TypeScript promise、async、await

目标

1. 熟悉TypeScript中的Promise对象
2. 掌握async/await的使用方法
3. 熟悉异步编程

大纲

# Promise

所谓 Promise，就是一个对象，Promise 对象是ECMAScript2015所提供的原生支持对象。

其目的是用来传递异步操作的消息。它代表了某个未来才会知道结果的事件（通常是一个异步操作），并且这个事件提供统一的接口API，可供进一步处理。

## 概述

Promise对象的出现，主要是为了解决“异步任务”中，一层又一层的回调函数嵌套问题，一旦需求发生改变，需要作出调整时，后果是非常可怕的，因为它们本身就是环环相扣，难以下手，这个现象也常被称为“回调地狱”。

比如，动画通常都是异步完成，当需要在A动画完成后，继续B动画；在B动画完成后再继续C动画……那么代码类似于：

*animateA*(  
 () => *animateB*(  
 () => *animateC*(  
 () =>{  
 // 全部完成  
 }  
 )  
 )  
);

当改用Promise对象后就可以让异步调用非常的干净、直观，增加可读性和可维护性。

*animateA*()  
.then( () => *animateB*() )  
.then( () => *animateC*() )  
.then( () =>{  
 // 全部完成  
} );

## 示例

**let *count***:**number** = 0;  
  
**function** *delay*( ms:**number** ):**Promise**<**void**>  
{  
 **return new *Promise***<**void**>( ( resolve, reject ) =>  
 {  
 *setTimeout*( ***count*** < 1 ? resolve : reject, ms, `done(${ms}ms): ${++***count***}th.` );  
 } );  
}  
  
*delay*( 1000 )  
.then( ( value ) =>  
{  
 ***console***.log( value );  
 **return** *delay*( 1000 );  
} )  
.then( ( value ) =>  
{  
 ***console***.log( value );  
 //throw new Error( "count 不能超过 3。" );  
 **return** *delay*( 1000 );  
} )  
.then( ( value ) =>  
{  
 ***console***.log( value + "**\n**all done..." );  
} )  
.catch( ( err ) =>  
{  
 ***console***.log( err );  
} );

## Promise对象状态

Promise对象在设计上具有原子性（要么完成，要么什么都不做），即只有三种状态：

* 等待（Pending）
* 成功（Fulfilled）
* 失败（Rejected）

## Promise对象的特点

### 1）状态不受外界影响

Promise对象代表一个异步操作，只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

### 2）状态确定后无法改变

Promise对象的状态改变，只有两种可能：从Pending变为Resolved和从Pending变为Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。如果改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。

这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

## 创建Promise对象

Promise对象是一个构造函数，用来生成Promise实例。

语法如下：

new Promise((resolve, reject)=>statements):Promise

示例代码：

// JavaScript：  
**let *promise*** = **new** Promise( **function**( resolve, reject )  
{  
 // ... some code  
  
 /\* 异步操作成功 \*/  
 **if**( ***a*** )  
 {  
 resolve( value );  
 }  
 **else** {  
 reject( error );  
 }  
} );

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由JavaScript引擎提供，不用自己部署。

resolve函数作用：将Promise对象状态从“未完成”变为“成功”（即从Pending变为Resolved）。

reject函数作用：将Promise对象状态从“未完成”变为“失败”（即从Pending变为Rejected）。

## 处理Promise状态

与resolve(value)和reject(reason)方法对应，Promise对象有两个处理状态变化的方法：

|  |  |
| --- | --- |
| **方 法** | **说 明** |
| promise.then(onFulfilled[,onRejected]) | 处理Promise对象成功状态并接收返回值，也可以处理失败状态（可选） |
| promise.catch(onRejected) | 处理Promise对象的失败状态并处理失败的原因 |

以上方法都会返回一个新的Promise对象，将返回的Promise对象进行组合便可以形成一个Promise对象链，呈现流水线模式的作业，而不是嵌套的回调函数。

***promise***.then( **function**( value )  
{  
 // success  
}, **function**( error )  
{  
 // failure  
} );

注意：Promise函数内若throw错误，则会自动进入rejected状态

## 使用Promise来异步加载图片

我们可以将图片的加载写成一个Promise，一旦加载完成，Promise的状态就发生变化。

|  |
| --- |
| **const** *loadImage* = **function**( path ){  **return new *Promise***( **function**( resolve, reject ){  **let** image = **new *Image***();  image.**onload** = **function**(){  resolve( image );  };  image.**onerror** = reject;  image.**src** = path;  } ); };  *loadImage*( "https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1511243529562&di=84aad70510a9ceb631818b5a69e2a582&imgtype=0&src=http%3A%2F%2Fimg15.3lian.com%2F2015%2Ff1%2F90%2Fd%2F107.jpg" ) .then( ( image ) =>{  ***console***.info( image );  **let** img:**HTMLElement** = <**HTMLElement**>image;  ***document***.**body**.appendChild( img ); } ); |

## 多个Promise对象同时启动

Promise 对象提供了两个静态类型的工具方法：Promise.all()和Promise.race()。

### Promise.all( iterable )

该方法接受一个数组对象，并最终返回一个Promise对象，该Promise对象会在所有传入的Promise完成（包括成功和失败）后被激活。

结果Promise的状态决定：

* 若所有Promise对象都进入成功状态，则结果Promise对象也进入成功状态，并返回包含每个Promise对象返回值的结果数组。
* 若某个Promise对象进入失败状态，那么结果Promise也进入失败状态，并以那个进入失败状态的Promise的错误信息作为结果Promise的错误信息。

***Promise***.all( [ *delay*( 500 ), *delay*( 1000 ), *delay*( 1500 ) ] )  
.then( **function**( results )  
{  
 ***console***.log( results );  
} );

例如：一个100MB的zip压缩包被拆分为5个20MB的zip，可以同时发起对5个小压缩包的请求，全部完成后再对文件进行拼接和使用。

### Promise.race( iterable )

同上类似，该方法同样接受一个Promise集合，但不同的是该方法会监听所有Promise对象，并等待其中第一个进入完成状态的Promise对象，一旦完成则结果Promise对象也立即进入完成状态（完成指的是成功或失败之一）。

例如：同一个图片，存在多个来源，为了最快显示图片，从而执行多个图片请求。

警告：不论是Promise.all()，还是Promise.race()，若在数组的Promise对象上增加then或者catch，相对应的会截获整个结果Promise对象所获得的结果。

## 调整TypeScript 2.x编译环境

### tsconfig.json编译选项调整

默认情况下，不配置tsconfig.json的话，会在使用Promise对象时以红色错误显示，但不影响编译结果和正常运行（因为Promise原本就是JavaScript支持的原生对象），要避免这个不必要的错误提示，在根目录增加tsconfig.json编译配置文件，并指定TypeScript编译时使用：

创建tsconfig.json：

tsc –init

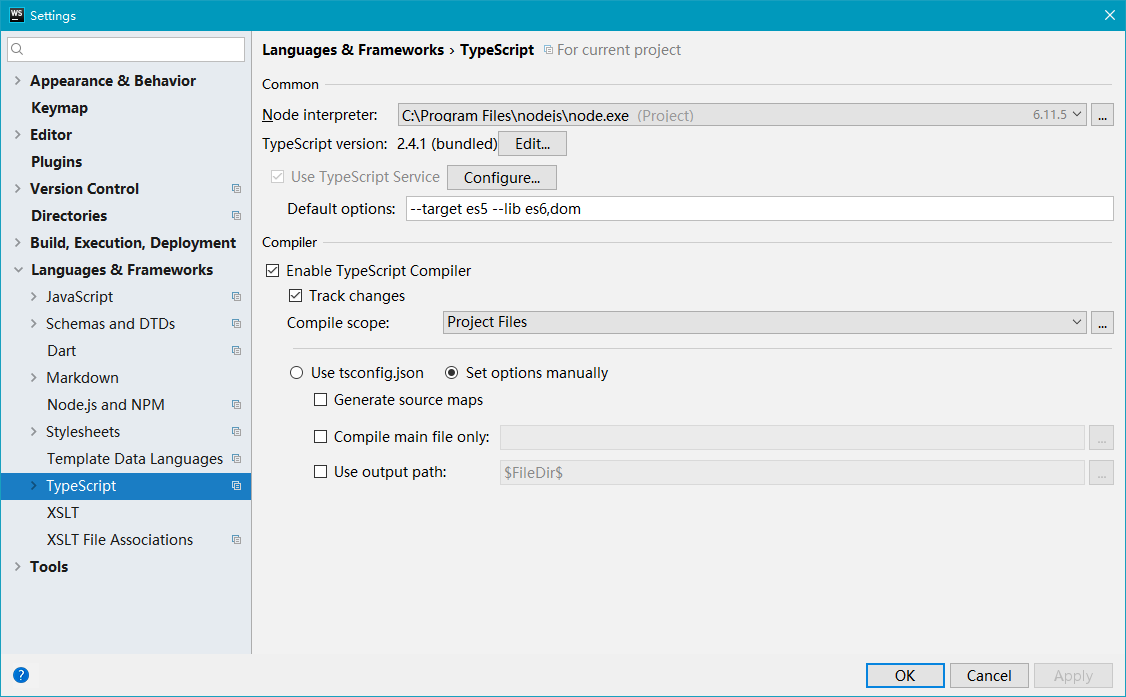
调整配置项目：

{  
 **"compilerOptions"**: {  
 **"target"**: "es5",  
 **"module"**: "commonjs",  
 **"lib"**: [ "es2015", "dom" ],  
 **"strict"**: **true** }  
}

关键就在于调整“lib”显示支持"es2015", "dom"是为了支持DOM相关的成员。

### WebStorm编译选项调整

因TypeScript默认情况下不能编译ES2015下的promise成员，因而要对其进行相应调整，在编译时携带相对应的库，WebStorm的TypeScript编译选项调整如下：



——————下午内容——————

# async/await

async用来标识一个函数作为异步处理的“特殊函数作用域”，该作用域内可以对Promise对象的调用加上await前缀，这样语句就会等待Promise返回结果为止，再继承执行下面的代码。

如此，将异步代码逻辑，以同步代码的书写方式进行运用，其可读性比单纯使用Promise对象来进行异步处理更简洁，更直观。

## async基本用法

**function** *getNumber*( ms:**number** ):**Promise**<**number**>  
{  
 **return new *Promise***<**number**>( ( resolve, reject ) =>  
 {  
 *setTimeout*( resolve, ms, ***Math***.random() \* 100 >> 0 );  
 } );  
}  
  
(**async function**()  
{  
 **let** t:**number** = **new *Date***().getTime();  
 **let** a:**number** = **await** *getNumber*( 500 );  
 **let** b:**number** = **await** *getNumber*( 500 );  
  
 **let** useTime:**number** = (**new *Date***().getTime()) - t;  
 ***console***.log( `用时 ${useTime}ms：`, a + b );  
})();

async函数的返回值仍然是 Promise 对象，你可以用then方法指定下一步的操作。

进一步说，async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖。

## await命令

await命令后一般为指定的某个Promise对象操作。

### 自动转换非Promise值

但仍然允许在await之后跟上任意值，会被转成一个立即resolve的 Promise 对象。

**async function** *func*()  
{  
 **return await** 123;  
}  
  
*func*()  
.then( v => ***console***.log( v ) );  
// 123

上面代码中，await命令的参数是数值123，它被转成 Promise 对象，并立即resolve。

### 其中reject，async也中断

如果await后面的异步操作出错，那么等同于async函数返回的 Promise 对象被reject。

只要一个await语句后面的 Promise 变为reject，那么整个async函数都会中断执行。

**async function** *func*()  
{  
 **await *Promise***.reject( '出错了' );  
 **await *Promise***.resolve( 'hello world' ); // 不会执行  
}  
*func*()  
.catch( e => ***console***.log( e ) );

### 阻止中断async

同上，若预计有可能出现reject来中断async的情况，可以catch可能发生的失败，以使后面的await能继续运行。

**async function** *func*()  
{  
 **await *Promise***.reject( '出错了' )  
 .catch( e => ***console***.log( e ) );  
 **return await *Promise***.resolve( 'hello world' );  
}  
  
*func*()  
.then( v => ***console***.log( v ) );  
// 出错了  
// hello world

### 减少不必要的等待时间

以下代码中，两个getNumber()之间并没有任何依赖关系，完全可以同时进行，以减少等待时间：

**function** *getNumber*( ms:**number** ):**Promise**<**number**>  
{  
 **return new *Promise***<**number**>( ( resolve, reject ) =>  
 {  
 *setTimeout*( resolve, ms, ***Math***.random() \* 100 >> 0 );  
 } );  
}  
  
(**async function**()  
{  
 **let** a:**number** = **await** *getNumber*( 500 );  
 **let** b:**number** = **await** *getNumber*( 500 );  
  
 ***console***.log( a + b );  
})();

修改如下：

(**async function**()  
{  
 // let a:number = await getNumber( 500 );  
 // let b:number = await getNumber( 500 );  
 **let** [ a, b ] = **await *Promise***.all( [ *getNumber*( 500 ), *getNumber*( 500 ) ] );  
  
 ***console***.log( a + b );  
})();

警告：await不能用在普通函数中，必须用在async标记的函数内。

# 练习题

## 同时加载多个图片

同时创建多个Promise对象来请求多个图片，各自的完成状态中可以立即显示图片，并且通过Promise.all对全部完成的状态进行管理。

## 异步实现“龟兔赛跑”

要求：

1. 设计两个类“龟、兔”，具备移动速度属性speed:number以及移动方法move()
2. 乌龟按设定移动速度匀速move()
3. 兔子按设定移动速度取0~speed随机数来决定移动范围
4. 各自有独立的移动距离统计moved:number和目标距离distance:number
5. 使用Promise.race()来观察谁先移动到指定目标距离

**龟**：移动速度0.2米/s，频率100ms，目标距离1000米

**兔**：移动速度1米/s，频率50ms，目标距离1000米

使用以下类来进行继承实现“龟、兔”两个类的不同移动逻辑：

|  |
| --- |
| **abstract class** Animal {  **public speed**:**number**;  **public distance**:**number**;   **protected \_moved**:**number**;// 只读属性  **get** moved():**number**{**return this**.**\_moved**;}   **abstract** move(); } |