

# 大咖助阵 | 海纳：聊聊语言中的类型系统与泛型

2022-02-14 海纳

《Tony Bai · Go语言第一课》

课程介绍 >



讲述：海纳

时长 12:57 大小 11.88M



你好，我是海纳，是极客时间 [《编程高手必学的内存知识》](#) 专栏的作者。

我们知道，编程语言中有非常重要的一个概念，就是数据类型。类型的概念伴随着我们学习一门具体语言的全过程，也深入到了程序员的日常开发之中。所以对于现代程序员而言，了解语言中的类型系统是一项非常重要的技能。

这一节课，我会简单地介绍什么是类型，类型的作用，以及由简单类型推导来的泛型编程的基本概念，接着再比较 **C++** 和 **Java** 两种语言的泛型实现。很多新的编程语言的泛型实现都有它们的影子，所以了解 **C++** 和 **Java** 泛型，会有助于你理解泛型设计的基本概念。

领资料

通过这节课的学习，你会得到一种新的学习语言的视角，那就是从类型的角度去进行分析。

比如我们在学习一门新的语言的时候，可以考虑以下几个问题：

1. 这门语言是强类型的吗？
2. 这门语言是动态类型吗？
3. 它支持多少种内建类型呢？
4. 它支持结构体吗？
5. 它支持字典 (Recorder) 吗？
6. 它支持泛型吗？
7. ....

这样，当我们拿到一门新的语言的规范（Specification）文档后，就可以带着这些问题去文档中寻找答案。等你把这些问题搞明白了，语言的很多特性也就掌握了。这是很多优秀程序员可以短时间内掌握一门新语言的秘技之一。

接下来，我们就从类型的基本概念开始讲起。

## 什么是类型

编程语言中的变量都是有类型的，而且变量的类型不一定一致。例如 Go 语言中的 `int` 和 `float` 声明的变量，它们的类型就不一致，如果你直接对它们执行加操作，Go 的编译器就会报错，很多隐式类型转换带来的问题，在编译阶段就可以发现了。比如，你可以看下面这个例子：

 复制代码

```
1 func main() {
2     var a int = 1
3     var b float64 = 1000.0
4     fmt.Print(a + b)
5 }
```

这种情况下，Go 的编译器会报这样的错误：`invalid operation: a + b (mismatched types int and float64)`。这就说明 Go 语言不支持整型和浮点型变量的加操作。

 领资料

相比 Go 语言，JavaScript 在类型上的要求就宽松很多，比如整数与字符串的加法操作，JavaScript 会把整数转换成字符串，然后再与目标字符串进行拼接操作。显然 Go 语言会对语言类型进行严格检查，我们就说它的**类型强度**高于 JavaScript。

Go 语言的类型系统还有一个特点，那就是一个变量声明成什么类型的，就不能再更改了。与之形成鲜明对比的是 Python。它们都具有比较高的类型强度，但是类型检查的时机不同。Go 是在编译期，而 Python 则是在运行期。我们看一个 Python 的例子：

 复制代码

```
1 >>> a = 1
2 >>> a + "hello"
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: unsupported operand type(s) for +: 'int' and 'str'
6 >>> a = "hello"
7 >>> a + "world"
8 'helloworld'
```

从上面的例子中，我们可以看到，Python 的类型检查也是比较严格的，在第 2 行，将一个整型值和字符串值相加是会引发 `Traceback` 的。但是一个变量却可以先使用整数为它赋值，再使用字符串为它赋值（第 6 行），然后再进行字符串值的加操作就没有问题了。这就说明了变量的类型是在程序运行的时候才去检查，而不是编译期间进行的。我们把这种语言称为**动态类型语言**。

同样的例子如果使用 Go 来实现，编译器就会报错：

 复制代码

```
1 var a int = 1
2 a = "hello"
3 fmt.Print(a)
```

由此可见，Go 语言是一种**静态的强类型语言**。

动态类型不仅仅表现在变量的类型可以更改，在面向对象的编程语言中，动态类型往往还意味着类的定义也可以动态更改。我们仍然以 Python 为例来观察动态类型的特点：

 领资料

 复制代码

```
1 >>> class A():
2 ...     pass
3 ...
4 >>> A.a = 1
5 >>> a = A()
```

从上述例子中，我们可以看到，类 **A** 的类属性是可以在运行时进行添加和修改的。这与静态编译的语言非常不同。

由此，我们可以得出结论，动态类型相比静态类型，它的优点在于：

1. 动态类型有更好的灵活性，在运行时可以修改变量的类型，也可以对类定义进行修改，所以针对动态类型语言的热更新就更容易设计；
2. 动态类型语言写起来很方便，非常适合用来编写小规模脚本。

同时，动态类型也往往具有一些缺点（通常是这样，但并不绝对）。

首先，动态类型语言的代码不容易阅读。据统计，程序员的日常工作中 **90%** 的时间是在阅读别人写的代码，只有不足 **10%** 的时间才是在开发新的功能。而动态类型语言，没有类型标注，代码会非常难懂，即使有一些动态类型语言有类型标注的，但因为可以运行时修改类型，往往会出现一个类的属性在不同的地方被修改的情况，这使得代码的阅读和维护变得困难；

第二点是，动态类型语言的性能往往会差一些，比如 **Python** 和 **JavaScript**，因为在编译期间缺少类型提示，编译器无法为对象安排合理的内存布局（你可以参考 [内存课导学三](#)），所以它们的对象布局相比 **Java/C++** 等静态类型语言会更加复杂，同时这也会带来性能的下降。

由此可见，我们并不能简单地说，**静态类型** 就比 **动态类型** 好，或者 **强类型** 就比 **弱类型** 好，还是要根据具体的场景来进行取舍。

比如要求快速开发，规模较小的工具，人们常常会选择使用 **Python**，而多人合作的大型项目，人们就会选择使用 **Java** 之类的静态强类型语言。

另外，类似 **int**、**String** 这种类型往往是语言的内建类型，而语言的内建类型在表达力上经常是不够的，这就需要人们通过将简单内建类型组合起来实现相应的功能，这就是 **复合类型**。

典型的复合类型包括枚举、结构、列表、字典等。这些类型在 **Go** 语言中都有相应的定义，你可以参考 **Go** 语言专栏进行学习。



在讲完了类型的基本概念以后，我们再讲解一个类型系统中非常常见，同时也是比较困难的一个话题，那就是泛型。

## 为什么要使用泛型

我们使用一个实际的例子来讲一下为什么要使用泛型。比如，这里有一个栈的 C++ 实现，栈里可以存放的变量是整型的，它的代码如下所示：

 复制代码

```
1  #include <iostream>
2  using namespace std;
3
4  class Stack {
5  private:
6      int _size;
7      int _top;
8      int* _array;
9
10 public:
11     Stack(int n) {
12         _size = n;
13         _top = 0;
14         _array = new int[_size];
15     }
16
17     void push(int t) {
18         if (_top < _size) {
19             _array[_top++] = t;
20         }
21     }
22
23     int pop() {
24         if (_top > 0) {
25             return _array[--_top];
26         }
27         return -1;
28     }
29 };
30
31 int main() {
32     Stack stack(3);
33     stack.push(1);
34     stack.push(2);
35     cout << stack.pop() + stack.pop() << endl;
36     return 0;
37 }
```

领资料

运行这个程序，一切看上去都还不错。但是，假如我们需要一个管理浮点数栈，或者管理字符串的栈，就不得不再将上述逻辑重新实现一遍。除了 `_array` 的类型不一样之外，整数栈和浮点数栈的逻辑都是相同的。这就会带来大量的重复代码，不利于工程代码的维护。

为了解决这个问题，很多带有类型的语言都引入了泛型。以 C++ 为例，泛型的栈可以这么实现：

 复制代码

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 class Stack {
6 private:
7     int _size;
8     int _top;
9     T* _array;
10
11 public:
12     Stack(int n) {
13         _size = n;
14         _top = 0;
15         _array = new T[_size];
16     }
17
18     void push(T t) {
19         if (_top < _size) {
20             _array[_top++] = t;
21         }
22     }
23
24     T pop() {
25         if (_top > 0) {
26             return _array[--_top];
27         }
28         return T();
29     }
30 };
31
32 int main() {
33     Stack<int> stack(3);
34     stack.push(1);
35     stack.push(2);
36     cout << stack.pop() + stack.pop() << endl;
37     Stack<string> sstack(3);
38     sstack.push("hello ");
39     sstack.push("world!");
40     cout << sstack.pop() + sstack.pop() << endl;
```

领资料





```
41     return 0;
42 }
```

执行这段代码，可以看到，控制台上可以成功打印出来 3 和 "hello world"。

这段代码的巧妙之处在于，栈的核心逻辑，我们只写了一遍（第 4 行至第 30 行），然后只需要使用一行简单的代码就可以创建用于存储整数的栈（第 33 行）和用于存储字符串的栈（第 37 行）。

使用这种方式，可以帮助我们节约大量的时间和代码篇幅。接下来，我们看一下 C++ 编译器是如何处理这段代码的。在 Linux 系统上，我们可以使用以下命令对这个文件进行编译，然后查看它的编译结果：

 复制代码

```
1 $ g++ -o stack -g stack.cpp
2 $ objdump -d stack
```

从这个结果中，可以看到，C++ 编译器生成了两个 push 方法，其参数类型分别是整型和字符串类型。也就是说，在 C++ 中，泛型类型在被翻译成机器码的时候，是真的创建了两不同的类型。

泛型使用最广泛的场景就是容量类，例如 vector、list、map 等等。C++ STL 中定义的容器类都是以模板的形式提供的。

我们可以再使用一种新的视角来理解泛型，那就是可以将泛型声明看作是类型之间的转换关系，或者换种说法，就是我们可以使用一种类型（甚至是值）得到另外一种新的类型。

## 泛型：使用类型得到新的类型

现在我们就用这个新视角来理解泛型，把泛型声明看成是一个输入参数是类型，返回值也为类型的函数。我使用一个 vector 的例子来说明这一点：

 复制代码

```
1 int main() {
2     vector<int> vi;
3     vector<double> vd;
```

领资料

```
4     vector* p = &v1;
5     return 0;
6 }
```

这里程序的第 4 行会报错，报错的信息显示“vector”并不是一个有效的类型。而“vector<int> ”和“vector<double>”则是有效的类型。从这个例子中，我们观察到，vector 类型必须指定一个类型参数才能变成一个有效的类型。所以我们可以把

 复制代码

```
1 template <typename T> class vector;
```


看成是一个函数，它接受一个类型 int 或者 double，得到一种新的类型 vector<int>，或者 vector<double>。

在 C++ 中，更神奇的是，泛型的类型参数不仅仅可以是一种类型，还可以是一个具体的值，例如：

 复制代码

```
1 template <int n> class A;
2
3 int main() {
4     A<0> a;
5     A<1> b;
6     return 0;
7 }
```

在上述代码中，A<0> 和 A<1> 分别是两个不同的类型。使用这种方法，我们可以在编译期间通过模板让编译器帮我们做一些计算，例如：

 复制代码

```
1 #include <iostream>
2 using namespace std;
3
4 template <int n>
5 struct fib {
6     static const int v = fib<n-2>::v + fib<n-1>::v;
7 };
8
9 template <>
10 struct fib<1> {
```

领资料





```

11     static const int v = 1;
12 };
13
14 template <>
15 struct fib<0> {
16     static const int v = 1;
17 };
18
19 int main() {
20     cout << fib<10>::v << endl;
21     return 0;
22 }

```

在这个例子中，编译以后的结果，`fib<10>::v` 会被直接替换成 `55`。这个计算的过程是由编译器完成的。

编译器会把 `fib<10>`，`fib<9>` 等等都看成一种类型。当编译器要计算 `fib<10>` 的值的时候就会先求解 `fib<9>` 和 `fib<8>` 的值，这样一直递归下去，就会找到 `fib<1>` 和 `fib<0>` 这里。而这两个值，我们已经提供了（第 9 行到第 18 行），递归就会结束。

在这个例子中，我们就看到了类型依赖于值的情况。

了解了 C++ 的泛型设计以后，我们再来看一下 Java 语言的泛型实现。

## Java 中的泛型实现

Java 语言的库的分发，往往采用这种形式：Java 的源代码会先被翻译成字节码文件，然后这些文件又会被打包进 jar 文件。jar 文件可以在网络上进行发布。

Java 的一个特性是，相同的字节码文件在不同的体系结构和平台上的行为都是相同的，再加上要做到对低版本代码的兼容，所以 Java 的泛型设计和 C++ 的差异就很大。总的来说，Java 的泛型设计是使用了一种叫做“泛型擦除”的办法来实现的。

我举一个例子来说明“泛型擦除”是怎么一回事。请看下面的代码：

领资料



复制代码

```

1 import java.util.ArrayList;
2
3 class Playground {
4     public static void main(String[ ] args) {

```

```

5     ArrayList<Integer> int_list = new ArrayList<Integer>();
6     ArrayList<String> str_list = new ArrayList<String>();
7     System.out.println(int_list.getClass() == str_list.getClass());
8 }
9 }

```

这段代码的输出是 **true**。

如果按照上一小节中关于 **C++** 泛型的实现，`ArrayList<Integer>` 和 `ArrayList<String>` 应该是不同的两种类型。但这里的结果却是 **true**，这是因为 **Java** 会把这两种 `ArrayList` 的泛型都擦除掉，从而导致整个程序中只有一种类型。

我们这里再举一个例子帮你理解一下 **Java** 的泛型：

 复制代码

```

1  import java.util.ArrayList;
2
3  class Playground {
4      public static void main(String[ ] args) {
5          System.out.println("Hello World");
6      }
7
8      public static void sayHello(ArrayList<String> list) {
9
10     }
11
12     public static void sayHello(ArrayList<Integer> list) {
13
14     }
15 }

```

这里，第 8 行定义的 `sayHello` 方法和第 12 行定义的 `sayHello` 方法是方法重载。我们知道，方法的重载的基本条件是两个同名方法的参数列表并不相同。



从字面上看，第一个 `sayHello` 方法的参数类型是 `ArrayList<String>`，第二个方法的参数类型是 `ArrayList<Integer>`，所以可以实现方法的重载。但是当我们尝试编译上述程序的时候，却会得到这样的错误提示：



 复制代码

```

1  Playground.java:12: error: name clash: sayHello(ArrayList<Integer>) and sayHell

```

```
2     public static void sayHello(ArrayList<Integer> list) {  
3  
4 1 error
```

这是因为当对泛型进行擦除以后，两个 `sayHello` 方法的参数类型都变成了 `ArrayList`，从而变成了同名方法，所以就会出现命名冲突报错。

通过上面两个例子，我们就能感觉到 **C++** 泛型和 **Java** 泛型的不同之处了。它们之间最核心的区别是 **C++** 不同的泛型参数会得到一种新的类型；而 **Java** 则不会，它会进行类型擦除，从而导致表面上不同的类型参数实际上指代的是同一种类型。

## 总结

在这节课里，我们先了解到什么是类型系统，并介绍了什么是强类型和弱类型，什么是静态类型和动态类型。然后通过举例来说明 **Python**，**JavaScript**，**Go** 和 **C++** 各自的类型系统的特点。

从这些例子中，我们看到静态强类型语言更容易阅读和维护，但灵活性不如动态弱类型语言。所以动态弱类型语言往往都是脚本语言，不太适合构建大型程序。

接下来，我们简单介绍了泛型的概念。我们使用了一个栈的例子来说明了使用泛型可以提高编程效率，节省代码量。**Go** 语言从 **1.18** 开始也支持泛型编程。

然后我们又提供了一个新的视角来理解泛型，这种新的视角是把泛型类看成是一种函数，它的输入参数可以是类型，也可以是值，它的返回值是一种新的类型。

最后，我们介绍了 **C++** 的泛型实现和 **Java** 的泛型实现。**C++** 不同的泛型参数会得到一种新的类型，这个过程我们也会称它为泛型的实例化。而 **Java** 则会进行类型擦除，从而导致表面上不同的类型参数实际上指代的是同一种类型。

领资料



分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

上一篇 大咖助阵 | 大明：Go泛型，泛了，但没有完全泛

下一篇 结束语 | 和你一起迎接Go的黄金十年

## 更多课程推荐

# 陈天 · Rust 编程第一课

## 实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



会员免费

¥199

200+ 原理图，详解 Rust 设计理念

## 精选留言 (1)

💬 写留言



一步

2022-02-14

typescript 的泛型的实现 应该是借鉴了 C++ 的实现，都是产生一个新的类型



👍 3

📄 领资料