**压力测试工具 Apache Bench**

测试命令格式

ab [options] [http://]hostname[:port]/path

参数

-n 总请求数

-c 并发数

-t 测试最大时间

-r 抛出异常继续执行

-p 包含post的数据文件

-T content-type头信息

-C 附加Cookie

**JMeter 可视化工具**

**线程安全性**

不管何种调度方式，不需要任何额外的同步，都能表现出正确的行为，则称之为线程安全的。

**1原子性 同一时刻只有一个线程操作**

**AtomicXXX** CAS compartAndSwap原理

unsafe类中封装。实现了一个do-while循环，每次循环先从底层取出对象的值，和传进来的值比较，如果相同执行更新操作，如果不相同，继续循环。

LongAdder，将核心数据value分离成多个单元，每个单元独立维护，更新时通过哈希等算法映射到某个单元执行更新，最后将单元组合起来得到更新后的值。

AtomicStampedReference 解决了CAS的ABA问题，通过版本号。

要实现某段代码只执行一次，可以使用AtomicBoolean，通过compareAndSet方法判断。

**synchronized**

1. 修饰代码块，可以指定作用的对象
2. 修饰方法，作用对象为当前对象this，注意：子类不能继承父类方法的synchronized，需要自己声明
3. 修饰静态方法，作用对象为类对象，相当于类.class

**比较**

synchronized 不可中断锁，适合竞争不激烈，可读性好

lock 可中断锁，多样化同步，竞争激烈时能维持常态

Atomic 竞争激烈时能维持常态，比lock性能好，只能同步一个值

**2可见性**

导致共享变量在线程间不可见的原因：

线程交叉执行；

重排序结合线程交叉执行；

共享变量更新后的值没有在工作内存和主内存间及时更新。

JMM对synchronized的两条规定

线程解锁前，必须把共享变量的值刷新到主内存；

线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时，需要从主内存中重新读取最新的值。

**volatile** 通过加入**内存屏障**和**禁止指令重排序**优化

对volatile变量写操作时，会在写操作后加入一条store屏障指令，将本地内存中的共享变量值刷新到主内存；对volatile变量读操作时，会在读操作前加入一条load屏障指令，从主内存中读取共享变量。

volatile不具有原子性

**3有序性**

Java内存模型中，允许编译器和处理器对指令进行重排序。

**安全发布对象的方法**

1. 在静态初始化函数中初始化一个对象引用；
2. 将对象的引用保存到volatile类型域或者AtomicReference对象中；
3. 将对象的引用保存到某个正确构造对象的final类型域中；
4. 将对象的引用保存到一个由锁保护的域中

创建单例，使用双重校验同步锁的方式并不是线程安全的，原因是JVM和cpu优化，发生指令重排，有可能出现线程A设置instance指向分配的内存，但并没有完成初始化，线程B此时判断instance!=null，就会返回instance。解决方式是给instance加上volatile。

通过枚举实现线程安全的单例

**public class** SingtonExample1 {  
 **private** SingtonExample1() {}  
 **public static** SingtonExample1 getInstance() {  
 **return** Singleton.***INSTANCE***.getInstance();  
 }  
 **private enum** Singleton {  
 ***INSTANCE***;  
 **private** SingtonExample1 **sington**;  
 Singleton() {  
 **sington** = **new** SingtonExample1();  
 }  
 **public** SingtonExample1 getInstance() {  
 **return sington**;  
 }  
 }  
}

JVM保证对象只创建一次。

**不可变对象**

final修饰

Collections.unmodifiableXXX:Collection List Set Map

Guava ImmutableXXX:Collection List Set Map

**线程封闭**

Ad-hoc线程封闭：程序控制实现，忽略；

堆栈封闭：局部变量，无并发问题；

ThreadLocal线程封闭：特别好的封闭方法；

**线程不安全类**

StringBuilder -> StringBuffer

SimpleDateFormat线程不安全，可以使用堆栈封闭的方式解决。

joda-time中的dateTimeFormatter是线程安全的。

ArrayList、HashSet、HashMap线程不安全

先检查再执行 if(condition(a)) {handle(a);}容易出现线程安全问题。

**线程安全同步容器**

ArrayList -> Vector、Stack

HashMap -> HashTable

Collections.synchronizedXXX()

**并发容器J.U.C**

ArrayList -> CopyOnWriteArrayList，进行写操作时先拷贝一份，需加锁，代价高，容易引起GC，能够保证最终一致性，实现读写分离，但是代码实时性不强，适用于多读少写的场景

HashSet、TreeSet ->CopyOnWriteArraySet、ConcurrentSkipListSet

HashMap、TreeMap ->ConcurrentHashMap、ConcurrentSkipListMap

**安全共享对象策略**

**线程限制**：一个被线程限制的对象，由线程独占，并且只能被占有它的线程修改；

**共享只读**：一个共享只读的对象，在没有额外同步的情况下，可以被多个线程并发访问，但是任何线程都不能修改它；

**线程安全对象**：一个线程安全的对象或者容器，在内部通过同步机制来保证线程安全，所以其他线程无需额外的同步就可以通过公共接口随意访问它；

**被守护对象**：被守护对象只能通过获取特定的锁来访问

**AQS AbstractQueuedSynchronizer**

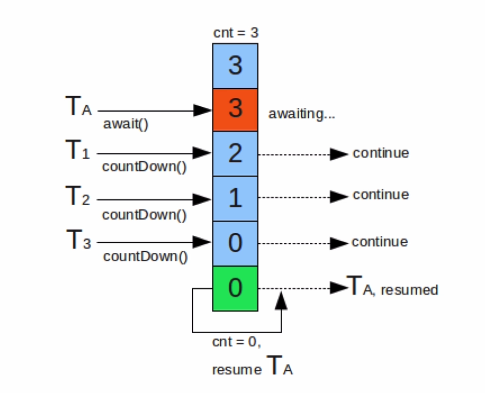
使用Node实现FIFO队列，可以用于构建锁或者其他同步装置的基础框架。

利用一个int类型表示状态

使用的方法是继承

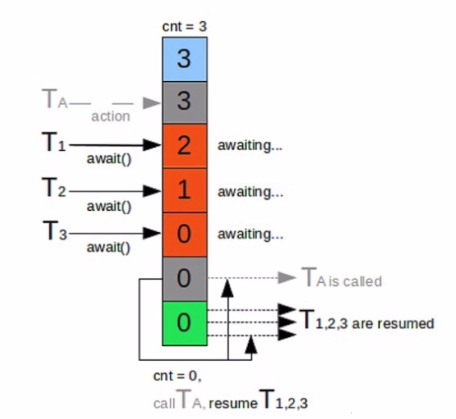
排他锁、共享锁

**CountDownLatch 闭锁** 一个或多个线程等待一组线程执行完成后再执行



**Semaphore 信号量** 控制并发数量

**CyclicBarrier 栅栏** 多个线程互相等待



计数器可以reset，多次使用。

**ReentrantLock**

ReentrantLock和synchronized都具备可重入性；

ReentrantLock由JDK控制，synchronized由JVM控制，synchronized不会引发死锁；

synchronized优化之后，性能和ReentrantLock相差无几（引入CAS，在用户态解决问题，避免在内核态产生阻塞）；

ReentrantLock细粒度更高，更加灵活；

synchronized只能是非公平锁，ReentrantLock可以指定是公平锁还是非公平锁；

ReentrantLock提供了一个Condition类，可以分组唤醒需要唤醒的线程；

ReentrantLock提供能够中断等待锁的线程机制，lock.lockInterruptibly()；

**ReentrantReadWriteLock**

悲观读取，执行写操作时，不能有其他读写锁。读多写少的情况，容易造成写饥饿。

**StampedLock**

可以指定使用乐观锁、悲观锁

乐观锁的使用思想：获得一个乐观锁tryOptimisticRead()，并保留票据，然后从内存读取到本地局部变量，通过票据检查读出后是否有其他写锁发生，如果有，再次获得一个悲观锁readLock()进行读取操作。

**Condition**

调用condition.await()时，从等待锁的队列中移除，加入到condition队列中。在其他线程condition.signalAll()后，将condition队列中的线程移除，加入到等待锁的队列。

**FutureTask**

传统创建线程的方式，无法获取执行结果。

通过Future可以获得Callable的方法call()的返回值，需要获取值时，调用future.get()方法，此时会阻塞当前线程，直到获取到结果。

FutureTask实现了Future和Runnable接口，创建时可以指定Callable参数。

**Fork/Join框架**

需继承RecursiveTask<>类，实现compute方法。将一个大任务拆分成若干个子任务，通过fork()执行子任务，join()合并子任务。

被窃取线程从head读取任务，窃取线程从tail读取任务。充分利用线程来执行任务。

不能有IO操作，不能存在检查异常

**BlockingQueue**



ArrayBlockingQueue初始化时指定大小，FIFO，队尾插入，队头移出；

DelayQueue实现Delayed接口，根据元素过期时间的优先级排序（应用场景：定时关闭连接、缓存对象、超时处理），用到锁和排序

LinkedBlockingQueue不指定大小，无边界（INT.MAXVALUE）

PriorityBlockingQueue插入的对象必须实现Comparable接口

SynchronousQueue只能容纳一个元素

**线程池**

**new Thread弊端：**

每次新建对象，性能差；

缺乏统一管理，可能无限制的新建线程；

功能少，如定期执行、线程中断

**线程池优点：**

重用存在的线程，减少对象的消亡、创建开销，性能好；

控制最大并发线程数，提高系统资源利用率，同时可以避免过多资源竞争，避免阻塞；

提供定时执行、定期执行、单线程、并发数控制等功能

**ThreadPoolExecutor**

参数：

corePoolSize 核心线程数量

maximumPoolSize 最大线程数

workQueue 阻塞队列，存储等待执行的任务

keepAliveTime 线程没有任务执行时保持时间

unit 单位

threadFactory 线程工厂

rejectHandler 当拒绝处理任务时的策略

方法：

execute() 提交任务

submit() 提交任务，能返回执行结果execute+Future

shutdown() 关闭线程池，等待任务都执行完

shutdownNow() 关闭线程池，不等待任务执行完

getTaskCount() 任务总数

getCompletedTaskCount() 已完成任务总数

getPoolSize() 当前线程数

getActiveCount() 正在执行任务的线程数量

**Executor框架接口，四种方式**

Executors.newCachedThreadPool

Executors.newFixedThreadPool

Executors.newScheduledThreadPool

Executors.newSingleThreadExecutor

注：ScheduledExecutorService可以延迟执行、定期执行。

合理配置：

CPU密集型任务，尽量压榨CPU，参考值NCPU+1

IO密集型任务，参考值2\*NCPU

**死锁**

互斥条件、请求和保持条件、不剥夺条件、环路等待条件

如何避免死锁：

1 加锁顺序

2 设置超时时间

3 死锁检测（回退重试、设置优先级部分回退）

**多线程并发最佳实践**

使用本地变量

使用不可变类

最小化锁的作用域范围：S=1/(1-a+a/n) S加速比，a 并行代码比例，n 并行数

使用线程池的Executor

宁可使用同步，也不要使用线程的wait和notify

使用BlockingQueue实现生产-消费模式

使用并发集合而不是加了锁的同步集合

使用Semaphore创建有界的访问

同步代码块代替同步方法

尽量不使用静态变量

**Spring的线程安全**

Spring bean的作用域：singleton prototype

单例的无状态对象是线程安全的。Spring并没有做多线程安全处理，原因是其创建的bean大多是无状态的，如果需要创建有状态的实例，就需要自己实现同步或者并发处理。

**HashMap和ConcurrentHapMap**

HashMap的底层实现是数组和链表，影响其性能的两个因素：初始容量和加载因子。

初始容量默认16，加载因子0.75

寻址方式：对于新插入的key取hashcode，根据数组长度取模，结果即为index，value插入链表的头部。

re哈希过程是线程不安全的，有可能出现死循环

ConcurrentHashMap，分成若干个Segment，每个Segment的操作类似于HashMap，分组基于哈希高位的运算。使用分段锁保证线程安全。

Java8中，HashMap产生哈希碰撞后形成的链表长度大于8时，采用红黑树优化性能，寻址时间复杂度从O(n)提升到O(logn)。

**扩容**

垂直扩容：提高系统部件能力

水平扩容：增加更多系统成员来实现

分布式服务器集群+数据库主从架构

**缓存**

特征：命中率、最大空间、清空策略

**清空策略包括：**

FIFO 先进先出

LFU 最少使用

LRU 最远最少使用，用于存储热点数据

过期时间

随机

影响缓存命中率的因素

业务场景和业务需求：读多写少，实时性要求越低越适合缓存

缓存的设计（粒度和策略）

缓存容量和基础设施

从架构师的角度，需要应用尽可能的命中缓存，并且避免缓存失效。

**分类：**

**本地缓存**：编程实现（成员变量、局部变量、静态变量）、**Guava Cache**

**分布式缓存**：Memcache、**Redis**

**Guava Cache**底层实现类似ConcurrentHashMap。自动将缓存的节点加入，当超出最大值时，采用LRU算法移除，缓存的key被封装在weak reference中，value被封装在weak reference或者soft reference中，可以统计命中率、异常率等信息。使用LoadingCache来存储信息，创建时借助CacheBuilder.newBuilder()然后设置参数。

**Memcache**，客户端采用一致性Hash算法路由到对应的Server，作为存储该key的Server，增加或者删除节点对整个集群影响不大。限制：chunk中总会有内存浪费；LRU算法不是针对全局的是针对某个slab的；key最大为250Byte，存放的value大小不能大于1M。

**Redis**，非关系型数据库，可以存放5种数据类型，可以持久化数据到硬盘，可以使用复制特性扩展读性能，可以使用客户端分片扩展写性能。性能极高，读速度11万/s，写速度8.1万/s，具有原子性。

Redis应用场景：取最新n个数据、排行榜、精准设置过期时间、计数器、唯一性检查、实时系统、垃圾系统、消息系统、消息队列、缓存。

**高并发下缓存问题：**

缓存一致性 更新缓存、数据库时，一方成功，一方失败

缓存并发问题 缓存未命中，需要从数据库查询更新缓存，此时要用到锁机制来避免线程安全问题

缓存穿透问题 缓存未命中，数据库也为空，就产生了很多不必要的查询。可以通过缓存空对象、单独过滤处理来解决。

缓存雪崩现象 大量请求到达数据库，导致服务器崩溃。造成这种现象的原因可能是缓存集中失效等原因。针对不同的场景采用限流、降级、熔断、多级缓存。多进行压力测试。

**消息队列**

业务无关：只做消息分发

FIFO：先投递先到达

容灾：节点的动态增删和消息的持久化

性能：吞吐量提升，系统内部通信效率提高

为什么使用消息队列？生产和消费的速度和稳定性等因素不一致

使用场景：业务解耦、最终一致性、广播、错峰与流控

**Kafka、RabbitMQ**

**应用拆分**

业务优先

循序渐进

兼顾技术：重构、分层

可靠测试

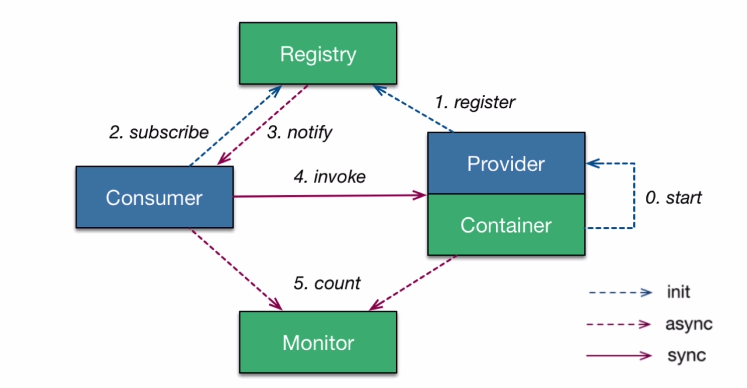
**思考**

应用之间通信：RPC(dubbo等)、消息队列

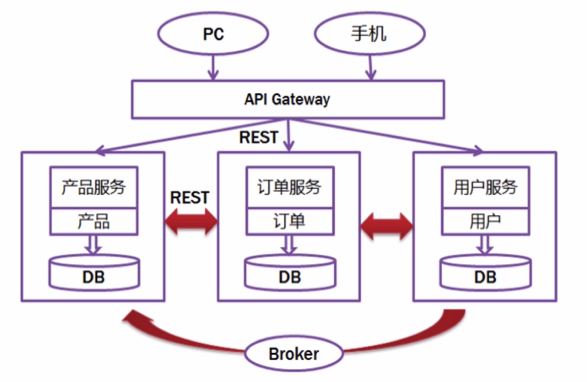
应用之间的数据库设计：每个应用都有独立的数据库

避免事务操作跨应用

Dubbo



微服务



**应用限流**

算法：

1）计数器法

2）滑动窗口

3）漏桶算法

4）令牌桶算法（允许一定程度的突发）

Guava的RateLimiter使用令牌桶的方式进行限流。

分布式限流实现，可以考虑用Redis的incrby key num

**服务降级**

如果当前请求出错了或者处理不了了，给一个默认的返回。

**服务熔断**

由于某些原因，出现过载，为防止系统故障，采取熔断措施。

服务降级分类：

自动降级：超时、失败次数、故障、限流

人工降级：秒杀、双11大促

**对比**

共性：目的、最终表现、粒度（服务级别）、自治

区别：触发原因、管理目标层次、实现方式

**服务降级要考虑的问题：**

核心服务、非核心服务

是否支持降级，降级策略

业务放通场景，策略

**Hystrix**

在通过第三方客户端访问依赖服务出现高延迟或者失败时，为系统提供保护和控制；

在分布式系统中防止级联失败；

快速失败（Fast Fail）同时能快速恢复；

提供失败回退（Fall Back）和优雅的服务降级机制；

提供近实时的监控、报警和运维控制手段。

**数据库切库、分库、分表**

数据库瓶颈：

单个库数据量太大（1T~2T）：多个库

单个数据库服务器压力过大、读写瓶颈：多个库

单个表数据量过大：分表

切库实际应用：读写分离，主库更新，从库读取

支持多数据源、分库

数据库分表：横向（水平）分表、纵向（垂直）分表

数据库分表的实现插件，mybatis的shardbatis2.0

**高可用的一些手段**

任务调度系统分布式：elastic-job + zookeeper

主备切换：apache curator + zookeeper分布式锁实现

监控报警机制