首页加载慢？
1.首页加载图片过多 2.首页的请求量过多 3.首页请求的静态资源（HTML、CSS、JS、图片...）过大
4.还有网速不好和电脑太渣

首页请求的资源（CSS、JS、图片...）过大怎么解决？
把资源变小，要分资源文件，CSS，JS，图片等要分开来处理：
1.CSS 和 JS 可以通过 Webpack 来进行混淆和压缩
混淆：将 JS 代码进行字符串加密（最大层度减少代码，比如将长变量名变成单个字母等等）
压缩：去除注释空行以及 console.log 等调试代码
2.图片也可以进行压缩：
（1）可以通过自动化工具来压缩图片
如用熊猫站（智能压缩 PNG 和 JPG 的一个网站），可以对图片进行等比例无损压缩，原理是通过相似颜色“量化”的技术来减少颜色数量，并且可以将 24 位的 PNG 文件转化成 8 位的彩色图片，同时可以将不必要的元数据（其它不相关的图片信息）进行剥离。当然，熊猫站很佛系的，它把图片压缩工具（源码） 开放 出来了。可以使用 npm 安装开源包，就可以在本地进行图片压缩了。
（2）对图片进行转码 -> base64 格式
base64 格式的图片的作用是减少资源的数量，通过把 src 路径的值转成一串字符串，但是 base64 格式的图片（可能）会增大原有图片的体积。
（3）使用 WebP 格式
根据 Google 的测试，同等条件等比例无损压缩后的 WebP 比 PNG 文件少了 26% 的体积。并且图片越多，压缩后的体积优势越明显。
（4）通过开启 gzip 进行全部资源压缩
gzip 是一种压缩文件格式，可对任何文件进行压缩（类比文件压缩），可通过 nginx 服务器的配置项进行开启。

首页加载图片过多怎么处理？
可以通过懒加载的方式来减少首屏图片的加载量；
对于纯色系小图标可以使用 iconfont 来解决，设置 font-family 的 CSS 属性；
对于一些彩色的小图片可以使用雪碧图，把所有小图片拼接到一张大图片上，并使用 background-position 的 CSS 属性来修改图片坐标。

懒加载的原理？
懒加载原理就是监听滚动条事件，如果（滚动条距离浏览器顶部的高度 === 图片距离顶部的高度），那么就将 data-src 的值赋值到 src 上。

首页的请求量过多怎么解决？
1.先通过工具来确定是哪些类型的资源请求过多
（1）通过浏览器的 Network（调试工具）可以确定首页加载的资源和请求量，
requests：请求数量 resources：前端资源总大小
DOMContentLoaded：时间，浏览器已经完全加载了 HTML，其他静态资源（ JS, CSS, 图片等）并没有下载完毕
Load：时间，浏览器已经加载了所有的静态资源（能用了）
（2）通过 converge 来查看代码的使用状况，只针对 JS 和 CSS，可以看出哪些代码虽然加载了但是没有执行，没有执行的代码可以考虑一下是否可以懒加载。
2.可以减少资源的请求量：
（1）通过 nginx 服务器来做资源文件合并 combo（请求地址 url 用逗号分成多个一齐请求，nginx 通过处理多个地址对应的文件，返回合并的文件），将多个 JavaScript、CSS 文件合并成一个。（也可用来做 CDN，用来处理静态资源）
日常企业项目中服务器按照功能区分：
应用服务器：服务端语言运行的服务器（Java，NodeJS...）；数据库服务器：放数据库的服务器；
存储服务器：放大型文件的服务器（例如各种网盘）；CDN 服务器：放静态资源的服务器（JS，CSS, 图片, 字体...）
（2）通过打包工具（Webpack）来做资源文件的物理打包（相对没有第一种灵活）

3.还可以从代码层面的优化：
（1）对于引入的一些比较大型的第三方库，如组件库（antd，element-ui）、函数库（lodash）等，可设定按需加载（一般都是用 Babel 插件来实现的）
（2）可通过前端路由懒加载的方式（所有组件不会一次性全部加载，而会分成多个对应文件）（只限 SPA 应用）：如使用 React lazy 进行前端动态路由的懒加载（拆包）（React 16.6 以上版本才可以使用 React lazy），从而可以减少首页的 JS 和 CSS 的大小。

为什么 React lazy 可以进行动态路由的加载？使用方式代码：

```
// 1. 引入 react lazy, 并且使用 import 动态导入组件
import { lazy } from 'react'; // 静态导入
lazy(() => import('./Home')); // 动态导入
// 2. 引入 Suspense 组件，并使用 Suspense 将根组件包裹起来，并使用 fallback props 传入 loading 组件
import { Suspense } from 'react';
// 注意：使用 lazy 加载的组件，必须是 Suspense 子组件，或者孙组件
<Suspense fallback={<div>Loading...</div>}>
    <OtherComponent />
</Suspense>
```

动态导入(dynamic import)：当代码运行 import 的时候，再导入组件

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
// 类似于 fetch，都是返回一个 Promise
fetch("./math").then(math => {
  console.log(math.add(16, 26));
});
```

首先 React lazy 是使用了 dynamic import 的标准，webpack 只要遇到了 dynamic import， 就会把里面引入的内容单独打一个包。由于 dynamic import 返回的是一个 Promise，所以可以使用 Promise 的状态来做渲染的流程控制。如果当前 Promise 是 pending 状态，那么就渲染 Loading 组件，如果 Promise 是 resolve 状态那么就渲染动态导入的组件。

怎么使用 Webpack 进行打包优化？
少 -> 使用 Webpack 进行物理打包。
小 -> 使用 Webpack 进行混淆和压缩，所有与 Webpack 优化相关的配置都是在 optimization 这个配置项里管理。
从 webpack 4 开始，会根据选择的 mode 来执行不同的优化，不过所有的优化还是可以手动配置和重写。
development：不混淆，不压缩，不优化　　production：混淆和压缩，自动内置优化

用 React lazy 对文件进行拆包处理，那么肯定会造成文件变多，是不是有矛盾？
其实不冲突，拆包后的文件（只是在服务器上面的文件变多），不可能同时加载的，所以就不会造成同一时间资源请求过多的请求。但是要注意打包策略，通常会把包分为两类：
第三方包（node_modules 里面的）；自己实现的代码（src 目录里面的），有公共的和非公共的。
可以把第三方包打一个包，公共的代码打一个包，非公共的代码打一个包。
第三方包：改动频率 -- 小。公共代码包：改动频率 -- 中。非公共代码包：改动频率 -- 高。
可以将 打包策略 结合 网络缓存 来做优化：
对于不需要经常变动的资源（第三方包），可以使用 Cache-Control: max-age=31536000（缓存一年） 并配合协商缓存 ETag 使用（一旦文件名变动才会下载新的文件）；对于需要频繁变动的资源（代码包），可以使用 Cache-Control: no-cache 并配合 ETag 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新。

CDN（解决方案，内容分发网络）：通过 nginx，距离最近的服务器，放静态资源的服务器（JS，CSS，图片，字体...），自动选择最近的节点，用来加速静态资源的下载，可以实现加速。

Http1.1 请求：对于同一个协议、域名、端口，浏览器允许同时打开最多 6 个 TCP 连接（最多同时发送 6 个请求）。而利用 cdn，把不同资源放到不同服务器上，由于 CDN 服务器的地址一般都跟主服务器的地址不同，所以可以破除浏览器对同一个域名发送请求的限制。
现在，Http2.0：引入了多路复用的机制，可以最大化发送请求数量。（取消限制了）

为什么一次性渲染很多条数据会造成浏览器卡顿？
无论是浏览器中的 DOM 和 BOM，还是 NodeJS，它们都是基于 JavaScript 引擎之上开发出来的，而 DOM 和 BOM 的处理最终都是要被转换成 JavaScript 引擎能够处理的数据，这个转换的过程很耗时，所以在浏览器中最消耗性能的就是操作 DOM。所以要尽可能的减少 DOM 的操作。
1.可以使用 document.createDocumentFragment 创建虚拟节点，从而避免引起没有必要的渲染
2.当所有的 li 都创建完毕后，一次性（分数量的）把虚拟节点里的 li 标签全部渲染出来
3.可以采取分段渲染的方式，如一次只渲染一屏的数据
4.最后使用 window.requestAnimationFrame 来逐帧渲染

```
// 插入十万条数据
const total = 100000;
let ul = document.querySelector('ul'); // 拿到 ul
// 懒加载的思路 -- 分段渲染// 1. 一次渲染一屏的量
const once = 20;
// 2. 全部渲染完需要多少次，循环的时候要用
const loopCount = total / once;
// 3. 已经渲染了多少次
let countHasRender = 0;
function add() {
  // 创建虚拟节点，（使用 createDocumentFragment 不会触发渲染）
  const fragment = document.createDocumentFragment();
  // 循环 20 次
  for (let i = 0; i < once; i++) {
    const li = document.createElement('li');
    li.innerText = Math.floor(Math.random() * total);
    fragment.appendChild(li);
  }
  // 最后把虚拟节点 append 到 ul 上
  ul.appendChild(fragment);
  // 4. 已渲染的次数 + 1
  countHasRender += 1;
  loop();
}
// 最重要的部分来了
function loop() {
  // 5. 如果还没渲染完，那么就使用 requestAnimationFrame 来继续渲染
  if (countHasRender < loopCount) {
    // requestAnimationFrame 叫做逐帧渲染 // 类似于 setTimeout(add, 16);
    // 帧：一秒钟播放多少张图片，一秒钟播放的图片越多，动画就约流畅  1000/60 = 16
    window.requestAnimationFrame(add);
  }
}
loop();
```