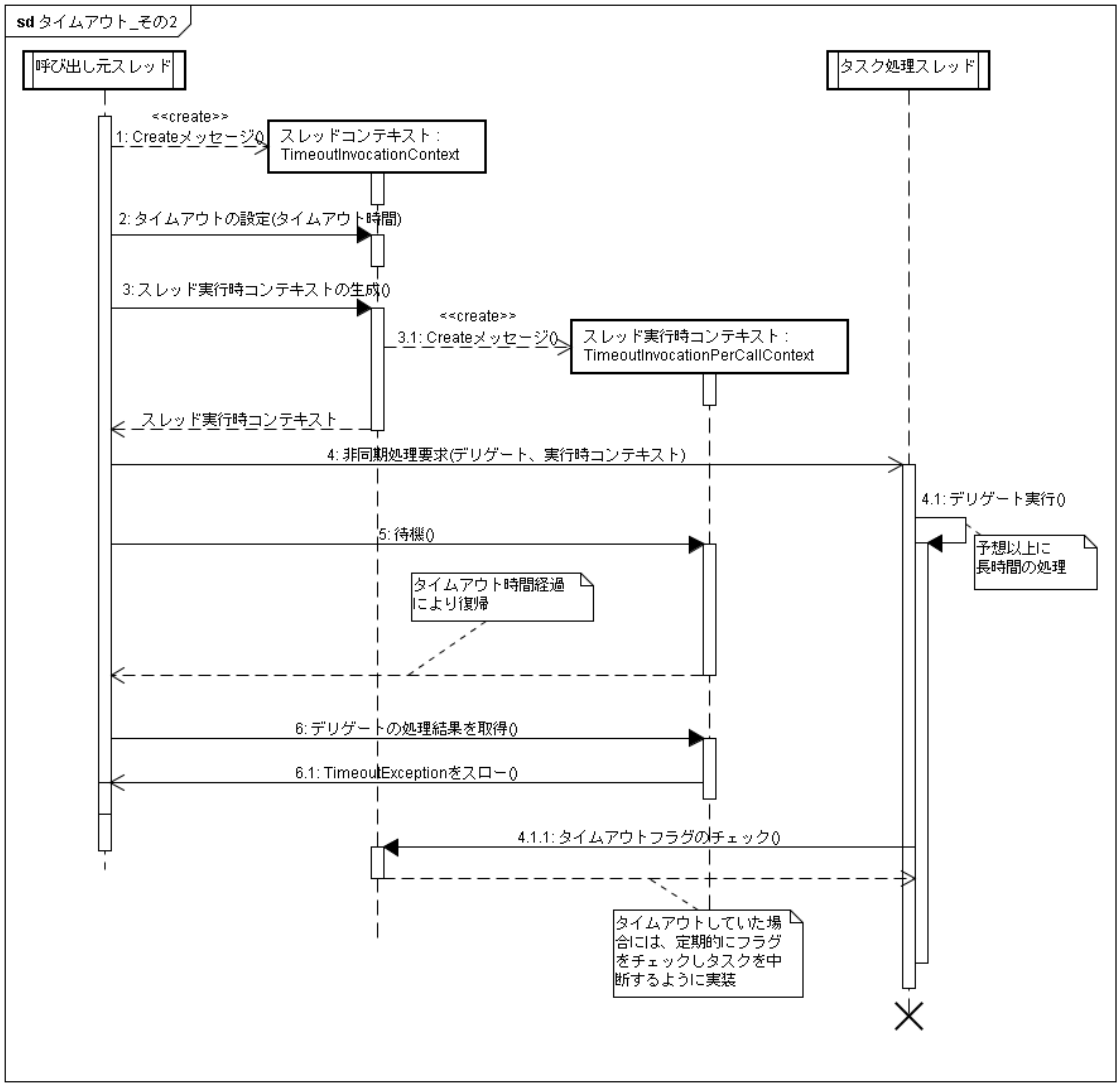

図 4 タイムアウトフラグの制御

- ✧ 長時間処理のキャンセル指示
  - ● ファイル送信や大量データ更新など、長時間にわたる業務処理を実行している場合に、キャンセルボタンを押下するなどして明示的に処理をキャンセルしたいという場面がある。.NET の標準機能により、非同期処理のキャンセルを実装する際には、以下の点について考慮が必要となる。
    - ➢ タスクの呼び出し元スレッドは、タスクスレッドの処理が完了し、結果を取得するまで待機する必要がある。このため、メインスレッドから直接タスクスレッドを呼び出してしまうと、メインスレッド自体が待機状態となってしまい、ユーザによるキャンセルボタン押下等のイベントを受け付けることができなくなってしまう。そこで、キャンセル指示用のスレッド（通常、メインスレッド）とは別のスレッドからタスクの非同期呼び出しを実施するなどの対処が必須となる。

- スレッドをアボートすることは推奨されていないため、非同期実行中のタスクスレッドを安全に終了させるためには、呼び出し元スレッドによるキャンセル指示により、タスクスレッドが自律的に処理を停止するように実装する必要がある。たとえば、呼び出し元スレッド側でスレッドセーフなオブジェクト上にあるキャンセル用のフラグをオンにし、タスクスレッド側で、そのフラグを定期的に確認し処理を中断するといった複雑な実装を業務開発者が行うことになる。

本機能では、長時間処理に対するキャンセル指示の機構を、開発者が複雑なマルチスレッドプログラミングモデルを意識することなく、簡易に実装できる仕組みを提供する。

- 図 5 のように、キャンセル指示をするスレッド（通常、メインスレッド）が、キャンセル用の「スレッドコンテキスト」(CancelableInvocationContext)に対してキャンセル指示すると、タスク処理の呼び出し元スレッド（メインスレッドとは異なる）は、待機状態から復帰し処理を継続することができる。

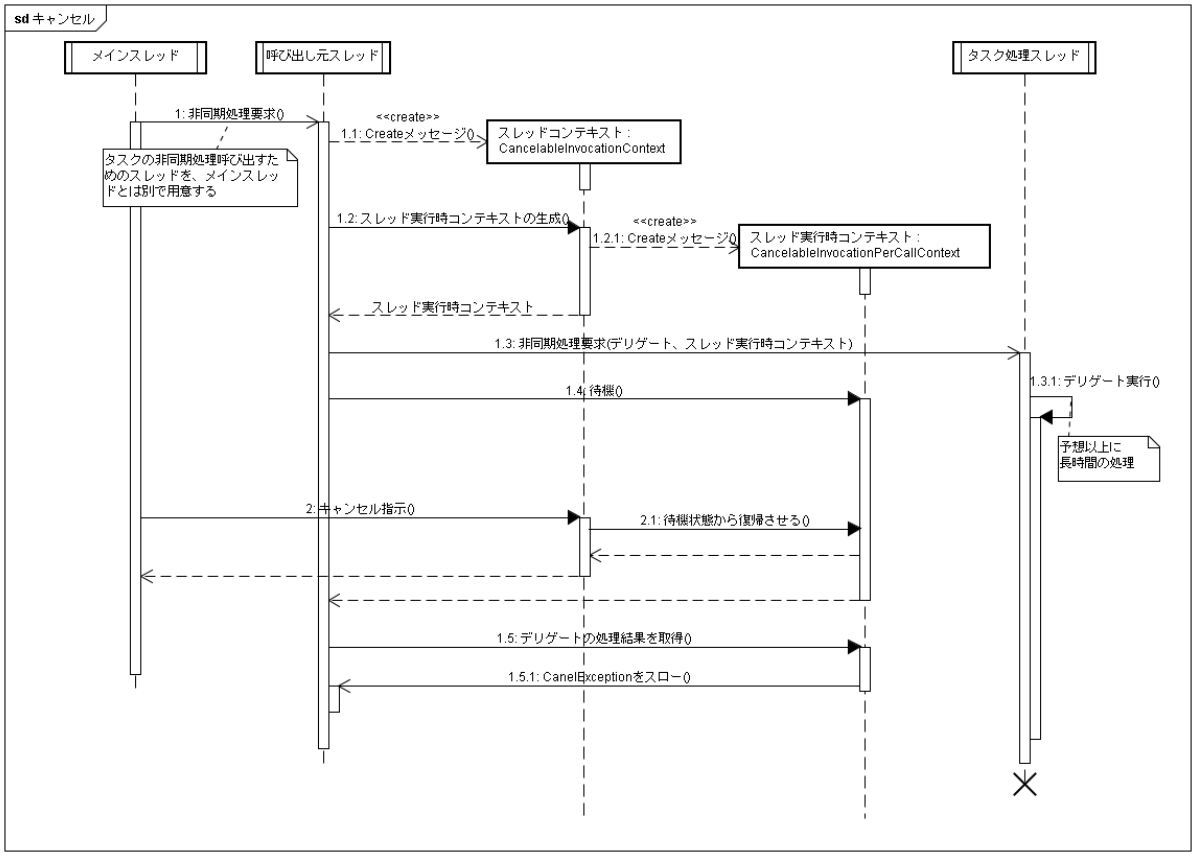

図 5 キャンセル時のマルチスレッド制御

- キャンセル後もタスクスレッドは実行を継続している。通常、たいていの処理であればリソースをそれほど消費することがないため、タスクスレッドの処理が自然に終了するまで放っておいても問題にならない。一方、無限ループを扱う処理や負荷が異常に高い処理、処理結果が反映されないように制御が必

要な処理など、キャンセル時に即座に処理を中止しないといけないケースでは、キャンセルフラグをチェックし、処理を中断するよう実装する必要がある。本機能では、図 4 のように、スレッドセーフな「スレッドコンテキスト」にあるキャンセルフラグを、簡単かつ安全に取得する機構を提供する

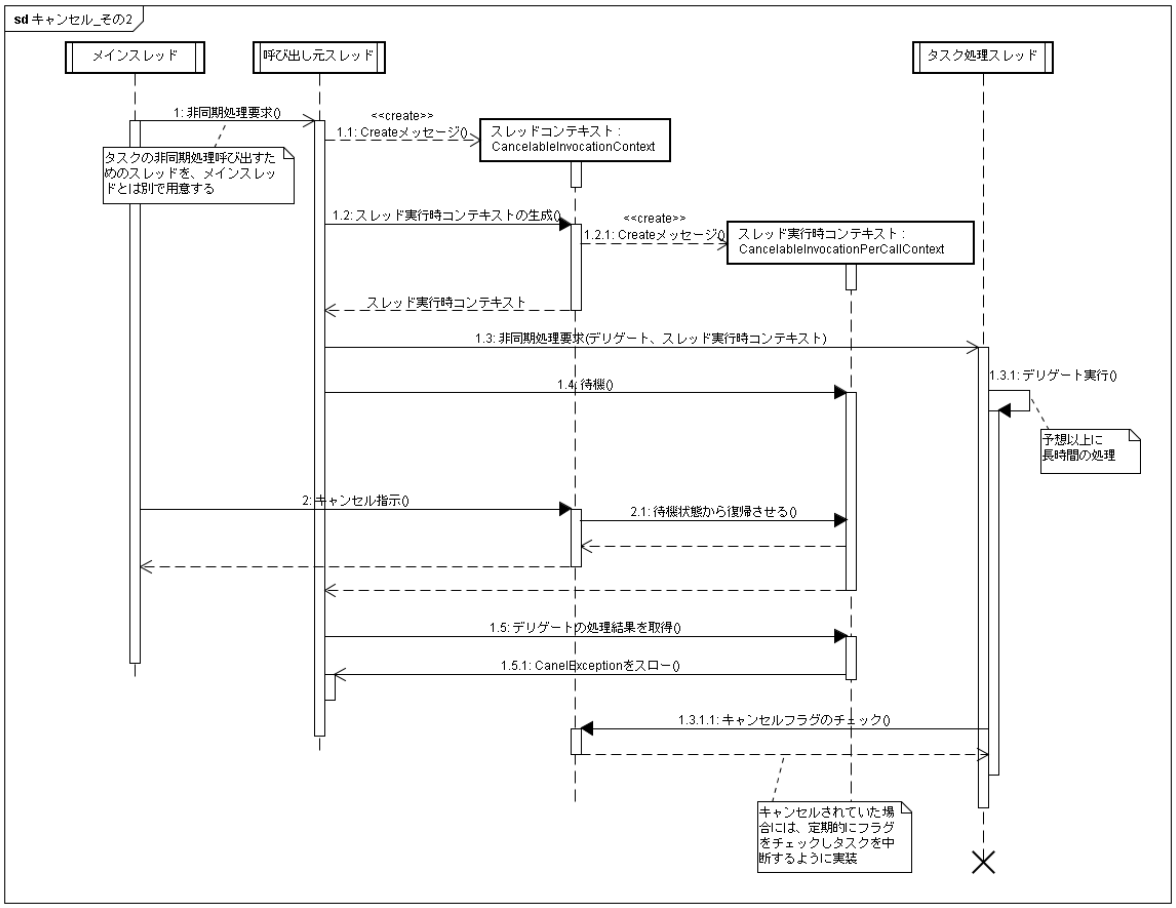

図 6 キャンセルフラグの制御

- 「コンテキスト」の生存期間を管理するための概念として、「スコープ」と呼ばれる区間を導入し、同一「スコープ」区間内では、同一の「スレッドコンテキスト」を暗黙的に（メソッド引数として渡すことなく）参照できるようになっている。「スコープ」の区間は、InvocationScope クラスを using 句で囲った範囲で表現できる。スコープ内ではコンテキストを、コンテキストの型ごとに管理しているため、同一型のコンテキストを複数定義することはできない。また、コンテキストを ThreadStatic（スレッドごとに一意な静的フィールド）に保持するので、スレッドごとに独立してコンテキストを管理することができる。また、スコープの入れ子にも対応しており、スコープが入れ子になっている場合には、現在のスコープで対象の型のコンテキストが設定されていない場合でも、外側（親側）のスコープで対象の型のコンテキストが存在すれば、そのコンテキストを利用することができる。

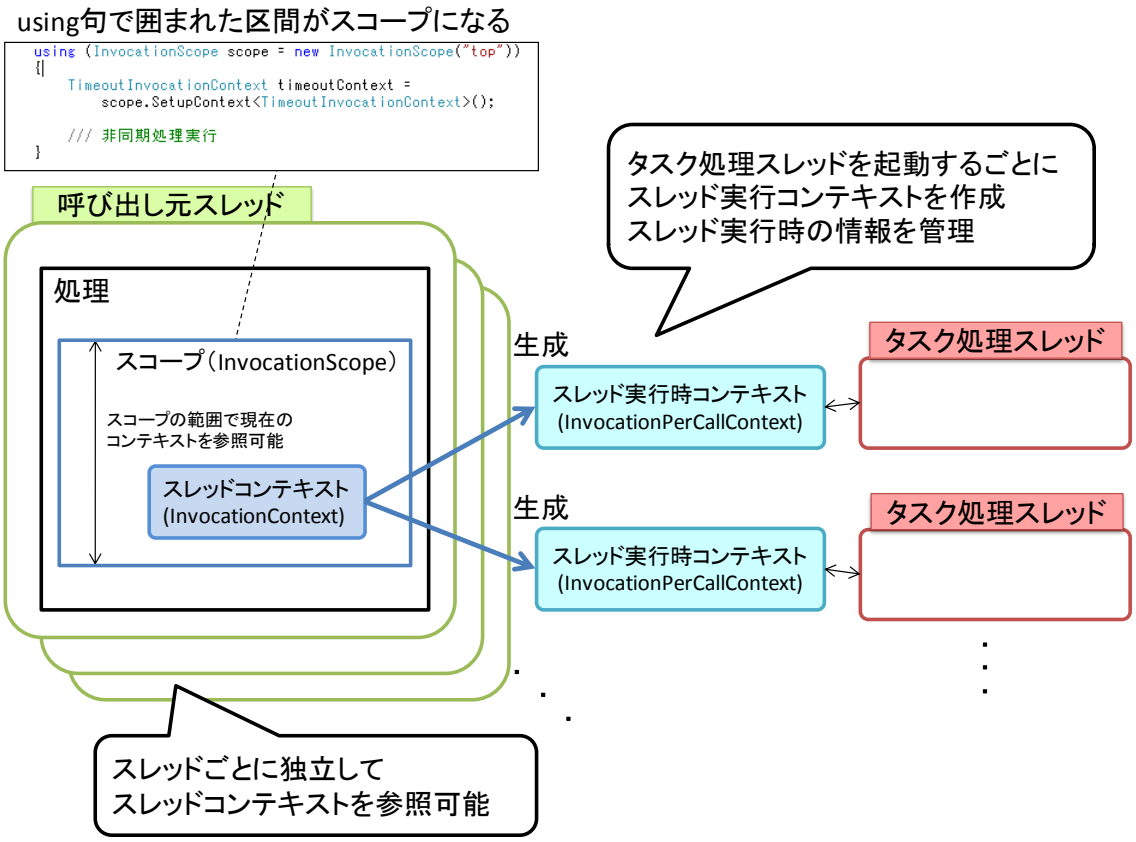

図 7 スコープとコンテキストの概念図

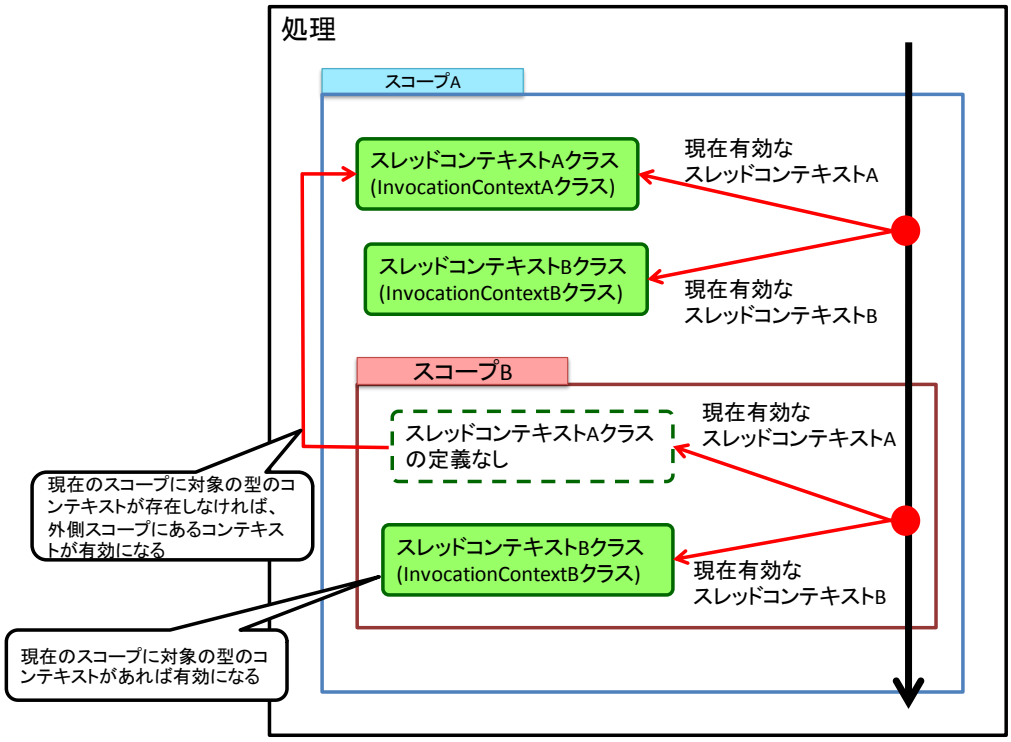

図 8 スコープによる「スレッドコンテキスト」の有効範囲の指定

- タイムアウト可能なタスクの透過的な非同期実行
  - 本機能により、タイムアウト制御の対象としたいメソッドに TimeoutCallHandler 属性を付与するだけで、設定した時間を経過するとタイムアウトして呼び出し元スレッドに制御を戻すといった処理を透過的に実現できる。
  - 本機能は、「CM-02 インスタンス管理機能」の AOP 機能を利用して、同期的なメソッド呼び出しを InvocationHelper クラスによる非同期実行処理に差し替えることで実現している。
  - 呼び出し元スレッドは、タスクスレッド側の処理結果が完了するまで、タイムアウト機能用の「スレッド実行時コンテキスト」(TimeoutInvocationPerCallContext)の中で、待機状態に入る。タスクスレッドは、処理が完了すると、「スレッド実行時コンテキスト」に対して処理完了を通知する。それを契機に、呼び出し元スレッドは即座に待機状態から抜け出し、処理結果を取得後、処理を継続する。
  - TimeoutCallHandler 属性に設定したタイムアウト時間を経過してもタスクスレッドの非同期処理が終了しない場合、呼び出し元スレッドは待機状態から抜け出し処理を再開する。この時、TimeoutException がスローされる。(例外をスローしないように設定することも可能)
  - 注意点として、タイムアウト後もタスクスレッドは中断されていない。上述したとおり、業務要件等に応じて、キャンセルしたかどうかをチェックし中断するロジックを実装する。

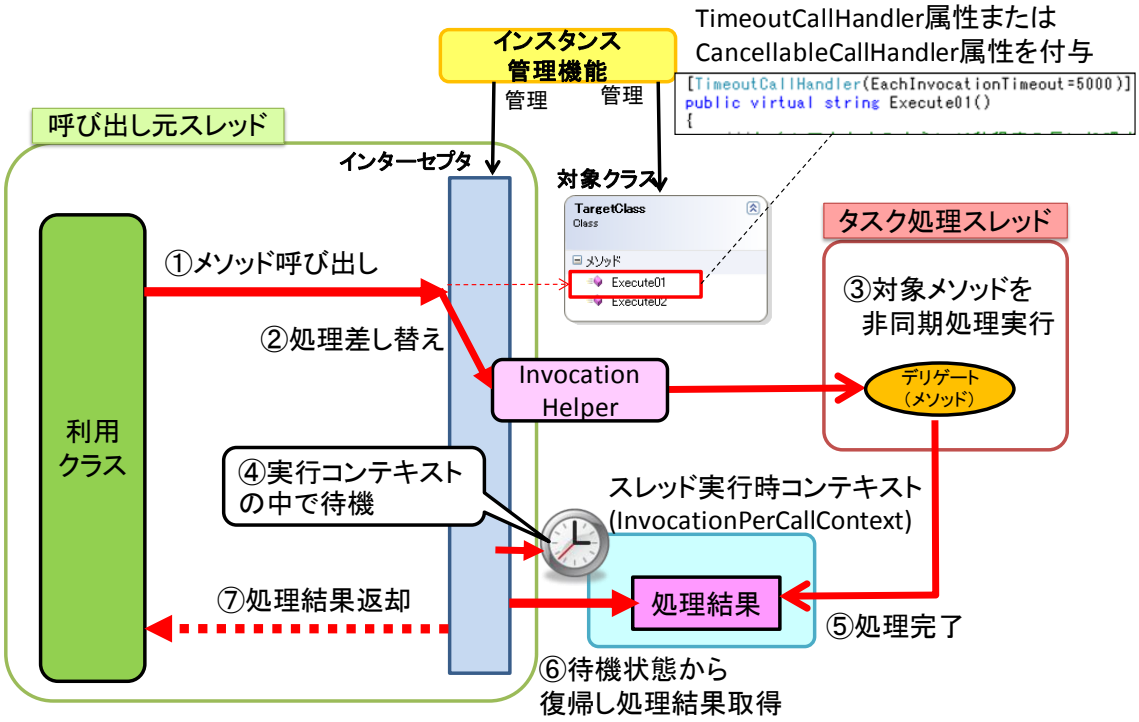

図 9 透過的な非同期処理実行

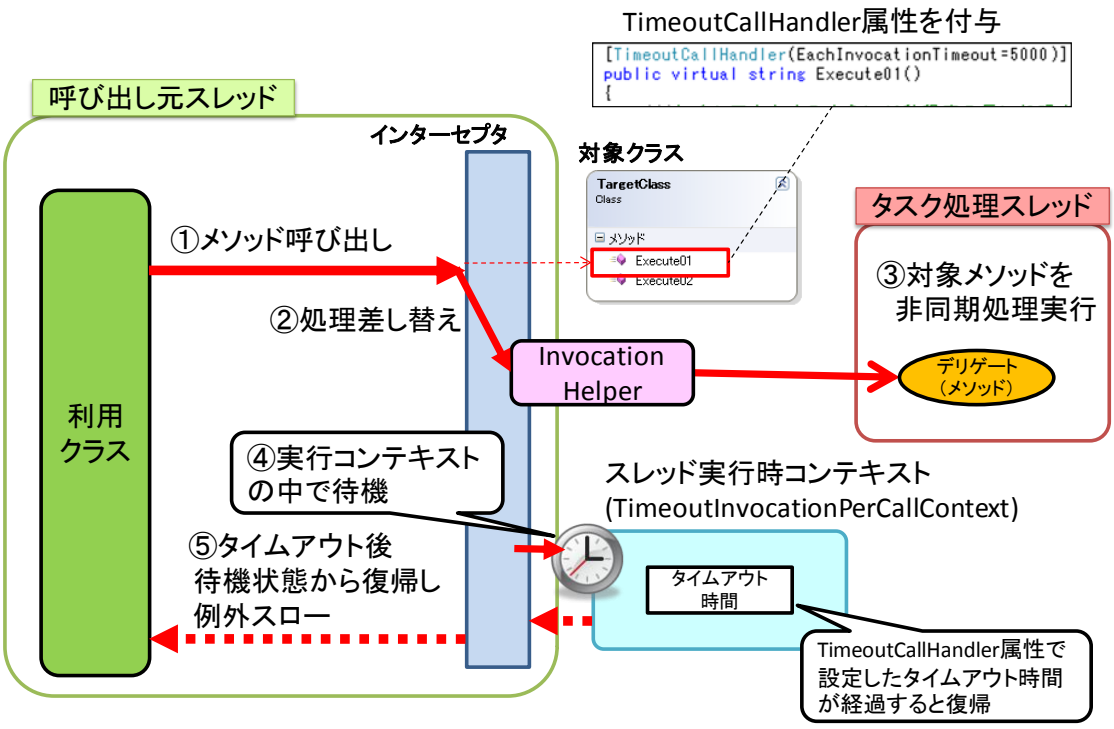

図 10 透過的な非同期処理実行（タイムアウト時）

- キャンセル可能なタスクの透過的な非同期実行
  - 本機能により、メソッドに CancelableCallHandler 属性を付与するだけで、キャンセル可能なタスク処理スレッドと実行することができる。
  - CancelableCallHandler は TimeoutCallHandler を継承しているため前述の簡易タイムアウト機能も利用できる。
  - タイムアウトと同様、「CM-02 インスタンス管理機能」の AOP を利用して、通常の同期的なメソッド呼び出しを InvocationHelper クラスによる非同期実行処理に差し替えている。
  - 上述したとおり、呼び出し元スレッドはタスクスレッドが完了するのを待機しているため、キャンセル指示を出すことができない。このため、呼び出し元スレッドとは別のスレッドでキャンセル指示を出す必要があり、通常はメインスレッドで実施する。キャンセル指示は、キャンセル用の「スレッドコンテキスト」である CancelableInvocationContext クラスが提供する API を利用する。
  - キャンセル実行後、呼び出し元スレッドは即座に待機状態から抜け出し処理を継続することができる。このとき InvocationCancelException がスローされる。（例外をスローしないように設定することも可能）
  - 注意点として、キャンセル後もタスクスレッドは中断されていない。上述したとおり、業務要件等に応じて、キャンセルしたかどうかをチェックし中断するロジックを実装する。

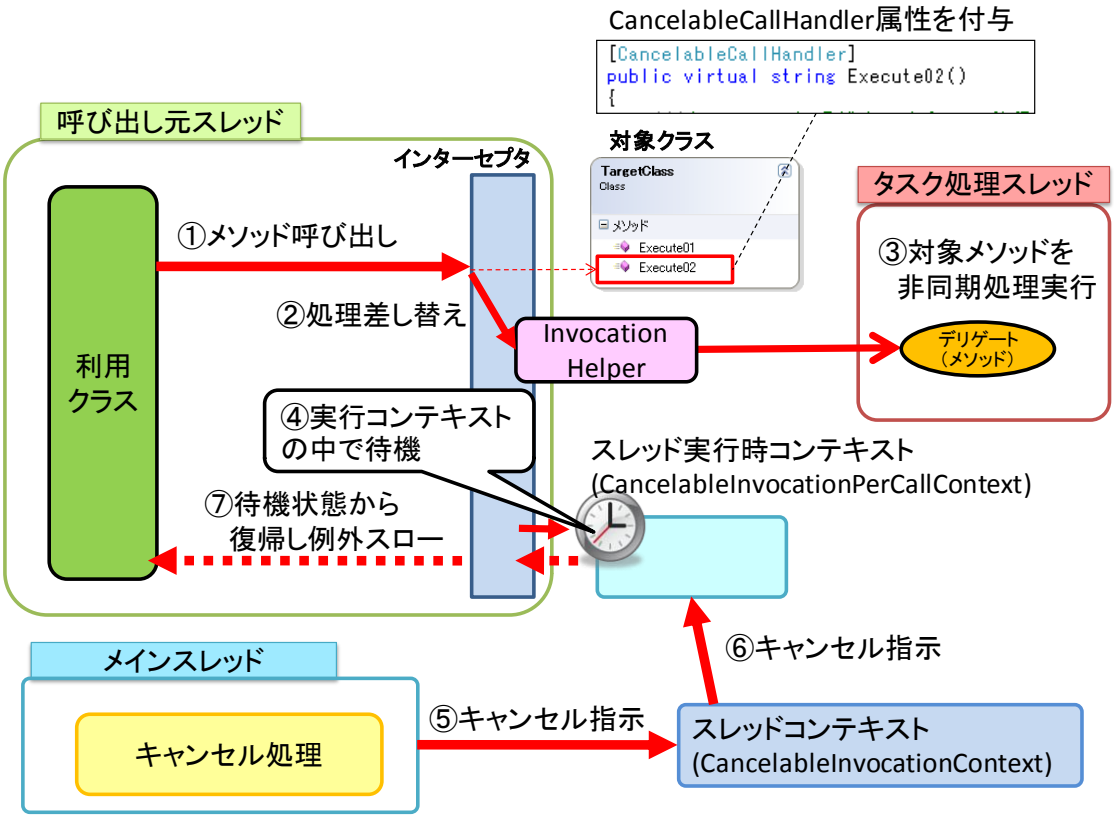

図 11 透過的な非同期処理実行(キャンセル時)

- 非 UI スレッドからの透過的な UI スレッド実行
  - Windows Forms アプリケーションでは、UI スレッド（UI コントロールを生成したスレッド。通常、メインスレッド）以外のスレッドからコントロールを直接操作してはならない。開発者は、System.Windows.Control クラスの Invoke メソッドや BeginInvoke メソッドを利用し、デリゲートに記述した UI コントロールの操作を UI スレッド上で実行する必要がある。通常、この種の開発では、スレッドを意識した実装を開発者に強いるため、開発者の負担も大きく、品質低下の可能性も高くなる。

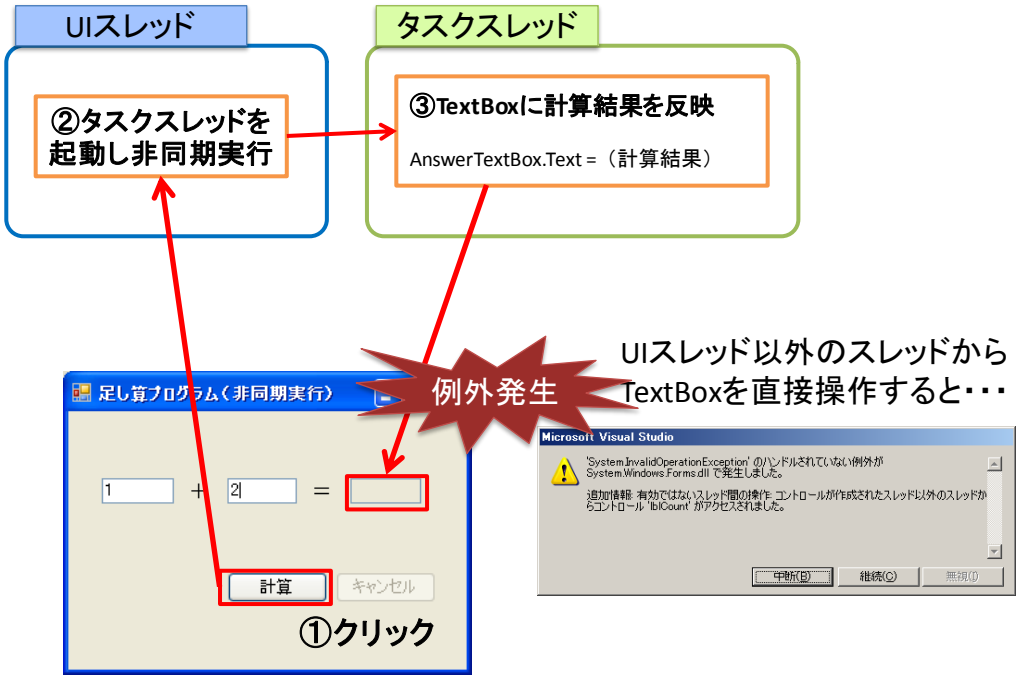

図 12 誤ってタスクスレッドから直接 UI コントロールを操作した時のイメージ

  - 本機能により、タスクスレッド（非 UI スレッド）側から UI コントロールを操作する処理を実行するメソッドに UICallHandler 属性を付与するだけで、スレッドの違いを意識することなく UI コントロールを操作できる。前述した「タスクの透過的な非同期実行」と同一の実装スタイルを実現している。
  - 「CM-02 インスタンス管理機能」の AOP を利用して、インターセプタが通常の同期的なメソッド呼び出しを System.Windows.Control クラスの Invoke メソッドの実行にすり替えている。
  - System.Windows.Control クラスの Invoke メソッドの実行は、UIInvocationHelper クラスにより実現されている。UIInvocatinHelper は、タスクの非同期実行時に使用した InvocationHelper クラスの UI スレッド実行機能版である。

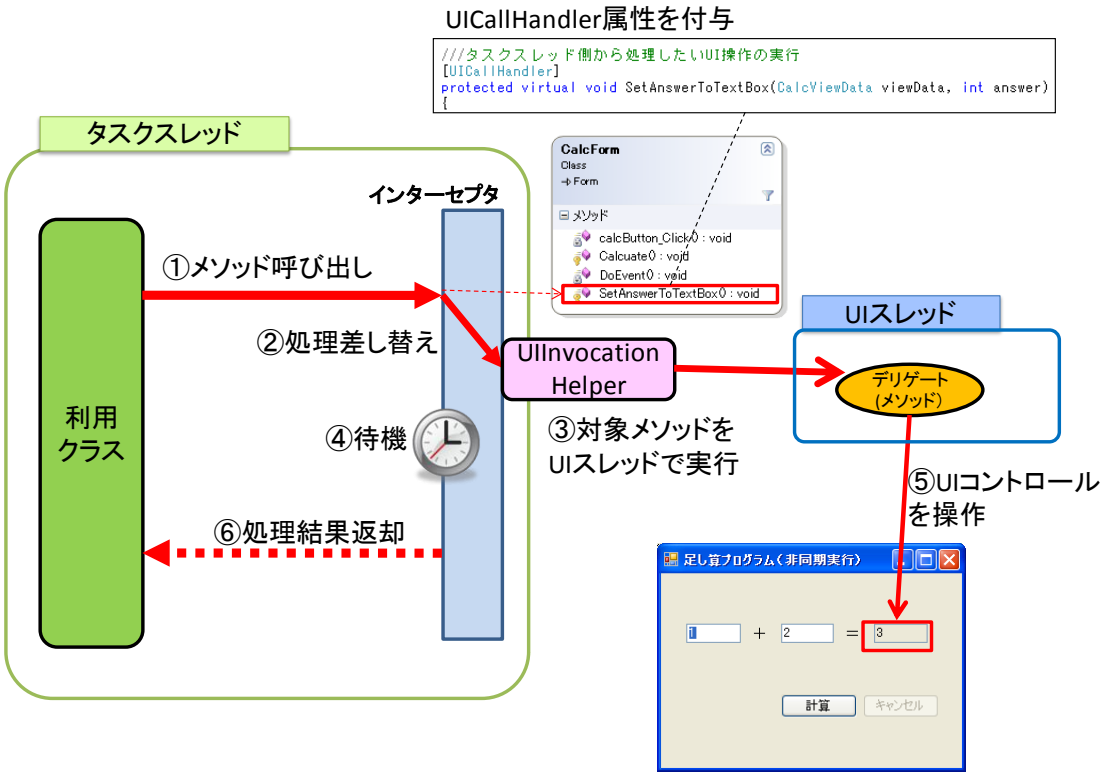

図 13 非 UI スレッドによる透過的な UI スレッド実行

■ 使用方法

◆ Invocation Helper クラスを使った非同期処理の実装

InvocationHelper クラスを使ってタスクの非同期実行を実施するには以下のメソッドを使用する。

表 1 InvocationHelper クラスのメソッド一覧

項番	メソッド	引数	戻り値	説明
1	public static IAsyncResult BeginInvoke(Delegate method, object[] args, AsyncCallback callback, object state)	第1引数:非同期実行するデリゲート 第2引数:デリゲートに渡す引数 第3引数: デリゲートの処理が完了する際のコールバックメソッドを参照する AsyncCallBack 第4引数: ユーザの引き渡しデータ	デリゲートの非同期実行に対する IAsyncResult オブジェクト	スコープ(InvocationScope)に設定された「スレッドコンテキスト」の情報をもとに、デリゲートを非同期実行する。
2	public static object EndInvoke(IAsyncResult ar)	デリゲートの非同期実行に対する IAsyncResult オブジェクト	デリゲートの実行結果	デリゲートの非同期実行結果を取得する。

InvocationHelper は内部的に、デリゲートの APM(Asynchronous Programming Model)を利用しており、BeginInvoke メソッドの第2引数～第4引数は、.NET のデリゲートの非同期実行時の BeginInvoke メソッドの引数とほぼ同じだが、第4引数の引き渡しデータはついては若干異なっているので注意が必要である。通常のデリゲートの非同期実行では、IAsyncResult オブジェクトの AsyncState プロパティに格納されているが、InvocationHelper による実行の場合は、IAsyncResult の AsyncState プロパティには Terasoluna.Threading.InvocationAsyncState オブジェクトが格納されており、IvocationAsyncState.State プロパティで実際の引き渡しデータを取得できる。
なお、InvocationHelper はスコープ(InvocationScope)に設定する「スレッドコンテキスト」の情報をもとにデリゲートを非同期実行するため、InvocationHelper の実行処理を InvocationScope の using 句で囲み、InvocationScope に「スレッドコンテキスト」(InvocationContext)をセットしておく。

以下に、InvocationHelper クラスを使った非同期実行処理の例を示す。[1]

---

[1] 実際に下記サンプルコードの Test メソッドを実行するためには、Test メソッドの実行前後で、「CM-01 アプリケーション起動・終了機能」が提供する TerasolunaFramework クラスによるフレームワークの起動・終了処理が必要である(ここでは省略している)。これ以降に説明しているサンプルコードについても同様である。

```
    class InvocationSample
    {
        public static void Test()
        {
            /// スレッド制御用のスコープの設定(using句で囲む)
            using (InvocationScope scope = new InvocationScope("root"))
            {
                Thread.CurrentThread.Name = "呼び出し元スレッド";
                Console.WriteLine("[{0}][{1}]非同期処理呼び出し開始",
                                                    Thread.CurrentThread.Name, DateTime.Now);
                ///  「スレッドコンテキスト」の生成
                InvocationContext context = scope.SetupContext<InvocationContext>();
                int num1 = 1;
                int num2 = 2;
                Func<int, int, int> func = new Func<int, int, int>(Add);   //デリゲート
                string stateObjct = "ABC"; //引き渡しデータ
                ///非同期処理実行
                IAsyncResult asyncResult = InvocationHelper.BeginInvoke(
                      func, new object[] {num1, num2 }, new AsyncCallback(Callback), stateObjct);
                Console.WriteLine("[{0}][{1}]非同期処理呼び出し完了",
                                                    Thread.CurrentThread.Name, DateTime.Now);
                ///非同期処理結果の取得（実行結果取得まで待機）
                object result = InvocationHelper.EndInvoke(asyncResult);
                Console.WriteLine("[{0}][{1}]処理結果:{2}+{3}={4}", Thread.CurrentThread.Name,
                                                        DateTime.Now, num1, num2, result);
            }
            Thread.Sleep(3000);   //3秒sleep
            Console.WriteLine("[{0}][{1}]メインスレッド終了",
                                                    Thread.CurrentThread.Name, DateTime.Now);
        }

        /// 非同期処理対象メソッド（足し算）
        private static int Add(int num1, int num2)
        {
            Thread.CurrentThread.Name = "タスクスレッド";
            Console.WriteLine("[{0}][{1}]Addメソッド開始",
                                                    Thread.CurrentThread.Name, DateTime.Now);
            Thread.Sleep(5000);   /// 5秒sleep
            Console.WriteLine("[{0}][{1}]Addメソッド終了",
                                                    Thread.CurrentThread.Name, DateTime.Now);
            /// ElapsedTimeプロパティで非同期処理実行時間を取得
            Console.WriteLine("[{0}]非同期処理実行時間：{1}ミリ秒",Thread.CurrentThread.Name,
    InvocationScope.Current.GetContext<InvocationContext>().ElapsedTime.TotalMilliseconds);
            // 足し算の結果を返す
            return num1 + num2;
        }
        /// コールバックメソッド
        private static void Callback(IAsyncResult result)
        {
            /// 引き渡しデータを取得
            Console.WriteLine("[{0}][{1}]CallBackメソッド実行：引き渡しデータ=¥"{2}¥"",
                Thread.CurrentThread.Name, DateTime.Now,
                  (result.AsyncState as InvocationAsyncState).State);
        }
    }
}
```

リスト 1 InvocationHelper による非同期処理の記述例

上記記述例の実行結果は、以下のとおりである。
Add メソッドが別スレッドにより非同期実行されているのが分かる。

```
[呼び出し元スレッド][2009/10/05 23:41:23]非同期処理呼び出し開始
[呼び出し元スレッド][2009/10/05 23:41:23]非同期処理呼び出し完了
[タスクスレッド][2009/10/05 23:41:23]Add メソッド開始
[タスクスレッド][2009/10/05 23:41:28]Add メソッド終了
[タスクスレッド]非同期処理実行時間：5031.3466 ミリ秒
[呼び出し元スレッド][2009/10/05 23:41:28]処理結果:1+2=3
[タスクスレッド][2009/10/05 23:41:28]CallBack メソッド実行：引き渡しデータ="ABC"
[呼び出し元スレッド][2009/10/05 23:41:31]メインスレッド終了
```

リスト 2 非同期処理の実行結果

◆ タスクの透過的な非同期実行の実装

タイムアウトやキャンセル可能なタスクを非同期実行するには、開発者はカスタム属性を付与するだけで、同期呼び出し時と同様の方法で、透過的に非同期実行処理を行うことができる。

(1) 非同期実行対象メソッドの実装

本機能は、「CM-02 インスタンス管理機能」の AOP により実現されているため、非同期実行対象のメソッドを持つクラスに SetDefautlInterceptor(または SetInterceptor 属性)を付与する必要がある。
インターセプタとして VirtualMethodInterceptor クラスを利用するため、
SetDefaultInterceptor 属性の引数には、VirtualMethodInterceptor の Type 型を指定する。

```
/// <summary>
///  非同期処理対象のクラス
/// Unityのインターセプタを利用するため、SetInterceptor属性を付与する必要がある
/// </summary>
[SetDefaultInterceptor(typeof(VirtualMethodInterceptor))]
public class TargetClass
{
・・・
```

リスト 3 SetInterceptor 属性の記述例

非同期処理対象のメソッドには、用途に合わせて、以下のどちらかのカスタム属性を付与する。
なお、CancelableCallHandler 属性は TimeoutCallHandler 属性を継承しており、簡易タイムアウト機能も持っている。

表 2 非同期実行対象のメソッドに付与する属性

項番	属性	利用方法
1	TimeoutCallHandlerAttribute	タイムアウト可能な処理として非同期実行したいメソッドに付与する。
2	CancelableCallHandlerAttribute	キャンセル可能な処理として非同期実行したいメソッドに付与する。

上記属性のパラメータとして、表 3 のように、タイムアウト時間や例外スロー有無の設定を行うことができる。EachInvocationTimeout と TotalTimeout の両方が設定されていた場合、先にどちらかのタイムアウト時間に達した時点でタイムアウトになる。

表 3 TimeoutCallHandler 属性、CancelableCallHandler 属性で指定可能なパラメータ

項番	パラメータ名	利用方法
1	Name	FW 構成ファイル(TerasolunaFramework.config)で、CallHandler の DI 設定をした際の name 属性の値(後述)
2	ThrowOnSpecialError	true の場合、タイムアウト(またはキャンセル)発生時に例外をスローする。false の場合はスローしない。 デフォルトは、true。
3	EachInvocationTimeout	非同期処理対象メソッドの実行ごとの経過時間によるタイムアウト時間(ミリ秒)。呼び出されるために計測される。 デフォルトは、無制限 (= TimeoutCallHandlerAttribute.IgnoreTimeout)。
4	TotalTimeout	「スレッドコンテキスト」生成時からの総経過時間によるタイムアウト時間(ミリ秒)。 デフォルトは、無制限 (= TimeoutCallHandlerAttribute.IgnoreTimeout)。

以下に、メソッドの記述例を示す。VirtualMethodInterceptor を使用するため、メソッドには virtual 修飾子を必ず付与すること。

```
/// タイムアウト可能な処理として非同期実行する場合は、
/// TimeoutCallHandler属性を付与する
/// VirtualMethodInterceptorを利用しているためvirtual修飾子を付与
/// 5秒でタイムアウトするように設定
[TimeoutCallHandler(EachInvocationTimeout = 5000)]
public virtual string Execute01()
{
・・・
}
```

リスト 4 非同期実行対象のメソッドの記述例

また、タイムアウト時間や例外スローの設定などを属性のパラメータで指定する代わりに、FW 構成ファイル(TerasolunaFramework.config)に記述することが可能である。この時、構成ファイルの type タグの name 属性に指定する名前は、上記 CallHandler 属性の Name パラメータの値と一致させる。なお、name 属性がない場合はデフォルト(CallHanalder 属性の Name パラメータ指定なし)に対して適用される。
以下に、FW 構成ファイルと TimeoutCallHandler 属性の記述例を示す。
FW 構成ファイルに定義する TimeoutCallHandler.EachInvocationTimeout プロパティは Nullable<System.Int32>(int?)型で、TimeoutCallHandler.ThrowOnSpecialError プロパティは Nullable<System.Boolea>(bool?)型なので、propertyType 属性の記述に注意すること。

```
<unity>
  <typeAliases>
    <typeAlias alias="int" type="System.Int32" />
    <typeAlias alias="bool" type="System.Boolean" />
  </typeAliases>
        <type name="TimeoutCallHandler"
                type="Terasoluna.Threading.TimeoutCallHandler, Terasoluna">
          <lifetime type="singleton" />
          <typeConfig>
            <!-- タイムアウト時間(Nullable<System.Int32>型) -->
            <property name="EachInvocationTimeout"
                        propertyType="System.Nullable`1[System.Int32]">
              <value value="5000" type="int"/>
            </property>
            <!-- タイムアウト時の例外をスローするかどうか(Nullable<System.Boolean>型) -->
            <property name="ThrowOnSpecialError"
                        propertyType="System.Nullable`1[System.Boolean]">
              <value value="false" type="bool"/>
            </property>
          </typeConfig>
        </type>
      </types>
    </container>
  </containers>
```

リスト 5 FW 構成ファイル(TerasolunaFramework.config)の記述例

```
[TimeoutCallHandler(Name = "TimeoutCallHandler")]
public virtual string Execute01()
{
```

構成ファイルで指定した name の値を設定

リスト 6 構成ファイルで DI 設定した TimeoutCallHandler 属性の設定例

以下に、非同期実行対象メソッドの記述例を示す。

```
/// <summary>
/// 非同期処理対象のクラス
/// Unityのインターセプタ機能を利用するため、SetInterceptor属性を付与する必要がある
/// </summary>
[SetInterceptor(typeof(VirtualMethodInterceptor))]
public class TargetClass
{
    /// タイムアウト可能な非同期処理として実行する場合は、
    /// TimeoutCallHandler属性を付与する
    /// VirtualMethdoInterceptorを利用しているためvirtual修飾子を付与
    /// 5秒でタイムアウトするように設定
    [TimeoutCallHandler(EachInvocationTimeout = 5000)]
    public virtual string Execute01()
    {
        Thread.CurrentThread.Name = "タスクスレッド";
        Console.WriteLine("[{0}][{1}]Execute01メソッド開始",
                          Thread.CurrentThread.Name, DateTime.Now);
        ///タイムアウトするように10秒程度の長い処理をシミュレート
        Thread.Sleep(10000);
        ///タイムアウトしても非同期処理は続行するので出力される
        Console.WriteLine("[{0}][{1}]Execute01メソッド終了",
                          Thread.CurrentThread.Name, DateTime.Now);
        return "Execute01メソッドの処理結果です";
    }
    /// キャンセル可能な非同期処理として実行する場合は、
    /// CancelableCallHandler属性を付与する
    /// VirtualMethdoInterceptorを利用しているためvirtual修飾子を付与
    [CancelableCallHandler]
    public virtual string Execute02()
    {
        Thread.CurrentThread.Name = "タスクスレッド";
        Console.WriteLine("[{0}][{1}]Execute02メソッド開始",
                    Thread.CurrentThread.Name, DateTime.Now);
        ///キャンセル処理が走るように10秒程度の長い処理をシミュレート
        Thread.Sleep(10000);
        ///キャンセルしても非同期処理は続行するので出力される
        Console.WriteLine("[{0}][{1}]Execute02メソッド終了",
                    Thread.CurrentThread.Name, DateTime.Now);
        return "Execute02メソッドの処理結果です";
    }
}
```

リスト 7 非同期処理対象クラスの記述例

(2) 非同期実行の呼び出し処理の実装

透過的な非同期実行を利用する場合、InvocationScope を使って using 句で囲みスコープを定義する。スコープにセットする「スレッドコンテキスト」は、用途に合わせて以下のコンテキストを選択する。コンテキストは1つの InvocationScope に対して複数指定可能である。

表 4「スレッドコンテキスト」クラス

項番	用途	クラス	利用方法
1	タイムアウト対応スレッドコンテキスト	TimeoutInvocationContext	経過時間の取得やタイムアウト時間の設定などタイムアウト固有の機能を提供する。
2	キャンセル対応スレッドコンテキスト	CancelableInvocationContext	タイムアウトの機能に加え、キャンセル要求、キャンセル状態の取得といったキャンセル固有の機能を提供する。

スコープへ「スレッドコンテキスト」をセットするには、InvocationScope.SetupContext メソッドを利用する。また、タスクをキャンセルするには、CancelableInvocationContext.Cancel メソッドを利用する。なお、キャンセル指示をする際、呼び出し元のスレッドは非同期処理が終了するまで待機状態に入っているので、別のスレッド(通常はメインスレッド)により Cancel メソッドを実行する。

```
///呼び出し元スレッドでスレッドコンテキストを作成
CancelableInvocationContext cancelableContext =
                                scope.SetupContext<CancelableInvocationContext>();
・・・
///キャンセル用スレッド（メインスレッド）でのキャンセル実施
cancelableContext.Cancel();
```

リスト 8 キャンセル処理の実行例

以下に、非同期実行対象メソッドの呼び出し処理の記述例を以下に示す。

```
    public class InvocationSample2
    {
        /// <summary>
        /// タイムアウト可能なタスクの非同期実行例
        /// </summary>
        public static void TimeoutThreadTest()
        {
            ///非同期実行対象クラスをUnityContainerでAOP可能なインスタンスとして生成
            IUnityContainer container = UnityManager.GetFrameworkContainer();
            TargetClass target = container.Resolve<TargetClass>();

            /// スレッドが利用するInvocationScopeを生成
            using (InvocationScope scope = new InvocationScope("top"))
            {
                /// 実行結果出力のためスレッド名を設定しておく
                Thread.CurrentThread.Name = "呼び出し元スレッド";
                /// スレッドコンテキストを生成しスコープにセット
                scope.SetupContext<TimeoutInvocationContext>();
                try
                {
                    Console.WriteLine("[{0}][{1}]非同期処理呼び出し開始",
                            Thread.CurrentThread.Name, DateTime.Now);
                    ///対象のメソッドを透過的に非同期実行
                    string result = target.Execute01();
                    ///処理結果を表示
                    ///この例では、その前にタイムアウトになるので到達しない
                    Console.WriteLine(result);
                }
                catch (TimeoutException e)
                {
                    ///タイムアウトにより例外発生
                    Console.WriteLine("[{0}][{1}]タイムアウトしました",
                            Thread.CurrentThread.Name, DateTime.Now);
                }
            }
        }
・・・
```

リスト 9 タイムアウト可能な処理の透過的な非同期実行の例

```
    public class InvocationSample2
    {
・・・
        /// キャンセル用のスレッドコンテキスト
        private static CancelableInvocationContext cancelableContext;

        /// <summary>
        /// キャンセル可能なタスクの非同期実行例
        /// </summary>
        public static void CancelThreadTest()
        {
            Thread.CurrentThread.Name = "メインスレッド";
            /// メインスレッドが待機しないようにタスクの呼び出し処理自体も非同期実行
            Action action = new Action(DoExecute);
            action.BeginInvoke(null, null);
            Thread.Sleep(3000); // タスク処理がある程度実行されるまでsleep
            Console.WriteLine("[{0}][{1}]キャンセル実行",
                                                Thread.CurrentThread.Name, DateTime.Now);
            /// キャンセル指示
            cancelableContext.Cancel();
        }

        /// <summary>
        /// タスクの呼び出し処理
        /// </summary>
        private static void DoExecute()
        {
            Thread.CurrentThread.Name = "呼び出し用スレッド";
            ///非同期実行対象クラスをUnityContainerでAOP可能なインスタンスとして生成
            IUnityContainer container = UnityManager.GetFrameworkContainer();
            UnityContainerUtility.SetupSingleConfigure<SetInterceptorExtension>(container);
            TargetClass target = container.Resolve<TargetClass>();
            /// InvocationScopeを生成
            using (InvocationScope scope = new InvocationScope("top"))
            {
                /// スレッドコンテキストを生成しスコープにセット
                cancelableContext = scope.SetupContext<CancelableInvocationContext>();
                try
                {
                    Console.WriteLine("[{0}][{1}]非同期処理呼び出し開始",
                                                Thread.CurrentThread.Name, DateTime.Now);
                    ///対象のメソッドを透過的に非同期処理実行
                    string result = target.Execute02();
                    ///この例では、キャンセル処理が実行されるので以下のコードには到達しない
                    Console.WriteLine(result);
                }
                catch (InvocationCancelException)
                {
                    //キャンセル時に例外発生
                    Console.WriteLine("[{0}][{1}]キャンセルしました",
                                                Thread.CurrentThread.Name, DateTime.Now);
                }
            }
        }
    }
```

リスト 10 キャンセル可能な処理の透過的な非同期実行の例

(3) 実行結果

- TimeoutThreadTest メソッド（タイムアウト可能なタスクの非同期実行例）の実行結果

TimeoutCallHandler で指定した時間(5 秒)経過後にタイムアウトし、呼び出し元スレッドに制御が戻ってきているのが分かる。
注意点として、タイムアウトして呼び出し元スレッドの制御が戻ってきても、タスクスレッドは処理を継続している。このため、タスクスレッドで「Excecute01 メソッド終了」が出力されている。

```
[呼び出し元スレッド][2009/10/06 17:29:19]非同期処理呼び出し開始
[タスクスレッド][2009/10/06 17:29:19]Execute01 メソッド開始
[呼び出し元スレッド][2009/10/06 17:29:24]タイムアウトしました
[タスクスレッド][2009/10/06 17:29:29]Execute01 メソッド終了
```

リスト 11　TimeoutThreadTest メソッドの実行結果

- CancelThreadTest メソッド（キャンセル可能なタスク非同期実行例）の実行結果

メインスレッドによってキャンセル指示された直後に、呼び出し元スレッドに制御が戻ってきているのが分かる。
注意点として、簡易タイムアウト機能と同様、キャンセルされて呼び出し元スレッドの制御が戻ってきても、タスクスレッドは処理継続している。このため、タスクスレッドで「Excecute02 メソッド終了」が出力されている。

```
[呼び出し元スレッド][2009/10/06 17:41:38]非同期処理呼び出し開始
[タスクスレッド][2009/10/06 17:41:38]Execute02 メソッド開始
[メインスレッド][2009/10/06 17:41:41]キャンセル実行
[呼び出し元スレッド][2009/10/06 17:41:41]キャンセルしました
[タスクスレッド][2009/10/06 17:41:48]Execute02 メソッド終了
```

リスト 12　CancelThreadTest の実行結果

◆ タイムアウトフラグおよびキャンセルフラグのチェック

タイムアウト時やキャンセル指示後も、非同期処理タスクは継続処理するため、無限ループを扱うタスクやキャンセル時に処理結果が反映されないように制御が必要なケースでは、タイムアウトフラグやキャンセルフラグをチェックし中断する実装が必要になる。
タイムアウトフラグをチェックする場合には、InvocationScope.Current プロパティから現在の InvocationScope オブジェクトを取得し、InvocationScope.GetContext メソッドを使って、TimeoutInvocatinContext オブジェクトを取得する。
取得した TimeoutInvocationContext オブジェクトに対して PerCallContexts プロパティに含まれる TimeoutInvocationPerCallContext オブジェクトを取得し、TimeoutInvocationPerCallContext.TimedOut プロパティが true であれば、タイムアウトしたと判断し、タスクを中断するように実装する。
このとき、実行中の非同期タスクごとにタイムアウトかどうか管理するため、TimeoutInvocationPerCallContext オブジェクトにタイムアウトフラグが入っている。1つの

TimeoutInvocationContext オブジェクトに対して、複数のタスクスレッドを同時に実行すると、TimeoutInvocationPerCallContext が同時に複数存在してしまうため、どれが自分のスレッドに対応する TimeoutInvocationPerCallContext オブジェクトなのかが判断できなくなってしまうこのため、タイムアウトフラグのチェックを実施する場合は、1つの TimeoutInvocationContext に対して同時に非同期実行できるタスクがを1つとなるよう制御する必要がある。なお、EventProcessWorker においては非同期実行を複数起動することができないため、問題になることは通常ない。以下に、タイムアウトフラグのチェックを行う、非同期処理対象クラスの実装例を示す。

```
/// 無限ループの処理ロジッククラスの例
[SetDefaultInterceptor(typeof(VirtualMethodInterceptor))]
public class InfiniteLoopTask
{
    /// タイムアウトしたかどうかのチェック
    private bool TimedOut
    {
        get
        {
            //TimeoutInvocationPerCallContextの取得
            IEnumerator<InvocationPerCallContext> perCallContexts =
              InvocationScope.Current.GetContext<TimeoutInvocationContext>()
                                .PerCallContexts.GetEnumerator();
            //TimeoutInvocationContextに対して
            //実行中の非同期タスクは１つしかない前提の実装
            //（つまり、TimeoutInvocationPerCallContextは１つしかない）
            perCallContexts.MoveNext();
            TimeoutInvocationPerCallContext currentPerCallContext
                = perCallContexts.Current as TimeoutInvocationPerCallContext;
            return currentPerCallContext.TimedOut;
        }
    }

    //タイムアウトフラグのチェックの実装例
    [TimeoutCallHandler(EachInvocationTimeout = 5000)]
    public virtual string DoExecute01()
    {
        Thread.CurrentThread.Name = "タスクスレッド";
        Console.WriteLine("[{0}][{1}]DoExecute01メソッド開始",
                    Thread.CurrentThread.Name, DateTime.Now);
        ///タイムアウトフラグのチェック
        while (!TimedOut)
        {
            Thread.Sleep(1000);
            Console.WriteLine("無限ループの処理");
        }
        ///タイムアウトしても非同期処理は続行するので出力される
        Console.WriteLine("[{0}][{1}]DoExecute01メソッド終了",
                    Thread.CurrentThread.Name, DateTime.Now);
        return "DoExecute01メソッドの処理結果です";
    }
```

リスト 13 タイムアウトフラグのチェックの例

また、キャンセルフラグをチェックする場合には、InvocationScope.Current プロパティから現在の InvocationScope オブジェクトを取得し、InvocationScope.GetContext メソッドを使って、CancelableInvocatinContext オブジェクトを取得する。
取得した CancelableInvocationContext オブジェクトに対して、CancellationPending プロパティが true かどうかをチェックして、true であればキャンセル指示されたと判断し、タスクを中断するように実装する。
以下に、キャンセルフラグのチェックを行う、非同期処理対象クラスの実装例を示す。
この例では、無限ループによる処理を実施しており、CancellationPending プロパティをチェックしてキャンセル指示の有無を定期的にチェックし、CancellationPending が true になったら、ループを抜けるようになっている。
なお、キャンセルフラグの場合、その時点で実行中の全てのスレッドを停止することになるため、上述したタイムアウトフラグのような問題が発生しない。なので、同時に複数のタスクスレッドを実行することも可能である(この場合、全てのスレッドがキャンセル指示により停止される)。

```
/// 無限ループの処理ロジッククラスの例
[SetDefaultInterceptor(typeof(VirtualMethodInterceptor))]
public class InfiniteLoopTask
{
    /// <summary>
    /// キャンセル可能なタスク
    /// </summary>
    [CancelableCallHandler]
    public virtual void DoExecute02()
    {
        Thread.CurrentThread.Name = "タスクスレッド";
        Console.WriteLine("[{0}][{1}]DoExecute02メソッド開始",
                                        Thread.CurrentThread.Name, DateTime.Now);

        //現在有効なコンテキストを取得
        CancelableInvocationContext context =
            InvocationScope.Current.GetContext<CancelableInvocationContext>();

        /// CancellationPendingをチェックしキャンセルされたかチェック
        /// キャンセル指示が実施されるとループを抜ける
        while (!context.CancellationPending)
        {
            /// 無限ループの処理
            Thread.Sleep(1000);
            Console.WriteLine("無限ループの処理");
        }

        Console.WriteLine("[{0}][{1}]DoExecute02メソッド終了",
                                        Thread.CurrentThread.Name, DateTime.Now);
    }
}
```

リスト 14　キャンセルフラグのチェックの例

◆ 非 UI スレッドによる透過的な UI スレッド実行の実装

非 UI スレッドによる透過的な UI スレッドの実行方法は、タスクの透過的な非同期実行と同じ実装スタイルを実現している。なお、現在実行中のスレッドが UI スレッドであった場合には通常の同期的なメソッド呼び出しとして実行するため、条件によって実行されるスレッドが異なる場合も、現在実行中のスレッドが UI スレッドか非 UI スレッドかを判断せずに実装できる。

UI スレッドの実行対象メソッドには、UICallHandlerAttribute を付与する。
このとき VirtulaMethodInterceptor を使用するため virtual 修飾子を必ず付与すること。

表 5 UI スレッド実行対象メソッドに付与する属性

項番	属性	利用方法
1	UICallHandlerAttribute	UI スレッドから処理したい画面更新処理を実装したメソッドに左記属性を付与する。

また、「スレッドコンテキスト」として UIInvocationContext クラスを利用する。

表 6 UI スレッド実行用の「スレッドコンテキスト」クラス

項番	用途	クラス	利用方法
1	UI スレッド実行用「スレッドコンテキスト」	UIInvocationContext	UI スレッドとの透過的な読み書きを可能にする。

開発者は、InvocationScope.SetupContext メソッドにより UIInvocationContext を作成し、UIInvocationContext.TargetForm プロパティに、更新対象の画面を指定する。

以下に、透過的な UI スレッド実行の実装例[2]を示す。

```
    /// <summary>
    /// 足し算を行うイベント処理クラス
    /// </summary>
    [SetInterceptor(typeof(VirtualMethodInterceptor))]
    public class AddEventProcess
    {
・・・
        /// UIスレッド以外のスレッドから非同期実行されるメソッド
        private void DoEvent(int num1, int num2)
        {
            ・・・
                int answer = 0;
                // answerに計算結果を格納する処理を実施
                ・・・
                using (InvocationScope scope2 = new InvocationScope("ui"))
                {
                    ///現在のスコープにUI処理実行用のスレッドコンテキストをセット
                    UIInvocationContext uiInvocationContext
                            = scope2.SetupContext<UIInvocationContext>();
                    ///スレッドコンテキストにTargetFormプロパティに現在の画面を設定
                    uiInvocationContext.TargetForm = TargetForm;
                    ///UIスレッド以外からUIコントロールの操作を実施
                    HandleCompleted(new CompletedEventArgs(answer));
                }

        }
        ///UIスレッド以外のスレッドから処理したいUI操作の実行
        [UICallHandler]
        protected virtual void HandleCompleted(CompletedEventArgs args)
        {
            EventHandler<CompletedEventArgs> handler = Completed;
            if (handler != null)
            {
                handler(this, args);
            }
            running = false;
        }
    }
```

リスト 15 透過的な UI スレッド実行の記述例

[2] 実装例の全体は、「スレッド制御機能を利用した Windows アプリケーションの例」のサンプルコードを参照。

◆ FW 構成ファイル(TerasolunaFrameowork.config)の設定

本機能を利用するためには、FW 構成ファイル(TerasolunaFramework.config)の /configuration/unity/containers/container/extensions/add タグで、以下の UnityContainerExtension 継承クラスを記述する。

- Microsoft.Practices.Unity.InterceptionExtension.Interception クラス
  - 「CM-02 インスタンス管理機能」の AOP 機能を利用するため必要な Extension
- Terasoluna.Unity.SetInterceptorExtension クラス
  - 「CM-02 インスタンス管理機能」の AOP 機能を利用するため必要な Extension

以下に、TerasolunaFramework.config の記述例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <unity>
    <containers>
      <container>
       <extensions>
          <!-- AOPの設定 -->
          <add type="Microsoft.Practices.Unity.InterceptionExtension.Interception,
                        Microsoft.Practices.Unity.Interception,
                        Version=1.2.0.0,
                        Culture=neutral,
                        PublicKeyToken=31bf3856ad364e35" />
          <add type="Terasoluna.Unity.SetInterceptorExtension,
                      Terasoluna" />
        </extensions>
      </container>
    </containers>
  </unity>
</configuration>
```

リスト 16 TerasolunaFramework.config の記述例

◆ スレッド制御機能を利用した Windows Forms アプリケーション

サーバとのファイル送受信や大量データ更新などの時間のかかる処理を UI スレッドで同期的に実行する場合、ボタン押下後、処理が完了するまでの間、画面がフリーズしてしまう。
「CL-03 イベント処理実行機能」は、本機能（スレッド制御機能）を利用してこのようなケースに対応している。複雑なマルチスレッド制御は隠ぺいされており、非同期実行処理を実現する Windows Forms アプリケーション容易に実装することができる。
以下では、「CL-03 イベント処理実行機能」の動作原理をサンプルコードを使って説明する。なお、これはアーキテクトが内部のアーキテクチャを深く理解するためのサンプルコードであり、「CL-03 イベント処理実行機能」を利用すれば業務開発者がこのような実装を意識する必要はない。

このサンプルプログラムでは、画面よりテキストボックスに2つの数字を入力し、「計算」ボタンを押下すると、3秒 sleep 後、入力した数字を足し算した結果を画面に反映するという重い計算処理をシミュレートしたものである。
このようなケースでは、ユーザの操作性を考慮し UI スレッド（メインスレッド）から計算処理を別スレッドで呼び出す。このとき、計算を実施するタスクスレッド（下図の「計算処理タスクスレッド」）のほかに、タスクスレッドを呼び出すための呼び出し元スレッド（下図の「計算処理呼び出しスレッド」）を用意する。
こうすることで、UI スレッドがタスクスレッドの処理結果を取得するまで待機する必要がなく、その間自由に画面を動かすことが可能となる。また、ユーザが処理をキャンセルしたい場合には、「キャンセル」ボタン押下により、UI スレッドでキャンセル処理を実施し、呼び出し元スレッド（計算処理呼び出しスレッド）を待機状態から復帰させることで、計算処理タスクスレッドの完了状態によらず、処理を終了させることができる。

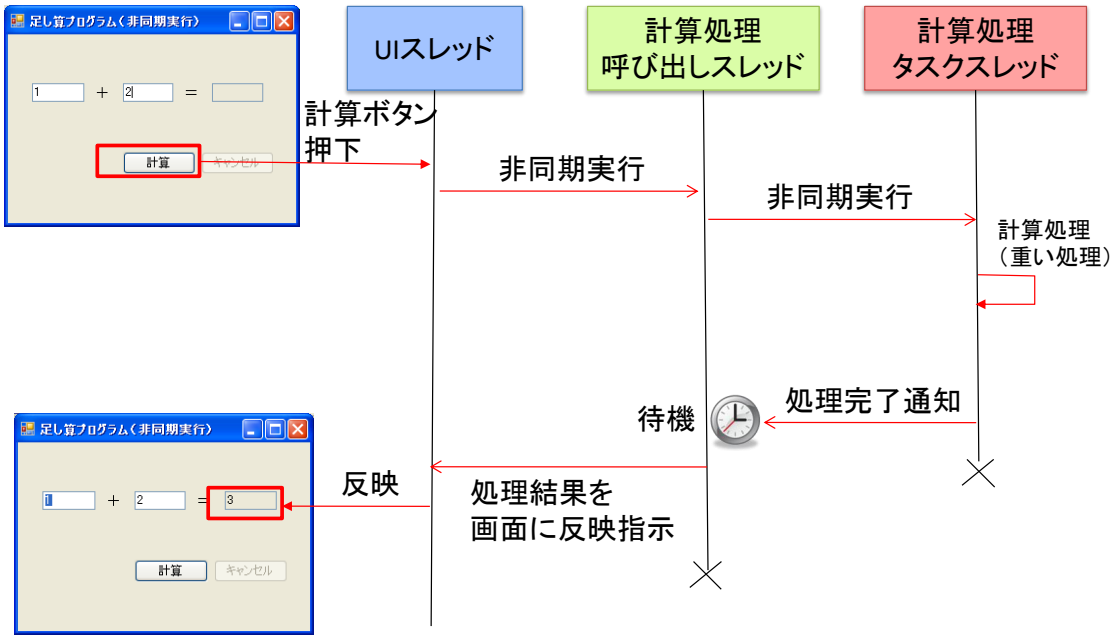

図 14 サンプルプログラムの動作イメージ（正常時）

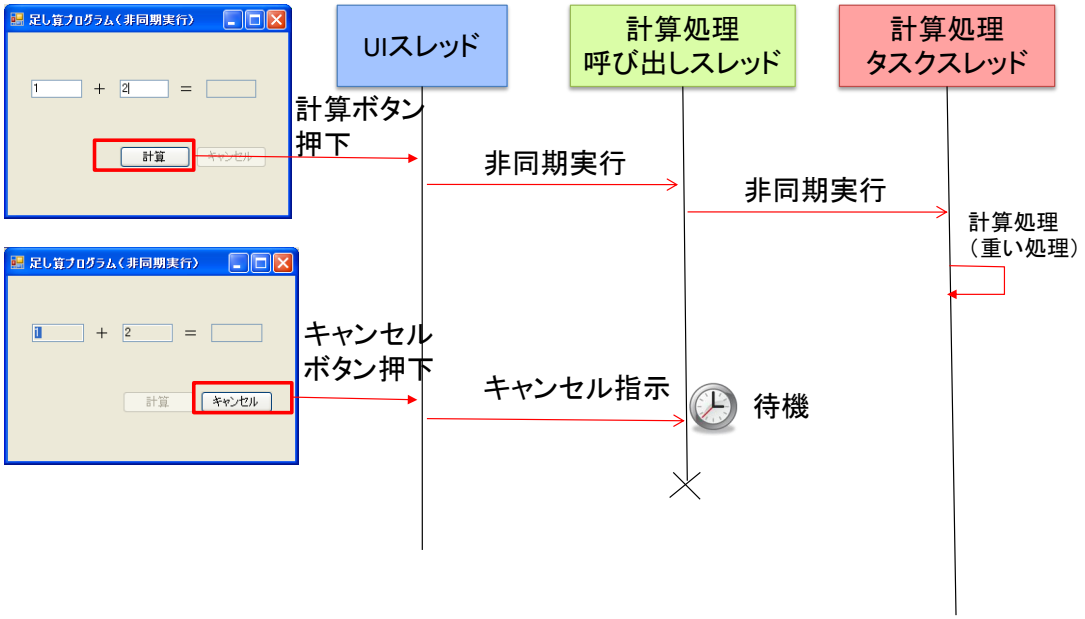

図 15 サンプルプログラムの動作イメージ(キャンセル時)

サンプルプログラムを以下に示す。

なお、画面クラス(CalcForm2)から呼び出している足し算処理のクラス(AddEventProccess)は、「CL-03 イベント処理実行機能」の仕組みを簡易にしたサンプルになっている。

```
public partial class CalcForm2 : Form
{
    /// 足し算を行うイベント処理
    private AddEventProcess eventProcess;

    public CalcForm2()
    {
        InitializeComponent();
        /// 足し算イベント処理の作成
        IUnityContainer container = UnityManager.GetFrameworkContainer();
        eventProcess = container.Resolve<AddEventProcess>();
    }

    /// <summary>
    /// 計算ボタン押下
    /// </summary>
    private void calcButton_Click(object sender, EventArgs e)
    {
        ///入力データの取得
        int num1 = int.Parse(num1TextBox.Text);
        int num2 = int.Parse(num2TextBox.Text);
        ///イベント処理の実行
        eventProcess.TargetForm = this;
        eventProcess.Completed += new
                    EventHandler<CompletedEventArgs>(eventProcess_Completed);
        eventProcess.RunAsync(num1, num2);
    }

    /// <summary>
    /// キャンセルボタン押下
    /// </summary>
    private void cancelButton_Click(object sender, EventArgs e)
    {
        ///計算実行実施キャンセル
        eventProcess.Cancel();
    }

    /// <summary>
    /// 足し算処理の完了時のイベント
    /// </summary>
    private void eventProcess_Completed(object sender, CompletedEventArgs e)
    {
        ///UIコントロールの操作
        answerTextBox.Text = e.Answer.ToString();
    }
}
```

リスト 17 スレッド制御機能を組み合わせた Form の実装例

```
    /// <summary>
    /// 足し算を行うイベント処理クラス
    /// フレームワークが提供するイベント処理機能を簡易にしたサンプル
    /// </summary>
    [SetInterceptor(typeof(VirtualMethodInterceptor))]
    public class AddEventProcess
    {
        /// キャンセル用スレッドコンテキスト
        private CancelableInvocationContext cancelableInvocationContext;
        /// タスクスレッドの実行中かどうかを判定するフラグ
        private bool running = false;

        /// 現在の画面
        public Form TargetForm { get; set; }

        /// 処理完了後のイベント
        public event EventHandler<CompletedEventArgs> Completed;

        /// <summary>
        /// タスクの非同期実行
        /// </summary>
        public void RunAsync(int num1, int num2)
        {
            if (running)
            {
                MessageBox.Show("計算処理がすでに実行中です");
                return;
            }
            running = true;
            ///メインスレッド(UIスレッド）に制御が戻り
            ///画面を自由に動かしたり、キャンセルボタンを押下できるように
            ///タスクの呼び出し自体も非同期実行
            Action<int, int> action = new Action<int, int>(DoEvent);
            action.BeginInvoke(num1, num2, null, null);
        }

        /// <summary>
        /// キャンセル
        /// </summary>
        public void Cancel()
        {
            if (cancelableInvocationContext != null)
            {
                cancelableInvocationContext.Cancel();
            }
        }
・・・
```

```
・・・
        /// 非同期で計算実行指示し、画面に反映
        private void DoEvent(int num1, int num2)
        {
            using (InvocationScope scope = new InvocationScope("task"))
            {
                ///スレッドコンテキストの生成
                cancelableInvocationContext =
                                        scope.SetupContext<CancelableInvocationContext>();
                int answer = 0;
                try
                {
                    answer = Calcuate(num1, num2);   //タスクの非同期実行
                }
                catch (InvocationCancelException)    //キャンセル時
                {
                    running = false;
                    return;
                }
                using (InvocationScope scope2 = new InvocationScope("ui"))
                {
                    ///現在のスコープにUI処理実行用のスレッドコンテキストをセット
                    UIInvocationContext uiInvocationContext
                            = scope2.SetupContext<UIInvocationContext>();
                    ///スレッドコンテキストにTargetFormプロパティに現在の画面を設定
                    uiInvocationContext.TargetForm = TargetForm;
                    ///UIスレッド以外からUIコントロールの操作を実施
                    HandleCompleted(new CompletedEventArgs(answer));
                }
            }
        }

        /// 重い計算を実行するタスクスレッド
        [CancelableCallHandler]
        protected virtual int Calcuate(int num1, int num2)
        {
            ///3秒待って長い処理をシミュレート
            Thread.Sleep(3000);
            return num1 + num2;
        }

        ///UIスレッド以外のスレッドから処理したいUI操作の実行
        [UICallHandler]
        protected virtual void HandleCompleted(CompletedEventArgs args)
        {
            EventHandler<CompletedEventArgs> handler = Completed;
            if (handler != null)
            {
                handler.Invoke(this, args);
            }
            running = false;
        }
    }
・・・
```

```
・・・・
    /// <summary>
    /// 処理完了時のEventArgs
    /// </summary>
    public class CompletedEventArgs：EventArgs
    {
        public CompletedEventArgs(int answer)
        {
            Answer = answer;
        }
        public int Answer { get; set; }
    }
```

リスト 18 Form が呼び出す非同期タスクの実装例

■ 内部構成

◆ クラス図

タスクの透過的な非同期実行に関するクラス図を図 16、透過的な UI スレッドの実行に関するクラス図を図 17 に示す。

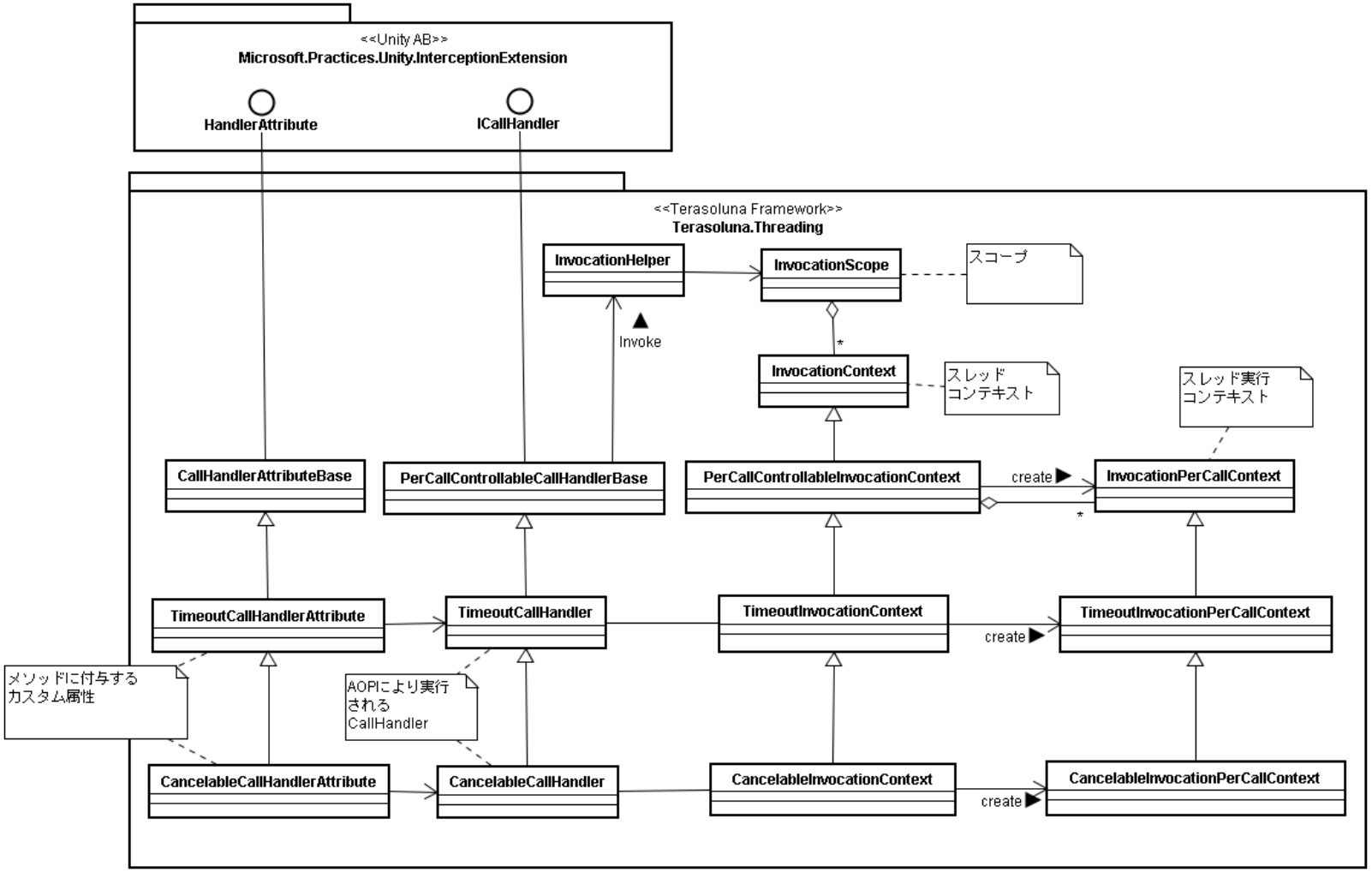

図 16 クラス図(タスクの透過的な非同期実行に関するクラス)

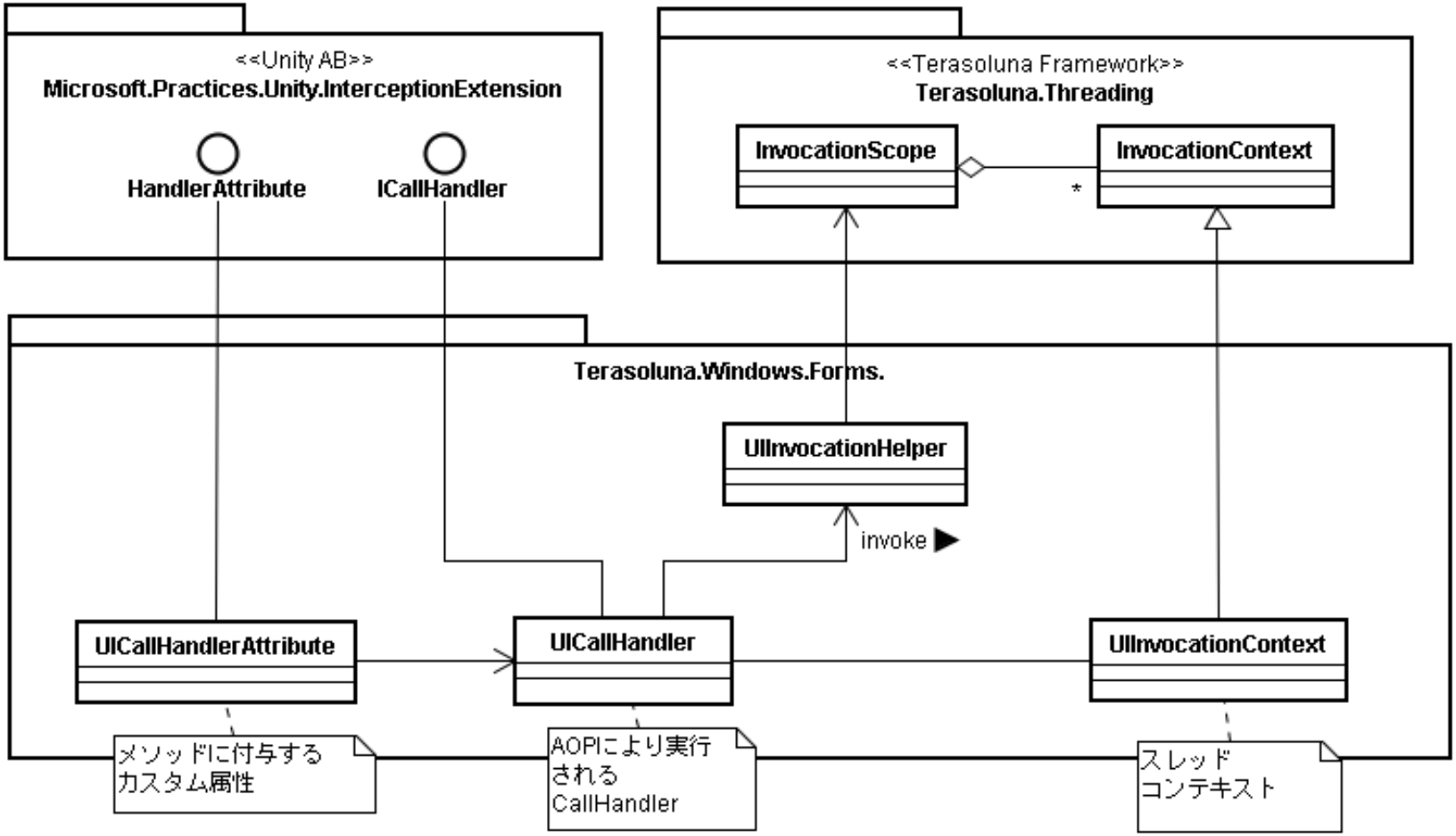

図 17 クラス図(透過的な UI スレッドの実行に関するクラス)

◆ シーケンス図

タスクの透過的な非同期実行に成功した場合のシーケンス図を図 18、キャンセルした場合のシーケンス図を図 19 に示す。
また、透過的な UI スレッド実行時のシーケンス図を図 20 に示す。

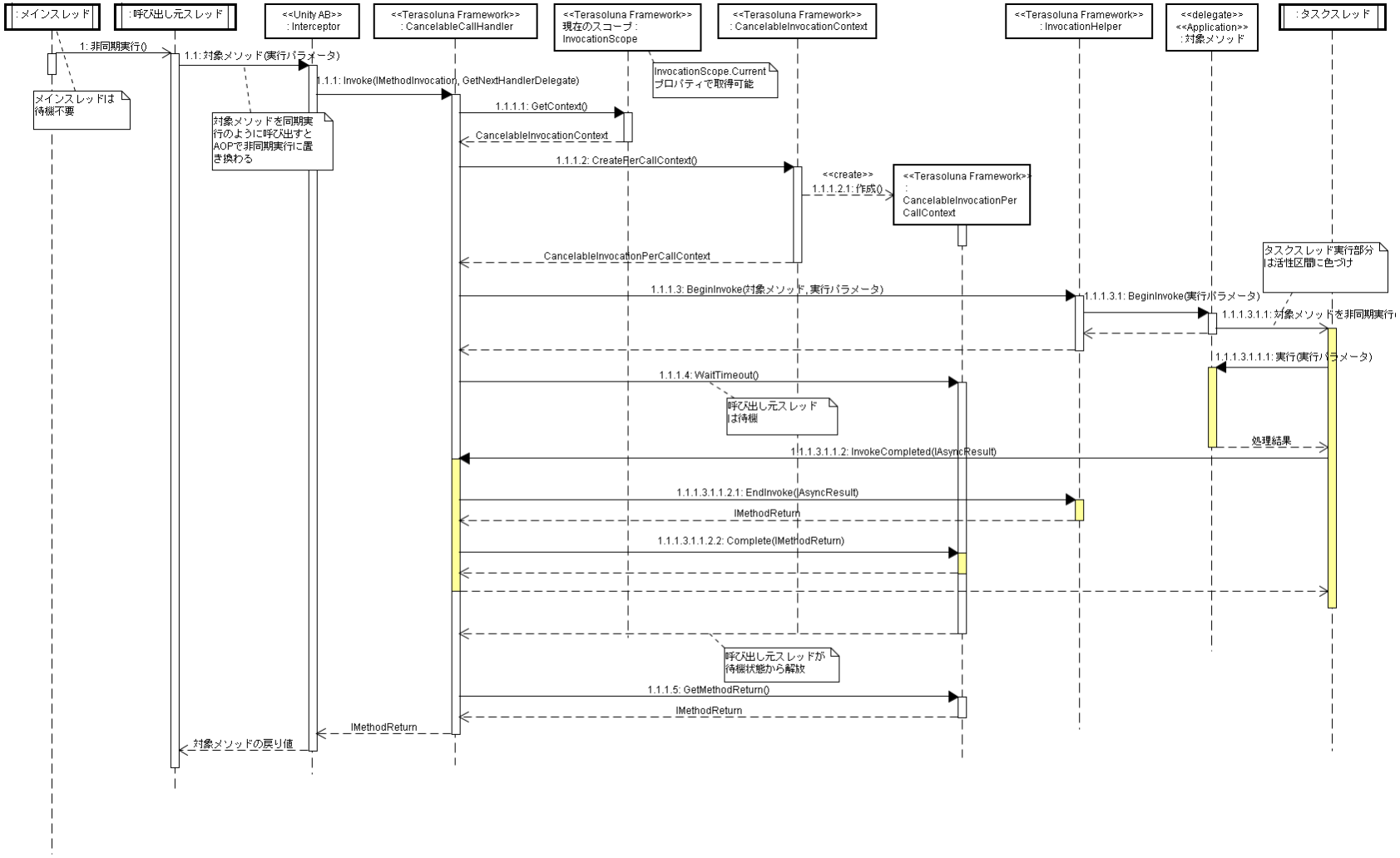

図 18 シーケンス図(タスクの透過的な非同期実行)

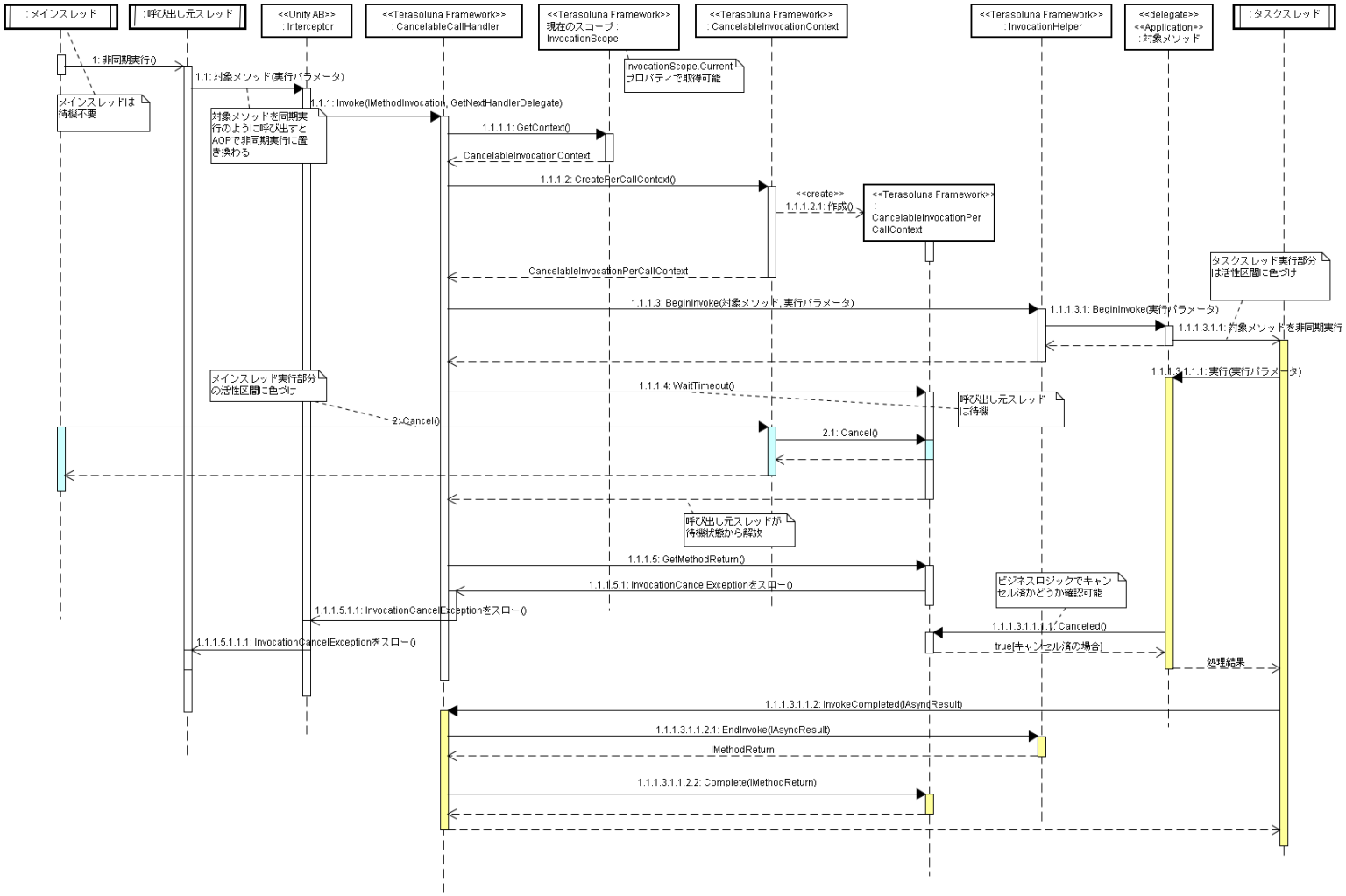

図 19 シーケンス図(タスクの透過的な非同期実行・キャンセル時)

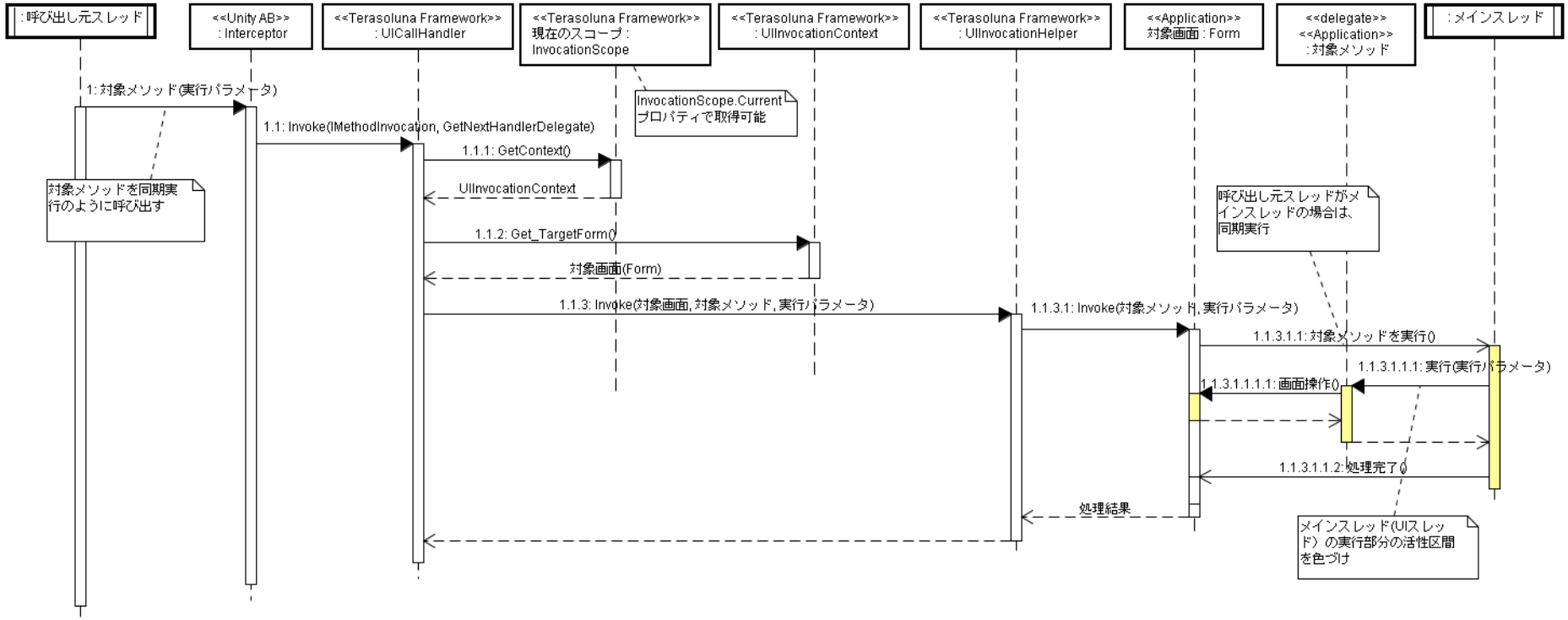

図 20 シーケンス図(透過的な UI スレッド実行)

◆ 構成クラス

表 7 構成クラス一覧

項番	クラス名	説明
Terasoluna.Threading 名前空間		
1	InvocationScope	「スコープ」を表すクラス
2	InvocationHelper	「スコープ」における「スレッドコンテキスト」情報をもとに非同期実行を実施するためのヘルパークラス
3	InvocationContext	「スレッドコンテキスト」を表す基底クラス
4	InvocationPerCallContext	「スレッド実行時コンテキスト」を表す基底クラス
5	InvocationAsyncCallback	非同期実行時のコールバックに関する情報を保持するクラス
6	InvocationAsyncState	非同期実行時の引き継ぎ情報を保持するクラス
7	InvocationCancelException	キャンセル実行時に発生する例外
8	CallHandlerBase	ICallHandler インタフェースを実装する基底クラス
9	CallHandlerAttributeBase	HandlerAttribute 継承する基底クラス
10	PerCallControllableInvocationContext	メソッドの呼び出しを制御する InvocationContext 継承クラス
11	PerCallControllableCallHandlerAttributeBase	メソッドの呼び出しを制御する ICallHandler 実装クラス
12	TimeoutCallHandler	タイムアウト可能なメソッドの非同期呼び出しを制御する ICallHandler 実装クラス
13	TimeoutCallHandlerAttribute	タイムアウト可能なメソッドの非同期呼び出しを制御する HandlerAttribute 継承クラス
14	TimeoutInvocationContext	タイムアウト可能なメソッドの非同期呼び出しを制御する「スレッドコンテキスト」クラス
15	TimeoutInvocationPerCallContext	タイムアウト可能なメソッドの非同期呼び出しを制御する「スレッド実行時コンテキスト」クラス

16	CancelableCallHandler	キャンセル可能なメソッドの非同期呼び出しを制御する ICallHandler 実装クラス
17	CancelableCallHandlerAttribute	キャンセル可能なメソッドの非同期呼び出しを制御する HandlerAttribute 継承クラス
18	CancelableInvocationContext	キャンセル可能なメソッドの非同期呼び出しを制御する「スレッドコンテキスト」クラス
19	CancelableInvocationPerCallContext	タイムアウト可能なメソッドの非同期呼び出しを制御する「スレッド実行時コンテキスト」クラス
Terasoluna.Windows.Forms.Threading 名前空間		
20	UIInvocationHelper	「スコープ」における「スレッドコンテキスト」情報をもとに UI スレッド実行を実施するためのヘルパークラス
21	UIThreadInvoker	UI スレッド実行を実施するクラス
22	UICallHandler	UI スレッド実行可能なメソッド呼び出しを制御する ICallHandler 実装クラス
23	UICallHandlerAttribute	UI スレッド実行可能なメソッド呼び出しを制御する HandlerAttribute 継承クラス
24	UIInvocationContext	UI スレッド実行可能なメソッド呼び出しを制御する「スレッドコンテキスト」クラス

■ 関連機能

- CM-02 インスタンス管理機能
- CL-03 イベント処理実行機能