1003.4.鸿蒙基础-通信-线程与进程.DOC

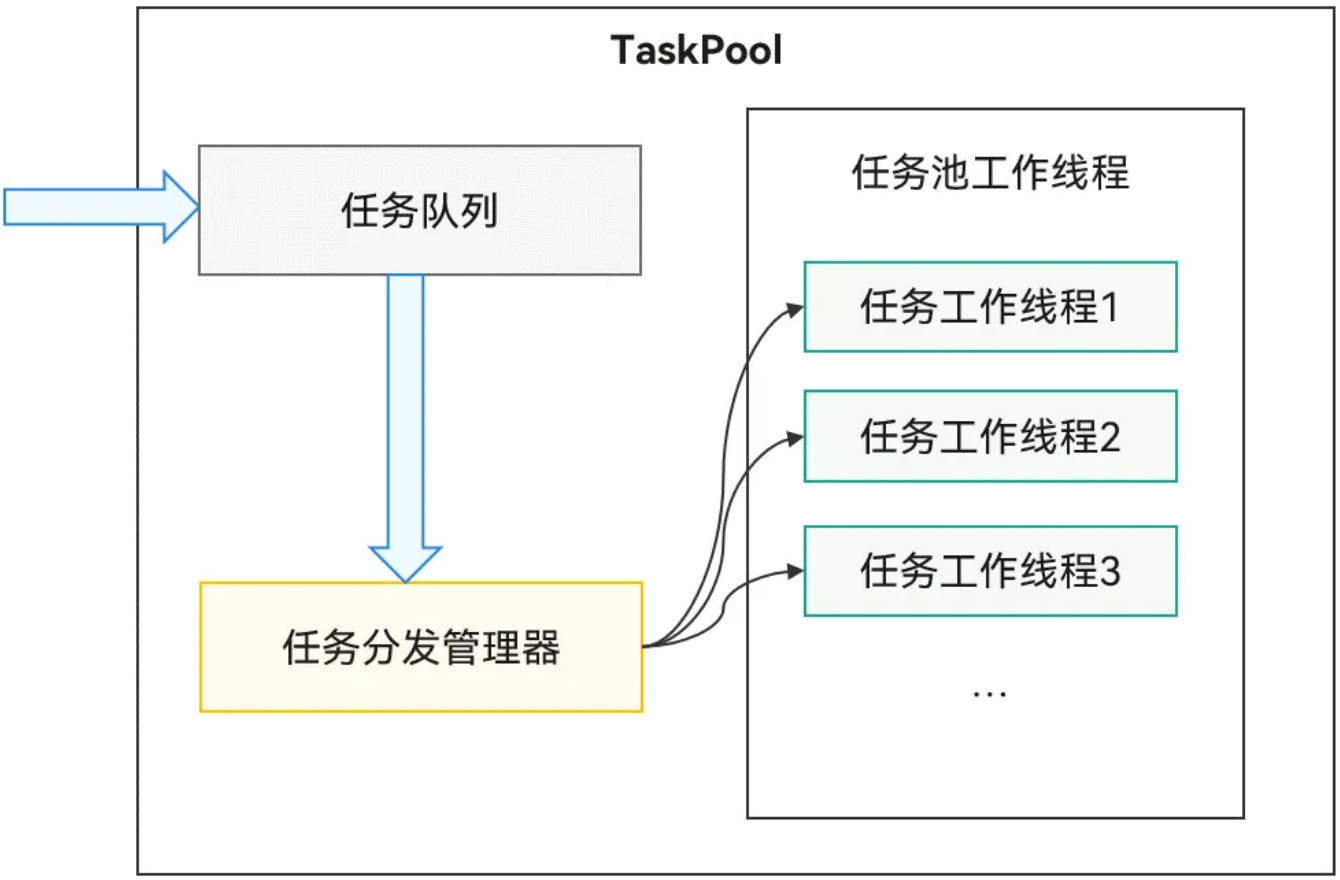
线程模型

Stage模型下的线程分类

- 主线程
  - 执行UI绘制。
  - 管理主线程的ArkTS引擎实例，使多个UIAbility组件能够运行在其之上。
  - 管理其他线程的ArkTS引擎实例，例如使用TaskPool（任务池）创建任务或取消任务、启动和终止Worker线程。
  - 分发交互事件。
  - 处理应用代码的回调，包括事件处理和生命周期管理。
  - 接收TaskPool以及Worker线程发送的消息。
- TaskPool线程
  - 用于执行耗时操作，支持设置调度优先级、负载均衡等功能，推荐使用。
- Worker线程
  - 用于执行耗时操作，支持线程间通信。

@ohos.taskpool（启动任务池）

运作机制



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。

开发流程

- **1.封装任务**
  - 实现任务的函数需要使用装饰器@Concurrent标注，且仅支持在.ets文件中使用

```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);
  }
}
```

- 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。

Priority

表示所创建任务（Task）执行时的优先级。工作线程优先级跟随任务优先级同步更新，对应关系参考QoS等级定义。

系统能力： SystemCapability.Utils.Lang

名称	值	说明
HIGH	0	任务为高优先级。 **元服务API**：从API version 11 开始，该接口支持在元服务中使用。
MEDIUM	1	任务为中优先级。 **元服务API**：从API version 11 开始，该接口支持在元服务中使用。
LOW	2	任务为低优先级。 **元服务API**：从API version 11 开始，该接口支持在元服务中使用。
IDLE[12+]	3	任务为后台任务。 **元服务API**：从API version 12 开始，该接口支持在元服务中使用。

- Promise不支持跨线程传递，不能作为concurrent function的返回值。

ArkTS

```
// 正例
@Concurrent
  async function asyncFunc(val1:number, val2:number): Promise<number> {
    let ret: number = await new Promise((resolve, reject) => {
      let value = val1 + val2;
      resolve(value);
    });
    return ret; // 支持。直接返回Promise的结果。
  }

function taskpoolExecute() {
  taskpool.execute(asyncFunc, 10, 20).then((result: Object) => {
    console.info("taskPoolTest task result: " + result);
  }).catch((err: string) => {
    console.error("taskPoolTest test occur error: " + err);
  });
}
taskpoolExecute()
```

ArkTS

```
// 反例1:
@Concurrent
  async function asyncFunc(val1:number, val2:number): Promise<number> {
    let ret: number = await new Promise((resolve, reject) => {
      let value = val1 + val2;
      resolve(value);
    });
    return Promise.resolve(ret); // 不支持。Promise.resolve仍是Promise，其状态是pending，无法作为返回值使用。
  }

// 反例2:
@Concurrent
  async function asyncFunc(val1:number, val2:number): Promise<number> {
    // 不支持。其状态是pending，无法作为返回值使用。
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        let value = val1 + val2;
        resolve(value);
      }, 2000);
    });
  }
```

- 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。
- 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。

- **2.添加函数或任务至任务队列等待分发执行**
  - **语法1：传递函数**

taskpool.execute

execute(func: Function, ...args: Object[]): Promise<Object>

  - **语法2：设置优先级**

taskpool.execute

execute(task: Task, priority?: Priority): Promise<Object>

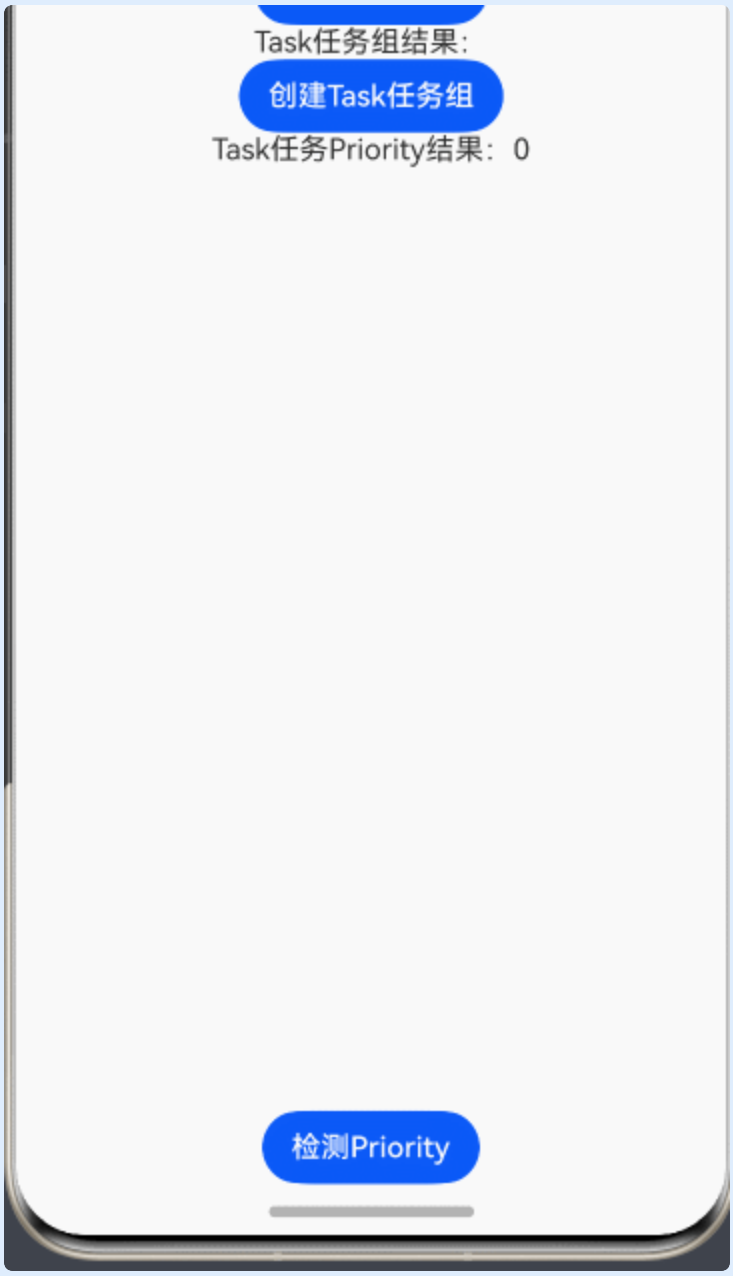
  - **语法3：传入任务组**

taskpool.execute 10+

execute(group: TaskGroup, priority?: Priority): Promise<Object[]>

- **3.等待执行任务结果**

传入任务设置优先级可以进行执行顺序查看



ArkTS

```
  async checkPriority() {
    let taskArray: Array<taskpool.Task> = [];
    // 创建100个任务并添加至taskArray
    for (let i: number = 0; i < 100; i++) {
      let task: taskpool.Task = new taskpool.Task(getData, 'task', i + ':');
      taskArray.push(task);
    }
    for (let i: number = 0; i < taskArray.length; i += 4) { // 4: 每次执行4个任务，循环取任务时需后移4项，确保执行的是不同的任务
      taskpool.execute(taskArray[i], taskpool.Priority.HIGH).then(res => {
        this.taskPriority.push(res.toString());
      })
      taskpool.execute(taskArray[i+1], taskpool.Priority.MEDIUM).then(res => {
        this.taskPriority.push(res.toString());
      })
      taskpool.execute(taskArray[i+2], taskpool.Priority.LOW).then(res => {
        this.taskPriority.push(res.toString());
      })
      taskpool.execute(taskArray[i+3], taskpool.Priority.IDLE).then(res => {
        this.taskPriority.push(res.toString());
      })
    }
  }
```

闲置状态下，中高级的任务会优先执行，其他状态会依次执行（电脑性能较弱的可以改小任务数量避免卡顿）

示例代码

ArkTS

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

// 1.创建任务
@Concurrent
async function getData(params1: string, params2: string) {
  await new Promise<boolean>((resolve) => {
    setTimeout(() => {
      return resolve(true)
    }, 3000)
  })
  return params1 + params2 + Math.random().toFixed(2)
}

@Entry
@Component
struct TaskPoolCase {
  @State
  addTaskResult: string = ''
  @State
  createTaskResult: string = ''
  @State
  taskGroup: string[] = []
  @State
  taskPriority: string[] = []

  async addTask() {
    //2.将Concurrent函数添加至队列
    const result = await taskpool.execute(getData, 'addTask', '-')
    // 3.等待处理结果
    this.addTaskResult = result.toString()
  }

  async createTask() {
    //2.创建task任务
    const task = new taskpool.Task(getData, 'createTask', '-')
    const result = await taskpool.execute(task, taskpool.Priority.LOW)
    // 3.等待处理结果
    this.createTaskResult = result.toString()
  }

  async createTaskGroup() {
    //2.将task任务添加至队列（自动分配）
    const group = new taskpool.TaskGroup()
    group.addTask(getData, 'createTaskGroup4', '-')
```

```
45      group.addTask(getData, 'createTaskGroup1', '-')
46      group.addTask(getData, 'createTaskGroup2', '-')
47      group.addTask(getData, 'createTaskGroup3', '-')
48      const result = await taskpool.execute(group, taskpool.Priority.LOW)
49      this.taskGroup = result.map((item: Object) => item.toString())
50    }
51
52    async checkPriority() {
53      let taskArray: Array<taskpool.Task> = [];
54      // 创建100个任务并添加至taskArray
55      for (let i: number = 0; i < 100; i++) {
56        let task: taskpool.Task = new taskpool.Task(getData, 'task', i +
    ':');
57        taskArray.push(task);
58      }
59      for (let i: number = 0; i < taskArray.length; i += 4) { // 4: 每次执行
    4个任务，循环取任务时需后移4项，确保执行的是不同的任务
60        taskpool.execute(taskArray[i], taskpool.Priority.HIGH).then(res =>
    {
61          this.taskPriority.push(res.toString());
62        })
63        taskpool.execute(taskArray[i+1], taskpool.Priority.MEDIUM).then(re
    s => {
64          this.taskPriority.push(res.toString());
65        })
66        taskpool.execute(taskArray[i+2], taskpool.Priority.LOW).then(res =
    > {
67          this.taskPriority.push(res.toString());
68        })
69        taskpool.execute(taskArray[i+3], taskpool.Priority.IDLE).then(res =
    > {
70          this.taskPriority.push(res.toString());
71        })
72      }
73    }
74
75    build() {
76      Column() {
77        Text('addTask任务结果：' + this.addTaskResult)
78        Button('添加Task任务')
79          .onClick(() => {
80            this.addTask()
81          })
82        Text('createTask任务结果：' + this.createTaskResult)
83        Button('创建Task任务')
84          .onClick(() => {
85            this.createTask()
```

```
86            })
87          Text('Task任务组结果：')
88          ForEach(this.taskGroup, (item: string) => {
89            Text(item)
90          })
91          Button('创建Task任务组')
92            .onClick(() => {
93              this.createTaskGroup()
94            })
95          Text('Task任务Priority结果：'+this.taskPriority.length)
96          List() {
97            ForEach(this.taskPriority, (item: string) => {
98              ListItem() {
99                Text(item)
100             }
101           })
102         }.layoutWeight(1)
103
104         Button('检测Priority')
105           .onClick(() => {
106             this.checkPriority()
107           })
108       }
109       .height('100%')
110       .width('100%')
111     }
```

通信测试

任务代码执行时发送通知，页面进行订阅

发送通知必须使用线程之间通信的emitter，使用eventHub会出现阻塞卡死的现象

- 发布

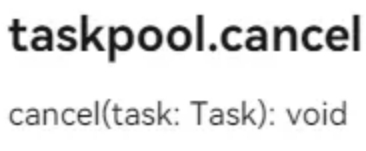
```ArkTS
@Concurrent
async function getData(params1: string, params2: string) {
  await new Promise<boolean>((resolve) => {
    setTimeout(() => {
      return resolve(true)
    }, 3000)
  })
  // 线程内无法通信，会阻塞await
  // getContext().eventHub.emit('taskpool')
  // 跨线程通信可以成功
  emitter.emit('taskpool')
  return params1 + params2 + Math.random().toFixed(2)
}
```

- 订阅

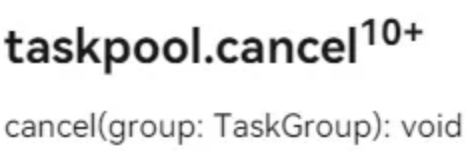
```ArkTS
  aboutToAppear(): void {
    // 线程内无法订阅到
    getContext().eventHub.on('taskpool', () => {
      this.emitNum++
    })
    // 线程间可以
    emitter.on('taskpool', () => {
      this.emitNum++
    })
  }
```

取消任务

- 取消单个任务



- 取消任务组



ArkTS

```
//代码示例：执行前点击可以取消执行
Button('取消任务11')
        .onClick(()=>{
          taskpool.cancel(this.taskArray[11])
        })
```

完整代码

ArkTS

```ts
import { taskpool } from '@kit.ArkTS'
import { emitter } from '@kit.BasicServicesKit'

// 1.创建任务
@Concurrent
async function getData(params1: string, params2: string) {
  await new Promise<boolean>((resolve) => {
    setTimeout(() => {
      return resolve(true)
    }, 3000)
  })
  // 线程内无法通信，会阻塞await
  // getContext().eventHub.emit('taskpool')
  // 跨线程通信可以成功
  emitter.emit('taskpool')
  return params1 + params2 + Math.random().toFixed(2)
}

@Entry
@Component
struct TaskPoolCase {
  @State
  addTaskResult: string = ''
  @State
  createTaskResult: string = ''
  @State
  taskGroup: string[] = []
  @State
  taskPriority: string[] = []
  @State
  emitNum: number = 0
  taskArray:taskpool.Task[] = []

  aboutToAppear(): void {
    // 线程内无法订阅到
    getContext().eventHub.on('taskpool', () => {
      this.emitNum++
    })
    // 线程间可以
    emitter.on('taskpool', () => {
      this.emitNum++
    })
  }

```

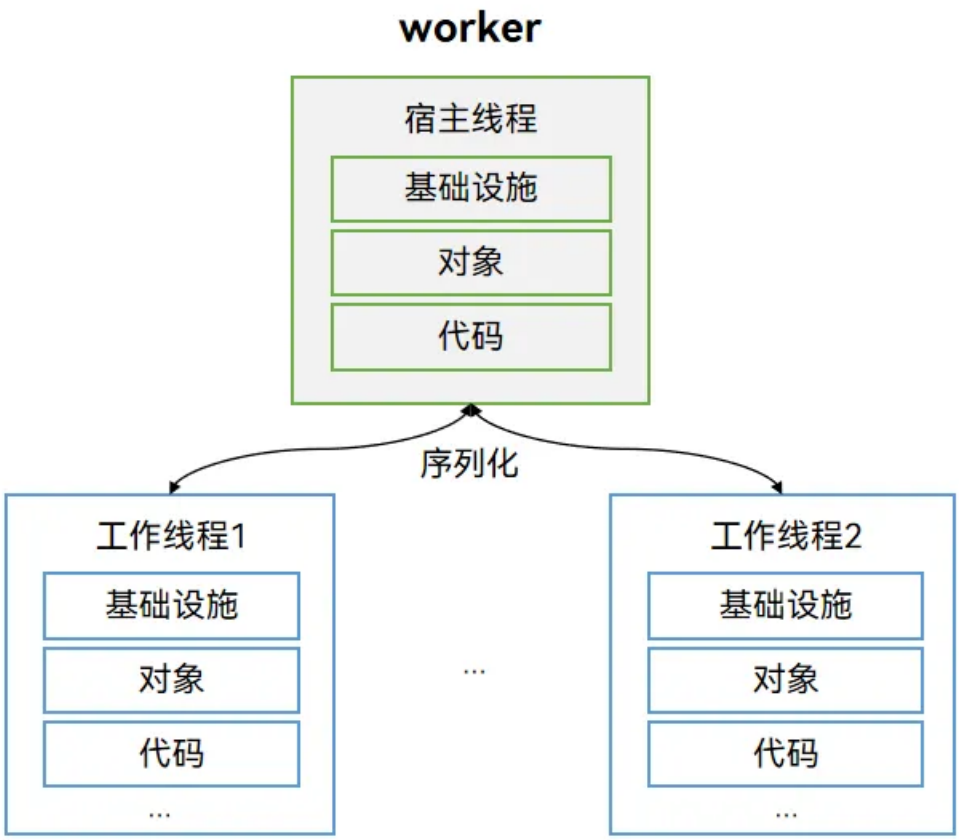
```
45    async addTask() {
46      //2.将Concurrent函数添加至队列
47      const result = await taskpool.execute(getData, 'addTask', '-')
48      // 3.等待处理结果
49      this.addTaskResult = result.toString()
50    }
51
52    async createTask() {
53      //2.创建task任务
54      const task = new taskpool.Task(getData, 'createTask', '-')
55      const result = await taskpool.execute(task, taskpool.Priority.LOW)
56      // 3.等待处理结果
57      this.createTaskResult = result.toString()
58    }
59
60    async createTaskGroup() {
61      //2.将task任务添加至队列（自动分配）
62      const group = new taskpool.TaskGroup()
63      group.addTask(getData, 'createTaskGroup4', '-')
64      group.addTask(getData, 'createTaskGroup1', '-')
65      group.addTask(getData, 'createTaskGroup2', '-')
66      group.addTask(getData, 'createTaskGroup3', '-')
67      const result = await taskpool.execute(group, taskpool.Priority.LOW)
68      this.taskGroup = result.map((item: Object) => item.toString())
69    }
70
71    async checkPriority() {
72      let taskArray: Array<taskpool.Task> = [];
73      // 创建100个任务并添加至taskArray
74      for (let i: number = 0; i < 100; i++) {
75        let task: taskpool.Task = new taskpool.Task(getData, 'task', i +
    ':');
76        taskArray.push(task);
77      }
78      this.taskArray = taskArray
79      for (let i: number = 0; i < taskArray.length; i += 4) { // 4: 每次执行
    4个任务，循环取任务时需后移4项，确保执行的是不同的任务
80        taskpool.execute(taskArray[i], taskpool.Priority.HIGH).then(res =>
    {
81          this.taskPriority.push(res.toString());
82        })
83        taskpool.execute(taskArray[i+1], taskpool.Priority.MEDIUM).then(re
    s => {
84          this.taskPriority.push(res.toString());
85        })
86        taskpool.execute(taskArray[i+2], taskpool.Priority.LOW).then(res =
    > {
```

```
87            this.taskPriority.push(res.toString());
88          })
89          taskpool.execute(taskArray[i+3], taskpool.Priority.IDLE).then(res =
    > {
90            this.taskPriority.push(res.toString());
91          })
92        }
93      }
94
95      build() {
96        Column() {
97          Text('emit次数：' + this.emitNum)
98          Text('addTask任务结果：' + this.addTaskResult)
99          Button('添加Task任务')
100           .onClick(() => {
101             this.addTask()
102           })
103         Text('createTask任务结果：' + this.createTaskResult)
104         Button('创建Task任务')
105           .onClick(() => {
106             this.createTask()
107           })
108         Text('Task任务组结果：')
109         ForEach(this.taskGroup, (item: string) => {
110           Text(item)
111         })
112         Button('创建Task任务组')
113           .onClick(() => {
114             this.createTaskGroup()
115           })
116         Text('Task任务Priority结果：' + this.taskPriority.length)
117         List() {
118           ForEach(this.taskPriority, (item: string) => {
119             ListItem() {
120               Text(item)
121             }
122           })
123         }.layoutWeight(1)
124
125         Button('检测Priority')
126           .onClick(() => {
127             this.checkPriority()
128           })
129         Button('取消任务11')
130           .onClick(()=>{
131             taskpool.cancel(this.taskArray[11])
132           })
```

```
133       }
134       .height('100%')
135       .width('100%')
136     }
```

@ohos.worker (启动一个Worker)

运作机制



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。

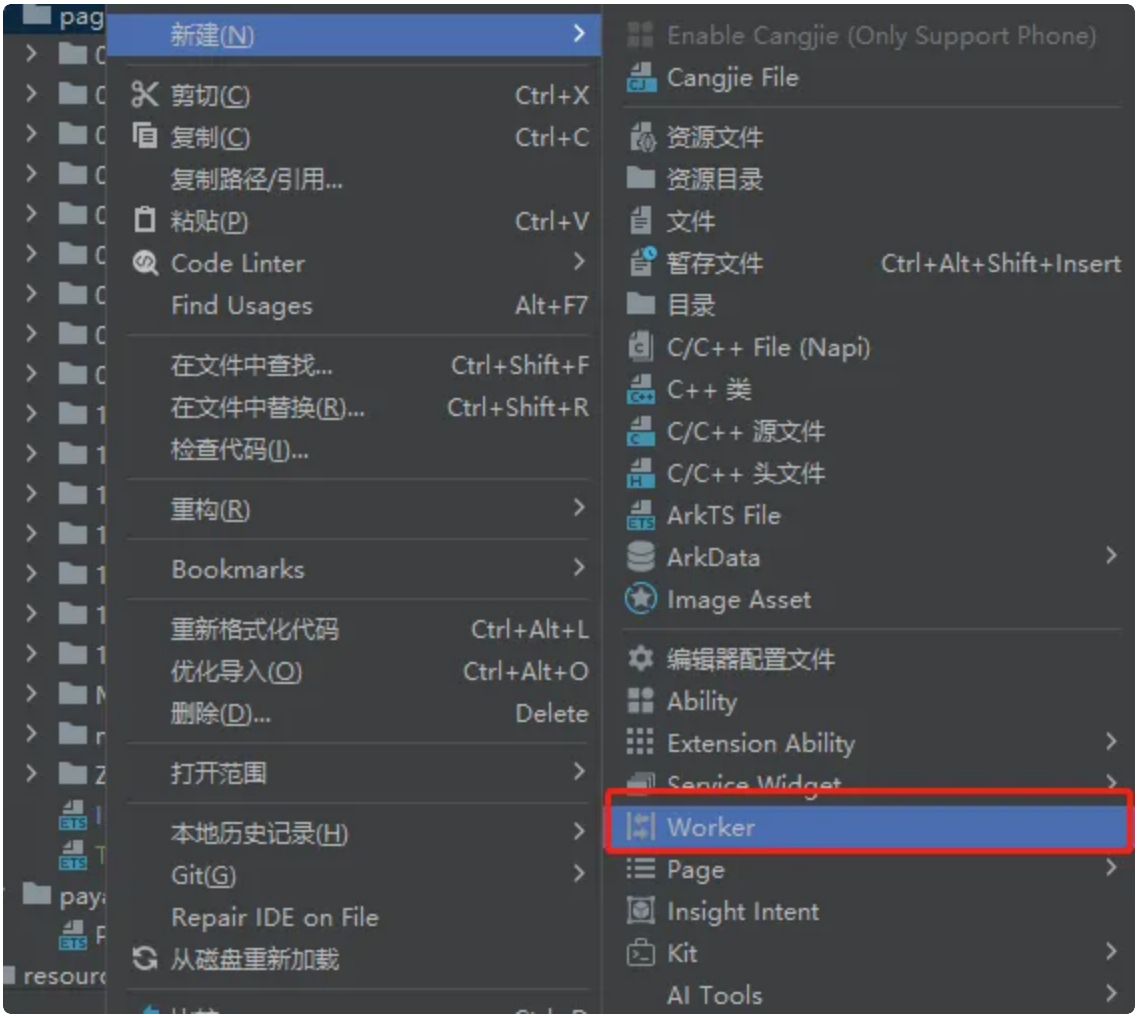
开发流程

- 手动创建（了解）：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。

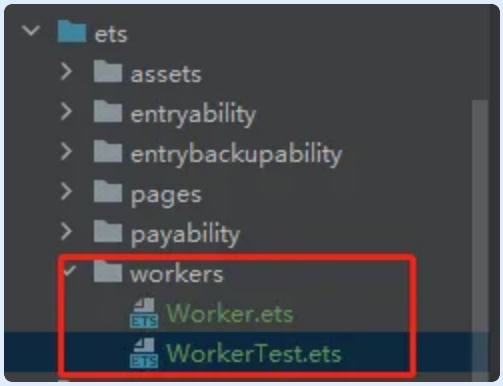
ArkTS

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

- 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。



创建worker模块后自动得到一个worker文件，可以创建多个，自动保存至Worker目录下



默认代码示例：

ArkTS

```
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) => {
}
```

- 1.创建worker:根据worker文件进行创建，需要指定路径

```ArkTS
createWorker(){
  // 1.创建worker
  // 路径规范：{模块}/ets/目录名称/文件名
  const myWorker = new worker.ThreadWorker("entry/ets/workers/worker")
}
```

此时worker已经创建成功，就这么简单，但是需要注意的是

- Worker的创建和销毁**耗费性能**，建议开发者合理管理已创建的Worker并重复使用。Worker**空闲时也会一直运行**，因此当不需要Worker时，可以调用terminate()接口或close()方法主动**销毁**Worker。若Worker处于已销毁或正在销毁等非运行状态时，调用其功能接口，会抛出相应的错误。
- Worker的数量由内存管理策略决定，设定的内存阈值为1.5GB和设备物理内存的60%中的较小者。在内存允许的情况下，**系统最多可以同时运行64个Worker**。如果尝试创建的Worker数量超出这一上限，系统将抛出错误：“Worker initialization failure, the number of workers exceeds the maximum.”。实际运行的Worker数量会根据当前内存使用情况动态调整。一旦所有Worker和主线程的累积内存占用超过了设定的阈值，系统将触发内存溢出（OOM）错误，导致应用程序崩溃。

- **2.worker文件内创建任务：此时开启了多线程，需要干什么，在worker文件中声明**

```ArkTS
//声明一个下载DevEco Studio的方法（下载连接可能会失效，使用官网最新的）
async function downLoadFile(cb: (progress: string) => void) {
  const task = await request.downloadFile(getContext(), {
    url: 'https://contentcenter-vali-drcn.dbankcdn.cn/pvt_2/DeveloperAlliance_package_901_9/bd/v3/lye4fygWRU-orI0FDvOVDw/devecostudio-windows-5.0.3.806.zip?HW-CC-KV=V1&HW-CC-Date=20240920T195541Z&HW-CC-Expire=7200&HW-CC-Sign=57B11A6656E3FA31A1E421EE38AB347E04A7215CA4CBBC94AB9D1BC06DE92DFE'
  })
  task.on('progress', (current, total) => {
    cb((current / total * 100).toFixed(2) + '%')
  })
}
```

- 在onmessage中调用该方法

```ArkTS
workerPort.onmessage = (e: MessageEvents) => {
  //   2.声明要做的事
  downLoadFile((progress) => {
    // 要告诉页面这个下载进度
    workerPort.postMessage({
      progress
    })
  })
}
```

- 但是这个方法真正的执行时机是收到消息的时候，需要在页面进行消息的发送和接收

```ArkTS
  createWorker() {
    // 1.创建任务
    // 路径规范：{模块}/ets/目录名称/文件名.ets
    const myWorker = new worker.ThreadWorker("phone/ets/workers/Worker.ets")
    // 2.去worker中设置任务
    // 3.需要发消息通知worker执行任务
    myWorker.postMessage({
      work:'start'
    })
    // 4.监听响应的结果
    myWorker.onmessage = (e) => {
      this.downloadProgress = e.data.progress as string
    }
  }
```

此时点击下载并不会执行下载任务，因为worker文件内的上下文根本取不到！
Worker是宿主线程，所有UI能力相关的事都做不了！
所以必须在页面将上下文传入worker执行

改造页面通知worker

```ArkTS
    myWorker.postMessage({
      work:'start',
      params:{
        context:getContext()
      }
    })
```

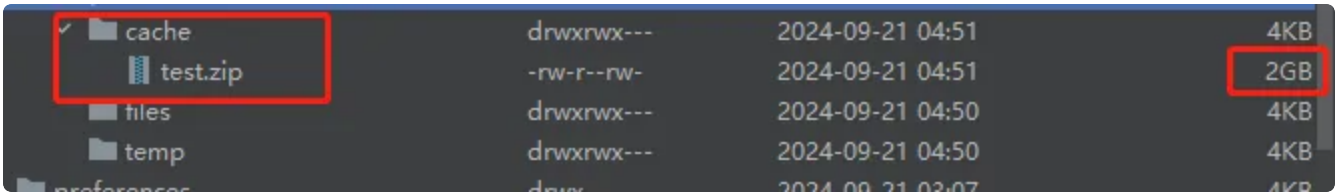
worker接收

```ArkTS
workerPort.onmessage = (e: MessageEvents) => {
  //   2.声明要做的事
  downLoadFile(e.data.params.context as Context, (progress) => {
    // 要告诉页面这个下载进度
    workerPort.postMessage({
      progress
    })
  })
}
```

下载方法修改上下文和沙箱路径

```ArkTS
async function downLoadFile(context: Context, cb: (progress: string) => void) {
  const task = await request.downloadFile(context, {
    url: 'https://contentcenter-vali-drcn.dbankcdn.cn/pvt_2/DeveloperAlliance_package_901_9/bd/v3/lye4fygWRU-orI0FDvOVDw/devecostudio-windows-5.0.3.806.zip?HW-CC-KV=V1&HW-CC-Date=20240920T204942Z&HW-CC-Expire=7200&HW-CC-Sign=0A8C4F25E03D6F3EF144D5F7818B4993B815F1089F825E10DC50C6C394F773B7',
    filePath: context.cacheDir + '/test.zip'
  })
  task.on('progress', (current, total) => {
    cb((current / total * 100).toFixed(2) + '%')
  })
}
```

测试成功



通信测试

worker的通信主要靠自身和页面的 `postMessage` 和 `onmessage` 所以，对通信的难度并不大
同理，我们也可以尝试测试eventHub和emitter进行通信
由于拿不到上下文，所以eventHub肯定找不到，所以还是emitter可以进行通信

下载过程中直接使用emitter代替postMessage

```ArkTS
    task.on('progress', (current, total) => {
      // cb((current / total * 100).toFixed(2) + '%')
      emitter.emit('worker',{
        data:{
          progress:(current / total * 100).toFixed(2) + '%'
        }
      })
    })
```

关闭worker

下载完成后，worker还处于开启的状态，仍然会占用内存，所以需要及时释放

- Worker的创建和销毁耗费性能，建议开发者合理管理已创建的Worker并重复使用。Worker空闲时也会一直运行，因此当不需要Worker时，可以调用**terminate()**接口或**close()**方法主动销毁Worker。若Worker处于已销毁或正在销毁等非运行状态时，调用其功能接口，**会抛出相应的错误**。

{"code":10200004,"name":"BusinessError"}

ArkTS

```
  task.on('progress', (current, total) => {
    // cb((current / total * 100).toFixed(2) + '%')
    emitter.emit('worker',{
      data:{
        progress:(current / total * 100).toFixed(2) + '%'
      }
    })
    //下载完成关闭worker
    if(current === total){
      workerPort.close()
    }
  })
```

可以下载中和下载完成测试通信查看时候关闭

完整代码

WorkerCase.ets ArkTS

```
import { MessageEvent, worker } from '@kit.ArkTS'
import { BusinessError, emitter } from '@kit.BasicServicesKit'
import { promptAction } from '@kit.ArkUI'
import { Context } from '@kit.AbilityKit'

export interface MessageInfo{
  work:string
  params?:Record<string,string>
  context?:Context
}

@Entry
@Component
struct WorkerCase {
  @State
  downloadProgress: string = '0%'

  aboutToAppear(): void {
    emitter.on('worker',(data)=>{
      this.downloadProgress =  data.data!.progress as string
    })
  }
  myWorker?:worker.ThreadWorker
  createWorker() {
    // 1.创建任务
    // 路径规范：{模块}/ets/目录名称/文件名.ets
    const myWorker = new worker.ThreadWorker("phone/ets/workers/Worker.ets")
    this.myWorker = myWorker
    // 2.去worker中设置任务
    // 3.需要发消息通知worker执行任务
    myWorker.postMessage({
      work:'start',
      context:getContext()
    } as MessageInfo)
    // 4.监听响应的结果
    myWorker.onmessage = (e) => {
      this.downloadProgress = e.data.progress as string
    }
  }

  tryWorker(){
    try{
      this.myWorker?.postMessage({
```

```
          work:'test'
        } as MessageInfo)
      } catch (err) {
        promptAction.showToast({
          message:JSON.stringify(err)
        })
      }
    }

    build() {
      Column() {
        Text('下载进度：' + this.downloadProgress)
        Button('下载DevEco Studio')
          .onClick(() => {
            this.createWorker()
          })
        Button('测试销毁')
          .onClick(()=>{
            this.tryWorker()
          })
      }
      .height('100%')
      .width('100%')
    }
  }
```

Worker.ets ArkTS

```
import { ErrorEvent, MessageEvents, ThreadWorkerGlobalScope, worker } fro
m '@kit.ArkTS';
import { emitter, request } from '@kit.BasicServicesKit';
import { Context } from '@kit.AbilityKit';
import { MessageInfo } from '../pages/WorkerCase';

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) => {
  console.log('test-progress','onmessage')
  const work = e.data as MessageInfo
  if(work.work==='start'){
    //   2.声明要做的事
    downLoadFile(work.context as Context, (progress) => {
      // 要告诉页面这个下载进度
      workerPort.postMessage({
        progress
      })
    })
  }
}

/**
 * Defines the event handler to be called when the worker receives a messa
ge 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) => {
}

async function downLoadFile(context: Context, cb: (progress: string) => void) {
  const task = await request.downloadFile(context, {
    url: 'https://contentcenter-vali-drcn.dbankcdn.cn/pvt_2/DeveloperAlliance_package_901_9/bd/v3/lye4fygWRU-orI0FDvOVDw/devecostudio-windows-5.0.3.806.zip?HW-CC-KV=V1&HW-CC-Date=20240920T204942Z&HW-CC-Expire=7200&HW-CC-Sign=0A8C4F25E03D6F3EF144D5F7818B4993B815F1089F825E10DC50C6C394F773B7',
    filePath: context.cacheDir + '/test.zip'
  })
  task.on('progress', (current, total) => {
    // cb((current / total * 100).toFixed(2) + '%')
    emitter.emit('worker',{
      data:{
        progress:(current / total * 100).toFixed(2) + '%'
      }
    })
    if(current === total){
      workerPort.close()
    }
  })
}
```

TaskPool和Worker的实现特点对比

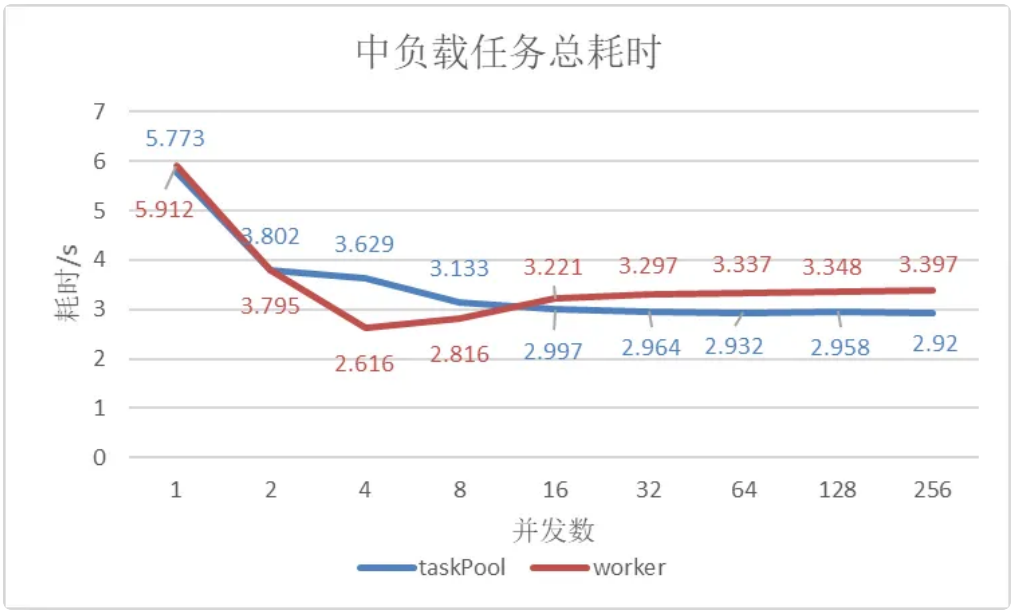
特点对比：

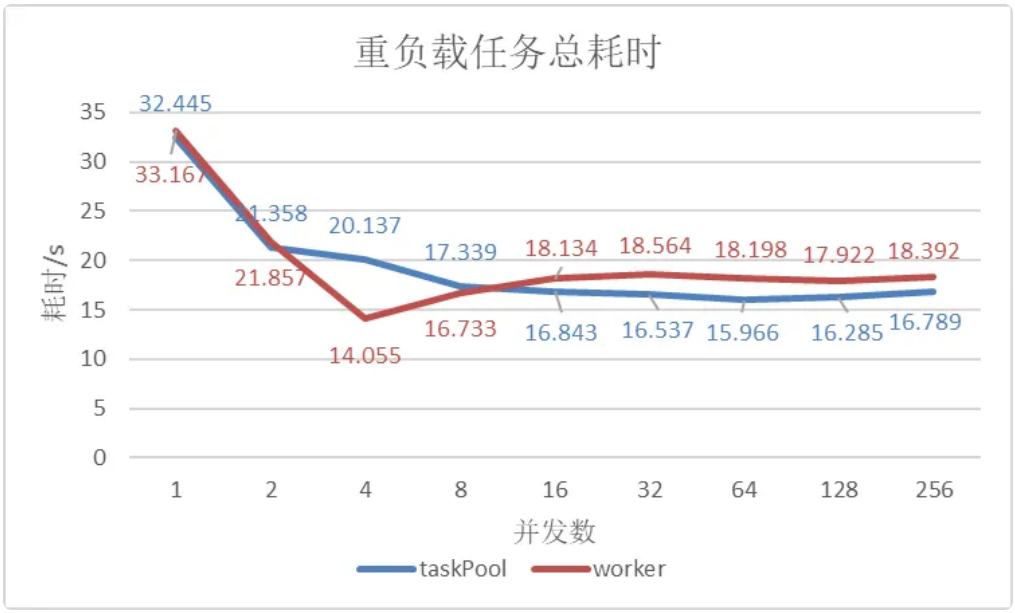
实现	TaskPool	Worker
内存模型	线程间隔离，内存不共享。	线程间隔离，内存不共享。
参数传递机制	采用标准的结构化克隆算法（Structured Clone）进行序列化、反序列化，完成参数传递。支持ArrayBuffer转移和SharedArrayBuffer共享。	采用标准的结构化克隆算法（Structured Clone）进行序列化、反序列化，完成参数传递。支持ArrayBuffer转移和SharedArrayBuffer共享。
参数传递	直接传递，无需封装，默认进行transfer。	消息对象唯一参数，需要自己封装。

方法调用	直接将方法传入调用。	在Worker线程中进行消息解析并调用对应方法。
返回值	异步调用后默认返回。	主动发送消息，需在onmessage解析赋值。
生命周期	TaskPool自行管理生命周期，无需关心任务负载高低。	开发者自行管理Worker的数量及生命周期。
任务池个数上限	自动管理，无需配置。**（补充官方文档：TaskPool最多可以创建（内核数-1）个线程，对于8核的手机来说最多可以创建7个线程，其中有一个任务串行执行）**	同个进程下，最多支持同时开启64个Worker线程，实际数量由进程内存决定。
任务执行时长上限	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），长时任务无执行时长上限。	无限制。
设置任务的优先级	支持配置任务优先级。	不支持。
执行任务的取消	支持取消已经发起的任务。	不支持。
线程复用	支持。	不支持。
任务延时执行	支持。	不支持。
设置任务依赖关系	支持。	不支持。
串行队列	支持。	不支持。
任务组	支持。	不支持。

性能对比（官方数据）

耗时对比



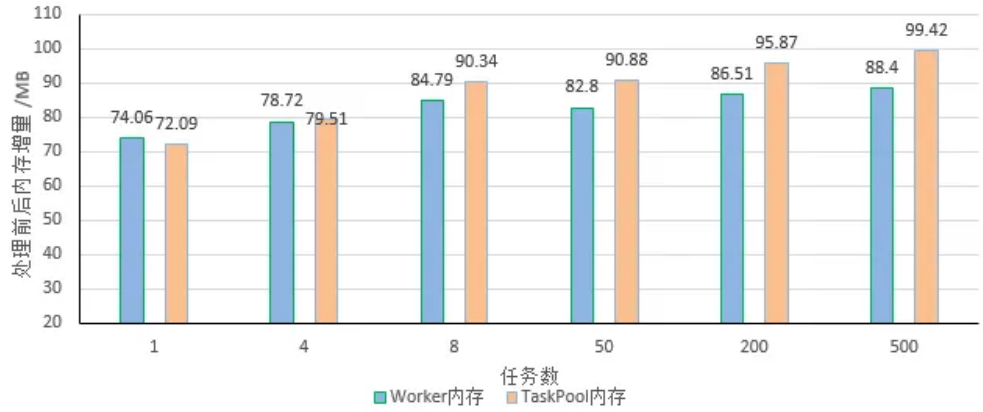


从模型实验数据可以看出：

1. 在并发任务数为1时，执行完任务TaskPool与Worker均相近；随着并发任务数的增多，TaskPool的完成任务的耗时大致上逐渐缩短，而Worker则先下降再升高。；
2. 在任务数为4时，Worker效率最高，相比于单任务减少了约57%的耗时；
3. TaskPool在并发数>8后优于Worker并趋于稳定，相比于单任务减少了约50%的耗时。

内存对比

中载模型下TaskPool与Worker运行时内存占用对比



重载模型下TaskPool与Worker运行时内存占用对比

从以上实验数据可以看出：
任务数较少时使用Worker与TaskPool的运行内存差别不大，随着任务数的增多TaskPool的运行内存明显比Worker大。
这是由于TaskPool在Worker之上做了更多场景化封装，TaskPool实现了调度器和Worker线程池，随着任务数的增多，运行时会多占用一些内存空间，待任务执行完毕之后都会进行回收和释放。

小结：

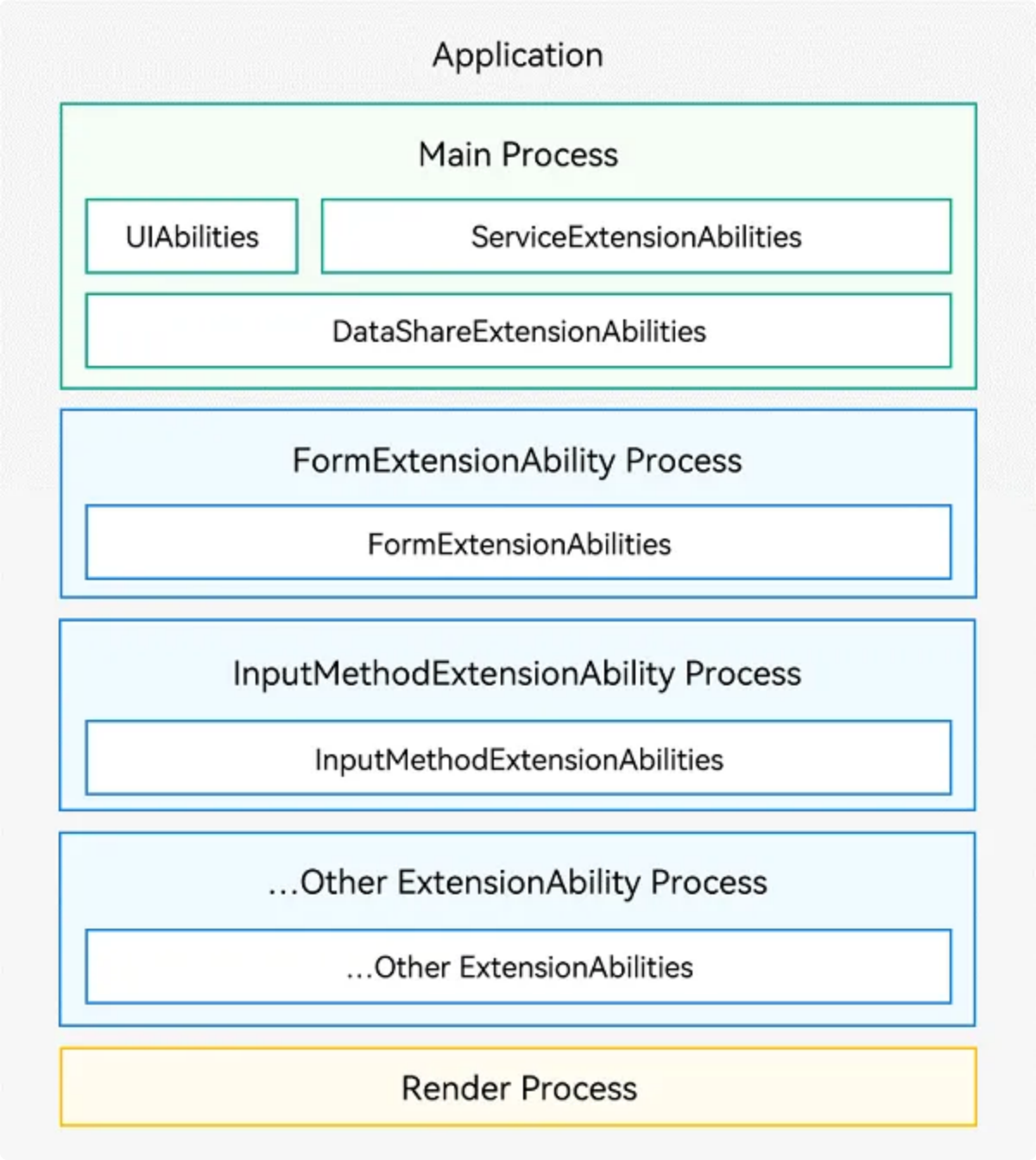
对比维度	Worker	TaskPool
编码效率	Worker需要开发者关注线程数量的上限，管理线程生命周期，随着任务的增多也会增加线程管理的复杂度。	TaskPool简单易用，开发者很容易上手。

数据传输	TaskPool与Worker都具有转移控制权、深拷贝两种方式，Worker不支持任务方法的传递，只能将任务方法写在Worker.js文件中。	传输方式与Worker相同；TaskPool支持任务方法的传递，因此相较于Worker，TaskPool多了任务方法的序列化与反序列化步骤。数据传输两者差异不大。
任务执行耗时	任务数较少时优于TaskPool，当任务数大于8后逐渐落后于TaskPool	任务数较少时劣于Worker，随着任务数的增多，TaskPool的高优先级任务模式能够更容易的抢占到系统资源，因此完成任务耗时比Worker少。
运行时内存占用	运行时占用内存较少。	随着任务数的增多占用内存比Worker高。

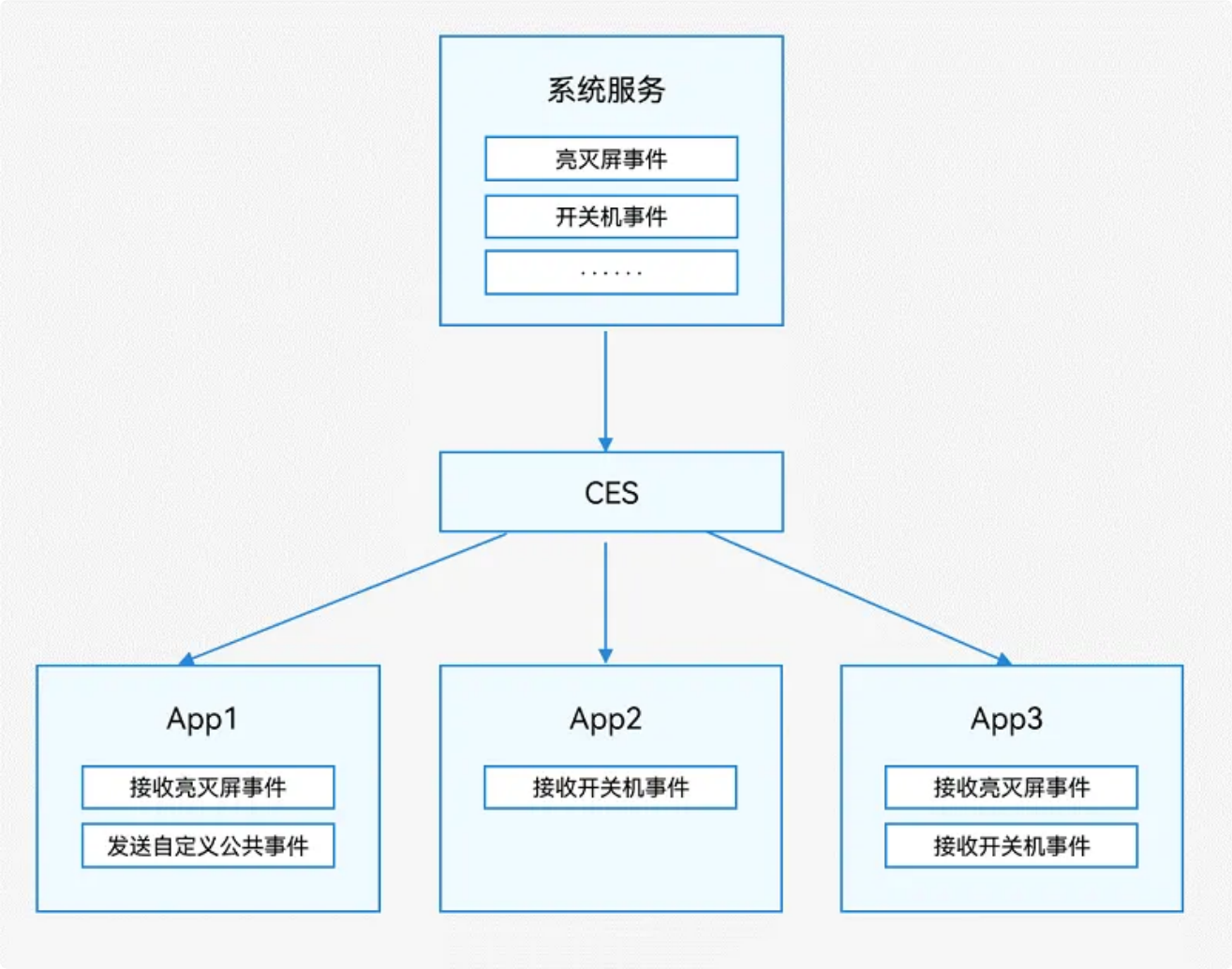
进程模型

系统的进程模型如下图所示

- 应用中（同一Bundle名称）的所有UIAbility、ServiceExtensionAbility和DataShareExtensionAbility均是运行在同一个独立进程（主进程）中，如下图中绿色部分的“Main Process”。
- 应用中（同一Bundle名称）的所有同一类型ExtensionAbility（除ServiceExtensionAbility和DataShareExtensionAbility外）均是运行在一个独立进程中，如下图中蓝色部分的“FormExtensionAbility Process”、“InputMethodExtensionAbility Process”、其他ExtensionAbility Process。
- WebView拥有独立的渲染进程，如下图中黄色部分的“Render Process”。


Application
Main Process
UIAbilities
ServiceExtensionAbilities
DataShareExtensionAbilities
FormExtensionAbility Process
FormExtensionAbilities
InputMethodExtensionAbility Process
InputMethodExtensionAbilities
...Other ExtensionAbility Process
...Other ExtensionAbilities
Render Process

公共事件机制



通过一个应用向系统发送事件从而影响其他应用

系统定义公共事件

卡片与应用通信

通过应用发送事件，通过卡片接收

通过卡片发送事件，通过应用接收

准备一个发布订阅工具

```ArkTS
import commonEventManager from '@ohos.commonEventManager'

class SubscriberClass {
  subscriber?: commonEventManager.CommonEventSubscriber
  publishCount: number = 0

  publish(eventType: string, data: string = '') {
    commonEventManager.publish(eventType, { data }, (err) => {
    })
  }
  subscribe(eventType: string, callback: (event: string) => void) {
    // 1.创建订阅者
    commonEventManager.createSubscriber({ events: [eventType] }, (err, dat
a) => {
      if (err) {
        return console.log('logData:', '创建订阅者失败')
      }
      // 2.data是订阅者
      this.subscriber = data
      if (this.subscriber) {
        // 3.订阅事件
        commonEventManager.subscribe(this.subscriber, (err, data) => {
          if (err) {
            return console.log('logData:', '订阅者事件失败')
          }
          if (data.data) {
            callback(data.data)
          }
        })
      }
    })
  }
}

export const subscriberClass = new SubscriberClass()
```

1.应用发

ArkTS

```
        Button('测试通知卡片')
          .onClick(()=>{
            subscriberClass.publish('cardUpdate','time')
          })
```

2.卡片收

卡片ability ArkTS

```
  onAddForm(want: Want) {
    // Called to return a FormBindingData object.
    let formData = '';
    let formId:string = want.parameters![formInfo.FormParam.IDENTITY_KEY]
as string
    subscriberClass.subscribe('cardUpdate',(event)=>{
      switch (event){
        case 'time':
          formProvider.updateForm(formId,formBindingData.createFormBinding
Data({
            time:Date.now()
          }))
      }
    })
    return formBindingData.createFormBindingData(formData);
  }
```

卡片页面 ArkTS

```
const localStorage = new LocalStorage()

@Entry(localStorage)
@Component
struct WidgetCard {
  @LocalStorageProp('time')
  time: number = 0

  build() {
    Column() {
      Text(this.time + '')
        .fontSize($r('app.float.font_size'))
        .fontWeight(FontWeight.Medium)
        .fontColor($r('app.color.item_title_font'))
    }
    .width('100%')
    .height('100%')
    .justifyContent(FlexAlign.Center)
  }
}
```

3.卡片发

卡片页面 ArkTS

```
const localStorage = new LocalStorage()

@Entry(localStorage)
@Component
struct WidgetCard {
  @LocalStorageProp('time')
  time: number = 0

  build() {
    Column() {
      Text(this.time + '')
        .fontSize($r('app.float.font_size'))
        .fontWeight(FontWeight.Medium)
        .fontColor($r('app.color.item_title_font'))
      Button('测试通知应用')
        .onClick(()=>{
          postCardAction(this,{
            'action':'message'
          })
        })
    }
    .width('100%')
    .height('100%')
    .justifyContent(FlexAlign.Center)
  }
}
```

卡片ability ArkTS

```
  onFormEvent(formId: string, message: string) {
    // Called when a specified message event defined by the form provider i
s triggered.
    subscriberClass.publish('appUpdate',Date.now().toString())
  }
```

4.页面收

```ArkTS
 aboutToAppear(): void {
   subscriberClass.subscribe('appUpdate',(event)=>{
      this.time = event
   })
 }

Text(this.time)
```

进程间通信服务(优化)

采用@ohos.rpc (RPC通信)改造卡片应用通信方案

卡皮发:

```ArkTS
postCardAction(this, {
     action: 'call',
     abilityName: 'EntryAbility',
     params: {
       method: 'time'
     }
   })
```

应用收:

```EntryAbility ArkTS
onCreate(want: Want, launchParam: AbilityConstant.LaunchParam): void {
    hilog.info(0x0000, 'testTag', '%{public}s', 'Ability onCreate');
    this.callee.on("time", (data) => {
      //TODO:拿到数据可以进行响应的操作
      return new Params() // 只是为了不报错
    })
 }
```