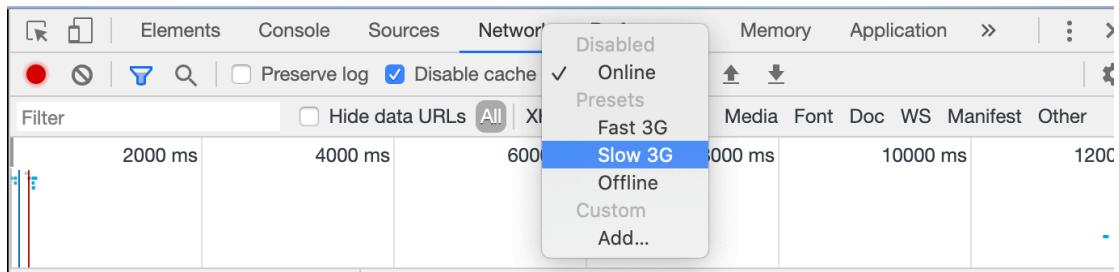


目标

为了提升用户体验，需要对主页的加载进行优化，主要考虑减少传输的文件的大小（css、html、js 以及图片视频）以及图片的懒加载和渐进式 jpg 格式处理

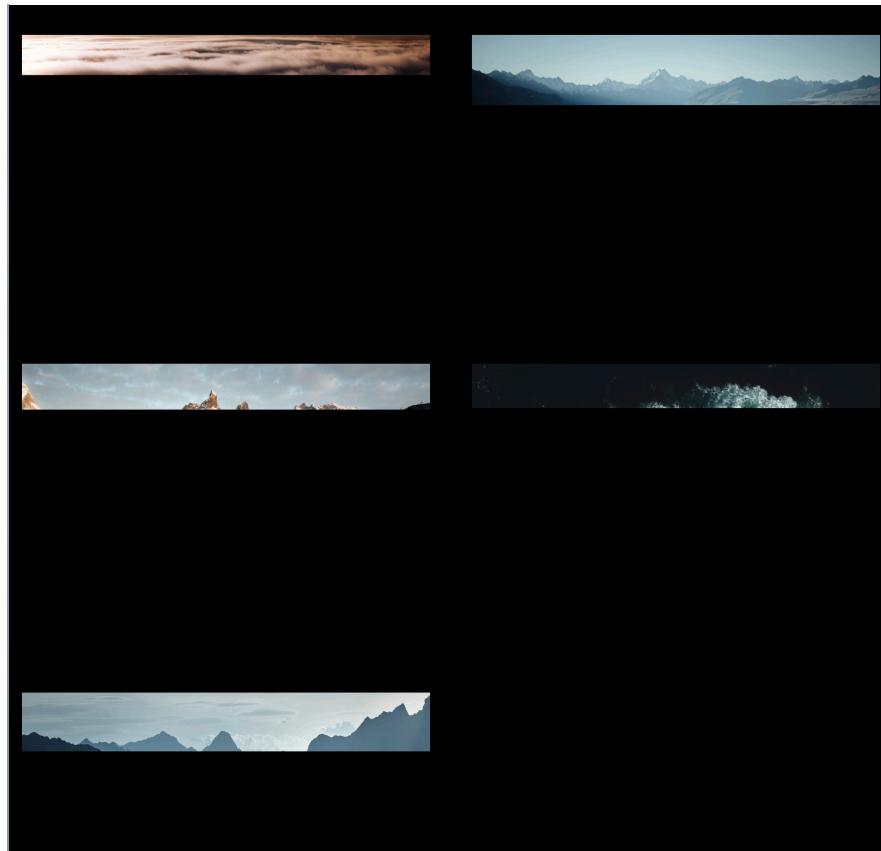
主要功能点

为了测试不同网速下的性能，开启浏览器开发者工具中调节网速的功能



渐进式 jpg

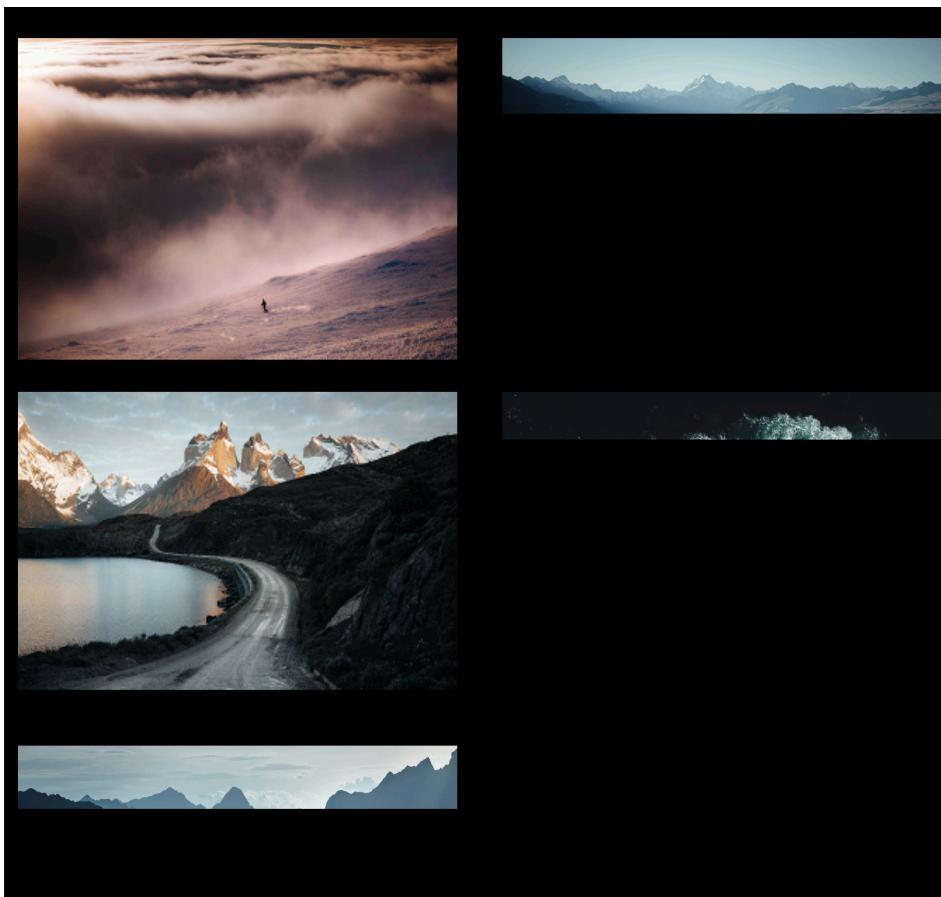
首先看不做任何处理的网站的主页加载情况：



可以看到图片加载速度非常慢，而且是一部分一部分加载，用户可以看到明显的加载过程，在开始加载的初期甚至看不到完整图片，只能看到黑屏，用户体验极差。

尝试将 jpg 格式转为渐进式 jpg 格式，使用 linux 下的 convert a.jpg -interlace Plane a.jpg 命令。

为了看到对比效果，下面先将图片的左边部分前排的 2 张图片做处理，和其他未做处理的对比如下：



可以看到做了渐进式 jpg 处理的两张图片瞬间就加载出来（放大看会看到做了模糊处理，后续再完全加载出来），而未做渐进式处理的图片还是从上到下逐渐加载

懒加载

渐进式 jpg 可以处理图片瞬间加载不出来的问题，但是如果图片太多，一次性请求所有图片仍然会导致图片加载过慢，因此考虑使用图片懒加载技术，当图片出现在用户的屏幕视野中再加载图片

通过网上查找资料调查发现，有如下几种实现方法：

1. 使用 img 属性 loading="lazy"

```
</div>
<img loading="lazy" d...
```

这种方式实现简单，但是因为是 chrome 浏览器新发布的特性，因此浏览器兼容性不是很好

2. 通过 js 计算图片是否出现在屏幕视野中

这一种方法主要是通过 js 获取 img 元素，然后通过计算图片距离屏幕顶部的高度以及用户当前的位置来判断图片是否出现在视野中，如果出现在视野中，则加载图片，大概的代码如下：(参考 <https://zhuanlan.zhihu.com/p/55311726>)

```
var imgs = document.querySelectorAll('img');

//offsetTop是元素与offsetParent的距离，循环获取直到页面顶部
function getTop(e) {
    var T = e.offsetTop;
    while(e = e.offsetParent) {
        T += e.offsetTop;
    }
    return T;
}

function lazyLoad(imgs) {
    var H = document.documentElement.clientHeight;//获取可视区域高度
    var S = document.documentElement.scrollTop || document.body.scrollTop;
    for (var i = 0; i < imgs.length; i++) {
        if (H + S > getTop(imgs[i])) {
            imgs[i].src = imgs[i].getAttribute('data-src');
        }
    }
}

window.onload = window.onscroll = function () { //onscroll()在滚动条滚动的时候触发
    lazyLoad(imgs);
}
```

但是分析以上代码可以发现，该方法监听滚动事件，而滚动事件是一个比较频繁发生的事件，每次滚动屏幕都调用检测方法的话对于性能也是一个瓶颈

3. 采用 IntersectionObserver

这一种处理的思路和上面一种方法类似，不同的是用了浏览器提供的 IntersectionObserver API，内部实现为一个观察者模式，通过监听图片与当前视野的交集来判断是否需要启动加载，具体代码如下：

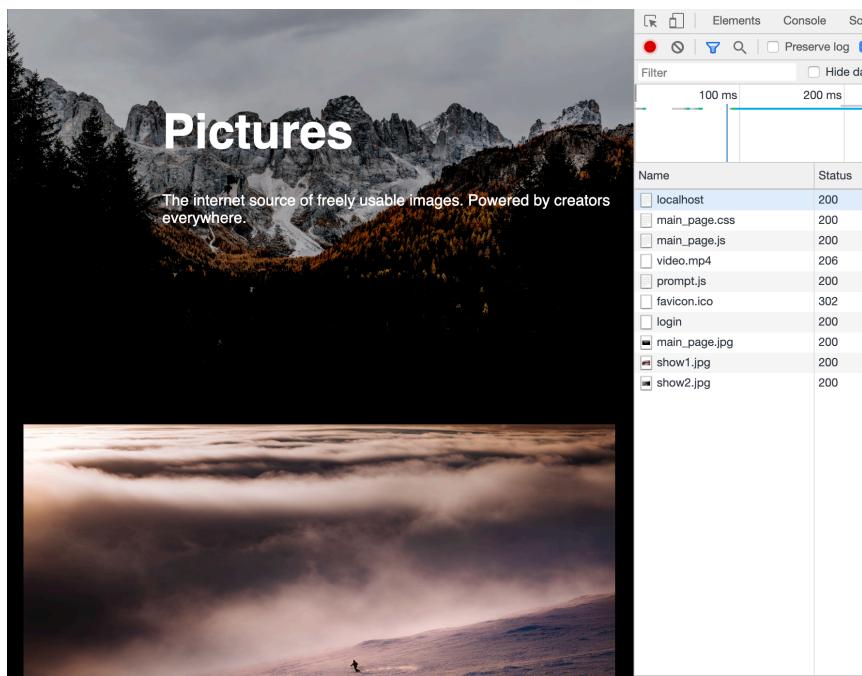
```
var observer = new IntersectionObserver(callback);

elements.forEach(item => {
    observer.observe(item);
})

if (imageSource) {
    e.setAttribute('src', imageSource);
    e.removeAttribute('data-src');
}
```

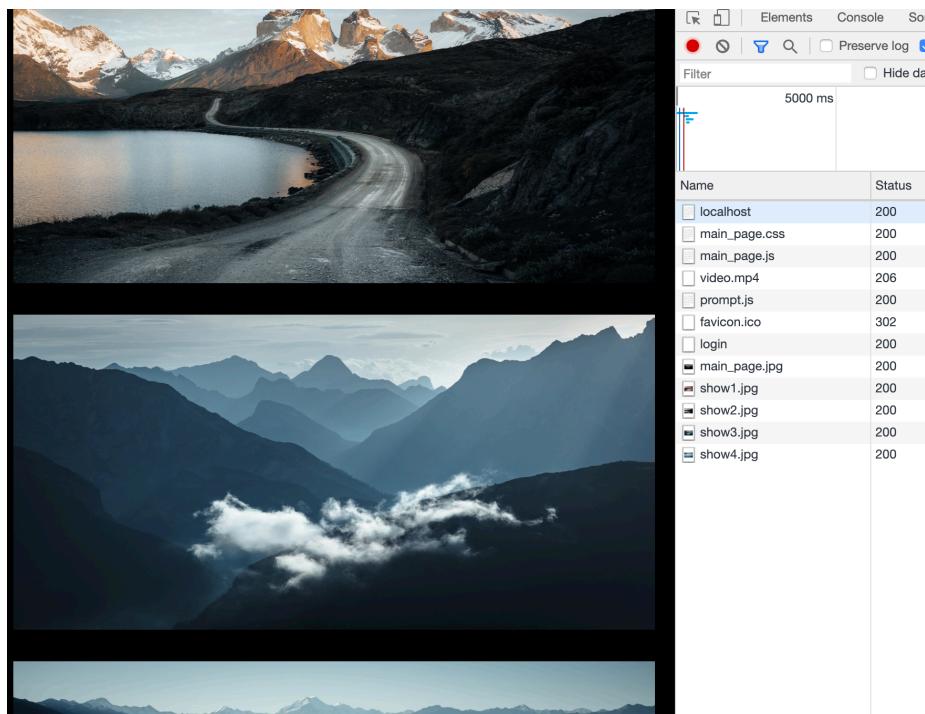
效果如下：

当主页只有前两张图片的时候：



可以看到右边就加载了两张 jpg

当有更多的图片进入用户视野的时候：



陆续请求后续图片的加载

文件压缩

最后考虑文件压缩，通过调研发现 gzip 压缩格式广泛应用于 web 文件压缩，nodejs 引入也比较简单，使用 express 框架只需要如下代码：

```
var compression = require('compression')
var app = express();
app.use(compression());
```

效果：

可以看到部分相应请求加上了一个响应头：

Content-Encoding: gzip

The screenshot shows the Network tab of a browser's developer tools. A file named "main_page.css" is selected in the list on the left. The right panel displays the "General" and "Response Headers" sections. In the General section, the Request URL is listed as `http://localhost:3000/main_page.css`, the Request Method is GET, the Status Code is 200 OK, and the Referrer Policy is no-referrer-when-downgrade. The Response Headers section lists several headers, including Accept-Ranges, Cache-Control, Connection, Content-Encoding (set to gzip), Content-Type, Date, ETag, Last-Modified, Transfer-Encoding, Vary, and X-Powered-By.

Request URL	Request Method	Status Code	Referrer Policy
http://localhost:3000/main_page.css	GET	200 OK	no-referrer-when-downgrade

Header	Value
Accept-Ranges	bytes
Cache-Control	public, max-age=0
Connection	keep-alive
Content-Encoding	gzip
Content-Type	text/css; charset=UTF-8
Date	Tue, 24 Dec 2019 15:14:01 GMT
ETag	W/"53f-16ef4f0c0e9"
Last-Modified	Wed, 11 Dec 2019 12:30:49 GMT
Transfer-Encoding	chunked
Vary	Accept-Encoding
X-Powered-By	Express

再看文件大小的变化：

压缩前：

main_page.css	200	styles...	(index)	1.6 KB	12 ms
---------------	-----	-----------	---------	--------	-------

压缩后：

main_page.css	200	styles...	(index)	951 B	27 ms
---------------	-----	-----------	---------	-------	-------

可以看到文件显著变小