React
TypeScript

Что такое типизация
Что такое TypeScript
Функционал TypeScript
React + TypeScript
Redux + TypeScript

Что такое строгая типизация

Типизация в программировании — это процесс определения типов данных, которые могут быть использованы в программе. Она определяет, какой тип данных может быть присвоен переменной, использован в функции или передан в качестве параметра.

Типизация бывает двух основных видов:

- **Статическая типизация** — типы данных определяются на этапе компиляции. Это означает, что тип переменной известен до выполнения программы. Например, такие языки, как TypeScript, Java, и C#, используют статическую типизацию.
- **Динамическая типизация** — типы данных определяются во время выполнения программы. В динамически типизированных языках (таких как JavaScript, Python или Ruby) переменная может содержать данные любого типа, и этот тип может изменяться в процессе работы программы.

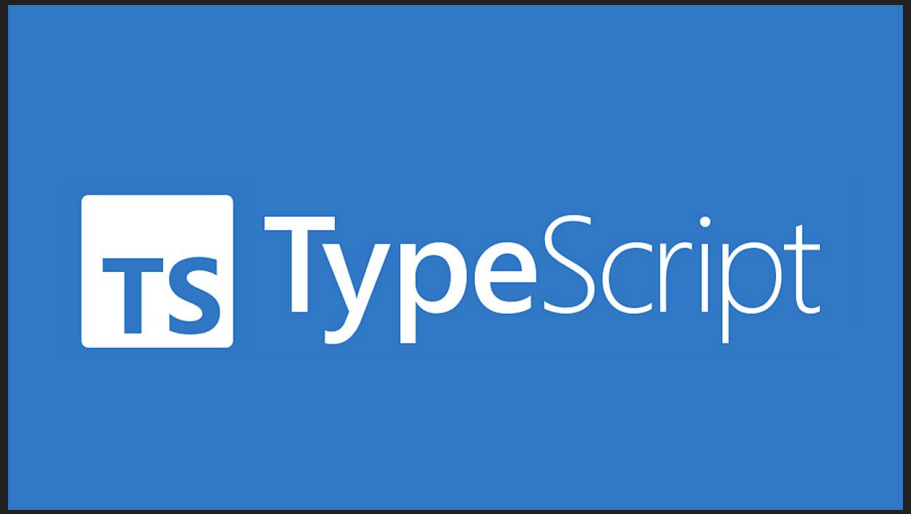
Также существует понятие сильной и слабой типизации:

- **Сильная типизация** — язык строго следит за типами данных и не допускает неявных преобразований типов. Например, в Python сложение числа и строки вызовет ошибку.
- **Слабая типизация** — язык позволяет неявные преобразования типов данных, что может привести к неожиданным результатам. JavaScript, например, пытается автоматически преобразовать типы при выполнении операций, что иногда может привести к непредсказуемому поведению.

Что такое TypeScript

TypeScript — это язык программирования, расширение JavaScript, созданный Microsoft. Он добавляет статическую типизацию и другие современные возможности в JavaScript. TypeScript компилируется (или транслируется) в обычный JavaScript, чтобы его можно было использовать в браузерах или на сервере.

Основная цель TypeScript — улучшить разработку больших и сложных JavaScript-приложений за счет добавления статической типизации, что помогает находить ошибки на этапе разработки, а также упрощает поддержку и расширение кода.



Плюсы TypeScript

Статическая типизация: Позволяет определить типы переменных, функций и параметров, что помогает обнаружить ошибки на этапе компиляции.

Поддержка современных стандартов JavaScript: TypeScript поддерживает все новейшие функции ECMAScript (ES), и это позволяет писать современный код, который затем компилируется в ES5 или ES6 для совместимости.

Автодополнение и интеллектуальная подсказка: Средства разработки, такие как Visual Studio Code, предлагают автодополнение и рефакторинг кода, что значительно улучшает производительность.

Упрощение отладки: Поскольку TypeScript предоставляет типы, можно быстрее находить ошибки, такие как неверное использование данных, которое может быть неочевидным в JavaScript.

Поддержка классов и интерфейсов: TypeScript добавляет более мощную поддержку ООП с использованием интерфейсов, абстрактных классов и других конструкций.

Совместимость с JavaScript: TypeScript — это подмножество JavaScript, поэтому любой корректный JavaScript-код является корректным TypeScript-кодом.

Минусы TypeScript

Более сложная настройка: TypeScript требует дополнительных шагов для настройки компилятора и инфраструктуры, что делает его использование несколько сложнее по сравнению с обычным JavaScript.

Более длительное время компиляции: Поскольку TypeScript необходимо компилировать в JavaScript, это добавляет шаг, который занимает дополнительное время.

Кривая обучения: Для новичков, уже знакомых с JavaScript, TypeScript может показаться сложным, особенно с учетом необходимости изучения типизации и других специфических функций.

Оверхед для небольших проектов: Использование TypeScript для небольших проектов может быть излишним, поскольку статическая типизация не всегда необходима для простого кода.

Как TypeScript работает под капотом

TypeScript компилируется в JavaScript при помощи специального компилятора, называемого **tsc (TypeScript Compiler)**. Этот компилятор принимает **.ts** файлы и преобразует их в **.js** файлы, которые могут выполняться в браузере или на сервере (например, в Node.js).

Процесс компиляции TypeScript выглядит следующим образом:

1. **Анализ кода:** Компилятор анализирует весь TypeScript-код, проверяет типы данных, обращается к декларациям типов и проверяет, соответствуют ли переданные параметры ожидаемым типам.
2. **Ошибки и предупреждения:** На этапе компиляции компилятор выводит ошибки, если типы не совпадают или если найдены другие проблемы (например, несоответствие интерфейсу).
3. **Генерация JavaScript:** После того как все ошибки исправлены, TypeScript компилируется в JavaScript. Этот JavaScript может быть ES5, ES6 или любой другой версией, что делает TypeScript гибким в плане поддержки различных сред выполнения.

Также TypeScript поддерживает **файлы деклараций типов (.d.ts)**, которые позволяют интегрироваться с библиотеками, написанными на чистом JavaScript, предоставляя информацию о типах и улучшая автодополнение в редакторах.

Функционал TypeScript-а

TypeScript — это надстройка над JavaScript, которая добавляет статическую типизацию и дополнительные возможности для разработки, позволяя создавать более масштабируемый и поддерживаемый код. Основные функции TypeScript включают в себя:

- Статическая типизация
- Вывод типов (Type Inference)
- Типы данных
- Интерфейсы и типы (Interfaces and Type Aliases)
- Классы и ООП
- Дженерики (Generics)
- Утилитарные типы (Utility Types)
- Аннотация типов для функций
- Асинхронность и типы Promise
- Типы Union и Intersection
- Типы Mapped и условные типы

Статическая типизация

Одно из основных преимуществ TypeScript — статическая типизация, которая позволяет определять типы переменных, параметров функций и возвращаемых значений. Это делает код более читаемым и позволяет обнаруживать ошибки до выполнения.

```
let count: number = 10;  // Переменная count имеет тип number
function greet(name: string): void {
  console.log(`Hello, ${name}`);
}
```

Вывод типов (Type Inference)

TypeScript умеет автоматически выводить типы на основе значения, присвоенного переменной, или контекста. Это помогает сократить количество кода, требующего явного указания типов.

```
let message = "Hello, TypeScript!"; // TypeScript автоматически понимает, что это строка
```

Типы данных

Примитивные типы:

- string
- number
- boolean
- null
- undefined
- symbol (ES6+)
- bigint (ES11+)

Специальные типы:

- **any:** переменная может быть любого типа.
- **unknown:** переменная может быть любого типа, но требуется проверка перед использованием.
- **void:** используется для функций, которые ничего не возвращают.
- **never:** для функций, которые никогда не возвращают значение (например, при выбросе ошибки).
- **tuple (кортеж):** фиксированный массив с элементами разных типов.
- **enum:** перечисление возможных значений.

Интерфейсы и типы (Interfaces and Type Aliases)

Интерфейсы (interfaces) и типы (type aliases) позволяют описывать форму объектов, контракт для структур данных. Они могут использоваться для определения структур, которые должны следовать конкретные объекты или функции.

```ts
interface User {
  name: string;
  age: number;
}

const user: User = {
  name: "Alice",
  age: 25,
};
```

```ts
type Point = {
  x: number;
  y: number;
};

let point: Point = { x: 10, y: 20 };
```

Разница между Type и Interface (типами и интерфейсами)

В TypeScript есть два способа для определения структур данных: type и interface. Оба они позволяют описывать типы данных и могут использоваться для определения объектов, функций, массивов и других структур. Однако между ними есть несколько важных различий и особенностей. Рассмотрим их подробнее:

Общие черты type и interface

- **Описывают структуры данных:** И типы, и интерфейсы могут использоваться для описания объектов, функций и массивов.
- **Являются взаимозаменяемыми в большинстве случаев:** В некоторых случаях type и interface могут выполнять одинаковые задачи, и выбор между ними зависит от предпочтений разработчика.

Расширение (наследование)

interface поддерживает расширение через ключевое слово extends. Один интерфейс может расширять другой, а также можно расширять несколько интерфейсов.

type также можно комбинировать с другими типами с помощью пересечений (intersection types) с использованием оператора &.

```ts
interface Person {
  name: string;
}

interface Employee extends Person {
  employeeId: number;
}

const emp: Employee = { name: "John", employeeId: 123 };
```

```ts
type Person = {
  name: string;
};

type Employee = Person & {
  employeeId: number;
};

const emp: Employee = { name: "Jane", employeeId: 456 };
```

Объявление нескольких раз слиянием

Интерфейсы могут быть объявлены несколько раз, и TypeScript объединит их определения (это называется "слияние деклараций").

Это удобно, когда необходимо расширить уже существующий интерфейс, например, для добавления новых свойств.

Типы не поддерживают слияние. Повторное определение типа вызовет ошибку.

```
type Car = {
  brand: string;
};

// Ошибка: Duplicate identifier 'Car'.
type Car = {
  model: string;
};
```

```
interface Car {
  brand: string;
}

interface Car {
  model: string;
}

const myCar: Car = { brand: "Toyota", model: "Corolla" };
```

Использование с примитивными типами и объединениями

Типы (type) позволяют использовать примитивные типы, объединения и пересечения, что делает их более гибкими для описания сложных структур.

Интерфейсы не могут быть использованы для описания примитивных типов или объединений.

```
type ID = string | number; // Может быть либо строкой, либо числом
type Status = "success" | "error" | "loading"; // Литеральный тип
```

```
// Такого с интерфейсом сделать нельзя.
interface ID = string | number; // Ошибка
```

Более удобные для классов

Интерфейсы чаще используются для описания классов. Они могут использоваться в качестве контрактов, чтобы гарантировать, что класс реализует определенные методы и свойства.

В TypeScript ключевое слово **implements** используется для указания, что класс реализует интерфейс или абстрактный класс.

- Если класс не реализует все свойства и методы интерфейса, TypeScript выдаст ошибку компиляции.
- Интерфейсы позволяют создать четкие контракты для классов, что делает код более структурированным и легким для понимания.

```
interface Animal {
  name: string;
  makeSound(): void;
}

class Dog implements Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  makeSound() {
    console.log("Woof!");
  }
}
```

Когда использовать interface, а когда type?

Интерфейсы рекомендуется использовать, когда нужно описывать объекты и классы, особенно если предполагается их расширение или слияние.

Типы лучше использовать, когда нужно описывать примитивы, объединения, пересечения, функции, а также когда требуется гибкость и возможность комбинирования.

Заключение

- **interface:** Идеально подходит для описания объектов и классов. Поддерживает расширение и слияние, что делает его полезным для описания сложных и расширяемых структур.
- **type:** Универсальный инструмент для описания примитивных типов, объединений и пересечений. Лучше подходит для сложных типов, таких как комбинации различных структур, примитивов и литералов.

Выбор между type и interface во многом зависит от конкретной задачи и личных предпочтений, но оба инструмента мощные и дают возможность сделать код TypeScript более выразительным и безопасным.

Классы и ООП

TypeScript расширяет функционал классов, добавляя поддержку модификаторов доступа **(public, private, protected)**, что позволяет лучше контролировать доступ к свойствам и методам. Также поддерживается наследование и абстрактные классы.

public: Доступен везде. Используется для методов и свойств, которые должны быть доступны внешнему коду.

private: Доступен только внутри класса. Используется для скрытия данных и обеспечения инкапсуляции.

protected: Доступен внутри класса и его подклассов. Используется для управления доступом в иерархии классов, не открывая данные для внешнего кода.

static: Доступен на уровне класса. Используется для методов и свойств, которые относятся ко всему классу, а не к конкретным экземплярам.

readonly: Доступен только для чтения. Используется для защиты свойств от изменений после их первоначальной установки.

abstract: Доступен только в абстрактных классах. Используется для объявления методов или классов, которые должны быть реализованы в подклассах.

```typescript
abstract class Shape {
    // protected свойство доступно только в классе и его подклассах
    protected readonly name: string;

    // статическое свойство, доступное на уровне класса
    static shapeCount: number = 0;

    constructor(name: string) {
        this.name = name;
        Shape.shapeCount++; // Увеличиваем счетчик форм
    }

    // абстрактный метод, который должны реализовать подклассы
    abstract area(): number;

    // public метод, доступный внешнему коду
    public getName(): string {
        return this.name;
    }

    // private метод, доступный только внутри класса
    private logShape(): void {
        console.log(`Shape: ${this.name}`);
    }

    // public метод для логирования формы
    public log(): void {
        this.logShape();
    }
}

// Подкласс Rectangle, который наследует от Shape
class Rectangle extends Shape {
    private width: number; // private свойство

    constructor(name: string, width: number) {
        super(name);
        this.width = width;
    }

    // Реализация абстрактного метода area
    public area(): number {
        return this.width * this.width; // Пример площади (квадрат)
    }
}

// Пример использования
const rect = new Rectangle("MyRectangle", 5);
console.log(rect.getName()); // Доступен public метод
console.log(rect.area()); // Доступен public метод
rect.log(); // Логирует форму
console.log(`Total shapes: ${Shape.shapeCount}`); // Доступ к статическому свойству
```

Дженерики (Generics)

Дженерики позволяют создавать компоненты, которые работают с разными типами данных, сохраняя при этом их типобезопасность. Они полезны для создания функций, классов и интерфейсов, которые могут быть использованы с разными типами.

```
function identity<T>(value: T): T {
  return value;
}

let result = identity<number>(42); // Тип T заменяется на number
```

Утилитарные типы (Utility Types)

TypeScript предоставляет утилитарные типы, которые помогают манипулировать существующими типами. Например:

- **Partial<T>:** делает все свойства типа T необязательными.
- **Readonly<T>:** делает все свойства типа T доступными только для чтения.
- **Pick<T, K>:** выбирает определенные свойства из типа T.

```
interface User {
  name: string;
  age: number;
  email: string;
}

type UserPreview = Pick<User, "name" | "age">;

const preview: UserPreview = { name: "Alice", age: 25 }; // только имя и возраст
```

Аннотация типов для функций

TypeScript позволяет аннотировать параметры и возвращаемое значение функций, что делает код более предсказуемым.

```
function add(a: number, b: number): number {
  return a + b;
}
```

Асинхронность и типы Promise

TypeScript поддерживает асинхронные функции и типы Promise, что позволяет описывать тип данных, который возвращается промисом.

```
async function fetchData(): Promise<string> {
  return "Data fetched";
}
```

Типы Union и Intersection

TypeScript поддерживает объединение типов (union) и пересечение типов (intersection), которые помогают создавать более гибкие типы данных.

```ts
function printId(id: number | string) {
  console.log("ID:", id);
}
```

```ts
interface Person {
  name: string;
}

interface Employee {
  employeeId: number;
}

type EmployeeDetails = Person & Employee;

const emp: EmployeeDetails = {
  name: "John",
  employeeId: 101,
};
```

Типы Mapped и условные типы

Mapped типы позволяют создавать новые типы на основе существующих, преобразуя их свойства. Условные типы (conditional types) позволяют определять типы в зависимости от условий.

```
type ConditionalType<T> = T extends U ? X : Y;
```

```
type MappedType<T> = {
    [K in keyof T]: NewType;
};
```

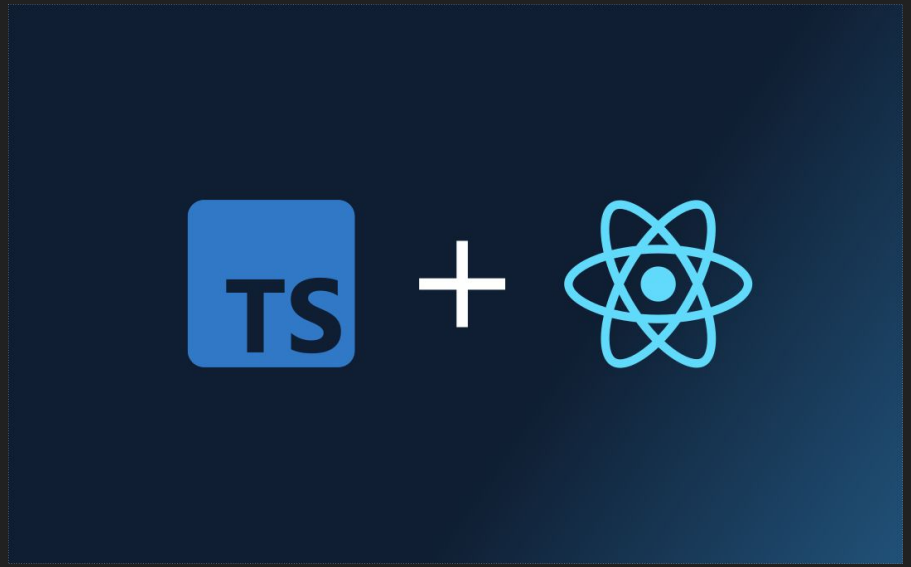
```
type Person = {
    name: string;
    age: number;
    isActive: boolean;
};

// Создаем новый тип, где тип свойства будет "string" если оно - "string" или "boolean", иначе "number"
type CustomMappedType<T> = {
    [K in keyof T]: T[K] extends string ? string : (T[K] extends boolean ? string : number);
};

// Результат: { name: string; age: number; isActive: string; }
type NewPerson = CustomMappedType<Person>;

const newPerson: NewPerson = {
    name: "Alice",
    age: 30, // будет number
    isActive: "true" // будет string
};
```

React + TypeScript

Внедрение TypeScript в React проект требует установки зависимостей, настройки конфигурационного файла, переименования файлов и исправления ошибок типизации. Это улучшает качество кода за счет проверки типов на этапе разработки, что особенно полезно в больших проектах.

Основные аспекты:

- Типизация компонентов
- Типизация хуков
- Типизация контекста
- Типизация событий
- Типизация библиотек



Файлы:

*.js => *.ts (основные файлы)

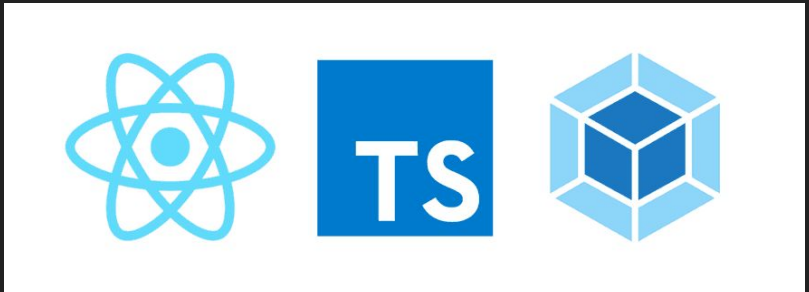
*.jsx => *.tsx (файлы компоненты)

*.d.ts (файлы глобальной типизации)

Создание проекта с TypeScript-ом

Создание нового проекта: Используйте Create React App с шаблоном TypeScript:
npx create-react-app my-app --template typescript

Чтобы использовать TypeScript в проекте React с помощью Vite, вы можете следовать следующей инструкции:
npm create vite@latest my-vite-app -- --template react-ts



Настройка TypeScript-а на webpack проекте

npm install --save-dev typescript ts-loader @types/react @types/react-dom

- typescript — сам TypeScript.
- ts-loader — загрузчик для компиляции TypeScript в JavaScript.
- @types/react и @types/react-dom — типы для React (если вы используете React).

Создайте файл tsconfig.json в корне вашего проекта, если его еще нет. Вот пример конфигурации

```jsonc
{
  "compilerOptions": {
    // Указывает целевую версию ECMAScript для компиляции (в данном случае ES5).
    "target": "es5",
    // Определяет набор библиотек, которые будут включены в проект.
    "lib": [
      "dom",          // Типы для стандартных объектов и функций веб-API.
      "dom.iterable", // Поддержка итерации по объектам DOM.
      "esnext"        // Типы для возможностей будущих версий JavaScript.
    ],
    // Позволяет использовать файлы JavaScript в проекте (обрабатывает .js файлы).
    "allowJs": true,
    // Пропускает проверку типов в файлах деклараций (.d.ts) для ускорения компиляции.
    "skipLibCheck": true,
    // Включает поддержку интероперабельности между модулями CommonJS и ES.
    "esModuleInterop": true,
    // Позволяет импортировать модули без явного экспорта по умолчанию.
    "allowSyntheticDefaultImports": true,
    // Включает строгую проверку типов для повышения безопасности кода.
    "strict": true,
    // Проверяет, что все импортируемые файлы имеют одинаковый регистр в именах.
    "forceConsistentCasingInFileNames": true,
    // Генерирует ошибку, если есть случаи, которые могут "упасть" в следующий в операторе switch.
    "noFallthroughCasesInSwitch": true,
    // Указывает тип модуля для генерации (в данном случае ESNext).
    "module": "esnext",
    // Определяет метод разрешения модулей (Node.js).
    "moduleResolution": "node",
    // Позволяет импортировать JSON-файлы как модули.
    "resolveJsonModule": true,
    // Проверяет, что каждый файл может быть обработан в изолированном режиме.
    "isolatedModules": true,
    // Не генерирует выходные файлы (.js) при компиляции.
    "noEmit": true,
    // Определяет, как будет обрабатываться JSX (в данном случае используется для React).
    "jsx": "react-jsx"
  },
  // Указывает, какие папки или файлы будут включены в проект.
  "include": [
    "src"  // Включает все файлы в папке src.
  ]
}
```

Настройка TypeScript-а на vite проекте

npm install --save-dev typescript @types/node @types/react @types/react-dom

Создание конфигурационного файла TypeScript: Создайте файл tsconfig.json в корневой директории вашего проекта, если он еще не создан.

```
{
    "compilerOptions": {
        "target": "esnext",
        "module": "esnext",
        "jsx": "react-jsx",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true,
        "baseUrl": ".",
        "paths": {
            "@/*": ["src/*"]
        }
    },
    "include": ["src/**/*"],
    "exclude": ["node_modules"]
}
```

Типизация Функциональных компонентов

Вы можете использовать React.FC или описать типы пропсов вручную.

```tsx
import React from 'react';

interface MyComponentProps {
    title: string;
    isActive: boolean;
}

const MyComponent: React.FC<MyComponentProps> = ({ title, isActive }) => {
    return (
        <div>
            <h1>{title}</h1>
            {isActive && <p>Активен</p>}
        </div>
    );
};
```

Типизация Классовых компонентов

Классовые компоненты можно типизировать, указывая тип пропсов и состояния.

```tsx
import React, { Component } from 'react';

type MyClassComponentProps = {
  initialCount: number;
};

type MyClassComponentState = {
  count: number;
};

class MyClassComponent extends Component<MyClassComponentProps, MyClassComponentState> {
  state: MyClassComponentState = {
    count: this.props.initialCount,
  };

  render() {
    return <div>Count: {this.state.count}</div>;
  }
}
```

Типизация хуков

Типизация состояния с использованием useState: При использовании хуков можно также указывать типы:

```
import React, { useState } from 'react';

const Counter: React.FC = () => {
    const [count, setCount] = useState<number>(0);

    return (
        <div>
            <p>Счетчик: {count}</p>
            <button onClick={() => setCount(count + 1)}>Увеличить</button>
        </div>
    );
};
```

Типизация контекста

При создании контекста также можно указывать типы для значений, которые он будет предоставлять:

```tsx
import React, { createContext, useContext } from 'react';

interface AuthContextType {
    isAuthenticated: boolean;
    login: () => void;
    logout: () => void;
}

const AuthContext = createContext<AuthContextType | undefined>(undefined);

const useAuth = () => {
    const context = useContext(AuthContext);
    if (!context) {
        throw new Error('useAuth must be used within an AuthProvider');
    }
    return context;
};
```

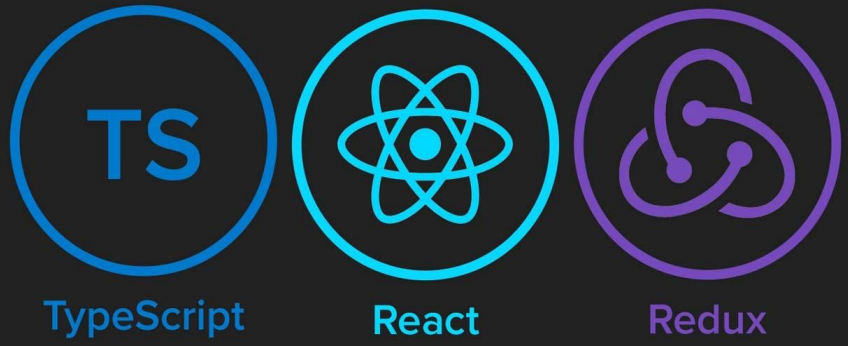
Типизация событий

Вы можете указывать типы для событий, что позволяет избежать ошибок при обработке событий:

```
const handleClick = (event: React.MouseEvent<HTMLButtonElement>) => {
    console.log(event.currentTarget);
};

return <button onClick={handleClick}>Нажми меня</button>;
```

Типизация Redux

Типизация Redux в TypeScript — это важный аспект, который позволяет вам работать с состоянием вашего приложения и действиями (actions) более безопасно и удобно. TypeScript помогает избежать ошибок во время компиляции, а также предоставляет автодополнение и поддержку IntelliSense. Давайте рассмотрим, как можно типизировать различные аспекты Redux, включая состояние, действия и редьюсеры.



npm install redux react-redux
npm install --save-dev @types/react-redux

Типизация состояния (State)

Сначала определите интерфейс для вашего состояния приложения. Это поможет вам установить структуру данных, которые будут храниться в Redux.

```
interface RootState {
    user: {
        name: string;
        age: number;
    };
    posts: Array<{
        id: number;
        title: string;
        content: string;
    }>;
}
```

Типизация действий (Actions)

Вам нужно определить типы действий, которые будут отправляться в ваш Redux store. Это можно сделать с помощью type или interface.

```
// Определяем типы действий
const ADD_USER = 'ADD_USER';
const ADD_POST = 'ADD_POST';

interface AddUserAction {
    type: typeof ADD_USER;
    payload: {
        name: string;
        age: number;
    };
}

interface AddPostAction {
    type: typeof ADD_POST;
    payload: {
        id: number;
        title: string;
        content: string;
    };
}

type UserActions = AddUserAction | AddPostAction;
```

Типизация редьюсеров (Reducers)

Редьюсеры принимают текущее состояние и действия и возвращают новое состояние. Вам нужно типизировать состояние и действия, передавая их в редьюсер.

```
const initialState: RootState = {
    user: {
        name: '',
        age: 0,
    },
    posts: [],
};

const rootReducer = (state = initialState, action: UserActions): RootState => {
    switch (action.type) {
        case ADD_USER:
            return {
                ...state,
                user: action.payload,
            };
        case ADD_POST:
            return {
                ...state,
                posts: [...state.posts, action.payload],
            };
        default:
            return state;
    }
};
```

Типизация селекторов (Selectors) и функций-диспетчеров (Dispatchers)

При использовании функций для отправки действий (dispatch), вам нужно убедиться, что они также типизированы.

Селекторы — это функции, которые извлекают данные из состояния. Типизация селекторов поможет избежать ошибок.

```
import { useDispatch } from 'react-redux';

const useAppDispatch = () => useDispatch<ThunkDispatch<RootState, void, UserActions>>();

const dispatch = useAppDispatch();

// Использование
dispatch({
    type: ADD_USER,
    payload: { name: 'Alice', age: 30 },
});
```

```
import { RootState } from './store';

const selectUser = (state: RootState) => state.user;
const selectPosts = (state: RootState) => state.posts;
```

Типизация Redux ToolKit

Определение интерфейсов

User: Интерфейс для описания объекта пользователя, который имеет свойства name и age.

Post: Интерфейс для описания объекта поста, который имеет свойства id, title и content.

RootState: Интерфейс, представляющий общее состояние вашего Redux store, которое включает user и массив posts.

```typescript
import { createSlice, PayloadAction } from '@reduxjs/toolkit';

interface User {
    name: string;
    age: number;
}

interface Post {
    id: number;
    title: string;
    content: string;
}

interface RootState {
    user: User;
    posts: Post[];
}

const initialState: RootState = {
    user: { name: '', age: 0 },
    posts: [],
};

const userSlice = createSlice({
    name: 'user',
    initialState,
    reducers: {
        addUser(state, action: PayloadAction<User>) {
            state.user = action.payload;
        },
        addPost(state, action: PayloadAction<Post>) {
            state.posts.push(action.payload);
        },
    },
});

// Экспортируем действия и редьюсер
export const { addUser, addPost } = userSlice.actions;
export default userSlice.reducer;
```

Ресурсы

Документация TypeScript-а на русском - ТЫК

Оф. документация TypeScript-а - ТЫК

Метанит руководство по TypeScript-у - ТЫК

Использования TypeScript в React - ТЫК

Типизация React (TypeScript) - ТЫК