南京理工大学

NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

移动应用开发技术--作业文档报告

学生姓名: 甘荣锋

班　　级 : 9221065501

学　　号 : 922106840222

题　　目 : HarmonyOS 个人健康管理应用开发

指导教师: 韦建楠

2025 年 04 月

目录

第一章 概述

1.1 系统概述

本系统为基于 ArkTs 的 Stage 模型实现的健康管理应用，本应用基于 Harmony 平台，实现了睡眠时长，每日饮水量，每日心率，以及步数记录等健康数据记录、进行可视化分析等功能，涵盖 UI 设计、数据管理、组件通信等核心知识点。

1.2 已实现所要求的功能

使用 SQLite 关系型数据库 relationalStore 存储至少 3 类健康数据（如步数、睡眠时长、饮水记录，心率记录，身高，体重等信息），使用 TextInput 或 Dialog 对每个数据给出独立的输入框，并实现存储的健康数据的增删改查功能。

共实现了用户注册，用户登录，信息概览，数据输入，步数显示，个人信息，修改个人信息，详细数据分析等 UI 界面。并且采用 Tabs 实现底部导航栏。使用了 Button，Progress，Chart，Divider，List 等多个系统组件。并集成了包括但不限于 Preferences API，Router，Picker，Want，Canvas，Chart，relationalStore，notificationManager 等系统级 API。

1.3 实现进阶功能：

数据可视化：使用 Canvas 与 Chart 绘制步数柱状图以及心率趋势折线图等图表。

通知功能：当每日步数目标达成或饮水量总数达到目标时触发系统通知。

1.4 技术规范：

该应用使用 Stage 模型开发，使用 SQLite 进行数据存储，实现数据的增删改查功能，并且通过 Preference 实现在登录页面是否记住用户密码等偏好设置。

页面间跳转均使用 Router 模块。

并且程序代码包含但不限于以下装饰器：@Component @Styles @Extend @Builder 等。并且设计了良好的用户交互界面，添加了便于用户理解的图标，视觉上更加和谐美观。

例如在 mainHome.ets 文件中：

```
// 定义可复用的文本样式
@Extend(Text) function titleTextStyle() {
  .fontSize(24)
  .fontColor('#1698CE')
  .fontWeight(FontWeight.Bold)
```

```
}

// 定义可复用的Tab样式
@Styles function mainTabsStyle() {
  .width('100%')
  .layoutWeight(1)
  .height(60)
}
```

以及 Login.ets 页面中的：

```
 @Builder
  HeaderSection() {
    Column() {
      Text("个人健康管理系统").fontSize(36).fontColor('#1698CE').margin({top:'20%'})
      Image($r('app.media.avater_1'))
        .width(100)
        .height(100)
        .borderRadius(50)
        .margin({ top: '10%' })
      Text('登录')
        .fontSize(30)
        .fontWeight('bold')
        .margin({ top: 20 })
        .textAlign(TextAlign.Center)
    }
  }
```

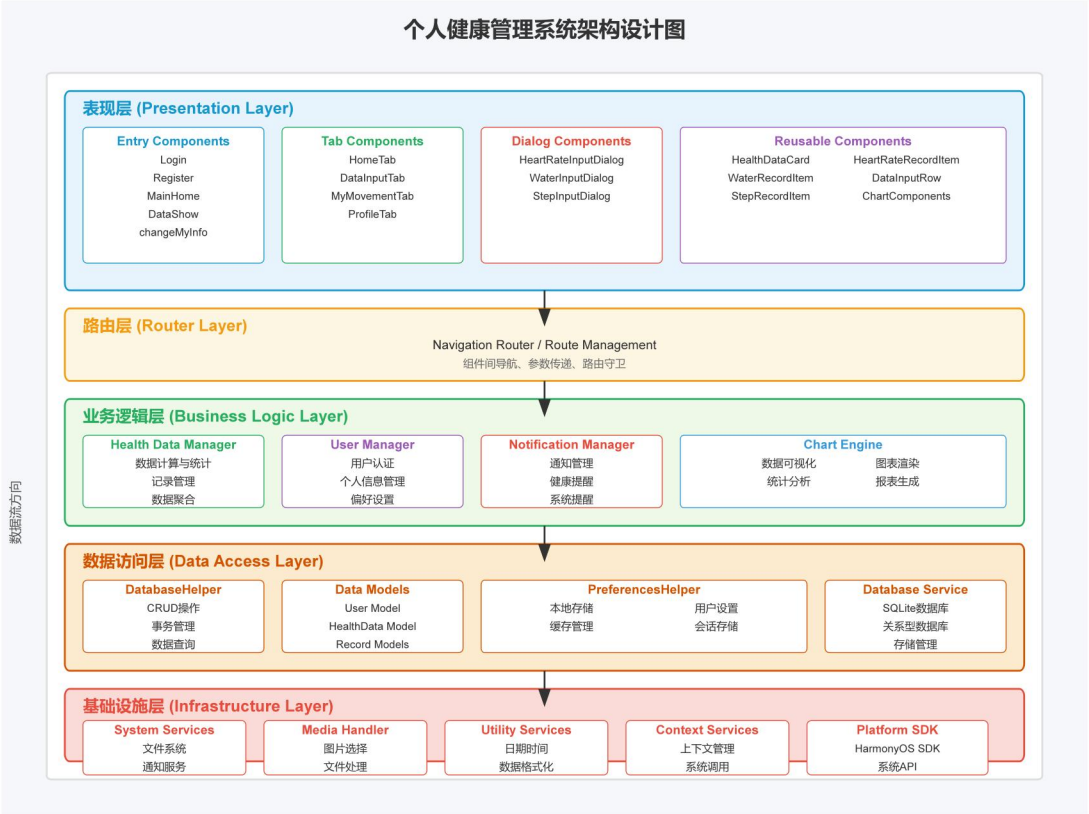
1.5 开发设备

本系统使用 DevEcoStudio 5.0.3 进行开发，测试设备使用 HarmonyOS 5.0.1(13)版本的模拟器进行测试。

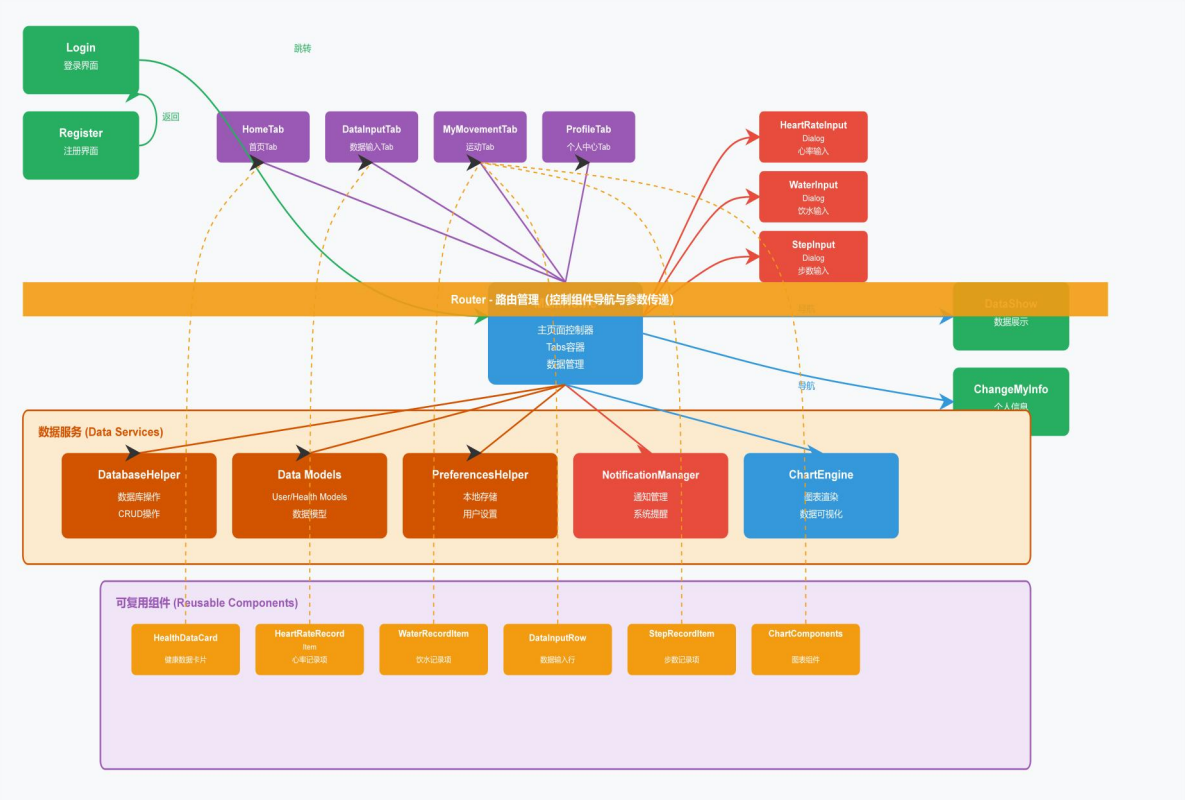
1.6 总体结构：

本应用共分为用户注册登录，健康数据输入，健康数据分析展示，用户信息修改，系统通知等五大模块，其中，健康数据分析展示模块又可细分为总体概览，心率数据和饮水量数据分析，运动步数分析等三个小模块，第三章将从这几个模块详细分析各个模块的功能。

1.7 系统架构图：

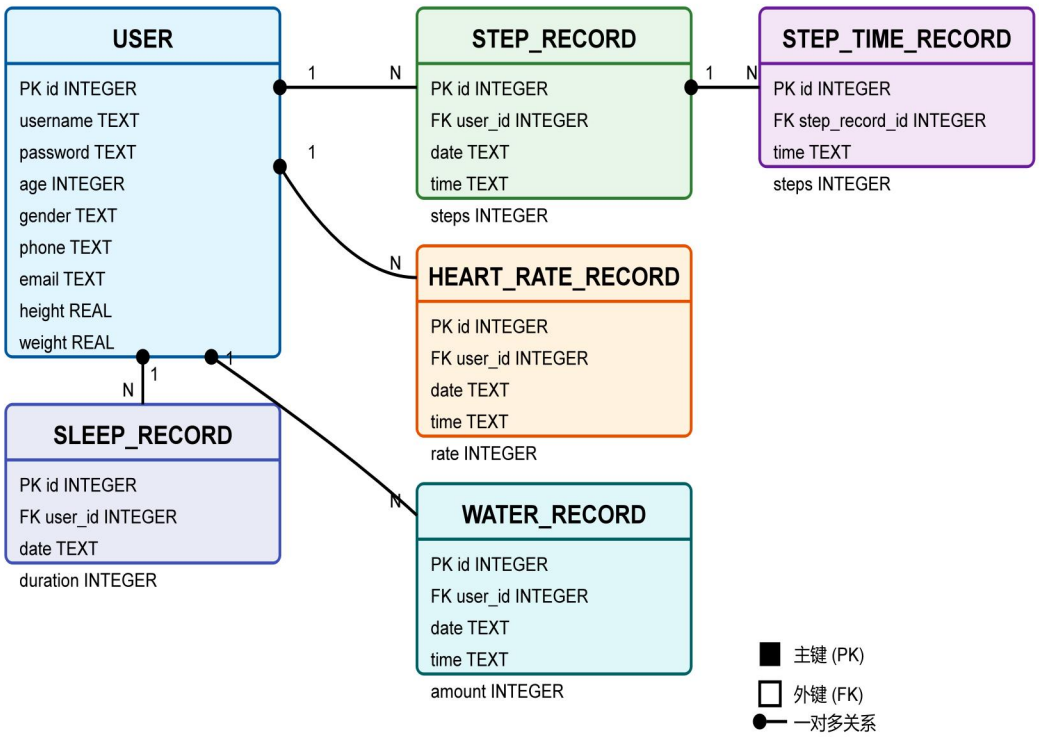


各组件关系图



第二章 数据库设计

2.1 数据库 E/R 图设计：



2.2 数据库表结构设计：

2.2.1 User 表

USER 表			
字段名	数据类型	描述	是否必填
id	INTEGER	主键,自增	是
username	TEXT	用户名	是
password	TEXT	密码	是
age	INTEGER	年龄	否
gender	TEXT	性别	否
phone	TEXT	电话	否
email	TEXT	邮箱	否
height	REAL	身高	否
weight	REAL	体重	否
avatarPath	TEXT	头像路径	否

实现语句：

```
const SQL_CREATE_USER_TABLE = 'CREATE TABLE IF NOT EXISTS user ' +
  '(id INTEGER PRIMARY KEY AUTOINCREMENT, ' +
  'username TEXT NOT NULL, ' +
  'password TEXT NOT NULL, ' +
  'age INTEGER, ' +
  'gender TEXT, ' +
  'phone TEXT, ' +
  'email TEXT, ' +
  'height REAL, ' +
  'weight REAL, ' +
  'emergency_contact TEXT, ' +
  'emergency_phone TEXT, ' +
  'medical_history TEXT, ' +
  'avatar_path TEXT)';
await this.rdbStore.executeSql(SQL_CREATE_USER_TABLE, []);
```

2.2.2 StepRecord 表：

StepRecord 表			
字段名	数据类型	描述	是否必填
id	INTEGER	主键,自增	是
userId	INTEGER	外键，关联 user 表 id	是
date	TEXT	日期，格式 YYYY-MM-DD	是
steps	INTEGER	步数	是
time	TEXT	时间，格式为 HH-MM	是

实现语句：

```
const SQL_CREATE_STEP_TABLE = 'CREATE TABLE IF NOT EXISTS step_record ' +
  '(id INTEGER PRIMARY KEY AUTOINCREMENT, ' +
  'user_id INTEGER NOT NULL, ' +
  'date TEXT NOT NULL, ' +
  'time TEXT NOT NULL, ' +
  'steps INTEGER NOT NULL, ' +
  'FOREIGN KEY (user_id) REFERENCES user(id))';
await this.rdbStore.executeSql(SQL_CREATE_STEP_TABLE, []);
```

2.2.3 HeartRateRecord 表：

HeartRateRecord			
字段名	数据类型	描述	是否必填
id	INTEGER	主键，自增	是
userid	INTEGER	用户 id，外键	是
date	TEXT	日期，格式为：YYYY-MM-DD	是
time	TEXT	时间，格式为 HH-MM	是
rate	INTEGER	心率值	是

实现语句：

```
const SQL_CREATE_HEART_RATE_TABLE = 'CREATE TABLE IF NOT EXISTS heart_rate_record ' +
  '(id INTEGER PRIMARY KEY AUTOINCREMENT, ' +
  'user_id INTEGER NOT NULL, ' +
  'date TEXT NOT NULL, ' +
  'time TEXT NOT NULL, ' +
  'rate INTEGER NOT NULL, ' +
  'FOREIGN KEY (user_id) REFERENCES user(id))';
await this.rdbStore.executeSql(SQL_CREATE_HEART_RATE_TABLE, []);
```

2.2.4 SleepRecord 表：

SleepRecord			
字段名	数据类型	描述	是否必填
id	INTEGER	主键，自增	是
userid	INTEGER	用户 id，外键	是
date	TEXT	日期，格式为 YYYY-MM-DD	是
duration	INTEGER	睡眠时间	是

实现语句：

```
const SQL_CREATE_SLEEP_TABLE = 'CREATE TABLE IF NOT EXISTS sleep_record ' +
  '(id INTEGER PRIMARY KEY AUTOINCREMENT, ' +
  'user_id INTEGER NOT NULL, ' +
  'date TEXT NOT NULL, ' +
  'duration INTEGER NOT NULL, ' +
  'FOREIGN KEY (user_id) REFERENCES user(id))';
    await this.rdbStore.executeSql(SQL_CREATE_SLEEP_TABLE, []);
```

2.2.5 WaterRecord 表

WaterRecord			
字段名	数据类型	描述	是否必填
id	INTEGER	主键，自增	是
userid	INTEGER	用户 id。外键	是
date	TEXT	日期，格式为：YYYY-MM-DD	是
time	TEXT	时间，格式为：HH-MM	是
amount	INTEGER	饮水量（ml）	是

实现语句：

```
const SQL_CREATE_WATER_TABLE = 'CREATE TABLE IF NOT EXISTS water_record ' +
  '(id INTEGER PRIMARY KEY AUTOINCREMENT, ' +
  'user_id INTEGER NOT NULL, ' +
  'date TEXT NOT NULL, ' +
  'time TEXT NOT NULL, ' +
  'amount INTEGER NOT NULL, ' +
  'FOREIGN KEY (user_id) REFERENCES user(id))';
await this.rdbStore.executeSql(SQL_CREATE_WATER_TABLE, []);
```

2.3 数据库模型类设计

2.3.1 User 模型类：

User 模型		
字段	数据类型	描述
id	number	用户 id
username	string	用户名
password	string	密码
age	number	年龄
phone	string	联系电话
email	string	邮箱
height	number	身高
weight	number	体重
avatarpath	string	头像存储路径

```
export class User {
  id?: number;
  username: string;
  password: string;
  age?: number;
  gender?: string;
  phone?: string;
  email?: string;
  height?: number;
  weight?: number;
  avatarPath?: string;
  constructor(
    username: string,
    password: string,
    id?: number,
    age?: number,
    gender?: string,
    phone?: string,
    email?: string,
    height?: number,
    weight?: number,
    avatarPath?: string
  ) {
    this.username = username;
    this.password = password;
    this.id = id;
    this.age = age;
    this.gender = gender;
    this.phone = phone;
    this.email = email;
    this.height = height;
    this.weight = weight;
    this.avatarPath = avatarPath;
  }
}
```

2.3.2 StepRecord 模型

StepRecord 模型		
字段	数据类型	描述
id	number	记录 id
userid	number	用户 id
date	string	日期（YYYY-MM-DD）
time	string	时间（HH-MM）
steps	number	步数

```
export class StepRecord {
  id?: number;
  userId: number;
  date: string;   // 格式：YYYY-MM-DD
  time: string;   // 格式：HH:MM
  steps: number;  // 步数值
  constructor(userId: number, date: string, time: string, steps: number, id?: number) {
    this.userId = userId;
    this.date = date;
    this.time = time;
    this.steps = steps;
    this.id = id;
  }
}
```

2.3.3 HeartRateRecord 模型

HeartRateRecord 模型		
字段	数据类型	描述
id	number	记录 id
userid	number	用户 id
date	string	日期（YYYY-MM-DD）
time	string	时间（HH-MM）
Rate	number	心率值

```
export class HeartRateRecord {
  id?: number;
  userId: number;
  date: string;  // 格式：YYYY-MM-DD
  time: string;  // 格式：HH:MM
  rate: number;  // 心率值
  constructor(userId: number, date: string, time: string, rate: number, id?: number) {
    this.userId = userId;
    this.date = date;
    this.time = time;
    this.rate = rate;
    this.id = id;
  }
}
```

2.3.4 SleepRecord 模型

SleepRecord 模型		
字段	数据类型	描述
id	number	记录 id
userid	number	用户 id
date	string	日期（YYYY-MM-DD）
duration	number	睡眠时长

```
// 睡眠记录模型
export class SleepRecord {
  id?: number;
  userId: number;
  date: string;    // 格式：YYYY-MM-DD
  duration: number; // 睡眠时长（分钟）
  constructor(userId: number, date: string, duration: number, id?: number) {
    this.userId = userId;
    this.date = date;
    this.duration = duration;
    this.id = id;
  }
  // 将分钟转换为小时和分钟的格式
  getDurationText(): string {
    const hours = Math.floor(this.duration / 60);
    const minutes = this.duration % 60;
    return `${hours}小时${minutes}分钟`;
  }
}
```

2.3.5 WaterRecord 模型

WaterRecord 模型		
字段	数据类型	描述
id	number	记录 id
userid	number	用户 id
date	string	日期（YYYY-MM-DD）
time	string	时间（HH-MM）
amount	number	饮水量（ml）

```
// 饮水记录模型
export class WaterRecord {
  id?: number;
  userId: number;
  date: string;   // 格式：YYYY-MM-DD
  time: string;   // 格式：HH:MM
  amount: number; // 饮水量（毫升）
  constructor(userId: number, date: string, time: string, amount: number, id?: number) {
```

```
    this.userId = userId;
    this.date = date;
    this.time = time;
    this.amount = amount;
    this.id = id;
  }
}
```

2.4 数据库操作关键代码：

2.4.1 用户偏好设置辅助类 PreferencesHelper 功能：

便捷登录体验：用户可以选择"记住我"功能，避免每次打开应用都需要重新输入凭据。

自动填充信息：当应用启动或进入登录页面时，可以自动填充上次保存的用户名和密码，方便下次登陆不用再输入密码，

实现数据持久化：使用 HarmonyOS 的 preferences API 确保数据在应用重启后仍然可用。

实现代码：

```
// 保存用户登录信息
async saveUserInfo(username: string, password: string, rememberMe: boolean): Promise<void> {
  if (rememberMe) {
    await this.preferences?.put(KEY_USERNAME, username);
    await this.preferences?.put(KEY_PASSWORD, password);
  } else {
    await this.preferences?.delete(KEY_USERNAME);
    await this.preferences?.delete(KEY_PASSWORD);
  }
  await this.preferences?.put(KEY_REMEMBER_ME, rememberMe);
  await this.preferences?.flush();
}
// 获取保存的用户信息
async getUserInfo(): Promise<UserInfo> {
  const username = await this.preferences?.get(KEY_USERNAME, '');
  const password = await this.preferences?.get(KEY_PASSWORD, '');
  const rememberMe = await this.preferences?.get(KEY_REMEMBER_ME, false);
  return { username, password, rememberMe } as UserInfo;
}
```

2.4.2 数据库操作辅助类 DatabaseHelper：

数据库初始化：initDatabase() 建立数据库连接并创建必要的表结构。

```
// 数据库配置
const STORE_CONFIG: StoreConfig = {
  name: 'HealthDB.db',           // 数据库名称
  securityLevel: relationalStore.SecurityLevel.S1  // 安全级别
};
```

```
// 初始化数据库方法
public async initDatabase(): Promise<boolean> {
  try {
    this.rdbStore = await relationalStore.getRdbStore(this.context, STORE_CONFIG);

    // 检查并升级步数记录表结构
    await this.upgradeStepRecordTable();

    // 创建各种数据表
    await this.rdbStore.executeSql(SQL_CREATE_USER_TABLE, []);
    await this.rdbStore.executeSql(SQL_CREATE_STEP_TABLE, []);
    await this.rdbStore.executeSql(SQL_CREATE_HEART_RATE_TABLE, []);
    await this.rdbStore.executeSql(SQL_CREATE_SLEEP_TABLE, []);
    await this.rdbStore.executeSql(SQL_CREATE_WATER_TABLE, []);

    // 插入测试用户数据
    await this.insertTestUser();
    return true;
  } catch (error) {
    // 错误处理
    return false;
  }
}
```

用户管理：

insertUser(): 注册新用户，代码如下所示。

validateUser(): 验证用户登录，代码如下所示。

updateUserInfo(): 更新用户资料，代码如下所示。

getUserById(): 获取用户详细信息，由于详细代码过长故此处省略。

```
// 插入新用户
public async insertUser(user: User): Promise<number> {
  const valuesBucket: relationalStore.ValuesBucket = {
    'username': user.username,
    'password': user.password
  };
  const rowId = await this.rdbStore.insert('user', valuesBucket);
  return rowId;
}
// 验证用户登录
public async validateUser(username: string, password: string): Promise<User | null> {
  const users = await this.queryUserByUsername(username);
  if (users.length === 0) return null;
  const user = users[0];
  return user.password === password ? user : null;
```

```
}
// 更新用户信息
public async updateUserInfo(user: User): Promise<boolean> {
  // 构建更新数据
  const valuesBucket: relationalStore.ValuesBucket = {};
  if (user.age !== undefined) valuesBucket.age = user.age;
  // ...其他字段类似处理

  const predicates = new relationalStore.RdbPredicates('user');
  predicates.equalTo('id', user.id);
  const rowCount = await this.rdbStore.update(valuesBucket, predicates);
  return rowCount > 0;
}
```

健康数据管理：

步数记录：

insertStepRecord()：向数据库插入一条新的步数记录，若成功返回 true。

getStepRecords()：根据用户 Id 查询用户的步数记录并返回 StepRecord 数组，包含查询到的所有步数记录。用于对指定日期的数据查询。

updateStepRecord()：更新现有的步数记录，若成功返回 true。

batchSaveStepRecords()：批量插入步数记录，允许一次性插入多个数据。

心率记录：

insertHeartRateRecord()：向数据库插入一条新的心率记录，若成功返回 true。

getHeartRateRecords()：根据用户 Id 查询用户的心率记录并返回 HeartRateRecord 数组，包含查询到的所有步数记录。用于对指定日期的数据查询。

updateHeartRateRecord()：更新现有的心率记录，若成功返回 true。

睡眠记录：

insertSleepRecord()：向数据库插入一条新的睡眠记录，若成功返回 true。用于单条记录的插入，不包括具体时间，只有日期和时长。

getSleepRecords()：查询用户的睡眠记录，用于查询指定日期的睡眠时长。

updateSleepRecord()：通过用户 Id 更新当日睡眠时长。

饮水记录：

insertWaterRecord()：向数据库插入一条新的饮水记录，包含用户 ID、日期、时间和饮水量。插入成功返回 true。

getWaterRecords()：根据用户 Id 查询用户的饮水记录，返回

WaterRecord 数组，包含查询到的所有饮水记录，用于查询指定日期的记录，也可以查询所有日期的记录。

updateWaterRecord()：更新现有的饮水记录。

此处只给出步数数据的 CURD 操作代码示例，其余几个健康数据的操作基本类似。

```
// 批量保存步数记录
async batchSaveStepRecords(userId: number, date: string, timeStepsArray: StepTimeInput[]): Promise<boolean> {
  // 开始事务
  await this.rdbStore.beginTransaction();

  for (const timeStep of timeStepsArray) {
    const valuesBucket = {
      'user_id': userId,
      'date': date,
      'time': timeStep.time,
      'steps': timeStep.steps
    };

    const rowId = await this.rdbStore.insert('step_record', valuesBucket);
    if (rowId < 0) {
      // 回滚事务
      await this.rdbStore.rollBack();
      return false;
    }
  }

  // 提交事务
  await this.rdbStore.commit();
  return true;
}
// 获取特定用户的步数记录
public async getStepRecords(userId: number, date?: string): Promise<StepRecord[]> {
  const predicates = new relationalStore.RdbPredicates('step_record');
  predicates.equalTo('user_id', userId);
  if (date) {
    predicates.equalTo('date', date);
  }
  predicates.orderByAsc('time'); // 按时间顺序排序

  const resultSet = await this.rdbStore.query(predicates, ['id', 'user_id', 'date', 'time', 'steps']);
  // 处理查询结果...
}
```

```
// 类似地还有心率、睡眠、饮水等数据的CRUD操作
```

数据更新检查：

checkDatabaseUpdates()：检查数据库是否有新的更新，用来检查用户的健康数据（步数、心率、睡眠和饮水记录）自上次检查以来是否有任何更新。用于决定是否需要刷新 UI 显示以及触发系统通知。通过 hasRecentUpdates 函数辅助实现。

```
public async checkDatabaseUpdates(userId: number, lastCheckTime: number): Promise<boolean> {
  // 首先检查数据库连接是否有效
  if (!this.rdbStore) {
    return false;
  }
  try {
    // 分别检查四种健康数据类型是否有更新
    const stepsUpdated = await this.hasRecentUpdates('step_record', userId, lastCheckTime);
    const heartRateUpdated = await this.hasRecentUpdates('heart_rate_record', userId, lastChec
kTime);
    const sleepUpdated = await this.hasRecentUpdates('sleep_record', userId, lastCheckTime);
    const waterUpdated = await this.hasRecentUpdates('water_record', userId, lastCheckTime);
    // 只要有任何一种数据更新，就返回true
    return stepsUpdated || heartRateUpdated || sleepUpdated || waterUpdated;
  } catch (error) {
    // 记录错误日志并返回false
    hilog.error(0x0000, TAG, `Failed to check for database updates: ${error}`);
    return false;
  }
}

private async hasRecentUpdates(tableName: string, userId: number, lastCheckTime: number): Prom
ise<boolean> {
  // 检查数据库连接是否有效
  if (!this.rdbStore) {
    return false;
  }
  try {
    // 获取当天的日期字符串，格式为"YYYY-MM-DD"
    const today = new Date().toISOString().split('T')[0];
    // 创建查询条件：指定用户ID和当天日期
    const predicates = new relationalStore.RdbPredicates(tableName);
    predicates.equalTo('user_id', userId);
    predicates.equalTo('date', today);
    // 执行查询，只需获取ID字段即可
    const resultSet = await this.rdbStore.query(predicates, ['id']);
    // 检查是否有记录（行数大于0）
    const hasUpdates = resultSet.rowCount > 0;
```

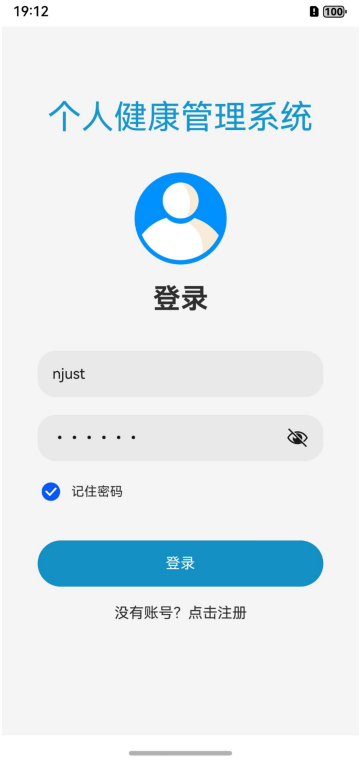
```
    // 关闭结果集，释放资源
    resultSet.close();
    return hasUpdates;
  } catch (error) {
    // 记录错误日志并返回false
    hilog.error(0x0000, TAG, `Failed to check for updates in ${tableName}: ${error}`);
    return false;
  }
}
```

第三章 模块功能分析

3.1 登录与注册模块

该模块由自定义组件 Login.ets 和 register.ets 组成，在 register 组件中能够进行新用户的注册功能，只需要输入用户名以及密码即可实现注册功能。在 Login 组件中只需要输出已注册的用户名以及密码即可登录，并且给出可选选项“记住密码”，选择后在下次进入程序时重新登录即可不用再重读输入密码，该功能通过 PreferenceHelper.ets 帮助实现，PreferenceHelper 主要负责处理用户偏好设置（preferences）的存储和获取，能够实现用户登录信息的持久化和数据管理。

程序截图：



关键实现代码：

```
async register() { //注册方法
    // 表单验证 ，确保所填字段非空
    if (!this.username || !this.password || !this.confirmPassword) {
      promptAction.showToast({    // Toast 提示框
        message: '请填写所有字段',
        duration: 2000
      });
      return;
    }
    if (this.password !== this.confirmPassword) { //确保两次密码一致 ，否则弹出提示
      promptAction.showToast({
        message: '两次输入的密码不一致',
```

```
        duration: 2000
      });
      return;
    }
    this.isLoading = true;  //两次密码一致 ， 将正在登录标记设置为true
    try {
      // 从数据库中检查用户名是否已存在
      const existingUsers = await this.dbHelper.queryUserByUsername(this.username);
      if (existingUsers.length > 0) {  // 用户存在时进行提示
        promptAction.showToast({
          message: '用户名已存在',
          duration: 2000    //持续两秒
        });
        this.isLoading = false;  // 正在登录标志设为false，此时无法注册
        return;
      }
      // 创建新用户
      const newUser = new User(this.username, this.password);
     // 尝试插入到数据库中，返回 true 即为插入成功
      const success = await this.dbHelper.insertUser(newUser);
      if (success) {
         promptAction.showToast({ //弹出提醒
          message: '注册成功', // 提醒信息
          duration: 2000 // 持续时间
        });
        // 返回登录页面
        router.back();
      } else {
        promptAction.showToast({
          message: '注册失败',
          duration: 2000
        });
      }
    } catch (error) {
      hilog.error(0x0000, TAG, `Register error: ${error}`);
      promptAction.showToast({
        message: '注册过程出错',
        duration: 2000
      });
    } finally {
      this.isLoading = false;
    }
  }
```

实现登录逻辑：

```
  async login() {  // 登录逻辑实现
    if (!this.username || !this.password) {  // 检查用户名和密码是否正确输入，否则弹出提示
      promptAction.showToast({
        message: '用户名和密码不能为空',
        duration: 2000
      });
      return;
    }
    this.isLoading = true;
    try {
      // 延迟 2 秒钟模拟后台认证过程，并且给出加载动画
      await new Promise<void>(resolve => setTimeout(resolve, 2000));
      const isValid = await this.dbHelper.validateUser(this.username, this.password); //检查用
户信息是否正确
      if (isValid) {
        await this.preferencesHelper.saveUserInfo(this.username, this.password, this.rememberM
e);
        promptAction.showToast({
          message: '登录成功',
          duration: 2000
        });
        this.onJumpClick();
      } else {
        promptAction.showToast({
          message: '用户名或密码错误',
          duration: 2000
        });
      }
    } catch (error) {
      hilog.error(0x0000, TAG, `Login error: ${error}`);
      promptAction.showToast({
        message: '登录过程出错',
        duration: 2000
      });
    } finally {
      this.isLoading = false;
    }
  }
```

saveUserInfo()函数：保存用户信息，如果"记住我"为 true 则保存用户名和密码，否则删除这些信息。使用 Preference 来实现免输入密码登录。

```
async saveUserInfo(username: string, password: string, rememberMe: boolean): Promise<void> {
  // 检查preferences对象是否已初始化，如果没有则初始化
  if (!this.preferences) {
    await this.initPreferences();
  }

  try {
    if (rememberMe) {
      // 当"记住我"选项为true时，保存用户名和密码到首选项存储
      await this.preferences?.put(KEY_USERNAME, username);
      await this.preferences?.put(KEY_PASSWORD, password);
    } else {
      // 当"记住我"选项为false时，从首选项存储中删除用户名和密码
      await this.preferences?.delete(KEY_USERNAME);
      await this.preferences?.delete(KEY_PASSWORD);
    }

    // 无论如何都保存"记住我"的设置状态
    await this.preferences?.put(KEY_REMEMBER_ME, rememberMe);
    // 调用flush()将内存中的更改持久化到存储中
    await this.preferences?.flush();
  } catch (error) {
    // 捕获并记录保存过程中可能发生的任何错误
    console.error(`Failed to save user info: ${error}`);
  }
}
```

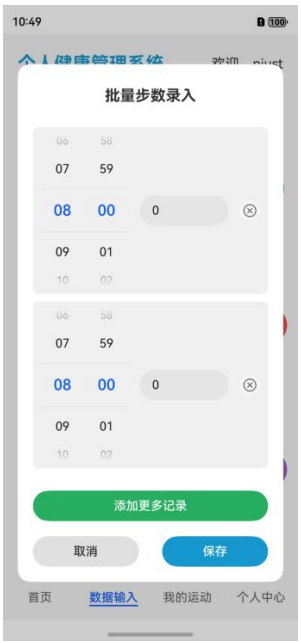
3.2 健康数据输入模块

该模块在 DataInputTab.ets 中实现，能够实现用户数据的输入，包括单个或批量的步数数据，心率数据，饮水量数据以及当日的睡眠时长数据的数据。

在该页面中，分别设置了三个弹出框用于批量输入数据，分别是批量输入心率（HeartRateInputDialog），批量输入饮水量（WaterInputDialog），批量输入步数数据（StepInputDialog)用于使用弹出框进行批量输入数据，并且采用了时间选择器 Timpicker 来进行输入数据时时间的选择，避免繁琐的手动输入数据，提高了交互性。

数据输入界面展示：

批量数据输入示例，这里仅显示批量输入步数，批量输入心率与饮水量逻辑一致。



保存数据的关键代码展示：

输入心率数据方法 saveMultiHeartRates():支持批量与单词输入心率数据,步骤为：

(1). 验证用户 ID 是否有效

(2).遍历心率输入数组，对每条记录:验证时间格式和心率值是否有效并创建心率记录对象并插入数据库。

(3).统计成功保存的记录数，并显示相应提示

(4).更新界面显示的数据

(5). 实现异常处理：与心率函数类似，捕获错误并提供反馈。

```
async saveMultiHeartRates() {
  // 验证用户 ID 是否有效
  if (this.userId <= 0) {
    promptAction.showToast({ message: '用户 ID 无效' });
    return;
  }
  try {
    let successCount = 0;  // 记录成功保存的条目数
```

```
    // 遍历所有输入行，处理每组时间和心率值
    for (let i = 0; i < this.heartRateInputTimes.length; i++) {
      const time = this.heartRateInputTimes[i].trim();  // 去除时间字符串前后的空格
      const rate = parseInt(this.heartRateInputValues[i]) || 0;  // 将心率转换为整数，如果转换失败则为0
      // 跳过无效的记录（空时间、无效时间格式或心率小于等于0）
      if (!time || !this.isValidTimeFormat(time) || rate <= 0) {
        continue;
      }
      // 创建心率记录对象
      const record = new HeartRateRecord(this.userId, this.currentDate, time, rate);

      // 尝试将记录插入数据库，如果成功则递增计数器
      if (await this.dbHelper.insertHeartRateRecord(record)) {
        successCount++;
      }
    }
    // 根据保存结果提供反馈
    if (successCount > 0) {
      promptAction.showToast({ message: `成功保存 ${successCount} 条心率记录` });
      this.refreshData();  // 刷新界面数据
    } else {
      promptAction.showToast({ message: '没有有效的心率记录被保存' });
    }
  } catch (error) {
    // 错误处理：提取错误信息并记录日志
    const errorMessage: string = error instanceof Error ? error.message : String(error);
    hilog.error(0x0000, TAG, `Failed to save heart rate records: ${errorMessage}`);
    promptAction.showToast({ message: '保存心率记录失败' });
  }
}
```

输入饮水量数据方法 saveMultiWaterRecords()：**支持批量与单词输入饮水量数据，步骤为：**

(1). 验证用户 ID 是否有效

(2). 遍历饮水输入数组，对每条记录：验证时间格式和饮水量是否有效并创建饮水记录对象并插入数据库。

(3). 统计成功保存的记录数，并显示相应提示

(4). 检查饮水量是否达标，可能发送通知

(5). 更新界面显示的数据

(6). 实现异常处理：与心率函数类似，捕获错误并提供反馈。

```
async saveMultiWaterRecords() {
  // 验证用户ID是否有效
```

```
  if (this.userId <= 0) {
    promptAction.showToast({ message: '用户 ID 无效' });
    return;
  }

  try {
    let successCount = 0;  // 记录成功保存的条目数

    // 遍历所有输入行，处理每组时间和饮水量
    for (let i = 0; i < this.waterInputTimes.length; i++) {
      const time = this.waterInputTimes[i].trim();  // 去除时间字符串前后的空格
      const amount = parseInt(this.waterInputValues[i]) || 0;  // 将饮水量转换为整数，如果转换失败则为0

      // 跳过无效的记录（空时间、无效时间格式或饮水量小于等于0）
      if (!time || !this.isValidTimeFormat(time) || amount <= 0) {
        continue;
      }

      // 创建饮水记录对象
      const record = new WaterRecord(this.userId, this.currentDate, time, amount);

      // 尝试将记录插入数据库，如果成功则递增计数器
      if (await this.dbHelper.insertWaterRecord(record)) {
        successCount++;
      }
    }
    // 根据保存结果提供反馈
    if (successCount > 0) {
      promptAction.showToast({ message: `成功保存 ${successCount} 条饮水记录` });
      this.checkWaterAndNotify(this.waterAmount);  // 检查饮水量并可能发送通知
      this.refreshData();  // 刷新界面数据
    } else {
      promptAction.showToast({ message: '没有有效的饮水记录被保存' });
    }
  } catch (error) {
    // 错误处理：提取错误信息并记录日志
    const errorMessage: string = error instanceof Error ? error.message : String(error);
    hilog.error(0x0000, TAG, `Failed to save water records: ${errorMessage}`);
    promptAction.showToast({ message: '保存饮水记录失败' });
  }
}
```

输入步数数据并保存 saveMultiStepRecords():**支持批量与单词输入步数数据，步骤为：**

(1). 验证用户 ID 是否有效

(2). 遍历步数输入数组，对每条记录：验证时间格式和步数值是否有效并将有效记录收集到一个数组中。

(3). 使用批量保存方法一次性将所有记录保存到数据库

(4). 计算总步数并显示成功保存的记录数

(5). 检查步数是否达标，可能发送通知

(6). 更新界面显示的数据

(7). 实现异常处理：与心率函数类似，捕获错误并提供反馈。

```
async saveMultiStepRecords(): Promise<void> {
  // 验证用户 ID 是否有效
  if (this.userId <= 0) {
    hilog.error(0x0000, TAG, `Invalid userId: ${this.userId}`);
    promptAction.showToast({ message: '用户 ID 无效' });
    return;
  }
  try {
    hilog.info(0x0000, TAG, `Starting to save multi step records for user ${this.userId}`);
    let successCount = 0;  // 记录成功验证的条目数
    // 收集有效的时间步数记录
    const validRecords: StepTimeInput[] = [];
    // 记录数据验证过程
    hilog.info(0x0000, TAG, `Total input rows: ${this.stepInputTimes.length}`);
    // 遍历所有输入行，处理每组时间和步数
    for (let i = 0; i < this.stepInputTimes.length; i++) {
      const time = this.stepInputTimes[i].trim();  // 去除时间字符串前后的空格
      const stepsStr = this.stepInputValues[i];
      const steps = parseInt(stepsStr) || 0;  // 将步数转换为整数，如果转换失败则为0
      hilog.info(0x0000, TAG, `Validating row ${i}: time=${time}, steps=${steps}`);
      // 验证时间是否为空
      if (!time) {
        hilog.warn(0x0000, TAG, `Row ${i}: Empty time`);
        continue;
      }
      // 验证时间格式
      if (!this.isValidTimeFormat(time)) {
        hilog.warn(0x0000, TAG, `Row ${i}: Invalid time format: ${time}`);
        continue;
      }
      // 验证步数是否有效
      if (steps <= 0) {
        hilog.warn(0x0000, TAG, `Row ${i}: Invalid steps value: ${steps}`);
        continue;
```

```
      }
      // 创建记录对象并添加到验证通过的记录数组
      const record: StepTimeInput = {
        time: time,
        steps: steps
      };
      validRecords.push(record);
      successCount++;
      hilog.info(0x0000, TAG, `Row ${i}: Valid record added: time=${time}, steps=${steps}`);
    }
    hilog.info(0x0000, TAG, `Collected ${successCount} valid records out of ${this.stepInputTimes.length} inputs`);
    // 处理验证结果
    if (successCount > 0) {
      // 使用批量保存方法将所有有效记录一次性保存到数据库
      hilog.info(0x0000, TAG, `Calling batchSaveStepRecords with ${validRecords.length} records`);
      const result = await this.dbHelper.batchSaveStepRecords(
        this.userId,
        this.currentDate,
        validRecords
      );
      if (result) {
        // 计算总步数
        const totalSteps = validRecords.reduce((sum, record) => sum + record.steps, 0);
        hilog.info(0x0000, TAG, `Successfully saved ${successCount} records with total steps: ${totalSteps}`);
        // 向用户显示保存成功的消息
        promptAction.showToast({
          message: `成功保存 ${successCount} 条步数记录，总步数: ${totalSteps}`
        });
        // 检查步数达成情况并可能发送通知，然后刷新界面数据
        hilog.info(0x0000, TAG, 'Refreshing data after saving');
        this.checkStepsAndNotify(totalSteps);
        await this.refreshData();
      } else {
        // 数据库操作失败
        hilog.error(0x0000, TAG, 'batchSaveStepRecords returned false');
        promptAction.showToast({ message: '保存步数记录失败' });
      }
    } else {
      // 没有有效的记录
      hilog.warn(0x0000, TAG, 'No valid records to save');
      promptAction.showToast({ message: '没有有效的步数记录被保存' });
```

```
    }
  } catch (error) {
    // 详细的错误处理：提取错误信息和堆栈跟踪并记录日志
    const errorMessage: string = error instanceof Error ? error.message : String(error);
    hilog.error(0x0000, TAG, `Exception in saveMultiStepRecords: ${errorMessage}`);
    hilog.error(0x0000, TAG, `Stack trace: ${error instanceof Error ? error.stack : 'No stack trace'}`);
    promptAction.showToast({ message: '保存步数记录失败: ' + errorMessage });
  }
}
```

3.3 健康数据分析展示模块

该模块通过三个页面实现，分别为首页展示健康数据概览(HomeTab.ets)，详细信息页面（dataShow.ets）显示用户的具体的心率与饮水量数据，运动步数展示页面（MyMovementTab.ets）。

在首页（HomeTab.ets）的健康数据概览中，页面上半部份为显示出今日的健康数据概览，分别为今日总步数，今最近的心率记录，今日的睡眠时长以及总饮水量，然后给出两个按键：“分享健康数据”（该功能暂未实现，这里显示是为了页面的美观）和“查看详细数据”（点击即可跳转至详细分析页面）。并且下半区域详细展示今日的饮水记录与心率记录，

如图所示：

该页面中使用@Extend 修饰器对 Text 和 Column 等组件实现了扩展组件样式；并且使用了@Link 装饰器实现了与父组件的双向数据绑定，使该页面能够获取到数据库中用户的数据。使用 Want 实现组件间通信，将数据从父组件传递到子组件中展示。

并且使用了多个自定义组件来展示不同数据的类型：

HealthDataCard：展示健康数据卡片

```
@Component
export struct HealthDataCard {
  @Prop title: string;       // 卡片标题，如"步数"、"心率"等
  @Prop value: string;       // 数值，如"8000"、"75"等
  @Prop unit: string = '';   // 单位，如"bpm"、"ml"等，默认为空
  @Prop color: string = '#1698CE'; // 数值文本颜色，默认为蓝色
  build() {
    Column() {
      // 显示卡片标题
      Text(this.title)
        .labelText()  // 使用自定义的 labelText 样式

      // 显示数值和单位（如果有单位则添加空格和单位）
      Text(this.value + (this.unit ? ' ' + this.unit : ''))
        .valueText()  // 使用自定义的 valueText 样式
        .fontColor(this.color)  // 应用传入的颜色
    }
    .width('45%')  // 卡片宽度占父容器的 45%
    .padding(10)   // 内边距 10 像素
    .borderRadius(8)  // 圆角 8 像素
    .backgroundColor('#f7f7f7')  // 背景色浅灰
  }
}
```

HeartRateRecordItem：展示单条心率记录

```
export struct HeartRateRecordItem {
  @Prop record!: HeartRateRecord;  // 心率记录数据，使用!表示非空断言
  build() {
    Row() {
      // 显示记录时间
      Text(this.record.time)
        .fontSize(16)
        .width('40%')  // 时间占 40% 宽度

      // 显示心率值和单位
      Text(`${this.record.rate} 次/分`)
        .fontSize(16)
        .fontColor('#3498DB')  // 蓝色文本
        .width('60%')  // 数值占 60% 宽度
    }
    .width('100%')  // 行宽度占满父容器
    .padding(10)    // 内边距 10 像素
```

```
    .justifyContent(FlexAlign.SpaceBetween)  // 两端对齐布局
  }
}
```

WaterRecordItem：展示单条饮水记录

```
@Component
export struct WaterRecordItem {
  @Prop record!: WaterRecord;  // 饮水记录数据，使用!表示非空断言

  build() {
    Row() {
      // 显示记录时间
      Text(this.record.time)
        .fontSize(16)
        .width('40%')  // 时间占40%宽度

      // 显示饮水量和单位
      Text(`${this.record.amount} ml`)
        .fontSize(16)
        .fontColor('#3498DB')  // 蓝色文本
        .width('60%')  // 数值占60%宽度
    }
    .width('100%')  // 行宽度占满父容器
    .padding(10)    // 内边距10像素
    .justifyContent(FlexAlign.SpaceBetween)  // 两端对齐布局
  }
}
```

关键实现代码：

shareHealthData 方法：实现将父组件的健康数据传递到子组件 datashow 中进行健康数据的分析。

```
async shareHealthData() {
  try {
    // 首先刷新数据，确保分享的是最新数据
    await this.refreshData();
    // 创建健康数据对象，包含用户所有相关健康信息
    const healthData: HealthData = {
      userId: this.userId,          // 用户ID
      username: this.username,      // 用户名
      date: this.currentDate,       // 当前日期
      steps: this.steps,            // 步数
      heartRate: this.heartRate,    // 心率
      sleepHours: this.sleepHours,  // 睡眠时长
```

```
      waterAmount: this.waterAmount // 饮水量
    };
    // 将健康数据对象转换为JSON字符串，便于传输
    const dataString = JSON.stringify(healthData);

    // 创建Want对象，用于启动目标应用
    // Want是鸿蒙OS中用于描述应用组件启动意图的对象
    const want: Want = {
      bundleName: 'com.example.target',  // 目标应用的包名
      abilityName: 'EntryAbility',       // 目标应用的Ability名称
      moduleName: 'entry',               // 目标应用的模块名称
      parameters: {
        healthData: dataString           // 将健康数据作为参数传递
      }
    };

    // 调用context的startAbility方法启动目标应用
    this.context.startAbility(want)
      .then(() => {
        // 启动成功，记录成功日志
        hilog.info(0x0000, TAG, 'Health data shared successfully');
      })
      .catch((err: Error) => {
        // 启动失败，记录错误日志
        hilog.error(0x0000, TAG, `Failed to start ability: ${err.message}`);
      });
  } catch (error) {
    // 捕获并处理整个分享过程中可能发生的任何错误
    // 将错误对象转换为字符串消息
    const errorMessage: string = error instanceof Error ? error.message : String(error);
    // 记录错误日志
    hilog.error(0x0000, TAG, `Failed to share health data: ${errorMessage}`);
  }
}
```

查看详细数据页面（dataShow.ets）：

该页面实现的功能有，心率部分：展示用户最近一次测量的心率数值，并使用 Canvas 绘制折线图展示一天内的心率变化趋势，仅对当日数据有效进行绘图，并给出心率统计数据：计算并显示静息心率、平均心率、最高心率以及最低心率。并在概览下方给出心率记录列表：以时间线形式展示所有心率测量记录。

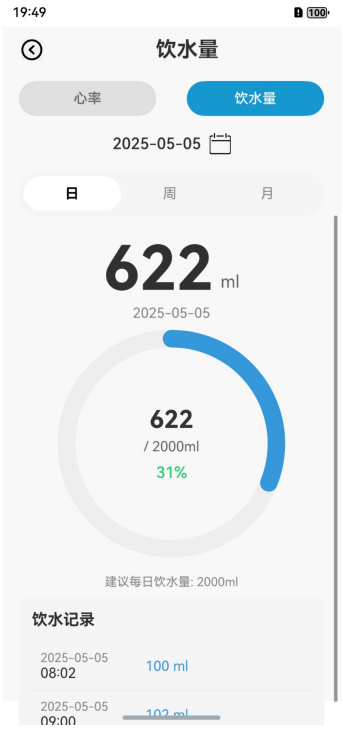
饮水量部分：首先显示总饮水量，并且显示特定时间段内（本日，本周或本月）的累计饮水量，并且使用环形进度条直观展示饮水完成比例，实现了饮水数据的可视化效果，并在下方给出饮水记录列表，记录每次饮水的时间和摄入量，页面底部给出饮水建议，提供科学

的饮水建议，更体现人性化。

此外，该页面还提供了“日”，“周”，“月”三种时间维度下的数据查看方式，并且能通过日历选择器自由选择想要查看日期的健康数据，并且能够在心率数据和饮水量数据之间灵活切换，支持点击左上角返回图标返回首页。

在心率数据显示截图：

饮水量数据显示截图：



三种时间维度截图：

关键代码实现：

加载周数据函数实现，加载月数据函数与此类似，这里不再给出。

```
// 加载周数据
async loadWeekData() {
  if (!this.userId) {
    return;
  }
  try {
    const userId = parseInt(this.userId);
    this.heartRateRecords = [];
    this.waterRecords = [];
    let totalHeartRate = 0;
    let totalHeartRateCount = 0;
    let allHeartRates: number[] = [];
    let totalWaterAmount = 0;
    this.totalSteps = 0;
    let totalSleepMinutes = 0;
    let sleepDaysCount = 0;
    // 遍历本周每一天加载数据
    for (const date of this.weekDates) {
      // 加载心率数据
      const heartRateRecords = await this.dbHelper.getHeartRateRecords(userId, date);
```

```
        this.heartRateRecords = [...this.heartRateRecords, ...heartRateRecords];
        if (heartRateRecords.length > 0) {
          const rates = heartRateRecords.map(record => record.rate);
          totalHeartRate += rates.reduce((sum, rate) => sum + rate, 0);
          totalHeartRateCount += rates.length;
          allHeartRates = [...allHeartRates, ...rates];
        }
        // 加载饮水数据
        const waterRecords = await this.dbHelper.getWaterRecords(userId, date);
        this.waterRecords = [...this.waterRecords, ...waterRecords];
        if (waterRecords.length > 0) {
          const dayWaterAmount = waterRecords.reduce((sum, record) => sum + record.amount, 0);
          totalWaterAmount += dayWaterAmount;
        }
        // 加载步数数据
        const stepRecords = await this.dbHelper.getStepRecords(userId, date);
        if (stepRecords.length > 0) {
          this.totalSteps += stepRecords[0].steps;
        }
        // 加载睡眠数据
        const sleepRecords = await this.dbHelper.getSleepRecords(userId, date);
        if (sleepRecords.length > 0) {
          totalSleepMinutes += sleepRecords[0].duration;
          sleepDaysCount++;
        }
      }
      // 计算心率统计信息
      if (allHeartRates.length > 0) {
        this.highestHeartRate = Math.max(...allHeartRates);
        this.lowestHeartRate = Math.min(...allHeartRates);
        this.averageHeartRate = Math.round(totalHeartRate / totalHeartRateCount);
        this.restingHeartRate = this.lowestHeartRate + 5;
        if (this.restingHeartRate > this.averageHeartRate) {
          this.restingHeartRate = this.lowestHeartRate;
        }
        const latestRecord = this.heartRateRecords[this.heartRateRecords.length - 1];
        this.heartRate = latestRecord.rate.toString();
        this.lastHeartRateTime = `${latestRecord.date} ${latestRecord.time}`;
      } else {
        this.heartRate = '0';
        this.restingHeartRate = 0;
        this.highestHeartRate = 0;
        this.lowestHeartRate = 0;
        this.averageHeartRate = 0;
```

```
        this.lastHeartRateTime = '';
      }
      // 设置总饮水量
      this.waterAmount = totalWaterAmount.toString();
      // 计算平均睡眠时间（小时）
      this.averageSleepHours = sleepDaysCount > 0 ? (totalSleepMinutes / sleepDaysCount) / 60 :
0;
    } catch (error) {
      console.error(`加载周数据失败: ${error}`);
    }
  }
```

绘制心率趋势图实现方法 drawHeartRateChart()：

主要实现：

数据处理：

对心率记录按时间排序

计算心率值的范围，确定 Y 轴刻度

根据所选视图类型（日/周/月）准备不同的数据展示形式

图表绘制：

绘制 Y 轴主线、刻度和水平网格线

绘制 X 轴主线和时间标签

绘制心率折线图、数据点和心率值标签

自适应显示：

当没有数据时显示提示信息

限制 X 轴标签数量，避免拥挤

根据视图类型调整标签显示内容

```
drawHeartRateChart() {
  // 清空整个画布，准备重新绘制
  this.context.clearRect(0, 0, this.canvasWidth, this.canvasHeight);
  // 检查是否有心率数据，如果没有则显示提示信息
  if (this.heartRateRecords.length === 0) {
    this.context.fillStyle = '#888888';  // 设置文字颜色为灰色
    this.context.textAlign = 'center';   // 文字居中对齐
    this.context.font = '32px sans-serif'; // 设置字体大小和类型
    this.context.fillText('暂无心率数据', this.canvasWidth / 2, this.canvasHeight / 2); // 在画
布中央显示提示文字
    return; // 没有数据则提前退出函数
  }
  // 对心率记录按时间排序，使用数组解构创建副本避免修改原数组
  const sortedRecords = [...this.heartRateRecords].sort((a, b) => {
    return a.time.localeCompare(b.time); // 使用字符串比较方法按时间升序排序
  });
```

```
// 计算心率的范围，用于确定Y轴刻度
const rates = sortedRecords.map(record => record.rate); // 提取所有心率值
const minRate = Math.max(Math.min(...rates) - 10, 0);  // 最小心率值减10，但不小于0
const maxRate = Math.max(...rates) + 10;               // 最大心率值加10，提供上下边距
const rateRange = maxRate - minRate;                   // 计算心率范围区间大小
// 定义图表的绘制区域，考虑了边距和轴的宽度
const chartX = this.chartPadding + this.yAxisWidth;    // 图表左边界位置
const chartY = this.chartPadding;                      // 图表上边界位置
const chartWidth = this.canvasWidth - this.yAxisWidth - this.chartPadding * 2;  // 图表宽度
const chartHeight = this.canvasHeight - this.xAxisHeight - this.chartPadding * 2; // 图表高度
// 准备数据点和X轴标签的数组
let dataPoints: HeartRateRecord[] | AggregatedDataPoint[] = [];
let xLabels: string[] = [];
// 根据选择的视图类型(日/周/月)准备相应的数据
if (this.selectedTab === 0) {
  // 日视图：直接使用排序后的记录
  dataPoints = sortedRecords;
  xLabels = sortedRecords.map(record => record.time); // X轴标签为时间
} else if (this.selectedTab === 1) {
  // 周视图：处理聚合的周数据
  const weekData = this.processAggregatedData(this.weekDates);
  dataPoints = weekData.dataPoints; // 获取处理后的数据点
  xLabels = weekData.labels;       // 获取处理后的标签
} else {
  // 月视图：处理聚合的月数据
  const monthData = this.processAggregatedData(this.monthDates);
  dataPoints = monthData.dataPoints;
  xLabels = monthData.labels;
}
// 绘制Y轴主线
this.context.strokeStyle = '#CCCCCC'; // 设置轴线颜色为浅灰色
this.context.lineWidth = 1;           // 设置线宽为1像素
this.context.beginPath();             // 开始一个新的路径
this.context.moveTo(chartX, chartY); // 移动到Y轴起点
this.context.lineTo(chartX, chartY + chartHeight); // 画线到Y轴终点
this.context.stroke();                // 绘制路径
// 绘制Y轴刻度标签和水平网格线
this.context.fillStyle = '#666666';  // 设置标签文字颜色为深灰色
this.context.textAlign = 'right';    // 文字右对齐
this.context.font = '32px sans-serif'; // 设置字体
const yLabelCount = 5;               // Y轴标签数量
for (let i = 0; i <= yLabelCount; i++) {
  // 计算当前刻度对应的心率值
  const yValue = minRate + (rateRange * (yLabelCount - i) / yLabelCount);
```

```
    // 计算当前刻度在Y轴上的位置
    const yPos = chartY + (i * chartHeight / yLabelCount);
    // 绘制Y轴刻度文字标签
    this.context.fillText(Math.round(yValue).toString(), chartX - 5, yPos + 4);
    // 绘制对应的水平网格线
    this.context.strokeStyle = '#EEEEEE'; // 设置网格线颜色为浅灰色
    this.context.beginPath();
    this.context.moveTo(chartX, yPos);    // 从Y轴刻度位置开始
    this.context.lineTo(chartX + chartWidth, yPos); // 画到图表右边界
    this.context.stroke();
  }
  // 绘制X轴主线
  this.context.strokeStyle = '#CCCCCC';
  this.context.beginPath();
  this.context.moveTo(chartX, chartY + chartHeight); // 从X轴起点开始
  this.context.lineTo(chartX + chartWidth, chartY + chartHeight); // 画到X轴终点
  this.context.stroke();
  // 绘制X轴标签
  this.context.textAlign = 'center';   // 文字居中对齐
  // 限制X轴标签数量，避免标签过多导致拥挤
  const maxLabels = Math.min(6, xLabels.length); // 最多显示6个标签或所有标签数(如果少于6个)
  const labelStep = Math.ceil(xLabels.length / maxLabels); // 计算标签间隔步长
  // 遍历并绘制X轴标签和垂直网格线
  for (let i = 0; i < xLabels.length; i += labelStep) {
    if (i < xLabels.length) {
      const label = xLabels[i];
      // 根据视图类型调整标签显示内容
      const displayLabel = this.selectedTab === 0
        ? label.substring(0, 5)     // 日视图只显示前5个字符
        : label.substring(5);       // 周/月视图显示从第5个字符开始的内容
      // 计算标签在X轴上的位置
      const xPos = chartX + (i / (xLabels.length - 1)) * chartWidth;
      // 绘制X轴标签文字
      this.context.fillText(displayLabel, xPos, chartY + chartHeight + 20);
      // 绘制垂直网格线
      this.context.strokeStyle = '#EEEEEE';
      this.context.beginPath();
      this.context.moveTo(xPos, chartY);              // 从图表上边界开始
      this.context.lineTo(xPos, chartY + chartHeight); // 画到图表下边界
      this.context.stroke();
    }
  }
  // 绘制心率折线图，至少需要两个点才能绘制
  if (dataPoints.length > 1) {
```

```
// 设置折线样式
this.context.strokeStyle = '#E74C3C'; // 设置折线颜色为红色
this.context.lineWidth = 2;          // 设置线宽为2 像素
this.context.beginPath();
// 绘制连接所有数据点的折线
for (let i = 0; i < dataPoints.length; i++) {
  const point = dataPoints[i];
  // 根据视图类型获取不同的心率值
  const rate = this.selectedTab === 0
    ? (point as HeartRateRecord).rate          // 日视图：直接使用记录的心率
    : (point as AggregatedDataPoint).averageRate; // 周/月视图：使用平均心率
  // 计算数据点在图表中的坐标
  const xPos = chartX + (i / (dataPoints.length - 1)) * chartWidth;
  // Y 坐标计算：将心率值映射到图表高度范围内
  const yPos = chartY + chartHeight - ((rate - minRate) / rateRange) * chartHeight;
  if (i === 0) {
    this.context.moveTo(xPos, yPos); // 第一个点移动到位置
  } else {
    this.context.lineTo(xPos, yPos); // 后续点连线到位置
  }
}
this.context.stroke(); // 绘制整条折线
// 绘制每个数据点的圆点和数值标签
this.context.fillStyle = '#FFFFFF';   // 设置圆点填充颜色为白色
this.context.strokeStyle = '#E74C3C'; // 设置圆点边框颜色为红色
this.context.lineWidth = 2;          // 设置边框宽度
this.context.textAlign = 'center';   // 文字居中对齐
// 遍历所有数据点
for (let i = 0; i < dataPoints.length; i++) {
  const point = dataPoints[i];
  // 根据视图类型获取心率值
  const rate = this.selectedTab === 0
    ? (point as HeartRateRecord).rate
    : (point as AggregatedDataPoint).averageRate;
  // 计算数据点在图表中的坐标
  const xPos = chartX + (i / (dataPoints.length - 1)) * chartWidth;
  const yPos = chartY + chartHeight - ((rate - minRate) / rateRange) * chartHeight;
  // 绘制数据点的圆圈
  this.context.beginPath();
  this.context.arc(xPos, yPos, 4, 0, Math.PI * 2); // 绘制半径为4 的圆
  this.context.fill();   // 填充圆内部
  this.context.stroke(); // 绘制圆边框
  // 绘制心率数值标签
  this.context.fillStyle = '#E74C3C';          // 设置文字颜色为红色
```

```
      this.context.font = 'bold 32px sans-serif';  // 设置粗体字
      this.context.fillText(rate.toString(), xPos, yPos - 15); // 在数据点上方显示心率值
      this.context.fillStyle = '#FFFFFF'; // 恢复填充颜色为白色，为下一个点做准备
    }
  }
}
```

我的运动显示页面（MyMovement.ets）：

在该页面将对步数数据进行详细分析，该页面主要功能为：对用户数据实现可视化，提供日、周、月三种时间维度的步数数据图表，使用使用 Chartt 绘制柱状图来展示不同时间段的步数情况，支持横向滚动查看更多历史数据；并在柱状图下方显示总步数统计，并计算总行走距离(公里)，展示日均步数和日均距离。

使用 List 列出当日详细的步数记录时间段，使用可滚动列表展示多条记录；最后显示用户的体重(kg)和身高(cm)数据，并且计算并显示 BMI 值。

整个页面采用了@Component、@Extend 等装饰器来构建界面组件。整体 UI 采用嵌套的 Column 和 Row 布局，并使用 Scroll 组件使内容可滚动。

页面截图：

绘制柱状图关键实现函数（UI 实现请查看程序代码）：

```
async updateStepChartData(forceRefresh: boolean = false) {
  try {
    let data: ChartDataItem[] = [];       // 初始化图表数据数组
    let total = 0;                        // 初始化总步数
    let dateDisplay = '';                 // 初始化日期显示文本
    const currentDateObj = new Date(this.currentDate); // 创建当前日期对象
    // 根据不同的图表模式处理数据
    switch (this.chartMode) {
      case '日':
```

```
        // 日视图：显示一天内不同时间段的步数分布
        total = this.totalSteps; // 使用已计算好的今日总步数
        // 定义一天中的6个时间段
        const timeSlots = ['00:00', '04:00', '08:00', '12:00', '16:00', '20:00'];
        if (this.stepTimeRecords.length > 0) {
          // 如果有详细的时间步数记录，按时间段汇总
          const slotData: number[] = new Array(timeSlots.length).fill(0); // 初始化每个时间段的步数为0
          // 遍历所有步数记录，将其归类到对应的时间段
          for (const record of this.stepTimeRecords) {
            const recordHour = parseInt(record.time.split(':')[0]); // 获取记录的小时
            let slotIndex = 0; // 初始化时间段索引
            // 确定记录属于哪个时间段
            for (let i = 1; i < timeSlots.length; i++) {
              const slotHour = parseInt(timeSlots[i].split(':')[0]); // 获取时间段的小时
              if (recordHour < slotHour) {
                // 找到第一个大于记录小时的时间段，则记录属于前一个时间段
                break;
              }
              slotIndex = i; // 更新时间段索引
            }
            slotData[slotIndex] += record.steps; // 累加该时间段的步数
          }
          // 创建图表数据项
          for (let i = 0; i < timeSlots.length; i++) {
            data.push({
              label: timeSlots[i],    // 时间段标签
              value: slotData[i]      // 该时间段的步数
            });
          }
        } else {
          // 如果没有详细时间记录，使用默认的时间分布比例
          // 定义各时间段占总步数的百分比
          const hourlyDistribution = [0.05, 0.05, 0.15, 0.30, 0.35, 0.10];
          // 根据分布比例计算每个时间段的步数
          for (let i = 0; i < timeSlots.length; i++) {
            const slotSteps = Math.round(total * hourlyDistribution[i]); // 根据分布比例计算步数
            data.push({
              label: timeSlots[i],
              value: slotSteps
            });
          }
        }
        // 设置Y轴最大显示值，取15000和总步数40%中的较大值
```

```
        this.maxDisplayValue = Math.max(15000, total * 0.4);
        dateDisplay = this.currentDate; // 设置日期显示文本为当前日期
        this.avgDailySteps = total;    // 日视图的平均步数即为总步数
        break;
      case '周':
        // 周视图：显示一周内每天的步数

        const today = new Date(this.currentDate); // 获取当前日期
        const currentDayOfWeek = today.getDay() || 7; // 获取星期几(0-6)，将周日的0转换为7
        const mondayOfWeek = new Date(today); // 创建本周一的日期
        mondayOfWeek.setDate(today.getDate() - currentDayOfWeek + 1); // 设置为本周的周一
        let weekTotal = 0; // 初始化周总步数
        let dayCount = 0;  // 初始化有步数记录的天数
        // 记录日志，表示正在更新周数据
        hilog.info(DOMAIN_NUMBER, TAG, `更新周视图数据，强制刷新: ${forceRefresh}`);
        // 清空数据数组，确保完全刷新
        data = [];
        // 遍历周一到周日的7天
        for (let i = 0; i < 7; i++) {
          // 创建当天的日期对象
          const date = new Date(mondayOfWeek);
          date.setDate(mondayOfWeek.getDate() + i);
          // 格式化日期为YYYY-MM-DD格式
          const dateStr = date.toISOString().split('T')[0];
          // 记录日志，表示正在获取特定日期的步数数据
          hilog.info(DOMAIN_NUMBER, TAG, `获取日期 ${dateStr} 的步数数据`);
          // 从数据库获取该日期的步数记录，确保获取最新数据
          const stepRecords = await this.dbHelper.getStepRecords(this.userId, dateStr);
          // 计算该日的总步数（累加所有记录的步数）
          const daySteps = stepRecords.reduce((sum, record) => sum + record.steps, 0);
          // 记录日志，显示计算出的总步数
          hilog.info(DOMAIN_NUMBER, TAG, `${dateStr} 的总步数: ${daySteps}`);
          // 添加图表数据项，标签为"月/日"格式
          data.push({
            label: `${date.getMonth() + 1}/${date.getDate()}`, // 月/日格式的标签
            value: daySteps                                    // 该日的总步数
          });
          weekTotal += daySteps; // 累加到周总步数
          if (daySteps > 0) {
            dayCount++;          // 如果当天有步数，有效天数+1
          }
        }
        total = weekTotal; // 设置总步数为周总步数
        // 设置Y轴最大显示值，取15000和(周平均步数*1.5)中的较大值
```

```
        this.maxDisplayValue = Math.max(15000, Math.ceil(weekTotal / 7) * 1.5);
        // 计算本周结束日期（周日）
        const endOfWeek = new Date(mondayOfWeek);
        endOfWeek.setDate(mondayOfWeek.getDate() + 6);

        // 设置日期范围显示文本，格式为"YYYY 年MM 月DD 日至DD 日"
        dateDisplay = `${mondayOfWeek.getFullYear()}年${mondayOfWeek.getMonth() +
          1}月${mondayOfWeek.getDate()}日至${endOfWeek.getDate()}日`;

        // 计算平均每日步数（只计算有步数记录的天数）
        this.avgDailySteps = dayCount > 0 ? Math.round(weekTotal / dayCount) : 0;
        // 记录日志，表示周数据更新完成
        hilog.info(DOMAIN_NUMBER, TAG, `周视图数据更新完成: 总步数=${total}, 平均步数
=${this.avgDailySteps}`);
        break;
      case '月':
        // 月视图: 显示一个月内每天的步数
        const year = currentDateObj.getFullYear();  // 获取年份
        const month = currentDateObj.getMonth();    // 获取月份（0-11）
        // 计算当月的天数（创建下月第0 天，即当月最后一天）
        const daysInMonth = new Date(year, month + 1, 0).getDate();
        let monthTotal = 0; // 初始化月总步数
        let activeDays = 0; // 初始化有步数记录的天数
        // 记录日志，表示正在更新月数据
        hilog.info(DOMAIN_NUMBER, TAG, `更新月视图数据, 强制刷新: ${forceRefresh}`);
        // 清空数据数组，确保完全刷新
        data = [];
        // 遍历当月的每一天
        for (let i = 1; i <= daysInMonth; i++) {
          // 格式化月份和日期，确保两位数（如01,02...）
          const monthStr = (month + 1).toString().padStart(2, '0');
          const dayStr = i.toString().padStart(2, '0');
          // 组合成YYYY-MM-DD 格式的日期字符串
          const dateStr = `${year}-${monthStr}-${dayStr}`;
          // 记录日志，表示正在获取特定日期的步数数据
          hilog.info(DOMAIN_NUMBER, TAG, `获取日期 ${dateStr} 的步数数据`);
          // 从数据库获取该日期的步数记录
          const stepRecords = await this.dbHelper.getStepRecords(this.userId, dateStr);
          // 计算该日的总步数（累加所有记录的步数）
          const daySteps = stepRecords.reduce((sum, record) => sum + record.steps, 0);
          // 记录日志，显示计算出的总步数
          hilog.info(DOMAIN_NUMBER, TAG, `${dateStr} 的总步数: ${daySteps}`);
          // 添加图表数据项，标签为日期数字
          data.push({
```

```
            label: `${i}`,   // 日期数字作为标签
            value: daySteps  // 该日的总步数
          });
          monthTotal += daySteps; // 累加到月总步数
          if (daySteps > 0) {
            activeDays++;        // 如果当天有步数，有效天数+1
          }
        }
        total = monthTotal; // 设置总步数为月总步数
        // 设置Y轴最大显示值，取15000和(月平均步数*1.5)中的较大值
        this.maxDisplayValue = Math.max(15000, Math.ceil(monthTotal / daysInMonth) * 1.5);
        // 设置日期显示文本，格式为"YYYY年MM月"
        dateDisplay = `${year}年${month + 1}月`;
        // 计算平均每日步数（只计算有步数记录的天数）
        this.avgDailySteps = activeDays > 0 ? Math.round(monthTotal / activeDays) : 0;
        // 记录日志，表示月数据更新完成
        hilog.info(DOMAIN_NUMBER, TAG, `月视图数据更新完成：总步数=${total}，平均步数
=${this.avgDailySteps}`);
        break;
    }
    // 更新图表数据、总步数和日期范围文本
    this.stepChartData = data;
    this.totalSteps = total;
    this.dateRangeText = dateDisplay;

  } catch (error) {
    // 捕获并处理任何可能发生的错误
    // 提取错误信息，针对Error对象和其他类型错误做不同处理
    const errorMessage: string = error instanceof Error ? error.message : String(error);
    // 记录错误日志
    hilog.error(0x0000, TAG, `Failed to update step chart data: ${errorMessage}`);
  }
}
```

3.4 用户信息与修改模块

该模块共分为两个页面，分别为个人中心页面（ProfileTab.ets）与用户信息修改页面（changeMyInfo.ets）：

个人中心页面（ProfileTab.ets）：

该页面实现首先展示个人信息，包括显示用户头像、用户名和用户 ID 等，并提供编辑个人资料的入口按钮，点击即可跳转到修改信息页面，并提供了多个功能按钮（由于时间原因暂未实现）以及退出登录按钮，点击即可退出登录。

页面采用垂直布局（Column），使用@Builder 修饰器进行封装。主要分为两大部分：顶部的个人信息卡片以及底部的功能菜单列表，页面截图：

关键代码实现：

```
@Builder
UserAvatar() {
  Column() {
    // 用户头像
    if (this.userAvatarPath) {
      // 如果有自定义头像，显示自定义头像
      Image(this.userAvatarPath)
        .width(80)
        .height(80)
        .borderRadius(40)
    } else {
      // 否则显示默认头像
      Image($r('app.media.avater_1'))
        .width(80)
        .height(80)
        .borderRadius(40)
    }
  }
  .margin({ right: 20 })
}
@Builder
UserInfo() {
  Column() {
    Row() {
      Text('用户名：')
        .fontSize(12)
        .fontWeight(FontWeight.Bold)
        .width('30%')
      Text(this.username)
        .fontSize(16)
```

```
          .width('70%')
      }
      .width('100%')
      .padding({ top: 5, bottom: 5 })
      Row() {
        Text('用户 ID: ')
          .fontSize(12)
          .fontWeight(FontWeight.Bold)
          .width('30%')
        Text(this.userId.toString())
          .fontSize(16)
          .width('70%')
      }
      .width('100%')
      .padding({ top: 5, bottom: 5 })
    }
    .layoutWeight(1)
    .alignItems(HorizontalAlign.Start)
  }
  @Builder
  ProfileCard() {
    Column() {
      Row() {
        this.UserAvatar()
        this.UserInfo()
      }
      .width('100%')
      .padding(10)
      Button('编辑个人资料')
        .primaryButton()
        .onClick(() => {
          hilog.info(DOMAIN_NUMBER, TAG,
            `Navigating to myInfo with username: ${this.username}, userId: ${this.userId}`);
          router.pushUrl({
            url: 'pages/changeMyInfo',
            params: {
              username: this.username,
              userId: this.userId,
              isEdit: true
            }
          }, router.RouterMode.Standard, (err) => {
            if (err) {
              console.error(`跳转到个人信息页面失败: ${err.code}, ${err.message}`);
              promptAction.showToast({ message: '跳转失败，请重试' });
```

```
            return;
          }
          console.info('跳转到个人信息页面成功');
        });
      })
  }
  .cardContainer()
  .margin({ bottom: 20 })
}
@Builder
MenuItem(icon: Resource, title: string, onClick: () => void) {
  Row() {
    Image(icon)
      .width(30)
      .height(30)
      .margin({ right: 10 })
    Text(title)
      .fontSize(16)
      .layoutWeight(1)
    Image(icon)
      .width(20)
      .height(20)
  }
  .menuItem()
  .onClick(onClick)
}
@Builder
MenuSection() {
  Column() {
    // 健康数据分析菜单项
    this.MenuItem($r('app.media.data'), '健康数据分析', () => {
      this.ClickJumpToShowData();
    })
    // 健康目标设置菜单项
    this.MenuItem($r('app.media.target'), '健康目标设置', () => {
      promptAction.showToast({
        message: '健康目标设置功能即将上线',
        duration: 2000
      });
    })
    // 健康提醒菜单项
    this.MenuItem($r('app.media.remind'), '健康提醒', () => {
      promptAction.showToast({
        message: '健康提醒功能即将上线',
```

```
          duration: 2000
        });
      })
      // 设置菜单项
      this.MenuItem($r('app.media.settings'), '设置', () => {
        promptAction.showToast({
          message: '设置功能即将上线',
          duration: 2000
        });
      })
      // 退出登录菜单项
      this.MenuItem($r('app.media.logout'), '退出登录', () => {
        router.clear();
        router.pushUrl({
          url: 'pages/Login'
        }, router.RouterMode.Standard, (err) => {
          if (err) {
            console.error(`退出登录失败 code: ${err.code}, message: ${err.message}`);
            promptAction.showToast({ message: '退出登录失败，请重试' });
            return;
          }
          console.info('退出登录成功');
        });
      })
    }
    .width('90%')
    .margin({ bottom: 20 })
  }
```

用户信息修改页面（changeMyInfo.ets）：

该页面首先实现的是从数据库加载用户信息并展示，并在用户信息展示的基础上允许用户修改原有数据并储存到数据库中。首先显示了用户的基本信息，包括用户名、年龄、性别等，联系方式信息：电话、邮箱等，显示身体信息：包括身高、体重，并实时计算出 BMI 指数。并支持自定义头像和默认头像，点击头像即可从系统中选择图片进行更换。在信息显示的基础上允许用户修改个人信息（除用户名外）与支持更换头像功能，并且提供了输入的数据验证（年龄、身高、体重必须为数字）。

页面截图：

关键代码实现：

选择头像进行修改：

```
async selectAvatar() {
    try {
      // 创建照片选择器实例
      const photoPicker = new picker.PhotoViewPicker();
      // 创建照片选择选项
      const photoSelectOptions = new picker.PhotoSelectOptions();
      // 设置 MIME 类型为图片
      photoSelectOptions.MIMEType = picker.PhotoViewMIMETypes.IMAGE_TYPE;
      // 设置最大选择数量为 1
      photoSelectOptions.maxSelectNumber = 1;
      // 调用选择方法，等待用户选择图片
      const photoSelectResult = await photoPicker.select(photoSelectOptions);
      // 如果有选择结果且选择了照片
      if (photoSelectResult && photoSelectResult.photoUris && photoSelectResult.photoUris.leng
th > 0) {
        // 获取选择的照片 URI
        const photoUri = photoSelectResult.photoUris[0];
        // 设置头像保存目录
        const appDir = this.context.filesDir + '/avatars/';
        try {
          // 尝试获取目录状态，检查目录是否存在
          const stat = await fs.stat(appDir);
        } catch (err) {
          // 如果目录不存在，则创建目录
          if (err.code === 'ENOENT') {
            await fs.mkdir(appDir);
          } else {
```

```
            // 如果是其他错误，则暂不处理
          }
        }
        // 生成头像文件名，使用用户 ID 和时间戳确保唯一性
        const avatarFileName = `avatar_${this.userId}_${Date.now()}.jpg`;
        // 组合完整的头像文件路径
        const avatarFilePath = appDir + avatarFileName;
        // 复制选择的照片到应用目录
        await fs.copyFile(photoUri, avatarFilePath);
        // 更新头像路径状态变量
        this.avatarPath = avatarFilePath;
        // 记录日志，表示头像选择成功
        hilog.info(0x0000, TAG, `Avatar selected and copied to: ${this.avatarPath}`);
      }
    } catch (error) {
      // 捕获并处理异常，记录错误日志并显示提示
      const errorMessage: string = error instanceof Error ? error.message : String(error);
      hilog.error(0x0000, TAG, `Failed to select avatar: ${errorMessage}`);
      promptAction.showToast({ message: '选择头像失败', duration: 2000 });
    }
  }
```

保存个人信息方法：

```
  async saveUserInfo() {
    // 检查 userId 是否有效
    if (this.userId < 0) {
      promptAction.showToast({ message: '用户 ID 无效' });
      return;
    }
    try {
      // 获取当前用户信息，以保持原有的用户名和密码
      const currentUser = await this.dbHelper.getUserById(this.userId);
      // 如果无法获取当前用户信息，则显示提示并返回
      if (!currentUser) {
        promptAction.showToast({ message: '无法获取用户信息' });
        return;
      }
      // 验证年龄输入，必须是数字
      if (this.age.trim() && isNaN(parseInt(this.age))) {
        promptAction.showToast({ message: '年龄必须是数字' });
        return;
      }
      // 验证身高输入，必须是数字
      if (this.myheight.trim() && isNaN(parseFloat(this.myheight))) {
```

```
        promptAction.showToast({ message: '身高必须是数字' });
        return;
      }
      // 验证体重输入，必须是数字
      if (this.weight.trim() && isNaN(parseFloat(this.weight))) {
        promptAction.showToast({ message: '体重必须是数字' });
        return;
      }
      // 创建用户对象，用于更新用户信息
      const user = new User(
        currentUser.username,  // 保持原有用户名
        currentUser.password,  // 保持原有密码
        this.userId,           // 用户ID
        this.age.trim() ? parseInt(this.age) : undefined,  // 年龄，如果有值则转换为数字
        this.gender !== '未设置' ? this.gender : undefined,  // 性别，如果不是"未设置"则使用
        this.phone.trim() || undefined,  // 电话号码，如果为空则使用undefined
        this.email.trim() || undefined,  // 电子邮箱，如果为空则使用undefined
        this.myheight.trim() ? parseFloat(this.myheight) : undefined,  // 身高，如果有值则转换为
数字
        this.weight.trim() ? parseFloat(this.weight) : undefined,  // 体重，如果有值则转换为数字
        this.avatarPath || undefined  // 头像路径，如果为空则使用undefined
      );
      // 记录日志，显示尝试保存的用户信息
      hilog.info(0x0000, TAG, `Attempting to save user info: ${JSON.stringify({
        id: user.id,
        age: user.age,
        gender: user.gender,
        phone: user.phone,
        email: user.email,
        height: user.height,
        weight: user.weight,
        avatarPath: user.avatarPath
      })}`);
      // 调用数据库帮助类方法更新用户信息
      const success = await this.dbHelper.updateUserInfo(user);
      // 根据结果显示提示并处理后续操作
      if (success) {
        promptAction.showToast({ message: '保存成功' });
        // 保存成功后返回上一页
        router.back();
      } else {
        promptAction.showToast({ message: '保存失败' });
      }
    } catch (error) {
```

```
      // 捕获并处理异常，记录错误日志并显示提示
      const errorMessage: string = error instanceof Error ? error.message : String(error);
      hilog.error(0x0000, TAG, `Failed to save user info: ${errorMessage}`);
      promptAction.showToast({ message: '保存失败' });
    }
  }
```

3.5 系统通知模块

该模块主要实现三个功能，分别是每日步数达标时的系统通知，每日饮水量达标时的系统通知以及睡眠时长是否达标的通知。

checkWaterAndNotify 函数监测用户的饮水量并发送通知,主要功能：

(1).当用户饮水量达到或超过 2000ml 时，发送一条通知

(2).通知内容包括标题"健康饮水提醒"和恭喜用户达成饮水目标的信息

(3).系统会记录最后通知的饮水量，避免重复发送相同数值的通知

(4).在成功发送通知后，会显示一个 toast 提示

代码实现：

```
private  checkWaterAndNotify(waterAmount:number) : void {
   hilog.info(DOMAIN_NUMBER, TAG, `检查饮水总量通知条件: 当前饮水量=${waterAmount}, 上次饮水量
=${this.lastNotifyWater}`);
   if (waterAmount >= 2000 && waterAmount !== this.lastNotifyWater) {
     hilog.info(DOMAIN_NUMBER, TAG, `饮水量已达到通知条件: ${waterAmount}ml >= 2000ml`);
     try {
       let notificationRequest: notificationManager.NotificationRequest = {
         id: 2,
         content: {
           notificationContentType: notificationManager.ContentType.NOTIFICATION_CONTENT_BASIC
_TEXT,
           normal: {
             title: '健康饮水提醒',
             text: `恭喜！ 您今日的饮水量为${waterAmount}ml，已超过 2000ml 目标！ `,
             additionalText: '继续保持健康生活方式!'
           }
         },
         isOngoing: false
       };
       // 使用单一API 发送通知
       notificationManager.publish(notificationRequest, (err) => {
         if (err) {
           hilog.error(DOMAIN_NUMBER, TAG, `发送步数通知失败. 错误码: ${err.code}, 错误信
息: ${err.message}`);
           return;
         }
```

```
          hilog.info(DOMAIN_NUMBER, TAG, `成功发送步数通知：当前步数 ${waterAmount}`);
          // 更新最后通知的步数
          this.lastNotifyWater = waterAmount;
          // 提示用户检查通知栏
          promptAction.showToast({
            message: '已发送步数达标通知，请检查通知栏'
          });
        });
      } catch (error) {
        const errMsg = error instanceof Error ? error.message : String(error);
        hilog.error(DOMAIN_NUMBER, TAG, `步数通知处理异常: ${errMsg}`);
      }
    } else {
      if (waterAmount < 8000) {
        hilog.info(DOMAIN_NUMBER, TAG, `饮水量未达到通知条件: ${waterAmount}ml < 2000ml`);
      } else if (waterAmount === this.lastNotifyWater) {
        hilog.info(DOMAIN_NUMBER, TAG, `已经为当前饮水量(${waterAmount})发送过通知，不重复发送`);
      }
    }
  }
```

checkSleepAndNotify 函数监测用户的睡眠时长并发送通知：
当用户睡眠时长达到或超过 8 小时或小于 6 小时时，均发送一条通知，
通知内容包括标题"健康步数提醒"和恭喜用户达成目标的信息
系统会记录最后通知的数据，避免重复发送相同数值的通知
在成功发送通知后，会显示一个 toast 提示

实现代码：

```
private checkSleepAndNotify(sleepHours: number): void {
    // 记录日志：当前检查的睡眠时长和上次通知的睡眠时长
    hilog.info(DOMAIN_NUMBER, TAG, `检查睡眠时长通知条件: 当前睡眠时长=${sleepHours}, 上次睡眠时长
=${this.lastNotifySleep}`);
    // 条件1：睡眠时长达标（≥8 小时）且与上次通知的时长不同
    if (sleepHours >= 8.0 && sleepHours !== this.lastNotifySleep) {
      // 记录睡眠达标的日志
      hilog.info(DOMAIN_NUMBER, TAG, `睡眠时长已达标: ${sleepHours}小时 >= 8 小时`);
      try {
        // 创建睡眠达标通知请求对象
        let notificationRequest: notificationManager.NotificationRequest = {
          id: 3,  // 通知的唯一标识符
          content: {
            notificationContentType: notificationManager.ContentType.NOTIFICATION_CONTENT_BASI
C_TEXT,  // 通知内容类型为基本文本
            normal: {
```

```
            title: '健康睡眠提醒',  // 通知标题
            text: `恭喜！您今日的总睡眠时长为${sleepHours}小时，已超过 8 小时目标！`,  // 通知主要内容
            additionalText: '继续保持健康生活方式!'  // 通知附加内容
          }
        },
        isOngoing: false  // 非持续性通知（一次性通知）
      };
      // 使用通知管理器发布通知
      notificationManager.publish(notificationRequest, (err) => {
        // 发送通知失败的处理
        if (err) {
          hilog.error(DOMAIN_NUMBER, TAG, `发送睡眠通知失败. 错误码: ${err.code}, 错误信息: ${err.message}`);
          return;
        }
        // 发送通知成功的处理
        hilog.info(DOMAIN_NUMBER, TAG, `成功发送睡眠通知：当前睡眠时长 ${sleepHours}`);
        // 更新最后通知的睡眠时长，避免重复通知
        this.lastNotifySleep = sleepHours;
        // 显示 toast 提示，告知用户查看通知
        promptAction.showToast({
          message: '已发送睡眠达标通知，请检查通知栏'  // 这里有一个错误，应该是"睡眠达标通知"而不是"步数达标通知"
        });
      });
    } catch (error) {
      // 异常处理：将错误转换为字符串并记录日志
      const errMsg = error instanceof Error ? error.message : String(error);
      hilog.error(DOMAIN_NUMBER, TAG, `通知处理异常: ${errMsg}`);
    }
  }
  // 条件 2：睡眠时长不足（≤6 小时）且与上次通知的时长不同
  else if (sleepHours <= 6.0 && sleepHours !== this.lastNotifySleep) {
    try {
      // 创建睡眠不足警告通知请求对象
      let notificationRequest1: notificationManager.NotificationRequest = {
        id: 4,  // 使用不同的 ID 以区分不同类型的通知
        content: {
          notificationContentType: notificationManager.ContentType.NOTIFICATION_CONTENT_BASIC_TEXT,
          normal: {
            title: '健康睡眠提醒',  // 通知标题
            text: `注意！您今日的总睡眠时长为${sleepHours}小时，请注意休息时间！`,  // 警告内容
```

```
            additionalText: '请保持健康生活方式!'  // 建议内容
          }
        },
        isOngoing: false
      };
      // 使用通知管理器发布通知
      notificationManager.publish(notificationRequest1, (err) => {
        // 发送通知失败的处理
        if (err) {
          hilog.error(DOMAIN_NUMBER, TAG, `发送睡眠通知失败. 错误码: ${err.code}, 错误信息: ${err.message}`);  // 这里有一个错误，应该是"睡眠通知"而不是"步数通知"
          return;
        }
        // 发送通知成功的处理
        hilog.info(DOMAIN_NUMBER, TAG, `成功发送通知：当前睡眠时长 ${sleepHours}`);
        // 更新最后通知的睡眠时长，避免重复通知
        this.lastNotifySleep = sleepHours;
        // 显示toast提示，告知用户查看通知
        promptAction.showToast({
          message: '已发送步数达标通知，请检查通知栏'  // 这里有一个错误，应该是"睡眠不足通知"而不是"步数达标通知"
        });
      });
    } catch (error) {
      // 异常处理：将错误转换为字符串并记录日志
      const errMsg = error instanceof Error ? error.message : String(error);
      hilog.error(DOMAIN_NUMBER, TAG, `通知处理异常: ${errMsg}`);
    }
  }
  // 条件3：不满足发送通知的条件
  else {
    // 睡眠时长在6-8小时之间，记录日志
    if (sleepHours > 6.0 && sleepHours < 8.0) {
      hilog.info(DOMAIN_NUMBER, TAG, `时长未达到通知条件: ${sleepHours}小时 < 8小时`);
    }
    // 当前睡眠时长与上次通知的时长相同，避免重复通知
    else if (sleepHours === this.lastNotifySleep) {
      hilog.info(DOMAIN_NUMBER, TAG, `已经为当前时长(${sleepHours})发送过通知，不重复发送`);
    }
  }
}
```

checkStepsAndNotify 函数监测用户的步数并发送通知，主要功能与实现方法与上述方法类似，这里不再赘述。

第四章 个人总结