Class Application

Table of contents

- Defined in File application.hpp

Inheritance Relationships

Base Type

- `public holoscan::Fragment` (Class Fragment)

Class Documentation

class Application : public holoscan::Fragment

Application class.

An application acquires and processes streaming data. An application is a collection of fragments where each fragment can be allocated to execute on a physical node of a Holoscan cluster.

Public Functions

explicit Application(const std::vector<std::string> &argv = {})

> Construct a new Application object.
>
> This constructor parses the command line for flags that are recognized by App Driver/Worker, and removes all recognized flags so users can use the remaining flags for their own purposes.
>
> The command line arguments are retrieved from /proc/self/cmdline so that the single-fragment application works as expected without any command line arguments.
>
> The arguments after processing arguments are stored in the `argv_` member variable and the reference to the vector of arguments can be accessed through the `argv()` method.
>
> Example:

```
#include <holoscan/holoscan.hpp> class MyPingApp : public holoscan::Application { // ... }; int main(int argc, char** argv) { auto my_argv = holoscan::Application({"myapp", "--driver", "my_arg1", "--address=10.0.0.1"}).argv(); HOLOSCAN_LOG_INFO(" my_argv: {}", fmt::join(my_argv, " ")); HOLOSCAN_LOG_INFO( " argv: {} (argc: {}) ", fmt::join(std::vector<std::string>(argv, argv + argc), " "), argc); auto app_argv = holoscan::Application().argv(); // do not use reference ('auto&') here HOLOSCAN_LOG_INFO("app_argv: {} (size: {})", fmt::join(app_argv, " "), app_argv.size()); auto app = holoscan::make_application<MyPingApp>(); HOLOSCAN_LOG_INFO("app->argv() == app_argv: {}", app->argv() == app_argv); app->run(); return 0; } // $ ./myapp --driver --input image.dat --address 10.0.0.20 // my_argv: myapp my_arg1 // argv: ./myapp --driver --input image.dat --address 10.0.0.20 (argc: 6) // app_argv: ./myapp --input image.dat (size: 3) // app->argv() == app_argv: true
```

Parameters

argv – The command line arguments.

~Application() override = default

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

Create a new fragment.

Template Parameters

FragmentT – The type of the fragment to create.

Parameters

- **name** – The name of the fragment.
- **args** – The arguments to pass to the fragment constructor.

Returns

The shared pointer to the created fragment.

template<typename FragmentT, typename ...ArgsT>
inline std::shared_ptr<FragmentT> make_fragment(ArgsT&&... args)

Create a new fragment.

Template Parameters

FragmentT – The type of the fragment to create.

Parameters

args – The arguments to pass to the fragment constructor.

Returns

The shared pointer to the created fragment.

std::string &description()

Get the application description.

Returns

The application description.

Application &description(const std::string &desc) &

Set the application description.

Parameters

desc – The application description

Returns

The reference to this application (for chaining).

Application &&description(const std::string &desc) &&

Set the application description.

Parameters

desc – The application description

Returns

The reference to this application (for chaining).

std::string &version()

Get the application version.

Returns

The application version.

Application &version(const std::string &version) &

Set the application version.

Parameters

version – The application version

Returns

The reference to this application (for chaining).

Application &&version(const std::string &version) &&

Set the application version.

Parameters

version – The application version

Returns

The reference to this application (for chaining).

std::vector<std::string> &argv()

Get the reference to the command line arguments after processing flags.

The returned vector includes the executable name as the first element.

Returns

The reference to the command line arguments after processing flags.

CLIOptions &options()

Get the reference to the CLI options.

Returns

The reference to the CLI options.

FragmentGraph &fragment_graph()

Get the fragment connection graph.

When two operators are connected through `add_flow(Fragment, Fragment)`, the fragment connection graph is automatically updated. The fragment connection graph is used to assign transmitters and receivers to the corresponding Operator instances in the fragment so that the application can be executed in a distributed manner.

Returns

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

virtual void add_fragment(const std::shared_ptr<Fragment> &frag)

Add a fragment to the graph.

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

Parameters

frag – The fragment to be added.

virtual void add_flow(const std::shared_ptr<Fragment> &upstream_frag, const std::shared_ptr<Fragment> &downstream_frag, std::set<std::pair<std::string, std::string>> port_pairs)

Add a flow between two fragments.

It takes two fragments and a vector of string pairs as arguments. The vector of string pairs is used to connect the output ports of the first fragment to the input ports of the second fragment. The input and output ports of the operators are specified as a string in the format of `<operator name>.<port name>`. If the operator has only one input or output port, the port name can be omitted.

```
class App : public holoscan::Application { public: void compose() override { using namespace holoscan; auto fragment1 = make_fragment<Fragment1> ("fragment1"); auto fragment2 = make_fragment<Fragment2>("fragment2"); add_flow(fragment1, fragment2, {{"blur_image", "sharpen_image"}}); } };
```

In the above example, the output port of the `blur_image` operator in `fragment1` is connected to the input port of the `sharpen_image` operator in `fragment2`. Since `blur_image` and `sharpen_image` operators have only one output/input port, the port names are omitted.

The information about the flow (edge) is stored in the Graph object and can be accessed through the `fragment_graph()` method.

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

Parameters

- **upstream_frag** – The upstream fragment.
- **downstream_frag** – The downstream fragment.
- **port_pairs** – The port pairs. The first element of the pair is the output port of the operator in the upstream fragment and the second element is the input port of the operator in the downstream fragment.

virtual void compose_graph() override

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

virtual void run() override

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() override

> 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.

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 (`<parameter name>:<index>` . 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.

Protected Functions

AppDriver &driver()

Get the application driver.

Returns

The reference to the application driver.

AppWorker &worker()

Get the application worker.

Returns

The reference to the application worker.

void process_arguments()

Protected Attributes

std::string app_description_ = {}

The description of the application.

std::string app_version_ = {"0.0.0"}

The version of the application.

CLIParser cli_parser_

The command line parser.

std::vector<std::string> argv_

The command line arguments after processing flags.

std::unique_ptr<FragmentGraph> fragment_graph_

The fragment connection graph.

std::shared_ptr<AppDriver> app_driver_

The application driver.

std::shared_ptr<AppWorker> app_worker_

The application worker.

Protected Static Functions

static expected<SchedulerType, ErrorCode> get_distributed_app_scheduler_env()

static expected<bool, ErrorCode> get_stop_on_deadlock_env()

static expected<int64_t, ErrorCode> get_stop_on_deadlock_timeout_env()

static expected<int64_t, ErrorCode> get_max_duration_ms_env()

static expected<double, ErrorCode> get_check_recession_period_ms_env()

static void set_scheduler_for_fragments(std::vector<FragmentNodeType> &target_fragments)

Set the scheduler for fragments object.

Set scheduler for each fragment to use multi-thread scheduler by default because UCXTransmitter/UCXReceiver doesn't work with GreedyScheduler with the following graph.

- Fragment (fragment1)
  - Operator (op1)
    - Output port: out
  - Operator (op2)
    - Output port: out
- Fragment (fragment2)
  - Operator (op3)
    - Input ports
      - in1
      - in2

With the following graph connections, due to how UCXTransmitter/UCXReceiver works, UCX connections between op1 and op3 and between op2 and op3 are not established (resulting in a deadlock).

- op1.out -> op3.in1

- op2.out -> op3.in2

Parameters

target_fragments – The fragments to set the scheduler.

Friends

friend class AppDriver

friend class AppWorker