Next.js

layout接收children、params，还有平行路由插槽@slot

template接收children

error接收error、reset，，必须客户端组件，可以使用useParams

loading不接收

not-found不接收

page接收params、searchParams

default接收params，配合平行路由使用，

generateStaticParams() 用于动态路由转换静态渲染

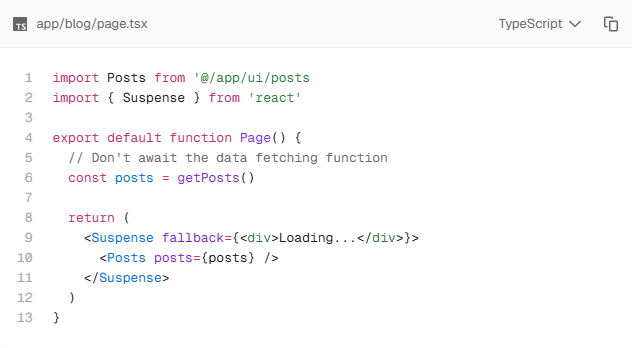
<Image/> 推荐使用，next.js 自动适配宽高度，做了延迟加载和模糊占位符

注意点 1、使用远程图像时需提供宽高，避免发生布局偏移；

2、next.config.ts中可配置远程图像url格式以防止恶意使用

Server Components 获取数据使用fetch，注意使用await

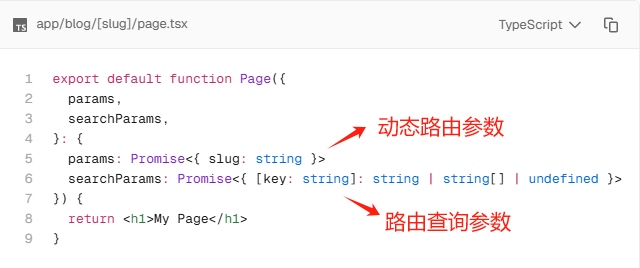
Client Components也可以在useEffect中直接使用fetch获取数据，但是不建议这样做，规范的做法是在Server Component中获取数据流式传入客户端组件渲染，例如：（但实战直接使用swr）



Stream流式渲染有两种实现方式:

1. loading.tsx ,适用于layout和page
2. <Suspense>局部渲染

page.tsx文件接收参数结构：



Server Components从上图searchParams获取URL查询参数，Client Components则通过useSearchParams获取查询参数

ISR重新验证数据有两种方式：

1. 基于时间的重新验证



1. 按需重新验证



表单提交：

useActionState用于在执行 action 时显示loading指示器

useFormStatus 在执行操作时显示loading指示器

服务器端表单验证可以使用zod库

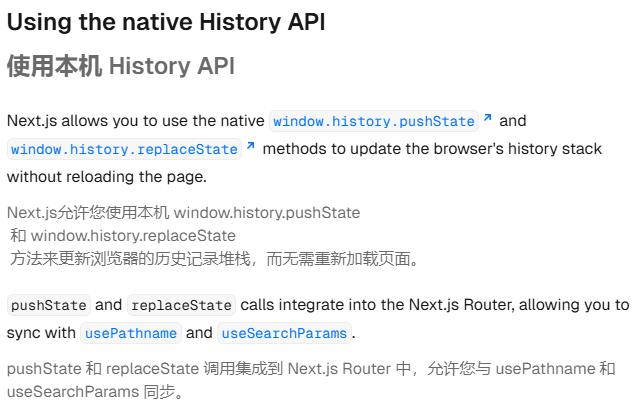
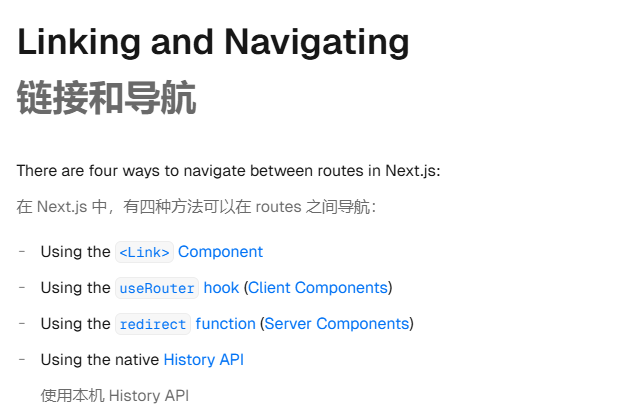
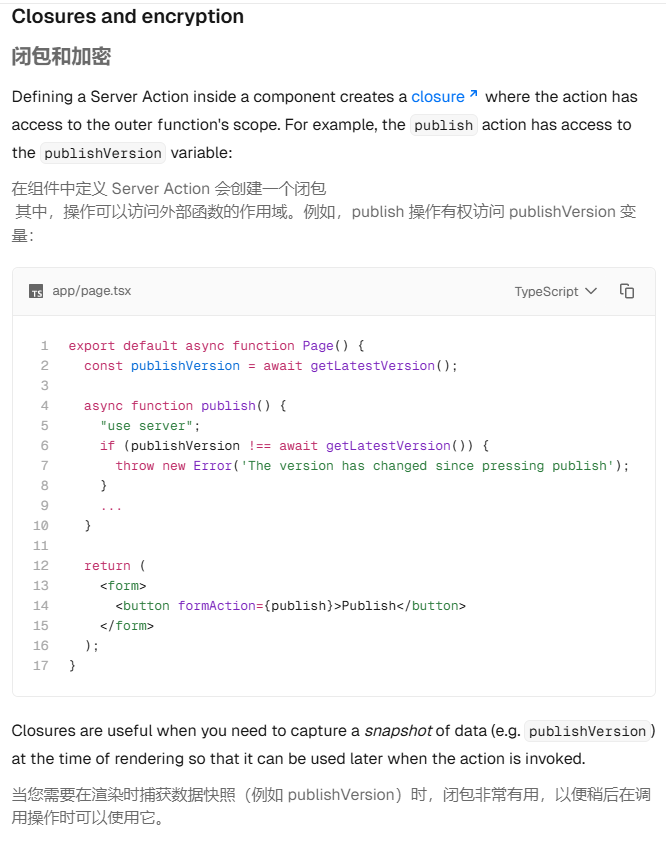
处理预期错误：

1. Server Components -> fetch 直接if
2. Server Actions中返回错误message

处理无法预期错误，使用错误边界：

1. error.tsx
2. global-error.tsx

乐观更新：useOptimistic



History可以在不重新加载页面的情况下对searchParams和pathname进行修改，只会影响客户端[usePathname](https://nextjs.org/docs/app/api-reference/functions/use-pathname) and [useSearchParams](https://nextjs.org/docs/app/api-reference/functions/use-search-params) 监听下的值，Server Component的props中的searchParams并不会监听到值的变化（因为页面没有重新加载）

重定向分为五种：



Redirect在事件函数中使用，permanentRedirect永久重定向（修改用户资料后不支持回退回到修改之前的页面），useRouter客户端使用，next.config.ts（改动了路由结构，重定向之前的旧路由），NextResponse.redirect（中间件权限控制重定向及大量重定向需求）



路由组的功能：分组以支持特定路由共享layout，loading；支持多个根布局创建多站点服务

动态路由作为 params 属性传递给 layout、page、route 和 generateMetadata 函数。

并行路由适合高度动态的页面。在插槽内添加layout，以允许用户独立导航槽，适合做标签页组，切换导航

并行路作为props传递给共享的父layout

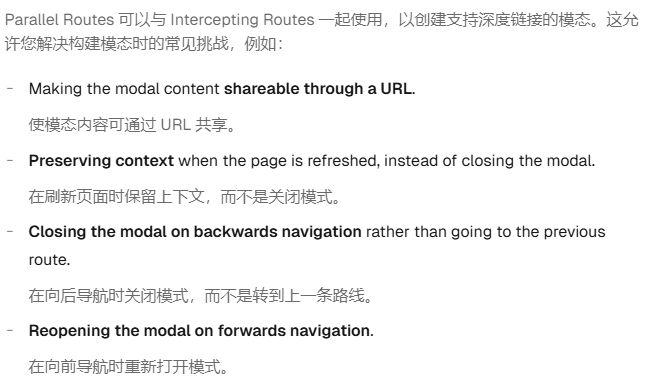
并行路由：不能在同一路由分段级别拥有单独的静态和动态插槽。如果一个槽是动态的，则该级别的所有槽都必须是动态的

注意：并行路由需要注意软导航和硬导航（创建完备的default.js），@children是隐式插槽，也需要default.js

useSelectedLayoutSegment（用于激活的导航UI） 和 useSelectedLayoutSegments（用于面包屑导航） 可以读取下一活动路由段，都接受 parallelRoutesKey 参数，该参数允许您读取指定slot 内的活动路由段，在客户端组件中使用并导入父layout中

注意：useSelectedLayoutSegments返回的分段包括 Route Groups，您可能不希望将其包含在 UI 中。你可以使用 filter（） 数组方法来删除以括号开头的项目

Parallel Routes 可以与 Intercepting Routes 一起使用，以创建支持深度链接的modal弹窗



实现国际化：

1、用动态路由[lang]包裹所有路由。

2、使用generateStaticParams实现静态渲染

3、在middleware中拦截转发，配合NextAuth.js的auth包装器

Data Fetching：

强制动态渲染：



通常会使用 cookie、headers 或从 page props 读取传入的 searchParams 等函数，这些函数将自动使页面动态呈现。在这种情况下，无需显式使用 force-dynamic

Cache:’force-cache’可缓存相同fetch请求



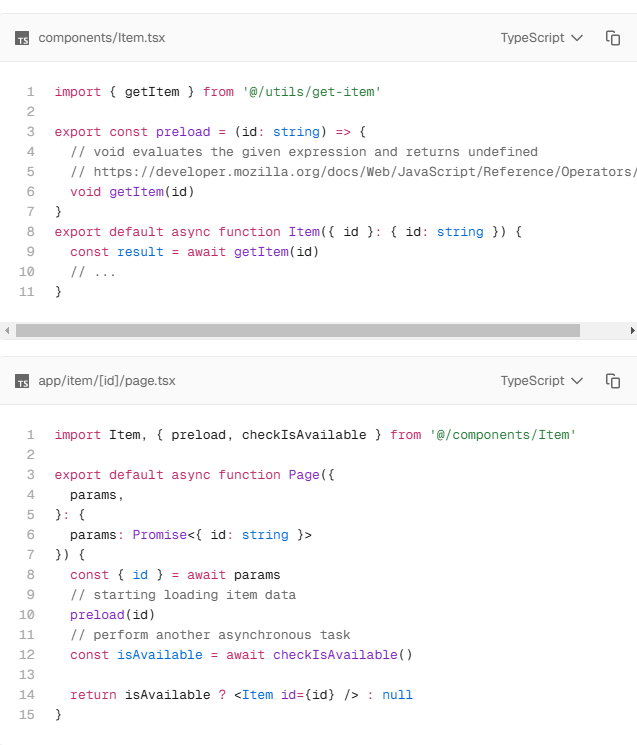
重新验证数据：ISR

数据获取分为两种，顺序获取和并行获取：

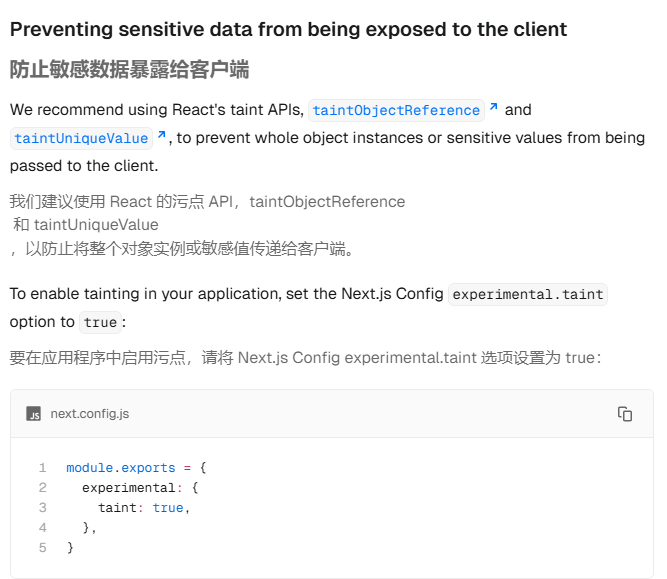
1、顺序获取：调用一个 fetch 取决于另一个 fetch 的结果

2、并行获取：默认情况下，两个fetch同时发出请求，但是，由于 async/await 的性质需逐步等待完成，所以使用await Promise.all

预加载数据：



注意：并行获取和预加载数据都可以防止瀑布流



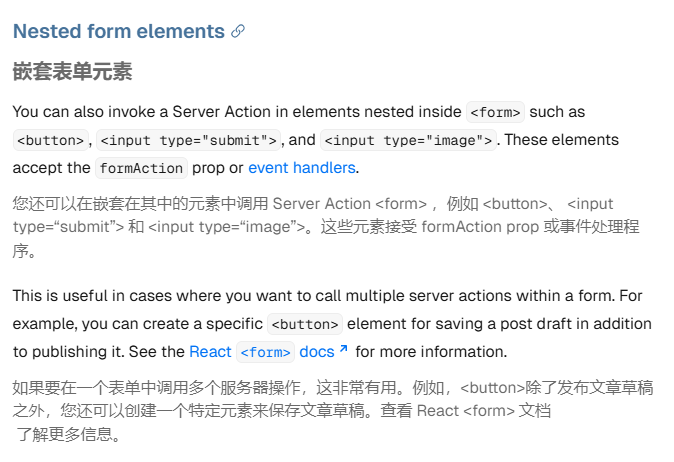
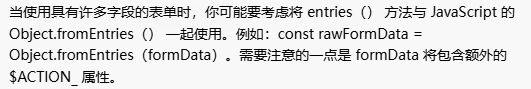
Server Action:（用于处理应用程序中的表单提交和数据更改）

可以使用 React “use server” 定义 Server Action

命令。您可以将该指令放在异步函数的顶部，以将该函数标记为 Server Action（内联），或放在单独文件的顶部，以将该文件的所有导出标记为 Server Actions。

Server Action也可以作为props传递给客户端组件，不过一般配合useActionState使用。

表单提交：



注意ISR：



generateStaticParams只会生成已知的id的页面，当有新id时，就会遇到404，这个配置可以动态按需生成



交错服务端和客户端组件：由于渲染策略是先渲染服务端，而后渲染客户端，所以正常情况下， 无法将服务端直接导入客户端使用，只能**将服务器组件作为 prop 传递给客户端组件，由于服务端组件远早于客户端渲染，这种提升模式可用于避免在父组件重新渲染时重新渲染嵌套的子组件**

PPR部分预呈现渲染可以将动态和静态组件合并到一个 HTTP 请求中，从而提升渲染效率，实验性功能

<Image>：

1、本地图像和远程图像，可以使用Patterns对图像来源URL进行限制

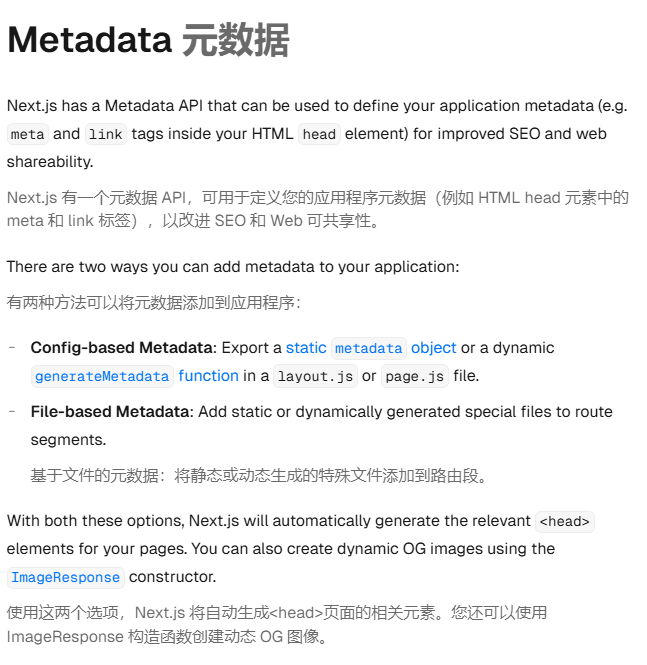
2、使用自定义装载机Loaders可以实现远程图像相对url的引用，

3、使用priority标记最大图像优先加载绘制，有助于LCP优化

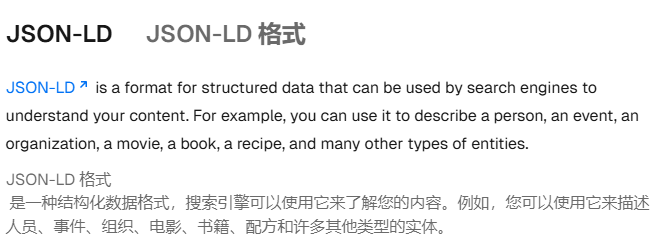
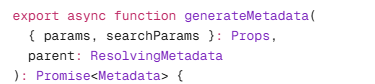
4、src可以直接使用/访问public中的静态资源

<video> 和<iframe>：video使用本地视频和上传至Vercel blob上的视频，iframe链接外部平台视频，引入外部视频时最好使用<Suspense>流式加载

Fonts：可以引用google等外部字体以及本地字体（localFont），可以配合**Tailwind CSS**使用



generateMetadata接收两个参数：



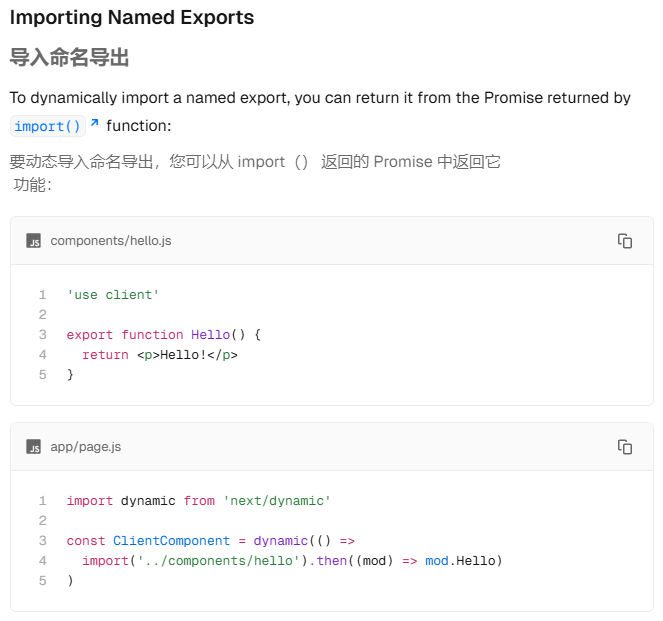
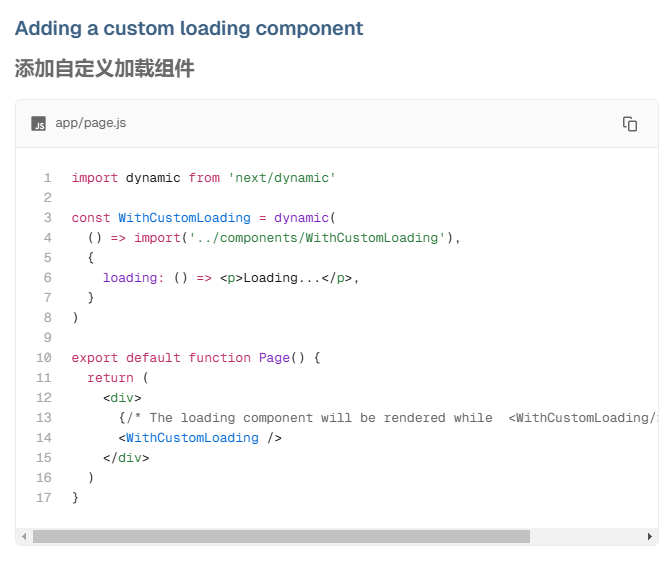
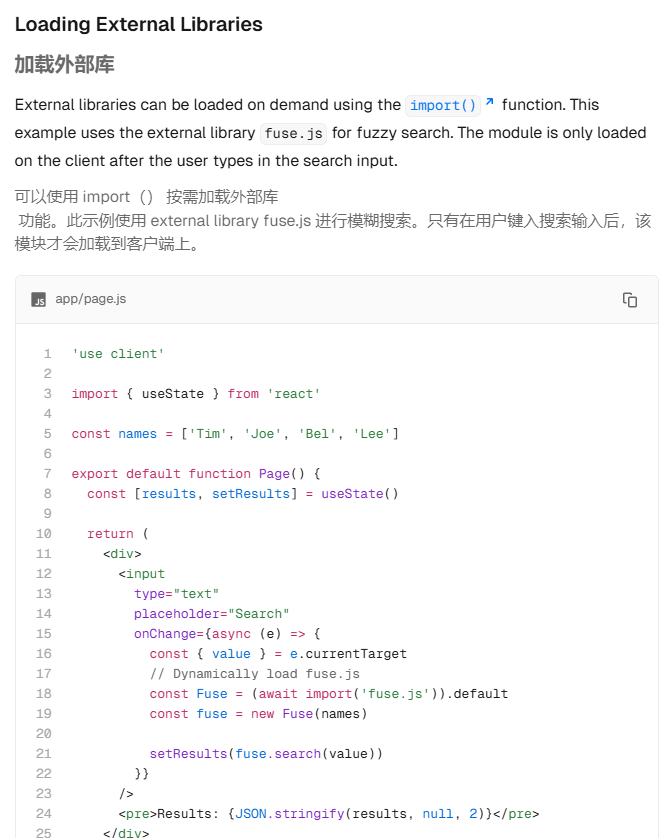
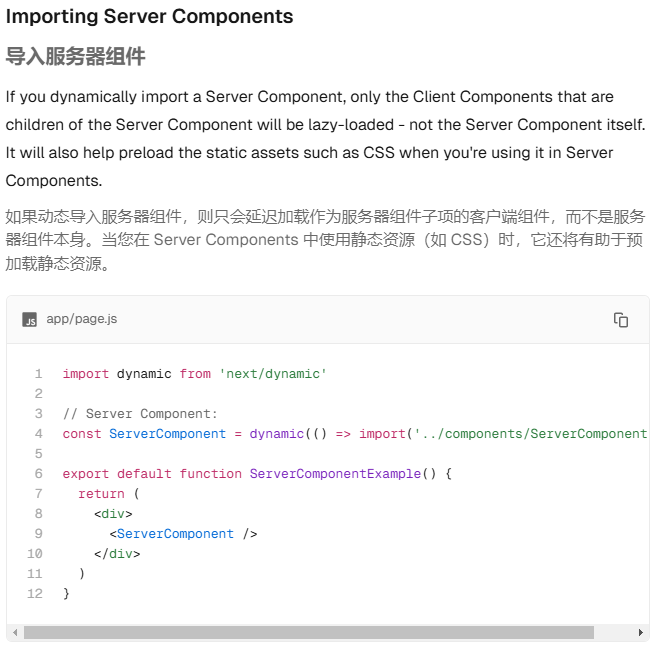
JSON-LD帮助搜索引擎更好地理解网页内容，Metadata用于网页的基本信息和社交媒体展示

<Script>引入第三方脚本或内联自定义脚本，strategy定义加载时机，onLoad、onReady等事件处理只能在客户端组件中使用

Next.js中Server Components 默认情况下会自动进行代码拆分 ，并且可以使用流式处理将 UI 片段从服务器逐步发送到客户端。所以需要懒加载的只是客户端组件。

Next.js提供next/dynamic可直接实现客户端组件懒加载，其原理React.lazy()结合Suspense。

客户端组件使用懒加载默认预渲染（SSR），可以通过{ ssr: false }禁用



使用平行路由配合error.tsx可以实现局部组件error

**Next.js 项目写 Tailwind CSS 基本都会遇到的两个问题**

问题 1：动态类名问题，需要保证类名完整存在

问题2：类名优先级问题，利用tailwind-merge，clsx及cva插件解决

next/after函数会在响应完成流式处理后安排要处理的工作，从而在不阻塞主要响应的情况下运行辅助任务，它可以在 Server Components 中使用（包括 generateMetadata

）、服务器操作、路由处理程序和中间件。

SPA单页面应用有SEO问题，不更新URL地址，而next.js项目既更新url，又使用SSR渲染，弥补了 SPA 在 SEO 上的问题，主要归功于next的导航概念

Next.js中解决跨域一般用服务器组件发送请求或者next.config.js及middleware重定向

CSR客户端渲染，首选服务器上预渲染一次，然后实际渲染时进行水合，水合不一致会导致水合错误

常见水合错误：

元素嵌套错误导致水合失败，客户端组件使用客户端api，导致预渲染跟实际渲染不一致从而水合失败（localstorage），使用时间api，浏览器插件导致

解决方案：将客户端api放在useeffect中使用，使用关键字suppressHydrationWarning忽略，使用hooks、利用next懒加载daynamic禁用组件ssr

十大常见错误：

1. 服务端组件调用路由处理程序，没必要这样做，服务器组件可直接调用
2. 调用接口，没有使用动态api,比如cookie，header等，会导致缓存数据不更新
3. 服务器及客户端组件都不应该调用接口，而是使用server actions调用
4. Suspense 组件放在外围
5. 尽量使用next官方api获取cookie，headers等
6. 使用Context 应该在根布局中使用，导入Context 并嵌套children
7. 使用 不必要的"use client"，服务端组件导入至客户端组件中会视为客户端组件进行渲染
8. 服务端组件如果要传入客户端，应当使用props传入
9. 数据更新后及时使用revalidatepath等函数更新数据
10. 不应该在try/catch中使用redirect，应当放置在finally 或try/catch之后使用

懒加载主要针对客户端

国际化，利用动态路由包裹所有路由段，移植根路由至动态路由下，可以配置词典，middwaler进行国际化转发，配合nextauth.js使用，genaraetestaticparams进行静态生成页面

notfound ，路由地址不匹配时使用跟not-founde默认404页面，组件中使用not-found函数触发同级notfounf页面，因而如果设置了多站点服务，有路由组的情况下，需要自定义40d4的话，

就需要利用动态路由的catchcall路由，捕获除根页面外所有页面用以触发notfounde函数

如果想要404页面不使用站点的layout布局，而是使用自己的，就需要在路由组下再一次分组加载自己的根布局

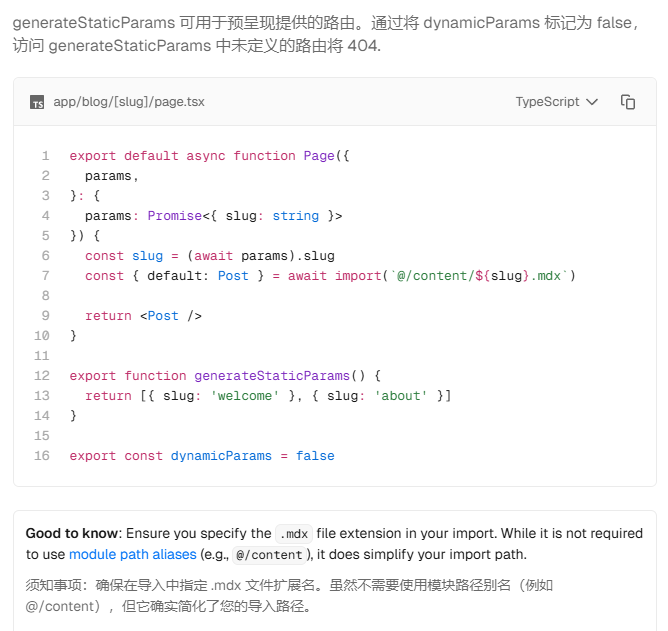
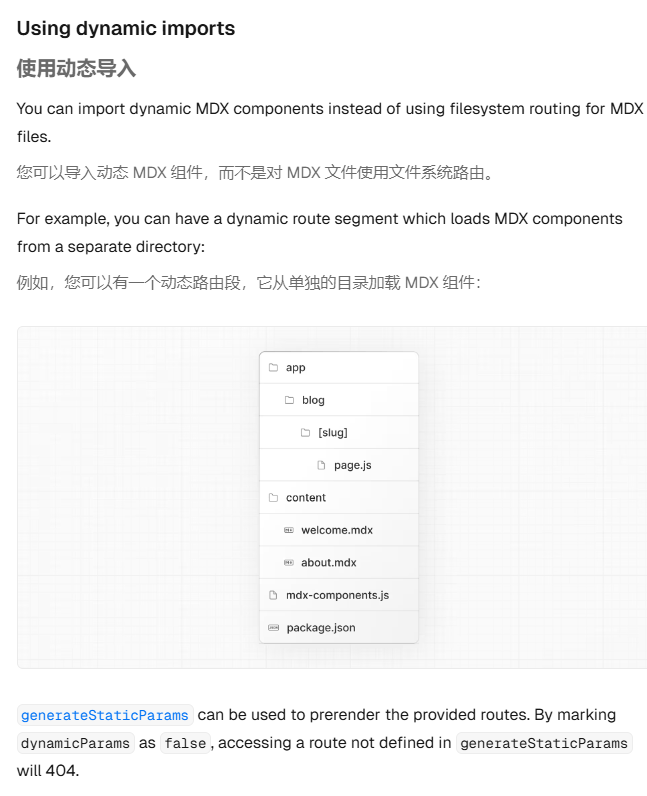
Next.js中缓存有哪些，由于next15之后去掉了很多缓存，所以，实际的缓存，

1. fetch中使用force-cahce接口缓存
2. SSG缓存并使用revalidate实现ISR更新数据
3. React cache使用next官方只推荐了一种方式，结合Import server-only及预加载数据

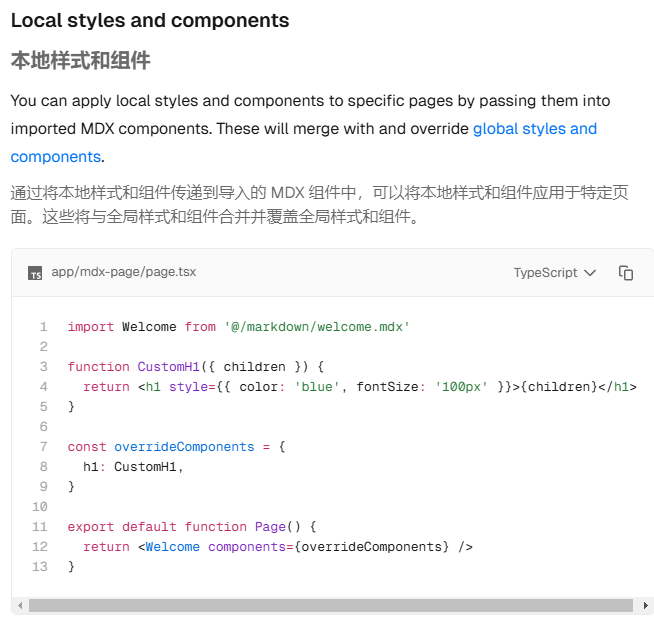
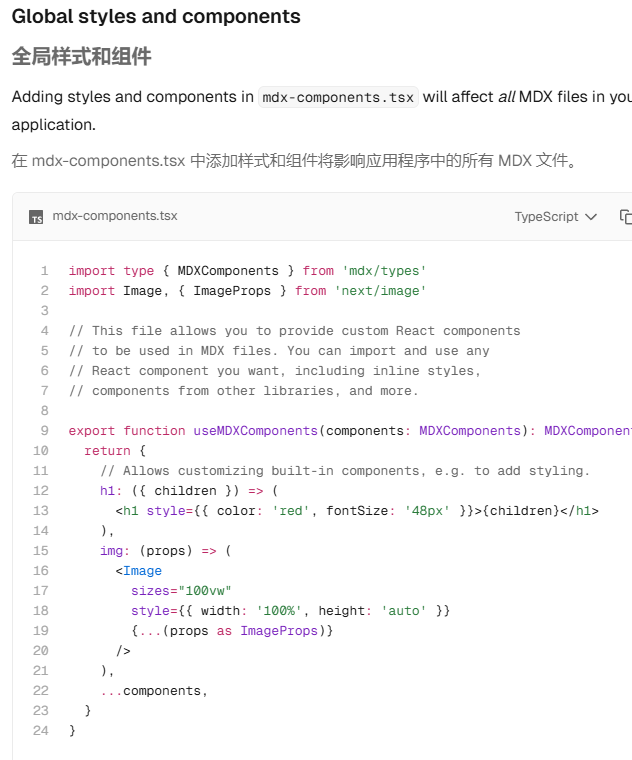
所有环境变量都应该在vercle上配置，并且可以从vercel上拉下来写入到.env.local中

引入MDX文件可以作为路由导航，可直接导入其他组件，.mdx也可以作为组件导入至其他文件，Next.js 插件处理将 markdown 和 React 组件转换为 HTML

MDX动态导入用法：

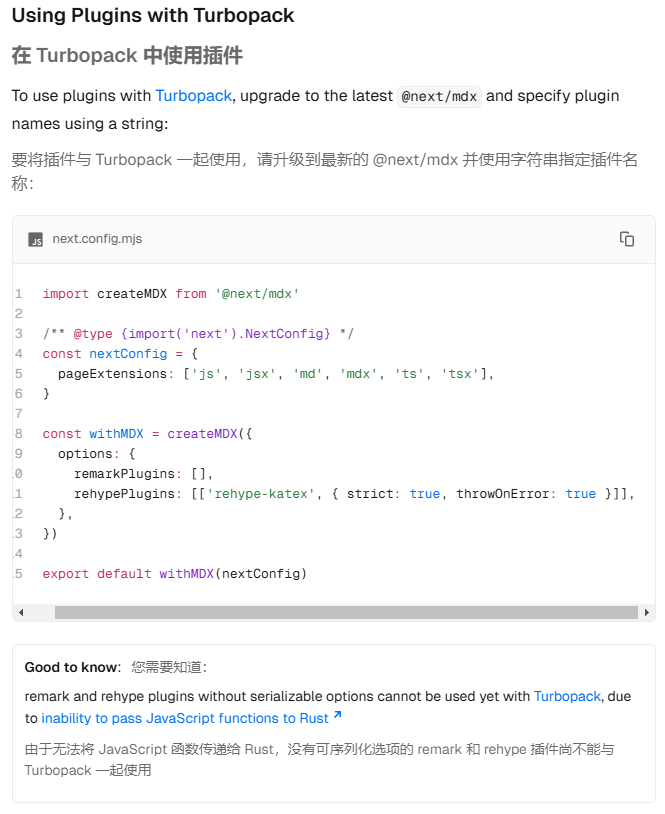
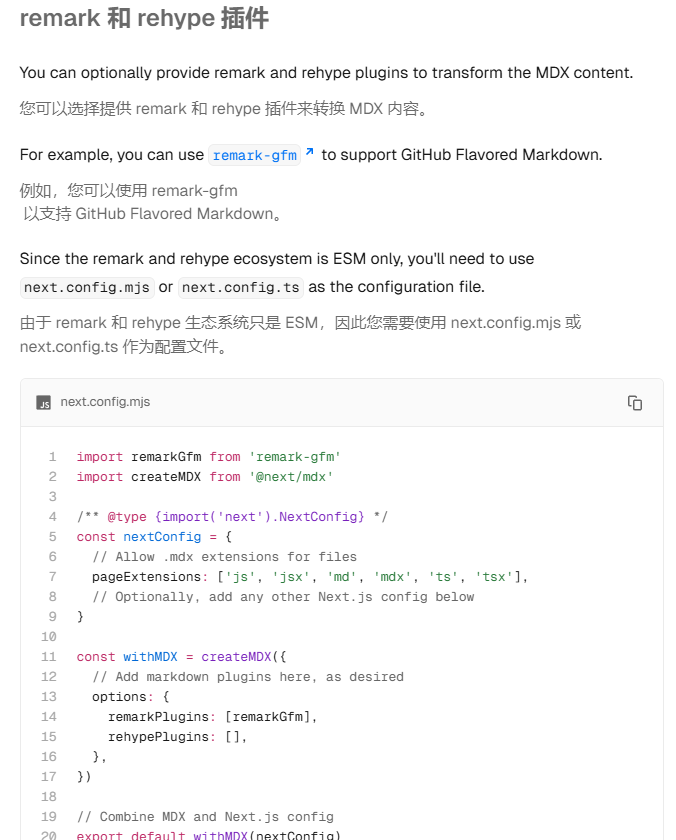


设置mdx样式的三种方式



MDX导出metadata可以直接使用：

export const *metadata*= {  
 author: "zhangsan",  
 };



Next.js项目可以设置CSP内容安全策略，可通过once绕开CSP严格策略，使用middleware每次生成并注入Script，官网有demo，也可以直接在next.config.ts中配置**无 Nonces的应用程序**

Next.js内置的Form组件可以做两件事import Form from 'next/form'

1. action接收字符串以更新url及搜索参数并导航，且可以预取路由
2. action接收函数调用actions

**投入生产之前在lighthouse上跑一遍结合GTM检测性能数据等**

**部署后，vercel提供了性能，访客pv数据分析面包，日志等指标，还可以集成第三方工具，大部分收费，部分免费，部分一定额度免费**

**Vercel有一个生产清单可参考（在部署模块）**

**Multi-Zones微前端，旨在将不同域的服务组织在一起**

**Multi-Zones主要原理：在vercel上部署两个项目，将其中一个项目的域放在主项目的.env中，在next.config.ts中做转发，子项目还需要设置assetPrefix处理静态资产。**

**注意事项：1、使用a标签而不是link进行url跳转**

**2、如果需要动态重写的放在middleware中**

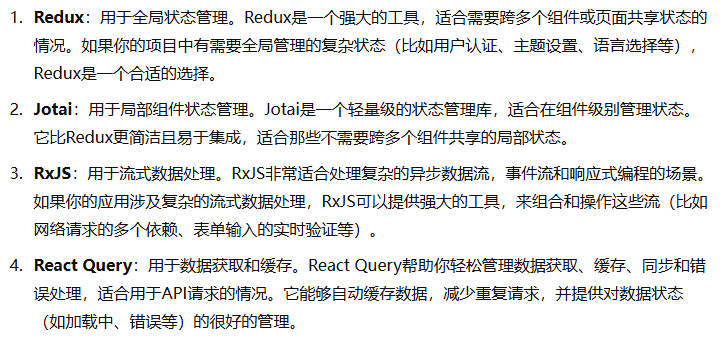
3、使用monorepo管理项目

4、需在nextConfig配置serverActions来源

userAgent扩展了request，可以获取设备信息，[**browser**](https://nextjs.org/docs/app/api-reference/functions/userAgent#browser)**浏览器信息，os操作系统，cpu信息等，一般在middleware中使用**

**NEXT\_PUBLIC\_前缀的env变量才可以在客户端访问**

**Env变量由于 webpack DefinePlugin 的性质，尝试解构 process.env 变量不起作用**

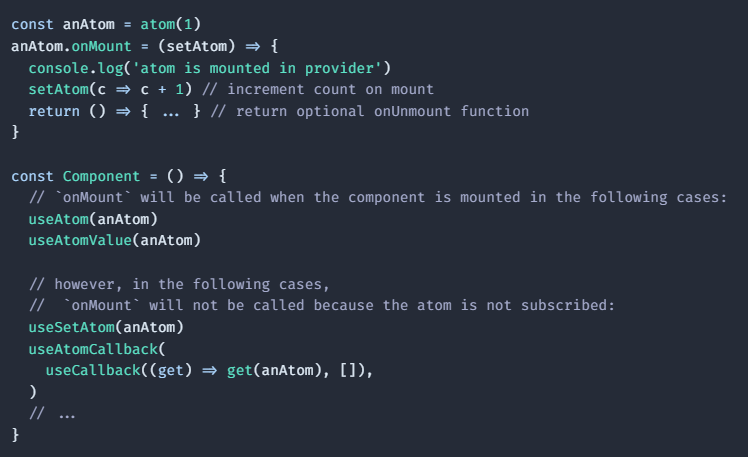


Jotai：四个核心：atom原子，useAtom，store，provider

1、Autom原子，派生原子（接收get,set）

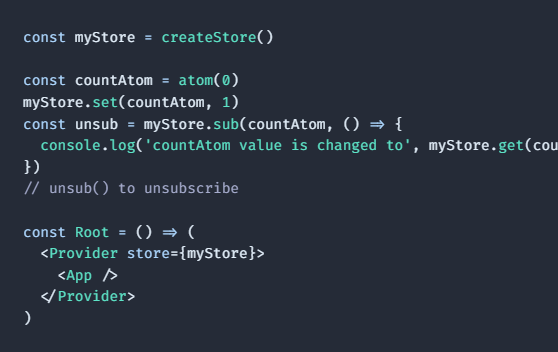
原子有两种：可写原子和只读原子。原始原子始终是可写的。如果指定了 write，则派生原子是可写的。

anAtom.onMount函数在首次在提供程序中订阅 atom 时调用，当它不再订阅时调用 onUnmount



2、useAtom，useSetAtom，useAtomValue

3、store结合provicer使用: createStore，只有store可以做到sub订阅消息及取消订阅

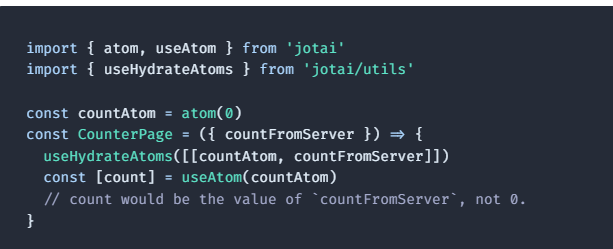


const defaultStore = getDefaultStore(); 全局默认store

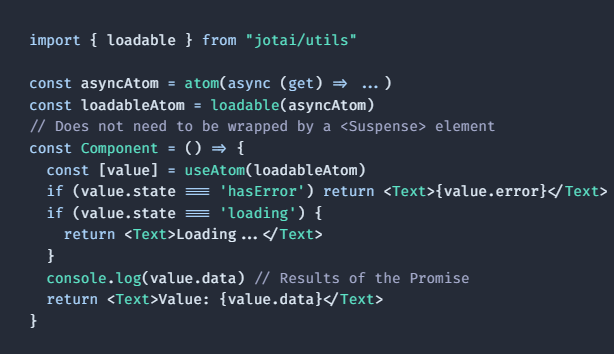
**jotai/utils包含的api**

1. atomWithStorage操作localstorage

**2、useHydrateAtoms SSR渲染时，从服务端组件传入客户端组件的props中获取值用于水合**



3、atom支持async异步写法，loadable用于包裹asynAtom



4、atomWithReset创建可reset重置的原子

**5、atomFamily原子工厂批量生产原子，简化代码：不用为每个不同的原子手动创建一个原子，减少了重复代码。**

**jotai扩展**

**jotai-tanstack-query：react-query扩展插件，某些请求返回的状态和值希望在局部组件中共享时可用**

**jotai-immer 扩展immer**

**jotai-effect副作用，observe主要监听以及store传递，需要手动挂载的atomEffect以及可以创建原子的withAtomEffect**

**observe:**



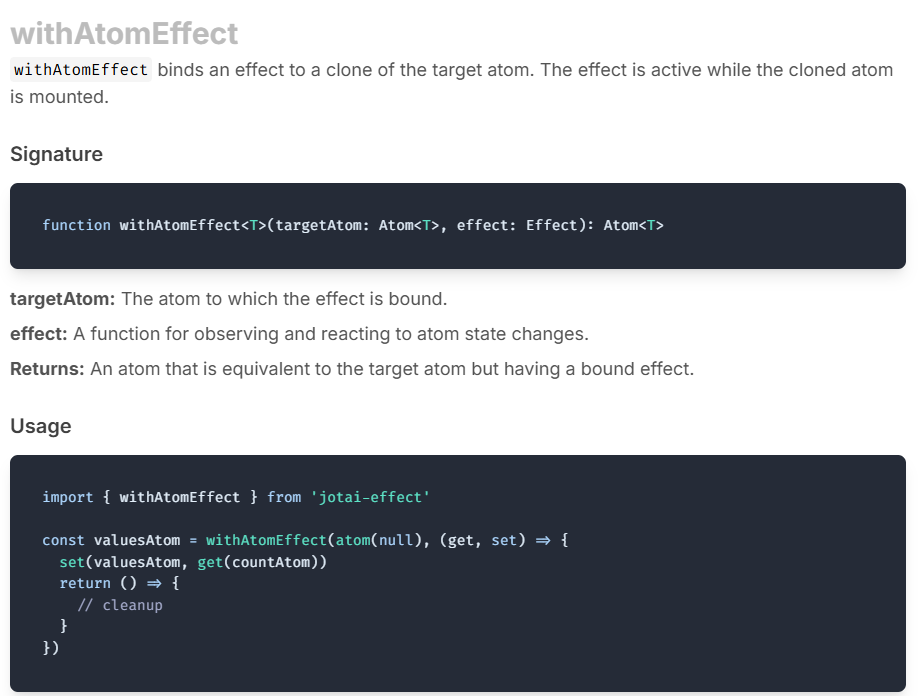
Observe传递store：



**atomEffect：（手动挂载）**



**withAtomEffect（可以创建原子的副作用）**



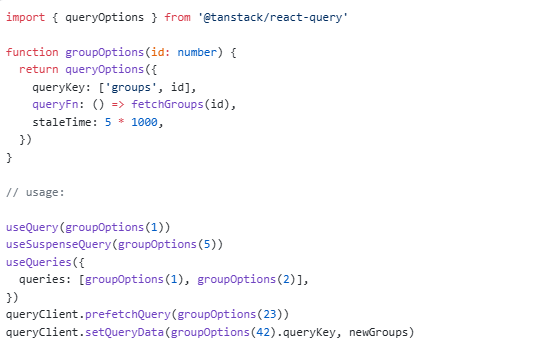
**React-Query主要分为两类，一个useQuery（查询），一个useMutation（增删改）**

**React-Query查询一般使用boolean（isPending，isError）类型值渲染，query有两个状态status和fetchStatus ， status一般常用状态，而fetchStatus 用于组合status深层次判断，例如后台获取**

**React-Query查询queryKey可以是任意可序列化值，string,object.数组都可以**

**Select相当于格式化数据返回**

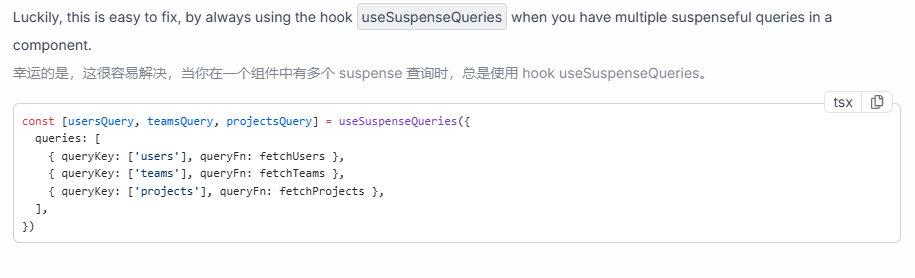
**queryOptions动态生产查询后台接口：**



**并行查询**



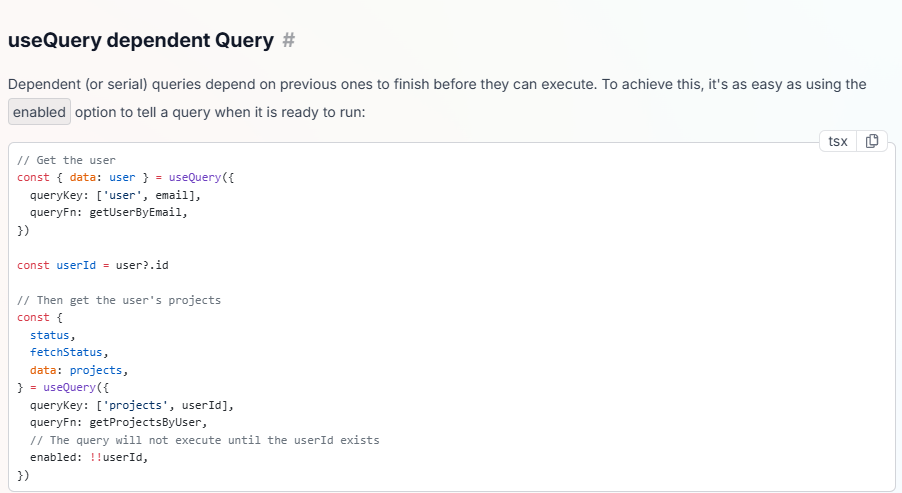
**注意：suspense 流式渲染会导致并行查询变为串行查询，要时刻注意suspense的影响，解决方法使用useSuspenseQueries**



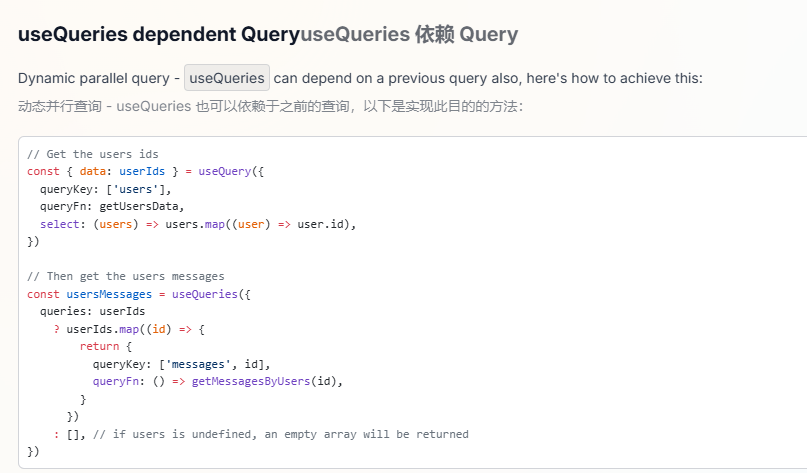
useQueries动态并行查询



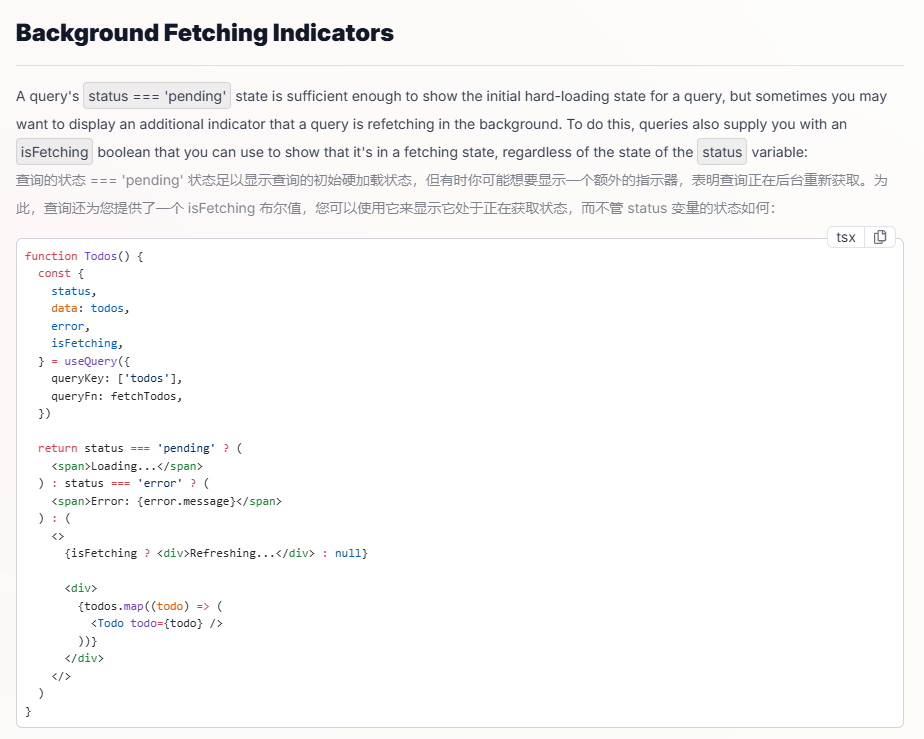
串行查询：字段enabled



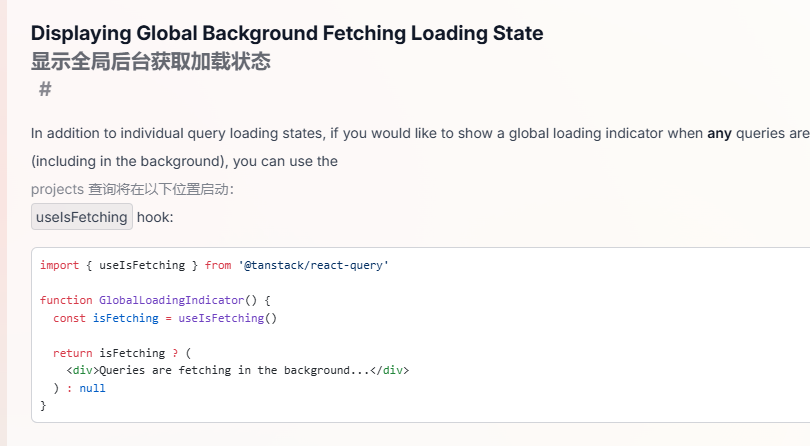
useQueries也可以依赖之前的接口组成串行查询，需用条件判断语句：



查询额外指示器isFetching，指示正在获取数据



全局fetching指示器：



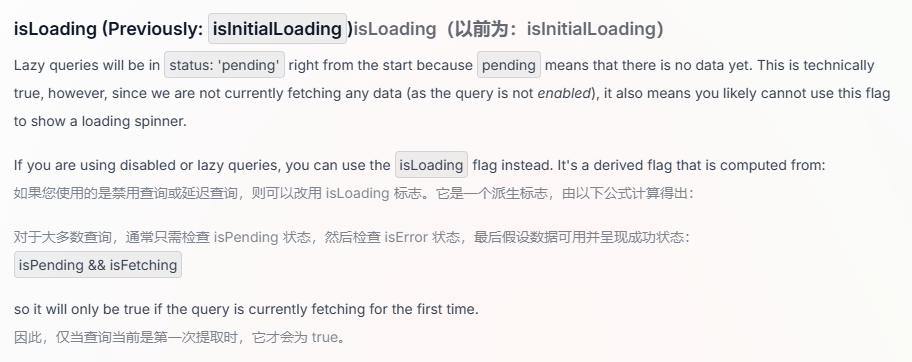
Refetch手动触发查询



延迟查询，条件语句判断



isLoading指示第一次成功查询

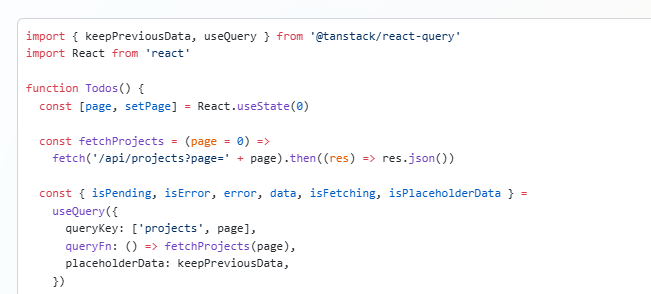


Skiptoken可以维持TS，结合上图layz-query看const [filter, setFilter] = React.useState('')

没有给与初始值，只给了类型，注意：但是不能再通过refetch手动触发query



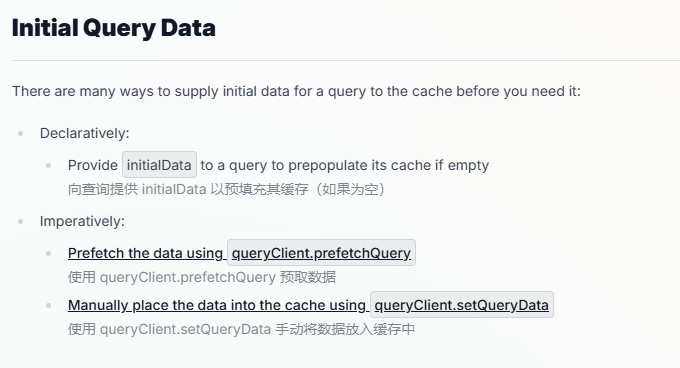
keepPreviousData分页查询占位



useInfiniteQuery滚动加载或点击more查询

初始化数据有两种方式，

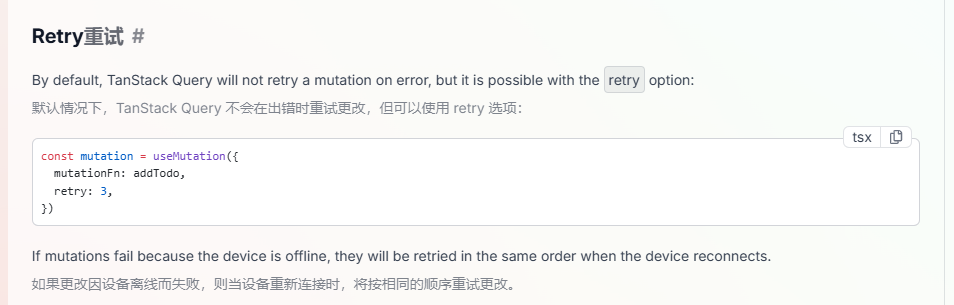
声明式：useQuery- initialData初始化数据和手动式



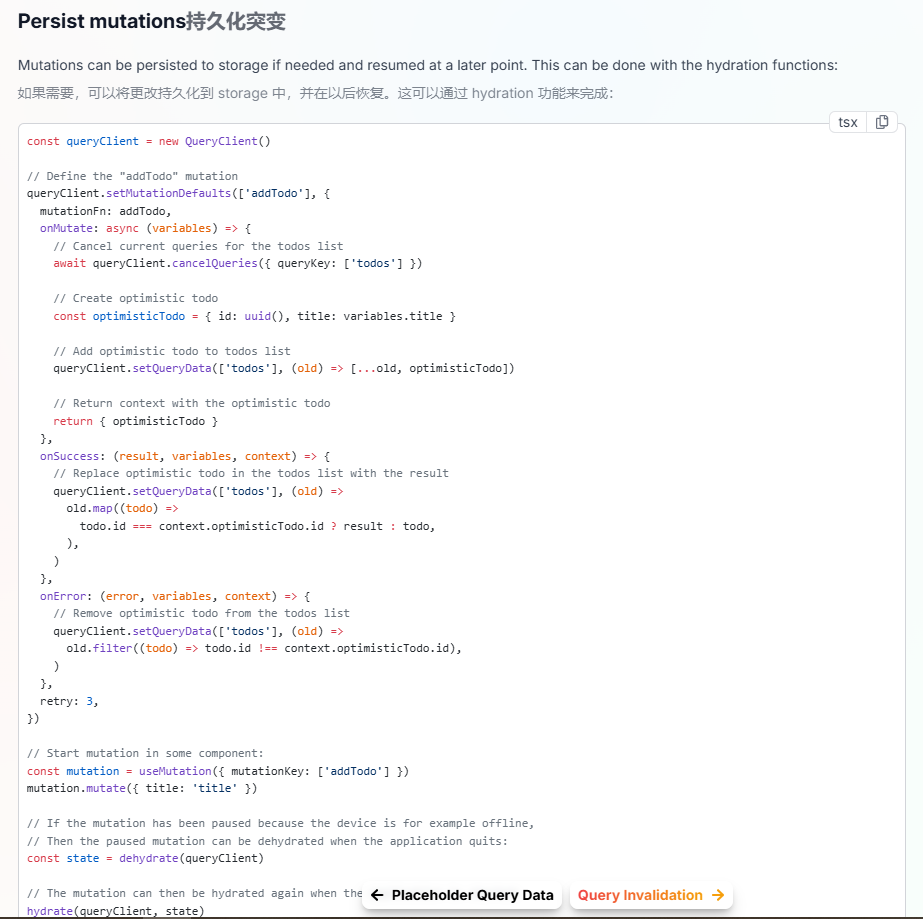
React-query有初始数据和占位符数据两种

useMutation用于增删改，手动触发

useMutation调用操作可以是mutate ，也可以是mutateAsync （返回promise）需要等待执行完毕



Mutation可以持久化（以便用户在网络重新连接后可以继续操作），而query不用



使用queryClient.invalidateQueries进行刷新是规范，而不使用refetch



乐观更新：

React Query 提供了两种方法，可以在更改完成之前乐观地更新 UI。你可以使用 onMutate 选项直接更新你的缓存:



，也可以利用返回的variables变量从 useMutation 结果更新你的 UI:

Variables包含表单提交的值

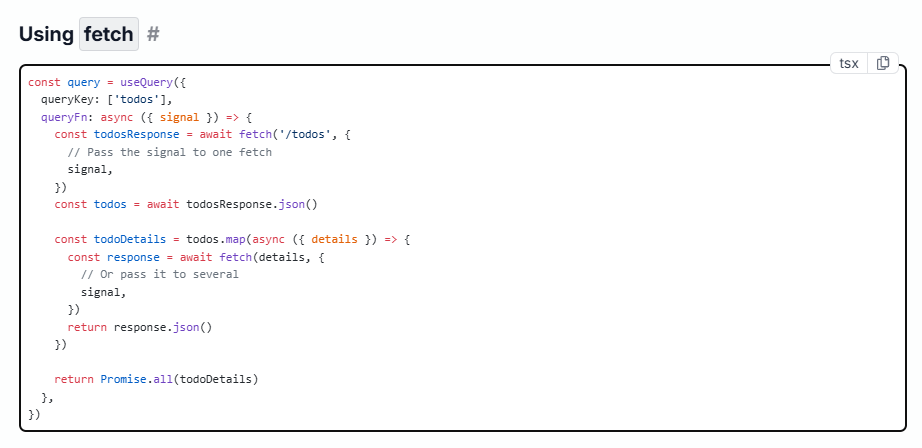
当需要实现乐观更新，而mutation和query又不在同一组件时，可以使用

useMutationState 钩子访问其他组件中的所有 mutation，这可以获取指定的mutations的变量值



**Query Cancellation查询取消分为自动取消和手动取消**

**自动取消返回siganl，将siganl传递给fetch会让fetch也取消：**



下面的例子是手动取消(包含自动取消，注意看注释)



预取数据的两个api



Redux有五个核心store（仓库）、state（数据），dispatch（唯一修改state数据的方式）、reducer（注册action的具体操作，用于返回state数据）、action（操作），还有一个切片Slice的概念

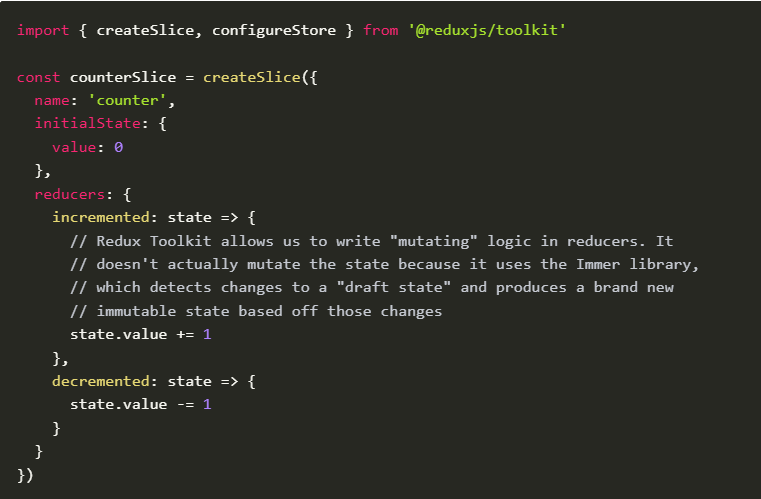
每一个切片就是一个state数据，切片创建各自的reducer并在store初始化时注册进去

Redux-Thunk用于支持reducer返回函数以支持异步操作

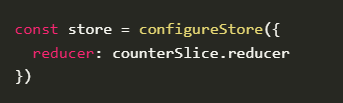
Redux Toolkit 的 createSlice(可以创建slice) 和 createAsyncThunk。(异步)

RTK Query（通过 createApi）来管理 API 请求和缓存

首先Slice切片创建一个基础对象，包含初始值，rdducer修改方法，name等，



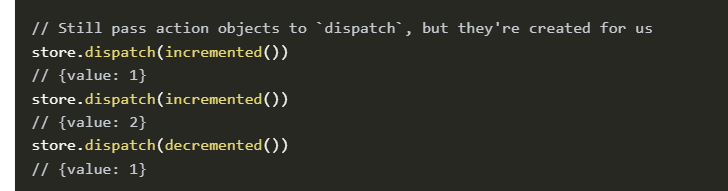
然后利用configureStore创建store对象，将切片的reduce方法注册到store



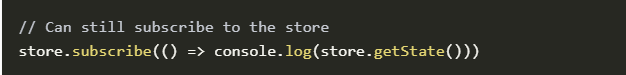
然后创建action给store的dispacth调用，action源于切片slice的renderc



Store dispacth调用action，dispacth也就是发布消息



最后还有一个概念，只有store可以订阅消息



**RxJS 是一个库，处理复杂的异步流和事件流，用于使用可观察序列编写异步和基于事件的程序。它提供了一个核心类型，即 Observable、附属类型（Observer、Schedulers、Subjects）和受 Array 方法启发的运算符（map、filter、reduce、every 等），以允许将异步事件作为集合处理。**

**Rxjs主要是包含几个概念**

**Observables（可观察对象），可以执行pipe链式调用，subscibe订阅**

**Observer（观察者） 定义Observable（可观察）的消费属性，一般包含next,error,complete，在Subscription（订阅）中使用**

**Operators（操作员）map，scan，filter等操作函数**

**Subscription（订阅）可使用subscription.add(childSubscription)绑定多个订阅，一键取消所有绑定的订阅unsubscribe**

**Subjects（主题），主要用于subscribe（订阅），可以通过方式**

1. **observable.subscribe(subject);对多个Observables（可观察）设置主题以实现多播功能，**
2. **简化使用方式有multicast，publish（这个最方便），然后使用connect() 来触发源 Observable 的执行，注意：方式1不需要手动触发**
3. **refCounted可以不在显式调用connect执行**
4. **变体BehaviorSubject（它存储发出的最新值）、ReplaySubject（可以获取指定数量的最新值）、AsyncSubject（只有在源 Observable 完成时才会向所有订阅者发出最后一个值）、Void subject（不指定具体类型）**

**Scheduler（调度程序）控制调用时机，例如通过observeOn(asyncScheduler)设置同步调用，注意放在最后**

**Of可以创建一个Observable**

**FromEvent可绑定dom创建一个Observable（可观察），**

**observable.subscibe订阅，包含next,error,complete（这些属于观察者Observer）**

**observable.pipe链式调用**

**（map，scan，filter）属于操作员Operators**

**throttleTime（1000）等待1000ms**

**scan累积累加等**

**map转换值**

**使用try catch包裹操作代码预防error发生：**



**Ethers.js：这是一个非常流行的 JavaScript 库，用于与以太坊和其他区块链网络交互。它提供了一系列的功能，包括连接钱包、发送交易、查询链上数据等。ether.js 主要用于在前端与区块链智能合约进行交互，操作相对简单、轻量，且对开发者友好。**

**核心概念：**

**Provider(MetaMask等钱包提供商)**

**window.ethereum == null判断是否有 钱包-浏览器插件 安装**

**Signer（签名账户，可以通过Provider钱包获取）**

**Signer可以对其他数据进行签名，例如signer.signMessage签名字符串**

**Transaction（交易）**

**signer.sendTransaction()发送交易**

**receipt = await tx.wait();等待交易完成返回收据**

**transfer.staticCall模拟调用，用于检查是否可以正常执行交易**

**Contract（调用智能合约）**

**contract = new Contract("dai.tokens.ethers.eth", abi, provider)；**

**contract.on设置监听事件**

**events = await contract.queryFilter(filter, -100)获取-100个区块信息**

**Receipt （收据，Transaction交易后获得的收据）**

**parseEther、parseUnits、formatEther、formatUnits等转换数值函数wei\Gwei**

**Wagmi（用hooks封装了viem.js，直接使用这个）：这是一个用于 React 应用程序的现代 Web3 库，能够简化与以太坊和其他链的交互。它的目标是通过一组 hooks 来处理区块链相关的操作，比如连接钱包、监听账户变化、发送交易等，使开发者能够更方便地在 React 中集成 Web3 功能。**

**Wagmi调用智能合约相关hooks时，传入的api必须以这两种形式传递，以便提供更好的ts：**



**Wagmi CLI 用于拉取智能合约abis并生成hooks方便使用**

**Appkit链接钱包（支持Sign In With Ethereum即SIWE/SIWX进行以太坊登录dapp）**

**Hooks：**

**UseAppkit（用于控制模态的钩子）**

**useAppKitAccount（用于访问帐户数据和连接状态的挂钩）**

**useAppKitNetwork（用于访问网络数据和方法的挂钩）**

**useAppKitState（获取模态状态的当前值的钩子）**

**useDisconnect（挂断会话连接）**

**useAppKitTheme（控制模态主题的钩子）,**

**useAppKitEvents（订阅模态事件的钩子）,**

**useWalletInfo（访问钱包信息的挂钩）**

**wagmi**

**Send Transaction自定义send-transaction.tsx发送交易组件**

**useSendTransaction发送交易，useWaitForTransactionReceipt等待交易收据**

**useReadContract （支持同时调用多个合约）读取合同不消耗汽油费**

[**useBlockNumber**](https://wagmi.sh/react/api/hooks/useBlockNumber)**监测区块更新以重新发布tansaction’交易**

**Write to Contract 自定义mint-nft.tsx，将数字资产写入区块链**

**useWriteContract写入合同，也需要等待交易收据**

**Viem：这是一个相对较新的 Web3 库，专注于与以太坊和其他链的智能合约交互。它是一个轻量级且快速的库，可以帮助开发者更高效地与链上数据进行交互。**

**Viem提供主要的Abi：ERC-20 是最基础的代币标准，ERC-721 是用于不可替代代币（NFT）的标准，ERC-4626 是用于简化去中心化金融（DeFi）协议的存款和借贷合约的标准**

**1. ERC-20（代币标准）**

**主要函数：**

* **balanceOf(address account)：查询某个地址的代币余额。**
* **transfer(address recipient, uint256 amount)：从当前账户向指定账户转账一定数量的代币。**
* **approve(address spender, uint256 amount)：批准某个地址可以花费指定数量的代币。**
* **transferFrom(address sender, address recipient, uint256 amount)：从已批准的地址转账代币。**
* **allowance(address owner, address spender)：查询已批准的某个地址可以支配的代币数量。**

**使用场景：**

* **查询用户余额，展示用户持有的代币。**
* **实现代币转账功能。**
* **支持用户授权其他地址使用他们的代币（例如在 DeFi 协议中进行借贷或交易）。**

**2. ERC-721（不可替代代币，NFT）**

**主要函数：**

* **ownerOf(uint256 tokenId)：查询某个特定 NFT 的拥有者地址。**
* **transferFrom(address from, address to, uint256 tokenId)：转移 NFT。**
* **approve(address to, uint256 tokenId)：批准某个地址管理某个特定 NFT。**
* **safeTransferFrom(address from, address to, uint256 tokenId)：安全地转移 NFT，确保接收方是一个合约且能够接收 NFT。**

**使用场景：**

* **展示用户拥有的 NFT（如头像、艺术品等）。**
* **支持用户买卖或转移 NFT。**
* **与 NFT 市场（如 OpenSea）交互，管理用户的 NFT。**

**3. ERC-4626（收益代币标准，DeFi）**

**主要函数：**

* **deposit(uint256 assets, address receiver)：将资产存入合约，通常用于 DeFi 协议中的存款。**
* **withdraw(uint256 assets, address receiver, address owner)：从合约中提取资产。**
* **getDepositTokensForShares(uint256 shares)：该函数用于查询一定数量的份额对应多少存款代币。例如，用户持有一定数量的份额，查询这些份额相当于多少存款**
* **getSharesForDeposit(uint256 assets)：该函数用于查询用户存入的资产数量对应多少份额。例如，用户存入 100 单位资产，合约返回这些资产对应的份额数量**

**使用场景：**

* **实现存款/提现功能，允许用户与 DeFi 协议交互（如借贷、流动性挖矿）。**
* **显示和管理用户的存款和收益份额。**

**DeFi 是如何赚钱的**

**DeFi（去中心化金融） 作为一种新兴的金融体系，提供了多种盈利模式，这些盈利模式通常不依赖于传统银行或金融机构，而是通过智能合约、去中心化协议、以及代币经济模型来实现收益。DeFi 的盈利方式主要来自以下几种模式：**

**1. 流动性提供（Liquidity Provision）**

**在 DeFi 中，流动性提供者（LPs）将他们的资产提供给去中心化交易所（DEX）或其他 DeFi 协议。作为回报，流动性提供者会获得 交易手续费 或 奖励。**

* **DEX（去中心化交易所）：**
  + **例如 Uniswap、SushiSwap 等去中心化交易所，通过自动做市商（AMM）算法匹配交易。**
  + **流动性提供者将两种或更多的代币（例如 ETH 和 USDT）存入交易池，以便其他用户进行交易。**
  + **作为回报，流动性提供者会获得交易费用的分成，通常是每笔交易费的 0.3%。**
* **收益：LPs 不仅可以从交易手续费中获得收益，还可以通过流动性池的 激励奖励（例如平台代币）获得额外的收益。**
* **风险：流动性提供者还面临 无常损失（Impermanent Loss） 的风险，即当流动性池中资产价格波动较大时，可能导致最终提取资产的价值低于最初存入时的价值。**

**2. 借贷（Lending & Borrowing）**

**在 DeFi 借贷协议中，用户可以通过 借贷平台 获得收益，既可以是 借款利息，也可以是 存款收益。**

* **存款收益：**
  + **用户将资产存入借贷协议（如 Aave、Compound 等）作为流动性，借贷平台会根据存款量和市场需求给出一个 利率。**
  + **存款者获得的收益来自借款者支付的利息。**
  + **举例来说，你将 USDT 存入 Aave 协议中，协议根据市场供需情况，为借款人提供 USDT，并按利率支付收益给存款者。**
* **借款利息：**
  + **如果你需要借贷加密货币，你需要向平台支付一定的 借款利息。这些利息的一部分会支付给提供流动性的存款者。**
* **风险：**
  + **主要风险来自 清算，即当借款人的抵押品不足以覆盖借款时，平台会进行清算，从而对流动性提供者的资金造成潜在损失。**

**3. 收益农场（Yield Farming）**

**收益农场是 DeFi 中的一种盈利模式，用户通过 流动性挖矿 或 治理代币挖矿 获得收益。**

* **流动性挖矿：**
  + **用户将代币存入某个 DeFi 协议的流动性池，并通过提供流动性获得平台代币或其他奖励。例如，在 SushiSwap 上，用户不仅获得交易费，还可以获得平台的原生代币 SUSHI。**
* **治理代币挖矿：**
  + **在一些 DeFi 协议中，用户提供流动性或参与借贷、质押等操作时，可以获得 治理代币。这些代币可以用于协议的治理投票，或者在其他市场中进行交易。**
* **收益：收益来自 流动性池中的交易手续费、平台奖励（原生代币） 和 治理代币。**
* **风险：**
  + **无常损失：流动性提供者面临价格波动带来的风险。**
  + **平台风险：DeFi 协议的安全性较为重要，智能合约漏洞或平台被攻击可能导致资金损失。**

**4. 质押（Staking）**

**质押是指用户将代币锁仓在某个协议中，以获得奖励。质押可以发生在 PoS（权益证明） 或 PoW（工作量证明） 网络中，或是 DeFi 协议中的 Staking Pools。**

* **加密货币质押：例如，在 Ethereum 2.0 中，用户将 ETH 质押到网络中，作为网络验证节点的一部分，获得奖励。**
* **DeFi 协议中的质押：用户将某种代币（例如 DAI、USDT）质押到协议中，获得原生代币奖励或交易手续费的分成。**
* **收益：质押代币通常可以获得 固定的年化收益率（例如 5% 到 20%），具体利率取决于质押的资产类型和平台规则。**
* **风险：**
  + **锁仓风险：质押的代币在一段时间内无法流动，且可能面临 减值风险。**
  + **协议风险：如果质押协议被攻击或出现漏洞，可能导致质押资产的损失。**

**5. 去中心化保险（Decentralized Insurance）**

**去中心化保险平台允许用户购买保险、参与提供保险服务、并从中获得收益。**

* **保险协议：例如 Nexus Mutual 这样的去中心化保险协议，允许用户购买智能合约、DeFi 协议等的保险。参与者可以通过提供保险资金获得 保费 回报。**
* **风险：保险协议的风险主要来自 平台的信用风险 和 极端市场波动，这些因素可能导致保险提供者面临亏损。**

**6. 治理代币的增值**

**很多 DeFi 协议会发行 治理代币，用户在参与平台操作时（如提供流动性、借贷等）会获得这些代币。**

* **治理代币增值：随着平台的增长，治理代币可能会增值，用户可以选择通过交易治理代币获利，或者通过参与治理获得决策权利。**
* **示例：**
  + **Uniswap（UNI）、Aave（AAVE） 等平台通过发放治理代币，奖励早期参与者或流动性提供者，治理代币的价格随着平台的使用和增长可能会上涨。**

**7. 去中心化衍生品交易**

**在 DeFi 中，去中心化衍生品交易平台（如 dYdX）允许用户进行杠杆交易、期权和期货交易。这些平台通常会收取 交易费用 或 利息，平台的治理代币和交易手续费也是 DeFi 协议的盈利来源之一。**

* **衍生品交易：例如，通过 dYdX 进行杠杆交易，平台会收取交易费，并为参与者提供高杠杆回报。**
* **风险：**
  + **市场波动性风险：衍生品交易具有较高的风险，价格波动较大可能会导致亏损。**

**总结：DeFi 的盈利方式**

1. **流动性提供：通过在 DEX 上提供流动性，赚取交易手续费和奖励。**
2. **借贷平台：通过存款赚取利息，或通过借款支付利息。**
3. **收益农场：通过流动性挖矿和治理代币挖矿获得收益。**
4. **质押：通过质押代币获得固定收益。**
5. **去中心化保险：通过提供保险资金获得保费回报。**
6. **治理代币增值：通过参与 DeFi 协议获得治理代币，获取增值收益。**
7. **去中心化衍生品交易：通过衍生品交易赚取手续费和利息。**

**每种盈利方式都有其优势和风险，投资者和用户应根据自己的风险承受能力和策略选择合适的方式。DeFi 的核心优势在于其去中心化、无中介、透明性和高收益潜力，但也要警惕智能合约漏洞、市场波动和平台风险等因素。**