記事
Toshihiko Minamoto · 2022年4月26日 8m read

クラスの全プロパティをリストする（ObjectScript がお気に入りな理由)

@Ming Zhou から素晴らしい質問
をいただきました。その回答は、まさに私がObjectScriptを愛用している理由を表しています。

初めて誰かに ObjectScript や IRIS を説明する際、必ず、クラスを記述してコンパイルし、テーブルを取得して、オブジェクトまたはリレーショナルのいずれか最も自然な観点からデータを操作できると説明しています。 いずれにせよ、これは単に、グローバルと呼ばれる非常に高速な内部データ構造を囲む薄めのラッパーであり、速度をさらにバーストさせる必要がある場合にも使用できます。

オタクレベルの人と話すときには、ObjectScipt
はあらゆる類の凝ったメタプログラミングが可
能だと説明します。*たった今書いたクラス*
が、オブジェクトやリレーショナルというアクセス方法から、さらなる高速なアクセスが必要な時に内部データ構造を使う方法まで操作ができるためです。

「継承されたプロパティも含め、クラス内のすべてのプロパティを取得するにはどうすればよいですか？」という質問に対する回答がわかるでしょう。

同じ回答を得られる 3 つの異なる方法を以下に示します。

```
Class DC.Demo.PropertyQuery Extends %Persistent
{

Property Foo As %String;

Property Bar As %Boolean;

/// ????????????????????????
ClassMethod Run()
{
    for method = "FromRelationship","WithQuery","AsQuicklyAsPossible" {
        write !,method,":"
        kill properties
        do $classmethod($classname(),"GetProperties"_method,.properties)
        do ..Print(.properties)
        write !
    }
}

ClassMethod Benchmark()
{
    for method = "FromRelationship","WithQuery","AsQuicklyAsPossible" {
        write !,method,":",!
        set start = $zhorolog
        set startGlobalRefs = $system.Process.GlobalReferences($job)
        set startLines = $system.Process.LinesExecuted($job)
        for i=1:1:1000 {
```

```
            kill properties
            do $classmethod($classname(),"GetProperties"_method,.properties)
        }
        set endLines = $system.Process.LinesExecuted($job)
        set endGlobalRefs = $system.Process.GlobalReferences($job)
        write "Elapsed time (1000x): ",($zhorolog-start)," seconds; ",(endGlobalRefs-
startGlobalRefs)," global references; ",(endLines-startLines)," routine lines",!
    }
}

/// %Dictionary.CompiledClass ????????????????????????
ClassMethod GetPropertiesFromRelationship(Output properties)
{
    // ?????: %OpenId ? Properties.GetNext() ????????????????????????????????
    // ????????????????????????
    set class = ##class(%Dictionary.CompiledClass).IDKEYOpen($classname(),,.sc)
    $$$ThrowOnError(sc)
    set key = ""
    for {
        set property = class.Properties.GetNext(.key)
        quit:key=""
        set properties(property.Name) = $listbuild(property.Type,property.Origin)
        // ???????????????
        do class.Properties.%UnSwizzleAt(key)
    }
}

/// %Dictionary.CompiledProperty ??????????????????????
ClassMethod GetPropertiesWithQuery(Output properties)
{
    // SQL ????????????????????????????????????
    set result = ##class(%SQL.Statement).%ExecDirect(,
        "select Name,Type,Origin from %Dictionary.CompiledProperty where parent = ?",
        $classname())
    if result.%SQLCODE < 0 {
        throw ##class(%Exception.SQL).CreateFromSQLCODE(result.%SQLCODE,result.%Messa
ge)
    }
    while result.%Next(.sc) {
        $$$ThrowOnError(sc)
        set properties(result.Name) = $listbuild(result.Type,result.Origin)
    }
    $$$ThrowOnError(sc)
}

/// ????????????????????????????????????????
ClassMethod GetPropertiesAsQuicklyAsPossible(Output properties)
{
    // ????????????????????????????????????????????????????
    // ?????????
    set key = ""
    set class = $classname()
    for {
        set key = $$$comMemberNext(class,$$$cCLASSproperty,key)
        quit:key=""
        set type = $$$comMemberKeyGet(class,$$$cCLASSproperty,key,$$$cPROPtype)
        set origin = $$$comMemberKeyGet(class,$$$cCLASSproperty,key,$$$cPROPorigin)
        set properties(key) = $listbuild(type,origin)
    }
```

```
}

ClassMethod Print(ByRef properties)
{
    set key = ""
    for {
        set key = $order(properties(key),1,data)
        quit:key=""
        set $listbuild(type,origin) = data
        write !,"property: ",key,"; type: ",type,"; origin: ",origin
    }
}

Storage Default
{
<Data name="PropertyQueryDefaultData">
<Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Foo</Value>
</Value>
<Value name="3">
<Value>Bar</Value>
</Value>
</Data>
<DataLocation>^DC.Demo.PropertyQueryD</DataLocation>
<DefaultData>PropertyQueryDefaultData</DefaultData>
<IdLocation>^DC.Demo.PropertyQueryD</IdLocation>
<IndexLocation>^DC.Demo.PropertyQueryI</IndexLocation>
<StreamLocation>^DC.Demo.PropertyQueryS</StreamLocation>
<Type>%Storage.Persistent</Type>
}

}
```

そしてもちろん、これらいずれかのアプローチを使用しても、答えは同じです。

```
d ##class(DC.Demo.PropertyQuery).Run()

FromRelationship:
property: %%OID; type: %Library.RawString; origin: %Library.RegisteredObject
property: %Concurrency; type: %Library.RawString; origin: %Library.Persistent
property: Bar; type: %Library.Boolean; origin: DC.Demo.PropertyQuery
property: Foo; type: %Library.String; origin: DC.Demo.PropertyQuery

WithQuery:
property: %%OID; type: %Library.RawString; origin: %Library.RegisteredObject
property: %Concurrency; type: %Library.RawString; origin: %Library.Persistent
property: Bar; type: %Library.Boolean; origin: DC.Demo.PropertyQuery
property: Foo; type: %Library.String; origin: DC.Demo.PropertyQuery

AsQuicklyAsPossible:
property: %%OID; type: %Library.RawString; origin: %Library.RegisteredObject
property: %Concurrency; type: %Library.RawString; origin: %Library.Persistent
property: Bar; type: %Library.Boolean; origin: DC.Demo.PropertyQuery
property: Foo; type: %Library.String; origin: DC.Demo.PropertyQuery
```

パフォーマンスを比較します。

```
d ##class(DC.Demo.PropertyQuery).Benchmark()

FromRelationship:
Elapsed time (1000x):
.78834 seconds; 1056000 global references; 2472003 routine lines

WithQuery:
Elapsed time (1000x): .095235 seconds; 28001 global references; 537007 routine lines

AsQuicklyAsPossible:
Elapsed time (1000x): .016422 seconds; 25000 global references; 33003 routine lines
```

これをちょっと分析する限り、まったく期待どおりの結果です。 クラスとプロパティのオブジェクトアクセスは、クラス定義とコンパイルされたクラスのメタデータが多数のグローバルに保存されているため、はるかにコストがかかります。$listbuild リストにすべてのデータが格納されているのではなく（グローバル参照を減らすため）、各グローバルノードに単一の値でツリーに格納されています。 オブジェクトを開くというのは、これらすべてを読み取ることになるため、もちろん「FromRelationship」手法がはるかに最も低速になります。
もちろん、これは IRIS における一般的なオブジェクトアクセスのパフォーマンスを表すものではありません。このケースは、オブジェクトを使用する際の特に悪いケースとしてたまたま発生しただけです。

このクエリと生のグローバルベースのアプローチは、ルーチン行ではなくグローバル参照という意味で似ています。 Dynamic SQL
を使用した上記の単純なアプローチには、イテレーションごとに行うクエリ準備のオーバーヘッドがあります。
このオーバーヘッドの一部を回避するには、準備しておいた %SQL.Statement
を再利用するか、カーソル付きの埋め込み SQLを使用するか（いくつかの理由で、私は気に入っていません）、次のように厄介な方法を取ることができます。

```
/// %Dictionary.CompiledProperty ????????????????????????
ClassMethod GetPropertiesWithEmbeddedQuery(Output properties)
{
    set classname = $classname()

    // ??: 1 ????????????????????????????????????????
    // ????????????????????????????????????????????????????????????
    &SQL(SELECT %DLIST($ListBuild(Name,Type,Origin)) INTO :allProperties FROM %Dictio
nary.CompiledProperty WHERE parent = :classname)
    if (SQLCODE < 0) {
        throw ##class(%Exception.SQL).CreateFromSQLCODE(SQLCODE,%msg)
    }
    if (SQLCODE = 100) {
        quit
    }
    set pointer = 0
    while $listnext(allProperties,pointer,propertyInfo) {
        set properties($list(propertyInfo)) = $list(propertyInfo,2,3)
    }
}
```

これをベンチマークに追加すると、次のようになります。

```
WithEmbeddedQuery:
Elapsed time (1000x): .024862 seconds; 25000 global references; 95003 routine lines

AsQuicklyAsPossible:
```

```
Elapsed time (1000x): .016422 seconds; 25000 global references; 33003 routine lines
```

非常に近い結果です！

#ObjectScript #オブジェクトデータモデル #InterSystems IRIS

ソースURL:
https://jp.community.intersystems.com/post/%E3%82%AF%E3%83%A9%E3%82%B9%E3%81%AE%E5%85%A8%E3%83%97%E3%83%AD%E3%83%91%E3%83%86%E3%82%A3%E3%82%92%E3%83%AA%E3%82%B9%E3%83%88%E3%81%99%E3%82%8B%EF%BC%88objectscript-%E3%81%8C%E3%81%8A%E6%B0%97%E3%81%AB%E5%85%A5%E3%82%8A%E3%81%AA%E7%90%86%E7%94%B1