Подход к типобезопасной разработке на TypeScript

Дмитрий Харитонов geakstr@me.com

Что обсудим

- Ценность TypeScript
- Проблемы в проектах на TypeScript
- Возможности TypeScript
- Подход к написанию типобезопасного кода
- Инструменты для реализации подхода

Ценность TypeScript

- Код рефакторится с удовольствием
- Баги чинятся безбоязненно
- Код документирует сам себя
- Упрощается вхождение в проект разработчиков

Поддерживание архитектуры приложений со строгой типизацией требует значительных усилий и заставляет писать код, который писать не хочется

```ts
// Injects props and removes them from the prop requirements.
// Will not pass through the injected props if they are passed in during
// render. Also adds new prop requirements from TNeedsProps.
export type InferableComponentEnhancerWithProps<TInjectedProps, TNeedsProps> =
  <C extends ComponentType<Matching<TInjectedProps, GetProps<C>>>>(
    component: C
  ) => ConnectedComponentClass<C, Omit<GetProps<C>, keyof Shared<TInjectedProps, GetProps<C>>> & TNeedsProps>;

// Injects props and removes them from the prop requirements.
// Will not pass through the injected props if they are passed in during
// render.
export type InferableComponentEnhancer<TInjectedProps> =
  InferableComponentEnhancerWithProps<TInjectedProps, {}>;

export type InferThunkActionCreatorType<TActionCreator extends ( ...args: any[]) => any> =
  TActionCreator extends ( ...args: infer TParams) => ( ...args: any[]) => infer TReturn
  ? ( ...args: TParams) => TReturn
  : TActionCreator;

export type HandleThunkActionCreator<TActionCreator> =
  TActionCreator extends ( ...args: any[]) => any
  ? InferThunkActionCreatorType<TActionCreator>
  : TActionCreator;

// redux-thunk middleware returns thunk's return value from dispatch call
// https://github.com/reduxjs/redux-thunk#composition
export type ResolveThunks<TDispatchProps> =
  TDispatchProps extends { [key: string]: any }
  ? {
    [C in keyof TDispatchProps]: HandleThunkActionCreator<TDispatchProps[C]>
  }
  : TDispatchProps;

.................................
.........................................
..................
.................................................
.................................
```

Но использовать просто

```
const Items = connect(
  (state: RootState) ⇒ ({ items: state.app.items }),
  { fetchItems }
)(props ⇒ {
  React.useEffect(props.fetchItems, []);
  return (
    <div>
      {props.items.map(item ⇒
        <div key={item.id}>{item.title}></div>
      )}
    </div>
  );
});
```

Не хочется писать типы — и не нужно*

Привычный код на JavaScript

Возможности TypeScript

Хотим писать строготипизированный TypeScript код, выглядящий как обычный JavaScript

- Функции-помощники — сложные
- Использование — простое

Literal types and union

```ts
type Animals = "cat" | "dog" | "parrot";
const isCat = (animal: Animals) => animal === "cat";
isCat("apple"); // !
```

```ts
type DiceValues = 1 | 2 | 3 | 4 | 5 | 6;
const rollDice = (): DiceValues => 6;
const storeDiceValue = (value: DiceValues) => { ... }
storeDiceValue(10); // !
storeDiceValue(rollDice()); // ✅
```

Выведение типов в условных выражениях

```ts
const lower = (value: string | null) => {
  return value ? value.toLowerCase() : null;
};
```

```ts
const lower = (value: string | number | null): string | null => {
  if (typeof value === "string") {
    return value.toLowerCase();
  } else if (typeof value === "number") {
    return value.toString();
  }
  return null;
};
```

Выведение типов в условных выражениях

```ts
enum ActionType {
  DAMAGE = "👊",
  HEAL = "💊"
}

type Action<T extends ActionType, P> = {
  type: T;
  payload: P;
};


type DamageAction = Action<ActionType.DAMAGE, { damage: number }>;
type HealAction = Action<ActionType.HEAL, { points: number }>;
type RootAction = DamageAction | HealAction;
```

Выведение типов в условных выражениях

```
const turn = (health: number, action: RootAction) ⇒ {
  switch (action.type) {
    case ActionType.HEAL:
      return Math.min(100, health + action.payload.points);
    case ActionType.DAMAGE:
      return Math.max(0, health - action.payload.damage);
    default:
      return health;
  }
};
```

Выведение типов в условных выражениях

```ts
type JsonResult =
  | { ok: true; obj: object }
  | { ok: false; err: string };

const parseJson = (json: string): JsonResult => {
  try {
    return { ok: true, obj: JSON.parse(json) };
  } catch (error) {
    return { ok: false, err: "Invalid Json" };
  }
};

const parsed = parseJson(" ... ");
if (parsed.ok) {
  console.log(parsed.obj);
} else {
  console.error(parsed.err);
}
```

keyof and generics

```ts
interface HTMLElementTagNameMap {
  "a": HTMLAnchorElement;
  "abbr": HTMLElement;
  "address": HTMLElement;
  "applet": HTMLAppletElement;
  "area": HTMLAreaElement;
  "article": HTMLElement;
  "aside": HTMLElement;
  "audio": HTMLAudioElement;
  // ...
}

// "a" | "abbr" | "address" ...
type HTMLElementTagNames = keyof HTMLElementTagNameMap;
```

keyof and generics

```ts
declare function createElement<K extends keyof HTMLElementTagNameMap>(
  tagName: K
): HTMLElementTagNameMap[K];
```

```ts
const a = createElement("a"); // HTMLAnchorElement
```

Conditional types

```ts
type NonNullable<T> = T extends null | undefined ? never : T;

NonNullable<"a" | 1 | [] | null | undefined>; // "a" | 1 | []
```

```ts
type EmailAddress = string | string[] | null | undefined;
NonNullable<EmailAddress>; // string | string[]
```

Mapped and conditional types

```ts
type Pick<T, K extends keyof T> = { [P in K]: T[P] };

Pick<{
  id: string;
  email: string;
  password: string;
}, "id" | "email">; // { id: string; email: string }
```

Mapped and conditional types

```ts
type Exclude<T, U> = T extends U ? never : T;

Exclude<"id" | "email" | "password", "password">; // "id" | "email"

Exclude<{
  id: string;
  email: string;
  password: string;
}, { password: string }>; // never 😕

Exclude<{
  id: string;
  email: string;
  password: string;
}, "password">; // { id: string; email: string; password: string } 😕
```

Mapped and conditional types

```ts
type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>;

type UserWithPassword = {
  id: string;
  password: string;
  email: string;
};

type User = Omit<UserWithPassword, "password">;
// { id: string, email: string }
```

Mapped and conditional types

```ts
type NonNullableObject<T> = Pick<T, {
  [P in keyof T]: null extends T[P] ? never : P
}[keyof T]>;
```

```ts
type User = {
  id: string;
  email: string | null;
};
NonNullableObject<User>; // { id: string }
```

Mapped and conditional types

```ts
type NonNullableObject<T> = Pick<T, {
  [P in keyof T]: null extends T[P] ? never : P
}[keyof T]>;
```

```ts
type User = {
  id: string;
  email: string | null;
};
NonNullableObject<User>; // { id: string }
```

Mapped and conditional types

```ts
type PickByCondition<T, C> = Pick<T, {
  [P in keyof T]: C extends T[P] ? P : never
}[keyof T]>;

type User = {
  id: string;
  email: string;
  age: number | null;
  login(): boolean;
  logout(): void;
};
type UserMethods = PickByCondition<User, ( ...args: any[]) => any>;
// { login(): boolean; logout(): void }
```

Mapped and conditional types

```ts
type ReturnType<T extends ( ... args: any[]) ⇒ any> =
  T extends ( ... args: any[]) ⇒ infer R ? R : any;

function createUser(login: string) {
  return {
    id: generateId(),
    login: login
  }
};
ReturnType<typeof createUser>; // { id: number, login: string}
```

Mapped and conditional types

```ts
type FirstArg<T> =
  T extends (a: infer A, ...args: any[]) => any ? A : any;
function formatUserTitle(name: string, age: number) {
  return `${name} (${age})`
};
FirstArg<typeof formatUserTitle>; // string
```

- Типы TypeScript могут быть сложным
- Но они должны делать свою работу незаметно для конечного разработчика
- В любой непонятной ситуации пишем функцию, которая инкапсулирует работу с типами

github.com/geakstr/safenv

Набор функций для создания типобезопасной среды разработки

- Применяем мощь TypeScript в утилитных функциях
- Используем их как обычный JavaScript
- Не перегружаем лишним шумом логику приложения
- Проще добавлять код, рефакторить, чинить баги
- Код документирует сам себя
- Упрощаем вхождение в проект разработчиков

Полезные ссылки

- TypeScript Docs — typescriptlang.org/docs
- Conditional types in TypeScript
  artsy.github.io/blog/2018/11/21/conditional-types-in-typescript
- TypeScript Evolution — Marius Schulz
  mariusschulz.com/blog/series/typescript-evolution
- piotrwitek/utility-types — набор полезных типов
- piotrwitek/typesafe-actions — работа с redux actions
- remeda/remeda — утилиты для обработки данных

Спасибо!

Дмитрий Харитонов

geakstr@me.com

twitter.com/geakstr