深入理解 HPDA

范学鹏

2023-06-20

序

HPDA 是一个使用 C++ 开发的数据分析框架。在实际的研发工作中，不管是需要开发 HPDA（在 HPDA 中增加功能）的开发人员，还是使用 HPDA 开发数据分析程序的开发人员，对于 HPDA 总是有各种问题。这些问题很庞杂，而且很多时候超出了 HPDA 本身。这一方面是因为 C++ 语言本身比较复杂，HPDA 中使用了一些不那么常见（至少超出了一般的教学范围）的编程技巧；另一方面也是因为 HPDA 的设计哲学更偏向于编译期。

本文试图汇总开发人员经常遇到的问题，并尽可能系统性的回答这些问题。本文按照 HPDA 的组件组织，分别介绍了每个组件的设计思路，介绍了一些用到的技术，并且在每个部分都给出了一些实例代码。对于不能归类到某个 HPDA 组件的问题，本文在最后也罗列了相应的问题。由于总会遇到新的问题，本文也会持续更新。

本文并不是讲解关于 C++ 编程技巧的文章（这方面的书籍、文章已经太多了），因此在讲到某些技术的时候，也仅以理解 HPDA 中的实践为目的，并不试图全面的讲解某个技术。考虑到国内对于 C++ 的教育情况，建议读者先了解 Modern C++。

本文不是 HPDA 的产品文档，也不能作为 HPDA 的开发文档或技术文档。虽然本文会覆盖一些如何使用 HPDA 的内容，但是一方面不够系统（不包含完整的接口或 API 列表），另一方面存在一定的延时。因此需要以实际的 HPDA 文档为准，本文只是帮助开发人员理解 HPDA 的实现、并为开发人员解决一些工程问题。

本文还在编辑中，如果有任何更新，或发现任何错误，请在如下的 Gitee 链接中提交相应的 Issue 或 Pull Request。

`https://gitee.com/YeeZTech/understanding-hpda`

目录

Chapter 1

HPDA 简介

HPDA(High Performance Data Analytics) 是一个数据分析框架。本章介绍 HPDA 的一些特点，尤其是我们为什么要再设计一个新的数据分析框架。本章还将介绍如何获取 HPDA、如何使用 HPDA 进行开发。最后，我们给出了一个简单的例子，以便能够快速入门。

1.1 HPDA

HPDA（High Performance Data Analytics，高性能数据分析）是一个数据分析框架。从用途上，HPDA 是一种 OLAP 分析框架，类似于 Hadoop、Clickhouse 等。

HPDA 适合结构化、以及非结构化数据的分析、处理。例如，统计一个班级的成绩、统计一个公司的薪资、年龄、学历等；同样的，HPDA 也可以用来处理图片、音频数据、视频数据等。

在更复杂的场景下，例如机器学习，HPDA 也提供了足够的扩展能力，通过集成相应的算子，可以获得相应的机器学习能力。

HPDA 仅适合数据分析，不适合作为事务处理（OLTP）的框架，例如维护图书信息管理系统等，这是因为 HPDA 仅考虑数据的读问题，几乎不考虑数据的更新问题，也不包含在并发操作时如何保证数据一致性的组件。

HPDA 有很弱的写数据算子，这仅仅是为了能够保存最终的数据分析结果，不能对已经存在的数据进行更新（例如插入和修改）。

HPDA 基于 C++ 开发，使用了诸多的 C++11 的特性。同时，HPDA 也提供了一些其他语言的绑定，例如 SQL。因此，开发者可以以各种方式使用 HPDA。

1.1.1 为什么要设计一个新的数据分析框架

看上去,HPDA 解决的场景和问题已经有足够多的框架了,例如 Hadoop、Hive、Clickhouse 等，那么为什么我们要再设计一个新的数据分析框架呢?

最为直接的原因，是我们需要一个能够在 Intel SGX 内执行的数据分析框架，已有的诸多框架要么存在诸多的迁移问题，要么不适合 Intel SGX 这样的执行环境。Intel SGX 是一种可信执行环境（Trusted Execution Environment，TEE），有如下几个特点：

- 内存受限：在早期的 Intel SGX 版本中，可以使用的内存只有数百兆字节，这使得大多数数据分析框架都难以使用，虽然后来的 Intel SGX 版本解除了这一限制（可使用的内存有几十 GB），但考虑到 TEE 技术的多样性（例如 TrustZone，Keystone，以及基于 PCI-E 的 TEE），内存受限将会是常态。

- 系统调用受限：为了减少攻击面，TEE 通常会限制可以使用的系统调用，例如，Intel SGX 中不能直接创建线程、进行 IO 等，这使得已有的数据分析框架不能直接在可信执行环境中使用。

- 数据加解密：为了保证数据安全，开发者通常需要设计一整套密码交互协议，例如对加载到 TEE 的数据加密，进入 TEE 后解密，同样的，对即将离开 TEE 的数据也需要进行加密操作，已有的数据分析框架对于密码协议的集成是困难的。

基于上述考虑，我们设计了新的数据分析框架。虽然名字中包含了高性能（High Performance），但目前，HPDA 的性能仍然有限，还需要诸多的优化。

同时，我们也加入了一些新的考虑，即利用编译器让 HPDA 程序更加健壮，这在以往的数据分析框架中是不太常见的。第2章详细讨论了这方面的考虑，并且说明了这种考虑是如何影响 HPDA 的设计的。

1.1.2 获取 HPDA

目前，HPDA 是 YPC 的一个组件，可以在以下链接找到 YPC。

`https://github.com/YeeZTech/YeeZ-Privacy-Computing`

在目录 `include/hpda` 中包括所有的头文件，在目录 `hpda` 中包括相应的源码。

独立使用时，可以链接 libhpda.so，也可以在 CMake 中链接 YPC::hpda。目录 `hpda/test` 中包含了一些测试例子，也可以看作示例。

1.2 一个例子

没有什么比一个 Hello World 程序更能说明一个程序的基本结构了。但考虑到 HPDA 是一个数据分析框架，我们使用一个同样简单、但更加贴切的例子来说明 HPDA 的基本结构。

我们考虑一个班级，这个班级有 n 个学生，在一次考试后，老师希望对这 n 个学生做一个简单的统计：看看哪些同学的成绩超过了 60 分。虽然可以使用 Excel 等工具完成这一工作，但这不妨碍我们以此说明 HPDA 的能力。

以下代码是一个完整的例子：

```
1 #include <iostream>
2 #include <string>
3 #include <hpda/hpda.h>
4 define_nt(name, std::string)
5 define_nt(score, int)
```

```
6  typedef hpda::ntobject<name, score> data_item_t;
7
8  int main(int argc, char * argv[]){
9      hpda::engine engine;
10     hpda::extractor::raw_data<name, score> rd;
11     rd.set_engine(&engine);
12
13     std::string n;
14     int s;
15
16     for (int i = 0; i < 50; i++) {
17       data_item_t d;
18       std::cout<<"input "<<i<<"th student name and
           score"<<std::endl;
19       std::cin>>n;
20       std::cin>>s;
21       d.set<name, score>(n, s);
22       rd.add_data(d);
23     }
24
25     hpda::processor::filter<name, score> f(&rd, [](
          const data_item_t &d) {
26       return d.get<score>() >= 60;
27     });
28
29     hpda::output::memory_output<name, score> mo(&f);
30
31     engine.run();
32     for(auto v : mo.values()){
```

```
33          std::cout<<v.get<name>()<<": "<<v.get<score>()
                <<std::endl;
34      }
35
36      return 0;
37  }
```

在这个例子中，我们从控制台输入学生的名字和分数，这当然有些繁琐，你也可以从文件读取。

接下来，我们来看一下整个程序。首先，你会注意到使用 HPDA 的时候，需要一个头文件，

```
1       #include <hpda/hpda.h>
```

这个头文件包含了 HPDA 大多数的头文件，在实际的项目中，你也可以根据自己的需要仅包含需要的头文件，这样编译速度会更快。在某些情况下，你可能只能使用特定的头文件，而不是直接使用 `hpda/hpda.h`。

紧接着，你会注意到以下三行代码，

```
1       define_nt(name, std::string)
2       define_nt(score, int)
3       typedef hpda::nt_object<name, score> data_item_t;
```

其中，`define_nt` 是一个宏，用于定义数据项的类型信息，包含了两个参数，第一个参数是该数据项的名字，第二个参数是该数据项的类型。因此，前两行代码定义了两个数据项，第一个是类型为 `std::string`，名字为 `name`，第二个是类型为 `int`，名字为 `score`。第三行则将这两个数据项定义为一个数据对象类型，`data_item_t`。关于这些宏的定义和使用，可以参考第2章中关于 `ntobject` 的说明。

在 `main` 函数中，首先定义了执行 HPDA 所需的引擎（上下文）`hpda::engine`，其中包含了执行时所需的内部数据结构、缓存等。紧接着，定义了数据源 `rd`，并将数据源的执行引擎设置为刚刚定义的 `engine`。

```
1     hpda::extractor::raw_data<name, score> rd;
2     rd.set_engine(&engine);
```

此时数据源中还没有数据，因此，我们需要将实际的数据填充到数据源中，接下来的几行从控制台中获取了相应的数据，并通过这行代码填充数据，

```
1     rd.add_data(d);
```

然后，定义一个过滤器 f，并将过滤器的输入定义为数据源 rd。定义过滤器时，需要提供一个判定函数，过滤器仅输出该判定函数返回为 true 的数据。

```
1     hpda::processor::filter<name, score> f(&rd, [](
         const data_item_t &d) {
2       return d.get<score>() >= 60;
3     });
```

然后，定义输出算子 mo，其输入为过滤器 f。HPDA 定义了多种输出算子，此处的输出算子是直接讲输出保存在内存中，仅适合不大的数据量。

```
1     hpda::output::memory_output<name, score> mo(&f);
```

最后，执行整个 HPDA 引擎，在输出算子中获得相应的数据。

```
1     engine.run();
```

这个 HPDA 程序可以看作一个计算图，计算图是一个有向图[1]，其中节点表示一个算子，边表示数据，仅有出边的算子为输入算子，仅有入边的算子为输出算子，第3详细讨论了计算图的细节。如图1.1所示，为本节所述例子对应的计算图。

1.2.1 执行过程

对于初次接触计算图的人来说，整个执行过程可能不太直观，因此，有必要简要的说明一下执行过程，详细的算法可以参考第??章。简单来说，在

[1]HPDA 目前仅支持有向无环图，未来会支持有向有环图

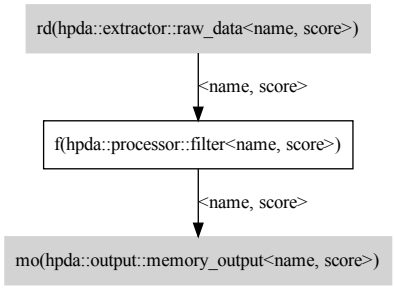

图 1.1: 一个简单的计算图。

`engine.run()` 被调用后，`engine` 会将输出算子入栈，并递归的将算子的输入算子入栈，当一个算子有输入时，则算子出栈、处理相应的输入数据。

考虑序列 $(a, 61), (b, 59), (c, 92)$ 在图1.1中的执行顺序。如图1.2所示。最终，执行序列为 `rd(a, 61)`, `f(a, 61)`, `mo(a, 61)`, `rd(b, 59)`, `f(b, 59)`, `rd(c, 92)`, `f(c, 92)`, `mo(c, 92)`。

1.2.2 扩展

下面考虑上述例子的一个简单扩展：分别输出及格的学生和不及格的学生。HPDA 提供了一个特别的算子，`split`。如图1.3所示，`split` 算子将自己的输入复制为两份，分别输出给两个过滤器，`greater` 和 `less`，从而实现了分别输出及格的学生和不及格的学生。

本文将本代码的实现留给读者，此处不再赘述。当然，你可以查阅第5章以更详细的了解如何开发 HPDA 程序。

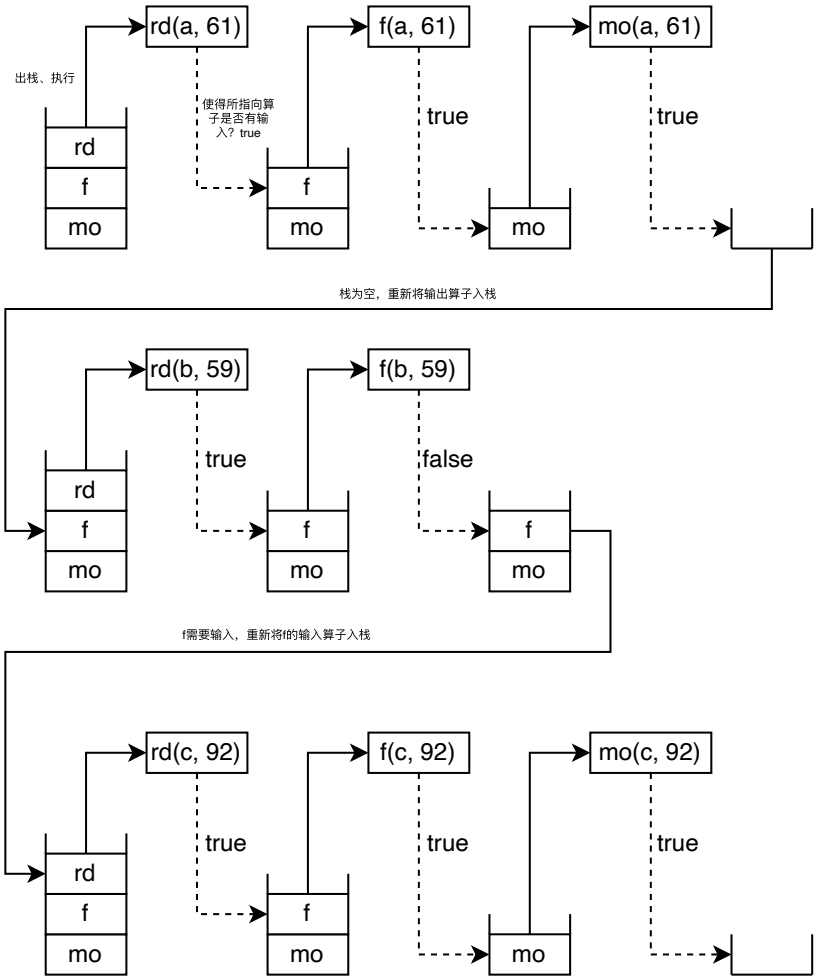

图 1.2: 序列 $(a, 61), (b, 59), (c, 92)$ 在图1.1所示的计算图上的执行顺序。

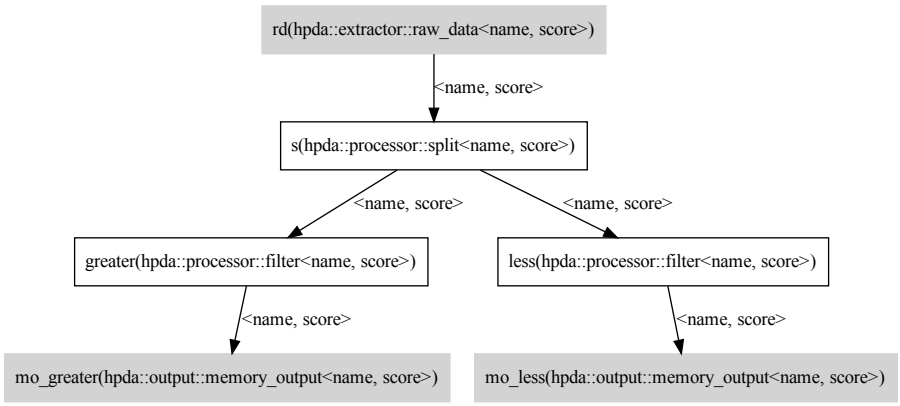

图 1.3: 同时获取及格和不及格学生的计算图。

Chapter 2

运行时 & 编译期

C++ 编译器如此强大，我们应该尽可能让编译器帮我们检查代码中的错误、优化代码。本章我们对比运行时的编程技巧和编译期的编程技巧。本文的目的并不是完整的介绍相关的编程技巧（那远远超出了本文的范围），因此，我们仅介绍 HPDA 中用到的那些技巧。最后，本文介绍了 HPDA 中的核心组件，`ntobject` 的实现以及如何使用 `ntboject`。

2.1 运行时

运行时（runtime），顾名思义，是指程序在运行时决定的行为，这实在是太常见了，我们日常接触到的代码大多数都是运行时的。例如：

```
1     int foo(bool a){
2         if(a){
3             std::cout<<"this is true"<<std::endl;
4         }else{
5             std::cout<<"this is false"<<std::endl;
6         }
7     }
```

其中 a 的值在运行时决定了输出的内容。

在面向对象的程序语言（例如 C++）中，运行时多态通常是指通过虚函数（virtual）实现运行时的调用转发。有诸多编程技巧是利用虚函数实现的，例如设计模式中的观察者模式等，此处不再赘述。

2.2 编译期

编译期（compile time），则是指程序的行为在编译时就决定了。更具体来说，一段代码的行为，在编译时才能确定。例如，下面这段代码的行为并不确定：当 T 为 int 时，sum 返回的是两个整数的和；当 T 为 std::string 时，sum 返回的是两个字符串拼接后的字符串；当 T 为其他类型是，sum 又有其他行为。

```
1 template<typename T>
2 T sum(const T & a, const T &b){
3     return a + b;
4 }
```

但是，不管怎么样，foo 的行为都需要在编译期，也就是 T 确定时，确定下来。C++ 的这种特性，我们称之为编译期行为。

这很大程度上是因为 C++ 的模版本身是图灵完备的，因此，理论上，可以使用 C++ 模版完成任意程序的行为。相关技巧被称为*元编程*或*泛型编程*。例如，可以使用该技巧实现阶乘运算。

```
1 template <int N> struct Factorial {
2     enum { value = N * Factorial<N - 1>::value };
3 };
4
5 template <> struct Factorial<0> {
6     enum { value = 1 };
7 };
```

```
8
9  void foo(){
10     int x = Factorial<4>::value; // == 24
11     int y = Factorial<0>::value; // == 1
12 }
```

详细的介绍元编程或泛型编程的技巧，将大大超出本文的范围，因此本文仅介绍个别相关的内容。

2.2.1 SFINAE

SFINAE 是指替换失败不是错误 (Substitution Failure Is Not An Error)。这是 C++ 函数模板的一个重要规则。

在函数模板的重载决议中会应用此规则：当模板形参在替换成显式指定的类型或推导出的类型失败时，从重载集中丢弃这个特化，而非导致编译失败。例如，在如下代码的最后一行中，第10行中 `h<int>` 可以对应两种不同的模版替换方案，但是在第一种方案（第4行到第5行）中，因为不存在 `int::type`，因此，该替换方案是失败的。根据 SFINAE 原则，替换失败时，会丢弃这个特化，因此，`h<int>` 仅剩下唯一的替换方案（第二种方案，第7行到第8行）。

```
1  template<typename A>
2  struct B { using type = typename A::type; };
3
4  template<class T>
5  typename T::type h(typename B<T>::type){};
6
7  template<class T>
8  void h(...) {}
9
10 using R = decltype(h<int>(0));
```

在实际开发中，会经常遇到如下的代码，正是利用了 SFINAE 特性。第4行中，is_type_in_type_list 是一个元函数，当该元函数中的值 value 为 false 时，编译器会丢弃该特化。

```
1 template<typename OT, typename CT>
2 auto check_type(const OT & t)
3     -> typename std::enable_if<
4         is_type_in_type_list<CT, typename OT::
              type_list>::value,
5         void>::type{
6         //...
7 }
```

使用 SFINAE 特性，还可以对某些特定的类型进行操作。

2.3 运行时 & 编译期

2.3.1 不要将编译期代码放到运行时执行

这是一个初学者容易犯的错误：将编译期代码和运行时代码混淆。下面的代码说明了这一错误。在函数 do_something 中，虽然第12行的判断是合理的，但是第13行却不合理。因为即使第12行的判定为 false，第13行仍然需要编译，但此行代码却可能由于类型 T 中不存在方法 size() 而编译失败。第21 行导致了这一编译错误。相对的，第22行则不存在编译错误。这也说明了，即使代码的编译、运行都是正确的，这份代码仍然有可能是错误的！

```
1 template<typename T>
2 struct support_type{
3     constexpr static bool value = false;
4 };
5 template<>
```

```
6  struct support_type<std::string>{
7      constexpr static bool value = true;
8  };
9
10 template<typename T>
11 void do_something(const T & val){
12     if(support_type<T>::value){
13         std::cout<<val.size()<<std::endl;
14     }else{
15         std::cout<<val<<std::endl;
16     }
17 }
18 int main(int argc, char *argv[]){
19     int a;
20     std::string b;
21     do_something(a); //产生编译错误，因为int类型没有
          size()方法。
22     do_something(b);
23 }
```

下面的代码则是一种正确的做法，使用重载而不是运行时的判断。

```
1  template<typename T>
2  struct support_type{
3      constexpr static bool value = false;
4  };
5  template<>
6  struct support_type<std::string>{
7      constexpr static bool value = true;
8  };
```

```
9
10 template<typename T>
11 auto do_something(const T & val) ->
12 typename std::enable_if<support_type<T>::value, void
      >::type{
13     std::cout<<val.size()<<std::endl;
14 }
15 template<typename T>
16 auto do_something(const T & val)->
17 typename std::enable_if<!support_type<T>::value, void
      >::type{
18     std::cout<<val<<std::endl;
19 }
20
21 int main(int argc, char *argv[]){
22     int a;
23     std::string b;
24     do_something(a);
25     do_something(b);
26 }
```

2.3.2 结合运行时和编译期

虽然区分运行时和编译期是重要的，但有时也需要将两者结合，以获得更简洁的代码。继承时使用自己作为模板参数实例化的类型作为父类就是一个典型的例子。下列代码是 C++11 中 `enable_shared_from_this` 的实现 [1]。

[1]注意，`shared_ptr` 中会通过重载的方式判断指针类型是否继承自 `enable_shared_from_this`，若是，会直接对其中的 `__weak_this` 赋值。

```cpp
template<typename _Tp>
class enable_shared_from_this{
    mutable weak_ptr<_Tp> __weak_this_;
protected:
    enable_shared_from_this() _NOEXCEPT {}

    enable_shared_from_this(enable_shared_from_this
       const&) _NOEXCEPT {}

    enable_shared_from_this& operator=(
       enable_shared_from_this const&) _NOEXCEPT{
        return *this;
    }

    ~enable_shared_from_this() {}
public:
    shared_ptr<_Tp> shared_from_this(){
        return shared_ptr<_Tp>(__weak_this_);
    }

    shared_ptr<_Tp const> shared_from_this() const{
        return shared_ptr<const _Tp>(__weak_this_);
    }

    template <class _Up> friend class shared_ptr;
};
```

下面的代码是一个使用 `enable_shared_from_this` 的例子[2]。

[2]使用其中的 `getptr()` 方法获取指针后，可以用作异步调用

```
class Good : std::enable_shared_from_this<Good> {
public:
 std::shared_ptr<Good> getptr() {
     return shared_from_this();
 }
}
```

另一个典型的例子是模板类继承自非模板的基类，这一技术有时候也被称为类型擦除 (Type Erasure)。使用类型擦除，可以将不同类型的子类对象放到一个容器中处理。下面的代码说明了如何使用类型擦除技术。HPDA 中使用了这一技术将不同类型的算子置于一个计算图中处理。

```
class Shape{
public:
    virtual void draw() = 0;
};
template <typename T>
class Point{
public:
    T x;
    T y;
};

template <typename T>
class Circle : public Shape{
public:
    virtual void draw(){
        std::cout<<"this is a circle"<<std::endl;
    }
    Point<T> & point(){return point;}
```

```
19      const Point<T> & point()const{return point;}
20      T & radius() {return radius;}
21      const T & radius() const {return radius;}
22  protected:
23      Point<T> point;
24      T radius;
25  };
26
27  class NullShape : public Shape{
28  public:
29      virtual void draw(){
30          std::cout<<"this is null shape"<<std::endl;
31      }
32  };
33  void generate_and_draw(){
34      std::vector<Shape *> all_shapes;
35      all_shapes.push_back(new Circle<double>());
36      all_shapes.push_back(new NullShape());
37
38      for(auto s : all_shapes){
39          s->draw();
40      }
41  }
```

需要注意的是，有时候需要使用 `std::is_base_of` 重新获得相应的类型，但 `std::is_base_of` 是运行时的，而不是编译期的。

2.4 优劣对比

C++ 是少有的在编译期提供了图灵完备能力的语言，对于模版相关特性的更新，也一直是 C++ 语言中非常活跃的部分。这也常常让 C++ 的用户感到困扰：何时应该使用元编程技巧、何时应该使用运行时编程技巧? 这方面的争论一直存在，也很难达成共识。HPDA 使用了诸多元编程技术，因此，本文从 HPDA 的角度说明这种选择的考虑。如果这种考虑在其他系统的设计与实现中也存在，编译期的技术将是十分有用的。

首先，编译期的技术有助于提高开发效率。编译器能够检查诸多类型错误，这能够大大减少运行时错误的频率，缩短开发时间。

举个例子，在 `ntobject` 中的存在着 `set` 方法，其实现如下[3]

```
1 template <typename CT>
2 void set(const typename internal::nt_traits<CT>::type
     &val) {
3     static_assert(is_type_in_type_list<CT, util::
        type_list<ARGS...>>::value,
4                   "Cannot set a value that's not in the
                       ntobject/row!");
5     const static int index =
6         get_index_of_type_in_typelist<CT, util::
            type_list<ARGS...>>::value;
7     std::get<index>(*m_content) = val;
8 }
```

其中行3的作用是在编译时检查其中是否存在类型错误。我们可以在行3存在以及行3 不存在两种情况下编译以下代码。注意，下列代码中存在明显的错误，即 `kname` 不在 `myobj_t` 中。

```
1 #include "ff/util/ntobject.h"
```

[3]代码位于 `fflib/include/ff/util/ntobject.h`

```
2
3  define_nt(email, std::string, "email");
4  define_nt(uid, uint64_t, "ui");
5  define_nt(uname, std::string, "uname");
6  define_nt(kname, std::string, "kname");
7
8  void use_ntobject() {
9    typedef ff::util::ntobject<email, uid, uname>
       myobj_t;
10   myobj_t obj;
11   obj.set<uname>("xuepeng");
12   obj.set<uid>(122);
13   obj.set<email>("xp@example.com");
14   obj.set<kname>("xp2@example.com");
15   std::cout<<"uname is "<<obj.get<uname>()<<std::endl;
16 }
```

行3存在时 编译器检查生效，编译器提示如下错误，可以看到，编译错误中明确提示了正确的错误。

```
In file included from /XXX/example_ntobject.cpp:27:
/XXX/fflib/include/ff/util/ntobject.h:83:5: error: static_assert failed
  due to requirement
  'is_type_in_type_list<kname, ff::util::type_list<email, uid, uname>>::value'
  "Cannot set a value that's not in the ntobject/row!"
    static_assert(is_type_in_type_list<CT, util::type_list<ARGS...>>::value,
    ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/XXX/fflib/example/sql/example_ntobject.cpp:41:7: note: in instantiation
  of function template specialization 'ff::util::ntobject<email, uid,
    uname>::set<kname>' requested here
  obj.set<kname>("xp2@example.com");
```

```
      ^
1 error generated.
```

行3不存在时　没有引入编译器检查，此时正常编译通过，但是当执行时，输出如下，可以看到输出的内容是错误的[4]。

```
uname is xp2@example.com
```

试想，在上述例子中，如果没有使用编译期的技术，而是任由这样的错误在运行时触发，开发人员将耗费数倍的时间和精力来解决这样的错误。因此，使用编译期的技术有助于提高开发效率。

其次，使用编译期技术能够对代码进行必要的优化。这体现在两个方面，减少分支预测和编译期计算。

最后，使用编译期技术（尤其是元编程技术）可能会导致代码大小过大，影响代码缓存的命中率。这一说法在理论上是成立的，但是笔者并没有发现详细的测试数据。

2.5 ntobject

ntobject 是 fflib[5]中的组件，HPDA 使用该组件描述需要处理的数据内容。ntobject 是一个供开发者定义数据结构的库，并且包含了相关的工具。

2.5.1 初衷

开发人员需要经常和各种各样的数据对象打交道。例如学生的基本信息、图书管理系统中书籍的信息、系统中一个待调度的任务的描述信息等。定义这些数据对象有两种方式，一种是对每种数据对象定义一个类，例如

[4]这是因为覆盖了 uname 中原本的内容

[5]https://github.com/AthrunArthur/fflib

下面的类描述了学生档案信息。使用这种方式的缺点是随着数据对象越来越多，开发人员的负担也越来越重[6]。

```
1 class student{
2 protected:
3     std::string name;
4     std::string id;
5     int gender;
6     date birthday;
7 };
```

另一种方式是使用动态的数据结构，例如 JSON 等，也可以自定义，下面的代码片段展示了一种自定义的方法。通过在运行时添加字段确实可以以更短的代码获得良好的扩展性，但是同时也失去了必要的类型信息，这容易将类型错误推迟到运行时，导致开发效率降低。

```
1 struct item{
2     std::string name;
3     int type;
4     char data[1];
5 };
6 std::vector<item *> student;
```

这些数据对象还有另一个方面需要考虑：数据对象的序列化问题。数据对象的序列化（也包括反序列化）通常有三个场景：

- 数据库
- 网络或文件（二进制化）
- JSON（可读）

[6]尤其是其中还需要添加对每个字段的读写方法

虽然存在诸多的序列化工具，但通常只针对某特定场景，而不是对一个数据对象支持三种场景。

ntobject 则提供了统一的定义数据对象的方式，并且支持尽可能多的编译期检查技术，提高开发效率。

2.5.2 ntobject 实现

ntobject 的实现十分短小，不足百行。ntobject 表示一个数据对象，其中包含了若干数据项，每个数据项则包含了相应的数据类型信息。ntobject 内部使用了 shared_ptr 维护该数据行的内存[7]，使用 shared_ptr 使得 ntobject 对象的拷贝更加高效。

```
1  template <typename... ARGS> class ntobject {
2  public:
3    typedef typename util::type_list<ARGS...> type_list;
4    typedef typename convert_type_list_to_tuple<typename
         nt_extract_content_type_list<util::type_list<
       ARGS...>>::type>::type content_type;
5
6    ntobject() : m_content(new content_type()) {}
7
8    template <typename OT>
9    ntobject(const OT &data) : m_content(new
       content_type()) {
10     assign_helper<OT, ARGS...>(data);
11   }
12
13   ntobject<ARGS...> make_copy() const {
14     ntobject<ARGS...> rt;
```

[7]出于性能考虑，不建议使用 std::vector<ntobject>，建议使用 ntarray

```
15      *rt.m_content = *m_content;
16      return rt;
17    }
18
19    template <typename CT>
20    void set(const typename internal::nt_traits<CT>::
          type &val) {
21      static_assert(is_type_in_type_list<CT, util::
            type_list<ARGS...>>::value, "Cannot set a value
             that's not in the ntobject/row!");
22      const static int index =
            get_index_of_type_in_typelist<CT, util::
            type_list<ARGS...>>::value;
23      std::get<index>(*m_content) = val;
24    }
25
26    template <typename CT, typename CT1, typename...
          CARGS, typename... PARGS>
27    void set(const typename internal::nt_traits<CT>::
          type &val, const typename internal::nt_traits<CT1
          >::type &val1, PARGS... params) {
28      static_assert(is_type_in_type_list<CT, util::
            type_list<ARGS...>>::value, "Cannot set a value
             that's not in the row!");
29      static_assert(is_type_in_type_list<CT1, util::
            type_list<ARGS...>>::value, "Cannot set a value
             that's not in the row!");
30      const static int index =
            get_index_of_type_in_typelist<CT, util::
```

```
        type_list<ARGS...>>::value;
31      std::get<index>(*m_content) = val;
32
33      set<CT1, CARGS...>(val1, params...);
34    }
35
36    template <typename CT> typename internal::nt_traits<
         CT>::type get() const {
37      static_assert(is_type_in_type_list<CT, util::
           type_list<ARGS...>>::value, "Cannot get a value
            that's not in the ntobject/row!");
38      const static int index =
           get_index_of_type_in_typelist<CT, util::
           type_list<ARGS...>>::value;
39      return std::get<index>(*m_content);
40    }
41
42    ntobject<ARGS...> &operator=(const ntobject<ARGS...>
        &data) {
43      if ((void *)&data == (void *)this) {
44        return *this;
45      }
46      assign_helper<ntobject<ARGS...>, ARGS...>(data);
47      return *this;
48    }
49
50    template <typename OT> ntobject<ARGS...> &operator=(
         const OT &data) {
51      if ((void *)&data == (void *)this) {
```

```
52        return *this;
53      }
54      assign_helper<OT, ARGS...>(data);
55      return *this;
56    }
57
58  protected:
59    template <typename OT, typename CT>
60    auto assign_helper(const OT &data) -> typename std::
          enable_if<is_type_in_type_list<CT, typename OT::
          type_list>::value, void>::type {
61      set<CT>(data.template get<CT>());
62    }
63    template <typename OT, typename CT>
64    auto assign_helper(const OT &data) -> typename std::
          enable_if<!is_type_in_type_list<CT, typename OT::
          type_list>::value, void>::type {}
65
66    template <typename OT, typename CT, typename CT1,
          typename... CARGS>
67    auto assign_helper(const OT &data) -> typename std::
          enable_if<is_type_in_type_list<CT, typename OT::
          type_list>::value, void>::type {
68      set<CT>(data.template get<CT>());
69      assign_helper<OT, CT1, CARGS...>(data);
70    }
71
72    template <typename OT, typename CT, typename CT1,
          typename... CARGS>
```

```
73    auto assign_helper(const OT &data) -> typename std::
          enable_if<!is_type_in_type_list<CT, typename OT::
          type_list>::value, void>::type {
74      assign_helper<OT, CT1, CARGS...>(data);
75    }
76
77  protected:
78    std::shared_ptr<content_type> m_content;
79  };
```

在 ntobject 中，一个数据项定义为一个结构体，如下

```
1  struct my_type{
2      constexpr static const char * name = "my_type";
3      typedef std::string type;
4  };
```

但是考虑到用户定义的数据项会使用 name，type，因此，为了避免名字冲突，ntobject 使用了一个冗长的字符串作为内部的变量名，并提供了一个宏 define_nt，让用户来定义一个数据项，其中第一个参数为名字，第二个参数为类型。可选的，还可以提供第三个参数，做为序列化（数据库、JSON）时的数据项名。如下代码所示。

```
#define define_nt(...) JOIN(define_nt_impl_, PP_NARG(__VA_ARGS__))(__VA_ARGS__)

#define define_nt_impl_2(_name, _dtype) \
  struct _name { \
    constexpr static const char *  \
    name_231bee33a5b2ee95e1b417bb350ea5c6b89aad1d81181b805e5ec957d9cea04a = #_name; \
    typedef _dtype \
    type_231bee33a5b2ee95e1b417bb350ea5c6b89aad1d81181b805e5ec957d9cea04a;  \
  };

#define define_nt_impl_3(_name, _dtype, _tname) \
```

```
  struct _name { \
    constexpr static const char * \
    name_231bee33a5b2ee95e1b417bb350ea5c6b89aad1d81181b805e5ec957d9cea04a = _tname; \
    typedef _dtype \
    type_231bee33a5b2ee95e1b417bb350ea5c6b89aad1d81181b805e5ec957d9cea04a;  \
  };
```

在 ntobject 中，要求每一个数据项对应的数据类型是存在默认构造函数的，并且可以复制 (重载了 operator= 操作)。因此，持有系统资源的 C++ 类不适合做为 ntobject 的数据项。

最后，ntobject 提供了相应的 traits，以免用户需要使用如此长的变量名。

```
template <typename T> struct nt_traits {
  constexpr static const char *name =
    T::name_231bee33a5b2ee95e1b417bb350ea5c6b89aad1d81181b805e5ec957d9cea04a;
  typedef typename
    T::type_231bee33a5b2ee95e1b417bb350ea5c6b89aad1d81181b805e5ec957d9cea04a type;
};
```

至此，我们介绍了 ntobject 的基本结构，ntobject 中还包括其他诸多代码和工具，读者可以自行阅读其中的代码，此处不再赘述。

2.6 使用 ntboject

使用 ntobject 时，需要首先定义数据项，并将数据项组合为数据对象。对于前文描述的学生信息一例，使用 ntobject 时的定义如下

```
1 define_nt(name,std::string);
2 define_nt(id, std::string);
3 define_nt(gender, birthday);
4 define_nt(birthday, date);
5 typedef ff::util::ntobject<name, id, gender, birthday>
      student;
```

其中 name, id, gender, birthday 是数据项而非类型，只有 student 才是类型。且 typedef ntobject<A, B, C, ...> T 中的 A, B, C 等必须全部是数据项而非类型。注意，define_nt 也可以在类或函数内使用，此时的作用域则是在类或函数内。

在下面的小节中，若不额外说明，我们均使用这一例子说明具体的用法。

2.6.1 读写操作

对 ntobject 的读写是直观的，如下代码是一种最简单的方式

```
1  student t;
2  t.set<name>("xuepeng");
3  std::cout<<t.get<name>()<<std::endl;
```

注意，ntobject.get 返回的不是引用，不能作为左值，所以下面的代码是错误的

```
1  student t;
2  t.get<name>() = "xuepeng";
```

ntobject 还支持多个字段的同时写操作，并且可以任意组合，并且与定义时声明的顺序无关，如下所示

```
1  student t;
2  t.set<name, gender, id>("xuepeng", male, "
       2023019847");
```

2.6.2 复合类型

ntobject 支持复合类型，例如基于容器的类型，如下

```
1  define_nt(friends, std::vector<std::string>);
2  typedef ff::util::ntobject<friends> my_friends;
```

更进一步的，ntobject 还能递归的定义类型，如下

```
1     define_nt(person, student);
2     define_nt(grade, uint);
3     typedef ff::util::ntobject<person, grade>
        grade_entry;
```

2.6.3 赋值操作

两个 ntobject 之间可以进行赋值操作，此时仅相同的数据项会被覆盖，如下代码所示，行6中，仅 name，id，gender 字段被赋值，birthday 字段则未被修改。

```
1     define_nt(address, std::string);
2     typedef ff::util::ntobject<name, id, gender,
        address> ya_student;
3     ya_student ya;
4     ya.set<name, id, gender, address>("xuepeng", "
        2023019847", male, "beijing");
5     student t;
6     t = ya;
```

注意：对于有嵌套定义的 ntobject，往往需要先（通常在栈内）制作内层 ntobject 的拷贝，然后读写后整体整体赋值。目前没有从外层 ntobject 直接对内层 ntobject 的数据项进行写的办法。

```
1     define_nt(person, student);
2     define_nt(grade, uint);
3     typedef ff::util::ntobject<person, grade>
        grade_entry;
4
5     student t; // 内层拷贝
```

```
6       t.set<name, gender, id>("xuepeng", male, "
            2023019847");
7
8       grade_entry g;
9       g.set<person>(t);
10      g.set<grade>(100);
```

2.6.4 遍历

当面对一个具体的 ntobject 对象时，遍历操作是简单的，例如在前面的 student 中，遍历所有的字段很简单。但是当面对一个尚未实例化的 ntobject 对象时，遍历操作就不那么直观了。下面展示了一段代码，调用 traverse 函数对各个字段进行遍历。注意此处是使用递归的方式对各个字段进行访问，而不是通常的循环遍历的方式。

```
1  template <int Index> struct traverse_helper {
2      template <typename T>
3      static auto do(T &&t) -> typename std::enable_if<(
           std::remove_reference<T>::type::type_list::len
           > Index), void>::type {
4          //do anything
5          std::cout<<std::get<Index>(*t.m_content)<<std
               ::endl;
6          traverse_helper<Index + 1>::do(std::forward(t)
               );
7      }
8
9      template <typename T>
10     static auto do(T && t) -> typename std::enable_if
           <(std::remove_reference<T>::type::type_list::
```

```
        len <= Index),void>::type {}
};

template<typename NT>
void traverse(const NT & obj){
    traverse_helper<0>::do(obj);
}
```

2.6.5 从数据项获得类型

在实际开发中，C++ 的类型推断和 auto 关键字能处理大部分类型使用的需求，但仍有一些情况我们希望从数据项获得类型。下面的代码展示如何使用 nt_traits 获得数据项的类型以声明一个哈希表

```
    define_nt(phone_number,long long int);
    define_nt(longitude, float);
    define_nt(latitude, float);
    define_nt(timestamp, long long int);
    typedef ff::util::ntobject<longitude, latitude,
       timestamp> NTO_loc_info;
    std::unordered_map<ff::util::internal::nt_traits<
       phone_number>::type,
        std::vector<NTO_loc_info>> groupMap;
```

2.6.6 数据库操作

fflib 提供了 SQL 封装（fflib::sql），以便对数据库进行操作。fflib::sql 也是基于 ntobject 开发，且兼容 MySQL。

稍有不同的是，数据库中一个表还涉及到主键等操作，因此，fflib::sql 提供了新的宏 define_column(name, type, dtype, tname) 其中

- name 表示数据项类型名;
- type 表示该列的类型，有三种取值，column，表示普通列，index，表示索引列，key，表示键值列;
- dtype 表示该列的数据类型，除了基本的数据类型，fflib::sql 还支持 MySQL 特有的数据类型[8];
- tname 表示该列在数据库中的名称。

下列代码说明了如何使用 ntobject 进行数据库操作。本文仅说明 ntobject 的能力，完整的说明如何使用 fflib::sql 超出本文的范围，此文不再赘述。

```
1  #include "ff/sql/mysql.hpp"
2  #include "ff/sql/table.h"
3
4  struct mymeta {
5    constexpr static const char *table_name = "yyy";
6  };
7
8  define_column(c1, column, uint64_t, "id");
9  define_column(c2, key, std::string, "event");
10 define_column(c3, index, uint64_t, "ts");
11
12 typedef ff::sql::default_table<mymeta, c1, c2, c3>
     mytable;
13
14 int main(int argc, char *argv[]) {
15   ff::sql::default_engine engine(CONNECTION, USER_NAME
       , PASSWORD, DB_NAME);
16   mytable::create_table(&engine);
```

[8]请参考 fflib 的源代码

```
17
18    mytable::row_collection_type rows;
19
20    mytable::row_collection_type::row_type t1, t2;
21    t1.set<c1, c2, c3>(1, "column1", 123435);
22    rows.push_back(std::move(t1));
23    t2.set<c1, c2, c3>(2, "column2", 1235);
24    rows.push_back(std::move(t2));
25
26    mytable::insert_or_replace_rows(&engine, rows);
27
28    auto ret1 = mytable::select<c1, c2, c3>(&engine).
        eval();
```

2.6.7 序列化为二进制

序列化为二进制在网络及文件 IO 中是常见的。fflib 中的 ffnet 定义了如何基于 `ntobject` 进行序列化的操作[9]。

```
1  template <uint32_t PackageID, typename... ARGS>
2  class ntpackage : public package, public ::ff::util::
     ntobject<ARGS...> {
3  public:
4    typedef typename util::type_list<ARGS...> type_list;
5    const static uint32_t package_id = PackageID;
6
7    ntpackage() : package(PackageID), ::ff::util::
       ntobject<ARGS...>() {}
8    template <typename OT>
```

[9]fflib/include/ff/net/middleware/ntpackage.h

```
 9   ntpackage(const OT &data): package(PackageID), ::ff
       ::util::ntobject<ARGS...>(data) {}
10
11   ntpackage<PackageID, ARGS...> make_copy() const {
12     ntpackage<PackageID, ARGS...> rt;
13     using base = ::ff::util::ntobject<ARGS...>;
14     *rt.base::m_content = *base::m_content;
15     return rt;
16   }
17
18   template <typename OT>
19   ntpackage<PackageID, ARGS...> &operator=(const OT &
       data) {
20     ::ff::util::ntobject<ARGS...>::operator=(data);
21     return *this;
22   }
23
24 protected:
25   virtual void archive(marshaler &ar) { archive_helper
       <0>::run(ar, *this); }
26
27   template <int Index> struct archive_helper {
28     template <typename VT>
29     static auto run(marshaler &ar, VT &val) ->
          typename std::enable_if<(VT::type_list::len >
          Index), void>::type {
30       ar.archive(std::get<Index>(*val.m_content));
31       archive_helper<Index + 1>::run(ar, val);
32     }
```

```
33
34      template <typename VT>
35      static auto run(marshaler &, VT &) -> typename std
           ::enable_if<(VT::type_list::len <= Index), void
           >::type {}
36    };
37  };
```

注意，ntpackage 引入了额外的参数，PackageID，用于标识一个二进制串的类型。例如，可以以如下方式定义一个可以序列化的 ntobject，ntpackage 的使用方法和 ntobject 几乎一样。

```
1     typedef ntpackage<1, name, id, gender, birthday>
         student_pkg;
2     student t;
3     t.set<name, gender, id>("xuepeng", male, "
         2023019847");
4     student_package tpkg, tpkg2;
5     tpkg = t;
6     tpkg2.set<name, gender, id>("xuepeng", male, "
         2023019847");
```

下面的代码展示了如何对 ntobject 进行序列化或反序列化[10]。

```
1  template <typename BytesType> struct make_bytes {
2    template <typename PackageType, typename... ARGS,
        typename... PARGS>
3    static BytesType for_package(PARGS... params) {
4      PackageType p;
5      p.template set<ARGS...>(params...);
```

[10]YeeZ-Privacy-Computing/include/ypc/corecommon/package.h

```
6       ff::net::marshaler lm(ff::net::marshaler::
            length_retriver);
7       p.arch(lm);
8
9       BytesType ret(lm.get_length());
10      ff::net::marshaler ld((char *)ret.data(), ret.size
            (), ff::net::marshaler::serializer);
11      p.arch(ld);
12      return ret;
13  }
14
15  template <typename ByteType, typename PackageType>
16  static void for_package(ByteType *data, uint32_t
        data_size,
17                          PackageType &pkg) {
18      ff::net::marshaler lm(ff::net::marshaler::
            length_retriver);
19      pkg.arch(lm);
20
21      BytesType ret(lm.get_length());
22      ff::net::marshaler ld((char *)data, data_size, ff
            ::net::marshaler::serializer);
23      pkg.arch(ld);
24  }
25  template <typename PackageType>
26  static BytesType for_package(const PackageType &_pkg
        ) {
27      auto &pkg = (PackageType &)_pkg;
28      ff::net::marshaler lm(ff::net::marshaler::
```

```
        length_retriver);
29      pkg.arch(lm);
30
31      BytesType ret(lm.get_length());
32      ff::net::marshaler ld((char *)ret.data(), ret.size
          (), ff::net::marshaler::serializer);
33      pkg.arch(ld);
34      return ret;
35    }
36  };
```

当需要序列化操作时，可以直接调用上述方法，如下代码所示

```
1       typedef ntpackage<1, name, id, gender, birthday>
          student_pkg;
2       student_package tpkg;
3       tpkg2.set<name, gender, id>("xuepeng", male, "
          2023019847");
4       ypc::bytes data = make_bytes<ypc::bytes>::
          for_package(tpkg);
```

同理，如下的代码展示了反序列化操作

```
1   template <typename PackageType> struct make_package {
2     template <typename BytesType>
3     static PackageType from_bytes(const BytesType &data)
        {
4       PackageType ret;
5       ::ff::net::marshaler ar((char *)data.data(), data.
          size(), ::ff::net::marshaler::deserializer);
6       ret.arch(ar);
7       return ret;
```

```
8   }
9   template <typename ByteType>
10  static PackageType from_bytes(const ByteType *data,
        uint32_t data_size) {
11    PackageType ret;
12    ::ff::net::marshaler ar((char *)data, data_size,
          ::ff::net::marshaler::deserializer);
13    ret.arch(ar);
14    return ret;
15  }
16 };
17
18 ypc::bytes data = make_bytes<ypc::bytes>::for_package(
     tpkg);
19 auto tpkg2 = make_package<student_pkg>::from_bytes(
     data);
```

2.6.8 序列化为 JSON

将 ntobject 序列化 JSON 字符串有利于直接阅读，并且有利于数据传输。

相关的代码在文件 YeeZ-Privacy-Computing/include/core/ntjson.h 中，此处仅说明如何使用。

```
1    student t;
2    t.set<name, gender, id>("xuepeng", male, "
       2023019847");
3    std::string str = ntjson::to_json(t);
4    ntjson::to_json_file(t, "file/path");
5    auto t1 = ntjson::from_json(str);
```

```
6       auto t2 = ntjson::from_json_file("file/path");
```

2.7 总结

本节主要介绍了 HPDA 中的 ntobject，由于 ntobject 中大量使用了编译期技术，因此本节也重点介绍了所使用到的编译期技术。

不难发现，ntobject 相关的工具仍然比较散落，尚未整理完全。因此，ntobject 的开发工作仍然有很多。

Chapter 3

计算图

在计算机科学中，计算图通常指一个用节点表示计算功能单元的有向无环图（DAG）。虽然并不一定直接使用 “计算图” 这一称谓，但 DAG 的使用颇为广泛[1]。例如，在 MySQL 中，执行计划（Execution Plan）用 DAG 表示[2]；在 Apache Spark DAG 中，DAG 表示了多个 Spark 任务直接的逻辑关系[3]；在机器学习中，则直接使用计算图描述各种神经网络[4]。

本文详细描述了在 HPDA 中的计算图，并且描述了如何以计算图的方式实现数据分析功能，并给出了一些例子。

3.1 什么是计算图

一个计算图是一个有向无环图（DAG），其中节点表示功能单元，又称为**算子**，边表示数据。自然地，只有出边的算子为**输入算子**，只有入边的算

[1]在某些场景中，使用的是有向有环图，例如在 Direct3D 中的 Graphics Pipeline（`https://learn.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline`）中；在 Intel 的 flow graph(`https://github.com/oneapi-src/oneTBB`) 中，则同时支持有向无环图和有向有环图。

[2]`https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html`

[3]`https://techvidvan.com/tutorials/apache-spark-dag-directed-acyclic-graph/`

[4]`https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/`

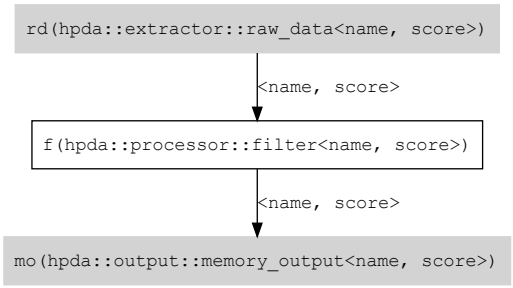

图 3.1: 一个简单的计算图。

子为**输出算子**，同时有出边和入边的算子为**处理算子**。

计算图描述了计算的逻辑。数据从输入算子产生，沿着边“流过”每个算子，直到输出算子。一个算子输出的数据是后继算子的输入。

如图 3.1所示。为了方便阅读，我们会在图中标识必要的信息。边上增加了数据类型，节点上增加了算子类型和名称，如果需要，也可以标注更多的信息。额外的，我们会用灰色标识输入算子和输出算子。每个算子都是独立的，不知道前趋算子或后继算子的信息。这使得计算图有着非常大的灵活性，可以任意组合。

3.2 以图的方式思考

计算图的处理逻辑并不直观，因此有必要为此着些笔墨。注意，计算图描述了计算的逻辑，并不描述实际的实现，同一个计算图，其实现方式、策略不同，实际的执行顺序也不同，因此，此处仅描述基本的逻辑。

如前所述，在计算图中，数据从输入算子产生，沿着边“流过”每个算子，直到输出算子。考虑到计算图所针对的数据条数通常很多，因此，输入算子并不是一次性产生所有数据，而是产生一条或一批数据。忽略实现的差异，我们说输入算子一次产生一条数据。这条数据沿着边“流过”每个算

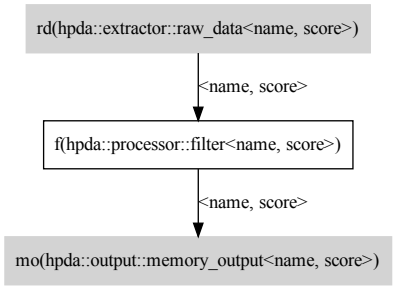

图 3.2: 一个简单的计算图。

子，每个算子会被执行多次，每次执行会对一条输入数据进行处理。当一条数据到达某个算子后，算子可能会生成新的数据，也可能不生成新的数据，还有可能生成多个数据。如果一个数据到达某个算子后，该算子不产生输出，我们就说，这个数据被该算子“消耗”掉了。一条数据最终总会被消耗掉：可能被中间的某个算子“消耗”掉，也可能到达输出算子从而被“消耗”掉。当一条数据被“消耗”掉后，输入算子继续产生新的数据，并重复上述过程，直到输入算子不能产生新的数据为止。

当以计算图的方式思考问题的时候，切记，数据是逐条流过计算图中的每个节点的，这和传统的编程方式十分不同：在传统的编程方式中，数据被看作一个集合，数据处理是从一个集合生成另一个集合的过程；而计算图中则是将数据逐条变换。以图3.2（该图也出现在第1章中）为例，图中 `f` 的条件为分数超过 60 分。在传统的编程方式中，会将所有学生的信息读取到内存（例如数组）中，然后将超过 60 分的信息转存到一个新的数组中，并输出最终的结果；而在计算图中，每个学生的信息逐条流过算子 `f`，算子 `f` 根据条件消耗掉不满足条件的数据，也就是当数据不满足条件时，该算子不产生输出数据，最终 `f` 输出的数据均为超过 60 分的信息。

那么为什么要采用这种看起来有些“繁琐”的方式来处理数据或思考问题呢？根本原因在于**内存受限，不能放下整个数据集**。对于前述的例子，当学生的信息只有 10MB 时，在内存中处理是方便的，然而当面临的数据为

几十 GB，甚至几十 TB 时，试图在内存中对集合进行处理是十分低效的。在 HPDA 面临的场景（如 Intel SGX），内存的限制则更为严苛，只有百兆内存可以使用。使用计算图的方式，十分有利于处理大规模的数据集。

在某些算法中，需要同时处理多条数据，例如排序、求最大值等，这似乎与逐条流过算子的思考方式是冲突的，这里也是计算图最不直观的地方：算子可以接收数据并*缓存*到内部，直到条件合适时再对内部缓存的多个数据进行处理，处理完毕后，再输出数据到后继算子。

3.2.1 推送（push）还是拉取（pull）?

在上面的描述中，你也许会产生一种错觉：算子不停产生数据，并*推送*到后继的算子中。实际上，算子是被动产生数据，并从前趋算子*拉取*数据。这两者是有区别的：如果是推送方式，算子不一定需要相应的数据，因此算子将不得不缓存接收到的数据，而使用拉取方式，算子会根据自身的需要，从前驱算子中获取数据，从而避免缓存数据。

虽然推送或者拉取更多是调度器的实现细节，但是理解这一点对于如何构造计算图是重要的。下面，我们通过两个例子来说这两者的区别，以及计算图中的算子是如何拉取数据的。

例子 1：多个输入算子 如图3.3所示，图中有两个输入算子，`input1` 和 `input2`，算子 `f` 的类型是 `concat`，其功能是直接输出输入数据，当第一个输入算子不再产生数据后，转而从第二个输入算子获取数据。

如果是推送方式，输入算子 `input2` 会在 `input1` 产生一条数据后，产生数据，而且此时处理算子 `fc` 也不接收 `input2` 产生的数据；而如果是拉取方式，输入算子 `input2` 会在输入算子 `input1` 不再产生数据时产生数据。由此可以明显的看出推送方式和拉取方式的区别。

例子 2：有多个输出的处理算子 在第1章中的图1.3，曾经提到过类型为 `split` 的算子，其产生两个输出，一个简化的例子如图3.4所示。`split` 算子接受一个数据，同时在两个输出边上产生两个数据。按照拉取的方式执行，

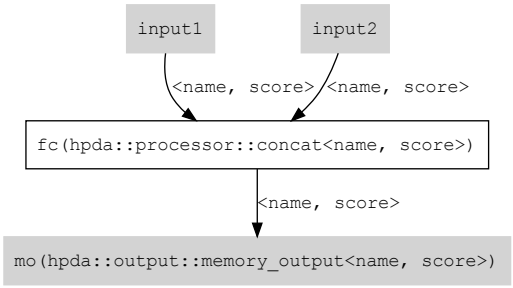

图 3.3: 一个有多个输入算子的计算图。

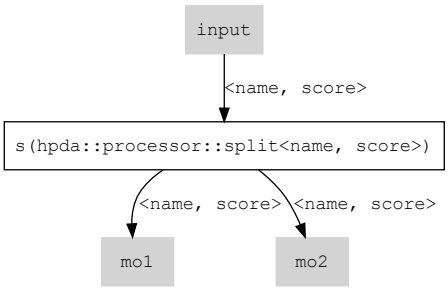

图 3.4: 一个有多个输出的处理算子的计算图。

则 mo1 拉取到全部的数据的同时，s 会在另外的输出边上，产生全部的数据，直到 mo2 拉取完全部数据。虽然 mo1 和 mo2 分别拉取到了全部的数据（一共拉取到了两份数据），但 input 只产生了一份数据。

显然，在 s 中缓存所有数据是不现实的，因此，实现中会调整拉取的顺序，使得 mo1 和 mo2 轮流拉取数据，使得 s 中数据总是不需要缓存。这种实现并不影响上文的逻辑，属于实现相关。

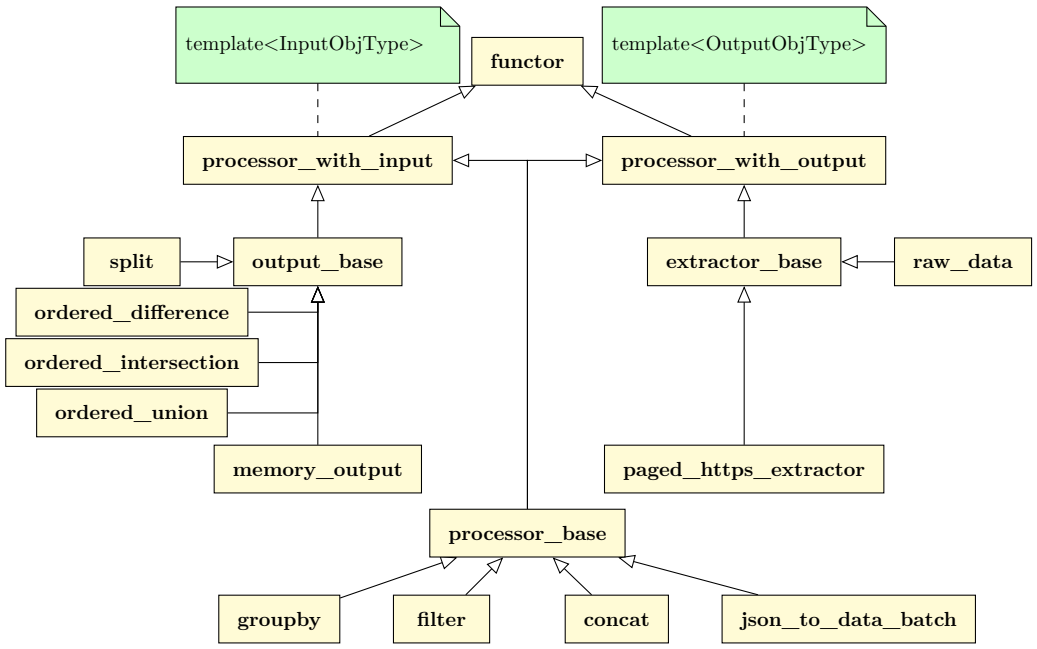

图 3.5: HPDA 中各种算子的继承关系。

3.3 HPDA 中的图

在 HPDA 中，不同的概念对应不同的类，并通过继承这些类来实现不同的功能。图 3.5所示为目前 HPDA 中的算子及继承关系。可以看到，HPDA 中涉及的算子还不够丰富，因此，HPDA 仍然在活跃的开发中。

前 qu 在 HPDA 中，一个算子被描述为一个 `functor`，其中包含了算子的前趋算子列表、执行引擎的指针和状态信息，如下代码所示。

```
1 class functor {
2 public:
3   functor();
4   virtual ~functor();
5   virtual bool process() = 0;
6   inline void reset_done_value() { m_has_value = false
      ; }
```

```
7    virtual void done_value();
8    inline bool has_value() const { return m_has_value;
       }
9    inline engine *get_engine() const { return m_engine;
       }
10
11   void set_engine(engine *e);
12
13   inline void add_predecessor(functor *pred) {
       m_predecessors.push_back(pred); }
14
15   inline const std::vector<functor *> predecessors()
       const {
16     return m_predecessors;
17   }
18
19 protected:
20   bool m_has_value;
21   std::vector<functor *> m_predecessors;
22   engine *m_engine;
23
24   friend class engine;
25   uint32_t m_status;
26 };
```

对于一个算子而言,最重要的状态是是否有可以输出的数据,即 `m_has_value`,当为 `true` 时表示算子中存在一个可以输出的数据，反之则没有可以输出的数据。

算子中还包括了输入、输出的类型信息，这些使用 C++ 中的模板参数表示。更具体的，输入、输出的类型为一个 `ntobject`。下面的代码描述了

一个有输入的算子 processor_with_input:

```
1  namespace internal {
2  template <typename InputObjType>
3  class processor_with_input : virtual public functor {
4  public:
5    typedef InputObjType input_type;
6
7    processor_with_input(processor_with_output<
       InputObjType> *input)
8        : functor(), m_upper_stream(input) {
9      functor::set_engine(input->get_engine());
10     add_predecessor(input);
11   };
12
13   virtual ~processor_with_input() {}
14
15   InputObjType input_value() const { return
       m_upper_stream->output_value(); }
16
17   inline bool has_input_value() const { return
       m_upper_stream->has_value(); }
18
19   inline void consume_input_value() { m_upper_stream->
       reset_done_value(); }
20
21   inline void change_upper_stream(
       processor_with_output<InputObjType> *input) {
22     m_upper_stream = input;
23   }
```

```
24
25 protected:
26   internal::processor_with_output<InputObjType> *
       m_upper_stream;
27 };
28 } // namespace internal
29 template <typename... ARGS>
30 using processor_with_input = internal::
     processor_with_input<ntobject<ARGS...>>;
```

类似的,下面的代码定义了一个有输出的算子 processor_with_output:

```
1  namespace internal {
2  template <typename OutputObjType>
3  class processor_with_output : virtual public functor {
4  public:
5    typedef OutputObjType output_type;
6    virtual ~processor_with_output() {}
7    virtual OutputObjType output_value() = 0;
8  };
9  } // namespace internal
10 template <typename... ARGS>
11 using processor_with_output = internal::
     processor_with_output<ntobject<ARGS...>>;
```

显然的,一个用于输出数据的算子 output_base 继承自 processor_with_input;一个用于产生数据的算子 extractor_base 继承自 processor_with_output;而一个处理算子则同时继承自 processor_with_input 和 processor_with_output。考虑到处理算子同时继承两者,存在棱形继承,因此在继承时使用了 virtual public。

3.3.1 HPDA 中的算子

下面罗列了 HPDA 中一些算子的原型，并说明了其功能。注意，为了隐藏实现，此处罗列的原型与实际代码有出入，仅用于说明如何使用。

- `memory_output`：表示将数据输出到内存中，其原型为

```
1 template <typename... ARGS>using memory_output =
    internal::memory_output_impl<ntobject<ARGS
    ...>>;
2 template <typename T> using memory_output_t =
    internal::memory_output_impl<T>;
```

  其中的 `T` 为需要输出到内存中的数据的类型。

- `split`：表示将一条输入数据拆分为多条，拆分的数量取决与自己有多少个后继算子，其原型为

```
1 template <typename... ARGS> using split =
    internal::split_impl<ntobject<ARGS...>>;
```

- `filter`：过滤器，仅输出满足条件的数据，其原型为

```
1 template <typename... ARGS> using filter =
    internal::filter_impl<ntobject<ARGS...>>;
2 template <typename T> using filter_t = internal::
    filter_impl<T>;
```

  `filter` 的输出类型和输入类型相同。

- `concat`：将多个相同类型的输出拼接到一起，依次输出相应的数据，原型为

```
1 template <typename... ARGS> using concat =
    internal::concat_impl<ntobject<ARGS...>>;
```

concat 中各个输入的类型必须相同，输出的类型和输入相同。

- groupby：对输入数据进行分组处理，原型为

```
1 template <typename InputObjType, typename
    GroupByType, typename AggregatorType, typename
    ... ARGS>
2 using groupby = internal::groupby_impl<
    InputObjType, ntobject<ARGS...>, GroupByType,
    AggregatorType>;
3
4 template <typename InputObjType, typename
    OutputObjType, typename GroupByType, typename
    AggregatorType>
5 using groupby_t = internal::groupby_impl<
    InputObjType, OutputObjType, GroupByType,
    AggregatorType>;
```

其中 InputObjType 为输入的数据类型，OutputObjType 为输出的数据类型，GroupByType 为进行分组的列的类型（ntobject 中的数据项），AggregatorType 为对分组进行操作的类型，有以下几种类型可选 sum，avg，max，min，count，对应的原型如下

```
1 namespace group {
2     template <typename InputObjType, typename
        OutputObjType> using sum = internal::sum<
        InputObjType, OutputObjType>;
3     template <typename InputObjType, typename
        OutputObjType> using avg = internal::avg<
        InputObjType, OutputObjType>;
4     template <typename InputObjType, typename
        OutputObjType> using max = internal::max<
```

```
        InputObjType, OutputObjType>;
5     template <typename InputObjType, typename
        OutputObjType> using min = internal::min<
        InputObjType, OutputObjType>;
6     template <typename InputObjType, typename
        OutputObjType> using count = internal::
        count<InputObjType, OutputObjType>;
7 }
```

- `json_to_data_batch`: 用于将一个 JSON 字符串转换为 `ntobject` 并输出，其原型为

```
1 template <typename... ARGS> using
    json_to_data_batch = internal::
    json_to_data_batch_impl<ntobject<ARGS...>>;
```

  注意，此处仅定义了输出类型，输入类型为 `std::string`。

- `raw_data`: 用于将内存中的数据逐条输出，其原型为

```
1 template <typename... ARGS> using raw_data =
    internal::raw_data_impl<ntobject<ARGS...>>;
```

3.3.2 静态计算图

HPDA 中的计算图是静态的，这有一些限制，比如不能动态的调整图的结构。但也有一些额外的好处，例如，获得类型检查的能力，还可以通过静态检查发现更多的错误或优化机会。

3.4 一个例子

本节会给出一个具体的例子[5]，不但说明如何定制一个具体的算子，如何实现相应的计算图，也说明如何利用网络完成分布式的计算。

假设现有大量电话号码在过去一年因位置移动导致基站变更的经纬度和时间戳数据，希望计算得知过去一年内旅行距离最长的 n 个电话号码。（假设旅行距离为时间相邻的两条数据的坐标的直线距离）原始数据以「电话号码、经度、纬度、时间戳」（`phone_number`, `longitude`, `latitude`, `timestamp`）的形式保存在一个文件中。由于不同基站数据提取有先后，时间戳先后次序没有保证，同一电话号码的数据会分布在原始文件中的不同位置。

在本节中，**节点**（*node*）指独立的运行 hpda 的进程，**算子**（`processor`）指 hpda 中对计算的抽象。

3.4.1 计算图

假设原始数据文件超过了内存容量，计算图使用了如图 3.6所示的设计：*splitter* 节点流式的读取原始文件、将原始数据根据电话号码的哈希值通过网络发送到任意数量的 *worker* 节点；*Worker* 节点对 `phone_number` 进行并归、对 `timestamp` 进行排序、计算距离之和、取最大的 n 个值后，通过网络传输出给 *aggregator* 节点；*aggregator* 收到每个 *worker* 节点最大的 n 个数据后，再取最大的 n 个输出。

3.4.2 流式算子和截流算子

本章伊始曾提到，数据是沿边流动。对于许多算子，例如 `calculate distance`、`concat`，数据可以流入算子后立刻流出。这类算子可以称为**流式算子**（`pipeline_processor`）。他们的 `process` 函数逻辑非常直观：读取一个 `input_value`，进行计算，使得 `output_value()` 返回处理后的输

[5]代码见：https://github.com/weixiao-zhan/understanding-hpda-code.git

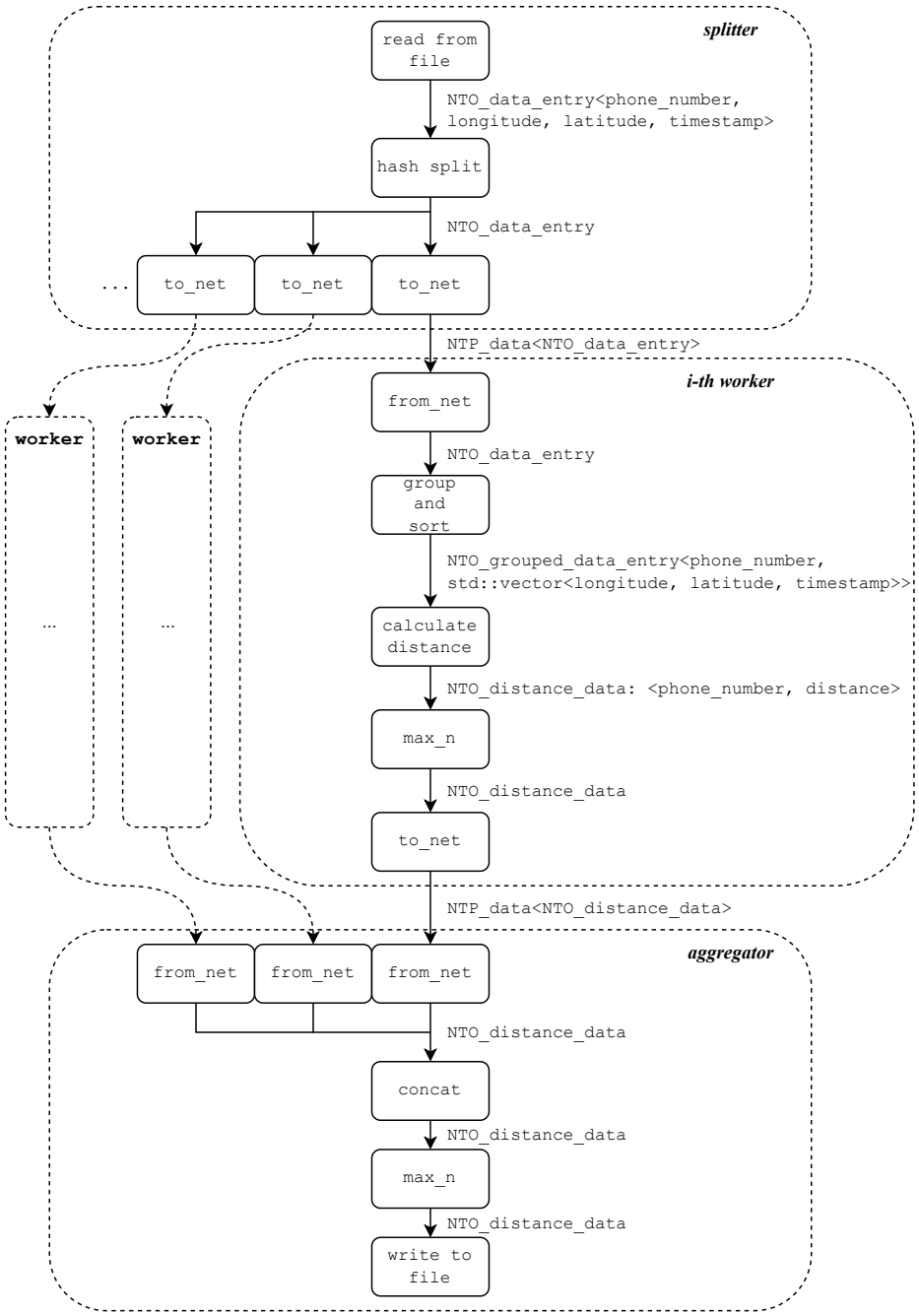
splitter
read from file
NTO_data_entry<phone_number, longitude, latitude, timestamp>
hash split
NTO_data_entry
...
to_net
to_net
to_net
NTP_data<NTO_data_entry>
worker
worker
...
...
i-th worker
from_net
NTO_data_entry
group and sort
NTO_grouped_data_entry<phone_number, std::vector<longitude, latitude, timestamp>>
calculate distance
NTO_distance_data: <phone_number, distance>
max_n
NTO_distance_data
to_net
NTP_data<NTO_distance_data>
aggregator
from_net
from_net
from_net
NTO_distance_data
concat
NTO_distance_data
max_n
NTO_distance_data
write to file

图 3.6: 计算图架构.

出（算法1的范式）。它们只需要保存一份输出数据，以便后续算子多次读取。当需要处理新的输入时，可以立刻丢弃先前缓存的数据。

然而还有更多的算子依赖于获得所有的输入，即取得全局信息后，才能进行计算和输出，例如 `group`、`sort`、`max_n`。这类算子可以称为*截流算子*（`pipeline_breaker_processor`）。数据流会被这些算子截断并积累在算子内，只有当截流算子所有的上游节点都没有输出时，这些算子才会开始输出。

这些算子的 `process()` 函数通常有前后两个阶段。第一阶段：流入数据，算子将输入的数据保存到数据结构中。当没有更多输入数据时，即 `has_input_value() == false`，进入第二阶段：流出数据。范式如算法2所示。注意，并不一定要全部数据流入完成才能开始计算，可以在流入数据的同时使用适合的数据结构，优化计算。例如[6]：`group` 可以使用哈希表、`sort` 可以使用堆、`max_n` 使用固定长度的链表。

第**??**章将详细介绍调度器是如何使流式算子和截流算子协同工作的。

Algorithm 1 `pipeline_processor.process()`

```
1: procedure PROCESS()
2:     if has_input_value() then
3:         do_some_process();                    ▷ user defined
4:         do_out_put_update();                  ▷ user defined
5:         consume_input_value();
6:         return true;
7:     end if
8:     return false;
9: end procedure
```

[6]这里提到的优化数据结构在本例代码中没有全部实现

Algorithm 2 `pipeline_breaker_processor.process()`

```
    procedure PROCESS
 2:     if has_input_value() then
            store_input_value_to_some_data_structure();   ▷ user defined
 4:         consume_input_value();
            return false;
 6:     end if
                                                  ▷ @here drained input
 8:                                                  ▷ end of phase 1
        if not processed then
10:         do_some_process();                          ▷ user defined
        end if
12:                                                 ▷ starting phase 2
        do_out_put_update();                            ▷ user defined
14:     return true;
    end procedure
```

3.4.3 分裂与聚合

在计算图模型中，分裂（split）和聚合（aggregate）是两种重要的操作。分裂操作用于将复杂的计算任务划分为可以并行处理的小任务，从而利用多核、多节点的计算能力。聚合操作则使我们可以在各个子任务完成计算之后，将其结果集中起来，形成一个统一的输出。分裂和聚合可以更好地实现计算任务的并行化、提高计算效率，在处理大规模数据处理和分布式计算场景中起着关键作用。这两种操作的组合排列也使计算图具有更强的灵活性和扩展性，可以适应各种复杂的计算需求。

分裂

前文曾提到计算图以拉取（pulling）的方式使数据流动时算子可以不缓存数据（因算法需求的截流算子除外）。但是分裂算子有其特殊性：分裂算子拉取到的数据并不一定属于正在拉取数据的下游算子。考虑图3.4中的例子，假设 `split` 算子将根据 `name` 的哈希值分配给 `mo1` 和 `mo2`。当 `mo1` 拉取数据时，split拉取到的的下一条数据，甚至下 n 条，乃至之后所有数据都可能被哈希到 `mo2`，而非正在拉取数据的 `mo1`。

一种解决方案是允许 `split` 推送（push）数据给 `mo2`。这就要求算子提供推送数据的方法。我们可以在 `split` 算子和输出算子之间增加一个队列算子来避免重复更改下游算子的实现。实际上 `hpda::processor::split` 的实现如图3.4.3所示：它会为每一个下游算子生成一个允许推送操作即允许 `add_data()` 的队列算子 [7]。从而使普通的算子也可以无缝地从 `split` 算子中拉取数据。

当下游算子 (`mo2`) 拉取数据时，队列算子优先出队，当队列为空时，则尝试从 `split` 算子拉取数据。`split` 算子从上游拉取数据时，可以在取得一条数据并推送到相应队列后立刻返回，也可以拉取多条数据直到取得一条能推送到正在拉取数据的下游算子的数据后再返回。虽然调度器会保证两者结果是相同的，但是前者能相对节约内存开支，后者能节约调度的计

[7]撰写本节时，`split` 的队列算子使用 `memory_output` 实现，应当优化为队列

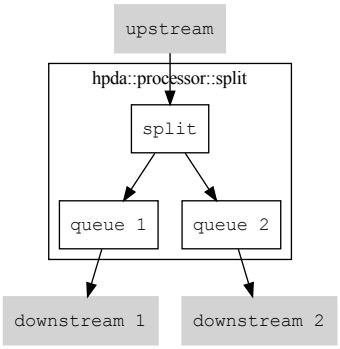

图 3.7: hpda::processor::split 的一种实现

算开支。

聚合

聚合算子比分裂算子简单得多。只需遍历输入算子，获得一条数据后即可返回。注意，可以采取不同的策略遍历输入算子：一种策略是首先读取一个算子的所有输出，然后再继续到下一个输入算子；另一种可能策略是尝试从每个算子那里获取一个输入，然后继续这个过程直到所有的输入算子都没有更多的输出可供获取。

3.4.4 流式网络传输算子

当把计算图部署到多个物理计算节点以实现分布式计算时，不可避免的要通过网络发送和接收数据。这一小节将对本例中使用到的网络传输做相关说明。

序列化

对于复合类型或嵌套定义的 ntobject，如果要将其通过网络发送，则需要开发者定义其序列化和反序列化的方法。在本例中 *splitter* 向 *worker* 发送 NTO_data_entry，*worker* 向 *aggregator* 发送 NTO_distance_entry。所以在 doc_example/3.4/common.h [8] 中定义了二者的序列化和反序列化方法。

fflib::net

fflib::net 提供了通过网络发送和接收 ntpackage 的支持。

ntpackage 的定义如下:

```
1 template <uint32_t PackageID, typename... ARGS>
2 class ntpackage : public package, public ::ff::util::
    ntobject<ARGS...>
```

ntpackage 有着和 ntobject 相似的读写方法，其中第一个数据项需定义为唯一的 PackageID。fflib::net 正是基于 PackageID 数据项调用相应的 package handler 函数。

使用 fflib::net 主要涉及以下几个基本组件:

- net_nervure 是基于网络事件的响应引擎，在连接成功，连接失败，和收到包裹时调用初始化时绑定的回调函数
- typed_pkg_hub 能解析包裹，调取适当的回调函数
- tcp_connection_base 是对 tcp 连接的抽象

关于 fflib::net 更详细的信息可以参考 源代码 [9] 和这个 示例 [10]

[8] https://github.com/weixiao-zhan/understanding-hpda-code/blob/main/doc_example/3.4/common.h

[9] https://github.com/weixiao-zhan/understanding-hpda-code/tree/main/fflib/src/net

[10] https://github.com/weixiao-zhan/understanding-hpda-code/tree/main/fflib/example/net/pingpong

netio

本例基于 `fflib::net` 提供了一对实现了网络单向传输 `ntobject` 的算子：`to_net` 和 `from_net`（统称为 `netio` [11]）。数据流入 `to_net` 算子，经网络传输后，从另一节点的 `from_net` 流出。使用这一对算子，本例的计算图可以部署到多个节点上，从而解决内存限制。

图3.8展示了 `to_net` 和 `from_net` 算子传输数据中一些关键事件的先后次序。其中黑色虚线代表控制流，蓝色实线代表数据流，绿色短线代表网络事件。

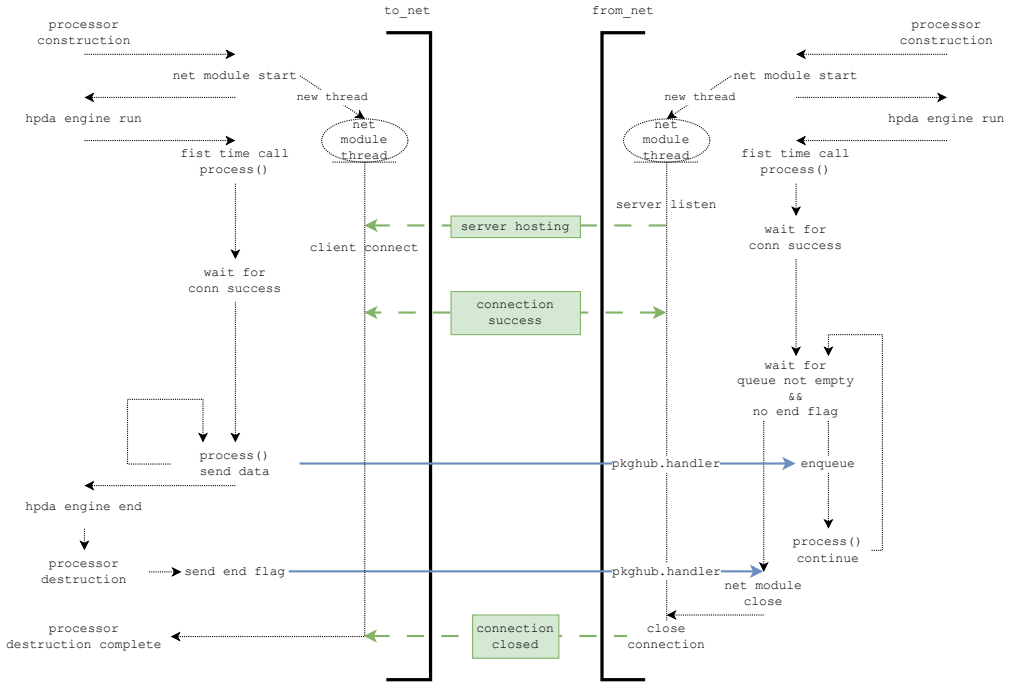

图 3.8: `to_net` 和 `from_net` 算子

[11] https://github.com/weixiao-zhan/understanding-hpda-code/blob/main/doc_example/3.4/netio.h

to_net

- 构造
  因为 fflib::net::net_nervure 会阻塞当前线程，所以在算子构造时会创建另一个线程来运行 net_nervure（称为 net_module_thread）。而主线程则返回 caller，并在之后进入 hpda::engine.run()。

- process
  每当 to_net 有可以处理的数据时，它先等待网络连接建立成功，将数据发送后即可丢弃。伪代码如下算法3所示。

Algorithm 3 to_net.process()

```
    procedure PROCESS
        wait for connection setup
3:      if has_input_value() then
            send data
            consume_input_value();
6:          return true;
        end if
        return false;
9:  end procedure
```

- 析构
  to_net 在析构时会发送一个 end flag 用于通知 from_net 已经发送完所有数据。待网络连接由服务器关闭后，析构函数返回。

  注意:to_net 在析构时也可以选择直接关闭网络连接而非等待 from_net 的服务器来关闭连接，但是这可能导致最后几个数据包被 from_net 所在节点的操作系统网络栈丢弃，所以选择等待服务器关闭连接。

from_net

- 构造
  from_net 的构造与 to_net 的构造完全相同。注意:开发者运行 netio 算子对时，需要确保 from_net 的 server 早于 to_net 的 client 绑定到端口，以便连接建立成功。即 from_net 的构造早于 to_net 的构造。

- process
  由于是推送数据，from_net 需要有一个队列保存接收到但还没处理的数据。考虑到入队和出队是由不同线程操作的，这里需要使用线程安全的队列实现。每当 from_net 的队列中没有可以处理的数据时，它先阻塞等待而非返回 false；否则调度器可能误认为已经没有更多的数据而导致计算错误。当且仅当收到 end flag 且队列为空时 process 才能返回 false。伪代码如算法4所示。

- 析构 from_net 析构不需要额外等待，释放动态分配的内存后即可返回。

3.4.5 节点启动

跨节点运行计算图的一个关键是确保各节点的 from_net.server 先于 to_net.client 绑定到端口。对于节点之间无环的简单计算图（如本例），一个简单的解决方案是确保下游节点的 hpda 进程先启动，其主线程会阻塞在第一个的 from_netio 的第一次 process 调用中等待连接建立，此时再启动他的上游节点，依此类推。

具体来说则是先启动 *aggregator* 节点，待其阻塞后，再启动 *worker* 节点。如果有多个 *worker* 节点，则他们可以以任意顺序启动。此时 *aggregator* 会提示已经与 *worker* 节点连接建立成功。*aggregator* 因为队列为空继续阻塞，*worker* 节点因为连接失败同样阻塞。最后再启动 *splitter* 节点，此时数据将通过多个调度器和网络流过计算图，完成所有的计算。

Algorithm 4 `from_net.process()`

```
    procedure PROCESS
        wait for connection setup
3:      wait for queue not empty AND not receive end flag
        if received end flag AND net module is running then
            close connection and stop net module        ▷ this should close the
    connection and allow to_net destructor to finish and return
6:      end if
        if received end flag AND queue is empty then
                ▷ there is no more data coming coming from net and all cached
    data has been consumed
9:          return false;
        end if
        dequeue
12:     return true;
    end procedure
```

虽然计算图不允许环的存在，但是节点之间仍可能有环。本例中因为 *worker* 节点的计算是截流的，所以 *splitter* 和 *aggregator* 的算子可以放到同一个节点的同一个调度器中节约资源。此时简单的节点启动顺序不足以确保连接建立成功，需要有其他的协调机制辅助建立网络连接。

3.5 总结

本节面向开发人员描述了 HPDA 中的计算图，说明了如何用计算图的方式思考问题，以及 HPDA 中对于计算图的实现。

最后，本节给出了一个有些复杂的例子，说明了 HPDA 的能力。作为开发人员，也可以从这个例子中获取一些灵感。

Chapter 4

调度器

调度器是驱动数据流过计算图的关键组件。它在运行时管理和控制计算图各算子的执行顺序，并确保数据按照预定的逻辑顺序（即计算图）被处理。调度器在确定算子的执行顺序时通常基于特定的调度策略，以满足数据依赖性、算子优先级、系统负载等需求。在 HPDA 中，调度器被抽象为 `hpda::engine`（也称调度引擎，hpda 引擎）。

由于调度器的复杂性，本章使用更加形式化的描述方式。首先给出计算图的形式化描述，然后给出调度器的形式化描述，并给出了 HPDA 中调度器的实现。

4.1 重新定义计算图

如前所述，计算图是一个有向无环图 (DAG)，形式化的，一个计算图 G 表示为

$$G = \{F, E\}$$

其中 F 为算子的集合, E 为边 (数据) 的集合, $\forall(f_1, f_2) \in E : f_1 \in F, f_2 \in F$。对于边 $(a, b) \in E$，意为 a 的输出是 b 的输入，也就是说 a 是 b 的前趋节点，b 是 a 的后继节点。

对于一个算子 $f \in F$，其前趋节点的集合表示为

$$\mathrm{Pred}(f) = \{i | \exists (i, f) \in E\},$$

类似的，后继节点的集合，表示为

$$\mathrm{Succ}(f) = \{i | \exists (f, i) \in E\}$$

如果一个算子的前趋节点集合为空集，我们称之为输入算子，输入算在的集合记为 $I, I \neq \varnothing$；如果一个算子的后继节点集合为空集，我们称之为输出算子，输出算子的集合记为 $O, O \neq \varnothing$。

4.2 调度器

4.2.1 算子的状态

一个算子 $f \in F$ 的状态被定义为

$$\begin{aligned} S^f &= \{h | h \in \{\mathrm{HasValue}, \mathrm{NotHasValue}\}\} \cup K(f) \\ K(f) &= \{S^p.h | p \in \mathrm{Pred}(f)\} \end{aligned} \tag{4.1}$$

其中 h 表示当前算子是否有值可以输出，$K(f)$ 表示当前算子的前趋节点是否有值可以输出，或者说当前的算子的输入是否都准备就绪。

如果一个算子 f 为输入算子，则 $K(f)$ 也为空集，h 的值取决于输入的数据是否读取完成；若 f 为输出算子，则 $S^f.h = \mathrm{NotHasValue}$。

4.2.2 状态转移函数

算子存在一个 `process` 函数，称为算子的状态转移函数。该函数的实际功能是处理数据，但调度器并不关心数据处理的细节，仅关心处理数据前后的状态变化。对于算子 f，在某个时刻 i 的状态为 S_i^f，调用 `process` 后，状态转变为 S_{i+1}^f，记为

$$S_i^f \xrightarrow{f.\mathtt{process}} S_{i+1}^f$$

算子 f 发生状态转移意味着三种情况中的一种

1. 消耗掉一个多或多个输入，产生输出：

$$\begin{aligned} S_{i+1}^{f}.h = \text{HasValue} | S_{i}^{f}.h = \text{HasValue} \\ \exists p \in \text{Pred}(f) : S_{i+1}^{p}.h = \text{NotHasValue} | S_{i}^{p}.h = \text{HasValue} \end{aligned} \tag{4.2}$$

2. 不消耗输入，产生输出：

$$\begin{aligned} S_{i+1}^{f}.h = \text{HasValue} \\ \forall p \in \text{Pred}(f) : S_{i+1}^{p}.h = S_{i}^{p}.h \end{aligned} \tag{4.3}$$

3. 消耗一个或多个输入，不产生输出：

$$\begin{aligned} S_{i+1}^{f}.h = \text{NotHasValue} \\ \exists p \in \text{Pred}(f) : S_{i+1}^{p}.h = \text{NotHasValue} | S_{i}^{p}.h = \text{HasValue} \end{aligned} \tag{4.4}$$

注意，总结上述三种情况，并不能简单的总结为 $S_{i+1}^{f} \neq S_{i}^{f}$，在情况 2 中，是可能存在 $S_{i+1}^{f} = S_{i}^{f}$ 的情况的。

4.2.3 计算图的状态及转移

计算图 $G = \{F, E\}$ 在时刻 i 的状态表示为所有算子在时刻 i 的状态，即

$$S_{i}^{G} = \{S_{i}^{f}.h | f \in F\}$$

注意，为了避免冗余，此处不再包含每个算子的前趋算子的状态。类似的，计算图的状态转移记为

$$S_{i}^{G} \xrightarrow{x.\texttt{process}, x \in F} S_{i+1}^{G}$$

也可以简单的记为

$$S_{i}^{G} \xrightarrow{x} S_{i+1}^{G}$$

即选取了某个算子 x 执行后的状态，可知，对于其他算子 $t \in F, t \neq x$，有 $S_{i+1}^{t} = S_{i}^{t}$。

4.2.4 计算图的初始状态和终止状态

在最开始时，计算图中只有输入算子能够产生输出数据，因此，初始状态记为

$$S_0^G = \{S^a.h = \text{HasValue} | a \in I, S^a.h = \text{NotHasValue} | a \in (F - I)\}$$

在终止时，计算图中所有算子都不能再产生数据，因此，终止状态记为

$$S_T^G = \{S^a.h = \text{NotHasValue} | a \in F\}$$

4.2.5 计算图调度问题

至此，我们可以形式化的定义调度问题：对计算图 G，寻求一个算子的序列 Q，依次执行序列中每个算子的转移函数，使得计算图从初始状态 S_0^G 转移到终止状态 S_T^G。

最优调度 对于一个任务图，可能存在多种有效的序列，都能从初始状态转移到终止状态，即存在多种有效的调度策略。然而，不同的调度策略通常涉及更多的优化目标，例如，最小的内存占用、最快的执行时间等。通常，需要找到一个优化目标下的最优调度。

4.2.6 一个例子

4.3 调度器的实现

HPDA 中的调度器的优化目标时占用尽可能少的内存。

4.3.1 算子的状态：局部视角

熟悉代码的读者对`processor_base`的以下两个方法一定不会感到陌生。

- `bool process()`

- `void consume_input_value()`

如果参考 `hpda::engine` 中所有 `process()` 调用的上下文:

```
1 bool v = f->process();
2 if (v) {
3     f->done_value();
4 } else {
5     f->reset_done_value();
6 }
```

以及 `functor::reset_done_value()`, `functor::done_value()`, `processor_with_input::consume_input_value()` 的定义:

```
1 void functor::reset_done_value() { m_has_value = false
    ; }
2 void functor::done_value() { m_has_value = true; }
3 void processor_with_input::consume_input_value() {
    m_upper_stream->reset_done_value(); }
```

不难看出 `hpda::engine` 调度的仅依赖 `functor.m_has_value` 域。在 `hpda::engine` 看来，每个算子总是处于*有数据*（`functor.m_has_value == true`）或 *无数据*（`functor.m_has_value == false`）的二元状态。`functor.process` 则提供了 `hpda::engine` 唤醒算子，执行算子设定的逻辑，并进行必要更新的方法。

聚焦到图的局部结构：考虑 `hpda::engine` 正在处理的当前算子和该算子的所有直接前驱算子的有无数据状态。[1] `hpda::engine` 会假设大部分算子只会在所有直接前驱算子均有数据时才能产生输出，即变为有数据的状态。所以 `hpda::engine` 会*尽力*（best effort）让当前算子的所有直接前驱算子均处于有数据状态后，才会调用当前算子的 `functor.process`。

注意，即使所有直接前驱算子（direct predecessor）都有数据，调用当前算子的 `functor.process` 并不一定能使当前算子变为有数据，也不一定会

[1]"本算子和本算子的所有直接前驱算子的有无数据状态"同样也是当前算子的感受野。

使一部分或所有直接前驱算子变为无数据的状态[2]。反之，即使所有的直接前驱算子都处于无数据的状态，当前算子也可能在调用 functor.process 后变成或维持有数据的状态。这里的例外主要是截流算子。hpda::engine 在尽力让所有直接前驱算子均处于有数据状态但未能满足时，仍会尝试调用当前算子的 functor.process。所以只要截流算子的实现是正确时，它们在计算图中的计算结果依然是正确。

还需要指出,其他几个常见的算子方法:如 input_value(),has_input_value(), output_value() 均为无状态(stateless)调用。在没有发生其他变化时,两次相邻的无状态调用应返回相同的值,且不应改变算子内部 functor.m_has_value 的值。

4.3.2 全局终止条件

在开始和运行时，一些算子可能处于有数据状态，而另一些处于无数据状态。但当计算已经完成，即所有数据已经流过计算图的标志时，所算子均处于无数据状态。这也是 hpda::engine 的终止条件。

4.3.3 调度

把局部视角下对算子的调度和全局终止条件相结合,便有了 hpda::engine 的调度算法[3]。

使用一个 to_process 队列从输出算子开始广度优点搜索图，把所有无数据的算子加入一个 processing 栈。当遍历完成时[4]，如若

1. 所有算子在直接前驱算子有数据时调用 functor.process 均会变成有数据

[2]hpda::engine 对于直接前驱算子在 functor.process 之后是否会变成无数据则没有假设。

[3]当前调度算法的实现不一定是内存占用的最有解

[4]此时 to_process 队列为空

2. 节点之间没有数据竞争（图为树或森林）

则沿出栈顺序调用栈内算子后，所有 `functor.process` 均应该返回 true，并成功把一条数据拉取到输出算子。但是由于截流算子（假设一不成立）和算子间的依赖关系导致的数据竞争（假设二不成立），出栈过程中可能会遇到 `functor.process` 返回 `false` 的算子。此时 `hpda::engine` 只需在以该节点为输出节点的子图上再次执行本操作即可。

具体实现中，对于最后一步的递归调用，因为此时 `to_process` 队列已经为空、`processing` 栈的先入后出（LIFO），可以优化为使用同一组队列和栈，化递归为循环。

Chapter 5

使用 HPDA 开发

C++, SQL

Chapter 6

并行