WeakReference

**TODO：接口耦合度级别**

接口参数如何做到合理规范？[Reference](https://blog.csdn.net/coslay/article/details/38293689)

ldc压入栈后是否在heap中创建对象？

TODO：static内部类

死锁

java lock monitor源码

synchronized源码，复习C++

对象在内存的结构

# 第一篇 J2SE

## BASIC

### JDK

JDK,JRE,JVM关系

[Reference](https://blog.csdn.net/songkai320/article/details/51819046)

OpenJDK和SunJDK区别

openjdk是sunjdk的开源版

和Google的Android关系

### final

**final Class:**

类不能被继承。方法不能rewrite

**final Method:**

方法不能rewrite

**final Variable**

修饰的基本类型的变量，初始化后不能更改。

修饰引用类型不能执行另一个对象，但对象内容可以更改。

### static

TODO：static内部类

### 数组

#### final数组

数组也是对象，引用value的地址不能改变，内容可改变



### String

#### immutable



数组value是也是对象，内容是可变的，可见String的不可变性在于底层的实现，而不是一个final。

**目的**

设计成immutable的类，可以像普通类型一样赋值，而不是普通的类。

**好处**

主要是为了“效率” 和 “安全性” 的缘故。

效率：对象的不可变行，所以可以共享。节省内存空间。

安全性：string作为对象，可以像基本类型值传递，而不改变对象的值。

不仅String 是final的。 Long, Double, Integer 之类的全都是final的

#### 内存位置

String str=”abc”;

heap创建对象abc,常量池存着指向对象的指针。栈中创建的变量str执行常量池。

String str=new String(“abc”);

#### new和直接赋值

String str=”a”

* str引用指向对象类型，String对象既可以存在heap也可以在常量池，且字节码结构相同。
* 字节码中常量池转换到方法区中常量池，字符串常量转换成对象，对象结构参考下章。



**为什么入栈后赋值?**

String str=new String(“a”)



line 4,5

执行过程

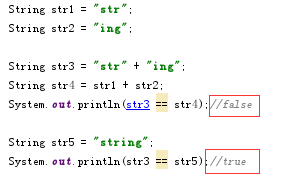
<https://www.ibm.com/developerworks/cn/java/j-lo-optmizestring/index.html>

intern（）作用？

String str=new String(“abc”),构造器中字符串是不是对象，存储位置，运算过程？

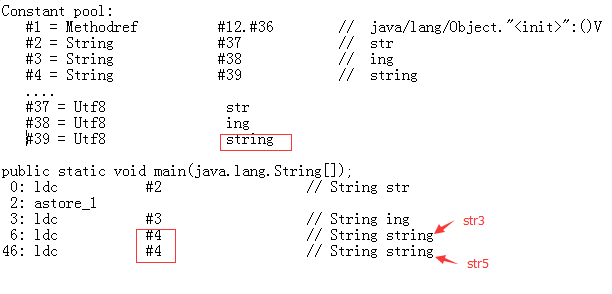
<http://www.hollischuang.com/archives/99>

**相等问题**



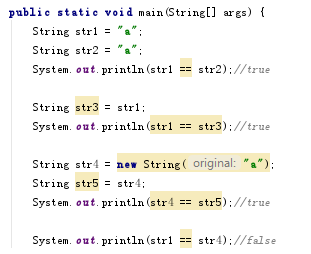
查看字节码

String ，str = "str"; 常量池创建对象“abc”，str指向池中对象”abc”,



可见编译期”str”+”ing”已经生成string, 和str5的符号引用指向同一字面量，到运行期，str3和str5都指向常量池的同一位置，才会相等。str1+str2到运行期动态生成，在heap中生成对象，

Code**:** String100



replaceFirst、replaceAll、replace区别

“+=”的重载

“+=” 是Java中仅有的两个重载过的操作符,而java并不允许我们重载任何操作符。

**对+的重载**

加号被转换为StringBuffer的append()方法来操作,最后toString生成新的对象。



可见String str3 = "str" + "ing"; 编译器对+重载，这种连接直接加入字符串池

String.valueOf和Integer.toString的区别、

### Integer

#### 包装类：

Java的8种基本类型都有对应的包装类。包装类与基本类型的转换：

**自动拆装箱**



装箱:Integer.valueOf()

拆箱：Integer.intValue()

#### valueOf

缓存机制



cache 是Integer的数组。Integer cache[];

可见

-128~128



int型实例变量是否会被缓存？

### 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，用来在散列存储结构中确定对象的存储地址的。

从官方文档的定义，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。

wait ()/notify ()/notifyAll ()

Code：WaitNotifyCase100

使用方法



warning:

线程wait后被notify，要重新获得lock才能从中断处继续执行。

### 常量

[Reference](https://www.jianshu.com/p/c7f47de2ee80)

**常量**

和变量相对，不能更改。在Java中，指final修饰的变量。有三种：静态变量、实例变量和局部变量。

final变量：

1.final类:不能被继承。

2.final方法不能重写

3.修饰引用类型，初始化后不能指向另一个对象。



4.修饰基本类型



好处：

* 使用final关键字，JVM会对方法、变量及类进行优化
* 提高了性能。JVM和Java应用都会缓存final变量。

在多线程环境下进行共享，而不需要额外的同步开销。

**常量池**

实现常量池技术的基本类型包装类：Integer, Long，String

### 变量

**局部变量**：方法中的变量。分配在stack

**类变量**：方法之外的变量，用 static 修饰。分配在方法区。

**实例变量**：方法之外的变量，不过没有 static 修饰。分配在堆

**成员变量**（全局变量）：包括实例变量和类变量

总结: 方法内定义的变量和引用全部在栈中，方法外定义的实例变量全部在堆中，静态变量在方法区。

### 引用类型

## 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 异常

异常分类



Checked异常

可以被处理的异常，必须显示处理。比如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

## 1.4 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。

内省（Introspector）

## 1.5 Thread

线程是操作系统能够进行运算调度的最小单位。

### 1.5.1 多线程

****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

* 线程**创建和销毁**所花时间，以及系统资源开销(线程池)
* 线程调度, **上下文切换** ： 有可能出现“过度切换”的问题
* **内存同步**的开销
* 设计更复杂

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

#### 线程状态

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

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

Blocked

Blocked VS Wait:

。

* 未获得锁进去等待队列EntrySet,已经获得锁主动释放，进入WaitSet。
* 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。
* blocked由jvm唤醒，wait由另一个线程。

Interrupt

Demo:ThreadInterrupt100

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；
* 线程运行时执行interrupt，线程只会设置 “打断标志” 。

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

#### 线程同步

同步指多线程环境，线程之间的协同和配合。线程同步通过加锁机制实现线程安全。

实现同步方法

* ThreadLocal
* sychronized
* Lock
* Atomic

#### 线程安全

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

多线程环境，如果每次运行结果和单线程运行的结果是一样的，就是线程安全的。

线程安全问题都是由**共享变量**(全局变量及静态变量)引起的。

线程不安全，会造成所得到的数据不一致或者数据污染，而出现脏数据。

只有读操作，这个全局变量是线程安全的；若同时执行写操作，需要考虑线程同步。

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****

保证**串行**的顺序。

[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, and 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。

**在java多线程里指的原子性即为互斥，串行执行。**

原子性操作



只有语句1是原子性操作

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。



x是共享变量，线程公有的。test方法读取变量到线程私有的栈中，x在内存中有两份copy，需要同步。和x的内存位置不同。

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。

实现原理

加入volatile关键字关键字时所生成的汇编代码（**非字节码**），会多出一个lock前缀指令

lock前缀指令，相当于一个内存屏障（也称内存栅栏）：

* 禁止指令重排。时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面。
* 强制将对cache的修改立即写入主存并使其他CPU中对应的缓存行无效；

使用场景1

在一些情况下即便是原子操作（flag的赋值）也可能会引发一些错误



线程访问对象的值时，将**堆内存中实例变量的值**copy到**工作内存**。Java HotSpot Server版VM并不会将修改的值写回内存。

使用场景2

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。**这里更新内存的时间点具有不确定性只能用在上述场景，不能用于下面的自增操作，**



保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行



自增并非原子操作,不同保证互斥串行操作。



volatile还可以禁止指令重排，保证有序性



如果语句2会在语句1之前执行，线程2中就使用未初始化的context，导致程序出错

****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。

#### 调用栈

理解JVM方法调用机制：



方法method1中currentThread也是Rt

### 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 ()

main默认group为main

### Thread Lock

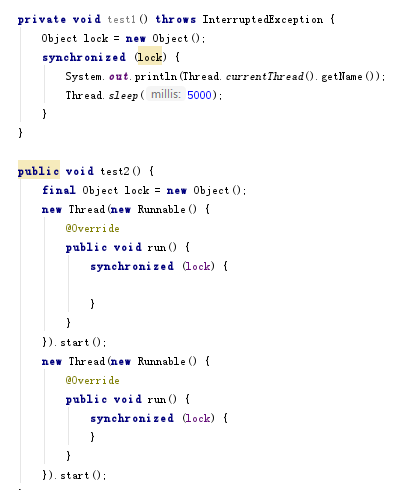
#### 实现原理

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()

#### 锁的使用方式



test1:多线程进入同一方法。

test2：在不同方法体中使用

注意：必须是同一锁才是实现锁定，不同对象new出来的公有变量lock不是同一把锁

#### 悲观锁

假定会出现并发问题，读取数据时即将数据锁定。

**悲观锁需要操作系统参与，会引起线程阻塞和上下文切换**。**相对于乐观锁，系统资源开销大。**

#### 乐观锁

所谓乐观锁就是，假定不会出现并发问题，每次不加锁，冲突失败就重试，直到成功为止。

**借助CPU提供的CAS（compare and swap）指令。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 .

CAS指令

CPU的CAS指令是原子性操作，执行完成前不可中断。本质上采用了缓存一致性协议（比如MESI），执行过程加了排他锁，不会出现线程安全问题。

注意：CAS操作只是实现了原子性,还需考虑可见性。

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。

#### 可重入锁

可重入锁，也叫做**递归锁**，指同一线程外层函数获得锁后，内层函数仍可获取该锁。

**特点：发生在同一线程内**

Code: ReentrantTest101



执行输出“method2”

可重入锁的最大作用就是可以避免死锁。

这里讲的是广义上的可重入锁，而不是单指JAVA下的ReentrantLock。ReentrantLock 和synchronized 都是可重入锁

实现原理

state不为0，

#### 死锁

[Reference](https://juejin.im/post/5aaf6ee76fb9a028d3753534)

多进程在执行过程中，由于竞争资源或者通信而造成的一种阻塞的现象。

竞争的资源，可以是线程池里的线程、网络连接池的连接，数据库中数据引擎提供的锁，等等一切可以被称作竞争资源的东西。

写一个死锁的程序

Java线程调度、线程安全和内存模型的关系

如何排查？

执行jstack命令，查看线程是否一致处于WAITING状态。



### Sychronized

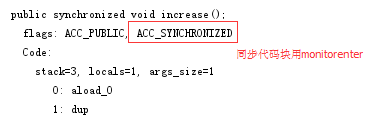
[Reference](http://blog.csdn.net/javazejian/article/details/72828483)

实现线程的互斥串行操作。保证接口的原子性和可见性

synchronized修饰实例方法

此处锁==synchronized(this)





monitorenter/monitorexit

注意synchronized不能保证修饰的方法块内指令的顺序

### ThreadLocal

#### 实现原理

Code: ThreadLocal101

在多线程中，ThreadLocal为每个线程提供一份共享变量的的副本。类Thread有一个成员变量ThreadLocalMap。它的Key值为ThreadLocal对象，线程使用共享变量时，会new一个存在ThreadLocalMap中。

warning:

多线程中中，每个线程都new一个共享对象，如果是同一个对象还是有并发访问问题。

创建不同的ThreadLocal对象来存储多个变量；

ThreadLocal VS Thread同步机制

都是为了解决线程安全问题。对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，访问串行化，对象共享化。ThreadLocal采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

#### 使用场景

一般情况下，通过ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的。各个线程中访问的是不同的对象。

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

如：

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

Spring中使用

**ThreadLocal.set()**将新建对象的引用保存到各线程的ThreadLocalMap。



传递参数

将ThreadLocal声明为static，这样同一线程在任何地方都可以获取set的值，

#### 内存泄漏问题

### Concurrent包

JDK5将Doug Lea的并发库引入到Java标准库中，提供新的启动(submit)、调度、管理线程的一大堆API了。增强了并发功能，简化了代码。主要特性：

* ReentrantLock
* 线程池
* 新增并发集合类ConcurrentHashMap，ConcurrentLinkedQueue等
* Semaphore

#### 线程池

实现原理：

[Reference](https://blog.csdn.net/cjh94520/article/details/70545202)

1. ThreadPoolExecutor 将线程包装成worker,



2. 有条件的进行着死循环，从而可以不断接受任务来进行。



3. 执行任务时直接调用线程的run方法。

可见池内线程可以执行不同任务。

线程池作用

* 减少在创建和销毁线程上花的时间
* 控制系统资源的开销。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

四种线程池

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

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(submit)、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。



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.

**DB Pool源码分析**

[Reference](https://blog.csdn.net/shuaihj/article/details/14223015)

#### ReentrantLock

实现互斥串行操作的

ReentrantLock VS synchronized

* Synchronized的锁是JVM层面上实现的,是原生语法层面。ReentrantLock是API层面的互斥锁（用起来会复杂一些，需要写控制锁的代码，加锁和解锁都需要显式写出）。
* ReentrantLock更灵活、更强大，增加了锁轮训、超时(tryLock[time])、中断(lockInterruptibly)等高级功能。
* synchronized采用悲观锁，而Lock用的是乐观锁方式。

实现原理

[Reference](https://www.cnblogs.com/maypattis/p/6403682.html)

**ReentrantLock源码**



可见：默认为非公平锁

内部类继承关系



**AQS**

AbstractQueuedSynchronizer简称AQS，是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题。



lock&unlock

NonfairSync



compareAndSetState ：CAS操作判断当前锁是否被占用

setExclusiveOwnerThread：设置当前占用该锁的线程。

**acquire**



**tryAcquire：**



**acquireQueued**

把已经追加到队列的线程节点（addWaiter方法返回值）进行阻塞



最终队列可能会如下图所示



 线程B和C都已经入队，并且都被挂起。当线程A释放锁的时候，就会去唤醒线程B去获取锁。

**addWaiter**

通过CAS把当前线程追加到队尾



selfInterrupt: Thread.interrupted()

#### 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。

#### BlockingQueue

定时线程的使用

#### 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。

AtomicReference

## 1.6 JVM

**谈谈你对JVM的理解？**

Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。Java源文件经编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。

JVM执行程序过程 ：加载class文件 🡪管理并分配内存(生成对象) 🡪执行垃圾收集

优化

JVM的编译优化

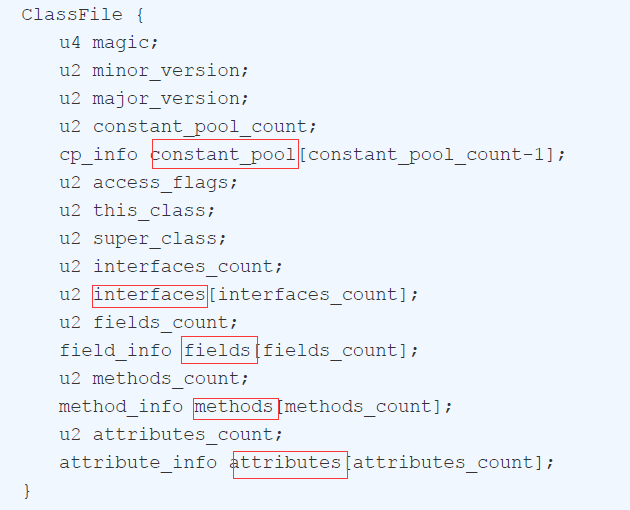
指令重排序，内存栅栏等

### Class文件格式

#### ClassFile结构

[Reference](http://blog.csdn.net/ns_code/article/details/17675609)1

[Reference2](https://blog.csdn.net/zhangjg_blog/article/details/22432599)



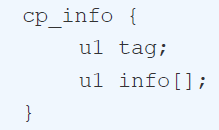
* 包含了类的字段、方法、接口、版本等描述信息外，还有constant\_pool.

Note

u1，u2和u4，分别代表了1、2和4个字节的无符号数

#### constant\_pool

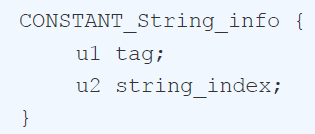
通用格式



tag项说明

|  |  |
| --- | --- |
| **常量类型** | **值** |
| CONSTANT\_Class | 7 |
| CONSTANT\_Fieldref | 9 |
| CONSTANT\_String | 8 |
| CONSTANT\_Integer | 3 |

**CONSTANT\_String\_info**



这里指Class文件中的常量池，用于存储编译期就生成的常量、符号引用。



**运行时的常量池**

存在于方法区，具有动态性。类加载的解析阶段会将符号引用转化为直接引用。

符号引用 VS 直接引用

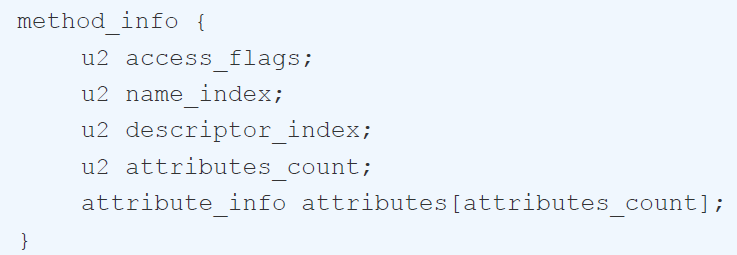
符号引用：编译后的class文件用字符串表示某个变量、接口的位置

直接引用：符号引用翻译出来的地址，**类加载阶段会将符号引用转化为直接引用。**

#### field\_info

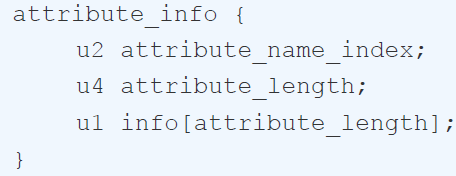
成员变量信息。field的操作会加入构造器中，不管引用类型还是基本类型,都用putfileld/getfield。

#### method\_info



规范定义的method\_info结构中，属性表可出现的成员有Code, Exception等

#### attribute\_info



### 运行时数据区



#### 寄存器

即程序计数器，保存当前执行指令的地址，线程私有(每个线程都有自己的程序计数器)。

#### Stack

每条线程都有自己私有的虚拟机栈, 这个栈与线程同时创建，用于存储栈帧（Frames）。

线程工作内存(高速缓存)，存放基本型，以及对象引用。

Frames

栈帧随方法调用而创建，随方法结束而销毁。和方法一一对应。

Frames

|  |
| --- |
| 局部变量表 |
| 操作数栈 |
| 动态链接 |
| 方法返回地址 |
| 方法调用 |

局部变量表和操作数栈编译期确定，通过method\_info的code属性保存，供栈帧使用。

**局部变量表**

存放方法参数和方法内的局部变量。以变量槽（slot）为单位，slot为32位。

**操作数栈**

编译后中产生的操作数压入。

**动态链接**

指向运行时常量池中该栈帧所属方法的引用,持有这个引用是为了支持方法调用过程中的动态连接.

方法调用：调用方法的reference

#### Heap

堆是可供各条线程共享的运行时内存区域，也是供所有类实例和数组对象分配内存的区。

存储对象实例，原则上所有的对象都在堆区上分配内存。



* Eden区：伊甸(yī diàn)园(传说夏娃和亚当偷吃禁果的地方，用来表示对象内存首次分配的区域)。绝大多数刚创建的对象会被分配在Eden区（大对象可以直接被创建在老年代）。Eden是连续的内存空间，因此在其上分配内存极快。
* Survivor: Eden区分配的多数对象很快就会消亡，Eden区满的时候，执行Young GC，将消亡的对象清理掉，并将剩余的对象复制到空闲的Survivor。回收过程中，总是有一块 Survivor 区域是空闲的。
* Old区：Young GC过程中, 在两个Survivor区多次切换（HotSpot虚拟机默认15次）的长生命周期对象会将被复制到老年代。

#### 方法区

很少GC，又叫永久区(Permanent)。Java8开始，取而代之的是Metaspace元数据空间。

用于存储类信息（field、方法、接口、版本等信息）、类变量、final常量、编译器的字节码等

#### metaspace

**TODO**

[Reference](http://ifeve.com/java-permgen-removed/)

aa

#### 运行时常量池

aa

#### 本地方法栈

aa

#### 直接内存（Direct Memory）

直接内存并不是JVM管理的内存，是JVM以外的机器内存。JDK中有一种基于通道（Channel）和缓冲区（Buffer）的内存分配方式，将由C语言实现的native函数库分配在直接内存中，用存储在JVM堆中的DirectByteBuffer来引用。由于直接内存收到本机器内存的限制，所以也可能出现OutOfMemoryError的异常。

#### Cache

cache的意义

缓解CPU和内存之间速度的不匹配问题。

原理

如果CPU需要的内容在cache里

局部性原理。/2-8理论

* 时间局部性：如果某个数据被访问，那么在不久的将来它很可能被再次访问；
* 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

cache的写回方式

cache的写操作方式可以追溯到大学教程《计算机组成原理》一书。

* write through（写通）：每次CPU修改了cache中的内容，立即更新到内存。每次CPU写共享数据，都会导致总线事务，导致总线事务的竞争，效率非常低。
* write back（写回）：每次CPU修改了cache中的数据，不会立即更新到内存，而是等到cache line在某一个必须或合适的时机才会更新到内存中；

无论是写通还是写回，在多线程环境下都需要处理缓存cache一致性问题。

线程安全

从cache的角度，产生线程不安全的原因：

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题

缓存一致性

1）通过在总线加LOCK#锁的方式

2）通过缓存一致性协议，比如MESI协议

MESI协议

处理缓存cache一致性问题。

### 1.6.3 类加载机制

#### classloader

#### 类加载过程

类加载过程即类的初始化（整个过程，不单指第三阶段那个初始化）的过程，是Class对象自身的实例化过程。

注意：这里的加载指类的初始化过程，和类是否实例化无关。

[Reference](https://blog.csdn.net/ns_code/article/details/17881581)

类加载分了三个阶段，loading，linking和initializing。

1.加载

1.加载Class编译后的二进制字节流到方法区

2.//方法区初始化静态变量

3.在堆中创建一个 Class 类**对象**。通过该对象访问方法区中的这些数据。

2.链接

验证--准备--解析

**准备**

* 为类变量**分配内存**（方法区）空间并设置默认值。
* 不包括实例变量。实例变量会在对象实例化时随着对象一块分配在堆中。
* 注意：public static i=1;此阶段i为0而不是1。而把value赋值为3的putstatic指令是在程序编译后，存放于类构造器<clinit>（）方法之中的，所以真正赋值要初始化阶段，调用putstatic

**解析**

* 解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。
* 解析可能在Initializing之前，也可能在之后这是为了支持 Java 语言的运行时绑定（也成为动态绑定或晚期绑定）

3.初始化

 初始化是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的代码**。**

从另一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。

**<clinit>**

Code: ClassClinit100

编译器自动收集，类变量的赋值语句和静态语句块中的语句合并产生。收集顺序是由源文件中出现的顺序所决定的。如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成此方法。



**<clinit> VS <init>**

两种都是编译器生成的方法。

**<clinit>：**是类（或接口）构造器，在类加载的初始化阶段对类进行初始化，类变量内存分配在方法区。

**<init>：**实例构造器。

<init>加入到构造器第一句执行



#### 触发条件

* JVM执行指令new/getstatic/putstatic/invokestatic/。
* 类反射。
* 当初始化一个类的时候，如果其父类还没有进行初始化，则需要先触发其父类的初始化。
* 当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类。

具体Case分析

1.类的初始化后并未类的实例生成，比如getstatic….



2.单例中类Singleton初始化阶段会生成对象instance

这里生成对象instance和Singleton的初始化顺序？

Code: Singleton101



几个阶段是按顺序**开始**，而不一定按顺序**完成**，通常都是交叉进行的。这里<clinit>执行过程激活new的执行。

**Q:对于成员变量是Object obj=new Object（）类加载过程？**

obj的引用在哪个阶段分配内存？

#### 双亲委派（破坏双亲委派）

#### 模块化

（jboss modules、osgi、jigsaw）

### 1.6.4 对象内存管理

#### 对象结构

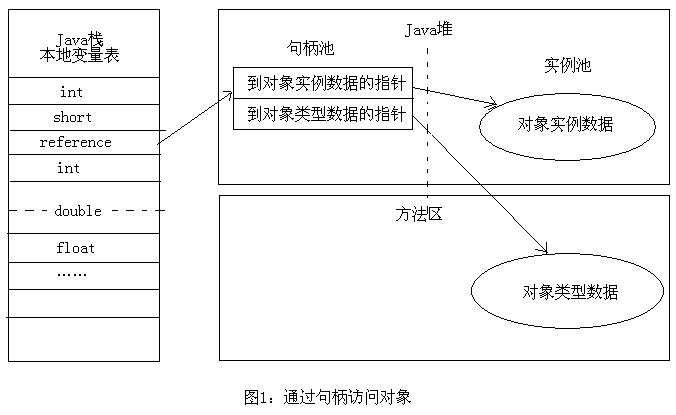
Java虚拟机规范不强制规定对象的内部结构应当如何表示 。

TODO:

#### 访问方式

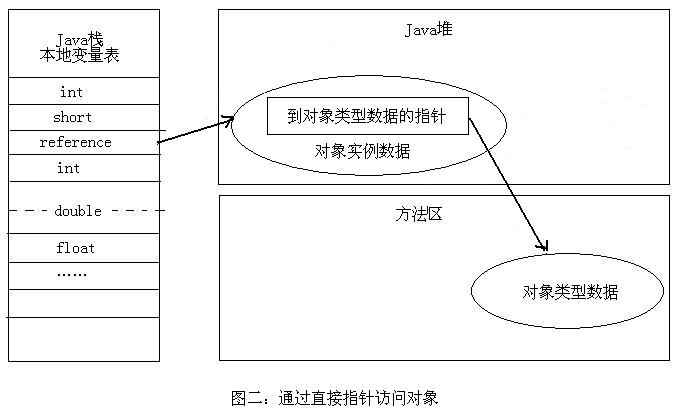
对象在不同内存中的有的访问方式，不同VM实现方式不同，主流两种：

使用句柄



相比直接引用多一个指向对象实例的指针。

直接指针



new过程：JVM加载类🡺为对象分配内存🡪 调用构造函数初始化成员变量(<init>)

#### 内存分配

[Reference](http://www.cnblogs.com/zhguang/p/3257367.html#access)

Java内存分配和回收的机制概括的说就是：分代分配，分代回收。

根据对象存活的时间分为：年轻代、年老代、永久代(方法区）,详细见见下文。。。

**Q:java虚拟机堆何时分配? 存放哪些数据? 何时回收？**

**Q: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?**

#### 垃圾收集器GC

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

概括地说，GC对对象的内存进行标记，并确定哪些内存需要回收。根据一定的回收策略，自动的回收。保证JVM中的内存空间，防止出现内存泄露和溢出问题。

学习Java GC机制，可以帮助我们在日常工作中排查各种内存溢出或泄露问题，解决性能瓶颈，达到更高的并发量，写出更高效的程序。

Java 中的堆也是 GC 收集垃圾的主要区域。

对象存活的判定

**Q:谁会被GC，什么时候GC,如果想不被GC怎么办?**

目前虚拟机基本都是采用可达性算法

**引用计数器**

每个对象有一个引用计数器，当对象被引用一次则计数器加1，当对象引用失效一次则计数器减1，对于计数器为0的对象意味着是垃圾对象，可以被GC回收。

* 优点：1.实时回收 2.回收应用无需挂起3.回收具有区域性，不用扫描全部对象。
* 缺点 ：1.无法解决循环引用问题2. 浪费资源，即使内存够用，仍然在计数统计

**可达性算法**

GC Roots可达的对象便都是存活对象，不可到达的对象便是垃圾回收的对象，随时可被GC回收。 需要遍历整个GC根节点来判断是否回收。

可作为GC Roots的对象：

* 栈的局部变量表所引用的对象；
* 本地方法栈的JNI所引用的对象；
* 方法区的静态变量和常量所引用的对象；



对象实例1、2、4、6都具有GC Roots可达性，也就是存活对象，不能被GC回收的对象，而对于对象实例3、5直接虽然连通，但并没有任何一个GC Roots与之相连，这便是GC Roots不可达的对象，这就是GC需要回收的垃圾对象。

垃圾回收算法

GC 分为Partial GC（Young GC）和Full GC。

**GC回收策略**

Mark-Sweep

标记-清除回收算法

1. 标记和清除过程的效率都不高 2.空间碎片太多。常用可达性算法。



Copying

复制收集算法

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



**Young GC**

* GC触发条件**：** Eden区分配满的时候触发。
* 回收过程： 在Eden区创建对象 🡪Eden区处理消亡对象 🡪复制剩余对象到Survivor 🡪 清理出一块空闲的Survivor 🡪复制长周期对象到老年代

**Old GC**

只收集old gen的GC。只有CMS的concurrent collection是这个模式。

**Full GC**

* 触发条件： old gen剩余的空间不够用。
* 回收范围：对整个新生代、老生代、元空间（metaspace）的全局范围的GC；。

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

4. 介绍GC 和GC Root不正常引用。

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

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

深入分析了Classloader，双亲委派机制

#### 内存溢出&泄漏

**内存溢出**

申请了10个字节的空间，写入11或以上字节的数据，就会溢出。

java.lang.OutOfMemoryError，一般都是程序问题或者JVM参数配置问题引起

* Java heap space:
* PermGen space ：1）应用中很多class，web服务器对JSP进行pre compile时2）webapp加载大量的第三方jar
* unable to create new native thread

Memory leak

[Reference](http://blog.csdn.net/dongnan591172113/article/details/79044678)

对象申请的内存不再使用，因为仍被引用着，GC无法移除，导致内存泄漏。

内存泄漏可能导致内存溢出。



Happen Case

**1. 长生命周期的对象持有短生命周期对象的引用**

静态集合类的生命周期跟应用程序一样长，有可能会发生内存泄漏，看下面代码：



集合类如果是局部变量，在方法栈退出后就没有引用了会被jvm正常回收，不会造成内存泄露。而如果是全局性的变量提供这样的删除机制或者定期清除策略非常必要。

Java集合类已经提供了删除机制



**2.集合里的对象属性被改变**



**3.监听器**

在Java中，我们经常会使用到监听器，如对某个控件添加单击监听器addOnClickListener()，但往往释放对象的时候会忘记删除监听器

**4.各种连接**

Java中的连接包括数据库连接、网络连接和io连接，如果没有显式调用其close()方法，是不会自动关闭的，这些连接就不能被GC回收而导致内存泄漏。一般情况下，在try代码块里创建连接，在finally里释放连接，就能够避免此类内存泄漏。

**5.外部模块的引用**

调用外部模块的时候，方法传入对象，也应该注意防止内存泄漏。如：

public void register(Object A)

这个方法传入对象A模块的引用，这时需要B模块提供了去除引用的方法，如unregister()。这种情况容易忽略，而且发生了内存泄漏的话，比较难察觉，应该在编写代码过程中就应该注意此类问题。

**6.单例模式**

因为单例对象初始化后将在JVM的整个生命周期内存在，如果它持有一个外部对象（生命周期比较短）的引用，那么这个外部对象就不能被回收，而导致内存泄漏。如果这个外部对象还持有其它对象的引用，那么内存泄漏会更严重，因此需要特别注意此类情况。

StackOverflowError（堆栈溢出）

请求的栈的深度大于虚拟机所允许的深度

同步性

类加载由JVM自动同步，而执行对象的构造器可以中断的并发原子操作

#### JAVA OPS优化

**Heap**

* -Xms heap的最小内存；
* -Xmx heap的最大内存；
* -Xmn young的大小，一般设置为Xmx的3、4分之一
* -XX:NewSize: young的大小
* –XX:MaxNewSize
* NewRatio: young:old,2表示1:2,即young/heap=1/5
* -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: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

**方法区Perm**

* -XX:PermSize=5M -XX:MaxPermSize=7M

**32G 内存配置示例：**

-Xms10g -Xmx10g -Xmn1024m -Xms10g -Xmx10g -Xmn1024m -Xshare:off

**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

**JVM调试方法：**

1.Jconsole查看内存各部分变化

2.gc 用jstat

### 1.6.5 VM 指令集

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51832690)

常量🡪栈顶。

|  |  |  |
| --- | --- | --- |
| iconst\_1 | 将int型的1推送至栈顶 | 4: iconst\_1 |
| fconst\_1 | 将float型的1推送至栈顶 |  |
| bipush | 将单字节(push byte)的常量值(-128~127)推送至栈顶 | 9: bipush 10 |
| sipush | 将一个短整型常量值(-32768~32767)推送至栈顶 |  |
| ldc | 超过32767。或者String常量 | 15: ldc #5 |

栈顶 ⬄ 本地变量

|  |  |  |
| --- | --- | --- |
| istore\_1 | 将栈顶int型数值存入第二个本地变量 |  |
| astore | 将栈顶**引用型数值**存入指定本地变量 |  |
| astore\_0 | 将栈顶引用型数值存入第一个本地变量 |  |
| astore | 将指定的引用类型本地变量推送至栈顶 |  |
| iload\_1 | 将第二个int型本地变量推送至栈顶 |  |
| aload\_0 | 将第一个引用类型本地变量推送至栈顶 |  |

pop

对栈顶进行操作

|  |  |  |
| --- | --- | --- |
| pop | 将栈顶数值弹出 |  |
| dup | 复制栈顶数值并将复制值压入栈顶 |  |

**invoke**

|  |  |  |
| --- | --- | --- |
| invokevirtual | 调用实例方法 |  |
| invokestatic | 调用静态方法 |  |
| invokeinterface | 调用接口方法 | Map.put |
| invokespecial | 调用超类构造方法/实例初始化方法/私有方法 |  |

5: invokespecial #3 // Method "<init>":(I)V

4: invokespecial #3 // Method "<init>":()V

域操作

成员变量

|  |  |  |
| --- | --- | --- |
| getstatic | 获取静态域，并将其值压入栈顶 |  |
| putstatic | 用栈顶的值为指定的类的静态域赋值 |  |
| getfiled | 获取实例域，并将其值压入栈顶 |  |
| putfileld | 用栈顶的值为指定实例域赋值 |  |

运算

|  |  |  |
| --- | --- | --- |
| iadd | 将栈顶两int型数值相加并将结果压入栈顶 |  |
| ishl | 将int型数值左移位指定位数并将结果压入栈顶 |  |

new及数组

|  |  |  |
| --- | --- | --- |
| new | 创建一个对象，并将其引用值压入栈顶 |  |
|  | 将int型数值左移位指定位数并将结果压入栈顶 |  |

javap

[Reference](http://blog.csdn.net/qq_24489717/article/details/53837493)

* -c 输出编译后的JVM指令



* -verbose 输出常量池，栈大小，方法参数。
* -l 输出行和变量的表。
* -public 只输出public方法和域
* -protected 只输出public和protected类和成员
* -package 只输出包，public和protected类和成员，这是默认的
* -p -private 输出所有类和成员
* -s 输出内部类型签名
* -constants 输出静态final常量
* 不加参数:只显示方法名
* -help 帮助

实例：

**int** a1=10;**int** a2=10;**int** a3=a1+a2;编译后的字节码



可见

变量赋值过程：数值通过命令的操作数push到栈顶->store到变量

Integer a1=10;



这里astore传的是引用！

局部变量。bipush/ldc等。

[Reference](https://blog.csdn.net/zhangjg_blog/article/details/22432599)

包含LineNumberTable和LocalVariableTable



LineNumberTable：描述了一条字节码和源码行号的对应关系

LocalVariableTable：局部变量与源代码中的局部变量之间的对应关系

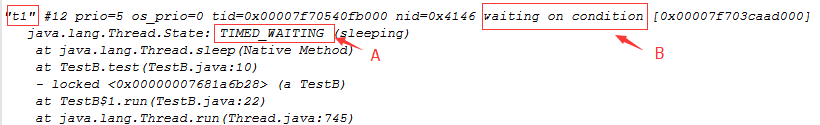
### 1.6.6 JTools

很多时候在线上出现问题时，往往会伴随着某些指标的异常。大部分情况下，在问题发生之前，某些指标就会提前有异常显示。

#### jstack

[Reference](http://www.hollischuang.com/archives/110)

生成线程快照。用法：jstack <pid>



线程状态

A处为线程状态。NEW/RUNNABLE/BLOCKED/WATING...

调用修饰

表示线程在方法调用时,额外的重要的操作。线程Dump分析的重要信息。

**locked**



申请对象锁成功才会出现此句。用法：locked <地址>

**waiting to lock**



申请对象锁未成功，t2没有locked语句，只有waiting to lock。线程状态：BLOCKED

**waiting on**



申请对象锁成功后,释放锁进入等待区。线程状态：WAITING

parking to wait for <地址> 目标

主要目的：定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。

**Thread State**

Wait on condition

死锁

http://www.hollischuang.com/archives/2409

http://www.hollischuang.com/archives/110

#### 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)

#### jps

显示java进程的pid



#### jconsole

配置

JAVA\_OPTS=“-Djava.rmi.server.hostname=192.168.1.211 -Dcom.sun.management.jmxremote.port=9527 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false

Total blocked

Total waited:

**state**

Wait on condition

#### Jvisual

JDK自带性能检测工具



**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

## 1.8 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

### Java发展

JDK 1.1

1997年发布。

* 反射
* JDBC；
* 支持内部类；
* 引入Java Bean；
* 引入RMI（Remote Method Invocation）；

JDK 1.2

1998年发布

* Collection
* 引入Java 插件；
* 对字符串常量做内存映射；
* 引入JIT（Just In Time）编译器；
* 引入对打包的Java文件进行数字签名；
* 引入控制授权访问系统资源的策略工具；
* 引入JFC（Java Foundation Classes），包括Swing 1.0、拖放和Java 2D类库；

JDK 1.4

2002年发布

* NIO
* XML处理；
* Java打印服务；
* 引入Logging API；
* 引入Java Web Start；
* 引入断言Assert；

引入Preferences API；

* 引入链式异常处理；
* 支持IPv6；
* 支持正则表达式；
* 引入Image I/O slot machine API。

JDK 5

2004年发布

[Reference](https://sakuraffy.github.io/java_jdk5.0/)

* 泛型
* concurrent包
* 枚举
* 元数据（Metadata）
* foreach循环
* 自动拆箱装箱
* 静态导入
* 可变参数 add(int... arr)
* 内省

JDK6

2006年发布

* 支持脚本语言

JDK7

2011年发布

* 支持动态语言
* 新的文件I/O库
* Lambda的实现

JDK 8

2014年发布

* Lambda表达式
* Pipelines和Streams
* Date和Time API
* Default方法
* Type注解
* Nashhorn JavaScript引擎
* 并发计数器
* Parallel操作
* 移除PermGen Error
* TLS SNIJava 8的内存分代改进

### 多态

实现原理

TODO: invokevirtual执行过程

[Reference](https://blog.csdn.net/qq_35614059/article/details/79141866)

### 泛型

# 第二篇 J2EE

## JAVA Web

### Servlet

#### 生命周期

实例化🡺init🡺service🡺destroy。

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

Security

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

权限

Oauth

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，而非传递参数。

登录模块

集群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

### 2.1.2 MVC

* 分层好处

1. 有利于分工
2. 复用
3. 好扩展，好维护

* 应用分层



1. **终端显示层**：PC端，移动端展示等。
2. **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； 需要进行网关安全控制、流量控制等。
3. **Web 层**：即Controller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。
4. **Service层**：相对具体的业务逻辑服务层。
5. **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；

1. 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。

1. **DAO 层**：数据访问层，与底层 MySQL、Oracle、HBase 等进行数据交互。

7.**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。

* 常规校验

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。

**参数的安全性判断应该写在controller还是service？异常处理尼？**

哪一层都是很有必要的，一般在controller进行，在service层进行校验更好的复用。

* 统一异常处理

1．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

2．在web层设置统一异常处理机制

3．记录日志时，尽可能带上参数信息，便于调试

4．开放接口层要将异常处理成错误码和错误信息方式返回。页面要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

DAO层和Service层，产生的异常类型有很多，无法用细粒度的异常进行 catch，避免直接throw new Exception,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。

* 菜单递归查找



### Spring

#### 好处

**IoC**：方便解耦，简化开发。通过Spring的IoC容器，用户可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度耦合。有了Spring，用户不必再为单实例模式类、属性文件解析等这些底层的需求编写代码，可以更专注于上层的应用。

**AOP**：方便进行面向切面的编程，很多传统OOP不容易实现的功能可以通过AOP轻松应对。

**声明式事务**：在Spring中，用户可以从单调烦闷的事务管理代码中解脱出来，通过声明的方式灵活地进行事务管理，提高开发效率和质量。

**方便程序的测试**：可以用非容器依赖的编程方式进行几乎所有的测试工作。在Spring里，测试不再是昂贵的操作，而是随手可做的事情。

**方便集成各种优秀的框架**，降低各种框架的使用难度。

**降低Java EE API的使用难度**：spring对j2ee API(如JDBC、远程调用)封装

**Java源码是经典的学习范例**。spring源码是java技术的最佳实践，值得研究学习。

#### Bean

生命周期

[Reference](https://www.zhihu.com/question/38597960)



作用域

singleton：生命周期同Spring Container，在容器中只有一个实例。

prototype：每次调用都会得到一个新的实例。

request, session, global session

以上三类是只有引入了Spring ApplicationContext才存在，生命周期大致对应HTTP中的Request, Session和Application。

#### MVC

**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

Object obj=new Object();

如何对new做AOP？

日志

#### 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。

spring ejb区别

### 2.1.4 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



### ibatis

### DB连接池

建立连接是一个费时的操作，频繁的进行数据库连接操作势必占用很多的系统资源，影响网站的响应速度。

如果查询完数据库后，不关闭连接，而是暂时存放起来，当别人使用时，把这个连接给他们使用。就避免了一次建立数据库连接和断开的操作时间消耗。

基本思想

为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去

## 2.4 网络通信

### 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(**指读写数据部分，连接还是one request on Thread**)。
* 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。

JAVA IO是同步非阻塞IO，同步和异步说的是消息的通知机制，阻塞非阻塞说的是线程的状态。

### 长连接&心跳机制

心跳要解决的问题：监控无效的连接并断开，多次超时无反应就断开的方式处理。

 1）TCP协议自带的keep-alive功能,打开即可



当keepAlive为true，此处服务端每隔一段时间向客户端发送一个数据包检查客户端的活动状态，如没response会将socket关闭。

2）应用层面实现心跳机制

自定义心跳消息头. 一般客户端主动发送, 服务器接收后进行回应(也可以不回应)

3）框架自带

MINA本身提供了一个过滤器类[KeepAliveFilter](http://mina.apache.org/mina-project/apidocs/org/apache/mina/filter/keepalive/KeepAliveFilter.html)  ,该过滤器用于在IO空闲的时候发送并且反馈心跳包

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。





## 2.5 远程调用

### Web service

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

聚合和发布日志内容而设计的，统一格式——每个条目都包括title，entry，id，和content/summary——这使得我能从一条Atom feed中得到有用的信息，而不管   这条feed来自哪一类型的应用。

### Dubbo

TODO

### 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。

## 2.6 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

引入消息中间件是用来降低应用之间的耦合的。

对于上游应用来说，它往往会依赖非常多的下游应用，如果全部用同步调用的方式，那么上游应用必须等到所有下游应用全部执行成功后，才能返回成功，这种等待往往是不可忍受的。为此，需要将同步异步化，而消息中间件可以实现这种异步化。

使用消息中间件后，上游应用只需要向消息服务器发送消息就可以了，它不用等待所有应用执行完，它可以提前返回执行成功，后续，消息服务器会去向这些下游应用发消息，通知它们执行。

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

**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

## 2.7分布式服务

分布式系统往往是把应用拆分成多个应用，每个团队维护一个应用，应用与应用通过远程过程调用或者消息中间件通信。这种系统的优点是能够做到高内聚低耦合，可以支撑业务的快速发展。

对于分布式系统中的某个应用来说，它的入口往往有

**三个入口：**远程调用，消息中间件，web访问

**出口则有很多**

* 远程过程调用：
* 消息中间件
* 访问分布式缓存
* 访问分布式日志
* 访问数据库
* 执行调度任务

分布式 VS 集群

分布式：一个业务分拆多出个子业务，部署在不同的服务器上

集群：同一个业务，部署在多个服务器上

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。

涉及的技术

* 远程服务框架

远程服务框架是分布式系统中不可或缺的，对于远程服务框架来说，关键的几个点是：

服务消费者、服务提供者、协议（java序列号、反序列化、hessian协议等等）

网络（NIO、多路复用等等）

地址列表

目前业界比较有名的开源框架有：Apache Thrift（跨语言）、Dubbo等等。

* 消息中间件

引入消息中间件是用来降低应用之间的耦合的。对于上游应用来说，它往往会依赖非常多的下游应用，如果全部用同步调用的方式，那么上游应用必须等到所有下游应用全部执行成功后，才能返回成功，这种等待往往是不可忍受的。为此，需要将同步异步化，而消息中间件可以实现这种异步化。

使用消息中间件后，上游应用只需要向消息服务器发送消息就可以了，它不用等待所有应用执行完，消息服务器会去向这些下游应用发消息，通知它们执行。

总结:发送异步消息，使上游应用不必等下游应用执行完，降低应用之间的耦合的。

* wrapper层

wrapper层主要是上面两个入口的包装层，进行一些异常管理，日志管理和AOP拦截

* 服务管理和配置管理

业务逻辑层：流程引擎和规则引擎，配合spring来治理极易变动的业务逻辑，后续有空单独写个流程引擎

负载均衡器单点故障

实时管理集群

分布式事务

### SOA

传统分布式按功能模块把应用拆分成多个应用。缺点：当垂直应用越来越多，模块间相互交叉调用越来越复杂，模块耦合度太高，扩展性差，不好维护和部署

SOA对系统水平拆分，将核心业务抽取出来，作为独立的服务,降低模块之间耦合度，好扩展，方便部署。

总结：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?

### Memcached

并发的处理

1.单个命令是原子操作。命令将被串行化、先后执行。

2.命令序列不是原子的。如果您通过get命令获取了一个item，修改了它，然后想把它set回Memcached，这个item没有被其他进程操作过。在并发的情况下，可能覆写其他进程set的item。

Memcached 1.2.5以及更高版本，提供了gets和cas命令，它们可以解决上面的问题。

JAVA客户端

1.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。

2.XMemcached

**分布式缓存**

cas单点登录

Zookeeper服务的注册和发现

Zookeeper配置中心

### 搜索引擎

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

### SpringCloud

### 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

## 2.8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

# 第三篇 系统架构

## 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

1. abstract class和interface最大的区别在于设计理念不同。其实abstract class表示的是"is-a"关系，是一种继承关系。interface表示的是"like-a"关系。一个类只能使用一次继承关系而一个类却可以实现多个interface。

2.语法上的区别，abstract class 可以有自己的数据成员，也可以有非abstract的成员方法

## 3.2 设计模式

#### 单例模式

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

**非同步**



**延迟加载**



**Double Checked locking**



* 1处volatile：new Singleton并非原子操作，instance为类变量，准备阶段

1.为什么加volatile



1.new给instance分配内存

2.invokespecial初始化对象

3.aload将instance指向分配的内存

JVM 的即时编译器中存在指令重排序的优化。最终的执行顺序可能是 1-2-3 也可能是 1-3-2

2处为什么判空?

防止阻塞Thread恢复后会多次new

#### 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

#### 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 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。

## 3.5 Design Tools

### UML图

1）实现



2）继承



3）依赖和关联

依赖和关联都是使用到另一个类，关联是一种更强的依赖关系。

* **依赖**：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;
* **关联**：成员变量



关联分为组合/聚合，区别在于存在的生命周期不同。

**聚合(aggregation):**



**组合(Composition)**

Room是house组成部分，有相同的生命周期。



# 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

## MySQL

安装[Reference](http://www.cnblogs.com/bookwed/p/5896619.html)

### 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。

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）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)

#### 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。

### 4.1.5 主从分离架构



### 4.1.6 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**

## 4.7 DB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、读写比例、并发量、数据量和数据增长数据。

* 读写比例🡺根据具体业务读写比例，降低主库压力
* 高并发🡺集群
* 数据是否需要强一致🡺NOSQL
* 数据量，数据增长快。

1.针对数量大的表进行历史表分离（如交易流水表）。

2.做分表分库，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，不要在创建表时就分库分表

1）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

2）查询优化

查询性能低下最基本原因是访问数据太多，确认应用服务检索必要的数据列或和数据行。

1.关联查询优化

需要联表查询

* 连表时，保证被关联的字段需要有索引，一般在连表顺序第二个create index。
* 数据类型必须绝对一致
* 考虑对相对应的查询条件做索引。加快查询速度。
* 一般联表消耗更多系统资源，还可能使索引失效，超过三个表禁止join。

分解关联查询

用单表还是联表，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。(好处),在高并发高读环境，遇到性能瓶颈时，将联表分解成单表冗余字段的形式，这样做有几个好处

* 好做缓存
* 建覆盖索引，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。
* dao层的复用率更好，更好维护，后期好扩展。

2.分析执行计划

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，避免全表扫描。 对于group by 或count/一些 会引起全表扫描，必要时使用单独的汇总表或者放Memcached。

**3）索引优化**

索引增加写数据的额外消耗，频繁写的表不宜建索引。

4）缓存

* 缓存重复查询和获取比较慢的数据。

5）搜索

页面搜索严禁左模糊或者全模糊，如果需要走搜索引擎来解决。

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。

扫描行数和访问类型

ALL-index-range-ref

# 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

配置文件server.xml

**maxThreads**：最大的线程数。Tomcat7默认值200。

**acceptCount**： 所有可用的线程数都被使用时，可以放到处理队列中的请求数。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 Linux系统资源分析

### top

[Reference](https://www.cnblogs.com/dragonsuc/p/5512797.html)

**参数：**

-p 监控某个进程

-H show thread, top –Hp pid

**快捷键**

P:CPU排序。默认按cpu占用量排序。

M:内存排序

H: show thread

1:show cups



4 users：当前登录user数

Tasks:进程数。-H切换成线程数。

**Load Average**

一段时间 (1 分钟、5分钟、15分钟) 内平均 Load

正常范围：

* 当系统负荷持续大于0.7，必须开始调查了，问题出在哪里，防止情况恶化。
* 当系统负荷持续大于1.0，必须动手寻找解决办法，把这个值降下来。
* 当系统负荷达到5.0，就表明系统有很严重的问题，长时间没有响应，或者接近死机了。

以上指标基于单CPU的，四核机器的负载最好保持在3(4\*0.7 = 2.8)以下。

**监控状态**



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

%CPU  CPU时间占用百分比

%MEM  进程使用的物理内存百分比

TIME+  进程使用的CPU时间总计，单位1/100秒  
 CMD 进程名称（命令名/命令行）

**命令补充**

监控最小单位是进程，看不到java线程和客户连接数，通常我用ps和netstate两个命令来补充top的不足。

### free

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



-/+ 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

-i 查看inodes

df –hl

df -h /home/nadong

### du

Disk usage of each File

du –sh

分区

### ps

Process Status

* 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

-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

### 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

### 文件属性

文件颜色

绿色文件---------- 可执行文件，可执行的程序

红色文件-----------压缩文件或者包文件

蓝色文件----------目录

白色文件----------一般性文件，如文本文件，配置文件，[源码](https://www.2cto.com/ym/)文件等

浅蓝色文件----------链接文件，主要是使用ln命令建立的文件



链接文件

分配不同权限

缩写路径

软链接 VS 硬链接

软链接和Windows的快捷方式类似

Chmod

chmod u+w test

chmod ugo+r test;

### 文件搜索

find

find [dir] -name filename

1.在当前及子文件夹查找

find –name common.jar

2.在etc及子文件夹查找

find /etc -name httpd.conf

3.模糊查找

find –name ‘httpd\*'

Grep

-n ：输出行号

-r ：在某一目录下递归[所有子目录]查找某一字串:

grep -r ‘str\*' . (注意此处有点)

grep -r a /scratch

Whereis

whereis useradd

### 文件查看及编辑

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

lsof

### jar

-c 创建一个jar包  
-t 显示jar中的内容列表  
-x 解压jar包  
-u 添加文件到jar包中  
-f 指定jar包的文件名  
-v生成详细的报造，并输出至标准设备

1）解压

Jar -xvf test.ear

2）创建

Jar -cvf test.jar ./\*

3）更新

jar -uvf RESTWebService.ear APP-INF/lib/AdfHcmUsersRestModel.jar

4）查看

jar tvf test.jar

5）搜索：

jar tvf ZhcLib.jar | grep 'pattern'

搜索整个目录下的所有jar文件：

find . -name "\*.jar"|awk '{print "jar -tvf "$1}' | sh | grep ".\*.xml"

## 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

### 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

### mysql

#启动服务 service mysqld start  
#连接 ./mysql/bin/mysql -uroot  
DDL  
show databases/tables  
desc table\_name;

### 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

## SHELL

变量

$# 是传给脚本的参数个数

$0 是脚本本身的名字

$1 是传递给该shell脚本的第一个参数

$2 是传递给该shell脚本的第二个参数

$@ 是传给脚本的所有参数的列表

$\* 是以一个单字符串显示所有向脚本传递的参数，与位置变量不同，参数可超过9个

$$ 是脚本运行的当前进程ID号

$? 是显示最后命令的退出状态，0 = success, 其他 = error

${var}等价于$var（简写）

if/else

[Reference](http://www.cnblogs.com/kangyoung/p/3556173.html)

比较

if ([ condition ] || [condition])

then

cmd

else

cmd

fi

if method()

method return 0=true ,1=false

文件表达式

-e filename 如果 filename存在，则为真  
-d filename 如果 filename为目录，则为真   
-f filename 如果 filename为常规文件，则为真  
-L filename 如果 filename为符号链接，则为真  
-r filename 如果 filename可读，则为真   
-w filename 如果 filename可写，则为真   
-x filename 如果 filename可执行，则为真  
-s filename 如果文件长度不为0，则为真  
-h filename 如果文件是软链接，则为真

整数变量

eq:=

ne:!=

gt:>

ge:>=

lt: <

le <=

函数

LaunchJDeveloper(){

DisplayBanner

}

### expr

手工命令行计数器



模式匹配

# 第七篇 总结

## 7.1 系统优化

### 前端优化

* 合并css、js，减少Http请求
* 启用压缩，减少数据传输
* 使用浏览器缓存。生成新的js时，只需更新Html中引用。
* 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

### 应用服务器

系统架构

* 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。
* 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

* 多线程,最大发挥CPU和处理IO阻塞。
* 使用NIO,非阻塞，基于事件驱动
* 资源复用，比如单例和线程池。
* 合理的数据结构
* 减少垃圾回收。
* 尽量不使用锁,减小锁的力度。

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。