



# IR自动生成 实验讲解

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年10月16日

# CONTENT

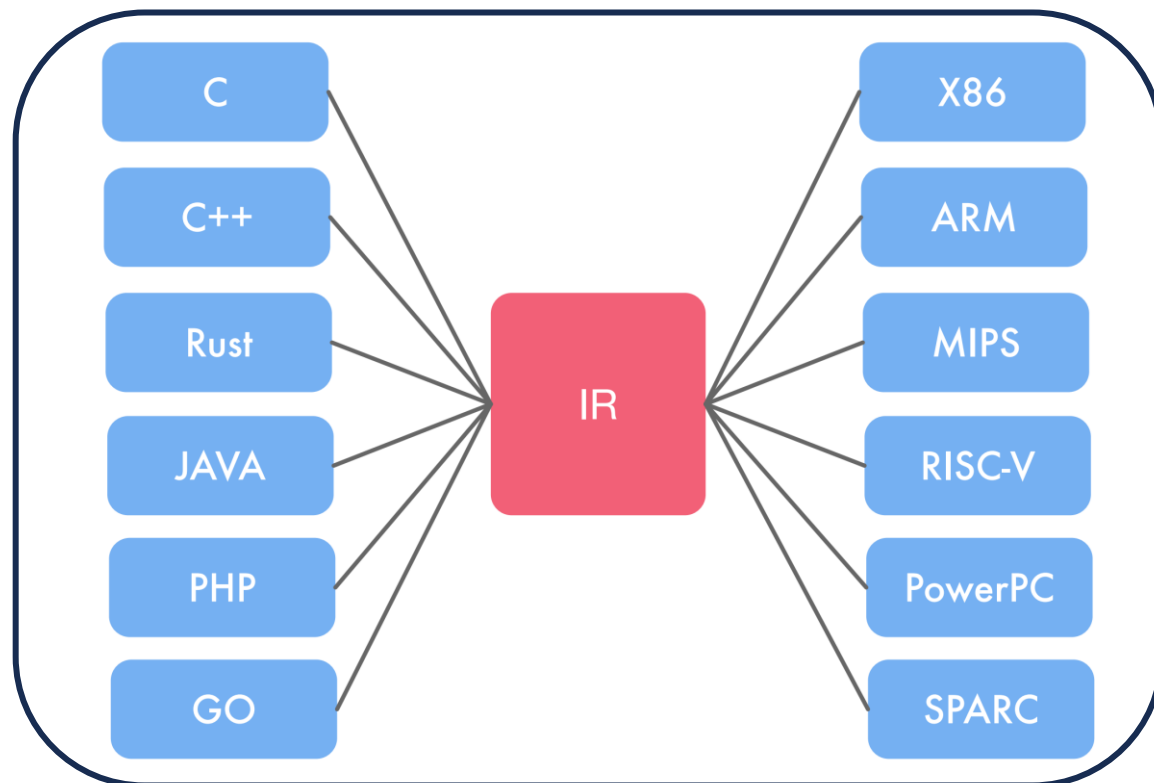


- LLVM 简介
- Light IR C++ 库
- IR 自动化生成框架

# 传统编译器 - Compiler Collection



## • 传统编译器 (GCC) 三段式架构



Compiler Collection



# 传统编译器架构面临的问题



- 推陈出新的语言、不断涌现新的目标平台

分类	名称	版本	扩展	初始年份
CISC	x86	16, 32, 64 (16→32→64)	x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSSE3, SSE4, BMI, AVX, AES, FMA, XOP, F16C	1978
RISC	MIPS	32	<a href="#">MDMX</a> , <a href="#">MIPS-3D</a>	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64, SSE3, AVX	2014

- 问题：传统编译器饱受**分层**和**抽象漏洞困扰**，添加新语言与目标平台的支持时**源代码重用**的难度较大

# LLVM 出现



- LLVM 制定了LLVM IR，将编译器需要的功能**包装成库**，解决了编译器**代码重用**的问题



Compiler Collection



Libraries Collection

- LLVM 是一个编译器
- LLVM 是一个编译框架
- LLVM 是一系列编译工具
- LLVM 是一个编译工具链
- LLVM 是一个开源C++的实现
- LLVM 项目发展为一个巨大的编译器相关的工具集合

# LLVM 在就业中的应用



## • 很多互联网公司急需懂得 LLVM 实践人才

### Program Language Engineer - LLVM Direction

Byte Dance 4.1

上海市 [+2地区](#)

languages... LLVM ecosystem... techniques, experience in LLVM, GCC, Go related compiler...

19 天前发布 · 更多.....

### 平头哥-编译器技术专家-AI软件-上海

阿里巴巴集团

上海市

对机器学习算法/深度学习有一定了解尤佳; 4. 有GCC、LLVM和Open64等开源编译器相关开发经验尤佳; 5. 有CUDA... project... GCC/LLVM/Open64...

30 多天前发布 · 更多.....

### LLVM Senior Compiler Engineer

Byte Dance 4.1

上海市 [+2地区](#)

1. Responsible LLVM new back-end... preferred: (1) Familiar with LLVM compiler development experience...

30 天前发布 · 更多.....

### SMTS Software Development Eng.

Xilinx 4.0

北京市

years of GCC/LLVM/Open64 industry... frameworks Experience of TVM/MLIR/LLVM/GCC is preferred Experience...

30 多天前发布 · 更多.....

Indeed 招聘信息截图

# Light IR C++库

**编译原理课程组**

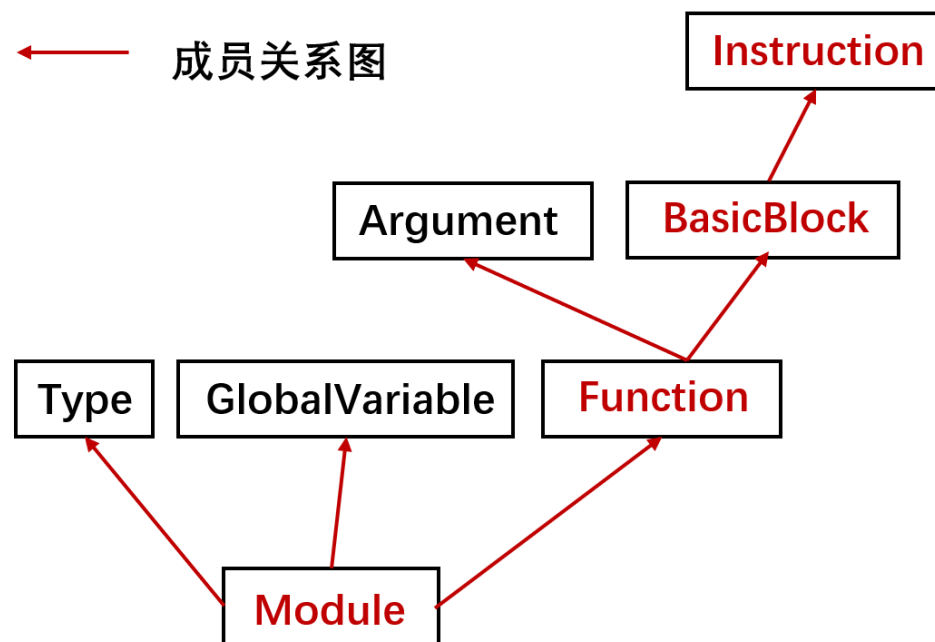
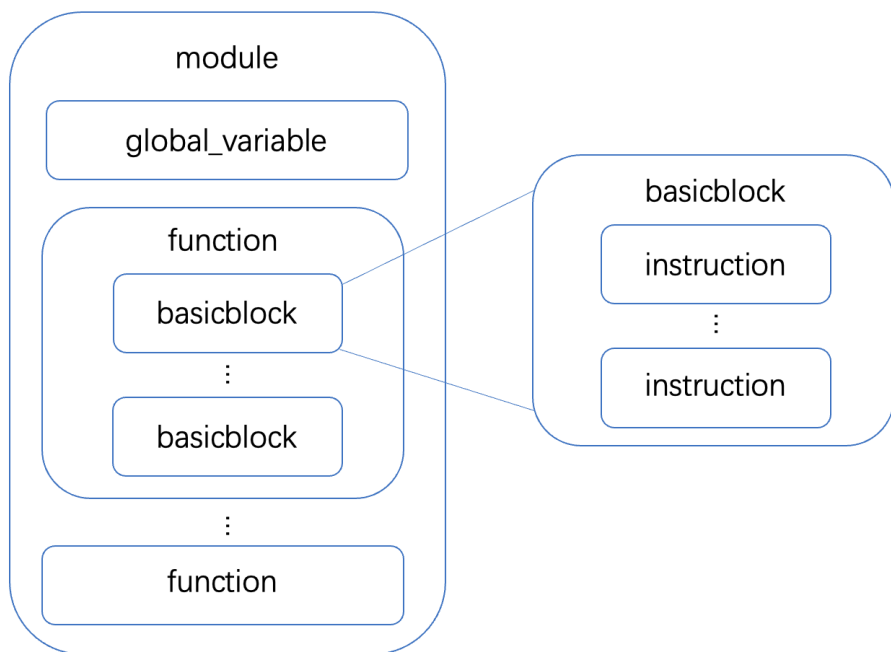
**中国科学技术大学**



# Light IR 结构与 C++ 类总览

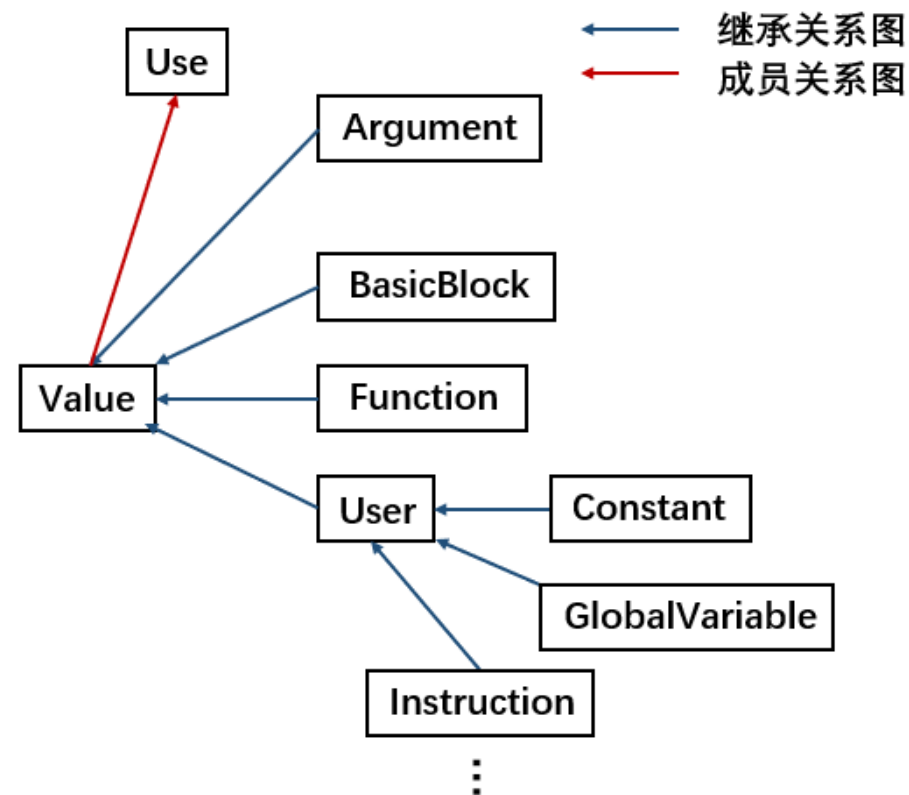


- Light IR 存在层次化结构
- Light IR C++ 库存在对应层次化类的设计

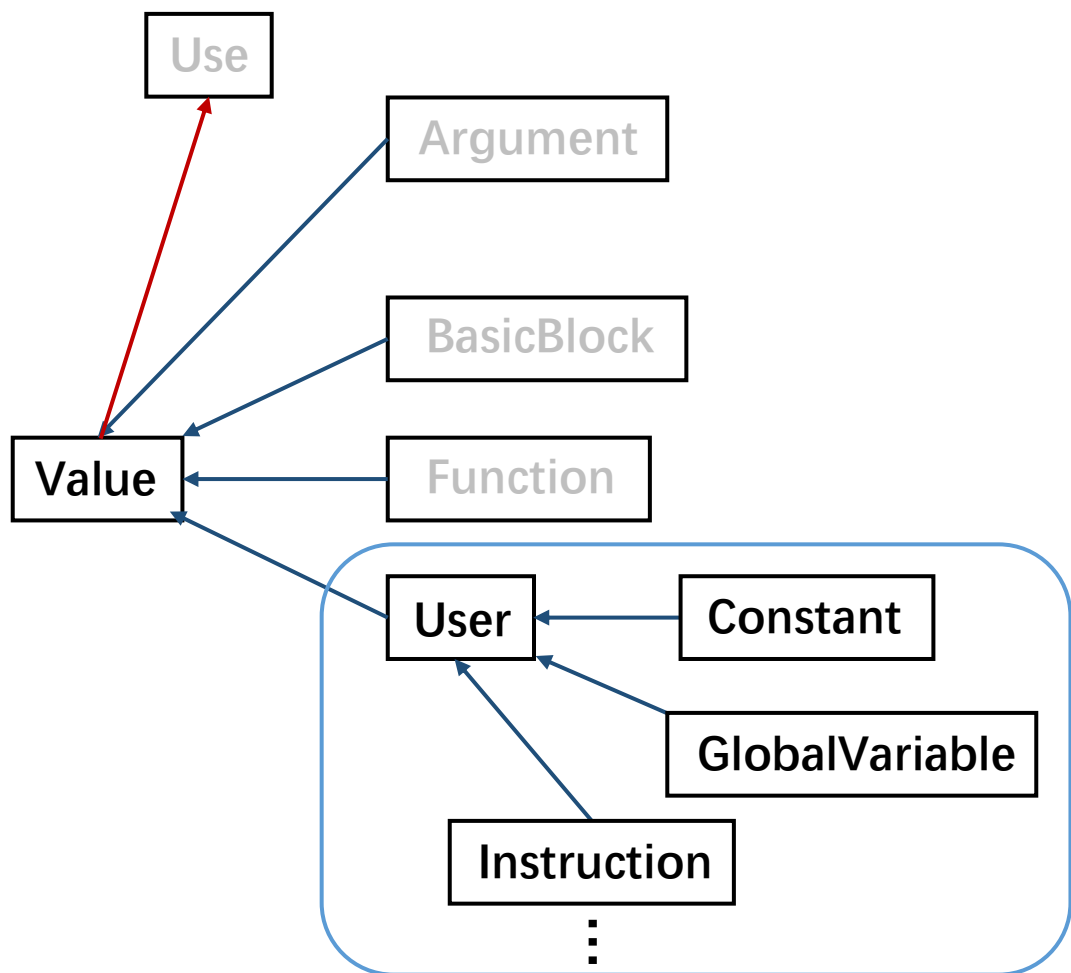


## • Value

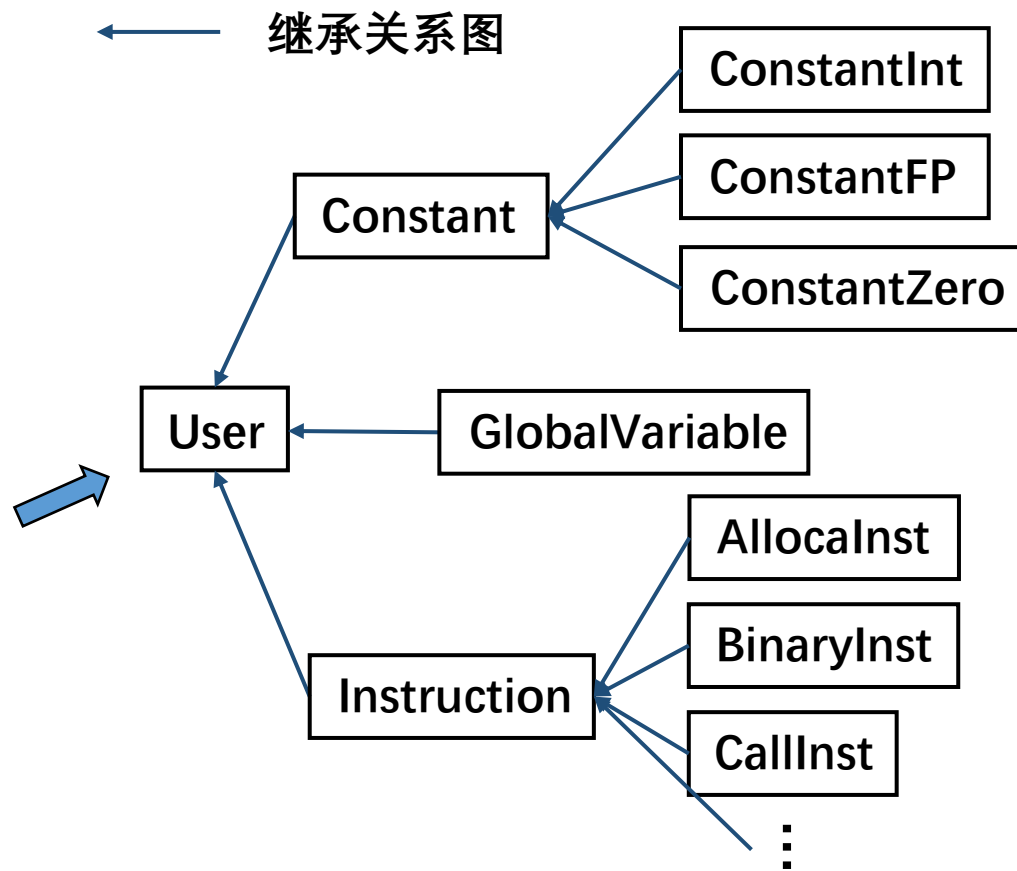
- **概念**: Value 类代表一个可用于指令操作数的带类型的数据, 包含众多子类
- **子类**: Instruction 也是其子类之一, 表示指令在创建后可以作为另一条指令的操作数
- **注**: Value 成员 use list 是 Use 类的列表, 每个 Use 类记录了该 Value 的一次被使用的情况



# Light IR 数据基类

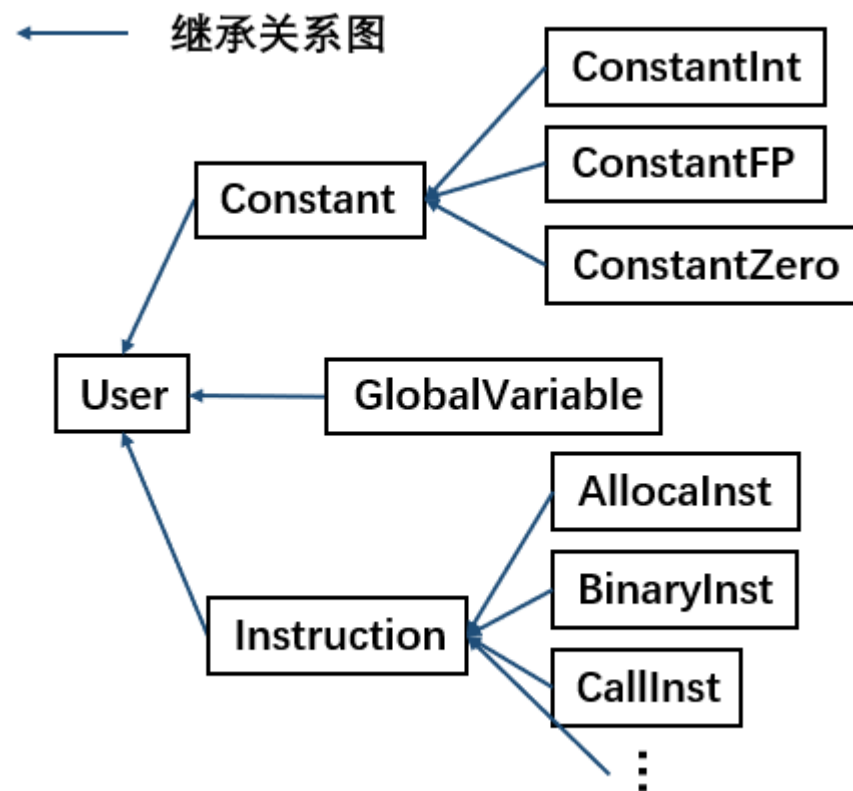


继承关系图



## • User

- **概念**: User 作为 Value 的子类, 含义是使用者
- **子类**: Instruction 也是其子类之一
- **注**: User 类成员 operands 是 Value 类的列表, 表示该使用者使用的操作数列表



- **Light IR 类型系统**

- 包含基本类型与组合类型，Type 类是所有类型的基类

- **Type**

- 子类:

- IntegerType, FloatType 对应表示 Light IR 中的 i1, i32, float 基本类型
- ArrayType, PointerType, FunctionType 对应表示组合类型: 数组类型, 指针类型, 函数类型

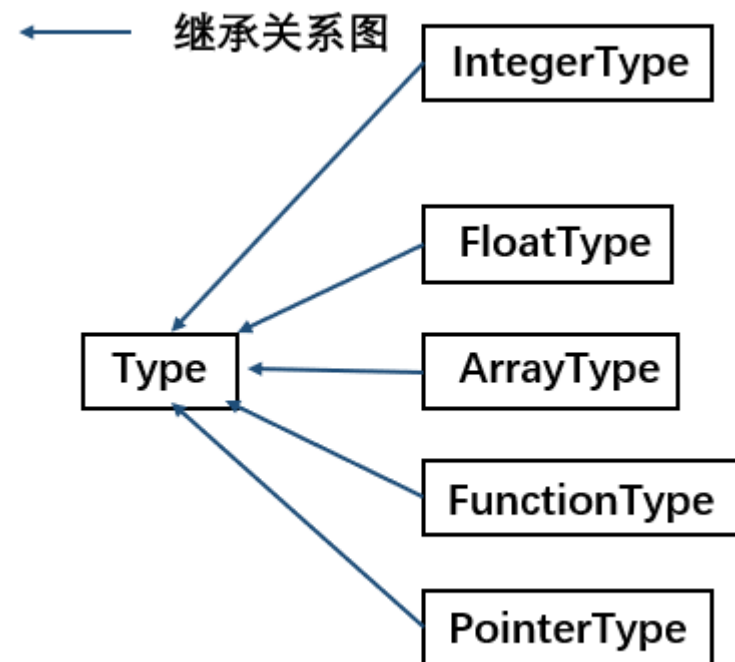
- API:

// 获取 i1 基本类型

auto int1\_type = module->get\_int1\_type();

// 获取 [2 x i32] 数组类型

auto array\_type = ArrayType::get(Int32Type, 2);



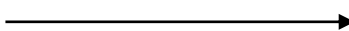
# 使用 Light IR C++ 库生成 IR



- 例如如下 Cminus-f 代码翻译为LightIR
- 如何用 Light IR C++库生成相应的IR?

```
int main(void) {  
    int x;  
    x = 72;  
    return x;  
}
```

Cminus-f代码



```
; ModuleID = 'cminus'  
define i32 @main() #0 {  
entry :  
    %1 = alloca i32  
    store i32 72, i32* %1  
    %2 = load i32, i32* %1  
    ret i32 %2  
}
```

翻译成 LightIR 代码

# 使用 Light IR C++ 库生成 IR



```
//头文件
#include "BasicBlock.hpp"
#include "Constant.hpp"
#include "Function.hpp"
#include "IRBuilder.hpp"
#include "Module.hpp"
#include "Type.hpp"


#include <iostream>
#include <memory>

int main() {
    ...
    return 0;
}
```

# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
```



```
; ModuleID = 'cminus'
define i32 @main() #0 {
}
```

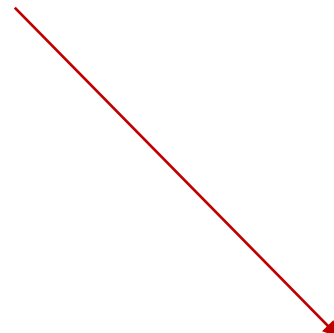
IR 情况



# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
```



```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry :
}
```

IR 情况

# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为“entry”基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = module->get_int32_type();
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
}
```

IR 情况

# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
// 创建main函数内的基本块"entry"
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为"entry"基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
}
```

IR 情况

# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
// 创建main函数内的基本块"entry"
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为"entry"基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
// 创建 load 指令, 将 x 内存值取出来
auto xLoad = builder->create_load(xAlloca);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
    %2 = load i32, i32* %1
}
```

IR 情况

# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
// 创建main函数内的基本块"entry"
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为"entry"基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
// 创建 load 指令, 将 x 内存值取出来
auto xLoad = builder->create_load(xAlloca);
// 创建 ret 指令, 将 x 取出的值返回
builder->create_ret(xLoad);
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
    %2 = load i32, i32* %1
    ret i32 %2
}
```

IR 情况

# 使用 Light IR C++ 库生成 IR



```
// 实例化module
auto module = new Module("cminus code");
// 实例化IRbuilder
auto builder = new IRBuilder(nullptr, module);
// 创建main函数
auto mainFun = Function::create(FunctionType::get(Int32Type, {}), "main", module);
// 创建main函数内的基本块“entry”
bb = BasicBlock::create(module, "entry", mainFun);
// 将IRBuilder插入指令位置设置为“entry”基本块
builder->set_insert_point(bb);
// 从module中获取 i32 类型
Type *Int32Type = Type::get_int32_type(module);
// 为变量 x 分配栈上空间
auto xAlloca = builder->create_alloca(Int32Type);
// 创建 store 指令, 将72常数存到 x 分配空间里
builder->create_store(ConstantInt::get(72, module), xAlloca);
// 创建 load 指令, 将 x 内存值取出来
auto xLoad = builder->create_load(xAlloca);
// 创建 ret 指令, 将 x 取出的值返回
builder->create_ret(xLoad);
std::cout << module->print(); //打印输出IR
```

```
; ModuleID = 'cminus'
define i32 @main() #0 {
entry:
    %1 = alloca i32
    store i32 72, i32* %1
    %2 = load i32, i32* %1
    ret i32 %2
}
```

IR 情况



# 一起努力 打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年10月16日