从泛型 (Generics)
到元编程 (Metaprogramming)

曹焕琦 @ TUNA

前言

- 本次分享假定各位已经具备对 ANSI C 和 C++ 11 的基本认知，包括宏、模板的基本使用、类继承中虚方法的语义。
- 如果你不具备以上知识，请不要害怕，抱着看风景的心态看看现代代码长什么样就好；
- 如果你只具备以上知识，看到不认识的语言、不认识的语法请不要害怕，抱着看风景的心态看看除了 C/C++ 以外的更加现代化的语言长什么样就好；
- 如果你是更专业的编程语言爱好者甚至研究者，还请随时指出问题。

泛型 | Generics

什么是泛型?

现在语言的泛型长什么样?

它们是如何实现的?

...

什么是泛型?

- 一个常见例子:

  我实现了一个数据结构或算法，想要它能够对许多种类型通用，而不想写多份代码。

- 泛型 + 类型参数 = 自动生成的类型!

```
template <typename T>
void my_sort(T *arr, int n) {
  // TODO: implement sort as if T can compare
}

void foo()
{
    int arr1[] = {3, 2, 1};
    my_sort(arr1, 3);
    // arr1 should be sorted
    string arr2[] = {"xxx"s, "yyy"s, "zzz"s};
    my_sort(arr1, 3);
    // arr2 should be sorted
}
```

在泛型代码中如何操作未知的类型参数?

- 例如前例，C++（20 以前的版本）中，在泛型代码中，对泛型的值的操作是无约束的。
- 右侧代码中：
  如果在实例化模板函数时 T 具备 > 运算符，就可以通过编译、生成正确的代码；
  如果不具备，编译器就会在 > 运算符的调用处进行报错。
- 实际上，C++ 20 引入了 Concept 进行泛型约束。

```
template <typename T>
void my_sort(T *arr, int n) {
  for (int i = 0; i < n; ++i)
    for (int j = i + 1; j < n; ++j)
      if (arr[i] > arr[j])
        std::swap(arr[i], arr[j]);
}
```

泛型约束!

- 以 Rust 为例，重复刚才的泛型版本冒泡排序。
- 通过 T: Ord 对泛型函数的类型参数 T 进行了约束，保证了 arr[i] 和 arr[j] 可以进行比较。

```rust
fn my_sort<T: Ord + Copy>(arr: &mut Vec<T>) {
    for i in 0..arr.len() {
        for j in i..arr.len() {
            let (ai, aj) = (arr[i], arr[j]);
            if ai > aj {
                arr[j] = ai;
                arr[i] = aj;
            }
        }
    }
}
fn main() {
    let mut arr1 = vec![3, 2, 1];
    my_sort(&mut arr1);
    println!("{:#?}", arr1);
    let mut arr2 = vec!["zz", "xx", "yy"];
    my_sort(&mut arr2);
    println!("{:#?}", arr2);
}
```

泛型约束！(cont.)

C#

```
public void MySort<T>(IList<T> arr) where T: IComparable {
    for (int i = 0; i < arr.Count; ++i)
        for (int j = i; j < arr.Count; ++j)
            if (arr[i].CompareTo(arr[j]) > 0)
            {
                var tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
}
```

Scala

```
def mySort[T]
  (arr: collection.mutable.IndexedSeq[T])
  (implicit ev: T ⇒ Ordered[T]) = {
  for (
    i ← 0 until arr.length;
    j ← i until arr.length
  ) {
    if (arr(i) > arr(j)) {
      val tmp = arr(i)
      arr(i) = arr(j)
      arr(j) = tmp
    }
  }
}
```

泛型约束！（C++20 ver.）

- 通过 std::totally_ordered 这一 concept 约束，保证了在使用时 T 是可以比较的。
- 如果不可比较，编译器会在入口处就发现并报错。

```cpp
#include <concepts>

template <std::totally_ordered T>
void my_sort(T *arr, int n) {
    for (int i = 0; i < n; ++i)
        for (int j = i + 1; j < n; ++j)
            if (arr[i] > arr[j])
                std::swap(arr[i], arr[j]);
}
```

如何实现泛型？

单态化

对每组类型参数，实例化不同的代码。

- C++
- Rust
- .NET unmanaged (C# 中 T 为 struct)
- …

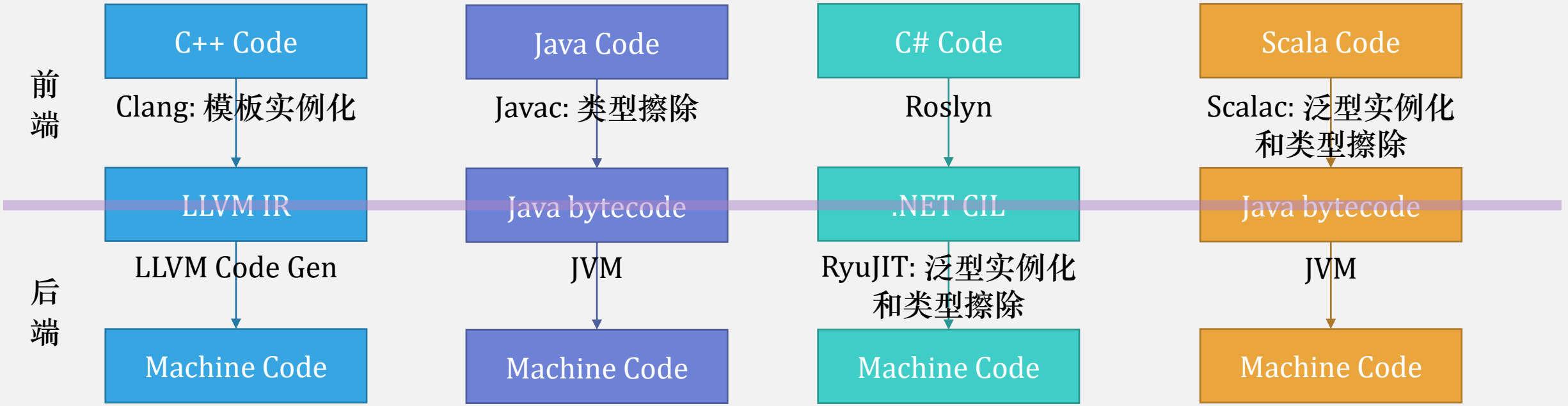
装箱、类型擦除

对所有类型参数，首先将类型参数的值装箱为相同的类型，然后进行类型擦除，用一份机器代码处理各种类型。

调用类型参数上的方法时，配合 interface call 机制进行动态分发。

- Java
- .NET managed (C# 中 T 为 class)
- …

泛型实现的几个例子


前端
后端
C++ Code
Clang: 模板实例化
LLVM IR
LLVM Code Gen
Machine Code
Java Code
Javac: 类型擦除
Java bytecode
JVM
Machine Code
C# Code
Roslyn
.NET CIL
RyuJIT: 泛型实例化和类型擦除
Machine Code
Scala Code
Scalac: 泛型实例化和类型擦除
Java bytecode
JVM
Machine Code

但泛型有时还不够。

Think for frameworks: roads toward metaprogramming

比泛型更强？

- 根据一些与类型定义有关的逻辑自动实现接口、trait 或函数

- Rust 的 #[derive(Debug)]：自动为一个类型生成 debug 打印代码
  - (实际上是更广泛的 custom derive macro 功能的一个实例)

```
#[derive(Debug)]
struct Pet {
    name: String,
}
```

```
struct Pet {
    name: String,
}

impl fmt::Debug for Pet {
    fn fmt(&self, f: &mut
```

比泛型更强？

- 根据一些与类型定义有关的逻辑自动实现接口、trait 或函数

- .NET 和 JVM 语言中的序列化库：自动为一个 class 根据标注生成序列化和反序列化代码
  - (基于反射和运行时中间码生成与加载)

```
[MessagePackObject]
public class Sample1
{
    [Key(0)]
    public int Foo { get; set; }
    [Key(1)]
    public int Bar { get; set; }
}
```

```
private class SomeObject {
  @SerializedName("custom_naming") private final String someField;
  private final String someOtherField;

  public SomeObject(String a, String b) {
    this.someField = a;
    this.someOtherField = b;
  }
}
```

比泛型更强？

- 在编译期，对代码结构（控制流、语句...）进行变换
- Scala async：自动为一段标记了 async-await 的代码进行 delimited CPS 变换，生成基于 Future API 的代码
  - (基于 Scala macro)

```
def combined: Future[Int] = async {
  val future1 = slowCalcFuture
  val future2 = slowCalcFuture
  await(future1) + await(future2)
}
```

```
val future1 = slowCalcFuture
val future2 = slowCalcFuture
def combined: Future[Int] = for {
  r1 <- future1
  r2 <- future2
} yield r1 + r2
```

比泛型更强？

- 在运行时，根据用户输入生成高效率的代码并执行

- Spark SQL：根据用户输入的 SQL 语句，动态生成 Java bytecode 进行高效的大数据处理
  - (基于运行时中间码生成与加载)

```
val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

元编程!

在一个编程语言中，编写程序对自身进行读取、分析、修改或生成的过程，称为元编程。

其中包括的主要方法有：

- Macro/宏：编译时对代码进行变换
  - 根据实现方法分类：匹配、替换类 (如 C)，编译器扩展类 (如 Rust, Scala)
  - 根据操作对象层级分类：Token 级 (如 Rust)，AST 级 (如 Scala, Lisp 系,)
- Staging：运行时生成代码并执行
- Reflection/反射：在有 JIT 运行时的环境 (JVM, .NET ...) 中，可以通过运行时接口对程序进行读取、生成甚至修改
  - （更进一步地）动态类型：在运行时通过构造新的 prototype 生成新的类型 (JS, Py, Lua...)

Lisp macros

- Lisp 的独特之处：程序与数据同相 (都是列表)。
- 基于此，虽然 Lisp 的宏只是最基本的传入参数程序、变换输出替换后的程序，它也具备极为强大的能力。

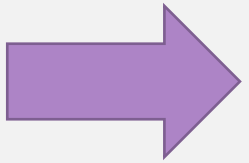
C macros

- 基于 preprocessor，匹配、替换，不具备读取、分析 C 代码的能力。
- 但仍然有一些有趣的技巧，如 X-Macros：

```
// Defining a macro
// with the values of colors.
#define COLORS \
    X(RED)     \
    X(BLACK)   \
    X(WHITE)   \
    X(BLUE)
```

```
// Creating an enum of colors
// by macro expansion.
enum colors {
    #define X(value) value,
        COLORS
    #undef X
};

// A utility that takes the enum value
// and returns corresponding string value
char* toString(enum colors value)
{
    switch (value) {
        #define X(color) \
            case color:  \
                return #color;
                COLORS
        #undef X
    }
}
```



```
// Creating an enum of colors.
enum colors {
    RED,
    BLACK,
    WHITE,
    BLUE
};

/*A utility that takes the enum value and returns
corresponding string value*/
char* toString(enum colors value)
{
    switch (value) {
    case RED:
        return "RED";
    case BLACK:
        return "BLACK";
    case WHITE:
        return "WHITE";
    case BLUE:
        return "BLUE";
    }
}
```

C++ Template & Constexpr

- 倒数第二脏的元编程（如果 Reflection 算元编程的话）
- 在编译期，template 和 constexpr 组成了一个隐含的语言子集，它被编译器解释执行，用于生成一个没有它们的 C++ 程序，然后进行后续的编译。
- 无法以任何形式解析源代码；只能在编译期根据模板参数进行奇妙的代码生成。

```
template<int N>
struct A {
    constexpr A() : arr() {
        for (auto i = 0; i != N; ++i)
            arr[i] = i;
    }
    int arr[N];
};

int main() {
    constexpr auto a = A<4>();
    for (auto x : a.arr)
        std::cout << x << '\n';
}
```

Java/C# Reflection

最脏的元编程（如果还算是元编程的话）

- 通过 JVM/.NET 提供的接口，获取程序信息
  - Java: java.lang.reflect.{Class, Method, ...}
  - C#: System.Reflection.{Type, TypeInfo, Method, MethodInfo, ...}
- 可以解析成员类型、方法实现...
- 配合其它功能（JVM 的 asm，.NET 的 Emit）可以进行代码生成和加载。

```
ILGenerator ILout = myMthdBld.GetILGenerator();
int numParams = mthdParams.Length;
for (byte x=0; x < numParams; x++)
    ILout.Emit(OpCodes.Ldarg_S, x);
if (numParams > 1)
    for (int y=0; y<(numParams-1); y++)
        switch (mthdAction)
        {
            case "A": ILout.Emit(OpCodes.Add); break;
            case "M": ILout.Emit(OpCodes.Mul); break;
            default: ILout.Emit(OpCodes.Add); break;
        }
ILout.Emit(OpCodes.Ret);
```

Rust declarative macros

- 类似 C macros，匹配、替换模式，但对传入的内容有所限定，更加安全。

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

```
let v: Vec<u32> = vec![1, 2, 3];
```

Rust procedural macros

- Rust 的 proc macros 更为强大。
- Proc macro 被声明为一个在其它模块中的 Rust 函数。
- 在 proc macro 被调用时，调用处的代码内容被以 TokenStream 的形式输入 proc macro 的实现函数中，在其中进行相应的处理后，实现返回新的 TokenStream，用以替换原本的代码内容。

```
#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

```
#[route(GET, "/")]
fn index() {
```

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Scala 3.0 metaprogramming

Scala 3.0 基于 inline 关键字、quote 和 splice 操作符重新实现了 metaprogramming 系统。

- Inline 是将函数或函数参数按原样替换进被使用的位置；
- Quote (`'{ ...some code... }`) 将被包含的代码转换为有类型的 Syntax Tree；
- Splice (`${ ...some code... }`) 将其中计算出的 Syntax Tree 重新展开为代码。

以上三个功能组合起来便可以优雅地实现 macro。

此外：

- staging.run 和 staging.withQuoteContext 分别为运行时生成并执行 Scala 程序和其它语言程序提供了支持；
- quoted patterns 和 tasty._ 为解析 syntax tree 提供了支持。

将 C++ Template 视作编译器执行的 Staging?

C++ Template

```cpp
template <int n>
double power(double x) {
  if constexpr (n < 0)
    return 1 / power<-n>(x);
  else if constexpr (n == 0)
    return 1.0;
  else if constexpr (n % 2 == 1) {
    auto x2 = x * x;
    return power<n / 2>(x2) * x;
  }
  else {
    auto x2 = x * x;
    return power<n / 2>(x2);
  }
}
```

Scala 3.0 Staging

```scala
def power (using QuoteContext) (n: Int)
  : Expr[Double => Double] =
  '{ x => ${
    if n < 0 then
      '{1 / ${power(-n)}(x)}
    else if n == 0 then
      '{1.0}
    else if n % 2 == 1 then '{
      val x2 = x * x
      ${power(n / 2)}(x2) * x
    }
    else '{
      val x2 = x * x
      ${power(n / 2)}(x2)
    }
  }}
```