

# Flutter Isolate

Flutter 的代码都是默认跑在 root Isolate 上的,那么 Flutter 中能不能自己创建一个 Isolate 呢? 当然可以! , 接下来我们就自己创建一个 Isolate!

## 创建自己的 Isolate

### dart:isolate

有关 Isolate 的代码, 都在isolate.dart文件中, 里面有一个生成 Isolate的方法:

```
external static Future<Isolate> spawn<T>(
    void entryPoint(T message), T message,
    {bool paused: false,
    bool errorsAreFatal,
    SendPort onExit,
    SendPort onError});
```

spawn 方法, 必传参数有两个, 函数 entryPoint 和参数 message, 其中

#### 1. 函数

函数必须是顶级函数或静态方法

#### 2. 参数

参数里必须包含 SendPort

# 开始动手写

创建的步骤，写在代码的注释里

```
import 'dart:async';
import 'dart:io';
import 'dart:isolate';

import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

//一个普普通通的Flutter应用的入口
//main函数这里有async关键字，是因为创建的isolate是异步的
void main() async{
  runApp(MyApp());

  //asyncFibonacci函数里会创建一个isolate，并返回运行结果
  print(await asyncFibonacci(20));
}

//这里以计算斐波那契数列为例，返回的值是Future，因为是异步的
Future<dynamic> asyncFibonacci(int n) async{
  //首先创建一个ReceivePort，为什么要创建这个？
  //因为创建isolate所需的参数，必须要有SendPort，SendPort需要ReceivePort来创建
  final response = new ReceivePort();
  //开始创建isolate,Isolate.spawn函数是isolate.dart里的代码，_isolate是我们自己实现的函数
  //_isolate是创建isolate必须有的参数。
  await
```

```
Isolate.spawn(_isolate, response.sendPort);
    //获取sendPort来发送数据
    final sendPort = await response.first as
SendPort;
    //接收消息的ReceivePort
    final answer = new ReceivePort();
    //发送数据
    sendPort.send([n, answer.sendPort]);
    //获得数据并返回
    return answer.first;
}

//创建isolate必须有的参数
void _isolate(SendPort initialReplyTo){
    final port = new ReceivePort();
    //绑定
    initialReplyTo.send(port.sendPort);
    //监听
    port.listen((message){
        //获取数据并解析
        final data = message[0] as int;
        final send = message[1] as SendPort;
        //返回结果
        send.send(syncFibonacci(data));
    });
}

int syncFibonacci(int n){
    return n < 2 ? n : syncFibonacci(n-2) +
syncFibonacci(n-1);
}
```

## 运行结果

直接运行程序就会在log里看到如下的打印：

```
flutter: 6765
```

## Isolate有什么用？

说了这么久，为什么要创建自己的 Isolate？有什么用？

因为 Root Isolate 会负责渲染，还有 UI 交互，如果我们有一个很耗时的操作呢？前面知道 Isolate 里是一个 Event loop（事件循环），如果一个很耗时的 task 一直在运行，那么后面的UI操作都被阻塞了，所以如果有耗时的操作，就应该放在 Isolate 里！

## 使用 Compute 写 isolate

前面讲了如何创建 Isolate，但那种方式使用起来比较麻烦，属于低级 API，本节讲用高级 API 来创建 Isolate 。

## 使用 Isolates 的方法

使用 Isolates 的方法种：

1. 高级API：Compute 函数 (用起来方便)
2. 低级API：ReceivePort

## Compute 函数

Compute 函数对 Isolate 的创建和底层的消息传递进行了封装，使得我们不必关系底层的实现，只需要关注功能实现。

首先我们需要：

1. 一个函数：必须是顶级函数或静态函数
2. 一个参数：这个参数是上面的函数定义入参（函数没有参数的话就没有）

比如，还是计算斐波那契数列：

```
void main() async{
    //调用compute函数，compute函数的参数就是想要在
    isolate里运行的函数，和这个函数需要的参数
    print( await compute(syncFibonacci, 20));
    runApp(MyApp());
}

int syncFibonacci(int n){
    return n < 2 ? n : syncFibonacci(n-2) +
    syncFibonacci(n-1);
}
```

运行后的结果如下：

```
flutter: 6765
```

是不是很简单，接下来看下 `compute` 函数的源码，这里的代码有点复杂，会把分析的添加到代码的注释里，首先介绍一个 `compute` 函数里用到的函数别名：

`ComputeCallback<Q, R>` 定义如下：

```
// Q R是泛型，ComputeCallback是一个有参数Q，返回值为R的
函数
typedef ComputeCallback<Q, R> = R Function(Q
message);
```

正式看源码：

```
//compute函数 必选参数两个，已经讲过了
Future<R> compute<Q, R>(ComputeCallback<Q, R>
callback, Q message, { String debugLabel }) async
{
    //如果是在profile模式下,debugLabel为空的话，就取
callback.toString()
    profile(() { debugLabel ??=
callback.toString(); });
    final Flow flow = Flow.begin();
    Timeline.startSync('$debugLabel: start', flow:
flow);
    final ReceivePort resultPort = ReceivePort();
    Timeline.finishSync();
    //创建isolate,这个和前面讲的创建isolate的方法一致
    //还有一个，这里传过去的参数是用
_IsolateConfiguration封装的类
    final Isolate isolate = await
Isolate.spawn<_IsolateConfiguration<Q, R>>(
    _spawn,
    _IsolateConfiguration<Q, R>(
        callback,
        message,
        resultPort.sendPort,
        debugLabel,
        flow.id,
    ),
    errorsAreFatal: true,
    onExit: resultPort.sendPort,
);
    final R result = await resultPort.first;
    Timeline.startSync('$debugLabel: end', flow:
```

```

Flow.end(flow.id));
    resultPort.close();
    isolate.kill();
    Timeline.finishSync();
    return result;
}

@immutable
class _IsolateConfiguration<Q, R> {
    const _IsolateConfiguration(
        this.callback,
        this.message,
        this.resultPort,
        this.debugLabel,
        this.flowId,
    );
    final ComputeCallback<Q, R> callback;
    final Q message;
    final SendPort resultPort;
    final String debugLabel;
    final int flowId;

    R apply() => callback(message);
}

void _spawn<Q, R>(_IsolateConfiguration<Q, R>
configuration) {
    R result;
    Timeline.timeSync(
        '${configuration.debugLabel}',
        () {
            result = configuration.apply();
        },

```

```
        flow: Flow.step(configuration.flowId),
    );
    Timeline.timeSync(
        '${configuration.debugLabel}: returning
result',
        () { configuration.resultPort.send(result);
    },
        flow: Flow.step(configuration.flowId),
    );
}
```

## ReceivePort

```
import 'dart:async';
import 'dart:io';
import 'dart:isolate';

import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

//一个普普通通的Flutter应用的入口
//main函数这里有async关键字，是因为创建的isolate是异步的
void main() async{
    runApp(MyApp());

    //asyncFibonacci函数里会创建一个isolate，并返回运行
    结果
    print(await asyncFibonacci(20));
}

//这里以计算斐波那契数列为例，返回的值是Future，因为是异步
```



的

```
Future<dynamic> asyncFibonacci(int n) async{
    //首先创建一个ReceivePort,为什么要创建这个?
    //因为创建isolate所需的参数,必须要有SendPort,
    SendPort需要ReceivePort来创建
    final response = new ReceivePort();
    //开始创建isolate,Isolate.spawn函数是isolate.dart
    里的代码,_isolate是我们自己实现的函数
    //_isolate是创建isolate必须有的参数。
    await
    Isolate.spawn(_isolate,response.sendPort);
    //获取sendPort来发送数据
    final sendPort = await response.first as
    SendPort;
    //接收消息的ReceivePort
    final answer = new ReceivePort();
    //发送数据
    sendPort.send([n,answer.sendPort]);
    //获得数据并返回
    return answer.first;
}

//创建isolate必须有的参数
void _isolate(SendPort initialReplyTo){
    final port = new ReceivePort();
    //绑定
    initialReplyTo.send(port.sendPort);
    //监听
    port.listen((message){
        //获取数据并解析
        final data = message[0] as int;
        final send = message[1] as SendPort;
        //返回结果
```

```
        send.send(syncFibonacci(data));
    });
}

int syncFibonacci(int n){
    return n < 2 ? n : syncFibonacci(n-2) +
syncFibonacci(n-1);
}
```