Class Operator

Table of contents

- Defined in File operator.hpp

Inheritance Relationships

Base Type

- public holoscan::ComponentBase (Class ComponentBase)

Derived Types

- public holoscan::ops::AJASourceOp (Class AJASourceOp)
- public holoscan::ops::AsyncPingRxOp (Class AsyncPingRxOp)
- public holoscan::ops::AsyncPingTxOp (Class AsyncPingTxOp)
- public holoscan::ops::BayerDemosaicOp (Class BayerDemosaicOp)
- public holoscan::ops::FormatConverterOp (Class FormatConverterOp)
- public holoscan::ops::ForwardOp (Class ForwardOp)
- public holoscan::ops::GXFOperator (Class GXFOperator)
- public holoscan::ops::HolovizOp (Class HolovizOp)
- public holoscan::ops::InferenceOp (Class InferenceOp)
- public holoscan::ops::InferenceProcessorOp (Class InferenceProcessorOp)
- public holoscan::ops::PingRxOp (Class PingRxOp)
- public holoscan::ops::PingTxOp (Class PingTxOp)
- public holoscan::ops::SegmentationPostprocessorOp (Class SegmentationPostprocessorOp)
- public holoscan::ops::VideoStreamRecorderOp (Class VideoStreamRecorderOp)

- `public holoscan::ops::VideoStreamReplayerOp` (Class VideoStreamReplayerOp)
- `public holoscan::ops::VirtualOperator` (Class VirtualOperator)

Class Documentation

class Operator : public holoscan::ComponentBase

Base class for all operators.

An operator is the most basic unit of work in Holoscan SDK. An Operator receives streaming data at an input port, processes it, and publishes it to one of its output ports.

This class is the base class for all operators. It provides the basic functionality for all operators.

> **Note**
>
> This class is not intended to be used directly. Inherit from this class to create a new operator.

Subclassed by holoscan::ops::AJASourceOp, holoscan::ops::AsyncPingRxOp, holoscan::ops::AsyncPingTxOp, holoscan::ops::BayerDemosaicOp, holoscan::ops::FormatConverterOp, holoscan::ops::ForwardOp, holoscan::ops::GXFOperator, holoscan::ops::HolovizOp, holoscan::ops::InferenceOp, holoscan::ops::InferenceProcessorOp, holoscan::ops::PingRxOp, holoscan::ops::PingTxOp, holoscan::ops::SegmentationPostprocessorOp, holoscan::ops::VideoStreamRecorderOp, holoscan::ops::VideoStreamReplayerOp, holoscan::ops::VirtualOperator

Public Types

enum class OperatorType

Operator type used by the executor.

Values:

enumerator kNative

Native operator.

enumerator kGXF

GXF operator.

enumerator kVirtual

Virtual operator. (for internal use, not intended for use by application authors)

Public Functions

template<typename ArgT, typename ...ArgsT, typename = std::enable_if_t<!std::is_base_of_v<holoscan::Operator, std::decay_t<ArgT>> && (std::is_same_v<holoscan::Arg, std::decay_t<ArgT>> || std::is_same_v<holoscan::ArgList, std::decay_t<ArgT>> || std::is_base_of_v<holoscan::Condition, typename holoscan::type_info<ArgT>::derived_type> || std::is_base_of_v<holoscan::Resource, typename holoscan::type_info<ArgT>::derived_type>)>>
inline explicit Operator(ArgT &&arg, ArgsT&&... args)

Construct a new Operator object.

Parameters

args – The arguments to be passed to the operator.

Operator() = default

~Operator() override = default

inline OperatorType operator_type() const

Get the operator type.

Returns

The operator type.

inline Operator &id(int64_t id)

Set the Operator ID.

Parameters

id – The ID of the operator.

Returns

The reference to this operator.

inline Operator &name(const std::string &name)

Set the name of the operator.

Parameters

name – The name of the operator.

Returns

The reference to this operator.

inline Operator &fragment(Fragment *fragment)

Set the fragment of the operator.

Parameters

fragment – The pointer to the fragment of the operator.

Returns

The reference to this operator.

inline Operator &spec(const std::shared_ptr<OperatorSpec> &spec)

Set the operator spec.

Parameters

spec – The operator spec.

Returns

The reference to this operator.

inline OperatorSpec *spec()

Get the operator spec.

Returns

The operator spec.

inline std::shared_ptr<OperatorSpec> spec_shared()

Get the shared pointer to the operator spec.

Returns

The shared pointer to the operator spec.

template<typename ConditionT>
inline std::shared_ptr<ConditionT> condition(const std::string &name)

Get a shared pointer to the Condition object.

Parameters

name – The name of the condition.

Returns

The reference to the Condition object. If the condition does not exist, return the nullptr.

inline std::unordered_map<std::string, std::shared_ptr<Condition>> &conditions()

Get the conditions of the operator.

Returns

The conditions of the operator.

template<typename ResourceT>
inline std::shared_ptr<ResourceT> resource(const std::string &name)

Get a shared pointer to the Resource object.

Parameters

name – The name of the resource.

Returns

The reference to the Resource object. If the resource does not exist, returns the nullptr.

inline std::unordered_map<std::string, std::shared_ptr<Resource>> &resources()

Get the resources of the operator.

Returns

The resources of the operator.

inline void add_arg(const std::shared_ptr<Condition> &arg)

Add a condition to the operator.

Parameters

arg – The condition to add.

inline void add_arg(std::shared_ptr<Condition> &&arg)

Add a condition to the operator.

Parameters

arg – The condition to add.

inline void add_arg(const std::shared_ptr<Resource> &arg)

Add a resource to the operator.

Parameters

arg – The resource to add.

inline void add_arg(std::shared_ptr<Resource> &&arg)

Add a resource to the operator.

Parameters

arg – The resource to add.

inline virtual void setup(OperatorSpec &spec)

Define the operator specification.

Parameters

spec – The reference to the operator specification.

bool is_root()

Returns whether the operator is a root operator based on its fragment's graph.

Returns

True, if the operator is a root operator; false, otherwise

bool is_user_defined_root()

Returns whether the operator is a user-defined root operator i.e., the first operator added to the graph.

Returns

True, if the operator is a user-defined root operator; false, otherwise

bool is_leaf()

Returns whether the operator is a leaf operator based on its fragment's graph.

Returns

True, if the operator is a leaf operator; false, otherwise

virtual void initialize() override

Initialize the operator.

This function is called when the fragment is initialized by Executor::initialize_fragment().

inline virtual void start()

Implement the startup logic of the operator.

This method is called multiple times over the lifecycle of the operator according to the order defined in the lifecycle, and used for heavy initialization tasks such as allocating memory resources.

inline virtual void stop()

Implement the shutdown logic of the operator.

This method is called multiple times over the lifecycle of the operator according to the order defined in the lifecycle, and used for heavy deinitialization tasks such as deallocation of all resources previously assigned in start.

inline virtual void compute(InputContext &op_input, OutputContext &op_output, ExecutionContext &context)

Implement the compute method.

This method is called by the runtime multiple times. The runtime calls this method until the operator is stopped.

Parameters

- **op_input** – The input context of the operator.
- **op_output** – The output context of the operator.
- **context** – The execution context of the operator.

virtual YAML::Node to_yaml_node() const override

Get a YAML representation of the operator.

Returns

YAML node including type, specs, conditions and resources of the operator in addition to the base component properties.

inline std::shared_ptr<nvidia::gxf::GraphEntity> graph_entity()

Get the GXF GraphEntity object corresponding to this operator.

Returns

graph entity corresponding to the operator

inline int64_t id() const

Get the identifier of the component.

By default, the identifier is set to -1. It is set to a valid value when the component is initialized.

With the default executor (GXFExecutor), the identifier is set to the GXF component ID.

Returns

The identifier of the component.

inline const std::string &name() const

Get the name of the component.

Returns

The name of the component.

inline Fragment *fragment()

Get a pointer to Fragment object.

Returns

The Pointer to Fragment object.

inline void add_arg(const Arg &arg)

Add an argument to the component.

Parameters

arg – The argument to add.

inline void add_arg(Arg &&arg)

Add an argument to the component.

Parameters

arg – The argument to add.

inline void add_arg(const ArgList &arg)

Add a list of arguments to the component.

Parameters

arg – The list of arguments to add.

inline void add_arg(ArgList &&arg)

Add a list of arguments to the component.

Parameters

arg – The list of arguments to add.

Public Static Functions

template<typename typeT>
static inline void register_converter()

Register the argument setter for the given type.

If the operator has an argument with a custom type, the argument setter must be registered using this method.

The argument setter is used to set the value of the argument from the YAML configuration.

This method can be called in the initialization phase of the operator (e.g., `initialize()`). The example below shows how to register the argument setter for the custom type (`Vec3`):

```
void MyOp::initialize() { register_converter<Vec3>(); }
```

It is assumed that `YAML::convert<T>::encode` and `YAML::convert<T>::decode` are implemented for the given type. You need to specialize the `YAML::convert<>` template class.

For example, suppose that you had a `Vec3` class with the following members:

```
struct Vec3 { // make sure you have overloaded operator==() for the comparison
double x, y, z; };
```

You can define the `YAML::convert<Vec3>` as follows in a '.cpp' file:

```
namespace YAML { template<> struct convert<Vec3> { static Node
encode(const Vec3& rhs) { Node node; node.push_back(rhs.x);
node.push_back(rhs.y); node.push_back(rhs.z); return node; } static bool
decode(const Node& node, Vec3& rhs) { if(!node.IsSequence() || node.size() !=
3) { return false; } rhs.x = node[0].as<double>(); rhs.y = node[1].as<double>();
rhs.z = node[2].as<double>(); return true; } }; }
```

Please refer to the yaml-cpp documentation for more details.

Template Parameters

typeT – The type of the argument to register.

static std::pair<std::string, std::string> parse_port_name(const std::string &op_port_name)

Return operator name and port name from a string in the format of "<op_name>[.<port_name>]".

template<typename typeT>
static inline void register_codec(const std::string &codec_name, bool overwrite = true)

Register the codec for serialization/deserialization of a custom type.

If the operator has an argument with a custom type, the codec must be registered using this method.

For example, suppose we want to emit using the following custom struct type:

```
namespace holoscan { struct Coordinate { int16_t x; int16_t y; int16_t z; } } // namespace holoscan
```

Then, we can define codec<Coordinate> as follows where the serialize and deserialize methods would be used for serialization and deserialization of this type, respectively.

```
namespace holoscan { template <> struct codec<Coordinate> { static expected<size_t, RuntimeError> serialize(const Coordinate& value, Endpoint* endpoint) { return serialize_trivial_type<Coordinate>(value, endpoint); } static expected<Coordinate, RuntimeError> deserialize(Endpoint* endpoint) { return deserialize_trivial_type<Coordinate>(endpoint); } }; } // namespace holoscan
```

In this case, since this is a simple struct with a static size, we can use the existing serialize_trivial_type and deserialize_trivial_type implementations.

Finally, to register this custom codec at runtime, we need to make the following call within the setup method of our Operator.

```
register_codec<Coordinate>("Coordinate");
```

Template Parameters

typeT – The type of the argument to register.

Parameters

- **codec_name** – The name of the codec (must be unique unless overwrite is true).
- **overwrite** – If true and codec_name already exists, the codec will be overwritten.

Protected Functions

gxf_uid_t initialize_graph_entity(void *context, const std::string &entity_prefix = "")

This function creates a GraphEntity corresponding to the operator.

Parameters

- **context** – The GXF context.
- **name** – The name of the entity to create.

Returns

The GXF entity eid corresponding to the graph entity.

virtual gxf_uid_t add_codelet_to_graph_entity()

Add this operator as the codelet in the GXF GraphEntity.

Returns

The codelet component id corresponding to GXF codelet.

void initialize_conditions()

Initialize conditions and add GXF conditions to graph_entity_.

void initialize_resources()

Initialize resources and add GXF resources to graph_entity_.

void update_params_from_args()

Update parameters based on the specified arguments.

virtual void set_parameters()

Set the parameters based on defaults (sets GXF parameters for GXF operators)

MessageLabel get_consolidated_input_label()

> This function returns a consolidated MessageLabel for all the input ports of an Operator. If there is no input port (root Operator), then a new MessageLabel with the current Operator and default receive timestamp is returned.
>
> Returns
>
> The consolidated MessageLabel

inline void update_input_message_label(std::string input_name, MessageLabel m)

> Update the input_message_labels map with the given MessageLabel a corresponding input_name.
>
> Parameters
>
> - **input_name** – The input port name for which the MessageLabel is updated
> - **m** – The new MessageLabel that will be set for the input port

inline void delete_input_message_label(std::string input_name)

> Delete the input_message_labels map entry for the given input_name.
>
> Parameters
>
> **input_name** – The input port name for which the MessageLabel is deleted

inline void reset_input_message_labels()

Reset the input message labels to clear all its contents. This is done for a leaf operator when it finishes its execution as it is assumed that all its inputs are processed.

inline std::map<std::string, uint64_t> num_published_messages_map()

> Get the number of published messages for each output port indexed by the output port name.

The function is utilized by the DFFTCollector to update the DataFlowTracker with the number of published messages for root operators.

Returns

The map of the number of published messages for every output name.

void update_published_messages(std::string output_name)

This function updates the number of published messages for a given output port.

Parameters

output_name – The name of the output port

virtual void reset_graph_entities()

Reset the GXF GraphEntity of any components associated with this operator.

void update_params_from_args(std::unordered_map<std::string, ParameterWrapper> &params)

Update parameters based on the specified arguments.

Protected Attributes

OperatorType operator_type_ = OperatorType::kNative

The type of the operator.

std::shared_ptr<OperatorSpec> spec_

The operator spec of the operator.

std::unordered_map<std::string, std::shared_ptr<Condition>> conditions_

The conditions of the operator.

std::unordered_map<std::string, std::shared_ptr<Resource>> resources_

The resources used by the operator.

std::shared_ptr<nvidia::gxf::GraphEntity> graph_entity_

GXF graph entity corresponding to the Operator

Protected Static Functions

template<typename typeT>
static inline void register_argument_setter()

Register the argument setter for the given type.

Please refer to the documentation of `register_converter()` for more details.

Template Parameters

typeT – The type of the argument to register.

Friends

friend class AnnotatedDoubleBufferReceiver

friend class AnnotatedDoubleBufferTransmitter

friend class DFFTCollector

friend class holoscan::gxf::GXFExecutor

friend class Fragment