Implement a pybind11-compatible bindings for pocketpy

Context

Personal Information

Detailed

- **Name**: ykiko
- **Github Profile** ykiko
- **Email**: ykikoykikoykiko@gmail.com
- **Time Zone** : UTC+08:00 (China)
- **Location** : Shanghai, China

About Me

I'm a sophomore at Tongji University. In the past two years, I have been using C++ and gained rich experience in C++ programming. My main interest lies in compilers and programming languages, leading me to become familiar with the underlying principles of compilers and the implementation of advanced programming language features.

Last year, I began to get involved in open-source projects. At first, I just reported issues for projects such as Clang, MSVC, and GCC. Gradually, I started trying to commit code. Recently, I successfully submitted my first PR for the LLVM project. This is just the beginning, and I intend to continue contributing to open-source projects. Although my involvement in open-source projects is relatively recent, I have become familiar with GitHub's workflow and using Git.

Contributions

Open Source Contributions

- Implemented small_vector: I implemented a `small_vector<T, N>`. It avoids dynamic memory allocation when the count of elements in the vector is less than N, which is useful for storing AST nodes. This is particularly applicable since the depth of the AST tree is typically small, with examples like the number of function arguments usually being less than 10.

- Added bit_length method to int: I added the builtin method `bit_length` to int, which returns the number of bits to represent an integer in binary.

- Adopted a more efficient division algorithm for bigint: I implemented a more efficient division algorithm for pkpy's bigint, enabling it to calculate extremely large numbers that were previously unattainable.

- Implemented a basic version of pybind11: This was the prototype of this proposal.

- Added missing Kinds to libclang Python bindings: I added some missing Kinds to libclang Python bindings, which provide more information from the AST.

Personal Projects

The following are some of the projects I have worked on:

- magic cpp： magic cpp is a header-only C++ library. It aims to make it easier for you to use C++, including a series of functions such as visualizing type names, reflection of structs and enumerations, etc. It can help you get rid of the compiler's error messages which are difficult to read.
- clanglite：It is usually difficult to get the AST of C++ source code because of the complexity of the C++ syntax and semantics. Luckily, the Clang project can also be used as a library to parse C++ source code. However, the Clang API is not user-friendly, and it is difficult to use. Besides, it is not easy to install the dependencies of the Clang project. Most of the time, you need to compile the Clang project from the source. So I created this project; it aims to provide a Python binding for the Clang C++ API through pybind11, making it easier to use. Additionally, all dependencies will be packaged into a wheel file, so you can install it via pip. Then parse the C++ source code and get the AST easily.

Project Abstract

The project aims to introduce a pybind11-compatible interface to pocketpy as an alternative method for creating bindings for functions and classes, thereby facilitating easier interaction between the Python virtual machine and C++ native code.

Project Description

Motivation

pocketpy is a lightweight Python interpreter and can be easily embedded into C++ applications. However, the current method of creating bindings for functions and classes is not very user-friendly. The current method requires users to write a lot of boilerplate code to create bindings for functions and classes, which can be tedious and error-prone.

pybind11 is a lightweight header-only library that provides a simple and elegant way to create bindings for functions and classes. It is wide adoption in the C++ community makes it the de-facto standard for creating Python bindings for C++ code.

If we can introduce a pybind11-compatible solution to pocketpy, existing pybind11 users can seamlessly port their bindings into pocketpy. New users can also benefit from this high-level binding solution.

Prototype

I have built a prototype for this proposal: basic implementation of pybind11.

```
struct Point {
  int x;
  int y;
  Point(int x, int y) : x(x), y(y) {}

  int sum(int z) { return x + y + z; }

  int sum(int a, int b) { return x + y + a + b; }
};

void register_point(module_ &m) {
  class_<Point>(m, "Point")
      .def(init<int, int>())
      .def_readwrite("x", &Point::x)
      .def_readwrite("y", &Point::y)
      .def("sum", static_cast<int (Point::*)(int)>(&Point::sum))
      .def("sum", static_cast<int (Point::*)(int, int)>(&Point::sum));
}
```

This prototype implements the commonly used interfaces from pybind11 and can be simply used to create bindings for functions and classes. However, it is not complete and has some limitations. There are still a lot of corner cases that need to be handled.

Implementation

This project includes two key aspects in its implementation: creating function bindings and creating class bindings.

Bind Functions

Suppose we have a C++ function that we want to call it from Python code:

```
int add(int a, int b) {
  return a + b;
}
```

Users may want to call this function in Python with the following syntax:

```
result = add(1, 2)
```

The Python interpreter cannot call the add function directly. Instead, before calling into C++ binding code, the Python interpreter wraps each function argument in a PyObject object, and calls the C++ binding code with an array of such objects. Since a Python function may accept an arbitrary number of parameters, the size of the argument array is unknown at compile time. Furthermore, due to Python being a dynamically typed language, the type of Python object wrapped in each PyObject object is also unknown at compile time. To allow eventual safe calls to the add function, we need a wrapper function that receives the argument array from the Python interpreter, performs necessary type checks on the arguments, and calls the `add` with proper C++ objects extracted from the argument array.

```
PyObject* call_wrapper(PyObject** args, size_t count){
    // check the underlying type of PyObject in runtime
    check_types<int, int>(args, count);

    // cast arguments and call C++ function
    int ret = add(args[0]->cast<int>, args[1]->cast<int>);

    // create a PyObject from int
    return py_create(ret);
}
```

The job of call_wrapper is simple: it checks the types of arguments and casts them to the correct types, then calls the native C++ function and returns the result as a PyObject.

However, it is impossible to write a wrapper function for every bound function by hand. We need a way to generate these wrapper functions automatically. This can be done by using template metaprogramming.

```
template <typename T>
struct member_fn_traits;

template <typename R, typename C, typename... Args>
struct member_fn_traits<R(*)(Args...)>{
    using return_type = R;
    using args_type = std::tuple<Args...>;
};

template <auto fn>
auto* type_ensure(){

    using traits = member_fn_traits<decltype(fn)>;
    using result_type = typename traits::return_type;
    using args_type = typename traits::args_type;

    return +[](PyObject** args, std::size_t count) -> PyObject* {
        check_types<args_type>(args, count);

        return [=]<std::size_t... Is>(std::index_sequence<Is...>) -> PyObject* {
            if constexpr (std::is_void_v<result_type>) {
                fn(args[Is].cast<std::tuple_element_t<Is, args_type>>()...);
                return py_create_none();
            } else {
                auto ret = fn(args[Is].cast<std::tuple_element_t<Is, args_type>>
()...);
                return py_create(ret);
            }
        }(std::make_index_sequence<std::tuple_size_v<args_type>>{});
    };
}
```

then we can use type_ensure to generate wrapper functions for any bound function.

```
auto add_wrapper = type_ensure<add>();
```

Another important thing is how to resolve overloaded functions. As we all know, C++ supports function overloading, allowing us to define multiple functions with the same name but different numbers and types of parameters. But Python does not support function overloading. So we need to find a way to resolve the overloaded functions.

We can build a dispatcher to solve this problem.

```
struct cxx_function{
    using func = std::function<PyObject*(PyObject**, size_t)>;
    std::vector<func> overload_set;

    PyObject* operator()(PyObject** args, size_t count) {
        for(auto& func : overload_set) {
            // check arguments compatibility
            // call the function ...
        }
        throw std::runtime_error("No matching function found");
    }
};
```

All overloaded functions will be stored in a `std::vector`. When a call is made, the dispatcher iterates through all functions, finds a matching function, and calls it, thus supporting function overloading. This also means that the function signature in Python must be in the form of `(*args, **kwargs)`. In pybind11, it iterates over the overloaded functions twice: the first iteration does not attempt implicit type conversion, while the second iteration does attempt implicit type conversion. If no matching function is found during both iterations, a `TypeError` exception is raised.

Bind Classes

Python and C++ have distinct object models. Consider the following example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(1, 2)
```

What happens behind the above code snippet? Let's illustrate with pseudocode.

```
PyObject* _1 = vm-create(1)
PyObject* _2 = vm-create(2)
PyObject* point = vm-new(Point)
point.attr("x") = _1
point.attr("y") = _2
```

A Python class instance is a dictionary-like container. It has a `__dict__` attribute that stores all the attributes of the instance. So `x` and `y` do not feature a compact arrangement. And the member access is just a dictionary lookup.

In contrast, an instance of a C++ class comprises a compact arrangement of its data members. A member access is done through memory address offsets rather than a dictionary lookup.

```
struct Point {
    int x;
    int y;
};

Point* p = new Point{1, 2};

// actually
// p = malloc(sizeof(Point));
// p + offsetof(Point, x) = 1;
// p + offsetof(Point, y) = 2;
```

Suppose we have bound the `Point` class to Python. How can we represent a `Point` instance in Python? If we still use a dictionary-like object, the high performance of C++ will be lost. We do not need to convert the layout, we just need to make the bound C++ class pretend to be a Python class. `x` and `y` will be the property of the class rather than the attribute of the instance.

We use the struct below to represent a C++ class instance.

```
struct CXXObject: PyObject {
    void* data;
    type_record* type;
    flags_t flags;
}
```

`data` is a pointer to the actual C++ class instance. `type_record` is a singleton object that stores the type info of the C++ class.

```cpp
struct type_record {
    std::string_view name;
    std::type_info& type;
    std::size_t size;
    std::size_t align;
    void (*destructor)(void*);
    void* (*copy)(void*, const void*);
    // ...
};
```

`flags` is used to store some special flags, the details will be discussed later.

now, let us bind the `Point` class to Python.

```cpp
class_<Point>(m, "Point")
    .def(init<int, int>())
    .def_property("x", &Point::x)
    .def_property("y", &Point::y);

// underlying code
// type cls = vm-new-type("Point")
// cls->attr("__init__") = py::function(init<int, int>)
// cls->attr("x") = py::function(&Point::x)
// cls->attr("y") = py::function(&Point::y)
```

`py::function` has been solved in the first part. It can make a C++ function callable in Python. Then just set the attributes of the class to the functions.

Then let's take a look at how to create a class in Python.

```python
p = Point(1, 2)

### underlying code
# p = vm-create CXXObject(Point)
# p->type = record_of<Point>()
# p->data = new Point(1, 2)
# p->flags = need_delete

x = p.x # property access

### underlying code
# x = vm-create CXXObject()
# x->type = record_of<int>()
# x->data = &p->data->x
# x->flags = no_delete
```

Now we can discuss the effect of the `CXXObject::flags` field because Python and C++ use fundamentally different ways of object memory and lifetime management. This can lead to issues when creating bindings for functions that return a non-trivial type. The return type does not specify whether Python should take charge of the returned value and eventually free its resources, or if this is handled on the C++ side.

pybind11 supports an `enum` called `return_value_policy` that can be passed to the `def` function to specify how the return value should be handled. This argument can be used to specify whether the Python virtual machine should take charge of the returned value or if the C++ side should handle it.

```
Point& get_point() {
    static Point p{1, 2};
    return p;
}

PYBIND11_MODULE(m) {
    m.def("get_point", &get_point, py::return_value_policy::reference);
}
```

```
p = get_point()
x1 = p.x
# p is a reference to the C++ object.
# x is a reference to the subobject of C++ object.
# and the garbage collector should not delete them.

p = Point(1, 2)
# p is created by the Python virtual machine.
# and the gc should delete the object.

x2 = p.x
# x2 is an internal reference to the object.
# garbage collector should not collect p before x is collected.
```

the flags will be different according to the `return_value_policy` argument.

Other

We've covered the main points earlier. Other aspects of the implementation, including type conversion rules and exception handling, are important but might feel repetitive. They're not hard, but they need careful attention when we're putting them into practice.

Also, because we're trying to achieve full compatibility with pybind11, we can use its tests as a guide and employ TDD methodology.

Development Process

Community Bonding

- Discuss the project with my mentors to clarify the project's goals and requirements.
- Set up a development environment and familiarize myself with the tools and libraries used in the project.

Week 1 - 3

- Write wrappers for Python built-in types, such as `int`, `float`, `str`, `list`, `tuple`, and `dict`.
- Write type cast function to verify whether the PyObject* can be cast to the C++ type.
- Support a basic interface for class binding, achieved through a class template `py::class_` to bind a C++ class to Python.

Week 4 - 6

- Implement support for binding functions, primarily using `py::def` to bind a C++ function to Python, overloading function also supported.
- Support all return value policies in pybind11, including `take_ownership`, `copy`, `move`, `reference`, `reference_internal`, `automatic`, `automatic_reference`.
- Support for binding member functions and properties, constructors, and destructors. Mainly implement the interface for `py::class_`, such as `def`, `def_property`, `def_readwrite`, `def_static`, etc.

Week 7 - 9

- Support for binding operators conveniently, such as `def(py::self == py::self)`, rather than `def("__eq__", &T::operator==)`.
- Support for binding enums.
- Support for class inheritance, including multiple inheritance and inheritance from Python classes.
- Support for exceptions.

Week 10 - 12

- Review the code beforehand and write unit tests for it.
- One of my mentors has a game engine project built with pybind11. I will assist them in migrating the codebase from pybind11 to pocketpy. This migration aims to ensure that the code remains compatible and functional in a production environment. If any incorrect behavior is detected, I will fix it.

Week 13

Buffer time for any unexpected issues.

Time Commitments

I will spend at least 30 hours per week on my project and can add more time if needed. My goal is to finish the GSoC project with nearly a week extra for finishing touches and testing. This extra time will also help with documenting my work and testing new features thoroughly.

Long Term Goals

I might not implement everything about pybind11 because it has lots of features. But I'll keep an eye on the project and help with any problems. Also, I will make easy-to-understand guides to help people start contributing to the project.

Acknowledgements and Excitement

Special thanks to @blueloveTH for their guidance and assistance throughout the development of this proposal. I would also like to express my gratitude to my mentors @lancern and @UnidayStudio for their invaluable feedback.

I'm eagerly looking forward to the upcoming summer break, where I'll have the opportunity to contribute to pocketpy. It's an exciting prospect, and I can't wait to collaborate with the developers in the PocketPy community to drive the project forward.