Program Listing for File io_context.hpp

Return to documentation for file (`include/holoscan/core/io_context.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_IO_CONTEXT_HPP #define HOLOSCAN_CORE_IO_CONTEXT_HPP
#include <any> #include <map> #include <memory> #include <string> #include
<typeinfo> #include <unordered_map> #include <utility> #include <vector>
#include <common/type_name.hpp> #include "./common.hpp" #include
"./domain/tensor_map.hpp" #include "./errors.hpp" #include "./expected.hpp"
#include "./gxf/entity.hpp" #include "./message.hpp" #include "./operator.hpp"
#include "./type_traits.hpp" namespace holoscan { static inline std::string
get_well_formed_name( const char* name, const std::unordered_map<std::string,
std::shared_ptr<IOSpec>>& io_list) { std::string well_formed_name; if (name ==
nullptr || name[0] == '\0') { if (io_list.size() == 1) { well_formed_name = io_list.begin()-
>first; } else { well_formed_name = ""; } } else { well_formed_name = name; } return
well_formed_name; } class InputContext { public: InputContext(ExecutionContext*
execution_context, Operator* op, std::unordered_map<std::string,
std::shared_ptr<IOSpec>>& inputs) : execution_context_(execution_context),
op_(op), inputs_(inputs) {} InputContext(ExecutionContext* execution_context,
Operator* op) : execution_context_(execution_context), op_(op), inputs_(op->spec()-
>inputs()) {} ExecutionContext* execution_context() const { return
execution_context_; } Operator* op() const { return op_; }
std::unordered_map<std::string, std::shared_ptr<IOSpec>>& inputs() const { return
inputs_; } bool empty(const char* name = nullptr) { // First see if the name could be
found in the inputs auto& inputs = op_->spec()->inputs(); auto it =
inputs.find(std::string(name)); if (it != inputs.end()) { return empty_impl(name); } //
Then see if it is in the parameters auto& params = op_->spec()->params(); auto it2 =
params.find(std::string(name)); if (it2 != params.end()) { auto& param_wrapper = it2-
```

```
>second; auto& arg_type = param_wrapper.arg_type(); if ((arg_type.element_type() !=
ArgElementType::kIOSpec) || (arg_type.container_type() !=
ArgContainerType::kVector)) { HOLOSCAN_LOG_ERROR("Input parameter with name
'{}' is not of type 'std::vector<IOSpec*>'", name); return true; } std::any& any_param
= param_wrapper.value(); // Note that the type of any_param is Parameter<typeT>*,
not Parameter<typeT>. auto& param =
*std::any_cast<Parameter<std::vector<IOSpec*>>*>(any_param); int num_inputs =
param.get().size(); for (int i = 0; i < num_inputs; ++i) { // if any of them is not empty
return false if (!empty_impl(fmt::format("{}:{}", name, i).c_str())) { return false; } }
return true; // all of them are empty, so return true. } HOLOSCAN_LOG_ERROR("Input
port '{}' not found", name); return true; } template <typename DataT>
holoscan::expected<DataT, holoscan::RuntimeError> receive(const char* name =
nullptr) { if constexpr (holoscan::is_vector_v<DataT>) { // It could either be a
parameter which is trying to receive from a vector // or a vector of values from the
inputs // First check, if it is trying to receive from a parameter auto& params = op_-
>spec()->params(); auto it = params.find(std::string(name)); if (it == params.end()) { //
the name is not a parameter, so it must be an input auto& inputs = op_->spec()-
>inputs(); if (inputs.find(std::string(name)) == inputs.end()) { auto error_message =
fmt::format("Unable to find input parameter or input port with name '{}'", name); //
Keep the debugging info on for development purposes
HOLOSCAN_LOG_DEBUG(error_message); return
make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} auto value = receive_impl(name); if (value.type() == typeid(nullptr_t)) { auto
error_message = fmt::format("No data is received from the input port with name
'{}'", name); HOLOSCAN_LOG_DEBUG(error_message); return
make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} try { DataT result = std::any_cast<DataT>(value); return result; } catch (const
std::bad_any_cast& e) { auto error_message = fmt::format( "Unable to cast input
(DataT of type std::vector) with input name '{}' ({}).", name, e.what());
HOLOSCAN_LOG_DEBUG(error_message); return
make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} } auto& param_wrapper = it->second; auto& arg_type = param_wrapper.arg_type();
if ((arg_type.element_type() != ArgElementType::kIOSpec) ||
(arg_type.container_type() != ArgContainerType::kVector)) { auto error_message =
fmt::format( "Input parameter with name '{}' is not of type 'std::vector<IOSpec*>'",
name); HOLOSCAN_LOG_ERROR(error_message); return
make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} std::any& any_param = param_wrapper.value(); // Note that the type of any_param is
Parameter<typeT>*, not Parameter<typeT>. auto& param =
*std::any_cast<Parameter<std::vector<IOSpec*>>*>(any_param);
std::vector<typename DataT::value_type> input_vector; int num_inputs =
param.get().size(); input_vector.reserve(num_inputs); for (int index = 0; index <
num_inputs; ++index) { // Check if the input name points to the parameter name of the
operator, // and the parameter type is 'std::vector<holoscan::IOSpec*>'. // In other
words, find if there is a receiver with a specific label // ('<parameter name>:<index>'. e.g,
'receivers:0') to return an object with // 'std::vector<std::shared_ptr<DataT_ElementT>'
type. auto value = receive_impl(fmt::format("{}:{}", name, index).c_str(), true); try { //
If the received data is nullptr, any_cast will try to cast to appropriate pointer // type.
Otherwise it will register an error. if constexpr (std::is_same_v<typename
DataT::value_type, std::any>) { input_vector.push_back(std::move(value)); } else {
auto casted_value = std::move(std::any_cast<typename DataT::value_type>(value));
input_vector.push_back(std::move(casted_value)); } } catch (const
std::bad_any_cast& e) { auto error_message = fmt::format("Unable to cast input
(DataT::value_type) with name '{}:{}' ({}).", name, index, e.what()); try { // An empty
holoscan::gxf::Entity will be added to the vector. typename DataT::value_type
placeholder; input_vector.push_back(std::move(placeholder)); error_message =
fmt::format("{}\tA placeholder value is added to the vector for input '{}:{}'.",
error_message, name, index); HOLOSCAN_LOG_WARN(error_message); } catch
(std::exception& e) { error_message = fmt::format( "{}\tUnable to add a placeholder
value to the vector for input '{}:{}' :{}. " "Skipping adding a value to the vector.",
error_message, name, index, e.what()); HOLOSCAN_LOG_ERROR(error_message);
continue; } } } return std::any_cast<DataT>(input_vector); } else { // If it is not a vector
then try to get the input directly and convert for respective data // type for an input auto
value = receive_impl(name); // If the received data is nullptr, then check whether nullptr
or empty holoscan::gxf::Entity // can be sent if (value.type() == typeid(nullptr_t)) {
HOLOSCAN_LOG_DEBUG("nullptr is received from the input port with name '{}'",
name); // If it is a shared pointer, or raw pointer then return nullptr because it might be
a valid // nullptr if constexpr (holoscan::is_shared_ptr_v<DataT>) { return nullptr; }
```

```
else if constexpr (std::is_pointer_v<DataT>) { return nullptr; } // If it's
holoscan::gxf::Entity then return an error message if constexpr
(is_one_of_derived_v<DataT, nvidia::gxf::Entity>) { auto error_message = fmt::format(
"Null received in place of nvidia::gxf::Entity or derived type for input {}", name);
return make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} else if constexpr (is_one_of_derived_v<DataT, holoscan::TensorMap>) { auto
error_message = fmt::format( "Null received in place of holoscan::TensorMap or
derived type for input {}", name); return
make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} } try { // Check if the types of value and DataT are the same or not if constexpr
(std::is_same_v<DataT, std::any>) { return value; } DataT return_value =
std::any_cast<DataT>(value); return return_value; } catch (const std::bad_any_cast&
e) { // If it is of the type of holoscan::gxf::Entity then show a specific error message if
constexpr (is_one_of_derived_v<DataT, nvidia::gxf::Entity>) { auto error_message =
fmt::format( "Unable to cast the received data to the specified type (holoscan::gxf::"
"Entity) for input {}: {}", name, e.what()); HOLOSCAN_LOG_DEBUG(error_message);
return make_unexpected<holoscan::RuntimeError>(
holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str()));
} else if constexpr (is_one_of_derived_v<DataT, holoscan::TensorMap>) { TensorMap
tensor_map; try { auto gxf_entity = std::any_cast<holoscan::gxf::Entity>(value); auto
components_expected = gxf_entity.findAll(); auto components =
components_expected.value(); for (size_t i = 0; i < components.size(); i++) { const
auto component = components[i]; const auto component_name = component-
>name(); if (std::string(component_name).compare("message_label") == 0) { // Skip
checking for Tensor as it's message label for DFFT continue; } if
(std::string(component_name).compare("cuda_stream_id_") == 0) { // Skip checking
for Tensor as it's a stream ID from CudaStreamHandler continue; }
std::shared_ptr<holoscan::Tensor> holoscan_tensor =
gxf_entity.get<holoscan::Tensor>(component_name); if (holoscan_tensor) {
tensor_map.insert({component_name, holoscan_tensor}); } } } catch (const
std::bad_any_cast& e) { auto error_message = fmt::format( "Unable to cast the
received data to the specified type (holoscan::TensorMap) for " "input {}: {}", name,
e.what()); HOLOSCAN_LOG_DEBUG(error_message); return
make_unexpected<holoscan::RuntimeError>(
```

holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str())); } return tensor_map; } auto error_message = fmt::format("Unable to cast the received data to the specified type ({}) for input {} of type {}: " "{}", nvidia::TypenameAsString<DataT>(), name, value.type().name(), e.what()); HOLOSCAN_LOG_DEBUG(error_message); return make_unexpected<holoscan::RuntimeError>(holoscan::RuntimeError(holoscan::ErrorCode::kReceiveError, error_message.c_str())); } } } protected: virtual bool empty_impl(const char* name = nullptr) { (void)name; return true; } virtual std::any receive_impl(const char* name = nullptr, bool no_error_message = false) { (void)name; (void)no_error_message; return nullptr; } ExecutionContext* execution_context_ = nullptr; Operator* op_ = nullptr; std::unordered_map<std::string, std::shared_ptr<IOSpec>>& inputs_; }; class OutputContext { public: OutputContext(ExecutionContext* execution_context, Operator* op) : execution_context_(execution_context), op_(op), outputs_(op->spec()->outputs()) {} OutputContext(ExecutionContext* execution_context, Operator* op, std::unordered_map<std::string, std::shared_ptr<IOSpec>>& outputs) : execution_context_(execution_context), op_(op), outputs_(outputs) {} ExecutionContext* execution_context() const { return execution_context_; } Operator* op() const { return op_; } std::unordered_map<std::string, std::shared_ptr<IOSpec>>& outputs() const { return outputs_; } enum class OutputType { kSharedPointer, kGXFEntity, kAny, }; template <typename DataT, typename = std::enable_if_t<!holoscan::is_one_of_derived_v< DataT, nvidia::gxf::Entity, std::any>>> void emit(std::shared_ptr<DataT>& data, const char* name = nullptr) { emit_impl(data, name); } template <typename DataT, typename = std::enable_if_t<holoscan::is_one_of_derived_v<DataT, nvidia::gxf::Entity>>> void emit(DataT& data, const char* name = nullptr) { *// if it is the same as nvidia::gxf::Entity then just pass it to emit_impl* if constexpr (holoscan::is_one_of_v<DataT, nvidia::gxf::Entity>) { emit_impl(data, name, OutputType::kGXFEntity); } else { *// Convert it to nvidia::gxf::Entity and then pass it to emit_impl // Otherwise, we will lose the type information and cannot cast appropriately in emit_impl* emit_impl(nvidia::gxf::Entity(data), name, OutputType::kGXFEntity); } } template <typename DataT, typename = std::enable_if_t<!holoscan::is_one_of_derived_v<DataT, nvidia::gxf::Entity>>> void emit(DataT data, const char* name = nullptr) { emit_impl(data, name, OutputType::kAny); } void emit(holoscan::TensorMap& data, const char* name = nullptr) { auto out_message = holoscan::gxf::Entity::New(execution_context_); for (auto& [key, tensor] : data) { out_message.add(tensor, key.c_str()); } emit(out_message, name); } protected: virtual void emit_impl(std::any data, const char* name = nullptr, OutputType out_type = OutputType::kSharedPointer) { (void)data; (void)name; (void)out_type; } ExecutionContext* execution_context_ = nullptr; Operator* op_ = nullptr; std::unordered_map<std::string, std::shared_ptr<IOSpec>>& outputs_; }; } *// namespace holoscan* #endif/* HOLOSCAN_CORE_IO_CONTEXT_HPP */