循环中正确使用 async 与 await

async 与 await 的使用方式相对简单。 蛤当你尝试在循环中使用 await 时，事情就会变得复杂一些。
在本文中，分享一些在如果循环中使用 await 值得注意的问题。

准备一个例子

假设你想从水果篮中获取水果的数量。

```
const fruitBasket = {
 apple: 27,
 grape: 0,
 pear: 14
};
```

你想从 fruitBasket 获得每个水果的数量。 要获取水果的数量，可以使用 getNumFruit 函数。

```
const getNumFruit = fruit => {
  return fruitBasket[fruit];
};

const numApples = getNumFruit('apple');
console.log(numApples); //27
```

现在，假设 fruitBasket 是从服务器上获取，这里我们使用 setTimeout 来模拟。

```
const sleep = ms => {
  return new Promise(resolve => setTimeout(resolve, ms))
};

const getNumFruie = fruit => {
  return sleep(1000).then(v => fruitBasket[fruit]);
};

getNumFruit("apple").then(num => console.log(num)); // 27
```

最后，假设你想使用 await 和 getNumFruit 来获取异步函数中每个水果的数量。

```
const control = async _ => {
  console.log('Start')

  const numApples = await getNumFruit('apple');
  console.log(numApples);

  const numGrapes = await getNumFruit('grape');
  console.log(numGrapes);
```

```
  const numPears = await getNumFruit('pear');
  console.log(numPears);

  console.log('End')
}
```

在 for 循环中使用 await

首先定义一个存放水果的数组：

```
const fruitsToGet = [“apple”, “grape”, “pear”];
```

循环遍历这个数组：

```
const forLoop = async _ => {
  console.log('Start');

  for (let index = 0; index < fruitsToGet.length; index++) {
    // 得到每个水果的数量
  }

  console.log('End')
}
```

在 for 循环中，过上使用 getNumFruit 来获取每个水果的数量，并将数量打印到控制台。
由于 getNumFruit 返回一个 promise，我们使用 await 来等待结果的返回并打印它。

```
const forLoop = async _ => {
  console.log('start');

  for (let index = 0; index < fruitsToGet.length; index ++) {
    const fruit = fruitsToGet[index];
    const numFruit = await getNumFruit(fruit);
    console.log(numFruit);
  }
  console.log('End')
}
```

当使用 await 时，希望 JavaScript 暂停执行，直到等待 promise 返回处理结果。这意味着 for 循环中的 await 应该按顺序执行。
结果正如你所预料的那样。

```
“Start”;
“Apple: 27”;
“Grape: 0”;
“Pear: 14”;
```

```
“End”;
```

这种行为适用于大多数循环(比如 while 和 for-of 循环)...
但是它不能处理需要回调的循环，如 forEach、map、filter 和 reduce。在接下来的几节中，我们将研究 await 如何影响 forEach、map 和 filter。

在 forEach 循环中使用 await

首先，使用 forEach 对数组进行遍历。

```
const forEach = _ => {
  console.log('start');

  fruitsToGet.forEach(fruit => {
    //...
  })

  console.log('End')
}
```

接下来，我们将尝试使用 getNumFruit 获取水果数量。 （注意回调函数中的 async 关键字。我们需要这个 async 关键字，因为 await 在回调函数中）。

```
const forEachLoop = _ => {
  console.log('Start');

  fruitsToGet.forEach(async fruit => {
    const numFruit = await getNumFruit(fruit);
    console.log(numFruit)
  });

  console.log('End')
}
```

我期望控制台打印以下内容：

```
“Start”;
“27”;
“0”;
“14”;
“End”;
```

但实际结果是不同的。在 forEach 循环中等待返回结果之前，JavaScrip 先执行了 console.log('End')。
实际控制台打印如下：

```
‘Start’
‘End’
```

```
‘27’
‘0’
‘14’
```

JavaScript 中的 forEach 不支持 promise 感知，也支持 async 和 await，所以不能在 forEach 使用 await 。

在 map 中使用 await

如果在 map 中使用 await, map 始终返回 promise 数组，这是因为异步函数总是返回 promise。

```
const mapLoop = async _ => {
  console.log('Start')
  const numFruits = await fruitsToGet.map(async fruit => {
    const numFruit = await getNumFruit(fruit);
    return numFruit;
  })

  console.log(numFruits);

  console.log('End')
}


“Start”;
“[Promise, Promise, Promise]”;
“End”;
```

如果你在 map 中使用 await，map 总是返回 promises，你必须等待 promises 数组得到处理。 或者通过 await Promise.all(arrayOfPromises)来完成此操作。

```
const mapLoop = async _ => {
  console.log('Start');

  const promises = fruitsToGet.map(async fruit => {
    const numFruit = await getNumFruit(fruit);
    return numFruit;
  });

  const numFruits = await Promise.all(promises);
  console.log(numFruits);

  console.log('End')
```

```
}
```

运行结果如下：

如果你愿意，可以在 promise 中处理返回值，解析后的将是返回的值。

```
const mapLoop = _ => {
  // ...
  const promises = fruitsToGet.map(async fruit => {
    const numFruit = await getNumFruit(fruit);
    return numFruit + 100
  })
  // ...
}

“Start”;
“[127, 100, 114]”;
“End”;
```

在 filter 循环中使用 await

当你使用 filter 时，希望筛选具有特定结果的数组。假设过滤数量大于 20 的数组。
如果你正常使用 filter （没有 await），如下：

```
const filterLoop =  _ => {
  console.log('Start')

  const moreThan20 =  fruitsToGet.filter(async fruit => {
    const numFruit = await fruitBasket[fruit]
    return numFruit > 20
  })

  console.log(moreThan20)
  console.log('END')
}
```

运行结果

```
Start
["apple"]
END
```

filter 中的 await 不会以相同的方式工作。 事实上，它根本不起作用。

```
const filterLoop = async _ => {
  console.log('Start')

  const moreThan20 =  await fruitsToGet.filter(async fruit => {
```

```
    const numFruit = fruitBasket[fruit]
    return numFruit > 20
  })

  console.log(moreThan20)
  console.log('END')
}


// 打印结果
Start
["apple", "grape", "pear"]
END
```

为什么会发生这种情况?

当在 `filter` 回调中使用 `await` 时，回调总是一个 `promise`。由于 `promise` 总是真的，数组中的所有项都通过 `filter` 。在 `filter` 使用 `await` 类以下这段代码

```
const filtered = array.filter(true);
```

在 `filter` 使用 `await` 正确的三个步骤

1. 使用 `map` 返回一个 promise 数组
2. 使用 `await` 等待处理结果
3. 使用 `filter` 对返回的结果进行处理

```
const filterLoop = async _ => {
console.log('Start');

const promises = await fruitsToGet.map(fruit => getNumFruit(fruit));

const numFruits = await Promise.all(promises);

const moreThan20 = fruitsToGet.filter((fruit, index) => {
const numFruit = numFruits[index];
return numFruit > 20;
})

console.log(moreThan20);
console.log('End')
}
```

在 reduce 循环中使用 await

如果想要计算 fruitBastet 中的水果总数。 通常，你可以使用 reduce 循环遍历数组并将数字相加。

```
const reduceLoop = _ => {
  console.log('Start');

  const sum = fruitsToGet.reduce((sum, fruit) => {
    const numFruit = fruitBasket[fruit];
    return sum + numFruit;
  }, 0)

  console.log(sum)
  console.log('End')
}
```

运行结果：

当你在 reduce 中使用 await 时，结果会变得非常混乱。

```
 const reduceLoop = async _ => {
  console.log('Start');

  const sum = await fruitsToGet.reduce(async (sum, fruit) => {
    const numFruit = await fruitBasket[fruit];
    return sum + numFruit;
  }, 0)

  console.log(sum)
  console.log('End')
}
```

[object Promise]14 是什么 鬼？？
剖析这一点很有趣。

1. 在第一次遍历中，sum 为 0。numFruit 是 27(通过 getNumFruit(apple)的得到的值)，0 + 27 = 27。
2. 在第二次遍历中，sum 是一个 promise。 （为什么？因为异步函数总是返回 promises！）numFruit 是 0.promise 无法正常添加到对象，因此 JavaScript 将其转换为[object Promise]字符串。 [object Promise] + 0 是 object Promise] 0。
3. 在第三次遍历中，sum 也是一个 promise。 numFruit 是 14. [object Promise] + 14 是[object Promise] 14。

解开谜团！

这意味着，你可以在 reduce 回调中使用 await，但是你必须记住先等待累加器！

```
const reduceLoop = async _ => {
  console.log('Start');
```

```
  const sum = await fruitsToGet.reduce(async (promisedSum, fruit) => {
    const sum = await promisedSum;
    const numFruit = await fruitBasket[fruit];
    return sum + numFruit;
  }, 0)

  console.log(sum)
  console.log('End')
}
```

但是从上图中看到的那样，`await` 操作都需要很长时间。 发生这种情况是因为 `reduceLoop` 需要等待每次遍历完成 `promisedSum`。
有一种方法可以加速 `reduce` 循环，如果你在等待 promisedSum 之前先等待 `getNumFruits()`，那么 `reduceLoop` 只需要一秒钟即可完成：

```
const reduceLoop = async _ => {
  console.log('Start');

  const sum = await fruitsToGet.reduce(async (promisedSum, fruit) => {
    const numFruit = await fruitBasket[fruit];
    const sum = await promisedSum;
    return sum + numFruit;
  }, 0)

  console.log(sum)
  console.log('End')
}
```

这是因为 `reduce` 可以在等待循环的下一个迭代之前触发所有三个 `getNumFruit` promise。然而，这个方法有点令人困惑，因为你必须注意等待的顺序。
在 reduce 中使用 wait 最简单(也是最有效)的方法是

1. 使用 `map` 返回一个 promise 数组
2. 使用 `await` 等待处理结果
3. 使用 `reduce` 对返回的结果进行处理
   const reduceLoop = async _ => {
   console.log('Start');

   const promises = fruitsToGet.map(getNumFruit);
   const numFruits = await Promise.all(promises);
   const sum = numFruits.reduce((sum, fruit) => sum + fruit);

```
console.log(sum)
console.log('End')
}
```

这个版本易于阅读和理解，需要一秒钟来计算水果总数。

从上面看出来什么

1. 如果你想连续执行 `await` 调用，请使用 `for` 循环(或任何没有回调的循环)。
2. 永远不要和 `forEach` 一起使用 `await`，而是使用 `for` 循环(或任何没有回调的循环)。
3. 不要在 `filter` 和 `reduce` 中使用 `await`，如果需要，先用 `map` 进一步骤处理，然后在使用 `filter` 和 `reduce` 进行处理。