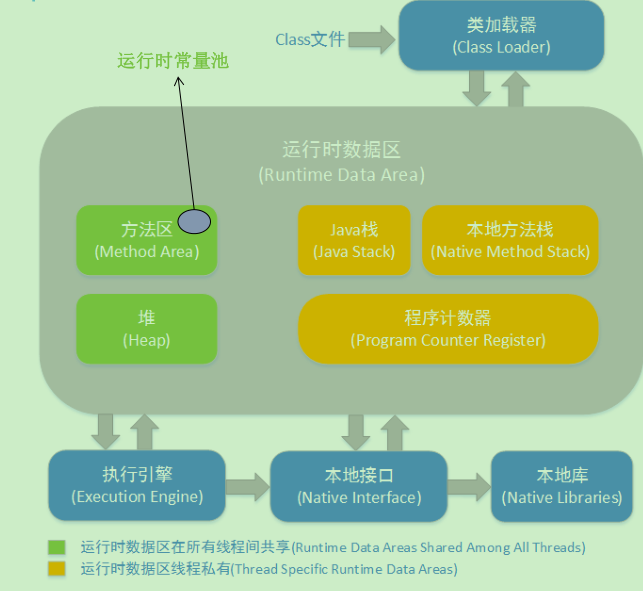
# Jvm

## Jvm内存模型



程序计数器：较小的内存空间，

当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响；

java 栈：线程私有，生命周期和线程，每个方法在执行的同时都会创建一个 栈帧用于

存储局部变量表，操作数栈，动态链接，方法出口等信息。方法的执行就对应着栈帧在虚拟机栈中入栈和出栈的过程；栈里面存放着各种基本数据类型和对象的引用；(简单来说就是方法放在里面)

本地方法栈：本地方法栈保存的是native方法的信息，当一个JVM创建的线程调用native方法后，JVM不再为其在虚拟机栈中创建栈帧，

JVM只是简单地动态链接并直接调用native方法；

(简单来说就是放本地(c++/第三方)方法的,比如sdk包)

堆：Java堆是程序员需要重点关注的一块区域，因为涉及到内存的分配(new关键字，反射等)与回收(回收算法，收集器等)；(对象,字符串常量池)

方法区：也叫永久区，用于存储已经被虚拟机加载的类信息，常量("zdy","123"等)，静态变量(static变量)等数据。

(jdk1.8已经将方法区去掉了，将方法区移动到直接内存)(元空间,类放在里面)

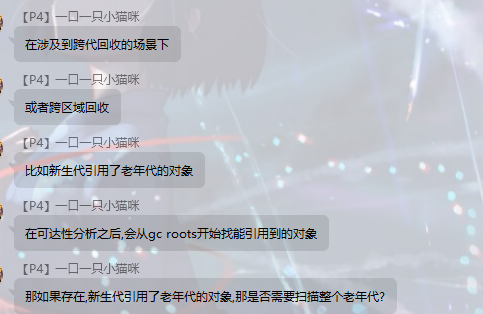
运行时常量池：

运行时常量池是方法区的一部分，用于存放编译期生成的各种字面("zdy","123"等)和符号引用。(元空间一部分,基本类型常量池)

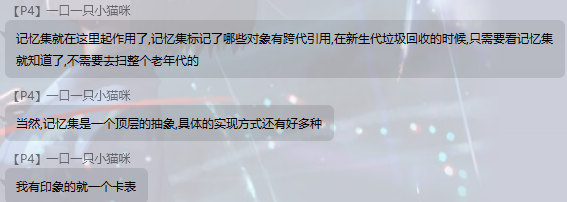
直接内存：不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域；

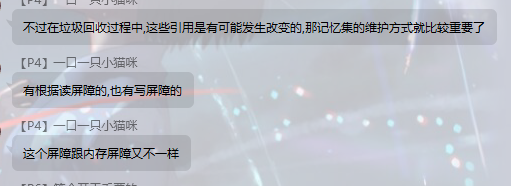
## Gc回收算法!

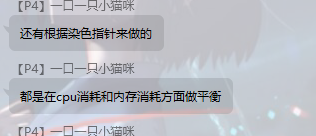
## 记忆集在gc回收中的作用!

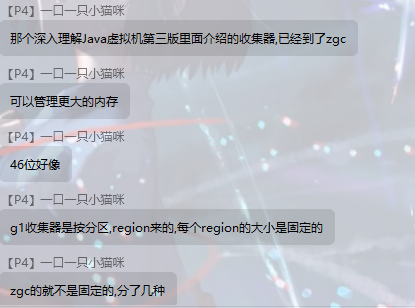












# 线程池

## ThreadPoolExecutor

所有线程池都调用了这个类

ThreadPoolExecutor(int corePoolSize 核心线程数,  
 int maximumPoolSize 最大线程数,  
 long keepAliveTime 最大存活时间,  
 TimeUnit unit 时间单位,  
 BlockingQueue<Runnable> workQueue 队列, 队列,  
 ThreadFactory threadFactory 线程工厂,  
 RejectedExecutionHandler handler 拒绝策略)

## ScheduledThreadPoolExecutor(定长线程池)

创建一个定长线程池，支持定时及周期性任务执行

ScheduledThreadPoolExecutor(int corePoolSize) {  
 super( corePoolSize, //核心线程数,就是你传的参数

Integer.*MAX\_VALUE* , // *最大线程数*(2^31)-1

0 , //存活时间

*NANOSECONDS* ,//时间单位*纳秒*  
 new DelayedWorkQueue()); //优先级队列

## newCachedThreadPool (可缓存线程池)

可缓存线程池，先查看池中有没有以前建立的线程，如果有，就直接使用。如果没有，就建一个新的线程加入池中，缓存型池子通常用于执行一些生存期很短的异步型任务

newCachedThreadPool() {  
 return new ThreadPoolExecutor(

0, //核心线程数

Integer.*MAX\_VALUE*,//最大线程数 2^31 -1

60L,//存活时间

TimeUnit.*SECONDS*,//时间单位  
 new SynchronousQueue<Runnable>()); //同步队列

## newFixedThreadPool(可重用固定个数线程池)

创建一个可重用固定个数的线程池，以共享的无界队列方式来运行这些线程

newFixedThreadPool(int nThreads) {  
 return new ThreadPoolExecutor(

nThreads, //核心线程数

nThreads,//最大线程数  
 0L, //存活时间

TimeUnit.*MILLISECONDS*,//时间单位  
 new LinkedBlockingQueue<Runnable>());//链表缓冲队列

## newSingleThreadExecutor(单例线程池/单一线程池)

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

newSingleThreadExecutor() {  
 return new FinalizableDelegatedExecutorService  
 (new ThreadPoolExecutor(

1, //核心线程数

1,//最大线程数  
 0L,//存活时间

TimeUnit.*MILLISECONDS*,//时间单位 new LinkedBlockingQueue<Runnable>()));//链表缓冲队列

### LinkedBlockingQueue

LinkedBlockingQueue大小不固定的BlockingQueue，若其构造时指定大小，生成的BlockingQueue有大小限制，不指定大小，其大小有Integer.MAX\_VALUE来决定。其所含的对象是FIFO顺序排序的

### BlockingQueue!

### ArrayBlockingQueue

ArrayBlockingQueue（int i）:规定大小的BlockingQueue，其构造必须指定大小。其所含的对象是FIFO顺序排序的

### DelayedWorkQueue!

DelayedWorkQueue 阻塞队列

成员变量

// 初始时，数组长度大小。

private static final int INITIAL\_CAPACITY = 16;

// 使用数组来储存队列中的元素。

private RunnableScheduledFuture<?>[] queue = new RunnableScheduledFuture<?>[INITIAL\_CAPACITY];

// 使用lock来保证多线程并发安全问题。

private final ReentrantLock lock = new ReentrantLock();

// 队列中储存元素的大小

private int size = 0;

//特指队列头任务所在线程

private Thread leader = null;

// 当队列头的任务延时时间到了，或者有新的任务变成队列头时，用来唤醒等待线程

private final Condition available = lock.newCondition();

# Mysql

## Innodb事务隔离级别

读已提交,读未提交,可重复度,串行化

## 特点!

## 默认事务隔离级别,为什么?!

可重复读,因为在5.0之前binlog只支持statement(记录修改的sql语句)这种模式,这种模式在

读已提交这个隔离级别下主从复制是有bug的

## 可重复读这个事务隔离级别是怎么实现的?!

乐观锁?

Redo log,undo log?

## Mysql事务提交成功后还会丢失吗?

会丢失,因为执行顺序是先缓冲到日志缓冲区logBuffer,然后mysql调用write写入操作系统缓冲区os cahe,然后再flush到硬盘。而mysql在写入系统缓冲区后就会认定已经提交成功,具体什么时候flush到硬盘,就和mysql没有关系了,如果这个时候操作系统崩溃,就会丢失数据。

### 解决办法

innodb\_flush\_log\_at\_trx\_commit参数能够控制事务提交时，刷redo log的策略

**策略一：最佳性能**(innodb\_flush\_log\_at\_trx\_commit=0)每隔一秒，才将Log Buffer中的数据批量write入OS cache，同时MySQL主动fsync。这种策略，如果数据库奔溃，有一秒的数据丢失。

**策略二：强一致**(innodb\_flush\_log\_at\_trx\_commit=1)每次事务提交，都将Log Buffer中的数据write入OS cache，同时MySQL主动fsync。这种策略，是InnoDB的默认配置，为的是保证事务ACID特性。

**策略三：折衷**(innodb\_flush\_log\_at\_trx\_commit=2)每次事务提交，都将Log Buffer中的数据write入OS cache；每隔一秒，MySQL主动将OS cache中的数据批量fsync。画外音：磁盘IO次数不确定，因为操作系统的fsync频率并不是MySQL能控制的。这种策略，如果操作系统奔溃，最多有一秒的数据丢失。

高并发业务，行业内的最佳实践，是：  
innodb\_flush\_log\_at\_trx\_commit=2