目录

[1 网络部分 2](#_Toc39737808)

[1.1 TCP通信 2](#_Toc39737809)

[1.1.1 两端数据字节流 2](#_Toc39737810)

[1.1.2 三次握手建立通信 3](#_Toc39737811)

[1.1.3 四次挥手断开通信 3](#_Toc39737812)

[1.1.4 滑动窗口与拥塞阻塞 4](#_Toc39737813)

[1.2 HTTP通信 5](#_Toc39737814)

[1.2.1 报文格式 5](#_Toc39737815)

[1.2.2 状态码 5](#_Toc39737816)

[1.3 HTTPS通信 5](#_Toc39737817)

[1.4 网页访问流程 5](#_Toc39737818)

[2 JAVA并发编程 6](#_Toc39737819)

[2.1 进程与线程 6](#_Toc39737820)

[2.2 并发编程的三个特性 6](#_Toc39737821)

[2.2.1 原子性 7](#_Toc39737822)

[2.2.2 可见性 7](#_Toc39737823)

[2.2.3 有序性 7](#_Toc39737824)

[2.3 JMM 8](#_Toc39737825)

[2.4 Volatile 9](#_Toc39737826)

[2.5 Synchronized 10](#_Toc39737827)

[2.6 JAVA中常用的锁 10](#_Toc39737828)

[2.6.1 锁的分类 10](#_Toc39737829)

[2.6.2 JAVA中常用的几类锁 11](#_Toc39737830)

[3 JAVA GC 14](#_Toc39737831)

[3.1 JAVA内存分区 14](#_Toc39737832)

[3.2 可达性分析 15](#_Toc39737833)

[3.3 分代理论 15](#_Toc39737834)

[3.4 回收算法 16](#_Toc39737835)

[3.4.1 标记-清除算法 16](#_Toc39737836)

[3.4.2 标记-复制算法 16](#_Toc39737837)

[3.4.3 标记-整理算法 17](#_Toc39737838)

[3.5 记忆集 17](#_Toc39737839)

[3.6 可达性分析的并发问题 17](#_Toc39737840)

[3.7 常见的GC 18](#_Toc39737841)

[3.7.1 Serial和Serial Old 18](#_Toc39737842)

[3.7.2 ParNew 19](#_Toc39737843)

[3.7.3 Parallel Scavenge和Parallel Old 19](#_Toc39737844)

[3.7.4 CMS 19](#_Toc39737845)

[3.7.5 G1 20](#_Toc39737846)

[4 反射 21](#_Toc39737847)

[4.1 反射思想 21](#_Toc39737848)

[4.2 具体方法 21](#_Toc39737849)

[5 动态代理 23](#_Toc39737850)

[5.1 代理模式 23](#_Toc39737851)

[5.2 静态代理 23](#_Toc39737852)

[5.3 动态代理 23](#_Toc39737853)

[6 AOP 26](#_Toc39737854)

[6.1 理论 26](#_Toc39737855)

[6.2 Spring AOP 26](#_Toc39737856)

# 网络部分

## TCP通信

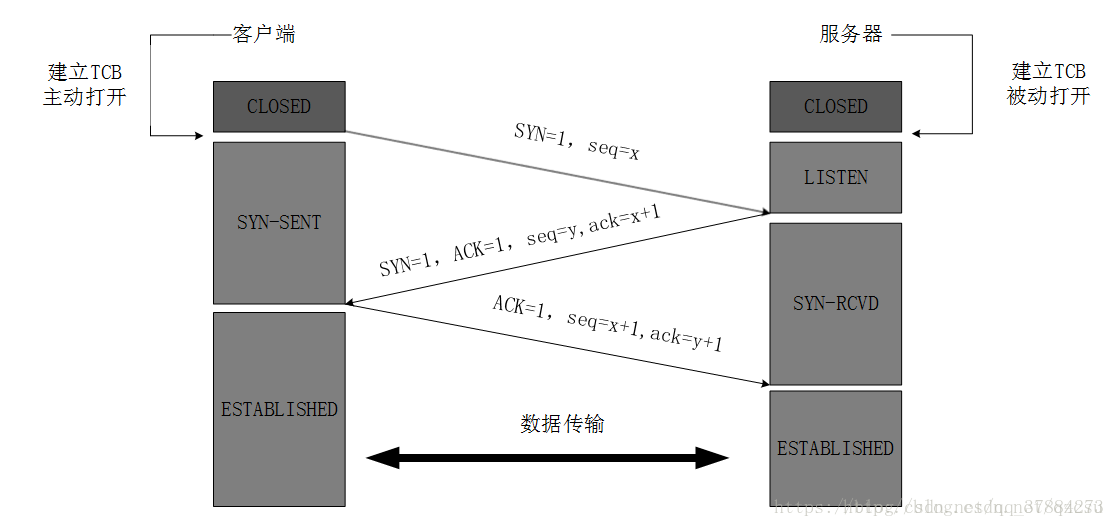
特点：在无序的IP层报文基础上实现了有序的数据传输，通过seq num和ack num字段解决了重传、乱序、漏包等问题。

### 两端数据字节流

1. 假设通信两端分别为A和B，两者都会维护一个字节数据序列，该序列由本端发送的数据组成。
2. 两者发送的报文都包含一个seq num和ack num
   1. Seq num：源端发送的这条报文中的数据部分在源端字节数据序列中的起始下标
   2. Ack num：目的端下条报文中数据部分在目的端字节数据序列中的起始下标

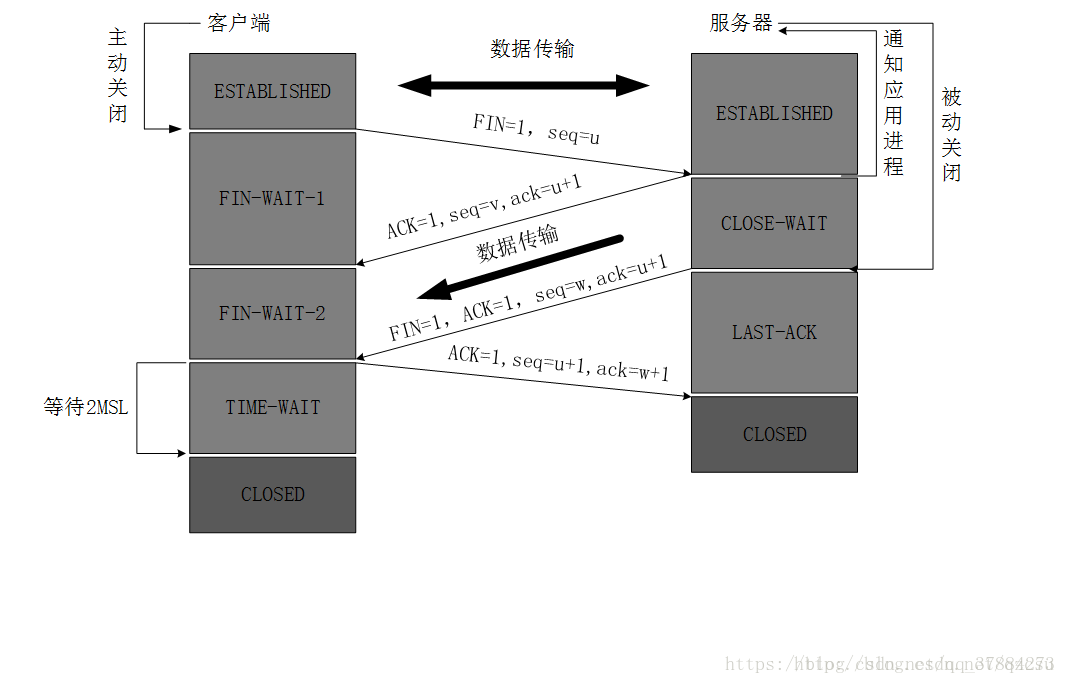
### 三次握手建立通信

1. 客户端发送建立连接请求
2. 服务端响应建立连接请求
3. 客户端发送确认报文



### 四次挥手断开通信

1. 客户端发送释放连接请求，停止发送数据，但还可以接收服务端数据
2. 服务端响应释放连接请求，进行CLOSE-WAIT（关闭等待）状态，等待上层应用完成最后的数据传输
3. 服务端所有数据传输完毕后，再发送释放连接报文，进入最后的LAST-ACK（最后确认）状态
4. 客户端发送确认报文，等待2MSL之后关闭连接；服务端收到客户端发送的确认报文后立刻关闭连接。
5. 最后等待2MSL时间是为了防止客户端发送确认报文出现问题(丢失或者错包)，服务端又发送了响应报文，客户端可能就必须重新发送确认报文再进行等待。



### 滑动窗口与拥塞阻塞

滑动窗口

1. 一般而言，TCP的确认应答策略中，必须等到收到上个数据段的ACK报文之后才能发送下一个数据段，这样效率太低
2. 滑动窗口是接收端的接收缓冲区，同时也是发生端的发送缓冲区。在TCP报文字段中指定，这样就可以一次接收对端的多个数据段。
3. 窗口大小指的是无需等待确认应答而可以继续发送数据的最大值。
4. 对于接收缓冲区中多个数据段，接收端会进行重排序，然后返回最大的ACK值，相当于同时确认了多个数据段正常接收了。

拥塞控制

1. 防止发生报文速度太快，超过了上层应用的处理速度
2. 阻塞条件：丢包(超时、收到连续3个相同的ACK报文)
3. 发送阻塞时，发送端调整发送窗口大小
   1. 发送阻塞，设置阈值为当前窗口的一半ssthresh = cur\_cwnd / 2
   2. 将当前窗口大小置1，Cur\_cwnd = 1
   3. 每收到一个确认报文，当前窗口大小翻倍Cur\_cwnd \*= 2
   4. 当达到阈值时，每收到一个确认报文，当前窗口大小加1

## HTTP通信

### 报文格式

请求报文：请求行、请求头、空行，数据(响应体)



响应报文：状态行，响应头，空行，数据(响应体)



### 状态码

1\*\* 信息，服务器收到请求，需要请求者继续执行操作

2\*\* 成功，操作被成功接收并处理

3\*\* 重定向，需要进一步的操作以完成请求

4\*\* 客户端错误，请求包含语法错误或无法完成请求

5\*\* 服务器错误，服务器在处理请求的过程中发生了错误

## HTTPS通信

在HTTP的基础上使用SSL协议对HTTP数据进行加密

## 网页访问流程

1. 通过域名获取IP，DNS缓存、查询DNS服务器
2. 建立TCP连接，三次握手
3. TLS协议协商，证书信任
4. 正常HTTP交互

# JAVA并发编程

## 进程与线程

进程：

1. 资源分配的最小单位，是操作系统进行资源分配和调度的一个独立单位，是线程的容器
2. 进程间资源独立，通过端口进行通信

线程：

1. 操作系统能够进行运算调度的最小单位，程序执行的基本单元，系统分配处理器时间资源的基本单元。
2. 同一个进程下的线程可以共享进程的资源，同时也有自己独立的资源
3. 线程间通信可以使用进程内的共享内存空间完成，不同线程可以通过访问和修改指定地址的数据完成通信。
4. 线程通信：共享内存(Java) 和 消息传递

线程实现：

1. 内核线程：1:1关系，一个线程对应一个轻量级进程，轻量级进程由OS内核管理，可以充分利用多核并行
2. 用户线程：1:N关系，由用户管理线程的所有操作，OS看不见这些线程，很难进行阻塞和处理器分配。
3. 混合实现：N:M关系，OS管理多个轻量级进程，每个轻量级进程对应多个线程
4. JAVA虚拟机中线程的实现方式取决于具体的操作系统支持哪种线程

线程创建和撤销消耗较少，进程通信麻烦

## 并发编程的三个特性

要想并发程序正确地执行，必须要保证原子性、可见性以及有序性

### 原子性

1. 原子性即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行，不可分割。
2. Java中的原子性操作
   1. 对基本数据类型的变量的读取和赋值操作是原子性操作
      1. Int i = 2; 原子操作
      2. Int j = i; 不是原子操作，先读取，后赋值
      3. i++;i+=1; 也不是原子操作，读取，加1，赋值
   2. Long和double类型的数据是64位，对于32位系统，读写操作可能分为两步进行，每次出理32位，所以不是原子操作；对于64位系统，就可能是原子操作，所以需要看JVM怎么实现。
   3. 引用类型的赋值JVM将其实现为了原子操作，因为其本质就是一个地址，和int类似。
      1. Int[] i; i=new int[]{1,2,3}; 是一个原子操作
      2. Int[] i; int[] j = new int[]{1,2,3}; i = j; 可能就不是一个原子操作了，先取出j的值再复制给i，是两步操作。

### 可见性

1. 一个线程修改了共享变量的值，其他线程都可以立刻得知这个修改
2. Java中volatile修饰的变量满足可见性特性
3. synchronized和Lock也能够保证可见性，线程在释放锁之前，会把共享变量值都刷回主存，但是synchronized和Lock的开销都更大

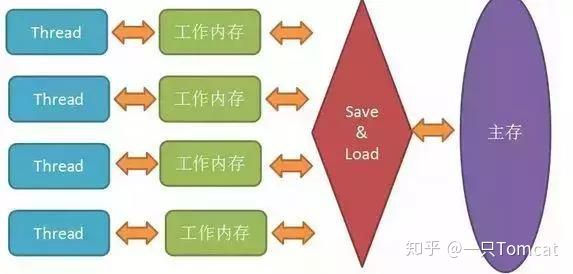
### 有序性

1. 程序执行的顺序按照代码的先后顺序执行
2. 真实环境下，从源码->中间码->汇编码->机器码中间的各部分都可能会发生指令重排操作，改变代码执行顺序
3. 指令重排虽然不会改变单线程的执行结果，但会影响到线程并发的正确性，需要使用volatile、synchronized、Lock进行特殊处理

## JMM

Java内存模型(JMM)，用于屏蔽硬件和操作系统的内存访问差异，让Java程序在各种平台上都能达到一致的内存访问效果。

JMM实现的内存访问机制如下

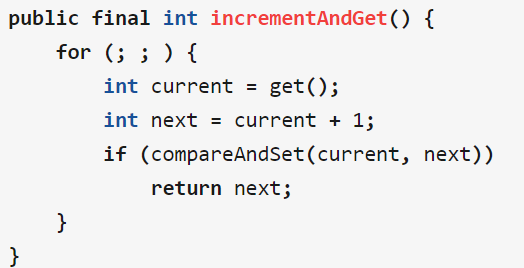


1. 主存相当于硬盘空间，数据都存储主存中
2. 工作/本地内存，线程独有
3. JMM中定义了8种基本原子操作：lock、unlock、read、load、use、assign、store、write
4. 数据读取时，先通过read+load从主存中将数据读取到线程的工作内存中，再使用use从工作内存加载至寄存器中进行运算
5. 数据写入时，先使用assign将数据写入工作内存，再使用strore+write从工作内存写入

JMM中会对指令进行重排序，因此也设计了一些规则保证重排序后程序执行的正确性：

1. as-if-serial
   1. 保证单线程执行情况下，程序执行结果不会有差别
2. happens-before规则
   1. 为了确保有序性设计的一些规则，对指令重排序加了很多限制条件
   2. 除了单线程执行结果不变外，还能保证正确同步的多线程程序的执行结果不被改变，由JMM来解决一些常见的并发问题，使得开发者不需要处理所有的并发情况。

## Volatile

1. 可以保证数据的可见性和有序性
   1. 可见性：从效果上看，volatile变量写到缓存中的值会直接刷新至主存(通过汇编指令lock实现，该指令会将本处理器缓存中的数据写入主存，并将其他处理器相应缓存数据置为无效)，读也是从主存中重新读取，当然中间也会使用工作内存。
   2. 有序性：对volatile变量操作指令前后加上相应的内存屏障指令，不允许其他指令与内存屏障内的指令进行重排序。也是lock指令完成的，因为lock将缓存写入内存，相当于之前的指令都执行完了，就形成了“指令重排序无法越过内存屏障”的效果。
2. Volatile只保证变量的单次操作(读/写)是原子性的，并不能保证线程安全
   1. 像i++这种操作，其实分为4条指令：get I; const 1; add I; put I;
   2. get i可以保证读取的是最新值，但在const 1; add I;指令执行时，其他线程可能对i进行了新的赋值，所以i就不是最新值了。
3. CAS：compare and set，当一个变量值是我们的预期值时，才对其进行更新。这样我们就能知道其他线程是否对变量进行修改。CAS操作具有原子性，是非阻塞的乐观锁。
   1. 先将数据从内存中取出current
   2. 对数据进行修改next
   3. 使用CAS方法，先比较当前内存中的值是否还是current，是就将next写入内存，否则就重新执行上述操作
   4. 
   5. 之所以是原子性是因为第3步复杂的CAS操作是CPU字节码级别的操作，而不是JAVA源码级别，所以能够保证原子性

## Synchronized

1. 通过JMM中的lock和unlock指令实现
2. 同步锁，使用一个对多线程可见的共享对象实例作为锁，对象实例有个标记字段锁对象指针指向一个monitor对象，该对象的访问是互斥的
3. 使用同一个锁的多个代码块，同一时刻只能有一个线程执行其中某一个代码块。这些共用公共资源的代码块叫做临界区。
4. 锁具有可重入性：一个线程获取锁之后还可以继续获取该锁，但获取多少次就得释放多少次，否则锁不会真正被释放
5. 获取锁实质上是访问某个对象实例，如果多线程都有该对象的引用，那么多线程就都可以使用该锁
6. 没有指定锁时，synchronized锁的是对象本身的实例；静态方法没有对象实例，所以锁的是JVM中该类的唯一class实例
7. 当线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中。当线程获取锁时，JMM 会把该线程对应的本地内存置为无效，从主存中重新加载数据。
8. Synchronized获取锁是阻塞式的，在无法获取到锁时一直阻塞，容易造成死锁

## JAVA中常用的锁

### 锁的分类

1. 公平锁和非公平锁：Synchronized、ReentrantLock
   1. 公平锁：多个线程按照申请锁的顺序来获取锁
   2. 非公平锁：运行“插队”，如果前面线程运行所需资源暂时无法满足，而后面线程所需运行资源都能满足的情况下，后面线程就能够提取申请获取锁，会造成饥饿现象(有的线程一直无法获取锁)。
   3. 默认都是非公平锁，能提高系统吞吐量
2. 可重入锁：Synchronized、ReentrantLock
3. 独享锁和共享锁
   1. 独享锁：Synchronized、ReentrantLock、ReadWriteLock的写锁
   2. 共享锁：多个进程可以同时获取，ReadWriteLock的读锁
4. 互斥锁、读写锁：独享锁和共享锁的具体实现
5. 乐观锁和悲观锁：两种并发编程理念，不是真的锁
   1. 悲观锁：认为并发程序一定会对共享数据进行写操作，需要显示的加锁编程，防止出现错误，并发代码只能顺序执行，阻塞同步。
   2. 乐观锁：认为并发程序很少会对共享数据进行写操作，不加锁编程，如果发生了冲突，再进行补偿措施。比如使用尝试更新（CAS）方法进行赋值，非阻塞同步。
6. 分段锁：一个集合分为多段，每一段都有一个独立的锁，多个线程可以同时操作这个集合，但集合中的每一段同一时刻只能有一个线程操作。
7. 自旋锁：
   1. 正常线程B等待线程A释放锁时，需要OS在用户态和内核态之间切换，然后阻塞线程B，消耗较大
   2. 自旋锁则是让线程B采用循环的方式去尝试获取锁，如果达到一定限制还没有获取锁，再进行阻塞。
   3. 自旋的缺点是循环会占用CPU，无法用于其他线程的处理
8. 偏向锁、轻量级锁、重量级锁
   1. 偏向锁：只有一个线程使用该锁，那么该线程会自动获取锁，降低获取锁的代价。
   2. 轻量级锁：当有另一个线程使用该锁时，锁从偏向锁转为轻量级锁，另一个线程以自旋的方式尝试获取锁，不会阻塞。
   3. 重量级锁：轻量级锁线程自旋一定次数之后还没有获取锁，就会进行阻塞，锁转变为重量级锁，所有申请线程都会阻塞。

### JAVA中常用的几类锁

Synchronized

1. Java语言直接提供的关键字，每个对象都可以作为锁，资源耗费较多
2. 在等待获取锁时必须阻塞
3. 可重入锁

ReentrantLock

1. 创建lock对象后，需要自行获取和释放锁
   1. Lock lock = new ReentrantLock();
   2. lock.lock(); lock.unlock();
2. 多了尝试获取锁功能lock.tryLock(time)，返回True或False，
   1. True代表获取到锁，执行正常操作，然后释放锁
   2. Fasle代表获取不到锁，可以执行其他操作，不需要阻塞，更安全。
3. 需要借助Condition对象实现wait和notify功能
4. 可重入锁

ReadWriteLock

1. 读写锁，读锁是共享锁，可以多个线程共同获取，写锁是独享锁，同一时刻只有一个线程能够获取
   1. ReadWriteLock rwlock = new ReentrantReadWriteLock();
   2. Lock rlock = rwlock.readLock(); //读锁
   3. Lock wlock = rwlock.writeLock(); //写锁
   4. rlock.lock(); rlock.unlock(); //需要自己获取和释放
2. 多个线程获取锁之间的关系
   1. 一个线程获取读锁之后，其他线程也可以获取读锁，但不能获取写锁
   2. 一个线程获取写锁之后，其他线程不能获取锁

|  |  |  |
| --- | --- | --- |
|  | 读锁 | 写锁 |
| 读锁 | 允许 | 不允许 |
| 写锁 | 不允许 | 不允许 |

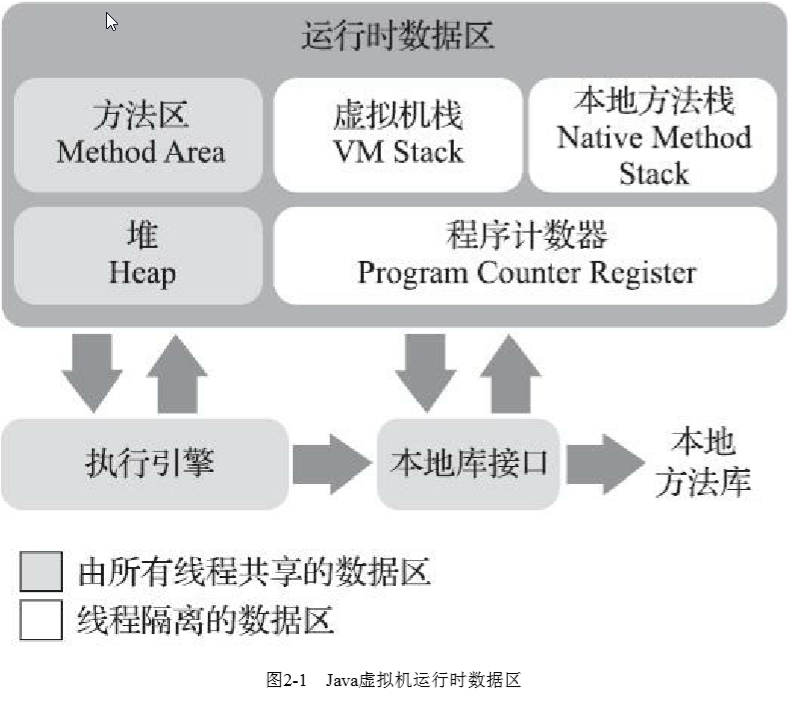
1. 适合多读少写的场景，想要获取写锁必须要等其他读/写锁全都释放
2. 读锁是一个悲观读锁，写锁必须等待读锁释放才能获取
3. 可重入锁

StampedLock

1. 读写锁，有三种锁：悲观读锁、写锁、乐观读锁
   1. StampedLock stampedLock = new StampedLock();
   2. long rstamp = stampedLock.readLock(); //悲观读锁
   3. long wstamp = stampedLock.writeLock(); //写锁
   4. long orstamp = stampedLock.tryOptimisticRead(); //乐观读锁
   5. stampedLock.unlockRead(rstamp); //释放悲观读锁
   6. stampedLock.unlockWrite(wstamp); //释放写锁
2. 与ReadWriteLock相比多了乐观读锁，可以在乐观读锁未释放时获取写锁。乐观读锁本质上不是个锁，而是一种检查机制，所以也不需要释放。
3. 锁的类型不再是lock，而是一个long，代表着锁内部的版本号
4. 在获取乐观读锁之后，可以通过validate()检验其他线程是否获取过写锁，因为版本号会发生变化
   1. stampedLock.validate(stamp)
5. 不可重入锁

# JAVA GC

## JAVA内存分区



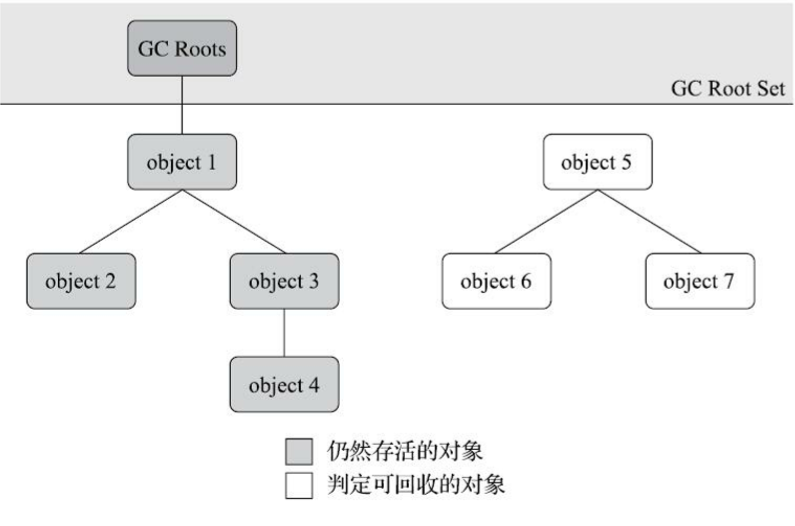
线程共享区域：方法区、堆

线程私有区域：虚拟机栈、本地方法栈、程序计数器

各部分介绍：

1. 方法区
   1. 存储已被虚拟机加载的类型信息、**常量、静态变量**、即时编译的代码缓存
   2. 常量除了一开始加载时放入方法区中的运行时常量池，还可以在运行期间添加，比如String类的intern()方法
2. 堆
   1. 存放所有的**对象实例**
   2. 是垃圾收集管理的内存区域，会为不同的线程分配不同的区域存储线程内的私有对象实例，方便回收
3. 虚拟机栈
   1. 用于存放线程内方法调用的形参
4. 本地方法栈
   1. 用于存放线程调用的本地方法的形参
5. 程序计数器
   1. 字节码指令地址
   2. 执行本地方法时为空

## 可达性分析

1. 用于判断对象是否存活
2. 先确定GC Roots，这些对象在当前时刻一定是存活的，包括
   1. 常量和静态变量
   2. 虚拟机栈中对象，即所有线程调用方法使用到的参数、局部变量等
   3. JAVA虚拟机内部的引用对象，各种基本Class对象
   4. 同步锁持有对象
   5. 本地方法使用的对象
3. 根据对象间的引用关系构造引用链
   1. 
4. 如果存在一个GC Root能够到达该对象，则该对象是存活的，否则可以判为可回收对象

## 分代理论

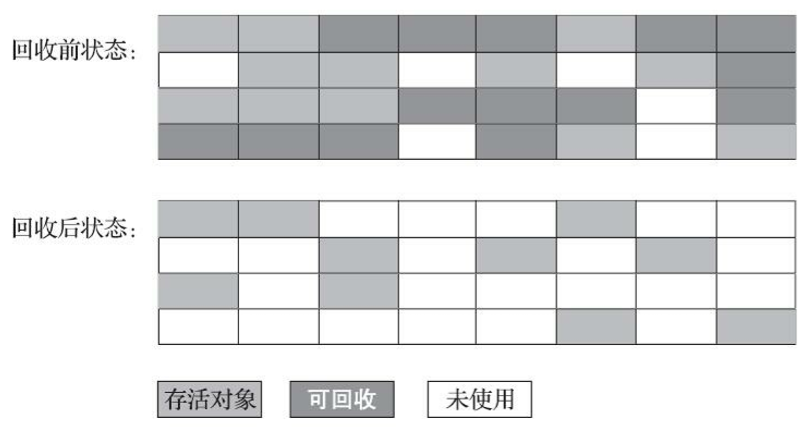
1. 垃圾收集器应该根据对象的“年龄”将JAVA堆分为两个不同的区域：新生代和老年代
2. 新创建的对象年龄较小，放在新生代中
   1. 新生代中的对象存活时间较短，大都撑不过一次垃圾回收，朝生夕灭，
   2. 这部分区域能够以较低的代价收回大量空间，垃圾回收频率较高
3. 撑过多轮垃圾回收的对象都是难以消亡的对象，放在老年区
   1. 老年区对象存活时间较长，可以使用较低的频率回收这部分内存

## 回收算法

标记-清除：不需要移动对象，可以与用户线程并发执行

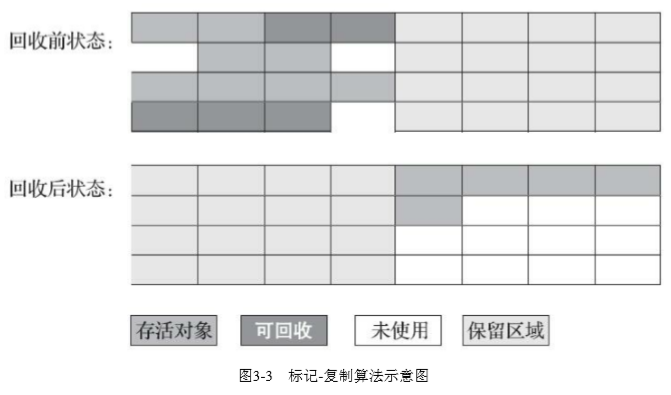
标记-复制/整理：需要移动对象，必须停止用户线程

### 标记-清除算法



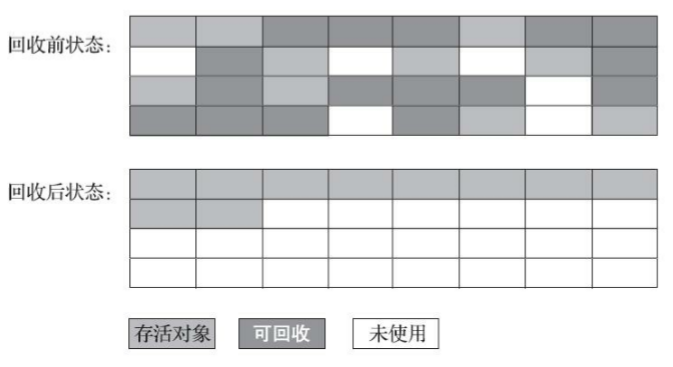
1. 直接将标记为可回收的空间进行回收，其他空间不处理
2. 回收效率较快，但会导致内存碎片化问题，使得内存分配变得困难

### 标记-复制算法



1. 将内存分为两块：Eden和Survivor
2. 真正使用的是Eden区域内存，垃圾回收时对其中的内存块进行标记，将存活的对象先复制到Survivor区域，然后将Eden区域整体清空
3. 回收效率较快，但增加了复制成本，适用于新生代，因为新生代存活对象较少
4. 同时分配内存时真正使用的内存区域是Eden，导致可分配内存只有原来的一半
5. 通常做法是将Eden和Survivor区域大小设为8:1，Survivor空间不足时将这些存活对象放入老年区空间

### 标记-整理算法



1. 先标记清楚，然后整体移动到一端
2. 解决了空间碎片化问题，但移动成本较高，与标记-清除相比，垃圾回收时停顿时间更长，但程序的整体吞吐量更高

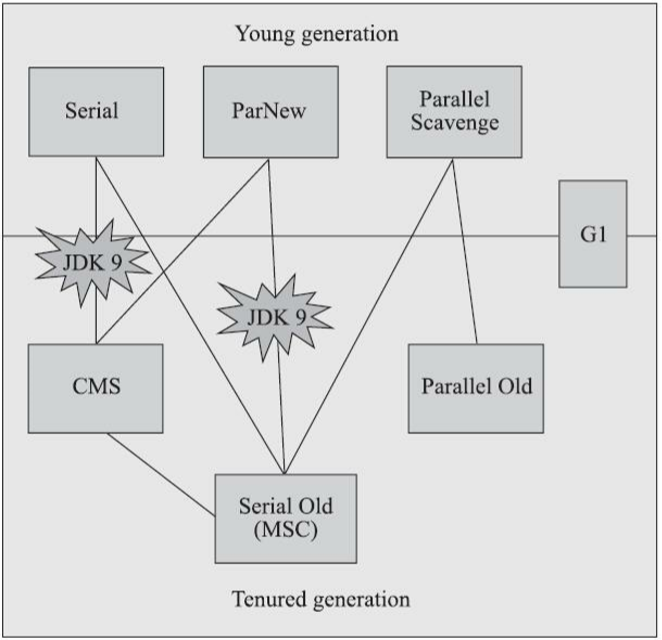
## 记忆集

1. 在回收新生代时，有的新生代对象会被老年代中的对象引用，因此也需要扫描老年代，但是扫描完整的老年代成本较大，因此设计了记忆集
2. 记忆集用于存储非收集区域是否存在指向收集区域的引用
3. 根据记忆集能够查出老年代哪些区域有指向新生代的引用，就只用扫描这些区域即可
4. 记忆集根据其记录的内存块的精度分为：
   1. 字长精度：精确到地址
   2. 对象精度：精确到对象
   3. 卡精度：精确到一块内存区域，也叫做**卡表**，最常用

## 可达性分析的并发问题

1. 在进行可达性分析时，如果用户线程没有停止，用户可能会修改对象之间的引用关系，则会出现小概率的并发问题
   1. 该回收的对象没有回收：这种情况可以忽略，留给下次回收
   2. 不该回收的对象回收了：这种情况必须处理，不然程序会崩溃
2. 针对第二个问题有两种解决方法
   1. 增量更新： 在可达性分析扫描过程中，用户如果修改了一些对象之间的引用关系，则进行适当的回溯，重新扫描部分节点
   2. 原始快照：记录下可达性分析时用户所做的改动，先进行第一遍可达性分析，然后根据用户的改动进行第二遍分析

## 常见的GC



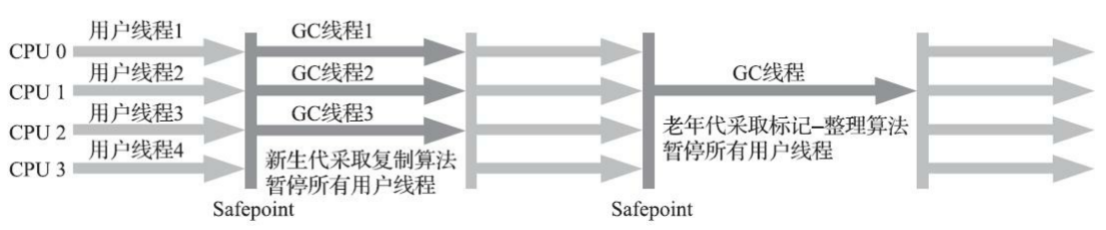
### Serial和Serial Old

1. Serial和Serial Old进行垃圾回收时必须暂停所有用户进程
2. Serial处理新生代，使用标记复制
3. Serial Old处理老年代，使用标记整理



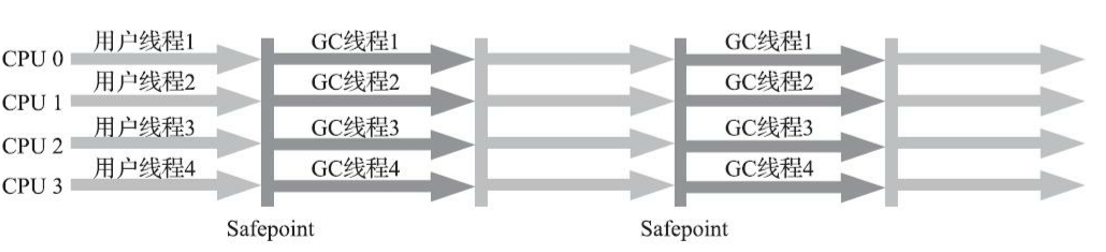
### ParNew

1. 在Serial的基础上进行改进，使用多线程并行进行垃圾回收，其他没有变化



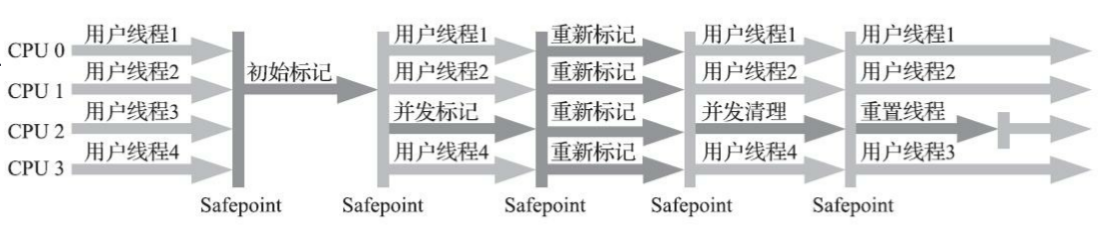
### Parallel Scavenge和Parallel Old

1. Parallel Scavenge处理新生代，Parallel Old处理老生代，都是并行处理
2. 可以设置最大垃圾回收时间控制停顿时间

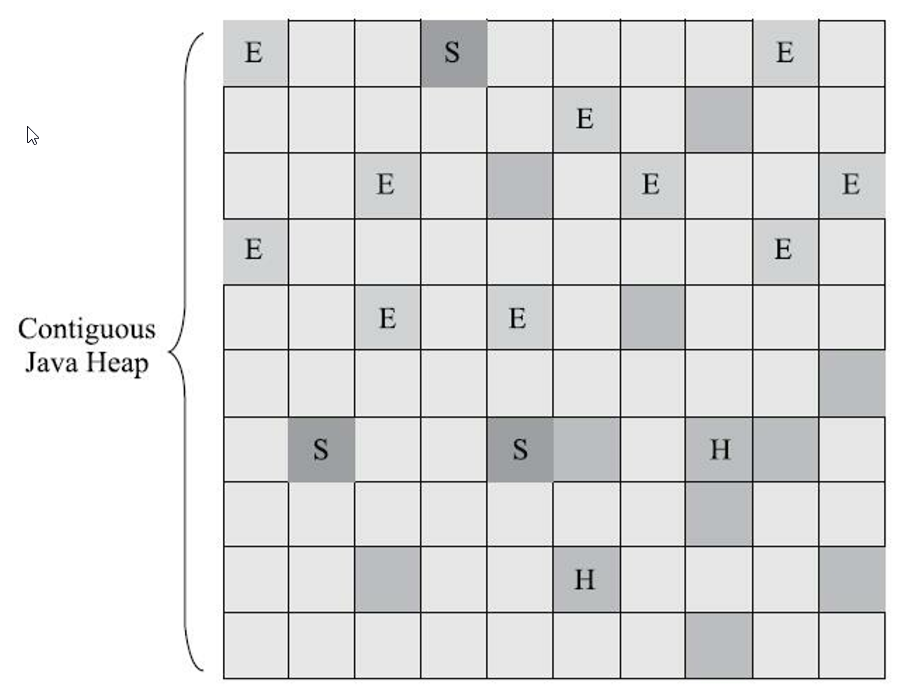


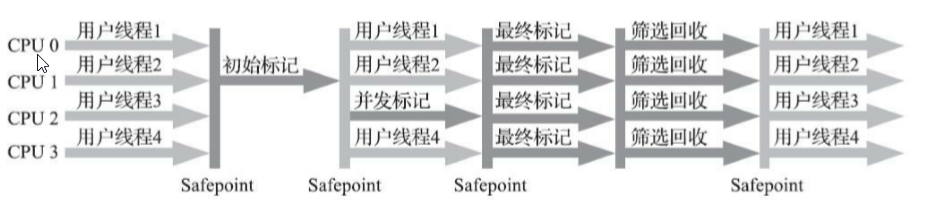
### CMS

1. 用于老年代，使用标记-清除算法
2. 分为四个阶段
   1. 初始标记：阻塞用户线程，只标记GC Roots能直接连到的对象，耗时较少
   2. 并发标记：并发式，和用户进程一起执行，耗时较长
   3. 重新标记：阻塞式，修正之前提到的并发标记问题，耗时较少
   4. 并发清理：并发式，因为使用标记-清除算法，不需要移动对象，所以不需要暂停用户进程



### G1

1. Garbage First，用于所有堆内存，分块回收理念，标记-复制
2. 将堆内存分为若干个大小相等的独立区域Region，每一个Region都可以根据需要充当新生代的Eden、Survivor、老年代和Humongous(专门用于存储大对象)这4种类型
   1. 
3. 处理流程和CMS比较类似，不过清理时会选择垃圾较多，回收收益较大的几个块进行回收，而不是整个内存
   1. 初始标记：阻塞式
   2. 并发标记：并发式
   3. 最终标记：阻塞式，进行并发问题修正
   4. 筛选回收：阻塞式，根据收益排名选择最优的Region进行回收，将还存活的对象复制到空的Region(Survivor)，然后清空整个旧Region



# 反射

## 反射思想

1. 反射是Java中的一种运行时机制，在获取到类加载至JVM中的静态class对象，就能够进一步的获取到类的各种属性和方法，并且执行这些方法
2. 反射是各种Java框架的基石，最常见的一种用途就是在启动时加载配置文件，执行各种预处理方法。比如在配置文件中设置不同的底层数据库
   1. 正常操作：通过if-else这种代码块，为每种数据库都写一段创建对象执行初始化操作的代码
   2. 使用反射：通过配置文件中的数据库类名获取到相应的class对象，执行相应的初始化方法
3. 在使用反射的情况下，只需要一段代码就能够处理各种不同配置的数据库的初始化加载操作，当然这需要所以数据库类提供统一的初始化方法。

## 具体方法

1. 获取class对象
   1. 通过静态class变量获取：Class cls = String.class;
   2. 通过类的实例化对象获取：String s = "Hello"; Class cls = s.getClass();
   3. 通过类名获取：：Class cls = Class.forName("java.lang.String");
   4. 获取父类class对象：Class father\_cls = cls. getSuperclass();
2. 获取class对象的所有字段
   1. Public字段
      1. Field [] field\_list = cls.getFields();
      2. Field field = cls.getField(field\_name);
   2. Private字段(不包括父类)：getDeclaredFields(),getDeclaredField(field\_name)
   3. Field对象拥有以下几个方法
      1. getName()：返回字段名称
      2. getType()：返回字段类型，也是一个Class实例，例如，String.class；
      3. getModifiers()：返回字段的修饰符
3. 获取class对象的方法
   1. Public方法
      1. Method[] getMethods();
      2. Method getMethod(method\_name, arg1.class,…)
         1. 第一个参数是方法名
         2. 后面的参数是方法的形参类型对应的class对象
         3. 因为重载的存在，相同的方法名有不同的实现
         4. 例如获取String类的substring(int n)方法
            1. Method m = String.class.getMethod("substring", int.class);
   2. Private方法(不包括父类)：getDeclaredMethods()， getDeclaredMethod(name, arg1.class,…)
   3. Method对象拥有以下几个方法
      1. getName()：返回方法名称
      2. getReturnType()：返回方法返回值类型，也是一个Class实例
      3. getParameterTypes()：返回方法的参数类型，是一个Class数组，
      4. getModifiers()：返回方法的修饰符
4. 构造方法
   1. Public构造方法：
      1. Constructor[] getConstructors()
      2. Constructor getConstructor(arg.class...)
   2. Private方法(不包括父类)
      1. Constructor[] getDeclaredConstructors()
      2. Constructor getDeclaredConstructor(arg.class...)
5. 普通方法调用：invoke(Object obj, Object... args);
   1. 第一个参数是调用该方法的对象
   2. 后面的是方法的参数
6. 构造方法调用：newInstance(Object ... initargs);
   1. 参数是构造方法的参数

# 动态代理

## 代理模式

1. 代理类与委托类**有同样的接口**，**代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等**。
2. 代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，**代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务**。
3. 简单的说就是，我们在访问实际对象时，是通过代理对象来访问的，代理模式就是在访问实际对象时引入一定程度的间接性，因为这种间接性，可以附加多种用途

## 静态代理

静态代理：由程序员创建或特定工具自动生成源代码，也就是在编译时就已经将接口，委托类，代理类等确定下来。在程序运行之前，代理类的.class文件就已经生成。

1. 接口就是一种委托类，接口的实现类是相应的代理类，接口实现类方法为接口方法进行数据预处理和事后处理
2. 因为接口方法什么都不做，所以接口实现类方法中就必要显示的调用接口的方法

## 动态代理

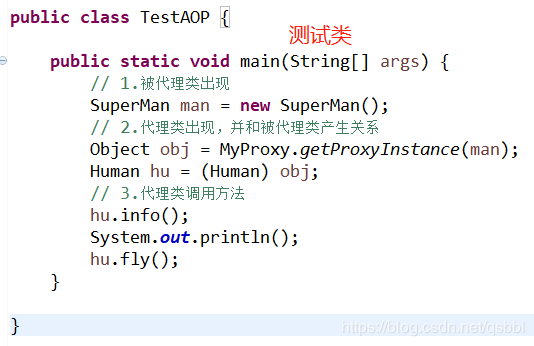
动态代理：

1. 通过反射原理，在运行时动态的创建代理类。
2. 创建的动态代理类对象是一个接口对象，必须转型成相应的接口，所以委托类必须继承于某个接口
3. 通过invoke方法执行委托类方法
4. 可以灵活的管理代理类方法，一个代理类可以支持任意多个委托类
   1. 在使用时，需要对哪个类进行代理，就可以临时创建一个代理类，将委托类对象实例传进去即可

优点：

1. 假如需要对很多委托类或者委托类的方法加上相同的预处理和后处理操作，使用静态代理就需要修改每个代理类的代码，而动态代理所以只需要修改一处代码。





补充说明：

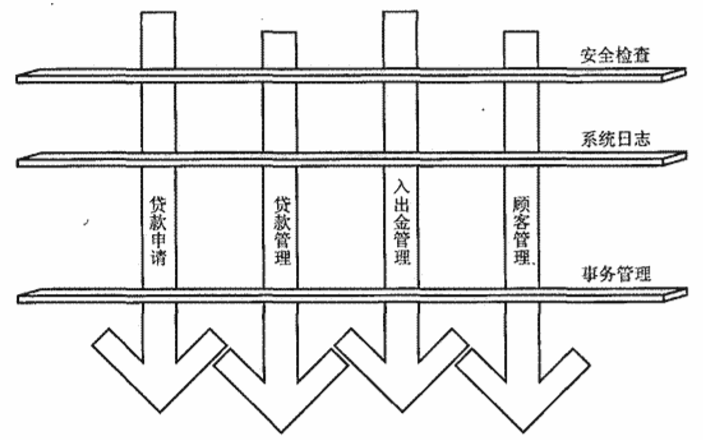
1. 为什么要将Object的代理类对象实例转型成Human接口类型？
   1. 因为代理类需要执行info()和fly()这些方法，而Object类里是没有定义这些方法的，只有相应的接口类Human里才定义了这些方法
   2. 当然，代理类初始化时传进去的是SuperMan对象实例，调用具体方法时也是调用的SuperMan类定义的方法实现
2. 根据上述要求可知，能够实现代理类的委托类，必须实现了某个接口，并且被代理的方法一定是这个接口中定义的方法，否则代理类转型成接口类型的对象实例后无法调用这些不在接口内定义的方法

动态代理类的对象为什么只能转型成一个接口：

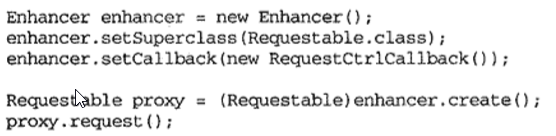
1. 动态代理机制是在运行时动态编译，动态生成代理类的class文件进行加载
2. 动态代理类的class文件中代理类默认继承于Proxy类，实现委托类的接口
   1. public final class $Proxy0 extends Proxy implements xxxInterface
3. 由于JAVA的单继承机制，代理类不可能再继承于其他类
4. 所以返回的Proxy类型的代理类对象只能强制转换成一个实现的委托类接口对象

# AOP

## 理论

1. 类似于代理类，当我们需要为各种不同的需求增加相似的新的需求，使用OOP编程方法，就需要修改每个原需求的所有类；而使用AOP编程方法，只需要修改代理类即可。
   1. 
2. 这种新的需求像横切面一样插入到众多原有需求中，所以叫做面向切面编程AOP，实现AOP这种思想的编程语言叫做AOL
3. JAVA平台的AOP实现机制
   1. 动态编程：基于反射，只对接口有效，动态编译，性能差一些
   2. 动态字节码增强：使用工具再运行期间，动态生成原有类的子类，子类中实现了新的需求，真正执行的是这些子类
   3. 自定义类加载器：在加载解析class文件时将横切逻辑织入
   4. 使用AOL语言编7A0B

## Spring AOP

1. 采用动态代理和动态字节码生成的方式，为目标对象创建一个代理对象，最终执行的是代理对象
   1. 当委托类有继承接口时，就使用动态代理，通过InvocationHandler实现横切逻辑，通过Proxy.newProxyInstance(委托类类加载器，委托类接口，横切逻辑处理对象)创建代理类
   2. 否则，使用CGLIB(Code Generation Library)的开源动态字节码生成类库生成代理对象实例
2. 动态字节码方法与动态代理相似，先通过proxy.Callback接口的实现类定义横切逻辑，然后使用CGLIB的Enhancer类的对象实例设置委托类(父类)和横切逻辑类(子类)，然后生成委托类的对象，不过该对象是一个子类对象，插入了横切逻辑
   1. 
3. 更复杂的AOP编程可以使用AspectJ实现

# 类加载机制

1. 类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 java.lang.Class对象，用来封装类在方法区内的数据结构。
2. 在预料到类将被使用时进行加载，在类首次被使用时抛出加载异常(如果有的话)
3. 加载的class文件来源：本地磁盘、压缩文件(jar/zar等)、网络、数据库、其他文件生成(JSP)