// src/main.rs

mod handlers;

mod models;

mod routes;

mod state;

mod middleware;

// 库模块导入

use tokio::net::TcpListener;

use tracing\_subscriber::fmt;

use sqlx::MySqlPool;

// 分离模块导入

use routes::create\_routes;

use state::AppState;

#[tokio::main]

async fn main() {

// 加载.env文件

dotenv::dotenv().ok();

let db\_url = std::env::var("DATABASE\_URL")

.expect("DATABASE\_URL must be set in .env");

// 初始化日志

fmt::init();

// 创建 MySQL 连接池

let db\_pool = MySqlPool::connect(&db\_url)

.await

.expect("Failed to create MySQL pool");

let state = AppState::new(db\_pool);

// 构建路由(注入状态)

let app = create\_routes().with\_state(state);

// 启动服务器

let listener = TcpListener::bind("0.0.0.0:3000").await.unwrap();

axum::serve(listener, app).await.unwrap();

}

// src/state.rs

use sqlx::MySqlPool;

#[derive(Clone)]

pub struct AppState {

pub db\_pool: MySqlPool,

}

impl AppState {

/// 创建带数据库连接池的应用状态

pub fn new(db\_pool: MySqlPool) -> Self {

Self { db\_pool }

}

/// 从环境变量初始化（高级用法）

pub async fn from\_env() -> sqlx::Result<Self> {

let db\_url = std::env::var("DATABASE\_URL")

.expect("DATABASE\_URL must be set");

let pool = MySqlPool::connect(&db\_url).await?;

Ok(Self::new(pool))

}

}

// src/routes.rs

// 库模块导入

use axum::{routing::{get, post}, Router, middleware};

use tower\_http::cors::{CorsLayer, Any};

use axum::http::{Method, HeaderName};

// 分离模块导入

use super::handlers;

use crate::middleware::auth\_middleware;

use crate::state::AppState;

// 构建路由并返回 Router 实例

pub fn create\_routes() -> Router<AppState> {

// CORS 中间件

let cors = CorsLayer::new()

.allow\_origin(Any)

.allow\_methods(vec![Method::GET, Method::POST])

.allow\_headers(vec![HeaderName::from\_static("content-type")]);

let public\_routes = Router::new()

.route("/", get(handlers::root))

.route("/register", post(handlers::register))

.route("/login", post(handlers::login));

let protected\_routes = Router::new() // 被保护的路由

.route("/protected", get(handlers::protected))

.route\_layer(middleware::from\_fn(auth\_middleware));

Router::new()

.merge(public\_routes)

.merge(protected\_routes)

.layer(cors)

}

// src/middleware.rs

use axum::{

body::Body,

http::{Request, StatusCode},

middleware::Next,

response::Response,

};

use jsonwebtoken::{decode, DecodingKey, Validation};

use crate::models::Claims;

pub async fn auth\_middleware(

request: Request<Body>,

next: Next,

) -> Result<Response, StatusCode> {

let token = request.headers()

.get("Authorization")

.and\_then(|v| v.to\_str().ok())

.and\_then(|s| s.strip\_prefix("Bearer "));

let token = token.ok\_or(StatusCode::UNAUTHORIZED)?;

decode::<Claims>(

token,

&DecodingKey::from\_secret(std::env::var("JWT\_SECRET").unwrap().as\_ref()),

&Validation::default()

).map\_err(|\_| StatusCode::UNAUTHORIZED)?;

Ok(next.run(request).await)

}

// src/handlers.rs

// 库模块导入

use axum::{

http::StatusCode,

Json,

};

use axum::extract::State;

use sqlx::MySqlPool;

use std::error::Error;

use argon2::{

password\_hash::{PasswordHash, PasswordVerifier, SaltString},

Argon2, PasswordHasher

};

use rand\_core::OsRng;

use jsonwebtoken::{encode, EncodingKey, Header};

use std::time::{SystemTime, UNIX\_EPOCH};

// 分离模块导入

use crate::{models::{

RegisterRequest,

RegisterResponse,

LoginRequest,

LoginResponse,

User,

Claims

}, state::AppState};

// 根路径处理函数

pub async fn root() -> &'static str {

"Hello, World!"

}

// 注册处理函数

pub async fn register(

State(state): State<AppState>,// 注入状态

Json(payload): Json<RegisterRequest>,// 解析为请求结构体

) -> Result<Json<RegisterResponse>, StatusCode> {

// 生成随机盐值

let salt = SaltString::generate(&mut OsRng);

// 配置Argon2参数

let argon2 = Argon2::default();

// 生成密码哈希

let password\_hash = argon2

.hash\_password(payload.password.as\_bytes(), &salt)

.map\_err(|\_| StatusCode::INTERNAL\_SERVER\_ERROR)?

.to\_string();

// 存储到数据库 (替换原有的明文存储)

sqlx::query!(

"INSERT INTO user\_info (account, password, username) VALUES (?, ?, ?)",

payload.account,

password\_hash,

payload.username,

)

.execute(&state.db\_pool)

.await

.map\_err(|\_| StatusCode::INTERNAL\_SERVER\_ERROR)?;

Ok(Json(RegisterResponse { success: true }))

}

// 登录处理函数

pub async fn login(

State(state): State<AppState>,

Json(payload): Json<LoginRequest>,

) -> Result<Json<LoginResponse>, StatusCode> {

match validate\_credentials(&state.db\_pool, &payload.account, &payload.password).await {

Ok(Some(username)) => {

// 生成JWT令牌

let token = generate\_jwt(&payload.account)

.map\_err(|\_| StatusCode::INTERNAL\_SERVER\_ERROR)?;

Ok(Json(LoginResponse {

username,

token,

}))

}

Ok(None) => Err(StatusCode::UNAUTHORIZED), // 认证失败， 返回401

Err(\_) => Err(StatusCode::INTERNAL\_SERVER\_ERROR),// 服务器内部错误， 返回500

}

}

// 登录验证逻辑函数

async fn validate\_credentials(

db\_pool: &MySqlPool,

account: &str,

password: &str,

) -> Result<Option<String>, Box<dyn Error>> {

// 从数据库中查询用户信息

let user = sqlx::query\_as::<\_, User>(

"SELECT \* FROM user\_info WHERE account = ?"

)

.bind(account)

.fetch\_optional(db\_pool)

.await?;

match user {

Some(user) => {

// 验证密码哈希

let parsed\_hash = PasswordHash::new(&user.password)

.map\_err(|\_| "密码哈希解析失败")?;

let argon2 = Argon2::default();

match argon2.verify\_password(password.as\_bytes(), &parsed\_hash) {

Ok(\_) => Ok(user.username), // 验证成功

Err(\_) => Ok(None), // 密码不匹配

}

}

None => Ok(None), // 用户不存在

}

}

// JWT生成函数

fn generate\_jwt(account: &str) -> Result<String, Box<dyn Error>> {

let now = SystemTime::now()

.duration\_since(UNIX\_EPOCH)?

.as\_secs() as usize;

let exp = now + 3600; // 1小时有效期

let claims = Claims {

sub: account.to\_string(),

exp,

iat: now,

};

let token = encode(

&Header::default(),

&claims,

&EncodingKey::from\_secret(std::env::var("JWT\_SECRET")?.as\_ref())

)?;

Ok(token)

}

// 保护处理函数

pub async fn protected() -> &'static str {

"Protected content!"

}

// src/models.rs

// 库模块导入

use serde::{Deserialize, Serialize};

use sqlx::FromRow;

// 用户表模型

#[derive(Debug, Deserialize, Serialize, FromRow)]

pub struct User {

pub account: String, // 主键 + 非空

pub password: String, // 非空

pub username: Option<String>, // 允许为空，保留Option

}

// 注册请求结构体

#[derive(Deserialize)]

pub struct RegisterRequest {

pub account: String,

pub password: String,

pub username: String,

}

// 注册响应结构体

#[derive(Serialize)]

pub struct RegisterResponse {

pub success: bool,

}

// 登录请求模型

#[derive(Deserialize)]

pub struct LoginRequest {

pub account: String,

pub password: String,

}

// 登录响应模型

#[derive(Serialize)]

pub struct LoginResponse {

pub username: String,

pub token: String, // JWT令牌

}

// JWT

#[derive(Debug, Serialize, Deserialize)]

pub struct Claims {

pub sub: String, // 用户账号

pub exp: usize, // 过期时间

pub iat: usize, // 签发时间

}

# src/Cargo.toml

[package]

name = "chat"

version = "0.1.0"

edition = "2024"

[dependencies]

#序列化与反序列化

serde = { version = "1.0", features = ["derive"] }

#axum基于tokio构建

tokio = { version = "1.46", features = ["full"] }

#日志依赖

tracing = "0.1"

tracing-subscriber = "0.3"

#数据库驱动

sqlx = { version = "0.7", features = ["mysql", "runtime-tokio", "tls-native-tls", "macros"] }

#读取环境变量

dotenv = "0.15"

#跨域中间件

tower-http = { version = "0.5.0", features = ["cors"] }

http = "0.2"

#密码哈希

argon2 = "0.5.3"

#用户认证和授权

jsonwebtoken = "9.0"

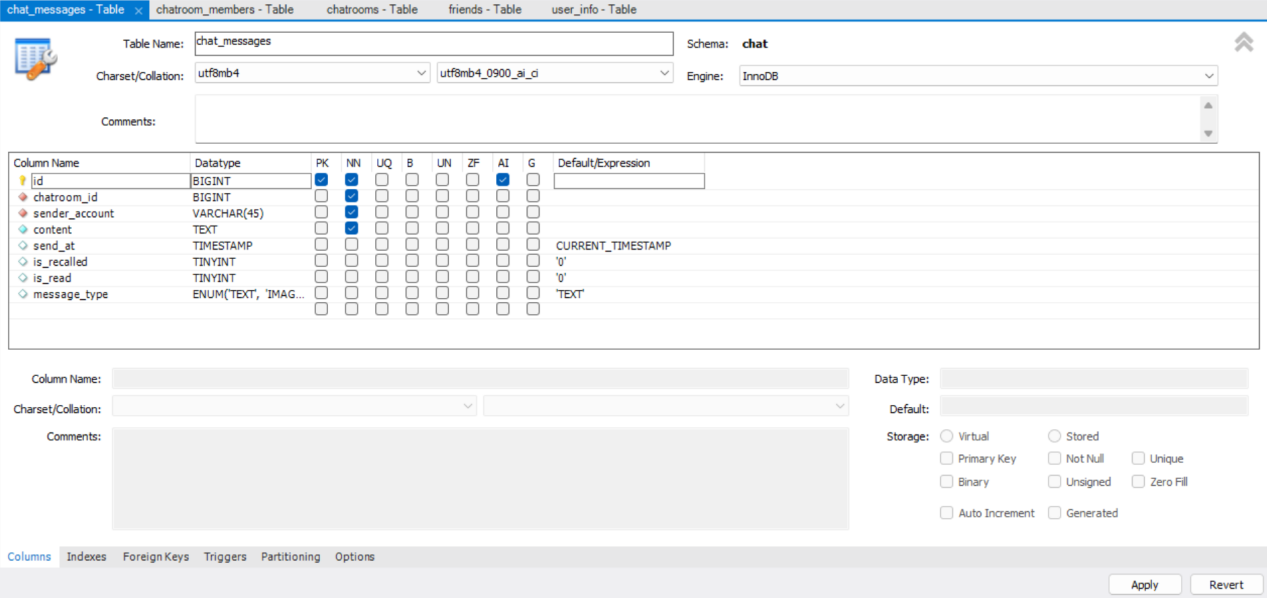
#随机数

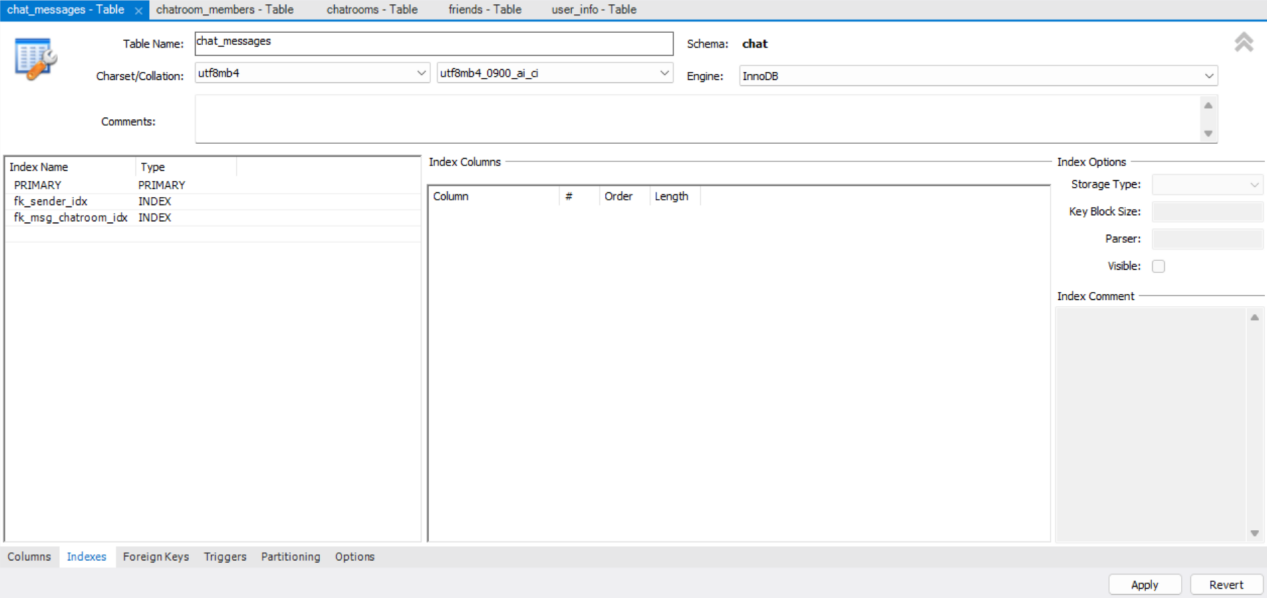
rand\_core = { version = "0.6", features = ["std"] }

#Axum框架

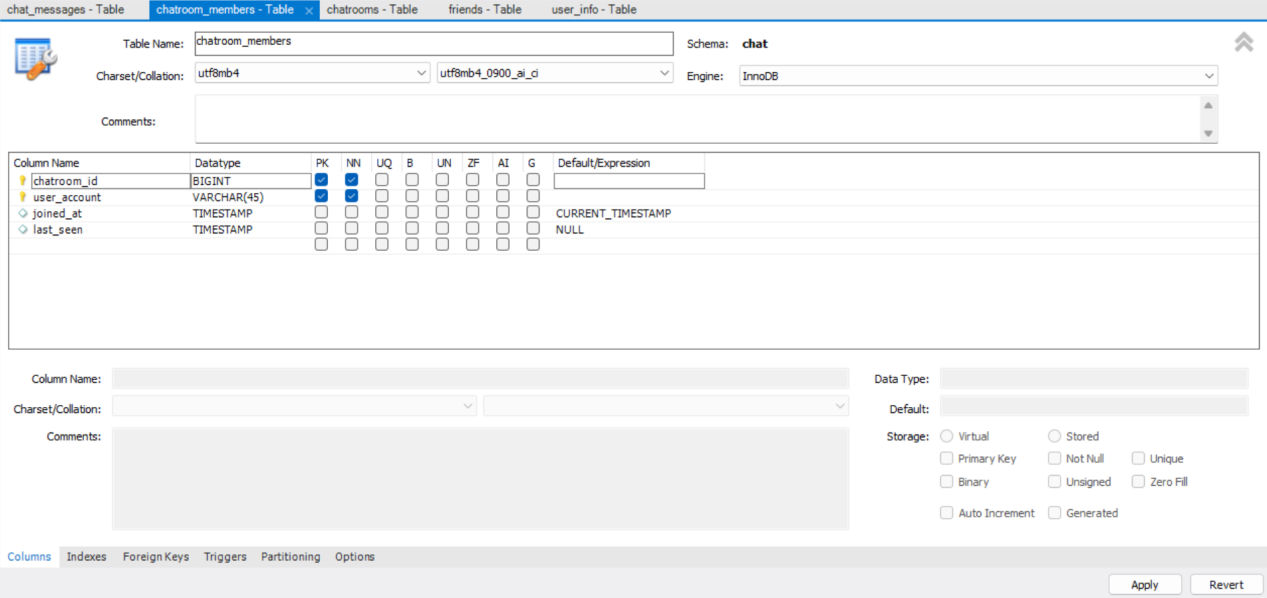
axum = "0.8.4"

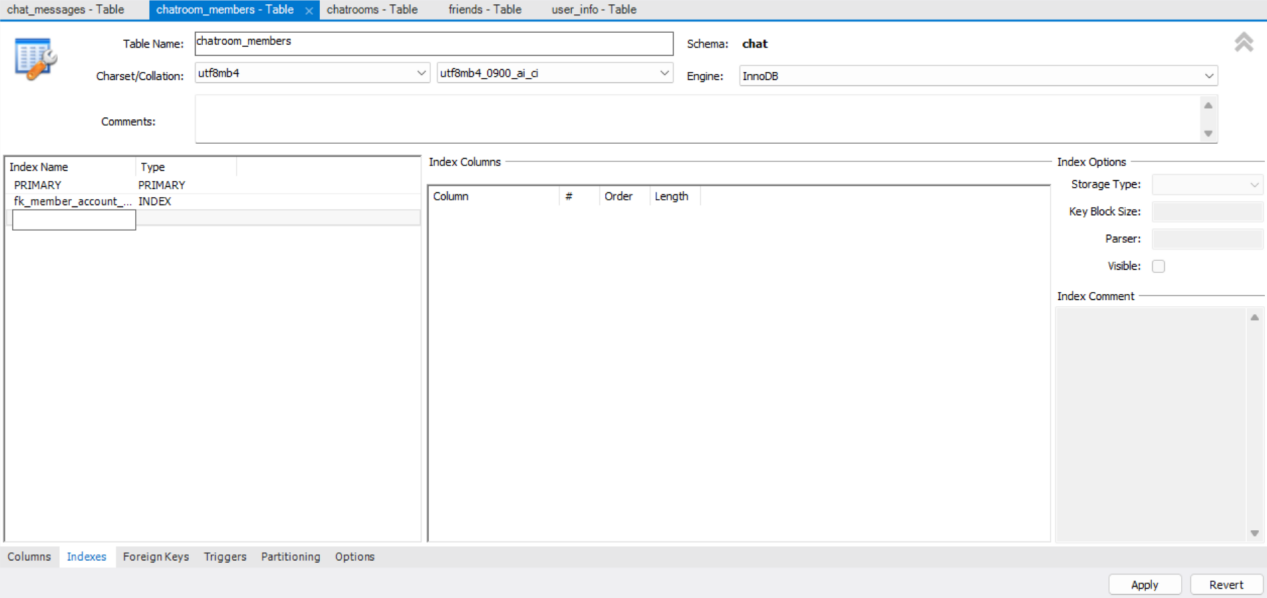
chat\_messages表



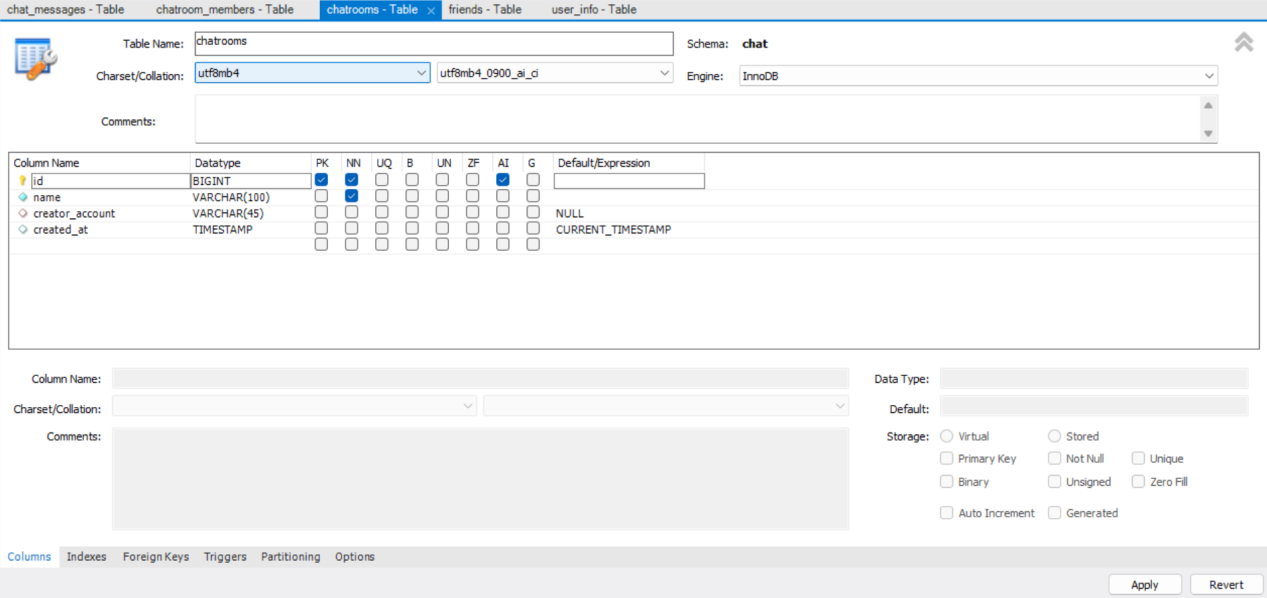


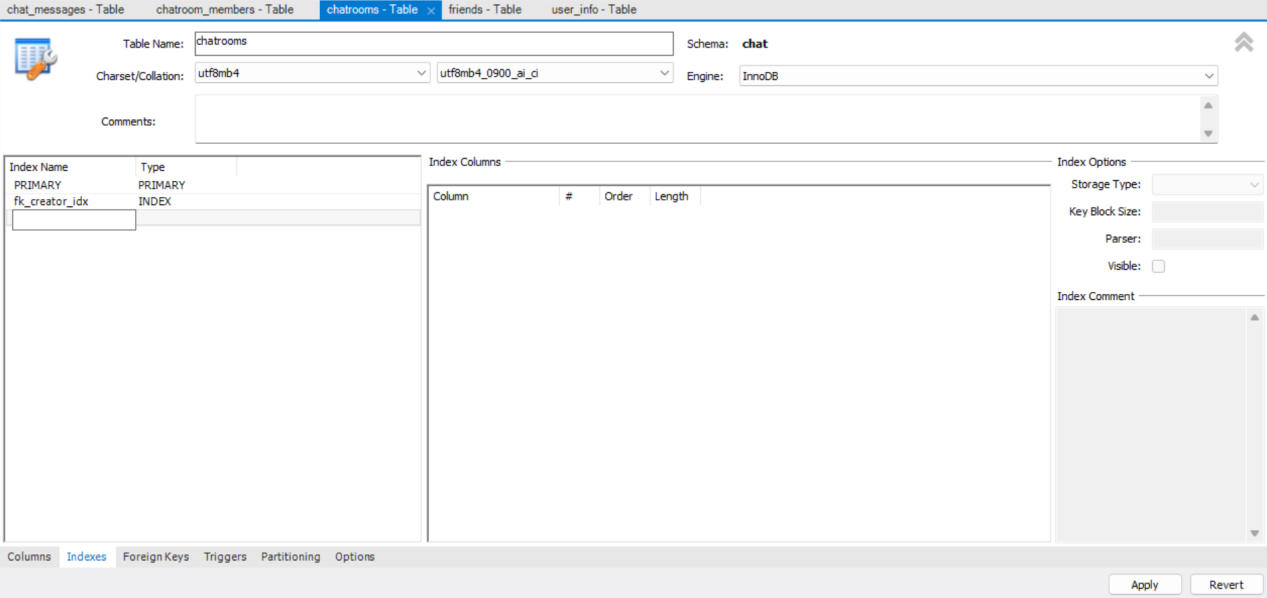
Chatroom\_members表



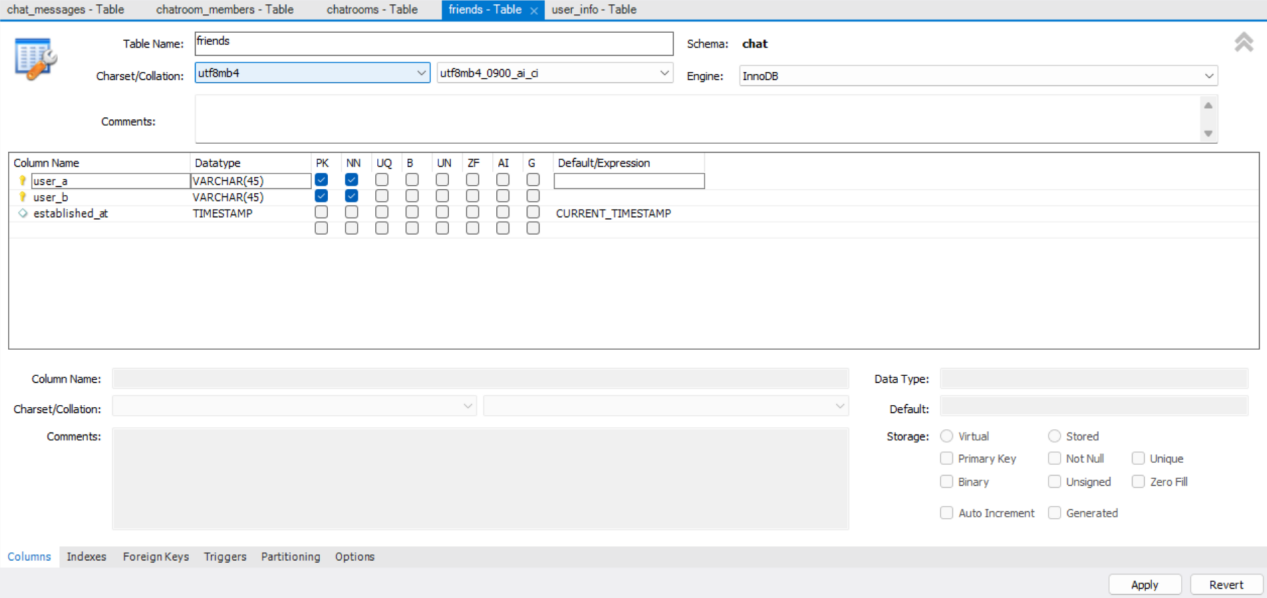


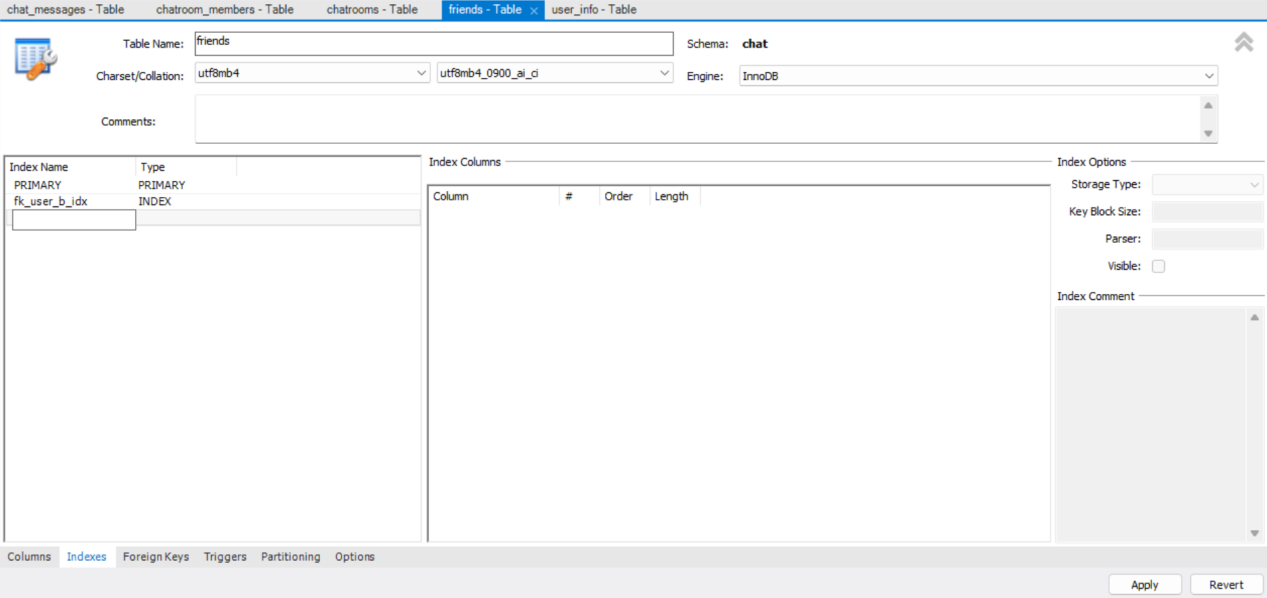
chatrroms表





Friend表





User\_info表

