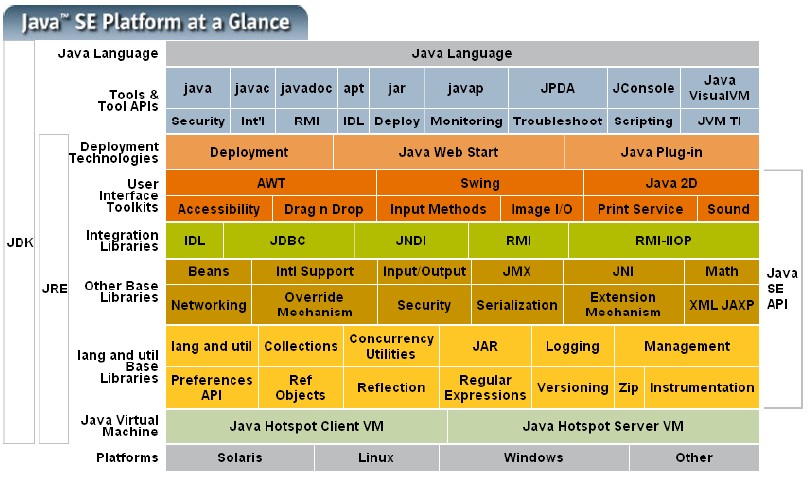
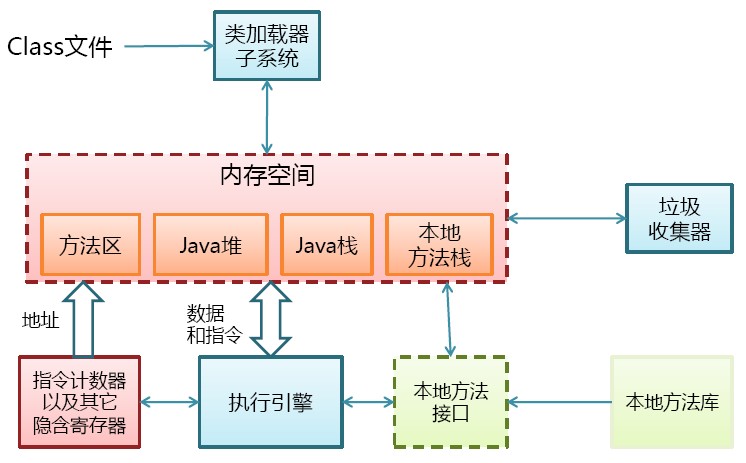
**[JVM学习笔记（一）------基本结构](http://www.blogjava.net/qileilove/archive/2011/11/11/363492.html)**

　从Java平台的逻辑结构上来看，我们可以从下图来了解JVM：

[](http://www.51testing.com/batch.download.php?aid=29055)

　　从上图能清晰看到Java平台包含的各个逻辑模块，也能了解到JDK与JRE的区别

　　对于JVM自身的物理结构，我们可以从下图鸟瞰一下：

[](http://www.51testing.com/batch.download.php?aid=29056)

　　对于JVM的**[学习](http://www.51testing.com/html/98/n-248498.html" \t "_self)**，在我看来这么几个部分最重要：

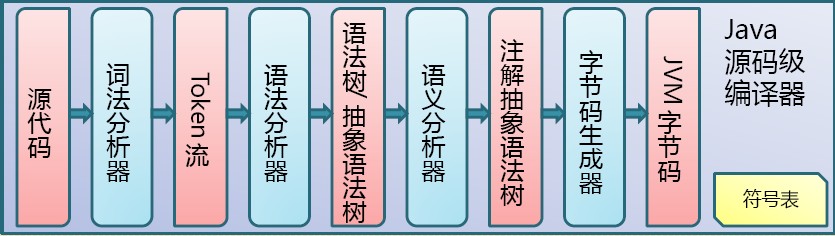
　　● Java代码编译和执行的整个过程

　　● JVM内存管理及垃圾回收机制

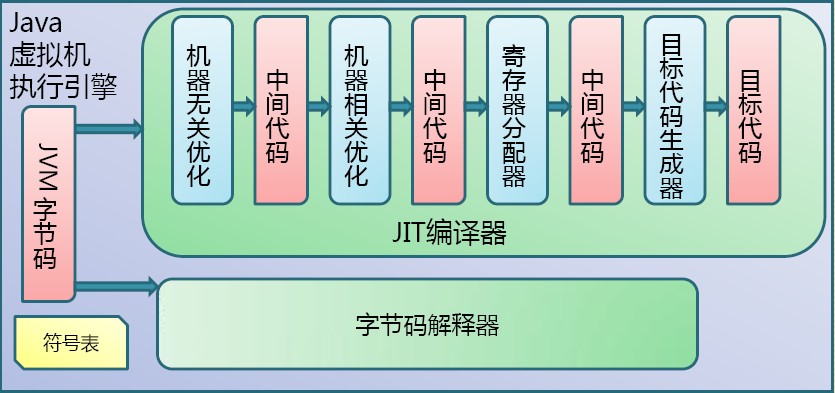
　　下面将这两个部分进行详细学习。

[**JVM学习笔记（二）------Java代码编译和执行的整个过程**](http://www.blogjava.net/qileilove/archive/2011/11/14/363659.html)

[**Java**](javascript:;)代码编译是由Java源码编译器来完成，流程图如下所示：

[](http://www.51testing.com/batch.download.php?aid=29091)

　　Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：

[](http://www.51testing.com/batch.download.php?aid=29095)

　　Java代码编译和执行的整个过程包含了以下三个重要的机制：

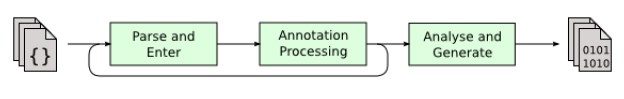
　　● Java源码编译机制  
　　● 类加载机制  
　　● 类执行机制

**Java源码编译机制**

　　Java 源码编译由以下三个过程组成：

　　● 分析和输入到符号表  
　　● 注解处理  
　　● 语义分析和生成class文件

　　流程图如下所示：

[](http://www.51testing.com/batch.download.php?aid=29092)

　　最后生成的class文件由以下部分组成：

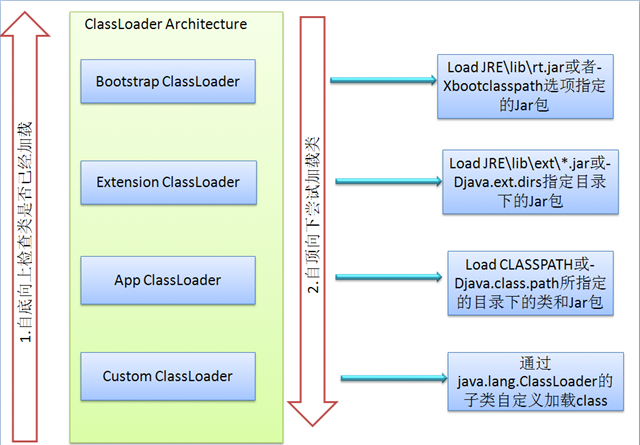
　　● 结构信息。包括class文件格式版本号及各部分的数量与大小的信息。

　　● 元数据。对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池。

　　● 方法信息。对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息。

**类加载机制**

　　JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：

[](http://www.51testing.com/batch.download.php?aid=29093)

　　1）Bootstrap ClassLoader  
　　负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

　　2）Extension ClassLoader  
　　负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

　　3）App ClassLoader  
　　负责记载classpath中指定的jar包及目录中class

　　4）Custom ClassLoader  
　　属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

　　加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

**类执行机制**

　　JVM是基于栈的体系结构来执行class字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。栈的结构如下图所示：

[](http://www.51testing.com/batch.download.php?aid=29094)

[**JVM学习笔记（二）------JVM内存管理**](http://www.blogjava.net/qileilove/archive/2011/11/14/363659.html)

不同的平台，内存模型是不一样的，但是jvm的内存模型规范是统一的。其实java的多线程并发问题最终都会反映在java的内存模型上，**所谓线程安全无非是要控制多个线程对某个资源的有序访问或修改。总结java的内存模型**，要解决两个主要的问题：**可见性和有序性**。  
我们都知道计算机有高速缓存的存在，处理器并不是每次处理数据都是取内存的。JVM定义了自己的内存模型，屏蔽了底层平台内存管理细节，对于java开发人员，要清楚在jvm内存模型的基础上，如果解决多线程的可见性和有序性。  
**那么，何谓可见性？ 多个线程之间是不能互相传递数据通信的，它们之间的沟通只能通过共享变量来进行。**Java内存模型（JMM）规定了**jvm有主内存，主内存是多个线程共享的**。当new一个对象的时候，也是被分配在主内存中，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本，当然线程的工作内存大小是有限制的。当线程操作某个对象时，执行顺序如下：  
**(1) 从主存复制变量到当前工作内存 (read and load)  
 (2) 执行代码，改变共享变量值 (use and assign)  
 (3) 用工作内存数据刷新主存相关内容 (store and write)**

JVM规范定义了线程对主存的操作指令：read，load，use，assign，store，write。当一个共享变量在多个线程的工作内存中都有副本时，如果一个线程修改了这个共享变量，那么其他线程应该能够看到这个被修改后的值，这就是多线程的可见性问题。  
        **那么，什么是有序性呢** ？线程在引用变量时不能直接从主内存中引用,如果线程工作内存中没有该变量,则会从主内存中拷贝一个副本到工作内存中,这个过程为read-load,完成后线程会引用该副本。当同一线程再度引用该字段时,有可能重新从主存中获取变量副本(read-load-use),也有可能直接引用原来的副本 (use),也就是说 read,load,use顺序可以由JVM实现系统决定。  
        线程不能直接为主存中中字段赋值，它会将值指定给工作内存中的变量副本(assign),完成后这个变量副本会同步到主存储区(store- write)，至于何时同步过去，根据JVM实现系统决定.有该字段,则会从主内存中将该字段赋值到工作内存中,这个过程为read-load,完成后线程会引用该变量副本，当同一线程多次重复对字段赋值时,比如：

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. for(int i=0;i<10;i++)
2. a++;

线程有可能只对工作内存中的副本进行赋值,只到最后一次赋值后才同步到主存储区，所以assign,store,weite顺序可以由JVM实现系统决定。假设有一个共享变量x，线程a执行x=x+1。从上面的描述中可以知道x=x+1并不是一个原子操作，它的执行过程如下：  
1 从主存中读取变量x副本到工作内存  
2 给x加1  
3 将x加1后的值写回主 存  
如果另外一个线程b执行x=x-1，执行过程如下：  
1 从主存中读取变量x副本到工作内存  
2 给x减1  
3 将x减1后的值写回主存   
那么显然，最终的x的值是不可靠的。假设x现在为10，线程a加1，线程b减1，从表面上看，似乎最终x还是为10，但是多线程情况下会有这种情况发生：  
1：线程a从主存读取x副本到工作内存，工作内存中x值为10  
2：线程b从主存读取x副本到工作内存，工作内存中x值为10  
3：线程a将工作内存中x加1，工作内存中x值为11  
4：线程a将x提交主存中，主存中x为11  
5：线程b将工作内存中x值减1，工作内存中x值为9  
6：线程b将x提交到中主存中，主存中x为9   
同样，x有可能为11，如果x是一个银行账户，线程a存款，线程b扣款，显然这样是有严重问题的，要解决这个问题，必须保证线程a和线程b是有序执行的，并且每个线程执行的加1或减1是一个原子操作。看看下面代码：

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. public class Account {
3. private int balance;
5. public Account(int balance) {
6. this.balance = balance;
7. }
9. public int getBalance() {
10. return balance;
11. }
13. public void add(int num) {
14. balance = balance + num;
15. }
17. public void withdraw(int num) {
18. balance = balance - num;
19. }
21. public static void main(String[] args) throws InterruptedException {
22. Account account = new Account(1000);
23. Thread a = new Thread(new AddThread(account, 20), "add");
24. Thread b = new Thread(new WithdrawThread(account, 20), "withdraw");
25. a.start();
26. b.start();
27. a.join();
28. b.join();
29. System.out.println(account.getBalance());
30. }
32. static class AddThread implements Runnable {
33. Account account;
34. int     amount;
36. public AddThread(Account account, int amount) {
37. this.account = account;
38. this.amount = amount;
39. }
41. public void run() {
42. for (int i = 0; i < 200000; i++) {
43. account.add(amount);
44. }
45. }
46. }
48. static class WithdrawThread implements Runnable {
49. Account account;
50. int     amount;
52. public WithdrawThread(Account account, int amount) {
53. this.account = account;
54. this.amount = amount;
55. }
57. public void run() {
58. for (int i = 0; i < 100000; i++) {
59. account.withdraw(amount);
60. }
61. }
62. }
63. }

第一次执行结果为10200，第二次执行结果为1060，每次执行的结果都是不确定的，因为线程的执行顺序是不可预见的。这是java同步产生的根源，synchronized关键字保证了多个线程对于同步块是互斥的，synchronized作为一种同步手段，解决java多线程的执行有序性和内存可见性，而volatile关键字之解决多线程的内存可见性问题。后面将会详细介绍。

**synchronized关键字**   
        上面说了，java用synchronized关键字做为多线程并发环境的执行有序性的保证手段之一。当一段代码会修改共享变量，这一段代码成为互斥区或临界区，为了保证共享变量的正确性，synchronized标示了临界区。典型的用法如下：

1. synchronized(锁){
2. 临界区代码
3. }

为了保证银行账户的安全，可以操作账户的方法如下：

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. public synchronized void add(int num) {
2. balance = balance + num;
3. }
4. public synchronized void withdraw(int num) {
5. balance = balance - num;
6. }

刚才不是说了synchronized的用法是这样的吗：

1. synchronized(锁){
2. 临界区代码
3. }

那么对于public synchronized void add(int num)这种情况，意味着什么呢？其实这种情况，锁就是这个方法所在的对象。同理，如果方法是public  static synchronized void add(int num)，那么锁就是这个方法所在的class。  
        理论上，每个对象都可以做为锁，但一个对象做为锁时，应该被多个线程共享，这样才显得有意义，在并发环境下，一个没有共享的对象作为锁是没有意义的。假如有这样的代码：

1. public class ThreadTest{
2. public void test(){
3. Object lock=new Object();
4. synchronized (lock){
5. //do something
6. }
7. }
8. }

**lock变量作为一个锁存在根本没有意义，因为它根本不是共享对象，每个线程进来都会执行Object lock=new Object();每个线程都有自己的lock，根本不存在锁竞争。**  
        每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列，就绪队列存储了将要获得锁的线程，阻塞队列存储了被阻塞的线程，当一个被线程被唤醒 (notify)后，才会进入到就绪队列，等待cpu的调度。当一开始线程a第一次执行account.add方法时，jvm会检查锁对象account 的就绪队列是否已经有线程在等待，如果有则表明account的锁已经被占用了，由于是第一次运行，account的就绪队列为空，所以线程a获得了锁，执行account.add方法。如果恰好在这个时候，线程b要执行account.withdraw方法，因为线程a已经获得了锁还没有释放，所以线程 b要进入account的就绪队列，等到得到锁后才可以执行。  
**一个线程执行临界区代码过程如下：  
1 获得同步锁  
2 清空工作内存  
3 从主存拷贝变量副本到工作内存  
4 对这些变量计算  
5 将变量从工作内存写回到主存  
6 释放锁**可见，synchronized既保证了多线程的并发有序性，又保证了多线程的内存可见性。  
  
  
**生产者/消费者模式**   
        生产者/消费者模式其实是一种很经典的线程同步模型，很多时候，并不是光保证多个线程对某共享资源操作的互斥性就够了，往往多个线程之间都是有协作的。  
        假设有这样一种情况，有一个桌子，桌子上面有一个盘子，盘子里只能放一颗鸡蛋，A专门往盘子里放鸡蛋，如果盘子里有鸡蛋，则一直等到盘子里没鸡蛋，B专门从盘子里拿鸡蛋，如果盘子里没鸡蛋，则等待直到盘子里有鸡蛋。其实盘子就是一个互斥区，每次往盘子放鸡蛋应该都是互斥的，A的等待其实就是主动放弃锁，B 等待时还要提醒A放鸡蛋。  
如何让线程主动释放锁  
很简单，调用锁的wait()方法就好。wait方法是从Object来的，所以任意对象都有这个方法。看这个代码片段：

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. Object lock=new Object();//声明了一个对象作为锁
2. synchronized (lock) {
3. balance = balance - num;
4. //这里放弃了同步锁，好不容易得到，又放弃了
5. lock.wait();
6. }

如果一个线程获得了锁lock，进入了同步块，执行lock.wait()，那么这个线程会进入到lock的阻塞队列。如果调用 lock.notify()则会通知阻塞队列的某个线程进入就绪队列。  
声明一个盘子，只能放一个鸡蛋

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. import java.util.ArrayList;
2. import java.util.List;
4. public class Plate {
6. List<Object> eggs = new ArrayList<Object>();
8. public synchronized Object getEgg() {
9. if (eggs.size() == 0) {
10. try {
11. wait();
12. } catch (InterruptedException e) {
13. }
14. }
16. Object egg = eggs.get(0);
17. eggs.clear();// 清空盘子
18. notify();// 唤醒阻塞队列的某线程到就绪队列
19. System.out.println("拿到鸡蛋");
20. return egg;
21. }
23. public synchronized void putEgg(Object egg) {
24. if (eggs.size() > 0) {
25. try {
26. wait();
27. } catch (InterruptedException e) {
28. }
29. }
30. eggs.add(egg);// 往盘子里放鸡蛋
31. notify();// 唤醒阻塞队列的某线程到就绪队列
32. System.out.println("放入鸡蛋");
33. }
35. static class AddThread extends Thread{
36. private Plate plate;
37. private Object egg=new Object();
38. public AddThread(Plate plate){
39. this.plate=plate;
40. }
42. public void run(){
43. for(int i=0;i<5;i++){
44. plate.putEgg(egg);
45. }
46. }
47. }
49. static class GetThread extends Thread{
50. private Plate plate;
51. public GetThread(Plate plate){
52. this.plate=plate;
53. }
55. public void run(){
56. for(int i=0;i<5;i++){
57. plate.getEgg();
58. }
59. }
60. }
62. public static void main(String args[]){
63. try {
64. Plate plate=new Plate();
65. Thread add=new Thread(new AddThread(plate));
66. Thread get=new Thread(new GetThread(plate));
67. add.start();
68. get.start();
69. add.join();
70. get.join();
71. } catch (InterruptedException e) {
72. e.printStackTrace();
73. }
74. System.out.println("测试结束");
75. }
76. }

  执行结果：

Html代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. 放入鸡蛋
2. 拿到鸡蛋
3. 放入鸡蛋
4. 拿到鸡蛋
5. 放入鸡蛋
6. 拿到鸡蛋
7. 放入鸡蛋
8. 拿到鸡蛋
9. 放入鸡蛋
10. 拿到鸡蛋
11. 测试结束

声明一个Plate对象为plate，被线程A和线程B共享，A专门放鸡蛋，B专门拿鸡蛋。假设  
1 开始，A调用plate.putEgg方法，此时eggs.size()为0，因此顺利将鸡蛋放到盘子，还执行了notify()方法，唤醒锁的阻塞队列的线程，此时阻塞队列还没有线程。  
2 又有一个A线程对象调用plate.putEgg方法，此时eggs.size()不为0，调用wait()方法，自己进入了锁对象的阻塞队列。  
3 此时，来了一个B线程对象，调用plate.getEgg方法，eggs.size()不为0，顺利的拿到了一个鸡蛋，还执行了notify()方法，唤醒锁的阻塞队列的线程，此时阻塞队列有一个A线程对象，唤醒后，它进入到就绪队列，就绪队列也就它一个，因此马上得到锁，开始往盘子里放鸡蛋，此时盘子是空的，因此放鸡蛋成功。  
4 假设接着来了线程A，就重复2；假设来料线程B，就重复3。   
整个过程都保证了放鸡蛋，拿鸡蛋，放鸡蛋，拿鸡蛋。

**volatile关键字**   
       volatile是java提供的一种同步手段，只不过它是轻量级的同步，为什么这么说，因为**volatile只能保证多线程的内存可见性，不能保证多线程的执行有序性。**而最彻底的同步要保证有序性和可见性，例如synchronized。任何被volatile修饰的变量，都不拷贝副本到工作内存，任何修改都及时写在主存。因此对于Valatile修饰的变量的修改，所有线程马上就能看到，但是volatile不能保证对变量的修改是有序的。什么意思呢？假如有这样的代码：

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. public class VolatileTest{
2. public volatile int a;
3. public void add(int count){
4. a=a+count;
5. }
6. }

        当一个VolatileTest对象被多个线程共享，a的值不一定是正确的，因为a=a+count包含了好几步操作，而此时多个线程的执行是无序的，因为没有任何机制来保证多个线程的执行有序性和原子性。volatile存在的意义是，**任何线程对a的修改，都会马上被其他线程读取到，因为直接操作主存，没有线程对工作内存和主存的同步**。所以，volatile的使用场景是有限的，在有限的一些情形下可以使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全,必须同时满足下面两个条件:  
1)对变量的写操作不依赖于当前值。  
2)该变量没有包含在具有其他变量的不变式中   
volatile只保证了可见性，所以Volatile适合直接赋值的场景，如

Java代码 [复制代码](http://www.iteye.com/topic/806990) [收藏代码](javascript:void())

1. public class VolatileTest{
2. public volatile int a;
3. public void setA(int a){
4. this.a=a;
5. }
6. }

在没有volatile声明时，多线程环境下，a的最终值不一定是正确的，因为this.a=a;涉及到给a赋值和将a同步回主存的步骤，这个顺序可能被打乱。如果用volatile声明了，读取主存副本到工作内存和同步a到主存的步骤，相当于是一个原子操作。所以简单来说，volatile适合这种场景：**一个变量被多个线程共享，线程直接给这个变量赋值。这是一种很简单的同步场景，这时候使用volatile的开销将会非常小。站内很多人都问我，所谓线程的“工作内存”到底是个什么东西？有的人认为是线程的栈，其实这种理解是不正确的。看看JLS（java语言规范）对线程工作内存的描述，线程的working memory只是cpu的寄存器和高速缓存的抽象描述。**

      可能 很多人都觉得莫名其妙，说JVM的内存模型，怎么会扯到cpu上去呢？在此，我认为很有必要阐述下，免得很多人看得不明不白的。先抛开java虚拟机不谈，我们都知道，现在的计算机，cpu在计算的时候，并不总是从内存读取数据，它的数据读取顺序优先级是：寄存器－高速缓存－内存。线程耗费的是CPU，线程计算的时候，原始的数据来自内存，在计算过程中，有些数据可能被频繁读取，这些数据被存储在寄存器和高速缓存中，当线程计算完后，这些缓存的数据在适当的时候应该写回内存。当个多个线程同时读写某个内存数据时，就会产生多线程并发问题，涉及到三个特性：原子性，有序性，可见性。在《线程安全总结》这篇文章中，为了理解方便，我把原子性和有序性统一叫做“多线程执行有序性”。支持多线程的平台都会面临这种问题，运行在多线程平台上支持多线程的语言应该提供解决该问题的方案。

       那么，我们看看JVM，JVM是一个虚拟的计算机，它也会面临多线程并发问题，java程序运行在java虚拟机平台上，java程序员不可能直接去控制底层线程对寄存器高速缓存内存之间的同步，那么java从语法层面，应该给开发人员提供一种解决方案，这个方案就是诸如 **synchronized,** **volatile,锁机制（如同步块，就绪队列，阻塞队列）等等。这些方案只是语法层面的，但我们要从本质上去理解它，不能仅仅知道一个synchronized 可以保证同步就完了。**   在这里我说的是jvm的内存模型，是动态的，面向多线程并发的，沿袭JSL的“working memory”的说法，只是不想牵扯到太多底层细节，因为 《线程安全总结》这篇文章意在说明怎样从语法层面去理解java的线程同步，知道各个关键字的使用场景。

      今天有人问我，那java的线程不是有栈吗？难道栈不是工作内存吗？工作内存这四个字得放到具体的场景中描述，方能体现它具体的意义，在描述JVM的线程同步时，工作内存指的是寄存器和告诉缓存的抽象描述，具体请自行参阅JLS。上面讲的都是动态的内存模型，甚至已经超越了JVM的范围，那么JVM的内存静态存储是怎么划分的？今天还有人问我，jvm的内存模型不是有eden区吗？也不见你提起。我跟他说，这是两个角度去看的，甚至是两个不同的范围，动态的线程同步的内存模型，涵盖了cpu，寄存器，高速缓存，内存；JVM的静态内存储模型只是一种对内存的物理划分而已，它只局限在内存，而且只局限在JVM的内存。那些什么线程栈，eden区都仅仅在JVM内存。

      说说JVM的线程栈和有个朋友反复跟我纠结的eden区吧。JVM的内存，被划分了很多的区域：

**1.程序计数器**  
每一个Java线程都有一个程序计数器来用于保存程序执行到当前方法的哪一个指令。  
**2.线程栈**  
线程的每个方法被执行的时候，都会同时创建一个帧（Frame）用于存储本地变量表、操作栈、动态链接、方法出入口等信息。每一个方法的调用至完成，就意味着一个帧在VM栈中的入栈至出栈的过程。如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果VM栈可以动态扩展（VM Spec中允许固定长度的VM栈），当扩展时无法申请到足够内存则抛出OutOfMemoryError异常。  
**3.本地方法栈  
4.堆**  
每个线程的栈都是该线程私有的，堆则是所有线程共享的。当我们new一个对象时，该对象就被分配到了堆中。但是堆，并不是一个简单的概念，堆区又划分了很多区域，为什么堆划分成这么多区域，这是为了JVM的内存垃圾收集，似乎越扯越远了，扯到垃圾收集了，现在的jvm的gc都是按代收集，堆区大致被分为三大块：新生代，旧生代，持久代（虚拟的）；新生代又分为eden区，s0区，s1区。新建一个对象时，基本小的对象，生命周期短的对象都会放在新生代的eden区中，eden区满时，有一个小范围的gc（minor gc），整个新生代满时，会有一个大范围的gc（major gc），将新生代里的部分对象转到旧生代里。  
**5.方法区**  
其实就是永久代（Permanent Generation），方法区中存放了每个Class的结构信息，包括常量池、字段描述、方法描述等等。VM Space描述中对这个区域的限制非常宽松，除了和Java堆一样不需要连续的内存，也可以选择固定大小或者可扩展外，甚至可以选择不实现垃圾收集。相对来说，垃圾收集行为在这个区域是相对比较少发生的，但并不是某些描述那样永久代不会发生GC（至 少对当前主流的商业JVM实现来说是如此），这里的GC主要是对常量池的回收和对类的卸载，虽然回收的“成绩”一般也比较差强人意，尤其是类卸载，条件相当苛刻。  
**6.常量池**  
 Class文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量表(constant\_pool table)，用于存放编译期已可知的常量，这部分内容将在类加载后进入方法区（永久代）存放。但是Java语言并不要求常量一定只有编译期预置入Class的常量表的内容才能进入方法区常量池，运行期间也可将新内容放入常量池（最典型的String.intern()方法）。

关于垃圾收集，在此不多说，流到垃圾收集那一章再详细说吧。关于java的同步，其实还有基于CPU原语的比较并交换的非阻塞算法（CAS），不过这个在java的并发包里已经实现了很多，因此关于这点，就留到java并发包那一章介绍吧。后面我会专门写一篇文章，JVM内存与垃圾收集。

[**JVM学习笔记（四）------内存调优**](http://www.blogjava.net/qileilove/archive/2011/11/15/363846.html)

　首先需要注意的是在对JVM内存调优的时候不能只看**[操作系统](http://www.51testing.com/html/44/n-248644.html" \t "_self)**级别[**Java**](http://www.51testing.com/html/44/n-248644.html)进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为GC过后这个值是不会变化的，因此内存调优的时候要更多地使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。

　　对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

　　● 旧生代空间不足  
　　调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象

　　● Pemanet Generation空间不足  
　　增大Perm Gen空间，避免太多静态对象

　　● 统计得到的GC后晋升到旧生代的平均大小大于旧生代剩余空间  
　　控制好新生代和旧生代的比例

　　● System.gc()被显示调用  
　　垃圾回收不要手动触发，尽量依靠JVM自身的机制

　　调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，下面来看看各部分比例不良设置会导致什么后果

　　1）新生代设置过小

　　一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

　　2）新生代设置过大

　　一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加

　　一般说来新生代占整个堆1/3比较合适

　　3）Survivor设置过小

　　导致对象从eden直接到达旧生代，降低了在新生代的存活时间

　　4）Survivor设置过大

　　导致eden过小，增加了GC频率

　　另外，通过-XX：MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

　　由上一篇博文JVM学习笔记（三）------内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM提供两种较为简单的GC策略的设置方式

　　1）吞吐量优先

　　JVM以吞吐量为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX：GCTimeRatio=n来设置

　　2）暂停时间优先

　　JVM以暂停时间为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX：MaxGCPauseRatio=n来设置

　最后汇总一下JVM常见配置

　　堆设置

　　-Xms：初始堆大小

　　-Xmx：最大堆大小

　　-XX：NewSize=n：设置年轻代大小

　　-XX：NewRatio=n：设置年轻代和年老代的比值。如：为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

　　-XX：SurvivorRatio=n：年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

　　-XX：MaxPermSize=n：设置持久代大小

　　收集器设置

　　-XX：+UseSerialGC：设置串行收集器

　　-XX：+UseParallelGC：设置并行收集器

　　-XX：+UseParalledlOldGC：设置并行年老代收集器

　　-XX：+UseConcMarkSweepGC：设置并发收集器

　　垃圾回收统计信息

　　-XX：+PrintGC

　　-XX：+PrintGCDetails

　　-XX：+PrintGCTimeStamps

　　-Xloggc：filename

　　并行收集器设置

　　-XX：ParallelGCThreads=n：设置并行收集器收集时使用的CPU数。并行收集线程数。

　　-XX：MaxGCPauseMillis=n：设置并行收集最大暂停时间

　　-XX：GCTimeRatio=n：设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

　　并发收集器设置

　　-XX：+CMSIncrementalMode：设置为增量模式。适用于单CPU情况。

-XX：ParallelGCThreads=n：设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

### Java内存分配、管理小结

2010-12-22 10:39:54|  分类： [默认分类](http://legend26.blog.163.com/blog/#m=0&t=1&c=fks_087068084095085064081080085095085085087070093086084069084) |  标签：[内存管理](http://legend26.blog.163.com/blog/#m=0&t=3&c=内存管理)  [jvm](http://legend26.blog.163.com/blog/" \l "m=0&t=3&c=jvm)  [变量](http://legend26.blog.163.com/blog/#m=0&t=3&c=变量)  [对象](http://legend26.blog.163.com/blog/#m=0&t=3&c=对象)  [java](http://legend26.blog.163.com/blog/#m=0&t=3&c=java)   |字号大中小 订阅

### 

|  |
| --- |
| 想写这篇总结酝酿了有个来月了，却始终感觉还差点什么东西，一直未敢动笔。         最近两天连夜奋战，重新整理下前面查阅的资料、笔记，还是决定将它写出来。         现在提出几个问题，如果都能熟练回答的大虾，请您飘过.如以往一样，我是小菜，本文自然也是针对小菜阶层的总结。 |

首先是概念层面的几个问题：

* Java中运行时内存结构有哪几种？
* Java中为什么要设计堆栈分离?
* Java多线程中是如何实现数据共享的？
* Java反射的基础是什么？

然后是运用层面：

* 引用类型变量和对象的区别？
* 什么情况下用局部变量，什么情况下用成员变量？
* 数组如何初始化？声明一个数组的过程中，如何分配内存？
* 声明基本类型数组和声明引用类型的数组，初始化时，内存分配机制有什么区？
* 在什么情况下，我们的方法设计为静态化，为什么？（上次胡老师问文奇，问的哑口无言，当时想回答，却老感觉表述不清楚，这里也简单说明一下）

好了，问题提完了，如果您都能一眼看出答案，那么，没有必要再浪费您宝贵的时间看下去了。

如果您还不太明白，请跟随我一路走下去。

Java中运行时内存结构

   1.1 方法区：

**方法区是系统分配的一个内存逻辑区域，是JVM在装载类文件时，用于存储类型信息的(类的描述信息)。**

方法区存放的信息包括：

            1.1.1**类的基本信息：**

1. 每个类的全限定名
2. 每个类的直接超类的全限定名(可约束类型转换)
3. 该类是类还是接口
4. 该类型的访问修饰符
5. 直接超接口的全限定名的有序列表

             1.1.2**已装载类的详细信息**：

1. 运行时常量池：

在方法区中，每个类型都对应一个常量池，存放该类型所用到的所有常量，常量池中存储了诸如文字字符串、final变量值、类名和方法名常量。它们以数组形式通过索引被访问，是外部调用与类联系及类型对象化的桥梁。（存的可能是个普通的字符串，然后经过常量池解析，则变成指向某个类的引用）

1. 字段信息：

字段信息存放类中声明的每一个字段的信息，包括字段的名、类型、修饰符。

字段名称指的是类或接口的实例变量或类变量，字段的描述符是一个指示字段的类型的字符串，如private A a=null;则a为字段名，A为描述符，private为修饰符

1. 方法信息：

类中声明的每一个方法的信息，包括方法名、返回值类型、参数类型、修饰符、异常、方法的字节码。

(在编译的时候，就已经将方法的局部变量、操作数栈大小等确定并存放在字节码中，在装载的时候，随着类一起装入方法区。)

|  |
| --- |
| 在运行时，JVM从常量池中获得符号引用，然后在运行时解析成引用项的实际地址，最后通过常量池中的全限定名、方法和字段描述符，把当前类或接口中的代码与其它类或接口中的代码联系起来。 |

1. 静态变量：

这个没什么好说的，就是类变量，类的所有实例都共享，我们只需知道，在方法区有个静态区，静态区专门存放静态变量和静态块。

1. 到类classloader的引用：到该类的类装载器的引用。
2. 到类class 的引用：虚拟机为每一个被装载的类型创建一个class 实例，用来代表这个被装载的类。

**由此我们可以知道反射的基础**：

|  |
| --- |
| 在装载类的时候，加入方法区中的所有信息，最后都会形成Class类的实例，代表这个被装载的类。方法区中的所有的信息，都是可以通过这个Class类对象反射得到。我们知道对象是类的实例，类是相同结构的对象的一种抽象。同类的各个对象之间，其实是拥有相同的结构（属性），拥有相同的功能（方法），各个对象的区别只在于属性值的不同。     同样的，我们所有的类，其实都是Class类的实例，他们都拥有相同的结构-----Field数组、Method数组。而各个类中的属性都是Field属性的一个具体属性值，方法都是Method属性的一个具体属性值。 |

 在运行时，JVM从常量池中获得符号引用，然后在运行时解析成引用项的实际地址，最后通过常量池中的全限定名、方法和字段描述符，把当前类或接口中的代码与其它类或接口中的代码联系起来。

1.2 Java栈

JVM栈是程序运行时单位，决定了程序如何执行，或者说数据如何处理。

在Java中，一个线程就会有一个线程的JVM栈与之对应，因为不过的线程执行逻辑显然不同，因此都需要一个独立的JVM栈来存放该线程的执行逻辑。

对方法的调用：

Java栈内存，以帧的形式存放本地方法的调用状态，包括方法调用的参数、局部变量、中间结果等（方法都是以方法帧的形式存放在方法区的），每调用一个方法就将对应该方法的方法帧压入Java 栈，成为当前方法帧。当调用结束(返回)时，就弹出该帧。

**这意味着：**

在方法中定义的一些基本类型的变量和引用变量都在方法的栈内存中分配。**当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后（方法执行完成后），Java 会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作它用**。--------**同时，因为变量被释放，该变量对应的对象，也就失去了引用，也就变成了可以被gc对象回收的垃圾。**

**因此我们可以知道成员变量与局部变量的区别：**

|  |
| --- |
| 局部变量，在方法内部声明，当该方法运行完时，内存即被释放。 成员变量，只要该对象还在，哪怕某一个方法运行完了，还是存在。 从系统的角度来说，声明局部变量有利于内存空间的更高效利用（方法运行完即回收）。 成员变量可用于各个方法间进行数据共享。 |

Java 栈内存的组成：  
局部变量区、操作数栈、帧数据区组成。  
（1）：局部变量区为一个以字为单位的数组，每个数组元素对应一个局部变量的值。调用方法时，将方法的局部变量组成一个数组，通过索引来访问。若为非静态方法，则加入一个隐含的引用参数this,该参数指向调用这个方法的对象。而静态方法则没有this参数。因此，对象无法调用静态方法。

**由此，我们可以知道，方法什么时候设计为静态，什么时候为非静态？**

|  |
| --- |
| 前面已经说过，对象是类的一个实例，各个对象结构相同，只是属性不同。 而静态方法是对象无法调用的。 所以，静态方法适合那些工具类中的工具方法，这些类只是用来实现一些功能，也不需要产生对象，通过设置对象的属性来得到各个不同的个体。 |

（2）：操作数栈也是一个数组，但是通过栈操作来访问。所谓操作数是那些被指令操作的数据。当需要对参数操作时如a=b+c,就将即将被操作的参数压栈，如将b 和c 压栈，然后由操作指令将它们弹出，并执行操作。虚拟机将操作数栈作为工作区。  
（3）：帧数据区处理常量池解析，异常处理等

1.3 java堆

      java的堆是一个运行时的数据区，用来存储数据的单元，存放通过new关键字新建的对象和数组，对象从中分配内存。  
      在堆中声明的对象，是不能直接访问的，必须通过在栈中声明的指向该引用的变量来调用。引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。

**由此我们可以知道,引用类型变量和对象的区别:**

|  |
| --- |
| 声明的对象是在堆内存中初始化的， 真正用来存储数据的。不能直接访问。  引用类型变量是保存在栈当中的，一个用来引用堆中对象的符号而已（指针）。 |

**堆与栈的比较**：  
JAVA堆与栈都是用来存放数据的，那么他们之间到底有什么差异呢？既然栈也能存放数据，为什么还要设计堆呢？

1.从存放数据的角度:

      前面我们已经说明:

      栈中存放的是基本类型的变量or引用类型的变量

       堆中存放的是对象or数组对象.

       在栈中，引用变量的大小为32位，基本类型为1-8个字节。  
       但是对象的大小和数组的大小是动态的，这也决定了堆中数据的动态性，因为它是在运行时动态分配内存的，生存期也不必在编译时确定，Java 的垃圾收集器会自动收走这些不再使用的数据。

2.**从数据共享的角度**:

    1).在单个线程类，栈中的数据可共享

    例如我们定义：

Java代码

1. int a=3;
2. int b=3;

    编译器先处理int a = 3；首先它会在栈中创建一个变量为a 的引用，然后查找栈中是否有3 这个值，如果没找到，就将3 存放进来，然后将a 指向3。接着处理int b = 3；在创建完b 的引用变量后，因为在栈中已经有3这个值，便将b 直接指向3。这样，就出现了a 与b 同时均指向3的情况。

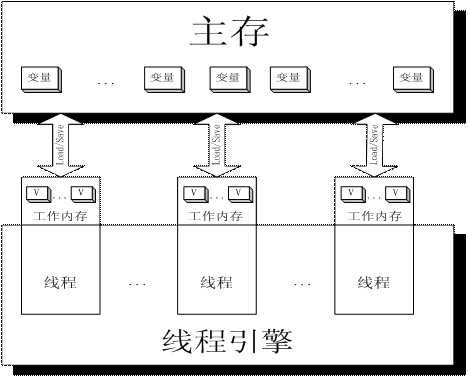
    而如果我们定义：

Java代码

1. Integer a=new Integer(3);//(1)
2. Integer b=new Integer(3);//(2)

   这个时候执行过程为：在执行(1)时，首先在栈中创建一个变量a，然后在堆内存中实例化一个对象，并且将变量a指向这个实例化的对象。在执行(2)时，过程类似，此时，在堆内存中，会有两个Integer类型的对象。

    2).**在进程的各个线程之间，数据的共享通过堆来实现**

        例：那么，在多线程开发中，我们的数据共享又是怎么实现的呢？  


  如图所示，堆中的数据是所有线程栈所共享的，我们可以通过参数传递，将一个堆中的数据传入各个栈的工作内存中，从而实现多个线程间的数据共享

（多个进程间的数据共享则需要通过网络传输了。）

3.从程序设计的的角度:

从软件设计的角度看，JVM栈代表了处理逻辑，而JVM堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

4.值传递和引用传递的真相

有了以上关于栈和堆的种种了解后，我们很容易就可以知道值传递和引用传递的真相：

|  |
| --- |
| 1.程序运行永远都是在JVM栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。  但是传引用的错觉是如何造成的呢?  在运行JVM栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。  但是当进入被调用方法时，被传递的这个引用的值，被程序解释(或者查找)到JVM堆中的对象，这个时候才对应到真正的对象。  如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是JVM堆中的数据。所以这个修改是可以保持的了。 |

最后：

从某种意义上来说对象都是由基本类型组成的。

|  |
| --- |
| 可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树(即非叶子节点)，基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点(即一个对象引用)，则可以修改这个节点下面的所有内容。 |

**其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别**。

面向对象的引入，只是改变了我们对待问题的思考方式，而更接近于自然方式的思考。

当我们把对象拆开，其实对象的属性就是数据，存放在JVM堆中;而对象的行为(方法)，就是运行逻辑，放在JVM栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。