第1题

```
let a = 1
function b(a) {
  a = 2
  console.log(a)
}
b(a)
console.log(a)
复制代码
```

答案

2、1

解析

首先基本类型数据是按值传递的，所以执行b函数时，b的参数a接收的值为1，参数a相当于函数内部的变量，当本作用域有和上层作用域同名的变量时，无法访问到上层变量，所以函数内无论怎么修改a，都不影响上层，所以函数内部打印的a是2，外面打印的仍是1。

第2题

```
function a (b = c, c = 1) {
  console.log(b, c)
}
a()
复制代码
```

答案

报错

解析

给函数多个参数设置默认值实际上跟按顺序定义变量一样，所以会存在暂时性死区的问题，即前面定义的变量不能引用后面还未定义的变量，而后面的可以访问前面的。

第3题

```
let a = b = 10
;(function(){
  let a = b = 20
})()
console.log(a)
console.log(b)
复制代码
```

答案

10、20

解析

连等操作是从右向左执行的，相当于b = 10、let a = b，很明显b没有声明就直接赋值了，所以会隐式创建为一个全局变量，函数内的也是一样，并没有声明b，直接就对b赋值了，因为作用域链，会一层一层向上查找，找了到全局的b，所以全局的b就被修改为20了，而函数内的a因为重新声明了，所以只是局部变量，不影响全局的a，所以a还是10。

第4题

```
var a = {n:1}
var b = a
a.x = a = {n:2}
console.log(a.x)
console.log(b.x)
复制代码
```

答案

undefined、{n: 2}

解析

恕笔者不才，这道题笔者做一次错一次。

反正按照网上大部分的解释是因为.运算符优先级最高，所以会先执行a.x，此时a、b共同指向的{n: 1}变成了{n: 1, x: undefined}，然后按照连等操作从右到左执行代码，a = {n: 2}，显然，a现在指向了一个新对象，然后a.x = a，因为a.x最开始就执行过了，所以这里其实等价于：({n: 1, x: undefined}).x = b.x = a = {n: 2}。

第5题

```
var arr = [0, 1, 2]
arr[10] = 10
console.log(arr.filter(function (x) {
  return x === undefined
}))
复制代码
```

答案

[]

解析

这题比较简单，arr[10]=10，那么索引3到9位置上都是undefined，arr[3]等打印出来也确实是undefined，但是，这里其实涉及到ECMAScript版本不同对应方法行为不同的问题，ES6之前的遍历方法都会跳过数组未赋值过的位置，也就是空位，但是ES6新增的for of方法就不会跳过。

第6题

```
var name = 'World'
;(function () {
  if (typeof name === 'undefined') {
    var name = "Jack"
    console.info('Goodbye ' + name)
  } else {
    console.info('Hello ' + name)
  }
})()
```
复制代码

答案

Goodbye Jack

解析

这道题考察的是变量提升的问题，var声明变量时会把变量自动提升到当前作用域顶部，所以函数内的name虽然是在if分支里声明的，但是也会提升到外层，因为和全局的变量name重名，所以访问不到外层的name，最后因为已声明未赋值的变量的值都为undefined，导致if的第一个分支满足条件。

第7题

```
console.log(1 + NaN)
console.log("1" + 3)
console.log(1 + undefined)
console.log(1 + null)
console.log(1 + {})
console.log(1 + [])
console.log([] + {})

```
复制代码

答案

NaN、13、NaN、1、1[object Object]、1、[object Object]

解析

这道题考察的显然是+号的行为：

1.如果有一个操作数是字符串，那么把另一个操作数转成字符串执行连接

2.如果有一个操作数是对象，那么调用对象的valueOf方法转成原始值，如果没有该方法或调用后仍是非原始值，则调用toString方法

3.其他情况下，两个操作数都会被转成数字执行加法操作

第8题

```
var a={},
    b={key:'b'},
    c={key:'c'}
a[b]=123
a[c]=456
console.log(a[b])
复制代码
```

答案

456

解析

对象有两种方法设置和引用属性，obj.name和obj['name']，方括号里可以字符串、数字和变量设置是表达式等，但是最终计算出来得是一个字符串，对于上面的b和c，它们两个都是对象，所以会调用toString()方法转成字符串，对象转成字符串和数组不一样，和内容无关，结果都是[object Obejct]，所以a[b]=a[c]=a['[object Object]']。

第9题

```
var out = 25
var inner = {
  out: 20,
  func: function () {
    var out = 30
    return this.out
  }
};
console.log((inner.func, inner.func)())
console.log(inner.func())
console.log((inner.func)())
console.log((inner.func = inner.func)())
复制代码
```

答案

25、20、20、25

解析

这道题考察的是this指向问题：

1.逗号操作符会返回表达式中的最后一个值，这里为inner.func对应的函数，注意是函数本身，然后执行该函数，该函数并不是通过对象的方法调用，而是在全局环境下调用，所以this指向window，打印出来的当然是window下的out

2.这个显然是以对象的方法调用，那么this指向该对象

3.加了个括号，看起来有点迷惑人，但实际上(inner.func)和inner.func是完全相等的，所以还是作为对象的方法调用

4.赋值表达式和逗号表达式相似，都是返回的值本身，所以也相对于在全局环境下调用函数

第10题

```
let {a,b,c} = { c:3, b:2, a:1 }
console.log(a, b, c)
```
复制代码

答案

1、2、3

解析

这题考察的是变量解构赋值的问题，数组解构赋值是按位置对应的，而对象只要变量与属性同名，顺序随意。

第11题

```
console.log(Object.assign([1, 2, 3], [4, 5]))
```
复制代码

答案

[4, 5, 3]

解析

是不是从来没有用assign方法合并过数组？assign方法可以用于处理数组，不过会把数组视为对象，比如这里会把目标数组视为是属性为0、1、2的对象，所以源数组的0、1属性的值覆盖了目标对象的值。

第12题

```
var x=1
switch(x++)
{
  case 0: ++x
  case 1: ++x
  case 2: ++x
}
console.log(x)
```
复制代码

答案

4

解析

这题考查的是自增运算符的前缀版和后缀版，以及switch的语法，后缀版的自增运算符会在语句被求值后才发生，所以x会仍以1的值去匹配case分支，那么显然匹配到为1的分支，此时，x++生效，x变成2，再执行++x，变成3，因为没有break语句，所以会进入当前case后面的分支，所以再次++x，最终变成4。

第13题

```
console.log(typeof undefined == typeof NULL)
console.log(typeof function () {} == typeof class {})
复制代码
```

答案

true、true

解析

1.首先不要把NULL看成是null，js的关键字是区分大小写的，所以这就是一个普通的变量，而且没有声明，typeof对没有声明的变量使用是不会报错的，返回'undefined'，typeof对undefined使用也是'undefined'，所以两者相等

2.typeof对函数使用返回'function'，class只是es6新增的语法糖，本质上还是函数，所以两者相等

第14题

```
var count = 0
console.log(typeof count === "number")
console.log(!!typeof count === "number")
复制代码
```

答案

true、false

解析

1.没啥好说的，typeof对数字类型返回'number'。

2.这题考查的是运算符优先级的问题，逻辑非!的优先级比全等===高，所以先执行!!typeof count，结果为true，然后执行true === 'number'，结果当然为false，可以点击这里查看优先级列表：**点我**[1]。

第15题

```
"use strict"
a = 1
var a = 2
console.log(window.a)
console.log(a)
复制代码
```

答案

2、2

解析

var声明会把变量提升到当前作用域顶部，所以a=1并不会报错，另外在全局作用域下使用var声明变量，该变量会变成window的一个属性，以上两点都和是否在严格模式下无关。

第16题

```
var i = 1
function b() {
  console.log(i)
}
function a() {
  var i = 2
  b()
}
a()
复制代码
```

答案

1

解析

这道题考察的是作用域的问题，作用域其实就是一套变量的查找规则，每个函数在执行时都会创建一个执行上下文，其中会关联一个变量对象，也就是它的作用域，上面保存着该函数能访问的所有变量，另外上下文中的代码在执行时还会创建一个作用域链，如果某个标识符在当前作用域中没有找到，会沿着外层作用域继续查找，直到最顶端的全局作用域，因为js是词法作用域，在写代码阶段就作用域就已经确定了，换句话说，是在函数定义的时候确定的，而不是执行的时候，所以a函数是在全局作用域中定义的，虽然在b函数内调用，但是它只能访问到全局的作用域而不能访问到b函数的作用域。

第17题

```
var obj = {
  name: 'abc',
  fn: () => {
    console.log(this.name)
  }
};
obj.name = 'bcd'
obj.fn()
复制代码
```

答案

undefined

解析

这道题考察的是this的指向问题，箭头函数执行的时候上下文是不会绑定this的，所以它里面的this取决于外层的this，这里函数执行的时候外层是全局作用域，所以this指向window，window对象下没有name属性，所以是undefined。

第18题

```
const obj = {
  a: {
    a: 1
  }
};
const obj1 = {
  a: {
    b: 1
  }
};
console.log(Object.assign(obj, obj1))
复制代码
```

答案

{a: {b: 1}}

解析

这道题很简单，因为assign方法执行的是浅拷贝，所以源对象的a属性会直接覆盖目标对象的a属性。

第19题

```
console.log(a)
var a = 1
var getNum = function() {
  a = 2
}
function getNum() {
  a = 3
}
console.log(a)
getNum()
console.log(a)
复制代码
```

答案

undefined、1、2

解析

首先因为var声明的变量提升作用，所以a变量被提升到顶部，未赋值，所以第一个打印出来的是undefined。

接下来是函数声明和函数表达式的区别，函数声明会有提升作用，在代码执行前就把函数提升到顶部，在执行上下文上中生成函数定义，所以第二个getNum会被最先提升到顶部，然后是var声明getNum的提升，但是因为getNum函数已经被声明了，所以就不需要再声明一个同名变量，接下来开始执行代码，执行到var getNum = fun...时，虽然声明被提前了，但是赋值操作还是留在这里，所以getNum被赋值为了一个函数，下面的函数声明直接跳过，最后，getNum函数执行前a打印出来还是1，执行后，a被修改成了2，所以最后打印出来的2。

第20题

```
var scope = 'global scope'
function a(){
  function b(){
    console.log(scope)
  }
  return b
  var scope = 'local scope'
}
a()()
复制代码
```

答案

undefined

解析

这题考查的还是变量提升和作用域的问题，虽然var声明是在return语句后面，但还是会提升到a函数作用域的顶部，然后又因为作用域是在函数定义的时候确定的，与调用位置无关，所以b的上层作用域是a函数，scope在b自身的作用域里没有找到，向上查找找到了自动提升的并且未赋值的scope变量，所以打印出undefined。

第21题

```
function fn (){
  console.log(this)
}
var arr = [fn]
arr[0]()
复制代码
```

答案

打印出arr数组本身

解析

函数作为某个对象的方法调用，this指向该对象，数组显然也是对象，只不过我们都习惯了对象引用属性的方法：obj.fn，但是实际上obj['fn']引用也是可以的。

第22题

```
var a = 1
function a(){}
console.log(a)

var b
function b(){}
console.log(b)

function b(){}
var b
console.log(b)
复制代码
```

答案

1、b函数本身、b函数本身

解析

这三小题都涉及到函数声明和var声明，这两者都会发生提升，但是函数会优先提升，所以如果变量和函数同名的话，变量的提升就忽略了。

1.提升完后，执行到赋值代码，a被赋值成了1，函数因为已经声明提升了，所以跳过，最后打印a就是1。

2.和第一题类似，只是b没有赋值操作，那么执行到这两行相当于都没有操作，b当然是函数。

3.和第二题类似，只是先后顺序换了一下，但是并不影响两者的提升顺序，仍是函数优先，同名的var声明提升忽略，所以打印出b还是函数。

第23题

```
function Foo() {
  getName = function () { console.log(1) }
  return this
}
Foo.getName = function () { console.log(2) }
Foo.prototype.getName = function () { console.log(3) }
var getName = function () { console.log(4) }
function getName() { console.log(5) }

//请写出以下输出结果:
Foo.getName()
getName()
Foo().getName()
getName()
new Foo.getName()
new Foo().getName()
new new Foo().getName()
复制代码
```

答案

2、4、1、1、2、3、3

解析

这是一道综合性题目，首先getName函数声明会先提升，然后getName函数表达式提升，但是因为函数声明提升在线，所以忽略函数表达式的提升，然后开始执行代码，执行到var getName= ...时，修改了getName的值，赋值成了打印4的新函数。

1.执行Foo函数的静态方法，打印出2。

2.执行getName，当前getName是打印出4的那个函数。

3.执行Foo函数，修改了全局变量getName，赋值成了打印1的函数，然后返回this，因为是在全局环境下执行，所以this指向window，因为getName已经被修改了，所以打印出1。

4.因为getName没有被重新赋值，所以再执行仍然打印出1。

5.new操作符是用来调用函数的，所以new Foo.getName()相当于new (Foo.getName)()，所以new的是Foo的静态方法getName，打印出2。

6.因为点运算符（.）的优先级和new是一样高的，所以从左往右执行，相当于(new Foo()).getName()，对Foo使用new调用会返回一个新创建的对象，然后执行该对象的getName方法，该对象本身并没有该方法，所以会从Foo的原型对象上查找，找到了，所以打印出3。

7.和上题一样，点运算符（.）的优先级和new一样高，另外new是用来调用函数的，所以new new Foo().getName()相当于new ((new Foo()).getName)()，括号里面的就是上一题，所以最后找到的是Foo原型上的方法，无论是直接调用，还是通过new调用，都会执行该方法，所以打印出3。

第24题

```
const person = {
 address: {
  country:"china",
  city:"hangzhou"
 },
 say: function () {
  console.log(`it's ${this.name}, from ${this.address.country}`)
 },
 setCountry:function (country) {
  this.address.country=country
 }
}

const p1 = Object.create(person)
const p2 = Object.create(person)

p1.name = "Matthew"
p1.setCountry("American")

p2.name = "Bob"
```

```
p2.setCountry("England")

p1.say()
p2.say()
```
复制代码

答案

it's Matthew, from England

it's Bob, from England

解析

Object.create方法会创建一个对象，并且将该对象的**proto**属性指向传入的对象，所以p1和p2两个对象的原型对象指向了同一个对象，接着给p1添加了一个name属性，然后调用了p1的setCountry方法，p1本身是没有这个方法的，所以会沿着原型链进行查找，在它的原型上，也就是person对象上找到了这个方法，执行这个方法会给address对象的country属性设置传入的值，p1本身也是没有address属性的，但是和name属性不一样，address属性在原型对象上找到了，并且因为是个引用值，所以会成功修改它的country属性，接着对p2的操作也是一样，然后因为原型中存在引用值会在所有实例中共享，所以p1和p2它们引用的address也是同一个对象，一个实例修改了，会反映到所有实例上，所以p2的修改会覆盖p1的修改，最终country的值为England。

第25题

```
setTimeout(function() {
  console.log(1)
}, 0)
new Promise(function(resolve) {
  console.log(2)
  for( var i=0 ; i<10000 ; i++ ) {
    i == 9999 && resolve()
  }
  console.log(3)
}).then(function() {
  console.log(4)
})
console.log(5)
```
复制代码

答案

2、3、5、4、1

解析

这道题显然考察的是事件循环的知识点。

js是一门单线程的语言，但是为了执行一些异步任务时不阻塞代码，以及避免等待期间的资源浪费，js存在事件循环的机制，单线程指的是执行js的线程，称作主线程，其他还有一些比如网络请求的线程、定时器的线程，主线程在运行时会产生执行栈，栈中的代码如果调用了异步api的话则会把事件添加到事件队列里，只要该异步任务有了结果便会把对应的回调放到【任务队列】里，当执行栈中的代码执行完毕后会去读取任务队列里的任务，放到主线

程执行，当执行栈空了又会去检查，如此往复，也就是所谓的事件循环。

异步任务又分为【宏任务】（比如setTimeout、setInterval）和【微任务】（比如promise），它们分别会进入不同的队列，执行栈为空完后会优先检查微任务队列，如果有微任务的话会一次性执行完所有的微任务，然后去宏任务队列里检查，如果有则取出一个任务到主线程执行，执行完后又会去检查微任务队列，如此循环。

回到这题，首先整体代码作为一个宏任务开始执行，遇到setTimeout，相应回调会进入宏任务队列，然后是promise，promise的回调是同步代码，所以会打印出2，for循环结束后调用了resolve，所以then的回调会被放入微任务队列，然后打印出3，最后打印出5，到这里当前的执行栈就空了，那么先检查微任务队列，发现有一个任务，那么取出来放到主线程执行，打印出4，最后检查宏任务队列，把定时器的回调放入主线程执行，打印出1。

第26题

```
console.log('1');

setTimeout(function() {
  console.log('2');
  process.nextTick(function() {
    console.log('3');
  });
  new Promise(function(resolve) {
    console.log('4');
    resolve();
  }).then(function() {
    console.log('5');
  });
});

process.nextTick(function() {
  console.log('6');
});

new Promise(function(resolve) {
  console.log('7');
  resolve();
}).then(function() {
  console.log('8');
});

setTimeout(function() {
  console.log('9');
  process.nextTick(function() {
    console.log('10');
  })
  new Promise(function(resolve) {
    console.log('11');
    resolve();
  }).then(function() {
    console.log('12')
```

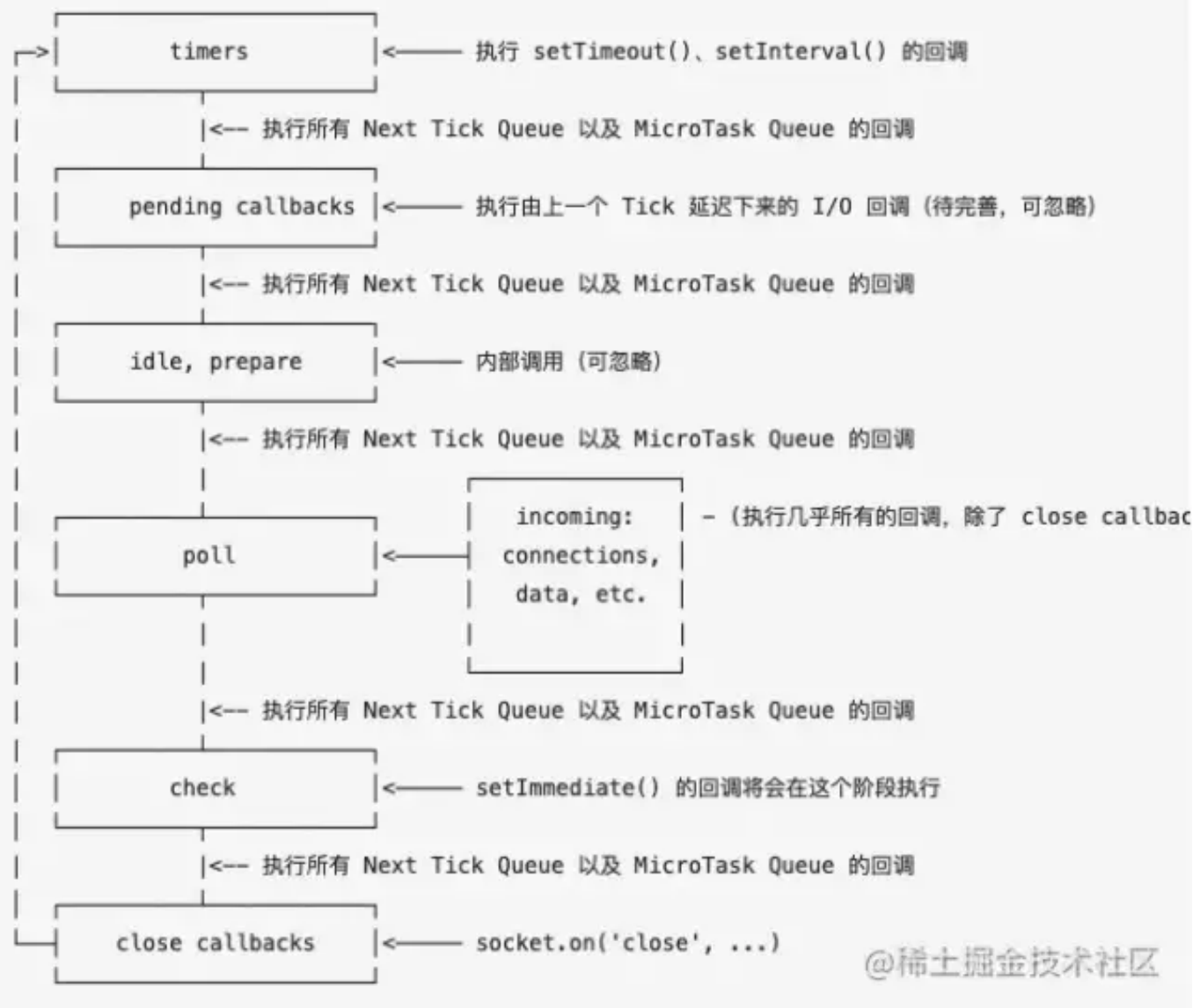
```
  });
})
```
复制代码

答案

1、7、6、8、2、4、9、11、3、10、5、12

解析

这道题和上一题差不多，但是出现了process.nextTick，所以显然是在node环境下，node也存在事件循环的概念，但是和浏览器的有点不一样，nodejs中的宏任务被分成了几种不同的阶段，两个定时器属于timers阶段，setImmediate属于check阶段，socket的关闭事件属于close callbacks阶段，其他所有的宏任务都属于poll阶段，除此之外，只要执行到前面说的某个阶段，那么会执行完该阶段所有的任务，这一点和浏览器不一样，浏览器是每次取一个宏任务出来执行，执行完后就跑去检查微任务队列了，但是nodejs是来都来了，一次全部执行完该阶段的任务好了，那么process.nextTick和微任务在什么阶段执行呢，在前面说的每个阶段的后面都会执行，但是process.nextTick会优先于微任务，一图胜千言：



理解了以后再来分析这道题就很简单了，首先执行整体代码，先打印出1，setTimeout回调扔进timers队列，nextTick的扔进nextTick的队列，promise的回调是同步代码，执行后打印出7，then回调扔进微任务队列，然后又是一个setTimeout回调扔进timers队列，到这里当前节点就结束了，检查nextTick和微任务队列，nextTick队列有任务，执行后打印出6，微任务队列也有，打印出8，接下来按顺序检查各个阶段，check队列、close callbacks队列都没有任务，到了timers阶段，发现有两个任务，先执行第一个，打印出2，然后nextTick的扔进nextTick的队列，执行promise打印出4，then回调扔进微任务队列，再执行第二个setTimeout的回调，打印出9，然后和刚才一样，nextTick的扔进nextTick的队列，执行promise打印出11，then回调扔进微任务队列，到这里timers阶段也结束了，执行nextTick队列的任务，发现又两个任务，依次执行，打印出3和10，然后检查微任务队列，也是两个任务，依次执行，打印出5和12，到这里是有队列都清空了。