

Yazi：一个百万并发的C++服务框架

介绍

框架特性

1. 操作系统：Linux
2. 编程语言：C++
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. 连接池
7. 对象池
8. 线程池
9. 任务队列
10. 业务引擎：插件

适合人群

1. 有一定C++基础的人，想进一步学习系统编程，网络编程，多线程编程
2. 有一定C++项目实践的人，想写出更好的代码，更稳定的服务

C++ 从入门到放弃

1. 世界上最复杂的语言，没有之一
2. C++强大的令人感到不可思、然而也复杂的令人发指
3. C++让你花费大量时间在学习这门语言的语法上，而不是解决问题本身上
4. 造轮子是C++的命，也是C++的病
5. 内存管理：永远怀着一颗不安的心

安装框架

编译插件

```
1 make plugin
```

编译框架

```
1 make
```

启动服务

```
1 ./main &
```

压力测试

压测工具

```
1 python3 client/python/bench.py
```

配置文件

文件：config/main.ini

```
1 [server]
2 ip = 127.0.0.1
3 port = 8080
4 threads = 64
5 max_conn = 10000
6 wait_time = 10
7
8 [client]
9 threads = 100
```

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

	A	B	C	D
1	客户并发数	连接时间（ms）	请求时间（ms）	一共耗时（ms）
2	100	0.281	0.299	0.637
3	1000	0.275	0.331	0.66
4	1万	0.265	0.344	0.672
5	10万	0.267	0.352	0.684
6	50万	11.074	1.879	13.027
7	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

	A	B	C	D
1	客户并发数	连接时间（ms）	请求时间（ms）	一共耗时（ms）
2	100	0.401	0.425	0.87
3	1000	0.385	0.414	0.859
4	1万	0.411	0.441	0.913
5	10万	0.441	0.466	0.971
6	50万	0.408	0.421	0.89
7	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

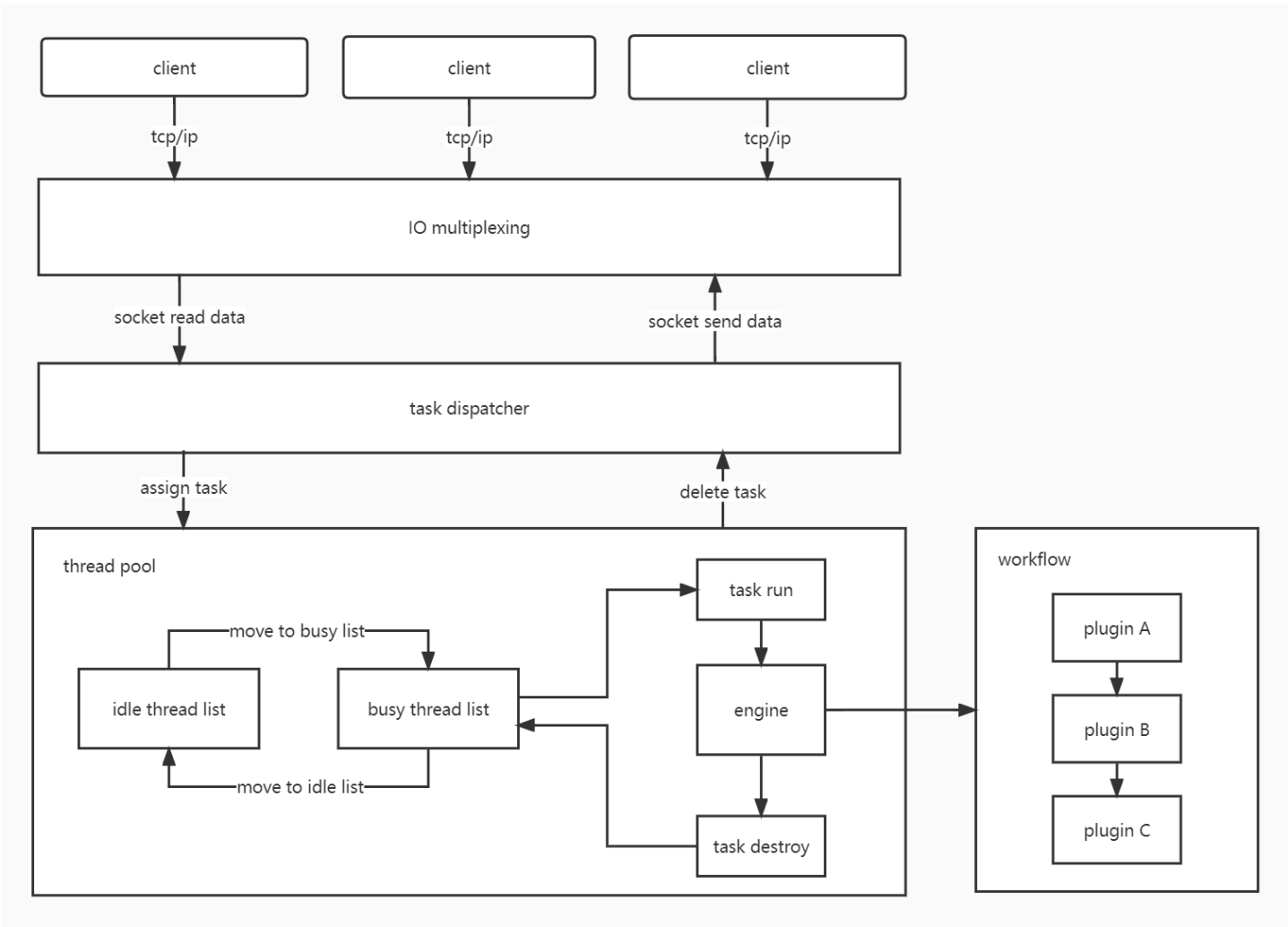
和服务端在同一台服务器上运行，客户端用多线程模拟并发

测试结果

	A	B	C	D
1	客户并发数	连接时间（ms）	请求时间（ms）	一共耗时（ms）
2	100	0.722	0.77	1.824
3	1000	0.646	0.68	1.607
4	1万	0.889	2.121	3.424
5	10万	1.349	1.488	3.391
6	50万	0.859	1.033	2.286
7	100万	1.214	1.488	3.172

架构设计

Reactor + 多线程模型



主线程

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

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：

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

开发工具：vscode

工具下载：<https://code.visualstudio.com/>

测试环境：docker

工具下载：<https://www.docker.com/>

准备材料：

- 1、Dockerfile：构建 centos 系统，设置 root 账号的密码为：password

```
1 FROM centos:7.8.2003
2
3 MAINTAINER oldjun <oldjun@sina.com>
4
5 RUN yum install -y initscripts && \
6     yum install -y gcc && \
7     yum install -y gcc-c++ && \
8     yum install -y kernel-devel && \
9     yum install -y make && \
10    yum install -y wget && \
11    yum install -y vim-enhanced \
12    yum install -y net-tools && \
13    yum install -y openssh && \
14    yum install -y openssh-server && \
15    yum install -y openssl-devel && \
```

```
16      yum install -y ncurses-devel && \  
17      yum install -y sqlite-devel && \  
18      yum install -y readline-devel && \  
19      yum install -y libffi-devel && \  
20      yum install -y git && \  
21      echo "root:password"|chpasswd && \  
22      ssh-keygen -A  
23  
24  ADD Python-3.9.6.tar.xz /root/  
25  
26  RUN cd /root/Python-3.9.6 && \  
27      ./configure prefix=/usr/local/python3 && \  
28      make && \  
29      make install && \  
30      ln -s /usr/local/python3/bin/python3.9 /usr/bin/python3 && \  
31      ln -s /usr/local/python3/bin/pip3 /usr/bin/pip3  
32  
33  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：

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

启动容器

1、在windows（管理员）终端，运行命令：

```
1 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

2、docker容器里，启动SSH服务

```
1 /usr/sbin/sshd
```

连接容器

vscode通过SSH连接docker容器

```
1 ssh root@127.0.0.1 -A
```

注：root密码：password

编译框架

```
1 make plugin
2 make
```

启动服务

```
1 ./main &
```

关闭服务

```
1 . kill.sh
```

测试工具

```
1 python3 client/python/client.py
```

注意：重启docker容器，需要删除文件：C:\Users\HUAWEI\.ssh\known_hosts

代码结构

目录结构



不依赖第三方库

全部模块和工具类都自己实现

模块独立

模块之间相互独立

各个模块可以单独出来，最好是在别的项目里可以复用

Linux 跨平台

代码尽量做到平台无关

目前：在Centos上编译运行没问题

未来：需要在其他Linux平台上正常编译运行

代码维护性

代码尽量做到简洁，易读

网络编程理论

传输层协议

1. TCP：一种面向连接的、可靠的、基于字节流的传输层通信协议（web HTTP 1.1/2.0）
2. UDP：一种无需建立连接就可以发送封装的 IP 数据包（IP电话、语音、视频聊天）
3. SCTP：一种在网络连接两端之间同时传输多个数据流的协议（SIP，电信业务，Centrex）

最简单的服务端

伪代码：

```
1 int main() {
2     // 第一步：创建 tcp socket
3     int sockfd = socket(TCP);
4
5     // 第二步：绑定 ip 和 port
6     bind(sockfd, ip, port);
7
8     // 第三步：监听端口
9     listen(sockfd);
10
11    while (true) {
12        // 第四步：接收客户端连接
13        int connfd = accept(sockfd);
14
15        // 第五步：读取客户端的数据
16        data = recv(connfd);
17
18        // 第六步：处理客户端的数据
19        ...
20
21        // 第七步：向客户端发送数据
22        send(connfd, data);
23
24        // 最后一步：关闭连接
25        close(connfd);
26    }
27    return 0;
28 }
```

1. 创建套接字，指定通信的协议（socket）；
2. 给socket绑定ip地址和端口号（bind）；
3. 开始监听这个socket（listen）；
4. 如果有socket连接到到来，就用accept函数获取（accept）；
5. 获取到连接之后，在一个while循环里读socket客户端发来的数据（recv）；
6. 给客户端发送数据（send）；
7. 关闭连接（close）。

最简单的客户端

伪代码：

```
1  int main() {  
2  
3      // 第一步：创建 tcp socket  
4      int sockfd = socket(TCP);  
5  
6      // 第二步：连接服务端  
7      connect(sockfd, ip, port);  
8  
9      // 第三步：向服务端发送数据  
10     send(sockfd, data);  
11  
12     // 第四步：接收服务端的数据  
13     data = recv(sockfd);  
14  
15     // 最后一步：关闭连接  
16     close(sockfd);  
17  
18     return 0;  
19 }
```

1. 创建套接字，指定通信的协议号（socket）；
2. 根据ip地址和端口号连接到服务端（connect）；
3. 发送消息（send）；
4. 接收消息（recv）；
5. 关闭连接（close）。

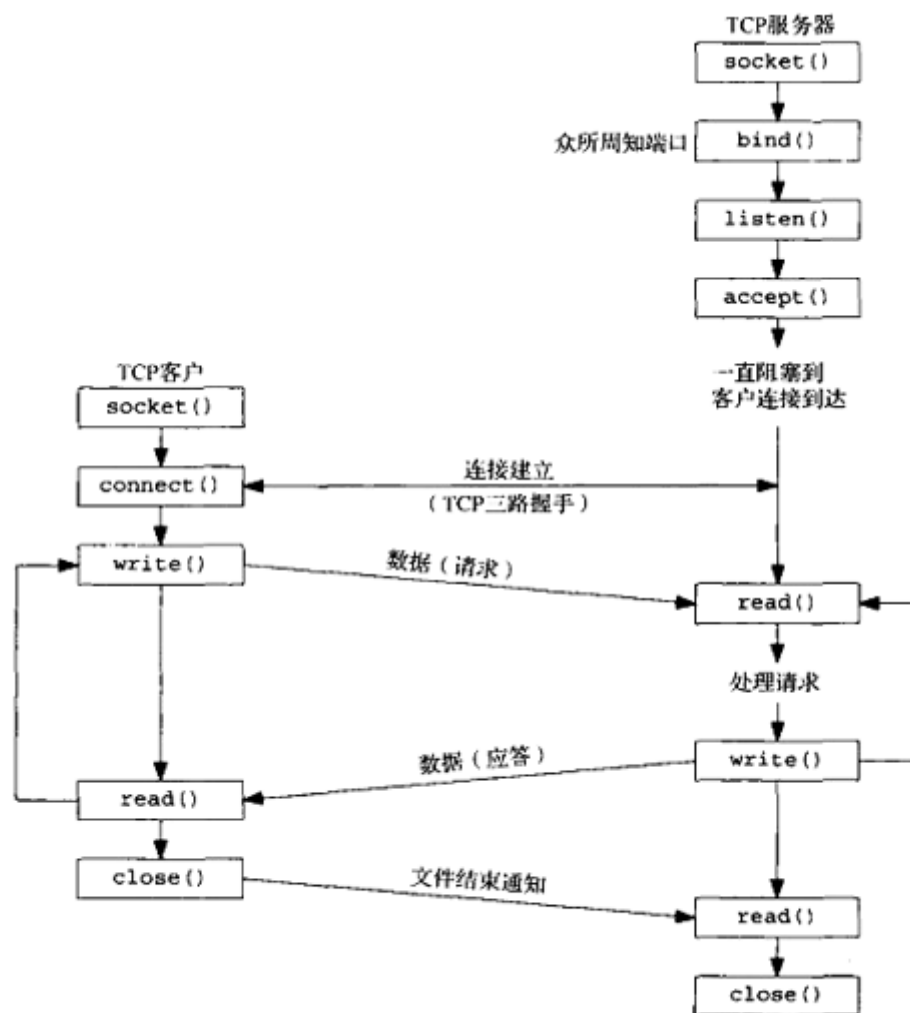


图4-1 基本TCP客户/服务器程序的套接字函数

阻塞 IO

当用户线程发出IO请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除阻塞状态。

非阻塞 IO

当用户线程发起read操作后，并不需要等待，而是马上得到结果。如果结果是一个error时，它就知道数据还没有准备好，于是它可以再次发起read操作。如果内核中的数据准备好了，它就将数据拷贝到用户线程。

在非阻塞IO模型中，用户线程需要不断地轮询内核数据是否就绪，也就是说非阻塞IO不会交出CPU，而会一直占用CPU。

```

1 while (true)
2 {
3     // 第五步：读取客户端的数据
4     data = recv(connfd);
5     if (data != error)
6     {
7         // 第六步：处理客户端的数据
8         ...
9         break;
10    }
11 }
  
```

IO 多路复用

在多路复用IO模型中，会有一个专门的线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。IO多路复用的优势在于，可以处理大量并发的IO，而不用消耗太多CPU/内存。

三种常用的轮询方法：select、poll、epoll

select

优点：

1. select 可移植性更好，在某些Unix系统上不支持 poll;
2. select 对于超时提供了更好的精度：微秒，而 poll 和 epoll 是毫秒。

缺点：

1. 单个进程可监听的fd数量有限制（最多允许1024个连接）；
2. 需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时，复制开销大；
3. 对fd进行扫描时是线性扫描。fd剧增后，IO效率较低，因为每次调用都对fd进行线性扫描遍历，所以随着fd的增加会造成遍历速度慢的性能问题。

poll

优点：

1. 没有描述符个数的限制；
2. poll 在应付大数目的文件描述符的时候相比于select速度更快；

缺点：

1. 大量的 fd 在用户空间和内核空间之间复制，复制开销大；
2. 与select一样，poll返回后，需要轮询 fd 来获取就绪的描述符；

epoll

优点：

1. 没有描述符个数的限制；
2. IO效率不随 fd 数目增加而线性下降；
3. 使用mmap内存映射加速内核与用户空间的数据传递，不存在复制开销；
4. 无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件唤醒而加入就绪队列的描述符集合就行了。
5. epoll除了提供select/poll 那种IO事件的电平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll_wait的调用，提高应用程序效率。

缺点：

1. epoll 跨平台性不够、只能工作在 linux 下；

epoll 只有三个函数：epoll_create、epoll_ctl、epoll_wait，使用非常简单：

```

1  int main() {
2
3      // 第一步：创建epoll描述符
4      int epfd = epoll_create(max_conn_size + 1);
5
6      // 第二步：注册一个 fd 到 epoll
7      epoll_ctl(epfd, op, fd, event);
8
9      while (true) {
10         // 第三步：监听全部 fd
11         int num = epoll_wait(epfd, timeout);
12
13         for (int i = 0; i < num; i++) {
14             // 第四步：读取客户端的数据
15             data = recv(fd);
16
17             // 第五步：处理客户端的数据
18             ...
19
20             // 第六步：向客户端发送数据
21             send(fd, data);
22         }
23     }
24 }

```

好处：

IO多路复用的优势在于，可以处理大量并发的IO，而不用消耗太多CPU/内存。

问题：

多路复用IO模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用IO模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

解决方案：

多线程可以解决事件响应体很大时，后续的事件迟迟得不到处理的问题。

主线程

系统初始化

```

1  int main()
2  {
3      System * sys = Singleton<System>::instance();
4      sys->init();
5  }

```

1. 日志初始化

2. 配置初始化

服务启动

```
1 int main() {
2     Server * server = Singleton<Server>::instance();
3     server->listen(ip, port);
4     server->start();
5 }
```

1. 创建分发线程

2. 启动网络服务

socket封装类

```
1 class Socket
2 {
3 public:
4     bool bind(const string &ip, int port);
5     bool listen(int backlog);
6     bool connect(const string &ip, int port);
7     bool close();
8
9     int accept();
10    int recv(char * buf, int len);
11    int send(const char * buf, int len);
12
13    bool set_non_blocking();
14    bool set_send_buffer(int size);
15    bool set_recv_buffer(int size);
16    bool set_linger(bool active, int seconds);
17    bool set_keep_alive();
18    bool set_reuse_addr();
19    bool set_reuse_port();
20
21 protected:
22     string m_ip;
23     int m_port;
24     int m_sockfd;
25 };
```

服务端socket

```
1 class ServerSocket : public Socket
2 {
3 public:
4     ServerSocket();
5     ServerSocket(const string &ip, int port);
```

```
6     virtual ~ServerSocket();
7 };
```

客户端socket

```
1 class ClientSocket : public Socket
2 {
3 public:
4     ClientSocket();
5     ClientSocket(const string &ip, int port);
6     virtual ~ClientSocket();
7 };
```

epoll封装类

```
1 class EventPoller
2 {
3 public:
4     void create(int max_connections);
5     void add(int fd, void * ptr, __uint32_t events);
6     void mod(int fd, void * ptr, __uint32_t events);
7     void del(int fd, void * ptr, __uint32_t events);
8     int wait(int millsecond);
9
10 protected:
11     void ctrl(int fd, void * ptr, __uint32_t events, int op);
12 };
```

Socket Handler类

```
1 class SocketHandler
2 {
3 public:
4     SocketHandler();
5     ~SocketHandler();
6
7     void listen(const string & ip, int port);
8     void attach(Socket * socket);
9     void detach(Socket * socket);
10    void remove(Socket * socket);
11    void handle(int max_connections, int wait_time);
12
13 private:
14     EventPoller * m_epoll;
15     Socket * m_server;
16     ObjectPool<Socket> m_sockpool;
```

```
17     Mutex m_mutex;  
18 };
```

1. handle: 实现IO多路复用功能
2. attach: 将客户端的socket加入epoll监听池, 进行监听
3. detach: 将客户端的socket移出epoll监听池, 不再监听

分发线程

Task 类

```
1 class Task  
2 {  
3 public:  
4     Task();  
5     Task(void* data);  
6     virtual ~Task();  
7  
8     void* get_data();  
9     void set_data(void* data);  
10  
11     virtual void run() = 0;  
12     virtual void destroy() = 0;  
13  
14 protected:  
15     void*      m_data;  
16     Mutex      m_mutex;  
17 };
```

Task类定义了两个方法:

1. run: 执行任务
2. destroy: 销毁任务

Task Dispatcher 类

```
1 class TaskDispatcher : public Thread  
2 {  
3 public:  
4     TaskDispatcher();  
5     ~TaskDispatcher();  
6  
7     void init(int threads);  
8     void assign(Task* task);  
9     void handle(Task* task);  
10    virtual void run();  
11  
12 protected:  
13    std::list<Task*> m_tasks;  
14 };
```


任务分发器：

1. assign：将请求任务加入任务队列
2. handle：将任务分发给线程池处理

线程池

Thread线程类

```
1 class Thread
2 {
3 public:
4     Thread();
5     virtual ~Thread();
6
7     virtual void run() = 0;
8
9     void start();
10    void stop();
11
12    void set_task(Task* task);
13    Task* get_task();
14
15 protected:
16     static void* thread_func(void* ptr);
17
18 protected:
19     pthread_t      m_tid;
20     Task*          m_task;
21     Mutex          m_mutex;
22     Condition      m_cond;
23 };
```

1. start：启动线程
2. stop：线程退出
3. run：线程执行过程

WorkThread工作线程

```
1 class WorkerThread : public Thread
2 {
3 public:
4     WorkerThread();
5     virtual ~WorkerThread();
6
7     virtual void run();
8
9     static void cleanup(void* ptr);
10 };
```

1. run：执行任务、销毁任务、从忙碌队列挪到空闲队列

ThreadPool线程池

```
1 class ThreadPool
2 {
3 public:
4     ThreadPool();
5     ~ThreadPool();
6
7     void create(int threads);
8
9     Thread* get_idle_thread();
10
11     void move_to_idle_list(Thread* thread);
12     void move_to_busy_list(Thread* thread);
13
14     int get_idle_thread_numbers();
15     int get_busy_thread_numbers();
16
17     void assign(Task* task);
18
19 private:
20     int m_threads;
21
22     std::set<Thread*> m_list_idle;
23     std::set<Thread*> m_list_busy;
24
25     Mutex m_mutex_idle;
26     Mutex m_mutex_busy;
27
28     Condition m_cond_idle;
29     Condition m_cond_busy;
30 };
```

业务插件

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.ref<string>("input");
8     ctx.ref<string>("output") = input + " 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

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

配置插件

文件：config/workflow.xml

```
1 <?xml version="1.0"?>
2 <workflow>
3     <work name="1" switch="on">
4         <plugin name="testplugin.so" switch="on" />
5     </work>
6 </workflow>
7
```

业务编号：name="1"

插件名称：name="testplugin.so"

插件开关：switch="on"

加载插件

```
1 . kill.sh  
2 ./main &
```

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

完结

