文章
Weiwei Gu · 二月 24, 2022 阅读大约需 7 分钟

列出类的所有属性 (我喜欢 ObjectScript 的原因)

@Ming Zhou 在 https://community.intersystems.com/post/how-get-all-properties-defined-c
上问了一个很好的问题.....而这个答案正好总结了为什么ObjectScript 是我最喜欢的语言。

当我第一次向别人介绍ObjectScript或IRIS时，我总是解释说，你可以写一个类，编译它，得到一个表，并从对象或关系的角度来处理你的数据--这是一种最自然的方式。

但无论哪种方式，它都只是一个薄薄的包装，包裹着被称为Globals的超级快速的内部数据结构，当你真的需要更快的速度时，你可以使用这些结构。

当我和比较爱较真的技术人员沟通时，我会告诉他们：ObjectScript允许各种炫酷的元编程，因为你可以用完全相同的方式与你刚刚写的类进行交互--从对象或关系的角度。或者在你需要额外的速度时直接使用超快的底层数据结构。

你可以在对这个问题的回答中看到答案： "我怎样才能得到一个类中的所有属性，包括继承的属性？"

这里有三种不同的方式来获得相同的答案。

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

Property Foo As %String;

Property Bar As %Boolean;

/// Demonstrates all the ways to skin this particular cat
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",!
    }
}

/// Get properties using the properties relationship in %Dictionary.CompiledClass
ClassMethod GetPropertiesFromRelationship(Output properties)
{
    // Minor problem: %OpenId and Properties.GetNext() are slow because they load mor
e data than you strictly need.
    // More global references = it takes longer.
    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)
        // Avoids consuming excess memory
        do class.Properties.%UnSwizzleAt(key)
    }
}

/// Get properties using a query against %Dictionary.CompiledProperty
ClassMethod GetPropertiesWithQuery(Output properties)
{
    // Getting properties with SQL avoids the overhead of unnecessary references
    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)
}

/// Get properties using macros wrapping direct global references
ClassMethod GetPropertiesAsQuicklyAsPossible(Output properties)
{
    // Getting properties via macro-
wrapped direct global references is harder to read,
    // but is the fastest way to do it.
    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>
}

}
```

当然，通过任何其他的这些方法，答案仍然是一致的：

```
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
```

比较一下表现：

```
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
```

稍微分析一下，我们看到的情况完全在意料之中。类和属性的对象访问要繁琐很多，因为类的定义和编译后的类元数据是存储在一堆Global中的--不是在一个$listbuild列表中的所有数据（为了减少Global引用），而是在一个"树"中，每个Global节点都有一个值。打开一个对象意味着读取所有这些，所以我们的 "FromRelationship " 方法到目前为止是最慢的。当然，这并不代表在IRIS中对象访问的一般性能--这只是碰巧是一个使用对象方式的最糟糕的一个情况。

我们的查询和基于global的原始方法与 Global reference 地址引用方面是相似的，但不是常规的那种行。上面用动态SQL的简单方法在准备进行查询的时候会有额外的消耗，我们在每次迭代时都会产生这个消耗。为了避免这样的情况，我们可以重新使用准备好的
“ %SQL.Statement ”，使用带有游标的嵌入式SQL（虽然我并不喜欢），或者做一些棘手的事情，比如：

```
/// Get properties using a query against %Dictionary.CompiledProperty
ClassMethod GetPropertiesWithEmbeddedQuery(Output properties)
{
    set classname = $classname()

    // Quick/easy, skip writing a cursor and just extract the data after running a qu
ery that returns one row.
    // The following approach outperforms cursors (left as an exercise), and I hate w
orking with cursors anyway.
    &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)
    }
}
```

把这个加入到基准中，我们可以看到：

```
WithEmbeddedQuery:
```

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

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

这就非常的接近了！

#对象数据模型 #ObjectScript #InterSystems IRIS

源URL: https://cn.community.intersystems.com/post/%E5%88%97%E5%87%BA%E7%B1%BB%E7%9A%84%E6%89%80%E6%9C%89%E5%B1%9E%E6%80%A7-%E6%88%91%E5%96%9C%E6%AC%A2-objectscript-%E7%9A%84%E5%8E%9F%E5%9B%A0