

# Flutter异步编程之 Future/Isolate

哈啰于先生 [关注](#)

2021.01.27 21:15:51 字数 2,573 阅读 1,419

## 前言

Flutter是用Dart实现的，在Dart中没有线程和进程的概念，我们编程使用多线程一般实现两种场景，一种是异步执行，一种是并行执行。那么如何在Flutter上实现异步编程呢。本篇文章将主要讨论以下问题：

- 1、Dart如何实现异步编程？
- 2、Event Loops是什么？
- 3、Isolate是什么呢？
- 4、如何实现Isolate？
- 5、Isolate底层原理是什么？

## 同步和异步

我们在写Dart代码的时候，就只有两种代码，

同步代码：就是一行行写下来的代码

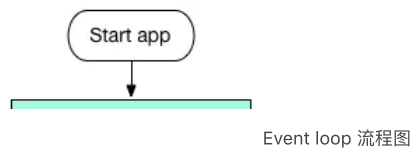
异步代码：就是以Future等修饰的代码

并不是指的我们平常异步，这两种代码的区别只有一个：代码运行的顺序是不同的，先运行同步代码，再运行异步代码，顺序执行。

而异步代码是运行在Event loop里的，Event loops就是事件循环机制。



这个很好理解，事件events加到Event queue里，Event loop循环从Event queue里取Event执行。



从这里看到，启动app（start app）后：

- 先查看MicroTask queue是不是空的，不是的话，先运行microtask
- 一个microtask运行完后，会看有没有下一个microtask，直到Microtask queue空了之后，才会去运行Event queue
- 在Event queue取出一个event task运行完后，又会跑到第一步，去运行microtask

这里多了两个名词：MicroTask和Event，这代表了两个不同的异步task而且可以看出：

如果能让任务能够尽快执行，就用MicroTask。

## MicroTask

是 `dart:async` 提供的异步方法，主要实现在 `schedule_microtask.dart` 中，使用方式

```

1 | scheduleMicrotask((){
2 |   // ...code goes here...
3 | });
  
```

或者

```

1 | new Future.microtask((){
2 |   // ...code goes here...
3 | });
  
```

## Event

Event我们就很熟悉的，就是Future修饰的异步方法，使用方式

```

1 | new Future(() {
2 |   // ...code goes here...
3 | });
  
```

### 对 Future 的理解

- 1、Future对象（futures）表示异步操作的结果，进程或者IO会延迟完成
- 2、可以在async函数中使用await来挂起执行，返回一个Future对象
- 3、在async函数中使用try-catch来捕获异常（或者使用catchError()）
- 4、await只能在async中使用

### Future示例代码

```

1 | // Future示例代码:
2 |
3 | void futureTest() {
4 |   print("future start");
5 |   Future.wait([
6 |     // 2秒后返回结果
7 |     Future.delayed(new Duration(seconds: 2), () {
8 |       print("hello");
9 |       return "hello";
10 |     }),
11 |     // 4秒后返回结果
12 |     Future.delayed(new Duration(seconds: 4), () {
13 |       print("world");
14 |       return " world";
  
```

```

15     }),
16     // 4秒后返回结果
17     Future.delayed(new Duration(seconds: 6), () {
18         print("!");
19         return " !";
20     })
21   ]).then((results) {
22   // 上面的两个任务执行完毕后进入
23     print("future finish");
24   }).catchError((e) {
25   // 执行失败会走到这里
26     print(e);
27   }).whenComplete(() {
28   // 无论成功或失败都会走到这里
29     });
30 }
31
32 /// 打印结果
33 future start
34 hello
35 world
36 !
37 future finish

```

Event loop示例代码，根据Event loop的执行流程，请问如下代码打印顺序是什么样的？

```

1 void eventLoopTest() {
2   print('eventLoopTest #1 of 2');
3   scheduleMicrotask(() => print('microtask #1 of 3'));
4   //使用delay方式，是将此task放到queue的尾部，
5   //若前面有耗时操作，不一定能准时执行
6   new Future.delayed(
7     new Duration(seconds: 1), () => print('future #1 (delayed)'));
8   //使用then，是表示在此task执行后立刻执行
9   new Future(() => print('future #2 of 4'))
10    .then(() => print('future #2a'))
11    .then(() {
12      print('future #2b');
13      scheduleMicrotask(() => print('microtask #0 (from future #2b)'));
14    }).then(() => print('future #2c'));
15
16   scheduleMicrotask(() => print('microtask #2 of 3'));
17
18   new Future(() => print('future #3 of 4'))
19    .then(() => new Future(() => print('future #3a (a new future)')))
20    .then(() => print('future #3b'));
21
22   new Future(() => print('future #4 of 4')).then(() {
23     new Future(() => print('future #4a'));
24   }).then(() => print('future #4b'));
25
26   scheduleMicrotask(() => print('microtask #3 of 3'));
27   print('eventLoopTest #2 of 2');
28 }
29
30 //打印结果
31 isolateTest #1 of 2
32 isolateTest #2 of 2
33 microtask #1 of 3
34 microtask #2 of 3
35 microtask #3 of 3
36 future #2 of 4
37 future #2a
38 future #2b
39 future #2c
40 microtask #0 (from future #2b)
41 future #3 of 4
42 future #4 of 4
43 future #4b
44 future #3a (a new future)
45 future #3b
46 future #4a
47 future #1 (delayed)

```

### 关键点

1、Future.delayed需要延迟执行的，是在延迟时间到了之后才将此task加到event queue的队尾，所以万一前面有很耗时的任务，那么你的延迟task不一定能准时运行。

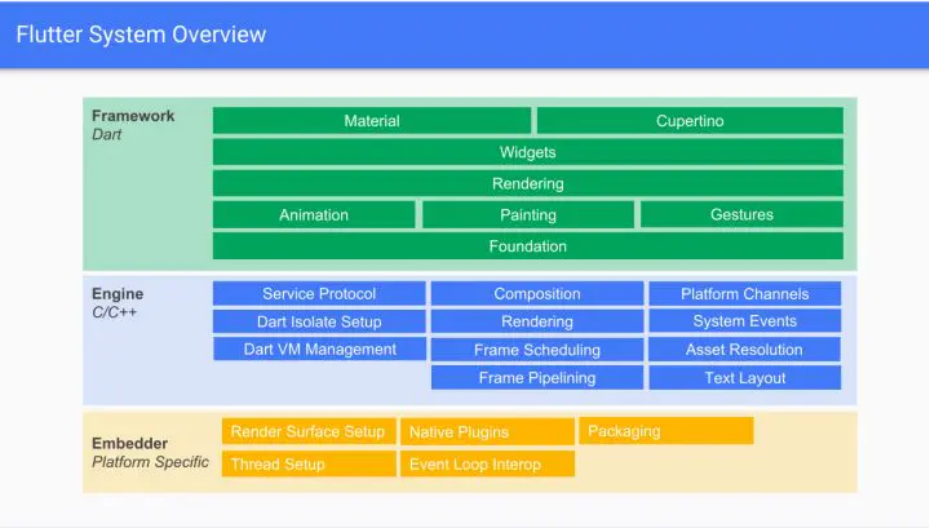
2、Future.then每次都会返回一个Future，默认是其本身。如果在then中函数也返回一个新的Future，则新Future会

重新加入到event queue中等待执行

3、一个event task运行完后，会先去查看Micro queue里有没有可以执行的micro task。没有的话，在执行下一个event task

我们知道 Dart 是单线程异步编程模型，Future解决了异步执行的问题。但是并行执行怎么处理呢？

Flutter引擎架构



Flutter体系图

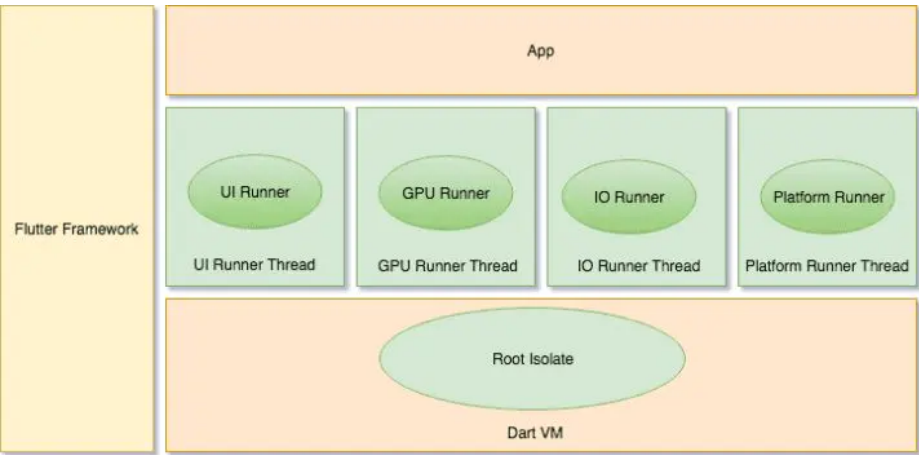
我们只关注线程相关信息

1、Framework:我们直接接触的层级

2、Engine： Dart Isolate Setup, 创建Isolate，类似于DartVM中的线程，他的架构就是一个循环：event loops。但这一层并不创建及管理线程，它要求Embeder提供四个Task Runner，类似于线程，并不是真正的线程。

3、Embedder： Thread Setup，真正的线程创建及管理，Embeder指的是将引擎移植到平台的中间层代码。

Task runners



Task runner

Embedder将自己管理的线程作为 task runner 提供给Flutter 引擎。

主要的 task runner 有：

Platform Task Runner

UI Task Runner  
GPU Task Runner  
IO Task Runner

## Isolate

Dart是一个单线程语言，它的"线程"概念被称为 **Isolate**，中文意思是隔离。

- 特点：

- 1、它与我们之前理解的 **Thread** 概念有所不同，各个 **isolate** 之间是无法共享内存空间。
- 2、**Isolate**是完全是独立的执行线，每个都有自己的 **event loop**。只能通过 **Port** 传递消息，所以它的资源开销低于线程。
- 3、**Dart**中的线程可以理解为微线程。
- 4、**Future**实现异步串行多个任务；**Isolate**可以实现异步并行多个任务

- 作用：

**Flutter**的代码都是默认跑在**root isolate**上的，将非常耗时的任务添加到**event loop**后，会拖慢整个事件循环的处理，甚至是阻塞。可见基于**Event loop**的异步模型仍然是有很大缺点的，这时候我们就需要**Isolate**。

- 使用场景

Dart中使用多线程计算的时候，在创建Isolate以及线程间数据传递中耗时要超过单线程，每当我们创建出来一个新的 Isolate 至少需要 2mb 左右的空间甚至更多，因此Isolate有合适的使用场景，不建议滥用Isolate。那么应该在什么时候使用Future，什么时候使用Isolate呢？

一个最简单的判断方法是根据某些任务的平均时间来选择：

方法执行在几毫秒或十几毫秒左右的，应使用Future

如果一个任务需要几百毫秒或之上的，则建议创建单独的Isolate

一些常见的可以参考的场景

JSON 解码

数据加密

图像处理

网络请求：加载资源、图片

## 如何使用Isolate

Isolate由一对Port分别由用于接收消息的 **ReceivePort** 对象，和用于发送消息的 **SendPort** 对象构成。其中 **SendPort** 对象不用单独创建，它已经包含在 **ReceivePort** 对象之中。需要注意，一对Port对象只能单向发消息，这就如同一根自来水管，**ReceivePort** 和 **SendPort** 分别位于水管的两头，水流只能从 **SendPort** 这头流向 **ReceivePort** 这头。因此，两个Isolate之间的消息通信肯定是需要两根这样的水管的，这就需要两对Port对象。

### 1、Dart中创建

我们可以通过 **Isolate.spawn** 创建一个 isolate。

```
1 | static Future<Isolate> spawn<T>(void entryPoint(T message),T message);
```

当我们调用 **Isolate.spawn** 的时候，它将会返回一个对 isolate 的引用的 Future。我们可以通过这个 isolate 来控制创建出的 Isolate，例如 **pause**、**resume**、**kill** 等等。

- **entryPoint**：这里传入我们想要在其他 isolate 中执行的方法，入参是一个任意类型的 **message**。**entryPoint** 只能是顶层方法或静态方法，且返回值为 **void**。
- **message**：创建 Isolate 第一个调用方法的入参，可以是任意值。

但是在此之前我们必须创建两个 isolate 之间沟通的桥梁。

```

1  import 'dart:isolate';
2  import 'dart:io';
3
4  void main() {
5      print("main isolate start");
6      create_isolate();
7      print("main isolate end");
8  }
9
10 // 创建一个新的 isolate
11 create_isolate() async{
12     ReceivePort rp = new ReceivePort();
13     SendPort port1 = rp.sendPort;
14
15     Isolate newIsolate = await Isolate.spawn(doWork, port1);
16
17     SendPort port2;
18     rp.listen((message){
19         print("main isolate message: $message");
20         if (message[0] == 0){
21             port2 = message[1];
22         }else{
23             port2?.send([1, "这条信息是 main isolate 发送的"]);
24         }
25     });
26 }
27
28 // 处理耗时任务
29 static void doWork(SendPort port1){
30     print("new isolate start");
31     ReceivePort rp2 = new ReceivePort();
32     SendPort port2 = rp2.sendPort;
33
34     rp2.listen((message){
35         print("doWork message: $message");
36     });
37
38     // 将新isolate中创建的SendPort发送到主isolate中用于通信
39     port1.send([0, port2]);
40     // 模拟耗时5秒
41     sleep(Duration(seconds:5));
42     port1.send([1, "doWork 任务完成"]);
43
44     print("new isolate end");
45 }
46
47 //运行结果
48 main isolate start
49 main isolate end
50 new isolate start
51 main isolate message: [0, SendPort]
52 new isolate end
53 main isolate message: [1, doWork 任务完成]
54 doWork message: [1, 这条信息是 main isolate 发送的]

```

运行后都会创建一个新Isolate的微进程，新的Isolate和主Isolate都双向绑定了消息通信的通道，即使新的Isolate中的任务完成了，它的微进程也不会立刻退出，因此，当使用完自己创建的Isolate后，最好调用 `newIsolate.kill(priority: Isolate.immediate)`；将Isolate立即杀死。

## 2、Flutter中创建

如果想在Flutter中创建Isolate，则有更简便的API，这是由Flutter官方进一步封装 `ReceivePort` 而提供的更简洁API。使用 `compute` 函数来创建新的Isolate并执行耗时任务。

```

1  import 'package:flutter/foundation.dart';
2  import 'dart:io';
3
4  // 创建一个新的Isolate，在其中运行任务doWork
5  create_new_task() async{
6      var str = "New Task";
7      var result = await compute(doWork, str);
8      print(result);
9  }
10
11 static String doWork(String value){
12     print("new isolate doWork start");
13     // 模拟耗时5秒
14     sleep(Duration(seconds:5));

```

```

15 |   print("new isolate doWork end");
16 |   return "complete:$value";
17 | }

```

`compute` 函数有两个必须的参数，第一个是待执行的函数，这个函数必须是一个顶级函数或静态方法，不能是类的实例方法，第二个参数为动态的消息类型，可以是被运行函数的参数。需要注意，使用 `compute` 应导入 `'package:flutter/foundation.dart'` 包。

## 实现线程管理器

- 线程管理器ThreadManagment

```

1 | import 'dart:isolate';
2 | typedef LikeCallback = void Function(Object value);
3 |
4 | class ThreadManagement {
5 |   //entryPoint 必须是静态方法
6 |   static Future<Map> runTask (void entryPoint(SendPort message), LikeCallback(Object value),
7 |     final response = ReceivePort();
8 |     Isolate d = await Isolate.spawn(entryPoint, response.sendPort);
9 |     // 调用sendReceive自定义方法
10 |    if(parameter!=null){
11 |      SendPort sendPort = await response.first;
12 |      ReceivePort receivePort = ReceivePort();
13 |      sendPort.send([parameter, receivePort.sendPort]);
14 |      receivePort.listen((value){
15 |        receivePort.close();
16 |        d.kill();
17 |        LikeCallback(value);
18 |      });
19 |      return {
20 |        'isolate': d,
21 |        "receivePort":receivePort,
22 |      };
23 |    }else{
24 |      response.listen((value){
25 |        response.close();
26 |        d.kill();
27 |        LikeCallback(value);
28 |      });
29 |      return {
30 |        'isolate': d,
31 |        "receivePort":response,
32 |      };
33 |    }
34 |  }
35 | }

```

- 耗时的任务

```

1 | // 无参数的任务
2 | static void getNoParamTask(SendPort port) async {
3 |   var c = await Future.delayed(Duration(seconds: 1), () {
4 |     return "banner data";
5 |   });
6 |   port.send(c);
7 | }
8 |
9 | // 需要参数的任务
10 | static getParamsTask(SendPort port) async {
11 |   ReceivePort receivePort = ReceivePort();
12 |   port.send(receivePort.sendPort);
13 |   // 监听外界调用
14 |   await for (var msg in receivePort) {
15 |     Map requestURL =msg[0];
16 |     SendPort callbackPort =msg[1];
17 |     receivePort.close();
18 |     var res = await Future.delayed(Duration(seconds: 1), () {
19 |       var requestUrl = requestURL["type"];
20 |       var after = requestURL["after"];
21 |       return "url = $requestUrl, after = $after";
22 |     });
23 |     callbackPort.send(res);
24 |   }
25 | }

```

- 执行耗时任务

```

1 | // 调用无参数的任务
2 | ThreadManagement.runTask(API.getNoParamTask, (value){
3 |     if(value != null){
4 |         //业务逻辑
5 |         print(value);
6 |     }
7 | });
8 |
9 | //调用有参数的任务
10 | ThreadManagement.runTask(API.getParamsTask, (value){
11 |     if(value != null){
12 |         //业务逻辑
13 |         print(value);
14 |     }
15 | }, parameter: {
16 |     "type": "hot",
17 |     "after": "1"
18 | });

```

## 线程池

如何减少 isolate 创建所带来的消耗。我们可以创建一个线程池，初始化到那里。当我们需要使用的时候再拿来用就好了。

**LoadBalancer** 是 dart team 已经为我们写好一个非常实用的 package。

我们现在在 `pubspec.yaml` 中添加 isolate 的依赖。

```
1 | isolate: ^2.0.3
```

我们可以通过 **LoadBalancer** 创建出指定个数的 isolate。

```
1 | Future<LoadBalancer> loadBalancer = LoadBalancer.create(2, IsolateRunner.spawn);
```

这段代码将会创建出一个 isolate 线程池，并自动实现了负载均衡。

下面我们再看看应该如何使用 **LoadBalancer** 中的 isolate。

```

1 | void testBalancer() async {
2 |     final lb = await loadBalancer;
3 |     int res = await lb.run(doWork, 110);
4 |     print(res);
5 | }
6 |
7 | int doWork(int value) {
8 |     // 模拟耗时5秒
9 |     print("new isolate doWork start");
10 |    sleep(Duration(seconds: 5));
11 |    return value;
12 | }
13 |
14 | //打印数据
15 | new isolate doWork start
16 | 110

```

我们关注的只有

```
1 | Future<R> run<R, P>(FutureOr<R> function(P argument), argument,)
```

方法。我们还是需要传入一个 function 在某个 isolate 中运行，并传入其参数 `argument`。run 方法将会返回我们执行方法的返回值。

整体和 `compute` 使用上差不多，但是当我们多次使用额外的 isolate 的时候，不再需要重复创建了。

并且 **LoadBalancer** 还支持 `runMultiple`，可以让一个方法在多线程中执行。

**LoadBalancer** 经过测试，它会在第一次使用其 isolate 的时候初始化线程池。




当应用打开后，即使我们在顶层函数中调用了 `LoadBalancer.create`，但是还是只会有一个 Isolate。

当我们调用 `run` 方法时，才真正创建出了实际的 isolate。

参考文章：

- [Dart 函数、箭头函数、匿名函数、立即执行函数及闭包](#)
- [Flutter中的异步编程——Future](#)
- [Flutter进阶Future异步详解](#)
- [Flutter异步编程](#)
- [深入了解Flutter的isolate](#)
- [Flutter/Dart中的异步编程之Isolate isolate](#)

 1人点赞 >



 Flutter





哈啰于先生

总资产1 共写了1.8W字 获得14个赞 共12个粉丝

关注