Flutter篇之你真的会使用Future吗?

入魔的冬瓜 LV.4
2019年07月19日 19:28 · 阅读 22284

关注

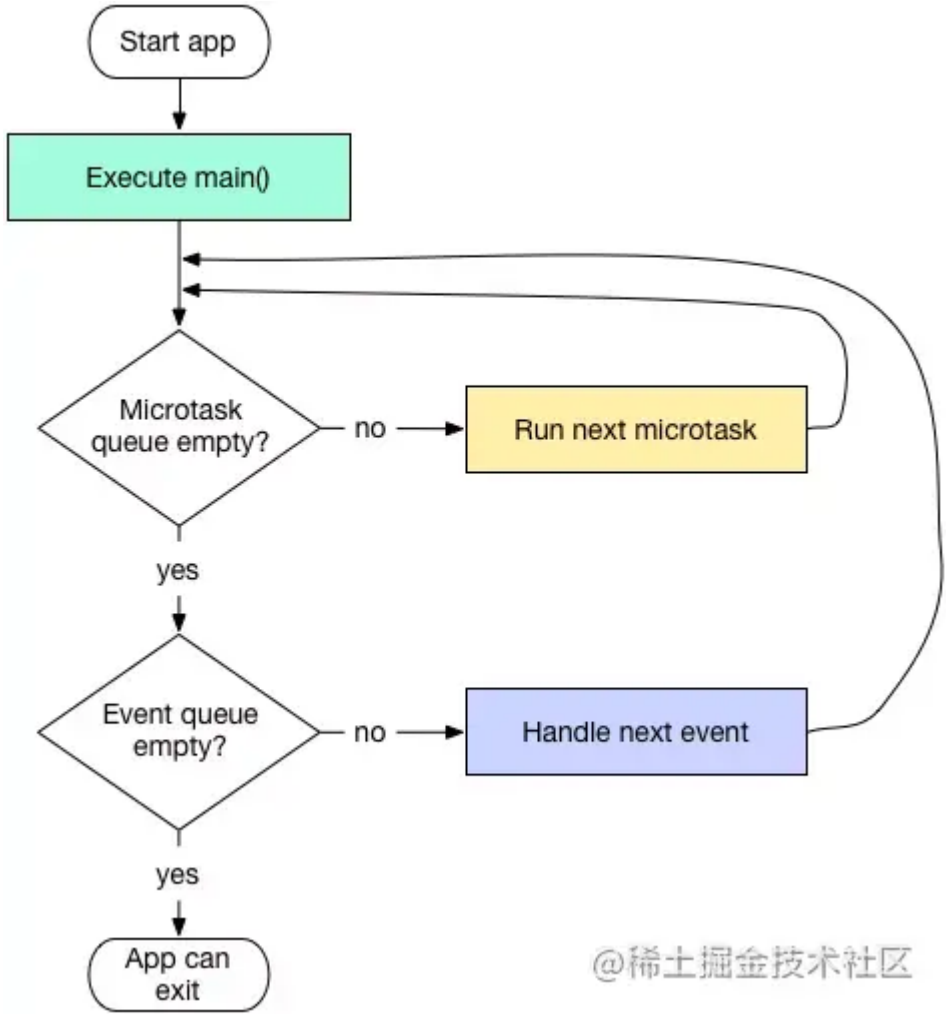
最近用Flutter写了个简单的“爬虫”的东西，就是要不断地请求某些接口，然后读取数据就可以了。之前只是会简单地await和async来试用Future，后面发现只会简单的这种方式不行，于是去学习了Future的其他用法，还是有点收获的，把Future的用法都学了一下，这里做个笔记。

哈哈，别问我为什么没用python去搞。。。刚好用的电脑没装环境并且python的API没那么熟，就有了个想法，反正都能实现，就用Flutter玩一下吧，可能后面还可以搞个可视化界面玩玩。

Dart的消息循环机制

推荐文章：Dart与消息循环机制

这里简单说下Dart的事件循环，更加具体的话，可以看推荐的这篇文章。



类似Android的Handler/Looper机制，Dart也有相应的消息循环机制。

- 一个Dart应用中有一个消息循环和两个队列，一个是event队列，一个是microtask队列。
- 优先级问题：microtask队列的优先级是最高的，当所有microtask队列执行完之后才会从event队列中读取事件进行执行。
- microtask队列中的event，
- event队列包含所有的外来事件：I/O，mouse events，drawing events，timers，isolate之间的message等。
- 一般Future创建的事件是属于event队列的（利用Future.microtask方法例外），所以创建一个Future后，会插入到event队列中，顺序执行。

Future的介绍

在写程序的过程中，肯定会有一部分比较耗时代码是需要异步执行的。比如网络操作，我们需要异步去请求数据，并且还需要处理请求成功和请求失败的两种情况。

在Flutter中，使用Future来执行耗时操作，表示在未来会返回某个值，并可以使用then（）方法和catchError（）来注册callback来监听Future的处理结果。

如下面所示，源码中对Future的解释，以及简单的一个例子。then（）方法和catchError（）返回的其实也是一个Future类型对象，所以可以写成链式调用的形式。

scss 复制代码

```
An object representing a delayed computation.
Future<int> future = getFuture();
future.then((value) => handleValue(value))
      .catchError((error) => handleError(error));
```

Flutter的状态

一个Future对象会有以下两种状态

- pending：表示Future对象的计算过程仍在执行中，这个时候还没有可以用的result。
- completed：表示Future对象已经计算结束了，可能会有两种情况，一种是正确的结果，一种是失败的结果。

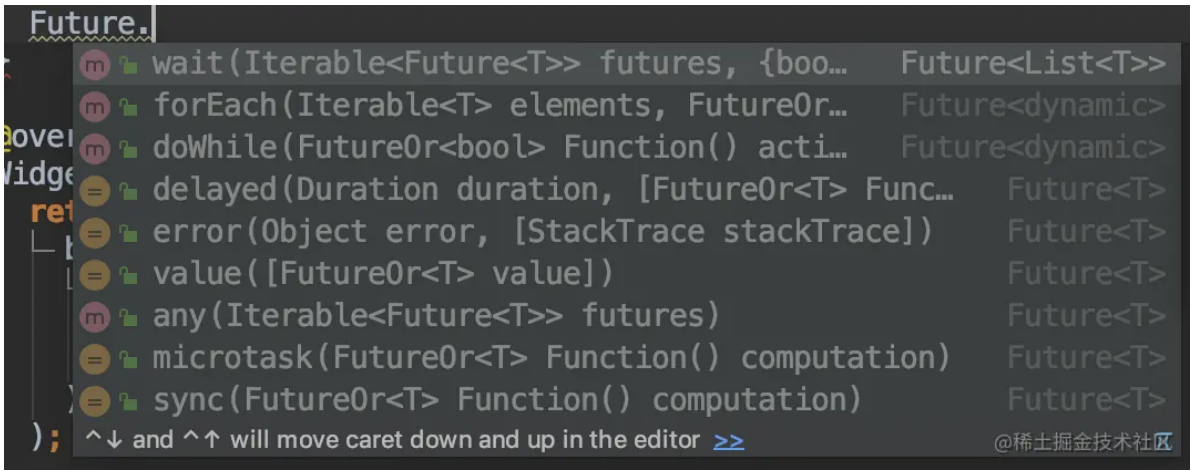
创建Future

注意：首先Future是个泛型类，可以指定类型。如果没有指定相应类型的话，默认返回是Future类型的。

scss 复制代码

```
Future createFuture()async{
  return 1;
}
var future=createFuture();
print(future.runtimeType);。//输出结果为Future<dynamic>
```

Future常用的创建有以下几种方式。



基本的用法

factory Future(FutureOr computation())

Future的构造方法，创建一个基本的Future

scss 复制代码

```
var future = Future(() {
  print("哈哈哈");
});
print("111");
//111
//哈哈哈
```

你会发现，结果是打印“哈哈哈”字符串是在后面才输出的，原因是默认future是异步执行的。如果改成下面的代码，在future前面加上await关键字，则会先打印“哈哈哈”字符串。await会等待直到future执行结束后，才继续执行后面的代码。

这个跟上面的事件循环是有关系的，转发大佬的解释：main方法中的普通代码都是同步执行的，所以肯定是main打印先全部打印出来，等main方法结束后会开始检查microtask中是否有任务，若有则执行，执行完继续检查microtask，直到microtask列队为空。所以接着打印的应该是microtask的打印。最后会去执行event队列。

scss 复制代码

```
var future = await Future(() {
  print("哈哈哈");
});
print("111");
//哈哈哈
//111
```

Future.value()

创建一个返回指定value值的Future

ini 复制代码

```ini
var future=Future.value(1);
print(future);
//Instance of 'Future<int>'
```

Future.delayed（）

创建一个延迟执行的future。 例如下面的例子，利用Future延迟两秒后可以打印出字符串。

ini 复制代码

```ini
var futureDelayed = Future.delayed(Duration(seconds: 2), () {
  print("Future.delayed");
  return 2;
});
```

高级的用法

Future.foreach(Iterable elements, FutureOr action(T element))

根据某个集合，创建一系列的Future，并且会按顺序执行这些Future。 比如下面的例子，根据{1,2,3}创建3个延迟对应秒数的Future。执行结果为1秒后打印1，再过2秒打印2，再过3秒打印3，总时间为6秒。

dart 复制代码

```dart
Future.forEach({1,2,3}, (num){
  return Future.delayed(Duration(seconds: num),(){print(num);});
});
```

Future.wait (Iterable<Future> futures,{bool eagerError: false, void cleanUp(T successValue)})

用来等待多个future完成，并收集它们的结果。那这样的结果就有两种情况了：

- 如果所有future都有正常结果返回：则future的返回结果是所有指定future的结果的集合
- 如果其中一个future有error返回：则future的返回结果是第一个error的值。

比如下面的例子，也是创建3个延迟对应秒数的Future。结果是总时间过了3秒后，才输出[1, 2, 3]的结果。可以与上面的例子对比一下，一个是顺序执行多个Future，一个是异步执行多个Future。

dart 复制代码

```dart
var future1 = new Future.delayed(new Duration(seconds: 1), () => 1);
var future2 =
    new Future.delayed(new Duration(seconds: 2), () => 2);
var future3 = new Future.delayed(new Duration(seconds: 3), () => 3);
Future.wait({future1,future2,future3}).then(print).catchError(print);
//运行结果： [1, 2, 3]
```

将future2和future3改为有error抛出，则future.wait的结果是这个future2的error值。

dart 复制代码

```dart
var future1 = new Future.delayed(new Duration(seconds: 1), () => 1);
var future2 =new Future.delayed(new Duration(seconds: 2), () => throw “throw error2");
var future3 = new Future.delayed(new Duration(seconds: 3), () => throw “throw error3");
Future.wait({future1,future2,future3}).then(print).catchError(print);
//运行结果： throw error2
```

Future.any（futures）

返回的是第一个执行完成的future的结果，不会管这个结果是正确的还是error的。（感觉这个的使用场景没几个的样子，想不到有什么场景要用到这个。）

例如下面的例子，使用Future.delayed（）延迟创建了三个不同的Future，第一个完成返回的是延迟1秒的那个future的结果。

arduino 复制代码

```arduino
Future
     .any([1, 2, 5].map(
         (delay) => new Future.delayed(new Duration(seconds: delay), () => delay)))
     .then(print)
     .catchError(print);
```

Future.doWhile()

类似java中doWhile的用法，重复性地执行某一个动作，直到返回false或者Future，退出循环。

使用场景:适用于一些需要递归操作的场景。

比如我就是要爬某个平台的商品分类的数据，那每个分类下面又有相应的子分类。那我就得拿对应父级分类的catId去请求接口，拿到这个分类下面的所有子分类。同样的，对所有的子分类，也要进行一样的操作，递归直到这个分类是一个叶子节点，也就是该节点下面没有子分类（有个leaf为true的字段）。哈哈，代码我就没贴了，其实也就是一个递归操作，直到满足条件退出future。

例如下面的例子，生成一个随机数进行等待，直到十秒之后，操作结束。

dart 复制代码

```dart
void futureDoWhile(){
  var random = new Random();
  var totalDelay = 0;
  Future
      .doWhile(() {
    if (totalDelay > 10) {
      print('total delay: $totalDelay seconds');
      return false;
    }
    var delay = random.nextInt(5) + 1;
    totalDelay += delay;
    return new Future.delayed(new Duration(seconds: delay), () {
      print('waited $delay seconds');
      return true;
    });
  })
      .then(print)
      .catchError(print);
}
//输出结果：
I/flutter (11113): waited 5 seconds
I/flutter (11113): waited 1 seconds
I/flutter (11113): waited 3 seconds
I/flutter (11113): waited 2 seconds
I/flutter (11113): total delay: 12 seconds
I/flutter (11113): null
```

Future.microtask(FutureOr computation())

创建一个在microtask队列运行的future。

在上面讲过，microtask队列的优先级是比event队列高的，而一般future是在event队列执行的，所以Future.microtask创建的future会优先于其他future进行执行。

例如下面的代码，

```scss
Future((){
  print("Future event 1");
});
Future((){
  print("Future event 2");
});
Future.microtask((){
 print("microtask event");
});
//输出结果
//microtask event
//Future event 1
//Future event 2
```

处理Future的结果

Flutter提供了下面三个方法，让我们来注册回调，来监听处理Future的结果。

```csharp
Future<R> then<R>(FutureOr<R> onValue(T value), {Function onError});
Future<T> catchError(Function onError, {bool test(Object error)});
Future<T> whenComplete(FutureOr action());
```

Future.then（）

用于在Future完成的时候添加回调。

注意：then（）的返回值也是一个Future对象。所以我们可以使用链式的方法去使用Future，将前一个Future的输出结果作为后一个Future的输入，可以写成链式调用。

例如下面的代码，将前面的future结果作为后面Future的输入。

```scss
Future.value(1).then((value) {
  return Future.value(value + 2);
}).then((value) {
  return Future.value(value + 3);
}).then(print);
//打印结果为6
```

Future.cathcError（）

注册一个回调，来处理有error的Future

```go
new Future.error('boom!').catchError(print);
```

与then方法中的onError的区别：then方法里面也有个onError的参数，也可以用来处理错误的Future。

两者的区别，onError只能处理当前的错误Future，而不能处理其他有错误的Future。catchError可以捕获到Future链中抛出的所有错误。

所以通常的做法是使用catchError来捕捉Future中的所有错误，不建议使用then方法中的onError方法。不然每个future的then方法都要加上onError回调的话，就比较麻烦了，而且代码看起来也是有点乱。

下面是两个捕捉错误的例子。

在那个抛出错误的future的then方法里添加onError回调的话，onError会优先被调用

```scss
Future.value(1).then((value) {
  return Future.value(value + 2);
}).then((value) {
  throw "error";
}).then(print, onError: (e) {
  print("onError find a error");
}).catchError((e) {
  print("catchError find a error");
});
//输出结果为onError find a error
```

使用catchError来监听链式调用Future里面抛出来的错误。

```scss
Future.value(1).then((value) {
  throw "error";
}).then((value) {
  return Future.value(3);
}).then(print).then((value) {
}).catchError((e) {
  print("catchError find a error");
});
//输出结果为catchError find a error"
```

Future.whenComplete

类似Java中的finally，Future.whenComplete总是在Future完成后调用，不管Future的结果是正确的还是错误的。

注意：Future.whenComplete的返回值也是一个Future对象。

scss 复制代码

```
Future.delayed(Duration(seconds: 3),(){
  print("哈哈");
}).whenComplete((){
  print("complete");
});
//哈哈
//complete
```

最常用的async和await

哈哈，上面说了一些Future的用法，其实有些可能并不会很常用，知道有这个用法就行了。

有个问题，如果有多个Future链接在一起的话，靠，代码可能会变得难以阅读。

所以，可以使用async和await来使用Future，使代码看起来像是同步的代码，但实际上它们还是异步执行的。

最常用的还是async和await来使用Future。

注意：await只能在async函数出现。 async函数，返回值是一个Future对象，如果没有返回Future对象，会自动将返回值包装成Future对象。 捕捉错误，一般是使用try/catch机制来做异常处理。 await 一个future，可以拿到Future的结果，直到拿到结果，才执行下一步的代码。

比如下面的例子，创建一个延迟3秒并返回结果为1的Future，使用await来获取到future的值。

csharp 复制代码

```
void main() async {
  var future = new Future.delayed(new Duration(seconds: 3), () {
    return 1;
  });
  var result = await future;
  print(result + 1);
}
 //输出为2
```

下面的代码结构，跟java一样的写法，使用try-catch-finally来捕捉错误。

dart 复制代码

```dart
try {
  var result1 = await Future.value(1);
  print(result1);//输出1
  var result2 = await Future.value(2);
  print(result2);//输出2
} catch (e) {
    //捕捉错误
} finally {

}
```

总结

- 创建Future的几种方法
- 用then、catchError、whenComplete来处理Future的结果
- async和await的使用
-

下一步

多学习多整理。靠，又懒了好长一段时间了。。。

分类： Android 标签： Flutter