Diplomat

Polyglot tool to use rust from other languages

Jan Cristina

2025 April 1

About me

Day Job: Head of AI at Starmind. Mostly work in Scala and Python. Some Typescript. Have been able toinject some rust code. More to come.

Rust experience: Started learning in 2020, been programming on the side since then.

What I like about rust: Speed is nice, but correctness is better, e.g. resources are cleaned up when they go out of scope. No accidental mutation. Other things are things I like in Scala too: ADTs, exhaustive matches, functional patterns for collections/iterators.

github: `@jcrist1`

mastodon: `@gigapixel@mathstodon.xyz`

What is diplomat?

What is diplomat?

A tool to create bindings in other languages for rust code.

What is diplomat?

A tool to create bindings in other languages for rust code.

Restricts to a certain subset of rust code, e.g. sum types / ADTs are not supported.

What is diplomat?

A tool to create bindings in other languages for rust code.

Restricts to a certain subset of rust code, e.g. sum types / ADTs are not supported.

General philosophy is write Rust, and be able to integrate it in other projects.
This is for unidirectional FFI: oher code calls rust

What is diplomat?

A tool to create bindings in other languages for rust code.

Restricts to a certain subset of rust code, e.g. sum types / ADTs are not supported.

General philosophy is write Rust, and be able to integrate it in other projects. This is for unidirectional FFI: oher code calls rust

Currently have backends in C, C++, JS, Dart, Kotlin (JVM), very experimental support for Java via Panama FFI

What is diplomat?

A tool to create bindings in other languages for rust code.

Restricts to a certain subset of rust code, e.g. sum types / ADTs are not supported.

General philosophy is write Rust, and be able to integrate it in other projects. This is for unidirectional FFI: oher code calls rust

Currently have backends in C, C++, JS, Dart, Kotlin (JVM), very experimental support for Java via Panama FFI

As with any FFI, important to consider performance considerations. It’s not automatically write rust and go vrrroooom

Who came up with it

Who came up with it

A lot of the motivation comes from icu4x, which is a tool for internationalization.

Who came up with it

A lot of the motivation comes from icu4x, which is a tool for internationalization.

Diplomat allows ergonomic integration of code like icu4x in many other languages.

Who came up with it

A lot of the motivation comes from icu4x, which is a tool for internationalization.

Diplomat allows ergonomic integration of code like icu4x in many other languages.

`@manishearth` is the primary maintainer, but several other active contributors. I'm quite far down on the list, with main contribution being the Kotlin backend

Alternatives

Alternatives

interoptopus - very similar. Backends for C, C#, Python

Alternatives

interoptopus - very similar. Backends for C, `C#`, Python

UniFFI – Does the same kind of thing, but interface specification is in a dedicated language.

Alternatives

interoptopus - very similar. Backends for C, `C#`, Python

UniFFI – Does the same kind of thing, but interface specification is in a dedicated language.

Language specific tools: PyO3, Napi

How does it work?

How does it work?

Create some kind a (cdy-) lib project, with diplomat, and diplomat-runtime as a depencies

How does it work?

Create some kind a (cdy-) lib project, with diplomat, and diplomat-runtime as a depencies

Write some rust code with special annotations

How does it work?

Create some kind a (cdy-) lib project, with diplomat, and diplomat-runtime as a depencies

Write some rust code with special annotations

As with any FFI, important to consider performance considerations. Not automatically write rust and go vrrroooom

What to do on the rust side

Let’s work through a simple example:

```
#[diplomat::bridge]
pub mod ffi {









}
```

What to do on the rust side

Let's work through a simple example:

```
#[diplomat::bridge]
pub mod ffi {
    #[diplomat::opaque]
    pub struct Wrapper(String);









}
```

What to do on the rust side

Let's work through a simple example:

```
#[diplomat::bridge]
pub mod ffi {
    #[diplomat::opaque]
    pub struct Wrapper(String);

    impl Wrapper {
        pub fn new() -> Box<Wrapper> {
            Box::new(Wrapper(String::new()))
        }
    }
}
```

What to do on the rust side

Let's work through a simple example:

```
#[diplomat::bridge]
pub mod ffi {
    #[diplomat::opaque]
    pub struct Wrapper(String);

    impl Wrapper {
        pub fn new() -> Box<Wrapper> {
            Box::new(Wrapper(String::new()))
        }
    }
}
```

What do we get on the other side?

For an opaque type we generate a class. This generated class will wrap the `<Wrapper>` returned from the native code, i.e. it holds onto a pointer:

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}

class Wrapper internal constructor (
    internal val handle: Pointer,
    // These ensure that anything that is borrowed is kept alive and not cleaned
    // up by the garbage collector.
    internal val selfEdges: List<Any>,
)  {

    internal class WrapperCleaner(val handle: Pointer, val lib: WrapperLib) : Runnable {
        override fun run() {
            lib.Wrapper_destroy(handle)
        }
    }

    companion object {
```

```
        ...
    }
}
```

What do we get on the other side?

For an opaque type we generate a class. This generated class will wrap the `<Wrapper>` returned from the native code, i.e. it holds onto a pointer:

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}

class Wrapper internal constructor (
    internal val handle: Pointer,
    // These ensure that anything that is borrowed is kept alive and not cleaned
    // up by the garbage collector.
    internal val selfEdges: List<Any>,
)  {

    internal class WrapperCleaner(val handle: Pointer, val lib: WrapperLib) : Runnable {
        override fun run() {
            lib.Wrapper_destroy(handle)
        }
    }

    companion object {
```

```
        ...
    }
}
```

What do we get on the other side?

For an opaque type we generate a class. This generated class will wrap the `<Wrapper>` returned from the native code, i.e. it holds onto a pointer:

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}

class Wrapper internal constructor (
    internal val handle: Pointer,
    // These ensure that anything that is borrowed is kept alive and not cleaned
    // up by the garbage collector.
    internal val selfEdges: List<Any>,
) {

    internal class WrapperCleaner(val handle: Pointer, val lib: WrapperLib) : Runnable {
        override fun run() {
            lib.Wrapper_destroy(handle)
        }
    }

    companion object {
```

```
        ...
    }
}
```

What do we get on the other side?

For an opaque type we generate a class. This generated class will wrap the `<Wrapper>` returned from the native code, i.e. it holds onto a pointer:

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}

class Wrapper internal constructor (
    internal val handle: Pointer,
    // These ensure that anything that is borrowed is kept alive and not cleaned
    // up by the garbage collector.
    internal val selfEdges: List<Any>,
)  {

    internal class WrapperCleaner(val handle: Pointer, val lib: WrapperLib) : Runnable {
        override fun run() {
            lib.Wrapper_destroy(handle)
        }
    }

    companion object {
```

```
        ...
    }
}
```

What do we get on the other side?

For an opaque type we generate a class. This generated class will wrap the `<Wrapper>` returned from the native code, i.e. it holds onto a pointer:

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}

class Wrapper internal constructor (
    internal val handle: Pointer,
    // These ensure that anything that is borrowed is kept alive and not cleaned
    // up by the garbage collector.
    internal val selfEdges: List<Any>,
)  {

    internal class WrapperCleaner(val handle: Pointer, val lib: WrapperLib) : Runnable {
        override fun run() {
            lib.Wrapper_destroy(handle)
        }
    }

    companion object {
```

```
        ...
    }
}
```

What do we get on the other side?

For an opaque type we generate a class. This generated class will wrap the `<Wrapper>` returned from the native code, i.e. it holds onto a pointer:

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}

class Wrapper internal constructor (
    internal val handle: Pointer,
    // These ensure that anything that is borrowed is kept alive and not cleaned
    // up by the garbage collector.
    internal val selfEdges: List<Any>,
)  {

    internal class WrapperCleaner(val handle: Pointer, val lib: WrapperLib) : Runnable {
        override fun run() {
            lib.Wrapper_destroy(handle)
        }
    }

    companion object {
```

```
        ...
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```

```
internal interface WrapperLib: Library {
    fun Wrapper_destroy(handle: Pointer)
    fun Wrapper_new(): Pointer
}
...
companion object {
    internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
    internal val lib: WrapperLib = Native.load("somelib", libClass)

    fun new_(): Opaque {

        val returnVal = lib.Opaque_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Opaque(handle, selfEdges)
        CLEANER.register(returnOpaque, Opaque.OpaqueCleaner(handle, Opaque.lib));
        return returnOpaque
    }
}
```