Article
Artem Daugel-Dauge · Mar 28, 2018 9m read

Caché Native Access — working with native libraries in Caché

There are numerous ways to interact with InterSystems Caché: We can start with ODBC/JDBC that are available via SQL gateway. There are API for .NET and Java too. But if we need to work with native binary libraries, such interaction is possible through Caché Callout Gateway, which can be tricky. You can read more about the ways of facilitating the work with native libraries directly from Caché in the article below.

Caché Callout Gateway

Caché uses Caché Callout Gateway for working with native code. This name applies to a few functions united under a single name, $ZF(). These functions are divided into two groups:

- $ZF(-1), $ZF(-2). The first group of functions allows you to work with system commands and shell scripts. It ' s an efficient tool, but its shortcoming is evident – the entire functionality of the library cannot be implemented in one or several programs.

  #### An example of using $ZF(-1)

  Creation of a new directory called " newdir " in the working directory:

  ```
  set name = "newdir"
  set status = $ZF(-1, "mkdir " _ name)
  ```

- $ZF(-3), $ZF(-5), $ZF(). The second group of functions provides access to dynamic and static libraries. It looks more like what we need. But it ' s not so easy: $ZF() functions do not work with any libraries, but only with libraries of a particular type — Callout Libraries. A Callout Library differs from a regular library in that its code has a special character table called ZFEntry, which contains a certain version of prototypes of exported functions. Moreover, the type of arguments of the exported functions is strictly limited — only int and a few other pointer types are supported. To make a Callout Library from an arbitrary one, you will most probably need to write a wrapper for the entire library, which is far from convenient.

  #### An example of creating a Callout Library and calling a function from it

  Callout Library, test.c

  ```
  #define ZF_DLL
  #include <cdzf.h> // the cdzf.h file is located in Cache/dev/cpp/include
  int
  square(int input, int *output)
  {
    *output = input * input;
    return ZF_SUCCESS;
  }
  ```

```
ZFBEGIN // character table
ZFENTRY("square", "iP", square) // "iP" means that square has two arguments - i
nt and int*
ZFEND
```

Compilation (mingw):

```
gcc -mdll -fpic test.c -o test.dll
```

For linux use -shared instead of -mdll.

Calling square() from Caché:

```
USER> do $ZF(-3, "test.dll", "square", 9)
81
```

Caché Native Access

To remove the limitations of a Callout Gateway and make the work with native libraries comfortable, the CNA project was created. The name is a copy of a similar project for a Java machine, JNA.

CNA capabilities:

- You can call functions from any dynamic (shared) library that is binary compatible with C
- To call functions, you only need code in ObjectScript – you don ' t need to write anything in C or any language compiled into the machine code
- Support of all simple types of the C language, sizet and pointers
- Support of structures (and nested structures)
- Support of Caché threads
- Supported platforms: Linux (x86-32/64), Windows (x86-32/64)

Installation

First, let ' s compile the C part, which is done with a single command —

```
make libffi && make
```

In Windows, you can use mingw or download pre-compiled binary files. After that, import the cna.xml file to any convenient namespace:

```
do $system.OBJ.Load("path to cna.xml", "c")
```

An example of working with CNA

The simplest native library that exists on every system is the C standard library. In Windows, it ' s usually located at C:\Windows\System32\msvcrt.dll, in Linux — /usr/lib/libc.so. Let ' s try calling a function from it, for example, strlen. It has the following prototype:

```
size_t strlen(const char *);


Class CNA.Strlen Extends %RegisteredObject
{
  ClassMethod Call(libcnaPath As %String, libcPath As %String, string As %String) As
%Integer
  {
    set cna = ##class(CNA.CNA).%New(libcnaPath)      // creates an object of CNA.CNA
    do cna.LoadLibrary(libcPath)                     // uploads libc to CNA

    set pString = cna.ConvertStringToPointer(string) // converts the string into the
C format an save a pointer to its beginning

    // Calling strlen: pass the function name, type of returned value,
    // list of argument type and a comma-delimited list of arguments
    set result = cna.CallFunction("strlen", cna.#SIZET, $lb(cna.#POINTER), pString)

    do cna.FreeLibrary()
    return result
  }
}
```

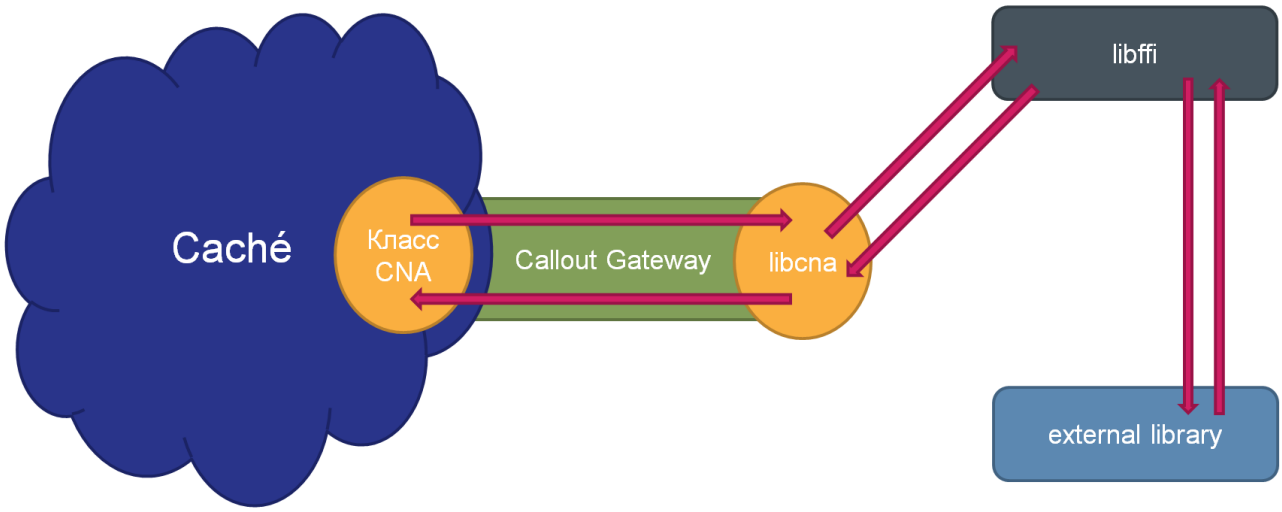
In the shell:

```
USER>w ##class(CNA.Strlen).Call("libcna.dll", "C:\Windows\system32\msvcrt.dll", "hell
o")
5
```

Implementation details

CNA is a link between a C library and a Caché class. CNA heavily relies on libffi. libffi is a library that lets developers organize a “ low-level ” interface for external functions (FFI). It helps forget about the existence of various call conventions and call functions at runtime, without providing their specifications during compilations. However, in order to call functions from libffi, you need addresses, and we ’ d like to do it using names only. To get a function ’ s address by its name, we ’ ll have to use some platform-dependent interfaces: POSIX and WinAPI. POSIX has the dlopen() / dlsym() mechanism for loading a library and searching for a function ’ s address; WinAPI has the LoadLibrary() and GetProcAddress() functions. This is one of the obstacles for porting CNA to other platforms, although virtually all modern system at least partially support the POSIX standard (except for Windows, of course).

libffi is written C and assembler. This makes libffi a native library and you need to use the Callout Gateway to access it from Caché. That is, you need to write a middleware layer that will connect libffi and Caché and be a Callout Library so that you can call it from ObjectScript. Schematically, CAN works like this:



At this stage, we face a data conversion issue. When we call a function from ObjectScript, we pass the parameters in the internal Caché format. We need to pass them to Callout Gateway, then to libffi, but we need to convert them to the C format at some point. However, Callout Gateway supports very few data types and if we converted data on the C side, we ' d have to pass everything as strings, then parse them, which is obviously inconvenient. Therefore, we made a decision to convert all data on the Cache side and pass all arguments as strings with binary data already in the C format.

Since all types of C data, except composite ones, are numbers, this task of data conversion effectively boils down to converting numbers into binary strings using ObjectScript. For this purpose, Caché has some great functions saving you the trouble of accessing data directly: $CHAR and $ASCII. They convert an 8-bit number into a character and back. There are corresponding functions for all the necessary numbers: 16-, 32- and 64-bit numbers and for double-precision floating-point numbers. There is one " but ", however – all these functions work only for signed or unsigned numbers (apparently, we are talking about integers). In C, however, a number of any size can be both signed and unsigned. Therefore, we ' ll need to manually customize these functions to fulfill their purpose.

In C the two ' s complement is used for representing signed numbers:

- The first bit is responsible for the sign of a number: 0 — plus, 1 — minus
- Positive numbers are coded as unsigned ones
- The maximum positive number is 2^{k-1}-1, where k is the number of bits
- The code of a negative number x is the same as that of an unsigned number 2^k+x

This method allows you to use the same addition operation that you use for unsigned numbers. This is achieved with the help of integer overflow.

Let ' s consider an example of converting unsigned 32-bit numbers. If the number is positive, we need to use the $ZLCHAR function, if it ' s negative, we need to find such an unsigned number that their binary representations are identical. The method of searching for this number becomes evident from the very definition of the extra code – we need to add the initial number to the minimal one that doesn ' t fit into 32 bits -2^{32} or $FFFFFFFF_{16}$ + 1. As a result, we have the following piece of code:

```
if (x < 0) {
    set x = $ZLCHAR($ZHEX("FFFFFFFF") + x + 1)
} else
    set x = $ZLCHAR(x)
}
```

The next problem is the transformation of the structures (composite type of C language). Things would be so much easier if the structures in the memory were represented in the same way they were written to it — in a sequence, field after field. However, every structure in the memory is located so that the address of every field is a product of a special field alignment number. Alignment is necessary because most platforms either do not support unaligned data or do it rather slowly. As a rule, the alignment value on the x86 platform is equal to the size of the field, but there are exceptions like the 32-bit Linux, where all fields over 4 bytes equal exactly 4 bytes. More information about data alignment can be found in this article.

Let ' s take this structure, for example:

```
struct X {
    char a, b; // sizeof(char) == 1
    double c;  // sizeof(double) == 8
    char d;
};
```

On the x86-32 platform, it will be located in the memory differently in different operating systems:

In practice, such representation of the structure is formed quite easily. You just need to sequentially write the fields to the memory but add padding – an empty space before each record – every time you perform a write operation. Padding is calculated using this formula:

```
set padding = (alignment - (offset # alignment)) # alignment //offset – the address o
f the end of the last record
```

What ' s not working yet

1) In Caché, integers are represented in a way that accurate work with them is only guaranteed for as long as the number doesn ' t exceed the boundaries of a 64-bit signed number. However, C also has a 64-bit unsigned type (unsigned long long). That is, you won ' t be able to pass a value exceeding the size of a 64-bit signed number, 2^{63}-1($9 * 10^{18}$), to an external function.

2) Caché has two variable types for working with real numbers: its own decimal and double-precision floating-point numbers compliant with the IEEE 754 standard. That is Caché has no equivalents of the float and long double types found in C. You can work with these types in CNA, but they will be automatically converted to double when passed to Caché.

3) If you work on Windows, the use of the long double type will most probably cause problems. This is caused by the fact that Microsoft and the mingw development team have completely different opinions about the size of the long double type. Microsoft believes that its size should be exactly 8 bytes both on 32- and 64-bit systems. Mingw thinks that it should be 12 bytes long on 32-bit systems and 16 bytes long on 64-bit systems. And since CNA is compiled using mingw, forget about using the long double type.

4) Unions and bitfields in structures are not supported. This is caused by the fact that libffi doesn ' t support them.

Any comments or suggestions will be highly appreciated.

The entire source code is available on github under the MIT license.
https://github.com/intersystems-community/cna

#Caché #Interoperability #Callout

Source

URL:https://community.intersystems.com/post/cach%C3%A9-native-access-%E2%80%94-working-native-libraries-cach%C3%A9