期中测试题（上）

1. 关于 ES6 Proxy（2018 今日头条）

已知如下对象，请基于es6的proxy方法设计一个属性拦截读取操作的例子，要求实现去访问目标对象example中不存在的属性时，抛出错误：Property “$(property)” does not exist（2018 今日头条）

```
// 案例代码
const man = {
  name:'jscoder',
  age:22
}
//补全代码
const proxy = new Proxy(...)

proxy.name //"jscoder"
proxy.age //22

proxy.location //Property "$(property)" does not exist
```

1.1. 知识点

- ECMAScript 6 Proxy
- throw 语句

1.2. 参考答案

```
// 案例代码
const man = {
  name:'jscoder',
  age:22
}

//补全代码
const proxy = new Proxy(man, {
  // 参数1：源对象
  // 参数2：被访问的属性名
  get (target, property) {
    if (!target[property]) {
      throw new Error(`Property "${property}" does not exist`)
    }
    return target[property]
  }
})

console.log(proxy.name) // "jscoder"
console.log(proxy.age) // 22
```

```
console.log(proxy.location) // Property "$(property)" does not exist
```

2. 简答 (字节跳动 二面)

- 你觉得typescript和javascript有什么区别
- typescript你都用过哪些类型
- typescript中type和interface的区别

2.1. 知识点

- TypeScript

2.2. 参考答案

你觉得typescript和javascript有什么区别

- 语言层面：
  - JavaScript 和 TypeScript 都是 ECMAScript 的具体实现
  - TypeScript 是静态类型，而 JavaScript 是动态类型
  - TypeScript 扩展了 JavaScript 并且完全包容 JavaScript
- 执行层面：
  - TypeScript 需要编译
  - JavaScript 不需要编译（除非需要兼容低版本运行环境）
- 厂商层面：
  - JavaScript 由 Netscape 率先推出，现在主要由各大浏览器厂商实现
  - 而 TypeScript 目前由微软进行设计和维护

typescript你都用过哪些类型

- 布尔类型
- 数字类型
- 字符串类型
- 数组类型
- 元组类型
- 枚举类型
- Any 类型
- Void 类型
- 联合类型
- 对象的类型：接口
- 函数类型
- 泛型
- ...

typescript中 type 和 interface 的区别

- interface 是类型
- type 是类型别名，不是真正的类型

参见：

- https://www.typescriptlang.org/docs/handbook/advanced-types.html#type-aliases
- https://www.typescriptlang.org/docs/handbook/advanced-types.html#interfaces-vs-type-aliases

关于类型别名：

- 类型别名为类型创建新名称。类型别名有时与接口相似，但是可以命名原始类型，联合类型，元组以及其他必须手工编写的其他类型。
- 别名实际上并不会创建新类型，而是会创建一个新名称来引用该类型。虽然可以将原始类型用作文档的一种形式，但它并不是非常有用。
- 就像接口一样，类型别名也可以是通用的-我们可以添加类型参数并在别名声明的右侧使用它们：

interface 和 type 都可以实现 extends 扩展。

Interface

Extending an interface

```
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}

const bear = getBear()
bear.name
bear.honey
```

Type

Extending a type via intersections

```
type Animal = {
  name: string
}

type Bear = Animal & {
  honey: Boolean
}

const bear = getBear();
bear.name;
bear.honey;
```

Type 别名不能给已有类型添加新属性。

Adding new fields to an existing interface

```
interface Window {
  title: string
}

interface Window {
  ts: import("typescript")
}

const src = 'const a = "Hello World"';
window.ts.transpileModule(src, {});
```

A type cannot be changed after being created

```
type Window = {
  title: string
}

type Window = {
  ts: import("typescript")
}

// Error: Duplicate identifier 'Window'.
```

总结：

- 由于 interface 通过开放扩展更紧密地映射 JavaScript 对象的工作方式，因此我们建议尽可能在类型别名上使用接口。
- 另一方面，如果您无法使用 interface 表达某种形状，而需要使用联合类型或元组类型，则通常使用类型别名。

3. async/await 如果右边方法执行出错该怎么办？（百度一面 2020）

3.1. 知识点

- async 函数
- async 函数的异常处理

3.2. 参考答案

如果`await`后面的异步操作出错，那么等同于`async`函数返回的 Promise 对象被`reject`。

```
async function f() {
  await new Promise(function (resolve, reject) {
    throw new Error('出错了');
  });
}

f()
.then(v => console.log(v))
.catch(e => console.log(e))
// Error：出错了
```

上面代码中，`async`函数`f`执行后，`await`后面的 Promise 对象会抛出一个错误对象，导致`catch`方法的回调函数被调用，它的参数就是抛出的错误对象。具体的执行机制，可以参考后文的“async 函数的实现原理”。

防止出错的方法，也是将其放在`try...catch`代码块之中。

```
async function f() {
  try {
    await new Promise(function (resolve, reject) {
      throw new Error('出错了');
    });
  } catch(e) {
  }
  return await('hello world');
}
```

如果有多个`await`命令，可以统一放在`try...catch`结构中。

```
async function main() {
  try {
    const val1 = await firstStep();
    const val2 = await secondStep(val1);
    const val3 = await thirdStep(val1, val2);

    console.log('Final: ', val3);
  }
  catch (err) {
    console.error(err);
  }
}
```

下面的例子使用`try...catch`结构，实现多次重复尝试。

```
const superagent = require('superagent');
const NUM_RETRIES = 3;

async function test() {
  let i;
  for (i = 0; i < NUM_RETRIES; ++i) {
    try {
      await superagent.get('http://google.com/this-throws-an-error');
      break;
    } catch(err) {}
  }
  console.log(i); // 3
}

test();
```

上面代码中，如果`await`操作成功，就会使用`break`语句退出循环；如果失败，会被`catch`语句捕捉，然后进入下一轮循环。

4. 说一下 event loop 的过程？promise 定义时传入的函数什么时候执行？（小米 三面）

4.1. 知识点

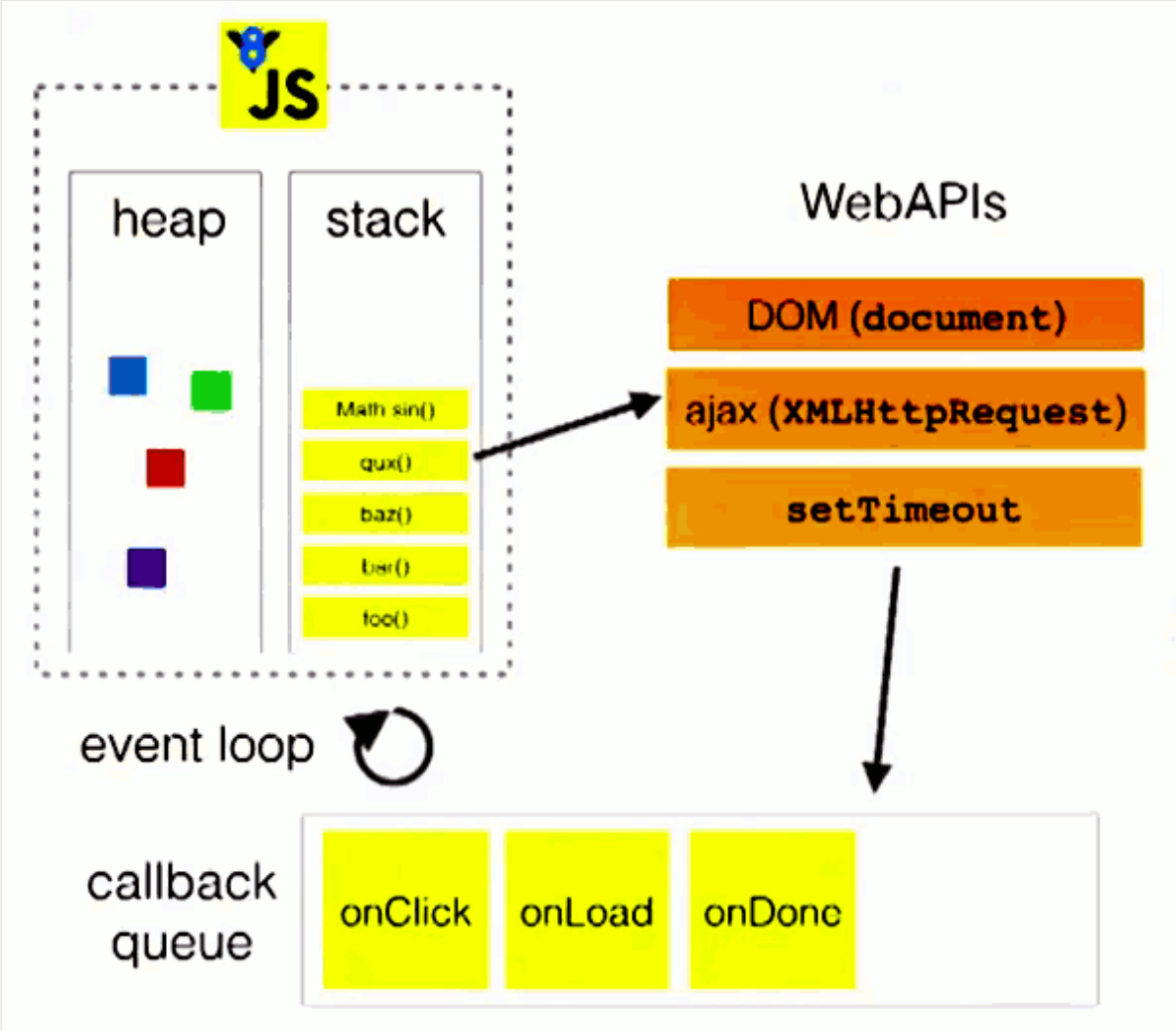
- JavaScript 运行机制
- 宏任务、微任务
- Promise

4.2. 参考答案

说一下 event loop 的过程?

> 建议参考：http://www.ruanyifeng.com/blog/2014/10/event-loop.html

- JavaScript 是单线程的，同一个时间只能做一件事
- 任务队列
- 事件和回调函数
- Event Loop



promise **定义时**传入的函数什么时候执行?

- 立即执行
- `new Promise` 在实例化的过程中所执行的代码都是同步进行的，而 `then` 中注册的回调才是异步执行的。

再来个小扩展:

```
setTimeout(_ => console.log(4))

new Promise(resolve => {
  resolve()
  console.log(1)
}).then(_ => {
  console.log(3)
})

console.log(2)
```

解答:

`setTimeout` 就是作为宏任务来存在的，而 `Promise.then` 则是具有代表性的微任务，上述代码的执行顺序就是按照序号来输出的。

所有会进入的异步都是指的事件回调中的那部分代码

也就是说 `new Promise` 在实例化的过程中所执行的代码都是同步进行的，而 `then` 中注册的回调才是异步执行的。

在同步代码执行完成后才回去检查是否有异步任务完成，并执行对应的回调，而微任务又会在宏任务之前执行。

所以就得到了上述的输出结论 `1、2、3、4`。

> 微任务、宏任务与Event-Loop
>
> https://juejin.im/post/6868849475008331783?utm_source=gold_browser_extension

5. 红灯三秒亮一次, 绿灯一秒亮一次, 黄灯2秒亮一次，实现一个函数，如何让三个灯不断交替重复亮灯? (用Promise实现) 三个亮灯函数已经存在:

```
function red () {
  console.log('red')
}

function green () {
  console.log('green')
}

function yellow () {
  console.log('yellow')
}
```

5.1. 知识点

- Promise

5.2. 参考答案

```
function red () {
  console.log('red')
}

function green () {
  console.log('green')
}

function yellow () {
  console.log('yellow')
}

function wait (cb, time) {
  return new Promise(resolve => {
    setTimeout(() => {
      cb()
      resolve()
    }, time)
  })
```

```
}

function main () {
  wait(red, 3000)
    .then(() => {
      return wait(yellow, 2000)
    })
    .then(() => {
      return wait(green, 1000)
    })
    .then(() => {
      main()
    })
}

main()
```

也可以使用 Async 函数：

```
function red () {
  console.log('red')
}

function green () {
  console.log('green')
}

function yellow () {
  console.log('yellow')
}

function wait (cb, time) {
  return new Promise(resolve => {
    setTimeout(() => {
      cb()
      resolve()
    }, time)
  })
}

async function main () {
  await wait(green, 1000)
  await wait(yellow, 2000)
  await wait(red, 3000)
  main()
}

main()
```

6. gulp 自己写过任务吗？说一下它的构建流程（阿里 2018）

6.1. 知识点

- Gulp

6.2. 参考答案

Gulp 构建过程的核心原理：

- 输入 => 加工 => 输出；
- 读取流 => 转换流 => 写入流

基本构建流程：

- 首先，安装 gulp, gulp-cli
- 创建 `gulpfile.js` 配置文件
- 在配置文件中，通过 `exports.foo = done => {...}` 创建并导出构建任务
- 在任务中通过 `gulp.src` 函数读取文件流，配合文件流的 `pipe()` 方法将文件流进行处理和转换（交给插件处理或者输出到指定文件中），最后通过 `gulp.dest` 方法将转换结果流输出到目标位置
- 然后经过处理后将执行结果返回，结束一个任务的执行
- 同时 gulp 也是支持异步任务的,可以配合 promise 和 async 创建任务。
- 接着通过 `gulp.series`、`gulp.parallel` 函数对任务组合成新的任务流程
- 最后将创建的任务通过 `exports` 导出
- 那么现在就可以运行定义的 gulp 任务

7. 说一下防抖函数的应用场景，并简单说下实现方式 （滴滴）

7.1. 知识点

- 函数防抖和函数节流
- 关联性能优化

7.2. 参考答案

应用场景：

- 页面滚动事件
  - 是否滚动到底部了
- 页面缩放事件
- 输入框输入事件
  - 搜索联想建议
  - 异步验证用户是否存在
  - ...
- 鼠标移动事件

函数防抖：

```
function debounce(fn, interval = 300) {
  let timeout = null;
  return function () {
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      fn.apply(this, arguments);
    }, interval);
  };
}
```

与之关联的还有一个函数节流：

```
function throttle(fn, interval = 300) {
  let canRun = true;
  return function () {
    if (!canRun) return;
    canRun = false;
    setTimeout(() => {
      fn.apply(this, arguments);
      canRun = true;
    }, interval);
  };
}
```

扩展：

- lodash
  - https://lodash.com/docs/4.17.15#debounce
  - https://lodash.com/docs/4.17.15#throttle

8. package-lock.json 有什么作用，如果项目中没有它会怎么样，举例说明

8.1. 知识点

- npm
- package-lock.json 文件
- 语义化版本

8.2. 参考答案

（1）package.json 文件中的 dependencies 作用

`dependencies` 字段指定了项目运行所依赖的模块，`devDependencies` 指定项目开发所需要的模块。

它们都指向一个对象。该对象的各个成员，分别由模块名和对应的版本要求组成，表示依赖的模块及其版本范围。

```
{
  "devDependencies": {
    "browserify": "~13.0.0",
    "karma-browserify": "~5.0.1"
  }
}
```

对应的版本可以加上各种限定，主要有以下几种：

> - **指定版本**：比如`1.2.2`，遵循“大版本.次要版本.小版本”的格式规定，安装时只安装指定版本。
> - **波浪号（tilde）+指定版本**：比如`~1.2.2`，表示安装1.2.x的最新版本（不低于1.2.2），但是不安装1.3.x，也就是说安装时不改变大版本号和次要版本号。
> - **插入号（caret）+指定版本**：比如ˆ1.2.2，表示安装1.x.x的最新版本（不低于1.2.2），但是不安装2.x.x，也就是说安装时不改变大版本号。需要注意的是，如果大版本号为0，则插入号的行为与波浪号相同，这是因为此时处于开发阶段，即使是次要版本号变动，也可能带来程序的不兼容。
> - **latest**：安装最新版本。

原来package.json文件只能锁定大版本，也就是版本号的第一位，并不能锁定后面的小版本，你每次npm install都是拉取的该大版本下的最新的版本，为了稳定性考虑我们几乎是不敢随意升级依赖包的，这将导致多出来很多工作量，测试/适配等，所以 package-lock.json 文件出来了，当你每次安装一个依赖的时候就锁定在你安装的这个版本。

有了 lock 文件之后，如果需要更新怎么办？两种方式：

- `npm install 包名@具体版本号`
- `npm install 包名` 默认会升级到最新稳定版

9. babel.config.js 和 .babelrc 有什么区别

9.1. 知识点

- Babel
- Babel 的配置文件

9.2. 参考答案

- https://babeljs.io/docs/en/configuration

babel 支持的配置文件主要有以下几种：

1. 在 `package.json` 中设置 `babel` 字段
2. `.babelrc` 文件或 `.babelrc.js`
3. `babel.config.js` 文件

第1种方式不用创建文件，`package.json` 加入 babel 的配置信息就行。

```
{
  "name":"babel-test",
  "version":"1.0.0",
  "devDependencies": {
    "@babel/core":"^7.4.5",
    "@babel/cli":"^7.4.4",
    "@babel/preset-env":"^7.4.5"
  },
  "babel": {
    "presets": ["@babel/preset-env"]
  }
}
```

第二种 `.babelrc` 和 `.babelrc.js` 是同一种配置方式，只是文件格式不同，一个是 json 文件，一个是 js 文件。

.babelrc

```
{
  "presets": ["@babel/preset-env"]
}
```

.babelrc.js

```
module.exports = {
  presets: ['@babel/preset-env']
};
```

这两个配置文件是针对文件夹的，即该配置文件所在的文件夹包括子文件夹都会应用此配置文件的设置，而且下层配置文件会覆盖上层配置文件，通过此种方式可以给不同的目录设置不同的规则。

而第3种 `babel.config.js` 虽然写法和 `.babelrc.js` 一样，但是 `babel.config.js` 是针对整个项目，一个项目只有一个放在项目根目录。

> 注意：
>
> - `.babelrc` 文件放置在项目根目录和 `babel.config.js` 效果一致，如果两种类型的配置文件都存在，`.babelrc` 会覆盖 `babel.config.js` 的配置。
> - 在 `package.json` 里面写配置还是创建配置文件都没有什么区别，看个人习惯。

10. 阐述一下 VUE中 eventbus 的原理 （猿辅导）

10.1. 知识点

10.2. 参考答案

```
class EventEmitter {
  constructor () {
    this.handlers = {
      // foo: [事件函数, 事件函数],
      // abc: [事件函数]
    }
```

```
  }

  on (eventName, callback) {
    // foo
    const callbacks = this.handlers[eventName]
    if (callbacks) {
      callbacks.push(callback)
    } else {
      this.handlers[eventName] = []
      this.handlers[eventName].push(callback)
    }
  }

  // foo, [1, 2, 3, 4] 剩余操作符，把所有剩余的参数收集到一个数组中
  emit (eventName, ...args) {
    const callbacks = this.handlers[eventName]
    if (callbacks) {
      callbacks.forEach(cb => {
        cb(...args) // 数组的展开操作符，一个一个的拿出来传递给函数
      })
    }
  }
}
```