Diplomat

Polyglot tool to use rust from other languages

Jan Cristina

2025 May 27

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`

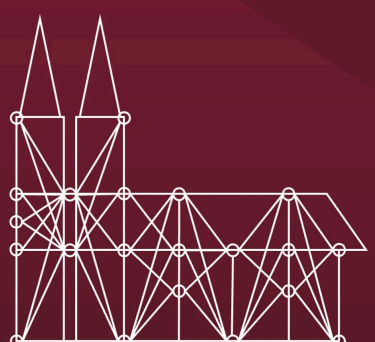
Talk src is available `https://github.com/jcrist1/diplomat-talk`

“== Small advertisement”

Basel Data Science & AI Meetup – June 2025

Wednesday 25.6.2025
18:30

Ascent | Güterstrasse 144, Basel



Max Buckley
Senior Software Engineer | Google

The Road to Robust RAG: Overcoming Risks, Limitations, and Challenges

Jan Cristina
Head of AI | Starmind

Speeding up your data processing in Python with Rust and PyO3

Armin Fanzott
Head of AI & Data Science | Ascent DACH

Large-Scale Forecasting with Machine Learning: Lessons from a real-world system

Hosted & supported by: ASCENT ML ARCHITECTS BASEL

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: other 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: other 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: other 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 think about 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.

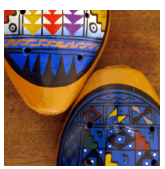
`@manishearth` is the primary maintainer, but several other active contributors. I'm quite far down on the list, with main contribution being the Kotlin backend

Who came up with it

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

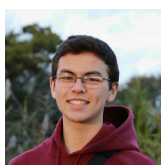
Diplomat allows ergonomic integration of code like icu4x in many other languages.

`@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



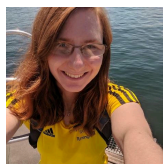
`@manishearth` and `@sffc` project and `C++` backend

`@robertbastian` Dart backend

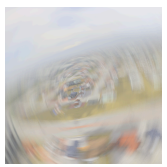


`@qnnokabayashi` and `@shadaj` initial setup

`@ambiguousname` javascript backend



`@emarteca` and `@jcrist1` (me) Kotlin backend



`@walter-reactor` Python-nanobind 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-rs

How does it work?

How does it work?

Create a dynamic library project, with `diplomat`, and `diplomat-runtime` as a depencies

How does it work?

Create a dynamic library project, with `diplomat`, and `diplomat-runtime` as a depencies

Write some rust code with special annotations

How does it work?

Create a dynamic library project, with `diplomat`, and `diplomat-runtime` as a depencies

Write some rust code with special annotations

Compile library artifact, and run code generation with `diplomat-tool`

How does it work?

Create a dynamic library project, with `diplomat`, and `diplomat-runtime` as a depencies

Write some rust code with special annotations

Compile library artifact, and run code generation with `diplomat-tool`

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

How does FFI work in rust?

Rust supports the C ABI with `extern C` annotations. In the end all FFI uses this

How does FFI work in rust?

Rust supports the C ABI with `extern C` annotations. In the end all FFI uses this

```
#[no_mangle]
extern "C" fn some_function() -> const * CVoid {
    ...
}
```

What to do on the rust side

Let’s work through a simple example:

```
}
```

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()))
        }
    }
}
```

```rust
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2024::*;
#[macro_use]
extern crate std;
pub mod ffi {
    use diplomat_runtime::DiplomatStr;
    pub struct Wrapper(String);
    impl Wrapper {
        pub fn new() -> Box<Wrapper> {
            Box::new(Wrapper(String::new()))
        }
```

```rust
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2024::*;
#[macro_use]
extern crate std;
pub mod ffi {
    use diplomat_runtime::DiplomatStr;
    pub struct Wrapper(String);
    impl Wrapper {
        pub fn new() -> Box<Wrapper> {
            Box::new(Wrapper(String::new()))
        }
    }
    use diplomat_runtime::*;
    use core::ffi::c_void;
    #[no_mangle]
    extern "C" fn Wrapper_new() -> Box<Wrapper> {
        Wrapper::new()
    }
    #[no_mangle]
    extern "C" fn Wrapper_destroy(this: Box<Wrapper>) {}
}
```

```rust
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2024::*;
#[macro_use]
extern crate std;
pub mod ffi {
    use diplomat_runtime::DiplomatStr;
    pub struct Wrapper(String);
    impl Wrapper {
        pub fn new() -> Box<Wrapper> {
            Box::new(Wrapper(String::new()))
        }
    }
    use diplomat_runtime::*;
    use core::ffi::c_void;
    #[no_mangle]
    extern "C" fn Wrapper_new() -> Box<Wrapper> {
        Wrapper::new()
    }
    #[no_mangle]
    extern "C" fn Wrapper_destroy(this: Box<Wrapper>) {}
}
```

Then we run our command in the shell

```
diplomat-tool -e rust/src/lib.rs -c rust/config.toml kotlin kotlin/
```

where our `config.toml` contains some basic config for the project.

Much like a procedural macro (using syn) the tool parses the code into a high level intermediate representation (HIR). This represents the structs, opaque types, enums, slices, and functions.

Much like a procedural macro (using syn) the tool parses the code into a high level intermediate representation (HIR). This represents the structs, opaque types, enums, slices, and functions.

The HIR is what allows for the diplomat to be polyglot. It is a “least common denominator” of functionality that can then be shared between language backends.

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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.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_(): Wrapper {
        val returnVal = lib.Wrapper_new();
        val selfEdges: List<Any> = listOf()
        val handle = returnVal
        val returnOpaque = Wrapper(handle, selfEdges)
        CLEANER.register(returnOpaque, Wrapper.WrapperCleaner(handle, Wrapper.lib));
        return returnOpaque
    }
}
```

Now let's add the kind of function we might actually want to use

```
#[allow(clippy::needless_lifetimes)]
pub fn return_inner<'a>(&'a self) -> &'a DiplomatStr {
    self.0.as_bytes()
}
```

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

class Wrapper internal constructor (
    internal val handle: Pointer,
    internal val selfEdges: List<Any>,
)  {
    companion object {
        internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
        internal val lib: WrapperLib = Native.load("somelib", libClass)
        ...
    }

    fun returnInner(): String {
        val returnVal = lib.Wrapper_return_inner(handle);
            return PrimitiveArrayTools.getUtf8(returnVal)
    }
}
```

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

class Wrapper internal constructor (
    internal val handle: Pointer,
    internal val selfEdges: List<Any>,
)  {
    companion object {
        internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
        internal val lib: WrapperLib = Native.load("somelib", libClass)
        ...
    }

    fun returnInner(): String {
        val returnVal = lib.Wrapper_return_inner(handle);
            return PrimitiveArrayTools.getUtf8(returnVal)
    }
}
```

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

class Wrapper internal constructor (
    internal val handle: Pointer,
    internal val selfEdges: List<Any>,
)  {
    companion object {
        internal val libClass: Class<WrapperLib> = OpaqueLib::class.java
        internal val lib: WrapperLib = Native.load("somelib", libClass)
        ...
    }

    fun returnInner(): String {
        val returnVal = lib.Wrapper_return_inner(handle);
            return PrimitiveArrayTools.getUtf8(returnVal)
    }
}
```

Where the `getUtf8` method is a standard function which is packaged in the library

```
fun getUtf8(slice: Slice): String {
    val byteArray = slice.data.getByteArray(0, slice.len.toInt())

    return byteArray.decodeToString()
}
```

How does this work with GC

- support several GC'd backends: Kotlin, JS, Dart
- Rust side handles allocation and deallocation
- GC needs to call the destroy function
- In JVM, the data is stored on the native heap and not JVM heap (important to keep in mind when allocating memory in say K8S)

Why should you use diplomat?

- Write once and share code between backends. Same API across backends.
- Want to benefit from native performance (especially things like vectorisation)
- Want to expose useful and performant Rust libraries

What is it really for 😐 ?

Let’s see what it looks like in javascript. First lets look at the typescript definitions

```
// generated by diplomat-tool
import type { pointer, codepoint } from "./diplomat-runtime.d.ts";



export class Wrapper {
    get ffiValue(): pointer;


    static new_(): Wrapper;

    returnInner(): string;

}
```

And the actual code looks like:

```
// generated by diplomat-tool
import wasm from "./diplomat-wasm.mjs";
import * as diplomatRuntime from "./diplomat-runtime.mjs";

const Wrapper_box_destroy_registry = new FinalizationRegistry((ptr) => {
    wasm.Wrapper_destroy(ptr);
});

export class Wrapper {
    // Internal ptr reference:
    #ptr = null;

    // Lifetimes are only to keep dependencies alive.
    // Since JS won't garbage collect until there are no incoming edges.
    #selfEdge = [];

    #internalConstructor(symbol, ptr, selfEdge) {
        if (symbol !== diplomatRuntime.internalConstructor) {
            console.error("Wrapper is an Opaque type. You cannot call its constructor.");
            return;
        }
        this.#ptr = ptr;
        this.#selfEdge = selfEdge;

        // Are we being borrowed? If not, we can register.
        if (this.#selfEdge.length === 0) {
            Wrapper_box_destroy_registry.register(this, this.#ptr);
        }

        return this;
    }
    get ffiValue() {
        return this.#ptr;
    }
```

```
    static new_() {
        const result = wasm.Wrapper_new();
        try {
            return new Wrapper(diplomatRuntime.internalConstructor, result, []);
        }
        finally {
        }
    }

    returnInner() {
        const diplomatReceive = new diplomatRuntime.DiplomatReceiveBuf(wasm, 8, 4, false);
        // This lifetime edge depends on lifetimes 'a
        let aEdges = [this];
        const result = wasm.Wrapper_return_inner(diplomatReceive.buffer, this.ffiValue);
        try {
            return new diplomatRuntime.DiplomatSliceStr(wasm, diplomatReceive.buffer,  "string8", aEdges).getValue();
        }
        finally {
            diplomatReceive.free();
        }
    }

    ownedBytes() {
        const write = new diplomatRuntime.DiplomatWriteBuf(wasm);
        const result = wasm.Wrapper_owned_bytes(this.ffiValue, write.buffer);
        try {
            return result === 0 ? null : write.readString8();
        }
        finally {
            write.free();
        }
    }

    constructor(symbol, ptr, selfEdge) {
        return this.#internalConstructor(...arguments)
    }
}
```

What should I be aware of?

- the FFI boundary is slow, especially string conversions.
  - If needed in a hot loop, try instead to send a bulk of data in a single flat array type and process it all at once.
- strings can operate differently (utf-16 vs utf-8), so indexes and offsets may have different meanings
  - Especially if you're trying to amortize that FFI cost as outlined above
- FFI is intrinsically unsafe. Diplomat tries to standardise the wrapper code which should help avoid mistakes.

Performance example

Let's look at a simple example, where we split a large text at all whitespace. This is a bad example because you are copying over a lot of data, especially in `benchSpliteratorFull` Which tries to create an ergonomic iterator

Kotlin

```
benchKt                avgt    4     96177,170 ±    5551,892  ns/op
```

Kotlin Diplomat

```
benchIdxs              avgt    4    279589,782 ±   22979,764  ns/op
benchSpliteratorFull   avgt    4  11236381,512 ± 4146516,540  ns/op
```

Rust

```
ws_split                time:   [80.557 µs 80.660 µs 80.794 µs]
```

Serious example – Markov Chains

A Markov chain is a probabilistic process that models transitions between states with the probability only dependent on the current state.

You can use them for simple autocomplete… or just to be silly

> Rust began sponsorship syntainring governal inteers at Rust package.
>
> Rust's expansion, Newsquestem, cit rarelegaokar substant fungin Rust pure as decade.
>
> Rust 0.1 was removing thership systepped focused feder 5,000 compiler the Rust published in Manished a people,: arough the the deceased, and a Reque aking good itself-hostly using initialized increas notation
>
> — My stochastic parrot trained on some of the rust wikipedia page

How do they compare

This is a task that involves “training” the model. It requires traversing the entire text and accumulating statistics into hashmaps. Once the statistics are calculated we can make the markov chain, which generates text for us.

Kotlin

```
benchTrainKt     avgt    4  4815836,004 ± 655981,779  ns/op
benchGenerateKt  avgt    4  3305800,670 ±  74799,994  ns/op
```

Kotlin Diplomat

```
benchTrainRs     avgt    4   969304,276 ± 101890,199  ns/op
benchGenerateRs  avgt    4    79349,769 ±   4450,638  ns/op
```

Rust

```
train_markov            time:   [861.30 µs 864.58 µs 868.76 µs]
markov_generate         time:   [65.937 µs 66.127 µs 66.346 µs]
```