Инженерный вестник Дона, №12 (2024)
ivdon.ru/ru/magazine/archive/n12y2024/9701

О разработке безопасных приложений на основе интеграции языка программирования Rust и СУБД PostgreSQL

С.Г. Ермаков[1], А.Д. Хомоненко[12], А. Шефнер[1], В.Е. Ляпунов[1], Н.Д. Кот[1], Х.А. Ахмедов[1]

[1]Петербургский государственный университет путей сообщения
[2]Военно-космическая академия им. А.Ф. Можайского, Санкт-Петербург

Аннотация: В настоящее время ключевыми аспектами разработки программного обеспечения являются безопасность и эффективность создаваемых приложений. Особое внимание уделяется безопасности данных и операций с базами данных. В статье освещаются методы и техники разработки безопасных приложений на основе интеграции языка программирования Rust и системы управления базами данных (СУБД) PostgreSQL. Рассматриваются ключевые концепции Rust, такие, как строгая типизация, программная идиома (шаблон) RAII, макроопределения, неизменяемость, и как они способствуют созданию надежных и производительных приложений при работе с базой данных. Рассматривается, как интеграция с PostgreSQL позволяет эффективно управлять данными при высоком уровне безопасности, исключая распространенные ошибки и уязвимости. Практический пример демонстрирует преимущества использования Rust в сочетании с PostgreSQL для создания системы управления расписаниями занятий, обеспечивая целостность и безопасность данных.
Ключевые слова: язык программирования Rust, безопасность памяти, RAII, метапрограммирование. СУБД, PostgreSQL

Введение

В современном мире разработки программного обеспечения (далее ПО) безопасность приложений выходит на первый план. Программисты и разработчики постоянно ищут новые методы и подходы для создания надёжных и защищённых систем. Одним из перспективных направлений в области безопасности является интеграция современных технологий и инструментов разработки. Среди прочего, язык программирования высокого уровня Rust [1] и система управления базами данных (далее СУБД) PostgreSQL [2] выделяются своими возможностями в построении надёжных и эффективных приложений [3].

Далее рассмотрено, как синергия двух мощных инструментов может способствовать созданию безопасного ПО. Rust – компилируемый язык

программирования общего назначения, отличающийся высокой эффективностью и мощным статическим анализатором, предотвращающий большинство ошибок управления памятью. PostgreSQL, в свою очередь, является самой продвинутой СУБД с открытым исходным кодом, предлагающей мощные функции для обеспечения целостности данных и надёжности хранения данных.

Интеграция Rust и PostgreSQL может открыть новые горизонты для разработки приложения, где безопасность и производительность не являются взаимоисключающими. В статье рассматриваются технические аспекты такой интеграции и реальные сценарии использования этих программных средств.

Общая характеристика Rust

Как сказано ранее, Rust – компилируемый язык со статической типизацией. Именно это и позволяет ему быть безопасным и эффективным. Далее рассмотрены конкретные примеры функций, позволяющие повысить безопасность и эффективность приложений.

1. Неизменяемость по умолчанию. При объявлении локальных переменных, они приобретают статус неизменяемых – это означает, что после присвоения переменной начального значения, ей больше нельзя присвоить новое значение или получить изменяемую ссылку на переменную. Это позволяет сделать сложные цепочки кода более простыми к пониманию – нужно следить лишь за теми переменными, которые объявлены как изменяемые [4].

```
1    fn main() {
2        let mut s = String::from("Hello");
3        let num = 5;
4        let vec = vec![1, 2, 3];
5
6        // Ошибки нет: s объявлена с ключевым словом mut
```

```
        s = String::from("World");
        // Ошибка: нельзя присвоить значение дважды
неизменяемой переменной
        num = 10;
        // Ошибка: нельзя заимствовать как изменяемое
        vec[0] = 5;
    }
```

2. Ограничение на получение ссылок. Компилятором гарантируется, что единовременно может существовать лишь одна изменяемая ссылка на объект. При этом, если существует одна изменяемая ссылка, единовременно не могут существовать другие неизменяемые ссылки. Это позволяет полностью исключить частые ошибки при разработке многопоточных приложения, например, условия гонки [5]. Если разработчику необходимо изменять ресурс в множестве потоков, он обязан использовать std::sync::Mutex, std::sync::RwLock или похожий тип, предоставляющий безопасный механизм синхронизации доступа к ресурсу посредством идиомы RAII.

```
fn main() {
    let mut s = String::from("Very long string");

    {
        // Ошибки нет: допускается получение множества
неизменяемых ссылок
        let s2 = &s;
        let s3 = &s;

        // Ошибка: нельзя заимствовать s как
изменяемое, потому что оно также заимствовано как неизменяемое
        let s_mut = &mut s;
    }

    {
```

```
14              // Ошибка: нельзя заимствовать s как изменяемое
более чем один раз одновременно
15              let s_mut1 = &mut s;
16              let s_mut2 = &mut s;
17          }
18      }
```

3. RAII – resource acquisition is initialization (рус. получение ресурса есть инициализация) [6, 7]. В Rust используется автоматическое управление памятью, при котором ресурс выделяется в момент создания объекта и освобождается в момент выхода переменной из области видимости, которая содержала этот объект. При этом компилятором гарантируется, что все ресурсы будут освобождены в правильном порядке, даже если произошла критическая ошибка и программу необходимо завершить.

```
1       fn main() {
2           let mut s = String::new();
3           {
4               let mut file = File::open("file.txt").unwrap();
5               file.read_to_string(&mut s).unwrap();
6               // Переменная file вышла из области видимости,
файловый ресурс возвращён операционной системе
7           }
8           println!("{s}");
9           // Переменная s вышла из области видимости, строка
освобождена
10      }
```

4. Времена жизни. Компилятор при сборке программы осуществляет проверку заимствований. В частности, компилятор предотвращает частые ошибки, связанные с висячими указателями – то есть ситуациями, при который ссылка на объект переживает его.

```
1       fn main() {
2           let s_ref: &String;
3           {
4               let s = String::from("Importrant");
```

```
5            // Ошибка: s не живет достаточно долго
6            s_ref = &s;
7        }
8        println!("{s_ref}");
9    }
```

5. Метапрограммирование. В Rust представлен мощный инструмент метапрограммирования – макроопределения. В отличие, к примеру, от препроцессора компиляторов C/C++, макроопределения в Rust позволяют изменять абстрактное дерево синтаксиса программы на этапе компиляции. Пакет proc-macro [8] расширяет эти возможности ещё сильнее, позволяя выполнять какой угодно код на языке Rust на этапе компиляции. Это позволяет проверять гораздо больше потенциальных ошибок и максимизировать эффективность программы для таких действий, как:

1. Сериализация и десериализация типов Rust в различные форматы разметки, такие как JSON, YAML, TOML и CSV с помощью макроопределения пакета serde [9].

```
1    use serde::{Serialize, Deserialize}
2
3    // Использование атрибутного макроса достаточно для
реализации функционала для сериализации и десериализации в множество
различных форматов
4    #[derive(Serialize, Deserialize)]
5    struct Article {
6        title: String,
7        text: String,
8    }
9
10   // Вложенность структур так же возможна
11   #[derive(Serialize, Deserialize)]
12   struct Scientist {
13       name: String,
14       cathedral: String,
15       articles: Vec<Article>,
```

```
16      }
```

2. Пакет askama предоставляет шаблонизатор, который позволяет эффективно и безопасно создавать HTML-страницы и другой текст – в стиле шаблонизатора Jinja, но с предотвращением различных ошибок на этапе компиляции [10].

```
1     use askama::Template;
2
3     #[derive(Template)]
4     #[template(source = "x = {{ x }}!", ext = "txt")]
5     struct ExampleTemplate {
6        x: i32,
7     }
8
9     fn main() {
10       let s = ExampleTemplate{ x: 10 }.render().unwrap();
11       println!("{s}"); // "x = 10!"
12    }
```

3. Взаимодействие с СУБД и преобразование типов Rust в тип хранящихся в базе данных, и наоборот, с помощью пакета sqlx [11]. Корректность SQL-запросов так же проверяется на этапе компиляции – это подробнее будет рассмотрено в разделе интеграции Rust и PostgreSQL.

Влияние использования Rust на разработку мирового ПО

В настоящее время язык Rust используется заметно реже таких популярных языков, как JavaScript, Python, Java, C# и C/C++ [12]. Несмотря на это, ряд крупных компаний и разработчиков видят в этом языке будущее.

Модули, написанные на исследуемом языке программирования, продолжают внедряться в ядра различных операционных систем. Компания Mozilla разрабатывает новый браузерный движок Servo на Rust. Язык также используется в сетевых и облачных приложениях. Исследования показывают, что это положительно влияет на безопасность управления памятью [13-15].

Характеристика применения Rust в реальных проектах так же дана в работах [16, 17].

Интеграция Rust и PostgreSQL

Ранее был упомянут пакет sqlx, предоставляющий функции для безопасного взаимодействия с различными СУБД. Основное отличие sqlx от своих аналогов для других языков программирования – статический анализ SQL-запросов при помощи реальной базы данных. Более того, sqlx эффективно использует асинхронные операции, что делает его не только безопасным, но и эффективным [18].

```
use sqlx::{PgPool, query, query_as};

// sqlx требует асинхронной среды выполнения, в данном случае выбрана tokio
#[tokio::main]
async fn main() {
    let pool = PgPool::connect("<URL базы данных>").await.unwrap();

    // Если в базе данных отсутствует схема users, или в этой схеме не будет колонок id или name, данный запрос вызовет ошибку компиляции. Так же будет вызвана ошибка, если тип id не является целочисленным, а name - текстовым
    let data: Vec<(i32, String)> = query!("SELECT id, name FROM users")
        .fetch_all(&pool).await.unwrap();

    struct User {
        id: i32,
        name: String,
    }

    // Пример привязки пользовательского типа
```

```
        let data = query_as!(User, "SELECT id, name FROM
users")
            .fetch_all(&pool).await.unwrap();
    }
```

Конвертация данных к типам Rust также возможна и не требует дополнительных действий.

Пример из практики: Хранение расписания занятий

В контексте проекта по исследованию диспетчеризации информации хранение и извлечение расписаний занятий из базы данных PostgreSQL играет ключевую роль в бесперебойной работе сервера, осуществляющего диспетчеризацию. Прежде всего, расписания занятий структурируются и хранятся в таблицах базы данных PostgreSQL по чётко определённой схеме, которая содержит такие важные данные, как: названия курсов, преподаватели, время и место проведения занятий. Для организации доступа к базе данных из языка программирования Rust используется sqlx. Для инкапсуляции запросов используется шаблон проектирования [19] «Репозиторий», который позволяет изолировать логику приложений от SQL запросов и сохранить единую структуру хранения данных. Пример реализации этого шаблона проектирования представлен в следующем листинге.

```
use super::{super::DbError, model::TabletInsertData};
use model::entities::schedule::Tablet;
use sqlx::{PgPool, query_file, query_file_as,
query_file_scalar};

// Объявление структуры, содержащей в себе логику
взаимодействия с базой данных
#[derive(Clone)]
pub struct TabletRepository {
    pool: PgPool,
}
```

```
10
11      // Блок реализации структуры TabletRepository,
содержащий в себе связанные функции и методы структуры
12      impl TabletRepository {
13          // Связанная функция-фабрика, создающая объект
структуры
14          pub fn new(pool: PgPool) -> Self {
15              Self { pool }
16          }
17
18          // Метод, возвращающий все записи из схемы tablets.
Возвращает перечисление (enum) Result, так как операция может быть
завершена с ошибкой. В Rust нет исключений, вместо этого ошибки всегда
возвращаются из функций и методов
19          pub async fn get_all(&self) -> Result<Vec<Tablet>,
DbError> {
20              // SQL-запрос вынесен в отдельный файл
queries/get_tablets.sql
21              let tablets = query_file_as!(Tablet,
"queries/get_tablets.sql")
22                  .fetch_all(&self.pool)
23                  // Оператор ? вернёт из функции вариант
перечисления Result - Err в случае, если метод fetch_all вернул ошибку
24                  .await?;
25
26              // Ok - один из вариантов перечисления Result,
содержащий в себе успешно полученное значение
27              Ok(tablets)
28          }
29      }
```

Отметим, что запросы вынесены в отдельные файлы с расширением .sql – это позволяет переиспользовать их в других контекстах помимо Rust приложения. Текст создания схемы базы данных и запроса «get_tablets.sql» представлены далее.

```
1  -- Добавление схемы tablets в схему базы данных
```

```
CREATE TABLE tablets (
    id SERIAL PRIMARY KEY NOT NULL,
    login TEXT NOT NULL,
    pass TEXT NOT NULL,
    room_id INTEGER,
    CONSTRAINT unique_login
        UNIQUE(login),
    CONSTRAINT fk_room
        FOREIGN KEY(room_id) REFERENCES rooms(id)
        ON DELETE CASCADE
)

-- Содержание get_tablets.sql
SELECT
    tablets.id as id,
    login,
    pass,
    rooms.name as room
FROM
    tablets
LEFT JOIN
    rooms
    ON room_id = rooms.id
```

Такое сочетание инструментов позволяет быть уверенным в надёжности сервера диспетчеризации информации – в нём отсутствуют критические уязвимости и ошибки, так как их предотвращает компилятор.

Заключение

Настоящая статья освещает важную роль функций обеспечения безопасности памяти в языке программирования Rust, особенно в контексте операций с базами данных и разработки приложений. Исследуя такие понятия, как приобретение и инициализация ресурсов, времена жизни, метапрограммирование и статический анализ, а также использование

макроопределений «query» и ему подобных, показано, как Rust обеспечивает надёжную основу для безопасной и эффективной работы с данными. Интеграция Rust и PostgreSQL для хранения и поиска расписания занятий показала возможности использования функций безопасности памяти исследуемого языка программирования для поддержания целостности данных и предотвращения уязвимостей, что делает его привлекательным выбором для разработчиков [20, 21].

Несмотря на отмеченные преимущества интеграции, существуют определённые ограничения. В частности, сложность языка программирования Rust и необходимость глубокого понимания специфики использования этого инструмента. Дальнейшие исследования, на наш взгляд, целесообразно продолжить в направлении улучшения инструментов интеграции и разработки практических руководств для разработчиков, стремящихся повысить надёжность и эффективность своих приложений [22].

Благодарности

Настоящая статья выполнена при поддержке Федеральным государственным бюджетным образовательным учреждением высшего образования «Петербургский государственный университет путей сообщения Императора Александра I» инициативных научных работ, выполняемых студенческими научными коллективами.

Литература

1. Клабник Стив, Николс Кэрол. Программирование на Rust. СПб.: Питер, 2021. 592 с. (Серия «Для профессионалов»).

2. Рогов Е. В. PostgreSQL изнутри. М.: ДМК Пресс, 2022. 660 с.

3. Москат Н.А., Станкевич Е.А. Показатели качества информационно-вычислительных систем железнодорожного транспорта // Инженерный вестник Дона. 2013. №3. URL: ivdon.ru/ru/magazine/archive/n3y2013/1789.

4. Coblenz Michael, Sunshine Joshua, Aldrich Jonathan, Myers Brad, Weber Sam, Shull Forrest Exploring language support for immutability // Laura Dillon, Proceedings of the 38th International Conference on Software Engineering. - Austin Texas: Association for Computing Machinery, ACM2016.

5. Афанасьев К. Е., Власенко А. Ю. Семантические ошибки в параллельных программах для систем с распределенной памятью и методы их обнаружения современными средствами отладки // СибСкрипт. 2009. №2. URL: cyberleninka.ru/article/n/semanticheskie-oshibki-v-parallelnyh-programmah-dlya-sistem-s-raspredelennoy-pamyatyu-i-metody-ih-obnaruzheniya-sovremennymi-sredstvami (дата обращения: 18.11.2024).

6. Bjarne Stroustrup. Why doesn't C++ provide a "finally" construct? // Bjarne Stroustrup's homepage URL: stroustrup.com/bs_faq2.html#finally (дата обращения: 18.11.2024).

7. How to Prevent Memory Leaks // AvenueCode URL: blog.avenuecode.com/how-to-prevent-memory-leaks (дата обращения: 18.11.2024).

8. Procedural Macros // The Rust Reference URL: doc.rust-lang.org/reference/procedural-macros.html (дата обращения: 18.11.2024).

9. serde // The Rust community’s crate registry. URL: crates.io/crates/serde (дата обращения: 18.11.2024).

10. askama // The Rust community’s crate registry. URL: crates.io/crates/askama (дата обращения: 18.11.2024).

11. sqlx // The Rust community’s crate registry. URL: crates.io/crates/sqlx (дата обращения: 18.11.2024).

12. Stack Overflow Developer Survey 2024 // Stack Overflow. URL: survey.stackoverflow.co/2024/technology/ (дата обращения: 18.11.2024).

13. Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, Mengwei Xu An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and

Compromise // 2024 USENIX Annual Technical Conference. Santa Clara, CA: USENIX Association, 2024. pp. 425-443.

14. Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, Mengwei Xu Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study // Seventeenth Symposium on Usable Privacy and Security. USENIX Association, 2021. pp. 597-616.

15. Anderson Brian, Bergstrom Lars, Goregaokar Manish, Matthews Josh, McAllister Keegan, Moffitt Jack, Sapin Simon Engineering the servo web browser engine using Rust // Laura Dillon, Willem Visser, Laurie Williams ICSE '16: Proceedings of the 38th International Conference on Software Engineering Companion. Austin Texas: Association for Computing Machinery, 2016. pp. 81-89.

16. Aggarwal I. High Performance Document Store Implementation in Rust. 2021. 38 p. DOI: 10.31979/etd.96kc-fcmu.

17. Bakken M., Soylu A. Chrontext: Portable SPARQL queries over contextualised time series data in industrial settings // Expert Systems With Applications. 2023. Т. 226. С. 120149. DOI: 10.1016/ j.eswa.2023.120149.

18. Акользин Д.Н. Обзор методов повышения производительности программного обеспечения диспетчерского центра // Инженерный вестник Дона. 2014. №2. URL: ivdon.ru/ru/magazine/archive/n2y2014/2394

19. Быстрова Татьяна Юрьевна Архитектура вне времени: идея шаблонов проектирования К. Александера // Академический вестник УралНИИпроект РААСН. 2011. №1. URL: cyberleninka.ru/article/n/arhitektura-vne-vremeni-ideya-shablonov-proektirovaniya-k-aleksandera (дата обращения: 18.11.2024).

20. Кайшев Д.А. Новое поколение языков программирования – Rust // StudNet. Т. 4, № 6, 2021. С. 1333-1337.

21. Lattuada A. et al. Proceedings of the ACM on Programming Languages. 2023. Т. 7. №. OOPSLA1. pp. 286-315.

22. Пугачев С.В., Хомоненко А.Д., Ярмолинский Ф.А. О разработке информационной системы грузоперевозок ОАО "РЖД" на основе безопасной интеграции приложений // Интеллектуальные технологии на транспорте. 2023. № 1 (33). С. 21-26.

References

1. Klabnik Steve, Nichols Carol. Programmirovaniye na Rust. [Programming in Rust]. St. Petersburg: Piter, 2021, 592 p., (Seriya “Dlya professionalov”).

2. Rogov E. V. PostgreSQL iznutri [PostgreSQL from the Inside]. Moscow: DMK Press, 2022. 660 p.

3. Moskat N. A., Stankevich E. A. Inzhenernyj vestnik Dona, 2013, №3, URL: ivdon.ru/en/magazine/archive/n3y2013/1789

4. Coblenz Michael, Sunshine Joshua, Aldrich Jonathan, Myers Brad, Weber Sam, Shull Forrest. 2016. "Exploring language support for immutability. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). Association for Computing Machinery", New York, NY, USA, pp. 736–747. URL: doi.org/10.1145/2884781.2884798

5. Afanas’yev K. E., and Vlasenko A. Yu. SibSkript, no. 2, 2009, pp. 13-20. URL: cyberleninka.ru/article/n/semanticheskie-oshibki-v-parallelnyh-programmah-dlya-sistem-s-raspredelennoy-pamyatyu-i-metody-ih-obnaruzheniya-sovremennymi-sredstvami.

6. Bjarne Stroustrup. Why Doesn't C++ Provide a "Finally" Construct? URL: stroustrup.com/bs_faq2.html#finally (accessed: 18.11.2024).

7. How to Prevent Memory Leaks, URL: blog.avenuecode.com/how-to-prevent-memory-leaks (accessed: 18.11.2024).

8. The Rust Reference: Procedural Macros. URL: doc.rust-lang.org/reference/procedural-macros.html (accessed: 18.11.2024).

9. serde, URL: crates.io/crates/serde (accessed: 18.11.2024).

10. askama, URL: crates.io/crates/askama (accessed: 18.11.2024).

11. sqlx, URL: crates.io/crates/sqlx (accessed: 18.11.2024).

12. Stack Overflow Developer Survey 2024, URL: survey.stackoverflow.co/2024/technology/ (accessed: 18.11.2024).

13. Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, Mengwei Xu. "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise", 2024 USENIX Annual Technical Conference. Santa Clara, CA: USENIX Association, 2024. pp. 425-443.

14. Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, Mengwei Xu. "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study", Seventeenth Symposium on Usable Privacy and Security. USENIX Association, 2021. pp. 597-616.

15. Anderson Brian, Bergstrom Lars, Goregaokar Manish, Matthews Josh, McAllister Keegan, Moffitt Jack, Sapin Simon. "Engineering the Servo Web Browser Engine Using Rust", Laura Dillon, Willem Visser, Laurie Williams. ICSE '16: Proceedings of the 38th International Conference on Software Engineering Companion. Austin, Texas: Association for Computing Machinery, 2016. pp. 81-89.

16. Aggarwal I. High Performance Document Store Implementation in Rust. 2021. 38 pp. DOI: 10.31979/etd.96kc-fcmu.

17. Bakken M., Soylu A. "Chrontext: Portable SPARQL Queries Over Contextualized Time Series Data in Industrial Settings", Expert Systems With Applications. 2023. Vol. 226. Article 120149. DOI: 10.1016/j.eswa.2023.120149.

18. Akolzin. D. N. Inzhenernyj vestnik Dona, 2014, № 2. URL: ivdon.ru/en/magazine/archive/n2y2014/2394

19. Bystrova T. Y. Akademicheskiy vestnik UralNIIproyekt RAASN, no. 1, 2011, pp. 41-46. URL: cyberleninka.ru/article/n/arhitektura-vne-vremeni-ideya-shablonov-proektirovaniya-k-aleksandera.

20. Kayshev D.A. StudNet, V. 4, No. 6, 2021. pp. 1333-1337. URL: stud.net.ru/wp-content/uploads/2021/06/160.pdf

21. Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Proc. ACM Program. Lang. 7, OOPSLA1, Article 85 (April 2023), pp. 286-315. URL: doi.org/10.1145/3586037

22. Pugachev S. V., Khomonenko A. D., and Yarmolinskiy F. A. Intellectual Technologies on Transport, no. 1 (33), 2023, pp. 21-26. doi:10.24412/2413-2527-2023-133-21-26

Дата поступления: 27.10.2024

Дата публикации: 12.12.2024