

Flutter 的 Event Loop(事件循环)及代码运行顺序

接触过 Flutter 的人都知道，Flutter 是用 Dart 来写的，Dart 没有进程和线程的概念，所有的 Dart 代码 都是在 Isolate 上 运行的，那么 Isolate 到底是什么？本节将详细讨论。这篇文章讨论事件队列(event loop)及Dart代码运行顺序。

同步代码和异步代码

我们在写 Dart 代码 的时候，对 Dart 代码 进行分类,就只有两类，同步代码和异步代码；

- 异步代码：就是以 Future 等修饰的代码
- 同步代码：除了异步代码，平常写的代码就是同步代码

在 Dart 中这两类代码是不同的：

1.运行顺序不同

同步代码和异步代码运行的顺序是不同的：

先运行同步代码，在运行异步代码

就是，即使我异步代码写在最前面，同步代码写在最后面，不好意思，我也是先运行后面的同步代码，同步代码都运行完后，在运行前面的异步代码。

2.运行的机制不同

异步代码是运行在 event loop 里的，这是一个很重要的概念，这里可以理解成 Android 里的 Looper 机制，是一个死循环，event loop 不断的从事件队列里取事件然后运行。

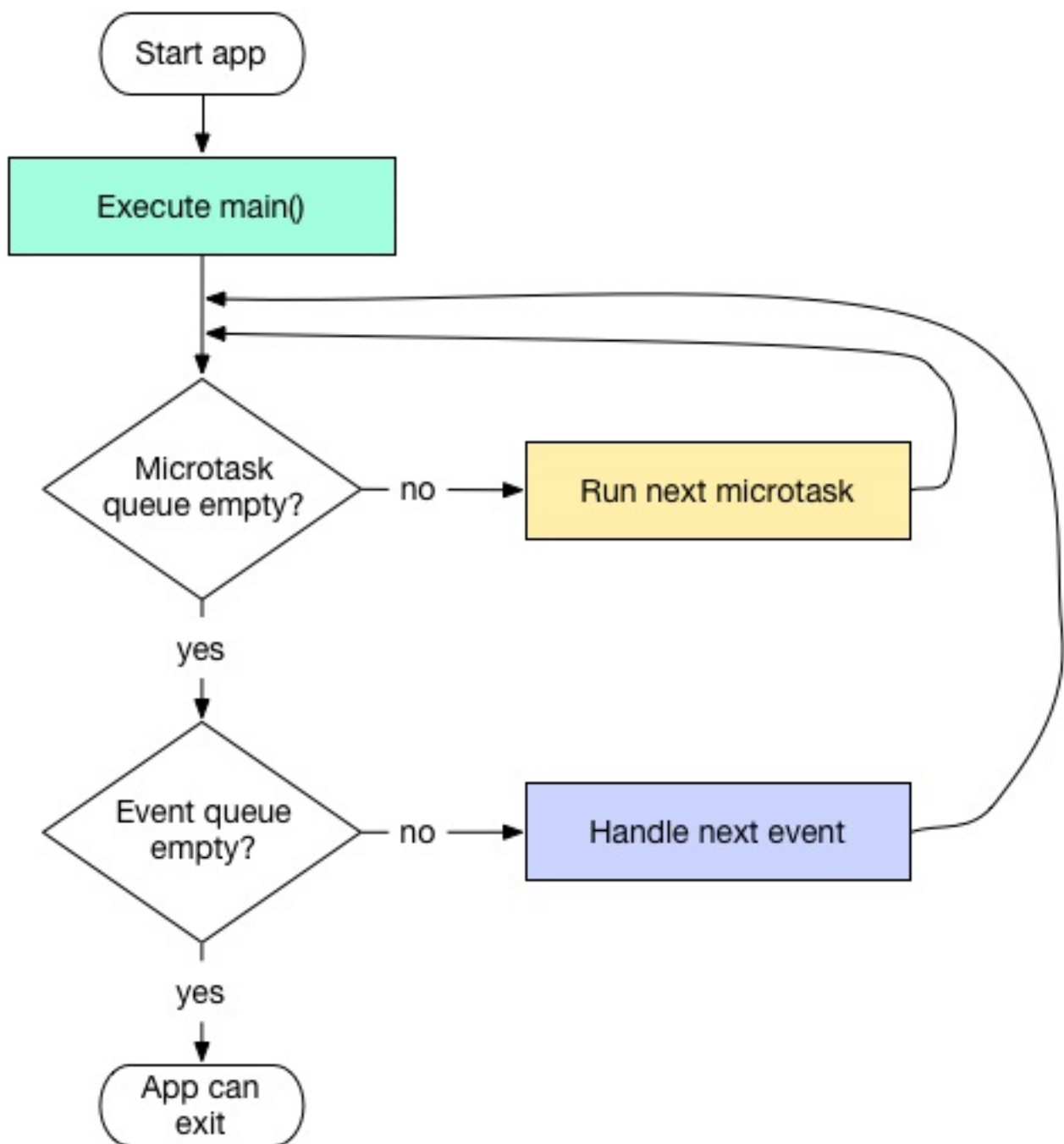
event loop 架构

下面是 event loop 大致的运行图：



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。

这个理解后，在看 event loop 详细的运行图：



从这里看到，启动 app（start app）后：

1. 先查看 MicroTask queue 是不是空的，不是的话，先运行 microtask
2. 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
3. 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

1. MicroTask

这个大家应该不太清楚，但是这个也是 dart:async 提供的异步方法，使用方式：

```
// Adds a task to the 先查看MicroTask queue.  
scheduleMicrotask((){  
    // ...code goes here...  
});
```

或者：

```
new Future.microtask((){  
    // ...code goes here...  
});
```

如果能让任务能够尽快执行，就用 MicroTask

2.Event

Event 我们就很清楚了，就是 Future 修饰的异步方法，使用方式：

```
// Adds a task to the Event queue.  
new Future(() {  
    // ...code goes here...  
});
```

代码运行顺序

纯粹讲理论知识不太好理解，我们直接上代码，讲一个例子，看如下的代码，请问打印顺序是什么样的？

```
import 'dart:async';
void main() {
  print('main #1 of 2');
  scheduleMicrotask(() => print('microtask #1 of 3'));

  new Future.delayed(new Duration(seconds:1),
    () => print('future #1 (delayed)'));

  new Future(() => print('future #2 of 4'))
    .then((_) => print('future #2a'))
    .then((_) {
      print('future #2b');
      scheduleMicrotask(() => print('microtask #0 (from future #2b)'));
    })
    .then((_) => print('future #2c'));

  scheduleMicrotask(() => print('microtask #2 of 3'));

  new Future(() => print('future #3 of 4'))
    .then((_) => new Future(
      () => print('future #3a (a new future)'))))
    .then((_) => print('future #3b'));

  new Future(() => print('future #4 of 4'))
    .then((_) {
      new Future(() => print('future #4a'));
    })
    .then((_) => print('future #4b'));
  scheduleMicrotask(() => print('microtask #3 of 4'));
```

```
3'));  
  print('main #2 of 2');  
}
```

1. 首先运行同步代码

所以是:

```
main #1 of 2  
main #2 of 2
```

2. 接下来是异步代码

Dart 的 Event Loop 是先判断 microtask queue 里有没有 task, 有的话运行 microtask, microtask 行完后, 在运行 event queue 里的 event task, 一个 event task 运行完后, 再去运行 microtask queue, 然后在运行 event queue。

3. microtask queue

这里就是:

```
microtask #1 of 3  
microtask #2 of 3
```

4. event queue

event queue 还有有特殊的情况需要考虑:

- Future.delayed

需要延迟执行的, Dart 是怎么执行的呢, 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

- Future.then

Future.then 里的 task 是不会加入到 event queue 里的，而是当前面的 Future 执行完后立即掉起，所以你如果想保证异步 task 的执行顺序一定要用 then，否则 Dart 不保证 task 的执行顺序

- scheduleMicrotask

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

这里就是：

```
future #2 of 4
future #2a
future #2b
future #2c
microtask #0 (from future #2b)
future #3 of 4
future #4 of 4
future #4b
future #3a (a new future)
future #3b
future #4a
future #1 (delayed)
```

这里你肯定好奇为啥 future #3 of 4 后面是 future #4 of 4，而不是 future #3a (a new future)，因为 future #3 of 4 的 then 里又新建了一个 Future: future #3a (a new future)，所以 future #3a (a new future) 这个 task 会加到 event queue 的最后面。

最后的结果就是：

```
main #1 of 2
main #2 of 2
microtask #1 of 3
microtask #2 of 3
microtask #3 of 3
future #2 of 4
future #2a
future #2b
future #2c
microtask #0 (from future #2b)
future #3 of 4
future #4 of 4
future #4b
future #3a (a new future)
future #3b
future #4a
future #1 (delayed)
```