行。无论是 then()、catch()还是 finally()添加的处理程序都是如此。

```
let p1 = Promise.resolve();
let p2 = Promise.reject();

p1.then(() => setTimeout(console.log, 0, 1));
p1.then(() => setTimeout(console.log, 0, 2));
// 1
// 2

p2.then(null, () => setTimeout(console.log, 0, 3));
p2.then(null, () => setTimeout(console.log, 0, 4));
// 3
// 4

p2.catch(() => setTimeout(console.log, 0, 5));
p2.catch(() => setTimeout(console.log, 0, 6));
// 5
// 6

p1.finally(() => setTimeout(console.log, 0, 7));
p1.finally(() => setTimeout(console.log, 0, 8));
// 7
// 8
```

7. 传递解决值和拒绝理由

到了落定状态后，期约会提供其解决值（如果兑现）或其拒绝理由（如果拒绝）给相关状态的处理程序。拿到返回值后，就可以进一步对这个值进行操作。比如，第一次网络请求返回的 JSON 是发送第二次请求必需的数据，那么第一次请求返回的值就应该传给 onResolved 处理程序继续处理。当然，失败的网络请求也应该把 HTTP 状态码传给 onRejected 处理程序。

在执行函数中，解决的值和拒绝的理由是分别作为 resolve()和 reject()的第一个参数往后传的。然后，这些值又会传给它们各自的处理程序，作为 onResolved 或 onRejected 处理程序的唯一参数。下面的例子展示了上述传递过程：

```
let p1 = new Promise((resolve, reject) => resolve('foo'));
p1.then((value) => console.log(value));    // foo

let p2 = new Promise((resolve, reject) => reject('bar'));
p2.catch((reason) => console.log(reason));  // bar
```

Promise.resolve()和 Promise.reject()在被调用时就会接收解决值和拒绝理由。同样地，它们返回的期约也会像执行器一样把这些值传给 onResolved 或 onRejected 处理程序：

```
let p1 = Promise.resolve('foo');
p1.then((value) => console.log(value));   // foo

let p2 = Promise.reject('bar');
p2.catch((reason) => console.log(reason)); // bar
```

8. 拒绝期约与拒绝错误处理

拒绝期约类似于 throw()表达式，因为它们都代表一种程序状态，即需要中断或者特殊处理。在期约的执行函数或处理程序中抛出错误会导致拒绝，对应的错误对象会成为拒绝的理由。因此以下这些期约都会以一个错误对象为由被拒绝：

```
let p1 = new Promise((resolve, reject) => reject(Error('foo')));
let p2 = new Promise((resolve, reject) => { throw Error('foo'); });
let p3 = Promise.resolve().then(() => { throw Error('foo'); });
let p4 = Promise.reject(Error('foo'));

setTimeout(console.log, 0, p1);  // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p2);  // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p3);  // Promise <rejected>: Error: foo
setTimeout(console.log, 0, p4);  // Promise <rejected>: Error: foo


// 也会抛出 4 个未捕获错误
```

期约可以以任何理由拒绝，包括 undefined，但最好统一使用错误对象。这样做主要是因为创建错误对象可以让浏览器捕获错误对象中的栈追踪信息，而这些信息对调试是非常关键的。例如，前面例子中抛出的 4 个错误的栈追踪信息如下：

```
Uncaught (in promise) Error: foo
    at Promise (test.html:5)
    at new Promise (<anonymous>)
    at test.html:5
Uncaught (in promise) Error: foo
    at Promise (test.html:6)
    at new Promise (<anonymous>)
    at test.html:6
Uncaught (in promise) Error: foo
    at test.html:8
Uncaught (in promise) Error: foo
    at Promise.resolve.then (test.html:7)
```

所有错误都是异步抛出且未处理的，通过错误对象捕获的栈追踪信息展示了错误发生的路径。注意错误的顺序：Promise.resolve().then()的错误最后才出现，这是因为它需要在运行时消息队列中**添加**处理程序；也就是说，在最终抛出未捕获错误之前它还会创建另一个期约。

这个例子同样揭示了异步错误有意思的副作用。正常情况下，在通过 throw()关键字抛出错误时，JavaScript 运行时的错误处理机制会停止执行抛出错误之后的任何指令：

```
throw Error('foo');
console.log('bar'); // 这一行不会执行

// Uncaught Error: foo
```

但是，在期约中抛出错误时，因为错误实际上是从消息队列中异步抛出的，所以并不会阻止运行时继续执行同步指令：

```
Promise.reject(Error('foo'));
console.log('bar');
// bar

// Uncaught (in promise) Error: foo
```

如本章前面的 Promise.reject()示例所示，异步错误只能通过异步的 onRejected 处理程序捕获：

```
// 正确
Promise.reject(Error('foo')).catch((e) => {});

// 不正确
```

```
try {
  Promise.reject(Error('foo'));
} catch(e) {}
```

这不包括捕获执行函数中的错误，在解决或拒绝期约之前，仍然可以使用 try/catch 在执行函数中捕获错误：

```
let p = new Promise((resolve, reject) => {
  try {
    throw Error('foo');
  } catch(e) {}

  resolve('bar');
});

setTimeout(console.log, 0, p); // Promise <resolved>: bar
```

then()和 catch()的 onRejected 处理程序在语义上相当于 try/catch。出发点都是捕获错误之后将其隔离，同时不影响正常逻辑执行。为此，onRejected 处理程序的任务应该是在捕获异步错误之后返回一个**解决**的期约。下面的例子中对比了同步错误处理与异步错误处理：

```
console.log('begin synchronous execution');
try {
  throw Error('foo');
} catch(e) {
  console.log('caught error', e);
}
console.log('continue synchronous execution');

// begin synchronous execution
// caught error Error: foo
// continue synchronous execution



new Promise((resolve, reject) => {
  console.log('begin asynchronous execution');
  reject(Error('bar'));
}).catch((e) => {
  console.log('caught error', e);
}).then(() => {
  console.log('continue asynchronous execution');
});

// begin asynchronous execution
// caught error Error: bar
// continue asynchronous execution
```

11.2.4　期约连锁与期约合成

多个期约组合在一起可以构成强大的代码逻辑。这种组合可以通过两种方式实现：期约连锁与期约合成。前者就是一个期约接一个期约地拼接，后者则是将多个期约组合为一个期约。

1. 期约连锁

把期约逐个地串联起来是一种非常有用的编程模式。之所以可以这样做，是因为每个期约实例的方法（then()、catch()和 finally()）都会返回一个**新的**期约对象，而这个新期约又有自己的实例方

法。这样连缀方法调用就可以构成所谓的“期约连锁”。比如：

```
let p = new Promise((resolve, reject) => {
  console.log('first');
  resolve();
});
p.then(() => console.log('second'))
 .then(() => console.log('third'))
 .then(() => console.log('fourth'));

// first
// second
// third
// fourth
```

这个实现最终执行了一连串**同步**任务。正因为如此，这种方式执行的任务没有那么有用，毕竟分别使用 4 个同步函数也可以做到：

```
(() => console.log('first'))();
(() => console.log('second'))();
(() => console.log('third'))();
(() => console.log('fourth'))();
```

要真正执行**异步**任务，可以改写前面的例子，让每个执行器都返回一个期约实例。这样就可以让每个后续期约都等待之前的期约，也就是串行化异步任务。比如，可以像下面这样让每个期约在一定时间后解决：

```
let p1 = new Promise((resolve, reject) => {
  console.log('p1 executor');
  setTimeout(resolve, 1000);
});

p1.then(() => new Promise((resolve, reject) => {
    console.log('p2 executor');
    setTimeout(resolve, 1000);
  }))
  .then(() => new Promise((resolve, reject) => {
    console.log('p3 executor');
    setTimeout(resolve, 1000);
  }))
  .then(() => new Promise((resolve, reject) => {
    console.log('p4 executor');
    setTimeout(resolve, 1000);
  }));

// p1 executor (1 秒后)
// p2 executor (2 秒后)
// p3 executor (3 秒后)
// p4 executor (4 秒后)
```

把生成期约的代码提取到一个工厂函数中，就可以写成这样：

```
function delayedResolve(str) {
  return new Promise((resolve, reject) => {
    console.log(str);
    setTimeout(resolve, 1000);
  });
}
```

```
delayedResolve('p1 executor')
  .then(() => delayedResolve('p2 executor'))
  .then(() => delayedResolve('p3 executor'))
  .then(() => delayedResolve('p4 executor'))

// p1 executor (1 秒后)
// p2 executor (2 秒后)
// p3 executor (3 秒后)
// p4 executor (4 秒后)
```

每个后续的处理程序都会等待前一个期约解决，然后实例化一个新期约并返回它。这种结构可以简洁地将异步任务串行化，解决之前依赖回调的难题。假如这种情况下不使用期约，那么前面的代码可能就要这样写了：

```
function delayedExecute(str, callback = null) {
  setTimeout(() => {
    console.log(str);
    callback && callback();
  }, 1000)
}

delayedExecute('p1 callback', () => {
  delayedExecute('p2 callback', () => {
    delayedExecute('p3 callback', () => {
      delayedExecute('p4 callback');
    });
  });
});

// p1 callback (1 秒后)
// p2 callback (2 秒后)
// p3 callback (3 秒后)
// p4 callback (4 秒后)
```

心明眼亮的开发者会发现，这不正是期约所要解决的回调地狱问题吗？

因为 `then()`、`catch()`和 `finally()`都返回期约，所以串联这些方法也很直观。下面的例子同时使用这 3 个实例方法：

```
let p = new Promise((resolve, reject) => {
  console.log('initial promise rejects');
  reject();
});

p.catch(() => console.log('reject handler'))
 .then(() => console.log('resolve handler'))
 .finally(() => console.log('finally handler'));

// initial promise rejects
// reject handler
// resolve handler
// finally handler
```

2. 期约图

因为一个期约可以有任意多个处理程序，所以期约连锁可以构建**有向非循环图**的结构。这样，每个期约都是图中的一个节点，而使用实例方法添加的处理程序则是有向顶点。因为图中的每个节点都会等待前一个节点落定，所以图的方向就是期约的解决或拒绝顺序。

下面的例子展示了一种期约有向图，也就是二叉树：

```
//     A
//    / \
//   B   C
//  /\  /\
//  D E F G

let A = new Promise((resolve, reject) => {
  console.log('A');
  resolve();
});

let B = A.then(() => console.log('B'));
let C = A.then(() => console.log('C'));

B.then(() => console.log('D'));
B.then(() => console.log('E'));
C.then(() => console.log('F'));
C.then(() => console.log('G'));

// A
// B
// C
// D
// E
// F
// G
```

注意，日志的输出语句是对二叉树的层序遍历。如前所述，期约的处理程序是按照它们添加的顺序执行的。由于期约的处理程序是**先**添加到消息队列，**然后**才逐个执行，因此构成了层序遍历。

树只是期约图的一种形式。考虑到根节点不一定唯一，且多个期约也可以组合成一个期约（通过下一节介绍的 `Promise.all()`和 `Promise.race()`），所以有向非循环图是体现期约连锁可能性的最准确表达。

3. Promise.all()和Promise.race()

`Promise`类提供两个将多个期约实例组合成一个期约的静态方法：`Promise.all()`和`Promise.race()`。而合成后期约的行为取决于内部期约的行为。

- **Promise.all()**

`Promise.all()`静态方法创建的期约会在一组期约全部解决之后再解决。这个静态方法接收一个可迭代对象，返回一个新期约：

```
let p1 = Promise.all([
  Promise.resolve(),
  Promise.resolve()
]);

// 可迭代对象中的元素会通过 Promise.resolve()转换为期约
let p2 = Promise.all([3, 4]);

// 空的可迭代对象等价于 Promise.resolve()
let p3 = Promise.all([]);

// 无效的语法
let p4 = Promise.all();
// TypeError: cannot read Symbol.iterator of undefined
```

合成的期约只会在每个包含的期约都解决之后才解决：

```
let p = Promise.all([
  Promise.resolve(),
  new Promise((resolve, reject) => setTimeout(resolve, 1000))
]);
setTimeout(console.log, 0, p); // Promise <pending>

p.then(() => setTimeout(console.log, 0, 'all() resolved!'));

// all() resolved! (大约 1 秒后)
```

如果至少有一个包含的期约待定，则合成的期约也会待定。如果有一个包含的期约拒绝，则合成的期约也会拒绝：

```
// 永远待定
let p1 = Promise.all([new Promise(() => {})]);
setTimeout(console.log, 0, p1); // Promise <pending>

// 一次拒绝会导致最终期约拒绝
let p2 = Promise.all([
  Promise.resolve(),
  Promise.reject(),
  Promise.resolve()
]);
setTimeout(console.log, 0, p2); // Promise <rejected>

// Uncaught (in promise) undefined
```

如果所有期约都成功解决，则合成期约的解决值就是所有包含期约解决值的数组，按照迭代器顺序：

```
let p = Promise.all([
  Promise.resolve(3),
  Promise.resolve(),
  Promise.resolve(4)
]);

p.then((values) => setTimeout(console.log, 0, values)); // [3, undefined, 4]
```

如果有期约拒绝，则第一个拒绝的期约会将自己的理由作为合成期约的拒绝理由。之后再拒绝的期约不会影响最终期约的拒绝理由。不过，这并不影响所有包含期约正常的拒绝操作。合成的期约会静默处理所有包含期约的拒绝操作，如下所示：

```
// 虽然只有第一个期约的拒绝理由会进入
// 拒绝处理程序，第二个期约的拒绝也
// 会被静默处理，不会有错误跑掉
let p = Promise.all([
  Promise.reject(3),
  new Promise((resolve, reject) => setTimeout(reject, 1000))
]);

p.catch((reason) => setTimeout(console.log, 0, reason)); // 3

// 没有未处理的错误
```

- **Promise.race()**

Promise.race()静态方法返回一个包装期约，是一组集合中最先解决或拒绝的期约的镜像。这个方法接收一个可迭代对象，返回一个新期约：

```
let p1 = Promise.race([
  Promise.resolve(),
  Promise.resolve()
]);

// 可迭代对象中的元素会通过 Promise.resolve()转换为期约
let p2 = Promise.race([3, 4]);

// 空的可迭代对象等价于 new Promise(() => {})
let p3 = Promise.race([]);

// 无效的语法
let p4 = Promise.race();
// TypeError: cannot read Symbol.iterator of undefined
```

Promise.race()不会对解决或拒绝的期约区别对待。无论是解决还是拒绝，只要是第一个落定的期约，Promise.race()就会包装其解决值或拒绝理由并返回新期约：

```
// 解决先发生，超时后的拒绝被忽略
let p1 = Promise.race([
  Promise.resolve(3),
  new Promise((resolve, reject) => setTimeout(reject, 1000))
]);
setTimeout(console.log, 0, p1); // Promise <resolved>: 3

// 拒绝先发生，超时后的解决被忽略
let p2 = Promise.race([
  Promise.reject(4),
  new Promise((resolve, reject) => setTimeout(resolve, 1000))
]);
setTimeout(console.log, 0, p2); // Promise <rejected>: 4

// 迭代顺序决定了落定顺序
let p3 = Promise.race([
  Promise.resolve(5),
  Promise.resolve(6),
  Promise.resolve(7)
]);
setTimeout(console.log, 0, p3); // Promise <resolved>: 5
```

如果有一个期约拒绝，只要它是第一个落定的，就会成为拒绝合成期约的理由。之后再拒绝的期约不会影响最终期约的拒绝理由。不过，这并不影响所有包含期约正常的拒绝操作。与 Promise.all()类似，合成的期约会静默处理所有包含期约的拒绝操作，如下所示：

```
// 虽然只有第一个期约的拒绝理由会进入
// 拒绝处理程序，第二个期约的拒绝也
// 会被静默处理，不会有错误跑掉
let p = Promise.race([
  Promise.reject(3),
  new Promise((resolve, reject) => setTimeout(reject, 1000))
]);

p.catch((reason) => setTimeout(console.log, 0, reason)); // 3

// 没有未处理的错误
```

4. 串行期约合成

到目前为止，我们讨论期约连锁一直围绕期约的串行执行，忽略了期约的另一个主要特性：异步产

生值并将其传给处理程序。基于后续期约使用之前期约的返回值来串联期约是期约的基本功能。这很像**函数合成**，即将多个函数合成为一个函数，比如：

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function addTen(x) {
  return addFive(addTwo(addThree(x)));
}

console.log(addTen(7)); // 17
```

在这个例子中，有 3 个函数基于一个值合成为一个函数。类似地，期约也可以像这样合成起来，渐进地消费一个值，并返回一个结果：

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function addTen(x) {
  return Promise.resolve(x)
    .then(addTwo)
    .then(addThree)
    .then(addFive);
}

addTen(8).then(console.log); // 18
```

使用 `Array.prototype.reduce()`可以写成更简洁的形式：

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function addTen(x) {
  return [addTwo, addThree, addFive]
      .reduce((promise, fn) => promise.then(fn), Promise.resolve(x));
}

addTen(8).then(console.log); // 18
```

这种模式可以提炼出一个通用函数，可以把任意多个函数作为处理程序合成一个连续传值的期约连锁。这个通用的合成函数可以这样实现：

```
function addTwo(x) {return x + 2;}
function addThree(x) {return x + 3;}
function addFive(x) {return x + 5;}

function compose(...fns) {
  return (x) => fns.reduce((promise, fn) => promise.then(fn), Promise.resolve(x))
}

let addTen = compose(addTwo, addThree, addFive);

addTen(8).then(console.log); // 18
```

> **注意**　本章后面的 11.3 节在讨论异步函数时还会涉及这个概念。

11.2.5 期约扩展

ES6 期约实现是很可靠的，但它也有不足之处。比如，很多第三方期约库实现中具备而 ECMAScript 规范却未涉及的两个特性：期约取消和进度追踪。

1. 期约取消

我们经常会遇到期约正在处理过程中，程序却不再需要其结果的情形。这时候如果能够取消期约就好了。某些第三方库，比如 Bluebird，就提供了这个特性。实际上，TC39 委员会也曾准备增加这个特性，但相关提案最终被撤回了。结果，ES6 期约被认为是“激进的”：只要期约的逻辑开始执行，就没有办法阻止它执行到完成。

实际上，可以在现有实现基础上提供一种临时性的封装，以实现取消期约的功能。这可以用到 Kevin Smith 提到的“取消令牌”（cancel token）。生成的令牌实例提供了一个接口，利用这个接口可以取消期约；同时也提供了一个期约的实例，可以用来触发取消后的操作并求值取消状态。

下面是 `CancelToken` 类的一个基本实例：

```
class CancelToken {
  constructor(cancelFn) {
    this.promise = new Promise((resolve, reject) => {
      cancelFn(resolve);
    });
  }
}
```

这个类包装了一个期约，把解决方法暴露给了 `cancelFn` 参数。这样，外部代码就可以向构造函数中传入一个函数，从而控制什么情况下可以取消期约。这里期约是令牌类的公共成员，因此可以给它添加处理程序以取消期约。

这个类大概可以这样使用：

```
<button id="start">Start</button>
<button id="cancel">Cancel</button>

<script>
class CancelToken {
  constructor(cancelFn) {
    this.promise = new Promise((resolve, reject) => {
      cancelFn(() => {
        setTimeout(console.log, 0, "delay cancelled");
        resolve();
      });
    });
  }
}

const startButton = document.querySelector('#start');
const cancelButton = document.querySelector('#cancel');

function cancellableDelayedResolve(delay) {
  setTimeout(console.log, 0, "set delay");

  return new Promise((resolve, reject) => {
    const id = setTimeout((() => {
      setTimeout(console.log, 0, "delayed resolve");
      resolve();
    }), delay);
```