記事
Toshihiko Minamoto · 2020年9月30日 14m read

InterSystems IRIS のクラスクエリ

InterSystems IRIS のクラスクエリ

InterSystems IRIS（および Cache、Ensemble、HealthShare）のクラスクエリは、SQL クエリを Object Script のコードから分離する便利なツールです。 このクエリの基本的な機能は、同じ SQL クエリを複数の場所で異なる引数で使用する場合にクエリの本文をクラスクエリとして宣言し、このクエリを名前で呼び出すことでコードの重複を回避できるというものです。
このアプローチは、次のレコードを取得するタスクを開発者が定義するカスタムクエリにも便利です。
興味が湧きましたか？ それではこのまま読み進めてください！

基本クラスクエリ

簡単に言うと、基本クラスクエリは SQL の SELECT クエリを表現できるようにするものです。
基本クラスクエリは SQL オプティマイザとコンパイラによって通常の SQL
クエリと同様に処理されますが、Caché Object Script のコンテキストから実行する際はより便利になります。
また、次のようにクラス定義（メソッドまたはプロパティと同様）でクエリ項目として宣言されます。

- タイプ: %SQLQuery
- SQL クエリのすべての引数を引数リストに含める必要があります。
- クエリタイプ: SELECT
- コロンを使用して各引数にアクセスします（静的SQLと同様）。
- 出力結果の名前とデータタイプに関する情報を含む ROWSPEC
  パラメーターをフィールドの順序と共に定義します。
- （任意）フィールドに ID が含まれている場合、 CONTAINID パラメーターにその列番号を定義します。
  ID を返す必要がない場合は、CONTAINID に 0 を割り当てます。
- （任意）静的 SQL の同様のパラメーターに対応し、SQL 式をコンパイルするタイミングを指定する
  COMPILEMODE パラメーターを定義します。 このパラメーターが
  IMMEDIATE（デフォルト）に設定されている場合、クエリはクラスと同時にコンパイルされます。
  このパラメーターを DYNAMIC に設定すると、クエリは初回実行の前にコンパイルされます（動的 SQL と同様）。
- （任意）クエリ結果の形式を指定する SELECTMODE パラメーターを定義します。
- このクエリを SQL プロシージャとして呼び出す場合は、SqlProc プロパティを追加します。
- クエリの名前を変更する場合は、SqlName プロパティを設定します。 SQL
  コンテキストでのクエリのデフォルト名は PackageName.ClassNameQueryName です。
- Caché Studio には、クラスクエリを作成するためのウィザードが搭載されています。

指定した文字で始まるすべてのユーザー名を返す ByName クエリを使った Sample.Person
クラスのサンプル定義

```
Class Sample.Person Extends %Persistent
{
Property Name As %String;
Property DOB As %Date;
Property SSN As %String;
Query ByName(name As %String = "") As %SQLQuery
    (ROWSPEC="ID:%Integer,Name:%String,DOB:%Date,SSN:%String",
     CONTAINID = 1, SELECTMODE = "RUNTIME",
```

```
     COMPILEMODE = "IMMEDIATE") [ SqlName = SP_Sample_By_Name, SqlProc ]
{
SELECT ID, Name, DOB, SSN
FROM Sample.Person
WHERE (Name %STARTSWITH :name)
ORDER BY Name
}
}
```

このクエリを次のように Caché Object Script から呼び出すことができます。

```
Set statement=##class(%SQL.Statement).%New()
Set status=statement.%PrepareClassQuery("Sample.Person","ByName")
If $$$ISERR(status) {
    Do $system.OBJ.DisplayError(status)
}
Set resultset=statement.%Execute("A")
While resultset.%Next() {
    Write !, resultset.%Get("Name")
}
```

または、自動生成されたメソッドである「クエリ名Func」を使用して結果セットを取得することもできます。

```
Set resultset = ##class(Sample.Person).ByNameFunc("A")
While resultset.%Next() {
    Write !, resultset.%Get("Name")
}
```

このクエリは、SQL コンテキストから次の 2 つの方法で呼び出すこともできます。

```
Call Sample.SP_Sample_By_Name('A')
Select * from Sample.SP_Sample_By_Name('A')
```

このクラスは、SAMPLES のデフォルト Caché
ネームスペースにあります。単純なクエリに関する説明は以上となります。
今度はカスタムクエリを説明します。

カスタムクラスクエリ

基本クラスクエリはほとんどの場合に正常に動作しますが、次のような場合にはアプリケーションでクエリの動作を完全に制御しなければならないことがあります。

- 選択条件が複雑な場合。 カスタムクエリでは次のレコードを返す Caché Object Script メソッドが独自に実装されているため、選択条件が要件に応じて複雑化している可能性があります。
- 使用したくない形式の API経由でしかデータにアクセスできない場合。
- データが（クラスなしで）グローバルに保存されている場合。
- データにアクセスするために権利を昇格する必要がある場合。
- データにアクセスするために外部 API を呼び出す必要がある場合。
- データにアクセスするためにファイルシステムにアクセスする必要がある場合。
- クエリを実行する前に、追加の操作を実行する必要があります（接続の確立、アクセス許可の確認など）。

では、どのようにカスタムクラスクエリを作成しますか？

まず、クエリのワークフロー全体（初期化から破棄まで）を実装する 4
つのメソッドを定義する必要があります。

- queryName — クエリに関する情報を提供します（基本クラスクエリと同様）
- queryName**Execute** — クエリを作成します
- queryName**Fetch** — クエリの次のレコードの結果を取得します
- queryName**Close** — クエリを破棄します

次に、これらのメソッドをさらに詳しく分析します。

queryName メソッド

queryName メソッドはクエリに関する情報を提供します。

- タイプ: %Query
- 本文を空白のままにします。
- 出力結果の名前とデータタイプに関する情報を含む ROWSPEC
  パラメーターをフィールドの順序と共に定義します。
- （任意）フィールドに ID が含まれている場合、番号順に対応する CONTAINID
  パラメーターを定義します。 ID を返さない場合は、CONTAINID に値を割り当てないでください。

例えば、新しい永続クラス Utils.CustomQuery のすべてのインスタンスを 1 つずつ出力する AllRecords
クエリ（queryName = AllRecords、メソッドは単に AllRecords と呼ばれる）を作成してみましょう。
まず、新しい永続クラス Utils.CustomQuery を作成します。

```
Class Utils.CustomQuery Extends (%Persistent, %Populate){
Property Prop1 As %String;
Property Prop2 As %Integer;
}
```

次に、AllRecords クエリを書きます。

```
Query AllRecords() As %Query(CONTAINID = 1, ROWSPEC = "Id:%String,Prop1:%String,Prop2
:%Integer") [ SqlName = AllRecords, SqlProc ]
{
}
```

queryNameExecute メソッド

queryNameExecute メソッドはクエリを完全に初期化します。 このメソッドのシグネチャは次のとおりです。

```
ClassMethod queryNameExecute(ByRef qHandle As %Binary, args) As %Status
```

説明:

- qHandle はクエリ実装の他のメソッドとの通信に使用されます。
- このメソッドは、qHandle を後で queryNameFetch メソッドに渡す状態に設定する必要があります。
- qHandle は OREF、変数、または多次元変数に設定できます。
- args はクエリに渡される追加のパラメーターです。
  必要な数の引数を追加できます（または引数をまったく使用しないでください）。
- このメソッドはクエリの初期化ステータスを返します。

では、再び例に戻りましょう。 複数の方法で範囲内を自由に反復処理できます（以下でカスタムクエリの基本的な処理方法を説明します）。ただし、この例では関数 $Order を使用してグローバルを反復処理します。

この場合、qHandle は現在の ID を格納します。追加の引数は必要ないため、arg 引数は必要ありません。
結果は次のようになります。

```
ClassMethod AllRecordsExecute(ByRef qHandle As %Binary) As %Status {
    Set qHandle = ""    Quit $$$OK
}
```

queryNameFetch メソッド

queryNameFetch メソッドは、単一の結果を $List 形式で返します。
このメソッドのシグネチャは次のとおりです。

```
ClassMethod queryNameFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
As %Integer = 0) As %Status [ PlaceAfter = queryNameExecute ]
```

説明:

- qHandle はクエリ実装の他のメソッドとの通信に使用されます。
- クエリが実行されると、qHandleには queryNameExecute または queryNameFetch の以前の呼び出しによって指定された値が割り当てられます。
- すべてのデータが処理されると、レコードが値 %List または空の文字列に設定されます。
- データの終端に達したら、AtEnd が 1 に設定されます。
- PlaceAfter キーワードは、int コードでのメソッドの位置を識別します。 "Fetch" メソッドは "Execute" メソッドの後に配置する必要がありますが、これは 静的 SQL（つまり、クエリ内の カーソル ）にのみ重要です。

一般に、このメソッド内では次の処理が実行されます。

1. データの終端に達したかどうかを確認します。
2. まだデータが残っている場合は新しい %List を作成し、Row 変数に値を割り当てます。
3. それ以外の場合は、AtEnd を 1 に設定します。
4. 次の結果を取得するために qHandle を準備します。
5. ステータスを返します。

この例では次のようになります。

```
ClassMethod AllRecordsFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
 As %Integer = 0) As %Status {
    #; ^Utils.CustomQueryD ?????
    #; qHandle ??? id ????????? id ??????????? val ?????
    Set qHandle = $Order(^Utils.CustomQueryD(qHandle),1,val)
    #; ??????????????????????
       If qHandle = "" {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    #; ???????? %List ?????
    #; val = $Lb("", Prop1, Prop2) Storage ??????
    #; Row =$lb(Id,Prop1, Prop2)  AllRecords ?????????? ROWSPEC ???
    Set Row = $Lb(qHandle, $Lg(val,2), $Lg(val,3))
    Quit $$$OK
}
```

queryNameClose メソッド

queryNameClose メソッドはすべてのデータが取得された時点でクエリを終了します。
このメソッドのシグネチャは次のとおりです。

```
ClassMethod queryNameClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = queryN
ameFetch ]
```

説明:

- Caché は queryNameFetch メソッドの最後の呼び出しの後にこのメソッドを実行します。
- つまり、クエリのデストラクタとも言い換えることができます。
- したがって、その実装ではすべての SQL カーソル、クエリ、ローカル変数を破棄する必要があります。
- このメソッドは現在のステータスを返します。

この例では、ローカル変数 qHandle を削除する必要があります。

```
ClassMethod AllRecordsClose(ByRef qHandle As %Binary) As %Status {
    Kill qHandle
    Quit $$$OK
  }
```

以上です！ クラスをコンパイルすると、基本クラスクエリと同様に %SQL.Statement から AllRecords
クエリを使用できるようになります。

カスタムクエリの反復ロジック手法

では、カスタムクエリにはどのような手法を使用できるのでしょうか？ 一般的には、次の 3
つの基本的な手法があります。

- グローバルによる反復
- 静的 SQL
- 動的 SQL

グローバルによる反復
この手法は、グローバルによる反復に $Order などの関数を使用することを基本にしています。
この手法は次の場合に使用できます。

- データが（クラスなしで）グローバルに保存されている場合。
- コード内の glorefs の数を減らしたい場合。
- 結果をグローバルの添え字で並べ替える必要がある/並べ替え可能な場合。

静的 SQL
この手法は、カーソルと静的 SQL を基本にしています。 この手法は次の目的で使用されます。

- int コードの可読性を高める。
- カーソルの処理を簡単にする。
- コンパイルプロセスの高速化（静的 SQL はクラスクエリに含まれるため、コンパイルは 1
  回だけ実行されます）。

注意事項:

- %SQLQuery タイプのクエリから生成されたカーソルには、Q14

などのように自動的に名前が付けられます。
- クラス内で使用されるすべてのカーソルは、異なる名前を持つ必要があります。
- エラーメッセージは、名前の末尾に追加の文字があるカーソルの内部名が関係しています。
  例えば、カーソル Q140 のエラーは実際にはカーソル Q14 によって引き起こされたものです。
- PlaceAfter を使用し、カーソルが宣言箇所と同じ int ルーチンで使用されるようにしてください。
- INTO は FETCH と組み合わせて使用する必要がありますが、DECLARE とは組み合わせないでください。

Utils.CustomQuery の 静的 SQL の例：

```
Query AllStatic() As %Query(CONTAINID = 1, ROWSPEC = "Id:%String,Prop1:%String,Prop2:
%Integer") [ SqlName = AllStatic, SqlProc ]
{
}

ClassMethod AllStaticExecute(ByRef qHandle As %Binary) As %Status
{
    &sql(DECLARE C CURSOR FOR
        SELECT Id, Prop1, Prop2
        FROM Utils.CustomQuery
     )
     &sql(OPEN C)
    Quit $$$OK
}

ClassMethod AllStaticFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
As %Integer = 0) As %Status [ PlaceAfter = AllStaticExecute ]
{
    #; INTO ? FETCH ??????????
    &sql(FETCH C INTO :Id, :Prop1, :Prop2)
    #; ????????????????????
    If (SQLCODE'=0) {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    Set Row = $Lb(Id, Prop1, Prop2)
    Quit $$$OK
}

ClassMethod AllStaticClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = AllSta
ticFetch ]
{
    &sql(CLOSE C)
    Quit $$$OK
}
```

動的 SQL
この手法は、他のクラスクエリと動的 SQL を基本にしています。 この手法は、複数のネームスペースで SQL クエリを実行する、クエリを実行する前に権限を昇格するなどの SQL
クエリ自体以外の追加処理を実行する必要がある場合に適切です。

Utils.CustomQuery の動的 SQL の例：

```
Query AllDynamic() As %Query(CONTAINID = 1, ROWSPEC = "Id:%String,Prop1:%String,Prop2
:%Integer") [ SqlName = AllDynamic, SqlProc ]
{
}
```

```
ClassMethod AllDynamicExecute(ByRef qHandle As %Binary) As %Status
{
    Set qHandle = ##class(%SQL.Statement).%ExecDirect(,"SELECT * FROM Utils.CustomQue
ry")
    Quit $$$OK
}

ClassMethod AllDynamicFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
 As %Integer = 0) As %Status
{
    If qHandle.%Next()=0 {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    Set Row = $Lb(qHandle.%Get("Id"), qHandle.%Get("Prop1"), qHandle.%Get("Prop2"))
    Quit $$$OK
}

ClassMethod AllDynamicClose(ByRef qHandle As %Binary) As %Status
{
    Kill qHandle
    Quit $$$OK
}
```

代替手法：%SQL.CustomResultSet

%SQL.CustomResultSet クラスからサブクラス化してクエリを作成することもできます。
この手法のメリットは次のとおりです。

- 若干の速度改善
- すべてのメタデータがクラス定義から取得されるため、ROWSPEC が不要であること
- オブジェクト指向の設計原則に準拠できること

%SQL.CustomResultSet クラスのサブクラスからクエリを作成するには、次の手順を実行してください。

1. 結果のフィールドに対応するプロパティを定義します。
2. クエリコンテキストが格納される private プロパティを定義します。
3. コンテキストを開始する %OpenCursor メソッド（queryNameExecute と同様）をオーバーライドします。 エラーが発生時に %SQLCODE と %Message も設定します。
4. 次の結果を取得する %Next メソッド（queryNameFetch と同様）をオーバーライドします。 プロパティを入力します。 このメソッドはすべてのデータが処理された場合に 0 を返し、一部のデータがまだ残っている場合に 1 を返します。
5. 必要に応じて %CloseCursor メソッド（queryNameClose と同様）をオーバーライドします。

Utils.CustomQuery の %SQL.CustomResultSet の例：

```
Class Utils.CustomQueryRS Extends %SQL.CustomResultSet
{
Property Id As %String;
Property Prop1 As %String;
Property Prop2 As %Integer;
Method %OpenCursor() As %Library.Status
{
    Set ..Id = ""
```

```
    Quit $$$OK
}

Method %Next(ByRef sc As %Library.Status) As %Library.Integer [ PlaceAfter = %Execute
 ]
{
    Set sc = $$$OK
    Set ..Id = $Order(^Utils.CustomQueryD(..Id),1,val)
    Quit:..Id="" 0
    Set ..Prop1 = $Lg(val,2)
    Set ..Prop2 = $Lg(val,3)
    Quit $$$OK
}
}
```

これは、次のように Caché Object Script コードから呼び出すことができます。

```
Set resultset= ##class(Utils.CustomQueryRS).%New()
       While resultset.%Next() {
        Write resultset.Id,!
 }
```

また、SAMPLES ネームスペースでは、Samples.Person のクエリを実装する Sample.CustomResultSet クラスという別の例を使用することもできます。

最後に

カスタムクエリは、Caché Object Script コードから SQL 式を分離し、純粋な SQL では難しすぎる可能性がある高度な動作を実装するのに役立ちます。

参考情報

クラスクエリ

グローバルによる反復

静的 SQL

動的 SQL

%SQL.CustomResultSet

Utils.CustomQuery クラス

Utils.CustomQueryRS クラス

この記事の執筆にご協力いただいた Alexander Koblov 氏に感謝します。

#ObjectScript #SQL #オブジェクトデータモデル #コンパイラ #言語 #Caché

ソースURL:https://jp.community.intersystems.com/post/intersystems-iris-%E3%81%AE%E3%82%AF%E3%83%A9%E3%82%B9%E3%82%AF%E3%82%A8%E3%83%AA