Yazi-RPC: 一个高性能的RPC框架

介绍

框架特性

1. 操作系统：Linux
2. 编程语言：C++14
3. 完全独立：不依赖任何第三方库
4. 高性能：微秒级响应
5. 高并发：单机百万连接
6. IO多路复用：epoll
7. 连接池
8. 线程池
9. 用法简单

服务端

文件：server.cpp

```cpp
#include <iostream>
using namespace std;

#include "Server.h"
using namespace yazi::rpc;

string hello(const string & name)
{
    return "hello, " + name;
}

int main()
{
    Server * server = Singleton<Server>::instance();
    server->listen("127.0.0.1", 8080);
    server->bind("hello", hello);
```

```
    server->start();

    return 0;
}

```

客户端

文件：client.cpp

```cpp
#include <iostream>
using namespace std;

#include "Client.h"
using namespace yazi::rpc;

int main()
{
    Client client;
    client.connect("127.0.0.1", 8080);

    auto reply = client.call<string>("hello", "kitty");
    std::cout << reply << std::endl;

    return 0;
}
```

更多示例

示例1：

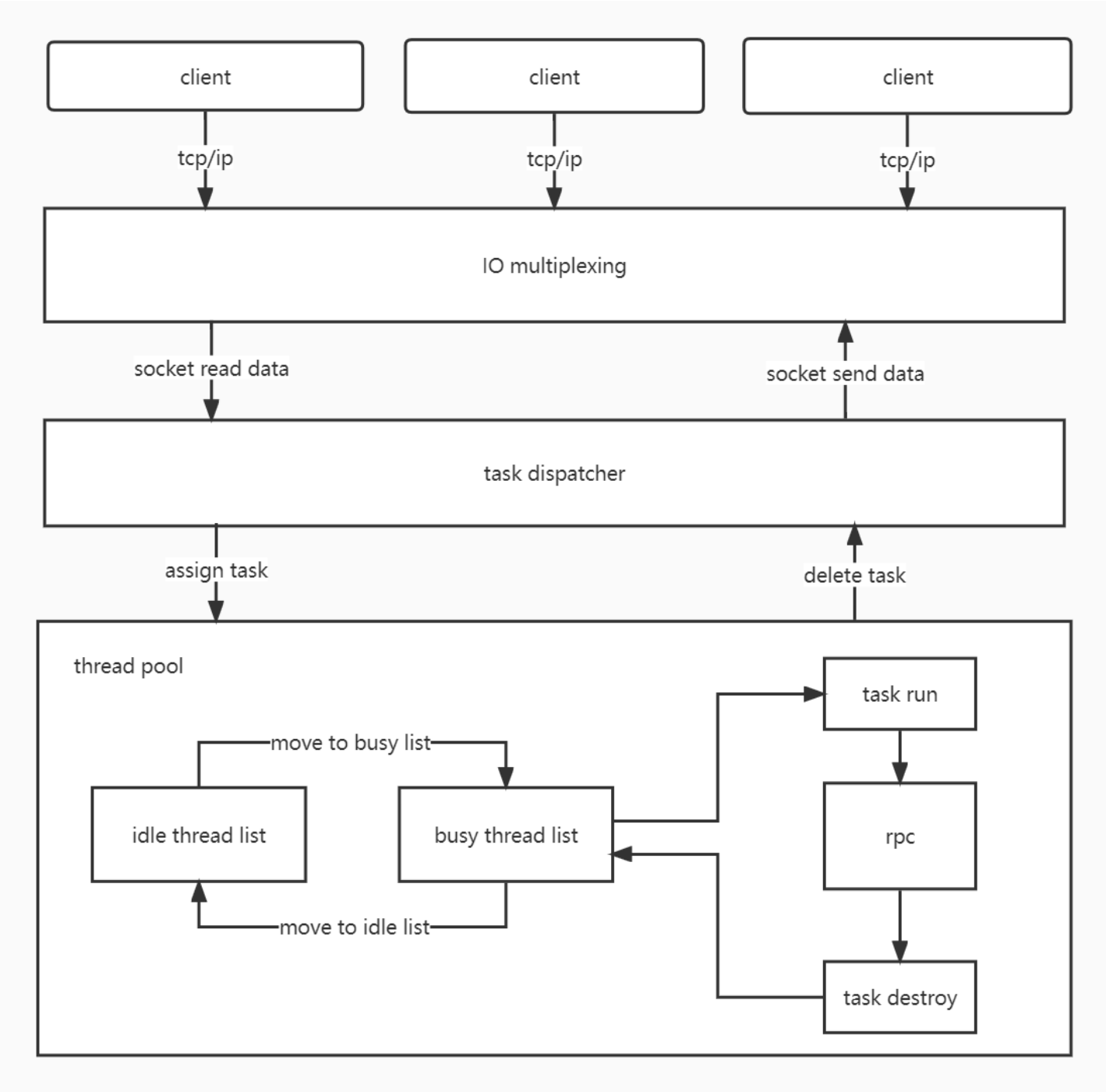
```cpp
int sum(int a, int b)
{
    return a + b;
}
```

示例2：

```cpp
#include <string>
#include <algorithm>
using namespace std;

class Request : public Serializable
{
public:
    Request() {}
    Request(const string & name) : m_name(name) {}
    ~Request() {}
    const string & name() const
    {
        return m_name;
    }
    SERIALIZE(m_name)
private:
    string m_name;
};

class Response : public Serializable
{
public:
    Response() {}
    Response(const string & name) : m_name(name) {}
    ~Response() {}
    const string & name() const
    {
        return m_name;
    }
    SERIALIZE(m_name)
private:
    string m_name;
};

Response upper(const Request & req)
{
    string name = req.name();
    transform(name.begin(), name.end(), name.begin(), ::toupper);
    return Response(name);
}
```

架构设计

整体架构



- IO多路复用模块：epoll
- 任务分发模块：task dispatcher
- 线程池：thread pool
- 任务执行模块：work task

代码结构

关键问题

1、高性能、高并发的网络框架

网络框架：yazi

https://www.bilibili.com/video/BV1hV4y1J7Ls/

2、客户端可变参数序列化

序列化组件：yazi-serialize

https://www.bilibili.com/video/BV1ad4y1x7VY/

3、服务端解包到不定参数列表

参考：buttonrpc

https://gitcode.net/mirrors/button-chen/buttonrpc_cpp14

需要用到c++14的特性

封装：rpc/FunctionHandler.h

关键代码：

```
template<typename R, typename F, typename Tuple>
typename std::enable_if<!std::is_same<R, void>::value, R>::type
FunctionHandler::call_impl(F func, Tuple args)
{
    return invoke<R>(func, args);
}

template<typename R, typename F, typename Tuple>
auto FunctionHandler::invoke(F && func, Tuple && t)
{
    constexpr auto size = std::tuple_size<typename std::decay<Tuple>::type>::val
    return invoke_impl<R>(std::forward<F>(func), std::forward<Tuple>(t), std::ma
}

template<typename R, typename F, typename Tuple, std::size_t... Index>
auto FunctionHandler::invoke_impl(F && func, Tuple && t, std::index_sequence<Ind
{
    return func(std::get<Index>(std::forward<Tuple>(t))...);
}

```

```cpp
template<typename Tuple, std::size_t... I>
Tuple FunctionHandler::get_args(DataStream & ds, std::index_sequence<I...>)
{
    Tuple t;
    initializer_list<int>{((get_arg<Tuple, I>(ds, t)), 0)...};
    return t;
}

template<typename Tuple, std::size_t Id>
void FunctionHandler::get_arg(DataStream & ds, Tuple & t)
{
    ds >> std::get<Id>(t);
}
```

注意：gcc 5及以上版本才支持 c++14

完结