Gwiazda morska i korutyny, czyli Seastar w praktyce

Seastar jest zaawansowaną, otwartoźródłową biblioteką C++ zaprojektowaną do implementacji aplikacji serwerowych wymagających ekstremalnej wydajności na nowoczesnym sprzęcie wielordzeniowym. Dzięki swojemu unikalnemu podejściu do alokacji zasobów oraz zorientowaniu na asynchroniczność, Seastar umożliwia pełne wykorzystanie potencjału współczesnych procesorów, minimalizując jednocześnie opóźnienia. Biblioteka ta jest sercem ScyllaDB – niezwykle wydajnej bazy danych NoSQL kompatybilnej z Apache Cassandra.

W tym artykule zaprezentuję wybrane zastosowania biblioteki Seastar, szczególnie pod kątem jej nastawienia na tworzenie wysoce asynchronicznego kodu. Tekst ten nie aspiruje do miana kompleksowej dokumentacji czy szczegółowego kursu i nie powinien być tak traktowany. Głównym jego celem jest wprowadzenie czytelnika w kluczowe koncepcje i możliwości tej technologii.

KONTYNUACJE, KORUTYNY I ASYNCHRONICZNOŚĆ

Kontynuacje (ang. *continuations*) to koncepcja programistyczna, która odnosi się do zapisywania bieżącego stanu wykonywania programu w sposób umożliwiający jego późniejsze wznowienie. W praktyce oznacza to, że program może zapamiętać, gdzie skończył, a następnie kontynuować działanie w innym punkcie czasu lub w innej części programu. Ta technika pozwala na eliminację tradycyjnego blokowania wątków, co czyni ją fundamentalnym elementem w aplikacjach wymagających wysokiej asynchroniczności. W Seastar kontynuacje są kluczowe dla implementacji asynchroniczności i harmonogramowania zadań.

Kontynuacje w Seastar

W Seastar kontynuacje są reprezentowane jako obiekty `seastar::future`. Kiedy zadanie asynchroniczne kończy się, przekazuje kontrolę do następnej funkcji, która została zarejestrowana jako kontynuacja. Dzięki temu można:

- **Eliminować blokowanie:** Kontynuacje pozwalają na rozbijanie zadań na mniejsze fragmenty, które są wykonywane, gdy wymagane zasoby są dostępne.
- **Zachować porządek w kodzie:** Zamiast pisać złożony kod oparty na wywołaniach zwrotnych (callbacki), programista może korzystać z kontynuacji, co w odpowiednich wypadkach może poprawiać czytelność. Należy jednak zauważyć, że często bywa całkowicie odwrotnie i kod traci na czytelności.
- **Efektywnie zarządzać zasobami:** Dzięki pętli zdarzeń (ang. *event loop*) zadania korzystające z kontynuacji są uruchamiane tylko wtedy, gdy są dostępne wszystkie zasoby konieczne do ich wykonania.

Jak to działa?

Każde asynchroniczne zadanie w Seastar zwraca obiekt wspomnianego już typu `seastar::future`, który reprezentuje jego wynik. Jego interfejs jest podobny do zbliżonych typów z biblioteki standardowej, biblioteki Boost, frameworka Qt lub bibliotek innych języków programowania.

Seastar udostępnia bogaty zestaw funkcji umożliwiających manipulację kontynuacjami. Oto najczęściej wykorzystywane z nich:

- `get()` – synchroniczne oczekiwanie na wynik `seastar::future`. Używane głównie w testach i procesach inicjalizacyjnych.
- `then()` – rejestracja kontynuacji wykonywanej po zakończeniu zadania.
- `then_wrapped()` – rejestracja kontynuacji, która zostanie wykonana po zakończeniu obecnego zadania, nawet jeśli nie zawiera ona wyniku, a wyjątek.
- `handle_exception()` – obsługa wyjątków zgłaszanych podczas wykonania zadania.
- `finally()` – definiowanie operacji wykonywanych niezależnie od wyniku poprzedzającego zadania.

Przykładowy kod tradycyjnego programu „Hello, World!” za pomocą kontynuacji znajduje się w Listingu 0.

Listing 0. Program „Hello, World”

```
#include <seastar/core/app-template.hh>

#include <print>

int main(int argc, char** argv) {
    seastar::app_template app;
    return app.run(argc, argv, [] {
        std::print("Hello, World!\n");
        return seastar::make_ready_future<>();
    });
}
```

Należy zwrócić uwagę na funkcję `app.run()`, która uruchamia pętlę zdarzeń i przyjmuje główną kontynuację. W tym przypadku jest to wyrażenie lambda zwracające obiekt typu `seastar::future` za pomocą funkcji `make_ready_future()`. Funkcja ta zwraca obiekt kontynuacji, który jest od razu gotowy do dalszego działania. Jest to wykorzystywane w przypadkach, gdy dalsze zawieszenie działania kodu

nie jest konieczne i znamy wartość wynikową, lub jeśli wykonaliśmy wszystko, co było w danej funkcji do wykonania.

Przejdźmy teraz do bardziej zaawansowanego przykładu, aby zilustrować, jak Seastar obsługuje zadania asynchroniczne z wykorzystaniem kontynuacji. W Listingu 1 przedstawiono program, który wstrzymuje swoje działanie na określony czas (10 sekund) za pomocą funkcji `seastar::sleep()`. Następnie wznawia działanie i wykonuje kolejną zarejestrowaną kontynuację, która wypisuje komunikat o zakończeniu przerwy.

Ten przykład pokazuje elastyczne zarządzanie przepływem programu w oparciu o zdarzenia i zależności czasowe, jednocześnie unikając blokowania wątków.

Listing 1. Program wykorzystujący seastar::sleep()

```
#include <seastar/core/app-template.hh>
#include <seastar/core/sleep.hh>

#include <print>

using namespace std::chrono_literals;

seastar::future<> continuation_main() {
    std::print("Sleeping for 10 seconds\n");
    return seastar::sleep(10s).then([] {
        std::print("Done sleeping\n");
        return seastar::make_ready_future<>();
    });
}

int main(int argc, char** argv) {
    seastar::app_template app;
    return app.run(argc, argv, continuation_main);
}
```

CZYM SĄ KORUTYNY

Korutyny to mechanizm programistyczny, który pozwala na wstrzymywanie i wznawianie wykonywania funkcji w kontrolowany sposób. W odróżnieniu od tradycyjnych funkcji, które po zakończeniu zwracają kontrolę do miejsca ich wywołania, korutyny umożliwiają wielokrotne wznawianie działania od miejsca, w którym zostały wstrzymane. Jest to kluczowe w aplikacjach asynchronicznych, gdzie blokowanie zasobów, np. podczas oczekiwania na operacje wejścia/wyjścia czy zakończenie zadań w tle, jest wysoce niepożądane i prowadzi do nieefektywności.

Podstawową różnicą między korutynami a kontynuacjami jest poziom abstrakcji i sposób zarządzania przepływem programu. Korutyny wprowadzają bardziej idiomatyczne podejście, które integruje wstrzymywanie i wznawianie operacji w składni samego języka, co często prowadzi do bardziej zwięzłego i czytelnego kodu. Kontynuacje natomiast działają na niższym poziomie abstrakcji, oferując precyzyjną kontrolę nad przepływem programu, ale mogą wprowadzać większą złożoność w strukturze kodu. W praktyce oba mechanizmy znajdują zastosowanie w różnych scenariuszach, a ich wybór zależy od specyfiki problemu.

Standard C++20 wprowadził obsługę korutyn, definiując słowa kluczowe:

» `co_await` – wstrzymuje wykonanie korutyny do czasu uzyskania wyniku obliczenia asynchronicznego.
» `co_yield` – zwraca wartość pośrednią, pozwalając na iteracyjne przetwarzanie danych.
» `co_return` – zwraca wynik końcowy i kończy działanie korutyny.

W Seastar korutyny są używane do uproszczenia pisania asynchronicznego kodu, który wcześniej wymagał ręcznego zarządzania obiektami `seastar::future` i `seastar::promise`. Dzięki temu kod staje się bardziej czytelny i łatwiejszy w utrzymaniu.

Jak działają korutyny w Seastar?

Seastar wykorzystuje korutyny do obsługi zadań asynchronicznych w ramach swojej architektury opartej na *shardach*. Każdy *shard* jest przypisany do jednego rdzenia procesora i zarządza własnym zestawem zasobów. Korutyny pozwalają na:

1. **Wstrzymywanie wykonywania zadania:** Dzięki `co_await` można wstrzymać wykonywanie korutyny, czekając na zakończenie asynchronicznej operacji, np. odczytu z sieci.
2. **Bezproblemowe przełączanie między zadaniami:** Seastar zarządza harmonogramem zadań w sposób optymalny, umożliwiając płynne przełączanie kontekstu między różnymi korutynami.
3. **Redukcję blokowania:** Korutyny eliminują potrzebę stosowania tradycyjnych mechanizmów blokujących, takich jak mutexy, dzięki czemu aplikacje działają szybciej i bardziej przewidywalnie.

Listing 2. Implementacja kodu z Listingu 1 przy użyciu korutyn

```
#include <seastar/core/app-template.hh>
#include <seastar/core/coroutine.hh>
#include <seastar/core/sleep.hh>

#include <print>

using namespace std::chrono_literals;

seastar::future<> coroutine_main() {
    std::print("Sleeping for 10 seconds\n");
    co_await seastar::sleep(10s);
    std::print("Done sleeping\n");
    co_return;
}

int main(int argc, char** argv) {
    seastar::app_template app;
    return app.run(argc, argv, coroutine_main);
}
```

W Listingu 2 zaprezentowano implementację kodu z Listingu 1, ale z wykorzystaniem korutyn, które zastępują tradycyjne podejście oparte na kontynuacjach. Funkcja `coroutine_main()` ma strukturę kodu niemal identyczną ze standardową, blokującą funkcją, ale działa asynchronicznie. Dzięki użyciu `co_await` możemy w prosty sposób wstrzymać wykonanie korutyny na 10 sekund, po czym jej działanie zostaje automatycznie wznowione. Należy zauważyć, że konieczne było dodanie nowego pliku do include: *seastar/core/coroutine.hh*.

Kod z Listingu 3 przedstawia bardziej zaawansowany przykład, w którym wykorzystano korutyny do zarządzania wieloma równoległymi zadaniami. W tym przypadku funkcja `wait_for()` wstrzymuje wykonanie na określony czas, po czym wznowienie działania jest sygnalizowane odpowiednim komunikatem logowania. Funkcja `coroutine_main()` ilustruje użycie `when_all()`, co pozwala na równoległe uruchamianie kilku korutyn, a następnie oczekiwanie na ich zakończenie.

W bibliotece Seastar wykonanie nowego zadania nastąpi na tym samym procesorze (shardzie), o ile nie zostanie *explicite* przekazane do innego za pomocą funkcji `seastar::smp::submit_to()`.

INDEX: 285358

www.programistamag.pl

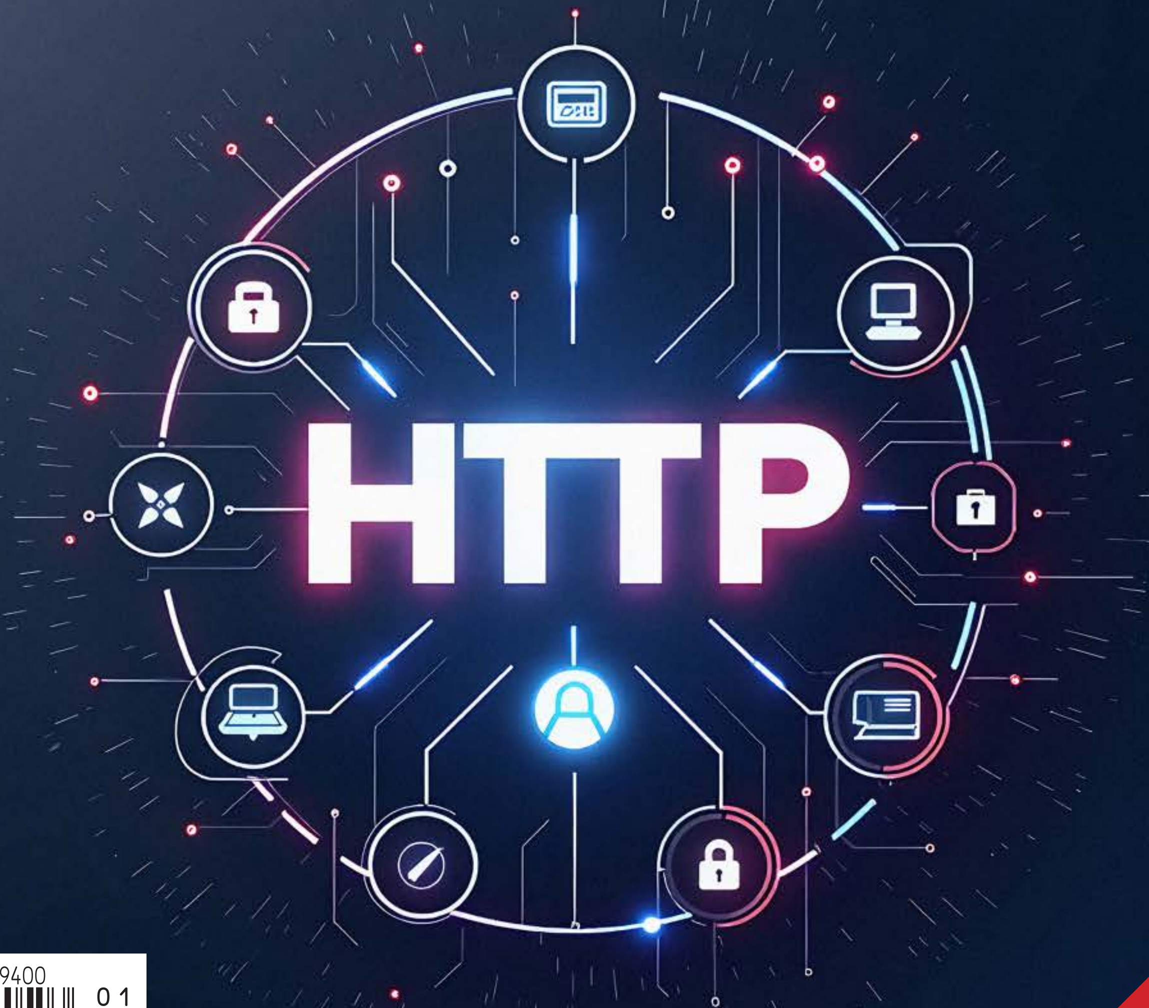
Magazyn programistów i liderów zespołów IT

programista

1/2025 (116)

Cena 32.90 zł (w tym VAT 8%)

JAK DZIAŁA INTERNET



Współbieżność CSP
w języku Go

Rekin w strumieniu...
...czyli hakujemy Stream Decka

Gwiazda m... korutyny,
czyli Sea... aktyce

CSS ... lasses
... adzie TailwindCSS

NOWY NUMER JUŻ W EMPIKACH