实验十九　本地数据

一、实验目的

1. 了解 DevEco Studio 的使用
2. 学习 ArkTS 语言及本地数据
3. 编写代码
4. 编译运行
5. 在模拟器上运行

二、实验原理

1. 鸿蒙开发原理
2. ArkTS，ArkUI 开发原理
3. 鸿蒙应用运行原理

三、实验仪器材料

1. 计算机实训室电脑一台
2. DevEco Studio 开发环境及鸿蒙手机模拟器

四、 实验步骤

本地数据主要讲两类，一类是用于管理应用的状态的数据，主要是指 LocalStorage、AppStorage、PersistentStorage 等， 另外一类是本地关系型数据库，RelationalStore。

状态数据理论讲解：（这个通常归类于“状态管理”）

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-application-state-management-overview-V5

本地关系型数据库：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-data-relationalstore-V5#relationalstoregetrdbstore

关于 ArkData:

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/arkdata-api-V5

1. 打开 DevEco Studio，点击 Create Project 创建工程

设置项目名称为 StorageDemo。

2. LocalStorage

页面级 UI 状态存储，通常用于 UIAbility 内、页面间的状态共享。理论讲解：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-localstorage-V5

- 基本的使用

创建一个新的文件 LSAbc.ets，内容是下面的完整代码：

```
let para: Record<string,number> = { 'PropA': 47 };

let storage: LocalStorage = new LocalStorage(para); // 创建新实例并使用给定对象初始化

let prop: SubscribedAbstractProperty<number> = storage.prop('PropA'); // prop.get() ==
47

let link1: SubscribedAbstractProperty<number> = storage.link('PropA'); // link1.get() ==
47

let link2: SubscribedAbstractProperty<number> = storage.link('PropA'); // link2.get() ==
47

console.log('Original Value:--->')

console.log('data of PropA in Storage: ', storage.get('PropA'))

console.log('data of prop in Storage: ', prop.get())

console.log('data of link1 in Storage: ', link1.get())

console.log('data of link2 in Storage: ', link2.get())


link1.set(48); // two-way sync
```

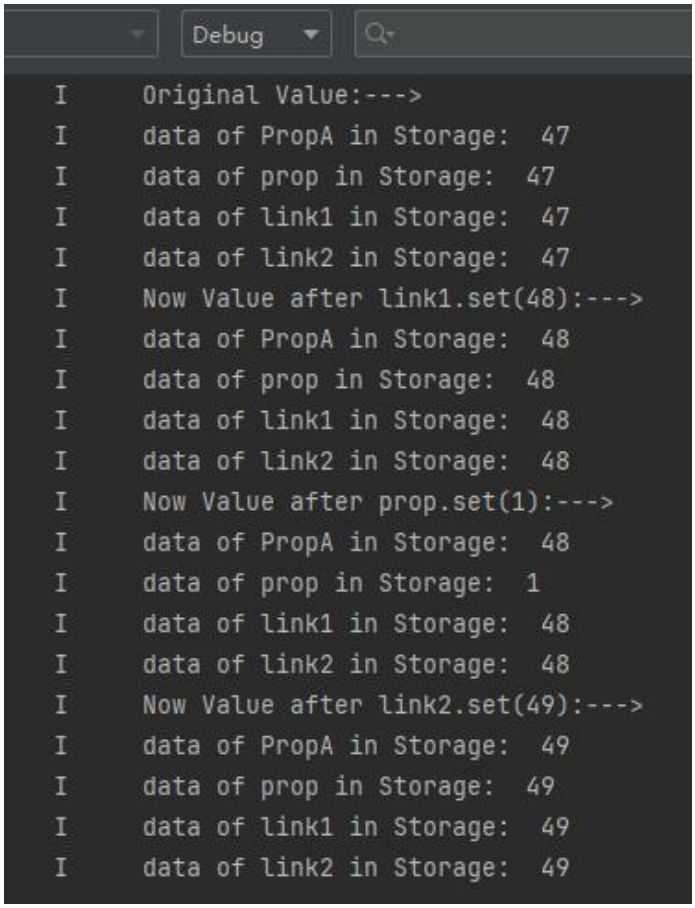
```
console.log('Now Value after link1.set(48):--->')
console.log('data of PropA in Storage: ', storage.get('PropA'))
console.log('data of prop in Storage: ', prop.get())
console.log('data of link1 in Storage: ', link1.get())
console.log('data of link2 in Storage: ', link2.get())

prop.set(1); // one-way sync
console.log('Now Value after prop.set(1):--->')
console.log('data of PropA in Storage: ', storage.get('PropA'))
console.log('data of prop in Storage: ', prop.get())
console.log('data of link1 in Storage: ', link1.get())
console.log('data of link2 in Storage: ', link2.get())

link2.set(49); // two-way sync
console.log('Now Value after link2.set(49):--->')
console.log('data of PropA in Storage: ', storage.get('PropA'))
console.log('data of prop in Storage: ', prop.get())
console.log('data of link1 in Storage: ', link1.get())
console.log('data of link2 in Storage: ', link2.get())

@Entry
@Component
```

```
struct LSAbc {
  build() {
  }
}
```

这里有几个重要的知识点：

- Record 是 TypeScript 内置的键值对数据类型
- SubscribedAbstractProperty<T> 是 LocalStorage 成员方法 link（）、prop（）、setAndLink（）和 setAndProp（）等等函数返回值的数据类型
- 通过 new LocalStorage 可以创建本地页面存储，并初始化，也就是把第一个键值存入
- 可以通过调用 LocalStorage 的 prop 和 link 函数，创建一些这个键值的别名，
  prop 是单向传递，也就是只读，
  而 link 是双向绑定
- 用 get 可以获取值，而 set 可以设置值

运行，查看 Log，请自己尝试理解这些输出的逻辑：



- 在 UI 内部使用

创建一个新的文件 LSUIBasic.ets，加入下面的完整代码：

```
let para: Record<string, number> = { 'PropA': 47 };

let storage: LocalStorage = new LocalStorage(para);


@Entry(storage)

@Component

struct CompA {

  @State tempA: number = Number(storage.get('PropA'))


  build() {

    Column({ space: 15 }) {
```

```
      Text(`PropA 在 LocalStorage 中的初始数值：${Number(storage.get('PropA'))}`)

      Text(`PropA 在 LocalStorage 中的数值：${this.tempA}`)

      Button(`PropA++`)
        .onClick(() => {
          this.tempA = Number(storage.get('PropA')) + 1
          storage.set('PropA', this.tempA)
        })
    }
  }
}
```

这段代码演示了在 ArkTS 中使用 LocalStorage 进行状态管理的基本用法，创建了一个简单的计数器界面。这段代码没有什么特别之处，唯一与普通页面的区别是在@Entry（storage）要加上了上面创建的 LocalStorage 实例，这样才能调用。@Entry(storage) 中的 storage 参数表示将 LocalStorage 实例 storage 与该入口组件关联，这样做的目的是让该组件及其子组件能够访问和操作这个共享的 LocalStorage 存储空间。

效果：

点击按钮，可以看到，数据在 LocalStorage 中被改变了。

加一个 PropB，红色字体：

```
let para: Record<string, number> = { 'PropA': 47 };
let storage: LocalStorage = new LocalStorage(para);
storage.setOrCreate('PropB', 20);
```

意思通过直接调用 setOrCreate 去创建一个新的键值。这个函数的意思是：如果键值不存在，就创建新的，如果存在，就设为新的值。

加一个状态变量：

```
  @State tempB: number = Number(storage.get('PropB'))
```

加一些组件：

```
      Text(`PropB 在 LocalStorage 中的初始数值：${Number(storage.get('PropB'))}`)


      Text(`PropB 在 LocalStorage 中的数值：${this.tempB}`)


      Button(`PropB--`)
        .onClick(() => {
          this.tempB = Number(storage.get('PropB')) - 1
          storage.set('PropB', this.tempB)
        })
```

效果：

用 setOrCreate 创建的，效果是一样的。

再加一个稍微复杂一点的键值，值为一个类的实例，@Entry 前面的代码（红色字体为新增）：

```
class PropC {
  username: string
  age: number

  constructor(username: string, age: number) {
    this.username = username;
    this.age = age;
  }
}

let para: Record<string, number> = { 'PropA': 47 };
```

```
let storage: LocalStorage = new LocalStorage(para);

storage.setOrCreate('PropB', 20);

storage.setOrCreate('PropC', new PropC('Michael', 20))
```

同样，加一个状态变量：

```
  @State tempC?: PropC = storage.get('PropC')
```

再加组件：

```
      Text(`PropC 在 LocalStorage 中的初始数值：${JSON.stringify(storage.get('PropC'))}`)


      Text(`PropC 在 LocalStorage 中的数值：${JSON.stringify(this.tempC)}`)


      Button(`PropC Age++`)
        .onClick(() => {
          if(this.tempC) {
            this.tempC.age = this.tempC.age + 1;
          }
          storage.set('PropC', this.tempC)
        })
```

效果：

可以看到，也可以存储类的实例，并操作改变。

- **@LocalStorageProp 变量装饰器**

LocalStorage 根据与@Component 装饰的组件的同步类型不同，提供了两个装饰器：

- @LocalStorageProp：@LocalStorageProp 装饰的变量与 LocalStorage 中给定属性建立单向同步关系。
- @LocalStorageLink：@LocalStorageLink 装饰的变量与 LocalStorage 中给定属性建立双向同步关系。

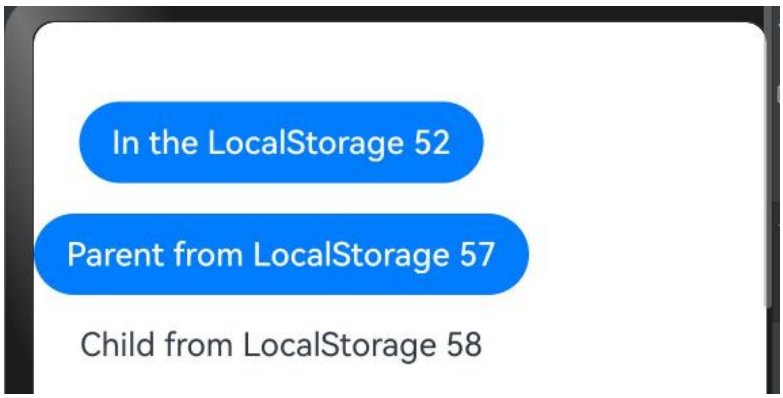
创建一个新的文件 LSProp.ets，加入下面的完整代码：

```
// 创建新实例并使用给定对象初始化
```

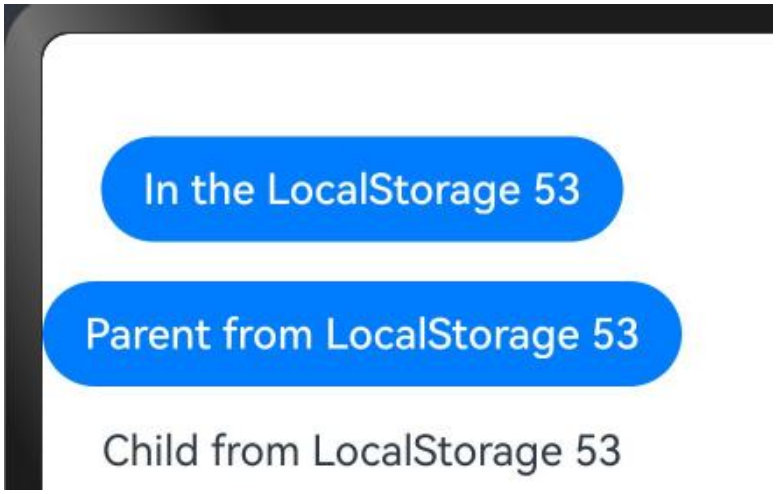
```
let para: Record<string, number> = { 'PropA': 47 };
let storage: LocalStorage = new LocalStorage(para);
// 使 LocalStorage 可从@Component 组件访问
@Entry(storage)
@Component
struct CompA {
  // @LocalStorageProp 变量装饰器与 LocalStorage 中的'PropA'属性建立单向绑定
  @LocalStorageProp('PropA') storageProp1: number = 1;
  @State temp: number = Number(storage.get('PropA'))

  build() {
    Column({ space: 15 }) {
      // 点击后从 47 开始加 1，只改变当前组件显示的 storageProp1，不会同步到 LocalStorage
中
      Button(`In the LocalStorage ${this.temp}`)
        .onClick(() => {
          this.temp = Number(storage.get('PropA')) + 1
          storage.set('PropA', this.temp)
        })

      Button(`Parent from LocalStorage ${this.storageProp1}`)
        .onClick(() => {
```

```
            this.storageProp1 += 1
        })
      Child()
    }
  }
}

@Component
struct Child {
  // @LocalStorageProp 变量装饰器与 LocalStorage 中的'PropA'属性建立单向绑定
  @LocalStorageProp('PropA') storageProp2: number = 2;

  build() {
    Column({ space: 15 }) {
      // 当 CompA 改变时，当前 storageProp2 不会改变，显示 47
      Text(`Child from LocalStorage ${this.storageProp2}`)
        .onClick(() => {
          this.storageProp2++
        })
    }
  }
}
```

效果：



在子组件中的值变化的时候不会影响 LocalStorage 中的值，而一旦 LocalStorage 中的值发生变化，子组件中的值会被同步。在上面的情况下，点击最上面的按钮：



注意，本地初始化是必要的，因为无法保证 LocalStorage 一定存在给定的 key（这取决于应用逻辑是否在组件初始化之前在 LocalStorage 实例中存入对应的属性）

- **@LocalStorageLink 变量装饰器**

创建一个新的文件 LSLink.ets，加入代码：

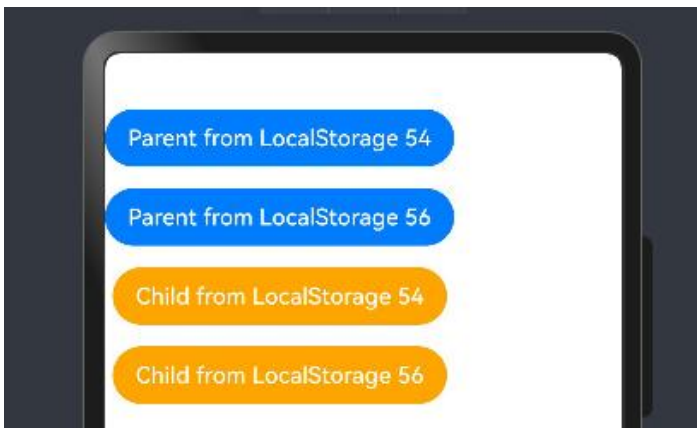
```
class PropB {
  code: number;

  constructor(code: number) {
```

```
    this.code = code;
  }
}
// 创建新实例并使用给定对象初始化
let para: Record<string, number> = { 'PropA': 47 };
let storage: LocalStorage = new LocalStorage(para);
storage.setOrCreate('PropB', new PropB(50));

@Component
struct Child {
  // @LocalStorageLink 变量装饰器与 LocalStorage 中的'PropA'属性建立双向绑定
  @LocalStorageLink('PropA') childLinkNumber: number = 1;
  // @LocalStorageLink 变量装饰器与 LocalStorage 中的'PropB'属性建立双向绑定
  @LocalStorageLink('PropB') childLinkObject: PropB = new PropB(0);

  build() {
    Column({ space: 15 }) {
      Button(`Child from LocalStorage ${this.childLinkNumber}`) // 更改将同步至
LocalStorage 中的'PropA'以及 Parent.parentLinkNumber
        .onClick(() => {
          this.childLinkNumber += 1;
        })
```

```
            .backgroundColor(Color.Orange)
        Button(`Child from LocalStorage ${this.childLinkObject.code}`) // 更改将同步至
LocalStorage 中的'PropB'以及 Parent.parentLinkObject.code
            .onClick(() => {
              this.childLinkObject.code += 1;
            })
            .backgroundColor(Color.Orange)
      }
    }
}
// 使 LocalStorage 可从@Component 组件访问
@Entry(storage)
@Component
struct CompA {
  // @LocalStorageLink 变量装饰器与 LocalStorage 中的'PropA'属性建立双向绑定
  @LocalStorageLink('PropA') parentLinkNumber: number = 1;
  // @LocalStorageLink 变量装饰器与 LocalStorage 中的'PropB'属性建立双向绑定
  @LocalStorageLink('PropB') parentLinkObject: PropB = new PropB(0);

  build() {
    Column({ space: 15 }) {
```

```
        Button(`Parent from LocalStorage ${this.parentLinkNumber}`) // initial value from
LocalStorage will be 47, because 'PropA' initialized already
          .onClick(() => {
            this.parentLinkNumber += 1;
          })

        Button(`Parent from LocalStorage ${this.parentLinkObject.code}`) // initial value from
LocalStorage will be 50, because 'PropB' initialized already
          .onClick(() => {
            this.parentLinkObject.code += 1;
          })
        // @Component 子组件自动获得对 CompA LocalStorage 实例的访问权限。
        Child()
      }
    }
}
```

效果：



可以看到，主组件里面放置了一个 Child 子组件。点击第一个按钮和第三个按钮中任何一个，后面的数值都是同步变化的。第二个按钮和第四个按钮也是一样，因为通过@LocalStorageLink 修饰的变量与 LocalStorage 是双向同步的。

3. AppStorage

AppStorage 是应用全局的 UI 状态存储，是和应用的进程绑定的，由 UI 框架在应用程序启动时创建，它的目的是为了提供应用状态数据的中心存储，这些状态数据在应用级别都是可访问的。AppStorage 将在应用运行过程保留其属性。属性通过唯一的键字符串值访问。

和 AppStorage 不同的是，LocalStorage 是页面级的，通常应用于页面内的数据共享。而 AppStorage 是应用级的全局状态共享，还相当于整个应用的“中枢”，持久化数据 PersistentStorage 和环境变量 Environment 都是通过 AppStorage 中转，才可以和 UI 交互。

理论讲解：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-appstorage-V5

AppStorage 是在应用启动的时候会被创建的单例，同时它的所有 API 都是静态的，因此可以直接调用而不需要创建对象的实例。

- **基础用法**

创建一个新的文件：ASAbc.ets，加入代码：

```
AppStorage.setOrCreate('PropA', 47);
```

```
let storage: LocalStorage = new LocalStorage();
storage.setOrCreate('PropA',17);
let propA: number | undefined = AppStorage.get('PropA') // propA in AppStorage == 47,
propA in LocalStorage == 17
```

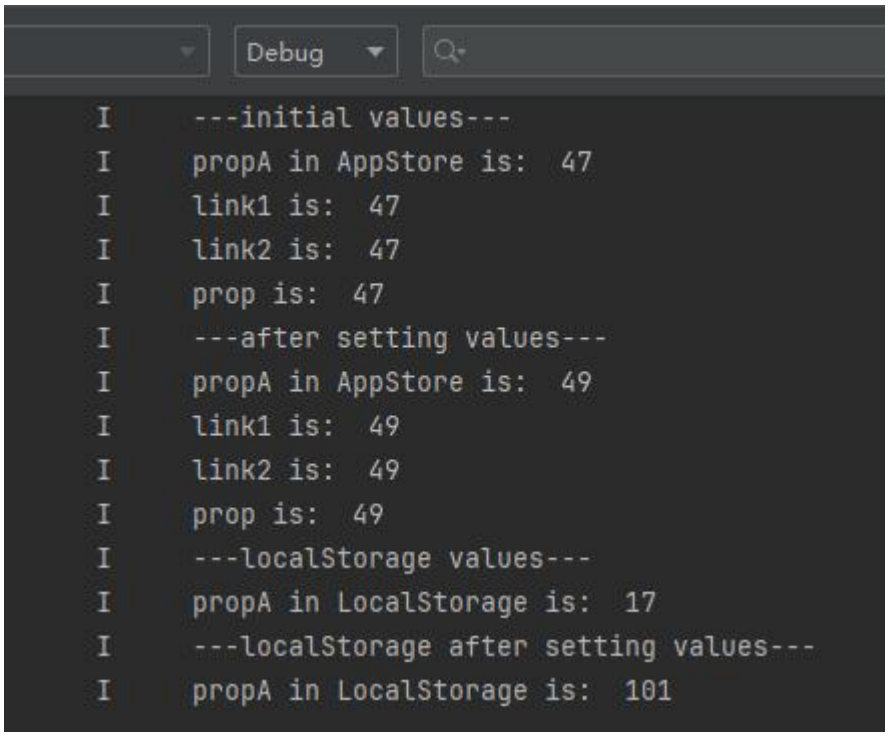
```
let link1: SubscribedAbstractProperty<number> = AppStorage.link('PropA'); // link1.get()
== 47
let link2: SubscribedAbstractProperty<number> = AppStorage.link('PropA'); // link2.get()
== 47
let prop: SubscribedAbstractProperty<number> = AppStorage.prop('PropA'); // prop.get()
== 47

console.log('---initial values---')
console.log('propA in AppStore is: ', propA)
console.log('link1 is: ', link1.get())
console.log('link2 is: ', link2.get())
console.log('prop is: ', prop.get())

link1.set(48); // two-way sync: link1.get() == link2.get() == prop.get() == 48
prop.set(1); // one-way sync: prop.get() == 1; but link1.get() == link2.get() == 48
link1.set(49); // two-way sync: link1.get() == link2.get() == prop.get() == 49

console.log('---after setting values---')
console.log('propA in AppStore is: ', AppStorage.get('PropA'))
console.log('link1 is: ', link1.get())
console.log('link2 is: ', link2.get())
console.log('prop is: ', prop.get())
```

```
console.log('---localStorage values---')
console.log('propA in LocalStorage is: ', storage.get<number>('PropA'))

storage.set('PropA', 101);

console.log('---localStorage after setting values---')
console.log('propA in LocalStorage is: ', storage.get<number>('PropA'))

@Entry
@Component
struct ASAbc {
  build() {

  }
}
```

这里同时使用了 AppStorage 和 LocalStorage，可以看到，AppStorage 是静态的，无需自己去创建实例，直接使用即可。这里在 AppStorage 和 LocalStorage 中都创建了一个叫 PropA 的键值，通过获取不同的 Storage 的值来区别。

和 LocalStorage 类似，这里也有 get，set，link，prop 这些函数。

运行，查看 Log：



这些应该是比较清楚，简单的，请自行运行并理解。

类似地，APPStorage 提供了两个装饰器：

- @StorageProp：单向数据同步
- @StorageLink：双向数据同步

- **在 UI 中使用（以@StorageLink 为例）**

创建一个新的文件：ASUIBasic.ets，加入代码：

```
class PropB {
  code: number;

  constructor(code: number) {
    this.code = code;
```

```
  }
}

AppStorage.setOrCreate('PropA', 47);
AppStorage.setOrCreate('PropB', new PropB(50));
let storage = new LocalStorage();
storage.setOrCreate('PropA', 48);
storage.setOrCreate('PropB', new PropB(100));

@Entry(storage)
@Component
struct CompA {
  @StorageLink('PropA') storageLink: number = 1;
  @State appStoragePropA?: number = AppStorage.get<number>('PropA')
  @LocalStorageLink('PropA') localStorageLink: number = 1;
  @State localStoragePropA?: number = storage.get<number>('PropA')
  @StorageLink('PropB') storageLinkObject: PropB = new PropB(1);
  @State appStoragePropB?: PropB = AppStorage.get<PropB>('PropB')
  @LocalStorageLink('PropB') localStorageLinkObject: PropB = new PropB(1);
  @State localStoragePropB?: PropB = storage.get<PropB>('PropB')

  build() {
```

```
    Column({ space: 20 }) {
      Column({ space: 20 }) {
        Text(`PropA in AppStorage ${this.appStoragePropA}`)
        Text(`PropA From AppStorage ${this.storageLink}`)
          .onClick(() => {
            this.storageLink += 1;
            this.appStoragePropA = AppStorage.get<number>('PropA')
          })
      }.backgroundColor(Color.Green)

      Column({ space: 20 }) {
        Text(`PropA in LocalStorage ${this.localStoragePropA}`)
        Text(`PropA From LocalStorage ${this.localStorageLink}`)
          .onClick(() => {
            this.localStorageLink += 1;
            this.localStoragePropA = storage.get<number>('PropA')
          })
      }.backgroundColor(Color.Gray)

      Column({ space: 20 }) {
        Text(`PropB in AppStorage ${this.appStoragePropB?.code}`)
        Text(`From AppStorage ${this.storageLinkObject.code}`)
```

```
            .onClick(() => {
              this.storageLinkObject.code += 1;
              this.appStoragePropB = AppStorage.get<PropB>('PropB')
            })
        }.backgroundColor(Color.Yellow)

        Column({ space: 20 }) {
          Text(`PropB in LocalStorage ${this.localStoragePropB?.code}`)
          Text(`From LocalStorage ${this.localStorageLinkObject.code}`)
            .onClick(() => {
              this.localStorageLinkObject.code += 1;
              this.localStoragePropB = storage.get<PropB>('PropB')
            })
        }.backgroundColor(Color.Pink)
    }
  }
}
```

上面的代码中，同时使用了 AppStorage 和 LocalStorage 去存储 PropA 和 PropB 的值。

通过@StorageLink 可以双向同步组件与 AppStorage 中的键值。

效果：

点击每一个区域的下面的文字，可以看到数据同步变化。

4. PersistentStorage

前两个小节介绍的 LocalStorage 和 AppStorage 都是运行时的内存存储，当程序退出后，值便不再保存。实际应用中，我们有时候需要应用退出后再次启动时，依然能保存选定的结果，这是应用开发中十分常见的现象，这就需要用到 PersistentStorage。

PersistentStorage 是应用程序中的可选单例对象。此对象的作用是持久化存储选定的 AppStorage 属性，以确保这些属性在应用程序重新启动时的值与应用程序关闭时的值相同。

理论讲解：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-persiststorage-V5

注意：PersistentStorage 将选定的 AppStorage 属性保留在设备磁盘上。应用程序通过 API，以决定哪些 AppStorage 属性应借助 PersistentStorage 持久化。 UI 和业务逻辑不直接访问 PersistentStorage 中的属性，所有属性访问都是对 AppStorage 的访问，AppStorage 中的更改会自动同步到 PersistentStorage。

PersistentStorage 和 AppStorage 中的属性建立双向同步。应用开发通常通过 AppStorage 访问

PersistentStorage，另外还有一些接口可以用于管理持久化属性，但是业务逻辑始终是通过 AppStorage 获取和设置属性的。

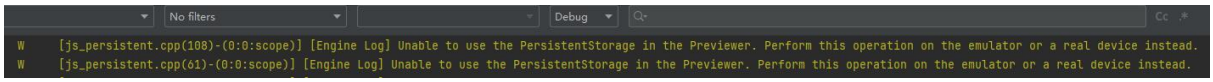
通过实例来理解，创建一个新的文件 PSAbc.ets，加入代码：

```
PersistentStorage.persistProp('aProp', 47);
AppStorage.setOrCreate('bProp', 20);

@Entry
@Component
struct PSAbc {
  @State message: string = 'aProp in PersistentStorage'
  @StorageLink('aProp') aProp: number = 48
  @StorageLink('bProp') bProp: number = 1

  build() {
    Row() {
      Column({space: 15}) {
        Text(this.message)
        // 应用退出时会保存当前结果。重新启动后，会显示上一次的保存结果
        Text(`${this.aProp}`)
          .onClick(() => {
            this.aProp += 1;
          })
```

```
        Text('bProp in AppStore')
        Text(`${this.bProp}`)
          .onClick(() => {
            this.bProp++
          })
      }
    }
    .width('100%')
    .height('100%')
    .justifyContent(FlexAlign.Center)
  }
}
```

在这里，aProp 是通过 PersistentStorage.persistProp 函数来创建的，创建的同时，其实也自动在 AppStorage 中创建了相同的键值。而 bProp 是仅仅在 AppStorage 中创建的。

在 Previewer 运行时，Log 中有提示：



也就是说在预览器中是看不出效果的。因此，我们在模拟器中运行，页面显示之后，点击两个数字让其增加：

我们可以在模拟器中运行程序，注意需要自行修改启动页面（还记得怎么修改么）。点击界面上的数字修改 aProp 和 bProp 的值，然后通过上滑操作，杀掉 APP 的进程：

然后在模拟器上再次打开：

可以看到，aProp 因为是保存在 PersistentStorage 里面，所以重新启动后 aProb 的值是我们修改后的值 52，而 bProp 仅仅是在 AppStorage 里面，所以重新被初始化恢复成了 20。

5. RelationalStore 本地关系型数据库

关系型数据库（Relational Database，RDB）是一种基于关系模型来管理数据的数据库。关系型数据库基于 SQLite 组件提供了一套完整的对本地数据库进行管理的机制，对外提供了一系列的增、删、改、查等接口，也可以直接运行用户输入的 SQL 语句来满足复杂的场景需要。

理论讲解：

https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-data-relationalstore-V5

我们通过实例来学习。

首先，创建一个新的项目，DataBaseDemo。

- **获取 RdbStore（创建一个数据库）**

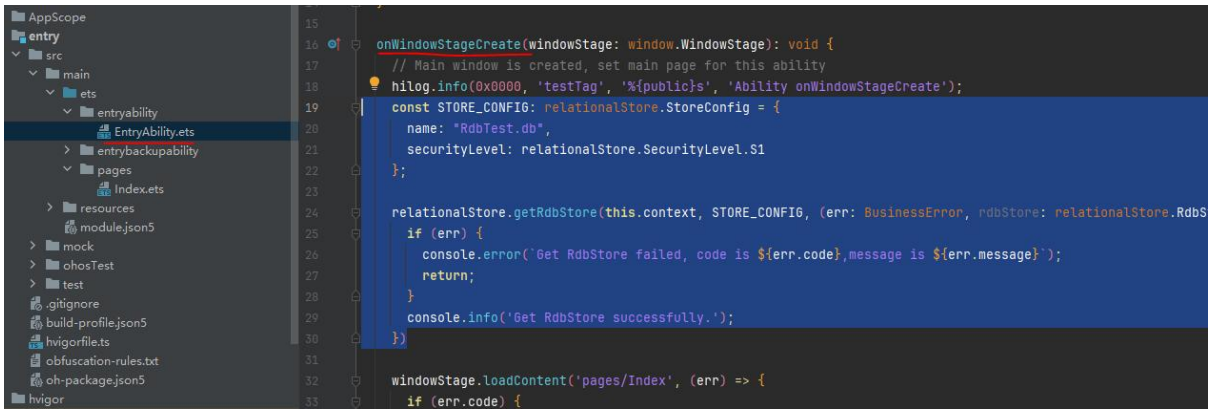
项目创建好之后，先到 EntryAbility.ets 文件中，找到生命周期函数 onWindowStageCreate，加入获取一个 RdbStore 的代码：

```
    const STORE_CONFIG: relationalStore.StoreConfig = {
        name: "RdbTest.db",
        securityLevel: relationalStore.SecurityLevel.S1
    };

    relationalStore.getRdbStore(this.context, STORE_CONFIG, (err: BusinessError, rdbStore:
relationalStore.RdbStore) => {
        if (err) {
            console.error(`Get RdbStore failed, code is ${err.code},message is ${err.message}`);
```

```
        return;
      }
      console.info('Get RdbStore successfully.');
    })
```

相应地，在文件最前面，要导入：

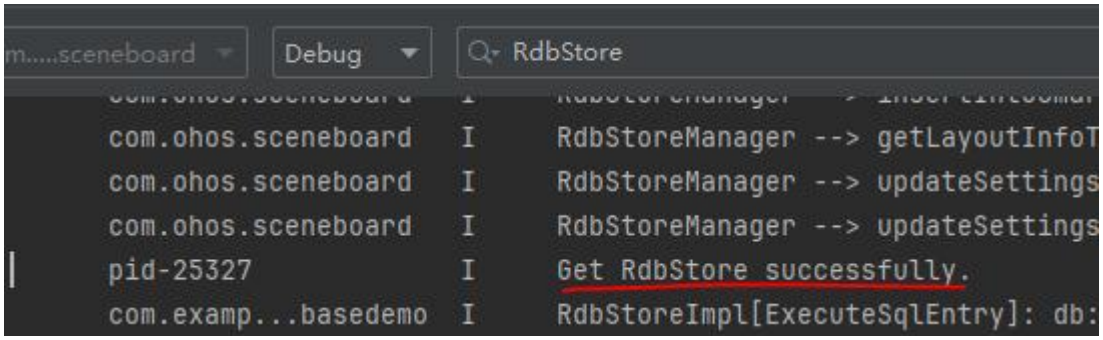
```
import { relationalStore } from '@kit.ArkData';
import { BusinessError } from '@kit.BasicServicesKit';
```



函数：getRdbStore(context: Context, config: StoreConfig, callback: AsyncCallback<RdbStore>): void

获得一个相关的 RdbStore，操作关系型数据库，用户可以根据自己的需求配置 RdbStore 的参数，然后通过 RdbStore 调用相关接口可以执行相关的数据操作，使用 callback 异步回调。

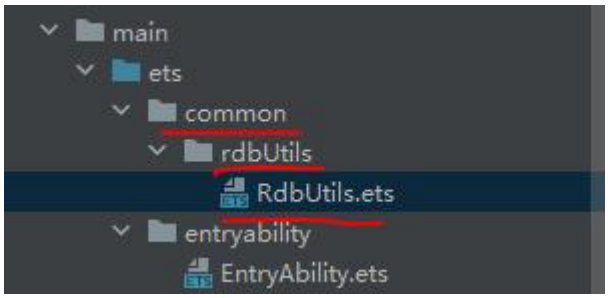
在模拟器上运行，可以看到 Log（注意要过滤一下）：



- **创建 RdbUtils 工具包**

为了方便我们后面在 UI 中操作数据库，我们在项目文件中创建 common/rdbUtils/RdbUtils.ts 文件，

作为数据库操作工具包：



加入代码：

```
import { relationalStore } from '@kit.ArkData';


export default class RdbUtils {
  private static rdbStore: relationalStore.RdbStore;
  static setStore(store: relationalStore.RdbStore) {
    RdbUtils.rdbStore = store;
  }
  static getStore(): relationalStore.RdbStore {
    return RdbUtils.rdbStore;
  }
}
```

这里有一个静态变量 rdbStore，用来保存我们获取的 RdbStore 的实例。

回到 EntryAbility.ets，加入代码（红色字体），保存 RdbStore 实例：

```
    relationalStore.getRdbStore(this.context, STORE_CONFIG, (err: BusinessError, rdbStore:
relationalStore.RdbStore) => {
        if (err) {
```

```
            console.error(`Get RdbStore failed, code is ${err.code},message is ${err.message}`);
            return;
        }
        console.info('Get RdbStore successfully.');
        // 保存 rdbStore，方便后续的操作
        RdbUtils.setStore(rdbStore)
    })
```

当然，前面会需要导入：

```
import RdbUtils from '../common/rdbUtils/RdbUtils';
```

- **创建数据库表格**

后续一系列的操作都需要运行 SQL 语句，所以，先到 RdbUtils.ets 文件中，给 RdbUtils 类添加一个 executeSql 方法，加入代码：

```
export default class RdbUtils {
...
  static executeSql(sql: string): Promise<void> {
    return RdbUtils.getStore().executeSql(sql)
  }
}
```

可以看到，其实是通过 RdbStore 本身的 executeSql 函数去执行，放在这里是因为正好可以调用 getStore()去获取当前的实例。

回到 Index.ets，把代码替换为：

```
import RdbUtils from '../common/rdbUtils/RdbUtils';
```

```
import { BusinessError } from '@kit.BasicServicesKit';


@Entry
@Component
struct Index {
  build() {
    Row() {
      Column({space: 10}) {
        Button('创建数据库表')
          .onClick(() => {
            const SQL_CREATE_TABLE = 'CREATE TABLE IF NOT EXISTS EMPLOYEE (ID
INTEGER PRIMARY KEY AUTOINCREMENT, NAME TEXT NOT NULL, AGE INTEGER, SALARY
REAL, CODES BLOB)'; // 建表 Sql 语句
            RdbUtils.executeSql(SQL_CREATE_TABLE)
              .then(() =>{
                openDialog('成功','successfully created table')
              }).catch((err: BusinessError) => {
                openDialog('错误', err.message);
            })
          })
      }
      .width('100%')
```

```
    }
    .height('100%')
  }
}

function openDialog(title: string, text: string) {
  AlertDialog.show(
    {
      title: title,
      message: text,
      autoCancel: true,
      alignment: DialogAlignment.Bottom,
      offset: { dx: 0, dy: -20 },
    }
  )
}
```

代码的核心就是定义了一个 SQL 语句创建一个数据表，然后执行这条语句。语句执行成功的话就给一个弹窗出来。

这个表格列名及类型:

ID	一个整数类型的列，被设置为主键，并且自动递增。这意味着每次添加新记录时，此列的值会自动增加，不允许重复。
NAME	文本类型的列，不允许为空。这意味着在插入或更新记录时，必须为这一列提供值。

AGE	整数类型的列，可以包含任何整数值。
SALARY	浮点数类型的列，可以用于存储工资或薪水的值。
CODES	BLOB 类型的列，通常用于存储二进制数据。它可以用于存储代码、数据或其他二进制格式的信息。

约束:

由于"ID"列被指定为主键，所以每个"ID"值必须是唯一的，并且不能为空。

"NAME"列被设置为"NOT NULL"，这意味着每次插入或更新记录时，必须为这一列提供值。

模拟器上运行，效果：

- **增删改查**

有了表格，就可以对其中的数据进行增删改查。

增 - 插入数据

先到 RdbUtils 中添加 insert()方法，insert 的方法说明：

insert

insert(table: string, values: ValuesBucket):Promise<number>

向目标表中插入一行数据，使用Promise异步回调。由于共享内存大小限制为2Mb，因此单条数据的大小需小于2Mb，否则会查询失败。

系统能力： SystemCapability.DistributedDataManager.RelationalStore.Core

参数：

参数名	类型	必填	说明
table	string	是	指定的目标表名。
values	ValuesBucket	是	表示要插入到表中的数据行。

返回值：

类型	说明
Promise<number>	Promise对象。如果操作成功，返回行ID；否则返回-1。

```
export default class RdbUtils {
...
  static insert(tableName: string, data: ValuesBucket): Promise<number> {
    return RdbUtils.getStore().insert(tableName, data);
  }
}
```

请自行添加导入。

然后，回到 Index.ets，添加一个新的按钮：

```
          Button('插入数据')
            .onClick(() => {
              const valueBucket: ValuesBucket = {
                'NAME': 'Lisa',
```

```
                'AGE': 18,
                'SALARY': 5000,
                'CODES': new Uint8Array([1, 2, 3, 4, 5])
            };
            RdbUtils.insert('EMPLOYEE', valueBucket)
                .then((updateNumber) => {
                    openDialog('插入数据成功', '已插入数据:' + updateNumber)
                }).catch((err: BusinessError) => {
                openDialog('操作失败', err.message)
            })
        })
```

注意 insert 执行成功的话，返回的 Promise 带的是插入的数据的行的 ID 值。

效果：

查 - 查询数据

查询数据可以通过 RdbStore 中的 query()方法执行查询：

query

query(predicates: RdbPredicates, columns?: Array<string>):Promise<ResultSet>

根据指定条件查询数据库中的数据，使用Promise异步回调。由于共享内存大小限制为2Mb，因此单条数据的大小需小于2Mb，否则会查询失败。

系统能力： SystemCapability.DistributedDataManager.RelationalStore.Core

参数：

参数名	类型	必填	说明
predicates	RdbPredicates	是	RdbPredicates的实例对象指定的查询条件。
columns	Array<string>	否	表示要查询的列。如果值为空，则查询应用于所有列。

这里要先利用 RdbPredicates 设置好查询条件，关于 RdbPredicates：

RdbPredicates

表示关系型数据库（RDB）的谓词。该类确定RDB中条件表达式的值是true还是false。谓词间支持多语句拼接，拼接时默认使用and()连接。不支持Sendable跨线程传递。

constructor

constructor(name: string)

构造函数。

系统能力： SystemCapability.DistributedDataManager.RelationalStore.Core

参数：

参数名	类型	必填	说明
name	string	是	数据库表名。

错误码：

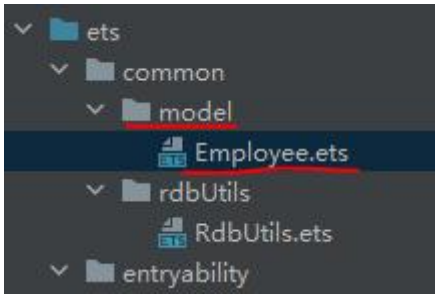
以下错误码的详细介绍请参见通用错误码。

错误码ID	错误信息
401	Parameter error. Possible causes: 1. Mandatory parameters are left unspecified; 2. Incorrect parameter types.

示例：

```
let predicates = new relationalStore.RdbPredicates("EMPLOYEE");
```

为了更好地使用数据，先在 common 目录下建一个 model 子目录，创建一个文件 Employee.ets:



加入代码，创建一个类：

```
export default class Employee {
  id: number; // 员工 ID
  name: string; // 姓名
  age: number | null; // 年龄，允许为 null
```

```
  salary: number | null; // 工资，允许为 null
  codes: Uint8Array | null; // 二进制数据，用于存储 BLOB 类型，允许为 null

  constructor(id: number, name: string, age: number | null, salary: number | null, codes:
Uint8Array | null) {
    this.id = id;
    this.name = name;
    this.age = age;
    this.salary = salary;
    this.codes = codes;
  }
}
```

在 RdbUtils 中，添加一个新的 queryAll 函数：

```
static queryAll(): Promise<Array<Employee>> {
    let predicates = new relationalStore.RdbPredicates('EMPLOYEE');
    return new Promise<Array<Employee>>((resolve, reject) => {
      RdbUtils.getStore().query(predicates).then((result) => {
        let employees = new Array<Employee>();
        while (result.goToNextRow()) {
          let employee = new Employee(
            result.getLong(0),
            result.getString(1),
```

```
                result.getLong(2),
                result.getDouble(3),
                result.getBlob(4)
            );
            employees.push(employee);
        }
        resolve(employees);
    }).catch((error: BusinessError) => {
        reject(error)
    })
  })
}
```

在 Index.ets 中，再加一个按钮：

```
Button('查询数据')
  .onClick(() => {
    RdbUtils.queryAll()
      .then((employees: Array<Employee>) => {
        openDialog('查询成功', 'employees: ' + JSON.stringify(employees))
      }).catch((error: BusinessError) => {
      openDialog('查询失败', error.message)
    })
  })
```

效果：

改 - 修改数据

修改数据可以通过 RdbStore 中的 update()方法执行修改：

update

update(values: ValuesBucket, predicates: RdbPredicates):Promise<number>

根据RdbPredicates的指定实例对象更新数据库中的数据，使用Promise异步回调。由于共享内存大小限制为2Mb，因此单条数据的大小需小于2Mb，否则会查询失败。

系统能力： SystemCapability.DistributedDataManager.RelationalStore.Core

参数：

参数名	类型	必填	说明
values	ValuesBucket	是	values指示数据库中要更新的数据行。键值对与数据库表的列名相关联。
predicates	RdbPredicates	是	RdbPredicates的实例对象指定的更新条件。

返回值：

类型	说明
Promise<number>	指定的Promise回调方法。返回受影响的行数。

先在 RdbUtils.ets 中添加函数：

```
static updateById(id: number, data: ValuesBucket) {
    let predicates = new relationalStore.RdbPredicates('EMPLOYEE');
    predicates.equalTo('ID', id)
    return RdbUtils.getStore().update(data, predicates);
}
```

注意，这里的 predicates.equalTo(‘ID’,id)方法可以帮助我们找到 id 对应的数据行，然后再执行操作。

到 Index.ets 中，再添加一个按钮：

```
        Button('修改数据')
            .onClick(() => {
                const valueBucket: ValuesBucket = {
```

```
            'NAME': 'Rose',
            'AGE': 99,
            'SALARY': 6000,
            'CODES': new Uint8Array([1, 2, 3, 4, 5])
        };
        RdbUtils.updateById(1, valueBucket)
          .then((updateNumber) => {
            openDialog('更新成功', '已更新数据:' + updateNumber.toString())
          }).catch((err: BusinessError) => {
          openDialog('更新失败', err.message)
        })
    })
```

效果：

再查询一次：

可以看到，数据已经是新的数据了。

删 - 删除数据

删除数据可以通过 RdbStore 中的 delete()方法执行删除：

delete

delete(predicates: RdbPredicates):Promise<number>

根据RdbPredicates的指定实例对象从数据库中删除数据，使用Promise异步回调。

系统能力： SystemCapability.DistributedDataManager.RelationalStore.Core

参数：

参数名	类型	必填	说明
predicates	RdbPredicates	是	RdbPredicates的实例对象指定的删除条件。

返回值：

类型	说明
Promise<number>	Promise对象。返回受影响的行数。

类似的方法，先到 RdbUtils.ets 中添加方法：

```
  static deleteById(id: number) {
    let predicates = new relationalStore.RdbPredicates('EMPLOYEE');
    predicates.equalTo('ID', id)
    return RdbUtils.getStore().delete(predicates);
  }
```

再到 Index.ets 中添加一个按钮：

```
          Button('删除数据')
            .onClick(() => {
              RdbUtils.deleteById(1)
                .then((updateNumber) => {
                  openDialog('删除成功', '已删除数据:' + updateNumber.toString())
                }).catch((err: BusinessError) => {
                  openDialog('删除失败', err.message)
                })
            })
```

效果：

再次查询，因为数据已经被删除，结果为空：

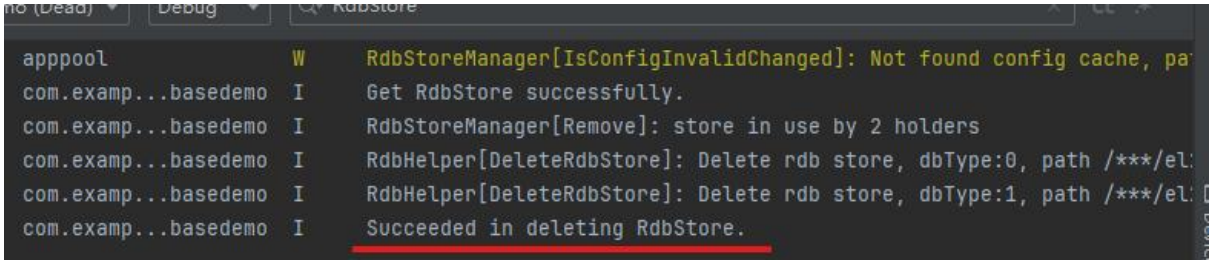
到此，我们完成了创建表格，基本的增删改查的功能。

- **删除数据库**

删除数据库可以通过调用 deleteRdbStore 的方法去删除数据库以及相关的文件。可以在 EntryAbility 的 onDestroy 生命周期函数中调用，代码：

```
onDestroy(): void {
    hilog.info(0x0000, 'testTag', '%{public}s', 'Ability onDestroy');
    relationalStore.deleteRdbStore(this.context, 'RdbTest.db', (err) => {
        if (err) {
            console.error(`Failed to delete RdbStore. Code:${err.code},
message:${err.message}`);
            return;
        }
        console.info('Succeeded in deleting RdbStore.');
    });
}
```

模拟器上运行之后，杀掉 APP 进程，查看 Log：



到此，我们就完成了一个本地数据库的创建，新建表格，对表格进行增删改查的操作，以及删除数据库的操作。

五、实验注意事项

1.注意教师的操作演示。

2.学生机与教师机内网连通，能接收和提交实验结果。

3.按实验要求输入测试数据，并查看输出结果是否符合实验结果。

六、思考题

1.通过这个实验，你学到了什么？