实验十四　自定义组件状态管理 V2

一、实验目的

1. 了解 DevEco Studio 的使用
2. 学习 ArkTS 语言，掌握自定义组件的状态管理
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

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

配置好项目名称（如 CompStateV2），存放位置（上图是放在 D 盘某个目录下），设备类型等，然后点击 Finish 按钮，进入到开发界面，项目创建成功。

2. 状态管理概述

在华为鸿蒙开发者官网上，状态管理概述：

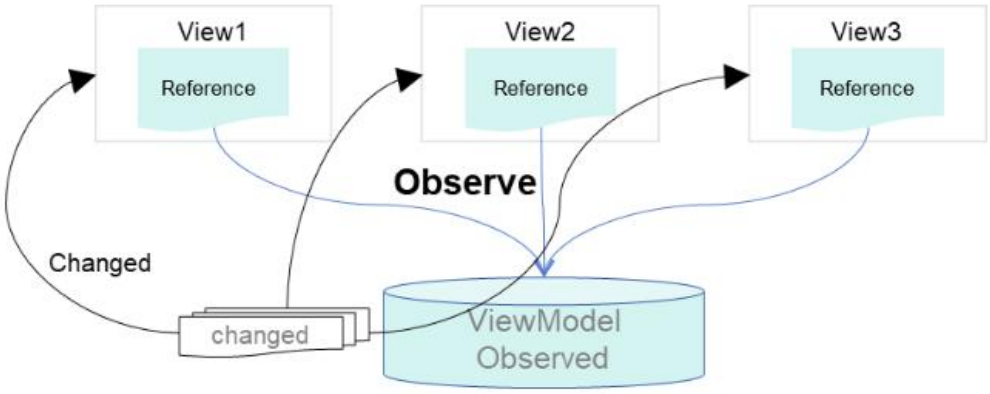
https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/arkts-state-management-overview-V5

这里关于状态管理 V2 版本，提到：“为了增强状态管理 V1 版本的部分能力，例如深度观察、属性级更新等，ArkUI 推出状态管理 V2 供开发者使用。”

V2 重大的逻辑变化：

状态管理V2将观察能力增强到数据本身，数据本身就是可观察的，更改数据会触发相应的视图的更新。相较于状态管理V1，状态管理V2有如下优点：

- 状态变量独立于UI，更改数据会触发相应视图的更新。
- 支持对象的深度观测和深度监听，且深度观测机制不影响观测性能。
- 支持对象中属性级精准更新及数组中元素的最小化更新。
- 装饰器易用性高、拓展性强，在组件中明确输入与输出，有利于组件化。



V2 推出了一整套全新的装饰器：

装饰器总览

状态管理（V2）提供了一套全新的装饰器。

- @ObservedV2：@ObservedV2装饰器装饰class，使得被装饰的class具有深度监听的能力。@ObservedV2和@Trace配合使用可以使class中的属性具有深度观测的能力。
- @Trace：@Trace装饰器装饰被@ObservedV2装饰的class中的属性，被装饰的属性具有深度观测的能力。
- @ComponentV2：使用@ComponentV2装饰的struct中能使用新的装饰器。例如：@Local、@Param、@Event、@Once、@Monitor、@Provider、@Consumer。
- @Local：@Local装饰的变量为组件内部状态，无法从外部初始化。
- @Param：@Param装饰的变量作为组件的输入，可以接受从外部传入初始化并同步。
- @Once：@Once装饰的变量仅初始化时同步一次，需要与@Param一起使用。
- @Event：@Event装饰方法类型，作为组件输出，可以通过该方法影响父组件中变量。
- @Monitor：@Monitor装饰器用于@ComponentV2装饰的自定义组件或@ObservedV2装饰的类中，能够对状态变量进行深度监听。
- @Provider和@Consumer：用于跨组件层级双向同步。
- @Computed：计算属性，在被计算的值变化的时候，只会计算一次。主要应用于解决UI多次重用该属性从而重复计算导致的性能问题。
- !!语法：双向绑定语法糖。

我们选其中几个重要的通过实例来理解。

3. @ObservedV2 装饰器和@Trace 装饰器：类属性变化观测

@ObservedV2 和@Trace 提供了对嵌套类对象属性变化直接观测的能力，是状态管理 V2 中相对核心的能力之一。

概述

@ObservedV2装饰器与@Trace装饰器用于装饰类以及类中的属性，使得被装饰的类和属性具有深度观测的能力：

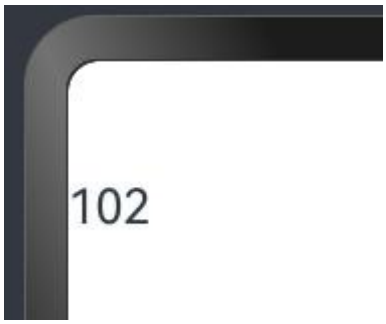
- @ObservedV2装饰器与@Trace装饰器需要配合使用，单独使用@ObservedV2装饰器或@Trace装饰器没有任何作用。
- 被@Trace装饰器装饰的属性property变化时，仅会通知property关联的组件进行刷新。
- 在嵌套类中，嵌套类中的属性property被@Trace装饰且嵌套类被@ObservedV2装饰时，才具有触发UI刷新的能力。
- 在继承类中，父类或子类中的属性property被@Trace装饰且该property所在类被@ObservedV2装饰时，才具有触发UI刷新的能力。
- 未被@Trace装饰的属性用在UI中无法感知到变化，也无法触发UI刷新。
- @ObservedV2的类实例目前不支持使用JSON.stringify进行序列化。

创建一个新的文件 ObservedV2Trace.ets，加入代码：

```
@ObservedV2
class Son {
  @Trace age: number = 100;
}
class Father {
  son: Son = new Son();
}
@Entry
```

```
@ComponentV2
struct Index {
  father: Father = new Father();

  build() {
    Column() {
      // 当点击改变 age 时，Text 组件会刷新
      Text(`${this.father.son.age}`)
        .onClick(() => {
          this.father.son.age++; // 第二层属性的变化
        })
    }
  }
}
```

此时，在 Previewer 上点击数字，可以看到数字会增长：

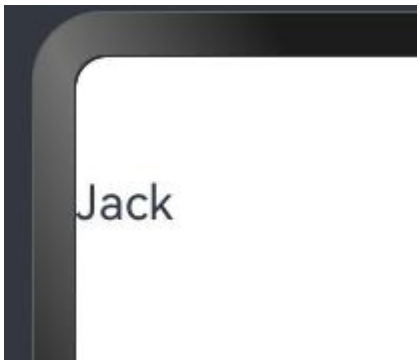


说明即便是第二层属性变化，也会被检测到，UI 会被刷新渲染。

将代码替换为：（如果同学们要保存之前的代码，可以创建新文件）

```
@ObservedV2
class Father {
  @Trace name: string = "Tom";
}
class Son extends Father {
}
@Entry
@ComponentV2
struct Index {
  son: Son = new Son();

  build() {
    Column() {
      // 当点击改变 name 时，Text 组件会刷新
      Text(`${this.son.name}`)
        .onClick(() => {
          this.son.name = "Jack";
        })
    }
  }
}
```

Son 继承了 Father 中的属性 name，而 name 是被@Trace 装饰的。点击 Tom，可以看到变为 Jack：



这个例子说明，在继承类中使用@Trace 装饰的属性具有被观测变化的能力。

代码替换为：

```
@ObservedV2
class Manager {
  @Trace static count: number = 1;
}
@Entry
@ComponentV2
struct Index {
  build() {
    Column() {
      // 当点击改变 count 时，Text 组件会刷新
      Text(`${Manager.count}`)
        .onClick(() => {
          Manager.count++;
```

```
        })
    }
  }
}
```

效果是点击数字会不断增加。这个例子说明，类中使用@Trace 装饰的静态属性具有被观测变化的能力。

代码替换为：

```
@ObservedV2
class Person {
  id: number = 0;
  @Trace age: number = 8;
}
@Entry
@ComponentV2
struct Index {
  person: Person = new Person();

  build() {
    Column() {
      // age 被@Trace 装饰，用在 UI 中可以触发 UI 刷新
      Text(`${this.person.age}`)
        .onClick(() => {
```

```
            this.person.age++; // 点击会触发 UI 刷新
        })
        // id 未被@Trace 装饰，用在 UI 中不会触发 UI 刷新
        Text(`${this.person.id}`) // 当 id 变化时不会刷新
          .onClick(() => {
            this.person.id++; // 点击不会触发 UI 刷新
        })
    }
  }
}
```

效果是点击 age 数字会增加，而点击 id，UI 没有变化。这个列子说明，对于非@Trace 装饰的成员属性，用在 UI 上无法触发 UI 刷新。

有一些注意事项：

- @ObservedV2 仅能装饰 class，无法装饰自定义组件。
- @Trace 不能用在没有被@ObservedV2 装饰的 class 上。
- @Trace 是 class 中属性的装饰器，不能用在 struct 中。
- @ObservedV2、@Trace 不能与@Observed、@Track 混合使用。
- 使用@ObservedV2 与@Trace 装饰的类不能和@State 等 V1 的装饰器混合使用，编译时报错。
- 继承自@ObservedV2 的类无法和@State 等 V1 的装饰器混用，运行时报错。

再看一个@Trace 装饰对象数组的例子：

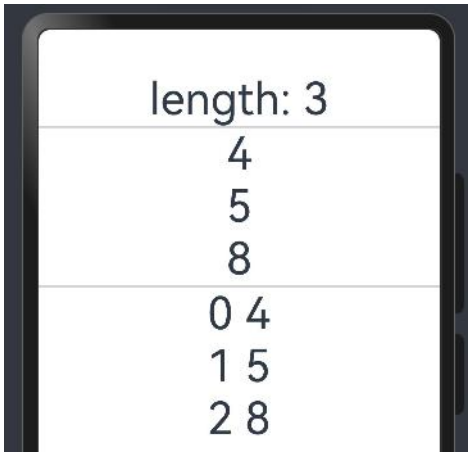
```
let nextId: number = 0;
```

```
@ObservedV2
class Person {
  @Trace age: number = 0;

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

@ObservedV2
class Info {
  id: number = 0;
  @Trace personList: Person[] = [];

  constructor() {
    this.id = nextId++;
    this.personList = [new Person(0), new Person(1), new Person(2)];
  }
}

@Entry
@ComponentV2
```

```
struct Index {
  info: Info = new Info();

  build() {
    Column() {
      Text(`length: ${this.info.personList.length}`)
        .fontSize(40)
      Divider().strokeWidth(2)
      if (this.info.personList.length >= 3) {
        Text(`${this.info.personList[0].age}`)
          .fontSize(40)
          .onClick(() => {
            this.info.personList[0].age++;
          })

        Text(`${this.info.personList[1].age}`)
          .fontSize(40)
          .onClick(() => {
            this.info.personList[1].age++;
          })

        Text(`${this.info.personList[2].age}`)
```

```
            .fontSize(40)
            .onClick(() => {
              this.info.personList[2].age++;
            })
        }

        Divider().strokeWidth(2)

        ForEach(this.info.personList, (item: Person, index: number) => {
          Text(`${index} ${item.age}`)
            .fontSize(40)
        })
    }
  }
}
```

效果：点击中间的 age 数字会增加，同时底下的每一项中的 age 同步增加：



@Trace 装饰对象数组 personList 以及 Person 类中的 age 属性，因此当 personList、age 改变时均可以观测到变化。

4. @Local 装饰器：组件内部状态

为了实现对@ComponentV2 装饰的自定义组件中变量变化的观测，开发者可以使用@Local 装饰器装饰变量。

概述

@Local表示组件内部的状态，使得自定义组件内部的变量具有观测变化的能力：

- 被@Local装饰的变量无法从外部初始化，因此必须在组件内部进行初始化。
- 当被@Local装饰的变量变化时，会刷新使用该变量的组件。
- @Local支持观测number、boolean、string、Object、class等基本类型以及Array、Set、Map、Date等内嵌类型。
- @Local的观测能力仅限于被装饰的变量本身。当装饰简单类型时，能够观测到对变量的赋值；当装饰对象类型时，仅能观测到对对象整体的赋值；当装饰数组类型时，能观测到数组整体以及数组元素项的变化；当装饰Array、Set、Map、Date等内嵌类型时，可以观测到通过API调用带来的变化。详见观察变化。
- @Local支持null、undefined以及联合类型。

使用@Local 装饰的变量具有被观测变化的能力。当装饰的变量发生变化时，会触发该变量绑定的 UI 组件刷新。

创建一个新的文件 LocalDemo.ets，加入代码：

```
@Entry
@ComponentV2
struct Index {
  @Local count: number = 0;
  @Local message: string = "Hello";
  @Local flag: boolean = false;
```

```
  build() {
    Column({space: 10}) {
      Text(`${this.count}`)
      Text(`${this.message}`)
      Text(`${this.flag}`)
      Button("change Local")
        .onClick(()=>{
          // 当@Local 装饰简单类型时，能够观测到对变量的赋值
          this.count++;
          this.message = this.message.split('').reverse().join('');
          this.flag = !this.flag;
        })
    }
  }
}
```

当装饰的变量类型为 boolean、string、number 时，可以观察到对变量赋值的变化。

当装饰的变量类型为类对象时，仅可以观察到对类对象整体赋值的变化，无法直接观察到对类成员属性赋值的变化，对类成员属性的观察依赖@ObservedV2 和@Trace 装饰器。注意，@Local 无法和@Observed 装饰的类实例对象混用。

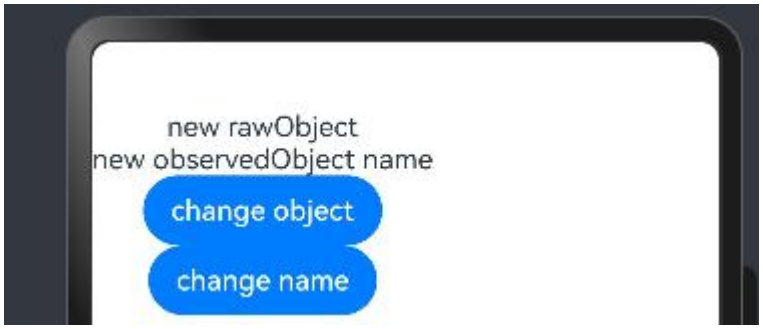
代码替换为：

```
class RawObject {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
@ObservedV2
class ObservedObject {
  @Trace name: string;
  constructor(name: string) {
    this.name = name;
  }
}
@Entry
@ComponentV2
struct Index {
  @Local rawObject: RawObject = new RawObject("rawObject");
```

```
    @Local observedObject: ObservedObject = new ObservedObject("observedObject");

    build() {
      Column() {
        Text(`${this.rawObject.name}`)
        Text(`${this.observedObject.name}`)
        Button("change object")
          .onClick(() => {
            // 对类对象整体的修改均能观察到
            this.rawObject = new RawObject("new rawObject");
            this.observedObject = new ObservedObject("new observedObject");
          })
        Button("change name")
          .onClick(() => {
            // @Local 不具备观察类对象属性的能力，因此对 rawObject.name 的修改无法观察到
            this.rawObject.name = "new rawObject name";
            // 由于 ObservedObject 的 name 属性被 @Trace 装饰，因此对 observedObject.name 的修改能被观察到
            this.observedObject.name = "new observedObject name";
          })
```

```
    }
  }
}
```

效果，点击 change object，可以看到两个名称都变化了。如果点击 change name，可以看到 rawObject 没有变化，而 observedObject 有变化。



@Local 装饰器存在以下使用限制：

- @Local 装饰器只能在@ComponentV2 装饰的自定义组件中使用。
- @Local 装饰的变量表示组件内部状态，不允许从外部传入初始化。

比如，下面的代码编译会报错：

```
@ComponentV2
struct ChildComponent {
  @Local message: string = "Hello World";
  build() {
  }
}
@ComponentV2
struct MyComponent {
```

```
  build() {
    ChildComponent({ message: "Hello" }) // 错误用法，编译时报错
  }
}
```

@Local与@State对比

@Local与@State的用法、功能对比如下：

	@State	@Local
参数	无。	无。
从父组件初始化	可选。	不允许外部初始化。
观察能力	能观测变量本身以及一层的成员属性，无法深度观测。	能观测变量本身，深度观测依赖@Trace装饰器。
数据传递	可以作为数据源和子组件中状态变量同步。	可以作为数据源和子组件中状态变量同步。

5. @Param：组件外部输入

为了增强子组件接受外部参数输入的能力，开发者可以使用@Param 装饰器。@Param 不仅可以接受组件外部输入，还可以接受@Local 的同步变化。

概述

@Param表示组件从外部传入的状态，使得父子组件之间的数据能够进行同步：

- @Param装饰的变量支持本地初始化，但是不允许在组件内部直接修改变量本身。
- 被@Param装饰的变量能够在初始化自定义组件时从外部传入，当数据源也是状态变量时，数据源的修改会同步给@Param。
- @Param可以接受任意类型的数据源，包括普通变量、状态变量、常量、函数返回值等。
- @Param装饰的变量变化时，会刷新该变量关联的组件。
- @Param支持观测number、boolean、string、Object、class等基本类型以及Array、Set、Map、Date等内嵌类型。
- 对于复杂类型如类对象，@Param会接受数据源的引用。在组件内可以修改类对象中的属性，该修改会同步到数据源。
- @Param的观测能力仅限于被装饰的变量本身。当装饰简单类型时，对变量的整体改变能够观测到；当装饰对象类型时，仅能观测对象整体的改变；当装饰数组类型时，能观测到数组整体以及数组元素项的改变；当装饰Array、Set、Map、Date等内嵌类型时，可以观测到通过API调用带来的变化。详见观察变化。
- @Param支持null、undefined以及联合类型。

装饰器说明

@Param变量装饰器	说明
装饰器参数	无。
能否本地修改	否，修改值需使用@Event装饰器的能力。
同步类型	由父到子单向同步。
允许装饰的变量类型	Object、class、string、number、boolean、enum等基本类型以及Array、Date、Map、Set等内嵌类型。支持null、undefined以及联合类型。
被装饰变量的初始值	允许本地初始化，若不在本地初始化，则需要和@Require装饰器一起使用，要求必须从外部传入初始化。

变量传递

传递规则	说明
从父组件初始化	@Param装饰的变量允许本地初始化，若无本地初始化则必须从外部传入初始化。当同时存在本地初始值与外部传入值时，会优先使用外部传入值进行初始化。
初始化子组件	@Param装饰的变量可以初始化子组件中@Param装饰的变量。
同步	@Param可以和父组件传入的状态变量数据源（即@Local或@Param装饰的变量）进行同步，当数据源发生变化时，会将修改同步给子组件的@Param。

使用@Param 装饰的变量具有被观测变化的能力。当装饰的变量发生变化时，会触发该变量绑定的 UI 组件刷新。

关于@Require 装饰器：

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

@Require 是校验@Prop、@State、@Provide、@BuilderParam、@Param 和普通变量(无状态装饰器修饰的变量)是否需要构造传参的一个装饰器。

当@Require 装饰器和@Prop、@State、@Provide、@Param、@BuilderParam、普通变量(无状态装饰器修饰的变量)结合使用时，在构造该自定义组件时，@Prop、@State、@Provide、@Param、@BuilderParam 和普通变量(无状态装饰器修饰的变量)必须在构造时传参。

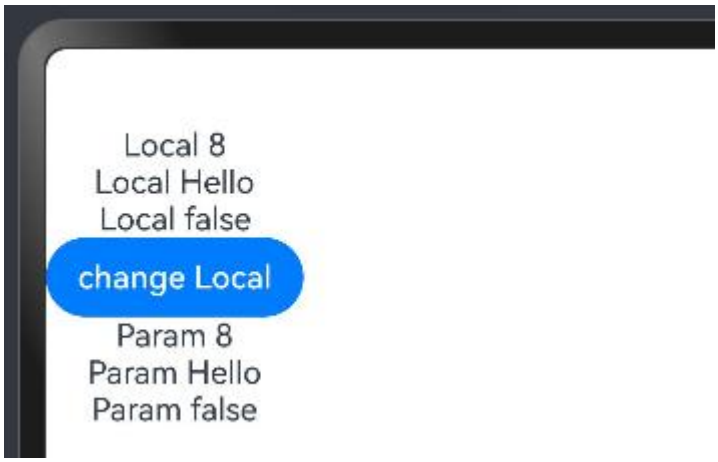
创建一个新的文件 ParamDemo.ets，代码：

```
@Entry
@ComponentV2
struct Index {
  @Local count: number = 0;
```

```
  @Local message: string = "Hello";
  @Local flag: boolean = false;
  build() {
    Column() {
      Text(`Local ${this.count}`)
      Text(`Local ${this.message}`)
      Text(`Local ${this.flag}`)
      Button("change Local")
        .onClick(()=>{
          // 对数据源的更改会同步给子组件
          this.count++;
          this.message = this.message.split('').reverse().join('');
          this.flag = !this.flag;
      })
      Child({
        count: this.count,
        message: this.message,
        flag: this.flag
      })
    }
  }
}
```

```
@ComponentV2
struct Child {
  @Require @Param count: number;
  @Require @Param message: string;
  @Require @Param flag: boolean;
  build() {
    Column() {
      Text(`Param ${this.count}`)
      Text(`Param ${this.message}`)
      Text(`Param ${this.flag}`)
    }
  }
}
```

运行效果：



这说明，当装饰的变量类型为 boolean、string、number 类型时，可以观察来自数据源同步的变化。

那么，当装饰的变量类型为类对象时，会怎么样呢？代码改为：

```
class RawObject {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
@ObservedV2
class ObservedObject {
  @Trace name: string;
  constructor(name: string) {
    this.name = name;
  }
}
@Entry
@ComponentV2
struct Index {
  @Local rawObject: RawObject = new RawObject("rawObject");
  @Local observedObject: ObservedObject = new ObservedObject("observedObject");
  build() {
    Column() {
```

```
        Text(`${this.rawObject.name}`)
        Text(`${this.observedObject.name}`)
        Button("change object")
          .onClick(() => {
            // 对类对象整体的修改均能观察到
            this.rawObject = new RawObject("new rawObject");
            this.observedObject = new ObservedObject("new observedObject");
        })
        Button("change name")
          .onClick(() => {
            // @Local 与@Param 均不具备观察类对象属性的能力，因此对rawObject.name 的修改无法观察到
            this.rawObject.name = "new rawObject name";
            // 由于 ObservedObject 的 name 属性被@Trace 装饰，因此对observedObject.name 的修改能被观察到
            this.observedObject.name = "new observedObject name";
        })
        Child({
          rawObject: this.rawObject,
          observedObject: this.observedObject
        })
```

```
    }
  }
}
@ComponentV2
struct Child {
  @Require @Param rawObject: RawObject;
  @Require @Param observedObject: ObservedObject;
  build() {
    Column() {
      Text(`${this.rawObject.name}`)
      Text(`${this.observedObject.name}`)
    }
  }

}
```

效果：

这说明，当装饰的变量类型为类对象时，仅可以观察到对类对象整体赋值的变化，无法直接观察到对类成员属性赋值的变化，对类成员属性的观察依赖@ObservedV2 和@Trace 装饰器。

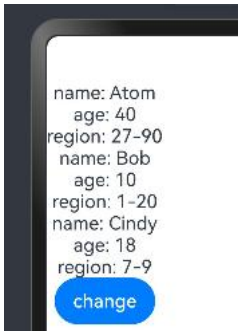
@Param 能够接受父组件@Local 或@Param 传递的数据并与之变化同步。代码：

```
@ObservedV2
class Region {
  @Trace x: number;
  @Trace y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}
@ObservedV2
class Info {
  @Trace name: string;
  @Trace age: number;
  @Trace region: Region;
  constructor(name: string, age: number, x: number, y: number) {
    this.name = name;
    this.age = age;
    this.region = new Region(x, y);
  }
}
@Entry
```

```
@ComponentV2
struct Index {
  @Local infoList: Info[] = [new Info("Alice", 8, 0, 0), new Info("Barry", 10, 1, 20),
new Info("Cindy", 18, 24, 40)];
  build() {
    Column() {
      ForEach(this.infoList, (info: Info) => {
        MiddleComponent({ info: info })
      })
      Button("change")
        .onClick(() => {
          this.infoList[0] = new Info("Atom", 40, 27, 90);
          this.infoList[1].name = "Bob";
          this.infoList[2].region = new Region(7, 9);
        })
    }
  }
}
@ComponentV2
struct MiddleComponent {
  @Require @Param info: Info;
  build() {
```

```
      Column() {
        Text(`name: ${this.info.name}`)
        Text(`age: ${this.info.age}`)
        SubComponent({ region: this.info.region })
      }
    }
}
@ComponentV2
struct SubComponent {
  @Require @Param region: Region;
  build() {
      Column() {
        Text(`region: ${this.region.x}-${this.region.y}`)
      }
    }
}
```

效果：



可以看到，子组件的数据的变化。

6. @Once：初始化同步一次

为了实现仅从外部初始化一次、不接受后续同步变化的能力，开发者可以使用@Once 装饰器搭配@Param 装饰器使用。

概述

@Once装饰器仅在变量初始化时接受外部传入值进行初始化，当后续数据源更改时，不会将修改同步给子组件：

- @Once必须搭配@Param使用，单独使用或搭配其他装饰器使用都是不允许的。
- @Once不影响@Param的观测能力，仅针对数据源的变化做拦截。
- @Once与@Param装饰变量的先后顺序不影响实际功能。
- @Once与@Param搭配使用时，可以在本地修改@Param变量的值。

装饰器使用规则说明

@Once装饰器作为辅助装饰器，本身没有对装饰类型的要求以及对变量的观察能力。

@Once变量装饰器	说明
装饰器参数	无。
使用条件	无法单独使用，必须配合@Param装饰器使用。

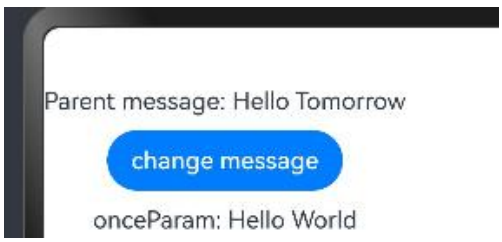
@Once 只能用在@ComponentV2 装饰的自定义组件中且仅能与@Param 搭配使用，@Once 与@Param 的先后顺序无关，可以写成@Param @Once 也可以写成@Once @Param。

创建新的文件 OnceDemo.ets，加入代码：

```
@ComponentV2
struct ChildComponent {
  @Param @Once onceParam: string = "";
  build() {
      Column() {
        Text(`onceParam: ${this.onceParam}`)
      }
```

```
  }
}
@Entry
@ComponentV2
struct MyComponent {
  @Local message: string = "Hello World";
  build() {
      Column() {
      Text(`Parent message: ${this.message}`)
      Button("change message")
        .onClick(() => {
          this.message = "Hello Tomorrow";
        })
      ChildComponent({ onceParam: this.message })
      }
  }
}
```

效果：



可以看到，@Once 修饰的变量仅初始化时同步数据源一次，之后不再继续同步变化。

当@Once 搭配@Param 使用时，可以解除@Param 无法在本地修改的限制，且修改能够触发 UI 刷新。此时，使用@Param @Once 相当于使用@Local，区别在于@Param @Once 能够接受外部传入初始化。

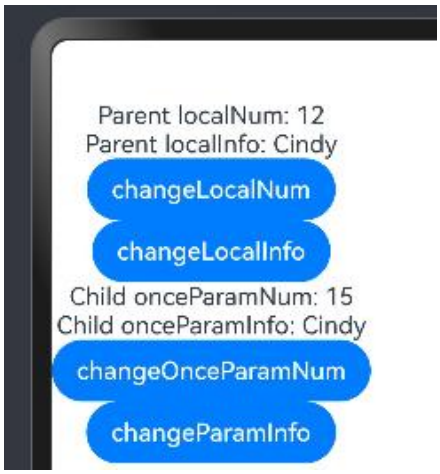
代码：

```
@ObservedV2
class Info {
  @Trace name: string;
  constructor(name: string) {
    this.name = name;
  }
}
@ComponentV2
struct Child {
  @Param @Once onceParamNum: number = 0;
  @Param @Once @Require onceParamInfo: Info;

  build() {
    Column() {
      Text(`Child onceParamNum: ${this.onceParamNum}`)
      Text(`Child onceParamInfo: ${this.onceParamInfo.name}`)
      Button("changeOnceParamNum")
        .onClick(() => {
```

```
            this.onceParamNum++;
        })
      Button("changeParamInfo")
        .onClick(() => {
            this.onceParamInfo = new Info("Cindy");
        })
    }
  }
}
@Entry
@ComponentV2
struct Index {
  @Local localNum: number = 10;
  @Local localInfo: Info = new Info("Tom");

  build() {
    Column() {
      Text(`Parent localNum: ${this.localNum}`)
      Text(`Parent localInfo: ${this.localInfo.name}`)
      Button("changeLocalNum")
        .onClick(() => {
            this.localNum++;
```

```
        })
      Button("changeLocalInfo")
        .onClick(() => {
          this.localInfo = new Info("Cindy");
        })
      Child({
        onceParamNum: this.localNum,
        onceParamInfo: this.localInfo
      })
    }
  }
}
```

效果：



可以看到，当@Once 搭配@Param 使用时，可以解除@Param 无法在本地修改的限制，且修改能够触发 UI 刷新。

7. @Event：规范组件输出

为了实现子组件向父组件要求更新@Param 装饰变量的能力，开发者可以使用@Event 装饰器。使用@Event 装饰回调方法是一种规范，表明子组件需要传入更新数据源的回调。所以，@Event 主要配合@Param 实现数据的双向同步。

由于@Param 装饰的变量在本地无法更改，使用@Event 装饰器装饰回调方法并调用，可以实现更改数据源的变量，再通过@Local 的同步机制，将修改同步回@Param，以此达到主动更新@Param 装饰变量的效果。

@Event 用于装饰组件对外输出的方法：

- @Event 装饰的回调方法中参数以及返回值由开发者决定。
- @Event 装饰非回调类型的变量不会生效。当@Event 没有初始化时，会自动生成一个空的函数作为默认回调。
- 当@Event 未被外部初始化，但本地有默认值时，会使用本地默认的函数进行处理。

@Param 标志着组件的输入，表明该变量受父组件影响，而@Event 标志着组件的输出，可以通过该方法影响父组件。使用@Event 装饰回调方法是一种规范，表明该回调作为自定义组件的输出。父组件需要判断是否提供对应方法用于子组件更改@Param 变量的数据源。

装饰器说明

@Event属性装饰器	说明
装饰器参数	无。
允许装饰的变量类型	回调方法，例如()=>void、(x:number)=>boolean等。回调方法是否含有参数以及返回值由开发者决定。
允许传入的函数类型	箭头函数。

使用@Event 可以更改父组件中变量，当该变量作为子组件@Param 变量的数据源时，该变化会同步回子组件的@Param 变量。

创建新文件 EventDemo.ets，加入代码：

```
@Entry
@ComponentV2
struct Index {
  @Local title: string = 'Title One';
  @Local fontColor: Color = Color.Red;

  build() {
    Column({space: 20}) {
      Text(`父组件中变量：${this.title}`)
        .fontColor(this.fontColor)
      Child({
        title: this.title,
        fontColor: this.fontColor,
        changeFactory: (type: number) => {
          if (type == 1) {
            this.title = 'Title One';
            this.fontColor = Color.Red;
          } else if (type == 2) {
            this.title = 'Title Two';
            this.fontColor = Color.Green;
          }
```

```
          }
        })
      }
    }
}

@ComponentV2
struct Child {
  @Param title: string = '';
  @Param fontColor: Color = Color.Black;
  @Event changeFactory: (x: number) => void = (x: number) => {};

  build() {
    Column({space: 10}) {
      Text(`子组件中变量：${this.title}`)
        .fontColor(this.fontColor)
      Button('change to Title Two')
        .onClick(() => {
          this.changeFactory(2);
        })
      Button('change to Title One')
        .onClick(() => {
```

```
            this.changeFactory(1);
        })
    }
  }
}
```

效果：

值得注意的是，使用@Event 修改父组件的值是立刻生效的，但从父组件将变化同步回子组件的过程是异步的，即在调用完@Event 的方法后，子组件内的值不会立刻变化。这是因为@Event 将子组件值实际的变化能力交由父组件处理，在父组件实际决定如何处理后，将最终值在渲染之前同步回子组件。代码：

```
@ComponentV2
struct Child {
  @Param index: number = 0;
  @Event changeIndex: (val: number) => void;

  build() {
    Column() {
```

```
        Text(`Child index: ${this.index}`)
          .onClick(() => {
            this.changeIndex(20);
            console.log(`after changeIndex ${this.index}`);
          })
      }
    }
}
@Entry
@ComponentV2
struct Index {
  @Local index: number = 0;

  build() {
        Column() {
          Child({
            index: this.index,
            changeIndex: (val: number) => {
              this.index = val;
            console.log(`in changeIndex ${this.index}`);
            }
          })
```

```
    }
  }
}
```

效果：

```
I      in changeIndex 20
I      after changeIndex 0
```

之后：

就是说顺序是这样的，先修改了父组件中的值，然后通过@Local 和子组件的@Param 再同步回子组件，然后渲染子组件。

8. @Provider 和@Consumer：跨组件层级双向同步

@Provider 和@Consumer 用于跨组件层级数据双向同步，可以使得开发者不用拘泥于组件层级。

@Provider，即数据提供方，其所有的子组件都可以通过@Consumer 绑定相同的 key 来获取@Provider 提供的数据。

@Consumer，即数据消费方，可以通过绑定同样的 key 获取其最近父节点的@Provider 的数据，当查找不到@Provider 的数据时，使用本地默认值。

@Provider 和@Consumer 装饰数据类型需要一致。开发者在使用@Provider 和@Consumer 时要注意：

- @Provider 和@Consumer 强依赖自定义组件层级，@Consumer 会因为所在组件的父组件

不同，而被初始化为不同的值。

- @Provider 和@Consumer 相当于把组件粘合在一起了，从组件独立角度，要减少使用@Provider 和@Consumer。

@Provider语法:

@Provider(alias?: string) varName : varType = initValue

@Provider属性装饰器	说明
装饰器参数	aliasName?: string，别名，缺省时默认为属性名。
支持类型	自定义组件中成员变量。属性的类型可以为number、string、boolean、class、Array、Date、Map、Set等类型。支持装饰箭头函数。
从父组件初始化	禁止。
本地初始化	必须本地初始化。
观察能力	能力等同于@Trace。变化会同步给对应的@Consumer。

@Consumer语法:

@Consumer(alias?: string) varName : varType = initValue

@Consumer属性装饰器	说明
装饰器参数	aliasName?: string，别名，缺省时默认为属性名，向上查找最近的@Provider。
可装饰的变量	自定义组件中成员变量。属性的类型可以为number、string、boolean、class、Array、Date、Map、Set等类型。支持装饰箭头函数。
从父组件初始化	禁止。
本地初始化	必须本地初始化。
观察能力	能力等同于@Trace。变化会同步给对应的@Provider。

@Provider 和@Consumer 可接受可选参数 aliasName，如果开发者没有配置参数，则使用属性名作为默认的 aliasName。注意：aliasName 是用于@Provider 和@Consumer 进行匹配的唯一指定 key。

我们读一下下面的几段代码：

```
@ComponentV2
struct Parent {
  // 未定义 aliasName，使用属性名'str'作为 aliasName
  @Provider() str: string = 'hello';
}

@ComponentV2
```

```
struct Child {
  // 定义 aliasName 为'str'，使用 aliasName 去寻找
  // 能够在 Parent 组件上找到，使用@Provider 的值'hello'
  @Consumer('str') str: string = 'world';
}
```

第二段：

```
@ComponentV2
struct Parent {
  // 定义 aliasName 为'alias'
  @Provider('alias') str: string = 'hello';
}

@ComponentV2 struct Child {
  // 定义 aliasName 为 'alias'，找到@Provider 并获得值'hello'
  @Consumer('alias') str: string = 'world';
}
```

第三段：

```
@ComponentV2
struct Parent {
  // 定义 aliasName 为'alias'
  @Provider('alias') str: string = 'hello';
```

```
}

@ComponentV2
struct Child {
  // 未定义 aliasName，使用属性名'str'作为 aliasName
  // 没有找到对应的@Provider，使用本地值'world'
  @Consumer() str: string = 'world';
}
```

我们先来看最重要的利用@Provider 和@Consumer 建立双向绑定，双向同步的例子。

创建新文件 ProviderConsumer.ets:

```
@Entry
@ComponentV2
struct Parent {
  @Provider() str: string = 'hello';

  build() {
    Column() {
      Button(this.str)
        .onClick(() => {
          this.str += '0';
        })
```

```
        Child()
      }
    }
}

@ComponentV2
struct Child {
  @Consumer() str: string = 'world';

  build() {
      Column() {
        Button(this.str)
          .onClick(() => {
            this.str += '0';
          })
      }
    }
}
```

上面的代码说明：

1. 自定义组件 Parent 和 Child 初始化：

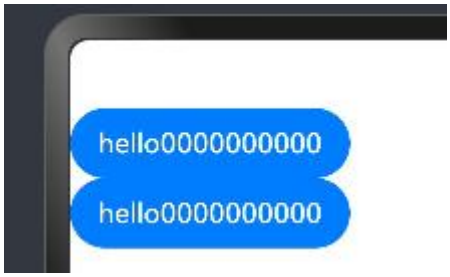
   • Child 中@Consumer() str: string = 'world'向上查找，查找到 Parent 中声明的 @Provider() str: string = 'hello'。

   • @Consumer() str: string = 'world'初始化为其查找到的@Provider 的值，即 'hello' 。

- 两者建立双向同步关系。

2. 点击 Parent 中的 Button，改变@Provider 装饰的 str，通知其对应的@Consumer，对应 UI 刷新。

3. 点击 Child 中 Button，改变@Consumer 装饰的 str，通知其对应的@Provider，对应 UI 刷新。



同学们可以自己尝试把父组件中的 str 改为另外的名称，比如 str1，看看效果。

当需要在父组件中向子组件注册回调函数时，可以通过使用@Provider 和@Consumer 装饰回调方法来解决。

```
@Entry
@ComponentV2
struct ParentPage {
  @Local message: string = "点击按钮更新我";

  // 通过@Provider 暴露状态更新方法
  @Provider() updateMessage: (newText: string) => void = (newText: string) => {
    this.message = newText;
  };
```

```
  build() {
    Column({ space: 20 }) {
      Text(this.message)
        .fontSize(20)
        .fontColor(Color.Blue)

      ChildComponent()
    }
    .width('100%')
    .height('100%')
    .padding(20)
  }
}

@ComponentV2
struct ChildComponent {
  // 通过@Consumer 接收父组件方法
  @Consumer() updateMessage: (newText: string) => void = (newText: string) => {};

  @Local btnText: string = "点击我";

  build() {
```

```
    Button(this.btnText)
      .width(150)
      .height(40)
      .onClick(() => {
        // 触发跨组件通信
        this.updateMessage(`子组件点击时间: ${new Date().toLocaleTimeString()}`);
        this.btnText = "已点击!";
        setTimeout(() => {
          this.btnText = "再次点击";
        }, 1000);
      })
  }
}
```

效果：

@Provider 可以在组件树上重名，@Consumer 会向上查找其最近父节点的@Provider 的数据。

```
@Entry
@ComponentV2
```

```
struct Index {
  @Provider() val: number = 10;

  build() {
    Column() {
      Parent()
    }
  }
}

@ComponentV2
struct Parent {
  @Provider() val: number = 20;
  @Consumer("val") val2: number = 0; // 10

  build() {
    Column() {
      Text(`${this.val2}`)
      Child()
    }
  }
}
```

```
@ComponentV2
struct Child {
  @Consumer() val: number = 0; // 20

  build() {
    Column() {
      Text(`${this.val}`)
    }
  }
}
```

效果：

到此，我们学习了状态管理 V2 版本中的一些重要的装饰器。另外还有几个比如@Monitor，@Computed 以及@Type，请同学自己到官网上学习。

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？