Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心

# 跨语言过程调用方法

门元昊

导师：罗森林

2022年10月23日

# 内容提要

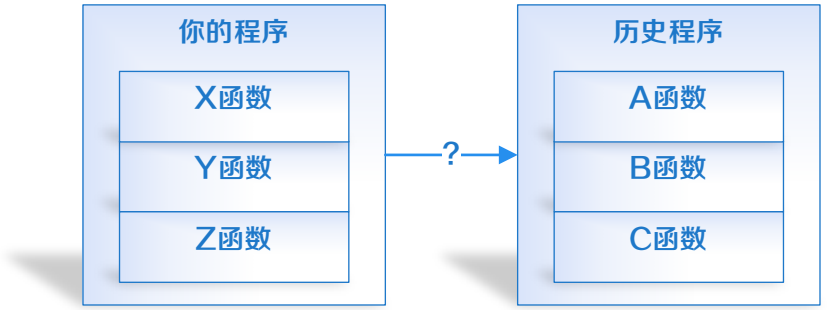
- 背景简介
- 基础概念
- 方法介绍
- 总结
- 参考文献

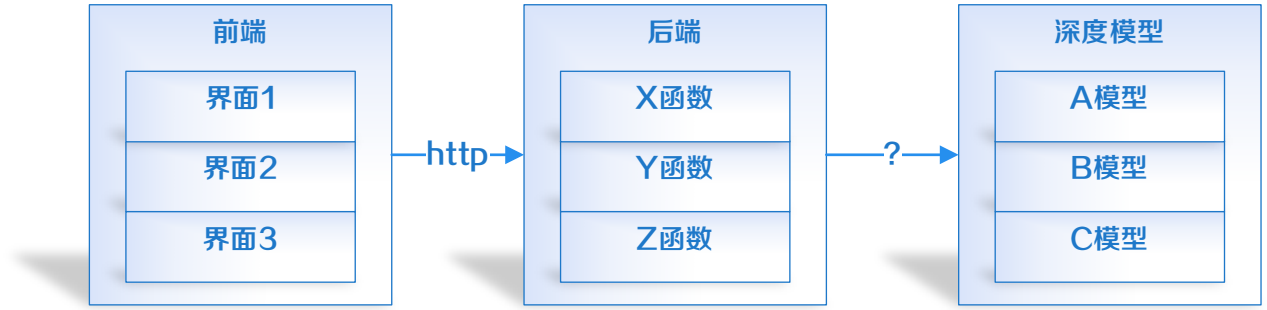
# 背景简介

- 预期收获
  - 1. 了解跨语言过程调用的应用场景
  - 2. 理解基于网络通信的跨语言调用方法
  - 3. 理解基于本地链接库的跨语言调用方法

# 背景简介

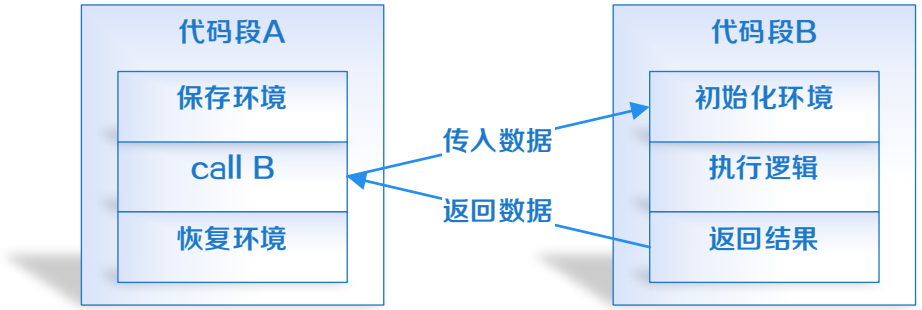
- 历史代码复用
  - 原有系统的二次开发
  - 师兄师姐的实验代码
- 多种环境需求
  - web框架+深度学习
- 知识产权保护
  - python的业务代码

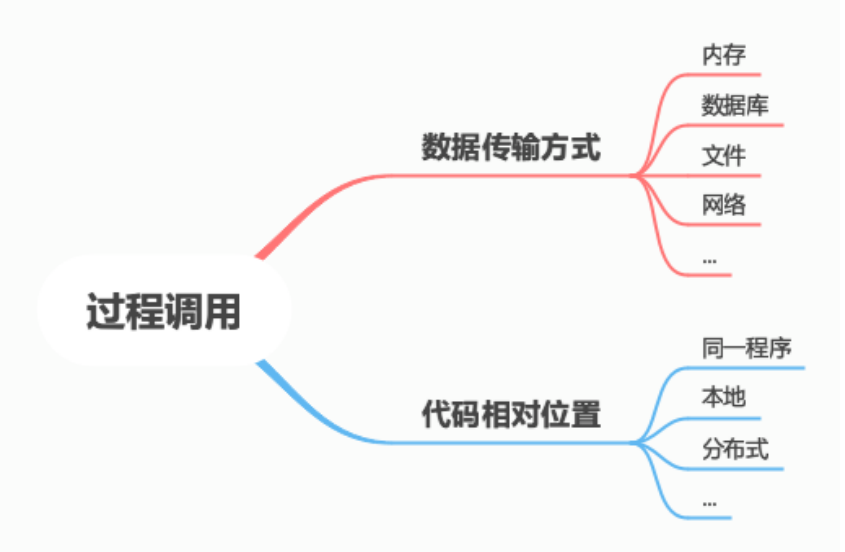


# 基本概念

- 过程调用
  - 调用：将程序执行的交给其他代码段，同时保存必要信息，从而保证被调用段执行完毕后能够返回调用点继续执行
  - 满足上述特性的操作，均可以用于不同代码段间的调用交互



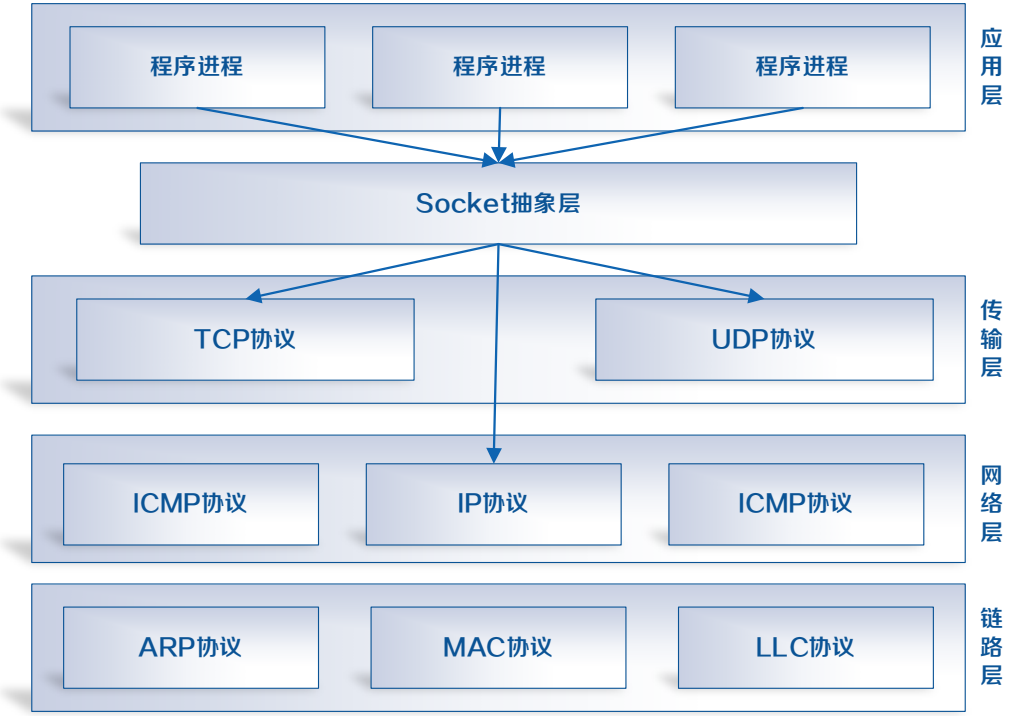
# 网络通信方式

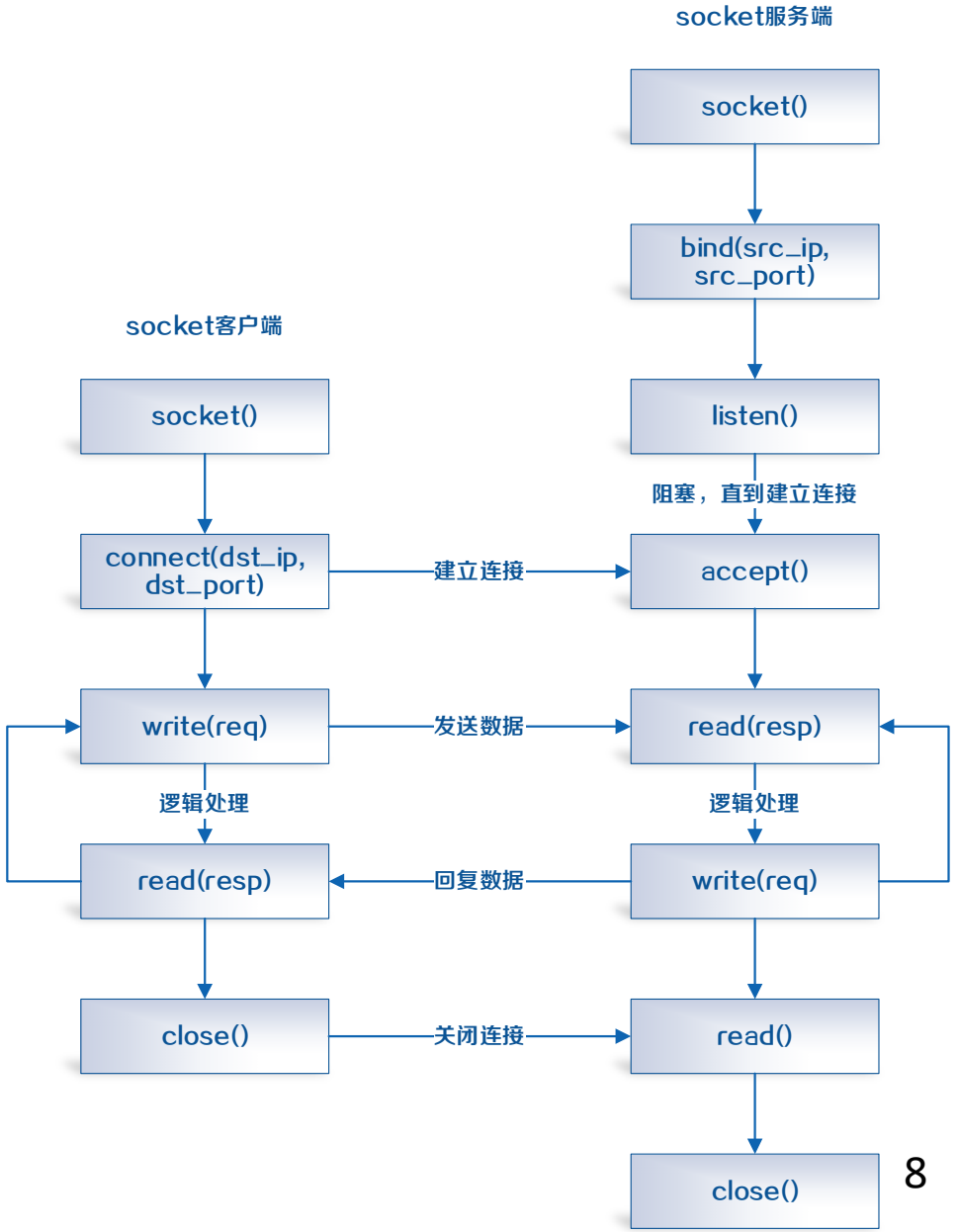
| | | |
|---|---|---|
| T | 目标 | 实现不同程序间的功能调用 |
| I | 输入 | 功能需求、功能代码 |
| P | 处理 | socket、http、rpc |
| O | 输出 | 具备跨程序功能调用能力的代码 |

| | | |
|---|---|---|
| P | 问题 | 不同程序间的功能无法直接相互调用 |
| C | 条件 | 程序源码已知，网络连通 |
| D | 难点 | 数据格式解析、网络调试 |
| L | 水平 | |

# Socket

- socket（套接字）
  - 网络抽象层，帮助程序在网络中收发数据
  - 允许程序如操作文件般打开、读写、关闭相关的网络IO



# Socket

- 服务端构建
  - bind函数
    - 设置监听ip与端口
  - listen函数
  - accept函数
    - 接受客户端连接
    - 获取客户端ip
    - 建立双向通信链路

```
bind port: 10086
listen started

connect estabulished, starting recieving
recieving data
recieved data: test
recieving data
recieved data: test
recieving data
recieved data: test
```

```c
// 创建socket
socket_handler = socket(AF_INET, SOCK_STREAM, 0);
// 设置监听地址
struct sockaddr_in server_sockaddr;
server_sockaddr.sin_family = AF_INET;
server_sockaddr.sin_port = htons(PORT);
server_sockaddr.sin_addr.s_addr = htonl(INADDR_ANY);
// 端口绑定
bind(socket_handler, (struct sockaddr*)&server_sockaddr, sizeof(server_sockaddr));
// 监听连接
listen(socket_handler, 20);
// 建立连接
struct sockaddr_in client_addr;
socklen_t length = sizeof(client_addr);
conn = accept(socket_handler, (struct sockaddr*)&client_addr, &length);
// ...
while(1) {
    // ..
    recv(conn, buffer, sizeof(buffer), 0);
    // ...
    send(conn, buffer, len , 0);
    // ..
}
// ...
```

# Socket

- 具体实现
  - 客户端构建
    - 构建socket
    - 连接服务端
      - connect函数
    - 实施通信
      - 数据编码

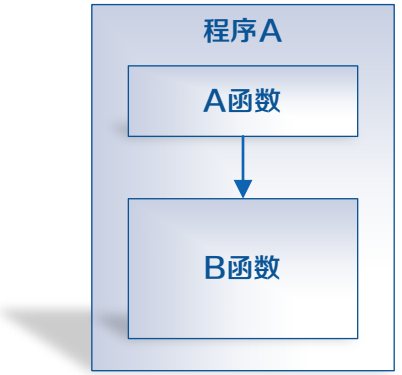
```
输入待发送数据:test
发送成功!
返回数据: b'test'
输入待发送数据:test
发送成功!
返回数据: b'test'
输入待发送数据:test
发送成功!
返回数据: b'test'
```

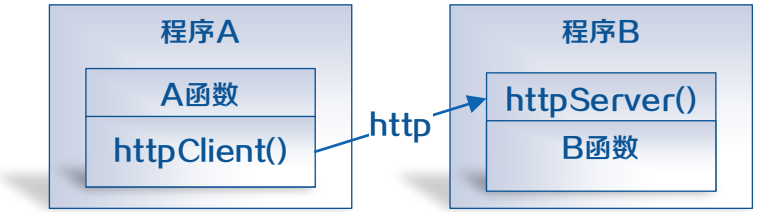
```python
import socket
if __name__ == "__main__":
    HOST = 'localhost'
    PORT = 10086
    ADDR = (HOST, PORT)
    ENCODING = 'utf-8'
    BUFFSIZE = 1024
    with socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM) as s:
        s.connect(ADDR)
        while True:
            req = input('输入待发送数据:')
            if req == 'q':
                break
            # 发送数据
            req = '{}'.format(req)
            s.send(req.encode(ENCODING))
            print('发送成功! ')
            # 接收数据
            resp = s.recv(BUFFSIZE)
            print('返回数据: {}'.format(resp))
        # 关闭
        s.close()
```

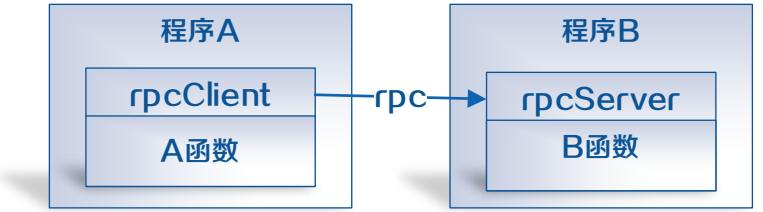
# Http

- Http协议
  - 超文本传输协议（Hyper Text Transfer Protocol）
  - 设计初衷是用于从www服务器传输超文本数据到本地浏览器
  - 其规定了一整套客户端和服务器的请求与应答标准
- Http特性
  - 请求-响应模式
    - 由客户端发出请求，服务器响应请求并返回
    - 服务器无法主动推送
  - 无状态
    - 协议本身不保存请求响应间的通信状态
  - 无连接
    - 每次连接仅处理一个请求
    - 1.1版本后允许处理多个请求

- 程序过程调用
  - 同一程序
    - 函数调用：本地、同语言、无缝调用
  - 不同程序
    - socket?
      - 数据格式需自定义
      - 调试相对复杂
    - http?
      - 通信开销高（3次握手）
      - 复杂度提升（路由处理、专门的请求和响应函数等）
    - rpc?

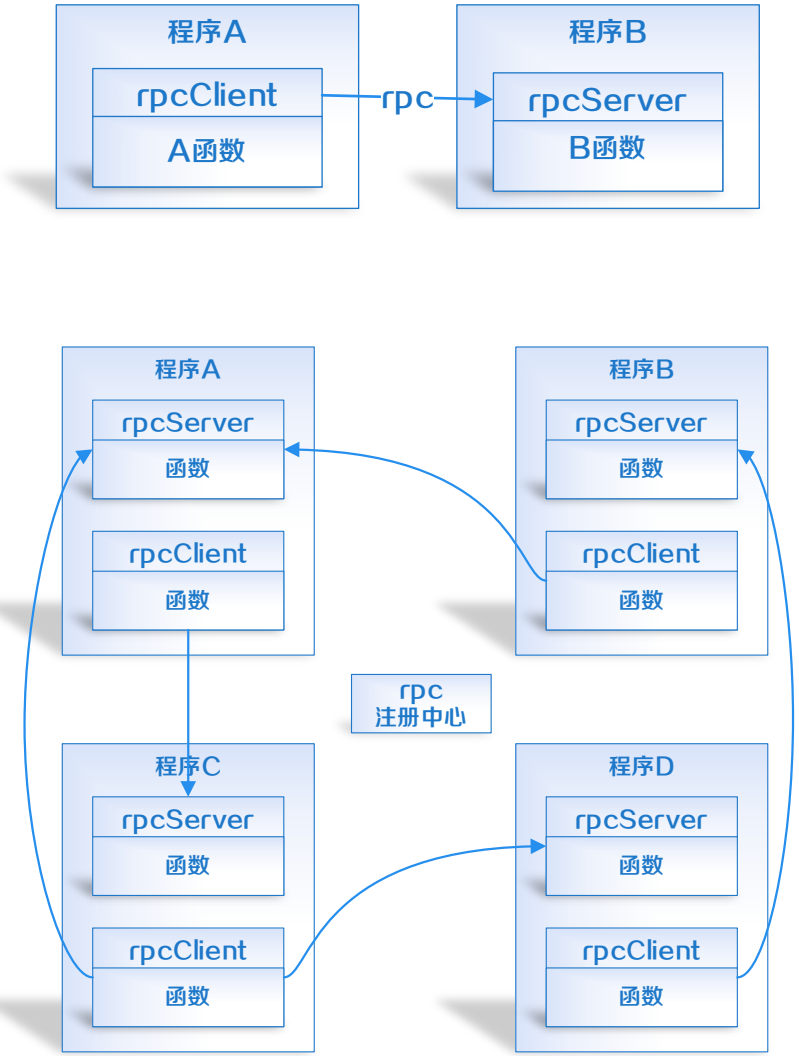


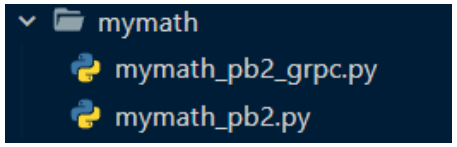
# RPC

- rpc，远程过程调用
  - 简化形式，尽可能接近本地调用
  - 降低开销，无需http的多次握手
  - 管理统一，有注册中心进行监控管理
  - 实现方便，可选择多种协议实现
- rpc相关实现
  - grpc
    - 基于protobuf的rpc实现
  - Thrift
  - Dubbo



# gRPC实战

- 编写接口文件
  - 定义服务
  - 定义函数
    - 限制传出和返回的参数数量各1个
  - 定义数据
    - protobuf支持常见的数据类型
    - 允许自定义类型和嵌套
- 生成接口代码
  - 利用grpcio-tools工具
  - 不同语言有不同生成工具




```proto
syntax = "proto3";
package mymath;
// 定义服务与函数
service MyMath {
    rpc add(AddRequest) returns (AddResponse) {}
}
// 定义通信数据
message Point {
    float x = 1;
    float y = 2;
}
message AddRequest {
    Point a = 1;
    Point b = 2;
}
message AddResponse {
    Point res = 1;
}
```

# gRPC实战

- 服务端构建
  - 导入接口代码文件
    - xxx_pb2：符号库
    - xxx_pb2_grpc：类库
  - 编写服务功能代码
    - 继承xxxServicer类
    - 实现接口文件中的函数
  - 配置rpc监听服务
    - 注册服务
    - 配置ip端口
  - 启动rpc服务

```python
import grpc
from concurrent import futures

import mymath_pb2 as mymath
import mymath_pb2_grpc as mymath_grpc


class MyMathServicer(mymath_grpc.MyMathServicer):
    def add(self, req, ctx):
        res_x = req.a.x + req.b.x
        res_y = req.a.y + req.b.y
        print(f'({req.a.x}, {req.a.y})')
        print(f'({req.b.x}, {req.b.y})')

        res = mymath.Point(x=res_x, y=res_y)
        print(f'({res.x}, {res.y})')
        return mymath.AddResponse(res=res)


def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    mymath_grpc.add_MyMathServicer_to_server(MyMathServicer(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()


if __name__ == '__main__':
    serve()
```

# gRPC实战

- 客户端构建
  - 导入接口代码文件
  - 连接rpc服务
    - 实例化服务客户端
  - 调用rpc服务函数
    - 传入参数格式
      - 建议按关键字传参
    - 返回参数格式

```
连接到rpc服务
传入参数:
    (1.0, 2.0)
    (3.0, 4.0)
接收结果:
    (4.0, 6.0)
```

```
开始监听...
接收到调用
传入参数:
    (1.0, 2.0)
    (3.0, 4.0)
返回结果:
    (4.0, 6.0)
```

```python
import grpc

import mymath.mymath_pb2 as mymath
import mymath.mymath_pb2_grpc as mymath_grpc


if __name__ == '__main__':
    # 连接到rpc服务
    with grpc.insecure_channel('localhost:50051') as channel:
        # 建立mymath服务
        stub = mymath_grpc.MyMathStub(channel)
        # 准备输入数据
        point_a = mymath.Point(x=1, y=2)
        point_b = mymath.Point(x=3, y=4)
        print(f'({point_a.x}, {point_a.y})')
        print(f'({point_b.x}, {point_b.y})')
        req = mymath.AddRequest(a=point_a, b=point_b)
        # 调用服务并接收回传数据
        res = stub.add(req)
        print(f'({res.res.x}, {res.res.y})')
```

# 优劣分析

| 方法 | socket | http | rpc |
| --- | --- | --- | --- |
| 优势 | 1. 轻量，通信资源开销最小<br>2. 语言支持性好，几乎所有支持网络编程的语言都直接支持socket | 1. 生态好，框架丰富，教程丰富，上手容易<br>2. 数据接口有指导标准，不容易混乱，适合协同开发<br>3. 辅助工具较多，便于调试 | 1. 近乎本地化的调用方法<br>2. 调用接口固定，不存在标准混乱的情况<br>3. 资源占用与易用性的平衡 |
| 劣势 | 1. 编写麻烦，少有框架辅助，基本都需要自己造轮子<br>2. 缺少数据接口标准，难以在进行标准化管理<br>3. 调试较复杂，需要用到tcp调试工具<br>4. 跨平台兼容性较差，win与linux的依赖存在差异，环境配置较麻烦 | 1. 通信资源开销大<br>2. 程序复杂度提升，需要引入一整套http框架 | 1. 需要新掌握一种接口定义语言，学习难度高于http<br>2. 语言支持性弱于socket<br>3. 接口文件需要专用工具处理翻译为对应的代码 |
| 适用场景 | 小型程序、资源开销要求严格等 | 相互调用、通信需求少或频率低的程序 | 优先推荐使用 |

# 本地调用方式

| | | |
|---|---|---|
| T | 目标 | 将c/c++代码提供给python调用 |
| I | 输入 | c/c++代码 |
| P | 处理 | ctypes/pybind11 |
| O | 输出 | python库文件/动态链接库 |

| | | |
|---|---|---|
| P | 问题 | 现有程序为c/c++编写，无法使用python直接调用 |
| C | 条件 | 程序源码已知 |
| D | 难点 | 数据类型的对应与转换 |
| L | 水平 | |

# Ctypes方法

- ctypes
  - 提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数
  - 可使用该模块以纯 Python 形式对这些库进行封装
  - python自带（2.5+）
- 基本原理
  - 通过调用对应平台的动态链接库加载方法，从而获取到库中的声明函数与变量
  - 基于一套类型映射规则，实现python类型与C类型的转化
  - 不同平台下实现方式不同
    - Windows：LoadLibrary()、GetProcAddress()
    - Linux/Mac：dlopen()、dlsym()

# Ctypes方法

- ctypes使用流程
  - C阶段
    - 编译动态链接库
      - gcc
  - Python阶段
    - 加载动态链接库
    - 调整包装输入变量
    - 设置返回值类型
    - 调用函数

```c
#include <stdio.h>
#include "cmymath.h"

float c_multiple(float param_a, float param_b) {
    float return_value = param_a * param_b;
    printf("In c_multiple : a: %.2f b: %.2f returning %.2f\n", param_a, param_b,
return_value);
    return return_value;
}

float c_divide(float param_a, float param_b) {
    float return_value = param_a / param_b;
    printf("In c_divide : a: %.2f b: %.2f returning %.2f\n", param_a, param_b,
return_value);
    return return_value;
}
```

推荐使用基本数据类型
struct数据类型难以debug

# Ctypes方法

- ctypes使用流程
  - C阶段
    - 编译动态链接库
  - Python阶段
    - 加载动态链接库
    - 调整包装输入变量
      - argstype函数
      - 如何处理自定义类型?
    - 设置返回值类型
    - 调用函数

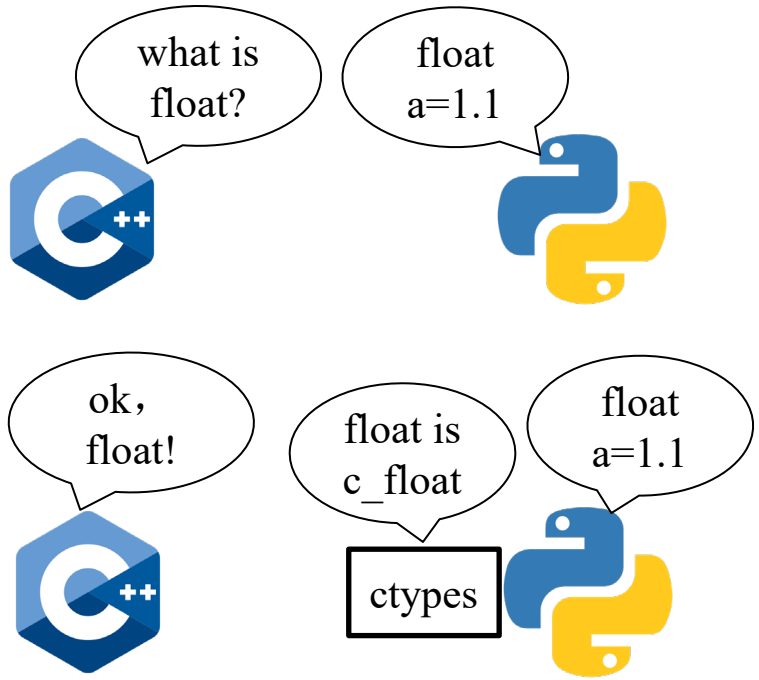
```python
import ctypes
import pathlib

if __name__ == "__main__":
    # 加载动态链接库
    libname = pathlib.Path().absolute() / "c_version" / "libcmymath.so"
    c_libcmymath = ctypes.CDLL(libname)
    # 调整并包装输入变量
    x, y = 6, 2.3
    a, b = ctypes.c_float(x), ctypes.c_float(y)
    # 设置返回值类型
    c_libcmymath.c_multiple.restype = ctypes.c_float
    # 调用函数
    answer = c_libcmymath.c_multiple(a, b)
    print(f"In Python: a: {x:.2f} b: {y:.2f} return val {answer:.2f}")
    # 设置返回值类型
    c_libcmymath.c_divide.restype = ctypes.c_float
    # 调用函数
    answer = c_libcmymath.c_divide(a, b)
    print(f"In Python: a: {x:.2f} b: {y:.2f} return val {answer:.2f}")
```

# Ctypes方法

- ctypes常见问题
  - 变量类型
    - ctypes默认将所有变量类型视为整型
      - 直接输入变量x，y

```
(common) men@men ~/workplace/python/c2python/ctypes python ctypes_test.py
Traceback (most recent call last):
  File "/home/men/workplace/python/c2python/ctypes/ctypes_test.py", line 14, in <module>
    answer = c_libcmymath.c_multiple(x, y)
ctypes.ArgumentError: argument 2: <class 'TypeError'>: Don't know how to convert parameter 2
```




    - 非整型变量都需预先包装
      - ctypes给出了部分与C语言兼容的基本数据类型
        - » 布尔：c_bool
        - » 字符：c_char、c_wchar
        - » 整型：c_byte、c_short、c_int、c_long
        - » 浮点：c_float、c_double、c_longdouble
        - » 字符串：c_char_p、c_wchar_p

# Ctypes方法

- ctypes常见问题
  - 整型截断问题
    - python整型类型几乎涵盖了C语言从char类型到long long int类型的全部范围
    - 在调用过程中，需要人为考虑和处理变量取值带来的截断问题

```
In Python: var: 100000
In c_print_var:
        char: -96
        short: -31072
        int: 100000
        long: 100000
```

```c
void c_print_var(
    char param_char, short param_short,
    int param_int, long param_long) {
    printf(
        "In c_print_var:\n\tchar: %d\n\tshort: %d\n\tint: %d\n\tlong: %ld\n",
        param_char, param_short, param_int, param_long);
}
```

```python
if __name__ == "__main__":
    pyvar_int = 100000
    cvar_char = ctypes.c_byte(pyvar_int)
    cvar_short = ctypes.c_short(pyvar_int)
    cvar_int = ctypes.c_int(pyvar_int)
    cvar_long = ctypes.c_long(pyvar_int)
    print(f"In Python: var: {pyvar_int}")
    c_libcmymath.c_print_var(cvar_char, cvar_short, cvar_int, cvar_long)
```

# Pybind方法

- pybind11
  - Seamless operability between C++11 and Python
  - C++11与Python的无缝可操作性
  - 轻量级C++库，能够将C++代码暴露给Python进行调用
- 安装方法
  - Github下载编译（推荐）
    - git clone https://github.com/pybind/pybind11.git
    - cd pybind11 && mkdir build && cd build && cmake ..
    - make check -j 4 && make install
  - python安装
    - pip install pybind11

# Pybind方法

- pybind11使用流程
  - C++阶段
    - 编写程序
      - 函数重载
      - 泛型模板
    - 编译动态链接库
  - Python
    - 加载动态链接库
    - 调用函数

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

int add(int i, int j) {
    printf("call function int add(int i, int j)\n");
    return i + j;
}

float add(float i, float j) {
    printf("call function float add(float i, float j)\n");
    return i + j;
}

int add(int i, int j, int k) {
    printf("call function int add(int i, int j, int k)\n");
    return i + j + k;
}

PYBIND11_MODULE(mylib, m) {
    m.def("add", pybind11::overload_cast<int, int>(&add));
    m.def("add", pybind11::overload_cast<float, float>(&add));
    m.def("add", pybind11::overload_cast<int, int, int>(&add));
}
```

# Pybind方法

- pybind11使用流程
  - C++阶段
    - 编写程序
    - 编译动态链接库
  - Python
    - 加载动态链接库
    - 调用函数

```
function int add(int i, int j)
3
function float add(float i, float j)
3.0
function int add(int i, int j, int k)
6
```

```python
import mylib

if __name__ == "__main__":
    print(mylib.add(1, 2))
    print(mylib.add(1.0, 2.0))
    print(mylib.add(1, 2, 3))
```

```
NAME
    mylib

FUNCTIONS
    add(...) method of builtins.PyCapsule instance
        add(*args, **kwargs)
        Overloaded function.

        1. add(arg0: int, arg1: int) -> int

        2. add(arg0: float, arg1: float) -> float

        3. add(arg0: int, arg1: int, arg2: int) -> int

FILE
    /home/men/workplace/python/c2python/pybind/cpp_version/mylib.so
```

# 优劣分析

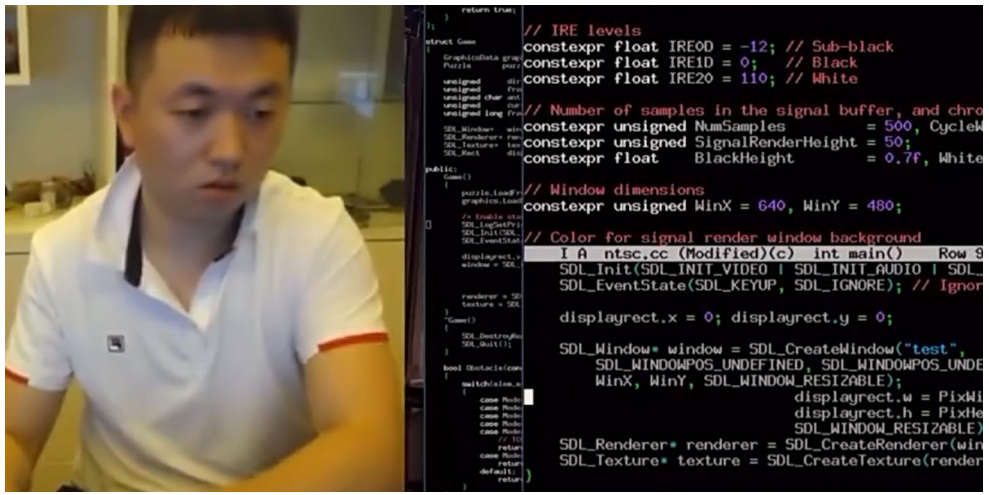
| 方法 | ctypes | pybind11 |
| --- | --- | --- |
| 优势 | 1. Python内置库，无需编译安装<br>2. 可直接调用动态链接库，无需重新修改和编译C源码 | 1. C++支持性好<br>2. Python调用方便，直接使用import即可 |
| 劣势 | 1. 非基本数据类型转换复杂，支持性差<br>2. 只能调用动态链接库中已声明导出的函数或变量<br>3. 仅对C语言支持程度较高，对C++相关特性支持性差 | 1. 环境配置较繁琐，依赖库需要自行编译安装<br>2. 需要修改和调整C++代码<br>3. 对C++14之后新特性的支持较弱 |
| 适用场景 | 纯C代码环境，数据类型简单 | C++环境，变量类型和函数调用需求复杂 |

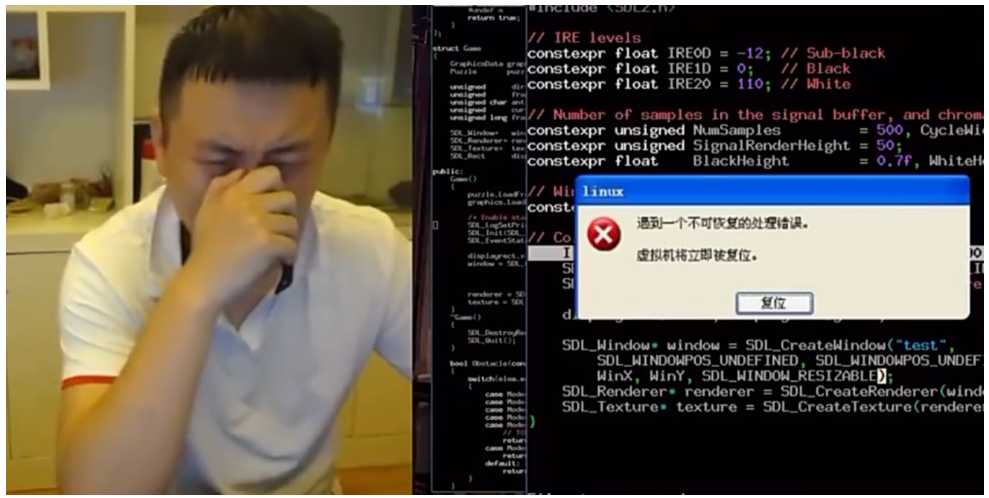
# 总结

# 总结

- 跨语言过程调用
  - 优先推荐使用rpc方法，简单易用且稳定性高
  - 如果必须要以本地调用形式，推荐使用pybind11

- 终极提醒
  - 尽可能避免出现跨语言调用的场景
  - 总会遇到各种各样的错误



# 参考文献


- https://grpc.io/docs/
- https://docs.python.org/zh-cn/3/library/ctypes.html
- https://pybind11.readthedocs.io/en/stable/index.html

# 道德经

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！