在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
       console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

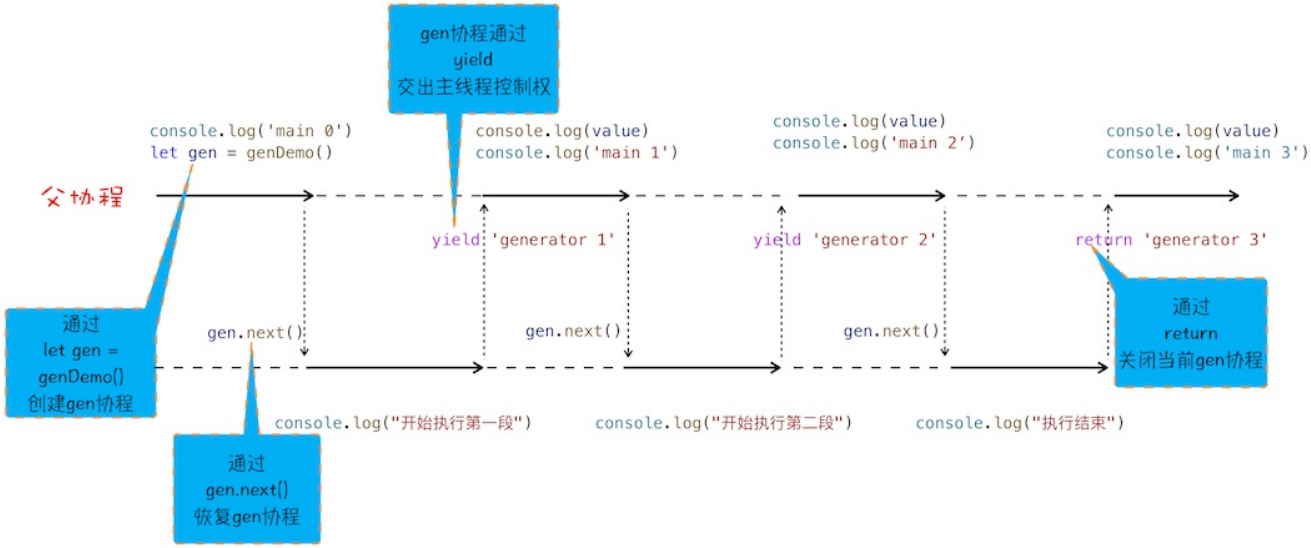
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

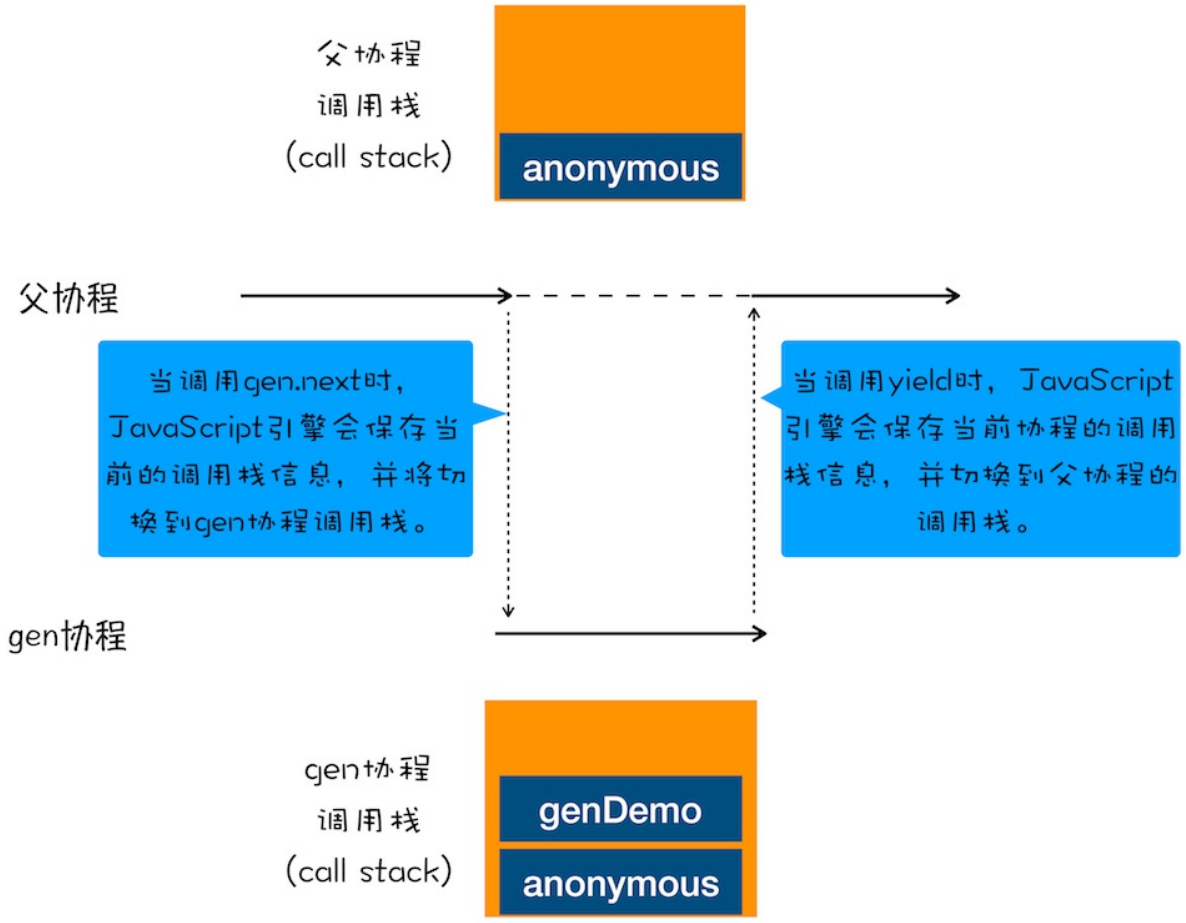
不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

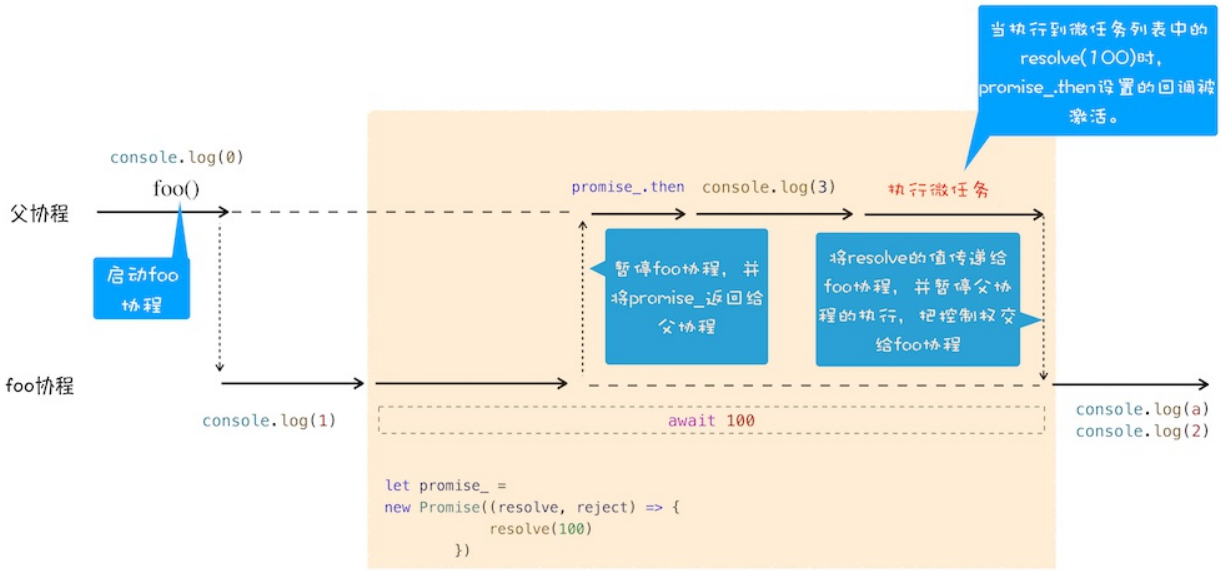
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
       console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

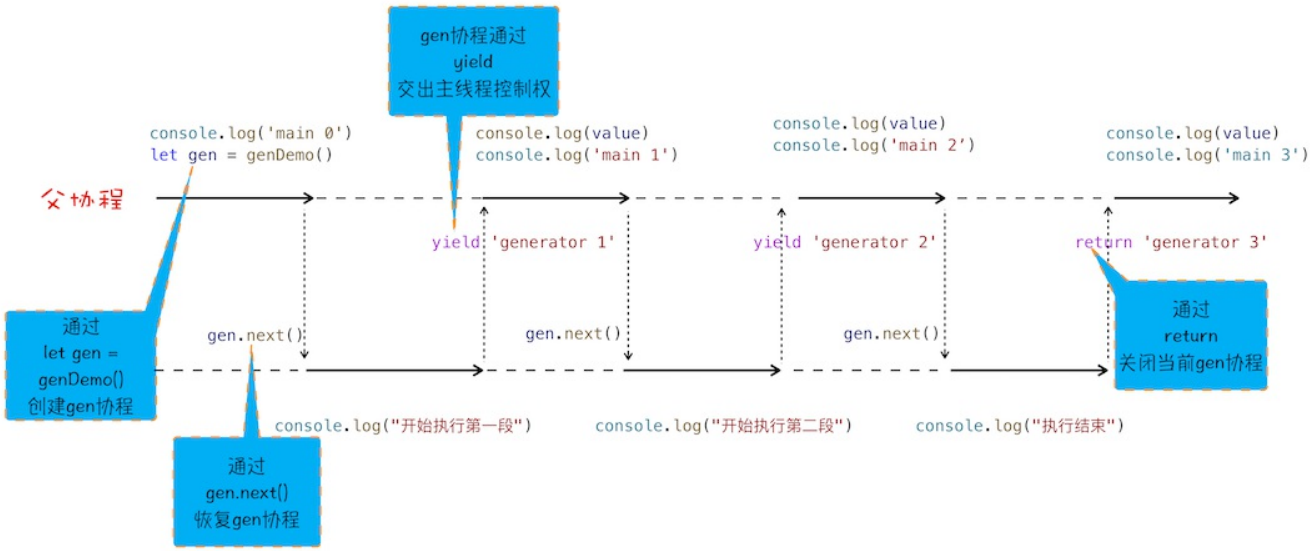
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切

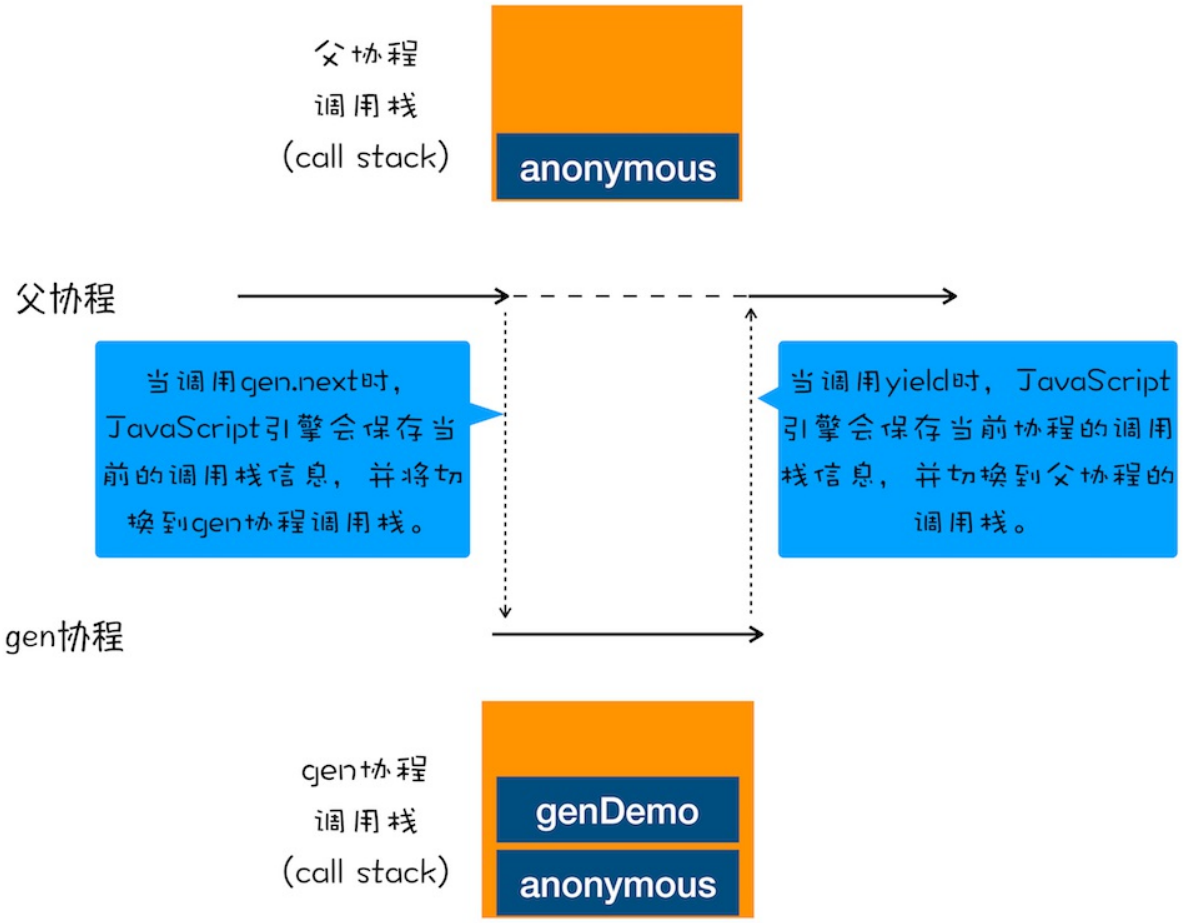
换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

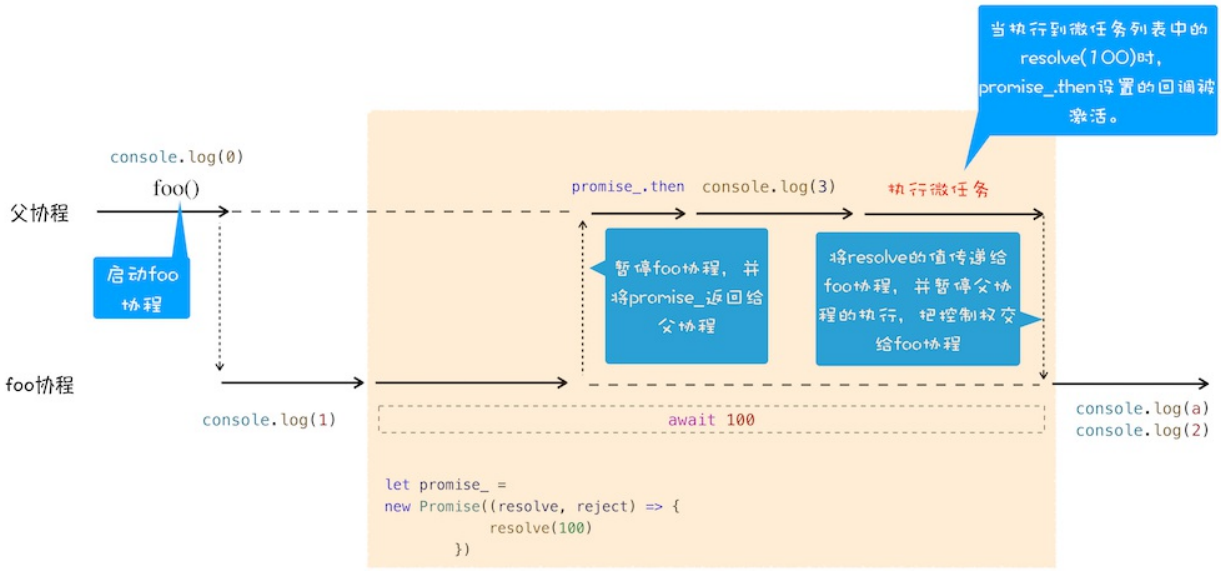
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事

情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
```

```
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
      console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

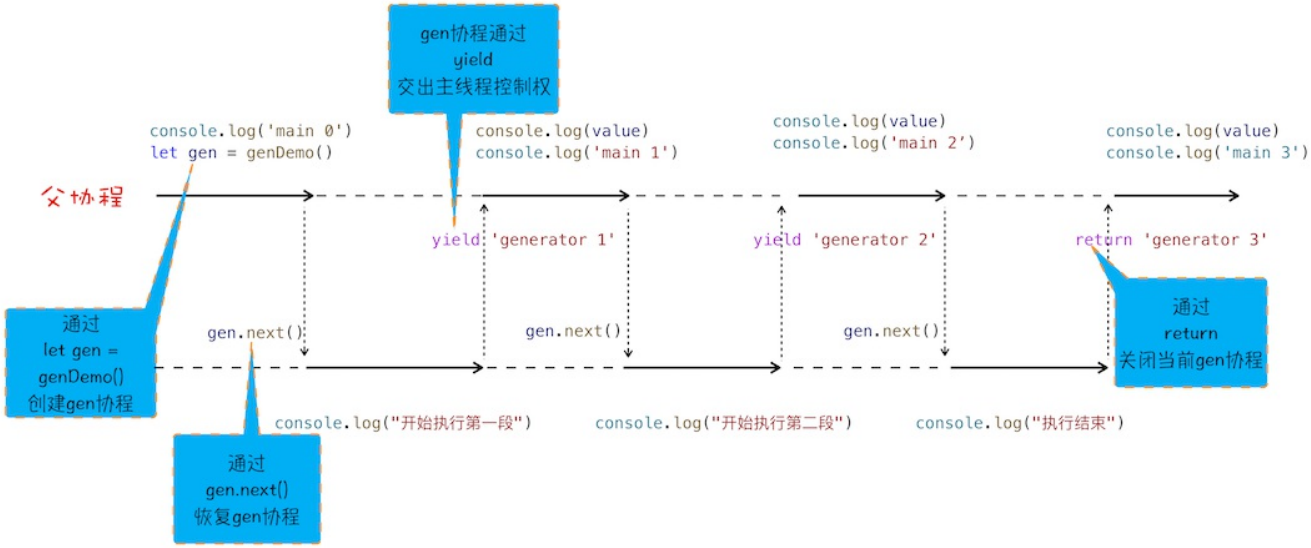
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

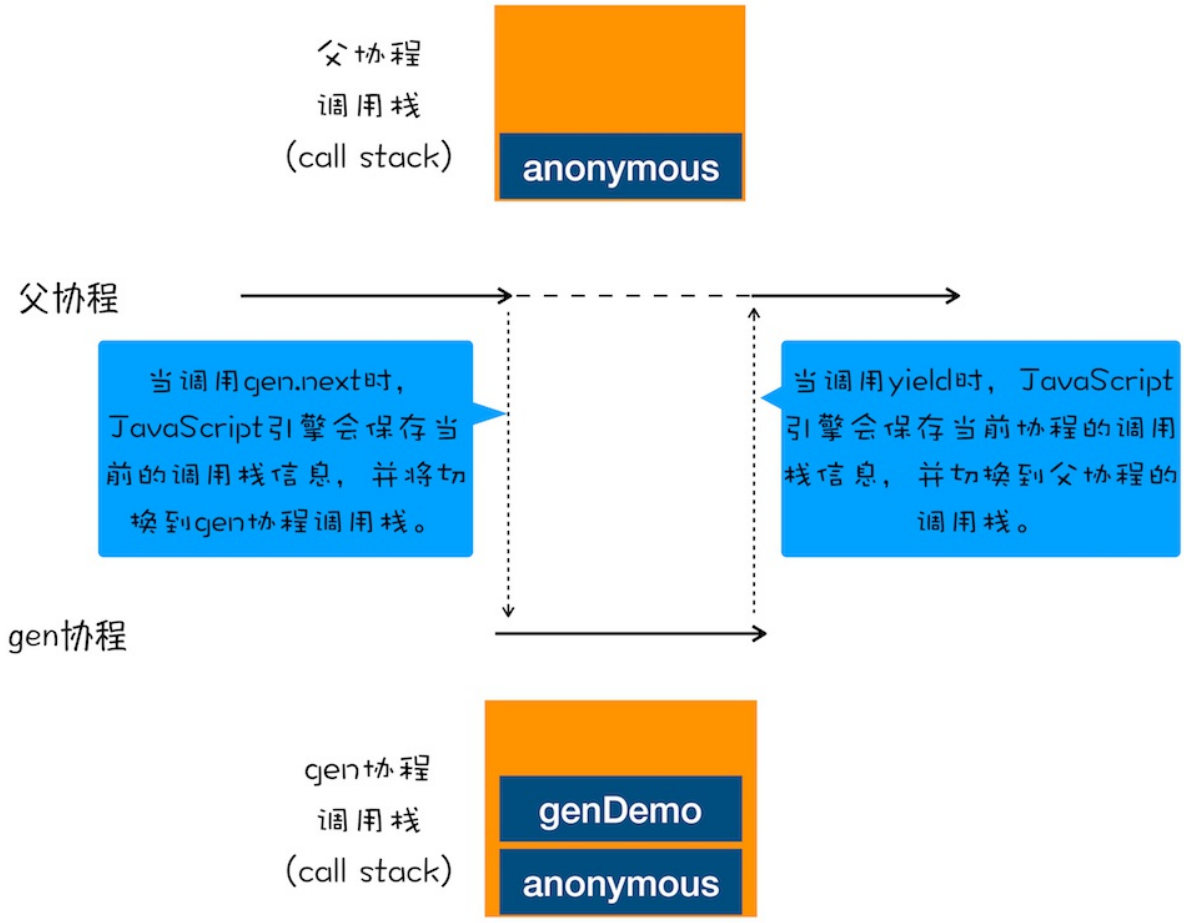
不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

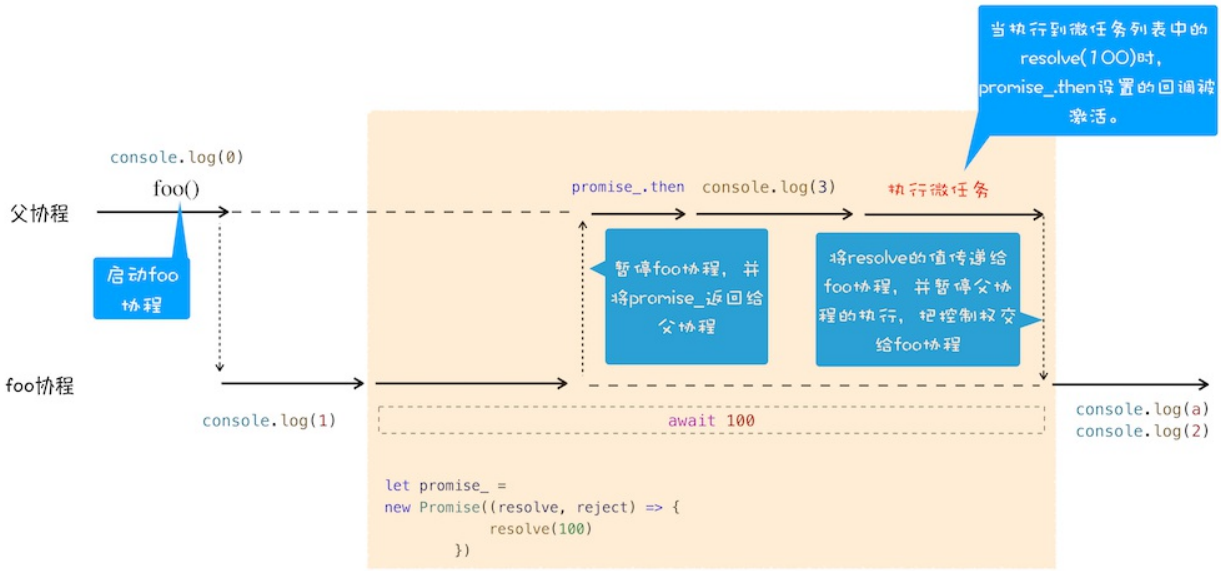
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
       console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

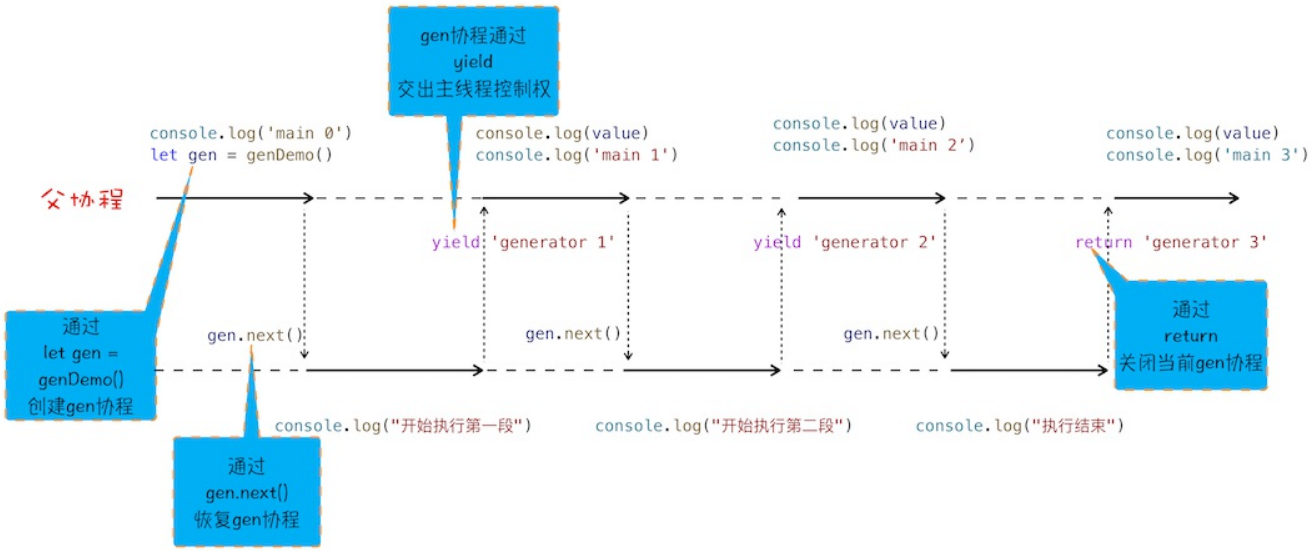
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切

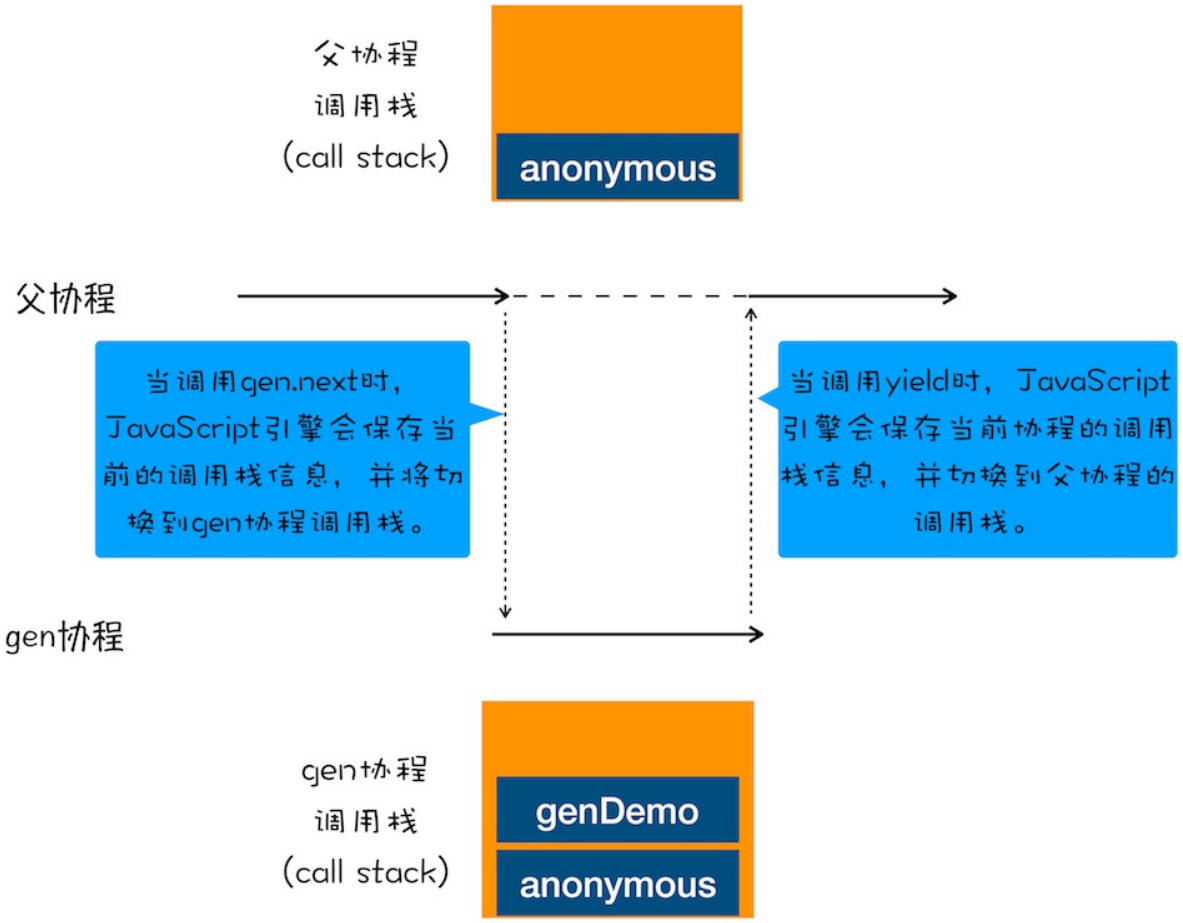
换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

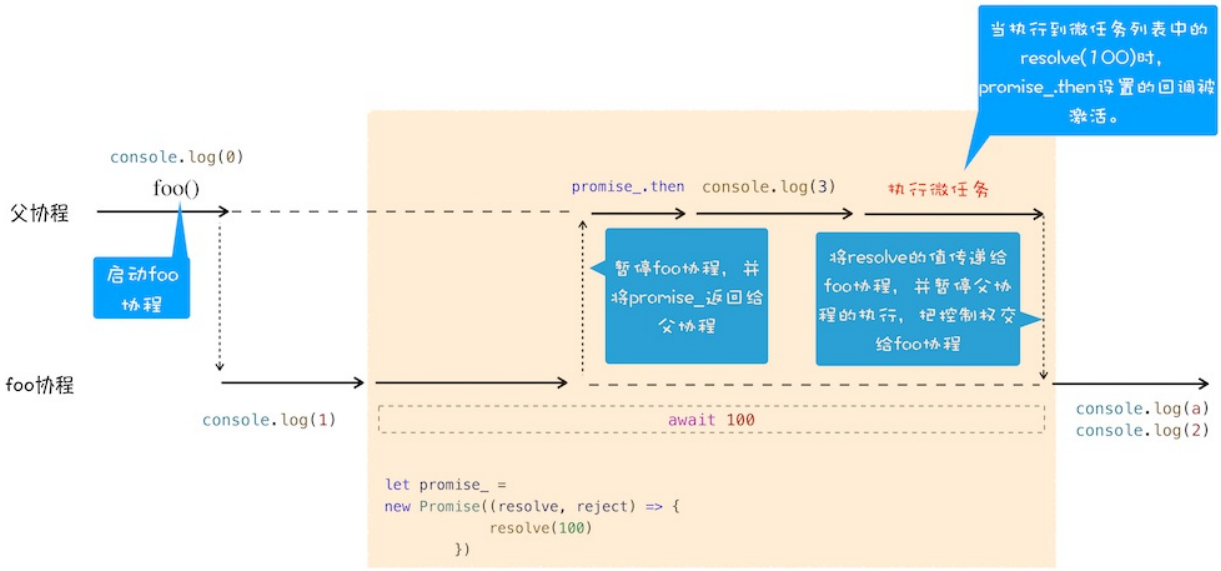
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事

情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
```

```
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
      console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

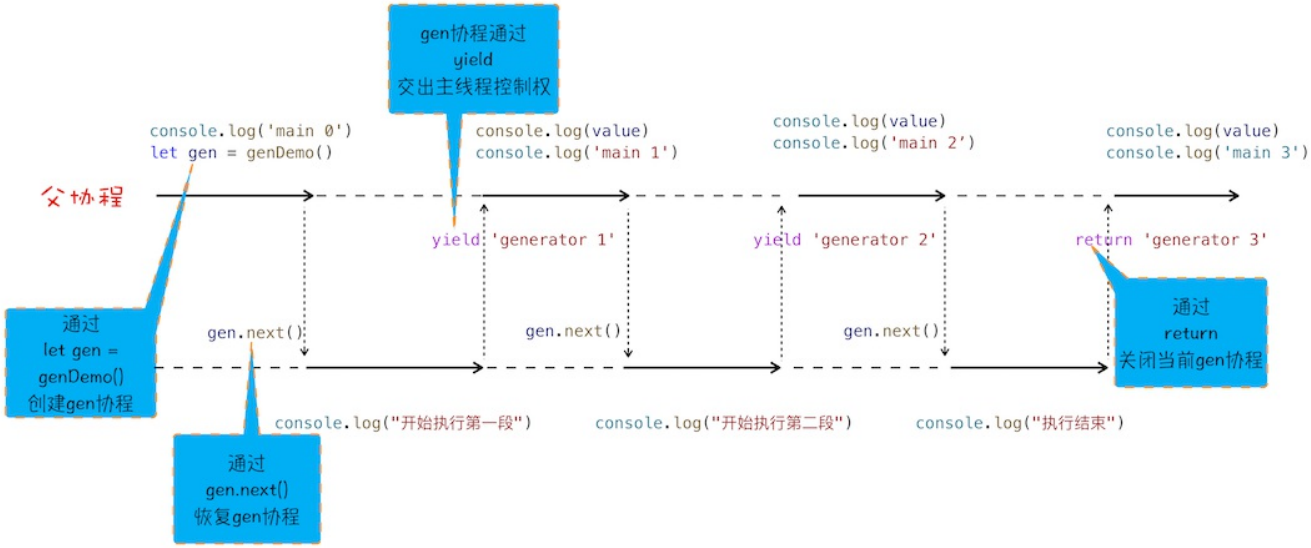
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

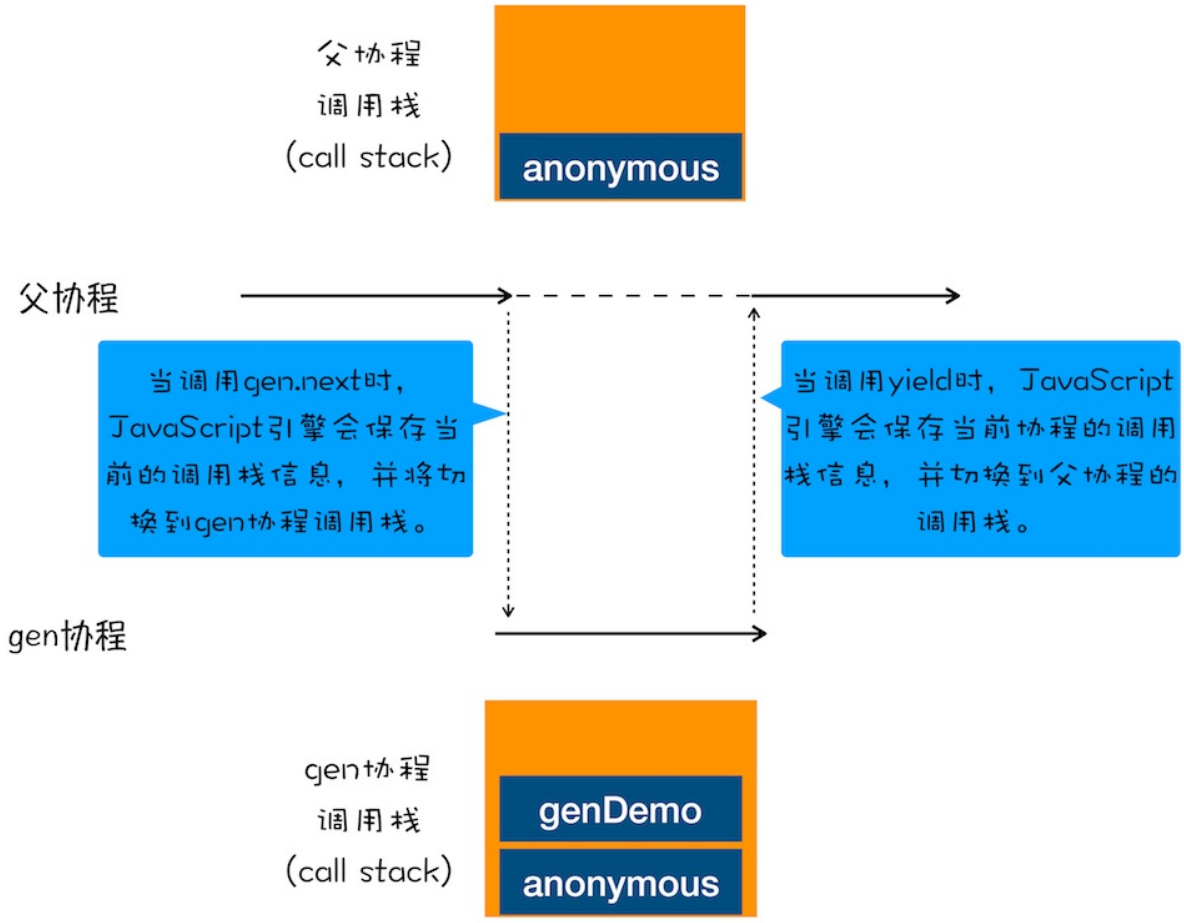
不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

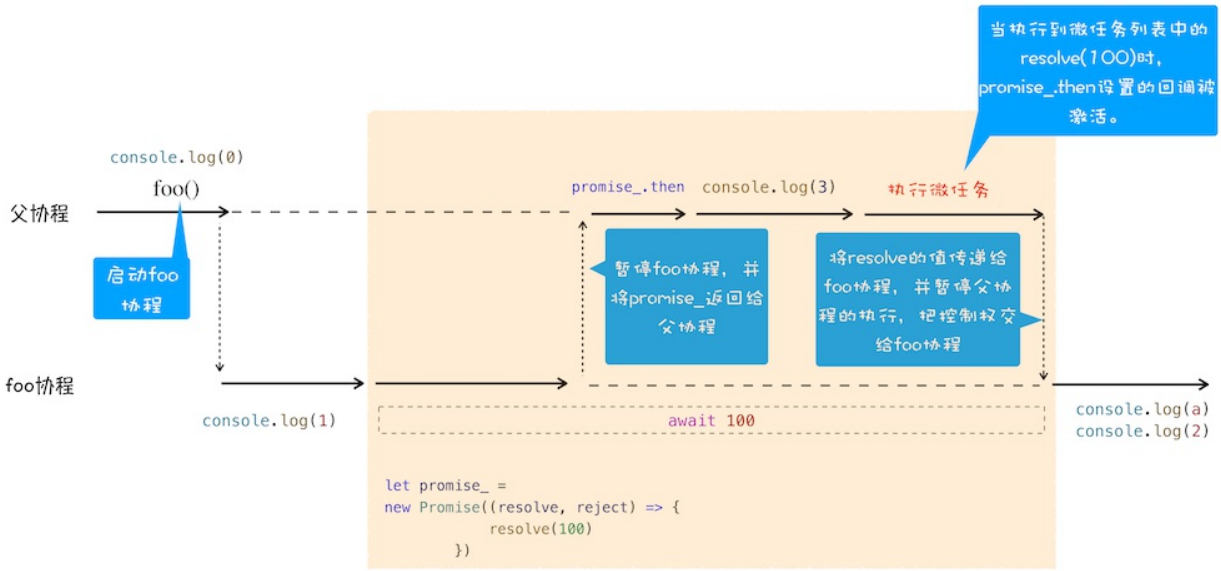
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
       console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

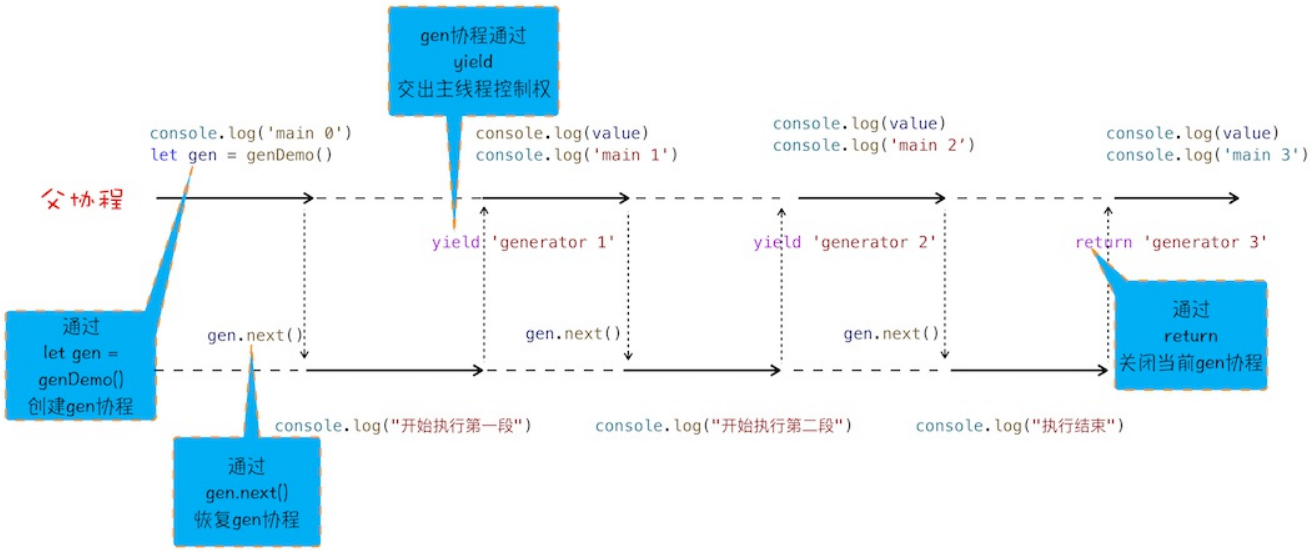
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切

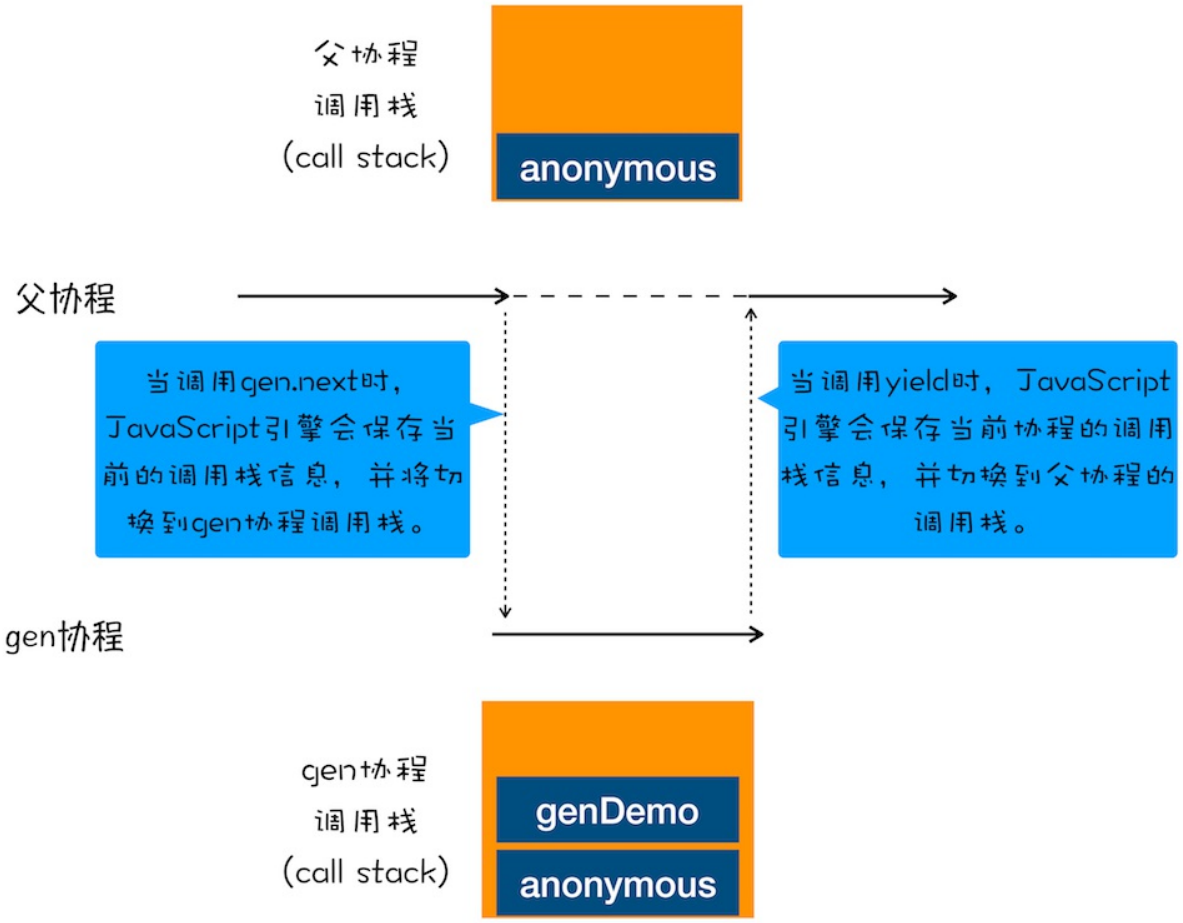
换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

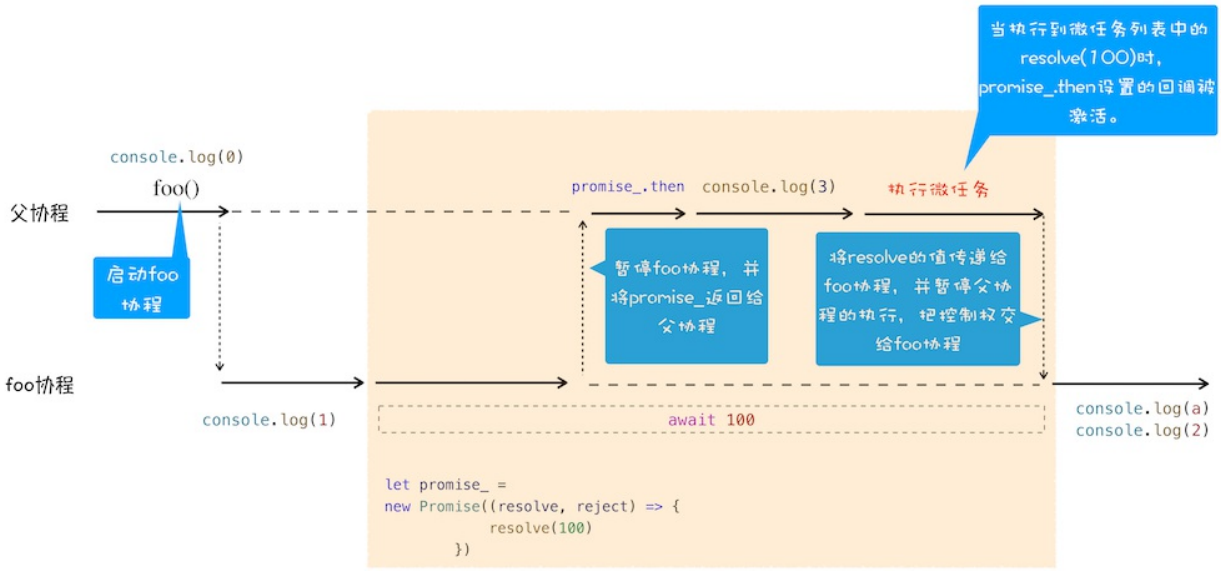
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事

情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
```

```
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
      console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

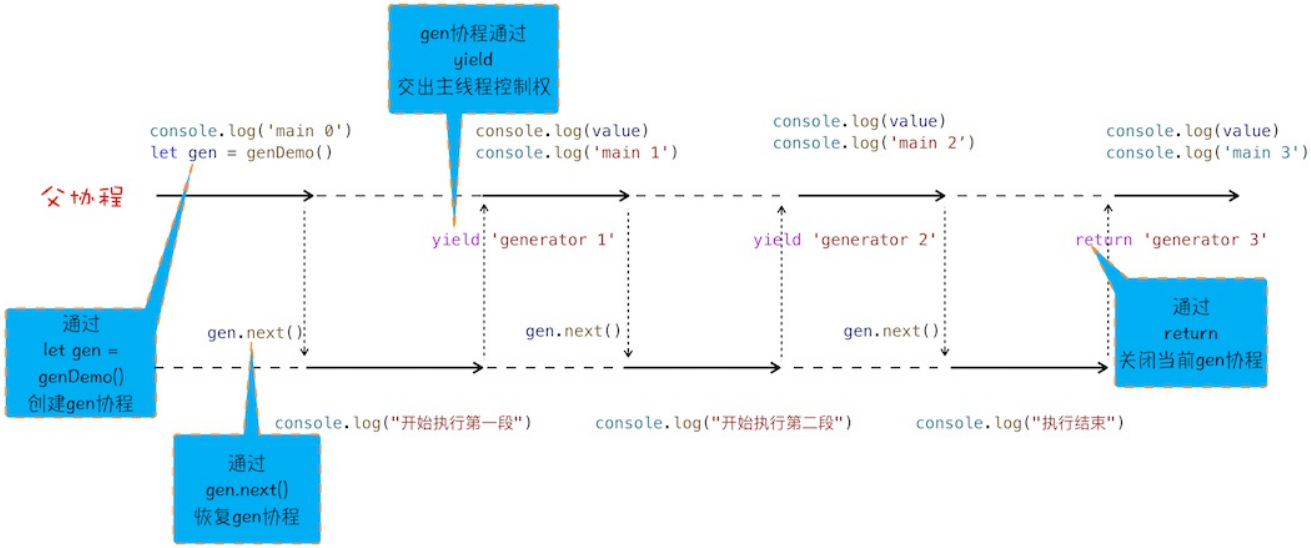
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

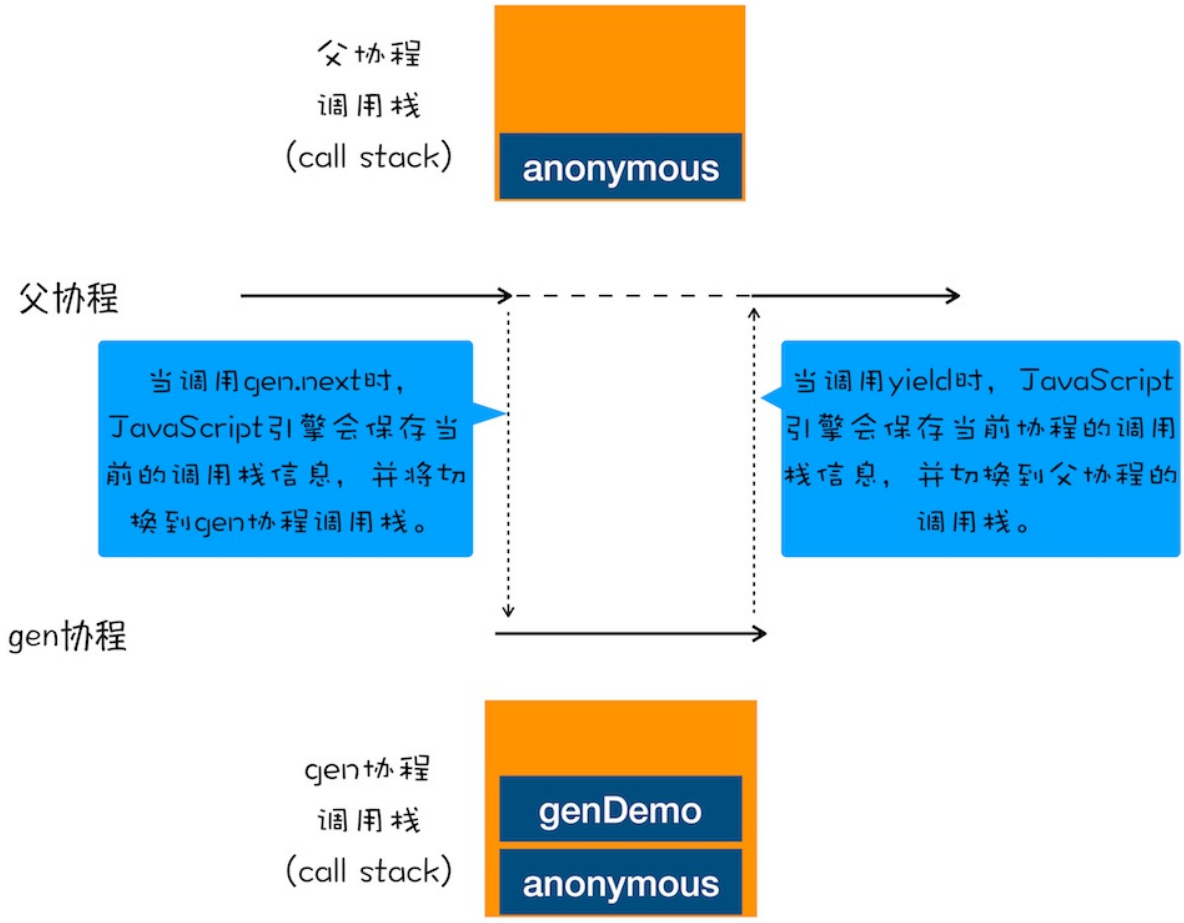
不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

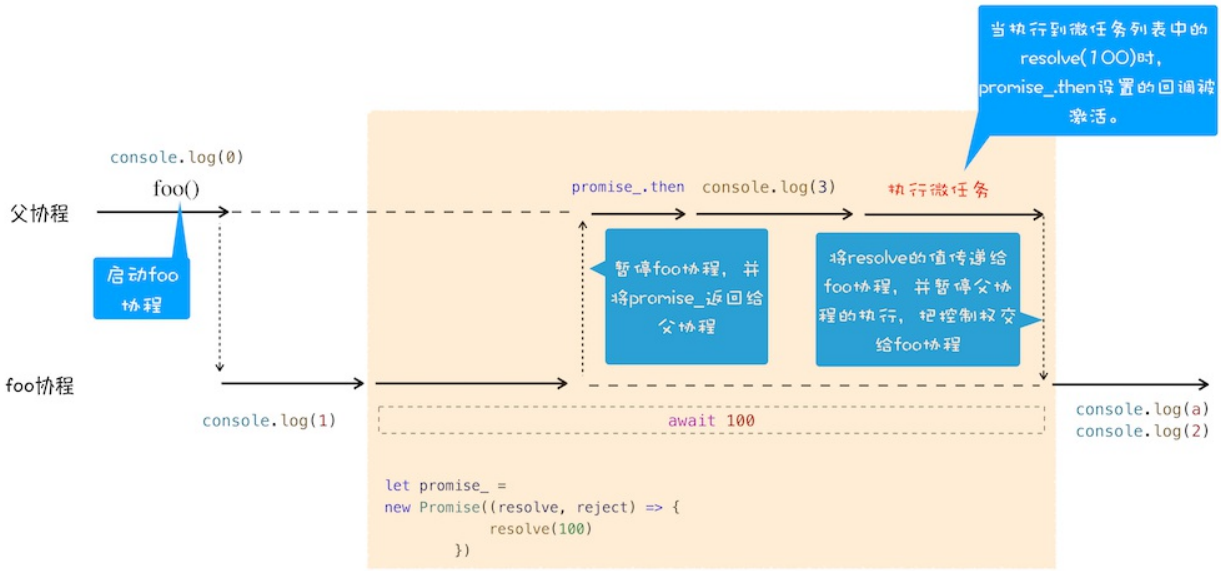
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

在上篇文章中，我们介绍了怎么使用Promise来实现回调操作，使用Promise能很好地解决回调地狱的问题，但是这种方式充满了Promise的then()方法，如果处理流程比较复杂的话，那么整段代码将充斥着then，语义化不明显，代码不能很好地表示执行流程。

比如下面这样一个实际的使用场景：我先请求极客邦的内容，等返回信息之后，我再请求极客邦的另外一个资源。下面代码展示的是使用fetch来实现这样的需求，fetch被定义在window对象中，可以用它来发起对远程资源的请求，该方法返回的是一个Promise对象，这和我们上篇文章中讲的XFetch很像，只不过fetch是浏览器原生支持的，并有没利用XMLHttpRequest来封装。

```
fetch('https://www.geekbang.org')
      .then((response) => {
          console.log(response)
          return fetch('https://www.geekbang.org/test')
      }).then((response) => {
          console.log(response)
      }).catch((error) => {
          console.log(error)
      })
```

从这段Promise代码可以看出来，使用promise.then也是相当复杂，虽然整个请求流程已经线性化了，但是代码里面包含了大量的then函数，使得代码依然不是太容易阅读。**基于这个原因，ES7 引入了async/await，这是JavaScript异步编程的一个重大改进，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力，并且使得代码逻辑更加清晰**。你可以参考下面这段代码：

```
async function foo(){
  try{
    let response1 = await fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = await fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
  }catch(err) {
       console.error(err)
  }
}
foo()
```

通过上面代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持try catch来捕获异常，这就是完全在写同步代码，所以是非常符合人的线性思维的。但是很多人都习惯了异步回调的编程思维，对于这种采用同步代码实现异步逻辑的方式，还需要一个转换的过程，因为这中间隐藏了一些容易让人迷惑的细节。

那么本篇文章我们继续深入，看看JavaScript引擎是如何实现async/await的。如果上来直接介绍async/await的使用方式的话，那么你可能会有点懵，所以我们就从其最底层的技术点一步步往上讲解，从而带你彻底弄清楚async和await到底是怎么工作的。

本文我们首先介绍生成器（Generator）是如何工作的，接着讲解Generator的底层实现机制——协程（Coroutine）；又因为async/await使用了Generator和Promise两种技术，所以紧接着我们就通过Generator和Promise来分析async/await到底是如何以同步的方式来编写异步代码的。

生成器 VS 协程

我们先来看看什么是生成器函数？

生成器函数是一个带星号函数，而且是可以暂停执行和恢复执行的。我们可以看下面这段代码：

```
function* genDemo() {
    console.log("开始执行第一段")
    yield 'generator 2'

    console.log("开始执行第二段")
    yield 'generator 2'

    console.log("开始执行第三段")
    yield 'generator 2'

    console.log("执行结束")
    return 'generator 2'
}

console.log('main 0')
let gen = genDemo()
console.log(gen.next().value)
console.log('main 1')
console.log(gen.next().value)
console.log('main 2')
console.log(gen.next().value)
console.log('main 3')
console.log(gen.next().value)
console.log('main 4')
```

执行上面这段代码，观察输出结果，你会发现函数genDemo并不是一次执行完的，全局代码和genDemo函数交替执行。其实这就是生成器函数的特性，可以暂停执行，也可以恢复执行。下面我们就来看看生成器函数的具体使用方式：

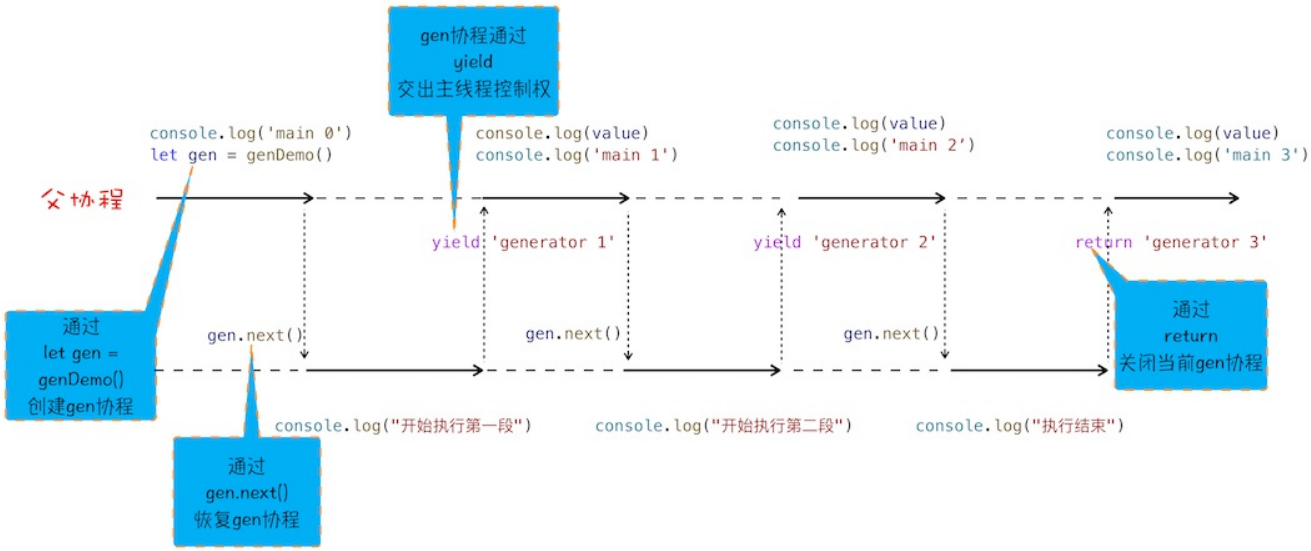
1. 在生成器函数内部执行一段代码，如果遇到yield关键字，那么JavaScript引擎将返回关键字后面的内容给外部，并暂停该函数的执行。
2. 外部函数可以通过next方法恢复函数的执行。

关于函数的暂停和恢复，相信你一定很好奇这其中的原理，那么接下来我们就来简单介绍下JavaScript引擎V8是如何实现一个函数的暂停和恢复的，这也会有助于你理解后面要介绍的async/await。

要搞懂函数为何能暂停和恢复，那你首先要了解协程的概念。**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程，比如当前执行的是A协程，要启动B协程，那么A协程就需要将主线程的控制权交给B协程，这就体现在A协程暂停执行，B协程恢复执行；同样，也可以从B协程中启动A协程。通常，**如果从A协程启动B协程，我们就把A协程称为B协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



协程执行流程图

从图中可以看出来协程的四点规则：

1. 通过调用生成器函数genDemo来创建一个协程gen，创建之后，gen协程并没有立即执行。
2. 要让gen协程执行，需要通过调用gen.next。
3. 当协程正在执行的时候，可以通过yield关键字来暂停gen协程的执行，并返回主要信息给父协程。
4. 如果协程在执行期间，遇到了return关键字，那么JavaScript引擎会结束当前协程，并将return后面的内容返回给父协程。

不过，对于上面这段代码，你可能又有这样疑问：父协程有自己的调用栈，gen协程时也有自己的调用栈，当gen协程通过yield把控制权交给父协程时，V8是如何切

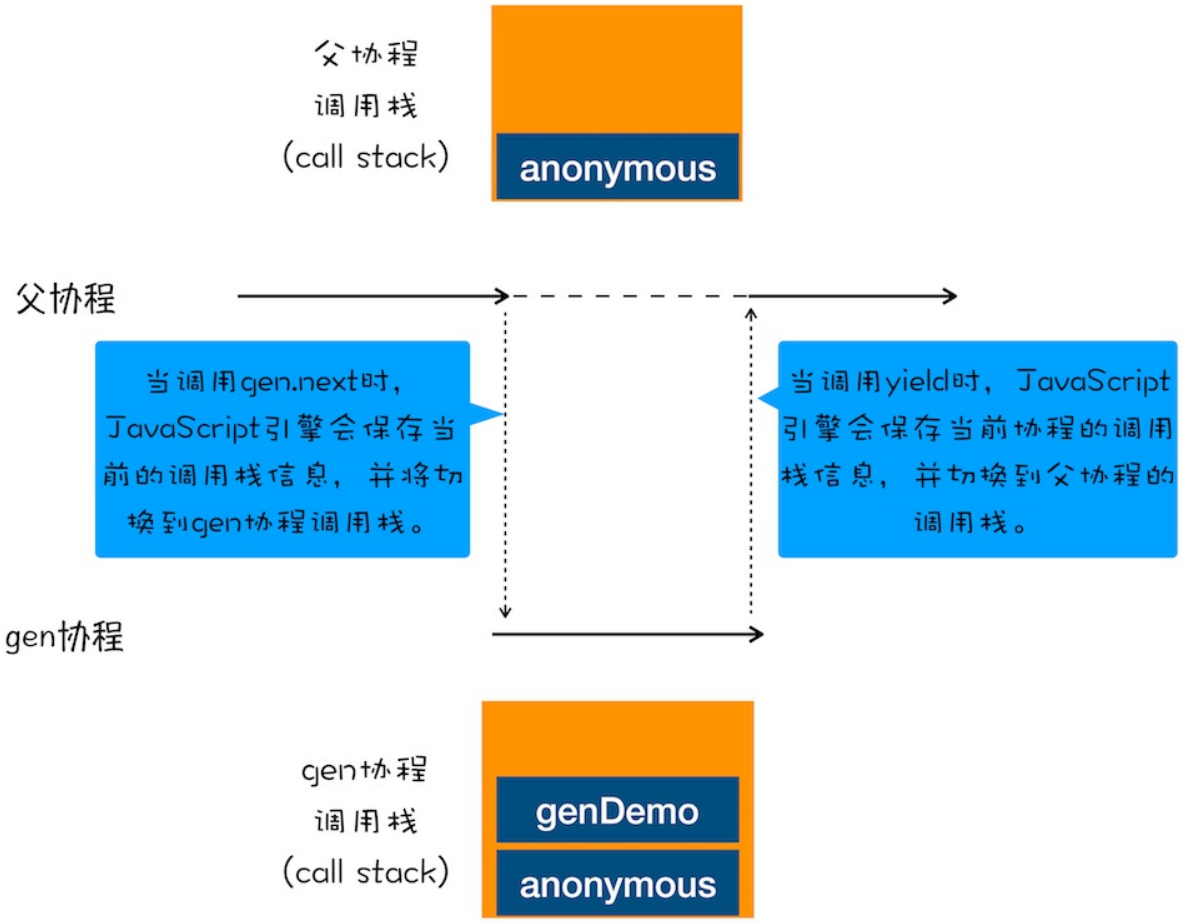
换到父协程的调用栈？当父协程通过gen.next恢复gen协程时，又是如何切换gen协程的调用栈？

要搞清楚上面的问题，你需要关注以下两点内容。

第一点：gen协程和父协程是在主线程上交互执行的，并不是并发执行的，它们之前的切换是通过yield和gen.next来配合完成的。

第二点：当在gen协程中调用了yield方法时，JavaScript引擎会保存gen协程当前的调用栈信息，并恢复父协程的调用栈信息。同样，当在父协程中执行gen.next时，JavaScript引擎会保存父协程的调用栈信息，并恢复gen协程的调用栈信息。

为了直观理解父协程和gen协程是如何切换调用栈的，你可以参考下图：



gen协程和父协程之间的切换

到这里相信你已经弄清楚了协程是怎么工作的，其实在JavaScript中，生成器就是协程的一种实现方式，这样相信你也就理解什么是生成器了。那么接下来，我们使用生成器和Promise来改造开头的那段Promise代码。改造后的代码如下所示：

```
//foo函数
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}

//执行foo函数的代码
let gen = foo()
function getGenPromise(gen) {
    return gen.next().value
}
getGenPromise(gen).then((response) => {
    console.log('response1')
    console.log(response)
    return getGenPromise(gen)
}).then((response) => {
    console.log('response2')
    console.log(response)
})
```

从图中可以看到，foo函数是一个生成器函数，在foo函数里面实现了用同步代码形式来实现异步操作；但是在foo函数外部，我们还需要写一段执行foo函数的代码，如上述代码的后半部分所示，那下面我们就来分析下这段代码是如何工作的。

- 首先执行的是`let gen = foo()`，创建了gen协程。
- 然后在父协程中通过执行gen.next把主线程的控制权交给gen协程。
- gen协程获取到主线程的控制权后，就调用fetch函数创建了一个Promise对象response1，然后通过yield暂停gen协程的执行，并将response1返回给父协程。
- 父协程恢复执行后，调用response1.then方法等待请求结果。
- 等通过fetch发起的请求完成之后，会调用then中的回调函数，then中的回调函数拿到结果之后，通过调用gen.next放弃主线程的控制权，将控制权交gen协程继续执行下个请求。

以上就是协程和Promise相互配合执行的一个大致流程。不过通常，我们把执行生成器的代码封装成一个函数，并把这个执行生成器代码的函数称为**执行器**（可参考著名的co框架），如下面这种方式：

```
function* foo() {
    let response1 = yield fetch('https://www.geekbang.org')
    console.log('response1')
    console.log(response1)
    let response2 = yield fetch('https://www.geekbang.org/test')
    console.log('response2')
    console.log(response2)
}
co(foo());
```

通过使用生成器配合执行器，就能实现使用同步的方式写出异步代码了，这样也大大加强了代码的可读性。

async/await

虽然生成器已经能很好地满足我们的需求了，但是程序员的追求是无止境的，这不又在ES7中引入了async/await，这种方式能够彻底告别执行器和生成器，实现更加直观简洁的代码。其实async/await技术背后的秘密就是Promise和生成器应用，往低层说就是微任务和协程应用。要搞清楚async和await的工作原理，我们就得对async和await分开分析。

1. async

我们先来看看async到底是什么？根据MDN定义，async是一个通过**异步执行**并**隐式返回 Promise** 作为结果的函数。

对async函数的理解，这里需要重点关注两个词：**异步执行**和**隐式返回 Promise**。

关于异步执行的原因，我们一会儿再分析。这里我们先来看看是如何隐式返回Promise的，你可以参考下面的代码：

```
async function foo() {
    return 2
}
console.log(foo())  // Promise {<resolved>: 2}
```

执行这段代码，我们可以看到调用async声明的foo函数返回了一个Promise对象，状态是resolved，返回结果如下所示：

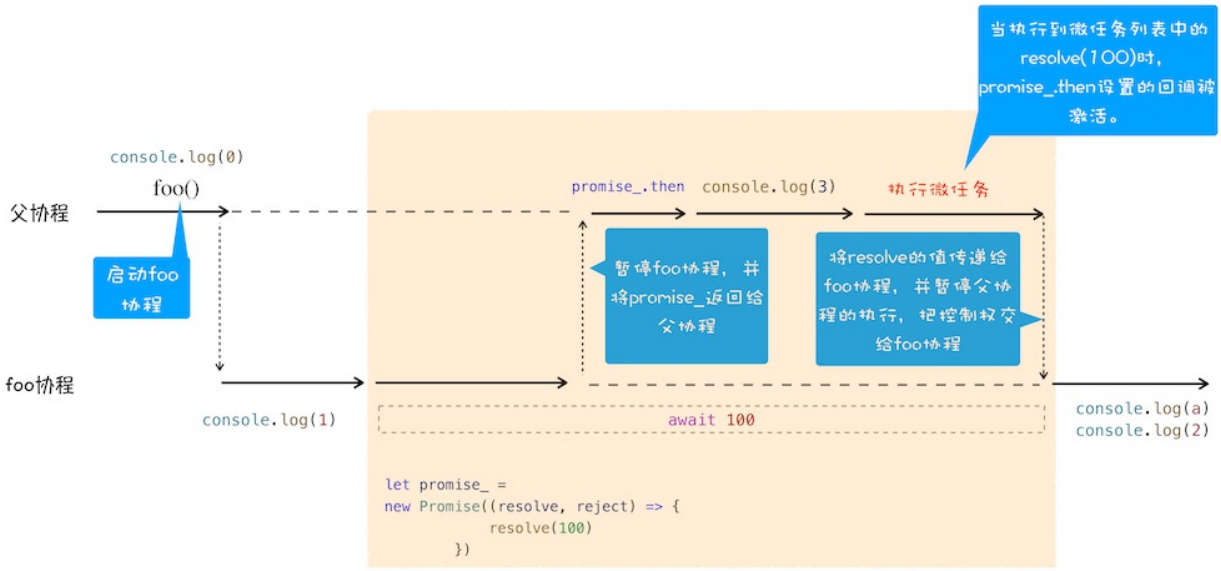
```
Promise {<resolved>: 2}
```

2. await

我们知道了async函数返回的是一个Promise对象，那下面我们再结合文中这段代码来看看await到底是什么。

```
async function foo() {
    console.log(1)
    let a = await 100
    console.log(a)
    console.log(2)
}
console.log(0)
foo()
console.log(3)
```

观察上面这段代码，你能判断出打印出来的内容是什么吗？这得先来分析async结合await到底会发生什么。在详细介绍之前，我们先站在协程的视角来看看这段代码的整体执行流程图：



async/await执行流程图

结合上图，我们来一起分析下async/await的执行流程。

首先，执行`console.log(0)`这个语句，打印出来0。

紧接着就是执行foo函数，由于foo函数是被async标记过的，所以当进入该函数的时候，JavaScript引擎会保存当前的调用栈等信息，然后执行foo函数中的`console.log(1)`语句，并打印出1。

接下来就执行到foo函数中的`await 100`这个语句了，这里是我们分析的重点，因为在执行`await 100`这个语句时，JavaScript引擎在背后为我们默默做了太多的事

情，那么下面我们就把这个语句拆开，来看看JavaScript到底都做了哪些事情。

当执行到`await 100`时，会默认创建一个Promise对象，代码如下所示：

```
let promise_ = new Promise((resolve,reject){
  resolve(100)
})
```

在这个promise_对象创建的过程中，我们可以看到在executor函数中调用了resolve函数，JavaScript引擎会将该任务提交给微任务队列（上一篇文章中我们讲解过）。

然后JavaScript引擎会暂停当前协程的执行，将主线程的控制权转交给父协程执行，同时会将promise_对象返回给父协程。

主线程的控制权已经交给父协程了，这时候父协程要做的一件事是调用promise_.then来监控promise状态的改变。

接下来继续执行父协程的流程，这里我们执行`console.log(3)`，并打印出来3。随后父协程将执行结束，在结束之前，会进入微任务的检查点，然后执行微任务队列，微任务队列中有`resolve(100)`的任务等待执行，执行到这里的时候，会触发promise_.then中的回调函数，如下所示：

```
promise_.then((value)=>{
   //回调函数被激活后
  //将主线程控制权交给foo协程，并将vaule值传给协程
})
```

该回调函数被激活以后，会将主线程的控制权交给foo函数的协程，并同时将value值传给该协程。

foo协程激活之后，会把刚才的value值赋给了变量a，然后foo协程继续执行后续语句，执行完成之后，将控制权归还给父协程。

以上就是await/async的执行流程。正是因为async和await在背后为我们做了大量的工作，所以我们才能用同步的方式写出异步代码来。

总结

好了，今天就介绍到这里，下面我来总结下今天的主要内容。

Promise的编程模型依然充斥着大量的then方法，虽然解决了回调地狱的问题，但是在语义方面依然存在缺陷，代码中充斥着大量的then函数，这就是async/await出现的原因。

使用async/await可以实现用同步代码的风格来编写异步代码，这是因为async/await的基础技术使用了生成器和Promise，生成器是协程的实现，利用生成器能实现生成器函数的暂停和恢复。

另外，V8引擎还为async/await做了大量的语法层面包装，所以了解隐藏在背后的代码有助于加深你对async/await的理解。

async/await无疑是异步编程领域非常大的一个革新，也是未来的一个主流的编程风格。其实，除了JavaScript，Python、Dart、C#等语言也都引入了async/await，使用它不仅能让代码更加整洁美观，而且还能确保该函数始终都能返回Promise。

思考时间

下面这段代码整合了定时器、Promise和async/await，你能分析出来这段代码执行后输出的内容吗？

```
async function foo() {
    console.log('foo')
}
async function bar() {
    console.log('bar start')
    await foo()
    console.log('bar end')
}
console.log('script start')
setTimeout(function () {
    console.log('setTimeout')
}, 0)
bar();
new Promise(function (resolve) {
    console.log('promise executor')
    resolve();
}).then(function () {
    console.log('promise then')
})
console.log('script end')
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。