Tipos de Literais de Template

Os tipos de literais de template se baseiam em tipos de literais de string e têm a capacidade de se expandir em muitas strings via uniões.

Eles têm a mesma sintaxe que strings de literais de template em JavaScript, mas são usados em posições de tipo. Quando usados com tipos literais concretos, um literal de template produz um novo tipo de literal de string ao concatenar os conteúdos.

```
type World = "world";

type Greeting = `hello ${World}`;

type Greeting = "hello world"
```

Quando uma união é usada na posição interpolada, o tipo é o conjunto de cada possível literal de string que poderia ser representado por cada membro da união:

```
type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";

type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;

type AllLocaleIDs = "welcome_email_id" | "email_heading_id" | "footer_title_id" |
"footer_sendoff_id"
```

Para cada posição interpolada no literal de template, as uniões são multiplicadas:

```
type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
type Lang = "en" | "ja" | "pt";

type LocaleMessageIDs = `${Lang}_${AllLocaleIDs}`;

type LocaleMessageIDs = "en_welcome_email_id" | "en_email_heading_id" |
"en_footer_title_id" | "en_footer_sendoff_id" | "ja_welcome_email_id" |
"ja_email_heading_id" | "ja_footer_title_id" | "ja_footer_sendoff_id" |
"pt_welcome_email_id" | "pt_email_heading_id" | "pt_footer_title_id" |
"pt_footer_sendoff_id"
```

Recomendamos geralmente que as pessoas usem a geração antecipada para grandes uniões de strings, mas isso é útil em casos menores.

Uniões de Strings em Tipos

O poder nos literais de template vem ao definir uma nova string com base nas informações dentro de um tipo.

Considere o caso em que uma função (`makeWatchedObject`) adiciona uma nova função chamada `on()` a um objeto passado. Em JavaScript, sua chamada pode ser algo como: `makeWatchedObject(baseObject)`. Podemos imaginar que o objeto base se pareça com:

```
const passedObject = {
  firstName: "Saoirse",
  lastName: "Ronan",
```

```
  age: 26,
};
```

A função `on` que será adicionada ao objeto base espera dois argumentos, um `eventName` (uma `string`) e um `callback` (uma `function`).

O `eventName` deve estar na forma `attributeInThePassedObject + "Changed"`; assim, `firstNameChanged` é derivado do atributo `firstName` no objeto base.

A função `callback`, quando chamada:

- Deve receber um valor do tipo associado ao nome `attributeInThePassedObject`; assim, como `firstName` é tipado como `string`, o callback para o evento `firstNameChanged` espera um `string` a ser passado a ele no momento da chamada. Da mesma forma, eventos associados a `age` devem esperar ser chamados com um argumento `number`.
- Deve ter o tipo de retorno `void` (para simplicidade da demonstração).

A assinatura ingênua da função `on()` poderia ser assim: `on(eventName: string, callback: (newValue: any) => void)`. No entanto, na descrição anterior, identificamos restrições de tipo importantes que gostaríamos de documentar em nosso código. Os tipos de literais de template nos permitem trazer essas restrições para o nosso código.

```
const person = makeWatchedObject({
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26,
});

// makeWatchedObject adicionou `on` ao objeto anônimo

person.on("firstNameChanged", (newValue) => {
  console.log(`firstName foi mudado para ${newValue}!`);
});
```

Observe que `on` escuta o evento `"firstNameChanged"`, não apenas `"firstName"`. Nossa especificação ingênua de `on()` poderia ser tornada mais robusta se assegurássemos que o conjunto de nomes de eventos elegíveis fosse restringido pela união de nomes de atributos no objeto observado com "Changed" adicionado ao final. Embora estejamos confortáveis em fazer tal cálculo em JavaScript, ou seja, `Object.keys(passedObject).map(x => `${x}Changed`)`, literais de template dentro do sistema de tipos fornecem uma abordagem semelhante para manipulação de strings:

```
type PropEventSource<Type> = {
    on(eventName: `${string & keyof Type}Changed`, callback: (newValue: any) =>
void): void;
};

/// Crie um "objeto observado" com um método `on`
/// para que você possa observar mudanças nas propriedades.
declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;
```

Com isso, podemos construir algo que gera erro ao receber a propriedade errada:

```
const person = makeWatchedObject({
  firstName: "Saoirse",
```

```
  lastName: "Ronan",
  age: 26
});

person.on("firstNameChanged", () => {});

// Previne erro humano fácil (usando a chave em vez do nome do evento)
person.on("firstName", () => {});
Argument of type '"firstName"' is not assignable to parameter of type
'"firstNameChanged" | "lastNameChanged" | "ageChanged"'.

// É resistente a erros de digitação
person.on("frstNameChanged", () => {});
Argument of type '"frstNameChanged"' is not assignable to parameter of type
'"firstNameChanged" | "lastNameChanged" | "ageChanged"'.
```

Inferência com Literais de Template

Note que não nos beneficiamos de todas as informações fornecidas no objeto passado original. Dada a mudança de um `firstName` (ou seja, um evento `firstNameChanged`), deveríamos esperar que o callback recebesse um argumento do tipo `string`. Da mesma forma, o callback para uma mudança na `age` deve receber um argumento do tipo `number`. Estamos usando ingênuamente `any` para tipar o argumento do `callback`. Novamente, os tipos de literais de template tornam possível garantir que o tipo de dado de um atributo será o mesmo tipo que o primeiro argumento do callback desse atributo.

A percepção chave que torna isso possível é a seguinte: podemos usar uma função com um genérico de modo que:

1. O literal usado no primeiro argumento é capturado como um tipo literal.
2. Esse tipo literal pode ser validado como pertencente à união de atributos válidos no genérico.
3. O tipo do atributo validado pode ser pesquisado na estrutura do genérico usando Acesso Indexado.
4. Essa informação de tipagem pode então ser aplicada para garantir que o argumento da função callback seja do mesmo tipo.

```
type PropEventSource<Type> = {
    on<Key extends string & keyof Type>
        (eventName: `${Key}Changed`, callback: (newValue: Type[Key]) => void): void;
};

declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;

const person = makeWatchedObject({
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26
});

person.on("firstNameChanged", newName => {

(parameter) newName: string
    console.log(`novo nome é ${newName.toUpperCase()}`);
```

```
});

person.on("ageChanged", newAge => {

(parameter) newAge: number
    if (newAge < 0) {
        console.warn("aviso! idade negativa");
    }
})
```

Aqui, transformamos `on` em um método genérico.

Quando um usuário chama com a string `"firstNameChanged"`, o TypeScript tentará inferir o tipo correto para `Key`. Para fazer isso, ele irá combinar `Key` com o conteúdo antes de `"Changed"` e inferir a string `"firstName"`. Assim que o TypeScript descobre isso, o método `on` pode buscar o tipo de `firstName` no objeto original, que é `string` neste caso. Da mesma forma, quando chamado com `"ageChanged"`, o TypeScript encontra o tipo para a propriedade `age`, que é `number`.

A inferência pode ser combinada de diferentes maneiras, frequentemente para deconstruir strings e reconstruí-las de maneiras diferentes.

Tipos Intrínsecos de Manipulação de Strings

Para ajudar com a manipulação de strings, o TypeScript inclui um conjunto de tipos que podem ser usados na manipulação de strings. Esses tipos vêm embutidos no compilador para desempenho e não podem ser encontrados nos arquivos .d.ts incluídos com o TypeScript.

`Uppercase<StringType>`

Converte cada caractere na string para a versão em maiúsculas.

Exemplo

```
type Greeting = "Hello, world"
type ShoutyGreeting = Uppercase<Greeting>

type ShoutyGreeting = "HELLO, WORLD"

type ASCIICacheKey<Str extends string> = `ID-${Uppercase<Str>}`
type MainID = ASCIICacheKey<"my_app">

type MainID = "ID-MY_APP"
```

`Lowercase<StringType>`

Converte cada caractere na string para o equivalente em minúsculas.

Exemplo

```
type Greeting = "Hello, world"
type QuietGreeting = Lowercase<Greeting>

type QuietGreeting = "hello, world"

type ASCIICacheKey<Str extends string> = `id-${Lowercase<Str>}`
type MainID = ASCIICacheKey<"MY_APP">
```

```
type MainID = "id-my_app"
```

`Capitalize<StringType>`

Converte o primeiro caractere da string para um equivalente em maiúsculas.

Exemplo

```
type LowercaseGreeting = "hello, world";
type Greeting = Capitalize<LowercaseGreeting>;

type Greeting = "Hello, world"
```

`Uncapitalize<StringType>`

Converte o primeiro caractere da string para um equivalente em minúsculas.

Exemplo

```
type UppercaseGreeting = "HELLO WORLD";
type UncomfortableGreeting = Uncapitalize<UppercaseGreeting>;

type UncomfortableGreeting = "hELLO WORLD"
```