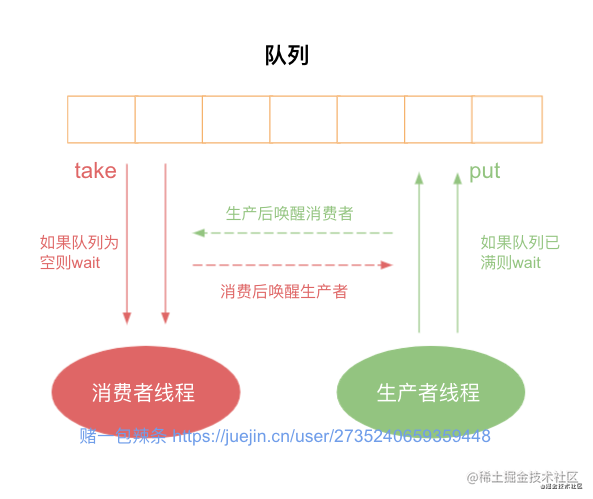
### 1.“生产者-消费者”模型

“生产者-消费者”模型是一个典型的线程协作通信的例子。在这一模型中有两类角色，即若干个生产者线程和若干个消费者线程。生产者线程负责提交用户请求，消费者线程负责处理生产者提交的请求。很多情况下，生产者与消费者不能够达到一定的平衡，即有时候生产者生产的速度过快，消费之来不及消费；而有时候可能是消费者过于旺盛，生产者来不及生产。在此情况下就需要一个生产者与消费者共享的内存缓存区来平衡二者的协作。生产者与消费者之间通过共享内存缓存区进行通信，从而平衡生产者与消费者线程，并将生产者和消费者解耦。如下图所示：



当队列容器中没有商品的时候，就需要让消费者处于等待状态，而当容器满了之后就需要生产者处于等待状态。而消费者每消费一个商品，又会通知正在等待的生产者可以进行生产了；当生产则生产一个商品，也会通知正在等待的消费者可以消费了。

2.使用synchronized实现“生产者-消费者”模型

了解了“生产者-消费者”模型后，我们尝试使用synchronized关键字结合wait()与notifyAll()方法来实现一个”生产者-消费者“模型的例子。

我们选一个比较经典的生产面包的例子来看，首先需要一个面包容器类，容器类中有放入面包和取出面包两个操作，代码如下：

public class BreadContainer {

LinkedList<Bread> list = new LinkedList<>();

// 容器容量

private final static int CAPACITY = 10;

/\*\*

\* 放入面包

\*/

public synchronized void put(Bread bread) {

while (list.size() == CAPACITY) {

try {

// 如果容器已满，则阻塞生产者线程

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

list.add(bread);

// 面包生产成功后通知消费者线程

notify();

System.out.println(Thread.currentThread().getName() + " product a bread" + bread.toString() + " size = " + list.size());

}

/\*\*

\* 取出面包

\*/

public synchronized Bread take() {

while (list.isEmpty()) {

try {

// 如果容器为空，则阻塞消费者线程

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

Bread bread = list.removeFirst();

// 消费后通知生产者生产面包

notify();

System.out.println("Consumer " + Thread.currentThread().getName() + " consume a bread" + bread.toString() + " size = " + list.size());

return bread;

}

}

在上述代码的put方法会将生产好的面包放入到容器中。如果容器已经满了，那么需要阻塞生产者线程来停止生产，当生产者成功将面包放入容器后则需要尝试唤醒等待中的消费者线程进行消费。

而take方法则是取出面包的操作，当容器为空，则阻塞消费者线程，让其进行等待，如果成功消费面包后则通知生产者开始生产。

接下来生产者与消费者的实现就比较简单了，代码如下：

// 生产者

public class Producer {

private final BreadContainer container;

public Producer() {

container = new BreadContainer();

}

public BreadContainer getContainer() {

return container;

}

// 生产者生产面包

public void makeBread() {

container.put(new Bread());

}

}

// 消费者

public class Consumer {

BreadContainer container;

public Consumer(BreadContainer container){

this.container = container;

}

public void takeBread() {

for (; ; ) {

Bread bread = container.take();

bread.eat();

}

}

}

接下来在测试代码中，同时开启多个生产者线程与多个消费者线程

public static void main(String[] args) {

// 实例化生产者

Producer producer = new Producer();

// 实例化消费者

Consumer consumer = new Consumer(producer.getContainer());

// 开启生产者线程

new Thread(() -> {

for (int i = 0; i < 100000; i++) {

producer.makeBread();

}

}).start();

// 消费者在主线程消费

consumer.takeBread();

}

此时运行main方法，生成者与消费者线程就可以很好的协同工作了。

注意，在main方法中我们实例化了一个BreadContainer对象，上边Flag处说的synchronized锁的对象即为这个container，调用的wait和notify方法也是container实例的方法。到这里不知道你是否会有疑问，究竟container的wait和notify方法对象成做了什么能让线程阻塞和唤醒呢？被阻塞的线程放到哪里去了？为什么要调用container对象中的wait和notify方法？如果换成调用其他对象的wait和notify是否可行呢？

调用wait方法的线程会被加入到一个\_WaitSet集合中，并会将线程挂起。但是，这里要再次强调一下\_WaitSet与\_EntryList这两个集合。\_EntryList集合中存放的是没有抢到锁，而被阻塞的线程，而\_WaitSet集合中存放的是调用了wait方法后，处于等待状态的线程。\*\*

想要证明上述的结论，就需要我们来看下wait和notify/notifyAll到底做了什么。

我们看下Object中wait、notify、notifyAll三个方法的实现

public class Object {

public final native void notify();

public final native void notifyAll();

public final void wait() throws InterruptedException {

wait(0L);

}

public final native void wait(long timeoutMillis) throws InterruptedException;

}

这几个方法都是native方法，也就是说这几个方法都是在虚拟机中使用C/C++实现的。既然如此，不妨扒一扒虚拟机的代码来一探究竟，毕竟口说无凭。

### 1.虚拟机对wait的实现

承接上一节中”生产者-消费者“模型的代码来分析，当生产者线程往容器里边放面包的时候发现容器已经满了，则调用wait方法，那么此时这个线程就会释放锁并进入到阻塞状态。

Object 中 wait 方法的实现是在 [objectMonitor.cpp](https://link.juejin.cn?target=https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.cpp" \o "https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.cpp" \t "/Users/zpw/Documents\\x/_blank) 中的 ObjectMonitor::wait(jlong millis, bool interruptible, TRAPS)函数中,ObjectMonitor::wait中的核心相关代码如下：

void ObjectMonitor::wait(jlong millis, bool interruptible, TRAPS) {

// ...省略其他代码

// 当前线程

Thread \* const Self = THREAD ;

// 将线程封装成ObjectWaiter

ObjectWaiter node(Self);

// 标记为Wait状态

node.TState = ObjectWaiter::TS\_WAIT ;

Self->\_ParkEvent->reset() ;

Thread::SpinAcquire (&\_WaitSetLock, "WaitSet - add") ;

// 调用 AddWaiter 方法将线程加入到等待队列中

AddWaiter (&node) ;

Thread::SpinRelease (&\_WaitSetLock) ;

// ...

// 释放 monitor 锁,并将自己挂起

exit (true, Self) ;

}

可以看到，调用 wait 函数后，线程被封装成了一个 ObjectWaiter 对象，并通过AddWaiter 函数将线程加入到等待队列中,先来看下 AddWaiter 函数的代码：

inline void ObjectMonitor::AddWaiter(ObjectWaiter\* node) {

// 如果 \_WaitSet 还没有初始化，先初始化 \_WaitSet

if (\_WaitSet == NULL) {

// 初始化 \_WaitSet 的头结点,此时只有一个node元素

\_WaitSet = node;

// 可以看出ObjectWaiter是一个双向链表，这里将node的首尾相连，说明\_WaitSet是一个循环链表

node->\_prev = node;

node->\_next = node;

} else {

// \_WaitSet 的头结点

ObjectWaiter\* head = \_WaitSet ;

// 环形链表头结点的prev就是尾结点

ObjectWaiter\* tail = head->\_prev;

assert(tail->\_next == head, "invariant check");

// 将node插入到\_WaitSet的尾结点中

tail->\_next = node;

head->\_prev = node;

node->\_next = head;

node->\_prev = tail;

}

}

AddWaiter函数的实现其实比较简单，会初始化一个\_WaitSet链表，并将node插入到\_WaitSet的队尾，从代码中也可以看出这个 \_WaitSet 链表是一个循环的双向链表。

完成线程的插入队列操作后，继续调用 exit 函数来释放 monito r锁,并挂起自己。关于这个方法，后边还会涉及到，源码后边再看。

2.虚拟机对notify的实现

在生产者生产完面包后则会调用notifyAll来唤醒消费者线程。notifyAll 方法会唤醒所有线程，而 notify 只会唤醒一个线程。此处我们以notify为例来看[objectMonitor.cpp](https://link.juejin.cn?target=https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.cpp" \o "https://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/782f3b88b5ba/src/share/vm/runtime/objectMonitor.cpp" \t "/Users/zpw/Documents\\x/_blank)中 notify 函数是如何唤醒线程的。

void ObjectMonitor::notify(TRAPS) {

int Policy = Knob\_MoveNotifyee ;

// DequeueWaiter是一个函数，会返回 \_WaitSet 的头结点

ObjectWaiter \* iterator = DequeueWaiter() ;

if (iterator != NULL) {

// 将阻塞队列赋值给 List

ObjectWaiter \* List = \_EntryList ;

// 根据策略执行不同的逻辑，Policy默认值为2

if (Policy == 0) { // prepend to EntryList

// ...

} else if (Policy == 1) { // append to EntryList

// ...

} else if (Policy == 2) { // prepend to cxq

// prepend to cxq

if (List == NULL) {

// iterator 的前驱与后继节点置空

iterator->\_next = iterator->\_prev = NULL ;

// \_EntryList指向这个节点，说明节点已被加入阻塞队列，等待获取锁

\_EntryList = iterator ;

} else {

iterator->TState = ObjectWaiter::TS\_CXQ ;

for (;;) { // 通过CAS将iterator插入到 \_cxq 队列

ObjectWaiter \* Front = \_cxq ;

iterator->\_next = Front ;

if (Atomic::cmpxchg\_ptr (iterator, &\_cxq, Front) == Front) {

break ;

}

}

}

} else if (Policy == 3) { // append to cxq

// ...

}

}

// ...

}

在 notify 函数中首先调用了DequeueWaiter 函数， DequeueWaiter 函数的作用是取出\_WaitSet链表的头结点，代码如下：

inline ObjectWaiter\* ObjectMonitor::DequeueWaiter() {

// dequeue the very first waiter

ObjectWaiter\* waiter = \_WaitSet;

if (waiter) {

DequeueSpecificWaiter(waiter);

}

return waiter;

}

// 将头结点 从队列中断开

inline void ObjectMonitor::DequeueSpecificWaiter(ObjectWaiter\* node) {

// ...

ObjectWaiter\* next = node->\_next;

if (next == node) {

// 此时，队列中只有一个元素，因此取出后，队列就是NULL了

\_WaitSet = NULL;

} else {

ObjectWaiter\* prev = node->\_prev;

// 这一操作就是将 node 从队列移除，并重新连接队列

next->\_prev = prev;

prev->\_next = next;

if (\_WaitSet == node) {

\_WaitSet = next;

}

}

// 将 node 的前驱节点与后继节点置空

node->\_next = NULL;

node->\_prev = NULL;

}

可以看到，DequeueWaiter 函数中又调用了 DequeueSpecificWaiter 函数，在这个函数中，如果队列只有一个节点，则将\_WaitSet置空，即取出头结点后，队列中没有元素了。如果有多个节点，那么 会将头结点从队列中取出，并重新拼接好 \_WaitSet 队列。然后将取出的这个节点的前驱节点和后继节点置空。

notify 函数接下来的代码判断如果 iterator 不为 NULL 说明存在等待状态的线程，需要将这个等待的线程转入阻塞线程的队列中去。接下来根据 Policy 来执行不同的逻辑，Policy 默认值为2，所以这里只关注默认情况情况。即当Policy为2时，接着将 \_EntryList赋值给List，如果List等于NULL，说明此时没有阻塞状态的线程。那么就将 \_EntryList 指向 iterator。标志着这个等待中的线程进入了阻塞状态，并且能够获取锁了，但此时线程还未被唤醒。如果List等于NULL，那么就通过CAS将等待状态的线程移入到了\_cxq 队列，\_cxq队列只是一个临时队列，在后边exit函数中最终还是会被移入\_EntryList中。这里一定要注意区分阻塞状态与等待状态，以及等待队列和阻塞队列。

3.虚拟机的exist函数

可见notify函数中只是对线程进行了队列转移，并没有被实际唤醒。而实际唤醒线程的操作就是在本章第1小节中已经提到的exist中实现的。只不过此时exist函数的调用时机是在虚拟机读取到 monitorexist 指令之后。看下简化后的 exit 函数代码，如下：

void ObjectMonitor::exit(bool not\_suspended, TRAPS) {

// ...

// 这里是一个死循环

for (;;) {

ObjectWaiter \* w = NULL ;

// QMode默认值为0

int QMode = Knob\_QMode ;

if (QMode == 2 && \_cxq != NULL) {

// ... 这里从\_cxq队列取头结点并唤醒，无关省略。

return ;

}

// ...

w = \_EntryList ;

// 先查看\_EntryList是否为空

if (w != NULL) {

// \_EntryList不为空，通过ExitEpilog函数唤醒\_EntryList队列的头结点

ExitEpilog (Self, w) ;

return ;

}

// 到这里说明\_EntryList为空，将则将 w 指向 \_cxq

w = \_cxq ;

// \_cxq 是 NULL 说明没有等待状态的线程需要唤醒，则继续执行循环

if (w == NULL) continue ;

// ...

// 走到这说明有处于等待状态，需要唤醒的线程

if (QMode == 1) {

// ...

} else {

// 如果走到此处说明\_cxq队列不为空

// QMode == 0 or QMode == 2

// 此时\_EntryList队列是空，将\_EntryList指向\_cxq队列

\_EntryList = w ;

ObjectWaiter \* q = NULL ;

ObjectWaiter \* p ;

// 将单向链表变成双向环链表

for (p = w ; p != NULL ; p = p->\_next) {

guarantee (p->TState == ObjectWaiter::TS\_CXQ, "Invariant") ;

p->TState = ObjectWaiter::TS\_ENTER ;

p->\_prev = q ;

q = p ;

}

}

if (\_succ != NULL) continue;

w = \_EntryList ;

if (w != NULL) {

guarantee (w->TState == ObjectWaiter::TS\_ENTER, "invariant") ;

// 唤醒\_EntryList的头结点

ExitEpilog (Self, w) ;

return ;

}

}

// 释放锁并唤醒线程

void ObjectMonitor::ExitEpilog (Thread \* Self, ObjectWaiter \* Wakee) {

assert (\_owner == Self, "invariant") ;

// Exit protocol:

// 1. ST \_succ = wakee

// 2. membar #loadstore|#storestore;

// 2. ST \_owner = NULL

// 3. unpark(wakee)

\_succ = Knob\_SuccEnabled ? Wakee->\_thread : NULL ;

ParkEvent \* Trigger = Wakee->\_event ;

Wakee = NULL ;

// 释放锁

OrderAccess::release\_store\_ptr (&\_owner, NULL) ;

OrderAccess::fence() ; // ST \_owner vs LD in unpark()

// 唤醒线程

Trigger->unpark() ;

//...

}

exist函数的代码比较繁杂，这里做了简化，由于QMode默认值是0，因此只讨论这种情况。

首先，如果 \_EntryList 不为NULL，那么直接调用 ExitEpilog 函数从 \_EntryList中取出头结点并唤醒线程；

如果 \_EntryList 为NULL,但是 \_cxq 队列不为 NULL，说明有等待状态的线程被notify了，但是还没真正的被唤醒，那么将 \_cxq队列中的所有元素移入\_EntryList队列中，并将其改造成一个双向链表。然后通过 ExitEpilog 唤醒\_EntryList的头结点。

Java代码中的 synchronized 关键字通过编译器编译成字节码的monitorenter/monitorexist指令，当虚拟机执行到相关指令后则会调用虚拟机底层相关的函数，进行拿锁和释放锁的操作。而由于锁对象Object关联了monitor对象，故可以调用这个Object对象中的 wait 和 notify/notifyAll 方法来阻塞和唤醒线程。而这两个方法亦是调用了虚拟机底层的相关函数，wait 函数会将线程封装成 WaitObject 并将其插入到等待队列中，而notify/notifyAll 则会将线程从等待队列中取出并转移到\_EntryList队列或者转移到\_cxq队列，等到持有锁的线程执行完毕并读取到 monitorexist 指令后调用了虚拟机的 exist 函数来释放锁并唤醒\_EntryList 队列或者\_cxq队列中的线程。

synchronized 锁的这种等待与唤醒机制很显然有一个弊端。仍然以”生产者-消费者“模型为例，由于生产者线程和消费者线程都会被加入到同一个WaitSet队列中，通过 notifyAll 方法并不能精确的控制唤醒哪一类线程。