独家Java面经

## 一.Java集合与数据结构

## 二.关键字，泛型等基础

## 三.设计模式（jdk和框架里的应用）

### 1．Design Pattern分类

设计模式一般按创建型，结构型和行为型三种来分。

创建型：单例模式(Bean，**线程池**)和工厂模式（dubbo源码缓存）

结构型：外观模式（日志门面）适配器模式（HandlerAdapter）和代理模式（aopProxy）

行为型：策略模式（HandlerMapping）和委派模式（Dispatcher）

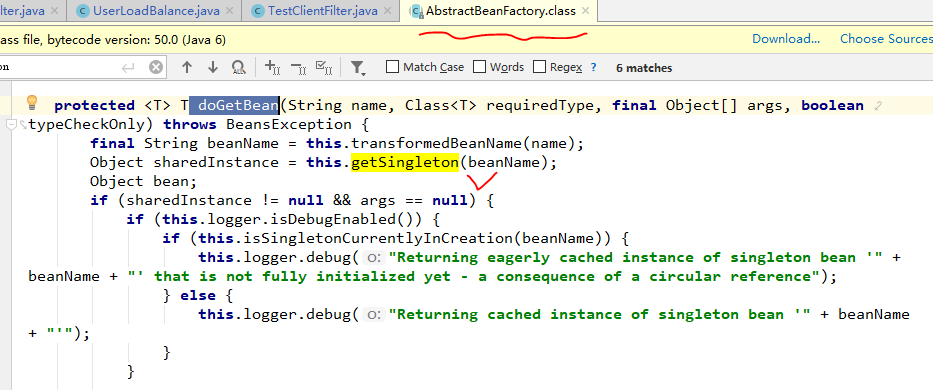
### 2.Spring使用的设计模式

#### 2.1 双重锁单例和工厂模式用于Ioc容器中 ApplicationContext调用BeanFactory创建bean对象

Spring依赖注入Bean实例默认是单例的。Spring提供了全局的访问点BeanFactory。

目的：需要频繁实例化然后销毁的对象。

SpringIoC的依赖注入（包括lazy-init方式）都是发生在 AbstractBeanFactory 的 getBean 里。getBean 的doGetBean 方法调用 getSingleton 进行bean的创建。



spring依赖注入时，使用了**双重判断加锁**的单例模式。下图是同步线程安全的单例核心代码：

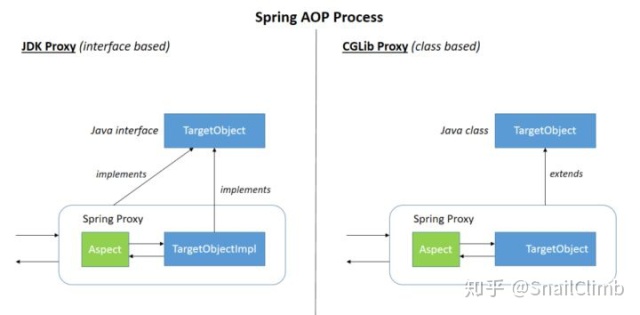


我们一路跟踪下去，发现实际上是调用了 AbstractAutowireCapableBeanFactory 的 doCreateBean 方法，返回了BeanWrapper包装并创建的bean实例。

#### 2.2策略模式实现了Controller，Servcie，Dao三层架构中只依赖上一层接口

策略模式，Spring实例化对象的时候用到，实现了Controller，Servcie，Dao三层架构中只依赖上一层接口；Spring策略对象依赖注入到context对象，context对象根据它的策略改变而改变它的相关行为。

### 3.JDK使用的设计模式

1. 适配器模式Adapter，日志的Mybatis接口和它对jdbc、log4j等各种日志框架的适配实现；模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
2. 代理模式 ，AOP用到了动态代理（如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDK Proxy创建代理对象，由于动态代理是基于接口实现的，所以对于没有实现接口的对象无法使用JDK Proxy 去进行代理了，这时Spring AOP会使用 Cglib代理， 从内存中构建一个子类来扩展目标对象的功能）如下图所示：

### 4.Mybatis遇到过的设计模式？（4种）

1.单例模式 ErrorContext和LogFactory；

其中ErrorContext是用在每个线程范围内的单例，用于记录该线程的执行环境错误信息，而LogFactory则是提供给整个Mybatis使用的日志工厂，用于获得针对项目配置好的日志对象。（详见Q4）

2.工厂模式，SqlSessionFactory、ObjectFactory、MapperProxyFactory；

3.适配器模式，例如Log的Mybatis接口和它对jdbc、log4j等各种日志框架的适配实现；

4. 代理模式是Mybatis核心使用的设计模式，正是用了代理模式，我们只需要编写Mapper接口，然后由Mybatis后台帮我们完成具体SQL的执行

当我们使用Configuration的getMapper方法时，会调用mapperRegistry.getMapper方法，而该方法又会调用mapperProxyFactory.newInstance(sqlSession)来生成一个具体的代理：

public class MapperProxyFactory<T> {

private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();

public MapperProxyFactory(Class<T> mapperInterface) {

this.mapperInterface = mapperInterface;

}

public Class<T> getMapperInterface() {

return mapperInterface;

}

public Map<Method, MapperMethod> getMethodCache()

{ return methodCache;

}

@SuppressWarnings("unchecked")

protected T newInstance(MapperProxy<T> mapperProxy) {

return (T) Proxy.newProxyInstance ( mapperInterface.getClassLoader(), new Class[

] { mapperInterface }, mapperProxy ) ;

}

public T newInstance(SqlSession sqlSession) {

final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);

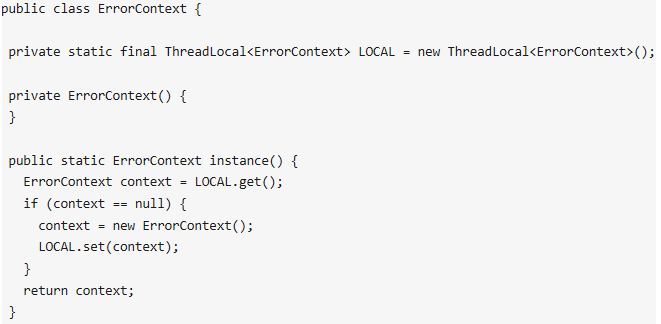
return newInstance(mapperProxy); }

}

### 5.最熟的的设计模式能用图或代码描述吗？（结合具体项目来说）

**🡪** [回答：使用原因+实现过程+使用效果]

A4:【单例模式】以Mybatis里用在每个线程范围内的单例ErrorContext为例：



有趣的地方是，LOCAL的静态实例变量使用了ThreadLocal修饰，也就是说它属于每个线程各自的数据，而在instance()方法中，先获取本线程的该实例，如果没有就创建该线程独有的ErrorContext。

java中饿单例模式性能优于懒单例模式，c++中一般使用懒单例模式

## 四.多线程与并发编程

# 四．多线程与并发

### 1进程VS线程

#### Q1：对进程与线程比较？为什么要用线程？

**A1:**进程是一个正在被执行的程序的实例（被CPU载入内存）。液压测控系统软件运行时即一个进程。

线程是能独立运行的最小单元。

主要区别有两个：

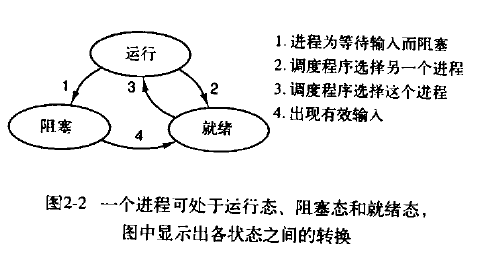
1.进程在执行过程中拥有独立的内存单元，多个线程共享同进程的所有资源，2. 进程是OS分配资源的最小单元，线程是OS调度的最小单元。同一进程中的多个线程之间可以并发执行。

注：使用多线程是为了提高任务的平均执行速度

但线程并非越多越好，类比搬家时人太多会拥堵可看出还要考虑CPU的负载能力

#### Q2:进程，线程状态？

进程有3种：



进程有新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态

1. 当程序使用new 关键字创建了一个线程之后，JVM 为其分配内存，并初始化其成员变量的值

2.当线程对象调用了start()方法之后， Java 虚拟机会为其创建方法调用栈和计数器，等待调度运行。

3. 开始执行run()方法的线程执行体，则该线程处于运行状态。

4.阻塞状态是指线程因为某种原因放弃了CPU使用权。分为三种：等待阻塞，同步阻塞和sleep阻塞

5.线程三种结束状态：执行完run()/call()正常结束;抛异常结束；调用Stop结束􂣦



注：线程则是JVM级别的，以Tomcat 为例，如果你在Web 应用中启动一个线程，这个线程的生命周期并不会和Web应用程序保持同步。也就是说，即使你停止了Web应用，这个线程依旧是活跃的。

#### Q3: Java中有哪两类线程，守护/后台线程怎么理解？

A3：User Thread(用户线程)和Daemon Thread(守护线程)

守护线程的优先级较低，用于为系统中的其它对象和线程提供服务。将一个用户线程设置为守护线程的方式是在线程对象创建之前用线程对象的setDaemon方法，将其置为true

Ex: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。

#### Q4:如何创建线程？（实现多线程）

A4:多线程的实现主要有三种方式:继承java.lang.Thread类，实现java.lang.Runnable接口,以及实现Callable并配合着Future接口和FutureTask类返回值。（实现详见JUC）

三者比较：Runable（重写接口的run方法）比继承thread对外暴露的细节少，使用者可以更专注Run方法，而且更适合多个代码相同的线程处理统一资源。前两种创建的线程执行完任务后无法直接获得执行结果。第三种借助共享变量获取Callable可以通过call获得返回值，和Future、FutureTask配合可以用来获取异步执行的结果。



#### Q5: （主动）终止线程方式？

1.用标志位退出线程；在run执行体中加while(!exit)

2.如用Interrupt方法：分为阻塞态和非阻塞态见下图代码

3.用stop方法强行停止（线程不安全）



#### Q6：线程间如何通信？

通信方法：1.wait()/notify()的等待通知机制2.锁结合线程间通信工具类【Semaphore 、CountDownLatch见JUC】3.隐式线程通信是用全局共享变量，并用volatile修饰保证线程的同步。【引申volatile】

等待/通知机制是指一个线程A调用对象O的wait()方法进入等待状态，而另一线程B调用对象O的notify()或notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。上述两个线程对象O来完成交互，而对象上的wait()和notify/notifyAll()的关系就如同开关信号一样，用来完成等待方通知方之间的交互工作

注：对于暂停恢复方法suspend()暂停易致死锁、resume()恢复、stop()停止线程但不释放锁，三种都不如等待/通知机制

### 2.线程们的安全

#### Q1: 线程安全的理解及三种特性的保证？

线程安全是指线程在操作类的过程中保证类行为正确，主要体现在原子性，有序性和可见性三个方面。

(1)对象只读；用final修饰类，方法避免被继承或覆盖；用static final修饰变量避免运行时被修改。

String,Integer这类不可变对象允许复制拒绝写入。

(2)用线程安全类；StringBuffer,Vector,Hashtable用synchronized修饰相关方法。

(3)数据仅单线程内可见；如ThreadLocal,存储在JVM的栈的局部变量表中，与其它线程毫无关系。

(4)同步和锁机制；要对某个对象做并发跟新且不属于上述三种情况，不过实现复杂【引申锁】

原子性：

1）Synchronized锁提供互斥访问，保证同一时刻只有一个线程进行访问。

2）Atomic包的AQS，每次只能保证同步一个值

3)Volatile可保证Long对象不原子的操作变成原子操作（Long对象的读取需要分两次）

保证共享变量在线程中的**可见性**：（多CPU时问题比原子性多得多）

不可见原因：线程交叉执行；共享变量的值改变后没有在主内存中及时更新；指令重排序结合线程交叉执行

(1)用Synchronized修饰，利用原理是JMM的规定，线程加锁时清空工作内存中该值从主存重新Load，而解锁前必须将共享变量最新值刷新到主存。

(2)，用volatile修饰禁止指令重排，

（3）final修饰的字段在构造器中一旦被初始化，那么在其他线程就能看见final字段的值。防止构造器没有把“this”的引用传递出去时，其他线程有可能通过这个引用访问到“初始化了一半”的对象。

有序性：一个线程观察其它线程的指令执行顺序

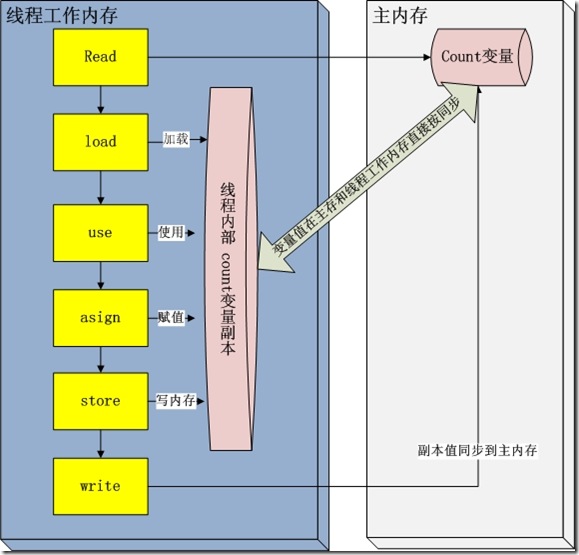
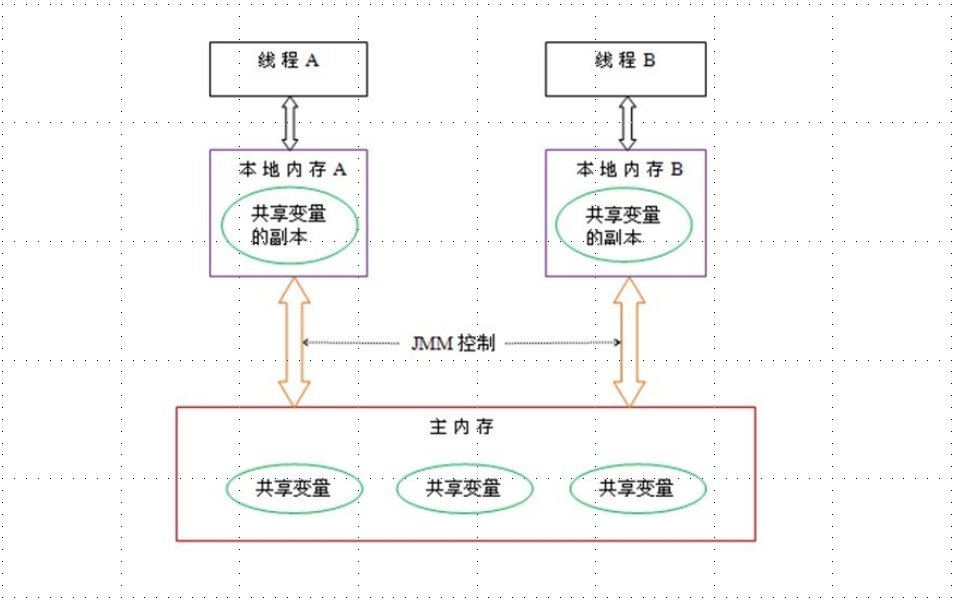
（1）见JVM

#### Q2: 线程同步的理解？

线程同步：多个线程对一个共享、可变对象的访问互斥。

两种实现方法，一种依靠关键字Synchronized,另一种是依靠ReentrantLock类实现（见锁）

#### Q3:JMM模型



Java Memory Model结构和八大基本原子操作

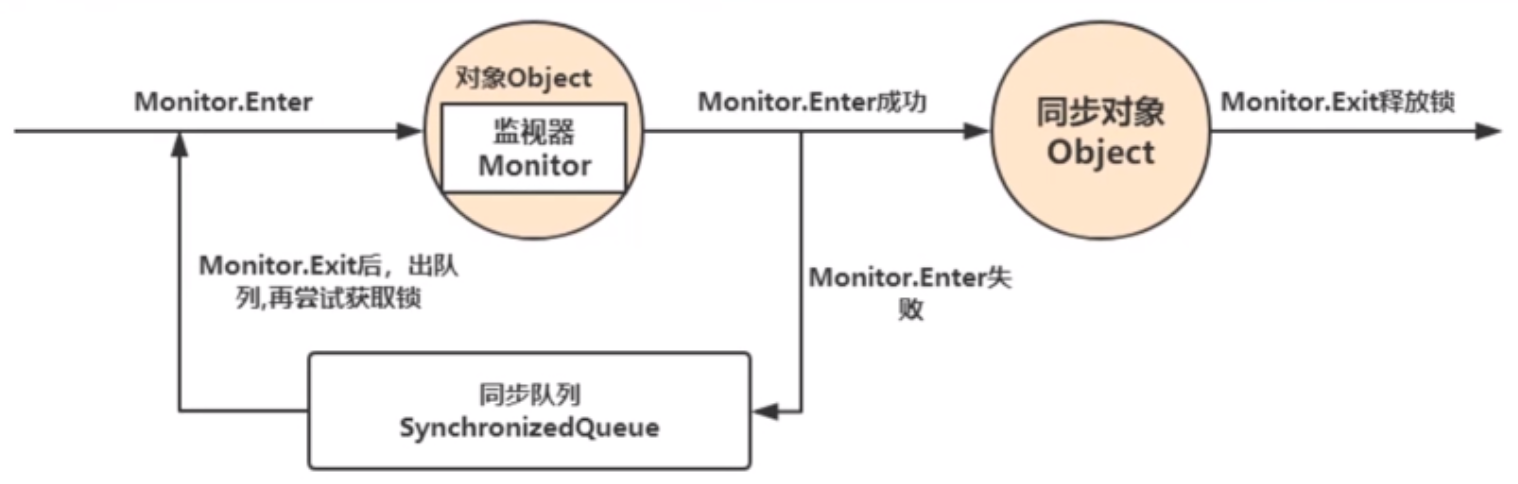
Java假设每个线程都跑在自己的处理器上，享有自己的内存，和共享的主存交互。

#### Q4：Synchronized实现原理（如何实现同步的）？

他属于独占式的悲观锁，非公平锁，偏向锁，同时属于可重入锁【引申锁】。

**A：**当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

Synchronized是通过JVM底层负责实现的，JVM的监视锁是每个对象与生俱来的隐藏字段，使用Synchronized时JVM会根据它的使用环境找到对应对象的monitor,再根据monitor的状态进行加、解锁。Java内存模型（JMM）定义了8大操作都是原子的（不可再分）。



**作用范围/如何加锁：**

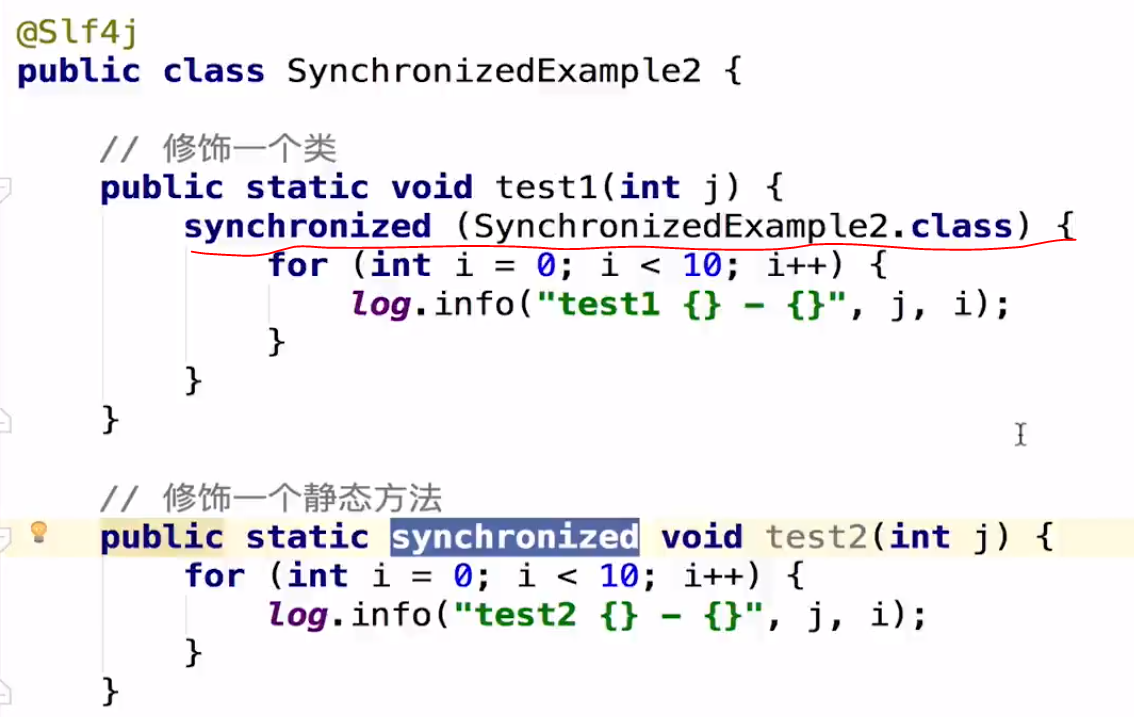
1. 锁实例方法时，锁住的是实例（this）对象；

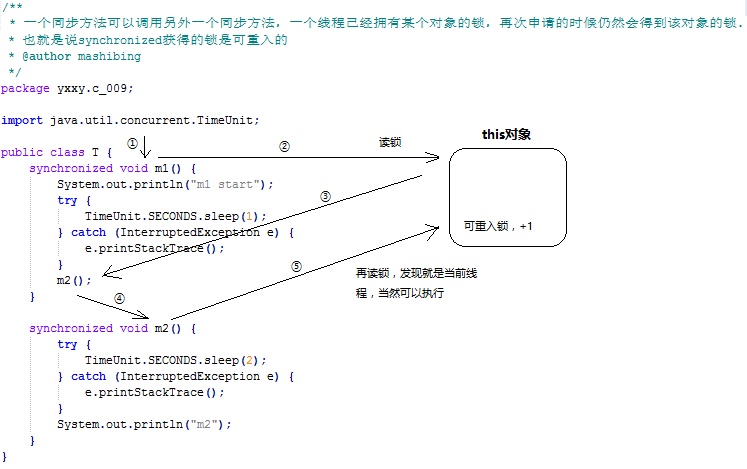
2. 作用于静态方法，锁住的是类（Class）对象，(静态方法锁相当于类的一个全局锁，会锁所有

调用该方法的线程；)

1. 作用于一个对象实例，锁住的是所有以该对象为锁的代码块。它有多个队列，
2. 作用于代码块时，锁住的是其当前对象，因此不同对象调用代码块可以交替执行互不影响，不会等一个全执行完再执行另一个。

注：子类继承了父类里带Synchronized的方法情况，作用在子类方法里无效。因为Synchronized不是声明的一部分。





**Synchronized偏向锁实现（不常见）：**

偏向锁是为了在没有多线程竞争资源时尽量减小锁带来的开销。在锁对象的对象头中有一个threadID,初始化的时候为null,如果有线程访问就将线程ID赋给threadID，每次获取锁时，判断当前线程ID是否等于锁对象的threadID，如果相等就不会再重复获取锁，偏向锁比轻量级锁还开销小。

#### Q5：volatile 关键字特性，为什么它能保证变量对所有线程的可见性？􁙗

**变量可见性：**指的是当某个线程修改了volatile修饰的变量，那么新值可被其他线程立即获取。因为volatile修饰意味着这个变量的所有操作在主内存中进行，不会产生副本，以保证共享变量的可见性。

**禁止重排序：**volatile 禁止了JVM自带的指令重排。

Volatile是比synchronized更轻量级的同步锁（不会使执行线程阻塞）；

使用场景：一个变量/字段被多个线程共享，线程直接给这个变量赋值。

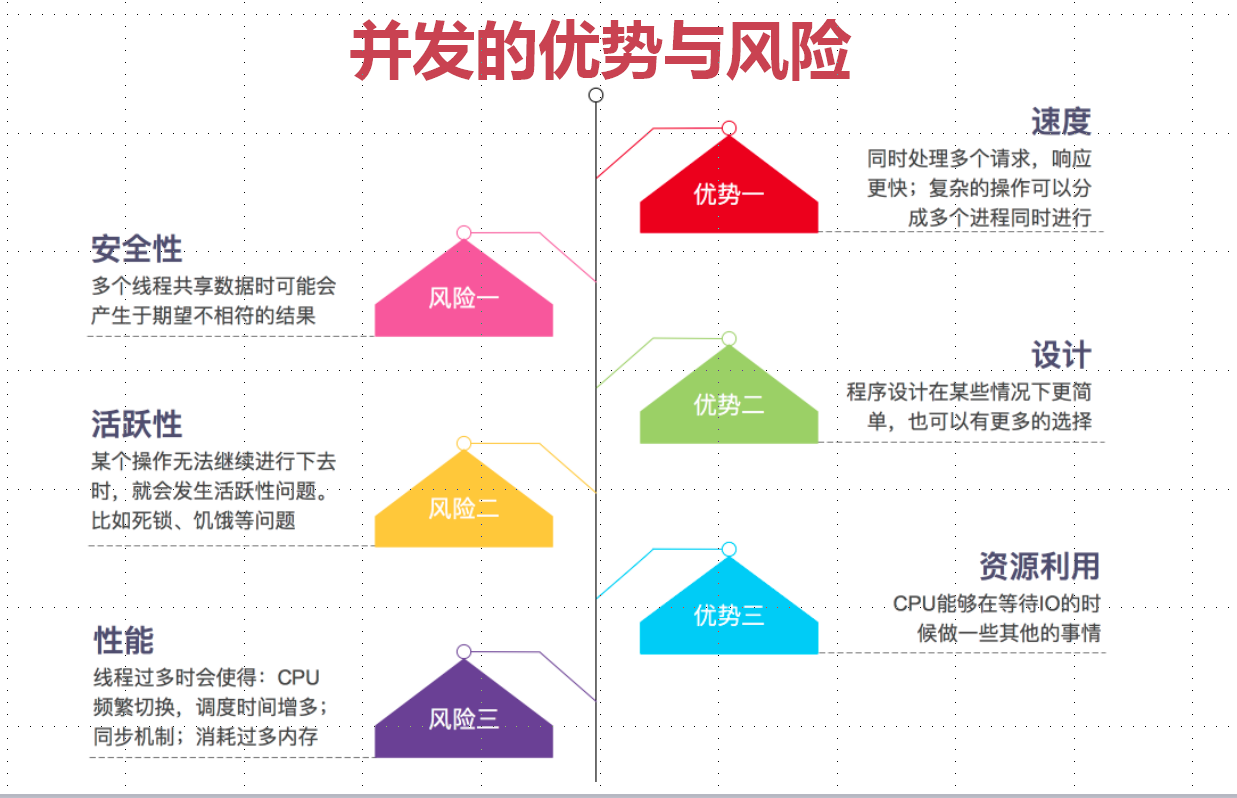
### 3.Java并发包JUC

#### Q1:并行与并发

关键区别：并发是一个进程断断续续做多个任务，并行是多个进程同一时间在执行各自任务

Q2:FutureTask组件的原理？

Callable是功能与Runnable一样的泛型接口；Future接口可监视Callable调用情况；FutureTask类的父类是RunnableFuture（继承了Runnable和Future两个接口），如果构造函数参数是runnable就会转换成Callable类型，既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。



#### Q2:同步容器AQS（AbstractQueueSynchronizer）

AbstractQueuedSynchronizer是JUC包实现同步的基础工具，提供了一个队列，大多数开发者可能从来不会直接用到AQS，AQS有个变量用来存放状态信息 state,可以通过protected的getState,setState,compareAndSetState函数进行调用。对于ReentrantLock来说，state可以用来表示该线程获可重入锁的次数，semaphore来说state用来表示当前可用信号的个数，FutuerTask用来表示任务状态（例如还没开始，运行，完成，取消）。

具体工具类：CountDownLatch, Semaphore和 CylicBarrier

<https://zhuanlan.zhihu.com/p/40669746>

【AQS的LOCKS接口可见锁机制】

#### Q3:AQS中的Locks接口

锁是用来控制多个线程访问共享资源的方式，一般用来防止多个线程同时访问共享资源但是有些锁允许多个线程并发访问共享资源。

### Java 线程池

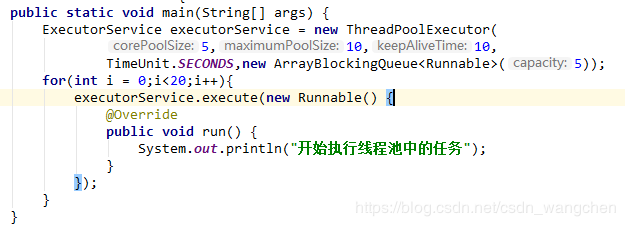
<https://www.zhihu.com/search?type=content&q=%E7%BA%BF%E7%A8%8B%E6%B1%A0>

#### Q1:为什么要用线程池？

**A1：**线程池好处（对比每任务每线程）1.重用线程，减少创建销毁等开销。 而且在请求到达时，工作者线已存在不需要花时间等待创建线程，因此提高了响应性。2.通过控制线程池的大小可以内存耗尽等。

#### Q2:Java 中的线程池是如何实现的？创建线程池的几个核心构造参数？

**A2：**JUC包直接提供了Excutors调用静态工厂方法**创建**四种不同线程池【引申ExcutorFixedThreadPool，SingleThreadPool，CachedThreadPool和ScheduledThreadPoolExecutors。】但是没有传入阻塞队列LinkedBlockQueue的长度，阻塞队列就是个可无限添加任务的无边界队列，高并发时会OOM。

因此用构造方法ThreadPoolExecutor手动设置核心参数控制最大并发量：

其可控的核心参数

int corePoolSize：该线程池中核心线程数最大值

int maximumPoolSize： 该线程池中线程总数最大值

BlockingQueue workQueue：该线程池中的任务队列，维护着等待执行的Runnable对象，如果队列满了，则新建非核心线程执行任务。

**🡪**通过Future和Callable子接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现了。

#### Q3: 描述线程池的创建，提交任务，监控到关闭的过程

A4:创建在上Q，可以向ThreadPoolExecutor**提交**两种任务：Callable和Runnable(引申Executor框架)；Callable是后加的算是Runnable的补充，Callable允许有返回值也允许抛异常，而Runnable均无，

关闭线程池有两种方式：shutdown和shutdownNow，关闭时，会遍历所有的线程，调用它们的interrupt函数中断线程。但这两种方式对于正在执行的线程处理方式不同。

#### Q4: 线程池异常处理

<https://www.zhihu.com/search?type=content&q=%E7%BA%BF%E7%A8%8B%E6%B1%A0>

#### Q5:说说对Excutor框架的理解

Java里面线程池的顶级接口是**Executor，**真正的线程池接口是**ExecutorService**



**🡪**用ThreadPoolExecutor自定义线程池要注意什么？

用ThreadPoolExecutor要看线程实际用途，如果任务量不大，可以用无界队列，如果任务量非常大，要用有界队列，防止OOM；最大线程数一般设为2N+1最好，N是CPU核数

Executor框架中的所有类可以分成三类：

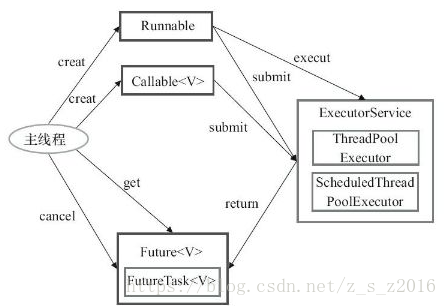
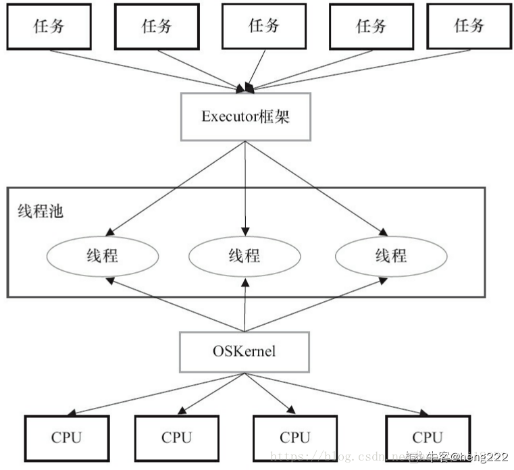
1.任务 ：Runnable和Callable。

2.任务执行器 ；Executor框架最核心的接口是Executor，它表示任务的执行器。Executor的子接口为ExecutorService。ExecutorService有两大实现类：ThreadPoolExecutor和ScheduledThreadPoolExecutor。

3.执行结果；Future接口表示异步的执行结果，它的实现类为FutureTask。

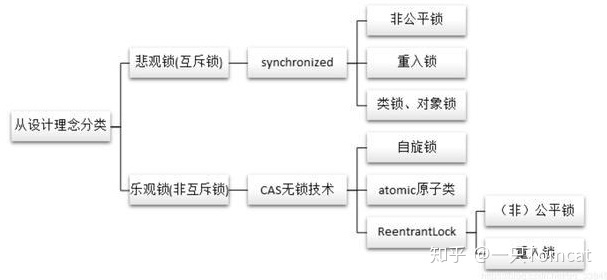
多线程程序通常把应用分解成若干个任务，然后Executor将任务映射为固定数量的线程，底层，系统内核将线程映射到cpu处理器上。应用程序通过Executor控制上层调度，而下层通过操作系统内核控制。

**Excutor框架两级调度模型**



### 5.多线程的锁机制

#### Q1：锁的主要分类



#### Q2：CAS怎么实现？（源码）

https://zhuanlan.zhihu.com/p/44871583

#### Q3：Synchronized &ReentrantLock 异同？（同步锁）

Synchronized（同步）是关键字，ReentrantLock（可重入锁）是类，意味着R比S多可以被继承、可以有方法、类变量，等灵活的功能。

1. ReentrantLock通过方法lock()与unlock()来进行加锁与解锁操作，与synchronized会被JVM自动解锁机制不同，ReentrantLock加锁后需要手动进行解锁。为了避免程序出现异常而无法正常解锁的情况，使用ReentrantLock必须在finally控制块中进行解锁操作。

2. R相比synchronized可中断、

3.R在构造函数中用队列实现是否公平锁的初始化方式。但是S只能用非公平锁机制。

#### Q4：死锁的概念，产生条件，解决方法与防止手段？

https://www.zhihu.com/search?type=content&q=%E6%AD%BB%E9%94%81

#### Q5：公平锁&非公平锁？

非公平锁 ：JVM按随机、就近原则分配锁的机制，非公平锁能更充分的利用cpu的时间片，尽量的减少cpu空闲的状态时间，实际执行的效率要远远超出公平锁，最常用非公平锁的分配机制。(Synchronized是非公平锁。Synchronized在线程进入ContentionList时，等待的线程会先尝试自旋获取锁，如果获取不到就进入ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占OnDeck线程的锁资源。)

公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁。使用场景是如果业务中线程占用(处理)时间要远长于线程等待，公平锁会给业务增强很多的可控制性。

公平锁比非公平锁多的一个判断方法：

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0) {

if (!hasQueuedPredecessors() &&compareAndSetState(0, acquires)) {

setExclusiveOwnerThread(current);

return true;

}

}

//以上部分两种所都有，hasQueuedPredecessors的实现只有公平锁

public final boolean hasQueuedPredecessors() {

Node t = tail; // Read fields in reverse initialization order

Node h = head;

Node s;

return h != t &&

((s = h.next) == null || s.thread != Thread.currentThread());

}

#### Q6：ReentrantReadWriteLock

它可以实现读写分离，多个线程同时进行读取，但是最多一个写线程存在

<https://zhuanlan.zhihu.com/p/87590807>

### 6.线程局部变量ThreadLocal

<https://mp.weixin.qq.com/s?__biz=MzA5MzI3NjE2MA==&mid=2650243742&idx=1&sn=27de324a1685ebbb112a239777eb7f52&chksm=886373f1bf14fae7c7515c76d65707dc6f026996c7cbfe3df16a52d86ced2904e154941f744f&scene=38#wechat_redirect>

(带你了解源码中的 ThreadLocal)

**ThreadLocaL是一个泛型类，一个线程中有可以有多个ThreadLocal，一个线程只有一个ThreadLocalMap，要获取数据时，先通过当前线程名获取当前线程ThreadLocalMap，ThreadLocalMap保存的是键值对，键值为不同的ThreadLocal对象，值为相应ThreadLocal保存的数据，获取数据时以ThreadLocal为键值去ThreadLocalMap中获取对应的数据**

#### Q1：什么时候采用ThreadLocal?

**🡪请谈谈 ThreadLocal 是怎么解决并发安全的？**

ThreadLocal 的核心其实也就是在 get() 和 set()，搞懂这两个方法的流程原理，那么也就基本理解 ThreadLocal 这个东西的原理了。

#### Q2:ThreadLocal 如何做到同一个对象却维护着不同线程的数据副本呢？

ThreadLocal 是一个线程内部的数据存储类，1.通过它可以在指定的线程中存储数据，数据存储以后，只有在指定线程中可以获取到存储的数据，对于其他线程来说则无法获取到数据。2.采用 ThreadLocal 可以让监听器作为线程内的全局对象而存在，在线程内部只要通过 get 方法就可以获取到监听器。

某些数据如果是以线程为作用域并且不同线程可以互不影响、独立运作的时候，那么就可以采用 ThreadLocal 了。

ThreadLocal 做到同一对象维护不同线程的数据副本。无外乎就是将数据保存在各自的线程中，这样不同线程的数据自然相互不影响。然后存值时再以当前 ThreadLocal 实例对象为 key，这样即使同一线程中，不同 ThreadLocal 虽然使用同一个容器，但 key 不一样，取值时也就不会相互影响。

#### Q3:ThreadLocal副作用及如何避免？

由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，发生GC时弱引用Key会被回收，而Value不会回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。

如何避免泄漏：在调用ThreadLocal的get()、set()方法时完成后再调用remove方法，将Entry节点和Map的引用关系移除，这样整个Entry对象在GC Roots分析后就变成不可达了，下次GC的时候就可以被回收。

## 五.spring框架（MVC和Mybatis）

## 六.JVM虚拟机（类加载和内存管理）

## 七.日志与linux（xshell）

#### Q1：如何在用日志定位解决问题？

**※聊联调定位,见项目**

#### Q2：怎么日志定位？常用哪些日志语句?

**🡪还用过哪些shell语句？**

#### Q3：如何在spring应用中使用slf4j

**🡪日志对象log为什么定义为静态？**

**🡪引申设计模式层面：Facade模式&通过AOP和IOC实现日志的松耦合**

#### Q4:说说你理解的日志框架

**🡪 日志作用？ 🡪 引申到dubbo**

#### Q5：linux 的命令用过哪些？

**🡪 du、top、free、netstat**

**A1:**遇到过两个问题：1.刚开始接手的项目用slf4j+logback模式，后来要加s要加上log4j报错。后来发现因为logback自身实现了slf4j接口是不需要引入额外的适配器，但log4j比slf4j早，需要引入对应适配器（jul-to-）解决接口不兼容问题（搭配logback就不需要）2.联调时需要看患者号源，但用日志输出对象实例的时候只是打印了对象的hashcode值，数据没有实际意义，后来在实例类里重写了toString方法解决了问题。

**A2:**项目测试环境用的是阿里云自带的日志服务，也用过xshell和docker做日志管理,xshell直接选择服务器可以自动连接。

常用命令：首先cd/切换到日志目录，ls列出所有日志文件，然后用tail和grep联合搜索文件名就可以看到实时打印出来的日志了。

XX.log==项目的log.xml文件里配置的包名/id

除了日志，还用过的命令有：

1. ps -ef |grep tomcat**🡪**kill pid 进程号——遇到启动工程时Tomcat进程端口被占用时，杀掉进程
2. top——查看cpu负载和内存情况
3. stop、start——重启tomcat ./catalina.sh stop
4. tail -f（循环取）/-n(取行数) file|grep "keyword" |grep "…"

tail可不断刷新日志文件,grep可以过滤关键字无关的日志

**A3:**slf4j（Simple Logging Facade for Java）相当于给日志库们解耦的日志接口，我之前做的spring框架项目中它配着log4j或logback日志一起用。第一步是将SLF4J的Logger和LoggerFactory导入。

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

IDEA里只需在使用日志的类或接口加@slf4j注解就能自动导jul包（项目的pom.xml引入依赖）。

第二步是在要打印日志的的类里用LoggerFactory获得logger实例：

Private static final Logger log = LoggerFactory.getLogger(XX.class)

最后，根据需求对相应的日志文件进行配置。

比如在文件里可以配输出位置（console&file）和输出格式(thread，level，msg)

就可以启动工程进行日志打印了。

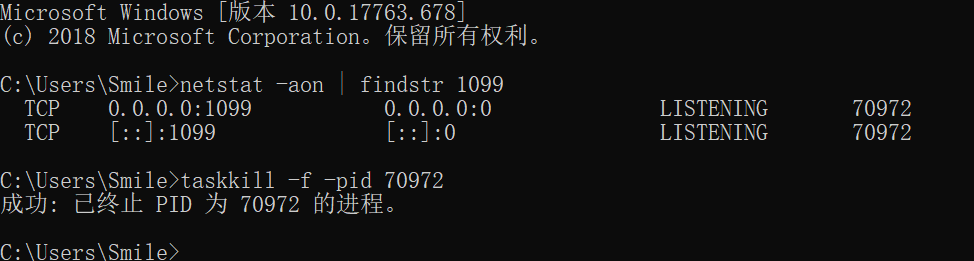
注：Log定义为static变量是因为可以把log对象和当前类绑定避免每次都new一个新对象浪费资源，甚至引发OutOfMemoryError问题；

**A4：**日志框架的体系一般是分为日志门面，日志库和适配器三部分。

日志主要三个方面作用1.定位问题来源，实现快速排查2.监控系统的内存，CPU使用情况3.应用的运行情况比如provider的响应时间，QPS等交互状态

**A5：**

**结束占用端口时用了netstats**

****

## 八.Mysql数据库（事务，索引和锁）

## 九.计算机网络（tcp/ip,http等）与操作系统

## 十.中间件dubbo与git，maven管理工具