javascript异步编程的前世今生，从onclick到await/async

javascript与异步编程

为了避免资源管理等复杂性的问题，
javascript被设计为单线程的语言，即使有了html5 worker，也不能直接访问dom.
javascript 设计之初是为浏览器设计的GUI编程语言，GUI编程的特性之一是保证UI线程一定不能阻塞，否则体验不佳，甚至界面卡死。
一般安卓开发，会有一个界面线程，一个后台线程，保证界面的流畅。
由于javascript是单线程，所以采用异步非阻塞的编程模式，javascript的绝大多数api都是异步api.
本文是本人的一个总结：从Brendan Eich刚设计的初版javascript到现在的ES7,一步步总结javascript异步编程历史。
说明：
本文所有源码请访问：
https://github.com/etoah/note/tree/master/async
请安装babel环境，用babel-node 执行对应例子

什么是异步编程

那么什么是异步编程，异步编程简单来说就是：执行一个指令不会马上返回结果而执行下一个任务，而是等到特定的事件触发后，才能得到结果。
以下是当有ABC三个任务，同步或异步执行的流程图：
示意图来自stackoverflow

同步

```
thread ->|----A-----||-----B-----------||-------C-------|
```

异步

```
        A-Start ------------------------------------ A-End
            | B-Start ----------------------------|--- B-End
            |   |     C-Start --------------------
```

```
C-End       |       |
            V    V         V
V           V       V
  thread-> |-A-|---B---|-C-|-A-|-C-|--A--|-B-|--C--
|---A-----|--B--|
```

显然，在宏观上，同步程序是串行的执行各任务，执行单个任务时会阻塞纯线程，异步可以"并行"的执行任务。
异步编程时就需要指定异步任务完成后需要执行的指令，总的来说有以下几种"指定异步指令"的方式：

1. 属性
2. 回调
3. Promise
4. Generator
5. await,async

下面会一步一步展现各种方式。
属性
每个编程语言对异步实现的方式不一样，C#可以用委托，java可以用接口或基类传入的方式，
早期的javascript的异步的实现也类似于这种类的属性的方式：每个类实例的相关回调事件有相应的handler(onclick,onchange,onload等)。
在DOM0级事件处理程序，就是将一个函数赋值给一个元素的属性。

```
element.onclick=function(){
    alert("clicked");
}
window.onload=function(){
    alert("loaded");
}
```

问题
这种写法简单明了，同时会有以下几个问题

- 耦合度高

所有的事件处理都需要写的一个函数中：

```
window.onload=function(){
    handlerA();
    handlerB();
    handlerc();
}
```

如果这三个handler来自三个不同的模块，那这个文件模块耦合度就为3(华为的计算方法)。依赖高，不利于复用和维护。

- 不安全，容易被重写

```
window.onload=function(){
    console.log("handler 1");
}
//...  很多其它框架，库，主题 的代码
var handlerbak=window.onload
window.onload=function(){
    handlerbak();  //这行注释的话上面handler 1就会被覆盖。
    console.log("handler 2");
}
```

当代码量大时，这种问题没有warning也没有error, 经验不丰富的前端可能花费大量的时间查找问题。

事件handler容易被重写，库/框架的安全，寄托于使用者的对框架的熟练程度，极不安全。

回调（发布/订阅）

由于javascript支持函数式编程，JavaScript语言对异步编程的实现可以用回调函数。

DOM2级事件解决了这个问题以上两个问题

```
element.addEventListener("click",function(){
    alert("clicked");
})
```

这里实际上是一个发布订阅模式，addEventListener相当于subscribe,
dispatchEvent相当于publish,
很好的解决了订阅者之前的依赖，jquery,vue,flux,angularjs均实现了类似的模式。
发布订阅模式虽解决了上面耦合和不安全的问题，但是在实现大型应用时，还会有以下问题。
问题

- 回调黑洞

多层回调嵌套，代码可读性差。

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});
```

- 异常无法捕捉

```
try{
 setTimeout(function(){
    JSON.parse("{'a':'1'}")
    console.log("aaaa")
 },0)
}
catch(ex){
 console.log(ex); //不能catch到这个异常
}
```

- 流程控制（异步代码，同步执行）

当C操作依赖于B操作和C操作,而B与A没有依赖关系时，不用第三方库（如async,eventproxy）的话,B与A本可以并行，却串行了，性能有很大的提升空间。
流程图如下：

```
graph LR
Start-->A
A-->B
B-->C
```

但用promise后，可以方便的用并行：
Promise:

```
graph LR
Start-->A
Start-->B
A-->C
B-->C
```

Promise(ECMAScript5)
如上流程图，Promise很好的解决了"并行"的问题，我们看看用promise库怎么发送get请求：

```
import fetch from 'node-fetch'
fetch('https://api.github.com/users/etoah')
.then((res)=>res.json())
.then((json)=>console.log("json:",json))
```

可以看到promise把原来嵌套的回调，改为级连的方式了，实际是一种代理(proxy)。
新建一个promise实例：

```
var promise = new Promise(function(resolve, reject)
{
  // 异步操作的代码
  if (/* 异步操作成功 */){
    resolve(value);
```

```
  } else {
    reject(error);
  }
});
```

promise把成功和失败分别代理到resolved 和 rejected .
同时还可以级连catch异常。
到这里异步的问题，有了一个比较优雅的解决方案了，如果要吹毛求疵，还有一些别扭的地方,需要改进。

问题

封装，理解相对回调复杂，这是以下我公司项目的一段代码(coffeescript),并发代码，加上resolved,rejected的回调,
即使是用了coffee，混个业务和参数处理，第一眼看上去还是比较懵，代码可读性并没有想象中的好。

```
        #并发请求companyLevel
        companyInfoP =
companyinfoServicesP.companyLevel({hid:
req.session.hid})
        requestP(userOption).success((userInfo)->
            roleOption =
                uri: "#
{config.server_host}/services/rights/userroles?
userid=#{user.userId}"
                method: 'GET'
            #保证companyInfo 写入
            Q.all([companyInfoP,
requestP(roleOption)]).spread(
                (companyinfo, roles)->
                    Util.session.init req, user,
roles.payload
                    Util.session.set(req,
"companyInfo", companyinfo.payload)
```

```
                    Util.session.set(req,
"roleids", roles.payload)
                    u = Util.session.getInfo req
                    return next {data:
Util.message.success(u)}
            , (err)->
                return next err
            )
```

在指明resolved 和 rejected的时，用的还是最原始的回调的方式。
能不能用同步的方式写异步代码？
在ES5前是这基本不可实现，但是，ES6的语法引入了Generator, yeild的关键字可以用同步的语法写异步的程序。

Generator(ECMAScript6)

简单来说generators可以理解为一个可遍历的状态机。
语法上generator,有两个特征：

1. function 关键字与函数名之前有一个星号。
2. 函数体内部使用yield关键字，定义不同的内部状态。

由于generator是一个状态机，所以需要手动调用next 才能执行，但TJ大神开发了co模块，可以自动执行generator。

```
import co from 'co';
co(function* (){
    var now = Date.now();
    yield sleep(150);    //约等待150ms
    console.log(Date.now() - now);
});

function sleep(ms){
    return function(cb){
        setTimeout(cb, ms);
    };
```

```
}
import fetch from 'node-fetch'

co(function* (){
    let result= yield  [
         (yield
fetch('https://api.github.com/users/tj')).json(),
         (yield
fetch('https://api.github.com/users/etoah')).json(),

        ];
    console.log("result:",result)
});
```

无论是延迟执行，还是并发的从两个接口获取数据，generator都可以用同步的方式编写异步代码。

注意：co模块约定，yield命令后面只能是Thunk函数或Promise对象

问题

- 需要手动执行

即使用了TJ的CO模块，不是标准的写法，感觉用hack解决问题

- 不够直观，没有语义化。

await,async(ECMAScript7)

ES7 引入了像C#语言中的 await,async关键字,而且babel已支持（引入plugins:transform-async-to-generator ）

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。

```
import fetch from 'node-fetch';
(async function (){
    let result= await
fetch('https://api.github.com/users/etoah');
```

```
    let json =await result.json();
    console.log("result:",json);
})();

//exception
(async function (){
    try{
        let result= await
fetch('https://api.3github.com/users/etoah');
        let json =await result.json();
        console.log("result:",json);
    }
    catch(ex){
        console.warn("warn:",ex);
    }
})()
```

简单比较会发现，async函数就是将Generator函数的星号（*）替换成async，将yield替换成await，同时不需要co模块，更加语义化。
但是与yeild又不完全相同，标准没有接收await*的语法(:(查看详情)，若需"并行"执行promise数组，推荐用Promise.All，所以需要并行请求时，需要这样写：

```
(async function (){
    let result= await Promise.all([
         (await
fetch('https://api.github.com/users/tj')).json(),
         (await
fetch('https://api.github.com/users/etoah')).json()
        ]);
    console.log("result:",result);
})();
```

虽说没有不能用 await*，总体来说结构还是简单清晰的

没有任何callback,流程和异常捕获是完全同步的写法。而且javascript语言级别支持这种写法。可以说这是异步的终极解决方案了。

总结

到这里，jser结合promise,yield,await的写法，可以和回调嵌套说拜拜了。

虽有这么多的不同的异步编程方式，但是异步编程的本质并没有变，只有对coder更友好了而已，但对工程化可读性和可维护性有很大的改进。

全文完，如有不严谨的地方，欢迎指正。