- 用户访问系统2的受保护资源
- 系统2发现用户未登录，跳转至sso认证中心，并将自己的地址作为参数
- sso认证中心发现用户已登录，跳转回系统2的地址，并附上令牌
- 系统2拿到令牌，去sso认证中心校验令牌是否有效
- sso认证中心校验令牌，返回有效，注册系统2
- 系统2使用该令牌创建与用户的局部会话，返回受保护资源

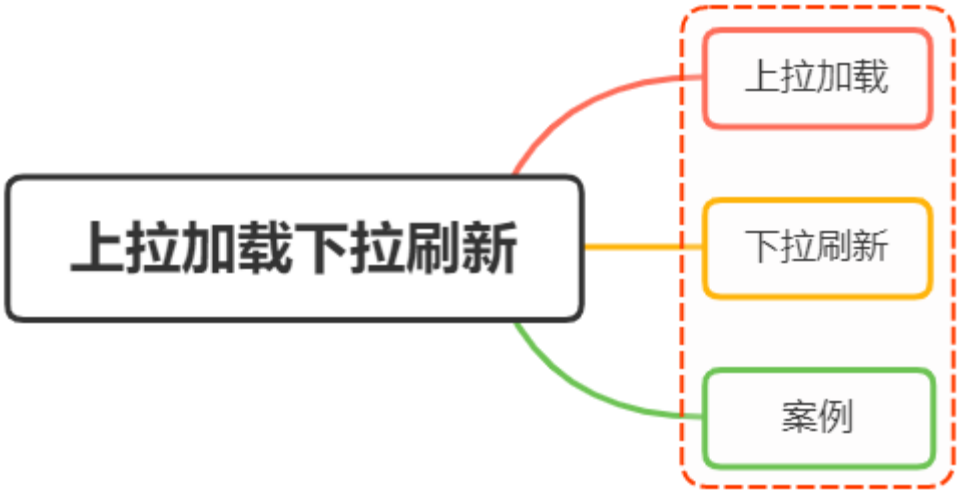
用户登录成功之后，会与 `sso` 认证中心及各个子系统建立会话，用户与 `sso` 认证中心建立的会话称为全局会话

用户与各个子系统建立的会话称为局部会话，局部会话建立之后，用户访问子系统受保护资源将不再通过 `sso` 认证中心

全局会话与局部会话有如下约束关系：

- 局部会话存在，全局会话一定存在
- 全局会话存在，局部会话不一定存在
- 全局会话销毁，局部会话必须销毁

28. 如何实现上拉加载，下拉刷新?



28.1. 前言

下拉刷新和上拉加载这两种交互方式通常出现在移动端中

本质上等同于PC网页中的分页，只是交互形式不同

开源社区也有很多优秀的解决方案，如 `iscroll` 、 `better-scroll` 、 `pulltorefresh.js` 库等等

这些第三方库使用起来非常便捷

我们通过原生的方式实现一次上拉加载，下拉刷新，有助于对第三方库有更好的理解与使用

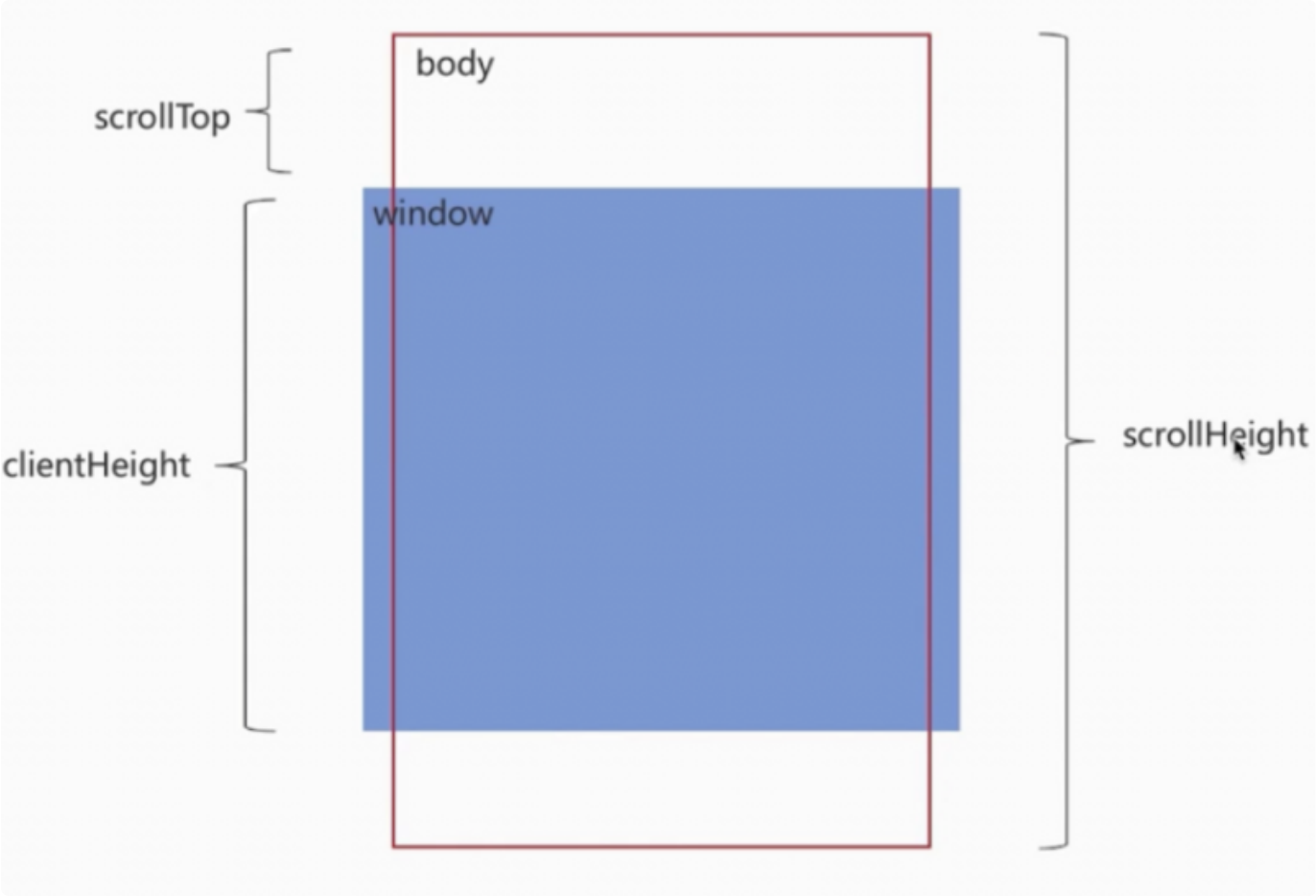
28.2. 实现原理

上拉加载及下拉刷新都依赖于用户交互

最重要的是要理解在什么场景，什么时机下触发交互动作

28.2.1. 上拉加载

首先可以看一张图



上拉加载的本质是页面触底，或者快要触底时的动作

判断页面触底我们需要先了解一下下面几个属性

- `scrollTop` ：滚动视窗的高度距离 `window` 顶部的距离，它会随着往上滚动而不断增加，初始

值是0，它是一个变化的值

- `clientHeight`:它是一个定值，表示屏幕可视区域的高度；
- `scrollHeight`：页面不能滚动时也是存在的,此时scrollHeight等于clientHeight。scrollHeight表示`body`所有元素的总长度(包括body元素自身的padding)

综上我们得出一个触底公式：

JavaScript 复制代码

```
scrollTop + clientHeight >= scrollHeight
```

简单实现

JavaScript 复制代码

```
let clientHeight  = document.documentElement.clientHeight; //浏览器高度
let scrollHeight = document.body.scrollHeight;
let scrollTop = document.documentElement.scrollTop;

let distance = 50;  //距离视窗还用50的时候，开始触发；

if ((scrollTop + clientHeight) >= (scrollHeight - distance)) {
    console.log("开始加载数据");
}
```

28.2.2. 下拉刷新

下拉刷新的本质是页面本身置于顶部时，用户下拉时需要触发的动作

关于下拉刷新的原生实现，主要分成三步：

- 监听原生`touchstart`事件，记录其初始位置的值，`e.touches[0].pageY`；
- 监听原生`touchmove`事件，记录并计算当前滑动的位置值与初始位置值的差值，大于`0`表示向下拉动，并借助CSS3的`translateY`属性使元素跟随手势向下滑动对应的差值，同时也应设置一个允许滑动的最大值；
- 监听原生`touchend`事件，若此时元素滑动达到最大值，则触发`callback`，同时将`translateY`重设为`0`，元素回到初始位置

举个例子：

`Html`结构如下：

JavaScript | 复制代码

```
<main>
    <p class="refreshText"></p >
    <ul id="refreshContainer">
        <li>111</li>
        <li>222</li>
        <li>333</li>
        <li>444</li>
        <li>555</li>
        ...
    </ul>
</main>
```

监听 `touchstart` 事件，记录初始的值

JavaScript | 复制代码

```
var _element = document.getElementById('refreshContainer'),
    _refreshText = document.querySelector('.refreshText'),
    _startPos = 0,  // 初始的值
    _transitionHeight = 0; // 移动的距离

_element.addEventListener('touchstart', function(e) {
    _startPos = e.touches[0].pageY; // 记录初始位置
    _element.style.position = 'relative';
    _element.style.transition = 'transform 0s';
}, false);
```

监听 `touchmove` 移动事件，记录滑动差值

JavaScript | 复制代码

```
_element.addEventListener('touchmove', function(e) {
    // e.touches[0].pageY 当前位置
    _transitionHeight = e.touches[0].pageY - _startPos; // 记录差值

    if (_transitionHeight > 0 && _transitionHeight < 60) {
        _refreshText.innerText = '下拉刷新';
        _element.style.transform = 'translateY('+_transitionHeight+'px)';

        if (_transitionHeight > 55) {
            _refreshText.innerText = '释放更新';
        }
    }
}, false);
```

最后，就是监听 `touchend` 离开的事件

```JavaScript
_element.addEventListener('touchend', function(e) {
    _element.style.transition = 'transform 0.5s ease 1s';
    _element.style.transform = 'translateY(0px)';
    _refreshText.innerText = '更新中...';
    // todo...

}, false);
```

从上面可以看到，在下拉到松手的过程中，经历了三个阶段：

- 当前手势滑动位置与初始位置差值大于零时，提示正在进行下拉刷新操作
- 下拉到一定值时，显示松手释放后的操作提示
- 下拉到达设定最大值松手时，执行回调，提示正在进行更新操作

28.3. 案例

在实际开发中，我们更多的是使用第三方库，下面以 `better-scroll` 进行举例：

HTML结构

```JavaScript
<div id="position-wrapper">
    <div>
        <p class="refresh">下拉刷新</p >
        <div class="position-list">
   <!--列表内容-->
        </div>
        <p class="more">查看更多</p >
    </div>
</div>
```

实例化上拉下拉插件，通过 `use` 来注册插件

JavaScript | 复制代码

```
import BScroll from "@better-scroll/core";
import PullDown from "@better-scroll/pull-down";
import PullUp from '@better-scroll/pull-up';
BScroll.use(PullDown);
BScroll.use(PullUp);
```

实例化 `BetterScroll`，并传入相关的参数

JavaScript | 复制代码

```javascript
let pageNo = 1,pageSize = 10,dataList = [],isMore = true;
var scroll= new BScroll("#position-wrapper",{
    scrollY:true,//垂直方向滚动
    click:true,//默认会阻止浏览器的原生click事件，如果需要点击，这里要设为true
    pullUpLoad:true,//上拉加载更多
    pullDownRefresh:{
        threshold:50,//触发pullingDown事件的位置
        stop:0//下拉回弹后停留的位置
    }
});
//监听下拉刷新
scroll.on("pullingDown",pullingDownHandler);
//监测实时滚动
scroll.on("scroll",scrollHandler);
//上拉加载更多
scroll.on("pullingUp",pullingUpHandler);

async function pullingDownHandler(){
    dataList=[];
    pageNo=1;
    isMore=true;
    $(".more").text("查看更多");
    await getlist();//请求数据
    scroll.finishPullDown();//每次下拉结束后，需要执行这个操作
    scroll.refresh();//当滚动区域的dom结构有变化时，需要执行这个操作
}
async function pullingUpHandler(){
    if(!isMore){
        $(".more").text("没有更多数据了");
        scroll.finishPullUp();//每次上拉结束后，需要执行这个操作
        return;
    }
    pageNo++;
    await this.getlist();//请求数据
    scroll.finishPullUp();//每次上拉结束后，需要执行这个操作
    scroll.refresh();//当滚动区域的dom结构有变化时，需要执行这个操作
}
function scrollHandler(){
    if(this.y>50) $('.refresh').text("松手开始加载");
    else $('.refresh').text("下拉刷新");
}
function getlist(){
    //返回的数据
    let result=....;
    dataList=dataList.concat(result);
```

```
        //判断是否已加载完
        if(result.length<pageSize) isMore=false;
        //将dataList渲染到html内容中
    }
```

注意点：

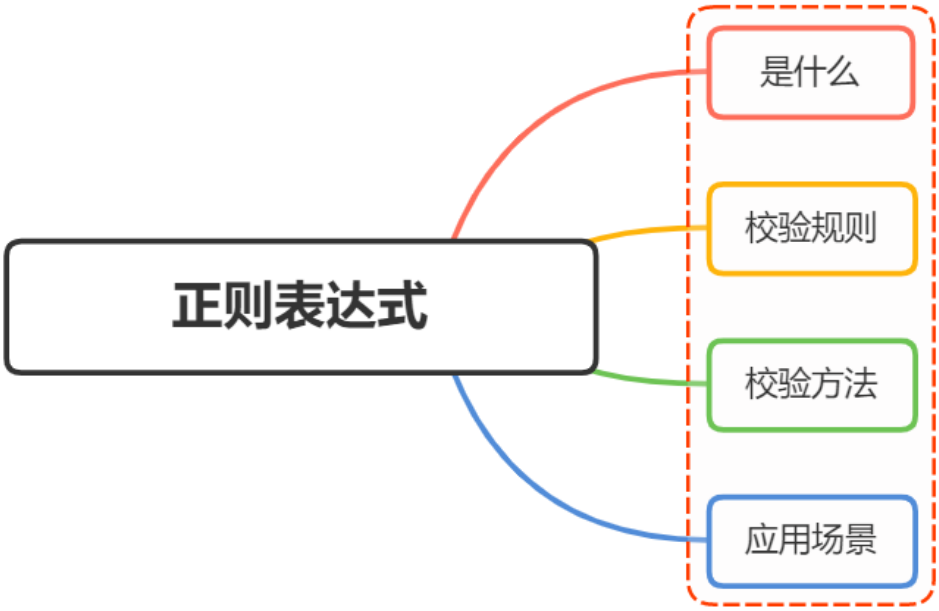
使用 `better-scroll` 实现下拉刷新、上拉加载时要注意以下几点：

- `wrapper` 里必须只有一个子元素
- 子元素的高度要比 `wrapper` 要高
- 使用的时候，要确定 `DOM` 元素是否已经生成，必须要等到 `DOM` 渲染完成后，再 `new BScroll()`
- 滚动区域的 `DOM` 元素结构有变化后，需要执行刷新 `refresh()`
- 上拉或者下拉，结束后，需要执行 `finishPullUp()` 或者 `finishPullDown()`，否则将不会执行下次操作
- `better-scroll`，默认会阻止浏览器的原生 `click` 事件，如果滚动内容区要添加点击事件，需要在实例化属性里设置 `click:true`

28.3.1. 小结

下拉刷新、上拉加载原理本身都很简单，真正复杂的是封装过程中，要考虑的兼容性、易用性、性能等诸多细节

29. 说说你对正则表达式的理解？应用场景？



29.1. 是什么

正则表达式是一种用来匹配字符串的强有力的武器

它的设计思想是用一种描述性的语言定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的

在 `JavaScript` 中，正则表达式也是对象，构建正则表达式有两种方式：

1. 字面量创建，其由包含在斜杠之间的模式组成

```JavaScript
const re = /\d+/g;
```

2. 调用 `RegExp` 对象的构造函数

```JavaScript
const re = new RegExp("\\d+","g");

const rul = "\\d+"
const re1 = new RegExp(rul,"g");
```

使用构建函数创建，第一个参数可以是一个变量，遇到特殊字符 `\` 需要使用 `\\` 进行转义

29.2. 匹配规则

常见的校验规则如下：

规则	描述
\	转义
^	匹配输入的开始
$	匹配输入的结束
*	匹配前一个表达式 0 次或多次
+	匹配前面一个表达式 1 次或者多次。等价于 `{1,}`
?	匹配前面一个表达式 0 次或者 1 次。等价于 `{0,1}`
.	默认匹配除换行符之外的任何单个字符
x(?=y)	匹配'x'仅仅当'x'后面跟着'y'。这种叫做先行断言
(?<=y)x	匹配'x'仅当'x'前面是'y'.这种叫做后行断言
x(?!y)	仅仅当'x'后面不跟着'y'时匹配'x'，这被称为正向否定查找
(?<!y)x	仅仅当'x'前面不是'y'时匹配'x'，这被称为反向否定查找
x|y	匹配‘x’或者‘y’
{n}	n 是一个正整数，匹配了前面一个字符刚好出现了 n 次
{n,}	n是一个正整数，匹配前一个字符至少出现了n次
{n,m}	n 和 m 都是整数。匹配前面的字符至少n次，最多m次
[xyz]	一个字符集合。匹配方括号中的任意字符
[^xyz]	匹配任何没有包含在方括号中的字符
\b	匹配一个词的边界，例如在字母和空格之间
\B	匹配一个非单词边界

\d	匹配一个数字
\D	匹配一个非数字字符
\f	匹配一个换页符
\n	匹配一个换行符
\r	匹配一个回车符
\s	匹配一个空白字符，包括空格、制表符、换页符和换行符
\S	匹配一个非空白字符
\w	匹配一个单字字符（字母、数字或者下划线）
\W	匹配一个非单字字符

29.2.1. 正则表达式标记

标志	描述
`g`	全局搜索。
`i`	不区分大小写搜索。
`m`	多行搜索。
`s`	允许 `.` 匹配换行符。
`u`	使用 `unicode` 码的模式进行匹配。
`y`	执行“粘性(`sticky`)”搜索,匹配从目标字符串的当前位置开始。

使用方法如下：

```JavaScript
var re = /pattern/flags;
var re = new RegExp("pattern", "flags");
```

在了解下正则表达式基本的之外，还可以掌握几个正则表达式的特性：

29.2.2. 贪婪模式

在了解贪婪模式前，首先举个例子：

```JavaScript
const reg = /ab{1,3}c/
```

在匹配过程中，尝试可能的顺序是从多往少的方向去尝试。首先会尝试 `bbb` ，然后再看整个正则是否能匹配。不能匹配时，吐出一个 `b` ，即在 `bb` 的基础上，再继续尝试，以此重复

如果多个贪婪量词挨着，则深度优先搜索

```JavaScript
const string = "12345";
const regx = /(\d{1,3})(\d{1,3})/;
console.log( string.match(reg) );
// => ["12345", "123", "45", index: 0, input: "12345"]
```

其中，前面的 `\d{1,3}` 匹配的是"123"，后面的 `\d{1,3}` 匹配的是"45"

29.2.3. 懒惰模式

惰性量词就是在贪婪量词后面加个问号。表示尽可能少的匹配

```JavaScript
var string = "12345";
var regex = /(\d{1,3}?)(\d{1,3})/;
console.log( string.match(regex) );
// => ["1234", "1", "234", index: 0, input: "12345"]
```

其中 `\d{1,3}?` 只匹配到一个字符"1"，而后面的 `\d{1,3}` 匹配了"234"

29.2.4. 分组

分组主要是用过 `()` 进行实现，比如 `beyond{3}` ，是匹配 `d` 字母3次。而 `(beyond){3}` 是匹配 `beyond` 三次

在 `()` 内使用 `|` 达到或的效果，如 `(abc | xxx)` 可以匹配 `abc` 或者 `xxx`

反向引用，巧用 `$` 分组捕获

JavaScript | 复制代码

```
let str = "John Smith";

// 交换名字和姓氏
console.log(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

29.3. 匹配方法

正则表达式常被用于某些方法，我们可以分成两类：

- 字符串（str）方法：`match` 、 `matchAll` 、 `search` 、 `replace` 、 `split`
- 正则对象下（regexp）的方法： `test` 、 `exec`

方法	描述
exec	一个在字符串中执行查找匹配的RegExp方法，它返回一个数组（未匹配到则返回 null）。
test	一个在字符串中测试是否匹配的RegExp方法，它返回 true 或 false。
match	一个在字符串中执行查找匹配的String方法，它返回一个数组，在未匹配到时会返回 null。
matchAll	一个在字符串中执行查找所有匹配的String方法，它返回一个迭代器（iterator）。
search	一个在字符串中测试匹配的String方法，它返回匹配到的位置索引，或者在失败时返回-1。
replace	一个在字符串中执行查找匹配的String方法，并且使用替换字符串替换掉匹配到的子字符串。

split	一个使用正则表达式或者一个固定字符串分隔一个字符串，并将分隔后的子字符串存储到数组中的 `String` 方法。

29.3.1. str.match(regexp)

`str.match(regexp)` 方法在字符串 `str` 中找到匹配 `regexp` 的字符

如果 `regexp` 不带有 `g` 标记，则它以数组的形式返回第一个匹配项，其中包含分组和属性 `index`（匹配项的位置）、`input`（输入字符串，等于 `str`）

JavaScript | 复制代码

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/);

console.log( result[0] );     // JavaScript（完全匹配）
console.log( result[1] );     // Script（第一个分组）
console.log( result.length ); // 2

// 其他信息：
console.log( result.index );  // 7（匹配位置）
console.log( result.input );  // I love JavaScript（源字符串）
```

如果 `regexp` 带有 `g` 标记，则它将所有匹配项的数组作为字符串返回，而不包含分组和其他详细信息

JavaScript | 复制代码

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/g);

console.log( result[0] ); // JavaScript
console.log( result.length ); // 1
```

如果没有匹配项，则无论是否带有标记 `g` ，都将返回 `null`

```JavaScript
let str = "I love JavaScript";

let result = str.match(/HTML/);

console.log(result); // null
```

29.3.2. str.matchAll(regexp)

返回一个包含所有匹配正则表达式的结果及分组捕获组的迭代器

```JavaScript
const regexp = /t(e)(st(\d?))/g;
const str = 'test1test2';

const array = [...str.matchAll(regexp)];

console.log(array[0]);
// expected output: Array ["test1", "e", "st1", "1"]

console.log(array[1]);
// expected output: Array ["test2", "e", "st2", "2"]
```

29.3.3. str.search(regexp)

返回第一个匹配项的位置，如果未找到，则返回 `-1`

```JavaScript
let str = "A drop of ink may make a million think";

console.log( str.search( /ink/i ) ); // 10（第一个匹配位置）
```

这里需要注意的是， `search` 仅查找第一个匹配项

29.4. str.replace(regexp)

替换与正则表达式匹配的子串，并返回替换后的字符串。在不设置全局匹配 `g` 的时候，只替换第一个匹配成功的字符串片段

```JavaScript
const reg1=/javascript/i;
const reg2=/javascript/ig;
console.log('hello Javascript Javascript Javascript'.replace(reg1,'js'));
//hello js Javascript Javascript
console.log('hello Javascript Javascript Javascript'.replace(reg2,'js'));
//hello js js js
```

29.4.1. str.split(regexp)

使用正则表达式（或子字符串）作为分隔符来分割字符串

```JavaScript
console.log('12, 34, 56'.split(/,\s*/)) // 数组 ['12', '34', '56']
```

29.4.2. regexp.exec(str)

`regexp.exec(str)` 方法返回字符串 `str` 中的 `regexp` 匹配项，与以前的方法不同，它是在正则表达式而不是字符串上调用的

根据正则表达式是否带有标志 `g`，它的行为有所不同

如果没有 `g`，那么 `regexp.exec(str)` 返回的第一个匹配与 `str.match(regexp)` 完全相同

如果有标记 `g`，调用 `regexp.exec(str)` 会返回第一个匹配项，并将紧随其后的位置保存在属性 `regexp.lastIndex` 中。 下一次同样的调用会从位置 `regexp.lastIndex` 开始搜索，返回下一个匹配项，并将其后的位置保存在 `regexp.lastIndex` 中

```JavaScript
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  console.log( `Found ${result[0]} at position ${result.index}` );
  // Found JavaScript at position 11
  // Found javascript at position 33
}
```

29.4.3. regexp.test(str)

查找匹配项，然后返回 `true/false` 表示是否存在

```JavaScript
let str = "I love JavaScript";

// 这两个测试相同
console.log( /love/i.test(str) ); // true
```

29.5. 应用场景

通过上面的学习，我们对正则表达式有了一定的了解

下面再来看看正则表达式一些案例场景：

验证QQ合法性（5~15位、全是数字、不以0开头）：

```JavaScript
const reg = /^[1-9][0-9]{4,14}$/
const isvalid = patrn.exec(s)
```

校验用户账号合法性（只能输入5-20个以字母开头、可带数字、“_”、“.”的字串）：

```JavaScript
var patrn=/^[a-zA-Z]{1}([a-zA-Z0-9]|[._]){4,19}$/;
const isvalid = patrn.exec(s)
```

将 `url` 参数解析为对象

JavaScript | 复制代码

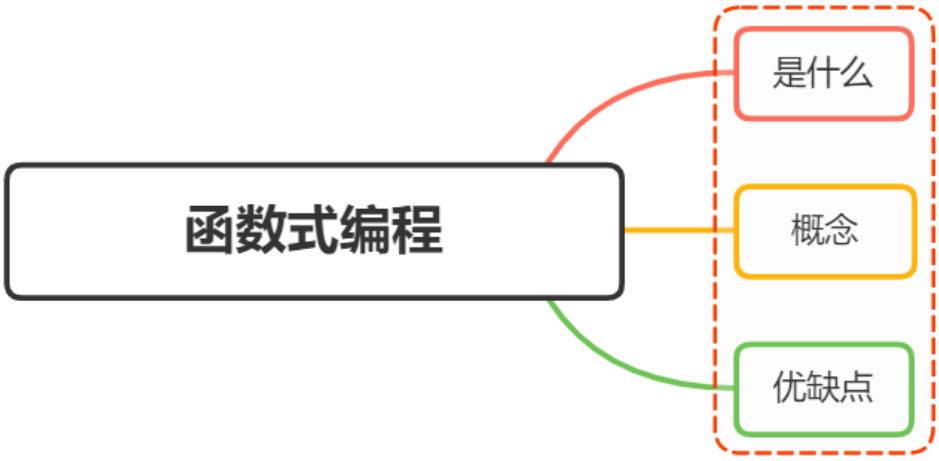
```javascript
const protocol = '(?<protocol>https?:)';
const host = '(?<host>(?<hostname>[^/#?:]+)(?::(?<port>\\d+))?)';
const path = '(?<pathname>(?:\\/[^/#?]+)*\\/?)';
const search = '(?<search>(?:\\?[^#]*)?)';
const hash = '(?<hash>(?:#.*)?)';
const reg = new RegExp(`^${protocol}\/\/${host}${path}${search}${hash}$`);
function execURL(url){
    const result = reg.exec(url);
    if(result){
        result.groups.port = result.groups.port || '';
        return result.groups;
    }
    return {
        protocol:'',host:'',hostname:'',port:'',
        pathname:'',search:'',hash:'',
    };
}

console.log(execURL('https://localhost:8080/?a=b#xxxx'));
protocol: "https:"
host: "localhost:8080"
hostname: "localhost"
port: "8080"
pathname: "/"
search: "?a=b"
hash: "#xxxx"
```

再将上面的 `search` 和 `hash` 进行解析

```javascript
function execUrlParams(str){
    str = str.replace(/^[#?&]/,'');
    const result = {};
    if(!str){ //如果正则可能配到空字符串，极有可能造成死循环，判断很重要
        return result;
    }
    const reg = /(?:^|&)([^&=]*)=?([^&]*?)(?=&|$)/y
    let exec = reg.exec(str);
    while(exec){
        result[exec[1]] = exec[2];
        exec = reg.exec(str);
    }
    return result;
}
console.log(execUrlParams('#'));// {}
console.log(execUrlParams('##'));//{'#':''}
console.log(execUrlParams('?q=3606&src=srp')); //{q: "3606", src: "srp"}
console.log(execUrlParams('test=a=b=c&&==&a='));//{test: "a=b=c", "":
"=", a: ""}
```

30. 说说你对函数式编程的理解？优缺点？



30.1. 是什么

函数式编程是一种"编程范式"（programming paradigm），一种编写程序的方法论

主要的编程范式有三种：命令式编程，声明式编程和函数式编程