一、异步与同步

理解异步与同步

1、在生活中，按照字面量来理解，异步指的是，比如我先吃完苹果再看电视，这是我们生活中理解的异步。同步就是我边吃苹果边看电视。

2、然而，对我们的电脑程序来说，同步与异步的概念恰恰跟我们在生活中的理解完全相反。同步指的是我先吃完苹果，然后再看电视（**执行完一个事件再去执行另一个事件，若上一个事件没执行完，下一个事件就无法执行**）；异步指的是我边吃苹果边看电视（**多个事件可以同时执行，不会发生阻塞**）。理解js中的异步和同步，如果对这两个概念混淆了，你只要想到跟我们生活中异步与同步的理解相反就行了。

谈谈JavaScript的单线程机制

单线程是JavaScript中一个比较重要的特性。这就表示在同一个时间只能做一件事情。

那为什么JavaScript不弄成多线程的呢？还能更加充分利用CPU呢。

这要从JavaScript的使用场景说起，JavaScript作为浏览器语言，主要用途是通过操作DOM，与用户进行交互。

我们设想一下，如果一个线程去更新某个DOM元素，而另一个线程去删除这个DOM元素，那么浏览器该执行哪个操作呢？这就出现冲突了。

因此，为了避免复杂的多线陈机制，JavaScript从设计之初就选择了单线程标准。

Event Loop 事件循环

单线程那就表示事件只能一个一个轮着执行，前面没执行完后面就无法执行（只能干等着，这样也太低效了）。比如，我用ajax向服务端请求一个数据（假设数据量很大，花费时间很久），这时候就会卡在那里，因为后面的只能等前一个事件把数据成功请求回来再执行接下的代码，这样对用户的体验就太不友好了。这时候 **任务队列** 就出场了。JS中将所有任务分为**同步任务**和 **异步任务**。

目前个人的理解是，只有所有的同步任务执行完才会去执行异步任务。常见的异步任务有setTimeout函数，http请求，数据库查询等。

> （1）所有任务都在主线程上执行，形成一个执行栈（execution context stack）。
> （2）主线程之外，还存在一个"任务队列"（task queue）。系统把异步任务放到"任务队 列"之中，然后继续执行后续的任务。
> （3）一旦"执行栈"(**事件循环队列**)中的所有任务**执行完毕**，系统就会读取"任务队列"。如果这个时候，异步任务已经结束了等待状态，就会从"任务队列"进入执行栈，恢复执行。
> （4）主线程不断重复上面的第三步。

详情查看此文章

```
    console.log('这里是调试1');
    setTimeout(() => {
        console.log('这里是调试2');
    },0)
    console.log('这里是调试3');
    console.log('这里是调试4');
    console.log('这里是调试5');
    // 输出
    这里是调试1
    这里是调试3
    这里是调试4
    这里是调试5
    这里是调试2 // setTimeout中的最后面才输出来
```

异步任务之微任务&宏任务

异步任务分为宏任务（**macrotasks**）和微任务（**microtasks**）。常见的宏任务有（`setTimeout`，`setInterval`，`setImmediate`），常见的微任务有（`Promise.then`）。在一个事件循环中，当执行完所有的同步任务后，会在任务队列中取出异步任务，这时候会**优先执行微任务**，接着**再执行宏任务**。换句话说，微任务就是为了插队而存在的。看一下下面这个例子：

```
setTimeout(() => console.log(1));
new Promise((resolve) => {
   resolve();
   console.log(2);
}).then(() => {
   console.log(3);
})
console.log(4);
```

输入结果为：2 - 4 - 3 - 1

setTimeout(...)相关

1、在`setTimeout(...)`中可以用`async(...)`，`await(...)`，但是只能当前`setTimeout(...)`执行栈中生效；
2、同一层级的多个`setTimeout(...)`相互独立，就算`setTimeout(...)`中有`await(...)`也不会影响到其他`setTimeout(...)`的执行；

阮一峰 JavaScript运行机制详解
朴灵 标注
mdn

选择异步还是同步？

1、在JS中默认的方式是同步的，个别是异步的（http请求，setTimeout）。假设现在有一个场景，我们需要从后端接口获取数据（异步任务），然后对该数据进行处理，这时候我们就需要将获取数据这一异步任务**阻塞**，也就是将其转化为同步任务。

2、常见的处理异步（将异步变为同步）的方式按照出现时间排序有：**回调函数**，**Promise**，**async与await**。

二、闭包与回调函数

> 1、无论通过何种手段将**内部函数**传递到这个函数定义时的**词法作用域**(定义时的词法作用域指的是函数定义时的那个位置)**以外**执行，内部函数都会持有对原始定义作用域的引用，无论在何处执行这个函数都会使用闭包。
> 2、无论函数在哪里被调用，也无论它如何被调用，它的词法作用域都只由函数被声明时所处的位置决定。
> 3、闭包使得函数可以继续访问定义时的词法作用域。
> 4、只要使用了回调函数，实际上就是在使用闭包。
> ---《你不知道的JavaScript》

1、一般来说，block（块级）作用域被执行完后，里面的数据就被回收了。

2、作用： 将某个块级作用域里面的变量可以被外部使用。

3、词法作用域：静态作用域，查找作用域的顺序是按照函数定义时的位置来决定的。

4、全局作用域是在V8启动过程中就创建了，且一直保存在内存中不会被销毁的，直到V8退出。而函数作用域是在执行该函数时创建的，当函数执行结束之后，函数作用域就随之被销毁了。

我们来看一个闭包的案例：

```
function foo() {
    let a = 2;
    function bar() {
        console.log(a);
        a++;
    }
    return bar;
}
let baz = foo();
baz(); // 2
baz(); // 3
```

函数`foo(...)`中返回了一个名为`bar`的函数，接着`let baz = foo();`就将`bar(...)`这个函数赋值给了`baz`，所以在一定程度上baz相当于`bar(...)`，接着我们就执行`baz`（在`bar`定义时的词法作用域以外执行了）（词法作用域： 函数在哪里定义，他的词法作用域就在哪里）；这也就印证了我们前面说的，**将一个内部函数传递到这个函数所在的词法作用域以外执行**，所以就产生了闭包，因为我们在外面获取到了foo函数内部的变量a，同时这也是闭包产生的效果。

3、总结：

- 什么时候产生闭包： 将一个函数传递到这个函数所在的词法作用域以外执行；
- 闭包的作用： 引用某个块级作用域里面变量（持续引用）

4、只要使用了回调函数，实际上就是在使用闭包。下面我们来看一个案例。

```
function wait(message) {
    setTimeout(function timer() {
      console.log(message);
    }, 1000);
}
wait('Hello, World');
```

将一个内部函数（名为timer）传递给`setTimeout(...)`。timer具有涵盖`wait(...)`作用域的闭包，因此还保有对变量`message`的引用。`wait(...)`执行1000毫秒后，它的内部作用域并不会消失，timer函数依然保有`wait(...)`作用域的闭包。因为它可以引用到`message`这个变量。

5、下面我们来看一个稍微复杂点的案例（利用回调函数进行同步传值）：

```
// 模拟获取ajax请求的数据(异步)
function getAjaxData(cb) {
    // 用setTimeout实现异步请求
    setTimeout(function() {
      // 假设data是我们请求得到的数据 我们需要将数据发送给别人
      const data = "请求得到的数据";
      cb(data);
    }, 1000)
}
// 获取ajax请求的响应数据并对数据进行处理
getAjaxData(function handleData(tempData) {
    tempData = tempData + '666';
    console.log(tempData); // 请求得到的数据666
});
```

将`handleData(...)`作为参数传进`getAjaxData(...)`中，因此`cb(data)`中的**data**就作为参数传进了`handleData(...)`中，这样也就达到了传值的作用了。

6、回调函数存在的问题：**信任问题**。以上面的例子进行改进。

```
function getAjaxData (cb) {
  // 用setTimeout实现异步请求
  setTimeout(function () {
    // 假设data是我们请求得到的数据 我们需要将数据发送给别人
    const data = "请求得到的数据";
    cb(data);
    cb(data);
  }, 1000)
}
// 获取ajax请求的响应数据并对数据进行处理
getAjaxData(function handleData (tempData) {
  tempData = tempData + '666';
  console.log(tempData); // 请求得到的数据666
});
```

假设`getAjaxData(...)`这个方法是由第三方库引进来的，我们并不清楚里面的代码逻辑细节，这样的话`handleData(...)`的执行就存在不确定性，比如上面我增加了一个`cb(data)`，这`handleData(...)`就会执行两次，当然这不是我们想要的效果，因此回调的处理就不可控了。

> 回调最大的问题是控制反转，它会导致信任链的完全断裂。
> 因为回调函数内部的调用情况是不确定的，可能不调用，也可能被调用了多次。
> ---《你不知道的JavaScript》

Promise的出现正是为了解决这个问题。

7、备注： 关于js中内存回收的问题

三、Promise

> Promise一旦决议（`resolve`），一直保持其决议结果不变
> 解决回调调用过早的问题
> 解决回调调用过晚的问题
> 解决回调未调用的问题
> 解决调用次数过少或过多

Promise API概述

1、new Promise(...)构造器

2、Promise.resolve(...)和Promise.reject(...)

3、then(...)和catch(...)

4、Promise.all([...])和Promise.race([...])

Promise源码解读

1、new Promise 时，需要传递一个executor执行器，执行器立刻执行；

2、executor 接受两个参数，分别是resolve和reject；

3、promise 只能从 pending 到 rejected，或者从 pending 到 fulfilled；

4、promise 的状态一旦确认，就不会再改变；

5、promise都有then方法，then接受两个参数，分别是promise成功的回调 onFulfilled，和promise失败的回调onRejected；

6、如果调用then时，promise已经成功，则执行onFulfilled，并将promise的值作为参数传递进去。如果promise已经失败，那么执行onRejected，并将promise失败的原因作为参数传递进去。如果promise的状态是pending，需要将onFulfilled和onRejected函数存放起来，等待状态确定后，再依次将对应的函数执行；

7、then的参数onFulfilled和onRejected可以缺省；

8、promise可以then多次，promise的then方法返回一个promise；

9、如果then返回的是一个结果，那么就会把这个结果作为参数，传递给下一个then的成功的回调（onFulfilled）；

10、如果then中抛出异常，那么就会把这个异常作为参数，传递给下一个then的失败的回调（onRejected）；

11、如果then返回的是一个promise，那么就会等这个promise执行完，promise如果成功，就走下一个then的成功，如果失败，就走下一个then的失败；

流程

1、当promise中有异步任务：如果promise中有异步任务的话，那他的status为pedding，之后所有的then都会放到待办任务数组里面。
2、链式调用的实现：then方法返回一个promise。
3、当then中有异步任务：两种情况：

- 仅仅是一个异步任务：按照正常情况来，会再最后面输出。
- 异步任务在一个promise中，并且该promise有return：后面的所有then会放进待办任务数组中，这种情况会等到return的promise `resolve(...)`之后，将状态改变之后才会再执行后面的(遍历

待办任务数组)，也就是相当于将`then(...)`中的异步任务阻塞了。

源码实现&解析

1、实现`promise(...)`方法，该方法有一个回调函数`exectue`参数；

当我们`new promise((resolve, reject) => {});`时，将会执行该回调函数 ；

并且`resolve`对应到`promise(...)`中的`res(...)`；

`reject`对应到promise(...)中的`rej(...)`；

当我们执行到`resolve(...)`时，才会执行`res(...)`；

```
function promise(exectue) {
  const res = (value) => { }
  const rej = (reason) => { }
  exectue(res, rej);
}

// resolve对应的是promise(...)中的res(...)
// reject对应的是promise(...)中的rej(...)
// 因此只有执行了resolve(...)或reject(...) 才改变status的值
const test = new promise((resolve, reject) => {
  console.log('这里是调试1');
  setTimeout(() => {
    console.log('这里是调试2');
    resolve();
  }, 3000)
})

// 这里是调试1
// 这里是调试2
// 执行了res
```

2、`promise(...)`方法中维护几个变量，用于存储**执行节点**的状态&数据：

- `fulfilled`：标志任务是否已完成状态；
- `pedding`：标志任务是否正在进行中状态（未完成状态）；
- `status`：标志当前执行节点的状态，节点的初始状态为`pedding`；
- `value`：存储`promise`状态成功时的值；
- `reason`：存储`promise`状态失败时的值；
- `onFulfilledCallbacks`：数组，存储成功的回调任务；
- `onRejectedCallbacks`：数组，存储失败的回调任务；

```
function promise(exectue) {
  this.fulfilled = 'fulfilled';
  this.pedding = 'pedding';
  this.rejected = 'rejected';
  this.status = this.pedding;
  this.value;
  this.reason;
  this.onFulfilledCallbacks = [];
  this.onRejectedCallbacks = [];
  const res = (value) => {
    if(this.status === this.pedding) {
      // 执行了resolve就要改变status为fulfilled
```

```
      this.status = this.fulfilled;
      this.value = value;
      // 遍历执行放在待办任务数组中的事件
      this.onFulfilledCallbacks.forEach(fn => fn());
    }
  }
  const rej = (reason) => {
    if(this.status === this.pedding) {
      // 执行了reject就要改变status为rejected
      this.status = this.rejected;
      this.reason = reason;
      this.onRejectedCallbacks.forEach(fn => fn());
    }
  }
  exectue(res, rej);
}
```

3、实现`then(...)`，为符合链式调用，`then(...)`方法必须返回一个`promise(...)`：

```
promise.prototype.then = function (onFulfilled, onRejected) {
  // this指向promise(...)对象
  const that = this;
  const promise2 = new promise((resolve, reject) => {
    if(that.status === that.fulfilled) {
      onFulfilled();
      resolve();
    }
    if(that.status === that.pedding) { }
    if(that.status === that.rejected) { }
  })
  return promise2;
}
// 调用
const test = new promise((resolve, reject) => {
  console.log('这里是调试1');
  resolve();
})
test.then(() => {
  console.log('这里是调试2');
}).then(() => {
  console.log('这里是调试3');
})

// 这里是调试1
// 这里是调试2
// 这里是调试3
```

4、到这里，我们已实现了基本的链式调用的功能了，那如果在`promise(...)`中是一个异步事件时，并且我们需要阻塞这个异步任务（将异步转化为同步），预计的效果是只有执行了`resolve(...)`才能执行`then(...)`中的代码，可以怎么实现呢？

当`promise(...)`中是需要阻塞的异步任务时，那么当执行到`then(...)`时，此时的`status`为`pedding`，需要将链式调用的执行节点根据`Fulfilled`或`Rejected`两种情况分别添加到`onFulfilledCallbacks`或`onRejectedCallbacks`这两个变量中，等到异步任务执行完，`resolve(...)`之后才轮到链式调用节点的执行，改进代码：

```
promise.prototype.then = function (onFulfilled, onRejected) {
  // this指向promise(...)对象
  const that = this;
  const promise2 = new promise((resolve, reject) => {
    if(that.status === that.fulfilled) {
      onFulfilled();
      resolve();
    }
    if(that.status === that.pedding) {
      console.log('这里是调试4');
      this.onFulfilledCallbacks.push(() => {
        onFulfilled();
        resolve();
      })
    }
    if(that.status === that.rejected) {
      this.onRejectedCallbacks.push(() => {
        onRejected();
        reject();
      })
    }
  })
  return promise2;
}

// 调用
const test = new promise((resolve, reject) => {
  setTimeout(() => {
    console.log('这里是调试1');
    resolve();
  }, 1000)
})
test.then(() => {
  console.log('这里是调试2');
}).then(() => {
  console.log('这里是调试3');
})

// 这里是调试4
// 这里是调试4
// 这里是调试1
// 这里是调试2
// 这里是调试3
```

5、接下来，这里又有一个场景，当我们在 `then(...)` 方法中有异步任务，我们同样想要让该异步任务阻塞，又该怎么弄呢？按照我们上面的逻辑是阻塞不了 `then(...)` 方法中的异步任务的，大家可以尝试一下。

```
const test = new promise((resolve, reject) => {
  setTimeout(() => {
    console.log('这里是调试1');
    resolve();
  }, 1000)
})
test.then(() => {
  setTimeout(() => {
    console.log('这里是调试2');
```

```
  }, 2000)
}).then(() => {
  console.log('这里是调试3');
})

// 输出
// 这里是调试4
// 这里是调试4
// 这里是调试1
// 这里是调试3
// 这里是调试2
```

要想阻塞`then(...)`中的异步任务，这里得分成两种情况来讨论：一种是要阻塞异步任务，需要在`then(...)`返回一个`promise(...)`，另一种是不需要阻塞异步任务，那就不需要返回任何值。继续改进代码：

```
promise.prototype.then = function (onFulfilled, onRejected) {
  // this指向promise(...)对象
  const that = this;
  const promise2 = new promise((resolve, reject) => {
    if(that.status === that.fulfilled) {
        let x = onFulfilled(that.value);
        if(x && typeof x === 'object' || typeof x === 'function') {
          let then = x.then;
          if(then) {
            then.call(x, resolve, reject);
          } else {
            resolve(x);
          }
    }
    if(that.status === that.pedding) {
      this.onFulfilledCallbacks.push(() => {
        // onFulfilled(...) 为then(...)方法中的回调函数
        let x = onFulfilled(that.value); // 执行了then(...)方法中的回调函数并将返回值
赋给x
        // 判断then(...)中回调函数返回值类型 是否为function
        if(x && typeof x === 'object' || typeof x === 'function') {
          let then = x.then;
          // 如果then存在 说明返回了一个promise
          if(then) {
            // 这里采用的方法是将resolve放在返回的promise的then里面执行 .
            // 这样只有当then(...)方法中的异步事件执行完才resolve(...) 很牛皮
            // call(...)绑定this的指向
            then.call(x, resolve, reject);
          }
        } else {
          resolve(x);
        }
      })
    }
    if(that.status === that.rejected) {
      this.onRejectedCallbacks.push(() => {
        let x = onRejected(that.reason);
        if(x && typeof x === 'object' || typeof x === 'function') {
          let then = x.then;
          if(then) {
            then.call(x, resolve, reject);
```

```
            } else {
              resolve(x);
            }
        })
      }
    })
    return promise2;
}


const test = new promise((resolve, reject) => {
  setTimeout(() => {
    console.log('这里是调试1');
    resolve();
  }, 1000)
})
test.then(() => {
  // 若要阻塞then(...)中的异步任务需要return一个promise
  return new promise((resolve1, reject1) => {
    setTimeout(() => {
      console.log('这里是调试2');
      resolve1();
    }, 2000)
  })
}).then(() => {
  console.log('这里是调试3');
})

// 这里是调试1
// 这里是调试2
// 这里是调试3
```

6、到这里，关于`promise`核心的内容基本已经实现了，再看一下以下案例：

```
new promise(function(resolve){
    console.log('这里是调试1');
    resolve();
}).then(function(){
    console.log('这里是调试2');
});
console.log('这里是调试3');

// 这里是调试1
// 这里是调试2
// 这里是调试3
```

```
new Promise(function(resolve){
    console.log('这里是调试1');
    resolve();
}).then(function(){
    console.log('这里是调试2');
});
console.log('这里是调试3');

// 这里是调试1
// 这里是调试3
// 这里是调试2
```

可以发现，我们的`promise`和原生的`Promise`输出结果不一样。根据`Promise`规范，在`.then(...)`方法中执行的程序属于`微任务`（异步任务分为`微任务`和`宏任务`，一般的异步事件属于宏任务，**微任务比宏任务先执行**，但我们自己实现的`promise`并不支持这个规则，原生的`Promise`才支持），因此需要将其转化为异步任务，下面我们用`setTimeout(...)`来模拟实现一下异步任务（原生`Promise`并不是用`setTimeout(...)`实现的）

```
promise.prototype.then = function (onFulfilled, onRejected) {
  const that = this;
  const promise2 = new promise((resolve, reject) => {
    if (that.status === that.fulfilled) {
      setTimeout(() => {
        let x = onFulfilled(that.value);
        if (x && typeof x === 'object' || typeof x === 'function') {
          let then = x.then;
          if (then) {
            then.call(x, resolve, reject);
          }
        } else {
          resolve(x);
        }
      })
    }
    if (that.status === that.pedding) {
      that.onFulfilledCallbacks.push(() => {
        setTimeout(() => {
          let x = onFulfilled(that.value);
          if (x && typeof x === 'object' || typeof x === 'function') {
            let then = x.then;
            if (then) {
              then.call(x, resolve, reject);
            }
          } else {
            resolve(x);
          }
        })
      })
      that.onRejectedCallbacks.push(() => {
        setTimeout(() => {
          let x = onRejected(that.reason);
          if (x && typeof x === 'object' || typeof x === 'function') {
            let then = x.then;
            if (then) {
              then.call(x, resolve, reject);
            }
```

```
          } else {
            resolve(x);
          }
        })
      })
    }
    if (that.status === that.rejected) {
      setTimeout(() => {
        let x = onRejected(that.reason);
        if (x && typeof x === 'object' || typeof x === 'function') {
          let then = x.then;
          if (then) {
            then.call(x, resolve, reject);
          }
        } else {
          resolve(x);
        }
      })
    }
  })
  return promise2;
}

// 这里是调试1
// 这里是调试3
// 这里是调试2
```

7、再来看一个案例：

```
const test = new promise((res, rej) => {
  setTimeout(() => {
    console.log('这里是调试1');
    res();
  }, 4000);
})
// 回调函数push进test.onFulfilledCallbacks数组中
test.then(() => {
  return new promise((res1) => {
    setTimeout(() => {
      console.log('这里是调试2');
      res1();
    }, 3000);
  })
})
// 回调函数也是push进test.onFulfilledCallbacks数组中，因此两个then(...)是相互独立的，执行顺序按照常规的来处理
test.then(() => {
  console.log('这里是调试3');
})

// 输出
// 这里是调试1
// 这里是调试3
// 这里是调试2
```

源码实现参考
宏任务微任务参考

四、async与await

- **async、await相对于promise多了等待的效果**;
- 原理： Promise + 生成器 = async/await;
- `async`函数一定会返回一个Promise对象。如果一个async函数的返回值看起来不是promise，那么它将会被隐式得包装在一个promise中;
- **重要: 使用async与await时，await后面的函数必须返回决议的Promise，不然，程序将停止在await这一步;**
- async、await相对与promise多了等待的效果；具体是怎么实现的？--- 使用了`yield`暂停函数，只有等前面的`promise`决议之后才执行`next(...)`，程序才继续往后面执行，表现出来的就是暂停程序的效果;

迭代器

1、一种特殊对象，用于为迭代过程设计的专有接口，简单来说就类似与遍历一个对象;

2、调用返回一个结果对象，对象中有next()方法;

3、next()方法返回一个对象： value: 表示下一个要返回的值；done：没有值可以返回时为true，否则为false;

4、迭代器简易内部实现:

```
function createIterator(items) {
  let i = 0;
  return {
    next: function() {
      let done = (i >= items.length);
      // items[i++] 相当于 items[i]; i++;
      let value = !done ? items[i++] : undefined;
      return {
        value: value,
        done: done // true表示后面没有值了
      };
    }
  };
}
// 只创建了一个实例 因此iterator一直是同一个
let iterator = createIterator([1,2,3]);
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

生成器

1、一个返回**迭代器**的函数;实现了一种顺序、看似同步的异步流程控制表达风格;
2、关键字yield：**表示函数暂停运行**;
3、过程：每当执行完一条yield语句后函数就暂停了,直到再次调用函数的next()方法才会继续执行后面的语句;
4、使用yield关键字可以返回任何值或表达式（表示要传递给下一个过程的值）;
5、注意点:

- **yield关键字只能在生成器函数的直接内部使用(在生成器函数内部的函数中使用会报错);**
- 不能用箭头函数来创建生成器;

简易使用示例：

```
function *createInterator() {
  yield 1;
  yield 2;
  yield 3;
}
// 生成器的调用方式与普通函数相同,只不过返回一个迭代器
let iterator = createInterator();
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

迭代消息传递：

```
function *foo(x) {
  const y = x * (yield);
  return y;
}
const it = foo(6);
// 启动foo(...)
it.next();
const res = it.next(7); // next(...)的数量总是比yield的多一个
console.log(res); // 42
```

首先，传入6作为参数。然后调用`it.next(...)`，这会启动`*foo(...)`；

接着，在`*foo(...)`内部，开始执行语句`const y = x ...`，但随后就遇到了一个`yield`表达式。它就会在这一点上**暂停**`*foo(...)`（在赋值语句中间），并非在本质上要求调用代码为`yield`表达式提供一个结果值。接下来，调用`it.next(7)`，这一句把值7传回作为被暂停的`yield`表达式结果。

所以，这是赋值语句实际上就是`const y = 6 * 7`。现在，`return y`返回值42作为调用`it.next(7)`的结果。

注意，根据你的视角不同，`yield`和`next(...)`调用有一个不匹配。一般来说，需要的`next(...)`调用要比`yield`语句多一个，前面的代码片段就有一个`yield`和两个`next(...)`调用。

因为第一个`next(...)`总是用来启动生成器，并运行到第一个`yield`处。不过，是第二个`next(...)`调用完成第一个被暂停的`yield`表达式，第三个`next(...)`完成第二个`yield`，以此类推。

消息是双向传递的：

`yield(...)`作为一个表达式，可以发出消息响应`next(...)`；

`next(...)`也可以向暂停的`yield`表达式发送值。注意：前面我们说到，第一个`next(...)`总是用来启动生成器的，此时没有暂停的`yield`来接受这样一个值，规范和浏览器都会默认丢弃传递给第一个`next(...)`的任何东西。因此，启动生成器一定要用不带参数的`next(...)`；

```
function *foo(x) {
  const y = x * (yield "Hello");
  return y;
}
const it = foo(6);
let res = it.next(); // 第一个next()，并不传入任何东西
console.log(res.value); // Hello
res = it.next(7); // 向等待的yield传入7
console.log(res.value); // 42
```

源码实现&解析

`async`&`await`是基于生成器+Promise封装的语法糖，使用`async`修饰的`function`实际上会被转化为一个生成器函数，并返回一个决议的`Promise(...)`，`await`对应到的是`yield`，因此`await`只能在被`async`修饰的方法中使用

下面看一个例子：这是一个常见的使用`async/await`的案例

```
// 模拟多个异步情况
function foo1() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(100); // 如果没有决议(resolve)就不会运行下一个
    }, 3000)
  })
}
function foo2() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(200);
    }, 2500)
  })
}
async function test() {
  const f1 = await foo1();
  const f2 = await foo2();
  console.log('这里是调试1');
  console.log(f1);
  console.log(f2);
  return 'haha';
}
test().then((data) => {
  console.log('这里是调试2');
  console.log(data);
});
// 这里是调试1
// 100
// 200
// 这里是调试2
// haha
```

接着我们来看一下如何将这个案例中的**async/await用生成器+Promise**来实现：

将`test(...)`转化为生成器函数：

```
function *test() {
  const f1 = yield foo1();
  const f2 = yield foo2();
  console.log('这里是调试1');
  console.log(f1);
  console.log(f2);
  return 'haha';
}
```

实现方法`run(...)`，用于调用生成器`*test(...)`：

```
function run(generatorFunc) {
  return function() {
    const gen = generatorFunc();
    // async 默认返回Promise
    return new Promise((resolve, reject) => {
      function step(key, arg) {
        let generatorResult;
        try {
          // 相当于执行gen.next(...)
          generatorResult = gen[key](arg);
        } catch (error) {
          return reject(error);
        }
        // gen.next() 得到的结果是一个 { value, done } 的结构
        const { value, done } = generatorResult;
        if (done) {
          return resolve(value);
        } else {
          return Promise.resolve(value).then(val => step('next', val), err =>
step('throw', err));
        }
      }
      step("next");
    })
  }
}

const haha = run(test);
haha().then((data) => {
  console.log('这里是调试2');
  console.log(data);
});

// 这里是调试1
// 100
// 200
// 这里是调试2
// haha
```

async/await 参考