Wywoływanie kodu natywnego w C++ z języka Ruby

Ruby to wysokopoziomowy język programowania, który znany jest przede wszystkim z tego, że pozwala na tworzenie eleganckiego i zwięzłego kodu. Niemniej jednak nie jest on powszechnie uznawany za język o wysokiej wydajności. Dlatego też czasem może zaistnieć potrzeba przeniesienia części obliczeń do kodu natywnego, aby zwiększyć szybkość działania programu. W tym artykule opiszę kilka sposobów, które umożliwią osiągnięcie tego celu.

Analiza wydajności będzie odbywała się na systemie Arch Linux, Ruby w wersji 3.2, i9-12900K, 128GB RAM. Dzięki temu porównanie czasu wykonania będzie racjonalne. Jako autor przestrzegam jednak przed wyciąganiem zbyt daleko idących wniosków na podstawie małej liczby próbek.

KOD W RUBY

Jako przykład kodu, który znacząco zyska na przeniesieniu do warstwy wykonywanej natywnie, wezmę szukanie liczb pierwszych w zadanym przedziale. Implementacja naiwnej funkcji `is_prime?` znajduje się w Listingu 0.

Listing 0. Definicja funkcji is_prime?

```
def is_prime? n
    return false if n < 2
    return true if n == 2
    return false if n % 2 == 0
    (3..Math.sqrt(n)).step(2) do |i|
        return false if n % i == 0
    end
    true
end

def find_primes range
    range.select{ |n| is_prime? n }
end
```

Przetestujmy ją na zakresie między $[10^{15}, 10^{15}+99]$. Oczekiwanym rezultatem jest znalezienie dwóch liczb pierwszych:

- $10^{15}+37$
- $10^{15}+91$

Listing 1. Kod implementujący pomiar czasu wykonania za pomocą modułu benchmark

```
Benchmark.bm do |x|
    x.report("find_primes") do
        raise "not equal" unless [
            1000000000000037,
            1000000000000091
        ] == find_primes(
            1_000_000_000_000_000..1_000_000_000_000_099
        )
    end
end
```

Program zostanie wykonany dwukrotnie: za pomocą standardowego interpretera (Listing 2) oraz z wykorzystaniem techniki JIT (ang. *just in time (compilation)*), która pozwala na znaczące przyspieszenie kodu w wielu interpreterach, jak i maszynach wirtualnych (Listing 3).

Listing 2. Wyniki pomiaru czasu wykonania

```
                user     system      total        real
find_primes  1.526844   0.000000   1.526844 (  1.527416)
```

Listing 3. Wyniki pomiaru czasu wykonania z użyciem techniki JIT

```
                user     system      total        real
find_primes  1.003313   0.000627   1.003940 (  1.004472)
```

Mając te wyniki, możemy zaobserwować, że zastosowanie JIT znacząco wpływa na wydajność naszego kodu. Czas wykonania funkcji `find_primes` uległ wyraźnemu skróceniu, co świadczy o korzyściach płynących z optymalizacji kodu przy użyciu tej techniki.

FIDDLE

Gem Fiddle (nie mylić z Fiddler) to biblioteka języka Ruby, która umożliwia bezpośredni dostęp do funkcji zadeklarowanych w języku C z języka Ruby. Pozwala na tworzenie i operowanie na obiektach, operowanie na wskaźnikach i strukturach, a także wywoływanie funkcji.

Fiddle jest elementem biblioteki standardowej, co oznacza, że jest dostępny bez konieczności instalacji dodatkowych gemów. Biblioteka ta działa na zasadzie wiązania dynamicznego (ang. *dynamic linking*).

Zaimplementujmy zbliżony algorytm do tego pokazanego w Listingu 0, ale tym razem używając C++. Funkcja `is_prime` zwraca typ `int`, a nie `bool`, ponieważ niektóre z później wykorzystanych technik będą tego wymagały. W Listingu 4 przedstawiono kod źródłowy algorytmu, a w Listingu 5 jego plik nagłówkowy. Plik projektu CMake znajduje się w Listingu 6.

Listing 4. prime.cpp

```
#include "prime.hpp"
#include <cmath>

extern "C"
int is_prime(int64_t n)
{
    if (n < 2) return 0;
    if (n == 2) return 1;

    if (n % 2 == 0) return 0;

    for (int64_t i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return 0;
    }

    return 1;
}
```

Listing 5. prime.hpp

```
#pragma once
#include <stdint.h>

extern "C"
int is_prime(int64_t n);
```

Listing 6. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.26)

project(prime VERSION 1.0 LANGUAGES CXX)

add_library(prime SHARED
    src/prime.cpp
    src/prime.hpp
)

target_compile_features(prime PUBLIC cxx_std_20)
```

Efektem budowy tego projektu będzie biblioteka libprime.so., którą będzie można linkować dynamicznie, co umożliwi wykorzystanie jej w innych programach – w tym przypadku w interpreterze Ruby.

Listing 7. Wywołanie CMake

```
% cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
-- The CXX compiler identification is GNU 13.1.1
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.2s)
-- Generating done (0.0s)
-- Build files have been written to: /[...]/build
% cmake --build build
[ 50%] Building CXX object CMakeFiles/[...]/prime.cpp.o
[100%] Linking CXX shared library libprime.so
[100%] Built target prime
% ls -lah build/libprime.so
-rwxr-xr-x 1 krzaq krzaq 15K Jun 30 18:40 build/libprime.so
```

Ze względu na poziom skomplikowania projektu odwołamy się do tej biblioteki bezpośrednio, bez instalowania jej w standardowych ścieżkach w systemie. Wykorzystamy do tego bibliotekę Fiddle.

Przede wszystkim trzeba ją dodać do programu, za pomocą komendy `require 'fiddle'`. Następnie ładujemy bibliotekę do jej obiektu-uchwytu:

Listing 8. Ładowanie biblioteki libprime.so za pomocą Fiddle

```
libprime = Fiddle::Handle.new("[...]/libprime.so")
```

Kolejnym krokiem będzie utworzenie uchwytu do konkretnej funkcji oraz zdefiniowanie jej sygnatury. Wykorzystana do tego zostanie klasa `Fiddle::Function`, do której konstruktora podamy następujące parametry:

- adres symbolu `is_prime` z biblioteki: `libprime['is_prime']`
- listę parametrów funkcji jako tablicę: `[Fiddle::TYPE_LONG_LONG]`
- wartość zwracaną przez funkcję: `Fiddle::TYPE_INT`

Listing 9. Definicja funkcji is_prime z użyciem Fiddle

```
IS_PRIME_FUNC = Fiddle::Function.new(
    libprime['is_prime'],
    [Fiddle::TYPE_LONG_LONG],
    Fiddle::TYPE_INT
)
```

Teraz docelowa funkcja może być wywołana za pomocą metody `call` utworzonego obiektu. Pełny przykład pokazany jest w Listingu A.

Listing A. Pełen przykład wykorzystania Fiddle w kodzie Ruby

```
require 'benchmark'
require 'minitest'
require 'fiddle'

libprime = Fiddle::Handle.new("[...]/libprime.so")

IS_PRIME_FUNC = Fiddle::Function.new(
    libprime['is_prime'],
    [Fiddle::TYPE_LONG_LONG],
    Fiddle::TYPE_INT
)

def is_prime? n
    IS_PRIME_FUNC.call(n) == 1
end

def find_primes range
    range.select{ |n| is_prime? n }
end

Benchmark.bm do |x|
    x.report("find_primes") do
        raise "not equal" unless [
            1000000000000037,
            1000000000000091
        ] == find_primes(
            1_000_000_000_000_000..1_000_000_000_000_099
        )
    end
end
```

Wywołanie tego kodu daje następujące rezultaty:

Listing B. Pomiar czasu wykonania programu z użyciem Fiddle

```
                user     system      total        real
find_primes  0.079105   0.000433   0.079538 (  0.079642)
```

Można zauważyć, że przyspieszenie jest prawie dwudziestokrotne w porównaniu do kodu w Ruby, mimo że nie została zastosowana żadna optymalizacja na poziomie algorytmicznym.

FFI

FFI jest biblioteką Ruby o działaniu bardzo zbliżonym do biblioteki Fiddle – pozwala ona na dynamiczne wiązanie bibliotek i wywoływanie zawartego w nich kodu. Różnice znajdują się głównie na warstwie syntaktycznej. FFI zainstalujemy za pomocą polecenia `gem install ffi`. Następnie definiujemy moduł odpowiedzialny za reprezentację natywnej biblioteki, którą chcemy wykorzystać – w tym przypadku będzie to dokładnie ta sama biblioteka, co wcześniej.

Listing C. Definicja modułu LibPrime wykorzystującego bibliotekę libprime.so

```
module LibPrime
    extend FFI::Library
    ffi_lib '[...]/libprime.so'
    attach_function :is_prime, [:long_long], :int
end
```

Warto zauważyć, że w tej definicji biblioteki od razu zawarte są definicje funkcji. Teraz, aby wywołać funkcję `is_prime`, możemy użyć jej bezpośrednio w naszym kodzie, korzystając z modułu `LibPrime`.

Listing D. Implementacja funkcji is_prime? z wykorzystaniem FFI

```
def is_prime? n
    LibPrime.is_prime(n) == 1
end
```

KRZYWE BÉZIERA | HISTORIA JEDNEGO REFAKTORA | DESERIALIZACJA W PHP

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

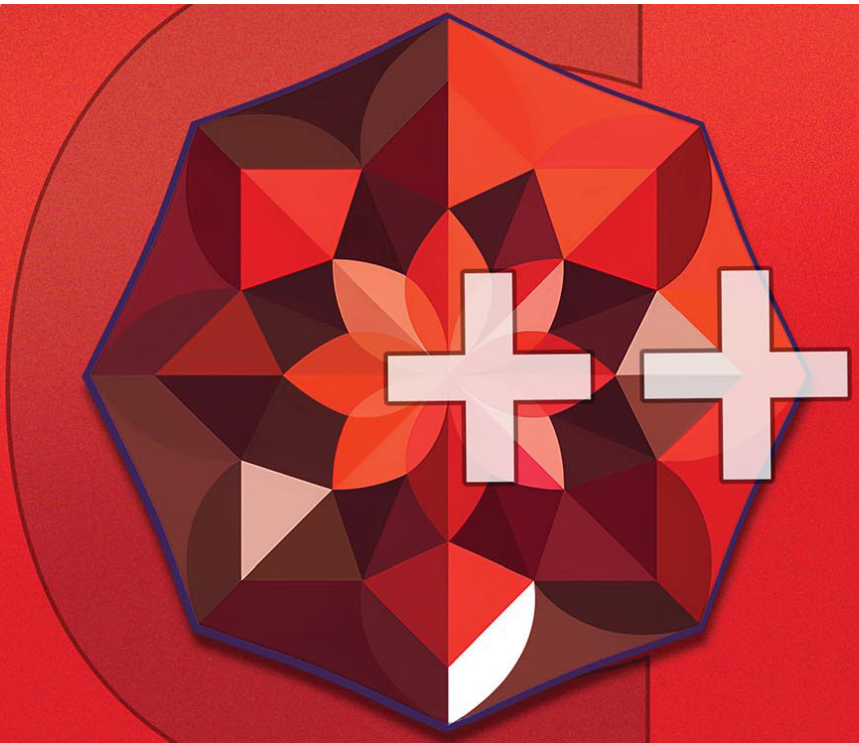
programista

3/2023 (108)

Cena 28,90 zł (w tym VAT 8%)

WYWOŁYWANIE KODU NATYWNEGO W C++ Z JĘZYKA RUBY

Aktualny numer już w sprzedaży



ISSN 2084-9400

03

9 772084 940305

Gdzie te dane? O zachowaniu spójności z Transactional Outbox Pattern

GPU Audio – od teorii do praktyki

MySQL Shell plugin dla Visual Studio Code

Jak AutoML zmienia sposób postrzegania uczenia maszynowego?