## 一、Java内存模型（JMM），基础中的基础，关键点

### 1．java指令重排序概念，优点

Java重排序是指编译器、操作系统、cpu为了高效执行，对指令进行重排序，java程序从编译到最终CPU执行，可能会经历两次重排序，编译器重排序和运行时重排序。

编译期重排序的典型就是通过调整指令顺序，在不改变程序语义的前提下，尽可能减少寄存器的读取、存储次数，充分复用寄存器的存储值。

运行期重排序是指CPU根据自身规则对执行指令重排序，提高执行效率。

Jvm通过if-as-serial保证单线程程序无论如何重排序都能得到相同结果。

优点：提高执行效率。

### 2．volatile关键字，要深入理解，内存屏障的概念和作用（禁止指令重排序）

volatile关键字修饰变量，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，这将保证线程对该变量的原子操作对其他线程是可见的。volatile是无锁的，比synchronized要高效。volatile修饰的变量禁止前后指令重排序，起到内存屏障作用。其实现原理是线程通过嗅探内存总线变化而失效自身本地内存缓存。

volatile通过相关规则插入内存屏障，来禁止普通变量和volatile变量之间读写操作的重排序，确保程序得以正确执行。

### 3．final关键字的语义，都知道不可变性，其实还有就是指令重排序相关。

final修饰的变量，赋值后不可再改变引用，但引用内的值任可改变。比如：final oAbject=new Object(a,b),oAbject.a=a11;仍然是成立的。

final修饰的方法不可被子类重写，final修饰的类，不可被继承。

构造函数内对final变量的赋值，与随后使用该变量引用不能重排序。JMM禁止编译器把final域的写重排序到构造函数之外，通过在return前插入Store屏障实现。

### 4．线程的工作内存（也称本地内存）、共享内存

jvm堆和方法区是所有线程共享的，本地方法区、线程栈区、程序计数器是每个线程独享的。

### 5．synchronized关键字的语义，本质实现，进入前读取共享内存最新数据到本地内存，出来后刷新本地内存到共享内存。

依赖对象监视器实现锁机制，通过monitorEnter和monitorExit实现进入监视器和退出。synchronize作用在对象方法上，锁住对象实例this，作用于类静态方法，锁住的是class类，作用于代码块对象，锁住该对象。

### 6．一个线程内部，指令也存在重排序，但最终指令是有序的，多个线程，是没有顺序的。

as-if-serial确保单线程内，指令可以被重排序，但结果始终是一致的。而多线程则不一定，但得出的结果不会是凭空得来的。

## 二、java虚拟机模型（JVM），重点，也是基础。

### 1．各种概念，堆和栈，栈帧，持久区、新生代、老年代

jvm内存模型分为5个区：方法区、jvm堆、本地方法区、虚拟机线程栈、程序计数器。其中方法区和jvm堆是线程间共享的，而本地方法区、线程栈和程序计数器是线程独自享有的。

jvm堆又分为新生代和老年代，新生代分为eden区和两个surviver区（from和to），老年代分为年老代、持久代。

eden区是创建新对象时在此区分配，大对象则直接分配到老年代。youngc后仍存活的将会被移动到生存区，多次youngc后还存活的则移动到年老代。

surviver区分为from区和to区，此二区不是固定的，轮换着充当from和to，这一次youngc后把存活的对象放在from区，下一次from就会变成to区。

年老代主要保存生命周期较长的对象。

持久代主要保存class,method,filed等对象。但是在java8之后，持久代被删除了，取而代之的是在本地堆内存中开辟一个内存区域叫元空间来替代原有持久代。

### 2．常见的GC收集器，区别，实现原理。

标记清除算法。从根节点开始标记。没被标记的则删除。

复制算法。移动标记过的对象到另一块surviver区，清除没被移动的对象。清除后的内存是连续的。

标记-压缩算法。将被标记的移到内存集中区域。然后释放连续区域。适合老年代。

### 3．常见的JVM调优，工具（jconsole、java visual vm、jprofiler）

①jps [options:-qmlv] hosted 查看jvm运行进程状态。

②jstack 主要用来查看java进程内的线程堆栈信息。语法格式如下：

jstack [option] pid

jstack [option] executable core

jstack [option] [server-id@]remote-hostname-or-ip

③jmap主要用来查看堆内存使用状况。一般结合jhat使用。

④jstat JVM监控统计工具。

⑤hprof能够展现CPU使用率，统计堆内存使用情况。

摘抄自：<http://my.oschina.net/feichexia/blog/196575> JVM性能调优监控工具jps、jstack、jmap、jhat、jstat、hprof使用详解。

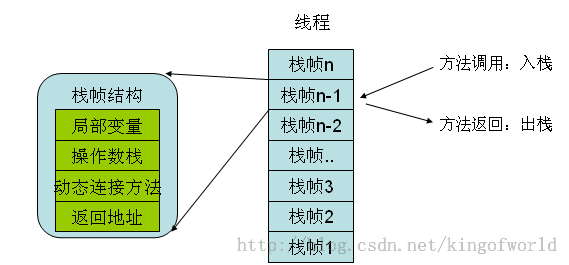
### 4．对象方法、静态属性、实例变量的jvm内存布局

JVM内存模型，主要分为方法区Method area、虚拟机栈VM stack、本地方法区native method area、jvm堆内存java heap、程序计数器program count register。

方法区：存放虚拟机加载类的元数据信息，包括常量、静态变量、即时编译后的代码。也即永久代。

虚拟机栈：存放线程信息，执行一个方法会创建栈帧存放该方法、变量表、操作数栈、动态链接方法、返回值、返回地址等信息。栈的大小可决定方法调用的可到达深度。-Xss设置栈的大小。如果请求的栈深度大于最大可用深度，则抛出stackOverflowError；如果栈是可动态扩展的，但没有内存空间支持扩展，则抛出OutofMemoryError。

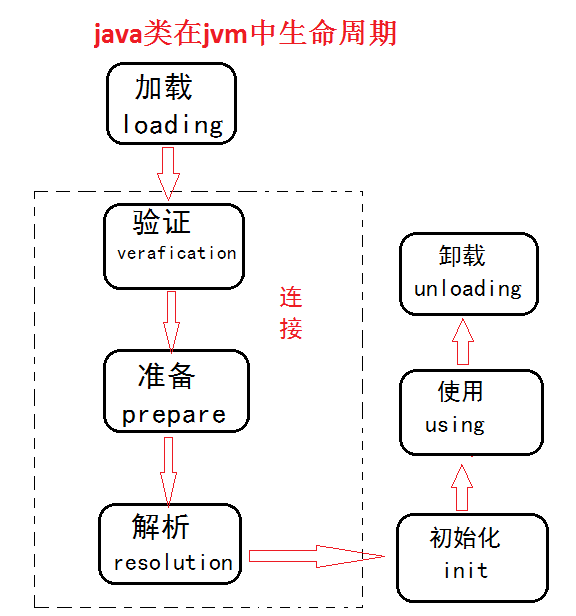
使用jclasslib工具可以查看class类文件的结构。下图为栈帧结构图：



本地方法区：存放native本地方法，该区native方法一般用c语言实现。

程序计数器：存放线程执行时指令快照，每个cpu按照时间片执行，操作系统时间片执行完后释放cpu资源给其他线程调度执行。此时线程下一条需要执行的指令被保存在计数器内。该区类似于快照。记录下线程执行的进度。

### 5．jvm加载类的过程，分别完成了什么。



\*.class加载寻找机制，先由jvm在启动时会运行产生一个①启动类加载器Bootstrap loader，自动加载ExtendedLoader 标准扩展类加载器，将其父设置为BootstrapLoader，ExtendedLoader会自动加载AppClassLoader，并将自己设置为其父。

当程序运行时，需要加载类时，

1、运行一个程序时，总是由AppClass Loader（系统类加载器）开始加载指定的类。

2、在加载类时，每个类加载器会将加载任务上交给其父，如果其父找不到，再由自己去加载。

3、Bootstrap Loader（启动类加载器）是最顶级的类加载器了，其父加载器为null.

Bootstrap Loader是用c语言实现的，所以实际监控时，看不到该加载器。

类加载有三种方式：

1、命令行启动应用时候由JVM初始化加载

2、通过Class.forName()方法动态加载

3、通过ClassLoader.loadClass()方法动态加载



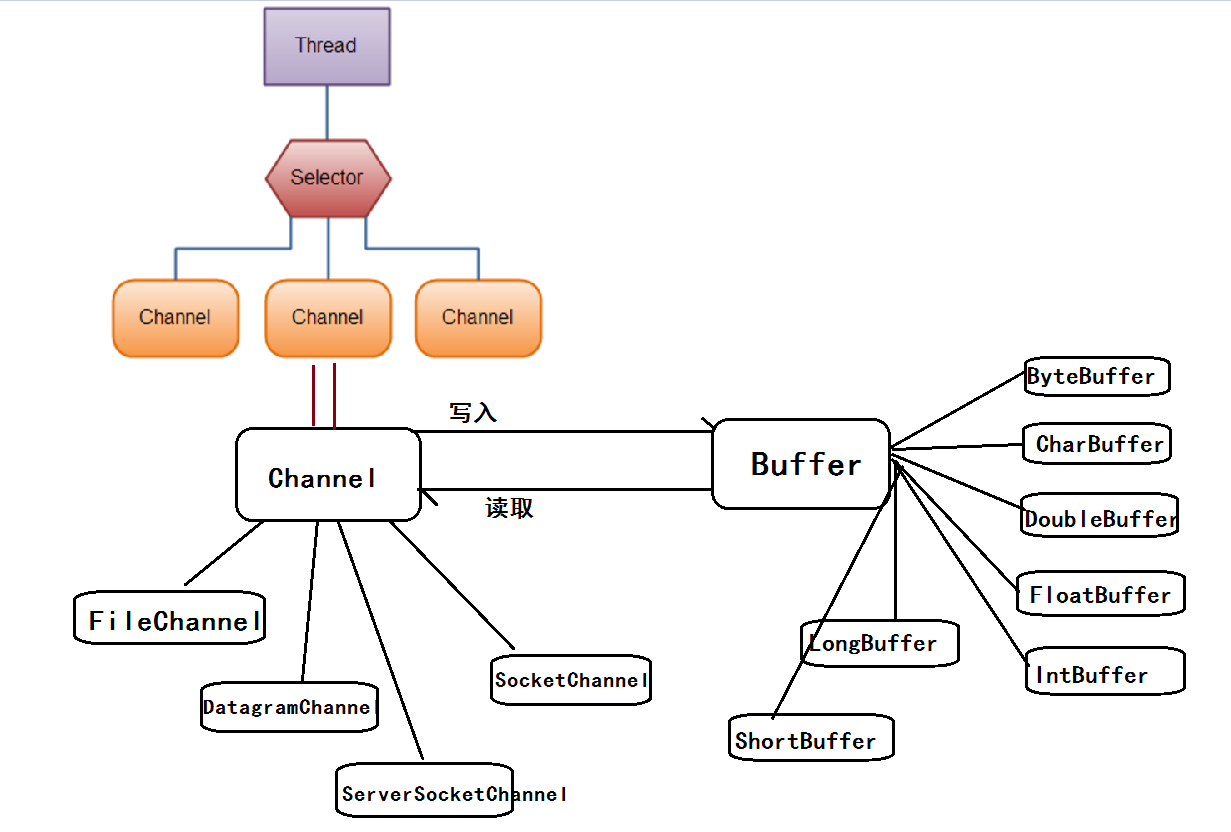
参考：<http://blog.csdn.net/a19881029/article/details/17068191>

参考：<http://lavasoft.blog.51cto.com/62575/184547/>

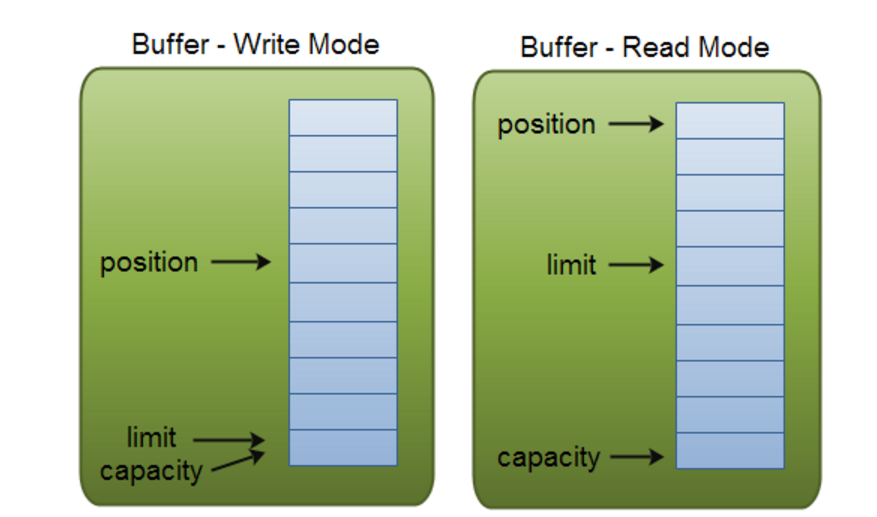
## 三、IO、NIO的关键点

### 1．简述对NIO的认识及重要的知识点

NIO主要由channel、buffer、selector三部分组成，channel类似于渠道管道、

提供了异步的套接字，socketchannel，具体机制,

☆buffer中很重要的三个变量：position、limit、capacity。读取数据模型如下：



1．JVM直接内存，DirectMemory直接在堆外分配内存，不受新生代eden区的young GC回收，受old区的full GC回收。其大小默认和堆大小一样，但不受堆大小-Xmx参数限制，有单独的参数控制：-XX:MaxDirectMemorySize=200M(此处可自由设置)。

2.内存文件映射：是由java进程直接建立起的一段虚拟内存空间和文件对象的关联映射关系。减少了数据在native堆和jvm堆之间的移动复制，所以高效。

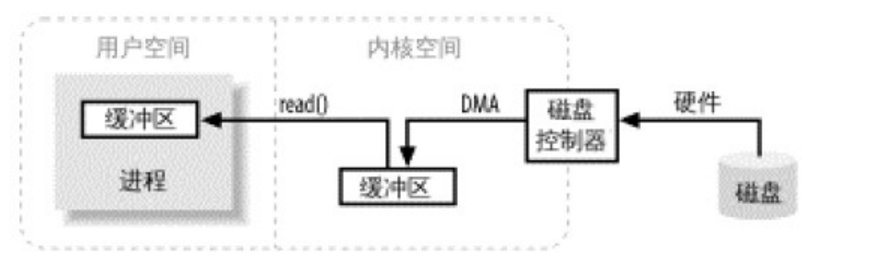
3.JVM最大内存不受操作系统内存大小限制，而是受操作系统虚拟内存大小限制。

传统IO操作中：数据流向是①从磁盘读入→内核空间→进程空间（JVM内存）。

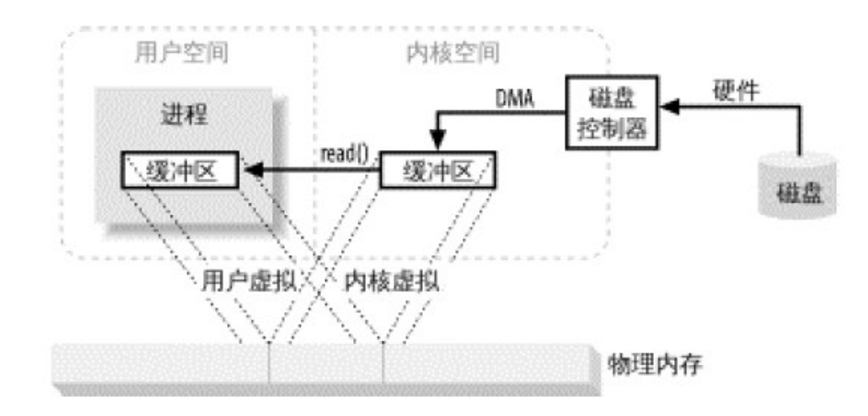
4.而内存文件映射：①磁盘空间→映射内存虚拟空间→用户空间操作读取。

5.NIO使用了虚拟映射技术提供了堆外缓冲区的功能，减少内存复制，传统IO是基于字节从磁盘到内核空间，再从内核空间到用户空间堆外，而NIO使用虚拟映射之后，数据不再复制，而是内核空间和用户空间同时虚拟映射一块内存作为缓冲区，用来填充或回写物理磁盘，少去了多次数据复制，所以操作效率更高。

传统数据流向：



NIO虚拟映射数据流量：

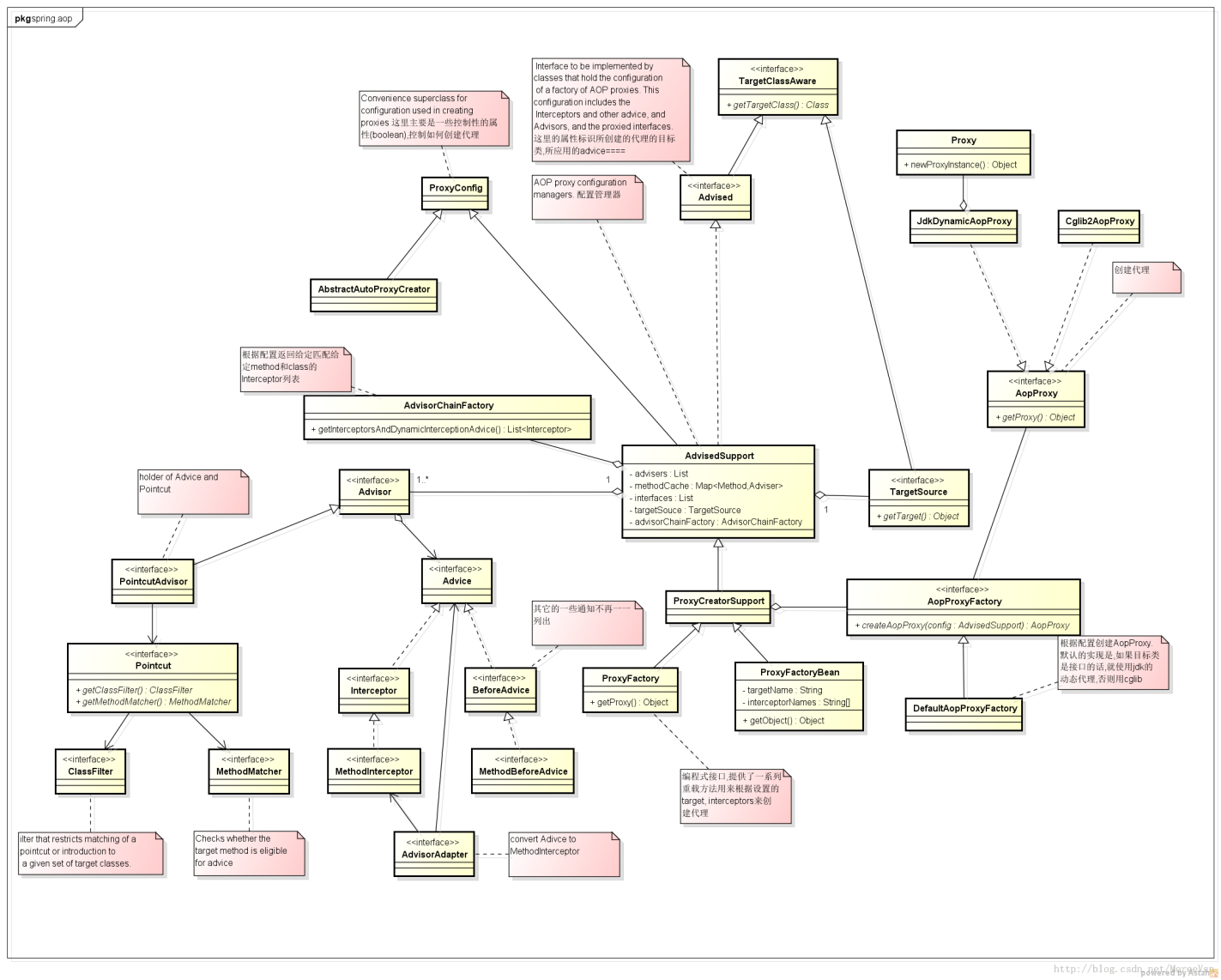


## 四、spring重点知识学习

### 1．AOP的实现，动态代理，分为jdk动态代理和cglib的方式，两种方式对被代理类的要求不一样。

SpringAop是指面向切面编程，利用反射技术和动态代理技术实现。动态代理又分为jdk动态道理和cglib代理。jdk动态代理只能代理实现了接口的类，是通过生成匿名子类重写被代理类的方法，在被代理类方法前后加入切面代码实现代理。而cglib可以代理未实现接口的类。但是cglib不能代理final方法（jdk动态代理也不能，接口定义时不允许定义成final），因为采用了非常底层的字节码技术，其原理是通过字节码技术为被代理类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

性能对比：创建对象，jdk动态代理性能高于cglib，方法调用方面，cglib高于jdk代理，单jdk6以上，jdk动态代理已经得到很大程度的提升，已经和cglib差不多。



思考：spring bean 如何设置bean里的方法互调，不走代理？所以写代码的时候，尽量把一些功能性方法写到spring管理外的类里。减少不相干的方法代理。

### 2．spring的扩展点，有几个，factoryBean和BeanPostProcessor

BeanFactory 类图如下：



根据不同的配置方式，使用不同的配置工厂创建和管理bean。BeanFactory持有对bean顶级操作接口，比如getBean(String name)，getBean(String name, Class requiredType) ，containsBean(String name)，isSingleton(String name)，getType(String name)，getType(String name)。

SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系。

FactoryBean实现了对bean对象的静态代理，可以装饰bean，当配置文件中 <bean>的class属性配置的实现类是FactoryBean时，通过getBean方法返回的不是FactoryBean本身，而是 FactoryBean#getObject()方法所返回的对象，相当于FactoryBean#getObject()代理了getBean()方 法。

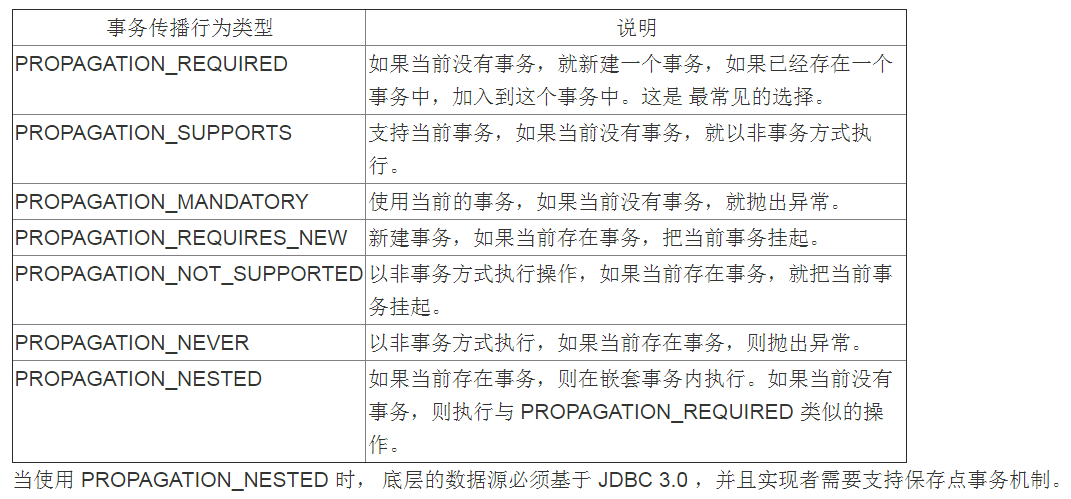
BeanPostProcessor能够实

现在bean初始化之前和初始化之后，做一些定制化的动作。

<http://www.cnblogs.com/ITtangtang/p/3978349.html> 请参考。

### 3.Spring事务相关知识：

#### 3.1Spring传播机制



#### 3.2事务隔离级别

数据库并发操作存在的异常情况：

1. 更新丢失（Lost update）： 两个事务都同时更新一行数据但是第二个事务却中途失败退出导致对数据两个修改都失效了这是系统没有执 行任何锁操作因此并发事务并没有被隔离开来。

事务行为描述：写事务A更新-写事务B更新-写事务A回滚

1. 脏读取（Dirty Reads）： 一个事务开始读取 了某行数据但是另外一个事务已经更新了此数据但没有能够及时提交。这是相当危险很可能所有操作都被回滚。

事务行为描述：读事务A-写事务B 读事务A读到旧数据。

1. 不可重复读取（Non-repeatable Reads）： 一 个事务对同一行数据重复读取两次但是却得到了不同结果。例如在两次读取中途有另外一个事务对该行数据进行了修改并提交。

事务行为描述：读事务A-写事务B-读事务A 两次读到不一致数据。

1. 两次更新问题（Second lost updates problem）： 无法重复读取特例，有两个并发事务同时读取同一行数据然后其中一个对它进行修改提交而另一个也进行了修改提交这就会造成 第一次写操作失效。

事务行为描述：写事务A先读后写-写事务B先读后写 事务B提交导致A提交丢失。

1. 幻读（Phantom Reads）： 也称为幻像（幻 影）。事务在操作过程中进行两次查询，第二次查询结果包含了第一次查询中未出现的数据（这里并不要求两次查询SQL语句相同）这是因为在两次查询过程中有 另外一个事务插入数据造成的。

事务行为描述：读事务A-读事务A 两次读到数据不一致

为了避免上面出现几种情况在标准SQL规范中定义了4个事务隔离级别，不同隔离级别对事务处理不同 。

1. 未授权读取（Read Uncommitted）： 也称 未提交读。允许脏读取但不允许更新丢失，如果一个事务已经开始写数据则另外一个数据则不允许同时进行写操作但允许其他事务读此行数据。该隔离级别可以通过 “排他写锁”实现。事务隔离的最低级别，仅可保证不读取物理损坏的数据。与READ COMMITTED 隔离级相反，它允许读取已经被其它用户修改但尚未提交确定的数据。

释义：写事务A禁止同时发生写事务B。 类似：写排他写锁。

1. 授权读取（Read Committed）： 也称提交 读。允许不可重复读取但不允许脏读取。这可以通过“瞬间共享读锁”和“排他写锁”实现，读取数据的事务允许其他事务继续访问该行数据，但是未提交写事务将 会禁止其他事务访问该行。SQL Server 默认的级别。在此隔离级下，SELECT 命令不会返回尚未提交（Committed） 的数据，也不能返回脏数据。

释义：读事务A允许读写事务，但写事务B禁止读写事务。类似：写排他读写锁

1. 可重复读取（Repeatable Read）： 禁止 不可重复读取和脏读取。但是有时可能出现幻影数据，这可以通过“共享读锁”和“排他写锁”实现，读取数据事务将会禁止写事务（但允许读事务），写事务则禁 止任何其他事务。在此隔离级下，用SELECT 命令读取的数据在整个命令执行过程中不会被更改。此选项会影响系统的效能，非必要情况最好不用此隔离级。

释义：读事务A禁止写事务。 类似：读排他写锁。

1. 串行（Serializable）： 也称可串行读。提 供严格的事务隔离，它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行。如果仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机 制保证新插入的数据不会被刚执行查询操作事务访问到。事务隔离的最高级别，事务之间完全隔离。如果事务在可串行读隔离级别上运行，则可以保证任何并发重叠 事务均是串行的。

释义：读写事务A禁止读写事务。 类似：完全排他锁。任何事务都只能串行执行。

#### 3.3注解式事务JDK动态代理方式和CGLIB方式的区别

a.基于JDK动态代理 ，可以将@Transactional放置在接口和具体类上，都会生效。

<tx:annotation-driven transaction-manager="txManager"/>

b.基于CGLIB类代理，只能将@Transactional放置在具体类上。

<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>

结论：在实际开发时全部将@Transactional放到具体类上，而不是接口上。

### 4.SpringMVC的扩展点

#### 4.1 Springmvc的第一个扩展点

HandlerMapping接口 -- 处理请求的映射

保存请求url到具体的方法的映射关系，，我们可以编写任意的HandlerMapping实现类，依据任何策略来决定一个web请求到HandlerExecutionChain对象的生成。

#### 4.2 SpringMVC的第二个扩展点

HandlerInterceptor 接口--拦截器

HandlerInterceptor，通过自定义拦截器，我们可以在一个请求被真正处理之前、请求被处理但还没输出到响应中、请求已经被输出到响应中之后这三个时间点去做任何我们想要做的事情。

#### 4.3springMVC的第三个扩展点

HandlerAdapter接口 – 处理适配器

真正调用Controller的地方，其实就是适配各种Controller。HandlerAdapter就是你可以提供自己的实现类来处理handler对象。

#### 4.4 SpringMVC的第四个扩展点

HandlerMethodArgumentResolver -- 处理方法参数解释绑定器

调用controller方法之前，对方法参数进行解释绑定（实现WebArgumentResolver接口，spring3.1以后推荐使用HandlerMethodArgumentResolver）；

#### 4.5 SpirngMVC的第五个扩展点

Converter -- 类型转换器

参数绑定时springmvc会对从前端传来的参数自动转化成方法定义的参数的类型，我们可自己定义此接口来实现自己的类型的转换

#### 4.6 SpirngMVC的第六个扩展点

ViewResolver接口 – 视图解析器

完成从ModelAndView到真正的视图的过程，ViewResolver接口是在DispatcherServlet中进行调用的，当DispatcherServlet调用完Controller后，会得到一个ModelAndView对象，然后DispatcherServlet会调用render方法进行视图渲染。

#### 4.7 SpringMVC提供的第七个扩展点：

HandlerExceptionResolver接口 --异常处理

## 五、jdk 并发包，java.util.concurrent

### 1．concurrentHashMap简介：

HashMap不是线程安全的，因为当多个线程同时put时，由于hashMap的容量不够了，就会扩容，扩容时需要将旧table里的数据复制到新扩容的table里，当两个线程同时发生扩容就会形成循环链表。导致死循环重复复制永远都有next。而concurrentHashMap是线程安全的，它的实现原理是分配多个锁，每个锁管一段数据。这样线程间竞争就会几何倍数减少，concurrentHashMap持有Segment数组，每个segment持有一个锁，segment的数量是2的N次方。concurrentHashMap通过二次hash散列到不同segment中的链表中。

Volatile定义每个segment的count和hashEntry的value，可以保证线程之间的可见性。但是写还是通过锁来控制的。

ConcurrentHashMap还对扩容进行了优化。比如插入时，是先进行判断是否足够，再扩容，扩容后再插入就能保证扩容操作不是多余的。避免插入后，不会再发生插入了，但是却扩容了，导致扩容动作是多余的。其次，扩容的只是segment，并非整个容器。

统计count进行优化，先进行两次统计，比较两次结果，然后再决定是否要加锁统计。除此之外，它还提供了一个modCount变量，记录每次put/remove/clean的操作计数器。

### 2．几种线程安全队列的大致实现，ConcurrentLinkedQueue，BlockingLinkedQueue

### 3．线程池，三种，实现原理，大致看看

### 4．原子类，AtomicInteger等，CAS操作中乐观锁的具体实现

## 六、sql方面

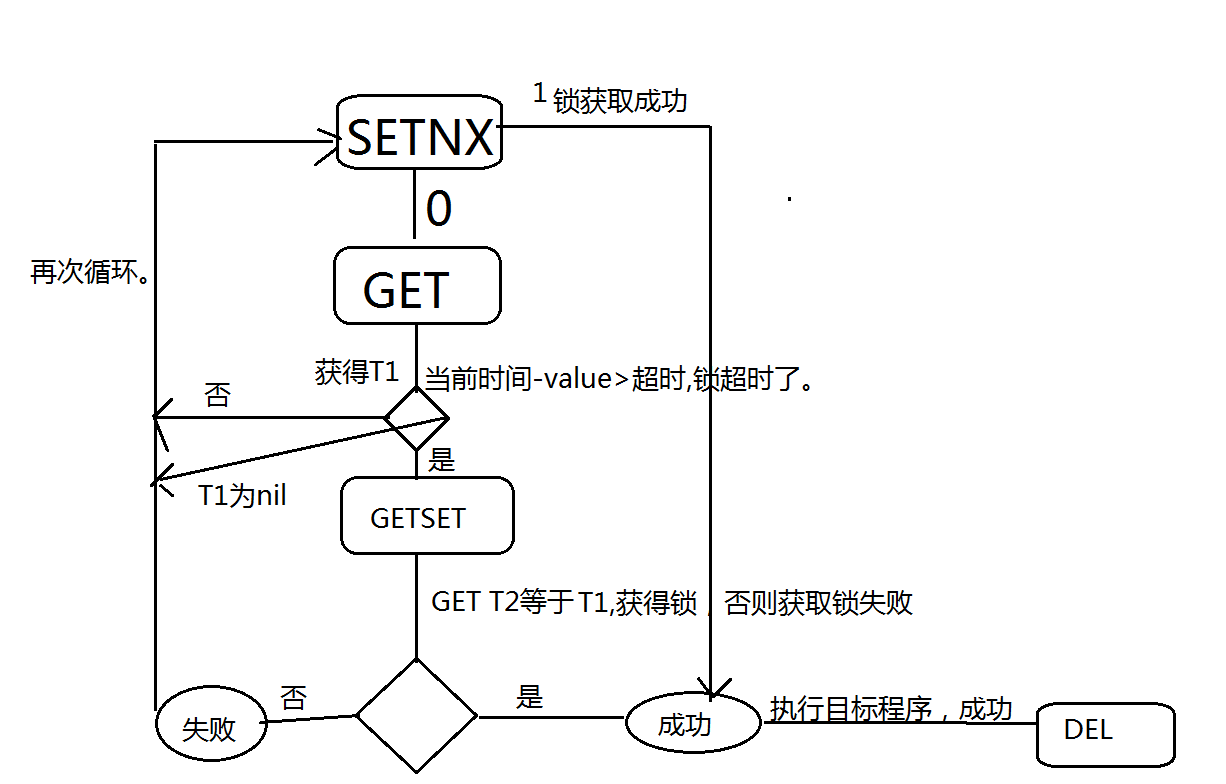
### 1．索引的数据结构，实现原理

### 2．常见的索引优化

### 3．分库分表的大致实现，spring多数据源+ThreadLocal（源码、实现）+关键字段的hash散列.

## 七、附加

### 1.分布式锁技术-redis版



Zookeeper版分布式锁-利用InterProcessMutex实现可重入。InterProcessSemaphoreMutex实现不可重入。

### 2.分布式事务

### 3.服务框架dubbo了解。

### 4.了解缓存系统redis，消息系统activeMQ，调度系统tbschedule。

### 5.常见设计模式，要好好掌握几个：

比如：模板方法，动态工厂模式，静态工厂模式，装饰者模式，单例模式（静态内部类实现），代理模式。

**常见23种，总体来说设计模式分为三大类：**

·**创建型模式，共五种**：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

·**结构型模式，共七种**：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

·**行为型模式，共十一种**：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

### 6.秒杀系统的设计