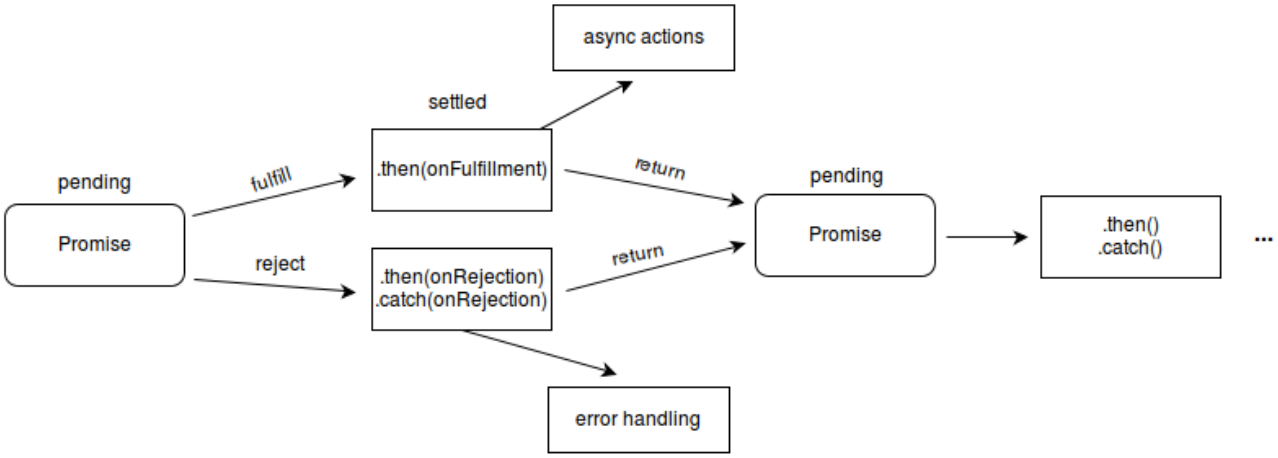
Promise




```
> console.dir(Promise)
▼ ƒ Promise()
  ▶ all: ƒ all()
    arguments: (...)
    caller: (...)
    length: 1
    name: "Promise"
  ▼ prototype: Promise
    ▶ catch: ƒ catch()
    ▶ constructor: ƒ Promise()
    ▶ finally: ƒ finally()
    ▶ then: ƒ then()
      Symbol(Symbol.toStringTag): "Promise"
    ▶ __proto__: Object
  ▶ race: ƒ race()
  ▶ reject: ƒ reject()
  ▶ resolve: ƒ resolve()
    Symbol(Symbol.species): (...)
  ▶ get Symbol(Symbol.species): ƒ [Symbol.species]()
  ▶ __proto__: ƒ ()
  ▶ [[Scopes]]: Scopes[0]
```

- Promise是一个构造函数，是异步编程的一种解决方案。
- 一个 Promise 可能有三种状态：等待（pending），已完成（fulfilled），已拒绝（rejected）。
- 一个 Promise 的状态只可能从 **pending** 转到 **fulfilled** 态或者 **rejected** 态，不能逆向转换，同时 **fulfilled** 态和 **rejected** 态不能相互转换 。
- Promise 必须实现 then 方法（可以说，then 就是 Promise 的核心），而且 then 必须返回一个 Promise，同一个 Promise 的 then 可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致 。
- then 方法接受两个参数，第一个参数是成功时的回调（resolve），在 Promise 由 **pending** 态转换到 **fulfilled** 态时调用，另一个是失败时的回调（reject），在 Promise 由 **pending** 态转换到 **rejected** 态时调用。同时，then 可以接受另一个 Promise 传入，也接受一个 “类then” 的对象或方法，即 thenable 对象。
- 按照标准来讲，resolve 函数能将 Promise 的状态置为 fullfiled，reject 函数能将 Promise 的状态置为 rejected。

我们用Promise的时候一般是包在一个函数中，在需要的时候去运行这个函数 ：

```
function runAsync(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('执行完成');
            resolve('一些数据');
        }, 2000);
    });
    return p;
}
runAsync()
```

这里有两个疑问：1.包装这么一个函数有什么用？2.resolve('随便什么数据');这是干什么的？

```
runAsync().then(function(data){
    console.log(data);//一些数据
    //后面可以用传过来的数据做些其他操作
    //......
});
```

在 runAsync() 的返回上直接调用 then 方法，then 接收一个参数，是函数，并且会拿到我们在 runAsync 中调用 resolve 时传的的参数。

运行这段代码，会在2秒后输出“执行完成”，紧接着输出 “一些数据“ 。这时候你应该有所领悟了，原来then里面的函数就跟我们平时的回调函数一个意思，能够在 runAsync 这个异步任务执行完成之后被执行。这就是Promise的作用了，简单来讲，就是能把原来的回调写法分离出来，在异步操作执行完后，用链式调用的方式执行回调函数。

```
function runAsync(callback){
    setTimeout(function(){
        console.log('执行完成');
        callback('随便什么数据');
    }, 2000);
}
runAsync(function(data){
    console.log(data);
});
```

效果也是一样的，还费劲用 Promise 干嘛。那么问题来了，有多层回调该怎么办？如果 callback 也是一个异步操作，而且执行完后也需要有相应的回调函数，该怎么办呢？总不能再定义一个 callback2，然后给callback传进去吧。而 Promise 的优势在于，可以在 then 方法中继续写 Promise 对象并返回，然后继续调用 then 来进行回调操作。

链式操作的用法

所以，从表面上看，Promise只是能够简化层层回调的写法，而实质上，Promise的精髓是 “状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用，它比传递callback函数要简单、灵活的多。所以使用Promise的正确场景是这样的：

```
runAsync1()
.then(function(data){
    console.log(data);
    return runAsync2();
})
.then(function(data){
    console.log(data);
    return runAsync3();
})
.then(function(data){
    console.log(data);
});
/*
 异步任务1执行完成
 数据1
 异步任务2执行完成
 数据2
 异步任务3执行完成
 数据3*/
```

```
function runAsync1(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('异步任务1执行完成');
            resolve('数据1');
        }, 1000);
    });
    return p;
}
function runAsync2(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('异步任务2执行完成');
            resolve('数据2');
        }, 2000);
    });
    return p;
}
function runAsync3(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            console.log('异步任务3执行完成');
            resolve('数据3');
        }, 2000);
    });
    return p;
}
```

在 then 方法中，你也可以直接 return 数据而不是 Promise 对象，在后面的 then 中就可以接收到数据了，比如我们把上面的代码修改成这样：

```
runAsync1()
.then(function(data){
    console.log(data);
    return runAsync2();
})
.then(function(data){
    console.log(data);
    return '直接返回数据';  //这里直接返回数据
})
.then(function(data){
    console.log(data);
});
/*
 异步任务1执行完成
 数据1
 异步任务2执行完成
 数据2
 直接返回数据
*/
```

reject 的用法

前面的例子都是只有“执行成功”的回调，还没有“失败”的情况，reject 的作用就是把 Promise 的状态置为 rejected，这样我们在then中就能捕捉到，然后执行“失败”情况的回调。看下面的代码。

```
function getNumber(){
    var p = new Promise(function(resolve, reject){
        //做一些异步操作
        setTimeout(function(){
            var num = Math.ceil(Math.random()*10); //生成1-10的随机数
            if(num<=5){
                resolve(num);
            }
            else{
                reject('数字太大了');
            }
        }, 2000);
    });
    return p;
}
getNumber().then(
    function(data){
        console.log('resolved');
        console.log(data);
    },
    function(data){
        console.log('rejected');
        console.log(data);
```

```
    }
);
```

catch 的用法

我们知道 Promise 对象除了 then 方法，还有一个 catch 方法，它是做什么用的呢？其实它和then的第二个参数一样，用来指定reject的回调，用法是这样：

```
getNumber()
.then(function(data){
    console.log('resolved');
    console.log(data);
})
.catch(function(reason){
    console.log('rejected');
    console.log(reason);
});
```

效果和写在 then 的第二个参数里面一样。不过它还有另外一个作用：在执行 resolve 的回调（也就是上面then中的第一个参数）时，如果抛出异常了（代码出错了），那么并不会报错卡死js，而是会进到这个 catch 方法中。

```
getNumber()
.then(function(data){
    console.log('resolved');
    console.log(data);
    console.log(somedata); //此处的somedata未定义
})
.catch(function(reason){
    console.log('rejected');
    console.log(reason);
});
```

在 resolve 的回调中，我们 console.log(somedata); 而 somedata 这个变量是没有被定义的。如果我们不用 Promise，代码运行到这里就直接在控制台报错了，不往下运行了。 但是用 catch 的话 ，结果如下，也就是说进到 catch 方法里面去了，而且把错误原因传到了 reason 参数中。即便是有错误的代码也不会报错了，这与我们的 try/catch 语句有相同的功能。

```
resolved
1
rejected
ReferenceError: somedata is not defined
    at <anonymous>:5:17
    at <anonymous>
```

all的用法

Promise的all方法提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。我们仍旧使用上面定义好的runAsync1、runAsync2、runAsync3这三个函数，看下面的例子：

```
Promise
.all([runAsync1(), runAsync2(), runAsync3()])
.then(function(results){
    console.log(results);
});
```

用 Promise.all 来执行，all 接收一个数组参数，里面的值最终都算返回 Promise 对象。这样，三个异步操作的并行执行的，等到它们都执行完后才会进到then里面。那么，三个异步操作返回的数据哪里去了呢？都在 then 里面呢，all 会把所有异步操作的结果放进一个数组中传给 then，就是上面的results。所以上面代码的输出结果就是：

```
异步任务1执行完成
异步任务2执行完成
异步任务3执行完成
(3) ["数据1", "数据2", "数据3"]
```

race的用法

all 方法的效果实际上是「谁跑的慢，以谁为准执行回调」，那么相对的就有另一个方法「谁跑的快，以谁为准执行回调」，这就是 race 方法，这个词本来就是赛跑的意思。race 的用法与all一样，我们把上面 runAsync1 的延时改为1秒来看一下：

```
Promise
.race([runAsync1(), runAsync2(), runAsync3()])
.then(function(results){
    console.log(results);
});
```

在 then 里面的回调开始执行时，runAsync2() 和 runAsync3() 并没有停止，仍旧再执行。

Promise.reject和Promise.resolve

MDN的解释如下：

- `Promise.reject(reason)`

  返回一个状态为失败的 Promise 对象，并将给定的失败信息传递给对应的处理方法

- `Promise.resolve(value)`

  返回一个状态由给定 value 决定的 Promise 对象。如果该值是一个 Promise 对象，则直接返回该对象；如果该值是 thenable (即，带有then方法的对象)，返回的 Promise 对象的最终状态由 then方法执行决定；否则的话(该 value 为空，基本类型或者不带 then 方法的对象),返回的 Promise 对象状态为 fulfilled，并且将该 value 传递给对应的 then 方法。通常而言，如果你不知道一个值是否是Promise 对象，使用 Promise.resolve(value) 来返回一个Promise对象,这样就能将该 value 以Promise对象形式使用。