Engineer

Rust

Lenguaje compilado que comprueba el uso de la memoria

Qué es

Rust es un lenguaje de programación para el trabajo que tradicionalmente se hace en C y C++, donde el programa gestiona su propia memoria y a menudo se enlaza con software escrito en otros lenguajes. Su compilador genera código máquina nativo antes de la ejecución. La versión 1.0 se anunció el 15 de mayo de 2015 como un lenguaje que combina el control de bajo nivel sobre el rendimiento con garantías de seguridad, sin recolector de basura ni entorno de ejecución, lo que permite que una biblioteca de Rust sustituya a una de C.

La memoria se gestiona mediante la propiedad, un conjunto de reglas que comprueba el compilador. Cada valor tiene un solo propietario a la vez. Cuando el propietario sale de su ámbito, se descarta el valor y se libera su memoria, en un momento fijado al compilar el programa, no cuando un recolector vuelva a examinarla. El código que necesita un valor sin apropiárselo toma prestada una referencia. Las reglas de préstamo permiten, en cualquier momento, una referencia mutable o cualquier número de referencias inmutables, y exigen que todas sean válidas. Un programa que incumple una regla no compila. Las mismas reglas excluyen las carreras de datos: dos hilos que escriben sobre los mismos datos sin sincronización provocan un error de compilación, en lugar de un fallo intermitente.

Rust no tiene excepciones. Una función que puede fallar de una forma prevista devuelve un `Result`, que contiene un valor o un error. El compilador avisa de forma predeterminada si quien la llama deja el resultado sin usar. En cambio, un defecto del que el programa no puede recuperarse, como leer más allá del final de un array, pasa por la macro `panic!`.

El análisis estático es conservador: el compilador rechaza algunos programas que habrían funcionado bien, y hay tareas, como controlar hardware o llamar a otro lenguaje, que no puede comprobar en absoluto. Para esos casos existe `unsafe`, que habilita cinco capacidades adicionales, entre ellas desreferenciar un puntero sin procesar y llamar a una función no segura, lo que de forma predeterminada incluye cualquier función declarada desde C. No desactiva el comprobador de préstamos. Marca las líneas en las que el programador, en lugar del compilador, responde por la seguridad de la memoria, por lo que la seguridad de esas operaciones depende de que el programador respete sus contratos, también en el código seguro que las rodea. El libro de Rust aconseja mantener esos bloques pequeños y envolverlos en una interfaz segura, como se hace con partes de la biblioteca estándar.

Cada seis semanas se publica un compilador nuevo. Los cambios que romperían código existente esperan a una edición, que llega aproximadamente cada tres años. Hay cuatro: 2015, 2018, 2021 y 2024. La más reciente pasó a ser estable el 20 de febrero de 2025 con Rust 1.85.0. Cada paquete indica su edición en `Cargo.toml`, cada compilador compila todas las ediciones anteriores y los paquetes de distintas ediciones se enlazan entre sí, por lo que cada proyecto avanza cuando lo decide. Cargo, el gestor de dependencias y herramienta de compilación, viene incluido con el lenguaje; rustfmt mantiene el estilo uniforme, y las bibliotecas de terceros llegan como paquetes llamados crates.

Utilice el concepto en una decisión

Aplique «¿Qué es Rust?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. La parte que el usuario no ve: dónde se guardan los datos, qué reglas se aplican y cómo pueden comunicarse otros sistemas con ella.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Diseño y operación de 6 flujos ETL/ELT — 6 flujos ETL en tres motores de bases de datos

Diseño y operación de 6 flujos ETL/ELT Póngase en contacto →

Por qué importa

En C y C++, el código que usa un puntero después de liberar su memoria o escribe los mismos datos desde dos hilos compila sin queja y falla después, de forma intermitente y a menudo lejos de la causa. En Rust, esos programas producen errores de compilación. Como las comprobaciones se hacen al compilar, el programa terminado no paga ningún coste por ellas.

Sin recolector de basura ni entorno de ejecución que incorporar, el código Rust puede servir de base a software de otros lenguajes. Cualquier software capaz de llamar a C puede llamar a un núcleo de Rust mediante una interfaz de C, de modo que una implementación bien probada puede atender a clientes en varias plataformas. El libro de Rust enumera herramientas de línea de comandos, servicios web, dispositivos embebidos y partes importantes del navegador Firefox entre sus usos en producción.

El coste se paga por adelantado. Hay que estructurar el código para que el compilador pueda ver que es seguro, lo que a menudo implica decidir quién es dueño de un valor en lugar de pasar referencias libremente. El propio libro reconoce que acostumbrarse a la propiedad lleva tiempo. Cuando el compilador rechaza un programa que no puede verificar, la solución suele ser una estructura distinta, no una forma de eludir la comprobación.

La estabilidad es la otra parte del trato. Desde la versión 1.0, se han descartado en gran medida los cambios incompatibles, por lo que un cambio capaz de romper código antiguo solo llega a través de una edición que el paquete adopta expresamente.

Cómo se aplica en la práctica

En el portafolio, Rust es el núcleo sobre el que se construye un cliente nativo de mensajería para macOS. El núcleo del protocolo, bien probado, está escrito en Rust y expone una interfaz de C. El cliente, con 11.141 líneas de Swift repartidas en 53 archivos, lo enlaza como una biblioteca estática compilada desde una revisión fijada del proyecto de origen. Es importante fijar esa revisión porque la interfaz de C es el contrato: un núcleo que cambia modifica ese contrato sin cambiar una línea de Swift. Todo lo que devuelve esa interfaz lleva una regla de propiedad. Incumplir una produce una fuga de memoria o un cierre inesperado, sin advertencia del compilador de por medio. Por eso, una capa fina de adaptación se hace cargo de cada puntero y cada cadena que cruza el límite, y ninguna capa superior ve un puntero sin procesar.

Enlazar ese núcleo en una aplicación implica distribuir todo aquello de lo que depende. De eso trata una auditoría de licencias de 644 crates de Rust. Cada crate declara su licencia como una expresión. En cada compilación, la auditoría comprueba todas las expresiones del grafo completo de dependencias frente a una lista de licencias permitidas, interpretando OR como una elección entre licencias y AND como la obligación de cumplir ambas. Cualquier expresión que no

se ajuste hace fallar la compilación. Como una comprobación que nunca ha fallado parece igual que otra incapaz de fallar, se provocó su fallo dos veces deliberadamente: una reescribiendo la licencia de un crate con un término que la lista rechaza, y otra cambiando la revisión fijada del núcleo sin regenerar la auditoría. Solo lee los metadatos declarados y no constituye asesoramiento jurídico.

Este sitio no tiene código Rust propio. Se encuentra con el lenguaje de forma indirecta, a través de Ruff, que está escrito en Rust y comprueba todos los scripts Python que toca un commit.

Dónde surgen los problemas

La mayoría de los problemas con Rust está en los límites de lo que comprueba el compilador: la aritmética en una compilación optimizada, la frontera con otro lenguaje y el código que llega con una dependencia.

La aritmética da la vuelta en una compilación para distribución. Un contador que durante las pruebas provocaba un pánico por desbordamiento se convierte, sin avisar, en un número pequeño en producción. En las compilaciones de depuración, Rust comprueba si la aritmética de enteros se desborda y provoca un pánico si detecta un desbordamiento. Una compilación con `--release` omite la comprobación y vuelve al principio del rango, por lo que 256 en un `u8` se convierte en 0. El libro considera un error depender de ese comportamiento. Donde un valor pueda desbordarse, elija explícitamente qué debe ocurrir: `checked_*` devuelve `None` si hay desbordamiento, `saturating_*` se detiene en el límite y `wrapping_*` da la vuelta de forma deliberada.

```
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)
}
```

Los errores de memoria vuelven en la frontera con C. Un programa cuya parte de Rust compila sin una sola advertencia presenta fugas de memoria o sufre un cierre inesperado dentro de una llamada externa. No es seguro llamar a las funciones declaradas en un bloque `extern`, porque el otro lenguaje no impone ninguna regla de Rust. Por eso, quién libera cada puntero es una convención que el compilador no puede ver. Asigne a una sola capa de adaptación la

responsabilidad sobre cada puntero y cadena que cruza el límite, escriba cada regla de propiedad junto a la llamada y exponga solo una interfaz segura hacia las capas superiores.

Un pánico alcanza C y el proceso aborta. Una función de Rust llamada desde C entra en pánico y, en lugar de devolver un error, se detiene todo el programa. Desde Rust 1.81, publicado el 5 de septiembre de 2024, un pánico que intenta salir de una función declarada con la interfaz normal `"C"` aborta el proceso. Convierta los fallos en valores de retorno antes de llegar al límite y declare una función con `"C-unwind"` solo donde el código del otro lado espere que se desenrolle la pila.

Una actualización rutinaria cambia la licencia de toda la compilación. Un cambio de versión tres niveles más abajo incorpora un crate cuya licencia el producto no puede aceptar, y nada en la compilación lo señala. Cada crate declara su propia licencia y las obligaciones se acumulan en todo el grafo. Compruebe la expresión de cada crate frente a una lista de licencias permitidas en cada compilación, haga que cualquier caso no admitido la haga fallar y provoque una vez el fallo de la comprobación para demostrar que se activa.

Lecturas adicionales

- El lenguaje de programación Rust — El libro oficial, desde la instalación hasta la propiedad, la gestión de errores y el código no seguro.
- ¿Qué es la propiedad? — Las tres reglas de propiedad y cómo se libera la memoria sin recolector de basura.
- Rust no seguro — Las cinco capacidades no seguras, los punteros sin procesar y las llamadas a C mediante `extern`.
- El formato del manifiesto — Los campos `edition`, `rust-version` y `license` de `Cargo.toml`.
- Anuncio de Rust 1.0 — Los objetivos originales y el compromiso de estabilidad que comenzó con 1.0.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es Rust?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

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

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/rust/