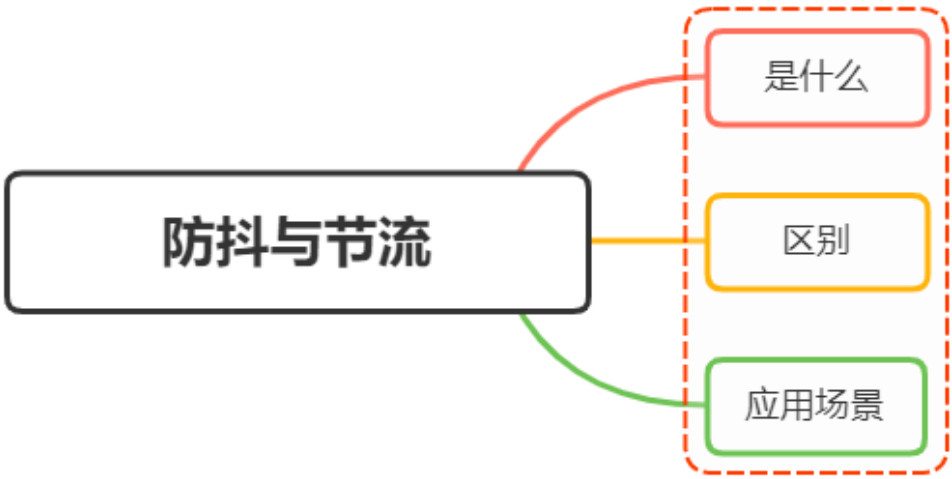
```javascript
ajax({
    type: 'post',
    dataType: 'json',
    data: {},
    url: 'https://xxxx',
    success: function(text,xml){//请求成功后的回调函数
        console.log(text)
    },
    fail: function(status){////请求失败后的回调函数
        console.log(status)
    }
})
```

25. 什么是防抖和节流？有什么区别？如何实现？



25.1. 是什么

本质上是优化高频率执行代码的一种手段

如：浏览器的 `resize` 、 `scroll` 、 `keypress` 、 `mousemove` 等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能

为了优化体验，需要对这类事件进行调用次数的限制，对此我们就可以采用 **防抖（debounce）** 和 **节流（throttle）** 的方式来减少调用频率

25.1.1. 定义

- 节流: n 秒内只运行一次，若在 n 秒内重复触发，只有一次生效
- 防抖: n 秒后在执行该事件，若在 n 秒内被重复触发，则重新计时

一个经典的比喻:

想象每天上班大厦底下的电梯。把电梯完成一次运送，类比为一次函数的执行和响应

假设电梯有两种运行策略 `debounce` 和 `throttle`，超时设定为15秒，不考虑容量限制

电梯第一个人进来后，15秒后准时运送一次，这是节流

电梯第一个人进来后，等待15秒。如果过程中又有人进来，15秒等待重新计时，直到15秒后开始运送，这是防抖

25.2. 代码实现

25.2.1. 节流

完成节流可以使用时间戳与定时器的写法

使用时间戳写法，事件会立即执行，停止触发后没有办法再次执行

JavaScript | 复制代码

```javascript
function throttled1(fn, delay = 500) {
    let oldtime = Date.now()
    return function (...args) {
        let newtime = Date.now()
        if (newtime - oldtime >= delay) {
            fn.apply(null, args)
            oldtime = Date.now()
        }
    }
}
```

使用定时器写法，`delay` 毫秒后第一次执行，第二次事件停止触发后依然会再一次执行

JavaScript | 复制代码

```javascript
function throttled2(fn, delay = 500) {
    let timer = null
    return function (...args) {
        if (!timer) {
            timer = setTimeout(() => {
                fn.apply(this, args)
                timer = null
            }, delay);
        }
    }
}
```

可以将时间戳写法的特性与定时器写法的特性相结合，实现一个更加精确的节流。实现如下

JavaScript | 复制代码

```javascript
function throttled(fn, delay) {
    let timer = null
    let starttime = Date.now()
    return function () {
        let curTime = Date.now() // 当前时间
        let remaining = delay - (curTime - starttime)  // 从上一次到现在，还剩下多少多余时间
        let context = this
        let args = arguments
        clearTimeout(timer)
        if (remaining <= 0) {
            fn.apply(context, args)
            starttime = Date.now()
        } else {
            timer = setTimeout(fn, remaining);
        }
    }
}
```

25.2.2. 防抖

简单版本的实现

JavaScript | 复制代码

```javascript
function debounce(func, wait) {
    let timeout;

    return function () {
        let context = this; // 保存this指向
        let args = arguments; // 拿到event对象

        clearTimeout(timeout)
        timeout = setTimeout(function(){
            func.apply(context, args)
        }, wait);
    }
}
```

防抖如果需要立即执行，可加入第三个参数用于判断，实现如下：

JavaScript | 复制代码

```javascript
function debounce(func, wait, immediate) {

    let timeout;

    return function () {
        let context = this;
        let args = arguments;

        if (timeout) clearTimeout(timeout); // timeout 不为null
        if (immediate) {
            let callNow = !timeout; // 第一次会立即执行，以后只有事件执行后才会再次触发
            timeout = setTimeout(function () {
                timeout = null;
            }, wait)
            if (callNow) {
                func.apply(context, args)
            }
        }
        else {
            timeout = setTimeout(function () {
                func.apply(context, args)
            }, wait);
        }
    }
}
```

25.3. 区别

相同点：

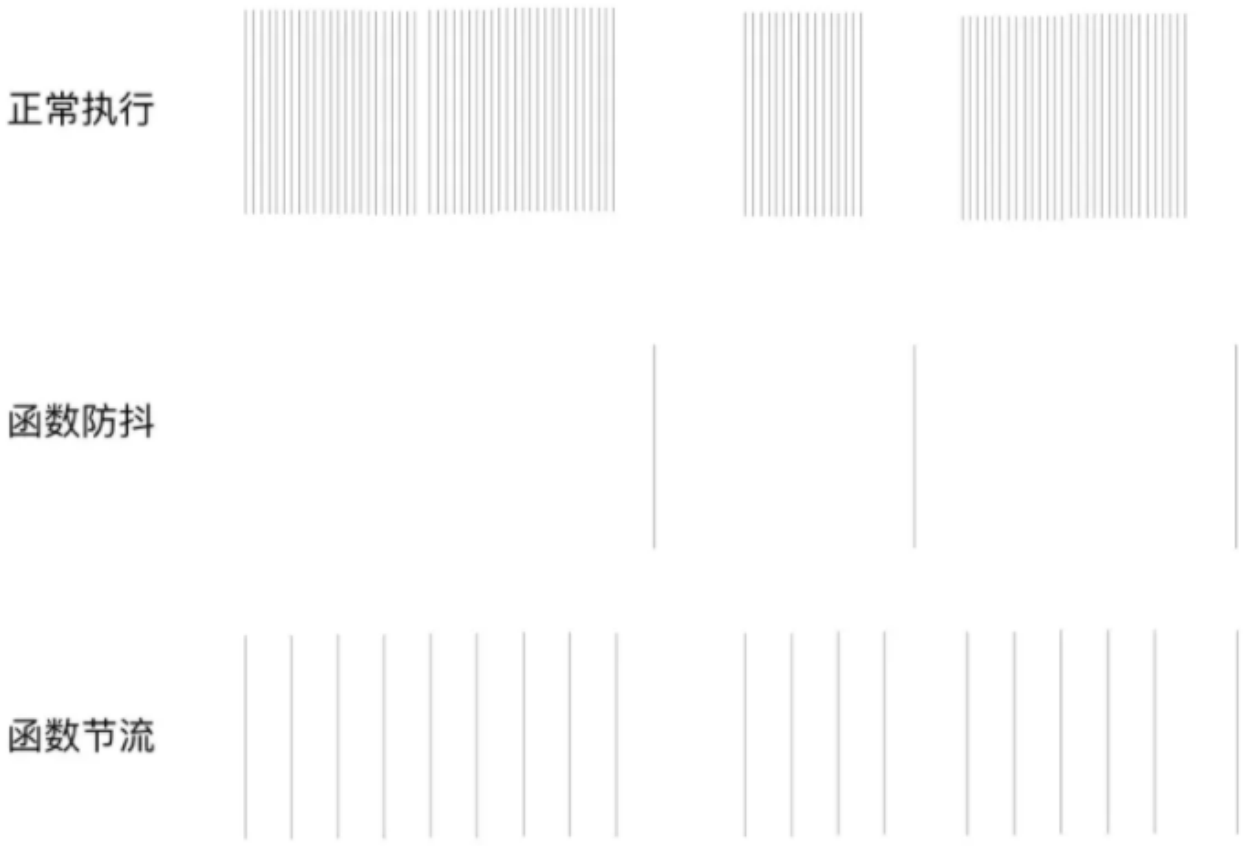
- 都可以通过使用 `setTimeout` 实现
- 目的都是，降低回调执行频率。节省计算资源

不同点：

- 函数防抖，在一段连续操作结束后，处理回调，利用 `clearTimeout` 和 `setTimeout` 实现。函数节流，在一段连续操作中，每一段时间只执行一次，频率较高的事件中使用来提高性能
- 函数防抖关注一定时间连续触发的事件，只在最后执行一次，而函数节流一段时间内只执行一次

例如，都设置时间频率为500ms，在2秒时间内，频繁触发函数，节流，每隔 500ms 就执行一次。防抖，则不管调动多少次方法，在2s后，只会执行一次

如下图所示：



25.4. 应用场景

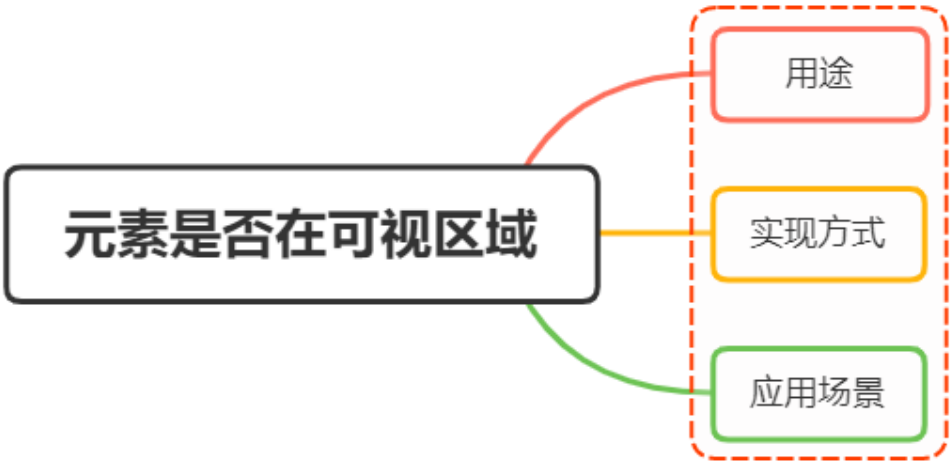
防抖在连续的事件，只需触发一次回调的场景有：

- 搜索框搜索输入。只需用户最后一次输入完，再发送请求
- 手机号、邮箱验证输入检测
- 窗口大小 `resize` 。只需窗口调整完成后，计算窗口大小。防止重复渲染。

节流在间隔一段时间执行一次回调的场景有：

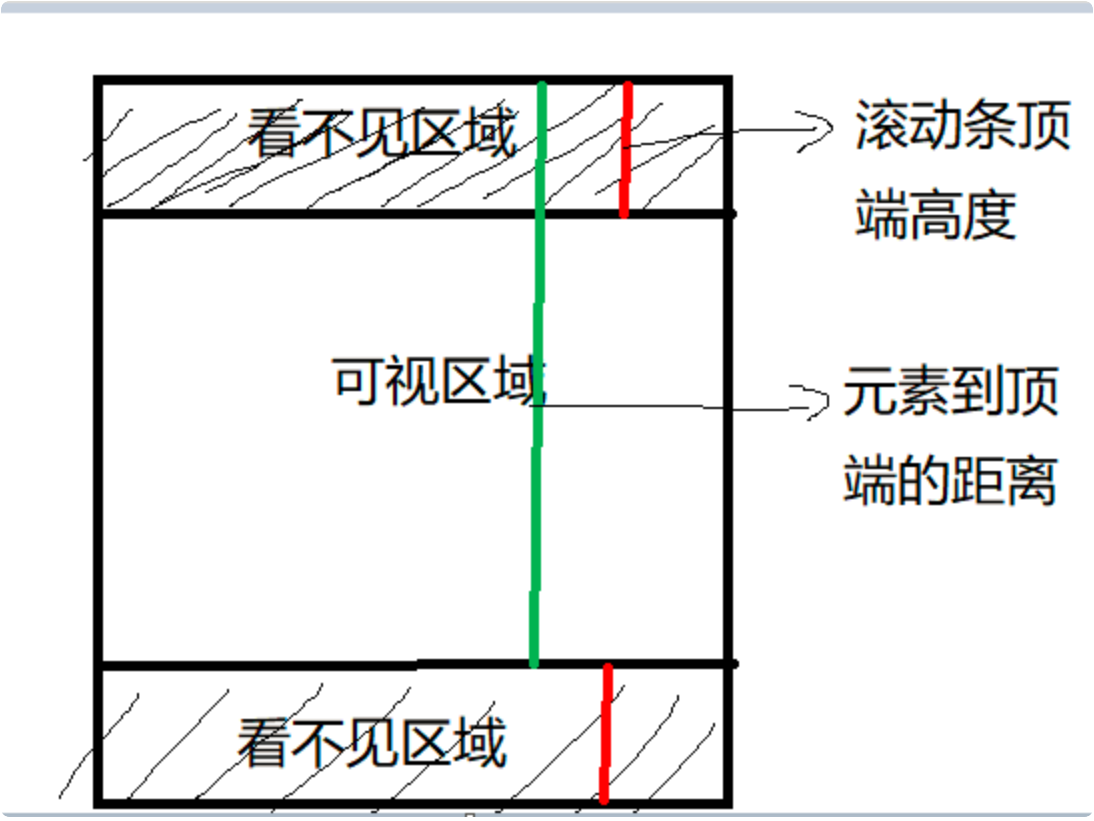
- 滚动加载，加载更多或滚到底部监听
- 搜索框，搜索联想功能

26. 如何判断一个元素是否在可视区域中?



26.1. 用途

可视区域即我们浏览网页的设备肉眼可见的区域，如下图



在日常开发中，我们经常需要判断目标元素是否在视窗之内或者和视窗的距离小于一个值（例如 100 px），从而实现一些常用的功能，例如：

- 图片的懒加载
- 列表的无限滚动
- 计算广告元素的曝光情况
- 可点击链接的预加载

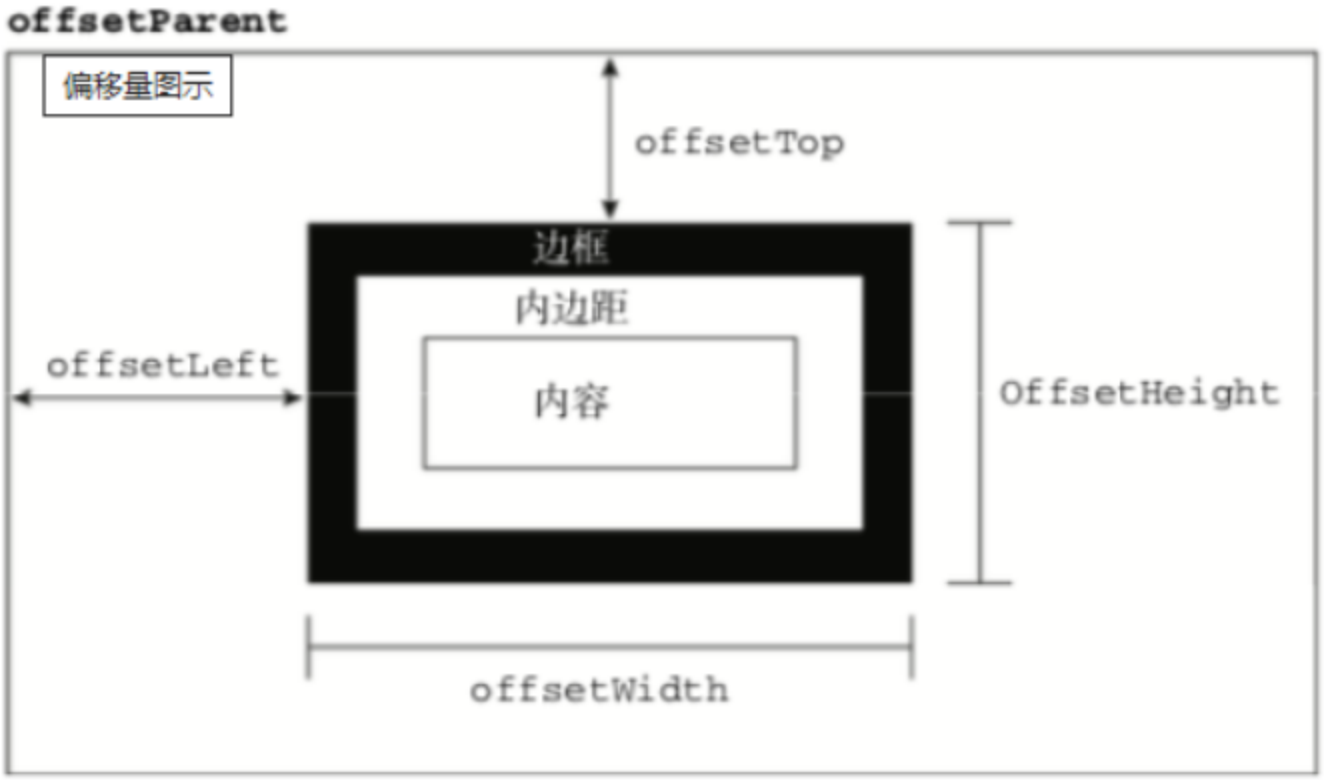
26.2. 实现方式

判断一个元素是否在可视区域，我们常用的有三种办法：

- offsetTop、scrollTop
- getBoundingClientRect
- Intersection Observer

26.2.1. offsetTop、scrollTop

`offsetTop`，元素的上外边框至包含元素的上内边框之间的像素距离，其他 `offset` 属性如下图所示：



下面再来了解下 `clientWidth` 、 `clientHeight` ：

- `clientWidth` ： 元素内容区宽度加上左右内边距宽度，即 `clientWidth = content + padding`
- `clientHeight` ： 元素内容区高度加上上下内边距高度，即 `clientHeight = content + padding`

这里可以看到 `client` 元素都不包括外边距

最后，关于 `scroll` 系列的属性如下：

- `scrollWidth` 和 `scrollHeight` 主要用于确定元素内容的实际大小
- `scrollLeft` 和 `scrollTop` 属性既可以确定元素当前滚动的状态，也可以设置元素的滚动位置
  - 垂直滚动 `scrollTop > 0`
  - 水平滚动 `scrollLeft > 0`
- 将元素的 `scrollLeft` 和 `scrollTop` 设置为 0，可以重置元素的滚动位置

26.2.1.1. 注意

- 上述属性都是只读的，每次访问都要重新开始

下面再看看如何实现判断：

公式如下：

```JavaScript
el.offsetTop - document.documentElement.scrollTop <= viewPortHeight
```

代码实现：

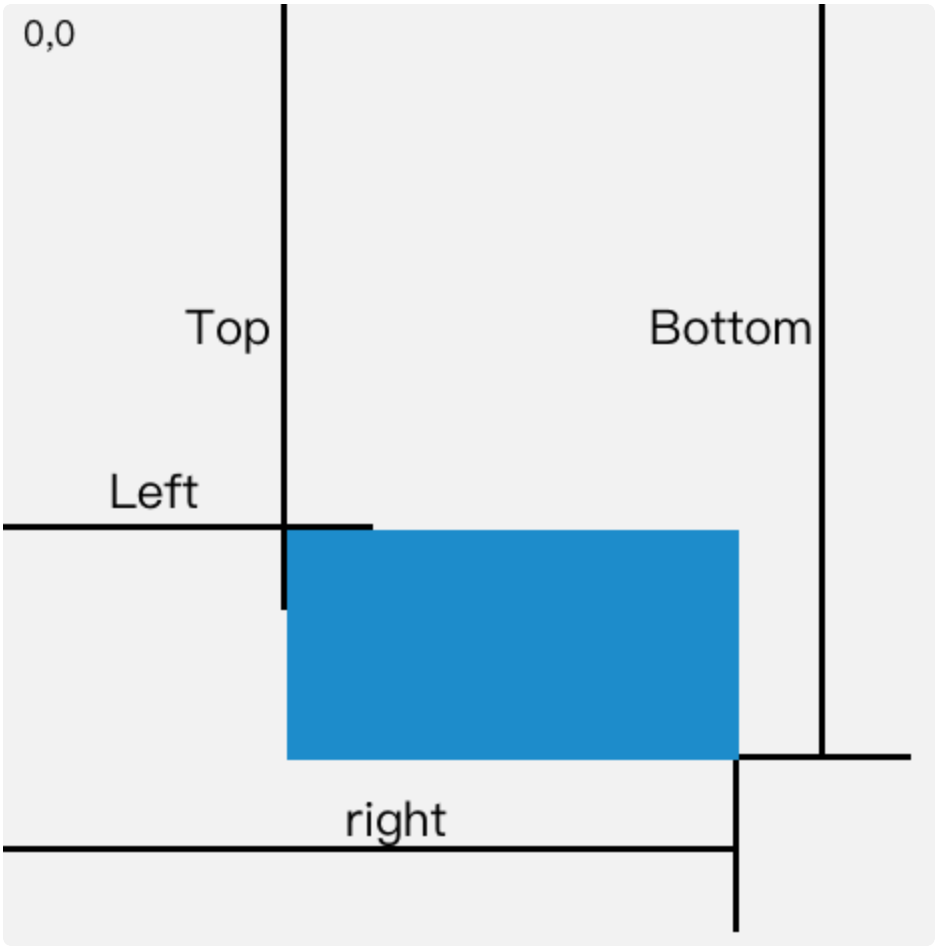
```JavaScript
function isInViewPortOfOne (el) {
    // viewPortHeight 兼容所有浏览器写法
    const viewPortHeight = window.innerHeight || document.documentElement.clientHeight || document.body.clientHeight
    const offsetTop = el.offsetTop
    const scrollTop = document.documentElement.scrollTop
    const top = offsetTop - scrollTop
    return top <= viewPortHeight
}
```

26.2.2. getBoundingClientRect

返回值是一个 `DOMRect` 对象，拥有 `left`，`top`，`right`，`bottom`，`x`，`y`，`width`，和 `height` 属性

```JavaScript
const target = document.querySelector('.target');
const clientRect = target.getBoundingClientRect();
console.log(clientRect);

// {
//   bottom: 556.21875,
//   height: 393.59375,
//   left: 333,
//   right: 1017,
//   top: 162.625,
//   width: 684
// }
```

属性对应的关系图如下所示：



当页面发生滚动的时候， `top` 与 `left` 属性值都会随之改变

如果一个元素在视窗之内的话，那么它一定满足下面四个条件：

- top 大于等于 0
- left 大于等于 0
- bottom 小于等于视窗高度
- right 小于等于视窗宽度

实现代码如下：

```javascript
function isInViewPort(element) {
  const viewWidth = window.innerWidth || document.documentElement.clientWidth;
  const viewHeight = window.innerHeight || document.documentElement.clientHeight;
  const {
    top,
    right,
    bottom,
    left,
  } = element.getBoundingClientRect();

  return (
    top >= 0 &&
    left >= 0 &&
    right <= viewWidth &&
    bottom <= viewHeight
  );
}
```

26.2.3. Intersection Observer

`Intersection Observer` 即重叠观察者，从这个命名就可以看出它用于判断两个元素是否重叠，因为不用进行事件的监听，性能方面相比 `getBoundingClientRect` 会好很多

使用步骤主要分为两步：创建观察者和传入被观察者

26.2.3.1. 创建观察者

```javascript
const options = {
  // 表示重叠面积占被观察者的比例，从 0 - 1 取值，
  // 1 表示完全被包含
  threshold: 1.0,
  root:document.querySelector('#scrollArea') // 必须是目标元素的父级元素
};

const callback = (entries, observer) => { ....}

const observer = new IntersectionObserver(callback, options);
```

通过 `new IntersectionObserver` 创建了观察者 `observer`，传入的参数 `callback` 在重叠比例超过 `threshold` 时会被执行`

关于 `callback` 回调函数常用属性如下：

JavaScript | 复制代码

```
// 上段代码中被省略的 callback
const callback = function(entries, observer) {
    entries.forEach(entry => {
        entry.time;               // 触发的时间
        entry.rootBounds;         // 根元素的位置矩形，这种情况下为视窗位置
        entry.boundingClientRect; // 被观察者的位置举行
        entry.intersectionRect;   // 重叠区域的位置矩形
        entry.intersectionRatio;  // 重叠区域占被观察者面积的比例（被观察者不是矩形时也按照矩形计算）
        entry.target;             // 被观察者
    });
};
```

26.2.3.2. 传入被观察者

通过 `observer.observe(target)` 这一行代码即可简单的注册被观察者

JavaScript | 复制代码

```
const target = document.querySelector('.target');
observer.observe(target);
```

26.2.4. 案例分析

实现：创建了一个十万个节点的长列表，当节点滚入到视窗中时，背景就会从红色变为黄色

`Html` 结构如下：

JavaScript | 复制代码

```
<div class="container"></div>
```

`css` 样式如下：

```css
.container {
    display: flex;
    flex-wrap: wrap;
}
.target {
    margin: 5px;
    width: 20px;
    height: 20px;
    background: red;
}
```

往 `container` 插入1000个元素

```javascript
const $container = $(".container");

// 插入 100000 个 <div class="target"></div>
function createTargets() {
  const htmlString = new Array(100000)
    .fill('<div class="target"></div>')
    .join("");
  $container.html(htmlString);
}
```

这里，首先使用 `getBoundingClientRect` 方法进行判断元素是否在可视区域

```javascript
function isInViewPort(element) {
    const viewWidth = window.innerWidth || document.documentElement.clientWidth;
    const viewHeight =
          window.innerHeight || document.documentElement.clientHeight;
    const { top, right, bottom, left } = element.getBoundingClientRect();

    return top >= 0 && left >= 0 && right <= viewWidth && bottom <= viewHeight;
}
```

然后开始监听 `scroll` 事件，判断页面上哪些元素在可视区域中，如果在可视区域中则将背景颜色设置为 `yellow`

```javascript
$(window).on("scroll", () => {
    console.log("scroll !");
    $targets.each((index, element) => {
        if (isInViewPort(element)) {
            $(element).css("background-color", "yellow");
        }
    });
});
```

通过上述方式，可以看到可视区域颜色会变成黄色了，但是可以明显看到有卡顿的现象，原因在于我们绑定了 `scroll` 事件， `scroll` 事件伴随了大量的计算，会造成资源方面的浪费

下面通过 `Intersection Observer` 的形式同样实现相同的功能

首先创建一个观察者

```javascript
const observer = new IntersectionObserver(getYellow, { threshold: 1.0 });
```

`getYellow` 回调函数实现对背景颜色改变，如下：

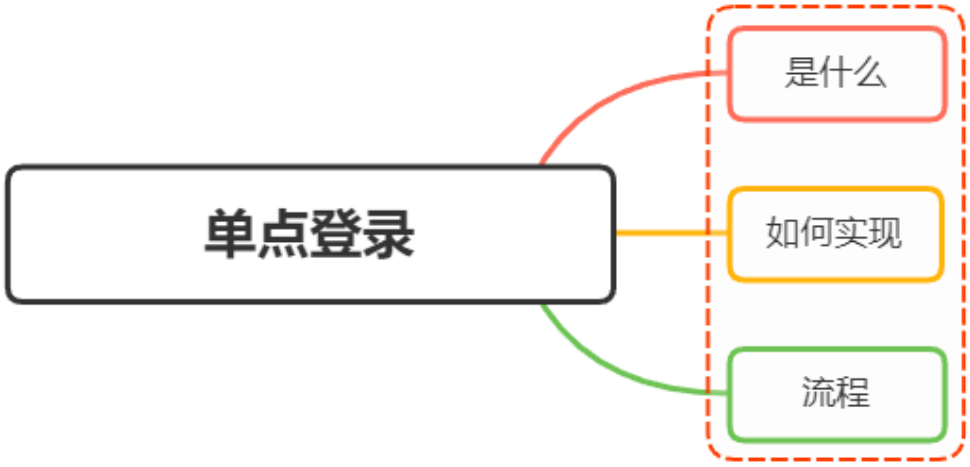
```javascript
function getYellow(entries, observer) {
    entries.forEach(entry => {
        $(entry.target).css("background-color", "yellow");
    });
}
```

最后传入观察者，即 `.target` 元素

```javascript
$targets.each((index, element) => {
    observer.observe(element);
});
```

可以看到功能同样完成，并且页面不会出现卡顿的情况

27. 什么是单点登录？如何实现？



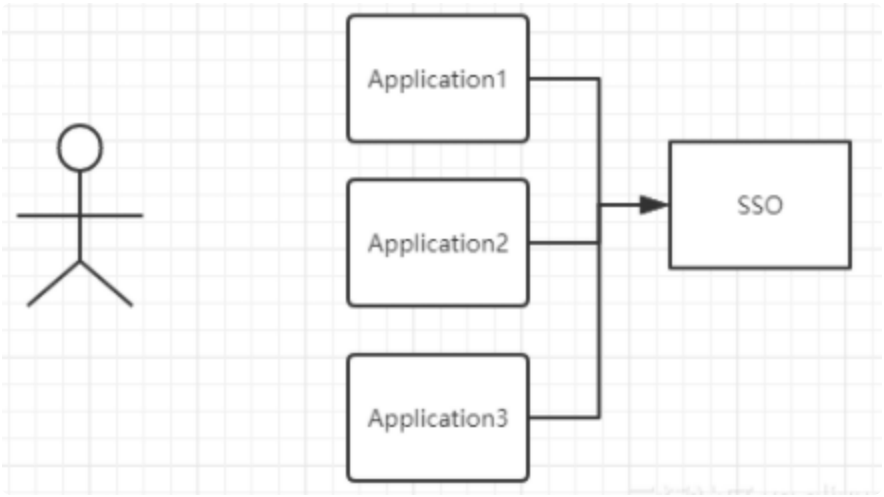
27.1. 是什么

单点登录（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一

SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统

SSO 一般都需要一个独立的认证中心（passport），子系统的登录均得通过 `passport` ，子系统本身将不参与登录操作

当一个系统成功登录以后， `passport` 将会颁发一个令牌给各个子系统，子系统可以拿着令牌会获取各自的受保护资源，为了减少频繁认证，各个子系统在被 `passport` 授权以后，会建立一个局部会话，在一定时间内可以无需再次向 `passport` 发起认证



上图有四个系统，分别是 `Application1` 、 `Application2` 、 `Application3` 、和 `SSO` ，当 `Application1` 、 `Application2` 、 `Application3` 需要登录时，将跳到 `SSO` 系统， `SSO` 系统完成登录，其他的应用系统也就随之登录了

27.1.1. 举个例子

淘宝、天猫都属于阿里旗下，当用户登录淘宝后，再打开天猫，系统便自动帮用户登录了天猫，这种现象就属于单点登录

27.2. 如何实现

27.2.1. 同域名下的单点登录

`cookie` 的 `domain` 属性设置为当前域的父域，并且父域的 `cookie` 会被子域所共享。 `path` 属性默认为 `web` 应用的上下文路径

利用 `Cookie` 的这个特点，没错，我们只需要将 `Cookie` 的 `domain` 属性设置为父域的域名（主域名），同时将 `Cookie` 的 `path` 属性设置为根路径，将 `Session ID` （或 `Token` ）保存到父域中。这样所有的子域应用就都可以访问到这个 `Cookie`

不过这要求应用系统的域名需建立在一个共同的主域名之下，如 `tieba.baidu.com` 和 `map.baidu.com` ，它们都建立在 `baidu.com` 这个主域名之下，那么它们就可以通过这种方式来实现单点登录

27.2.2. 不同域名下的单点登录(一)

如果是不同域的情况下， `Cookie` 是不共享的，这里我们可以部署一个认证中心，用于专门处理登录请求的独立的 `Web` 服务

用户统一在认证中心进行登录，登录成功后，认证中心记录用户的登录状态，并将 `token` 写入 `Cookie` （注意这个 `Cookie` 是认证中心的，应用系统是访问不到的）

应用系统检查当前请求有没有 `Token` ，如果没有，说明用户在当前系统中尚未登录，那么就将页面跳转至认证中心

由于这个操作会将认证中心的 `Cookie` 自动带过去，因此，认证中心能够根据 `Cookie` 知道用户是否已经登录过了

如果认证中心发现用户尚未登录，则返回登录页面，等待用户登录

如果发现用户已经登录过了，就不会让用户再次登录了，而是会跳转回目标 `URL` ，并在跳转前生成一个 `Token` ，拼接在目标 `URL` 的后面，回传给目标应用系统

应用系统拿到 `Token` 之后，还需要向认证中心确认下 `Token` 的合法性，防止用户伪造。确认无误后，应用系统记录用户的登录状态，并将 `Token` 写入 `Cookie` ，然后给本次访问放行。（注意这个 `Cookie` 是当前应用系统的）当用户再次访问当前应用系统时，就会自动带上这个 `Token` ，应用系统验证 Token 发现用户已登录，于是就不会有认证中心什么事了

此种实现方式相对复杂，支持跨域，扩展性好，是单点登录的标准做法

27.2.3. 不同域名下的单点登录(二)

可以选择将 `Session ID` （或 `Token` ）保存到浏览器的 `LocalStorage` 中，让前端在每次向后端发送请求时，主动将 `LocalStorage` 的数据传递给服务端

这些都是由前端来控制的，后端需要做的仅仅是在用户登录成功后，将 `Session ID` （或 `Token` ）放在响应体中传递给前端

单点登录完全可以在前端实现。前端拿到 `Session ID` （或 `Token` ）后，除了将它写入自己的 `LocalStorage` 中之外，还可以通过特殊手段将它写入多个其他域下的 `LocalStorage` 中

关键代码如下：

JavaScript | 复制代码

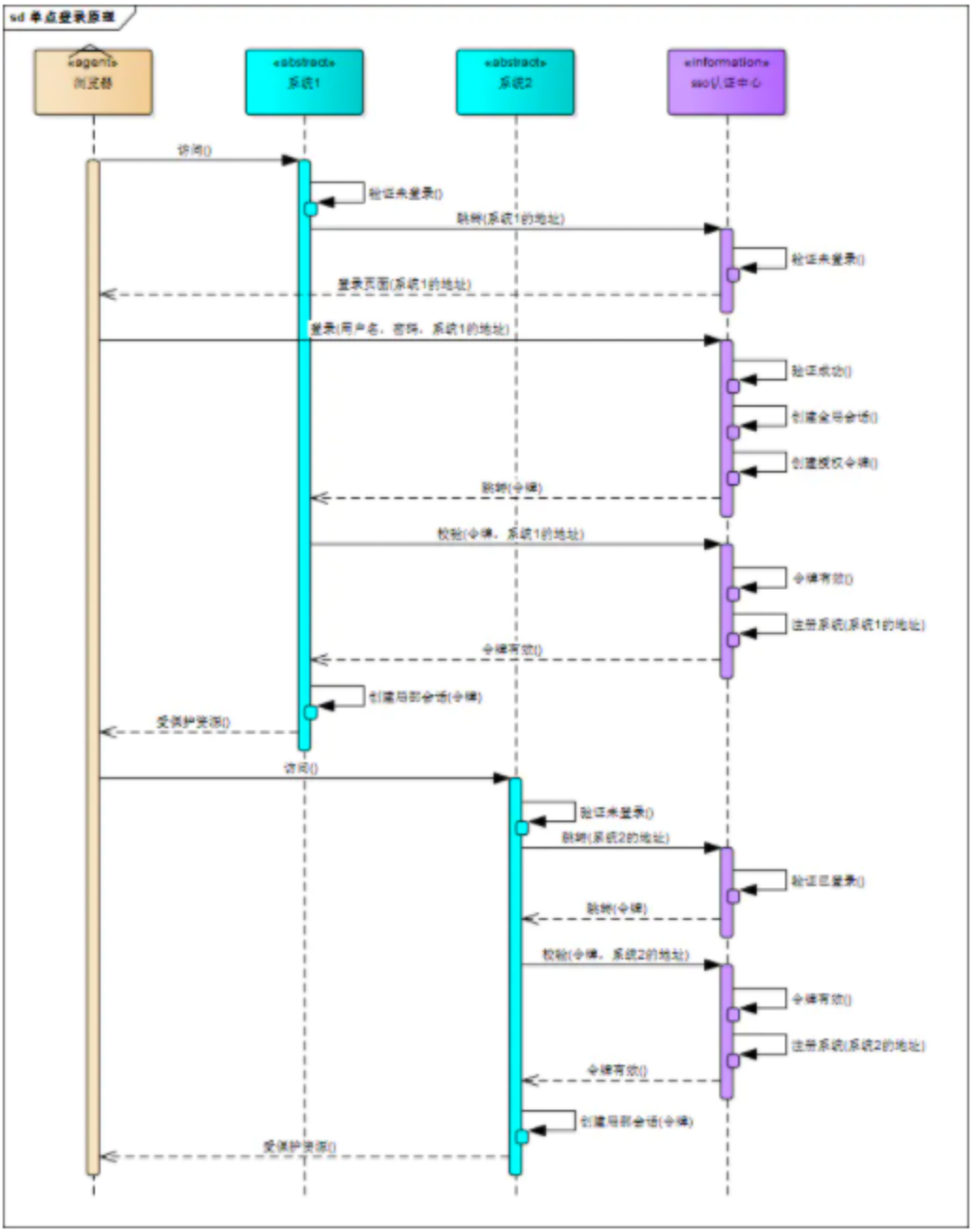
```javascript
// 获取 token
var token = result.data.token;

// 动态创建一个不可见的iframe，在iframe中加载一个跨域HTML
var iframe = document.createElement("iframe");
iframe.src = "http://app1.com/localstorage.html";
document.body.append(iframe);
// 使用postMessage()方法将token传递给iframe
setTimeout(function () {
    iframe.contentWindow.postMessage(token, "http://app1.com");
}, 4000);
setTimeout(function () {
    iframe.remove();
}, 6000);

// 在这个iframe所加载的HTML中绑定一个事件监听器，当事件被触发时，把接收到的token数据写入localStorage
window.addEventListener('message', function (event) {
    localStorage.setItem('token', event.data)
}, false);
```

前端通过 `iframe` + `postMessage()` 方式，将同一份 `Token` 写入到了多个域下的 `LocalStorage` 中，前端每次在向后端发送请求之前，都会主动从 `LocalStorage` 中读取 `Token` 并在请求中携带，这样就实现了同一份 `Token` 被多个域所共享

此种实现方式完全由前端控制，几乎不需要后端参与，同样支持跨域

27.3. 流程

单点登录的流程图如下所示：



- 用户访问系统1的受保护资源，系统1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认证中心发现用户未登录，将用户引导至登录页面
- 用户输入用户名密码提交登录申请
- sso认证中心校验用户信息，创建用户与sso认证中心之间的会话，称为全局会话，同时创建授权令牌
- sso认证中心带着令牌跳转会最初的请求地址（系统1）
- 系统1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1
- 系统1使用该令牌创建与用户的会话，称为局部会话，返回受保护资源