Class Fragment

Table of contents

- Defined in File fragment.hpp

Inheritance Relationships

Derived Types

- `public holoscan::Application` (Class Application)
- `public holoscan::gxf::OperatorWrapperFragment` (Class OperatorWrapperFragment)

Class Documentation

class Fragment

The fragment of the application.

A fragment is a building block of the Application. It is a directed graph of operators. A fragment can be assigned to a physical node of a Holoscan cluster during execution. The run-time execution manages communication across fragments. In a Fragment, Operators (Graph Nodes) are connected to each other by flows (Graph Edges).

Subclassed by holoscan::Application, holoscan::gxf::OperatorWrapperFragment

Public Functions

Fragment() = default

virtual ~Fragment() = default

Fragment(Fragment&&) = default

Fragment &operator=(Fragment&&) = default

Fragment &name(const std::string &name) &

Set the name of the operator.

Parameters

name – The name of the operator.

Returns

The reference to this fragment (for chaining).

Fragment &&name(const std::string &name) &&

Set the name of the operator.

Parameters

name – The name of the operator.

Returns

The reference to this fragment (for chaining).

const std::string &name() const

Get the name of the fragment.

Returns

The name of the fragment.

Fragment &application(Application *app)

Set the application of the fragment.

Parameters

app – The pointer to the application of the fragment.

Returns

The reference to this fragment (for chaining).

Application *application() const

Get the application of the fragment.

Returns

The pointer to the application of the fragment.

void config(const std::string &config_file, const std::string &prefix = "")

Set the configuration of the fragment.

The configuration file is a YAML file that has the information of GXF extension paths and some parameter values for operators.

The `extensions` field in the YAML configuration file is a list of GXF extension paths. The paths can be absolute or relative to the current working directory, considering paths in `LD_LIBRARY_PATH` environment variable.

The paths can consist of the following parts:

- GXF core extensions
    - built-in extensions such as `libgxf_std.so` and `libgxf_cuda.so`.
    - `libgxf_std.so`, `libgxf_cuda.so`, `libgxf_multimedia.so`, `libgxf_serialization.so` are always loaded by default.
    - GXF core extensions are copied to the `lib` directory of the build/installation directory.
- Other GXF extensions
    - GXF extensions that are required for operators that this fragment uses.
    - some core GXF extensions such as `libgxf_stream_playback.so` are always loaded by default.
    - these paths are usually relative to the build/installation directory.

The extension paths are used to load dependent GXF extensions at runtime when `run()` method is called.

For other fields in the YAML file, you can freely define the parameter values for operators/fragments.

For example:

> extensions: - libmy_recorder.so replayer: directory: "../data/racerx" basename: "racerx" frame_rate: 0 # *as specified in timestamps* repeat: false # *default: false* realtime: true # *default: true* count: 0 # *default: 0 (no frame count restriction)* recorder: out_directory: "/tmp" basename: "tensor_out"

You can get the value of this configuration file by calling `from_config()` method.

If the application is executed with `--config` option or HOLOSCAN_CONFIG_PATH environment, the configuration file is overridden by the configuration file specified by the option or environment variable.

Parameters

- **config_file** – The path to the configuration file.
- **prefix** – The prefix string that is prepended to the key of the configuration. (not implemented yet)

void config(std::shared_ptr<Config> &config)

Set the configuration of the fragment.

If you want to set the configuration of the fragment manually, you can use this method. However, it is recommended to use `config` method because once you set the configuration manually, you cannot get the configuration from the override file (through `--config` option or HOLOSCAN_CONFIG_PATH environment variable).

Parameters

config – The shared pointer to the configuration of the fragment (`Config` object).

Config &config()

Get the configuration of the fragment.

Returns

The reference to the configuration of the fragment (`Config` object.)

OperatorGraph &graph()

Get the graph of the fragment.

Returns

The reference to the graph of the fragment (`Graph` object.)

Executor &executor()

Get the executor of the fragment.

Returns

The reference to the executor of the fragment (`Executor` object.)

std::shared_ptr<Scheduler> scheduler()

Get the scheduler used by the executor.

Returns

The reference to the scheduler of the fragment's executor (`Scheduler` object.)

void scheduler(const std::shared_ptr<Scheduler> &scheduler)

std::shared_ptr<NetworkContext> network_context()

Get the network context used by the executor.

Returns

The reference to the network context of the fragment's executor (`NetworkContext` object.)

void network_context(const std::shared_ptr<NetworkContext> &network_context)

ArgList from_config(const std::string &key)

Get the Argument(s) from the configuration file.

For the given key, this method returns the value of the configuration file.

For example:

```
source: "replayer" do_record: false # or 'true' if you want to record input video stream. aja: width: 1920 height: 1080 rdma: true
```

`from_config("aja")` returns an ArgList (vector-like) object that contains the following items:

- `Arg`
- `Arg`
- `Arg`

You can use '.' (dot) to access nested fields.

`from_config("aja.rdma")` returns an ArgList object that contains only one item and it can be converted to `bool` through `ArgList::as()` method:

```
bool is_rdma = from_config("aja.rdma").as<bool>();
```

Parameters

key – The key of the configuration.

Returns

The argument list of the configuration for the key.

std::unordered_set<std::string> config_keys()

Determine the set of keys present in a Fragment's config.

Returns

The set of valid keys.

template<typename OperatorT, typename StringT, typename ...ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>>
inline std::shared_ptr<OperatorT> make_operator(StringT name, ArgsT&&... args)

Create a new operator.

Template Parameters

OperatorT – The type of the operator.

Parameters

- **name** – The name of the operator.
- **args** – The arguments for the operator.

Returns

The shared pointer to the operator.

template<typename OperatorT, typename ...ArgsT>
inline std::shared_ptr<OperatorT> make_operator(ArgsT&&... args)

Create a new operator.

Template Parameters

OperatorT – The type of the operator.

Parameters

args – The arguments for the operator.

Returns

The shared pointer to the operator.

template<typename ResourceT, typename StringT, typename ...ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>>
inline std::shared_ptr<ResourceT> make_resource(StringT name, ArgsT&&... args)

Create a new (operator) resource.

Template Parameters

ResourceT – The type of the resource.

Parameters

- **name** – The name of the resource.
- **args** – The arguments for the resource.

Returns

The shared pointer to the resource.

template<typename ResourceT, typename ...ArgsT>
inline std::shared_ptr<ResourceT> make_resource(ArgsT&&... args)

Create a new (operator) resource.

Template Parameters

ResourceT – The type of the resource.

Parameters

args – The arguments for the resource.

Returns

The shared pointer to the resource.

template<typename ConditionT, typename StringT, typename ...ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>>
inline std::shared_ptr<ConditionT> make_condition(StringT name, ArgsT&&... args)

Create a new condition.

Template Parameters

ConditionT – The type of the condition.

Parameters

- **name** – The name of the condition.
- **args** – The arguments for the condition.

Returns

The shared pointer to the condition.

template<typename ConditionT, typename ...ArgsT>
inline std::shared_ptr<ConditionT> make_condition(ArgsT&&... args)

Create a new condition.

Template Parameters

ConditionT – The type of the condition.

Parameters

args – The arguments for the condition.

Returns

The shared pointer to the condition.

template<typename SchedulerT, typename StringT, typename ...ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>>
inline std::shared_ptr<SchedulerT> make_scheduler(StringT name, ArgsT&&... args)

Create a new scheduler.

Template Parameters

SchedulerT – The type of the scheduler.

Parameters

- **name** – The name of the scheduler.
- **args** – The arguments for the scheduler.

Returns

The shared pointer to the scheduler.

template<typename SchedulerT, typename ...ArgsT>
inline std::shared_ptr<SchedulerT> make_scheduler(ArgsT&&... args)

Create a new scheduler.

Template Parameters

SchedulerT – The type of the scheduler.

Parameters

args – The arguments for the scheduler.

Returns

The shared pointer to the scheduler.

template<typename NetworkContextT, typename StringT, typename ...ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>>
inline std::shared_ptr<NetworkContextT> make_network_context(StringT name, ArgsT&&... args)

Create a new network context.

Template Parameters

NetworkContextT – The type of the network context.

Parameters

- **name** – The name of the network context.
- **args** – The arguments for the network context.

Returns

The shared pointer to the network context.

template<typename NetworkContextT, typename ...ArgsT>
inline std::shared_ptr<NetworkContextT> make_network_context(ArgsT&&... args)

Create a new network context.

Template Parameters

NetworkContextT – The type of the network context.

Parameters

args – The arguments for the network context.

Returns

The shared pointer to the network context.

virtual void add_operator(const std::shared_ptr<Operator> &op)

Add an operator to the graph.

The information of the operator is stored in the Graph object. If the operator is already added, this method does nothing.

Parameters

op – The operator to be added.

virtual void add_flow(const std::shared_ptr<Operator> &upstream_op, const std::shared_ptr<Operator> &downstream_op)

Add a flow between two operators.

An output port of the upstream operator is connected to an input port of the downstream operator. The information about the flow (edge) is stored in the Graph object.

If the upstream operator or the downstream operator is not in the graph, it will be added to the graph.

If there are multiple output ports in the upstream operator or multiple input ports in the downstream operator, it shows an error message.

Parameters

- **upstream_op** – The upstream operator.
- **downstream_op** – The downstream operator.

virtual void add_flow(const std::shared_ptr<Operator> &upstream_op, const std::shared_ptr<Operator> &downstream_op, std::set<std::pair<std::string, std::string>>

port_pairs)

Add a flow between two operators.

An output port of the upstream operator is connected to an input port of the downstream operator. The information about the flow (edge) is stored in the Graph object.

If the upstream operator or the downstream operator is not in the graph, it will be added to the graph.

In `port_pairs`, an empty port name (“”) can be used for specifying a port name if the operator has only one input/output port.

If a non-existent port name is specified in `port_pairs`, it first checks if there is a parameter with the same name but with a type of `holoscan::IOSpec` in the downstream operator. If there is such a parameter (e.g., `receivers`), it creates a new input port with a specific label (`&lt;parameter name&gt;:&lt;index&gt;`. e.g., `receivers:0`), otherwise it shows an error message.

For example, if a parameter `receivers` want to have an arbitrary number of receivers,

```
class HolovizOp : public holoscan::ops::GXFOperator { ... private:
Parameter<std::vector<holoscan::IOSpec*>> receivers_; ...
```

Instead of creating a fixed number of input ports (e.g., `source_video` and `tensor`) and assigning them to the parameter (`receivers`):

```
void HolovizOp::setup(OperatorSpec& spec) { ... auto& in_source_video =
spec.input<holoscan::gxf::Entity>("source_video"); auto& in_tensor =
spec.input<holoscan::gxf::Entity>("tensor"); spec.param(receivers_, "receivers",
"Input Receivers", "List of input receivers.", {&in_source_video, &in_tensor}); ...
```

You can skip the creation of input ports and assign them to the parameter (`receivers`) as follows:

```
void HolovizOp::setup(OperatorSpec& spec) { ... spec.param(receivers_,
"receivers", "Input Receivers", "List of input receivers.", {&in_source_video,
&in_tensor}); ...
```

This makes the following code possible in the Application's `compose()` method:

```
add_flow(source, visualizer_format_converter);
add_flow(visualizer_format_converter, visualizer, {{"", "receivers"}});
add_flow(source, format_converter); add_flow(format_converter, inference);
add_flow(inference, visualizer, {{"", "receivers"}});
```

Instead of:

```
add_flow(source, visualizer_format_converter);
add_flow(visualizer_format_converter, visualizer, {{"", "source_video"}});
add_flow(source, format_converter); add_flow(format_converter, inference);
add_flow(inference, visualizer, {{"", "tensor"}});
```

By using the parameter (`receivers`) with `holoscan::IOSpec` type, the framework creates input ports (`receivers:0` and `receivers:1`) implicitly and connects them (and adds the references of the input ports to the `receivers` vector).

Parameters

- **upstream_op** – The upstream operator.
- **downstream_op** – The downstream operator.
- **port_pairs** – The port pairs. The first element of the pair is the port of the upstream operator and the second element is the port of the downstream operator.

virtual void compose()

Compose a graph.

The graph is composed by adding operators and flows in this method.

virtual void run()

Initialize the graph and run the graph.

This method calls `compose()` to compose the graph, and runs the graph.

virtual std::future<void> run_async()

Initialize the graph and run the graph asynchronously.

This method calls `compose()` to compose the graph, and runs the graph asynchronously.

Returns

The future object.

DataFlowTracker &track(uint64_t num_start_messages_to_skip = kDefaultNumStartMessagesToSkip, uint64_t num_last_messages_to_discard = kDefaultNumLastMessagesToDiscard, int latency_threshold = kDefaultLatencyThreshold)

Turn on data frame flow tracking.

A reference to a DataFlowTracker object is returned rather than a pointer so that the developers can use it as an object without unnecessary pointer dereferencing.

Parameters

- **num_start_messages_to_skip** – The number of messages to skip at the beginning.
- **num_last_messages_to_discard** – The number of messages to discard at the end.
- **latency_threshold** – The minimum end-to-end latency in milliseconds to account for in the end-to-end latency metric calculations.

Returns

A reference to the DataFlowTracker object in which results will be stored.

inline DataFlowTracker *data_flow_tracker()

Get the DataFlowTracker object for this fragment.

Returns

The pointer to the DataFlowTracker object.

virtual void compose_graph()

Calls compose() if the graph is not composed yet.

FragmentPortMap port_info() const

Get an easily serializable summary of port information.

The FragmentPortMap class is used by distributed applications to send port information between application workers and the driver.

Returns

An unordered_map of the fragment’s port information where the keys are operator names and the values are a 3-tuple. The first two elements of the tuple are the set of input and output port names, respectively. The third element of the tuple is the set of “receiver” parameters (those with type std::vector<IOSpec*>).

Protected Functions

template<typename ConfigT, typename ...ArgsT>
inline std::shared_ptr<Config> make_config(ArgsT&&... args)

template<typename GraphT>
inline std::unique_ptr<GraphT> make_graph()

template<typename ExecutorT>
inline std::shared_ptr<Executor> make_executor()

template<typename ExecutorT, typename ...ArgsT>
inline std::unique_ptr<Executor> make_executor(ArgsT&&... args)

void reset_graph_entities()

Cleanup helper that will by called by GXFExecutor prior to GxfContextDestroy.

Protected Attributes

std::string name_

The name of the fragment.

Application *app_ = nullptr

The application that this fragment belongs to.

std::shared_ptr<Config> config_

The configuration of the fragment.

std::shared_ptr<Executor> executor_

The executor for the fragment.

std::unique_ptr<OperatorGraph> graph_

The graph of the fragment.

std::shared_ptr<Scheduler> scheduler_

The scheduler used by the executor.

std::shared_ptr<NetworkContext> network_context_

The network_context used by the executor.

std::shared_ptr<DataFlowTracker> data_flow_tracker_

The DataFlowTracker for the fragment.

bool is_composed_ = false

Whether the graph is composed or not.

Friends

friend class Application

friend class AppDriver

friend class gxf::GXFExecutor