Atitit 编程语言类型系统概念

目录

[类型系统 1](#_Toc22666)

[类型检查 2](#_Toc7501)

[Implicit type conversions 3](#_Toc13184)

[Pointers 3](#_Toc10528)

[Untagged unions 4](#_Toc11915)

[Weekly typed 4](#_Toc32409)

[强类型和弱类型 4](#_Toc29680)

[类型推断 5](#_Toc4113)

[类型理论 6](#_Toc301)

[Polymorphism type 6](#_Toc28556)

[Dependent types 12](#_Toc26821)

[Linear types 12](#_Toc21452)

[Intersection types 13](#_Toc25494)

[Union types 14](#_Toc715)

[Existential types 16](#_Toc13791)

[Upper Bounded Wildcards 16](#_Toc13625)

[Lower Bounded Wildcards 18](#_Toc19797)

## **类型系统**

什么是类型？在软件执行的过程中，变量可以为很多值， ****定义变量的边界的描述即类型**** 。变量可以被赋予类型(即变量有边界)的语言称为类型语言( ****typed language**** )，无类型语言( ****untyped language**** )没有类型，或者说只有一个全局类型，能够存储所有的值。 类型语言我们见得多了，无类型的呢？lambda演算(pure λ-calculus)是无类型的，汇编和LISP也是无类型的.

变量类型的指定可以是显式的

// golang

var foo int

也可以是隐式的

-- haskell

fac :: Int -> Int -- 这一行可以省略

fac 0 = 1

fac n = n \* fac (n - 1)

fac 0 = 1

fac n = n \* fac (n - 1)

类型系统会自动赋予变量类型，这种行为称为类型推断(type inference)。

### **类型检查**

类型系统是类型语言的首要组成部分。类型系统的职责之一是跟踪变量的类型，判断代码是否满足类型约束，这种行为称为类型检查( ****typechecking**** ), 类型检查是保证程序稳定运行的手段，同时又分为运行时检查(runtime checks)和静态检查(static checks), 运行时检查也叫动态检查( ****dynamic checking**** ).

类型系统做了静态检查，还有必要做动态检查嘛？有，比如数组的边界检查，就必须在运行时做。运行时的类型检查会导致程序运行终止(fail-stop)，那为什么还要检查呢？让它运行到无法继续执行为止不就好了？类型检查虽然会出错，但是阻止了更恶劣的错误(untrapped errors)的发生，比如保证gc等机制能够正常运转，让程序能够更平滑地退出。动态检查的缺点是会导致fail-stop，也会消耗资源，影响性能，所以通常我们认为拥有静态检查的类型系统的语言会更稳定高效。但是静态检查就足够安全了吗？不一定，因为某些语言在静态检查时没有检查一些危险操作，比如 *C* 语言中指针的运算和转换，这类语言称为 ****weekly checked**** , 反之, 程序在编译期间能够尽可能发现所有的类型错误, 称为 ****strongly checked**** .

那么延伸一下，怎么区分一门语言是 *weekly checked* , 还是 *strongly checked* ? 以下几点可以作为判断的依据。

### **Implicit type conversions**

可以进行隐式类型转换的语言属于 *weekly checked* , 如c++

int a = 3;

double b = 4.5;

a + b; // a将会被自动转换为double类型，转换的结果和b进行加法操作

### **Pointers**

允许指针运算的语言属于 *weekly checked* , 比如c

int num [] = {1,3,6,8,10,15,22};

int \*pointer = num;

printf("\*pointer:%d\n",\*pointer);

pointer++;

printf("\*pointer(p++):%d\n",\*pointer);

### **Untagged unions**

*union type* 即联合类型，之后的内容会介绍。联合类型中的不同类型的值会被存储在同一地址，这也是不稳定因素之一，所以拥有 *untagged unions* 的语言属于 *weekly checked* . 和 *untagged* 相对的是 *tagged union type* , *tagged union* 会用tag字段显式地标记当前正在使用的类型，因此要相对安全一些。

### **Weekly typed**

一般弱类型语言属于 *weekly checked*

除此之外，往往是通过语言之间的比较来进行判断，并没有明显的界限。类型系统除了提供类型检查，保证程序的安全性之外，还可以提供信息给编译器，以优化执行效率。同时，类型系统是对现实世界的抽象，可读性高，也是更高级别抽象(如泛型，\*\*OOP\*\*s)的基础。所以，无类型的语言在选型时基本会被排除掉。

### **强类型和弱类型**

在语言的抉择上， 静态检查 ， 动态检查 和 检查范围 这几个角度影响的是程序的稳定性和执行效率，那么开发效率呢？此时要引入另外一个维度， 强类型 和 弱类型 ，强类型是指一旦被指定类型，不可变，弱类型则可变，可变也分为隐式和显式两种。

// js 弱类型，隐式可变

x = 1

y = "2"

z = x + y

弱类型提升了我们的开发效率。

总结，在语言的抉择上，我们可以从类型系统的静态检查，动态检查，检查范围及类型是否可变这几个维度来考虑， ****在开发效率，执行效率及安全性之间做权衡**** 。无类型的语言可以很安全，但是可维护性差，基本被排除在系统开发之外。

看到这里，你会发现文中并没有讲什么是动态类型语言，静态类型语言。因为这个维度并不是影响我们抉择的本质因素。动态类型语言其实指的就是只有动态检查的语言，静态类型语言指拥有静态检查的语言，此类型非彼类型，个人觉得是翻译的锅。

### **类型推断**

类型推断(type inference)是类型系统中推测一段代码(声明，表达式等)的类型的过程。类型推断让我们在编写代码时能够略去类型的声明，并且不会影响到类型检查。如果值的类型仅在运行时才能被确定，这类语言被称为动态语言(dynamically typed), 同样的，如果值的类型仅在编译时被确定，这类语言被称为静态语言(statically typed).

## **类型理论**

类型理论涉及到的内容和类型系统会有部分重叠，你可以理解为，类型理论是服务于类型系统的，即，我们使用了哪些理论来构建我们的类型系统，或者说该语言的类型系统实现了哪些特性。一门语言的类型系统一般只会实现其中的某几种特性，这也是语言之争的根源。

### **Polymorphism type**

polymorphism翻译为多态性，但不单单指面向对象里的多态，而是指类型系统里的多态性质。编译时多态，是在编译时就能推导出类型或调用关系，宏也是一种编译时多态。运行时多态的实现依赖于虚函数机制(virtual function), 是在运行时确定调用关系。多态性质的引入可以提高代码的复用率。

*Ad hoc polymorphism* : 一个函数会根据有限的类型组合而拥有不同的实现，函数重载(function overloading)和操作符重载(operator overloading)依赖于此. 从polymorphism性质实现的角度讲，属于编译时多态(static polymorphism).

// java

public int Add(int a, int b) {

return a + b;

}

public String Add(String a, String b, String c) {

return a + b +c;

}

*Parametric polymorphism* : 声明的类型未被指定为某一类型，而在实现时可以指定为任意类型，即通常我们所说的泛型，在C++里称为模板(template). 从polymorphism性质实现的角度讲，属于编译时多态(static polymorphism).

// java

...

public class ObjectsServiceFactory<T> {

public T save(T o) throws Exception {

try {

return repository.save(o);

} catch (Exception e) {

throw new DatabaseOperationException(String.format("save object %s failed, %s", o.toString(), e.getMessage()));

}

}

public void delete(T o) {

repository.delete(o);

}

}

...

ObjectsServiceFactory s = new ObjectsServiceFactory<String>();

ObjectsServiceFactory i = new ObjectsServiceFactory<Integer>();

s.save("demo");

i.save(100)

*Subtype polymorphism* : 也叫subtyping, 一个类型的变量可以指代 ****多个该类的子类实例**** ，即我们常说的多态。从多态性质实现的角度讲，此类属于运行时多态(dynamic polymorphism).

// c++

#include<iostream>

using namespace std;

class Animal {

public :

virtual void shout() = 0;

virtual ~Animal(){}

};

class Dog :public Animal {

public:

virtual void shout(){ cout << "dog"<<endl; }

};

class Cat :public Animal {

public:

virtual void shout(){ cout << "cat"<<endl; }

};

class Bird : public Animal {

public:

virtual void shout(){ cout << "bird"<<endl; }

};

int main() {

Animal \* animal1 = new Dog;

Animal \* animal2 = new Cat;

Animal \* animal3 = new Bird;

animal1->shout();

animal2->shout();

animal3->shout();

delete(animal1);

delete(animal2);

delete(animal3);

return 0;

}

*Row polymorphism* : 也叫duck typing，针对结构体类型，从功能(purpose)的角度对类型归类。通常，对象是根据它们的类型来确定彼此之间的关系，比如subtyping中的父类/子类关系，而duck typing是通过函数，如果它们实现了相同的函数，就认为它们是同一类。

If it walks like a duck and it quacks like a duck, then it must be a duck.

如果它走路像鸭子(实现了walk()函数)，也会像鸭子一样发出嘎嘎声(实现了gaga())函数，那它就是一只鸭子(属于同一类型)。

# python

class Duck:

def fly(self):

print("Duck flying")

class Airplane:

def fly(self):

print("Airplane flying")

class Whale:

def swim(self):

print("Whale swimming")

def lift\_off(entity):

entity.fly()

duck = Duck()

airplane = Airplane()

whale = Whale()

lift\_off(duck) # prints `Duck flying`

lift\_off(airplane) # prints `Airplane flying`

lift\_off(whale) # Throws the error `'Whale' object has no attribute 'fly'`

duck typing也是go语言的主要特性，但是严格来说并不算，因为duck typing发生在运行时，且没有显式的 ****interface**** 声明，上面的python示例就是典型的duck typing

*Polytypism* : 函数式编程语言里的泛型特性。以Haskell为例，其函数的定义比较具体化，单一化，缺乏可扩展性和高度复用性，在Haskell语言上可以引入一种泛型机制解决上述问题，这种泛型机制主要体现在泛型函数的定义上，泛型函数的定义不同于以往的函数定义方法，当泛型函数遇到某种未定义的类型参数时，它依靠泛型算法分析参数类型的结构，进行相关转换，可以自动生成函数定义，这种方法可以提高程序的复用程度。 [2]

### **Dependent types**

依赖类型（或依存类型，dependent type）是指依赖于值的类型, 此特性通过极其丰富的类型表达能力使得程序得以借助类型的形式被检查，从而有效减少程序错误。依赖类型的两个常见实例是依赖函数类型(又称 ****依赖乘积类型**** , ****Π-类型**** )和依赖值对类型(又称 ****依赖总和类型**** 、 ****Σ-类型**** )。[4]

一个依赖函数的返回值的类型可以依赖于某个参数的具体值，而非仅仅参数的类

## **Linear types**

Linear types的思想来源于Linear Logic, 它确保对象在程序运行期间有且仅有一个它的引用，这种类型用来描述不能被修改的值，比如文件描述符。linear 类型系统允许引用，但不允许别名(被多个变量引用), 类似于C++的 *unique\_ptr* 指针, 只能被移动，不能被复制。

### **Intersection types**

一个intersection type(交叉类型)是多个type的结合, 以此，你能够得到一个包含 ****多个类型**** 的所有成员(members)的新类型！比如，现有三个类Person, Serializable 和 Loggable, 新的类型 T = Person & Serializable & Loggable, 那么类型T拥有Person，Serializable及Loggable的所有成员。

// TypeScript mixin example

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() {

console.log("papapa!");

}

}

var jim = extend(new Person("Jim"), new ConsoleLogger());

var n = jim.name;

jim.log();

### **Union types**

学过C语言的对此类型并不陌生，和intersection type类似，一个union type可以为多个类型，但是在任意时刻，它的值的类型只能是其中所有类型中的一种。

//c

union a\_bc {

int i;

char mm;

};

*TypeScript* 里的联合类型用竖线 | 来分隔每个类型，所以 value : number | string | boolean 表示一个值可以是 number ，或 string ，或 boolean 。

/\*\*

\* TypeScript

\* 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 side.

\*/

function padLeft(value: string, padding: string | number) {

if (typeof padding === "number") {

return Array(padding + 1).join(" ") + value;

}

if (typeof padding === "string") {

return padding + value;

}

}

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

let ok = padLeft("Hello world", 0) // compile ok

### **Existential types**

在理解存在类型(existential type)之前，我们先看下java的类型通配符 ? , 它代表一个未知类型.

### **Upper Bounded Wildcards**

通过声明通配的上限(父类)来匹配，如果你的函数入参可能是List\, List\或者 List\ ，你可以使用? 声明

public static void add(List<? extends Number> list)

Number的所有子类都可以作为入参, 例如:

//Java program to demonstrate Upper Bounded Wildcards

import java.util.Arrays;

import java.util.List;

class WildcardDemo

{

public static void main(String[] args)

{

//Upper Bounded Integer List

List<Integer> list1= Arrays.asList(4,5,6,7);

//printing the sum of elements in list

System.out.println("Total sum is:"+sum(list1));

//Double list

List<Double> list2=Arrays.asList(4.1,5.1,6.1);

//printing the sum of elements in list

System.out.print("Total sum is:"+sum(list2));

}

private static double sum(List<? extends Number> list)

{

double sum=0.0;

for (Number i: list)

{

sum+=i.doubleValue();

}

return sum;

}

}

省略写法也是 *Upper Bounded*

Collection<?> c = new ArrayList<String>();

c.add(new Object()); // compile error

当类型不可推断时，上限是 *Object* 即

List<? extends Object>

### **Lower Bounded Wildcards**

通过声明通配的下限(子类)来匹配, 比如函数的入参声明为:

List<? super Integer>

那么，Integer类型的所有父类都可以作为入参。例如:

//Java program to demonstrate Lower Bounded Wildcards

import java.util.Arrays;

import java.util.List;

class WildcardDemo

{

public static void main(String[] args)

{

//Lower Bounded Integer List

List<Integer> list1= Arrays.asList(4,5,6,7);

//Integer list object is being passed

printOnlyIntegerClassorSuperClass(list1);

//Number list

List<Number> list2= Arrays.asList(4,5,6,7);

//Integer list object is being passed

printOnlyIntegerClassorSuperClass(list2);

}

public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list)

{

System.out.println(list);

}

那java的类型通配符和存在类型有什么关系呢？先看看scala的缔造者 *Martin Odersky* 对scala引入存在类型的回答

Bill Venners: Existential types were added to Scala relatively recently. The justification I heard for existentential types was that they allow you to map all Java types, in particular Java's wildcard types, to Scala types. Are existential types larger than that? Are they a superset of Java's wildcard types? And is there any other reason for them that people should know about?

Martin Odersky: It is hard to say because people don't really have a good conception of what wildcards are. The original wildcard design by Atsushi Igarashi and Mirko Viroli was inspired by existential types. In fact the original paper had an encoding in existential types. But then when the actual final design came out in Java, this connection got lost a little bit. So we don't really know the status of these wildcard types right now.

*Martin Odersky* 在scala邮件组里的回答

The original Java wildcard types (as described in the ECOOP paper by Igarashi and Viroli) were indeed just shorthands for existential types. I am told and I have read in the FOOL ’05 paper on Wild FJ that the final version of wildcards has some subtle differences with existential types. I would not know exactly in what sense (their formalism is too far removed from classical existential types to be able to pinpoint the difference), but maybe a careful read of the Wild FJ paper would shed some light on it.

可见，java类型通配符的设计思想来源于存在类型，但有些不同。scala引入存在类型是为了兼容java和jvm，所以会比 ? 更强大。那么既然区别不大，理解了类型通配即理解了存在类型。这是一个曲线救国的方式。

接下来，我们从数学定义的角度来理解。

*existential type* 里的existential来源于存在量词 ∃ , ∃ 在谓词逻辑中的解释:

∃ x: P(x) 表示存在至少一个 x 使得 P(x) 为真。

存在类型的公式化表示:

T = ∃X { X a; int f(X); }

类型X是存在类型, 即存在一个类型X，满足此表达式，在编程语言里我们称之为 ****可实现**** 。存在类型适合用来定义接口，不论是模块之间还是语言之间。

这里要提下泛型(即前面讲到的 *Parametric polymorphism* , 也叫 *Universal type* ), 以避免混淆。*Universal type* 中的universal来源于全称量词 ∀ , ∀ 在谓词逻辑中的解释:

∀ x: P(x) 表示 P(x) 对于所有 x 为真。

泛型的公式化表示:

T = ∀X { X a; int f(X); }

即，对于所有类型X，满足此表达式。