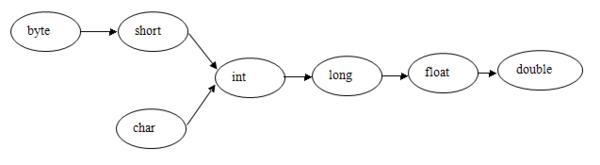
**Java 学习笔记**

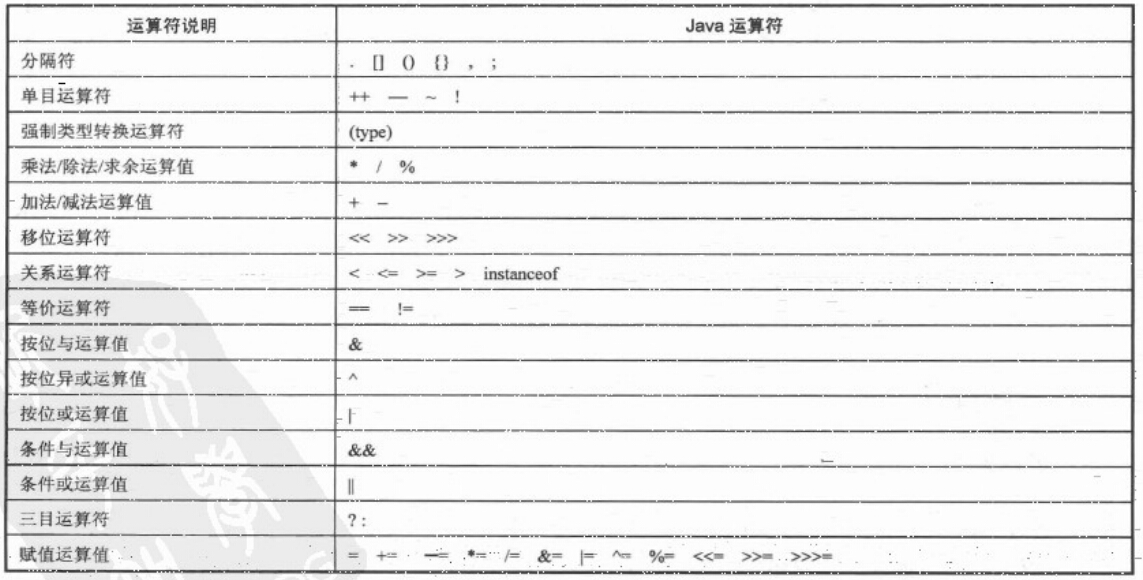
1. **数据类型和运算符**
2. 基本数据类型：byte,short,int,long,char ,float ,double,boolean

byte 占8位，short 16位，int 32位，long 64位, char 占2个字节

基本类型自动转换规则：



1. 运算符：
2. 基本类型的变量、值不能和引用类型的变量、值使用==进行比较运算
3. boolean类型的变量、值不能与其他任意类型的变量、值使用==比较运算
4. 如果两个引用类型之间没有父子或继承关系，那么他们的变量之间不能使用==进行比较运算
5. 运算符优先级

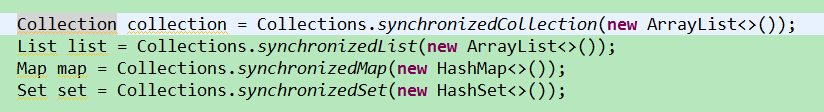


1. **数组**
2. Java中，数组是一种数据类型，它本身也是一种引用类型，int是一个基本类型，但是int[]就是一种引用类型
3. 数组声明定义有两种方式：（1）type[] arrayName （2）type arrayName[], 在实际开发中，建议尽量使用第（1）种方式，因为这种方式可读性比较好，而且更符合数据类型的定义规范。
4. 内存中数组：实际的数组对象是被存储在堆内存中的，如果数组引用变量是个局部变量，那么它就被存储在栈内存区。
5. **面向对象**
6. Java中，所有类都是引用数据类型
7. Static修饰：有static修饰的成员表示该成员属于类本身，没有的话，表示属于类的实例对象
8. 对象和引用：Person p = new Person（），这句代码产生了两个东西：一个是p变量，一个是Person对象，而p变量是个引用，也就是C++里的指针。
9. 传参：java中，只有一种传参形式就是：值传递。
10. 非引用类型传递的不改变实参的内容
11. 引用类型传参会改变原来的实参的内容，这是因为，他也是拷贝了一个东西，而这个东西是个引用，所以才会改变原来实参的内容。
12. Final修饰：
13. 当final修饰基本类型变量的时候，不能对基本类型变量重新赋值，因此基本类型变量不能被改变
14. 当final修饰引用类型变量时候，只能保证这个引用类型变量所引用的地址不会改变，即一直引用同一个对象，但是引用说指的对象内容是可以改变的。
15. Final方法：主要用于该方法不希望被子类重写。
16. Final类：主要用于该类不希望有子类。
17. abstract修饰:
18. 有抽象方法的类只能定义为抽象类，但是抽象类可以没有抽象方法
19. 抽象类可以包含抽象方法，但是抽象类不能实例化。
20. static和abstract不能同时修饰一个方法
21. private和abst不能同时修饰一个方法
22. 接口（Interface）:
23. java中，接口中的方法全部都是抽象方法
24. 接口中的常量都是 默认使用 public static final来修饰的
25. 接口和类不一样，支持多继承，但是只能继承接口，不能继承类。
26. 抽象类和接口的区别：
27. 接口里只能包含抽象方法，而抽象类可以包含普通方法，
28. 接口不能定义静态方法，而抽象类可以
29. 接口里只能定义静态常量的属性，抽象类可以定义普通属性和静态常量属性
30. 接口没有构造方法，而抽象类有，但是不是用来创建实例的，而是让其之类调用这些构造方法类完成属于抽象类的初始化操作。
31. **集合**
32. HashSet：
33. 不能保证元素的排列顺序，顺序可能发生变化
34. HashSet不是同步的，不是线程安全的，即多个线程同时访问一个HashSet时，必须通过代码来保证其他同步。
35. 集合中的元素可以为null
36. LinkedHashSet：（HashSet的子类）：

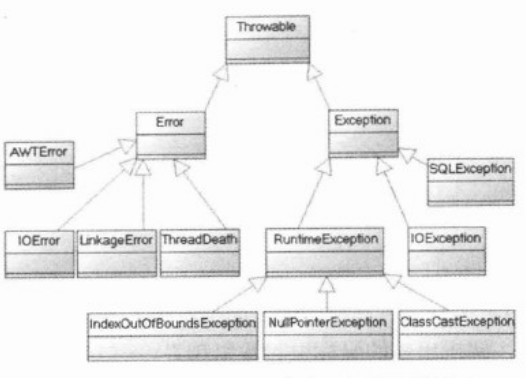
该类对于普通的插入，删除操作比HashSet略慢些，因为维护插入顺序链表需要额外的开销；不过，因为有了链表，遍历操作会比HashSet快。

1. List集合是一个元素有序的、可重复的集合。通过索引来访问元素。ArrayList是线程不安全的。Vector是线程安全的。
2. LinkedList:LinkedList是一个List，还实现了Deque接口，因而它可以双端队列来使用。也可以当作“栈”来使用。
3. HashMap与Hashtable的区别：
4. Hashtable是个线程安全的Map实现，而HashMap是线程不安全的
5. Hashtable不允许使用null作为key和value，因为这样会发生空指针异常；但HashMap可以使用null作为key或value。
6. 使用Collections工具类同步非线程安全的集合类对象：

如创建4个线程同步的集合对象：



1. **泛型**
2. 泛型本质上就是参数化类型，和C++中的模板类似。也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。
3. 泛型的好处：
4. 类型安全
5. 消除强制类型转换
6. 潜在的性能优化收益
7. 泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，提高代码的重用率
8. 泛型在使用中的一些规则和限制：
9. 泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。
10. 同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。
11. 泛型的类型参数可以有多个。
12. 泛型的参数类型可以使用extends语句，例如<T extends superclass>。习惯上成为“有界类型”。
13. 泛型的参数类型还可以是通配符类型。例如Class<?> classType = Class.forName(Java.lang.String)。
14. 假如Circle 是Shape的子类，那么注意： List<Circle>不是List<Shape>的子类，所以List<Circle>不能作为对象赋值给List<Shape>
15. 通配符上限：如：<? Extends T> 表示 它必须是 T本身或T的子类
16. 通配符下限：如：<? Super T>表示 它必须是 T本身或T的父类
17. Java不支持创建泛型数组，如 List<String>[] lsa = new List<String>[10]，这样是不允许的，还有List<String>[] lsa = new ArrayList[10]这样也是不行的。
18. 泛型的应用场景
19. 首先最常用的就是集合中类型定义
20. 代码优化与重构
21. 多态的实现
22. 自定义泛型类（模板类）与泛型方法（模板方法）的应用
23. **异常处理**
24. Java中的非正常情况分为两种，一种是错误（Error类），一种是异常（Exception类），这两个类都继承于Throwable父类。Error错误一般指与虚拟机相关的问题，如系统崩溃，虚拟机错误，动态连接失败等，这种错误是无法恢复或者不可能捕获的，将导致程序中断。通常应用程序无法处理这些错误，所以应用程序不应该用catch块来捕获Error对象。Exception一般是程序员导致的，应该在程序中用代码捕获处理的。
25. Java中常见异常类的继承关系图：



1. 运行异常捕获时，应先捕获小异常，再捕获大异常
2. 异常信息:异常类对象包含了以下几个异常信息访问方法：

（1）getMessage():返回该异常的详细描述字符串

（2）printStackTrace():将异常的跟踪栈信息输出到标准错误输出

（3）printStackTrace(PrintStream s):将异常的跟踪栈信息输出到指定的输出流

（4）getStackTrace():返回该异常的跟踪栈信息

1. Finally：除非在try或catch中调用了退出虚拟机的方法，否则finally肯定会被执行。避免在finally块中使用return或throw等导致方法终止的语句。
2. Exception：java中，Exception分为Checked异常和Runtime异常（运行时异常）。所有的RuntimeException类及其子类实例都是Runtime异常；非Runtime异常被称为Checked异常。Checked异常都是可以被处理（修复）的异常，因而代码中必须显示得处理Checked异常；如果没有处理，则无法通过编译。
3. throws与throw 的区别：

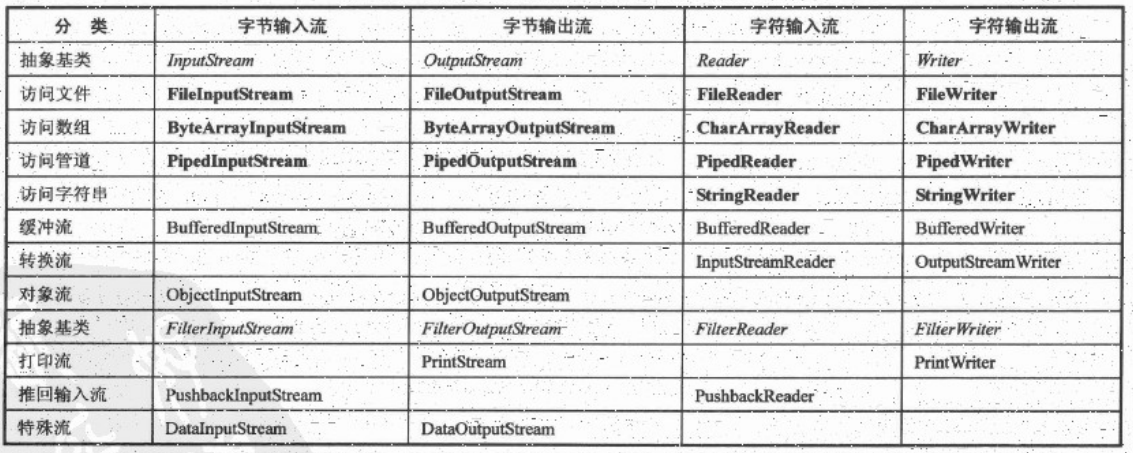
（1）throws出现在方法函数头；而throw出现在函数体。

（2）throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某种异常。

（3）两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

（4）throws说明你有那个可能，倾向。throw的话，那就是你把那个倾向变成真实的了。

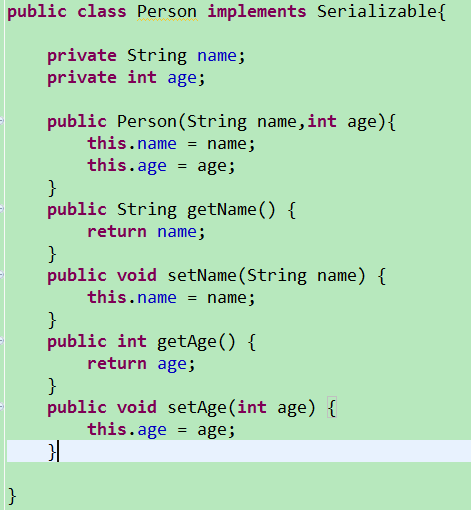
1. **I/O(输入/输出)**
2. I/O流的分类：
3. 按照流向分为输入流和输出流，这是以程序运行的内存为角度划分的。输入流的基类为InputStream和Reader，输出流的基类为OutputStream和Writer。
4. 按照操作数据单元分为字节流和字符流，字节流操作单元为8位的字节，字符流操作单元为16位的字符。字节流主要由InputStream和OutputStream作为基类，字符流由Reader和Writer作为基类。
5. 按照流的角色为为节点流和处理流。节点流指的是可以从/向一个特定的I/O设备（如磁盘、网络）读/写数据的流，节点流也成为低级流；处理流指的是对一个已经存在的流进行连接或封装，通过封装后的流来实现数据的读/写功能，也成为高级流
6. 字符流和字节流：
7. 不要忘记开启的对应流。
8. 输出字符串内容时候，windows下的换行符是：”\r\n”，Linux/Uniux/BSD等平台下是”\n”
9. 使用规则：若进行输入输出的内容是文本内容（如文本文件，文档文件等），则应考虑使用字符流；若进行输入输出的内容是二进制内容（如，图片，声音，影像，视频等文件），则应考虑使用字节流
10. Java输入/输出流体系中常用的流分类：



1. BufferedReader:由于BufferedReader具有一个readLine()方法，可以非常方便一次读入一行内人，所以经常把文本内容的输入流包装成BufferedReader，用来方便的读取输入流的文本内容。用法示例如下：



1. 对象序列化：通常使用Serializable来序列号，实现起来比较简单，就是让目标类实现Serializable这个接口即可。序列化示例：





输出结果如下：



序列化注意点：

（1）对某个对象进行序列化时，系统会自动将全部的Field都序列化，如果某个Field引用到另一个对象，则被引用的对象也会被序列化，如果被引用的对象也引用了其他对象，则被引用的其他对象也会被序列化，也就是说序列化是递归的，具有递归性。

（2）如果哪个Field不想被序列化，则需要用transient关键字来修饰，该关键字只能修饰Field，不能修饰程序其他部分。比如上面的Person类中 在 age前面 加 transient修饰符，那么输出结果为：



（3）除了实现Serializable 接口来实现序列化，java还提供了另外一种方法：就是实现Externalizable接口。这种序列化方式完全有程序员决定存储和恢复对象数据。

（4）为了反序列化时确保序列化版本的兼容性，最好为序列化类添加序列化版本号,即：*serialVersionUID*

1. Java 7的NIO（即：New IO（新IO））

（1）Buffer（缓冲）：最常用或用的最多的是ByteBuffer和CharBuffer，其中ByteBuffer还有一个子类：MappedByteBuffer，它表示Channel将磁盘文件的部分或全部内容映射到内存中后得到的结果。

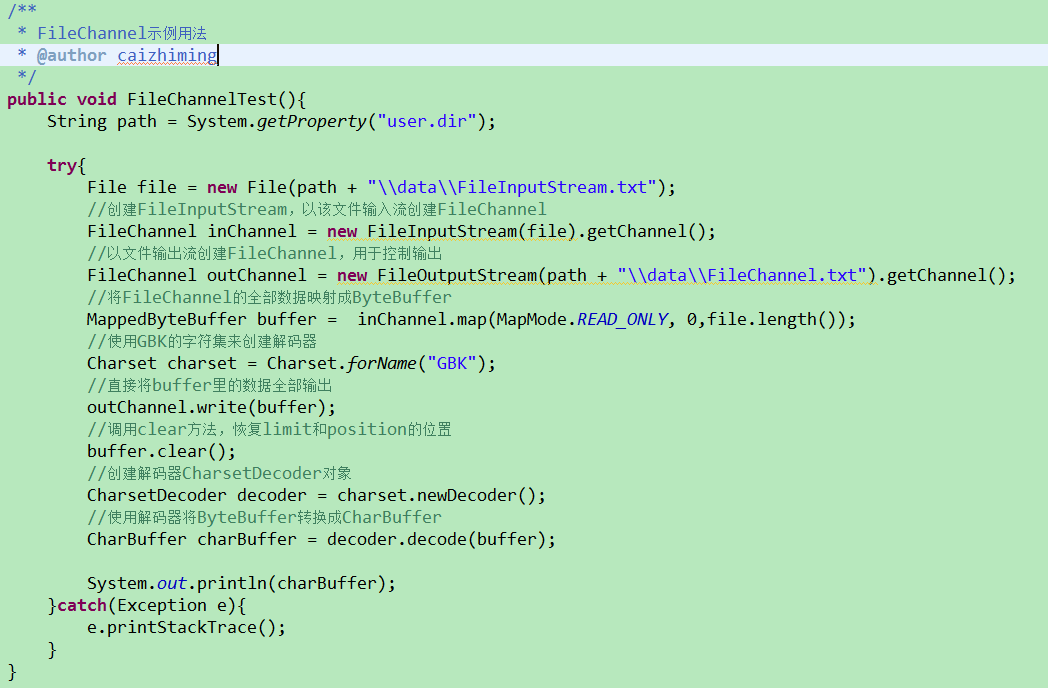
（2）Channel:类似于传统的流对象，但与传统的流对象有两个主要的区别：

a. Channel可以直接将指定文件的部分或全部直接映射成Buffer。

b. 程序不能直接访问Channel中的数据，包括读写都不行，Channel只能与Buffer进行交互。即如果要从Channel中取得数据，必须先用Buffer从Channel中取出来一些数据，然后让程序从Buffer取出这些数据。写入数据类似。

Channel示例：

FileChannel示例：



RandomFileChannel示例：



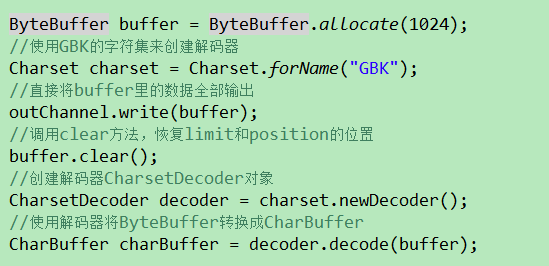
使用FileChannel像传统的流对象一样读文件示例：



1. 字符集和CharSet
2. 创建charset字符集类型：使用Charset的forName（）方法：

如：Charset cs = Charset.forName(“UTF-8”)。

1. 实现ByteBuffer与CharBuffer之间的转换示例：



1. **多线程**
2. 进程与线程
3. 进程：是运行过程中的程序，并且具有一定的独立功能，进程是系统进行资源分配和调度的一个独立单位。线程是[进程](http://baike.baidu.com/view/19746.htm)中某个单一顺序的[控制流](http://baike.baidu.com/view/1462753.htm)。也被称为轻量[进程](http://baike.baidu.com/view/19746.htm)（lightweight processes）。计算机科学术语，指[运行](http://baike.baidu.com/view/1026025.htm)中的程序的调度单位。
4. 线程与进程的区别可以归纳为以下4点：

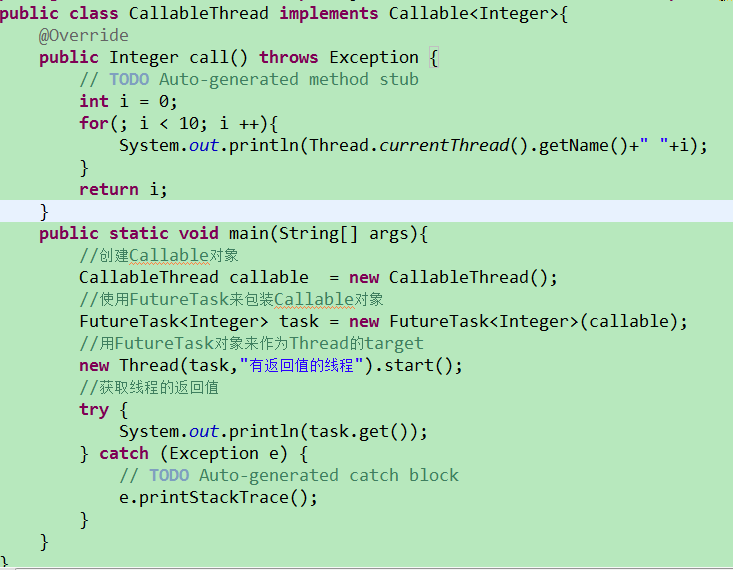
1）地址空间和其它资源（如打开文件）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。

2）通信：[进程间通信](http://baike.baidu.com/view/1492468.htm)[IPC](http://baike.baidu.com/view/373.htm)，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要[进程同步](http://baike.baidu.com/view/1971924.htm)和互斥手段的辅助，以保证数据的一致性。

3）调度和切换：线程上下文切换比进程上下文切换要快得多。

4）在多线程OS中，进程不是一个可执行的实体。

1. 线程的创建：线程的创建主要有两种：一种是实现Runnable接口，另一种是继承Thread类，其实这种本质上和第一种一样，因为Thread类已经实现了Runnable接口了。
2. 线程创建的另外一种方式：使用Callable和Futrue方式
3. Callable方式：与Runnable类似的是，Callable是用一个call（）方法作为线程的执行体，而不是run（）方法。但是call（）方法比run（）方法功能强大。Callable接口有泛型限制，Callable接口里的泛型形参类型必须与call（）方法的返回值一致。
4. call（）方法可以有返回值
5. call（）方法可以声明抛出异常
6. 创建并启动有返回值线程的步骤：
7. 创建Callable接口的实现类，并实现call（）方法，该call（）方法的返回值，并作为线程的执行体。
8. 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call方法的返回值
9. 使用FutureTask对象作为Thread对象的target创建、并启动新线程
10. 调用FutureTask对象的get（）方法来获得子线程执行结束后的返回值
11. 实例：



1. 三种创建线程方法对比：

实现Runnable接口和实现Callable接口方式比较类似，所以分为以下来对比：

1. 采用实现Runnable、Callable方式创建线程：
2. 线程类只是实现了Runnable或Callable接口，还可以继承其他类
3. 该方式多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而将CPU、代码和数据分开，形成清晰的模型，比较好的体现面向对象的思想。
4. 缺点是编程稍微比较复杂，得用Thread.currentThread（）方法获取当前线程。
5. 采用继承Thread类创建线程：
6. 因为继承了Thread类，不能再继承其他类。
7. 编程简单，访问当前线程无需用Thread.currentThread（）方法。
8. 线程的生命周期:
9. 新建和就绪状态：new一个线程后，处于新建状态，执行start（）方法后，线程进入就绪状态，即等待执行状态。
10. 运行和阻塞状态：

当发生如下情况时，线程将会进入阻塞状态：

A、线程调用sleep方法，主动放弃占用的处理器资源

        B、线程调用了阻塞式IO方法，在该方法返回前，该线程被阻塞

        C、线程试图获得一个同步监视器，但该同步监视器正被其他线程所持有。

         D、线程等待某个通知（notify）

         E、程序调用了suspend方法将该线程挂起。不过这个方法容易导致死锁，尽量不免使用该方法

针对上面线程阻塞的情况，发生以下特定的情况可以解除阻塞，让进程进入就绪状态：

        A、调用sleep方法的经过了指定的休眠时间

         B、线程调用的阻塞IO已经返回，阻塞方法执行完毕

         C、线程成功获得了试图同步的监视器

         D、线程正在等待某个通知，其他线程发出了通知

         E、处于挂起状态的线程调用了resume恢复方法

1. 线程死亡：线程会以如下3种方式结束，结束后就处于死亡状态：

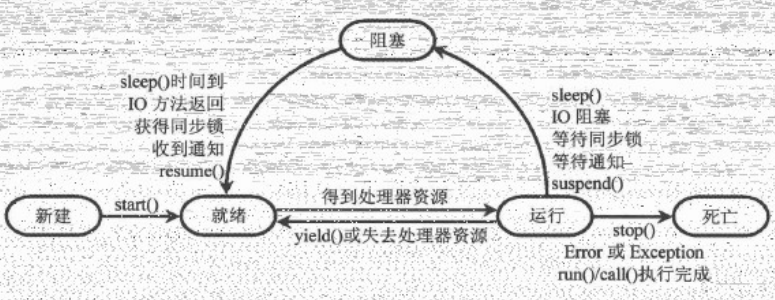
A、run方法执行完成，线程正常结束

    B、线程抛出未捕获的异常或Error

C、直接调用该线程的stop方法来结束线程———-该方法易导致死锁，通常不推荐使用。

注意：当主线程结束时候，其他线程不受任何影响，并不会随之结束。一旦子线程启动后，它就拥有和主线程相同的地位，它不会受主线程的影响。

1. 线程状态转换图：



1. 线程控制
2. join 线程：Thread提供了让一个线程等待另一个完成的方法-join（）方法。当某个程序执行流中调用其它线程的join方法时，调用线程将被阻塞，直到被join（）方法加入的join线程执行完为止。
3. 后台线程：后台线程（Daemon Thread），又称呼为“守护线程”或“精灵线程”。它是在后台运行的，它的任务是为其他线程提供服务。JVM的垃圾回收线程就是典型的后台线程。

特征：如果所有的前台线程都死亡了，那么后台线程会自动死亡。

正是由于上面特征，下面的代码中后台线程无法运行到999，验证了该特征：

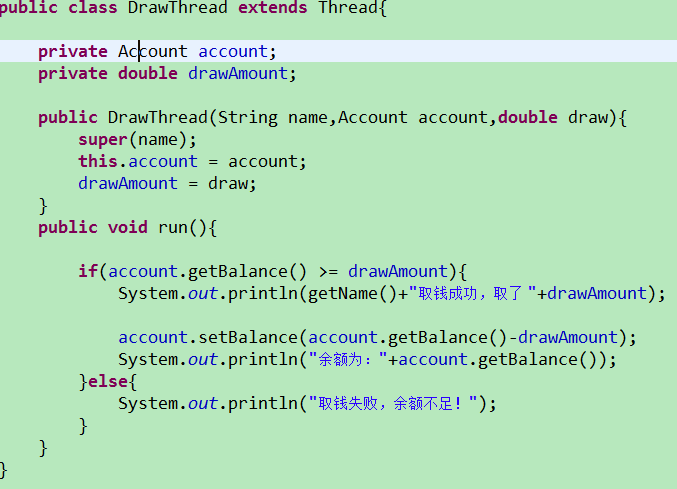


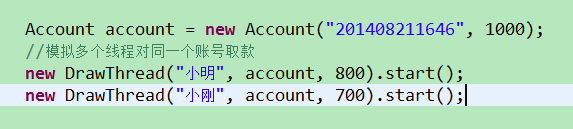
1. 线程睡眠：sleep ：如果需要让当前正在执行的线程暂停一段时间，并进入阻塞状态，则可以通过调用Thread类的静态sleep（）方法来实现。
2. 线程让步：yield（）方法是让当前正在执行的线程暂停，但它不会阻塞该线程，她只是将该线程转入就绪状态。

Sleep与yield的区别：

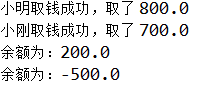
1. Sleep方法暂停当前线程后，会给其他线程执行机会，不会理会其他线程的优先级；但yield方法只会给优先级相同或优先级更高的线程执行机会
2. Sleep方法会将线程转入阻塞状态，直到经过阻塞时间才会转入就绪状态；而yield方法不会转入阻塞状态，它只是强制当前线程进入就绪状态。因此完全有可能某个线程调用那个yield方法暂停之后，立即再次获得处理器资源被执行。
3. Sleep方法声明抛出了InterruptedException异常，而yield方法没有抛出任何异常。
4. Sleep方法比yield饭否具有更好的移植性，一般不建议使用yield方法来控制并发线程的执行。
5. 线程优先级：优先级高的线程获得较多的执行机会，相反，优先级低的执行机会就较少。为了保证程序的可移植性以及不同系统的兼容性，设定优先级时候，尽量避免用数字直接设定优先级，而是使用MAX\_PRIOORITY、MIN\_PRIORITY和NORM\_PRIORITY三个静态常量来设置优先级。
6. 线程同步：
7. 线程安全：多线程操作进程会产生线程安全问题，主要是对同一个资源进行操作导致的。比如如下取钱问题，下面两个线程模拟取钱操作导致最后不合理的结果：





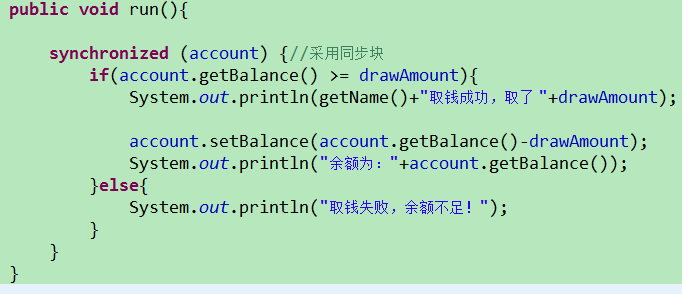


多次运行出现如下结果：

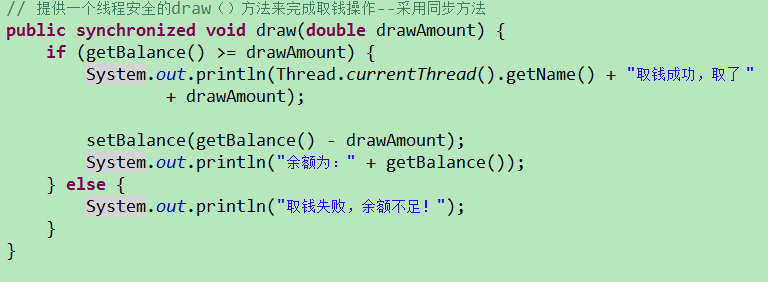


这就是线程安全问题。

1. 同步代码块：用同步监视器synchronized关键字来同步代码块，从而解决线程同步问题。上面的DrawThread类中run方法中代码进行代码块同步如下：

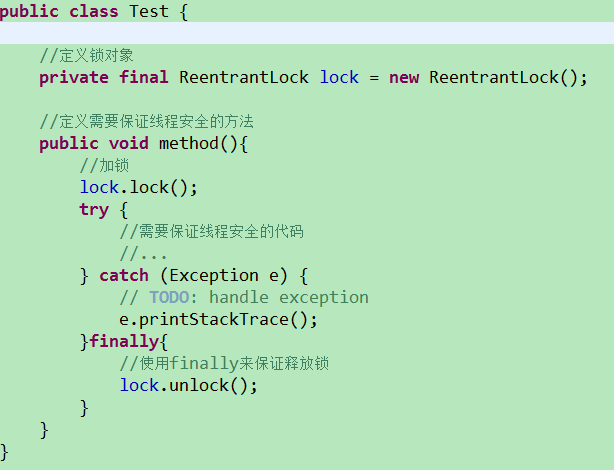


1. 同步方法：用同步监视器synchronized关键字来同步方法，解决上面的线程安全及同步问题：在Account类中添加一个线程安全的draw（）方法来解决。





1. Synchronized：该关键字可以修饰方法，也可以修饰代码块，但不能修饰构造器和属性等。
2. 释放同步监视器的锁定：
3. 同步方法、同步代码块执行结束，释放
4. 执行同步方法、同步代码块中遇到break、return终止了代码的执行，释放
5. 执行同步方法、同步代码块出现Error或Exception，释放
6. 执行当前线程的同步方法或同步代码块时，执行了同步监视器对象的wait（）方法，则当前线程暂停，并释放同步监视器
7. 同步锁（Lock）：Lock是控制多个线程对共享进行访问的工具。Lock，ReadWriteLock是Java 5新提供的两个根接口，并为Lock提供了ReentrantLock（可重入）实现类；为ReadWriteLock提供了ReentrantReadWriteLock实现类。在实现线程安全控制中，比较常用的是ReentrantLock（可重入锁），使用该Lock可以显式地加锁，释放锁。通常使用如下：



ReentrantLock锁具有可重入性，即一个线程可以对已被加锁的ReentrantLock锁再次加锁。

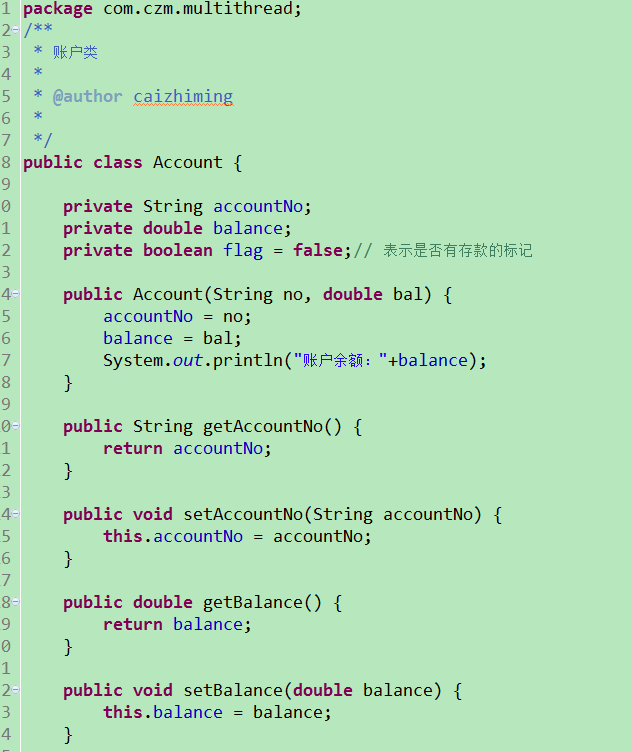
1. 死锁：java中一旦出现死锁，程序既不会发生任何异常，也不会任何提示，只是所有线程都处于阻塞状态，无法继续。容易导致死锁的情况：
2. 对共享资源的同步不当访问。
3. Thread类的suspend方法很容易导致死锁，所以不推荐使用该方法来执行线程暂停。
4. 线程通信：
5. 传统的线程通信：Object类有三个特殊的方法,分别是 wait(),notify(),notifyAll()这3个方法,他们不属于Thread,但和Thread却是息息相关.他们3个必须是由同步监视器对象来调用.
6. 对于使用synchronized修饰的同步方法,因为该类的默认实例(this)就是同步监视器,所以可以在同步方法总直接调用3个方法.
7. 对于使用synchronized修饰的同步代码块,同步监视器是synchronized后括号里的对象,就要用这个对象来调用.

（i） wait():导致当前线程等待,直接其他线程用notify或者notifyAll唤醒.wait有三种1.无时间的,会一直等待.2.两种带时间的.

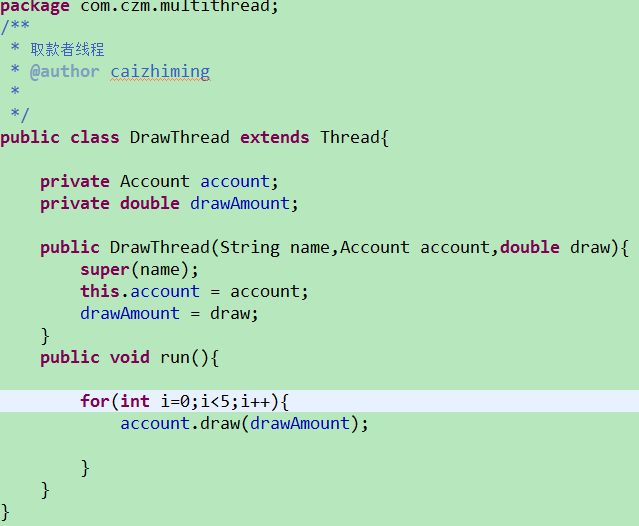
（ii） notify():唤醒在此同步监视器上等待的某个单个线程.选择是随意性的.

（iii）notifyAll():唤醒所有在此同步监视器上等待的线程.

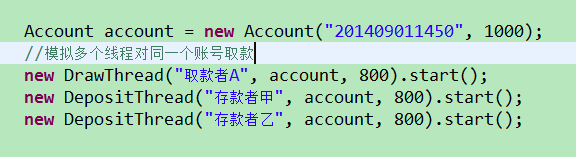
实例解析：











1. 使用Condition控制线程通信：如果程序不使用synchronized关键字来保证同步,而直接用lock.那么系统中不存在隐式的同步监视器,就不能使用wait,notify,notifyAll方法进行线程通信了.但使用Lock对象来保证同步时,Java提供了一个Condition类来保持协调,使用Condition可以让那些已经得到Lock对象但无法继续执行的线程释放Lock对象,Condition对象也可以唤醒其他等待的线程.这情况下,Lock替代了同步方法或同步代码块,Condition替代了同步监视器的功能.

Condition实例绑定在Lock对象上,要获得需要调用Lock对象的newCondition方法.有3个方法:

1. await():类似于隐式同步监视器上的wait方法.该await方法有更多的重载方法.查阅API可知.
2. （ii）signal():任意唤醒一个Lock对象上等待的单个线程.只有当前线程放弃对该Lock对象的锁定后(使用await()方法),才可以执行被唤醒的线程.
3. signalAll():唤醒在此Lock对象上等待的所有线程。只有当前线程放弃对该Lock对象的锁定后，才可以执行被唤醒的线程。

实例：



1. 使用阻塞队列(BlockingQueue)控制线程通信：

虽然Java5 提供了BlockingQueue这个接口，该类是Queue的子接口，但是其主要作用不是容器，而是作为线程同步的工具。BlockingQueue具有一个特征：当生产者线程试图向BlockingQueue中放入元素时，如果该队列已满，则该线程被阻塞；当消费者线程试图从BlockingQueue中取出元素时，如果该队列已空，这该线程被阻塞。

BlockingQueue提供2个阻塞的方法:

（a） put(E e):把E放入到BlockingQueue队尾中,如果队列元素已满,就阻塞该线程.

（b）take():从头部取出元素,如果队列为空,就阻塞该线程.

BlockingQueue继承了Queue接口,自然可以使用Queue的方法.分为3组:

（a）在队列尾部插入元素.包括add(E e),offer(E e),put(E e),队列满的时候,分别是抛异常,返回false,阻塞队列.

（b）在队列头部删除并返回删除的元素,包括remove(),poll().take(),队列已空的时候,分别是抛异常,返回false,阻塞队列.

（c）在队列头部取出但不删除元素,包括element(),peek方法.队列已空的时候,分别是抛异常,返回false,

BlockingQueue包含5个实现类(javaSE 7):

（a）ArrayBlockingQueue:基于数组实现的BlockingQueue.

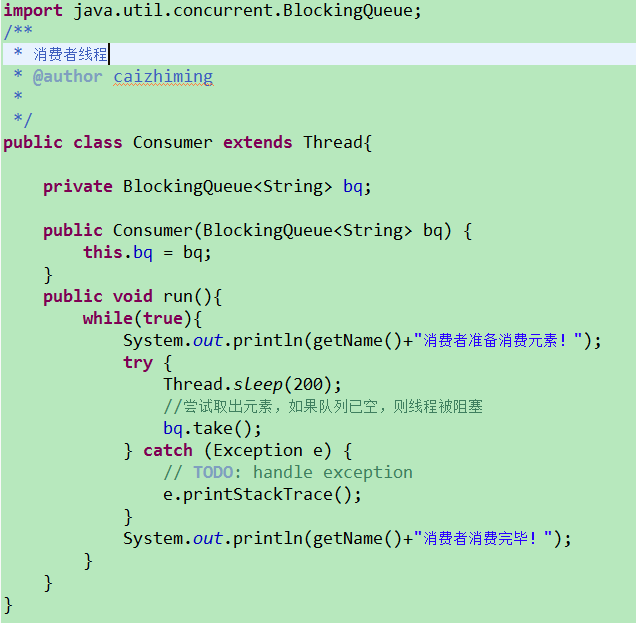
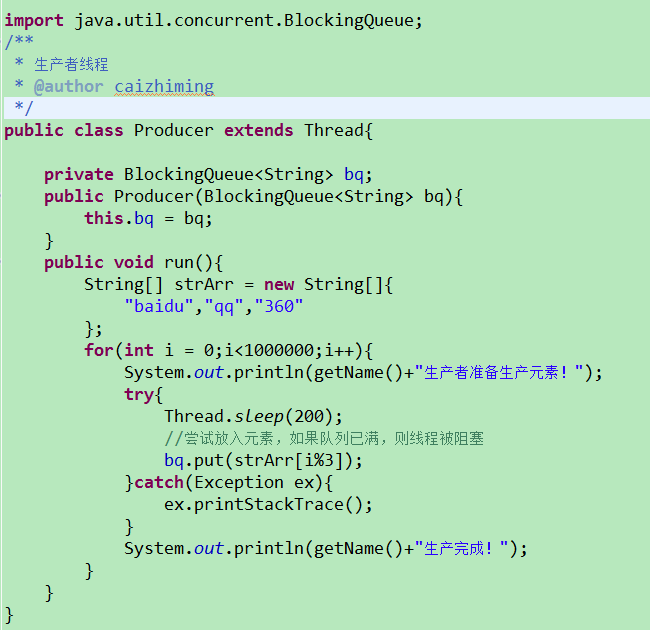
（b）LinkedBlockingQueue:基于链表实现的BlockingQueue.

（c）PriorityBlockingQueue:查API去.不是标准的注射队列.取元素的时候不是取出在队列中存在时间最长的,而是队列中最小的元素.

（d）SynchronousQueue:同步队列,对该队列的存取操作必须交替进行.

（e）DelayQueue:特殊的BlockingQueue,底层基于PriorityBlockingQueue实现.具体API去.

实例解析：



1. 线程组和未处理的异常：

Java使用ThreadGroup来表示线程组,它可以对一批线程进行分类管理,Java允许程序直接对线程组进行控制。对线程组的控制相当于同时控制这批线程。用户创建的所有线程都属于指定线程组,如果程序没有显式指定线程属于哪个线程组,则该线程属于默认线程组.

在默认情况下,子线程和创建它的父线程处于同一个线程组内,列如A线程创建了B线程,并且没有指定B线程的线程组,则B线程属于A线程所在的线程组.

一旦某个线程加入了指定线程组之后,该线程将一直属于该线程组,直到该线程死亡,线程运行中途不能改变它所属的线程组.

Thread类提供了如下几个构造器来设置新创建的线程属于哪个线程组:

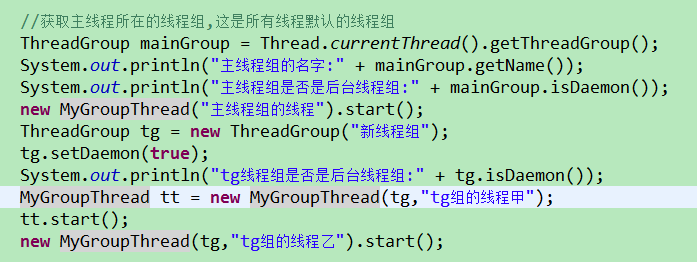
　　（1） Thread(ThreadGroup group,Runnable target): 以target的run()方法作为线程执行体创建新线程,属于group线程组.

　　（2）Thread(ThreadGroup group,Runnable target,String name):以target的run()方法作为线程执行体创建新线程,该线程属于group线程组,且线程名为name.

　 （3）Thread(ThreadGroup group,String):创建新线程,新线程名为name,属于group线程组.

实例解析：





ThreadGroup内还定义了一个很有用的方法:void uncaughtException(Thread t,Throwable e),该方法可以处理该线程组内的任意线程所抛出的未处理的异常.

　　从Java5开始,Java加强了线程的异常处理,如果线程执行过程中抛出了一个未处理的异常,JVM在结束该线程之前会自动查找是否有对应的Thread.UncaughtExceptionHandler对象,如果找到了该处理器对象,则会调用该对象的uncaughtException(Thread t,Throwable e)方法来处理该异常.

　　Thread.UncaughtExceptionHandler是Thread类的一个静态内部接口,该接口内只有一个方法:void uncaughtException(Thread t,Throwable e),该方法中的t代表出现异常的线程,而e代表该线程抛出的异常.

　　Thread类提供了如下两个方法来设置异常处理器:

　　（1）static setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh):为该线程类的所有线程实例设置默认的异常处理器.

　　（2）setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh):为指定的线程实例设置异常处理器.

　　ThreadGroup类实现了Thread.UncaughtExceptionHandler接口,所以每个线程所属的线程组将会作为默认的异常处理器.

　　当一个线程抛出未处理的异常时,JVM会首先查找该异常对应的异常处理器(setUncaughtExceptionHandler()方法设置的异常处理器),如果找到该异常处理器,则将调用该异常处理器处理该异常;否则,JVM将会调用该线程所属的线程组对象的uncaughtException()方法来处理该异常.

　　线程组处理异常的默认流程如下:

　　（1）如果该线程组有父线程组,则调用父线程组的uncaughtException()方法来处理该异常.

　 （2）如果该线程实例所属的线程类有默认的异常处理器(由一个提供的方法，也就是setDefaultUncaughtExceptionHandler()方法设置的异常处理器),那么就调用该异常处理器来处理该异常.

　　（3）如果该异常对象是ThreadDeath的对象,则不做任何处理;否则,将异常跟踪栈的信息打印到System.err错误输出流,并接收该线程.

实例解析：（主线程设置了异常处理器,当主线程运行抛出未处理异常时,该异常处理器将会起作用）。



1. 线程池：
2. Java 5实现的线程池

开始在Java 5版本中，Java内建支持线程池，新增了一个java.util.concurrent.Executors工厂类来产生线程池。Executors主要包含以下几类方法：

（a）newCachedTheradPool()：用来创建一个具有缓存功能的线程池，系统根据需要创建线程，这些线程将被缓存在线程池中。

（b）newFixedTheradPool(intthreadNum)：用于创建一个可重用的、固定线程数的线程池。

（c）newSingleThreadExecutor()：创建单一的线程池。

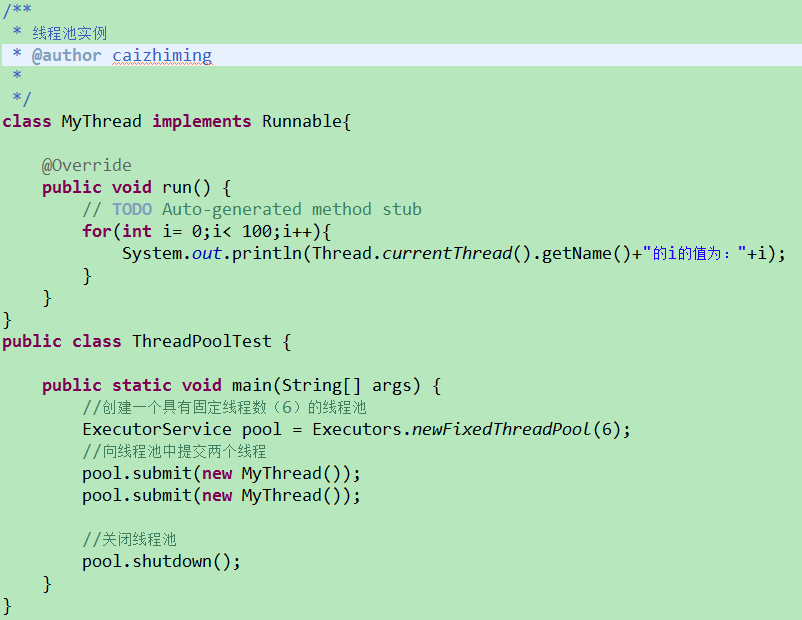
（d）ScheduledThreadPool(intnum)：创建具有指定线程数的线程池，它可以在指定延迟后执行线程任务。num是指池中所保存的线程数。

（e）newSingleThreadScheduledExecutor()：创建单一线程的线程池，它可以在指定延迟后执行线程任务。

上面五个方法中，前三个方法返回都是一个ExecutorService对象，该对象代表一个线程池，而且是立即执行。 后两个方法返回都是一个ScheduledExecutorService对象，它是ExecutorService的子类，可以指定延迟后执行线程任务。

使用线程池来执行线程任务的步骤如下：  
（a）调用Executors类的静态工厂方法创建一个ExecutorService对象，该对象代表一个线程池。  
（b）创建Runnable实现类或Callable实现类的实例，作为线程执行任务。  
（c）调用ExecutorService对象的submit方法来提交Runnable实例或Callable实例  
（d）当不想提交任何任务时，调用ExecutorService对象的shutdown方法来关闭线程池。

实例解析（使用线程池来执行指定Runnable对象所代表的任务）：



1. Java 7新增的ForkJoinPool:

开始在多核时代，程序设计人员不得不开始关注并行编程领域，在Java 7 中，新加入的 Fork/Join 模式是处理并行编程的方法。Fork/Join模式利用多个 CPU 提供的计算资源来协作完成一个复杂的计算任务。在Java 7中，应用ForkJoinPool类把一个 任务拆分成多个“小任务”并行计算，再把多个“小任务”的结果合并成总的计算结果。ForkJoinPool类是ExecutorService的实现类，是一种特殊的线程池。

ForkJoinPool提供了以下两个常用的构造方法：

1）ForkJoinPool(intnum)：创建一个保护num个并行线程的ForkJoinPool；

2）ForkJoinPool()；以Runtime.availableProcessors()方法的返回值作为"num参数"来创建ForkJoinPool。

创建了ForkJoinPool实例之后，可以调用ForkJoinPool的submit(ForkJoinTasktask)或invoke(ForkJoinTask task)方法来执行指定任务， ForkJoinPool 提供的如下函数：

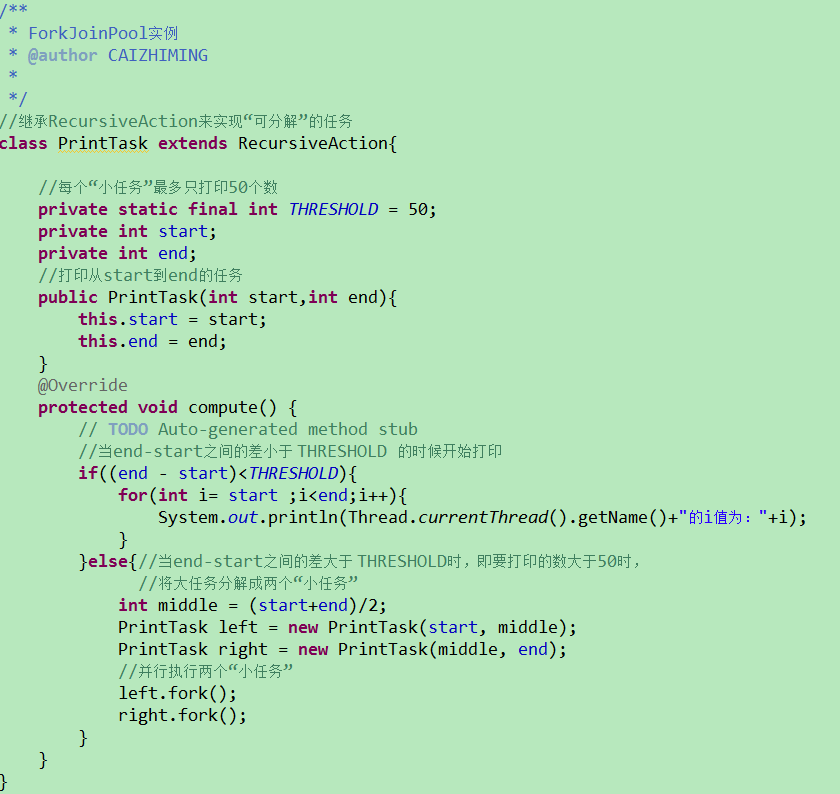
1）submit()：将 ForkJoinTask 类的对象提交给 ForkJoinPool，ForkJoinPool 将立刻开始执行 ForkJoinTask。

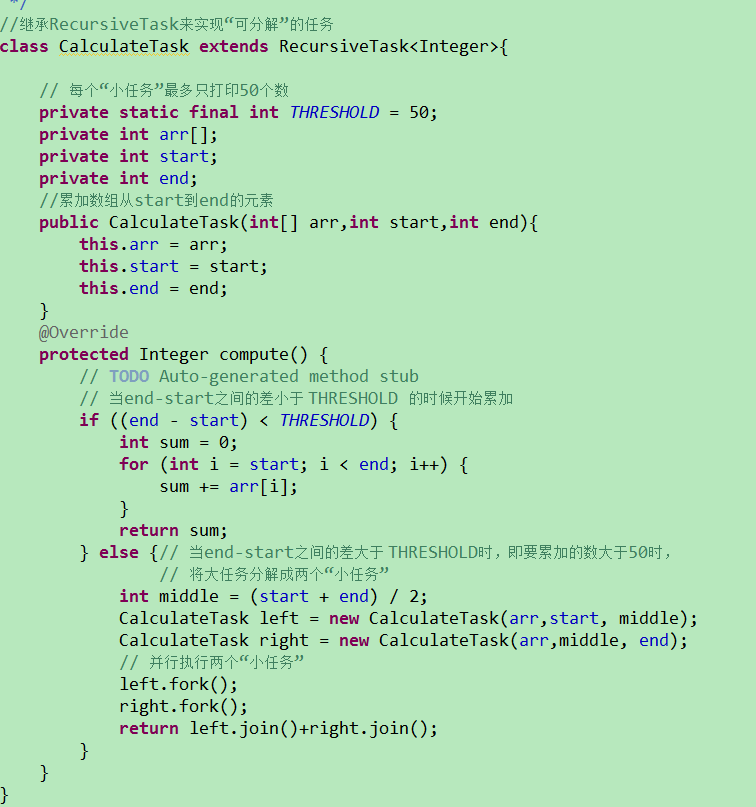
2）shutdown()：执行此方法之后，ForkJoinPool 不再接受新的任务，但是已经提交的任务可以继续执行。如果希望立刻停止所有的任务，可以尝试 shutdownNow() 方法。

3）awaitTermination()：阻塞当前线程直到 ForkJoinPool 中所有的任务都执行结束。

RecursiveAction和RecursiveTask是ForkJoinTask的两个抽象子类。其中RecursiveTask代表又返回值的任务，而RecursiveAction代表没有返回值的任务。

实例解析：







1. 线程相关类：
2. ThreadLocal类

在JDK5后ThreadLocal引入了泛型的支持，通过使用ThreadLocal可以简化多多线程的编程时是并发访问，使用这个工具类可以帮我们更好的实现多线程。

ThreadLocal，是Thread Local Variable（线程的局部变量）的意思。线程局部变量功能非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是每一个线程都可以独立的改变自己的副本，而不会和其他线程的副本冲突。

ThreadLocal提供常用方法：

A、T get()：返回此线程局部变量中当前线程副本中的值

B、void remove()：删除此线程局部变量中当前线程副本中的值

C、void set(T value)：设置此线程局部变量中当前线程副本中的值

ThreadLocal和其他所有同步机制都是为了解决多线程中对同一变量的访问冲突，在普通的同步机制中，是通过对象加锁来实现多个线程对同一变量的安全访问的。

ThreadLocal并不能代替同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是多个线程之间进行通信的有效方式；而ThreadLocal是隔离多个线程的数据共享，从根本上避免了多个线程之间共享资源（变量），也就不需要对多个线程进行同步了。

通常认为：如果需要进行多个线程之间的共享资源，已到达线程之间的通信功能，就使用步机制，如果仅仅需要隔离多个线程之间的共享冲突，就用ThreadLocal。

实例解析：



1. 包装线程不安全的集合

开始Java集合中的ArrayList，LinkedList，HashSet，TreeSet，HashMap都是线程不安全的（线程不安全就是当多个线程想这些集合中放入一个元素时，可能会破坏这些集合数据的完整性）

如何将上面的集合类包装成线程安全的呢？例如，我们需要在多线程中使用线程安全的HashMap对象，则可以采用如下代码：



1. 线程安全的集合类

从Java 5开始，在java.util.concurrent包下提供了大量支持高效并发访问的集合接口和实现类 。主要分为如下两类：

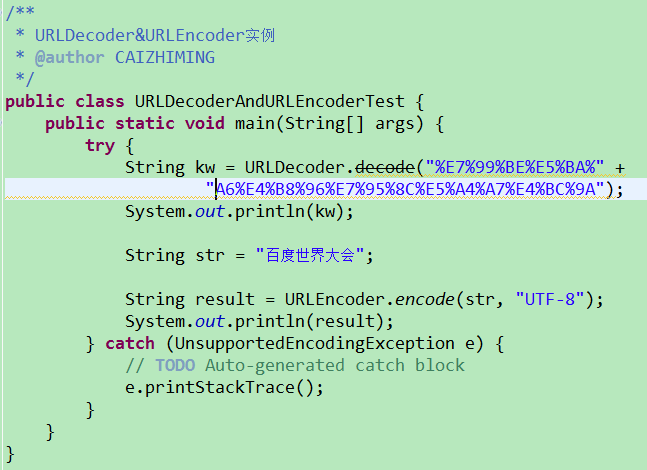
1. 以Concurrent开头的集合类，如ConcurrentHashMap，ConcurrentSkipListMap,ConcurrentSkipListSet, ConcurrentLinkedQueue, ConcurrentLinkedDeque。
2. 以CopyOnWrite开头的集合类，如CopyOnWriteArrayList和CopyOnWriteArraySet。

上面两类的集合类中，（a）类型的支持并发写入，且读取操作不必锁定，在并发写入时候有较好的性能；（b）类型是采用复制底层数组的方式来实现写操作的：即底层复制一份新的数组，然后对新数组进行写入操作，因而写入操作都是对副本进行的，因而是线程安全的。由于需要频繁的复制数组，所以性能比较差，但由于读写的不是一个数组，所以读操作也无需加锁，所以读比较快。所以适合读操作远远大于写操作的情况，比如缓存等。

1. **网络编程**
2. Java的网络基础类
3. InetAddress：该类是表示一个ip地址。实例：

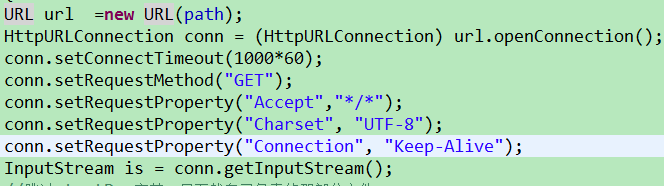


1. URLDecoder & URLEncode：这两个类主要用于完成普通字符串和application/x-www-form-urlencoded MIME字符串之间的相互转换。



1. URL&URLConnection：URL是统一资源定位器（uniform resource locator），URI是统一资源标识符（uniform resource identifiers），java的URI不能用于定位任何资源，唯一的作用就是解析。URL则是包含一个可以打开到达该资源的输入流，可以理解为URL是URL的一种特例。

URLConnection这是一个与URL资源的连接。用法实例：



注意：如果既要使用输入流读取URLConnection响应的内容，又要使用输出流发送请求的参数，则必须先使用输出流，再使用输入流。

1. 基于TCP协议的网络编程-Socket编程

Server端Listen(监听)某个端口是否有连接请求，Client端向Server 端发出Connect(连接)请求，Server端向Client端发回Accept（接受）消息。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。

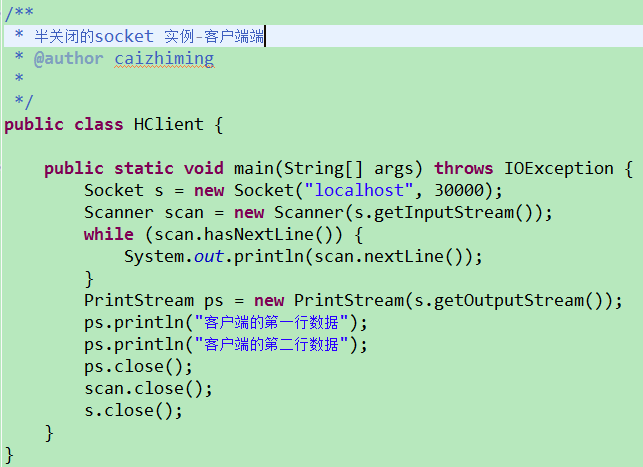
1. 客户端和服务器通讯动作如下：

客户端：  
1）.用服务器的IP地址和端口号实例化Socket对象。  
2）.调用connect方法，连接到服务器上。  
3）.将发送到服务器的IO流填充到IO对象里，比如BufferedReader/PrintWriter。  
4）.利用Socket提供的getInputStream和getOutputStream方法，通过IO流对象，向服务器发送数据流。  
5）. 通讯完成后，关闭打开的IO对象和Socket。  
服务器：  
1）. 在服务器，用一个端口来实例化一个 ServerSocket对象。此时，服务器就可以这个端口时刻监听从客户端发来的连接请求。  
2）.调用ServerSocket的accept方法，开始监听连接从端口上发来的连接请求。 　　  
3）.利用accept方法返回的客户端的Socket对象，进行读写IO的操作  
通讯完成后，关闭打开的流和Socket对象。

1. 半关闭的Socket：

只关闭Socket的输入流和输出流，用以表示输入或输出数据已经发送完毕  
     shutdownInput():关闭Socket的输入流，程序还可以通过该Socket的输出流输出数据  
     shutdownOuput():关闭Socket的输出流，程序还可以通过该Socket的输入流读取数据  
     当调用shutdownInput()或shutdownOutpu()方法关闭Socket的输入流或输出流之后，该Socket处于“半关闭”状态。Socket可通过方法，isInputShutdown()判断Socket是否处于半读状态(read-half)，通过方法isOutputShutdown()判断该Socket是否处于半写状态(read-half),即使同一个Socket实例先后调用shutdownInput()\shutdownOutput()方法，该Socket实例依然没有被关闭，只是该Socket既不能输出数据，也不能读取数据而已. 实例解析：





1. 使用NIO实现非阻塞的Socket通信

**Select编程特点**：select采用不同的工作方式。通过selet你把输入流注册到一个Selector对象上。当某个流发生I/O活动时，selector将会通知你。以这种方式就可以只用一个线程读入多个数据源。尽管Selector不能帮你读取数据，但是它可以监听网络连接请求和越过较慢的通道进行写数据。Java的NIO为非阻塞式的Socket通信提供如下几个特殊类：

**Selector：**它是SelectableChannel对象的多路复用器，所有希望采用非阻塞方式进行通信的Channel都应该注册到Selector对象。可通过调用此类的静态open()方法来创建Selector实例，该方法将使用系统默认的Selector来返回新的Selector。Selector可以同时监控多个SelectableChannel的IO状况，是非阻塞IO的核心。

一个Selector实例有3个SelectionKey的集合：

A.        所有SelectionKey集合：代表了注册在该Selector上的Channel，这个集合可以通过keys()方法返回。

B.        被选择的SelectionKey集合：代表了所有可通过select()方法监测到、需要进行IO处理的Channel，这个集合可以通过selectedKeys()返回。

C.        被取消的SelectionKey集合：代表了所有被取消注册关系的Channel，在下一次执行select()方法时，这些Channel对应的SelectionKey会被彻底删除，程序通常无须直接访问该集合。

Select 相关的方法：

A.        int select() ：监控所有注册的Channel，当它们中间有需要处理的IO操作时，该方法返回，并将对应的SelectionKey加入被选择的SelectionKey集合中，该方法返回这些Channel的数量。

B.        int select(long timeout)：可以设置超时时长的select()操作。

C.        int selectNow()：执行一个立即返回的select()操作，相对于无参数的select()方法而言，该方法不会阻塞线程。

D.        Selector wakeup()：使一个还未返回的select()方法立刻返回。

**SelectableChannel：**

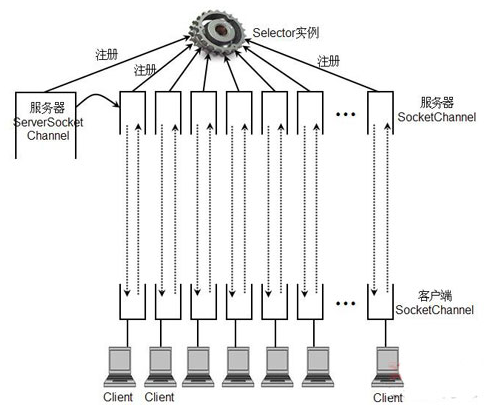
它代表可以支持非阻塞IO操作的Channel对象，可以将其注册到Selector上，这种注册的关系由SelectionKey实例表示。应用程序可调用SelectableChannel 的register()方法将其注册到指定Selector上，当该Selector上某些SelectableChannel上有需要处理的IO操作时，程序可以调用Selector实例的select()方法获取它们的数量，并可以通过selectedKeys()方法返回它们对应的SelectKey集合——通过该集合就可以获取所有需要处理IO操作的SelectableChannel集。

SelectableChannel对象支持阻塞和非阻塞两种模式（所有channel默认都是阻塞模式），必须使用非阻塞式模式才可以利用非阻塞IO操作。

SelectableChannel提供了如下两个方法来设置和返回该Channel的模式状态：

SelectableChannel configureBlocking(boolean block)：设置是否采用阻塞模式。

boolean isBlocking()：返回该Channel是否是阻塞模式。



NIO非阻塞式服务器示意图

总结如下：

**服务器端：**

1) -获得Selector对象

2) 获得ServerSocketChannel 对象serverChannel

3) 把serverChannel跟指定的ip和端口（serverChannel对应的ServerSocket）

4) 设置非阻塞模式

5) 把serverChannel 注册到selector,指定模式（SelectionKey.OP\_ACCEPT）

6) 反复执行

        1> select()选择可以选择的通道

        2> 遍历所有的selectedKeys

isAcceptable:打开监听 SocketChannel channel = serverChannel.accept();

                        把通过监听获得到的channel设置为无阻塞模式

                        注册channel到selector上，同时指定模式（OP\_READ）

                        为sk的interest集合插入一个值（为下次操作做准备）

                        sk.interestOps(SelectionKey)

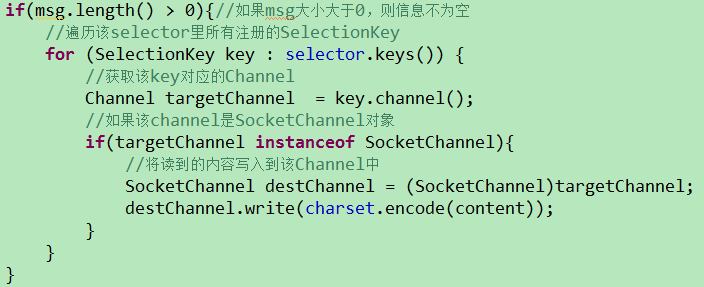
sk.isReadable:根据sk获得它对应的通道：SelectableChannel sc = sk.channel();

                        类型检查if(sc.instanceof SocketChannel)

 如果满足条件做类型转换：SocketChannel socketChannel = (SocketChannel)sc

                        读取数据（读取ByteBuffer）socketChannel.read(buffer);

                        向客户端发送：



**客户端：**

1) 获得一个Selector对象selector;

2) 获得SocketChannel同时指定要连接的服务器的InetSocketAddress对象

3) 设置为非阻塞模式

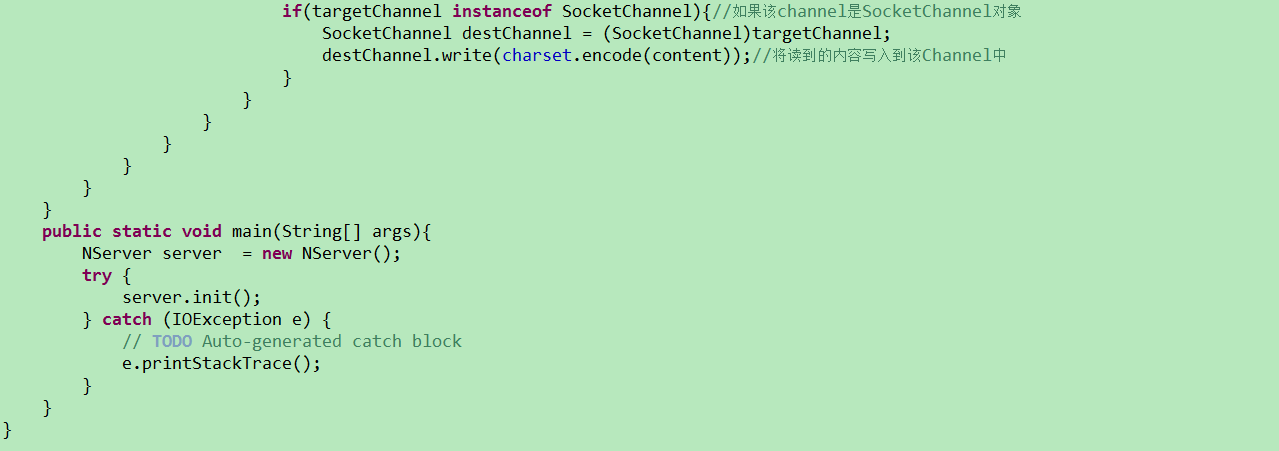
4) 注册SocketChannel到selector同时指定模式为SelectionKey.OP\_READ

5) 使用主线程向服务器发送数据

6) 使用单独的一个线程读取服务器发送来的数据

实例解析：









1. 使用NIO.2中的AIO（Asynchronous IO）实现非阻塞通信：

上面的基于Channel的非阻塞IO和传统的IO都是同步IO，而现在我们介绍的是异步IO。

Java nio 2.0的主要改进就是引入了异步IO（包括文件和网络），这里主要介绍下异步网络IO API的使用以及框架的设计，以TCP服务端为例。首先看下为了支持AIO引入的新的类和接口：  
 **java.nio.channels.AsynchronousChannel**：标记一个channel支持异步IO操作。  
 **java.nio.channels.AsynchronousServerSocketChannel**: ServerSocket的aio版本，创建TCP服务端，绑定地址，监听端口等。  
**java.nio.channels.AsynchronousSocketChannel：**面向流的异步socket channel，表示一个连接。  
 **java.nio.channels.AsynchronousChannelGroup：**异步channel的分组管理，目的是为了资源共享。一个AsynchronousChannelGroup绑定一个线程池，这个线程池执行两个任务：处理IO事件和派发CompletionHandler。AsynchronousServerSocketChannel创建的时候可以传入一个AsynchronousChannelGroup,那么通过AsynchronousServerSocketChannel创建的AsynchronousSocketChannel将同属于一个组，共享资源。  
 **java.nio.channels.CompletionHandler**：异步IO操作结果的回调接口，用于定义在IO操作完成后所作的回调工作。AIO的API允许两种方式来处理异步操作的结果：返回的Future模式或者注册CompletionHandler，我更推荐用CompletionHandler的方式，这些handler的调用是由AsynchronousChannelGroup的线程池派发的。显然，线程池的大小是性能的关键因素。

实例解析：

AIOServer.java:





AIOClient.java：





1. 基于UDP协议的网络编程

UDP协议（User Datagram Protocol），即用户数据报协议. UDP协议是一种面向非连接的协议，面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送。至于对方是否可以接收到这些数据内容，UDP协议无法控制，因此说UDP协议是一种不可靠的协议。UDP适用于一次只传送少量数据、对可靠性要求不高的应用环境。因为UDP协议是面向非连接的协议，没有建立连接的过程，因此它的通信效率高；但也正因为如此，它的可靠性不如TCP协议高。

TCP与UDP区别：

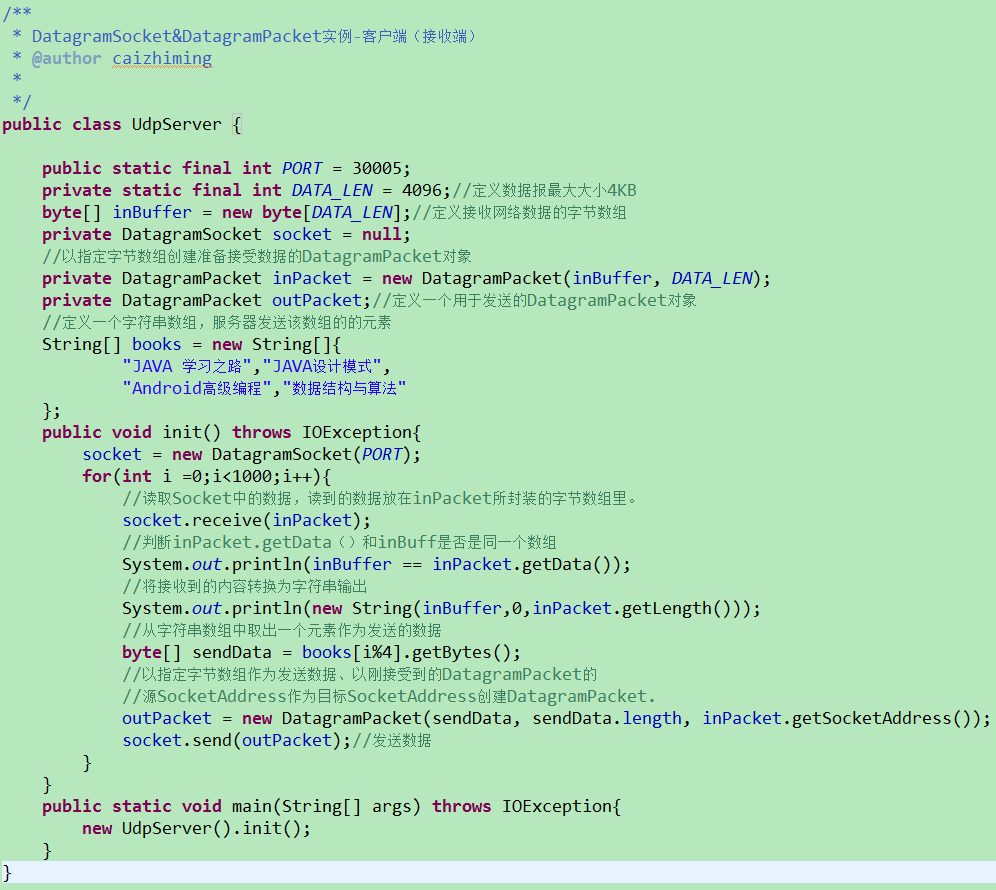
TCP协议：可靠，传输大小无限制，但是需要连接建立时间，差错控制开销大。

UDP协议：不可靠，差错控制开销较小，传输大小限制在64KB以下，不需要建立连接。

1. DatagramSocket &DatagramPacket：

DatagramSocket本身只是码头，不维护状态，不能产生IO流，它的唯一作用就是接收和发送数据报，Java使用DatagramPacket来代表数据报，DatagramSocket接收和发送的数据都是通过DatagramPacket对象完成的。使用DatagramSocket发送数据报时，DatagramSocket并不知道将该数据报发送到哪里，而是由DatagramPacket自身决定数据报的目的。就像码头并不知道每个集装箱的目的地，码头只是将这些集装箱发送出去，而集装箱本身包含了该集装箱的目的地。

当Client/Server程序使用UDP协议时，实际上并没有明显的服务器和客户端，因为两方都需要先建立一个DatagramSocket对象，用来接收或发送数据报，然后使用DatagramPacket对象作为传输数据的载体。通常固定IP、固定端口的DatagramSocket对象所在的程序被称为服务器，因为该DatagramSocket可以主动接收客户端数据。实例解析：



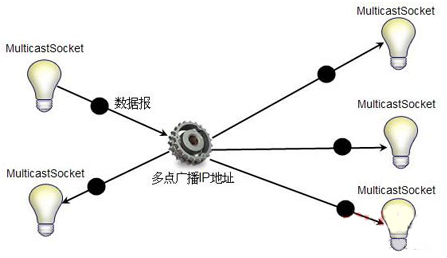


1. MulticastSocket实现广播：

DatagramSocket只允许数据报发送给指定的目标地址，而MulticastSocket可以将数据报以广播方式发送到数量不等的多个客户端。

　　若要使用多点广播时，则需要让一个数据报标有一组目标主机地址，当数据报发出后，整个组的所有主机都能收到该数据报。IP多点广播（或多点发送）实现了将单一信息发送到多个接收者的广播，其思想是设置一组特殊网络地址作为多点广播地址，每一个多点广播地址都被看做一个组，当客户端需要发送、接收广播信息时，加入到该组即可。

IP协议为多点广播提供了这批特殊的IP地址，这些IP地址的范围是224.0.0.0至239.255.255.255. 多点广播的示意图如下图所示：



通过Java实现多点广播时，MulticastSocket类是实现这一功能的关键，当MulticastSocket把一个DatagramPacket发送到多点广播IP地址，该数据报将被自动广播到加入该地址的所有MulticastSocket.MulticastSocket类既可以将数据报发送到多点广播地址，也可以接收其他主机的广播信息。

　　MulticastSocket有点像DatagramSocket,事实上MulticastSocket是DatagramSocket的一个子类，也就是说MulticastSocket是特殊的DatagramSocket.若要发送一个数据报时，可使用随机端口创建MulticastSocket,也可以在指定端口来创建MulticastSocket.

MulticastSocket用于发送、接收数据报的方法与DatagramSocket的完全一样。但MulticastSocket比DatagramSocket多一个setTimeToLive（int ttl）方法，该ttl参数设置数据报最多可以跨过多少个网络，当ttl为0时，指定数据报应停留在本地主机；当ttl的值为1时，指定数据报发送到本地局域网；当ttl的值为32时，意味着只能发送到本站点的网络上；当ttl为64时，意味着数据报应保留在本地区；当ttl的值为128时，意味着数据报应保留在本大洲；当ttl为255时，意味着数据报可发送到所有地方；默认情况下，该ttl的值为1。实例解析：





1. 使用代理服务器:

从JDK1.5开始，Java在java.net包下提供了Proxy和ProxySelector两个类，其中Proxy代表一个代理服务器，可以在打开URLConnection连接时指定所用的Proxy实例，也可以在创建Socket连接时指定Proxy实例。而ProxySelector代表一个代理选择器，它提供了对代理服务器更加灵活的控制，它可以对HTTP、HTTPS、FTP、SOCKS等分别设置，而且还可以设置不需要通过代理服务器的主机和地址。通过使用ProxySelector可以达到像在Internet Explorer、FireFox等软件中设置代理服务器类似的效果。

1. 直接使用Proxy创建连接

Proxy有如下一个构造器：Proxy（Proxy.Type type, SocketAddress sa）：创建表示代理服务器的Proxy对象。而sa参数指定代理服务器的地址，其中type是该代理服务器的类型，该服务器类型有如下三种：

　　Proxy.Type.DIRECT:表示直接连接或缺少代理。

　　Proxy.Type.HTTP:表示高级协议的代理，如 HTTP 或 FTP.

　　Proxy.Type.SOCKS:表示 SOCKS（V4 或 V5）代理。

　　一旦创建了Proxy对象之后，程序就可以在使用URLConnection打开连接时，或创建Socket连接时传入一个Proxy对象，作为本次连接所使用的代理服务器。

　　其中URL包含了一个URLConnection openConnection（Proxy proxy）方法，该方法使用指定的代理服务器来打开连接；而Socket则提供了一个Socket（Proxy proxy）构造器，该构造器使用指定的代理服务器创建一个没有连接的Socket对象。

　　下面以URLConnection为例来介绍如何在URLConnection中使用代理服务器。



1. 使用ProxySelector自动选择代理服务器

如果希望每次打开都有默认的代理服务器可以使用ProxySelector来实现,由于ProxySelector是个抽象类，因而一般是继承ProxySelector类来实现自己的代理。实现ProxySelector就是定义继承ProxySelector的类，并让该类实现如下两个方法：

List<Proxy> select(URI uri):返回代理服务器列表，该方法返回的Proxy会作为默认的代理服务器

connectFailed(URI uri,SocketAddress sa, IOExecption ioe):连接代理服务器失败时候调用。

实现了自己的ProxySelector类之后，就可以调用ProxySelector的setDefault(ProxySelector ps)静态方法来注册该代理服务器

下面是使用ProxySelector来自动选择代理服务器的实例解析：



1. **类加载机制及反射**
2. 类的加载、连接和初始化

是的说法是

byte 占8位，short 16位，int 32位，long 64位, char 占2个字节

基本类型自动转换规则：

1. 类加载器
2. 通过反射查看类信息
3. 使用反射生成并操作对象
4. 使用反射生成JDK动态代理
5. 反射和泛型