介绍

软件工程中，我们不仅要创建一致的定义良好的API，同时也要考虑可重用性。
组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

在像C#和Java这样的语言中，可以使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。
这样用户就可以以自己的数据类型来使用组件。

泛型之Hello World

下面来创建第一个使用泛型的例子：identity函数。
这个函数会返回任何传入它的值。
你可以把这个函数当成是`echo`命令。

不用泛型的话，这个函数可能是下面这样：

```
function identity(arg: number): number {
    return arg;
}
```

或者，我们使用`any`类型来定义函数：

```
function identity(arg: any): any {
    return arg;
}
```

虽然使用`any`类型后这个函数已经能接收任何类型的arg参数，但是却丢失了一些信息：传入的类型与返回的类型应该是相同的。
如果我们传入一个数字，我们只知道任何类型的值都有可能被返回。

因此，我们需要一种方法使返回值的类型与传入参数的类型是相同的。
这里，我们使用了*类型变量*，它是一种特殊的变量，只用于表示类型而不是值。

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

我们给identity添加了类型变量`T`。
`T`帮助我们捕获用户传入的类型（比如：`number`），之后我们就可以使用这个类型。
之后我们再次使用了`T`当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。
这允许我们跟踪函数里使用的类型的信息。

我们把这个版本的`identity`函数叫做泛型，因为它可以适用于多个类型。
不同于使用`any`，它不会丢失信息，像第一个例子那像保持准确性，传入数值类型并返回数值类型。

我们定义了泛型函数后，可以用两种方法使用。
第一种是，传入所有的参数，包含类型参数：

```
let output = identity<string>("myString");  // type of output will be 'string'
```

这里我们明确的指定了`T`是`string`类型，并做为一个参数传给函数，使用了`<>`括起来而不是`()`。

第二种方法更普遍。利用了*类型推论* -- 即编译器会根据传入的参数自动地帮助我们确定T的类型：

```
let output = identity("myString");  // type of output will be 'string'
```

注意我们没必要使用尖括号（`<>`）来明确地传入类型；编译器可以查看`myString`的值，然后把`T`设置为它的类型。
类型推论帮助我们保持代码精简和高可读性。如果编译器不能够自动地推断出类型的话，只能像上面那样明确的传入T的类型，在一些复杂的情况下，这是可能出现的。

使用泛型变量

使用泛型创建像`identity`这样的泛型函数时，编译器要求你在函数体必须正确的使用这个通用的类型。
换句话说，你必须把这些参数当做是任意或所有类型。

看下之前`identity`例子：

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

如果我们想同时打印出`arg`的长度。
我们很可能会这样做：

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

如果这么做，编译器会报错说我们使用了`arg`的`.length`属性，但是没有地方指明`arg`具有这个属性。
记住，这些类型变量代表的是任意类型，所以使用这个函数的人可能传入的是个数字，而数字是没有`.length`属性的。

现在假设我们想操作`T`类型的数组而不直接是`T`。由于我们操作的是数组，所以`.length`属性是应该存在的。
我们可以像创建其它数组一样创建这个数组：

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

你可以这样理解`loggingIdentity`的类型：泛型函数`loggingIdentity`，接收类型参数`T`，和函数`arg`，它是个元素类型是`T`的数组，并返回元素类型是`T`的数组。
如果我们传入数字数组，将返回一个数字数组，因为此时`T`的的类型为`number`。
这可以让我们把泛型变量T当做类型的一部分使用，而不是整个类型，增加了灵活性。

我们也可以这样实现上面的例子：

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

使用过其它语言的话，你可能对这种语法已经很熟悉了。
在下一节，会介绍如何创建自定义泛型像`Array<T>`一样。

泛型类型

上一节，我们创建了identity通用函数，可以适用于不同的类型。
在这节，我们研究一下函数本身的类型，以及如何创建泛型接口。

泛型函数的类型与非泛型函数的类型没什么不同，只是有一个类型参数在最前面，像函数声明一样：

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

let myIdentity: <T>(arg: T) => T = identity;
```

我们也可以使用不同的泛型参数名，只要在数量上和使用方式上能对应上就可以。

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

let myIdentity: <U>(arg: U) => U = identity;
```

我们还可以使用带有调用签名的对象字面量来定义泛型函数：

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

let myIdentity: {<T>(arg: T): T} = identity;
```

这引导我们去写第一个泛型接口了。
我们把上面例子里的对象字面量拿出来做为一个接口：

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}

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

```
}

let myIdentity: GenericIdentityFn = identity;
```

一个相似的例子，我们可能想把泛型参数当作整个接口的一个参数。
这样我们就能清楚的知道使用的具体是哪个泛型类型（比如：`Dictionary<string>`而不只是`Dictionary`）。
这样接口里的其它成员也能知道这个参数的类型了。

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}

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

let myIdentity: GenericIdentityFn<number> = identity;
```

注意，我们的示例做了少许改动。
不再描述泛型函数，而是把非泛型函数签名作为泛型类型一部分。
当我们使用`GenericIdentityFn`的时候，还得传入一个类型参数来指定泛型类型（这里是：`number`），锁定了之后代码里使用的类型。
对于描述哪部分类型属于泛型部分来说，理解何时把参数放在调用签名里和何时放在接口上是很有帮助的。

除了泛型接口，我们还可以创建泛型类。
注意，无法创建泛型枚举和泛型命名空间。

泛型类

泛型类看上去与泛型接口差不多。
泛型类使用（`<>`）括起泛型类型，跟在类名后面。

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

`GenericNumber`类的使用是十分直观的，并且你可能已经注意到了，没有什么去限制它只能使用`number`类型。
也可以使用字符串或其它更复杂的类型。

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

与接口一样，直接把泛型类型放在类后面，可以帮助我们确认类的所有属性都在使用相同

的类型。

我们在类那节说过，类有两部分：静态部分和实例部分。
泛型类指的是实例部分的类型，所以类的静态属性不能使用这个泛型类型。

泛型约束

你应该会记得之前的一个例子，我们有时候想操作某类型的一组值，并且我们知道这组值具有什么样的属性。
在`loggingIdentity`例子中，我们想访问`arg`的`length`属性，但是编译器并不能证明每种类型都有`length`属性，所以就报错了。

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

相比于操作any所有类型，我们想要限制函数去处理任意带有`.length`属性的所有类型。
只要传入的类型有这个属性，我们就允许，就是说至少包含这一属性。
为此，我们需要列出对于T的约束要求。

为此，我们定义一个接口来描述约束条件。
创建一个包含`.length`属性的接口，使用这个接口和`extends`关键字还实现约束：

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no m
    return arg;
}
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3);  // Error, number doesn't have a .length property
```

我们需要传入符合约束类型的值，必须包含必须的属性：

```
loggingIdentity({length: 10, value: 3});
```

在泛型约束中使用类型参数

你可以声明一个类型参数，且它被另一个类型参数所约束。
比如，现在我们想要用属性名从对象里获取这个属性。
并且我们想要确保这个属性存在于对象`obj`上，因此我们需要在这两个类型之间使用约束。

```
function copyFields<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = source[id];
    }
```

```
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a"); // okay
getProperty(x, "m"); // error: Argument of type 'm' isn't assignable to 'a' | '
```

在泛型里使用类类型

在TypeScript使用泛型创建工厂函数时，需要引用构造函数的类类型。比如，

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

一个更高级的例子，使用原型属性推断并约束构造函数与类实例的关系。

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function findKeeper<A extends Animal, K> (a: {new(): A;
    prototype: {keeper: K}}): K {

    return a.prototype.keeper;
}

findKeeper(Lion).nametag;  // typechecks!
```