SSR当代最强变种 - Island架构

前端娱乐圈最新花活

纸贵-沈昊

目录

什么是SSR

- SSR (Server Side Rendering) 指的就是在服务端组装网页内容的渲染技术, 例如JSP,PHP应用, 对于客户端而言可以避免多次的HTTP请求, 更利于SEO/首屏渲染速度提升.

开发时:

```
<body>
    我是类服务端模板引擎生成的html页面, 我通常是这样渲染数据的:
    ${data.year}-${data.month}-${data.day}
</body>
```

服务端渲染后, 客户端拿到的html网页:

```
<body>
    我是类服务端模板引擎生成的html页面, 我通常是这样渲染数据的:
    2022-1-1
</body>
```

爬虫优先

为了更好理解SSR渲染模式对于SEO有什么好处, 我们可以尝试了解一波爬虫(百度叫蜘蛛, 谷歌叫爬虫); 爬虫会在互联网上抓取一部分网页, 抽取其中的url放在一个待抓取的队列, 然后通过dns解析拿到一批网站的ip, 再通过网站下载器下载网站; 那么其中“抓取部分”, 就是重中之重的部分;

> 抓取的策略也很简单, 就是最重要的网页;

- SSR组装的工作是在服务端, 它确保了内容在客户端立即呈现, 不需要等待资源加载完成, 因此它呈现结果是异常高效的, 所以爬虫就非常喜欢这样的页面.

(查看网页源代码)

```
<div class="search-hots-fline" data-spm-ab="fline">
  <a href="http://s.taobao.com/search?spm=1.7274553.1997520241-2.2.TpEKPQ&amp;q=耳机&amp;refpid=420463_1006&amp;source=t
  <a href="http://s.taobao.com/search?spm=1.7274553.1997520241-2.2.TpEKPQ&amp;q=女包&amp;refpid=420464_1006&amp;source=t
  <a href="http://s.taobao.com/search?spm=1.7274553.1997520241-2.2.TpEKPQ&amp;q=沙发&amp;refpid=420465_1006&amp;source=t
</div>
```

高权重的电商网站/官网, 需要用搜索流量引流的主要业务, SEO ER都会非常注重爬虫效率

什么是CSR

- CSR (Client Side Rendering) 相对于SSR, 是完全相反的渲染技术, 它把在服务端的组装工作放到了客户端, 在初次请求的时候仅仅会返回一个引导性HTML, 剩余的页面交互和渲染通常会通过JS在一个页面中切换视图;

> 在我们熟悉的VUE,REACT这一类的视图层框架中, 就是默认沿用CSR渲染的设计

CSR只有一个空html文档, 通常会引用一个或多个js文件, 这就意味着爬虫需要等待js文件加载完成, 才可以显示网页, 比如Vue (Vite)的模板代码就是这样:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

强交互性

> SSR虽然对于高权重的电商网站/官网, 是非常有用的, 但是对于低权重的网站, 它只是一个简单的静态页面, 并不能满足用户的需求, 因此它不是一个好的选择; 当我们用户需要在网站上做强交互时, 比如频繁跳转页面等, 那么使用SSR就会造成过多的网络往返 (或者服务器压力).

vue router.ts

```ts
import { createRouter, createWebHashHistory } from 'vue-router'

const router = createRouter({
  history: createWebHashHistory(),
  routes: [
    //...
  ],
})
```

通常的单页应用路由是用hash实现, 它是一个锚点, vue会根据#后的url去切换不同的页面, 这个切换的过程全程是js完成的, 因此不需要额外的服务器压力, 只需要等待js文件加载完成, 就可以显示网页了.

```
https://localhost:8080/#/home
```

针对缺点和优点进行选型

> SSR的主要优点

- SEO效果很好
- 首屏渲染性能更好

> CSR的主要优点

- 强交互场景下节省网络往返
- 没有服务端压力

鱼和熊掌可以兼得-同构架构

> 一般的通用渲染方案: 同构架构即SSR + CSR

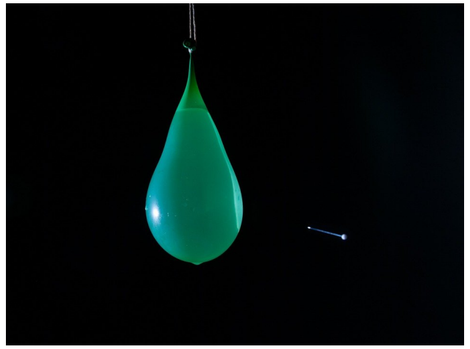
在首屏下, 通过Nodejs渲染页面, 在非首屏下, 通过浏览器渲染页面. 这样既保证了SEO的需求和首屏可交互性, 也保证了在强交互场景下用户所需要的用户体验, 也减少了服务器压力.

```
asyncData({ store, fetch, route }) {
  // 首次加载页面, 会经过nodejs请求api
  return fetch('/api/data').then(res => res.json())
}
```

Nuxt.js / Next.js 都是这么做的, 在同构架构下, 我们的js代码需要同时运行在浏览器和服务器上, 所以我们需要框架来尽可能的磨平客户端和服务端的api差异, 并且提供一个独立的模块, 来保存同构代码.

> 那么Nuxt这一类的全栈框架是如何做到静态页面使其仍然可以交互呢?

喝水 / 脱水 / 注水

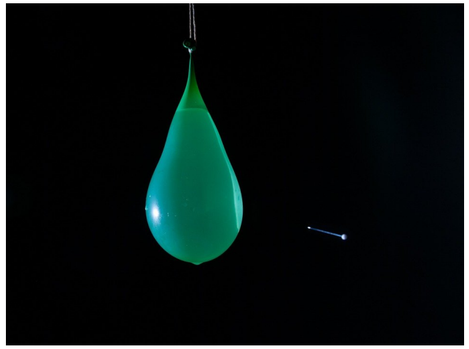


喝水

将数据流补充到页面上下文中(服务端)

脱水

客户端需要直出HTML, 需要把数据嵌入到静态的HTML



注水

客户端执行/下载静态页面中的JS, 使其页面重新变得可交互

Vite中实现asyncData

> 在Vite中给我们提供了一部分的SSR功能, 虽然是实验性的, 但是非常有助于我们理解注水实现

- 实现asyncData函数

```
export const render = async (url) => {
  const { app, router } = createApp()
    router.push(url)
    await router.isReady()
    let data = {}
    // 命中路由组件，且执行asyncData这个函数
    if (router.currentRoute.value.matched[0].components.default.asyncData) {
      const asyncFunc = router.currentRoute.value.matched[0].components.default.asyncData
      data = asyncFunc.call()
    }
    const html = await renderToString(app)
    return { html, data }
}
```

Vite脱水序列化

- 我们需要把上一个部分拿到的data和html字符串在服务端进行脱水, 原理就是把data合并到html中

```
app.use('*', async (req, res) => {
  try {
    const url = req.originalUrl
    let template = readFileSync(resolve('index.html'), 'utf-8')
    template = await vite.transformIndexHtml(url, template)
    const { render } = await vite.ssrLoadModule('./src/entry-server.js')
    const { html: appHtml, data } = await render(url)
    // 拼接标签，把data序列化插入到文档中
    const html = template.replace(`<!--ssr-outlet-->`, `${appHtml}<script>window.__data__=${JSON.stringify(data)}</scri
    res.status(200).set({ 'Content-Type': 'text/html' }).end(html)
  } catch (error) {}
})
```

- 同理, 我们的Vuex数据也是这样持久化data的, 当在客户端时, 只需要将各种类型的data, 重新注入到实例中, 我们的Vue组件就可以正常交互了

注水

- 将已经序列化的data和初始化的空data进行合并

```
router.isReady().then(() => {
  const component = router.currentRoute.value.matched[0].components.default
  let _data = {}
  // 判断是否是函数
  if (typeof component.data === 'function') {
    _data = component.data.call()
  }
  // 判断是否有脱水的data
  if (window.__data__) {
    _data = {
      ..._data,
      ...window.__data__
    }
  }
  component.data = () => _data
  app.mount('#app')
})
```

SSR变种

> 随着前端业务发展, SSR和CSR已经不能满足需求, 所以衍生出了很多变种方案, 但是核心理念都会围绕以下几个点:

- 服务器运维成本降低
- 页面更新频率 (是否特别依赖及时更新, 比如资讯网站需要频繁更新)
- 偏向服务端侧渲染 (Pre-rendering) 或者在客户端渲染上做文章
- 根据前端全栈框架演变的不同渲染方法 (框架的不同也可能造就渲染模式的差异, 因为其核心理念不同)
- 根据部署平台的特点, 比如Vercel, 也有独有的渲染方法ISR (静态增量渲染生成)

SSG (Pre-rendering)

> SSG指的就是在服务端间歇性生成静态页面的渲染方式

比如在资讯网站,用户看到的文章内容都是一样的,那么就可以把需要多次渲染的文章页面,只用渲染生成一次缓存到服务器中,当下一次请求就可以直接返回已经生成好的html给客户端; 这就意味着用户看到的内容不会是最新的,有可能服务器会在1分钟/1小时才会生成/缓存一次.

静态生成的网站, 你可以非常方便的通过CI部署在Vercel / Cloudflare上



Nuxt3 - 混合模式

在这一章中主要介绍偏Nuxt3的新通用渲染模式, 在传统的全栈框架, 我们可以在整个程序中选择单独一种作为渲染方案, 而现在你可以根据路由使用多种渲染方案:

- a路由构建生成静态页面
- b路由使用服务端渲染
- c路由使用客户端渲染

Nuxt得益于最新的JS服务器引擎 (Nitro), 实现了路由缓存以及边缘侧渲染

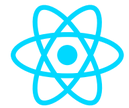
```
export default defineNuxtConfig({
  routes: {
    '/': { prerender: true }, // Once per build (via builder)
    '/blog/*': { static: true }, // Once on-demand per build (via lambda)
    '/stats/*': { swr: '10 min' }, // Once on-demand each 10 minutes (via lambda)
    '/admin/*': { ssr: false }, // Client-Side rendered
    '/react/*': { redirect: '/vue' }, // Redirect Rules
  }
})
```

SPA & MPA

> 前端网页程序中分为单页应用(SPA)和多页应用(MPA), 我也相信前端同学们都已经非常了解它们了 (老八股文题目了😄)

- 多页应用程序是由多个html文档组成的网站, 当用户访问新的导航时, 浏览器会有一个新请求到达服务器, 传统的MPA框架有很多, 比如ruby on rails, 还有php laravel等等.

- 单页应用指的是单个js应用程序组成的网站, 只有一个页面, 在浏览器中加载js->操作视图->然后呈现html; 但是spa也可能在服务器上生成html, 比如我们熟知的nuxtjs, nextjs, remix, vue/react等等

> 我们简单回顾了SPA & MPA的概念, 这将有助我们理解接下来的Island架构和其背后代表的全栈框架

现代Web网站开发困局

极大部分的现代网站都需要JS, 不仅可以完成复杂的需求以及交互体验, 但是随着时间的推移, 脚本的数量将会成指数型增长. 那么很显然JS脚本过多并不是一个好事情:

- 网络带宽, 大量的JS代码被发送到客户端, 不仅对服务器是一个挑战, 在客户端低网速下, 等待也是很昂贵的
- 更久的启动时间, 客户端需要大量JS激活组件, 使其变得可交互

而且JS是单线程的, 无法利用多核CPU解决根本问题.

> 解决问题的方法也很简单, 不要JS不就得了!

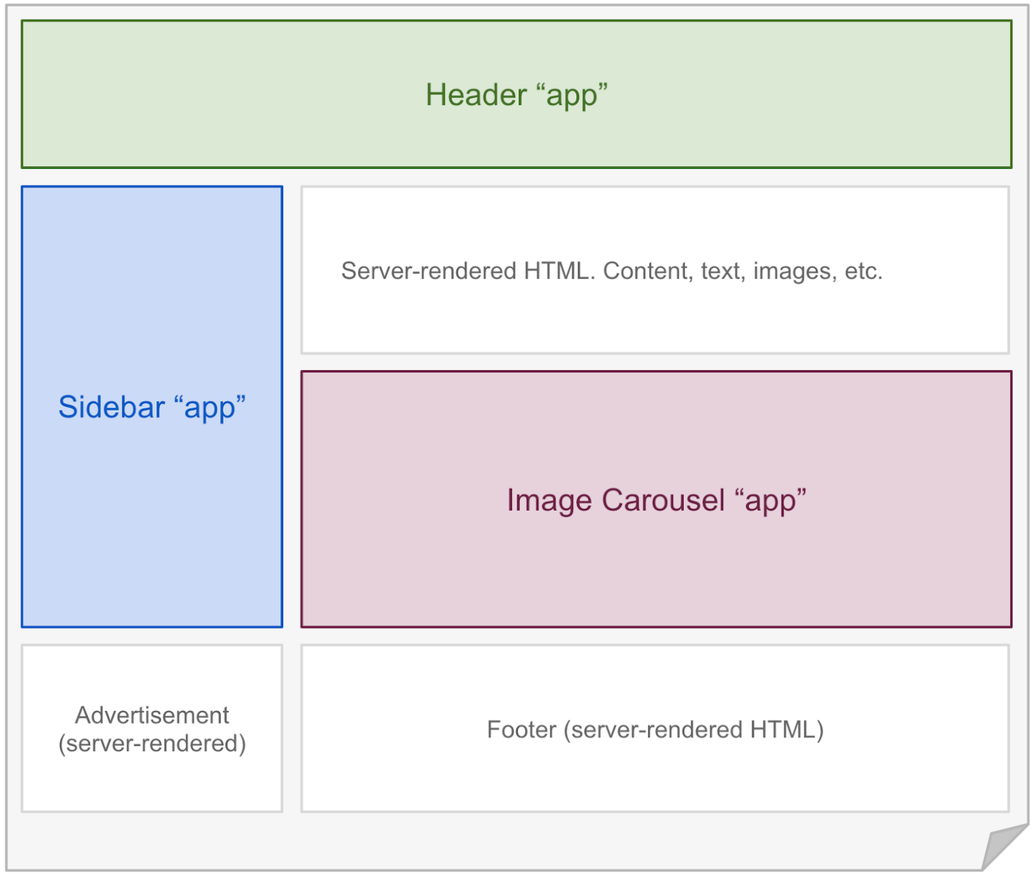
Island(孤岛)架构

现代网站过于依赖JS, 不管我们使用的是什么框架, 网页(CSR)都需要加载过多的JS文件, 在传统的SSR架构下, 服务端需要给客户端发送巨多的JS代码(混合在HTML中)使页面重新注水; 而且大多数情况下, 网站动态的内容虽然只有一小部分但是却用了臃肿的框架/库去额外渲染, 造成了其他组件的阻塞

所以孤岛架构的目标就是CSR的优秀交互体验以及SSR的SEO性能, 用最自然的描述语言(HTML)去编写现代高性能网站. 除了上述这些, 它还拥有几个特点:

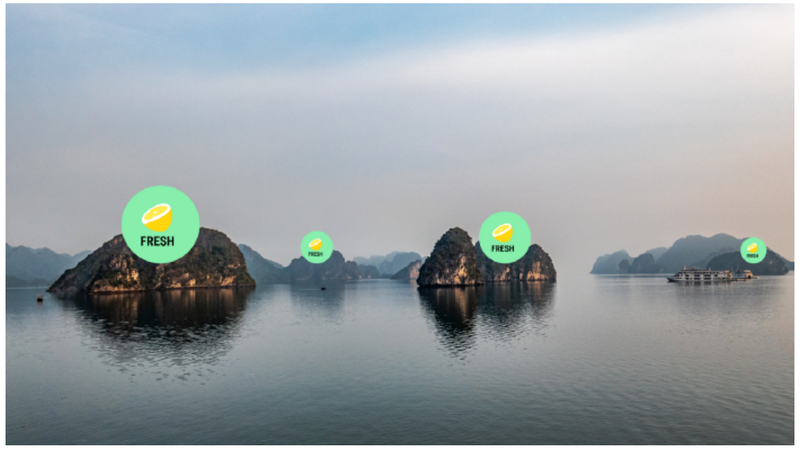
- 组件化
- 动静分离, 静态内容无需经过处理, 动态内容自我注水(此体系所基于的技术就是部分/选择注水)

孤岛一词的含义



网站可分为动态交互内容和静态内容, 孤岛架构网站的动态内容每一块都是单独开发和维护的, 不会限制你的技术栈(即UI层渲染, 你可以使用Vue/React/Preact); 在常见的网站中, 比如官网/电商/博客网站, 在静态不变的内容下, 通常都会有交互式组件, 比如聊天对话框, 可变的链接跳转, 那么这些交互功能即可以理解为一个又一个“孤岛”, 孤岛和孤岛之间互不干扰.

动态孤岛组件



> 我们需要再次梳理一下, 孤岛架构中的动态孤岛组件主要解决了什么问题, 或许你在前面的章节中已经了解过它们的优点, 那么我们这一部分将对优点进行剖析.

- 每一个交互式孤岛组件是完全隔离的, 意味着不会影响彼此的渲染, 静态内容亦是如此.
- 不会限制你的技术栈, 即UI层渲染
- 异步交互, 在初次渲染时不会有JS

我们了解到了动态孤岛组件的特征之后, 就可以来了解一下其实现细节

动态孤岛组件实现细节

> 当我们实现一个孤岛动态组件时, 在服务端会将js剥离:

```
import { useState } from "preact/hooks";

const Card = () => {
  const [count, setCount] = useState(0);
  const add = () => {
    setCount(count + 1);
  };
  return (
    <div>
      <div>{count}</div>
      <button onclick={add}>ADD</button>
    </div>
  );
};
```

```
<Island url="***/card" renderer-url="nodemodules/preact.js">
  <div>
    <div>0</div>
    <button>ADD</button>
  </div>
</Island>
```

动态孤岛组件渲染 & 激活

> 渲染的时机取决于Island相关框架的设计, 但是默认情况下会在服务端渲染为静态html, 那么动态组件重新激活/注水的时机有可能是立即的, 也有可能是异步的.

如果在业务场景下, 你希望你的购买按钮能够立即显示, 那么此时页面显示时就会让组件重新注水

```
<button>立即购买</button>
```

那么还有一些场景下, 组件并不在可视区域, 如果此时立即注水, 那么就会造成性能浪费

```
<footer>
  <Concat></Concat>
</footer>
```

> 所以Island体系下的框架, 会给动态组件提供多种激活策略, 比如: 可视区域激活, 延迟激活, 立即激活, 媒体查询激活, 甚至是手动激活.

微前端? 还是渐进式增强?

- 微前端?

我们现在对Island架构有了一个初步的认识, 你也有可能和我一样在初次学习时和微前端概念有所混淆, 因为它们都提倡把应用程序分解为独立单元(进行部署), 但是微前端的每个单元并不完全使用html实现.

- 渐进式增强?

众所周知Vue是一门渐进式框架, 大家也都相当熟悉渐进式的意思, 它与Island一样, 有着渐进式概念; Island在SSR的基础上加入了类微前端概念提供了独立的组件单元, 使其交互变得更自由, 性能更贴近原始HTML. 在传统渐进式增强中, 我们通常会在客户端查询元素并且在元素上初始化一个动态函数, 而在Island中, 动态组件将经过服务端, 由服务端生成一个专属此组件的js文件, 让组件自给自足.

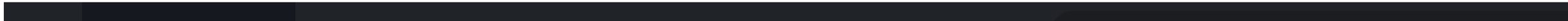
Island框架盘点 - Astro

> 实现Island架构的优秀框架数量并不多, 每一个框架都有各自的优点, 在此次分享中我会着重介绍这三款框架; 尤其将Astro作为我们的主角, 介绍它的设计思路, 以及它的实现原理.

- 不限制你的UI层渲染框架, 你可以自由选择solid.js, svelte, react, preact, vue等等, 也具有.astro的单组件提供静态渲染以及囊括多组件的能力(在astro中你可以同时编写多个框架组件, 并且可以一起工作)
- 提供了优秀的动态组件激活时机, 比如可见激活, 立即激活, 延迟激活, 媒体查询激活, 手动激活等等
- 优秀的部署能力, 不仅可以优雅降级到SSR还可以部署到Vercel,Cloudflare等边缘平台

> 开始演示代码

Astro代码演示



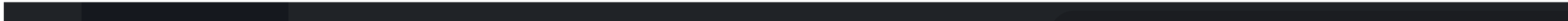
Island框架盘点 - Qwik

> 有一个很有趣的事情是, 在Astro官网中比较了多个框架的功能和性能, 它唯独没有与Qwik进行比较;

- 称之为最细节的Island框架并不夸大, 它拥有颗粒度最细的部分注水功能, 在其余大部分框架都在以组件为单位做激活工作时, Qwik可以把激活放到一个函数单位上
- 基于jsx语法进行开发, 并且和其他框架一样, 可以很好的和ts,tailwindcss等现代工具结合
- 在应用启动时上做了很多优化, 它尽可能的延迟执行和下载脚本, Qwik只需要1kb的运行时代码就可以让整个应用程序动态组件进行激活, 使其交互

> 开始演示代码

Qwik代码演示



Island框架盘点 - Fresh

> Fresh是deno核心作者开源的一套“下一代web框架”, 它基于deno且仅使用Preact作为UI层渲染

- 基于deno, 有安全性, 以及高性能的运行时(jsx & ts)
- 因为运行时是deno缘故, fresh是没有构建流程的, 而且island组件都是异步即时加载(JIT)
- 同样由于deno, 它对部署平台非常苛刻, 如果你选择在边缘部署, 就只能选择官方的Deno Deploy
- fresh的核心就是路由系统 + 模板引擎, 所以页面都是在服务端即使渲染的

个人不推荐使用fresh部署你的网站, 因为从以下几个角度出发

- 基于deno的生态, 需要一定的学习成本和踩坑成本
- 它仅支持preact作为你的UI渲染
- 孤岛组件在客户端激活策略单一

你应该选择细度更高的孤岛吗?

> 你已经了解了 3 个孤岛框架的特点, 我们将着重分析Astro以及Qwik的优势, 它们在激活组件策略上有很大的不同;

Astro:

```
<Card client:visible/>
```

Qwik:

```
<input
  onInput$={(event) => {
    const input = event.target as HTMLInputElement;
    state.name = input.value;
  }}
></input>
```

> 如果你的网站中需要针对动态孤岛组件进行非常细度定制化, 比如在“联系我们”组件中, 输入邮箱的操作是低频的, 其余组件是静态组件, 我们就可以使用Qwik框架将性能损耗压到最小; 但是反之你的网站有大量的交互组件, 而且你希望有多种激活策略(且可以用多种UI框架去构建), 那么Astro就是你的最佳选择;

国内外前端技术走向

> 不知道你有没有注意到, 从APP的潮流出现, 瓜分了大量的web流量之后, 国内外前端技术就走了不同的道路, 国外前端技术往往更注重用户体验, 国内前端技术却更注重业务需要和技术快速变现;

- 以前的时代, 我们都是用类jquery/jsp/php去完成前端, 出现了很多由服务端渲染的页面, 到此国内外研究的技术是一样的.
- app的出现将web流量抢占, 按理说我们应该提升web体验去留住用户; 但是我们却发明了APP中打开APP(小程序), 而且每一家平台的小程序标准都不一样, 所以又有了一类的跨端小程序框架去帮助我们完成需求.
- 而国外开发者为了留住web用户进而推进了PWA规范, 从nuxt, next, remix这一类服务端/同构渲染框架开始, 又演变了如今的island架构, 他们真正的目标是为了提升用户体验, 将业务最主要的功能(比如购买按钮)最先展示.

> 用户最先看到核心业务, 能给企业带来什么?

island架构带来了什么?

页面访问速度很重要么? 对于商业网站来说的确是的, 缓慢的访问速度可能会损失数百万美元, 在web.dev中有一份例子:

- 对于Mobify，每 低100 毫秒的转化速度提高 1%,主页加载速度每降低 100 毫秒,基于会话的转化率就会增加 1.11%, 平均年收入增长近 380,000 美元.
- 当 AutoAnything 将页面加载时间缩短一半时,销售额提高了50%，销售额提高了12%到13%。
- Pinterest的注册速度提高了40%，从15%的注册时间缩短了40%，搜索引擎流量和注册次数增加了15%。
- BBC发现，他们的网站每加载一秒钟，就会额外失去10%的用户。

> 所以, 请重视你的互联网产品页面加载速度以及TTI时间