Promise 物件代表一個即將完成、或失敗的非同步操作，以及它所產生的值。

Promise 語法：

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

參數：

依序接收 resolve、reject 這兩個參數，executor 函式在傳入參數 resolve 與 reject 後立刻被執行，通常 executor 函式會發起一些非同步操作。接著，在成功完成後執行 resolve 以完成 promise；或如果有錯誤，執行 rejects。

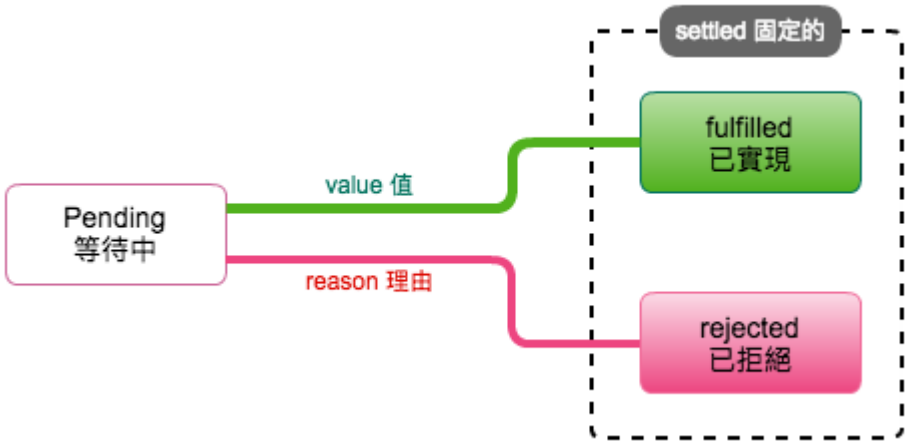
如果 executor 函式在執行中拋出錯誤，promise 會被拒絕（rejected），回傳值也將被忽略。

Promise 狀態只有三種：Pending(等待)、fulfilled(實現)、rejected(拒絕)

處在 Pending(等待)狀態時：可能轉變為 fulfilled(實現)、rejected(拒絕)狀態

處在 fulfilled(實現)狀態時：必定不會再轉變到其他任何狀態、必定有不能再更動的值(合法的 JavaScript 值，包含[undefined、thenable、promise])

處在 rejected(拒絕)狀態時：必定不會再轉變到其他任何狀態、必定有不能再更動的值(reason(理由)通常是一個 Error 物件，用於錯誤處理。)



一個 Promise 物件透過 new 及其建構式建立，第一個函式（resolve）在非同步作業成功完成時，以該作業之結果值被呼叫。第二個函式（reject）在作業失敗時，以失敗訊息，通常是一個 error object，被呼叫。

以下範例：

當非同步作業成功時，呼叫 resolve(...),而失敗時則呼叫 reject(...)，以程式碼來看在 0.5 秒會先呼叫 reject('Fail')，所以錯誤訊息就為一個 error object

```
▶Unhandled Promise rejection: Fail ; Zone: <root> ; Task: Promise.then ; Value: Fail undefined
> |
```

```
const myPromise = new Promise( (resolve, reject) => {
  setTimeout( () => {
    resolve('Success');
  }, 1000);

  setTimeout( () => {
    reject('Fail');
  }, 500);
});

// Message 是任何由上方 resolve(...) 或是reject(...)傳入的東西。
myPromise.then( ( Message ) => {
  console.log(Message);
});

// 或是這樣寫
// myPromise.then( function( value ) {
//   console.log(value);  成功時，所跑的程式碼段落
// }, function(reason){
//   console.log(reason); 失敗時，所跑的程式碼段落
// });
```

若以第二段 myPromise.then(function (value)...)承接，結果為下：

```
Fail
> 
```

then 與 catch

then 方法是 Promise 的最核心方法，有定義 then 方法的物件被稱之為 thenable 物件，語法如下

```
p.then(onFulfilled, onRejected);

p.then(function(value) {
   // fulfillment
  }, function(reason) {
  // rejection
});
```

then 方法一樣用兩個函式當作傳入參數，onFulfilled 是當 promise 物件的狀態轉為 fulfilled(實現)呼叫的函式，有一個傳入參數值可用，就是 value(值)。而 onRejected 是當 promise 物件的狀態轉為 rejected(拒絕)呼叫的函式，也有一個傳入參數值可以用，就是 reason(理由)，設計的主要目的，是要能作連鎖(chained)的語法結構，如果回傳值相同的函式，可以使用連鎖的語法。像下面這樣的程式碼：

```
const promise = new Promise(function(resolve, reject) {
  resolve(1)
})

promise.then(function(value) {
  console.log(value) // 1
  return value + 1
}).then(function(value) {
  console.log(value) // 2
  return value + 2
}).then(function(value) {
  console.log(value) // 4
})
```

onFulfilled 函式，也就是第一個函式傳入參數，它是有值時使用的函式，經過連鎖的結構，如果要把值往下傳遞，可以用回傳值的方式，例子可以看到用 return 語句來回傳值，這個值可以繼續的往下面的 then 方法傳送。

onRejected 函式，也就是 then 方法中第二個函式的傳入參數，也有用回傳值往下傳遞的特性，不過它是使用於錯誤的處理，不論是 onFulfilled 函式或 onRejected 函式的傳遞值，都只會記錄到新產生的 Promise 物件

catch：它就是要取代同步 try...catch 語句用的異步例外處理方式。

在中途發生有 promise 物件有 rejected(已拒絕)的情況，這樣會導致下一個執行被強

迫只能使用 catch 方法。

下圖範例正常執行下，所印出的值為 4、6、8，

```
const myPromise = new Promise( (resolve, reject) => {
  resolve(4);
});

// Message 是任何由上方 resolve(...) 或是reject(...)傳入的東西。
myPromise.then( ( val: any ) => {
  console.log(val); // 4
  return val + 2;
})
.then( (val) => {
    console.log(val); // 6
    return val + 2;
})
.then( (val) => {
  console.log(val); // 8
  return val + 2;
})
.catch((err) => { // catch無法抓到上個promise的回傳值
  console.log(err.message);
  // 這裡如果有回傳值，下一個then可以抓得到
  // return 100
});
```

下圖在第一個 then 丟出一個例外錯誤，會造成下一個 then 沒辦法接收到值(合法的

JavaScript 值，包含[undefined、thenable、promise])，而只能接收到理由(`reason`

通常是一個 Error 物件，用於錯誤處理。)

```
// Message 是任何由上方 resolve(...) 或是reject(...)傳入的東西。
myPromise.then( ( val: any ) => {
  console.log(val); // 4
  throw new Error('error!');
})
.then( (val) => {
    console.log(val); // 6
    return val + 2;
})
.then( (val) => {
  console.log(val); // 8
  return val + 2;
})
.catch((err) => { // catch無法抓到上個promise的回傳值
  console.log(err.message);
  // 這裡如果有回傳值，下一個then可以抓得到
  // return 100
});
```

```
4
Angular is running in the
error!
>
```