Vue SSR 在 bilibili 国际版的实践

@敖天羽

自我介绍 @敖天羽

- 前后端反复横跳的全干工程师
- bilibili 国际版 Web 站技术负责人
- 微博：敖天羽 / GitHub：cswvolf
- 博客：https://www.codesky.me

我们为什么需要 SSR

我们为什么需要 SSR

- SEO：白嫖的流量，Web 的一大流量入口
- 首屏性能：更优的性能和首屏体验，用户更早看到数据

SSR & SSG

- SSG：静态生成页面，所有用户获得的数据都一样，不经常变化
- SSR：动态渲染页面，千人千面，动态实时变化的数据

SSR 设计

SSR 现成实现

- Nuxt
- Quasar
- Vite SSR

在做 SSR 时，我们要注意什么

用 SSR 模式进行开发

```
// Create Vite server in middleware mode and configure the app type as
// 'custom', disabling Vite's own HTML serving logic so parent server
// can take control
const vite = await createViteServer({
  server: { middlewareMode: true },
  appType: 'custom'
})
```

vite ssr 中的 dev mode

用 SSR 模式进行开发

- 日常使用 SSR mode 进行开发
- 配合降级测试 CSR 的情况是否 work

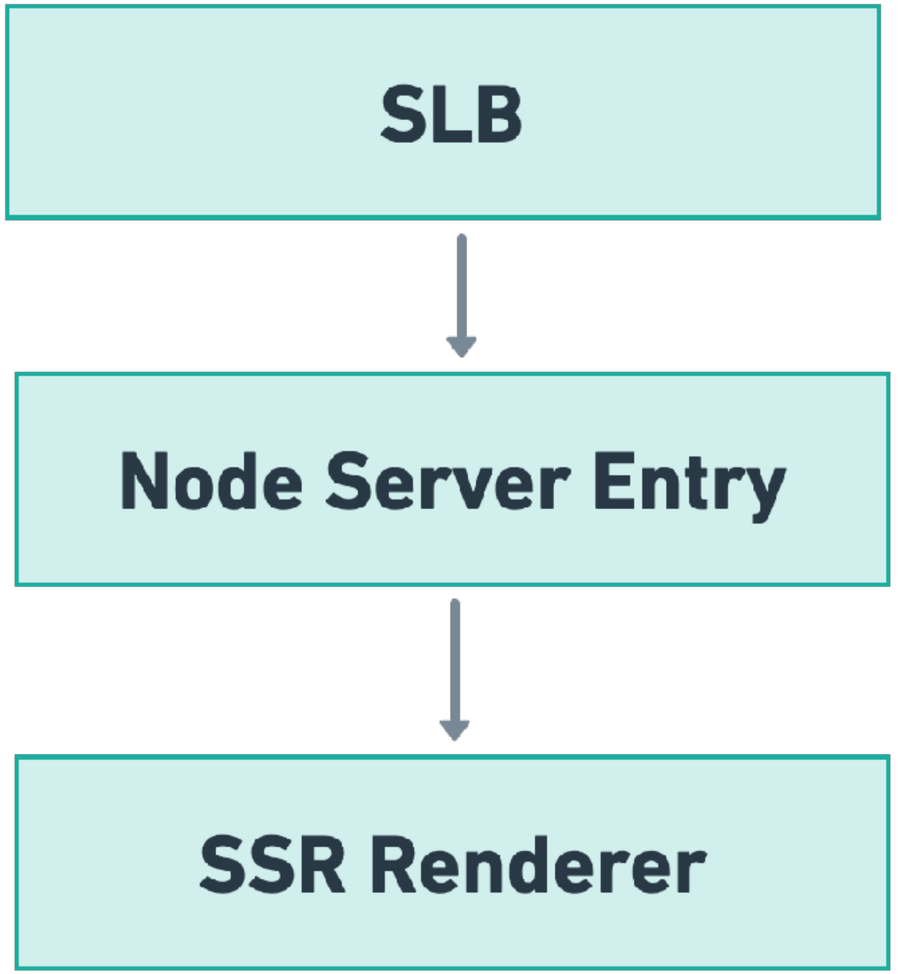
```
"dev:client": "vite --config build/vite.dev.config.ts",
"dev:server": "NODE_ENV=development ts-node src/server",
```

注意作用域

- 在每个请求中为整个应用创建全新的实例，避免状态污染
- router、store、context

请求上下文

- 如果 SLB 超时时间是 800ms，那么超过 800ms 的任务没有意义
- 类似于 Golang 的 context.WithTimeout()
- 使用 provide / inject 建立上下文体系



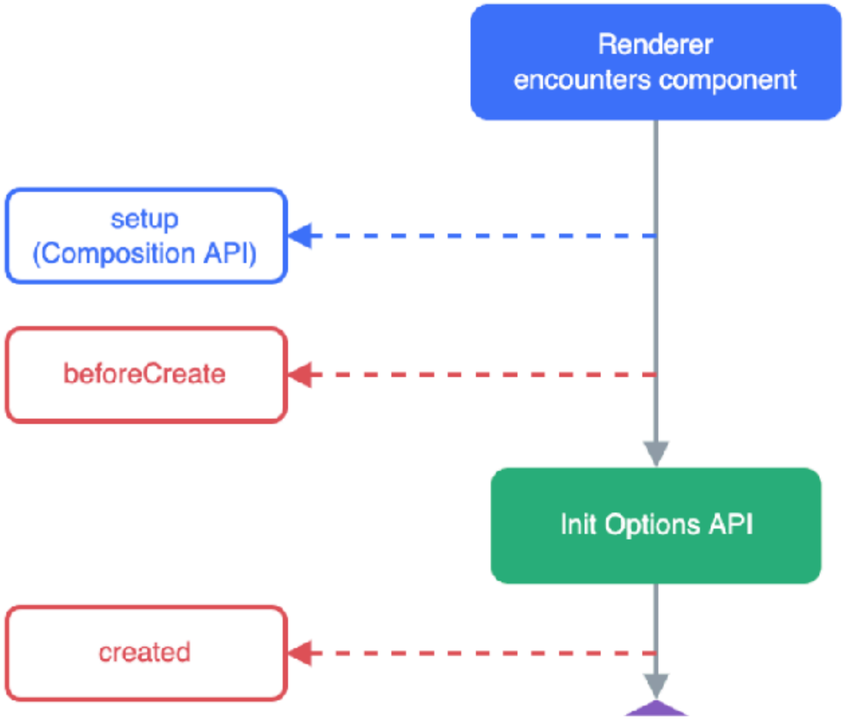
请求上下文

```ts
export const setFetchInstance = (app: App, options: SetFetchOptions) => {
  const { context, interceptor, timeout = 2000, keepAlive, maxSockets, useParamsFn, ssr, baseUrl } = options
  if (ssr) {
    // 针对 context 实例化的 Instance
    const abortController = new AbortController()
    app.provide(FetchInstance, Ajax( options: {
      baseUrl,
      ctx: context,
      abortController,
      timeout,
      interceptor,
      keepAlive,
      maxSockets,
      useParamsFn,
      ssr
    }))
    setTimeout( handler: () => {
      abortController.abort()
    }, timeout)
  } else {
    app.provide(FetchInstance, Ajax( options: { timeout, useParamsFn, ssr, baseUrl }))
  }
}
```

请求上下文

```ts
export const useFetchInstance = (config: AxiosRequestConfig = {}, ajaxOption: AjaxOptions = {}) => {
  // 一个实例，传入 baseUrl 来覆盖请求默认值
  return inject<(config: AxiosRequestConfig) => AjaxT>(FetchInstance, Ajax(ajaxOption))?.(config) as AjaxT
}
```

生命周期与 Context 函数



onServerPrefetch

- 当组件实例在服务器上被渲染之前要完成的异步函数。
- 这个钩子仅会在服务端渲染中执行，可以用于执行一些仅在服务端才有的数据抓取过程。

onServerPrefetch

```ts
/**
 * useFallback SSR 获取不到就降级在 CSR 拿数据
 * @param fn 需要在 onServerPrefetch 中执行的函数
 * @param dependencies 如果指定依赖项为 false，就会在 onMounted 再次获取数据
 */
export const useFallback = (fn: () => Promise<void>, dependencies: unknown[]) => {
  onServerPrefetch( hook: async () => {
    await fn()
  })

  onMounted( hook: () => {
    if (dependencies.map(v => !!v).includes(false)) {
      fn()
    }
  })
}
```

useSSRContext

- 一个运行时 API，用于获取已传递给 renderToString() 或其他服务端渲染 API 的上下文对象。

```
if (import.meta.env.SSR) {
  const ctx = useSSRContext()
  // ...给上下文对象添加属性
}
```

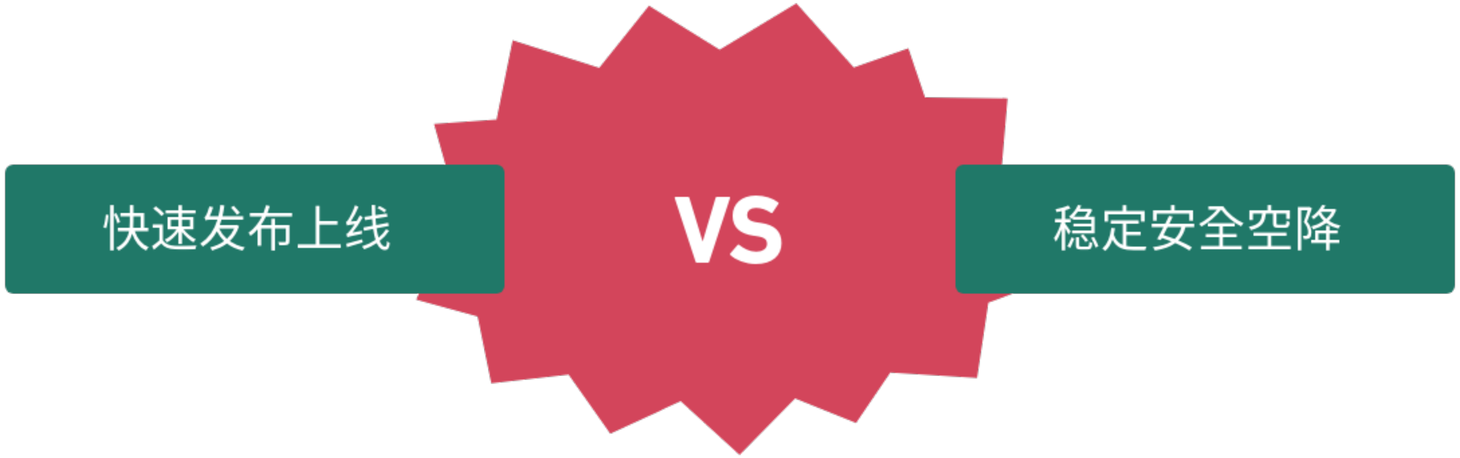
其他注意事项

- 使用无全局 window / document 的依赖
- 减少复杂度高的运算，避免 O(n^2)

系统化建设

构建-发布方案

基于业务属性进行选择



两种方案

- 全量资源发布：全量代码一起构建跑在容器内
- SSR 与 CSR 整散结合：SSR 构建 server，剩下的在 CSR 构建时进行，同步到容器中

全量资源发布

优点

1. 完整独立可单独运行的系统

2. 可以利用 Docker 特性直接回滚

3. 可以基于容器做灰度

4. 构建包时刻都是最新的（依赖最新）

缺点

1. 独立运行，因此无法保证最终一致性

2. 无法检测对应的降级是否真正可用

3. 整个发布时间相对较长

SSR+CSR 整散结合

优点

1. 可以保证最终一致性

2. 可以利用 CSR 模版检测资源是否有效

3. 发布和回滚更快

4. 支持服务解耦

缺点

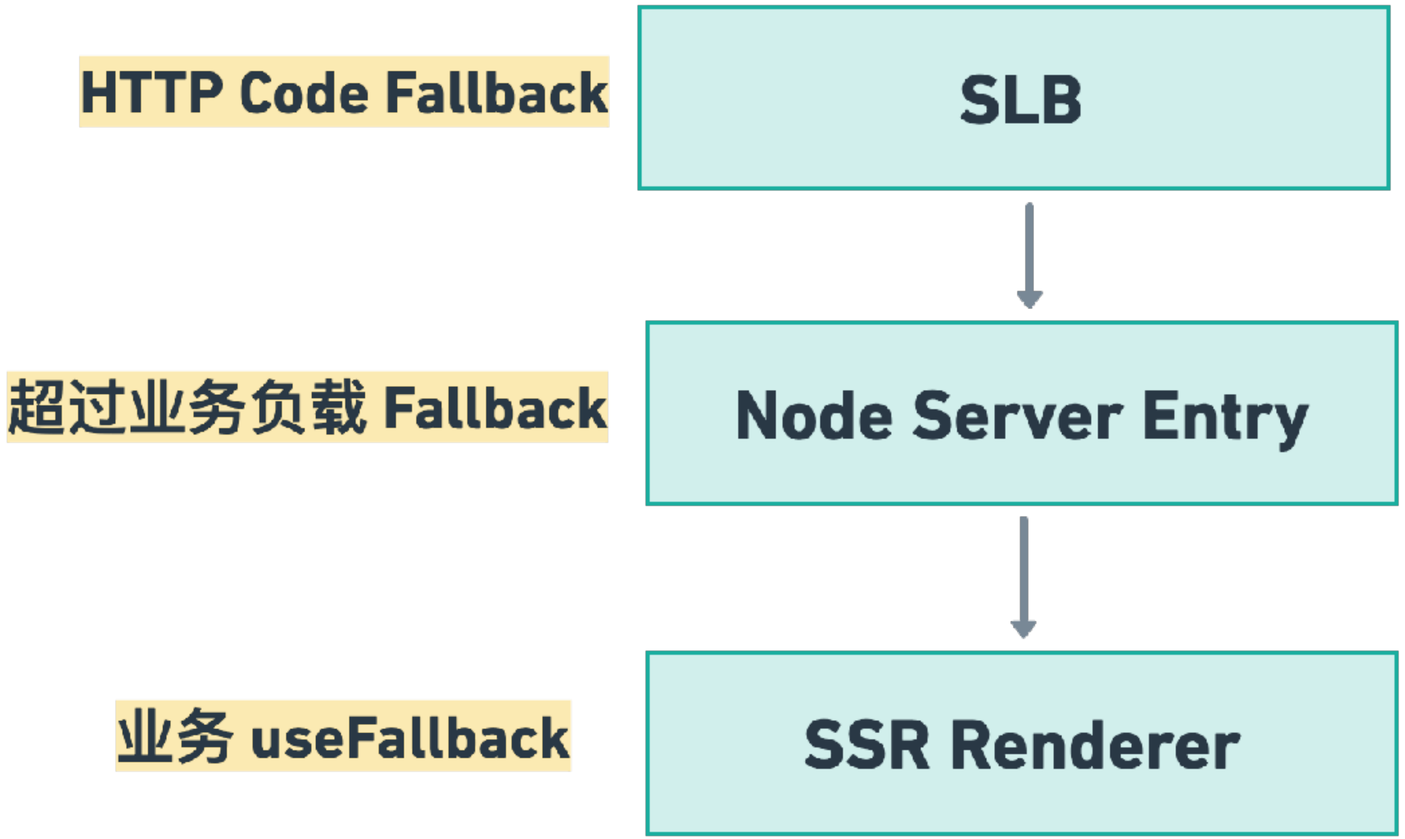
1. CDN 无法保证资源访问安全性

2. 网络 IO 可能会导致启动失败

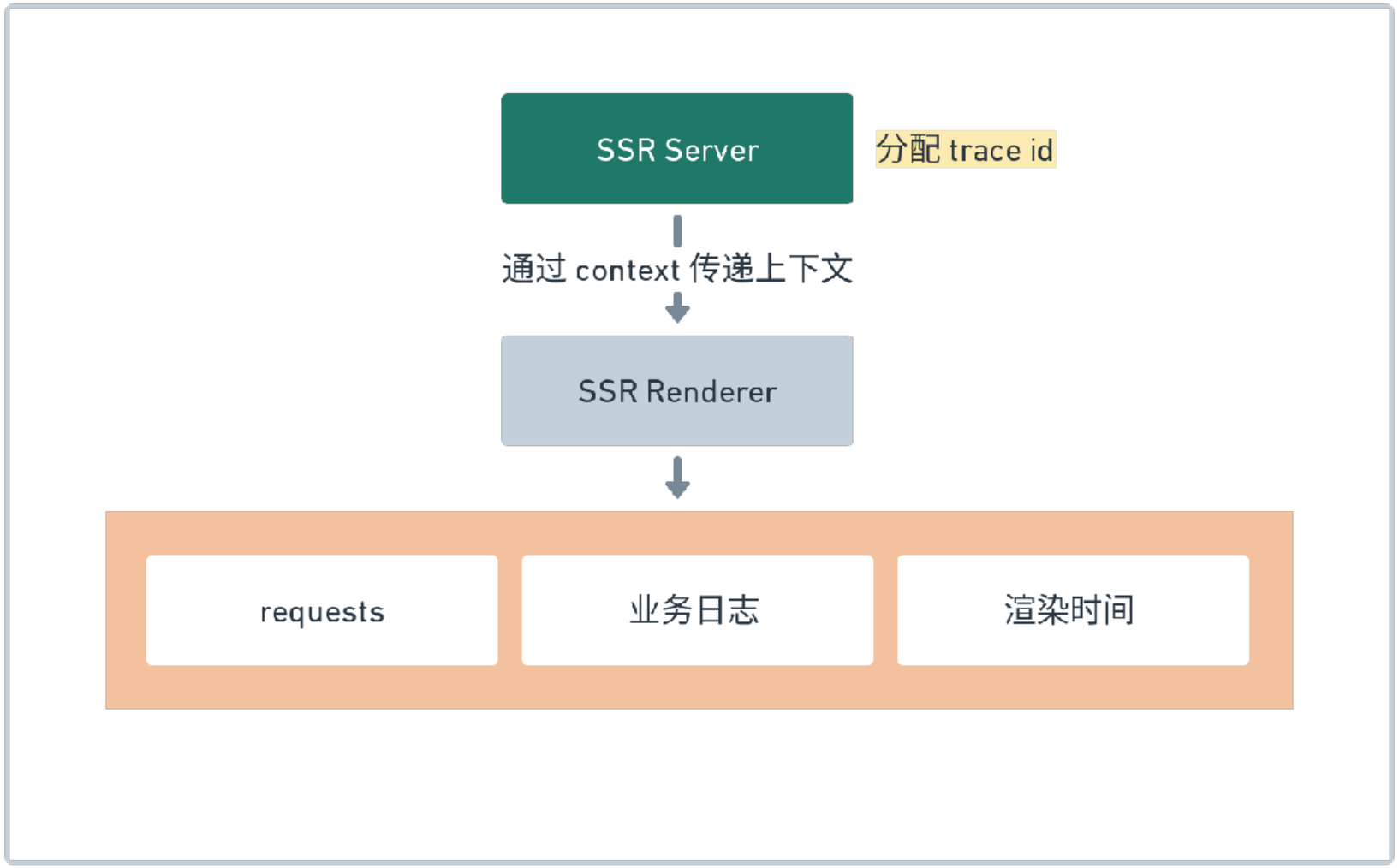
3. 无法针对 SSR 进行资源灰度

4. 可能会忘了更新依赖导致翻车

降级策略


HTTP Code Fallback
SLB
超过业务负载 Fallback
Node Server Entry
业务 useFallback
SSR Renderer

日志与告警


SSR Server
分配 trace id
通过 context 传递上下文
SSR Renderer
requests
业务日志
渲染时间

缓存设计
你是不是真的需要缓存

总结

- 快糙猛的一个 SSR：使用 Vue 生态中的开源系统
- 完整的 SSR 应用：考虑到系统的稳定性、健壮性和可观测性进行基于自有基建的设计

“系统设计是没有银弹的”

——敖天羽（？）

Reference

- Vue 官方文档：https://cn.vuejs.org/guide/scaling-up/ssr.html
- Vite 官方文档：https://cn.vitejs.dev/guide/ssr.html
- 前端 SSR 系统设计思路谈：https://www.codesky.me/archives/frontend-ssr-system-design.wind

敖天羽

上海 嘉定

扫一扫上面的二维码图案，加我为朋友。

哔哩哔哩技术

微信扫描二维码，关注我的公众号