Specialized Generic Methods in Rust

Adrian Kosmaczewski

2021-05-07

I have not been so interested in a language in years as I am right now with Rust. Readers of De Programmatica Ipsum have probably sensed this last Monday when we published "The Great Rewriting in Rust", which has been a hit article this week.

To say that I'm fascinated by Rust is an understatement.

So, as an exercise, I wanted to replicate in Rust one of the subsystems I used for my master thesis project back in 2008. That one was a C++03 project; back then C++11 was still in the works, so I could not use many new features that are nowadays common in the standard, like std::tuple and others.

That project had a homemade ORM, basically implementing all objects as `std::map` of properties using the `Poco::Any` object. Those "properties" were just generic key/value pairs, and I got the inspiration from a C++ book I had read for work, "Financial Instrument Pricing Using C++" by Daniel Duffy (book which has since been updated to C++11). The final goal of that ORM was to save a complete object graph into a SQLite file.

Here is the original code for the implementation of the property pattern I used in my project.

So, how would we do this in Rust? I started with the most naive implementation, and to my surprise (given my rookie level of Rust knowledge) it just worked:

```
struct Property<T> {
    name: String,
    value: T,
}
```

I added a convenience function attached to that struct, just to have a nicer way to create new instances in my code:

```
impl<T> Property<T> {
    fn create(name: &str, value: T) -> Property<T> {
        Property {
            name: String::from(name),
            value: value,
```

```
        }
    }
}
```

Now I needed to create a bag structure holding lots of these things:

```
struct PropertyMap<T> {
    pub values: HashMap<String, Property<T>>,
}
```

And of course I added a set of convenience functions to help me use this type:

```
impl<T> PropertyMap<T> {
    fn create(size: usize, properties: Vec<Property<T>>) -> PropertyMap<T> {
        let mut values: HashMap<String, Property<T>> = HashMap::with_capacity(size as usize)
        for property in properties {
            let name = property.name.clone();
            values.insert(name, property);
        }
        PropertyMap { values: values }
    }

    fn get(&self, key: &str) -> Result<&Property<T>, String> {
        if self.values.contains_key(key) {
            return Ok(&self.values[key]);
        }
        return Err(format!("Value '{}' not found", key));
    }
}
```

Beautiful implementation of `get()` with the very handy `Result` type. Of course, now I need a way to pretty-print these things in the console:

```
impl<T> fmt::Display for Property<T>
where
    T: fmt::Display,
{
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{} = {}", self.name, self.value)
    }
}

impl<T> fmt::Display for PropertyMap<T>
where
    T: fmt::Display,
{
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        for (key, value) in &(self.values) {
            writeln!(f, "{} => {}", key, value)?;
```

```
        }
        return Ok(());
    }
}
```

So far, so good. Now I can start playing with those things:

```
fn main() {
    let a = Property::create("age", 22);

    // Create map and print it
    let map: PropertyMap<i32> =
        PropertyMap::create(30, vec![a.clone()]);
    println!("Full map =>");
    println!("{}", map);

    // Try to find item in map
    for key in ["tata", "toti", "age"].iter() {
        let found = map.get(key);
        match found {
            Ok(value) => println!("FOUND '{}' => {}", key, value),
            Err(message) => println!("{}", message),
        }
    }

    // Create properties of other types
    let b = Property::create("name", "Roger");
    let c = Property::create("valid", true);
    let d = Property::create("cost", 6666.25);
}
```

Looking good! Now I also implemented equality, so that I can compare properties with one another:

```
impl<T> cmp::Eq for Property<T> where T: cmp::PartialEq {}

impl<T> cmp::PartialEq for Property<T>
where
    T: cmp::PartialEq,
{
    fn eq(&self, other: &Property<T>) -> bool {
        return self.name == other.name && self.value == other.value;
    }
}

fn main() {
    let a = Property::create("age", 22);
    let a1 = a.clone();
```

```
    let a2 = Property::create("age", 256);
    let a3 = Property::create("whatever", 22);

    println!("{}", a == a1); // true
    println!("{}", a == a2); // false
    println!("{}", a == a3); // false
}
```

Of course, it does not make sense to compare properties that are not of the same type; the compiler would complain, anyway.

Now I wanted to push things a bit further; when creating SQL statements for inserting or updating, we need to surround strings with single quotes; my first reaction was to create a generic `format_for_database()` function, and then provide a specialized version for `Property<String>` and `Property<&'static str>`, but I quickly found out that this is currently not possible in Rust.

So I ended up doing the following instead:

```
type IntProperty = Property<i32>;
type FloatProperty = Property<f64>;
type BoolProperty = Property<bool>;
type StringProperty = Property<&'static str>;

impl IntProperty {
    fn format_for_database(&self) -> String {
        format!("{} = {}", self.name, self.value)
    }
}

impl BoolProperty {
    fn format_for_database(&self) -> String {
        let val = if self.value { "TRUE" } else { "FALSE" };
        format!("{} = {}", self.name, val)
    }
}

impl FloatProperty {
    fn format_for_database(&self) -> String {
        format!("{} = {}", self.name, self.value)
    }
}

impl StringProperty {
    fn format_for_database(&self) -> String {
        format!("{} = '{}'", self.name, self.value)
    }
}
```

Now we can do very polymorphic calls like this:

```
fn main() {
    let a = Property::create("age", 22);
    println!("Property a => {}", a.format_for_database());

    let b = Property::create("name", "Roger");
    println!("{}", b.format_for_database());

    let c = Property::create("valid", true);
    println!("{}", c.format_for_database());

    let d = Property::create("cost", 6666.25);
    println!("{}", d.format_for_database());
}
```

Verbose, but gets things done, waiting until the language supports such feature. Ideally I would have implemented a generic `format_for_database()` function in the generic `impl<T>` block, but if I do that, I get error `E0592`, complaining about a duplicate definition.

You can find the final code in my GitLab account. The next step for this project will be to use Rust's own Any trait and translate the original C++ code into a working Rust version.