第六章 muduo网络库简介

1. 为什么需要网络库

网络库的价值在于能方便地处理并发连接。

1. muduo是基于Reactor模式的网络库，其核心是事件循环Eventloop，用于响应计时器和IO事件。

Reactor模式：

1. 前向声明(forward declaration)

前向声明是与超前引用有关的，所谓超前引用就是A类中使用了B类，而在B类中使用了A类，而在C++中一个类在使用前必须被定义，而在编译时编译器要么先编译A类，要么先编译B类，那么先编译的那个类中就无法识别后编译的类型，编译器可能会报未定义错误，这就是超前引用。

解决的办法就是在先编译的类之前声明另外的类为一个类型，但是并没有定义(这就是一个不完全类型)。这样的方法有一个限制，**完全类型只能用于定义指向该类型的指针及引用，或者用于声明(而不是定义)使用该类型作为形参类型或返回类型的函数。示例代码：**

class A;

class B

{

public:

A\* m\_a; //（不能A m\_a）

}

#include "A.h"

class B

{

public:

A\* m\_a; //(或者A m\_a)

}

类的前向声明只适用于指针和引用的定义，如果是普通类类型就得使用include了。

前向声明好处是: 不必要的#include，会增加编译时间。混乱随意的#include可能导致循环#include，可能出现编译错误。

1. TCP网络编程最本质的是处理三个半事件：
2. 连接的建立。包括客户端的connect和服务端的accept；
3. 连接的断开，包括主动断开（close、shutdown）和被动断开（read(2)返回0）；
4. 消息到达、文件描述符可读，这是最为重要的事件，对它的处理决定了网络编程的风格（阻塞/非阻塞，如何处理分包，应用层的缓存如何设计）；
5. 消息发送完毕，算半个。
6. muduo库解决了网络编程中遇到的如下问题：

* 如果要主动关闭链接，如何保证对方收到自己的全部数据？如果应用层有缓冲，那么如何保证先发送完缓冲区中的数据然后再断开连接？

回答：(p191)由于TCP是一个全双工协议，同一个文件描述符既可以读也可以写，muduo使用系统调用shutdown()来关闭连接，shutdown()会主动关闭本地“写“方向的连接，保留了”读”方向，这称为TCP half-close。如果直接close，那么该连接既不能读也不能写了。所以主动关闭使用shutdown()，如果还有未接收的对方数据的话，还会读取，知道read()返回0，关闭连接。

至于如果缓存中还有数据未发送完，muduo也不会立刻关闭连接，而是等待数据发送完毕再shutdown，从而对方避免漏收数据。在TcpConnection对象析构的时候，该连接持有的Scoket对象的析构函数会close(sockfd\_)，Socket对象是一个RAII handler。

综上，muduo使用半关闭(shutdown)的原因是为了收发数据的完整性

* 如果主动发起连接，但是对方主动拒绝，如何定期重试？
* 在非阻塞网络编程中，为什么要使用应用层发送缓冲区？

回答：如果等待OS缓冲区可用会阻塞当前线程，因为不知道对方什么时候收到收到并读取数据。

* 如果应用程序要发送40kB数据，但是操作系统的TCP发送缓冲区只有25kB剩余空间，那么剩余的15kB的数据怎么办？

回答：承上回答，因此网络库应该把这15kB数据缓存起来，这样放到这个TCP链接的应用发送缓冲区中，

* 在非阻塞网络编程中，为什么要使用应用层的接受缓冲区？

回答：如果数据没有一次性收全，已经收到的数据会累积在应用层的接收缓冲区中，以等待后续数据到达，程序不会阻塞，这样即便服务器一个一个字节地发送数据，代码还是能正常工作。

* 如果一次读到的数据不够一个完整的数据包，那么这些已经读到的数据是不是应该先暂存在某个地方，等剩余的数据收到之后再一并处理？
* 在非阻塞网络编程中，如何设计并使用缓冲区？在多开辟空间可一次多读数据从而减少系统调用和少占用内存中如何取舍？

回答：muduo使用readv(2)并结合栈上空间解决了第二个问题。

(下面的内容在7.4.3，p208)具体的做法是，在栈上准备一个65536字节的extrabuf，然后利用readv()来读取数据，iovec有两块，第一块指向muduo Buffer中的writable字节，另一块指向栈上的extrabuf。如果读入的数据不多，那么全部读到Buffer中去；如果长度超多Buffer的writable字节数，就会读到栈上的extrabuf里，然后程序再把extrabuf里的数据append()到Buffer中，代码如下：

ssize\_t Buffer::readFd(int fd, int\* savedErrno)

{

// saved an ioctl()/FIONREAD call to tell how much to read

char extrabuf[65536];

struct iovec vec[2];

const size\_t writable = writableBytes();

vec[0].iov\_base = begin()+writerIndex\_;

vec[0].iov\_len = writable;

vec[1].iov\_base = extrabuf;

vec[1].iov\_len = sizeof extrabuf;

// when there is enough space in this buffer, don't read into extrabuf.

// when extrabuf is used, we read 128k-1 bytes at most.

const int iovcnt = (writable < sizeof extrabuf) ? 2 : 1;

const ssize\_t n = sockets::readv(fd, vec, iovcnt);

if (n < 0)

{

\*savedErrno = errno;

}

else if (implicit\_cast<size\_t>(n) <= writable)

{

writerIndex\_ += n;

}

else

{

writerIndex\_ = buffer\_.size();

append(extrabuf, n - writable); //append()负责将扩展buffer\_大小，并复制数据

}

return n;

}

* 如果使用发送方缓冲区，万一接收方处理缓慢，数据会不会一支堆积在发送方，造成内存暴涨？如何做应用层的流量控制？

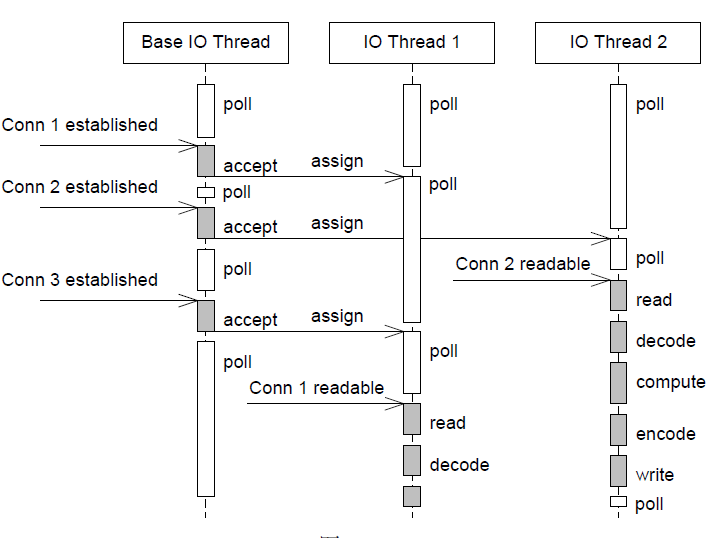
1. Reactor模式的意义：

Doug Schmit指出，其实网络编程中有很多是事物性(routine)的工作，可以提取为公用的框架或库，而用户只需要填上关键的业务逻辑代码，并将回调注册到框架中，就可以实现完整的网络服务，这正是Reactor模式的主要思想。Reactor的意义在于将消息(IO事件)分发到用户提供的(回调)处理函数，并保持网络部分的通用代码不变，独立于用户的业务逻辑。

1. 在使用非阻塞IO ＋事件驱动方式编程注意：

一定要避免在事件回调中执行耗时的操作，包括阻塞IO等，否则会影响程序的响应。

1. muduo内置的多线程方案—Reactors in threads：

这种方案的特点电视one loop per thread，有一个main Reactor负责accept连接，然后把连接挂在某个sub Reactor中(muduo采用轮询的方式来选择sub Reactor)，这样该连接的所有操作都在那个sub Reactor所处的线程中完成。多个连接可被分派到多个线程中，以充分利用CPU。这种方案把IO分派给多个线程，防止出现一个Reactor的处理能力饱和。线程模型如下：****

总结

**实用的网络多线程模型有：thread-per-connection、单线程Reactor、Reactor+线程池（总结8）、one loop per thread、one loop per thread +线程池（第三章总结5），其中muduo支持后四种。**

第七章 muduo编程示例(略)

第八章 muduo网络库设计与实现

1. 粘包问题：

在TCP的长连接流传输中（TCP短连接和UDP中不存在），由于发送端需要等缓冲区满才将数据发送，或者接受方不及时接收缓冲区的包，就会造成“粘包问题”，即本不属于同一消息或同一帧的数据粘在一起被应用层读取。顾名思义，分包就是解决粘包问题的，在TCP这种字节流协议上做应用层分包是网络编程的基本需求。分包有四种方法：

1. 消息长度固定
2. 使用特殊的字符或字符串作为消息的边界，例如HTTP协议的header以\r\n为字段的分隔符
3. 在每条消息的头部加一个长度字段，这是最常用的做法
4. 利用消息本身的格式来分包，如XML等。
5. epoll：Edge and Level Trigger Polling (epoll)：

边缘触发(edge trigger)和条件触发(level trigger)

边缘触发是指每当状态变化时发生一个IO事件，条件触发是只要满足条件就发

生一个IO事件。举个读socket的例子，假定经过长时间的沉默后，现在来了100个字

节，这时无论边缘触发和条件触发都会产生一个read ready notification通知应用程

序可读。应用程序读了50个字节，然后重新调用API等待IO事件。这时条件触发的API会因为还有50个字节可读从而立即返回用户一个read ready notification。而边缘触发 的API会因为可读这个状态没有发生变化而陷入长期等待。因此在使用边缘触发的API时，要注意每次都要读到socket返回EWOULDBLOCK为止，否则这个socket就算废了。而使用条件触发的API时，如果应用程序不需要写就不要关注 socket可写的事件，否则就会无限次的立即返回一个write ready notification。大家常用的select就是属于条件触发这一类。

1. 解决服务端文件描述符使用达到系统上限的有效方式：
2. 准备一个空闲的文件描述符。遇到达到上限的情况，先关闭这个空闲文件，获得一个文件描述符的名额；在accept(2)拿到新socket连接的描述符；随后立刻close(2)它，这样就优雅地断开了客户端的连接；最后重新打开一个空闲文件，继续占有那个空闲的文件描述符，以备再次出现这种情况后使用。但是这种情况在多线程下会出现竞态条件，因为可能多个线程都需要一个文件描述符，那个这仅剩的一个文件描述符就是个竞争资源。
3. 还有一种简单的方法：文件描述符是hard limit（硬限制），我们可以自己设一个稍低一点的soft limit(软限制)，如果超过soft limit就主动关闭新连接，这样就可避免触及“文件描述符耗尽”这种边界条件。这种方法可以积极地防止耗尽file descriptor。