Chapitre 23. Fonctions étrangères

> *Cyberespace. Complexité impensable. Les lignes de lumière variaient dans le non-espace de l'esprit, les amas et les constellations de données. Comme les lumières de la ville, reculant...*
>
> —William Gibson, *neuromancien*

Tragiquement, tous les programmes dans le monde ne sont pas écrits en Rust. Il existe de nombreuses bibliothèques et interfaces critiques implémentées dans d'autres langages que nous aimerions pouvoir utiliser dans nos programmes Rust. *L'interface de fonction étrangère* (FFI) de Rust permet au code Rust d'appeler des fonctions écrites en C et, dans certains cas, en C++. Étant donné que la plupart des systèmes d'exploitation offrent des interfaces C, l'interface de fonction étrangère de Rust permet un accès immédiat à toutes sortes d'installations de bas niveau.

Dans ce chapitre, nous allons écrire un programme lié à , une bibliothèque C pour travailler avec le système de contrôle de version Git. Tout d'abord, nous allons montrer ce que c'est que d'utiliser les fonctions C directement à partir de Rust, en utilisant les fonctionnalités dangereuses démontrées dans le chapitre précédent. Ensuite, nous montrerons comment construire une interface sûre pour , en s'inspirant de la caisse open source, qui fait exactement cela. `libgit2 libgit2 git2-rs`

Nous supposerons que vous êtes familier avec C et la mécanique de compilation et de liaison de programmes C. L'utilisation de C++ est similaire. Nous supposerons également que vous êtes un peu familier avec le système de contrôle de version Git.

Il existe des caisses Rust pour communiquer avec de nombreux autres langages, y compris Python, JavaScript, Lua et Java. Nous n'avons pas de place pour les couvrir ici, mais en fin de compte, toutes ces interfaces sont construites à l'aide de l'interface de fonction étrangère C, donc ce chapitre devrait vous donner une longueur d'avance, quelle que soit la langue avec laquelle vous devez travailler.

Recherche de représentations de données communes

Le dénominateur commun de Rust et C est le langage machine, donc pour anticiper à quoi ressemblent les valeurs Rust du code C, ou vice versa,

vous devez prendre en compte leurs représentations au niveau de la machine. Tout au long du livre, nous avons tenu à montrer comment les valeurs sont réellement représentées en mémoire, vous avez donc probablement remarqué que les mondes de données de C et Rust ont beaucoup en commun: un Rust et un C sont identiques, par exemple, et les structs sont fondamentalement la même idée dans les deux langues. Pour établir une correspondance entre les types Rust et C, nous allons commencer par les primitives, puis nous nous frayer un chemin jusqu'à des types plus compliqués. `usize size_t`

Compte tenu de son utilisation principale en tant que langage de programmation de systèmes, C a toujours été étonnamment lâche sur les représentations de ses types: an est généralement long de 32 bits, mais pourrait être plus long, ou aussi court que 16 bits; a C peut être signé ou non signé; et ainsi de suite. Pour faire face à cette variabilité, le module de Rust définit un ensemble de types rust qui sont garantis d'avoir la même représentation que certains types C (tableau 23-1). Ceux-ci couvrent les types d'entiers et de caractères
primitifs. `int char std::os::raw`

Tableau 23-1. types dans Rust `std::os::raw`

Type C	**Correspondant** `std::os::raw type`
`short`	`c_short`
`int`	`c_int`
`long`	`c_long`
`long long`	`c_longlong`
`unsigned short`	`c_ushort`
`unsigned, unsigned int`	`c_uint`
`unsigned long`	`c_ulong`
`unsigned long long`	`c_ulonglong`
`char`	`c_char`
`signed char`	`c_schar`
`unsigned char`	`c_uchar`
`float`	`c_float`
`double`	`c_double`
`void *, const void *`	`*mut c_void, *const c_void`

Quelques remarques sur le tableau 23-1 :

- À l'exception de , tous les types rust ici sont des alias pour certains types rust primitifs : , par exemple, est soit ou . `c_void c_char i8 u8`
- Un Rust est équivalent à un C ou C++ . `bool bool`
- Le type 32 bits de Rust n'est pas l'analogue de , dont la largeur et l'encodage varient d'une implémentation à l'autre. Le type de C est plus proche, mais son encodage n'est toujours pas garanti d'être Unicode. `char wchar_t char32_t`
- Les primitives et les types de Rust ont les mêmes représentations que les C et . `usize isize size_t ptrdiff_t`

- Les pointeurs C et C++ et les références C++ correspondent aux types de pointeurs bruts de Rust, et . `*mut T *const T`
- Techniquement, la norme C permet aux implémentations d'utiliser des représentations pour lesquelles Rust n'a pas de type correspondant : entiers 36 bits, représentations de signe et de magnitude pour les valeurs signées, etc. En pratique, sur chaque plate-forme sur laquelle Rust a été porté, chaque type d'entier C commun a une correspondance dans Rust.

Pour définir des types rust struct compatibles avec les structS C, vous pouvez utiliser l'attribut. Placer au-dessus d'une définition struct demande à Rust de disposer les champs de la struct en mémoire de la même manière qu'un compilateur C présenterait le type C struct analogue. Par exemple, le fichier d'en-tête *git2/errors.h* de devient définit la structure C suivante pour fournir des détails sur une erreur précédemment signalée : `#[repr(C)] #[repr(C)] libgit2`

```
typedef struct {
    char *message;
    int klass;
} git_error;
```

Vous pouvez définir un type Rust avec une représentation identique comme suit :

```
use std::os::raw::{c_char, c_int};

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}
```

L'attribut affecte uniquement la disposition de la structure elle-même, pas les représentations de ses champs individuels, donc pour correspondre à la structure C, chaque champ doit également utiliser le type C: pour , pour , et ainsi de suite. `#[repr(C)] *const c_char char * c_int int`

Dans ce cas particulier, l'attribut ne modifie probablement pas la disposition de . Il n'y a vraiment pas beaucoup de façons intéressantes de disposer un pointeur et un entier. Mais alors que C et C++ garantissent que les membres d'une structure apparaissent en mémoire dans l'ordre dans lequel ils sont déclarés, chacun à une adresse distincte, Rust réorganise

les champs pour minimiser la taille globale de la structure, et les types de taille zéro ne prennent pas de place. L'attribut indique à Rust de suivre les règles de C pour le type donné. `#[repr(C)] git_error #[repr(C)]`

Vous pouvez également utiliser pour contrôler la représentation des énumérations de style C : `#[repr(C)]`

```
#[repr(C)]
#[allow(non_camel_case_types)]
enum git_error_code {
    GIT_OK          =  0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
}
```

Normalement, Rust joue à toutes sortes de jeux lorsqu'il choisit comment représenter les enums. Par exemple, nous avons mentionné l'astuce que Rust utilise pour stocker en un seul mot (si est dimensionné). Sans , Rust utiliserait un seul octet pour représenter l'enum ; avec , Rust utilise une valeur de la taille d'un C , tout comme C. `Option<&T> T # [repr(C)] git_error_code #[repr(C)] int`

Vous pouvez également demander à Rust de donner à un enum la même représentation qu'un type entier. Commencer la définition précédente par vous donnerait un type 16 bits avec la même représentation que l'énumération C++ suivante : `#[repr(i16)]`

```
#include <stdint.h>

enum git_error_code: int16_t {
    GIT_OK          =  0,
    GIT_ERROR       = -1,
    GIT_ENOTFOUND   = -3,
    GIT_EEXISTS     = -4,
    ...
};
```

Comme nous l'avons mentionné précédemment, cela s'applique également aux syndicats. Les champs d'unions commencent toujours au premier bit de la mémoire de l'union, c'est-à-dire l'index 0. `#[repr(C)] # [repr(C)]`

Supposons que vous ayez une structure C qui utilise une union pour contenir des données et une valeur de balise pour indiquer quel champ de

l'union doit être utilisé, semblable à un enum de Rust.

```
enum tag {
    FLOAT = 0,
    INT   = 1,
};

union number {
    float f;
    short i;
};

struct tagged_number {
    tag t;
    number n;
};
```

Le code Rust peut interagir avec cette structure en s'appliquant aux types enum, structure et union, et en utilisant une instruction qui sélectionne un champ d'union dans une structure plus grande en fonction de la balise :`#[repr(C)]` `match`

```
#[repr(C)]
enum Tag {
    Float = 0,
    Int = 1
}

#[repr(C)]
union FloatOrInt {
    f: f32,
    i: i32,
}

#[repr(C)]
struct Value {
    tag: Tag,
    union: FloatOrInt
}

fn is_zero(v: Value) -> bool {
    use self::Tag::*;
    unsafe {
        match v {
            Value { tag: Int, union: FloatOrInt { i: 0 } } => true,
            Value { tag: Float, union: FloatOrInt { f: num } } => (num ==
            _ => false
        }
```

```
        }
    }
```

Même les structures complexes peuvent être facilement utilisées à travers la frontière FFI en utilisant ce type de technique.

Passer des cordes entre Rust et C est un peu plus difficile. C représente une chaîne sous forme de pointeur vers un tableau de caractères, terminé par un caractère nul. Rust, d'autre part, stocke explicitement la longueur d'une chaîne, soit en tant que champ de a, soit en tant que deuxième mot d'une référence grasse. Les chaînes rust ne sont pas terminées par une valeur NULL ; en fait, ils peuvent inclure des caractères nuls dans leur contenu, comme tout autre caractère. `String &str`

Cela signifie que vous ne pouvez pas emprunter une chaîne Rust en tant que chaîne C : si vous passez un pointeur de code C dans une chaîne Rust, il pourrait confondre un caractère null incorporé avec la fin de la chaîne ou s'exécuter à la fin à la recherche d'un null de terminaison qui n'est pas là. En allant dans l'autre sens, vous pourrez peut-être emprunter une chaîne C comme un Rust , tant que son contenu est bien formé UTF-8. `&str`

Cette situation oblige effectivement Rust à traiter les chaînes C comme des types entièrement distincts de et . Dans le module, les types et représentent des tableaux d'octets détenus et empruntés à terminaison NULL. Par rapport à et , les méthodes sur et sont assez limitées, limitées à la construction et à la conversion à d'autres types. Nous montrerons ces types en action dans la section suivante. `String &str std::ffi CString CStr String str CString CStr`

Déclaration de fonctions et de variables étrangères

Un bloc déclare des fonctions ou des variables définies dans une autre bibliothèque avec laquelle l'exécutable Rust final sera lié. Par exemple, sur la plupart des plates-formes, chaque programme Rust est lié à la bibliothèque C standard, de sorte que nous pouvons informer Rust de la fonction de la bibliothèque C comme ceci: `extern strlen`

```
use std::os::raw::c_char;

extern {
```

```
        fn strlen(s: *const c_char) -> usize;
    }
```

Cela donne à Rust le nom et le type de la fonction, tout en laissant la définition être liée plus tard.

Rust suppose que les fonctions déclarées à l'intérieur des blocs utilisent des conventions C pour passer des arguments et accepter des valeurs de retour. Ils sont définis comme des fonctions. Ce sont les bons choix pour : il s'agit bien d'une fonction C, et sa spécification en C nécessite que vous lui passiez un pointeur valide vers une chaîne correctement terminée, qui est un contrat que Rust ne peut pas appliquer. (Presque toutes les fonctions qui prennent un pointeur brut doivent être : rust sûr peut construire des pointeurs bruts à partir d'entiers arbitraires, et le déréférencement d'un tel pointeur serait un comportement indéfini.) `extern unsafe strlen unsafe`

Avec ce bloc, nous pouvons appeler comme n'importe quelle autre fonction Rust, bien que son type le donne en tant que touriste: `extern strlen`

```
use std::ffi::CString;

let rust_str = "I'll be back";
let null_terminated = CString::new(rust_str).unwrap();
unsafe {
    assert_eq!(strlen(null_terminated.as_ptr()), 12);
}
```

La fonction génère une chaîne C terminée par une valeur NULL. Il vérifie d'abord son argument pour les caractères nuls incorporés, car ceux-ci ne peuvent pas être représentés dans une chaîne C, et renvoie une erreur s'il en trouve (d'où la nécessité du résultat). Sinon, il ajoute un octet nul à la fin et renvoie une propriété des caractères résultants. `CString::new unwrap CString`

Le coût de dépend du type de passe-t-il. Il accepte tout ce qui implémente . La transmission d'un implique une allocation et une copie, car la conversion en construit une copie allouée au tas de la chaîne pour que le vecteur soit propriétaire. Mais passer une valeur by consomme simplement la chaîne et prend en charge son tampon, donc à moins que l'ajout du caractère null ne force le tampon à être redimensionné, la conversion ne nécessite aucune copie du texte ou allocation. `CString::new Into<Vec<u8>> &str Vec<u8> String`

CString déréférence à , dont la méthode renvoie un pointage au début de la chaîne. C'est le type qui s'attend. Dans l'exemple, exécute la chaîne, recherche le caractère null qui s'y trouve et renvoie la longueur, sous la forme d'un nombre d'octets. CStr as_ptr *const c_char strlen strlen CString::new

Vous pouvez également déclarer des variables globales dans des blocs. Les systèmes POSIX ont une variable globale nommée qui contient les valeurs des variables d'environnement du processus. En C, il est déclaré : extern environ

```
extern char **environ;
```

Dans Rust, vous diriez :

```
use std::ffi::CStr;
use std::os::raw::c_char;

extern {
    static environ: *mut *mut c_char;
}
```

Pour imprimer le premier élément de l'environnement, vous pouvez écrire :

```
unsafe {
    if !environ.is_null() && !(*environ).is_null() {
        let var = CStr::from_ptr(*environ);
        println!("first environment variable: {}",
                 var.to_string_lossy())
    }
}
```

Après s'être assuré qu'il a un premier élément, le code appelle à construire un qui l'emprunte. La méthode renvoie un : si la chaîne C contient UTF-8 bien formé, l'emprunte son contenu sous la forme d'un , sans inclure l'octet null de fin. Sinon, fait une copie du texte dans le tas, remplace les séquences UTF-8 mal formées par le caractère de remplacement Unicode officiel, et construit une propriété à partir de cela. Quoi qu'il en soit, le résultat implémente , de sorte que vous pouvez l'imprimer avec le paramètre
format. environ CStr::from_ptr CStr to_string_lossy Cow<str> C ow &str to_string_lossy Cow Display {}

Utilisation des fonctions des bibliothèques

Pour utiliser les fonctions fournies par une bibliothèque particulière, vous pouvez placer un attribut au-dessus du bloc qui nomme la bibliothèque avec laquelle Rust doit lier l'exécutable. Par exemple, voici un programme qui appelle les méthodes d'initialisation et d'arrêt de ', mais ne fait rien d'autre: `#[link] extern libgit2`

```
use std::os::raw::c_int;

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
}

fn main() {
    unsafe {
        git_libgit2_init();
        git_libgit2_shutdown();
    }
}
```

Le bloc déclare les fonctions externes comme précédemment. L'attribut laisse une note dans la caisse à l'effet que, lorsque Rust crée l'exécutable final ou la bibliothèque partagée, il doit être lié à la bibliothèque. Rust utilise l'éditeur de liens système pour créer des exécutables; sous Unix, cela passe l'argument sur la ligne de commande de l'éditeur de liens ; sous Windows, il passe . `extern #[link(name = "git2")] git2 -lgit2 git2.LIB`

`#[link]` les attributs fonctionnent également dans les caisses de bibliothèque. Lorsque vous créez un programme qui dépend d'autres caisses, Cargo rassemble les notes de lien de l'ensemble du graphique de dépendance et les inclut toutes dans le lien final.

Dans cet exemple, si vous souhaitez suivre sur votre propre machine, vous devrez construire pour vous-même. Nous avons utilisé `libgit2` version 0.25.1. Pour compiler, vous devrez installer l'outil de génération CMake et le langage Python; nous avons utilisé CMake version 3.8.0 et Python version 2.7.13. `libgit2 libgit2`

Les instructions complètes pour la construction sont disponibles sur son site Web, mais elles sont assez simples pour que nous leur montrions l'essentiel ici. Sous Linux, supposons que vous avez déjà décompressé la source de la bibliothèque dans le répertoire */home/jimb/libgit2-0.25.1* : `libgit2`

```
$ cd /home/jimb/libgit2-0.25.1
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

Sous Linux, cela produit une bibliothèque partagée */home/jimb/libgit2-0.25.1/build/libgit2.so.0.25.1* avec le nid habituel de liens symboliques pointant vers elle, dont une nommée *libgit2.so*. Sous macOS, les résultats sont similaires, mais la bibliothèque est nommée *libgit2.dylib*.

Sur Windows, les choses sont également simples. Supposons que vous avez décompressé la source dans le répertoire *C:\Users\JimB\libgit2-0.25.1*. Dans une invite de commandes Visual Studio :

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

Ce sont les mêmes commandes que celles utilisées sous Linux, sauf que vous devez demander une version 64 bits lorsque vous exécutez CMake la première fois pour correspondre à votre compilateur Rust. (Si vous avez installé la chaîne d'outils Rust 32 bits, vous devez omettre l'indicateur à la première commande.) Cela produit une bibliothèque d'importation *git2.LIB* et une bibliothèque de liens *dynamiques git2.DLL*, tous deux dans le répertoire *C:\Users\JimB\libgit2-0.25.1\build\Debug*. (Les instructions restantes sont affichées pour Unix, sauf lorsque Windows est sensiblement différent.) `-A x64 cmake`

Créez le programme Rust dans un répertoire séparé :

```
$ cd /home/jimb
$ cargo new --bin git-toy
     Created binary (application) `git-toy` package
```

Prenez le code montré plus haut et mettez-le dans *src/main.rs*. Naturellement, si vous essayez de construire cela, Rust n'a aucune idée de l'endroit

où trouver le vous avez construit: `libgit2`

```
$ cd git-toy
$ cargo run
   Compiling git-toy v0.1.0 (/home/jimb/git-toy)
error: linking with `cc` failed: exit status: 1
  |
  = note: /usr/bin/ld: error: cannot find -lgit2
          src/main.rs:11: error: undefined reference to 'git_libgit2_init
          src/main.rs:12: error: undefined reference to 'git_libgit2_shut
          collect2: error: ld returned 1 exit status


error: could not compile `git-toy` due to previous error
```

Vous pouvez indiquer à Rust où rechercher des bibliothèques en écrivant un *script de build*, du code Rust que Cargo compile et exécute au moment de la build. Les scripts de build peuvent faire toutes sortes de choses : générer du code dynamiquement, compiler du code C à inclure dans la caisse, etc. Dans ce cas, tout ce dont vous avez besoin est d'ajouter un chemin de recherche de bibliothèque à la commande link de l'exécutable. Lorsque Cargo exécute le script de build, il analyse la sortie du script de build pour obtenir des informations de ce type, de sorte que le script de build doit simplement imprimer la bonne magie sur sa sortie standard.

Pour créer votre script de build, ajoutez un fichier nommé *build.rs* dans le même répertoire que le fichier *Cargo.toml*, avec le contenu suivant :

```
fn main() {
    println!(r"cargo:rustc-link-search=native=/home/jimb/libgit2-0.25.1/b
}
```

C'est la bonne voie pour Linux; sous Windows, vous devez modifier le chemin d'accès suivant le texte en . (Nous coupons quelques coins pour garder cet exemple simple; dans une application réelle, vous devriez éviter d'utiliser des chemins absolus dans votre script de génération. Nous citons la documentation qui montre comment le faire correctement à la fin de cette section.) `native= C:\Users\JimB\libgit2-0.25.1\build\Debug`

Maintenant, vous pouvez presque exécuter le programme. Sur macOS, cela peut fonctionner immédiatement; sur un système Linux, vous verrez probablement quelque chose comme ceci:

```
$ cargo run
   Compiling git-toy v0.1.0 (/tmp/rustbook-transcript-tests/git-toy)
    Finished dev [unoptimized + debuginfo] target(s)
     Running `target/debug/git-toy`
target/debug/git-toy: error while loading shared libraries:
libgit2.so.25: cannot open shared object file: No such file or directory
```

Cela signifie que, bien que Cargo ait réussi à lier l'exécutable à la bibliothèque, il ne sait pas où trouver la bibliothèque partagée au moment de l'exécution. Windows signale cet échec en faisant apparaître une boîte de dialogue. Sous Linux, vous devez définir la variable d'environnement : `LD_LIBRARY_PATH`

```
$ export LD_LIBRARY_PATH=/home/jimb/libgit2-0.25.1/build:$LD_LIBRARY_PATH
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy`
```

Sur macOS, vous devrez peut-être définir à la place. `DYLD_LIBRARY_PATH`

Sous Windows, vous devez définir la variable d'environnement : `PATH`

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug;%PATH%
> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy`
>
```

Naturellement, dans une application déployée, vous voudriez éviter d'avoir à définir des variables d'environnement juste pour trouver le code de votre bibliothèque. Une alternative consiste à lier statiquement la bibliothèque C à votre caisse. Cela copie les fichiers objet de la bibliothèque dans le fichier *.rlib* de la caisse, à côté des fichiers objet et des métadonnées du code Rust de la caisse. L'ensemble de la collection participe ensuite au lien final.

C'est une convention Cargo qu'une caisse qui donne accès à une bibliothèque C doit être nommée , où est le nom de la bibliothèque C. Une caisse ne doit contenir rien d'autre que la bibliothèque liée statiquement et les modules Rust contenant des blocs et des définitions de type. Les interfaces de niveau supérieur appartiennent alors à des caisses qui dépendent de la caisse. Cela permet à plusieurs caisses en amont de dépendre de la même caisse, en supposant qu'il existe une seule version de la caisse

qui répond aux besoins de chacun. `LIB-sys LIB -sys extern -sys -sys -sys`

Pour plus de détails sur la prise en charge par Cargo des scripts de build et des liens avec les bibliothèques système, consultez la documentation cargo en ligne. Il montre comment éviter les chemins absolus dans les scripts de construction, contrôler les indicateurs de compilation, utiliser des outils tels que , etc. La caisse fournit également de bons exemples à imiter; son script de build gère certaines situations complexes. `pkg-config git2-rs`

Une interface brute vers libgit2

Comprendre comment utiliser correctement se décompose en deux questions: `libgit2`

- Que faut-il pour utiliser les fonctions dans Rust? `libgit2`
- Comment pouvons-nous construire une interface Rust sûre autour d'eux?

Nous répondrons à ces questions une à la fois. Dans cette section, nous allons écrire un programme qui est essentiellement un bloc géant rempli de code Rust non idiomatique, reflétant le choc des systèmes de type et des conventions inhérent au mélange des langages. Nous appellerons cela l'interface *brute*. Le code sera désordonné, mais il indiquera clairement toutes les étapes qui doivent se produire pour que le code Rust puisse utiliser . `unsafe libgit2`

Ensuite, dans la section suivante, nous allons construire une interface sécurisée qui met les types de Rust à utiliser en appliquant les règles imposées à ses utilisateurs. Heureusement, il s'agit d'une bibliothèque C exceptionnellement bien conçue, de sorte que les questions que les exigences de sécurité de Rust nous obligent à poser ont toutes de très bonnes réponses, et nous pouvons construire une interface Rust idiomatique sans fonctions. `libgit2 libgit2 libgit2 unsafe`

Le programme que nous allons écrire est très simple : il prend un chemin comme argument de ligne de commande, y ouvre le référentiel Git et imprime la validation de tête. Mais cela suffit à illustrer les stratégies clés pour construire des interfaces Rust sûres et idiomatiques.

Pour l'interface brute, le programme finira par avoir besoin d'une collection un peu plus grande de fonctions et de types que ce que nous utilisions auparavant, il est donc logique de déplacer le bloc dans son propre

module. Nous allons créer un fichier nommé *raw.rs* dans *git-toy/src* dont le contenu est le suivant : `libgit2 extern`

```
#![allow(non_camel_case_types)]

use std::os::raw::{c_int, c_char, c_uchar};

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
    pub fn giterr_last() -> *const git_error;

    pub fn git_repository_open(out: *mut *mut git_repository,
                               path: *const c_char) -> c_int;
    pub fn git_repository_free(repo: *mut git_repository);

    pub fn git_reference_name_to_id(out: *mut git_oid,
                                    repo: *mut git_repository,
                                    reference: *const c_char) -> c_int;

    pub fn git_commit_lookup(out: *mut *mut git_commit,
                             repo: *mut git_repository,
                             id: *const git_oid) -> c_int;

    pub fn git_commit_author(commit: *const git_commit) -> *const git_sig
    pub fn git_commit_message(commit: *const git_commit) -> *const c_char
    pub fn git_commit_free(commit: *mut git_commit);
}

#[repr(C)] pub struct git_repository { _private: [u8; 0] }
#[repr(C)] pub struct git_commit { _private: [u8; 0] }

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}

pub const GIT_OID_RAWSZ: usize = 20;

#[repr(C)]
pub struct git_oid {
    pub id: [c_uchar; GIT_OID_RAWSZ]
}

pub type git_time_t = i64;
```

```
#[repr(C)]
pub struct git_time {
    pub time: git_time_t,
    pub offset: c_int
}

#[repr(C)]
pub struct git_signature {
    pub name: *const c_char,
    pub email: *const c_char,
    pub when: git_time
}
```

Chaque élément ici est modelé sur une déclaration des propres fichiers d'en-tête de . Par exemple, *libgit2-0.25.1/include/git2/repository.h* inclut cette déclaration : `libgit2`

```
extern int git_repository_open(git_repository **out, const char *path);
```

Cette fonction tente d'ouvrir le référentiel Git à l'adresse . Si tout se passe bien, il crée un objet et stocke un pointeur vers celui-ci à l'emplacement indiqué par . La déclaration de rouille équivalente est la suivante : `path git_repository out`

```
pub fn git_repository_open(out: *mut *mut git_repository,
                           path: *const c_char) -> c_int;
```

Les fichiers d'en-tête publics définissent le type comme un typedef pour un type struct incomplet : `libgit2 git_repository`

```
typedef struct git_repository git_repository;
```

Étant donné que les détails de ce type sont privés pour la bibliothèque, les en-têtes publics ne définissent jamais, ce qui garantit que les utilisateurs de la bibliothèque ne peuvent jamais créer eux-mêmes une instance de ce type. Un analogue possible à un type de structure incomplet dans Rust est le suivant: `struct git_repository`

```
#[repr(C)] pub struct git_repository { _private: [u8; 0] }
```

Il s'agit d'un type struct contenant un tableau sans éléments. Puisque le champ n'est pas , les valeurs de ce type ne peuvent pas être construites en dehors de ce module, ce qui est parfait comme reflet d'un type C qui ne

devrait jamais construire, et qui est manipulé uniquement par des pointeurs bruts. `_private pub libgit2`

Écrire de gros blocs à la main peut être une corvée. Si vous créez une interface Rust vers une bibliothèque C complexe, vous pouvez essayer d'utiliser la caisse, qui dispose de fonctions que vous pouvez utiliser à partir de votre script de build pour analyser les fichiers d'en-tête C et générer automatiquement les déclarations Rust correspondantes. Nous n'avons pas d'espace pour montrer en action ici, mais la page de `bindgen` sur crates.io comprend des liens vers sa documentation. `extern bindgen bindgen`

Ensuite, nous allons *réécrire complètement main.rs*. Tout d'abord, nous devons déclarer le module: `raw`

```
mod raw;
```

Selon les conventions de , les fonctions faillibles renvoient un code entier positif ou nul en cas de succès et négatif en cas d'échec. Si une erreur se produit, la fonction renvoie un pointeur vers une structure fournissant plus de détails sur ce qui s'est mal passé. possède cette structure, nous n'avons donc pas besoin de la libérer nous-mêmes, mais elle pourrait être écrasée par le prochain appel de bibliothèque que nous faisons. Une interface Rust appropriée utiliserait , mais dans la version brute, nous voulons utiliser les fonctions telles quelles, nous devrons donc lancer notre propre fonction pour gérer les erreurs: `libgit2 giterr_last git_error libgit2 Result libgit2`

```
use std::ffi::CStr;
use std::os::raw::c_int;

fn check(activity: &'static str, status: c_int) -> c_int {
    if status < 0 {
        unsafe {
            let error = &*raw::giterr_last();
            println!("error while {}: {} ({})",
                     activity,
                     CStr::from_ptr(error.message).to_string_lossy(),
                     error.klass);
            std::process::exit(1);
        }
    }

    status
}
```

Nous utiliserons cette fonction pour vérifier les résultats d'appels comme celui-ci : `libgit2`

```
    check("initializing library", raw::git_libgit2_init());
```

Cela utilise les mêmes méthodes utilisées précédemment: construire le à partir d'une chaîne C et le transformer en quelque chose que Rust peut imprimer. `CStr from_ptr CStr to_string_lossy`

Ensuite, nous avons besoin d'une fonction pour imprimer un commit:

```
    unsafe fn show_commit(commit: *const raw::git_commit) {
        let author = raw::git_commit_author(commit);

        let name = CStr::from_ptr((*author).name).to_string_lossy();
        let email = CStr::from_ptr((*author).email).to_string_lossy();
        println!("{} <{}>\n", name, email);

        let message = raw::git_commit_message(commit);
        println!("{}", CStr::from_ptr(message).to_string_lossy());
    }
```

Donné un pointeur vers un , appelle et pour récupérer les informations dont il a besoin. Ces deux fonctions suivent une convention que la documentation explique comme suit
: `git_commit show_commit git_commit_author git_commit_messa ge libgit2`

> *Si une fonction renvoie un objet en tant que valeur de retour, cette fonction est un getter et la durée de vie de l'objet est liée à l'objet parent.*

En termes de rouille, et sont empruntés à : n'a pas besoin de les libérer lui-même, mais il ne doit pas les conserver après sa libération. Étant donné que cette API utilise des pointeurs bruts, Rust ne vérifiera pas leur durée de vie pour nous: si nous créons accidentellement des pointeurs pendants, nous ne le découvrirons probablement pas avant que le programme ne plante. `author message commit show_commit commit`

Le code précédent suppose que ces champs contiennent du texte UTF-8, ce qui n'est pas toujours correct. Git autorise également d'autres encodages. Interpréter correctement ces cordes impliquerait probablement d'utiliser la caisse. Par souci de brièveté, nous passerons sous silence ces questions ici. `encoding`

La fonction de notre programme se lit comme suit : `main`

```
use std::ffi::CString;
use std::mem;
use std::ptr;
use std::os::raw::c_char;

fn main() {
    let path = std::env::args().skip(1).next()
        .expect("usage: git-toy PATH");
    let path = CString::new(path)
        .expect("path contains null characters");

    unsafe {
        check("initializing library", raw::git_libgit2_init());

        let mut repo = ptr::null_mut();
        check("opening repository",
              raw::git_repository_open(&mut repo, path.as_ptr()));

        let c_name = b"HEAD\0".as_ptr() as *const c_char;
        let oid = {
            let mut oid = mem::MaybeUninit::uninit();
            check("looking up HEAD",
                  raw::git_reference_name_to_id(oid.as_mut_ptr(), repo, c
            oid.assume_init()
        };

        let mut commit = ptr::null_mut();
        check("looking up commit",
              raw::git_commit_lookup(&mut commit, repo, &oid));

        show_commit(commit);

        raw::git_commit_free(commit);

        raw::git_repository_free(repo);

        check("shutting down library", raw::git_libgit2_shutdown());
    }
}
```

Cela commence par du code pour gérer l'argument path et initialiser la bibliothèque, ce que nous avons déjà vu. Le premier nouveau code est le suivant :

```
let mut repo = ptr::null_mut();
check("opening repository",
      raw::git_repository_open(&mut repo, path.as_ptr()));
```

L'appel à tente d'ouvrir le référentiel Git au chemin d'accès donné. S'il réussit, il lui alloue un nouvel objet et définit pour pointer vers cela. Rust contraint implicitement les références à des pointeurs bruts, donc passer ici fournit l'appel
attendu. `git_repository_open git_repository repo &mut repo *mut *mut git_repository`

Cela montre une autre convention en cours d'utilisation (à partir de la documentation): `libgit2 libgit2`

> *Les objets qui sont renvoyés via le premier argument en tant que pointeur à pointeur appartiennent à l'appelant et il est responsable de les libérer.*

En termes de rouille, des fonctions telles que transmettre la propriété de la nouvelle valeur à l'appelant. `git_repository_open`

Ensuite, considérez le code qui recherche le hachage d'objet de la validation d'en-tête actuelle du référentiel :

```
let oid = {
    let mut oid = mem::MaybeUninit::uninit();
    check("looking up HEAD",
          raw::git_reference_name_to_id(oid.as_mut_ptr(), repo, c_name));
    oid.assume_init()
};
```

Le type stocke un identificateur d'objet, c'est-à-dire un code de hachage 160 bits que Git utilise en interne (et tout au long de son interface utilisateur agréable) pour identifier les validations, les versions individuelles des fichiers, etc. Cet appel permet de rechercher l'identificateur d'objet de la validation en cours. `git_oid git_reference_name_to_id "HEAD"`

En C, il est parfaitement normal d'initialiser une variable en lui passant un pointeur vers une fonction qui remplit sa valeur ; c'est ainsi qu'on s'attend à traiter son premier argument. Mais Rust ne nous laissera pas emprunter une référence à une variable non initialisée. Nous pourrions initialiser avec des zéros, mais c'est un gaspillage: toute valeur qui y est stockée sera simplement écrasée. `git_reference_name_to_id oid`

Il est possible de demander à Rust de nous donner une mémoire non initialisée, mais parce que la lecture de la mémoire non initialisée à tout moment est un comportement instantané indéfini, Rust fournit une abstraction, , pour faciliter son utilisation. dit au compilateur de mettre de côté suffisamment de mémoire pour votre type, mais de ne pas le toucher

jusqu'à ce que vous disiez qu'il est sûr de le faire. Bien que cette mémoire appartienne au , le compilateur évitera également certaines optimisations qui pourraient autrement provoquer un comportement indéfini, même sans accès explicite à la mémoire non initialisée de votre code. `MaybeUninit MaybeUninit<T> T MaybeUninit`

`MaybeUninit` fournit une méthode, , qui produit un pointage vers la mémoire potentiellement non initialisée qu'il enveloppe. En passant ce pointeur à une fonction étrangère qui initialise la mémoire, puis en appelant la méthode non sécurisée sur le pour produire un comportement entièrement initialisé , vous pouvez éviter un comportement indéfini sans la surcharge supplémentaire qui provient de l'initialisation et de la suppression immédiate d'une valeur. est dangereux car l'appeler sur un sans être certain que la mémoire est réellement initialisée provoquera immédiatement un comportement indéfini. `as_mut_ptr() *mut T assume_init MaybeUninit T assume_init MaybeUninit`

Dans ce cas, il est sûr car initialise la mémoire appartenant au . Nous pourrions également utiliser pour les variables et, mais comme il ne s'agit que de mots simples, nous allons de l'avant et les initialisons à null: `git_reference_name_to_id MaybeUninit MaybeUninit repo c ommit`

```
let mut commit = ptr::null_mut();
check("looking up commit",
      raw::git_commit_lookup(&mut commit, repo, &oid));
```

Cela prend l'identificateur d'objet de la validation et recherche la validation réelle, en stockant un pointeur sur la réussite. `git_commit commit`

Le reste de la fonction doit être explicite. Il appelle la fonction définie précédemment, libère les objets commit et repository et arrête la bibliothèque. `main show_commit`

Maintenant, nous pouvons essayer le programme sur n'importe quel dépôt Git prêt à portée de main:

```
$ cargo run /home/jimb/rbattle
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy /home/jimb/rbattle`
Jim Blandy <jimb@red-bean.com>

Animate goop a bit.
```

Une interface sécurisée vers libgit2

L'interface brute est un exemple parfait d'une fonctionnalité dangereuse: elle peut certainement être utilisée correctement (comme nous le faisons ici, pour autant que nous le sachions), mais Rust ne peut pas appliquer les règles que vous devez suivre. La conception d'une API sécurisée pour une bibliothèque comme celle-ci consiste à identifier toutes ces règles, puis à trouver des moyens de transformer toute violation de celles-ci en une erreur de type ou de vérification d'emprunt. `libgit2`

Voici donc les règles de fonctionnement des fonctionnalités utilisées par le programme : `libgit2`

- Vous devez appeler avant d'utiliser une autre fonction de bibliothèque. Vous ne devez utiliser aucune fonction de bibliothèque après avoir appelé . `git_libgit2_init git_libgit2_shutdown`
- Toutes les valeurs transmises aux fonctions doivent être entièrement initialisées, à l'exception des paramètres de sortie. `libgit2`
- Lorsqu'un appel échoue, les paramètres de sortie transmis pour contenir les résultats de l'appel ne sont pas initialisés et vous ne devez pas utiliser leurs valeurs.
- Un objet fait référence à l'objet dont il est dérivé, de sorte que le premier ne doit pas survivre au second. (Ce n'est pas précisé dans la documentation; nous l'avons déduit de la présence de certaines fonctions dans l'interface, puis vérifié en lisant le code source.) `git_commit git_repository libgit2`
- De même, a est toujours emprunté à un donné, et le premier ne doit pas survivre au second. (La documentation couvre ce cas.) `git_signature git_commit`
- Le message associé à une validation ainsi que le nom et l'adresse e-mail de l'auteur sont tous empruntés à la validation et ne doivent pas être utilisés après la libération de la validation.
- Une fois qu'un objet a été libéré, il ne doit plus jamais être utilisé. `libgit2`

Il s'avère que vous pouvez créer une interface Rust qui applique toutes ces règles, soit via le système de type Rust, soit en gérant les détails en interne. `libgit2`

Avant de commencer, restructurons un peu le projet. Nous aimerions avoir un module qui exporte l'interface sécurisée, dont l'interface brute du programme précédent est un sous-module privé. `git`

L'arborescence source entière ressemblera à ceci :

```
git-toy/
├── Cargo.toml
├── build.rs
└── src/
    ├── main.rs
    └── git/
        ├── mod.rs
        └── raw.rs
```

En suivant les règles que nous avons expliquées dans « Modules dans des fichiers séparés », la source du module apparaît dans *git/mod.rs*, et la source de son sous-module va dans *git/raw.rs*. `git git::raw`

Encore une fois, nous allons *réécrire complètement main.rs*. Il doit commencer par une déclaration du module: `git`

```
mod git;
```

Ensuite, nous devrons créer le sous-répertoire *git* et *y déplacer raw.rs*:

```
$ cd /home/jimb/git-toy
$ mkdir src/git
$ mv src/raw.rs src/git/raw.rs
```

Le module doit déclarer son sous-module. Le fichier *src/git/mod.rs* doit indiquer : `git raw`

```
mod raw;
```

Comme ce n'est pas le cas, ce sous-module n'est pas visible par le programme principal. `pub`

Dans un peu, nous devrons utiliser certaines fonctions de la caisse, nous devons donc ajouter une dépendance dans *Cargo.toml*. Le fichier complet se lit maintenant comme suit : `libc`

```
[package]
name = "git-toy"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2021"

[dependencies]
libc = "0.2"
```

Maintenant que nous avons restructuré nos modules, considérons la gestion des erreurs. Même la fonction d'initialisation de Même peut renvoyer un code d'erreur, nous devrons donc le régler avant de pouvoir commencer. Une interface Rust idiomatique a besoin de son propre type qui capture le code d'échec ainsi que le message d'erreur et la classe de . Un type d'erreur approprié doit implémenter les traits habituels , et . Ensuite, il a besoin de son propre type qui utilise ce type. Voici les définitions nécessaires dans *src/git/mod.rs*

```
: libgit2 Error libgit2 giterr_last Error Debug Display Resu
lt Error
```

```
use std::error;
use std::fmt;
use std::result;

#[derive(Debug)]
pub struct Error {
    code: i32,
    message: String,
    class: i32
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> result::Result<(), fmt::Erro
        // Displaying an `Error` simply displays the message from libgit2
        self.message.fmt(f)
    }
}

impl error::Error for Error { }

pub type Result<T> = result::Result<T, Error>;
```

Pour vérifier le résultat des appels de bibliothèque bruts, le module a besoin d'une fonction qui transforme un code de retour en un

```
: libgit2 Result
```

```
use std::os::raw::c_int;
use std::ffi::CStr;

fn check(code: c_int) -> Result<c_int> {
    if code >= 0 {
        return Ok(code);
    }

    unsafe {
```

```
            let error = raw::giterr_last();

            // libgit2 ensures that (*error).message is always non-null and r
            // terminated, so this call is safe.
            let message = CStr::from_ptr((*error).message)
                .to_string_lossy()
                .into_owned();

            Err(Error {
                code: code as i32,
                message,
                class: (*error).klass as i32
            })
        }
    }
```

La principale différence entre cela et la fonction de la version brute est que cela construit une valeur au lieu d'imprimer un message d'erreur et de quitter immédiatement. `check Error`

Nous sommes maintenant prêts à nous attaquer à l'initialisation. L'interface sécurisée fournira un type qui représente un référentiel Git ouvert, avec des méthodes pour résoudre les références, rechercher des validations, etc. En continuant dans *git/mod.rs*, voici la définition de `: libgit2 Repository Repository`

```
    /// A Git repository.
    pub struct Repository {
        // This must always be a pointer to a live `git_repository` structure
        // No other `Repository` may point to it.
        raw: *mut raw::git_repository
    }
```

Le champ de A n'est pas public. Étant donné que seul le code de ce module peut accéder au pointeur, l'obtention correcte de ce module devrait garantir que le pointeur est toujours utilisé correctement. `Repository raw raw::git_repository`

Si la seule façon de créer un est d'ouvrir avec succès un nouveau référentiel Git, cela garantira que chacun pointe vers un objet distinct `: Repository Repository git_repository`

```
    use std::path::Path;
    use std::ptr;

    impl Repository {
```

```
        pub fn open<P: AsRef<Path>>(path: P) -> Result<Repository> {
            ensure_initialized();

            let path = path_to_cstring(path.as_ref())?;
            let mut repo = ptr::null_mut();
            unsafe {
                check(raw::git_repository_open(&mut repo, path.as_ptr()))?;
            }
            Ok(Repository { raw: repo })
        }
    }
```

Étant donné que la seule façon de faire quoi que ce soit avec l'interface sécurisée est de commencer par une valeur, et commence par un appel à , nous pouvons être sûrs que cela sera appelé avant toute fonction. Sa définition est la suivante

```
:Repository Repository::open ensure_initialized ensure_init
ialized libgit2
```

```
    fn ensure_initialized() {
        static ONCE: std::sync::Once = std::sync::Once::new();
        ONCE.call_once(|| {
            unsafe {
                check(raw::git_libgit2_init())
                    .expect("initializing libgit2 failed");
                assert_eq!(libc::atexit(shutdown), 0);
            }
        });
    }

    extern fn shutdown() {
        unsafe {
            if let Err(e) = check(raw::git_libgit2_shutdown()) {
                eprintln!("shutting down libgit2 failed: {}", e);
                std::process::abort();
            }
        }
    }
```

Le type permet d'exécuter le code d'initialisation de manière sécurisée. Seul le premier thread à appeler exécute la fermeture donnée. Tous les appels suivants, par ce thread ou tout autre, se bloquent jusqu'à ce que le premier soit terminé, puis reviennent immédiatement, sans exécuter à nouveau la fermeture. Une fois la fermeture terminée, l'appel est bon marché, ne nécessitant rien de plus qu'une charge atomique d'un dra-

peau stocké dans
`.std::sync::Once ONCE.call_once ONCE.call_once ONCE`

Dans le code précédent, la fermeture de l'initialisation appelle et vérifie le résultat. Il s'agite un peu et utilise simplement pour s'assurer que l'initialisation a réussi, au lieu d'essayer de propager les erreurs à l'appelant. `git_libgit2_init expect`

Pour s'assurer que le programme appelle , la fermeture d'initialisation utilise la fonction de la bibliothèque C, qui prend un pointeur vers une fonction à appeler avant la fermeture du processus. Les fermetures de rouille ne peuvent pas servir de pointeurs de fonction C : une fermeture est une valeur d'un type anonyme portant les valeurs de toutes les variables qu'elle capture ou auxquelles elle fait référence ; un pointeur de fonction C n'est qu'un pointeur. Cependant, les types Rust fonctionnent bien, tant que vous les déclarez afin que Rust sache utiliser les conventions d'appel C. La fonction locale correspond à la facture et garantit une fermeture
correcte. `git_libgit2_shutdown atexit fn extern shutdown libgit2`

Dans « Unwinding », nous avons mentionné qu'il est un comportement indéfini pour une panique de traverser les frontières linguistiques. L'appel de à est une telle limite, il est donc essentiel de ne pas paniquer. C'est pourquoi vous ne pouvez pas simplement utiliser pour gérer les erreurs signalées par . Au lieu de cela, il doit signaler l'erreur et mettre fin au processus lui-même. POSIX interdit d'appeler dans un gestionnaire, donc appelle pour arrêter le programme
brusquement. `atexit shutdown shutdown shutdown .expect raw:: git_libgit2_shutdown exit atexit shutdown std::process::abort`

Il peut être possible de prendre des dispositions pour appeler plus tôt, par exemple lorsque la dernière valeur est supprimée. Mais quelle que soit la façon dont nous organisons les choses, l'appel doit être la responsabilité de l'API sûre. Au moment où il est appelé, tous les objets existants deviennent dangereux à utiliser, de sorte qu'une API sécurisée ne doit pas exposer cette fonction
directement. `git_libgit2_shutdown Repository git_libgit2_shutdown libgit2`

Le pointeur brut d'un doit toujours pointer vers un objet actif. Cela implique que la seule façon de fermer un référentiel est de supprimer la valeur qui le possède : `Repository git_repository Repository`

```
impl Drop for Repository {
    fn drop(&mut self) {
        unsafe {
            raw::git_repository_free(self.raw);
        }
    }
}
```

En appelant uniquement lorsque le seul pointeur vers le est sur le point de disparaître, le type garantit également que le pointeur ne sera jamais utilisé après sa libération. `git_repository_free` `raw::git_repository` `Repository`

La méthode utilise une fonction privée appelée , qui a deux définitions, l'une pour les systèmes de type Unix et l'autre pour Windows : `Repository::open` `path_to_cstring`

```
use std::ffi::CString;

#[cfg(unix)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // The `as_bytes` method exists only on Unix-like systems.
    use std::os::unix::ffi::OsStrExt;

    Ok(CString::new(path.as_os_str().as_bytes())?)
}

#[cfg(windows)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // Try to convert to UTF-8. If this fails, libgit2 can't handle the p
    // anyway.
    match path.to_str() {
        Some(s) => Ok(CString::new(s)?),
        None => {
            let message = format!("Couldn't convert path '{}' to UTF-8",
                                  path.display());
            Err(message.into())
        }
    }
}
```

L'interface rend ce code un peu délicat. Sur toutes les plates-formes, accepte les chemins d'accès en tant que chaînes C terminées par une valeur NULL. Sous Windows, suppose que ces chaînes C contiennent un UTF-8 bien formé et les convertit en interne en chemins d'accès 16 bits requis par Windows. Cela fonctionne généralement, mais ce n'est pas idéal. Win-

dows autorise les noms de fichiers qui ne sont pas bien formés Unicode et ne peuvent donc pas être représentés en UTF-8. Si vous avez un tel fichier, il est impossible de passer son nom à
. libgit2 libgit2 libgit2 libgit2

Dans Rust, la représentation correcte d'un chemin de système de fichiers est un , soigneusement conçu pour gérer tout chemin pouvant apparaître sur Windows ou POSIX. Cela signifie qu'il existe des valeurs sur Windows auxquelles on ne peut pas passer , car elles ne sont pas bien formées UTF-8. Donc, bien que le comportement de 'soit loin d'être idéal, c'est en fait le meilleur que nous puissions faire étant donné l'interface de
.. std::path::Path Path libgit2 path_to_cstring libgit2

Les deux définitions qui viennent d'être présentées reposent sur des conversions de notre type : l'opérateur tente de telles conversions, et la version Windows appelle explicitement . Ces conversions ne sont pas remarquables : path_to_cstring Error ? .into()

```
impl From<String> for Error {
    fn from(message: String) -> Error {
        Error { code: -1, message, class: 0 }
    }
}

// NulError is what `CString::new` returns if a string
// has embedded zero bytes.
impl From<std::ffi::NulError> for Error {
    fn from(e: std::ffi::NulError) -> Error {
        Error { code: -1, message: e.to_string(), class: 0 }
    }
}
```

Ensuite, voyons comment résoudre une référence Git à un identificateur d'objet. Étant donné qu'un identificateur d'objet n'est qu'une valeur de hachage de 20 octets, il est parfaitement correct de l'exposer dans l'API sécurisée :

```
/// The identifier of some sort of object stored in the Git object
/// database: a commit, tree, blob, tag, etc. This is a wide hash of the
/// object's contents.
pub struct Oid {
    pub raw: raw::git_oid
}
```

Nous allons ajouter une méthode pour effectuer la recherche
: `Repository`

```
use std::mem;
use std::os::raw::c_char;

impl Repository {
    pub fn reference_name_to_id(&self, name: &str) -> Result<Oid> {
        let name = CString::new(name)?;
        unsafe {
            let oid = {
                let mut oid = mem::MaybeUninit::uninit();
                check(raw::git_reference_name_to_id(
                        oid.as_mut_ptr(), self.raw,
                        name.as_ptr() as *const c_char))?;
                oid.assume_init()
            };
            Ok(Oid { raw: oid })
        }
    }
}
```

Bien qu'elle ne soit pas initialisée lorsque la recherche échoue, cette fonction garantit que son appelant ne peut jamais voir la valeur non initialisée simplement en suivant l'idiome de Rust : soit l'appelant obtient une valeur correctement initialisée, soit il obtient un
`.oid Result Ok Oid Err`

Ensuite, le module a besoin d'un moyen de récupérer les commits du référentiel. Nous allons définir un type comme suit : `Commit`

```
use std::marker::PhantomData;

pub struct Commit<'repo> {
    // This must always be a pointer to a usable `git_commit` structure.
    raw: *mut raw::git_commit,
    _marker: PhantomData<&'repo Repository>
}
```

Comme nous l'avons mentionné précédemment, un objet ne doit jamais survivre à l'objet dont il a été récupéré. Les durées de vie de Rust permettent au code de capturer cette règle avec
précision. `git_commit git_repository`

L'exemple plus haut dans ce chapitre utilisait un champ pour indiquer à Rust de traiter un type comme s'il contenait une référence avec une

durée de vie donnée, même si le type ne contenait apparemment aucune référence de ce type. Le type doit faire quelque chose de similaire. Dans ce cas, le type du champ est , indiquant que Rust doit traiter comme s'il détenait une référence à vie à certains
. `RefWithFlag PhantomData Commit _marker PhantomData<&'repo Repository> Commit<'repo> 'repo Repository`

La méthode de recherche d'une validation est la suivante :

```
impl Repository {
    pub fn find_commit(&self, oid: &Oid) -> Result<Commit> {
        let mut commit = ptr::null_mut();
        unsafe {
            check(raw::git_commit_lookup(&mut commit, self.raw, &oid.raw)
        }
        Ok(Commit { raw: commit, _marker: PhantomData })
    }
}
```

Comment cela relie-t-il la durée de vie de la '? La signature de omet les durées de vie des références impliquées selon les règles décrites dans « Omis des paramètres de durée de vie ». Si nous devions écrire les durées de vie, la signature complète se lirait comme suit: `Commit Repository find_commit`

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
    -> Result<Commit<'repo>>
```

C'est exactement ce que nous voulons: Rust traite le retourné comme s'il empruntait quelque chose à , qui est le . `Commit self Repository`

Lorsque a est lâché, il doit libérer son : `Commit raw::git_commit`

```
impl<'repo> Drop for Commit<'repo> {
    fn drop(&mut self) {
        unsafe {
            raw::git_commit_free(self.raw);
        }
    }
}
```

À partir d'un , vous pouvez emprunter un (un nom et une adresse e-mail) et le texte du message de validation : `Commit Signature`

```
impl<'repo> Commit<'repo> {
    pub fn author(&self) -> Signature {
```

```
            unsafe {
                Signature {
                    raw: raw::git_commit_author(self.raw),
                    _marker: PhantomData
                }
            }
        }

        pub fn message(&self) -> Option<&str> {
            unsafe {
                let message = raw::git_commit_message(self.raw);
                char_ptr_to_str(self, message)
            }
        }
    }
```

Voici le type : `Signature`

```
    pub struct Signature<'text> {
        raw: *const raw::git_signature,
        _marker: PhantomData<&'text str>
    }
```

Un objet emprunte toujours son texte à d'autres endroits; en particulier, les signatures retournées en empruntant leur texte au . Donc, notre type de sécurité comprend un pour dire à Rust de se comporter comme s'il contenait un avec une durée de vie de . Tout comme auparavant, relie correctement cette vie du il revient à celle du sans que nous ayons besoin d'écrire une chose. La méthode fait de même avec la conservation du message de
validation. `git_signature git_commit_author git_commit Signat ure PhantomData<&'text str> &str 'text Commit::author 'text Signature Commit Commi t::message Option<&str>`

A inclut des méthodes pour récupérer le nom et l'adresse e-mail de l'auteur : `Signature`

```
    impl<'text> Signature<'text> {
        /// Return the author's name as a `&str`,
        /// or `None` if it is not well-formed UTF-8.
        pub fn name(&self) -> Option<&str> {
            unsafe {
                char_ptr_to_str(self, (*self.raw).name)
            }
        }
```

```
        /// Return the author's email as a `&str`,
        /// or `None` if it is not well-formed UTF-8.
        pub fn email(&self) -> Option<&str> {
            unsafe {
                char_ptr_to_str(self, (*self.raw).email)
            }
        }
    }
```

Les méthodes précédentes dépendent d'une fonction utilitaire privée : `char_ptr_to_str`

```
    /// Try to borrow a `&str` from `ptr`, given that `ptr` may be null or
    /// refer to ill-formed UTF-8. Give the result a lifetime as if it were
    /// borrowed from `_owner`.
    ///
    /// Safety: if `ptr` is non-null, it must point to a null-terminated C
    /// string that is safe to access for at least as long as the lifetime of
    /// `_owner`.
    unsafe fn char_ptr_to_str<T>(_owner: &T, ptr: *const c_char) -> Option<&s
        if ptr.is_null() {
            return None;
        } else {
            CStr::from_ptr(ptr).to_str().ok()
        }
    }
```

La valeur du paramètre n'est jamais utilisée, mais sa durée de vie l'est. Rendre explicites les durées de vie dans la signature de cette fonction nous donne : `_owner`

```
    fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
        -> Option<&'o str>
```

La fonction renvoie un dont la durée de vie est complètement illimitée, puisqu'elle a été empruntée à un pointeur brut déréférencé. Les durées de vie illimitées sont presque toujours inexactes, il est donc bon de les contraindre dès que possible. L'inclusion du paramètre entraîne l'attribution par Rust de sa durée de vie au type de la valeur renvoyée, afin que les appelants puissent recevoir une référence délimitée plus précise. `CStr::from_ptr &CStr _owner`

Il n'est pas clair dans la documentation si un 's et des pointeurs peuvent être nuls, bien que la documentation soit assez bonne. Vos auteurs ont fouillé dans le code source pendant un certain temps sans pouvoir se per-

suader d'une manière ou d'une autre et ont finalement décidé qu'il valait mieux se préparer à des pointeurs nuls au cas où. Dans Rust, ce genre de question est répondu immédiatement par le type: si c'est , vous pouvez compter sur la chaîne pour être là; si c'est , c'est facultatif. `libgit2 git_signature email author libgit2 char_ptr _to_str &str Option<&str>`

Enfin, nous avons fourni des interfaces sécurisées pour toutes les fonctionnalités dont nous avons besoin. La nouvelle fonction dans *src / main.rs* est un peu allégée et ressemble à un vrai code Rust: `main`

```
fn main() {
    let path = std::env::args_os().skip(1).next()
        .expect("usage: git-toy PATH");

    let repo = git::Repository::open(&path)
        .expect("opening repository");

    let commit_oid = repo.reference_name_to_id("HEAD")
        .expect("looking up 'HEAD' reference");

    let commit = repo.find_commit(&commit_oid)
        .expect("looking up commit");

    let author = commit.author();
    println!("{} <{}>\n",
             author.name().unwrap_or("(none)"),
             author.email().unwrap_or("none"));

    println!("{}", commit.message().unwrap_or("(none)"));
}
```

Dans ce chapitre, nous sommes passés d'interfaces simplistes qui ne fournissent pas beaucoup de garanties de sécurité à une API sécurisée enveloppant une API intrinsèquement dangereuse en faisant en sorte que toute violation du contrat de cette dernière soit une erreur de type Rust. Le résultat est une interface que Rust peut s'assurer que vous utilisez correctement. Pour la plupart, les règles que nous avons fait appliquer par Rust sont le genre de règles que les programmeurs C et C ++ finissent par s'imposer de toute façon. Ce qui rend Rust beaucoup plus strict que C et C ++, ce n'est pas que les règles sont si étrangères, mais que cette application est mécanique et complète.

Conclusion

Rust n’est pas un langage simple. Son but est de couvrir deux mondes très différents. C’est un langage de programmation moderne, sûr par conception, avec des commodités comme des fermetures et des itérateurs, mais il vise à vous donner le contrôle des capacités brutes de la machine sur laquelle il fonctionne, avec une surcharge d’exécution minimale.

Les contours de la langue sont déterminés par ces objectifs. Rust parvient à combler la majeure partie de l’écart avec un code sûr. Son vérificateur d’emprunt et ses abstractions à coût nul vous rapprochent le plus possible du métal nu sans risquer un comportement indéfini. Lorsque cela ne suffit pas ou lorsque vous souhaitez tirer parti du code C existant, le code dangereux et l’interface de la fonction étrangère sont prêts. Mais encore une fois, la langue ne vous offre pas seulement ces fonctionnalités dangereuses et vous souhaite bonne chance. L’objectif est toujours d’utiliser des fonctionnalités dangereuses pour créer des API sécurisées. C’est ce que nous avons fait avec . C’est aussi ce que l’équipe Rust a fait avec , , les autres collections, canaux, et plus encore : la bibliothèque standard est pleine d’abstractions sécurisées, implémentées avec du code dangereux dans les coulisses. `libgit2` `Box` `Vec`

Un langage avec les ambitions de Rust n’était peut-être pas destiné à être le plus simple des outils. Mais Rust est sûr, rapide, simultané et efficace. Utilisez-le pour construire des systèmes volumineux, rapides, sécurisés et robustes qui tirent parti de toute la puissance du matériel sur lequel ils s’exécutent. Utilisez-le pour améliorer le logiciel.