# 复习

CSS预编译语言：编译之前，不是CSS。编译之后，才是CSS。

Less：CSS预编译语言之一。

编译方式：

前端编译：通过借助less.js文件，进行编译。编译过程发生在前端。会影响用户的看到页面样式的时间。

Node编译：通过借助less模块。安装到全局，使用lessc命令。

Webpack编译：通过借助style-loader、css-loader、less-loader、less模块。

语法：

嵌套式写法：子元素的选择器与样式列表可以被嵌套在父元素的样式列表中。

&：在样式列表中表示当前选择器

变量：@key: value;

混合：直接复用已有的选择器的样式。

方法：与混合类似，但是允许传递不同的属性值。

参数默认值：在方法的形参列表中通过：赋值符号进行默认值的设置。当传递了参数时，使用参数，忽略默认值。当没有传递参数时，使用默认值。

判定语句：

方法() when() {}

方法() when not() {}

方法() when () and () {}

插值语法：@{变量} 可以插入在任何地方 在属性值中要使用字符串

运算：

加减法计算：

如果只有一个运算数有单位，则使用该单位

如果有多个运算数有单位，则使用第一个单位，单位若不相同，则会转换为第一个单位

乘除法计算：

如果只有一个运算数有单位，则使用该单位

如果有多个运算数有单位，则使用第一个单位，忽略剩余单位

内置数学函数：

ceil 向上取整

floor 向下取整

round 四舍五入

percentage 转换为百分比

内置字符串函数：

e 原样输出

escape 转为URL转码之后的内容

replace 替换

内置色彩：

单词定义： red、orange、tan、gray等

rgb： rgb(红，绿，蓝);

rgba： rgba(红，绿，蓝，透明度);

十六进制： 四种定义方式：#aaa #aaaa #aaaaaa #aaaaaaaa

hsl： hsl(色阶, 饱和度, 亮度)

hsla: hsla(色阶，饱和度，亮度，透明度)

色彩通道函数：

red() 获取一个颜色的红色通道值

green() 获取一个颜色的绿色通道值

blue() 获取一个颜色的蓝色通道值

调整色彩函数：

saturate(颜色，百分比); 在原来的饱和度基础上，将颜色调高饱和度

desaturate(颜色，百分比); 在原来的饱和度基础上，将颜色降低饱和度

lighten(颜色，百分比); 在原来的亮度基础上，调高亮度

darken(颜色，百分比); 在原来的亮度基础上，降低亮度

fadeIn(颜色，百分比)；在原来的透明度基础上，将颜色变得更不透明

fadeOut(颜色，百分比)；在原来的透明度基础上，将颜色变得更透明

fade(颜色，百分比)；将颜色的透明度设置为百分比

引入文件：

@import 文件路径

@import url(文件路径)

Sass：也是CSS预编译语言之一。

需要安装ruby应用程序。

需要安装sass模块：gem install sass

编译方式：

ruby编译：安装完毕sass之后提供的全局命令 sass scss

sass scss文件 css文件

node编译：使用的模块是node-sass

webpack编译：使用的模块 style-loader、css-loader、sass-loader、node-sass

语法：

嵌套式写法：同less

&：同less

变量：$key: value

混合：

定义：@mixin 混合名 {}

调用：@include 混合名;

继承：

@extend 其它选择器的名称;

方法：

定义： @mixin 方法() {}

调用： @include 方法();

方法的参数默认值：

在定义方法的时候，通过：赋值符号给形参一个默认的值。

如果调用的时候，没有传递，则会使用该值。

插值语法：#{变量}

判定语句：

@if expression {

} @else if expression {

} @else {

}

for循环：

第一种： @for $i from start to end {} 不会执行到end

第二种： @for $i from start through end {} 会执行到end

while循环：

@while expression {

记得在这里面改变expression中的变量的值，否则死循环。

}

each循环：

@each $i in enum {}

三元语法：

if (expression, true, false);

当expression为真时，使用true值。否则使用false值。

引入文件：

@import 文件路径

ES6：ECMAScript2015

对象的定义改变：

1 当定义变量时，可以在属性名部分通过[]使用变量、表达式、运算等

2 当属性名和属性值一致时，可以省略为一个

3 当定义方法时，可以省略: function

# 一、let

ES6中，新增了let关键字。用于定义变量。

var依旧可用

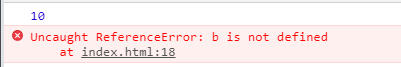
var定义的变量依旧遵循以前的ES5规则。

let定义的变量遵循ES6规则。

## 1.1 块级作用域

|  |
| --- |
| // 使用if语句开辟一个{}  if (true) {  var a = 10;  let b = 11;  }  // 在{}之后，访问a和b  console.log(a);  console.log(b); |

报错：



原因：if大括号开辟了一个块级作用域。将变量b封死在该作用域内。

## 1.2 没有变量声明的提升

let定义的变量，是没有声明提升的。

|  |
| --- |
| // 尝试在定义之前，访问  console.log(a); // undefined  console.log(b); // 报错  // 定义变量a和b  var a = 10;  let b = 11; |

报错：

|  |
| --- |
|  |

## 1.3 不可以重复定义

let定义的变量，不允许重复。var过的变量也不可以let。

|  |
| --- |
| var a = 10;  var a = 11;  // 连续通过var定义多个同名变量，是被允许的。  let a = 11; |

报错：

|  |
| --- |
|  |

## 1.4 不会挂载到window

var的全局变量是会被挂载到window身上作为属性存在的。

let的变量，就算是全局也不会挂载到window。

demo:

|  |
| --- |
| var a = 10;  let b = 11;  console.log(window.a); // 10  console.log(window.b); // undefined |

输出：

|  |
| --- |
|  |

## 1.5 for循环中

let定义的循环变量，在for循环中，循环时是多少，就是多少。

|  |
| --- |
| // 定义一个数组  var arr = [];  // 通过let定义  for (let i = 0; i < 10; i++) {  arr.push(function() {  console.log(i);  });  }  // 让不同的数组成员执行 结果是输出？  arr[0](); // 0  arr[1](); // 1  arr[2](); // 2  arr[3](); // 3  arr[7](); // 7 |

# const

const是ES6中新增的关键字。用于定义“常量”。常量指的是一直不变的量。

## 2.1 不可被=修改

常量一旦定义，将不可修改。

demo:

|  |
| --- |
| 1. const a = 10; 2. a = 11; // 尝试修改a常量 |

报错：

|  |
| --- |
|  |

demo1: 常量保存对象

|  |
| --- |
| // 使用常量保存应用类型的数据  const obj = {};  obj.a = 10;  console.log(obj); |

不会报错：

|  |
| --- |
|  |

结论：const只关心变量是否被=修改。至于变量保存的内容是否变化，const不关心。

## 2.2 其它特点

const与let一块出来的。所以也遵循let的其它特点： 没有变量声明的提升、不会挂载到window、不可以重复定义。

注：千万不要将循环变量用const来定义。

# 字符串

## 3.1 多行字符串

在ES6之前，我们想要在JS中定义一段HTML代码。需要：要么使用数组、要么换行（需要连字符特别难看）、要么定义在不可执行的script标签中。

在ES6中，新增了多行字符串的定义方式：``

demo:

|  |
| --- |
| // 现在  let str2 = `<div>大家好，我是一个多行字符串</div>  <ul>  <li>1</li>  <li>1</li>  <li>1</li>  <li>1</li>  <li>1</li>  <li>1</li>  <li>1</li>  <li>1</li>  </ul>  `; |

输出：

|  |
| --- |
|  |

## 3.2 插值语法

ES6中，在多行字符串中，提供了插值语法： ${}

开辟了一个JS执行环境

内部可以调用任何的JS语法

demo:

|  |
| --- |
| // 通过ajax请求一个对象回来  let obj = {  username: "王老五",  bank: {  title: "招商银行",  content: "只生一个好"  }  }  // 现在  let str2 = `<div>个人简介</div>  <div><span>${obj.username}</span></div>  <div><span>${obj.bank.title}</span></div>  <div><span>${obj.bank.content}</span></div>  `;  console.log(str2); |

输出:

|  |
| --- |
|  |

## 3.3 startsWith

str.startsWith(str1, pos); 该方法用于判定一个字符串是否以另外一个字符串开头（或者从指定位置开头）

str1: 开头字符串

pos: 指定的位置 下标slice的第一个参数的值

结果：布尔值 如果str的pos下标位置是str1.则返回真。否则返回假。

demo:

|  |
| --- |
| // 定义一个字符串  let str = "今天天气不错";  // 定义另外一个字符串  let str1 = "今天";  // 判定str1是不是str的开头字符串  let result = str.startsWith(str1);  console.log(result); |

结果：

|  |
| --- |
|  |

## 3.4 endsWith

str.endsWith(str1, pos); 该方法用于判定一个字符串是否以另外一个字符串结尾（或者从指定位置结尾）

str1: 结尾字符串

pos: 指定的位置 slice方法的第二个参数

结果：布尔值 如果str的pos下标位置是str1.则返回真。否则返回假。

demo

|  |
| --- |
| // 定义一个字符串  let str = "今天天气不错";  // 定义另外一个字符串  let str1 = "天气";  // 判定 str 是否是以 str1 作为结尾  let result = str.endsWith(str1, 4);   1. console.log(result); |

输出

|  |
| --- |
|  |

## 3.5 includes

str.includes(str1, pos); 该方法用于判定一个字符串是否包含另外一个字符串（或者从指定位置之后是否包含）

str1: 被包含的字符串

pos: 截取位置 从该位置开始往后截取

return: 布尔值 如果从该位置开始往后截取的字符串中包含str1字符串，则为真，否则为假。

demo:

|  |
| --- |
| // 定义一个字符串  let str = "今天天气不错";  // 定义另外一个字符串  let str1 = "天气";  // 判定 str1 是否在str中  let result = str.includes(str1, 2);  console.log(result); |

输出：

|  |
| --- |
|  |

## 3.6 repeat

str.repeat(num); 该方法用于将str复制num次。

num：被复制的次数

最终的长度：str.length \* num;

返回值：被复制之后的新字符串

demo:

|  |
| --- |
| 1. // 定义一个字符串 2. let str = "你好"; 3. let str1 = str.repeat(100); 4. console.log(str1, str1.length); |

# 对象的新增方法

## 4.1 Object.is

该方法用于判定两者是否全等。

demo1:

|  |
| --- |
| 0 === -0 // true  Object.is(0, -0); // false |

demo2:

|  |
| --- |
| 1. NaN === NaN // false 2. Object.is(NaN, NaN); // true |

输出:

|  |
| --- |
|  |

## 4.2 Object.assign

Object.assign(target); 该方法用于给某一个对象增加其它对象的属性和方法。

该方法接收任意个参数，第一个是被赋予者。剩余的都是赋予者。

target: 被增加属性和方法的对象

剩余的所有参数都是提供属性和方法的对象。

注：这是浅复制、浅拷贝。

demo:

|  |
| --- |
| // 定义一个对象  let obj = {  };  // 定义一些其它对象  let obj1 = {  color: "red",  sayHello() {  console.log(123);  }  }  // 定义其它对象  let obj2 = {  name: "张三",  dog: {  name: "旺财"  }  }  // 将obj1和obj2所具备的属性和方法交给obj对象  Object.assign(obj, obj1, obj2);  // 输出obj  console.log(obj); |

输出：

|  |
| --- |
|  |

# 数组的方法

## 5.1 Array.of

该方法用于定义一个新的数组。

Array.of接收任意个参数，每一个参数都被当做数组的成员去处理。

demo:

|  |
| --- |
| 1. let arr3 = Array.of(1); // [1] 2. let arr4 = Array.of(1, 2); // [1, 2] |

## 5.2 Array.from

该方法用于将一个类数组对象，转化为数组。

类数组对象：数字下标、length属性

demo:

|  |
| --- |
| 1. // 定义一个类数组对象 2. let likeArr = { 3. 0: "a", 4. 1: "b", 5. 2: "c", 6. 3: "d", 7. length: 4 8. } 9. let arr = Array.from(likeArr); 10. console.log(arr); |

结果：

|  |
| --- |
|  |

## 5.3 find

arr.find(handler) 该方法用于模糊查询数组中的成员

handler: 函数 该函数必须返回一个布尔值

该函数中有3个参数：value、index、arr

返回值：返回的是符合handler布尔值表达式的成员

demo:

|  |
| --- |
| // 定义数组  let arr = ["东邪", "西毒", "南帝", "北丐", "中神通"];  // 查询数组中 带有“北”这个字的成员  let result = arr.find(function(value) {  return value.includes("北");  });  console.log(result); |

输出：

|  |
| --- |
|  |

还原：

|  |
| --- |
| // 还原  Array.prototype.find = function(handler) {  for (var i = 0; i < this.length; i++) {  if (handler(this[i], i, this)) {  return this[i];  }  }  } |

# 下午复习

let：ES6中新增的关键字。

作用：定义变量

效果：定义的变量遵循ES6的新规范。

新规范：

块级作用域：if、while、for循环的{}都可以开辟一个JS作用域，这个作用域叫做块级作用域。

注：作用域，在书写完代码之后，就已经确定了。

没有变量声明的提升

不可以重复定义

定义在全局作用域中，也不会挂载到window对象身上

for循环中，可以记住当时的值

const：ES6中新增的关键字。

作用：定义常量

效果：定义的常量遵循ES6的新规范

新规范：同上（除了最后一条，因为常量不能当变量用）

注：定义的常量保存的值如果是值类型，则无论如何无法修改。如果是引用类型，可以通过点语法或者方括号语法，或者delete关键字进行属性的增加、修改、移除

字符串：

多行字符串： 使用飘键定义 ``

插值语法： ${} 括号内是一个JS执行环境，可以书写任何的JS代码

startsWith：判定一个字符串是否从指定位置开始是另外一个字符串

endsWith: 判定一个字符串是否从指定位置结束是另外一个字符串

includes: 判定一个字符串是否从指定位置开始往后包含另外一个字符串

repeat: 将一个字符串重复多少次并返回新字符串

对象的新增方法：

Object.is: 用于判定两者是否全等 与 === 重合性很高

Object.is(0, -0); 作为有符号整数和无符号整数进行判定的

Object.is(NaN, NaN); 作为是否是“不是一个数”性质进行判定的

Object.assign: 用于将从第二个参数开始的所有对象的属性和方法浅复制给第一个参数。

数组的新方法：

Array.of: 用于定义数组 参数不论是什么类型，都是作为数组成员存在。

Array.from: 用于将一个类数组对象转化为数组。

类数组对象：必须有连续的数字属性名，还要有length属性

find：用于模糊查询数组中的某一个成员 找到以后就立即停止

## 5.4 findIndex

arr.findIndex(handler) 该方法用于模糊查询数组中的成员的索引

handler: 函数 该函数必须返回一个布尔值

该函数中有3个参数：value、index、arr

返回值：返回的是符合handler布尔值表达式的成员的索引 如果没有找到，则返回 -1

demo:

|  |
| --- |
| 1. // 定义数组 2. let arr = ["东邪", "西毒", "南帝", "北丐", "中神通", "北乔峰"]; 3. // 查询数组中 带有“北”这个字的成员的索引 4. let result = arr.findIndex(function(value) { 5. return value.includes("慕容"); 6. }); 7. console.log(result); |

输出：

|  |
| --- |
|  |

## 5.5 数组的内部复制

arr.copyWithin(pos, start, end); 该方法用于数组的内部复制 将数组内的start开始（包含）到end结束（不包含）复制，从pos位置开始粘贴（替换）。

pos: 粘贴（替换）的起始位置

start: 复制的开始位置（包含）

end: 复制的结束位置（不包含）

demo:

|  |
| --- |
| // 定义数组  let arr = ["东邪", "西毒", "南帝", "北丐", "中神通", "北乔峰"];  // 调用copyWithin方法  arr.copyWithin(1, 4, 6);  // 输出arr  console.log(arr); |

输出：

|  |
| --- |
|  |

还原1：

|  |
| --- |
| Array.prototype.copyWithin = function(pos, start, end) {  // 截取从开始到结束的位置  let newArr = this.slice(start, end);  // 保存原数组的长度  let length = this.length;  // 从pos位置开始粘贴（替换）  for (let i = 0; i < newArr.length; i++) {  // 将原数组的从pos位置开始替换  this[i + pos] = newArr[i];  }  // 重新设置数组的长度  this.length = length;  // 返回this  return this;  } |

还原2:

|  |
| --- |
| 1. Array.prototype.copyWithin = function(pos, start, end) { 2. // 截取从开始到结束的位置 3. let newArr = this.slice(start, end); 4. // 保存原数组的长度 5. let length = this.length; 6. // [pos, newArr.length].concat(newArr) => [pos, newArr.length, newArr[1], newArr[2], ……] 7. this.splice.apply(this, [pos, newArr.length].concat(newArr)); 8. // this.splice(pos, newArr.length, newArr[1], newArr[2], ……) 9. this.length = length; 10. return this; 11. } |

# Symbol

ES6之前，一共有6种数据类型：string、number、boolean、undefined、null、object

ES6中，新增了一种symbol类型。它表示一种“独一无二的符号”类型。属于值类型。

## 6.1 定义symbol

Symbol函数每执行一次，定义一个symbol数据值。

参数只有说明的作用，没有任何其它作用。

Symbol函数不是一个构造函数，不要使用new、不要使用new、不要使用new。



demo:

|  |
| --- |
| let s = Symbol("你好");  let s1 = Symbol("你好"); console.log(s);  console.log(s1);  console.log(s === s1); |

输出:

|  |
| --- |
|  |

## 6.2 作用

通常是用来解决对象的属性名无法重复的问题。

demo:

|  |
| --- |
| // 定义对象  let obj = {  [s]: "你好",  [s1]: "你好1"  }  console.log(obj); |

输出:

|  |
| --- |
|  |

# 代理对象

ES6中，新增了一个构造函数，用于生成代理对象。

代理：访问的是对象A，其实访问的是另外一个对象B。

A是B的代理对象。

demo:

|  |
| --- |
| let proxy = (function() {  // 源对象  let star = {  name: "胡歌",  age: 38,  sex: "男",  hobby: ["读书", "跑步", "摄影", "旅游"]  };  // 代理对象  return new Proxy(star, {  // get方法在通过proxy对象进行属性的读取或者访问时，会执行  get: function(star, prop, proxy) {  console.log("想要读取属性", arguments);  if (prop === "age") {  return star[prop] - 10;  }  return star[prop];  },  // set方法在通过proxy对象进行属性的设置时，会执行  set: function() {  console.log("想要设置属性", arguments);  }  });  })(); |

# Number的新增方法

## 8.1 isNaN

在ES6之前，window身上有一个isNaN方法。 判定一个值是否是NaN

在ES6中，Number身上也增加了一个isNaN方法 判定一个数字是否是NaN

demo:

|  |
| --- |
| 1. // 定义一些值 2. let num = 1/0; 3. let num1 = 0/0; 4. let num2 = "0"; 5. let num3 = NaN; 6. let num4 = "1asdfdsaf23"; 7. // window.isNaN 8. console.log(window.isNaN(num)); // false 9. console.log(window.isNaN(num1)); // true 10. console.log(window.isNaN(num2)); // false 11. console.log(window.isNaN(num3)); // true 12. console.log(window.isNaN(num4)); // true 13. // Number.isNaN 14. console.log(Number.isNaN(num)); // false 15. console.log(Number.isNaN(num1)); // true 16. console.log(Number.isNaN(num2)); // false 17. console.log(Number.isNaN(num3)); // true 18. console.log(Number.isNaN(num4)); // false |

## 8.2 isFinite

window.isFinite 用于判定一个值是否是有限的

Number.isFinite 用于判定一个数字是否是有限的 如果参数不是数字 直接false

demo:

|  |
| --- |
| 1. // 定义一些值 2. let num = 1/0; 3. let num1 = 0/0; 4. let num2 = "0"; 5. let num3 = NaN; 6. let num4 = "1asdfdsaf23"; 7. // window.isFinite 8. console.log(window.isFinite(num)); // false 9. console.log(window.isFinite(num1)); // false 10. console.log(window.isFinite(num2)); // true 11. console.log(window.isFinite(num3)); // false 12. console.log(window.isFinite(num4)); // false 13. // Number.isFinite 14. console.log(Number.isFinite(num)); // false 15. console.log(Number.isFinite(num1)); // false 16. console.log(Number.isFinite(num2)); // false 17. console.log(Number.isFinite(num3)); // false 18. console.log(Number.isFinite(num4)); // false |

## 8.3 isInteger

Number.isInteger 该方法用于判定一个数字是否是整数

demo

|  |
| --- |
| 1. // 定义一些值 2. let num = 1/0; 3. let num1 = 0.0; 4. let num2 = "0"; 5. let num3 = NaN; 6. // Number.isInteger 7. console.log(Number.isInteger(num)); // false 8. console.log(Number.isInteger(num1)); // true 9. console.log(Number.isInteger(num2)); // false 10. console.log(Number.isInteger(num3)); // false |

# 解构

解构：解除构造、解除结构。反义词：封装。

解构有两种：解构对象、解构数组。

## 9.1 解构对象

解构语法： let {key, key1, key2……} = obj;

封装对象：

|  |
| --- |
| // 如下是一个工厂函数，它将创建对象的代码 封装在函数内部  function createObjectFactory(username, age, sex) {  var obj = {  username: username,  age: age,  sex: sex  }  return obj;  }  // 其实是将name、age、sex三个形参封装起来。  let obj = createObjectFactory("小白", 13, "女"); |

解构：

|  |
| --- |
| // 以前的解构代码  var username = obj.username;  var age = obj.age;  var sex = obj.sex;  console.log(username, age, sex);  // ES6中 针对不同的数据结构，提供了不同的解构方式  var {username, age, sex} = obj;  console.log(username, age, sex); |

输出：

|  |
| --- |
|  |

## 9.2 解构数组

解构语法： let [a, b, c,……] = arr;

|  |
| --- |
| // 封装数组  let arr = [1, 2, 3];  // 解构数组  let [a, b, c] = arr;  // 等价于  // let a = arr[0];  // let b = arr[1];  // let c = arr[2]; |

# ...语法（拓展语法）

ES6中，因为新增了箭头函数，而箭头函数有一个特点就是不可以再使用arguments。所以，想要获取所有的参数，就需要别的方式。...语法就是补充。

## 10.1 第一种 函数中使用

|  |
| --- |
| // 定义函数  function fun(a, b, c, ...arg) { // 通过...语法获取剩余参数 该语法只能够写在最后面  console.log(arg);  }  // 调用函数  fun(1, 2, 3);  fun(1, 2, 3, 4, 5);  fun(1, 2, 3, 4, 5, 6, 7); |

## 10.2 第二种 解构时使用

|  |
| --- |
| 1. // 定义数组 2. let arr = [1, 2, 3, 4, 5, 6, 7]; 3. // 解构 只要前3项，剩余的要放入另外一个数组 4. let [a, b, c, ...d] = arr; 5. console.log(a, b, c, d); |

## 10.3 第三种 传递参数时使用

|  |
| --- |
| // 定义数组  let arr = [1, 2, 3, 4];  // 输出数组  console.log(arr);  console.log(1, 2, 3, 4);  console.log(...arr); |

# 箭头函数

## 11.1 定义

ES6中新增了一种函数，叫做箭头函数。

定义语法： var fun = () => {}

demo:

|  |
| --- |
| 1. // 定义箭头函数 2. var fun = () => { 3. console.log("你好，我是一个箭头函数"); 4. } 5. // 执行 6. fun(); |

输出：

|  |
| --- |
|  |

## 11.2 箭头函数中的this

箭头函数中的this，在定义的时候，就会确定下来。（不是写完代码就能确定的）

|  |
| --- |
| 1. // 定义箭头函数 2. var fun = () => { 3. console.log(this); 4. } 5. fun(); // window 6. fun.call(document.body); // window 7. document.onclick = fun; // widnow |

如果想要改变箭头函数中的this的指向，就将箭头函数的定义代码放在一个普通函数内。在调用普通函数的时候，改变该函数的this。也就改变了箭头函数的this.

demo:

|  |
| --- |
| 1. // 尝试修改箭头函数中的this 2. function createArrowFun() { 3. // 返回一个箭头函数 4. return () => { 5. console.log(this); 6. } 7. } 8. var arrowFun1 = createArrowFun(); 9. arrowFun1(); // window 10. var arrowFun2 = createArrowFun.call(document.body); 11. arrowFun2(); // document.body |

## 11.3 参数默认值

ES6中，给所有的函数，添加了参数默认值功能。

语法： function fun(a = x, b = x, c = x) {}

demo:

|  |
| --- |
| // 参数默认值 不仅仅箭头函数拥有 普通函数也拥有  function fun(a = 1, b = 2) {  return a + b;  }  console.log(fun()); // 3 |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |