Komparativna analiza dva HTTP servera implementiranih u Java i Rust programskim jezicima

Uvod

Tema rada je poređenje dva HTTP servera implementiranih koristeći dva različita programska jezika, u ovom slučaju Java i Rust. Pre svega ćemo posmatrati različite principe i paradigme vezane za oba programska jezika nakon čega ćemo se pozabaviti karakteristikama i performansama oba pristupa.

Dodatni materijali za ovaj seminarski rad mogu se pronaći na sledećem linku:
https://github.com/rruzicic/pdaj-seminarski

Pregled tehnologija

Rust

Rust je moderni, statički tipizirani programski jezik dizajniran za razvoj sigurnih i efikasnih sistema. Razvio ga je Mozilla Research team, a prvi put je predstavljen 2010. godine. Fokusiran je na rukovanje niskim nivoima programiranja bez gubitka sigurnosti i udobnosti visokog nivoa jezika.

Rust je takođe nešto što se u industriji naziva sistemski programski jezik ili, programski jezik fokusiran, između ostalog, na razvijanje sistemskog softvera. U red sistemskih jezika spada i C programski jezik koji se između ostalog koristi za implementaciju Linux kernela, od skoro je i Rust našao svoj put do Linux kernela pa ga neki zbog toga smatraju naslednikom C-a.[1]

Zašto baš Rust?

Zato što ima jedinstven pristup rukovanjem memorijom, on koristi nešto što se naziva pozajmljivanje i vlasništvo("borrow and ownership") nad podacima, ovo je jedna od glavnih osobina ovog programskog jezika i kao takva zaslužna je za veliki broj odluka prilikom razvoja istog. Kako bi Rust obezbedio adekvatno rukovanje vlasništvima on koristi "borrow checker" mehanizam. Ovaj sistem je ključni deo Rust-ove strategije za bezbedno upravljanje memorijom i sprečavanje uobičajenih grešaka kao što su dereferenciranje "null" pokazivača, upisivanje izvan granica bafera i slično. U Rust-u, kada se vrednost pozajmi ("borrow") nekoj promenljivoj, sistem prati kada i gde ta vrednost može biti korišćena tokom životnog veka pozajmljivanja. Ovaj proces praćenja se naziva "borrow checking" i obavlja se tokom faze kompilacije. "Borrow checker" analizira kod i osigurava da ne dolazi do određenih problema vezanih za konkurenciju i vlasništvo nad podacima. Ovaj sistem je ključan za postizanje visokog stepena sigurnosti i izbegavanje trka za podacima ("data races") u Rust-u. Iako može izazvati izazove u učenju i pisanju koda, dugoročno doprinosi pouzdanosti i bezbednosti Rust programa.[2]

Java

Java je objektno-orijentisani programski jezik visokog nivoa apstrakcije koji je dizajniran za pisanje legacy koda. Prvi put je predstavljen od strane Sun Microsystems-a 1995. godine, a danas se koristi širom sveta u različitim domenima, uključujući web razvoj, mobilni razvoj, korporativne aplikacije.[3]

Zašto baš Java?

Java se tokom decenija izdvojila kao pogodan programski jezik za pisanje aplikacija koje imaju dugačak životni vek. Pod terminom dugačak životni vek mislimo na dve različite stvari: 1. business(enterprise) aplikacije koje se održavaju ponekad i decenijama, 2. vreme izvršavanja Java programa zna da traje danima/nedeljama/mesecima pa čak i godinama.

Java se izvršava na "Java Virtual Machine" (JVM), što je čini nezavisnom od hardverske platforme. Ovo omogućava pokretanje Java programa na različitim uređajima bez potrebe za prekompilacijom. Na bazi JVM su nastali i drugi poznati programski jezici poput: Scala, Kotlin, Groovy, Clojure ... itd.

Upravljanje memorijom u Javi vrši se automatski, tako da je sva logika vezana za "memory management" prebačena sa programera na interni mehanizam koji se zove "garbage collector". Zadatak "garbage collector"-a je prilično jednostavan, on zauzima memoriju kada program od njega to zatraži i nakon toga će konstantno proveravati da li je ta memorija referencirana u programu, ako nije "garbage collector" će istu osloboditi.[4]

Poređenje Rust i Java programskih jezika

Rust	**Java**
Sistemski programski jezik koji je orijentisan ka "memory safety".	Objektno orijentisani programski jezik visokog nivoa fokursiran na viši nivo apstrakcije.
Ručno upravljanje memorijom	"Garbage collector" koji upravlja memorijom umesto programera.
Većina grešaka se pronađe tokom faze kompajliranja.	Većina grešaka se pronađe tokom faze izvršavanja.
Kompajlira se i izvršava kao jedan binarni fajl.	Kompajlira se i onda se izvršava uz pomoć JVM.
Standardna biblioteka se fokusira na primitivne koncepte.	Standardna biblioteka fokusirana na uvođenje što više funkcionalnosti.

Implementacija rešenja

Java

Za implementaciju webservera u Javi koristili smo standardni `com.sun.net.httpserver` paket. Kao što možemo videti u navedenom isečku koda bilo je potrebno kreirati instancu klase `com.sun.net.httpserver.HttpServer` uz pomoć statičke metode iste klase. Nakon toga smo uz pomoć metode naveli "endpoint" i objekat `StoreHandler` klase koja vrši procesiranje zahteva. Kako bismo obezbedili konkurentnost koristili smo `setExecutor` metodu koja prima instancu `Executor` klase. U našem slučaju koristili smo `Executor` koji koristi keširani connection pool.

```
    public static void main(String[] args) throws Exception {
        int port = 8080;
        System.out.println("Starting web server on port " + port);
        HttpServer httpServer = HttpServer.create(new
InetSocketAddress(port), 0);
        httpServer.createContext("/test", new StoreHandler());

httpServer.setExecutor(java.util.concurrent.Executors.newCachedThreadPool()
);
        httpServer.start();
    }
```

Da bismo implementirali procesiranje zahteva morali smo da napravimo `StoreHandler` klasu. Ta klasa nasleđuje `com.sun.net.httpserver.HttpHandler` klasu i "override"-uje njenu `handle(com.sun.net.httpserver.HttpExchange t)` metodu koja kasnije pozivom pomoćnih metoda parsira URI i čuva podatke u "key value store"-u.

Key value store definisan je kao privatno polje unutar `StoreHandler` klase zajedno sa pripadajućim "getter"-om i "setter"-om. Primetićete da je tip vrednosti mape objekat tipa `ValueMapper`. Klasa `ValueMapper` kreirana je kako bi omogućila čuvanje različitih tipova kao vrednost mape. Unutar te klase nalazi se podrazumevani konstruktor čiji je zadatak da proba da parsira prvo "floating point" vrednost, pa ako to ne uspe da proba da parsira celobrojnu vrednost a ako ni to ne uspe onda će sačuvati datu vrednost kao `String`.

```
    private Map<Integer, ValueWrapper> keyValueStore;
```

Rust

Kako bismo obezbedili bezbedno čuvanje "key value store"-a morali smo postojeću HashMap-u obaviti u `Arc` pametni pokazivač koji se koristi u situacijama kada imamo višenitne programe čije niti dele istu promenljivu.

```
    let key_value_store: Arc<RwLock<HashMap<i64, Value>>> =
Arc::new(RwLock::new(HashMap::new()));
```

Kao što možemo videti naša `HashMap`-a u ovom primeru za ključ uzima `i64` a za vrednost uzima enum `Value` koji smo definisali zbog potrebe čuvanja više različitih tipova promenljivih u ključu naše "HashMap"-e.

```
enum Value {
    Int(i64),
    Float(f64),
```

```
    String(String),
}
```

Da bismo obezbedili konkurentnost izvršavanja programa napravili smo jednu for(each) petlju čiji je zadatak da za svaku novu TCP konekciju koja je usmerena ka nama "spawn"-uje novu nit koja će procesirati zahtev.

```
for stream in listener.incoming() {
        let stream = stream.unwrap();
        let cloned_kv_store = Arc::clone(&key_value_store);
        thread::spawn(move || {
            let this_map = cloned_kv_store.write().expect("msg");
            handle_connection(stream, this_map);
        });
    }
```

Unutar `handle_connection` funkcije nalazi se kod koji će vršiti procesiranje zahteva, ovo je zapravo najdosadniji deo programa. Potrebno je isparsirati zahtev pristupajući ulaznom `TcpStream`-u i obraditi isti kreativnom upotrebom stringova.

Performanse

Kako bismo testirali performanse koristili smo cURL alat. cURL obezbedjuje slanje više uzastopnih zahteva, za svaki taj zahtev možemo izdvojiti metrike koje nas interesuju na primer, vreme odziva, brzina preuzimanja, veličina odgovora itd. Testiranje smo vršili tako što smo poslali prvo 1000 PUT zahteva kako bismo popunili memoriju a zatim smo poslali 1000 GET zahteva. Za testiranje smo koristili sledece komande:

```
curl -s -o /dev/null -X PUT 'http://127.0.0.1:8080/test?key=[1-
1000]&value=testtest' -w "%{time_connect}|%{time_starttransfer}|%
{time_total}|%{speed_download}|%{size_download}\n" > stats-java-put.csv
```

```
curl -s -o /dev/null -X GET 'http://127.0.0.1:8080/test?key=[1-
1000]&value=testtest' -w "%{time_connect}|%{time_starttransfer}|%
{time_total}|%{speed_download}|%{size_download}\n" > stats-java-get.csv
```

```
curl -s -o /dev/null -X PUT 'http://127.0.0.1:7878/test?key=[1-
1000]&value=testtest' -w "%{time_connect}|%{time_starttransfer}|%
{time_total}|%{speed_download}|%{size_download}\n" > stats-rust-put.csv
```

```
curl -s -o /dev/null -X GET 'http://127.0.0.1:7878/test?key=[1-
1000]&value=testtest' -w "%{time_connect}|%{time_starttransfer}|%
{time_total}|%{speed_download}|%{size_download}\n" > stats-rust-get.csv
```

Nakon obrade podataka dosli smo do sledeće tabele koja predstavlja vreme izvršavanja po zahtevu.

	Java	**Rust**
PUT	0.0440	0.0000089
GET	0.0441	0.00070

Ovi brojevi na prvi pogled ne znače puno ali kada uzmemo u obzir da smo slali 1000 zahteva ispostavi se da Rust HTTP server prihvati i obradi sve PUT zahteve za manje od sekunde tj 0.7 sekundi a da je Java HTTP serveru za isti posao potrebno oko 44 sekunde. Moramo napomenuti da je za obe aplikacije moguće uvesti još dosta optimizacija kao i podešavanja različitih parametara tokom kompilacije i pokretanja programa tako da treba biti obazriv prilikom izvođenja zaključaka.

Reference

[1] https://en.wikipedia.org/wiki/Rust_(programming_language)

[2] https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html

[3] https://en.wikipedia.org/wiki/Java_(programming_language)

[4] https://www.baeldung.com/jvm-garbage-collectors