交叉类型（Intersection Types）

交叉类型是将多个类型合并为一个类型。
这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。
例如，`Person & Serializable & Loggable`同时是`Person`*和*`Serializable`*和*`Loggable`。
就是说这个类型的对象同时拥有了这三种类型的成员。

我们大多是在混入（mixins）或其它不适合典型面向对象模型的地方看到交叉类型的使用。
（在JavaScript里发生这种情况的场合很多！）
下面是如何创建混入的一个简单例子：

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

联合类型（Union Types）

联合类型与交叉类型很有关联，但是使用上却完全不同。
偶尔你会遇到这种情况，一个代码库希望传入`number`或`string`类型的参数。
例如下面的函数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left si
 */
```

```
function padLeft(value: string, padding: any) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

padLeft存在一个问题，padding参数的类型指定成了any。
这就是说我们可以传入一个既不是number也不是string类型的参数，但是TypeScript却不报错。

```
let indentedString = padLeft("Hello world", true); // 编译阶段通过，运行时报错
```

在传统的面向对象语言里，我们可能会将这两种类型抽象成有层级的类型。
这么做显然是非常清晰的，但同时也存在了过度设计。
padLeft原始版本的好处之一是允许我们传入原始类型。
这样做的话使用起来既简单又方便。
如果我们就是想使用已经存在的函数的话，这种新的方式就不适用了。

代替any， 我们可以使用*联合类型*做为padding的参数：

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left si
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

联合类型表示一个值可以是几种类型之一。
我们用竖线（|）分隔每个类型，所以number | string | boolean表示一个值可以是number，string，或boolean。

如果一个值是联合类型，我们只能访问此联合类型的所有类型里共有的成员。

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}
```

```
let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();    // errors
```

这里的联合类型可能有点复杂，但是你很容易就习惯了。
如果一个值的类型是`A | B`，我们能够*确定*的是它包含了`A`*和*`B`中共有的成员。
这个例子里，`Bird`具有一个`fly`成员。
我们不能确定一个`Bird | Fish`类型的变量是否有`fly`方法。
如果变量在运行时是`Fish`类型，那么调用`pet.fly()`就出错了。

类型保护与区分类型（Type Guards and Differentiating Types）

联合类型适合于那些值可以为不同类型的情况。
但当我们想确切地了解是否为`Fish`时怎么办？
JavaScript里常用来区分2个可能值的方法是检查成员是否存在。
如之前提及的，我们只能访问联合类型中共同拥有的成员。

```
let pet = getSmallPet();

// 每一个成员访问都会报错
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}
```

为了让这段代码工作，我们要使用类型断言：

```
let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}
```

用户自定义的类型保护

这里可以注意到我们不得不多次使用类型断言。
假若我们一旦检查过类型，就能在之后的每个分支里清楚地知道`pet`的类型的话就好了。

TypeScript里的*类型保护*机制让它成为了现实。
类型保护就是一些表达式，它们会在运行时检查以确保在某个作用域里的类型。
要定义一个类型保护，我们只要简单地定义一个函数，它的返回值是一个*类型谓词*：

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}
```

在这个例子里，`pet is Fish`就是类型谓词。
谓词为`parameterName is Type`这种形式，`parameterName`必须是来自于当前函数签名里的一个参数名。

每当使用一些变量调用`isFish`时，TypeScript会将变量缩减为那个具体的类型，只要这个类型与变量的原始类型是兼容的。

```
// 'swim' 和 'fly' 调用都没有问题了

if (isFish(pet)) {
    pet.swim();
}
else {
    pet.fly();
}
```

注意TypeScript不仅知道在`if`分支里`pet`是`Fish`类型；
它还清楚在`else`分支里，一定*不是*`Fish`类型，一定是`Bird`类型。

typeof类型保护

现在我们回过头来看看怎么使用联合类型书写`padLeft`代码。
我们可以像下面这样利用类型断言来写：

```
function isNumber(x: any): x is number {
    return typeof x === "number";
}

function isString(x: any): x is string {
    return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
    if (isNumber(padding)) {
        return Array(padding + 1).join(" ") + value;
    }
    if (isString(padding)) {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

然而，必须要定义一个函数来判断类型是否是原始类型，这太痛苦了。
幸运的是，现在我们不必将`typeof x === "number"`抽象成一个函数，因为TypeScript可以将它识别为一个类型保护。
也就是说我们可以直接在代码里检查类型了。

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

这些*typeof类型保护*只有两种形式能被识别：`typeof v === "typename"`和`typeof v !== "typename"`，`"typename"`必须是`"number"`，`"string"`，`"boolean"`或`"symbol"`。
但是TypeScript并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

instanceof类型保护

如果你已经阅读了`typeof`类型保护并且对JavaScript里的`instanceof`操作符熟悉的话，你可能已经猜到了这节要讲的内容。

*`instanceof`类型保护*是通过构造函数来细化类型的一种方式。
比如，我们借鉴一下之前字符串填充的例子：

```
interface Padder {
    getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
    constructor(private numSpaces: number) { }
    getPaddingString() {
        return Array(this.numSpaces + 1).join(" ");
    }
}

class StringPadder implements Padder {
    constructor(private value: string) { }
    getPaddingString() {
        return this.value;
    }
}

function getRandomPadder() {
    return Math.random() < 0.5 ?
        new SpaceRepeatingPadder(4) :
        new StringPadder("  ");
}

// 类型为SpaceRepeatingPadder | StringPadder
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
    padder; // 类型细化为'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
    padder; // 类型细化为'StringPadder'
}
```

`instanceof`的右侧要求是一个构造函数，TypeScript将细化为：

1. 此构造函数的`prototype`属性的类型，如果它的类型不为`any`的话
2. 构造签名所返回的类型的联合

以此顺序。

可以为null的类型

TypeScript具有两种特殊的类型，`null`和`undefined`，它们分别具有值null和undefined.
我们在[基础类型](./Basic Types.md)一节里已经做过简要说明。
默认情况下，类型检查器认为`null`与`undefined`可以赋值给任何类型。
`null`与`undefined`是所有其它类型的一个有效值。
这也意味着，你阻止不了将它们赋值给其它类型，就算是你想要阻止这种情况也不行。
`null`的发明者，Tony Hoare，称它为[价值亿万美金的错误](https://en.wikipedia.org/wiki/Null_pointer#History)。

`--strictNullChecks`标记可以解决此错误：当你声明一个变量时，它不会自动地包含`null`或`undefined`。
你可以使用联合类型明确的包含它们：

```
let s = "foo";
s = null; // 错误, 'null'不能赋值给'string'
let sn: string | null = "bar";
sn = null; // 可以

sn = undefined; // error, 'undefined'不能赋值给'string | null'
```

注意，按照JavaScript的语义，TypeScript会把`null`和`undefined`区别对待。
`string | null`，`string | undefined`和`string | undefined | null`是不同的类型。

可选参数和可选属性

使用了`--strictNullChecks`，可选参数会被自动地加上`| undefined`:

```
function f(x: number, y?: number) {
    return x + (y || 0);
}
f(1, 2);
f(1);
f(1, undefined);
f(1, null); // error, 'null' is not assignable to 'number | undefined'
```

可选属性也会有同样的处理：

```
class C {
    a: number;
    b?: number;
}
let c = new C();
c.a = 12;
c.a = undefined; // error, 'undefined' is not assignable to 'number'
c.b = 13;
c.b = undefined; // ok
c.b = null; // error, 'null' is not assignable to 'number | undefined'
```

类型保护和类型断言

由于可以为null的类型是通过联合类型实现，那么你需要使用类型保护来去除`null`。
幸运地是这与在JavaScript里写的代码一致：

```
function f(sn: string | null): string {
    if (sn == null) {
        return "default";
    }
    else {
        return sn;
    }
}
```

这里很明显地去除了`null`，你也可以使用短路运算符：

```
function f(sn: string | null): string {
    return sn || "default";
}
```

如果编译器不能够去除`null`或`undefined`，你可以使用类型断言手动去除。
语法是添加`!`后缀：`identifier!`从`identifier`的类型里去除了`null`和`undefined`：

```
function broken(name: string | null): string {
  function postfix(epithet: string) {
    return name.charAt(0) + '.  the ' + epithet; // error, 'name' is possibly n
  }
  name = name || "Bob";
  return postfix("great");
}

function fixed(name: string | null): string {
  function postfix(epithet: string) {
    return name!.charAt(0) + '.  the ' + epithet; // ok
  }
  name = name || "Bob";
  return postfix("great");
}
```

本例使用了嵌套函数，因为编译器无法去除嵌套函数的null（除非是立即调用的函数表达式）。
因为它无法跟踪所有对嵌套函数的调用，尤其是你将内层函数做为外层函数的返回值。
如果无法知道函数在哪里被调用，就无法知道调用时`name`的类型。

类型别名

类型别名会给一个类型起个新名字。
类型别名有时和接口很像，但是可以作用于原始值，联合类型，元组以及其它任何你需要手写的类型。

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    }
    else {
        return n();
    }
}
```

起别名不会新建一个类型 - 它创建了一个新*名字*来引用那个类型。
给原始类型起别名通常没什么用，尽管可以做为文档的一种形式使用。

同接口一样，类型别名也可以是泛型 - 我们可以添加类型参数并且在别名声明的右侧传入：

```
type Container<T> = { value: T };
```

我们也可以使用类型别名来在属性里引用自己：

```
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

与交叉类型一起使用，我们可以创建出一些十分稀奇古怪的类型。

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
```

然而，类型别名不能出现在声明右侧的任何地方。

```
type Yikes = Array<Yikes>; // error
```

接口 vs. 类型别名

像我们提到的，类型别名可以像接口一样；然而，仍有一些细微差别。

其一，接口创建了一个新的名字，可以在其它任何地方使用。
类型别名并不创建新名字—比如，错误信息就不会使用别名。
在下面的示例代码里，在编译器中将鼠标悬停在`interfaced`上，显示它返回的是`Interface`，但悬停在`aliased`上时，显示的却是对象字面量类型。

```
type Alias = { num: number }
interface Interface {
    num: number;
}
declare function aliased(arg: Alias): Alias;
declare function interfaced(arg: Interface): Interface;
```

另一个重要区别是类型别名不能被`extends`和`implements`（自己也不能`extends`和`implements`其它类型）。
因为[软件中的对象应该对于扩展是开放的，但是对于修改是封闭的](), 你应该尽量去使用接口代替类型别名。

另一方面，如果你无法通过接口来描述一个类型并且需要使用联合类型或元组类型，这时通常会使用类型别名。

字符串字面量类型

字符串字面量类型允许你指定字符串必须的固定值。
在实际应用中，字符串字面量类型可以与联合类型，类型保护和类型别名很好的配合。
通过结合使用这些特性，你可以实现类似枚举类型的字符串。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

你只能从三种允许的字符中选择其一来做为参数传递，传入其它值则会产生错误。

```
Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" |
```

字符串字面量类型还可以用于区分函数重载：

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
    // ... code goes here ...
}
```

可辨识联合（Discriminated Unions）

你可以合并字符串字面量类型，联合类型，类型保护和类型别名来创建一个叫做*可辨识联合*的高级模式，它也称做*标签联合*或*代数数据类型*。
可辨识联合在函数式编程很有用处。
一些语言会自动地为你辨识联合；而TypeScript则基于已有的JavaScript模式。
它具有3个要素：

1. 具有普通的字符串字面量属性—*可辨识的特征*。
2. 一个类型别名包含了那些类型的联合—*联合*。

3. 此属性上的类型保护。

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}
```

首先我们声明了将要联合的接口。
每个接口都有`kind`属性但有不同的字符串字面量类型。
`kind`属性称做*可辨识的特征*或*标签*。
其它的属性则特定于各个接口。
注意，目前各个接口间是没有联系的。
下面我们把它们联合到一起：

```
type Shape = Square | Rectangle | Circle;
```

现在我们使用可辨识联合:

```
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

完整性检查

当没有涵盖所有可辨识联合的变化时，我们想让编译器可以通知我们。
比如，如果我们添加了`Triangle`到`Shape`，我们同时还需要更新`area`:

```
type Shape = Square | Rectangle | Circle | Triangle;
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
    // should error here - we didn't handle case "triangle"
}
```

有两种方式可以实现。
首先是启用`--strictNullChecks`并且指定一个返回值类型：

```
function area(s: Shape): number { // error: returns number | undefined
    switch (s.kind) {
        case "square": return s.size * s.size;
```

```
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

因为switch没有包涵所有情况，所以TypeScript认为这个函数有时候会返回undefined。
如果你明确地指定了返回值类型为number，那么你会看到一个错误，因为实际上返回值的类型为number | undefined。
然而，这种方法存在些微妙之处且--strictNullChecks对旧代码支持不好。

第二种方法使用never类型，编译器用它来进行完整性检查：

```
function assertNever(x: never): never {
    throw new Error("Unexpected object: " + x);
}
function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
        default: return assertNever(s); // error here if there are missing case
    }
}
```

这里，assertNever检查s是否为never类型—即为除去所有可能情况后剩下的类型。
如果你忘记了某个case，那么s将具有一个真实的类型并且你会得到一个错误。
这种方式需要你定义一个额外的函数，但是在你忘记某个case的时候也更加明显。

多态的this类型

多态的this类型表示的是某个包含类或接口的*子类型*。
这被称做*F*-bounded多态性。
它能很容易的表现连贯接口间的继承，比如。
在计算器的例子里，在每个操作之后都返回this类型：

```
class BasicCalculator {
    public constructor(protected value: number = 0) { }
    public currentValue(): number {
        return this.value;
    }
    public add(operand: number): this {
        this.value += operand;
        return this;
    }
    public multiply(operand: number): this {
        this.value *= operand;
        return this;
    }
    // ... other operations go here ...
}

let v = new BasicCalculator(2)
            .multiply(5)
            .add(1)
            .currentValue();
```

由于这个类使用了this类型，你可以继承它，新的类可以直接使用之前的方法，不需要做任何的改变。

```
class ScientificCalculator extends BasicCalculator {
    public constructor(value = 0) {
        super(value);
    }
    public sin() {
        this.value = Math.sin(this.value);
        return this;
    }
    // ... other operations go here ...
}

let v = new ScientificCalculator(2)
        .multiply(5)
        .sin()
        .add(1)
        .currentValue();
```

如果没有this类型，ScientificCalculator就不能够在继承BasicCalculator的同时还保持接口的连贯性。
multiply将会返回BasicCalculator，它并没有sin方法。
然而，使用this类型，multiply会返回this，在这里就是ScientificCalculator。

索引类型（Index types）

使用索引类型，编译器就能够检查使用了动态属性名的代码。
例如，一个常见的JavaScript模式是从对象中选取属性的子集。

```
function pluck(o, names) {
    return names.map(n => o[n]);
}
```

下面是如何在TypeScript里使用此函数，通过**索引类型查询**和**索引访问**操作符：

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
  return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

编译器会检查name是否真的是Person的一个属性。
本例还引入了几个新的类型操作符。
首先是keyof T，**索引类型查询操作符**。
对于任何类型T，keyof T的结果为T上已知的公共属性名的联合。
例如：

```
let personProps: keyof Person; // 'name' | 'age'
```

keyof Person是完全可以与'name' | 'age'互相替换的。
不同的是如果你添加了其它的属性到Person，例如address: string，那么keyof Person会自动变为'name' | 'age' | 'address'。
你可以在像pluck函数这类上下文里使用keyof，因为在使用之前你并不清楚可能出现的属性名。
但编译器会检查你是否传入了正确的属性名给pluck：

```
pluck(person, ['age', 'unknown']); // error, 'unknown' is not in 'name' | 'age'
```

第二个操作符是T[K]，**索引访问操作符**。
在这里，类型语法反映了表达式语法。
这意味着person['name']具有类型Person['name'] — 在我们的例子里则为string类型。
然而，就像索引类型查询一样，你可以在普通的上下文里使用T[K]，这正是它的强大所在。
你只要确保类型变量K extends keyof T就可以了。
例如下面getProperty函数的例子：

```
function getProperty<T, K extends keyof T>(o: T, name: K): T[K] {
    return o[name]; // o[name] is of type T[K]
}
```

getProperty里的o: T和name: K，意味着o[name]: T[K]。
当你返回T[K]的结果，编译器会实例化键的真实类型，因此getProperty的返回值类型会随着你需要的属性改变。

```
let name: string = getProperty(person, 'name');
let age: number = getProperty(person, 'age');
let unknown = getProperty(person, 'unknown'); // error, 'unknown' is not in 'na
```

索引类型和字符串索引签名

keyof和T[K]与字符串索引签名进行交互。
如果你有一个带有字符串索引签名的类型，那么keyof T会是string。
并且T[string]为索引签名的类型：

```
interface Map<T> {
    [key: string]: T;
}
let keys: keyof Map<number>; // string
let value: Map<number>['foo']; // number
```

映射类型

一个常见的任务是将一个已知的类型每个属性都变为可选的：

```
interface PersonPartial {
    name?: string;
    age?: number;
}
```

或者我们想要一个只读版本：

```
interface PersonReadonly {
    readonly name: string;
    readonly age: number;
}
```

这在JavaScript里经常出现，TypeScript提供了从旧类型中创建新类型的一种方式 — **映射类型**。
在映射类型里，新类型以相同的形式去转换旧类型里每个属性。
例如，你可以令每个属性成为`readonly`类型或可选的。
下面是一些例子：

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
}
type Partial<T> = {
    [P in keyof T]?: T[P];
}
```

像下面这样使用：

```
type PersonPartial = Partial<Person>;
type ReadonlyPerson = Readonly<Person>;
```

下面来看看最简单的映射类型和它的组成部分：

```
type Keys = 'option1' | 'option2';
type Flags = { [K in Keys]: boolean };
```

它的语法与索引签名的语法类型，内部使用了`for .. in`。
具有三个部分：

1. 类型变量`K`，它会依次绑定到每个属性。
2. 字符串字面量联合的`Keys`，它包含了要迭代的属性名的集合。
3. 属性的结果类型。

在个简单的例子里，`Keys`是硬编码的的属性名列表并且属性类型永远是`boolean`，因此这个映射类型等同于：

```
type Flags = {
    option1: boolean;
    option2: boolean;
}
```

在真正的应用里，可能不同于上面的`Readonly`或`Partial`。
它们会基于一些已存在的类型，且按照一定的方式转换字段。
这就是`keyof`和索引访问类型要做的事情：

```
type NullablePerson = { [P in keyof Person]: Person[P] | null }
type PartialPerson = { [P in keyof Person]?: Person[P] }
```

但它更有用的地方是可以有一些通用版本。

```
type Nullable<T> = { [P in keyof T]: T[P] | null }
```

```
type Partial<T> = { [P in keyof T]?: T[P] }
```

在这些例子里，属性列表是`keyof T`且结果类型是`T[P]`的变体。
这是使用通用映射类型的一个好模版。
因为这类转换是同态的，映射只作用于`T`的属性而没有其它的。
编译器知道在添加任何新属性之前可以拷贝所有存在的属性修饰符。
例如，假设`Person.name`是只读的，那么`Partial<Person>.name`也将是只读的且为可选的。

下面是另一个例子，`T[P]`被包装在`Proxy<T>`类里：

```
type Proxy<T> = {
    get(): T;
    set(value: T): void;
}
type Proxify<T> = {
    [P in keyof T]: Proxy<T[P]>;
}
function proxify<T>(o: T): Proxify<T> {
   // ... wrap proxies ...
}
let proxyProps = proxify(props);
```

注意`Readonly<T>`和`Partial<T>`用处不小，因此它们与`Pick`和`Record`一周被包含进了TypeScript的标准库里：

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
}
type Record<K extends string | number, T> = {
    [P in K]: T;
}
```

`Readonly`，`Partial`和`Pick`是同态的，但`Record`不是。
因为`Record`并不需要输入类型来拷贝属性，所以它不属于同态：

```
type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>
```

非同态类型本质上会创建新的属性，因此它们不会从它处拷贝属性修饰符。

由映射类型进行推断

现在你了解了如何包装一个类型的属性，那么接下来就是如果拆包。
其实这也非常容易：

```
function unproxify<T>(t: Proxify<T>): T {
    let result = {} as T;
    for (const k in t) {
        result[k] = t[k].get();
    }
    return result;
}

let originalProps = unproxify(proxyProps);
```

注意这个拆包推断只适用于同态的映射类型。
如果映射类型不是同态的，那么需要给拆包函数一个明确的类型参数。