# C++11/14/17 新语言特性重难点解析

对于 gcc/g++ 编译器（最低版本 4.8），如果要使用 C++11/14/17 语言规范，则需要编译时设置相应的选项值。例如将 test.cpp 文件编译成名为 test 的可执行文件，同时需要 C++ 11 语言规范支持，则使用：g++ -g -o test test.cpp -std=c++11；在使用 Makefile 文件编译项目时可以这样指定：make CXXFLAGS="-g -O0 -std=c++11"；在使用 cmake 编译项目时可以在 CMakeLists.txt 文件中添加如下行：set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11 -g -Wall -O0 -Wno-unused-variable")。

**-std** 选项的值可以按需换成 **c++14** 或 **c++17**。

## 1.1 统一的类成员初始化语法与 std::initializer\_list<T>

在 **C++ 11** 标准中，可以使用花括号（即 {}）对任意类型的变量进行初始化，且不用是 static 类型。

**std::initializer\_list<T>**，这是一个模板对象，接收一个自定义参数类型 **T**，**T** 既可以是基础数据类型（如编译器内置的 bool、char、int 等）也可以是自定义复杂数据类型。为了使用 **std::initializer\_list<T>**，需要包含头文件 头文件#include <initializer\_list>。

## 1.2 注解标签（attributes）

C++11 引入了的常用的注解标签有 [[noreturn]]，这个注解的含义是告诉编译器某个函数没有返回值。

C++14 引入了 [[deprecated]] 标签用于表示一个函数或者类型等已经被弃用，当你使用这些被弃用的函数或者类型，编译时编译器会给出相应的警告，有的编译器直接产生编译错误。

## final/override/=default/=delete 语法

final 关键字修饰一个类，这个类将不允许被继承，在 **C++ 11** 中 **final** 关键字要写在类名的后面，其他语言是写在 **class** 关键字的前面。

override 关键字，被 **override** 修饰的类方法是改写父类的同名方法，加了该关键字后在编译阶段，编译器会作相应的检查，如果其父类不存在相同签名格式的类方法，编译器会给出相应的错误提示。

=DEFAULT 语法，如果一个 C++ 类没有显式地给出构造函数、析构函数、拷贝构造函数、operator = 这几类函数的实现，在需要它们时，编译器会自动生成。=default 在开发中简化了那些构造函数中没有实际的初始化代码的写法，尤其是声明和实现分别属于一个 .h 和 \*.cpp 文件。

=DELETE 语法，有强制让编译器生成构造函数、析构函数、拷贝构造函数、operator =的语法，那么也应该有禁止编译器生成这些函数的语法，没错，就是 **=delete。一**般在一些工具类中，我们不需要用到构造函数、析构函数、拷贝构造函数、operator= 这四个函数，为了防止编译器自己生成，同时也是为了减小生成的可执行文件的体积，建议使用 **=delete** 语法将这四个函数“删除”。

## 1.4 auto 关键字

**auto** 用于修饰所有局部变量，即这个变量具有“自动”生命周期。 **auto** 一般会用于让编译器推导一些复杂的模板数据类型，简化语法。

## 1.5 Range-based 循环语法

在 C++ 11 规范中增加了for-each 语法，**for-each** 中的迭代器类型与数组或集合中的**元素的类型**完全一致，for-each 语法中对于复杂数据类型，迭代器是原始数据的拷贝，而不是原始数据的引用，对于容器中是复杂数据类型，尽量使用引用原始数据的方式，以避免复杂数据类型不必要的调用构造函数的开销。

## 1.6 结构化绑定

**std::pair** 一般只能表示两个元素，C++11 标准中引入了 **std::tuple** 类型，有了这个类型，我们就可以放任意个元素了，原来需要定义成结构体的 POD 对象我们可以直接使用 **std::tuple** 表示。

C++17 引入的**结构化绑定**，

auto [a, b, c, ...] = expression;

auto [a, b, c, ...] { expression };

auto [a, b, c, ...] ( expression );

我们可以给用于绑定到目标的变量名起一个有意义的名字。

std::map<std::string, int> cities;

for (const auto& [cityName, cityNumber] : cities)

{

std::cout << cityName << ": " << cityNumber << std::endl;

}

当绑定类型不是基础数据类型时，如果你的本意不是想要得到绑定目标的副本，为了避免拷贝带来的不必要开销，建议使用引用，如果不需要修改绑定目标建议使用 const 引用。

## 1.7 stl 容器新增的实用方法介绍

C++ 11 提供了替代 push\_back 的方法— **emplace\_back**，对于像 **std::list**、**std::vector** 这样的容器，其 push/push\_front 方法在 C++11 中也有对应的改进方法即 emplace/emplace\_front 方法。C++ Reference 上将这里的 **emplace** 操作称之为“**原位构造元素**”。

## 1.8 std::thread

C++11 新标准引入了一个新的类 std::thread（需要包含头文件<thread>），使用这个类的可以将任何签名形式的函数作为线程函数。使用 **std::thread** 类时，必须保证线程函数运行期间，其线程对象有效。**std::thread** 对象提供了一个 **detach** 方法，这个方法让**线程对象**与**线程函数**脱离关系，这样即使**线程对象**被销毁，仍然不影响**线程函数**的运行。然而，在实际编码中，这也是一个不推荐的做法，原因是我们需要使用**线程对象**去控制和管理线程的运行和生命周期。所以，我们的代码应该尽量保证**线程对象**在线程运行期间有效，而不是单纯地调用 **detach** 方法使线程对象与线程函数的运行分离。

## 1.9 thread\_local

C++11 标准提供了一个新的关键字 **thread\_local** 来定义一个线程变量。

thread\_local int t\_data = 1; 对于线程变量，每个线程都会有该变量的一个拷贝，并行不悖，互不干扰。系统的**线程局部存储区域**内存空间并不大，所以尽量不要利用这个空间存储大的数据块，如果不得不使用大的数据块，可以将大的数据块存储在堆内存中，再将该堆内存的地址指针存储在线程局部存储区域。

## 1.10 std::mutex和std::condition\_variable

 C++ 11 标准中新增的用于线程同步的 **std::mutex** 和 **std::condition\_variable** 对象的用法。

mutex,最基本的互斥量

timed\_mutex,有超时机制的互斥量

recursive\_mutex,可重入的互斥量

recursive\_timed\_mutex结合timed\_mutex 和 recursive\_mutex 特点的互斥量

lock\_guard,基于作用域的互斥量管理

unique\_lock,更加灵活的互斥量管理

shared\_lock,共享互斥量的管理

C++ 11 提供了 std::condition\_variable 这个类代表条件变量，与 Linux 系统原生的条件变量一样，同时提供了等待条件变量满足的 wait 系列方法（wait、wait\_for、wait\_until 方法），发送条件信号使用 notify 方法（notify\_one 和 notify\_all 方法）。

使用 std::condition\_variable 对象时需要绑定一个 std::unique\_lock 或 std::lock\_guard 对象。

在 Linux 平台上，std::condition\_variable 也存在虚假唤醒这一现象：  
while(条件不满足){

condition\_wait(cond, mutex);

}

## 1.11 原子操作类

C++ 11提供了对整型变量原子操作的相关库，即 std::atomic ，这是一个模板类型。可以传入具体的整型类型（如bool、char、short、int、uint等）对模板进行实例化，实际上 stl 库也提供了这些实例化的模板类型，如std::atomic\_bool，std::atomic\_char…

## 1.12 智能指针类

std::unique\_ptr、std::shared\_ptr 和 std::weak\_ptr。

std::unique\_ptr 对其持有的堆内存具有唯一拥有权，也就是说引用计数永远是 1，std::unique\_ptr 对象销毁时会释放其持有的堆内存。C++11 新增了 std::make\_shared() 方法创建一个 std::shared\_ptr 对象，却没有提供相应的 std::make\_unique()方法创建一个 std::unique\_ptr 对象，这个方法直到 C++14 才被添加进来。

std::unique\_ptr 类的拷贝构造函数和赋值运算符（operator =）被标记为 delete。既然 std::unique\_ptr 不能复制，那么如何将一个 std::unique\_ptr 对象持有的堆内存转移给另外一个呢？答案是使用移动构造, std::move()。禁止复制语义也存在特例，即可以通过一个函数返回一个 std::unique\_ptr。

std::shared\_ptr 持有的资源可以在多个 std::shared\_ptr 之间共享，每多一个 std::shared\_ptr 对资源的引用，资源引用计数将增加 1。多个线程之间，递增和减少资源的引用计数是安全的。

实际开发中，有时候需要在类中返回包裹当前对象（this）的一个 **std::shared\_ptr** 对象给外部使用，C++ 新标准也为我们考虑到了这一点，有如此需求的类只要继承自 std::enable\_shared\_from\_this<T> 模板对象即可。

std::weak\_ptr 是一个不控制资源生命周期的智能指针。std::weak\_ptr 可以从一个 std::shared\_ptr 或另一个 std::weak\_ptr 对象构造，std::shared\_ptr 可以直接赋值给 std::weak\_ptr ，也可以通过 std::weak\_ptr 的 lock() 函数来获得 std::shared\_ptr。

它的构造和析构不会引起引用计数的增加或减少。std::weak\_ptr 可用来解决 std::shared\_ptr 相互引用时的死锁问题（即两个std::shared\_ptr 相互引用，那么这两个指针的引用计数永远不可能下降为 0， 资源永远不会释）。

**std::weak\_ptr** 提供了一个 **expired()** 方法来做这一项检测，返回 true，说明其引用的资源已经不存在了；返回 false，说明该资源仍然存在，这个时候可以使用 **std::weak\_ptr** 的 **lock()** 方法得到一个 **std::shared\_ptr** 对象然后继续操作资源。

//tmpConn\_ 是一个 std::weak\_ptr<TcpConnection> 对象

//tmpConn\_引用的TcpConnection已经销毁，直接返回

if (tmpConn\_.expired())

return;

std::shared\_ptr<TcpConnection> conn = tmpConn\_.lock();

if (conn)

{

//对conn进行操作，省略...

}

**std::weak\_ptr** 的正确使用场景是那些资源如果可能就使用，如果不可使用则不用的场景。

在 32 位机器上，**std\_unique\_ptr** 占 4 字节，**std::shared\_ptr** 和 **std::weak\_ptr** 占 8 字节；在 64 位机器上，**std\_unique\_ptr** 占 8 字节，**std::shared\_ptr** 和 **std::weak\_ptr** 占 16 字节。也就是说，**std\_unique\_ptr** 的大小总是和原始指针大小一样，**std::shared\_ptr** 和 **std::weak\_ptr** 大小是原始指针的一倍。

智能指针在使用时应注意：

一旦一个对象使用智能指针管理后，就不该再使用原始裸指针去操作；

除了 std::weak\_ptr，std::unique\_ptr 和 std::shared\_ptr 都提供了获取原始指针的方法——get() 函数。

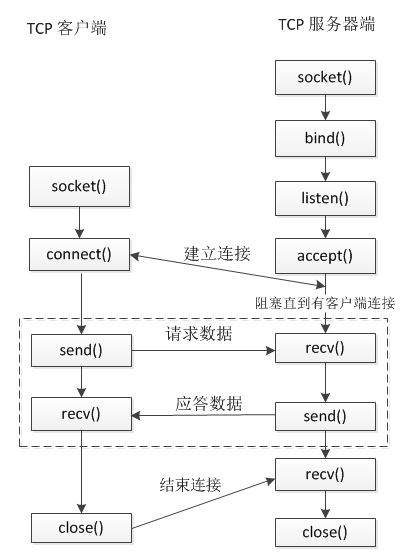
避免操作某个引用资源已经释放的智能指针。

作为类成员变量时，一般在 \*.h 文件对于**指针类型**尽量使用前置声明，而不是直接包含对应类的头文件。在头文件中当使用智能指针对象作为类成员变量时，也应该优先使用前置声明去引用智能指针对象的包裹类，而不是直接包含包裹类的头文件。

# C/C++ 多线程编程

# 网络通信基础重难点

## 3.1 TCP 通信基本流程



为什么客户端调用 close() ，会和服务器端 recv() 函数有关。这个涉及到 recv() 函数的返回值意义。

## 3.2 bind函数分析

INADDR\_ANY 相当于地址 0.0.0.0。

TCP 通信双方中一般服务器端端口号是固定的，而客户端端口号是连接发起时由操作系统随机分配的（不会分配已经被占用的端口）。端口号是一个 C short 类型的值，其范围是0～65535。

## 3.2 select函数分析

Linux下：select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*ecxeptfd, struct timeval \*timeout);

在 Linux 平台上，select 函数的第一个参数必须设置成需要检测事件的所有 fd 中的最大值加 1。

select 函数的作用是检测一组 socket 中某个或某几个是否有“事件”就绪，这里的“事件”一般分为如下三类:

读事件就绪：

socket 内核中，接收缓冲区中的字节数大于等于低水位标记 SO\_RCVLOWAT，此时调用 recv 或 read 函数可以无阻塞的读该文件描述符， 并且返回值大于0；

TCP 连接的对端关闭连接，此时调用 recv 或 read 函数对该 socket 读，则返回 0；

侦听 socket 上有新的连接请求；

socket 上有未处理的错误。

写事件就绪：

socket 内核中，发送缓冲区中的可用字节数(发送缓冲区的空闲位置大⼩) 大于等于低水位标记 SO\_SNDLOWAT，此时可以无阻塞的写, 并且返回值大于0；

socket 的写操作被关闭(调用了 close 或者 shutdown 函数)（ 对一个写操作被关闭的 socket 进行写操作, 会触发 SIGPIPE 信号）；

socket 使⽤非阻塞 connect 连接成功或失败之后；

异常事件就绪:

socket 上收到带外数据。

select 函数调用前后会修改 readfds、writefds 和 exceptfds 这三个集合中的内容（如果有的话），所以如果想下次调用 select 复用这个变量，在下次调用前再次调用 select 前先使用 FD\_ZERO 将集合清零，然后调用 FD\_SET 将需要检测事件的 fd 再次添.加进去。

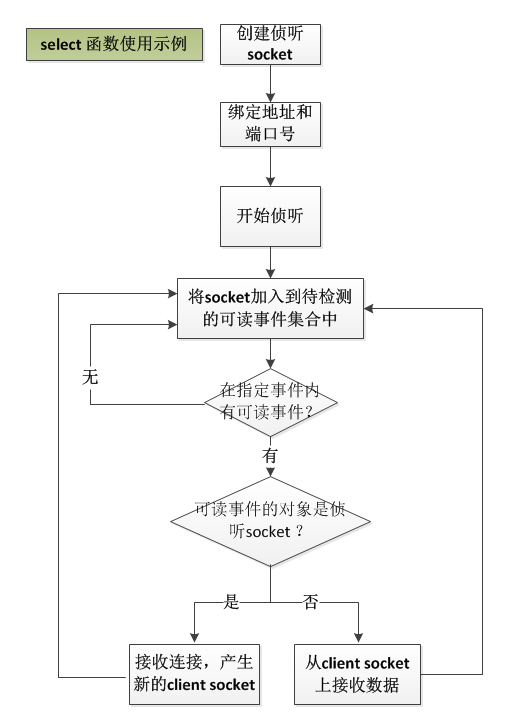
由于不同系统的实现不一样，man 手册的建议将 select 函数修改 timeval 结构体的值的行为当作是未定义的，如果下次使用 select 函数复用这个变量时，得重新赋值。

Linux select 函数的缺点：

每次调用 select 函数，都需要把 fd 集合从用户态拷贝到内核态，这个开销在 fd 较多时会很大，同时每次调用 select 函数都需要在内核遍历传递进来的所有 fd，这个开销在 fd 较多时也很大；

单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，可以通过修改宏定义然后重新编译内核的方式提升这一限制，这样非常麻烦而且效率低下；

select 函数在每次调用之前都要对传入参数进行重新设定，这样做比较麻烦而且会降低性能。



## 3.3阻塞模式和非阻塞模式

阻塞模式，就当某个函数“执行成功的条件”当前不能满足时，该函数会阻塞当前执行线程，程序执行流在超时时间到达或“执行成功的条件”满足后恢复继续执行。而非阻塞模式恰恰相反。

，即使某个函数的“执行成功的条件”不当前不能满足，该函数也不会阻塞当前执行线程，而是立即返回，继续运行执行程序流。

阻塞和非阻塞模式下，我们常讨论的具有不同行为表现的 socket 函数一般有如下几个：

connect

accept

send (Linux 平台上对 socket 进行操作时也包括 write 函数，下文中对 send 函数的讨论也适用于 write 函数)

recv (Linux 平台上对 socket 进行操作时也包括 read 函数，下文中对 recv 函数的讨论也适用于 read 函数)。

无论是 Windows 还是 Linux 平台，默认创建的 socket 都是阻塞模式的。

在 Linux 平台上，可以使用 fcntl() 函数或 ioctl() 函数给创建的 socket 增加 O\_NONBLOCK 标志来将 socket 设置成非阻塞。

int oldSockFlag = fcntl(sockfd, F\_GETFL,0);

int newSockFlag = oldSockFlag | O\_NONBLOCK;

fcntl(sockfd, F\_SETFL, newSockFlag);

Linux 下的socket() 创建函数也可以直接在创建时将 socket 设置为非阻塞模式:

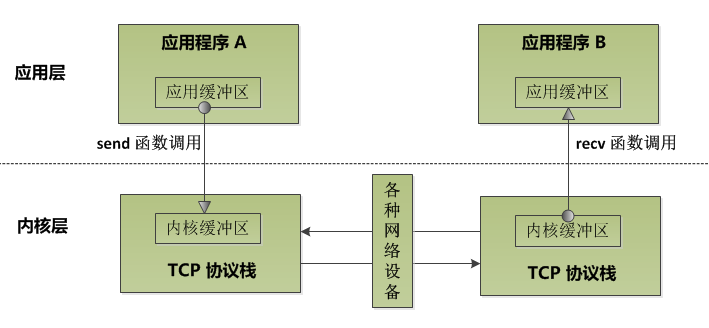
int s = socket(AF\_INET, SOCK\_STREAM | SOCK\_NONBLOCK, IPPROTO\_TCP);

Linux 系统下利用 accept() 函数返回的代表与客户端通信的 socket 也提供了一个扩展函数 accept4()，直接将 accept 函数返回的 socket 设置成非阻塞的。

**send和recv函数在阻塞和非阻塞模式下的行为：**

send 函数本质上并不是往网络上发送数据，而是将应用层发送缓冲区的数据拷贝到内核缓冲区（下文为了叙述方便，我们以“网卡缓冲区”代指）中去，至于什么时候数据会从网卡缓冲区中真正地发到网络中去要根据 TCP/IP 协议栈的行为来确定，这种行为涉及到一个叫 nagel 算法和 TCP\_NODELAY 的 socket 选项。

recv 函数本质上也并不是从网络上收取数据，而只是将内核缓冲区中的数据拷贝到应用程序的缓冲区中，当然拷贝完成以后会将内核缓冲区中该部分数据移除。



不同的程序进行网络通信时，发送的一方会将内核缓冲区的数据通过网络传输给接收方的内核缓冲区。在应用程序 A 与 应用程序 B 建立了 TCP 连接之后，假设应用程序 A 不断调用 send 函数，这样数据会不断拷贝至对应的内核缓冲区中，如果 B 那一端一直不调用 recv 函数，那么 B 的内核缓冲区被填满以后，A 的内核缓冲区也会被填满，此时 A 继续调用 send 函数会是什么结果呢？ 具体的结果取决于该 socket 是否是阻塞模式。

# 网络通信故障排查常用命令

## 4.1 ifconfig

**ifconfig** 是我们最常用的查看当前系统的网卡和 ip 地址信息的命令。

默认情况下，ifconfig 命令只会显示激活的网卡信息，如果要显示所有的（包括未激活）的网卡，我们可以使用 -a 参数。

可以使用 -s 选项显示网卡信息的精简列表。

如果我们想激活或者禁用某个网卡，可以使用 **ifconfig up** 和 **ifconfig down** 命令；

ifconfig 还可以将一个 ip 地址绑定到某个网卡上，或将一个 ip 从某个网卡上解绑:

ifconfig 网卡名 add ip地址

ifconfig 网卡名 del ip地址

## 4.2 ping

如果目标网络不可达，**ping** 命令会一直发送而无输出结果，直到我们按 Ctrl + C 中断，才会统计结果。

ping 命令的目标也可以是一个域名，这样通过 ping 这个域名，我们可以得到这个域名解析后的 ip 地址。

## 4.3 telnet

**telnet ip port** ，可以通过 telnet 命令去检测指定 ip 地址和端口号的侦听服务是否存在。如果不指定端口号，**telnet** 会使用默认 **23** 号端口。

telnet命令不仅可以连接某个服务器还可以与服务器进行交互，这通常用于操作一些接受纯文本数据的服务器程序。

## 4.4 netstat

**netstat** 命令是我们使用的较多的查看网络连接状态的命令。

-a (all)显示所有选项，netstat 默认不显示LISTEN相关。

-t (tcp)仅显示 tcp 相关选项

-u (udp)仅显示 udp 相关选项

-n 不显示别名，能显示数字的全部转化成数字（默认情况下，对于一些有别名的 ip 地址和端口号 **netstat** 会以其别名来显示，例如 **127.0.0.1** 会显示成 **localhost**，**80** 端口会显示成 **http**）。

-l 仅列出有在 Listen (监听) 的服務状态

-p 显示建立相关链接的程序名

-r 显示路由信息，路由表

-e 显示扩展信息，例如uid等

-s 按各个协议进行统计 (重要)

-c 每隔一个固定时间，执行该 netstat 命令

## 4.5 lsof

**lsof**的含义是 list opened filedesciptor （列出已经打开的文件描述符）。在 Linux 系统中，所有的与资源句柄相关的东西都可以统一抽象成文件描述符。

lsof -p pid 能过滤出指定的进程打开的 fd 信息。

lsof 命令第一栏进程名在显示的时候，默认显示前 n 个字符，如果我们需要显示完整的进程名以方便过滤的话，可以使用 +c 选项, 如果你设置值太大， lsof 便不会采用你设置的最大值，而是使用默认最大值。

socket 也是一种 fd，如果需要仅显示系统的网络连接信息，使用的是 -i 选项即可，这个选项可以形象地显示出系统当前的出入连接情况。

和 netstat 命令一样，lsof -i 默认也会显示 ip 地址和端口号的别名，我们只要使用 -n 和 -P 选项就能相对应地显示 ip 地址和端口号了，综合起来就是 lsof -Pni。

## 4.6 nc (netcat)

nc命令常见的用法是模拟一个服务器程序被其他客户端连接，或者模拟一个客户端连接其他服务器，连接之后就可以进行数据收发。

模拟一个服务器程序：

使用 -l 选项（单词 listen 的第一个字母）在某个 ip 地址和端口号上开启一个侦听服务，以便让其他客户端连接。**通常为了显示更详细的信息，会带上 -v 选项**。

模拟一个客户端程序：

用 **nc** 命令模拟一个客户端程序时，我们不需要使用 **-l** 选项，直接写上 ip 地址（或域名，**nc** 命令可以自动解析域名）和端口号即可，如nc –v ww.baidu.com 80。

可以使用 nc 命令作为客户端时可以使用 -p 选项指定使用哪个端口号连接服务器。

如果在使用 nc 命令发消息时不小心输入错误，可以使用 Ctrl + Backspace 键删除。

nc命令不仅可以发消息，也可以发文件（服务器端保存接收的文件名时不一定要使用客户端发送的文件名）：

nc –l ip地址 端口号 > 接受的文件名

nc ip地址 端口号 < 发送的文件名

## 4.7 curl

curl 命令是 Linux 上可以模拟发送 http 请求的一个非常常用的命令。

http 常用的请求方式是 GET 和 POST 两种方式，我们可以使用 -X 选项来显式指定请求是 GET 还是 POST 方式（不指定使用默认 GET 方式）。

如果使用 **GET** 方式，**curl** 提供了另外一个专门的选项 **-G** （或 **–get**）来设置, 如果是 **POST** 方法，除了需要使用 **-X** 选项（或 **–request**）指定请求方法，还需要使用 **-d** （或 **–data**）指定 POST 的数据内容。

有时候需要在发送 http 请求时设置增加一些头部信息，可以使用 **-H**（或 **–header**）指定，如果有多个选项可以使用多次 **-H** 来逐一设置。

如果希望 http 的应答结果中包含 http 头部信息，可以使用 **-i** 选项（或 **–include**）。

如果只想显示 http 应答的头部信息（**不是 html 文档头部** ）可以使用 **-I** 选项（或 **–head**）。

http 协议的头部每一行以 \r\n 结束，整个头部再以一个 额外的 \r\n 结束。

## 4.8 tcpdump

如果要使用 **tcpdump** 命令必须具有 **sudo** 权限。

**tcpdump** 常用的选项有：

**-i** 指定要捕获的目标网卡名，如果要抓所有网卡的上的包，可以使用 **any** 关键字；

**-X** 以 ASCII 和十六进制的形式输出捕获的数据包内容，减去链路层的包头信息；**-XX** 以 ASCII 和十六进制的形式输出捕获的数据包内容，包括链路层的包头信息；

**-n** 不要将 ip 地址显示成别名的形式；**-nn** 不要将 ip 地址和端口以别名的形式显示；

**-S** 以绝对值显示包的 ISN 号（包序列号），默认以上一包的偏移量显示；

**-vv** 抓包的信息详细地显示；**-vvv** 抓包的信息更详细地显示；

**-w** 将抓取的包的原始信息（不解析，也不输出）写入文件中，后跟文件名；

**-r** 从利用 **-w** 选项保存的包文件中读取数据包信息；

除了可以使用选项以外，**tcpdump** 还支持各种数据包过滤的表达式：

## 仅显示经过端口 8888 上的数据包（包括tcp:8888和udp:8888）：

tcpdump -i any 'port 8888'

## 仅显示经过端口是 tcp:8888 上的数据包：

tcpdump -i any 'tcp port 8888'

## 仅显示源端口是 tcp:8888 的数据包：

tcpdump -i any 'tcp src port 8888'

## 仅显示源端口是 tcp:8888 或目标端口是 udp:9999 的包：

tcpdump -i any 'tcp src port 8888 or udp dst port 9999'

## 仅显示源地址是127.0.0.1 且源端口是 tcp:9999 的包 ，以 ASCII 和十六进制显示详细输出，不显示 ip 地址和端口号的别名：

tcpdump -i any 'src host 127.0.0.1 and tcp src port 9999' -XX -nn –vv

假如我们连接的服务器 ip 地址存在，但监听端口号不存在，这个时候客户端发送 **SYN**，服务器应答 **ACK+RST**，这个应答包会导致客户端的 connect 连接失败返回。

实际情形中，还存在一种情况是客户端访问一个很遥远的 ip，或者网络繁忙，服务器对客户端发送的 TCP 三次握手的网络 SYN 报文没有应答，tcpdump 抓取数据包会重试，重试的时间间隔是1 秒，2秒，4秒，8秒，16秒，最后返回超时失败。这个重试次数在 **/proc/sys/net/ipv4/tcp\_syn\_retries** 内核参数中设置，默认为 **6** 。

# GDB 调试

# 性能服务的基本结构

# 高性能服务的其他组件设计

# libevent源码深度剖析

## 8.1 Reactor模式

1）事件源：Linux上是文件描述符，统一称为“句柄集”。

2) 多路分发机制：由操作系统提供的I/O多路复用机制，比如**select**和**epoll**。程序首先将其关心的句柄（事件源）及其事件注册到event demultiplexer上；当有事件到达时，event demultiplexer会发出通知“在已经注册的句柄集中，一个或多个句柄的事件已经就绪”； 程序收到通知后，就可以在非阻塞的情况下对事件进行处理了。

libevent使用结构体**eventop**进行了封装，以统一的接口来支持这些I/O多路复用机制，达到了对外隐藏底层系统机制的目的

3） Reactor--反应器：是事件管理的接口，内部使用event demultiplexer注册、注销事件，并运行事件循环，当有事件进入就绪状态，调用注册事件的回调函数处理事件。

对应到libevent中就是event\_base结构体。

一个典型的Reactor声明：

class Reactor

{

public:

int register\_handler(EventHandler \*phandle,int event);

int remove\_handler();

void handle\_events(timeval \*ptv);

}

4) Event handler--事件处理程序: 事件处理程序提供了一组接口，每个接口对应了一种类型的事件, 供Reactor在相应的事件发生时调用，执行相应的事件处理。通常它会绑定一个有效的句柄。

对应到libevent中，就是event结构体。

下面是两种典型的Event Handler类声明方式:

class Event\_Handler{

public:

virtual void handle\_read() = 0;

virtual void handle\_write() = 0;

virtual void handle\_timeout() = 0;

virtual void handle\_close() = 0;

virtual HANDLE get\_handle() = 0;

// ...

};

class Event\_Handler{

public:

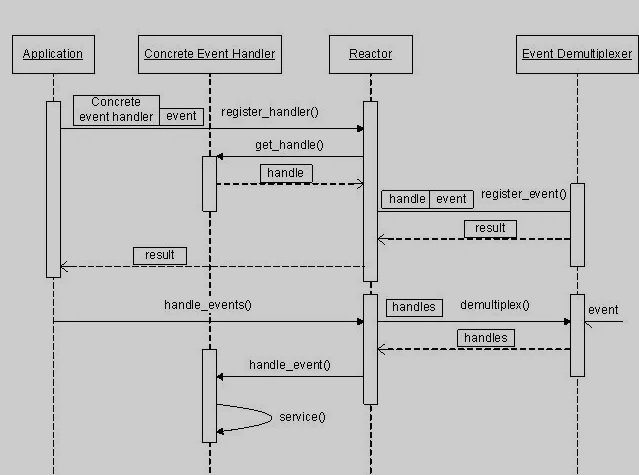
// events maybe read/write/timeout/close .etc

virtual void handle\_events(int events) = 0;

virtual HANDLE get\_handle() = 0;

// ...

};



## 8.2 libevent基本使用场景和事件流程

使用livevent设置定时器:

1. 初始化libevent库，并保存返回的指针

struct event\_base \*base = event\_init();

1. 初始化事件event，设置回调函数和关注的事件：

evtimer(&ev, timer\_cb, NULL);

event\_set的函数原型是：

void event\_set(struct event \*ev, int fd, short event, void (\*cb)(int, short, void \*), void \*arg);

ev：执行要初始化的event对象；

fd：该event绑定的“句柄”，对于信号事件，它就是关注的信号；

event：在该fd上关注的事件类型，它可以是EV\_READ, EV\_WRITE, EV\_SIGNAL；

cb：这是一个函数指针，当fd上的事件event发生时，调用该函数执行处理，它有三个参数，调用时由event\_base负责传入，按顺序，实际上就是event\_set时的fd, event和arg；

arg：传递给cb函数指针的参数

由于定时事件不需要fd，并且定时事件是根据添加时（event\_add）的超时值设定的，因此这里event也不需要设置。

这一步相当于初始化一个event handler，在libevent中事件类型保存在event结构体中。

注意：libevent并不会管理event事件集合，这需要应用程序自行管理；

1. 设置event从属的event\_base

event\_base\_set(base, &ev);

1. 正式添加事件

event\_add(&ev, timeout);

1. 程序进入死循环，等待就绪事件进行处理

event\_base\_dispatch(base);

struct evnt ev;

strcut timeval tv;

void time\_cb(int fd, short event,void \*argc)

{

printf(“timer wakeup”);

event\_add(&ev, &tv);  
}

int main()

{

struct event\_base \*base = event\_init();

tv.tv\_sec = 10;

tv.tv\_usec = 0;

evtimer\_set(&ev, time\_cb, NULL);

event\_base\_dispatch(base);  
}

事件处理流程:

1）首先应用程序准备并初始化event，设置好事件类型和回调函数；这对应于前面第步骤2和3；

2）向libevent添加该事件event。对于定时事件，libevent使用一个小根堆管理，key为超时时间；对于Signal和I/O事件，libevent将其放入到等待链表（wait list）中，这是一个双向链表结构；

3）程序调用event\_base\_dispatch()系列函数进入无限循环，等待事件，以select()函数为例；每次循环前libevent会检查定时事件的最小超时时间tv，根据tv设置select()的最大等待时间，以便于后面及时处理超时事件；

当select()返回后，首先检查超时事件，然后检查I/O事件；

Libevent将所有的就绪事件，放入到激活链表中；

然后对激活链表中的事件，调用事件的回调函数执行事件处理；

## 8.3 libevent事件处理框架

## libevent的核心：事件event

event就是Reactor框架中的事件处理程序组件；它提供了函数接口，供Reactor在事件发生时调用，以执行相应的事件处理，通常它会绑定一个有效的句柄。

struct event {

  TAILQ\_ENTRY (event) ev\_next;

  TAILQ\_ENTRY (event) ev\_active\_next;

  TAILQ\_ENTRY (event) ev\_signal\_next;

  unsigned int min\_heap\_idx;  /\* for managing timeouts \*/

  struct event\_base \*ev\_base;

  int ev\_fd;

  short ev\_events;

  short ev\_ncalls;

  short \*ev\_pncalls;  /\* Allows deletes in callback \*/

  struct timeval ev\_timeout;

  int ev\_pri;   /\* smaller numbers are higher priority \*/

  void (\*ev\_callback)(int, short, void \*arg);

  void \*ev\_arg;

  int ev\_res;   /\* result passed to event callback \*/

  int ev\_flags;

};

**ev\_events：event关注的事件类型**，它可以是以下类型：

# redis