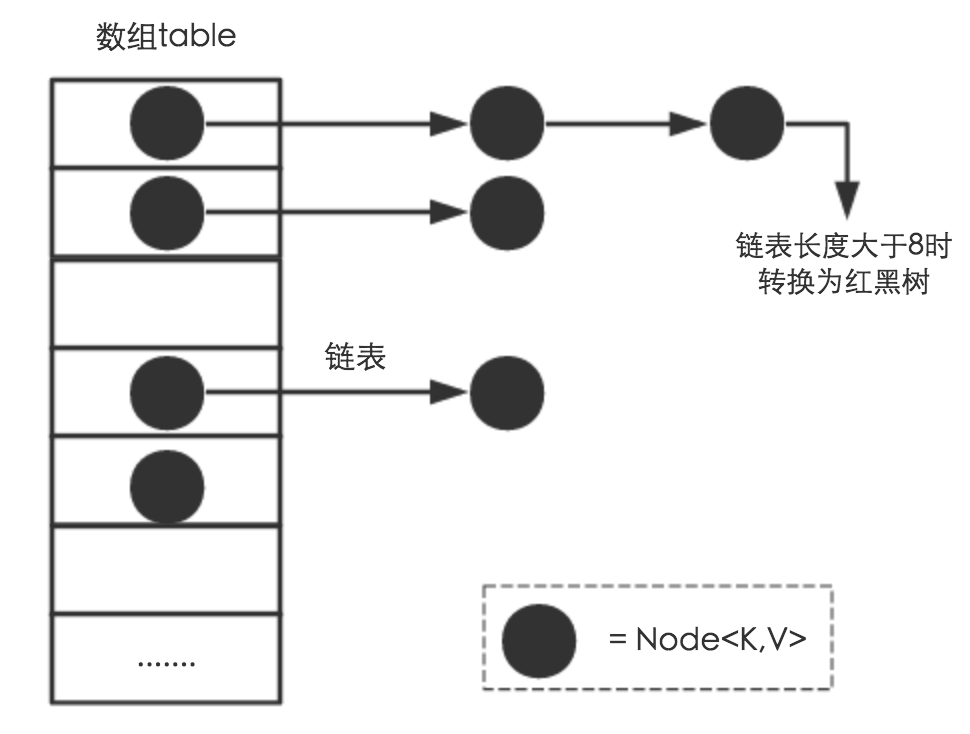
**1、HashMap源码，实现原理，JDK8以后对HashMap做了怎样的优化。**

　　答：HashMap是基于哈希表的Map接口的非同步实现，提供所有可选的映射操作，并允许使用null值和null键，不保证映射的顺序；HashMap是一个“链表散列”的数据结构，即数组和链表的结合体；它的底层就是一个数组结构，数组中的每一项又是一个链表，每当新建一个HashMap时，就会初始化一个数组；

可参考博客：[彻底搞懂JAVA集合HashMap,HashTable,ConcurrentHashMap之关联](https://www.cnblogs.com/wang-meng/p/5808006.html)

　　而在JDK8中引入了红黑树的部分，当存入到数组中的链表长度大于（默认）8时，即转为红黑树；利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，想了解更多红黑树数据结构的工作原理可以参考<http://blog.csdn.net/v_july_v/article/details/6105630>。



可参考博客：[JAVA8系列之重新认识HashMap](https://tech.meituan.com/java-hashmap.html)

**2、HashMap的扩容是怎样扩容的，为什么都是2的N次幂的大小。**

　　答：可以参考上文  JAVA8系列之重新认识HashMap  有详细的讲解

**3、HashMap，HashTable，ConcurrentHashMap的区别**

　　答：

　　a、HashMap是非线程安全的，HashTable是线程安全的。

　　b、HashMap的键和值都允许有null值存在，而HashTable则不行。

　　c、因为线程安全的问题，HashMap效率比HashTable的要高。

　　HashMap：它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap。

　　 Hashtable：Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。

**4、极高并发下HashTable和ConcurrentHashMap哪个性能更好，为什么，如何实现的。**

　　答：当然是ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁，而HashTable则使用的是方法级别的锁；因此在新版本中一般不建议使用HashTable，不需要线程安全的场合可以使用HashMap，而需要线程安全的场合可以使用ConcurrentHashMap；

**5、HashMap在高并发下如果没有处理线程安全会有怎样的隐患，具体表现是什么。**

　　答：可能造成死循环，具体表现链表的循环指向；

**6、JAVA中四种修饰符的限制范围。**

　　private：修饰的成员只能在同类中别访问，而在同包、子类和其他包中都不能被访问

　　public：修饰的成员在同类、同包、子类（继承自本类）、其他包都可以访问

　　protected：修饰的成员在同类、同包、子类中可以访问，其他包中不能被访问

　　default：修饰的成员在同类、同包中可以访问，但其他包中不管是不是子类都不能被访问

**7、Object中的方法**

　　构造函数

　　hashCode()：用户获取对象的hash值，用于检索

　　queals()：用于确认两个对象是否相等；补充，哈希值相同的对象不一定equals()，但equals()的两个对象，hash值一定相等

　　toString()：返回一个String对象，用来标识自己

　　getClass()：返回一个class对象，打印的格式一般为  class package.name.xxx，经常用于java的反射机制

　　clone()：用来另存一个当前存在的对象

　　finalize()：垃圾回收的时候回用到，匿名对象回收之前会调用到

　　wait()：用于让当前线程失去操作权限，当前线程进入等待序列

　　wait(long)、wait(long,int)：用户设定下一次获取锁的距离当前释放锁的间隔时间

　　notify()：用于随机通知一个持有对象锁的线程获取操作的权限

　　notifyAll()：用于通知所有持有对象锁的线程获取操作权限

**8、接口和抽象类的区别**

　　答：一个类可以实现多个接口，但只能继承一个抽象类；抽象类可以包含具体的方法，接口所有的方法都是抽象的（JDK8开始新增功能接口中有default方法）；抽象类可以声明和使用字段，接口则不能，但可以创建静态的final常量；抽象类的方法可以是protected、public、private或者默认的package，接口的方法都是public；抽象类可以定义构造函数，接口不能；接口被声明为public，省略后，包外的类不能访问接口；

**9、动态代理的两种方式，以及区别**

　　答：jdk动态代理和cglib动态代理；

　　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称final，final可以阻止继承和多态；

**10、java序列化的方式**

　　答：实现Serializable接口、实现Externalizable接口（一般只希望序列化一部分数据，其他数据都使用transient修饰的话有点麻烦，这时候可以使用externalizable接口，指定序列化的属性）

**11、传值和传引用的区别，java是怎么样的，有没有传值传引用**

　　答：首先，java中是没有指针的，只存在值传递；而我们经常看到对于对象的传递似乎有点像引用传递，可以改变对象中的某个属性的值，请不要被这个假象蒙蔽了双眼，实际上这个传入函数的值是对象引用的拷贝，即传递的是引用的地址值，所以还是按值传递；

　　传值调用时，改变的是形参的值，并没有改变实参的值，实参的值可以传递给形参，但是这个传递是单向的，形参不能传递会实参；

　　传引用调用时，如果参数是对象，无论是对象做了何种操作，都不会改变实参对象的引用，但是如果改变了对象的内容，就会改变实参对象的内容；

**12、@transactional注解在什么情况下会失效，为什么。**

　　答：一个目标对象的方法调用改目标对象的另外一个方法时，即使被调用的方法已使用了@Transactional注解标记，事务也不会有效执行；Spring的官方说明在代理下（默认或者配置为proxy-targer-class="true"），只有当前代理类的外部方法调用注解方法时代理才会被拦截。

数据结构和算法

**1、B+树**

　　参考：[B+树介绍](https://www.cnblogs.com/wade-luffy/p/6292784.html)

**2、八大排序算法**

　　参考：[八大排序算法JAVA实现](https://www.cnblogs.com/lxf20061900/p/5063977.html)

**3、一致性Hash算法，一致性Hash算法的应用**

　　答：一致性hash算法是一个负载均衡算法，可以用在分布式缓存、数据库的分库分表等场景，还可以应用在负载均衡器中作为负载均衡算法。在多台服务器时，对于某个请求资源通过hash算法，映射到某一台服务器，当增加或者减少一台服务器时，可能会改变这些资源对应的hash值，这样可能导致一部分缓存或者数据的丢失。一致性hash就是尽可能在将同一个资源请求到同一台服务器中；

JVM

**1、JVM的内存结构**

　　答：主要分为三大块 堆内存、方法区、栈；栈又分为JVM栈、本地方法栈

　　　　堆（heap space），堆内存是JVM中最大的一块，有年轻代和老年代组成，而年轻代又分为三分部分，Eden区，From Survivor，To Survivor，默认情况下按照8:1:1来分配

　　　　方法区（Method area），存储类信息、常量、静态变量等数据，是线程共享的区域

　　　　程序计数器（Program counter Register），是一块较小的内存空间，是当前线程所执行的字节码的行号指示器

　　　　JVM栈（JVM stacks），也是线程私有的，生命周期与线程相同，每个方法被执行时都会创建一个栈帧，用于存储局部变量表、操作栈、动态链接、方法出口等信息

　　　　本地方法栈（Native Mthod Stacks）,为虚拟机使用的native方法服务

**2、关于垃圾回收和常见的GC算法**，请参考：[GC专家系列-理解java垃圾回收](https://segmentfault.com/a/1190000004233812)

多线程

**1、JAVA实现多线程的几种方式**

　　a、继承Thread类实现

public class MyThread extends Thread {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

MyThread myThread1 = new MyThread();

MyThread myThread2 = new MyThread();

myThread1.start();

myThread2.start();

　　b、实现Runnable接口

　　如果自己的类已经extends另一个类，就无法直接extends Thread，此时，必须实现一个Runnable接口，如下：

public class MyThread extends OtherClass implements Runnable {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

MyThread myThread = new MyThread();

Thread thread = new Thread(myThread);

thread.start();

　　c、使用ExecutorService、Callable、Future实现有返回结果的多线程

import java.util.concurrent.\*;

import java.util.Date;

import java.util.List;

import java.util.ArrayList;

/\*\*

\* 有返回值的线程

\*/

@SuppressWarnings("unchecked")

public class Test {

public static void main(String[] args) throws ExecutionException,

InterruptedException {

System.out.println("----程序开始运行----");

Date date1 = new Date();

int taskSize = 5;

// 创建一个线程池

ExecutorService pool = Executors.newFixedThreadPool(taskSize);

// 创建多个有返回值的任务

List<Future> list = new ArrayList<Future>();

for (int i = 0; i < taskSize; i++) {

Callable c = new MyCallable(i + " ");

// 执行任务并获取Future对象

Future f = pool.submit(c);

// System.out.println(">>>" + f.get().toString());

list.add(f);

}

// 关闭线程池

pool.shutdown();

// 获取所有并发任务的运行结果

for (Future f : list) {

// 从Future对象上获取任务的返回值，并输出到控制台

System.out.println(">>>" + f.get().toString());

}

Date date2 = new Date();

System.out.println("----程序结束运行----，程序运行时间【"

+ (date2.getTime() - date1.getTime()) + "毫秒】");

}

}

class MyCallable implements Callable<Object> {

private String taskNum;

MyCallable(String taskNum) {

this.taskNum = taskNum;

}

public Object call() throws Exception {

System.out.println(">>>" + taskNum + "任务启动");

Date dateTmp1 = new Date();

Thread.sleep(1000);

Date dateTmp2 = new Date();

long time = dateTmp2.getTime() - dateTmp1.getTime();

System.out.println(">>>" + taskNum + "任务终止");

return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";

}

}

**2、Callable和Future**

　　答：Callable接口类似于Runnable，但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable更强大，被线程执行以后，可以返回值，这个返回值就是通过Future拿到，也就是说，Future可以拿到异步执行任务的返回值，可以看以下例子：

import java.util.Random;

import java.util.concurrent.Callable;

import java.util.concurrent.FutureTask;

public class Test {

public static void main(String[] args) {

Callable<Integer> callable = new Callable<Integer>() {

@Override

public Integer call() throws Exception {

return new Random().nextInt(100);

}

};

FutureTask<Integer> futureTask = new FutureTask<Integer>(callable);

new Thread(futureTask).start();

try {

Thread.sleep(1000);

System.err.println(futureTask.get());

} catch (Exception e) {

e.printStackTrace();

}

}

}

ExecutorService继承自Executor，目的是为我们管理Thread对象，从而简化并发变成，Executor使我们无需显示的去管理线程的声明周期，是JDK5之后启动任务的首选方式。

执行多个带返回值的任务，并取得多个返回值，代码如下：

import java.util.concurrent.Callable;

import java.util.concurrent.CompletionService;

import java.util.concurrent.ExecutorCompletionService;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class CallableAndFuture {

public static void main(String[] args) {

ExecutorService threadPool = Executors.newCachedThreadPool();

CompletionService<Integer> cs = new ExecutorCompletionService<Integer>(threadPool);

for( int i = 0; i < 5; i++ ){

final int taskId = i;

cs.submit(new Callable<Integer>() {

@Override

public Integer call() throws Exception {

return taskId;

}

});

}

for( int i = 0; i < 5; i++ ){

try {

System.err.println(cs.take().get());

} catch (Exception e) {

e.printStackTrace();

}

}

}

}

**3、线程池的参数有哪些，在线程池创建一个线程的过程**

　　corePoolSize：核心线程数，能够同时执行的任务数量

　　maximumPoolSize：除去缓冲队列中等待的任务，最大能容纳的任务数（其实就是包括了核心线程池的数量）

　　keepAliveTime：超出workQueue的等待任务的存活时间，就是指maximumPoolSize里面的等待任务的存活等待时间

　　unit：时间单位

　　workQueue：阻塞等待线程的队列，一般使用new LinkedBlockingQueue()这个，如果不指定容量，会一直往里添加，没有限制，workQueue永远不会满，一般选择没有容量上限的队列

　　threadFactory：创建线程的工厂，使用系统默认的类

　　handler：当任务数超过maximumPoolSize时，对任务的处理策略，默认策略是拒绝添加

　　执行流程：当线程数小于corePoolSize时，每添加一个任务，则立即开启线程执行；当corePoolSize满的时候，后面添加的任务将放入缓冲队列workQueue等待；当workQueue满的时候，看是否超过maximumPoolSize线程数，如果超过，则拒绝执行，如果没有超过，则创建线程理解执行；

1 import java.util.concurrent.Executors;

2 import java.util.concurrent.LinkedBlockingQueue;

3 import java.util.concurrent.ThreadPoolExecutor;

4 import java.util.concurrent.TimeUnit;

5

6 /\*\*

7 \* 对线程池进行管理和封装

8 \* @author guoqing

9 \*

10 \*/

11 public class ThreadPoolManager {

12

13 private static ThreadPoolManager mInstance = new ThreadPoolManager();

14 private ThreadPoolExecutor executor;

15

16 private int corePoolSize; //核心线程池数量，表示能够同时执行的任务数量

17 private int maximumPoolSize; //最大线程池数量，其实是包含了核心线程池数量在内的

18 private long keepAliveTime = 1; //存活时间，表示最大线程池中等待任务的存活时间

19 private TimeUnit unit = TimeUnit.HOURS; //存活时间的时间单位

20

21 public static ThreadPoolManager getInstance() {

22 return mInstance;

23 }

24

25 private ThreadPoolManager() {

26 //核心线程数量的计算规则：当前设备的可用处理器核心数\*2+1，能够让cpu得到最大效率的发挥

27 corePoolSize = Runtime.getRuntime().availableProcessors()\*2+1;

28 maximumPoolSize = corePoolSize; //虽然用不到，但是不能为0，否则会报错

29 //线程池机制：领工资的机制

30 executor = new ThreadPoolExecutor(corePoolSize,

31 maximumPoolSize,

32 keepAliveTime,

33 unit,

34 new LinkedBlockingQueue<Runnable>(), //缓冲队列，超出核心线程池的任务会被放入缓冲队列中等待

35 Executors.defaultThreadFactory(), //创建线程的工厂类

36 new ThreadPoolExecutor.AbortPolicy() //当最大线程池也超出的时候，则拒绝执行

37 );

38 }

39

40 /\*\*

41 \* 往线程池中添加任务

42 \* @param r

43 \*/

44 public void executor(Runnable r) {

45 if(r!=null) {

46 executor.execute(r);

47 }

48 }

49

50 /\*\*

51 \* 从线程池中移除任务

52 \* @param r

53 \*/

54 public void remove(Runnable r) {

55 if(r!=null) {

56 executor.remove(r);

57 }

58 }

59 }

**4、volatile关键字的作用，原理**

　　答：保证内存可见性和禁止指令重排。实现原理可参考：[JAVA并发变成--valatile关键字剖析](https://www.cnblogs.com/dolphin0520/p/3920373.html)

**5、synchronized关键字的用法，优缺点**

　　答：java关键字，当它用来修饰一个方法或者代码块的时候，能够保证在同一时刻最多只有一个线程执行该代码段的代码；

　　　　synchronized修饰的方法或者对象，只能以同步的方式执行，会引起性能问题；无法中断一个正在等候获得锁的线程，也无法通过投票获得锁；一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险；

**6、Lock接口有哪些实现类，使用场景是什么**

　　答：Lock接口有三个实现类，一个是ReentrantLock，另两个是ReentrantReadWriteLock类中的两个静态内部类ReadLock和WriteLock。

　　使用场景：一般应用于多度少写，因为读的线程之间没有竞争，所以比起synchronzied，性能要好很多；

**7、悲观锁、乐观锁的优缺点，CAS有什么缺陷，该如何解决**

　　悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次拿数据的时候都会上锁，这样别人拿数据的时候就会阻塞知道它拿到锁；比如关系型数据库的行锁、表锁、读锁、写锁；比如java里面的同步原语synchronized关键字的实现也是悲观锁；

　　乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下再次期间别人有没有更新这个数据。乐观锁适用于多读的应用类型，可以提高吞吐量。java中java.util.conncurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的；

　　CAS：CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其他线程都失败，失败的线程不会被挂起，而是被告知这次竞争失败，并可以再次尝试；

　　CAS的缺陷：ABA问题、循环时间长开销大，只能保证一个共享变量的原子操作；

**8、ABC三个线程如何保证顺序执行**

　　答：用Thread.join() 方法，或者线程池newSingleThreadExecutor（原理是会将所有线程放入一个队列，而队列则保证了FIFO）,也可以通过ReentrantLock，state整数用阿里判断轮到谁来执行

**9、线程的状态都有哪些（五大状态）**

　　新建状态（new）：当用new操作符创建一个线程时，如new Thread()，线程还没有开始运行，此时处于仙剑状态；

　　就绪状态（runnable）：一个新创建的线程并不自动开始运行，要执行线程，必须要调用线程的start()方法，当线程对象调用start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行run()方法。当start（）方法返回后，线程就处于就绪状态；

　　运行状态（running）：当线程获得cpu时间后，他才进入运行状态，真正开始实行run()方法

　　阻塞状态（blocked）：当线程运行过程中，可能由于各种原因进入阻塞状态；

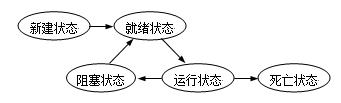
　　　　a.线程通过调用sleep方法进入睡眠状态

　　　　b.线程调用一个在I/O上被阻塞的操作，即该操作在输入输出操作完成之前不会返回到它的调用者

　　　　c.线程试图得到一个锁，而该锁正被其他线程持有

　　　　d.线程正等待某个触发条件

　　死亡状态（dead）：run方法自然退出而自然死亡，或者一个未捕获的异常终止了run方法而使线程猝死



**10、sleep和wait的区别**

　　答：首先，sleep()方法属于Thread类的，而wait()方法是属于Object类的；sleep()方法导致了程序暂停执行指定的时间，让出cpu给其他线程，但是他的监控状态依然保持，当指定的时间到了又自动回恢复运行状态，调用了sleep()方法的过程中，线程不会释放对象锁；而当调用了wait()方法的时候，线程回放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备。

**11、notify()和notifyAll()的区别**

　　答：notify()方法表示，当前线程已经放弃对资源的占有，通知等待的线程来获取对资源的占有权，但是只有一个线程能够从wait状态中恢复；notifyAll()方法表示，当前的线程已经放弃对资源的占有，通知所有的等待线程从wait()方法后的语句开始执行，但最终只有一个线程能竞争获得锁并执行；notify()是对notifyAll()的一个优化，

**12、ThreadLocal的了解，实现原理。**

 　　答：ThreadLocal，线程本地变量。定义了一个ThreadLocal，每个线程往这个ThreadLocal中读写都是线程隔离的，互相之间不会影响，他提供了一种将可变数据通过每个线程有自己的独立副本从而实现线程封闭的机制；实现的思路，Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，也就是说每个线程都有一个自己的ThreadLocalMap。ThreadLocalMap有自己的独立实现，可以简单的将它的key视作ThreadLocal，value为代码中放入的值（实际上key并不是ThreadLocal本省，而是它的一个弱引用）。每个线程在往ThreadLocal里set值的时候，都会往自己的ThreadLocalMap里存，读也是已某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程的隔离。如果想详细了解，可以参考：[ThreadLocal源码解读](https://www.cnblogs.com/micrari/p/6790229.html)

数据库相关

**1、常见的数据库优化手段**

　　答：库表优化，表设计合理化，符合三大范式；添加适当的索引（普通索引、主键索引、唯一索引、全文索引）；分库分表；读写分离等；sql语句优化，定位执行效率低，慢sql的语句，通过explain分析低效率的原因；

**2、索引的优缺点，什么字段上建立索引**

　　答：优点方面：第一，通过创建唯一索引可以保证数据的唯一性；第二，可以大大加快数据的检索速度，是主要目的；第三；在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间；第四，可以在查询中使用优化隐藏器，提高系统的性能；

　　缺点方面：第一，创建索引和维护索引要耗费时间，并且随着数据量的增加而增加；第二，每一个索引需要占用额外的物理空间，需要的磁盘开销更大；第三，当对表中的数据进行增加、删除、修改操作时，索引也要动态维护，降低了数据的维护速度；

　　一般来说，在经常需要搜索的列上，强制该列的唯一性和组织表中数据的排列结构的列，在经常用在链接的列上，在经常需要排序的列上，在经常使用在where字句的列上可以添加索引，以提升查询速度；同样，对于一些甚少使用或者参考的列，只有很少数值的列（如性别），定义为text,image,bit的列，修改性能远远大于检索性能的列不适合添加索引；

**3、数据库连接池**

　　答：数据库连接池（Connection pooling）是程序启动时建立足够的数据库连接，并将这些连接组成一个连接池，由程序动态的对池中的连接进行申请、使用、释放；

　　（1）程序初始化时创建连接池

　　（2）使用时向连接池申请可用连接

　　（3）使用完毕，将连接返还给连接池

　　（4）程序退出时，断开所有的连接，并释放资源

计算机网络

**1、TCP和UDP的区别**

　　答：TCP（传输控制协议），UDP（用户数据报协议）

　　（1）TCP面向连接（如打电话先拨号建立连接）；UDP是无连接的，即发送数据之前不需要建立连接；

　　（2）TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序达到；UDP尽最大努力交付，即不保证可靠交付；

　　（3）TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流；UDP是面向报文，UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）

　　（4）每一条TCP连接只能是点到点的，UDP支持一对一，一对多，多对一和多对多的交互通信；

　　（5）TCP首部开销20字节，UDP首部开销8字节；

　　（6）TCP的逻辑通信信道是全双工的可靠信道，DUP则是不可靠信道；

**2、三次握手，四次挥手，为什么要四次挥手。**

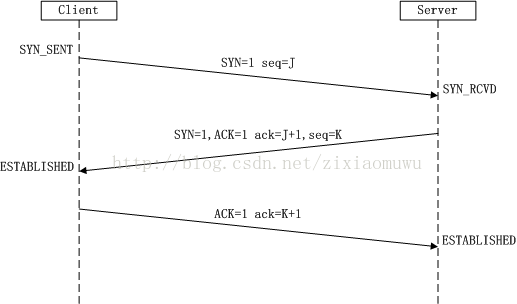
　　答：三次握手的目的是建立可靠的通信信道，简单来说就是数据的发送与接收，主要目的是双方确认自己与对方的发送和接收机能正常；

　　　　第一次握手：Client什么都不能确认，Server确认了对方发送正常；

　　　　第二次握手：Clent确认了，自己发送、接收正常，对方发送、接收正常；Server确认了自己接收正常，对方发送正常；

　　　　第三次握手：Clent确认了，自己发送、接收正常，对方发送、接收正常；Server确认了自己发送、接收正常，对方发送、接收正常；

　　所以，经过三次握手之后，就能确认双方收发功能都正常；



　　四次挥手：

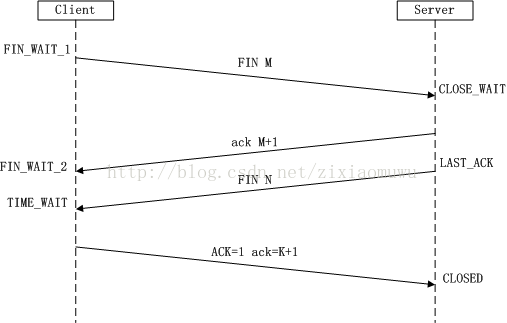
　　　　A:“喂，**我不说了 (FIN)**。”A->FIN\_WAIT1

　　　　B:“我知道了**(ACK)**。等下，**上一句还没说完**。Balabala…..（**传输数据**）”B->CLOSE\_WAIT | A->FIN\_WAIT2

　　　　B:”好了，说完了，我也不说了（**FIN**）。”B->LAST\_ACK

　　　　A:”我知道了（**ACK**）。”A->TIME\_WAIT | B->CLOSED

　　　　A**等待2MSL**,保证B收到了消息,否则重说一次”我知道了”,A->CLOSED



**3、长连接和短连接。**

　　短连接：连接=》传输数据=》关闭连接

　　HTTP是无状态的，浏览器和服务器之间每进行一次http操作，就建立一次连接，但任务结束就中断连接；也可以理解为短连接是指socket连接后，发送接收完数据马上断开连接；

　　长连接：连接=》传输数据=》保持连接=》传输数据=》。。。=》关闭连接

　　长连接指建立socket连接后不管是否使用都保持连接，但安全性较差；

 设计模式

　　此处推荐阅读：[java23种设计模式 深入理解](https://www.cnblogs.com/foryang/p/5849402.html)

**1、单例模式的几种写法**

　　懒汉模式

public class Singleton {

private static Singleton instance = null;

private Singleton(){}

public static synchronized Singleton getInstance(){

//如果还没有被实例化过，就实例化一个，然后返回

if(instance == null){

instance = new Singleton();

}

return instance;

}

}

　　饿汉模式

public class Singleton {

//类加载的时候instance就已经指向了一个实例

private static Singleton instance = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return instance;

}

}

　　双重检验锁

public class Singleton {

private static Singleton instance = null;

private Singleton(){}

public static Singleton getInstance(){

if(instance == null){

synchronized (Singleton.class){

if(instance == null){

instance = new Singleton();

}

}

}

return instance;

}

}

　　静态内部类：因为JAVA静态内部类的特性，加载的时候不会加载内部静态类，使用的时候才会加载，而使用的时候类加载又是线程安全的，这就完美达到了效果；

public class Singleton {

private static class SingletonHolder{

private static Singleton instance = new Singleton();

}

private Singleton(){}

public static Singleton getInstance(){

return SingletonHolder.instance;

}

}

　　枚举：

public enum Singleton {

INSTANCE;

}

**2、Spring使用了哪些设计模式**

　　（1）工厂模式，在各种BeanFactory以及ApplicationContext创建中都用到了；

　　（2）模板模式，也是在各种BeanFactory以及ApplicationContext创建中都用到了；

　　（3）代理模式，在AOP实现中用到了JDK的动态代理；

　　（4）单例模式，比如创建bean的时候；

　　（5）策略模式，第一个地方，加载资源文件的地方，使用了不同的方法，比如：classPathResource,FileSystemResource,ServletContextResource,UrlResource但他们都有共同的接口Resource；第二个地方就是AOP的实现中，采用了不同的方式，JDK动态代理和CGLIB代理；

分布式相关

**1、分布式事务的控制**

　　可以参考[分布式系统事务一致性解决方案](https://blog.csdn.net/braveheartzj/article/details/52485986)

**2、分布式锁**

　　答：一般使用zk瞬时有序节点实现的分布式锁，或者利用redis的setnx()封装分布式锁；提供思路，具体的可以自行详细理解；

**3、分布式session如何设计**

　　答：一个比较成熟的方案是通过redis进行session共享。详细的原理可以参考[一种分布式session实现方案](https://blog.csdn.net/hounanjsj/article/details/56279590)

**4、关于dubbo**

　　可以参考博文：[Dubbo学习总结（2）——Dubbo架构详解](https://blog.csdn.net/u012562943/article/details/49025303)

**5、可以了解zk相关知识**

缓存相关

**1、redis和memcached的区别**

　　（1）redis和memcache都是将数据放入内存中，都是内存数据库。但是memcache可以缓存图片、视频等数据；

　　（2）redis不仅仅支持简单的k/v数据，还提供list、set、hash等数据结构的存储；

　　（3）虚拟内存--redis当物理内存用完时，可以将一些很久没有用到的value交换到磁盘；

　　（4）过期策略--memcache在set时就指定，例如set key1008，即永不过期，redis通过expire设定；

　　（5）分布式--设定memcache集群，利用magent做一主多从；redis可以做一主多从或一主一从；

　　（6）存储数据安全--memcache挂掉后，数据没了，redis可以定期保存到磁盘进行持久化；

　　（7）灾难恢复--memcache挂掉后，数据不可恢复。redis数据丢失后可以通过aof恢复；

　　（8）redis支持数据备份，即master-slave主备模式；

**2、redis是单线程的么（是的）**

**3、redis的持久化策略**

　　答：rdb：快照形式是直接把内存中的数据保存到一个dump文件中，定时保存

　　　　aof：把所有的对redis的服务器进行修改的命令都存到一个文件里，命令的集合

框架相关

**1、SpringMvc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（如有则生成）一并返回给DispatcherServlet

　　（4）DispatcherServlet调用HandlerAdapter处理器映射器

　　（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

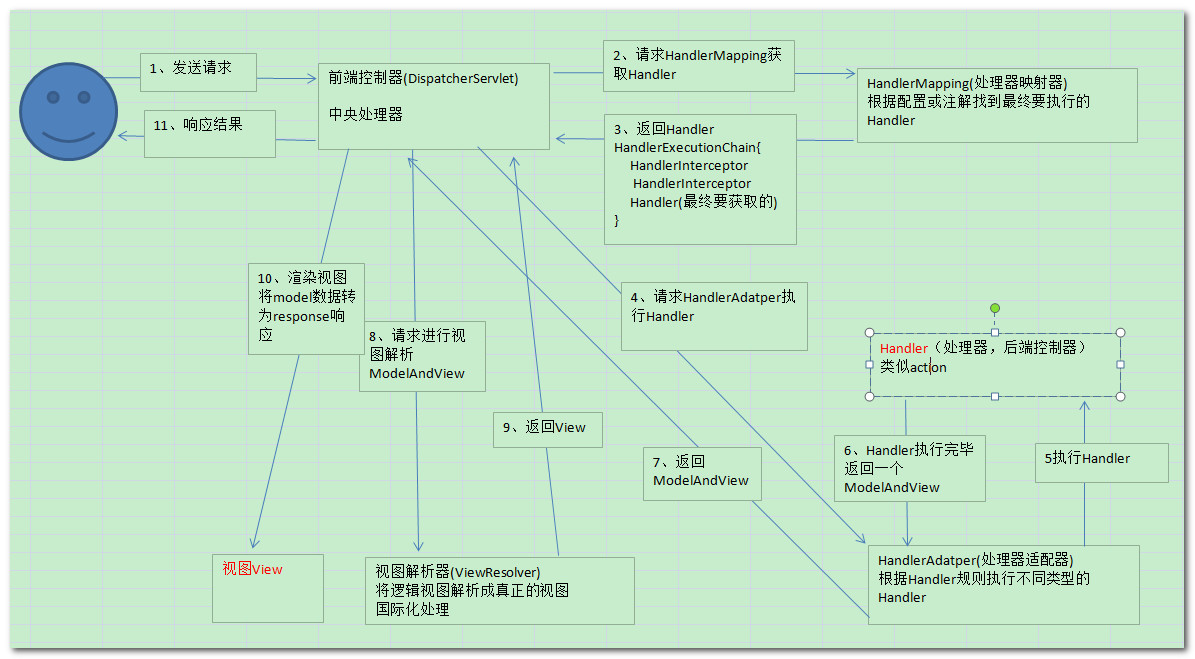
　　（7）HandlerAdapter将Controller执行结果ModelAndView返回给DispatcherServlet

　　（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器

　　（9）ViewResolver解析后返回具体的view

　　（10）DispatcherServlet根据view进行试图渲染（即将模型数据填充至视图中）

　　（11）DispatcherServlet响应用户



　　以下组件通常使用框架提供实现：

　　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。

**2、Quartz概念及原理**

　　org.quartz.Job：它是一个抽象接口，表示一个工作，也是我们要执行的具体的内容，只定义了一个接口方法：void execute(JobExecutionContext context)

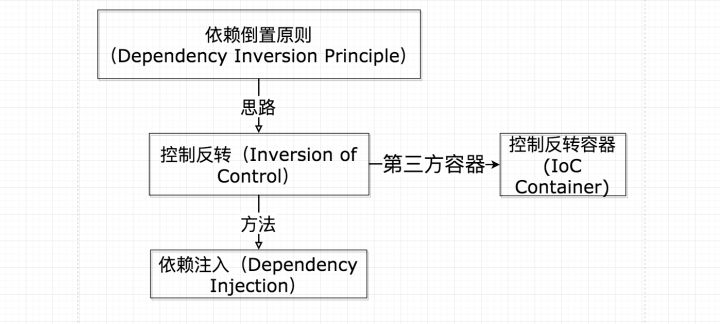
　　org.quartz.JobDetail：JobDetail表示一个具体的可执行的调度程序，Job是这个可执行调度程序所要执行的内容，它包含了这个调度任务的方案和策略

　　org.quartz.Trigger：Trigger是一个抽象接口，表示一个调度参数的配置，通过配置他，来告诉调度器什么时候去调用JobDetail

　　org.quartz.Scheduler：一个调度容器，可以注册多个Trigger和JobDetail。当Trigger和JobDetail组合，就可以被Scheduler容器调度了

**3、Spring的IOC有什么优势**

　　答：要了解IOC首先要明白依赖倒置原则（Dependency Inversion Principle），就是把原本的高层建筑依赖底层建筑倒置过来，变成底层建筑依赖高层建筑。高层建筑决定需要什么，底层去实现这样的需求，但是高层并不用管底层的是怎么实现的；而控制反转（Inversion of Control）就是依赖倒置原则的一种代码的设计思路；



　　IOC思想的核心，资源不由使用资源的双方管理，由不适用资源的第三方管理。

　　优势：资源集中管理，实现资源的可配置和易管理；降低了使用资源双方的依赖程度，也就是降低了耦合度；