《移动应用开发》实验报告

专　　业：计算机科学与技术 2301 班

班　　级： 教学班 1206

学　　号：202313407028

姓　　名：张裕航

实验 1 熟悉鸿蒙应用开发环境

一、实验目的

1. 掌握 HarmonyOS 开发工具 DevEco Studio 的安装和环境配置。
2. 掌握 HarmonyOS 工程项目的创建。

二、实验内容和步骤

任务一：安装开发环境

目标：掌握开发环境的安装和设置方法，能够新建 helloworld 工程，自命名工程能在模拟器上正常运行。

实验步骤：

1. 下载安装 DevEco Studio

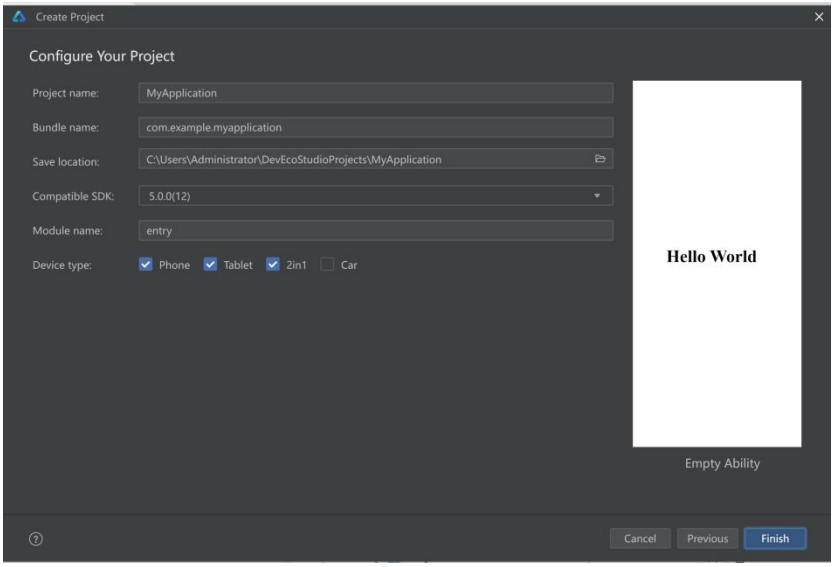
进入 HarmonyOS 官方开发者网站，下载与电脑系统对应的 DevEco Studio 安装包。

按默认方式安装，完成后启动 DevEco Studio。

2. 新建空工程 Helloworld

打开 DevEco Studio → Create Project → Choose “Application” 模板。

自定义工程名 MyApplication。



3. 配置并启动模拟器

点击右上角 Device Manager。

创建一个新的模拟器设备（Mate 70 pro）。

启动模拟器，等待加载完成。

4. 测试安装环境是否成功设置完成

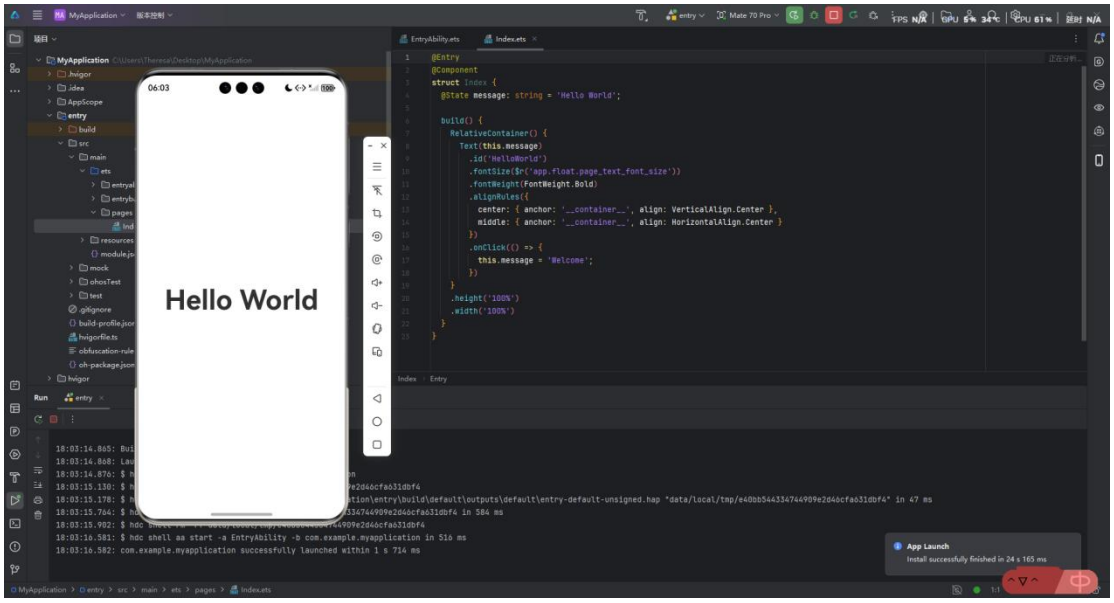
打开模拟器（手机界面显示开机完毕进入桌面，时间比较长）

等待同步 sync finished（状态栏显示 sync finished，编译构建完成，时间比较长）

run 运行程序（工具栏绿色三角）

等待 APP 部署到模拟器(观察状态栏和 run 窗口显示 deploy 过程)

在模拟器上看到 hello world 界面正确显示



任务二：熟悉 Deveco Studio,修改 helloworld 工程界面（加入日志调试信息，改变文字内容格式，加入按钮等新组件）

实验步骤：

1. 导入日志模块：

在 Index.ets 文件顶部引入 import hilog from '@ohos.hilog'; 以支持日志打印。

2. 修改布局容器：

将默认的 RelativeContainer 替换为 Column 容器，并设置 justifyContent(FlexAlign.Center) 使组件在屏幕垂直居中。

3. 定制文本组件：

将 Text 组件绑定的 @State 变量 message 初始值修改为 'Hello HarmonyOS'。

设置 .fontSize(50) 增大字号，.fontColor(Color.Blue) 将文字改为蓝色，并添加 .fontWeight(FontWeight.Bold) 加粗显示。

4. 添加按钮组件：

在 Text 组件下方添加 Button('点击我')。

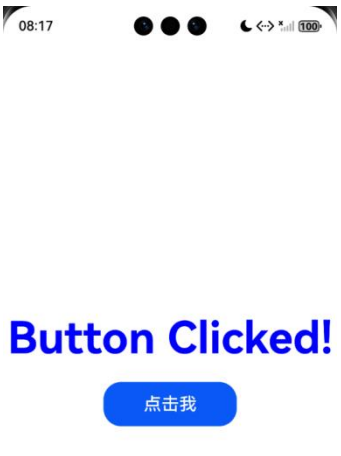
设置按钮宽高及字体大小。

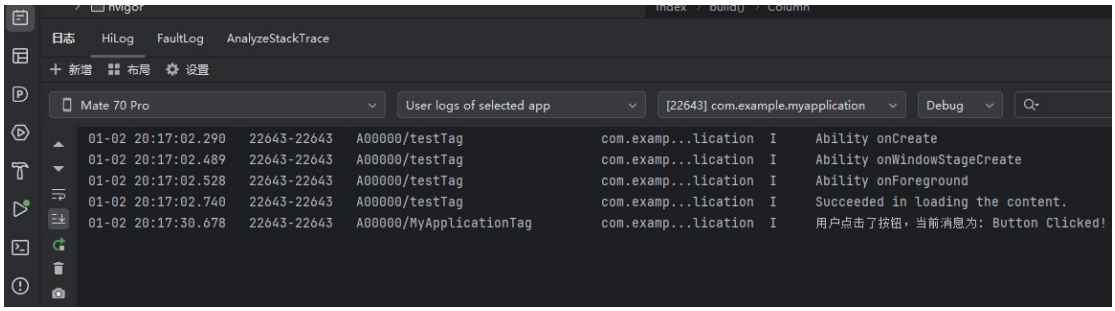
5. 实现交互逻辑：

在按钮的 .onClick 事件中，修改 this.message 的值为 'Button Clicked!'。

调用 hilog.info(0x0000, 'MyApplicationTag', ...) 输出调试日志。

6. 运行与调试：点击运行按钮启动应用，并打开底部的 Log 面板。





完整代码如下:

```
import hilog from '@ohos.hilog';

@Entry
@Component
struct Index {
  @State message: string = 'Hello HarmonyOS'; // 修改初始文字内容
  build() {
    // 使用 Column 替代 RelativeContainer 以便更容易垂直排列组件
    Column() {
      Text(this.message)
        .id('HelloWorld')
        .fontSize(50) // 修改字体大小
        .fontColor(Color.Blue) // 修改字体颜色
        .fontWeight(FontWeight.Bold)
        .margin({ bottom: 20 }) // 添加下边距
```

```
        Button('点击我') // 添加新组件：按钮
          .width(150)
          .height(50)
          .fontSize(20)
          .onClick(() => {
            // 1. 修改状态变量
            this.message = 'Button Clicked!';
            // 2. 加入日志调试信息
            hilog.info(0x0000, 'MyApplicationTag', '用户点击了按钮，当前消息为: %{public}s',
this.message);
          })
      }
      .height('100%')
      .width('100%')
      .justifyContent(FlexAlign.Center) // 居中对齐
    }
}
```

三、实验小结及思考

通过本次实验，我完成了 HarmonyOS 开发环境的搭建，成功安装并配置了 DevEco Studio，掌握了新建 HarmonyOS 应用工程的基本流程，并能够将 Helloworld 程序正确运行在模拟器上。

在熟悉 DevEco Studio 的过程中，我了解了工程目录结构及 ArkTS 页面文件的作用，能够对界面进行简单修改。

在实验中，通过将默认布局容器替换为 Column 容器，实现了界面组件的纵向排列，并对 Text 组件的字体大小、颜色和粗细进行了设置。同时，成功添加了 Button 按钮组件，并在按钮点击事件中修改页面状态变量，实现了界面内容的动态更新。此外，通过引入 hilog 模块并输出日志信息，我掌握了基本的日志调试方法，能够在 Log 面板中查看程序运行过程中的调试信息。

在上机实验过程中也遇到了一些问题：

1. 模拟器启动速度较慢，首次启动时等待时间较长，需要耐心等待系统完全进入桌面。

2. 项目首次构建时间较长，在出现 sync finished 前无法运行程序，这是由于首次下载依赖和构建工程所致。

3. 日志信息未及时显示，通过确认 Log 面板打开并选择正确的应用进程后，日志才能正常查看。

通过本次实验，我对 HarmonyOS 应用开发的整体流程有了初步认识，掌握了界面组件的基本使用方法和调试手段，为后续进行更复杂的鸿蒙应用开发和功能实现打下了良好的基础。

实验 2 界面设计

一、实验目的

1. 掌握 HarmonyOS 的常用布局。
2. 掌握界面基本组件的使用。
3. 掌握界面高级组件的使用。
4. 掌握 HarmonyOS 的交互机制。

二、实验内容和步骤

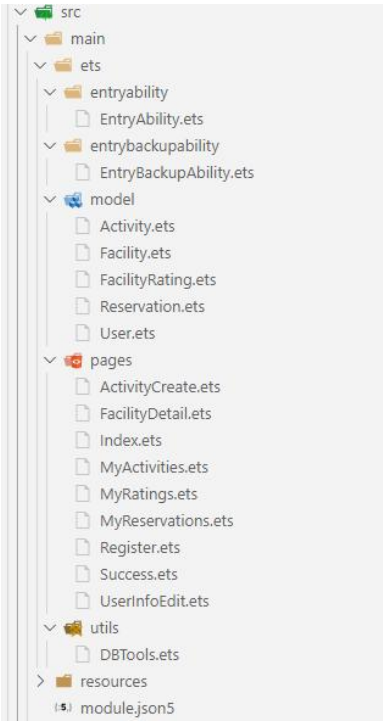
任务一：设计简单的多页面交互程序

目标：通过构造简单的交互程序，熟悉工程文件结构和基础编码方法

1. 工程结构分析与构建

本项目采用标准的 HarmonyOS Stage 模型结构，实现了模块化的代码组织，确保了代码的可维护性。

- 页面层 (pages/)：所有的 UI 界面文件均存放于此。
  - Index.ets：应用入口（登录页）。
  - Register.ets：用户注册页。
  - Success.ets：应用主页（包含仪表盘、列表等）。
  - FacilityDetail.ets：设施详情展示页。
  - UserInfoEdit.ets：个人信息编辑页。
- 模型层 (model/)：定义了 User, Facility, Reservation 等数据接口，规范了数据结构。
- 工具层 (utils/)：封装了 DBTools 类，通过调用其方法完成了数据的获取与展示流程。



2. 基础编码与页面骨架搭建

在 Index.ets 和 Register.ets 中，构建了标准的应用入口骨架。

- 入口逻辑：程序启动后首先加载 Index 页面。
- 交互逻辑：
  - 在登录页点击“立即注册”，程序通过路由机制跳转至 Register 页面。
  - 在登录页验证通过后，跳转至 Success 主页。
  - 在主页点击设施卡片，跳转至 FacilityDetail 详情页。 这种多页面的结构设计，模拟了真实商业应用的完整用户路径。

任务二：界面优化

目标：学习 ArkUI 简单界面设计。

1. 常用布局的应用 (Layout Implementation)

本项目广泛使用了 ArkUI 的核心布局容器来构建复杂的页面结构。

- 线性布局 (Column & Row)
  - 应用场景：几乎所有页面。例如在 UserInfoEdit.ets 中，使用 Column 垂直排列表单项，在每个表单项内部使用 Row 水平排列“标签”和“输入框”。
  - 优化技巧：使用了.layoutWeight(1)属性。在 UserInfoEdit.ets 的输入行中，让 TextInput 占据剩余空间，实现了自适应宽度的表单设计。
- 列表布局 (List & ListItem)
  - 应用场景：Success.ets 的推荐列表、MyReservations.ets 的预约记录列表。
  - 优化技巧：设置了.space(10)属性控制列表项间距，配合 ForEach 循环渲染，实现了高效的长列表滚动视图，避免了手动计算坐标的繁琐。
- 层叠布局 (Stack)
  - 应用场景：Success.ets 中的推荐卡片。
  - 优化技巧：

```
Stack({ alignContent: Alignment.TopEnd }) {
  Image(...) // 底层设施图片
  Text(item.facility_type) // 上层类型标签
}
```

利用 Stack 将“设施类型”标签悬浮在图片右上角，增强了视觉层次感。

2. 界面组件的深度使用与美化

- 基础组件美化 (TextInput & Button)
  - 去原生化：在 Index.ets 和 Register.ets 中，没有使用原生的输入框样式。而是将 TextInput 背景设为透明，外层包裹 Row 容器，并对 Row 应用了.backgroundColor(Color.White), .borderRadius(25) 和 .shadow()。
  - 效果：实现了具备悬浮感、圆角柔和的现代 UI 风格，提升了用户体验。
  - 功能区分：在 UserInfoEdit.ets 中，通过.type(InputType.PhoneNumber)优化键盘体验，通过.enabled(false)和灰色背景区分“只读”与“可编辑”状态。
- 高级组件应用 (Tabs & CustomDialog)
  - 底部导航 (Tabs)

在 Success.ets 中，使用 Tabs 组件构建了应用的主框架。通过@Builder 自定义了 TabBuilder 函数，实现了选中状态下图标和文字颜色的动态切换（选中为蓝色 #1698CE，未选中为灰色），替代了系统默认的单调样式。

  - 自定义弹窗 (CustomDialog)
    - 在 FacilityDetail.ets 中实现了 ReservationDialog（时间选择）。

- 在 MyReservations.ets 中实现了 RatingDialog（评分）。
- 将复杂的交互逻辑（如 DatePicker,Slider）封装在弹窗中，避免了主页面过于拥挤，保持了界面的整洁性。

3. 交互机制的实现

- 状态驱动视图 (@State)
  - 原理：ArkUI 采用声明式开发范式。
  - 实现：在 Success.ets 中，定义了@State currentIndex: number = 0。当用户点击底部 Tab 时，只需修改 currentIndex 的值，界面会自动刷新 Tab 的高亮状态和内容区域，无需手动操作 DOM 节点。
  - 表单绑定：在 UserInfoEdit.ets 中，TextInput 的 text 属性绑定了@State 变量，配合.onChange()事件，实现了数据与界面的实时同步。
- 条件渲染 (Conditional Rendering)
  - 场景：在 MyReservations.ets 中。
  - 实现：

```
if (this.isLoading) {
  LoadingProgress() // 加载中
} else if (this.reservations.length === 0) {
  Text('暂无预约记录') // 空状态
} else {
  List() { ... } // 数据列表
}
```

  - 意义：根据数据状态动态展示不同 UI，提升了应用的健壮性和用户体验。

三、实验小结及思考

通过本次界面设计实验，我对 HarmonyOS 应用的界面构建流程有了更加系统和深入的认识。

在实验过程中，我完成了多页面交互程序的设计与实现，熟悉了 Stage 模型下的工程结构，并能够合理地将页面层、模型层和工具层进行分离，从而提高了代码的可维护性和可读性。

在界面布局方面，我熟练运用了 ArkUI 中的常用布局容器，如 Column、Row、List 以及 Stack。通过组合不同布局容器，成功实现了表单布局、列表展示和卡片叠加等复杂界面效果。在实际操作中，我体会到合理使用布局容器可以有效避免传统坐标式布局带来的适配问题，使界面在不同屏幕尺寸下具有更好的自适应能力。

在组件使用方面，我不仅掌握了 Text、TextInput、Button 等基础组件的属性设置与样式美化方法，还进一步学习并应用了 Tabs、CustomDialog 等高级组件。通过自定义 TabBuilder 实现底部导航的高亮切换，以及将预约、评分等复杂交互封装到弹窗中，提升了界面的整体美观性与交互体验，使应用结构更加清晰。

在交互机制实现过程中，我深入理解了 ArkUI 的声明式开发思想。通过使用 @State 变量驱动界面变化，避免了手动刷新界面的复杂操作，实现了界面与数据的自动同步。同时，通过条件渲染的方式，根据数据状态动态显示不同的页面内容，提高了应用的健壮性。

在上机实验过程中也遇到了一些问题：

1. 多页面路由跳转初期不熟悉，在参数传递和页面跳转顺序上出现错误，通过查阅官方文档并反复调试后得以解决。

2. 复杂布局嵌套时样式冲突，部分组件显示异常，需要不断调整布局层级和属性设置。

3. 自定义弹窗逻辑较复杂，状态变量较多，容易出现数据未及时更新的问题，通过合理拆分状态并统一管理后问题得到解决。

通过本次实验，我不仅提升了 HarmonyOS 界面设计与交互实现的能力，也加深了对 ArkUI 声明式 UI 思想的理解，为后续进行更完整、更复杂的鸿蒙应用开发奠定了良好的实践基础。

实验 3 复杂界面设计

一、实验目的

1. 掌握应用内的页面跳转
2. 掌握应用内的数据传递
3. 掌握 Ability 的创建及其生命周期

二、实验内容和步骤

1. 应用内的页面跳转 (Page Navigation)

HarmonyOS 提供了 router 模块来管理页面栈。本项目主要使用了 pushUrl 进行跳转和 back 进行返回。

- 普通跳转： 在 Index.ets（登录页）中，当用户点击“立即注册”时，程序将注册页压入页面栈。

```
// Index.ets
import { router } from '@kit.ArkUI';
// ...
.onClick(() => {
  router.pushUrl({ url: 'pages/Register' })
})
```

- 带逻辑的跳转：在登录成功后，程序先执行业务逻辑，再跳转至主页 Success.ets。

```
// Index.ets
if (data > 0) {
  // 登录成功，跳转至主页
  router.pushUrl({ url: 'pages/Success' })
}
```

- 页面返回： 在 Register.ets 和 FacilityDetail.ets 等二级页面中，通过自定义的“返回”按钮调用 router.back() 返回上一页，模拟了导航栏行为。

```
// FacilityDetail.ets
Text('⬅ BACK')
  .onClick(() => {
    router.back(); // 弹出当前页面，返回栈顶页面
  })
```

2. 应用内的数据传递 (Data Transfer)

实验中使用了两种主要的数据传递方式：路由参数传递（用于页面间特定数据）和 AppStorage（用于全局会话数据）。

- 方式一：路由参数传递 (router.getParams) 在 Success.ets 跳转到 FacilityDetail.ets 时，需要告诉详情页展示哪个设施的信息。这里直接传递了 Facility 对象。
  - 发送方 (Success.ets)

```
router.pushUrl({
  url: 'pages/FacilityDetail',
  params: {
    facility: item // 传递选中的设施对象
  }
})
```

- 接收方 (FacilityDetail.ets)： 在页面加载生命周期中获取参数并类型转换。

```
aboutToAppear() {
  const params = router.getParams() as Record<string, Object>;
  if (params && params['facility']) {
    this.facility = params['facility'] as Facility;
    // 获取参数后，立即根据 ID 加载相关数据
    this.loadMyReservations();
  }
}
```

- 方式二：全局状态共享 (AppStorage) 用户登录后的 userId 是全局通用的凭证，不适合在每个页面跳转时都手动传递。因此使用了 AppStorage。
  - 存储 (Index.ets)：登录成功后写入

```
AppStorage.setOrCreate('currentUserId', data);
```

  - 读取 (UserInfoEdit.ets, MyReservations.ets 等)：在任何页面均可直接读取。

```
const userId = AppStorage.get<number>('currentUserId');
```

3. Ability 与页面生命周期 (Lifecycle Management)

本实验重点利用了页面生命周期接口来管理数据的加载时机，确保 UI 总是显示最新数据。

- aboutToAppear() - 组件初始化 这是组件创建后、build 执行前调用的接口。项目中所有的初始数据加载都放在这里。
  - 案例：在 Success.ets 中，页面一加载就请求设施列表和推荐数据。

```
// Success.ets
aboutToAppear() {
  this.loadFacilities();
  this.loadUserInfo();
  this.loadRecommendations();
}
```

- onPageShow() - 页面显示 当页面每次显示时触发（包括首次加载和从其他页面返回时）。这对于数据刷新至关重要。

- 案例：当用户在 UserInfoEdit.ets 修改个人信息并返回 Success.ets 时，Success.ets 不会重新创建，因此 aboutToAppear 不会再次执行。为了更新主页显示的用户名，必须使用 onPageShow。

```
// Success.ets
onPageShow() {
  // 每次页面可见时，重新加载用户信息，确保修改后能立即看到变化
  this.loadUserInfo();
  this.loadRecommendations();
}
```

- EntryAbility 的角色　虽然代码主要集中在 Page 层，但 EntryAbility 是整个应用的入口。它负责加载 pages/Index 作为 UI 的根节点，并管理应用级的生命周期（如应用切后台、销毁等）。

三、实验小结及思考

通过本次复杂界面设计实验，我系统地掌握了 HarmonyOS 应用中多页面跳转、页面间数据传递以及 Ability 与页面生命周期的基本使用方法。

在实验过程中，通过实际编码与调试，将多个页面有机地组织在一起，构建了较为完整的应用交互流程。

页面跳转方面，我熟练使用了 router 模块管理页面栈，能够通过 pushUrl 实现页面跳转，并在二级页面中使用 router.back() 返回上一页。通过在跳转前加入业务逻辑判断，使页面切换更加符合真实应用场景，提升了应用的交互合理性。

数据传递方面，对于设施详情等页面，通过路由参数将具体对象传递给目标页面，减少了冗余数据请求；对于用户登录信息等全局通用数据，则通过 AppStorage 进行统一管理，避免了在多个页面间重复传参的问题，使数据管理更加清晰和高效。

生命周期管理方面，本实验加深了我对 ArkUI 页面生命周期函数的理解。通过合理区分 aboutToAppear 与 onPageShow 的使用场景，确保页面在首次加载和返回显示时都能正确刷新数据，解决了页面不重新创建导致数据不同步的问题。同时，也认识到 EntryAbility 在整个应用生命周期管理中的重要作用。

上机实验过程中遇到了一些问题：

1. 页面参数类型不匹配，在接收路由参数时未及时进行类型转换，导致运行时报错，通过检查接口定义并进行类型约束后解决。

2. 生命周期函数使用不当，初期将数据刷新逻辑全部放在 aboutToAppear 中，导致返回页面时数据未更新，调整为在 onPageShow 中处理后问题得到解决。

3. AppStorage 状态更新不及时，在部分页面修改数据后未触发界面刷新，通过配合 @State 和重新读取状态后恢复正常。

通过本次实验，我对 HarmonyOS 应用中复杂界面交互和数据管理有了更深入的理解，提升了对页面结构设计和生命周期控制的能力，为后续进行更大规模的鸿蒙应用开发和课程设计提供了重要的实践经验。

实验 4 数据管理

一、实验目的

1. 掌握程序数据数据持久化。
2. 掌握 HarmonyOS 数据库的使用

实验内容和步骤

1. 数据库设计与初始化 (Database Initialization)

在 HarmonyOS 中，关系型数据库（RDB）基于 SQLite。实验中在应用入口 EntryAbility 的生命周期中完成了数据库的构建。

- 数据库配置：在 EntryAbility.ets 中配置了 StoreConfig，指定数据库文件名为 demodb.db。
- 表结构设计

  项目设计了多张关联表来支撑业务：
  - users：存储用户账号、密码哈希、个人信息。
  - facilities：存储场馆设施信息（名称、类型、图片映射）。
  - reservations：存储预约记录，包含时间段和状态。
  - activities&activity_participants：存储活动及报名关系。
  - ratings：存储用户评价。
- 建表与初始化数据

  使用 store.executeSql()执行 SQL 建表语句。为了提升测试体验，在建表成功后，通过 store.batchInsert()预置了初始数据（如默认用户 "bean" 和三个基础场馆设施）。

```
// EntryAbility.ets 核心代码片段
relationalStore.getRdbStore(this.context, STORE_CONFIG, (err, store) => {
  // 1. 建表
  store.executeSql(SQL_CREATE_USERS);
  store.executeSql(SQL_CREATE_FACILITIES);
  // ... 其他建表语句

  // 2. 将 store 实例注入工具类，供全局使用
  DBTools.setStore(store);

  // 3. 预置数据（防止重复插入）
  let pd = new relationalStore.RdbPredicates('facilities');
  store.query(pd, (err, resultSet) => {
    if (resultSet.rowCount === 0) {
      // 插入默认场馆数据...
    }
  })
});
```

2. 数据库工具类封装 (DBTools Encapsulation)

为了将业务逻辑与底层数据库操作解耦，实验封装了 DBTools.ets 工具类。该类采用静态方法提供服务，并广泛使用了 Promise 来处理异步数据库操作。

- 谓词查询 (RdbPredicates)： 对于简单的单表查询，使用 RdbPredicates 构建查询条件。

- 案例：登录验证 (queryByUser)。

```
static queryByUser(username: string, password: string): Promise<number> {
  let pd = new relationalStore.RdbPredicates('users');
  // 链式调用构建 AND 条件
  pd.equalTo('username', username).and().equalTo('password_hash', password);
  return DBTools.getStore().query(pd).then(...)
}
```

- 原生 SQL 查询 (Raw SQL)：对于复杂的多表关联查询（JOIN），直接执行 SQL 语句。
  - 案例：查询预约记录时，需要同时获取场馆名称 (queryReservationsByUserId)。

```
static queryReservationsByUserId(userId: number): Promise<Array<Reservation>> {
  const sql = `
    SELECT r.*, f.facility_name
    FROM reservations r
    LEFT JOIN facilities f ON r.facility_id = f.facility_id
    WHERE r.user_id = ?
    ORDER BY r.start_time DESC
  `;
  return DBTools.getStore().querySql(sql, [userId]).then(...)
}
```

3. 数据持久化业务实现 (Implementation)

实验实现了完整的增删改查业务流程，以下为举例：

- Create (增)
  - 用户注册：在 Register.ets 中，收集用户输入构建 ValuesBucket 对象，调用 DBTools.insert('users', data)将新用户写入数据库。
  - 创建预约：在 FacilityDetail.ets 中，插入预约记录前，先调用 checkReservationConflict 检查时间段是否冲突，保证了数据的逻辑完整性。
- Read (查)
  - Success.ets 页面加载时调用 queryFacilities。
  - MyReservations.ets 调用 queryReservationsByUserId 获取历史记录。
  - UserInfoEdit.ets 通过 queryUserById 获取当前用户信息并填充到输入框。
- Update (改)
  - 个人信息修改：在 UserInfoEdit.ets 中，调用 DBTools.updateById 更新用户的电话、邮箱等信息。
  - 状态更新：在 MyReservations.ets 中，点击“提前结束”会更新预约记录的 end_time。
- Delete (删)
  - 删除评价：在 MyRatings.ets 中，调用 DBTools.deleteRating 删除指定的评价记录。

三、实验小结及思考

通过本次数据管理实验，我系统地掌握了 HarmonyOS 应用中数据持久化的实现方法，深入理解了关系型数据库（RDB）在实际项目中的使用流程。

在实验过程中，我完成了数据库的设计、初始化以及业务数据的增删改查操作，使应用能够实现完整的数据存储与管理功能。

数据库设计阶段，我在 EntryAbility 的生命周期中完成了数据库的初始化工作，通过配置 StoreConfig 并创建多张业务关联表，构建了较为完整的数据结构体系。通过合理拆分 users、facilities、reservations、ratings 等数据表，保证了数据的规范性和可扩展性。同时，在建表完成后预置部分测试数据，提高了后续功能调试的效率。

数据库操作方面，通过封装 DBTools 工具类，将底层数据库操作与页面业务逻辑进行解耦。使用 Promise 处理异步数据库访问，使数据操作更加清晰和易于维护。对于简单查询，采用 RdbPredicates 构建条件，提高了代码可读性；对于复杂的多表关联查询，则直接使用原生 SQL，有效解决了 JOIN 查询需求。

具体业务实现过程，我完成了用户注册、设施查询、预约管理、个人信息修改以及评价管理等功能，覆盖了数据库操作的完整 CRUD 流程。在预约创建前增加时间冲突检测，保证了数据的逻辑正确性；在更新和删除操作中，通过主键条件精确定位数据，避免了误操作。

上机实验过程中也遇到了一些问题

1. 数据库文件路径不明确，初期无法定位 demodb.db 文件，通过查看应用沙箱目录并使用 hdc 工具导出后问题得到解决。

2. 异步数据库操作顺序问题，在未正确处理 Promise 的情况下出现数据尚未写入就执行查询的情况，通过使用 async/await 规范流程后解决。

3. SQL 语句书写错误，在多表关联查询中字段名冲突导致查询失败，通过显式指定表名和别名避免了歧义。

通过本次实验，我不仅掌握了 HarmonyOS 数据库的基本使用方法，也理解了数据层封装在实际项目中的重要性，为后续进行更复杂的业务逻辑开发和系统优化打下了坚实的基础。