Conversion traits in Rust

Jakub Pastuszek - March 2017

Rust

- Rust is a strongly typed language
- Type conversions are very important
- Type inference plays important role
- Type conversion methods are defined as traits in *corelib* and *stdlib* under `core::convert` and `std::convert` modules

Conversion traits

- By-value conversion
  - `Into`
  - `From`
  - `TryInto`/`TryFrom` (unstable/RFC 1542)
- Reference conversion
  - `AsRef`/`AsMut` - convert something to a reference in a generic way

Borrowing and dereferencing traits

- Dereferencing values
  - `Deref` - specify the functionality of dereferencing operations, like `*v`
- Borrowing
  - `Borrow`/`BorrowMut` - abstract over different kinds of borrowing data; used in data structure that treats owned and borrowed values in equivalent ways; e.g.: `HashMap`
  - `ToOwned` - go from borrowed to owned
  - `Cow` - clone-on-write

Into

- Conversion by value - transfers ownership into new value
- By the naming convention the operation may not be cheap
- Conversion cannot fail
- Used as method: `a.into()`

Into example

```
1  #[derive(Debug)]
2  struct Miles(f32);
3  #[derive(Debug)]
4  struct Meters(f32);
```

Into example

```
6  impl Into<Miles> for Meters {
7      fn into(self) -> Miles {
8          Miles(self.0 / 1609.344)
9      }
10 }
11
12 impl Into<Meters> for Miles {
13     fn into(self) -> Meters {
14         Meters(self.0 * 1609.344)
15     }
16 }
```

Into example

```
18 fn main() {
19     let a = Miles(2.0);
20     let b: Meters = a.into();
21
22     //println!("{:?}", a); // use of moved value: `a`
23     println!("{:?}", b);
24 }
```

```
Meters(3218.688)
```

Into and generic argument bound

```
18 fn distance<A, B>(a: A, b: B) -> Miles where A: Into<Miles>, B: Into<Miles> {
19     Miles(b.into().0 - a.into().0)
20 }
```

```
22 fn main() {
23     println!("{:?}", distance(Meters(20.0), Miles(3.0)));
24     println!("{:?}", distance(Miles(2.0), Miles(10.0)));
25     println!("{:?}", distance(Meters(200.0), Meters(234.0)));
26 }
```

```
Miles(2.9875727)
Miles(8)
Miles(0.021126628)
```

From

- Mirror of `Into`
- Used as trait function: `From::from(a)`
- Used as constructor function: `Miles::from(a)`
- Implements `Into: impl<T, U> Into<U> for T where U: From<T>`

From example

```
6  impl From<Miles> for Meters {
7      fn from(miles: Miles) -> Meters {
8          Meters(miles.0 * 1609.344)
9      }
10 }
11
12 impl From<Meters> for Miles {
13     fn from(meters: Meters) -> Miles {
14         Miles(meters.0 / 1609.344)
15     }
16 }
```

From example

```
18 fn main() {
19     let a = Miles(2.0);
20     let b: Meters = From::from(a);
21
22     //println!("{:?}", a); // use of moved value: `a`
23     println!("{:?}", b);
24 }
```

```
Meters(3218.688)
```

From implements Into

```
18 fn main() {
19     let a = Miles(2.0);
20     let b: Meters = a.into();
21
22     //println!("{:?}", a); // use of moved value: `a`
23     println!("{:?}", b);
24 }
```

```
Meters(3218.688)
```

From as a constructor

```
22 fn main() {
23     println!("{:?}", Meters::from(distance(Meters(20.0), Miles(3.0))));
24     println!("{:?}", Meters::from(distance(Miles(2.0), Miles(10.0))));
25     println!("{:?}", Meters::from(distance(Meters(200.0), Meters(234.0))));
26 }
```

```
Meters(4808.032)
Meters(12874.752)
Meters(34.00001)
```

Error handling with From

- Used internally by `try!` and `?` to convert returned error to error type in returned `Result`
- Used to force trait implementers user to provide conversion for custom error types

```rust
#[derive(Debug)]
pub struct InternalError(i32);

#[derive(Debug)]
pub enum ModuleError {
    Internal(InternalError),
    Other,
}

impl From<InternalError> for ModuleError {
    fn from(error: InternalError) -> ModuleError {
        ModuleError::Internal(error)
    }
}

fn calculate(a: i32, b: i32) -> Result<i32, InternalError> {
    if a >= b {
        Ok(a - b)
    } else {
        Err(InternalError(a + b))
    }
}

pub fn do_calculation(a: i32, b: i32) -> Result<i32, ModuleError> {
    Ok(a + calculate(a, b)?)
}

fn main() {
    println!("{:?}", calculate(2, 3));
    println!("{:?}", do_calculation(2, 3));
}
```

```
Err(InternalError(5))
Err(Internal(InternalError(5)))
```

```rust
trait Calculate {
    type Error: From<i32>;

    fn calculate(&self, b: i32) -> Result<i32, Self::Error> {
        let a = self.a();

        if a >= b {
            Ok(a - b)
        } else {
            Err(From::from(a + b))
        }
    }

    fn a(&self) -> i32;
}
```

```rust
#[derive(Debug)]
pub struct MyError(i32);

impl From<i32> for MyError {
    fn from(number: i32) -> MyError {
        MyError(number)
    }
}

struct Foo(i32);

impl Calculate for Foo {
    type Error = MyError;

    fn a(&self) -> i32 {
        self.0
    }
}

fn main() {
    let foo = Foo(10);

    println!("{:?}", foo.calculate(3));
    println!("{:?}", foo.calculate(13));
}
```

```
Ok(7)
Err(MyError(23))
```

From bound in *futures* crate

```
impl<T> Sink for Sender<T>
```

```
type SinkItem = T
```

The type of value that the sink accepts.

```
type SinkError = SendError<T>
```

The type of value produced by the sink when an error occurs.

```
[-]fn start_send(&mut self, msg: T) -> StartSend<T, SendError<T>>
```

Begin the process of sending a value to the sink. Read more

Trait futures::stream::Stream

```
[-]fn forward<S>(self, sink: S) -> Forward<Self, S>
    where S: Sink<SinkItem=Self::Item>,
          Self::Error: From<S::SinkError>,
          Self: Sized
```

Type inference

```
18 fn main() {
19     let mut route = Vec::new();
20     route.push(Meters(2000.0).into());
21     route.push(Miles(2.0));
22     route.push(From::from(Meters(3000.0)));
23
24     println!("{:?}", route);
25 }
```

Type inference

```
18 fn main() {
19     let mut route = Vec::new();
20     route.push(Meters(2000.0).into());
21     route.push(Miles(2.0));
22     route.push(From::from(Meters(3000.0)));
23
24     println!("{:?}", route);
25 }
```

```
[Miles(1.2427424), Miles(2), Miles(1.8641136)]
```

Naming convention

Methods prefixed with:

- `as_` - takes self by reference and return a reference; should be cheap; e.g.: `as_str()`, `as_slice()`
- `to_` - takes self by reference and returns new value; may be expensive; e.g.: `to_owned()`, `to_bytes()`
- `into_` - takes self by value (move/consume) and return new value; it may or may not be cheap; e.g: `into()`, `into_iter()`, `into_vec()`
- `try_` - methods that may fail/return `Result`; e.g.: `try_unwrap()`, `try_into()`

! type (RFC 1216)

! represents empty type that has no value:

- they never exist at runtime because there is no way to create one
- they have no logical machine-level representation
- code that handles them never executes
- represent the return type of functions that never return
- can be converted to any other type
- e.g.: `baz() -> !`, `foo() -> Result<!, io::Error>`, `bar() -> Result<u32, !>`

! and TryFrom/TryInto

- `TryFrom` and `TryInto` with `Error` associated type of `!` would be infallible like `From` and `Into`
- There is plan is to implement `TryFrom/TryInto<Error=!>` for every type that implements `From/Into`
- Currently it is supported only as return type:

```
1 fn foo() -> ! {
2     panic!("boom!")
3 }
```

Thank you!