

[命令行动画龟兔赛跑] Node 的语言基础 – JS (ES5/6/7/8)



本节目标：【实现一个龟兔赛跑小动画】 - 没有金刚钻，不揽瓷器活，必要的 JS 知识是学习 Node 的第一板斧。

名词概念解释

- JS 对 JavaScript 的简称，实际意义与 JavaScript 相同
- JavaScript 通用叫法，代指实现了 ECMAScript 标准的任何语言版本
- ES 对 ECMAScript 的简称，实际意义与 ECMAScript 相同
- ECMAScript [欧洲计算机制造商协会 \(http://www.ecma-international.org/\)](http://www.ecma-international.org/) 对 Javascript 语言制定的工业标准，随着时间推移版本不断更新，如 ES3/4/5
- ES2/3/4/5 是语言标准的版本，ES2015/ES2016 是发布标准的年份称谓，如 ES2015/ES6 意义等同
- ES.Next 代指下一个要推出的语言标准，它永远指向下一个，是动态的

1995 年网景浏览器发布 LiveScript，后来改名 JavaScript，再后来提交 JavaScript 到 ECMA 进行标准化，也就有了官方学名 ECMAScript，只是 ECMAScript 这个名字不怎么讨人喜爱，大家仍然习惯叫 JavaScript。

到如今已过去 20 多年了，按道理说，根据 JavaScript 语言进化的速度，到现在应该有至少十几个 ES 版本出来，然后并没有，原因是标准协会负责定标准，但实际执行标准实现（落地）的是浏览器厂商，每个厂商都夹带私货，对标准也不是全部认同，标准协会自己干活也不怎么接地气，甚至在 1999 年 ES3 ~ 2009 年 ES5 这个黄金十年是停滞不前的，并没有多少群众呼声大的特性加入标准，人生有多少个十年，也不能说它没干活，1999 年还是推出了 ES4，但太过激进这个版本就报废了，没有在厂商落地，直到 2009 年语言标准才有步入正轨，这个停滞的 10 年中标准协会以及厂商的撕逼内幕非常精彩，大家有兴趣可以去扒一扒。

Webpack 中用到的部分语法特性

在 Node 里面自 v6.14.4 之后，[就支持了 99% 以上的 ES6 语法特性 \(https://node.green/\)](https://node.green/)，在 v10.x 之后，也支持了几乎全部的 ES7/ES2016、ES8/ES2017、ES9/ES2018 的可用语法特性，我们主要关注 ES6/ES2015 中新增的语法特性和部分 ES7/8/9 的语法特性，在小册子我们会使用 Node v10.11.x 的版本来运行一些示例代码。

首先，给大家列一些 Webpack 中用到的语法特性，源码也都是从 Webpack 或者它依赖的模块中扒出来的，如果大家对这里面大部分特性看不懂的，建议参考如下资料系统学习荡平一下 Javascript 的入门门槛，再往下进入到我们的编码环节。

- [Mozilla 文档 \(https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_)
- [阮老师的 ES6 入门 \(http://es6.ruanyifeng.com/#docs/intro\)](http://es6.ruanyifeng.com/#docs/intro)

语法特性 示例

变量声明

`{let 具有`

块级域，
const 通
常声明不
可变量

```
let debugId = 1000  
const Compiler = require('./Compiler')
```

模板字
符串

{字符串
的简洁用
法，可内
嵌函数与
变量}

```
let message = `* ${m.identifier()}`  
const ma = `${a.message}`
```

箭头函数
{省略

function
的函数定
义}

```
const exportPlugins = (obj, mappings) => {}
```

解构赋值
{从目标
数组或对
象提取特
定值}

```
// 从 tapable 模块对象中提取 SyncHook 函数  
const {  
  SyncHook,  
} = require('tapable')
```

rest 参
数与扩展
运算符
{参数到
数组与数
组铺开为
参数，可
以看做互
逆}

```
// 多余参数转成一个数组 args[]  
function(name, ...args) {  
  // 把数组以 ... 扩展为参数  
  
  this.hooks[name.replace(/**/)].call(...args)  
}
```

Symbol

| | |
|--|---|
| 对象 {可生成 一个独一 无二的 值，永不 重复} | // 原始数据对象 // 接收参数（可选）来生成一个独一无二的值 const MAYBEEND = Symbol('maybeEnd') const WRITING = Symbol('writing') |
|--|---|

Set 对象

| | |
|--|--|
| {一种数 据结构， 可存任意 数据类 型，且保 证值唯 一} | // 可接收一个可迭代对象作为参数，构造一个新的 Set 对象 const queue = new Set(this.groupsIterable) // 可接收空参数生成一个 Set 对象 const chunksProcessed = new Set() |
|--|--|

Map 对

| | |
|---------------------------|---|
| 象 {哈希结 构的键值 对集合} | // 创建一个空集合，交给 fileTimestamps const fileTs = (compiler.fileTimestamps = new Map()) |
|---------------------------|---|

Promise

对象
{管理异
步状态的
对象，在
某个时刻
来回调返
回异步执
行结果}

```
// fs 模块读出文件内容并转成 JSON
// 把整个异步过程包装成一个 Promise 返回
return new Promise((resolve, reject) => {
  require('fs').readFile(filename, (err,
content) => {
    try {
      var update = JSON.parse(content)
    } catch (e) {
      return reject(e)
    }
    resolve(update)
  })
})
```

for ... of

循环

{遍历循

环所有成

员}

```
// 遍历拿到所有该模块的依赖项
for (const dependency of
module.dependencies) {}
```

除了上面这些，还有 Class/Async Function/Generator Function 等等许许多多的特性，不再一一列举。

编程练习 - 实现一个命令行龟兔赛跑动画

我们来基于上面的知识结果，可以借助 chalk-animation 模块，来实现一个在命令行的进度条动画 - 龟兔赛跑，先看下效果：



要实现这样一个效果，思路是很简单的，就是不断的去刷新当前的终端日志就可以了，所以本质上是把字符串按照我们计算的规则，每隔比如 1 秒钟，就刷新一下进度，至于龟兔，控制好它俩在的几个状

态就行。

首先是起始状态：

```
|.....»
```

其次是兔子超越乌龟，并且兔子停下的状态：

```
|.....乌龟.....兔子.....»
```

再次是乌龟超越兔子时的状态：

```
|.....兔子乌龟.....»
```

最后是乌龟越过终点的状态：

```
|.....兔子.....» 乌龟
```

那我们首先可以用很简陋的代码来实现这个逻辑，让兔子速度是乌龟的 3 倍，等待 2 秒后，它俩开始离开起点，每隔 150 毫秒，就计算下它俩走多远了，也就是速度 × 轮询的次数，比如 450 毫秒后，乌龟走过去了三米，兔子走过去了九米距离，那么第一次写的代码可能就是这样的比较挫的代码：

```
// 声明 2 个比赛队员
const rabbit = '兔子'
const turtle = '乌龟'
// 声明一坨变量，作为赛道起点终点字符串
const start = '|'
const end = '»'
// 赛道上一米一米的距离，用 . 表示
const pad = '.'
// 速度是 1 米/150 毫秒
const speed = 1
// 赛道一共有 50 米
```

```
const steps = 50
// 约定兔子在 42 米的时候停下
const stopAt = 42
// 判断兔子是否停下
let stoped = false
// 默认从 0 开始轮询
let t = 0
// 一个定时器的句柄而已
let timer

// 计算兔子距离终点
const getRabbitLastSteps = () => {
  return steps - t * speed - t * speed * 3
}

// 计算乌龟距离终点
const getTurtleLastSteps = () => {
  return steps - t * speed
}

// 计算龟兔间距
const getGapSteps = () => {
  return stopAt - t * speed
}

// 初始赛道状态
const checkRaceInitState = () => {
  return
  ` ${rabbit} ${turtle} ${start} ${pad.repeat(steps)} ${
    end} `
}

// 兔子领先时的赛道状态
```

```
const checkRaceState = () => {
  return `${start}${pad.repeat(t *
speed)}${turtle}${pad.repeat(t * speed *
3)}${rabbit}${pad.repeat(getRabbitLastSteps())}${
end}`
}
```

// 分情况计算赛道的实时状态

```
const checkBackRaceState = () => {
  if (getGapSteps() <= 0) {
    if (getTurtleLastSteps() === 0) {
      return
`${start}${pad.repeat(stopAt)}${rabbit}${pad.repe
at(steps - stopAt)}${end}${turtle}`
    } else {
      return
`${start}${pad.repeat(stopAt)}${rabbit}${pad.repe
at(t * speed -
stopAt)}${turtle}${pad.repeat(getTurtleLastSteps(
))}${end}`
    }
  } else {
    return `${start}${pad.repeat(t *
speed)}${turtle}${pad.repeat(getGapSteps())}${rab
bit}${pad.repeat(steps - stopAt)}${end}`
  }
}
```

// 等待时间，把定时器包装成一个 Promise

```
const wait = (sec) => new Promise(resolve =>
setTimeout(() => resolve(), sec))
```

// 可以支持特效刷新的命令行日志模块


```
const chalkWorker = require('chalk-animation')
const initState = checkRaceInitState()
const racing = chalkWorker.rainbow(initState)

const updateRaceTrack = (state) => {
  racing.replace(state)
}

const race = () => {
  timer = setInterval(() => {
    // 判断是否兔子停下
    if (!stoped) {
      if (getRabbitLastSteps() <= (steps -
stopAt)) {
        stoped = true
      }
    }

    if (stoped) {
      let state = checkBackRaceState()
      updateRaceTrack(state)

      if (getTurtleLastSteps() === 0) {
        // 乌龟过线后就停止定时器
        clearInterval(timer)
        return
      }
    } else {
      let state = checkRaceState()
      updateRaceTrack(state)
    }
  })

  t++
}
```

```
    }, 150);  
}  
  
// 等待 20 秒再开始启动比赛  
wait(2000).then(() => {  
    race()  
})
```

在这一坨代码里面，我们用到了箭头函数/Promise/const/let/模板字符串，可以让代码变得清爽一些，并且有了 Promise，可以避免过度的 callback 嵌套，通过这个代码是为了让大家了解到，Nodejs 首先是 JS，其次是 Node，而 JS 里面又有不同时代的 ES 标准，所以大家在学习 Node 之前，是需要花一些时间来了解下 JS 尤其是 ES 最新标准带来的新语法的，比如我们可以增加 class 来把上面的 race function 再抽象下：

```
const chalkWorker = require('chalk-animation')  
  
class Race extends Object {  
    constructor(props = {}) {  
        super(props)  
        ;[  
            ['rabbit', '兔子'],  
            ['turtle', '乌龟'],  
            ['turtleStep', 0],  
            ['rabbitStep', 0],  
            ['start', '|'],  
            ['end', '》'],  
            ['pad', '.'],  
            ['speed', 1],  
            ['steps', 50],  
            ['stopAt', 42]  
        ].forEach(elem => {
```

```

    const [key, value] = elem
    if (!(key in props)) {
      this[key] = value
    }
  })
}

```

```

getRaceTrack () {
  const {
    start,
    pad,
    turtle,
    turtleStep,
    rabbit,
    rabbitStep,
    steps,
    end
  } = this

```

```

    if (!turtleStep && !rabbitStep) {
      return
      `${turtle}${rabbit}${start}${pad.repeat(steps)}${
end}`
    }

```

```

    const [
      [minStr, min],
      [maxStr, max]
    ] = [
      [turtle, turtleStep],
      [rabbit, rabbitStep]
    ].sort((a, b) => a[1] - b[1])

```

```

    const prefix = `${pad.repeat((min || 1) -
1)} `
    const middle = `${pad.repeat(max - min)} `
    const suffix = `${pad.repeat(steps - max)} `

    const _start = `${start}${prefix}${minStr} `
    const _end = suffix ?
`${maxStr}${suffix}${end}` : `${end}${maxStr} `
    return `${_start}${middle}${_end} `
  }

  updateRaceTrack (state, racing) {
    racing.replace(state)
  }

  updateSteps () {
    if (this.turtleStep >= this.steps) return
    if (this.rabbitStep <= this.stopAt) {
      this.rabbitStep += 3 * this.speed
    }
    this.turtleStep += 1 * this.speed
  }

  race () {
    const initState = this.getRaceTrack()
    const racing = chalkWorker.rainbow(initState)
    let t = 0
    let timer = setInterval(() => {
      if (t <= 6) {
        t += 1
        return
      }
    })
    const state = this.getRaceTrack()

```

```
        this.updateRaceTrack(state, racing)
        this.updateSteps()
    }, 150)
}
}

const proxy = new Proxy(Race, {
  apply (target, ctx, args) {
    const race = new target(...args)
    return race.race()
  }
})

proxy()
```

除了 class，还增加了一些 rest 参数，解构赋值等，但依然实现过程比较猥琐且过度设计，大家能 Get 到 语法特性能带来一些编程实现方式的差异性 这一点，这一章节的目的就达到了。

思考

基于上面的代码，给大家布置一个作业，如果要实现龟兔赛跑，但可以让兔子停留的位置随机应该怎么实现？