高级篇 -- Yazi：一个百万并发的C++服务框架

课程大纲

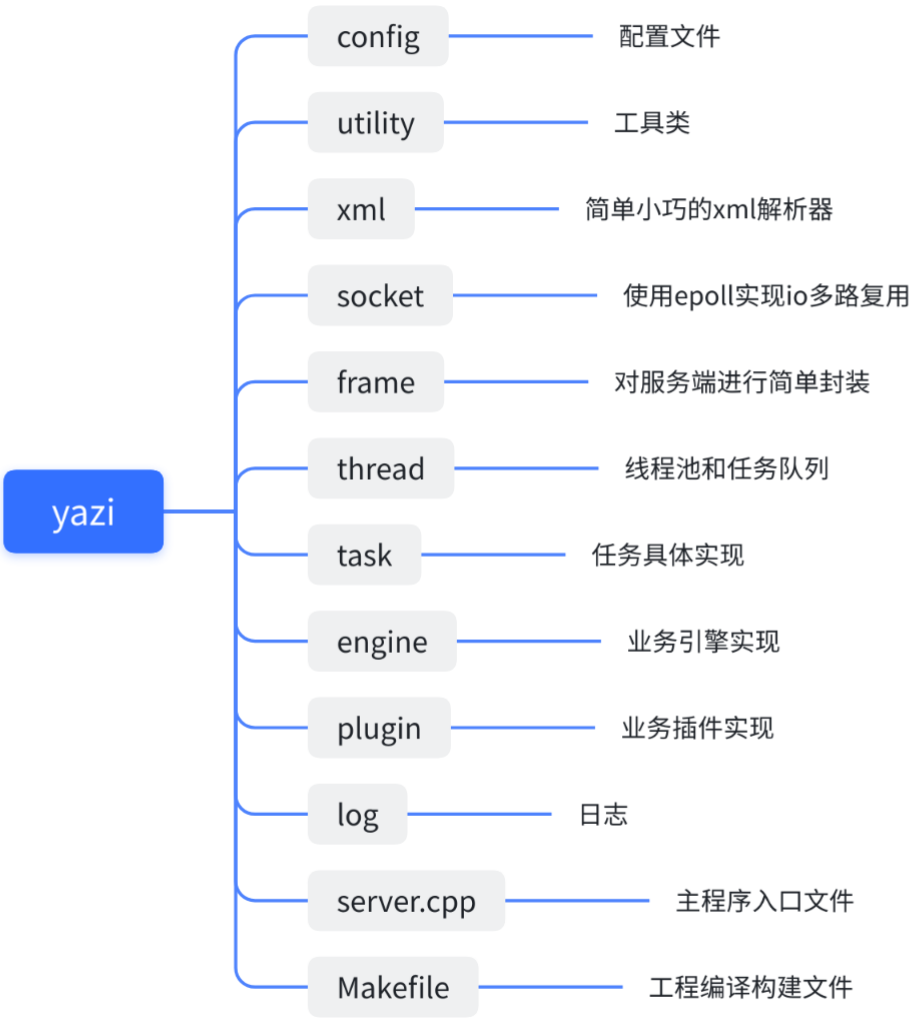
- 框架介绍
- 开发环境搭建
- 框架设计与实现
- 性能测试

框架介绍

框架特性

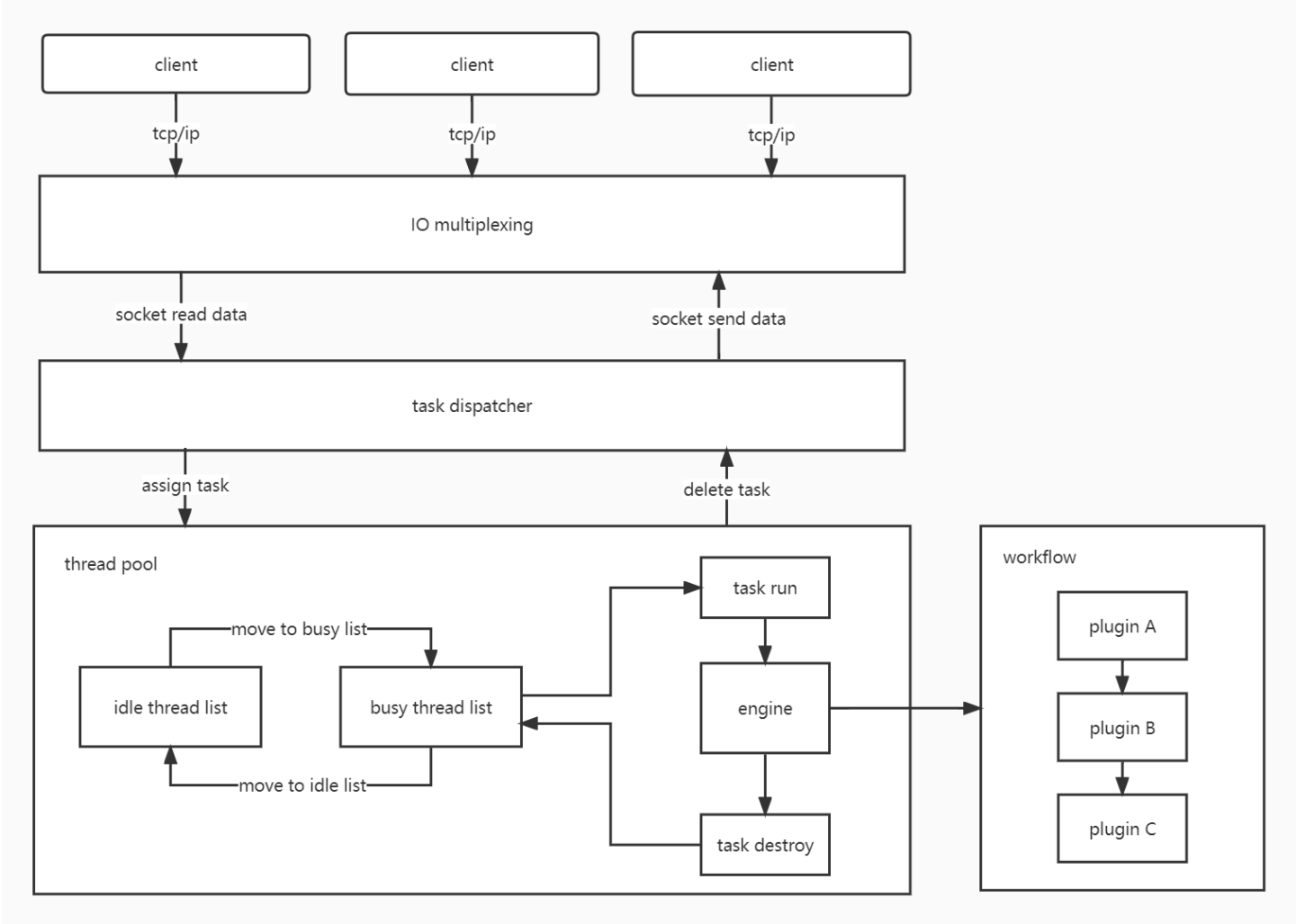
1. 操作系统：Linux
2. 编程语言：C++11
3. 高并发：单机百万连接
4. 高性能：微秒级响应
5. IO多路复用：epoll
6. 线程池
7. 任务队列
8. 业务引擎：插件

代码结构



架构设计

Reactor + 多线程模型



主线程（SocketHandler）

1. 监听和建立客户端的连接；
2. 接收客户端的请求，创建一个任务，并把该任务放入任务队列；
3. 告诉分发线程，有请求任务过来了，叫他赶紧去处理；
4. 重复以上三个步骤；

注意：主线程不处理具体请求。

分发线程（TaskDispatcher）

1. 查看任务队列，看是否有请求任务？没有任务则继续睡觉，否则把任务取出来，然后分发给线程池；
2. 线程池有空闲的线程，则把该任务交给空闲的线程处理，否则该任务乖乖呆在队列里等待，直到有空闲的线程为止；
3. 重复以上两个步骤；

注意：分发线程也不处理具体请求。

工作线程（WorkerThread）

1. 执行任务；
2. 销毁任务；
3. 重复以上两个步骤；

注意：工作线程处理具体请求。

开发环境

- 云服务器
- 虚拟机：VMware
- Docker

准备材料

1、Dockerfile：构建 centos 系统

```
FROM centos:7.8.2003

MAINTAINER oldjun <oldjun@sina.com>

RUN yum install -y initscripts
RUN yum install -y gcc
RUN yum install -y gcc-c++
RUN yum install -y kernel-devel
RUN yum install -y make
RUN yum install -y wget
RUN yum install -y vim-enhanced
RUN yum install -y net-tools
RUN yum install -y openssh
RUN yum install -y openssh-server
RUN yum install -y openssl-devel
```

```
RUN yum install -y ncurses-devel
RUN yum install -y sqlite-devel
RUN yum install -y readline-devel
RUN yum install -y libffi-devel
RUN yum install -y git
RUN echo "root:password"|chpasswd
RUN ssh-keygen -A

ADD Python-3.9.6.tar.xz /root/

RUN cd /root/Python-3.9.6 && \
    ./configure prefix=/usr/local/python3 && \
    make && \
    make install

RUN ln -s /usr/local/python3/bin/python3 /usr/bin/python3
RUN ln -s /usr/local/python3/bin/pip3 /usr/bin/pip3

EXPOSE 22
```

2、Python-3.9.6.tar.xz

下载地址：https://www.python.org/downloads/release/python-396/

构建镜像

在windows终端，运行命令：

```
docker build -t centos .
```

启动容器

在windows（管理员）终端，运行命令：

```
docker run -it -p 22:22 -v D:\yazi\yazi:/root/yazi --name centos centos
```

本地的项目代码自动挂载到docker容器里：D:\yazi\yazi --> /root/yazi

编译框架

```
make clean
make
```

编译插件

```
make plugin
```

启动服务

```
./server &
```

关闭服务

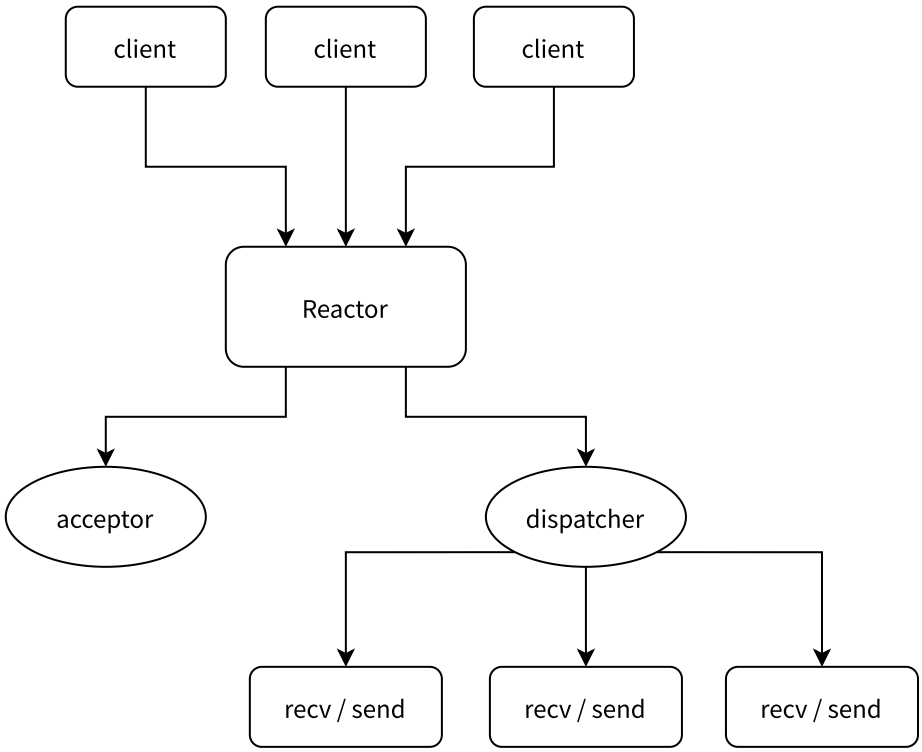
```
. kill.sh
```

功能测试

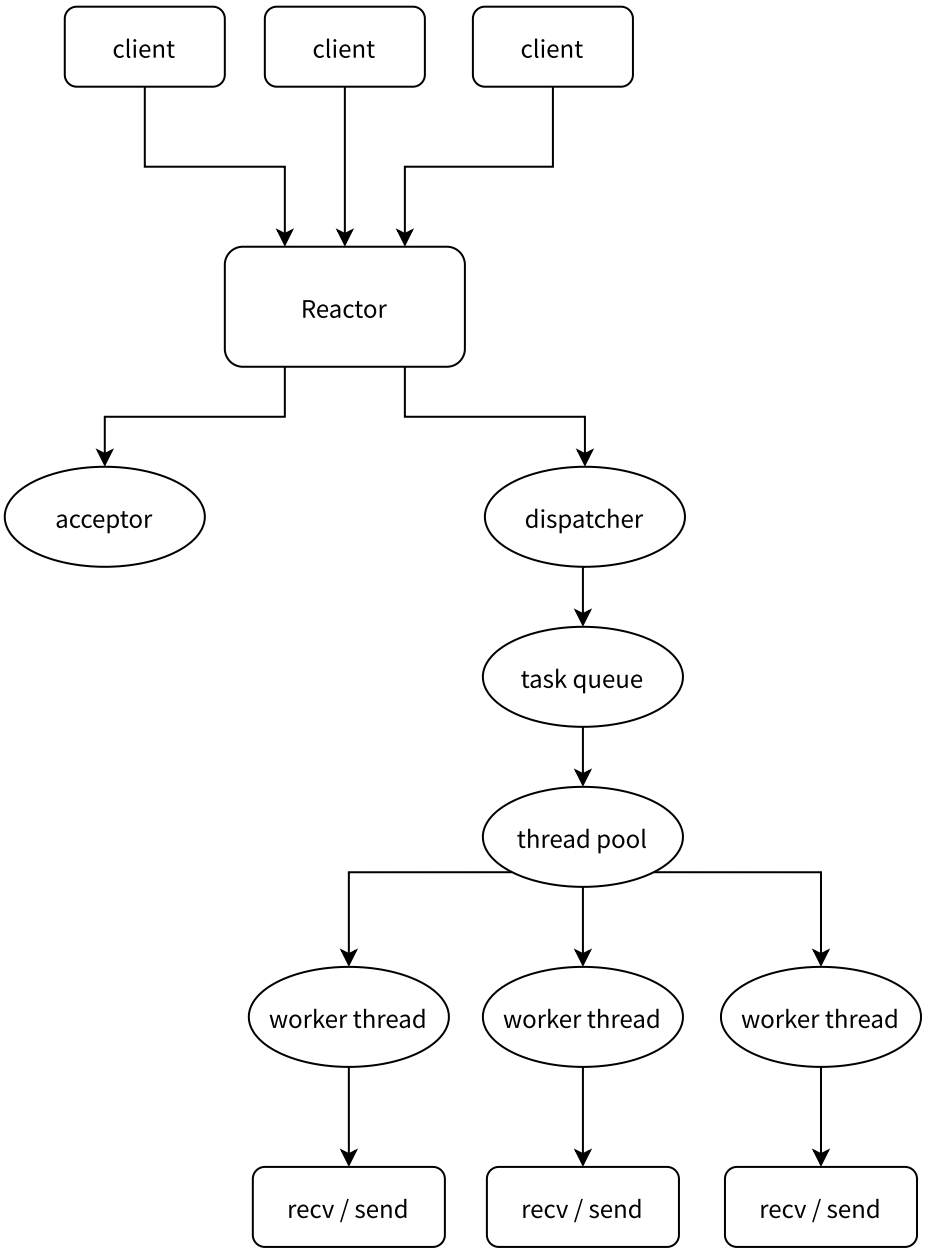
```
python3 client.py
```

框架设计与实现

Reactor + 单线程



Reactor + 多线程



服务配置封装

配置文件

config/server.ini

```
[server]
ip = 127.0.0.1
port = 8080
threads = 64
max_conns = 1024
wait_time = 10
```

```
7 log_level = 0
```

获取当前程序路径

```
1 #include <unistd.h>
2 ssize_t readlink(const char *path, char *buf, size_t len);
```

生成 core 文件

```
1 struct rlimit x;
2 x.rlim_cur = RLIM_INFINITY;
3 x.rlim_max = RLIM_INFINITY;
4 setrlimit(RLIMIT_CORE, &x);
```

动态链接库

加载动态库

```
1 #include <dlfcn.h>
2 void * dlopen(const char * pathname, int mode);
```

参数：

- pathname: 动态库 so 名称
- mode：打开方式

mode 枚举值如下所示：

- RTLD_DEEPBIND：动态库里的函数优先调用本动态库的符号，优先级甚至高于LD_PRELOAD
- RTLD_LAZY：等有需要时才解析出符号，所以如果有未定义的符号,在没调用之前也不执行解析
- RTLD_NOW：在dlopen返回前，解析出所有的未定义符号，如果解析不出来，返回 NULL

- RTLD_GLOBAL：动态库中符号表全局打开，因此符号可被其后打开的其它库重定位
- RTLD_LOCAL：与RTLD_GLOBAL作用相反，动态库中符号表非全局打开，定义的符号不能被其后打开的其它库重定位

返回值：

- 成功：返回库的引用
- 失败：返回 NULL

符号查找

```
1 #include <dlfcn.h>
2 void * dlsym(void* handle, const char* symbol);
```

参数：

- handle：动态库引用
- symbol：符号名称

返回值：

符号函数指针

关闭动态库

```
1 #include <dlfcn.h>
2 int dlclose(void *handle);
```

参数：

- handle：动态库引用

错误描述

```
#include <dlfcn.h>
char * dlerror();
```

返回一个描述最后一次调用dlopen、dlsym 或 dlclose 的错误信息的字符串。

示例：

plugin/foo.h

```
#pragma once

extern "C" int add(int a, int b);
```

plugin/foo.cpp

```
#include <plugin/foo.h>

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

编译插件

```
g++ -shared -fPIC -I. plugin/foo.cpp -o plugin/foo.so
```

调用动态库

```
#include <iostream>

#include <dlfcn.h>

```

```
typedef int (*plugin_func)(int, int);

int main()
{
    void * handle = dlopen("./plugin/foo.so", RTLD_GLOBAL | RTLD_LAZY);
    if (handle == nullptr)
    {
        printf("dlopen error: %s\n", dlerror());
        return -1;
    }

    void * symbol = dlsym(handle, "add");
    if (symbol == nullptr)
    {
        printf("dlsym error: %s\n", dlerror());
        return -1;
    }

    auto func = (plugin_func)symbol;

    int sum = func(1, 2);
    std::cout << sum << std::endl;

    return 0;
}
```

查看动态库符号

```
nm plugin/foo.so
```

业务引擎开发

业务插件配置

```
<?xml version="1.0"?>
<workflow>
    <work id="1" switch="on">
        <plugin name="echo_plugin.so" switch="on" />
        <plugin name="echo_plugin2.so" switch="off" />
```

```
        <plugin name="echo_plugin3.so" switch="on" />
    </work>
    <work id="2" switch="on">
        <plugin name="test_plugin.so" switch="on" />
    </work>
</workflow>
```

Plugin 接口类

```
class Plugin
{
public:
    Plugin() : m_switch(false) {};
    virtual ~Plugin() = default;

    void set_name(const string & name) { m_name = name; }
    const string & get_name() const { return m_name; }

    void set_switch(bool flag) { m_switch = flag; }
    const bool get_switch() { return m_switch; }

    virtual bool run() = 0;

private:
    string m_name;
    bool m_switch;
};

#define DEFINE_PLUGIN(className)                    \
    extern "C" Plugin * create()                    \
    {                                               \
        return new (std::nothrow) className();      \
    }                                               \
    extern "C" void destroy(Plugin * p)             \
    {                                               \
        delete p;                                   \
        p = nullptr;                                \
    }
```

PluginHelper 加载类

```
class PluginHelper
{
public:
    PluginHelper() = default;
    ~PluginHelper() = default;

    bool load(const string & plugin);
    void unload(const string & plugin);
    void * symbol(const string & plugin, const string & symbol);

    void show() const;

private:
    std::map<string, void *> m_plugins;
};
```

Work 类

```
class Work
{
public:
    Work();
    ~Work();

    void set_id(int id) { m_id = id; };
    int get_id() const { return m_id; }

    void set_switch(bool flag) { m_switch = flag; }
    bool get_switch() const { return m_switch; }

    void append(Plugin * plugin);
    bool run();

private:
    int m_id;
    bool m_switch;
    std::vector<Plugin *> m_plugins;
};
```

Workflow 类

```
class Workflow
{
    SINGLETON(Workflow);
public:
    bool load(const string & config);
    bool run(int id, const string & input, string & output);

private:
    bool load_plugin(Work * work, Xml & elem);

private:
    std::map<int, Work *> m_works;
};
```

Context 上下文类

```
class Context
{
public:
    Context() = default;
    ~Context() { clear(); }

    template <typename T>
    void set(const string & key, T value);

    void set(const string & key, const char * value);
    void set(const string & key, const string & value);
    void set(const string & key, Object * value);

    template <typename T>
    T get(const string & key);

    void clear();

private:
    std::map<string, bool> m_bool;
    std::map<string, char> m_char;
    std::map<string, int> m_int;
    std::map<string, double> m_double;
    std::map<string, string> m_str;
```

```
25      std::map<string, Object *> m_obj;
26  };
```

业务插件开发

编写插件

头文件：plugin/TestPlugin.h

```
1  class TestPlugin : public Plugin
2  {
3  public:
4      TestPlugin();
5      virtual ~TestPlugin();
6
7      virtual bool run(Context & ctx);
8
9  };
10 DEFINE_PLUGIN(TestPlugin)
```

源文件：plugin/TestPlugin.cpp

```
1  TestPlugin::TestPlugin() : Plugin() {}
2
3  TestPlugin::~TestPlugin() {}
4
5  bool TestPlugin::run(Context & ctx)
6  {
7      string input = ctx.get<string>("input");
8      ctx.set("output", "test plugin run!");
9      return true;
10 }
```

Context：插件上下文

1. 获取客户端的输入数据
2. 向客户端输出数据

3. 不同插件之间共享数据

编译插件

```
rm -rf plugin/*.so
make plugin
```

配置插件

文件：config/workflow.xml

```
<?xml version="1.0"?>
<workflow>
    <work id="1" switch="on">
        <plugin name="test_plugin.so" switch="on" />
    </work>
</workflow>
```

业务编号：id="1"

插件名称：name="test_plugin.so"

插件开关：switch="on"

加载插件

```
. kill.sh
./server &
```

目前需要重启服务来加载插件，以后可以优化，做成动态加载插件，无需重启服务。

业务引擎整合

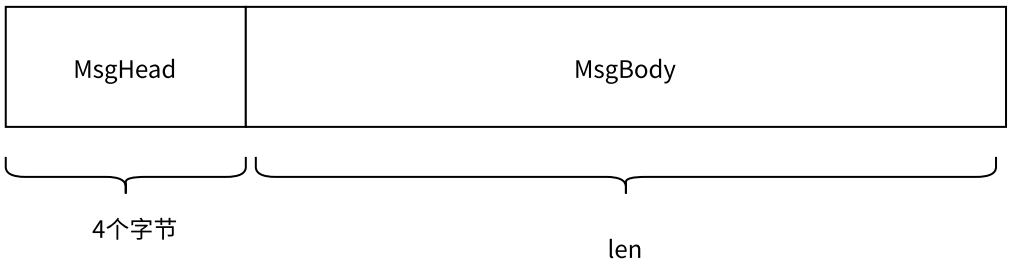
task/work_task.h

```cpp
class WorkTask : public Task
{
public:
    WorkTask() = delete;
    WorkTask(int sockfd);
    ~WorkTask() = default;

    virtual void run();
    virtual void destroy();

private:
    int m_sockfd = 0;
    bool m_closed = false;
};
```

自定义消息格式




```cpp
struct MsgHead {
    uint16_t cmd;
    uint16_t len;
};
```

MsgHead 长度是4个字节。

压力测试

压测工具

```
python3 bench.py 10000
```

场景一

服务器

OS：centos 7.8

CPU：2核（Intel(R) Xeon(R) Gold 6149 CPU @ 3.10GHz）

内存：4G

客户端

和服务端在同一台服务器上运行，客户端用多线程模拟并发

测试结果

客户并发数	连接时间（ms）	请求时间（ms）	一共耗时（ms）
100	0.281	0.299	0.637
1000	0.275	0.331	0.66
1万	0.265	0.344	0.672
10万	0.267	0.352	0.684
50万	11.074	1.879	13.027
100万	59.216	3.434	62.744

场景二

服务器

OS：centos 6.10

CPU：2核（Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz）

内存：8G

客户端

和服务端在同一台服务器上运行，客户端用多线程模拟并发

测试结果

客户并发数	连接时间（ms）	请求时间（ms）	一共耗时（ms）
100	0.401	0.425	0.87
1000	0.385	0.414	0.859
1万	0.411	0.441	0.913
10万	0.441	0.466	0.971
50万	0.408	0.421	0.89
100万	1.36	0.531	1.957

场景三

服务器

OS：centos 6.7

CPU：8核（Intel(R) Xeon(R) CPU E5-2603 v2 @ 1.80GHz）

内存：48G

客户端

和服务端在同一台服务器上运行，客户端用多线程模拟并发

测试结果

客户并发数	连接时间（ms）	请求时间（ms）	一共耗时（ms）
100	0.722	0.77	1.824
1000	0.646	0.68	1.607
1万	0.889	2.121	3.424
10万	1.349	1.488	3.391
50万	0.859	1.033	2.286
100万	1.214	1.488	3.172

完结