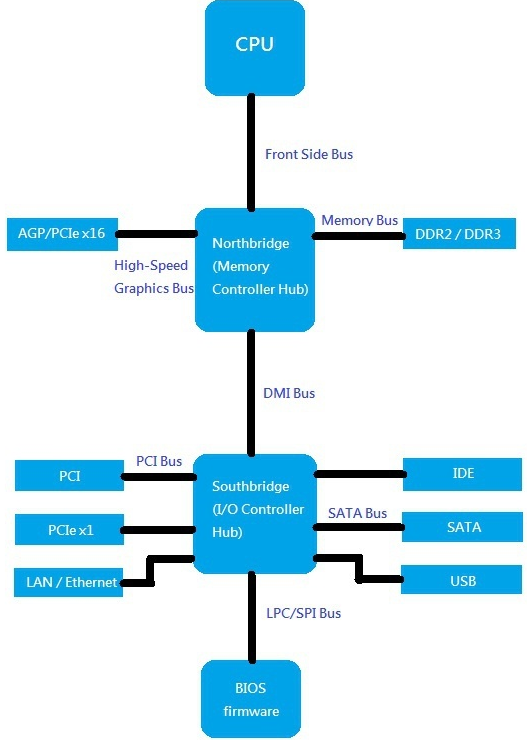
# C++ Note v3



承接《C++ Note v3》本文档主要包含：cpu、设计模式等内容

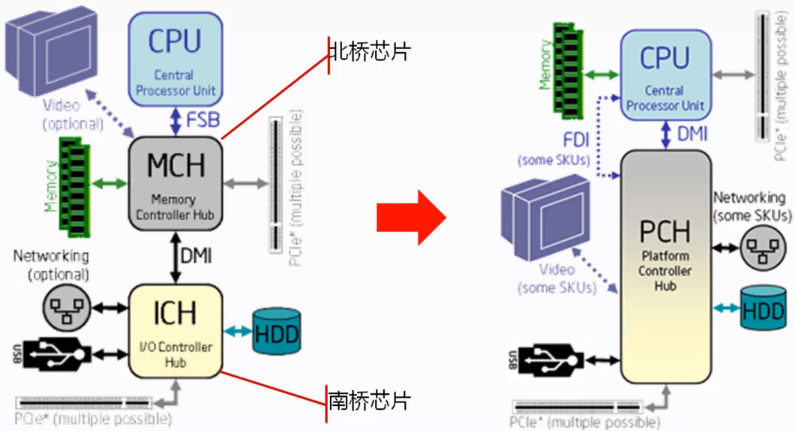
# 主板芯片组架构：

早先时候，计算机主板上有CPU、北桥（MCH）和南桥（ICH）这三个主要芯片。

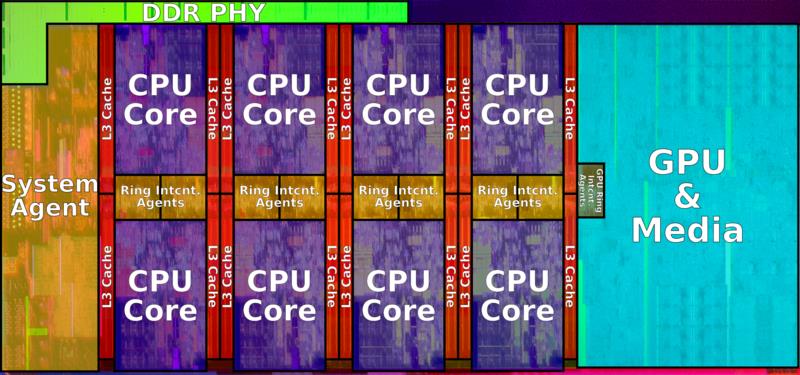
之后由于FSB变成了系统效能的瓶颈，MCH被集成进CPU中。

CPU中MCH原来的部分，在桌面CPU中叫做System Agent（SA），在服务器CPU中叫做uncore（和内核core对应）。它基本还负责原来的功能，即内存管理和提供至少16个Lane的PCIe Root port来驱动显卡

ICH到PCH的转变相对很小，如果把PCH也整合进CPU，则为单芯片解决方案，即SoC（system on a chip）

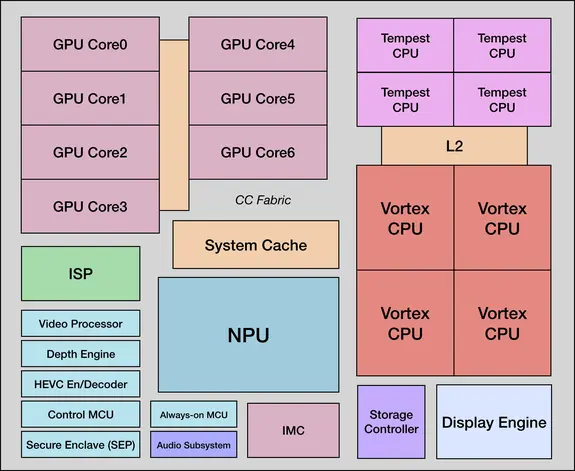


Intel core i9-9900K：



芯片中间紫色部分是8个CPU核心（俗称8核i9），8个核心共享红色的L3 cache；左边是System Agent；右边蓝色的是“核显”Intel® UHD Graphics 630。

apple A12X即是SoC封装：



由上图可以看到，A12X系统一共8核CPU（4大Vortex，4小tempest）；神经网络处理器NPU在芯片中间；GPU在左上角；左下角有很多CPU之外的芯片，比如ISP(image signal Processor)，景深Depth Engine等。

# lambda捕获列表初始化

C++ 14支持lambda capture initializers

//按值捕获target，但是在Lambda内部的变量名叫做v

auto cnt =std::count\_if(books.begin(), books.end(), [v = target](const std::string& book) {return book.find(v) != std::string::npos;});

//按引用捕获target，但是在Lambda内部的名字叫做r

auto cnt =std::count\_if(books.begin(), books.end(), [&r = target](const std::string& book) {return book.find(r) != std::string::npos;});

Lambda捕获列表初始化最重要的一点是“支持Capture by Move”。在C++14之前，Lambda是不支持捕获一个Move-Only的对象的，比如：

std::unique\_ptr<int> uptr = std::make\_unique<int>(123);

auto callback = [uptr]() { // 编译错误，uptr is move-only

std::cout << \*uptr << std::endl;

};

通过捕获列表初始化，完成Move-Only对象的“Capture by Move”：

std::unique\_ptr<int> uptr = std::make\_unique<int>(123);

auto callback = [uptr = std::move(uptr)]() { //将uptr移动给Lambda表达式中的参数

std::cout << \*uptr << std::endl;

};

//... 将callback传给另一个线程

//return=>uptr是nullptr

# 堆：

heap本质上是用array或者vector实现的完全二叉树，二叉树的root节点代表整个heap的最大值（大根堆）或最小值（小根堆）

C++并没有将heap作为容器，而是作为算法放到<algorithm>中，默认是大根堆，可以通过指定比较算法构造小根堆。常用的API有以下几个：

* std::make\_heap(RandomIt first, RandomIt last, Compare comp): 在范围 [first, last) 中构造最大堆。
* std::push\_heap(RandomIt first, RandomIt last, Compare comp): 范围[first, last-1)已经是最大堆，将位于位置last-1的元素插入堆中。
* std::pop\_heap(RandomIt first, RandomIt last, Compare comp): 交换在位置 first 的值和在位置 last-1 的值，并令子范围 [first, last-1) 变为堆。这拥有从范围 [first, last) 所定义的堆移除首个元素的效果。

make\_heap之后的堆顶元素需要使用front()，push\_heap之后max元素在front()，pop\_heap之后[max元素](https://www.zhihu.com/search?q=max%E5%85%83%E7%B4%A0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22452217326%22%7D)在back()

typedef std::vector<int> Heap;

Heap ints = {10,8,9,3,5,6,7,1,2,4};

std::make\_heap(ints.begin(), ints.end()); //front是10

std::pop\_heap(ints.begin(), ints.end()); //back是10，front是9

ints.pop\_back();

ints.push\_back(12);

std::push\_heap(ints.begin(), ints.end()); //front是12

# if(bool)汇编：

eax寄存器：累加器

MOVZX dst,src

将操作数src零扩展,再传给dst，操作数src空间必须小于操作数dst

MOVZX用0来扩展填充操作数dst的余下空间。MOVSX用操作数src的符号位扩展填充操作数A的余下空间，如果是负数则符号位为1，如果是正数则和MOVZX功能相同

AND dst,src

按位与，结果置于dst中。AND 指令总是清除溢出和进位标志位，并根据目标操作数的值来修改符号标志位、零标志位和奇偶标志位

TEST dst,src

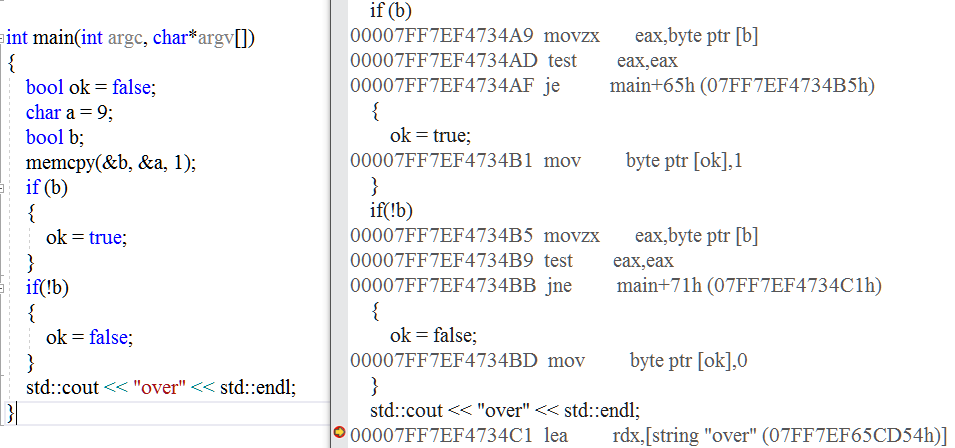
逻辑与运算，两个操作数不会被改变，根据运算结果设置符号标志位、零标志位和奇偶标志位

test eax，100b；b后缀意为二进制

jnz \*\*\*\*\*\*；如果eax右数第三个位为1，jnz将会跳转

jnz或jne跳转的条件是ZF=0, ZF=0意味着ZF(零标志)没被置位，即按位与的结果非0

jz或je跳转的条件是ZF=1，即ZF(零标志)被置位



if(b)命中的条件是：字节b中有任一（或多）bit不为0

if(!b)命中的条件是：字节b中所有bit为0

**SIMD：**

SSE(Streaming SIMD Extensions， 流式SIMD扩展)指令集

SSE引入了16个新的128位寄存器(XMM0至XMM15)

SSE支持单个寄存器存储4个32位单精度浮点数

之后的SSE2则支持单个寄存器存储2个64位双精度浮点数，2个64位整数或4个32位整数或8个16位短整形

AVX（Advanced Vector Extensions）

AVX引入了16个256位寄存器(YMM0至YMM15)，AVX的256位寄存器和SSE的128位寄存器存在着相互重叠的关系(XMM寄存器为YMM寄存器的低位)，所以最好不要混用AVX与SSE指令集，否在会导致transition penalty(过渡处罚)

AVX与SSE支持的数据类型如下：



使用simd指令的方法之一是使用内置函数(intrinsics)

//使用SSE \_mm\_add\_ps 内置函数，一次执行8个单精度浮点数的加法

#include <xmmintrin.h>

int main()

{

\_\_m128 v0 = \_mm\_set\_ps(1.0f, 2.0f, 3.0f, 4.0f);

\_\_m128 v1 = \_mm\_set\_ps(1.0f, 2.0f, 3.0f, 4.0f);

\_\_m128 result = \_mm\_add\_ps(v0, v1);

}

SSE/AVX指令主要定义于以下一些头文件中：

immintrin.h头文件：定义了AVX指令, 支持同时操作8个单精度浮点数或4个双精度浮点数。

同时immintrin.h还包括了SSE~SSE4.2的头文件

SSE/AVX提供的数据类型和函数的命名规则如下：

* 数据类型通常以\_mxxx(T)的方式进行命名，其中xxx代表数据的位数，如SSE提供的\_\_m128为128位，AVX提供的\_\_m256为256位。T为类型，若为单精度浮点型则省略，若为整形则为i，如\_\_m128i，若为双精度浮点型则为d，如\_\_m256d。
* 操作浮点数的内置函数命名方式为：\_mm(xxx)\_name\_PT。 xxx为SIMD寄存器的位数，若为128m则省略，如\_mm\_addsub\_ps，若为\_256m则为256，如\_mm256\_add\_ps。 name为函数执行的操作的名字，如加法为\_mm\_add\_ps，减法为\_mm\_sub\_ps。 P代表的是对矢量(packed data vector)还是对标量(scalar)进行操作，如\_mm\_add\_ss是只对最低位的32位浮点数执行加法，而\_mm\_add\_ps则是对4个32位浮点数执行加法操作。 T代表浮点数的类型，若为s则为单精度浮点型，若为d则为双精度浮点，如\_mm\_add\_pd和\_mm\_add\_ps。
* 操作整形的内置函数命名方式为：\_mm(xxx)\_name\_epUY。xxx为SIMD寄存器的位数，若为128位则省略。 name为函数的名字。U为整数的类型，若为无符号类型则为u，否在为i，如\_mm\_adds\_epu16和\_mm\_adds\_epi16。Y为操作的数据类型的位数，如\_mm\_cvtpd\_pi32。

算术运算：

\_\_m128 \_mm\_dp\_ps(\_\_m128, \_\_m128, const int /\* mask \*/);

计算点积，但要考虑mask

First performs a SIMD multiplication of the lower four packed single-precision floating-point elements (float32 elements) from the first source vector m1 with corresponding elements in the second source vector m2.

Each of the four resulting single-precision elements is conditionally summed depending on the high four bits in the mask parameter.

The resulting summed value is broadcast to each of the lower 4 positions in the destination vector, if the corresponding lower bit of the mask is "1". If the corresponding lower bit of the mask is zero, the corresponding lower element in the destination vector is set to zero.

The process is then replicated with the high elements of the source vectors.

\_\_m128 v0 = \_mm\_set\_ps(4.0f, 3.0f, 2.0f, 1.0f);

\_\_m128 v1 = \_mm\_set\_ps(8.0f, 7.0f, 6.0f, 5.0f);//将4个32位浮点数按相反顺序赋值给\_\_m128中的4个浮点数

\_\_m128 r7 = \_mm\_dp\_ps(v0, v1, 0xff); //点乘，{70,70,70,70}

\_\_m128 r8 = \_mm\_dp\_ps(v0, v1, 0xf0); //点乘，{0,0,0,0}

\_\_m128 r9 = \_mm\_dp\_ps(v0, v1, 0xf4); //点乘，{0,0,70,0}

\_\_m128 r10 = \_mm\_dp\_ps(v0, v1, 0x0f); //点乘，{0,0,0,0}

\_\_m128 r11 = \_mm\_dp\_ps(v0, v1, 0x1f); //点乘，{5,5,5,5}

\_\_m128 r12 = \_mm\_dp\_ps(v0, v1, 0x3f); //点乘，{17,17,17,17}

\_\_m128 \_mm\_cmpeq\_ps(\_\_m128 \_A, \_\_m128 \_B);

逐分量比较4个浮点数是否相等，The result in the output vector will be 0xffffffff if the input elements were equal, or 0 otherwise

\_\_m128 v0 = \_mm\_set\_ps(4.0f, 3.0f, 2.0f, 1.0f);

\_\_m128 v1 = \_mm\_set\_ps(8.0f, 7.0f, 2.0f, 5.0f);

\_\_m128 r3 = \_mm\_cmpeq\_ps(v0, v1);//逐分量比较是否相等，{0,-nan,0,0}，即00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00

SSE/AVX提供的逻辑运算操作包括：

* \_mm\_and\_pd对两个数据逐分量and
* \_mm\_andnot\_ps先对第一个数进行not，然后再对两个数据进行逐分量and
* \_mm\_or\_pd对两个数据逐分量or
* \_mm\_xor\_ps对两个数据逐分量xor

{

\_declspec(align(16)) uint8\_t f0[16] = { 0x00, 0x01, 0x03, 0x11 };

\_declspec(align(16)) uint8\_t f1[16] = { 0x01, 0x01, 0x02, 0x44 };

\_\_m128 vf0 = \_mm\_load\_ps((float\*)f0);

\_\_m128 vf1 = \_mm\_load\_ps((float\*)f1);

\_\_m128 r1 = \_mm\_andnot\_ps(vf0, vf1); //01 00 00 44 00 00 00 00 00 00 00 00 00 00 00 00

\_\_m128 r2 = \_mm\_xor\_ps(vf0, vf1); // 01 00 01 55 00 00 00 00 00 00 00 00 00 00 00 00

\_declspec(align(16)) uint8\_t d0[16] = { 0x00, 0x01, 0x03, 0x11 };

\_declspec(align(16)) uint8\_t d1[16] = { 0x01, 0x01, 0x02, 0x44 };

\_\_m128d vd0 = \_mm\_load\_pd((double\*)d0);

\_\_m128d vd1 = \_mm\_load\_pd((double\*)d1);

\_\_m128d r3 = \_mm\_and\_pd(vd0, vd1); //00 01 02 00 00 00 00 00 00 00 00 00 00 00 00 00

\_\_m128d r4 = \_mm\_or\_pd(vd0, vd1);//01 01 03 55 00 00 00 00 00 00 00 00 00 00 00 00

printf("0x%x\n", \*(unsigned\*)&vd0);

}

\_mm\_shuffle\_ps读取两个\_\_m128类型的数据a和b，并按照\_MM\_SHUFFLE提供的索引将返回的\_\_m128类型数据的低两位设置为a中按索引值取得到的对应值，将高两位设置为按索引值从b中取得到的对应值。索引值在0到3之间，分别以相反的顺序对应\_\_m128中的四个浮点数

\_declspec(align(16)) float p0[] = { 1, 2, 3, 4 };

\_declspec(align(16)) float p1[] = { 5, 6, 7, 8 };

\_\_m128 a = \_mm\_load\_ps(p0);

\_\_m128 b = \_mm\_load\_ps(p1);

\_\_m128 v0 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 0, 3, 2)); // 3, 4, 5, 6

\_\_m128 v1 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(2, 2, 3, 3)); // 4, 4, 7, 7

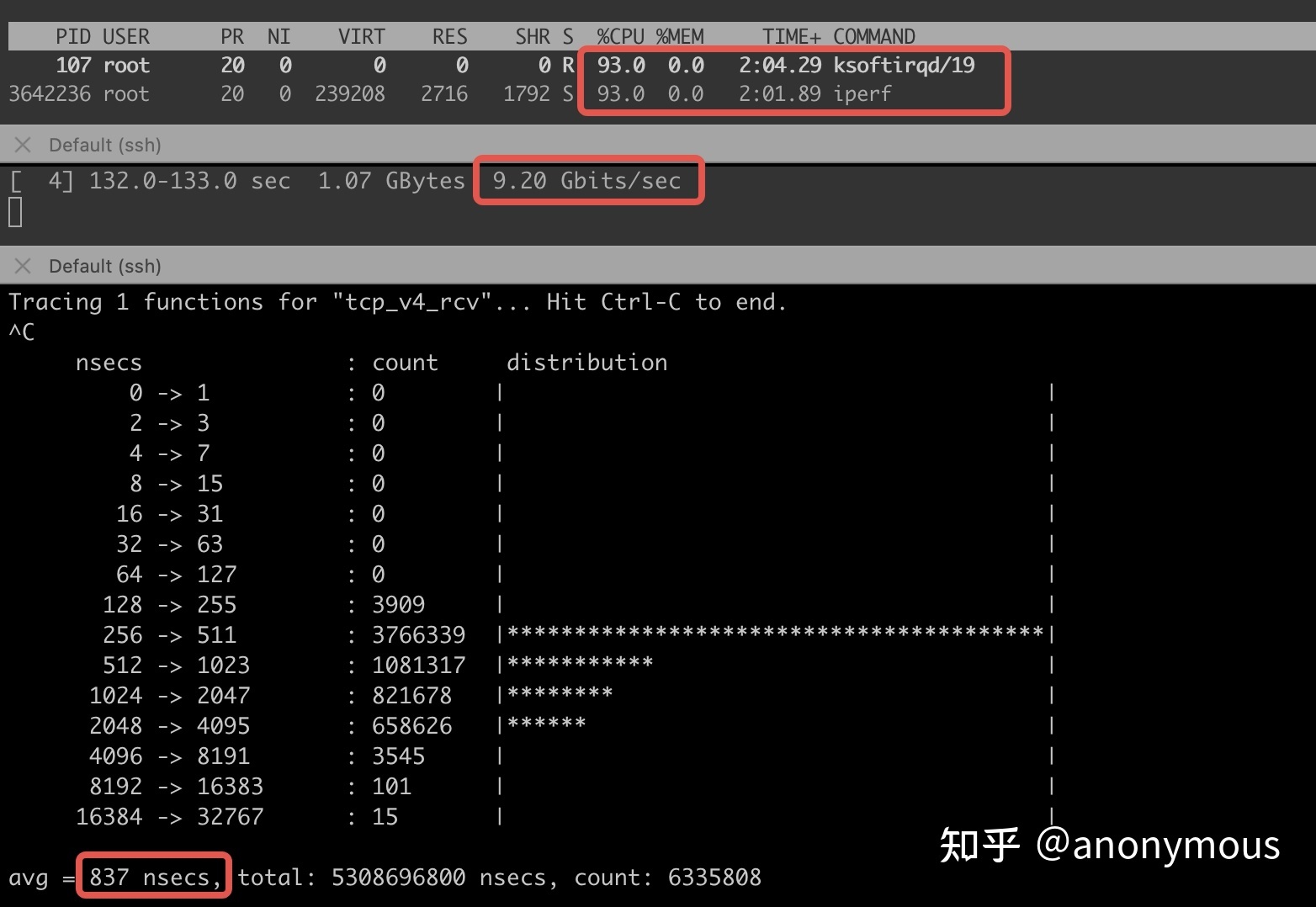
\_\_m128 v2 = \_mm\_shuffle\_ps(a, b, \_MM\_SHUFFLE(1, 3, 2, 3)); //4, 3, 8, 6

**tcp吞吐量：**

tcp单流400Gbps吞吐需要每秒收发(400/8)\*1000\*1000\*1000/1460个数据包，大致34246575个，即每微秒处理34个数据包

对tcp，合适的MSS= MTU（最大传输单元）-IP报文头长度-tcp报文头长度=1500-20-20=1460

大致估算一下 25Gbps 场景下 Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz 的收包能力，关闭 GRO/LRO，下图三栏分别为 top/iperf -s -i 1/funclantency tcp\_v4\_rcv 的结果：



一款不错的 CPU 单核极限能力不到 10Gbps，发送端关闭 TSO/GSO，接收端 CPU 下降，但 tcp\_v4\_rcv 的执行能力已达上限，接收端打开 GRO/LRO，tcp\_v4\_rcv 开销变大，达到 1500us 以上。

**VDSO**

vsyscalls(virtual system calls)

VDSO (Virtual Dynamically Shared Objects)

VDSO是一个名称叫linux-vdso.so.1的.so文件，可以执行ldd `which bash`查看：

[root@localhost study]$ ldd `which bash`

linux-vdso.so.1 => (0x00007ffde4733000)

libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007f15ac77b000)

libdl.so.2 => /lib64/libdl.so.2 (0x00007f15ac577000)

libc.so.6 => /lib64/libc.so.6 (0x00007f15ac1a9000)

/lib64/ld-linux-x86-64.so.2 (0x00007f15ac9a5000)

Linux上，内核每加载一个ELF可执行程序时，都会在其进程地址空间中建立一个叫做vDSO mapping的内存区域

[root@localhost zcj]$ cat /proc/11716/maps

00400000-00402000 r-xp 00000000 fd:02 3358074883 /home/zcj/study/main

……

7fd66c253000-7fd66c254000 rw-p 00022000 fd:00 41952560 /usr/lib64/ld-2.17.so

7ffd01dfa000-7ffd01dfc000 r-xp 00000000 00:00 0 [vdso]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

VDSO加速某些系统调用函数的执行效率。以gettimeofday()为例，该系统调用返回的结果实际并不涉及任何数据安全问题，因为特权用户root和非特权用户都会获得相同的结果。因此与其费尽心力一定要通过陷入内核的方式来读取这些数据（昂贵的用户/内核态上下文切换开销），不如在内核与用户态之间建立一段共享内存区域（VDSO），内核定期“推送”最新值到该区域，调用gettimeofday()时并不真正执行系统调用，而是从该区域直接获取，相当于将系统调用改造成了函数调用。

VDSO支持的系统调用：gettimeofday()、time()、getcpu()、clock\_gettime()

gettimeofday()提供us级精度, clock\_gettime提供ns级精度

#include <sys/time.h>

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

int settimeofday(const struct timeval \*tv, const struct timezone \*tz);

其中，tv参数是从Epoch算起的秒数和微秒数

struct timeval {

time\_t tv\_sec; /\* seconds \*/

suseconds\_t tv\_usec; /\* microseconds \*/

};

一般tz参数置0

函数返回0表示成功，返回-1失败

**TSC：**

x86 cpu提供TSC（time stamp counter）寄存器

用户态可直接用一条RDTSC指令读取TSC，性能消耗远小于借助VDSO的gettimeofday()、clock\_gettime()。

Intel cpu的TSC能力：

1. Variant TSC：TSC的增加受cpu频率变化的影响
2. Constant TSC：TSC以固定速率增加，不受cpu频率变化影响。但当cpu处于deep C-state状态时，TSC停止
3. Invariant TSC：TSC以固定速率增加，无论处于P-state、C-state还是T-state

Linux定义CPU feature bits描述cpu TSC能力：

1. X86\_FEATURE\_TSC：cpu支持TSC
2. X86\_FEATURE\_CONSTANT\_TSC：cpu支持constant TSC.
3. X86\_FEATURE\_NONSTOP\_TSC：与X86\_FEATURE\_CONSTANT\_TSC同时存在时表示cpu支持Invariant TSC

检查cpu tsc能力：cat /proc/cpuinfo | grep -E "constant\_tsc|nonstop\_tsc"

不同cpu的TSC如何保持同步：

1. 无同步机制。在早期SMP系统上，TSC并不会进行同步。这意味着如果程序在读取TSC后被OS切换到另一个核，再次读取TSC时可能出现“时间回退”现象。Variant TSC cpu和Constant TSC cpu存在此问题。
2. 同一主板上的多个CPU间同步。对支持Invariant TSC的cpu，多数SMP系统都会在cpu间同步TSC。在引导阶段，所有使用同一个RESET信号的CPU会被重置，它们的TSC以相同速率增加
3. 没有跨机柜、刀片或主板的同步机制。根据主板和计算机制造商的设计，不同主板上的CPU可能会连接到不同的时钟信号上，无法保证TSC同步

由以上分析可知，即使cpu支持Invariant TSC，SMP系统上，仍然无法保证TSC的可靠性。因此Linux内核执行boot time or runtime testing，如果在内核boot阶段通过了同步性测试，下面的文件会指定TSC作为时钟源：

$ cat /sys/devices/system/clocksource/clocksource0/current\_clocksource

tsc

如果要进行性能测试的代码非常短，最好使用LFENCE 或 RDTSCP 重新实现rdtsc函数。否则，CPU的乱序执行会对测量带来精度问题。CPU并不会等待前面的指令全部执行完再读取TSC值，后续指令同样有可能在读取操作前执行。如果需要在前面指令都执行完后再读取TSC值，开发者可以使用RDTSCP指令（如果CPU支持）或依次使用LFENCE、RDTSC两个指令。

**字符串分割**

查看进程内存泄漏时，pmap -p 34942和cat /proc/34942/maps配合使用

//注意，以下实现有一个缺陷，那就是相连的分隔符会被忽视，如"1,,3,4,5"相连的两个逗号会被当做一个

void split(const std::string& s, std::vector<std::string>& tokens, char delim = ',')

{

std::string::size\_type lastPos = s.find\_first\_not\_of(delim, 0);

std::string::size\_type pos = s.find\_first\_of(delim, lastPos);

while (pos != std::string::npos || lastPos != std::string::npos)

{

tokens.emplace\_back(s.substr(lastPos, pos - lastPos));

lastPos = s.find\_first\_not\_of(delim, pos);

pos = s.find\_first\_of(delim, lastPos);

}

}

**std::thread异常coredump导致调用堆栈丢失问题**

gcc 4.8.5

测试程序：

#include <iostream>

#include <thread>

#include <vector>

void thread\_func(){

std::cout << "thread\_func start ..." << std::endl;

std::vector<int> vec;

//vec.push\_back(1);

//vec.push\_back(2);

std::cout << vec.at(1) << std::endl;

}

int main (void){

std::thread th1(thread\_func);

th1.join();

return 0;

}

g++ -g -o main main.cpp -lpthread -std=c++11

[root@localhost study]$ ldd main

linux-vdso.so.1 => (0x00007ffc90fcf000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f2848112000)

libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f2847e0b000)

libm.so.6 => /lib64/libm.so.6 (0x00007f2847b09000)

libgcc\_s.so.1 => /lib64/libgcc\_s.so.1 (0x00007f28478f3000)

libc.so.6 => /lib64/libc.so.6 (0x00007f2847525000)

/lib64/ld-linux-x86-64.so.2 (0x00007f284832e000)

gdb ./main

(gdb) bt

#0 0x00007ffff7008387 in raise () from /lib64/libc.so.6

#1 0x00007ffff7009a78 in abort () from /lib64/libc.so.6

#2 0x00007ffff79187d5 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler() () from /lib64/libstdc++.so.6

#3 0x00007ffff7916746 in ?? () from /lib64/libstdc++.so.6

#4 0x00007ffff7916773 in std::terminate() () from /lib64/libstdc++.so.6

#5 0x00007ffff796d105 in ?? () from /lib64/libstdc++.so.6

#6 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#7 0x00007ffff70d0b0d in clone () from /lib64/libc.so.6

解决方法：

1. 给线程函数thread\_func()加上noexcept声明

void thread\_func() noexcept {……}

(gdb) bt

#0 0x00007ffff7008387 in raise () from /lib64/libc.so.6

#1 0x00007ffff7009a78 in abort () from /lib64/libc.so.6

#2 0x00007ffff79187d5 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler() () from /lib64/libstdc++.so.6

#3 0x00007ffff7916746 in ?? () from /lib64/libstdc++.so.6

#4 0x00007ffff79156f9 in ?? () from /lib64/libstdc++.so.6

#5 0x00007ffff7916364 in \_\_gxx\_personality\_v0 () from /lib64/libstdc++.so.6

#6 0x00007ffff73af8a3 in ?? () from /lib64/libgcc\_s.so.1

#7 0x00007ffff73afc3b in \_Unwind\_RaiseException () from /lib64/libgcc\_s.so.1

#8 0x00007ffff7916986 in \_\_cxa\_throw () from /lib64/libstdc++.so.6

#9 0x00007ffff796b857 in std::\_\_throw\_out\_of\_range(char const\*) () from /lib64/libstdc++.so.6

#10 0x0000000000401595 in std::vector<int, std::allocator<int> >::\_M\_range\_check (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:794

#11 0x0000000000401313 in std::vector<int, std::allocator<int> >::at (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:812

#12 0x0000000000400fde in thread\_func () at main.cpp:9

#13 0x000000000040262f in std::\_Bind\_simple<void (\*())()>::\_M\_invoke<>(std::\_Index\_tuple<>) (this=0x606040)

at /usr/include/c++/4.8.2/functional:1732

#14 0x0000000000402589 in std::\_Bind\_simple<void (\*())()>::operator()() (this=0x606040)

at /usr/include/c++/4.8.2/functional:1720

#15 0x0000000000402522 in std::thread::\_Impl<std::\_Bind\_simple<void (\*())()> >::\_M\_run() (this=0x606028)

at /usr/include/c++/4.8.2/thread:115

#16 0x00007ffff796d070 in ?? () from /lib64/libstdc++.so.6

#17 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#18 0x00007ffff70d0b0d in clone () from /lib64/libc.so.6

1. gdb catch throw

(gdb)catch throw

(gdb)bt

#0 0x00007ffff7008387 in raise () from /lib64/libc.so.6

#1 0x00007ffff7009a78 in abort () from /lib64/libc.so.6

#2 0x00007ffff79187d5 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler() () from /lib64/libstdc++.so.6

#3 0x00007ffff7916746 in ?? () from /lib64/libstdc++.so.6

#4 0x00007ffff79156f9 in ?? () from /lib64/libstdc++.so.6

#5 0x00007ffff7916364 in \_\_gxx\_personality\_v0 () from /lib64/libstdc++.so.6

#6 0x00007ffff73af8a3 in ?? () from /lib64/libgcc\_s.so.1

#7 0x00007ffff73afc3b in \_Unwind\_RaiseException () from /lib64/libgcc\_s.so.1

#8 0x00007ffff7916986 in \_\_cxa\_throw () from /lib64/libstdc++.so.6

#9 0x00007ffff796b857 in std::\_\_throw\_out\_of\_range(char const\*) () from /lib64/libstdc++.so.6

#10 0x0000000000401595 in std::vector<int, std::allocator<int> >::\_M\_range\_check (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:794

#11 0x0000000000401313 in std::vector<int, std::allocator<int> >::at (this=0x7ffff6fd0e60, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:812

#12 0x0000000000400fde in thread\_func () at main.cpp:9

#13 0x000000000040262f in std::\_Bind\_simple<void (\*())()>::\_M\_invoke<>(std::\_Index\_tuple<>) (this=0x606040)

at /usr/include/c++/4.8.2/functional:1732

#14 0x0000000000402589 in std::\_Bind\_simple<void (\*())()>::operator()() (this=0x606040)

at /usr/include/c++/4.8.2/functional:1720

#15 0x0000000000402522 in std::thread::\_Impl<std::\_Bind\_simple<void (\*())()> >::\_M\_run() (this=0x606028)

at /usr/include/c++/4.8.2/thread:115

#16 0x00007ffff796d070 in ?? () from /lib64/libstdc++.so.6

#17 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#18 0x00007ffff70d0b0d in clone () from /lib64/libc.so.6

1. 修改gcc源码bug

gcc-4.8.5/libstdc++-v3/src/c++11/thread.cc中

**namespace**

{

**extern** "C" **void\***

execute\_native\_thread\_routine(**void\*** \_\_p)

{

**thread::**\_Impl\_base**\*** \_\_t **=** **static\_cast<thread::**\_Impl\_base**\*>**(\_\_p);

**thread::**\_\_shared\_base\_type \_\_local;

\_\_local.swap(\_\_t**->**\_M\_this\_ptr);

**\_\_try**

{

\_\_t**->**\_M\_run();

}

\_\_catch(**const** \_\_cxxabiv1**::**\_\_forced\_unwind**&**)

{

\_\_throw\_exception\_again;

}

\_\_catch(...)

{

std**::**terminate();

}

**return** 0;

}

}

\_\_t->\_M\_run() 是运行线程的代码，当运行过程中触发了throw抛异常操作，且该异常类型为非 \_\_cxxabiv1::\_\_forced\_unwind 类型，则进入了std::terminate分支。由于异常捕获，这里调用堆栈信息已经丢失了了，所以期望看到的调用堆栈信息都没了

修改方法：

      /\*modify

      \_\_try

      {

        \_\_t->\_M\_run();

      }

       \_\_catch(const \_\_cxxabiv1::\_\_forced\_unwind&)

      {

        \_\_throw\_exception\_again;

      }

      \_\_catch(...)

      {

        std::terminate();

      }

      \*/

      \_\_t->\_M\_run();

[root@localhost study]$ export LD\_LIBRARY\_PATH=/home/zcj/gcccompile/gcc-4.8.5/install\_dir/lib64

[root@localhost study]$ vi main.cpp

[root@localhost study]$ g++ -g -o main main.cpp -lpthread -std=c++11

[root@localhost study]$ ldd main

linux-vdso.so.1 => (0x00007fff6051c000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f0f437d6000)

libstdc++.so.6 => /home/zcj/gcccompile/gcc-4.8.5/install\_dir/lib64/libstdc++.so.6 (0x00007f0f434d4000)

libm.so.6 => /lib64/libm.so.6 (0x00007f0f431d2000)

libgcc\_s.so.1 => /home/zcj/gcccompile/gcc-4.8.5/install\_dir/lib64/libgcc\_s.so.1 (0x00007f0f42fbc000)

libc.so.6 => /lib64/libc.so.6 (0x00007f0f42bee000)

/lib64/ld-linux-x86-64.so.2 (0x00007f0f439f2000)

(gdb) bt

#0 0x00007ffff700d387 in raise () from /lib64/libc.so.6

#1 0x00007ffff700ea78 in abort () from /lib64/libc.so.6

#2 0x00007ffff791c705 in \_\_gnu\_cxx::\_\_verbose\_terminate\_handler ()

at ../../.././libstdc++-v3/libsupc++/vterminate.cc:95

#3 0x00007ffff791a896 in \_\_cxxabiv1::\_\_terminate (handler=<optimized out>)

at ../../.././libstdc++-v3/libsupc++/eh\_terminate.cc:38

#4 0x00007ffff791a8c3 in std::terminate () at ../../.././libstdc++-v3/libsupc++/eh\_terminate.cc:48

#5 0x00007ffff791aae3 in \_\_cxxabiv1::\_\_cxa\_throw (obj=0x7ffff0000940,

tinfo=0x7ffff7ba31d0 <typeinfo for std::out\_of\_range>, dest=

0x7ffff792e930 <std::out\_of\_range::~out\_of\_range()>) at ../../.././libstdc++-v3/libsupc++/eh\_throw.cc:87

#6 0x00007ffff796bc17 in std::\_\_throw\_out\_of\_range (\_\_s=<optimized out>)

at ../../../.././libstdc++-v3/src/c++11/functexcept.cc:80

#7 0x00000000004015b7 in std::vector<int, std::allocator<int> >::\_M\_range\_check (this=0x7ffff6fd5e50, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:794

#8 0x0000000000401335 in std::vector<int, std::allocator<int> >::at (this=0x7ffff6fd5e50, \_\_n=1)

at /usr/include/c++/4.8.2/bits/stl\_vector.h:812

#9 0x0000000000400fdf in thread\_func () at main.cpp:9

#10 0x0000000000402651 in std::\_Bind\_simple<void (\*())()>::\_M\_invoke<>(std::\_Index\_tuple<>) (this=0x606040)

at /usr/include/c++/4.8.2/functional:1732

#11 0x00000000004025ab in std::\_Bind\_simple<void (\*())()>::operator()() (this=0x606040)

at /usr/include/c++/4.8.2/functional:1720

#12 0x0000000000402544 in std::thread::\_Impl<std::\_Bind\_simple<void (\*())()> >::\_M\_run() (this=0x606028)

at /usr/include/c++/4.8.2/thread:115

#13 0x00007ffff796d260 in std::(anonymous namespace)::execute\_native\_thread\_routine (\_\_p=<optimized out>)

at ../../../.././libstdc++-v3/src/c++11/thread.cc:96

#14 0x00007ffff7bc6ea5 in start\_thread () from /lib64/libpthread.so.0

#15 0x00007ffff70d5b0d in clone () from /lib64/libc.so.6

**Linux权限管理：**

SUID（SetUID）：文件的一种特殊权限，当设置了SUID位的文件被执行时，该文件将以所有者的身份运行，即无论谁来执行这个文件，他都有文件所有者的特权。如，如果所有者是root的话，那么执行人就有root权限了，因此不要轻易设置该位。

只有文件所有者能为文件设置suid。bash的所有者是root，普通用户想通过给bash设置suid然后拿到root权限是不可能的，这个权限只有root能给。

chmod u+s temp：为temp文件添加suid标志

chmod u-s temp：移除temp文件的suid标志

设置完suid标志后，可以用ls-l来查看。如果有suid标志，则会在原来执行标志x的位置上显示。原来的执行标志x到哪里去了呢？系统规定，如果本来有执行权限，则显示为小写字母s，如果本来无执行权限则显示为大写字母S

-rw-r--r-- 1 root root 3633 Jan 6 02:26 runlog

-rwxr-xr-x 1 root root 1314 Jan 6 02:21 memcheck.sh

chmod u+s runlog

chmod u+s memcheck.sh

-rwSr--r-- 1 root root 3633 Jan 6 02:26 runlog

-rwsr-xr-x 1 root root 1314 Jan 6 02:21 memcheck.sh

su程序有suid权限：

ll `which su`

-rwsr-xr-x. 1 root root 32128 Aug 9 2019 /usr/bin/su

文件

$ ls -l /etc/hosts

-rw-r--r-- 1 root root 619 Oct 1 08:51 /etc/hosts

第一组rw-：  表示文件的拥有者对它的权限

第二组r--：  表示文件的所属组对它的权限

第三组r--：  表示其他用户对这个文件的权限

两个root：分别为文件的拥有者和所属的组

文件上的用户信息和权限信息是文件属性，是静态的，用来限制谁能对该文件执行什么操作

而对文件执行操作最终是由进程完成的，所以进程拥有的权限才是决定能否对文件执行操作的标准。

EUID、RUID、EGID、RGID都是进程的属性

EUID(Effective UID)，有效用户ID；

RUID(Real UID)，实际用户ID

EGID有效组ID；

RGID实际组ID

进程在执行某些涉及权限的操作时，内核将使用进程的【有效用户/组ID】作为凭证来判断是否有权限执行对应操作。例如，rm命令删除文件a.txt时，内核将使用rm进程的EUID、EGID和a.txt文件的用户ID、组ID做比较，进而再比较对应的权限位，从而判断rm进程是否有权限删除a.txt

既然涉及到权限的操作都以有效ID作为依据，那实际ID有什么作用？RUID和RGID用于确定进程所属的用户和组，主要用于判断是否有权限向进程发送信号：对于非特权用户，如果发送信号的进程A的RUID和目标进程B的RUID相同，则进程A可以发送信号。由于子进程会继承父进程的实际ID，所以父子进程的RUID相等，父子进程可互相发信号

$ useradd -m abc //创建用户abc

$ passwd abc //设置用户abc的密码

$ cat /etc/passwd //查看系统中用户列表

$ su abc //切换到用户abc

$ whoami //查看当前用户

$ id //查看当前用户id

uid=1002(abc) gid=1002(abc) groups=1002(abc)

id root //查看root用户id

uid=0(root) gid=0(root) groups=0(root)

$ userdel abc //删除用户abc

用户执行程序时，默认将当前用户或指定的用户(如sudo方式)设置为该进程的EUID和实际用户RUID

进程内部可通过代码修改进程的EUID。例如，以root身份运行的程序，初始时其有EUID为0(即root)，具有特权，之后在程序内部修改其EUID为nobody，修改后该进程将失去大量权限。

为程序文件设置suid，则该程序在执行时，其EUID被设置为程序文件所有者的UID

举个例子，分析用户user1执行/bin/app删除a.txt文件时的身份设置过程（只考虑用户不考虑组）：

1.获取/bin/app权限信息，判断user1是否有权限执行该程序，若有权，则exec加载成功。

2.如果app文件未设置SUID，则设置进程的有效用户ID和实际用户ID都为user1。

3.如果app文件设置了SUID，则设置进程的有效用户ID为app文件的用户ID。

4.app程序内部可能也会通过系统调用修改进程的有效用户ID。

5.app进程执行删除a.txt文件代码时，内核将比较此时app进程的有效用户ID和a.txt文件的用户ID，从而判断权限位

举个例子，分析登录界面的安全性：

登录界面本身可能需要拿到用户头像之类的个性化信息，所以它必须具有root身份，否则就无权翻看所有用户的profile。

但这样一来，当登录界面本身存在问题时，一旦被人抓住漏洞，就被拿到了root权限。这是非常危险的。

怎么办呢？

Linux还有个seteuid调用，这个调用可以拿来放弃root权限。

登录界面初始时有root权限，遍取所有用户的profile；但拿到profile之后，就没必要保留root权限了——调用个seteuid，放弃root身份，以访客身份提供登陆服务！

除非黑客在读取用户profile时成功攻击登录界面，否则拿不到任何好处；但读取profile时登录界面不对外提供服务，对外提供服务时，又已经放弃了root身份——无懈可击！

**网络并发模型**

消费队列数据的推拉方式：

工作线程消费队列数据的方式，分为『推』和『拉』两种模型。

推模型，即若队列无数据，则阻塞休眠，当生产者写入数据到队列后，则会唤醒休眠的工作线程来处理。

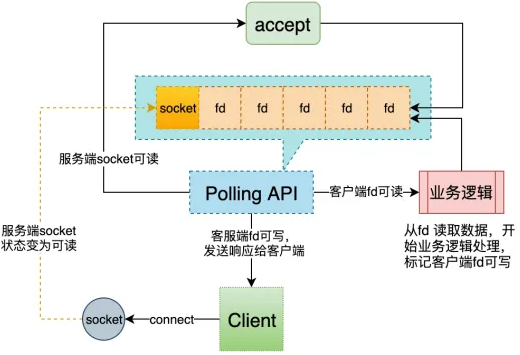
拉模型，即不需要条件变量，工作线程内死循环，不停地轮询查找取出队列数据。

两种模型各有利弊，主要要看实际业务场景，如果是IO密集型的，比如并发度特别高，以至于几乎总能取到数据，那么就采用拉模型。

条件变量的伪唤醒：

1. pthread的条件变量等待pthread\_cond\_wait是使用阻塞的系统调用实现的（比如Linux上的futex），这些阻塞的系统调用在进程被信号中断后，通常会中止阻塞、直接返回EINTR错误。即没有调notify但是被唤醒。
2. 有对应的唤醒，但条件不成立。由于线程调度，被条件变量唤醒的线程在本线程内真正执行「加锁并返回」前，另一个线程抢先插了进来，完整地进行了一套「拿锁、改条件、还锁」的操作。
3. wait、wait\_for和wait\_until，都有带判断式的重载版本，可避免伪唤醒的危害

单线程IO多路复用：



Polling API泛指select/poll/epoll/kqueue这种IO多路复用API

让Polling API监控服务端socket的状态，然后开始死循循环，循环过程中主要有三种逻辑分支：

1. 服务端socket的状态变为可读，则表示有客户端发起连接，此时就调用accept建立连接，得到一个客户端fd。将其加入到Polling API的监控集合，并标记其为可读。
2. 客户端fd的状态变为可读，则调用read/recv从fd读取数据，然后执行业务逻辑，处理完，再将其加入到Polling API的监控集合，并标记其为可写。
3. 客户端fd的状态变为可写，则调用write/send将数据发送给客户端

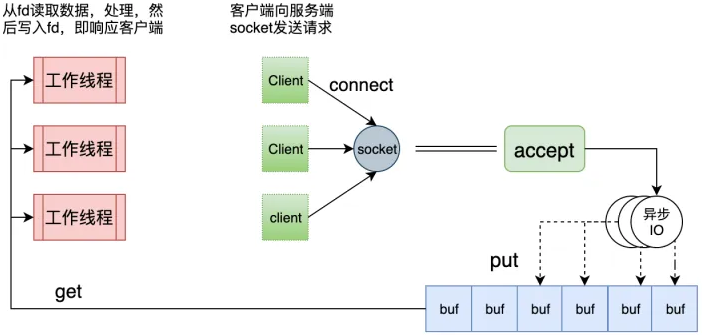
多线程/多进程IO：

主线程创建多个子线程，然后每个子线程内部开启死循环，listen、accept、read、send

借助SO\_REUSEPORT，不同socket可以绑定到相同ip:port上，同时Linux内核提供负载均衡功能

半同步半异步IO：

Half-Sync/Half-Async（HSHA）：异步IO层+队列+同步处理层



一般是一个IO线程和多个工作线程。IO线程负责异步地从客户端fd获取客户端的请求数据，而工作线程则是并发的对该数据进行处理

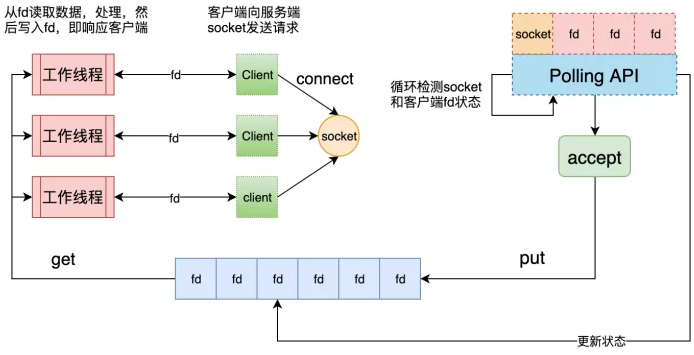
异步IO，调read后立即返回不阻塞，有数据到来时，内核自动将数据拷贝到用户缓冲区，然后发送signal通知用户或执行预设的回调函数

从IO线程到工作线程如何交换数据呢？借助一个单生产者多消费者队列。

处理完后返回数据如何发送？实现方式可以有很多，比如处理完成之后，直接在工作线程中向客户端发送数据。或者再创建一个写入队列，将返回数据和客户端信息（比如fd）放入该队列，然后由一组专门负责发送的线程来取元素和发送

半同步半反应堆IO：

Half-Sync/Half-Reactor（HSHR）：同步IO+队列+同步处理层



死循环之初，Polling API只监听服务端socket，当监测到服务端socket可读，进行accept，获得客户端fd，将其加入Polling API的监控集合中，继续进入循环。

循环体内Polling API返回，将fd取出。如果是服务socket再重复accept，如果是其他fd就丢进队列中。

和HSHA不同，HSHR的队列中存放的不是请求数据，而是fd。

多个工作线程竞争消费队列，从队列中取到的不是数据，而是客户端fd以及一些状态信息。工作线程的逻辑循环内从队列取到fd后，对fd进行read/recv获取请求数据，然后进行处理，最后直接write/send客户端fd，将数据返回给客户端。

**模板：**

模板类的模板构造函数：

template<class T>

class A{

public:

template<class T>

A(T) {cout << typeid(T).name() << endl;}

void show() {cout << typeid(T).name() << endl;}

};

int main(){

A<double> a(3); //int

a.show(); //double

}

There is no way to explicitly specify the template arguments when calling a constructor template, so they have to be deduced through argument deduction. This is because if you say:

Foo<int> f = Foo<int>();

The <int> is the template argument list for the type Foo, not for its constructor.

模板特化实现模板参数变换：

//默认特化不实现

template <typename T >

class Magic;

template<class Ret, class Arg>

struct Magic<Ret(Arg)>

{

void show(){ cout << typeid(Ret).name() << " " << typeid(Arg).name() << endl;}

};

int main(){

Magic<double(int)> m; //double int

m.show();

}

多态实现类型擦除：

用Function类包装所有返回值为void参数为int的可调用对象，如下：

class Function

{

private:

struct CallableBase{

virtual void operator()(int a0) = 0;

virtual struct CallableBase \*copy() const = 0;

virtual ~CallableBase() {};

};

template <typename T>

struct CallableDerived : public CallableBase{

T f;

CallableDerived(T functor) : f(functor) {}

void operator()(int arg) { return f(arg); }

CallableBase \*copy() const { return new CallableDerived<T>(f); }

};

private:

CallableBase \*m\_ptr;

public:

template <typename T>

Function(T functor) : m\_ptr(new CallableDerived<T>(functor)) {}

Function() : m\_ptr(nullptr) {}

void operator()(int arg) { return (\*m\_ptr)(arg); }

Function(const Function &f) { m\_ptr = f.m\_ptr->copy(); }

Function &operator=(const Function &f) {

if (this != &f){

delete m\_ptr;

m\_ptr = f.m\_ptr->copy();

}

return \*this;

}

Function(Function &&f) {

m\_ptr = f.m\_ptr;

f.m\_ptr = nullptr;

}

Function &operator=(Function &&f) {

if (this != &f){

delete m\_ptr;

m\_ptr = f.m\_ptr;

f.m\_ptr = nullptr;

}

return \*this;

}

~Function() { delete m\_ptr; }

};

int main(int argc, char \*\*argv)

{

Function showFunc = [](int a) {cout << "par=" << a << endl; };

showFunc(9);

}

**Lambda优于std::bind**

lambda更易内联

lambda延迟求值

using namespace std::chrono;

using Time = steady\_clock::time\_point;

enum class Sound { Beep, Siren, Whistle };

//在时间t，使用s声音响铃

void setAlarm(Time t, Sound s){

cout << "start time=" << duration\_cast<seconds>(t.time\_since\_epoch()).count() << " alram" << endl;

}

int main(int argc, char \*\*argv)

{

//设置一个小时后响铃的警报器， 但是，具体声音仍未确定

auto setSoundL =[](Sound s){setAlarm(steady\_clock::now() + hours(1), s);};

//警报器将被设置为在调用setSoundL后一小时发出声音

setSoundL(Sound::Beep); //setAlarm函数体在这可以很好地内联

//在std::bind调用中，将steady\_clock::now() + 1h作为实参传递给了std::bind，而不是setAlarm。这意味着将在调用std::bind时对表达式进行求值，并且该表达式产生的时间将存储在产生的bind对象中

//警报器将被设置为在调用std::bind后一小时发出声音，而不是在调用setAlarm一小时后发出

auto setSoundB = std::bind(&setAlarm, steady\_clock::now() + hours(1), std::placeholders::\_1);

setSoundB(Sound::Beep); //对std::bind的调用是将函数指针传递给setAlarm，对setAlarm的调用是通过一个函数指针，编译器不太可能通过函数指针内联函数

}

按值传递、按引用传递

struct Widget{

int size = 8;

Widget() = default;

Widget(const Widget& rhs) :size(rhs.size) { cout << "Widget Copy Constructor" << endl; }

};

struct CompLevel{

int lev = 0;

CompLevel() = default;

CompLevel(const CompLevel& rhs) :lev(rhs.lev) { cout << "CompLevel Copy Constructor" << endl; }

CompLevel(CompLevel&& rhs) :lev(rhs.lev) { cout << "CompLevel Move Constructor" << endl; }

};

void compress(const Widget& w, CompLevel lev) { //制作w的压缩副本

cout << "compressed , size=" << w.size << endl;

}

int main(int argc, char \*\*argv)

{

Widget w; //w.heavy=8

auto compressRateL = [&w](CompLevel lev) {return compress(w, lev); }; //显示指明按引用传递

auto compressRateB = std::bind(compress, w, std::placeholders::\_1); //默认按值传递，CompLevel拷贝构造

auto compressRateBRef = std::bind(compress, std::ref(w), std::placeholders::\_1); //引用传递

w.size = 9;

CompLevel lev;

compressRateL(lev); //实参lev按值传递，总共两次CompLevel的拷贝构造，size=9

compressRateB(lev); //实参lev默认按引用传递，一次CompLevel的拷贝构造，size=8

}

**emplace插入：**

拷贝初始化和直接初始化：

struct A {

explicit A(int x) { }

};

int main(){

A r1 = 10; //错误！不能编译

A r2(10); //正确

}

在标准术语中，用于初始化r1的语法（使用等号“=”）称为**拷贝初始化**；

用于初始化r2的语法（使用小括号或花括号）称为**直接初始化**

拷贝初始化不允许使用explicit构造函数

对r1，int类型10首先隐式转换为A，而根据构造函数定义中的explicit，这样的隐式转换不被允许，从而产生编译时期的报错。如果无explicit，则仅会调用构造函数，而非构造函数+拷贝\移动赋值函数

emplace插入和explicit：

struct A {

explicit A(int x) { }

};

int main(){

std::vector<A> vec;

vec.push\_back(10); //错误

vec.emplace\_back(11); //正确

}

对push\_back(10)，实际是拷贝初始化

emplace\_back实际是直接初始化，不受explicit影响，相当于A a(10)

emplace插入和存储资源管理类对象的容器：

对存储资源管理类对象的容器（比如std::list<std::shared\_ptr<Widget>>）调用插入函数时，函数的形参类型通常确保在资源的获取（比如使用new）和资源管理对象的创建之间没有其他操作。在emplace函数中，完美转发推迟了资源管理对象的创建，直到可以在容器的内存中构造它们为止，这给“异常导致资源泄漏”提供了可能。

class Widget {

};

int main(){

void killWidget(Widget\* pWidget);

std::list<std::shared\_ptr<Widget>> ptrs;

ptrs.push\_back(std::shared\_ptr<Widget>(new Widget, killWidget));

ptrs.emplace\_back(new Widget, killWidget);

}

对push\_back(std::shared\_ptr<Widget>(new Widget, killWidget))

1. 在上述的调用中，一个std::shared\_ptr<Widget>的临时对象被创建来持有“new Widget”返回的原始指针。称这个对象为temp。
2. push\_back通过引用接受temp。在存储temp的副本的*list*节点的内存分配过程中，内存溢出异常被抛出。
3. 随着异常从push\_back的传播，temp被销毁。作为唯一管理这个Widget的std::shared\_ptr，它自动销毁Widget，在这里就是调用killWidget。

对ptrs.emplace\_back(new Widget, killWidget)：

1. 通过new Widget创建的原始指针完美转发给emplace\_back中，*list*节点被分配的位置。如果分配失败，还是抛出内存溢出异常。
2. 当异常从emplace\_back传播，原始指针是仅有的访问堆上Widget的途径，但是因为异常而丢失了，那个Widget的资源（以及任何它所拥有的资源）发生了泄漏。

**=delete：**

C++中只声明一个函数，不定义，不调用，是可以编译链接并运行的

C++11前实现禁止拷贝语义，常

class basic\_ios {

private:

basic\_ios(const basic\_ios& ); // not defined

basic\_ios& operator=(const basic\_ios&); // not defined

};

将它们声明为private可以防止客户端调用这些函数。故意不定义它们意味着假如还是有代码用它们（比如成员函数或者类的友元friend），就会在链接时报错缺少函数定义（missing function definitions）。

C++11后，

class basic\_ios {

public:

basic\_ios(const basic\_ios& )=delete;

basic\_ios& operator=(const basic\_ios&)=delete;

};

deleted函数不能以任何方式被调用，否则会在编译期报错，而使用私有未定义函数的方式，可能在链接时才报错。

deleted函数一般被声明为public而不是private。当客户端代码试图调用成员函数，C++会在检查deleted状态前检查它的访问性。当客户端代码调用一个私有的deleted函数，一些编译器只会给出该函数是private的错误（而没有诸如该函数被deleted修饰的错误），即使函数的访问性不影响它是否能被使用。

**{}初始化：**

Widget w1(10); //使用实参10调用Widget的一个构造函数

但是如果你尝试使用相似的语法调用没有参数的Widget构造函数，它就会变成函数声明：

Widget w2(); //最令人头疼的解析！声明一个函数w2，返回Widget

由于函数声明中形参列表不能使用花括号，所以使用花括号初始化没有问题：

Widget w3{}; //调用无参构造函数构造对象

C++规定任何能被决议为一个声明的东西必须被决议为声明

当auto声明的变量使用花括号初始化，变量类型就会被推导为std::initializer\_list，尽管使用相同内容的其他初始化方式会产生正常的结果。如auto y = { 10 };//y被推导为std::initializer\_list<int>类型