```
    const cancelToken = new CancelToken((cancelCallback) =>
      cancelButton.addEventListener("click", cancelCallback));

    cancelToken.promise.then(() => clearTimeout(id));
  });
}

startButton.addEventListener("click", () => cancellableDelayedResolve(1000));
</script>
```

每次单击“Start”按钮都会开始计时，并实例化一个新的 CancelToken 的实例。此时，“Cancel”按钮一旦被点击，就会触发令牌实例中的期约解决。而解决之后，单击“Start”按钮设置的超时也会被取消。

2. 期约进度通知

执行中的期约可能会有不少离散的“阶段”，在最终解决之前必须依次经过。某些情况下，监控期约的执行进度会很有用。ECMAScript 6 期约并不支持进度追踪，但是可以通过扩展来实现。

一种实现方式是扩展 Promise 类，为它添加 notify()方法，如下所示：

```
class TrackablePromise extends Promise {
  constructor(executor) {
    const notifyHandlers = [];

    super((resolve, reject) => {
    return executor(resolve, reject, (status) => {
      notifyHandlers.map((handler) => handler(status));
    });
  });

    this.notifyHandlers = notifyHandlers;
  }

  notify(notifyHandler) {
    this.notifyHandlers.push(notifyHandler);
    return this;
  }
}
```

这样，TrackablePromise 就可以在执行函数中使用 notify()函数了。可以像下面这样使用这个函数来实例化一个期约：

```
let p = new TrackablePromise((resolve, reject, notify) => {
  function countdown(x) {
    if (x > 0) {
      notify(`${20 * x}% remaining`);
      setTimeout(() => countdown(x - 1), 1000);
    } else {
      resolve();
    }
  }

  countdown(5);
});
```

这个期约会连续 5 次递归地设置 1000 毫秒的超时。每个超时回调都会调用 notify()并传入状态值。假设通知处理程序简单地这样写：

```
...

let p = new TrackablePromise((resolve, reject, notify) => {
  function countdown(x) {
    if (x > 0) {
      notify(`${20 * x}% remaining`);
      setTimeout(() => countdown(x - 1), 1000);
    } else {
      resolve();
    }
  }

  countdown(5);
});

p.notify((x) => setTimeout(console.log, 0, 'progress:', x));

p.then(() => setTimeout(console.log, 0, 'completed'));

//  (约 1 秒后) 80% remaining
//  (约 2 秒后) 60% remaining
//  (约 3 秒后) 40% remaining
//  (约 4 秒后) 20% remaining
//  (约 5 秒后) completed
```

notify()函数会返回期约，所以可以连缀调用，连续添加处理程序。多个处理程序会针对收到的每条消息分别执行一遍，如下所示：

```
...

p.notify((x) => setTimeout(console.log, 0, 'a:', x))
 .notify((x) => setTimeout(console.log, 0, 'b:', x));

p.then(() => setTimeout(console.log, 0, 'completed'));

//  (约 1 秒后)  a: 80% remaining
//  (约 1 秒后)  b: 80% remaining
//  (约 2 秒后)  a: 60% remaining
//  (约 2 秒后)  b: 60% remaining
//  (约 3 秒后)  a: 40% remaining
//  (约 3 秒后)  b: 40% remaining
//  (约 4 秒后)  a: 20% remaining
//  (约 4 秒后)  b: 20% remaining
//  (约 5 秒后)  completed
```

总体来看，这还是一个比较粗糙的实现，但应该可以演示出如何使用通知报告进度了。

> **注意** ES6 不支持取消期约和进度通知，一个主要原因就是这样会导致期约连锁和期约合成过度复杂化。比如在一个期约连锁中，如果某个被其他期约依赖的期约被取消了或者发出了通知，那么接下来应该发生什么完全说不清楚。毕竟，如果取消了 Promise.all()中的一个期约，或者期约连锁中前面的期约发送了一个通知，那么接下来应该怎么办才比较合理呢？

11.3 异步函数

异步函数，也称为“async/await”（语法关键字），是 ES6 期约模式在 ECMAScript 函数中的应用。async/await 是 ES8 规范新增的。这个特性从行为和语法上都增强了 JavaScript，让以同步方式写的代码

能够异步执行。下面来看一个最简单的例子，这个期约在超时之后会解决为一个值：

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));
```

这个期约在 1000 毫秒之后解决为数值 3。如果程序中的其他代码要在这个值可用时访问它，则需要写一个解决处理程序：

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));

p.then((x) => console.log(x));  // 3
```

这其实是很不方便的，因为其他代码都必须塞到期约处理程序中。不过可以把处理程序定义为一个函数：

```
function handler(x) { console.log(x); }

let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));

p.then(handler); // 3
```

这个改进其实也不大。这是因为任何需要访问这个期约所产生值的代码，都需要以处理程序的形式来接收这个值。也就是说，代码照样还是要放到处理程序里。ES8 为此提供了 async/await 关键字。

11.3.1　异步函数

ES8 的 async/await 旨在解决利用异步结构组织代码的问题。为此，ECMAScript 对函数进行了扩展，为其增加了两个新关键字：async 和 await。

1. async

async 关键字用于声明异步函数。这个关键字可以用在函数声明、函数表达式、箭头函数和方法上：

```
async function foo() {}

let bar = async function() {};

let baz = async () => {};

class Qux {
  async qux() {}
}
```

使用 async 关键字可以让函数具有异步特征，但总体上其代码仍然是同步求值的。而在参数或闭包方面，异步函数仍然具有普通 JavaScript 函数的正常行为。正如下面的例子所示，foo()函数仍然会在后面的指令之前被求值：

```
async function foo() {
  console.log(1);
}

foo();
console.log(2);

// 1
// 2
```

不过，异步函数如果使用 return 关键字返回了值（如果没有 return 则会返回 undefined），这个值会被 Promise.resolve()包装成一个期约对象。异步函数始终返回期约对象。在函数外部调用这

个函数可以得到它返回的期约：

```
async function foo() {
  console.log(1);
  return 3;
}

// 给返回的期约添加一个解决处理程序
foo().then(console.log);

console.log(2);

// 1
// 2
// 3
```

当然，直接返回一个期约对象也是一样的：

```
async function foo() {
  console.log(1);
  return Promise.resolve(3);
}

// 给返回的期约添加一个解决处理程序
foo().then(console.log);

console.log(2);

// 1
// 2
// 3
```

异步函数的返回值期待（但实际上并不要求）一个实现 thenable 接口的对象，但常规的值也可以。如果返回的是实现 thenable 接口的对象，则这个对象可以由提供给 then()的处理程序“解包”。如果不是，则返回值就被当作已经解决的期约。下面的代码演示了这些情况：

```
// 返回一个原始值
async function foo() {
  return 'foo';
}
foo().then(console.log);
// foo

// 返回一个没有实现 thenable 接口的对象
async function bar() {
  return ['bar'];
}
bar().then(console.log);
// ['bar']

// 返回一个实现了 thenable 接口的非期约对象
async function baz() {
  const thenable = {
    then(callback) { callback('baz'); }
  };
  return thenable;
}
baz().then(console.log);
// baz
```

```
// 返回一个期约
async function qux() {
  return Promise.resolve('qux');
}
qux().then(console.log);
// qux
```

与在期约处理程序中一样，在异步函数中抛出错误会返回拒绝的期约：

```
async function foo() {
  console.log(1);
  throw 3;
}

// 给返回的期约添加一个拒绝处理程序
foo().catch(console.log);
console.log(2);

// 1
// 2
// 3
```

不过，拒绝期约的错误不会被异步函数捕获：

```
async function foo() {
  console.log(1);
  Promise.reject(3);
}

// Attach a rejected handler to the returned promise
foo().catch(console.log);
console.log(2);

// 1
// 2
// Uncaught (in promise): 3
```

2. await

因为异步函数主要针对不会马上完成的任务，所以自然需要一种暂停和恢复执行的能力。使用 await 关键字可以暂停异步函数代码的执行，等待期约解决。来看下面这个本章开始就出现过的例子：

```
let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));

p.then((x) => console.log(x)); // 3
```

使用 async/await 可以写成这样：

```
async function foo() {
  let p = new Promise((resolve, reject) => setTimeout(resolve, 1000, 3));
  console.log(await p);
}

foo();
// 3
```

注意，await 关键字会暂停执行异步函数后面的代码，让出 JavaScript 运行时的执行线程。这个行为与生成器函数中的 yield 关键字是一样的。await 关键字同样是尝试“解包”对象的值，然后将这个值传给表达式，再异步恢复异步函数的执行。

await 关键字的用法与 JavaScript 的一元操作一样。它可以单独使用，也可以在表达式中使用，如

下面的例子所示：

```
// 异步打印"foo"
async function foo() {
  console.log(await Promise.resolve('foo'));
}
foo();
// foo


// 异步打印"bar"
async function bar() {
  return await Promise.resolve('bar');
}
bar().then(console.log);
// bar

// 1000 毫秒后异步打印"baz"
async function baz() {
  await new Promise((resolve, reject) => setTimeout(resolve, 1000));
  console.log('baz');
}
baz();
// baz (1000 毫秒后)
```

await 关键字期待（但实际上并不要求）一个实现 thenable 接口的对象，但常规的值也可以。如果是实现 thenable 接口的对象，则这个对象可以由 await 来“解包”。如果不是，则这个值就被当作已经解决的期约。下面的代码演示了这些情况：

```
// 等待一个原始值
async function foo() {
  console.log(await 'foo');
}
foo();
// foo

// 等待一个没有实现 thenable 接口的对象
async function bar() {
  console.log(await ['bar']);
}
bar();
// ['bar']

// 等待一个实现了 thenable 接口的非期约对象
async function baz() {
  const thenable = {
    then(callback) { callback('baz'); }
  };
  console.log(await thenable);
}
baz();
// baz

// 等待一个期约
async function qux() {
  console.log(await Promise.resolve('qux'));
}
qux();
// qux
```

等待会抛出错误的同步操作，会返回拒绝的期约：

```
async function foo() {
  console.log(1);
  await (() => { throw 3; })();
}

// 给返回的期约添加一个拒绝处理程序
foo().catch(console.log);
console.log(2);

// 1
// 2
// 3
```

如前面的例子所示，单独的 Promise.reject()不会被异步函数捕获，而会抛出未捕获错误。不过，对拒绝的期约使用 await 则会释放（unwrap）错误值（将拒绝期约返回）：

```
async function foo() {
  console.log(1);
  await Promise.reject(3);
  console.log(4); // 这行代码不会执行
}

// 给返回的期约添加一个拒绝处理程序
foo().catch(console.log);
console.log(2);

// 1
// 2
// 3
```

3. await 的限制

await 关键字必须在异步函数中使用，不能在顶级上下文如<script>标签或模块中使用。不过，定义并立即调用异步函数是没问题的。下面两段代码实际是相同的：

```
async function foo() {
  console.log(await Promise.resolve(3));
}
foo();
// 3

// 立即调用的异步函数表达式
(async function() {
  console.log(await Promise.resolve(3));
})();
// 3
```

此外，异步函数的特质不会扩展到嵌套函数。因此，await 关键字也只能直接出现在异步函数的定义中。在同步函数内部使用 await 会抛出 SyntaxError。

下面展示了一些会出错的例子：

```
// 不允许：await 出现在了箭头函数中
function foo() {
  const syncFn = () => {
    return await Promise.resolve('foo');
  };
  console.log(syncFn());
```

```
}

// 不允许：await 出现在了同步函数声明中
function bar() {
  function syncFn() {
    return await Promise.resolve('bar');
  }
  console.log(syncFn());
}

// 不允许：await 出现在了同步函数表达式中
function baz() {
  const syncFn = function() {
    return await Promise.resolve('baz');
  };
  console.log(syncFn());
}

// 不允许：IIFE 使用同步函数表达式或箭头函数
function qux() {
  (function () { console.log(await Promise.resolve('qux')); })();
  (() => console.log(await Promise.resolve('qux')))();
}
```

11.3.2 停止和恢复执行

使用 await 关键字之后的区别其实比看上去的还要微妙一些。比如，下面的例子中按顺序调用了 3 个函数，但它们的输出结果顺序是相反的：

```
async function foo() {
  console.log(await Promise.resolve('foo'));
}

async function bar() {
  console.log(await 'bar');
}

async function baz() {
  console.log('baz');
}

foo();
bar();
baz();

// baz
// bar
// foo
```

async/await 中真正起作用的是 await。async 关键字，无论从哪方面来看，都不过是一个标识符。毕竟，异步函数如果不包含 await 关键字，其执行基本上跟普通函数没有什么区别：

```
async function foo() {
  console.log(2);
}

console.log(1);
foo();
console.log(3);
```

```
// 1
// 2
// 3
```

要完全理解 await 关键字，必须知道它并非只是等待一个值可用那么简单。JavaScript 运行时在碰到 await 关键字时，会记录在哪里暂停执行。等到 await 右边的值可用了，JavaScript 运行时会向消息队列中推送一个任务，这个任务会恢复异步函数的执行。

因此，即使 await 后面跟着一个立即可用的值，函数的其余部分也会被**异步**求值。下面的例子演示了这一点：

```
async function foo() {
  console.log(2);
  await null;
  console.log(4);
}

console.log(1);
foo();
console.log(3);

// 1
// 2
// 3
// 4
```

控制台中输出结果的顺序很好地解释了运行时的工作过程：

(1) 打印 1；

(2) 调用异步函数 foo()；

(3)（在 foo()中）打印 2；

(4)（在 foo()中）await 关键字暂停执行，为立即可用的值 null 向消息队列中添加一个任务；

(5) foo()退出；

(6) 打印 3；

(7) 同步线程的代码执行完毕；

(8) JavaScript 运行时从消息队列中取出任务，恢复异步函数执行；

(9)（在 foo()中）恢复执行，await 取得 null 值（这里并没有使用）；

(10)（在 foo()中）打印 4；

(11) foo()返回。

如果 await 后面是一个期约，则问题会稍微复杂一些。此时，为了执行异步函数，实际上会有两个任务被添加到消息队列并被异步求值。下面的例子虽然看起来很反直觉，但它演示了真正的执行顺序：[①]

```
async function foo() {
  console.log(2);
  console.log(await Promise.resolve(8));
  console.log(9);
}

async function bar() {
```

① TC39 对 await 后面是期约的情况如何处理做过一次修改。修改后，本例中的 Promise.resolve(8)只会生成一个异步任务。因此在新版浏览器中，这个示例的输出结果为 123458967。实际开发中，对于并行的异步操作我们通常更关注结果，而不依赖执行顺序。——译者注

```
  console.log(4);
  console.log(await 6);
  console.log(7);
}

console.log(1);
foo();
console.log(3);
bar();
console.log(5);

// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

运行时会像这样执行上面的例子：

(1) 打印 1；

(2) 调用异步函数 `foo()`；

(3)（在 `foo()` 中）打印 2；

(4)（在 `foo()` 中）`await` 关键字暂停执行，向消息队列中添加一个期约在落定之后执行的任务；

(5) 期约立即落定，把给 `await` 提供值的任务添加到消息队列；

(6) `foo()` 退出；

(7) 打印 3；

(8) 调用异步函数 `bar()`；

(9)（在 `bar()` 中）打印 4；

(10)（在 `bar()` 中）`await` 关键字暂停执行，为立即可用的值 6 向消息队列中添加一个任务；

(11) `bar()` 退出；

(12) 打印 5；

(13) 顶级线程执行完毕；

(14) JavaScript 运行时从消息队列中取出解决 `await` 期约的处理程序，并将解决的值 8 提供给它；

(15) JavaScript 运行时向消息队列中添加一个恢复执行 `foo()` 函数的任务；

(16) JavaScript 运行时从消息队列中取出恢复执行 `bar()` 的任务及值 6；

(17)（在 `bar()` 中）恢复执行，`await` 取得值 6；

(18)（在 `bar()` 中）打印 6；

(19)（在 `bar()` 中）打印 7；

(20) `bar()` 返回；

(21) 异步任务完成，JavaScript 从消息队列中取出恢复执行 `foo()` 的任务及值 8；

(22)（在 `foo()` 中）打印 8；

(23)（在 `foo()` 中）打印 9；

(24) `foo()` 返回。