实验二十二　ArkTS 并发

一、实验目的

1. 了解 DevEco Studio 的使用
2. 学习 ArkTS 语言及并发机制
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

并发是指在同一时间内，存在多个任务同时执行的情况。对于多核设备，这些任务可能同时在不同 CPU 上并行执行。对于单核设备，多个并发任务不会在同一时刻并行执行，但是 CPU 会在某个任务休眠或进行 I/O 操作等状态下切换任务，调度执行其他任务，提升 CPU 的资源利用率。

为了提升应用的响应速度与帧率，避免耗时任务对主线程的影响，ArkTS 提供了**异步并发**和**多线程并发**两种处理策略：

- **异步并发**是指异步代码在执行到一定程度后会被暂停，以便在未来某个时间点继续执行，这种情况下，同一时间只有一段代码在执行。ArkTS 通过 **Promise** 和 **async/await** 提供异步并发能力，适用于**单次 I/O** 任务的开发场景。Promise 和 async/await 提供异步并发能力，是标准的 JS 异步语法。异步代码会被挂起并在之后继续执行，同一时间只有一段代码执行，适用于单次 I/O 任务的场景开发，例如一次网络请求、一次文件读写等操作。无需另外启动线程执行。异步语法是一种编程语言的特性，允许程序在执行某些操作时不必等待其完成，而是可以继续执行其他操作。

- **多线程并发**允许在同一时间段内同时执行多段代码。在主线程继续响应用户操作和更新 UI 的同时，后台线程也能执行耗时操作，从而避免应用出现卡顿。ArkTS 通过 **TaskPool** 和 **Worker** 提供多线程并发能力，适用于耗时任务等并发场景。

并发多线程场景下，不同并发线程间需要进行数据通信，不同类别对象的传输方式存在差异，包括拷贝或内存共享等。

并发能力在多种场景中都有应用，其中包括异步并发任务、耗时任务（CPU 密集型任务、I/O 密集型任务和同步任务等）、长时任务、常驻任务等。开发者可以根据不同的任务诉求和场景，选择相应的并发策略进行优化和开发，也可以具体查看应用多线程开发实践案例。

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

设置项目名称为 ConCurrencyDemo。

2. Promise

- 基本的使用

创建一个新的文件 PromiseDemo.ets，创建基本的页面 Component，然后在@Entry 前面加入代码：

```
let p: Promise<string> = new Promise((resolve: Function, reject: Function) => {
  let a = 1 + 1
  if(a==2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})
```

```
p.then((message: string) => {
  console.log('This is the promise fulfilled: ' + message)
}).catch((message: string) => {
  console.log('This is the promise rejected: ' + message)
})
```

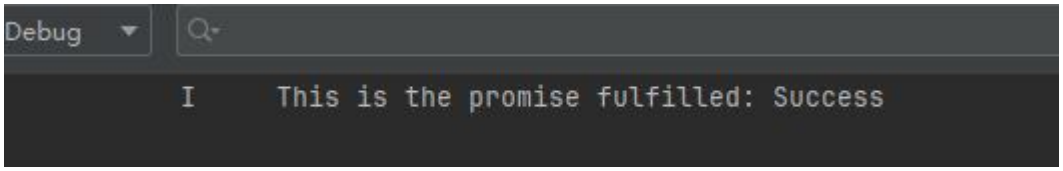
Promise 其实从字面意思去理解，就是“承诺”的意思，也就是说承诺要做一件事情，那么结果可能就是达成了承诺(fulfilled)或者食言了，失败了(rejected, failed)。相应地，如果 fulfilled，就调用函数 resolve，也就是解决了的意思；如果失败了，就调用 rejected。

使用 Promise 的核心就是调用.then()和.catch()函数，因为 Promise 是个异步的（承诺通常都是过段时间再看结果），.then()就是说如果承诺达成了，就如何如何；相应地如果没有达成，失败了，就调用.catch，然后做相应的处理。then 方法的回调函数接收 Promise 对象的成功结果作为参数，并将其输出到控制台上。如果 Promise 对象进入 rejected 状态，则 catch 方法的回调函数接收错误对象作为参数，并将其输出到控制台上。

上面的代码，红色字体的那一行：

```
let a = 1 + 1
```

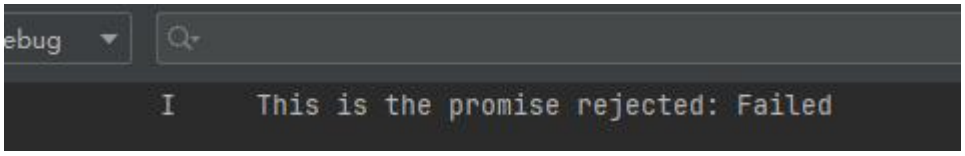
此时 a 的值为 2，a==2 满足，所以承诺达成，此时 Log 中显示：



如果改为其他的值，比如改为：

```
let a = 2 + 1
```

运行，查看 Log：



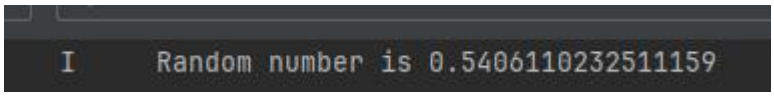
再看一个例子，注释掉前面的代码，添加代码：

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


const promise: Promise<number> = new Promise((resolve: Function, reject: Function) => {
  setTimeout(() => {
    const randomNumber: number = Math.random();
    if (randomNumber > 0.5) {
      resolve(randomNumber);
    } else {
      reject(new Error(`Random number is too small as it is ${randomNumber}`));
    }
  }, 1000);
})


promise.then((result: number) => {
  console.info(`Random number is ${result}`);
}).catch((error: BusinessError) => {
  console.error(error.message);
});
```

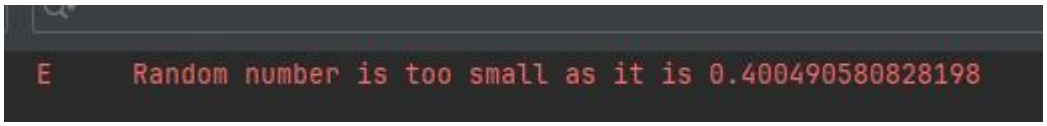
这里红色的代码含义是，随机生成一个 0 到 1 之间的数，如果这个数大于 0.5，认为 Promise 达成，

否则就认为是失败。

执行代码，如果随机数大于 0.5：



如果小于 0.5：



- Promise.then()顺序执行，Promise.all()以及 Promise.race()

添加代码，创建三个新的 Promise:

```
const actionPromise1: Promise<string> = new Promise((resolve: Function, reject: Function)
=> {
  setTimeout(() => {
    resolve('Action 1 fulfilled after 1 second!')
  }, 1000);
})

const actionPromise2: Promise<string> = new Promise((resolve: Function, reject: Function)
=> {
  setTimeout(() => {
    resolve('Action 2 fulfilled after 5 seconds!')
  }, 5000);
})
```

```
const actionPromise3: Promise<string> = new Promise((resolve: Function, reject: Function)
=> {
  setTimeout(() => {
    resolve('Action 3 fulfilled after 10 seconds!')
  }, 10000);
})
```

可以看到，我们使用了 setTimeout 这个来模拟“异步”，因为这个函数的第二个参数是时间，就是说等待这个时长（毫秒数）之后再执行前面的箭头函数。

添加代码：

```
actionPromise1.then((message: string)=> {
  console.log('Action 1: ' + message)
  actionPromise2.then((message: string) => {
    console.log('Action 2: ' + message)
    actionPromise3.then((message: string) => {
      console.log('Action 3: ' + message)
    })
  })
})
```

这里可以看到.then()函数的顺序执行，也就是说第一个 Promise 达成了之后，再执行第二个，然后再执行第三个。查看 Log：

可以看到分别延迟一段时间后，顺序执行这几个 Promise。注意 Previewer 的时间可能不够准确，但是应该能够看到效果了。这个例子并不是很好，在下一节有个例子更合适理解。

注释掉上面这段代码，添加代码：

```
Promise.all([
  actionPromise1,
  actionPromise2,
  actionPromise3
]).then((messages: string[]) => {
  console.log(messages.toString())
})
```

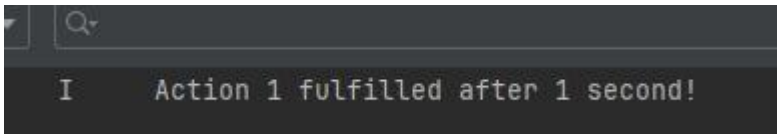
Promise.all()函数是执行所有的 Promise，放在一个数组中，需要等所有的 Promise 达成了才会执行.then，而且这里是把所有的消息都打印出来了：

```
I     Action 1 fulfilled after 1 second!,Action 2 fulfilled after 5 seconds!,Action 3 fulfilled after 10 seconds!
```

注释掉这段.all()代码，添加代码：

```
Promise.race([
  actionPromise1,
  actionPromise2,
  actionPromise3
]).then((message: string) => {
  console.log(message)
})
```

Promise.race()是“race”也就是竞争的意思，就是数组中的 Promise 哪个先达成就先跑对应的 resolve 结果，这里因为三个 Promise 分别是 1 秒，5 秒和 10 秒，所以总是第一个胜出：



Promise 看起来比较复杂，难以理解。它的出现的主要作用是取代 callback，回调函数。好处是可以避免出现所谓的 callback hell。我们的目标是首先能看懂并理解 Promise，然后在实际的开发中灵活运用。由于后来出现了 async/await，在大多数的情况下，使用 async/await 更好，更容易理解，但是还是有很多的场景下会用到 Promise，所以需要理解其含义。

3. async/await

async/await 是一种用于处理异步操作的 Promise 语法糖，使得编写异步代码变得更加简单和易读。通过使用 async 关键字声明一个函数为异步函数，并使用 await 关键字等待 Promise 的解析（完成或拒绝），**以同步的方式**编写异步操作的代码。

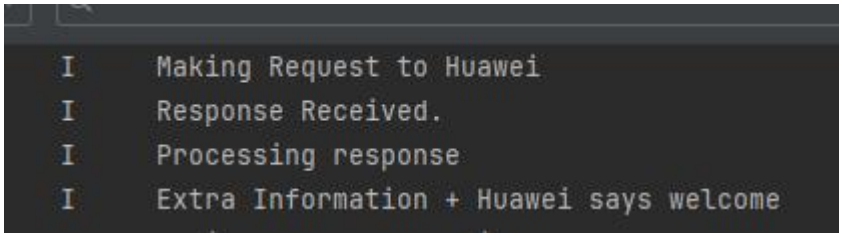
async 函数是一个返回 Promise 对象的函数，用于表示一个异步操作。在 async 函数内部，可以使用 await 关键字等待一个 Promise 对象的解析，并返回其解析值。如果一个 async 函数抛出异常，那么该函数返回的 Promise 对象将被拒绝，并且异常信息会被传递给 Promise 对象的 onRejected() 方法。

创建一个新的文件 AsyncAwaitDemo.ets，添加基本的页面 Component，在@Entry 前面添加代码：

```
function makeRequest(location: string) {
  return new Promise<string>((resolve: Function, reject: Function) => {
    console.log(`Making Request to ${location}`)
    if(location == 'Huawei') {
      resolve('Huawei says welcome')
    } else {
      reject('We can only talk to Huawei')
```

```
    }
  })
}

function processRequest(response: string) {
  return new Promise<string>((resolve: Function, reject: Function) => {
    console.log('Processing response')
    resolve(`Extra Information + ${response}`)
  })
}

makeRequest('Huawei').then((response: string): Promise<string> => {
  console.log('Response Received.');
  return processRequest(response);
}).then((processedResponse: string)=> {
  console.log(processedResponse)
}).catch((err: string)=>{
  console.log(err)
})
```

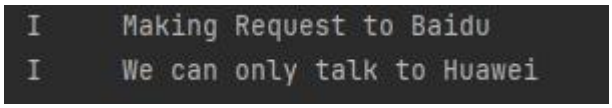
可以看到，这里我们用了两个函数，makeRequest 函数返回一个 Promise，只有当参数为‘Huawei’的时候，Promise 是达成，此时才会执行.then()里面的代码，这时会输出‘Response Received’，然后再执行 processRequest，参数是收到的 response，即“Huawei says welcome”，然后因为 processRequest 只有 resolve，所以会输出 Extra Information：



如果我们改一下参数，改为’Baidu’：

```
makeRequest('Baidu').then((response: string): Promise<string> => {
```

由于第一个 Promise 没有达成，直接走.catch()那里了，所以 Log 显示：



也就是说，processRequest 不会被执行。

前面说了，async/await 是处理异步操作的 Promise 语法糖。我们来尝试把上面的代码功能改为 async/await 的方式。先注释掉这段：

```
// makeRequest('Baidu').then((response: string): Promise<string> => {
//     console.log('Response Received.');
//     return processRequest(response);
// }).then((processedResponse: string)=> {
//     console.log(processedResponse)
// }).catch((err: string)=>{
//     console.log(err)
// })
```

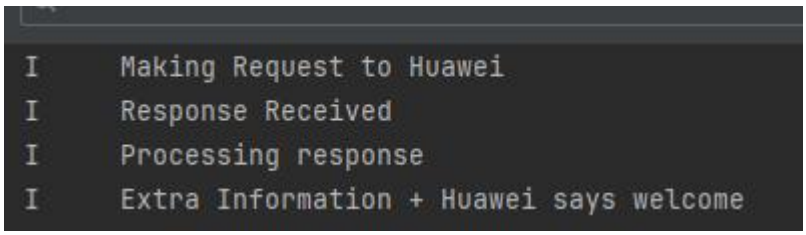
然后添加代码：

```
async function doWork() {
  const response = await makeRequest('Huawei')
  console.log('Response Received')
  const processedResponse = await processRequest(response)
  console.log(processedResponse)
}
```

```
doWork()
```

这里在 doWork()函数前面加上了 async，表明这个函数内部有异步执行的内容。里面需要异步执行的函数前面都加上了 await，表示我们要等待这个函数执行的结果然后才往下继续。从英文单词的含义上我们也可以尝试去理解。Async 表示 asynchronization，也就是异步的意思，await 就是“等候”的意思，就是说这个时候再往下执行之前，我们要等到这个结果才行。

此时，执行的结果：

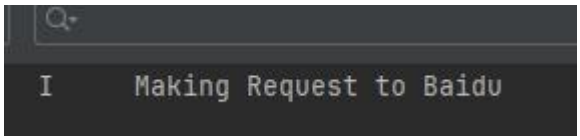


此时问题来了，如果上面的代码中，makeRequest()里面的参数不是‘Huawei’会怎么样？

```
async function doWork() {
  const response = await makeRequest('Baidu')
  console.log('Response Received')
  const processedResponse = await processRequest(response)
  console.log(processedResponse)
}
```

```
doWork()
```

比如上面我们改为‘Baidu’，此时：

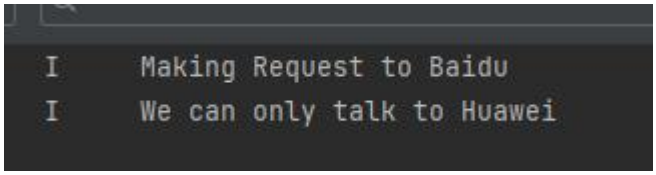


而后面的都不会显示。Previewer 也显示不出来。

对应错误的处理，我们可以采用 try...catch 对，把代码改为：

```
async function doWork() {
  try {
    const response = await makeRequest('Baidu')
    console.log('Response Received')
    const processedResponse = await processRequest(response)
    console.log(processedResponse)
  } catch(e) {
    console.log(e)
  }
}

doWork()
```

此时可以看到，reject 的情况也被正确处理了：



可以看到，async/await 的好处是代码看上去像是在写同步的代码。注意，async/await 需要成对的出现，通常是用函数的方式来实现。

再看一个例子，把这个 AsyncAwaitDemo.ets 中的所有代码注释掉，加入代码：

```
async function myAsyncFunction(): Promise<string> {
  const result: string = await new Promise((resolve: Function) => {
    setTimeout(() => {
      resolve('Hello, world!');
    }, 3000);
  });
  console.info(result); // 输出：  Hello, world!
  return result
}


@Entry
@Component
struct Index {
  @State message: string = 'Hello World';
  build() {
    Row() {
      Column() {
```

```
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          .onClick(async () => {
            let res = await myAsyncFunction();
            console.info("res is: " + res);
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

关键是看红色字体的代码。也就是说，在调用我们定义的 async 函数的时候，也要用到 await，加在前面，相应的箭头函数前面也要加上 async。

4. 多线程并发

并发模型是用来实现不同应用场景中并发任务的编程模型，常见的并发模型分为**基于内存共享的并发模型**和**基于消息通信的并发模型**。

内存共享并发模型指多线程同时执行任务，这些线程**依赖同一内存并且都有权限访问**，线程访问内存前**需要抢占并锁定内存的使用权**，没有抢占到内存的线程需要等待其他线程释放使用权再执行。

Actor 并发模型每一个线程都是一个独立 Actor，每个 Actor 有自己**独立的内存，Actor 之间通过消息传递机制触发对方 Actor 的行为**，不同 Actor 之间不能直接访问对方的内存空间。

Actor 并发模型对比内存共享并发模型的优势在于**不同线程间内存隔离**，不会产生不同线程竞争同一内存资源的问题。开发者不需要考虑对内存上锁导致的一系列功能、性能问题，提升了开发效率。由于 Actor 并发模型线程之间不共享内存，需要通过线程间通信机制传输并发任务和任务结果。

当前 ArkTS 提供了 TaskPool 和 Worker 两种并发能力，TaskPool 和 Worker 都**基于 Actor 并发模型**实现。

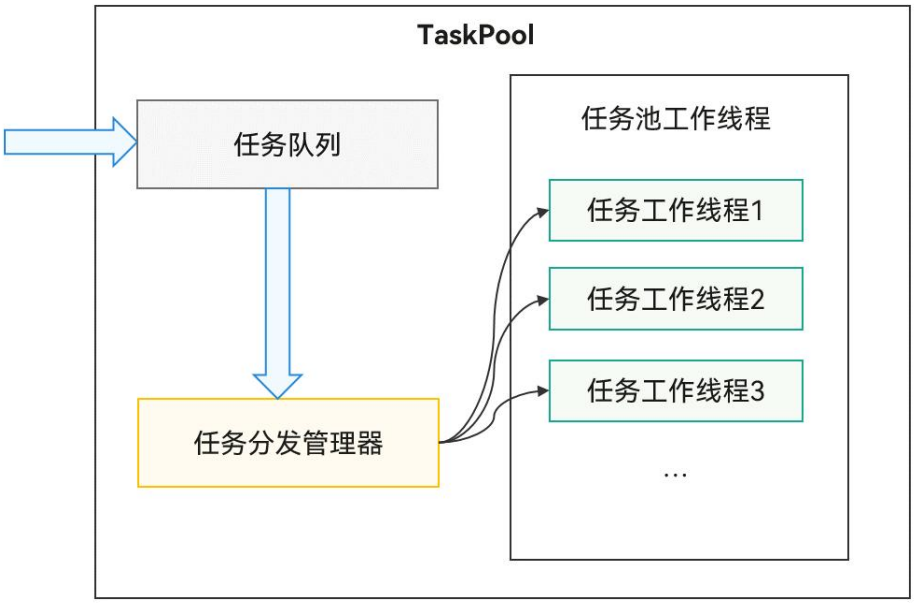
- **TaskPool 任务池**

官方文档：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/taskpool-introduction#taskpool%E8%BF%90%E4%BD%9C%E6%9C%BA%E5%88%B6

任务池（TaskPool）作用是为应用程序提供一个多线程的运行环境，降低整体资源的消耗、提高系统的整体性能，且您无需关心线程实例的生命周期。

TaskPool 运作机制示意图：



TaskPool 支持开发者在主线程封装任务抛给任务队列，系统选择合适的工作线程，进行任务的分发及执行，再将结果返回给主线程。接口直观易用，支持任务的执行、取消，以及指定优先级的能力，同时通过系统统一线程管理，结合动态调度及负载均衡算法，可以节约系统资源。系统默认会启动一个任务工作线程，当任务较多时会扩容，工作线程数量上限跟当前设备的物理核数相关，具体数

量内部管理，保证最优的调度及执行效率，长时间没有任务分发时会缩容，减少工作线程数量。

TaskPool注意事项

- 实现任务的函数需要使用@Concurrent装饰器标注，且仅支持在.ets文件中使用。
- 从API version 11开始，跨并发实例传递带方法的实例对象时，该类必须使用装饰器@Sendable装饰器标注，且仅支持在.ets文件中使用。
- 任务函数在TaskPool工作线程的执行耗时不能超过3分钟（不包含Promise和async/await异步调用的耗时，例如网络下载、文件读写等I/O任务的耗时），否则会被强制退出。
- 实现任务的函数入参需满足序列化支持的类型，详情请参见线程间通信对象。
- ArrayBuffer参数在TaskPool中默认转移，需要设置转移列表的话可通过接口setTransferList()设置。
- 由于不同线程中上下文对象是不同的，因此TaskPool工作线程只能使用线程安全的库，例如UI相关的非线程安全库不能使用。
- 序列化传输的数据量大小限制为16MB。
- Priority的IDLE优先级是用来标记需要在后台运行的耗时任务（例如数据同步、备份），它的优先级别是最低的。这种优先级标记的任务只会在所有线程都空闲的情况下触发执行，并且只会占用一个线程来执行。
- Promise不支持跨线程传递，如果TaskPool返回pending或rejected状态的Promise，会返回失败；对于fulfilled状态的Promise，TaskPool会解析返回的结果，如果结果可以跨线程传递，则返回成功。
- 不支持在TaskPool工作线程中使用AppStorage。

@Concurrent装饰器

在使用TaskPool时，执行的并发函数需要使用该装饰器修饰，否则无法通过相关校验。

说明
从API version 9开始，支持使用@Concurrent装饰器声明并校验并发函数。

装饰器说明

@Concurrent并发装饰器	说明
装饰器参数	无。
使用场景	仅支持在Stage模型的工程中使用。仅支持在.ets文件中使用。
装饰的函数类型	允许标注async函数或普通函数。禁止标注generator、箭头函数、method。不支持类成员函数或者匿名函数。
装饰的函数内的变量类型	允许使用local变量、入参和通过import引入的变量。禁止使用闭包变量。
装饰的函数内的返回值类型	支持的类型请查线程间通信对象。

基本使用

先看一个基本使用的例子，将 Index.ets 中的代码全部替换为：

```
import { taskpool } from '@kit.ArkTS';
```

```
@Concurrent
function add(num1: number, num2: number): number {
  return num1 + num2;
}

async function ConcurrentFunc(): Promise<void> {
  try {
    let task: taskpool.Task = new taskpool.Task(add, 1, 2);
    console.info("taskpool res is: " + await taskpool.execute(task));
  } catch (e) {
    console.error("taskpool execute error is: " + e);
  }
}

@Entry
@Component
struct Index {
  @State message: string = 'Hello World'

  build() {
    Row() {
```

```
        Column() {
          Text(this.message)
            .fontSize(50)
            .fontWeight(FontWeight.Bold)
            .onClick(() => {
              ConcurrentFunc();
            })
        }
        .width('100%')
      }
      .height('100%')
    }
}
```

注意这里红色字体的部分：

```
import { taskpool } from '@kit.ArkTS';
```

从@kit.ArkTS 中导入 taskpool。

@Concurrent 放在函数前面作为装饰器，用来声明并校验并发函数。

```
 let task: taskpool.Task = new taskpool.Task(add, 1, 2);
```

是创建一个或者说是在任务池中使用一个任务，而这个任务是执行函数 add，参数是 1,2。

```
await taskpool.execute(task))
```

通过调用 taskpool.execute 来执行 task，而且这个是一个异步函数，需要 await 等待执行的结果。

程序需要在模拟器上运行才能看到效果，点击“Hello World”，查看 Log：

```
[(:576)(ExecuteTask)] taskpool:: Task Allocation: taskId : 140692236586048, priori
[(:404)(PerformTask)] taskpool:: Task Perform: name : add, taskId : 14069223658604
[(:458)(HandleTaskResultCallback)] taskpool:: Task PerformTask End: taskId : 1406
taskpool res is: 3
[(:53)(~RunningScope)] taskpool:: RunningScope destruction
```

可以看到，系统会从任务池中分配一个 Task，然后执行这个 Task，执行的是函数 add，也就是用 @ConCurrent 装饰器装饰了的函数；结果是返回 3，也就是 1+2 的结果。

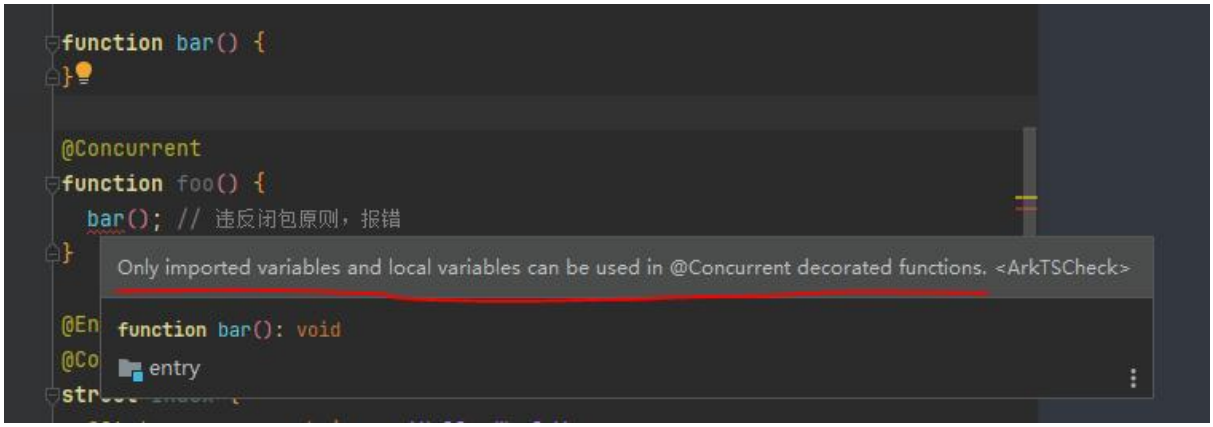
另外有一点要注意：

说明

由于@Concurrent标记的函数不能访问闭包，因此@Concurrent标记的函数内部不能调用当前文件的其他函数，例如：

```
function bar() {
}

@Concurrent
function foo() {
bar(); // 违反闭包原则，报错
}
```

报错截图：



就是说不能直接调用在同一个文件中的其他函数，需要把这个函数给挪到别的文件中，导出然后在这里再导入使用。

并发函数返回 Promise

并发函数中返回 Promise 的表现需关注，其中并发同步函数会处理返回该 Promise 并返回结果，用一个例子来演示，同样，把 Index.ets 中的代码替换为：

```
import { taskpool } from '@kit.ArkTS';

@Concurrent
function testPromise(args1: number, args2: number): Promise<number> {
  return new Promise<number>((testFuncA, testFuncB)=>{
    testFuncA(args1 + args2);
  });
}

@Concurrent
async function testPromise1(args1: number, args2: number): Promise<number> {
  return new Promise<number>((testFuncA, testFuncB)=>{
    testFuncA(args1 + args2);
  });
}

@Concurrent
async function testPromise2(args1: number, args2: number): Promise<number> {
  return await new Promise<number>((testFuncA, testFuncB)=>{
    testFuncA(args1 + args2)
  });
```

```
}

@Concurrent
function testPromise3() {
  return Promise.resolve(1);
}

@Concurrent
async function testPromise4(): Promise<number> {
  return 1;
}

@Concurrent
async function testPromise5(): Promise<string> {
  return await new Promise((resolve) => {
    setTimeout(()=>{
      resolve("Promise setTimeout after resolve");
    }, 1000)
  });
}

async function testConcurrentFunc() {
```

```
let task1: taskpool.Task = new taskpool.Task(testPromise, 1, 2);
let task2: taskpool.Task = new taskpool.Task(testPromise1, 1, 2);
let task3: taskpool.Task = new taskpool.Task(testPromise2, 1, 2);
let task4: taskpool.Task = new taskpool.Task(testPromise3);
let task5: taskpool.Task = new taskpool.Task(testPromise4);
let task6: taskpool.Task = new taskpool.Task(testPromise5);

taskpool.execute(task1).then((d:object)=>{
    console.info("task1 res is: " + d)
}).catch((e:object)=>{
    console.info("task1 catch e: " + e)
})
taskpool.execute(task2).then((d:object)=>{
    console.info("task2 res is: " + d)
}).catch((e:object)=>{
    console.info("task2 catch e: " + e)
})
taskpool.execute(task3).then((d:object)=>{
    console.info("task3 res is: " + d)
}).catch((e:object)=>{
    console.info("task3 catch e: " + e)
})
```

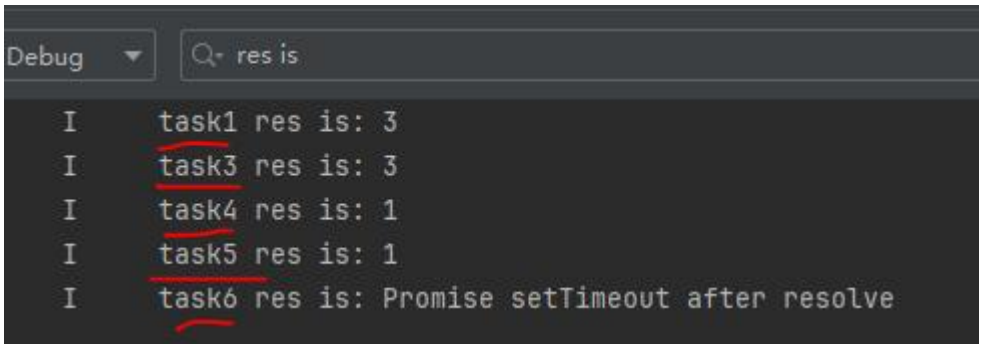
```
  taskpool.execute(task4).then((d:object)=>{
    console.info("task4 res is: " + d)
  }).catch((e:object)=>{
    console.info("task4 catch e: " + e)
  })
  taskpool.execute(task5).then((d:object)=>{
    console.info("task5 res is: " + d)
  }).catch((e:object)=>{
    console.info("task5 catch e: " + e)
  })
  taskpool.execute(task6).then((d:object)=>{
    console.info("task6 res is: " + d)
  }).catch((e:object)=>{
    console.info("task6 catch e: " + e)
  })
}

@Entry
@Component
struct Index {
  @State message: string = 'Hello World';
```

```
  build() {
    Row() {
      Column() {
        Button(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          .onClick(() => {
            testConcurrentFunc();
          })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

注意，这里并发函数返回的是一个 Promise，所以在调用了 taskpool.execute 之后要用到.then 以及.catch 来处理 fulfilled 或者 rejected 的结果。

```
  taskpool.execute(task1).then((d:object)=>{
    console.info("task1 res is: " + d)
  }).catch((e:object)=>{
    console.info("task1 catch e: " + e)
  })
```

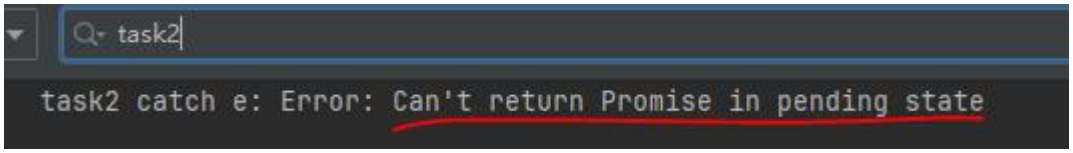
运行上面的代码，查看 Log:



注意，这里的 task6 是过了一秒中后才显示出来的，因为 testPromise5 中有一个 setTimeOut 后面设定的时间是 1 秒：

```
@Concurrent
async function testPromise5(): Promise<string> {
  return await new Promise((resolve) => {
    setTimeout(()=>{
      resolve("Promise setTimeout after resolve");
    }, 1000)
  });
}
```

这里的问题是，为什么没有 task2？查看 Log：



也就是说，task2 返回异常了。它的 Promise 始终是 pending 状态。

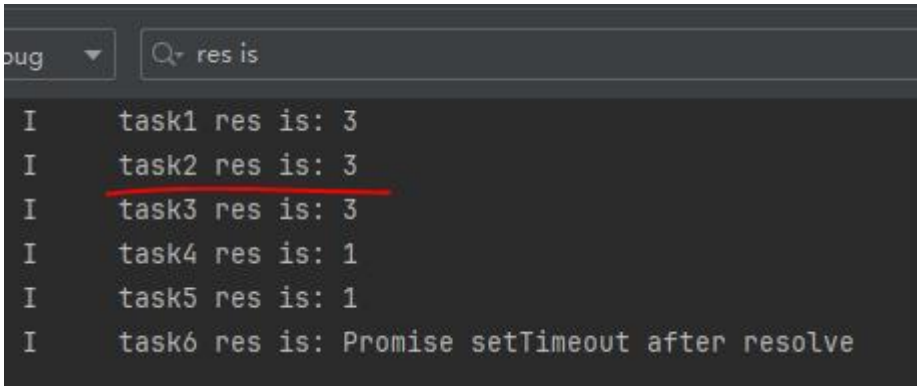
回到代码之中，task2 是调用的 testPromise1：

```
let task2: taskpool.Task = new taskpool.Task(testPromise1, 1, 2);
```

在 testPromise1 那里我们加一个 await:

```
@Concurrent
async function testPromise1(args1: number, args2: number): Promise<number> {
  return await new Promise<number>((testFuncA, testFuncB)=>{
    testFuncA(args1 + args2);
  });
}
```

这个 await 是之前没有的。此时再次在模拟器中运行，查看 Log:



六个任务都正常了。为什么要加这个 await？请大家讨论。

API Reference 中的例子

更详细的说明：

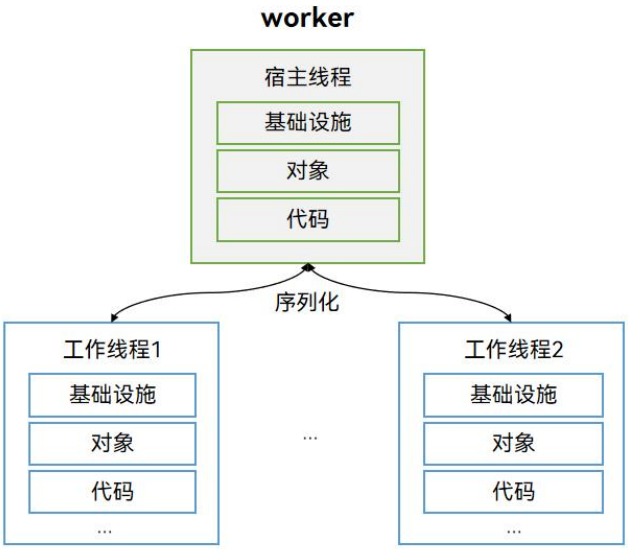
https://developer.huawei.com/consumer/cn/doc/harmonyos-references-V5/js-apis-taskpool-V5

在最后有八个实例，可以运行一下，加深理解。

- **Worker 多线程机制**

Worker 主要作用是为应用程序提供一个多线程的运行环境，可满足应用程序在执行过程中与主线程分离，在后台线程中运行一个脚本进行耗时操作，极大避免类似于计算密集型或高延迟的任务阻塞主线程的运行。

Worker 运作机制示意图：



创建 Worker 的线程称为**宿主线程**（不一定是主线程，工作线程也支持创建 Worker 子线程），Worker 自身的线程称为 **Worker 子线程**（或 **Actor 线程**、**工作线程**）。每个 Worker 子线程与宿主线程拥有**独立的实例**，包含基础设施、对象、代码段等，因此每个 Worker 启动存在一定的内存开销，需要限制 Worker 的子线程数量。**Worker 子线程和宿主线程之间的通信是基于消息传递**的，Worker 通过序列化机制与宿主线程之间相互通信，完成命令及数据交互。

Worker注意事项

- 创建Worker时，有手动和自动两种创建方式，手动创建Worker线程目录及文件时，还需同步进行相关配置，详情请参考创建Worker的注意事项。
- 使用Worker能力时，构造函数中传入的Worker线程文件的路径在不同版本有不同的规则，详情请参见文件路径注意事项。
- Worker创建后需要手动管理生命周期，且最多同时运行的Worker子线程数量为64个，详情请参见生命周期注意事项。
- 由于不同线程中上下文对象是不同的，因此Worker线程只能使用线程安全的库，例如UI相关的非线程安全库不能使用。
- 序列化传输的数据量大小限制为16MB。
- 使用Worker模块时，需要在主线程中注册onerror接口，否则当Worker线程出现异常时会发生jscrash问题。
- 不支持跨HAP使用Worker线程文件。
- 创建Worker对象时仅允许加载本模块下存在的Worker线程文件，不支持加载其他模块的Worker线程文件。若依赖其他模块提供的Worker功能，需要将Worker实现的整套逻辑封装到方法中，将方法导出后供其他模块使用。
- 引用HAR/HSP前，需要先配置对HAR/HSP的依赖，详见引用共享包。
- 不支持在Worker工作线程中使用AppStorage。

先尝试一下手动创建 Worker:

创建Worker的注意事项

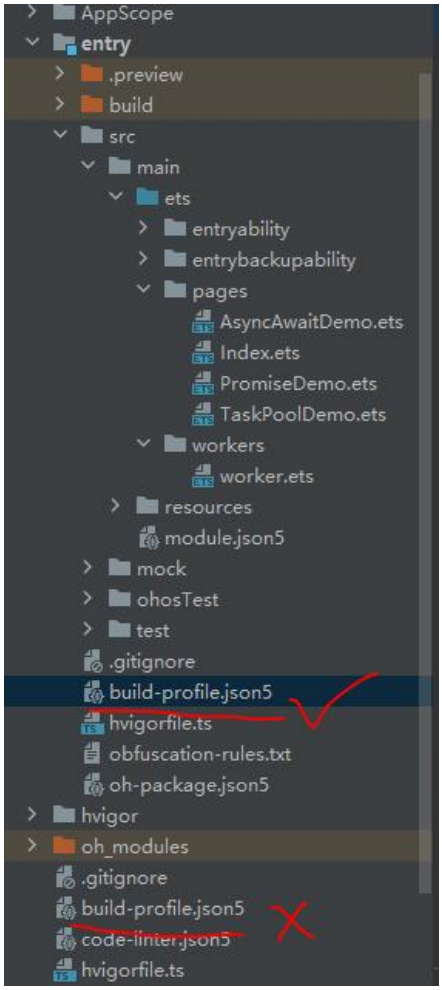
Worker线程文件需要放在"{moduleName}/src/main/ets/"目录层级之下，否则不会被打包到应用中。有手动和自动两种创建Worker线程目录及文件的方式。

- 手动创建：开发者手动创建相关目录及文件，此时需要配置build-profile.json5的相关字段信息，Worker线程文件才能确保被打包到应用中。

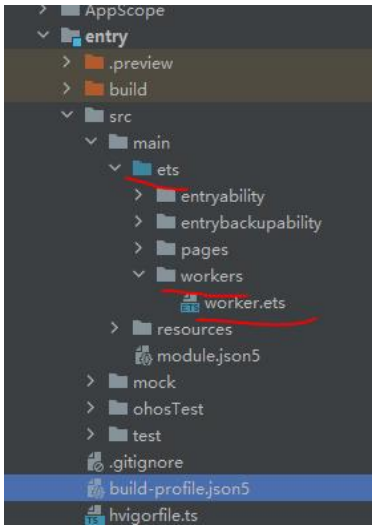
  Stage模型：

```
"buildOption": {
  "sourceOption": {
    "workers": [
      "./src/main/ets/workers/worker.ets"
    ]
  }
}
```

找到模块的 build-profile.json5 文件，注意不是项目的：



先在 src/main/ets 目录下面创建一个子目录 workers，然后创建一个新的文件 worker.ets:



在 build-profile.json5 中的 buildOption 里面添加配置代码：


```
"sourceOption": {
  "workers": [
    "./src/main/ets/workers/worker.ets"
  ]
}
```

在 worker.ets 中添加代码：

```
// worker.ets

import { worker, MessageEvents, ErrorEvent } from '@kit.ArkTS';
```

```
// 创建 worker 线程中与主线程通信的对象
const workerPort = worker.workerPort

// worker 线程接收主线程信息
workerPort.onmessage = (e: MessageEvents): void => {
  // data：主线程发送的信息
  let data: string = e.data;
  console.log("WorkerTest: worker.ets onmessage", data);

  // worker 线程向主线程发送信息
  workerPort.postMessage("123")
}

// worker 线程发生 error 的回调
workerPort.onerror = (err: ErrorEvent) => {
  console.log("WorkerTest: worker.ets onerror" + err.message);
}
```

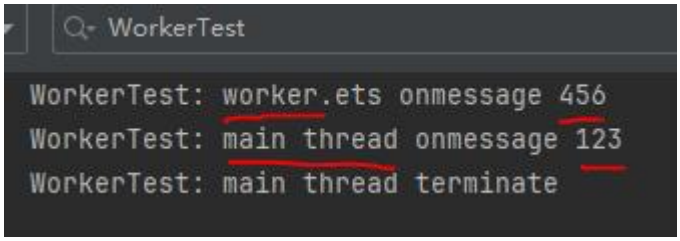
将 Index.ets 替换为：

```
// main thread（以不同目录为例）
import { worker, MessageEvents, ErrorEvent } from '@kit.ArkTS';
```

```
// 主线程中创建 Worker 对象
const workerInstance = new worker.ThreadWorker("../workers/worker.ets");

// 主线程向 worker 线程传递信息
workerInstance.postMessage("456");

// 主线程接收 worker 线程信息
workerInstance.onmessage = (e: MessageEvents): void => {
  // data：worker 线程发送的信息
  let data: string = e.data;
  console.log("WorkerTest: main thread onmessage", data);

  // 销毁 Worker 对象
  workerInstance.terminate();
}
// 在调用 terminate 后，执行 onexit
workerInstance.onexit = (code) => {
  console.log("WorkerTest: main thread terminate");
}

workerInstance.onerror = (err: ErrorEvent) => {
  console.log("WorkerTest: main error message " + err.message);
```

```
}

@Entry
@Component
struct Index {
  @State message: string = 'Hello World';

  build() {
    Row() {
      Column() {
        Button(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

注意，Worker 子线程向主线程发送的是 123，而主线程向子线程发送的是 456。查看 Log:

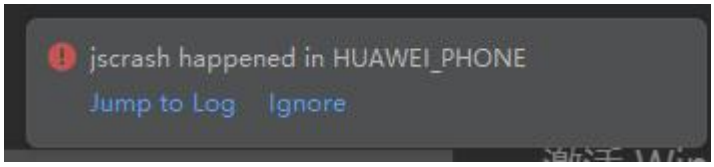


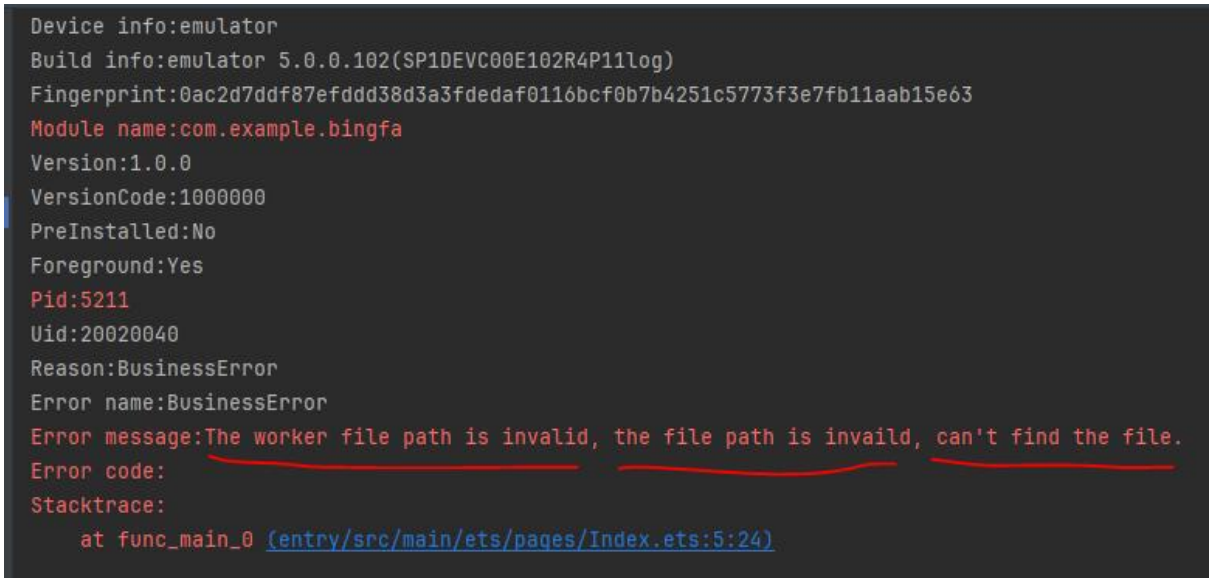
这个例子基本上演示了主线程如何创建 Worker 子线程，以及互相之间是如何通信的。

注意，这里的路径一定要设置正确：

```
// 主线程中创建 Worker 对象
const workerInstance = new worker.ThreadWorker("../workers/worker.ets");
```

如果改为一个不正确的路径，会出现 jscrash 报错：



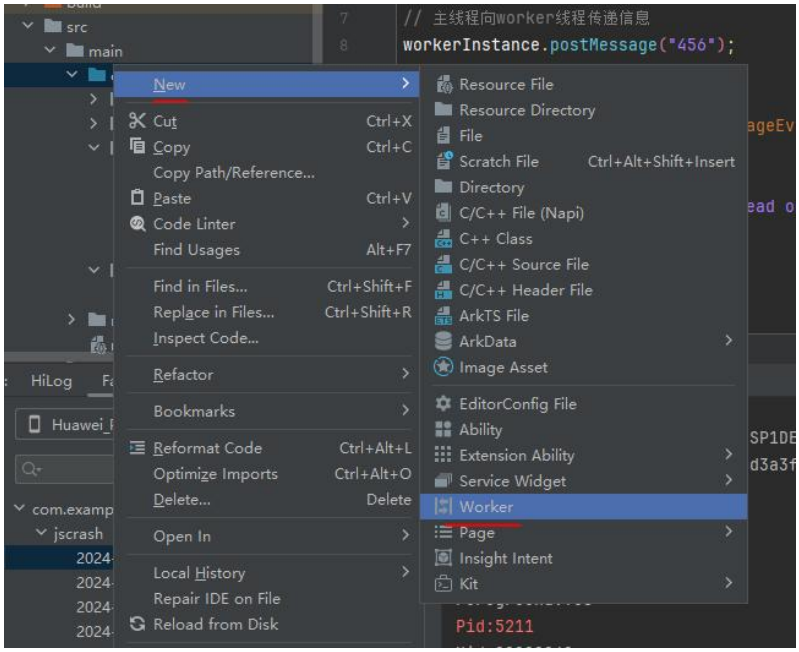


Worker 的配置也可以自动创建：

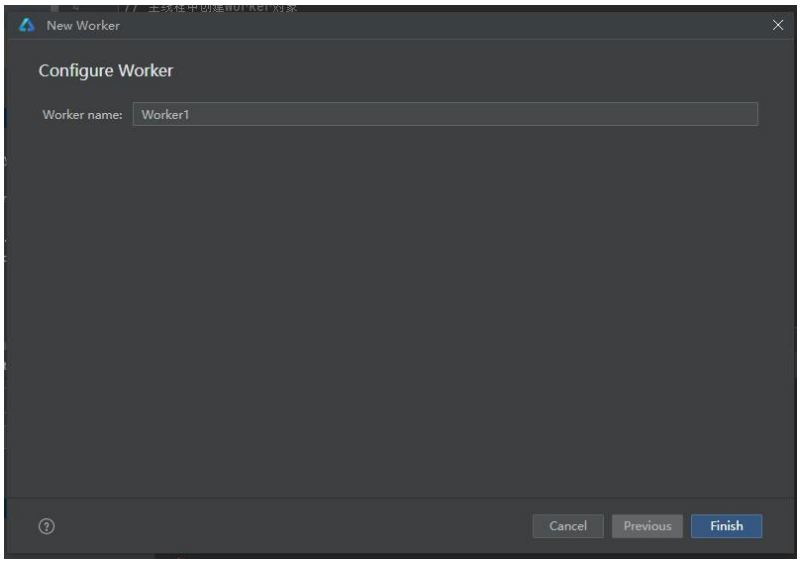
DevEco Studio 支持一键生成 Worker，在对应的{moduleName}目录下任意位置，点击鼠标右键 > New > Worker，即可自动生成 Worker 的模板文件及配置信息，无需再手动在 build-profile.json5

中进行相关配置。

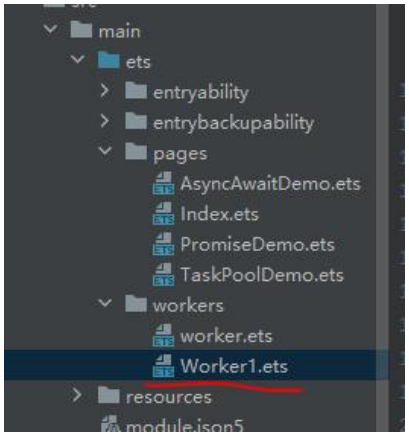
右键单击 ets 目录，选择 New > Worker



进入页面：



点击 Finish 之后，可以看到，系统自动创建了一个文件 Worker1.ets，在 workers 目录下：



代码模板也生成了：

```
import { ErrorEvent, MessageEvents, ThreadWorkerGlobalScope, worker } from '@kit.ArkTS';

const workerPort: ThreadWorkerGlobalScope = worker.workerPort;

/**
 * Defines the event handler to be called when the worker thread receives a message sent by the host thread.
 * The event handler is executed in the worker thread.
 *
 * @param e message data
 */
workerPort.onmessage = (e: MessageEvents) => {
}

/**
```

```
 * Defines the event handler to be called when the worker receives a message that cannot be deserialized.
 * The event handler is executed in the worker thread.
 *
 * @param e message data
 */
workerPort.onmessageerror = (e: MessageEvents) => {
}

/**
 * Defines the event handler to be called when an exception occurs during worker execution.
 * The event handler is executed in the worker thread.
 *
 * @param e error message
 */
workerPort.onerror = (e: ErrorEvent) => {
}
```

相应地，在 build-profile.json5 中，也做了修改：

所以使用自动生成的方式更为方便。

5. TaskPool 和 Worker 的对比

学到这里，只是学了点皮毛。

TaskPool（任务池）和 Worker 的作用是为应用程序提供一个多线程的运行环境，用于处理耗时的计算任务或其他密集型任务。可以有效地避免这些任务阻塞主线程，从而最大化系统的利用率，降低整体资源消耗，并提高系统的整体性能。

仔细读一下这里的解释：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/taskpool-vs-worker-V5

另外，这里有比较好的实例：

https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/multithread-develop-guide-V5

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？