



下载APP



24 | 增强编译器前端功能第3步：全面的集合运算

2021-10-08 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 22:07 大小 20.26M



你好，我是宫文学。

在上一节课，我们扩展了我们语言的类型体系，还测试了几个简单的例子。从中，我们已经能体会出一些 TypeScript 类型体系的特点了。

不过，TypeScript 的类型体系其实比我们前面测试的还要强大得多，能够在多种场景下进行复杂的类型处理。

今天这节课，我们会通过多个实际的例子，来探索 TypeScript 的类型处理能力。并且在这个过程中，你还会进一步印证我们上一节课的一个知识点，就是**类型计算实际上就是集合运算**。在我们今天的这些例子中，你会见到多种集合运算，包括子集判断、重叠判断，以及交集、并集和补集的计算。



首先，让我们看几个例子，来理解一下类型计算的使用场景。

类型计算的场景

我们先看第一个例子：

```
1 function foo1(age : number|null){
2     let age1 : string|number;
3     age1 = age;    //编译器在这里会检查出错误。
4     console.log(age1);
5 }
```

[复制代码](#)

在这个例子中，我们用到了 `age` 和 `age1` 两个变量，它们都采用了联合类型。一个是 `number|null`，一个是 `string|number`。

如果你用 `--strict` 选项来编译这个程序，那么 `tsc` 会报错：

```
→ 24 git:(master) × tsc --strict example_type2.ts
example_type2.ts:10:5 - error TS2322: Type 'number | null' is not assignable
to type 'string | number'.
  Type 'null' is not assignable to type 'string | number'.

10     age1 = age;
    ~~~~


Found 1 error.
```

这个错误信息的意思是：类型 `number|null` 不能赋给类型 `string|number`。具体来说，`null` 是不能赋给 `string|number` 的。

这说明什么呢？这说明对于赋值语句，比如 `x = y` 来说，它会有一个默认要求，要求 `y` 的类型要么跟 `x` 一样，要么是 `x` 的子集才可以。我们把这个关系记做 `y.type <= x.type`。

那么，其他的二元运算，是不是也像赋值运算那样，需要一个类型是另一个类型的子集呢？


不是的。不同的运算，做类型检查的规则是不同的。比如，对于 `"=="` 和 `"!="` 这两个运算符，只需要两个类型有交集就可以。你可以用 `tsc` 编译一下这个例子：

 复制代码

```
1 function foo2(age1 : number|null, age2:string|number){
2     if (age1 == age2){        //OK。只要两个类型有交集就可以。
3         console.log("same age!");
4     }
5 }
```

你会看到，编译器并不会报错。这说明，两个不同的类型，只要它们有交集，就可以进行等值和不等值比较。并且，即使 age1 的值是 null，age2 的值是一个字符串，等值比较仍然是有意义的，比较的结果是不相等。

那如果两个类型没有交集，会发生什么情况呢？我们看看下面的例子，参数 x 和 y 属于不同的类型，它们之间没有交集。

 复制代码

```
1 function foo3(x : number|null, y:string|boolean){
2     if (x == y){        //编译器报错：两个类型没有交集
3         console.log("x and y is the same");
4     }
5 }
```

这次，如果你用 tsc 去编译，即使不加-strict 选项，编译器也会报错：

```
→ 24 git:(master) x tsc example_type2.ts
example_type2.ts:21:9 - error TS2367: This condition will always return 'false' since the types 'number' and 'string | boolean' have no overlap.

21     if (x == y){
    ~~~~~

Found 1 error.
```

编译器会说，这个条件表达式会永远返回 false，因为这两个类型没有交集。

到此为止，我们就了解清楚等值比较的规则了，也就是要求两个类型有交集才可以，或者说两个类型要存在重叠。

那其他的比较运算符，比如 >，>=，<，<=，也遵循相同的规则吗？

我们把 foo2 中的 == 运算符改为 >= 运算符，得到一个新的示例程序：

[复制代码](#)

```
1 function foo4(age1 : number|null, age2:string|number){
2     if (age1 >= age2){ //编译器报错
3         console.log("bigger age!");
4     }
5 }
```

我们再把这个示例程序用 tsc --strict 模式编译一下，编译器也会报错：

```
24 git:(master) x tsc --strict example_type2.ts
example_type2.ts:27:9 - error TS2531: Object is possibly 'null'.
```

```
27     if (age1 >= age2){ //编译器报错
    ~~~~
```

```
Found 1 error.
```

这次报错的原因，是 age1 有可能取值为 null，而 null 是不能做大小的比较的。这说明，有些类型是不能做大小比较的。

那什么类型之间可以做大小比较呢？number、string、boolean 类型之间都是可以的。但 object 类型、undefined 类型，就不可以做比较了。所以，如果我们把 foo4 示例程序中 age1 的类型中去掉 null 值之后，编译器就不会报错了。

[复制代码](#)

```
1 function foo5(age1 : number, age2:string|number){
2     if (age1 >= age2){ //OK。
3         console.log("bigger age!");
4     }
5 }
```

刚才我们总结了等值比较和大小比较做类型检查的规则。你还可以进一步研究一下加减乘除这几个运算的类型检查的规则，我这里也整理了一下。

首先是 + 号运算符。+ 号运算符有两个语义：一个是作为字符串的连接符使用，这时候类型检查的规则是，只要 + 号运算符一边的类型是 string 型的，那么另一边可以是任意的类

型，因为任意的类型都可以转化成字符串；+ 号的另一个语义是做数字的算术运算，在这种情况下，运算符两边只能是数字类型，包括 number 类型和枚举类型。

其次是 - 号、* 号和 / 号。它们做类型检查的规则，跟 + 号的第二个语义的规则是一样的，也就是运算符的两边只能是数字类型。

再进一步，你还可以研究一下逻辑运算符，也就是 &&、|| 和 ! 这几个运算符。这几个运算符要求操作数是 boolean 值的。不过，在 TypeScript 中，任意类型都可以转化为 boolean 值，包括 string、number、object 和 undefined 类型。这些类型中，有些值等价于 true，而其他值等价于 false。等价于 false 的值包括：数字 0、对象 null、空字符串、NaN，以及 undefined。

所以，看来类型检查中涉及的语义规则，还真是挺丰富的。而要实现上面这些类型检查功能，关键点就是实现类型的计算。在上面的例子中，我们看到需要实现两个运算：**LE 运算**和 **overlap 运算**，我们又把它们叫做**子类型判断**和**重叠判断**。

子类型和重叠的判断

我们这里提到了一个术语，子类型。什么是子类型呢？如果 A 是 B 的子类型，那意味着 A 的每个成员，也都是 B 的成员。如果采用集合的术语，这意味着 A 集合是 B 集合子集。

在学习面向对象的时候，你肯定知道子类 and 父类的概念，它们之间的关系就是子类型关系。我们举一个例子：我们都知道“人”是“哺乳动物”的子类，那么我们可以说任何一个“人”，都肯定是一个“哺乳动物”。“人”的集合是“哺乳动物”集合的子集。

不过，除了面向对象的子类关系外，还有其他的子类型，**只要二者之间具有子集的关系就行**。比如：

类型 “0|1”，也就是只能取 0 和 1 两个值的类型，它是 number 的子类型，也是 “0|1|2” 的子类型；

类型 string，是 “string|number” 的子类型，也是 “string|null” 的子类型。

实现子类型判断，你可以参考 TypeUtil 类的 LE 方法。它里面的运算规则比较多，我挑重点的和你解释一下：

首先，我们要如何判断一个 NamedType 是另一个 NamedType 的子类型呢？

对于 string、number 和 boolean 这些基础类型来说，它们都是并列的，相互之间没有子类型的关系。不过，由于后面我们会讲到面向对象特性，而面向对象中的类型之间是有子类型关系的，所以我这里预先做了准备。

你看看 NamedType 类的设计，会发现它有一个 upperTypes 属性，这里就存了该类型的多个父类型。所以，基于这个 upperTypes 属性，父类和子类就被关联到了一起，我们的编译器也就能够基于此来判断一个类型是否是另一个类型的子类型。

而且，在 PlayScript 中，我提供了 Number 的两个子类型，分别是 Integer 和 Decimal，可以用来验证这个特性。在类型消解的时候，编译器会把整型字面量的类型标注为 Integer 的，而把浮点型字面量的类型标注为 Decimal 的。在赋值的时候，Integer 和 Decimal 类型的值，都可以赋给 number 类型的变量。

第二，如何判断一个值类型是否是某个 NamedType 的子类型呢？

这个问题比较简单，我们对每个值类型都记录了它所属的 NamedType 基础类型。比如，整数值类型是 Integer，所以整数的值类型一定是 Integer 的子类型。而 Integer 又是 number 的子类型，那么整数值类型也是 number 的子类型。

第三，如何判断一个 NamedType 或 ValueType 是 UnionType 的子类型呢？

你会看到，UnionType 中有一个 types 数组，代表了多个类型的联合。如果一个 NamedType 或 ValueType 是 UnionType 中任何一个元素的子类型，那么它就是该 UnionType 的子类型。

最后，我们如何判断一个 UnionType 是另一个 UnionType 的子类型呢？比如 “0|1” 是否是 “0|1|2” 的子类型，或者是否是 “number|string” 的子类型呢？

这就要求第一个 UnionType 类型的每个成员都得是第二个 UnionType 的子类型才可以。

好了，上面就是我们做子类型的检查的思路了。那第二个运算，检查类型之间是否有交集，或者说是否重叠，也可以借鉴类似的思路，算法上稍有区别。你参考下 TypeUtil 类的

overlap 方法就行。

在实现了子类型和 overlap 的检测以后，我们就能完成前面那些场景中的类型检查了。


不过，TypeScript 做类型计算的能力不止于此，它还有些更强大的能力，这会用到其他的集合运算，包括交集、并集和补集的计算。

让我们通过这些新的场景来探索一下。

交集、并集和补集的计算场景

我们还是回到这节课的第一个例子程序来分析分析。在这个例子中，编译器不允许把 age 赋值给 age1，因为 age 可能取值为 null，不能赋值给 “string|number”。

现在我们把这个例子改一下，加一句 “age = 18;”，然后再给 age1 赋值，看看会发生什么变化：

 复制代码

```
1 function foo6(age : number|null){
2     let age1 : string|number;
3     age = 18;        //age的值域现在变成了一个值类型：18
4     age1 = age;      //这里编译器不再报错。
5     console.log(age1);
6 }
```

你看看这个例子程序，现在 age 的值是 18。那么这时再把 age 的值赋给 age2，编译器还会不会报错呢？

你肯定不希望编译器报错，因为现在 age 的值是一个 number，不是 null，所以肯定是可以赋给 age1 的呀。

确实如你所愿，tsc 编译器这次没有再报错了。这个编译器真的是挺聪明的。

可是，这个编译器是如何做到这一点的呢？不是说 age 必须是 age1 的子类型吗？

原来，TypeScript 的编译器，结合数据流分析技术，随着程序的执行，可以动态地改变变量的值域，也就是取值范围。比如，在“age = 18”这句之后，age 的值域就变成了一个值类型 18。如果用我们这个新的类型来做类型检查，自然就不会出错。

那如果我们再给 age 赋一个新值，让它等于 null 会怎样呢？你可以参考下面的例子程序。

[复制代码](#)

```
1 function foo7(age : number|null){
2     let age1 : string|number;
3     age = 18;      //age的值域现在变成了一个值类型：18
4     age1 = age;    //OK。
5     age = null;    //age的值域现在变成了null
6     age1 = age;    //错误！
7     console.log(age1);
8 }
```

这个时候，age 的值域变成了 null。如果我们这个时候把 age 赋给 age1，那么编译器就会报错。

除了赋值语句、变量初始化语句能够改变变量的值域以外，if 语句中的条件，也会影响到变量的值域，你再看看下面的例子。

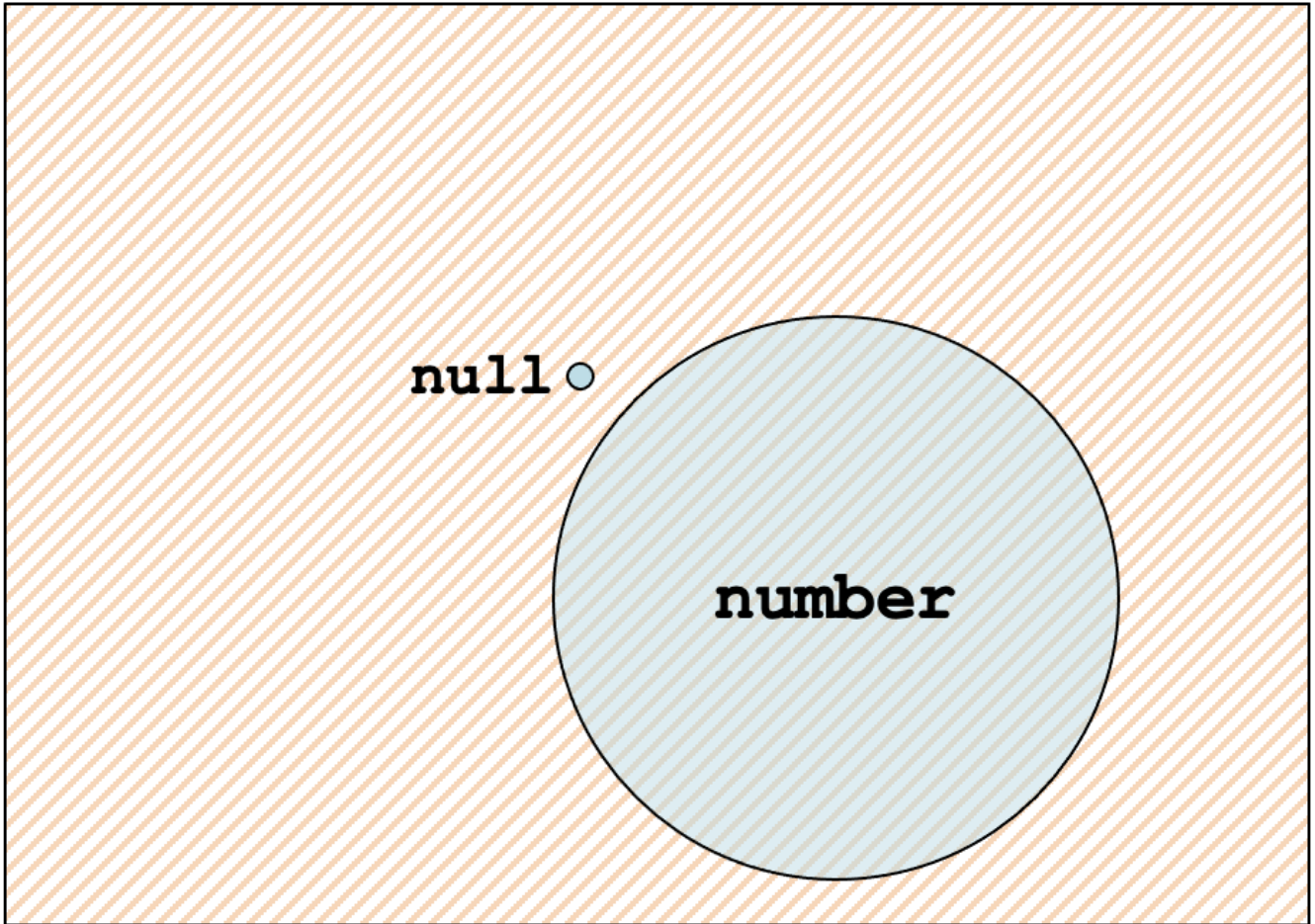
[复制代码](#)

```
1 function foo8(age : number|null){
2     let age1 : string|number;
3     if (age != null){    //age的值域现在是number
4         age1 = age;      //OK!
5         console.log(age1);
6     }
7     else{                //age==null, 值域现在变成了null
8         console.log("age is empty!");
9     }
10 }
```

在这个例子中，if 条件是“age!=null”。这个条件跟 age 原来的类型“number|null”相结合，就会求出 if 块中的 age 类型变成了“number”，把 null 这个选项去掉了。

这个过程是怎么实现的呢？

在这里，你可以把求 age 值域的过程看做是做集合运算的过程。在 if 条件中的表达式，会生成一个 age 的值域，这个值域是所有不等于 null 的值，我记做!null。这个值域跟原来 age 的值域 “number|null” 做交集运算，最后的结果就是 number。我画了一张示意图，表示交集运算的过程，你可以看一下：

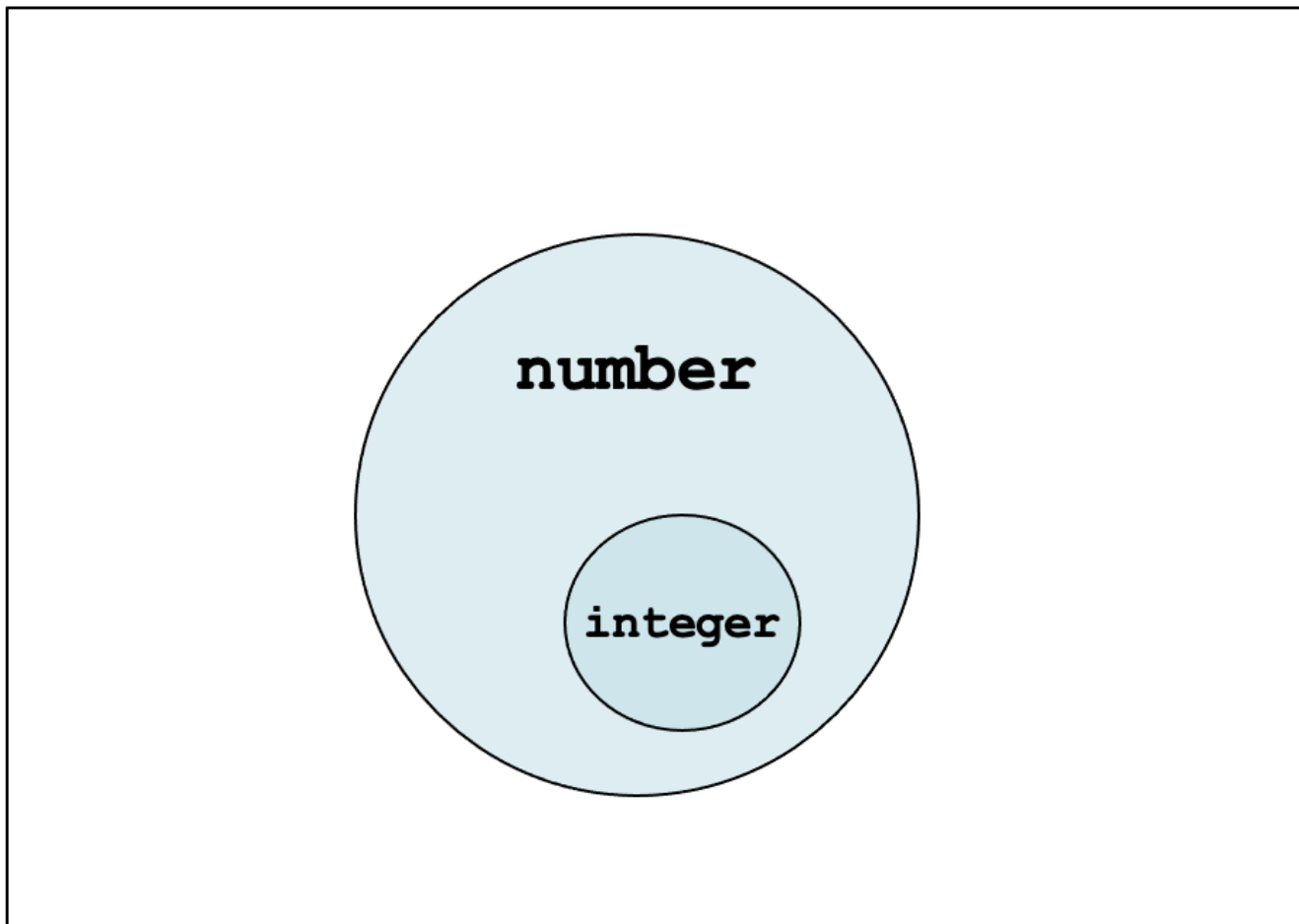


在这个示意图里，长方形的区域表示全集。全集里面有一个蓝色的圈，表示 number 集合。还有一个小点，表示 null 这个值，这两个部分都是蓝色的，代表了 “number|null”。

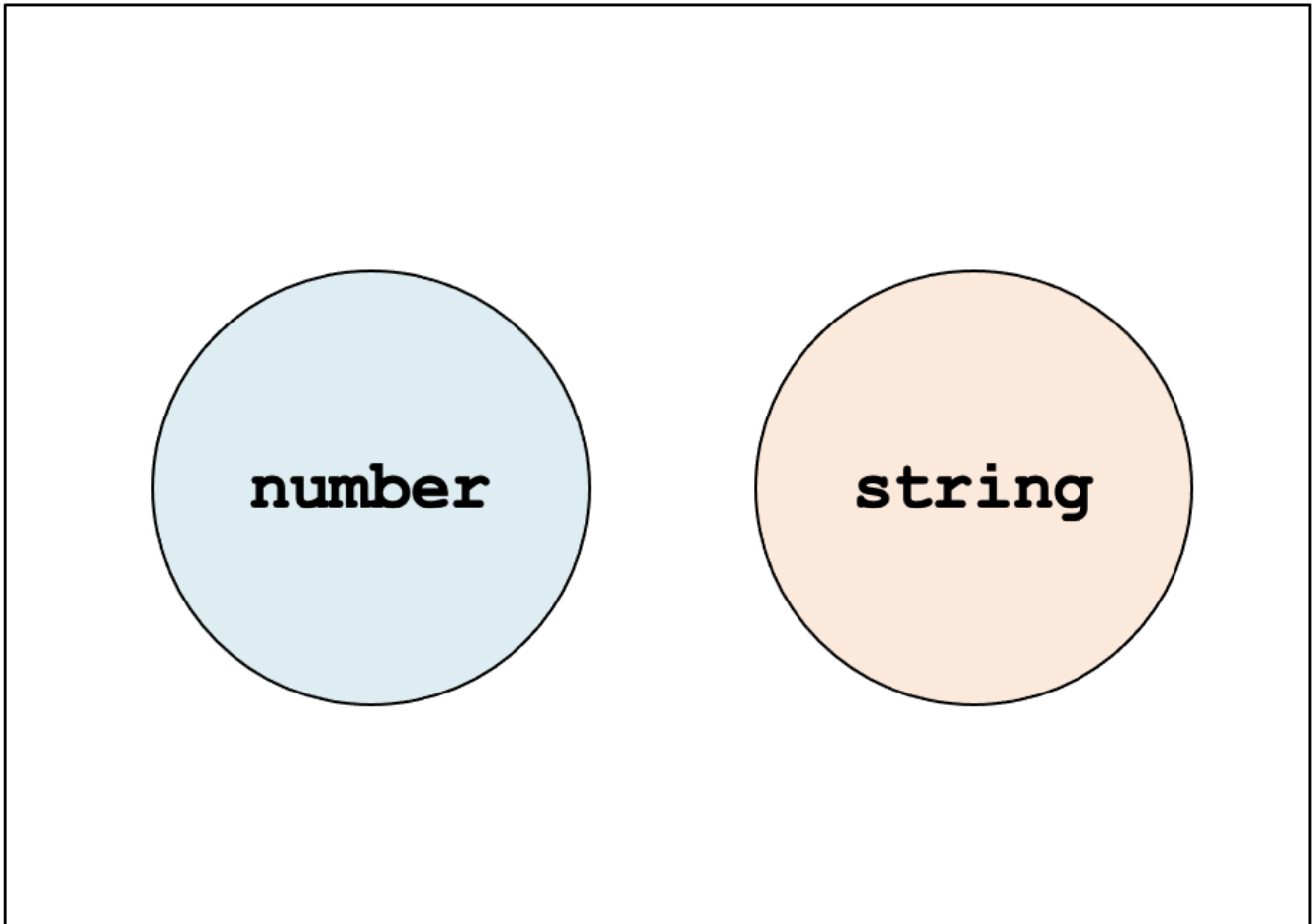
而打斜线的区域，是在全集中抠去 null 那个点后剩余的部分，是{null}的补集，我记做!null。这两个集合的交集，就是代表 number 的那个圆圈。所以，这个圆圈既带有蓝色，又打了斜线。

在这里，你会发现，我们的算法又需要支持两个集合运算。第一个运算，是**求交集运算**。在这里，我们可以先来简单总结一下交集运算的规则：

规则一：对于两个 NamedType，如果一个是一个的子类型，那么交集就是子类型。在下图中，number 和 integer 的交集是 integer。



规则二：如果两个类型之间没有子类型关系，那么它们的交集就是空集。就像下图中的 number 和 string。



在算法中，用什么来表示空集呢？在 PlayScript 的代码中，我用了一个特殊的 NamedType，叫做 Never，它对应了 TypeScript 中内置的类型 never。关于 never 类型的介绍，你可以参考 [TypeScript 手册中的内容](#)。

说完了 NamedType 之间求交集，你还可以进一步思考一下如何在 NamedType、ValueType 和 UnionType 之间互相求交集。总体上，遵循集合运算的规则就行了。

除了交集运算，还需要**求补集**。在前面的例子中，我们用到了 null 的补集。那如何表达 null 的补集呢？我用的方法，是在 ValueType 对象里加了一个 isComplement 属性。如果这个属性为 true，就代表这个值对象其实是该值的补集。

除了值对象有补集，其实 NamedType 也可以有补集。比如，你在 if 条件中可以放 "typeof age !== 'string'" 这样的语句，这时候 age 的值域就是 !string，也就是 string 的补集。

在上面的示例程序中，我们还有一个地方用到了补集，这就是计算 else 块中的值域的时候。在 else 块中，要对 if 条件生成的值域取补集，也就是把 !null 再做一次补集运算，得

到的结果就是 `null`。也就是说，在 `else` 块中，`age` 的取值肯定是 `null`。

现在我们就说完了求交集和补集这两个集合运算。不过，你可能马上又会想到，**在集合运算里还有求并集的运算呀，那在我们类型计算里是不是也有这个场景呢？**

有的。如果我们把 `if` 条件复杂化一点，用上逻辑运算 “`||`”，那么这个 `if` 条件形成的值域就是 `18|81`。这里就做了一次求并集的运算，把 `18` 和 `81` 两个值类型并在了一起。我们让这个并集再跟 “`number|null`” 做交集运算，结果仍然是 `18|81`。

[复制代码](#)

```
1 function foo9(age : number|null){
2     if (age == 18 || age == 81){ //age的值域是 18|81
3         console.log("18 or 81");
4     }
5     else{ //age的值域是 (number | null) & !18 & !81
6         console.log("age is empty!")
7     }
8 }
```

我们把这个例子再往下深化分析一下。如果 `if` 块中 `age` 的值域是 `18|81`，那么 `else` 块中是什么呢？那就是 `number|null` 中去掉 `18` 和 `81` 就可以了。

具体计算过程，是先对 `18|81` 求补集，也就是 `!(18|81) = !18 & !81`。其中 `&` 是交集的意思。然后再跟 `(number|null)` 求交集，得到的结果是 `(number|null) & !18 & !81`。

到这里，我们又必须引入另一个表示类型的对象，叫做 `IntersectionType`，用于表示多个类型之间的交集。交集类型中，多个成员之间使用 `&` 连接的。

并且，运用集合运算的知识，你还能意识到交集对象和联合对象之间是有转换关系的。`UnionType` 的补集，就是一个 `IntersectionType`，而 `IntersectionType` 的补集呢，则是 `UnionType`。

那到目前为止，我们用于表示类型的对象体系就更加完善了。我们增加了一个新的类型，是 `IntersectionType`。并且，我们还把 `ValueType` 和 `NamedType` 都加上了是否是补集的属性。

好了，求集合的补集、交集和并集等运算我们都讲过了。但对于刚才这个例子。我们还需要用数据流分析方法动态地求变量的值域。不过，今天的新知识点已经足够多了，我们还是把这个任务放到下一节课。在下一节课，我们会综合运用多种语义分析技术，来获得更强大的效果。

课程小结

在今天这节课，我们举了多个例子，示范了多个类型计算的场景。在这个过程中，我们接触到了个多种集合运算，包括子集判断、重叠判断、求补集、求交集和求并集。

第一，子集判断的典型场景是赋值运算。比如 $x=y$ 语句，要求 y 的类型和 x 的类型相等，或者 y 的类型是 x 的类型的子集。如果 y 的类型是 x 的类型的子集，我们可以简单地说 y 是 x 的子类型。面向对象编程中的继承关系就是子类型的一种体现。

第二，重叠判断的典型场景是 $==$ 和 $!=$ 运算符。它们要求两边的类型有重叠的部分。如果没有重叠的部分，编译器就会报错。

第三，并集的使用场景，是 if 条件中带有逻辑运算 $||$ 的情况，在下一节你还会看到另一个使用场景。

第四，交集的使用场景，一个是 if 条件中有逻辑运算 $\&\&$ 的情况，另一个是 if 条件中得到的值域，与变量原来的值域求交集，得到 if 块中变量的值域。

第五，补集有多个使用场景，一个是针对 $!=$ 运算符，第二个是 if 条件中使用 $!$ 运算符，还有一个是求 $else$ 块中的变量值域时。如果对 $UnionType$ 求补集，我们会得到一个 $IntersectionType$ 。

这节课的例子，你都可以通过 `node play example_type2.ts` 来运行。关于类型计算的实现，可以参考 `TypeChecker` 和 `TypeUtil` 两个类。

我们这节课的内容，全面地运用了集合计算，这也是 TypeScript 具备强大的类型处理能力的原因。很多现代语言的类型处理能力现在也变得越来越强，这背后的数学知识都是集合运算。所以，你一定要重视这个知识点。你要学会像这节课这样，分析在各种场景下，编译器到底是如何使用集合运算来做类型处理的。

思考题

在这节课，我们涉猎了很多集合运算的知识点，让整个类型处理变成了一个自治的体系。今天的思考题，我们继续把这种类型计算的方式跟其他语言做一下对比。你在其他语言做过对类型做交集、并集、补集、判断子集和判断重叠的运算吗？欢迎在留言区分享你的发现。

欢迎你把这节课分享给更多对类型计算感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

1. 这节课的示例代码目录在 [🔗 这里](#)！
2. 主要看语义分析的代码 ([🔗 semantic.ts](#)) 和类型体系的代码 ([🔗 types.ts](#))。
3. 例子代码：[🔗 example_type2.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 [23 | 增强编译器前端功能第2步：增强类型体系](#)

下一篇 [25 | 增强编译器前端功能第4步：综合运用多种语义分析技术](#)

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 (2)

写留言



blackonion

2021-10-12

rust也可以，例如`let mut maybe_some = None ; maybe_some = Some(666)` 编译器可以算出`maybe_some`的类型是`Option<i32>`。

展开



奋斗的蜗牛

2021-10-08

之前看typescript编译器的类型检验代码，看得很晕，里面的类型推导是真复杂，谢谢老师讲到这块

