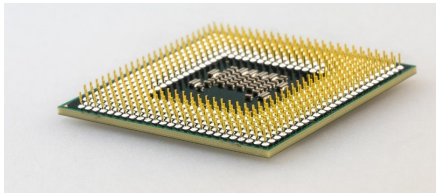
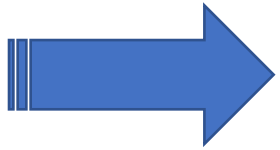
编程语言简介

冯新宇

南京大学

编程语言发展历史





Machine Language

```
00100101 11010011
00100100 11010100
10001010 01001001 11110000
01000100 01010100
01001000 10100111 10100011
11100101 10101011 00000010
00101001
11010101
11010100 10101000
10010001 01000100
```

Assembly Language

```
        ST 1,[801]
        ST 0,[802]
TOP:    BEQ [802],10,BOT
        INCR [802]
        MUL [801],2,[803]
        ST [803],[801]
        JMP   TOP
BOT:    LD A,[801]
        CALL PRINT
```

A Simple C Program

```
/* This is the sample program to print a
message hello world. This is done by
course teacher */

#include <stdio.h>

#include <conio.h>


main ( )

{
    clrscr();
    printf("Hello World\n");
    getch();

}
```

13

机器语言

汇编语言

高级语言

编程语言发展历史

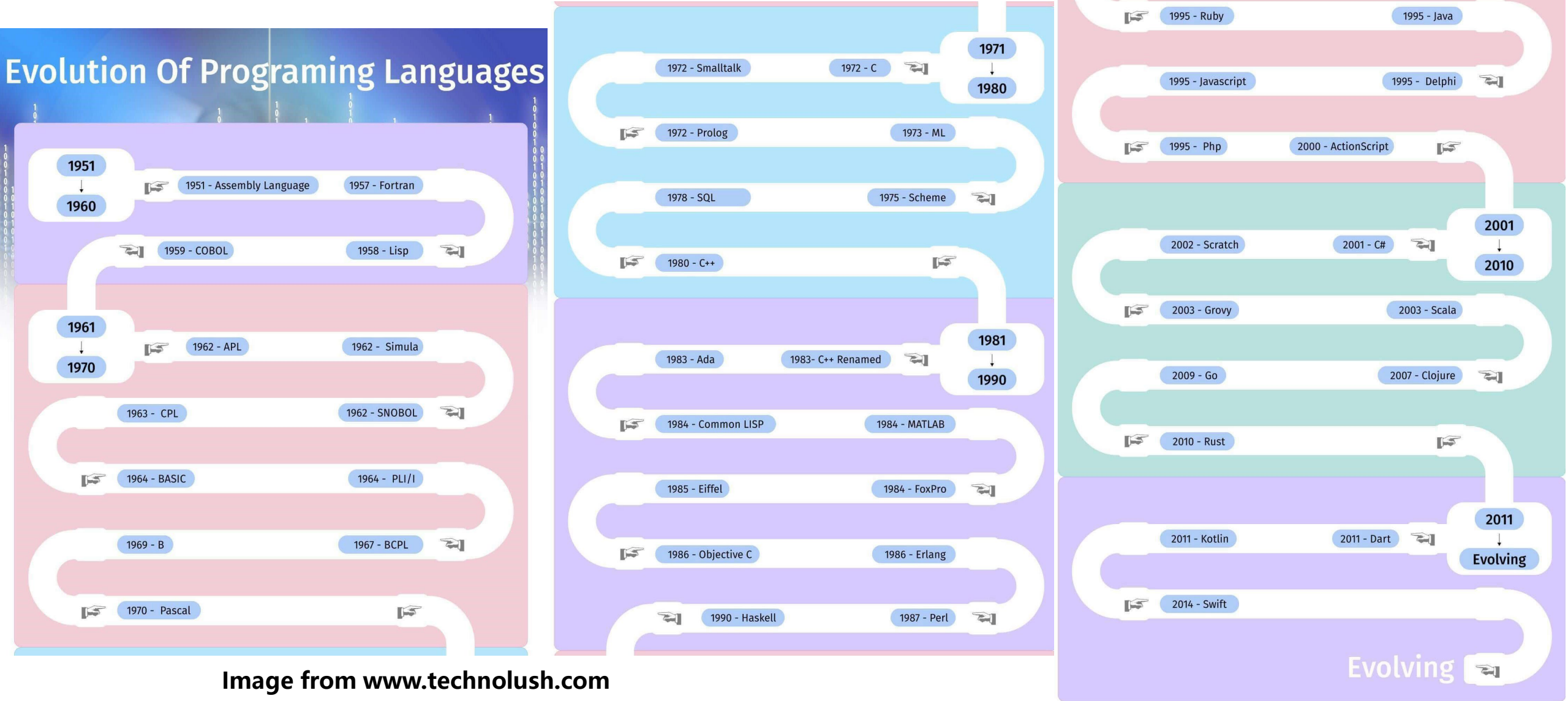


Image from www.technolush.com

编程语言排名

- TIOBE 编程社区指数

  定位：衡量语言全球讨论热度与搜索频率（基于Google、Bing、维基百科等引擎数据）

- GitHub 开发者生态排名

  定位：基于全球最大代码托管平台的活跃仓库、开发者数量与企业采用率

- IEEE Spectrum

- PYPL（PopularitY of Programing Language）

  定位：基于Google教程搜索量，Python连续5年居首（初学者首选）

- RedMonk

  定位：结合GitHub与Stack Overflow数据，JavaScript/Python/Java稳居前三

内容由腾讯元宝（DeepSeek）生成

编程语言排名

- TIOBE 编程社区指数

定位：衡量语言全球讨论热度与搜索频率（基于Google、Bing、维基百科等引擎数据）

2025年8月前十排名：

排名	语言	占比	趋势变化	关键驱动因素
1	Python	26.14%	↑8.10% (历史新高)	AI编程助手普及（效率提升20%）
2	C++	9.18%	↓0.86%	游戏/嵌入式高性能需求
3	C	9.03%	↓0.15%	操作系统/驱动开发基石
4	Java	8.59%	↓0.58%	企业级后端与Android生态
5	C#	5.52%	↓0.87%	Unity游戏与.NET跨平台
6	JavaScript	3.15%	↓0.76%	全栈Web开发（TypeScript辅助）
7	Visual Basic	2.33%	↑0.15%	旧系统维护需求
8	Go	2.11%	↑0.08%	云原生（K8s/Docker）
9	**Perl**	2.08%	↑1.17% (最大黑马)	文本处理与历史脚本复兴
10	Delphi/Object Pascal	1.82%	↑0.19%	工业控制软件升级

趋势洞察：

- **AI赋能循环**：
  Python因AI工具支持（如Copilot）进一步扩大优势
- **保守技术回流**：
  VB/Delphi因旧系统维护需求逆势上升
- **安全语言崛起**：
  Rust（第18位）因内存安全受Linux/Windows驱动项目青睐

内容由腾讯元宝（DeepSeek）生成

编程语言排名

- GitHub 开发者生态排名

定位：基于全球最大代码托管平台的活跃仓库、开发者数量与企业采用率

2024年综合影响力前十：

排名	语言	活跃仓库占比	年度增长	核心应用场景
1	Python	21.8%	+5.7%	AI/数据科学（PyTorch/TensorFlow）
2	JavaScript	19.3%	-1.2%	Web全栈（React/Node.js）
3	Java	12.1%	+0.8%	企业级后端（Spring Boot 3.x）
4	**TypeScript**	11.7%	**+18%**	工程化前端与后端渗透
5	C#	8.5%	+3.1%	游戏（Unity）/.NET云服务
6	Go	6.9%	+12.4%	云原生（K8s/Docker）
7	**Rust**	4.3%	**+31%**	系统安全（Linux内核集成）
8	PHP	3.8%	-4.5%	传统Web（Laravel生态收缩）
9	Kotlin	3.5%	+9.2%	Android开发与协程架构
10	Swift	2.7%	+6.8%	苹果生态（Vapor服务端框架）

趋势洞察：

- **类型系统普及**：
  TypeScript增长迅猛，推动JS大型项目健壮性

- **云原生主导**：
  Go在CNCF项目中占比61%（如Kubernetes）

- **安全刚性需求**：
  Rust增速最快，关键基础设施采用率飙升至22%

内容由腾讯元宝（DeepSeek）生成

为什么会存在这么多编程语言？

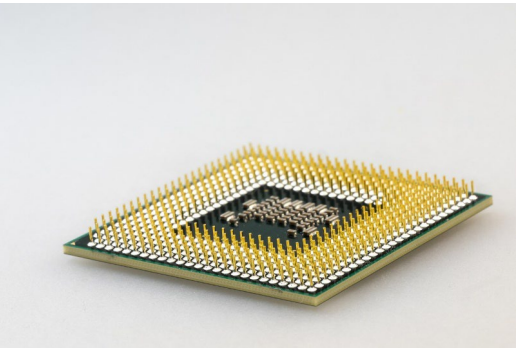
什么样的语言是好的语言？

或者：评价一个语言关心哪些方面？

编程语言领域有哪些新的发展？

为什么会存在这么多编程语言?

程序设计语言：人与计算机交流的语言



交流的内容：计算、通信、世界的建模 ...

描述计算有不同的模型

计算任务的多样性：AI、图形动画、数据处理、科学计算（天气预报、仿真）

交流的对象

- 人：简单、高效、正确的表达自己的计算任务，有较强主观性
- 机器：用尽量少的时间和资源（内存、能耗等）完成计算任务

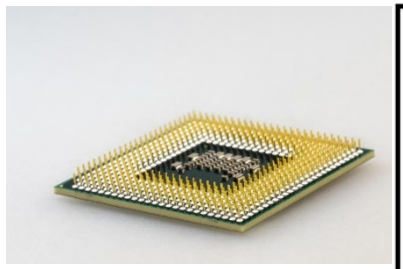
编程语言要解决的3个核心问题：效率、性能、安全

编程语言：表达人类（计算）思维，程序易写易读

```
main() {
    print("Hello World")
}
```

编译器 自动翻译

机器语言：硬件只能理解二进制表达



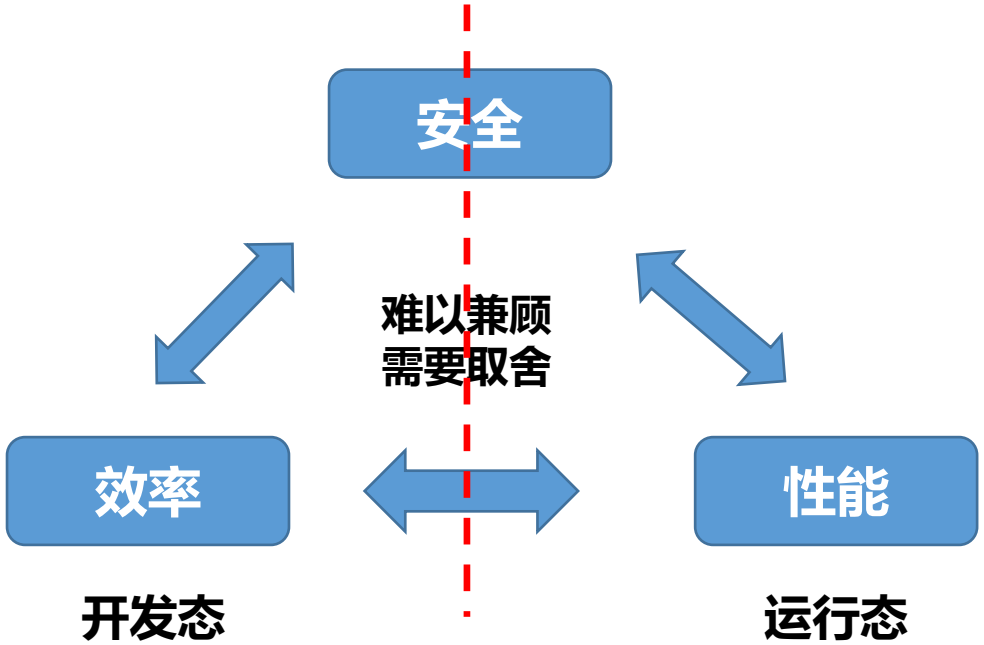
Machine Language

```
00100101 11010011
00100100 11010100
10001010 01001001 11110000
01000100 01010100
01001000 10100111 10100011
11100101 10101011 00000010
00101001
11010101
11010100 10101000
10010001 01000100
```

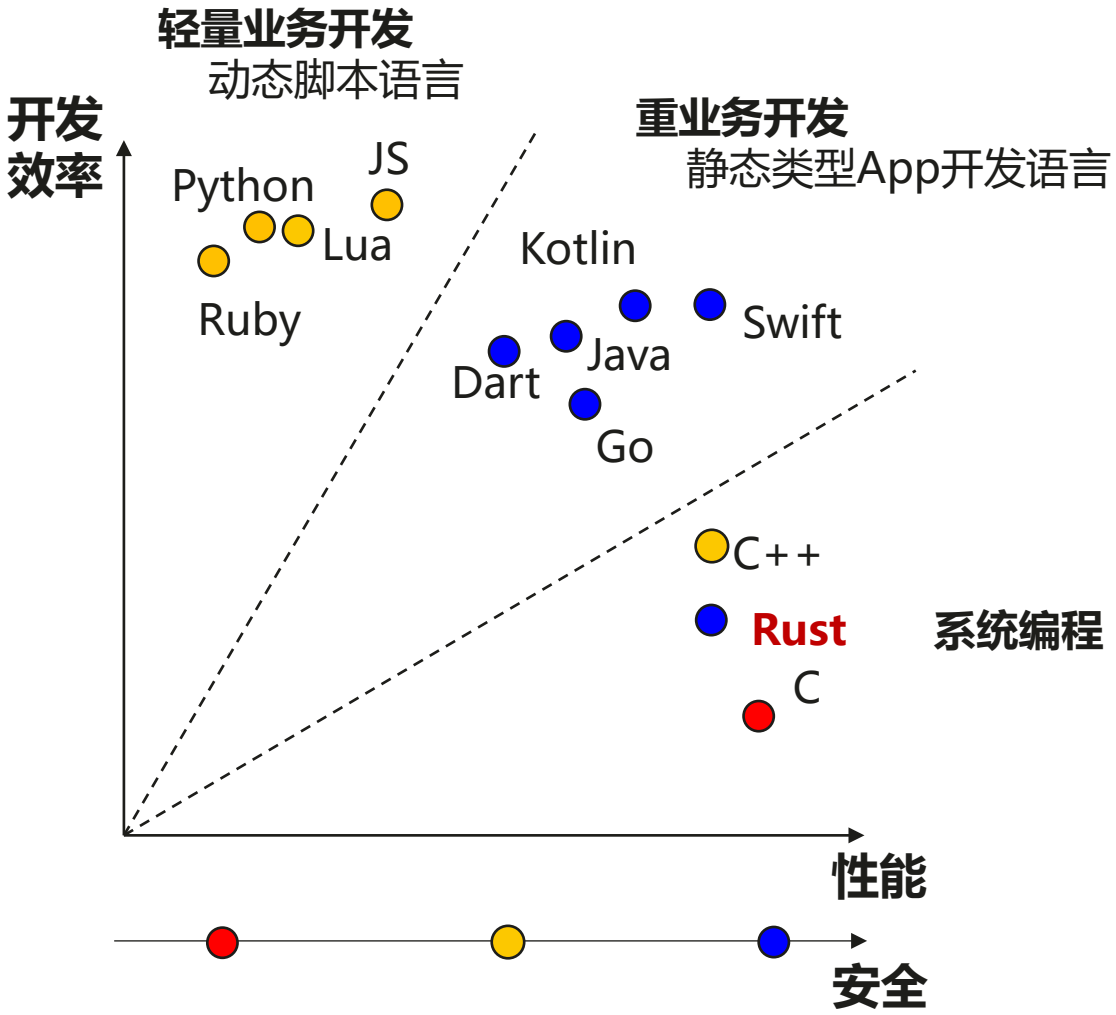
效率：如何让程序**写得快**、易理解

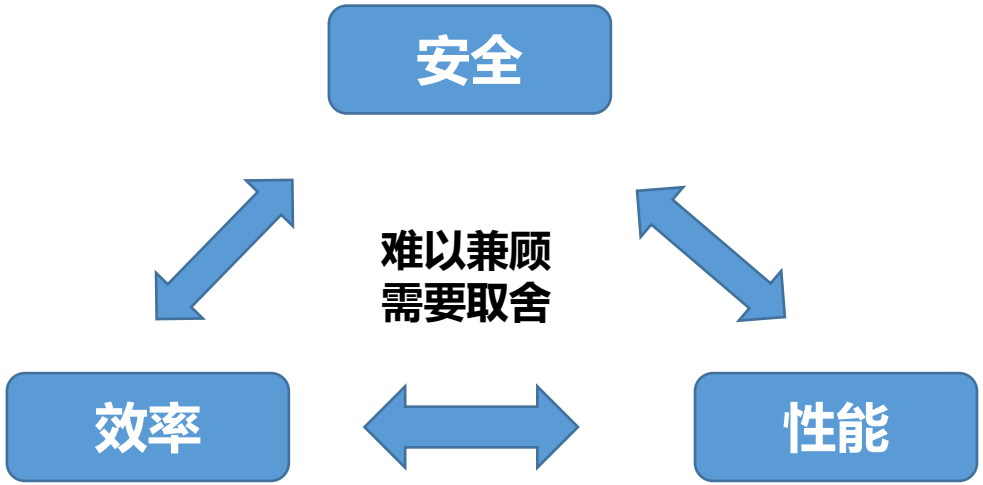
性能：如何让程序**跑得快**、消耗资源少

安全：如何让程序**写得对**、运行中不出错



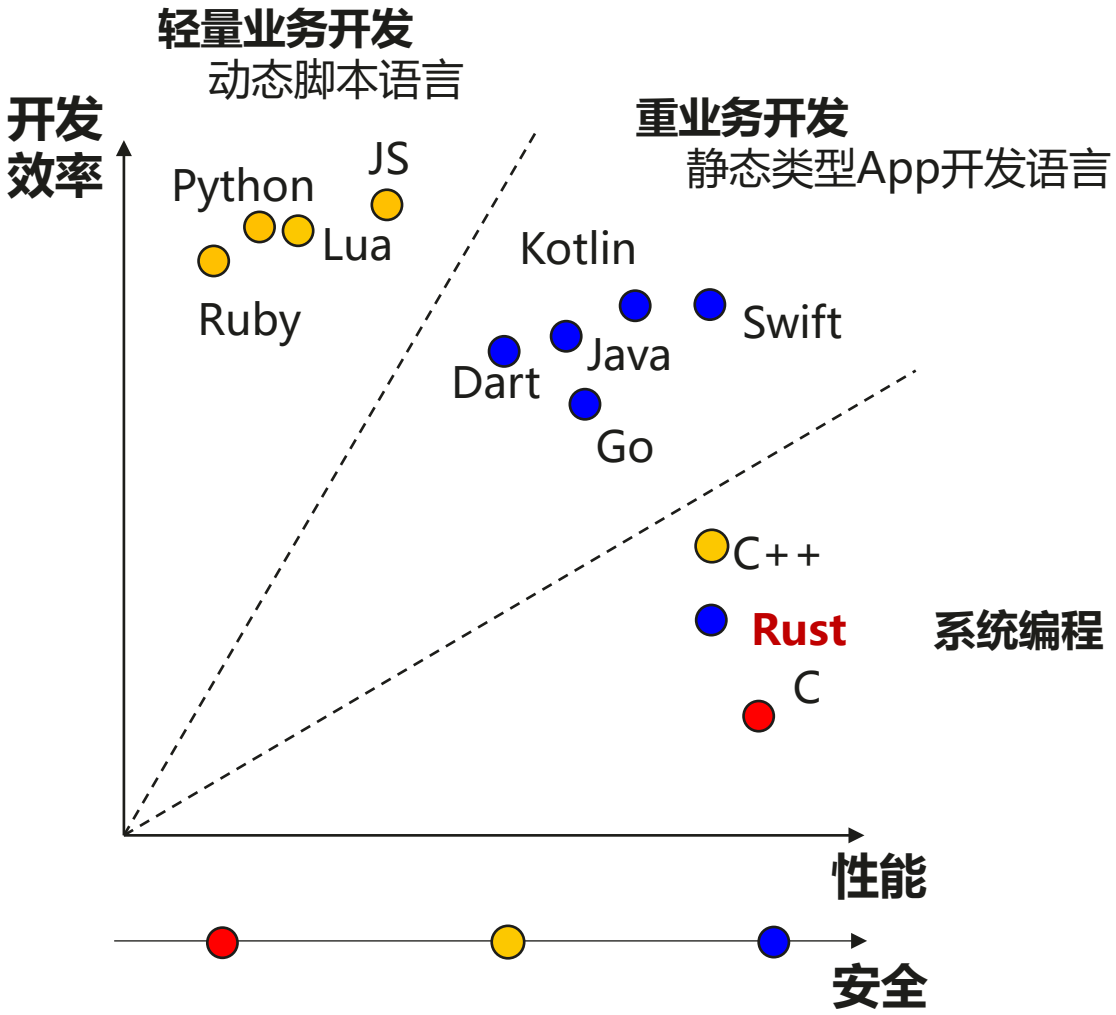
语言的分类：应用场景视角





注：此图仅为示意三类语言的划分，并不追求各种语言在坐标中位置的准确性

语言的分类：应用场景视角



注：此图仅为示意三类语言的划分，并不追求各种语言在坐标中位置的准确性

动态脚本语言：

串珠子的绳子

代码量小、快速编写、解释执行

易学易用、不追求安全、不追求性能

语言开发容易，种类繁多

```
> typeof NaN
"number"
> 9999999999999999
10000000000000000
> 0.5+0.1==0.6
true
> 0.1+0.2==0.3
false
> Math.max()
-Infinity
> Math.min()
Infinity
> true+true+true===3
true
> true-true
0
> true==1
true
> true===1
false
> (!+[]+[]+![]).length
9
> 9+"1"
"91"
> 91-"1"
90
> []==0
true
> []+{}
"[object Object]"
> {}+[]
0
```

https://www.zhihu.com/question/29823322

知乎 首页 会员 发现 等你来答 《永劫无间》制作人亲自答

人工智能 编程语言 Python 编程 Python 入门

Python 有什么不为人知的坑?

首先得声明一下，本人可不是在黑Python，事实上目前正在用Python做项目，而且在接触Python这些时间之后，已经不大使用C++和Delphi了...显示全部

关注问题 写回答 邀请回答 好问题 131 4 条评论 分享 ···

73 个回答 默认排序

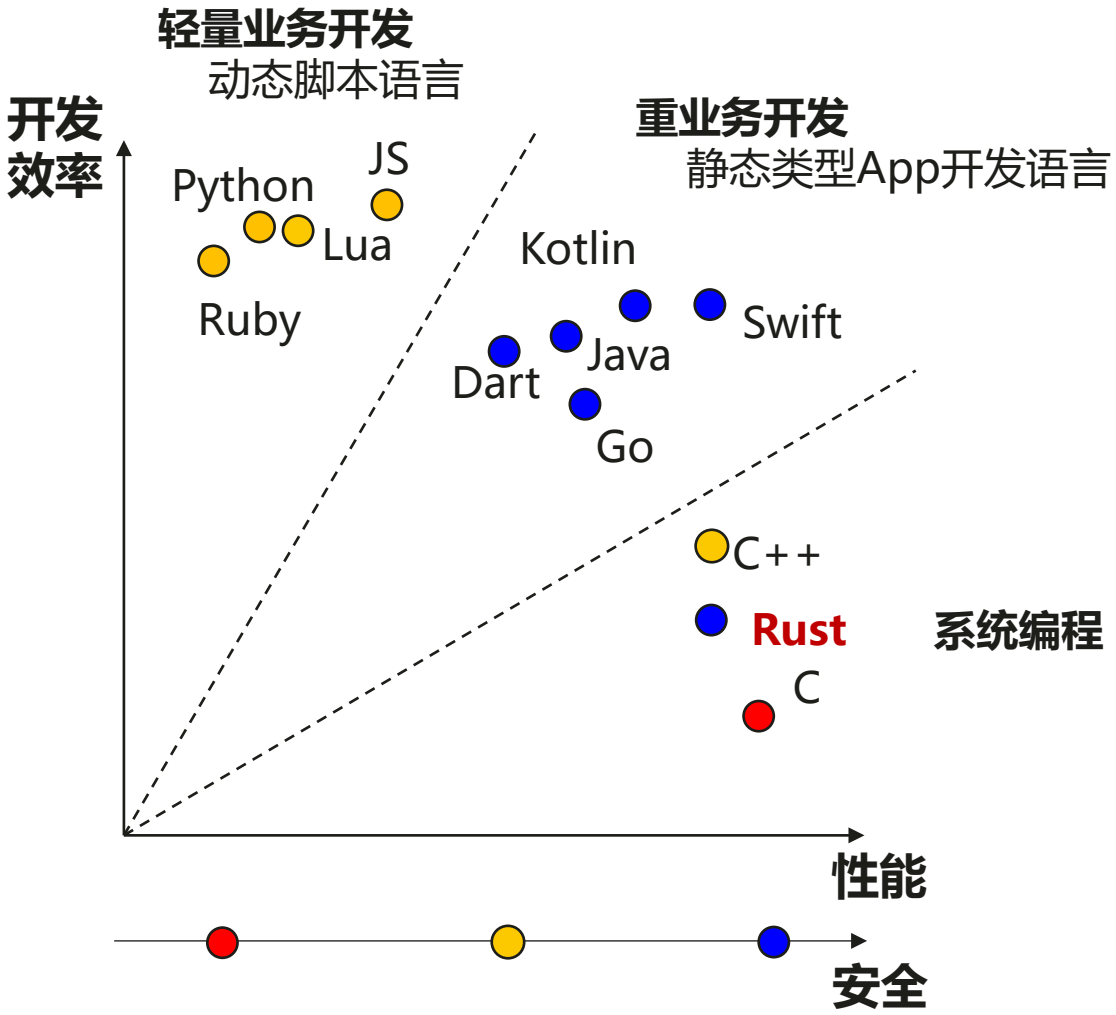
七月在线 七仔
关注公众号：七月在线实验室 领取AI干货大礼包

2,360 人赞同了该回答

这51项对于小白来说可能就是“天坑”了，掉进去还一直挠头纳闷的那种……

下面的每一个都会出现一些出乎意料的输出结果，如果你是个老司机也许会了解部分，但是我相信这里面还是会出现你不知道的。

语言的分类：应用场景视角



注：此图仅为示意三类语言的划分，并不追求各种语言在坐标中位置的准确性

动态脚本语言：

串珠子的绳子

代码量小、快速编写、解释执行

易学易用、不追求安全、不追求性能

语言开发容易，种类繁多

系统开发语言：

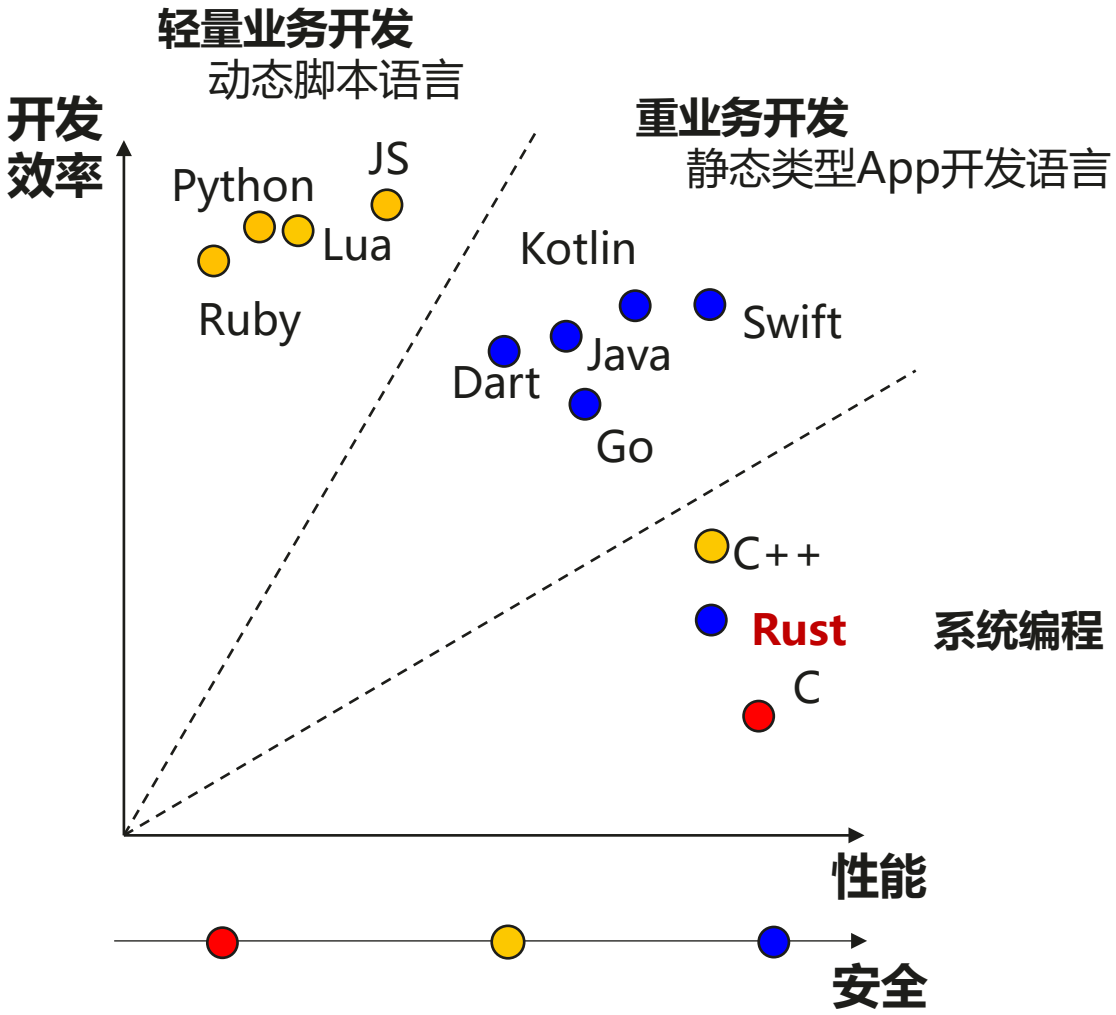
极致性能

与底层系统的对话能力

开发者能力强

HUAWEI

语言的分类：应用场景视角



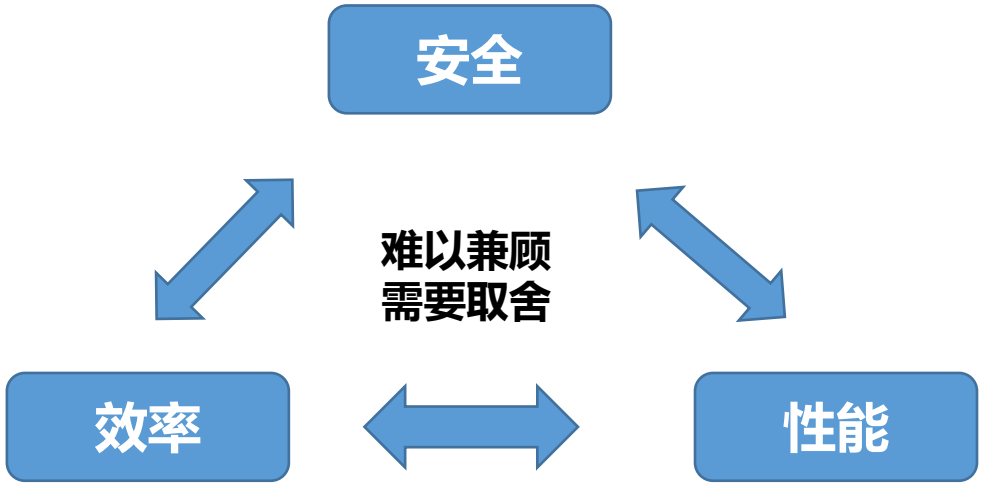
注：此图仅为示意三类语言的划分，并不追求各种语言在坐标中位置的准确性

静态类型App开发语言

面向开放场景，追求应用生态：吸引开发者

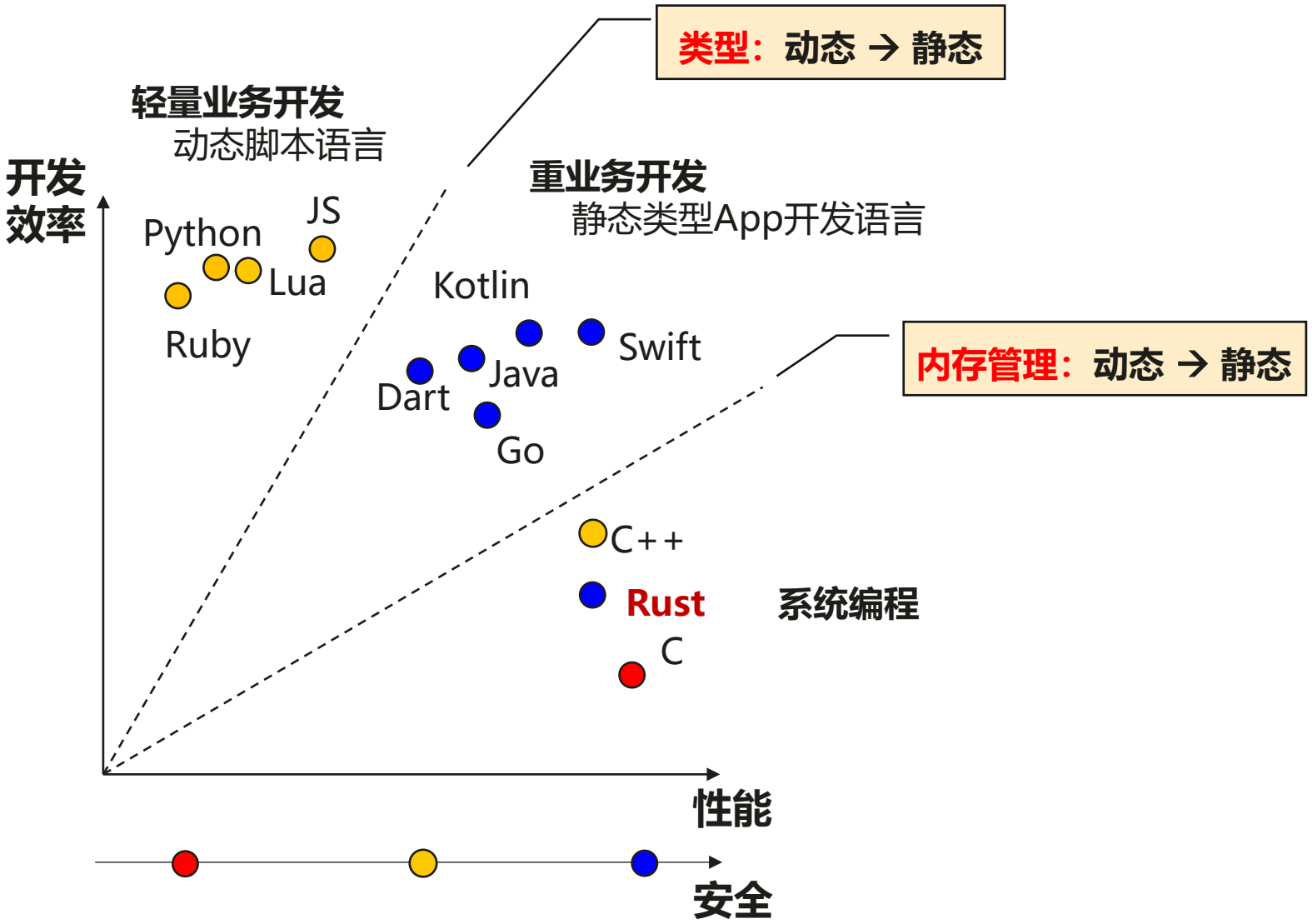
代码规模较大，追求性能、安全与易用性的平衡

静态类型、类型安全、自动内存管理



三者的取舍取决于动态/静态的选择

语言的分类：从动-静态技术选型的视角看



注：此图仅为示意三类语言的划分，并不追求各种语言在坐标中位置的准确性

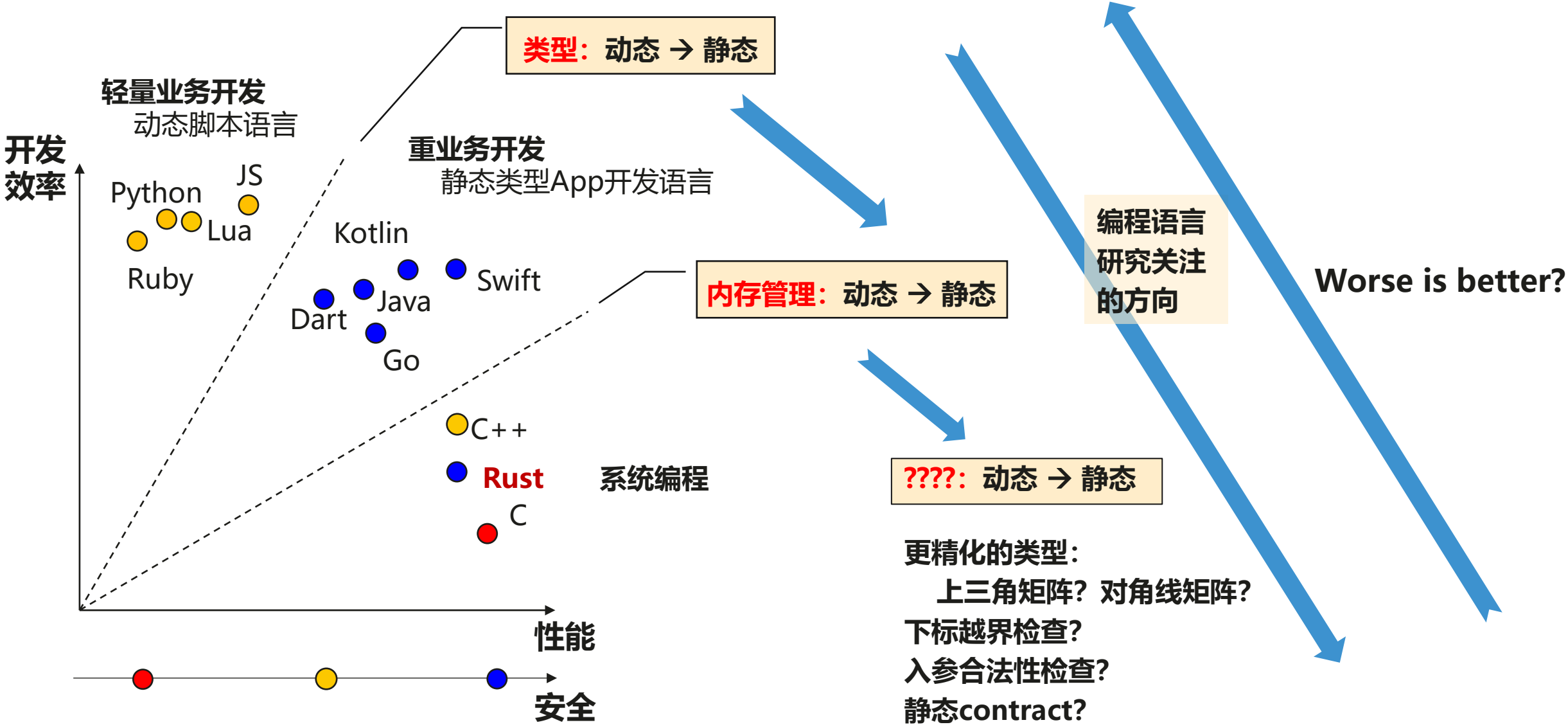
类型检查：动态 vs. 静态

	动态	静态
效率	约束少，表达力强	约束强、规则保守
性能	• 运行时类型检查 • 难以静态优化	编译器可获得更多信息指导优化
安全	易写难读，“编码一时爽，重构火葬场”	编译期发现错误

内存管理：动态（GC） vs. 静态

	动态	静态
效率	开发者无感知/少感知	对内存的使用更严格遵循某种模式，学习使用成本高
性能	• 运行时开销 • 非确定性时延	零成本抽象 更固定的内存使用模式带来更多编译优化
安全	内存安全	内存/并发安全

语言的分类：从动-静态技术选型的视角看



注：此图仅为示意三类语言的划分，并不追求各种语言在坐标中位置的准确性

语言的分类：从动-静态技术选型的视角看

Seat Belts or Handcuffs?

The biggest debate in language design is probably the one between Those who think that a language should prevent programmers from doing stupid things, and those who think programmers should be allowed to do whatever they want. Java is in the former camp, and Perl in the latter. (Not surprisingly, the DoD is big on Java.)

Advocates of static typing argue that it helps to prevent bugs and helps compilers to generate fast code (both true). Advocates of dynamic typing argue that static typing restricts what programs you can write (also true). I prefer dynamic typing. I hate a language that tells me what to do. But some smart people seem to like static typing, so the question must still be an open one.

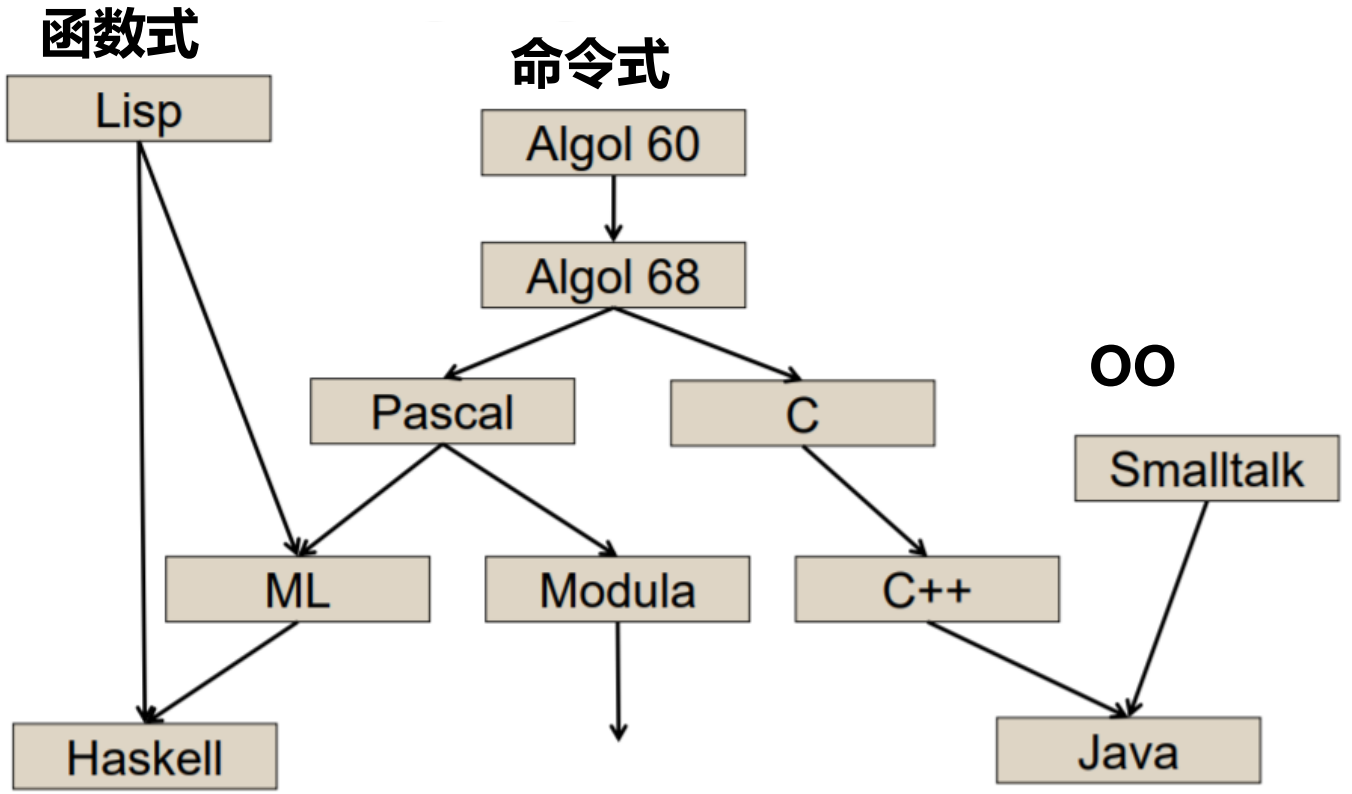
语言的分类：按计算模型和编程风格

函数式语言

命令式（过程式）语言

面向对象（OO）语言

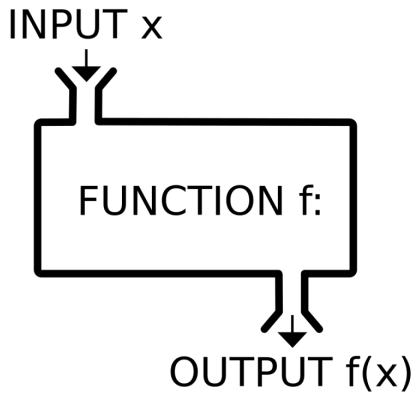
逻辑式语言

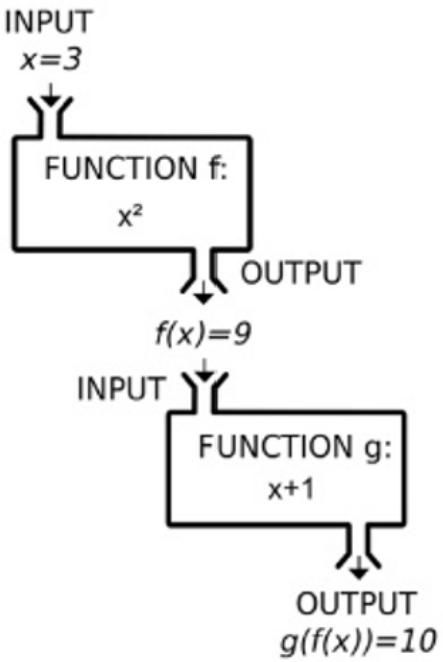


Many others: Algol 58, Algol W, Scheme, EL1, Mesa (PARC), Modula-2, Oberon, Modula-3, Fortran, Ada, Perl, Python, Ruby, C#, Javascript, F#…

语言的分类：按计算模型和编程风格

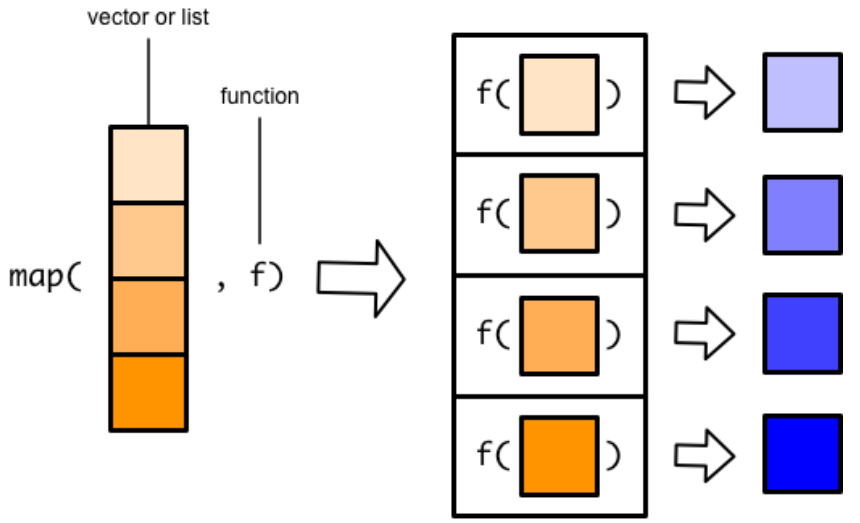
函数式语言：通过函数和函数的组合完成计算





Python

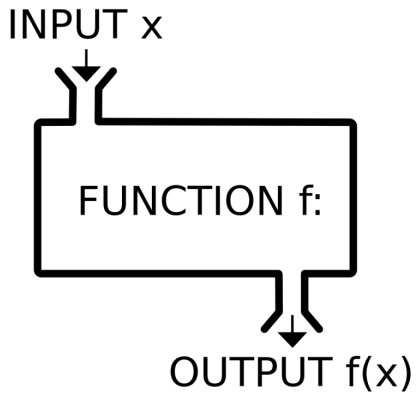
```
>>> add_one = lambda x: x + 1
>>> add_one(2)
3
```

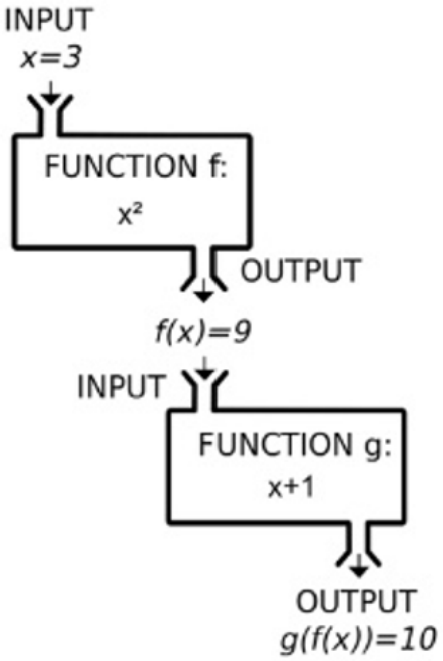



```
>>> map([1, 2, 3], add_one)
[2, 3, 4]
```

语言的分类：按计算模型和编程风格

函数式语言：通过函数和函数的组合完成计算



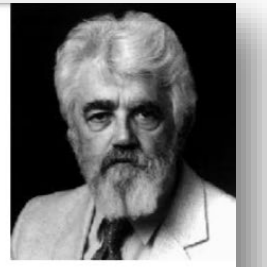


计算模型：λ-演算（Lambda-演算），由Church在20世纪30年代提出

Alonzo Church (1903–1995)

经典函数式语言

Lisp: John McCarthy



- Pioneer in AI
  - Formalize common-sense reasoning
- Also
  - Propose
  - Mathem
  - ....
- Lisp

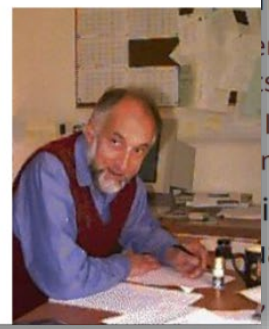
stems fr
symboli
(math, logi

LISP, 1958

ML, 1973

ML programming language

- Statically typed, general-purpose programming language
  - "Meta-Language" of the LCF theorem proving system
- Type safe, with formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- Used in printed textbook as example language



Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, …

Haskell, 1990

Haskell

ng language is

eral-purpose, strongly typed, higher-order,
s type inference, interactive and compiled use
lazy evaluation, purely functional core, rapidly
m

ittee in 80's and 90's to unify research
ages

Haskell '98, Haskell' ongoing

- "A History of Haskell: Being Lazy with Class" HOPL 3

Paul Hudak

John Hughes

Simon
Peyton Jones

Phil Wadler

语言的分类：按计算模型和编程风格

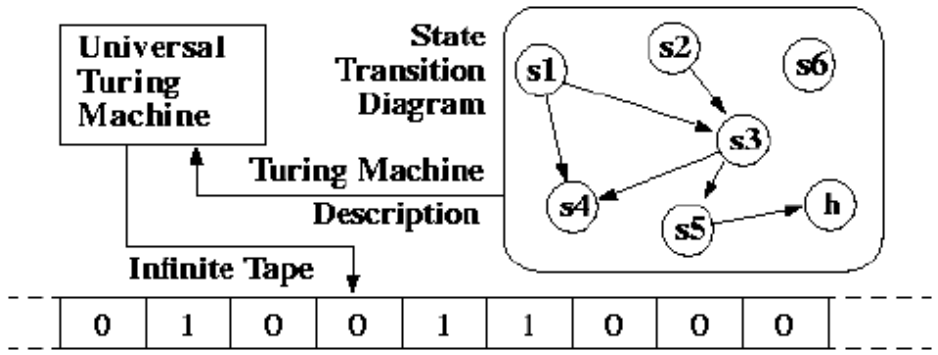
命令式（过程式）语言：编排命令序列，逐步修改机器状态，完成计算

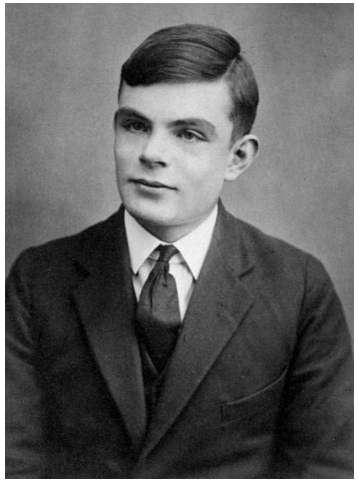
经典的命令式（过程式）语言：Algol (1958), BASIC (1964), Pascal (1970), C (1972) ...

```
int mylist[] = {1, 2, 3};
int length = 3;
for (int i = 0; i < length; i++){
    mylist[i] = mylist[i] + 1;
}
```

按照特定过程
修改机器状态

计算模型：图灵机（图灵1936年提出）





Alan Turing,
1912 - 1954

语言的分类：按计算模型和编程风格

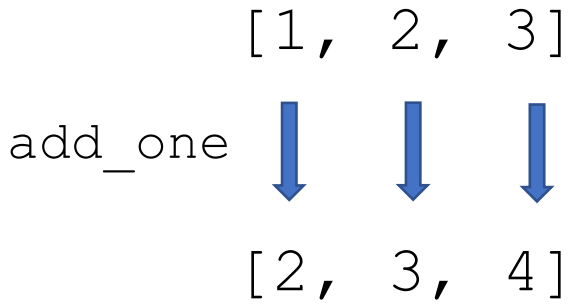
命令式（过程式）语言：编排命令序列，逐步修改机器状态，完成计算

```
int mylist[] = {1, 2, 3};
int length = 3;
for (int i = 0; i < length; i++){
    mylist[i] = mylist[i] + 1;
}
```

按照特定过程
修改机器状态

命令式 和 **函数式** 的对比：

```
add_one = lambda x: x+1
map([1, 2, 3], add_one)
```



语言的分类：按计算模型和编程风格

命令式（过程式）语言：编排命令序列，逐步修改机器状态，完成计算

命令式 和 **函数式** 的对比：

C:

```
long factorial(int n)
{
  int c;
  long result = 1;

  for (c = 1; c <= n; c++)
    result = result * c;

  return result;
}
```

Haskell

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

数学定义：

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fact(n-1) & \text{if } n > 0 \end{cases}$$

语言的分类：按计算模型和编程风格

面向对象语言：命令式的一种，数据（属性）和计算（行为）的封装

```
class Pokemon:
    """A Pokemon."""
```

All Pokemon have:

- a name
- a trainer
- a level
- an amount of HP (life)
- an attack: tackle

Pokemon can:

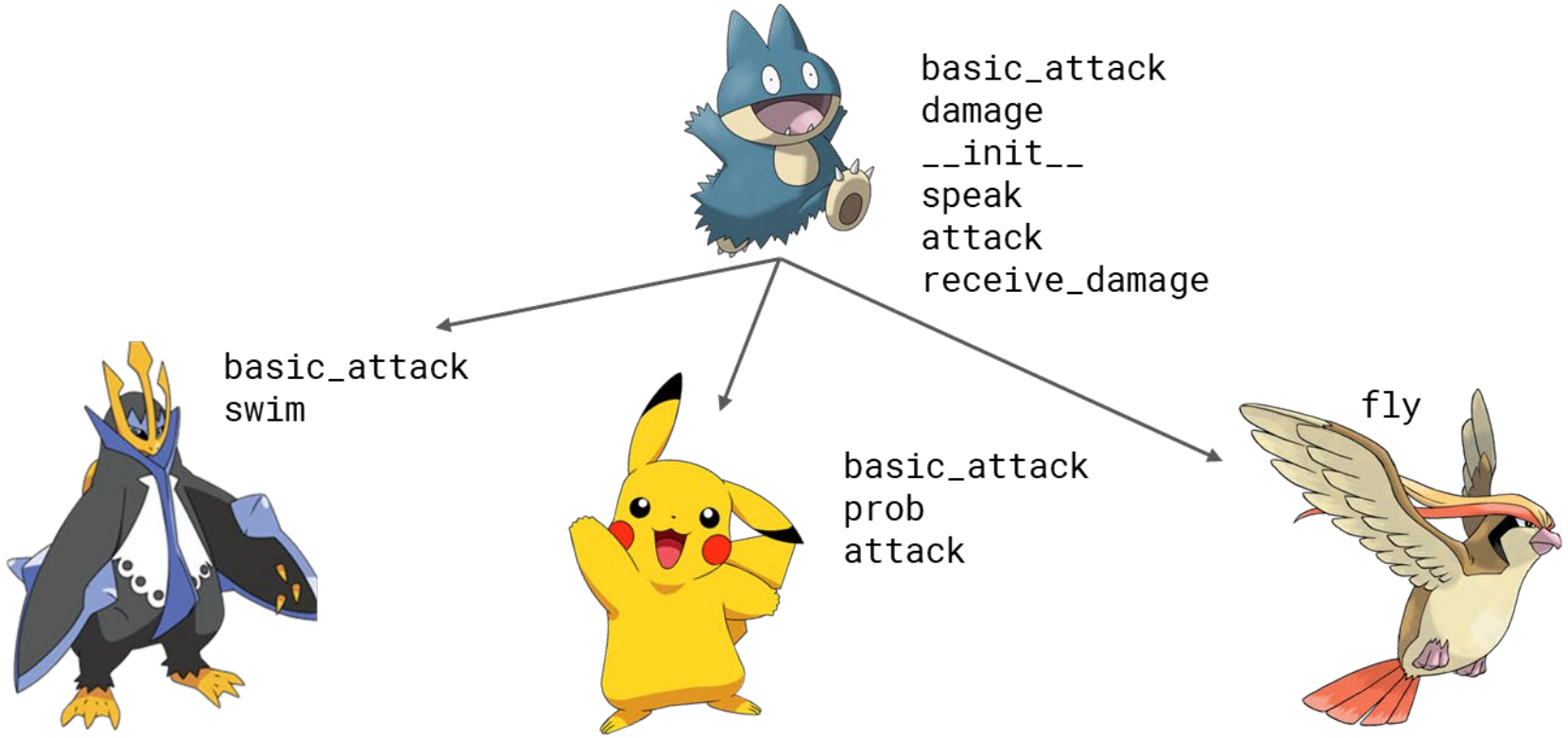
- say their name
- attack other Pokemon
- receive damage

```
class Pokemon:
    basic_attack = 'tackle'
    damage = 40

    def __init__(self, name, trainer):
        self.name, self.trainer = name, trainer
        self.level, self.hp = 1, 50
        self.paralyzed = False

    def speak(self):
        print(self.name + '!')

    def attack(self, other):
        if not self.paralyzed:
            self.speak()
            print(self.name, 'used', self.basic_attack, '!')
            other.receive_damage(self.damage)

    def receive_damage(self, damage):
        self.hp = max(0, self.hp - damage)
        if self.hp == 0:
            print(self.name, 'fainted!')
```

Berkeley CS61a courseware

语言的分类：按计算模型和编程风格

面向对象语言：命令式的一种，数据（属性）和计算（行为）的封装



语言的分类：按计算模型和编程风格

面向对象语言：命令式的一种，数据（属性）和计算（行为）的封装

经典的面向对象语言：

Simula 67 (1967), Smalltalk (1972), C++ (1980), Java (1995)

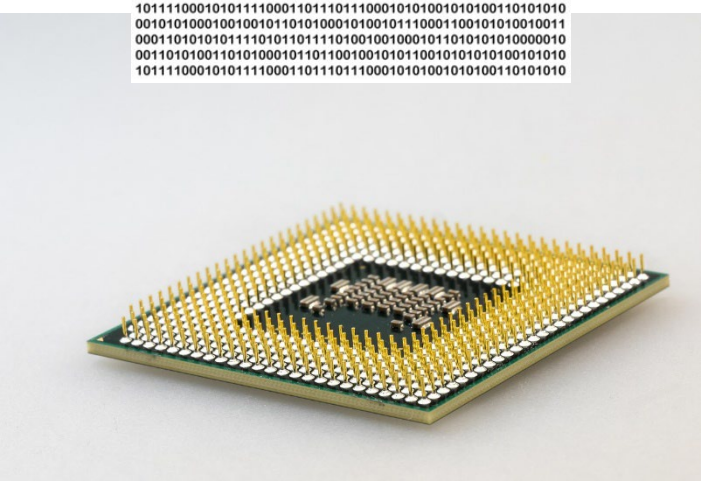
最初的动机：科学仿真

语言的分类：按程序执行方式

```
long factorial(int n)
{
  int c;
  long result = 1;

  for (c = 1; c <= n; c++)
    result = result * c;

  return result;
}
```

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

? ?



如何让计算机理解并执行程序?

语言的分类：按程序执行方式

如何让计算机理解并执行程序?

静态编译型

动态解释型

字节码型（二者结合）

如何让计算机理解并执行程序？
对比现实生活中对自然语言的翻译

提前翻译（如电影的翻译和配音）

提前翻译、配音
整部电影

向观众发布
整部电影

现场翻译

现场逐句翻译
（慢，但方便修改）

听众

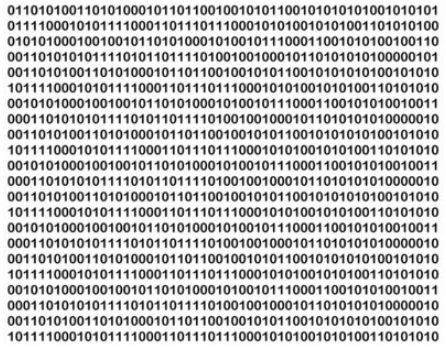
语言的分类：按程序执行方式

静态编译型

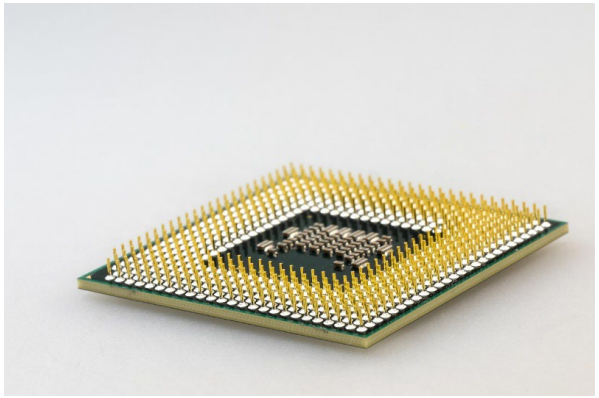
```
long factorial(int n)
{
  int c;
  long result = 1;

  for (c = 1; c <= n; c++)
    result = result * c;

  return result;
}
```

开发

发布



运行



优点：优化容易，执行速度快

缺点：代码难以动态修改

语言的分类：按程序执行方式

动态解释型

```
a = 1
b = 2
c = a + b
print("Hello, world!")
```

开发

发布

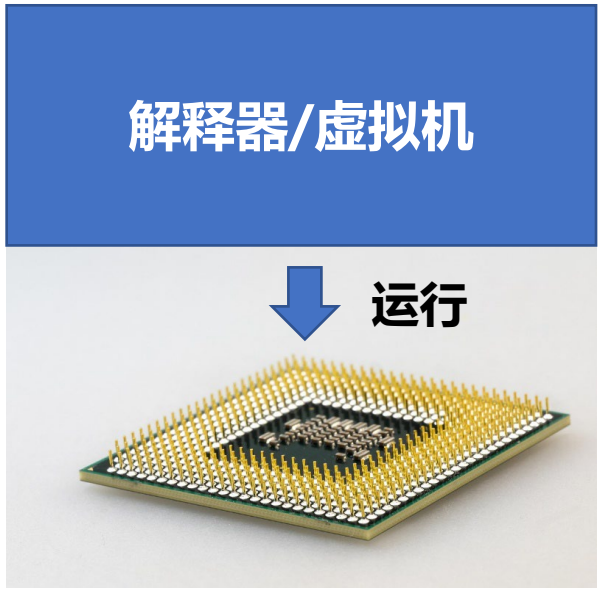
```
a = 1
b = 2
c = a + b
print("Hello, world!")
```

a =1

b =2

c = a+b

运行



优点：写完马上可以执行，无需编译，方便修改

缺点：执行速度慢（没有事先优化）

语言的分类：按程序执行方式

字节码型

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

编译器

```
 1       ILOAD j    // i = j + k
 2       ILOAD k
 3       IADD
 4       ISTORE i
 5       ILOAD i    // if (i < 3)
 6       BIPUSH 3
 7       IF_ICMPEQ L1
 8       ILOAD j    // j = j – 1
 9       BIPUSH 1
10       ISUB
11       ISTORE j
12       GOTO L2
13  L1:  BIPUSH 0     // k = 0
14       ISTORE k
15  L2:
```

开发

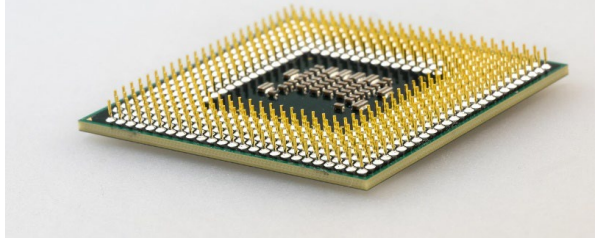
发布

```
 1       ILOAD j    // i = j + k
 2       ILOAD k
 3       IADD
 4       ISTORE i
 5       ILOAD i    // if (i < 3)
 6       BIPUSH 3
 7       IF_ICMPEQ L1
 8       ILOAD j    // j = j – 1
 9       BIPUSH 1
10       ISUB
11       ISTORE j
12       GOTO L2
13  L1:  BIPUSH 0     // k = 0
14       ISTORE k
15  L2:
```

运行

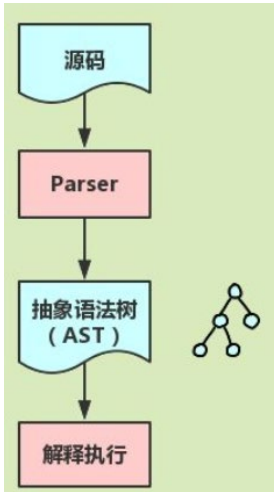
解释器/虚拟机

运行

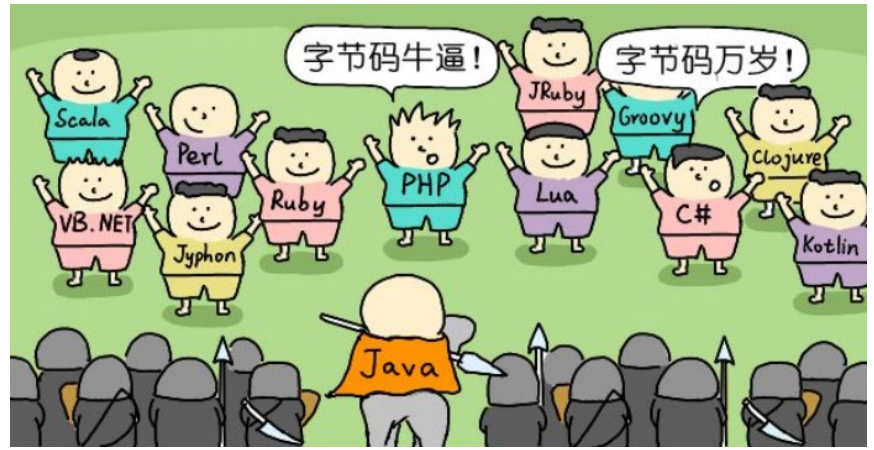


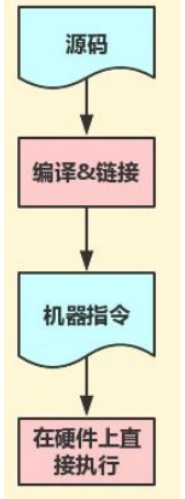
优缺点：介于编译型和解释型之间

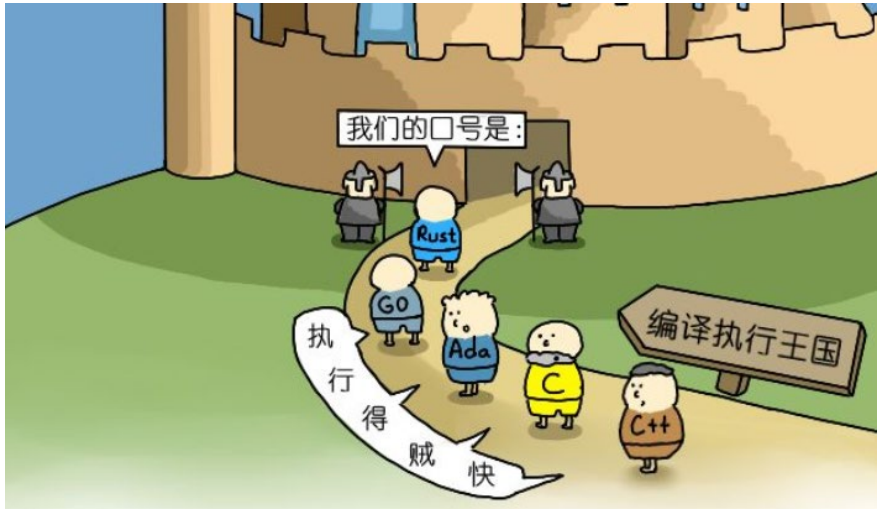
语言的分类：按程序执行方式

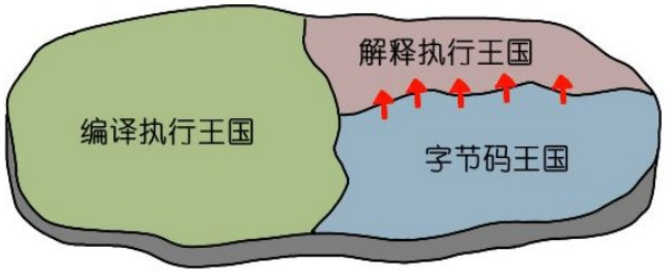


公众号“码农翻身”：《字节码万岁！！！》

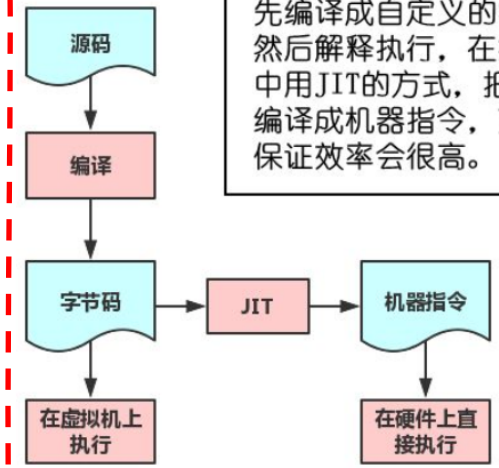








比如我们可以先把Java源码先编译成自定义的字节码，然后解释执行，在执行过程中用JIT的方式，把热点代码编译成机器指令，直接执行，保证效率会很高。



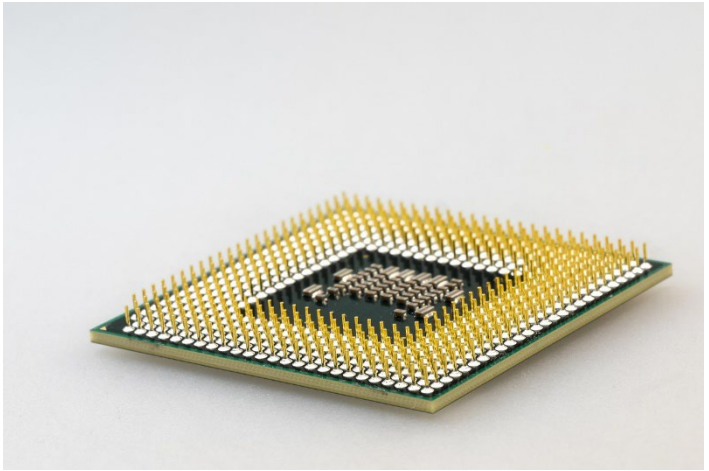
为什么会存在这么多编程语言？

什么样的语言是好的语言？

或者：评价一个语言关心哪些方面？

编程语言领域有哪些新的发展？

什么样的语言是好的语言？



简单、高效、正确的
表达自己的计算任务

用尽量少的时间和资源（内存、
能耗等）完成计算任务

语言设计关注什么？

良好的抽象：

关注业务逻辑，隐藏实现细节

高阶函数

多态：泛型、继承、重载

模块化封装和信息隐藏

可组合性：

由基本元素构造复合元素的能力

数据的组合：复合数据类型的定义

控制的组合：函数、协程、actor等

正确性/安全性：

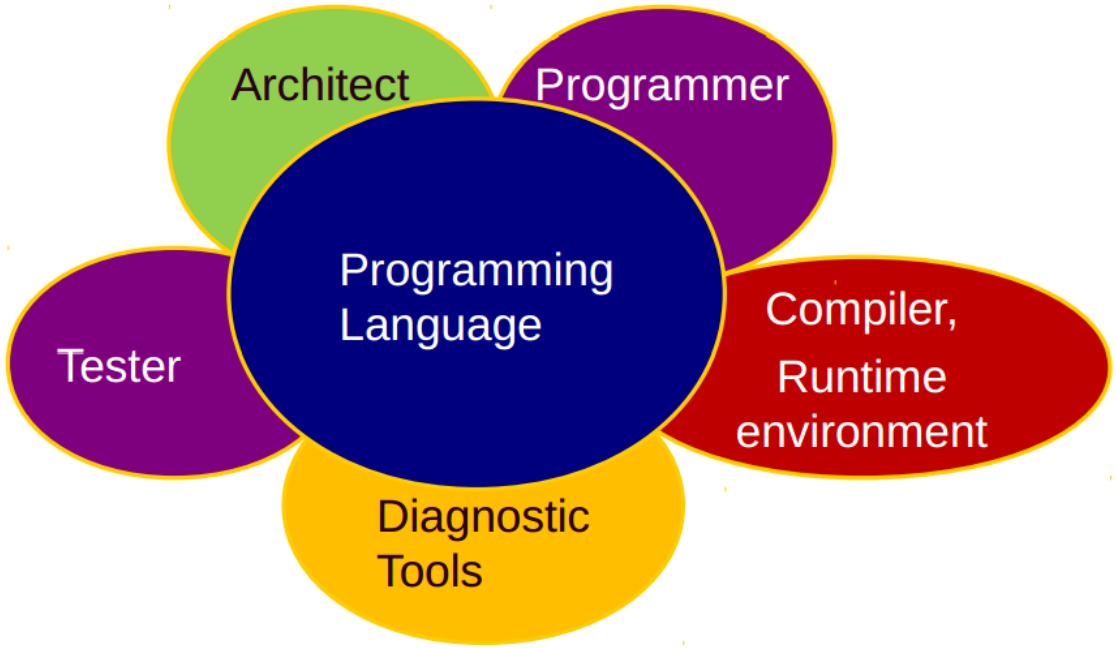
静态类型安全

动态检查：如下标越界检查

程序性能：

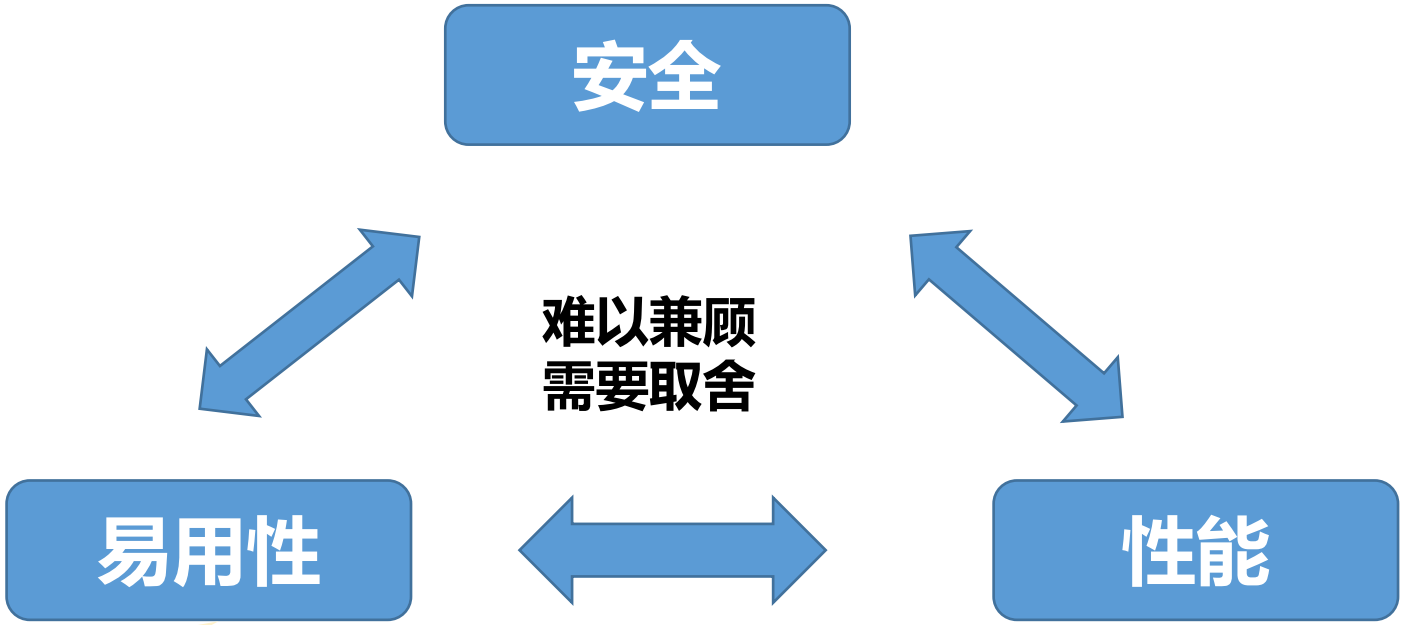
充分发挥硬件能力

例如：并行计算、异构编程等



语言设计关注什么？

类型安全：动态（牺牲性能）/静态（牺牲易用性）
内存安全：垃圾收集（牺牲性能）
动态检查：下标越界、数值溢出（牺牲性能）



动态类型：牺牲安全、性能
自动内存管理：GC/RC，牺牲性能
动态派遣: 牺牲性能

指针算数运算、类型转换：牺牲安全
手工内存管理：牺牲安全、易用性
ownership和生命周期机制：牺牲易用性

抽象能力

隐藏无关细节，专注于特定问题本身

具体

抽象

抽象能力

声明式编程：
描述想要什么（what）

```
add_one = lambda x: x+1
map([1, 2, 3], add_one)
```

过程式编程：
描述如何才能达到想要的效果

```
int mylist[] = {1, 2, 3};
int length = 3;
for (int i = 0; i < length; i++){
    mylist[i] = mylist[i] + 1;
}
```

抽象能力

高阶函数：

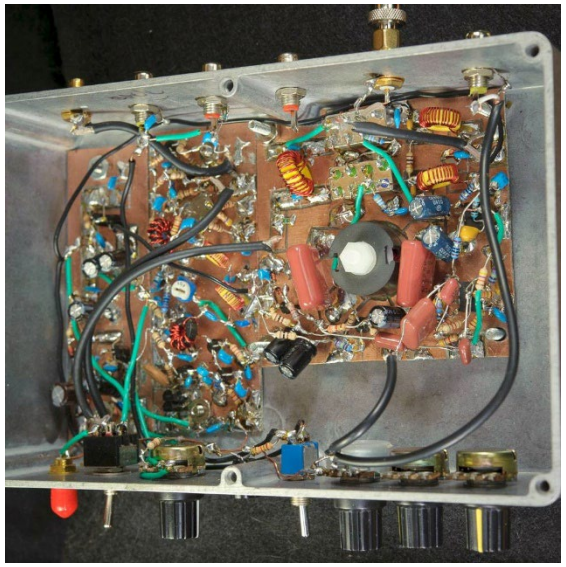
```
add_one = lambda x: x+1
map([1, 2, 3], add_one)
```

泛型：

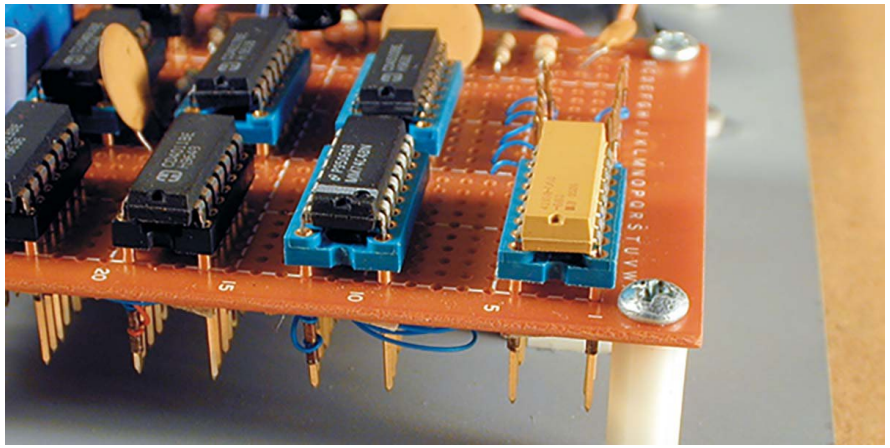
```
class Stack<A>{
 void push(A a) { ... }
 A pop() { ... }
 ...}
```

```
String s = "Hello";
Stack<String> st =
        new  Stack<String>();
st.push(s);
...
s = st.pop();
```

模块化和可复用性 —— 信息隐藏

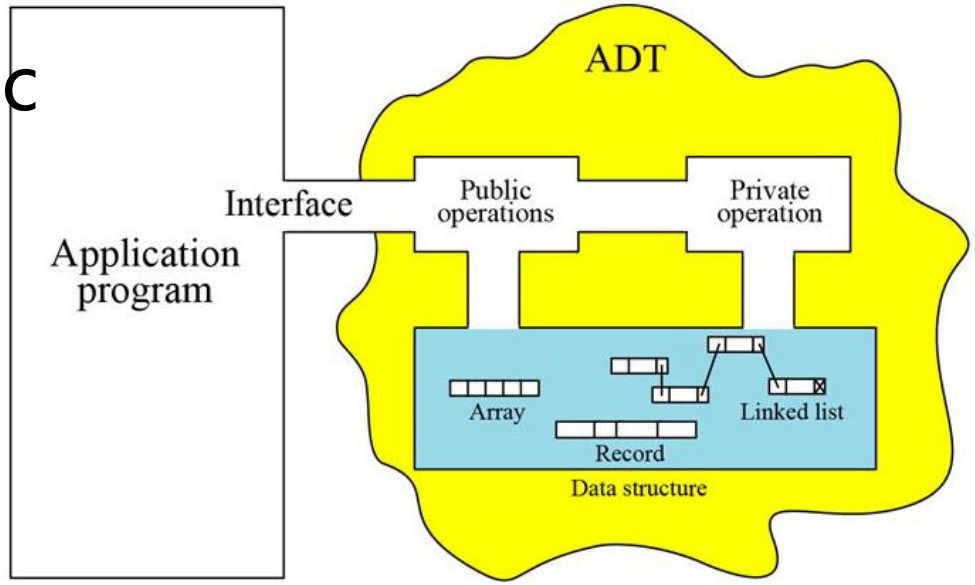


v.s.



模块化和可复用性 —— 信息隐藏

- 对象的封装：private/protected/public
- 抽象数据类型
- 模块系统

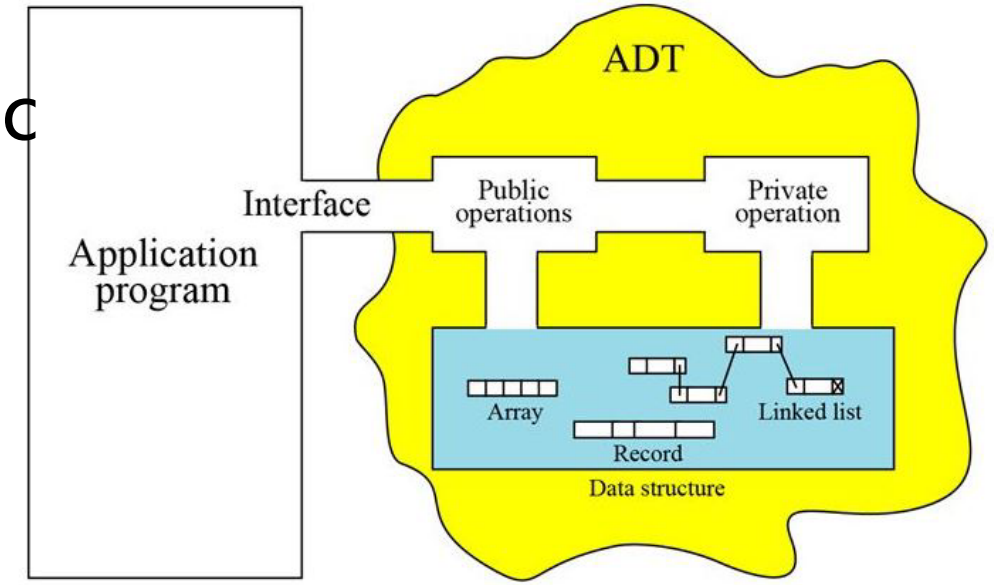



```
functor F( P: EQ ) : EQ =
struct
  type t = P.t * P.t
  fun eq((x,y),(u,v)) = P.eq(x,u) andalso P.eq(y,v)
end;

structure IntEq : EQ =
struct
  type t = int
  val eq = (op =)
end;

structure IntTupleEq = F(IntEq);

IntTupleEq.eq((3,4),(3,4));
```

```
signature EQ =
sig
  type elem
  val eq: elem * elem -> bool
end;
```

模块化和可复用性 —— 信息隐藏

- 对象的封装：private/protected/public
- 抽象数据类型
- 模块系统



当实现发生改变时，使用者不需要相应做修改

正确性和安全性

限制编程的自由度，以减少出错的机会

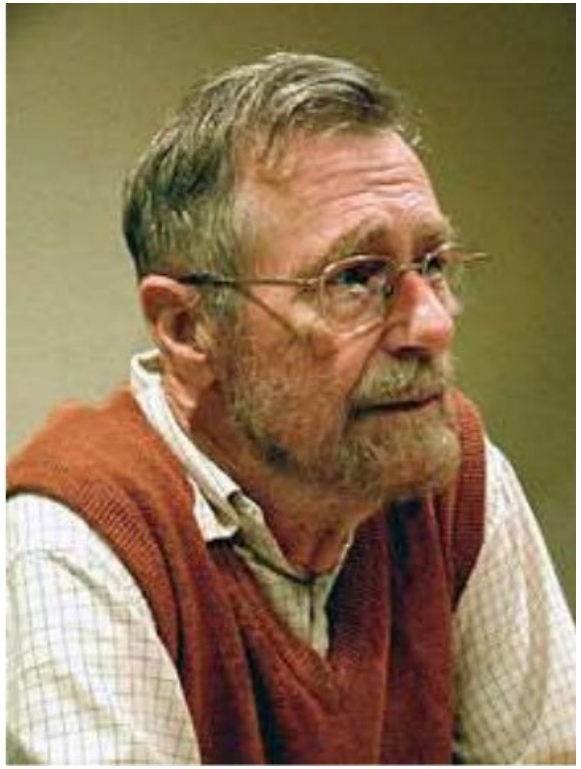
结构化编程

if-then-else, loops, function calls, exceptions …

```
one:
  for (i = 0; i < number; ++i){
    test += i;
    goto two;
  }

two:
  if (test > 5) {
    goto three;
  }
  ...
```

Go-to statement considered harmful (1968)

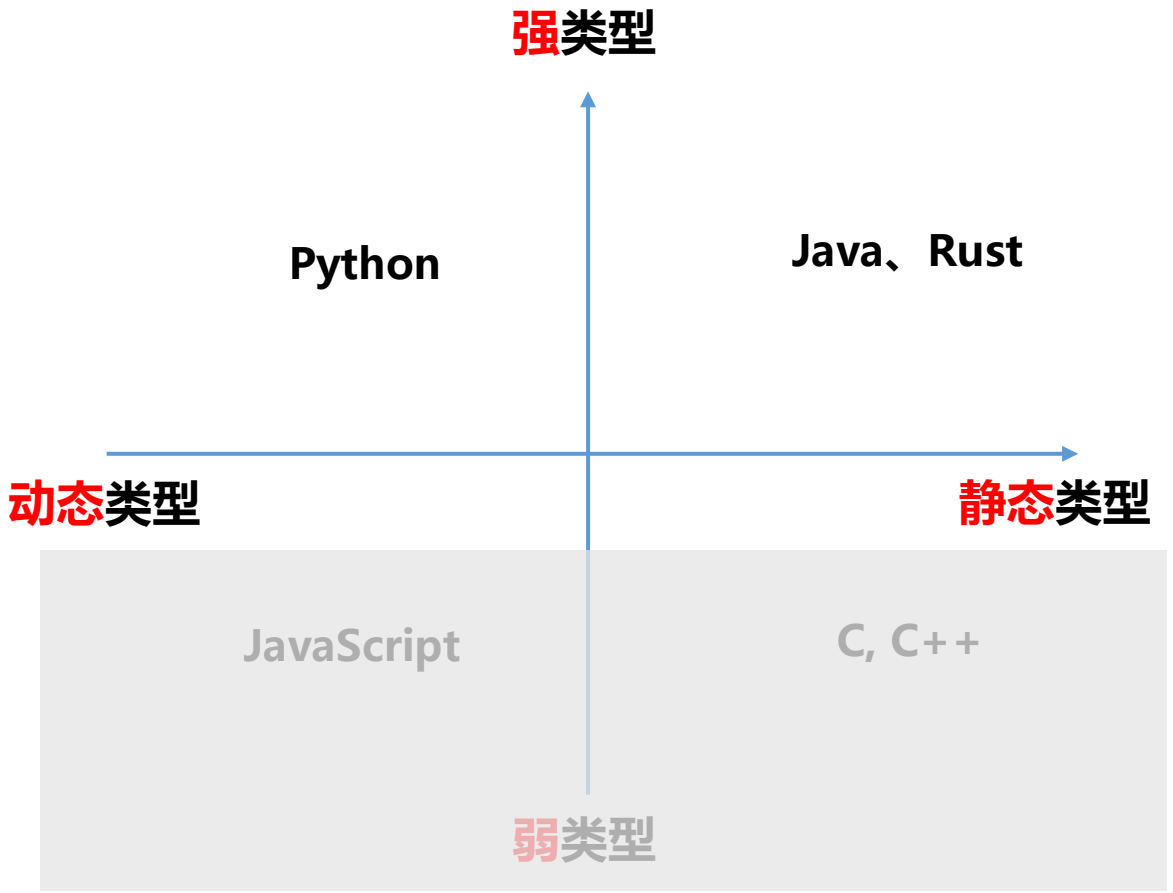
Edsger W. Dijkstra



正确性和安全性 —— 类型系统

- 各种规则约束的表达
  - 及时发现各种程序错误
  - 类型安全、内存安全等

```
int* x, y, z;
int o = 4;
…
z = x + y; ✗
z = o + “123”;✗
```

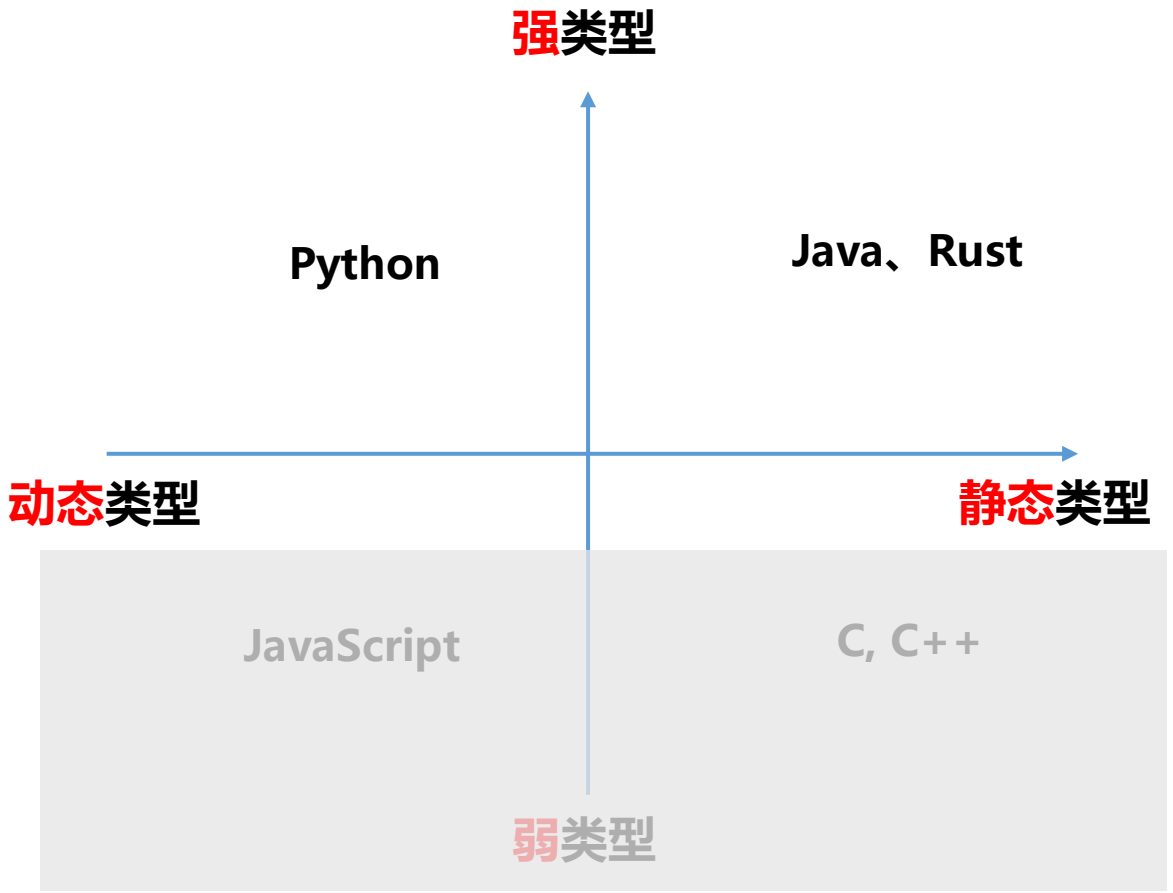


弱类型：难以保证安全性。新的语言中很少采用

正确性和安全性 —— 类型系统

- 各种规则约束的表达
  - 及时发现各种程序错误
  - 类型安全、内存安全等

- 静态类型检查 vs. 动态类型检测
  - 原则：越在开发的早期发现问题，解决问题的代价越小
  - 静态类型：编译时做类型检查
    - 典型案例：Java, Rust, Typescript
  - 动态类型：运行时做类型检查
    - 典型案例：Python



弱类型：难以保证安全性。新的语言中很少采用

类型检查：动态类型 vs. 静态类型

```
"hello, world" + 123 ✗
```

```
int factorial(int n)
{
  int c;
  int result = 1;

  for (c = 1; c <= n; c++)
    result = result * c;

  return result;
}

factorial("hello");  ✗
```

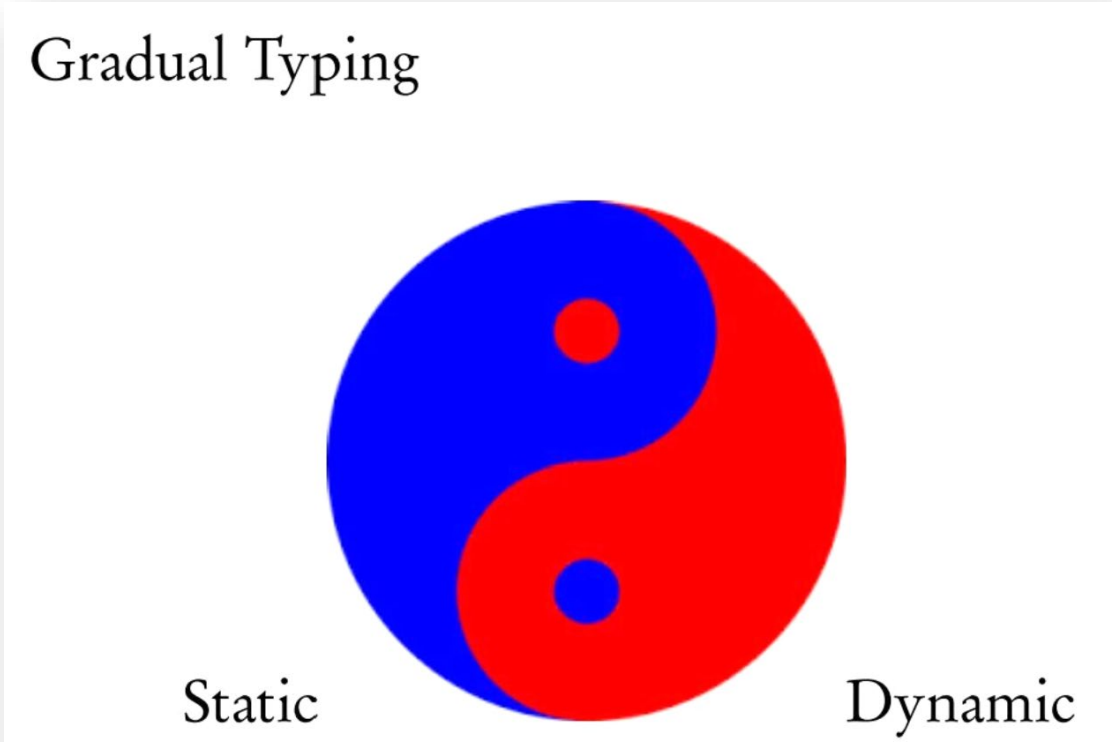
静态类型报错，动态类型不报错，在运行时做类型检查

```
if ( f(x) > 0 ):
  y = "hello";
else:
  y = 3;

if ( g(x) > 0 ):
  z = y + "world";  ? ? ?
else:
  z = y + 4;  ? ? ?
```

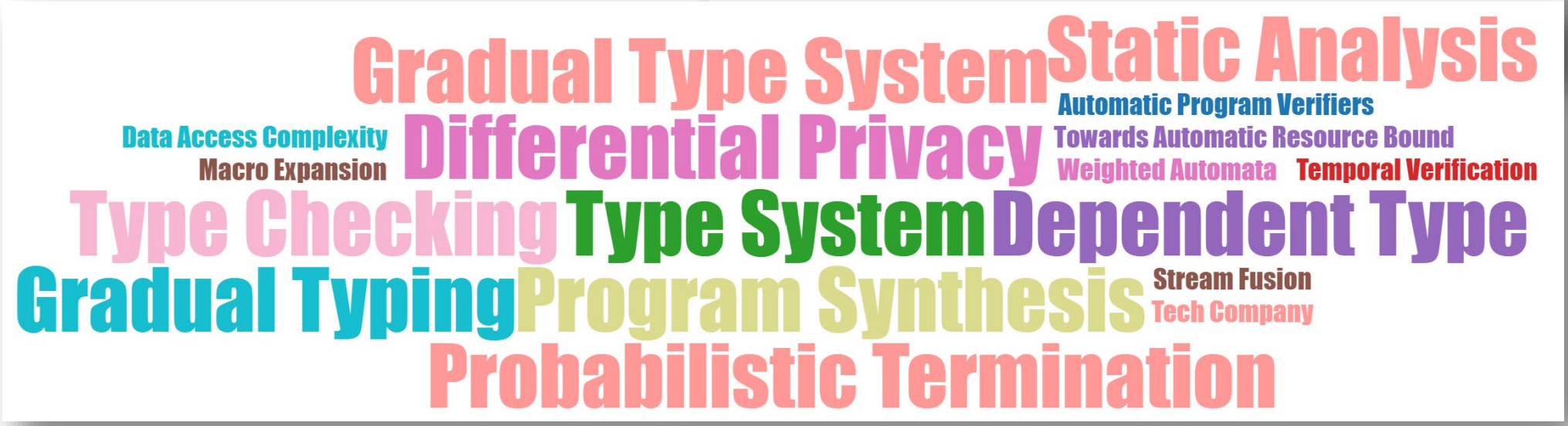
f(x) > 0 iff g(x) > 0 ???

类型系统研究

	表达力/ 自由度	易学习 易上手	程序 安全性	代码可读性/ 可维护性
动态类型	+	+	-	-
静态类型	-	-	+	+



POPL 2015 - 2021



语言的实现

- 高性能运行
  - 编译器、虚拟机/解释器、字节码、JIT、运行时和垃圾收集等
  - 关注特性：性能、内存消耗、能耗、实时响应等
- 编译时长
- 编译得到的二进制/字节码文件的大小

什么样的语言是好的语言 —— 其他因素

- 丰富、易用的库函数和编程框架
- IDE
- 测试工具和框架
- 调试
- 模块/包管理
- ...

编程语言的构成 —— 狭义和广义

构建一个工业级编程语言是一个庞大的系统工程！

为什么会存在这么多编程语言？

什么样的语言是好的语言？

或者：评价一个语言关心哪些方面？

编程语言领域有哪些新的发展？

为什么新的语言不断出现?

PPT from: Java Language Features – Late 2019 Edition
Brian Goetz, Java Language Architect

New Release Cadence

Java 9 – released Sept 2017
- 3 ½ years in the making
- Over 90 JEPs
- Somewhat disruptive release…

Java 10 – March 2018 (new, and already o
- 6 months in the making…
- 12 JEPs

Java 11 – Sept 2018
- First "LTS" release under new cadence
- 17 JEPs

Java 12 – March 2018

Java 13 – Just released

Java 14 – underway…

Java：每6个月一个新版本

语言需要不断演进

Evolution and Programming Languages

Programming Languages

Programming Problems

Computing Hardware

语言的一些发展趋势（1）

- 多范式编程：函数式编程和面向对象编程的融合
  - OCaml (1996), Scala (2003)
  - Python, Javascript
  - C++11, Java 8 (2014)：增加lambda机制
  - Rust (2010), Kotlin (2011), Dart (2011), Swift (2014)等新语言

Scala

Scala is an object-oriented and functional language which is completely interoperable with Java and .NET.

It removes some of the more arcane constructs of these environments instead:

(1) a uniform object model,

(2) pattern matching and higher-order functions,

(3) novel ways to *abstract* and *compose* programs.

OO

FP

所有编程语言的核心任务

The Scala Experiment

–

Can We Provide Better Language Support for Component Systems?

Martin Odersky, EPFL

Invited Talk,
ACM Symposium on Programming Languages and Systems (POPL),
Charleston, South Carolina, January 11, 2006.

Why Unify FP and OOP?

Both have complementary strengths for composition:

Functional programming: Makes it easy to build interesting things from simple parts, using

- higher-order functions,
- algebraic types and pattern matching,
- parametric polymorphism.

Object-oriented programming: Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions.

语言的一些发展趋势（2）

- 更加注重安全
  - Null safety: non-nullable
    - 告别空引用异常

“Null References: The Billion Dollar Mistake”
- Tony Hoare

```
var a: String = "abc"
a = null  // 编译错误
val l = a.length  // ok, a必然不会是null
```

```
var b: String? = "abc" // can be set null
b = null   // ok
val l = b.length   // 编译错误, b可能是null
val l = if (b != null) b.length else -1  // ok
```

语言的一些发展趋势（2）

“Null References: The Billion Dollar Mistake”
- Tony Hoare

- 更加注重安全
  - Null safety: non-nullable
    - 告别空引用异常

- 动态类型语言中引入静态类型检查机制
  - 能够在写程序的时候尽量多的发现错误
  - Typescript
  - Python type hints

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

```
def f(x, y):
  if x >100:
    return x
  else
    return x + y

def g(): // 静态类型报错
  return f(100, “123”)

>>> g() // 动态类型报错
```

语言的一些发展趋势（2）

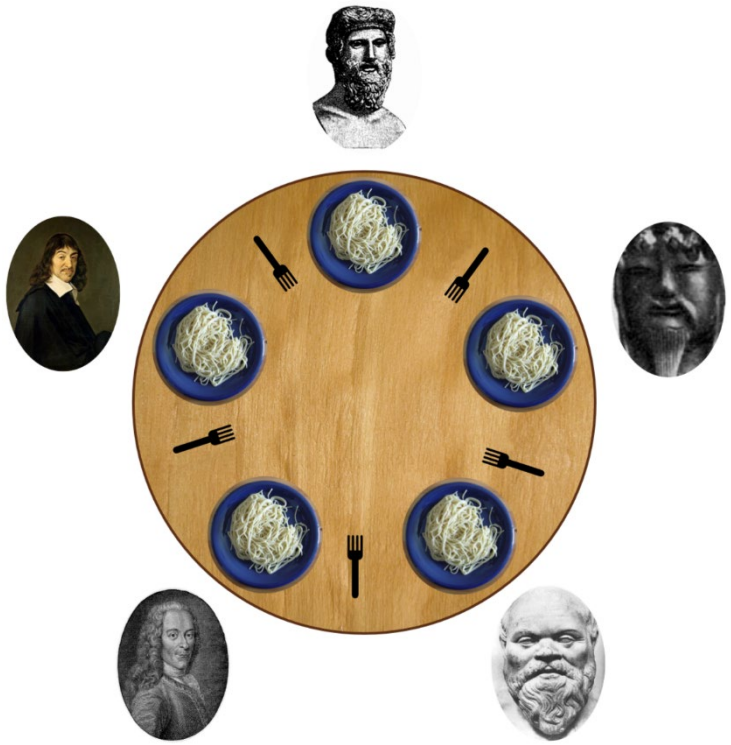
“Null References: The Billion Dollar Mistake”
- Tony Hoare

- 更加注重安全
  - Null safety: non-nullable
    - 告别空引用异常
  - 动态类型语言中引入静态类型检查机制
  - 可信系统编程语言
    - C++安全特性持续增强
    - Rust
    - Verona：微软开发中

      https://github.com/microsoft/verona

语言的一些发展趋势 – 学术前沿（1）

- 面向特定领域的语言设计和实现
  - 机器学习
  - 概率编程
  - 量子编程
  - 网络编程
  - 并行、并发、分布式编程

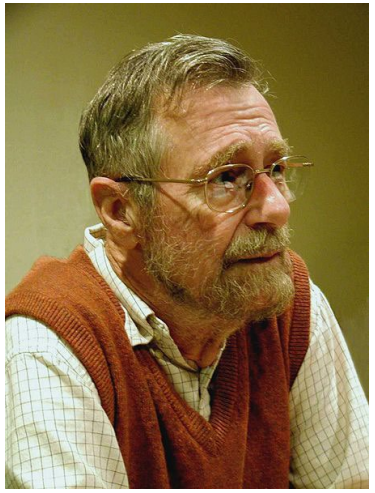


语言的一些发展趋势 – 学术前沿（2）

- 语言的形式化语义和程序的形式化验证
  - 用数学方法来理解和研究程序的行为
  - 能否证明一个程序一定是正确的?

Testing shows the presence, not the absence of bugs.

Edsger W. Dijkstra



```
{ n ≥ 0 }
long factorial(int n)
{
  int c = n;
  long result = 1;

  while (c>1) {
    result = result * c;
    c = c-1;
  }

  { result = n!/c! ∧ c=1}

  return result;
}
{ result = n! }
```

{ result = n!/c!}

编程语言发展趋势

The Next 700 Programming Languages

[CACM 9(3), 1966]

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

Peter Landin

ISWIM语言（族）：

1）嵌套结构和缩进；

2）函数作为一等公民、lambda表达式

3）赋值和控制流语句相关的命令式语言特性

编程语言发展趋势

The Next 7000 Programming Languages

[LNCS, vol. 10,000, 2019]

Robert Chatley[1], Alastair Donaldson[1], and Alan Mycroft[2(✉)]

[1] Department of Computing, Imperial College, London, UK
{robert.chatley,alastair.donaldson}@imperial.ac.uk
[2] Computer Laboratory, University of Cambridge, Cambridge, UK
alan.mycroft@cl.cam.ac.uk



Darwinian evolution in the context of programming languages …

区分看待物种（完整语言）的成功和基因（语言概念和特性）的成功

气候变化 ←→ 硬件/基础设施的变化 → 语言的兴衰/进化

共生体对物种繁荣的贡献 ←→ 库/工具对语言生态的贡献

编程语言发展趋势

The Next 7000 Programming Languages

[LNCS, vol. 10,000, 2019]

Robert Chatley[1], Alastair Donaldson[1], and Alan Mycroft[2(✉)]

[1] Department of Computing, Imperial College, London, UK
{robert.chatley,alastair.donaldson}@imperial.ac.uk
[2] Computer Laboratory, University of Cambridge, Cambridge, UK
alan.mycroft@cl.cam.ac.uk



趋势预测：

1. 安全系统编程：新语言（Rust）、C/C++安全增强（包括软硬件结合技术）等
2. Gradual typing融合动态类型、静态类型、程序验证等特性
3. 并行编程：难以统一，反而可能更加碎片化
4. 语言对分布式容错编程的支持（类似Erlang）
5. 语言与软工工具的深度结合
6. 程序合成和AI对IDE工具的改进
   （但难有大的突破）

大模型时代，是否还需要编程语言？

On the foolishness of “natural language programming”
论"自然语言编程"的愚蠢
Edsger W.Dijkstra, EWD667, 1978

In order to make machines significantly easier to use, it has been proposed (to try) to design machines that we could instruct in our native tongues. this would, admittedly, make the machines much more complicated, but, it was argued, by letting the machine carry a larger share of the burden, life would become easier for us. It sounds sensible provided you blame the obligation to use a formal symbolism as the source of your difficulties. But is the argument valid? I doubt.

为了让机器更"易于使用"，有人提议设计能听懂自然语言的计算机——这样固然会让机器系统变得更复杂，但支持者认为，让机器多分担些工作，人类就能轻松许多。乍听之下挺合理，前提是你真把"必须使用形式化符号"当作麻烦的根源。但这种论调站得住脚吗？我深表怀疑。

知乎

关于“自然语言编程”的愚蠢。Dijkstra锐评AI编程

mikubest
在非平凡的困惑中寻找答案可能于事无补，但你依然可以试试看

+ 关注他

https://zhuanlan.zhihu.com/p/1895989957333607441

The virtue of formal texts is that their manipulations, in order to be legitimate, need to satisfy only a few simple rules; they are, when you come to think of it, an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid.

形式化文本的精妙之处在于：其操作过程只需遵守少量简单规则即可确保合法性。细想之下，这实在是种惊人的高效工具——它能剔除各类荒谬错误，而这些错误在我们使用自然语言时几乎无可避免。

My guess is that a hundred years from now people will still tell computers what to do using programs we would recognize as such. There may be tasks that we solve now by writing programs and that in a hundred years you won't have to write programs to solve, but I think there will still be a good deal of programming of the type we do today.

"Hackers & Painters", Paul Graham, 2003

大模型时代编程语言思考

大模型时代，自然语言会不会取代编程语言，或者成为新的编程语言？

不会，但形态可能会发生变化

编程语言的双重属性：**人-机交流**和**人-人交流**

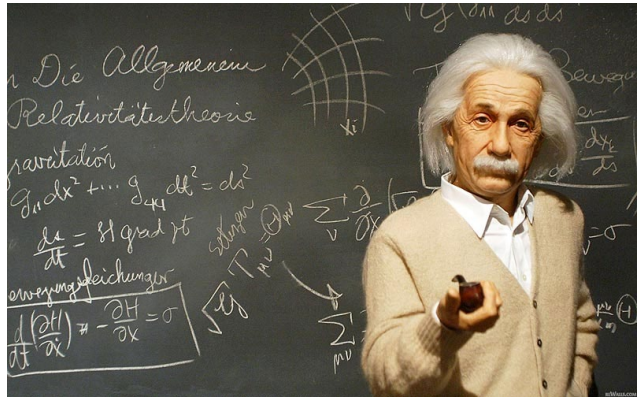
编程语言作为一种**无二义性的数学语言**，是**软件工程师交流**的重要媒介，是软件资产/智力资产的重要承载工具

编程语言之于软件，类似于数学之于（自然）科学

大模型时代编程语言思考

编程语言之于软件，类似于数学之于（自然）科学

科学家足够智能，对自然语言使用和理解无碍，
但仍然需要数学作为表达和交流的工具



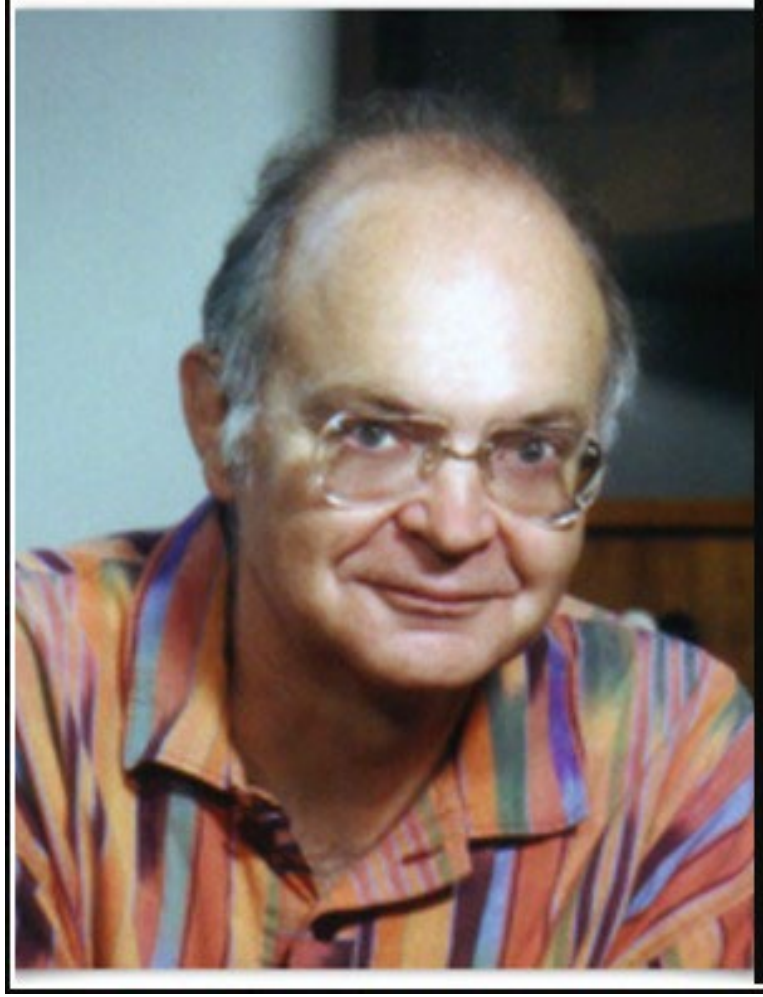
需要人来读代码

代码不会消失，编程语言就不会消失，但编程语言的形态可能会发生变化

语言设计现状：强调**易写易读**，很多时候**易写 > 易读**

大模型时代：代码生成代价变小，但审核/验证变得重要，**易读 >> 易写**

我们需要让程序更容易**审核/测试/验证**的编程语言



Programming is the art of telling another human being what one wants the computer to do.

— Donald Knuth —

AZ QUOTES