Using Rust code in R packages

The rextendr package provides two utility functions for developing R packages with Rust code using extendr:

- `rextendr::use_extendr()`: create the scaffolding to use extendr, similar to `usethis::use_cpp11()`.
- `rextendr::document()`: compile Rust code and generate package documentation, similar to `devtools::document()`.

One thing we want to emphasize here is that these functions are needed solely for package development. An R package using extendr Rust code doesn't have to depend on or import the rextendr package, just like R packages don't usually add the devtools package to `Depends` or `Imports` no matter how often the package developers use the functions provided by devtools.

Create a template package

Creating an R package with extendr is very easy with the rextendr package.

First, create an empty R package. You can do this with `usethis::create_package()`. Let's pick `myextendr` as the package name.

```
usethis::create_package("path/to/myextendr")
```

Then, execute `rextendr::use_extendr()` inside the package directory to create the scaffolding to use extendr.

```
rextendr::use_extendr()
#> ✓ Creating src/rust/src.
#> ✓ Setting active project to 'path/to/myextendr'
#> ✓ Writing 'src/entrypoint.c'
#> ✓ Writing 'src/Makevars'
#> ✓ Writing 'src/Makevars.win'
#> ✓ Writing 'src/.gitignore'
#> ✓ Writing src/rust/Cargo.toml.
#> ✓ Writing 'src/rust/src/lib.rs'
#> ✓ Writing 'R/extendr-wrappers.R'
#> ✓ Finished configuring extendr for package myextendr.
#> • Please update the system requirement in DESCRIPTION file.
#> • Please run `rextendr::document()` for changes to take effect.
```

Now we are just one step away from calling Rust functions from R. As the message says, we need to run `rextendr::document()`. But, before moving forward, let's look at the files added.

Package structure

The following files have been added by `rextendr::use_extendr()`:

```
.
├── R
│   └── extendr-wrappers.R
...
└── src
    ├── Makevars
    ├── Makevars.win
    ├── entrypoint.c
    └── rust
        ├── Cargo.toml
        └── src
            └── lib.rs
```

- **`R/extendr-wrappers.R`**: This file contains auto-generated R functions from Rust code. We don't modify this file by hand.
- **`src/Makevars`**, **`src/Makevars.win`**: These files hook `cargo build` at the installation of the R package. In most cases, we don't edit these.
- **`src/entrypoint.c`**: This file is needed to avoid the linker removing the static library. In 99.9% of cases, we don't edit this (except for changing the crate name).
- **`src/rust/`**: Rust code of a crate using extendr-api. This is where we mainly write code.

So, in short, what we should really look at is only these two files:

`src/rust/Cargo.toml`

```
[package]
name = 'myextendr'
version = '0.1.0'
edition = '2018'

[lib]
crate-type = [ 'staticlib' ]

[dependencies]
extendr-api = '*'
```

The crate name is the same name as the R package's name by default. You can change this, but it might be a bit cumbersome to tweak other files accordingly, so we recommend leaving this.

You will probably want to specify a concrete extendr version, for example `extendr-api = '0.2'`. To try the development version of the extendr, you can modify the last line to read

```
extendr-api = { git = 'https://github.com/extendr/extendr' }
```

`src/rust/src/lib.rs`

```
use extendr_api::prelude::*;

/// Return string `"Hello world!"` to R.
/// @export
#[extendr]
fn hello_world() -> &'static str {
    "Hello world!"
}

// Macro to generate exports.
// This ensures exported functions are registered with R.
// See corresponding C code in `entrypoint.c`.
extendr_module! {
    mod myextendr;
    fn hello_world;
}
```

Let's explain this file line by line.

The first line, containing the `use` statement, declares the commonly used extendr API functions to the Rust compiler.

```
use extendr_api::prelude::*;
```

Next, you may notice that `/` is repeated three times, while the usual Rust comments require only two slashes (i.e., `//`). This is one of Rust's "doc comment" notation to generate the crate's documentation. In extendr, these lines are copied to the auto-generated R code as roxygen comments. This is analogous to Rcpp/cpp11's `//'`.

```
/// Return string `"Hello world!"` to R.
/// @export
```

The next line is the core of extendr's mechanism. If the function is marked with this macro, the corresponding R function will be generated automatically. This is analogous to Rcpp's `[[Rcpp::export]]` and cpp11's `[[cpp11::register]]`.

```
#[extendr]
```

The last 3 lines are the macro for generating exports, as the comment explains. If we implement another function than just `hello_world`, it needs to be listed here as well as marking it with `#[extendr]` macro.

```
// Macro to generate exports.
// This ensures exported functions are registered with R.
// See corresponding C code in `entrypoint.c`.
extendr_module! {
    mod myextendr;
    fn hello_world;
}
```

Compile and use the package

Compile

Compiling Rust code into R functions is as easy as executing this one command:

```
rextendr::document()
#> ✓ Saving changes in the open files.
#> i Generating extendr wrapper functions for package: myextendr.
#> ! No library found at src/myextendr.so, recompilation is required.
#> Re-compiling myextendr
#> ─  installing *source* package 'myextendr' ... (347ms)
#>    ** using staged installation
#>    ** libs
#>    rm -Rf myextendr.so ./rust/target/release/libmyextendr.a entrypoint.o
#>    gcc -std=gnu99 -I"/usr/share/R/include" -DNDEBUG      -fpic  -g -O2 -fdebug-prefix-
      map=/build/r-base-tbZjLv/r-base-4.1.0=. #> -fstack-protector-strong -Wformat -
      Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -g  -UNDEBUG -Wall -pedantic -g
      -O0 #> -fdiagnostics-color=always -c entrypoint.c -o entrypoint.o
#>    cargo build --lib --release --manifest-path=./rust/Cargo.toml
#>        Updating crates.io index
#>       Compiling proc-macro2 v1.0.27
#>       Compiling unicode-xid v0.2.2
#>       Compiling libR-sys v0.2.1
#>       Compiling syn v1.0.72
#>       Compiling extendr-engine v0.2.0
#>       Compiling lazy_static v1.4.0
#>       Compiling quote v1.0.9
#>       Compiling extendr-macros v0.2.0
#>       Compiling extendr-api v0.2.0
#>       Compiling myextendr v0.1.0 (path/to/myextendr/src/rust)
#>        Finished release [optimized] target(s) in 19.05s
#>    gcc -std=gnu99 -shared -L/usr/lib/R/lib -Wl,-Bsymbolic-functions -Wl,-z,relro -o
      myextendr.so entrypoint.o -L./rust/target/release #> -lmyextendr -L/usr/lib/R/lib -lR
#>    installing to /tmp/RtmpfMcL08/devtools_install_e2d6351b843c/00LOCK-
      myextendr/00new/myextendr/libs
#>    ** checking absolute paths in shared objects and dynamic libraries
#> ─  DONE (myextendr)
#> ✓ Writing 'R/extendr-wrappers.R'.
#> i Updating myextendr documentation
#> i Loading myextendr
#> Writing NAMESPACE
#> Writing NAMESPACE
#> Writing hello_world.Rd
```

You might wonder why compilation is triggered while the function name is just `document()`. Well, this is because the compilation is actually needed to generate documentation and R wrapper code from the Rust code. This is consistent with the behavior of `devtools::document()` for packages using C or C++.

By doing the above, the following files are updated or generated:

```
.
...
├── NAMESPACE                       ----------(4)
├── R
│   └── extendr-wrappers.R          ----------(3)
├── man
│   └── hello_world.Rd              ----------(4)
└── src
    ├── myextendr.so          ----------(2)
    └── rust
        └── target
            └── release
                ├── libmyextendr.a   ---(1)
                ...
```

1. **`src/rust/target/release/libmyextendr.a`** (the extension depends on the OS): This is the static library built from Rust code. This will be then used for compiling the shared object `myextendr.so`.
2. **`src/myextendr.so`** (the extension depends on the OS): This is the shared object that is actually called from R.
3. **`R/extendr-wrappers.R`**: The auto-generated R functions, including roxygen comments, go into this file. The roxygen comments are accordingly converted into Rd files and `NAMESPACE`.
4. **`man/`**, **`NAMESPACE`**: These are generated from roxygen comments.

Load and use

After running `rextendr::document()`, we can just load the package with `devtools::load_all()` (or alternatively install it and call with `library()`) and then call the function we have implemented in Rust.

```
devtools::load_all(".")

hello_world()
#> [1] "Hello world!"
```

Rust code vs generated R code

While we never edit the R wrapper code by hand, it might be good to know what R code is generated from the Rust code. Let's look at `R/extendr-wrappers.R`:

```
# Generated by extendr: Do not edit by hand
#
# This file was created with the following call:
#   .Call("wrap__make_myextendr_wrappers", use_symbols = TRUE, package_name = "myextendr")

#' @docType package
#' @usage NULL
#' @useDynLib myextendr, .registration = TRUE
NULL

#' Return string `"Hello world!"` to R.
#' @export
hello_world <- function() .Call(wrap__hello_world)
```

Here, `.Call("wrap__make_myextendr_wrappers", use_symbols = ...` is a function call that was executed by `rextendr::document()`.

A section of `@docType package` is needed to generate the `useDynLib(myextendr, .registration = TRUE)` entry in `NAMESPACE`.

The last section is for `hello_world()`. We can see the roxygen comments are copied to here. As the Rust function `hello_world()` has no arguments this R function also has no arguments. If the function had arguments, such as

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

then the generated function wrapper also would have arguments:

```
add <- function(x, y) .Call(wrap__add, x, y)
```

Implement a new Rust function

Now that we have roughly figured out how extendr works, let's implement a new Rust function. The development flow would be:

1. Modify `src/rust/src/lib.rs`
2. Run `rextendr::document()`
3. Run `devtools::load_all(".")` and test the function

As an exercise, let's add the `add(i32, i32)` function from the previous subsection.

1. Modify `src/rust/src/lib.rs`

Add the function with `@export`, so it will get exported from the generated R package. (Without this tag, the function would be available internally for package programming but not externally to users of the package.)

```
/// @export
#[extendr]
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Don't forget to add the function to `extendr_module!`:

```
extendr_module! {
    mod myextendr;
    fn hello_world;
    fn add;
}
```

2. Run `rextendr::document()`

Just run this command:

```
rextendr::document()
```

3. Run `devtools::load_all(".")` and test the function

Now you can load the package and call `add()`:

```
devtools::load_all(".")
```

```
add(11, 25)
# => 36
```