# 课程简介：

## 课程名称：Java 网络编程

## 课程代码：150905099

## 学分/学时：3学分/48学时（其中理论32学时、上机16学时）

## 教学目标：

通过本课程理论学习，使学生掌握IO流、套接字、服务器套接字、阻塞与非阻塞通信、数据报通信、对象的序列化与反序列化、反射机制、RMI框架及安全网络通信，使学生具备实现网络客户端和服务器端编程的能力。通过上机实践完成基本IO对象编程、Socket网络程序设计、多线程服务器、非阻塞服务器。

## 教学任务及学时分配：

1. 网络编程基础（2学时）
2. Java IO流编程（6学时）
3. Java 线程编程（6学时）
4. Java Socket（BIO）编程（6学时）
5. Java SocketChannel（NIO）编程（6学时）
6. Java RMI编程（6学时）

上机一：基本IO对象编程（2学时）

上机二：使用TCP协议的Socket网络程序设计（2学时）

上机三：多线程服务器（2学时）

上机四：使用UDP协议的Socket网络程序设计（2学时）

上机五：使用线程池的服务器 （2学时）

上机六：非阻塞服务器（2学时）

上机七：远程调用中反射机制的应用（2学时）

上机八：使用RMI完成远程调用（2学时）

## 考核及成绩评定方式：

平时成绩（50分）：考勤（10分），实验8个（40分）

期末考试（50分）：IO编程、线程编程、Socket编程、SocketChannel编程、RMI编程

## 参考书：

董相志等编著 Java网络编程案例教程 北京：清华大学出版社， 2017

Elliotte Rusty Harold著. 李帅 荆涛等译. Java网络编程（第4版）. 北京：中国电力出版社，2014.

# 第1章 网络编程基础

## 1.1 网络编程的概念：

### 计算机网络：

计算机之间通过传输介质、通信设施和网络通信协议互联，实现资源共享和数据传输。

### 网络编程本质：

0、1序列流如何在计算机之间流动，即如何使用程序完成计算机之间数据的传输。为了实现计算机之间的数据传输，C语言和Java语言都提供了一系列的接口，使得开发人员可以方便的实现。

### 网络基础知识：

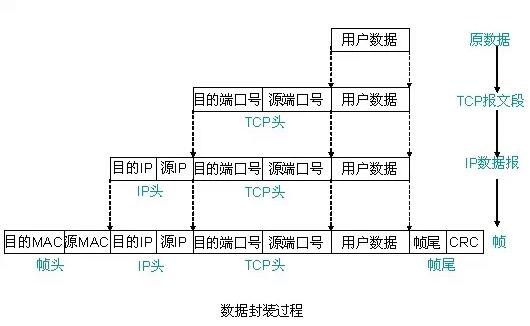
传输介质：有线（光纤、双绞线）、无线（电波）

传输设备：路由器、交换机、Hub、网卡

通讯协议：IEEE802.3（以太网协议），TCP/IP协议（互联网协议）等等

TCP/IP协议：将数据传输过程分为应用层协议（http、telnet、smtp、ftp等），传输层协议（TCP/UDP协议），网络层(IP协议)和数据链路层（IEEE802.3协议）

数据封装过程如下：



协议的真实含义：通信双方都能理解数据流，并能正确转换这些数据流。

应用层协议：应用程序双方如何交流信息

传输层：如何找到对方计算机上的哪个程序（端口号）

网络层：如何找到哪个网络的哪台计算机（IP地址+路由器或交换机）

链路层：如何在各种线路上传输数据（二进制流的传输）

## 1.2 学习网络编程的理由

1、Tomcat的工作原理

2、QQ是如何实现的

3、迅雷是如何实现多点下载的

4、无人驾驶如何实现

## 1.3 要求掌握内容（网络编程的主要内容）：

1、端口号：定位主机上的特定的应用

2、IP地址：如何准确地定位网络上一台或多台主机

3、传输协议：TCP或UDP（如何将数据传输到指定计算机上指定的程序来处理）

## 1.4 如何学好网络编程

提出自己的需求，编程实践。

# 第2章 Java IO

Javad IO流分为：

输入流：

字节流：(InputStream)

字符流：（Reader）

输出流：

字节流：(OutputStream)

字符流：（Writer）

## 2.1 InputStream

public abstract class InputStream extends Object implements Closeable

此抽象类是表示字节输入流的所有类的超类。

直接已知子类：

[AudioInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/javax/sound/sampled/AudioInputStream.html), [ByteArrayInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/ByteArrayInputStream.html), [FileInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/FileInputStream.html), [FilterInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/FilterInputStream.html), [InputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/org/omg/CORBA/portable/InputStream.html), [ObjectInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/ObjectInputStream.html), [PipedInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/PipedInputStream.html), [SequenceInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/SequenceInputStream.html), [StringBufferInputStream](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/StringBufferInputStream.html)

方法：

1、public abstract int read()throws [IOException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/IOException.html)

从输入流中读取数据的下一个字节。返回 0 到 255 范围内的 int 字节值。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。在输入数据可用、检测到流末尾或者抛出异常前，此方法一直阻塞。

子类必须提供此方法的实现。

返回：

下一个数据字节；如果到达流的末尾，则返回 -1。

抛出：

[IOException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/IOException.html) - 如果发生 I/O 错误。

2、public int read(byte[] b) throws [IOException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/IOException.html)

从输入流中读取一定数量的字节，并将其存储在缓冲区数组 b 中。以整数形式返回实际读取的字节数。在输入数据可用、检测到文件末尾或者抛出异常前，此方法一直阻塞。

如果 b 的长度为 0，则不读取任何字节并返回 0；否则，尝试读取至少一个字节。如果因为流位于文件末尾而没有可用的字节，则返回值 -1；否则，至少读取一个字节并将其存储在 b 中。

将读取的第一个字节存储在元素 b[0] 中，下一个存储在 b[1] 中，依次类推。读取的字节数最多等于 b 的长度。设 k 为实际读取的字节数；这些字节将存储在 b[0] 到 b[k-1] 的元素中，不影响 b[k] 到 b[b.length-1] 的元素。

返回：

读入缓冲区的总字节数；如果因为已经到达流末尾而不再有数据可用，则返回 -1。

抛出：

[IOException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/IOException.html) - 如果不是因为流位于文件末尾而无法读取第一个字节；如果输入流已关闭；如果发生其他 I/O 错误。

[NullPointerException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/lang/NullPointerException.html) - 如果 b 为 null。

3、public void close() throws [IOException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/IOException.html)

关闭此输入流并释放与该流关联的所有系统资源。

抛出：

[IOException](mk:@MSITStore:F:\jdk_api_1_6_zh_cn\JDK_API_1_6_zh_CN.CHM::/java/io/IOException.html) - 如果发生 I/O 错误。

## 2.2 OutputStream

查看API，掌握OutpuStream抽象类的直接子类及常用方法

## 2.3 案例1

使用基本字节流的各种读写方式完成一个较大文件的复制，比较各种读写方法的效率。便于以后编程使用

要求：1）用read()，write()完成，记录所用时间

2）用read(byte[] b)和write(byte[] b,int off,int len)完成，记录所用时间

代码框架：

public static void main(String[] args){

long start=System.currentTimeMillis();

Copy1();

long end= System.currentTimeMillis();

System.out.println(“方法1所用时间:”+(end-start)+”毫秒”);

}

static void Copy1(){

//1、创建字节输入流和输出流

//2、用read()和write()完成文件的读写

//3、关闭流

}

## 2.4 BufferedInputStream及BufferedOutputStream的使用

掌握缓冲字节流的概念：缓冲字节流是基本字节流的一个装饰类，读写效率高。

查看API，掌握构造方法和读写方法

## 2.5 案例2

用缓冲字节流完成案例1，记录需要的时间。

## 2.6 字符流（Reader、Writer）

字节流处理中文不是很方便。每次读一个字节一般会出问题，若用字节数组读，可能不出问题，但可能存在隐患。

字节流处理中文可能出现的问题：

用字节流读取一个含有中文的文件。比如文件中的内容是“hello中国”，用字节流读文件，并显示其内容会出现看不懂的情况，JDK1.1版本增加了字符流API。

Java提供了另一个类：Reader和Writer，用于读取字符，称为字符流，它是一个转换流，由字节流+编码表构成

编码表：

\* ASCII '0'----48 'A'----64 'a'----96 字符对应一个数值 0-127

\* ISO-8859-1: latin码表 一个字节 8位 0-255 扩展ASCII表

\* GB2312表：汉字编码表

\* GBK:汉字升级编码表

\* GB18030：GBK取代版

\* BIG5:繁体字编码表

\* Unicode编码表：国际编码表，每个字符用两个字节，java语言采用Unicode编码

\* UTF-8编码：变长编码，最短1个字节，最长三个字节，很好融合了ASCII,又兼顾其他语言字符

Reader的直接子类

InputStreamReader(InputStream is):采用默认编码读取字符，Windows默认编码是GBK，Linux默认编码是UTF-8，因此采用默认编码会导致程序在不同系统中出现问题。

InputStreamReader(InputStream is,String charsetName):采用指定编码读取字符（常用）

Reader的方法：

int read() 读一个字符 返回-1表示没读成功。问题：为啥不用char而用int？输出时还要强制转换为char，干嘛不直接返回char?

int read(char[] cbuf) 将字符读入数组 返回-1和字符个数，注意：Java中的字符采用的是Unicode编码（二字节编码）。

Writer的直接子类：

OutputStreamWriter(OutputStream out):

OutputStreamWriter(OutputStream out,String charsetName);(常用)

将int或char按默认或指定的字符编码写入输出流中。

注意：这里涉及三个编码，输入流的编码，Java内存编码，输出流的编码。Java内存中若用int，字符流的值是0~65535，字节流是0~255。Java中的char是unicode，它们之间的转换是否正确取决于源是否正确的读入。

字符流在网络编程中主要用于读取通信双方发送的字符数据。发送方可以将字符数据转换成字节流发送，也可以用字符流发送，接收方同样可以用字节流或字符流接收，但双方必须采用相同的编码。

通信双方的底层提供的都是字节流输入输出，若要用字符流输入输出，需要将字节流转换成字符流，转换中一般采用明确的编码，默认编码可能导致不同操作系统不兼容的问题。

## 2.7 BufferedReader及BufferedWriter(缓冲字符流)

将普通字符流装饰成缓冲字符流，实现高效输入输出。

BufferedReader的构造方法：

BufferedReader(Reader in) 创建一个使用默认大小输入缓冲区的缓冲字符输入流。

BufferedReader(Reader in, int sz) 创建一个使用指定大小输入缓冲区的缓冲字符输入流。

Reader的子类是InputStreamReader，InputStreamReader有FileReader子类。

将一个字节流封装成BufferedReader:

BufferedReader br=new BufferedReader(new InputStreamReader(InputStream in,”编码”));

将一个文件流封装为BufferedReader的用法有两种：

1. BufferedReader br=new BufferedReader(new InputStreamReader(new FileInputStream(“xxx”),”编码”));
2. BufferedReader br=new BufferedReader(new FileReader(“文件”);

FileReader是InputStreamReader的子类，采用的是默认字符编码和默认字节缓冲区大小，若要采用指定的编码需要使用第1种方法。

BufferedReader的方法：

Reader中的所有方法外还有：

public String readLine()

读取一个文本行。通过下列字符之一即可认为某行已终止：换行 ('\n')、回车 ('\r') 或回车后直接跟着换行。

返回：

包含该行内容的字符串，不包含任何行终止符，如果已到达流末尾，则返回 null

由于读取到的String中不包括回车换行这些字符，因此在输出接收到的字符串后还需要输出换行。（不同系统的换行符是不同的，Linux是\n，Mac OS是\r，Windows是\r\n，BufferedWriter中提供newLine()方法可以在不同系统中兼容)

## 2.8 案例3

编写一个程序，用BufferedReader和BufferedWriter将键盘上输入的若干行内容写入一个文本文件，文件中字符编码要求用UTF-8编码。

# 第3章 线程

## 3.1 线程的概念

线程（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运行单位。一个线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每个线程并行执行不同的任务。

进程：当一个进程执行一个fork调用的时候，会创建出进程的一个新拷贝，新进程将拥有它自己的变量和它自己的PID。这个新进程的运行时间是独立的，它在执行时几乎完全独立于创建它的进程。

线程：在进程里面创建一个新线程的时候，新的执行线程会拥有自己的堆栈（因此也就有自己的局部变量），但要与它的创建者共享全局变量、文件描述符、信号处理器和当前的工作目录状态。

问题1、在网络编程中若程序有read()阻塞，后面的客户无法连接服务，要解决其他客户的访问问题，如何办？

问题2、Tomcat中多个客户访问同一个Servlet，该Servlet如何响应？

线程的本质：增加程序的执行线路。

## 3.2 线程的调度

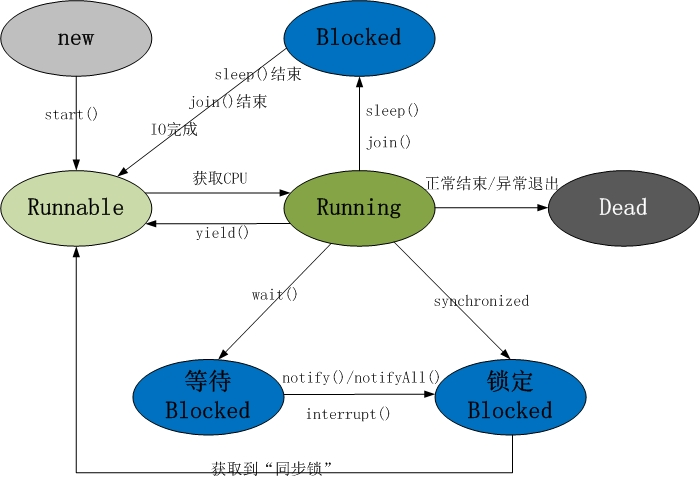
所有的Java虚拟机都有一个线程调度器，用来确定那个时刻运行那个线程。主要包含两种：抢占式线程调度和协作式线程调度。

1.抢占式线程调度，每个线程可能会有自己的优先级，但是优先及并不意味着高优先级的线程一定会被调度，而是由cup随机的选择，所谓抢占式的线程调度，就是说一个线程在执行自己的任务时，虽然任务还没有执行完，但是cpu会迫使它暂停，让其它线程占有cpu的使用权。

2.协作式线程调度，每个线程可以有自己的优先级，但优先级并不意味着高优先级的线程一定会被最先调度，而是由cpu时机选择的，所谓协作式的线程调度，就是说一个线程在执行自己的任务时，不允许被中途打断，一定等当前线程将任务执行完毕后（同步机制）才会释放对cpu的占有，其它线程才可以抢占该cpu。

二者的比较：抢占式线程调度不易发生饥饿现象（一个线程由于优先级太低，抢不到资源，不能被执行，称为饥饿），不易因为一个线程的问题而影响整个进程的执行，但是其频繁阻塞与调度，会造成系统资源的浪费。协作式的线程调度很容易因为一个线程的问题导致整个进程中其它线程饥饿。

## 3.3 线程的生命周期



Java线程具有五中基本状态

新建状态（New）：当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread();

就绪状态（Runnable）：当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

运行状态（Running）：当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就 绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

阻塞状态（Blocked）：处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；

2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；

3.其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期

## 3.4 线程的操作

### 3.4.1 线程的创建

三种方式创建线程

1. 实现Runnable接口的类
2. 继承Thread类
3. 实现Callable接口的类

#### 1、实现Runnable接口的类

步骤：

\* A:写一个类，实现Runnable接口

\* B:实现run()方法

\* C:创建该类对象

\* D:创建Thread对象，将C步创建的对象作为Thread的构造参数

\* E:执行Thread对象的start()方法，启动线程

Thread(Runnable target, String name) 构造一个带名字的线程对象

Thread(Runnable target) 构造一个默认名字的线程对象

同一个类对象创建多个线程，多个线程共享类中的成员变量

#### 2、继承Thread类

步骤：

\* A:写一个类，继承Thread类

\* B:重写run()方法

\* C:创建该类对象

\* D:执行该类对象的start()方法，启动线程

启动线程前，调用对象的setName()方法给线程赋名，或在创建对象时给出线程名。

MyThread mt1=new MyThread();

Mt1.setName(“线程名”);

或者：

MyThread mt1=new MyThread("线程1");

多个线程多个对象，多个线程可以共享同一个类对象成员变量，成员变量必须是静态的。

### 3.4.2 线程同步操作

需求：编写程序模拟多窗口售电影票的过程

要求完成3个窗口出售100张电影票。

通过该例引出线程安全问题：

判断是否会出现安全问题的标准：

\* A: 是否是多线程环境？

\* B: 是否有共享数据？

\* C: 操作由多条语句操作共享数据？

实例满足上面3个标准，必然会出现安全问题。

如何解决上面的问题呢？

A: B:不是我们能解决的，只能在C:上想办法解决。

能否将多条操作语句封装为一个整体，使它具有原子性，即在执行这几条语句时不能被其他线程抢走执行权。

Java提供了一个叫同步的机制可以坚决该问题，用法如下：

synchronized(对象){

需要同步的代码块；

}

A:对象是什么？可以随便创建一个对象试试

B: 需要同步的代码是哪些？就是共享数据操作的那些代码

同步解决了线程安全问题，但同步降低了程序执行效率，当线程很多时，每个线程都要判断同步是否上锁，线程未获得锁资源会进入阻塞状态，称为同步阻塞。

很耗费资源。

注意：锁必须是同一把锁。

1. 在代码块上加同步锁（任意对象都可以作为锁对象）
2. 在方法上加同步锁（普通方法上使用的锁是this，静态方法使用的锁对象是该类的class对象,比如SellTicket.class对象锁）

在jdk5中提供专门的锁对象

private Lock lock=new ReentrantLock(); //创建锁对象

try{

lock.lock(); //获取锁

if(tickets>0){

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+"正在出售第"+tickets--+"张票");

}

}finally{

lock.unlock();//释放锁

}

线程同步保证操作的原子性。

### 3.4.3 线程等待和唤醒

典型的线程等待发生在生产和消费线程

生产线程生产资源，消费线程消费资源，线程既涉及到同步锁又涉及到等待和唤醒操作

在Object类中，定义了wait(), notify()和notifyAll()等接口。wait()的作用是让当前线程进入等待状态，同时，wait()也会让当前线程释放它所持有的锁，让被唤醒的线程获得锁。（若仅有同步操作，是在代码块执行完后才释放同步锁）。notify()和notifyAll()的作用，则是唤醒当前对象上的等待线程；notify()是唤醒单个线程，而notifyAll()是唤醒所有的线程。

需求：通过设置线程（生产者）和获取线程（消费者）对同一个学生对象进行操作

\* 分析：

\* A:资源类 Student

\* B:设置线程 SetThread (生产者)

\* C:获取线程 GetThread (消费者)

\* D:测试类 StudentDemo

\* 问题1：每个线程都创建一个学生对象，这不是对同一个资源操作。

\* 应该在外界创建一个资源，将资源传递给线程

\* 问题2：出现数据混乱的情况

\* 原因：线程安全问题

\* 解决办法：加锁，一定是同一把锁。把外界传进来的学生对象作锁就可以了。

\* 问题3：若消费者线程先抢到CPU的执行权，而生产线程还没有设置好学生信息，消费者输出的数据会存在问题。

\* 若生产者线程先抢到CPU的执行权，会产生学生对象，若继续拥有执行权，则会继续生产，而前面生产的数据还没有消费，就被扔掉了。这不符合实际情况。

\* 解决办法：使用等待/唤醒机制

\* Object类提供了3个方法：wait()等待，notify()唤醒单个线程，notifyAll()唤醒所有线程

\* 这些方法必须通过锁对象调用，而锁对象可以是任意对象，因此由Object类提供这些方法可以用在任意对象上

\* 问题4：怎样判断学生对象是否已消费或未消费？即什么时候线程该等待？

### 3.4.4 实现Callable接口的线程

线程实现的第三种方式：实现Callable接口

\* 之前的多线程的两种实现中，不管是继承thread类还是实现runnable接口，都无法保证获取到线程的执行结果。

\* 通过实现Callable接口，并用Future可以来接收多线程的执行结果。

Future接口:

表示异步计算的结果，通过Future接口提供的方法，可以很方便的查询异步计算任务是否执行完成，

获取异步计算的结果，取消未执行的异步任务，或者中断异步任务的执行

Future接口主要包括5个方法

get（）方法可以获取当前任务结束后返回一个结果，如果调用时，工作还没有结束，则会阻塞线程，直到任务执行完毕

get（long timeout,TimeUnit unit）等待timeout的时间就会返回结果

cancel（boolean mayInterruptIfRunning）方法可以用来停止一个任务，

如果任务可以停止（通过mayInterruptIfRunning来进行判断），

则可以返回true,如果任务已经完成或者已经停止，或者这个任务无法停止，则会返回false.

isDone（）方法判断当前方法是否完成

isCancel（）方法判断当前方法是否取消

Future表示一个可能还没有完成的异步任务的结果，针对这个结果可以添加Callback

以便在任务执行成功或失败后作出相应的操作。

举个例子：比如去吃早点时，点了包子和凉菜，包子需要等3分钟，凉菜只需1分钟，如果是串行的一个执行， 在吃上早点的时候需要等待4分钟，但是因为你在等包子的时候，可以同时准备凉菜，所以在准备凉菜的过程中，可以同时准备包子，这样只需要等待3分钟。那Future这种模式就是后面这种执行模式。

演示这两种模式：（时间改为秒）

演示第一种串行执行方式：

需求场景：等待早餐过程中，包子需要3秒，凉菜需要1秒，普通的多线程需要四秒才能完成。先等凉菜，再等包子，

因为等凉菜时，普通多线程启动start()方法， 执行run()中具体方法时，没有返回结果，所以如果要等有返回结果，

必须是要1秒结束后才知道结果。

分析：

A:先实现两个线程类 BumThread和ColdDishThread，分别等待3秒和1秒

B:执行BumThread线程，完成后再执行ColdDishThread线程，要等待线程完成，用join方法控制，它可以使得线程之间的并行执行变为串行执行。

在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。

比如：程序在main线程中调用t1线程的join方法，则main线程放弃cpu控制权，并返回t1线程继续执行直到线程t1执行完毕。

C:计算时间

long start = System.currentTimeMillis();

// 等凉菜 -- 必须要等待返回的结果，所以要调用join方法

Thread t1 = new ColdDishThread();

t1.start();

t1.join();

// 等包子 -- 必须要等待返回的结果，所以要调用join方法

Thread t2 = new BumThread();

t2.start();

t2.join();

long end = System.currentTimeMillis();

System.out.println("准备完毕时间："+(end-start));

第二种异步执行方式：

需求场景：在准备凉菜的过程中， 可以同时准备包子，这样只需要等待3秒。

Java提供的Future接口就是异步执行方式。

Future接口的实现类FutureTask。

分析：

A:先实现两个线程类 BumThread和ColdDishThread，分别等待3秒和1秒

B:执行BumThread线程，完成后再执行ColdDishThread线程，要等待线程完成，用join方法控制

C:计算时间

long start=System.currentTimeMillis();

FutureTask<String> ft1=new FutureTask<String>(new BumThread());

new Thread(ft1).start();

FutureTask<String> ft2=new FutureTask<String>(new ColdDishThread());

new Thread(ft2).start();

System.out.println(ft1.get());

System.out.println(ft2.get());

long end = System.currentTimeMillis();

System.out.println("准备完毕时间："+(end-start));

### 3.4.5 线程池操作

线程池的好处：线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。

\*

\* 如何实现线程的代码呢?

\* A:创建一个线程池对象，控制要创建几个线程对象。

\* public static ExecutorService newFixedThreadPool(int nThreads)

\* B:这种线程池的线程可以执行：

\* 可以执行Runnable对象或者Callable对象代表的线程

\* C:调用如下方法即可

\* Future<?> submit(Runnable task)

\* <T> Future<T> submit(Callable<T> task)

\* D:结束线程池(可选)

线程池执行Runnable线程对象：

ExecutorService pool = Executors.newFixedThreadPool(2);

// 可以执行Runnable对象或者Callable对象代表的线程

pool.submit(new MyRunnable());

pool.submit(new MyRunnable());

// 结束线程池

pool.shutdown();

实现Callable接口的线程：

public class MyCallable implements Callable<Integer> {

private int number;

public MyCallable(int number) {

this.number = number;

}

@Override

public Integer call() throws Exception {

int sum = 0;

for (int x = 1; x <= number; x++) {

sum += x;

}

return sum;

}

}

线程池执行Callable线程对象：

ExecutorService pool = Executors.newFixedThreadPool(2);

// 可以执行Runnable对象或者Callable对象代表的线程

Future<Integer> f1 = pool.submit(new MyCallable(100));

Future<Integer> f2 = pool.submit(new MyCallable(200));

// V get()

Integer i1 = f1.get();

Integer i2 = f2.get();

System.out.println(i1);

System.out.println(i2);

// 结束线程池

pool.shutdown();

# 第4章 Java Socket

## 4.1 什么是Socket?

        Socket的英文原义是“孔”或“插座”。在网络编程中，网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket。

        Socket套接字是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

        Socket本质是编程接口(API)，对TCP/IP的封装，TCP/IP也要提供可供程序员做网络开发所用的接口，这就是Socket编程接口；HTTP是轿车，提供了封装或者显示数据的具体形式；Socket是发动机，提供了网络通信的能力。

## 4.2 Socket的原理

        Socket实质上提供了进程通信的端点。进程通信之前，双方首先必须各自创建一个端点，否则是没有办法建立联系并相互通信的。正如打电话之前，双方必须各自拥有一台电话机一样。

        套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认。

        1、服务器监听：是服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

        2、客户端请求：是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

        3、连接确认：是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

## 4.3 基于java的Socket网络编程实现

        Server端Listen监听某个端口是否有连接请求，Client端向Server 端发出连接请求，Server端向Client端发回Accept接受消息。这样一个连接就建立起来了。Server端和Client端都可以通过Send，Write等方法与对方通信。

        对于一个功能齐全的Socket，都要包含以下基本结构，其工作过程包含以下四个基本的步骤：

1. 创建Socket；

　 2、 打开连接到Socket的输入/出流；

        3、按照一定的协议对Socket进行读/写操作；

        4、关闭Socket。

### 4.3.1 基于TCP的socket实现

案例1：实现一个最简单的客户端和服务器端通信（服务器一对一）

需求：客户端发送一条消息，服务器接受消息，并将消息回送客户端

服务器端程序分析：

1. 创建服务器套接字，打开监听端口：ServerSocket ss=new ServerSocket(int port);
2. 用服务器套接字监听客户端的连接，创建服务器端的通信套接字:

Socket sc=ss.accept();

1. 创建通信套接字的输入输出流对象
2. 使用流的读写操作完成接收和发送客服端的消息
3. 释放资源

客户端程序分析：

1. 连接服务器，创建客户端通信套接字：Socket sc= new Socket("IP地址",port);
2. 创建通信套接字的输入输出流对象
3. 使用流的读写操作完成发送和接收服务器端的消息
4. 释放资源

案例2：修改服务器的程序，能接收多个客户端的连接（服务器一对多）

服务器端程序分析：

1. 创建服务器端套接字（监听套接字）,打开端口；
2. 循环监听客户端的连接
3. 处理连接
4. 释放资源

案例3：案例2的处理方式是一种阻塞模式，某客户端连接后若长时间未结束（输入输出阻塞），其他客户端无法连接，

解决方案：连接后，使用线程处理该客户端的输入输出，服务器端主线程可以接收其他客户端的连接

public class SocketServer {

public static void main(String[] args) {

try {

// 创建服务端监听socket

ServerSocket serverSocket = new ServerSocket(8088);

// 创建客户端通信socket

Socket socket = new Socket();

//循环监听等待客户端的连接

while(true){

// 监听客户端是否有连接

socket = serverSocket.accept();

//该方法是一个阻塞方法，没有连接则阻塞，程序不执行下一步操作

//创建线程处理某客户端的通信

ServerThread thread = new ServerThread(socket);

thread.start();

}

} catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

}

}

处理客户端通信过程的线程：

public class ServerThread extends Thread{

private Socket socket = null;

InetAddress address;

public ServerThread(Socket socket) {

this.socket = socket;

address=socket.getInetAddress();

}

@Override

public void run() {

InputStream is=null;

InputStreamReader isr=null;

BufferedReader br=null;

OutputStream os=null;

PrintWriter pw=null;

try {

is = socket.getInputStream();

isr = new InputStreamReader(is);

br = new BufferedReader(isr);

os = socket.getOutputStream();

pw = new PrintWriter(os);

String info = null;

while((info=br.readLine())!=null){

if(info.equals(“bye”){

break;

}

System.out.println("客户端“+ address.getHostAddress()+”发送来的信息："+info);

//将收到的信息回显给客户端

pw.write(“服务器收到的信息是：“+info);

pw.flush();

}

socket.shutdownInput();

} catch (Exception e) {

} finally{

//关闭资源

try {

if(pw!=null)

pw.close();

if(os!=null)

os.close();

if(br!=null)

br.close();

if(isr!=null)

isr.close();

if(is!=null)

is.close();

if(socket!=null)

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

public class SocketClient {

public static void main(String[] args) throws InterruptedException {

try {

// 和服务器创建连接

Socket socket = new Socket("IP地址",8088);

// 要发送给服务器的信息

OutputStream os = socket.getOutputStream();

PrintWriter pw = new PrintWriter(os);

pw.write("客户端发送信息");

pw.flush();

socket.shutdownOutput();

// 从服务器接收的信息

InputStream is = socket.getInputStream();

BufferedReader br = new BufferedReader(new InputStreamReader(is));

String info = null;

while((info = br.readLine())!=null){

System.out.println("我是客户端，服务器返回信息："+info);

}

br.close();

is.close();

os.close();

pw.close();

socket.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

### 4.3.2 基于UDP的DatagramSocket编程

/\*

\* UDP协议接收数据：

\* A:创建接收端Socket对象

\* B:创建一个数据包(接收容器)

\* C:调用Socket对象的接收方法接收数据

\* D:解析数据包，并显示在控制台

\* E:释放资源

\*/

public class ReceiveDemo {

public static void main(String[] args) throws IOException {

// 创建接收端Socket对象

// DatagramSocket(int port)

DatagramSocket ds = new DatagramSocket(10086);

// 创建一个数据包(接收容器)

// DatagramPacket(byte[] buf, int length)

byte[] bys = new byte[1024];

int length = bys.length;

DatagramPacket dp = new DatagramPacket(bys, length);

// 调用Socket对象的接收方法接收数据

// public void receive(DatagramPacket p)

ds.receive(dp); // 阻塞式

// 解析数据包，并显示在控制台

// 获取对方的ip

// public InetAddress getAddress()

InetAddress address = dp.getAddress();

String ip = address.getHostAddress();

// public byte[] getData():获取数据缓冲区

// public int getLength():获取数据的实际长度

byte[] bys2 = dp.getData();

int len = dp.getLength();

String s = new String(bys2, 0, len);

System.out.println(ip + "传递的数据是:" + s);

// 释放资源

ds.close();

}

}

/\*

\* UDP协议发送数据：

\* A:创建发送端Socket对象

\* B:创建数据，并把数据打包

\* C:调用Socket对象的发送方法发送数据包

\* D:释放资源

\*/

public class SendDemo {

public static void main(String[] args) throws IOException {

// 创建发送端Socket对象

// DatagramSocket()

DatagramSocket ds = new DatagramSocket();

// 创建数据，并把数据打包

// DatagramPacket(byte[] buf, int length, InetAddress address, int port)

// 创建数据

byte[] bys = "hello,udp,我来了".getBytes();

// 长度

int length = bys.length;

// IP地址对象

InetAddress address = InetAddress.getByName("192.168.12.92");

// 端口

int port = 10086;

DatagramPacket dp = new DatagramPacket(bys, length, address, port);

// 调用Socket对象的发送方法发送数据包

// public void send(DatagramPacket p)

ds.send(dp);

// 释放资源

ds.close();

}

}