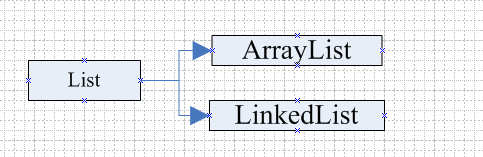
# 1.java集合

## 1.1 java有那些集合

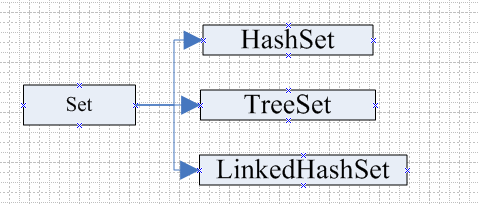
* List



ArrayList：底层基于数组实现，所以随机遍历的速度较快；

LinkedList：底层基于链表的实现，所以插入和删除比较快；

* Set

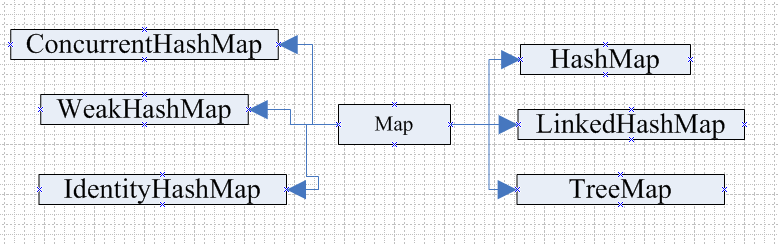


HashSet：使用散列

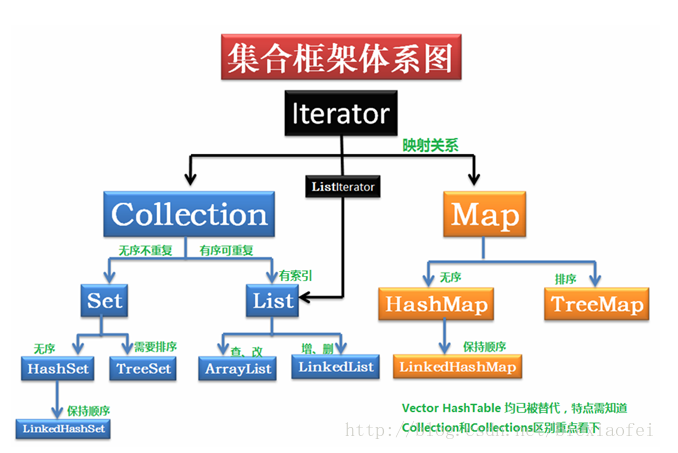
TreeSet 使用红黑树，会对元素排序，接口是SortedSet。初始化TreeSet时可以传入Comparator对象的实现做为排序函数

LinkedHashSet 也使用散列，但使用链表来维护插入顺序

* Map

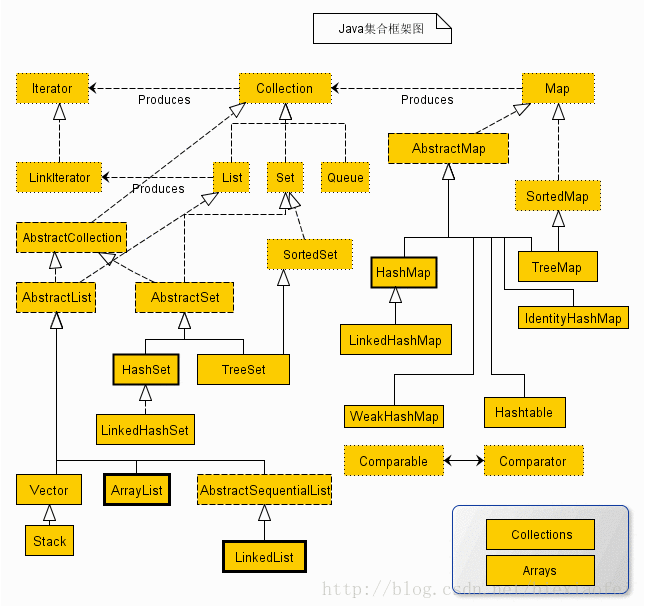


## 1.2 集合的集成图



从上面可以看到，Map是不实现Collection，所以遍历元素不能直接使用foreach

集合框架图：



## 1.3 Vector,ArrayList, LinkedList的区别

回答这种问题，首先从结构开始答起，接着性能，接着使用或者其它需要注意点。

* Vector是线程安全的，而ArrayList、LinkedList不是线程安全的，Vector线程安全的原因是每个方法都加上了synchronized方法（线程安全）所以性能上比ArrayList要差
* Vector、ArrayList都是以类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储，而数组形式适合查找，链表形式适合插入和删除
* ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间，链表不需要定长，所以不需要提前扩充空间

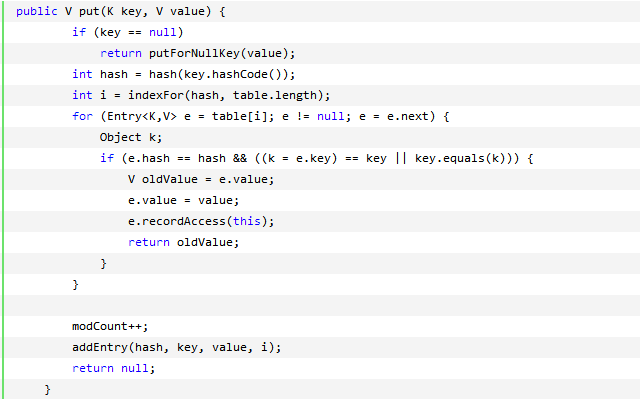
相同点：都实现了Collection接口

## 1.4 HashMap如何存储数据？

参考文章：https://www.cnblogs.com/whgk/p/6071617.html

这里面涉及到一个散列值hashcode的内容，下面根据网上的文章来总结下我的看法，首先我总结一下HashMap的存储：

要进行存储一个元素，put(key, value)，这里key和value都是一个Object，下面贴出put方法的代码，结合代码进行讲解



* 第一步：会先使用hash函数（hash算法点击上面参考文章的链接）获取key对象的散列值；
* 第二步：接着在hash表中，根据散列值找到key对象应该存储的位置，但是这个位置可能存在多个key对象，有可能存在相同的key对象，也有可能该key对象不存在，那么此时会使用key对象的equal方法（Object对象自带的）去逐个比较，如果不存在相同的，那么直接把该key对象和value的存储地址存放在这个位置，存在相同的则刷新value对象的存储地址；

通过以上这几个步骤，可以避免Map逐个key对象进行比较。接着我们还要注意几个地方：

* 首先，不同的key对象使用hash函数取得的hashcode，有可能会一样，但这是否会出现一个问题：hash表中存在很多的key对象，答案是不会，因为真实的hash表非常的大，位置非常的多，所以一个hashcodo的位置不会放太多的key对象；
* 另外，网上的文章提到很多修改equal方法后，为啥要修改hashcode方法，要首先知道，equal和hashcode方法都是Object对象自带的方法，key对象是可以重写覆盖的，如果重写了equal方法，那么必然要修改hashcode方法，否则。因为String的equal方法是比较字符串的值是否相等，而==则是比较内存地址，但Object的equal方法则相当于String的==，是对内存地址进行比较。如何修改了equal方法，修改了比较相等的规则，那么当使用key对象的equal方法进行逐个比较时，原本存在的key对象可能会认为不存在。（String类是对equal和hashcode方法进行了重写）；

参考文章：https://www.cnblogs.com/dolphin0520/p/3681042.html

## 1.5 Hashmap和Hashtable的比较

参考文章：

<http://www.importnew.com/24822.html>

**相同点：**

* HashMap和HashTable都使用hash表来存储键值对

**差异点：**

* HashMap的put方法，key可以为null，而hashtable的key为null时抛异常；
* HashTable是线程安全（使用了synchronized），而HashMap不是线程安全；
* 初始容量大小和每次扩充容量大小的不同；

## 1.6 HashMap的查询时间复杂度

说到这里，我们先来恶补一下时间复杂度的概念，参考：

<https://www.cnblogs.com/huangbw/p/7398418.html>

时间复杂度按n越大算法越复杂来排的话：常数阶O(1)、对数阶O(logn)、线性阶O(n)、线性对数阶O(nlogn)、平方阶O(n²)、立方阶O(n³)、……k次方阶O(n的k次方)、指数阶O(2的n次方)。

O(1)：则当元素越多时，运算的量级始终不变；

O(n): 当元素越多，运算呈线性增长；

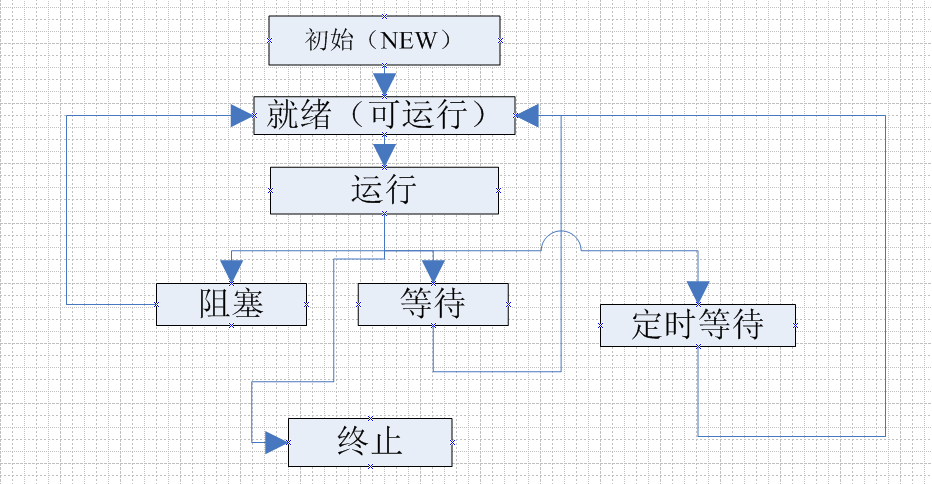
所以，由于HashMap是利用hashcode和hashcode表来进行查询，跟元素的量级无关，这里的时间复杂度为O(1)；

# 2.多线程问题

## 2.1 线程的几个状态

参考链接：<https://www.cnblogs.com/GooPolaris/p/8079490.html>

* 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法。
* 运行(RUNNABLE)：Java线程中将就绪（ready）和运行中（running）两种状态笼统的成为“运行”。线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得cpu 时间片后变为运行中状态（running）。
* 阻塞(BLOCKED)：表线程阻塞于锁。
* 等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
* 定时等待(TIME\_WAITING)：该状态不同于WAITING，它可以在指定的时间内自行返回。
* 终止(TERMINATED)：表示该线程已经执行完毕。



运行通常分为**可运行（就绪）**和**运行状态**。

可以运行但是没有获得cpu时间片的状态为**可运行**。获得cpu时间片的状态才能运行。

在运行期间调用wait方法时，运行状态会变为**等待状态**。

如果wait方法带有时间参数，就成为**定时等待状态**；等待状态和定时等待状态会释放锁，唤醒后要重新争夺锁资源。

而当线程在运行期间，如果调用了sleep方法或者进入了I/O状态，争夺不到I/O资源，又或者当前线程中有别的线程进行join操作，那么就会成为阻塞状态。阻塞状态也不释放锁。

等待状态和阻塞状态的区别在于，阻塞状态有可能是被动的（当然也可能主动），等待状态为主动的。获取不到锁的状态为阻塞（原因可能是别的线程调用了join方法），但这个是被动的情况，等待状态和阻塞状态唤醒后都会成为可运行状态，等待cpu时间片。任务执行完成后为终止状态。

## 2.2 ReenrrantLock的用法

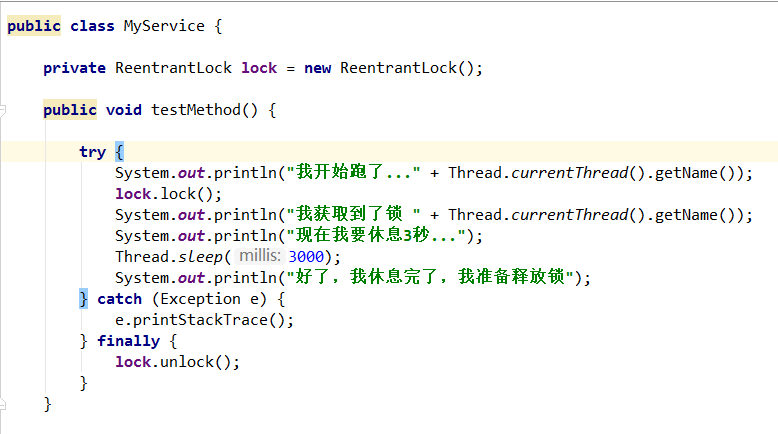
Java demo：

ThreadDigger\src\main\java\com\dig\lock\_demo\reenrrantlock\MyThread.java

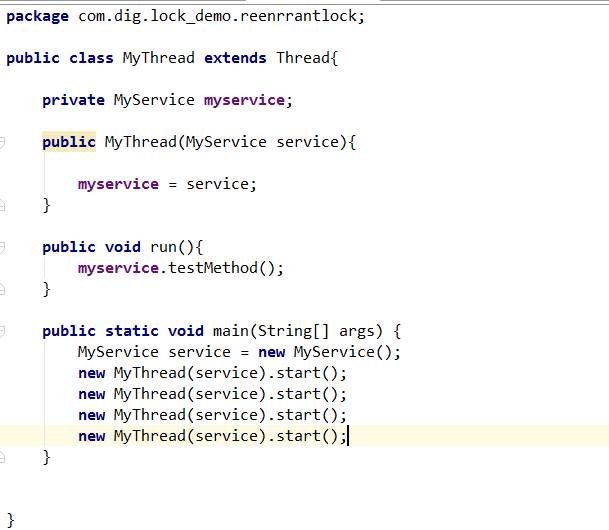
Demo参考链接：

<https://blog.csdn.net/cds86333774/article/details/51025245>

ReenrrantLock的用法很简单，主要作为一个同步锁资源存在，同一时间只允许允许单个线程占用了，其他线程如果要获取锁，则进行阻塞等待，直到获取到锁的线程把锁释放了。下面看个demo：

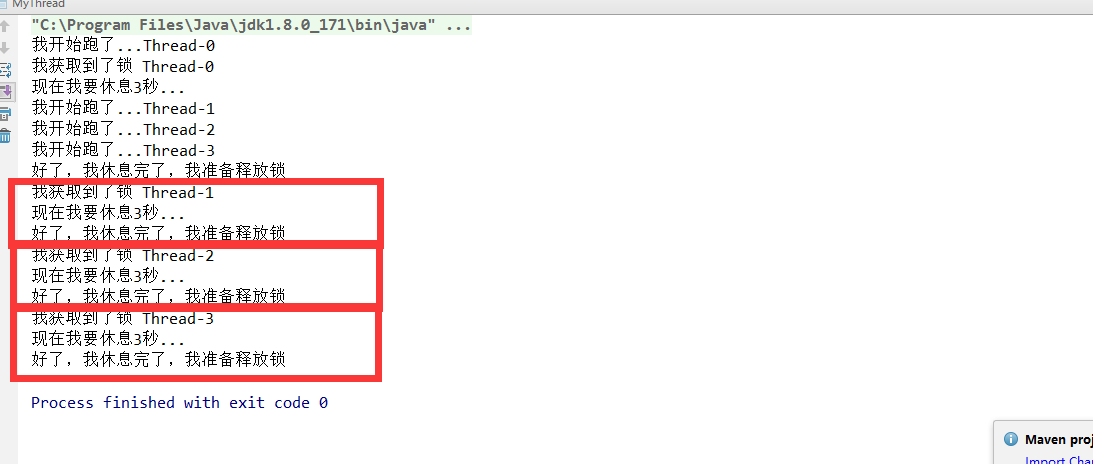


先建立一个服务，服务里定义一个ReenrrantLock锁资源，然后实现多线程，MyService作为多线程的构造器的参数。



所有的线程在启动时，会执行testMethod方法，testMethodd方法中需要对ReenrrantLock资源进行争夺。此时，只允许只有一个线程能获取到锁，其它获取不到锁资源的会继续阻塞等待。

运行main方法后，查看打印：



可以看到，线程如果没获取到锁之前，会一直呈现一个阻塞状态。只有获取到锁的线程才能继续往下走。而且从打印中可以看到，获取锁呈现了顺序的排序，即先到先得（即优先考虑等待时间最长的哪个线程）。

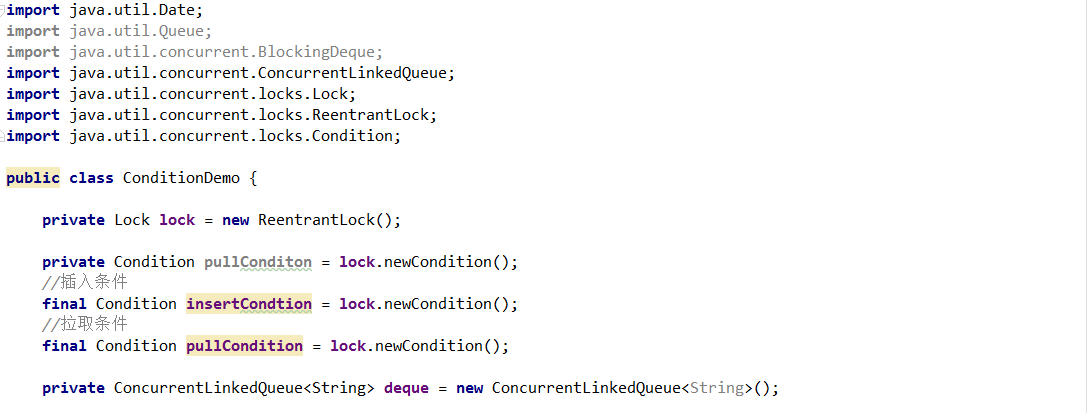
## 2.3 条件变量Condition的用法

Condition是配合Lock（java.util.concurrent.locks.Lock

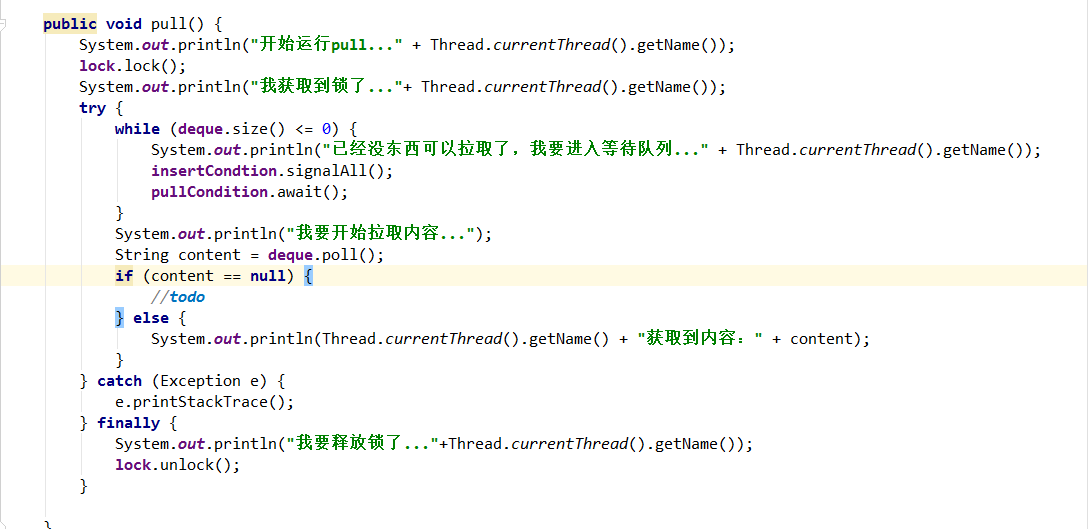
，这里的锁可以是ReenrrantLock，因为ReenrrantLock也是实现Lock接口的类）一起来使用的条件变量，主要的作用是可以让线程进行单个/同条件批量进入等待状态（等待状态会释放lock），或者，唤醒单个线程/同条件批量线程重新争夺锁资源，进入可运行状态。

下面查看demo：

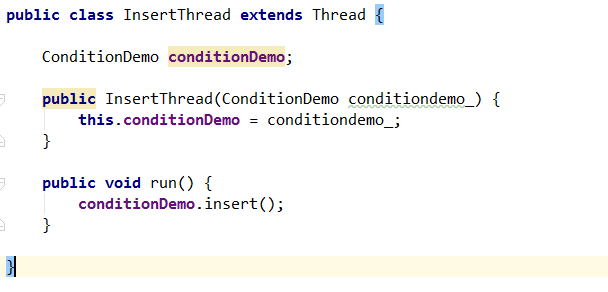
ThreadDigger\src\main\java\com\dig\lock\_demo\condition\ConditionDemo.java

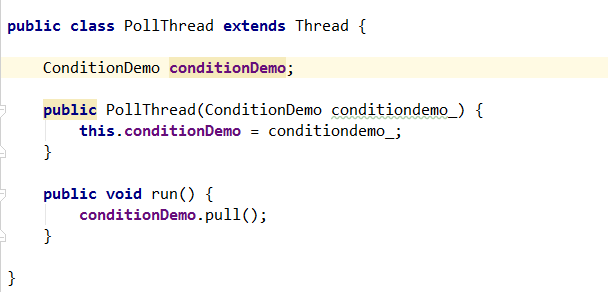




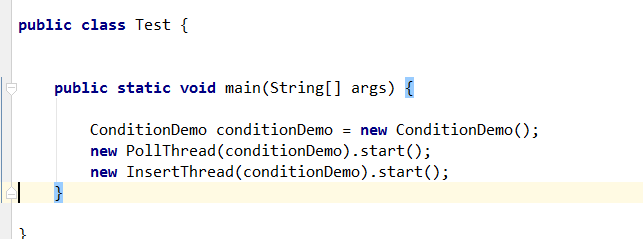


接着实现插入数据和拉取数据的线程：

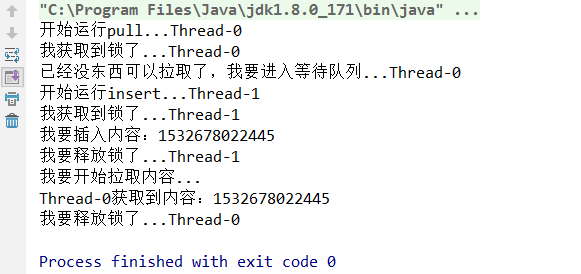




编写运行线程方法



查看打印：



Thread-0是拉取线程，Thread-1是插入线程。

开始运行，先运行拉取操作，拉取操作获取到锁后，开始运行，拉取操作发现没数据，会唤醒插入线程（实际上这时没必要唤醒插入线程，因为插入线程没有执行await方法，而是一直在等待lock资源），然后拉取线程进入等待状态，而等待状态会释放锁，所以插入线程会获取到被释放的锁，插入线程获取到锁后，进入运行状态，往队列中插入数据，会唤醒拉取线程，并且释放锁，这时拉取线程获取到了锁，注意，拉取线程被唤醒后，会从获取锁的哪一步代码开始执行，而不是睡眠的动作后面的代码开始。拉取线程获取到锁后，进入数据拉取，拉取完成后释放锁，至此，全部结束。

## 2.4 synchronized的等待操作

## 2.5 synchronized和ReenrrantLock的区别

## 2.6 volatile关键字解析

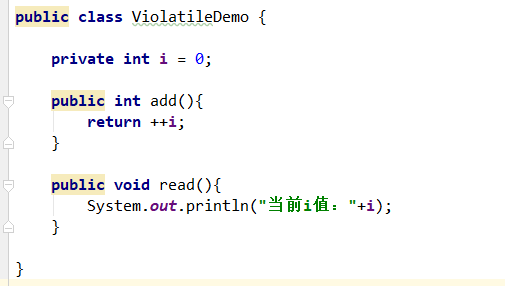
参考文章：

<https://www.cnblogs.com/chengxiao/p/6528109.html>

说的比较好的文章:

https://www.cnblogs.com/fswhq/p/java\_volatile.html

volatile作用是实现在两个线程间实现变量共享



例如上面的程序，如果一个线程执行add操作，另一个线程执行read操作，add操作改变了i值，但不能保证read值读取到的值就是最新的。因为没一个计算程序都有自己的本地内存，操作变量时，习惯性的会把主内存中的变量拉起拉取到本地内容，所以主内容发生的变化，本地内存不一定能及时清楚。

## 2.7 线程安全集合ConcurrentHashMap

首先ConcurrentHashMap跟HashMap、HashTable进行比较：

* HashMap不是原子性的（即多个线程同时访问），非线程安全；
* HashTable使用synchronized进行锁同步，所以效率受到影响。

接下来，我们要解析ConcurrentHashMap的原理和结构：

首先，ConcurrentHashMap的并发是使用了分段锁，不同数据段的hashcode需要竞争的锁不一样，不同数据段的hashcode不存在锁竞争，ConcurrentHashMap锁原理是使用了重入锁ReentrantLock；

以上的说法只是简单的说了一下线程安全的实现大体逻辑。ConcurrentHashMap是由segment数组和HashEntry数组来组成，segment是一把重入锁（ReentrantLock），代表一个数据段需要竞争的资源，而HashEntry则是用来存储数据，一个ConcurrentHashMap会有很多个HashEntry，并且平均分配给segment，只有获取到segment代表的锁，才有资格对该segment拥有的HashEntry进行变更。读操作不需要获取锁。

另外需要清楚的知识点：

* Key对象在定位segment时，时先获取hashcode，再使用hashcode进行hash函数得出hash值；
* 读操作不需要加锁，但使用了volatile，这个关键字在多线程篇解决；

## 2.8 ConcurrentLinkedQueue和LinkedBlockingQueue

参考文章：

<https://www.cnblogs.com/starcrm/p/4998067.html>

先来了解这两个集合是什么？

两个都是基于queue的改造。

ConcurrentLinkedQueue：非阻塞线程安全队列，基于链接节点的、无界的、线程安全。此队列按照 FIFO（先进先出）原则对元素进行排序。队列的头部 是队列中时间最长的元素。队列的尾部 是队列中时间最短的元素。新的元素插入到队列的尾部，队列检索操作从队列头部获得元素。当许多线程共享访问一个公共 collection 时，ConcurrentLinkedQueue 是一个恰当的选择。此队列不允许 null 元素。

LinkedBlockingQueue: 阻塞线程安全队列，按 FIFO（先进先出）排序元素。队列的头部 是在队列中时间最长的元素。队列的尾部 是在队列中时间最短的元素。新元素插入到队列的尾部，并且队列检索操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

看了许多文章，并没有那些文章说明两个队列的优劣，只是一个是阻塞，一个是非阻塞，根据情况进行使用，例如像socket通讯，写入的内容一定要按照顺序，那么这种情况适合使用阻塞。又或者，很多线程共享一个集合，并且没有顺序要求，那么此种情况适合非阻塞队列。

## 2.9 资源释放yield

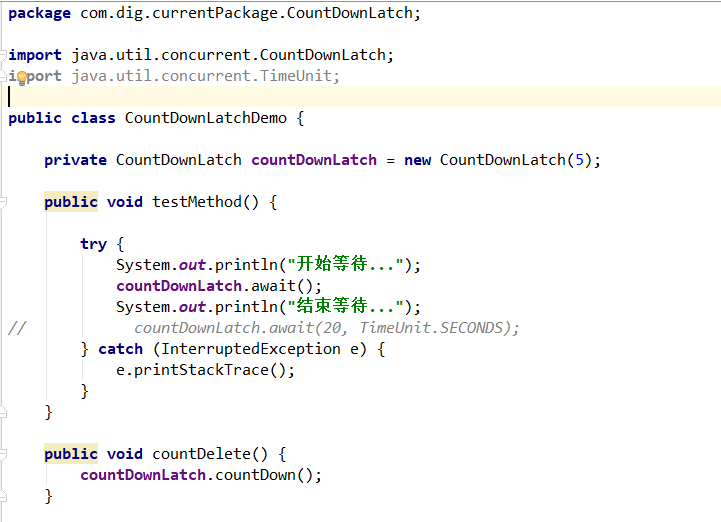
这个是资源暂时释放,把cpu时间片让出去，重新等待cpu时间片，原理是，把当前线程的状态调整为可运行的状态，即等待cpu时间片的一个状态。

## 2.9 线程包的重点关注类

### 2.9.1 计数CountDownLatch和屏障CyclicBarrier

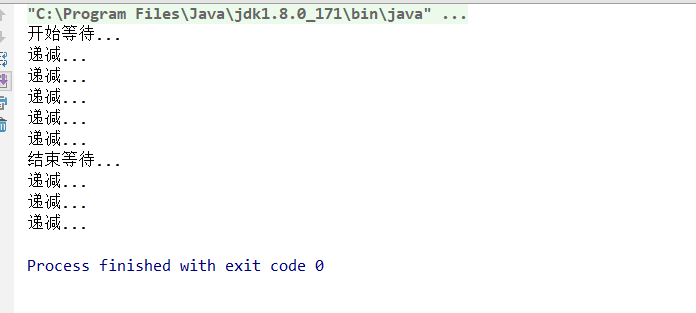
CountDownLatch是一个计算器，可以预定一个初始值，然后开始阻塞等待，然后通过在线程中进行递减，当递减至0时，就可以唤醒等待程序，继续完成后面的动作。

查看demo：  
ThreadDigger\src\main\java\com\dig\currentPackage\CountDownLatch\CountDownLatchDemo.java





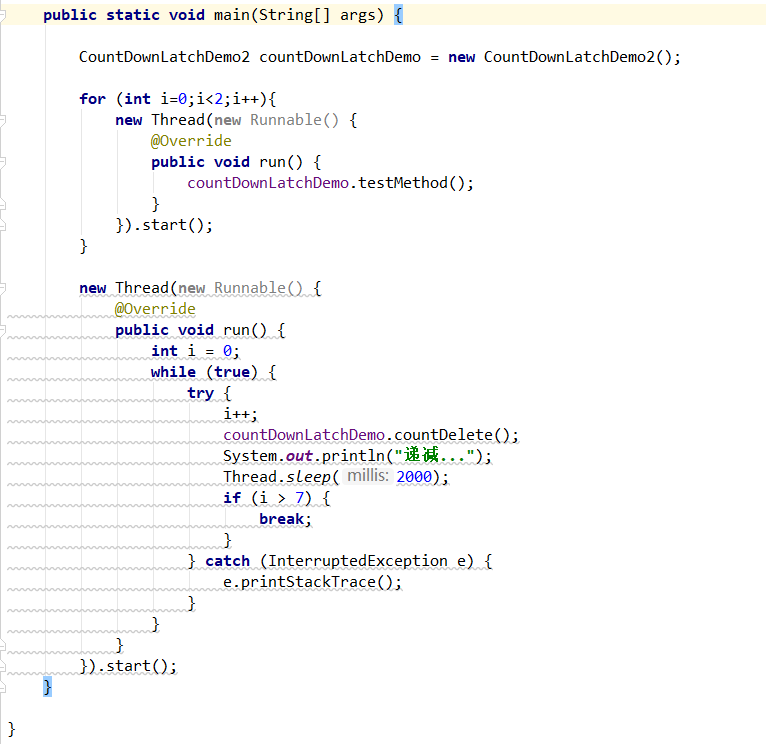
查看执行结果：



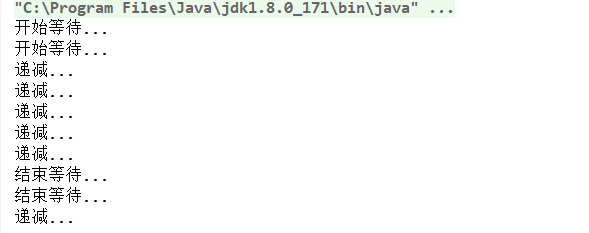
可以看到，当**countDownLatch**递减至0时，阻塞等待的代码才能继续往下走。注意，这里**countDownLatch**设置超时时间，如果过了超时时间，那么阻塞等待的代码也会继续往下走。

当然，也可以实现多个线程一起使用一个**countDownLatch** 实现等待；

看demo：  
ThreadDigger\src\main\java\com\dig\currentPackage\CountDownLatch\CountDownLatchDemo2.java



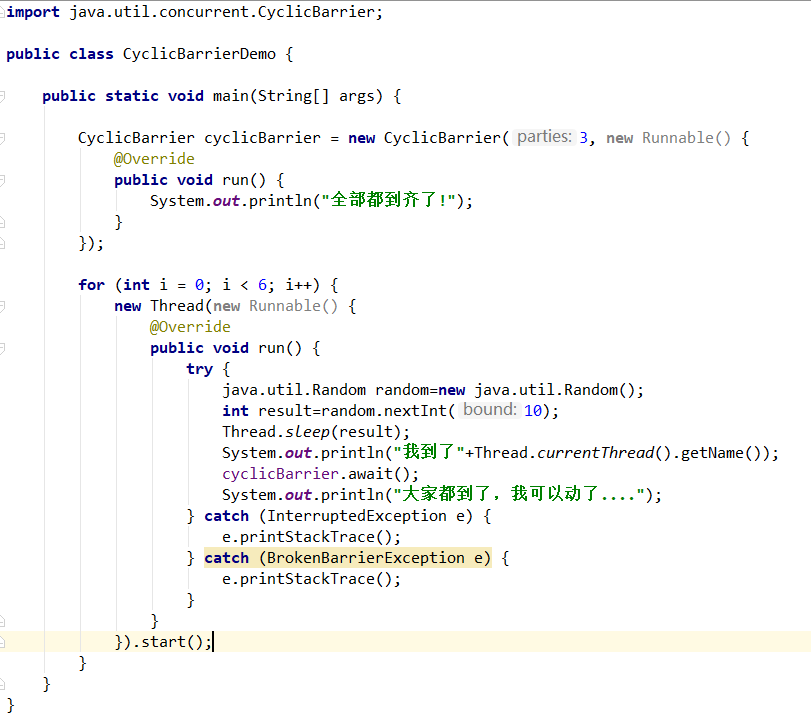
查看打印：



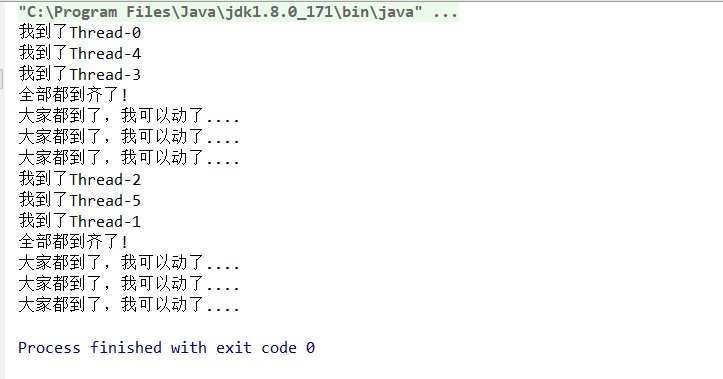
接下来，再看CyclicBarrier，CyclicBarrier跟CountDownLatch不一样的地方在于，CountDownLatch是某一个或某几个线程等待计数为0的时刻，而CyclicBarrier则是所有的线程都在等待CyclicBarrier为0的时候，CyclicBarrier不为0时，线程就一直呈阻塞状态。

看demo：

ThreadDigger\src\main\java\com\dig\currentPackage\CyclicBarrier\CyclicBarrierDemo.java



查看打印：



可以看到，当每3线程都执行await后，这3个线程才能告诉别人“我可以动了”。所以这里的CyclicBarrier的3并非单纯的只做3次递减，而是多个3次递减。同样的await可以设置超时间，如果过了超时时间，还没够3次，那么该线程就不再继续等待下去了。

### 2.9.2 阶段器phaser

读音：phaser ['feɪzə]

Phase是相对于CyclicBarrier的复杂版，可以分段阻塞。我们直接使用demo来进行讲解：

Demo：

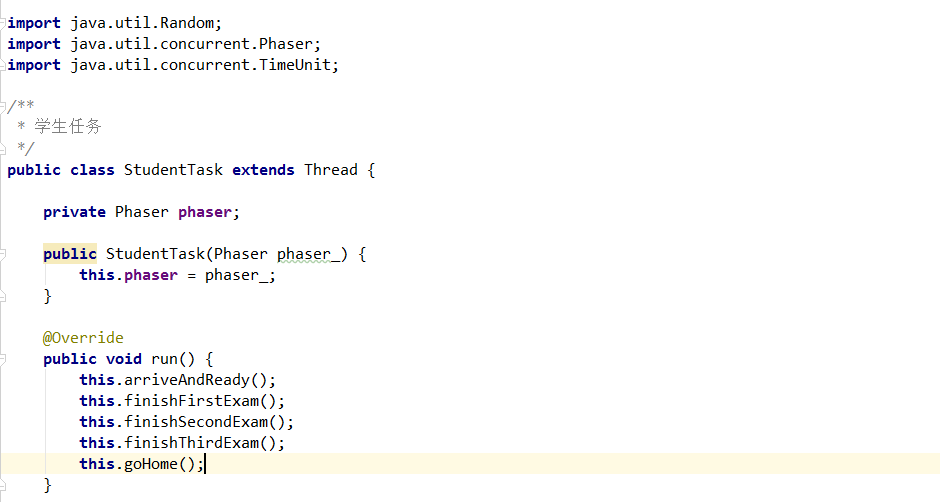
ThreadDigger\src\main\java\com\dig\currentPackage\phaser\PhaserDemo.java

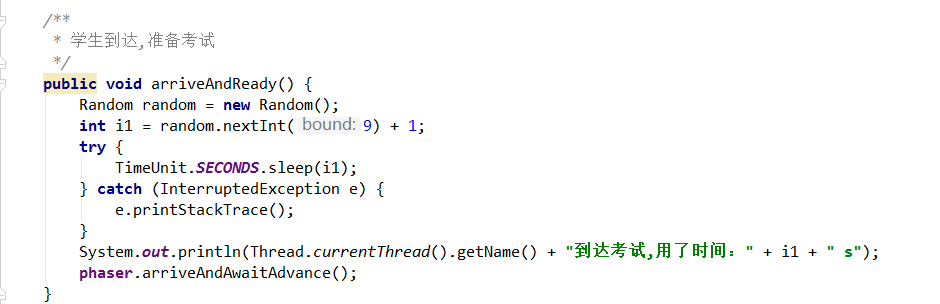
首先，先实现一个分段器：

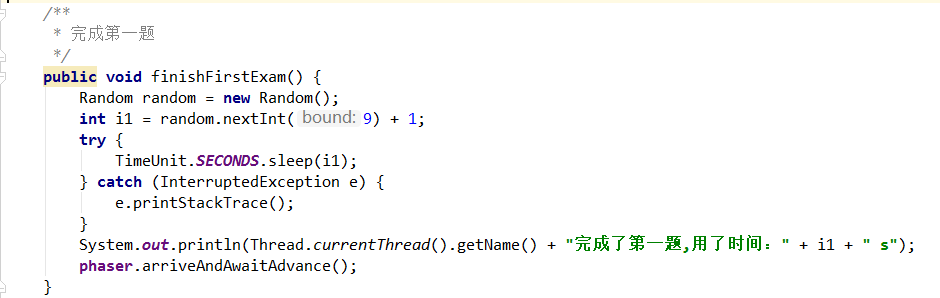


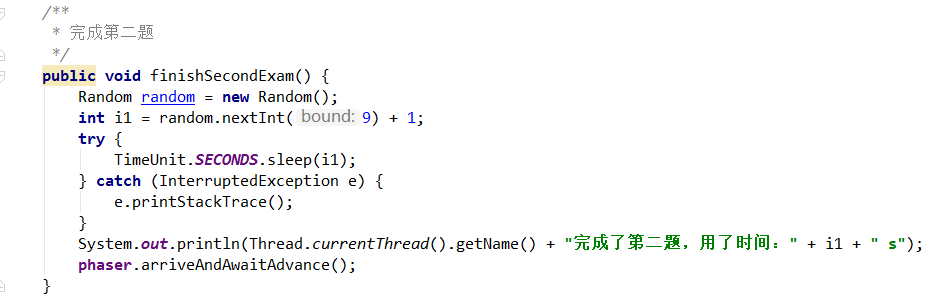
这里复写onAdvance（）方法，用来说明：当多个线程完成一个阶段的事情时，输入一些打印。这里的0/1/2/3，是阶段器自动计算的，当足够的线程都执行**phaser**.arriveAndAwaitAdvance()时，那么就是第一个阶段，对应0，当这批线程再次执行**phaser**.arriveAndAwaitAdvance()时，那么就是第二个阶段，对应1，一次类推。

接着我们来实现一个阶段任务：模拟一个学生考试的流程，当所有的学生都到达后，考试才能开始；当所有的学生完成第一题后，考生才能执行第二题；以此类推，每完成一个阶段，考生才能完成下一个阶段的事情。





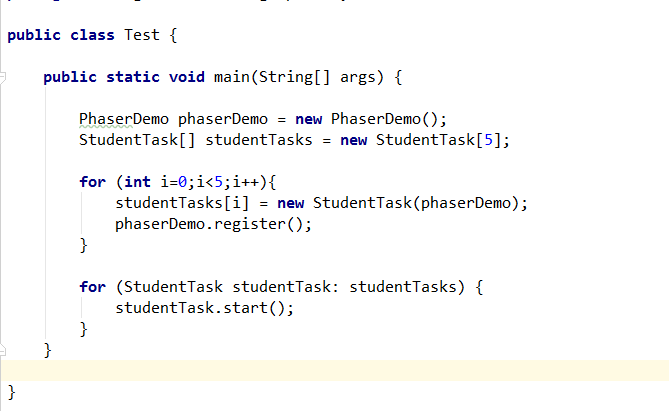






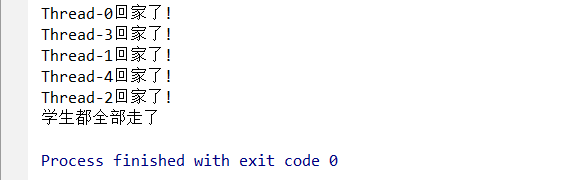


编写测试demo：使用phaserDemo.register()来指明，分阶段器管控多少个线程。phaserDemo.register()执行一次就管控一个线程，执行n次，就管控n个线程，n个线程执行了**phaser**.arriveAndAwaitAdvance()才算是完成一个阶段。



运行程序，查看打印：



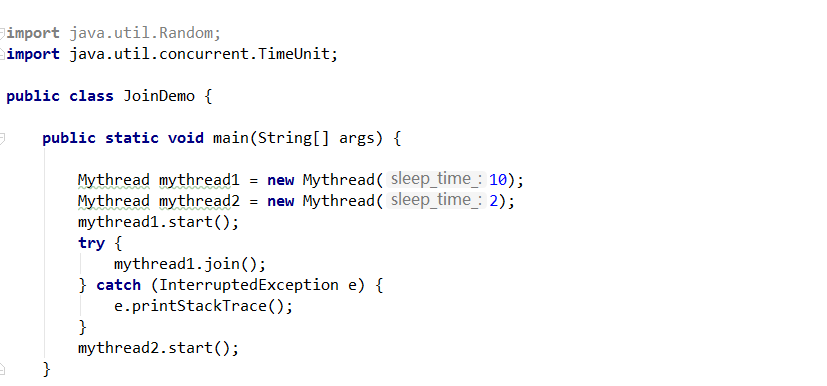


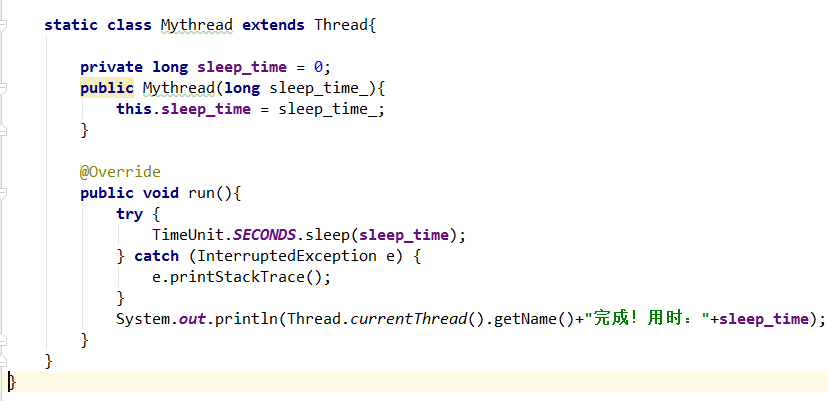
当onAdvance返回true时，阶段器就完成了它的使命，不再对线程管控或者阻塞线程。

### 2.9.3 串行join( )

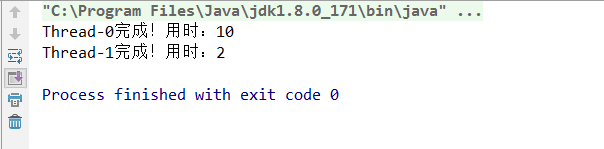
Join（）的作用是让线程从并行变成串行，原理结合下面demo说明：

当mythread1启动后，执行了mythread1.join（）方法，该步骤会回收当前线程main方法的cpu时间片，只有当mythread1线程执行完成，才会重新分配时间片给main，以执行mythread1.join（）后面的mythread2的代码。





查看打印：



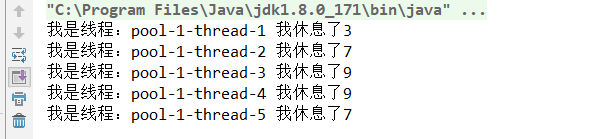
## 2.9.4 completionService和ExecutorCompletionService

completionService的作用是解决Future的阻塞同步性，多个Callable运行时，要打印所有的Future返回结果，那么按先执行的Callable先返回Future，而completionService的作用是获取提前完成任务的Callable的future结果。看demo：

ThreadDigger\src\main\java\com\dig\currentPackage\compleService\noCompletionServiceDemo\NoCompletionServiceDemo.java



接着运行main方法，查看返回结果：



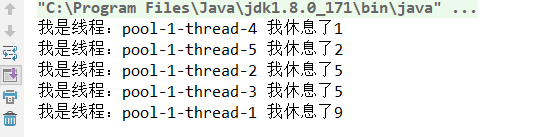
可以看到返回结果的顺序，并非按照使用时间，而是执行的顺序，即使线程5相对3和4较快，但还是最后才返回。

现在使用completionService看看效果：

Demo:  
ThreadDigger\src\main\java\com\dig\currentPackage\compleService\ComplationServiceDemo\CompletionServiceDemo.java



查看打印：



可以看到，返回的内容按照执行时间的长短。

ExecutorCompletionService跟completionService的区别在于，ExecutorCompletionService自带线程池。

## 2.10 线程池**ThreadExcutor**

# 3. IO流

## 3.1 IO和NIO的区别

主要区别是阻塞和非阻塞。

阻塞在这里的定义是，一个线程向一个地址发送请求，在获取到应答时，线程会一直处于一个阻塞的状态，而不会去做别的事情。

非阻塞则相反，一个线程向一个地址发送请求，如果没有应答，线程不会一直处于阻塞等待的状态，而是可以自由的去进行别的事情。原理上NIO通过监听去代替阻塞等待的操作。

## 3.2 netty

Netty是由JBOSS提供的一个java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

也就是说，Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户、服务端应用。Netty相当于简化和流线化了网络应用的编程开发过程，例如：基于TCP和UDP的socket服务开发。

# 4. 框架知识

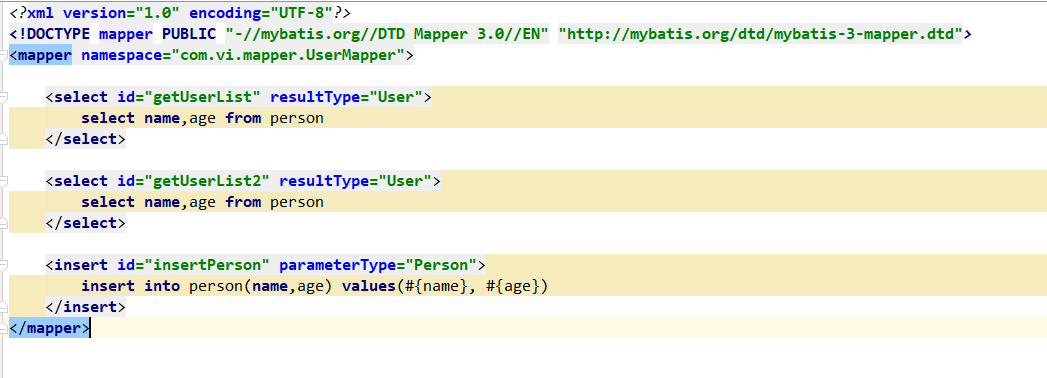
## 4.1 mybatis

Mybatis还不够熟，现在我先来结合文章进行猜测：

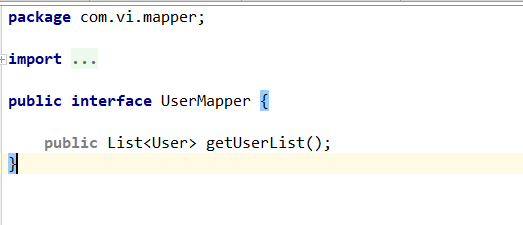
<https://www.cnblogs.com/yxfmp426756/p/8882852.html>

主要原理是结合了动态代理。Sqlsession和sqlsessionFactory是mybatis两个主要的类，sqlsession包涵了所有操作DB的方法。Mybatis是如何调起sqlsession的呢，mybatis有个很奇特的现象，只需要写对应执行方法的接口类，而不需要写它的实现类，这是因为使用了java的动态代理，结合我的项目工程来讲解：

有一个mapper：



也有对应的接口：



(而这个接口没有实现类)

代码在获取这个mapper的实现类时，SqlSession.getMapper实际上会调用了

MapperProxyFactory.newInstance 这个操作，返回一个代理类，而代理类在执行方法时，都会调起invoke方法，而在invoke方法里面，会调用mapperMethod.execute(sqlSession, args) 方法，这个实际就是sqlsession的入口。

# 5.Spring

## 5.1 Spring容器的实现原理

参考:

<https://blog.csdn.net/benhuo931115/article/details/74611464>

## 5.2 Spring注解

### 5.2.1 requestBody

### 5.2.2 requestParam

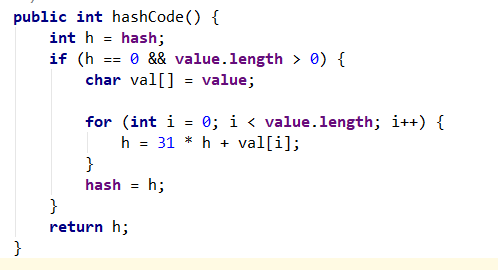
## 5.3 Spring上下文概念和使用场景

5.3 如何动态注册bean

# .10.其它不归类问题

## 10.1 String的hashcode是如何取得的？

String重写了Object的equal方法和hashcode方法：



有点困难，可以参考：

https://blog.csdn.net/myspacedemen/article/details/53353480

## 10.2 jvm的分类

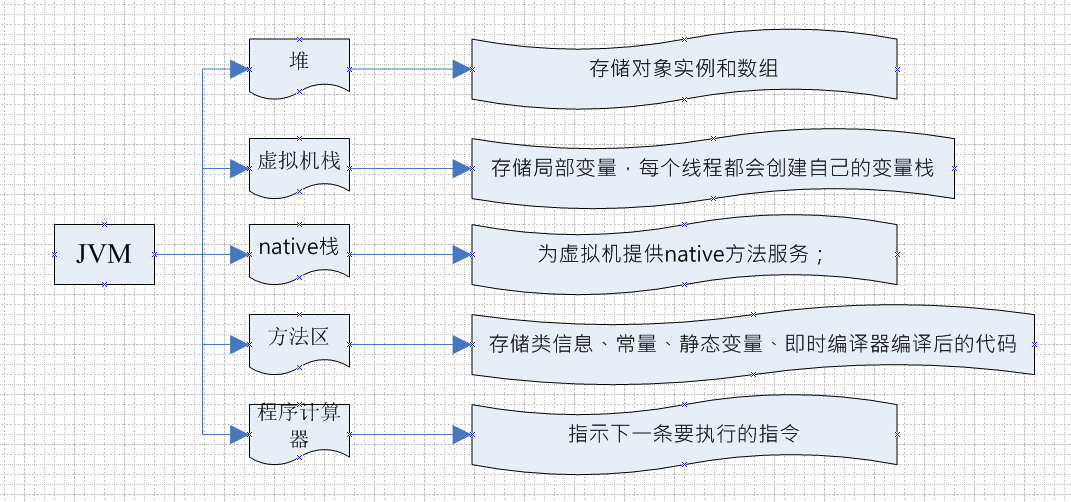
native栈：为虚拟机提供native方法服务；

堆：存储对象实例和数组；

方法区：存储类信息、常量、静态变量、即时编译器编译后的代码；

程序计算器：指示下一条要执行的指令；

虚拟机栈：存储局部变量，每个线程都会创建自己的变量栈；



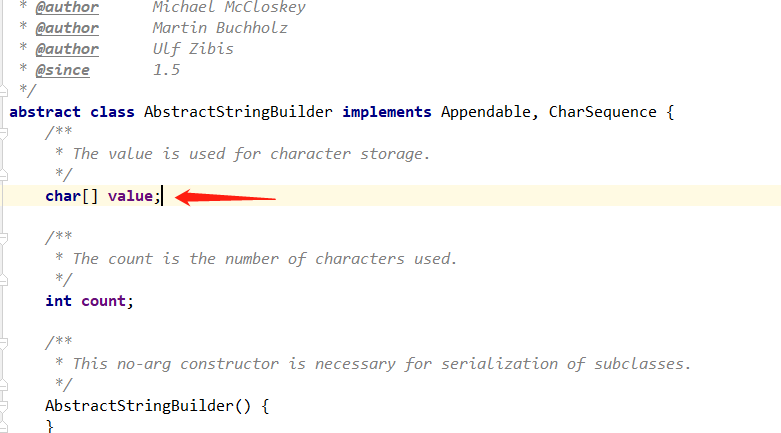
### 10.2.1 新生代和老年代

新生代和老年代存放的内容？

新生代什么时候变成老年代？

## 10.3 String、StringBuffer、StringBuilder的区别

* String一旦赋值就是一个对象，修改String内容会重新销毁和创建新的String对象；
* 对StringBuffer和StringBuilder赋值实际是改变关联的属性值(如截图)，而并不会销毁原本的StringBuffer或StringBuilder对象；
* StringBuffer和StringBuilder的区别在于，StringBuffer是使用synchronized来保证线程安全，而StringBuilder是线程不安全的。



## 10.4 http和tcp

参考文档：

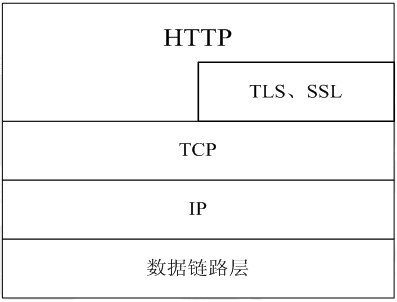
<https://www.cnblogs.com/wangning528/p/6388464.html>

### 10.4.1 http的概念

* http是超文本传输协议的缩写，用于把超文本从www服务器传输到浏览器的一种协议。超文本的官方解释是将各种不同空间的文字信息组织在一起的网状文本，通俗的讲就是展示给用户的文字等信息。

### 10.4.2 http和https的区别

* http位于tcp之上，有时也承载于TLS或者SSL之上，当承载于TLS或SSL之上时，就成了https。



### 10.4.3 http的工作流程

http的工作流程很简单

第一步：建立连接；

第二步：发送请求；

第三步：返回应答；

第四步：展示结果、断开连接；

### 10.4.4 http的无状态，这里的状态是如何理解？

这里的无状态有以下几种意思：

1）同一个请求不存在任何关系，服务端没有记忆能力，相同的请求，上一次对下一次不造成影响；

2）服务端不会保存客户端的状态，客户端每次访问服务端都必须带上自己的状态；

3）cookie和session的存在才让服务端变得有状态；

### 10.4.5 http的长连接和短连接

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

Connection:keep-alive

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

10.4.6 TCP协议

客户端和服务端要完成会话，必须要建立连接，会话完成，则要释放连接。在这里，连接的建立需要三次握手，而释放则需要四次握手。这里的握手协议称之为TCP协议。

### 10.4.7 http和socket的区别？

首先，http是一种超文本传输协议，而socket是一个接口，封装了TCP协议，socket接口的原理是使用TCP协议进行数据流的传输。所以这里是不同范畴的东西，没得比较。

### 10.4.8 servlet的生命周期

参考：

<https://blog.csdn.net/danielzhou888/article/details/70835418>

初始化 -- 实例化 -- 服务响应 -- 销毁，释放资源

具体过程：

1. Web Client 向Servlet容器（Tomcat）发出Http请求
2. Servlet容器接收Web Client的请求
3. Servlet容器创建一个HttpRequest对象，将Web Client请求的信息封装到这个对象中。
4. Servlet容器创建一个HttpResponse对象
5. Servlet容器调用HttpServlet对象的service方法，把HttpRequest对象与HttpResponse对象作为参数传给HttpServlet 对象。
6. HttpServlet调用HttpRequest对象的有关方法，获取Http请求信息。
7. HttpServlet调用HttpResponse对象的有关方法，生成响应数据。
8. Servlet容器把HttpServlet的响应结果传给Web Client。

## 10.5 反射的原理和类加载器

参考：<http://www.importnew.com/23902.html>

类加载参考：

https://www.cnblogs.com/fingerboy/p/5456371.html

反射实际上是用了Class对象的提供的api，让用户可以获取字节码的属性、方法等信息，并且对这种属性和方法进行实例化并返回。

类加载：当系统需要某个类时，jvm会把这个类的.class文件加载到jvm的内存中，这个过程叫类加载；

参考：

# 待解决问题：

* HashMap使用hash表来存储键值对用了什么数组？HashEntry又是什么东西？