Program Listing for File operator.hpp

Return to documentation for file (include/holoscan/core/operator.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2022-2024 NVIDIA CORPORATION & AFFILIATES. All rights reserved. * SPDX-License-Identifier: Apache-2.0 * * Licensed under the Apache License, Version 2.0 (the "License"); * you may not use this file except in compliance with the License. * You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License. */ #ifndef HOLOSCAN_CORE_OPERATOR_HPP #define HOLOSCAN_CORE_OPERATOR_HPP #include <stdio.h> #include <algorithm> #include <iostream> #include <map> #include <memory> #include <stdexcept> #include <string> #include <type_traits> #include <unordered_map> #include <utility> #include "./arg.hpp" #include "./argument_setter.hpp" #include "./codec_registry.hpp" #include "./common.hpp" #include "./component.hpp" #include "./condition.hpp" #include "./forward_def.hpp" #include "./messagelabel.hpp" #include "./operator_spec.hpp" #include "./resource.hpp" #include "gxf/core/gxf.h" #include "gxf/app/graph_entity.hpp" #define HOLOSCAN_OPERATOR_FORWARD_TEMPLATE() \ 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>)>> #define HOLOSCAN_OPERATOR_FORWARD_ARGS(class_name) \ HOLOSCAN_OPERATOR_FORWARD_TEMPLATE() \ class_name(ArgT&& arg, ArgsT&&... args) \ : Operator(std::forward<ArgT>(arg), std::forward<ArgsT>(args)...) {} #define HOLOSCAN_OPERATOR_FORWARD_ARGS_SUPER(class_name, super_class_name) \ HOLOSCAN_OPERATOR_FORWARD_TEMPLATE() \ class_name(ArgT&& arg, ArgsT&&... args) \ : super_class_name(std::forward<ArgT>(arg), std::forward<ArgsT>(args)...) {} namespace holoscan { namespace gxf { class GXFExecutor; } *// namespace gxf* class Operator : public ComponentBase { public:

enum class OperatorType { kNative, kGXF, kVirtual, }; HOLOSCAN_OPERATOR_FORWARD_TEMPLATE() explicit Operator(ArgT&& arg, ArgsT&&... args) { add_arg(std::forward<ArgT>(arg)); (add_arg(std::forward<ArgsT>(args)), ...); } Operator() = default; ~Operator() override = default; OperatorType operator_type() const { return operator_type_; } using ComponentBase::id; Operator& id(int64_t id) { id_ = id; return *this; } using ComponentBase::name; Operator& name(const std::string& name) { *// Operator::parse_port_name requires that "." is not allowed in the Operator name* if (name.find(".") != std::string::npos) { throw std::invalid_argument(fmt::format("The . character is reserved and cannot be used in the operator name ('{}').", name)); } name_ = name; return *this; } using ComponentBase::fragment; Operator& fragment(Fragment* fragment) { fragment_ = fragment; return *this; } Operator& spec(const std::shared_ptr<OperatorSpec>& spec) { spec_ = spec; return *this; } OperatorSpec* spec() { return spec_.get(); } std::shared_ptr<OperatorSpec> spec_shared() { return spec_; } template <typename ConditionT> std::shared_ptr<ConditionT> condition(const std::string& name) { if (auto condition = conditions_.find(name); condition != conditions_.end()) { return std::dynamic_pointer_cast<ConditionT>(condition->second); } return nullptr; } std::unordered_map<std::string, std::shared_ptr<Condition>>& conditions() { return conditions_; } template <typename ResourceT> std::shared_ptr<ResourceT> resource(const std::string& name) { if (auto resource = resources_.find(name); resource != resources_.end()) { return std::dynamic_pointer_cast<ResourceT>(resource->second); } return nullptr; } std::unordered_map<std::string, std::shared_ptr<Resource>>& resources() { return resources_; } using ComponentBase::add_arg; void add_arg(const std::shared_ptr<Condition>& arg) { if (conditions_.find(arg->name()) != conditions_.end()) { HOLOSCAN_LOG_ERROR("Condition '{}' already exists in the operator. Please specify a unique " "name when creating a Condition instance.", arg->name()); } else { conditions_[arg->name()] = arg; } } void add_arg(std::shared_ptr<Condition>&& arg) { if (conditions_.find(arg->name()) != conditions_.end()) { HOLOSCAN_LOG_ERROR("Condition '{}' already exists in the operator. Please specify a unique " "name when creating a Condition instance.", arg->name()); } else { conditions_[arg->name()] = std::move(arg); } } void add_arg(const std::shared_ptr<Resource>& arg) { if (resources_.find(arg->name()) != resources_.end()) { HOLOSCAN_LOG_ERROR("Resource '{}' already exists in the operator. Please specify a unique " "name when creating a Resource instance.", arg->name()); } else { resources_[arg->name()] = arg; } } void add_arg(std::shared_ptr<Resource>&& arg) { if (resources_.find(arg->name()) !=

resources_.end()) { HOLOSCAN_LOG_ERROR("Resource '{}' already exists in the operator. Please specify a unique " "name when creating a Resource instance.", arg->name()); } else { resources_[arg->name()] = std::move(arg); } } virtual void setup(OperatorSpec& spec) { (void)spec; } bool is_root(); bool is_user_defined_root(); bool is_leaf(); void initialize() override; virtual void start() { *// Empty default implementation* } virtual void stop() { *// Empty default implementation* } virtual void compute(InputContext& op_input, OutputContext& op_output, ExecutionContext& context) { (void)op_input; (void)op_output; (void)context; } template <typename typeT> static void register_converter() { register_argument_setter<typeT>(); } static std::pair<std::string, std::string> parse_port_name(const std::string& op_port_name); template <typename typeT> static void register_codec(const std::string& codec_name, bool overwrite = true) { CodecRegistry::get_instance().add_codec<typeT>(codec_name, overwrite); } YAML::Node to_yaml_node() const override; std::shared_ptr<nvidia::gxf::GraphEntity> graph_entity() { return graph_entity_; } protected: *// Making the following classes as friend classes to allow them to access // get_consolidated_input_label, num_published_messages_map, update_input_message_label, // reset_input_message_labels and update_published_messages functions, which should only be called // externally by them* friend class AnnotatedDoubleBufferReceiver; friend class AnnotatedDoubleBufferTransmitter; friend class DFFTCollector; *// Make GXFExecutor a friend class so it can call protected initialization methods* friend class holoscan::gxf::GXFExecutor; *// Fragment should be able to call reset_graph_entities* friend class Fragment; gxf_uid_t initialize_graph_entity(void* context, const std::string& entity_prefix = ""); virtual gxf_uid_t add_codelet_to_graph_entity(); void initialize_conditions(); void initialize_resources(); using ComponentBase::update_params_from_args; void update_params_from_args(); virtual void set_parameters(); MessageLabel get_consolidated_input_label(); void update_input_message_label(std::string input_name, MessageLabel m) { input_message_labels[input_name] = m; } void delete_input_message_label(std::string input_name) { input_message_labels.erase(input_name); } void reset_input_message_labels() { input_message_labels.clear(); } std::map<std::string, uint64_t> num_published_messages_map() { return num_published_messages_map_; } void update_published_messages(std::string output_name); template <typename typeT> static void register_argument_setter() {

```
ArgumentSetter::get_instance().add_argument_setter<typeT>( [] (ParameterWrapper& param_wrap, Arg& arg) { std::any& any_param = param_wrap.value(); // If arg has no name and value, that indicates that we want to set the default value for // the native operator if it is not specified. if (arg.name().empty() && !arg.has_value()) { auto& param = *std::any_cast<Parameter<typeT>*> (any_param); param.set_default_value(); return; } std::any& any_arg = arg.value(); // Note that the type of any_param is Parameter<typeT>*, not Parameter<typeT>. auto& param = *std::any_cast<Parameter<typeT>*>(any_param); const auto& arg_type = arg.arg_type(); (void)param; auto element_type = arg_type.element_type(); auto container_type = arg_type.container_type(); HOLOSCAN_LOG_DEBUG( "Registering converter for parameter {} (element_type: {}, container_type: {})", arg.name(), static_cast<int>(element_type), static_cast<int>(container_type)); if (element_type == ArgElementType::kYAMLNode) { auto& arg_value = std::any_cast<YAML::Node&> (any_arg); typeT new_value; bool parse_ok = YAML::convert<typeT>::decode(arg_value, new_value); if (!parse_ok) { HOLOSCAN_LOG_ERROR("Unable to parse YAML node for parameter '{}'", arg.name()); } else { param = std::move(new_value); } } else { try { auto& arg_value = std::any_cast<typeT&>(any_arg); param = arg_value; } catch (const std::bad_any_cast& e) { HOLOSCAN_LOG_ERROR( "Bad any cast exception caught for argument '{}': {}", arg.name(), e.what()); } } }); } virtual void reset_graph_entities(); OperatorType operator_type_ = OperatorType::kNative; std::shared_ptr<OperatorSpec> spec_; std::unordered_map<std::string, std::shared_ptr<Condition>> conditions_; std::unordered_map<std::string, std::shared_ptr<Resource>> resources_; std::shared_ptr<nvidia::gxf::GraphEntity> graph_entity_; private: void set_op_backend(); bool has_ucx_connector(); std::unordered_map<std::string, MessageLabel> input_message_labels; std::map<std::string, uint64_t> num_published_messages_map_; void* op_backend_ptr = nullptr; }; } // namespace holoscan #endif/* HOLOSCAN_CORE_OPERATOR_HPP */
```