第 4 章 - 路由(route)

以檔案為基礎的路由 (file-based route)

Next 是以檔案為基礎的路由系統，意即所有的頁面的觀看網址路徑，會依照在`pages`目錄中的檔案與資料夾結構來決定，`pages`目錄所在位置即網站網址根路徑。

例如在`pages`目錄下有個`about.js`的網址路徑即是`/about`。

Next 中是透過`useRouter`這個勾子(hooks)，在每個頁面元件裡(或在裡面的各使用子元件)，獲取本頁路由資訊的方式。

例如要獲取目前的網址路徑的範例如下:

```
import { useRouter } from 'next/router'

export default function About() {
  // asPath: 呈現在瀏覽器包含搜尋參數的路徑字串。不包含`basePath`與`locale`
  const { asPath } = useRouter()
  // 如果瀏覽 http://localhost:3000/about 結果會是 `Path: /about`
  return <div>Path: {asPath}</div>
}
```

> 註: 注意勾子(hooks)只能在函式型元件中使用，另有一個withRouter方法是設計給類別型元件用的。

以索引檔為基礎的路由(index-based route)

在資料夾中的`index.js`檔案會視為路由到這個檔案的預設(基礎)路由。

例如以下的目錄，在`/pages/products/index.js`的網址路徑即是`/products`:

```
pages
 └── products
      └── index.js // path: /products
```

巢狀路由 比較 索引檔路由

當同時使用一般的靜態的"巢狀路由"與"索引檔路由"時，會產生衝突，最終會以檔案與資料夾組成的巢狀路由為主，而非索引檔路由。

例如以下的目錄結構:

```
pages
 ├── products.js
 └── products
      └── index.js
```

Next 在執行時會發出以下的警告訊息:

```
warn Duplicate page detected. pages\products.js and
pages\products\index.js resolve to /products
```

最後瀏覽/products網址是，看到的結果會是pages/products.js這個頁面元件的內容，而不是另一個。當然這是不建議的用法，路由的對應需要避免衝突或造成混淆不清。

動態路由(dynamic routing)

> 註: 靜態路由(有檔案名稱的頁面元件)，Next 中執行優先權高於動態路由(以方括號與屬性變數命名的檔案或資料夾)。

使用方括號(brackets，[])可以用於定義動態路由。動態的部份是一種變動的屬性定義，在經過路由後會傳遞到 roter 物件中的 query 參數屬性中。當然為了使用上的方便，最好也按一般的 JS 中的識別名稱規則命名，例如[pid],[product-id], [user-name]等等。

```
pages
 └── users
      └── [name].js
```

```
import { useRouter } from 'next/router'

export default function Name() {
  const { query, asPath } = useRouter()

  // 當瀏覽 /user/eddy 時，query 會得到 `{ name: 'eddy' }`
  console.log(query)

  return (
    <>
      <div>Name: {query.name}</div>
      <div>Path: {asPath}</div>
    </>
  )
}
```

當導覽到users/eddy時，會呈現以下的文字回應在網頁上:

```
Name: eddy
Path: /users/eddy
```

上面範例僅只能一個階層的路徑，如果再往下使用/的路徑，例如users/eddy/abc，則會回應 404 (找不到頁面)錯誤。

多階層動態路由(multiple dynamic routing)

```
[category]
 └──[sub-category]
      └──[id].js
```

多層次路由可以加上帶有方括號([])名稱的"資料夾"來達成。

使用useRouter勾子可以在當導覽到[id].js頁面時，從query物件中得到三個對應的參數值:

```
import { useRouter } from 'next/router'

export default function IdPage() {
  const { query, asPath } = useRouter()

  // { category: 'starbucks', 'sub-category': 'coffee', id: 'abc123' }
  console.log(query)

  return (
    <>
      <p>path: {asPath}</p>
      <p>category: {query.category}</p>
      <p>sub-category: {query['sub-category']}</p>
      <p>id:{query.id}</p>
    </>
  )
}
```

> 註: 這裡的[sub-category]的雖然使用的是 kebab-case 的命名法，但可以看到要存取值要改用類似陣列的語法。

但useRouter是只能在客戶端使用的(CSR)功能，意指應用這種方式，所有的動態參數將由客戶端的 React 來控制對應的網址參數該呈現什麼內容，而無法預先產生頁面(SSG)，或以伺服器端來控制呈現內容(SSR)。

伺服器在進行打包編輯時，並不知道所謂的網址上的動態參數是什麼，或該預先產生什麼頁面。Next 中的 SSG 技術，需要使用getStaticPaths與getStaticProps兩個特殊的、只能在頁面元件中使用的方法限制，或預先告知 Next 有哪些頁面是需要打包或預先產生的，但這兩個方法只會在伺服器端打包時使用，它們讀取不到useRouter回傳資料的(userRouter 回傳的資料是動態的)。

要使用 SSG 功能，需要先建立一整個對照於目前動態參數結構的 paths 陣列，以上面的 useRouter 中 query 物件，所需的 paths 陣列中的每個參數值應會是像下面這樣:

```
{ params: { category: 'starbucks', 'sub-category': 'coffee', id: 'abc123'
} }
```

所以paths的結構需要像下面這樣，這定義了三個"合法的"路由，意即只有這三種路由是可以讓使用者使用的，其它都不合法:

```
const paths = [
  { params: { category: 'starbucks', 'sub-category': 'coffee', id:
'abc123' } },
  { params: { category: 'starbucks', 'sub-category': 'cup', id: 'abc001' }
},
  { params: { category: 'louisa', 'sub-category': 'coffee', id: 'b002' }
},
  //...
]
```

加入getStaticPaths後，先以上面的範例來產生對應的 paths 對照陣列。當然使用了getStaticPaths也得要加上getStaticProps，它是用於瀏覽到某個特定合法的網址後，得到 params 物件後，這個一進入後的頁面元件該得到哪些內容:

```
export async function getStaticProps({ params }) {
  // 範例內容（這裡視情況fetch伺服器要求資料）
  const products = [
    { id: 'abc123', content: 'starbucks coffee' },
    { id: 'a001', content: 'starbucks cup' },
    { id: 'b002', content: 'louisa coffee' },
  ]

  // 這裡可以得到目前的網址參數(category, sub-category, id)
  console.log(params)

  // 範例: 由id得到對應的商品內容，傳遞給 IdPage 元件
  const product = products.find((v) => v.id === params.id)

  return {
    props: {
      product,
    },
  }
}

export async function getStaticPaths() {
  const paths = [
    {
      params: { category: 'starbucks', 'sub-category': 'coffee', id:
'abc123' },
    },
    { params: { category: 'starbucks', 'sub-category': 'cup', id: 'a001' }
},
    { params: { category: 'louisa', 'sub-category': 'coffee', id: 'b002' }
},
    //...
```

```
    ]
    return {
      paths,
      fallback: false,
    }
  }
```

IdPage元件的內容改為以下，product屬性是由getStaticProps傳給它的:

```
export default function IdPage({ product }) {
  const { query, asPath } = useRouter()

  // { category: 'starbucks', 'sub-category': 'coffee', id: 'abc123' }
  console.log(query)

  return (
    <>
      <p>path: {asPath}</p>
      <p>category: {query.category}</p>
      <p>sub-category: {query['sub-category']}</p>
      <p>id:{query.id}</p>
      <p>Product Content: {product.content}</p>
    </>
  )
}
```

當使用了上述的機制後，除了在getStaticPaths回傳的paths陣列中的網址動態參數外，導覽到不在這個對照陣列中的路徑均會回傳 404 頁面(頁面不存在)。

這種作法才能讓 Next 在打包期間進行預先產生頁面(SSG)，也就是說要事先準備好該使用哪些動態參數值，以及它的對應內容是什麼才行。

> 註: SSG 實作會屬於進階實作，流程會較為複雜，而且需要進行打包測試。所以可以先由純 CSR 的 useRouter 應用來開發。

獲得所有分段動態路由(Catch-all Segments Routes)

如果有過多層級的動態路由(三層以上)，或是連有多少分段(segment)都是不確定或動態的情況，會變得更複雜處理。

Next 中提供了另一種動態路由，稱為"獲得所有分段路由(Catch-all Segments Routes)"，意指這種路由可以獲得所有的分段(segment)或網址參數。

> 註: 網址分段(URL segments)指的是網址的一部份或是由斜線(/)分隔的路徑(或稱為網址參數)。如果有個路徑是/path/to/page/，則"path"、"to"與 "page"每一個都算是一個網址分段(URL segment)

比較於下列兩種動態路由檔案的結構，可以很清楚理解這種動態路由，不限於多少層級，可以獲得所有層級的分段(網址參數)。

第 1 種: 一般動態路由

只能匹配例如/user/login, /user/profile等等一個層級的動態路由。

```
user
 └──[slug].js
```

第 2 種: 獲得所有分段動態路由(Catch-all Segments Routes)

可以匹配任何在/user下的路由，例如/user, /user/profile, /user/profile/edit...等等，不限於層級或多少分段。

```
user
 └──[[...slug]].js
```

> 註: [[...slug]].js通常以slug命名，雙層方括號([[...]])代表它是可選的，以上例來說/user也是合法可用網址，只有單層方括號([...])時不能使用/user，必需要有下一層的分段(參數)例如/user/profile這才是合法可用網址。

當然，表面上看起來這種路由相當具有開發時的彈性，不過它在實作時又較前述幾種更複雜多，尤其是要搭配 SSG 時，仍需考量到要提供paths的路徑對照表時，過多或混在一起的分段會造成難以維護。

另外，將不同功能特性或版面、相差過大的頁面，全部都塞到同一頁面元件中實作，除非真的有必要，這也不是理想的實作方式，這一點是要先注意的。

首先，CSR 中一樣使用useRouter中的query屬性，可以得到所有網址上的參數(分段)，這與上一節的技巧一致:

```
import { useRouter } from 'next/router'

export default function SlugPage() {
  const { query, asPath } = useRouter()

  // { slug: [ 'admin', 'profile', 'edit' ] }
  console.log(query)

  return (
    <>
      <p>path: {asPath}</p>
    </>
  )
}
```

可以見到query物件回傳了以下的物件，代表目前所有網址上的分段，slug是對應你取名的檔案名稱[[...slug]].js，所有分段(參數)是一個陣列值:

```
{
  slug: ['admin', 'profile', 'edit']
}
```

因為一樣`useRouter`是只能設計給 CSR 用，要轉變為 SSG 產生頁面時，一樣要加上`getStaticPaths`與`getStaticProps`。

`getStaticPaths`的回傳要與這上面的 query 物件一致，而且是一個對照的陣列組合，例如以下的可用對照陣列，注意還需要加上`params`這個鍵(key)才行。以下的定義會限制出可用的網址所有組合，它是為了要預先產生靜態網頁使用的:

```
const paths = [
  { params: { slug: [] } }, // 對應`/user`網址，沒有slug的情況
  { params: { slug: ['login'] } },
  { params: { slug: ['register'] } },
  { params: { slug: ['admin', 'order'] } },
  { params: { slug: ['admin', 'profile', 'edit'] } },
]
```

以下是一個簡單的範例，真正實作上會比這複雜得多，而且要需要視情況決定，這種特殊的動態路由通常是在某些特定的第三方函式庫中會被使用:

```
// pages/user/[[...slug]].js
export default function SlugPage({ slug }) {
  // slug沒值會是null
  console.log(slug)

  const adminPage = (
    <>
      <h1>User Admin</h1>
      <p>this is Amin page</p>
    </>
  )

  const UserPage = (
    <>
      <h1>User Page</h1>
      <p>this is Amin page</p>
    </>
  )

  return <>{slug && slug[0] === 'admin' ? adminPage : UserPage}</>
}

export async function getStaticProps({ params }) {
  // 範例內容（這裡視情況fetch伺服器要求資料）

  // 這裡可以得到目前的網址參數(category, sub-category, id)
```

```
  console.log(params)

  return {
    props: {
      slug: params.slug || null, // 沒有slug要改傳null
    },
  }
}

export async function getStaticPaths() {
  // 範例內容（這裡視情況fetch伺服器要求資料）
  const paths = [
    { params: { slug: [] } }, // 對應`/user`網址，沒有slug的情況
    { params: { slug: ['login'] } },
    { params: { slug: ['register'] } },
    { params: { slug: ['admin', 'order'] } },
    { params: { slug: ['admin', 'profile', 'edit'] } },
  ]

  return {
    paths,
    fallback: false,
  }
}
```

總結 動態路由簡單速查表

- `/page/[page-id].js` 可以匹配像 `/page/1` 或 `/page/2`，但不匹配 `/page/1/2`
- `/page/[...slug].js` 可以匹配像 `/page/1/2`，但不匹配 `/page/`
- `/page/[[...slug]].js` 可以匹配像 `/page/1/2` 或 `/page/`

應用建議

- 獲得所有分段動態路由(Catch-all Segments Routes)相對來說實作比較複雜，除非有必要再使用它
- 多階層動態路由(multiple dynamic routing)用二層或三層，或單個動態路由(dynamic routing)實作上比較容易，可以先從`useRouter`的實作開始，
- 如果要應用 Next 中完整的 SSG 功能，需要了解如何讓 Next 產生對應的靜態頁面

查詢字串 (Query String)

查詢字串指的是在一特定網址後加上問號(?)，之後是以和號(&)串連的各指定值，例如`/product/book?id=1&year=2022`

查詢字串一樣可以直接用 `useRouter` 中的 `query` 參數可以得到，每個查詢的查詢參數(或搜尋參數)會轉化為物件的其一個屬性值，但因網址為字串資料類型，所以一定是字串值:

```
// pages/product/[name].js
import { useRouter } from 'next/router'
```

```
export default function ProductPage() {
  const router = useRouter()
  // `/product/book?id=1&year=2022`
  //  query = { name :'book', id:'1', year: '2022'}
  const query = router.query

  return (
    <div>
      <pre>{JSON.stringify(query)}</pre>
    </div>
  )
}
```

查詢字串並無法直接使用 Next 中的 SSG 功能達成預先產生頁面，會被認為是動態的參數。只能透過 SSR 或 CSR 功能來應用。

問題: 用於 CSR 的 useRouter 勾子，在動態網址時會進行渲染兩次

只有單使用 useRouter 時，沒有使用 SSG 功能時會發生。

例如`asPath`值會得到兩次，第一次是未改變的原路由`product/[name]`，第二次才是真正路由`/product/book`

這情況是 Next 中預期的正常行為，以下說明原文來自Next 官網，或參考Next.js issue#12010 答覆:

> 當在進行預先渲染時，router的query物件會是空白的，因為在這階段時我們沒有可以提供query的資訊。要在水合作用(hydration)之後，Next.js將會對你的應用觸發一個更新，由此才能在query物件中提供路由參數。
>
> 以下幾種情況，將會導致在水合作用(hydration)之後，觸發另一次的渲染，query要進行更新:
>
> - 頁面是動態路由(dynamic-route)的
> - 頁面在網址上有查詢值
> - `next.config.js`有設定重寫(Rewrites)
>
> 如果要為了區別出query是否已經完成更新，準備好可以被使用，你可以利用`next/router`中的`isReady`屬性

> 註: 水合作用(hydration)是 SSR 專用術語，指的是使用類似於 React 原本的瀏覽器 DOM render 技術，但在伺服器進行，具取而代之的是，會先讓使用者載入一個已具有 HTML DOM 元素網頁(類似框架)，再載入對應的 JS 碼，之後進行水合化(hyrate)，進行檢查已載入的 DOM 的結構是否相符，並將相對應的事件加上，使之成為 CSR 元件，具有事件監聽和互動性。 更多參考: 官方 React 18 SSR 架構說明(有圖解) / hydration SSR 圖解說明

結論，除非利用 Next 中的 `getStaticProps` 或`getServerSideProps` (或舊的 `getInitialProps`)不然會因 Next 中進行自動靜態最佳化(automatic static optimization)後，觸發重新渲染是必然的。

CSR 中可以像下面這樣以`isReady`來判斷是否已經準備好可以使用 `query`，以下範例來自這裡問答 :

```jsx
// pages/user/[id].jsx
import { useEffect } from 'react'
import { useRouter } from 'next/router'

export default function UserPage() {
  const { query, isReady } = useRouter()

  useEffect(() => {
    if (isReady) {
      // fetch(`your.domain/user/${query.id}`)...;
    }
  }, [isReady])

  if (!isReady) {
    return null
  }

  return query.id
}
```