Engineer

Rust

コンパイル時にメモリーの扱いを検査する言語

どのようなものか

Rustは、従来CやC++が担ってきた分野のためのプログラミング言語です。その分野では、プログラムが自らメモリーを管理し、他の言語で書いたソフトウェアにリンクすることもよくあります。コンパイラーは、実行前にネイティブの機械語を生成します。バージョン1.0は2015年5月15日に発表され、ガベージコレクターもランタイムも使わずに、性能を低いレベルで制御する力と安全性の保証を両立する言語とされました。そのため、Rustのライブラリーを Cのものの代わりに使えます。

メモリーは、コンパイラーが検査する規則の集合である所有権を通じて管理します。各値には、一度に一つの所有者があり、所有者がスコープを出ると、値が破棄され、メモリーを解放します。その地点は、ガベージコレクターが次に調べるときではなく、コンパイル時に決まります。所有権を取らずに値を必要とするコードは、参照を借用します。借用の規則は、どの時点でも、変更可能な参照を一つ、または変更不可能な参照を任意の数だけ認め、すべての参照が有効であることを求めます。規則に反するプログラムはコンパイルできません。同じ規則はデータ競合も排除するため、同期せずに二つのスレッドが同じデータを書き換えると、不定期に起こる障害ではなく、コンパイルエラーになります。

Rustには例外がありません。想定内の形で失敗し得る関数は、値かエラーのどちらかを含む `Result` を返します。呼び出し側が結果を使わずに放置すると、コンパイラーは既定で警告を出します。配列の末尾を越えて読むなど、プログラムが復旧できない不具合は、代わりに `panic!` マクロを通ります。

静的解析は保守的なので、実際には問題なく動くプログラムを拒否することもあります。また、ハードウェアの操作や他の言語の呼び出しなど、まったく検査できない処理もあります。そのために `unsafe` があります。生のポインターの参照外し、安全でない関数の呼び出しなど、追加の五つの操作を可能にします。既定では、外部のC関数として宣言したものもすべて、安全でない関数に含まれま

す。借用チェッカーを無効にするものではありません。コンパイラーではなく、プログラマーがメモリーの安全性を引き受ける行を示すため、これらの操作の安全性は、周囲の安全なコードも含め、プログラマーがその契約を守ることに依存します。Rustの公式書籍は、そのブロックを小さくし、安全なインターフェースで包むよう勧めています。標準ライブラリーの一部も、この方法で作られています。

新しいコンパイラーは6週間ごとに公開されます。既存のコードを壊す変更は、約3年ごとのエディションまで待ちます。エディションは2015、2018、2021、2024の四つで、最新のものは2025年2月20日、Rust 1.85.0とともに安定版になりました。各パッケージは `Cargo.toml` でエディションを指定します。すべてのコンパイラーは、それより前のすべてのエディションをビルドし、異なるエディションのパッケージ同士もリンクできるため、プロジェクトは、自分で時期を選んで移行できます。依存関係の管理とビルドの道具であるCargoも言語に付属し、rustfmtがコードのスタイルを統一します。外部のライブラリーは、クレートと呼ぶパッケージとして取得します。

概念を判断に生かす

「Rustとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 データの保存先、処理のルール、他のシステムとの接続方法など、利用者の目に見えない部分をつくります。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 ETL/ELTパイプライン6本を設計・運用 — 6 3種類のDBエンジンにまたがるETLパイプライン

ETL/ELTパイプライン6本を設計・運用 お問い合わせ →

なぜ重要なのか

CやC++では、解放済みのメモリーを指すポインターを使ったり、二つのスレッドが同じデータを書き換えたりしても、コンパイルは警告なく通り、後になって、不定期に、しばしば原因から遠い場所で失敗します。Rustは、そのようなプログラムをコンパイルエラーにします。検査はビルド時に行うため、完成したプログラムに、その検査の負担はありません。

ガベージコレクターも同梱するランタイムも不要なので、Rustのコードは、他の言語のソフトウェアの下で動かせます。Cを呼べるものなら、Cインターフェース経由でRustの中核を呼べるため、十分にテストした一つの実装を、複数のプラットフォームのクライアントで使えます。Rustの公式書籍は、実運用での用途として、コマンドラインの道具、ウェブサービス、組み込み機器、Firefoxブラウザーの主要部分を挙げています。

その代わり、開発の初めに手間がかかります。コンパイラーが安全だと判断できる形でコードを書く必要があり、参照を自由に渡す代わりに、誰が値を所有するかを決めなければならないことも多くあります。公式書籍自身も、所有権に慣れるには時間がかかると述べています。安全性を検証できず、コンパイラーに拒否されたときは、通常、検査を回避するのではなく、構造を変えることで対処します。

こうした負担と引き換えに得られる、もう一つの利点が安定性です。1.0以降、互換性を壊す変更は大幅に排除され、古いコードを壊し得る変更は、パッケージが選んで移行するエディションを通じてだけ導入されます。

実際の活用例

実績にあるネイティブのmacOSメッセージングクライアントでは、その下の中核をRustが担っています。十分にテストしたプロトコルの中核をRustで書き、Cインターフェースを公開しています。11,141行のSwiftを53ファイルに分けたクライアントは、固定した上流リビジョンからコンパイルした静的アーカイブとして、それをリンクします。固定が重要なのは、Cインターフェースが契約だからです。中核のバージョンが変わると、Swiftを一行も変えなくても契約が変わります。Cインターフェースが返すすべてに所有権の規則があり、一つ間違えるだけで、コンパイラーの警告なしに、リークやクラッシュになります。そのため、薄いラッパーが、境界を渡るすべてのポインターと文字列を所有し、上の層には生のポインターを見せません。

その中核をアプリケーションにリンクすると、依存するものもすべて配布することになります。それを扱うのが、644のRustクレートのライセンス監査です。各クレートはライセンスを式で宣言します。監査は、毎回のビルドで依存関係グラフ全体の各式を許可リストと照合し、ORをライセンス間の選択、ANDを両方に従う義務として読みます。一致しないものがあれば、ビルドは失敗します。一度も失敗したことのない検査は、失敗できない検査と見分けが付かないため、意図して二度失敗させました。一度はクレートのライセンスをリストが拒否するものに書き換え、もう一度は監査を生成し直さず、固定した中核のリビジョンを変えました。読むのは宣言されたメタデータだけであり、法的助言ではありません。

このサイト自身にはRustのソースはありません。この言語との接点は、Rustで書かれ、コミットで変更するPythonスクリプトをすべて検査するRuffを介した間接的なものです。

問題が起きるところ

Rustの問題の大半は、コンパイラーの検査が及ぶ範囲の端で起こります。最適化したビルドの算術、他の言語との境界、依存関係から入るコードです。

リリースビルドで算術が折り返す。 テストでオーバーフローしてパニックになったカウンターが、本番では黙って小さな数に変わります。デバッグビルドでは、Rustは整数の算術のオーバーフローを検査し、検出するとパニックにします。`--release` のビルドは検査を省き、折り返すため、256は `u8` では0になります。公式書籍は、その折り返しに依存することを誤りとしています。オーバーフローし得る値では、動作を明示的に選びます。`checked_*` はオーバーフロー時に `None` を返し、`saturating_*` は限界値で止まり、`wrapping_*` は意図して折り返します。

```
fn add_stock(on_hand: u8, delivered: u8) -> Option<u8> {
    // None when the sum does not fit, in debug and release alike
    on_hand.checked_add(delivered)
}
```

Cとの境界で、メモリーの不具合が戻ってくる。 Rust側は警告一つなくビルドできても、他の言語の呼び出しの中でリークしたり、クラッシュしたりします。`extern` ブロックで宣言した関数の呼び出しが安全でないのは、相手の言語がRustの規則を何も強制しないためです。各ポインターを誰が解放するかは、コンパイラーには見えない慣例になります。境界を渡るすべてのポインターと文字列を一つのラッパーに管理させ、各呼び出しの横に所有権の規則を書き、その上には安全なインターフェースだけを公開します。

パニックがCに達すると、プロセスが中断する。 Cから呼ばれたRustの関数でパニックが起こり、エラーを返す代わりにプログラム全体が停止します。Rust 1.81（2024年9月5日公開）以降、通常の `"C"` インターフェースで宣言した関数から、パニックが抜けようとすると、プロセスを中断します。境界に届く前に、失敗を戻り値へ変換します。関数を `"C-unwind"` で宣言するのは、相手側のコードがスタックの巻き戻しを想定する場所だけにします。

通常の更新で、ビルド全体のライセンスが変わる。 依存関係の三段先のバージョン更新で、製品では採用できないライセンスのクレートが入り、ビルドは何も知らせません。各クレートが独自のライセンスを宣言し、義務はグラフ全体で積

み重なります。毎回のビルドで、すべてのクレートの式を許可リストと照合し、一致しないものがあれば失敗させます。また、一度は意図して検査を失敗させ、動作することを確かめます。

参考資料

- The Rust Programming Language — 導入から所有権、エラー処理、安全でないコードまでを扱う公式書籍です。
- 所有権とは — 所有権の三つの規則と、ガベージコレクターなしでメモリーを解放する仕組みです。
- Unsafe Rust — 五つの安全でない操作、生のポインター、 `extern` によるCの呼び出しを説明します。
- マニフェストの形式 — `edition` 、 `rust-version` 、 `license` という、 `Cargo.toml` のフィールドを説明します。
- Rust 1.0の発表 — 当初の目標と、1.0から始まる安定性の約束です。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「Rustとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/rust/