介绍

函数是JavaScript应用程序的基础。
它帮助你实现抽象层，模拟类，信息隐藏和模块。
在TypeScript里，虽然已经支持类，命名空间和模块，但函数仍然是主要的定义*行为*的地方。
TypeScript为JavaScript函数添加了额外的功能，让我们可以更容易地使用。

函数

和JavaScript一样，TypeScript函数可以创建有名字的函数和匿名函数。
你可以随意选择适合应用程序的方式，不论是定义一系列API函数还是只使用一次的函数。

通过下面的例子可以迅速回想起这两种JavaScript中的函数：

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

在JavaScript里，函数可以使用函数体外部的变量。
当函数这么做时，我们说它‘捕获’了这些变量。
至于为什么可以这样做以及其中的利弊超出了本文的范围，但是深刻理解这个机制对学习JavaScript和TypeScript会很有帮助。

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

函数类型

为函数定义类型

让我们为上面那个函数添加类型：

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x+y; };
```

我们可以给每个参数添加类型之后再为函数本身添加返回值类型。

TypeScript能够根据返回语句自动推断出返回值类型，因此我们通常省略它。

书写完整函数类型

现在我们已经为函数指定了类型，下面让我们写出函数的完整类型。

```
let myAdd: (x:number, y:number)=>number =
    function(x: number, y: number): number { return x+y; };
```

函数类型包含两部分：参数类型和返回值类型。
当写出完整函数类型的时候，这两部分都是需要的。
我们以参数列表的形式写出参数类型，为每个参数指定一个名字和类型。
这个名字只是为了增加可读性。
我们也可以这么写：

```
let myAdd: (baseValue:number, increment:number) => number =
    function(x: number, y: number): number { return x + y; };
```

只要参数类型是匹配的，那么就认为它是有效的函数类型，而不在乎参数名是否正确。

第二部分是返回值类型。
对于返回值，我们在函数和返回值类型之前使用(=>)符号，使之清晰明了。
如之前提到的，返回值类型是函数类型的必要部分，如果函数没有返回任何值，你也必须指定返回值类型为`void`而不能留空。

函数的类型只是由参数类型和返回值组成的。
函数中使用的捕获变量不会体现在类型里。
实际上，这些变量是函数的隐藏状态并不是组成API的一部分。

推断类型

尝试这个例子的时候，你会发现如果你在赋值语句的一边指定了类型但是另一边没有类型的话，TypeScript编译器会自动识别出类型：

```
// myAdd has the full function type
let myAdd = function(x: number, y: number): number { return x + y; };

// The parameters `x` and `y` have the type number
let myAdd: (baseValue:number, increment:number) => number =
    function(x, y) { return x + y; };
```

这叫做“按上下文归类”，是类型推论的一种。
它帮助我们更好地为程序指定类型。

可选参数和默认参数

TypeScript里的每个函数参数都是必须的。
这不是指不能传递`null`或`undefined`作为参数，而是说编译器检查用户是否为每个参数都传入了值。
编译器还会假设只有这些参数会被传递进函数。

简短地说，传递给一个函数的参数个数必须与函数期望的参数个数一致。

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // ah, just right
```

JavaScript里，每个参数都是可选的，可传可不传。
没传参的时候，它的值就是undefined。
在TypeScript里我们可以在参数名旁使用?实现可选参数的功能。
比如，我们想让last name是可选的：

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob");  // works correctly now
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");  // ah, just right
```

可选参数必须跟在必须参数后面。
如果上例我们想让first name是可选的，那么就必须调整它们的位置，把first name放在后面。

在TypeScript里，我们也可以为参数提供一个默认值当用户没有传递这个参数或传递的值是`undefined`时。
它们叫做有默认初始化值的参数。
让我们修改上例，把last name的默认值设置为`"Smith"`。

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // works correctly now, return
let result2 = buildName("Bob", undefined);       // still works, also returns "
let result3 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result4 = buildName("Bob", "Adams");         // ah, just right
```

在所有必须参数后面的带默认初始化的参数都是可选的，与可选参数一样，在调用函数的时候可以省略。
也就是说可选参数与末尾的默认参数共享参数类型。

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

和

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
```

```
}
```

共享同样的类型`(firstName: string, lastName?: string) => string`。
默认参数的默认值消失了，只保留了它是一个可选参数的信息。

与普通可选参数不同的是，带默认值的参数不需要放在必须参数的后面。
如果带默认值的参数出现在必须参数前面，用户必须明确的传入`undefined`值来获得默认值。
例如，我们重写最后一个例子，让`firstName`是带默认值的参数：

```
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // okay and returns "Bob Adams
let result4 = buildName(undefined, "Adams");     // okay and returns "Will Adam
```

剩余参数

必要参数，默认参数和可选参数有个共同点：它们表示某一个参数。
有时，你想同时操作多个参数，或者你并不知道会有多少参数传递进来。
在JavaScript里，你可以使用`arguments`来访问所有传入的参数。

在TypeScript里，你可以把所有参数收集到一个变量里：

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

剩余参数会被当做个数不限的可选参数。
可以一个都没有，同样也可以有任意个。
编译器创建参数数组，名字是你在省略号（...）后面给定的名字，你可以在函数体内使用这个数组。

这个省略号也会在带有剩余参数的函数类型定义上使用到：

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

this

学习使用JavaScript里`this`就好比一场成年礼。
由于TypeScript是JavaScript的超集，TypeScript程序员也需要弄清`this`工作机制并且当有bug的时候能够找出错误所在。

幸运的是，TypeScript能通知你错误地使用了this的地方。
如果你想了解JavaScript里的this是如何工作的，那么首先阅读Yehuda Katz写的
Understanding JavaScript Function Invocation and "this"。
Yehuda的文章详细的阐述了this的内部工作原理，因此我们这里只做简单介绍。

this和箭头函数

JavaScript里，this的值在函数被调用的时候才会指定。
这是个既强大又灵活的特点，但是你需要花点时间弄清楚函数调用的上下文是什么。
但众所周知，这不是一件很简单的事，尤其是在返回一个函数或将函数当做参数传递的时候。

下面看一个例子：

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        return function() {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

可以看到createCardPicker是个函数，并且它又返回了一个函数。
如果我们尝试运行这个程序，会发现它并没有弹出对话框而是报错了。
因为createCardPicker返回的函数里的this被设置成了window而不是deck对象。
因为我们只是独立的调用了cardPicker()。
顶级的非方法式调用会将this视为window。
（注意：在严格模式下，this为undefined而不是window）。

为了解决这个问题，我们可以在函数被返回时就绑好正确的this。
这样的话，无论之后怎么使用它，都会引用绑定的‘deck’对象。
我们需要改变函数表达式来使用ECMAScript 6箭头语法。
箭头函数能保存函数创建时的this值，而不是调用时的值：

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // NOTE: the line below is now an arrow function, allowing us to captur
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
```

```
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

更好事情是，TypeScript会警告你犯了一个错误，如果你给编译器设置了`--noImplicitThis`标记。
它会指出`this.suits[pickedSuit]`里的`this`的类型为`any`。

this参数

不幸的是，`this.suits[pickedSuit]`的类型依旧为`any`。
这是因为`this`来自对象字面量里的函数表达式。
修改的方法是，提供一个显式的`this`参数。
`this`参数是个假的参数，它出现在参数列表的最前面：

```
function f(this: void) {
    // make sure `this` is unusable in this standalone function
}
```

让我们往例子里添加一些接口，`Card` 和 `Deck`，让类型重用能够变得清晰简单些：

```
interface Card {
    suit: string;
    card: number;
}
interface Deck {
    suits: string[];
    cards: number[];
    createCardPicker(this: Deck): () => Card;
}
let deck: Deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    // NOTE: The function now explicitly specifies that its callee must be of t
    createCardPicker: function(this: Deck) {
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

现在TypeScript知道`createCardPicker`期望在某个`Deck`对象上调用。
也就是说`this`是`Deck`类型的，而非`any`，因此`--noImplicitThis`不会报错了。

this参数在回调函数里

你可以也看到过在回调函数里的this报错，当你将一个函数传递到某个库函数里稍后会被调用时。
因为当回调被调用的时候，它们会被当成一个普通函数调用，this将为undefined。
稍做改动，你就可以通过this参数来避免错误。
首先，库函数的作者要指定this的类型：

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type.
Second, annotate your calling code with `this`:

```
class Handler {
    info: string;
    onClickBad(this: Handler, e: Event) {
        // oops, used this here. using this callback would crash at runtime
        this.info = e.message;
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // error!
```

指定了this类型后，你显式声明onClickBad必须在Handler的实例上调用。
然后TypeScript会检测到addClickListener要求函数带有this: void。
改变this类型来修复这个错误：

```
class Handler {
    info: string;
    onClickGood(this: void, e: Event) {
        // can't use this here because it's of type void!
        console.log('clicked!');
    }
}
let h = new Handler();
uiElement.addClickListener(h.onClickGood);
```

因为onClickGood指定了this类型为void，因此传递addClickListener是合法的。
当然了，这也意味着不能使用this.info.
如果你两者都想要，你不得不使用箭头函数了：

```
class Handler {
    info: string;
    onClickGood = (e: Event) => { this.info = e.message }
}
```

这是可行的因为箭头函数不会捕获this，所以你总是可以把它们传给期望this: void的函数。
缺点是每个Handler对象都会创建一个箭头函数。
另一方面，方法只会被创建一次，添加到Handler的原型链上。
它们在不同Handler对象间是共享的。

重载

JavaScript本身是个动态语言。
JavaScript里函数根据传入不同的参数而返回不同类型的数据是很常见的。

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { su
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

`pickCard`方法根据传入参数的不同会返回两种不同的类型。
如果传入的是代表纸牌的对象，函数作用是从中抓一张牌。
如果用户想抓牌，我们告诉他抓到了什么牌。
但是这怎么在类型系统里表示呢。

方法是为同一个函数提供多个函数类型定义来进行函数重载。
编译器会根据这个列表去处理函数的调用。
下面我们来重载`pickCard`函数。

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    // if so, they gave us the deck and we'll pick the card
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    // Otherwise just let them pick the card
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { su
```

```
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

这样改变后，重载的`pickCard`函数在调用的时候会进行正确的类型检查。

为了让编译器能够选择正确的检查类型，它与JavaScript里的处理流程相似。
它查找重载列表，尝试使用第一个重载定义。
如果匹配的话就使用这个。
因此，在定义重载的时候，一定要把最精确的定义放在最前面。

注意，`function pickCard(x): any`并不是重载列表的一部分，因此这里只有两个重载：
一个是接收对象另一个接收数字。
以其它参数调用`pickCard`会产生错误。