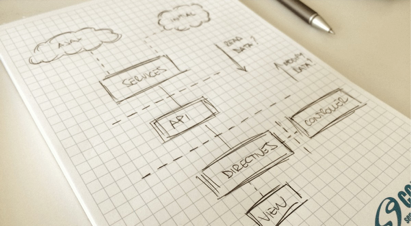
构建一个应用程序总是会面对异步调用，不论是在 Web 前端界面，还是 Node.js 服务端都是如此，JavaScript 里面处理异步调用一直是非常恶心的一件事情。以前只能通过回调函数，后来渐渐又演化出来很多方案，最后 Promise 以简单、易用、兼容性好取胜，但是仍然有非常多的问题。其实 JavaScript 一直想在语言层面彻底解决这个问题，在 ES6 中就已经支持原生的 Promise，还引入了 Generator 函数，终于在 ES7 中决定支持 async 和 await。



## **基本语法**

async/await 究竟是怎么解决异步调用的写法呢？简单来说，就是将异步操作用同步的写法来写。先来看下最基本的语法（ES7 代码片段）：

**const** f = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => {

resolve(123);

}, 2000);

});

};

**const** testAsync = **async** () => {

**const** t = **await** f();

console.log(t);

};

testAsync();

首先定义了一个函数 **f** ，这个函数返回一个 Promise，并且会延时 2 秒， **resolve** 并且传入值 123。 **testAsync** 函数在定义时使用了关键字 **async** ，然后函数体中配合使用了 **await** ，最后执行 **testAsync** 。整个程序会在 2 秒后输出 123，也就是说 **testAsync** 中常量 **t** 取得了**f** 中 **resolve** 的值，并且通过 **await** 阻塞了后面代码的执行，直到 **f** 这个异步函数执行完。

## **对比 Promise**

仅仅是一个简单的调用，就已经能够看出来 async/await 的强大，写码时可以非常优雅地处理异步函数，彻底告别回调恶梦和无数的 **then** 方法。我们再来看下与 Promise 的对比，同样的代码，如果完全使用 Promise 会有什么问题呢？

const f = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => {

resolve(123);

}, 2000);

});

};

const testAsync = () => {

f().**then**((t) => {

console.log(t);

});

};

testAsync();

从代码片段中不难看出 Promise 没有解决好的事情，比如要有很多的 **then** 方法，整块代码会充满 Promise 的方法，而不是业务逻辑本身，而且每一个 **then** 方法内部是一个独立的作用域，要是想共享数据，就要将部分数据暴露在最外层，在 **then** 内部赋值一次。虽然如此，Promise 对于异步操作的封装还是非常不错的，所以 **async/await** 是基于 Promise 的， **await** 后面是要接收一个 Promise 实例。

## **对比 RxJS**

RxJS 也是非常有意思的东西，用来处理异步操作，它更能处理基于流的数据操作。举个例子，比如在 Angular2 中 http 请求返回的就是一个 RxJS 构造的 Observable Object，我们就可以这样做：

$http.get(url)

.map(**function**(value) {

**return** value + 1;

})

.filter(**function**(value) {

**return** value !== null;

})

.forEach(**function**(value) {

console.log(value);

})

.subscribe(**function**(value) {

console.log('do something.');

}, **function**(err) {

console.log(err);

});

如果是 ES6 代码可以进一步简洁：

$http.get(url)

.map(value => value + 1)

.filter(value => value !== null)

.forEach(value => console.log(value))

.subscribe((value) => {

console.log('do something.');

}, (err) => {

console.log(err);

});

可以看出 RxJS 对于这类数据可以做一种类似流式的处理，也是非常优雅，而且 RxJS 强大之处在于你还可以对数据做取消、监听、节流等等的操作，这里不一一举例了，感兴趣的话可以去看下 RxJS 的 API。

这里要说明一下的就是 RxJS 和 async/await 一起用也是可以的，Observable Object 中有 **toPromise** 方法，可以返回一个 Promise Object，同样可以结合 **await** 使用。当然你也可以只使用 async/await 配合 underscore 或者其他库，也能实现很优雅的效果。总之，RxJS 与 async/await 不冲突。

## **异常处理**

通过使用 async/await，我们就可以配合 try/catch 来捕获异步操作过程中的问题，包括 Promise 中 reject 的数据。

**const** f = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => {

reject(234);

}, 2000);

});

};

**const** testAsync = **async** () => {

**try** {

**const** t = **await** f();

console.log(t);

} **catch** (err) {

console.log(err);

}

};

testAsync();

代码片段中将 **f** 方法中的 **resolve** 改为 **reject** ，在 **testAsync** 中，通过 **catch** 可以捕获到 **reject** 的数据，输出 err 的值为 234。 **try/catch** 使用时也要注意范围和层级。如果**try** 范围内包含多个 **await** ，那么 **catch** 会返回第一个 **reject** 的值或错误。

**const** f1 = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => {

reject(111);

}, 2000);

});

};

**const** f2 = () => {

**return** **new** Promise((resolve, reject) => {

setTimeout(() => {

reject(222);

}, 3000);

});

};

**const** testAsync = **async** () => {

**try** {

**const** t1 = **await** f1();

console.log(t1);

**const** t2 = **await** f2();

console.log(t2);

} **catch** (err) {

console.log(err);

}

};

testAsync();

如代码片段所示， **testAsync** 函数体中 **try** 有两个 **await** 函数，而且都分别 **reject** ，那么 **catch** 中仅会触发 **f1** 的 **reject** ，输出的 err 值是 111。

## **开始使用**

无论是 Web 前端还是 Node.js 服务端，都可以通过预编译的手段实现使用 ES6 和 ES7 来写代码，目前最流行的方案是通过 ***[Babel](http://babeljs.io/" \t "http://www.tuicool.com/articles/_blank)*** 将使用 ES7、ES6 写的代码编译为 E6 或 ES5 的代码来执行。

### **Node.js 服务端配置**

服务端使用 Babel，最简单的方式是通过 **require** hook。

首先安装 Babel：

$ npm **install** babel-core --save

安装 async/await 支持：

$ npm **install** babel-preset-stage-3 --save

在服务端代码的根目录中配置 .babelrc 文件，内容为：

{

"presets": ["stage-3"]

}

在顶层代码文件（server.js 或 app.js 等）中引入 Babel 模块：

require("babel-core/register");

在这句后面引入的模块，都将会自动通过 babel 编译，但当前文件不会被 babel 编译。另外，需要注意 Node.js 的版本，如果是 4.0 以上的版本则默认支持绝大部分 ES6，可以直接启动。但是如果是 0.12 左右的版本，就需要通过 **node -harmony** 来启动才能够支持。因为 stage-3 模式，Babel 不会编译基本的 ES6 代码，环境既然支持又何必要编译为 ES5？这样做也是为了提高性能和编译效率。

### **配置 Web 前端构建**

可以通过增加 Gulp 的预编译 task 来支持。

首先安装 gulp-babel 插件：

$ npm **install** gulp-babel --save-dev

然后编写配置：

**var** gulp = **require**('gulp');**var** babel = **require**('gulp-babel');

gulp.task('babel', **function**() {

**return** gulp.src('src/app.js')

.pipe(babel())

.pipe(gulp.dest('dist'));

});

除了 Gulp-babel 插件，也可以使用官方的 Babel-loader 结合 Webpack 或 Browserify 使用。

要注意的是，虽然官方也有纯浏览器版本的 Babel.js，但是浏览器限制非常多，而且对客户端性能影响也较大，不推荐使用。

## **LeanEngine Full Stack**

LeanEngine（云引擎）是LeanCloud 推出的服务器端运行环境，支持 Node.js 和 Python 环境，功能强大而且目前免费，结合 LeanCloud JavaScript SDK，使原本复杂的开发工作变得简单高效。目前也支持 Redis 和海外节点，轻松满足你的业务需求。

