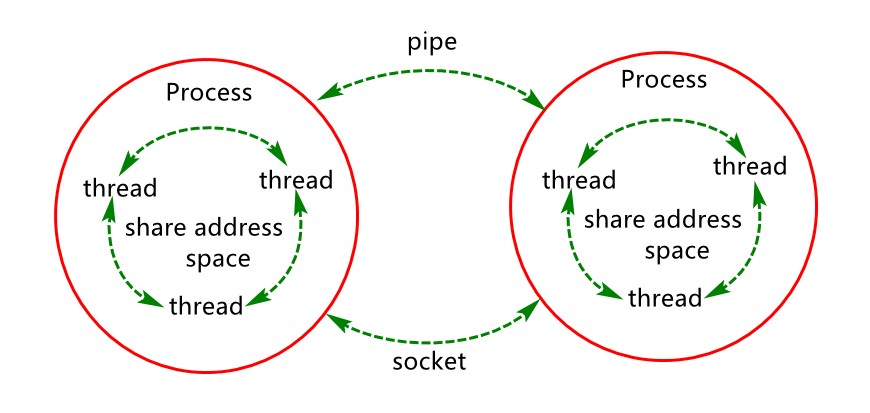
JAVA 核心基础增强线程应用加强

1. [线程通讯与进程通讯应用基础 1-1](#_bookmark0)
   1. [如何理解进程与线程通讯？ 1-1](#_bookmark1)
   2. [如何实现进程内部线程之间的通讯？ 1-2](#_bookmark2)
      1. [基于 wait/nofity/notifyall 实现 1-2](#_bookmark3)
      2. [基于 Condition 实现 1-5](#_bookmark4)
   3. [终端消息模型架构分析及实现？ 1-7](#_bookmark5)
2. [进程间通讯方式实现进阶分析 2-7](#_bookmark6)
   1. [如何实现进程之间间通讯（IPC）？ 2-7](#_bookmark7)
      1. [基于 socket 实现进程间通讯？ 2-8](#_bookmark8)
3. [课后练习与加强 3-9](#_bookmark9)
   1. [线程同步应用练习 3-9](#_bookmark10)
   2. [线程通讯练习 3-9](#_bookmark11)

# 进程线程通讯应用基础

## 如何理解进程通讯与线程通讯？

### 线程通讯：java 中的多线程通讯主要是共享内存（变量）等方式。进程通讯：java 中进程通讯（IPC）主要是 Socket，MQ 等。



## 如何实现进程内部线程之间的通讯？

#### 基于wait/nofity/notifyall 实现

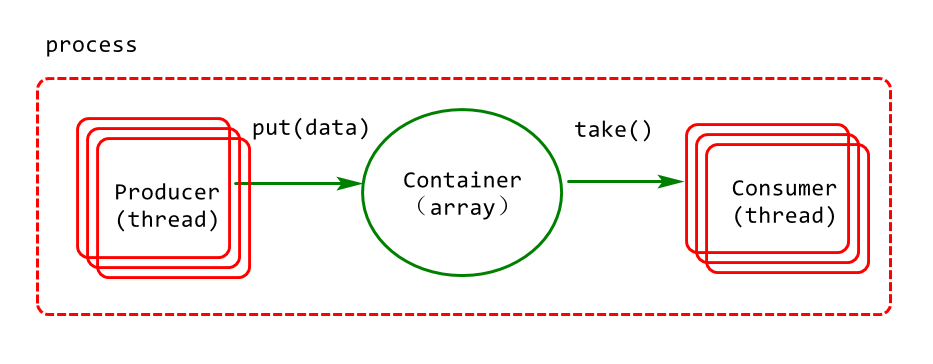
1. wait()/notify()/notifyall（）方法定义说明：
2. Wait:阻塞正在使用监视器对象的线程，同时释放监视器对象
3. notify: 唤醒在监视器对象上等待的单个线程，但不释放监视器对象，此时调用该方法的代码继续执行，直到执行结束才释放对象锁
4. notifyAll: 唤醒在监视器对象上等待的所有线程，但不释放监视器对象，此时调用该方法的代码继续执行，直到执行结束才释放对象锁
5. wait()/notify()/notifyall（）方法应用说明
6. 这些方法必须应用在同步代码块或同步方法中
7. 这些方法必须由监视器对象（对象锁）调用

说明：使用 wait/notify/notifyAll 的作用一般是为了避免轮询带来的性能损失。

1. wait()/notify()/notifyall()应用案例实现：

手动实现阻塞式队列，并基于 wait()/notifyAll()方法实现实现线程在队列上的通讯。

案例：现有一生产者消费者模型，生产者和消费者并发操作容器对象。



代码实现：实现一线程安全的容器类(基于数组实现一个阻塞式队列)

/\*\*

* 有界消息队列：用于存取消息
* 1)数据结构：数组(线性结构)
* 2)具体算法：FIFO(先进先出)-First in First out

\*/

**public class** BlockContainer<T> {//类泛型

/\*\*用于存储数据的数组\*/

**private** Object[] array;

/\*\*记录有效元素个数\*/

**private int** size;

**public** BlockContainer () {

**this**(16);//this(参数列表)表示调用本类指定参数的构造函数

}

**public** BlockContainer (**int** cap) {

array=**new** Object[cap];//每个元素默认值为null

}

}

向容器添加put方法，用于放数据。

/\*\*

* 生产者线程通过put方法向容器放数据
* 数据永远放在size位置
* 说明：实例方法内部的this永远指向
* 调用此方法的当前对象(当前实例)
* 注意：静态方法中没有this，this只能
* 应用在实例方法，构造方法，实例代码块中

\*/

**public synchronized void** put(T t){//同步锁：this

//1.判定容器是否已满，满了则等待

**while**(size==array.length)

**try**{**this**.wait();}**catch**(Exception e){}

//2.放数据array[size]=t;

//3.有效元素个数加1 size++;

//4.通知消费者取数据

**this**.notifyAll();

}

向容器类添加take方法，用于从容器取数据。

/\*\*

* 消费者通过此方法取数据
* 位置：永远取下标为0的位置的数据
* **@return**

\*/ @SuppressWarnings("unchecked") **public synchronized** T take(){

//1.判定容器是否为空，空则等待**while**(size==0) **try**{**this**.wait();}**catch**(Exception e){}

//2.取数据

Object obj=array[0];

//3.移动元素System.arraycopy(

array,//src 原数组

1, //srcPos 从哪个位置开始拷贝array, //dest 放到哪个数组0, //destPost 从哪个位置开始放size-1);//拷贝几个

//4.有效元素个数减1 size--;

//5.将size位置为null array[size]=**null**;

//6.通知生产者放数据

**this**.notifyAll();//通知具备相同锁对象正在wait线程**return** (T)obj;

}

#### 基于 Condition 实现

1. Condition 类定义说明

Condition 是一个用于多线程间协同的工具类，基于此类可以方便的对持有锁的线程进行阻塞或唤醒阻塞的线程。它的强大之处在于它可以为多个线程间建立不同的 Condition，通过 signal()/signalall()方法指定要唤醒的不同线程。

1. Condition 类应用说明
2. 基于 Lock 对象获取 Condition 对象
3. 基于 Condition 对象的 await()/signal()/signalall()方法实现线程阻塞或唤醒。
4. Condition 类对象的应用案例实现：

手动实现阻塞式队列，并基于 wait()/notifyAll()方法实现实现线程在队列上的通讯。

/\*\*

* 有界消息队列：用于存取消息
* 1)数据结构：数组(线性结构)
* 2)具体算法：FIFO(先进先出)-First in First out

\*/

**public class** BlockContainer<T> {//类泛型

/\*\*用于存储数据的数组\*/

**private** Object[] array;

/\*\*记录有效元素个数\*/

**private int** size;

**public** BlockContainer() {

**this**(16);//this(参数列表)表示调用本类指定参数的构造函数

}

**public** BlockContainer(**int** cap) {

array=**new** Object[cap];//每个元素默认值为null

}

//JDK1.5以后引入的可重入锁(相对于synchronized灵活性更好)

**private** ReentrantLock lock=**new** ReentrantLock(**true**);// true表示使用公平锁， 默认是非公平锁

**private** Condition producerCondition=lock.newCondition();//通讯条件

**private** Condition consumerCondition=lock.newCondition();//通讯条件

}

向容器中添加put方法，用于向容器放数据

/\*\*

* 生产者线程通过put方法向容器放数据
* 数据永远放在size位置
* 说明：实例方法内部的this永远指向
* 调用此方法的当前对象(当前实例)
* 注意：静态方法中没有this，this只能
* 应用在实例方法，构造方法，实例代码块中

\*/

**public void** put(T t){//同步锁：this System.out.println("put"); lock.lock();

**try**{

//1.判定容器是否已满，满了则等待

**while**(size==array.length)

//等效于Object类中的wait方法

**try**{producerCondition.await();}**catch**(Exception e){e.printStackTrace();}

//2.放数据array[size]=t;

//3.有效元素个数加1 size++;

//4.通知消费者取数据

consumerCondition.signalAll();//等效于object类中的notifyall()

}**finally**{ lock.unlock();

}

}

在容器类中添加take方法用于从容器取数据

/\*\*

* 消费者通过此方法取数据
* 位置：永远取下标为0的位置的数据
* **@return**

\*/ @SuppressWarnings("unchecked") **public** T take(){ System.out.println("take"); lock.lock();

**try**{

//1.判定容器是否为空，空则等待

**while**(size==0) **try**{consumerCondition.await();}**catch**(Exception e){}

//2.取数据

Object obj=array[0];

//3.移动元素System.arraycopy(

array,//src 原数组

1, //srcPos 从哪个位置开始拷贝array, //dest 放到哪个数组0, //destPost 从哪个位置开始放size-1);//拷贝几个

//4.有效元素个数减1 size--;

//5.将size位置为null array[size]=**null**;

//6.通知生产者放数据

producerCondition.signalAll();//通知具备相同锁对象正在wait线程

**return** (T)obj;

}**finally**{ lock.unlock();

}

}

## 终端消息模型架构分析及实现？

### 基本架构分析 代码分析实现：

1. **进程间通讯方式实现进阶分析**
   1. **如何实现进程之间间通讯？**

#### 基于 socket 实现进程间通讯？

基于 BIO 实现的简易 server 服务器

**public class** BioMainServer01 {

**private** Logger log=LoggerFactory.*getLogger*(BioMainServer01.**class**);

**private** ServerSocket server;

**private volatile boolean** isStop=**false**;

**private int** port;

**public** BioMainServer01(**int** port) {

**this**.port=port;

}

**public void** doStart()**throws** Exception { server=**new** ServerSocket(port); **while**(!isStop) {

Socket socket=server.accept(); log.info("client connect"); doService(socket);

}

server.close();

}

**public void** doService(Socket socket) **throws** Exception{ InputStream in=socket.getInputStream();

**byte**[] buf=**new byte**[1024]; **int** len=-1;

**while**((len=in.read(buf))!=-1) {

String content=**new** String(buf,0,len); log.info("client say {}", content);

}

in.close(); socket.close();

}

**public void** doStop() { isStop=**false**;

}

**public static void** main(String[] args)**throws** Exception { BioMainServer01 server=**new** BioMainServer01(9999); server.doStart();

}

}

启动服务，然后打开浏览器进行访问或者通过如下客户端端访问

**public class** BioMainClient {

**public static void** main(String[] args) **throws** Exception{ Socket socket=**new** Socket();

socket.connect(**new** InetSocketAddress("127.0.0.1", 9999)); OutputStream out=socket.getOutputStream();

Scanner sc=**new** Scanner(System.***in***); System.***out***.println("client input:"); out.write(sc.nextLine().getBytes()); out.close();

sc.close(); socket.close();

}

}

1. **课后练习与加强**

## 线程同步应用练习

1. 基于链表结构实现一个线程安全的阻塞队列？

## 线程通讯练习

1. 基于 BIO 方式实现 Socket 跨进程通讯。