介绍

TypeScript里的类型兼容性是基于结构子类型的。
结构类型是一种只使用其成员来描述类型的方式。
它正好与名义（nominal）类型形成对比。（译者注：在基于名义类型的类型系统中，数据类型的兼容性或等价性是通过明确的声明和/或类型的名称来决定的。这与结构性类型系统不同，它是基于类型的组成结构，且不要求明确地声明。）
看下面的例子：

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

在使用基于名义类型的语言，比如C#或Java中，这段代码会报错，因为Person类没有明确说明其实现了Named接口。

TypeScript的结构性子类型是根据JavaScript代码的典型写法来设计的。
因为JavaScript里广泛地使用匿名对象，例如函数表达式和对象字面量，所以使用结构类型系统来描述这些类型比使用名义类型系统更好。

关于可靠性的注意事项

TypeScript的类型系统允许某些在编译阶段无法确认其安全性的操作。当一个类型系统具此属性时，被当做是“不可靠”的。TypeScript允许这种不可靠行为的发生是经过仔细考虑的。通过这篇文章，我们会解释什么时候会发生这种情况和其有利的一面。

开始

TypeScript结构化类型系统的基本规则是，如果`x`要兼容`y`，那么`y`至少具有与`x`相同的属性。比如：

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: 'Alice', location: 'Seattle' };
x = y;
```

这里要检查`y`是否能赋值给`x`，编译器检查`x`中的每个属性，看是否能在`y`中也找到对应属

性。
在这个例子中，y必须包含名字是name的string类型成员。y满足条件，因此赋值正确。

检查函数参数时使用相同的规则：

```
function greet(n: Named) {
    alert('Hello, ' + n.name);
}
greet(y); // OK
```

注意，y有个额外的location属性，但这不会引发错误。
只有目标类型（这里是Named）的成员会被一一检查是否兼容。

这个比较过程是递归进行的，检查每个成员及子成员。

比较两个函数

相对来讲，在比较原始类型和对象类型的时候是比较容易理解的，问题是如何判断两个函数是兼容的。
下面我们从两个简单的函数入手，它们仅是参数列表略有不同：

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

要查看x是否能赋值给y，首先看它们的参数列表。
x的每个参数必须能在y里找到对应类型的参数。
注意的是参数的名字相同与否无所谓，只看它们的类型。
这里，x的每个参数在y中都能找到对应的参数，所以允许赋值。

第二个赋值错误，因为y有个必需的第二个参数，但是x并没有，所以不允许赋值。

你可能会疑惑为什么允许忽略参数，像例子y = x中那样。
原因是忽略额外的参数在JavaScript里是很常见的。
例如，Array#forEach给回调函数传3个参数：数组元素，索引和整个数组。
尽管如此，传入一个只使用第一个参数的回调函数也是很有用的：

```
let items = [1, 2, 3];

// Don't force these extra arguments
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach((item) => console.log(item));
```

下面来看看如何处理返回值类型，创建两个仅是返回值类型不同的函数：

```
let x = () => ({name: 'Alice'});
let y = () => ({name: 'Alice', location: 'Seattle'});

x = y; // OK
```

```
y = x; // Error because x() lacks a location property
```

类型系统强制源函数的返回值类型必须是目标函数返回值类型的子类型。

函数参数双向协变

当比较函数参数类型时，只有当源函数参数能够赋值给目标函数或者反过来时才能赋值成功。
这是不稳定的，因为调用者可能传入了一个具有更精确类型信息的函数，但是调用这个传入的函数的时候却使用了不是那么精确的类型信息。
实际上，这极少会发生错误，并且能够实现很多JavaScript里的常见模式。例如：

```
enum EventType { Mouse, Keyboard }

interface Event { timestamp: number; }
interface MouseEvent extends Event { x: number; y: number }
interface KeyEvent extends Event { keyCode: number }

function listenEvent(eventType: EventType, handler: (n: Event) => void) {
    /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MouseEvent) => console.log(e.x + ',' + e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) => console.log((<MouseEvent>e).x + ','
listenEvent(EventType.Mouse, <(e: Event) => void>((e: MouseEvent) => console.lo

// Still disallowed (clear error). Type safety enforced for wholly incompatible
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

可选参数及剩余参数

比较函数兼容性的时候，可选参数与必须参数是可互换的。
源类型上有额外的可选参数不是错误，目标类型的可选参数在源类型里没有对应的参数也不是错误。

当一个函数有剩余参数时，它被当做无限个可选参数。

这对于类型系统来说是不稳定的，但从运行时的角度来看，可选参数一般来说是不强制的，因为对于大多数函数来说相当于传递了一些`undefinded`。

有一个好的例子，常见的函数接收一个回调函数并用对于程序员来说是可预知的参数但对类型系统来说是不确定的参数来调用：

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
    /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ', ' + y));

// Confusing (x and y are actually required) and undiscoverable
```

```
invokeLater([1, 2], (x?, y?) => console.log(x + ', ' + y));
```

函数重载

对于有重载的函数，源函数的每个重载都要在目标函数上找到对应的函数签名。
这确保了目标函数可以在所有源函数可调用的地方调用。

枚举

枚举类型与数字类型兼容，并且数字类型与枚举类型兼容。不同枚举类型之间是不兼容的。比如，

```
enum Status { Ready, Waiting };
enum Color { Red, Blue, Green };

let status = Status.Ready;
status = Color.Green;  //error
```

类

类与对象字面量和接口差不多，但有一点不同：类有静态部分和实例部分的类型。
比较两个类类型的对象时，只有实例的成员会被比较。
静态成员和构造函数不在比较的范围内。

```
class Animal {
    feet: number;
    constructor(name: string, numFeet: number) { }
}

class Size {
    feet: number;
    constructor(numFeet: number) { }
}

let a: Animal;
let s: Size;

a = s;  //OK
s = a;  //OK
```

类的私有成员

私有成员会影响兼容性判断。
当类的实例用来检查兼容时，如果目标类型包含一个私有成员，那么源类型必须包含来自同一个类的这个私有成员。
这允许子类赋值给父类，但是不能赋值给其它有同样类型的类。

泛型

因为TypeScript是结构性的类型系统，类型参数只影响使用其做为类型一部分的结果类型。比如，

```
interface Empty<T> {
}
let x: Empty<number>;
let y: Empty<string>;

x = y;  // okay, y matches structure of x
```

上面代码里，`x`和`y`是兼容的，因为它们的结构使用类型参数时并没有什么不同。
把这个例子改变一下，增加一个成员，就能看出是如何工作的了：

```
interface NotEmpty<T> {
    data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y;  // error, x and y are not compatible
```

在这里，泛型类型在使用时就好比不是一个泛型类型。

对于没指定泛型类型的泛型参数时，会把所有泛型参数当成`any`比较。
然后用结果类型进行比较，就像上面第一个例子。

比如，

```
let identity = function<T>(x: T): T {
    // ...
}

let reverse = function<U>(y: U): U {
    // ...
}

identity = reverse;  // Okay because (x: any)=>any matches (y: any)=>any
```

高级主题

子类型与赋值

目前为止，我们使用了兼容性，它在语言规范里没有定义。
在TypeScript里，有两种类型的兼容性：子类型与赋值。
它们的不同点在于，赋值扩展了子类型兼容，允许给`any`赋值或从`any`取值和允许数字赋值给枚举类型或枚举类型赋值给数字。

语言里的不同地方分别使用了它们之中的机制。
实际上，类型兼容性是由赋值兼容性来控制的甚至在`implements`和`extends`语句里。
更多信息，请参阅TypeScript语言规范.