- 待定（pending）
- 兑现（fulfilled，有时候也称为“解决”，resolved）
- 拒绝（rejected）

待定（pending）是期约的最初始状态。在待定状态下，期约可以**落定**（settled）为代表成功的**兑现**（fulfilled）状态，或者代表失败的**拒绝**（rejected）状态。无论落定为哪种状态都是不可逆的。只要从待定转换为兑现或拒绝，期约的状态就不再改变。而且，也不能保证期约必然会脱离待定状态。因此，组织合理的代码无论期约解决（resolve）还是拒绝（reject），甚至永远处于待定（pending）状态，都应该具有恰当的行为。

重要的是，期约的状态是私有的，不能直接通过 JavaScript 检测到。这主要是为了避免根据读取到的期约状态，以同步方式处理期约对象。另外，期约的状态也不能被外部 JavaScript 代码修改。这与不能读取该状态的原因是一样的：期约故意将异步行为封装起来，从而隔离外部的同步代码。

2. 解决值、拒绝理由及期约用例

期约主要有两大用途。首先是抽象地表示一个异步操作。期约的状态代表期约是否完成。“待定”表示尚未开始或者正在执行中。“兑现”表示已经成功完成，而“拒绝”则表示没有成功完成。

某些情况下，这个状态机就是期约可以提供的最有用的信息。知道一段异步代码已经完成，对于其他代码而言已经足够了。比如，假设期约要向服务器发送一个 HTTP 请求。请求返回 200~299 范围内的状态码就足以让期约的状态变为“兑现”。类似地，如果请求返回的状态码不在 200~299 这个范围内，那么就会把期约状态切换为“拒绝”。

在另外一些情况下，期约封装的异步操作会实际生成某个值，而程序期待期约状态改变时可以访问这个值。相应地，如果期约被拒绝，程序就会期待期约状态改变时可以拿到拒绝的理由。比如，假设期约向服务器发送一个 HTTP 请求并预定会返回一个 JSON。如果请求返回范围在 200~299 的状态码，则足以让期约的状态变为兑现。此时期约内部就可以收到一个 JSON 字符串。类似地，如果请求返回的状态码不在 200~299 这个范围内，那么就会把期约状态切换为拒绝。此时拒绝的理由可能是一个 `Error` 对象，包含着 HTTP 状态码及相关错误消息。

为了支持这两种用例，每个期约只要状态切换为兑现，就会有一个私有的内部**值**（value）。类似地，每个期约只要状态切换为拒绝，就会有一个私有的内部**理由**（reason）。无论是值还是理由，都是包含原始值或对象的不可修改的引用。二者都是可选的，而且默认值为 `undefined`。在期约到达某个落定状态时执行的异步代码始终会收到这个值或理由。

3. 通过执行函数控制期约状态

由于期约的状态是私有的，所以只能在内部进行操作。内部操作在期约的执行器函数中完成。执行器函数主要有两项职责：初始化期约的异步行为和控制状态的最终转换。其中，控制期约状态的转换是通过调用它的两个函数参数实现的。这两个函数参数通常都命名为 `resolve()`和 `reject()`。调用 `resolve()`会把状态切换为兑现，调用 `reject()`会把状态切换为拒绝。另外，调用 `reject()`也会抛出错误（后面会讨论这个错误）。

```
let p1 = new Promise((resolve, reject) => resolve());
setTimeout(console.log, 0, p1); // Promise <resolved>

let p2 = new Promise((resolve, reject) => reject());
setTimeout(console.log, 0, p2); // Promise <rejected>
// Uncaught error (in promise)
```

在前面的例子中，并没有什么异步操作，因为在初始化期约时，执行器函数已经改变了每个期约的状态。这里的关键在于，执行器函数是**同步**执行的。这是因为执行器函数是期约的初始化程序。通过下面的例子可以看出上面代码的执行顺序：

```
new Promise(() => setTimeout(console.log, 0, 'executor'));
setTimeout(console.log, 0, 'promise initialized');

// executor
// promise initialized
```

添加 setTimeout 可以推迟切换状态：

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

// 在 console.log 打印期约实例的时候，还不会执行超时回调（即 resolve()）
setTimeout(console.log, 0, p);  // Promise <pending>
```

无论 resolve()和 reject()中的哪个被调用，状态转换都不可撤销了。于是继续修改状态会静默失败，如下所示：

```
let p = new Promise((resolve, reject) => {
  resolve();
  reject(); // 没有效果
});

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

为避免期约卡在待定状态，可以添加一个定时退出功能。比如，可以通过 setTimeout 设置一个10秒钟后无论如何都会拒绝期约的回调：

```
let p = new Promise((resolve, reject) => {
  setTimeout(reject, 10000);  // 10 秒后调用 reject()
  // 执行函数的逻辑
});

setTimeout(console.log, 0, p);      // Promise <pending>
setTimeout(console.log, 11000, p);  // 11 秒后再检查状态

// (After 10 seconds) Uncaught error
// (After 11 seconds) Promise <rejected>
```

因为期约的状态只能改变一次，所以这里的超时拒绝逻辑中可以放心地设置让期约处于待定状态的最长时间。如果执行器中的代码在超时之前已经解决或拒绝，那么超时回调再尝试拒绝也会静默失败。

4. Promise.resolve()

期约并非一开始就必须处于待定状态，然后通过执行器函数才能转换为落定状态。通过调用 Promise.resolve()静态方法，可以实例化一个解决的期约。下面两个期约实例实际上是一样的：

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

这个解决的期约的值对应着传给 Promise.resolve()的第一个参数。使用这个静态方法，实际上可以把任何值都转换为一个期约：

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

setTimeout(console.log, 0, Promise.resolve(3));
```

```
// Promise <resolved>: 3

// 多余的参数会忽略
setTimeout(console.log, 0, Promise.resolve(4, 5, 6));
// Promise <resolved>: 4
```

对这个静态方法而言，如果传入的参数本身是一个期约，那它的行为就类似于一个空包装。因此，Promise.resolve()可以说是一个幂等方法，如下所示：

```
let p = Promise.resolve(7);

setTimeout(console.log, 0, p === Promise.resolve(p));
// true

setTimeout(console.log, 0, p === Promise.resolve(Promise.resolve(p)));
// true
```

这个幂等性会保留传入期约的状态：

```
let p = new Promise(() => {});

setTimeout(console.log, 0, p);                    // Promise <pending>
setTimeout(console.log, 0, Promise.resolve(p));  // Promise <pending>

setTimeout(console.log, 0, p === Promise.resolve(p)); // true
```

注意，这个静态方法能够包装任何非期约值，包括错误对象，并将其转换为解决的期约。因此，也可能导致不符合预期的行为：

```
let p = Promise.resolve(new Error('foo'));

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

5. Promise.reject()

与 Promise.resolve()类似，Promise.reject()会实例化一个拒绝的期约并抛出一个异步错误（这个错误不能通过 try/catch 捕获，而只能通过拒绝处理程序捕获）。下面的两个期约实例实际上是一样的：

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

这个拒绝的期约的理由就是传给 Promise.reject()的第一个参数。这个参数也会传给后续的拒绝处理程序：

```
let p = Promise.reject(3);
setTimeout(console.log, 0, p); // Promise <rejected>: 3

p.then(null, (e) => setTimeout(console.log, 0, e)); // 3
```

关键在于，Promise.reject()并没有照搬 Promise.resolve()的幂等逻辑。如果给它传一个期约对象，则这个期约会成为它返回的拒绝期约的理由：

```
setTimeout(console.log, 0, Promise.reject(Promise.resolve()));
// Promise <rejected>: Promise <resolved>
```

6. 同步/异步执行的二元性

Promise 的设计很大程度上会导致一种完全不同于 JavaScript 的计算模式。下面的例子完美地展示

了这一点，其中包含了两种模式下抛出错误的情形：

```
try {
  throw new Error('foo');
} catch(e) {
  console.log(e); // Error: foo
}

try {
  Promise.reject(new Error('bar'));
} catch(e) {
  console.log(e);
}
// Uncaught (in promise) Error: bar
```

第一个 try/catch 抛出并捕获了错误，第二个 try/catch 抛出错误却**没有**捕获到。乍一看这可能有点违反直觉，因为代码中确实是同步创建了一个拒绝的期约实例，而这个实例也抛出了包含拒绝理由的错误。这里的同步代码之所以没有捕获期约抛出的错误，是因为它没有通过**异步模式**捕获错误。从这里就可以看出期约真正的异步特性：它们是同步对象（在同步执行模式中使用），但也是**异步**执行模式的媒介。

在前面的例子中，拒绝期约的错误并没有抛到执行同步代码的线程里，而是通过浏览器异步消息队列来处理的。因此，try/catch 块并不能捕获该错误。代码一旦开始以异步模式执行，则唯一与之交互的方式就是使用异步结构——更具体地说，就是期约的方法。

11.2.3 期约的实例方法

期约实例的方法是连接外部同步代码与内部异步代码之间的桥梁。这些方法可以访问异步操作返回的数据，处理期约成功和失败的结果，连续对期约求值，或者添加只有期约进入终止状态时才会执行的代码。

1. 实现 Thenable 接口

在 ECMAScript 暴露的异步结构中，任何对象都有一个 then() 方法。这个方法被认为实现了 Thenable 接口。下面的例子展示了实现这一接口的最简单的类：

```
class MyThenable {
  then() {}
}
```

ECMAScript 的 Promise 类型实现了 Thenable 接口。这个简化的接口跟 TypeScript 或其他包中的接口或类型定义不同，它们都设定了 Thenable 接口更具体的形式。

> **注意** 本章后面再介绍异步函数时还会再谈到 Thenable 接口的用途和目的。

2. Promise.prototype.then()

Promise.prototype.then()是为期约实例添加处理程序的主要方法。这个 then()方法接收最多两个参数：onResolved 处理程序和 onRejected 处理程序。这两个参数都是可选的，如果提供的话，则会在期约分别进入“兑现”和“拒绝”状态时执行。

```
function onResolved(id) {
  setTimeout(console.log, 0, id, 'resolved');
```

```
}
function onRejected(id) {
  setTimeout(console.log, 0, id, 'rejected');
}

let p1 = new Promise((resolve, reject) => setTimeout(resolve, 3000));
let p2 = new Promise((resolve, reject) => setTimeout(reject, 3000));

p1.then(() => onResolved('p1'),
        () => onRejected('p1'));
p2.then(() => onResolved('p2'),
        () => onRejected('p2'));

// (3 秒后)
// p1 resolved
// p2 rejected
```

因为期约只能转换为最终状态一次，所以这两个操作一定是互斥的。

如前所述，两个处理程序参数都是可选的。而且，传给 then()的任何非函数类型的参数都会被静默忽略。如果想只提供 onRejected 参数，那就要在 onResolved 参数的位置上传入 undefined。这样有助于避免在内存中创建多余的对象，对期待函数参数的类型系统也是一个交代。

```
function onResolved(id) {
  setTimeout(console.log, 0, id, 'resolved');
}
function onRejected(id) {
  setTimeout(console.log, 0, id, 'rejected');
}

let p1 = new Promise((resolve, reject) => setTimeout(resolve, 3000));
let p2 = new Promise((resolve, reject) => setTimeout(reject, 3000));

// 非函数处理程序会被静默忽略，不推荐
p1.then('gobbeltygook');

// 不传 onResolved 处理程序的规范写法
p2.then(null, () => onRejected('p2'));

// p2 rejected (3 秒后)
```

Promise.prototype.then()方法返回一个新的期约实例：

```
let p1 = new Promise(() => {});
let p2 = p1.then();
setTimeout(console.log, 0, p1);          // Promise <pending>
setTimeout(console.log, 0, p2);          // Promise <pending>
setTimeout(console.log, 0, p1 === p2);   // false
```

这个新期约实例基于 onResovled 处理程序的返回值构建。换句话说，该处理程序的返回值会通过 Promise.resolve()包装来生成新期约。如果没有提供这个处理程序，则 Promise.resolve()就会包装上一个期约解决之后的值。如果没有显式的返回语句，则 Promise.resolve()会包装默认的返回值 undefined。

```
let p1 = Promise.resolve('foo');

// 若调用 then()时不传处理程序，则原样向后传
let p2 = p1.then();
```

```
setTimeout(console.log, 0, p2); // Promise <resolved>: foo

// 这些都一样
let p3 = p1.then(() => undefined);
let p4 = p1.then(() => {});
let p5 = p1.then(() => Promise.resolve());

setTimeout(console.log, 0, p3);  // Promise <resolved>: undefined
setTimeout(console.log, 0, p4);  // Promise <resolved>: undefined
setTimeout(console.log, 0, p5);  // Promise <resolved>: undefined
```

如果有显式的返回值，则 Promise.resolve()会包装这个值：

```
...

// 这些都一样
let p6 = p1.then(() => 'bar');
let p7 = p1.then(() => Promise.resolve('bar'));

setTimeout(console.log, 0, p6);  // Promise <resolved>: bar
setTimeout(console.log, 0, p7);  // Promise <resolved>: bar

// Promise.resolve()保留返回的期约
let p8 = p1.then(() => new Promise(() => {}));
let p9 = p1.then(() => Promise.reject());
// Uncaught (in promise): undefined

setTimeout(console.log, 0, p8);  // Promise <pending>
setTimeout(console.log, 0, p9);  // Promise <rejected>: undefined
```

抛出异常会返回拒绝的期约：

```
...

let p10 = p1.then(() => { throw 'baz'; });
// Uncaught (in promise) baz

setTimeout(console.log, 0, p10);  // Promise <rejected> baz
```

注意，返回错误值不会触发上面的拒绝行为，而会把错误对象包装在一个解决的期约中：

```
...

let p11 = p1.then(() => Error('qux'));

setTimeout(console.log, 0, p11); // Promise <resolved>: Error: qux
```

onRejected 处理程序也与之类似：onRejected 处理程序返回的值也会被 Promise.resolve()包装。乍一看这可能有点违反直觉，但是想一想，onRejected 处理程序的任务不就是捕获异步错误吗？因此，拒绝处理程序在捕获错误后不抛出异常是符合期约的行为，应该返回一个解决期约。

下面的代码片段展示了用 Promise.reject()替代之前例子中的 Promise.resolve()之后的结果：

```
let p1 = Promise.reject('foo');

// 调用then()时不传处理程序则原样向后传
let p2 = p1.then();
// Uncaught (in promise) foo
```

```
setTimeout(console.log, 0, p2);  // Promise <rejected>: foo

// 这些都一样
let p3 = p1.then(null, () => undefined);
let p4 = p1.then(null, () => {});
let p5 = p1.then(null, () => Promise.resolve());

setTimeout(console.log, 0, p3); // Promise <resolved>: undefined
setTimeout(console.log, 0, p4); // Promise <resolved>: undefined
setTimeout(console.log, 0, p5); // Promise <resolved>: undefined



// 这些都一样
let p6 = p1.then(null, () => 'bar');
let p7 = p1.then(null, () => Promise.resolve('bar'));

setTimeout(console.log, 0, p6); // Promise <resolved>: bar
setTimeout(console.log, 0, p7); // Promise <resolved>: bar

// Promise.resolve()保留返回的期约
let p8 = p1.then(null, () => new Promise(() => {}));
let p9 = p1.then(null, () => Promise.reject());
// Uncaught (in promise): undefined

setTimeout(console.log, 0, p8); // Promise <pending>
setTimeout(console.log, 0, p9); // Promise <rejected>: undefined



let p10 = p1.then(null, () => { throw 'baz'; });
// Uncaught (in promise) baz

setTimeout(console.log, 0, p10); // Promise <rejected>: baz



let p11 = p1.then(null, () => Error('qux'));

setTimeout(console.log, 0, p11); // Promise <resolved>: Error: qux
```

3. Promise.prototype.catch()

Promise.prototype.catch()方法用于给期约添加拒绝处理程序。这个方法只接收一个参数：onRejected处理程序。事实上，这个方法就是一个语法糖，调用它就相当于调用Promise.prototype.then(null, onRejected)。

下面的代码展示了这两种同样的情况：

```
let p = Promise.reject();
let onRejected = function(e) {
  setTimeout(console.log, 0, 'rejected');
};

// 这两种添加拒绝处理程序的方式是一样的：
p.then(null, onRejected);  // rejected
p.catch(onRejected);       // rejected
```

Promise.prototype.catch()返回一个新的期约实例：

```
let p1 = new Promise(() => {});
let p2 = p1.catch();
setTimeout(console.log, 0, p1);          // Promise <pending>
setTimeout(console.log, 0, p2);          // Promise <pending>
setTimeout(console.log, 0, p1 === p2);   // false
```

在返回新期约实例方面，Promise.prototype.catch()的行为与 Promise.prototype.then()的 onRejected 处理程序是一样的。

4. Promise.prototype.finally()

Promise.prototype.finally()方法用于给期约添加 onFinally 处理程序，这个处理程序在期约转换为解决或拒绝状态时都会执行。这个方法可以避免 onResolved 和 onRejected 处理程序中出现冗余代码。但 onFinally 处理程序没有办法知道期约的状态是解决还是拒绝，所以这个方法主要用于添加清理代码。

```
let p1 = Promise.resolve();
let p2 = Promise.reject();
let onFinally = function() {
  setTimeout(console.log, 0, 'Finally!')
}

p1.finally(onFinally); // Finally
p2.finally(onFinally); // Finally
```

Promise.prototype.finally()方法返回一个新的期约实例：

```
let p1 = new Promise(() => {});
let p2 = p1.finally();
setTimeout(console.log, 0, p1);          // Promise <pending>
setTimeout(console.log, 0, p2);          // Promise <pending>
setTimeout(console.log, 0, p1 === p2);   // false
```

这个新期约实例不同于 then()或 catch()方式返回的实例。因为 onFinally 被设计为一个状态无关的方法，所以在大多数情况下它将表现为父期约的传递。对于已解决状态和被拒绝状态都是如此。

```
let p1 = Promise.resolve('foo');

// 这里都会原样后传
let p2 = p1.finally();
let p3 = p1.finally(() => undefined);
let p4 = p1.finally(() => {});
let p5 = p1.finally(() => Promise.resolve());
let p6 = p1.finally(() => 'bar');
let p7 = p1.finally(() => Promise.resolve('bar'));
let p8 = p1.finally(() => Error('qux'));

setTimeout(console.log, 0, p2);  // Promise <resolved>: foo
setTimeout(console.log, 0, p3);  // Promise <resolved>: foo
setTimeout(console.log, 0, p4);  // Promise <resolved>: foo
setTimeout(console.log, 0, p5);  // Promise <resolved>: foo
setTimeout(console.log, 0, p6);  // Promise <resolved>: foo
setTimeout(console.log, 0, p7);  // Promise <resolved>: foo
setTimeout(console.log, 0, p8);  // Promise <resolved>: foo
```

如果返回的是一个待定的期约，或者 onFinally 处理程序抛出了错误（显式抛出或返回了一个拒绝期约），则会返回相应的期约（待定或拒绝），如下所示：

```
...

// Promise.resolve()保留返回的期约
let p9 = p1.finally(() => new Promise(() => {}));
let p10 = p1.finally(() => Promise.reject());
// Uncaught (in promise): undefined

setTimeout(console.log, 0, p9);  // Promise <pending>
setTimeout(console.log, 0, p10); // Promise <rejected>: undefined

let p11 = p1.finally(() => { throw 'baz'; });
// Uncaught (in promise) baz

setTimeout(console.log, 0, p11); // Promise <rejected>: baz
```

返回待定期约的情形并不常见，这是因为只要期约一解决，新期约仍然会原样后传初始的期约：

```
let p1 = Promise.resolve('foo');

// 忽略解决的值
let p2 = p1.finally(
  () => new Promise((resolve, reject) => setTimeout(() => resolve('bar'), 100)));

setTimeout(console.log, 0, p2); // Promise <pending>

setTimeout(() => setTimeout(console.log, 0, p2), 200);

// 200 毫秒后：
// Promise <resolved>: foo
```

5. 非重入期约方法

当期约进入落定状态时，与该状态相关的处理程序仅仅会被**排期**，而非立即执行。跟在添加这个处理程序的代码之后的同步代码一定会在处理程序之前先执行。即使期约一开始就是与附加处理程序关联的状态，执行顺序也是这样的。这个特性由 JavaScript 运行时保证，被称为“非重入”（non-reentrancy）特性。下面的例子演示了这个特性：

```
// 创建解决的期约
let p = Promise.resolve();

// 添加解决处理程序
// 直觉上，这个处理程序会等期约一解决就执行
p.then(() => console.log('onResolved handler'));

// 同步输出，证明 then()已经返回
console.log('then() returns');

// 实际的输出：
// then() returns
// onResolved handler
```

在这个例子中，在一个解决期约上调用 `then()`会把 `onResolved` 处理程序推进消息队列。但这个处理程序在当前线程上的同步代码执行完成前不会执行。因此，跟在 `then()`后面的同步代码一定先于处理程序执行。

先添加处理程序后解决期约也是一样的。如果添加处理程序后，同步代码才改变期约状态，那么处理程序仍然会基于该状态变化表现出非重入特性。下面的例子展示了即使先添加了 `onResolved` 处理程

序，再同步调用 resolve()，处理程序也不会进入同步线程执行：

```
let synchronousResolve;

// 创建一个期约并将解决函数保存在一个局部变量中
let p = new Promise((resolve) => {
  synchronousResolve = function() {
    console.log('1: invoking resolve()');
    resolve();
    console.log('2: resolve() returns');
  };
});

p.then(() => console.log('4: then() handler executes'));

synchronousResolve();
console.log('3: synchronousResolve() returns');

// 实际的输出：
// 1: invoking resolve()
// 2: resolve() returns
// 3: synchronousResolve() returns
// 4: then() handler executes
```

在这个例子中，即使期约状态变化发生在添加处理程序之后，处理程序也会等到运行的消息队列让它出列时才会执行。

非重入适用于 onResolved/onRejected 处理程序、catch()处理程序和 finally()处理程序。下面的例子演示了这些处理程序都只能异步执行：

```
let p1 = Promise.resolve();
p1.then(() => console.log('p1.then() onResolved'));
console.log('p1.then() returns');

let p2 = Promise.reject();
p2.then(null, () => console.log('p2.then() onRejected'));
console.log('p2.then() returns');

let p3 = Promise.reject();
p3.catch(() => console.log('p3.catch() onRejected'));
console.log('p3.catch() returns');

let p4 = Promise.resolve();
p4.finally(() => console.log('p4.finally() onFinally'));

console.log('p4.finally() returns');

// p1.then() returns
// p2.then() returns
// p3.catch() returns
// p4.finally() returns
// p1.then() onResolved
// p2.then() onRejected
// p3.catch() onRejected
// p4.finally() onFinally
```

6. 邻近处理程序的执行顺序

如果给期约添加了多个处理程序，当期约状态变化时，相关处理程序会按照添加它们的顺序依次执