Flutter异步编程之isolate

小瓶子 Zgp

1.Dart单线程异步编程模型

在开发中，我们经常会遇到一些耗时的操作需要完成，比如网络请求，上传和下载，文件读取等等；如果在主线程中一直等待这些耗时操作完成，就会发生阻塞，无法响应如用户点击等操作。

1.1 处理耗时操作

1. 多线程，比如Java、C++，OC，我们普遍的做法是开启一个新的线程（Thread），在新的线程中完成这些异步的操作，再通过线程间通信的方式，将拿到的数据传递给主线程。
2. 单线程+事件循环，比如JavaScript、Dart都是基于单线程加事件循环来完成耗时操作的处理。

单线程是如何来处理网络通信、IO操作它们返回的结果呢?答案就是事件循环

1.2 Dart事件循环

单线程模型中主要就是在维护着一个事件循环（Event Loop）。

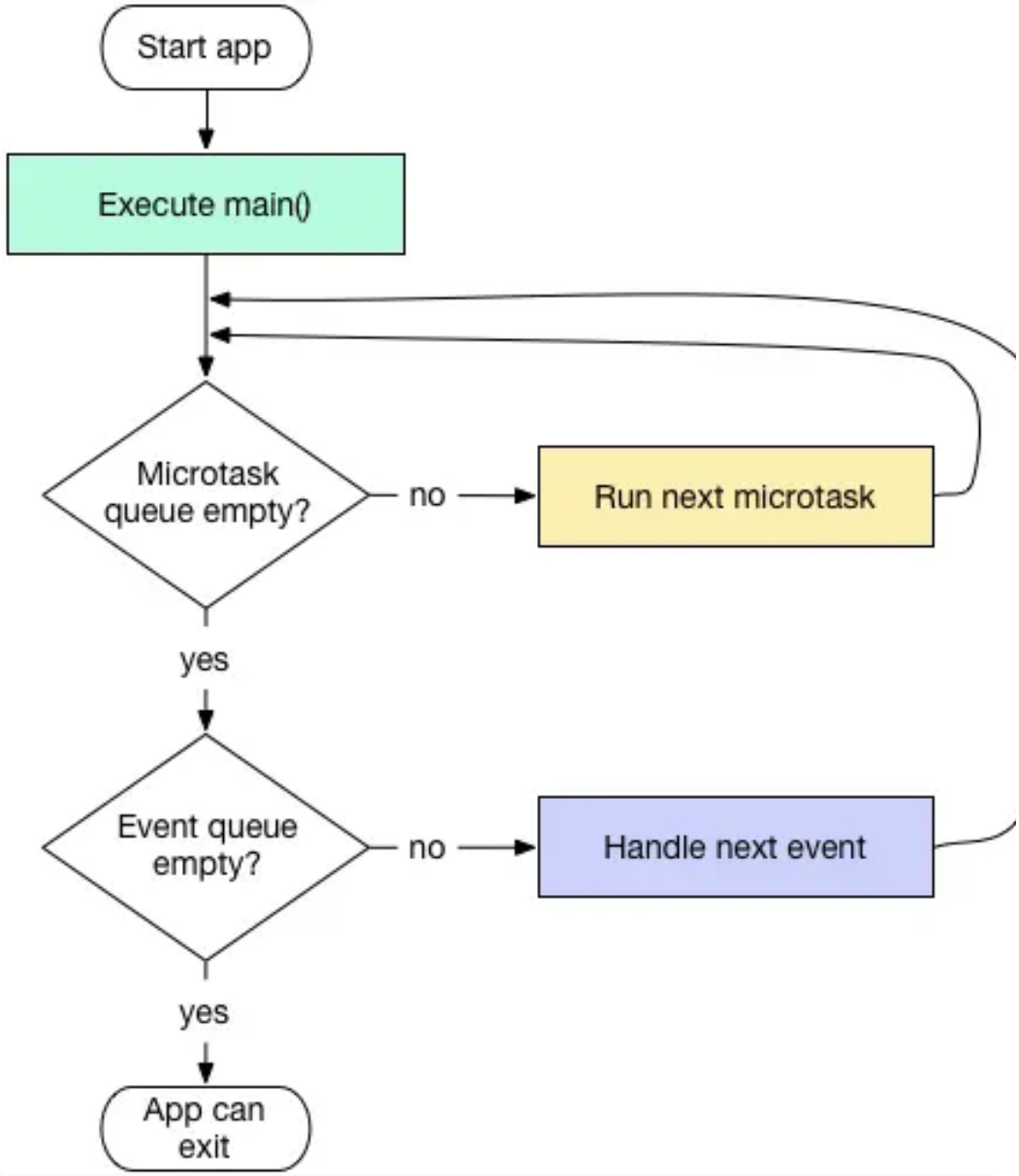
- 将需要处理的一系列事件（包括点击事件、IO事件、网络事件）放在一个事件队列（Event Queue）中。
- 不断的从事件队列（Event Queue）中取出事件，并执

行其对应需要执行的代码块，直到事件队列清空位置。严格来划分的话，在Dart中还存在另一个队列：微任务队列（Microtask Queue）。

- 微任务队列的优先级要高于事件队列，也就是说事件循环都是优先执行微任务队列中的任务，再执行 事件队列 中的任务；
- 所有的外部事件任务都在事件队列中，如IO、计时器、点击、以及绘制事件等；
- 而微任务通常来源于Dart内部，并且微任务非常少。这是因为如果微任务非常多，就会造成事件队列排不上队，会阻塞任务队列的执行（比如用户点击没有反应的情况）；

在Dart的单线程中，代码执行顺序：

1. Dart的入口是main函数，所以main函数中的代码会优先执行；
2. main函数执行完后，会启动一个事件循环（Event Loop）就会启动，启动后开始执行队列中的任务；
3. 首先，会按照先进先出的顺序，执行 微任务队列（Microtask Queue）中的所有任务；
4. 其次，会按照先进先出的顺序，执行 事件队列（Event Queue）中的所有任务；



如果在多核 CPU 中，单线程是不是就没有充分利用 CPU 呢?

2. isolate

❝ 我们知道 Dart 是单线程模型，也就是实现异步需要借助 EventLoop 来进行事件驱动。所以 Dart 只有一个主线程，其实在 Dart 中并不是叫 Thread ,而是有个专门名词叫 **「isolate(隔离)。其实在 Dart 也会遇到一些耗时计算的任**

> 务，不建议把任务放在主 isolate 中，否则容易造成 UI 卡顿，需要开辟一个单独 isolate 来独立执行耗时任务，然后通过消息机制把最终计算结果发送给主 isolate 实现 UI 的更新。」 在 Dart 中异步是并发方案的基础，Dart 支持单个和多个 isolate 中的异步。
>
> ”

2.1 为什么需要 isolate

在 Dart/Flutter 应用程序启动时，会启动一个主线程其实也就是 Root Isolate, 在 Root Isolate 内部运行一个 EventLoop 事件循环。所以所有的 Dart 代码都是运行在 Isolate 之中的，它就像是机器上的一个小空间，具有自己的私有内存块和一个运行事件循环的单个线程。isolate 是提供了 Dart/Flutter 程序运行环境，包括所需的内存以及事件循环 EventLoop 对事件队列和微任务队列的处理。

2.2 什么是 isolate

用官方文档中定义一句话来概括: An isolated Dart execution context .大概的意思就是 **「isolate 实际就是一个隔离的 Dart 执行的上下文环境 (或者容器)」** 。isolate 是 Dart 对 **「Actor 并发模型」** 的实现。运行中的 Dart 程序由一个或多个「Actor」组成，这些「Actor」其实也就是 Dart 中的 isolate。 **「isolate 是有自己的内存和单线程控制的事件循环」** 。是一条独立的执行线，它们之间只能通过发送消息通

信，所以它的资源开销低于线程。isolate本身的意思是“隔离”，因为 **「isolate之间的内存在逻辑上是隔离的，不像Java一样是共享内存的」**。isolate中的代码是按顺序执行的，**「任何Dart程序的并发都是运行多个isolate的结果」**。因为 **「Dart没有共享内存的并发」**，没有竞争的可能性所以不需要锁，也就不存在死锁的问题。

2.3 isolate并发模型特点

isolate可以理解为是概念上Thread线程，但是它和Thread线程唯一不一样的就是 **「多个isolate之间彼此隔离且不共享内存空间，每个isolate都有自己独立内存空间，从而避免了锁竞争」**。由于每个isolate都是隔离，它们之间的通信就是 **「基于Actor并发模型中发送异步消息来实现通信的」**，所以更直观理解把一个isolate当作Actor并发模型中一个Actor即可。在isolate中还有「Port」的概念，分为`send port`和`receive port`可以把它理解为Actor模型中每个Actor内部都有对mailbox(信箱)的实现，可以很好地管理Message。

send port actor 演员

3.如何使用isolate

3.1 isolate包介绍

使用isolate类进行并发操作，需要导入 isolate

```
import 'dart:isolate';
```

该Library主要包含下面:

- `Isolate` 类: Dart代码执行的隔离的上下文环境
- `ReceivePort` 类: 它是一个接收消息的 Stream，`ReceivePort` 可以生成 `SendPort`，`ReceivePort` 接收消息，可以把消息发送给其他的 `isolate`, 所以要发送消息就需要生成 `SendPort`，然后再由 `SendPort` 发送给对应isolate的 `ReceivePort`.
- `SendPort` 类: 将消息发送给isolate, 准确的来说是将消息发送到isolate中的 `ReceivePort`

此外可以使用 `spawn` 方法生成一个新的 isolate 对象， `spawn` spawn 大量生产是一个静态方法返回的是一个 `Future<Isolate>`，必传参数有两个， 函数 `entryPoint` 和参数 `message`,其中 **entryPoint函数必须是顶层函数或静态方法，参数message需要包含 SendPort.** entryPoint 入口点

```
external static Future<Isolate> spawn<T>(
      void entryPoint(T message), T message,
      {bool paused = false,
      bool errorsAreFatal = true,
      SendPort? onExit,
      SendPort? onError,
      @Since("2.3") String? debugName});
```

3.2 isolate单向通信

```
import "dart:isolate";
```

```dart
void main() async {
  // 1.创建管道
  ReceivePort receivePort= ReceivePort();

  // 2.创建新的Isolate
  Isolate isolate = await Isolate.spawn<SendPort>(foo,
receivePort.sendPort);

  // 3.监听管道消息
  receivePort.listen((data) {
    print('单向通信Data: $data');
    // 不再使用时，我们会关闭管道
    receivePort.close();
    // 需要将isolate杀死
    isolate.kill(priority: Isolate.immediate);
  });
}

void foo(SendPort sendPort) {
  sendPort.send("Hello World");
}

// 打印
// [log] 单向通信data: Hello World
```

3.3 isolate双向通信

```dart
  /// 双向通信
```

返回值是被Future包裹的SendPort类型的数据

```
  Future<SendPort> initIsolate() async {
    log('initIsolate == ${Isolate.current.debugName}');
    Completer<SendPort> completer = Completer<SendPort>();
    //主isolate中的接收者（接收子isolate中发送的消息）
    ReceivePort mainReceivePort = ReceivePort();
    //接受者的监听
    mainReceivePort.listen((data) {
      if (data is SendPort) {
        //接收到子isolate中创建的 SendPort，可使用该SendPort向子isolate发送消息
        SendPort newSendPort = data;
        completer.complete(newSendPort); Completer 完成符
      } else {  //接收并返回被Future包裹的newReceivePort.sendPort
        log('[newIsolateToMainStream] $data');
      }
    });
    //创建子isolate，传入 入口函数 和 接受者sendPort  ，子isolate可使用该sendPort向主isolate发送消息
    Isolate newIsolateInstance =
        await Isolate.spawn(newIsolate, mainReceivePort.sendPort);
    log('newIsolateInstance == ${newIsolateInstance.debugName}');
    return completer.future;
  }

  /// 子Isolate的入口函数，可以在该函数中做耗时操作
```

```dart
static void newIsolate(SendPort mainSendPort) {
  log('newIsolate == ${Isolate.current.debugName}');
  ReceivePort newReceivePort = ReceivePort();
  mainSendPort.send(newReceivePort.sendPort);
  //子isolate使用mainReceivePort.sendPort向主isolate发送newReceivePort.sendPort

  newReceivePort.listen((data) { //子isolate通过newReceivePort监听数据
    log('[mainToNewIsolateStream] $data'); //子isolate使newReceivePort进行监听
    var sum = 0;
    for (int i = 1; i <= data; i++) {
      sum += I;
    }
    mainSendPort.send('计算结果: $sum'); // 子isolate使用mainReceivePort.sendPort向主发送计算结果
  });
}

void twoWaystart() async {
  SendPort newSendPort = await initIsolate();
  //接收到子ioslate中的 SendPort    可向子isolate中发送消息
  newSendPort.send(1000000000);
} //主isolate接收到ioslate发送的 SendPort，向isolate中发送消息1000000000


// 打印
// [log] initIsolate == main
// [log] newIsolate == newIsolate
// [log] newIsolateInstance == newIsolate
// [log] [mainToNewIsolateStream] 1000000000
// [log] [newIsolateToMainStream] 计算结果:
```

```
5000000005000000000
```

3.4 isolate的暂停、恢复、结束

```
    //恢复 isolate 的使用
    isolate.resume(isolate.pauseCapability); capability 能力

    //暂停 isolate 的使用
    isolate.pause(isolate.pauseCapability);

    //结束 isolate 的使用
    isolate.kill(priority: Isolate.immediate);
```

3.5 compute函数

```
// 创建一个新的Isolate，在其中运行任务doWork
 createNewTask() async {
    var str = 'New Task';
    var result = await compute(doWork, str);
    print(result);   函数名称，参数

    var result2 = await compute(summ, 1000000000);
    print(result2);
 }

 static String doWork(String value) {
    print('new isolate doWork start == ${DateTime.now()}');
    // 模拟耗时5秒
    sleep(const Duration(seconds: 5));
```

```
    print('new isolate doWork end == ${DateTime.now()}');
    return "complete:$value";
  }

  //计算0到 num 数值的总和
  static num summ(int num) {
    print('开始计算');
    int count = 0;
    while (num > 0) {
      count = count + num;
      num--;
    }
    print('计算结束');
    return count;
  }

// 打印
flutter: new isolate doWork start == 2022-06-21
20:37:43.615597
flutter: new isolate doWork end == 2022-06-21
20:37:48.622028
flutter: complete:New Task
flutter: 开始计算
flutter: 计算结束
flutter: 5000000005000000000
```

3.6 isolate存在的限制

Platform-Channel 通信仅仅由主 isolate 支持。该主 isolate 对应于应用启动时创建的 isolate。
也就是说，通过编程创建的 isolate 实例，无法实现 Platform-Channel 通信.....

3.7 isolate 和普通 Thread 的区别

isolate 和普通 Thread 的区别需要从不同的维度来区分:

1. 从底层操作系统维度
   在 isolate 和 Thread 操作系统层面是一样的，都是会去创建一个 OSThread，也就是说最终都是委托创建操作系统层面的线程。
2. 从所起的作用维度
   都是为了应用程序提供一个运行时环境。
3. 从实现机制的维度
   isolate 和 Thread 有着明显区别就是大部分情况下的 Thread 都是共享内存的，存在资源竞争等问题，但是 isolate 彼此之间是不共享内存的。

3.8 什么场景该使用 Future 还是 isolate

用户将根据不同的因素来评估应用的质量，比如:

- 特性
- 外观
- 用户友好性
- .....

你的应用可以满足以上所有因素，但如果用户在一些处理过程中遇到了**卡顿**，这极有可能对你不利。

因此，以下是你在开发过程中应该系统考虑的一些点：

1. 如果代码片段不能被中断，使用传统的同步过程（一个或多个相互调用的方法）；
2. 如果代码片段可以独立运行而不影响应用的性能，可以考虑通过 **Future** 使用 **事件循环**；
3. 如果繁重的处理可能需要一些时间才能完成，并且可能影响应用的性能，考虑使用 **Isolate**。

换句话说，建议尽可能地使用 **Future**（直接或间接地通过 async 方法），因为一旦事件循环拥有空闲时间，这些 **Future** 的代码就会被执行。这将使用户感觉事情正在被并行处理（而我们现在知道事实并非如此）。

另外一个可以帮助你决定使用 **Future** 或 **Isolate** 的因素是运行某些代码所需要的平均时间。

- 如果一个方法需要**几毫秒** => **Future**
- 如果一个处理流程需要**几百毫秒** => **Isolate**

以下是一些很好的 **Isolate** 选项：

- **JSON** 解码：解码 JSON（HttpRequest 的响应）可能需要一些时间 => 使用 **compute**
- 加密：加密可能非常耗时 => **Isolate**
- 图像处理：处理图像（比如：剪裁）确实需要一些时间来完成 => **Isolate**

- 从 Web 加载图像：该场景下，为什么不将它委托给一个完全加载后返回完整图像的 **Isolate**?