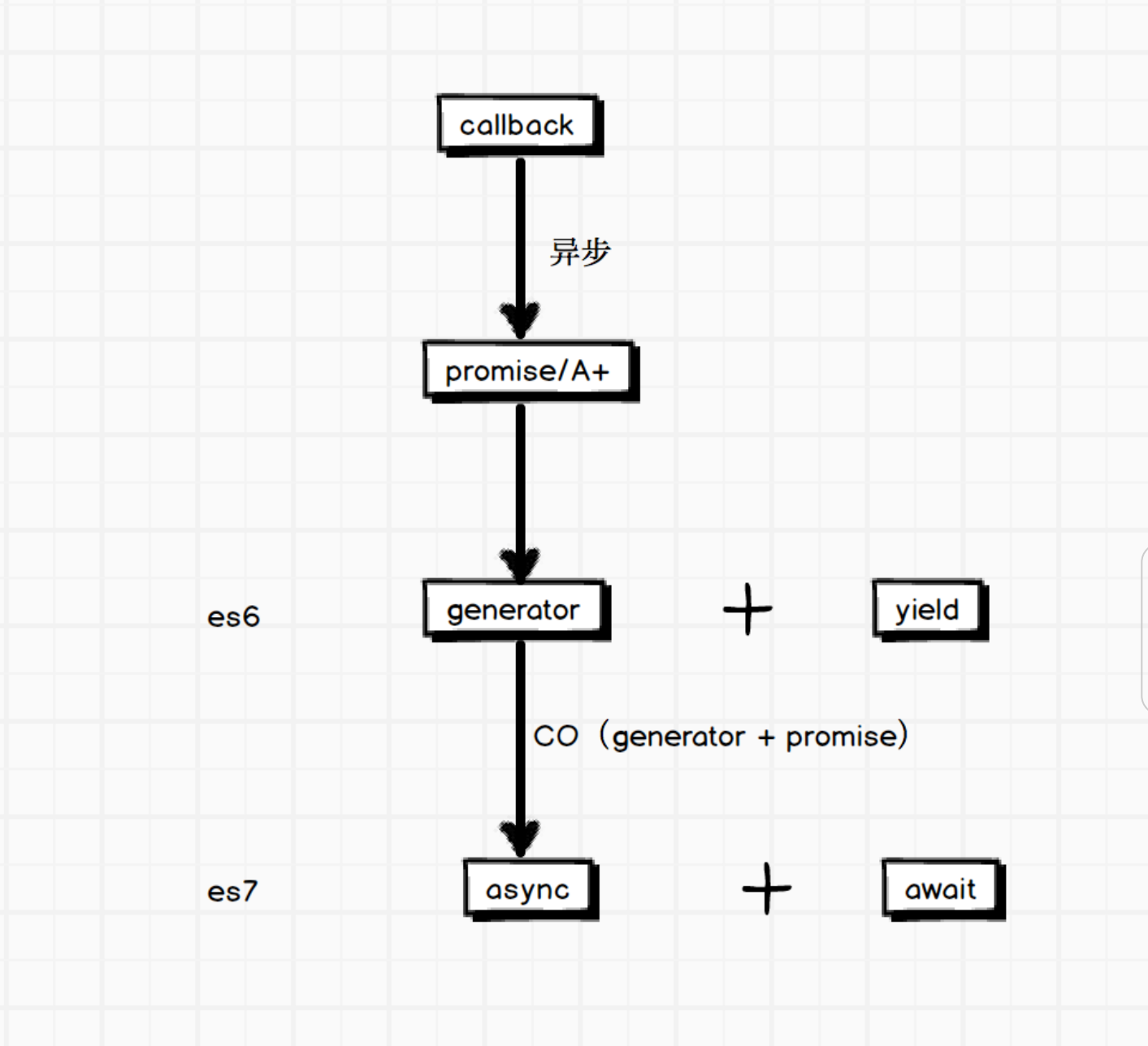
精华 **Node.js最新技术栈之Promise篇**

前言

js流程控制的演进过程，分以下5部分

- 1. 回调函数Callbacks
- 1. 异步JavaScript
- 1. Promise/a+
- 1. 生成器Generators/ yield
- 1. Async/ await

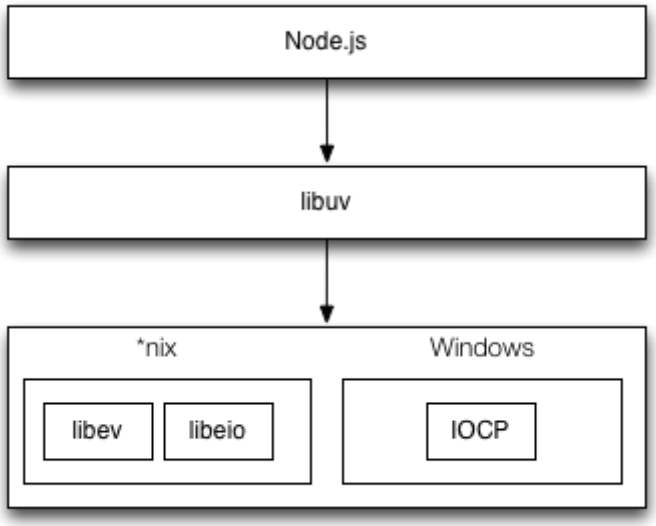


首先回调和异步是大家常见的，这里跳过，本节课主要讲promise/a+原理、实现和实践，并展望下一代的Generators/yield和Async/await等。

回到顶部

0）痛点

我们知道nodejs的最大的优点是高并发，适合适合I/O密集型应用



每一个函数都是异步，并发上去了，但痛苦也来了，比如

我现在要执行4个步骤，依次执行，于是就有下面的代码

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

这里只有4个，那如果更多呢？会不会崩溃，反正我会

有解决办法么？

答案是就是：promise/a+规范，下面我们看一下promise/a+规范

我们再来个伏笔，看一下链式写法

链式写法

先举个jQuery的链式写法例子

```
$('#tab').eq($(this).index()).show().siblings().hide();
```

链式写法的核心是：每个方法都返回this（自己看jquery源码）

下面举一个this的简单例子

```
var obj = {
  step1:function(){
    console.log('a');
    return this;
  },
  step2:function(){
    console.log('b');
    return this;
  },
  step3:function(){
    console.log('c');
    return this;
  },
  step4:function(){
    console.log('d');
    return this;
  }
}

console.log('-----\n');
obj.step1().step2().step3();
console.log('-----\n');
obj.step4().step2().step1();
```

这只是一个简单返回this的链式写法而已

执行结果

```
$ node doc/this.js
-----

a
b
c
-----

c
b
a
```

为啥要讲这个呢？先回到异步调用的场景，我们能这样调用么？如果每一个操作都可以连起来，是不是很爽

比如

```
return step1().step2().step3().step4()
```

回到顶部

这样做的好处

- 1）每一个操作都是独立的函数
- 2）可组装，拼就好了

我们还可以再苛刻点

- 1）如果要是上一步的结果作为下一步的输入就更好了（linux里的pipe：ps -ef|grep node|awk '{print $2}'|xargs kill -9）
- 2）如果出错能捕获异常就更好了
- 3）如果能在函数里面也能控制流程就更好了

会有人说："你要求的是不是太多了？"

"but，很好，我们来制定一个叫promisesaplus的规范吧，涵盖上面的所有内容"

于是就有了promise/a+规范...

1）promise/a+规范

这里面讲很多链式相关的东西，是我们异步操作里常见的问题，我们希望有更好的解决方案，promise/a+规范便应运而生

什么是promise/a+规范?

Promise表示一个异步操作的最终结果。与Promise最主要的交互方法是通过将函数传入它的then方法从而获取得Promise最终的值或Promise最终最拒绝（reject）的原因。

想想我们刚才的那个例子，让我们来庖解一下

- a) 异步操作的最终结果

简单点看，它也是一个链式操作,类似

```
return step1().step2().step3().step4()
```

- b) 与Promise最主要的交互方法是通过将函数传入它的then方法

但它有点差别

```
return step1().then(step2).then(step3).then(step4)
```

- c1) 从而获取得Promise最终的值或Promise最终最拒绝（reject）的原因

比如step1执行成功了，那么就会执行step2，如果step2执行成功了就会执行下一个，以此类推

那么如果失败呢？于是应该改成这样：

回到顶部

```
return step1().then(step2).then(step3).then(step4).catch(function(err){
  // do something when err
});
```

加了一个catch error回调，当出现异常就会直接到这个流程，会不会很好呢？

- c2) 从而获取得Promise最终的值或Promise最终最拒绝（reject）的原因

这句话了有一个reject，其实还对应一个resolve方法

- reject 是拒绝，跳转到catch error
- resolve 是解决，下一步，即跳转到下一个promise操作

我们还以上面的例子来说，如果step1失败了，走step3，step3失败了step4,如果step4出错就结束，报错，看一下代码

```
function step1(){
  if(false){
    return step3().then(step4).catch(function(err){
      // do something when err
    });
  }

}
return step1().then(step2).then(step3).then(step4).catch(function(err){
  // do something when err
});
```

总结一下：以上就是promise/a+规范的定义，我们拆解了一下，涵盖了上面我提出的几个痛点

术语

promise

是一个包含了兼容promise规范then方法的对象或函数

我们可以这样理解，每一个promise只要返回的可以then的都可以。就像上面举例返回的this一样，只要每一个都返回this，她就可以无限的链式下去，

这里的this约定为每一个对象或函数返回的都是兼容promise规范then方法。

thenable

是一个包含了then方法的对象或函数。

回到顶部

这个可以这样理解，上面的例子，每个方法都返回this，比较麻烦，

那能不能只有then方法promise对象，每一个操作都返回一样的promise对象，是不是就可以无限链接下去了？

value

是任何Javascript值。 (包括 undefined, thenable, promise等).

promise/a+规范约定的几个概念而已

exception

是由throw表达式抛出来的值。

上面讲的，当流程出现异常的适合，把异常抛出来，由catch err处理。

reason

是一个用于描述Promise被拒绝原因的值。

就是给你一个犯了错误，自我交待，争取宽大处理的机会。

参考

还有一些promise里流程处理相关的东西，比较复杂，没啥大意义，有兴趣的可以看看官方网址

- https://promisesaplus.com/ (https://promisesaplus.com/)

总结一下

- 1. 每个操作都返回一样的promise对象，保证链式操作
- 1. 每个链式都通过then方法
- 1. 每个操作内部允许犯错，出了错误，统一由catch error处理
- 1. 操作内部，也可以是一个操作链，通过reject或resolve再造流程

理解这些，实际上nodejs里就没有啥难点了。当然我还要告诉更多好处，让你受益一生（吹吹牛）

八卦一下

回到顶部

哪个语言不需要回调呢？callback并不是js所特有的，举个例子oc里的block（如果愿意扯，也可以扯扯函数指针），如果用asi或者afnetworking处理http请求，你都可以选择用block方式，那么你如果一个业务里有多个请求呢？

你也只能嵌套，因为嵌套没有太多层，因为可以有其他解耦方式，其实我最想表达的是，它也可以使用promise方式的。

这里展示一个例子

```
[self login].then(^{

    // our login method wrapped an async task in a promise
    return [API fetchKittens];

}).then(^(NSArray *fetchedKittens){

    // our API class wraps our API and returns promises
    // fetchKittens returned a promise that resolves with an array of kittens
    self.kittens = fetchedKittens;
    [self.tableView reloadData];

}).catch(^(NSError *error){

    // any errors in any of the above promises land here
    [[[UIAlertView alloc] init…] show];

});
```

详见http://promisekit.org/introduction/

是不是很奇怪，咋和上面的promise写法一样呢？

官方解释是

```
PromiseKit is not just a promises implementation
```

好吧，这只是冰山一角

如果各位熟悉前端js，相信你一定了解

- jQuery（1.5+）的deferred
- Angularjs的$q对象

nodejs里的实现

- bluebird （https://github.com/petkaantonov/bluebird (https://github.com/petkaantonov/bluebird)
  后面继续讲，保持神秘）

- q （https://github.com/kriskowal/q (https://github.com/kriskowal/q) Angularjs的$q对象是q的精简版）
- then （teambition作品 https://github.com/teambition/then.js (https://github.com/teambition/then.js) 没用过）
- when （https://github.com/cujojs/when (https://github.com/cujojs/when) 没用过）
- async （https://github.com/caolan/async (https://github.com/caolan/async) 最简单的）
- eventproxy（朴灵作品 https://github.com/JacksonTian/eventproxy，使用event来处理流程，也是不错的尝试）(https://github.com/JacksonTian/eventproxy%EF%BC%8C%E4%BD%BF%E7%94%A8event%E6%9D%A5%E5%A4%84%E7%90%86%E6%B5%81%E7%A8%8B%EF%BC%8C%E4%B9%9F%E6%98%AF%E4%B8%8D%E9%94%99%E7%9A%84%E5%B0%9D%E8%AF%95%EF%BC%89)

其他语言实现，详见 https://promisesaplus.com/implementations (https://promisesaplus.com/implementations)

其实，只要掌握了promise/a+规范，你就可以在n种语言里使用了

2）如何实现

promise/a+规范是一个通用解决方案，不只是对nodejs管用，只要你掌握了原理，就只是换个语言实现而已

下面我就带着大家看一下js里是如何实现的

回到顶部

```
var Promise = function () {
};

var isPromise = function (value) {
    return value instanceof Promise;
};

var defer = function () {
    var pending = [], value;
    var promise = new Promise();
    promise.then = function (callback) {
        if (pending) {
            pending.push(callback);
        } else {
            callback(value);
        }
    };
    return {
        resolve: function (_value) {
            if (pending) {
                value = _value;
                for (var i = 0, ii = pending.length; i < ii; i++) {
                    var callback = pending[i];
                    callback(value);
                }
                pending = undefined;
            }
        },
        promise: promise
    };
};
```

首先

1） 声明promise对象

var promise = new Promise();

2）给promise对象增加then方法

promise.then = function (callback) {

3）给defer对象返回resolve和promise

4）value，在resolve事件里传参，是第几个就执行第几个

本来想讲的更多一点，但时间不允许啊（后面还有很多内容），此处暂时讲到这里

cnode里的William17写的挺好的，完整的实现可以参考

回到顶部

- https://cnodejs.org/topic/5603cb8a152fdd025f0f5014 (https://cnodejs.org/topic/5603cb8a152fdd025f0f5014)
- https://github.com/William17/taxi (https://github.com/William17/taxi)

关于Q

q是一个不错的项目，也是比较早的promise实现，而且angularjs的$q就是它的精简版，如果掌握了q，学习angular和bluebird都会比较简单

当然它还有更棒的，它把q的7个版本是如何实现的都详细记录了，刚才我给出的v1实际上就q的早期版本

https://github.com/kriskowal/q/tree/v1/design (https://github.com/kriskowal/q/tree/v1/design)

很多人都以为是跟着人学，但人有不确定性，而且互联网让世界是平的了，我们如果能够学会从开源项目里学习，这才是长久的学习力

曾经写过一篇《如何学习之善用github篇：向 @Pana (/user/Pana) 学习》，有兴趣的可以去翻翻

http://mp.weixin.qq.com/s?__biz=MzAxMTU0NTc4Nw==&mid=223428929&idx=1&sn=8296f5a96aa34f836ab83000f7ad995b#rd (http://mp.weixin.qq.com/s?__biz=MzAxMTU0NTc4Nw==&mid=223428929&idx=1&sn=8296f5a96aa34f836ab83000f7ad995b#rd)

3）真实项目里的实践

技术选项：先看基准测试

2015-01-05 当时最新的模块，比较结果如下

bench doxbee-sequential

```
results for 10000 parallel executions, 1 ms per I/O op

file                                       time(ms)  memory(MB)
callbacks-baseline.js                           232       35.86
promises-bluebird-generator.js                  235       38.04
promises-bluebird.js                            335       52.08
promises-cujojs-when.js                         405       75.77
promises-tildeio-rsvp.js                        468       87.56
promises-dfilatov-vow.js                        578      125.98
callbacks-caolan-async-waterfall.js             634       88.64
promises-lvivski-davy.js                        653      109.64
promises-calvinmetcalf-lie.js                   732      165.41
promises-obvious-kew.js                        1346      261.69
promises-ecmascript6-native.js                 1348      189.29
generators-tj-co.js                            1419      164.03
promises-then-promise.js                       1571      294.45
promises-medikoo-deferred.js                   2091      262.18
observables-Reactive-Extensions-RxJS.js        3201      356.76
observables-caolan-highland.js                 7429      616.78
promises-kriskowal-q.js                        9952      694.23
observables-baconjs-bacon.js.js               25805      885.55

Platform info:
Windows_NT 6.1.7601 x64
Node.JS 1.1.0
V8 4.1.0.14
Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz × 4
```

bench parallel (`--p 25`)

```
results for 10000 parallel executions, 1 ms per I/O op

file                                       time(ms)  memory(MB)
callbacks-baseline.js                           211       25.57
promises-bluebird.js                            389       53.49
promises-bluebird-generator.js                  491       55.52
promises-tildeio-rsvp.js                        785      108.14
promises-dfilatov-vow.js                        798      102.08
promises-cujojs-when.js                         851       60.46
promises-calvinmetcalf-lie.js                  1065      187.69
promises-lvivski-davy.js                       1298      135.43
callbacks-caolan-async-parallel.js             1780      101.11
promises-then-promise.js                       2438      338.91
promises-ecmascript6-native.js                 3532      301.96
promises-medikoo-deferred.js                   4207      357.60
promises-obvious-kew.js                        8311      559.24

Platform info:
Windows_NT 6.1.7601 ia32
Node.JS 0.11.14
V8 3.26.33
Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz × 4
```

对比一下结果

顺序执行

```
promises-bluebird-generator.js                235       38.04
promises-bluebird.js                          335       52.08
```

并发执行

```
promises-bluebird.js                      389       53.49
promises-bluebird-generator.js            491       55.52
```

不管是哪种，bluebird都是前三名。然后扒一扒第一名的 `callbacks-baseline.js` ,其实就我们最讨厌的最基本的回调写法，也就说，bb实际比原生js稍慢，但比其他promise库都快

综合一下bb的特性

- 速度最快
- api和文档完善，（对各个库支持都不错）
- 支持generator等未来发展趋势
- github活跃
- 还有让人眼前一亮的功能点（保密，下面会讲）

我们的结论只有bb是目前最好的最好的选择

我个人的一个小喜好，对caolan的async挺喜欢的，我的选项

- 公司项目是bb
- 开源大点的还是bb

回到顶部

- 小工具啥的我会选async

其实when，then，eventproxy等也不错，只是懒得折腾

大家在直播过程中有任何不明白或者想提问的可以随时私信给小助手，我会在答疑阶段统一回答

场景

下面举几个实际场景

神器，bluebird的promisify

promisify原理

就是你给他传一个对象或者prototype，它去遍历，给他们加上async方法，此方法返回promise对象，你就可以为所欲为了

- 优点：使用简单
- 缺点：谨防对象过大，内存问题

nodejs api支持

```
//Read more about promisification in the API Reference:
//API.md
var fs = Promise.promisifyAll(require("fs"));

fs.readFileAsync("myfile.json").then(JSON.parse).then(function (json) {
    console.log("Successful json");
}).catch(SyntaxError, function (e) {
    console.error("file contains invalid json");
}).catch(Promise.OperationalError, function (e) {
    console.error("unable to read file, because: ", e.message);
});
```

熟悉fs的API的都知道，fs有

- fs.readFile
- fs.readFileSync

但没有

- fs.readFileAsync

实际上 `fs.readFileAsync` 是bluebird加上去的

```
var fs = Promise.promisifyAll(require("fs"));
```

回到顶部

其实也没什么神奇的，只是bb做的更多而已，让调用足够简便

下面我们来看一下mvc里的model层如何使用bb，我们先假定使用的是mongoose啊

先定义模型

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var Promise = require("bluebird");

UserSchema = new Schema({
  username: String,
  password: String,
  created_at: {
    type: Date,
    "default": Date.now
  }
});

var User = mongoose.model('User', UserSchema);

Promise.promisifyAll(User);
Promise.promisifyAll(User.prototype);
```

利用上面定义好的User模型，我们就可以写业务逻辑了

```
User.findAsync({username: username}).then(function(data) {
   ...
}).catch(function(err) {
   ...
});
```

再来个复杂的

回到顶部

```
TeamPartner.updateAsync({
    ...
}, {
    $set: {
        ...
    }
}).then(function() {
    return User.findByIdAsync(user_id);
}).then(function(team_owner) {
    if(typeof team_owner.partner_count != 'undefined') {
      team_new_count = team_owner.partner_count + 1;
  }else{
      team_new_count = 1;
  }

  // 创始人成员+1
  return User.findByIdAndUpdateAsync(user_id, {
    $set: {
       ...
    }
  }, {
      upsert: true
  });
}).then(function() {
    return Notify.findByIdAndUpdateAsync(notify_id, {
        $set: {
            read_status: true
        }
    });
}).then(function() {
  var notifySave = new Notify({
      ...
  });

  return notifySave.saveAsync();
}).then(function() {
    return User.findByIdAsync(user_join);
}).then(function(joiner) {
  var teamSave = new Team({
      ...
  });
  return teamSave.saveAsync();
}).then(function() {
    res.status(200).json({
        data: {},
        status: {
            code: 0,
```

```
            msg: 'success'
        }
    });
}).catch(function(err) {
    console.log(err);
    res.status(200).json({
        data: {},
        status: {
            code: err.code,
            msg: err.name
        }
    });
});
```

很抱歉，伤害大家了，但确切是有这样用，我想问，脑子抽么？

优化方案

- mongoose上的static和method上扩展，别暴漏太多细节在控制层
- 面向promise，保证每个操作都是一个函数，让流程可以组装，不要忘了最初then的初衷

```
function find_user() {
    return User.findByIdAsync(user_id);
}

function find_user2() {
    return User.findByIdAsync(user_id);
}

function error(ex){
  console.log(ex);
}

TeamPartner.updateByXXAsync(a,b,c).then(find_user).then(find_user2).catch(error)
```

回到顶部

开源项目里的**promise**

- ioredis
- mongoose && mongoskin

ioredis

https://github.com/luin/ioredis (https://github.com/luin/ioredis)

ioredis是redis库里的首选，其实它是基于node_redis的，增加和优化了很多，其中关于promise的一点是

```
Delightful API. It works with Node callbacks and Bluebird promises.
```

用法

```
redis.get('foo').then(function (result) {
  console.log(result);
});
```

真心比node_redis好用的多。

luin是《Redis入门指南》一书作者，90后，全栈+设计，写过不少nodejs模块，目前就在群里

mongoose && mongoskin

mongoose是nodejs处理mongodb的不二选择，简单说一下mongoskin

官方说

```
The promise wrapper for node-mongodb-native.
```

做了很多api上的优化，还是不错的，可是mongoose也出了promise

mongoose仿佛在喊：“用我，我也可以then”。。。。

4）最后给出未来展望

未来主要是es6和es7上的实现

- es6上的Generators/yield
- es7上的Async/await

都是好东西，可以尝试，毕竟早晚js还是要升级的，先做技术储备吧

生成器Generators/yield

生成器是ES6（也被称为ES2015）的新特性。

想象下面这样的一个场景：

> 当你在执行一个函数的时候，你可以在某个点暂停函数的执行，并且做一些其他工作，然后再返回这个函数继续执行， 甚至是携带一些新的值，然后继续执行。

上面描述的场景正是JavaScript生成器函数所致力于解决的问题。

当我们调用一个生成器函数的时候，它并不会立即执行， 而是需要我们手动的去执行迭代操作（next方法）。也就是说，你调用生成器函数，它会返回给你一个迭代器。迭代器会遍历每个中断点。

看例子

```
function* foo () {
  var index = 0;
  while (index < 2) {
    yield index++; //暂停函数执行，并执行yield后的操作
  }
}

var bar =  foo(); // 返回的其实是一个迭代器

console.log(bar.next());    // { value: 0, done: false }
console.log(bar.next());    // { value: 1, done: false }
console.log(bar.next());    // { value: undefined, done: true }
```

如果你想更轻松的使用生成器函数来编写异步JavaScript代码，我们可以使用 co 这个库，co是著名的tj大神写的，是一个为Node.js和浏览器打造的基于生成器的流程控制工具，借助于Promise，你可以使用更加优雅的方式编写非阻塞代码。

使用co，改写前面的示例代码：

```
co(function* (){
  yield Something.save();
}).then(function() {
  // success
})
.catch(function(err) {
  //error handling
});
```

你可能会问：如何实现并行操作呢？答案可能比你想象的简单，如下

```
yield [a.save(), b.save()];
```

其实它就是Promise.all，换种写法而已

Async/await

在ES7（还未正式标准化）中引入了Async函数的概念，目前如果你想要使用的话，只能借助于babel 这样的语法转换器将其转为ES5代码。

使用async关键字，可以轻松地达成之前使用生成器和co函数所做到的工作。当然，hack除外

回到顶部

可以这样理解，async函数实际使用的是Promise，然后使用await来执行异步操作，这里的await类似于上面讲的yield

因此，使用async重写上面的代码：

```
async function save(Something) {
  try {
    await Something.save(); // 等待await后面的代码执行完，类似于yield
  } catch (err) {
    //error handling
  }
  console.log('success');
}
```

下一代web框架Koa也支持async函数，如果你也在使用koa，那么你现在就可以借助babel使用这一特性了

```
import koa from koa;
let app = koa();

app.experimental = true;

app.use(async function (){
    this.body = await Promise.resolve('Hello Reader!')
})

app.listen(3000);
```

这里报个料，群里fundon (https://github.com/fundon) 大神写的https://github.com/trekjs/trek是基于koa的，使用babel编译，并有更多相当棒的特性。爱折腾的可以试试

最后还是要归纳总结一下，

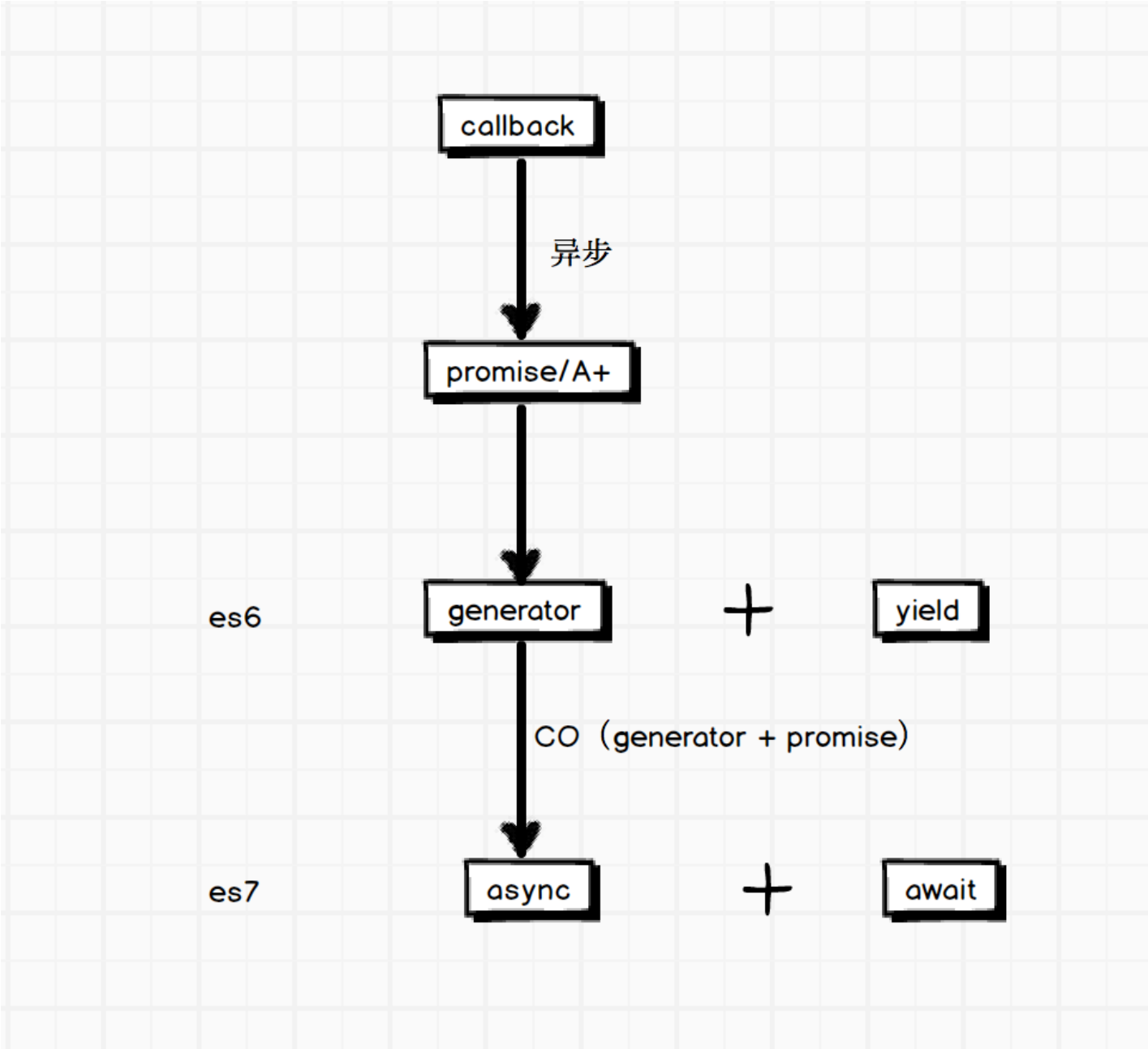
先复习promise/a+的四个要点

- a) 异步操作的最终结果，尽可能每一个异步操作都是独立操作单元
- b) 与Promise最主要的交互方法是通过将函数传入它的then方法（thenable）
- c) 捕获异常catch error
- d) 根据reject和resolve重塑流程

已经第三遍，该记住了！

继续看第一个图

回到顶部



再来看这张图，从callback到promose是历史的必然产物

generator是一种新的定义方式，定义操作单元，尤其在迭代器的情况，搭配yield来执行，可读性上差了很多，好处是真的解耦了

co是一个中间产品，可以说是给generator增加了promise实现，可读性和易用性是愿意好于generator + yield的

最后我们看看async，它实际上是通过async这个关键词，定义的函数就可以返回promise对象，可以说async就是能返回promise对象的generator。yield关键词以及被generator绑架了，那它就换个名字，叫await

其实从这段历史来看，反复就是promise上的折腾，只是加了generator这个别名，只是async是能返回promise的generator

这样理解是不是更简单呢?

万变不离其宗，想精通js或者nodejs，promise是大家必须迈过去的坑。

回到顶部

谢谢大家，今天讲的内容就到这里，如果有什么讲的不对的、不合理的，请不吝赐教，共同学习