



下载APP



44 | 数据处理：应用程序和数据如何打交道？

2021-12-10 陈天

《陈天 · Rust 编程第一课》

课程介绍 >



讲述：陈天

时长 11:41 大小 10.71M



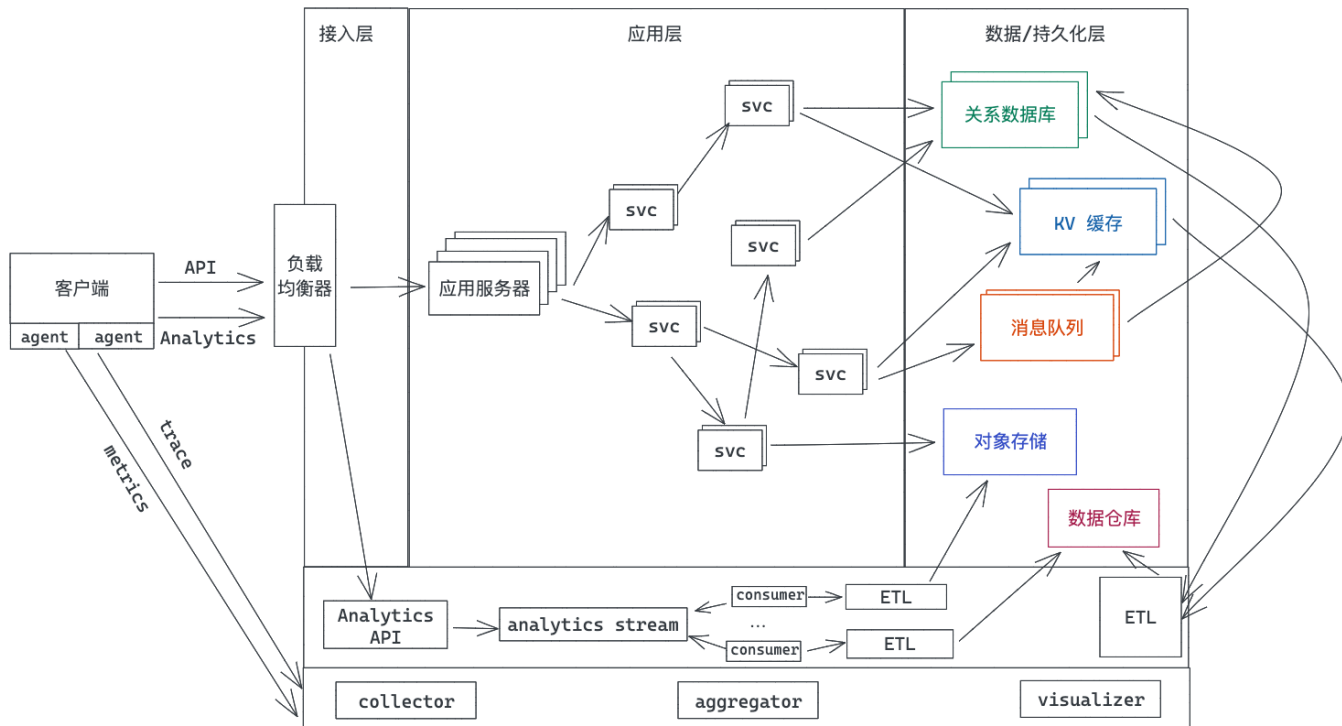
你好，我是陈天。

我们开发者无论是从事服务端的开发，还是客户端的开发，和数据打交道是必不可少的。

对于客户端来说，从服务端读取到的数据，往往需要做缓存（内存缓存或者 SQLite 缓存），甚至需要本地存储（文件或者 SQLite）。

对于服务器来说，跟数据打交道的场景就更加丰富了。除了数据库和缓存外，还有大量数据的索引（比如搜索引擎）、实时的消息队列对数据做流式处理，或者非实时的批处理对数据仓库（data warehouse）中的海量数据进行 ETL（Extract、Transform and Load）。





今天我们就来讲讲如何用 Rust 做数据处理，主要讲两部分，如何用 Rust 访问关系数据库，以及如何用 Rust 对半结构化数据进行分析 and 处理。希望通过学习这一讲的内容，尤其是后半部分的内容，能帮你打开眼界，对数据处理有更加深刻的认识。

访问关系数据库

作为互联网应用的最主要的数据存储和访问工具，关系数据库，是几乎每门编程语言都有良好支持的数据库类型。

在 Rust 下，有几乎所有主流关系数据库的驱动，比如 [rust-postgres](#)、[rust-mysql-simple](#) 等，不过一般我们不太会直接使用数据库的驱动来访问数据库，因为那样会让应用过于耦合于某个数据库，所以我们会使用 ORM。

Rust 下有 [diesel](#) 这个非常成熟的 ORM，还有 [sea-orm](#) 这样的后起之秀。diesel 不支持异步，而 sea-orm 支持异步，所以，有理由相信，随着 sea-orm 的不断成熟，会有越来越多的应用在 sea-orm 上构建。

如果你觉得 ORM 太过笨重，繁文缛节太多，但又不想直接使用某个数据库的驱动来访问数据库，那么你还可以用 [sqlx](#)。sqlx 提供了对多种数据库（Postgres、MySQL、SQLite、MSSQL）的异步访问支持，并且不使用 DSL 就可以对 SQL query 做编译时检

查，非常轻便；它可以从数据库中直接查询出来一行数据，也可以通过派生宏自动把行数据转换成对应的结构。

今天，我们就尝试使用 `sqlx` 处理用户注册和登录这两个非常常见的功能。

sqlx

构建下面的表结构来处理用户登录信息：

[复制代码](#)

```
1 CREATE TABLE IF NOT EXISTS users
2 (
3     id                INTEGER PRIMARY KEY NOT NULL,
4     email             VARCHAR UNIQUE     NOT NULL,
5     hashed_password   VARCHAR           NOT NULL
6 );
```

特别说明一下，在数据库中存储用户信息需要非常谨慎，尤其是涉及敏感的数据，比如密码，需要使用特定的哈希算法存储。OWASP 对密码的存储有如下 [安全建议](#)：

1. 如果 Argon2id 可用，那么使用 Argon2id（需要目标机器至少有 15MB 内存）。
2. 如果 Argon2id 不可用，那么使用 bcrypt（算法至少迭代 10 次）。
3. 之后再考虑 scrypt / PBKDF2。

Argon2id 是 Argon2d 和 Argon2i 的组合，Argon2d 提供了强大的抗 GPU 破解能力，但在特定情况下会容易遭受 [旁路攻击](#)（side-channel attacks），而 Argon2i 则可以防止旁路攻击，但抗 GPU 破解稍弱。所以只要是编程语言支持 Argon2id，那么它就是首选的密码哈希工具。

Rust 下有完善的 [password-hashes](#) 工具，我们可以使用其中的 [argon2](#) crate，用它生成的一个完整的，包含所有参数的密码哈希长这个样子：

[复制代码](#)

```
1 $argon2id$v=19$m=4096,t=3,p=1$l7IEIWV7puJYJAZHyyut8A$0PxL090Dxp/xDQEnlG1NWd0sT
```

这个字符串里包含了 argon2id 的版本 (19)、使用的内存大小 (4096k)、迭代次数 (3 次)、并行程度 (1 个线程)，以及 base64 编码的 salt 和 hash。

所以，当新用户注册时，我们使用 argon2 把传入的密码哈希一下，存储到数据库中；当用户使用 email/password 登录时，我们通过 email 找到用户，然后再通过 argon2 验证密码。数据库的访问使用 sqlx，为了简单起见，避免安装额外的数据库，就使用 SQLite 来存储数据（如果你本地有 MySQL 或者 PostgreSQL，可以自行替换相应的语句）。

有了这个思路，我们创建一个新的项目，添加相关的依赖：

[复制代码](#)

```
1 [dev-dependencies]
2 anyhow = "1"
3 argon2 = "0.3"
4 lazy_static = "1"
5 rand_core = { version = "0.6", features = ["std"] }
6 sqlx = { version = "0.5", features = ["runtime-tokio-rustls", "sqlite"] }
7 tokio = { version = "1", features = ["full"] }
```

然后创建 examples/user.rs，添入代码，你可以对照详细的注释来理解：

[复制代码](#)

```
1 use anyhow::{anyhow, Result};
2 use argon2::{
3     password_hash::{rand_core::OsRng, PasswordHash, PasswordHasher, SaltString},
4     Argon2, PasswordVerifier,
5 };
6 use lazy_static::lazy_static;
7 use sqlx::{sqlite::SqlitePoolOptions, SqlitePool};
8 use std::env;
9
10 /// Argon2 hash 使用的密码
11 const ARGON_SECRET: &[u8] = b"deadbeef";
12 lazy_static! {
13     /// Argon2
14     static ref ARGON2: Argon2<'static> = Argon2::new_with_secret(
15         ARGON_SECRET,
16         argon2::Algorithm::default(),
17         argon2::Version::default(),
18         argon2::Params::default()
19     )
20     .unwrap();
21 }
```

```

22
23 /// user 表对应的数据结构, 处理 login/register
24 pub struct UserDb {
25     pool: SqlitePool,
26 }
27
28 /// 使用 FromRow 派生宏把从数据库中读取出来的数据转换成 User 结构
29 #[allow(dead_code)]
30 #[derive(Debug, sqlx::FromRow)]
31 pub struct User {
32     id: i64,
33     email: String,
34     hashed_password: String,
35 }
36
37 impl UserDb {
38     pub fn new(pool: SqlitePool) -> Self {
39         Self { pool }
40     }
41
42     /// 用户注册: 在 users 表中存储 argon2 哈希过的密码
43     pub async fn register(&self, email: &str, password: &str) -> Result<i64> {
44         let hashed_password = generate_password_hash(password)?;
45         let id = sqlx::query("INSERT INTO users(email, hashed_password) VALUES
46             .bind(email)
47             .bind(hashed_password)
48             .execute(&self.pool)
49             .await?
50             .last_insert_rowid();
51
52         Ok(id)
53     }
54
55     /// 用户登录: 从 users 表中获取用户信息, 并用验证用户密码
56     pub async fn login(&self, email: &str, password: &str) -> Result<String> {
57         let user: User = sqlx::query_as("SELECT * from users WHERE email = ?")
58             .bind(email)
59             .fetch_one(&self.pool)
60             .await?;
61         println!("find user: {:?}", user);
62         if let Err(_) = verify_password(password, &user.hashed_password) {
63             return Err( anyhow!("failed to login"));
64         }
65
66         // 生成 JWT token (此处省略 JWT token 生成的细节)
67         Ok("awesome token".into())
68     }
69 }
70
71 /// 重新创建 users 表
72 async fn recreate_table(pool: &SqlitePool) -> Result<()> {
73     sqlx::query("DROP TABLE users").execute(pool).await?;

```

```
74     sqlx::query(
75         r#"CREATE TABLE IF NOT EXISTS users(
76             id            INTEGER PRIMARY KEY NOT NULL,
77             email          VARCHAR UNIQUE      NOT NULL,
78             hashed_password VARCHAR            NOT NULL)"#,
79     )
80     .execute(pool)
81     .await?;
82     Ok(())
83 }
84
85 /// 创建安全的密码哈希
86 fn generate_password_hash(password: &str) -> Result<String> {
87     let salt = SaltString::generate(&mut OsRng);
88     Ok(ARGON2
89         .hash_password(password.as_bytes(), &salt)
90         .map_err(|_| anyhow!("failed to hash password"))?
91         .to_string())
92 }
93
94 /// 使用 argon2 验证用户密码和密码哈希
95 fn verify_password(password: &str, password_hash: &str) -> Result<()> {
96     let parsed_hash =
97         PasswordHash::new(password_hash).map_err(|_| anyhow!("failed to parse
98 ARGON2
99         .verify_password(password.as_bytes(), &parsed_hash)
100         .map_err(|_| anyhow!("failed to verify password"))?;
101     Ok(())
102 }
103
104 #[tokio::main]
105 async fn main() -> Result<()> {
106     let url = env::var("DATABASE_URL").unwrap_or("sqlite:///./data/example.db".
107 // 创建连接池
108     let pool = SqlitePoolOptions::new()
109         .max_connections(5)
110         .connect(&url)
111         .await?;
112
113     // 每次运行都重新创建 users 表
114     recreate_table(&pool).await?;
115
116     let user_db = UserDb::new(pool.clone());
117     let email = "tyr@awesome.com";
118     let password = "hunter42";
119
120     // 新用户注册
121     let id = user_db.register(email, password).await?;
122     println!("registered id: {}", id);
123
124     // 用户成功登录
125     let token = user_db.login(email, password).await?;
```




```

126     println!("Login succeeded: {}", token);
127
128     // 登录失败
129     let result = user_db.login(email, "badpass").await;
130     println!("Login should fail with bad password: {:?}", result);
131
132     Ok(())
133 }

```

在这段代码里，我们把 argon2 的能力稍微包装了一下，提供了 generate_password_hash 和 verify_password 两个方法给注册和登录使用。对于数据库的访问，我们提供了一个连接池 SqlitePool，便于无锁访问。

你可能注意到了这句写法：

 复制代码


```

1 let user: User = sqlx::query_as("SELECT * from users WHERE email = ?")
2     .bind(email)
3     .fetch_one(&self.pool)
4     .await?;

```

是不是很惊讶，一般来说，这是 ORM 才有的功能啊。没错，它再次体现了 Rust trait 的强大：我们并不需要 ORM 就可以把数据库中的数据跟某个 Model 结合起来，只需要在查询时，提供想要转换成的数据结构 T: FromRow 即可。

看 query_as 函数和 FromRow trait 的定义（[🔗 代码](#)）：

 复制代码

```

1 pub fn query_as<'q, DB, O>(sql: &'q str) -> QueryAs<'q, DB, O, <DB as HasArgum
2 where
3     DB: Database,
4     O: for<'r> FromRow<'r, DB::Row>,
5 {
6     QueryAs {
7         inner: query(sql),
8         output: PhantomData,
9     }
10 }
11
12 pub trait FromRow<'r, R: Row>: Sized {
13     fn from_row(row: &'r R) -> Result<Self, Error>;
14 }

```

要让一个数据结构支持 `FromRow` , 很简单 , 使用 `sqlx::FromRow` 派生宏即可 :

[复制代码](#)

```
1 #[derive(Debug, sqlx::FromRow)]
2 pub struct User {
3     id: i64,
4     email: String,
5     hashed_password: String,
6 }
```

希望这个例子可以让你体会到 Rust 处理数据库的强大和简约。我们用 Rust 写出了 Node.js / Python 都不曾拥有的直观感受。另外, `sqlx` 是一个非常漂亮的 crate , 有空的话建议你也看看它的源代码, 开头介绍的 `sea-orm` , 底层也是使用了 `sqlx`。

用 Rust 对半结构化数据进行分析

在生产环境中, 我们会累积大量的半结构化数据, 比如各种各样的日志、监控数据和分析数据。

以日志为例, 虽然通常会将其灌入日志分析工具, 通过可视化界面进行分析和问题追踪, 但偶尔我们也需要自己写点小工具进行处理, 一般, 会用 Python 来处理这样的任务, 因为 Python 有 `pandas` 这样用起来非常舒服的工具。然而, `pandas` 太吃内存, 运算效率也不算高。有没有更好的选择呢?

在第 6 讲我们介绍过 [polars](#) , 也用 `polars` 和 [sqlparser](#) 写了一个处理 csv 的工具, 其实 `polars` 底层使用了 [Apache arrow](#)。如果你经常进行大数据处理, 那么你对列式存储 ([columnar datastore](#)) 和 [Data Frame](#) 应该比较熟悉, `arrow` 就是一个在内存中进行存储和运算的列式存储, 它是构建下一代数据分析平台的基础软件。

由于 Rust 在业界的地位越来越重要, `Apache arrow` 也构建了完全用 [Rust 实现的版本](#) , 并在此基础上构建了高效的 in-memory 查询引擎 [datafusion](#) , 以及在某些场景下可以取代 Spark 的分布式查询引擎 [ballista](#)。

Apache arrow 和 datafusion 目前已经有很多重磅级的应用，其中最令人兴奋的是 [InfluxDB IOx](#)，它是 [下一代的 InfluxDB 的核心引擎](#)。

来一起感受一下 datafusion 如何使用：

[复制代码](#)

```
1 use datafusion::prelude::*;
2 use datafusion::arrow::util::pretty::print_batches;
3 use datafusion::arrow::record_batch::RecordBatch;
4
5 #[tokio::main]
6 async fn main() -> datafusion::error::Result<()> {
7     // register the table
8     let mut ctx = ExecutionContext::new();
9     ctx.register_csv("example", "tests/example.csv", CsvReadOptions::new()).await
10
11     // create a plan to run a SQL query
12     let df = ctx.sql("SELECT a, MIN(b) FROM example GROUP BY a LIMIT 100").await
13
14     // execute and print results
15     df.show().await?;
16     Ok(())
17 }
```

在这段代码中，我们通过 `CsvReadOptions` 推断 CSV 的 schema，然后将其注册为一个逻辑上的 `example` 表，之后就可以通过 SQL 进行查询了，是不是非常强大？

下面我们就使用 datafusion，来构建一个 Nginx 日志的命令行分析工具。

datafusion

在这门课程的 [GitHub repo](#) 里，我放了个从网上找到的样本日志，改名为 `nginx_logs.csv`（注意后缀需要是 `csv`），其格式如下：

[复制代码](#)

```
1 93.180.71.3 - - "17/May/2015:08:05:32 +0000" GET "/downloads/product_1" "HTTP/
2 93.180.71.3 - - "17/May/2015:08:05:23 +0000" GET "/downloads/product_1" "HTTP/
3 80.91.33.133 - - "17/May/2015:08:05:24 +0000" GET "/downloads/product_1" "HTTP
```

这个日志共有十个域，除了几个 “-” ，无法猜测到是什么内容外，其它的域都很好猜测。

由于 `nginx_logs` 的格式是在 Nginx 配置中构建的，所以，日志文件，并不像 CSV 文件那样有一行 header，没有 header，就无法让 `datafusion` 直接帮我们推断出 schema，也就是说**我们需要显式地告诉 `datafusion` 日志文件的 schema 长什么样。**

不过对于 `datafusion` 来说，创建一个 schema 很简单，比如：

[复制代码](#)

```
1 let schema = Arc::new(Schema::new(vec![
2     Field::new("ip", DataType::Utf8, false),
3     Field::new("code", DataType::Int32, false),
4 ]));
```

为了最大的灵活性，我们可以对应地构建一个简单的 schema 定义文件，里面每个字段按顺序对应 `nginx` 日志的字段：

[复制代码](#)

```
1 ---
2 - name: ip
3   type: string
4 - name: unused1
5   type: string
6 - name: unused2
7   type: string
8 - name: date
9   type: string
10 - name: method
11  type: string
12 - name: url
13  type: string
14 - name: version
15  type: string
16 - name: code
17  type: integer
18 - name: len
19  type: integer
20 - name: unused3
21  type: string
22 - name: ua
23  type: string
```

这样，未来如果遇到不一样的日志文件，我们可以修改 schema 的定义，而无需修改程序本身。

对于这个 schema 定义文件, 使用 [serde](#) 和 [serde-yaml](#) 来读取, 然后再实现 From trait 把 SchemaField 对应到 datafusion 的 Field 结构:

[复制代码](#)

```

1  #[derive(Debug, Clone, Serialize, Deserialize, PartialEq, Eq, PartialOrd, Ord,
2  #[serde(rename_all = "snake_case")]]
3  pub enum SchemaDataType {
4      /// Int64
5      Integer,
6      /// Utf8
7      String,
8      /// Date64,
9      Date,
10 }
11
12 #[derive(Serialize, Deserialize, Debug, Clone, PartialEq, Eq, Hash, PartialOrd
13 struct SchemaField {
14     name: String,
15     #[serde(rename = "type")]
16     pub(crate) data_type: SchemaDataType,
17 }
18
19 #[derive(Serialize, Deserialize, Debug, Clone, PartialEq, Eq, Hash, PartialOrd
20 struct SchemaFields(Vec<SchemaField>);
21
22 impl From<SchemaDataType> for DataType {
23     fn from(dt: SchemaDataType) -> Self {
24         match dt {
25             SchemaDataType::Integer => Self::Int64,
26             SchemaDataType::Date => Self::Date64,
27             SchemaDataType::String => Self::Utf8,
28         }
29     }
30 }
31
32 impl From<SchemaField> for Field {
33     fn from(f: SchemaField) -> Self {
34         Self::new(&f.name, f.data_type.into(), false)
35     }
36 }
37
38 impl From<SchemaFields> for SchemaRef {
39     fn from(fields: SchemaFields) -> Self {
40         let fields: Vec<Field> = fields.0.into_iter().map(|f| f.into()).collect
41         Arc::new(Schema::new(fields))
42     }
43 }

```

有了这个基本的 schema 转换的功能，就可以构建我们的 nginx 日志处理结构及其功能了：

[复制代码](#)

```
1  /// nginx 日志处理的数据结构
2  pub struct NginxLog {
3      ctx: ExecutionContext,
4  }
5
6  impl NginxLog {
7      /// 根据 schema 定义，数据文件以及分隔符构建 NginxLog 结构
8      pub async fn try_new(schema_file: &str, data_file: &str, delim: u8) -> Res
9          let content = tokio::fs::read_to_string(schema_file).await?;
10         let fields: SchemaFields = serde_yaml::from_str(&content)?;
11         let schema = SchemaRef::from(fields);
12
13         let mut ctx = ExecutionContext::new();
14         let options = CsvReadOptions::new()
15             .has_header(false)
16             .delimiter(delim)
17             .schema(&schema);
18         ctx.register_csv("nginx", data_file, options).await?;
19
20         Ok(Self { ctx })
21     }
22
23     /// 进行 sql 查询
24     pub async fn query(&mut self, query: &str) -> Result<Arc<dyn DataFrame>> {
25         let df = self.ctx.sql(query).await?;
26         Ok(df)
27     }
28 }
```

仅仅写了 80 行代码，就完成了 nginx 日志文件的读取、解析和查询功能，其中 50 行代码还是为了处理 schema 配置文件。是不是有点不敢相信自己的眼睛？

datafusion/arrow 也太强大了吧？这个简洁的背后，是 10w 行 arrow 代码和 1w 行 datafusion 代码的功劳。

再来写段代码调用它：

[复制代码](#)

```
1  #[tokio::main]
```


```

2  async fn main() -> Result<()> {
3      let mut nginx_log =
4          NginxLog::try_new("fixtures/log_schema.yml", "fixtures/nginx_logs.csv"
5      // 从 stdin 中按行读取内容，当做 sql 查询，进行处理
6      let stdin = io::stdin();
7      let mut lines = stdin.lock().lines();
8
9      while let Some(Ok(line)) = lines.next() {
10         if !line.starts_with("--") {
11             println!("{}", line);
12             // 读到一行 sql，查询，获取 dataframe
13             let df = nginx_log.query(&line).await?;
14             // 简单显示 dataframe
15             df.show().await?;
16         }
17     }
18
19     Ok(())
20 }

```

在这段代码里，我们从 stdin 中获取内容，把每一行输入都作为一个 SQL 语句传给 nginx_log.query，然后显示查询结果。

来测试一下：

 复制代码

```

1  > echo "SELECT ip, count(*) as total, cast(avg(len) as int) as avg_len FROM ng
2  SELECT ip, count(*) as total, cast(avg(len) as int) as avg_len FROM nginx GROU
3  +-----+-----+-----+
4  | ip                | total | avg_len |
5  +-----+-----+-----+
6  | 216.46.173.126    | 2350  | 220     |
7  | 180.179.174.219   | 1720  | 292     |
8  | 204.77.168.241    | 1439  | 340     |
9  | 65.39.197.164     | 1365  | 241     |
10 | 80.91.33.133       | 1202  | 243     |
11 | 84.208.15.12       | 1120  | 197     |
12 | 74.125.60.158      | 1084  | 300     |
13 | 119.252.76.162     | 1064  | 281     |
14 | 79.136.114.202     | 628   | 280     |
15 | 54.207.57.55       | 532   | 289     |
16 +-----+-----+-----+

```

是不是挺厉害？我们可以充分利用 SQL 的强大表现力，做各种复杂的查询。不光如此，还可以从一个包含了多个 sql 语句的文件中，一次性做多个查询。比如我创建了这样一个文

件 analyze.sql :

[复制代码](#)

```

1 -- 查询 ip 前 10 名
2 SELECT ip, count(*) as total, cast(avg(len) as int) as avg_len FROM nginx GROU
3 -- 查询 UA 前 10 名
4 select ua, count(*) as total from nginx group by ua order by total desc limit
5 -- 查询访问最多的 url 前 10 名
6 select url, count(*) as total from nginx group by url order by total desc limi
7 -- 查询访问返回 body 长度前 10 名
8 select len, count(*) as total from nginx group by len order by total desc limi
9 -- 查询 HEAD 请求
10 select ip, date, url, code, ua from nginx where method = 'HEAD' limit 10
11 -- 查询状态码是 403 的请求
12 select ip, date, url, ua from nginx where code = 403 limit 10
13 -- 查询 UA 为空的请求
14 select ip, date, url, code from nginx where ua = '-' limit 10
15 -- 复杂查询, 找返回 body 长度的 percentile 在 0.5-0.7 之间的数据
16 select * from (select ip, date, url, ua, len, PERCENT_RANK() OVER (ORDER BY le

```

那么, 我可以这样获取结果:

[复制代码](#)

```

1 > cat fixtures/analyze.sql | cargo run --example log --quiet
2 SELECT ip, count(*) as total, cast(avg(len) as int) as avg_len FROM nginx GROU
3 +-----+-----+-----+
4 | ip          | total | avg_len |
5 +-----+-----+-----+
6 | 216.46.173.126 | 2350 | 220     |
7 | 180.179.174.219 | 1720 | 292     |
8 | 204.77.168.241 | 1439 | 340     |
9 | 65.39.197.164  | 1365 | 241     |
10 | 80.91.33.133   | 1202 | 243     |
11 | 84.208.15.12   | 1120 | 197     |
12 | 74.125.60.158  | 1084 | 300     |
13 | 119.252.76.162 | 1064 | 281     |
14 | 79.136.114.202 | 628  | 280     |
15 | 54.207.57.55   | 532  | 289     |
16 +-----+-----+-----+
17 select ua, count(*) as total from nginx group by ua order by total desc limit
18 +-----+-----+
19 | ua                                | total |
20 +-----+-----+
21 | Debian APT-HTTP/1.3 (1.0.1ubuntu2) | 11830 |
22 | Debian APT-HTTP/1.3 (0.9.7.9)      | 11365 |
23 | Debian APT-HTTP/1.3 (0.8.16~exp12ubuntu10.21) | 6719 |
24 | Debian APT-HTTP/1.3 (0.8.16~exp12ubuntu10.16) | 5740 |

```



```
25 | Debian APT-HTTP/1.3 (0.8.16~exp1ubuntu10.22) | 3855 |
26 | Debian APT-HTTP/1.3 (0.8.16~exp1ubuntu10.17) | 1827 |
27 | Debian APT-HTTP/1.3 (0.8.16~exp1ubuntu10.7) | 1255 |
28 | urlgrabber/3.9.1 yum/3.2.29 | 792 |
29 | Debian APT-HTTP/1.3 (0.9.7.8) | 750 |
30 | urlgrabber/3.9.1 yum/3.4.3 | 708 |
31 +-----+-----+
32 select url, count(*) as total from nginx group by url order by total desc limit 10
33 +-----+-----+
34 | url | total |
35 +-----+-----+
36 | /downloads/product_1 | 30285 |
37 | /downloads/product_2 | 21104 |
38 | /downloads/product_3 | 73 |
39 +-----+-----+
40 select len, count(*) as total from nginx group by len order by total desc limit 10
41 +-----+-----+
42 | len | total |
43 +-----+-----+
44 | 0 | 13413 |
45 | 336 | 6652 |
46 | 333 | 3771 |
47 | 338 | 3393 |
48 | 337 | 3268 |
49 | 339 | 2999 |
50 | 331 | 2867 |
51 | 340 | 1629 |
52 | 334 | 1393 |
53 | 332 | 1240 |
54 +-----+-----+
55 select ip, date, url, code, ua from nginx where method = 'HEAD' limit 10
56 +-----+-----+-----+-----+
57 | ip | date | url | code |
58 +-----+-----+-----+-----+
59 | 184.173.149.15 | 23/May/2015:15:05:53 +0000 | /downloads/product_2 | 403 |
60 | 5.153.24.140 | 23/May/2015:17:05:30 +0000 | /downloads/product_2 | 200 |
61 | 5.153.24.140 | 23/May/2015:17:05:33 +0000 | /downloads/product_2 | 403 |
62 | 5.153.24.140 | 23/May/2015:17:05:34 +0000 | /downloads/product_2 | 403 |
63 | 5.153.24.140 | 23/May/2015:17:05:52 +0000 | /downloads/product_2 | 200 |
64 | 5.153.24.140 | 23/May/2015:17:05:43 +0000 | /downloads/product_2 | 200 |
65 | 5.153.24.140 | 23/May/2015:17:05:42 +0000 | /downloads/product_2 | 200 |
66 | 5.153.24.140 | 23/May/2015:17:05:46 +0000 | /downloads/product_2 | 200 |
67 | 5.153.24.140 | 23/May/2015:18:05:10 +0000 | /downloads/product_2 | 200 |
68 | 184.173.149.16 | 24/May/2015:18:05:37 +0000 | /downloads/product_2 | 403 |
69 +-----+-----+-----+-----+
70 select ip, date, url, ua from nginx where code = 403 limit 10
71 +-----+-----+-----+-----+
72 | ip | date | url | ua |
73 +-----+-----+-----+-----+
74 | 184.173.149.15 | 23/May/2015:15:05:53 +0000 | /downloads/product_2 | Wget/1.
75 | 5.153.24.140 | 23/May/2015:17:05:33 +0000 | /downloads/product_2 | Wget/1.
76 | 5.153.24.140 | 23/May/2015:17:05:34 +0000 | /downloads/product_2 | Wget/1.
```

```

77 | 184.173.149.16 | 24/May/2015:18:05:37 +0000 | /downloads/product_2 | Wget/1.
78 | 195.88.195.153 | 24/May/2015:23:05:05 +0000 | /downloads/product_2 | curl/7.
79 | 184.173.149.15 | 25/May/2015:04:05:14 +0000 | /downloads/product_2 | Wget/1.
80 | 87.85.173.82 | 17/May/2015:14:05:07 +0000 | /downloads/product_2 | Wget/1.
81 | 87.85.173.82 | 17/May/2015:14:05:11 +0000 | /downloads/product_2 | Wget/1.
82 | 194.76.107.17 | 17/May/2015:16:05:50 +0000 | /downloads/product_2 | Wget/1.
83 | 194.76.107.17 | 17/May/2015:17:05:40 +0000 | /downloads/product_2 | Wget/1.
84 +-----+-----+-----+-----+-----+-----+-----+-----+
85 select ip, date, url, code from nginx where ua = '-' limit 10
86 +-----+-----+-----+-----+-----+-----+-----+-----+
87 | ip | date | url | code |
88 +-----+-----+-----+-----+-----+-----+-----+-----+
89 | 217.168.17.150 | 01/Jun/2015:14:06:45 +0000 | /downloads/product_2 | 200 |
90 | 217.168.17.180 | 01/Jun/2015:14:06:15 +0000 | /downloads/product_2 | 200 |
91 | 217.168.17.150 | 01/Jun/2015:14:06:18 +0000 | /downloads/product_1 | 200 |
92 | 204.197.211.70 | 24/May/2015:06:05:02 +0000 | /downloads/product_2 | 200 |
93 | 91.74.184.74 | 29/May/2015:14:05:17 +0000 | /downloads/product_2 | 403 |
94 | 91.74.184.74 | 29/May/2015:15:05:43 +0000 | /downloads/product_2 | 403 |
95 | 91.74.184.74 | 29/May/2015:22:05:53 +0000 | /downloads/product_2 | 403 |
96 | 217.168.17.5 | 31/May/2015:02:05:16 +0000 | /downloads/product_2 | 200 |
97 | 217.168.17.180 | 20/May/2015:23:05:22 +0000 | /downloads/product_2 | 200 |
98 | 204.197.211.70 | 21/May/2015:02:05:34 +0000 | /downloads/product_2 | 200 |
99 +-----+-----+-----+-----+-----+-----+-----+-----+
100 select * from (select ip, date, url, ua, len, PERCENT_RANK() OVER (ORDER BY le
101 +-----+-----+-----+-----+-----+-----+-----+-----+
102 | ip | date | url | ua |
103 +-----+-----+-----+-----+-----+-----+-----+-----+
104 | 54.229.83.18 | 26/May/2015:00:05:34 +0000 | /downloads/product_1 | urlgrab
105 | 54.244.37.198 | 18/May/2015:10:05:39 +0000 | /downloads/product_1 | urlgrab
106 | 67.132.206.254 | 29/May/2015:07:05:52 +0000 | /downloads/product_1 | urlgrab
107 | 128.199.60.184 | 24/May/2015:00:05:09 +0000 | /downloads/product_1 | urlgrab
108 | 54.173.6.142 | 27/May/2015:14:05:21 +0000 | /downloads/product_1 | urlgrab
109 | 104.156.250.12 | 03/Jun/2015:11:06:51 +0000 | /downloads/product_1 | urlgrab
110 | 115.198.47.126 | 25/May/2015:11:05:13 +0000 | /downloads/product_1 | urlgrab
111 | 198.105.198.4 | 29/May/2015:07:05:34 +0000 | /downloads/product_1 | urlgrab
112 | 107.23.164.80 | 31/May/2015:09:05:34 +0000 | /downloads/product_1 | urlgrab
113 | 108.61.251.29 | 31/May/2015:10:05:16 +0000 | /downloads/product_1 | urlgrab
114 +-----+-----+-----+-----+-----+-----+-----+-----+

```

小结

今天我们介绍了如何使用 Rust 处理存放在关系数据库中的结构化数据，以及存放在文件系统中的半结构化数据。

虽然在工作中，我们不太会使用 arrow/datafusion 去创建某个“下一代”的数据处理平台，但拥有了处理半结构化数据的能力，可以解决很多非常实际的问题。

比如每隔 10 分钟扫描 Nginx / CDN，以及应用服务器过去 10 分钟的日志，找到某些非正常的访问，然后把该用户 / 设备的访问切断一阵子。这样的特殊需求，一般的数据平台很难处理，需要我们自己撰写代码来实现。此时，arrow/datafusion 这样的工具就很方便。

思考题

1. 请你自己阅读 diesel 或者 sea-orm 的文档，然后尝试把我们直接用 sqlx 构建的用户注册 / 登录的功能使用 diesel 或者 sea-orm 实现。
2. datafusion 不但支持 csv，还支持 ndJSON / parquet / avro 等数据类型。如果你公司的生产环境下有这些类型的半结构化数据，可以尝试着阅读相关文档，使用 datafusion 来读取和查询它们。

感谢你的收听。恭喜你完成了第 44 次 Rust 学习，打卡之旅马上就要结束啦，我们下节课见。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 43 | 生产环境：真实世界下的一个 Rust 项目包含哪些要素？

小争哥新书

数据结构与算法之美

图书+专栏，双管齐下，拿下算法

打包价 **¥159** 原价¥319

仅限 300 套



精选留言 (3)

写留言



乌龙痘

2021-12-10

课程进入到了尾声，感觉啥也没学到，感觉又学到了很多。整个课程内容夯实，体系结构清晰。值得反复品味。这是 Rust 编程的第一课，而真正的 Rust 之旅才刚刚开始。



罗杰

2021-12-10

哈，我也觉得 ORM 太过笨重，还是 sqlx 直观。我的项目全部都是 sqlx，C++ 和 Go 都是。



pedro

2021-12-10

最近工作很忙，经常加班，有时间抽空来看看，仍然收获颇丰。

看完今天这一章，有一种强烈的感受：这不仅是 Rust 编程第一课，很有可能也是唯一的一课，内容太丰富了。

展开 ✓

