

深入理解并发/行技术



By 公众号@极客重生



内容

并发/并行技术核心是什么

并发/并行问题技术大局观

深入理解处理器并行技术

深入理解操作系统并行技术

深入理解编程语言并行技术

并发/并行技术核心是什么

- **核心1：弄清问题**
我们要解决什么问题？
- **核心2：理解重要概念**
一些重要的概念深刻理解
- **核心4：场景分析**
不同场景不同实现技术

并发/并行技术核心是什么

核心1：弄清问题，我们要解决什么问题？

- 如何节省大型和复杂问题的解决时间，即如何提高性能—提高程序**并发/并行度**
- 并发/并行提升性能后带来的问题如何解决
 - 缓存优化导致的可见性问题—内存模型，缓存一致性，内存屏障
 - 程序原子性问题—硬件锁技术
 - 编译优化带来的有序性问题--内存模型，内存屏障
 - 资源冲突问题-锁技术

并发技术的本质：

- 在多核编程中代码的执行顺序和编程顺序不一样，可能会导致不符合预期结果。
- 共享资源的争夺，最小粒度分时复用。


并发/并行技术核心是什么

核心2：理解重要的概念

- 并发和并行
- 进程，线程，协程
- 内存可见性，指令重排，内存屏障，内存模型，
- 临界区，空转，阻塞，唤醒
- 同步，异步

并发/并行技术核心是什么

核心2：理解重要的概念—内存可见性

内存可见性  延伸
 多个变量
同一CPU执行的
内存操作循序

什么叫内存可见性：

就是让其他CPU看到本CPU代码写内存的顺序

代码写好后，程序员认为代码执行循序是固定的

但其他CPU看到顺序可能不一样。

```
1 Litmus Test: Message Passing
2 Can this program see r1 = 1, r2 = 0?
3
4 // Thread 1           // Thread 2
5 x = 1                 r1 = y
6 y = 1                 r2 = x
```

- 读内存，也就是加载操作(load)，从内存读到寄存器
- 写内存，就就是存储操作(store)，从寄存器写入到内存

- load指令后接load指令 (L-L, 读读)
- store指令后接store指令 (S-S, 写写)
- load指令后接store指令 (L-S, 读写)
- store指令后接load指令 (S-L, 写读)

可见性即当一个变量修改后，这个变量会马上更新到主存中，其他线程会收到通知这个变量修改过了，使用这个变量的时候重新去主存获取。

实际上，CPU和编译可能会在程序员不知情的情况下(为了提高性能)修改x和y赋值顺序(就是写内存顺序)：

```
1 y=1;
2 x=1;
```

这样可能就会导致不符合程序员预期行为。

比如左边例子：r1=1, r2=0 可能产生，但这个结果不在程序员预期之内，所以需要明确的内存模型来限制或者避免这种行为。

程序员认为代码执行循序是固定的(代码就是这样写的)：

```
1 x=1;
2 Y=1;
```

然后程序员推断：CPU2理论上可以看到6种循序

$x = 1$ $y = 1$ $r1 = y (1)$ $r2 = x (1)$	$x = 1$ $y = 1$ $r1 = y (0)$ $r2 = x (1)$	$x = 1$ $y = 1$ $r1 = y (0)$ $r2 = x (1)$
$x = 1$ $y = 1$ $r1 = y (0)$ $r2 = x (1)$	$x = 1$ $y = 1$ $r1 = y (0)$ $r2 = x (1)$	$x = 1$ $y = 1$ $r1 = y (0)$ $r2 = x (0)$

并发/并行技术核心是什么

核心2：理解重要的概念—指令重排

在执行程序时为了提高性能,编译器和处理器常常会对指令做重排序。重排序分三种类型:

- **编译器优化的重排序。**编译器在不改变单线程程序语义的前提下,可以重新安排语句的执行顺序。
- **指令级并行的重排序。**现代处理器采用了指令级并行技术(Instruction-Level Parallelism, ILP)来将多条指令重叠执行。如果不存在数据依赖性,处理器 可以改变语句对应机器指令的执行顺序。
- **内存系统的重排序。**由于处理器使用缓存和读/写缓冲区,这使得加载和存储操作看上去可能是在乱序执行。

并发/并行技术核心是什么

核心2：理解重要的概念—内存屏障

内存屏障（英语：Memory barrier），也称**内存栅栏**，**内存栅障**，**屏障指令**等，是一类同步屏障指令，它使得 CPU 或编译器在对内存进行操作的时候，严格按照一定的顺序来执行，也就是说在内存屏障之前的指令和之后的指令不会由于系统优化等原因而导致乱序。

大多数现代计算机为了提高性能而采取乱序执行，这使得内存屏障成为必须。

语义上，内存屏障之前的所有写操作都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。因此，对于敏感的程序块，写操作之后、读操作之前可以插入内存屏障。



并发/并行技术核心是什么

核心2：理解重要的概念—内存模型

什么是Memory Model

Memory Model(Memory Consistency/Memory Consistency Model)是系统和程序员之间的规范，它规定了在一个共享存储器的多线程程序中的存储器访问应该表现出怎样的行为。这个规范影响了系统的性能，因为它决定了多处理器/编译器能应用哪些优化；也影响了可编程性(Programmability)，因为多线程程序的正确性取决于Memory Model，从而约束了程序员的编程方式。

内存模型是一种解决多线程场景下的一个主存操作规范，多线程场景下访问主存保证原子性、可见性、有序性，对于并发程序的正确性非常关键

为什么需要Memory Model

- 在单线程程序中，编译器的各种优化如冗余代码消除、循环融合(Loop Fusion)、指令调度等技术，会重写代码造成存储器访问顺序变化，而寄存器分配会改变存储器访问次数；类似的，处理器的乱序执行机制也会通过让后一条指令先执行，来掩盖前一条指令导致的流水线停顿和存储器停顿。尽管会改变不同地址存储器的访问顺序，但系统的这些优化对程序员是透明的，看起来程序仍然是在顺序执行。
- 然而在多线程程序中，编译器和多处理器并无手段自动发现多个线程间的协作关系，使得那些可能改变存储器访问顺序和次数的优化，同时对多个线程透明。没有程序员的帮助，要保持多线程程序的正确性，系统只能禁用这些作用在共享存储器上的优化，而这将严重损害性能。
- 为最大限度保留编译器和多处理器的优化能力，同时使多线程程序的执行结果是可预测的，系统需要程序员的帮助。最后的方案是，系统提供所谓Memory Model的规范，程序员通过规范中同步设施(各种内存屏障(Memory Barrier)和Atomic指令)来标记多个线程间的协作关系，使得不仅是单线程，系统的优化对多线程程序也将透明。

并发/并行技术核心是什么

核心2：理解重要的概念—内存模型

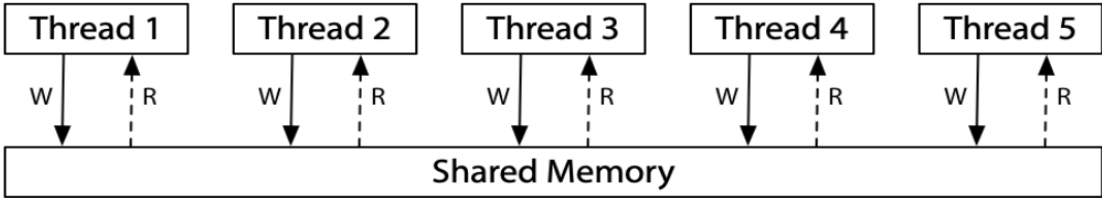
- 顺序一致性，理想模型
- 非顺序一致性，x86，ARM等，放弃严格的顺序一致性可以让硬件更快地执行程序，所以所有现代硬件在各方面都会偏离了顺序一致性，松散的内存模型可能会非常混乱，写操作可能会无序，读操作可能会返回不是我们想要的值，为了解决这些问题，我们需要使用内存栅栏（memory fences,内存屏障）

```
1 Litmus Test: Message Passing
2 Can this program see r1 = 1, r2 = 0?
3
4 // Thread 1           // Thread 2
5 x = 1                 r1 = y
6 y = 1                 r2 = x
7 On sequentially consistent hardware: no.
8 On x86 (or other TSO): no.
```

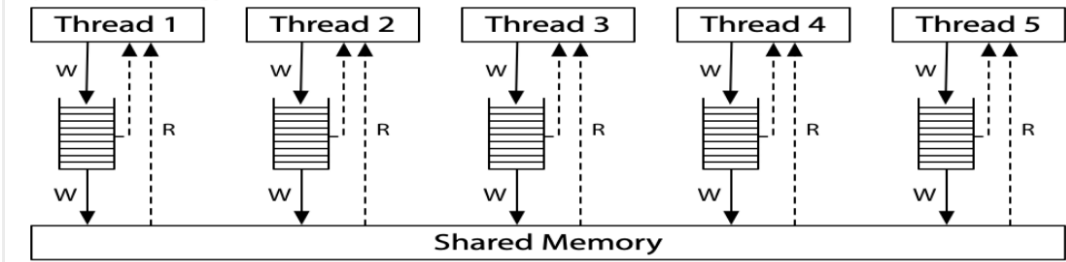
```
1 Litmus Test: Message Passing
2 Can this program see r1 = 1, r2 = 0?
3
4 // Thread 1           // Thread 2
5 x = 1                 r1 = y
6 y = 1                 r2 = x
7 On sequentially consistent hardware: no.
8 On x86 (or other TSO): no.
9 On ARM/POWER: yes!
```

```
1 // Thread 1           // Thread 2
2 x = 1;                while(done == 0) { /* loop */ }
3 done = 1;             print(x);
```

如果线程1和线程2都运行在自己专用处理器上，都运行到完成，这个程序能打印 0 吗？

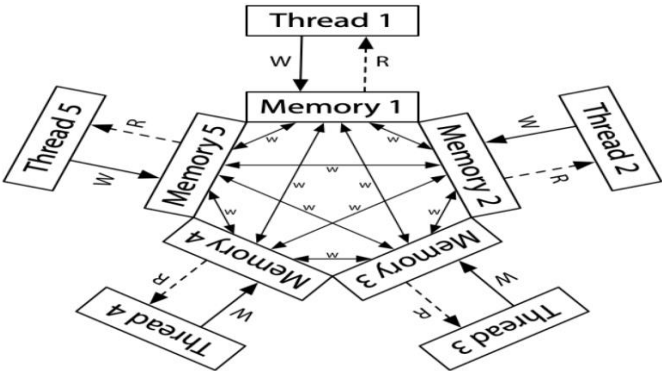


顺序一致性内存模型



x86 内存模型-Total Store Order (强内存模型)

内存操作顺序方式（两个）：
读读, 读写, 写写, **写读**, TSO维持了前三种，但不维持**写读**的顺序（因为有写队列存在，读不一定读到最新的值，要保证**写读**之间非乱序，需要在写和读之间加一个内存屏障mfence指令后面会讲）

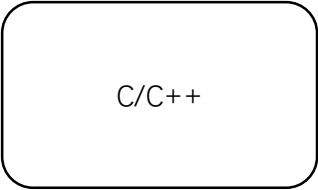


ARM内存模型-弱内存模型，

内存操作顺序方式：
读读, 读写, 写写, 写读, ARM不保证顺序（可以乱序）,属于relaxed model或者弱内存模型（weak order），要保证**上面操作**之间不乱序，需要在操作之间加一个内存屏障mfence指令。

并发/并行技术核心是什么

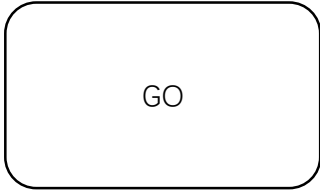
核心2：理解重要的概念—内存模型



- sequential consistent(排序一致序列, 强同步, Happen before)
- acquire release(获取-释放序列, 弱同步)
- relaxed(松散序列, 无同步)

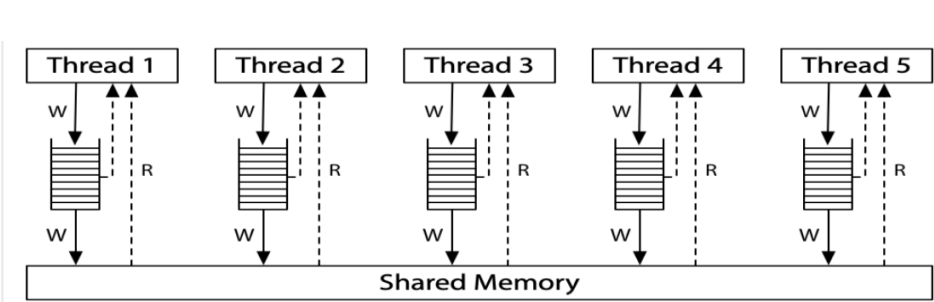


- Happen before规则
- JMM统一内存模型



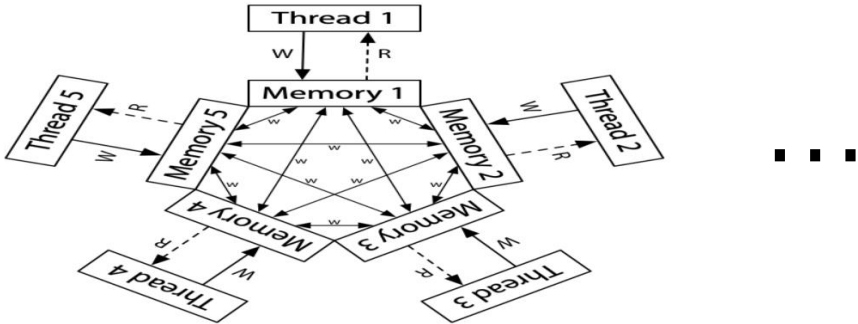
- Happen before规则
- 更高层, 更简单

编程语言内存模型
用于屏蔽各种硬件和操作系统的内存访问差异, 以实现让程序在各种平台都能达到一致的内存访问效果。



x86

内存模型-Total Store Order (强内存模型)

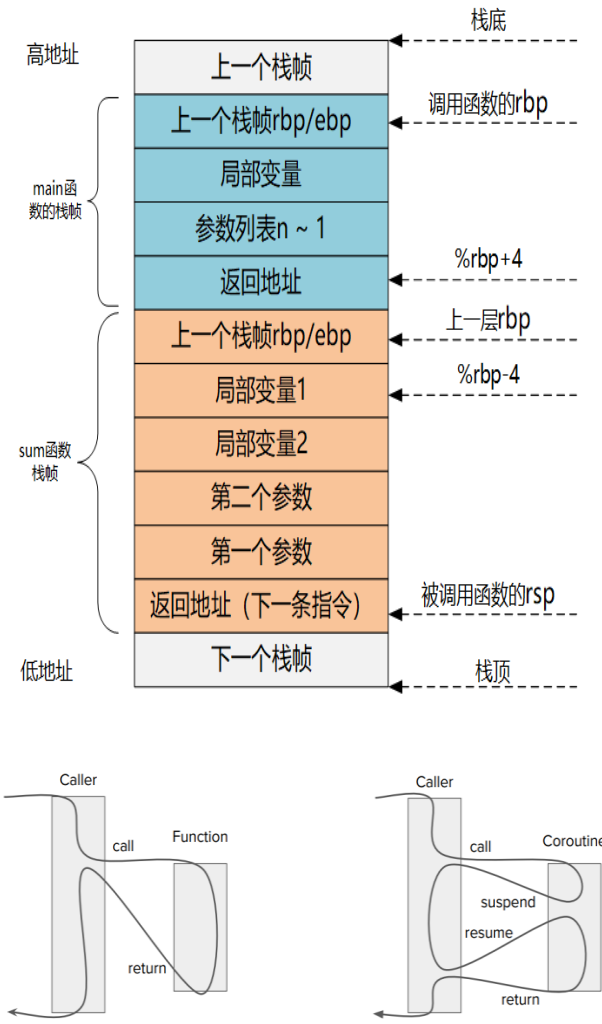
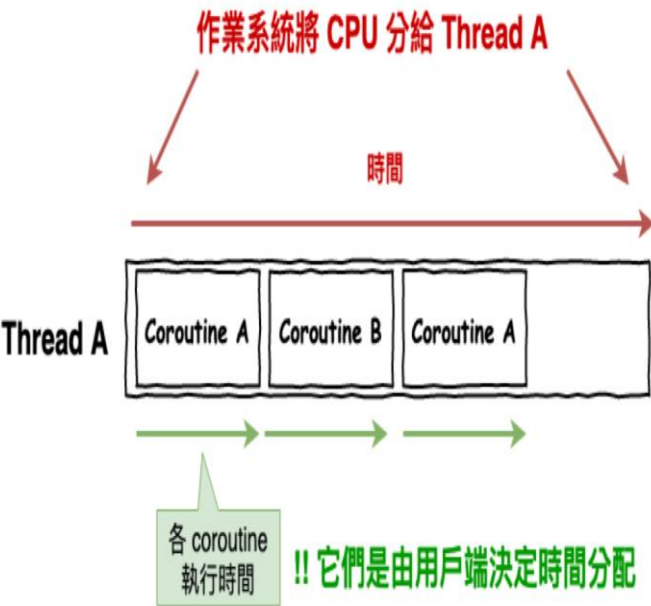


ARM

内存模型-弱内存模型

并发/并行技术核心是什么

核心2：理解重要的概念—协程



```
1 #include <stdio.h>
2
3 int sum (int a,int b)
4 {
5     int c = a + b;
6     return c;
7 }
8
9 int main()
10 {
11     int x = 5,y = 10,z = 0;
12     z = sum(x,y);
13     printf("%d\\r\\n",z);
14     return 0;
15 }
```

```
1 0000000000000000 <sum>:
2 0: 55 push %rbp
3 1: 48 89 e5 mov %rsp,%rbp
4 4: 89 7d ec mov %edi,-0x14(%rbp) # 参数传递
5 7: 89 75 e8 mov %esi,-0x18(%rbp) # 参数传递
6 a: 8b 55 ec mov -0x14(%rbp),%edx
7 d: 8b 45 e8 mov -0x18(%rbp),%eax
8 10: 01 d0 add %edx,%eax
9 12: 89 45 fc mov %eax,-0x4(%rbp) # 局部变量
10 15: 8b 45 fc mov -0x4(%rbp),%eax # 存储结果
11 18: 5d pop %rbp
12 19: c3 retq
13
14 000000000000001a <main>:
15 1a: 55 push %rbp # 保存%rbp. rbp, 栈底的地址
16 1b: 48 89 e5 mov %rsp,%rbp # 设置新的栈指针. rsp 栈指针, 指向栈顶
17 1e: 48 83 ec 10 sub $0x10,%rsp # 分配 16字节栈空间. %rsp = %rsp-16
18 22: c7 45 f4 05 00 00 00 movl $0x5,-0xc(%rbp) # 赋值
19 29: c7 45 f8 0a 00 00 00 movl $0xa,-0x8(%rbp) # 赋值
20 30: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp) # 赋值
21 37: 8b 55 f8 mov -0x8(%rbp),%edx
22 3a: 8b 45 f4 mov -0xc(%rbp),%eax
23 3d: 89 d6 mov %edx,%esi # 参数传递, 从右向左
24 3f: 89 c7 mov %eax,%edi # 参数传递
25 41: e8 00 00 00 00 callq 46 <main+0x2c> # 调用sum
26 46: 89 45 fc mov %eax,-0x4(%rbp)
27 49: 8b 45 fc mov -0x4(%rbp),%eax # 存储计算结果
28 4c: 89 c6 mov %eax,%esi
29 4e: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 55 <main+0x3b>
30 55: b8 00 00 00 00 mov $0x0,%eax
31 5a: e8 00 00 00 00 callq 5f <main+0x45>
32 5f: b8 00 00 00 00 mov $0x0,%eax
33 64: c9 leaveq %eax
34 65: c3 retq
35
```

实现一个协程的关键点在于如何保存、恢复和切换上下文

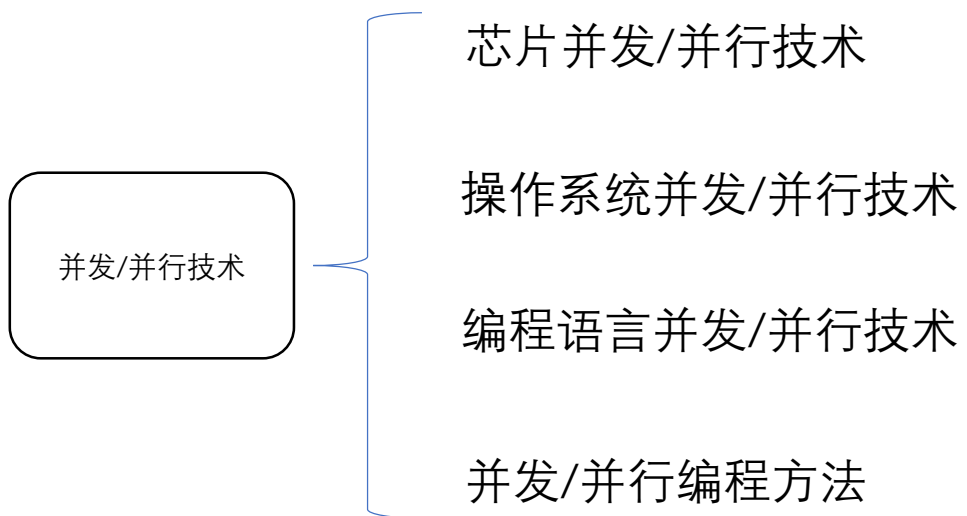
Coroutine就是函数，只不过是可以通过suspend和resume的函数

并发/并行技术核心是什么

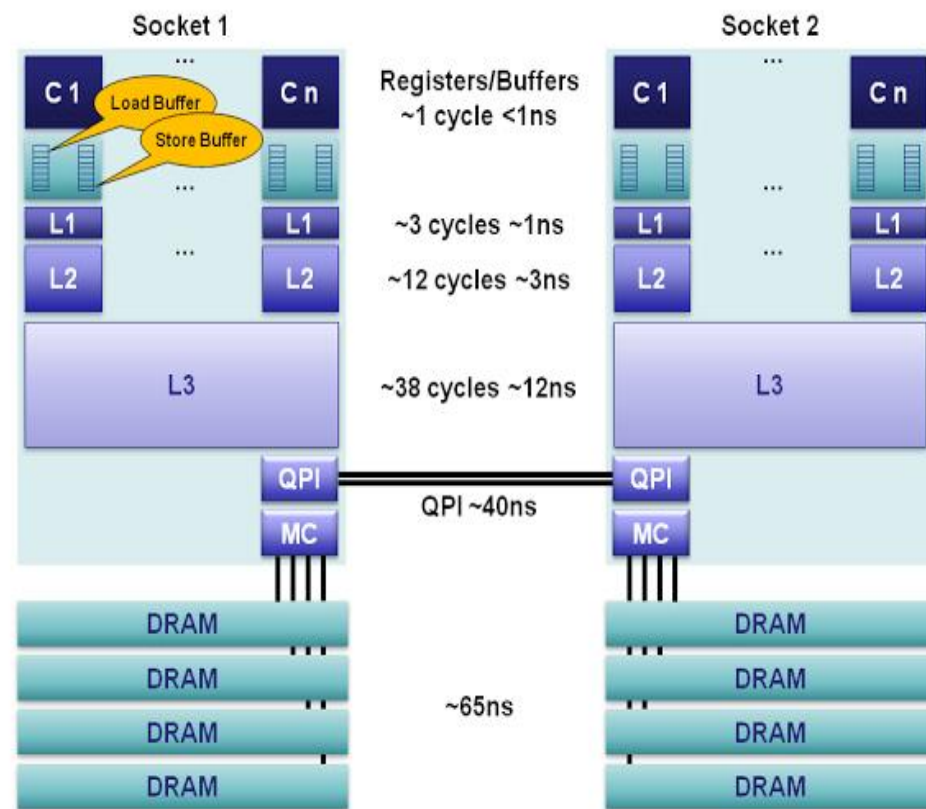
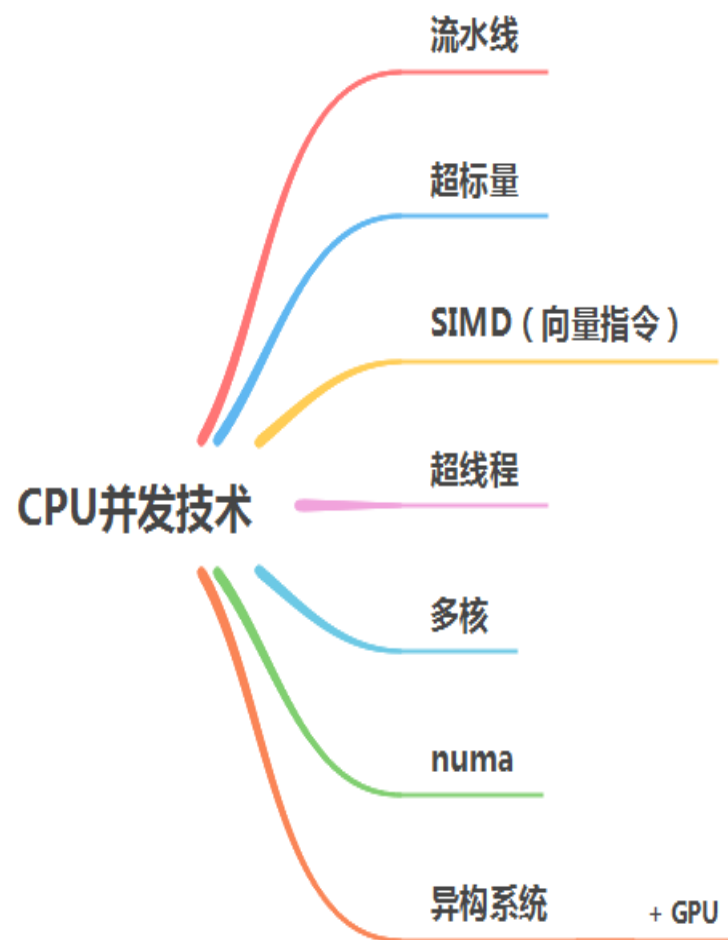
核心4：场景分析（不同场景不同实现技术）

并行计算可分为时间上的并行和空间上的并行。

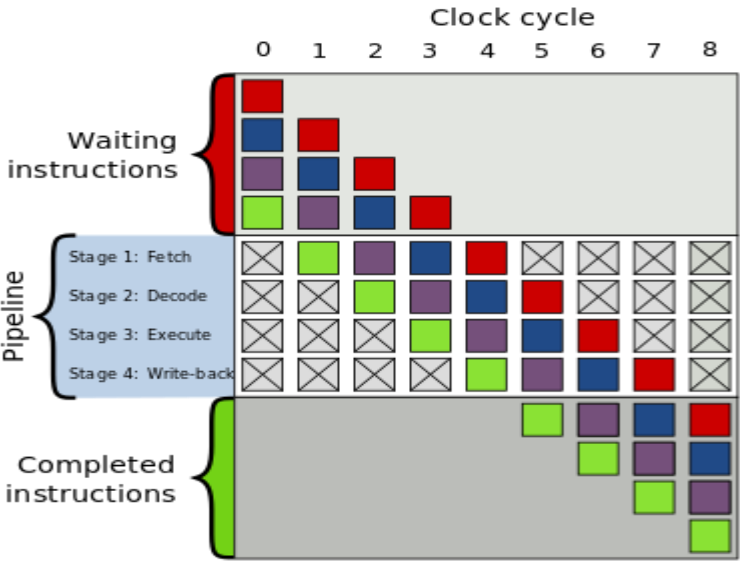
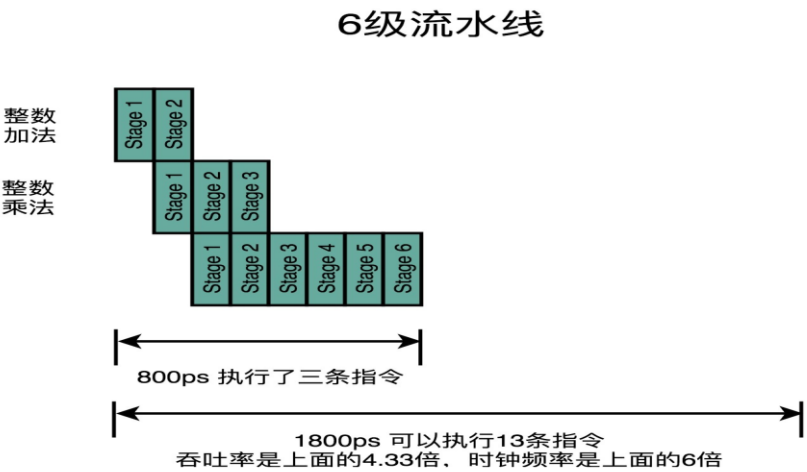
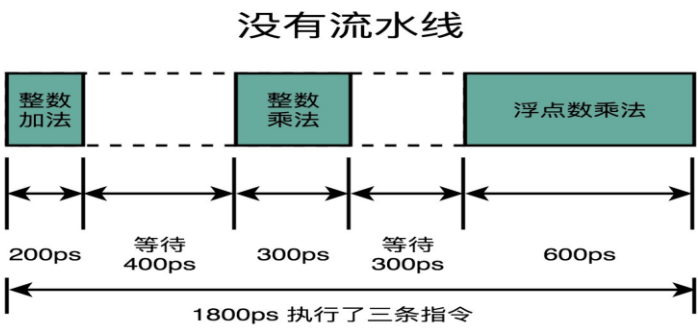
- 时间上的并行是指流水线技术
- 空间上的并行是指多个处理器（或者计算单元）并发的执行计算。



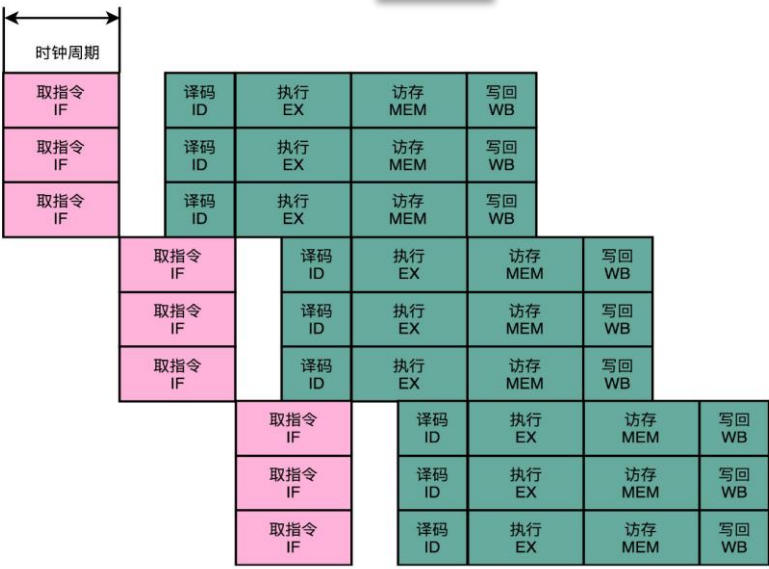
深入理解处理器并行技术



深入理解处理器并行技术--流水线/超标量

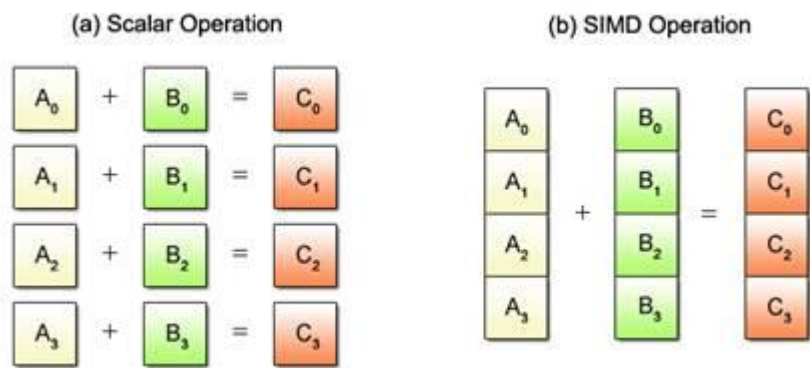


流水线



超标量+流水线

深入理解处理器并行技术-SIMD(向量指令)

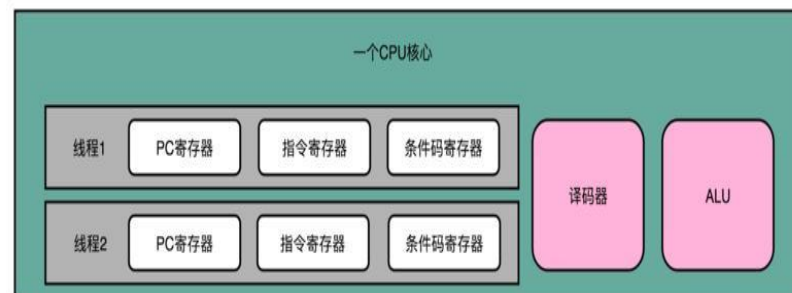
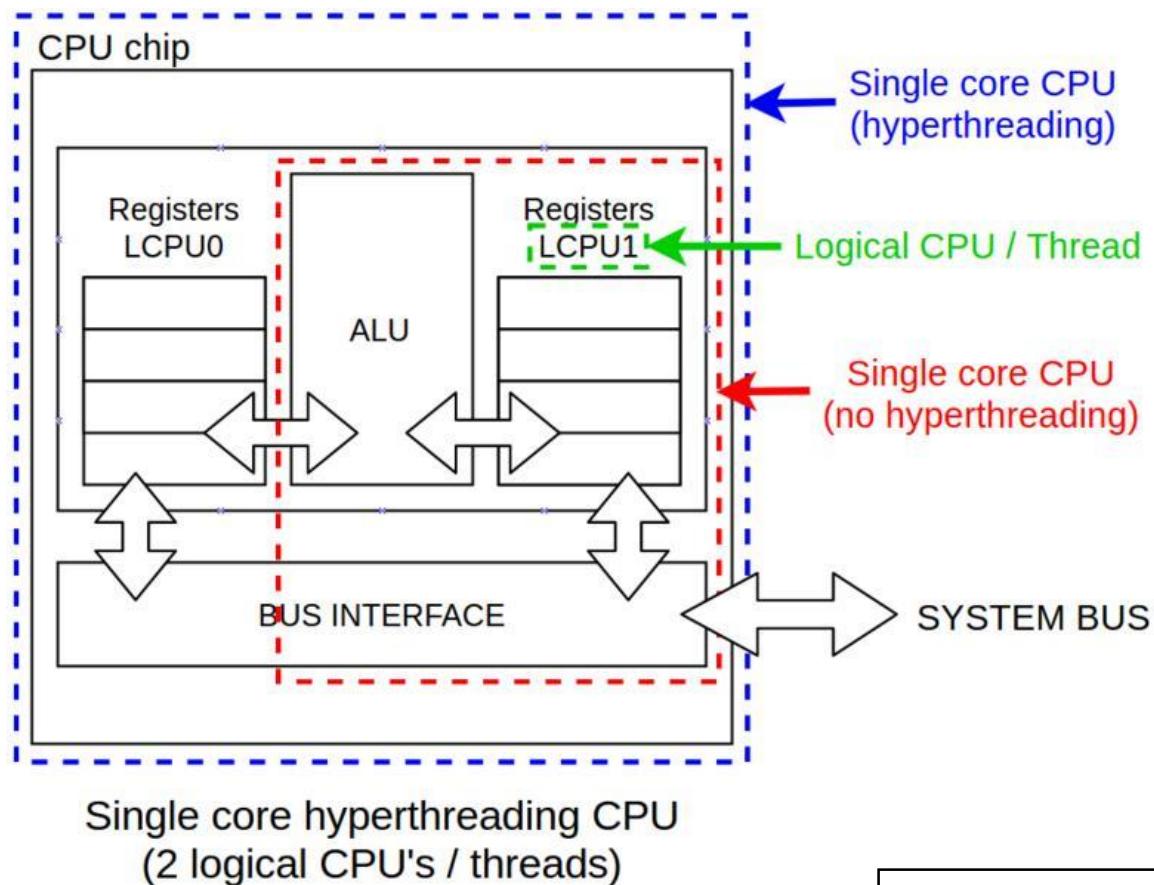


Scalar vs. SIMD Operations

- MMX
- SSE/SSE2/SSE3/SSE4
- AVX

MMX	64位整型
SSE	128位浮点运算，整数运算仍然要使用MMX 寄存器，只支持单精度浮点运算
SSE2	对整型数据的支持，支持双精度浮点数运算，CPU快取的控制指令
SSE3	扩展的指令包含寄存器的局部位之间的运算，例如高位和低位之间的加减运算；浮点数到整数的转换，以及对超线程技术的支持。
SSE4	
AVX	256位浮点运算
AVX2	对256位整型数据的支持，三运算指令（ 3-Operand Instructions ）
AVX512	512位运算

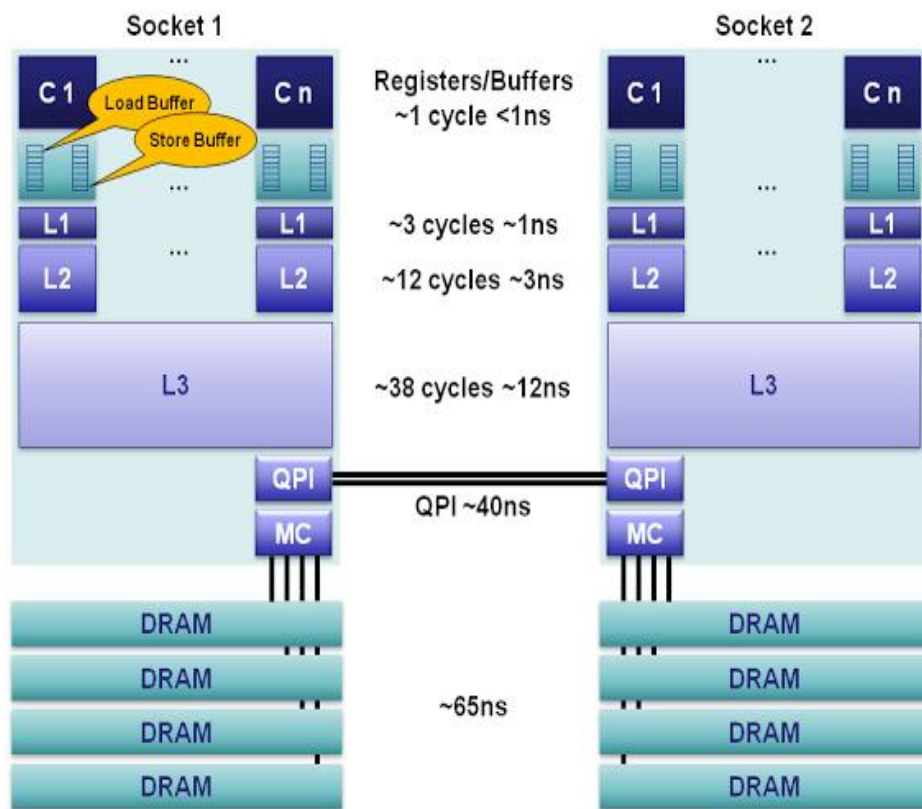
深入理解处理器并行技术—超线程



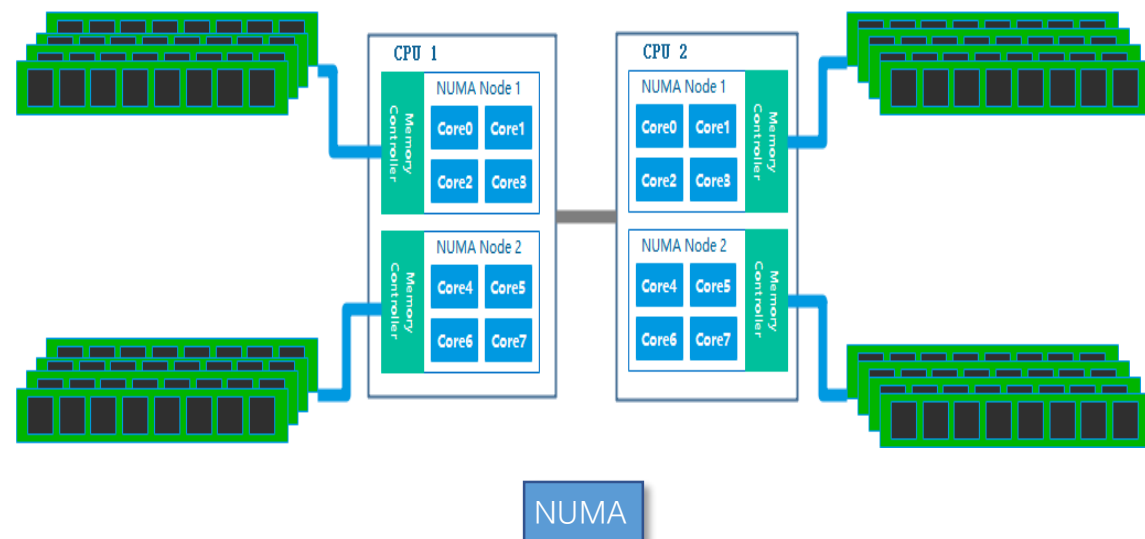
硬件线程一般是指逻辑核（逻辑CPU），逻辑核就是超线程抽象出来的，有独立寄存器，但ALU和一些译码器是共享的，不是真正意义单独CPU，但操作系统看到CPU就是逻辑CPU，操作系统将软件线程的任务分发在多个硬件线程上，通过负载均衡，可以分配在各个硬件线程。

什么情况下需要关闭超线程？

深入理解处理器并行技术-多核和numa



多核

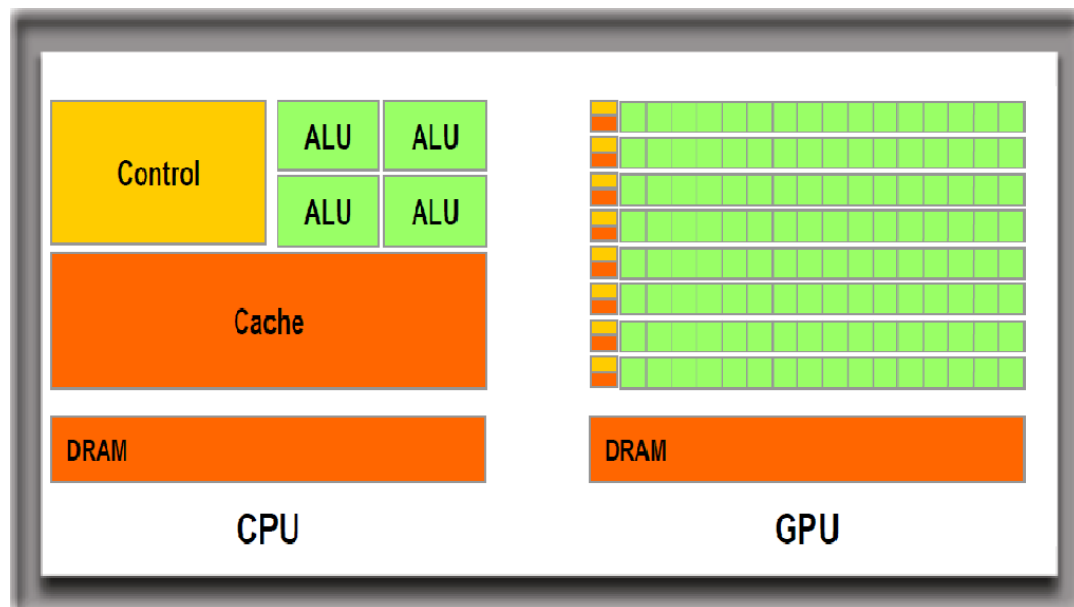


- 多核 AMD pk intel
- NUMA 扩展

多核编程会带来哪些挑战？

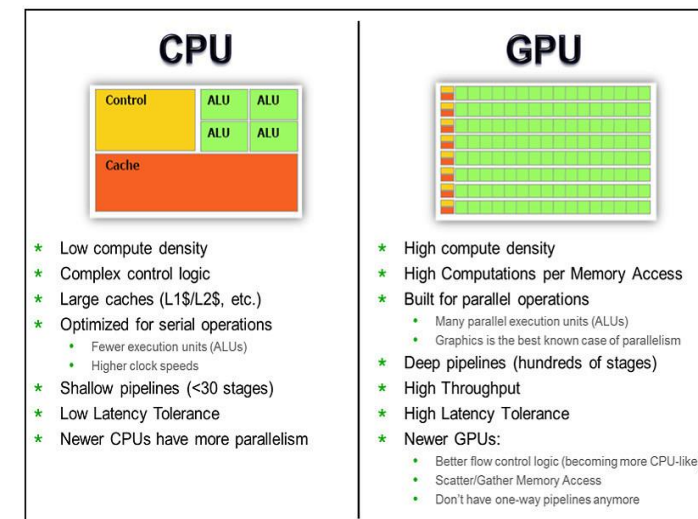
NUMA 编程需要注意什么？

深入理解处理器并行技术—异构系统之GPU



- CPU—通用性
- GPU—专业并行计算

CPU + GPU



深入理解操作系统(Linux)并行技术

Linux并发技术

计算（调度）：多线程，多进程，CPU抢占，CPU亲和（绑定），中断亲和，CPU独占隔离，PerCPU

网络：中断亲和，多队列网卡（RSS）、RPS、RFS、XPS、SO_REUSEPORT

IO：DMA，零拷贝，**COW**，bypass kernel，异步IO，并行IO，并行文件系统

并发问题：阻塞锁（mutex，信号量，rwlock）
原子技术（ACCESS_ONCE()、READ_ONCE()
and WRITE_ONCE(), barrier(), atomic，内存屏障），
非阻塞（无锁）技术（spinlock，seqlock，RCU）

深入理解操作系统并行技术—计算（调度）

多线程，多进程，CPU抢占，CPU亲和（绑定），中断亲和，CPU独占隔离，PerCPU

后续课程更新

深入理解操作系统并行技术—网络

中断亲和，多队列网卡（RSS）、RPS、RFS、XPS、SO_REUSEPORT

后续课程更新

深入理解操作系统并行技术—IO

DMA，零拷贝，**COW**，bypass kernel，异步IO，并行IO（MPI I/O，HDFS I/O），并行文件系统

后续课程更新

深入理解操作系统(Linux)并行技术—并发问题

锁

Mutex：互斥等待。

信号量：多对1等待。

Rwlock：读多写少，读优先，写等待。

无锁技术

自旋锁（spinlock）：临界区短，无堵塞。

Seqlock：读多写少，写优先，读重试。

RCU：读多写少，读不影响写，复制更新。

原子技术

ACCESS_ONCE()/READ_ONCE()/WRITE_ONCE()：禁止编译器对数据访问的优化，强制从内存而不是缓存中获取数据；

barrier()：乱序访问内存屏障，限制编译器的乱序优化；

atomic：硬件级加锁（粒度很小）

atomic_inc()/atomic_read()等：整型原子操作；

CAS:原子方式对内存执行读-改-写操作，无锁技术的基础；

内存屏障：

smb_wmb()：写内存屏障，刷新store buffer，同时限制编译器和CPU的乱序优化；

smb_rmb()：读内存屏障，刷新invalidate queue，同时限制编译器和CPU的乱序优化；

smb_mb()：读写内存屏障，同时刷新store buffer和invalidate queue，同时限制编译器和CPU的乱序优化；

深入理解后端编程语言并行技术

C/C++

多线程（pthread, std::thread），异步（std::async, std::future, std::packaged_task, std::promise），锁（std::mutex, std::lock_guard, std::call_once(), std::shared_mutex, std::condition_variable），原子操作（std::atomic, CAS），内存模型（std::memory_order_xxx），协程(std::coroutine)

JAVA

多线程（Thread），并发理论（JMM(内存模型), 重排序, happens-before规则), 并发关键字(Synchronized, volatile, final), concurrent模块(atomic（CAS），阻塞队列，lock体系（AQS），并发容器，线程池（Executor体系）等)

GO

基本并发原语（Mutex、RWMutex、Waitgroup、Cond、Pool、Context等），**原子操作**，Goroutine，Channel，内存模型，GPM调度模型

深入理解后端编程语言并行技术— C++

C++ 内存模型

- memory_order_relaxed,
- memory_order_consume,
- memory_order_acquire,
- memory_order_release,
- memory_order_acq_rel,
- memory_order_seq_cst



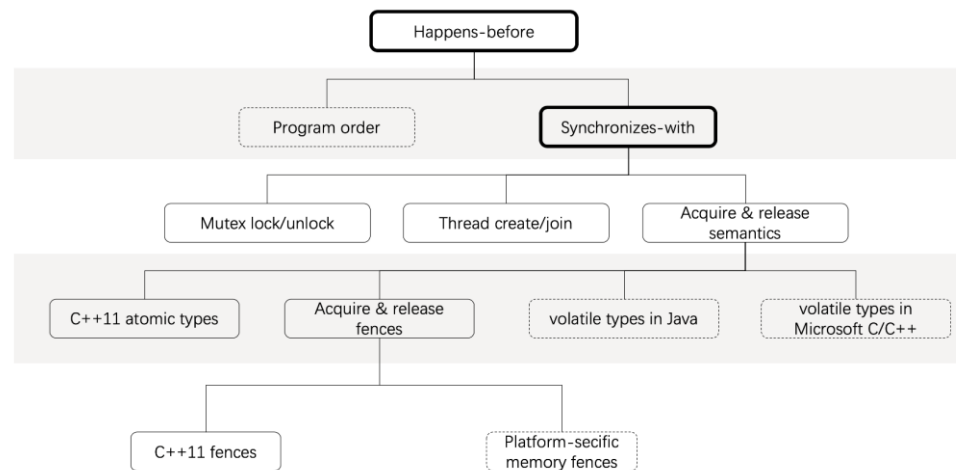
- sequential_consistent(排序一致序列, 强同步, memory_order_seq_cst, happens before);
- relaxed(松散序列, 无同步, memory_order_relaxed);
- acquire-release(获取-释放序列, 弱同步, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel);

C++11最终代表3种内存模型

C++11 提供六个内存序列选项可应用于对原子类型的操作

C++ 为什么要提供多种内存模型？

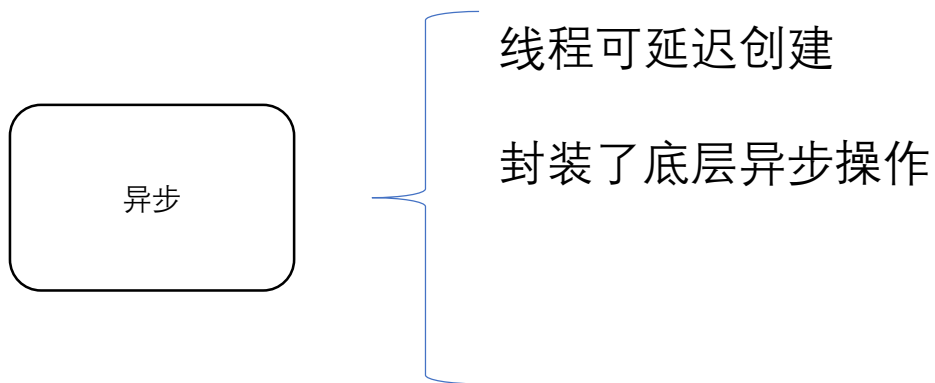
Acquire语义：修饰读，读后面所有指令不能在读之前执行（乱序），等于=读内存屏障
Release语义：修饰写，写前面所有指令不能在写之后执行（乱序），等于=写内存屏障
Relaxed语义：宽松的内存序，使用这个内存序的原子操作就真的仅仅是保证自身的原子性
Consume语义：修饰读，比Acquire稍弱，Acquire限制所有，Consume只限制和读有依赖的。
acq_rel语义：修改同时读写操作，相当于Acquire语义+ Release语义；
seq_cst语义：顺序一致性，相当于Acquire语义+ Release语义+ acq_rel语义；



深入理解后端编程语言并行技术— C++

C++ 异步，协程

解决异步问题带来了新的思路



- **std::future**
该类主要作为异步结果传输通道，方便获取线程函数的返回值；
- **std::packaged_task**
包装一个可调用对象，和future配合使用，方便异步调用
- **std::promise**
用来包装一个值，和future绑定使用，方便线程赋值
- **std::async**
可以直接创建异步的task类，异步返回结果保存在future中，在获取线程函数返回结果时，使用get()获取返回值，如果不需要值则使用wait()方法

```
#include <iostream>
#include <future>
#include <thread>

int main()
{
    // future from a packaged_task
    std::packaged_task<int> task([]{ return 7; }); // wrap the function
    std::future<int> f1 = task.get_future(); // get a future
    std::thread t(std::move(task)); // launch on a thread

    // future from an async()
    std::future<int> f2 = std::async(std::launch::async, []{ return 8; });

    // future from a promise
    std::promise<int> p;
    std::future<int> f3 = p.get_future();
    std::thread( [&p]{ p.set_value_at_thread_exit(9); }).detach();

    std::cout << "Waiting..." << std::flush;
    f1.wait();
    f2.wait();
    f3.wait();
    std::cout << "Done!\nResults are: "
              << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';

    t.join();
}
```

Output:

```
Waiting...Done!
Results are: 7 8 9
```

C++ 异步，协程

解决异步问题带来了新的思路



共享栈-协程帧(coroutine frame)

C++20的协程没有协程调度器，由编译器负责安插代码，进行协程帧的创建、销毁，同时负责调用栈的保存、恢复。

返回值由编译器传给promise对象，保存到promise对象的成员中。调用者通过coroutin_handle即可拿到promise对象，进而拿到返回值

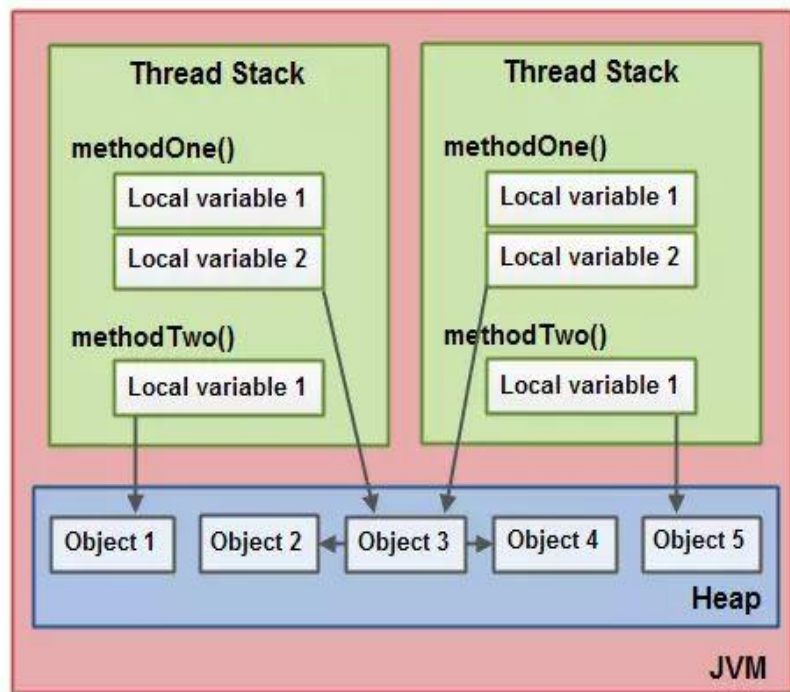
- `co_await some_awaitable`: 如果 `some_awaitable` 没有 ready, 就保存当前协程的执行状态并挂起。
- `co_yield some_value` : 保存当前协程的执行状态并挂起, 返回 `some_value` 给调用者。
- `co_return some_value` : 彻底结束当前协程, 返回 `some_value` 给协程调用者。
- `Promisetype` : `Promise` 类型定义了协程本身的行为, 比如协程被调用时发生什么, 返回时发生什么, 以及内部的 `co_await` 或 `co_yield` 行为, 每次协程调用都会在本协程专用内存池 (一个 heap 上分配好的本协程专用内存池) 中构造 `promise` 对象。
- `Awaiter type` : 暂停一个协程的求值, 等待表达式所表示的计算过程的结束, 支持 `co_await operator` 的类型即为 `Awaitable` 类型。需支持 `await_ready`, `await_suspend`, `await_resume` 接口。

```
55
56 future_type three_step_coroutine(){
57     std::cout<<"three_step_coroutine begin"<<std::endl;
58     co_await std::suspend_always();
59     std::cout<<"three_step_coroutine running"<<std::endl;
60     co_await std::suspend_always();
61     std::cout<<"three_step_coroutine end"<<std::endl;
62 }
63 int main(){
64     future_type ret = three_step_coroutine();
65     std::cout<<"=====calling first resume====="<<std::endl;
66     ret.resume();
67     std::cout<<"=====calling second resume====="<<std::endl;
68     ret.resume();
69     std::cout<<"=====calling third resume====="<<std::endl;
70     ret.resume();
71     std::cout<<"=====main end====="<<std::endl;
72
73     return 0;
74 }
```

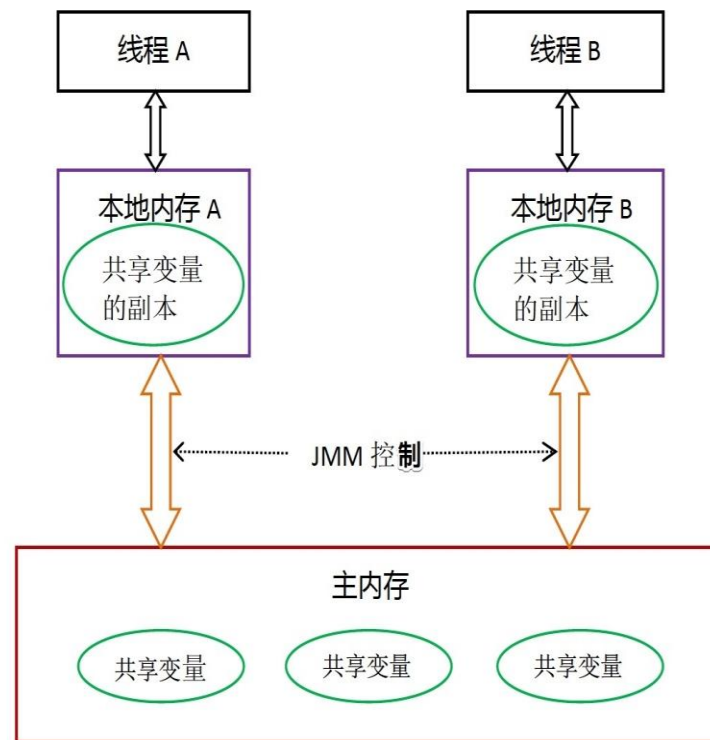
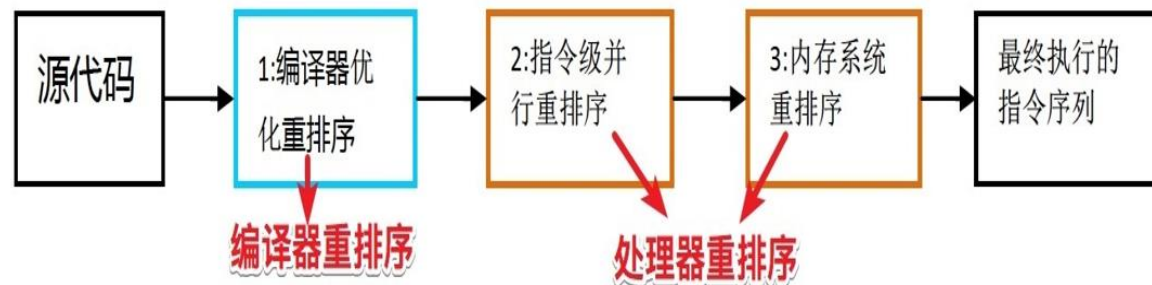
```
1 | promise_type constructor
2 | get_return_object
3 | initial_suspend
4 | future_type constructor
5 | =====calling first resume=====
6 | three_step_coroutine begin
7 | =====calling second resume=====
8 | three_step_coroutine running
9 | =====calling third resume=====
10 | three_step_coroutine end
11 | return_void
12 | final_suspend
13 | =====main end=====
14 | future_type destructor
15 | promise_type destructor
```

深入理解后端编程语言并行技术—Java

Java内存模型-JMM 那JVM内存模型是什么？

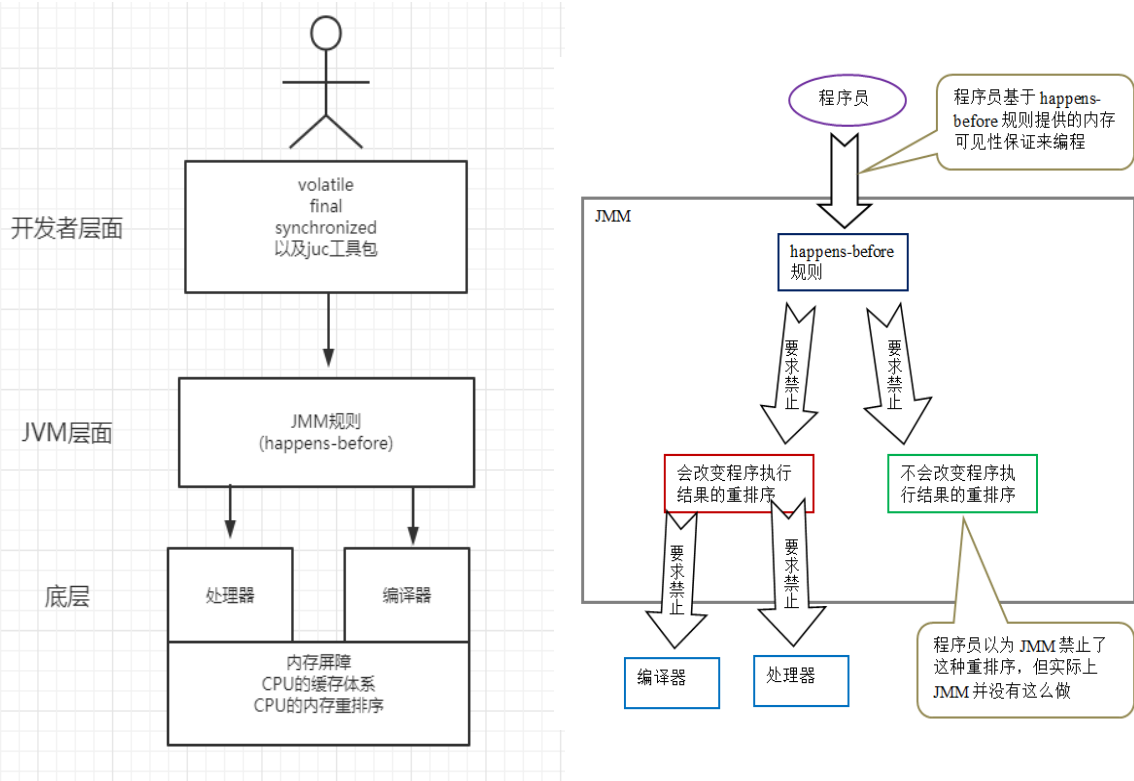


Java 内存模型是共享内存的并发模型，线程之间主要通过读-写共享变量来完成隐式通信



深入理解后端编程语言并行技术—Java

Java并发核心是深入理解Java内存模型（JMM）



JMM的设计目标

- 一方面，对于程序员来说，我们希望内存模型易于理解、易于编程，为此 JMM 的设计者要为程序员提供足够强的内存可见性保证，专业术语称之为“**强内存模型**”。
- 而另一方面，编译器和处理器则希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化（比如重排序）来提高性能，因此 JMM 的设计者对编译器和处理器的限制要尽可能地放松，专业术语称之为“**弱内存模型**”。

JMM的设计方案

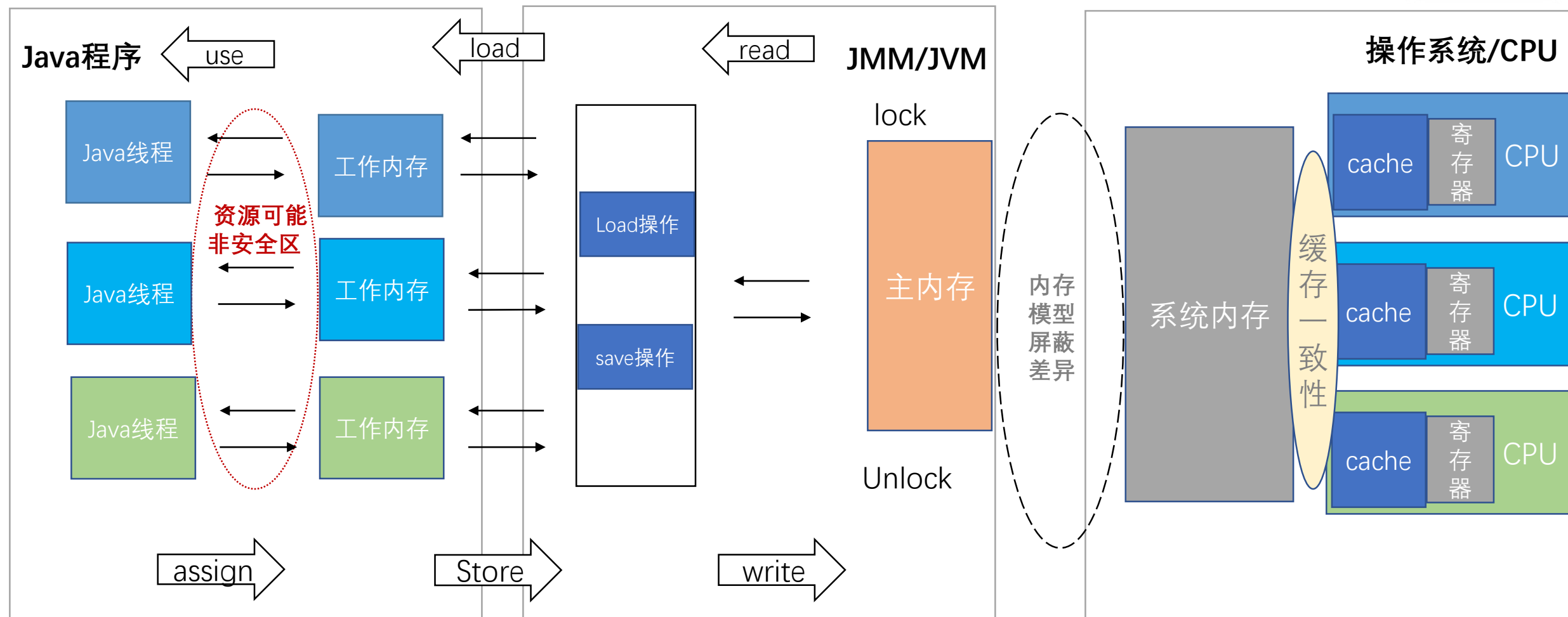
- 单线程程序。**单线程程序不会出现内存可见性问题。编译器，runtime 和处理器会共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同。
- 正确同步的多线程程序。**正确同步的多线程程序的执行将具有顺序一致性（程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同）。这是 JMM 关注的重点，JMM 通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。
- 未同步 / 未正确同步的多线程程序。**JMM 为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0, null, false）。

深入理解后端编程语言并行技术—Java

有了内存模型JMM,为啥程序员还要考虑并发编程问题呢？



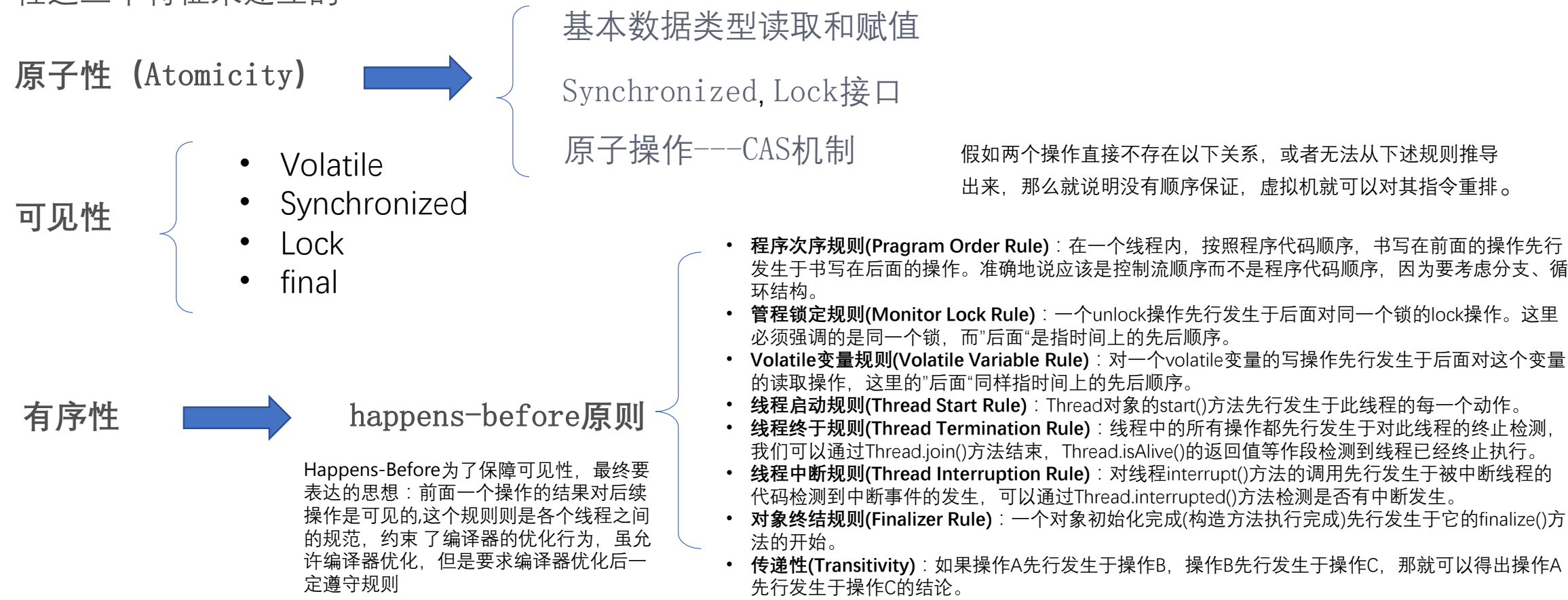
By 公众号@极客重生



深入理解后端编程语言并行技术—Java

Java并发核心是深入理解Java内存模型（JMM）

Java内存模型是围绕着并发编程中原子性、可见性、有序性这三个特征来建立的



深入理解后端编程语言并行技术—Java

Java并行编程-Concurrent包



- 最底层是volatile读/写和CAS；
- 第二层基础类是AQS、非阻塞数据结构和原子变量类；这些基础类使用类似的实现方式：
 - (1) 声明共享变量（状态）为volatile类型；
 - (2) 使用CAS原子更新完成线程之间的同步；
 - (3) 利用volatile读/写的内存语义和CAS同时具备的volatile读和写的内存语义实现线程之间的通信。
- 第三层高层类是Lock、同步器、阻塞队列、Executor和并发容器。高层类基于第二层的基础类实现。

深入理解后端编程语言并行技术—GO

Memory Model in Golang

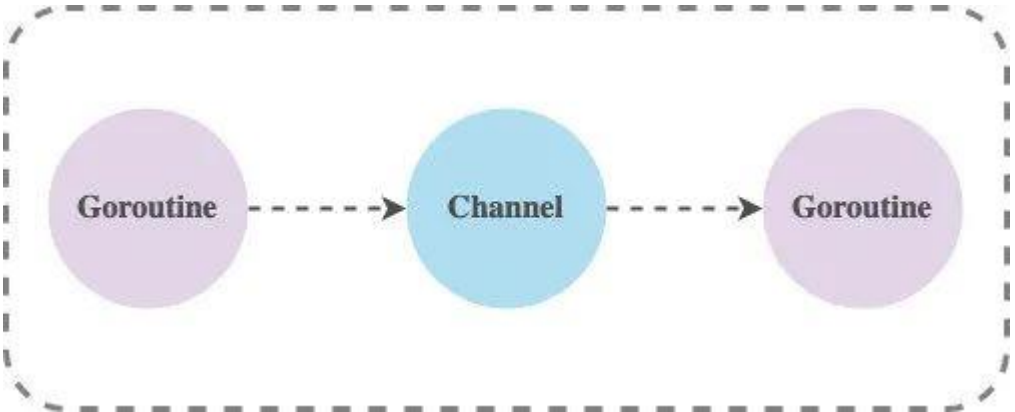
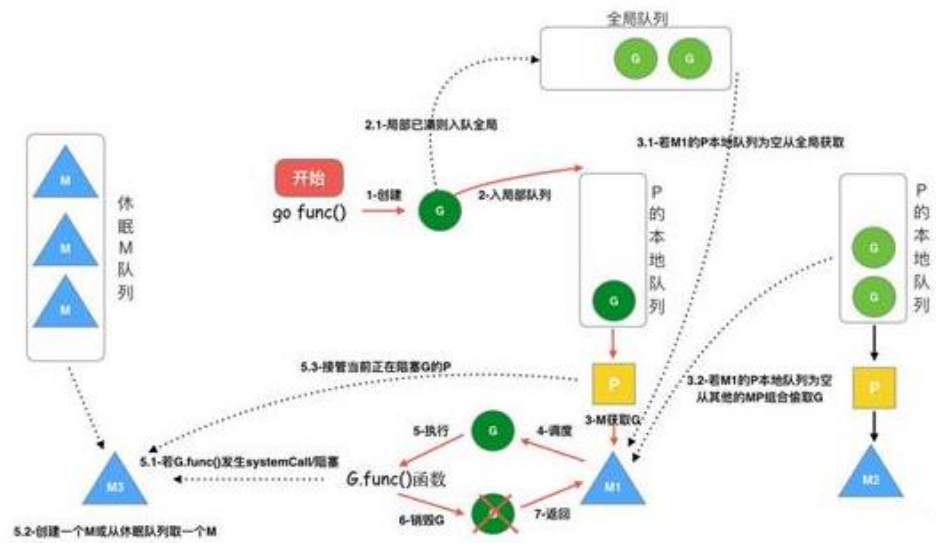
- 先于关系(happens-before):内存操作之间的偏序关系,所有的操作存在先于关系或者并发
- A导入了B, B的init函数先执行
- main package的init函数先于main函数执行
- go语句先于其创建的协程执行
- Unlock先于Lock
- 单个goroutine中满足程序次序

```
var c = make(chan int, 10)
var a string

func f() {
    a = "hello, world"
    c <- 0
}

func main() {
    go f()
    <-c
    print(a)
}
```

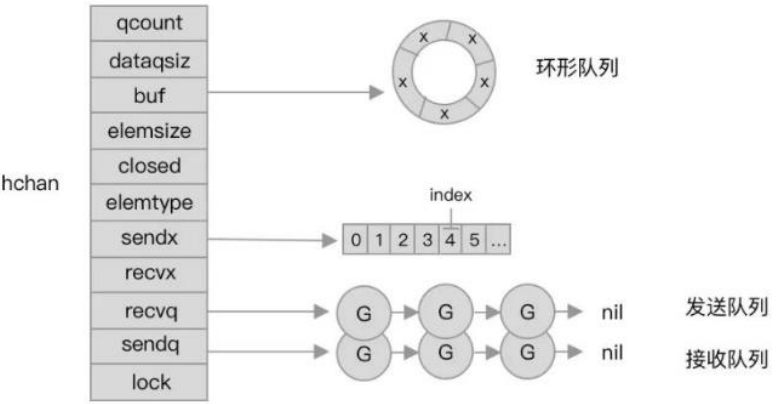
深入理解后端编程语言并行技术—GO



channel

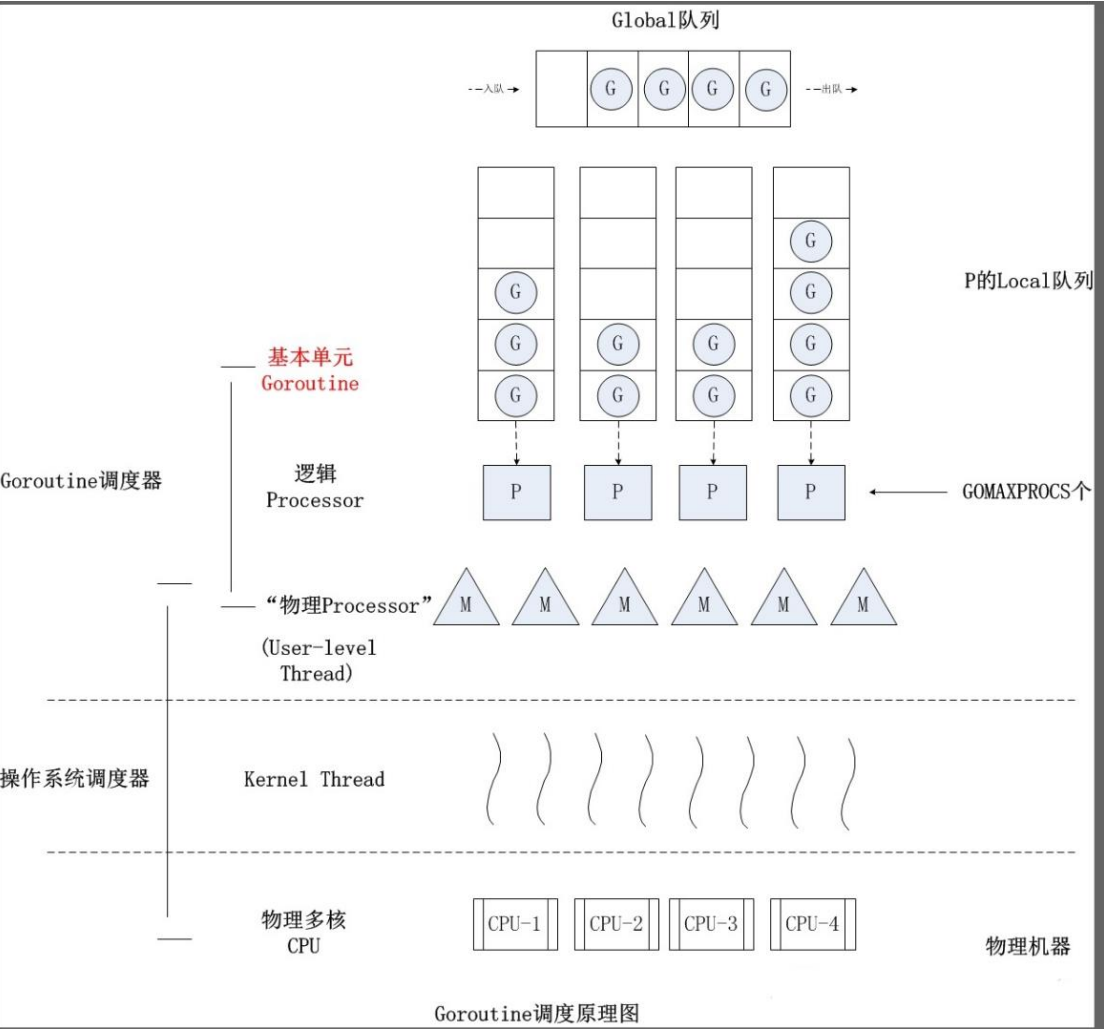
环形队列 上互斥锁,
阻塞/非阻塞,
缓冲/非缓冲,
缓存出队列,
拷贝数据,
解互斥锁,
协程调度

- 单线程调度器 • [0.x](#) GM 模型
- 多线程调度器 • [1.0](#) GM 模型
- 任务窃取调度器 • [1.1](#) GMP 模型
- 抢占式调度器 • [1.2](#) ~ 至今 GMP 模型



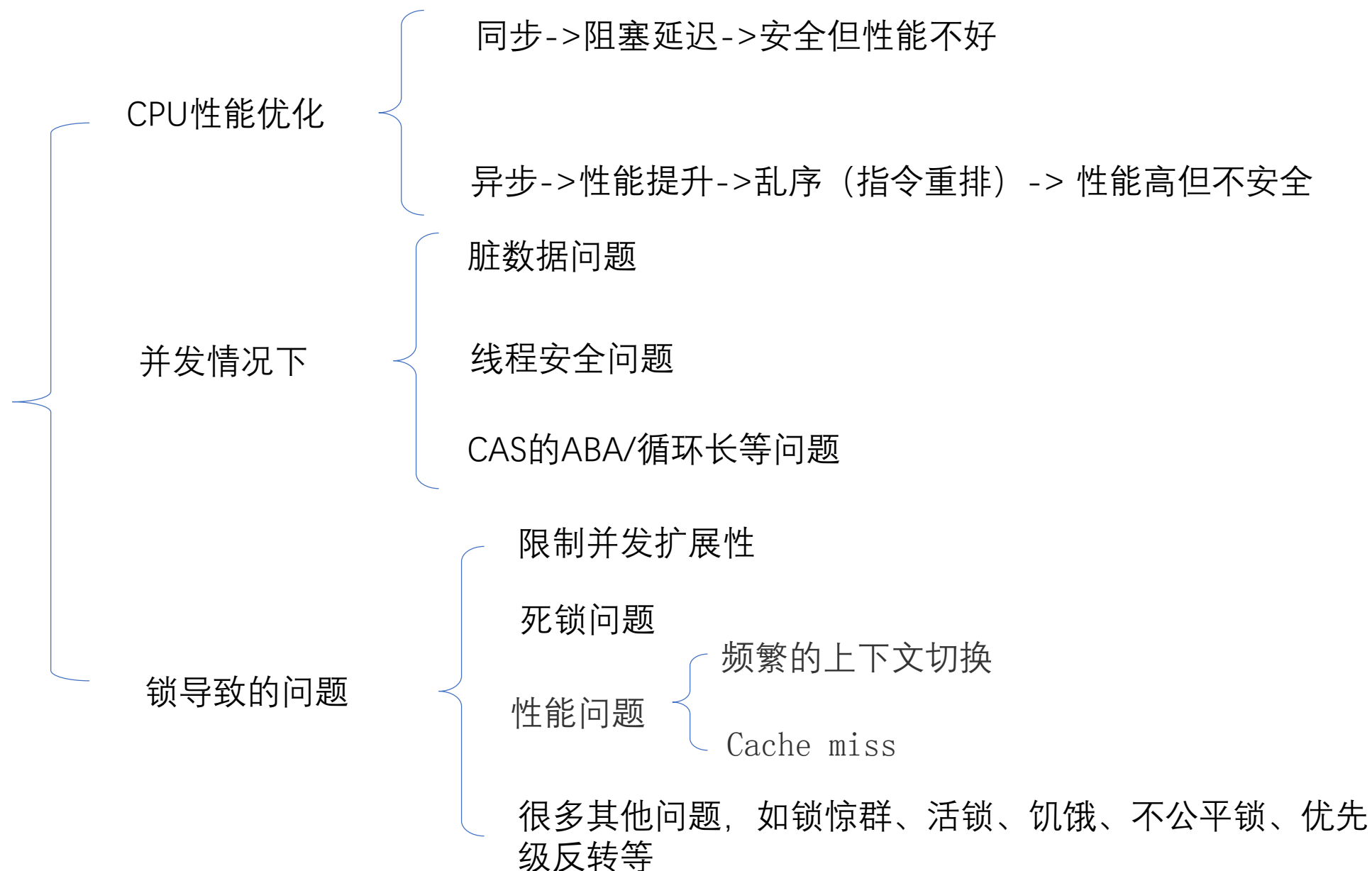
深入理解后端编程语言并行技术—GO

GPM调度模型



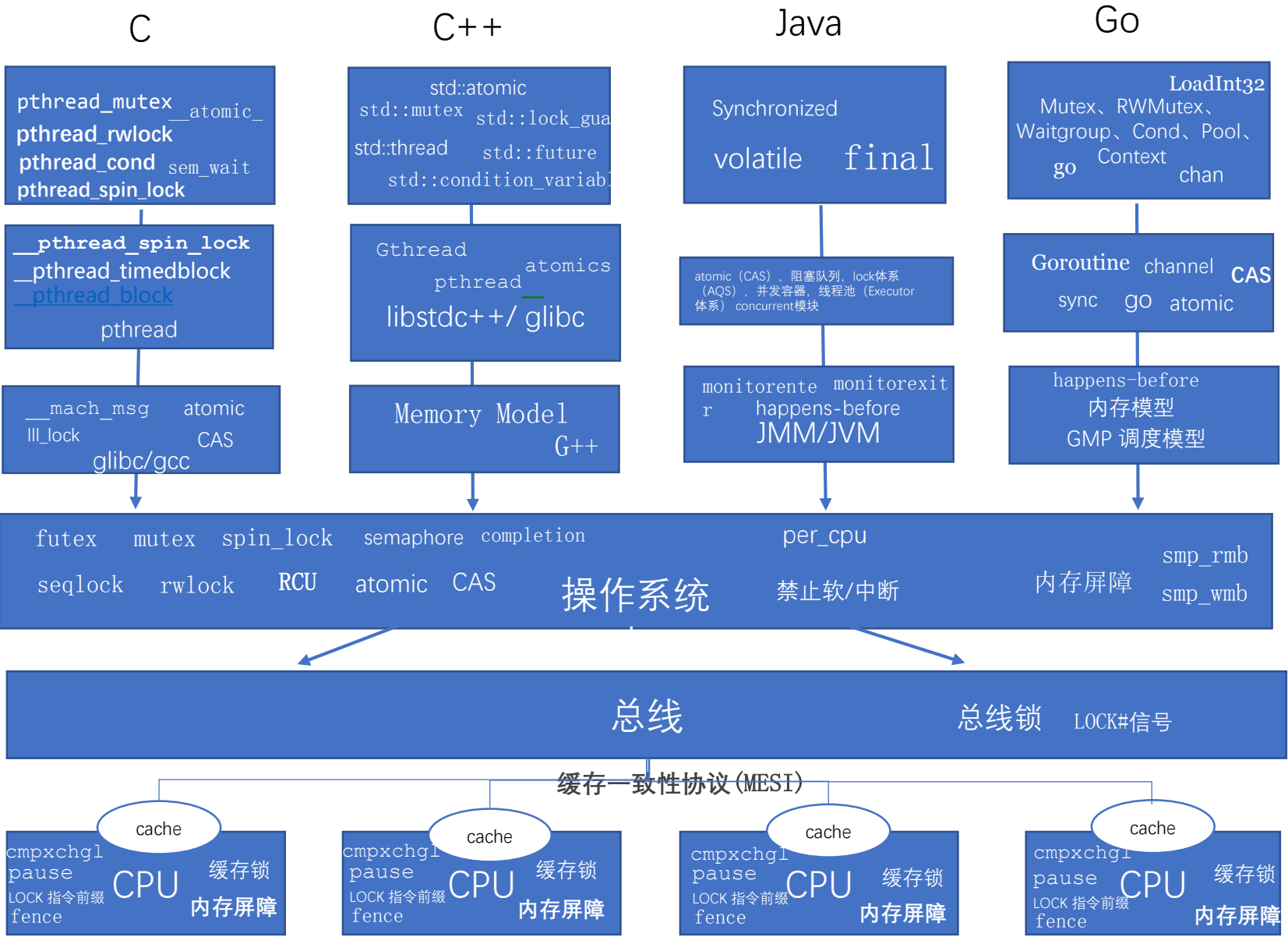
- 在 Go 中，线程是运行 goroutine 的实体，调度器的功能是把可运行的 goroutine 分配到工作线程上。
- 全局队列（Global Queue）**：存放等待运行的 G。
- P 的本地队列**：同全局队列类似，存放的也是等待运行的 G，存的数量有限，不超过 256 个。新建 G' 时，G' 优先加入到 P 的本地队列，如果队列满了，则会把本地队列中一半的 G 移动到全局队列。
- P 列表**：所有的 P 都在程序启动时创建，并保存在数组中，最多有 GOMAXPROCS(可配置) 个。
- M**：线程想运行任务就得获取 P，从 P 的本地队列获取 G，P 队列为空时，M 也会尝试从全局队列拿一批 G 放到 P 的本地队列，或从其他 P 的本地队列偷一半放到自己 P 的本地队列。M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。
- Goroutine 调度器**和 OS 调度器是通过 M 结合起来的，每个 M 都代表了 1 个内核线程，OS 调度器负责把内核线程分配到 CPU 的核上执行。

并发优化是把双刃剑—带来的问题如何解决



并发/并行问题技术大局观

公众号：极客重生



编程语言

标准库, 虚拟机, 编译器

操作系统

硬件



并行问题如何解决

cache coherence（缓存一致性协议）解决的是单一地址（单个变量）的写问题，可以使多核心对同一地址的写入序列化。

而memory consistency（内存一致性，内存模型）说的是不同地址（多个变量）的读写的顺序问题。即全局视角对读写的观测顺序问题。

不同内存地址读写的可见性问题，要解决（内存一致性，内存模型）的问题，需要使用memory barrier（**内存屏障**）之类的接口。

内存屏障：保证多个变量读写顺序符合预期

smb_wmb()：写内存屏障，刷新store buffer，同时限制编译器和CPU的乱序优化；

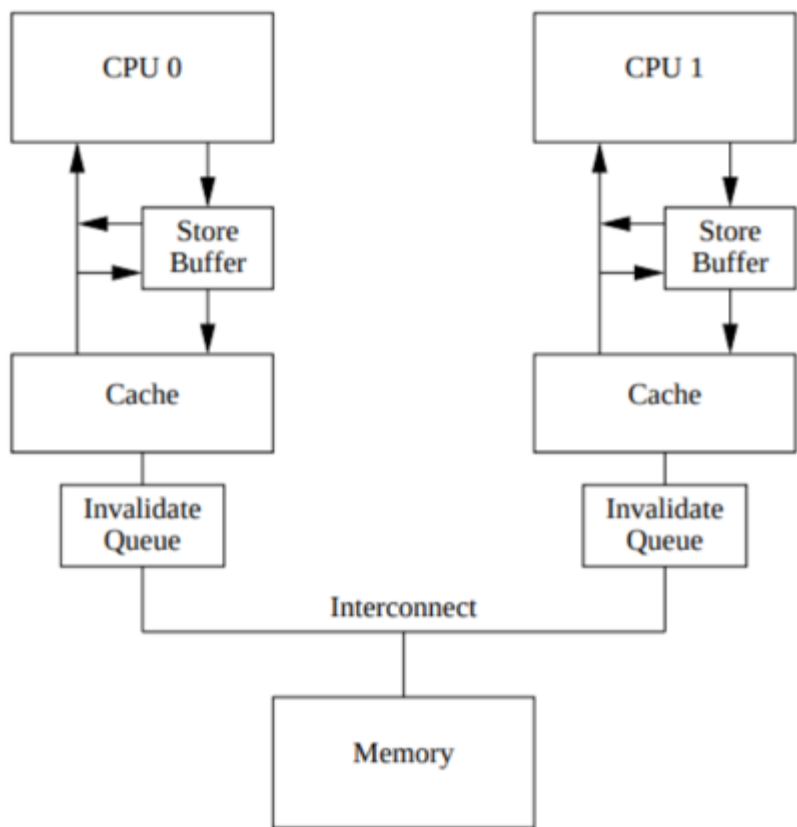
smb_rmb()：读内存屏障，刷新invalidate queue，同时限制编译器和CPU的乱序优化；

smb_mb()：读写内存屏障，同时刷新store buffer和invalidate queue，同时限制编译器和CPU的乱序优化；保证多个变量读写顺序符合预期

处理器并行导致乱序问题如何解决

被动：为了异步化指令的执行，引入Store Buffer和Invalidate Queue，却导致了「指令顺序改变」的副作用。

主动：编译优化，会造成乱序；分支预测、多流水线等CPU硬件优化技术，会造成乱序；



- store buffer模块改善cache write由于应答延迟而造成的写停顿问题；
- invalidate queue模块改善使无效应答的时延，把使无效命令放入queue后就立即发送应答；
- store buffer引起的延迟处理，会造成乱序；
- invalidate queue引起的延迟处理，会造成乱序；
- 如何解决Store Buffer和Invalidate Queue带来的乱序问题？
- Read Memory Barrier：
 - flush invalidate queue
 - all loads preceding the barrier -> loads following the barrier
- Write Memory Barrier：
 - flush store buffer
 - all stores preceding the barrier -> stores following the barrier

CPU->cache->MESI->Store Buffer->Store Forwarding->写屏障指令->Invalidate Queue->内存屏障（软件层面）

并发/并行编程技术

并发编程中主要需要解决两个问题：

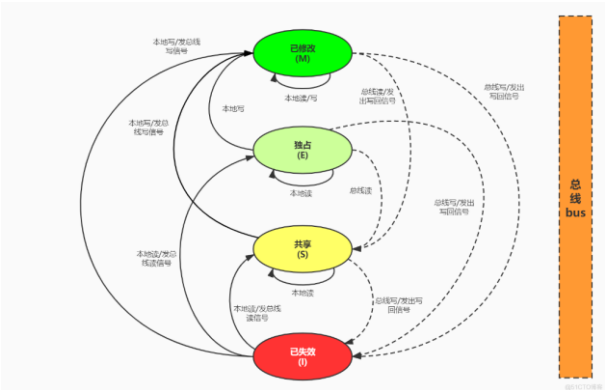
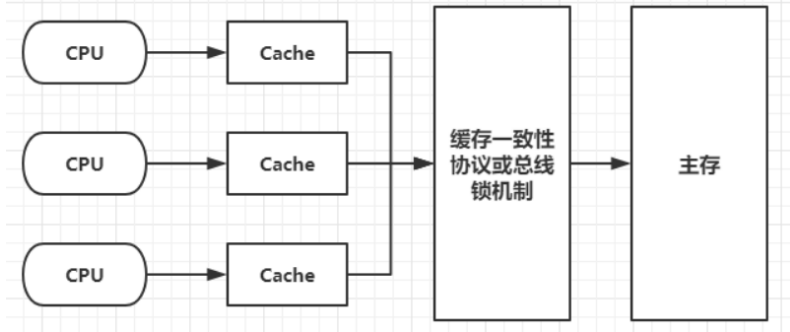
1. 线程之间如何通信
2. 线程之间如何完成同步

- 多线程
- 多进程
- 多核编程
- 无锁编程
- 并行编程框架：
 - MPI
 - OpenCL
 - CUDA

并行问题编程如何解决

整数并发问题 → 原子操作 → 硬件级加锁 → CPU lock指令

锁总线
锁缓存



缓存一致性协议

锁总线：
具体方法是，一旦遇到了Lock指令，就由仲裁器选择一个核心独占总线。其余的CPU核心不能再通过总线与内存通讯。从而达到“原子性”的目的
这种方式的确能解决问题，但是非常不高效。为了个原子性结果搞得其他CPU都不能干活了。

锁缓存：
相比总线锁，缓存锁即降低了锁的力度。核心机制是基于缓存一致性协议来实现的。
MESI（缓存一致性协议）大致的意思是：若干个CPU核心通过ringbus连到一起。每个核心都维护自己的Cache的状态。如果对于同一份内存数据在多个核里都有cache，则状态都为S（shared）。一旦有一核心改了这个数据（状态变成了M），其他核心就能瞬间通过ringbus感知到这个修改，从而把自己的cache状态变成I（Invalid），并且从标记为M的cache中读过来。
同时，这个数据会被原子的写回到主存。最终，cache的状态又会变为S。这相当于给cache本身单独做了一套总线，避免了真的锁总线，相比总线锁，缓存锁即降低了锁的力度，一致性协议核心就是同步各个CPU的cache 和分布式一致性同步各个节点数据类似：
详细介绍：<https://www.cnblogs.com/xiaolincoding/p/13886559.html>

并行问题编程如何解决

避免了多线程间变量值不一致的问题（可见性）

Java : volatile
final

禁止重排

内存屏障

解决JMM的可见性和重排序问题的

CPU mfence指令

C/C++ : volatile

C/C++ : 内存模型+内存屏障

CPU mfence指令

非原子性要保证原子性

加锁

Synchronized
Lock
Mutex
Rwlock
Spinlock
...

代价大，编程简单

Java : monitorenter/monitorexit
Go : Sync

Futex 阻塞

CAS

自旋

堵塞

原子操作

CPU lock指令

无锁

CAS操作

CPU lock指令

CPU CMPXCHG指令（原子性操作）

代价小，编程复杂

compare_and_swap

并行问题编程如何解决---无锁编程

无锁编程的核心是：CAS接口和内存模型

核心编程框架：

```
for（无限循环）{  
    a = get();  
    // if a==b,then ,a = c ;  
    if(cas(a,b,c)){  
        do_something();  
        break;  
    }  
}
```

```
1.  bool compare_and_swap (int *addr, int oldval, int newval)  
2.  {  
3.      if ( *addr != oldval ) {  
4.          return false;  
5.      }  
6.      *addr = newval;  
7.      return true;  
8.  }
```

核心思想:当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值

CAS语义

```
58:   
59: #if __GNUC_PREREQ(4,1)  
60: #define atomic_compare_and_exchange_val_acq(mem, newval, oldval) \   
61: __sync_val_compare_and_swap(mem, oldval, newval)  
62: #define atomic_compare_and_exchange_bool_acq(mem, newval, oldval) \   
63: (!__sync_bool_compare_and_swap(mem, oldval, newval))  
64: #else  
65: #define __arch_compare_and_exchange_val_8_acq(mem, newval, oldval) \   
66: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgb %b2, %1" \   
67: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
68: ret; })  
69: #define __arch_compare_and_exchange_val_16_acq(mem, newval, oldval) \   
70: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgw %w2, %1" \   
71: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
72: ret; })  
73: #define __arch_compare_and_exchange_val_32_acq(mem, newval, oldval) \   
74: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgl %l2, %1" \   
75: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
76: ret; })  
77: #define __arch_compare_and_exchange_val_64_acq(mem, newval, oldval) \   
78: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgl %l2, %1" \   
79: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
80: ret; })  
81: #define __arch_compare_and_exchange_val_8_acq(mem, newval, oldval) \   
82: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgb %b2, %1" \   
83: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
84: ret; })  
85: #define __arch_compare_and_exchange_val_16_acq(mem, newval, oldval) \   
86: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgw %w2, %1" \   
87: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
88: ret; })  
89: #define __arch_compare_and_exchange_val_32_acq(mem, newval, oldval) \   
90: ({ __typeof__(*mem) ret; __asm__ volatile (LOCK_PREFIX "cmpxchgl %l2, %1" \   
91: : "=a" (ret), "=m" (*mem) : "r" (newval), "r" (oldval)); \   
92: ret; })  
93: #endif  
94:   
95:
```

glibc

并行问题编程如何解决---无锁编程

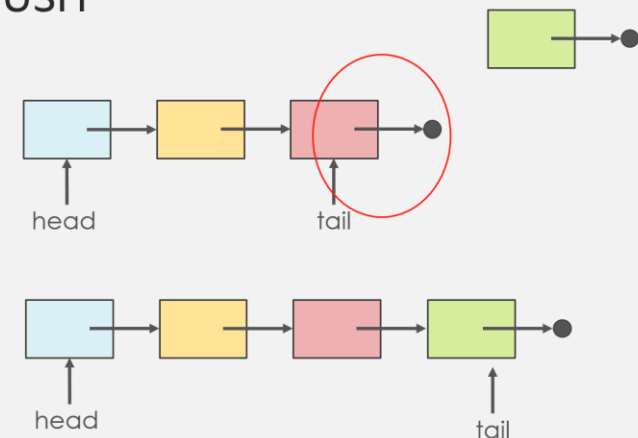
无锁编程的核心是：CAS接口和内存模型

无锁队列的链表实现

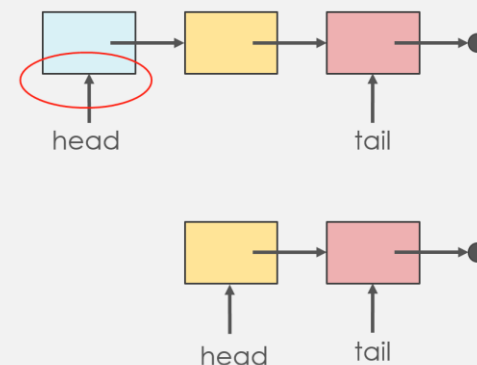
```
1. EnQueue(Q, data) //进队列
2. {
3.     //准备新加入的结点数据
4.     n = new node();
5.     n->value = data;
6.     n->next = NULL;
7.
8.     do {
9.         p = Q->tail; //取链表尾指针的快照
10.    } while( CAS(p->next, NULL, n) != TRUE);
11.    //while条件注释: 如果没有把结点链在尾指针上, 再试
12.
13.    CAS(Q->tail, p, n); //置尾结点 tail = n;
14. }
```

```
enqueue(Q: pointer to queue_t, value: data type)
E1:  node = new_node()    // Allocate a new node from the free list
E2:  node->value = value   // Copy enqueued value into node
E3:  node->next.ptr = NULL // Set next pointer of node to NULL
E4:  loop                  // Keep trying until Enqueue is done
E5:      tail = Q->Tail    // Read Tail.ptr and Tail.count together
E6:      next = tail.ptr->next // Read next ptr and count fields together
E7:      if tail == Q->Tail // Are tail and next consistent?
          // Was Tail pointing to the last node?
E8:          if next.ptr == NULL
              // Try to link node at the end of the linked list
E9:              if CAS(&tail.ptr->next, next, <node, next.count+1>)
E10:                 break // Enqueue is done. Exit loop
E11:             endif
E12:          else // Tail was not pointing to the last node
              // Try to swing Tail to the next node
E13:              CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E14:          endif
E15:      endif
E16:  endloop
```

PUSH



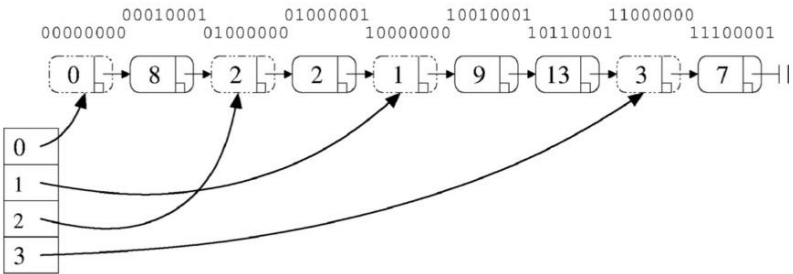
POP



并行问题编程如何解决---无锁编程

无锁编程例子

```
320:
321: private:
322: #ifndef BOOST_DOXYGEN_INVOKED
323: ... template<bool Bounded>
324: ... bool do_push(T const& t)
325: ... {
326: ... node* n = pool.template construct<true, Bounded>(t, pool.null_handle());
327: ... handle_type node_handle = pool.get_handle(n);
328:
329: ... if (n == NULL)
330: ... return false;
331:
332: ... for (;;) {
333: ... tagged_node_handle tail = tail.load(memory_order_acquire);
334: ... node* tail_node = pool.get_pointer(tail);
335: ... tagged_node_handle next = tail_node->next.load(memory_order_acquire);
336: ... node* next_ptr = pool.get_pointer(next);
337: ... tagged_node_handle tail2 = tail.load(memory_order_acquire);
338:
339: ... //进行比较是为了防止其他线程进行Push导致了tail变化
340: ... //内存模型memory_order_acquire保证tail,tail2取值的顺序性
341: ... if (BOOST_LIKELY(tail == tail2)) {
342: ... if (next_ptr == 0) { //Tail.next必须指向0才进行push下一步关键操作
343: ... tagged_node_handle new_tail_next(node_handle, next.get_next_tag());
344: ... //这里next一般指向0, 尝试将Tail.next赋值成新node.next的数值
345: ... //若是第一个push node那么Head.next也会改变
346: ... //赋值的目的是为了将新node与原有node形成链状关系
347: ... if (!tail_node->next.compare_exchange_weak(next, new_tail_next)) {
348: ... //成功赋值, 将Tail前移 (指向新node)
349: ... tagged_node_handle new_tail(node_handle, tail.get_next_tag());
350: ... tail.compare_exchange_strong(tail, new_tail);
351: ... return true;
352: ... }
353: ... }
354: ... else { //若不为0, 可能另外一个线程提前push, 我们进入else将Tail指向Tail.next
355: ... //移动Tail
356: ... tagged_node_handle new_tail(pool.get_handle(next_ptr), tail.get_next_tag());
357: ... tail.compare_exchange_strong(tail, new_tail);
358: ... }
359: ... }
360: ... } //end-for;;
361: ... } //end-do_push
362: #endif
363:
364: public:
365:
```



```
void *lf_dynarray_lvalue(LF_DYNARRAY *array, uint idx) {
// 通过边界值, 确定属于哪个Level
for (i = LF_DYNARRAY_LEVELS - 1; idx < dynarray_idxes_in_prev_levels[i]; i--)

// 取出对应Level的起始地址
std::atomic<void*> *ptr_ptr = &array->level[i];

// 减去初始偏移地址后, idx就变成了Level内部的相对偏移地址
idx -= dynarray_idxes_in_prev_levels[i];

for (; i > 0; i--) {
if (!ptr = *ptr_ptr) {
// 如果bucket不存在, 则申请一块新内存
void *alloc = my_malloc(key_memory_lf_dynarray,
LF_DYNARRAY_LEVEL_LENGTH * sizeof(void *),
MYF(MY_WME | MY_ZEROFILL));

if (unlikely(!alloc)) {
return (NULL);
}

// CAS操作,
if (atomic_compare_exchange_strong(ptr_ptr, &ptr, alloc)) {
ptr = alloc;
} else {
// 更新失败, 说明已经有其它线程更新成功, 释放当前分块即可
my_free(alloc);
}
}

// 计算块内的偏移地址
ptr_ptr =
((std::atomic<void*> *)ptr) + idx / dynarray_idxes_in_prev_level[i];
idx %= dynarray_idxes_in_prev_level[i];
}
}
```

Boost-无锁队列

MySQL-无锁HASH实现 (LF_HASH)

并行问题编程如何解决---无锁编程

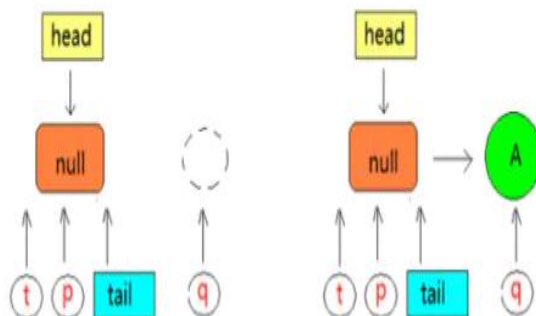
无锁编程例子

```
public boolean offer(E e) {
    checkNotNull(e);
    final Node<E> newNode = new Node<E>(e);

    for (Node<E> t = tail, p = t;;) {
        Node<E> q = p.next;
        if (q == null) {
            if (p.casNext( cmp: null, newNode)) { ①
                if (p != t) ②
                    casTail(t, newNode); ③
                return true;
            }
        }
        else if (p == q) ④
            p = (t != (t = tail)) ? t : head; ⑤
        else
            p = (p != t && t != (t = tail)) ? t : q; ⑥
    }
}
```

添加A:

添加A时, 进入循环 t=p=tail, q=p.next, 因为p=tail为最后节点, 进入q==null分支, 执行①, p.next=newNode, 此时p!=t=tail, 直接退出



JAVA-无锁队列 (ConcurrentLinkedQueue)

```
1 // updateTimerModifiedEarliest updates the recorded nextwhen field of the
2 // earlier timerModifiedEarlier value.
3 // The timers for pp will not be locked.
4 func updateTimerModifiedEarliest(pp *p, nextwhen int64) {
5     for {
6         old := atomic.Load64(&pp.timerModifiedEarliest)
7         if old != 0 && int64(old) < nextwhen {
8             return
9         }
10        if atomic.Cas64(&pp.timerModifiedEarliest, old, uint64(nextwhen)) {
11            return
12        }
13    }
14 }
```

```
// TryRLock tries to lock rw for reading and reports whether it succeeded.
// Note that while correct uses of TryRLock do exist, they are rare,
// and use of TryRLock is often a sign of a deeper problem
// in a particular use of mutexes.
func (rw *RWMutex) TryRLock() bool {
    if race.Enabled {
        _ = rw.w.state
        race.Disable()
    }
    for {
        c := atomic.LoadInt32(&rw.readerCount)
        if c < 0 {
            if race.Enabled {
                race.Enable()
            }
            return false
        }
        if atomic.CompareAndSwapInt32(&rw.readerCount, c, c+1) {
            if race.Enabled {
                race.Enable()
                race.Acquire(unsafe.Pointer(&rw.readerSem))
            }
            return true
        }
    }
}
```

Golang-库大量用无锁编程技术

并行问题编程如何解决---无锁编程

资料文章

- [无锁队列的实现 | 酷 壳 – CoolShell](#)
- [Yet another implementation of a lock-free circular array queue | CodeProject](#)
- [无锁数据结构（基础篇）：内存模型](#)
- [无锁数据结构（机制篇）：内存管理规则](#)
- [Category: 并行编程 - Yebangyu's Blog](#)
- [awesome-lockfree](#)

Go并发编程学习矩阵

知识主线



基本并发原语



原子操作



Channel



扩展并发原语



分布式并发原语

学习主线

01

基础用法

02

实现原理

03

易错场景

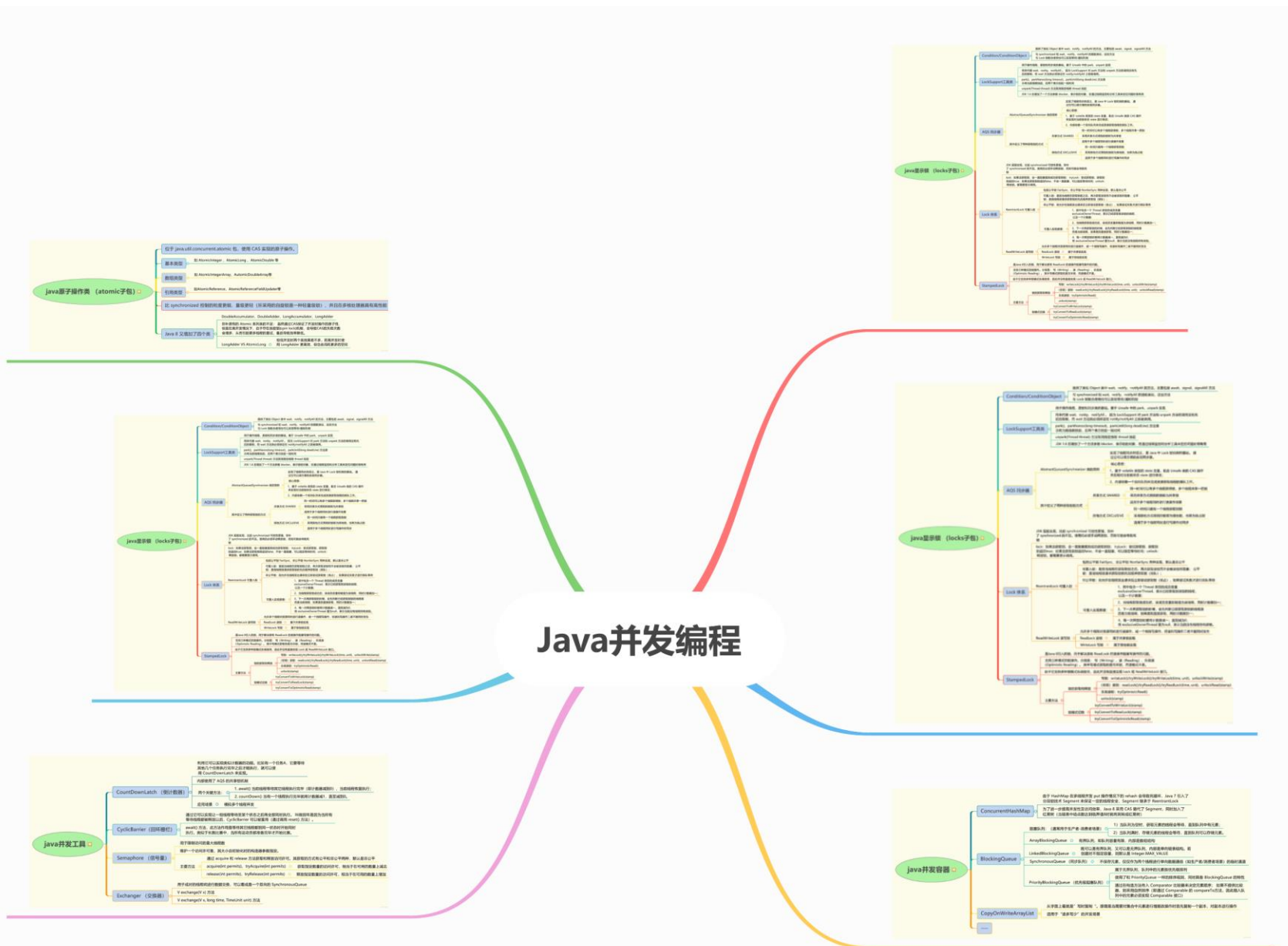
04

知名项目中的Bug

并行技术总结—并发技术图谱--go



并行技术总结—并发技术图谱--Java



并行技术总结—并发技术图谱—C++

C++多线程

第二章：线程管理

- 基本线程管理
 - 启动线程
 - 等待线程完成
 - 在异常环境下的等待
 - 在后台运行线程
- 传递参数给线程函数
- 转移线程的所有权
- 在运行时选择线程数量
- 标识线程

第三章：在线程间共享数据

- 线程间共享数据
 - 竞争条件
 - 避免有问题的竞争条件
- 用互斥元保护共享数据
 - 使用C++中的互斥元
 - 为保护共享数据精心组织代码
 - 发现接口中固有竞争条件
 - 死锁：问题和解决方案
 - 避免死锁的进一步指南
 - 在std::unique_lock灵活锁定
 - 在作用域之间转移的所有权
 - 锁定在恰当的粒度
- 用于共享数据保护的替代工具
 - 在初始化时保护共享数据
 - 保护很少更新的数据结构
 - 递归锁

第四章：同步并发操作

- 等待事件或其他条件
 - 用条件变量等待条件
 - 使用条件变量建立一个线程安全队列
- 使用future等待一次性事件
 - 从后台任务中返回值
 - 将任务与future相关联
 - 生成std::promise
 - 为future保存异常
 - 等待自多个线程
- 有时限制的等待
 - 时钟
 - 时间段
 - 接受超时的函数
- 使用同步操作来简化代码
 - 带有future的函数式编程
 - 具有消息传递的同步操作

第五章：C++内存模型和原子类型上操作

- 内存模型基础
 - 对象和内存位置
 - 对象、内存位置以及并发
 - 修改顺序
- C++中的原子操作及类型
 - 标准原子类型
 - std::atomic_flag上的操作
 - 基于std::atomic<bool>的操作
 - std::atomic<T*>上的操作：指针算术操作
 - 标准原子整型的操作
 - std::atomic<> 初级类模板
 - 原子操作的自由函数
- 同步操作和强制顺序
 - synchronizes-with关系
 - happens-before关系
 - 原子操作的内存顺序
 - 释放序列和synchronizes-with
 - 屏障
 - 用原子操作排序非原子操作

第六章：设计基于锁的并发数据结构

- 为开发设计的含义时什么
- 基于锁的并发数据结构
 - 使用安全锁的线程安全栈
 - 使用锁和条件变量的线程安全队列
 - 使用细粒度锁和条件变量的线程安全队列
- 设计更复杂的基于锁的数据结构
 - 编写一个使用锁的线程安全查找表
 - 编写一个使用细粒度锁的线程安全字典

第七章：设计无锁的并发数据结构

- 定义和结果
 - 非阻塞数据结构的类型
 - 无锁数据结构
 - 无等待的数据结构
 - 无锁数据结构的优点和缺点
- 无锁数据结构的例子
 - 编写不用锁的线程安全栈
 - 停止恼人的泄露：在无锁数据结构中管理内存
 - 用风险指针检测不能被回收的结点
 - 使用引用技术检测结点
 - 将内存模型应用至无锁栈
 - 编写不用锁的线程安全队列
- 编写无锁数据结构的准则
 - 准则：使用std::memory_order_seq_cst作为原型
 - 准则：使用无锁内存回收模式
 - 准则：当心ABA问题
 - 准则：识别忙于等待的循环以及辅助其他线程

第八章设计并发代码

- 在线程间划分工作的技术
 - 处理开始前在线程间划分数据
 - 递归地划分数据
 - 以任务类型划分工作
 - 有多少处理器
 - 数据竞争和乒乓缓存
- 影响并发代码性能的因素
 - 假共享
 - 数据应该多紧密
 - 过度订阅和过多的任务切换
- 为多线程性能设计数据结构
 - 为复杂操作划分数组元素
 - 其他数据结构中的数据访问方式
- 为并发设计时的额外考虑
 - 并行算法中的异常安全
 - 可扩展性和阿姆达尔定律
 - 用多少线程隐藏延时
 - 用并发提高响应性
- 在实践中设计并发代码
 - std::for_each的并行实现
 - std::find的并行实现
 - std::partial_sum的并行实现

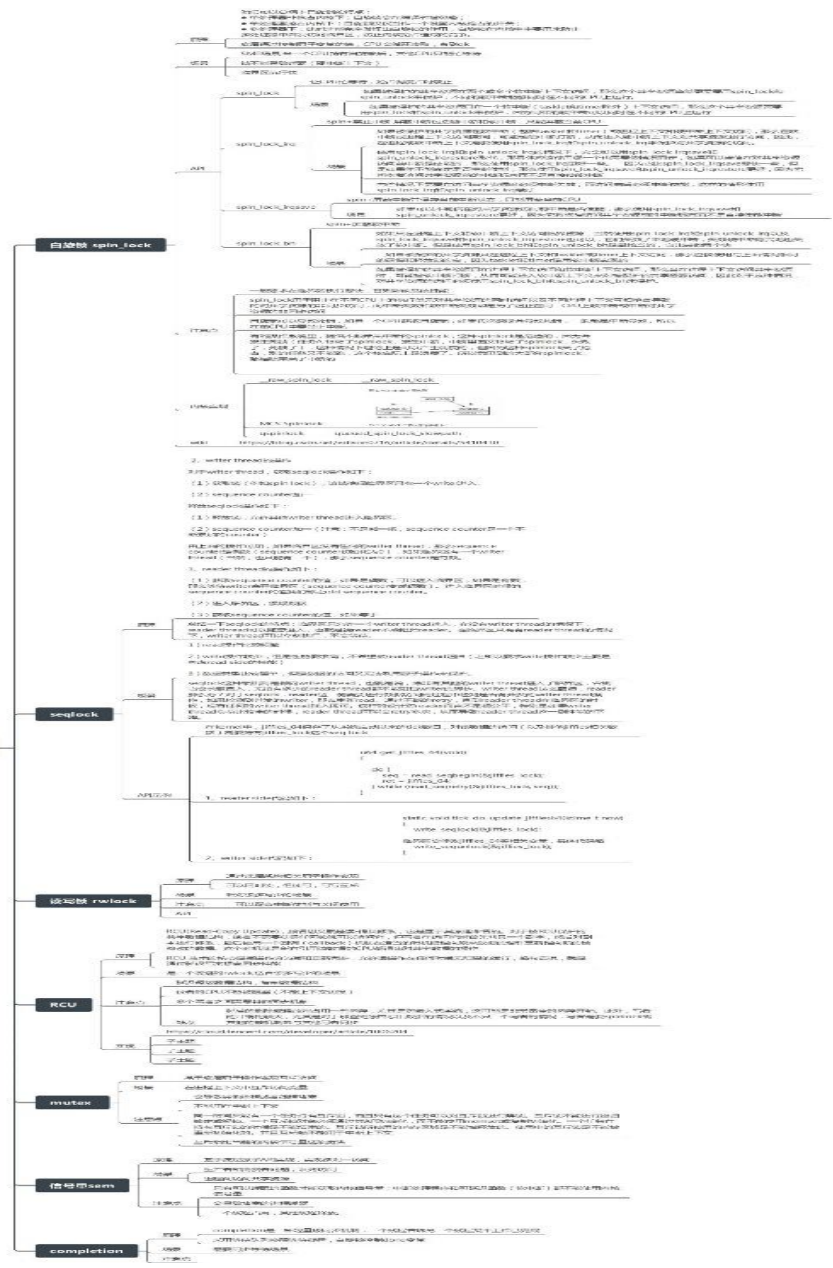
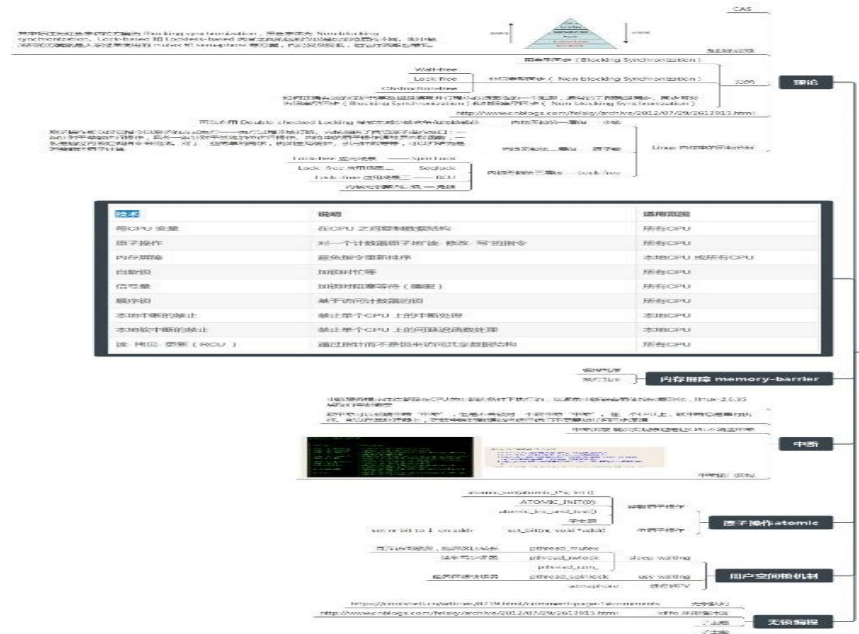
第九章：高级线程管理

- 线程池
 - 最简单的线程池
 - 等待提交给线程池的任务
 - 等待其他任务的任务
 - 避免工作队列上的竞争
 - 工作窃取
- 中断线程
 - 启动和中断另一个线程
 - 检测一个线程是否被中断
 - 中断等待条件变量
 - 中断在std::condition_variable_any上的其他等待
 - 中断其他阻塞调用
 - 处理中断
 - 在应用退出时中断后台任务

第十章：多线程应用的测试与调试

- 开发相关错误的类型
 - 不必要的阻塞
 - 竞争条件
- 定位并发相关的错误技巧
 - 审阅代码以及定位潜在的错误
 - 通过测试定位并发相关的错误
 - 可测试性设计
 - 多线程测试技术
 - 构建多线程的测试代码
 - 测试多线程代码性能

并行技术总结—并发技术图谱—Linux内核



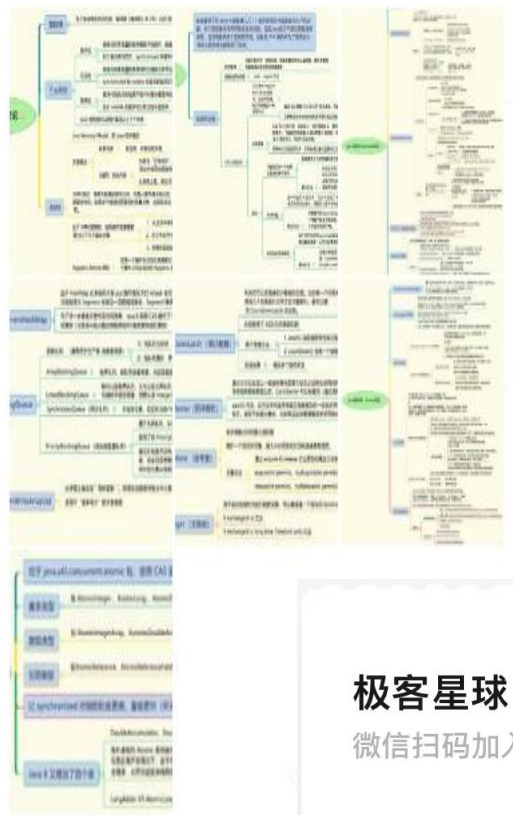
并行技术总结—并发技术图谱-高清版都在极客星球



大师兄|荣哥
4分钟前

收进专栏

JAVA并发核心知识点
#Java#



Java

极客星球
微信扫码加入星球



知识星球



大师兄|荣哥
2小时前

收进专栏

并发知识图谱#Java# #cpp# #Go#Linux内核#Go#



Java cpp Go Linux内核



Apricity、大师兄|荣哥 觉得很赞

查看详情