9、如何实现数组去重？

答：indexof数组重组；对象空间换时间；利用from new set()。

时间开销：其中indexof每次都会遍历整个数组，时间开销极大；对象的方案时间开销极少，把value作为通过下标的形式存入一个object内，下标的引用要比用indexof搜索数组快的多。

空间开销：object方式会创建一个hash表，占用的空间是比数组形式的更多的。

hash局限：下标会被转换为字符串，1和'1'就会被认为是重复的，但实际应该保留。可以把hash表的值从true改为一个数组，里面保存出现过的类型就行了。

10：如何用正则实现string.trim()?

答：

```
str.replace(/(^\s+)|(\s+$)/g, "");
```

则表达式是用于匹配字符串中字符组合的模式。在JS中，正则表达式也是对象，这些模式被用于RegExp的exec和test方法，以及String的match、replace、search和split方法。

创建一个正则表达式

你可以使用一下两种方法之一构建一个正则表达式：

```
/*使用一个正则表达式字面量，其由包含在斜杠之间的模式组成。
在加载脚本后，正则表达式字面值提供正则表达式的编译。当正则表达式保持不变时，使用此方法可获得更好的性能。
    /pattern/flags
*/

const regex = /ab+c/;

const regex = /^[a-zA-Z]+[0-9]*\W?_$/gi;
```

```
/*调用RegExp对象的构造函数。
使用构造函数提供正则表达式的运行时编译。使用构造函数，当你知道正则表达式模式将会改变，或者你不知道模式，并从另一个来源，如用户输入。
    new RegExp(pattern [, flags])
*/

let regex = new RegExp('ab+c');

let regex = new RegExp(/^[a-zA-Z]+[0-9]*\W?_$/, 'gi');

let regex = new RegExp('^[a-zA-Z]+[0-9]*\\W?_$', 'gi');
```

编写一个正则表达式的模式

一个正则表达式模式是由简单的字符所构成的，比如/abc/，或者是简单和特殊字符的组合，比如/ab*c/或/Chapter (\d+)\.\d*/。后者用到了括号，它在正则表达式中可以被用作是一个记忆设备。这一部分正则所匹配的字符将会被记住，在后面可以被利用。

使用简单的模式：简单的模式是由你找到的直接匹配所构成的。比如：/abc/这个模式就匹配了在一个字符串

中，仅仅字符‘abc’同时出现并按照这个顺序。
使用特殊字符：当你需要搜索一个比直接匹配需要更多条件的匹配时，比如寻找一个或多个'b‘，或者寻找空格，那么这时模式将要包含特殊字符。比如：/ab*c/匹配了一个单独的'a'后面跟了零个或多个‘b'，且后面跟着’c‘的任何字符组合。

下面的表格列出了一个我们在正则表达式中可以利用的特殊字符的完整列表和描述。

字符	含义
\	匹配将依照下列规则： 在非特殊字符之前的反斜杠表示下一个字符是特殊的，不能从字面上解释。例如，没有前面'\'的'b'通常匹配小写'b'，无论它们出现在哪里。如果加了'\',这个字符变成了一个特殊意义的字符，意思是匹配一个字符边界。 反斜杠也可以将其后的特殊字符，转义为字面量。例如，模式 /a*/ 代表会匹配 0 个或者多个 a。相反，模式 /a*/ 将 '*' 的特殊性移除，从而可以匹配像 "a*" 这样的字符串。 使用 new RegExp("pattern") 的时候不要忘记将 \ 进行转义，因为 \ 在字符串里面也是一个转义字符。
^	匹配输入的开始。如果多行标志被设置为true，那么也匹配换行符后紧跟的位置。 例如，/^A/ 并不会匹配 "an A" 中的 'A'，但是会匹配 "An E" 中的 'A'。 当 '^' 作为第一个字符出现在一个字符集合模式时，它将会有不同的含义。补充字符集合 一节有详细介绍和示例。
$	匹配输入的结束。如果多行标示被设置为true，那么也匹配换行符前的位置。 例如，/t$/ 并不会匹配 "eater" 中的 't'，但是会匹配 "eat" 中的 't'。
*	匹配前一个表达式0次或多次。等价于 {0,}。 例如，/bo*/会匹配 "A ghost booooed" 中的 'booooo' 和 "A bird warbled" 中的 'b'，但是在 "A goat grunted" 中将不会匹配任何东西。
+	匹配前面一个表达式1次或者多次。等价于 {1,}。 例如，/a+/匹配了在 "candy" 中的 'a'，和在 "caaaaaaandy" 中所有的 'a'。
?	匹配前面一个表达式0次或者1次。等价于 {0,1}。 例如，/e?le?/ 匹配 "angel" 中的 'el'，和 "angle" 中的 'le' 以及"oslo' 中的'l'。 如果紧跟在任何量词 *、 +、? 或 {} 的后面，将会使量词变为非贪婪的（匹配尽量少的字符），和缺省使用的贪婪模式（匹配尽可能多的字符）正好相反。

	例如，对 "123abc" 应用 /\d+/ 将会返回 "123"，如果使用 /\d+?/,那么就只会匹配到 "1"。 还可以运用于先行断言，如本表的 `x(?=y)` 和 `x(?!y)` 条目中所述。
`.`	（小数点）匹配除换行符之外的任何单个字符。 例如，/.n/将会匹配 "nay, an apple is on the tree" 中的 'an' 和 'on'，但是不会匹配 'nay'。
`(x)`	匹配 'x' 并且记住匹配项，就像下面的例子展示的那样。括号被称为 捕获括号。 模式`/(foo) (bar) \1 \2/`中的 '(foo)' 和 '(bar)' 匹配并记住字符串 "foo bar foo bar" 中前两个单词。模式中的 \1 和 \2 匹配字符串的后两个单词。注意 \1、\2、\n 是用在正则表达式的匹配环节。在正则表达式的替换环节，则要使用像 $1、$2、$n 这样的语法，例如，'bar foo'.replace(/(...) (...)/, '$2 $1')。
`(?:x)`	匹配 'x' 但是不记住匹配项。这种叫作非捕获括号，使得你能够定义为与正则表达式运算符一起使用的子表达式。来看示例表达式 /(?:foo){1,2}/。如果表达式是 /foo{1,2}/，{1,2}将只对 'foo' 的最后一个字符 'o' 生效。如果使用非捕获括号，则{1,2}会匹配整个 'foo' 单词。
`x(?=y)`	匹配'x'仅仅当'x'后面跟着'y'.这种叫做正向肯定查找。 例如，/Jack(?=Sprat)/会匹配到'Jack'仅仅当它后面跟着'Sprat'。/Jack(?=Sprat|Frost)/匹配'Jack'仅仅当它后面跟着'Sprat'或者是'Frost'。但是'Sprat'和'Frost'都不是匹配结果的一部分。
`x(?!y)`	匹配'x'仅仅当'x'后面不跟着'y',这个叫做正向否定查找。 例如，/\d+(?!\.)/匹配一个数字仅仅当这个数字后面没有跟小数点的时候。正则表达式/\d+(?!\.)/.exec("3.141")匹配'141'但是不是'3.141'
`x|y`	匹配'x'或者'y'。 例如，/green|red/匹配"green apple"中的'green'和"red apple"中的'red'
`{n}`	n是一个正整数，匹配了前面一个字符刚好发生了n次。 比如，/a{2}/不会匹配"candy"中的'a',但是会匹配"caandy"中所有的a，以及"caaandy"中的前两个'a'。
`{n,m}`	n 和 m 都是整数。匹配前面的字符至少n次，最多m次。如果 n 或者 m 的值是0， 这个值被忽略。 例如，/a{1, 3}/ 并不匹配"cndy"中的任意字符，匹配"candy"中得a，匹配"caandy"中的前两个a，也匹配"caaaaaaandy"中的前三个a。注意，当匹配"caaaaaaandy"时，匹配的值是"aaa"，即使原始的字符串中有更多的a。

`[xyz]`	一个字符集合。匹配方括号的中任意字符，包括转义序列。你可以使用破折号（-）来指定一个字符范围。对于点（.）和星号（*）这样的特殊符号在一个字符集中没有特殊的意义。他们不必进行转义，不过转义也是起作用的。 例如，[abcd] 和[a-d]是一样的。他们都匹配"brisket"中得‘b’,也都匹配“city”中的‘c’。/[a-z.]+/ 和/[\w.]+/都匹配“test.i.ng”中得所有字符。
`[^xyz]`	一个反向字符集。也就是说， 它匹配任何没有包含在方括号中的字符。你可以使用破折号（-）来指定一个字符范围。任何普通字符在这里都是起作用的。 例如，[^abc] 和 [^a-c] 是一样的。他们匹配"brisket"中得‘r’，也匹配“chop”中的‘h’。
`[\b]`	匹配一个退格(U+0008)。（不要和\b混淆了。）
`\b`	匹配一个词的边界。一个词的边界就是一个词不被另外一个词跟随的位置或者不是另一个词汇字符前边的位置。注意，一个匹配的词的边界并不包含在匹配的内容中。换句话说，一个匹配的词的边界的内容的长度是0。（不要和[\b]混淆了） 例子： /\bm/匹配“moon”中得‘m’; /oo\b/并不匹配"moon"中得'oo'，因为'oo'被一个词汇字符'n'紧跟着。 /oon\b/匹配"moon"中得'oon'，因为'oon'是这个字符串的结束部分。这样他没有被一个词汇字符紧跟着。 /\w\b\w/将不能匹配任何字符串，因为一个单词中的字符永远也不可能被一个非词汇字符和一个词汇字符同时紧跟着。 > 注意: JavaScript的正则表达式引擎将特定的字符集定义为“字”字符。不在该集合中的任何字符都被认为是一个断词。这组字符相当有限：它只包括大写和小写的罗马字母，小数位数和下划线字符。不幸的是，重要的字符，例如“é”或“ü”，被视为断词。
`\B`	匹配一个非单词边界。他匹配一个前后字符都是相同类型的位置：都是单词或者都不是单词。一个字符串的开始和结尾都被认为是非单词。 例如，/\B../匹配"noonday"中得'oo', 而/y\B./匹配"possibly yesterday"中得‘ye‘
`\cX`	当X是处于A到Z之间的字符的时候，匹配字符串中的一个控制符。 例如，`/\cM/` 匹配字符串中的 control-M (U+000D)。

`\d`	匹配一个数字。 等价于`[0-9]`。 例如， `/\d/` 或者 `/[0-9]/` 匹配"B2 is the suite number."中的'2'。
`\D`	匹配一个非数字字符。 等价于`[^0-9]`。 例如， `/\D/` 或者 `/[^0-9]/` 匹配"B2 is the suite number."中的'B' 。
`\f`	匹配一个换页符 (U+000C)。
`\n`	匹配一个换行符 (U+000A)。
`\r`	匹配一个回车符 (U+000D)。
`\s`	匹配一个空白字符，包括空格、制表符、换页符和换行符。 等价于[\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]。 例如, `/\s\w*/` 匹配"foo bar."中的' bar'。
`\S`	匹配一个非空白字符。 等价于`[^ \f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`。 例如， `/\S\w*/` 匹配"foo bar."中的'foo'。
`\t`	匹配一个水平制表符 (U+0009)。
`\v`	匹配一个垂直制表符 (U+000B)。
`\w`	匹配一个单字字符（字母、数字或者下划线）。 等价于`[A-Za-z0-9_]`。 例如, `/\w/` 匹配 "apple," 中的 'a'，"$5.28,"中的 '5' 和 "3D." 中的 '3'。
`\W`	匹配一个非单字字符。 等价于`[^A-Za-z0-9_]`。 例如, `/\W/` 或者 `/[^A-Za-z0-9_]/` 匹配 "50%." 中的 '%'。

`\n`	当 n 是一个正整数，一个返回引用到最后一个与有n插入的正则表达式(counting left parentheses)匹配的副字符串。 比如 `/apple(,)\sorange\1/` 匹配"apple, orange, cherry, peach."中的'apple, orange,' 。
`\0`	匹配 NULL (U+0000) 字符， 不要在这后面跟其它小数，因为 `\0<digits>` 是一个八进制转义序列。
`\xhh`	与代码 hh 匹配字符（两个十六进制数字）
`\uhhhh`	与代码 hhhh 匹配字符（四个十六进制数字）。

11、JS原型是什么？

答：一个函数可以看成一个类，原型是所有类都有的一个属性，原型的作用就是给这个类的每一个对象都添加一个统一的方法。

解释一：MDN 继承与原型链

对于有基于类的语言经验（如java或c++）的开发人员来说，JavaScript有点令人困惑，因为它是动态的，并且本身不提供一个class实现。（在ES2015/ES6中引入了class关键字，但只是语法糖，**JavaScript仍然是基于原型的**）。

当谈到继承时，JavaScript只有一种结构：对象。每个对象都有一个私有属性（称之为[[Prototype]]），它指向它的原型对象（prototype）。该prototype对象又具有一个自己的prototype，层层向上直到一个对象的原型为null。根据定义，null没有原型，并作为这个原型链中的最后一个环节。 几乎所有JavaScript中的对象都是位于原型链顶端的Object的实例。 原型继承经常被视为JavaScript的一个弱点，但事实上，原型继承模型比经典的继承模型更加强大。例如，在一个原型模型之上构建一个经典模型是相当容易的。

基于原型链的继承

JS对象是动态的属性“包”（指其自己的属性）。JS对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，知道找到一个名字匹配的属性或到达原型链的末尾。

在原型链上查找属性比较耗时，对性能有副作用，这在性能要求苛刻的情况下很重要。另外，试图访问不存在的属性时会遍历整个原型链。

结论

在用原型继承编写复杂代码之前，了解原型继承模型非常重要。同时，要注意代码中的原型链的长度，并在必要时将其分解，以避免潜在的性能问题。此外，永远不要扩展原生对象的原型，除非是为了兼容新的JS特性

解释二：理解JS原型

*什么是原型：原型是一个对象，其他对象可以通过它实现属性继承。

*任何一个对象都可以成为原型吗：是

*哪些对象有原型：所有的对象在默认的情况下都有一个原型，因为原型本身也是对象，所以每个原型自身又有一个原型（只有一种例外，默认的对象原型在原型链的顶端）

*那什么又是对象呢：在JS中，一个对象就是任何无序键值对的集合，如果它不是一个主数据类型（undefined、null、boolean、number、or string），那么它就是一个对象。

C.prototype == new C().__proto__;

1.每个函数都有一个prototype属性：指向该函数的原型对象，默认是一个空对象。

2.每个对象都有一个__proto__属性：指向构造它的函数的原型对象。只有Object的原型对象为null。

3.js中对象有两个概念，一个是function类型对象，一个是object类型的对象
原型链：o-->Object-->null；f-->Function-->Object-->null。
4.每个原型对象都有一个constructor（构造函数）和__proto__属性。C==C.prototype.constructor==new C().__proto__.constructor。__proto__属性指向了该原型对象的构造函数的原型对象（及原型对象的原型对象），通过这个属性就构成了一个原型链。
5、通过原型链就实现了继承。当调用一个对象的方法或属性时它首先会在本对象找，然后在原型对象找，再往原型对象的原型对象找，直到最后一个null，这也就是为什么，每个自定义的对象都可以调用Object对象的方法，它的内部机制就是原型链。
6、每个函数的原型对象的类型都是object类型，除了自定义函数的构造函数（C.constructor）。
7、每个对象都有一个constructor属性，指向它的构造函数。C==new C().constructor
12、ES6中的class了解吗？
答：ES2015中引入的JS类实质上是JS现有的基于原型的继承的语法糖。类语法不会为JS引入新的面向对象的继承模型。
　类实际上是个“特殊的函数”，就像你能够定义的函数表达式和函数声明一样，雷雨发有两个组成部分：类表达式和类声明。
13、JS如何实现继承？
答：JS的设计理念：一切皆对象？对象间通过原型来实现继承！
面向对象三大特征：封装、继承和多态。

弱类型实现继承的理论可行性

　“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被成为鸭子。”
　基础弱类型语言多了就会渐渐听说鸭子类型（duck typing），在这种风格中，一个对象有效的语义，不是有继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定的，在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。通俗来说一个对象如果拥有某个对象的方法和属性，就可以看作是这个对象。

JS继承实现原理

　根据duck typing的理论，在JS中如果对象subType包含了superType的属性和方法，就可以认为其继承了superType。在JS将原型链作为实现继承的主要方法。基本原理是利用原型让一个subType引用superType的属性和方法。可以将subType的prototype属性设为superType的实例，这时候subType的原型链中就包含了superType的属性和方法，大致是这样的：

```
function SuperType() {
    this.property = true;
}
SuperType.prototype.getSuperValue = function() {
    return this.property;
}
function SubType() {
    this.subProperty = false;
}
SubType.prototype = new SuperType();
SubType.prototype.getSubValue = function() {
    return this.subProperty;
}
```

　这样SubType就实现了对SuperType的继承

格式化文本

标题1

标题2

标题3

标题4

标题5

标题6

段落
地址

预格式文本

普通