TypeScript 2.1 现已发布。现在下载 (/#download-links)我们最新版本！

文档目录 ▾

# JSX

## 介绍

JSX (https://facebook.github.io/jsx/)是一种嵌入式的类似XML的语法。 它可以被转换成合法的JavaScript，尽管转换的语义是依据不同的实现而定的。 JSX因 React (http://facebook.github.io/react/)框架而流行，但是也被其它应用所使用。 TypeScript支持内嵌，类型检查和将JSX直接编译为JavaScript。

## 基本用法

想要使用JSX必须做两件事：

1. 给文件一个 `.tsx` 扩展名
2. 启用 `jsx` 选项

TypeScript具有两种JSX模式： `preserve` 和 `react` 。 这些模式只在代码生成阶段起作用 - 类型检查并不受影响。 在 `preserve` 模式下生成代码中会保留JSX以供后续的转换操作使用（比如：Babel (https://babeljs.io/)）。 另外，输出文件会带有 `.jsx` 扩展名。 `react` 模式会生成 `React.createElement` ，在使用前不需要再进行转换操作了，输出文件的扩展名为 `.js` 。

| 模式 | 输入 | 输出 | 输出文件扩展名 |
|---|---|---|---|
| `preserve` | `<div />` | `<div />` | `.jsx` |
| `react` | `<div />` | `React.createElement("div")` | `.js` |

你可以通过在命令行里使用 `--jsx` 标记或tsconfig.json (./tsconfig-json.html)里的选项来指定模式。

> *注意： `React` 标识符是写死的硬代码，所以你必须保证React（大写的R）是可用的。 Note: The identifier `React` is hard-coded, so you must make React available with an uppercase R.*

## as 操作符

回想一下怎么写类型断言：

```
var foo = <foo>bar;
```

这里我们断言 `bar` 变量是 `foo` 类型的。 因为TypeScript也使用尖括号来表示类型断言，JSX的语法带来了解析的困难。因此，TypeScript在 `.tsx` 文件里禁用了使用尖括号的类型断言。

为了弥补 `.tsx` 里的这个功能，新加入了一个类型断言符号： `as` 。 上面的例子可以很容易地使用 `as` 操作符改写：

```
var foo = bar as foo;
```

`as` 操作符在 `.ts` 和 `.tsx` 里都可用，并且与其它类型断言行为是等价的。

## 类型检查

为了理解JSX的类型检查，你必须首先理解固有元素与基于值的元素之间的区别。 假设有这样一个JSX表达式 `<expr />` ， `expr` 可能引用环境自带的某些东西（比如，在DOM环境里的 `div` 或 `span` ）或者是你自定义的组件。 这是非常重要的，原因有如下两点：

1. 对于React，固有元素会生成字符串（ `React.createElement("div")` ），然而由你自定义的组件却不会生成（ `React.createElement(MyComponent)` ）。
2. 传入JSX元素里的属性类型的查找方式不同。 固有元素属性 *本身*就支持，然而自定义的组件会自己去指定它们具有哪个属性。

TypeScript使用与React相同的规范 (http://facebook.github.io/react/docs/jsx-in-depth.html#html-tags-vs.-react-components) 来区别它们。 固有元素总是以一个小写字母开头，基于值的元素总是以一个大写字母开头。

### 固有元素

固有元素使用特殊的接口 `JSX.IntrinsicElements` 来查找。 默认地，如果这个接口没有指定，会全部通过，不对固有元素进行类型检查。 然而，如果接口存在，那么固有元素的名字需要在 `JSX.IntrinsicElements` 接口的属性里查找。 例如：

```
declare namespace JSX {
    interface IntrinsicElements {
        foo: any
    }
}

<foo />; // 正确
<bar />; // 错误
```

在上例中， `<foo />` 没有问题，但是 `<bar />` 会报错，因为它没在 `JSX.IntrinsicElements` 里指定。

> 注意：你也可以在 `JSX.IntrinsicElements` 上指定一个用来捕获所有字符串索引：
>
> ```
> declare namespace JSX {
>    interface IntrinsicElements {
>        [elemName: string]: any;
>    }
> }
> ```

### 基于值的元素

基于值的元素会简单的在它所在的作用域里按标识符查找。

```
import MyComponent from "./myComponent";

<MyComponent />; // 正确
<SomeOtherComponent />; // 错误
```

可以限制基于值的元素的类型。 然而，为了这么做我们需要引入两个新的术语： *元素类的类型*和*元素实例的类型*。

现在有 `<Expr />` ，*元素类的类型*为 `Expr` 的类型。 所以在上面的例子里，如果 `MyComponent` 是ES6的类，那么它的类类型就是这个类。 如果 `MyComponent` 是个工厂函数，类类型为这个函数。

一旦建立起了类类型，实例类型就确定了，为类类型调用签名的返回值与构造签名的联合类型。 再次说明，在ES6类的情况下，实例类型为这个类的实例的类型，并且如果是工厂函数，实例类型为这个函数返回值类型。

```
class MyComponent {
  render() {}
}

// 使用构造签名
var myComponent = new MyComponent();

// 元素类的类型 => MyComponent
// 元素实例的类型 => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {
    }
  }
}

// 使用调用签名
var myComponent = MyFactoryFunction();

// 元素类的类型 => FactoryFunction
// 元素实例的类型 => { render: () => void }
```

元素的实例类型很有趣，因为它必须赋值给 `JSX.ElementClass` 或抛出一个错误。 默认的 `JSX.ElementClass` 为 `{}` ，但是它可以被扩展用来限制JSX的类型以符合相应的接口。

```
declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} }
}

<MyComponent />; // 正确
<MyFactoryFunction />; // 正确

class NotAValidComponent {}
function NotAValidFactoryFunction() {
  return {};
}

<NotAValidComponent />; // 错误
<NotAValidFactoryFunction />; // 错误
```

### 属性类型检查

属性类型检查的第一步是确定*元素属性类型*。 这在固有元素和基于值的元素之间稍有不同。

对于固有元素，这是 `JSX.IntrinsicElements` 属性的类型。

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { bar?: boolean }
  }
}

// `foo`的元素属性类型为`{bar?: boolean}`
<foo bar />;
```

对于基于值的元素，就稍微复杂些。 它取决于先前确定的在元素实例类型上的某个属性的类型。 至于该使用哪个属性来确定类型取决于 `JSX.ElementAttributesProperty` 。 它应该使用单一的属性来定义。 这个属性名之后会被使用。

```
declare namespace JSX {
  interface ElementAttributesProperty {
    props; // 指定用来使用的属性名
  }
}

class MyComponent {
  // 在元素实例类型上指定属性
  props: {
    foo?: string;
  }
}

// `MyComponent`的元素属性类型为`{foo?: string}`
<MyComponent foo="bar" />
```

元素属性类型用于的JSX里进行属性的类型检查。 支持可选属性和必须属性。

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: { requiredProp: string; optionalProp?: number }
  }
}

<foo requiredProp="bar" />; // 正确
<foo requiredProp="bar" optionalProp={0} />; // 正确
<foo />; // 错误, 缺少 requiredProp
<foo requiredProp={0} />; // 错误, requiredProp 应该是字符串
<foo requiredProp="bar" unknownProp />; // 错误, unknownProp 不存
在
<foo requiredProp="bar" some-unknown-prop />; // 正确, `some-unk
nown-prop`不是个合法的标识符
```

> 注意：如果一个属性名不是个合法的JS标识符（像 `data-*` 属性），并且它没出现在元素属性类型里时不会当做一个错误。

延展操作符也可以使用：

```
var props = { requiredProp: 'bar' };
<foo {...props} />; // 正确

var badProps = {};
<foo {...badProps} />; // 错误
```

## JSX结果类型

默认地JSX表达式结果的类型为 `any`。 你可以自定义这个类型，通过指定 JSX.Element`接口。 然而，不能够从接口里检索元素，属性或JSX的子元素的类型信息。 它是一个黑盒。

## 嵌入的表达式

JSX允许你使用 `{ }` 标签来内嵌表达式。

```
var a = <div>
  {['foo', 'bar'].map(i => <span>{i / 2}</span>)}
</div>
```

上面的代码产生一个错误，因为你不能用数字来除以一个字符串。 输出如下，若你使用了 `preserve` 选项：

```
var a = <div>
  {['foo', 'bar'].map(function (i) { return <span>{i / 2}</span
>; })}
</div>
```

## React整合

要想一起使用JSX和React，你应该使用React类型定义 (https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/react)。 这些类型声明定义了 `JSX` 合适命名空间来使用React。

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>
  }
}

<MyComponent foo="bar" />; // 正确
<MyComponent foo={0} />; // 错误
```

制作 ♥ 于北京 微博 @TypeScript中文网 (http://weibo.com/tslangcn) Star

TypeScript中文网 | 推动TypeScript语言在中国的发展

TypeScript中文网基本会保持和TypeScript官网 (http://www.typescriptlang.org/)一致