面经综合

面向对象的六个基本原则

单一职责、开放封闭、里氏替换、依赖倒置、合成聚合复用、接口隔离

单一职责：一个类只做它该做的事情(高内聚)。在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责

开放封闭；软件实体应当对扩展开放，对修改关闭。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂 在一起，系统将变得复杂而换乱。

里氏替换：任何时候都可以用子类型替换掉父类型。子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。

依赖倒置：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代）

合成聚合引用：优先使用聚合或合成关系复用代码。

接口隔离：接口要小而专，绝不能大而全。臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。

迪米特法则：迪米特法则又叫做最少知识原则，一个对象应当对其他对象尽可能少的了解。

项目中用到的原则：单一职责，开放封闭，合成聚合复用，接口隔离。

Final

修饰变量：

编译器常量：类加载过程完成初始化，编译后代入到任何计算式中，只能是基本类型

运行时常量：基本数据类型或引用数据类型。引用不可变，但引用的对象内容可变。

HasmMap的注意：

HashMap的值不能是可变对象，如果是可变对象的话，对象中的属性改变，则对象的hashCode也进行相应的改变，导致下次无法查找到已存在Map中的数据。

线程安全：

定义：

某个类的行为与其规范一致

不管多个线程是怎样执行顺序和优先级，或是wait(),sleep(),join等控制方式，如果一个类在多线程访问下一切运转正常，并且访问类不需要进行额外的同步处理或者协调，那么我们就认为他是线程安全的。

如何保证线程安全：

对变量使用volitate

对程序段进行加锁

多线程公用一个数据变量需要注意什么?

1. 当我们在线程对象中定义了全局变量，run方法会修改变量时，如果多个线程同时使用该线程对象，那么就会造成全局变量的值同时被修改，造成错误。
2. ThreadLocal是JDK引入的一种机制，他用于解决线程间共享变量，使用ThreadLocal声明的变量，即使在线程中属于全局变量，针对这个线程来讲，这个变量也是独立的。
3. volatile变量每次被线程访问时，都强迫线程从主内存中重读该变量的最新值，而当该变量发生修改变化时，也会强迫线程将最新的值刷回主内存中。这样一来，不同的线程都即使看到该变量的最新值。

ThreadLocal的工作原理

1. Thread类中有一个成员变量属于ThreadLocalMap类（一个定义在ThreadLocal类中的内部类），他是一个Map，他的key是ThreadLocal实例对象。
2. 当为ThreadLocal类的对象set值时，首先获得当前线程的ThreadLocalMap类属性，然后以ThreadLocal类的对象为key，设定value.get值时类似。
3. ThreadLocal变量的活动范围为某县城，是该线程专有的，独自霸占的，对该变量的所有操作均由该线程完成。也就是说，ThreadLocal不是用来解决共享对象的多线程访问的竞争问题的，因为ThreadLocal.set()到线程中的对象应该是线程自己使用的对象，其他线程不需要访问，也访问不到。当线程终止后，这些值会别垃圾回收。

4. ThreadLocal的工作原理据定了：每个线程独自拥有一个变量，并非共享的。

java是否有内存泄漏和内存溢出

1. 静态集合类，使用Set、Vector、HashMap等集合类的时候要特别注意。当这些类被定义成静态的时候，由于他们的生命周期跟应用程序一样长，这样有时候就可能发生内存泄漏。
2. 监听器：在java编程中，我们都需要和监听器打交道，通常一个应用中会用到很多监听器，我们会调用一个空间，注入addXXXListenner（）方法来增加监听器，但往往在释放的到时候却没有去删除这些监听器，从而增加了内存泄漏的机会。
3. 物理连接：一些物理连接，比如数据库连接和网络连接，除非其显示的关闭了连接，否则是不会自动被GC回收的。java数据库连接一般用DataSource.getConnection()来创建，当不再使用时必须用Close（）来释放，因为这些连接是独立于JVM的。对于Resultset和Statement对象可以不进行显示回收，但Connection一定要显式回收，因为Connection在任何时候都无法自动回收，而Connection一旦回收，Resultset和Statement对象就会立即为null。但是如果使用连接池，情况就不一样了，除了要显式的关闭连接，该必须显示的关闭ResultSet statement对象（关闭其中一个，另一个也会关闭），否则就会造成大量的Statement对象无法释放，从而引起内存泄漏。
4. 内部类和外部模块的引用：内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后积累对象没有释放。在调用外部模块的时候，也应该注意防止内存泄漏，如果模块A调用了外部模块B的一个方法，如：public void register(Object o)这个方法就有课额能是的A模块持有传入对象的引用，这时候需要查看b模块是否提供了去除引用的方法，这种情况容易忽略，而且发生内存泄漏的话，还比较难察觉。
5. 单例模式：因为单例对象初始化后将在JVM的整个生存周期内存在，如果他持有一个外部对象的（生命周期比较短）引用，那么这个外部对象就不能被回收，从而导致内存泄漏。

concurrent包下面，都用过什么？

concurrent下面的包：

Executor 用来创建线程池，再实现Callable接口时，添加线程

FeatureTask 此FutureTask的get方法返回的结果类型。

TimeUnit

Semaphore

LinkedBlockingQueue

所用过的类：

Executor

volatile关键字如何保证内存的可见性

volatile关键字的作用

1. 保证内存可见性
2. 防止指令重排
3. 注意：volatile并不保证原子性

内存可见性：

volatile保证可见性的原理是在每次访问变量时都会进行一次刷新，因此每次访问都是主内存中最新的版本。所以volatile关键字的作用之一就是保证变量修改的实时可见性。

当且仅当满足一下条件时，才应该使用volatile变量

对变量的写入操作不依赖变量 的当前值，或者你能确保只有单个线程更新变量的值

该变量没有包含在具有其他变量的不变式中

volatile使用建议

在两个或者更多的线程需要访问的成员上使用volatile。当要访问的变量已在synchronized代码块中，或者为变量时，没必要使用volatile

由于使用volatile屏蔽掉了jvm中必要的代码优化，所以在效率上比较低，因此一定要在必要时才使用这个关键字

volatile和synchronized区别

volatile不会进行加锁操作

volatile变量是一种稍弱的同步机制，在访问volatile变量时不会执行加锁操作，因此也就不会执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。

volatile变量作用类似与同步变量读写操作

从内存可见性的角度看，写入volatile变量相当于退出同步代码块，而读取volatile变量相当于进入同步代码块。

volatile不如synchronized安全：

在代码中如果过度依赖volatile变量来控制状态的可见性，通常会比使用所的代码更脆弱，也更难以理解。仅当volatile变量能简化代码的实现以及能对同步策略进行验证时，才应该使用它，一般来说同步机制更安全些。

volatile无法同时保证内存可见性和原子性

加锁机制既可以保证可见性也可以保证原子性，而volatile变量只能保证可见性，原因是生命为volatile的简单变量如果当前值与该变量以前的值相关，那么volatile关键字不起作用。

sleep和wait分别是那个类的方法，有什么区别?

sleep和wait

sleeep是Thread类的方法

wait是Object类的方法

有什么区别：

sleep()方法（休眠）是线程类的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态）

wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool）,只有调用对象的notify()方法时才能唤醒等待池中的线程进入等锁池，如果线程重新获得对象的锁就可以进入就绪状态。

synchronizde与lock的区别，使用场景。看过synchronized的源码没?

synchronized与lock的区别

sychronized(隐式锁)：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。

lock(显式锁)：需要制定起始位置和终止位置。一般使用ReentrantLock类作为锁，多个线程中必须要使用一个ReentrantLock类作为对象才能保证锁生效，且在加锁和解锁处需要通过lock()和unlock()显式之处。所以一般会在finaly块中写unlock防止死锁。

synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。在java1.5中，synchronize是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用java提供的Lock对象，性能更高一些。但是到了java1.6，发生了变化。synchronized在语义上很清晰，可以进行很多优化，有适应性自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在java1.6上synchronize的性能并不比Lock差。

synchronized原始采用的是CPU悲观锁机制，即线程获得独占锁。独占锁意味着其他线程 只能依靠阻塞来等待线程释放锁。Lock采用的乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作。

synchronized底层如何实现的？用在代码块和方法上有什么区别

用在代码块和方法上的区别：

synchronized用在代码块锁的是调用该方法的对象（this）,也可以选择锁住任何一个对象

synchronized用在方法上锁的是调用该方法的对象

synchronized用在代码块可以减小锁的粒度，从而提高并发性能。

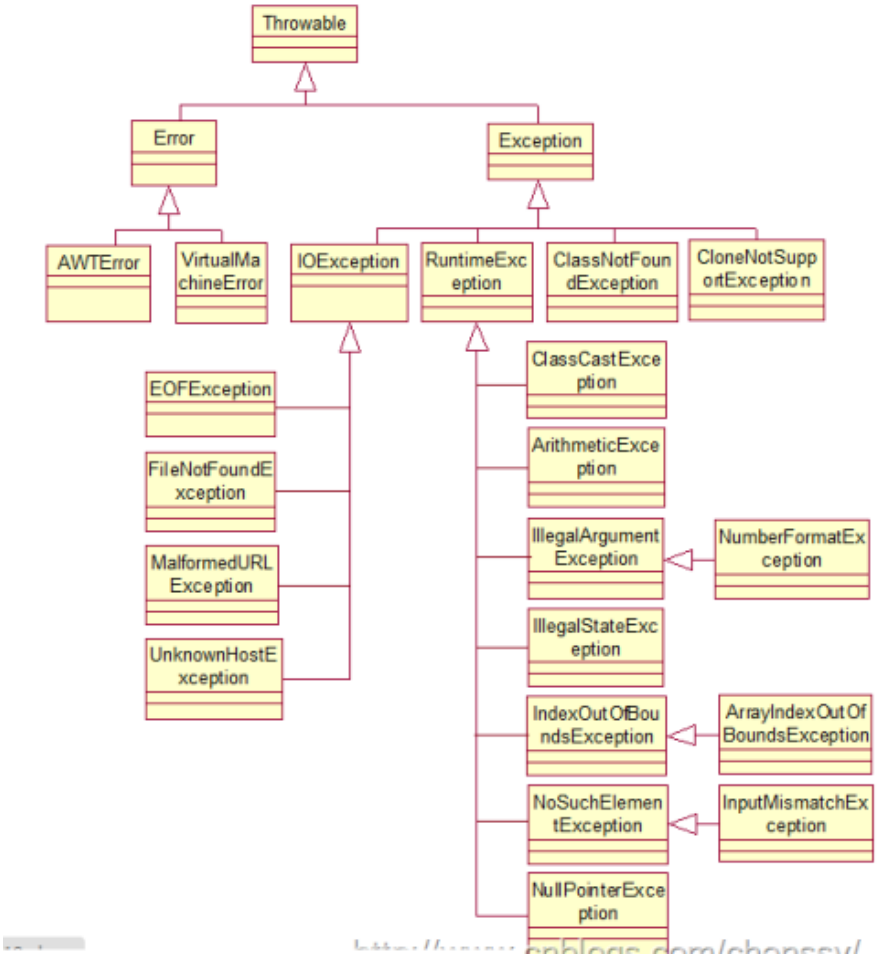
无论用在代码块上，还是用在方法上，都是获取对象的锁；每一个对象只有一个锁与之相关联；实现同步需要很大的系统开销作为代价，甚至可能造成死锁，所以尽量避免无谓的同步控制。

synchronized和static synchronized的区别

synchronized是对类的当前实例进行加锁，防止其他线程同时访问该类的该实例的所有synchronized块，同一个类的两个不同实例就没有这种约束

那么static synchronized恰好就是要控制类的所有实例的访问了，static synchronized是限制线程同时访问jvm中该类的所有实例同时访问对应的代码块。

java中异常的体系



java中NIO，BIO，AIO分别是什么?

BIO：

同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个链接不做任何事情就会造成不必要的开销，当然可以通过线程池机制改善。

BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前是唯一选择，但程序直观简单易理解。

NIO

同步非阻塞，服务器实现模式为多个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

NIO方式适用于链接数目多且连接比较短的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，jdk1.4开始支持。

AIO

异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了在通知服务器应用去启动线程进行处理。

AIO方式使用于连接数目较多且连接比较长的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，jdk1.7开始支持。

匿名内部类是什么？如何访问骑在外面定义的变量？

匿名内部类是什么？

匿名内部类是没有访问修饰符的

所以当所在方法的形参需要被匿名内部类使用，那么这个形参就必须为final

匿名内部类是没有构造方法的，因为他连名字都没有何来构造方法。

如何访问其在外面定义的变量

所以当所在方法的形参需要被匿名内部类使用，那么这个形参就必须是final

为什么要实现内存模型

内存模型就是为了在现代计算机平台中保证程序可以正确性执行，但是不同的平台的实现是不同的。

编译器中生成的指令顺序，可以与源代码中的顺序不同。

编译器可能把变量保存在寄存器中而不是内存中

处理器可以采用乱序或并行等方式来执行指令。

缓存可能会改变将写入变量提交到主内存的次序。

保存在处理器本地缓存中的值，对其他处理器是不可见的。

Redis的存储结构，或者说如何工作的，与mysql的区别？有哪些数据类型？

Redis的数据结构：

String:可以是字符串、整数或者浮点数

List:一个链表，链表上的每个节点都包含一个字符串。

Set:包含字符串的无序收集器，并且包含的每个字符串都是独一无二，各不相同的。

HAST:包含键值对的无序散列表。

ZSET:字符串成员与浮点数分支之间的有序映射，元素的排列顺序由分值的大小确定。

MyISAM和InnoDB引擎的区别

主要区别： ○ MyISAM是非事务安全型的，而InnoDB是事务安全型的。 ○ MyISAM锁的粒度是表级，而InnoDB支持行级锁定。 ○ MyISAM支持全文类型索引，而InnoDB不支持全文索引。 ○ MyISAM相对简单，所以在效率上要优于InnoDB，小型应用可以考虑使用MyISAM。 ○ MyISAM表是保存成文件的形式，在跨平台的数据转移中使用MyISAM存储会省去不少的麻烦。 ○ InnoDB表比MyISAM表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表（alter table tablenametype=innodb）。

应用场景：

MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。 ○ InnoDB用于事务处理应用程序，具有众多特性，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。

JVM垃圾处理方法（标记清除、复制、标记整理）

• 标记-清除算法 ○ 标记阶段：**先通过根节点，标记所有从根节点开始的对象，未被标记的为垃圾对象** ○ 清除阶段：清除所有未被标记的对象 • 复制算法 ○ 将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，然后清除正在使用的内存块中的所有对象。 • 标记-整理 ○ 标记阶段：先通过根节点，标记所有从根节点开始的可达对象，为被标记的为垃圾对象 ○ 整理阶段：将所有的存活对象压缩到内存的一段，之后清理边界所有的空间 • 三种算法的比较 ○ 效率：复制算法 > 标记/整理算法 > 标记/清除算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如 此）。 ○ 内存整齐度：复制算法=标记/整理算法>标记/清除算法。 ○ 内存利用率：标记/整理算法=标记/清除算法>复制算法。

JVM如何GC，新生代，老年代，持久代都存储那些东西，以及各个区的作用?

新生代：

在方法中取new一个对象，这方法调用完毕后，对象就会被回收，这就是一个典型的新生代对象。

老年代：

在新生代中经历了N此垃圾回收后仍然存活的对象就会被放到老年代中，而且大对象直接进入老年代。

在Survivor空间不够用时，需要依赖于老年代进行分配担保，所以大对象直接进入老年代。

永久带：

即方法区。

作为GC Root的对象

1. java虚拟机栈中的对象
2. 方法去中的静态变量
3. 方法区中的常量引用对象
4. 方法区中的JNI（Native）引用对象。

什么时候进行MinGC，FullGC

• MinGC ○ 新生代中的垃圾收集动作，采用的是复制算法 ○ 对于较大的对象，在Minor GC的时候可以直接进入老年代 • FullGC ○ Full GC是发生在老年代的垃圾收集动作，采用的是标记-清除/整理算法。 ○ 由于老年代的对象几乎都是在Survivor区熬过来的，不会那么容易死掉。因此Full GC发生的次数不会有Minor GC那么频繁，并且Time(Full GC)>Time(Minor GC)

Student s = new Student();在内存中做了哪些事情?

1. 加载Student.class文件进内存
2. 在栈内存为S开辟空间
3. 在堆内存为学生对象开辟空间
4. 对学生对象的成员变量进行默认初始化
5. 对学生对象的成员变量进行显示初始化
6. 通过构造方法对学生对象的成员变量赋值
7. 学生对象初始化完毕，把对象地址赋值给S变量。

Servlet的生存周期

servlet接口定义了5个方法，其中前三个方法与Servlet声明周期相关：

void init(ServletConfig config) throws ServletException

void service(ServletRequest req,ServletResponse resp) throws Serlet,java.io.IOException

void destory()

java.lang.String getServletInfo()

ServletConfig getServletConfig()

web容器加载Servlet并将其实例化后，Servlet生命周期开始，容器运行期init()方法进行Servlet的初始化；请求到达时调用Servlet的service()方法，service（）方法会根据需要调用与请求对应的doGet()或doPost等方法；当服务器关闭或项目被卸载的时候服务器将Servlet实例销毁，此时调用Servlet的destory（）方法。

jsp和Servlet的区别

Servlet是一个特殊的java程序，他运行于服务器的JVM中，能够依靠服务器的支持向浏览器提供显示内容。JSP本质上是Servlet的一种建议形式，JSP会被服务器处理成一个类似于Servlet的java程序，可以简化页面内容的生成。Servlet和JSP主要的不同点是，Servlet的应用逻辑在java文件中，并且完全从表示层中的HTML文件分离开来。而JSP的情况是java和HTML可以组合成一个扩展名为.jsp的文件。有人说，Servlet就是在Java中写HTML，而JSP就是在HTML中写Java代码，当然这个说法是很片面且不够准确的。JSP侧重于视图，Servlet更侧重于控制逻辑，在MVC架构模式中，JSP适合充当视图（view）而Servlet适合充当控制器。

SpringIOC、AOP的理解以及实现的原理

Spring IOC

IoC叫控制反转，是Inversion of Control的缩写，DI（Dependency Injection）叫依赖注入，是对IoC更简单的诠释。控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。DI是对IoC更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。 举个例子：一个类A需要用到接口B中的方法，那么就需要为类A和接口B建立关联或依赖关系，**最原始的方法是在类A中创建一个接口B的实现类C的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类A中定义好用于关联接口B的方法（构造器或setter方法），将类A和接口B的实现类C放入容器中，通过对容器的配置来实现二者的关联。**

Spring IOC的实现原理

通过反射创建实例

获取需要注入的接口实现类并将其赋值给该接口。

```
public interface BeanFactory {

    //对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象，
    //如果需要得到工厂本身，需要转义
    String FACTORY_BEAN_PREFIX = "&";

    //根据bean的名字，获取在IOC容器中得到bean实例
    Object getBean(String name) throws BeansException;

   //根据bean的名字和Class类型来得到bean实例，增加了类型安全验证机制。

    Object getBean(String name, Class requiredType) throws BeansException;
```

```
    Object getBean(String name, Class requiredType) throws BeansException;


    //提供对bean的检索，看看是否在IOC容器有这个名字的bean
    boolean containsBean(String name);

    //根据bean名字得到bean实例，并同时判断这个bean是不是单例
    boolean isSingleton(String name) throws
NoSuchBeanDefinitionException;

    //得到bean实例的Class类型
    Class getType(String name) throws NoSuchBeanDefinitionException;

    //得到bean的别名，如果根据别名检索，那么其原名也会被检索出来
    String[] getAliases(String name);

}
```

https://www.cnblogs.com/ITtangtang/p/3978349.html(未看)

Spring AOP

AOP指一种程序设计泛型，该泛型以一种称为切面（aspect）的语言构造为基础，切面是一种新的模块化机制，用来描述分散在对象、类或方法中的横切关注点。

“横切关注”是会影响到整个应用程序的关注功能，他跟正常的业务逻辑是正交的，没有必然联系，但是几乎所有的业务逻辑都会涉及这些关注功能。通常，事务、日志、安全性等关注就是应用中横切关注功能。

Spring AOP实现原理

动态代理（利用反射和动态编译将代理模式变成动态的）

JDK动态代理

JDKProxy放回动态代理类，是目标类所实现接口的另一实现版本，他实现了对目标类的代理（如同UserDAOProxy与UserDAOImpl）

cglib动态代理

CGLibProxy返回动态代理类，则是目标代理类的一个子类（代理类扩展了UserDaoImpl）

IOC容器的加载过程

1. 创建IOC配置文件的抽象资源

2. 创建一个BeanFactory
3. 把读取配置信息的BeanDefinitionReader，这里是XMLBeanDefinitionReader配置给BeanFactory
4. 从定义好的资源位置读入配置信息，具体解析过程由XmlBeanDefinitionReader来完成，这样就完成整个载入bean定义的过程。

动态代理与CGlib实现的区别

JDK动态代理只能对实现了接口的类生成代理，而不能针对类. • CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法因为是继承，所以该类或方法最好不要声明成 final。 • JDK代理是不需要以来第三方的库，只要JDK环境就可以进行代理 • CGLib 必须依赖于CGLib的类库，但是它需要类来实现任何接口代理的是指定的类生成一个子类，覆盖其中的方法，是一种继承

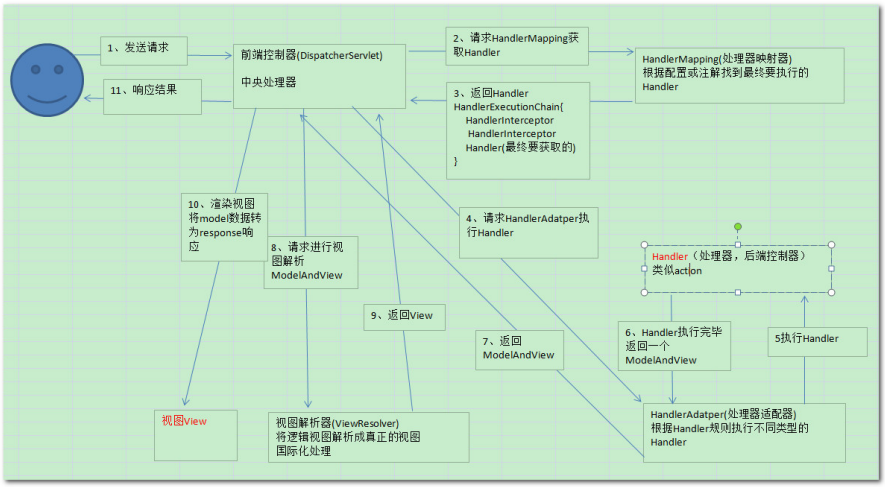
Spring AOP实现AOP的原理

http://blog.csdn.net/zbuger/article/details/51011117(未看)

Hibernate一级缓存和二级缓存之间的区别

• Hibernate的Session提供了一级缓存的功能，默认总是有效的，当应用程序保存持久化实体、修改持久化实体时，Session并不会立即把这种改变提交到数据库，而是缓存在当前的Session中，除非显示调用了Session的flush()方法或通过close()方法关闭Session。通过一级缓存，可以减少程序与数据库的交互，从而提高数据库访问性能。 • SessionFactory级别的二级缓存是全局性的，所有的Session可以共享这个二级缓存。不过二级缓存默认是关闭的，需要显示开启并指定需要使用哪种二级缓存实现类（可以使用第三方提供的实现）。一旦开启了二级缓存并设置了需要使用二级缓 存的实体类，SessionFactory就会缓存访问过的该实体类的每个对象，除非缓存的数据超出了指定的缓存空间。 • 一级缓存和二级缓存都是对整个实体进行缓存，不会缓存普通属性，如果希望对普通属性进行缓存，可以使用查询缓存。查询缓存是将HQL或SQL语句以及它们的查询结果作为键值对进行缓存，对于同样的查询可以直接从缓存中获取数据。查询缓存默认也是关闭的，需要显示开启。

Spring MVC的工作原理



1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理映射器
3. 处理器映射器找到具体的处理器（可以根据XML配置、注解进行查找），生成处理器对象及处理器拦截器（如果有则生成）一并返回给DispatcherServlet。
4. DispatcherServlet调用HandlerAdapter处理器适配器
5. HandlerAdapter经过适配调用具体的处理器（Controller也叫后端控制器）
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewResolver视图解释器。
9. ViewResolver解析后返回具体的View
10. DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）
11. DispatcherServlet相应客户

组件说明：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。

ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel等。

简述Hibernate常见的优化策略

• 制定合理的缓存策略（二级缓存、查询缓存）。 • 采用合理的Session管理机制。 • 尽量使用延迟加载特性。 • 设定合理的批处理参数。 • 如果可以，选用UUID作为主键生成器。 • 如果可以，选用基于版本号的乐观锁替代悲观锁。 • 在开发过程中, 开启hibernate.show_sql选项查看生成的SQL，从而了解底层的状况；开发完成后关闭此选项。 • 考虑数据库本身的优化，合理的索引、恰当的数据分区策略等都会对持久层的性能带来可观的提升，但这些需要专业的DBA（数据库管理员）提供支持。

如何理解分布式锁?

分布式锁，是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。

线程同步与阻塞的关系？同步一定阻塞吗？阻塞一定同步吗?

线程同步与阻塞的关系

线程同步与阻塞没有一点关系

同步和异步关注的是消息通信机制。所谓同步，就是在发出一个“调用”时，在没有得到结果之前，该“调用”不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由“调用者”主动等待这个“调用”的结果。而异步则是相反，“调用”翻出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在“调用”发出后，“被调用者”通过状态、通知来通知调用者，或通过回调函数来处理这个调用。

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）是的状态。阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

操作系统如何进行分页调度?

Linux下如何进行进程调度的?

Linux你常用的命令有哪些?

常用的hash算法有哪些?

1. 加法Hash：把输入元素一个一个的加起来构成最后的结果
2. 位运算Hash：这类型Hash函数通过利用各种位运算来充分的混合输入元素
3. 乘法Hash：这种类型的Hash函数利用了乘法的不相关性（乘法的这种性质，最有名的莫过于平方取头尾的随机数生成算法，虽然这种算法效果并不好）；jdk5.0里面的String类的hashCode()方法也使用乘法Hash；32位FNV算法
4. 除法Hash；除法和乘法一样，同样具有表面上看起来的不相关性。不过，因为除法太慢，这种方式几乎找不到真正的应用
5. 查表Hash；查表Hash最有名的例子莫过于CRC系列算法。虽然CRC系列算法本身并不是查表，但是，查表是它的一种最快的实现方式。查表Hash中有名的例子有：Universal Hashing和Zobrist Hashing。他们的表格都是随机生成的。
6. 混合Hash；混合Hash算法利用了以上各种方式。各种常见的Hash算法，比如MD5、Tiger都属于这个范围。它们一般很少在面向查找的Hash函数里面使用

什么是一致性hash？用来解决什么问题？

在设计分布式cache系统的时候，我们需要让key的分布均衡，并且在增加cache server后，cache的迁移做到最少。

Spring事务

事务的基本原理

spring事务的本质其实就是对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。

Spring事务的传播事务

所谓spring事务的传播属性，就是定义在存在多个事务同时存在的时候，spring应该如何处理这些事务的行为。这些属性在TransactionDefinition中定义，具体常量的解释见下表：

PROPAGATION_REQUIRED 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择，也是Spring默认的事务的传播。

PROPAGATION_REQUIRES_NEW:新建事务，如果当前存在事务，就把当前事务挂起。新建的事务和将被挂起的事务没有任何关系，是两个独立的事务，外层事务失败回滚之后，不能回滚内层事务执行的结果，内层事务失败抛出异常，外层事务驳货，也可以不做处理回滚操作。

PROPAGATION_SUPPORTS:支持当前事务，如果当前没有事务，就以非事务的方式运行。

PROPAGATION_MANDATORY:支持当前事务，如果当前没有事务，则抛出异常。

PROPAGATION_NOT_SUPPORTED :以非事务的方式执行，如果当前存在事务，就把事务挂起。

PROPAGATION_NEVER 以非事务的方式运行，如果当前存在事务，则抛出异常。

PROPAGATION_NESTED:如果一个活动事务存在，则运行在一个嵌套的事务中。如果没有事务，则按REQUIRED属性执行。他使用了一个单独的事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事物造成影响。他只对DataSourceTransactionManager事务管理器奇效。

Spring中的隔离级别

常量:

ISOLATION_DEFAULT:这是个PlatfromTransactionManager默认隔离级别，使用数据库默认的隔离级别。另外四个与JDBC四个隔离级别相对应。

ISOLATION_READ_UNCOMMITTED 这是事务最低的隔离级别，它充许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。 ISOLATION_READ_COMMITTED 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。ISOLATION_REPEATABLE_READ 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。 ISOLATION_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。