Chapitre 23. Fonctions étrangères

> *Cyberespace. Complexité impensable. Des lignes de lumière s'étendaient dans le non-espace de l'esprit, des grappes et des constellations de données. Comme les lumières de la ville, qui s'éloignent. . .*
>
> —William Gibson, *neuromancien*

Tragiquement, tous les programmes du 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 qui relieavec `libgit2` , 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 des fonctions C directement depuis Rust, en utilisant les fonctionnalités non sécurisées présentées dans le chapitre précédent. Ensuite, nous montrerons comment construire une interface sécurisée pour , en nous inspirant du crate `libgit2` open source `git2-rs` , qui fait exactement cela.

Nous supposerons que vous connaissez le C et les mécanismes de compilation et de liaison des programmes C. Travailler avec C++ est similaire. Nous supposerons également que vous êtes quelque peu familiarisé avec le système de contrôle de version Git.

Il existe des caisses Rust pour communiquer avec de nombreux autres langages, notamment Python, JavaScript, Lua et Java. Nous n'avons pas la place de 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, quel que soit le langage avec lequel vous devez travailler.

Trouver des représentations de données communes

Le dénominateur commun de Rust et C est le langage machine, donc pour anticiper à quoi ressemblent les valeurs de Rust pour le code C, ou vice versa, vous devez considérer leurs représentations au niveau de la machine. Tout au long du livre, nous nous sommes efforcés de montrer comment les valeurs sont réellement représentées en mémoire, vous avez donc probablement remarqué que les mondes de données de C et de Rust ont beaucoup en commun : un Rust `usize` et un C `size_t` sont identiques, par exemple , et les structures sont fondamentalement la même idée dans les deux langages. Pour établir une correspondance entre les types Rust et C, nous allons commencer par les primitives, puis progresser vers des types plus compliqués.

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

Tableau 23-1. `std::os::raw` types en rouille

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

- Les pointeurs C et C++ et les références C++ correspondent aux types de pointeurs bruts de Rust, `*mut T` et `*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 grandeur 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 de structures Rust compatibles avec les structures C, vous pouvez utiliser l' `#[repr(C)]` attribut. Placer `#[repr(C)]` au-dessus d'une définition de structure demande à Rust de disposer les champs de la structure en mémoire de la même manière qu'un compilateur C disposerait le type de structure C analogue. Par exemple, `libgit2` le fichier d'en-tête *git2/errors.h* de définit la structure C suivante pour fournir des détails sur une erreur précédemment signalée :

```
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' `#[repr(C)]` attribut n'affecte que 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 : `*const c_char` for `char *`, `c_int` for `int`, etc.

Dans ce cas particulier, l' `#[repr(C)]` attribut ne change probablement pas la disposition de `git_error`. Il n'y a vraiment pas trop 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 nulle ne prennent pas de place. L' `#`

`[repr(C)]` attribut indique à Rust de suivre les règles de C pour le type donné.

Vous pouvez également utiliser `#[repr(C)]` pour contrôler la représentation des énumérations de style 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 énumérations. Par exemple, nous avons mentionné l'astuce que Rust utilise pour stocker `Option<&T>` en un seul mot (si `T` est dimensionné). Sans `#[repr(C)]` , Rust utiliserait un seul octet pour représenter l' `git_error_code` énumération ; avec `#[repr(C)]` , Rust utilise une valeur de la taille d'un C `int` , tout comme C le ferait.

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

```
#include <stdint.h>

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

Comme mentionné précédemment, `#[repr(C)]` s'applique également aux syndicats. Les champs des `#[repr(C)]` unions commencent toujours au premier bit de la mémoire de l'union, index 0.

Supposons que vous ayez une structure C qui utilise une union pour contenir certaines données et une valeur de balise pour indiquer quel champ de l'union doit être utilisé, similaire à une énumération 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 `#[repr(C)]` aux types enum, structure et union, et en utilisant une `match` instruction qui sélectionne un champ union dans une structure plus grande basée sur la balise :

```
#[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 == 0.0
            _ => false
        }
    }
}
```

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

Qui passeles cordes entre Rust et C sont un peu plus dures. C représente une chaîne sous la forme d'un 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, `String` soit en tant que deuxième mot d'une référence fat `&str` . Les chaînes de rouille ne sont pas terminées par null ; en fait, ils peuvent inclure des caractères nuls dans leur contenu, comme tout autre caractère.

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

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

Déclarer des fonctions étrangères et des variables

Un `extern` blocdéclareles fonctionsou 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, nous pouvons donc informer Rust de la `strlen` fonction de la bibliothèque C comme ceci :

```
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 à lier plus tard.

Rust suppose que les fonctions déclarées à l'intérieur `extern` des blocs utilisent les conventions C pour passer des arguments et accepter les valeurs de retour. Ils sont définis comme `unsafe` des fonctions. Ce sont les bons choix pour `strlen` : 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, ce qui est un contrat que Rust ne peut pas appliquer. (Presque toute fonction qui prend un pointeur brut doit être `unsafe` : safe Rust peut construire des pointeurs bruts à partir d'entiers arbitraires, et déréférencer un tel pointeur serait un comportement indéfini.)

Avec ce `extern` bloc, on peut appeler `strlen` comme n'importe quelle autre fonction Rust, bien que son type le trahisse en touriste :

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

Le coût `CString::new` dépend du type que vous lui passez. Il accepte tout ce qui implémente `Into<Vec<u8>>` . Passer a `&str` implique une allocation et une copie, car la conversion en `Vec<u8>` construit une copie allouée par tas de la chaîne que le vecteur doit posséder. Mais le passage d'une `String` valeur par consomme simplement la chaîne et prend le contrôle de son tampon, donc à moins que l'ajout du caractère nul ne force le redimensionnement du tampon, la conversion ne nécessite aucune copie de texte ni aucune allocation.

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

Vous pouvez également déclarer des variables globales dans des `extern` blocs. Les systèmes POSIX ont une variable globale nommée

`environ` qui contient les valeurs des variables d'environnement du processus. En C, il est déclaré :

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

En 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 `environ` d'avoir un premier élément, le code appelle `CStr::from_ptr` pour construire un `CStr` qui l'emprunte. La `to_string_lossy` méthode renvoie a `Cow<str>` : si la chaîne C contient de l'UTF-8 bien formé, le `Cow` emprunte son contenu en tant que a `&str`, sans compter l'octet nul de fin. Sinon, `to_string_lossy` fait une copie du texte dans le tas, remplace les séquences UTF-8 mal formées par le caractère de remplacement Unicode officiel `&#xfffd;`, et crée un propriétaire `Cow` à partir de cela. Dans tous les cas, le résultat implémente `Display`, vous pouvez donc l'imprimer avec le `{}` paramètre format.

Utiliser les fonctions des bibliothèques

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

```
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 `extern` bloc déclare les fonctions externes comme précédemment. L' `#[link(name = "git2")]` 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 `git2` bibliothèque. Rust utilise l'éditeur de liens système pour créer des exécutables ; sous Unix, cela passe l'argument `-lgit2` sur la ligne de commande de l'éditeur de liens ; sous Windows, ça passe `git2.LIB` .

`#[link]` les attributs fonctionnent également dans les caisses de bibliothèque. Lorsque vous construisez 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 créer `libgit2` vous-même. Nous avons utilisé `libgit2` la version 0.25.1. Pour compiler `libgit2` , vous devrez installer l'outil de compilation CMake et le langage Python ; nous avons utilisé CMake version 3.8.0 et Python version 2.7.13.

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

```
$ 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 un nommé *libgit2.so* . Sur macOS, les résultats sont similaires, mais la bibliothèque est nommée *libgit2.dylib* .

Sous 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 :

```
> cdC:\Users\JimB\libgit2-0.25.1
 > mkdirbuild
 > cdbuild
 >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' `-A x64` indicateur de la première `cmake` commande.) Cela produit une bibliothèque d'importation *git2.LIB* et une bibliothèque de liens dynamiques *git2.DLL* , toutes 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.)

Créez le programme Rust dans un répertoire séparé :

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

Prenez le code montré précédemment et placez-le dans *src/main.rs* . Naturellement, si vous essayez de construire ceci, Rust n'a aucune idée d'où trouver ce `libgit2` que vous avez construit :

```
$ cd $course de fret
 git-toy
   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_shutdow
          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 construction*, Code de rouille que Cargocompile et s'exécute au moment de la construction. Les scripts de construction peuvent faire toutes sortes de choses : générer du code dynamiquement, compiler du code C à inclure dans le crate, etc. Dans ce cas, tout ce dont vous avez besoin est d'ajouter un chemin de recherche de bibliothèque à la commande de lien de l'exécutable. Lorsque Cargo exécute le script de construction, il analyse la sortie du script de construction à la recherche d'informations de ce type, de sorte que le script de construction a simplement besoin d'imprimer la bonne magie sur sa sortie standard.

Pour créer votre script de construction, 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/buil
}
```

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

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

```
$course de fret
   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 `LD_LIBRARY_PATH` variable d'environnement :

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

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

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

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug ; %PATH%
>course de fret
    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 dans votre crate. Cela copie les fichiers objets de la bibliothèque dans le fichier *.rlib* de la caisse , ainsi que les fichiers objets et les métadonnées du code Rust de la caisse. L'ensemble de la collection participe alors au lien final.

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

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

Une interface brute vers libgit2

Figurantcomment bien l'utiliser `libgit2` se décompose en deux questions :

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

Nous allons répondre à ces questions une par une. Dans cette section, nous allons écrire un programme qui est essentiellement un seul `unsafe` bloc géant rempli de code Rust non idiomatique, reflétant le conflit des systèmes de types et des conventions inhérent au mélange de langages. Nous l'appellerons l' interface *brute* . Le code sera désordonné, mais il expliquera clairement toutes les étapes qui doivent se produire pour que le code Rust utilise `libgit2` .

Ensuite, dans la section suivante, nous construirons une interface sécurisée `libgit2` qui utilisera les types de Rust en appliquant les règles `libgit2` imposées à ses utilisateurs. Heureusement, `libgit2` est 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 bonnes réponses, et nous pouvons construire une interface Rust idiomatique sans `unsafe` fonctions.

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 affiche le commit principal. Mais cela suffit pour illustrer les stratégies clés pour créer des interfaces Rust sûres et idiomatiques.

Pour l'interface brute, le programme finira par avoir besoin d'une collection de fonctions et de types un peu plus grande `libgit2` que celle que nous utilisions auparavant, il est donc logique de déplacer le `extern` bloc dans son propre module. Nous allons créer un fichier nommé *raw.rs* dans *git-toy/src* dont le contenu est le suivant :

```
#![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_signat
    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
}
```

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

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

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

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

Les `libgit2` fichiers d'en-tête publics définissent le `git_repository` type en tant que typedef pour un type de structure incomplet :

```
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 `struct git_repository`, garantissant que les utilisateurs de la bibliothèque ne peuvent jamais créer eux-mêmes une instance de ce type. Voici un analogue possible d'un type de structure incomplet dans Rust :

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

Il s'agit d'un type struct contenant un tableau sans éléments. Puisque le `_private` champ n'est pas `pub`, les valeurs de ce type ne peuvent pas être construites en dehors de ce module, qui est parfait en tant que reflet d'un type C qui ne `libgit2` devrait jamais être construit, et qui est manipulé uniquement par des pointeurs bruts.

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

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

```
mod raw;
```

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

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

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

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

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());
}
```

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

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

En termes Rust, `author` et `message` sont empruntés à `commit` : `show_commit` n'a pas besoin de les libérer lui-même, mais il ne doit pas les conserver après avoir `commit` été libéré. 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 saurons probablement pas avant que le programme ne plante.

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 chaînes impliquerait probablement l'utilisation de la `encoding` caisse. Par souci de brièveté, nous passerons sous silence ces questions ici.

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

```
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_na
                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 le code pour gérer l'argument path et initialiser la bibliothèque, ce que nous avons déjà vu. Le premier code roman est celui-ci :

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

L'appel à `git_repository_open` essaie d'ouvrir le dépôt Git au chemin donné. S'il réussit, il lui alloue un nouvel `git_repository` objet et `repo` pointe vers celui-ci. Rust contraint implicitement les références en pointeurs bruts, donc le passage `&mut repo` ici fournit `*mut *mut git_repository` l'appel attendu.

Cela montre une autre `libgit2` convention utilisée (à partir de la `libgit2` documentation):

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

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

Ensuite, considérez le code qui recherche le hachage d'objet du commit principal actuel 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 `git_oid` type stocke un identifiant d'objet, un code de hachage de 160 bits que Git utilise en interne (et dans son interface utilisateur conviviale) pour identifier les validations, les versions individuelles des fichiers, etc. Cet appel à `git_reference_name_to_id` recherche l'identifiant d'objet du `"HEAD"` commit en cours.

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 `git_reference_name_to_id` qu'il s'attend à traiter son premier argument. Mais Rust ne nous laissera pas emprunter une référence à une variable non initialisée. On pourrait initialiser `oid` avec des zéros, mais c'est du gâchis : toute valeur qui y est stockée sera simplement écrasée.

Il est possible de demander à Rust de nous donner de la mémoire non initialisée, mais comme la lecture de mémoire non initialisée à tout moment est un comportement instantané et indéfini, Rust fournit une abstraction, `MaybeUninit` , pour faciliter son utilisation. `MaybeUninit<T>` indique au compilateur de réserver suffisamment de mémoire pour votre type `T` , mais de ne pas y toucher jusqu'à ce que vous disiez que vous pouvez le faire en toute sécurité. Bien que cette mémoire appartienne à `MaybeUninit` , le compilateur évitera également certaines optimisations qui pourraient autrement provoquer un comportement indéfini même sans aucun accès explicite à la mémoire non initialisée dans votre code.

`MaybeUninit` fournit une méthode, `as_mut_ptr()` , qui produit un `*mut T` pointage vers la mémoire potentiellement non initialisée qu'elle encapsule. En transmettant ce pointeur à une fonction étrangère qui initialise la mémoire, puis en appelant la méthode unsafe `assume_init` sur le `MaybeUninit` pour produire un complètement initialisé `T` , vous pouvez éviter un comportement indéfini sans la surcharge supplémentaire résultant de l'initialisation et de la suppression immédiate d'une valeur. `assume_init` n'est pas sûr car l'appeler sur a `MaybeUninit` sans être certain que la mémoire est réellement initialisée provoquera immédiatement un comportement indéfini.

Dans ce cas, il est sûr car `git_reference_name_to_id` initialise la mémoire appartenant au `MaybeUninit` . Nous pourrions également utiliser `MaybeUninit` pour les variables `repo` et , mais comme ce ne sont que

des mots simples, nous allons simplement de l'avant et les initialisons à null : `commit`

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

Cela prend l'identifiant d'objet du commit et recherche le commit réel, en stockant un `git_commit` pointeur en cas `commit` de succès.

Le reste de la `main` fonction devrait être explicite. Il appelle la `show_commit` fonction définie précédemment, libère les objets de validation et de référentiel et arrête la bibliothèque.

Maintenant, nous pouvons essayer le programme sur n'importe quel référentiel 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 pour libgit2

L'interface bruteto `libgit2` est un exemple parfait d'une fonctionnalité non sécurisée : 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. Concevoir une API sûre 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.

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

- Vous devez appeler `git_libgit2_init` avant d'utiliser toute autre fonction de la bibliothèque. Vous ne devez utiliser aucune fonction de bibliothèque après avoir appelé `git_libgit2_shutdown` .
- Toutes les valeurs transmises aux `libgit2` fonctions doivent être entièrement initialisées, à l'exception des paramètres de sortie.
- Lorsqu'un appel échoue, les paramètres de sortie transmis pour conserver les résultats de l'appel ne sont pas initialisés et vous ne de-

vez pas utiliser leurs valeurs.

- Un `git_commit` objet fait référence à l' `git_repository` objet dont il est dérivé, de sorte que le premier ne doit pas survivre au second. (Ceci n'est pas précisé dans la `libgit2` documentation ; nous l'avons déduit de la présence de certaines fonctions dans l'interface, puis nous l'avons vérifié en lisant le code source.)
- De même, a `git_signature` est toujours emprunté à un donné `git_commit` , et le premier ne doit pas survivre au second. (La documentation couvre ce cas.)
- Le message associé à un commit ainsi que le nom et l'adresse e-mail de l'auteur sont tous empruntés au commit et ne doivent pas être utilisés après la libération du commit.
- Une fois qu'un `libgit2` objet a été libéré, il ne doit plus jamais être utilisé.

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

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

L'ensemble de l'arborescence des sources 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 in Separate Files" , la source du `git` module apparaît dans *git/mod.rs* et la source de son `git::raw` sous-module va dans *git/raw.rs* .

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

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

```
mod raw;
```

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

Dans quelques instants, nous aurons besoin d'utiliser certaines fonctions de la `libc` caisse, nous devons donc ajouter une dépendance dans *Cargo.toml* . Le fichier complet lit maintenant :

```
[forfait]
nom = "git-jouet"
version = "0.1.0"
auteurs = ["Vous <vous@exemple.com>"]
édition = "2021"

[dépendances]
libc = "0.2"
```

Maintenant que nous avons restructuré nos modules, considérons la gestion des erreurs. Même `libgit2` la fonction d'initialisation de peut renvoyer un code d'erreur, nous devrons donc régler ce problème avant de pouvoir commencer. Une interface Rust idiomatique a besoin de son propre `Error` typequi capture le `libgit2` code d'échec ainsi que le message d'erreur et la classe de `giterr_last` . Un type d'erreur approprié doit implémenter les traits habituels `Error` , `Debug` et . `Display` Ensuite, il a besoin de son propre `Result` type qui utilise ce `Error` type. Voici les définitions nécessaires dans *src/git/mod.rs* :

```
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::Error
        // 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 bruts à la bibliothèque, le module a besoin d'une fonction qui transforme un `libgit2` code de retour en un `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 null
        // 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 ceci et la `check` fonction de la version brute est que cela construit une `Error` valeur au lieu d'afficher un message d'erreur et de quitter immédiatement.

Nous sommes maintenant prêts à nous attaquer à l' `libgit2` initialisation. L'interface sécurisée fournira un `Repository` type qui représente un référentiel Git ouvert, avec des méthodes pour résoudre les références, rechercher des commits, etc. En continuant dans *git/mod.rs* , voici la définition de `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 `Repository` de A n'est `raw` pas public. Étant donné que seul le code de ce module peut accéder au `raw::git_repository` pointeur, obtenir ce module correctement devrait garantir que le pointeur est toujours utilisé correctement.

Si la seule façon de créer un `Repository` est d'ouvrir avec succès un nouveau référentiel Git, cela garantira que chacun `Repository` pointe vers un objet distinct `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 `Repository` valeur, et `Repository::open` commence par un appel à `ensure_initialized` , nous pouvons être sûrs que `ensure_initialized` sera appelé avant toute `libgit2` fonction. Sa définition est la suivante :

```
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 `std::sync::Once` type permet d'exécuter le code d'initialisation de manière thread-safe. Seul le premier thread à appeler `ONCE.call_once` exécute la fermeture donnée. Tous les appels suivants, par ce thread ou tout autre, 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 `ONCE.call_once` est bon marché, ne nécessitant rien de plus qu'une charge atomique d'un indicateur stocké dans `ONCE` .

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

Pour s'assurer que le programme appelle `git_libgit2_shutdown` , la fermeture d'initialisation utilise la `atexit` fonction de la bibliothèque C, qui prend un pointeur vers une fonction à invoquer avant que le processus ne se termine. Les fermetures Rust 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 y fait référence ; un pointeur de fonction C n'est qu'un pointeur. Cependant, `fn` les types Rust fonctionnent bien, tant que vous les déclarez `extern` afin que Rust sache utiliser les conventions d'appel C. La fonction locale fait `shutdown` l'affaire et garantit une `libgit2` fermeture correcte.

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

Il pourrait être possible de s'arranger pour appeler `git_libgit2_shutdown` plus tôt, par exemple, lorsque la dernière `Repository` valeur est supprimée. Mais peu importe comment nous organisons les choses, l'appel `git_libgit2_shutdown` doit être la responsabilité de l'API sécurisée. Au moment où elle est appelée, tous les objets existants `libgit2` deviennent dangereux à utiliser, donc une API sûre ne doit pas exposer directement cette fonction.

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

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

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

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

```
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 path
    // 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' `libgit2` interface rend ce code un peu délicat. Sur toutes les plates-formes, `libgit2` accepte les chemins en tant que chaînes C terminées par un caractère nul. Sous Windows, `libgit2` suppose que ces chaînes C contiennent de l'UTF-8 bien formé et les convertit en interne en chemins 16 bits dont Windows a réellement besoin. Cela fonctionne généralement, mais ce n'est pas idéal. Windows autorise les noms de fichiers qui ne sont pas bien formés en 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` .

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

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

```
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 }
    }
}
```

Voyons ensuite comment résoudre une référence Git en un identifiant d'objet. Puisqu'un identifiant d'objet n'est qu'une valeur de hachage de 20 octets, il est parfaitement acceptable 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 `Repository` pour effectuer la recherche :

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

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

```
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 `git_commit` objet ne doit jamais survivre à l' `git_repository` objet à partir duquel il a été récupéré. Les durées de vie de Rust permettent au code de capturer précisément cette règle.

L' `RefWithFlag` exemple précédent dans ce chapitre utilisait un `PhantomData` 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 `Commit` type doit faire quelque chose de similaire. Dans ce cas, le `_marker` type du champ est `PhantomData<&'repo Repository>` , indiquant que Rust doit traiter `Commit<'repo>` comme s'il contenait une référence avec une durée de vie `'repo` à certains `Repository` .

La méthode pour rechercher un commit 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 `Commit` durée de vie de 's à celle de `Repository` 's? La signature de `find_commit` omet les durées de vie des références concernées conformément aux règles décrites dans « Omettre les paramètres de durée de vie » . Si nous devions écrire les durées de vie, la signature complète se lirait :

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

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

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

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

```
        }
    }
```

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

```
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 `Signature` genre :

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

Un `git_signature` objet emprunte toujours son texte ailleurs ; en particulier, les signatures renvoyées par `git_commit_author` empruntent leur texte au `git_commit`. Ainsi, notre type de sécurité `Signature` inclut a `PhantomData<&'text str>` pour dire à Rust de se comporter comme s'il contenait un `&str` avec une durée de vie de `'text`. Comme précédemment, `Commit::author` relie bien cette `'text` durée de vie du `Signature` il revient à celle du `Commit` sans qu'on ait besoin d'écrire quoi que ce soit. La `Commit::message` méthode fait de même avec le `Option<&str>` maintien du message de validation.

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

```
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 d'utilité 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<&str>
        if ptr.is_null() {
            return None;
        } else {
            CStr::from_ptr(ptr).to_str().ok()
        }
    }
```

La `_owner` 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 :

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

La `CStr::from_ptr` fonction renvoie un `&CStr` dont la durée de vie est totalement illimitée, puisqu'il a été emprunté 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 `_owner` paramètre oblige Rust à attribuer sa durée de vie au type de la valeur de retour, afin que les appelants puissent recevoir une référence délimitée plus précisément.

Il n'est pas clair d'après la `libgit2` documentation si un `git_signature` 's `email` et des `author` pointeurs peuvent être nuls, bien que la documentation `libgit2` soit assez bonne. Vos auteurs ont creusé dans le code source pendant un certain temps sans pouvoir se persuader d'une manière ou d'une autre et ont finalement décidé qu'il `char_ptr_to_str` valait mieux se préparer aux pointeurs nuls au cas où. Dans Rust, ce genre de question est répondu immédiatement par le type : si c'est `&str` , vous pouvez compter sur la chaîne pour être là ; si c'est le cas `Option<&str>` , c'est facultatif.

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

```
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 n'offrent pas beaucoup de garanties de sécurité à une API sûre enveloppant une API intrinsèquement non sûre 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 vous assurer d'utiliser correctement. Pour la plupart, les règles que nous avons imposées à Rust sont le genre de règles que les programmeurs C et C++ finissent par s'imposer de toute façon. Ce qui rend Rust tellement plus strict que C et C++, ce n'est pas que les règles soient si étrangères, mais que cette application soit mécanique et complète..

Conclusion

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

Les contours de la langue sont déterminés par ces buts. Rust parvient à combler la majeure partie de l'écart avec un code sécurisé. Son vérificateur d'emprunt et ses abstractions à coût zéro 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 non sécurisé et l'interface de fonction étrangère sont prêts. Mais encore une fois, le langage ne se contente pas de vous offrir ces fonctionnalités dangereuses et vous souhaite bonne chance. L'objectif est toujours d'utiliser des fonctionnalités non sécurisées pour créer des API sécurisées. C'est ce qu'on a fait avec `libgit2`. C'est aussi ce que l'équipe Rust a fait avec `Box`, `Vec`, les autres collections, canaux, etc. : la bibliothèque standard regorge d'abstractions sûres, implémentées avec du code dangereux en coulisses.

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 de grands systèmes rapides, sécurisés et robustes qui tirent parti de toute la puissance du matériel sur lequel ils s'exécutent. Utilisez-le pour améliorer le logiciel.