第3章

HarmonyOS 事件处理


课程思政 3

本章要点

- HarmonyOS 基于监听的事件处理
- HarmonyOS 线程管理
- HarmonyOS 线程间通信

本章知识结构图（见图 3-1）

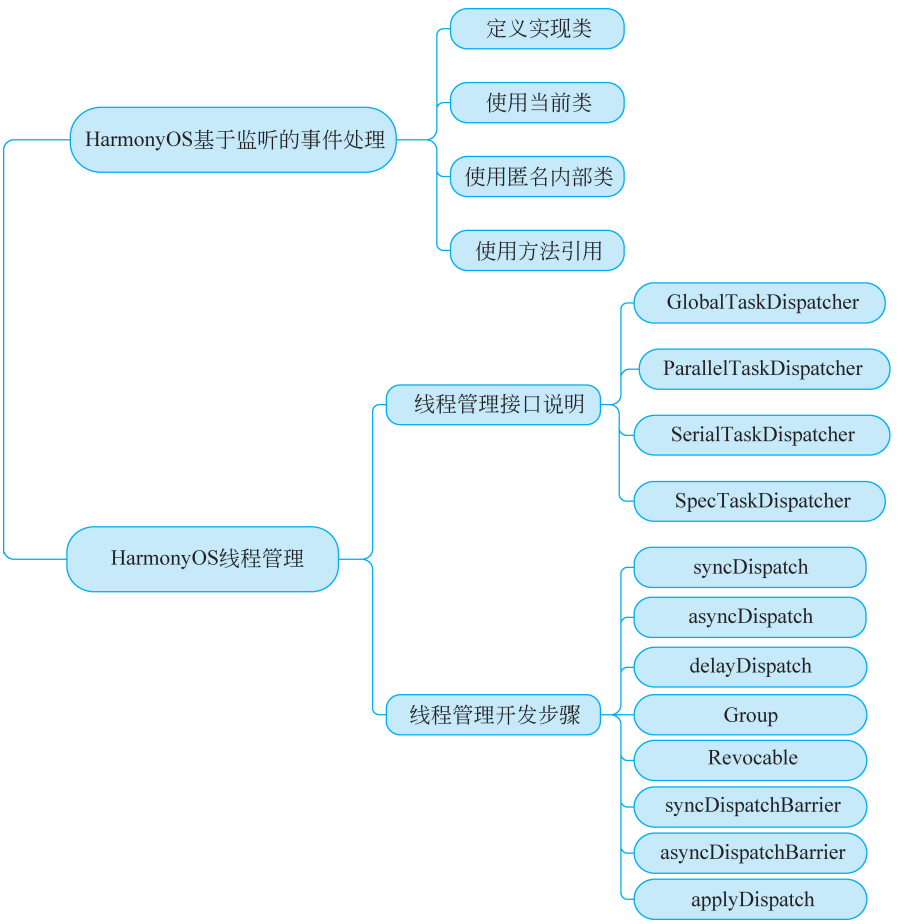

图 3-1　本章知识结构图

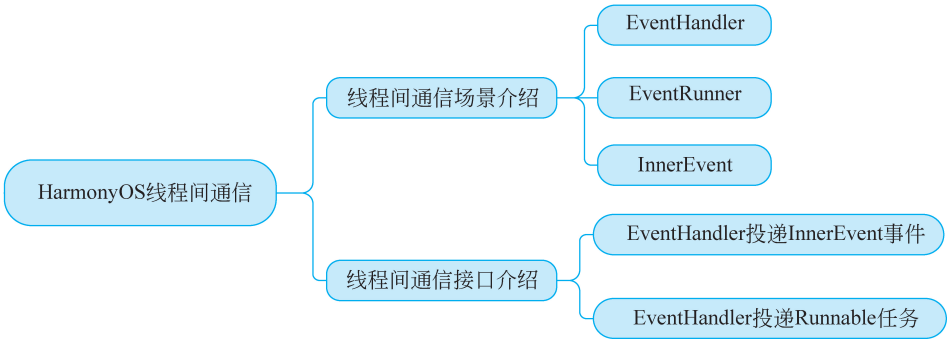

图　3-1（续）

本章示例（见图 3-2）

图 3-2　本章示例图

通过前面两章学习了 HarmonyOS 所提供的一些功能强大的界面控件，这些控件主要用来进行界面的设计与数据的显示，只能实现静态 UI，但是如果用户想与之进行动态交互，获取用户的意图信息，实现更加具体的功能，则还需要相应事件处理的辅助。当用户在程序界面上执行各种操作时，如单击一个按钮，此时一个体验良好的应用程序通常会做出相应的响应，这种响应就是通过事件处理来完成的。在前面章节的学习中，已经使用到了 HarmonyOS 中的一些事件处理，例如单击按钮实现数字加一。

在 HarmonyOS 中，主线程又可以叫作 UI 线程，用户界面属于主线程，默认情况下，所有的操作都在主线程上执行。如果需要执行比较耗时的任务，可创建其他线程（或子线程）来处理，否则主线程会因为耗时操作被阻塞，从而出现异常，所以需要不同的任务分发器来应对不同的操作情况。此外，还需要思考如何才能动态地显示用户界面。本章将介绍通过 EventHandler 消息传递来动态更新界面。

如果在事件处理中需要做一些比较耗时的操作，直接放在主线程中将会阻塞程序的

运行，给用户不好的体验，甚至程序会没有响应或强制退出。本章将学习通过同步派发任务以外的方式来处理耗时的操作。

学完本章之后，再结合前面所学知识，读者将可以开发出界面友好、人机交互良好的HarmonyOS 应用。

3.1 HarmonyOS 基于监听的事件处理

不管是什么手机应用，都离不开与用户的交互，只有通过用户的操作，才能知道用户的需求，从而实现具体的业务功能，因此，应用中经常需要处理的就是用户的操作，也就是需要为用户的操作提供响应，这种为用户操作提供响应的机制就是事件处理。

HarmonyOS 的基于监听的事件处理模型，与 Java 的 AWT、Swing 的处理方式几乎完全一样，只是相应的事件监听器和事件处理方法名有所不同。在基于监听的事件处理模型中，主要涉及以下 3 类对象。

- Eventsource(事件源)：即事件发生的源头，通常为某一控件，如图片、列表等。
- Event(事件)：用户具体某一操作的详细描述。事件封装了该操作的相关信息，如果程序需要获得事件源上所发生事件的相关信息，一般通过 Event 对象来取得，例如按键事件按下的是哪个键、触摸事件发生的位置等。
- Eventlistener(事件监听器)：负责监听用户在事件源上的操作，并对用户的各种操作做出相应的响应。事件监听器中可包含多个事件处理器，一个事件处理器实际上就是一个事件处理方法。

在基于监听的事件处理中，这 3 类对象又是如何协作的呢？实际上，基于监听的事件处理是一种委托式事件处理。事件源即普通控件将整个事件处理委托给特定的对象——事件监听器：当该事件源发生指定的事情时，系统自动生成事件对象，并通知所委托的事件监听器，由事件监听器相应的事件处理器处理这个事件。基于监听的事件处理模型如图 3-3 所示。

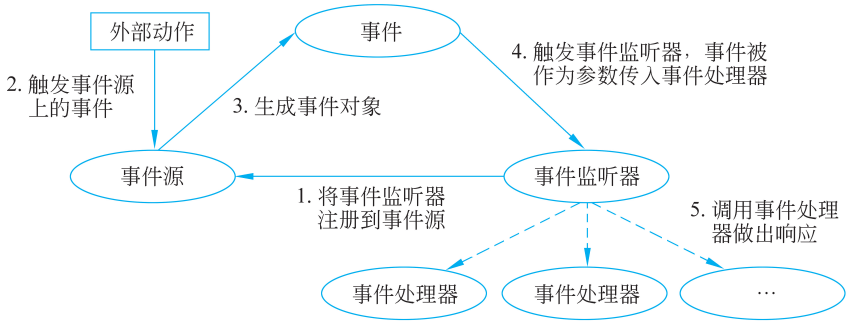

图 3-3　基于监听的事件处理模型

当用户在 HarmonyOS 控件上进行操作时，系统会自动生成事件对象，并将这个事件对象以参数的形式传给注册到事件源上的事件监听器，事件监听器调用相应的事件处理来处理。该过程类似于生活中我们每个人的能力都有限，当碰到一些自己处理不了的事情时，就

委托给某个机构处理。我们需要把所遇到的事情和要求描述清楚，这样，其他人才能比较好地解决问题，然后该机构会选派具体的员工来处理这件事。其中，我们自己就是事件源，遇到的事情就是事件，该机构就是事件监听器，具体解决事情的员工就是事件处理器。

单击事件的写法分别为定义实现类、使用当前类作为实现类、使用匿名内部类、使用方法引用四种，接下来通过按钮的单击事件来展示监听事件的处理。

1. 定义实现类

新建一个项目，在 layout 文件夹下打开 ability_main.xml，定义一个按钮，详细代码如程序清单 3-1 所示。

程序清单 3-1：hmos\ch03\01\Listener1\entry\src\main\java\resource\base\layout\ability_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<DirectionalLayout
    xmlns:ohos="http://schemas.huawei.com/res/ohos"
    ohos:height="match_parent"
    ohos:width="match_parent"
    ohos:alignment="center"
    ohos:orientation="vertical">
    <Button
        ohos:id="$+id:button1"
        ohos:height="match_content"
        ohos:width="match_content"
        ohos:text="自定义按钮"
        ohos:padding="15vp"
        ohos:text_size="100"
        ohos:background_element="#00ffff"/>
</DirectionalLayout>
```

定义实现类实现单击事件，在 MainAbilitySlice 类中设计一个实现 Component.ClickedListener 接口的 MyLister 类，降低代码耦合度，在该类中重写 onClick()方法从而满足单击事件需求，之后该类的方法可以在主类中的 onStart()方法中被调用，详细代码如程序清单 3-2 所示。

程序清单 3-2：hmos\ch03\01\Listener1\entry\src\main\java\com\example\listener1\slice\MainAbilitySlice.java

```
public class MainAbilitySlice extends AbilitySlice {
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        Button button = (Button) findComponentById(ResourceTable.Id_
button1);
```

```
7           button.setClickedListener((Component.ClickedListener) new
    MyListener());
8       }
9       class MyListener implements Component.ClickedListener {
10          @Override
11          public void onClick(Component component) {
12              new ToastDialog(getContext())
13                      .setText("按钮被单击了")
14                      .show();
15          }
16      }
17  }
```

单击按钮屏幕下方会显示一个对话框，效果如图 3-4 所示。

图 3-4　Button 单击示例图

2. 使用当前类作为实现类

使用当前类作为实现类实现单击事件，就是把上个例子中的 MyListener 类与主类合二为一，减少代码量，详细代码如程序清单 3-3 所示，实现效果与图 3-4 相同。

程序清单 3-3：hmos\ch03\01\Listener2\entry\src\main\java\com\example\listener2\slice\MainAbilitySlice.java

```
1   public class MainAbilitySlice extends AbilitySlice implements Component.
    ClickedListener{
2       @Override
```

```
3       public void onStart(Intent intent) {
4           super.onStart(intent);
5           super.setUIContent(ResourceTable.Layout_ability_main);
6           Button button = (Button) findComponentById(ResourceTable.Id_
    button1);
7           button.setClickedListener(this);
8       }
9       @Override
10      public void onClick(Component component) {
11          new ToastDialog(getContext())
12                  .setText("按钮被单击了")
13                  .show();
14      }
15  }
```

3. 使用匿名内部类

使用匿名内部类实现单击事件。匿名内部类，就是没有名字的一种嵌套类，使用匿名内部类的方式，可以无须创建新的类，减少代码冗余，详细代码如程序清单 3-4 所示，实现效果与图 3-4 相同。

程序清单 3-4：hmos\ch03\01\Listener3\entry\src\main\java\com\example\listener3\slice\MainAbilitySlice.java

```
1   public class MainAbilitySlice extends AbilitySlice {
2       @Override
3       public void onStart(Intent intent) {
4           super.onStart(intent);
5           super.setUIContent(ResourceTable.Layout_ability_main);
6           Button button = (Button) findComponentById(ResourceTable.Id_
    button1);
7           button.setClickedListener(new Component.ClickedListener() {
8               @Override
9               public void onClick(Component component) {
10                  new ToastDialog(getContext())
11                          .setText("按钮被单击了")
12                          .show();
13              }
14          });
15      }
16  }
```

4. 使用方法引用

使用方法引用实现单击事件，详细代码如程序清单 3-5 所示，实现效果与图 3-4 相同。

程序清单 3-5：hmos\ch03\01\Listener4\entry\src\main\java\com\example\listener4\slice\MainAbilitySlice.java

```
1  public class MainAbilitySlice extends AbilitySlice{
2      @Override
3      public void onStart(Intent intent) {
4          super.onStart(intent);
5          super.setUIContent(ResourceTable.Layout_ability_main);
6          Button button = (Button) findComponentById(ResourceTable.Id_
   button1);
7          button.setClickedListener(this::onClick);
8      }
9      public void onClick(Component component){
10         new ToastDialog(getContext())
11                 .setText("按钮被单击了")
12                 .show();
13     }
14 }
```

3.2 HarmonyOS 线程管理

进程是应用程序的执行实例，是程序的一次动态执行过程。动态过程是进程本身从产生、发展到最终消亡的过程。线程是比进程更小的执行单位，进程可进一步细化为线程，是一个程序内部的一条执行路径。

多线程是实现并发机制的一种有效手段，指一个进程在执行的过程中产生多个线程，这些线程可以同时存在。一般出现以下 3 种情况需要使用多线程：①程序需要同时执行两个或多个任务时；②程序需要实现一些需要等待的任务时，如用户输入、文件读写操作时；③需要一些后台运行的程序时。

当应用启动时，系统会创建一个主线程。该线程随着应用的动态变化而创建和消亡，是应用的核心线程。主线程负责处理大部分业务，负责应用和 UI 组件发生交互。所以，主线程又称为 UI 线程。因为主线程在任何时候都有较高的响应速度，默认情况下所有操作都在主线程上执行。如果需要执行比较耗时的操作，比如发起一条网络请求时，考虑到网速等原因，服务器未必会立刻响应请求，可创建其他子线程运行这些操作，否则主线程会因为耗时操作被阻塞，从而出现异常，从而影响用户对软件的正常使用。

3.2.1 线程管理接口说明

当应用的业务逻辑复杂时，需要处理多项任务，可能需要开发者创建多个线程来执行多个任务。基于这种情况，代码复杂，难以维护，任务与线程的交互更加繁杂，导致运行效率降低。为了解决这一问题，开发者可以使用 TaskDispatcher 分发不同的任务。

TaskDispatcher 是一个任务分发器，它是 Ability(Ability 在本书 4.1 节有详细介绍)分发任务的基本接口，采用了多线程技术，隐藏任务所在线程的实现细节，TaskDispatcher 获取到执行的任务和方法就可以自动创建合适的线程完成任务。

为保证应用有更好的响应性，需要设计任务的优先级。在 UI 线程上运行的任务默认以高优先级运行，如果某个任务无须等待结果，则可以用低优先级。任务优先级如表 3-1 所示。

表 3-1　任务优先级

优　先　级	详 细 描 述
HIGH	最高任务优先级，比默认优先级、低优先级的任务有更高的概率执行
DEFAULT	默认任务优先级，比低优先级的任务有更高的概率执行
LOW	低任务优先级，比高优先级、默认优先级的任务有更低的概率执行

TaskDispatcher 具有不同的实现，每种实现对应不同的任务分发器。在分发任务时可以指定任务的优先级，由同一个任务分发器分发出的任务具有相同的优先级。系统提供的任务分发器有 GlobalTaskDispatcher(全局并发任务分发器)、ParallelTaskDispatcher(并发任务分发器)、SerialTaskDispatcher(串行任务分发器)、SpecTaskDispatcher(专有任务分发器)。

- GlobalTaskDispatcher 由 Ability 执行 getGlobalTaskDispatcher(TaskPriority.priority)获取，中文含义为全局并发任务分发器，priority 参数代表任务的优先级，可取值为 HIGH、DEFAULT、LOW，全局并发任务分发器适用于任务没有联系的情况。一个应用只有一个 GlobalTaskDispatcher，它在程序结束时才被销毁，具体代码如程序清单 3-6 所示。

程序清单 3-6

```
1   TaskDispatcher globalTaskDispatcher = getGlobalTaskDispatcher
    (TaskPriority.DEFAULT);
```

- ParallelTaskDispatcher，它由 Ability 执行的 createParallelTaskDispatcher(String name,TaskPriority.priority)创建并返回，中文含义为并发任务分发器，其中 name 参数为任务分发器的名称，priority 参数代表任务的优先级，可取值为 HIGH、DEFAULT、LOW，与 GlobalTaskDispatcher 不同的是，ParallelTaskDispatcher 不具有全局唯一性，可创建多个。开发者在创建或销毁 dispatcher 时，需要持有对应的对象引用，具体代码如程序清单 3-7 所示。

程序清单 3-7

```
1   String dispatcherName = "parallelTaskDispatcher";
2   TaskDispatcher parallelTaskDispatcher = createParallelTaskDispatcher
    (dispatcherName, TaskPriority.DEFAULT);
```

- SerialTaskDispatcher 是由 Ability 执行的 createSerialTaskDispatcher(String name, TaskPriority.priority)创建并返回的,中文含义为串行任务分发器,由该分发器分发的所有任务都是按顺序执行的,但是执行这些任务的线程并不是固定的。如果要执行并行任务,应使用 ParallelTaskDispatcher 或者 GlobalTaskDispatcher,而不是创建多个 SerialTaskDispatcher。如果任务之间没有依赖,应使用 GlobalTaskDispatcher 来实现。它的创建和销毁由开发者自己管理,开发者在使用期间需要持有该对象引用,具体代码如程序清单 3-8 所示。

程序清单 3-8

```
1  String dispatcherName = "serialTaskDispatcher";
2  TaskDispatcher serialTaskDispatcher = createSerialTaskDispatcher
   (dispatcherName, TaskPriority.DEFAULT);
```

- SpecTaskDispatcher 的中文含义为专有任务分发器,它是绑定到专有线程上的任务分发器。目前,专有线程是主线程。UITaskDispatcher 和 MainTaskDispatcher 同属 SpecTaskDispatcher。官方更加推荐 UITaskDispatcher。
- UITaskDispatcher:绑定到应用主线程的专有任务分发器,由 getUITaskDispatcher() 创建并返回。由该分发器分发的所有任务都在主线程上按顺序执行,它在应用程序结束时被销毁,具体代码如程序清单 3-9 所示。

程序清单 3-9

```
1  TaskDispatcher uiTaskDispatcher = getUITaskDispatcher();
```

- MainTaskDispatcher:由 Ability 执行 getMainTaskDispatcher() 创建并返回,具体代码如程序清单 3-10 所示。

程序清单 3-10

```
1  TaskDispatcher mainTaskDispatcher= getMainTaskDispatcher();
```

3.2.2 线程管理开发步骤

1. syncDispatch(Runnable runnable)

同步派发任务:派发任务并在当前线程等待任务执行时完成。在返回前,当前线程会被阻塞。同步任务是那些没有被引擎挂起、在主线程上排队执行的任务。只有前一个任务执行完毕,才能执行后一个任务。使用时需实现 Runnable 接口的 run()方法,在该方法中定义需要执行的任务。

如程序清单 3-11 展示了如何使用 GlobalTaskDispatcher 派发同步任务。

程序清单 3-11：hmos\ch03\02\syncGlobalTaskDispatcher\entry\src\main\java\com\example\syncglobaltaskdispatcher \slice\MainAbilitySlice.java

```
1  static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201, "MY_TAG");
2   @Override
3   public void onStart(Intent intent) {
4       super.onStart(intent);
5       super.setUIContent(ResourceTable.Layout_ability_main);
6       TaskDispatcher globalTaskDispatcher = getGlobalTaskDispatcher
    (TaskPriority.DEFAULT);
7       globalTaskDispatcher.syncDispatch(new Runnable() {
8           @Override
9           public void run() {
10              HiLog.info(LABEL, "同步任务一启动");
11          }
12      });
13      HiLog.info(LABEL, "同步任务一结束");
14      globalTaskDispatcher.syncDispatch(new Runnable() {
15          @Override
16          public void run() {
17              HiLog.info(LABEL, "同步任务二启动");
18          }
19      });
20      HiLog.info(LABEL, "同步任务二结束");
21      globalTaskDispatcher.syncDispatch(new Runnable() {
22          @Override
23          public void run() {
24              HiLog.info(LABEL, "同步任务三启动");
25          }
26      });
27      HiLog.info(LABEL, "同步任务三结束");
28  }
29  //执行结果如下:
30  //同步任务一启动
31  //同步任务一结束
32  //同步任务二启动
33  //同步任务二结束
34  //同步任务三启动
35  //同步任务三结束
```

2. asyncDispatch(Runnable runnable)

异步派发任务：派发任务，并立即返回，返回值是一个可用于取消任务的接口，需要实现 Runnable 接口的 run()方法，在该方法中定义需要执行的任务。与同步派发任务不同的是，该任务不会阻塞后面代码的执行。

如下代码示例展示了如何使用 GlobalTaskDispatcher 派发异步任务，详细代码如程序清单 3-12 所示。

程序清单 3-12：hmos\ch03\02\asyncGlobalTaskDispatcher\entry\src\main\java\com\example\asyncGlobalTaskDispatcher \slice\MainAbilitySlice.java

```
1  public class MainAbilitySlice extends AbilitySlice {
2  static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201, "MY_TAG");
3      @Override
4      public void onStart(Intent intent) {
5          super.onStart(intent);
6          super.setUIContent(ResourceTable.Layout_ability_main);
7        TaskDispatcher globalTaskDispatcher = getGlobalTaskDispatcher
   (TaskPriority.DEFAULT);
8          globalTaskDispatcher.asyncDispatch(new Runnable() {
9              @Override
10             public void run() {
11                 HiLog.info(LABEL, "async task1 run");
12             }
13         });
14         HiLog.info(LABEL, "after async task1");
15     }
16 }
17 //执行结果如下:
18 //after async task1
19 //async task1 run
```

3. delayDispatch(Runnable runnable,long delay)

异步延迟派发任务：异步执行，函数立即返回，内部会在延时指定时间后将任务派发到相应队列中。该方法的第 1 个参数表示需执行的任务，第 2 个参数用于设置执行异步任务延迟的时间。延时时间参数仅代表在这段时间以后任务分发器会将任务加入队列中，任务的实际执行时间可能晚于这个时间。具体比这个数值晚多久，取决于队列及内部线程池的繁忙情况。

如下代码示例展示了如何使用 GlobalTaskDispatcher 延迟派发任务，详细代码如程序清单 3-13 所示。

程序清单 3-13：hmos\ch03\02\delayGlobalTaskDispatcher\entry\src\main\java\com\example\delayGlobalTaskDispatcher\slice\MainAbilitySlice.java

```
1  public class MainAbilitySlice extends AbilitySlice {
2      static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201,
   "MY_TAG");
3      @Override
4      public void onStart(Intent intent) {
5          super.onStart(intent);
6          super.setUIContent(ResourceTable.Layout_ability_main);
7          long callTime = System.currentTimeMillis();
```

```
        long delayTime = 500L;
      TaskDispatcher globalTaskDispatcher = getGlobalTaskDispatcher
(TaskPriority.DEFAULT);
        Revocable revocable = globalTaskDispatcher.delayDispatch(new
Runnable() {
            @Override
            public void run() {
                HiLog.info(LABEL, "delayDispatch task1 run");
                final long actualDelay = System.currentTimeMillis() - callTime;
                HiLog.info(LABEL, "actualDelayTime >= delayTime: %
{public}b", (actualDelay >= delayTime));
            }
        }, delayTime);
        HiLog.info(LABEL, " delayDispatch task1 finish");
    }
}
//执行结果可能如下:
//delayDispatch task1 finish
//delayDispatch task1 run
//actualDelayTime >= delayTime : true
```

4. Group

任务组：表示一组任务，且该组任务之间有一定的联系，由 TaskDispatcher 执行 createDispatchGroup 创建并返回。通过 asyncGroupDispatch(Group group, Runnable runnable)方法将任务加入分组中，通过 groupDispatchNotify(Group group, Runnable runnable)方法可以设置该分组中的所有任务执行完成后再执行指定任务。

如下代码示例展示了任务组的使用方式：将一系列相关联的任务放入一个任务组，执行完任务后关闭应用，详细代码如程序清单 3-14 所示。

程序清单 3-14：hmos\ch03\02\Group\entry\src\main\java\com\example\Group\slice\MainAbilitySlice.java

```
public class MainAbilitySlice extends AbilitySlice {
    static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201,
"MY_TAG");
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        String dispatcherName = "parallelTaskDispatcher";
            TaskDispatcher dispatcher = createParallelTaskDispatcher
(dispatcherName, TaskPriority.DEFAULT);
//创建一个 group 线程任务组
        Group group = dispatcher.createDispatchGroup();
```

```
11  //将任务 1 加入线程任务组中,可以返回一个用于取消线程任务的接口
12          dispatcher.asyncGroupDispatch(group, new Runnable() {
13              @Override
14              public void run() {
15                  HiLog.info(LABEL, " task1 is running");
16              }
17          });
18  //将与任务 1 相关联的任务 2 加入任务组
19          dispatcher.asyncGroupDispatch(group, new Runnable() {
20              @Override
21              public void run() {
22                  HiLog.info(LABEL, " task2 is running");
23              }
24          });
25  //在任务组中的所有任务执行完成后执行指定任务
26          dispatcher.groupDispatchNotify(group, new Runnable() {
27              @Override
28              public void run() {
29  HiLog.info(LABEL, "the close task is running after all tasks in the group
    are completed");
30              }
31          });
32  //执行结果可能如下:
33  //task1 is running
34  //task2 is running
35  //the close task is running after all tasks in the group are completed
36  //另外一种可能的执行结果:
37  //task2 is running
38  //task1 is running
39  //the close task is running after all tasks in the group are completed
40      }
41  }
```

5. Revocable

取消任务：Revocable 是取消一个异步任务的接口。异步任务包括通过 asyncDispatch、delayDispatch、asyncGroupDispatch 派发的任务。如果任务已经在执行中或已经执行完成，则会返回取消失败的信息。

以下示例展示了如何取消一个异步延时任务，详细代码如程序清单 3-15 所示。

程序清单 3-15：hmos\ch03\02\Revocable\entry\src\main\java\com\example\Revocable\slice\MainAbilitySlice.java

```
1   public class MainAbilitySlice extends AbilitySlice {
2       static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201,
    "MY_TAG");
```

```
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        TaskDispatcher dispatcher = getUITaskDispatcher();
        Revocable revocable = dispatcher.delayDispatch(new Runnable() {
            @Override
            public void run() {
        HiLog.info(LABEL, "delay dispatch");
            }
        }, 10);
        boolean revoked = revocable.revoke();
        HiLog.info(LABEL, "%{public}b", revoked);
//一种可能的结果如下 :
//true
    }
}
```

6. syncDispatchBarrier

同步设置屏障任务：在任务组上设立任务执行屏障，同步等待任务组中的所有任务执行完成，再执行指定任务。

以下示例展示了如何同步设置屏障，详细代码如程序清单 3-16 所示。

程序清单 3-16：hmos\ch03\02\syncDispatchBarrier\entry\src\main\java\com\example\syncDispatchBarrier\slice\MainAbilitySlice.java

```
public class MainAbilitySlice extends AbilitySlice {
  static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201,
  "MY_TAG");
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        String dispatcherName = "parallelTaskDispatcher";
        TaskDispatcher dispatcher = createParallelTaskDispatcher
(dispatcherName, TaskPriority.DEFAULT);
//创建任务组
        Group group = dispatcher.createDispatchGroup();
//将任务加入任务组,返回一个用于取消任务的接口
        dispatcher.asyncGroupDispatch(group, new Runnable() {
            @Override
            public void run() {
                HiLog.info(LABEL, "task1 is running");       //1
```

```
            }
        });
        dispatcher.asyncGroupDispatch(group, new Runnable() {
            @Override
            public void run() {
                HiLog.info(LABEL, "task2 is running");        //2
            }
        });
        dispatcher.syncDispatchBarrier(new Runnable() {
            @Override
            public void run() {
                HiLog.info(LABEL, "barrier");                 //3
            }
        });
        HiLog.info(LABEL, "after syncDispatchBarrier");       //4
//1 和 2 的执行顺序不定;3 和 4 总是在 1 和 2 之后按顺序执行
//可能的执行结果:
//task1 is running
//task2 is running
//barrier
//after syncDispatchBarrier

//另外一种执行结果:
//task2 is running
//task1 is running
//barrier
//after syncDispatchBarrier
    }
}
```

7. asyncDispatchBarrier

异步设置屏障任务：在任务组上设立任务执行屏障后直接返回，指定任务将在任务组中的所有任务执行完成后再执行。

以下示例展示了如何异步设置屏障，详细代码如程序清单 3-17 所示。

程序清单 3-17：hmos\ch03\02\asyncDispatchBarrier\entry\src\main\java\com\example\asyncDispatchBarrier\slice\MainAbilitySlice.java

```
public class MainAbilitySlice extends AbilitySlice {
    static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201,
"MY_TAG");
    @Override
    public void onStart(Intent intent) {
```

```
5           super.onStart(intent);
6           super.setUIContent(ResourceTable.Layout_ability_main);
7           TaskDispatcher dispatcher = createParallelTaskDispatcher
    ("dispatcherName", TaskPriority.DEFAULT);
8   //创建任务组
9           Group group = dispatcher.createDispatchGroup();
10  //将任务加入任务组,返回一个用于取消任务的接口
11          dispatcher.asyncGroupDispatch(group, new Runnable() {
12              @Override
13              public void run() {
14                  HiLog.info(LABEL, "task1 is running");       //1
15              }
16          });
17          dispatcher.asyncGroupDispatch(group, new Runnable() {
18              @Override
19              public void run() {
20                  HiLog.info(LABEL, "task2 is running");       //2
21              }
22          });
23
24          dispatcher.asyncDispatchBarrier(new Runnable() {
25              @Override
26              public void run() {
27                  HiLog.info(LABEL, "barrier");                //3
28              }
29          });
30          HiLog.info(LABEL, "after asyncDispatchBarrier");    //4
31  //1 和 2 的执行顺序不定,但总在 3 之前执行;4 不需要等待 1、2、3 执行完成
32  //可能的执行结果:
33  //task1 is running
34  //task2 is running
35  //after asyncDispatchBarrier
36  //barrier
37      }
38  }
```

8. applyDispatch

执行多次任务：对指定任务执行多次。以下示例展示了如何执行多次任务，详细代码如程序清单 3-18 所示。

程序清单 3-18：hmos\ch03\02\applyDispatch\entry\src\main\java\com\example\applyDispatch\slice\MainAbilitySlice.java

```
1   public class MainAbilitySlice extends AbilitySlice {
```

```
    static final HiLogLabel LABEL = new HiLogLabel(HiLog.LOG_APP, 0x00201,
"MY_TAG");
    @Override
    public void onStart(Intent intent) {
        super.onStart(intent);
        super.setUIContent(ResourceTable.Layout_ability_main);
        final int total = 10;
        final CountDownLatch latch = new CountDownLatch(total);
        final List<Long> indexList = new ArrayList<>(total);
        TaskDispatcher dispatcher = getGlobalTaskDispatcher
(TaskPriority.DEFAULT);
//执行任务 total 次
        dispatcher.applyDispatch((index) -> {
            indexList.add(index);
            latch.countDown();
        }, total);
//设置任务超时
        try {
            latch.await();
        } catch (InterruptedException exception) {
            HiLog.error(LABEL, "latch exception");
        }
        HiLog.info(LABEL, "list size matches, %{public}b", (total ==
indexList.size()));
//执行结果:
//list size matches, true
    }
}
```

3.3 HarmonyOS 线程间通信

3.3.1 线程间通信场景介绍

在开发过程中,经常需要在当前线程中处理耗时的操作,比如联网读取数据,或者读取本地较大的一个文件,不能把这些操作放在主线程中,因为放在主线程中,界面会出现假死现象,但是又不希望当前的线程受到阻塞,所以会在子线程中执行耗时操作。并且UI 控件不是线程安全的,所以系统不允许子线程更新 UI,之后提交请求将子线程的数据传递给主线程,让主线程做 UI 更新。此时就可以使用 EventHandler 机制。

EventHandler 是 HarmonyOS 用于处理线程间通信的一种机制,可以在多个线程并发更新 UI 的同时保证线程安全,可以通过 EventRunner 创建新线程,将耗时的操作放到新线程上执行。在不阻塞原来的线程基础下,合理地处理任务。当熟悉了 EventHandler

的原理之后我们知道，EventHandler 不仅能将子线程的数据传递给主线程，它还能实现任意两个线程的数据传递。

EventRunner 是一种事件循环器，循环从该 EventRunner 创建的新线程的事件队列中获取 InnerEvent 事件或者 Runnable 任务。InnerEvent 是 EventHandler 投递的事件。

EventHandler 是一种用户在当前线程上投递 InnerEvent 事件或者 Runnable 任务到异步线程上处理的机制。每一个 EventHandler 和指定的 EventRunner 所创建的新线程绑定，并且该新线程内部有一个事件队列。EventHandler 可以投递指定的 InnerEvent 事件或 Runnable 任务到这个事件队列。EventRunner 从事件队列里循环地取出事件，如果取出的事件是 InnerEvent 事件，将在 EventRunner 所在线程执行 processEvent 回调；如果取出的事件是 Runnable 任务，将在 EventRunner 所在线程执行 Runnable 的 run 回调。一般地，EventHandler 主要有两个作用：①在不同线程间分发和处理 InnerEvent 事件或 Runnable 任务；②延迟处理 InnerEvent 事件或 Runnable 任务。EventHandler 的运作机制如图 3-5 所示。

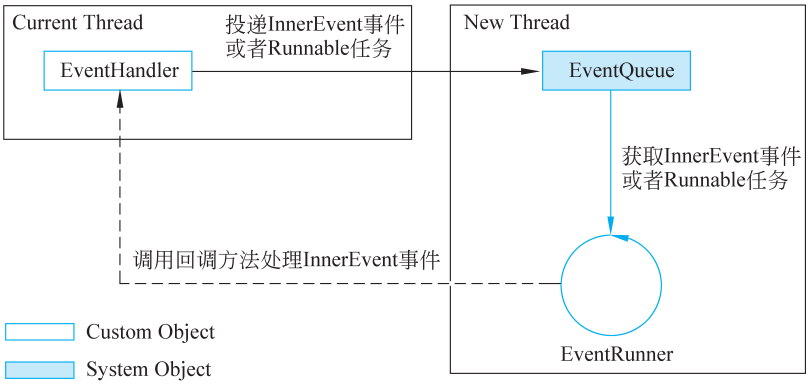

图 3-5　EventHandler 的运作机制

EventHandler 实现线程间通信的主要流程如下。

首先，EventHandler 投递具体的 InnerEvent 事件或者 Runnable 任务到 EventRunner 所创建的线程的事件队列。然后，EventRunner 循环从事件队列中获取 InnerEvent 事件或 Runnable 任务。最后，处理事件或任务，EventRunner 取出事件为 InnerEvent 事件，则触发 EventHandler 的回调方法并触发 EventHandler 的处理方法，在新线程上处理该事件；如果 EventRunner 取出的事件为 Runnable 任务，则 EventRunner 直接在新线程上处理 Runnable 任务。注意：在进行线程间通信的时候，EventHandler 只能和 EventRunner 所创建的线程进行绑定，EventRunner 创建时需要判断是否创建成功，获取的 EventRunner 非空时，才可以使用 EventHandler 绑定 EventRunner。一个 EventHandler 只能同时与一个 EventRunner 绑定，一个 EventRunner 上可以创建多个 EventHandler。

EventHandler 的主要功能是将 InnerEvent 事件或者 Runnable 任务投递到其他线程进行处理，其使用场景包括：

(1) 开发者需要将 InnerEvent 事件投递到新的线程,按照优先级和延时进行处理。投递时,EventHandler 的优先级可在 IMMEDIATE、HIGH、LOW、IDLE 中选择,并设置合适的 delayTime。

(2) 开发者需要将 Runnable 任务投递到新的线程,并按照优先级和延时进行处理。投递时,EventHandler 的优先级可在 IMMEDIATE、HIGH、LOW、IDLE 中选择,并设置合适的 delayTime。

(3) 开发者需要在新创建的线程里投递事件到原线程进行处理。

EventRunner 的工作模式可以分为托管模式和手动模式。两种模式是在调用 EventRunner 的 create()方法时,通过选择不同的参数来实现的,详见 API 参考。默认为托管模式。

托管模式:不需要开发者调用 run()和 stop()方法启动和停止 EventRunner。当 EventRunner 实例化时,系统调用 run()启动 EventRunner;当 EventRunner 不被引用时,系统调用 stop()停止 EventRunner。

手动模式:需要开发者自行调用 EventRunner 的 run()方法和 stop()方法确保线程的启动和停止。

3.3.2 线程间通信接口介绍

EventHandler 的属性 Priority(优先级)介绍:EventRunner 将根据优先级的高低从事件队列中获取事件或者 Runnable 任务进行处理。

EventHandler 的属性如表 3-2 所示。

表 3-2 EventHandler 的属性

属 性	详 细 描 述
Priority.IMMEDIATE	表示事件被立即投递
Priority.HIGH	表示事件先于 LOW 优先级投递
Priority.LOW	表示事件先于 IDLE 优先级投递,事件的默认优先级是 LOW
Priority.IDLE	表示在没有其他事件的情况下才投递该事件

EventHandler 的主要接口如表 3-3 所示。

表 3-3 EventHandler 的主要接口

接 口 名	详 细 描 述
EventHandler(EventRunner runner)	利用已有的 EventRunner 创建 EventHandler
current()	在 processEvent 回调中获取当前的 EventHandler
processEvent(InnerEvent event)	回调处理事件,由开发者实现
sendEvent(InnerEvent event, long delayTime)	发送一个延时事件到事件队列,优先级为 LOW
sendEvent(InnerEvent event, long delayTime, EventHandler.Priority priority)	发送一个指定优先级的延时事件到事件队列

续表

接　口　名	详 细 描 述
postSyncTask（Runnable task，EventHandler.Priority priority）	发送一个指定优先级的 Runnable 同步任务到事件队列，延时为 0ms
postTask（Runnable task，long delayTime，EventHandler.Priority priority）	发送一个指定优先级的 Runnable 延时任务到事件队列
sendTimingEvent（InnerEvent event，long taskTime，EventHandler.Priority priority）	发送一个带优先级的事件到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，则立即执行
postTimingTask（Runnable task，long taskTime，EventHandler.Priority priority）	发送一个带优先级的 Runnable 任务到队列，在 taskTime 时间执行，如果 taskTime 小于当前时间，则立即执行
removeEvent(int eventId，long param，Object object）	删除指定 id、param 和 object 的事件

EventRunner 的主要接口如表 3-4 所示。

表 3-4　EventRunner 的主要接口

接　口　名	详 细 描 述
create()	创建一个拥有新线程的 EventRunner
create(booleaninNewThread)	创建一个拥有新线程的 EventRunner，参数为 true 时，EventRunner 为托管模式，系统自动管 EventRunner；参数为 false 时，EventRunner 为手动模式
create(String newThreadName)	创建一个拥有新线程的 EventRunner，新线程的名字是 newThreadName
current()	获取当前线程的 EventRunner
run()	EventRunner 为手动模式时，调用该方法启动新的线程
stop()	EventRunner 为手动模式时，调用该方法停止新的线程

InnerEvent 的主要接口如表 3-5 所示。

表 3-5　InnerEvent 的主要接口

接　口　名	详 细 描 述
drop()	释放一个事件实例
get()	获得一个事件实例
get(inteventId)	获得一个指定的 eventId 的事件实例
get(inteventId，long param)	获得一个指定的 eventId 和 param 的事件实例
get(inteventId，long param，Object object)	获得一个指定的 eventId、param 和 object 的事件实例
get(inteventId，Object object)	获得一个指定的 eventId 和 object 的事件实例
PacMap getPacMap()	获取 PacMap，如果没有，就新建一个