Klauzurní Práce
Dokumentace

David Frnoch

Chatovací platforma
Fresh

Creative Hill College

3. ročník, 2. pololetí

2023

Úvod

Fresh Chat je jednoduchá chatovací aplikace založená na terminálu, která se skládá ze serveru a několika připojených klientů. Cílem této dokumentace je vysvětlit strukturu kódu a klíčová návrhová rozhodnutí aplikace.

I. Přehled struktury kódu

Struktura kódu je rozdělena na tři hlavní části: serverový kód v adresáři `server`, klientský kód v adresáři `client` a společný adresář `common`.

Klíčové komponenty jsou:

- `common` - Sdílený kód mezi serverem a klientem, včetně:
  - `proto.rs` - Definuje kódování/dekódování JSON zpráv přes TCP.
  - `room.rs` - objekty Room a User reprezentující chatovací místnosti a klienty
- `server` - Obsahuje kód chatovacího serveru, který spravuje:
  - `connection.rs` - Nastavení TCP liestenera a přijímání nových klientů
  - `processing.rs` - Zpracování příchozích zpráv a generování odpovědí
  - `message.rs` - Definuje obálku pro zprávy odesílané mezi komponentami
  - `main.rs` - Spouští server a spravuje smyčku událostí
- `client` - Obsahuje kód klienta chatu založeného na terminálu:
  - `screen.rs` - Spravuje uživatelské rozhraní terminálu pomocí crosstermu
  - `line.rs` - Reprezentuje formátovaný řádek textu
  - `input.rs` - Zpracovává uživatelský vstup a různé režimy
  - `message.rs` - Zpracovává příchozí zprávy ze serveru
  - `main.rs` - Spouští klienta a spravuje smyčku událostí

II. Architektura klient-server

Server a klienti komunikují prostřednictvím připojení TCP pomocí kódování JSON. Modul `proto.rs` v `common` knihovně definuje výčet `Sndr` reprezentující zprávy odesílané klienty a výčet `Rcvr` pro zprávy přijímané klienty.

Například varianta `Sndr::Text` představuje textovou zprávu odeslanou z klienta do chatovací místnosti:

```
Sndr::Text { who: &'a str, lines: &'a [&'a str] }
```

Server zpracovává příchozí zprávy přes `Rcvr` a generuje jednu nebo více odpovědí `Sndr`. Tyto odpovědi jsou zakódovány do JSON a odeslány zpět klientovi.

Struktura `Env` obaluje zprávu `Sndr` a určuje zdrojový a cílový koncový bod zprávy, kterým může být uživatel, místnost nebo celý server:

```
pub enum End {
    User(u64),
```

```
    Room(u64),
    Server,
    All
}

struct Env {
    source: End,
    dest: End,
    data: Sndr: Vec<u8> // JSON-encoded Sndr
}
```

Server udržuje objekty `User` a `Room` pro každého připojeného klienta a chatovací místnost. Když přijde zpráva pro uživatele nebo místnost, je doručena pomocí metod `User.deliver()` a `Room.deliver()`.

V případě dotazů od klientů se v odpovědích serveru používají varianty `Sndr::Info`, `Sndr::Err` a `Sndr::Misc`, které poskytují strukturovaná data zpět klientovi.

Celkově tato architektura umožňuje zachovat jednoduchost klientského kódu tím, že většinu logiky deleguje na server.

III. Architektura serveru

Server spravuje dva hlavní typy objektů - `Users` reprezentující připojené klienty a `Rooms` reprezentující chatovací místnosti.

Ve struktuře `User` jsou uloženy informace jako:

- Jméno klienta
- Kumulované bajty načtené z klienta
- Kvóta bajtů pro omezení rychlosti zpráv
- Seznam uživatelů, které uživatel zablokoval
- Paměť zpráv ve frontě k odeslání

Ve struktuře `Room` jsou zase informace jako:

- Název a ID místnosti
- Seznam uživatelů, kteří jsou aktuálně v místnosti
- ID operátora místnosti
- Zda je místnost otevřená nebo zavřená
- Seznam zakázaných a pozvaných uživatelů

Když přijde nová zpráva od klienta, server vytvoří strukturu `Context`, aby měl během zpracování zprávy přístup ke stavu serveru:

```
struct Context<'a> {
    current_room_id: u64, // Aktuální ID místnosti
    current_user_id: u64, // Aktuální ID uživatele
    users_by_id: &'a mut HashMap<u64, User>,
    ...
}
```

Funkce `process_room()` zpracovává všechny zprávy pro danou místnost. Je to:

- Vytáhne ID všech uživatelů, kteří jsou aktuálně v místnosti.
- Iteruje nad příchozími zprávami každého uživatele.
- Vytvoří `kontext` pro aktuálního uživatele
- Zpracovává příchozí zprávu voláním funkcí jako `do_text()` nebo `do_priv()`.
- shromažďuje vygenerované odpovědi `Sndr v` poli `Envs`
- doručuje odpovědi příslušným uživatelům nebo celé místnosti.

Tato architektura umožňuje oddělit problémy - `User` a `Room` spravují stav, zatímco `Context` zpracovává zprávy pro konkrétního uživatele.

IV. Architektura klienta

Klient používá ke správě uživatelského rozhraní terminálu knihovnu crossterm.

Struktura `Screen` uchovává:

- Vyrovnávací paměť `Lines` představující historii chatu.
- Aktuální vstupní řádek
- Řádky s názvem a ID na stavovém řádku

Struktura `Line` zpracovává formátovaný text pomocí escape kódů ANSI a zachovává:

- Vektor znaků představující text
- Vektor struktur `Fmtr, které` ukládají informace o formátování v konkrétních indexech.

Po přijetí zprávy ze serveru klient zavolá funkci `process_msg(),` aby:

- Parsnul varianty `Rcvr`
- Aktualizoval stav `Screen`
  - Posunutí nového řádku pro textové zprávy
  - Aktualizace seznamu uživatelů
  - Aktualizace stavového řádku pro autentifikační zprávy

Klient zpracovává vstupy od uživatele tak, že:

- Zavolá `funkce process_user_typing()` ve smyčce
- Volá `funkce input_key()` nebo `command_key() v` závislosti na aktuálním režimu.
- Aktualizuje obsah vstupního řádku
- Změní režimy na základě zadaných kláves

Když uživatel stiskne klávesu Enter, zavolá se `funkce respond_to_user_input(),` která:

- Parsne vstupní řádek jako příkaz nebo text
- Zavolá příslušnou variantu `Sndr::`
- Odešle zprávu na server

Struktura `Globals` uchovává data sdílená mezi komponentami, jako jsou:

- Uživatelské jméno
- Aktuální název chatovací místnosti
- Socket
- Znak pro příkaz

Konfigurace a možnosti

Serverové a klientské aplikace načítají konfiguraci ze souborů `.toml`.

`ServerConfig` ukládá:

```toml
[server]
address = "127.0.0.1:1234"
tick_ms = 500
time_to_ping_ms = 10000
time_to_kick_ms = 20000
max_user_name_length = 24
max_room_name_length = 24
lobby_name = "Lobby"
welcome_message = "Vítejte na serveru."
log_file = "server.log"
log_level = 2
byte_limit = 512
bytes_per_tick = 6
```

`ClientConfig` ukládá:

```toml
[client]
address = "127.0.0.1:1234"
name = "Joe"
tick_ms = 100
cmd_char = '/'
roster_width = 24
read_size = 1024
max_scrollback = 2000
min_scrollback = 1000
```

Server a klient volají funkce `configure()`, aby načetly nastavení buď z explicitní cesty, nebo z výchozího umístění.

Argumenty příkazového řádku lze předat také za účelem přepsání konkrétních konfiguračních polí.

To umožňuje přizpůsobit chování serverových a klientských aplikací bez nutnosti upravovat kód a vytvářet konfigurace pro konkrétní případy použití.

Možnosti konfigurace zahrnují:

- Síťové a časové parametry
- Možnosti omezení sazeb a kvót

- Velikosti uživatelského rozhraní
- Úrovně a cesty protokolu
- Omezení délky názvu

Výchozí hodnoty mají za cíl poskytnout rozumné počáteční hodnoty, ale většinu možností lze pro konkrétní nasazení vyladit.

V. Logování a zpracování chyb

Aplikace používají k logování balíček simplelog.

Aplikace volá `WriteLogger::init()`, aby nakonfigurovala úroveň logu a výstup:

```
simplelog::WriteLogger::init(
    simplelog::LevelFilter::Trace,
    simplelog::Config::default(),
    std::fs::File::create(&cfg.log_file)?
).unwrap();
```

příklad logování:

```
log::debug!("Received message: {:?}", msg);
log::info!("User {} joined room {}", name, room);
```

Struktura `Socket` uchovává vektor hodnot `SocketError` reprezentující chyby, které se vyskytly při vstupu/výstupu zásuvky.

Při výskytu chyby v soketu je `UserError` použit k šíření chyby nahoru po zásobníku volání:

```
pub enum UserError {
    Socket(SocketError),
    Protocol(String),
    ...
}

impl Display for UserError {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        match self {
            UserError::Socket(err) => write!(f, "Socket error: {}", err),
            ...
        }
    }
}
```

To umožňuje oddělit problémy - `Socket` zpracovává nízkoúrovňové I/O, zatímco `User` a vyšší vrstvy vidí pouze `UserError`.

Celkově tento přístup poskytuje:

- Konfigurovatelné úrovně protokolu pro různé fáze vývoje
- ošetření chyb zaměřené na příslušnou abstrakční vrstvu

VI. Závěr

Aplikace Fresh demonstruje několik důležitých technik pro vytváření rozšiřitelného kódu:

- Rozdělení kódu do komponent s přesně definovanými rozhraními
- Použití datových struktur
- Použití architektury předávání zpráv klient-server
- Abstrahování nízkoúrovňových detailů za rozumné typy chyb
- Poskytování konfigurovatelných možností pro přizpůsobení chování
- Logování s různými úrovněmi.

Ačkoli je původní verze poměrně jednoduchá, obsahuje základy pro mnoho možných rozšíření a vylepšení.

Cesta od tohoto výchozího bodu k plnohodnotné chatovací aplikaci by byla skvělou zkušeností, která by zahrnovala oblasti jako:

- Distribuované systémy
- Optimalizace výkonu
- Osvědčené postupy zabezpečení
- Škálování na velkou uživatelskou základnu
- Přijetí nových technologií

Nejdůležitější je nyní pokračovat ve zkoumání, učit se z chyb a aplikaci iterativně vylepšovat.

VII. Využité zdroje

Využité zdroje při tvorbě aplikace jsou:

- **directories**: Tento crate poskytuje funkce pro práci se systémovými adresáři.
- **toml**: Slouží pro parsování a serializaci TOML souborů.
- **lazy_static**: Umožňuje vytváření globálních statických dat, které jsou inicializovány jednou.
- **serde**: Generalizovaná knihovna pro (de)serializaci dat v Rustu.
- **serde_json**: Implementuje serializaci a deserializaci JSON dat pomocí serde.
- **log**: Poskytuje sadu maker pro logování.
- **simplelog**: Implementace loggeru, která poskytuje několik konfigurovatelných loggerů.
- **crossterm**: Knihovna umožňující manipulaci s konzolí/terminálem.
- **unicode-normalization**: Poskytuje funkce pro Unicode normalizaci řetězců.
- **clap**: Knihovna pro analýzu příkazové řádky a generování nápovědy.
- **smallvec**: Nabízí datový typ SmallVec, který je optimalizován pro malé velikosti.

Dále byly použity standardní knihovny jazyka Rust a dokumentace jednotlivých balíčků