**Использование Redux с TypeScript**

**Стандартная настройка проекта Redux Toolkit с использованием** [**TypeScript**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#standard-redux-toolkit-project-setup-with-typescript)

Мы предполагаем, что типичный проект Redux использует вместе Redux Toolkit и React Redux.

[**Redux Toolkit**](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://redux-toolkit.js.org) **(RTK)** — это стандартный подход для написания современной логики Redux. RTK уже написан на TypeScript, и его API разработан так, чтобы обеспечить удобство использования TypeScript.

Определения типов React Redux находятся в отдельном [***@types/react-reduxпакете typedefs***](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://npm.im/@types/react-redux) в NPM. Помимо ввода функций библиотеки, типы также экспортируют некоторые помощники, чтобы упростить написание типобезопасных интерфейсов между вашим хранилищем Redux и вашими компонентами React.

Начиная с React Redux v7.2.3, react-reduxпакет зависит от ***@types/react-redux***, поэтому определения типов будут автоматически установлены вместе с библиотекой. В противном случае вам придется установить их вручную (обычно ***npm install @types/react-redux***).

Шаблон [Redux+TS для **Create-React-App**](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://github.com/reduxjs/cra-template-redux-typescript) содержит рабочий пример уже настроенных шаблонов.

**Определение корневого состояния и** [**типы**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#define-root-state-and-dispatch-types)

Использование [**configureStore**](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://redux-toolkit.js.org/api/configureStore) не требует дополнительных вводов. Однако вам потребуется извлечь **RootState** тип и **Dispatch** тип, чтобы на них можно было ссылаться при необходимости. Вывод этих типов из самого хранилища означает, что они правильно обновляются при добавлении дополнительных фрагментов состояния или изменении настроек промежуточного программного обеспечения.

Поскольку это типы, их можно безопасно экспортировать непосредственно из файла настройки вашего магазина, например, app/store.tsи импортировать непосредственно в другие файлы.

приложение/store.ts

import { configureStore } from '@reduxjs/toolkit'  
// ...  
  
export const store = configureStore({  
 reducer: {  
 posts: postsReducer,  
 comments: commentsReducer,  
 users: usersReducer  
 }  
})  
  
// Infer the `RootState` and `AppDispatch` types from the store itself  
export type RootState = ReturnType<typeof store.getState>  
// Inferred type: {posts: PostsState, comments: CommentsState, users: UsersState}  
export type AppDispatch = typeof store.dispatch

**Определение типизированных** [**хуков**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#define-typed-hooks)

Хотя можно импортировать типы **RootState** и **AppDispatch** в каждый компонент, лучше создать предварительно типизированные версии перехватчиков **useDispatch** и **useSelector** для использования в вашем приложении. Это важно по нескольким причинам:

* Ибо useSelector это избавляет вас от необходимости (state: RootState) каждый раз вводить текст
* Для типа useDispatch по умолчанию Dispatch не известно о переходниках или другом промежуточном программном обеспечении. Чтобы правильно отправлять преобразователи, вам необходимо использовать определенный настраиваемый AppDispatch тип из хранилища, который включает в себя типы промежуточного программного обеспечения преобразователей, и использовать его с useDispatch. Добавление предварительно введенного useDispatch хука не позволит вам забыть импортировать AppDispatch туда, где это необходимо.

Поскольку это реальные переменные, а не типы, важно определить их в отдельном файле, например app/hooks.ts, а не в файле настройки хранилища. Это позволяет импортировать их в любой файл компонента, которому необходимо использовать перехватчики, и позволяет избежать потенциальных проблем циклического импорта.

**.withTypes()**

Раньше подход к «предварительному вводу» хуков в настройках вашего приложения был немного другим. Результат будет выглядеть примерно так, как показано ниже:

приложение/hooks.ts

import type { TypedUseSelectorHook } from 'react-redux'  
import { useDispatch, useSelector, useStore } from 'react-redux'  
import type { AppDispatch, AppStore, RootState } from './store'  
  
// Use throughout your app instead of plain `useDispatch` and `useSelector`  
export const useAppDispatch: () => AppDispatch = useDispatch  
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector  
export const useAppStore: () => AppStore = useStore

React Redux v9.1.0 добавляет к каждому из этих перехватчиков новый .withTypes метод, аналогичный методу, [.withTypes](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://redux-toolkit.js.org/usage/usage-with-typescript%23defining-a-pre-typed-createasyncthunk) найденному в Redux Toolkit createAsyncThunk.

Теперь настройка выглядит так:

приложение/hooks.ts

import { useDispatch, useSelector, useStore } from 'react-redux'  
import type { AppDispatch, AppStore, RootState } from './store'  
  
// Используйте во всем приложении вместо простых `useDispatch` и `useSelector`.  
export const useAppDispatch = useDispatch.withTypes<AppDispatch>()  
export const useAppSelector = useSelector.withTypes<RootState>()  
export const useAppStore = useStore.withTypes<AppStore>()

[**Использование**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#application-usage) **приложения**

**Определение состояния среза и** [**типы**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#define-slice-state-and-action-types)

Каждый файл слайса должен определять тип своего начального значения состояния, чтобы createSlice можно было правильно определить тип state редуктора в каждом случае.

Все сгенерированные действия должны быть определены с использованием PayloadAction<T> типа из Redux Toolkit, который принимает тип поля action.payload в качестве общего аргумента.

Здесь вы можете безопасно импортировать RootState тип из файла магазина. Это циклический импорт, но компилятор TypeScript может правильно обработать его для типов. Это может понадобиться для таких случаев, как написание функций селектора.

функции/счетчик/counterSlice.ts

import { createSlice, PayloadAction } from '@reduxjs/toolkit'  
import type { RootState } from '../../app/store'  
  
// Определите тип состояния среза  
interface CounterState {  
 value: number  
}  
  
// Определите начальное состояние, используя этот тип  
const initialState: CounterState = {  
 value: 0  
}  
  
export const counterSlice = createSlice({  
 name: 'counter',  
 // ` createSlice` определит тип состояния из аргумента `initialState`  
 initialState,  
 reducers: {  
 increment: state => {  
 state.value += 1  
 },  
 decrement: state => {  
 state.value -= 1  
 },  
 // Используйте тип PayloadAction для объявления содержимого action.payload.  
 incrementByAmount: (state, action: PayloadAction<number>) => {  
 state.value += action.payload  
 }  
 }  
})  
  
export const { increment, decrement, incrementByAmount } = counterSlice.actions  
  
// Другой код, например селекторы, могут использовать импортированный тип RootState.  
export const selectCount = (state: RootState) => state.counter.value  
  
export default counterSlice.reducer

Созданные создатели действий будут правильно типизированы, чтобы принять payload аргумент в зависимости от PayloadAction<T> типа, который вы указали для редуктора. Например, incrementByAmount требует number в качестве аргумента.

В некоторых случаях [TypeScript может излишне ужесточать тип начального состояния](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://github.com/reduxjs/redux-toolkit/pull/827) . Если это произойдет, вы можете обойти это as, приведя начальное состояние с помощью вместо объявления типа переменной:

// Обходной путь: приведение состояния вместо объявления типа переменной.  
const initialState = {  
 value: 0  
} as CounterState

[**Использование**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#application-usage) **типизированных хуков в** [**компонентах**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#use-typed-hooks-in-components)

В файлах компонентов импортируйте предварительно введенные хуки вместо стандартных хуков из React Redux.

функции/счетчик/Counter.tsx

import React, { useState } from 'react'  
  
import { useAppSelector, useAppDispatch } from 'app/hooks'  
  
import { decrement, increment } from './counterSlice'  
  
export function Counter() {  
 // Аргумент `state` уже правильно введен как `RootState`.  
 const count = useAppSelector(state => state.counter.value)  
 const dispatch = useAppDispatch()  
  
 // omit rendering logic  
}

**ПРЕДУПРЕЖДение О НЕПРАВИЛЬНОМ ИМПОРТЕ**

ESLint может помочь вашей команде легко импортировать нужные перехватчики. Правило typescript [-eslint/no-restricted-imports](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://github.com/typescript-eslint/typescript-eslint/blob/main/packages/eslint-plugin/docs/rules/no-restricted-imports.md) может отображать предупреждение, если случайно используется неправильный импорт.

Вы можете добавить это в свою конфигурацию ESLint в качестве примера:

"no-restricted-imports": "off",  
"@typescript-eslint/no-restricted-imports": [  
 "warn",  
 {  
 "name": "react-redux",  
 "importNames": ["useSelector", "useDispatch"],  
 "message": "Use typed hooks `useAppDispatch` and `useAppSelector` instead."  
 }  
],

**Ввод дополнительной** [**логики**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#typing-additional-redux-logic)

**Типовые** [**редукторы**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#type-checking-reducers)

[Редьюсеры](https://redux-js-org.translate.goog/tutorials/fundamentals/part-3-state-actions-reducers?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk) — это чистые функции, которые получают текущие state и входящие значения action в качестве аргументов и возвращают новое состояние.

Если вы используете Redux Toolkit createSlice, вам вряд ли понадобится специально вводить редуктор отдельно. Если вы действительно пишете автономный редуктор, обычно достаточно объявить тип значения initialState и ввести actionas UnknownAction:

import { UnknownAction } from 'redux'  
  
interface CounterState {  
 value: number  
}  
  
const initialState: CounterState = {  
 value: 0  
}  
  
export default function counterReducer(  
 state = initialState,  
 action: UnknownAction  
) {  
 // logic here  
}

Однако ядро Redux экспортирует Reducer<State, Action> тип, который вы также можете использовать.

**Middleware для проверки типов**

**Middleware (**промежуточное ПО) — это механизм расширения хранилища Redux. Middlewareсостоит из конвейера, который обертывает метод хранилища dispatch и имеет доступ к хранилищам dispatch и getState методам.

Ядро Redux экспортирует Middleware тип, который можно использовать для правильного ввода функции Middleware:

export interface Middleware<  
 DispatchExt = {}, // необязательное переопределение возвращаемого поведения `dispatch`  
 S = any, // type of the Redux store state  
 D extends Dispatch = Dispatch // тип способа отправки  
>

Пользовательское Middleware должно использовать этот тип и при необходимости Middleware передавать общие аргументы для S(состояния) и (отправки)D:

import { Middleware } from 'redux'  
  
import { RootState } from '../store'  
  
export const exampleMiddleware: Middleware<  
 {}, // Большинство Middleware не изменяют возвращаемое значение отправки.  
 RootState  
> = storeApi => next => action => {  
 const state = storeApi.getState() // правильно введено как RootState  
}

Общий вариант отправки, скорее всего, понадобится только в том случае, если вы отправляете дополнительные блоки внутри промежуточного программного обеспечения.

В тех случаях type RootState = ReturnType<typeof store.getState>, когда используется [циклическая ссылка на тип между определениями Middleware и хранилища,](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://github.com/reduxjs/redux/issues/4267) можно избежать, переключив определение типа RootState на:

const rootReducer = combineReducers({ ... });  
type RootState = ReturnType<typeof rootReducer>;

Переключение определения типа RootStateс помощью примера Redux Toolkit:

// вместо определения редукторов в поле редуктора configureStore, объедините их здесь:  
const rootReducer = combineReducers({ counter: counterReducer })  
  
// затем установите rootReducer в качестве объекта редуктора configureStore  
const store = configureStore({  
 reducer: rootReducer,  
 middleware: getDefaultMiddleware =>  
 getDefaultMiddleware().concat(yourMiddleware)  
})  
  
type RootState = ReturnType<typeof rootReducer>

**Проверка типов Redux** [**Thunks**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#type-checking-redux-thunks)

[Redux Thunk](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://github.com/reduxjs/redux-thunk) — это стандартное промежуточное программное обеспечение для написания синхронизирующей и асинхронной логики, которая взаимодействует с хранилищем Redux. Функция thunk получает dispatchи getState в качестве параметров. Redux Thunk имеет встроенный ThunkAction тип, который мы можем использовать для определения типов этих аргументов:

export type ThunkAction<  
 R, // Тип возвращаемого значения функции thunk

S, // тип состояния, используемый getState

E, // любой «дополнительный аргумент», введенный в преобразователь  
 A extends Action // известные типы действий, которые могут быть отправлены  
> = (dispatch: ThunkDispatch<S, E, A>, getState: () => S, extraArgument: E) => R

Обычно вам потребуется предоставить общие аргументы R(тип возвращаемого значения) и S(состояние). К сожалению, TS не позволяет предоставлять только некоторые общие аргументы, поэтому обычные значения для остальных аргументов — unknownfor E и UnknownActionfor A:

import { UnknownAction } from 'redux'  
import { sendMessage } from './store/chat/actions'  
import { RootState } from './store'  
import { ThunkAction } from 'redux-thunk'  
  
export const thunkSendMessage =  
 (message: string): ThunkAction<void, RootState, unknown, UnknownAction> =>  
 async dispatch => {  
 const asyncResp = await exampleAPI()  
 dispatch(  
 sendMessage({  
 message,  
 user: asyncResp,  
 timestamp: new Date().getTime()  
 })  
 )  
 }  
  
function exampleAPI() {  
 return Promise.resolve('Async Chat Bot')  
}

Чтобы уменьшить повторение, вы можете определить повторно используемый AppThunk тип один раз в файле хранилища, а затем использовать этот тип всякий раз, когда пишете переходник:

export type AppThunk<ReturnType = void> = ThunkAction<  
 ReturnType,  
 RootState,  
 unknown,  
 UnknownAction  
>

Обратите внимание, что это предполагает отсутствие значимого возвращаемого значения от преобразователя. Если ваш преобразователь возвращает обещание, и вы хотите [использовать возвращенное обещание после отправки преобразователя](https://redux-js-org.translate.goog/tutorials/essentials/part-5-async-logic?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#checking-thunk-results-in-components), вы можете использовать его как AppThunk<Promise<SomeReturnType>>.

**ОСТОРОЖНО**

Не забывайте, что перехватчик по умолчанию useDispatch не знает о thunks, поэтому отправка thunk вызовет ошибку типа. Обязательно [используйте обновленную форму Dispatch в своих компонентах, которая распознает thunks как приемлемый тип для отправки](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#define-root-state-and-dispatch-types) .

Использование с React [Redux](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#usage-with-react-redux)

Хотя [React Redux](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://react-redux.js.org) — это отдельная от самого Redux библиотека, она обычно используется с React.

Полное руководство о том, как правильно использовать React Redux с TypeScript, можно найти на [странице «Статическая типизация» в документации React Redux](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://react-redux.js.org/using-react-redux/static-typing). В этом разделе будут рассмотрены стандартные шаблоны.

Если вы используете TypeScript, типы React Redux поддерживаются отдельно в DefinitelyTyped, но включаются как зависимость пакета React-Redux, поэтому их следует устанавливать автоматически. Если вам все равно нужно установить их вручную, запустите:

npm install @types/react-redux

**Типизируем** [**хук**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#typing-the-useselector-hook) **useSelector**

Объявите тип параметра state в функции селектора, и тип возвращаемого значения useSelectorбудет определен в соответствии с типом возвращаемого значения селектора:

interface RootState {  
 isOn: boolean  
}  
  
// TS infers type: (state: RootState) => boolean  
const selectIsOn = (state: RootState) => state.isOn  
  
// TS infers `isOn` is boolean  
const isOn = useSelector(selectIsOn)

Это также можно сделать встроенно:

const isOn = useSelector((state: RootState) => state.isOn)

Однако вместо этого предпочтительнее создать предварительно набранный useAppSelector хук с правильным типом stateвстроенного кода.

**Типизируем** [**хук**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#typing-the-usedispatch-hook) **useDispatch**

По умолчанию возвращаемое значение — useDispatchэто стандартный Dispatchтип, определенный основными типами Redux, поэтому никаких объявлений не требуется:

const dispatch = useDispatch()

Однако вместо этого предпочтительнее создать предварительно набранный useAppDispatch хук с правильным типом Dispatchвстроенного кода.

**Типизируем** [**компонент**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#typing-the-connect-higher-order-component) **connect более высокого порядка**

Если вы все еще используете connect, вам следует использовать ConnectedProps<T> тип, экспортированный с помощью @types/react-redux^7.1.2, чтобы автоматически определить типы реквизитов connect. Для этого необходимо разделить connect(mapState, mapDispatch)(MyComponent) вызов на две части:

import { connect, ConnectedProps } from 'react-redux'  
  
interface RootState {  
 isOn: boolean  
}  
  
const mapState = (state: RootState) => ({  
 isOn: state.isOn  
})  
  
const mapDispatch = {  
 toggleOn: () => ({ type: 'TOGGLE\_IS\_ON' })  
}  
  
const connector = connect(mapState, mapDispatch)  
  
// Выведенный тип будет выглядеть так:  
// {isOn: boolean, toggleOn: () => void}  
type PropsFromRedux = ConnectedProps<typeof connector>  
  
type Props = PropsFromRedux & {  
 backgroundColor: string  
}  
  
const MyComponent = (props: Props) => (  
 <div style={{ backgroundColor: props.backgroundColor }}>  
 <button onClick={props.toggleOn}>  
 Toggle is {props.isOn ? 'ON' : 'OFF'}  
 </button>  
 </div>  
)  
  
export default connector(MyComponent)

**Использование с Redux** [**Toolkit**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#usage-with-redux-toolkit)

В разделе [«Настройка проекта стандартного набора инструментов Redux с TypeScript»](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#standard-redux-toolkit-project-setup-with-typescript) уже описаны обычные шаблоны использования configureStore и createSlice, а [на странице «Использование с TypeScript» набора инструментов Redux](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://redux-toolkit.js.org/usage/usage-with-typescript) подробно описаны все API-интерфейсы RTK.

Вот некоторые дополнительные шаблоны ввода, которые вы часто встретите при использовании RTK.

**Типизация configureStore**

configureStore выводит тип значения состояния из предоставленной функции корневого редуктора, поэтому никаких конкретных объявлений типов не требуется.

Если вы хотите добавить в хранилище дополнительное промежуточное программное обеспечение, обязательно используйте специализированные методы .concat()и .prepend(), включенные в массив, возвращаемый getDefaultMiddleware(), поскольку они правильно сохранят типы добавляемого промежуточного программного обеспечения. (При использовании простого расширения массива JS эти типы часто теряются.)

const store = configureStore({  
 reducer: rootReducer,  
 middleware: getDefaultMiddleware =>  
 getDefaultMiddleware()  
 .prepend(  
 // можно просто использовать правильно типизированное middlewares  
 additionalMiddleware,  
 // вы также можете ввести тип middlewares вручную  
 untypedMiddleware as Middleware<  
 (action: Action<'specialAction'>) => number,  
 RootState  
 >  
 )  
 // вызовы prepend и concat могут быть объединены в цепочку  
 .concat(logger)  
})

**Соответствие** [**действия**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#matching-actions) **(Action)**

У создателей действий, созданных с помощью RTK, есть matchметод, который действует как [предикат типа](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://www.typescriptlang.org/docs/handbook/2/narrowing.html%23using-type-predicates) . Вызов someActionCreator.match(action)выполнит сравнение строки с action.type строкой и, если он используется в качестве условия, сузит тип action до правильного типа TS:

const increment = createAction<number>('increment')  
function test(action: Action) {  
 if (increment.match(action)) {  
 // action.payload inferred correctly here  
 const num = 5 + action.payload  
 }  
}

Это особенно полезно при проверке типов действий в промежуточном программном обеспечении Redux, таком как пользовательское промежуточное программное обеспечение redux-observableи метод RxJS filter.

**Типизация createSlice**

**Определение отдельных** [**редукторов**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#defining-separate-case-reducers)

Если у вас слишком много редукторов регистра и их встроенное определение будет неудобным, или вы хотите повторно использовать редукторы регистра в разных срезах, вы также можете определить их вне вызова createSlice и ввести их как CaseReducer:

type State = number  
const increment: CaseReducer<State, PayloadAction<number>> = (state, action) =>  
 state + action.payload  
  
createSlice({  
 name: 'test',  
 initialState: 0,  
 reducers: {  
 increment  
 }  
})

**Типизация extraReducers**

Если вы добавляете extraReducersполе в createSlice, обязательно используйте форму «обратного вызова построителя», поскольку форма «простой объект» не может правильно определить типы действий. Передача создателя действия, сгенерированного RTK, builder.addCase()правильно определит тип action:

const usersSlice = createSlice({  
 name: 'users',  
 initialState,  
 reducers: {  
 // заполните здесь основную логику  
 },  
 extraReducers: builder => {  
 builder.addCase(fetchUserById.pending, (state, action) => {  
 // и `state`, и `action` теперь правильно набраны

// на основе состояния среза и создателя (creator) ожидающего действия  
 })  
 }  
})

**Типизация** [**обратных вызовов**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#typing-prepare-callbacks) **(Callbacks) prepare**

Если вы хотите добавить свойство meta или error к своему действию или настроить payload свое действие, вам необходимо использовать prepare нотацию для определения преобразователя регистра. Использование этой записи в TypeScript выглядит так:

const blogSlice = createSlice({  
 name: 'blogData',  
 initialState,  
 reducers: {  
 receivedAll: {  
 reducer(  
 state,  
 action: PayloadAction<Page[], string, { currentPage: number }>  
 ) {  
 state.all = action.payload  
 state.meta = action.meta  
 },  
 prepare(payload: Page[], currentPage: number) {  
 return { payload, meta: { currentPage } }  
 }  
 }  
 }  
})

**Исправление круговых типов в экспортированных** [**фрагментах**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#fixing-circular-types-in-exported-slices)

Наконец, в редких случаях вам может потребоваться экспортировать редуктор фрагментов определенного типа, чтобы решить проблему циклической зависимости типов. Это может выглядеть так:

export default counterSlice.reducer as Reducer<Counter>

**Типизация createAsyncThunk**

Для базового использования единственный тип, который вам нужно указать, createAsyncThunk — это тип единственного аргумента для обратного вызова создания полезных данных. Вы также должны убедиться, что возвращаемое значение обратного вызова введено правильно:

const fetchUserById = createAsyncThunk(  
 'users/fetchById',  
 // Объявите тип аргумента функции здесь:  
 async (userId: number) => {  
 const response = await fetch(`https://reqres.in/api/users/${userId}`)  
 // Предполагаемый тип возвращаемого значения: Promise<MyData>  
 return (await response.json()) as MyData  
 }  
)  
  
// здесь параметр `fetchUserById` автоматически преобразуется в `number`

// и отправка результирующего thunkAction вернет обещание правильно введенного

// «выполненного» или «отклоненного» действия.  
const lastReturnedAction = await store.dispatch(fetchUserById(3))

Если вам необходимо изменить типы параметра thunkApi, например указать тип возвращаемого stateзначения getState(), вы должны указать первые два общих аргумента для типа возвращаемого значения и аргумента полезной нагрузки, а также те «поля аргументов thunkApi», которые имеют отношение к объекту:

const fetchUserById = createAsyncThunk<  
 // Тип возвращаемого значения создателя полезной нагрузки  
 MyData,  
 // Первый аргумент создателя полезной нагрузки  
 number,  
 {  
 // Дополнительные поля для определения типов полей thunkApi  
 dispatch: AppDispatch  
 state: State  
 extra: {  
 jwt: string  
 }  
 }  
>('users/fetchById', async (userId, thunkApi) => {  
 const response = await fetch(`https://reqres.in/api/users/${userId}`, {  
 headers: {  
 Authorization: `Bearer ${thunkApi.extra.jwt}`  
 }  
 })  
 return (await response.json()) as MyData  
})

**Типизация createEntityAdapter**

При вводе createEntityAdapterтребуется только указать тип сущности в качестве единственного универсального аргумента. Обычно это выглядит так:

interface Book {  
 bookId: number  
 title: string  
 // ...  
}  
  
const booksAdapter = createEntityAdapter({  
 selectId: (book: Book) => book.bookId,  
 sortComparer: (a, b) => a.title.localeCompare(b.title)  
})  
  
const booksSlice = createSlice({  
 name: 'books',  
 // The type of the state is inferred here  
 initialState: booksAdapter.getInitialState(),  
 reducers: {  
 bookAdded: booksAdapter.addOne,  
 booksReceived(state, action: PayloadAction<{ books: Book[] }>) {  
 booksAdapter.setAll(state, action.payload.books)  
 }  
 }  
})

**Дополнительные** [**рекомендации**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#additional-recommendations)

**Используйте React Redux Hooks** [**API**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#use-the-react-redux-hooks-api)

**Мы рекомендуем использовать API перехватчиков React Redux в качестве подхода по умолчанию** . API перехватчиков гораздо проще использовать с TypeScript, так как useSelector это простой перехватчик, принимающий функцию селектора, а тип возвращаемого значения легко выводится из типа аргумента state.

Несмотря на то connect, что он по-прежнему работает нормально и его *можно* печатать, правильно печатать гораздо сложнее.

**Избегайте** [**объединений**](https://redux-js-org.translate.goog/usage/usage-with-typescript?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=uk#avoid-action-type-unions)

**Мы особенно *не* рекомендуем пытаться создавать объединения типов действий** , поскольку это не дает реальной пользы и фактически в некотором смысле вводит компилятор в заблуждение. См. сообщение специалиста по сопровождению RTK Ленца Вебера « [Не создавайте типы объединения с типами действий Redux»](https://translate.google.com/website?sl=auto&tl=ru&hl=uk&u=https://phryneas.de/redux-typescript-no-discriminating-union) , где объясняется, почему это проблема.

Кроме того, если вы используете createSlice, вы уже знаете, что все действия, определенные этим фрагментом, обрабатываются правильно.