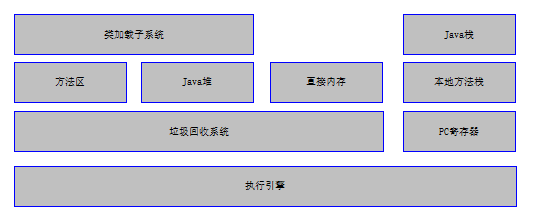
## JVM基本结构



JVM结构

***类加载子系统***：负责从文件系统或者网络中加载Class信息，加载的类信息存放在被称为方法区的内存空间；

***方法区***：存放类信息，运行时常量(字符串常量，数字常量)

***Java堆***：虚拟机启动时建立，Java程序主要的内存工作区域，几乎所有的Java对象实例都存放在Java堆中，是所有线程共享

***直接内存***：是在Java堆外，直接向系统申请的内存，访问速度优于Java堆，读写频繁的场合会考虑使用直接内存，NIO库运行Java程序使用直接内存，大小不受限于Xmx指定的最大堆大小，受限于系统内存。

***垃圾回收系统***：对方法区，Java堆和直接内存进行回收

***Java栈***：线程创建的时候被创建，是线程私有的，保存着帧信息，局部变量，方法参数，同时和Java方法的调用，返回密切相关

***本地方法栈***：与Java栈类似用于本地方法(通常是C编写)的调用

***PC寄存器***：每个线程的空间，如果当前线程执行的方法不是本地方法，则指向被执行的指令，否则值是undefined

***执行引擎***：负责执行虚拟机的字节码

### 对象去哪里：Java堆

s1

s0

eden

Tenured

新生代

老年代

上图显示了一个堆 空间的一般结构。

s0和s1也别称为from和头区域，是两块大小相等，可以互换角色的内存空间。

public class SimpleHeap {

private int id;

public SimpleHeap(int id) {

this.id = id;

}

public void show() {

System.out.println("my id is " + id);

}

public static void main(String[] args) {

SimpleHeap s1 = new SimpleHeap(1);

SimpleHeap s2 = new SimpleHeap(2);

s1.show();

s2.show();

}

}

**Java栈**

**方法区**

**堆**

s1实例

SimpleHeap

类及其方法实现

局部变量s1

局部变量s2

s2实例

上图显示了堆，方法区，栈的关系

Java堆与程序数据密切相关

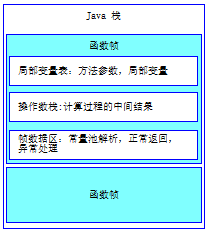
### 函数如何调用：出入Java栈

Java栈是线程私有的内存空间，线程执行的基本行为是函数调用，每次函数调用的数据都是通过Java栈传递的。

Java栈保存的内外为栈帧，每次函数调用，都会有对应的栈帧被压入Java栈，每一个函数调用结束，都会有一个栈帧被弹出Java栈。

当前执行函数的帧就是当前的帧(位于栈顶)。

每个栈帧包含：局部变量表,操作数栈(中间运算结果),帧数据(常量池解析+正常方法返回+异常处理)。



### 类去哪里了：识别方法区

与Java堆一样，方法区是一块所有线程共享的区域，保存了系统的类信息(类的字段、方法、常量池)。方法区的大小决定了系统可以保存多少个类。

## 常用的Java虚拟机参数

### 垃圾回收相关

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintHeapAtGC

-XX:+PrintGCTimeStamps

-XX:+PrintGCApplicationConcurrentTime

-XX:+PrintGCApplicationStoppedTime

-XX:+PrintReferenceGC

### 类加载/卸载相关

-XX:+TraceClassLoading

-XX:+TraceClassUnloading

-XX:+PrintClassHistogram

### 系统参数相关

-XX:+PrintVMOptions

-XX:+PrintCommandLineFlags

### 堆相关

-Xms

-Xmx

-XX:SurvivorRatio

-XX:NewRatio

-XX:+HeapDumpOnOutOfMemoryError

-XX:HeapDumpPath

### 非堆内存参数配置

-XX:PermSize

-XX:MaxPermSize

-XX:MaxMetaspaceSize

-Xss

-XX:MaxDirectMemorySize

## 垃圾回收相关算法

## Class装载系统

连接

初始化

加载

解析

准备

验证

**类加载的条件**：**将新类型添加到运行中的JVM进程里的过程**

一个类或接口在初次使用前，必须进行初始化，这里的使用指“主动使用”，只有下列几种情况：

\* 1.创建类实例,如new、反射、克隆、反序列化

\* 2.调用类的静态方法

\* 3.使用类和接口的静态字段(final常量除外)

\* 4.反射类的方法时

\* 5.初始化子类时，要求先初始化父类

\* 6.作为入口类，含有main方法的类

**加载类：**

\* 1.通过类的全名，获取类的二进制数据流

\* 2.解析类的二进制数据流为方法区内的数据结构

\* 3.创建java.lang.Class类的实例，表示该类型

**PS：ClassLoader获取类的二进制信息**

**验证类：**

\* 1.格式检查：检查魔数、版本、长度

\* 2.语义检查：是否继承final、是否有父类、抽象方法是否有实现

\* 3.字节码验证：跳转指令是否指向正确位置、操作数类型是否合理

\* 4.符号引用验证：符号引用的直接引用是否存在

**准备：验证通过后，为类分配相应的内存空间并设置默认值**

\*

**解析类：将类、接口、字段和方法的符号引用转化为直接引用**

\*

\*

**初始化：初始化静态变量、运行静态初始化代码块**

## 并行计算

### 常见概念

\* 同步(Synchronous)和异步(Asynchronous)：

\* 并发(Concurrency)与并行(Paranllelism)：

\* 临界区：表示一种公共资源或者说是共享数据，可以被多个线程使用

\* 阻塞(Blocking)和非阻塞(Non-Blocking)：

\* 死锁(Deadlock)、饥饿(Starvation)和活锁(Livelock)：

饥饿：线程因为种种原因无法获得所需资源，导致一直无法执行

活锁：由于谦让，资源不断在两个线程中跳动，而没有一个线程可以同时拿到所有资源而正常执行。

\* 并发级别：阻塞、无饥饿、无障碍、无锁、无等待

### JMM：Java Momery Model Java 内存模型

围绕着并发过程中如何处理可见性、原子性、有序性而建立的模型。

\* 可见性：当一个线程修改了某一个共享变量的值，其他线程能够立即知道这个修改，JMM提供了volatile变量定义、final、synchronized块来保证可见性。

\* 原子性：指一个操作是不可中断的，即时是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。JMM提供保证了访问基本数据类型的原子性，但是实际业务处理场景往往是需要更大的范围的原子性保证，所以模型也提供了synchronized块来保证。

有序性：这个概念是相对而言的，如果在本线程内，所有的操作都是有序的，如果在一个线程观察另一个线程，所有的操作都是无序的，前句是“线程内表现为串行行为”，后句是“指令的重排序”和“工作内存和主内存同步延迟”现象，模型提供了volatile和synchronized来保证线程之间操作的有序性。

重排序：在执行程序时为了提高性能，编译器和处理器常常会对指令做重排序(编译器、处理器)，就是因为这些重排序，所以可能会导致多线程程序出现内存可见性问题(数据安全问题)和有序性问题。

对于编译器，JMM的编译器重排序规则会禁止特定类型的编译器重排序

对于处理器重排序，JMM的处理器重排序规则会要求java编译器在生成指令序列时，插入特定类型的内存屏障（memory barriers，intel称之为memory fence）指令，通过内存屏障指令来禁止特定类型的处理器重排序。

### 线程状态图

结束

start

Terminated

Runnable

New

notify通知

同步块

notify通知

wait等待

wait等待

synchronized

Timed\_Waiting

Waiting

Blocked

\* New状态表示刚刚创建的线程，还没开始执行；

\* 调用线程的start()方法时，线程开始执行，处于Runnable状态

\* 线程执行过程中，遇到synchronized同步块，会进入Blocked阻塞状态

\* Waiting会进入一个无时间限制的等待

\* Timed\_Waiting会进行一个有限时的等待

### 线程基本操作

新建线程 🡪 终止线程 🡪 中断线程

线程中断并不会使线程立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出，至于线程如何处理，则完全由目标线程决定。

线程中断，只是给目标线程增加一个标识位。

🡪 等待(wait)和通知(notify)

这两个方法是在Object类中，意味着任何对象都可以调用这两个方法。

调用Object.wait()方法，会进入Object对象的等待队列，这个等待队列，可能会有多个线程，因为系统运行多个线程同时等待某一个对象；wait()方法执行后，会释放object的锁

当object.notify()被调用时，会**随机**从这个等待队列中唤醒一个线程。

\* Object.wait()和Thread.sleep()方法都会让线程等待若干时间，wait()可以被唤醒并且释放目标对象的锁，而Thread.sleep()不会释放任何资源。

🡪 挂起(suspend)和继续执行(resume)

已被废弃，因为挂起会导致线程暂停的同时，不会释放任何资源，其他任何线程想要访问被它暂用的锁时，都将会进入阻塞。

🡪 等待线程结束(join)和谦让(yield)

join的本质是让调用线程wait在当前线程对象实例上。

yield会让当前线程让出CPU

🡪 守护进程(Daemon)

守护线程的设置需要在线程启动前设置。

守护线程是一种特殊的线程，当只有守护线程时，Java虚拟机就会自然退出。

🡪 线程优先级

优先级高的线程在竞争资源时会更有优势，更可能抢占资源。

### 线程安全和synchronized

线程安全：线程执行后数据状态是一致的。

synchronized：用于实现线程同步，有如下的用法：

\* 加锁对象：进入同步代码前要获得给定对象的锁

\* 作用于实例方法：进入同步代码前要获得当前实例的锁

\* 作用于静态方法：进入同步代码前要获得当前对象的锁

### 多线程下的错误

\* ArrayList扩容过程，内部一致性被破坏，但由于没有锁的保护，另外一个线程访问到了不一致的内部状态，导致出现越界问题。

\* HashMap put()方法，多线程下链表成环，导致死循环

### 错误的加锁

\* 加锁对象不能是可变的

### 学习资源

HashMap多线程卡死的细节分析

<http://coolshell.cn/articles/9606.html>

## JDK并发包

### 线程同步

\* synchronized：获得锁之前会一直阻塞

\* Object.wait()和Object.notify()

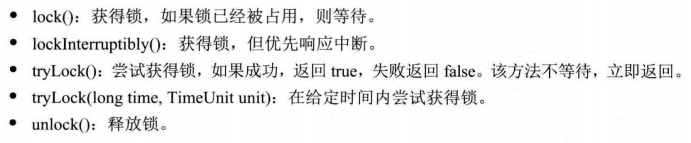
\* 重入锁(ReentrantLock)：

获得锁的线程可以反复进入该锁，而无需重新进行锁的申请(一个线程可以连续两次获得同一把锁)；

等待锁的过程，可以获得通知，而停止工作。

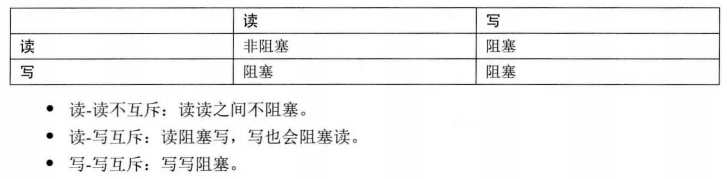
限时加锁等待；

公平锁



\* 信号量：允许多个线程同时访问

\* 读写锁：ReadWriteLock



### 线程池(需要再看3遍)

线程数量过大时，会耗尽CPU和内存资源：首先，线程的创建和关闭需要花费时间；其次，线程本身也是要占用内存空间的，大量线程会抢占宝贵的内存资源，即时没有，也会加大GC的压力，延长GC的停顿时间。

线程池与数据库连接池。

\* <p>

\* Executors : 线程池工厂

\* <br/>

\* newFixedThreadPool():返回一个固定线程数量的线程池，当新的任务提交时，线程池如果有空闲线程，则立即执行，

\* 否则，新任务会暂存在一个任务队列中。

\* <br/>

\* newScheduledThreadPool():

\* <br/>

\* newSingleThreadExecutor():创建一个只有单个线程的线程池

\* <br/>

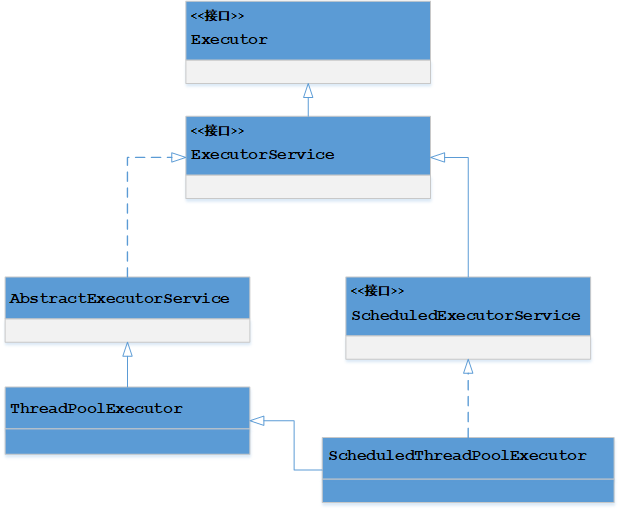
\* newCachedThreadPool():返回一个可根据实际情况调整线程数量的线程池，优先使用空闲线程，

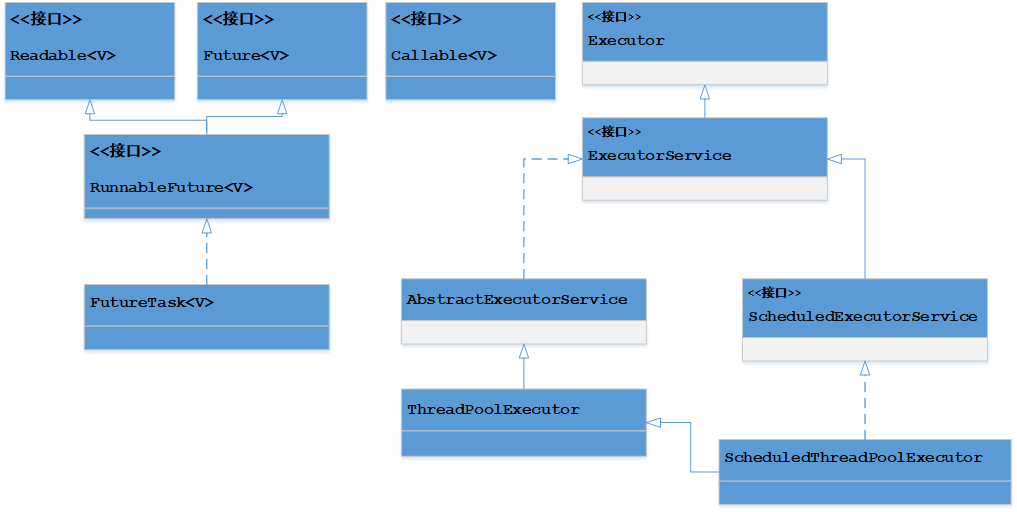
\* 当所有线程都在工作时，又有新的任务提交，会创建新的线程处理任务。可指定最大线程数

\* <br/>

\* </p>

ThreadPoolExecutor





### JDK的并发容器

java.util.concurrent.

ConcurrentHashMap:高效的并发HashMap，可理解为一个线程安全的HashMap

CopyOnWriteArrayList:适用于读多写少的场合

ConcurrentLinkedQueue:高效的并发队列，使用链表实现，可理解为一个线程安全的LinkedList

BlockingQueue:表示阻塞队列，适用于数据共享的通道

ConcurrentSkipListMap:跳表的实现。