



下载APP



23 | 增强编译器前端功能第2步：增强类型体系

2021-09-29 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 18:33 大小 17.00M



你好，我是宫文学。

你可能也注意到了，我们在第二部分的主要任务，是要让 PlayScript 扩展到支持更多的类型。在这个任务中，对类型的处理能力就是一个很重要的功能。

其实在第一部分，我们已经实现了一定的类型处理功能，包括类型检查、类型自动推断等，但其实还有更多的类型处理能力需要支持。

对于一门语言来说，类型系统是它的核心。语言之间的差别很多时候都体现在类型系统的设计上，程序员们通常也会对类型处理的内部机制很感兴趣。而 TypeScript 比 JavaScript 语言增强的部分，恰恰就是一个强大而又灵活的类型系统，所以我们就更有必要讨论一下与类型有关的话题了。



那么通过今天这节课，我们就来增强一下 PlayScript 的类型处理能力，在这过程中，我们也能学习到更多与类型系统有关的知识点，特别是能对类型计算的数学实质有所认知。

首先，我们来看看 TypeScript 的类型系统有什么特点。

TypeScript 的类型系统

从 TypeScript 的名字上，你就可以看出来，这门语言在类型系统的设计上，一定是下了功夫的。也确实是这样，TypeScript 在设计之初，就想弥补 JavaScript 弱类型、动态类型所带来的缺点。特别是，当程序规模变大的时候，弱类型、动态类型很容易不经意地引入一些错误，而且还比较难以发现。

所以 TypeScript 的设计者，希望通过提供一个强类型体系，让编译器能够检查出程序中潜在的错误，这也有助于 IDE 工具提供更友好的特性，比如准确提示类的属性和方法，从而帮助程序员编写更高质量的程序。

而 TypeScript 也确实实现了这个设计目标。它的类型系统功能很强大，表达能力很强，既有利于提高程序的正确性，同时又没有削弱程序员自由表达各种设计思想的能力。

那么我们现在就来看一看 TypeScript 的类型系统到底有什么特点。

首先，TypeScript 继承了 JavaScript 的几个预定义的类型，比如 number、string 和 boolean 等。

在 JavaScript 中，我们不需要声明类型，比如下面两句代码就是。在程序运行的时候，系统会自动给 age 和 name1 分别关联一个 number 和 string 类型的值。

```
1 var age = 18;  
2 var name1 = "richard";
```

[复制代码](#)

而在 TypeScript 中呢，你需要用 let 关键字来声明变量。在下面的示例程序中，age 和 name1 被我们用 let 关键字分别赋予了 number 和 string 类型。

```
1 let age = 18;
2 let name1 = "richard";
```

[复制代码](#)

这两行代码里的类型是被推导出来的，它们跟显式声明类型的方式是等价的。

```
1 let age:number = 18;
2 let name1:string = "richard";
```

[复制代码](#)

第二，TypeScript 禁止了变量类型的动态修改。

在 JavaScript 中，我们可以动态地修改变量的类型。比如在下面两行代码中，age 一开头是 number 型的，后来被改成了 string 型，也是允许的：

```
1 var age = 18;
2 age = "eighteen";
```

[复制代码](#)

但在 TypeScript 中，如果你一开头给 age 赋一个 number 的值，后面再赋一个 string 类型的值，编译器就会报错：

```
1 let age = 18;
2 age = "eighteen"; //错误！
```

[复制代码](#)

这是因为，上面的第一行代码等价于显式声明 age 为 number 类型，因为 TypeScript 会根据变量初始化的部分，来推断出 age 的类型。而这个类型一旦确定，后面就不允许再修改了。

```
1 let age:number = 18;
2 age = "eighteen";
```

[复制代码](#)

不过，如果完全不允许类型动态变化，可能会失去 JavaScript 灵活性这个优点，会让某些程序员觉得用起来不舒服。所以，TypeScript 还留了一个口子，就是 any 类型。

第三，只有 any 类型允许动态修改变量的类型。

在 TypeScript 中，如果你声明变量的时候不指定任何类型，或者显式地指定变量类型为 any，那变量的类型都是 any，程序也就可以动态地修改变量的类型，我们可以看看下面这个例子：

[复制代码](#)

```
1 let age; //等价于 let age:any;
2 age = 18;
3 console.log(typeof age);
4 age = "eighteen";
5 console.log(typeof age);
```

如果我们编译并运行这个示例程序，我们会得到这样的结果：

```
→ 23 git:(master) × tsc example.ts
→ 23 git:(master) × node example
number
string
```

你会看到，在我们第二次给 age 赋值的时候，age 的类型真的被改变了。

第四，TypeScript 支持联合类型。

你在使用 TypeScript 编程的时候，应该会很快注意到它的联合类型的特性。比如，在下面这个例子中，value 的类型可以是 number 或 string，那你给 value 赋这两种类型的值都是可以的。当然，如果你给 value 赋一个 boolean 值，那仍然是错误的，因为联合类型中不包含 boolean 类型。

[复制代码](#)

```
1 let value:number|string;
2 value = 18; //OK
```

```
3 value = "richard"; //也OK
4 value = true;      //错误！
```

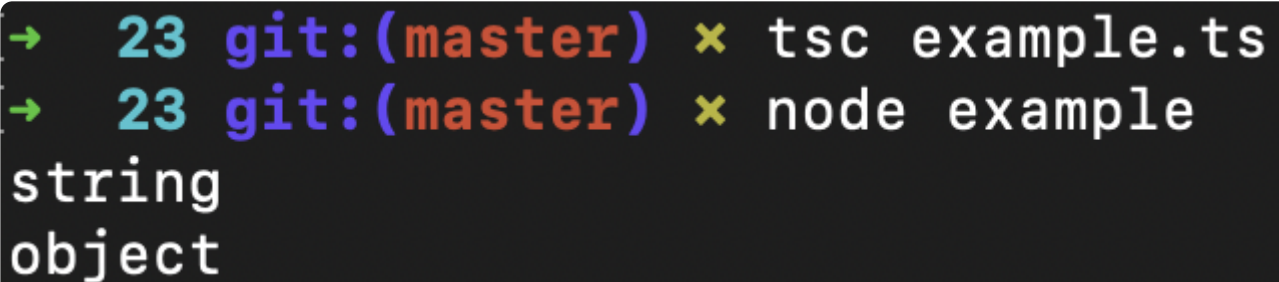
第五，TypeScript 支持把值作为类型。

什么意思呢？我们依然来看一个例子。你看，在下面的语句中，我们可以让 name1 取值为 string 或者 null。这在编程中很方便，特别是在声明对象属性的时候。因为我们可能一开始不知道名称是什么，我们就可以先让它的值为 null。之后，等知道了名称以后，我们再给 name1 赋予正式的值就好了。

[复制代码](#)

```
1 var name1;
2 name1 = null;
3 console.log(typeof name1); //输出：object
4 name1 = "richard";
5 console.log(typeof name1); //输出：string
```

不过，这个 null 并不是类型，而是一个值，它实际的类型是对象（object），你可以看看示例程序运行的结果来验证一下，我这里也放了张截图：



```
→ 23 git:(master) x tsc example.ts
→ 23 git:(master) x node example
string
object
```

除了在联合类型里使用值，你甚至还可以用一个单独的值作为类型注解。比如在下面的示例程序中，value2 只能取 0 值。这可能在实际编程中没有太大的用，因为 value2 相当于是一个常量，而不是变量。我们更可能像 value1 那样，规定合法的取值是多个值中的一个。

[复制代码](#)

```
1 let value1 : 0|1;
2 let value2 : 0;
3 value1 = 0;    //OK
4 value1 = 2;    //错误！
5 value2 = 2;    //错误！
```



其实，TypeScript 的类型系统还有更多丰富的特性，比如数组类型、交集类型（Intersection Type）、通过 class 和 interface 实现自定义的类型、泛型等等，非常强大。不过我们还是按照循序渐进的原则，先从比较简单的特性入手，然后逐步深化。

所以，我们就先聚焦在前面提到的几点特性上。特别是**联合类型**和**值类型**，这都是我们之前没有实现过的，我们就先来实现一下。在这个过程中，我们就能先小小体会一下 TypeScript 类型系统到底有多强大。

那么现在，我们首先针对联合类型、值类型，来升级我们对类型的支持能力。

支持联合类型和值类型

在目前的语法解析器中，我们对类型的解析很简单。比如像 “let a : number;” 这样一个简单的语句，我们只要把 number 作为一个关键字提取出来，然后再转换成一个代表 number 类型的内部对象就可以了。所以，我们语法解析器中与变量声明有关的语法规则是很简单的，我总结了写在了下面：

 复制代码

```
1 variableDecl : Identifier typeAnnotation? ('=' expression)? ;  
2 typeAnnotation : ':' typeName;  
3 typename : 'number' | 'boolean' | 'string' | 'any' | 'void' ;
```

不过，我们在课程里已经很久没有写过语法规则了，不知道你还能不能重拾对它们的记忆呢？我帮你简单解释一下这三条规则。

变量声明 (variableDecl)：在变量声明里，标识符后面可以跟一个可选的类型注解；

类型注解 (typeAnnotation)：类型注解以 “:” 号开头，后面跟一个类型名称。

类型名称 (typeName)：类型名称可以是 number、boolean、string、any 或 void 这几个值之一，这也是目前 PlayScript 所能识别的少量类型。

但是，我们现在对我们的要求可不一样了。我们现在需要 PlayScript 支持联合类型和值类型，并且后面还要支持更复杂的类型体系，那我们现在就必须扩展一下我们语言中针对类型的语法规则了。

新的语法规则我放在了下面，你可以阅读一下，看看能不能读懂它们的含义：

 复制代码

```
1 typeAnnotation : ':' type_ ;
2 type_ : unionOrIntersectionOrPrimaryType ;
3 unionOrIntersectionOrPrimaryType : primaryType ('|' | '&' primaryType)* ;
4 primaryType : predefinedType | literal | typeReference | '(' type_ ')' | prima
5 predefinedType : 'number' | 'string' | 'boolean' | 'any' | 'void' ;
```

你可以看到，这个新的规则有一些不同。

首先，我们的类型注解 (typeAnnotation) 有了变化，现在改成了 “:” 号后面跟着类型 (type_)；

然后，我们在看类型 (type_)，类型可以有多种，但目前我们的语法规则里只有 unionOrIntersectionOrPrimaryType 这一类，以后还可以拓展。

第三个，unionOrIntersectionOrPrimaryType：它的字面意思是联合类型、交集类型或基础类型。从规则中你可以看到，联合类型是由一个个基础类型用 “|” 号连接在一起的。交集类型与联合类型相似，差别在于它使用 “&” 号来连接基础类型的。当然，交集类型目前我们还用不到，但我们先在语法规则中预留下它的位置。

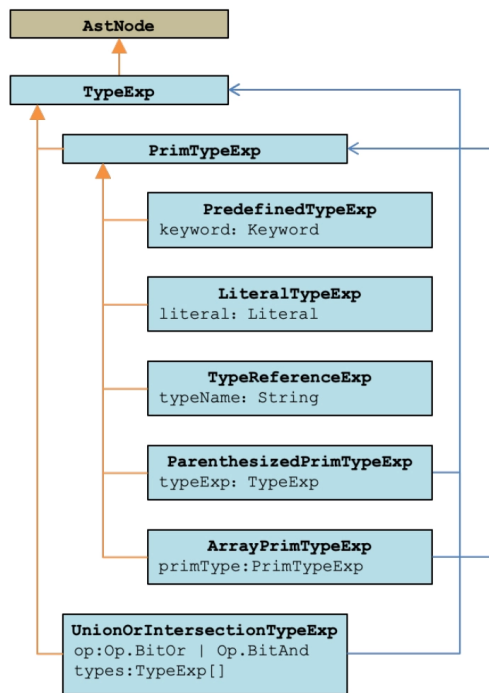
第四个不同点在于基础类型 (primaryType)，基础类型包括预定义的类型、字面量、类型引用、被括号括起来的类型，以及数组类型。目前我们只会用到前两个，预定义的类型就是之前的 number、string 这些。而字面量包括数字字面量、字符串字面量、布尔字面量等，它们在解析后会形成我们前面提到的值类型。

好了，现在我们已经把与类型有关的语法规则写好了。接下来，就要升级一下解析器，让它能够支持这些新的语法规则。

根据我们之前学习过的语法分析的知识，其实你只要写出来了语法规则，照着规则来实现语法分析程序并不难。你可以看一下 parseType、parseUnionOrIntersectionOrPrimaryType、parsePrimTypeExp 这几个方法。

不过呢，语法解析的结果，是要形成 AST，所以我们这里还必须**增加一些 AST 节点**，来代表解析出来的这些类型信息。

与类型有关的 AST 节点，我也画成了类图，并放在了下面。这里面包括 TypeExp、PrimTypeExp、PredefinedTypeExp、LiteralTypeExp 和 UnionOrIntersectionTypeExp 等节点，并且它们之间还有继承和引用的关系。

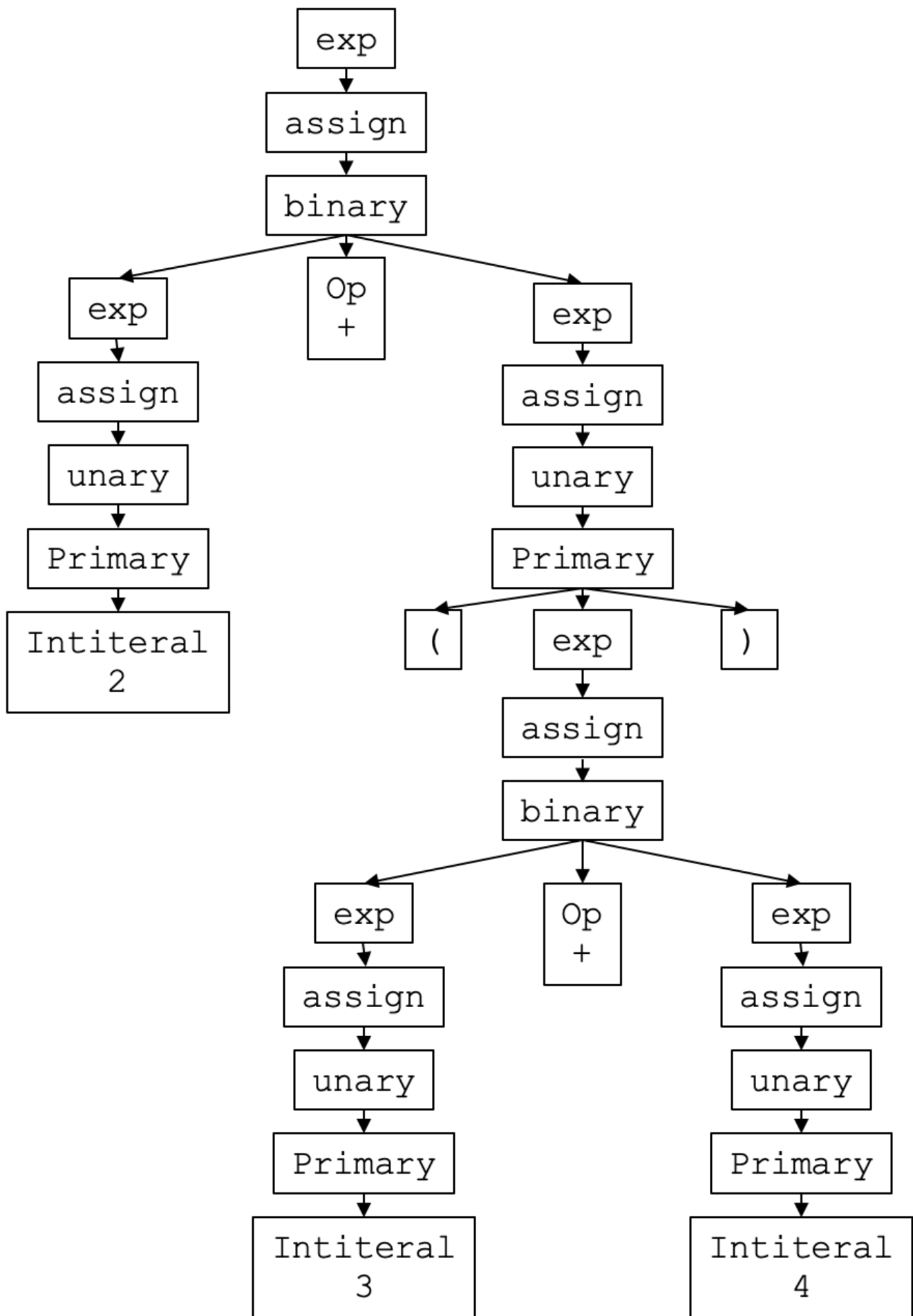


极客时间

不过，课程上到这里，在谈到 AST 节点设计的时候，我觉得有必要插一个小知识点，也就是关于 AST 和 CST 的区别。

这里出现了一个新的缩略词 CST。CST 是 Concrete Syntax Tree 的缩写，意思是具体语法树。从这个名称，你就能看出来，它跟 AST，也就是抽象语法树 (Abstract Sytax Tree) 是相对的。那么它们有什么差别呢？

你可能已经注意到了，我在课程里所设计的 AST 节点，并不是跟语法规则的名称完全一样的，所形成的语法树也跟解析的过程不完全一致。以解析表达式 $2+(3+4)$ 为例，如果忠实地按照解析的过程来形成语法树，跟现在的语法树会有很大的不同：



你能看出来，相比 AST，CST 更加忠实地体现了源代码的结构。比如，它没有丢掉源代码中的任何一个 Token，包括 + 号和圆括号，这样可以把 AST 和源代码精准地对应起来，

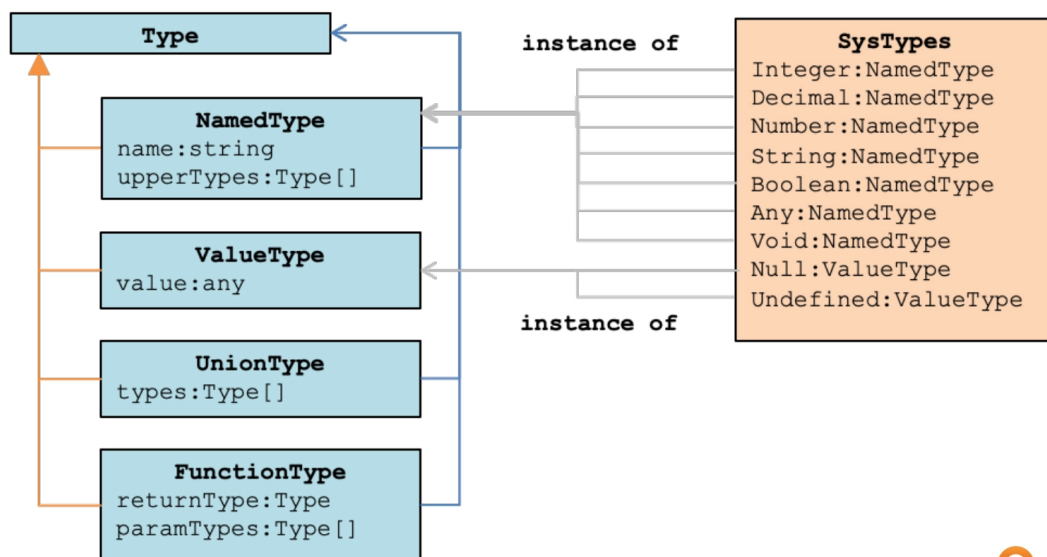
这也能更友好地显示一些错误信息。比如，你可以在 IDE 里标出某个括号用错了。CST 作为对解析过程和源代码的忠实体现，它也被称作解析树（Parse Tree）。

而在 AST 里，更多体现的源代码的内在含义，而不是去纠结有没有跟源代码一一对应上。如果我们把它们两种翻译比作英语翻译的话，AST 是意译，而 CST 是直译。不过 AST 的特点也很明显。一方面它更简洁，另一方面 AST 节点的对象设计更容易体现程序的内在含义。

在实际使用中呢，有的编译器会先生成 CST，再基于 CST 生成 AST。但这样显然会增加编译过程的计算量，降低编译器的性能。所以，大部分编译器都是直接生成 AST。不过，如果你写的编译器主要是用于支持 IDE 工具，那么 CST 可能会发挥更大的作用，因为它跟源代码的直接对应能力更强。

好，回到我们的主线上来。现在你应该明白了，为什么我们关于类型的 AST 节点不是跟语法规则直接对应的了。这样的设计会更有利于对类型进行进一步的处理。

不过到这还没有完。我们刚才设计的只是有关类型的 AST 节点的对象结构。但是我们在程序内部保存类型信息的时候，是不可能去保存一些 AST 节点的，还需要一套对象结构来表示类型。我也画出了相应的类图，你可以看一下。



在这里，我们把 Type 作为各种类型的基类，这下面有 NamedType、ValueType、FunctionType、UnionType 等等类型。另外，我还设计了一个 SysTypes 类，里面存放了预定义的一些类型的实例，比如 Boolean、Number、String、Any 和 Void 都是 NamedType 的实例，而 Null、Undefined 是值类型的实例。

设计完表达类型的对象结构以后，我们还要**基于 AST 解析出类型信息**来。针对这个功能，我写了一个语义分析程序，叫做 TypeResolver。它能够把与类型有关的 AST 节点，转化成类型对象。这个 TypeResolver 程序，我们还会不断地演化，让它能够处理更复杂的类型。特别是在我们后面实现了自定义类型的时候，类型的消解算法还会变得比现在复杂一些。

好了，现在类型信息也能够被正确地消解了。换句话说，现在我们的程序，已经能够正确解析带有联合类型和值类型的程序了。

我们现在就动手试试看，我这里给了一个示例程序：

```
1 let age : string|number;  
2 let name1 : string|null;
```

[复制代码](#)

你可以用 `node play example.ts -v` 命令，显示出解析后的 AST。你会看到 AST 里面已经体现了与类型有关的 AST 节点信息，以及类型消解后的结果。

```
Prog
  VariableStatement
    VariableDecl age(string | number)
      UnionType
        String
        Number
      no initialization.
  VariableStatement
    VariableDecl name1(string | null)
      UnionType
        String
        LiteralType: null
      no initialization.
  ReturnStatement
```

但是还差一点，如果我们要给示例程序中的变量赋值，那么我们还必须升级类型检查的算法，来支持新的联合类型和值类型。

升级类型检查功能

我们都知道，在给变量赋值的时候，我们必须要进行类型的检查。我们之前也介绍过类型检查功能。不过，这节课我想带你从不同的视角来认识类型检查，这会涉及我们之前提到过的类型计算技术。

在这节课，当我们介绍 TypeScript 的类型体系的时候，你可能或多或少会有一种感觉：**怎么 TypeScript 的类型有点像集合呀？**没错！你看，联合类型就是多个类型的合集。我们还隐约提到了交集类型，而交集显然也是对集合的一种运算。

是的，集合这种数学方法，可以非常好地用于表达类型的计算。你可以这么想，什么是类型呢？类型就是一组值的集合。比如，number 类型就是所有数字的集合，string 类型就是所有字符串的集合。当然，我们也可以做个更大的集合，同时包含 number 和 string，这就形成了一个新类型；你也可以做一个更小的集合，比如只包含几个数字，这也是一个新类型。

那么从集合运算的角度看，什么是类型检查呢？

类型检查的规则，就是给变量所赋的值，一定要属于变量的类型所对应的集合。对于 “let a : number = 1;” 这个语句来说，显然 1 属于 number 的集合，所以是合法的。而对于 “let a : 0 | 1 = 1” 来说，1 也属于集合{0,1}，因此也是合法的。

所以，我们升级的类型检查算法，就用到了集合运算。了解了原理以后，你就可以再去看看 [@TypeChecker](#)和 [@Type](#)类中的实现，应该就比较容易看懂了。

不过，类型检查只是类型计算技术的一个体现。在下一节课里，我们还会结合前一节课的数据流分析技术和这节课的类型计算技术，实现更多有趣的特性，比如实现 null 安全性，你也可以先简单预习一下相关的知识。

课程小结

这节课到这里就讲完了，今天我希望你能够记住下面这几个知识点：

首先，我们学习了 TypeScript 类型体系的部分特征。TypeScript 能够通过显式的方式来声明类型。除了 any 类型之外，TypeScript 变量的类型都是不能动态改变的。在基础类型之上，TypeScript 还支持值类型和联合类型。

第二，在升级 TypeScript 与类型相关的语法规则的时候，我们设计了一些新的代表 AST 节点的类。在这里，我们穿插介绍了 CST 的概念。与 AST 相比，CST 更加忠实地反映源代码的结构、语法规则和解析过程，能够对 IDE 工具有更好的支持。但它也有缺点，就是树的结构更大，在后序处理时会更加繁琐。

第三，在升级类型检查功能的时候，我们介绍了类型计算所采用的数学方法，也就是集合计算。在程序里把变量 a 的值赋给变量 b，就是要求类型 a 的集合是类型 b 的集合的子集。你可以试着以集合的思维去重新解读，你以前了解的与类型有关的知识，比如类的继承关系，相信会给你带来崭新的视角。

思考题

在你熟悉的其他语言中，有类似 TypeScript 的联合类型和值类型的特性吗？或者，它们跟 TypeScript 的特性有什么差别呢？欢迎你在留言区分享观点。

欢迎你把这节课分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

1. [这节课示例代码目录](#)
2. 与类型有关的 [AST 节点](#)
3. [类型对象](#)
4. 这节课的测试程序 [example_type.ts](#)

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | 增强编译器前端功能第1步：再识数据流分析技术

下一篇 24 | 增强编译器前端功能第3步：全面的集合运算

4 周年庆限定

299元随心畅学卡

五门课程任你选，总价值高达千元

超值拿下



精选留言 (3)

写留言



qinsi

2021-10-02

TypeScript的union type就是Haskell ADT里的sum type，不过是untagged，所以在判断类型上有诸多不便。TypeScript也可以定义product type:

```
````typescript
type pair = [string, number]...
```

展开



奋斗的蜗牛

2021-09-29

赞，类型系统的实现一直看不太明白，是个难点

展开



quanee

2021-09-29

老师，我们写的语言最后能自举吗？

展开



