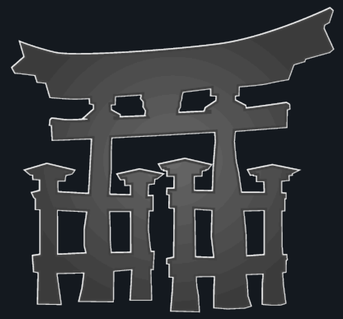
Runtime Objects in Rust

How MiniJinja Works Internally



MiniJinja

Armin @mitsuhiko Ronacher

buckle up …

this will be rough

“ugh unsafe”

MiniJinja

A Rust Implementation of Jinja2

- https://github.com/mitsuhiko/minijinja
- https://docs.rs/minijinja/

- Follow me on Twitter^WX: ***@mitsuhiko***

The Problem

What are we trying to do here?

- “Implement Jinja2 Templates for Rust”
- Sometimes you need to generate text/HTML etc. from templates
- These templates are executed at runtime
- How do you expose Rust objects into the template engine?
- How do you extract Rust objects out of the engine again?

```rust
use minijinja::{Value, render};

fn main() {
    let rv = render!(r#"
    Hello {{ name|title }}!
    {%- for item in seq %}
        - {{ item }}
    {%- endfor %}
    "#, name => "schikaneder", seq => Value::from(vec![1, 2, 3]));
    println!("{}", rv);
}
```

```
Hello Schikaneder!
    - 1
    - 2
    - 3
```

```rust
use minijinja::Value;

fn main() {
    let value = Value::from_object(vec![1i32, 2, 3]);
    println!("{} ({})", value, value.kind());
    let v: &Vec<i32> = value.downcast_object_ref().unwrap();
    println!("{:?}", value);
}
```

```
[1, 2, 3] (sequence)
[1, 2, 3]
```

Dynamic Language

Lifetimes and Garbage Collection

- Value is reference counted
- Internally holds something similar to an `Arc<ActualValue>`
- Clone on Value -> Increment refcount
- How to represent objects?

```
pub trait Object: Debug + Send + Sync {
    fn repr(self: &Arc<Self>) → ObjectRepr {
        ObjectRepr::Map
    }

    fn get_value(self: &Arc<Self>, key: &Value) → Option<Value> {
        None
    }

    fn enumerate(self: &Arc<Self>) → Enumerator {
        Enumerator::NonEnumerable
    }

    ...
}
```

```rust
pub enum ObjectRepr {
    Plain,
    Map,
    Seq,
    Iterable,
}

pub enum Enumerator {
    NonEnumerable,
    Empty,
    Str(&'static [&'static str]),
    Iter(Box<dyn Iterator<Item = Value> + Send + Sync>),
    Seq(usize),
    ...
}
```

```rust
#[derive(Debug)]
struct Point(f32, f32);

impl Object for Point {
    fn repr(self: &Arc<Self>) -> ObjectRepr {
        ObjectRepr::Seq
    }

    fn get_value(self: &Arc<Self>, key: &Value) -> Option<Value> {
        match key.as_usize()? {
            0 => Some(Value::from(self.0)),
            1 => Some(Value::from(self.1)),
            _ => None,
        }
    }

    fn enumerate(self: &Arc<Self>) -> Enumerator {
        Enumerator::Seq(2)
    }
}

let seq_point = Value::from_object(Point(1.0, 2.5));
```

```rust
#[derive(Debug)]
struct Point(f32, f32);

impl Object for Point {
    fn repr(self: &Arc<Self>) -> ObjectRepr {
        ObjectRepr::Map
    }

    fn get_value(self: &Arc<Self>, key: &Value) -> Option<Value> {
        match key.as_str()? {
            "x" => Some(Value::from(self.0)),
            "y" => Some(Value::from(self.1)),
            _ => None,
        }
    }

    fn enumerate(self: &Arc<Self>) -> Enumerator {
        Enumerator::Str(&["x", "y"])
    }
}

let map_point = Value::from_object(Point(1.0, 2.5));
```

```
{{ seq_point }}
  [1.0, 2.5]
{{ seq_point[0] }}
  1.0
{{ seq_point|list }}
  [1.0, 2.5]

{{ map_point }}
  {"x": 1.0, "y": 2.5}
{{ map_point.x }}
  1.0
{{ map_point|list }}
  ["x", "y"]
```

Values and Objects

In short

- Values hold primitives
  - integers
  - strings (Arc<str>)
  - etc.
- Values hold objects
  - Arc<dyn Object>

```rust
use std::sync::Arc;

trait Object {
    fn do_something(self: &Arc<Self>);
}

struct X;

impl Object for X {
    fn do_something(self: &Arc<Self>) { }
}

let obj = Arc::new(X) as Arc<dyn Object>;
```

```
error[E0038]: the trait `Object` cannot be made into an object
   |
4  |     fn do_something(self: &Arc<Self>);
   |                           ---------- help: consider changing method `do_something`'s `self` parameter to be `&self`: `&Self`
...
14 |     let obj = Arc::new(X) as Arc<dyn Object>;
   |                              ^^^^^^^^^^^^^^^ `Object` cannot be made into an object
   |
note: for a trait to be "object safe" it needs to allow building a vtable to allow the call to be resolvable dynamically
   |
3  | trait Object {
   |       ------ this trait cannot be made into an object...
4  |     fn do_something(self: &Arc<Self>);
   |                           ^^^^^^^^^^ ...because method `do_something`'s `self` parameter cannot be dispatched on
```

Error is Revealing

“for a trait to be "object safe" it needs to allow building a vtable”

- Rust cannot build a vtable
- A vtable is a struct of virtual functions
- Can we do it ourselves?
- Plan: replace `Arc<dyn Object>` with a custom `DynObject`

```rust
pub struct DynObject {
    ptr: Arc<Object>,
    vtable: &'static DynObjectVTable,
}

struct VTable {
    repr: fn(&Arc<Object>) → ObjectRepr,
    get_value: fn(&Arc<Object>, key: &Value) → Option<Value>,
    enumerate: fn(&Arc<Object>) → Enumerator,
    type_id: fn() → TypeId,
    type_name: fn() → &'static str,
    drop: fn(Arc<Object>),
}
```

too big, can we just store a pointer?

and how do we accomplish this?

and how do we auto generate this?!

```
type_erase! {
    pub trait Object ⇒ DynObject {
        fn repr(&self) → ObjectRepr;
        fn get_value(&self, key: &Value) → Option<Value>;
        fn enumerate(&self) → Enumerator;
    }
}
```

```rust
pub struct DynObject {
    ptr: *const (),
    vtable: *const (),
}

const _: () = {
    struct VTable {
        repr: fn(*const ()) → ObjectRepr,
        get_value: fn(*const (), key: &Value) → Option<Value>,
        enumerate: fn(*const ()) → Enumerator,
        __type_id: fn() → TypeId,
        __type_name: fn() → &'static str,
        __drop: fn(*const ()),
    }

    fn vt(e: &DynObject) → &VTable {
        unsafe { &*(e.vtable as *const VTable) }
    }

    impl DynObject {
        ...
    }
}
```

rust for "void raw pointer"

trick to declare a hidden type in scope

lots of raw pointers

casts the void pointer to our vtable

brace for impact

```rust
impl DynObject {
    pub fn new<T: Object + 'static>(v: Arc<T>) -> Self {
        let ptr = Arc::into_raw(v) as *const T as *const ();
        let vtable = &VTable {
            repr: |ptr| unsafe {
                Arc::<T>::increment_strong_count(ptr as *const T);
                let arc = Arc::<T>::from_raw(ptr as *const T);
                <T as Object>::repr(&arc)
            },
            __type_id: || TypeId::of::<T>(),
            __type_name: || type_name::<T>(),
            __drop: |ptr| unsafe {
                Arc::from_raw(ptr as *const T);
            },
        };
        Self {
            ptr,
            vtable: vtable as *const VTable as *const (),
        }
    }

    pub fn repr(&self) -> ObjectRepr {
        (vt(self).repr)(self.ptr)
    }
}
```

convert to raw pointer

panic safety!

invoke

reconstruct the Arc

drop

invoke trampoline via vtable

```rust
impl DynObject {
    pub fn downcast_ref<T: 'static>(&self) -> Option<&T> {
        if (vt(self).__type_id)() == TypeId::of::<T>() {
            unsafe {
                return Some(&*(self.ptr as *const T));
            }
        }
        None
    }

    pub fn downcast<T: 'static>(&self) -> Option<Arc<T>> {
        if (vt(self).__type_id)() == TypeId::of::<T>() {
            unsafe {
                Arc::<T>::increment_strong_count(self.ptr as *const T);
                return Some(Arc::<T>::from_raw(self.ptr as *const T));
            }
        }
        None
    }
}
```

only downcast if compatible type

```rust
impl Clone for DynObject {
    fn clone(&self) → Self {
        unsafe {
            std::sync::Arc::increment_strong_count(self.ptr);
        }
        Self {
            ptr: self.ptr,
            vtable: self.vtable,
        }
    }
}

impl Drop for DynObject {
    fn drop(&mut self) {
        (vt(self).__drop)(self.ptr);
    }
}
```

So… how to codegen?

trait Object => DynObject

```
macro_rules! type_erase {
    ($v:vis trait $t_name:ident ⇒ $erased_t_name:ident {
        $(fn $f:ident(&self $(, $p:ident: $t:ty $(,)?)*) $(→ $r:ty)?;)*
    }) ⇒ {
        $v struct $erased_t_name {
            ptr: *const (),
            vtable: *const (),
        }

        const _: () = {
            struct VTable {
                $($f: fn(*const (), $($p: $t),*) $(→ $r)?,)*
                __type_id: fn() → TypeId,
                __type_name: fn() → &'static str,
                __drop: fn(*const ()),
            }

            fn vt(e: &$erased_t_name) → &VTable {
                unsafe { &*(e.vtable as *const VTable) }
            }

            impl $erased_t_name {
                // ...
            }
        };
    };
}
```

match on function declarations

our scope trick again

generate all the function pointer slots

cast to VTable we have in scope

```
impl $erased_t_name {
    $v fn new<T: $t_name + 'static>(v: Arc<T>) -> Self {
        let ptr = Arc::into_raw(v) as *const T as *const ();
        let vtable = &VTable {
            $(
                $f: |ptr, $($p),*| unsafe {
                    Arc::<T>::increment_strong_count(ptr as *const T);
                    let arc = Arc::<T>::from_raw(ptr as *const T);
                    <T as $t_name>::$f(&arc, $($p),*)
                },
            )*
            __type_id: || TypeId::of::<T>(),
            __type_name: || type_name::<T>(),
            __drop: |ptr| unsafe { Arc::from_raw(ptr as *const T); },
        };
        Self { ptr, vtable: vtable as *const VTable as *const () }
    }
}
```

generate all trampolines

```
impl $erased_t_name {
    $(
        $v fn $f(&self, $($p: $t),*) $(-> $r)? {
            (vt(self).$f)(self.ptr, $($p),*)
        }
    )*
    $v fn type_name(&self) -> &'static str {
        (vt(self).__type_name)()
    }
    $v fn downcast_ref<T: 'static>(&self) -> Option<&T> {
        if (vt(self).__type_id)() == TypeId::of::<T>() {
            unsafe { Some(&*(self.ptr as *const T)) }
        } else {
            None
        }
    }
    $v fn downcast<T: 'static>(&self) -> Option<Arc<T>> {
        if (vt(self).__type_id)() == TypeId::of::<T>() {
            unsafe {
                Arc::<T>::increment_strong_count(self.ptr as *const T);
                Some(Arc::<T>::from_raw(self.ptr as *const T));
            }
        } else {
            None
        }
    }
}
```

generate wrapper methods

type name accessor

cast helper

owned cast helper

Documentation

```
#[doc = concat!("Type-erased version of [`", stringify!($t_name), "`]")]
$v struct $erased_t_name {
    ptr: *const (),
    vtable: *const (),
}

$(
    #[doc = concat!(
        "Calls [`", stringify!($t_name), "::", stringify!($f),
        "`] of the underlying boxed value."
    )]
    $v fn $f(&self, $($p: $t),*) $(→ $r)? {
        (vt(self).$f)(self.ptr, $($p),*)
    }
)*
```

generate documentation comments from pieces

Type-erased version of `Object`

Implementations

```
impl DynObject
```

```
pub fn new<T: Object + 'static>(v: Arc<T>) -> Self
```

Returns a new boxed, type-erased `Object`.

```
pub fn repr(&self) -> ObjectRepr
```

Calls `Object::repr` of the underlying boxed value.

```
pub fn get_value(&self, key: &Value) -> Option<Value>
```

Calls `Object::get_value` of the underlying boxed value.

```
pub fn enumerate(&self) -> Enumerator
```

Calls `Object::enumerate` of the underlying boxed value.

Putting it Together

```rust
#[derive(Clone)]
pub(crate) enum ValueRepr {
    Undefined,
    Bool(bool),
    U64(u64),
    I64(i64),
    F64(f64),
    None,
    String(Arc<str>, StringType),
    Bytes(Arc<Vec<u8>>),
    Object(DynObject),
}

#[derive(Clone)]
pub struct Value(pub(crate) ValueRepr);
```

```rust
impl Value {
    pub fn from_object<T: Object + Send + Sync + 'static>(value: T) -> Value {
        Value::from(ValueRepr::Object(DynObject::new(Arc::new(value))))
    }

    pub fn downcast_object_ref<T: 'static>(&self) -> Option<&T> {
        match self.0 {
            ValueRepr::Object(ref o) => o.downcast_ref(),
            _ => None,
        }
    }

    pub fn downcast_object<T: 'static>(&self) -> Option<Arc<T>> {
        match self.0 {
            ValueRepr::Object(ref o) => o.downcast(),
            _ => None,
        }
    }
}
```

fin.