选择器(selector)是用于获取存储状态片的纯函数。@ngrx/store提供了一些帮助函数来优化这个选择器。

选择器提供许多功能时，选择切片的状态:

在使用createSelector和createFeatureSelector函数时，@ngrx/store将跟踪调用选择器函数的最新参数。因为选择器是纯函数，所以在参数匹配时可以返回最后的结果，而无需重新调用选择器函数。这可以提供性能优势，特别是对于执行昂贵计算的选择器。这种做法称为记忆。

**\*\* 使用一个状态的选择器**

import { createSelector } from '@ngrx/store';

export interface FeatureState { counter: number;}

export interface AppState { feature: FeatureState;}

export const selectFeature = (state: AppState) => state.feature;

export const selectFeatureCount = createSelector(

selectFeature,

(state: FeatureState) => state.counter

);

**\*\* 对多个状态使用选择器**

createSelector可用于基于相同状态的多个片从状态中选择一些数据。

createSelector函数可以使用多达8个选择器函数来进行更完整的状态选择。

例如，假设状态中有一个selectedUser对象。您还拥有图书对象的allBooks数组。

你想要显示当前用户的所有书籍。

您可以使用createSelector来实现这一点。你的可见书籍将永远是最新的，

即使你更新他们在所有的书。如果选中了一本，它们将始终显示属于用户的图书，

如果没有选中用户，它们将显示所有图书。

结果将只是你的state的一部分，由state的另一部分过滤。它将永远是最新的。

import { createSelector } from '@ngrx/store';

export interface User { id: number; name: string; }

export interface Book { id: number; userId: number; name: string;}

export interface AppState { selectedUser: User; allBooks: Book[];}

export const selectUser = (state: AppState) => state.selectedUser;

export const selectAllBooks = (state: AppState) => state.allBooks;

export const selectVisibleBooks = createSelector(

selectUser,

selectAllBooks,

(selectedUser: User, allBooks: Book[]) => {

if (selectedUser && allBooks) {

return allBooks.filter((book: Book) => book.userId === selectedUser.id);

} else {

return allBooks;

} });

**\*\*\* Using selectors with props**

要根据store中不可用的数据选择状态块，可以将props传递给选择器函数。

这些props通过每个选择器和投影仪函数。为此，我们必须在组件内使用选择器时指定这些props。

例如，如果我们有一个计数器，我们想要乘以它的值，我们可以添加乘数作为prop:

选择器或投影仪的最后一个参数是props参数，在我们的例子中它看起来如下

export const getCount = createSelector(

getCounterValue,

(counter, *props*) => counter \* *props.multiply*

);

在组件内部，我们可以定义prop:

ngOnInit() {

this.counter = this.store.pipe(select(fromRoot.getCount, *{ multiply: 2 }*))

}

请记住，选择器只在其缓存中保存以前的输入参数。

如果您将这个选择器与另一个乘法因子重用，选择器将总是不得不重新计算它的值。

这是因为它接收了两个乘数(例如，一个是2，另一个是4).为了正确记忆选择器，

包装选择器在一个工厂函数，以创建不同的实例的选择器。

下面是使用按id区分的多个计数器的示例。

export const getCount = () =>

createSelector(

(state, props) => state.counter[props.id],

(counter, props) => counter \* props.multiply

);

组件的选择器现在调用工厂函数来创建不同的选择器实例:

ngOnInit() {

this.counter2 = this.store.pipe(

select(fromRoot.getCount(), { id: 'counter2', multiply: 2 }));

this.counter4 = this.store.pipe(

select(fromRoot.getCount(), { id: 'counter4', multiply: 4 }));

this.counter6 = this.store.pipe(

select(fromRoot.getCount(), { id: 'counter6', multiply: 6 }));

}

**\*\*\* Selecting Feature States**

createFeatureSelector: 是返回顶级特性状态的方便方法。它为状态的特性片返回一个类型选择器函数。

export const featureKey = 'feature';

export interface FeatureState { counter: number;}

export interface AppState { feature: FeatureState;}

export const selectFeature =

createFeatureSelector<AppState, FeatureState>(featureKey);

export const selectFeatureCount = createSelector(

selectFeature,

(state: FeatureState) => state.counter

);

下面的选择器无法编译，因为fooFeatureKey ('foo')不是AppState的一个功能片。

export const selectFeature =

createFeatureSelector<AppState, FeatureState>(fooFeatureKey);

**\*\* Resetting Memoized Selectors**

通过调用createSelector或createFeatureSelector返回的selector函数最初具有一个空的记忆值。在第一次调用选择器时，将其记忆值存储在内存中。如果随后使用相同的参数调用选择器，它将返回记忆值。如果使用不同的参数调用选择器，它将重新计算和更新其记忆值。考虑以下:

export interface State { counter1: number; counter2: number;}

export const selectCounter1 = (state: State) => state.counter1;

export const selectCounter2 = (state: State) => state.counter2;

export const selectTotal = createSelector(

selectCounter1,

selectCounter2,

(counter1, counter2) => counter1 + counter2

); // selectTotal的记忆值为null，因为它还没有被调用。

let state = { counter1: 3, counter2: 4 };

selectTotal(state); // 计算3和4的和，返回7。selectTotal现在的记忆值是7

selectTotal(state); // 不计算3和4的和。相反，selectTotal返回记忆值7

state = { ...state, counter2: 5 };

selectTotal(state); // 计算3和5的和，返回8。selectTotal现在的记忆值是8

选择器的记忆值无限期地留在内存中。例如，如果缓存值是不再需要的大型数据集，

则可以将缓存值重置为null，以便从内存中删除大型数据集。这可以通过调用选择器上的release方法来完成。

selectTotal(state); // 返回记忆值8

selectTotal.release(); // ：selectTotal的记忆值现在为null

释放一个选择器也递归释放任何祖先选择器。考虑以下:

export interface State { evenNums: number[]; oddNums: number[];}

export const selectSumEvenNums = createSelector(

(state: State) => state.evenNums,

evenNums => evenNums.reduce((prev, curr) => prev + curr)

);

export const selectSumOddNums = createSelector(

(state: State) => state.oddNums,

oddNums => oddNums.reduce((prev, curr) => prev + curr)

);

export const selectTotal = createSelector(

selectSumEvenNums,

selectSumOddNums,

(evenSum, oddSum) => evenSum + oddSum

);

selectTotal({ evenNums: [2, 4], oddNums: [1, 3],});

/\*\*

\* 调用selectTotal.release()之前的记忆值

\* selectSumEvenNums 6

\* selectSumOddNums 4

\* selectTotal 10

\*/

selectTotal.release();

/\*\*

\* 调用selectTotal.release()后的记忆值

\* selectSumEvenNums null

\* selectSumOddNums null

\* selectTotal null

\*/

**\*\*\* 使用没有类型泛型的存储 Using Store Without Type Generic**

当将存储注入组件和其他注入表时，可以忽略泛型类型。 如果注入时没有泛型，则应用默认的泛型，如下所示:Store<T = object>。

从存储中选择信息的最常见方法是使用由createSelector定义的选择器函数。

在这样做时，TypeScript能够从选择器函数自动推断类型，因此减少了在存储泛型中定义类型的需要。

如果您正在使用选择器的字符串版本，因为不能在这些实例中自动推断类型，那么继续提供存储类型泛型是很重要的。

下面的示例演示了不提供通用型的Store的使用:

export class AppComponent {

counter$ = this.store.select(fromCounter.selectCounter);

constructor(*private readonly store: Store*) {}

}

**\*\*\* 高级用法**

选择器允许您为应用程序状态编写一个读取模型。就CQRS架构模式而言，NgRx将读模型(selectors)与写模型(reducers)分离开来。

一种高级技术是将选择器与RxJS可伸缩操作符结合起来。

本节将介绍比较选择器和可伸缩操作符的一些基础知识，并演示如何使用createSelector和scan来显示状态转换的历史。

import { map, filter } from 'rxjs/operators';

store

.pipe(

map(state => selectValues(state)),

filter(val => val !== undefined)

)

.subscribe(/\* .. \*/);

可以使用NgRx中的select()实用函数对上面的内容进行进一步重写:

import { select } from '@ngrx/store';

import { map, filter } from 'rxjs/operators';

store

.pipe(

select(selectValues),

filter(val => val !== undefined)

)

.subscribe(/\* .. \*/);

解决方案:提取一个可伸缩操作符 (Solution: Extracting a pipeable operator )

为了使select()和filter()行为成为可重用的代码，我们使用RxJS pipe()实用函数提取了一个管道操作符:

export const selectFilteredValues = pipe(

select(selectValues),

filter(val => val !== undefined)

);

store.pipe(selectFilteredValues).subscribe(/\* .. \*/);

\*\*\* 高级示例:选择最近的{n}状态转换

让我们在一个高级示例中研究结合NgRx选择器和RxJS操作符的技术。

在本例中，我们将编写一个选择器函数，它将从应用程序状态的两个不同片段中投影值。

当两个状态片都有一个值时，投射状态将产生一个值。否则，选择器将发出一个未定义的值。

export const selectProjectedValues = createSelector(

selectFoo,

selectBar,

(foo, bar) => {

if (foo && bar) {

return { foo, bar };

}

return undefined;

}

);

然后，组件应该可视化状态转换的历史。我们不仅对当前状态感兴趣，而且希望显示最后的n个状态。

这意味着我们将把一个状态值流(1,2,3)映射到一个状态值数组([1,2,3])。

// The number of state transitions is given by the user (subscriber)

状态转换的数量由用户(订阅者)给出

export const selectLastStateTransitions = (count: number) => {

return pipe(

// Thanks to `createSelector` the operator will have memoization "for free"

select(selectProjectedValues),

// Combines the last `count` state values in array

scan((acc, curr) => {

return [ curr, ...acc ].filter((val, index) => index < count && val !== undefined)

}, [] as {foo: number; bar: string}[]) // XX: Explicit type hint for the array.

// Equivalent to what is emitted by the selector

);

}

最后，组件将订阅存储，告诉它希望显示的状态转换的数量:

// Subscribe to the store using the custom pipeable operator

使用自定义可移植操作符订阅存储

store.pipe(selectLastStateTransitions(3)).subscribe(/\* .. \*/);