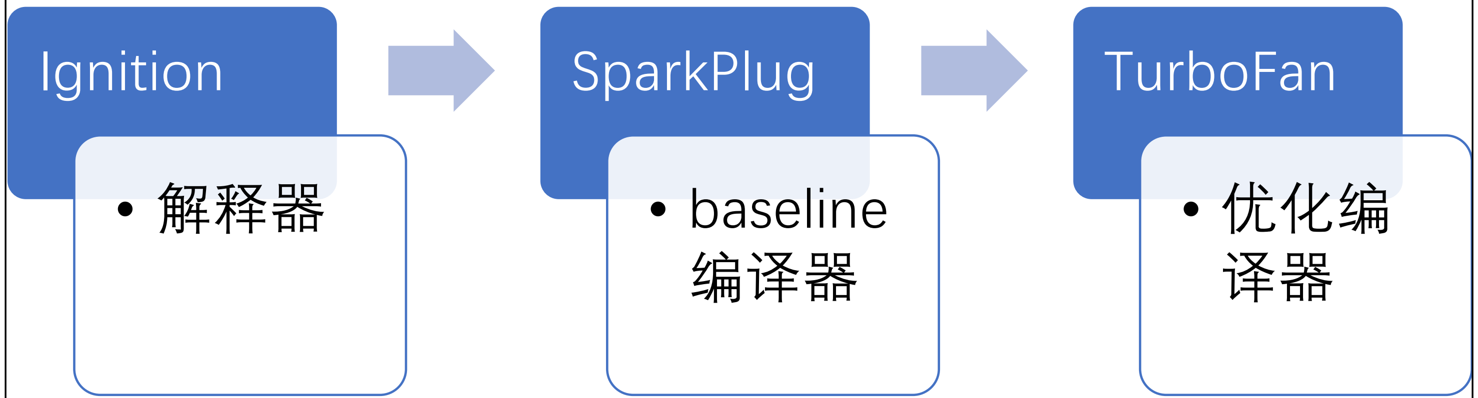
ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

V8 Turbofan
IR 之Node数据结构

智能软件研究中心 邱吉
qiuji@iscas.ac.cn

2021/12/10

课程转入新的篇章-继续深入V8的执行引擎



V8执行引擎架构图

之前的课程讲解了解释器的部分

《V8中helloworld的执行过程》已完结

Ignition
- 解释器

→

SparkPlug
- baseline编译器

→

TurboFan
- 优化编译器

https://www.bilibili.com/video/BV1hp4y1t7Mx?p=8
https://www.bilibili.com/video/BV1hp4y1t7Mx?p=10
https://www.bilibili.com/video/BV1hp4y1t7Mx?p=11
https://www.bilibili.com/video/BV1hp4y1t7Mx?p=13

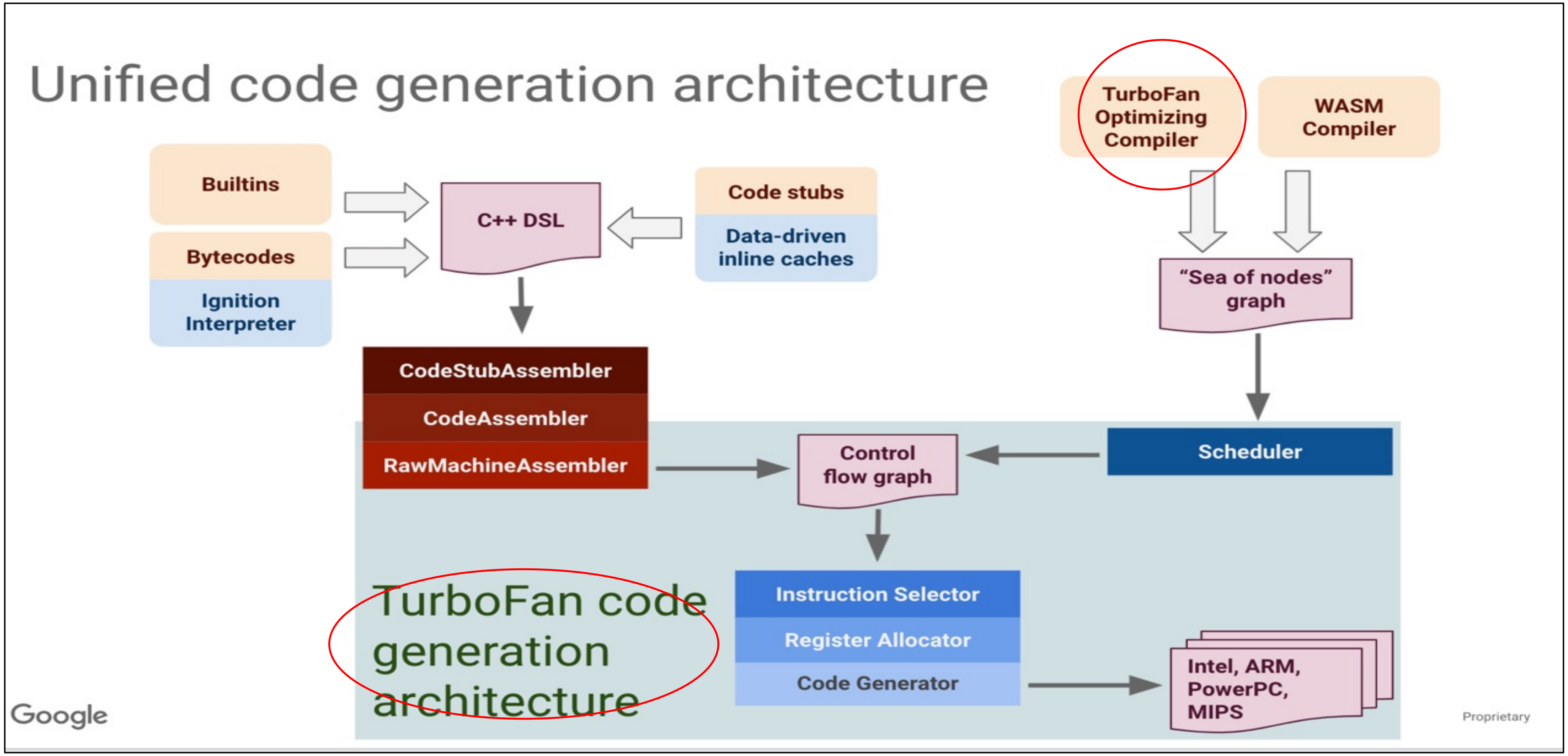
课程转入新的篇章-TurboFan

Ignition
- 解释器

SparkPlug
- baseline编译器

TurboFan
- 优化编译器

TurboFan是V8执行引擎的核心组成部分



https://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition

今天的内容： TurboFan IR之Node数据结构

- 基本概念
- Node的内存布局
- Node的构建过程
- 如何进行def-use和use-def的遍历

今天的内容： TurboFan IR之Node数据结构

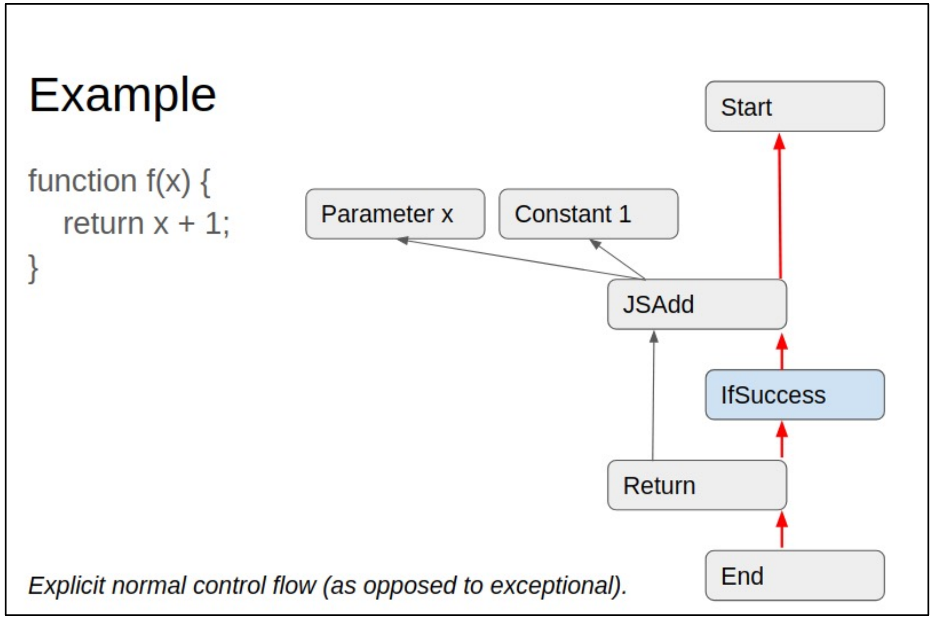
- 基本概念
- Node的内存布局
- Node的构建过程
- 如何进行def-use和use-def的遍历

基本概念

- TurboFan 的 Sea-of-Node IR

> - Graph based IR
>   - Nodes for operations.
>   - Edges for value flow, *control flow* and dependencies.
>   - No distinction between basic blocks and statements.
>   - Single-static assignment.

-- “Turbofan IR，Jaroslav Sevcik”



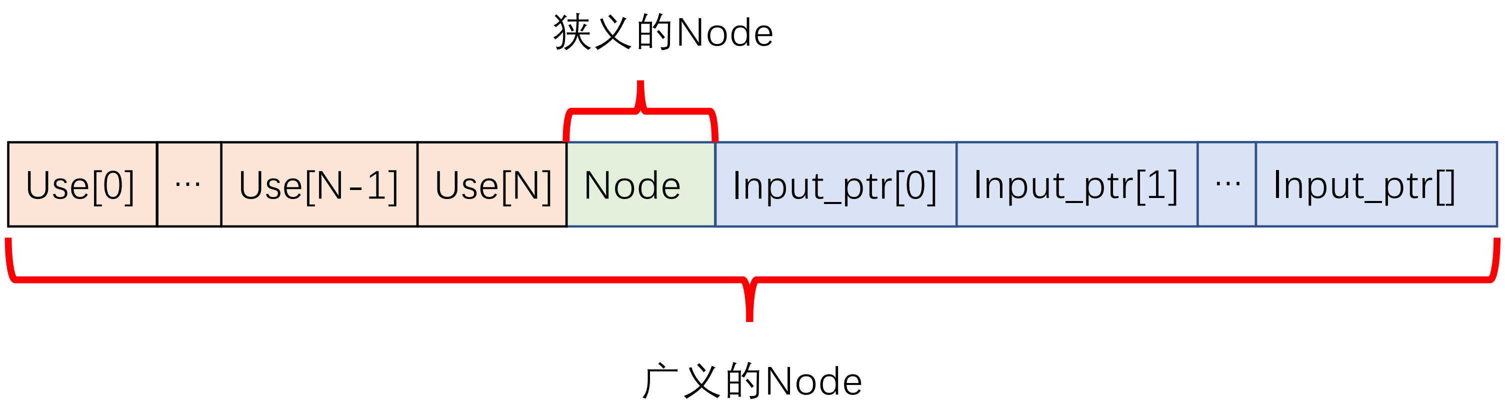
- 一个简单的例子：右图->
  - 灰色边：反向后的数据流
  - 红色边：反向后的控制流
- **V8中confusing的叫法：**
- **Use/To：箭头指向的Node，输入**
- **From：箭头出发的Node，消费者**

https://docs.google.com/presentation/d/1Z9iIHojKDrXvZ27gRX51UxHD-bKf1QcPzSijntpMJBM/edit#slide=id.p

今天的内容： TurboFan IR之Node数据结构

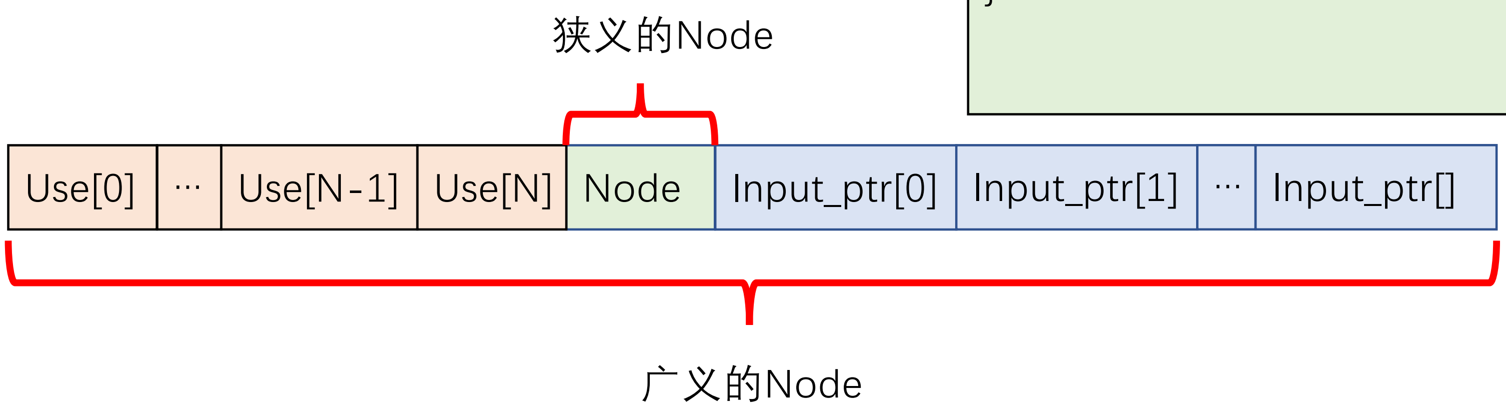
- 基本概念
- Node的内存布局
- Node的构建过程
- 如何进行def-use和use-def的遍历

Node的内存布局（inline case)



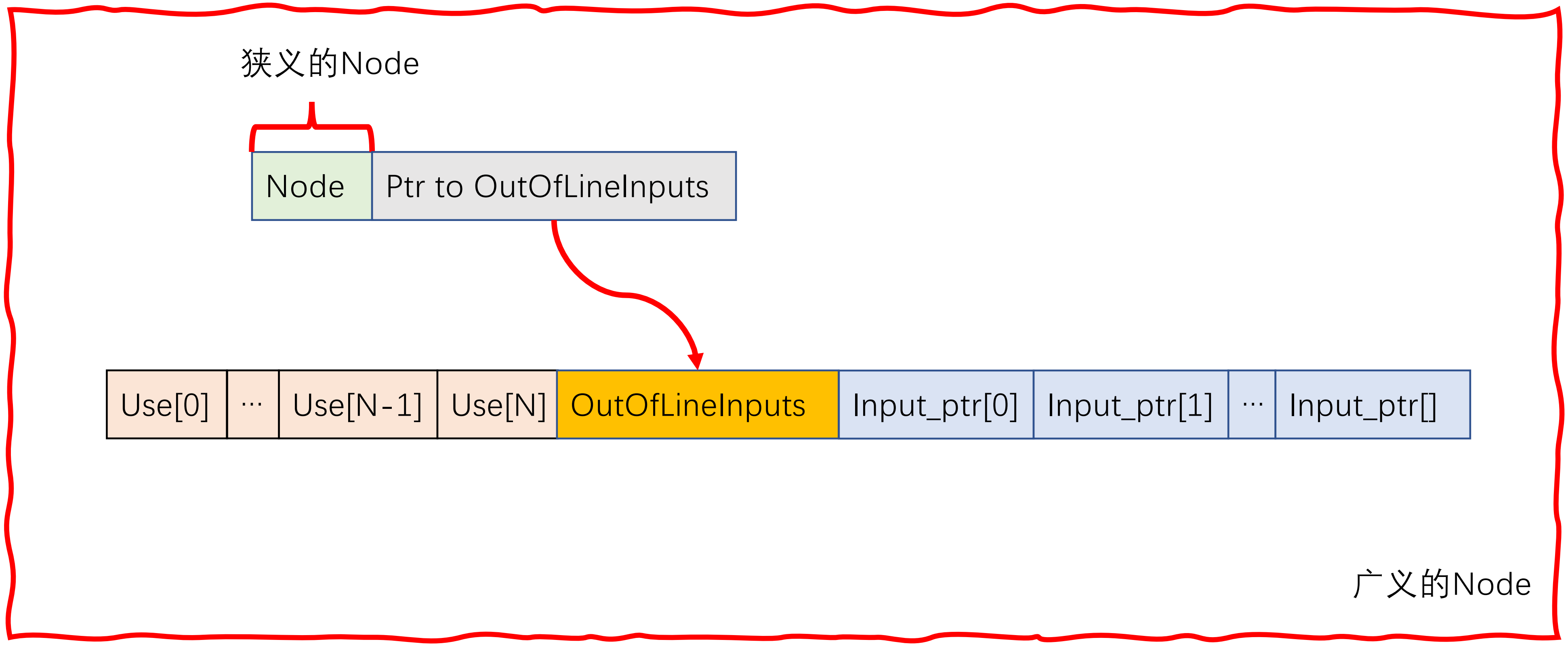
Node的内存布局（inline case)

Node {
opcode：操作码
type_： 类型
mark_：记号
bit_field_：编码nodeid/InlineCount/InlineCapactity
first_use_: 本Node的使用者链表
}



Use {
next：*
prev：*
bit_field_：编码InputIdex/InlinedField
}

Node的内存布局（out of line case)



Node的内存布局（out of line case)

Node {
opcode：操作码
type_： 类型
mark_：记号
bit_field_：编码nodeid/InlineCount/InlineCapactity
first_use_: 本Node的使用者链表
}

狭义的Node

Node	Ptr to OutOfLineInputs

Use[0]	…	Use[N-1]	Use[N]	OutOfLineInputs	Input_ptr[0]	Input_ptr[1]	…	Input_ptr[]

Use {
next：*
prev：*
bit_field_：编码InputIdex/InlinedField
}

struct OutOfLineInputs {
ZoneNodePtr node_;
int count_;
int capacity_;

广义的Node

今天的内容： TurboFan IR之Node数据结构

- 基本概念
- Node的内存布局
- Node的构建过程
- 如何进行def-use和use-def的遍历

Node的构建过程-New
src/compiler/node.h & node.cc

```
Node* Node::New(Zone* zone, NodeId id, const Operator* op, int input_count,
                Node* const* inputs, bool has_extensible_inputs) {
  return NewImpl(zone, id, op, input_count, inputs, has_extensible_inputs);
}
```

Node的构建过程-NewImpl

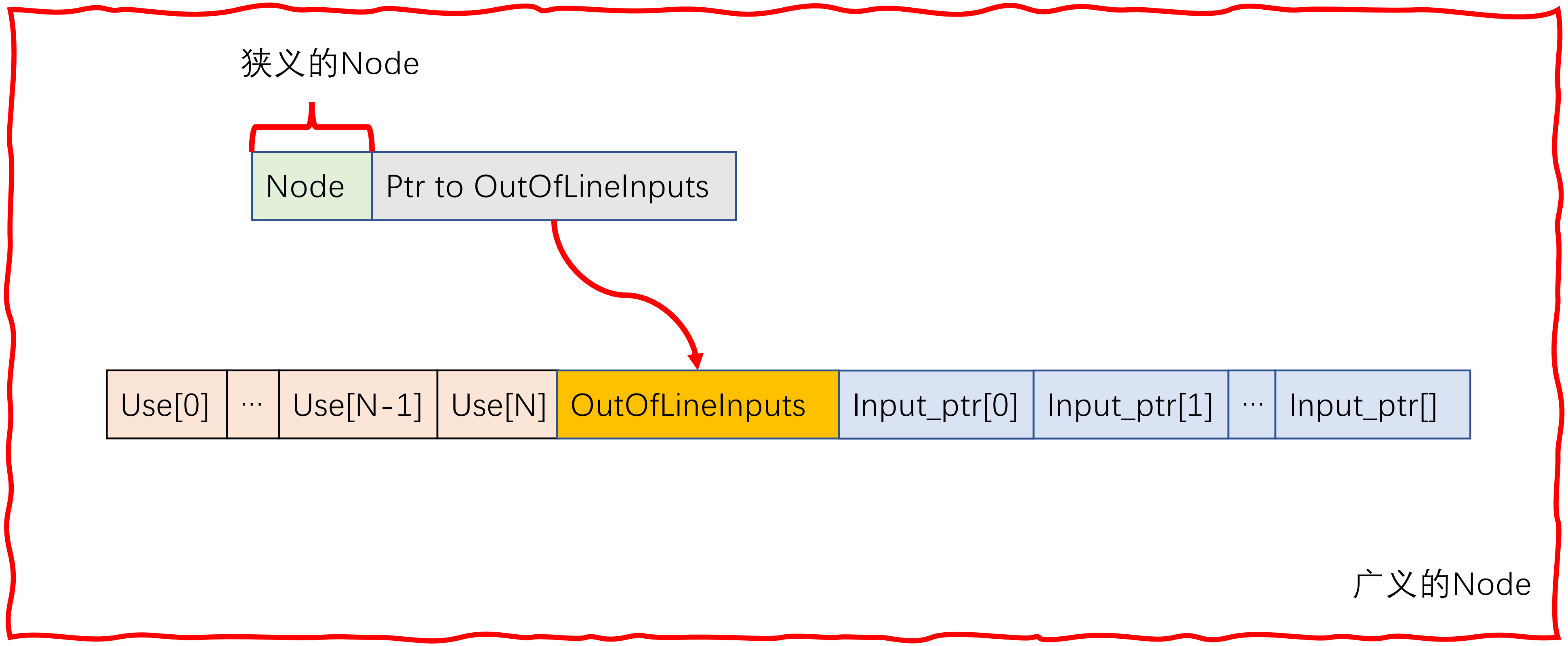
```
Node* Node::NewImpl(Zone* zone, NodeId id, const Operator* op, int input_count,
                  NodePtrT const* inputs, bool has_extensible_inputs) {
 // Node uses compressed pointers, so zone must support pointer compression.
 DCHECK_IMPLIES(kCompressGraphZone, zone->supports_compression());
 DCHECK_GE(input_count, 0);
 ZoneNodePtr* input_ptr;
 Use* use_ptr;
 Node* node;
 bool is_inline;
 // Verify that none of the inputs are {nullptr}.
 for (int i = 0; i < input_count; i++) {
   if (inputs[i] == nullptr) {
     FATAL("Node::New() Error: #%d:%s[%d] is nullptr", static_cast<int>(id),
           op->mnemonic(), i);
   }
 }
…
```

Node的构建过程-NewImpl-2-OutOfLine Case

中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

```
if (input_count > kMaxInlineCapacity) {
  // Allocate out-of-line inputs.
  int capacity =
      has_extensible_inputs ? input_count + kMaxInlineCapacity : input_count;
  OutOfLineInputs* outline = OutOfLineInputs::New(zone, capacity);

  // Allocate node, with space for OutOfLineInputs pointer.
  void* node_buffer = zone->Allocate<NodeWithOutOfLineInputs>(
      sizeof(Node) + sizeof(ZoneOutOfLineInputsPtr));
  node = new (node_buffer) Node(id, op, kOutlineMarker, 0);
  node->set_outline_inputs(outline);

  outline->node_ = node;
  outline->count_ = input_count;

  input_ptr = outline->inputs();
  use_ptr = reinterpret_cast<Use*>(outline);
  is_inline = false;
} else {
```

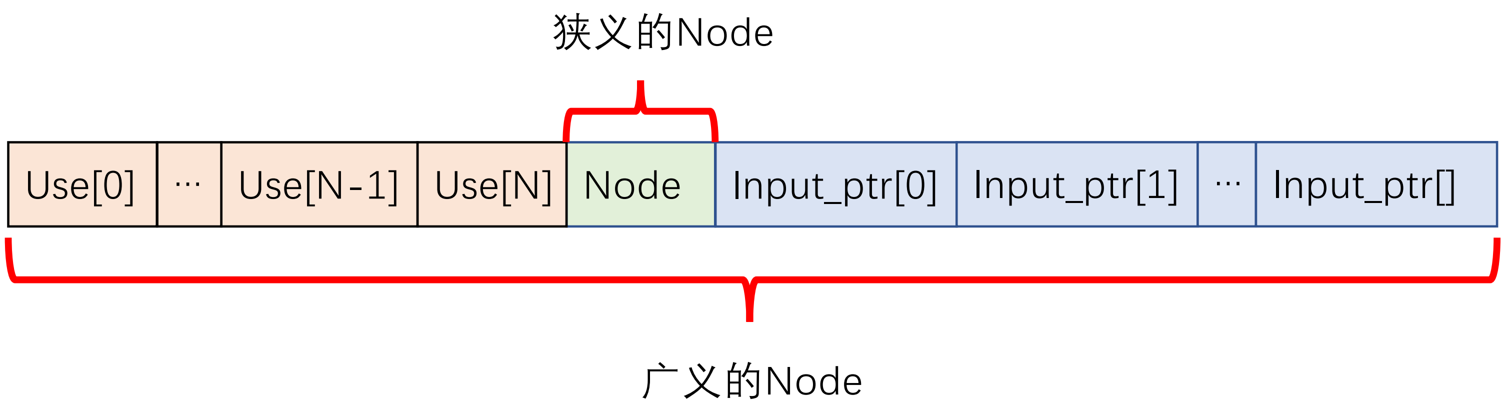
Node的内存布局（out of line case)



Node的构建过程-NewImpl-3-Inline Case


```
} else {
  // Allocate node with inline inputs. Capacity must be at least 1 so that
  // an OutOfLineInputs pointer can be stored when inputs are added later.
  int capacity = std::max(1, input_count);
  if (has_extensible_inputs) {
    const int max = kMaxInlineCapacity;
    capacity = std::min(input_count + 3, max);
  }

  size_t size = sizeof(Node) + capacity * (sizeof(ZoneNodePtr) + sizeof(Use));
  intptr_t raw_buffer =
      reinterpret_cast<intptr_t>(zone->Allocate<NodeWithInLineInputs>(size));
  void* node_buffer =
      reinterpret_cast<void*>(raw_buffer + capacity * sizeof(Use));

  node = new (node_buffer) Node(id, op, input_count, capacity);
  input_ptr = node->inline_inputs();
  use_ptr = reinterpret_cast<Use*>(node);
  is_inline = true;
}
```

Node的内存布局（inline case)



ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

Node的构建过程-NewImpl-4-Use chain的建立

```
  // Initialize the input pointers and the uses.
  CHECK_IMPLIES(input_count > 0,
                Use::InputIndexField::is_valid(input_count - 1));
  for (int current = 0; current < input_count; ++current) {
    Node* to = *inputs++;
    input_ptr[current] = to;
    Use* use = use_ptr - 1 - current;
    use->bit_field_ = Use::InputIndexField::encode(current) |
                      Use::InlineField::encode(is_inline);
    to->AppendUse(use);
  }
  node->Verify();
  return node;
}
```

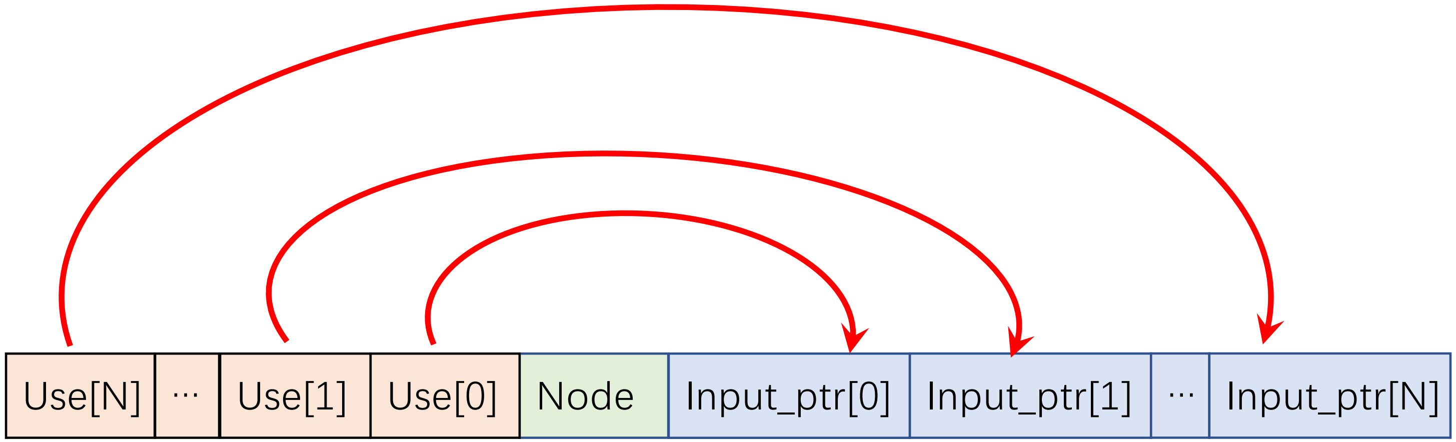
Node的构建过程-狭义Node的New

```
Node::Node(NodeId id, const Operator* op, int inline_count, int inline_capacity)
   : op_(op),
     mark_(0),
     bit_field_(IdField::encode(id) | InlineCountField::encode(inline_count) |
                InlineCapacityField::encode(inline_capacity)),
     first_use_(nullptr) {
  // Check that the id didn't overflow.
  STATIC_ASSERT(IdField::kMax < std::numeric_limits<NodeId>::max());
  CHECK(IdField::is_valid(id));

  // Inputs must either be out of line or within the inline capacity.
  DCHECK(inline_count == kOutlineMarker || inline_count <= inline_capacity);
  DCHECK_LE(inline_capacity, kMaxInlineCapacity);
}
```

今天的内容： TurboFan IR之Node数据结构

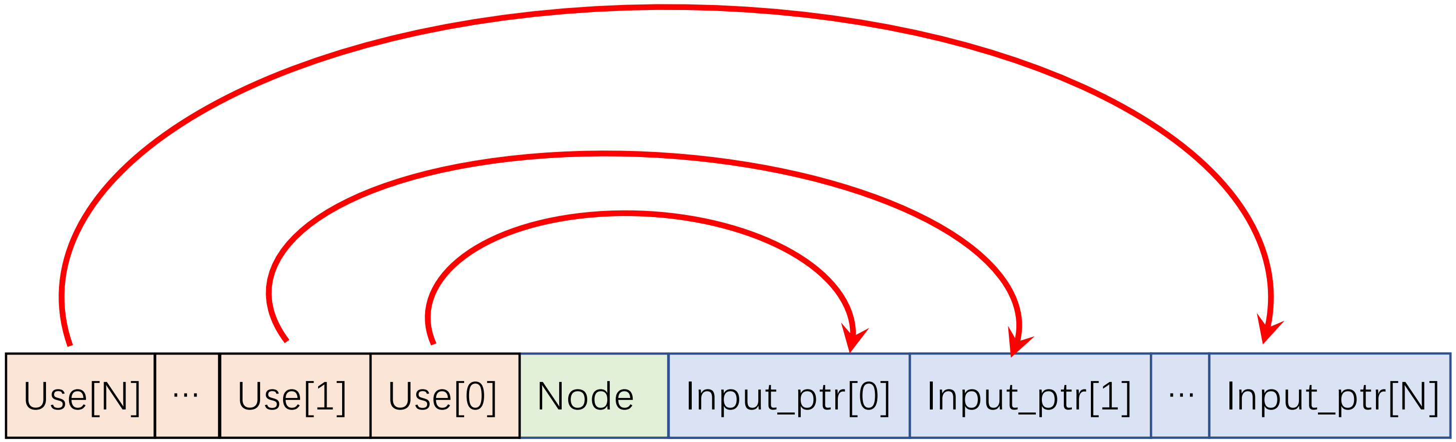
- 基本概念
- Node的内存布局
- Node的构建过程
- 如何实现SSA，如何进行def-use和use-def的遍历

Inline Case中Use和Input的对应关系 Mirrored



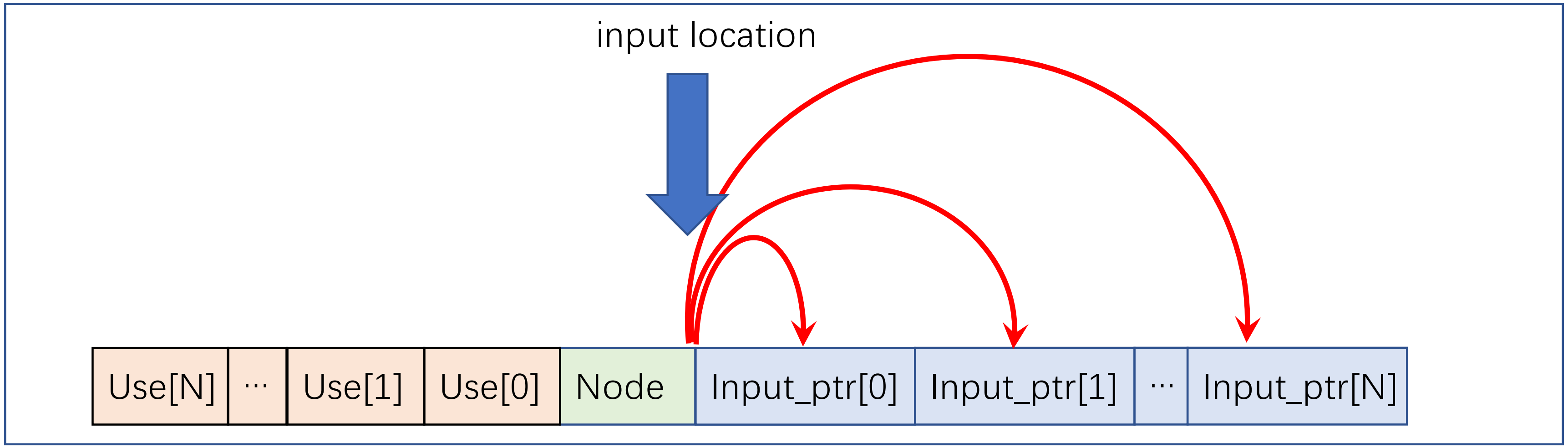
Input是有序的

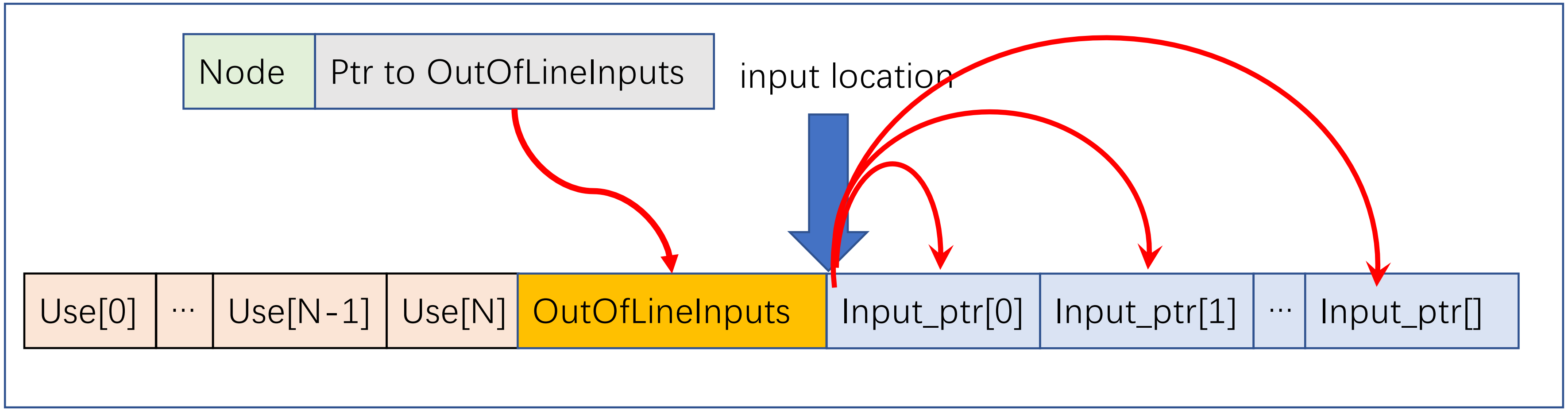
SON图：Input有序，且指针是天然的SSA实现



1. Input是有序的
2. 一个Node只能表示一个操作，操作的输出（可能有多个）就是Node本身
3. 由于Node的指针在编译器的进程空间中是唯一的，因此，SON图天然而然地是SSA形式的IR

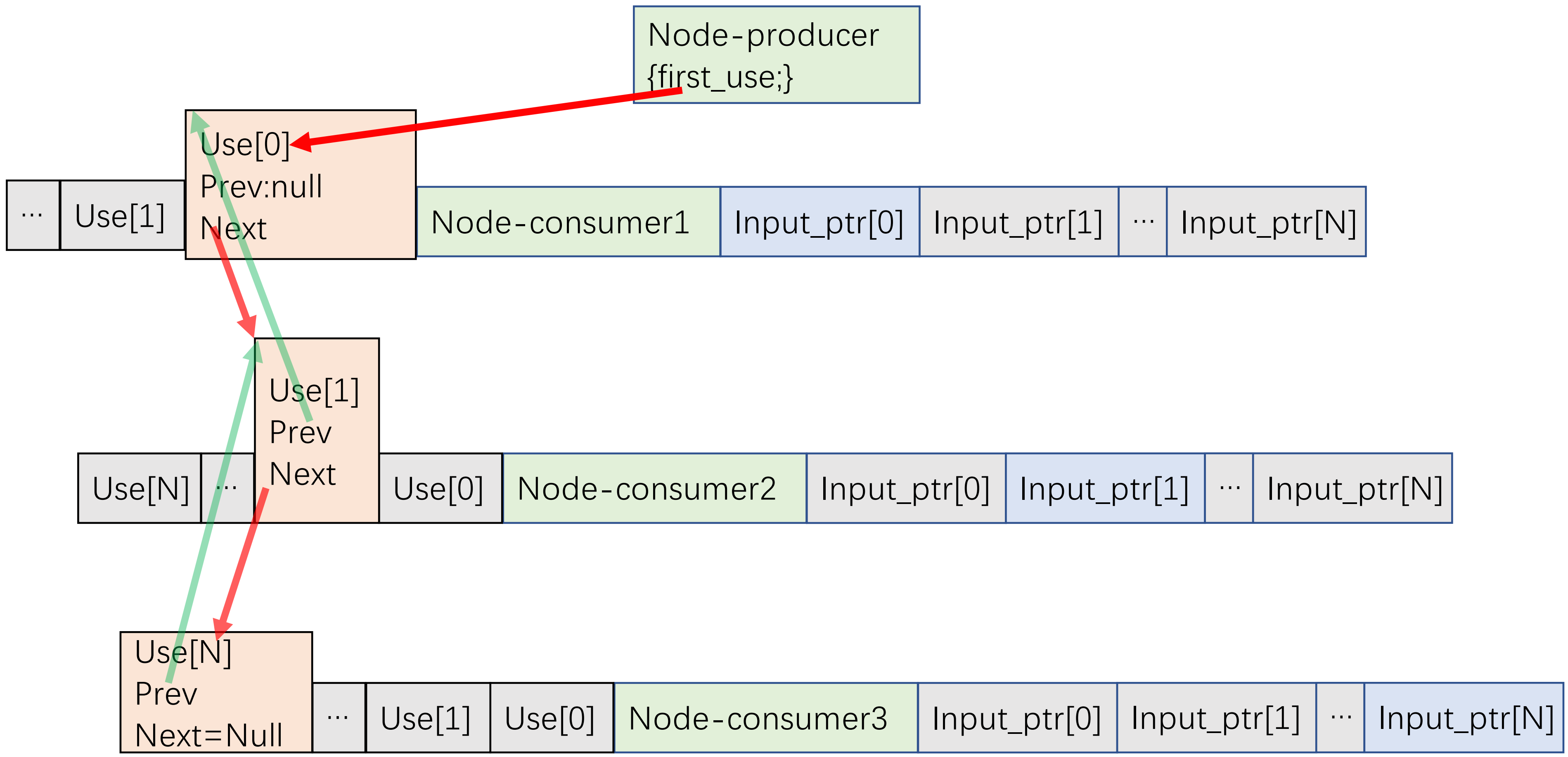
Use-Def的遍历：已知Node如何获得所有Input



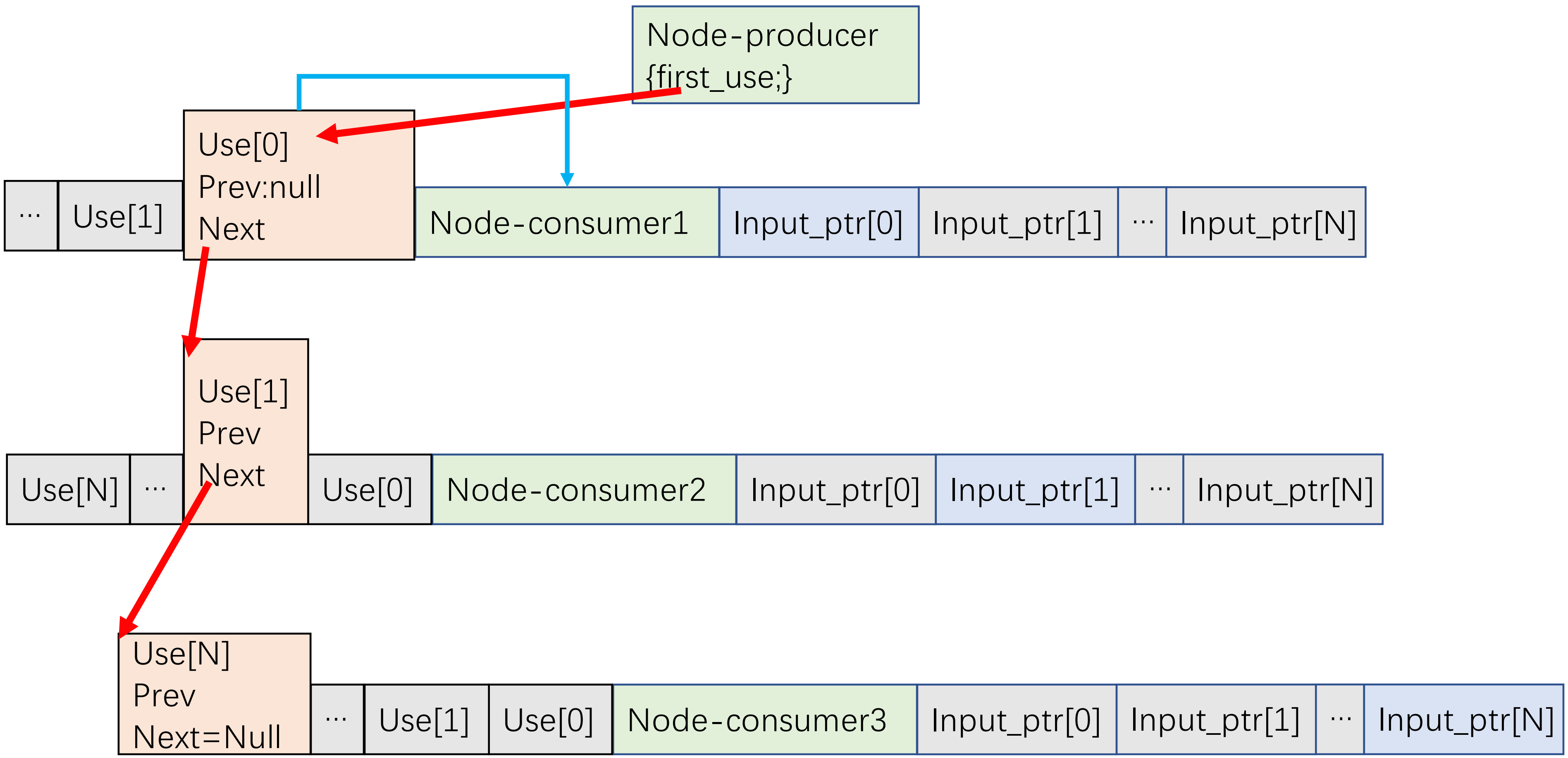


class Node::Inputs::const_iterator

Def-Use的遍历：已知Node如何获得所有的使用者



Def-Use的遍历：已知Node如何获得所有的使用者



Class Nodes-2

- 再看看这一大段、重要的注释(too long and must read&understand)

```
//===========================================================================
//== Memory layout ==========================================================
//===========================================================================
// Saving space for big graphs is important. We use a memory layout trick to
// be able to map {Node} objects to {Use} objects and vice-versa in a
// space-efficient manner. 为了节省空间地实现Nodes到Use的互相映射（查找），也就是说有Node指针，要能找到所有Use信息，有Use信息，要能找到Node指针
//
// {Use} links are laid out in memory directly before a {Node}, followed by
// direct pointers to input {Nodes}. Use在Node之前（低地址），且有指针指向Node的input（其实一个Node的input就是它的Use，它们是一一对应的，这里我怕也费解，为什么既要在Node后的高地址存input数组，又要在Node前的低地址存一个跟input数组对应的Use chain？为啥不把高地址的input数组直接搞成链表的形式呢？精妙在？）
//
// inline case://内联的形式：很直观，当input少于16个时，input指针数组直接从0~N存在Node后连续的高地址，对应的N个Use，从N~0反着存在Node前连续的低地址（注意！input直接存的一个个的指针，没有其他内容）
// |Use #N  |Use #N-1|...|Use #1  |Use #0  |Node xxxx |I#0|I#1|...|I#N-1|I#N|
//          ^                                ^                  ^
//          + Use                            + Node             + Input
//
// Since every {Use} instance records its {input_index}, pointer arithmetic
// can compute the {Node}. 一个Node是40Byte，一个Use struct是24Byte，每个Use中有bitfiled存着它对应input的index，所以可以简单地根据存储规则计算出任何一个Use对应的Node和input的地址（绕，需要看代码）
//
// out-of-line case: 还没看懂，也没遇到过这样的实例。。。😷
//      |Node xxxx |
//      ^         + outline ------------------+
//      +-----------------------------------------+
//                                        |        |
//                                        v        | node
// |Use #N  |Use #N-1|...|Use #1  |Use #0  |OOL xxxxx |I#0|I#1|...|I#N-1|I#N|
//          ^                                                 ^
//          + Use                                             + Input
//
// Out-of-line storage of input lists is needed if appending an input to
// a node exceeds the maximum inline capacity.
```

Class Nodes-3

- 最后看代码（src/compiler/node.h，省略不粘贴了
- 以最简单的inline case来理解Node的layout和实现，有文档提到这样实际上达到了TF IR的SSA化的效果。。。也是很费解
- 我的理解：
  - SSA是静态单一赋值，就是每个Name不能被重复地def，在这种实现中，被def的是Node本身，它的Use（也就是input）肯定来源于其他Nodes，且用地址（ptr）来作为name，所以，这样的内存组织形式，就达到了SSA的效果。。。
- 所以这种实现虽然有点绕，但一方面节省空间，另一方面还SSA化了IR，精妙绝伦，值得玩味（：

Struct Use（详看蓝色注释）

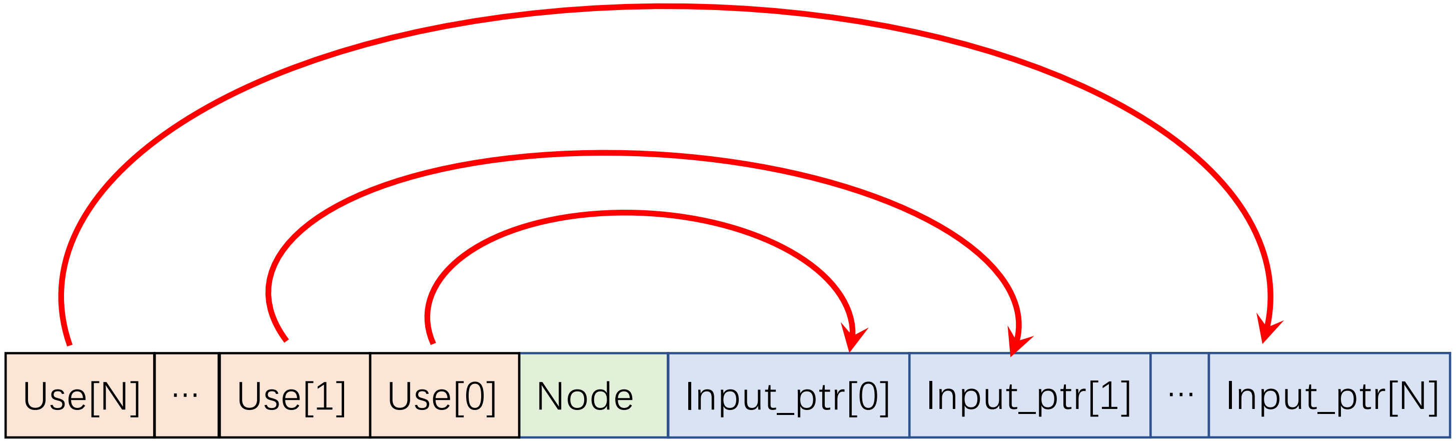
-

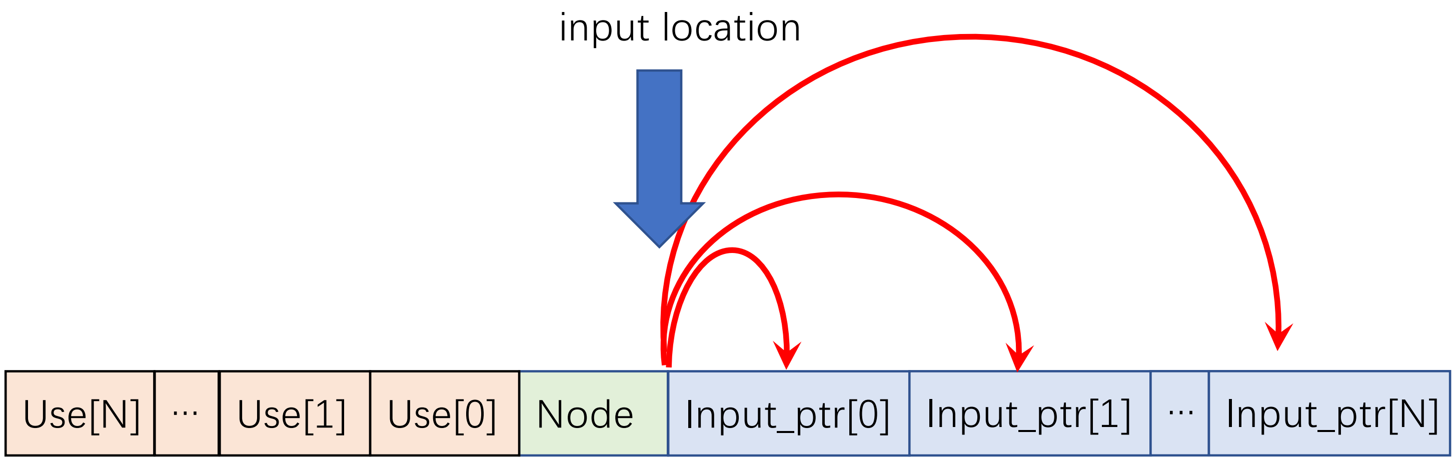
```
  // A link in the use chain for a node. Every input {i} to a node {n} has an
  // associated {Use} which is linked into the use chain of the {i} node.
  struct Use {
    Use* next;
    Use* prev;  //形成链表结构
    uint32_t bit_field_;//Use的标识信息，用32位的bit_field来节省空间


    int input_index() const { return InputIndexField::decode(bit_field_); }//对应的input_index信息就在Use
    bool is_inline_use() const { return InlineField::decode(bit_field_); }//对应的input是不是inline的信息也在use中
    Node** input_ptr() { //计算对应的input的地址：步骤是先算出它对应的Node地址，再根据是否inline来计算inputs数组收地址，最后根据index来计算对应的input地址
      int index = input_index();
      Use* start = this + 1 + index;
      Node** inputs = is_inline_use()
                          ? reinterpret_cast<Node*>(start)->inline_inputs()
                          : reinterpret_cast<OutOfLineInputs*>(start)->inputs();
      return &inputs[index];
    }


    Node* from() { //from的意思，就是前述“反向数据/控制流”的“入节点”，对应地，“入节点”就是这个Use自身，所以这里是在计算这个Use节点对应的Nodes （有点绕。。。
      Use* start = this + 1 + input_index();
      return is_inline_use() ? reinterpret_cast<Node*>(start)
                             : reinterpret_cast<OutOfLineInputs*>(start)->node_;
    }


    using InlineField = base::BitField<bool, 0, 1>;
    using InputIndexField = base::BitField<unsigned, 1, 31>;
```

Node {
opcode：操作码
type_： 类型
mark_：记号
bit_field_：编码nodeid/InlineCount/InlineCapactity
first_use_: 本Node的使用者链表
}

```
Use {
next : *
prev : *
bit_field_ : 编码InputIdex/InlinedField
}
```

Use[0]	···	Use[N-1]	Use[N]	Node	Input_ptr[0]	Input_ptr[1]	···	Input_ptr[]


Use[N]
···
Use[1]
Use[0]
Node
Input_ptr[0]
Input_ptr[1]
···
Input_ptr[N]


input location
Use[N]
···
Use[1]
Use[0]
Node
Input_ptr[0]
Input_ptr[1]
···
Input_ptr[N]

谢 谢

欢迎交流合作

2020/02/27