编程语言选型时，你需要考虑的几个方面

目录

[1.1. 看完这篇文章能收获什么? 4](#_Toc10180)

[1.2. 类型系统 5](#_Toc31113)

[1.2.1. 类型检查 6](#_Toc16220)

[1.2.2. Implicit type conversions 6](#_Toc26864)

[1.2.3. Pointers 7](#_Toc13752)

[1.2.4. Untagged unions 7](#_Toc710)

[1.2.5. Weekly typed 7](#_Toc4506)

[1.2.6. 强类型和弱类型 8](#_Toc4759)

[1.2.7. 类型推断 9](#_Toc6373)

[1.3. 类型理论 9](#_Toc29977)

[1.3.1. Polymorphism type 9](#_Toc12884)

[1.3.2. Dependent types 15](#_Toc26512)

[1.4. Linear types 16](#_Toc6594)

[1.4.1. Intersection types 17](#_Toc876)

[1.4.2. Union types 18](#_Toc9177)

[1.4.3. Existential types 20](#_Toc20418)

[1.4.4. Upper Bounded Wildcards 20](#_Toc29737)

[1.4.5. Lower Bounded Wildcards 22](#_Toc24453)

[1.5. 语言规范 25](#_Toc24811)

[1.5.1. Types 25](#_Toc27320)

[1.5.2. Primitive types 25](#_Toc23047)

[1.5.3. Compound types 26](#_Toc486)

[1.5.4. Statements 27](#_Toc5394)

[1.5.5. Empty statement 27](#_Toc13988)

[1.5.6. Labeled statement 28](#_Toc3672)

[1.5.7. Expression statement 30](#_Toc10481)

[1.5.8. If statement 30](#_Toc12087)

[1.5.9. Assert statement 30](#_Toc10404)

[1.5.10. Switch statement 30](#_Toc11347)

[1.5.11. While statement 32](#_Toc21328)

[1.5.12. Do statement 33](#_Toc621)

[1.5.13. For statement 34](#_Toc6725)

[1.5.14. For-in statement 34](#_Toc28385)

[1.5.15. For-of statement 35](#_Toc14740)

[1.5.16. For-range statement 36](#_Toc26898)

[1.5.17. Break statement 37](#_Toc12358)

[1.5.18. Continue statement 38](#_Toc3156)

[1.5.19. Return statement 41](#_Toc2844)

[1.5.20. Throw statement 41](#_Toc7703)

[1.5.21. Goto statement 42](#_Toc7838)

[1.5.22. Operators 44](#_Toc31318)

[1.5.23. Unary operators 44](#_Toc29241)

[1.5.24. Binary operators 44](#_Toc16540)

[1.5.25. Ternary operators 45](#_Toc28095)

[1.5.26. Expressions 46](#_Toc11807)

[1.5.27. Variable expression 46](#_Toc9103)

[1.5.28. Arithmetic expression 46](#_Toc13760)

[1.5.29. Relational expression 46](#_Toc9501)

[1.5.30. Function expression 46](#_Toc11500)

[1.5.31. Index expression 46](#_Toc31935)

[1.5.32. Scoping 47](#_Toc23735)

[1.5.33. Expression scope 47](#_Toc32490)

[1.5.34. Block scope 47](#_Toc172)

[1.5.35. Function scope 48](#_Toc30275)

[1.5.36. File scope 49](#_Toc18717)

[1.5.37. Module scope 49](#_Toc2340)

[1.5.38. Global scope 49](#_Toc25852)

[1.5.39. Packages 49](#_Toc27801)

[1.6. 编程范式 52](#_Toc30601)

[1.6.1. IP(Imperative Programming) 52](#_Toc22482)

[1.6.2. SP(Structured Programming) 53](#_Toc16469)

[1.6.3. PP(Procedure Programming) 54](#_Toc1187)

[1.6.4. DP(Declarative Programming) 55](#_Toc12390)

[1.6.5. LP(Logic Programming) 56](#_Toc11403)

[1.6.6. FP(Functional Programming) 56](#_Toc15748)

[1.6.7. FRP(Functional Reactive Programming) 57](#_Toc12110)

[1.6.8. MP(Meta Programming) 57](#_Toc3326)

[1.7. 依赖管理 58](#_Toc14135)

[1.8. 标准库 58](#_Toc6491)

[1.9. 选型参考 58](#_Toc4210)

[1.9.1. 术语说明 59](#_Toc21208)

[1.10. 参考文献 63](#_Toc21530)

[1.10.2. 最新文章 92](#_Toc30887)

[1.10.3. 归档 92](#_Toc19705)

[1.10.4. 热门文章 94](#_Toc6468)

[1.10.5. 最新评论 95](#_Toc27406)

[1.10.6. 目录 96](#_Toc23138)

[1.10.7. 分类专栏 101](#_Toc1092)

## **看完这篇文章能收获什么?**

* 对于开发人员，可以系统地了解各种类型特性，语法特性和编程范式
* 对于架构师或者技术leader，可以为编程语言的选型提供理论依据

编程语言往大了讲是很复杂的，语法，类型系统，编译原理，编译器，解释器，内存模型，并发模型，工具链等等，单拿一点出来都能写一本书。限于篇幅，本文要介绍的内容限定在类型系统(type systems)，语法(syntax)和编程范式(programming paradigms)上。

## **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，而非仅仅参数的类型，例如，一个输入参数为整型值n的函数可能返回一个长度为n的数组

// Idris

// 连接两个列表

// Vect n a 是依赖函数类型，a是列表元素的类型，n是输入参数，Vect n a 返回一个长度为n的列表

app : Vect n a -> Vect m a -> Vect (n + m) a

一个依赖值对类型中的第二个值可以依赖于第一个值，例如，可表示一对这样的值：它由一对整数组成，其中的第二个数总是大于第一个数。

def do(i : {i:Int | i<=j}, j : Int) :=

// do something

do(10, 1) // compile error

do(1, 10) // ok

以依赖类型系统为基础的编程语言大多同时也作为构造证明与可验证程序的辅助工具而存在，如 Coq 和 Agda（但并非所有证明辅助工具都以类型论为基础)。近年来，一些以通用和系统编程为目的的编程语言被设计出来，如 Idris。

## **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。

## **语言规范**

在语言学里有三个基本概念(也是三个分支)， ****syntax**** , ****semantics**** , ****pragmatics**** . 即语法，语义(编译结果)和语用(最佳实践, 标准库, 生态)。这里主要讲语法层面。

### **Types**

一门语言的规范，首先是类型(type), 声明(statements), 表达式(expressions)等， 然后是作用域(scoping)。前面的内容介绍了类型系统, 那么该类型系统定义了哪些类型，实现了哪些特性是我们首先要了解的。通常一门语言的语法规范会以grammar [6] 的形式来定义，例如golang中对于浮点数字的描述:

float\_lit = decimals "." [ decimals ] [ exponent ] |

decimals exponent |

"." decimals [ exponent ] .

decimals = decimal\_digit { decimal\_digit } .

exponent = ( "e" | "E" ) [ "+" | "-" ] decimals .

这段文本描述了浮点数字的书写规则，也是一棵语法树。 *grammar* 对于没有学过的人来说有些难懂，所以本文中用类型实例的方式表示。

### **Primitive types**

即基础类型, 或者叫内置(builtin)类型, 是程序以及 [复合类型](https://www.tuicool.com/choose-a-programming-language/zh.wikipedia.org/wiki/%E8%A4%87%E5%90%88%E9%A1%9E%E5%9E%8B) 的创建基础。



### **Compound types**

即复合类型， 也叫\*Composite type\*s. 复合类型可以由基础类型和复合类型所构成。



### **Statements**

statement的直译是声明，但在这里按照代码的逻辑单元来理解，一个statement是逻辑单元的开始或结束。在书籍中，通常描述为xxx语句，比如 ****if**** 语句。

### **Empty statement**

空语句, 不做任何事情。grammar规则:

EmptyStatement:

;

### **Labeled statement**

标签语句, 通常作为 ****goto**** , ****break**** , ****continue**** 的目标，例如c/c++里的 ****goto**** 语句，一个单词加上冒号即是 *labeled statement* 。下面代码中的标签语句 *LOOP:* 即是 ****goto**** 的目标。

/\*c\*/

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

LOOP:do {

if( a == 15) {

/\* skip the iteration \*/

a = a + 1;

goto LOOP;

}

printf("value of a: %d\n", a);

a++;

}while( a < 20 );

return

它的grammar规则为:

LabeledStmt = Label ":" Statement .

Label = identifier .

*labeled statement* 作为 ****goto**** 的目标大家应该见的很多，这里再举一个作为 ****continue**** 的目标的例子，便于理解。

// golang

guestList := []string{"bill", "jill", "joan"}

arrived := []string{"sally", "jill", "joan"}

CheckList:

for \_, guest := range guestList {

for \_, person := range arrived {

fmt.Printf("Guest[%s] Person[%s]\n", guest, person)

if person == guest {

fmt.Printf("Let %s In\n", person)

continue CheckList

}

}

}

### **Expression statement**

某些 *expression* (表达式), 比如赋值，函数调用等，可以作为一个 *statement* ，称之为 *expression statement*

### **If statement**

我们常说的 ****if**** 条件语句，也叫 ****If-then-else**** , 如果条件满足则执行此逻辑, 否则执行它的 ****else****(如果存在)逻辑.

// golang

if x > max {

x = max

}

### **Assert statement**

断言语句， ****assert**** 加真假值表达式, 如果表达式结果为 *false* , 程序退出。

// c

assert( size <= LIMIT );

不是所有语言都有断言语句.

### **Switch statement**

switch条件语句，判断表达式的值，满足不同的条件执行时执行不同的逻辑, 当所有条件都不满足时，执行默认逻辑(如果存在).

switch(expression) {

case n:

code block

break;

case n:

code block

break;

default:

code block

}

对于golang, *switch statement* 分成了两类, 一类是常规的 *expression switch* (上面的例子), 一类是 *type switch* (下面的例子)

// golang type switch

switch i := x.(type) {

case nil:

printString("x is nil") // type of i is type of x (interface{})

case int:

printInt(i) // type of i is int

case float64:

printFloat64(i) // type of i is float64

case func(int) float64:

printFunction(i) // type of i is func(int) float64

case bool, string:

printString("type is bool or string") // type of i is type of x (interface{})

default:

printString("don't know the type") // type of i is type of x (interface{})

}

### **While statement**

while循环语句, 重复判断bool表达式的值，如果为真则执行，直到表达式的值为假

// c

while(condition)

{

statement(s);

}

tip: 在golang里， ****for**** 语句加上bool表达式，可以实现 ****while**** 语句

// golang

for {

// code block

}

for a < b {

f()

}

### **Do statement**

do语句，和 ****while**** 配合使用，先执行一次逻辑，再判断条件

// c

#include <stdio.h>

int main(void){

int sum=0,i;

scanf("%d",&i);

do{

sum=sum+i;

i++;

}

while(i<=10);

printf("sum=%d",sum);

return 0;

}

### **For statement**

for语句, 一般由三部分组成，初始化表达式，条件表达式和一个末尾表达式(post statement)，初始化表达式只在开始时执行一次，条件表达式用来判断本次执行是否退出，末尾表达式在每次 ****for**** 语句的代码逻辑执行完后都会执行。

// golang

for i := 0; i < 10; i++ {

f(i)

}

### **For-in statement**

for语句常用来迭代，于是出现了多个变种， ****for-in**** 常见于脚本语言，如 *TypeScipt* ， *Groovy* , 用于迭代可枚举的(enumerable types)数据类型

// JavaScript

var person = {fname:"John", lname:"Doe", age:25};

var text = "";

var x;

for (x in person) {

text += person[x] + " ";

}

这里，x被赋值为person这个map的key。当遍历的对象是map时，x为key，当遍历的对象是数组时，x为索引下标

### **For-of statement**

for语句变种, 类似于 ****for-in**** , 用来迭代可迭代的(iterable)数据类型, 比如数组和字符串

// JavaScript

function\* foo(){

yield 1;

yield 2;

}

for (let v of foo()) {

console.log(v);

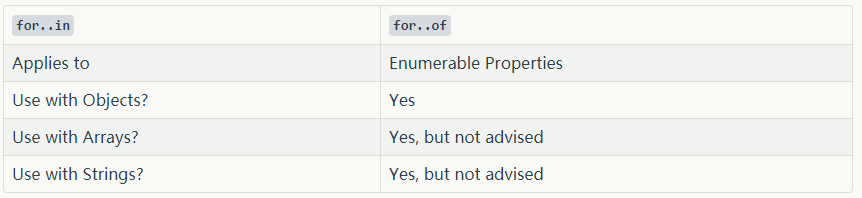
// expected output: 1

break; // closes iterator, triggers return

}

这里变量v被赋值为foo函数返回的对象。被赋值的变量总是可迭代类型里的元素。

For-invs ****For-of****



### **For-range statement**

for语句变种，在golang中用来迭代数组, 或者map

// golang

nums := []int{2, 3, 4}

sum := 0

for \_, num := range nums {

sum += num

}

kvs := map[string]string{"a": "apple", "b": "banana"}

for k, v := range kvs {

fmt.Printf("%s -> %s\n", k, v)

}

### **Break statement**

跳出语句，用于立即跳出一个逻辑单元，当不配合 *labeled statement* 使用时，立即(abruptly)跳出最里层的一个封闭(enclosing)逻辑单元, 如 ****switch**** , ****do**** , ****while**** , ****for**** . 当配合 *labeled statement*使用时，立即跳出label标定的层级的封闭逻辑单元。

// c

#include <stdio.h>

int main ()

{

/\* 局部变量定义 \*/

int a = 10;

/\* while 循环执行 \*/

while( a < 20 )

{

printf("a 的值： %d\n", a);

a++;

if( a > 15)

{

/\* 使用 break 语句终止循环 \*/

break;

}

}

return 0;

}

*labeled break* 已在 *labeled statement* 中说明，这里不再赘述.

### **Continue statement**

continue语句，立即结束当前层级的逻辑，跳转到 ****for**** 循环的末尾表达式，开始下一次循环.

// golang

package main

import "fmt"

func main() {

rows := []int{1, 3, 5}

colunms := []int{2, 4, 6}

for \_, row := range rows {

for \_, column := range colunms {

if column == 4 {

continue

}

fmt.Printf("%d-%d\n", row, column)

}

}

}

上述代码的输出为:

1-2

1-6

3-2

3-6

5-2

5-6

continue和 *labeled statement* 配合使用时，不仅会结束当前层级的逻辑，还会跳转到 *label* 标签指定的位置。我们看下下面的代码逻辑.

// golang

package main

import "fmt"

func main() {

rows := []int{1, 3, 5}

colunms := []int{2, 4, 6}

RowLoop:

for \_, row := range rows {

for \_, column := range colunms {

if column == 4 {

continue RowLoop

}

fmt.Printf("%d-%d\n", row, column)

}

}

}

当column等于4时，结束逻辑，此时不是跳转到当前层级的 *post statement* , 而是跳转到RawLoop, 所以输出结果应该为:

1-2

3-2

5-2

不仅4没有输出，6也被 ****continue**** 掉了.

### **Return statement**

return语句跳出当前函数，回到函数的调用方, 同时将一个或者多个返回值传给调用方。本应出现在最后一行的 ****return**** 语句，在没有返回值的情况下，可以省略。 *Groovy* 语言的 ****return**** 语句是可选的。

// sum method

def static sum(n, closure) {

for(int i = 2; i <= n; i += 2) {

closure(i)

}

// return 2, 可以简写成2

2

}

### **Throw statement**

在一些语言中比如 *Java* , *JavaScript* 等使用 ****throw**** 语句来抛出错误，以便上层的调用方能够通过 ****try-catch-throw**** 的方式捕捉并处理。未捕捉的 ****throw**** 语句会导致线程/进程终止。对于 *Java* ,****throw**** 的的对象必须是 *Exception* 或者其子类，对于 *JavaScript* , throw的对象可以是任意类型.

// JavasScript

function getRectArea(width, height) {

if (isNaN(width) || isNaN(height)) {

throw "Parameter is not a number!";

}

}

try {

getRectArea(3, 'A');

}

catch(e) {

console.log(e);

// expected output: "Parameter is not a number!"

}

### **Goto statement**

goto语句和 *labeled statement* 配合使用，用于逻辑跳转，程序执行流程会直接跳转到标签处. *goto statement* 只有部分语言提供，而且写法也有不同，比如对于 *golang* , *labeled statement* 必须在 *goto statement* 之前, 而 *C* 语言则无此限制。

// c

#include <stdio.h>

int main ()

{

/\* 局部变量定义 \*/

int a = 10;

/\* do 循环执行 \*/

LOOP:do

{

if( a == 15)

{

/\* 跳过迭代 \*/

a = a + 1;

goto LOOP;

}

printf("a 的值： %d\n", a);

a++;

}while( a < 20 );

return 0;

}

以上都是常见的语句，除此之外，语言也会有实现一些非常规的语句，比如 *golang* 的 ****defer**** 语句。

### **Operators**

在介绍表达式之前，先介绍它的组成元素之一，操作符。其中，操作符分为一元(unary)操作符，二元(binary)操作符和三元操作符. 优先级决定了在多个操作符同时出现时，先使用哪个来求值。数字越大，优先级越高。

### **Unary operators**

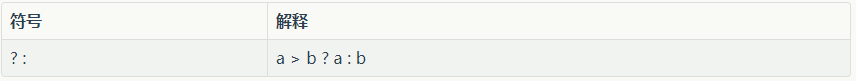


### **Binary operators**



### **Ternary operators**

在计算机中也叫条件运算符(conditional operator)



### **Expressions**

在编程语言里一个表达式(expression [7] )是由一个或多个常量，变量，操作符或函数组成。通过对表达式求值(evaluate)来得到一个新的值，这个新的值可以是基础类型，也可以是复合类型。表达式在运算时，会进行类型推断和类型检查。从之前讲的内容可以看出， *statements* 的作用主要是控制代码逻辑的执行顺序， *expressions* 则是具体的代码逻辑.

### **Variable expression**

// golang

c := a + b

### **Arithmetic expression**

2 + 3

### **Relational expression**

3 != 4

### **Function expression**

// golang

v := f(1)

### **Index expression**

a[x]

表达式并没有严格的分类，各个语言也不尽相同，这里仅列举了部分例子来说明。

### **Scoping**

即作用域, 作用域是名称(比如变量的声明)和其实体(entity, 比如变量的定义)的绑定规则。作用域约束了实体的作用范围，保证程序是无歧义的。

### **Expression scope**

实体仅在表达式内可用。

// c

({ int x = f(); x \* x; })

临时变量 *x* 接受函数的返回值并平方，这样避免两次调用函数 *f* .

### **Block scope**

通常编程语言都会使用花括号 {} 来将代码包裹成块(block), 在block内声明的实体，仅在block内有效。

// golang

{

var a int

b := a

}

b := a // compile error

// golang

{

var a int

b := a

}

b := 1 // ok

### **Function scope**

在函数内声明的实体，仅在函数内有效。

def square(n):

return n \* n

def sum\_of\_squares(n):

total = 0

i = 0

while i <= n:

total += square(i)

i += 1

return total

为了不和 *block scope* 混淆，这里用 *python* 的例子。

### **File scope**

在代码文件内声明的全局变量，仅在当前文件内有效。

### **Module scope**

在某些现代语言中，一个实体可以在一个模块内的各个文件内有效，比如 *golang* . 部分语言，一个文件就是一个独立的module，此时，也属于 *file scope* .

### **Global scope**

在所有模块，所有文件内都有效的实体称为全局实体，此类作用域属于 *global scope* . 在编程实践当中，应尽量避免使用。

### **Packages**

一个复杂程序一般会由多个包(或者叫模块)组成, 这种机制能让程序的结构和逻辑更加清晰可读，提高代码的复用能力，也可以借助 *module scope* 来避免同名之间的冲突。这里仅列举几种常见的包引入方式。

// Java 包名是一个层级结构

import org.apache.commons.net.util.SubnetUtils;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.stereotype.Service;

import org.springframework.util.StringUtils;

Subnetutils sbt = new Subnetutils("xx")

// python

from sys import argv

顺便说下python里 *from* 和常规 *import* 之间，在包的的使用上会有差异。

>>> from os import path

>>> path.

File "<stdin>", line 1

path.

^

SyntaxError: invalid syntax

>>> path

<module 'posixpath' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpath.pyc'>

>>> os.path

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'os' is not defined

>>> import os.path

>>> os.path

<module 'posixpath' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpath.pyc'>

>>>

// node

var circle = require('./circle.js');

console.log( 'The area of a circle of radius 4 is ' + circle.area(4));

// golang

import (

"fmt"

"github.com/xx/xx"

)

// TypeScript

import { cube, foo } from './mylib';

console.log(cube(3));

console.log(foo);

// c/c++

#include "my\_header.h"

int foo(char\* name) {

//do stuff

return 0;

}

## **编程范式**

编程范式(programming paradigms)，是在编程的理论与实践当中提炼出的概念模型。

Programming Paradigm: A conceptual model underlying the theories and practice of programming

### **IP(Imperative Programming)**

即指令式编程。程序由一系列指令和流程控制语句组成，运行过程中指令不断改变程序的状态，由此达到最终的编码意图。 ****IP**** 范式会显式地指定代码的执行流程，以及运算逻辑。汇编是典型的使用 ****IP**** 范式的编程语言。

result = []

i = 0

start:

numPeople = length(people)

if i >= numPeople goto finished

p = people[i]

nameLength = length(p.name)

if nameLength <= 5 goto nextOne

upperName = toUpper(p.name)

addToList(result, upperName)

nextOne:

i = i + 1

goto start

finished:

return sort(result)

### **SP(Structured Programming)**

即结构化编程，在 ****IP**** 的基础上，我们可以将用 ****goto**** 来控制流程的代码，以 ****for**** 语句， ****while****语句等此类结构化的代码块(block structure)组织起来，使得代码的可读性更高，那么此种编码方式即为结构化范式。 ****SP**** 是现代语言都支持的一种基础范式。

result = [];

for i = 0; i < length(people); i++ {

p = people[i];

if length(p.name)) > 5 {

addToList(result, toUpper(p.name));

}

}

return sort(result);

### **PP(Procedure Programming)**

即过程式编程，单看中文可能难以理解， *procedure* 来源于 *procedure call* , 即函数调用，主要是因为 ****PP**** 在 ****IP**** 的基础上引入了函数及函数调用, 将可提炼的逻辑用函数封装起来，以复用代码和提高可读性。

OOP(Object-oriented Programming)

即我们常说的面向对象编程。在 ****SP**** 和 ****PP**** 的范畴里，数据类型是松散的，数据结构和算法之间也是松散的，而 ****OOP**** 则提供了一种类似于人类对现实世界建模的方法，对二进制世界的类型和逻辑进行建模和封装，并在此基础上提供了更多的类型和语法特性。 ****OOP**** 的优点简列如下(封装，继承，多态):

* 当我们对一组数据类型进行抽象，封装成类(class, 类是 ****OOP**** 的基本概念)时，我们可以定义该类的子类，来共享它的数据类型和逻辑，此方式称为继承(inheritance), 能够有效减少我们的开发时间。
* 当类被定义后，通常它只需要关注它自身的数据和逻辑，通过语法特性，一般是 *public* / *private* 关键字，将这类数据和逻辑隐藏起来，避免被非法(或者说不合理的, 不应当的)访问，提升程序和系统的安全性。
* 一个封装好的类，不仅能被它的创建者使用，也可以分发(在网络上)给其他人使用, 比如 *Java*的jar包。
* 一门语言不可能把开发者所需要的所有的类型都定义好，class的概念则很好地解决了这个问题，开发者可以定义任意自己想要的数据类型。
* ****OOP**** 的多态性质可以让代码更加灵活。

### **DP(Declarative Programming)**

即，描述性范式。和 ****IP**** 相反，此类语言只描述最终的编码意图，不描述达到意图的过程。举个例子，如何用程序来回答你是怎么回家的？

*IP* :

Go out of the north exit of the parking lot and take a left. Get on I-15 south until you get to the Bangerter Highway exit. Take a right off the exit like you’re going to Ikea. Go straight and take a right at the first light. Continue through the next light then take your next left. My house is #298.

*DP* :

My address is 298 West Immutable Alley, Draper Utah 84020

典型的 ****DP**** 范式语言如 *SQL* , 仅描述目的，达到目的的逻辑被隐藏。

SELECT \* FROM Users WHERE Country=’Mexico’;

### **LP(Logic Programming)**

即逻辑编程，它属于 ****DP**** 的范畴。逻辑编程的要点是将数学中的逻辑风格带入计算机程序设计之中。它设置匹配规则来解决问题(rule-based)，而非设置步骤来解决问题, 即事实+规则=结果。*Prolog* 是典型的 ****LP**** 范式语言，此类语言主要应用在人工智能，专家系统等领域。

### **FP(Functional Programming)**

即函数式编程，也是 ****DP**** 的子集, 在函数式编程里，所有的计算都是通过函数调用完成的，函数里的 ****SP**** 逻辑尤其是控制流逻辑，被隐藏了起来. 假设我们要编写一个函数，将一个数组的每个元素都乘以2， ****PP**** 风格的代码如下:

// TypeScript

function double (arr) {

let results = []

for (let i = 0; i < arr.length; i++){

results.push(arr[i] \* 2)

}

return results

}

上述代码，详细地写明了整个计算过程，包括迭代过程和计算方法。所以 ****IP**** 范畴的范式会详细描述计算机是如何完成这件事的，有篇文章是这么描述 ****IP**** 的

First do this, then do that.

FP则不会描述数组是如何迭代的，也不会显式地修改变量, 仅仅描述了我们想要什么，我们想要将元素乘以2.

function double (arr) {

return arr.map((item) => item \* 2)

}

FP将开发者从机器的执行模型切换到了人的思维模型上，可读性会更高。需要注意的是，某些支持 ****FP**** 的语言本身是属于 ****IP**** 的，同时也可以认为其属于 ****DP**** , 不必过于纠结。

### **FRP(Functional Reactive Programming)**

即，函数式响应型编程。

### **MP(Meta Programming)**

即元编程, 也写做 *Metaprogramming* 。元编程是一种可以将程序当作数据来操作的技术，元编程能够读取，生成，分析或转换其他的程序代码，甚至可以在运行时修改自身. *C++* 的template即属于 *meta programming* 的范畴，编译器在编译时生成具体的源代码。在web框架 *Ruby on Rails*里，元编程被普遍使用。比如，在SQL数据库的表里，有一个表users，在ruby中用类 *User* 表示，你需要根据user的email字段来获取相应的结果，通常的做法是写个sql查询语句去完成，但是*Ruby on Rails* 在元编程的加持下，会让这件事变得异常简单。

User.find\_by\_email('songtianyi630@163.com')

*find\_by\_email* 并不是我们自己的定义的函数，它是如何完成这件事的呢？框架会根据函数名 *find\_by\_email* 动态生成一个查询函数去查询数据库。除了这种黑魔法，元编程还能够动态修改一个类的成员函数，在运行时创建函数等等，这里不展开讲，在 *Ruby* 或者 *Groovy* 的相关书籍里会有详细介绍。语言的反射特性和对模板的支持是实现元编程的主要基础，虽然 *c++* 并不支持反射，但 *c++* 的模板提供了元编程的能力。

编程范式还有很多细分项，比如 *Event-driven programming* , *Distributed programming* [10] 等, 这里不再一一列举。

## **依赖管理**

有了合适的类型系统，且该语言支持了我们所需要的编程范式，那么它就是不二之选了嘛？不一定。

长期从事开发，或者从事过大型项目开发的程序员会发现代码的依赖管理也是开发和维护过程当中的重点。好的依赖管理会降低我们的心智负担，也会降低业务风险。

TODO

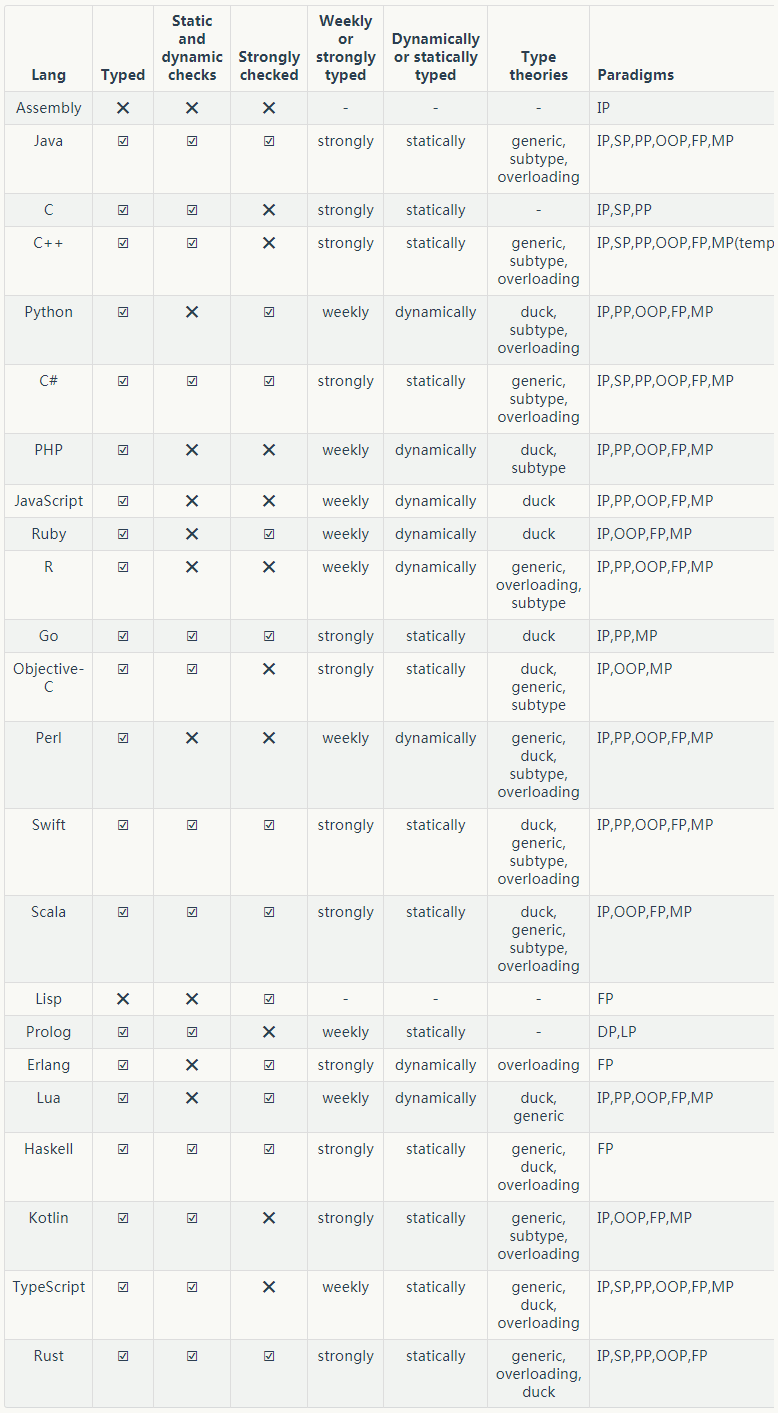
## **标准库**

TODO

## **选型参考**

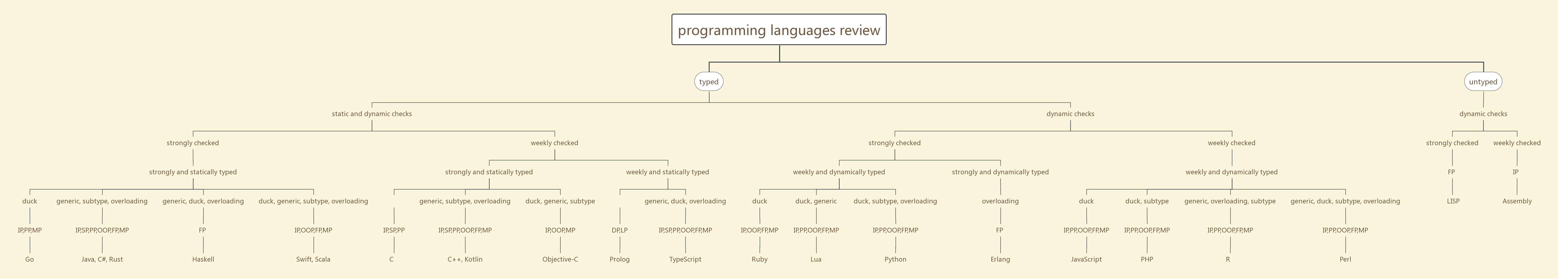
### **术语说明**

* *static and dynamic checks* : 指该语言的类型系统会进行静态检查和动态检查，注意，所有语言都有动态检查。
* *strongly checked* : 以检查范围的大小作为标准，意味着类型系统的类型检查(type checking)已经尽可能消除了类型错误, 有些地方会称为 *strongly typing* , 有的地方甚至会称为 *strongly typed* , 要注意区分.
* *weekly typed* : 类型可变
* *strongly typed* : 类型不可变
* *dynamically typed* : 值的类型仅在运行时确定
* *statically typed* : 值的类型仅在编译时确定
* *duck* : duck typing, 也叫row polymorphism
* *generic* : parametric polymorphism
* *subtype* : subtype polymorphism
* *overloading* : Ad hoc polymorphism, 函数重载或操作符重载或两者都有



整理完这个对比表才发现一些有意思的事情:

* 原来大部分语言都有 ****OOP**** , *js* 也是.
* *Go* 的优势并不在自己的类型系统和语法特性上, 它的设计充分体现了 *less is more* 的设计哲学, 上手十分简单, 工程实践效果很好。



## **参考文献**

1. [《Type Systems》, Luca Cardelli](http://www.cs.colorado.edu/~bec/courses/csci5535/reading/cardelli-typesystems.pdf)
2. [《函数式语言泛型特性的研究与实现》, LI Yang, YU Shangchao, WANG Peng](http://cea.ceaj.org/CN/article/downloadArticleFile.do?attachType=PDF&id=29391)
3. [《Types and Programming Languages》, Benjamin C. Pierce](https://www.asc.ohio-state.edu/pollard.4/type/books/pierce-tpl.pdf)
4. [《依赖类型》, wikipedia](https://www.tuicool.com/choose-a-programming-language/zh.wikipedia.org/wiki/%E4%BE%9D%E8%B5%96%E7%B1%BB%E5%9E%8B)
5. [《what is dependent type?》, StackOverflow](https://stackoverflow.com/questions/9338709/what-is-dependent-typing)
6. [《Antlr Docs》, github](https://github.com/antlr/antlr4/blob/master/doc/lexicon.md)
7. [《Expression (computer science)》, wikipedia](https://en.wikipedia.org/wiki/Expression_(computer_science))
8. [《List of programming languages by type》, wikipedia](https://en.wikipedia.org/wiki/List_of_programming_languages_by_type" \l "Metaprogramming_languages)
9. [《Comparison of programming languages》, wikipedia](https://en.wikipedia.org/wiki/Comparison_of_programming_languages)
10. [《Comparison of multi-paradigm programming language》, wikipedia

文章最后发布于: 2018-08-17 10:11:04

有 0 个人打赏

IMG_271

[IMG_272](https://blog.csdn.net/u011001084/article/details/javascript:void(0);)

窗体顶端

