# J2SE

## 1.1 KEY Class

### String

final

主要是为了“效率” 和 “安全性” 的缘故

String str = new String("xyz");创建了几个字符串对象(2)？   
答：两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象。

### Object

equals



“==”比较两个对象的的内存地址，可见equals相同的两个对象一定相等

重写equals为何要重写hashCode

hashCode的存在主要是用于查找的快捷性，hashCode是用来在散列存储结构中确定对象的存储地址的。

官方文档的定义，可以得到equals与hashCode间的关系：

1、两个对象相同，hashCode一定要相同。

2、两个对象的hashCode相同，它们并不一定相同。

notify ()/notifyAll ()/wait ()

必须在synchronized修饰的方法或代码块中，wait会释放占有的锁,notify和notifyAll不会释放占用的锁.

## 1.2 Collection

### List

排序

### HashSet

https://github.com/pzxwhc/MineKnowContainer/issues/21

### HashMap

1）结构

[Reference](http://blog.csdn.net/dongnan591172113/article/details/8770923)

HashMap底层就是一个数组结构，数组中的每一项又是一个链表。



2）读取



3）扩容机制:

当size>capacity \* loadFactor（loadFactor=0.75），用一个新的数组代替已有的容量小的数组 newCapacity=2 \* table.length

4）碰撞冲突

对于位置链表的next

### ConcurrentHashMap

同步的数据结构，例如concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高

**实现原理**

HashTable: 每个方法加synchronized

ConcurrentHashMap: 引入了一个“分段锁”的概念,就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中：

**LinkedHashMap**

## 1.3 Thread

### 1.3.1 BASE

线程是操作系统能够进行运算调度的最小单位。

#### 线程状态

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51967711)

New->Runable(start)->Running->Blocked

Blocked VS Wait:

* 让出CUP方式:block等待monitor lock, wait主动让出
* 线程唤醒的对象不同，blocked由jvm唤醒，还是另一个线程。

从Linux内核来看，这些线程都是等待状态，没本质区别。jvm出于管理的需要，做了这种区分。

#### Interrupt

每个线程都有一个打断标志。

* 线程在 sleep/wait/join 时，别的进程调用此线程的interrupt() 方法，此线程会被唤醒并被要求处理InterruptedException；
* 线程在运行时，此线程的 “打断标志” 会被设置。

所以说，interrupt() 方法能够中断线程的等待过程,不会中断一个正在运行的线程 。

#### Sleep

TimeUnit.SECONDS

#### 多线程

线程上下文切换

保存线程A的执行现场->载入线程B的执行现场

引起切换的原因

* 时间片用完，CPU正常调度下一个任务
* 被其他优先级更高的任务抢占
* 执行任务碰到IO阻塞，调度器挂起当前任务，切换执行下一个任务
* 用户代码主动挂起当前任务让出CPU时间
* 多任务抢占资源，由于没有抢到被挂起
* 硬件中断

****多线程 VS 单线程****

**多线程不一定就比单线程程序跑的快，**多线程的代价

* 设计更复杂
* 线程创建和消耗所花时间和系统资源开销：大量线程造成资源耗尽
* 线程调度
* 内存同步
* 上下文切换 ： 有可能出现“过度切换”的问题

线程资源必须通过线程池提供

减少在创建和销毁线程上花的时间以及系统资源的开销。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题

#### LOCK

乐观锁 VS 悲观锁

* 乐观锁:假定不会出现并发问题，只有更新数据时才检查数据是否被锁定
* 悲观锁:假定会出现并发问题，读取数据时即将数据锁定

可重入锁

Code: ReentrantTest100->102

可重入锁的最大作用就是 可以避免死锁。



加锁优先级

无锁数据结构>锁区块>方法>对象锁>类锁

说明：尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用RPC方法

死锁

对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会造成死锁。

说明：线程一需要对表A、B、C依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序也必须是A、B、C，否则可能出现死锁。

**锁优化**  
高效并发是从JDK 1.5到JDK 1.6的一个重要改进，HotSpot虚拟机开发团队在这个版本上  
花费了大量的精力去实现各种锁优化技术，如适应性自旋（Adaptive  Spinning）、锁消除  
（Lock Elimination）、锁粗化（Lock Coarsening）、轻量级锁（Lightweight Locking）和偏向  
锁（Biased Locking）等，这些技术都是为了在线程之间更高效地共享数据，以及解决竞争问  
题，从而提高程序的执行效率。

### 1.3.2 线程安全

[Reference](http://www.jasongj.com/java/thread_safe/)

提到线程安全,我们需要同时关注原子性、可见性和有序性和问题。

#### ****原子性****

[Reference](http://blog.psjay.com/posts/summary-of-java-concurrency-two-synchronized-and-atomicity/)

原子操作：不可中断的一个或一系列操作。

An atomic operation is one that **cannot be interrupted** by the scheduler; if the operation begins, then it will run to **completion** before the possibility of a context switch.

这里的原子性指java规范中定义的原子操作，指原始类型或引用类型的读取/赋值。

synchronized/Lock

要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。这里的原子性是互斥。

可以中断，但即使中断了，组中的其他方法也不能被执行，保证是串行的顺序。

原子性操作



只有语句1是原子性操作，其他三个语句都不是原子性操作

语句1是直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中。

x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。

****非原子操作：****

**只要是非原子操作，操作就可能被中断或阻塞，需要同步来保证线程安全。**

#### ****可见性****

在一些情况下即便是原子操作也可能会引发一些错误，Java提供了volatile关键字来保证可见性。

volatile ([Reference](http://www.cnblogs.com/dolphin0520/p/3920373.html))

使用恰当的话，它比synchronized的使用和执行成本会更低，因为它不会引起线程上下文的切换和调度。

使用场景

在一些情况下即便是原子操作也可能会引发一些错误



线程访问对象的值时，将堆内存中的值copy到工作内存。Java HotSpot Server版本编译器并不会将修改的值写回内存。

当使用volatile修饰某个变量时，它会保证对该变量的修改会立即被更新到内存中，并且将其它缓存中对该变量的缓存设置成无效，因此其它线程需要读取该值时必须从主内存中读取，从而得到最新的值。

#### ****有序性****

Java中可通过volatile在一定程序上保证顺序性，另外还可以通过synchronized和锁来保证顺序性。

### 1.3.3 Thread API

#### ThreadGroup

[Reference](http://www.cnblogs.com/0616--ataozhijia/p/3710571.html)

Code: ThreadGroup100

使用线程组的好处是可以对这一组的线程进行整体操作。

Thread.currentThread ().getThreadGroup ()

#### Thread Pool

四种线程池

* **newCachedThreadPool** 根据处理需要，可灵活回和新建线程。

maximumPoolSize = Integer.MAX\_VALUE(Integer是4位，-2^31~2^31-1)

* **newFixedThreadPool** 定长线程池，超出的线程会在队列中等待。
* **newScheduledThreadPool** 定长线程池，支持定时及周期性任务执行。
* **newSingleThreadExecutor** 单线程化的线程池，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

ExecutorService

Code: ThreadPool101

在Java5之后，并发线程这块发生了根本的变化，最重要的莫过于新的启动(Executor启动)、调度、管理(shutdown)线程的一大堆API了。

**submit VS execute**

Code: ThreadPoolFutureGetException101

submit有返回值，而execute没有,submit方便Exception处理。

void execute (Runnable command);

<T> Future<T> submit (Callable<T> task);

Future



* isDone()： 执行状态
* get(): 获取执行结果。 Waits if necessary for the computation to complete, and then retrieves its result.

ThreadPoolExecutor

### 1.3.4 线程同步

每一个线程自顾自的做自己的工作固然好。但是线程之间经常会相互影响（竞争或者合作），比如多个线程需要同时操作一个资源（比如一个对象）。这个时候，如果不进行同步，就可能会引发难以预料的错误

实现同步方法

* ThreadLocal
* sychronized
* Lock
* Atomic

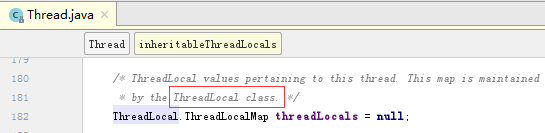
#### ThreadLocal

ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本, 为解决多线程程序的并发问题提供了一种新的思路

[Reference](http://www.cnblogs.com/dolphin0520/p/3920407.html)

实现原理

1.ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的；



ThreadLocalMap的Key值为ThreadLocal对象. 通过创建不同的ThreadLocal对象来存储多个变量；

**set方法**



**get方法的实现：**



应用场景

最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。

如：



下面这段代码摘自：

<http://www.iteye.com/topic/103804>



Note:

ThreadLocal无法解决共享对象的更新问题，ThreadLocal对象建议使用static修饰。这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享此静态变量 ，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只要是这个线程内定义的)都可以操控这个变量

#### ReentrantLock

Code: ReentrantLock110

lock () /unlock / tryLock (time)

使用场景

场景1：tryLock()

用在定时任务时,忽略重复触发

操作界面重复点击

场景2：tryLock(time)

场景3：lock(),类似synchronized

Lock有更灵活的锁定方式，公平锁与不公平锁，而synchronized永远是公平的。

这种情况主要用于对资源的争抢（如：有状态的操作，文件操作，同步消息发送等）

场景 4：可中断等待休眠线程(lockInterruptibly)

线程的interrupt() 方法能够中断线程的等待过程。

公平锁

Synchronized锁非公平锁，ReentrantLock可以创建公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，但公平锁表现的性能不是很好。

ReentrantLock VS synchronized

* 两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，JVM层面上实现的,是原生语法层面的互斥。而ReentrantLock是API层面的互斥锁，需要写的控制锁的代码。
* 使用 synchronized,不能中断,lock在等待锁的过程可以被中断。
* 出现异常,synchronized 会自动释放锁，lock 则不会自动释放锁。
* 公平锁，ReentrantLock可以创建公平锁。
* 锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象
* 性能
* synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。

而Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（Compare and Swap）。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。

#### ReadWriteLock

Compare with synchronized

 synchronized的修饰区域每次只有一个线程可以访问。这种强烈的互斥性使得每次不管是读数据还是写数据都只能有一个线程可以操作。**在有多个读线程可以并行执行**的情况下，它并不是一个理想的选择。

Compare with volatile

使用读写锁的方式在单个写线程加多个读线程的情况下，其实差别不大。在使用volatile变量的时候，因为每次操作修改的结果对于全局都是可见的。那么在只有一个写线程的情况下，只有这个线程可以唯一修改数据。不会存在有几个写线程而产生的竞争条件。对于有**多个写线程的情况下，volatile变量就不能保证数据的一致性了**。而ReentrantReadWriteLock却可以有锁的机制保证互斥。它同时也尽可能保证了足够大的并行性。

#### sychronized

synchronized加锁，实现线程的互斥操作。

[Reference](http://blog.csdn.net/javazejian/article/details/72828483)

原理:保证接口的原子性和可见性

每个对象都存着一个 monitor对象，所以Java中任意对象可以作为锁，数据结构如下



* \_owner指向持有ObjectMonitor对象的线程
* \_EntryList 等待锁block状态的线程
* \_WaitSet 调用wait()

修饰实例方法

锁==synchronized(this)



#### Atomic

Atom包下的原子操作类实现了一种乐观锁，其本质是使用了CPU级别的CAS指令。

CAS指令



一种赋值操作，CPU的CAS指令是原子性操作，执行完成前不可中断。本质上执行过程加了排他锁，但是线程加锁过程并未有线程block和上下文切换，其开销比需要操作系统参与的锁的开销小。

乐观锁

借助CAS（compare and swap），实现乐观锁，在冲突不激烈的多线程环境，会有比较好的性能。

实现原理



ABA问题

潜在问题:如果需要关注A值变化过程，是会漏掉一段时间窗口的监控 .

#### BlockingQueue

定时线程的使用

## 1.6 JVM

优化

JVM的编译优化

指令重排序，内存栅栏等

### 内存模型

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51832737)



**JAVA OPS**

**Heap**

* -Xms heap的最小内存；
* -Xmx heap的最大内存；
* -Xmn young generation的大小，一般设置为Xmx的3、4分之一
* NewRatio: young:old,2表示1:2,即yound/heap=1/5
* -XX:NewSize: young的大小
* –XX:MaxNewSize
* -server 启用jdk 的 server 版；一定要作为第一个参数，在多个CPU时性能佳

**No Heap**

* -XX:PermSize 内存永久保留区域
* -XX:MaxPermSize 内存最大永久保留区域
* -XX:MaxNewSize:
* -XX:SurvivorRatio:2\*Survivor:eden,8代表1:1:8
* -XX:+UseParNewGC ：缩短minor收集的时间
* -XX:+UseConcMarkSweepGC ：缩短major收集的时间。此选项在Heap Size 比较大而且Major收集时间较长的情况下使用更合适

**32G 内存配置示例：**

-Xms10g -Xmx10g -Xmn1024m -XX:PermSize=1g -XX:MaxPermSize=2g -Xshare:off

Heap

**Java堆中各代分布：**



Young：主要是用来存放新生的对象。

Old：主要存放应用程序中生命周期长的内存对象

Stack

线程工作内存(高速缓存)

方法区

又叫永久区(Permanent)存放类信息、常量、静态变量、编译器编译后的代码等数据；参数设置示例：-XX:PermSize=5M -XX:MaxPermSize=7M)

### 类加载机制

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51838938)  
**Object obj = new Object ();**

new并非是一个原子操作，Class的装载分了三个阶段，loading，linking和initializing。

**Loading**: 加载Class二进制字节流，方法区初始化静态变量, 在堆中创建一个 Class 类对象。

**Linking**

验证--准备--解析

类变量（static）根据数据类型分配内存(方法区)并初始化成默认值(如 0、0L、null、false 等)。

**Initializing**:初始化类变量，调用构造函数初始化成员变量。

Linking的子阶段解析可能在Initializing之前，也可能在之后，

解析(Linking的子阶段): 将instance对象指向分配的内存空间

java虚拟机堆何时分配，存放哪些数据

深入分析了Classloader，双亲委派机制

**2. jvm性能调优**

* 控制GC的行为. 频繁GC导致性能下降,因此经常会根据系统运行的程序的特性来更改GC行为
* 调整JVM堆栈大小.xmx xxx
* 控制JVM线程的内存分配.xss

4. 介绍GC 和GC Root不正常引用。

6. jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析

7. 数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold ，问通常多大，没做过一问便知）

 老年代中数组的访问方式

谁会被 GC ，什么时候 GC,  如果想不被 GC 怎么办

9. GC 算法，永久代对象如何 GC ， GC 有环怎么处理

11. 12. 如果想在 GC 中生存 1 次怎么办

### GC

[Reference](http://www.cnblogs.com/ityouknow/p/5614961.html)

**垃圾回收算法**

Java 中的堆也是 GC 收集垃圾的主要区域。GC 分为两种：Minor GC、Full GC

* 标记-清除”（Mark-Sweep）:1.标记和清除过程的效率都不高 2.空间碎片太多
* 复制”（Copying）的收集算法:这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低

Minor GC

Minor GC 是发生在新生代中的垃圾收集动作，所采用的是复制算法。每次垃圾收集时都发现有大批对象死去，只有少量存活,只需要付出少量存活对象的复制成本。

当对象在 Eden ( 包括一个 Survivor 区域，这里假设是 from 区域 ) 出生后，在经过一次

Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(上面已经假设为 from 区域，这里应为 to 区域，即 to 区域有足够的内存空间来存储 Eden 和 from 区域中存活的对象 )，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 ( 即 to 区域 ) 中，然后清理所使用过的 Eden 以及 Survivor 区域 ( 即 from 区域 )，并且将这些对象的年龄设置为1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 ( 默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定 )，这些对象就会成为老年代。

但这也不是一定的，对于一些较大的对象 ( 即需要分配一块较大的连续内存空间 ) 则是直接进入到老年代

Full GC

JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的。

GC 和 Full GC 有什么区别？

GC（或Minor GC）：收集 生命周期短的区域(Young area)。

Full GC （或Major GC）：收集生命周期短的区域(Young area)和生命周期比较长的区域(Old area)对整个堆进行垃圾收集。

垃圾收集器

* Serial收集器:串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会Stop The World（服务暂停）

参数控制：-XX:+UseSerialGC 串行收集器

* ParNew收集器:ParNew收集器其实就是Serial收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩

参数控制：-XX:+UseParNewGC ParNew收集器

-XX:ParallelGCThreads 限制线程数量

### JAVA OPS优化

**JVM调试方法：**

1.Jconsole查看内存各部分变化

2.gc 用jstat

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

**jps:** jps pid

**top :** top –Hp pid

### 附件 Debug Tools

#### jstat

jstat -option pic < interval >

**gcutil**



S0：年轻代中第一个survivor（幸存区）已使用的占当前容量百分比   
S1：年轻代中第二个survivor（幸存区）已使用的占当前容量百分比   
E：年轻代中Eden（伊甸园）已使用的占当前容量百分比   
O：old代已使用的占当前容量百分比

P：perm代已使用的占当前容量百分比   
YGC：从应用程序启动到采样时年轻代中gc次数   
YGCT：从应用程序启动到采样时年轻代中gc所用时间(s)

FGC：从应用程序启动到采样时old代(全gc)gc次数   
FGCT：从应用程序启动到采样时old代(全gc)gc所用时间(s)   
GCT：从应用程序启动到采样时gc用的总时间(s)   
**gc**



 S0C：年轻代中第一个survivor（幸存区）的容量 (byte)   
 S1C：年轻代中第二个survivor（幸存区）的容量 (byte)   
 S0U：年轻代中第一个survivor（幸存区）目前已使用空间 (byte)   
 S1U：年轻代中第二个survivor（幸存区）目前已使用空间 (byte)   
 EC：年轻代中Eden（伊甸园）的容量 (byte)   
 EU：年轻代中Eden（伊甸园）目前已使用空间 (byte)   
OC：Old代的容量 (byte)   
OU：Old代目前已使用空间 (byte)   
PC：Perm(持久代)的容量 (byte)   
PU：Perm(持久代)目前已使用空间 (字节)

**gcnew**

年轻代对象的信息



**gcold**

 old代对象的信息



**gcoldcapacity**

old代对象的信息及其占用量



OGCMN：old代中初始化(最小)的大小 (字节)   
OGCMX：old代的最大容量 (字节)   
OGC：old代当前新生成的容量 (字节)   
OC：Old代的容量 (byte)

**gcnewcapacity:**



 NGCMN：young初始化(最小)的大小 (字节)   
 NGCMX：年轻代(young)的最大容量 (字节)   
 NGC：年轻代(young)中当前的容量 (字节)

S0CMX：年轻代中第一个survivor（幸存区）的最大容量 (字节)   
S1CMX ：年轻代中第二个survivor（幸存区）的最大容量 (字节)

ECMX：年轻代中Eden（伊甸园）的最大容量 (字节)   
**gcpermcapacity**

 PGCMN：perm代中初始化(最小)的大小 (字节)   
  PGCMX：perm代的最大容量 (字节)     
  PGC：perm代当前新生成的容量 (字节)   
PC：Perm(持久代)的容量 (byte)

#### jstack

"Thread-186" daemon prio=10 tid=0x00007f82e0042000 nid=0x3b5a runnable [0x00007f83313d3000]

java.lang.Thread.State: RUNNABLE

at java.net.SocketInputStream.socketRead0(Native Method)

at java.net.SocketInputStream.read(SocketInputStream.java:152)

at java.net.SocketInputStream.read(SocketInputStream.java:122)

at java.net.SocketInputStream.read(SocketInputStream.java:108)

at love.moon.spring.controller.mock.BioServer$1.run(BioServer.java:46)

at java.lang.Thread.run(Thread.java:745)

**Thread status**

Wait on condition

#### jconsole

Total blocked

Total waited:

**state**

Wait on condition

## 1.7 面向对象

**什么是java序列化，如何实现java序列化?**

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题

Java7、Java8的新特性

Java 8的内存分代改进

### 反射

Class.forName和classloader的区别

Class.forName(xxx);的作用是要求JVM查找并加载指定的类。

### 异常



Checked异常

Java认为Checked异常都是可以被处理的异常，所以Java程序必须显示处理。

常见Checked Exception

ClassNotFoundException/IOException/NoSuchMetodException

RuntimeException

RuntimeException产生频繁，处理麻烦(如除数是0和数组下标越界等)，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。

当然也可以显示捕获。



常见Runtime Exception

NullPointerException/ArrayStoreExcetpion/IndexOutOfBoundsException

统一异常处理

大多数情况下我们选择异常出现后只进行记录日志和UI用户提示,

service层,dao层，等底层只负责将异常抛出，在web层设置统一异常处理机制，并将一些异常转化为用户所能理解的提示信息。 并对其他未捕获异常的处理,统一跳转到500友好的页面。

开放接口层要将异常处理成错误码和错误信息方式返回。

分层异常处理规约(ali)

在 DAO 层，产生的异常类型有很多，无法用细粒度的异常进行 catch，使用 catch(Exception e)方式，并 throw new DAOException(e)，不需要打印日志，因为日志在 Manager/Service 层一定需要捕获并打到日志文件中去，如果同台服务器再打日志，浪费性能和存储。在 Service 层出现异常时，必须记录出错日志到磁盘，尽可能带上参数信息，相当于保护案发现场。如果 Manager 层与 Service 同机部署，日志方式与 DAO层处理一致，如果是单独部署，则采用与 Service 一致的处理方式。 Web 层绝不应该继续往上抛异常，因为已经处于顶层，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面， 加上用户容易理解的错误提示信息。

自定义异常

定义时区分unchecked / checked 异常，避免直接抛出new RuntimeException()，更不允许抛出Exception或者Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException等。

规避RuntimeException

Java 类库中定义的一类RuntimeException可以通过预先检查进行规避，而不应该通过catch 来处理，比如：IndexOutOfBoundsException，NullPointerException等等。

说明：无法通过预检查的异常除外，如在解析一个外部传来的字符串形式数字时，通过catch NumberFormatException来实现。

正例：if (obj != null) {...}

反例：try { obj.method() } catch (NullPointerException e) {...}

Note:

异常不可做流程控制/条件控制，因为异常的处理效率比条件分支低。

避免对大段代码进行try-catch

### 开发模式

#### 敏捷开发

把产品开发引向了快速迭代、小步快跑的路线上

#### 瀑布模型

将功能的实现与设计分开，便于分工协作，将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护等六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落

**缺点**

项目各个阶段之间极少有反馈，不适应用户需求的变化

我们的系统也从第一代平台开始到现在第四代平台更换中，对这四代平台做一个简单的介绍： 第一代平台，主要是集中式，以快速上线为目的；

第二代平台主要是分布式改造，缓解各服务压力；

第三代平台主要做服务端SOA治理，后台统一账户中心；

第四代微服务化改造，已达到灰度上线、动态部署集中管理的目的。

我也从负责Java端，到负责整个技术团队，慢慢的在领导的信任下测试交给了我，再后来分公司独立后将运维也交给了我，于是成了整个分公司的技术负责人。这就是我的故事。未来仍然有更多的挑战，感谢我们团队的兄弟姐妹，感谢工作中遇到的所有同事和领导。

# 第二篇 J2EE

## Web

### 2.1.1 session机制

Http是无状态协议，浏览器的每一次请求，服务器会独立处理，不与之前或之后的请求产生关联, 所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session。

在服务端保存Session的方法很多，内存、数据库、文件都有。集群的时候要考虑Session的共享。Servlet容器会创建和管理HttpSession，像Tomcat/Jetty都是保存在内存中的。数据过多影响服务器性能。

session一般被用来保存临时全局变量，非常少被用于来传递参数。

登录模块

#### session共享

Tomcat

原理：服务端会生成的JSESSIONID，写客户端cookie.之后的request的head都带着。

（如果不设置cookie过期时间，cookie会在会话结束后销毁，称为会话cookie）



Apache 做负载均衡，如何实现session共享?

* session复制
* session sticky 用户的请求转发到特定的Tomcat服务器上

**session共享方式**

1.宕机时可用性

2.可以逐台机器发布

Spring Session

[Reference](http://blog.csdn.net/xiao__gui/article/details/52706243)

设计一个Filter，利用HttpServletRequestWrapper，实现自己的 getSession()方法，接管创建和管理Session数据的工作。

Nginx + Redis

自定义filter, Nginx +tomcat+ redis

Nginx配置为non-sticky运行模式，也即每一个请求都可以被分配到集群中的任何节点

WebSocket session

#### 单点登录

[Reference](http://www.cnblogs.com/ywlaker/p/6113927.html)

### Http

#### 三次握手

#### 窗口滑动机制

#### GET/POST

1.根据HTTP规范，GET用于信息获取，而且应该是安全的和幂等的。POST表示可能修改变服务器上的资源的请求

2.GET请求的数据会附在URL之后,如：login.action?name=tom。POST把提交的数据则放置在是HTTP包body中

3.GET方式提交的数据最多只能是1024字节，理论上POST没有限制，可传较大量的数据

Unicode编码

### Security

1. 自定义filter
2. Spring-security
3. Shiro

#### 权限

Oauth

### Servlet

HttpServletRequest

HTTP请求头中的所有信息都封装在这个对象中。

Code：RequestDemo01

* request.getRequestURL 获得客户机信息
* request.getParameter 表单POST数据
* request.getSession.setAttribute() 作用域是整个会话期间
* request.setAttribute()，作用域是请求和被请求页面之间。request.setAttribute()是只在此action的下一个forward需要使用的时候使用；

判断未登录或session过期

request.getSession (false) ==null

request.getSession (false).getAttribute (“user”) ==null

HttpServletResponse

response.sendRedirect (“redirectURL”);

Filter

过滤请求。过滤器的执行顺序跟xml文件中定义的先后关系有关。

过滤字符编码，逻辑判断，如是否登录，

权限控制

**Filter vs AOP**

AOP无法拦截直接访问静态资源的请求。Filter仅仅过滤请求。

Listener

它也是随web应用的启动而启动，只初始化一次，随web应用的停止而销毁。主要用做一些初始化

## 2.2 WEB Project

### 应用分层



* 开放接口层：可直接封装 Service 方法暴露成 RPC 接口； 通过 Web 封装成 http 接口； 进行网关安全控制、流量控制等。
* 终端显示层：各个端的模板渲染并执行显示的层。当前主要是 velocity 渲染，JS 渲染，

JSP 渲染，移动端展示等。

* Web 层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
* Service 层：相对具体的业务逻辑服务层。
* Manager 层：通用业务处理层，它有如下特征：

1） 对第三方平台封装的层，预处理返回结果及转化异常信息；

2） 对 Service 层通用能力的下沉，如缓存方案、中间件通用处理；

3） 与 DAO 层交互，对多个 DAO 的组合复用。

* DAO 层：数据访问层，与底层 MySQL、Oracle、Hbase 等进行数据交互。
* 外部接口或第三方平台：包括其它部门 RPC 开放接口，基础平台，其它公司的 HTTP 接口。

在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制，

如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。

说明： 如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其

它用户，并造成短信平台资源浪费。

### 参数校验

用户请求传入的任何参数必须做有效性验证。

说明：忽略参数校验可能导致：

 page size 过大导致内存溢出

 恶意 order by 导致数据库慢查询

 任意重定向

 SQL 注入

 反序列化注入

 正则输入源串拒绝服务 ReDoS

说明：Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，

但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

### MVC

### Spring

#### MVC

分层设计1.有利于分工 2.复用3.好扩展，好维护

原理

Request🡪 DispatcherServlet (web.xml)🡪HandlerMapping(All Controller Map)🡪Controller

#### IOC

Spring作为第三方管理资源对象

1)资源集中管理，实现资源的可配置和易管理。**(面向接口编程,扩展性高)**。

2)降低了资源双方的耦合度。让开发人员更多的关注业务代码

#### AOP

实现原理：

1. Aspectj静态代理



2)CGLIB动态代理/JDK动态代理: JDK针对接口

日志

#### 2.2.4 事务管理

使用AOP切面实现



**Propagation**

传播行为，ServiceB.methodB ------🡪 ServiceA.methodA

* REQUIRED：没有就起一个新的事务。methodA或methodB出现异常，事务都会被回滚。
* SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。
* NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* REQUIRES\_NEW：新建事务，如果当前存在事务，把当前事务挂起。
* MANDATORY：支持当前事务，如果当前没有事务，就抛出异常。
* NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

**Isolation**

与数据库的隔离级别相对应。

READ\_UNCOMMITTED/ READ\_COMMITTED/ REPEATABLE\_READ/ SERIALIZABLE

**Rollback-for**

人为控制事务

**Read-only**

提高事务处理的性能。

### Hibernate

#### 缓存机制

* Session的缓存: 应用场景太单一，系统中大量的列表式查询缓存起不到作用。系统中通过ThreadLocal在线程中重用Session，每个线程可能需要大量处理不用的业务逻辑，缓存命中率很低。
* SessionFactory应用级缓存。EHCache是Hibernate中的二级缓存插件，使用Hibernate的系统可以直接使用EHCache缓存。

#### 丢失更新

TransA在读的过程中，B可以修改并提交数据。即便数据在TransB中发生改变，A返回相同结果集合

处理的数据变化引起其他潜在问题,比如营业额增加，银行转账。

Oracle可以把隔离级别设置成read-only。Mysql解决的方法：

* 乐观锁，Hibernate框架已经实现。注意REPEATABLE-READ中不能使用乐观锁。因为它是可重复读的。
* 悲观锁(for update)
* 设置为Serializable

加锁策略

并发修改同一记录时，避免更新丢失，需要加锁。要么在应用层加锁，要么在缓存加锁，要么在数据库层使用乐观锁，使用version作为更新依据。

说明：如果每次访问冲突概率小于20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次数不得小于3次。

Hibernate实现方案

只读事务

加锁是为了解决丢失更新问题。针对一些场景动态的调整事务隔离级别。



Mysql：true的时候会读到最新值

乐观锁

[Reference](http://www.voidcn.com/article/p-sdnzagtc-oy.html)



悲观锁

查询的时候对表数据进行锁定

依靠数据库提供的锁机制。实际应用中悲观锁是很少被使用的，因为它大大限制了并发性

select \* from student for update



## 2.3 通信

### NIO

NIO是Java SE 1.4及后续版本提供的一种新的I/O操作方式。是一个基于缓冲区、并能提供非阻塞I/O操作的Java API。

* 非阻塞: 避免创建大量线程，相对于多线程开销较小。
* 基于事件驱动: 线程上下文的切换都是有意义的。
* 复用单一长连接，并使用线程池并发处理请求，减少握手和提高并发效率，性能较好。长连接双向异步推送，比轮询、阻塞（线程无效切换的开销）效率高。

### Netty

对JDK网络编程的封装，屏蔽了繁杂的编程细节，提供便于用户开发网络应用程序的api,让开发者可以更加专注于业务逻辑的实现。

**基础通信组件,RPC工具**

* 很多中间件把netty作为基础通信组件。
* 阿里分布式服务框架 Dubbo
* 淘宝的消息中间件 RocketMQ 的消息生产者和消息消费者之间, 也采用 Netty 进行高性能、异步通信。

**1.异步**

非阻塞，事件驱动机制

**2.高性能的通信框架**

不用依赖于容器去进行部署,易扩展

### Mina

架构

多工作线程Reactor模式



类结构



**Acceptor**：

处理连接请求

**NioProcessor**

循环读



连接管理

session相对http较长,手动管理。





心跳机制

## 2.4 中间件

[Reference](https://kb.cnblogs.com/page/196448/)

一般情况，中间件应用于分布式环境，主要解决异构网络环境下模块的互连与互操作问题，提供标准接口、协议，屏蔽实现细节（将具体业务和底层逻辑解耦的组件），提高应用系统易移植性。

分类：数据访问中间件，远程调用中间件，消息中间件，交易中间件，对象中间件。

### 消息中间件

优势：事件驱动（订阅发布）可一对多推送消息，异步，消息可靠

[Reference](http://blog.csdn.net/dongnan591172113/article/details/8771441)

**消息队列**

通过消息分解系统解耦性，不同子系统处理同一消息

**使用场景**

1. 系统整合：

系统整合：分布式环境中,用于同构/异构系统的整合,而且系统间耦合度较低，

1)AB系统,A宕机

2)性能：系统差不干扰B

3) 系统整合模块交互点统一，变化集中在一点，不用逐个模块修改,垂直切分容易

2.分解系统: 通过消息分解系统。

3. 异步:1.推消息 2.流量削峰(据同步:web应用->缓存，搜索，db)

#### Kafka

日志处理

#### ActiveMQ

基于JMS

## 2.5分布式服务

谈谈分布式

通过接口分解系统解耦性，不同子系统通过相同的接口描述进行服务调用。

### SOA

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，

1.扩展性 2提高开发和维护效率 3.伸缩性?

**面临的挑战**

1. 远程调RPC用
2. 服务管理和配置管理
3. 负载均衡器单点故障
4. 实时管理集群
5. 分布式事务

**SOA服务治理**

cas单点登录

Zookeeper服务的注册和发现

Zookeeper配置中心

Web service

Web service是什么

WebService，顾名思义就是基于Web的服务。基于HTTP传输协议的程序。接收和响应外部系统的某种请求。从而实现远程调用。

### Restful

规范

接口授权OAuth

### Atom

协议

### Zookeeper

Zookeeper是一个高性能分布式应用协调服务。分布式应用可以基于它实现同步，配置管理，集群管理，命名名空间

 随着线上项目变的日益庞大，每个项目都散落着各种配置文件；因为采用分布式的开发模式，项目之间的相互引用随着服务的不断增多，相互之间的调用复杂度成指数升高，每次投产或者上线新的项目时苦不堪言，因此需要引用配置中心治理。

分布式琐

### Dubbo

### 微服务

微服务 vs SOA

soa和微服务有很多概念是相通的，并且微服务是由soa演化过来的，soa更加专注服务之间的治理，微服务考虑的更全面一些，微服务更独立一些，最明显微服务要求有自己独立的库，soa只是强调服务的独立。

### SpringCloud

## 2.6 Solr

页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决

## 2.7前端技术

### Freemarker

在java领域，表现层技术主要有三种：jsp、freemarker、velocity。

优点：1.不能写java代码，实现严格的mvc分离2.可使用表达式语言

## 缓存

### Memcached

Memcached client for java (code: MemCache120)

官方提供, 运行比较稳定，使用阻塞IO，不支持CAS操作。

XMemcached

# 第三篇 Design

## 3.1设计准则

接口优于抽象类

将变化控制到最小

**工厂方法模式的优点**

低耦合、高内聚，符合开放封闭原则

1开闭原则2单一职责 2依赖倒转

Interface vs Abstract

**接口和抽象类的区别是什么？**

1.abstract是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个interface。

2.在abstract class 可以有自己的数据成员，也可以有非abstract的成员方法

3. abstract class和interface所反映出的设计理念不同。其实abstract class表示的是"is-a"关系，interface表示的是"like-a"关系。

## 3.2 设计模式

### 单例模式

[单例模式](http://blog.csdn.net/dongnan591172113/article/details/51027374)

### 观察者模式



## 3.3 软件架构分析

**软件架构优劣的衡量要素：**1.性能2.可用性 3.伸缩性4.扩展性5.安全性

### 性能

#### 1性能优化指标

* 并发数：能同时处理的用户数。
* 响应时间：发出请求到收到相应数据的时间
* 吞吐量：系统单位处理的请求数。QPS（query per s）
* 性能计数器：包括System Load、对象和线程数、内存、CPU、磁盘、网络IO

#### 2 性能测试方法

* 性能测试：性能预期
* 负荷测试：找到性能安全临界线
* 压力测试：超过安全符合继续施压，获得最大压力承受能力

#### 3 Web前端性能优化

* 减少Http请求:合并css、js
* 使用浏览器缓存:生成新的js时，只需更新Html中引用
* 启用压缩：减少数据传输
* CSN加速：缓存静态资源，如图片、文件、CSS、JavaScript脚本、静态网页等。
* 反向代理

#### 4) 应用服务器的优化

* 缓存：加快数据读取，降低DB访问压力。网站遇到性能瓶颈，第一个想到的解决方案
* 异步操作：加快请求响应及实现削峰。需要修改业务流程进行配合。
* 使用集群：使用负载均衡，有更好的响应延迟，提高吞吐能力。
* 代码优化：1)多线程,最大发挥CPU和处理IO阻塞。 2)资源复用，比如单例和线程池。 3)数据结构 4)垃圾回收

### 伸缩性

**不同功能物理分离实现伸缩**

分割的粒度可以非常小，甚至可以一个关键网页部署一个独立服务，比如对电商非常重要的商品详情页面，商铺页面，搜索列表页面，每个页面都可以独立部署，专门维护

**单一功能通过集群规模实现伸缩:**应用服务器，搜索，缓存，数据库

**缓存**

**数据存储服务**

1）主从分离

2）分表分库

3) NoSql

HBase: MongoDB:

### 扩展性

**纵向拆分**：按功能模块拆分为多个应用系统，新增业务可以设计成独立的web系统。

**横向拆分：**拆分出复用的基础公用业务，独立部署为分布式服务。

分布式系统集成方式：1）分布式消息队列2）分布式服务

消息中间件

通过消息分解系统解耦性，不同子系统处理同一消息

分布式服务

通过接口分解系统解耦性，不同子系统通过相同的接口描述进行服务调用。

### 可用性

实现高可用架构的手段是数据冗余备份和服务失效转移。

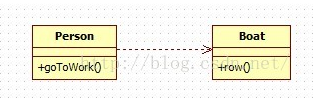
服务器：由无状态服务器和有状态的Session服务器组成。

## 3.4 Design Tools

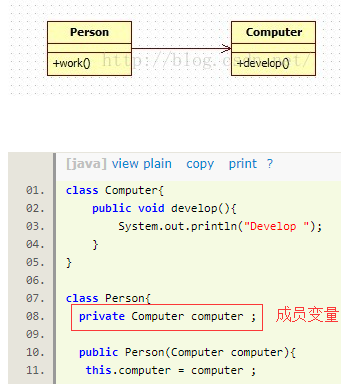
### UML图

依赖/管理/继承/实现

1.依赖(Dependency ): 局部变量(方法里new对象)、方法的参数或者是静态方法的调用;



2.关联



关联：组合/聚合

**组合(Composition):** Room是house组成部分，有相同的生命周期。



**聚合(aggregation):**



# 第四篇 DB

## 4.1范式

[Reference](http://blog.csdn.net/dongnan591172113/article/details/7910988)

## 4.2事务

### 4.2.1事务特性

**Atomic**（原子性）

要么全部成功，要么全部失败。

**Consistency**（一致性）

数据不会因为事务的执行而遭受破坏.只有合法的数据可以被写入数据库，否则事务应该将其回滚到最初状态。

实现: 完整性约束：域约束，基本表约束和断言

**Isolation（隔离性）**

事务允许多个用户对同一个数据进行并发访问，而不破坏数据的正确性和完整性。

实现：事务的锁来进行控制。

**Durability（持久性）**

### 4.2.2 隔离级别

[Reference](http://blog.csdn.net/dongnan591172113/article/details/47948935)

* Read uncommitted可能出现脏读
* Read committed 同一查询在同一事务中多次进行，由于其他提交事务所做的修改或删除，每次返回不同的结果集
* Repeatable read  其他事务插入，此时发生幻像读。行级锁。mysql默认隔离级别。
* Serializable 序列化。表锁。

上述针对A读，B写来描述的。

A读，B写 ->OK,丢失更新问题。

A写，B写 ->NO

锁定的资源包括行、表、页、簇和数据库。为了最小化锁的成本，尽量锁定比较小的对象

## 4.3 MySQL

一条sql执行过长时间，如何优化

1.一般联表消耗更多系统资源，还可能使索引失效，涉及多表的联表或者子查询

1）看是否能进行业务拆分2）相关字段冗余或者3）创建汇总表。4)分解关联查询，单表查询之后的结果进行字段整合  
2.需要联表查询

1)那么考虑对相对应的查询条件做索引。加快查询速度  
3、针对数量大的表进行历史表分离（如交易流水表）  
4、读写分离，降低主库压力

### 4.3.1逻辑架构



每个虚线框为一层，总共三层。

第一层，服务层(为客户端服务):为请求做连接处理，授权认证，安全等。

第二层，核心层:查询解析，分析，优化，缓存，提供内建函数;存储过程，触发器，视图。

第三层，存储引擎层，数据存储和提取数据，事务处理。

**查询的生命周期**

客户端-> 服务器 ->解析-> 生成执行计划-> 执行-> 客户端

### 4.3.2索引

#### 索引类型

B-Tree索引

Hash索引

全文索引

B树/B+树/区别/数据库如何使用B+树建立索引？

Normal vs Unique

1. 提高访问速度 2.保证数据唯一性

Full Text

#### 优点

1）索引大大减少了服务器需要扫描的数据量

2）索引可以帮助服务器避免排序和临时表

3）索引可以将随机I/O变成顺序I/O

哪些字段要索引

Where column, order by

#### 索引策略

**聚簇索引**:按物理位置顺序存放，引擎缓存索引和数据

插入慢，排序快。非聚簇索引（二级索引）则没有按序存放，需要额外消耗资源来排序。

**组合索引**

### 4.3.3存储引擎

影响引擎性能的因素：数据大小，I/O请求量，主键还是二级索引

#### InnoDB

优先选择InnoDB，除非用到InnoDB不具备的特性，没有其他办法替代。

1. 事务2.热备份3.崩溃恢复4.聚簇索引

#### MyISAM

一般读写性能高，但随着应用压力上升，各种锁争用，崩溃数据丢失问题

1. MyISAM引擎缓存索引，而不缓存数据
2. 不支持事务、外键和行级锁,适合小项目

优点:1.读多写少的项目，效率高2.没有磁盘碎片，节省空间

### 4.1.4 Explain执行计划

http://static.oschina.net/uploads/space/2012/0530/153119_3rF0_195637.jpg

**id**

id越大则优先级越高，越先会被执行。如果id相同，则执行顺序从上至下

**select\_type**

simple: 不使用UNION或子查询等

primary: 最外层的select

subquery，

derived:里层的子查询



union，

unionresult

**type**

all，index，range，ref，eq\_ref，const，system，NULL

* ALL，即full table scan，（到存储引擎）遍历全表
* index为full index scan，遍历全部的索引树。当查询只使用作为单索引一部分的列时



* range: 1.使用索引 2.只检索给定范围的行



* ref: Normal Index 1.使用索引 2.匹配条件用于查找索引列上的值



* eq\_ref: 类似ref，区别:索引列是PRIMARY KEY或者unique index
* const: 表最多有一个匹配行



* NULL表示在执行语句中，不用查表或索引

**key**

表示查询时使用的索引

PRIMARY

**rows**

估算读取的行数

**Extra**

* using index：通过查询索引就能得到结果。
* using where: 需要查询磁盘里存储的数据
* using temporay表示用临时表来存储结果集，常见于排序和分组查询
* using filesort，mysql中无法用索引完成的排序成为文件排序。

## 4.4 MySQL性能优化

 根据不同业务场景有不同的侧重，优化的策略不同，比如数据量、访问量（QPS每秒查询率）、并发量、数据是否快速增长、数据需不需要强一致、读写比例？对于一些典型的场景可能会有成熟的解决方案。

1. 索引优化 2.查询优化 3.库表结构优化

Note:

索引增加写数据的额外消耗

### 查询优化

查询性能低下最基本原因是访问数据太多。

#### 应用层

* 确认应用服务是否检索大量查过需要的数据（访问太多的行或列）

加limit

* 返回所有列或者多表关联返回需要的列
* 重复查询相同的数据：缓存

#### MySQL服务器

Explain 分析服务器执行计划**，**确认MySQL服务器是否在分析大量超过需要的数据行

**优化技巧**：

1)使用覆盖索引扫描，把所有需要的列都放到索引中，这样存储引擎无须返回表获取对应行就可以返回结果了。

2)改变库表结构，使用单独的汇总表。

3)重写这个复杂的查询。

#### 优化特定的查询

##### Join

超过三个表禁止join。需要join的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引。

##### 分表分库

单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。

说明：如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表

#### 缓存表和汇总表

冗余是在一张表。

汇总表：group by 或count

缓存表：存储从schema中获取比较慢的数据表。

例如主键+部分列。用缓存或solr(搜索效率更高)

**优点**：避免复杂，昂贵的实时跟新操作

##### Count

1. MyISAM 2. 索引覆盖扫描3.汇总表 4.增加memcached

##### Where

一般MYSQL能够使用如下三种方式应用where条件记录，**从好到坏**依次为

1. 在存储索引层，在索引中使用where条件过滤。



2) 在mysql服务器层, 扫描索引过滤并返回记录。无需在回表查询记录(Using index)。

3) 在mysql服务器层，先从数据表读出记录然后过滤(Using Where)

##### 关联查询

1. ON或using列有索引,一般在连表顺序第二个create index。
2. Group by和Order By之设计一个表的列
3. **1)复杂查询还是简单查询**
4. MySQL吞吐量：10w/s查询
5. 扫描内存：100+w行/s
6. **2)分解关联查询**
7. **单表 vs 联表**
8. 看具体业务场景。没到阀值前，都可以，有的业务也许永远到达这个阀值。
9. 高并发高读环境：单表or冗余，可做缓存。建覆盖索引。
10. 联表后期扩展和维护工作量大，简单说dao层的复用率低。

**子查询：**尽量 关联子查询替代，当前MySQL版本需要

##### GROUP BY和DISTINCT

LIMIT：一般ORDER BY …LIMIT

##### 搜索

页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。

说明：索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索

引。

### 库表结构优化

#### 选择合适的数据类型

* 更小的通常更好：占用更少的磁盘、内存和CPU缓存，CUP处理周期
* 简单就好：简单类型CPU处理快。例如整形比字符操作代价低
* 尽量避免NULL：NULL使索引更复杂，需要更多存储空间

#### 范式和反范式

范式需要联表，代价昂贵。还可能使一些索引策略无效，比如覆盖索引。

### 索引优化

一般oracle会判断,使用索引返回的数据量不超过总表的5%时,才走索引..否则就是全表扫描

#### 扫描行数和访问类型

ALL-index-range-ref

[MYSQL数据库优化](http://blog.csdn.net/dongnan591172113/article/details/52174619)

**减少索引和数据的碎片**

Optimize table;

## 主从分离架构



## MySQL Workbench



Key Efficiency

<https://gxnotes.com/article/72479.html>

## 4.5 ORACLE

### 常用函数

#### Decode

IF 条件=值1 THEN  
　　　　RETURN(翻译值1)  
ELSIF 条件=值2 THEN  
　　　　RETURN(翻译值2)  
　　　　......  
ELSIF 条件=值n THEN  
　　　　RETURN(翻译值n)  
ELSE  
　　　　RETURN(缺省值)  
END IF

### PLSQL

**优势**

1能够使一组SQL语句的功能更具**模块化**程序特点；

2.采用了**过程性语言控制程序**的结构；

3.集成在数据库中，调用更快；

4.**减少了网络的交互**，有助于提高程序性能。

SQL语言只是访问、操作数据库的语言，并不是一种具有流程控制的程序设计语言，而只有程序设计语言才能用于应用软件的开发。PL/SQL是一种高级数据库程序设计语言。PL/SQL同样具备高级语言的一些特性，比如模块化，过程控制，异常等。

1) 多条SQL语句实现功能时，每条语句都需要在[客户端](http://baike.baidu.com/view/930.htm)和[服务端](http://baike.baidu.com/view/1087294.htm)传递，而且每条语句的执行结果也需要在网络中进行交互，占用了大量的网络带宽，消耗了大量网络传输时间，而在网络中传输的那些结果，往往都是中间结果，而不是我们所关心的。

2) PL/SQL程序代码存储在数据库中，程序的分析和执行完全在数据库内部进行，用户所需要做的就是在[客户端](http://baike.baidu.com/view/930.htm)发出调用PL/SQL的执行命令，数据库接收到执行命令后，在数据库内部完成整个PL/SQL程序的执行，并将最终的执行结果返馈给用户。所以整体程序的执行性能会有明显的提高。

Print Switch:

set serveroutput on dbms\_output.put\_line('cursor:')

## 4.6 NOSQL

现代互联网应用，关系数据库的很多主要特性却往往无用武之地

1.数据库事务一致性需求

2.数据库的写实时性和读实时性需求

3.对复杂的SQL查询，特别是多表关联查询的需求

去掉关系型数据库的两大重要基础：以关系代数为基础的结构化查询语句（SQL）和事务一致性保证（ACID）

1. 高并发读写2)海量数据的高效率存储和访问3)高可用性和可伸缩性

### 4.6.1 Redis

redis是一款开源的高性能key-value数据库，Redis支持更丰富的数据结构，例如hashes, lists, sets等。除此之外，Redis还提供一些类数据库的特性，比如事务，HA，主从库。与一般数据库不同，redis是使用内存作为主存，周期性的将数据写到硬盘上。因为redis读写数据都使用内存，所以它的速度是非常快的，很适合我们来存放一些临时性的数据。

Redis vs Memcached

1.使用简单的key-value存储的话，Memcached的内存利用率更高

2. 由于Redis只使用单核，而Memcached可以使用多核， Memcached相对性能更高

3.拥有更多的数据结构和并支持更丰富的数据操作。

4.集群管理的不同

**管理Session**

# 第五章 服务器

## Tomcat

### 性能优化

#### 1.内存优化

参考JVM优化

#### 2.线程优化

**1) Max Number of threads:**

(MaxProcessMemory - JVMMemory - ReservedOsMemory) / ThreadStackSize

* MaxProcessMemory 指的是一个进程的最大内存。在32位的 windows下是 2G
* JVMMemory         JVM内存
* ReservedOsMemory  保留的操作系统内存
* ThreadStackSize      线程栈的大小(-Xss)

2)线程数量

((CUP时间+CUP等待时间)/CUP时间)\*CUP数量

3)Jvisual



Running:运行

Park:驻留。线程池?

Monitor:监视锁

Running: thread is still running.   
Sleeping: thread is sleeping (method yield() was called on the thread object)  
Wait: thread was blocked by a mutex or a barrier, and is waiting for another thread to release the lock  
Park: parked threads are suspended until they are given a permit. Unparking a thread is usually done by calling method unpark() on the thread object  
Monitor: threads are waiting on a condition to become true to resume execution

**在tomcat配置文件server.xml中的配置中，和连接数相关的参数有：**

**maxThreads**：最大的线程数。Tomcat7默认值200。

acceptCount： 当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。默认值10。

**minSpareThreads**： Tomcat初始化时创建的线程数。默认值25。

**maxSpareThreads**： 一旦创建的线程超过这个值，Tomcat就会关闭不再需要的socket线程。默认值75。

enableLookups： 是否反查域名，默认值为true。为了提高处理能力，应设置为false

connnectionTimeout： 网络连接超时，默认值60000，单位：毫秒。设置为0表示永不超时，这样设置有隐患的。通常可设置为30000毫秒。

maxKeepAliveRequests： 保持请求数量，默认值100。 bufferSize： 输入流缓冲大小，默认值2048 bytes。

compression： 压缩传输，取值on/off/force，默认值off。 其中和最大连接数相关的参数为maxThreads和acceptCount。如果要加大并发连接数，应同时加大这两个参数。

**32G 内存配置示例：**

<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" maxThreads="1000" minSpareThreads="60" maxSpareThreads="600" acceptCount="120"

redirectPort="8443" URIEncoding="utf-8"/>

#### 3 IO优化

### 发布

热部署vs热加载，热加载是运行时通过重新加载改变类信息，直接改变程序行为。

1)静态部署

2)动态部署:不用重新启动服务器。

优雅停机

shutdown.sh

### Apache和tomcat

apache是web服务器，专门提供HTTP服务。处理静态内容，比如HTML/图片。

tomcat是java应用服务器，servlet容器，是apache的扩展

### 源代码

**下载**:<http://blog.csdn.net/dalinsi/article/details/53493559>

**Connector BIO**



Http11ConnectionHandler

|- Http11ConnectionHandler

|- JIoEndpoint

JIoEndpoint

|- Acceptor

|- SocketProcessor

|- Handler

Connector NIO

**源码分析：**<http://tyrion.iteye.com/blog/2256896>

**分层建模**

tomcat架构的高度模块化。这些细分的模块，使得tomcat非常健壮，通过一些配置和模块定制化，可以很大限度的扩展tomcat。



* Wrapper封装了具体的访问资源，例如 index.html
* Context 封装了各个wrapper资源的集合，例如 app
* Host 封装了各个context资源的集合，例如 www.mydomain.com

## Apache

## Nginx

**优点**

* 异步非阻塞，高并发环境消耗更少的资源
* 配置简洁,

### Nginx与apache

Nginx性能高，而apache是阻塞的。但比较稳定。

1.反向代理，缓存静态文件 2.负载均衡

# 第六篇 Linux

## 6.1系统资源分析

### TOP

<http://www.cnblogs.com/ggjucheng/archive/2012/01/08/2316399.html>

Load Average 就是一段时间 (1 分钟、5分钟、15分钟) 内平均 Load

PR 优先级

VIRT 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES

RES 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA

SHR 共享内存大小，单位kb

1)进程ID来仅仅监控某个进程

top –p pid

top –Hp pid

**快捷键**

P:CPU排序

M:内存排序

H: show thread

1:show cups

### FREE

[Reference](http://www.cnblogs.com/coldplayerest/archive/2010/02/20/1669949.html)

[nadong@slc11fsp conf]$ free -m

total used free shared buffers cached

Mem: 16885 16186 698 7 4442 6753

-/+ buffers/cache: 4991 11893

Swap: 16559 35 16524

-/+ buffers/cache: 应用程序认为系统被用掉多少内存

设置缓存大小：

### DF

Disk File

df –hl

df -h /home/nadong

–i 查看inodes

### du

Disk usage of each File

du –sh

分区

### PS

Process Status

-ef 查看进程的父进程ID和完整的COMMAND命令

-aux 查看进程的CPU占用率和内存占用率

a：显示现行终端机下的所有程序，包括其他用户的程序

u 以用户为主的格式来显示程序状况

 x 显示所有程序，不以终端机来区分

### netstat

**Param**

-n 拒绝显示别名(域名)，能显示数字的全部转化成数字(ip)。

-a (all)显示所有选项，默认不显示LISTEN相关  
-t (tcp)仅显示tcp相关选项  
-u (udp)仅显示udp相关选项  
-p 显示建立相关链接的程序名

-l 仅列出有在 Listen (监听) 的服務状态

-x**只列出所有监听 UNIX 端口**

提示：LISTEN和LISTENING的状态只有用-a或者-l才能看到

p 显示建立相关链接的程序名  
-r 显示路由信息，路由表  
-e 显示扩展信息，例如uid等  
-s 按各个协议进行统计  
-c 每隔一个固定时间，执行该netstat命令

Active Internet connections，称为有源TCP连接

Active UNIX domain sockets，称为有源Unix域套接口(和网络套接字一样，但是只能用于本机通信，性能可以提高一倍)

**State**

ESTABLISHED: 意思是建立连接。表示两台机器正在通信

CLOSE\_WAIT: 对方主动关闭连接或者网络异常导致连接中断

TIME\_WAIT: 我方主动调用close()断开连接，收到对方确认后状态变为TIME\_WAIT

LISTEN

**常用**

1)netstat -at|grep slc01boa

2)统计80端口连接数

netstat -nat|grep -i "8080"|wc –l

3)Linux下查看tomcat连接数

netstat -na | grep ESTABLISHED | grep 8080 | wc -l

1. 统计httpd协议连接数  
   ps -ef|grep httpd|wc –l
2. 统计已连接上的，状态为“established  
   netstat -na|grep ESTABLISHED|wc -l

3)找出程序运行的端口

netstat -ap | grep java

**4)**[**根据端口查找进程**](http://www.cnblogs.com/paul8339/p/6638370.html)

netstat -apn | grep 8989

netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

## 6.2文件

### chmod

chmod u+w test

chmod ugo+r test;

### find

find -name ‘httpd\*' 当前及子文件夹查找

find /etc -name httpd.conf

### whereis

whereis useradd

### lsof

### more

Enter:一行

Space:翻页

### less

less 的用法比起 more 更加的有弹性。在 more 的时候，我们并没有办法向前面翻， 只能往后面看

Space:翻页

G:倒序查看

### vim

**全部删除：**按esc后，然后dG

## 6.3 用户权限

### 用户组

**1)groups**:  查看当前登录用户的组内成员

**2)/etc/group**

group\_name:passwd:GID:user\_list

**3)groupadd**

group add group1

**4)groupdel**

删除群组。先删除group的用户，方能删除群组

**usermod**

usermod -a -G root nadong

将一个用户添加到用户组中c

**chown**

将指定文件的拥有者改为指定的用户或组

chown -R mysql:mysql ./

### 用户

**useradd**

adduser moon

新建用户::useradd -d /home/moon -m moon

修改用户的密码: passwd tes

**userdel**

userdel  moon

1. 将新用户添加到管理组gpasswd -a test admin

3、给 test 用户创建自己的目录：

cd /home

mkdir test

chown test /home/test

4、重新启动，

reboot 然后用 test 登录，

**su**

5. :Su switch use /sudo输入当前用户的密码。

**sudo**

1.需要root赋予sudo权限

su root🡪 hadoop ALL=(ALL) ALL

## 6.4 CMD

1.nginx  
1)启动操作  
nginx -c /usr/local/nginx/conf/nginx.conf  
2)停止操作  
从容停止Nginx  
快速停止Nginx：  
kill -TERM 主进程号  
  
强制停止Nginx：  
kill -9 主进程号  
3)  
nginx -s reload  
  
2.mysql  
#启动服务 service mysqld start  
#连接 ./mysql/bin/mysql -uroot  
DDL  
show databases/tables  
desc table\_name;  
  
3.redis  
1)启动： redis-server.exe redis.windows.conf

开启memcache，并连接测试：  
以守护进程模式启动memcache  
memcached-1.5.2/bin/memcached -d -l slc11fsp.us.oracle.com -p 11211 -m 2048 -u root