ldc压入栈后是否在heap中创建对象？

TODO：static 局部的static和类static存储位置。

啊啊

# 第一篇 J2SE

## Base Data Type

### String

TODO1:hostspot 加载string后是否在持久区创建对象？

## Collection

### 1.2.1 概述



**Collection VS Array**

数组为定长，Collection均可动态增长。

### 1.2.2 List

元素保持特定的顺序

ArrayList

基于数组实现，可动态的扩展和缩减。

1. **private** **transient** Object[] elementData;

排序：Comparator

### 1.2.3 Set

HashSet

元素不重复，无顺序。根据对象HashCode()返回值决定存储位置

LinkedHashSet

维护插入顺序，性能略低于HashSet

### 1.2.3 Map

每个元素为key-value形式

HashMap

底层是一个数组结构，数组中的每一项又是一个链表。非线程安全

碰撞冲突

hashMap数组每一项是一个链表，添加到对应位置链表最后一项的next。

HashMap冲突很厉害，最差性能，你会怎么解决?从O（n）提升到log（n）咯，用二叉排序树的思路说了一通

ConcurrentHashMap

线程安全且效率高

**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

### Queue

队列，保证先进先出。

### Tree

TODO1:TreeMap 内部红黑树排序

## IO

### 1.3.1 简介

NIO

NIO是一种同步非阻塞IO模型，基于IO多路复用（事件驱动），解决高并发环境的大量连接，有较高吞吐量。

优势

NIO的优势在于单个Thread可以同时处理多个网络连接的IO，避免创建大量的线程和线程切换造成资源的浪费。（原理：NIO优化了IO的读写, 最大化压榨CPU，线程一个时间片可以处理更多IO）

IO多路复用

多路指N个连接，复用指复用一个或少量线程。

同步 VS 异步

按照POSIX标准来划分只分为两类：同步IO和异步IO。主语是IO的操作。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。

**同步**：NIO调用后没有返回，在调用没有得到结果之前，并未返回。主动等待这个调用结果，一旦调用返回，就得到返回值了。

**异步**：调用后直接返回，由被调用者通知来通知调用者，或通过回调函数处理这个调用。

### 1.3.2 IO优化

BIO

特点：JDK1.4以前的唯一选择，程序直观简单。

原理：同步阻塞。

使用场景：连接数少且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中。

NIO

对于大部分的IO场景, 它都能适应. 但是, 由于它的阻塞性, 每一个流的读写都需要占用一个线程. 这意味着, 流IO的可伸缩性很差. NIO就是IO Multiplexing在Java中的实现。IO Multiplexing在系统级语言如C/C++中应用了很长时间. 使用IO Multiplexing, IO的伸缩性大大提高, 使用单个线程, 就可以处理大量的IO对象.

特点：JDK1.4开始支持.**有响应实时性的优势**

原理：同步非阻塞。需一个线程去轮询所有IO操作的状态改变。

使用场景：连接数目多且每个连接只发送少量数据，例如一个聊天服务器。编程比较复杂，

AIO

特点：编程比较复杂，JDK7开始支持.

原理：异步非阻塞。无需轮询，充分调用OS参与并发操作，由系统通知对应的线程来处理。

使用场景：连接数目多。比如相册服务器

TODO: 通常来说，异步操作会让性能的吞吐率有很大提升（Throughput），但是会牺牲系统的响应时间（latency）。这需要业务上支持。

理想状态CPU，内存，IO外设都满负荷工作，大部分情况瓶颈都在IO的读写部分，NIO是优化了IO的读写。NIO最大化压榨CPU，把时间片都更好利用起来。如果系统的瓶颈不在这里，那么BIO和NIO在性能上没有区别。

### 1.3.3 实践

TODO1 用netty+spring VS servlet +tomcat 比较

Netty&Mina

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。

NoSql/redies todo3 如何使用NIO的

TODO3 Reactor模式

TODO2:tomcat能实现多大的并发

## 1.4 Thread

### 1.5.1 多线程

多线程本质上是利用闲置的CUP资源，加快的处理速度。

#### 线程 VS 进程

进程是资源(CPU时间，内存等)分配的基本单位，线程是CPU调度的基本单位。

#### ****多线程 VS 单线程****

**多线程不一定就比单线程程序跑的快，**取决于应用场景及程序的设计，有时候可能引入多线程带来的性能提升抵不过多线程而引入的开销。

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** ： **上下文切换过程并不廉价**
* **内存同步**的开销
* 设计更复杂

#### 线程切换开销

[Reference](http://ifeve.com/java-context-switch/)

上下文切换会带来直接和间接两种性能开销

**直接消耗**: CPU寄存器需要保存和加载, 系统调度器的代码需要执行, 刷新TLB(TLB：translation lookaside buffer，快表，可以理解为页表缓冲，地址变换高速缓存), CPU 的pipeline需要刷掉;

**间接消耗**：多核的cache之间得共享数据,对于程序的影响要看线程工作区操作数据的大小。

#### 适用CASE

理想状态CPU，内存，IO外设都满负荷工作，大部分情况瓶颈都在IO的读写部分，适合计算性case。

#### 实现原理

linux源码，详细见additional.doc

### 1.5.2 线程同步

#### 线程状态

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

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

#### 线程同步

#### 同步和互斥

同步：多个线程并发访问共享资源时，保证同一时刻只有一个（信号量可以多个）线程使用。

实现同步的方法：

1）临界区（CriticalSection）：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。进程内可用。

2）互斥量：**互斥量用于线程的互斥。只能为0/1。**一个互斥量只能用于**一个资源**的互斥访问，可跨进程使用。

3）信号量：**信号线用于线程的同步。可以为非负整数，**可实现**多个同类资源**的**多线程**互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问。可跨进程使用。

4）事件：用来通知线程有一些事件已发生，从而启动后继任务的开始,可跨进程使用。

synchronized的底层实现就用到了临界区和互斥锁（重量级锁的情况下）这两个概念。

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。

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

#### 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

### 1.5.3 Sychronized

实现线程的互斥串行操作。保证接口的原子性和可见性。

#### 实现原理

在HotSpot虚拟机中，synchronized通过对象内部的监视器锁（monitor）来实现。

[Reference](http://www.cnblogs.com/dennyzhangdd/p/6734638.html)

**修饰代码段:**synchronized在JVM被编译为monitorenter、monitorexit指令来获取和释放互斥锁.。

**修饰方法：**方法flags标记为ACC\_SYNCHRONIZED

**Hotspot实现**

加锁值存在对象头部，即mark中。cas操作实现轻量级锁,但仅用于交替执行同步块的情况。对于同一Thread可用偏向锁，减少cas操作。多个线程在同一时刻进入临界区膨胀为重量级锁。依赖OS层面实现。

### 1.5.4 线程锁

加锁是保护共享资源逻辑一致性的通用方案：在并发环境下，通过锁使得在某个时刻只有一个线程或进程可以访问共享的资源

#### 重量级锁

即悲观锁。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。

实现原理：monitor依赖OS的Mutex Lock来实现的。操作系统实现线程之间的切换需要从用户态转换到核心态，这个成本非常高，需要相对比较长的时间。

#### 轻量级锁

轻量级锁即乐观锁。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。

原理：

**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，

#### 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，这些技术都是为了在线程之间更高效地共享数据，减少获得锁和释放锁所带来的性能消耗，以及解决竞争问题，从而提高程序的执行效率。

**锁粗化**:将多个连续的锁扩展成一个范围更大的锁，用以减少频繁互斥同步导致的性能损耗。

**锁消除:** JIT通过逃逸分析，去除堆上不会被其他线程访问到的代码锁。。

**轻量级锁：**轻量级锁即乐观锁。多个线程交替进入临界区，只需要依靠一条CAS原子指令就可以完成锁的获取及释放。

**偏向锁：**对轻量级锁进一步优化。使用CAS指令置换对象头ThreadID，下一次同一个线程进入则偏向该线程，无需任何同步操作。当有竞争，需解除偏向锁，进入轻量级锁

**适应性自旋:**为了避免线程频繁挂起、恢复的状态切换消耗。产生了忙循环（循环时间固定），即自旋。JDK1.6引入了自适应自旋。自旋时间根据之前锁自旋时间和线程状态，动态变化，用以期望能减少阻塞的时间。

#### 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。

如何排查？

执行jstack命令，查看线程是否一致处于WAITING状态。



### 1.5.5 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

#### 1.5.5.1 线程池

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。

原理

复用执行前面任务的线程，add过程受阻将添加到workQueue

TODO1 如何复用

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

**DB Pool源码分析**

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

TODO4

TODO1:如何检查线程池内线程运行情况，busy？ study监控API

#### 1.5.5.2 ReentrantLock

实现互斥串行操作的

ReentrantLock VS synchronized

本质上synchronized的lock升级到重量级锁依赖OS的mutex lock，有线程切换，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。

* 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。

#### 1.5.5.3 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。

#### 1.5.5.5 Semaphore

基于java同步器AQS

#### 1.5.5.6 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

对象引用变量读和写都是原子性的

原理：它是通过"volatile"和"Unsafe提供的CAS函数实现"原子操作

#### LimitLatch

### 1.5.6 fork-join

[Reference](https://www.liaoxuefeng.com/article/001493522711597674607c7f4f346628a76145477e2ff82000)

TODO1

## 1.5 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

### 1.5.1 JVM优化

JVM的编译优化

指令重排序，内存栅栏等

#### 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

### 1.5.2 类加载机制

#### 类加载过程

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。

hotspot源码显示最后调用allocate\_permanent方法在PermGen分配内存，所以instanceKlass对象保存在永久代区域；

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。

**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。

#### 模块化

（jboss modules、osgi、jigsaw）

### 对象内存管理

#### 运行时数据区



JVM区域总体分两类，heap区和非heap区。

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 Space（伊甸园）&Survivor Space(幸存者区)&Old Gen（老年代）。

非heap

Code Cache(代码缓存区)&Perm Gen（永久代）&Jvm Stack&Local Method Statck

#### 内存分配和回收

[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。。。

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>)

* 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。

**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?**

#### 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。

**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

### 1.6.3 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。

#### 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

#### 对象存活判定

**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GC回收的对象。

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；。

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

### 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 Feature

本章主要讲述各特性的原理。

JDK

TODO3JDK&JRE&JVM

和Google的Android关系https://www.zhihu.com/question/270271649

### 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
* 异步IO（NIO 2）
* Lambda的实现

JDK 8

2014年发布

* Lambda表达式
* Optional防NPE
* 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

### 泛型

TODO3

### 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。

反射效率低。需要读取字节码到内存然后分配内存生成类实例最后执行。

内省（Introspector）

### Assert

TODO3

### transient

TODO2

### Lambda

面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象，本质是java的语法糖，让开发更加专注于业务实现。

代码简洁，可读性好。比如事件处理，集合处理，回调。可以传递函数。

### Optional

TODO2:

### Pipelines&Streams

TODO1:JDK8,9新特性

JDK

# 第二篇 Java Web

j2ee

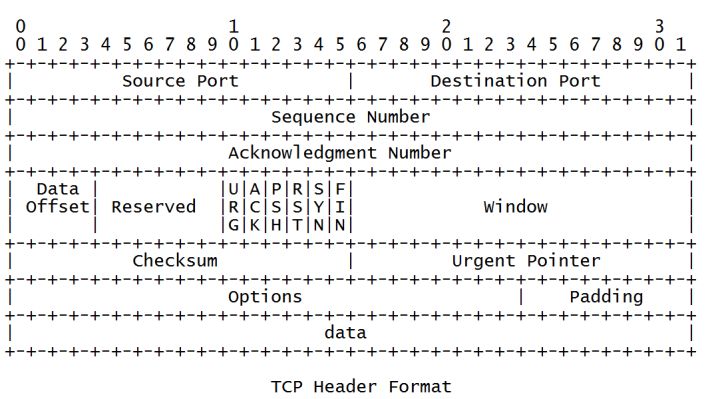
J2EE是由SUN提出的用于简化开发企业级应用程序的一系列规范的组合。

## 2.1 网络技术

### 2.1.1 TCP

[Reference](https://github.com/jawil/blog/issues/14)

#### TCP头部



一行32位 4byte，options和padding可选，所以header至少20bytes

TODO: 通过浏览器看到的request header存在哪个字段？

Header

Sequence Number 标识发送数据

Acknowledgment Number：接收端确认叔叔，并标识期望序号

Window

Options

Data

应用层数据包，比如HTTP

TODO 协议大小 [Refer](http://www.ruanyifeng.com/blog/2017/06/tcp-protocol.html)

#### 三次握手



#### 四次挥手



客户端收到FIN=1才Close,

**State**

LISTEN 服务监听

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

**TIME\_WAIT占用的资源：**少量内存（查资料大概4K）和一个fd。

#### TIME\_WAIT

产生原因

解决方案

### 2.1.2 HTTP

[Reference](https://developer.mozilla.org/zh-CN/docs/Web/HTTP)

超文本传输​​协议，用于传输诸如HTML的**超媒体文档**的应用层协议。它被设计**用于Web浏览器和Web服务器之间的通信**。基于TCP/IP层（也可以在任何传输层上使用,如UDP）。

#### HTTP/1.0

HTTP/1.0 版的主要缺点是，每个TCP连接只能发送一个请求。

无连接

早期浏览器与服务器交互数据频率低，发送数据完毕，连接就关闭，及时释放资源。如果还要请求其他资源，就必须再新建一个连接。

TCP连接的新建成本很高，因为需要客户端和服务器三次握手，并且开始时发送速率较慢（slow start）。所以，HTTP 1.0版本的性能比较差。随着网页加载的外部资源越来越多，这个问题就愈发突出了。

无状态

早期web多用于浏览，服务器完成响应即关闭连接，无状态解放了服务器。缺点在于每次请求会传输大量重复的内容信息。

解决方案 Cookie/Session。

#### HTTP/1.1

##### 长连接

不关闭连接。通过Connection: keep-alive/close控制，默认开启。对于同一个域名，大多数浏览器允许同时建立6个持久连接。实现原理:

管道机制（pipelining）

客户端可以同时发送多个请求，服务器还是按顺回应

Content-Length

服务器告诉浏览器完成回应，根据此字段判断response完成。在1.0版中，浏览器发现服务器关闭了TCP连接，就表明收到的数据包已经全了。

分块传输编码

对于一些很耗时的动态操作来说，这意味着，服务器要等到所有操作完成，才能发送数据，显然这样的效率不高。更好的处理方法是，产生一块数据，就发送一块，采用"流模式"（stream）取代"缓存模式"（buffer）

#### HTTP/2

#### Session

Http是无状态协议，Session在服务端需要记录用户的状态。所以session一般被用来保存临时全局变量，而非传递参数。

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

集群session共享

集群的时候要考虑Session的共享。

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)

TODO3:三次握手/窗口滑动机制

Unicode编码

#### Method

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

## 2.2 Servlet规范

Servlet接口是一套处理网络请求的规范，J2EE规范之一，主要为了扩展java的web功能。应用开发人员根据规范编写业务程序，servlet容器实现HTTP服务(面向接口编程)。

TODO2 菜单递归查找



## 2.3 Spring

### 2.3.1 IoC

**依赖倒置**：高层对抽象接口编程，**低层依赖于高层模块的需求抽象。**

**好处：**通过IoC容器管理对象之间的依赖关系，降低了资源双方的耦合度。Spring容器管理资源对象和对象之间的依赖关系，实现资源的可配置和易管理。

### 2.3.2 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

通过声明的方式灵活地进行事务管理，开发不用写事务管理代码（提高开发效率和质量）。

日志

### 2.3.3 Spring MVC

基于Servlet

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

在HandlerMapping中找到对应服务方法并调用。这里是反射call，CglibAopProxy静态代理。

MVC分层好处

1. 有利于分工
2. 复用
3. 好扩展，好维护

除了上述特性，使用spring，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的使用难度等。

## 2.4 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。

#### API

##### SessionFactory

openSession打开一个新的session

getCurrentSession获取当前线程里的session，如果没有才打开新的

##### Session

[Reference](http://coolvinson.iteye.com/blog/520610)

get

load 返回代理对象，延迟加载

#### 丢失更新

略

实现方案

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。
* 悲观锁(for update)，在DB层面实现。
* 设置为Serializable

LockMode

LockMode.NONE ：有缓存用缓存，没缓存则从数据库读   
LockMode.READ ：直接从数据库读，不使用缓存数据   
LockMode.WRITE ：在insert update数据的时候，HIBERNATE内部使用的。   
以上3种均为HIBERNATE级别的锁，也就是缓存级别的锁。   
下面2种为数据库级别的锁：   
LockMode.UPGRADE：相当于SQL语句select for update，被select的数据都被数据库锁住了，不能被其他事务修改。   
LockMode. UPGRADE\_NOWAIT ：是ORACLE数据库特有的select for update nowait

只读事务

[Reference](https://www.cnblogs.com/hackem/p/3890656.html)

设置了readonly后，connection都会被赋予readonly，效果取决于数据库的实现



Mysql：true的时候会读到最新值

##### 乐观锁

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



实现原理:



对于version已经改变的update,rowCount是0，如下图将throw StaleStateException。



##### 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



Thread2执行时select..将阻塞到报错。

Caused by: java.sql.SQLException: Lock wait timeout exceeded; try restarting transaction

##### 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。

### 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

TODO3:C3P0源码

## 2.5 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

### 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。在并发的情况下，get and set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，可解决。

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。

1. XMemcached

**分布式缓存**

### 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

## 2.6 应用服务器

TODO1:为什么大多数的服务都运行在Linux

TODO2如何做压力测试出负荷值

### 2.6.1 Tomcat

遵循J2EE Servlet规范，实现Servlet和JSP的支持，又叫Servlet容器。在中小型系统和并发访问用户不是很多的场合下被普遍使用。

Apache VS 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

#### 5.2.1 性能优化

Tomcat的默认配置作为生产环境，尤其是内存和线程的配置，默认都很低，容易成为性能瓶颈。 可以从三方面入手，内存/线程/IO

### 5.2 Jetty

### 5.2 Apache

### 5.3 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

# 第三篇 分布式应用

TODO2 JTA分布式事务等

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

## RPC

### Restful

规范

接口授权OAuth

TODO1

## SOA

传统分布式按功能模块把应用垂直拆分成多个应用。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?

### 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。

## CAP理论

## 分布式事务

## 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。

TODO1

# 第四篇 架构设计

## 4.1 OO设计准则

基本思想：尽可能的抽象，将变化控制到最小，做到高内聚、低耦合。

实现方法：使用对象、类、继承、封装、多态等基本概念来进行程序设计。

### 3.1.1 Object-Oriented

Object-Oriented是一种程序设计方法，从现实世界中客观存在的事物出发来构造软件系统，尽可能运用人类的自然思维方式。

基本思想：使用对象、类、继承、封装、多态等基本概念来进行程序设计。

**OO VS Procedure-oriented**

都是模块化编程，Procedure-oriented把问题分割成以功能函数为单位的模块中，

Object-Oriented把问题抽象描述成以对象为单位的模块中。

注意：本章主要讲述各特性的原理。

#### 封装

封装数据结构和函数，隐藏细节(private/public),避免被外界破坏。

好处1.符合人的自然思维2.从单一职责的角度来说？。

#### 多态

实现原理

方法的动态绑定，实现运行时的类型决定对象的行为。多态的表现形式是父类指针或引用指向子类对象，在这个指针上调用的方法使用子类的实现版本。多态是IOC、模板模式实现的关键

TODO4: invokevirtual执行过程

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

C++多态是通过虚函数来实现的。

#### 接口&抽象

接口优于抽象类：接口更加抽象，所以更稳定。

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

### 3.1.2 低耦合&高内聚

模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准――耦合性和内聚性。

#### 耦合性

指模块间相互联系紧密程度。为提高模块的独立性，模块间尽可能松散的耦合。

耦合性分类(低🡪高):

* 无直接耦合: 无调用关系，通过主模块的控制和调用产生联系。比如controller调用的service。

下面几种模块之间存在调用关系：

* 数据耦合: 传递简单的数据值，即值传递。
* 标记耦合: 传递的是数据结构，如数组名、类。
* 控制耦合: 传递的是控制变量（如开关、标志等）。
* 公共耦合: 访问同一个全局数据结构（包括单个变量），
* 内容耦合: 可能在汇编语言中出现。大多数高级语言都已设计成不允许出现。

1.高耦合度导致系统难维护，牵一发动全身，又会引入新的BUG。2.且不好拓展。

#### 内聚性

模块内各元素（变量，语句、、、）联系的越紧密，则它的内聚性就越高。在模块划分时，遵循“一个模块，一个功能”的原则，尽可能使模块达到功能内聚。

内聚性分类(低🡪高)

* 偶然内聚: 模块内各元素之间没有任何联系。
* 逻辑内聚: 模块把几种相关的功能组合在一起，每次被调用时，由传送给模块参数来确定该模块应完成哪一种功能 。
* 时间内聚: 把需要同时执行的动作组合在一起。(无顺序)
* 过程内聚：必须以特定次序执行。（有顺序）
* 通信内聚:处理使用相同的输入数据或者产生相同的输出数据。
* 顺序内聚: 顺序+同一功能。前一功能元素输出就是下一功能元素的输入。（）
* 功能内聚: 顺序+同一功能 +不可再分

好处见单一职责模式。设计类或者模块尽量做到“高内聚，低耦合”。

### 3.1.3 设计准则

[Reference](https://blog.csdn.net/dongnan591172113/article/details/51996421)

需求总是变化，所以就需要根据准则设计软件，变化满足需求的同时还能保持软件内部的封装体系稳定，不被需求的变化影响。且使我们的代码稳健，易于复用，易于拓展，灵活优雅。

设计准则 VS 设计模式

不同的设计模式对应不同的需求，而设计原则则代表永恒的灵魂，需要在实践中时时刻刻地遵守。

在实际编程中，根据设计准则，总结如下几点：

* 低层模块尽量都要有抽象类或接口，或者两者都有。
* 变量的声明类型尽量是抽象类或接口。
* 使用继承时遵循里氏替换原则。

#### 3.1.4.1 单一职责

一个类(模块)，一个功能。职责过多，可能引起它变化的因素就越多，增加系统错误风险。

#### 3.1.4.2 开放封闭原则

核心思想

对修改封闭的，对扩展开放。

具体实现

**封闭：**对抽象编程，而不对具体编程。**因为抽象相对稳定**。让类依赖于固定的抽象，所以修改就是封闭的。

**开放：**通过面向对象的继承和多态机制，重写/重载方法来改变固有行为，实现拓展，所以就是开放的。

#### 3.1.4.3 接口隔离

核心思想

使用多个小的专门的接口，而不要使用一个大的总接口。接口应该是内聚的，应该避免“胖”接口。

一个类对另外一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。  
    胖接口存在明显的弊端，会导致实现的类型必须完全实现接口的所有方法、属性等；而某些时候，实现类型并非需要所有的接口定义，而且对胖接口的修改将导致一连串的客户端程序需要修改

具体实现

分离的手段主要有以下两种：

1、委托分离，通过增加一个新的类型来委托客户的请求，隔离客户和接口的直接依赖，但是会增加系统的开销。

2、多重继承分离，通过接口多继承来实现客户的需求，这种方式是较好的。

分离后，每个类对应一个功能方法，将接口组装到类中，有个性化需求时，只要跟换新的实现类即可。

#### 3.1.4.4替换原则

核心思想：子类必须能够替换其基类。这一思想体现为对继承机制的约束规范

好处：保证继承复用是可靠地，违反必然导致违反开放封闭原则。

实现：面向接口编程

#### 3.1.4.5 依赖倒置

**核心思想**

**面向接口编程**。**上层模块不应该依赖底层模块，它们都应该依赖于抽象，低层依赖于高层模块的需求抽象。**

面向接口

依赖存在于类与类、模块与模块之间。当两个模块之间存在紧密的耦合关系时，最好的方法就是分离接口和实现，来有效控制耦合关系，达到依赖于抽象的设计目标。  
    抽象的稳定性决定了系统的稳定，依赖于抽象是面向对象设计的精髓，也是依赖倒置原则的核心。  
传递依赖：1.接口传递 2.构造方法3.setter

**倒置**



（图一） （图二）

图1依赖关系为高层依赖底层，图2依赖关系被颠倒（反转），从而使得低层次模块依赖于高层次模块的需求抽象。

## 4.2 设计技巧

### 3.2.1 继承&封装&多态

面向接口

### 3.2.2 内部类

TODO1

**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。

### 3.2.3 异常

[Reference](https://www.jianshu.com/p/503901e5be2b)

#### 3.2.2.1 异常分类



**Error**

编译和系统的错误，不允许捕获。

Checked异常

可以被处理的异常，必须显示处理。比如ClassNotFoundException / IOException

RuntimeException

产生频繁，处理麻烦，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。

#### 3.2.2.1 使用技巧

1处理好异常能保证系统的健壮性,安全性

2.重要的调试手段，根据抛出的异常能判断发生的位置和出了什么出错，为什么出错。

3.代码可读性，干掉if else，减少嵌套?

仅异常情况下使用异常

异常不要用来做流程控制，条件控制。异常设计的**初衷**是解决程序运行中的各种意外情况，且异常的处理**效率**比条件判断方式要低很多。



（虽然这里的异常可以用return Result，来减少嵌套并携带错误信息）

异常带上参数信息

记录日志时，尽可能带上参数信息，便于调试

不要捕捉所有异常

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (Exception e) {  }

不推荐的写法，应尽量避免,这样些会可能带来一些问题：

* 调用的接口的潜在Bug被忽略
* 不同类型的异常可能需要不同的处理，写在一起内聚性比较差

上面的代码可以更改为：

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (IOException e) {
5. // Do something handle IOException
6. } **catch** (CertificateException e) {
7. // Do something handle CertificateException
8. }

如果所有异常处理方式一致，且JDK版本高于1.7，可以简化如下：

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (IOException | CertificateException e) {
5. // Do something handle these exception
6. }

抛异常 VS 返回错误码

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。

应用内部推荐异常抛出；

跨应用间使用Result(Success,Code,message).

开放接口必须使用“错误码”

{"errcode":40018,"errmsg":"invalid name"}

说明：关于RPC方法返回方式使用Result方式的理由： 1）如果使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。 2）如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

#### 3.2.2.3 异常设计

异常框架要 1方便定位到错误 2使代码整洁 3给用户友好的错误提示。

一般一个项目都会有一套自己的异常框架。如下几种设计

1）设计的很细

异常太多，太麻烦。比如User表格字段校验异常。

2）AppException

只使用一个AppException，在这个异常类中添加一个type属性，通过type来区分不同的异常。type可以用一个枚举来维护，这个type值可以是上面提到的参数格式、业务校验、服务。也可以只用一个AppException，所有的异常都封装成这个，不再按type细分。

3）按层定义

DaoException/ServiceException/AppException。使用的时候会在服务层把DaoException转化为ServiceException，再在应用层把ServiceException封装成AppException。最外层有一个拦截全局异常，拦截AppException就可以了。

4）分类

ArgumentException: 参数格式异常，在所有的参数格式校验出错的时候使用。

BusinessException: 业务校验异常

ServiceException: 服务异常

### 3.2.4 接口设计

TODO1

## 4.3 设计模式

TODO:32中设计模式的好处从5大准则的角度来答。

### 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.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



观察者 VS 事件驱动

TODO3

## 4.4 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

### 3.3.1 性能

#### 性能指标

* 并发数：Concurrency。能同时处理的用户数。高并发
* 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 second）
* 性能计数器：包括System Load、对象和线程数、内存、CPU、磁盘、网络IO

#### 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

### 3.3.2 伸缩性

随着不断上升的并发访问压力和数据增长，需要不断向集群加入服务器的手段来缓解。伸缩性衡量是否方便构建集群，是否容易添加服务器。

#### 架构设计

**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

#### 应用服务器

#### DB集群伸缩性

关系数据库

1）主从分离

2）分表分库

NoSql

#### 缓存设计

### 3.3.3 扩展性

指现有系统影响最小的情况下，系统扩展或提升的能力。

**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。

## 4.5 软件工程

### 软件类型

这个世界变化实在是快，各种新名词层出不穷，让人眼花缭乱（云，大数据，AR,AI）。其实很多东西只不过新瓶装老酒而已。

单机类型

最开始的那些不需要互联网的单机软件。

C/S

通过安装的客户端访问服务。早期多为局域网。



B/S

浏览器/服务器架构。软件云端部署，用户通过通过浏览器来使用。



SaaS

软件即服务，SaaS不是一种软件架构，而是一种软件销售方式。本质上还是采用B/S架构。

但通过对每个使用者收取年租费或月租费来销售

优势：云端办公和较低资费。

目标用户：中小企业用户，因为他们要求不高，不像大企业那般“挑食”，通用版开发难度也低。

ERP型SaaS

互联网

### 云计算

**服务模式**

SaaS（软件即服务）、PaaS（平台即服务）和IaaS（基础设施即服务）



**部署方式**

私有云、社区云、公有云和混合云

**基本特征**

按需自助服务：按需使用，按需付费

广泛的网络访问、资源共享、快速的可伸缩性和可度量的服务。

企业级云

### 企业级软件

面向的用户为企业，政府，业务比较复杂。按功能划分为财务会计、ERP（企业资源规划）、CRM（客户关系管理）、SCM（供应链管理）、HRM（人力资源管理）、BI（商务智能）、CMS（内容管理系统）和企业通信工具等。也可以按行业划分为制造业、零售业、医药业等解决方案。

ERP

Enterprise Resource Planning，是一种供应链管理思想。包括采购、销售、制造、财务等功能

#### 复杂度高

政府学校和企业的采购模式、利益诉求不尽相同；不同生产、管理环节对软件特性、服务模式等的需求大相径庭；不同企业对同种软件也总会有个性化的要求；同一个软件在同一个企业内的同一个功能，也可能因部门间的厉害纠葛，导致需求不可理喻地分裂。

#### 国内现状

### 互联网公司

广义上来讲，企业规划、业务的开展都是基于互联网的前提之下施行的，都可以叫互联网公司。企业直接面向真实用户，即TO C。

特点：

追求高效，扁平化管理，去层级化，人本主义。

TODO3：和传统企业有什么本质区别？

2B VS 2C

2C，用户即购买者，更注重用户体验。2B，购买者为决策层，更注重提供生产和管理

国内现状：

用户有限，一天就是 24 小时，除去睡觉、工作，留给互联网的时间非常少，却有成千上万的产品在争夺这少得可怜的时间。而 B 端用户在工作时间使用产品，粘性大，高频刚需，还有可观的服务费收入

### 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。

## 4.6 Design Tools

### ER图

TODO4

### UML图

1）实现(implements)



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



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。



# 第五篇 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 Problem:脏读。
* Read committed Problem:其他提交事务所做的修改或删除，出现不可重复读
* Repeatable read  Problem:其他事务插入，此时发生幻像读。mysql默认隔离级别。
* Serializable 序列化。表锁。

MYSQL

1.查看InnoDB存储引擎 系统级的隔离级别和会话级的隔离级别

select @@global.tx\_isolation,@@tx\_isolation;

Mysql默认Repeatable-Read，除了幻想读，还会出现丢失更新。

丢失更新

[Reference](http://weifuwu.io/2015/12/30/mysql-second-kind-lose-update/)

|  |  |  |
| --- | --- | --- |
| **时间** | **trx1** | **trx2** |
| T1 | select，100 |  |
| T2 | update to 200.lock X |  |
| T3 |  | select，100元 |
| T4 |  | update to 200元 |
| T5 |  | [此时commit,会阻塞到超时] |
| T6 | commit |  |
| T7 |  | select，100 |
| T8 |  | commit |

**Repeatable read:**

trx2在T7返回100（忽略T4处的update），体现Repeatable read。

解决思路：

* 加悲观锁: 。trx1和trx2在select都for update
* 乐观锁。对数据version化，update前检查数据一致性，此时业务上就可以返回失败或者重新select再计算。数据库自动加排他锁，update本身是原子操作。

实际开发中，如更新数据基于原始数据，比如库存数，要注意lost update问题。

### 4.2.3 锁

DB提供的锁机制有行级锁....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

#### 行级锁

加锁粒度最小，但加锁的开销也最大。行级锁分为[共享锁](http://www.hollischuang.com/archives/923)和[排他锁](http://www.hollischuang.com/archives/923)。

共享锁(Share Lock)

事务T1对数据A加共享锁后，Mysql会对查询结果中的**每行都加共享锁。事务T2只能对A再加共享锁，不能加排他锁。**

用法

1. **SELECT** ... LOCK IN SHARE MODE;

排他锁（eXclusive Lock）

排他锁又称写锁，如果事务T1对数据A加上排他锁后，**T2不能再对A加任任何类型的锁，但是可以读**。

用法

1. **SELECT** ... **FOR** **UPDATE**;

innoDB对update、insert、delete会**自动加排他锁（**Select不加**）**。

意向锁

意向锁是表级锁，其设计目的主要是为了在一个事务中揭示下一行将要被请求锁的类型。InnoDB中的两个表锁：

意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁

意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

意向锁是InnoDB自动加的，不需要用户干预。

在 Mysql 中，行级锁并不是直接锁记录，而是**锁索引**。索引分为主键索引和非主键索引两种，如果一条sql 语句操作了主键索引，Mysql 就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。

InnoDB 行锁是通过给索引项加锁实现的，如果没有索引，InnoDB 会通过隐藏的聚簇索引来对记录加锁。也就是说：**如果不通过索引条件检索数据，那么InnoDB将对表中所有数据加锁，实际效果跟表锁一样。**因为没有了索引，找到某一条记录就得扫描全表，要扫描全表，就得锁定表。

#### 表级锁

#### 页级锁

#### 悲观锁&乐观锁

不是数据库中真正存在的锁，是并发控制采用的技术手段。其实不仅仅是关系型数据库系统中有乐观锁和悲观锁的概念，像memcache、hibernate、tair等都有类似的概念。

悲观锁

实现：在mysql层常用的悲观锁实现方式是加一个排他锁。

悲观并发控制实际上是“先取锁再访问”的策略，

不足：增加额外的开销还会降低了并行性 。

乐观锁

乐观并发控制直到提交时锁定，跟新锁定的数据是原子串行操作，所以不会产生任何锁和死锁。注意要加version控制。有点类似线程的cas操作，cas是原始数据和target数据比较。

## 4.3 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

InnoDB VS MyISAM

一是 InnoDB 支持**事务**。二是 InnoDB 采用了**行级锁**。

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。

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

MyISAM引擎缓存索引，而不缓存数据

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

|  |  |  |
| --- | --- | --- |
| **时间** | **Tran1** | **Tran 2** |
| T1 | start transaction | start transaction |
| T2 | select，100 | select，100元 |
| T3 |  | update to 200元 |
| T4 | update to 200元 |  |
| T5 |  | commit |
| T6 |  |  |
| T7 | commit |  |
| T8 |  |  |

1.T4处update或select..for update将阻塞到报错



### 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>

## 4.4 ORACLE

### PLSQL

**优势**

1 SQL语言只是访问、操作数据库的语言。PLSQL具备高级语言的特性，比如模块化，流程控制，异常.

2.集成在数据库中，调用更快。减少了网络的交互，从而减少网络传输时间。

## 4.5 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**

## 4.6 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