Rust-Postgres

An idiomatic, native Postgres driver

Steven Fackler - sfackler

July 31, 2014

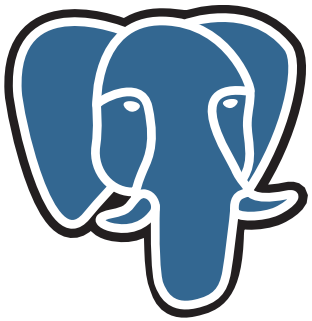
Outline

Background
- PostgreSQL
- Rust-Postgres

Overview
- Usage
- Design
- Macros

Future

What's PostgreSQL?



PostgreSQL
the world's most advanced open source database

"PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness."

Connecting

Connect with a standard psql-style URI:

```
use postgres::{PostgresConnection, NoSsl};
let url = "postgresql://sfackler@localhost:15410/mydb";
let conn = try!(
        PostgresConnection::connect(url, &NoSsl));
```

Connecting

Unix sockets are supported as well:

```
use postgres::{PostgresConnection, NoSsl};
let url = "postgresql://sfackler@%2Frun%2Fpostgres/mydb";
let conn = try!(
        PostgresConnection::connect(url, &NoSsl));
```

Connecting

Alternatively, pass a 'PostgresConnectParams' struct:

```
use postgres::{PostgresConnection, NoSsl,
               PostgresConnectParams, TargetUnix};
let params = PostgresConnectParams {
    target: TargetUnix(Path::new("/run/postgres")),
    port: Some(1234),
    .....
};
let conn = try!(
        PostgresConnection::connect(params, &NoSsl));
```

Statement Preparation

Queries must first be *prepared* before they can be executed. They may be parameterized. Parameters are denoted by `$n`, and are 1-indexed.

```
let query = "SELECT name, height
             FROM people
             WHERE age < $1";
let stmt = try!(conn.prepare(query));
```

Execution

The `execute` method takes a slice of values to bind to the query parameters and returns the number of rows modified.

```
let query = "UPDATE users SET name = $1
             WHERE age = $2";
let stmt = try!(conn.prepare(query));
let rows_updated = try!(stmt.execute(
        [&"Steven", &Some(24i32)]));
```

Querying

query is similar to execute but it returns an iterator over the rows returned by a query. Columns may be accessed by index or name.

```
let query = "SELECT name, age FROM users
             WHERE age < $1";
let stmt = try!(conn.prepare(query));
for row in try!(stmt.query([&18i32])) {
    let name: String = row.get(0u);
    let age: Option<i32> = row.get("age");
    println!("{} is {}", name, age);
}
```

Parameterization

Use it. Seriously.

Parameterization

```
fn update_grade(conn: &PostgresConnection,
                name: &str, grade: f32)
                -> PostgresResult<()> {
    let query = format!("UPDATE students SET grade = {}
                         WHERE name = '{}'",
                        grade, name);
    try!(stmt.batch_execute(query.as_slice()));
    Ok(())
}
```

Parameterization

```
let name = "Robert'); DROP TABLE Students;--";
let grade = 100f32;
update_grade(&conn, name, grade);
```

Parameterization

```
fn update_grade(conn: &PostgresConnection,
                name: &str, grade: f32)
                -> PostgresResult<Student> {
    let query = "UPDATE students SET grade = $1
                 WHERE name = $1";
    let stmt = try!(conn.prepare(query));
    try!(stmt.update([&grade, &name]));
    Ok(())
}
```

Transactions

Transactions are managed by the `PostgresTransaction` object:

```
let trans = try!(conn.transaction());
let stmt = try!(trans.prepare(...));
....

if the_coast_is_clear {
    trans.set_commit();
}

try!(trans.finish()); // COMMIT / ROLLBACK here
```

RAII and Lifetimes are Awesome

Statically guarantee that resources are cleaned up in the right order without needing to resort to reference counting or garbage collection.

Strict Typing

Like Rust itself, Rust-Postgres won't perform implicit type conversions. A Postgres type is only convertable to an "equivalent" Rust type and vice versa.

```
CREATE TABLE foo (id INT PRIMARY KEY, bar BIGINT)
```

```
let stmt = try!conn.prepare("SELECT id FROM foo"));
for row in try!(stmt.query([])) {
    row.get::<i32>(0u); // ok
    row.get::<Option<i32>>(0u); // ok
    row.get::<uint>(0u); // not ok
}
```

```
let stmt = try!(conn.prepare("UPDATE foo SET bar = $1"));
try!(stmt.execute([&1i64])); // ok
try!(stmt.execute([&Some(2i64)])); // ok
try!(stmt.execute([&"1"])); // not ok
```

Extensibility

Postgres allows for extensions which define new types and operations.

- PostGIS
- HStore

Rust-Postgres provides an extendible API for conversions to and from Postgres values.

Type Conversion

All conversions are done through two traits

```
pub trait ToSql {
    fn to_sql(&self, ty: &PostgresType)
              -> PostgresResult<(Format,
                                 Option<Vec<u8>>)>;
}

pub trait FromSql {
    fn from_sql(ty: &PostgresType,
                raw: &Option<Vec<u8>>)
                -> PostgresResult<Self>;
}
```

Compile Time Checks

Compile time checks are awesome, but SQL opens up a series of issues that won't be discovered until runtime.

- Invalid syntax

```
SELECT * FORM foo
```

- Parameter count mismatch

```
try!(conn.execute("UPDATE foo SET a = $1
                              WHERE b = $2",
                        [&1i32]));
```

- Schema mismatch

```
SELECT nmae FROM users
```

Syntax Extensions to the Rescue

Link against PostgreSQL's query parser and have it do the heavy lifting!

```
#[phase(plugin)]
extern crate postgres_macros;


let query = sql!("SELECT * FORM foo");

test.rs:8:18: 8:35 error: Invalid syntax at
    position 10: syntax error at or near "FORM"
test.rs:8     let query = sql!("SELECT * FORM foo");
                                ^~~~~~~~~~~~~~~~~
```

Syntax Extensions to the Rescue

```
#[phase(plugin)]
extern crate postgres_macros;

try!(execute!(conn,
              "UPDATE foo SET a = $1
                WHERE b = $2",
              &1i32));
```

```
test.rs:7:1: 10:23 error: Expected 2 query parameters
        but got 1
test.rs:7 try!(execute!(conn,
test.rs:8                   "UPDATE foo SET a = $1
test.rs:9                     WHERE b = $2",
test.rs:10                   &1i32));
```

What's Missing

Rust-Postgres defines the basics, but much of the infrastructure on top is still missing.

- Connection Pool - There is a pool in Rust-Postgres but it's no where near sufficient.
- ORM - Syntax extensions could allow for an interesting ORM system.

That's It!

Questions?