目录

[**一、** **Spring** 5](#_Toc484529552)

[1.1 Bean生命周期 5](#_Toc484529553)

[1.1.1 简略流程 5](#_Toc484529554)

[1.1.2 简略流程图 6](#_Toc484529555)

[1.2 Singleton Bean 7](#_Toc484529556)

[1.2.1 循环依赖 7](#_Toc484529557)

[1.3 BeanFactory 7](#_Toc484529558)

[1.3.1 BeanFactory 和FactoryBean的区别 7](#_Toc484529559)

[1.4 ApplicationContext 8](#_Toc484529560)

[1.4.1 简述 8](#_Toc484529561)

[1.4.2 FileSystemXmlApplicationContext 8](#_Toc484529562)

[1.4.3 XmlWebApplicationContext 8](#_Toc484529563)

[1.4.4 ClassPathXmlApplicationContext 8](#_Toc484529564)

[1.5 BeanPostProcesser 9](#_Toc484529565)

[1.6 InstantiationAwareBeanPostProcessor 9](#_Toc484529566)

[1.7 BeanDefinition 10](#_Toc484529567)

[1.7.1 RootBeanDefinition 10](#_Toc484529568)

[1.7.2 GenericBeanDefinition 10](#_Toc484529569)

[1.7.3 BeanDefinitionHolder 11](#_Toc484529570)

[1.8 NamespaceHandlerSupport 11](#_Toc484529571)

[1.9 Spring事务 12](#_Toc484529572)

[1.9.1 配置 12](#_Toc484529573)

[1.9.2 传播性 17](#_Toc484529574)

[1.9.3 隔离性 17](#_Toc484529575)

[1.9.4 不生效的情况 19](#_Toc484529576)

[**二、** **Java基础** 20](#_Toc484529577)

[2.1 集合 20](#_Toc484529578)

[2.1.1 HashMap 20](#_Toc484529579)

[2.1.2 ConCurrentHashMap 21](#_Toc484529580)

[2.1.3 ArrayList 21](#_Toc484529581)

[2.2 队列 22](#_Toc484529582)

[2.2.1 BlockingQueue 22](#_Toc484529583)

[2.3 生产者和消费者 23](#_Toc484529584)

[2.4 SPI机制 23](#_Toc484529585)

[2.5 增强型for循环 23](#_Toc484529586)

[**三、** **数据库** 25](#_Toc484529587)

[3.1 主从复制 25](#_Toc484529588)

[3.1.1 简述 25](#_Toc484529589)

[3.1.2 常见问题 25](#_Toc484529590)

[3.2 读写分离 26](#_Toc484529591)

[3.2.1 简述 26](#_Toc484529592)

[3.3 分库分表 26](#_Toc484529593)

[3.3.1 简述 26](#_Toc484529594)

[3.4 Sql优化 27](#_Toc484529595)

[3.4.1 MySQL 27](#_Toc484529596)

[3.5 悲观锁和乐观锁 30](#_Toc484529597)

[**四、** **线程** 31](#_Toc484529598)

[4.1 线程与进程 31](#_Toc484529599)

[4.2 线程池 31](#_Toc484529600)

[4.2.1 内部原理 31](#_Toc484529601)

[4.2.2 常见类型 32](#_Toc484529602)

[4.3 线程安全 33](#_Toc484529603)

[**五、** **JVM** 34](#_Toc484529604)

[5.1 JVM 内存模型 34](#_Toc484529605)

[5.2.1 程序计数器 34](#_Toc484529606)

[5.2.2 虚拟机栈 34](#_Toc484529607)

[5.2.3 本地方法区 35](#_Toc484529608)

[5.2.4 JAVA堆 35](#_Toc484529609)

[5.2.5 方法区 36](#_Toc484529610)

[5.2 OOM原因 36](#_Toc484529611)

[5.2.1 Heap 36](#_Toc484529612)

[5.2.2 StackOverflowError 37](#_Toc484529613)

[5.2.3 Perm 37](#_Toc484529614)

[5.2.4 GC 38](#_Toc484529615)

[5.2.5 Native Thread Created 38](#_Toc484529616)

[5.2.6 Allocate Huge Array 39](#_Toc484529617)

[5.2.7 Small Swap 39](#_Toc484529618)

[5.2.8 Exhausted Native Memory 40](#_Toc484529619)

[5.3 GC算法 41](#_Toc484529620)

[5.3.1 标记-清除算法(Mark-Sweep) 41](#_Toc484529621)

[5.3.2 复制算法(copying) 41](#_Toc484529622)

[5.3.3 标记-压缩算法(Mark-compact) 41](#_Toc484529623)

[5.4 优化参数 41](#_Toc484529624)

[**六、** **分布式** 43](#_Toc484529625)

[6.1 分布式主键生成策略 43](#_Toc484529626)

[6.2 分布式锁 43](#_Toc484529627)

[6.2.1 Redis 43](#_Toc484529628)

[6.2.2 ZooKeeper 43](#_Toc484529629)

[6.3 分布式事务 44](#_Toc484529630)

[6.3.1 一阶段提交 44](#_Toc484529631)

[6.3.2 二阶段提交 44](#_Toc484529632)

[6.3.3 三阶段提交 46](#_Toc484529633)

[**七、** **RPC** 49](#_Toc484529634)

[7.1 序列化方式 49](#_Toc484529635)

[7.2 Dubbo（了解） 49](#_Toc484529636)

[7.3 Netty（了解） 49](#_Toc484529637)

[**八、** **大数据** 49](#_Toc484529638)

[8.1 Hadoop（了解） 49](#_Toc484529639)

[8.2 Spark（了解） 49](#_Toc484529640)

[**九、** **语言** 50](#_Toc484529641)

[9.1 Python（了解） 50](#_Toc484529642)

[9.2 Node.js（了解） 50](#_Toc484529643)

[**十、** **总结** 50](#_Toc484529644)

1. **Spring**

## 1.1 Bean生命周期

### 1.1.1 简略流程

1.Spring容器读取XML文件中bean的定义并实例化bean。（很笼统的说法）

2. 如果任何InstantiationAwareBeanPostProcessors和该bean相关，则调用相应InstantiationAwareBeanPostProcessor的postProcessBeforeInitialization()方法。

3.Spring根据bean的定义通过populateBean()方法填充属性值。

4.如果该Bean实现了BeanNameAware接口，Spring将bean的id传递给setBeanName()方法。

5.如果该Bean实现了BeanFactoryAware接口，Spring将beanfactory传递给setBeanFactory()方法。

6.如果任何BeanPostProcessors 和该bean相关，则调用相应BeanPostProcessor的postProcessBeforeInitialization()方法。

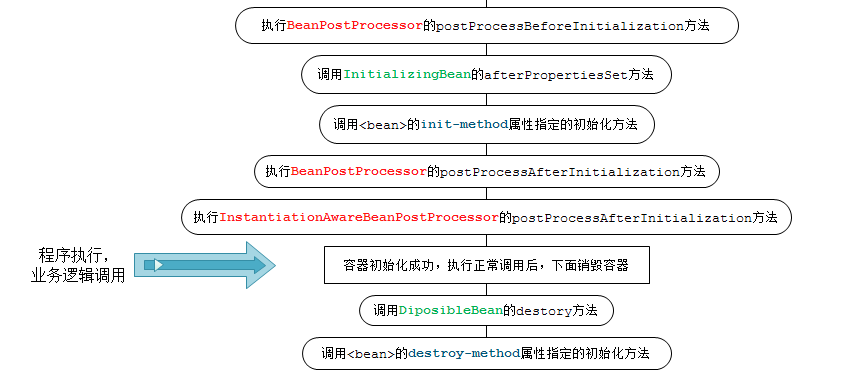
7.如果该Bean实现了InitializingBean接口，调用Bean中的afterPropertiesSet()方法。如果bean有初始化函数声明（即：init-method），调用相应的初始化方法。

8.如果任何BeanPostProcessors 和该bean相关，则调用相应BeanPostProcessor的postProcessAfterInitialization()方法。

9.如果任何InstantiationAwareBeanPostProcessors和该bean相关，则调用相应InstantiationAwareBeanPostProcessors的postProcessAfterInitialization()方法。

10.如果该bean实现了DisposableBean，调用destroy()方法。如果bean有销毁函数声明（即：destroy-method），调用相应的销毁方法。

### 1.1.2 简略流程图



## 1.2 Singleton Bean

### 1.2.1 循环依赖

#### 1.2.1.1 产生原因

当一个bean A的属性是另一个bean B时，这样A的创建则依赖B,同样的若B依赖C,C依赖A,这样就形成一个死循环，即循环依赖。容易造成内存溢出。

#### 1.2.1.2 Spring的解决方式

Spring只解决了单例情况下的属性循环依赖（非构造函数循环依赖）,首先通过无參构造器创建出一个exposedObject并将其放置在一个缓存Map中（提前暴露引用），在属性填充阶段，创建其他bean，而其他bean的属性填充阶段也依赖A时，Setter注入缓存中的exposedObject

## 1.3 BeanFactory

### 1.3.1 BeanFactory 和FactoryBean的区别

BeanFactory： 以Factory结尾，表示它是一个工厂类，是用于管理Bean的一个工厂;

FactoryBean：以Bean结尾，表示它是一个Bean，不同于普通Bean的是：它是实现了FactoryBean<T>接口的Bean，根据该Bean的Id从BeanFactory中获取的bean实际上是FactoryBean的getObject()返回的对象，而不是FactoryBean本身， 如果要获取FactoryBean对象，可以在id前面加一个&符号来获取。

## 1.4 ApplicationContext

### 1.4.1 简述

ApplicationContext继承自BeanFactory接口，除了包含BeanFactory的所有功能之外，在国际化支持、资源访问（如URL和文件）、事件传播等方面进行了良好的支持。

ApplicationContext容器实例化后会自动对所有的单实例Bean进行实例化与依赖关系的装配，使之处于待用状态。而BeanFactory容器实例化后并不会自动实例化Bean，只有当Bean被使用时BeanFactory容器才会对该Bean进行实例化与依赖关系的装配。

### 1.4.2 FileSystemXmlApplicationContext

从指定的文件系统路径中寻找指定的XML配置文件，找到并装载完成ApplicationContext的实例化工作。

### 1.4.3 XmlWebApplicationContext

从Web应用中寻找指定的XML配置文件，找到并装载完成ApplicationContext的实例化工作。这是为Web工程量身定制的，使用WebApplicationContextUtils类的getRequiredWebApplicationContext方法可在JSP与Servlet中取得IoC容器的引用。

### 1.4.4 ClassPathXmlApplicationContext

从类路径ClassPath中寻找指定的XML配置文件，找到并装载完成ApplicationContext的实例化工作。

## 1.5 BeanPostProcesser

Factory hook that allows for custom modification of new bean instances,

e.g. checking for marker interfaces or wrapping them with proxies.

ApplicationContexts can autodetect BeanPostProcessor beans in their bean definitions and apply them to any beans subsequently created.

Plain bean factories allow for programmatic registration of post-processors,

applying to all beans created through this factory.

Typically, post-processors that populate beans via marker interfaces or the like will implement postProcessBeforeInitialization,

while post-processors that wrap beans with proxies will normally implement postProcessAfterInitialization.

简单来说，如果我们需要在Spring容器完成Bean的实例化、配置和其他的初始化前后添加一些自己的逻辑处理，我们就可以定义一个或者多个BeanPostProcessor接口的实现，然后注册到容器中。

## 1.6 InstantiationAwareBeanPostProcessor

InstantiationAwareBeanPostProcessor是BeanPostProcessor的子接口，可以在Bean生命周期的另外两个时期提供扩展的回调接口。可看之前的bean生命周期的流程图。

## 1.7 BeanDefinition

### 1.7.1 RootBeanDefinition

A root bean definition represents the merged bean definition that backs a specific bean in a Spring BeanFactory at runtime.

It might have been created from multiple original bean definitions that inherit from each other,

typically registered as GenericBeanDefinitions.

A root bean definition is essentially the 'unified' bean definition view at runtime.

简单来说，RootBeanDefinition是个最基本的复杂类型的BeanDefinition。

### 1.7.2 GenericBeanDefinition

GenericBeanDefinition is a one-stop shop for standard bean definition purposes.

Like any bean definition, it allows for specifying a class plus optionally constructor argument values and property values.

Additionally, deriving from a parent bean definition can be flexibly configured through the "parentName" property.

简单来说，GenericBeanDefinition是一站式的标准的BeanDefinition。除了具有指定类、可选的构造参数值和属性参数这些其它bean definition一样的特性外，它还具有通过parenetName属性来灵活设置parent bean definition。

### 1.7.3 BeanDefinitionHolder

Holder for a BeanDefinition with name and aliases.

Can be registered as a placeholder for an inner bean.

Can also be used for programmatic registration of inner bean definitions.

If you don't care about BeanNameAware and the like,

registering RootBeanDefinition or ChildBeanDefinition is good enough.

简单来说，BeanDefinitionHolder持有一个BeanDefinition，名称，和别名数组。在spring内部，它用来临时保存BeanDefinition来传递BeanDefinition。

## 1.8 NamespaceHandlerSupport

Support class for implementing custom NamespaceHandlers.

Parsing and decorating of individual Nodes is done via BeanDefinitionParser and BeanDefinitionDecorator strategy interfaces, respectively.

Provides the registerBeanDefinitionParser and registerBeanDefinitionDecorator methods for registering a BeanDefinitionParser or BeanDefinitionDecorator to handle a specific element.

简单来说，主要用于xml标签的解析注册处理（包括默认的官方标签和自定义标签），

部分解析流程可参考<http://blog.csdn.net/boneix/article/category/6846164>

## 1.9 Spring事务

### 1.9.1 配置

根据代理机制的不同，总结了五种Spring事务的配置方式，配置文件如下：

第一种方式：每个Bean都有一个代理

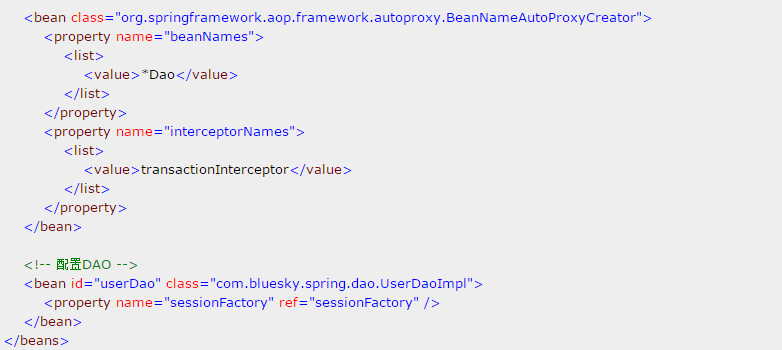


第二种方式：所有Bean共享一个代理基类



第三种方式：使用拦截器



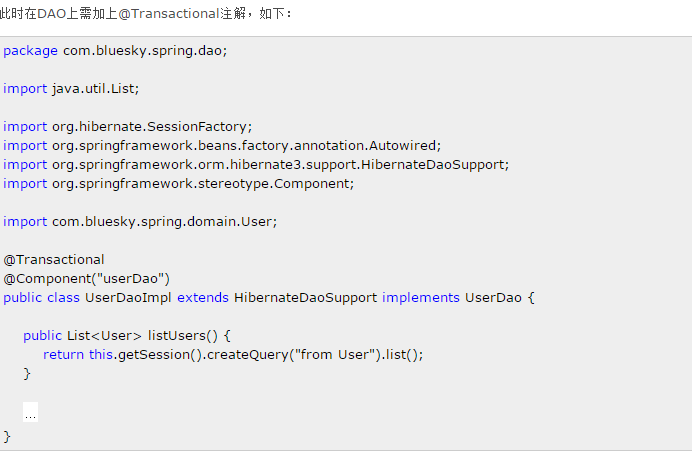


第四种方式：使用tx标签配置的拦截器

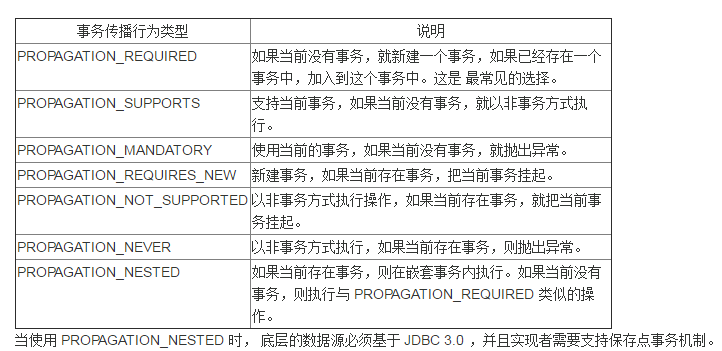


第五种方式：全注解





### 1.9.2 传播性



### 1.9.3 隔离性

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。

2. 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。

3. 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。

4. 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。

5. 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。

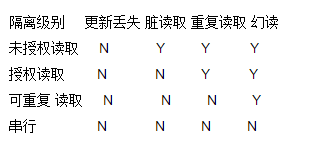
为了避免上面出现几种情况在标准SQL规范中定义了4个事务隔离级别，不同隔离级别对事务处理不同 。

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。

2. 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。

3. 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。

4. 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。



### 1.9.4 不生效的情况

1、首先使用如下代码 确认你的bean 是代理对象吗？

必须是spring定义（通过XML或注解定义都可以）的Bean才接受事务。

直接new出来的对象添加事务是不起作用的。

可以通过以下方式判断是否是代理对象：

AopUtils.isAopProxy(Object object)

AopUtils.isCglibProxy(Object object) //cglib

AopUtils.isJdkDynamicProxy(Object object) //jdk动态代理

2、如果你使用了springmvc，可能是context:component-scan重复扫描引起的。

3、如使用MySQL且引擎是MyISAM造成的（因为不支持事务），改成InnoDB即可。

1. **Java基础**

## 2.1 集合

### 2.1.1 HashMap

#### 2.1.1.1 Hash算法

典型的哈希算法包括 MD2、MD4、MD5 和 SHA-1。它是一种单向密码体制,即它是一个从明文到密文的不可逆的映射,只有加密过程,没有解密过程。同时,哈希函数可以将任意长度的输入经过变化以后得到固定长度的输出。哈希函数的这种单向特征和输出数据长度固定的特征使得它可以生成消息或者数据。

#### 2.1.1.2 内部原理

初始容量，加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为0.75，一般情况下我们是无需修改的。

当我们想一个HashMap中添加一对key-value时，系统首先会计算key的hash值，然后根据hash值确认在table中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其key的hash值。如果两个hash值相等且key值相等(e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的Entry的value覆盖原来节点的value。如果两个hash值相等但key值不等 ，则将该节点插入该链表的链头。

**参考资料**：<http://www.cnblogs.com/chenssy/p/3521565.html>

**面试资料**：<http://www.admin10000.com/document/3322.html>

#### 2.1.1.3 线程不安全的原因

addEntry（操作），resize（扩容：双倍）

### 2.1.2 ConCurrentHashMap

#### 2.1.2.1 内部原理

查找segment中的位置，分段实现线程安全，HashEntry内部为final ，所以删除时，复制删除节点前所有元素在和后面的重新组成新segment；添加时在表头添加。

### 2.1.3 ArrayList

#### 2.1.3.1 内部原理

(01) ArrayList 实际上是通过一个数组去保存数据的。当我们构造ArrayList时；若使用默认构造函数，则ArrayList的默认容量大小是10。

(02) 当ArrayList容量不足以容纳全部元素时，ArrayList会重新设置容量：新的容量=“(原始容量x3)/2 + 1”。

(03) ArrayList的克隆函数，即是将全部元素克隆到一个数组中。

(04) ArrayList实现java.io.Serializable的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。

## 2.2 队列

### 2.2.1 BlockingQueue

#### 2.2.1.1 内部原理

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

阻塞队列提供了四种处理方法:

1.抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常 。

2.返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null

3.一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。

4.超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

## 2.3 生产者和消费者

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

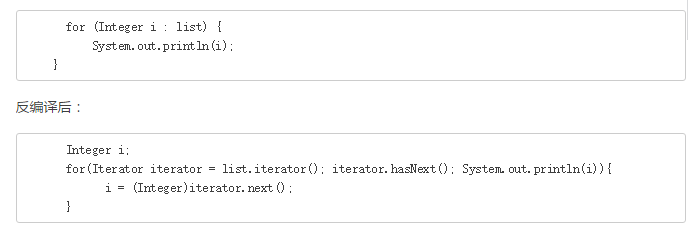
实现方式有很多，比如使用BlockingQueue自己实现，或者诸如MQ(ActiveMQ、RocketMQ)、Redis等等

## 2.4 SPI机制

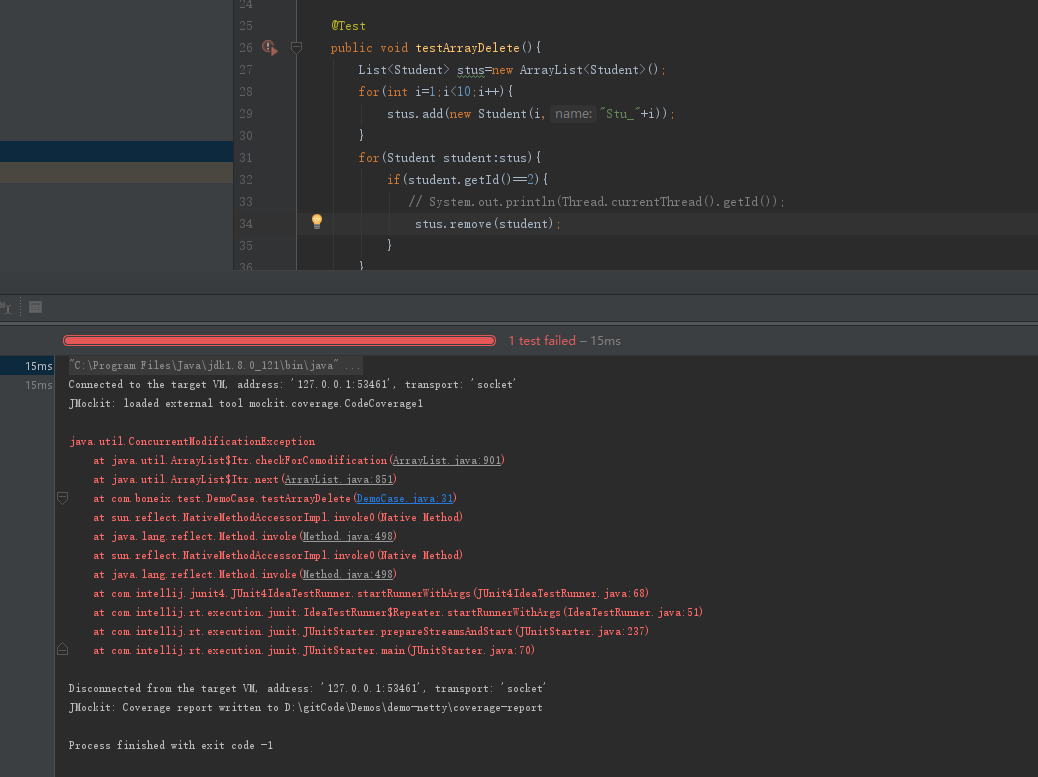
简介参考： <http://www.cnblogs.com/softlin/p/4321955.html>

## 2.5 增强型for循环

增强型for循环底层通过Iterator实现，

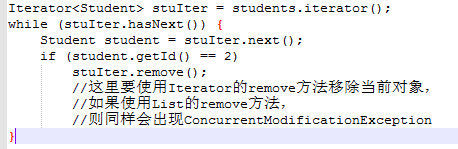


Iterator是工作在一个独立的线程中，并且拥有一个 mutex 锁。 Iterator被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出java.util.ConcurrentModificationException异常。



所以 Iterator 在工作的时候是不允许被迭代的对象被改变的。但你可以使用 Iterator 本身的方法 remove() 来删除对象，Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性。

正确的在遍历的同时删除元素的姿势：



1. **数据库**

## 3.1 主从复制

### 3.1.1 简述

主服务器将更新写入二进制日志文件，并维护文件的一个索引以跟踪日志循环。这些日志可以记录发送到从服务器的更新。当一个从服务器连接主服务器时，它通知主服务器从服务器在日志中读取的最后一次成功更新的位置。从服务器接收从那时起发生的任何更新，然后封锁并等待主服务器通知新的更新。所有对复制中的表的更新必须在主服务器上进行。

### 3.1.2 常见问题

#### 3.1.2.1主从数据库没有同步

如果主从库数据相差不大，或者要求数据可以不完全统一的情况，数据要求不严格，可以忽略错误后，继续同步。

如果从库数据相差较大，或者要求数据完全统一，则需要重新做主从，完全同步：

1. 先进入主库，进行锁表，防止数据写入
2. 进行数据备份
3. 查看master 状态
4. 把mysql备份文件传到从库机器，准备进行数据恢复
5. 停止从库的状态
6. 然后到从库执行mysql命令，导入数据备份
7. 设置从库同步，注意该处的同步点。
8. 重新开启从同步，查看同步状态

## 3.2 读写分离

### 3.2.1 简述

基本的原理是让主数据库处理事务性增、改、删操作（INSERT、UPDATE、DELETE），而从数据库处理SELECT查询操作。数据库复制被用来把事务性操作导致的变更同步到集群中的从数据库。

## 3.3 分库分表

### 3.3.1 简述

随着时间和业务的发展，库中的表会越来越多，表中的数据也越来越多。当表的存储数据在千万数据以上时，随着数据量的增加，数据操作增删改查的开销也会越来越大。

为了避免业务上一个功能的表的风险操作导致整个库宕机，引入了垂直切分和水平切分的概念。

#### 3.3.1.1 垂直切分

垂直切分的依据原则是：将业务紧密，表间关联密切的表划分在一起，例如同一模块的表。对于某些变动几率较低的共用概率较高的数据可以做适当的冗余。

#### 3.3.1.2 水平切分

当一个表中的数据量过大时，把该表的数据按照某种规则，例如ID、时间、地域等等，进行划分，然后存储到多个结构相同的表，和不同的库上。

#### 3.3.1.3 常见架构

Cobar、MyCat等等，进行数据操作时依赖根据业务制定的分表路由和分库路由

#### 3.3.1.4 常见问题

1. 事务问题

在执行分库分表之后，由于数据存储到了不同的库上，数据库事务管理出现了困难。**解决方案**：按业务需求使用分布式事务

2. 跨库跨表的join问题

表的关联操作将受到限制，无法join位于不同分库的表。

**解决方案**：部分公用的数据可以适当冗余。理论上少用join一定程度上可以避免慢查

3.分页

**解决方案**：既然有钱搞分库分表了，不如再多花点钱搞个搜索服务器诸如：ElasticSearch、Lucene等等，加快查询效率。

## 3.4 Sql优化

### 3.4.1 MySQL

#### 3.4.1.1 常见类型

**MyISAM**

MyISAM表无法处理事务，这就意味着有事务处理需求的表，不能使用MyISAM存储引擎。MyISAM存储引擎特别适合在以下几种情况下使用：

1.选择密集型的表。MyISAM存储引擎在筛选大量数据时非常迅速，这是它最突出的优点。

2.插入密集型的表。MyISAM的并发插入特性允许同时选择和插入数据。例如：MyISAM存储引擎很适合管理邮件或Web服务器日志数据。

**InnoDB**

InnoDB是一个健壮的事务型存储引擎，这种存储引擎已经被很多互联网公司使用，为用户操作非常大的数据存储提供了一个强大的解决方案。我的电脑上安装的MySQL 5.6.13版，InnoDB就是作为默认的存储引擎。InnoDB还引入了行级锁定和外键约束，在以下场合下，使用InnoDB是最理想的选择：

1.更新密集的表。InnoDB存储引擎特别适合处理多重并发的更新请求。

2.事务。InnoDB存储引擎是支持事务的标准MySQL存储引擎。

3.自动灾难恢复。与其它存储引擎不同，InnoDB表能够自动从灾难中恢复。

4.外键约束。MySQL支持外键的存储引擎只有InnoDB。

5.支持自动增加列AUTO\_INCREMENT属性。

一般来说，如果需要事务支持，并且有较高的并发读取频率，InnoDB是不错的选择。

**MEMORY**

使用MySQL Memory存储引擎的出发点是速度。为得到最快的响应时间，采用的逻辑存储介质是系统内存。虽然在内存中存储表数据确实会提供很高的性能，但当mysqld守护进程崩溃时，所有的Memory数据都会丢失。获得速度的同时也带来了一些缺陷。它要求存储在Memory数据表里的数据使用的是长度不变的格式，这意味着不能使用BLOB和TEXT这样的长度可变的数据类型，VARCHAR是一种长度可变的类型，但因为它在MySQL内部当做长度固定不变的CHAR类型，所以可以使用。

一般在以下几种情况下使用Memory存储引擎：

1.目标数据较小，而且被非常频繁地访问。在内存中存放数据，所以会造成内存的使用，可以通过参数max\_heap\_table\_size控制Memory表的大小，设置此参数，就可以限制Memory表的最大大小。

2.如果数据是临时的，而且要求必须立即可用，那么就可以存放在内存表中。

3.存储在Memory表中的数据如果突然丢失，不会对应用服务产生实质的负面影响。

**MERGE**

MERGE存储引擎是一组MyISAM表的组合，这些MyISAM表结构必须完全相同，尽管其使用不如其它引擎突出，但是在某些情况下非常有用。说白了，Merge表就是几个相同MyISAM表的聚合器；Merge表中并没有数据，对Merge类型的表可以进行查询、更新、删除操作，这些操作实际上是对内部的MyISAM表进行操作。Merge存储引擎的使用场景。

对于服务器日志这种信息，一般常用的存储策略是将数据分成很多表，每个名称与特定的时间端相关。例如：可以用12个相同的表来存储服务器日志数据，每个表用对应各个月份的名字来命名。当有必要基于所有12个日志表的数据来生成报表，这意味着需要编写并更新多表查询，以反映这些表中的信息。与其编写这些可能出现错误的查询，不如将这些表合并起来使用一条查询，之后再删除Merge表，而不影响原来的数据，删除Merge表只是删除Merge表的定义，对内部的表没有任何影响。

**ARCHIVE**

Archive是归档的意思，在归档之后很多的高级功能就不再支持了，仅仅支持最基本的插入和查询两种功能。在MySQL 5.5版以前，Archive是不支持索引，但是在MySQL 5.5以后的版本中就开始支持索引了。Archive拥有很好的压缩机制，它使用zlib压缩库，在记录被请求时会实时压缩，所以它经常被用来当做仓库使用。

#### 3.4.1.2 优化方式

1. EXPLAIN 你的 SELECT 查询;

2. 当只要一行数据时使用 LIMIT 1;

3. 为搜索字段建索引;

## 3.5 悲观锁和乐观锁

悲观锁(Pessimistic Lock), 每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁(Optimistic Lock),每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

1. **线程**

## 4.1 线程与进程

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

## 4.2 线程池

### 4.2.1 内部原理

**使用部分**：

http://www.cnblogs.com/easycloud/p/3726089.html

**缺陷**：

http://www.ibm.com/developerworks/cn/java/j-jtp0730/

**好处**：

1.减少在创建和销毁线程上所花的时间以及系统资源的开销

2.如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及”过度切换”。

**总结**:

一个任务通过execute(Runnable)方法被添加到线程池，任务必须是一个 Runnable类型的对象，任务的执行方法就是调用Runnable类型对象的run()方法。当一个任务通过execute(Runnable)方法欲添加到线程池时，会做一下几步：

1. 如果此时线程池中的数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。

2. 如果此时线程池中的数量大于等于corePoolSize，但是缓冲队列 workQueue未满，那么任务被放入缓冲队列。

3. 如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理添加的任务。

4. 如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过 handler所指定的策略来处理此任务。也就是处理任务的优先级为：核心线程corePoolSize、任务队列workQueue、最大线程maximumPoolSize，如果三者都满了，使用handler处理被拒绝的任务。

5. 当线程池中的线程数量大于corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。

### 4.2.2 常见类型

1、newFixedThreadPool创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

2、newCachedThreadPool创建一个可缓存的线程池。这种类型的线程池特点是：

1).工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

2).如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

3、newSingleThreadExecutor创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，如果这个线程异常结束，会有另一个取代它，保证顺序执行(我觉得这点是它的特色)。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的 。

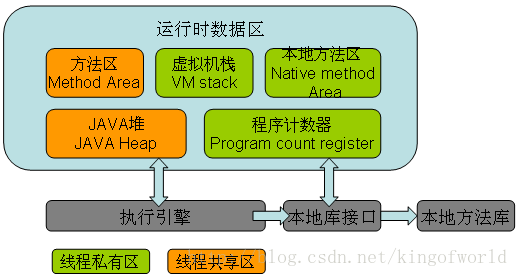
4、newScheduleThreadPool创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer。

## 4.3 线程安全

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。 线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

1. **JVM**

## 5.1 JVM 内存模型

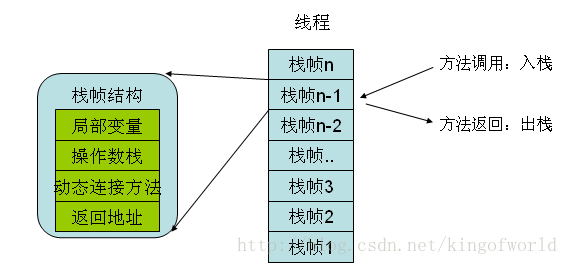


### 5.2.1 程序计数器

多线程时，当线程数超过CPU数量或CPU内核数量，线程之间就要根据时间片轮询抢夺CPU时间资源。因此每个线程有要有一个独立的程序计数器，记录下一条要运行的指令。线程私有的内存区域。如果执行的是JAVA方法，计数器记录正在执行的java字节码地址，如果执行的是native方法，则计数器为空。

### 5.2.2 虚拟机栈

线程私有的，与线程在同一时间创建。管理JAVA方法执行的内存模型。每个方法执行时都会创建一个桢栈来存储方法的的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法，-Xss参数可以设置虚拟机栈大小）。栈的大小可以是固定的，或者是动态扩展的。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：



### 5.2.3 本地方法区

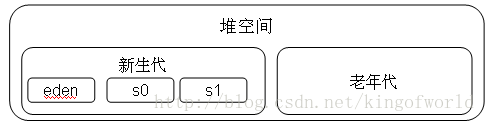
和虚拟机栈功能相似，但管理的不是JAVA方法，是本地方法，本地方法是用C实现的。

### 5.2.4 JAVA堆

线程共享的，存放所有对象实例和数组。垃圾回收的主要区域。可以分为新生代和老年代(tenured)。

新生代用于存放刚创建的对象以及年轻的对象，如果对象一直没有被回收，生存得足够长，老年对象就会被移入老年代。

新生代又可进一步细分为eden、survivorSpace0(s0,from space)、survivorSpace1(s1,to space)。刚创建的对象都放入eden,s0和s1都至少经过一次GC并幸存。如果幸存对象经过一定时间仍存在，则进入老年代(tenured)。



### 5.2.5 方法区

线程共享的，用于存放被虚拟机加载的类的元数据信息：如常量、静态变量、即时编译器编译后的代码。也成为永久代。如果hotspot虚拟机确定一个类的定义信息不会被使用，也会将其回收。回收的基本条件至少有：所有该类的实例被回收，而且装载该类的ClassLoader被回收

**参考资料**：<http://www.cnblogs.com/AloneSword/p/4262255.html>

## 5.2 OOM原因

### 5.2.1 Heap

**java.lang.OutOfMemoryError: Java heap space**

**分析:**

此OOM是由于JVM中heap的最大值不满足需要，将设置heap的最大值调高即可，参数样例为：-Xmx2G

JVM堆的设置是指Java程序运行过程中JVM可以调配使用的内存空间的设置.JVM在启动的时候会自动设置Heap size的值，其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。Heap size 的大小是Young Generation 和Tenured Generaion 之和。

提示：在JVM中如果98％的时间是用于GC且可用的Heap size 不足2％的时候将抛出此异常信息。

提示：Heap Size 最大不要超过可用物理内存的80％，一般的要将-Xms和-Xmx选项设置为相同，而-Xmn为1/4的-Xmx值。

**解决参考:**

调高heap的最大值，即-Xmx的值调大。

### 5.2.2 StackOverflowError

**Exception in thread "main" java.lang.StackOverflowError**

**分析:**

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常。

一般在单线程程序情况下无法产生OutOfMemoryError异常，使用多线程方式也会出现OutOfMemeoryError，因为栈是线程私有的，线程多也会方法区溢出。

**解决参考:**

检查程序是否有深度递归。

### 5.2.3 Perm

**java.lang.OutOfMemoryError: PermGen space**

**分析**:

PermGen space的全称是Permanent Generation space,是指内存的永久保存区域,这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到PermGen space中,它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的应用中有很多CLASS的话,就很可能出现PermGen space错误,这种错误常见在web服务器对JSP进行pre compile的时候。如果你的WEB APP下都用了大量的第三方jar, 其大小超过了jvm默认的大小那么就会产生此错误信息了。由于JVM在默认的情况下，Perm默认为64M，而很多程序需要大量的Perm区内 存，尤其使用到像spring等框架的时候，由于需要使用到动态生成类，而这些类不能被GC自动释放，所以导致OutOfMemoryError: PermGen space异常。解决方法很简单，增大JVM的 -XX:MaxPermSize 启动参数，就可以解决这个问题，如过使用的是默认变量通常是64M，改成128M就可以了，-XX:MaxPermSize=128m。如果已经是128m，就改成 256m。我一般在服务器上为安全起见，改成256m。

**解决参考:**

调高Perm的最大值，即-XX:MaxPermSize的值调大。

另外，注意一点，Perm一般是在JVM启动时加载类进来，如果是JVM运行较长一段时间而不是刚启动后溢出的话，很有可能是由于运行时有类被动态加载进来，此时建议用CMS策略中的类卸载配置。

### 5.2.4 GC

**java.lang.OutOfMemoryError: GC overhead limit exceeded**

**分析**:

此OOM是由于JVM在GC时，对象过多，导致内存溢出，建议调整GC的策略，在一定比例下开始GC而不要使用默认的策略，或者将新代和老代设置合适的大小，需要进行微调存活率。

**解决参考**:

改变GC策略，在老代80%时就是开始GC，并且将-XX:SurvivorRatio（-XX:SurvivorRatio=8）和-XX:NewRatio（-XX:NewRatio=4）设置的更合理。

### 5.2.5 Native Thread Created

**java.lang.OutOfMemoryError: unable to create new native thread**

**分析**:

这个异常问题本质原因是我们创建了太多的线程，而能创建的线程数是有限制的，导致了异常的发生。

**解决参考:**

如果JVM内存调的过大或者可利用率小于20%，可以建议将heap及perm的最大值下调，并将线程栈调小，即-Xss调小，如：-Xss128k。

在JVM内存不能调小的前提下，将-Xss设置较小，如：-Xss:128k。

### 5.2.6 Allocate Huge Array

**Exception in thread "main": java.lang.OutOfMemoryError: Requested array size exceeds VM limit**

**分析**:

此类信息表明应用程序试图分配一个大于堆大小的数组。例如，如果应用程序new一个数组对象，大小为512M，但是最大堆大小为256M，因此OutOfMemoryError会抛出，因为数组的大小超过虚拟机的限制。

**解决参考:**

（1）、首先检查heap的-Xmx是不是设置的过小

（2）、如果heap的-Xmx已经足够大，那么请检查应用程序是不是存在bug，例如：应用程序可能在计算数组的大小时，存在算法错误，导致数组的size很大，从而导致巨大的数组被分配。

### 5.2.7 Small Swap

**Exception in thread "main": java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?**

**分析**:

抛出这类错误，是由于从native堆中分配内存失败，并且堆内存可能接近耗尽。这类错误可能跟应用程序没有关系，例如下面两种原因也会导致错误的发生：

（1）操作系统配置了较小的交换区

（2）系统的另外一个进程正在消耗所有的内存

**解决参考:**

（1）、检查os的swap是不是没有设置或者设置的过小

（2）、检查是否有其他进程在消耗大量的内存，从而导致当前的JVM内存不够分配。

注意：虽然有时<reason>部分显示导致OOM的原因，但大多数情况下，<reason>显示的是提示分配失败的源模块的名称，所以有必要查看日志文件，如crash时的hs文件。

### 5.2.8 Exhausted Native Memory

**java.lang.OutOfMemoryErr java.io.FileInputStream.readBytes(Native Method)**

**分析**:

从错误日志来看，在OOM后面没有提示引起OOM的原因，进一步查看stack trace发现，导致OOM的原因是由Native Method的调用引起的，另外检查[Java](http://lib.csdn.net/base/java)heap，发现heap的使用正常，因而需要考虑问题的发生是由于Native memory被耗尽导致的。

**解决参考:**

从根本上来说，解决此问题的方法应该通过检测发生问题时的环境下，native memory为什么被占用或者说为什么native memory越来越小，从而去解决引起Native memory减小的问题。但是如果此问题不容易分析时，可以通过以下方法或者结合起来处理。  
（1）、cpu和os保证是64位的，并且jdk也换为64位的。  
（2）、将java heap的-Xmx尽量调小，但是保证在不影响应用使用的前提下。  
（3）、限制对native memory的消耗，比如：将thread的-Xss调小，并且限制产生大量的线程；限制文件的io操作次数和数量；限制网络的使用等等。

## 5.3 GC算法

### 5.3.1 标记-清除算法(Mark-Sweep)

从根节点开始标记所有可达对象，其余没标记的即为垃圾对象，执行清除。但回收后的空间是不连续的。

### 5.3.2 复制算法(copying)

将内存分成两块，每次只使用其中一块，垃圾回收时，将标记的对象拷贝到另外一块中，然后完全清除原来使用的那块内存。复制后的空间是连续的。复制算法适用于新生代，因为垃圾对象多于存活对象，复制算法更高效。在新生代串行垃圾回收算法中，将eden中标记存活的对象拷贝未使用的s1中，s0中的年轻对象也进入s1，如果s1空间已满，则进入老年代；这样交替使用s0和s1。这种改进的复制算法，既保证了空间的连续性，有避免了大量的内存空间浪费。

### 5.3.3 标记-压缩算法(Mark-compact)

适合用于老年代的算法（存活对象多于垃圾对象）。

标记后不复制，而是将存活对象压缩到内存的一端，然后清理边界外的所有对象。

## 5.4 优化参数

-XX:+PrintGCDetails 打印垃圾回收信息

-Xms 为Heap区域的初始值，线上环境需要与-Xmx设置为一致，否则capacity的值会来回飘动

-Xmx 为Heap区域的最大值

-Xss（或-ss） 线程栈大小（指一个线程的native空间）1.5以后是1M的默认大小

-XX:PermSize与-XX:MaxPermSize 方法区（永久代）的初始大小和最大值（但不是本地方法区）

-XX:NewRatio 老年代与新生代比率

-XX:SurvivorRatio Eden与Survivor的占用比例。例如8表示，一个survivor区占用 1/8 的Eden内存，即1/10的新生代内存，为什么不是1/9？因为我们的新生代有2个survivor，即S1和S22。所以survivor总共是占用新生代内存的 2/10，Eden与新生代的占比则为 8/10。

-XX:MaxHeapFreeRatio GC后，如果发现空闲堆内存占到整个预估的比例小于这个值，则减小堆空间。

-XX:MinHeapFreeRatio GC后，如果发现空闲堆内存占到整个预估的比例大于这个值，则增大堆空间。

-XX:NewSize 新生代大小

1. **分布式**

## 6.1 分布式主键生成策略

1. Redis生成ID

2. Twitter的snowflake算法

3. zookeeper生成唯一ID

4. MongoDB的ObjectId

**参考资料**：<http://www.cnblogs.com/haoxinyue/p/5208136.html>

## 6.2 分布式锁

### 6.2.1 Redis

基本的使用方式是首先setnx(key)，如果返回结果为1则表示抢锁成功，执行expire(30)命令，设置锁的过期时间为30秒，如果抢锁失败，可以做一个锁的重入（再一次去抢锁）进行优化。

**参考资料**：http://doc.redisfans.com/

### 6.2.2 ZooKeeper

过段时间再研究了，自行百度~~

## 6.3 分布式事务

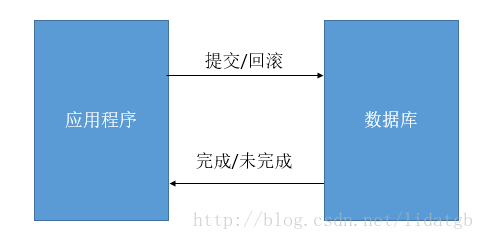
### 6.3.1 一阶段提交

然后在应用程序发出提交/回滚请求后，数据库执行操作，而后将成功/失败返回给应用程序，程序继续执行。

一阶段提交协议相对简单，简单带来的优点就是，它不用再与其他的对象交互，节省了判断步骤和时间。

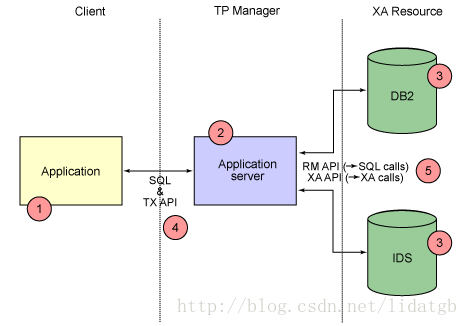
缺点是：

1. 数据库确认执行事务的时间较长，出问题的可能性就随之增大。
2. 如果有多个数据源，一阶段提交协议无法协调他们之间的关系。



### 6.3.2 二阶段提交

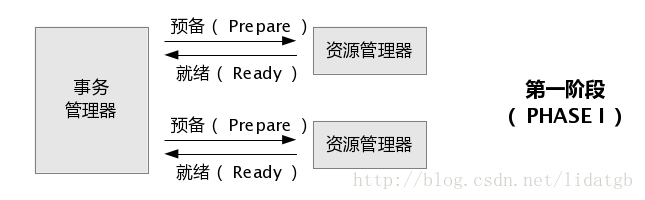
二阶段协议通过将两层变为三层，增加了中间的管理者角色，从而协调多个数据源之间的关系。



第一阶段：提交事务请求

1．事务管理器通知参与该事务的各个资源管理器，通知他们开始准备事务。

2、执行事务。各参与者节点执行事务奥做，并将Undo和Redo信息记入事务日志中。

3、各参与者向协调者反馈事务询问的响应。

第二阶段：执行事务提交

假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务提交。

1、发送提交请求。

2、事务提交。

3、反馈事务提交结果。参与者在完成事务提交之后，会向协调者发送Ack消息。

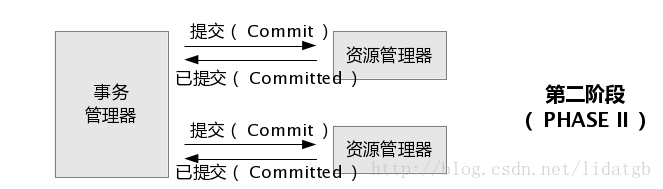
4、完成事务。

中断事务：

1、发送回滚请求。协调者向参与者发出rollback请求。

2、事务回滚。参与者接收到Roolback请求利用阶段一种记录的Undo信息来执行事务回滚动作。

3、反馈事务回滚结果。

4、中断事务。

### 6.3.3 三阶段提交

阶段一：CanCommit

1、事务询问。

2、各参与者向协调这反馈事务询问的响应。

阶段二：PreCommit

假设协调者从所有的参与者获得的都是Yes响应，那么将执行事务预提交。

1、发送预提交请求。协调者向所有参与者节点发出preCommit请求，进入prepared阶段。

2、事务预提交。参与者接收到preCommt请求，执行事务擦偶走，将Undo和Redo信息记录到事务日志中。

3、各参与者向协调者反馈事务提交的响应。

假设任何一个参与者向协调者反馈了No反应，活着在等待超时之后，协调者无法获得所有参与者的响应，那么将执行事务的中断。

1、发送终端请求。协调者向所有参与者发出abort请求。

2、中断事务。无论接到abort请求还是等待协调者请求过程出现超时情况，参与者都会中断事务。

阶段三：doCommit

该阶段将进行真正的事务提交：

执行提交

1、发送提交请求。进入这一阶段，假设协调者从正常的工作状态，并且接收到所有的参与者的ack响应，它将从预提交状态转换到提交状态，向所有参与者发送doCommit请求。

2、事务提交。参与者接收到doCommit请求后，正式执行事务提交操作。并在提交后释放在整个事务执行期间占用的事务资源。

3、反馈事务提交结果。参与者完成事务提交之后，向协调者发送Ack消息。

4、完成事务。协调者接收到所有参与者的Ack消息，完成事务。

中断事务

中断事务的4步操作与提交事务完全一致，只不过从提交事务变成了事务回滚。

最大优点就是降低了参与者的阻塞范围，并且能够在出现单点故障后继续达成一致。

缺点就是在去除阻塞的情况下引入了新的问题，那就是参与者接收到了PreCommit消息，然后网络出现问题，参与者和协调者无法通信，这种情况下，参与者依然会执行事务的提交。

**参考资料**：倪超的《从Paxos到zookeeper》

<http://blog.csdn.net/lidatgb/article/details/38468073>

<http://www.cnblogs.com/congsg2016/p/5400958.html>

1. **RPC**

## 7.1 序列化方式

**常见方式**：Json（FastJson、Jackson）、Xml(基本很少用了)、ProtoStuff（RPC中用的较多）

**参考资料**：<https://github.com/eishay/jvm-serializers/wiki>

## 7.2 Dubbo（了解）

**参考资料：**[**https://github.com/dangdangdotcom/dubbox**](https://github.com/dangdangdotcom/dubbox)

## 7.3 Netty（了解）

**参考资料：**[**https://github.com/netty/netty**](https://github.com/netty/netty)

1. **大数据**

## Hadoop（了解）

过段时间再研究了，自行百度~~

## Spark（了解）

过段时间再研究了，自行百度~~

1. **语言**

## 9.1 Python（了解）

**参考资料：**<http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000>

## 9.2 Node.js（了解）

参考资料：<http://nodejs.cn/api/>

1. **总结**

千里之行，始于足下。且行且努力~~