Try-Catch のよくある質問

Version 2023.1
2024-01-02

InterSystems Corporation　1 Memorial Drive　Cambridge MA 02142　www.intersystems.com

Try-Catch のよくある質問
InterSystems IRIS Data Platform　Version 2023.1　2024-01-02

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

目次

Try-Catch のよくある質問 1
一般的な質問 1
Try-Catch と以前のエラー処理メカニズム 5

Try-Catch のよくある質問

一般的な質問

Try-Catch とは何ですか。

Try-Catch は、ObjectScript の言語構造で、例外と呼ばれる例外的な条件の処理をアプリケーションで実行できます。Try は、ペアの Catch ブロックによって処理される例外のためのコードのブロックを定義します。例外には、言語でエラーが発生したときに暗黙的にスローされる <DIVIDE> や <UNDEFINED> などのすべての ObjectScript システム・エラーが含まれます。これらは、アプリケーションで Throw コマンドを使用して明示的にスローできる他のタイプの例外的な条件もカプセル化します。Try ブロックで例外がスローされると、制御が Catch ブロックに移動し、そこで実行が再開されます。

例外は、定義済みの Try ブロックにないコードからスローできます。これが発生すると、スタックの次の例外ハンドラが例外を検出し、必要に応じてスタックを巻き戻します。例外を検出する例外ハンドラは Catch にできますが、代わりに $ZTRAP ハンドラにもできます (詳細は後述)。

例外がスローされると、アプリケーションは、通常の制御フローからそれて、スタック (最深のスタック・レベル) の最初に使用可能な例外ハンドラで実行を再開し、そのスタックが見つかるまで必要に応じてスタックを巻き戻します。Try-Catch を使用すると、通常最初に使用可能な例外ハンドラは Catch ブロックです。例外オブジェクトは Catch ブロックで使用でき、検査して例外に関する情報をリカバリできます。

例外とエラーの違いは何ですか。

“エラー” という用語には複数の意味があるため、このページでは技術用語としてこの用語を使用しません。例外は、%Exception.AbstractException クラスのサブクラスであるオブジェクトです。いくつかのタイプの例外は %Exception.AbstractException のサブクラスとしてモデル化されています。

<DIVIDE> や <UNDEFINED> などの ObjectScript システム・エラーはクラス %Exception.SystemException の例外です。この形式の例外は、このようなエラーが発生すると、システムによって自動的にインスタンス化され、スローされます。アプリケーションによってインスタンス化され Throw コマンドを使用してスローできる他の例外のクラスもあります。%Exception.StatusException は %Status エラーをモデル化する例外クラスで、%Exception.SQL は SQLCODE エラーをモデル化します。これらの例外クラスを拡張して、独自の例外クラスを作成することも可能です。

エラー %Status 値を返すメソッドを呼び出しました。これを例外としてスローするにはどうすればよいでしょうか。

%Exception.StatusException にはメソッド CreateFromStatus があり、Throw コマンドを使用してスローできる例外オブジェクトを作成します。例えば、変数 sc に %Status 値が含まれている場合、以下のコードによってこれが例外としてスローされます。

```
if $$$ISERR(sc) {
  throw ##class(%Exception.StatusException).CreateFromStatus(sc)
}
```

注釈　　この機能は、マクロ $$$ThrowStatus および $$$THROWONERRORからもアクセスできます。

エラー SQLCODE から例外をスローするにはどうすればよいでしょうか。

%Exception.SQL にはメソッド CreateFromSQLCODEがあり、Throw コマンドを使用してスローできる例外オブジェクトを作成します。例えば、変数 SQLCODE に SQLCODE 値が含まれていて %msg にそのメッセージが含まれている場合、以下のコードによってこれが例外としてスローされます。

```
if SQLCODE<0 {
  throw ##class(%Exception.SQL).CreateFromSQLCODE(SQLCODE,%msg)
}
```

Catch ブロック内で例外が発生するとどうなりますか。

Catch ブロック内でスローされる例外は、Try ブロックの外側の任意の場所で発生する例外と同様です。スタックの次に使用可能な例外ハンドラがこれを処理します。例外処理コード内の追加の例外を検出するために、その Catch ブロック自体の中に別の Try-Catch を入れ子にできます。

例外から %Status または SQLCODE に変換できますか。

はい、例外オブジェクトには、この操作を行うメソッド AsStatus と AsSQLCODE があります。

例外を検出した場合、どうすればよいでしょうか。

Catch ブロックは、完全に機能する ObjectScript 環境です。必要なあらゆるコマンドを使用できます。例外を処理するために一般的に実行できることは複数あります。それらについてここで説明します。それらのアクションは、Catch ブロック内に完全に含まれている必要はありません。これらは、必要に応じて Catch ブロックに続くコードで実行できます。

まず、Catch は複数の種類の例外を処理するため、実行することを決定するためにアプリケーションでさまざまな例外を区別する必要があります。$classname 関数または %IsA メソッド (InterSystems IRIS® **%Library.Base** クラスから継承) を使用して、例外オブジェクトのクラスまたはスーパークラスを判別できます。例外オブジェクトの Name プロパティと Code プロパティを検査してエラーのタイプを判別できます。

例外の前に実行された作業を元に戻したり、ロックや他のリソースを解放したり、トランザクションをロール・バックしたりする必要があることも多いです。

LOG^%ETN を呼び出すことによって、上記の操作を標準のアプリケーション・エラー・ログに記録できます。例外が **%Exception.SystemException** ではない場合、LOG^%ETNを呼び出す前に $ZERROR を意味のある値に設定します。この値は、ログ・エントリのエラー・メッセージ・フィールドとして使用されます。(アプリケーション・エラー・ログは、管理ポータルの **[アプリケーションエラーログ]** ページに表示されます。)また、例外のまとめを取得し、例外オブジェクトの DisplayString メソッドを使用してユーザに対して表示できます。

上記をすべて完了したら、通常以下のいずれかを実行します。

- 処理を続行するまたは現在のプロシージャから戻る
- スタックの次の例外ハンドラに例外を再スローする
- 新しい例外をスローする
- プロセスを停止する

以下は、これらの概念の一部を示す例です。

```
func(id) public {
  Try {
    ; Flag indicates if we locked the global
    Set locked=0
    ; If we cannot get the lock, throw a user-created
    ; exception with the information we need
    Lock +^mygbl(id):0 If '$test {
      Throw ##class(Exception.MyException).%New("Unable to
```

```
          lock",$name(^mygbl(id)))
    }
    Set locked=1
    Set sc=$system.OBJ.Compile("MyClass")
    If $$$ISERR(sc) {
      Throw ##class(%Exception.StatusException).CreateFromStatus(sc)
    }
    ; Some further processing which may throw exceptions
    ; ...

    If locked { Lock -^mygbl(id) }
  }

Catch exception {
    ; Release the lock resource before doing anything else
    If locked { Lock -^mygbl(id) }
    ; First determine what sort of exception this is
    If exception.%IsA("%Exception.SystemException") {
      ; Log error in error log
      Do BACK^%ETN
      ; Throw my exception class rather than the system exception
      Throw ##class(Exception.MyException).CreateFromSystemException(exception)
    } ElseIf $classname(exception)="Exception.MyException" {
      ; Ignore this sort of exception and just return to code
      ; after the catch block
    } Else {
      ; We will just throw these to outer error handler
      Throw exception
    }
  }
}
```

外部レベルのプロシージャで Try-Catch を使用していて、このプロシージャが別のプロシージャを呼び出し、その別のプロシージャがさらに別のプロシージャを呼び出します。ある深いスタック・レベルにおいて、例外が発生し、外部レベルの Catch で検出されました。例外が発生したコール・スタックを回復するにはどうすればよいでしょうか。

クラス **%Exception.SystemException** の例外 (<UNDEFINED> ObjectScript システム・エラーなど) の場合、$stack 関数を使用してエラー・スタックを検査できます。他の例外クラスの場合、例外をスローするコードを変更し、例外ハンドラがスタックを回復できるようにする必要があります。

以下の例は、システム例外に対して $stack 関数を使用する方法および他のクラスの例外に対してスタックを検出する 1 つの方法を示します。これは 2 つの部分から構成されます。システム例外と他のタイプの例外の両方に対して、スタック情報で **%Exception.StatusException** を拡張するカスタム例外クラスと、検出した情報のログへの記録と表示の両方を行うルーチン例です。

例外クラスを以下に示します。

```
Class MyException.Status Extends %Exception.StatusException
{

  Property Stack [ MultiDimensional ];

  /// Convert a %Status into an exception
  ClassMethod CreateFromStatus(pSC As %Status)
      As %Exception.AbstractException
  {
    // You could choose to override %OnNew and put this code that
    // captures the stack there instead of here in CreateFromStatus.
    // We put it here because we only need to capture the stack in
    // the outer exception, and it is more simply insulated from
    // future changes in the superclasses.

    // First, call CreateFromStatus in the superclass to instantiate
    // the object and fill in the standard exception information.
    set exc=##super(pSC)

    // Clear $ecode so that $stack() refers to the current stack,
    // not the error stack.
    set $ecode=""

    // Subtract one level because we don't need
    // to see this method itself in the stack.
    set exc.Stack=$stack-1

    for i=1:1:exc.Stack {
```

```
      set exc.Stack(i)=$stack(i)_
          " "_$stack(i,"PLACE")_" "_$stack(i,"MCODE")
    }
    quit exc
  }
}
```

例外情報のログへの記録と表示の両方を行うルーチン例を以下に示します。

```
#include %occInclude
testexc(throwsystemexception) {
  try {
    do sub1($g(throwsystemexception))
  } catch exc {
    if exc.%IsA("%Exception.SystemException") {
      set stack=$stack(-1)
      // For System Exceptions, get the stack from the
      // built-in error stack using $stack().
      for i=1:1:stack {
        set stack(i)=$stack(i)_
            " "_$stack(i,"PLACE")_" "_$stack(i,"MCODE")
      }
    } else {
      if $extract($classname(exc),1,12)="MyException." {
        // Exceptions from package MyException will carry the
        // stack of the exception in the multidimensional
        // Stack property.
        merge stack=exc.Stack
      }
      // Set $ze explicitly because it's needed by BACK^%ETN
      // and only SystemExceptions set it implicitly.
      set $ze=exc.DisplayString()
    }
    do BACK^%ETN
    write !,"Exception occurred: ",exc.DisplayString()
    write !,"  class: ",$classname(exc)
    write !,"  name: ",exc.Name
    write !,"  code: ",exc.Code
    if $data(stack) {
      write !,"  stack:"
      for i=1:1:stack {
        write !,"    ",stack(i)
      }
    }
    write !
  }
}
sub1(throwsystemexception) {
  if throwsystemexception {
    do systemexception
  } else {
    do myexception
  }
}
myexception() PUBLIC {
  set sc=$$$ERROR($$$GeneralError,"this is my status code")
  throw ##class(MyException.Status).CreateFromStatus(sc)
}
systemexception() PUBLIC {
  // get a <DIVIDE> error
  set x=1\0
}
```

テスト・ルーチンの出力を以下に示します。

```
USER>do ^testexc(1)

Exception occurred: <DIVIDE> 18 systemexception+2^testexc
  class: %Exception.SystemException
  name: <DIVIDE>
  code: 18
  stack:
    DO +3^testexc +1     do sub1($g(throwsystemexception))
    DO +40^testexc +1     do systemexception
    DO systemexception+2^testexc +1    set x=1¥0

USER>do ^testexc(0)

Exception occurred: ERROR #5001: this is my status code
  class: MyException.Status
  name: 5001
  code: 5001
```

```
 stack:
   DO +3^testexc +1     do sub1($g(throwsystemexception))
   DO +42^testexc +1     do myexception
   DO myexception+2^testexc +1    throw ##class(MyException.Status).CreateFromStatus(sc)

USER>do ^%ERN


For Date: T  16 Feb 2012   2 Errors

Error: ?L

 1. <DIVIDE>systemexception+2^testexc  at  1:25 pm.  $I=/dev/ttys001  ($X=0  $Y=299)
    $J=8225  $ZA=0  $ZB=$c(13)  $ZS=16384 ($S=16504448)
          set x=1¥0

 2. ERROR #5001: this is my status code  at  1:25 pm.  $I=/dev/ttys001  ($X=0  $Y=310)
    $J=8225  $ZA=0  $ZB=$c(13)  $ZS=16384 ($S=16504336)
```

Try-Catch と以前のエラー処理メカニズム

InterSystems IRIS は、例外を処理する他のメカニズムとして $ZTRAP などをサポートしています。どちらを使用すればよいのでしょうか。

Try-Catch を使用してください。このメカニズムは、以下に示すいくつかの理由により InterSystems IRIS で推奨される例外処理メカニズムです。

1. ほとんどの場合、Try-Catch を使用すると、わかりやすく洗練されたコードを作成でき、アプリケーションの管理が容易になります。
2. アクティビティが成功した場合 (つまり、例外がない場合)、実行時のパフォーマンス・コストがありません。このことにより、パフォーマンスの優位性が得られます。
3. Try-Catch コードは使用しやすいため、エラー発生率が下がります。(例えば、無限ループができてしまう可能性がある $ZTRAP を含む構築を回避できます。)
4. 既存のアプリケーションでは、例外を処理する InterSystems の他のメカニズムを使用してコードをカプセル化することによって、一貫性のある例外処理インタフェースへのパスを提供できます。
5. Try-Catch を使用すると、例外オブジェクトへのアクセスが可能になり、発生した例外のタイプに関係なくスローされた例外に関するすべての情報を復元できるようになります。

Try-Catch および例外は以前の ObjectScript エラー・ハンドラとどのようにやり取りしますか。

例外がスローされ、以前のエラー・ハンドラがスタックで最初に使用可能な例外ハンドラである場合、制御は通常の方法でそのエラー・ハンドラに渡されます。しかし、例外オブジェクトは使用できません。スローされる例外がシステム例外 (%Exception.SystemException) の場合、$ZERROR 値は期待どおりに設定されます。$ZTRAP によって検出される他の例外クラスの場合、$ZERROR 値は <NOCATCH> に設定されます。どちらの場合でも、制御のフローは同じです。

Try ブロックのコードが $ZTRAP を設定するプロシージャ、メソッド、またはサブルーチンを呼び出し、そのプロシージャ内で例外が発生すると、この例外はより深いスタック・レベルで発生するため、$ZTRAP はその例外を検出します。コードが $ZTRAP を使用していて、Try-Catch を使用するプロシージャ、メソッド、またはサブルーチンを呼び出し、Try内で例外が発生すると、Catch がこの例外を検出します (ここでも、この例外がより深いスタック・レベルで発生するため)。要するに、例外処理はより深いスタック・レベルのものを使用します。