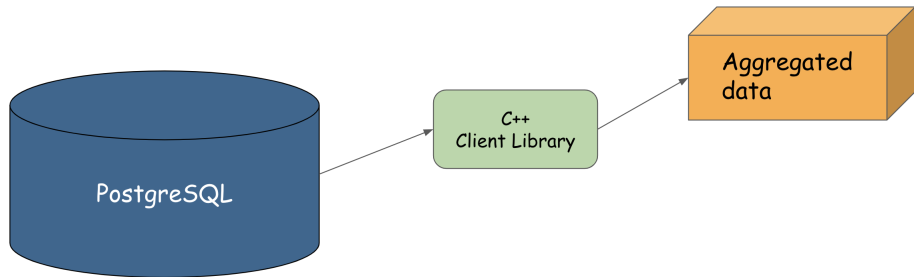
Фреймворк для автоматической генерации интерфейса библиотек с C++ на Python

Выполнил: Кучер Кирилл Владимирович, БПМИ216

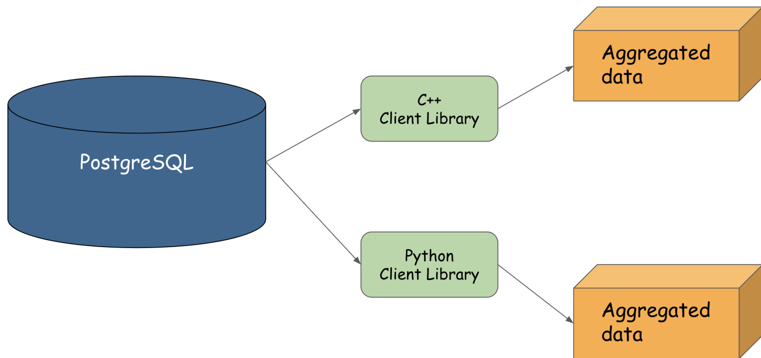
Научный руководитель: Хамитов Камиль Георгиевич, ООО "Яндекс Технологии"

Введение

Клиентская библиотека для агрегирования больших данных:



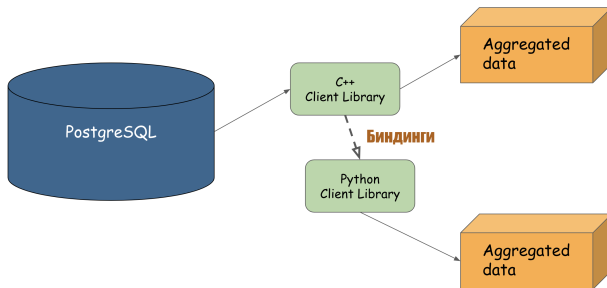
Как поддержать тот же интерфейс, но на Python для новых заказчиков?



Повторение всей логики обработки на python

Биндинги (*обвязки*) — механизмы, которые позволяют использовать объекты, написанные на одном языке программирования, в коде, написанном на другом языке.

Одна библиотека + обёртка для интерфейса на python



Как могла бы выглядеть структура с биндингами

Актуальность

Проблема: как поддерживать биндинги единообразно и без затрат для разработчика?

1. **Востребованность:** поддержка библиотеки агрегации логов для аналитиков в Яндексе; необходимость быстрых вычислений
2. **Нерешённость:** существующие решения не актуальны для свежих стандартов C++, сложная настройка и внедрение

Цель работы

Реализация фреймворка для автоматической генерации биндингов поверх C++ библиотек для Python-интерфейсов в соответствии со следующими критериями:

1. **Входные данные**:
   1. список папок, в которых располагаются .h-файлы с интерфейсом
   2. название входной библиотеки
2. **Выходные данные**: сгенерированный .so-файл с python-модулем

Поставленные задачи

1. Изучить варианты для синтаксического анализа классов и функций на C++
2. Изучить варианты для создания и генерации биндингов
3. Реализовать все логические части части фреймворка
4. Протестировать написанное приложение с разными сценариями использования и разными объектами в C++ коде

Обзор

Существующие решения

1. **CythonWrapper**
   - отсутствие поддержки генерации зависимых модулей
   - отсутствие поддержки стандартной библиотеки: умные указатели, контейнеры
   - отсутствие поддержки — последний коммит в репозиторий **6 лет назад**
2. **Binder**
   - ручная компиляция сгенерированных биндингов — полное отсутствие поддержки систем сборок, из-за чего нет гибкости для проектов под разными системами (CMake, ya.make)
   - ручная и долгая конфигурация: явное указание необходимых объектов для генерации — не подходит для больших библиотек

LLVM LibTooling — библиотека от создателей компилятора Clang, позволяющая писать произвольные инструменты для кода, например — анализ AST.

- Подробная документация
- Множество примеров использования

Библиотека для биндингов

Для биндингов был выбран фреймворк **pybind11**

Плюсы:

- Простота описания биндингов
- Поддержка относительно свежих объектов C++ (умные указатели)
- Header-only

Пример создания простого модуля с одной функцией

```
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional
        module docstring

    m.def("add", &add, "A function that adds two
        numbers");
}
```

Пример генерации кода

- Простота использования — генерация строк
- Нет внешних зависимостей

```
// m.def("foo", &foo)
void FoundFunction(const FunctionDecl* function)
    override
{
     functions_.emplace_back(std::format(
         "\n\tm.def(\"{}\", &{})",
         function->getNameAsString(),
         function->getNameAsString()
     ));
}
```

Архитектура программы

Программная часть

Программная часть

Всего 4 основных объекта в программе:

1. **RunGen**
2. **HeaderManager**
3. **ObjectFinder**
4. **Generator**

RunGen

Точка входа в программу, принимающая контекст исполнения.
Назначение: связывает между собой объекты, конфигурирует обход AST, запускает генерацию.

HeaderManager

Вспомогательный класс, необходимый для обхода .h-файлов с интерфейсом библиотеки.

Назначение: получение абсолютных путей до файлов с интерфейсом и списка include-файлов

```
HeaderManager(const std::vector<std::string>&
    source_dirs);

const std::string& GetIncludes() const;
const std::vector<std::string>& GetHeaders() const;
```

ObjectFinder

Класс, реализующий интерфейс обработки элемента из AST.
Хранит в себе контекст обхода.

Назначение: фильтрация, классификация и передача в *Generator* очередного объекта из AST.

```
ClassFinder(const std::string& outputPath, const
    std::string& moduleName, const HeaderManager&
    headerManager);

virtual void run(const MatchFinder::MatchResult&
    result) override;  // object from AST
```

Generator

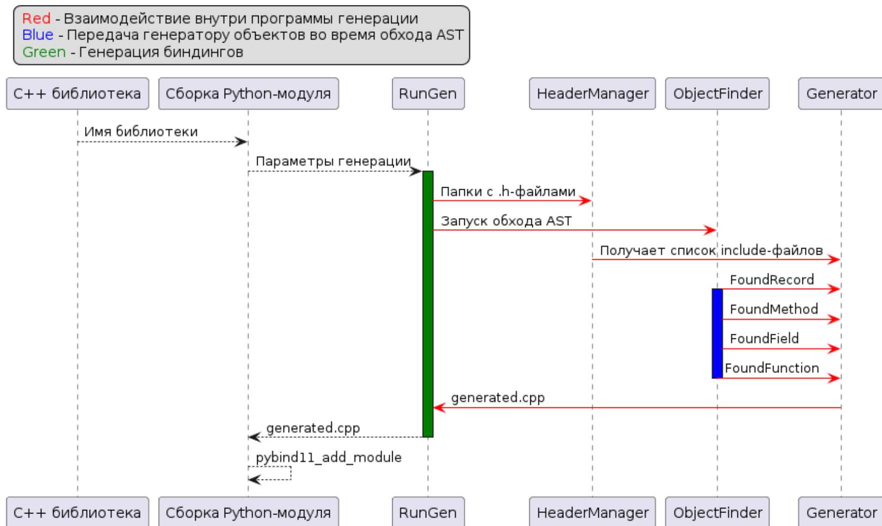
Класс, реализующий логику генерации кода.
Хранит в себе контекст генерации для каждого полученного объекта.

Назначение: получает очередной объект из AST, описывает биндинги, генерирует исполняемый файл.

```
void FoundRecord(const CXXRecordDecl* record);
void FoundConstructor(const CXXConstructorDecl* ctor);
void SetDefaultConstructor(const CXXRecordDecl*
    record);
void FoundField(const FieldDecl* field);
void FoundMethod(const CXXMethodDecl* method);

void FoundFunction(const FunctionDecl* function);
```

Взаимодействие объектов



Тестирование

- Написана тестовая библиотека под разные сценарии биндингов
- Для тестирования python-модуля выбран фреймворк **pytest**

```
// simple class in test_lib
class A {
public:
    int third{0};

private:
    void bar();
};
```

```
import py_test_lib

def test_simple_class():
    a = py_test_lib.A()
    assert a.third == 0
```

Тестирование

1. Простые классы (как на предыдущем слайде)
2. Статические функции
3. STL-контейнеры
4. Наследование, множественное наследование
5. Аннотации для игнорирования объекта
6. Перегрузка методов и функций
7. Вложенные структуры
8. Неймспейсы

Результаты

1. Изучены библиотеки для создания биндингов — выбран pybind11
2. Изучены варианты для синтаксического анализа классов и функций на C++ — выбран LLVM LibTooling
3. Фреймворк реализован и протестирован

Сверх плана: поддержана система сборки *ya.make*

Спасибо за внимание!