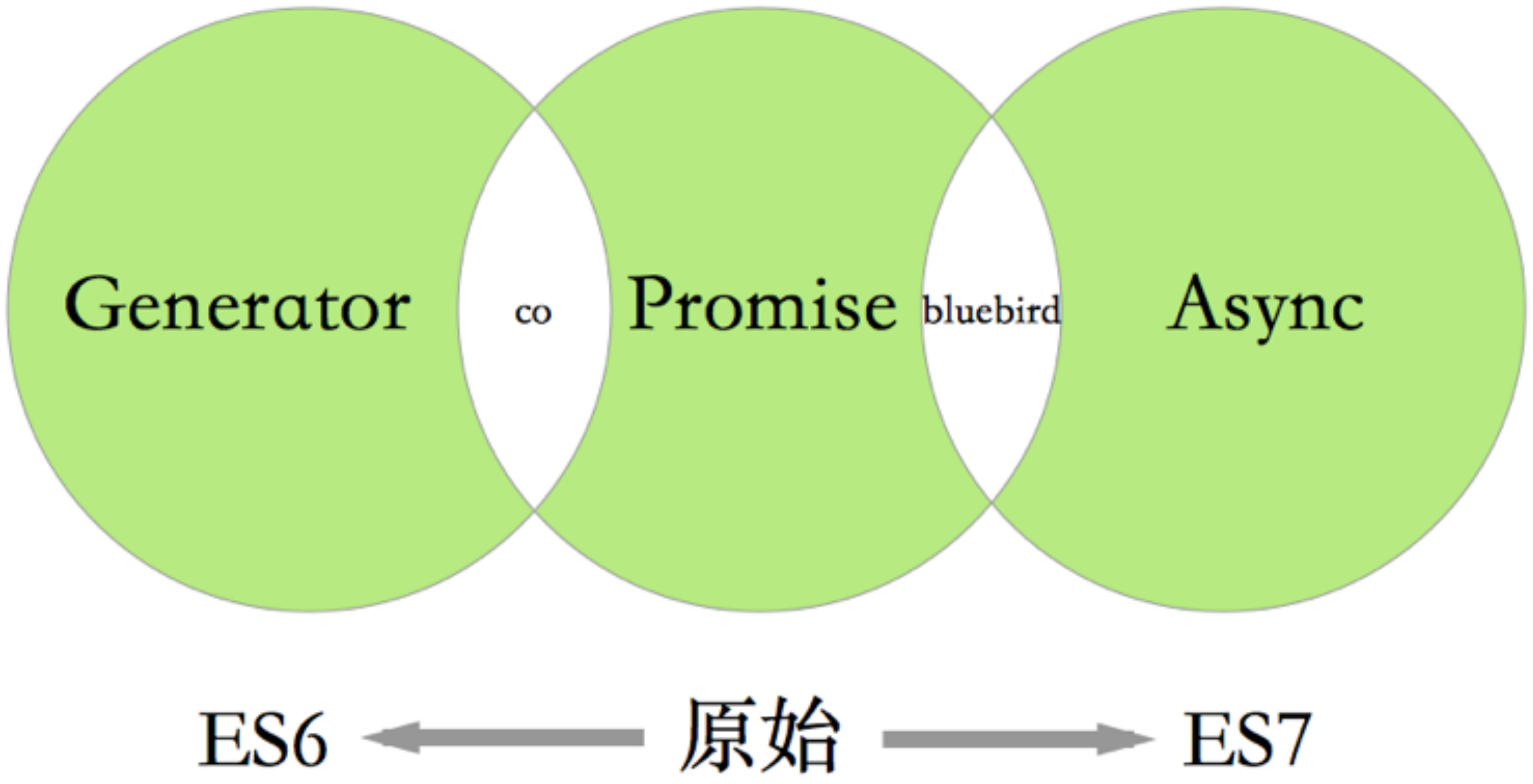
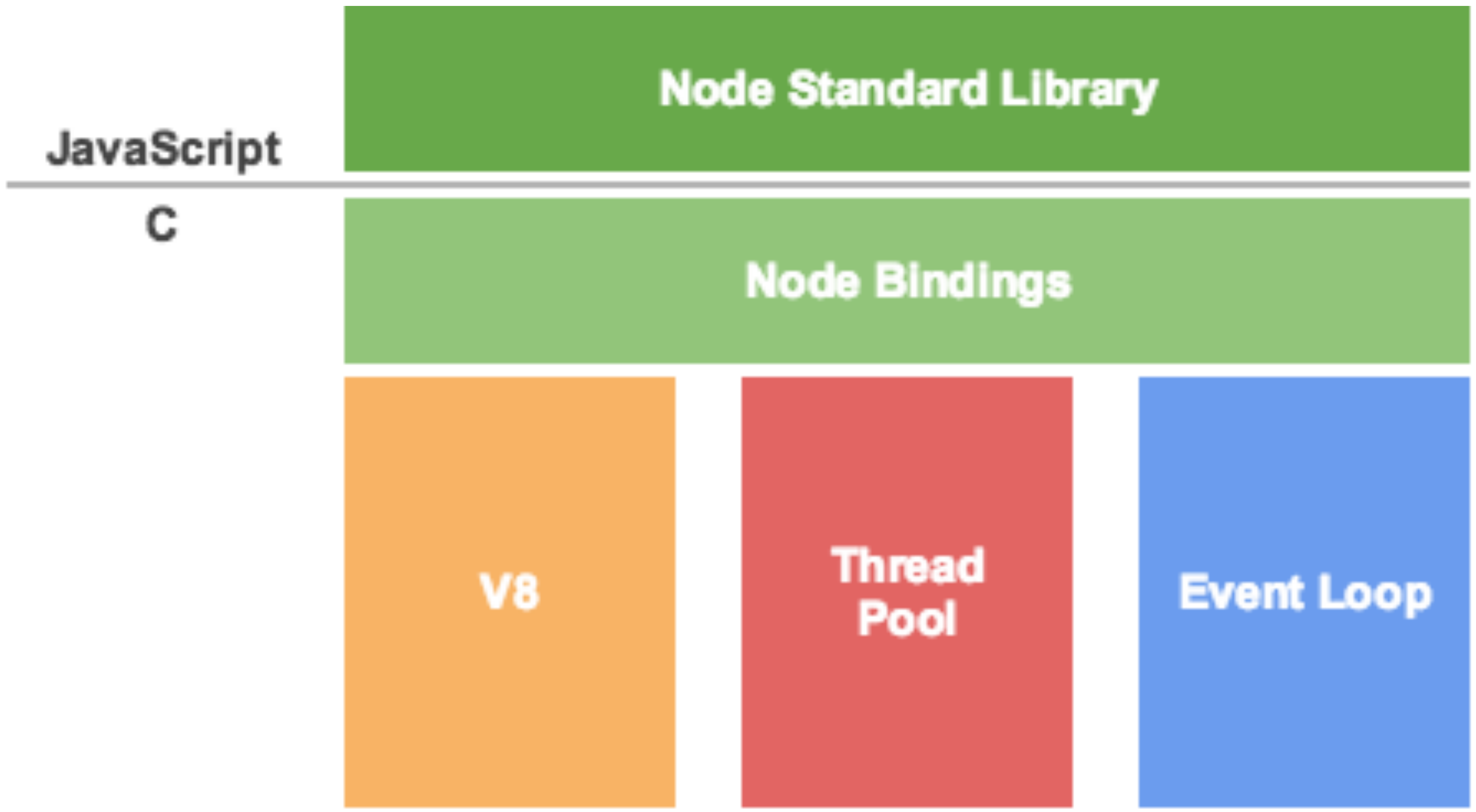
Node.js异步流程控制

@i5ting

何去何从?



异步的Node


Node Standard Library
JavaScript
C
Node Bindings
V8
Thread Pool
Event Loop

If Google’s V8 Engine is the heart of your Node.js application,

then callbacks are its veins.

约定写法

- CommonJS
- Error-first callback (SDK写法)
- Event Emiter（观察者模式）

示例

```
function(err, res) {
    // process the error and result
}
```

示例

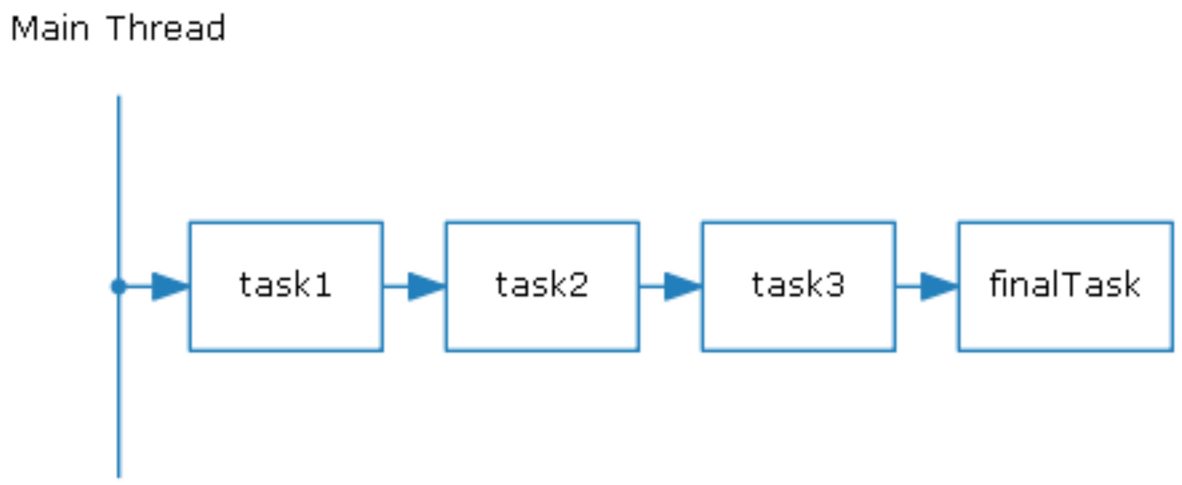
```
var fs = require(‘fs');

function readJSON(filePath, callback) {
  fs.readFile(filePath, function(err, data) {
    callback(null, JSON.parse(data));
  });
}

readJSON('./package.json', function (err, pkg) { ... }
```

回调地狱

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



理想的写法

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

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

Promise/a+

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…

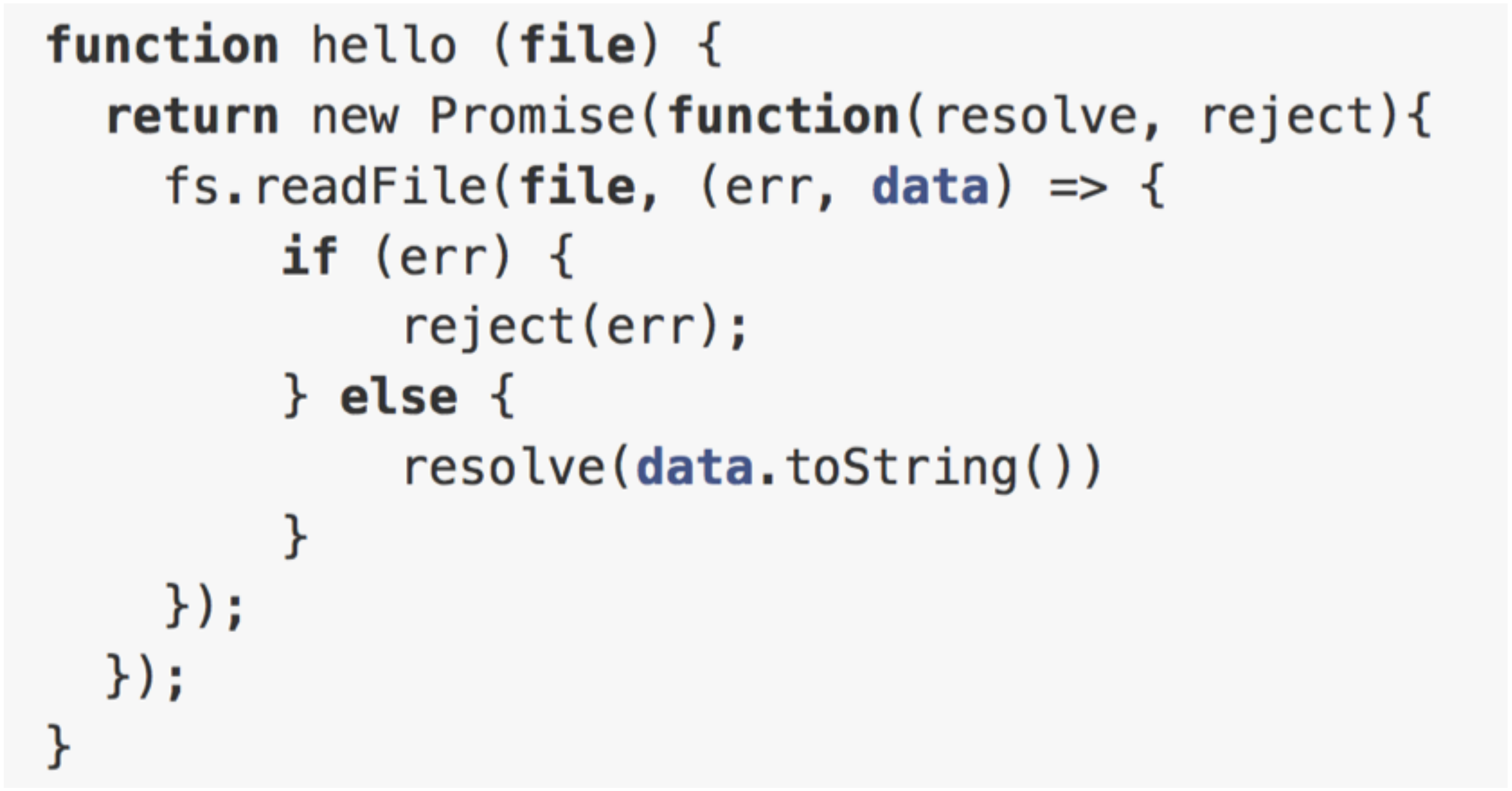
a) 异步操作的最终结果

b) 与Promise最主要的交互方法是通过将函数传入它的then方法

约定

a) 异步操作的最终结果

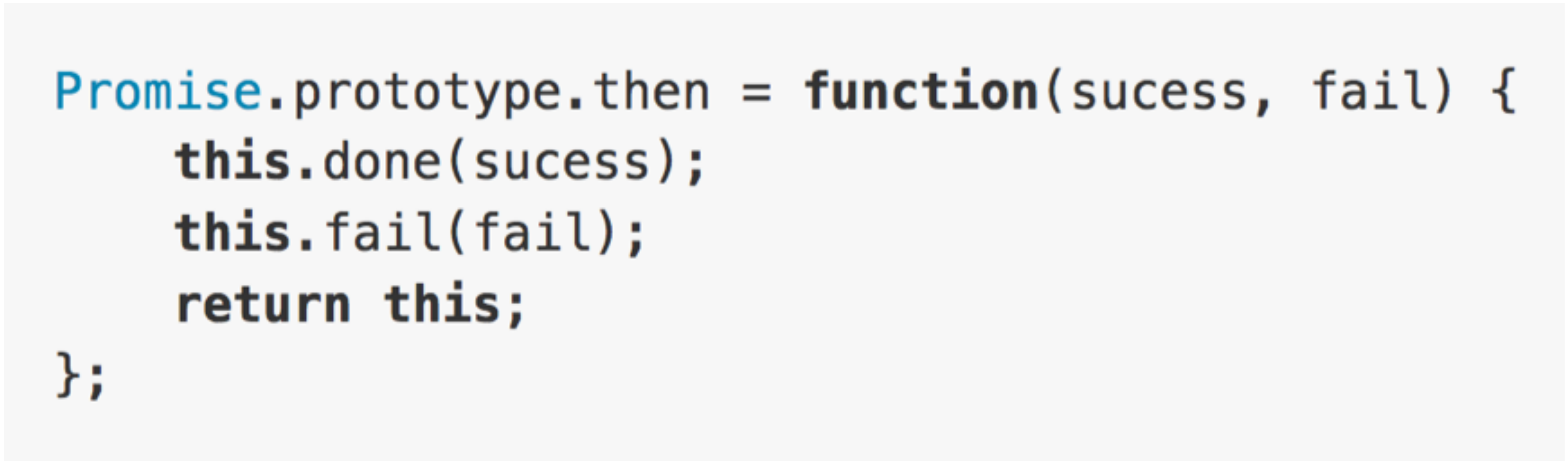
b) 与Promise最主要的交互方法是通过将函数传入它的then方法



```
function hello (file) {
  return new Promise(function(resolve, reject){
    fs.readFile(file, (err, data) => {
        if (err) {
            reject(err);
        } else {
            resolve(data.toString())
        }
    });
  });
}
```

约定

链式的thenable

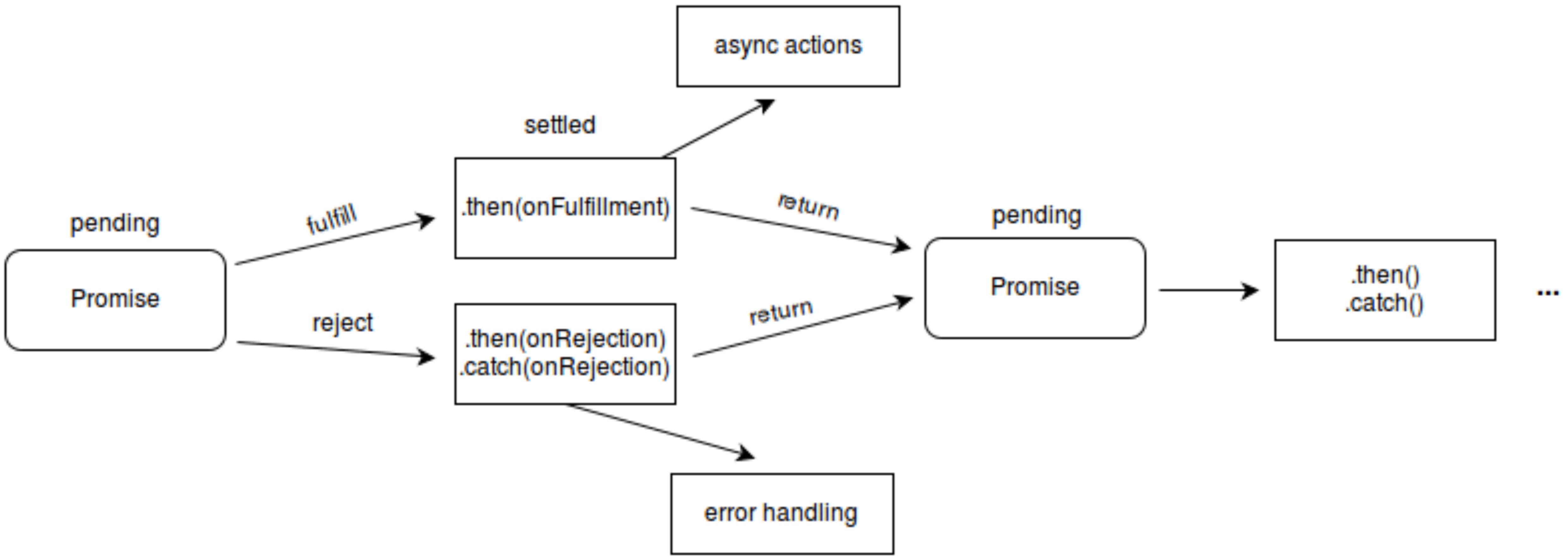


```
Promise.prototype.then = function(sucess, fail) {
    this.done(sucess);
    this.fail(fail);
    return this;
};
```

Promise应该长成这样

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

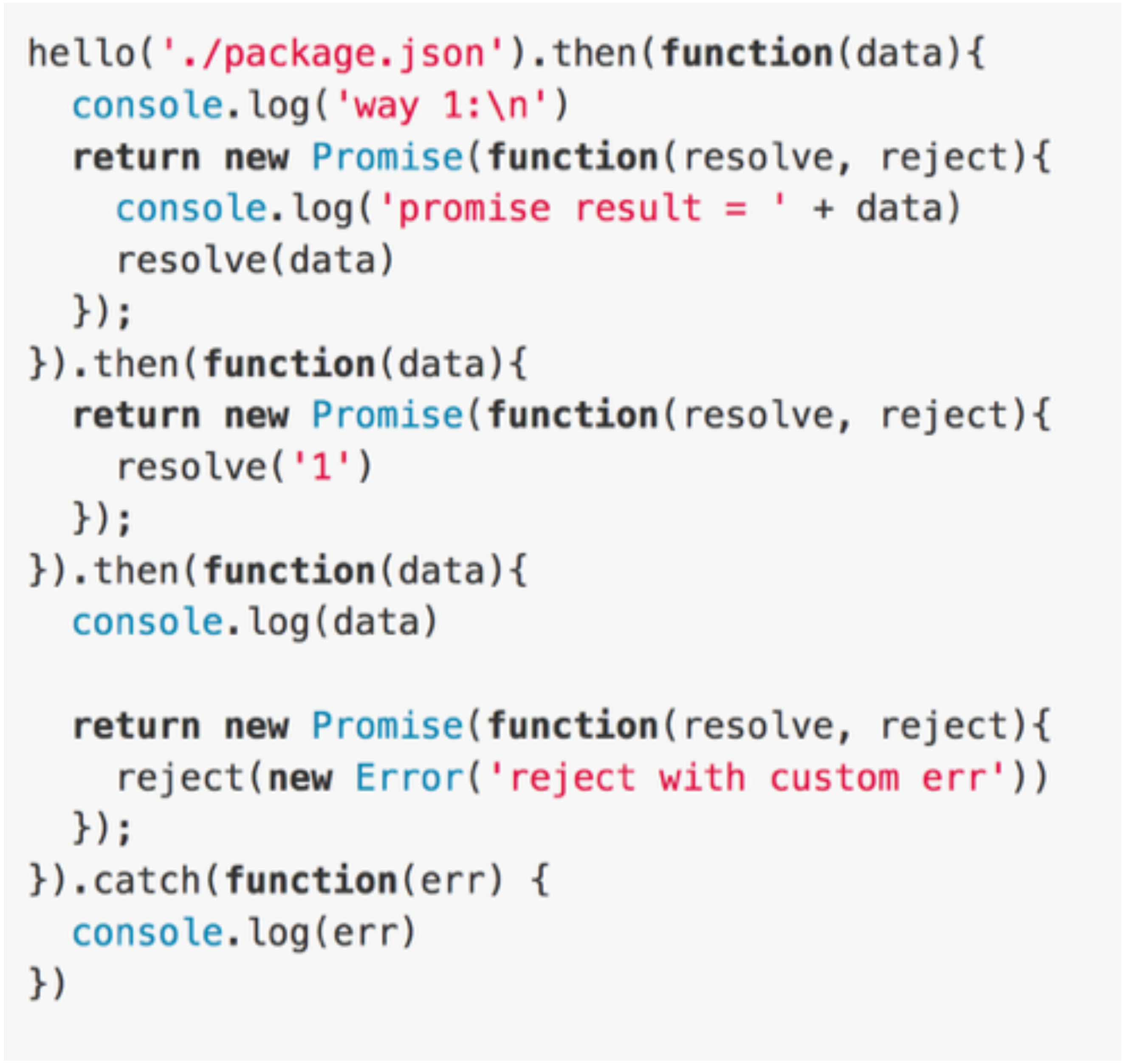
状态转换


async actions
settled
pending
Promise
fulfill
.then(onFulfillment)
return
pending
Promise
.then()
.catch()
...
reject
.then(onRejection)
.catch(onRejection)
return
error handling

总结：四个要点

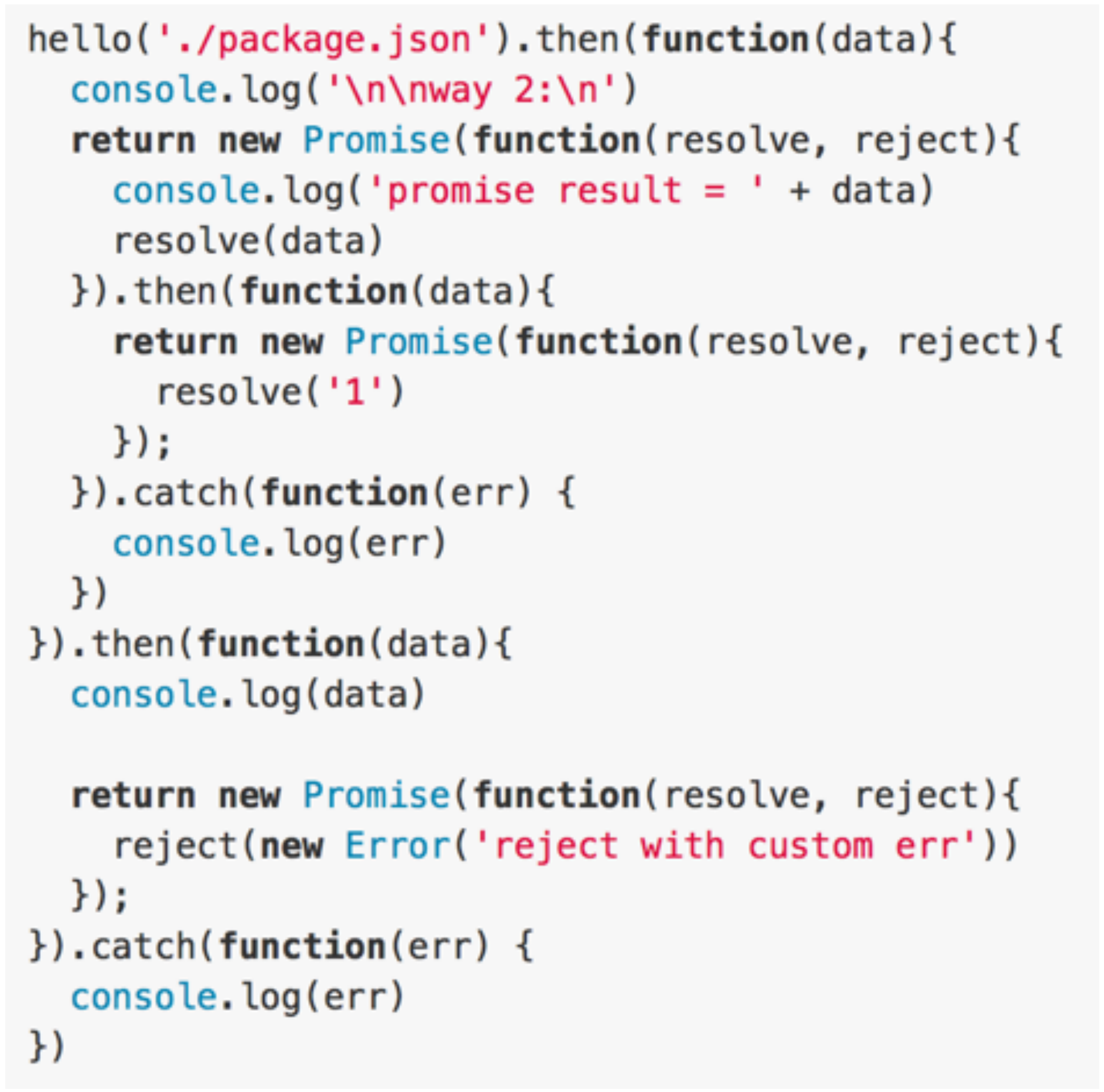
- 每个操作都返回一样的promise对象，保证链式操作
- 每个链式都通过then方法
- 每个操作内部允许犯错，出了错误，统一由catch error处理
- 操作内部，也可以是一个操作链，通过reject或resolve再造流程

流程重塑



```
hello('./package.json').then(function(data){
  console.log('way 1:\n')
  return new Promise(function(resolve, reject){
    console.log('promise result = ' + data)
    resolve(data)
  });
}).then(function(data){
  return new Promise(function(resolve, reject){
    resolve('1')
  });
}).then(function(data){
  console.log(data)

  return new Promise(function(resolve, reject){
    reject(new Error('reject with custom err'))
  });
}).catch(function(err) {
  console.log(err)
})
```



```
hello('./package.json').then(function(data){
  console.log('\n\nway 2:\n')
  return new Promise(function(resolve, reject){
    console.log('promise result = ' + data)
    resolve(data)
  }).then(function(data){
    return new Promise(function(resolve, reject){
      resolve('1')
    });
  }).catch(function(err) {
    console.log(err)
  })
}).then(function(data){
  console.log(data)

  return new Promise(function(resolve, reject){
    reject(new Error('reject with custom err'))
  });
}).catch(function(err) {
  console.log(err)
})
```

```
var step1 = function(data){
  console.log('\n\nway 3:\n')
  return new Promise(function(resolve, reject){
    console.log('promise result = ' + data)
    resolve(data)
  }).then(function(data){
    return new Promise(function(resolve, reject){
      resolve('1')
    });
  }).catch(function(err) {
    console.log(err)
  })
}

var step2 = function(data){
  console.log(data)

  return new Promise(function(resolve, reject){
    reject(new Error('reject with custom err'))
  });
}

hello('./package.json').then(step1).then(step2).catch(function(err) {
  console.log(err)
})
```

```
var requireDirectory = require('require-directory');
module.exports = requireDirectory(module);
```

```
var fs = require("fs");

module.exports = function hello (file) {
  return new Promise(function(resolve, reject){
    fs.readFile(file, (err, data) => {
        if (err) {
            reject(err);
        } else {
            resolve(data.toString())
        }
    });
  });
}
```

```
var tasks = require('./tasks')

tasks.hello('./package.json').then(tasks.step1).then(tasks.step2).catch(function(err) {
  console.log(err)
})
```

异常处理

常用的处理方式是全局处理，即所有的异步操作都由一个catch来处理

```
promise.then(function(result) {
  console.log('Got data!', result);
}).catch(function(error) {
  console.log('Error occurred!', error);
});
```

当然，then方法的第二个参数也是可以的

```
promise.then(function(result) {
  console.log('Got data!', result);
}).then(undefined, function(error) {
  console.log('Error occurred!', error);
});
```

```
try {
  throw new Error('never will know this happened')
} catch (e) {}
```

在promises里可以这样写

```
readFile()
  .then(function (data) {
    throw new Error('never will know this happened')
  })
```

为了打印errors，这里以简单的.then(null, onRejected)语句为例

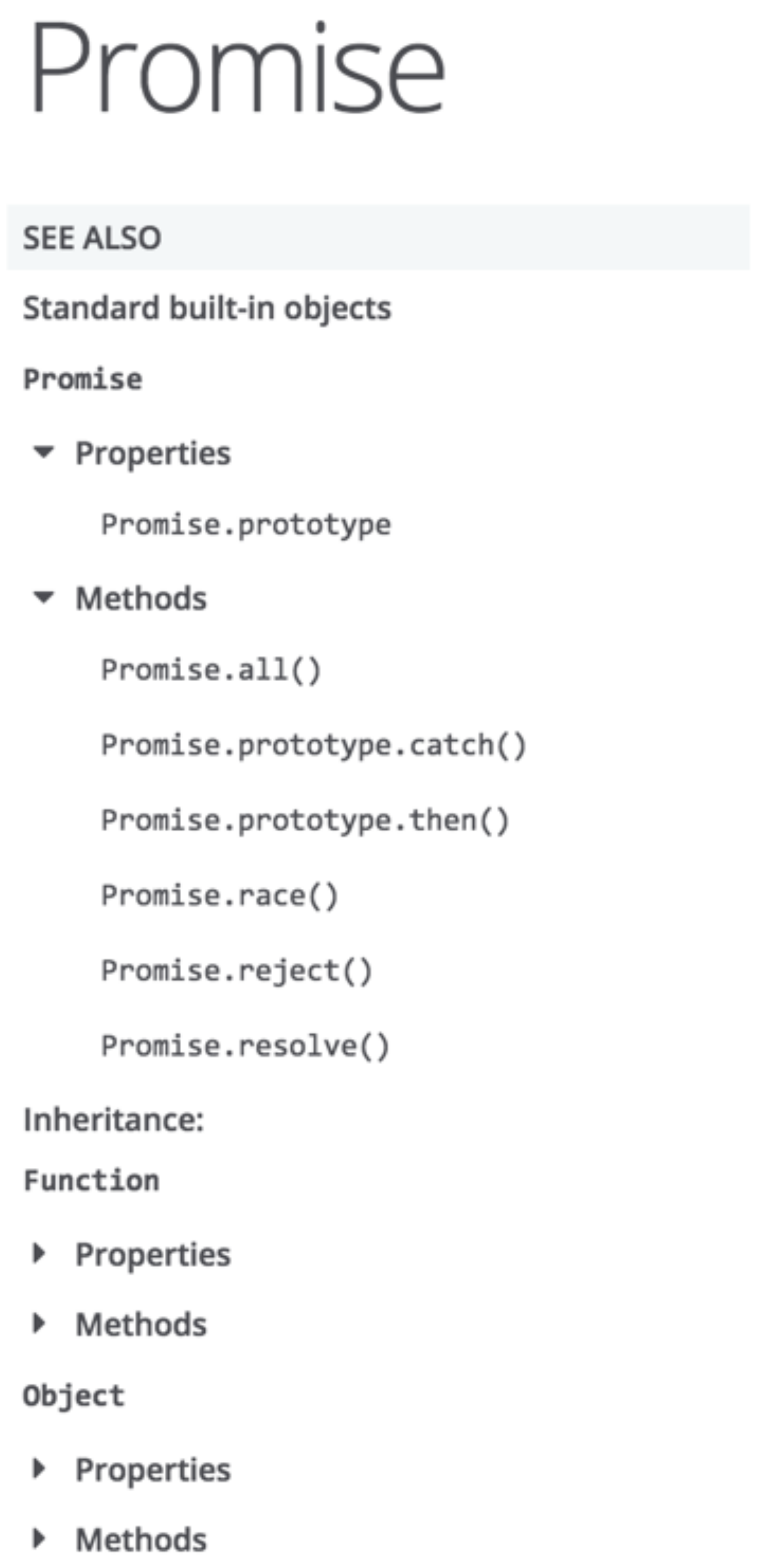
```
readFile()
  .then(function (data) {
    throw new Error('now I know this happened')
  })
  .then(null, console.error)
```

```
var p1 = new Promise(function(resolve, reject) {
  resolve('Success');
});

p1.then(function(value) {
  console.log(value); // "Success!"
  return Promise.reject('oh, no!');
}).catch(function(e) {
  console.log(e); // "oh, no!"
  // return Promise.reject('oh, no! 2');
}).then(function(){
  console.log('after a catch the chain is restored');
}, function () {
  console.log('Not fired due to the catch');
});
```

子流程也极其重要

你需要了解的 Promise Api



Promise

SEE ALSO

Standard built-in objects

`Promise`

- Properties
  - `Promise.prototype`
- Methods
  - `Promise.all()`
  - `Promise.prototype.catch()`
  - `Promise.prototype.then()`
  - `Promise.race()`
  - `Promise.reject()`
  - `Promise.resolve()`

Inheritance:

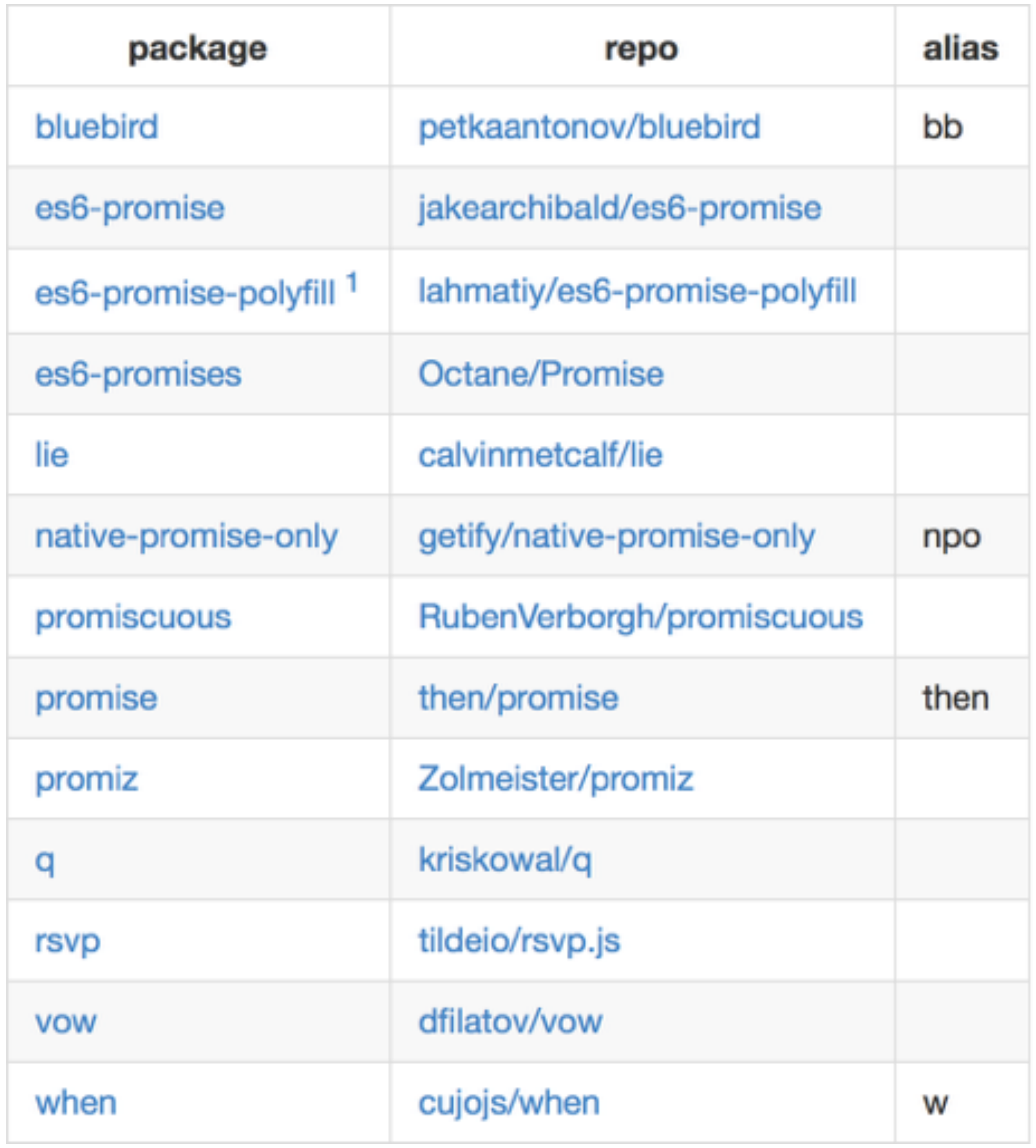
`Function`

- Properties
- Methods

`Object`

- Properties
- Methods

Promise选型



package	repo	alias
bluebird	petkaantonov/bluebird	bb
es6-promise	jakearchibald/es6-promise	
es6-promise-polyfill [1]	lahmatiy/es6-promise-polyfill	
es6-promises	Octane/Promise	
lie	calvinmetcalf/lie	
native-promise-only	getify/native-promise-only	npo
promiscuous	RubenVerborgh/promiscuous	
promise	then/promise	then
promiz	Zolmeister/promiz	
q	kriskowal/q	
rsvp	tildeio/rsvp.js	
vow	dfilatov/vow	
when	cujojs/when	w

Promise/A+是通用规范

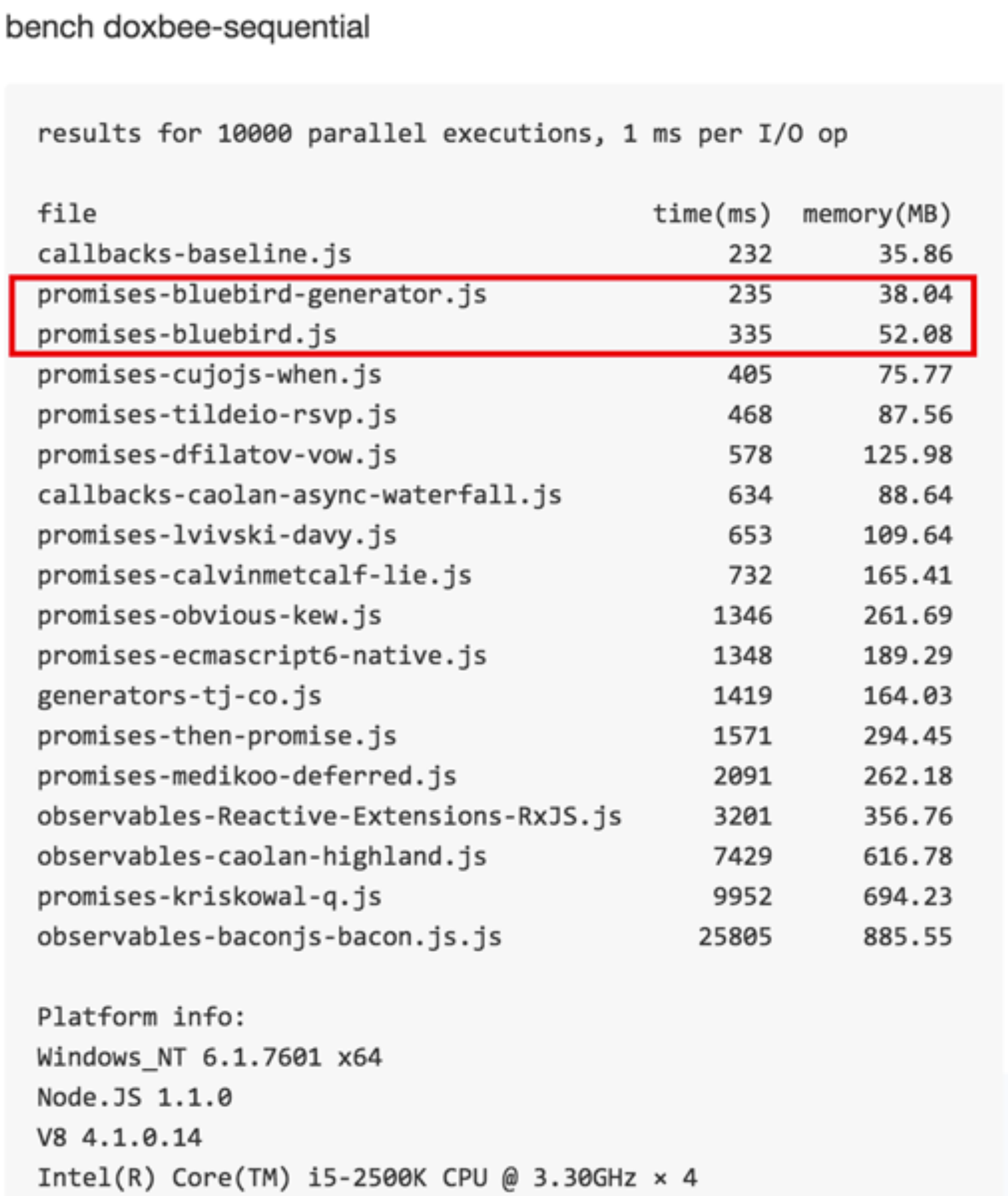
所以有无数实现

Node从4.0开始正式

支持es6规范里的Promise

为啥不用Node内置的?

基准测试

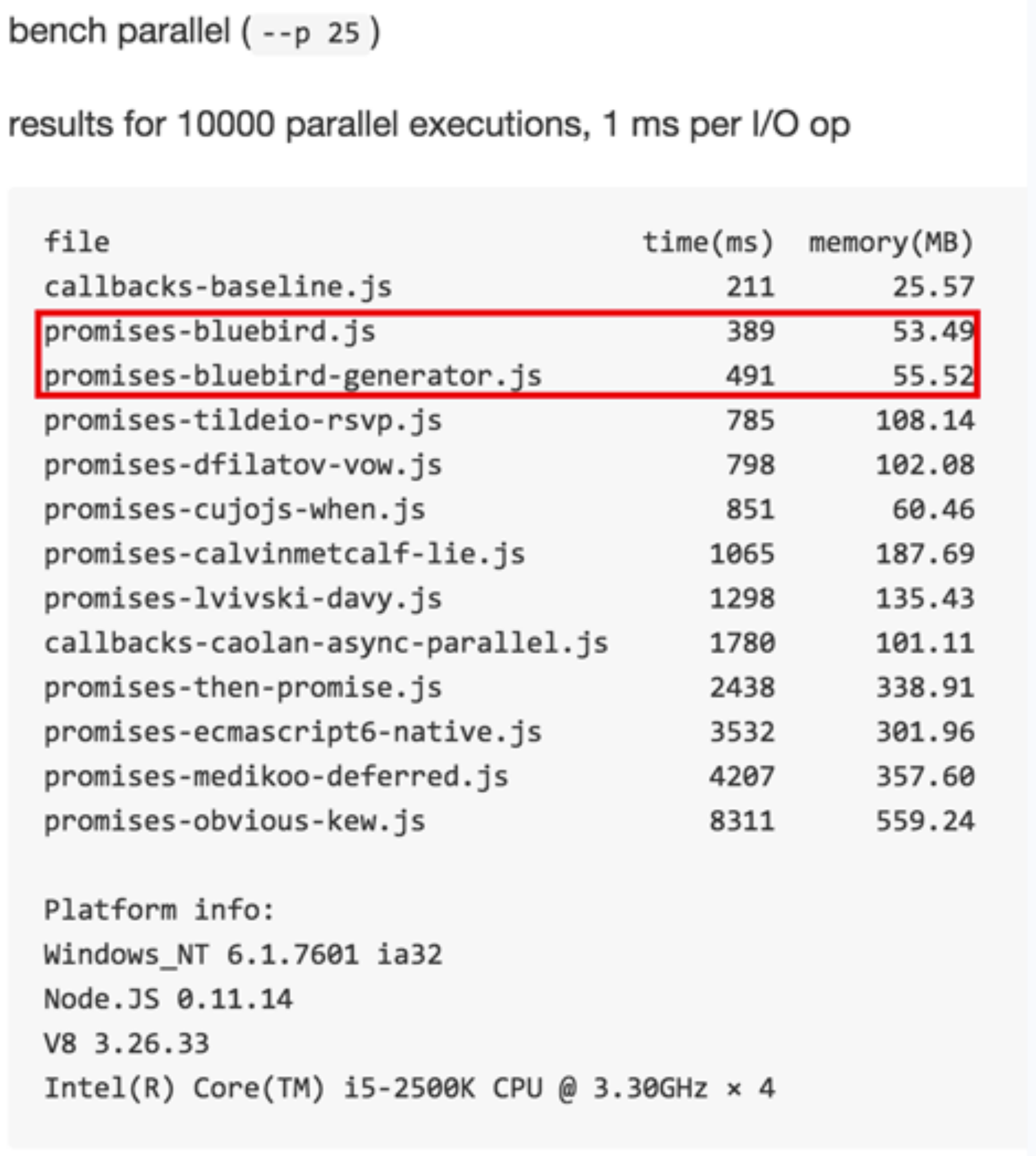


bench doxbee-sequential

```
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
```

bluebird

- 兼容性好
- 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

Promise的几种创建方法

- new Promise()实例化
- Promise.resolve和Promise.reject静态方法
- Promise.promisify（包裹，bluebird提供）
- Promise.promisifyAll （包裹所有，bluebird提供）

> Promisification means converting an existing promise-unaware API to a promise-returning API.

偷懒的利器

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

fs.readFileAsync("./package.json", "utf8").then(function(contents) {
    console.log(contents);
}).catch(function(e) {
    console.error(e.stack);
});
```

只有优点么？

```javascript
var Promise = require("bluebird");

var obj  = {
  a: function(){
    console.log('a')
  },

  b: function(){
    console.log('b')
  },

  c: function(){
    console.log('c')
  }
}

Promise.promisifyAll(obj);

obj.aAsync().then(obj.bAsync()).then(obj.cAsync()).catch(function(err){
  console.log(err)
})
```

生成器Generators

本应用于计算的，却因异步改进而知名

```
function* doSomething() {
    console.log('1');
    yield; // Line (A)
    console.log('2');
}

var gen1 = doSomething();

gen1.next(); // Prints 1 then pauses at line (A)
gen1.next(); // resumes execution at line (A), then prints 2
```

说明

- gen1是产生出来的Generator对象
- 第一个next，会打印出1，之后悬停在 yield所在行，即Line (A)
- 第二个next，恢复line (A)点的执行，之后打印出2

如果generator里有多个yield呢?

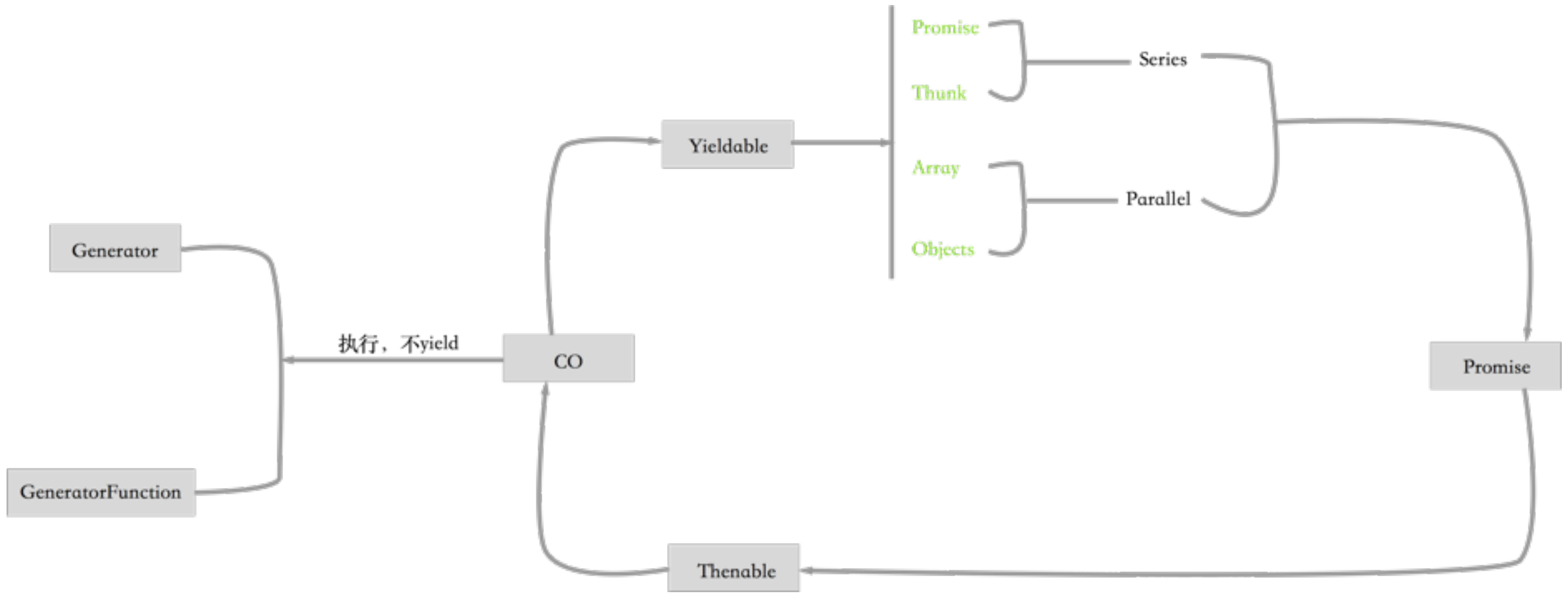
co是Generator的执行器

```
co(function* () {
  var result = yield Promise.resolve(true);
  return result;
}).then(function (value) {
  console.log(value);
}, function (err) {
  console.error(err.stack);
});
```

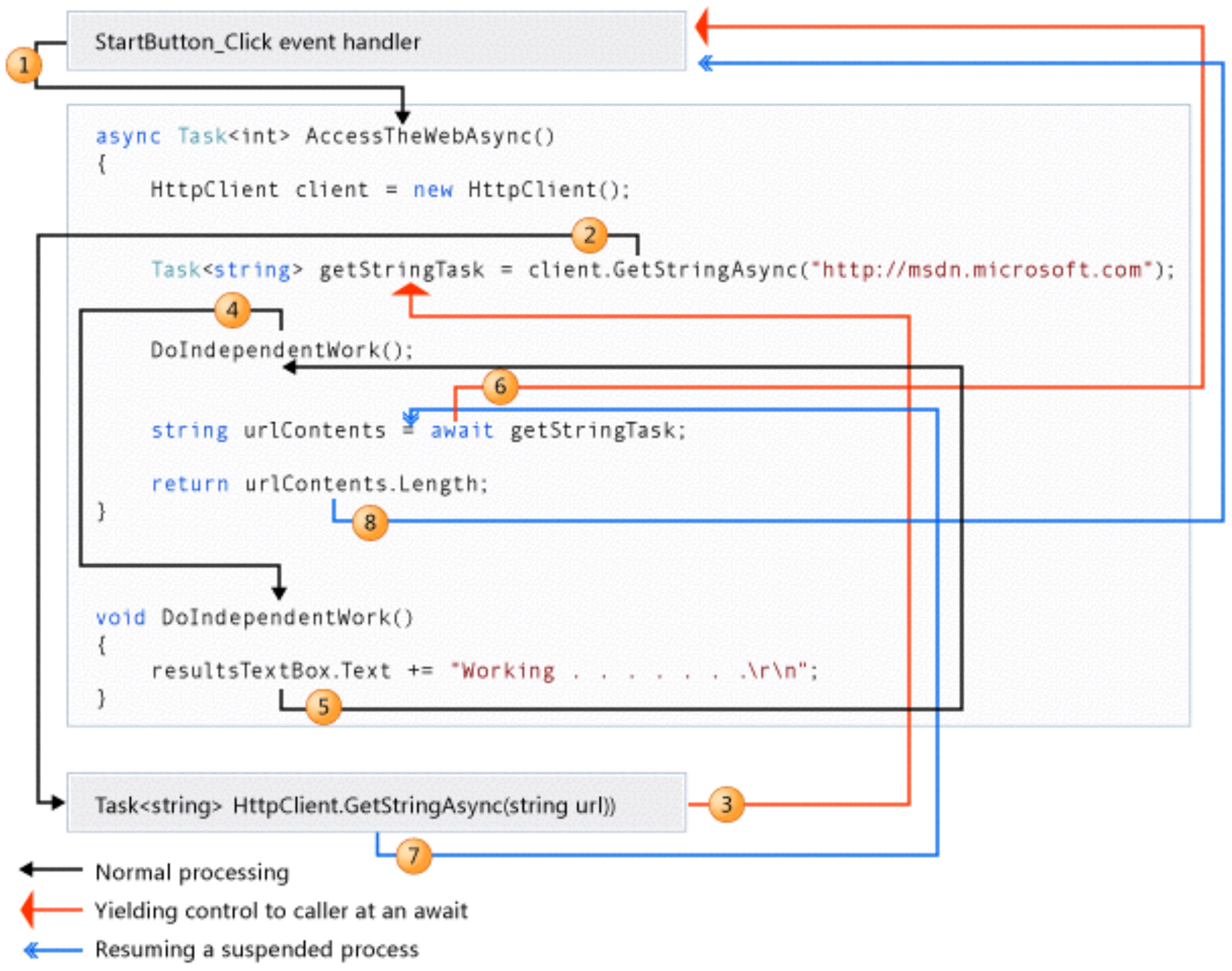
要点

- co是Generator执行器，你直需管好yield
- co的返回值是Promise

5种yieldable



Async函数


StartButton_Click event handler
Task<string> HttpClient.GetStringAsync(string url))
1
2
3
4
5
6
7
8
Normal processing
Yielding control to caller at an await
Resuming a suspended process

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}

void DoIndependentWork()
{
    resultsTextBox.Text += "Working . . . . . . .\r\n";
}
```

C#

示例

```
exports.list = async (ctx, next) => {
  try {
    let students = await Student.getAllAsync();

    await ctx.render('students/index', {
      students : students
    })
  } catch (err) {
    return ctx.api_error(err);
  }
};
```

Await的3种可能情况

- Await + Async函数
- Await + Promise
- await + co

```
async function a2() {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, 1000);
  })
}

async function a1() {
  console.log("hello a1 and start a2");
  await a2();
  console.log("hello end a2");
}

async function a0() {
  console.log("hello a0 and start a1");
  await a1();
  console.log("hello end a1");
}

a0()
```

异常处理

```
try {
  console.log(await asyncFn());
} catch (err) {
  console.error(err);
}
```

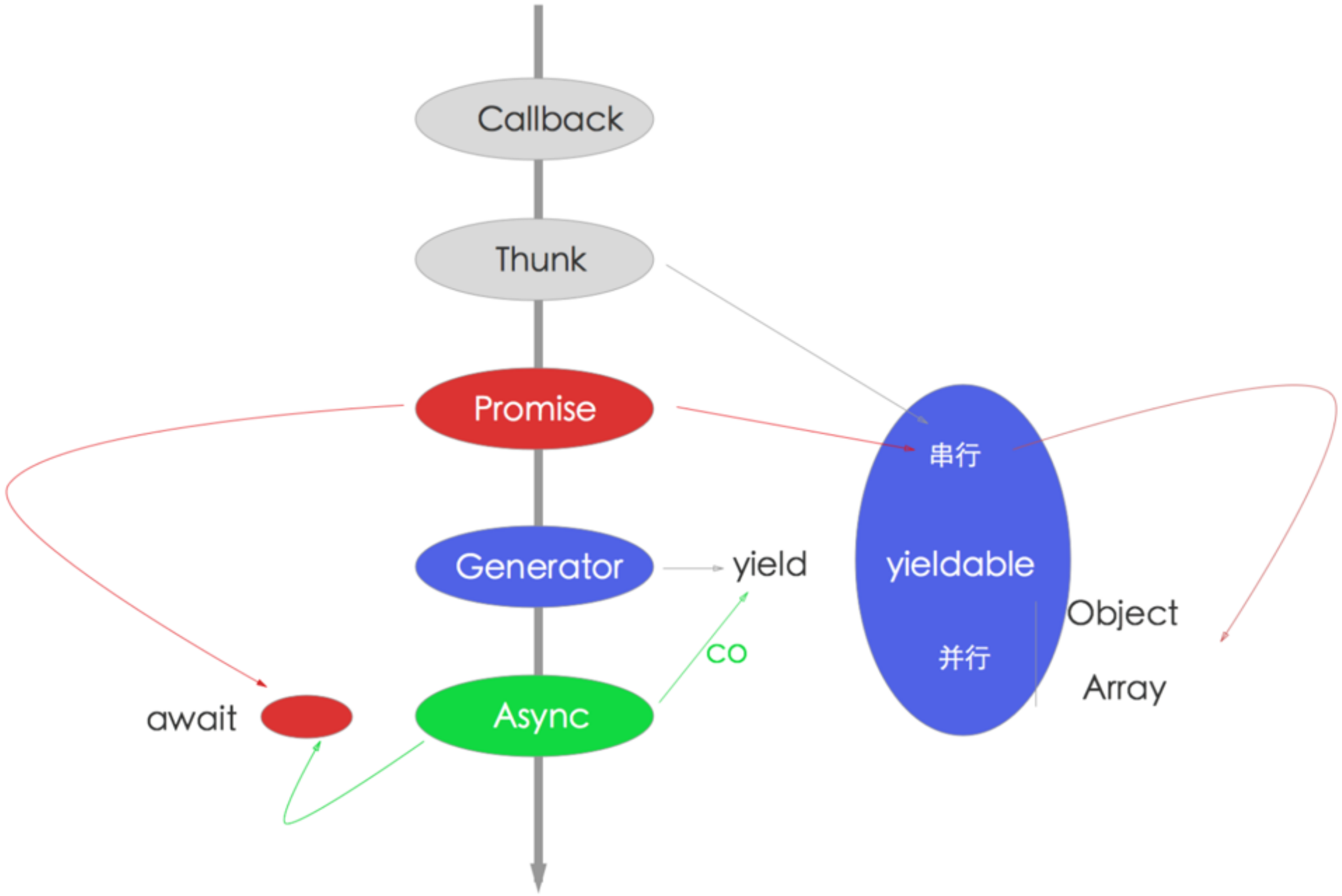
Promise里有2种处理异常的方法

- then(onFulfilled, onRejected)里的onRejected，处理当前Promise里的异常
- catch处理全局异常

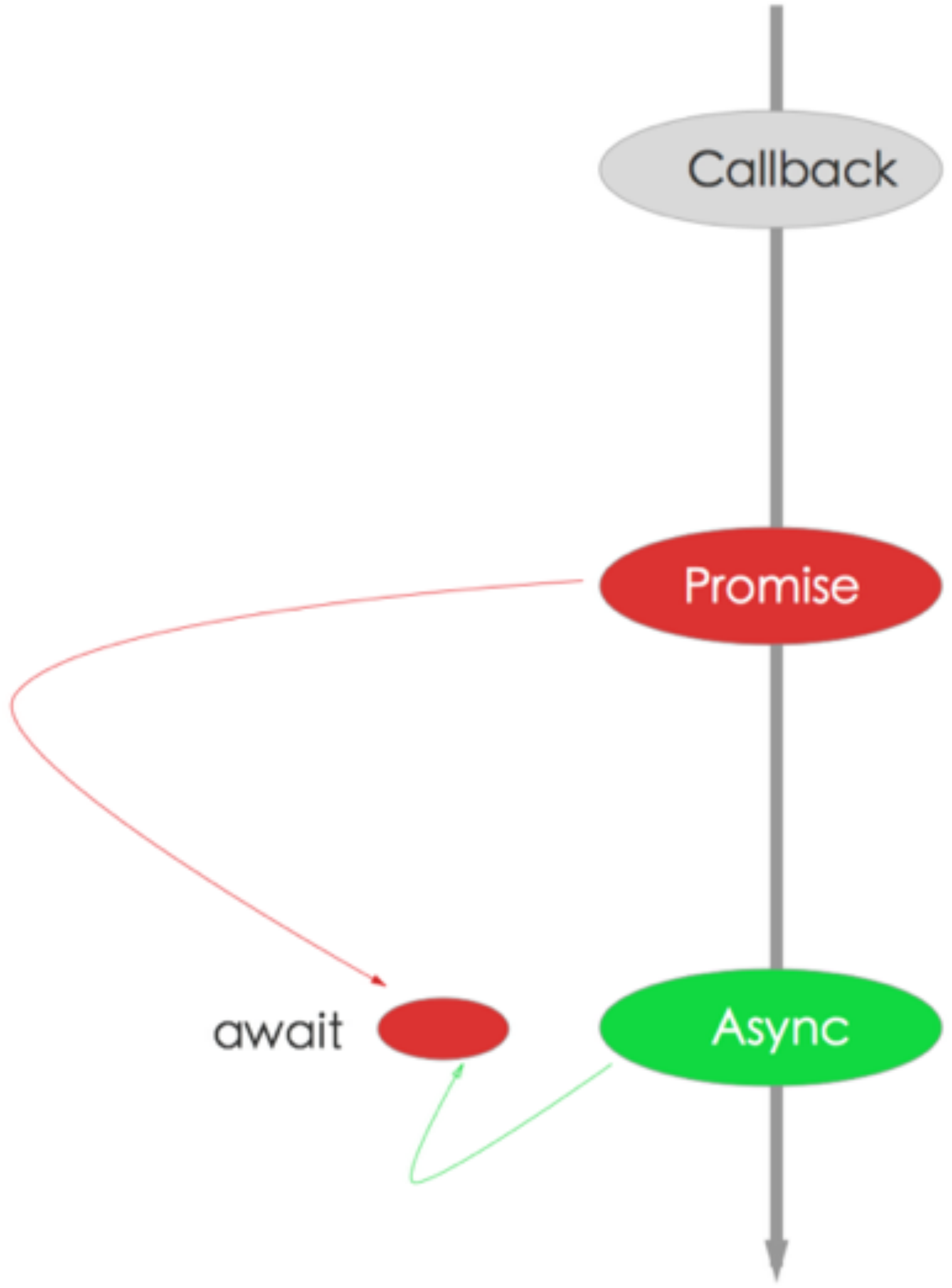
Async函数总结

- Async函数语义上非常好
- Async不需要执行器，它本身具备执行能力，不像Generator
- Async函数的异常处理采用try/catch和Promise的错误处理，非常强大
- Await接Promise，Promise自身就足够应对所有流程了
- Await释放Promise的组合能力，外加Promise的then，基本无敌

异步流程概览


Callback
Thunk
Promise
Generator
yield
Async
await
co
串行
yieldable
并行
Object
Array

学习重点



问题

- Async函数里，如何做到并行执行?

Q & A

少抱怨，多思考，未来更美好。有的时候我看的不是你一时的能力，而是你面对世界的态度。

console.log('The End, Thanks~')