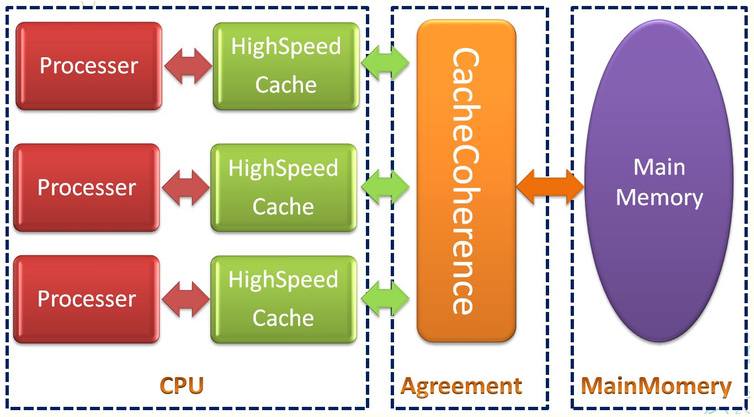
# 1 基础

## 1.1 背景

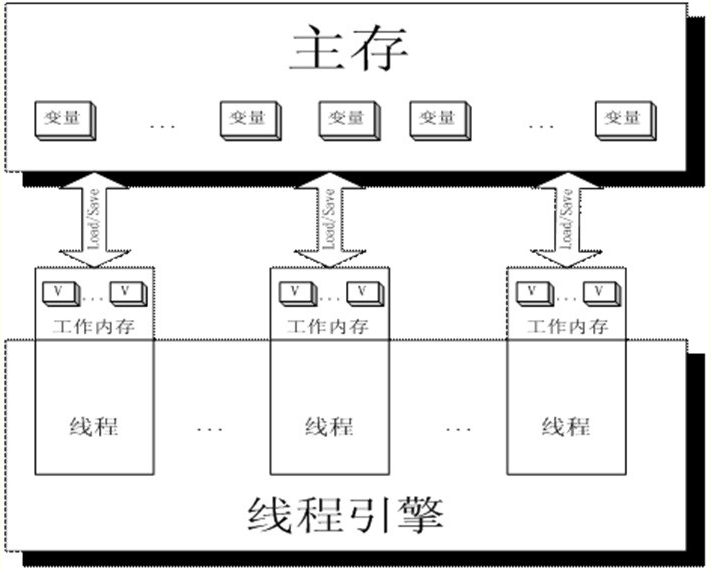
我们知道，计算机CPU和内存的交互是最频繁的，内存是我们的高速缓存区，用户磁盘和CPU的交互，而CPU运转速度越来越快，磁盘远远跟不上CPU的读写速度，才设计了内存，用户缓冲用户IO等待导致CPU的等待成本，但是随着CPU的发展，内存的读写速度也远远跟不上CPU的读写速度，因此，为了解决这一纠纷，CPU厂商在每颗CPU上加入了高速缓存，用来缓解这种症状。

同样，单核CPU的主频不可能无限制的增长，要想很多的提升新能，需要多个处理器协同工作基于高速缓存的存储交互很好的解决了处理器与内存之间的矛盾，也引入了新的问题：缓存一致性问题。在多处理器系统中，每个处理器有自己的高速缓存，而他们又共享同一块内存（下文成主存，main memory 主要内存），当多个处理器运算都涉及到同一块内存区域的时候，就有可能发生缓存不一致的现象。为了解决这一问题，需要各个处理器运行时都遵循一些协议，在运行时需要将这些协议保证数据的一致性。这类协议包括MSI、MESI、MOSI、Synapse、Firely、DragonProtocol等。如下图所示



## 1.2 Java

虚拟机内存模型中定义的访问操作与物理计算机处理的基本一致！如下图：



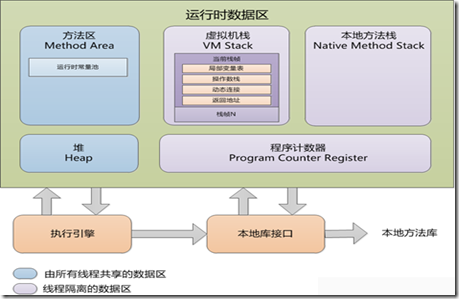
# 2 JVM 内存管理

## 2.1 运行时数据区域

Java虚拟机定义了若干种程序运行期间会使用到的运行时数据区。其中有一些会随着虚拟机启动而创建，随着虚拟机退出而销毁；另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

Java虚拟机所管理的内存将会包括以下几个运行时数据区域，如下图：

* GC主要是回收方法区和堆的内存空间



### 2.1.1 程序计数器（Program Counter Register）

亦可称为PC寄存器。

Java虚拟机可以支持多条线程同时执行（可参考《Java语言规范》第17章），每一条Java虚拟机线程都有自己的PC（Program Counter）寄存器。在任意时刻，一条Java虚拟机线程只会执行一个方法的代码，这个正在被线程执行的方法称为该线程的当前方法（Current Method）。

* 它是一块较小的内存空间，它的作用可以看做是当先线程所执行的字节码的信号指示器。
* 每一条JVM线程都有自己的PC寄存器，各条线程之间互不影响，独立存储，这类内存区域被称为“线程私有”内存
* 在任意时刻，一条JVM线程只会执行一个方法的代码。该方法称为该线程的当前方法（Current Method）
* 如果该方法是java方法，那PC寄存器保存JVM正在执行的字节码指令的地址
* 如果该方法是native，那PC寄存器的值是undefined。
* 此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

### 2.1.2 Java虚拟机栈

每一条Java虚拟机线程都有自己私有的Java虚拟机栈（Java Virtual Machine Stack），这个栈与线程同时创建，用于存储栈帧（Frames, §2.6）。Java虚拟机栈的作用与传统语言（例如C语言）中的栈非常类似，就是用于存储局部变量与一些过程结果的地方。另外，它在方法调用和返回中也扮演了很重要的角色。因为除了栈帧的出栈和入栈之外，Java虚拟机栈不会再受其他因素的影响，所以栈帧可以在堆中分配，Java虚拟机栈所使用的内存不需要保证是连续的。

* 与PC寄存器一样，Java虚拟机栈(Java Virtual Machine Stack)也是线程私有的。每一个JVM线程都有自己的java虚拟机栈，这个栈与线程同时创建，它的生命周期与线程相同。
* 虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。
* JVM stack 可以被实现成固定大小，也可以根据计算动态扩展。
* 如果采用固定大小的JVM stack设计，那么每一条线程的JVM Stack容量应该在线程创建时独立地选定。JVM实现应该提供调节JVM Stack初始容量的手段；如果采用动态扩展和收缩的JVM Stack方式，应该提供调节最大、最小容量的手段。
* 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；
* 如果JVM Stack可以动态扩展，但是在尝试扩展时无法申请到足够的内存时抛出OutOfMemoryError。

### 2.1.3 Java堆

在Java虚拟机中，堆（Heap）是可供各条线程共享的运行时内存区域，也是供所有类实例和数组对象分配内存的区域。在Java虚拟机启动时创建，是JVM所管理的内存中最大的一块。

Java堆在虚拟机启动的时候就被创建，它存储了被自动内存管理系统（Automatic Storage Management System，也即是常说的“Garbage Collector（垃圾收集器）”）所管理的各种对象，这些受管理的对象无需，也无法显式地被销毁。本规范中所描述的Java虚拟机并未假设采用什么具体的技术去实现自动内存管理系统。虚拟机实现者可以根据系统的实际需要来选择自动内存管理技术。Java堆的容量可以是固定大小的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。Java堆所使用的内存不需要保证是连续的。

* Java堆的容量可以是固定大小，也可以随着需求动态扩展（-Xms和-Xmx），并在不需要过多空间时自动收缩。
* Java堆所使用的内存不需要保证是物理连续的，只要逻辑上是连续的即可。
* JVM实现应当提供给程序员调节Java 堆初始容量的手段，对于可动态扩展和收缩的堆来说，则应当提供调节其最大和最小容量的手段。
* 如果堆中没有内存完成实例分配并且堆也无法扩展，就会抛OutOfMemoryError。

### 2.1.4 方法区

在Java虚拟机中，方法区（Method Area）是可供各条线程共享的运行时内存区域。方法区与传统语言中的编译代码储存区（Storage Area Of Compiled Code）或者操作系统进程的正文段（Text Segment）的作用非常类似，它存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法。

* 方法区在虚拟机启动的时候创建，虽然方法区是堆的逻辑组成部分，但是简单的虚拟机实现可以选择在这个区域不实现垃圾收集。
* 方法区的容量可以是固定大小的，也可以随着程序执行的需求动态扩展，并在不需要过多空间时自动收缩。
* 方法区在实际内存空间中可以是不连续的。
* Java虚拟机实现应当提供给程序员或者最终用户调节方法区初始容量的手段，对于可以动态扩展和收缩方法区来说，则应当提供调节其最大、最小容量的手段。
* 当方法区无法满足内存分配需求时就会抛OutOfMemoryError。

### 2.1.5 运行时常量池

运行时常量池（Runtime Constant Pool）是每一个类或接口的常量池（Constant\_Pool）的运行时表示形式，它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。运行时常量池扮演了类似传统语言中符号表（Symbol Table）的角色，不过它存储数据范围比通常意义上的符号表要更为广泛。每一个运行时常量池都分配在Java虚拟机的方法区之中，在类和接口被加载到虚拟机后，对应的运行时常量池就被创建出来。

在创建类和接口的运行时常量池时，可能会发生如下异常情况：

* 当创建类或接口的时候，如果构造运行时常量池所需要的内存空间超过了方法区所能提供的最大值，那Java虚拟机将会抛出一个OutOfMemoryError异常。

### 2.1.6 本地方法栈

Java虚拟机实现可能会使用到传统的栈（通常称之为“C Stacks”）来支持native方法（指使用Java以外的其他语言编写的方法）的执行，这个栈就是本地方法栈（Native Method Stack）。当Java虚拟机使用其他语言（例如C语言）来实现指令集解释器时，也会使用到本地方法栈。如果Java虚拟机不支持natvie方法，并且自己也不依赖传统栈的话，可以无需支持本地方法栈，如果支持本地方法栈，那这个栈一般会在线程创建的时候按线程分配。

Java虚拟机规范允许本地方法栈被实现成固定大小的或者是根据计算动态扩展和收缩的。如果采用固定大小的本地方法栈，那每一条线程的本地方法栈容量应当在栈创建的时候独立地选定。一般情况下，Java虚拟机实现应当提供给程序员或者最终用户调节虚拟机栈初始容量的手段，对于长度可动态变化的本地方法栈来说，则应当提供调节其最大、最小容量的手段。

本地方法栈可能发生如下异常情况：

* 如果线程请求分配的栈容量超过本地方法栈允许的最大容量时，Java虚拟机将会抛出一个StackOverflowError异常。
* 如果本地方法栈可以动态扩展，并且扩展的动作已经尝试过，但是目前无法申请到足够的内存去完成扩展，或者在建立新的线程时没有足够的内存去创建对应的本地方法栈，那Java虚拟机将会抛出一个OutOfMemoryError异常。

### 2.1.7直接内存（Direct Memory）

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError异常出现。

JDK1.4加的NIO中，ByteBuffer有个方法是allocateDirect(int capacity) ，这是一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括RAM及SWAP区或者分页文件）的大小及处理器寻址空间的限制。服务器管理员配置虚拟机参数时，一般会根据实际内存设置-Xmx等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError异常。

示例：

public class Test2 {

public static void main(String[] args) {

public Test2 t2 = new Test2();

//JVM将Test2类信息加载到方法区,new Test2()实例保存在堆区,Test2引用保存在栈区

}

}

## 2.2 OutOfMemoryError异常

### 2.2.1堆溢出

会出现java.lang.OutOfMemoryError，紧接着还会跟一条Java heap space，at…

先通过内存映像分析工具堆dump出来的堆快照分析，分清楚是内存泄漏还是内存溢出

对于内存泄漏，查一下GC Roots的引用链，掌握泄露对象的类型信息及GC Roots引用链的信息，定位出问题的地方

对于内存溢出，看看-Xms和-Xmx，看看能不能调大，从代码上检查是否有些对象生命周期过长等

### 2.2.2虚拟机栈和本地方法溢出

Hotspot中不区分虚拟机栈和本地方法栈，所以对于它来说-Xoss（设置本地方法栈大小）没啥用，栈容量只由-Xss来设置

一般来说，默认的Xss够用了，而且即使爆栈也会有足够的提示来解决，有的32位系统受限于寻址能力，因为多线程引起爆栈，有时候需要减少Xmx和减少栈容量来换取更多的线程

# 2 GC和内存分配策略

# 3 类加载机制

## 3.1类文件结构

3.1.1 class类文件结构

3.1.2

## 3.2 类加载过程

在Java语言里，类的加载、连接和初始化过程都是在程序运行期间完成的

# 5 Java内存模型

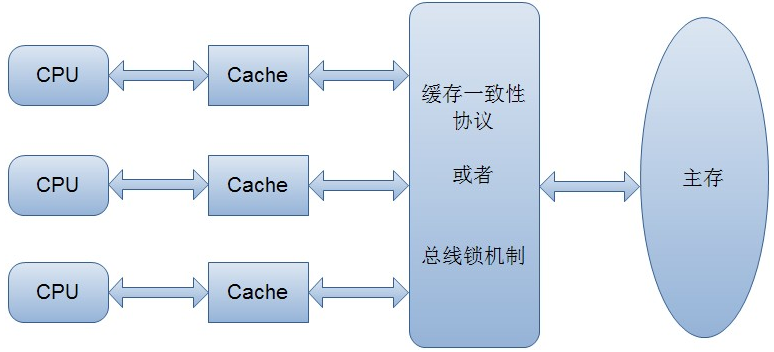
### 5.1 什么是内存模型

内存模型简称为JMM(Java Memory Model)，JMM规定了JVM必须遵循一组最小保证，规定了对变量的写入操作在何时将对于其它线程可见。JMM在设计时就在可测性和程序的易于开发性进行了权衡，从而在各主流的处理器体系架构上能够实现高性能的JVM。

**平台内存模型**

在共享内存的多处理器体系结构中，每个处理器都拥有自己的缓存，并且定期地与主存进行协调。在不同的处理器架构提供了不同级别的缓存一致性（Cache Coherence），其中一部分提供了最小的保证，即允许不同的处理器在任意时刻从同一个存储位置上看到不同的值。

Java提供了自己的内存模型，JVM通过在适当的位置上插入内存栅栏来屏蔽在JMM与底层平台内存模型之间的差异。



**重排序**

在没有充分的同步的程序中，交替执行不同线程的操作将导致不正确的结果。更糟糕的是，JMM还使得不同线程看到的操作执行顺序是不同的，要推断执行顺势几乎不可能。各种使操作延迟或者看似乱序执行的不同原因，都是重排序。

内存级的重排序会使得行为变得不可预测，而要确保程序正确哋使用同步却是非常容易的。同步将限制编译器、运行时和硬件对内存操作重排序的方式，从而实施重排序时不会破坏JVM提供的可见性保证。

### 2.9.2 内存模型简介

JMM是通过各种操作来定义的，包括对变量的读、写操作，监视器的加锁、释放锁操作，线程启动和合并操作。JMM为程序定义了一个偏序关系称为Happens-Before关系。想要保证操作B的线程看到操作A的结果（无论A、B是否发生在同一个线程中），A和B之间就必须满足Happens-Before关系。如果两个操作缺乏Happens-Before关系，那么JVM可以对它们任意重排序。Happens-Before的规则如下：

* **程序顺序规则：**如果程序中操作A在操作B之前，那么线程中A操作将在B操作之前执行。
* **监视器规则：**在监视器锁上的解锁操作必须在同一个监视器锁上的加锁操作之前执行。
* **volatile变量规则：**对volatile的变量写入操作必须在对改变量的读操作之前执行。
* **线程启动规则：**在线程上对Thread.Start的调用必须在该线程中执行任何操作之前执行。
* **线程结束规则：**线程中的任何操作都必须在其他线程监测到该线程已经结束之前执行，或者从Thread.join中成功返回，或者调用Thread.isAlive时返回false。
* **中断规则：**当一个线程在另外一个线程上调用interrupt时，必须在被中断线程检测到interrupt调用之前执行（通过抛出InterruptedException，或者调用isInterrupted和interrupt）。
* **终结器规则：**对象的构造函数必须在启动该对象的终结器之前执行完成。
* **传递性：**如果操作A在操作B之前执行，并且操作B在操作C之前执行，那么操作A必须在操作C之前执行。

虽然这些操作只满足偏序关系，但是同步操作，如锁的获取与释放等操作，以及volatile变量的读写操作，都满足全序关系。

4 JVM性能监控与故障处理

4.1 参数

4.2 调优方法

4.3 调优工具

4 JVM 监控