**WebServer**

**Channel类**

功能 存储一个文件描述符的相关信息:关心的事件(event),发生的事件(revent),各种事件的回调函数

数据成员 loop\_, event\_, revent\_, readCallback\_, writeCallback\_, errorCallback\_, close Callback\_, index\_, eventHandling\_

**技术点** 1)handleEvent()成员函数根据发生的事件调用相应的回调函数

2) Channel的更新和移除都发生在所属的Loop线程

**Poller类**

功能 监听文件描述符，返回活跃的Channnel; Channel的更新和移除

数据成员 loop\_, pollfds\_, channels\_

**技术点** 1)用map将文件描述符与对应的Channel进行关联；将所有发生事件的文件描述符对应的Channel返回

**TimeQueue类**

功能 利用timerfd文件描述符管理定时器，并将自己注册到EventLoop中

数据成员 loop\_, timerfd\_, timerfdChannel\_, timers\_, activeTimers\_, cancelTimers, callingExpiredTimers\_

**技术点** 1)利用timerfd来管理定时器

2)用timer的序列号来移除timer;利用地址来移除定时器的潜在错误是：同一地址可能存在不同时间分配的timer。

**TimeStamp类**

功能 存储自1970年1月1日0时以来的微秒数

数据成员 microSecondsSinceEpoch\_

**技术点** 调用gettimeofday()获得时间

**EventLoop类**

功能 1)负责监听用户关心的文件描述符事件 2)可以定时调用某个函数 3)可以执行其他线程分配的任务

数据成员 threadId\_, looping\_, quit\_, pollReturnTime\_, callingPendingFunctors\_, poller\_, timeQueue\_, wakeupFd\_, wakeupChannel\_, activeChannels\_, mutex\_, pendingFunctors\_

**技术点** 1)利用eventfd文件描述符来唤醒poll(),以便执行其他线程分配的任务

2)任何线程都可以停止某个EventLoop的loop()

3)利用互斥锁来同步多个线程给同一线程分配任务。

4)利用线程变量t\_loopInThisThread来限制 one thread, one loop。在EventLoop的构造函数中检查t\_loopInThisThread

5)doPendingFunctors()利用vector的swap，使得其他线程可以继续向队列中添加任务，这样不会阻塞其他线程

**Timer类**

功能 存储一个定时器的相关信息

数据成员 cb\_, expiration\_, interval\_, repeat\_, sequence\_

**技术点** 原子性的操作定时器的序列号

**mutex互斥量**

技术点 1)多线程中才使用同步机制

2)MutexLock构造函数负责init互斥量，析构函数负责destory互斥量；

MutexLockGuard()的构造函数负责lock互斥量，析构函数负责unlock互斥量

3)注意MutexLockGuard保存的是Mutex对象的引用

4)互斥量不可拷贝

**Thread类**

功能 存储线程相关信息，创建线程

数据成员 bool **started\_**,bool **join**\_,string **name\_**,pthread **pthreadId\_**,shard\_ptr<pid\_t> **pTid\_**, ThreadFunc **func\_**;

**技术点** 1)首先，将线程id返回给父线程，其次将子线程的线程名都存储在线程变量中；最后执行子线程函数

2)传递给pthread\_create()的线程函数是全局函数startThread。即Thread创建的线程的main函数是startThread

3)当线程函数运行结束时，线程名设置为“finished”,并且把ThreadData释放掉

4)利用一个ThreadNameInitializer类对象

1)设置主线程的线程名为“main“

2)设置atfork函数:在fork之后，设置线程名为“main“，并存储线程id到线程变量中

5)将创建子线程的情况，可以通过Thread类对象来查看

**ThreadPool类**

功能 创建指定数目的线程，创建的子线程们执行任务队列中的任务

数据成员 bool **running\_**,

int **numThreads\_**,

boost::ptr\_vector<Thread> **threads\_,**

stirng **name\_**,

MutexLock **mutex\_**,

Condition **cond\_**,

queue<Task> **tasks\_**

**技术点** 1)线程池中的线程的线程函数为ThreadPool:: runInThread ()，即线程池中每个线程执行ThreadPool对象的一个成员函数

2)线程池中的线程从任务队列中拿任务并执行，使用互斥量和条件变量进行多线程同步操作任务队列

3)当调用线程池对象的stop时，最终所有子线程结束运行。此时，队伍队列中可能仍有任务没有执行

4)当ThreadPool对象结束生命周期时，析构函数将running\_设为false,并等待所有子线程的结束，释放了子线程占用的资源

5) ThreadPool对象通过run(Task task)向任务队列添加任务

**EventLoopThread类**

功能 创建的线程有一个EventLoop的栈对象，并执行EventLoop::loop()

数据成员 EventLoop\* **loop\_**;

bool **exiting**\_;

Thread **thread\_**;

MutexLock **mutex\_**;

Condition **cond\_**;

**技术点** 1)子线程的EventLoop对象地址，存储在EventLoopThread对象中

2)子线程执行的是EventLoop对象的成员函数ThreadFun()

3)子线程和父线程通过互斥量和条件量来同步对loop\_的访问

4)当EventLoop对象析构时，调用loop\_->quit()停止子进程的EventLoop(),并等待子线程终止

**EventLoopThreadPool类**

功能 创建线程池，每个线程都是一个EventLoopThread线程

数据成员 EventLoop\* **baseLoop\_**;

bool **started\_**;

int **numThreads**\_;

int **next\_**;

boost::ptr\_vector<EventLoopThread> **threads\_**;

std::vector<EventLoop\*> **loops\_**;

**技术点** 1)利用boost::ptr\_vector管理EventLoopThread, 使得EventLoopThreadPool对象析构时，EventLoopThread的析构函数自动调用，即自动终止线程池中的所有线程

2)Round Robin的方式使用线程池中的线程

3) EventLoopThreadPool使用baseLoop\_。当线程池中没有线程时，返回此baseLoop\_

Acceptor类

功能 负责监听套接字

数据成员 EventLoop\* loop\_;

Socket acceptSocket\_;

Channel acceptChannel\_;

NewConnectionCallback newConnectionCallback\_;

bool listenning\_;

技术点 1)acceptor的handleRead()在accept一个套接字之后，调用newConnectionCallback函数

**TcpConnection类**

功能 负责一个TCP连接

数据成员 EventLoop\* **loop\_**;

std::string **name\_**;

StateE **state\_**;

boost::scoped\_ptr<Socket> **socket\_**;

boost::scoped\_ptr<Channel> **channel\_**;

InetAddress **localAddr\_**;

InetAddress **peerAddr\_**;

ConnectionCallback **connectionCallback\_**;

MessageCallback **messageCallback\_**;

WriteCompleteCallback **writeCompleteCallback\_**;

CloseCallback **closeCallback\_**;

Buffer **inputBuff**er\_;

Buffer **outputBuffer\_**;

**技术点** 1)利用scoped\_ptr管理socket\_和channel\_

2)channel\_的readCallback,writeCallback,closeCallback,errorCallback回调函数是TcpConnection对象的成员函数handleRead(),handleWrite(),handleClose(),handleError()

3)connectEstablished ()和connectDestroyed()都调用了connectionCallback回调函数

4)提供了send(message)作为发送数据的接口

5)提供了shutdown()成员接口函数，用来关闭套接字的写端

6)connectEstablished()将channel\_添加到loop\_的poller\_中，监听套接字

7)connectDestroyed()将channel\_从loop\_的poller\_中移除，不再监听套接字

**TcpServer类**

功能 创建一个多线程的TCP服务器，监听套接字独属一个线程。每个TCP连接是线程池中的某个线程负责

数据成员 EventLoop\* loop\_; // the acceptor loop

const std::string name\_;

boost::scoped\_ptr<Acceptor> acceptor\_; // avoid revealing Acceptor

boost::scoped\_ptr<EventLoopThreadPool> threadPool\_;

ConnectionCallback connectionCallback\_;

MessageCallback messageCallback\_;

WriteCompleteCallback writeCompleteCallback\_;

bool started\_;

int nextConnId\_; // always in loop thread

ConnectionMap connections\_;

技术点 1)一个套接字只属于一个线程；一个线程负责多个套接字

2)使用scoped\_ptr管理EventLoopThreadPool，使得TcpServer析构时，所有的线程停止loop,正常退出

3)acceptor\_的newConnectionCallback是TcpServer的成员函数newConnection

使用方法 1)调用setConnectionCallback(), setMessageCallback(), setWriteCompleteCallback()函数设置回调函数

2)调用start(numThread)：创建并开启线程池，acceptor\_的监听套接字注册到TcpServer所属的EventLoop中。

3) TcpServer所属的EventLoop调用loop()函数

**日志系统**

**LogFile类**

功能 将日志信息写入文件，并负责按大大小和周期滚动日志文件

数据成员 baseName\_, rollSize\_, flushInterval\_, count\_, mutex\_, startOfPeriod\_, lastRoll\_, lastFlush\_, file\_

设计思想 1)内部类File使用文件流将c字符串写入文件中，并设置文件流的buffer大小为64k。

2)将日志消息利用File类写入文件中后，按大小和时间滚动文件。

1)超过指定大小，滚动文件，往新文件中写入日志

2) 每三秒刷新文件缓冲区，写入磁盘；每60\*60\*24s换一个日志文件写日志，每天的日志文件也不一样

3)File类负责调用fwrite\_unlock()向文件写数据，LogFile在File类写完数据后，判断是否滚动日志

4)日志文件的文件名的设计：baseName.time.pid.log

5)LogFile用scoped\_ptr管理File,使得滚动文件时，scoped\_ptr.reset(自动调用上一个日志文件的析构函数，导致文件刷新缓冲区，确保上一文件写入完整

**技术点** 1)利用有缓存文件流写数据到文件中，减少了write()的系统调用次数

2)设置每3秒刷新日志文件，防止大量日志未来得及写入磁盘中

3)进程id采用缓存的形式，只调用系统调用一次

**FixBuffer模板类**

功能 将c字符串写入buffer中

数据成员 void (\*cookie\_)();

char data\_[SIZE];//buffer

char\* cur\_;//下一个写入位置

**技术点** 在cc文件中实例化4000和4000\*1000大小的FixBuffer

**LogStream类**

功能 将各种基础类型转化成c字符串后存入FixBuffer的buffer中

数据成员 FixBuffer<kSmallBufferSize> buffer\_

**技术点** 1)重载operator<<操作符，使得各种类型的数据以流的方式存到buffer中

2)使用Matthew Wilson的数值转字符串方法[“9876543210123456789”]

3)将地址转成16进制的字符串

4)利用snprintf将double类型转成c字符串

5)提供Fmt类，将数值以自定义的printf格式转化成字符串

**Logger类**

功能 构建日志消息格式，并将日志写入指定的地方

数据成员 Impl impl;

技术点 1)利用宏定义生成Logger类临时对象，并提供LogStream对象进行日志消息的写入

2)在构造函数中，构造日志消息的前部分格式；在析构函数中，完成日志格式的后半部分，并调用全局函数将impl.buffer中的日志消息写入指定地方

3)利用Impl内部类进行日志消息的构造和输出全过程。impl的数据成员：Timestamp time\_,LogStream stream\_,int line\_,fullname\_,basename\_.LogLevel level\_

4)利用编译器变量\_\_FILE\_\_,\_\_LINE\_\_,\_\_func\_\_得到日志消息产生的文件名，行数，函数名

5)将LogFile::append()函数设置为g\_outputFunc就可以实现将日志消息写入文件中

6)日志消息的时间采用了秒缓存，微妙更新的方法

每个线程保存上一条日志消息的秒数；如果此时产生的日志消息的秒数和上一条日志消息的一致，则秒不用进行转化。每次只更新微秒的字符串

7)线程的ID采用了缓存，只有第一次获得线程ID时，才通过系统调用得到线程ID

8)利用全局函数来进行日志消息的写入和刷新

总结：

1)每个类负责一定的功能，不同的类在已有类功能的基础上，进行功能的扩展

2)FixBuffer负责将数据写入缓存，LogStream负责将数据格式话化；将FixBuffer作为LogStream的数据成员，即可实现将数据格式化后存入缓存中，

3)Logger类负责构造日志消息；将LogStream作为logger的数据成员,即可实现将日志消息存入缓存中

4)File类负责将数据写入文件中，LogFile负责文件的滚动；将File类作为LogFile的数据成员，即可实现将数据写入文件，并按时间和大小进行文件的滚动

5)将LogFile对象的Logfile.append()作为Logger对象的outputFunc，即可实现将日志消息存入日志文件中。即日志系统构建完成