利用交叉观察器解锁懒加载新姿势

原创：宋成林 全栈探索 前天

> 懒加载，一个在我们前端性能优化中高频出现的词汇，无论是懒加载图片还是懒加载模块，无非都是希望用户可以在滚动指定视区再去加载相应的资源，从而达到节省用户流量、提升首次加载时间、减轻服务器的压力的目的。

“懒加载”不是一个新的概念，对于经验丰富的你们来说，一定积累了很多实现方法，但为什么本文还要提出来呢？因为我们开发过程中，常用的实现方式都是通过监听页面的 scroll 事件来实现的，这种方法在使用过程中，scroll 事件会被高频触发，强制浏览器重排和重绘，从而不断增加浏览器的压力，导致浏览器性能的损失，有时可能出现卡顿或是闪烁的现象，即使使用了节流与去抖等方法，但还是会产生高昂的计算开销。那是否有更好的方法来解决呢，本文会讲解一种新实现方式，在此之前，先让我们回顾一下常用的实现姿势吧！

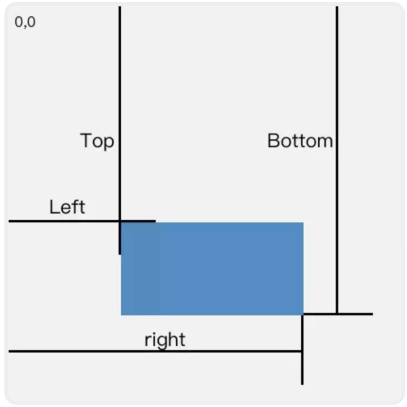
常用姿势

- 姿势一：getBoundingClientRect() 方法

通过 Element.getBoundingClientRect() 方法可以返回元素的大小及其相对于视口的位置。

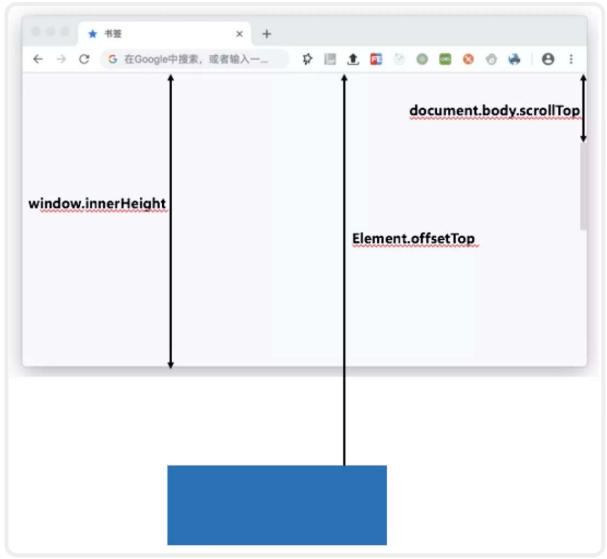
rectObject = object.getBoundingClientRect() ;返回值 rectObject 对象包含了一组用于描述边框的只读属性—— left 、top 、right 和 bottom ，单位为像素。除了 width 和 height 外的属性都是相对于视口的左上角位置而言的。

如下图：



随着页面滚动，我们可以获取懒加载元素相对于视口的顶点位置 rectObject.top 的像素值，当这个值小于可视区高 window.innerHeight ，表示已经进入可视区，开始加载数据，如果要提前加载，可以设置一个 threshold 阈值。

- 姿势二：



首先我们获取元素 Element.offsetTop ，获取可视区高度 window.innerHeigh 。

随着页面滚动，实时获取滚动条的高度 scrollTop ，如果元素 Element.offsetTop - 滚动条的高度 scrollTop 小于可视区高度 window.innerHeight 的时候，开始加载数据。

这两种实现方式都是通过监听页面的 scroll 事件的方法来实现，一直以来，我们都没有找到一个完美可靠的解决办法，试想，如果有一个方法，可以在指定的懒加载元素进入可视区域就通知我们就好了。

今天我就和大家介绍一个厉害的 API （ IntersectionObserver API ） ，它可以为我们提供一个简单直接的办法，让我们知道一个被观测的元素什么时候进入或离开可视区的视口。

IntersectionObserver 是什么？

简单介绍一下，截取 MDN [1]上的介绍：

" IntersectionObserver 接口(从属于 Intersection Observer API)为开发者提供了一种可以异步监听目标元素与其祖先或视窗(viewport)交叉状态的手段。 "

也就是说这个 API 提供了一种方法，可以异步观察目标元素与祖先元素或顶层文件的交集变化。网站不需要为了监听两个元素的交集变化而在主线程里面做任何多余的操作，浏览器就可以帮助我们优化和管理两个元素的交集变化, 且它是一个异步的 API，也就是说只有线程空闲下来，才会执行观察器。

说了这么多，你可能很想知道，这个API能否在实际项目中应用，下面我们先来看一下 IntersectionObserver API 的浏览器支持情况。

浏览器支持

大家可以在caniuse [3]上面查看最新的结果。

尽管现在还没有达到全部浏览器的支持。但是，如果你想尝试使用，并支持一些未支持的浏览器，可以使用 WICG 提供的 polyfill 帮大家解决这个问题。但值得的注意的是，使用 polyfill 是无法获取到原生实现

带来的性能优势的。如果你很在意，或许，你也可以选择判断浏览器是否支持，从而对不支持的浏览器使用其他方法去实现。

入门

在正式使用之前，我们先简单了解一下 API 中所有的参数和实例方法。以便我们在后续例子的讲解。

创建构造函数：

```
new IntersectionObserver(callback, options)

callback === (entries, observe) => {
    console.log（entries）
}
```

从上面可以看到，IntersectionObserver支持两个参数：

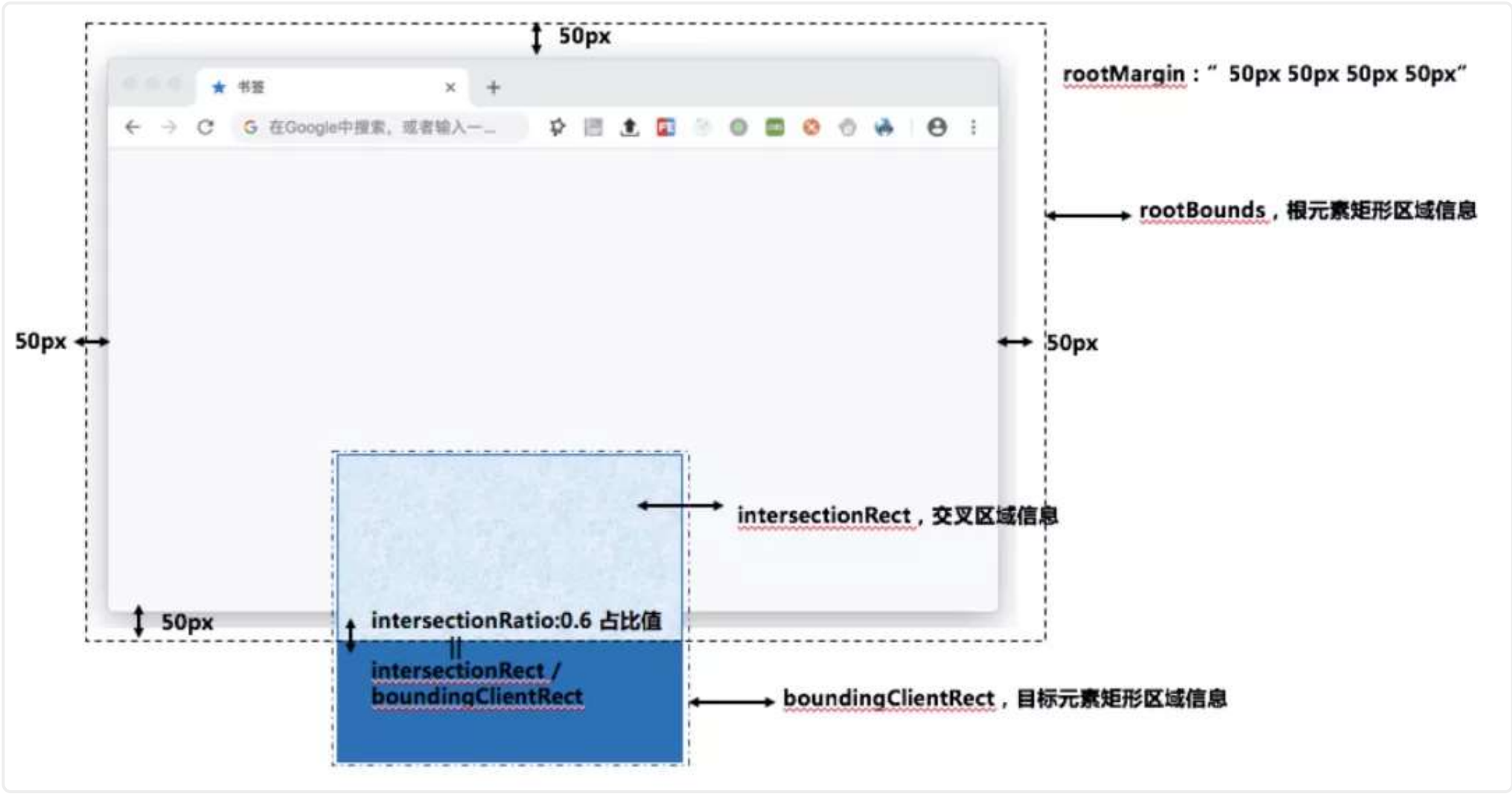
（1）callback, 触发的回调函数，当被观察元素与根元素视口发生交集，或交集部分大小发生变化的时候被执行，返回每次相交时的交集信息。接受两个参数，第一个参数是交集信息，以 IntersectionObserverEntry 对象数组形式返回，第二个是观察者本身。下面先来打印一下交集的返回信息。

```
▾ [IntersectionObserverEntry]
  ▾ 0: IntersectionObserverEntry
    ▸ boundingClientRect: DOMRectReadOnly {x: 0, y: 845, width: 1210, height: 494, top: 845, …}
      intersectionRatio: 0.008281192742288113
    ▸ intersectionRect: DOMRectReadOnly {x: 0, y: 845, width: 150, height: 33, top: 845, …}
      isIntersecting: true
    ▸ rootBounds: DOMRectReadOnly {x: 0, y: 0, width: 150, height: 878, top: 0, …}
    ▸ target: div.custom-transition
      time: 145640.8999999985
    ▸ __proto__: IntersectionObserverEntry
     length: 1
  ▸ __proto__: Array(0)
```

我们看到，IntersectionObserverEntry 对象有6个属性：

- boundingClientRect ：返回包含目标元素的矩形区域的信息，边界的计算方式与 Element.getBoundingClientRect() 相同。
- intersectionRect ：返回根元素和目标元素的交叉区域的信息。
- intersectionRatio ：返回目标元素的可见比例,也就是 intersectionRect 占 boundingClientRect 的比例值。
- isIntersecting ：返回一个布尔值, 如果目标元素与根元素相交，则返回 true ，否则返回 false 。
- rootBounds ：返回包含根元素的矩形区域的信息。
- target ：返回目标元素的 dom 节点对象。
- time ：返回交叉被触发的时间的时间戳，可见性发生变化的时间，是一个高精度时间戳，单位为毫秒。

可以简单用一张图来标注一下。



（2）options ,配置对象，包含三个可选属性，一旦 IntersectionObserver 被创建，则无法更改其配置。

- root ：被观察对象的祖先元素，也就是根元素，默认值是 null，也就是根元素指向的是浏览器的视口窗口，当前窗口里的所有元素都是根元素的后代元素，都是可以被观察的。
- rootMargin ：可以扩大和缩小根元素的判定范围，所有的偏移量均可用像素或百分比(来表达, 默认值为" 0px 0px 0px 0px "，如果设百分比，相当扩展根元素的对应宽高的百分比的值，例如：" 0% 0% 50% 0% "，就相当于向下扩展根元素的对应宽高的一半。

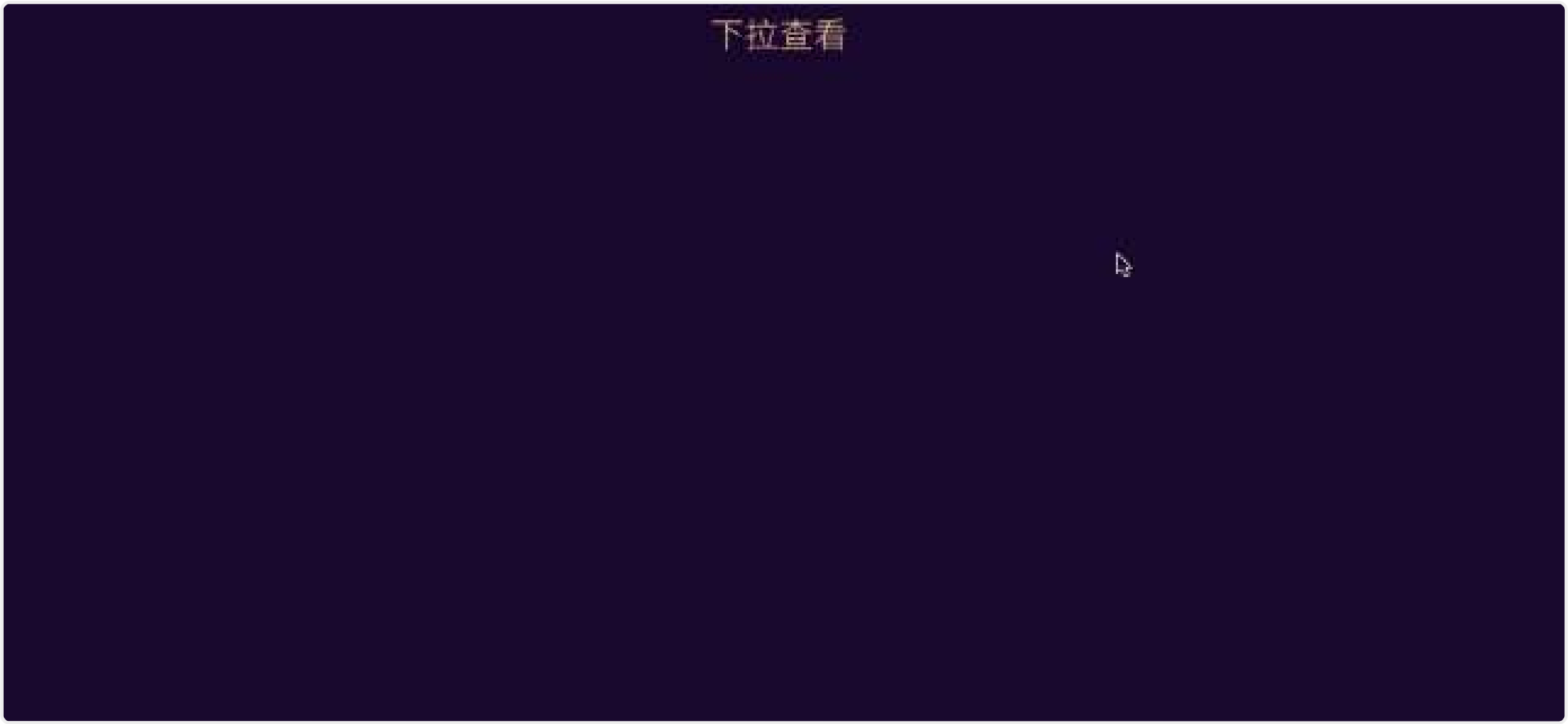
- thresholds ：一个包含阈值的数组, 数组中的每个阈值可以是 0~1 之间的任意数值，该数值是目标元素和根元素相交的面积相对于目标元素面积的值，等于或是大于指定的阈值时就会触发回调函数。默认值为[0]，也就是开始进入，就会触发。

API提供了四个实例方法：

- observer.observe() ：开始观察一个目标元素。
- observer.unobserve() ：停止观察指定的目标元素。
- observer.disconnect() ：关闭观察器。
- observer.takeRecords() ：当观察到交互动作发生时,回调函数并不会立即执行,而是在空闲时期使用 requestIdleCallback 来异步执行回调函数。该方法可以让 IntersectionObserver 同步执行。

项目中实践使用

利用 IntersectionObserver 实现懒加载页面模块，把每个模块都作为一个单独的目标元素去观察，当被观察的目标元素进入根元素的可视区内，开始加载组件模块内容。先看一下实现的效果。



我们把懒加载需求抽成一个单独的组件，在未加载的时候设置了占位骨架，可以自定义显示样式。

```
<template>
    <transition :name="fade">
        <div v-if="isLoad">
            <slot></slot>
        </div>
        <div v-else>
            <slot name="skeleton"></slot>
```

```
        </div>
    </transition>
</template>
<script>
export default {
    data() {
        return {
            isLoad: false,
            observer: null
            ...
        };
    },
    mounted() {
        ...
    }
```

被观察的目标元素是 dom 元素，且使用的 vue 框架开发，所以需要在 mounted 生命周期在实例化构造函数。

```
mounted() {
    // 观察根元素和被观察目标元素的交叉情况
    this.observer = new IntersectionObserver((entries, observer) => {
        entries.forEach((entry) => {
            if (entry.isIntersecting || entry.intersectionRatio) {
                // isLoad参数用于判断数据是否加载的，如果未加载值设置为 true ，开始后加载模块数据
                if (!this.isLoad) {
                    this.isLoad = true;
                    // 停止观察目标元素
```

```
                    this.observer.unobserve(this.$el);
                }
            }
        })
    }, { // 因为我们设置了50px的偏移量，所以只要被观察的元素靠近可视区50px，就会加载。
        rootMargin: '50px 0px',
        root: null,
        threshold: [0,  0.01]
    });

    // 观察目标元素
    this.observer.observe(this.$el);
}
```

组件销毁的时候，要记得关闭观察器

```
beforeDestroy() {
    // 停止观察所有的目标元素
    if (this.observer) {
        this.observer.disconnect();
    }
}
```

接下来，直接在页面中调用就好了，因为项目中是需要支持一些低版本浏览器的，所以用到了前面提到了 polyfill 来解决。

```
npm install intersection-observer
//安装之后在 .vue 页面 import 即可。
import 'intersection-observer';
```

如果你使用的其他技术栈，也可以参考https://github.com/w3c/IntersectionObserver/tree/master/polyfill上的使用方式。

IntersectionObserver 生存期

发生以下几种情况，观察器将停止观察目标元素：

- 对目标元素调用unobserve(target)。
- 调用disconnect()。
- 目标元素被删除。
- 交叉观察期的根元素被删除。

更多应用的地方

IntersectionObserver 还可以应用到 iframe 页面， 帮助我们检测 iframe 是否出现在的视口里。再此之前，对于跨域的 iframe ，我们是无法判断的。如果你想在 iframe 中使用，记得根元素一定要要设置成 null 或iframe 的祖先元素哦！

最后

IntersectionObserver 不仅可以帮我们解决传统方式带来的高昂计算开销，并且使用方便，只需要几行代码，就可以轻松搞定我们的需求，还在等什么，抓紧试用一下吧！最后，欢迎关注“全栈探索”公众号，我们每周都会推送好的文章！

扩展阅读：

[1] https://developer.mozilla.org/zh-CN/docs

/Web/API/IntersectionObserver

[2] https://github.com/w3c/IntersectionObserver

/tree/master/polyfill

[3] https://caniuse.com/#feat=intersectiono

bserver

End