Flutter await、async、Isolate、compute 异步处理

maskerII

1. 基本概念

1.1 Dart是单线程的

1.2 阻塞式调用和非阻塞式调用

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。

- 阻塞式调用： 调用结果返回之前，当前线程会被挂起，调用线程只有在得到调用结果之后才会继续执行。
- 非阻塞式调用： 调用执行之后，当前线程不会停止执行，只需要过一段时间来检查一下有没有结果返回即可。

1.3 Dart事件循环

事件循环是什么呢?

- 事实上事件循环并不复杂，它就是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。

事件循环伪代码

```
// 这里我使用数组模拟队列，先进先出的原则
List eventQueue = [];
var event;

// 事件循环从启动的一刻，永远在执行
while (true) {
  if (eventQueue.length > 0) {
    // 取出一个事件
    event = eventQueue.removeAt(0);
    // 执行该事件
    event();
  }
}
```

2. Dart的异步操作

Dart中的异步操作主要使用Future以及async、await

2.1 Future

2.1.1 通过Future获取异步处理结果

将需要异步处理的操作放到Future的函数中，再通过Future

的then获取异步处理的结果。

```
void futureTest1() {
  print("main function start");
  // 使用变量接收getNetworkData返回的future
  var future = getNetworkData();
  // 当future实例有返回结果时，会自动回调then中传入的函数
  // 该函数会被放入到事件循环中，被执行
  future.then((value) {
    print(value);
  });
  print(future);
  print("main function end");
}

Future<String> getNetworkData() {
  return Future(() {
    // 休眠2s 模拟请求过程
    sleep(Duration(seconds: 2));
    // 直接返回数据，Dart会使用Future包裹返回数据
    return "请求结果";
  });
}
```

上面代码执行结果

```
main function start
Instance of 'Future<String>'
main function end
请求结果
```

2.1.2 异常处理

使用 catchError 获取异常

```dart
void futureTest1() {
  print("main function start");
  // 使用变量接收getNetworkData返回的future
  var future = getNetworkData();
  // 当future实例有返回结果时，会自动回调then中传入的函数
  // 该函数会被放入到事件循环中，被执行
  future.then((value) {
    print(value);
  }).catchError((error) {
    print(error);
  });
  print(future);
  print("main function end");
}

Future<String> getNetworkData() {
  return Future(() {
    // 休眠2s 模拟请求过程
    sleep(Duration(seconds: 2));
    // 直接返回数据，Dart会使用Future包裹返回数据
    // return "请求结果";
    throw Exception("网络请求出现错误");
  });
}
```

上面代码运行结果

```
main function start
Instance of 'Future<String>'
main function end
Exception: 网络请求出现错误
```

总结：

- 1、创建一个Future（可能是我们创建的，也可能是调

用内部 API 或者第三方 API 获取到的一个 Future，总之你需要获取到一个 Future 实例，Future 通常会对一些异步的操作进行封装）；

- 2、通过.then(成功回调函数) 的方式来监听 Future 内部执行完成时获取到的结果；
- 3、通过.catchError(失败或异常回调函数) 的方式来监听 Future 内部执行失败或者出现异常时的错误信息

2.1.3 Future 链式调用

我们可以在 then 中继续返回值，会在下一个链式的 then 调用回调函数中拿到返回的结果

```
void futureTest2() {
  print("main function start");
  var future = getNetworkData2();
  future.then((value) {
    print(value);
    return "$value - content data2";
  }).then((value) {
    print(value);
    return "$value - content data3";
  }).then((value) {
    print(value);
  });

  print("main function end");
}
```

return后，value的值为"content data1 - content data2"

return后，value的值为"content data1 - content data2 - content data3"

```
Future<String> getNetworkData2() {
  return Future<String>(() {
    sleep(Duration(seconds: 2));
    return "content data1";
  });
}
```

运行结果如下：

```
main function start
main function end
content data1
content data1 - content data2
content data1 - content data2 - content data3
```

模拟三次网络请求，第二次网络请求依赖第一次网络请求结果，第三次网络请求依赖第二次网络请求结果

```
void futureTest3() {
  print("main function start");
  var future = getNetworkData3("content data1");
  future.then((value) {
    print(value);
    return getNetworkData3("$value - content data2");
  }).then((value) {
    print(value);
    return getNetworkData3("$value - content data3");
  }).then((value) {
    print(value);
  });

  print("main function end");
}

Future<String> getNetworkData3(String msg) {
```

```
  return Future<String>(() {
    sleep(Duration(seconds: 2));
    return msg;
  });
}
```

运行结果如下

```
main function start
main function end
content data1
content data1 - content data2
content data1 - content data2 - content data3
```

2.1.4 Future其他API

Future.value(value)

直接获取一个完成的Future，该Future会直接调用then的回调函数

```
void futureTest4() {
  print("main function start");
  Future.value("哈哈哈").then((value) {
    print("$value");
  });
  print("main function end");
}
```

运行结果如下：

```
main function start
main function end
哈哈哈
```

疑惑：为什么立即执行，但是哈哈哈是在最后打印的呢?
这是因为 Future 中的 then 会作为新的任务会加入到事件队列中（Event Queue），加入之后你肯定需要排队执行了

Future.error(object)

直接获取一个完成的 Future，但是是一个发生异常的 Future，该 Future 会直接调用 catchError 的回调函数

```
void futureTest5() {
  print("main function start");
  Future.error(Exception("错误信息")).catchError((error) {
    print("$error");
  });
  print("main function end");
}
```

运行结果如下：

```
main function start
main function end
Exception: 错误信息
```

Future.delayed(时间, 回调函数)

在延迟一定时间时执行回调函数，执行完回调函数后会执行 then 的回调；

```
void futureTest6() {
  print("main function start");
  Future.delayed(Duration(seconds: 3), () {
```

```
    return "3s后的信息";
  }).then((value) {
    print(value);
  });
  print("main function end");
}
```

运行结果如下：

```
main function start
main function end
3s后的信息
```

3. await、async

await、async是什么呢?

- 它们是Dart中的关键字
- 它们可以让我们用同步的代码格式，去实现异步的调用过程。
- 并且，通常一个async的函数会返回一个Future

3.1 案例代码演练

我们知道，如果直接这样写代码，代码是不能正常执行的：
因为Future.delayed返回的是一个Future对象，我们不能把它看成同步的返回数据："network data"去使用
也就是我们不能把这个异步的代码当做同步一样去使用！

```
import "dart:io";
```

```
main(List<String> args) {
  print("main function start");
  print(getNetworkData());
  print("main function end");
}

String getNetworkData() {
  var result = Future.delayed(Duration(seconds: 3), () {
    return "network data";
  });

  return  "请求到的数据: " + result;
}
```

现在我使用 await 修改下面这句代码：

你会发现，我在 Future.delayed 函数前加了一个 await。

一旦有了这个关键字，那么这个操作就会等待 Future.delayed 的执行完毕，并且等待它的结果。

```
String getNetworkData() {
  var result = await Future.delayed(Duration(seconds: 3),
() {
    return "network data";
  });

  return  "请求到的数据: " + result;
}
```

修改后执行代码，会看到如下的错误：

```
The await expression can only be used in an async function.
Try marking the function body with either 'async' or
'async*
```

错误非常明显：await关键字必须存在于async函数中。

所以我们需要将getNetworkData函数定义成async函数。

继续修改代码如下：

也非常简单，只需要在函数的()后面加上一个async关键字就可以了

```
String getNetworkData() async {
  var result = await Future.delayed(Duration(seconds: 3),
() {
    return "network data";
  });

  return  "请求到的数据：" + result;
}
```

运行代码，依然报错：

```
Functions marked 'async' must have a return type assignable
to 'Future'
```

错误非常明显：使用async标记的函数，必须返回一个Future对象。

所以我们需要继续修改代码，将返回值写成一个Future。

继续修改代码如下：

```
Future<String> getNetworkData() async {
  var result = await Future.delayed(Duration(seconds: 3),
() {
    return "network data";
```

```
  });

  return "请求到的数据: " + result;
}
```

这段代码应该是我们理想当中执行的代码了

我们现在可以像同步代码一样去使用Future异步返回的结果;

等待拿到结果之后和其他数据进行拼接，然后一起返回;

返回的时候并不需要包装一个Future，直接返回即可，但是返回值会默认被包装在一个Future中;

3.2 读取json案例

```
import 'package:flutter/services.dart' show rootBundle;
import 'dart:convert';
import 'dart:async';

main(List<String> args) {
  getAnchors().then((anchors) {
    print(anchors);
  });
}

class Anchor {
  String nickname;
  String roomName;
  String imageUrl;

  Anchor({
    this.nickname,
    this.roomName,
```

```
    this.imageUrl
  });

  Anchor.withMap(Map<String, dynamic> parsedMap) {
    this.nickname = parsedMap["nickname"];
    this.roomName = parsedMap["roomName"];
    this.imageUrl = parsedMap["roomSrc"];
  }
}

Future<List<Anchor>> getAnchors() async {
  // 1.读取json文件
  String jsonString = await rootBundle.loadString("assets/
yz.json");

  // 2.转成List或Map类型
  final jsonResult = json.decode(jsonString);

  // 3.遍历List，并且转成Anchor对象放到另一个List中
  List<Anchor> anchors = new List();
  for (Map<String, dynamic> map in jsonResult) {
    anchors.add(Anchor.withMap(map));
  }
  return anchors;
}
```

4. Dart的异步补充

我们知道Dart中有一个事件循环（Event Loop）来执行我们的代码，里面存在一个事件队列（Event Queue），事件循环不断从事件队列中取出事件执行。

但是如果我们严格来划分的话，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；

那么在Flutter开发中，哪些是放在事件队列，哪些是放在微任务队列呢?

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，代码到底是怎样执行的呢?

- **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；

4.1 如何创建微任务

在开发中，我们可以通过dart中async下的scheduleMicrotask来创建一个微任务：

```
import "dart:async";

void main(List<String> args) {
  scheduleMicrotask(() {
    print("Hello Microtask");
  });
}
```

在开发中，如果我们有一个任务不希望它放在Event Queue中依次排队，那么就可以创建一个微任务了。

4.2 Future的代码是加入到事件队列还是微任务队列呢?

Future中通常有两个函数执行体：

- Future构造函数传入的函数体
- then的函数体（catchError等同看待）

那么它们是加入到什么队列中的呢?

Future构造函数传入的函数体放在事件队列中

then的函数体要分成三种情况：

情况一：Future没有执行完成（有任务需要执行），那么then会直接被添加到Future的函数执行体后；

情况二：如果Future执行完后就then，该then的函数体被放到如微任务队列，当前Future执行完后执行微任务队列；

情况三：如果Future是链式调用，意味着then未执行完，下一个then不会执行

```
// future_1加入到eventqueue中，紧随其后then_1被加入到eventqueue
中
Future(() => print("future_1")).then((_) =>
print("then_1"));

// Future没有函数执行体，then_2被加入到microtaskqueue中
Future(() => null).then((_) => print("then_2"));

// future_3、then_3_a、then_3_b依次加入到eventqueue中
Future(() => print("future_3")).then((_) =>
print("then_3_a")).then((_) => print("then_3_b"));
```

4.3 代码执行顺序

```
void asyncTest() {
  print("main start");

  Future(() => print("task 1"));

  final future = Future(() => null);
  Future(() => print("task 2")).then((value) {
    print("task 3");
    scheduleMicrotask(() {
```

```
      print("task 4");
    });
  }).then((value) {
    print("task 5");
  });

  future.then(
    (value) => print("task 6"),
  );

  scheduleMicrotask(() => print('task 7'));

  Future(() => print("task 8")).then((_) {
    Future(() => print("task 9")).then((_) {
      print("task 10");
    });
  });

  print("main end");
}
```

执行顺序

```
main start
main end
task 7
task 1
task 6
task 2
task 3
task 5
task 4
task 8
task 9
task 10
```

代码分析：

1、main 函数先执行，所以 main start 和 main end 先执行，没有任何问题；

2、main 函数执行过程中，会将一些任务分别加入到 EventQueue 和 MicrotaskQueue 中；

3、task7 通过 scheduleMicrotask 函数调用，所以它被最早加入到 MicrotaskQueue，会被先执行；

4、然后开始执行 EventQueue，task1 被添加到 EventQueue 中被执行；

5、通过 final future = Future(() => null);创建的 future 的 then 被添加到微任务中，微任务直接被优先执行，所以会执行 task6；

6、一次在 EventQueue 中添加 task2、task3、task5 被执行；

7、task3 的打印执行完后，调用 scheduleMicrotask，那么在执行完这次的 EventQueue 后会执行，所以在 task5 后执行 task4（注意：scheduleMicrotask 的调用是作为 task3 的一部分代码，所以 task4 是要在 task5 之后执行的）

8、task8、task9、task10 一次添加到 EventQueue 被执行；

5. 多核 CPU 的利用

在Dart中，有一个Isolate的概念，它是什么呢?

我们已经知道Dart是单线程的，这个线程有自己可以访问的内存空间以及需要运行的事件循环；我们可以将这个空间系统称之为是一个Isolate；　isolate 隔离，孤立

比如Flutter中就有一个Root Isolate，负责运行Flutter的代码，比如UI渲染、用户交互等等；在 Isolate 中，资源隔离做得非常好，每个 Isolate 都有自己的 Event Loop 与 Queue，Isolate 之间不共享任何资源，只能依靠消息机制通信，因此也就没有资源抢占问题。

但是，如果只有一个Isolate，那么意味着我们只能永远利用一个线程，这对于多核CPU来说，是一种资源的浪费。

如果在开发中，我们有非常多耗时的计算，完全可以自己创建Isolate，在独立的Isolate中完成想要的计算操作。

5.1 如何创建Isolate呢?

创建Isolate是比较简单的，我们通过Isolate.spawn就可以创建了：　spawn 大量生产

```
import 'dart:isolate';

void main(List<String> args) {
```

```
  Isolate.spawn(foo, "Hello Isolate");
}

void foo(info) {
  print("新的isolate: $info");
}
```

5.2 Isolate通信机制

在真实开发中，我们不会只是简单的开启一个新的Isolate，而不关心它的运行结果： isolate 隔离

我们需要新的Isolate进行计算，并且将计算结果告知Main Isolate（也就是默认开启的Isolate）；

Isolate 通过发送管道（SendPort）实现消息通信机制；

我们可以在启动并发 Isolate 时将 Main Isolate 的发送管道作为参数传递给它；

并发在执行完毕时，可以利用这个管道给 Main Isolate 发送消息；

```
void IsolateTest2() async {
  // 1.创建管道
  ReceivePort receivePort = ReceivePort();

  // 2.创建新的Isolate
  Isolate isolate = await
Isolate.spawn<SendPort>((sendport) {
    // 发送消息
    sendport.send("Hello World");
  }, receivePort.sendPort);
```

```
  // 3. 监听管道信息
  receivePort.listen((message) {
    print("Data $message");
    // 不再使用时，我们会关闭管道
    receivePort.close();
    // 需要杀死isolate
    isolate.kill(priority: Isolate.immediate);
  });
}
```

immediate 即刻的，紧迫的

注意：上面的代码只能单向通信

双向通信

Flutter提供了支持并发计算的compute函数，它内部封装了Isolate的创建和双向通信；利用它我们可以充分利用多核心CPU，并且使用起来也非常简单；

```
main(List<String> args) async {
  int result = await compute(powerNum, 5);
  print(result);
}

int powerNum(int num) {
  return num * num;
}
void computeTest() async {
  String res = await compute<int, String>(powerNum, 5);
  print("TTTTTT $res");
}

FutureOr<String> powerNum(int num) {
  return (num * num).toString();
}
```