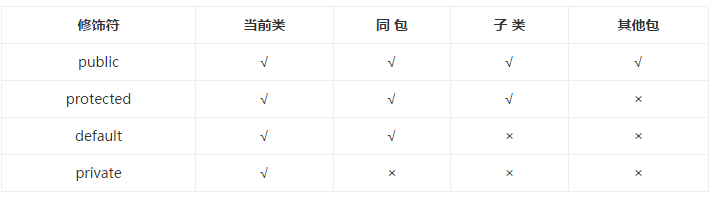
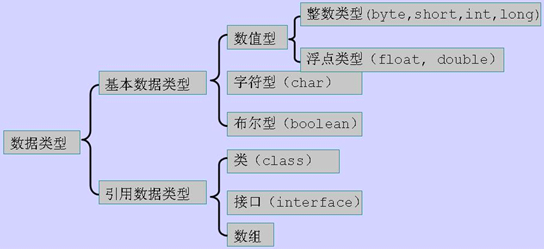
1. Java中面向对象的理解
   1. 封装：隐藏内部变量和方法，向外部提供访问接口；提高代码的可用性和可维护性；
   2. 继承：从基类继承变量和方法来创建子类；提高代码的重用性和可拓展性；
   3. 多态：不同子类型的对象对同一消息做出不同的响应。
      * overload：方法重载，编译时多态，在同一个类里面两个或者是多个方法的方法名相同但是参数不同的情况。
      * override：方法重写，运行时多态，子类重新定义父类的方法，必须有相同的方法名，参数列表和返回类型。
      * 继承+重写+向上转型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。
2. 抽象类与接口
   1. Abstract class：包含抽象方法的类，用来被继承
      * 包含抽象方法，也可以包含具体方法
      * 只要包含抽象方法就一定要被声明为抽象类
      * Abstract不能与final、private、static并列修饰一个类
      * 单继承，抽象方法由子类进行重写，不能实例化
   2. Interface：完全抽象的类，没有提供任何具体实现
      * 所有方法自动被声明为public
      * 所有变量自动被声明为public static final
      * Implements接口的非抽象类必须实现接口的所有方法
      * 不能被实例化，可同时实现多个接口
3. Static与final
   1. Static
      * 变量：全局变量，当声明一个对象时，并不产生static变量的拷贝，而是该类所有的实例变量共用同一个static变量；在类装载时，分配一块存储空间，所有此类的对象都可操作此块存储空间；
      * 方法：ClassName.functionName，只能调用static方法和static变量；
      * 类：只能修饰内部类；
      * 代码块：JVM加载时，按照在类中出现的先后顺序依次执行。
   2. Final
      * 变量：表示常量，不能被修改；可以先声明，在运行时初始化；
      * 方法：不允许子类覆盖该方法；
      * 类：不能被继承。
   3. static final：全局常量、全局方法。
4. 访问修饰符



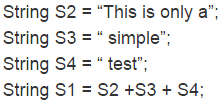
1. 数据类型（primitive type、reference type）



* 1. 基本类型：byte（1字节，-127至128）、short（2字节）、int（4字节）、long（8字节）、float（4字节）、double（8字节）、char（2字节，unicode编码）、boolean（true&false）
  2. 包装类：为了在各种类型间转化，通过各种方法的调用。
     + 自动装箱：JVM自动调用包装器的valueOf方法，Integer obj = m。
     + 自动拆箱：JVM自动调用包装器的 xxxValue方法，int n = obj。
  3. 引用类型（内存管理中细说）
  4. 数据类型之间的转换
     + 自动转换：数值型自动转为字节大的类型；char转换为asc码；将"大"数据转换为"小"数据时，可以使用强制类型转换。
     + 表达式数据类型自动提升：①所有的byte,short,char型的值将被提升为int型；②如果有一个操作数是long型，计算结果是long型；③如果有一个操作数是float型，计算结果是float型；④如果有一个操作数是double型，计算结果是double型。
     + 包装类过渡类型转换
     + 字符串与其它类型间的转换：X.toString()、String.valueOf(X)、parseXXX方法

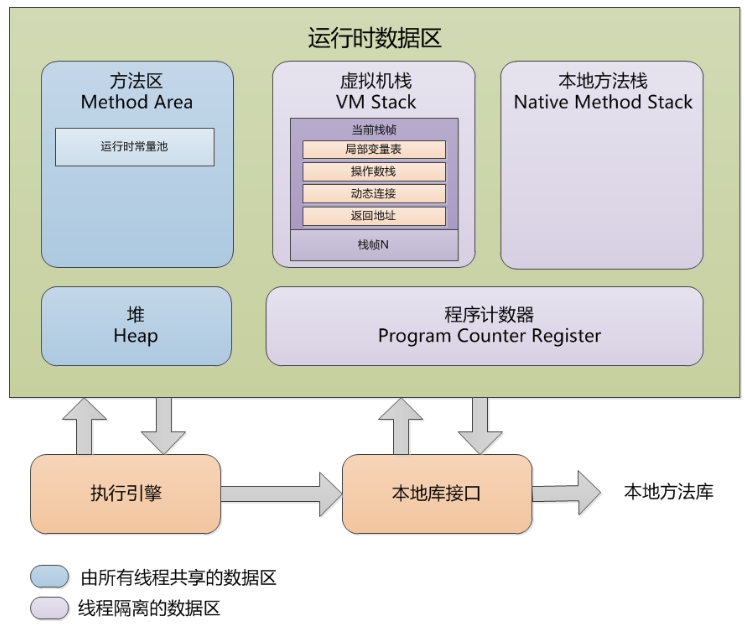
1. String和StringBuilder、StringBuffer
   1. String：只读字符串，引用的字符串内容是不能被改变的；在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，那速度是一定会相当慢的。
   2. StringBuffer：线程安全，多线程环境；每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下， String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢。



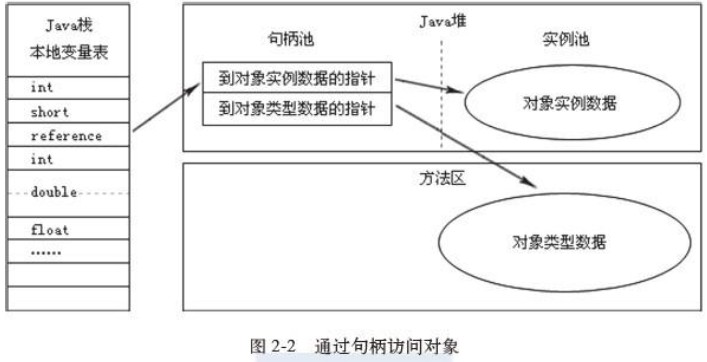


* 1. StringBuilder：单线程环境下使用的，效率较高。

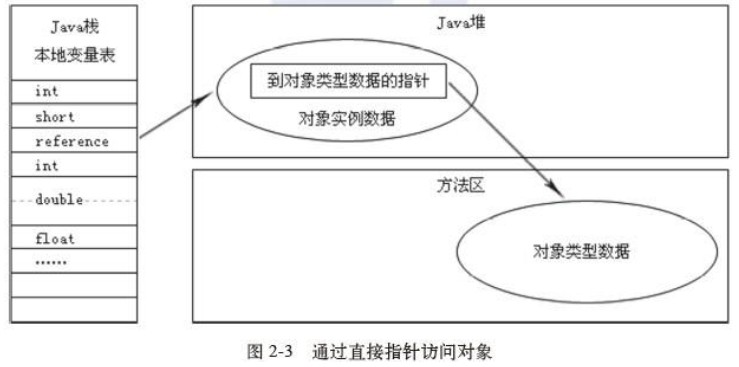
1. 数据存储与参数传递
   1. 数据存储
      * 基本类型：栈
      * 引用类型：引用地址保存在栈中，实例数据保存在堆中，类型数据保存在方法区中。
   2. 参数传递：按值传递
      * 基本类型：传递的是原值的拷贝，不会影响原值；
      * 引用类型：传递的是引用地址的拷贝，对象属性可能会改变，但是对引用的改变不会影响原值。
2. JAVA虚拟机：有虚拟的处理器、堆栈等硬件以及指令系统，通过在实际的计算机上模拟各种计算机功能。可以屏蔽具体操作系统平台的相关信息，使得Java程序只需生成在java虚拟机上运行的字节码文件，就可以在多种平台上不加修改地运行。
   1. 生命周期： java命令启动虚拟机进程，虚拟机进程成功启动之后，读取参数并把它作为初始类加载到内存中，对这个类进行初始化和动态链接，然后从这个类的main方法开始执行。程序执行结束后，虚拟机进程结束。
   2. 体系结构
      * 类加载器：在运行时根据需要来加载类；
      * 执行引擎：执行字节码指令的部分；
      * 垃圾收集：自动释放无用的对象。
   3. 总结
      * 在操作系统的角度看来，它只是一个特殊的进程；可以用来加载我们编写的class文件，并执行class文件中的字节码指令，同时可以自动的管理内存。
3. 内存管理



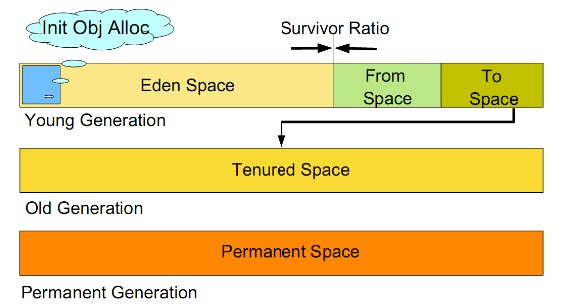
* 1. 程序计数器：当前线程所执行的字节码的行号指示器。
  2. 栈
     + 本地方法栈：线程在调用本地方法时，来存储本地方法的局部变量表，本地方法的操作数栈等等信息。
     + JVM栈：保存当前线程中局部基本类型的变量（）、非基本类型的对象在JVM栈上仅存放一个指向堆上的地址。
     + 栈帧：每次方法调用都会新建一个新的栈帧并把它压栈到栈顶。当方法正常返回或者调用过程中抛出未捕获的异常时，栈帧将出栈。每个栈帧包含局部变量数组（基本类型变量）、返回值、操作数栈、运行时常量池引用（动态链接）。
  3. 堆：JVM用来存储对象实例以及数组值的区域，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回收。
     + 堆是JVM中所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁，这也导致了new对象的开销是比较大的。
  4. 方法区：用于存储已被虚拟机加载的类信息、常量、静态变量，即编译器编译后的代码等数据。当java虚拟机通过类加载器加载这个类的时候，这个类的信息就会保存到方法区中。
     + 常量池：存储了类、方法、接口等中的常量，当然也包括字符串常量。
     + 字符串常量池：编译期类中产生的字符串类型数据。
     + 运行时常量池：虚拟机加载Class后把常量池中的数据放入到运行时常量池。。
  5. 对象访问　Object obj = new Object()
     + 句柄访问方式：Java堆中将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息。对象被移动时只需要改变句柄的实例数据指针。



* + - 直接指针访问方式：Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference中直接存储的就是对象地址。访问时速度更快，节省了一次指针定位的开销。



1. 垃圾回收（Garbage Collection，GC）
   1. 回收算法
      * 引用计数：堆中每个对象实例都有一个引用计数。原理简单，速度较快，无法检测出循环引用。
      * 根搜索：把所有的引用关系看作一张图，从一个根节点开始寻找对应的引用节点，一直到所有的引用节点寻找完毕。剩余未被搜索到的引用节点即为吴用的节点。
      * 标记-清除：从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收；会造成内存碎片。
      * 标记-整理：采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。成本较高，解决内存碎片问题。
      * Copying：把堆分成一个对象面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，垃圾收集就从根集中扫描活动对象，并将每个活动对象复制到空闲面，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。
      * Generation：分代的垃圾回收策略，是基于这样一个事实：不同的对象的生命周期是不一样的。新生代（新生成的对象）、年老代（在年轻代中经历了N次垃圾回收后仍然存活的对象）、持久代（存放静态文件）。



* 1. 垃圾收集器
     + 串行：冻结其他线程，使用一个单独的线程进行垃圾回收；
     + 并行：JVM默认的垃圾回收器，同时开启多个垃圾回收线程；
     + 并发标记扫描：JVM默认的垃圾回收器，同时开启多个垃圾回收线程；
     + G1：将堆内存分割成不同的区域，并且并发的对其进行垃圾回收。G1也可以在回收内存之后对剩余的堆内存空间进行压缩。
  2. 执行机制
     + Scavenge GC：当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。
     + Full GC：Full GC因为需要对整个对进行回收，所以比Scavenge GC要慢，因此应该尽可能减少Full GC的次数。

1. 异常处理

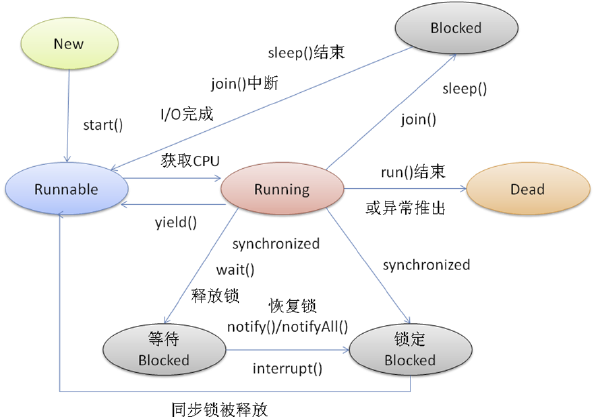


* 1. Error：程序无法处理的错误，表示运行应用程序中较严重问题。
  2. Exception：程序本身可以处理的异常。
     + RuntimeException：运行时异常，一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。例如，若试图使用空值对象引用、除数为零或数组越界NullPointerException、ArithmeticException、ArrayIndexOutOfBoundException。Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。
     + 非运行时异常：从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。
     + 可查异常（编译器要求必须处置的异常）：正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。
     + 不可查异常(编译器不要求强制处置的异常)：运行时异常（RuntimeException与其子类）和错误（Error）。
  3. 处理机制：抛出异常，捕捉异常，处理异常。
     + try－catch-finally语句
     + throws抛出异常
     + throw抛出异常

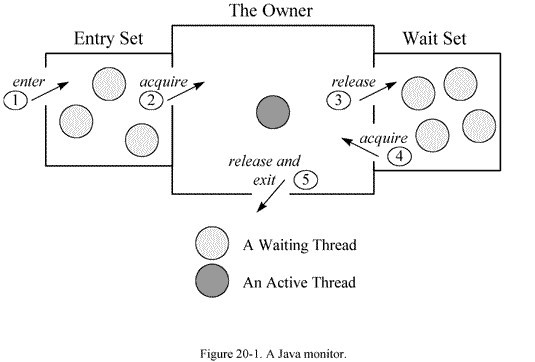
1. 集合类
   1. Collection
      * Set：不包含重复的元素。
        + HashSet：根据hashcode和equals来判断是否是同一个对象，如果hashcode一样，并且equals返回true，则是同一个对象，不能重复存放。存储自定义对象时，需要重写hashCode()和equals()方法来区分同一对象后。
        + TreeSet：使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator 进行排序。自定义对象需要实现Comparable接口。
      * List：有序的集合，可以包含重复元素。
        + ArrayList：长度可以改变的数组，可以对元素进行随机的访问，插入与删除元素的速度慢。
        + LinkedList：采用链表数据结构，插入和删除速度快，访问速度慢。

* 遍历
  + for(int i=0; i<list.size();i++){ System.out.println(list.get(i)); }
  + Iterator it=list.iterator(); while(it.hashNext){ System.out.println(it.next); }
  1. Map：key-value，不包含重复的key，但是可以包含相同的value
     + HashTable：线程安全，任意时刻只有一个线程可以写，导致速度较慢。
     + HashMap：根据键的HashCode值存储数据，根据键可以直接获取它的值，具有很快的访问速度，遍历时，取得数据的顺序是完全随机的。因为键对象不可以重复，所以HashMap最多只允许一条记录的键为Null，允许多条记录的值为Null，是非同步的。
     + TreeMap：能够把它保存的记录根据键排序，默认是按键值的升序排序（自然顺序），也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。不允许key值为空，非同步的。
     + 遍历
       - KeySet()：Iterator it = map.keySet().iterator();
       - entrySet()：Iterator it = map.entrySet().iterator();while(it.hasNext()){Entry e =(Entry) it.next();System.out.println("键"+e.getKey () + "的值为" + e.getValue());}

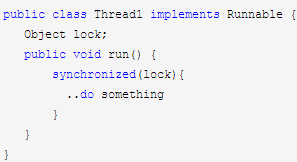
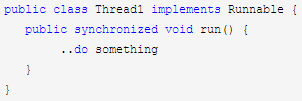
1. 并行编程
   1. 线程状态



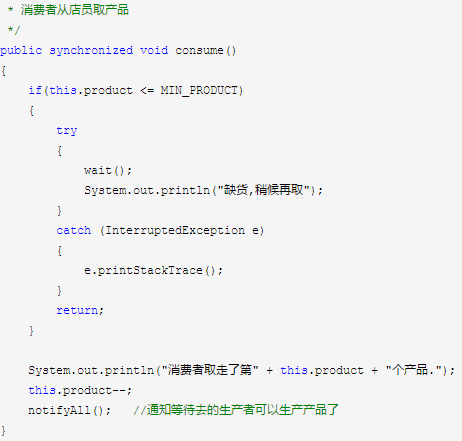
* + - 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
    - 调用wait()，使该线程处于等待池(wait blocked pool)，直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）
    - 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。
  1. 同步工具



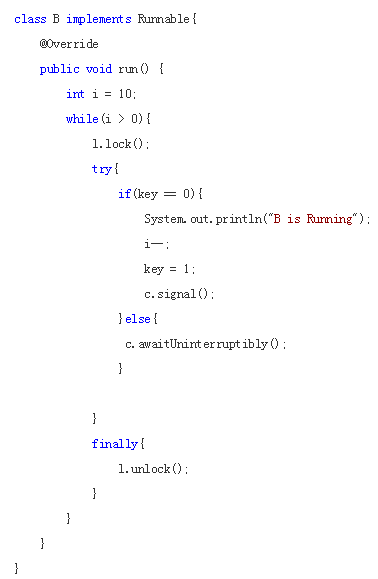
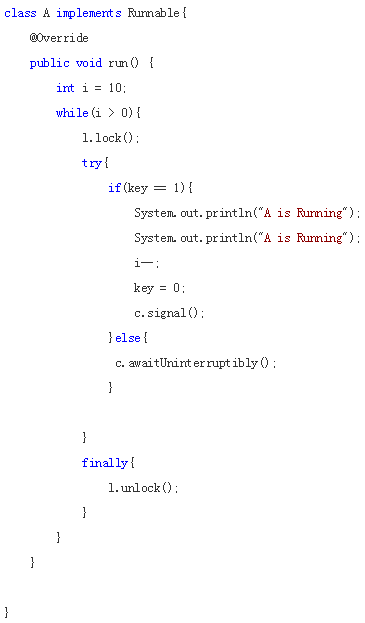
* + - Monitor：Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized 范围内，监视器发挥作用。
    - synchronized单独使用
      * 代码块：获取了lock实例的monitor
      * 方法：获取的是Thread1类的monitor

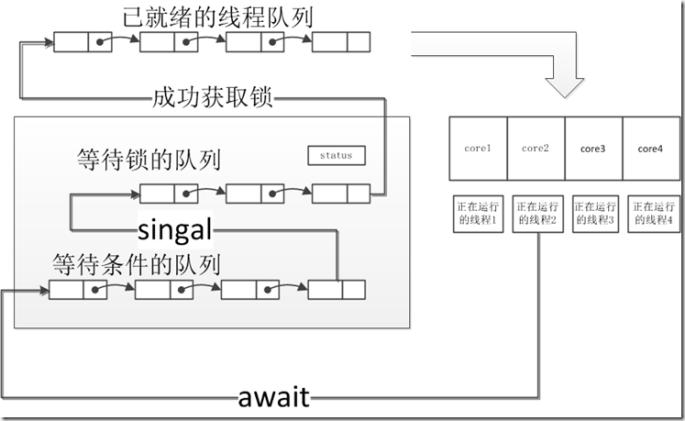
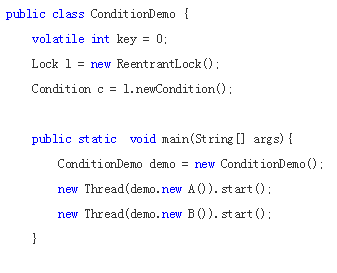
 

* + - synchronized, wait, notify结合

* + - volatile：多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存load到本地栈，完成操作后再save回去(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。
  1. 基本线程类
     + Thread类
       - class ThreadTest extends Thread，重写run方法，ThreadTest tt = new ThreadTest()，tt.start();
       - Thread.sleep()、Thread.yield()、Thread.join()
       - object.wait()、object.notify()/notifyAll()
     + Runnable接口
       - class ThreadTest implements Runnable，重写run方法，ThreadTest tt = new ThreadTest()，Thread t = new Thread(tt)，t.start();
     + Callable接口
       - 泛型接口，call()函数返回的类型就是传递进来的V类型
       - Future可以拿到异步执行任务的返回值
  2. 高级多线程控制类：java.util.concurrent
     + ThreadLocal类：当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。
       - void set(Object value)设置当前线程的线程局部变量的值。
       - public Object get()该方法返回当前线程所对应的线程局部变量。
       - public void remove()将当前线程局部变量的值删除，目的是为了减少内存的占用。
       - protected Object initialValue()返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。
     + 原子类（硬件实现原子操作）
       - 乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，适用于多读的应用类型，这样可以提高吞吐量。
       - 悲观锁：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。
       - AtomicInteger、AtomicBoolean、AtomicLong、AtomicReference
     + Lock类
       - Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的
       - 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的条件下提供一种退出的机制。
       - 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以 及设置等待时限等方式退出条件队列。
       - synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式。
       - ReentrantLock可重入锁：有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。
       - ReentrantReadWriteLock可重入读写锁
       - Condition：一个实例的内部实际上维护了一个队列，队列中的节点表示由于（某些条件不满足而）线程自身调用await方法阻塞的线程。
         * Await：入列到条件队列，释放锁，阻塞自身线程；被唤醒后尝试获取锁（此时线程位于等待队列），成功获取则从await中返回并执行线程后面的代码；否则，继续阻塞等待；
         * Signal：将条件队列的队首节点取出，放入等待锁队列的队尾，唤醒该节点对应的线程；





* + - 容器类
      * BlockingQueue：阻塞队列，单向队列，可以在队列头添加元素和在队尾删除或取出元素。
      * ConcurrentHashMap：基于分段锁，提高效率；ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。
    - 管理类
      * ThreadPoolExecutor
      * ScheduledThreadPoolExecutor



1. SSH框架：Struts作为系统的整体基础架构，负责MVC的分离，在Struts框架的模型部分，控制业务跳转，利用Hibernate框架对持久层提供支持。Spring一方面作为一个轻量级的IoC容器，负责查找、定位、创建和管理对象及对象之间的依赖关系，另一方面能使Struts和Hibernate更好地工作。
   1. Struts：它通过采用 Java Servlet/JSP 技术，实现了基于Java EEWeb应用的MVC设计模式的应用框架，是MVC经典设计模式中的一个经典产品。
   2. Spring：Spring是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架。
      * 控制反转：依赖注入，当我们在创建一个类或方法的时候，不再主动创建，而是将控制权交给容器，容器创建完之后把这个对象注入到我们自己的类中，使我们可以直接使用。
      * 面向切面：多个逻辑过程中相同的操作抽离出来，例如日志管理
   3. Hibernate：对JDBC进行了非常轻量级的对象封装，大大简化了数据访问层繁琐的重复性代码。简化DAO层的编码工作，将软件开发人员从大量相同的数据持久层相关编程工作中解放出来，使开发更对象化了。移植性好，支持各种数据库，如果换个数据库只要在配置文件中变换配置就可以了，不用改变hibernate代码。支持透明持久化，因为hibernate操作的是纯粹的（pojo）java类，没有实现任何接口，没有侵入性。