# 一、Java核心知识

## 1.JDK1.8、JDK1.9、JDK10，JDK11新特性

java 8 在 Java 历史上是一个开创新的版本，下面 JDK 8 中 5 个主要的特性：

1. \*\*Lambda\*\* 表达式，允许像对象一样传递匿名函数

2. Stream API，充分利用现代多核CPU，可以写出很简洁的代码

3. Date 与 TimeAPI，最终，有一个稳定、简单的日期和时间库可供你使用。

4. 扩展方法，现在，\*\*接口中可以有静态、默认方法\*\*。

5. 重复注解，现在你可以将相同的注解在同一类型上使用多次。

## 2.跨平台特性

### 什么是JDK?什么是JRE？什么是JVM？三者之间的联系与区别

这几个是Java中很基本很基本的东西，但是我相信一定还有很多人搞不清楚！为什么呢？因为我们大多数时候在使用现成的编译工具以及环境的时候，并没有去考虑这些东西。

JDK: 顾名思义它是给开发者提供的开发工具箱,是给程序开发者用的。它除了包括完整的JRE（Java Runtime Environment），Java运行环境，还包含了其他供开发者使用的工具包。

JRE: 普通用户而只需要安装JRE（Java Runtime Environment）来运行Java程序。而程序开发者必须安装JDK来编译、调试程序。

JVM： 当我们运行一个程序时，JVM负责将字节码转换为特定机器代码，JVM提供了内存管理/垃圾回收和安全机制等。这种独立于硬件和操作系统，正是java程序可以一次编写多处执行的原因。

### 区别与联系：

JDK用于开发，JRE用于运行java程序 ；

JDK和JRE中都包含JVM ；

JVM是java编程语言的核心并且具有平台独立性。

## 3.Java四个基本特性

### 抽象

### 封装

### 继承

### 多态

允许不同类对象对同一消息做出响应，即同一消息可以根据发送对象的不同而采用多种不同行为方式。

多态的实现方式：1.接口实现；2.继承父类重写方法；3.同一类中进行方法重载。

## 4.Object类及其方法实现（尤其是equals() and hashCode()）

## 5.基本数据类型与引用数据类型

8中基本数据类型：byte(1字节，-128~127) short(2字节，-2^15~2^15-1) int(4字节，-2^31~2^31-1) long(8字节，-2^63~2^63-1) float(4字节) double(8字节) boolean(1字节) char(2字节)。

封装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

### String 转出 int型，判断能不能转？如何转？

可以转，得处理异常 Integer.parseInt(s) 主要为NumberFormatException：1）当你输入为字母时，也就是内容不是数字时，如abcd 2）当你输入为空时 3）当你输入超出int上限时 Long.parseLong("123")转换为long。

### short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 +=1;有什么错?

- 1) 对于short s1=1;s1=s1+1来说，在s1+1运算时会\*\*自动提升表达式的类型为int\*\*，那么将int赋予给short类型的变量s1会出现类型转换错误。

- 2) 对于short s1=1;s1+=1来说 \*\*+=是java语言规定的运算符，java编译器会对它进行特殊处理\*\*，因此可以正确编译。

### Integer与int的区别

## 6.Exception vs Error

CheckedException，RuntimeException的区别

## 7.finally vs final vs finalize

### 1.finalize用途

答：垃圾回收器(garbage colector)决定回收某对象时，就会运行该对象的finalize()方法 但是在Java中很不幸，如果内存总是充足的，那么垃圾回收可能永远不会进行，也就是说filalize()可能永远不被执行，显然指望它做收尾工作是靠不住的。 那么finalize()究竟是做什么的呢？它最主要的用途是回收特殊渠道申请的内存。Java程序有垃圾回收器，所以一般情况下内存问题不用程序员操心。但有一种JNI(Java Native Interface)调用non-Java程序（C或C++），finalize()的工作就是回收这部分的内存。

### 2.finally

finally 一定会被执行，如果 finally 里有 return 语句，则覆盖 try/catch 里的 return ， 比较爱考的是 finally 里没有 return 语句，这时虽然 finally 里对 return 的值进行了修改，但 return 的值并不改变这种情况。

在try中return之前会执行finally中的代码，如果finally中有return则直接return，值为finally中修改的值。

如果finally中没有return，则执行try中return，数值仍然是try中的值。

### 3．finally代码块和finalize()方法有什么区别？

无论是否抛出异常，finally代码块都会执行，它主要是用来释放应用占用的资源。finalize()方法是Object类的一个protected方法，它是在对象\*\*被垃圾回收之前由Java虚拟机来调用的。\*\*

### 4.final的用途。

## 8.四种引用：强引用、软引用、若引用、幻象引用的比较

## 9.重载 vs 重写

\*\*重载\*\*： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

\*\*重写\*\*： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为private则子类就不能重写该方法。

## 10.接口 vs 抽象类

抽象类和接口的区别，类可以继承多个类么，接口可以继承多个接口么,类可以实现多个接口么？

1. 抽象类里可以有构造方法，而接口内不能有构造方法。

2. 抽象类中可以有普通成员变量，而接口中不能有普通成员变量。

3. 抽象类中可以包含非抽象的普通方法，而接口中所有的方法必须是抽象的，不能有非抽象的普通方法。

4. 抽象类中的抽象方法的访问类型可以是public ，protected和private，但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。

5. 抽象类中可以包含静态方法，接口内不能包含静态方法。

6. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static类型，并且默认为public static final类型。

7. 一个类可以实现多个接口，但只能继承一个抽象类。

8. 抽象类可以有默认的方法实现,java 8之前,接口中不存在方法的实现.

## 11.深克隆 vs 浅克隆

## 12.String vs StringBuffer vs StringBuilder （底层实现）

## 13.IO vs NIO vs AIO vs BIO

## 14.反射与动态代理

### 1. 反射的原理，反射创建类实例的三种方式是什么。

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

### 2. 反射中，Class.forName和ClassLoader区别 。

### 3. java动态代理实现与原理详细分析

### 4. 描述动态代理的几种实现方式，分别说出相应的优缺点。

### 5. 动态代理与cglib实现的区别。

### 6. 为什么CGlib方式可以对接口实现代理。

## 15.序列化与反序列化（底层实现）

什么是序列化，怎么序列化，为什么序列化，反序列化会遇到什么问题，如何解决。

## 16.Vector vs ArrayList vs LinkedList

ArrayList和LinkedList有什么区别。

## 17.HashTable vs HashMap vs TreeMap

## 18.HashTable vs HashMap vs ConcurrentHashMap

## 19.有序Map：TreeMap和LinkedHashMap

## 20.HashSet vs TreeSet

## 21.写出三种单例模式实现 。

## 22.如何在父类中为子类自动完成所有的hashcode和equals实现？这么做有何优劣。

## 23.请结合OO设计理念，谈谈访问修饰符public、private、protected、default在应用设计中的作用。

## 24.数组和链表数据结构描述，各自的时间复杂度。

## 25.在自己的代码中，如果创建一个java.lang.String类，这个类是否可以被类加载器加载？为什么。

不能自己写以"java."开头的类，其要么不能加载进内存，要么即使你用自定义的类加载器去强行加载，也会收到一个SecurityException。

说一说你对java.lang.Object对象中hashCode和equals方法的理解。在什么场景下需要重新实现这两个方法。

## 26.在jdk1.5中，引入了泛型，泛型的存在是用来解决什么问题。

## 27.java运算符 与（&）、非（~）、或（|）、异或（^）

## 28.hashCode()与equals()

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java中，这就意味着Java中的任何类都包含有hashCode() 函数。

### 为什么要有hashCode

我们以“HashSet如何检查重复”为例子来说明为什么要有hashCode：

当你把对象加入HashSet时，HashSet会先计算对象的hashcode值来判断对象加入的位置，同时也会与其他已经加入的对象的hashcode值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同hashcode值的对象，这时会调用equals（）方法来检查hashcode相等的对象是否真的相同。如果两者相同，HashSet就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。

### hashCode（）与equals（）的相关规定

如果两个对象相等，则hashcode一定也是相同的

两个对象相等,对两个对象分别调用equals方法都返回true

两个对象有相同的hashcode值，它们也不一定是相等的

因此，equals方法被覆盖过，则hashCode方法也必须被覆盖

hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

## 29.Java和C++的区别

1. 都是面向对象的语言，都支持封装、继承和多态

2. Java不提供\*\*指针\*\*来直接访问内存，程序内存更加安全

3. Java的类是单继承的，\*\*C++支持多重继承；虽然Java的类不可以多继承\*\*，但是接口可以多继承。

4. Java有自动内存管理机制，不需要程序员手动释放无用内存

# 二、并发与多线程(应用及底层实现)

https://mp.weixin.qq.com/s/0CI9od4DIxRrmOGFJw0SuQ

## 1.线程的创建方式

多线程的几种实现方式，什么是线程安全。

\*\*线程安全\*\*：当多个线程访问同一对象，如果不用考虑这些线程在运行时环境下的调度和交替运行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象行为都可以获取正确的结果，那么这个对象就是线程安全。

通俗的来说就是程序在多线程环境下访问相同的资源，如同一内存，数据库或者文件能够得到正确的结果，在实际开发当中，一般只有出现写的情况下才会出现线程安全问题，单独进行读操作不会出现线程安全问题。

1. 继承Thread类。

\*\*new Thread的弊端：\*\*

- 每次new Thread新建对象，性能差。

- 线程缺乏统一管理，可能无限制的新建线程，相互竞争，有可能占用过多系统资源导致死机或OOM。

- 缺乏更多功能，如更多执行，定期执行，线程中断。

2. 实现Runable接口。

```

Thread thread = new Thread(new Demo());

thread.start();

```

3. 使用匿名内部类new Thread(){run 方法}.start();。

4. 使用Callable接口，重写call()方法，call()方法有返回值。

```

FutureTask<Integer> task = new FutureTask<>(d);

Thread t = new Thread(task);

t.start();

```

5. 定时器(quartz框架)。

6. 线程池

```

public static void main(String[] args){

//创建带有10个线程的线程池

//Executor threadPool = Executors.newFixedThreadPool(10);

ExecutorService threadPool = Executors.newCachedThreadPool();//认为够用就回收，不够用就创建线程

//线程任务

for(int i=0;i<100;i++) {

threadPool.execute(new Runnable() {

@Override

public void run() {

System.out.println(Thread.currentThread().getName());

}

});

}

}

```

## 2.多线程应用场景

https://blog.csdn.net/hll814/article/details/50816268

https://mp.weixin.qq.com/s/N1YAZmBLX-bC-XkuC7yl\_Q

## 3.线程状态与转换

画一个线程的生命周期状态图。

![image](http://zlcalma.top/upload/2018/12/110764qd6ehaprp15emd9cp59u.png)

1. 新建状态（New）：新创建了一个线程对象。

2. 就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3. 运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4. 阻塞状态（Blocked）：阻塞状态是线程因为某种原因放\*\*弃CPU使用权\*\*，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)

（二）、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。（注意,\*\*sleep是不会释放持有的锁\*\*）

5. 死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 4.线程安全与同步机制：volatile vs synchronized vs Lock(ReentrantLock)

### 首先看看volatile和synchronized的区别：

1. volatile是一种更加轻量级的同步机制，volatile\*\*不会造成线程阻塞\*\*，synchronized可能会造成线程的阻塞。

2. volatile本质是在告诉当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

3. volatile仅能使用在\*\*变量级别\*\*；synchronized则可以使用在\*\*变量、方法、和类级别\*\*的。

4. volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性，另外volatile可以保证有序性。

### 再来看看synchronized和Lock的区别：

1. synchronized是作用在jvm层面的，Lock实际是Java中的类。

2. synchronized在执行完代码块或者出现异常的情况下，JVM会自动释放锁，而Lock必须在finally语句中释放锁。

3. synchronized和Lock都是可重入锁，但是Lock可以设置为公平或者非公平锁，但是synchronized只能是非公平的。

## 5.volatile底层原理

volatile的主要作用是保证\*\*可见性和禁止指令重排序\*\*。

### 那么如何解决并发编程中的可见性问题？

根据Java内存模型，每个线程都有自己的工作内存，并共享一个主内存，如上图所示，当两个线程并发读写一个共享变量data，初始值为0，当线程1将工作内存中的data更新为1时，主内存中和线程2中的工作内存读到data数据都为0，因此，在这种场景下就出现了并发编程中的可见性，即线程1更新自己data变量后，线程2无法看到，其一直看到自己工作内存中的一个旧的副本。

当针对上述变量data增加volatile关键字后，会产生三种效果：

1. 当线程1更新本地内存中data值后，\*\*强制将data变量最新值刷回主内存\*\*，使得主内存中的值立马变为线程1更新的值。

2. 其他线程(线程2)中如果存在这个data变量，那么就会强制让其他线程工作内存中的\*\*data变量缓存直接过期\*\*，不允许再次读取和使用。

3. 如果线程2需要读取data值，当其尝试读取本地内存中的data值时，就会发现data已经过期，它必须\*\*重新从主内存中读取最新的data值\*\*。

### 禁止指令重排序

Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 \*\*happens-before 原则\*\*。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

volatile变量规则：对一个变量的\*\*写操作先行发生于后面对这个变量的读操作\*\*。

### volatile原理

观察加入volatile关键字和没有加入volatile关键字时，所生成的汇编代码发现，加入volatile关键时，会多出一个\*\*lock前缀指令\*\*。lock前缀指令其实相当于一个\*\*内存屏障\*\*。内存屏障是一组处理指令，用来实现对内存操作的顺序限制。volatile底层就是通过内存屏障来实现的。

volatile底层实现是通过插入内存屏障，JMM采用了保守策略：

在每一个volatile写操作前面插入一个StoreStore屏障。

在每一个volatile写操作后面插入一个StoreLoad屏障。

在每一个volatile读操作后面插入一个LoadLoad屏障。

在每一个volatile读操作后面插入一个LoadStore屏障。

StoreStore屏障可以保证在volatile写之前，其前面所有普通写操作\*\*\*\*都已经刷新到主内存。

StoreLoad屏障的作用是避免volatile写与后面可能有的volatile读/写操作重排序。

LoadLoad屏障用来禁止处理器把上面的volatile读与下面的普通读重排序。

LoadStore屏障用来禁止处理器把上面的volatile读与下面的普通写重排序。

[http://zlcalma.top/article/concurrent-03](http://zlcalma.top/article/concurrent-03)

## 6.synchronized底层原理及其锁的升级与降级

### 作用域：

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础。其的作用域主要有：

- 放在\*\*普通方法\*\*上，作用于调用这个方法的对象，内置锁作用域该实例对象，所谓的互斥锁即为独占锁。

- 修饰\*\*静态方法\*\*时，内置锁作用域是当前Class字节码对象，作用于类的所有对象。

- 修饰\*\*类\*\*：括号括起来的部分，作用于类所有对象。

- 修饰\*\*代码块\*\*时，内置锁作用Synchronized作用域括号里配置的对象。

### 获取与释放

同步代码块使用\*\*monitorenter和monitorexit\*\*指令实现，同步方法依靠方法修饰符上的\*\*ACC\_SYNCHRONIZED\*\*实现。

- 同步代码块：monitorenter指令插入到同步代码块的开始位置，monitorexit指令插入到同步代码块的结束位置。JVM需要保证每一个monitorenter都有一个monitorexit与之对应。任何对象都有一个monitor与之关联，并且一个monitor被持有之后，它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor所有权，及尝试获取对象锁。

- 同步方法：synchronized方法会被翻译成普通方法和返回指令：invokevirtual、areturn指令，在VM字节码层面并没有任何特别的指令来实现被synchronized修饰的方法，而是在Class文件方法表中将该方法的access\_flags字段中synchronized标志位置1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的Class在JVM的内部对象表示为锁对象。

### Java对象头和monitor

synchronized使用的锁是在Java对象头里的，Hotspot虚拟机对象头主要包括两部分数据：\*\*Mark Word\*\*(标记字段),\*\*类型指针\*\*。其中类型指针是对象指向它类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个实例，Mark Word用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键。

Monitor是用来实现Java对象的同步工具，任何一个Java对象。都会包含一个内部锁或者Monitors锁。

### 锁的优化和降级

- 自旋锁：所谓自旋锁，实际是让该线程等待一段时间，不会被挂起，看持有锁的线程是否会很快释放锁，所谓等待就是执行一段无意义的循环语句。

- 轻量级锁：轻量级锁是在\*\*没有多线程的竞争下\*\*，减少重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁。

- 偏向锁：引入偏向锁的目的是，为了在没有多线程竞争的情况下尽量减少不必要的轻量级锁执行路径。上面提到在轻量级锁加锁解锁的操作需要依赖多次原子指令，偏向锁可以减少不必要的CAS操作。

[http://zlcalma.top/article/concurrent-02](http://zlcalma.top/article/concurrent-02)

## 7.Lock(ReentrantLock)底层原理

AQS是Java并发包的基础类，ReentrantLock、ReetranReadWriteLock底层都是基于AQS来实现的。所谓AQS就是AbstractQueuedSynchronizer,即抽象队列同步器。

### 那么ReentrantLock加锁和释放锁的底层原理？

当一个线程尝试使用ReentrantLock的lock()方法进行加锁时，AQS对象内部有一个核心变量state，代表加锁的状态。初始条件下，这个state值为0。另外AQS内部还有一个关键变量，用来记录当前加锁的是哪个线程，初始情况下这个变量为null。

[http://zlcalma.top/article/concurrent-07](http://zlcalma.top/article/concurrent-07)

## 8.ThreadLocal

ThreadLocal，即线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构，一个线程可以通过ThreadLocal变量查询到绑定在这个线程上的值。ThreadLocal与线程同步机制不同，线程同步机制是多个线程共享同一个变量，而ThreadLocal是为每一个线程创建一个单独的变量副本，因此每一个线程都可以独立的改变自己所拥有的变量副本，而不会影响其他线程对应的副本。可以说ThreadLocal为多线程变量问题提供了一种解决思路。

http://zlcalma.top/article/Concurrent-04

## 9.线程通信

## 10.线程池（底层实现）

## 11.死锁的出现场景、定位以及修复

## 12.CAS 与 Atomic\*类型实现原理

CAS机制是什么，如何解决ABA问题。

## 13.AQS：并发包基础技术

## 14.Java并发包（java.util.concurrent及其子包）提供的并发工具类：

## 15.比synchronized更加高级的各种同步结构，如：Semaphore，CyclicBarrier， CountDownLatch；

countdowlatch和cyclicbarrier的内部原理和用法，以及相互之间的差别(比如countdownlatch的await方法和是怎么实现的)。

## 16.各种线程安全的容器（主要有四类：Queue,List,Set，Map），如ConcurrentHashMap,ConcurrentSkipListMap,CopyOnWriteArrayList；

## 17.各种并发队列的实现，如各种BlockingQueue的实现（ArrayBlockingQueue, LinkedBlockingQueue, SynchorousQueue, PriorityBlockingQueue,DelayQueue,LinkedTranferQueue）等

## 18.Executor框架与线程池

用过线程池吗，如果用过，请说明原理，并说说newCache和newFixed有什么区别，构造函数的各个参数的含义是什么，比如coreSize，maxsize等。

线程池的关闭方式有几种，各自的区别是什么。

## 19.sleep和wait的区别。

## 20.sleep和sleep(0)的区别。

Thread.Sleep(0) 并非是真的要线程挂起0毫秒，意义在于这次调用Thread.Sleep(0)的当前线程确实的被冻结了一下，让其他线程有机会优先执行。Thread.Sleep(0) 是你的线程暂时放弃cpu，也就是释放一些未用的时间片给其他线程或进程使用，就相当于一个让位动作。

在线程中，调用sleep（0）可以释放cpu时间，让线程马上重新回到就绪队列而非等待队列，sleep(0)释放当前线程所剩余的时间片（如果有剩余的话），这样可以让操作系统切换其他线程来执行，提升效率

https://blog.csdn.net/qiaoquan3/article/details/56281092

## 21.量级锁，可重入锁，公平锁，非公平锁，乐观锁，悲观锁。

### 公平与非公平锁

[http://zlcalma.top/article/concurrent-07](http://zlcalma.top/article/concurrent-07)

### 乐观锁和悲观锁

\*\*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，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如\*\*行锁，表锁等，读锁，写锁\*\*等，都是在做操作之前先上锁。再比如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在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.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。

## 22.用过哪些原子类，他们的原理是什么。

## 23.假如有一个第三方接口，有很多个线程去调用获取数据，现在规定每秒钟最多有10个线程同时调用它，如何做到。

## 24.spring的controller是单例还是多例，怎么保证并发的安全。

## 25.用三个线程按顺序循环打印abc三个字母，比如abcabcabc。

## 26.如果让你实现一个并发安全的链表，你会怎么做。

## 27.讲讲java同步机制的wait和notify。

## 28.多线程如果线程挂住了怎么办。

## 29.使用synchronized修饰静态方法和非静态方法有什么区别。

## 30.简述ConcurrentLinkedQueue和LinkedBlockingQueue的用处和不同之处。

## 31.导致线程死锁的原因？怎么解除线程死锁。

## 32.非常多个线程（可能是不同机器），相互之间需要等待协调，才能完成某种工作，问怎么设计这种协调方案。

## 33.用过读写锁吗，原理是什么，一般在什么场景下用。

## 34.延迟队列的实现方式，delayQueue和时间轮算法的异同。

## 35.Java中的守护线程和本地线程的区别？

java中的线程分为两种：守护线程（Daemon）和用户线程（User）。

任何线程都可以设置为守护线程和用户线程，通过方法Thread.setDaemon(bool on)；true则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon()必须在Thread.start()之前调用，否则运行时会抛出异常。

\*\*两者的区别\*\*：

唯一的区别是判断虚拟机(JVM)何时离开，Daemon是为其他线程提供服务，如果全部的User Thread已经撤离，Daemon没有可服务的线程，JVM撤离。也可以理解为守护线程是JVM自动创建的线程（但不一定），用户线程是程序创建的线程；比如JVM的\*\*垃圾回收线程是一个守护线程\*\*，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是Java虚拟机上仅剩的线程时，Java虚拟机会自动离开。

## 36.在Java中Executor和Executors的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。

Executor 接口对象能执行我们的线程任务。

ExecutorService接口继承了Executor接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。

使用ThreadPoolExecutor 可以创建自定义线程池。

Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并可以使用get()方法获取计算的结果。

## 37.在Java中CycliBarriar和CountdownLatch有什么区别？

CyclicBarrier可以重复使用(计数器可以重置)，而CountdownLatch不能重复使用(计数器不能重置，线程数是在初始化时设定好的)。

Java的concurrent包里面的CountDownLatch其实可以把它看作一个计数器，只不过这个\*\*计数器的操作是原子操作\*\*，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。

所以在当前计数到达零之前，await方法会一直受阻塞。之后，会释放所有等待的线程，await的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

CountDownLatch的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个CountDownLatch对象的await()方法，其他的任务执行完自己的任务后调用同一个CountDownLatch对象上的countDown()方法，这个调用await()方法的任务将一直阻塞等待，直到这个CountDownLatch对象的计数值减到0为止

CyclicBarrier一个\*\*同步辅助类\*\*，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrierpoint)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

# 三、JVM

## 1.JVM运行时数据区域 vs Java内存模型 （这2不一样！！！）

http://zlcalma.top/article/JVM01

https://mp.weixin.qq.com/s/kxP8usV2hHBMuapXOIeq-g

## 3.内存泄露、内存溢出以及栈溢出

### 1）内存溢出

* 堆上无内存可完成实例分配且堆无法扩展时->OutOfMemeoryError。
* 方法区(以及内部常量池)无法满足内存分配需求时->OutOfMemoryError。
* 虚拟机栈(本地方法栈)扩展时无法申请到足够的内存->OutofMemoryError。

### 2）内存泄漏

* 1.程序动态的配了内存，但是在程序结束时，没有释放这部分内存，导致那部分内存不可用。
* Java也会有内存泄漏：当分配的对象可达但已经没有任何作用时，

Stutdent s1 = new Student();Student s2 = new student();将这两个对象放入ArrayList中，在把s1和s2设为null，但是这两个对象所占内存并没有被释放，因为ArrayList中还存在对象的引用。

### 3）检测内存泄漏

可以通过一些性能监测分析工具，如JProfiler、Optimizeit Profiler。

### 4）避免内存泄漏和溢出

* 尽早释放无用对象的引用。
* 使用临时变量的时候，让引用变量在退出活动区域之后自动设置为null，暗示垃圾收集器来收集对象，防止发生内存泄漏。
* 程序进行字符串处理时，尽量避免使用String，而使用StirngBuffer,因为每个String对象都会独立占用一块区域。

### 5）什么情况下会发生栈内存溢出。

* 栈溢出就是方法执行是创建的栈帧超过了栈的深度。那么最有可能的就是方法递归调用产生这种结果。
* 堆中主要存储的是对象。如果不断的new对象则会导致堆中的空间溢出。

## 4.JVM类加载机制及其作用与对象的初始化

## 5.如何判断对象已经死亡？引用计数法 vs 可达性分析

## 6.如何回收对象？垃圾收集算法

https://mp.weixin.qq.com/s/qVRvMoqiKKwi9kEe59SEPg

## 7.Minor GC vs Full GC

## 8.常用的垃圾收集器及其特点

## 9.内存分配与回收策略

## 10.GC调优

## 11.GC调优的思路

## 12.JVM常用参数

## 13.基于JDK命令行工具监控Java进程， 如 jps,jinfo,jstat,jmap,jstack

- jsp:虚拟机进程状况工具：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类(Main Class,main()函数所在类)名称以及这些进程本地虚拟机唯一ID。

- jinfo：jinfo的作用是实时地查看和调整虚拟机各项参数。

- jstat:虚拟机统计信息监视工具，它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

- jmap：用于生成堆转储快照。jmap的作用并不仅仅是为了获取dump文件，它还可以查询finalize执行队列、Java堆和永久代的信息。

- jstack:用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈集合。

## 16.JVM内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为Eden和Survivor。

## 17.JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代

## 说说你知道的几种主要的JVM参数。

- Xms：设置堆的最小空间大小。

- Xmx：设置堆的最大空间大小。

- XX:NewSize设置新生代最小空间大小。

- XX：MaxNewSize设置新生代最大空间大小。

- Xss：设置每个线程的堆栈大小。

## 18.讲下cms和G1，包括原理，流程，优缺点。

G1，包括原理，流程，优缺点。

CMS收集器原理

g1和cms区别,吞吐量优先和响应优先的垃圾收集器选择。

## 20.简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。

### 类加载过程及类加载器：

[http://zlcalma.top/article/JVM03](http://zlcalma.top/article/JVM03)

[https://blog.csdn.net/justloveyou\_/article/details/72466105](https://blog.csdn.net/justloveyou\_/article/details/72466105)

### 破坏双亲委派模型

1.在JDK1.2之前，用户去继承java.lang.ClassLoader的唯一目的就是为了重写loadClass方法，由于用户自己重写了loadClass，那么也就是用户自己去自定义加载类，故事破坏

2.JDBC,JDNI等的SPI的加载都是父类的加载器去请求子类的加载器去加载累；

3.OSGI的热部署就是自定义类加载器机制的实现；

## 21.Java对象的创建过程

当一个对象被创建时，虚拟机就会为其分配内存来存放对象自己的实例变量及其从父类继承过来的实例变量(即使这些从超类继承过来的实例变量有可能被隐藏也会被分配空间)。在为这些实例变量分配内存的同时，这些实例变量也会被赋予默认值(零值)。在内存分配完成之后，Java虚拟机就会开始对新创建的对象按照程序猿的意志进行初始化。在Java对象初始化过程中，主要涉及三种执行对象初始化的结构，分别是 \*\*实例变量初始化、实例代码块初始化 以及 构造函数初始化。\*\*

[https://blog.csdn.net/justloveyou\_/article/details/72466416](https://blog.csdn.net/justloveyou\_/article/details/72466416)

## 22.怎么打出线程栈信息。

什么是线程堆栈？线程堆栈也称线程调用堆栈，是虚拟机中线程（包括锁）状态的一个瞬间快照，即系统在某一个时刻所有线程的运行状态，包括每一个线程的调用堆栈，锁的持有情况。虽然不同的虚拟机打印出来的格式有些不同，但是线程堆栈的信息都包含：

1、线程名字，id，线程的数量等。

2、线程的运行状态，锁的状态（锁被哪个线程持有，哪个线程在等待锁等）

3、调用堆栈（即函数的调用层次关系）调用堆栈包含完整的类名，所执行的方法，源代码的行数。

Java虚拟机提供了线程转储（thread dump）的后门，通过这个后门可以把线程堆栈打印出来。通常我们将堆栈信息重定向到一个文件中，便于我们分析，由于信息量太大，很可能超出控制台缓冲区的最大行数限制造成信息丢失。这里介绍一个jdk自带的打印线程堆栈的工具，\*\*jstack\*\*用于打印出给定的Java进程ID或core file或远程调试服务的Java堆栈信息。

示例：$jstack –l 23561 >> xxx.dump

命令 : $jstack [option] pid >> 文件

>>表示输出到文件尾部，实际运行中，往往一次dump的信息，还不足以确认问题，建议产生三次dump信息，如果每次dump都指向同一个问题，我们才确定问题的典型性。

# 四、面向对象与设计模式

## 1.面向对象的六个基本原则

## 2.单例模式（手撕各种单例模式）

工厂模式（Spring的BeanFactory,ApplicationContext）vs.抽象工厂模式

构造器模式（应用场景：解决构造复杂对象的麻烦。复杂：需要输入的参数的组合较多）

代理模式（Spring AOP(动态代理模式：2种实现方式，即jdk & 字节码方式)）

装饰器模式（应用场景：若想对一个类从不同角度进行功能扩展，例如java.io中，InputStream是一个抽象类，标准类库中提供了FileInputStream\ByteArrayInputStream等各种不同的子类，分别从不同角度对InputStream进行了功能扩展。这些不同的实现类其构造函数的输入均为InputStream（的实现类），然后对InputStream添加不同层次的逻辑，从而实现不同的功能，这就是装饰。）

观察者模式（各种事件监听器）

策略模式（比较器Comparator）

迭代器模式（ArrayList等集合框架中的迭代器）

生产者消费者模式（消息队列）要能手撕

# 五、数据结构与算法

[https://github.com/zlweiyang/leetcode/blob/master/README.md](https://github.com/zlweiyang/leetcode/blob/master/README.md)

# 六、计算机网络

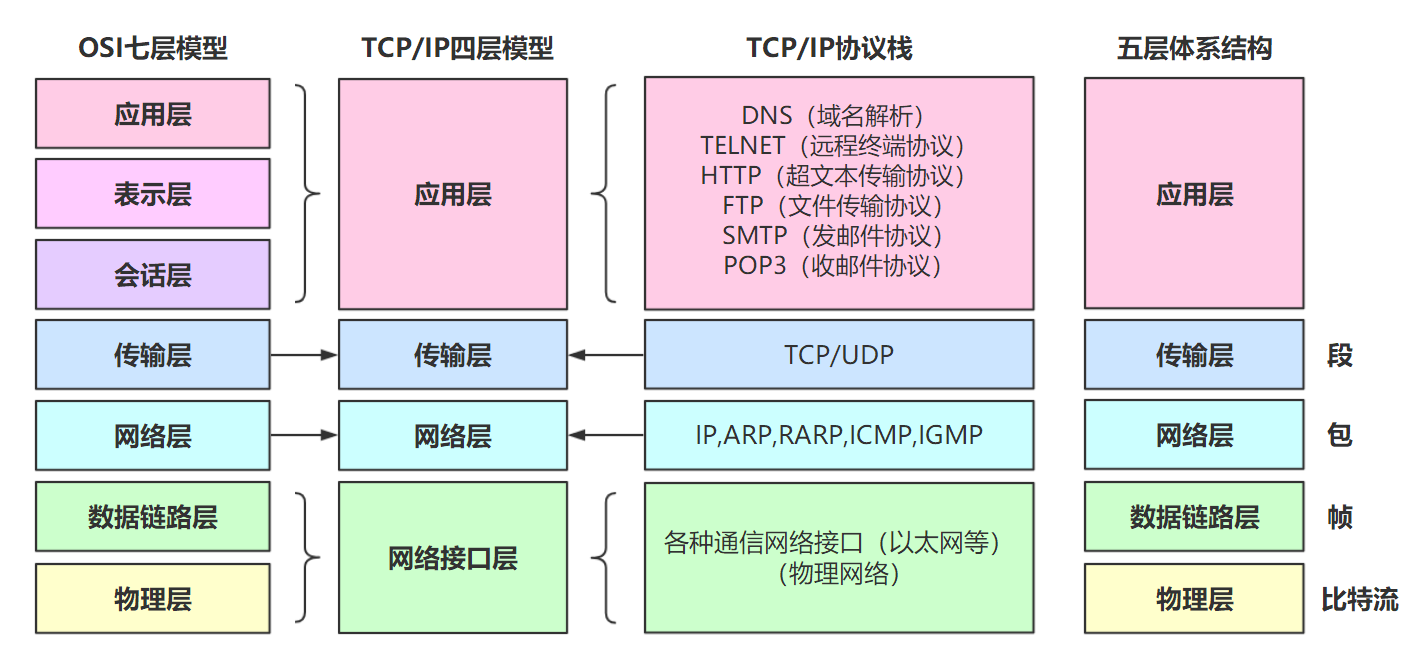
https://mp.weixin.qq.com/s/0lky--UdBNMGnaPkecDLfg

## 1.OSI七层模型 vs. TCP/IP 五层模型及各层协议

**OSI7层模型**：1.物理层,2.数据链路层,3.网络层,4.传输层,5.会话层,6.表示层,7.应用层

**TCP/IP 4层模型**:1.网络接口层,2.运输层,3.网际层,4.网络接口层

**五层协议**：1.物理层，2.数据链路层，3.网络层，4.运输层，5.应用层。



## 2.给定一个网址，访问这个网址经过了那些过程？涉及哪些协议？

https://www.javazhiyin.com/21441.html

## 3.HTTP and HTTPS

HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

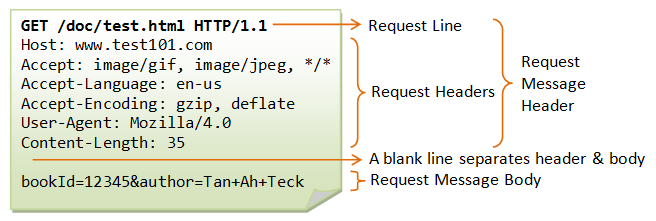
HTTPS和HTTP的区别主要如下：

-  http协议是运行在TCP之上(工作在应用层)，所有传输内容都是明文，而https运行在SSL/TLS之上(工作在传输层)，而SSL/TLS运行在TCP之上，所有传输内容都要经过加密，加密一般采用对称加密。但对称密钥的进行了非对称加密，可以进行客户端服务端的身份验证。  
-  http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。  
-  HTTP 的 URL 以 http:// 开头，而 HTTPS 的 URL 以 https:// 开头。  
-  HTTP 无需证书，而 HTTPS 需要认证证书，证书需要交费。

## 4.HTTP报文格式

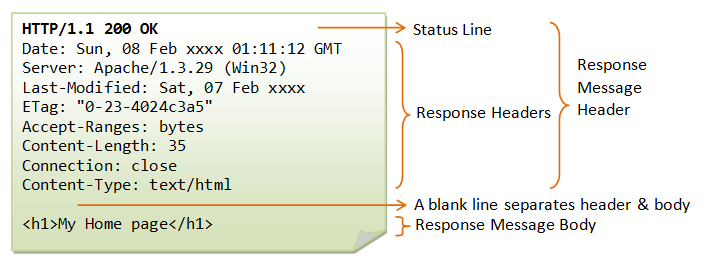
### 1)请求报文

HTTP请求报文由请求行(request line)、请求头(header)、空行和请求数据4各部分组成。



### 2）响应报文

http响应报文由状态行，响应头部，空行，响应数据组成。



1.状态码(Status Code)：描述了响应的状态。可以用来检查是否成功的完成了请求。请求失败的情况下，状态码可用来找出失败的原因。如果Servlet没有返回状态码，默认会返回成功的状态码HttpServletResponse.SC\_OK。

2.HTTP头部(HTTP Header)：它们包含了更多关于响应的信息。比如：头部可以指定认为响应过期的过期日期，或者是指定用来给用户安全的传输实体内容的编码格式。如何在Serlet中检索HTTP的头部看这里。

3.空行

4.主体(Body)：它包含了响应的内容。它可以包含HTML代码，图片，等等。主体是由传输在HTTP消息中紧跟在头部后面的数据字节组成的。

## 5.HTTP方法及相互区别

**GET**：获取资源

当前网络请求中，绝大部分使用的是 GET 方法。

\*\*HEAD：\*\*获取报文首部

和 GET 方法类似，但是不返回报文实体主体部分。主要用于确认URL的有效性以及资源更新的日期时间等。

\*\*POST：\*\*传输实体主体

POST 主要用来向指定资源提交数据进行处理请求(例如提交表单和上传文件)，POST请求会导致新的资源建立和修改已有资源，而GET主要用来获取资源。

\*\*PUT：\*\*上传文件

由于自身不带验证机制，任何人都可以上传文件，因此存在安全性问题，一般不使用该方法。

PUT /new.html HTTP/1.1 Host: example.com Content-type: text/html Content-length: 16

New File

**PATCH**：对资源进行部分修改

PUT 也可以用于修改资源，但是只能完全替代原始资源，PATCH 允许部分修改。

PATCH /file.txt HTTP/1.1 Host: www.example.com Content-Type: application/example If-Match: "e0023aa4e" Content-Length: 100

[description of changes]

\*\*DELETE：\*\*删除文件

请求服务器删除Request-URL所标识的资源。

DELETE /file.html HTTP/1.1

\*\*OPTIONS：\*\*查询支持的方法

查询指定的URL能够支持的方法。会返回 Allow: GET, POST, HEAD, OPTIONS 这样的内容。

**CONNECT：** 要求在与代理服务器通信时建立隧道

使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。

### get与post的区别

* get请求不安全而post请求时安全的。
* 浏览器对get请求的URL长度有限制(1024字节)。get请求不能代替post请求发送大量的数据。
* get请求时幂等的，所有幂等是指一次或多次请求某一个资源应该具有相同的作用。
* get请求访问时数据会在地址栏后可见，而post数据在地址栏不可见。
* get请求可以保存在cookie中，post请求不能被缓存。

## 6.HTTP的无状态

服务器不维护任何有关客户端过去所发请求的信息。

## 7.HTTP状态码



说说你知道的几种HTTP响应码，比如200, 302, 404。

- 200 OK，服务器已经成功处理了请求。  
- 301 永久移除，请求网页已经永久跳转到新的位置，服务器返回此响应时，会自动将请求跳转到新位置。

- 302 临时移动：服务器暂时从不同位置的网页响应请求，但请求者继续使用原有位置来进行以后的请求，会出现网址加持问题，搜索引擎抓取新的内容而保留旧的地址。  
- 400 Bad Request：服务器不理解请求的语法。  
- 404 Not Found：服务器找不到请求的资源。

- 502 服务器作为网关和代理，从上游服务器收到无效响应。

- 503 服务器目前暂时无法使用。

- 500 (服务器内部错误)。  
- 505 HTTP Version Not Surpported，服务器不支持请求中所用的HTTP协议版本。

## 8.HTTP 1.0 vs 1.1 vs 2.0

### 1)http1.1与http1.0的区别

1.http1.0需要**keep-alive参数**来告知服务器要建立一个长连接，而http1.1默认支持长连接。

2.HTTP 1.1支持只发送header信息(不带任何body信息)，如果服务器认为客户端有权限请求服务器，则返回100，否则返回401。客户端如果接受到100，才开始把请求body发送到服务器。这样当服务器返回401的时候，客户端就可以不用发送请求body了，节约了带宽。

3.**host域** ：http1.0 没有host域，http1.1才支持这个参数。

4.**带宽优化及网络连接的使用**，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

### 2)http2.0 与 http1.0的区别

**新的二进制格式**（Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。

**多路复用**（MultiPlexing），即连接共享，建立起一个连接请求后，可以在这个链接上一直发送，不要等待上一次发送完并且受到回复后才能发送下一个（http1.0是这样），是可以同时发送多个请求，互相并不干扰。

**header压缩**，如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0利用HPACK对消息头进行压缩传输，客服端和服务器维护一个动态链表（当一个头部没有出现的时候，就插入，已经出现了就用表中的索引值进行替代），将既避免了重复header的传输，又减小了需要传输的大小。（Hpack https://www.jianshu.com/p/f44b930cfcac）

**服务端推送**（server push），就是客户端请求html的时候，服务器顺带把此html需要的css,js也一起发送给客服端，而不像http1.0中需要请求一次html，然后再请求一次css，然后再请求一次js。

## 10.HTTP底层协作协议（如传输层为TCP协议）

## 11.TCP vs UDP

* TCP 面向连接：客户机/服务器进程间需要建立连接,UDP不需要建立连接。
* TCP是可靠的传输，UDP传输不可靠。
* TCP只能一对一通信，UDP支持一对一，一对多，多对一和多对多的交互通信。
* TCP面向字节流(流入到进程或流出进程的字节流),UDP面向数据报。
* TCP首部20字节，UDP首部开销小只有8字节。

## 12.TCP如何保证可靠传输？

* **确认和重传**：接收方收到报文就会确认，发送方发送一段时间后没有收到确认就重传。
* **数据校验**：TCP将保持它首部和数据的检验和。这是一个端到端的检验和。目的是检测数据在传输过程中的任何变化。如果收到的段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。（**不给出响应，超时重传）**
* **数据合理分片和排序**：UDP：IP数据报大于1500字节,大于MTU.这个时候发送方IP层就需要分片(fragmentation).把数据报分成若干片,使每一片都小于MTU.而接收方IP层则需要进行数据报的重组.这样就会多做许多事情,而更严重的是,由于UDP的特性,当某一片数据传送中丢失时,接收方便无法重组数据报.将导致丢弃整个UDP数据报.tcp会按MTU合理分片，接收方会缓存未按序到达的数据，重新排序后再交给应用层。
* **流量控制**：当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。
* **拥塞控制**：当网络拥塞时，减少数据的发送。

## 13.TCP的三次握手和四次挥手及其原因

<http://zlcalma.top/article/network-02>

## 14.TCP的流量控制与滑动窗口

<http://zlcalma.top/article/network-02>

## 15.TCP的拥塞控制与拥塞窗口

<http://zlcalma.top/article/network-02>

## 16.TCP的拆包

## 18.TIMEWAIT和CLOSEWAIT的区别。

## 19.如何避免浏览器缓存。

https://blog.csdn.net/u014034854/article/details/50374709

## 20.简述Http请求get和post的区别以及数据包格式。

- get请求不安全而post请求时安全的。  
- 浏览器对get请求的URL长度有限制(1024字节)。get请求不能代替post请求发送大量的数据。  
- get请求时幂等的，所有幂等是指一次或多次请求某一个资源应该具有相同的作用。  
- get请求访问时数据会在地址栏后可见，而post数据在地址栏不可见。  
- get请求可以保存在cookie中，post请求不能被缓存。

## 21.HTTP的长连接与短连接。

在HTTP/1.0中默认使用短链接，即客户端与服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。  
从HTTP/1.1开始，默认使用长连接，用来保持连接性，使用长连接的HTTP协议，会在响应头加入下面的代码：  
    Connection:keep-alive  
在使用长连接的情况下，当一个网页打开完成之后，客户端与服务端之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep alive不会永久保持，可以在不同服务器中设定时间。  
#### 什么时候用长连接，短连接？  　　  
 长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。   
 而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好

## 22.Http和https的三次握手有什么区别。

## 23.Session和cookie的区别。

**Cookie** Cookie是服务端发送到用户浏览器并保存到**本地的一小块数据**，它会在浏览器再次向同一服务器发起请求时被携带并发送到服务器上。它用于告诉服务端两个请求是否来自同一浏览器，并保持用户登录状态。

**Seesion** 用户信息也可以存储到**服务器**中，这样更加安全。比如存储在服务器上的数据库或者Redis或者其他文件中。

**1）存取方式的不同**

Cookie中只能保管ASCII字符串，假如需求存取Unicode字符或者二进制数据，需求先进行编码。Cookie中也不能直接存取Java对象。若要存储略微复杂的信息，运用Cookie是比拟艰难的。而Session中能够存取任何类型的数据，包括而不限于String、Integer、List、Map等。Session中也能够直接保管Java Bean乃至任何Java类，对象等，运用起来十分便当。能够把Session看做是一个Java容器类。

**2、隐私策略的不同**

Cookie存储在客户端阅读器中，对客户端是可见的，客户端的一些程序可能会窥探、复制以至修正Cookie中的内容。而Session存储在服务器上，对客户端是透明的，不存在敏感信息泄露的风险。

**3、有效期上的不同**

使用过Google的人都晓得，假如登录过Google，则Google的登录信息长期有效。用户不用每次访问都重新登录，Google会持久地记载该用户的登录信息。要到达这种效果，运用Cookie会是比较好的选择。只需要设置Cookie的过期时间属性为一个很大很大的数字。由于Session依赖于名为SESSIONID的Cookie，而Cookie SESSIONID的过期时间默许为–1，只需关闭了阅读器该Session就会失效，因而Session不能完成信息永世有效的效果。运用URL地址重写也不能完成。而且假如设置Session的超时时间过长，服务器累计的Session就会越多，越容易招致内存溢出。

## 24.RPC协议（经常和Dubbo一起问）

## 25.Socket编程

## 26.异步IO多路复用：select poll epoll

# 七、操作系统

## 1.内核态 vs 用户态

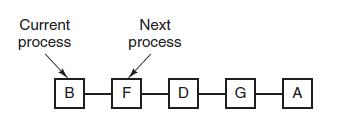
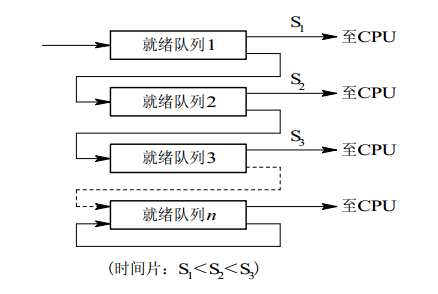
## 2.进程 vs 线程

进程是系统进行**资源分配和调度**的一个独立单位，最小的资源管理单位。线程是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，最小的CPU执行单元。

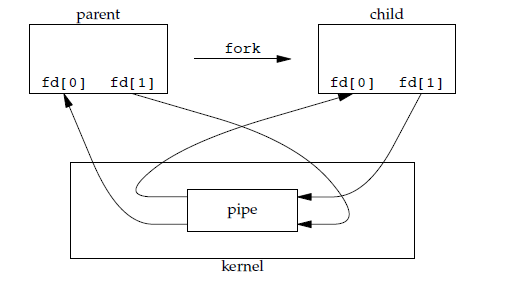
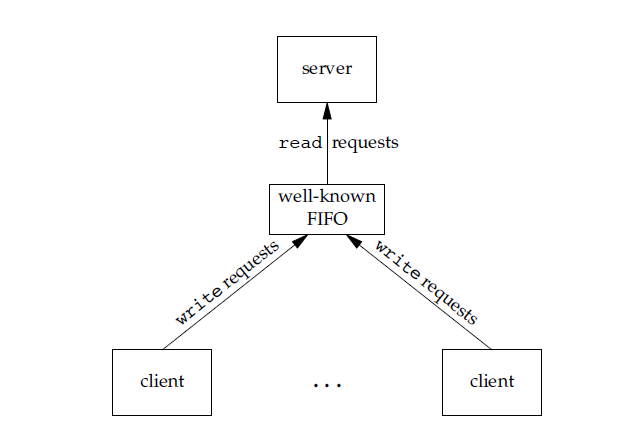
线程拥有的资源：程序计数器 寄存器 栈 状态字

进程和线程的区别   
**1. 拥有资源**：进程是资源分配的基本单位，线程不拥有资源，但可以访问线程隶属的进程资源。  
**2. 调度\***线程是独立调度的基本单位，在同一进程中，线程之间的切换不会引起进程切换，从一个进程内的线程切换到另一个进程中的线程时，会引起进程切换。  
**3.系统开销：**进程的创建和撤销，系统需要为之分配或回收资源，如内存空间、I/O设备等，其所付出的开销远大于创建或撤销线程时的开销。同样地，在进行进程切换时，需要对当前执行进程CPU环境的保存，以及对新调度进程CPU环境的设置，而线程切换时，只需要保存和设置少量寄存器内容，开销很小。  
**4.通信方面**：进程间的通信需要进程同步和互斥手段的辅助，保证数据的一致性。而线程间可以通过直接读/写同一进程中的数据段(如全局变量)来进行通信。

## 3.进程调度算法

1)1.针对批处理系统   
## 1.1先来先服务(FCFS) ##  
按照请求的顺序进行调度。有利于长作业，但不利于短作业，因为短作业必须等待前面的长作业执行完毕才能执行而长作业又需要执行很长时间，造成了短作业等待时间过长。  
## 1.2短作业(进程)优先调度算法 ##  
按照估计运行时间最短的顺序进行调度。  
这种算法可能会导致长作业饿死，处于一直等待短作业执行完毕的状态，因为如果一直有短作业到来，那么长作业永远得不到调度。  
## 1.3 最短剩余时间优先 ##  
按照剩余时间最短进行调度。  
2.交互式系统   
## 2.1 时间片轮转 ##  
将所有就绪进程按照FCFS原则排成一个队列，每次调度是，把CPU时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时将CPU的时间分配给队首的进程。  
  
**\*\*时间片轮转算法的效率和时间片的大小有很大关系：\*\***  
  
- 因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小会导致进程切换太频繁。  
- 如果时间片过长，那么实时性就得不到保障。  
  
  
## 2.2 优先级调度算法 ##  
  
为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远得不到调度，可以随着时间的推移增加等待进程的优先级。  
  
## 2.3 多级反馈队列 ##  
  
一个进程需要执行100个时间片，如果采用时间片轮转调度算法，那么需要交换100次。多级队列是为了这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如1，2，4，8.。进程在第一个队列没执行完，就会被移动到下一个队列。这种方式下，之前的进程只需要交换7次。  
  
每个队列优先权不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程，可以将这种调度算法看成**\*\*时间片轮转调度算法和优先级调度算法\*\***的结合。  
  
## 3.实时系统 ##  
  
实时系统要求一个请求在一个确定时间内得到响应。  
分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

## 4.进程间通信的几种方式

**1.管道**  
管道是通过调用pipe函数创建的，fd[0]用于读，fd[1]用于写。  
    #include <unistd.h>  
    int pipe(int fd[2]);  
**限制条件：**  
- 只支持半双工通信(单向交替传输)。  
- 只能在父子进程中使用。  
  
**2.FIFO**：也称为**命名管道**，改善了管道只能在父子进程使用的限制。  
    #include <sys/stat.h>  
    int mkfifo(const char \*path, mode\_t mode);  
    int mkfifoat(int fd, const char \*path, mode\_t mode);  
FIFO常用于客户-服务器程序中，FIFO用作汇聚点，在客户进程和服务器进程之间传递数据。  


**3.消息队列**  
  
相比于FIFO，消息队列具有以下优点：  
- 消息队列可以独立于读写进程存在，从而避免了FIFO中同步管道的打开和关闭时可能产生的困难；  
- 避免了FIFO的同步阻塞问题，不需要进程自己提供同步方法。  
- 读进程可以根据消息类型有选择的接收消息，而不像FIFO那样只能默认地接收。  
**4.信号量**  
它是一个计数器，用于为多个进程提供对共享数据对象的访问。  
**5.共享存储**  
允许多个进程共享一个给定存储区。因为数据不需要在进程间复制，所以这是进程通信最快的一种方式。但是需要借助**\*\*4.信号量\*\***用来同步对共享存储的访问。  
多个进程可以使用同一个文件映射到它们的地址空间来实现共享内存。  
**6.套接字**  
可用于不同机器之间的进程通信。

## 6.什么是死锁？哲学家就餐问题

死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

## 7.死锁的必要条件

**产生死锁的必要条件**：

* 互斥条件：所谓互斥就是进程在某一时间内独占资源。
* 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* 不剥夺条件:进程已获得资源，在末使用完之前，不能强行剥夺。
* 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

## 8.死锁的应对方法

### 1）预防死锁：

资源一次性分配：（破坏请求和保持条件）

可剥夺资源：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

### 2）避免死锁:

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得 较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的**避免死锁算法是银行家算法和安全状态算法。**

### 3）死锁检测和恢复

首先为每个进程和每个资源指定一个唯一的号码；然后建立资源分配表和进程等待表。

在检测到运行系统进入死锁状态后，进行恢复。

\*\*死锁检测算法\*\*是通过检测有向图是否存在环来实现，从一个节点出发进行深度优先搜索，对访问的结点进行标记，如果访问到已经标记的节点，就表示有向图存在环，也就是检测到死锁发生。

\*\*死锁恢复：\*\*1.利用抢占恢复；2.利用回滚恢复，3.通过杀死进程恢复

### 4）解除死锁:

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有：

**剥夺资源**：从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态；

**撤消进程**：可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态.消除为止；所谓代价是指优先级、运行代价、进程的重要性和价值等。

## 9.死锁的检测与恢复

## 10.死锁的动态避免：银行家算法

我们可以把[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F)看作是银行家，操作系统管理的资源相当于银行家管理的资金，进程向操作系统请求分配资源相当于用户向银行家贷款。

为保证资金的安全，银行家规定：

(1) 当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客；

(2) 顾客可以分期贷款，但贷款的总数不能超过最大需求量；

(3) 当银行家现有的资金不能满足顾客尚需的贷款数额时，对顾客的贷款可推迟支付，但总能使顾客在有限的时间里得到贷款；

(4) 当顾客得到所需的全部资金后，一定能在有限的时间里归还所有的资金.

[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F)按照银行家制定的规则为进程分配资源，当进程首次申请资源时，要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。当进程在执行中继续申请资源时，先测试该进程本次申请的资源数是否超过了该资源所剩余的总量。若超过则拒绝分配资源，若能满足则按当前的申请量分配资源，否则也要推迟分配。

**死锁避免-安全序列**

**安全序列**

安全序列是指对当前申请资源的进程排出一个序列，保证按照这个序列分配资源完成进程，不会发生“酱油和醋”的尴尬问题。

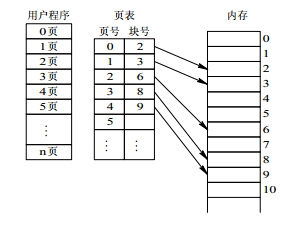
我们假设有进程P1,P2,.....Pn则安全序列要求满足：Pi(1<=i<=n)需要资源<=剩余资源 + 分配给Pj(1 <= j < i)资源  
为什么等号右边还有已经被分配出去的资源？想想银行家那个问题，分配出去的资源就好比第二个开发商，人家能还回来钱，咱得把这个考虑在内。

## 11.死锁的静态防止

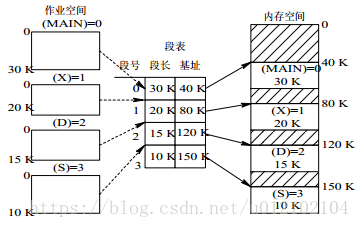
破坏死锁的必要条件之一

## 12.段页式内存管理

### 1)页存储

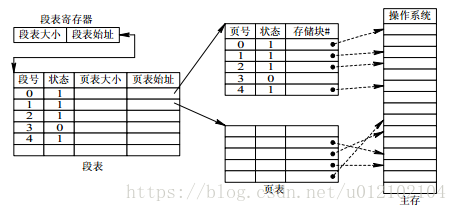
  为了便于在内存中找到进程的每个页面所对应的物理块，系统为每个进程建立一张页表，记录页面在内存中对应的物理块号，页表一般存放在内存中。在配置了页表后，进程执行时，通过查找该表，即可找到每页在内存中的物理块号。可见页表作用是实现从页号到物理块号的地址映射，这种是**页存储**管理方式。如下图所示：   


### 2)段存储

  将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，**以段为单位**，段与段在内存中可以不相邻接，也实现了离散分配，这种是**段存储**管理方式。如下图所示：   


### 3)段页存储

  作业的地址空间首先被分成若干个逻辑分段，每段都有自己的段号，然后再将每段分成若干个大小相等的页。对于主存空间也分成大小相等的页，主存的分配以页为单位，这种是**段页存储**管理方式。



## 13.页面置换算法

缺页中断：缺页中断就是要访问的页不在主存，需要操作系统将其调入主存后再进行访问。

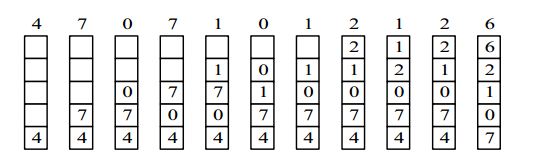
页面置换算法：在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生[缺页中断](https://baike.baidu.com/item/%E7%BC%BA%E9%A1%B5%E4%B8%AD%E6%96%AD/5029040)。当发生缺页中断时，如果操作系统内存中没有空闲页面，则[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192)必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

常见页面置换算法：

###1.最佳  
所选择的被换出的页面将是最长时间内不再被访问，通常可以保证或得最低的缺页率。是一种理论的算法，因为无法知道一个页面多长时间不再被访问。  
例如：一个系统为某进程分配三个物理块，并有如下页面引用序列：  
                 701203042  
开始运行时，先将7,0,1三个页面装入内存。当进程要访问页面2时，产生缺页中断，会将页面7换出，因为页面7再次被访问时间最长。

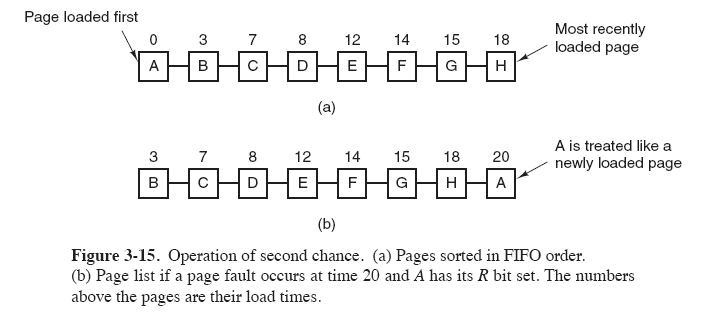
###2.先进先出  
选择换出的页面是最先进入的页面，这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即**先进入内存的页，先退出内存**。由于该算法会将经常访问的页面换出，所以缺页率较高。

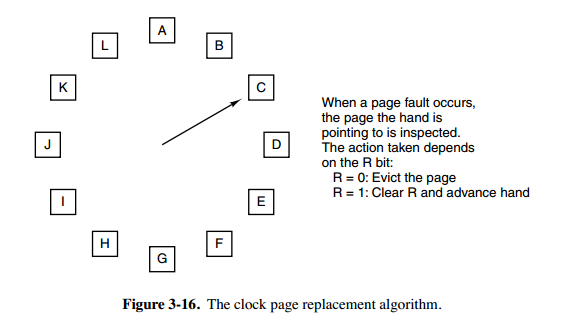
###3.最近最久未使用  
虽然无法知道将来要使用页面情况，但是可以知道过去使用页面的情况。LRU将最近最久未使用页面换出。为了实现LRU，需要在内存中维护一个**\*\*链表\*\***。当一个页面被访问时，将这个页面移到**\*\*链表表头。\*\***这样就能保证**\*\*链表表尾的页面是最近最久未访问的\*\***。  
  
由于每次都需要更新链表，因此这种方式实现的LRU代价很高。

### 4最近未使用 ###

每个页面都有两个状态位：R与M，当页面被访问的时候R=1，当页面被修改时M=1。其中R位会被定时清零。可以将页面分成以下四类：  
- R=0,M=0  
- R=0,M=1  
- R=1,M=0  
- R=1,M=1  
当页面发生中断时，NRU算法随机地从类编号最小的非空类中挑选出一个页面将他换出。NRU优先换出已经被修改的脏页面(R=0,M=1)，而不是频繁的使用干净的页面(R=1,M=0)。

###5.第二次机会算法  
 FIFO算法可能会将经常访问的页面置换出去，为了避免这个问题，进行修改，当页面被访问时设置该页面的R位为1。需要替换的时候，检查老页面的R位，如果R位是0，那么这个页面既老又没有别使用，可以被置换；如果R=1，将R清0，并把该页面放到链表的尾端，修改它的装入时间就像它刚刚被装入一样，然后继续从链表的头部开始搜索。



###6.时钟  
第二次机会算法需要在链表中移动页面，降低了效率。时钟算法使用环形链表将页面连接起来，再使用一个指针指向最老的页面。  


## 14.磁盘调度算法

## 1.先来先服务 ##  
> FCFS:First Come First Served  
  
## 2.最短寻道时间优先 ##  
> SSTF:Shortest Seek Time First  
  
## 3.电梯算法 ##

## 15.Linux系统常用的命令有哪些？是否会Shell,Python?

## 16.Linux系统下你关注过哪些内核参数，说说你知道的。

## 17.Linux下IO模型有几种，各自的含义是什么。

## 18.epoll和poll有什么区别。

# 八、数据库MySQL

## 1.数据库范式

1NF:符合1NF的关系中的每个属性都不可再分。

2NF：属性完全依赖于主键。

3NF：属性不依赖于其他非主属性[消除传递依赖]。

BCNF：在1NF基础上，任何非主属性不能对主键子集依赖。

4NF：要求把同一表内的多对多关系删除。

5NF：从最终结构建立原始结构。

## 2.数据库事务

## 4.事务特性：ACID

1.原子性（Atomicity）：原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚，回滚可以用日志来实现，日志记录着事物所执行的修改操作，在回滚时反向执行这些操作即可。

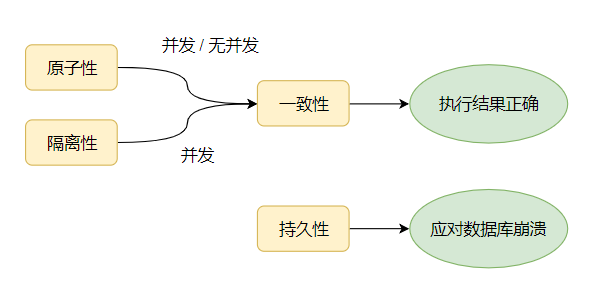
2.一致性（Consistency）：从一个一致性状态到另一个一致性状态。

例如现有完整性约束a+b=10，如果一个事务改变了a，那么必须得改变b，使得事务结束后依然满足a+b=10，否则事务失败。

3.隔离性（Isolation）：一个事务所做的修改在最终提交以前，对其它事务不可见。

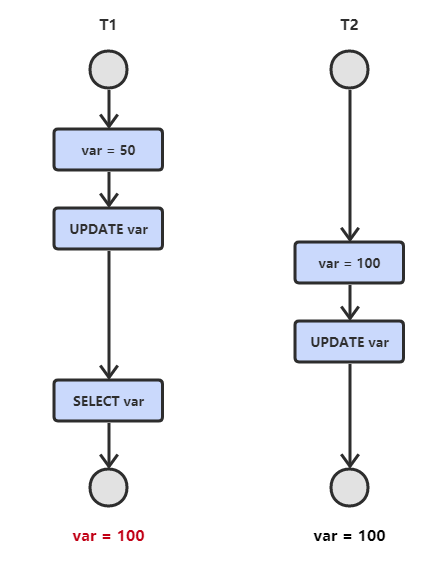
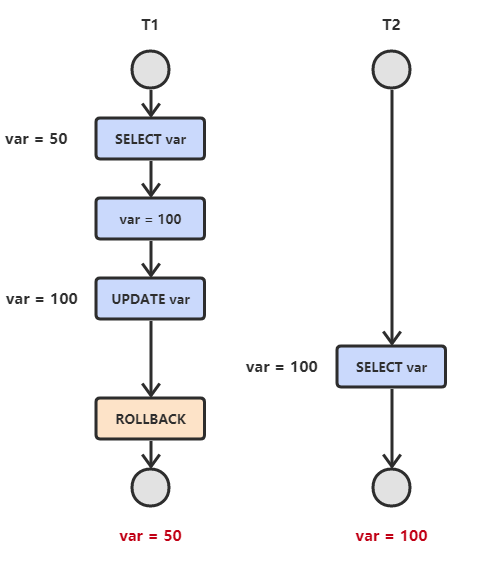
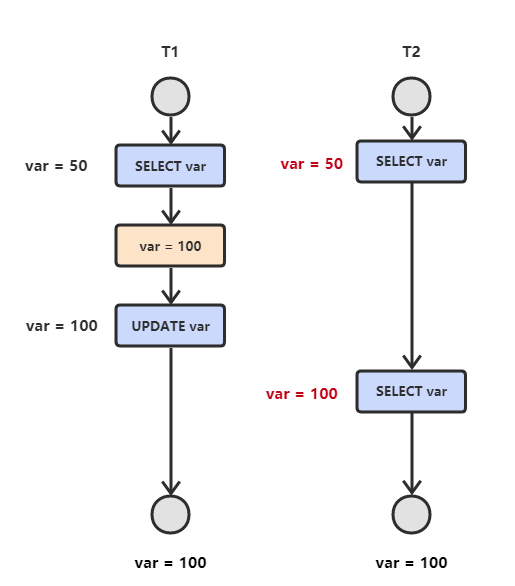
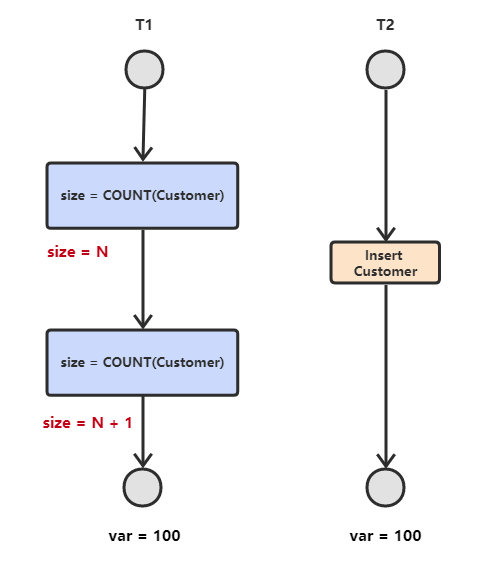
比如现有有个交易是从A账户转100元至B账户，在这个交易还未完成的情况下，如果此时B查询自己的账户，是看不到新增加的100元的。

4.(Durability) 持久性  
持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。

这四种特性之间具备一定的关系：  
-

只有满足一致性，执行结果才正确。  
 再无并发的情况下，事物串行执行，隔离性可以不考虑,只要满足原子性就能满足一致性。  
在并发情况下。多个事物并发执行，事物不仅满足原子性，还要满足隔离性，，才能满足一致性。  
事物满足持久化是为了应对数据库的崩溃。

## 5.事务隔离级别以及脏读、不可重复读、幻读

## 1.丢失修改 ##  
T1和T2两个事物都对一个数据进行修改，T1先修改，T2随后修改，T2的修改覆盖了T1的修改。  
  
事物T1本来将值更新为50，可是事物T2在T1查询之前将值修改为100，导致T1原本值丢失。  
## 2.脏读 ##  
脏读是指无效数据读出，当一个事物读取另一个事物未提交的数据时就做脏读。  
  
事物T1原本数据为50，现在将其更新为100，处于未提交状态，此时T2读取该数据，随后T1回滚，数据变为50。  
  
**\*\*解决办法\*\***：可以将事物隔离级别调整到READ\_COMMITTED(读已经提交)，即只能读取其他事物提交的数据，不能读取未提交数据。  
  
## 3.不可重复读 ##  
  
在同一个事物内，两个相同的查询返回了不同结果，即事物1分别读取事物2操作时和提交后的数据，导致读取记录不一致。  
  
  
当T1对数据未更新时，T2读取结果为50，当T2将数据更新为100并提交后，T2读取结果为100。  
  
**\*\*解决办法：\*\***将数据库事物隔离级别调整为REPEATABLE\_READ(可重复读)。即只有修改事物完成提交之后才可以读取数据。  
  
## 4.幻读 ##  
  
与不可重复读类似，幻读一般出现在插入和删除数据，导致读取数据记录不一致。  
  
  
  
当T1读取某个范围的数据，T2在这个范围内插入新的数据，T1再次读取这个范围的数据，此时的结果与第一次读取结果不一致。  
  
**\*\*解决方法\*\***：把数据库隔离级别调整为SERIALIZABLE\_READ(串行化),即在操作事物完成数据处理之前，任何其他事物不可以添加新数据。即所有事物一个接着一个执行，串行化要求在执行范围查询的时候，需要获取范围锁。

## 6.分布式事务

## 7.数据库索引

https://mp.weixin.qq.com/s/jet58AXUQZNLTYbzuEraJA

## 8.索引的应用场景

## 9.索引特点

1.可以加快数据库的检索速度；2.降低数据库插入、修改、删除等维护的速度；3.只能创建在表上，不能创建在视图上；4.既可以直接创建又可以间接创建；5.可以在优化隐藏中使用索引；6.使用查询处理器执行SQL语句，在一个表上。一次只能使用一个索引。

## 10.索引的优点和缺点

### 1）索引的优点：

* 创建唯一索引，保证数据库表中每一行数据的唯一性；
* 大大加快数据的检索速度，这是创建索引的最主要因素；
* 加快数据库表之间的连接。特别是在实现数据参考完整性方面特别有意义；
* 通过使用索引，可以在查询中使用优化隐藏器，提高系统的性能。

### 2）索引的缺点

* 创建索引和维护索引要耗费时间，这种时间随着数量的增加而增加；
* 索引需要占用物理空间，除了数据表占用物理空间，如果建立聚簇索引，那么需要的空间就会更大；
* 当对表中的数据进行增加、删除、修改时，索引也需要维护，降低数据维护的速度。

## 11.索引的分类

* 组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一
* 聚集索引：定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。
* 非聚集索引：唯一索引 普通索引 主键索引 全文索引
* UNIQUE(唯一索引)：不可以出现相同的值，可以有NULL值
* INDEX(普通索引)：允许出现相同的索引内容
* PROMARY KEY(主键索引)：不允许出现相同的值
* fulltext index(全文索引)：可以针对值中的某个单词，但效率确实不敢恭维

## 12.索引选择原则

## 13.索引的优化

## 14.索引失效

## 15.索引的底层实现（B-Tree vs B+Tree）

## 16.对于建立索引的列，数据均匀分布好还是不均匀好？

## 17.数据库的锁

### 1）行锁 vs 表锁

- **\*\*1.表级锁：\*\***MySQL中粒度最大的锁，对整张表加锁，资源消耗比较少，加锁快不会出现死锁。，MyISAM和InnoDB引擎都支持表级锁。  
- **\*\*2.行级锁：\*\***MySQL中粒度最小的锁，对当前操作的行加锁，行级锁能大大减少数据库操作的冲突，加锁粒度最小，并发度高，但加锁开销最大，加锁慢，会出现死锁。InnoDB支持行级锁。

虽然行级锁并发度高、粒度小，但是表级锁也是必需的：  
1.事务更新大表中的大部分数据直接使用表级锁效率更高；  
2.事务比较复杂，使用行级锁很可能引起死锁导致回滚。

### 2）读锁 vs 写锁

- \*\***共享锁\***\*(也称为读锁)，多个线程可以同一时间访问同一资源，而不受干扰。

- \*\***独占锁\***\*(也称写锁、排他锁、互斥锁)，会阻塞其它写锁和读锁。

### 3）乐观锁 vs 悲观锁以及如何实现？（MVCC）

**\*\*\*\***乐观锁**\*\*\*\***  
用数据版本(Version)记录机制实现，这是乐观锁最常用的一种实现方式。所谓数据版本就是为数据增加一个版本标识。一般是通过为数据库表增加一个数字类型的“version”字段实现。当读取数据时，将version字段的值一同读出，数据没更新一次。对version值加1.当我们提交更新时，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。  
1.设计数据库表三个字段，分别是id,value,version  
  
    select id ,value,version from TABLE where id=#{id}  
  
2.每次更新表中的value字段时，为了防止冲突：  
  
    update TABLE  
    set value = 2，version=version+1;  
    where id=#{id} and version=#{version};  
      
**\*\*6.悲观锁\*\***  
  
悲观锁在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能对相同数据进行操作。悲观锁是由数据库自己实现，需要使用时，直接调用数据库相关语句就好啦。实际上**共享锁与排它锁**都是悲观锁的具体实现。在使用悲观锁需要闭关MySQL数据库的自动提交属性。  
  
其基本操作如下面所示：  
  
    set autocommit=0;  
    //1.开始事务  
    begin;begin work;start transaction;  
    2.查询表信息  
    select status from TABLE where id=1 for update;  
    3.插入一条数据  
    insert into TABLE(id,value) values(2,2);  
    4.修改数据  
    update TABLE set value=2 where id=1;  
    5.提交事务  
    commit;commit work

加锁的方式

分布式锁

## 18.数据库存储引擎比较

### InnoDB与MyISAM引擎区别

* MyISAM是非事物安全型的，而InnoDB是事物安全型的。
* MyISAM锁的粒度是表级，而Innodb支持行级锁。
* MyISAM支持全文索引，InnoDB在MySQL5.6之后支持全文索引。
* MyISAM相对简单，所以在效率上要由于InnoBD，小型应用可以考虑使用MyISAM。
* MyISAM不支持外键，而InnoDB支持。
* InnoDB表比MyISAM表更安全，可以在保证数据不丢失的情况下，切换非事物表到事物表(alter table tablename)。

## 19.MySQL连接池对比

## 20.MySQL命令

## 21.数据库分库分表

## 22.主从复制

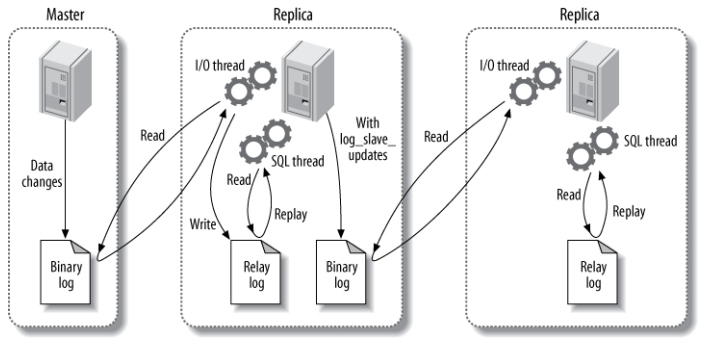
### 1）主从复制

主从设计可以将数据从一台服务器(Master)复制到多台独立的服务器(Slaves),主要涉及三个线程:binlog线程、I/O线程和SQL线程。

- \*\*binlog线程\*\*：负责将主服务器上的数据更改写入二进制文件(binlog)中。

- \*\*I/O线程\*\*：负责从主服务器上读取二进制文件，并写入从服务器的中继日志中。

- \*\*SQL线程\*\*：负责读取中继日志并重放其中的SQL语句。



### 2读写分离

主服务器用来处理写操作以及实时性要求比较高的读操作，而从服务器用来处理读操作。

读写分离常用代理方式实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

\*\*为什么读写分离能够提高数据库的性能？\*\*

主从服务器负责各自的读和写，极大程度上缓解了锁的争用；

从服务器可以配置MyISAM引擎，提升查询性能以及节约系统开销；

增加冗余，提高可用性。

# E:\database_learning\master-slave-proxy.png九、数据库Redis

https://mp.weixin.qq.com/s/SAagS7l-czClD7ZGxJgH1A

## 1.Redis的优点与缺点

### 优点：

1. 读写性能好，速度快。

2. 支持数据持久化，支持AOF和RDB两种持久化的方式(相比memcached)

3. 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

4. 数据结构丰富：除了支持string类型的value外，还支持hash，set，list，sortedset,list等数据结构。

### 缺点：

1. 较难支持在线扩容，当集群容量达到上限时在线扩容会变得很复杂，所以上限时需要预估内存资源空间。

2. redis不具备自动容灾和恢复功能，主机从机的宕机都会导致前端部分读写数据读写请求失败，需要等待重启或者手段切换前端的IP才能恢复。

## 2.Redis的数据类型、底层实现以及各种数据类型的使用场景

Redis 是速度非常快的非关系型（NoSQL）内存键值数据库，可以存储键和五种不同类型的值之间的映射。

键的类型只能为字符串，值支持五种数据类型：字符串、列表、集合、散列表、有序集合。

### 使用场景

**\*\*String\*\***

- 计数器：可以对 String 进行自增自减运算，从而实现计数器功能。

- 可完全替换memcached,存储简单k-v。

- 存储少量热点数据,分布式锁(秒杀系统有应用)。

- 会话缓存(session cache)。

**\*\*List(列表)\*\***

- 取出最新的几个消息。

- 作为队列。

- 发布/订阅消息，还可以可作为基于发布/订阅的脚本触发器。

**\*\*SET\*\***

一些会不重复的数据,进行交集,并集,差集等灵活操作，比如\*\*关注人,粉丝,\*\*可分别放在一个set中。

**\*\*Hash\***\*

查找表，主要用来存储对象。

**\*\*sorted set\*\***

排序(排行榜)：sorted set 比set多了一个score的(分数),可以通过这个scored来排序,相同scored的数据按自然顺序,比如微信朋友圈就可以用sorted set 来排序,分数用时间来表示。

## 3.Redis是单线程吗？单线程的Redis为啥这么快(每秒10万次读写操作)？

单进程单线程，利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。

- 1.redis是基于\*\*内存的\*\*，内存的读写速度非常快(cpu不是redis的瓶颈，唯一的瓶颈是机器的内存，为了不浪费CPU的闲置，可以开启多个进程，因为数据之间不存较大依赖性，只要客户端能够分清楚可以所在的进程即可)；

- 2.redis是\*\*单线程的\*\*，省去了很多上下文切换线程的时间；不需要考虑各种锁的问题，不存在加锁和释放锁操作，也不会因为出现死锁导致性能消耗。为了提高高并发的能力。可以采用单线程、多进程的集群方案。

- 3.redis使用\*\*多路复用技术\*\*，可以处理并发的连接。非阻塞IO 内部实现采用epoll，采用了epoll+自己实现的简单的\*\*事件框架\*\*。epoll中的读、写、关闭、连接都转化成了事件，然后利用epoll的多路复用特性，绝不在io上浪费一点时间。这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗，且Redis在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了Redis具有很高的吞吐量。

- 4. \*\*数据结构\*\*性能丰富，Redis全程使用hash结构，读取速度快，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，跳表，使用有序的数据结构加快读取的速度。

[https://mp.weixin.qq.com/s/HuwgTvNtmHJ9tpF7V53opg](https://mp.weixin.qq.com/s/HuwgTvNtmHJ9tpF7V53opg)

## 4.Redis持久化的方式

Redis 支持两种持久化策略：RDB快照和AOF日志，而 Memcached 不支持持久化，默认采用RDB持久化方式。

[http://zlcalma.top/article/redis02](http://zlcalma.top/article/redis02)

## 5.谈谈Redis的过期策略以及内存淘汰机制

- **过期策略**：即redis针对过期的key使用的清除策略，策略为，定时删除，定期删除和惰性删除。

- **内存淘汰机制**：即内存占用达到内存限制设定值时触发的redis的淘汰策略来删除键。

### 过期策略

- 1.定时策略：设置键的过期时间时，创建一个Timer，当过期时间到临时，立刻删除键。内存友好型策略，一旦键过期，就会被删除，并释放所占用的内存，对CPU不友好，当一批数量比较多的键过期时，正好遇上CPU紧张的时段，这时候需要的是CPU处理能力，而不是内存，显然CPU时间用在删除过期键上，会对服务器的响应时间和吞吐量造成影响。另外当前Redis时间事件（无序链表O(N)无法高效处理大量时间事件，所以定时删除并不是一种好的定时删除策略。

- 2.惰性删除：不管过期的键，在这种策略下，当键在键空间中被取出时，首先检查取出的键是否过期，若过期删除该键，否则，返回该键。惰性删除依赖过期键的被动访问，对于内存不友好，如果一些键长期没有被访问，会造成内存泄露（垃圾数据占用内存）。我们知道，Redis是依赖内存的，所以惰性删除也不是一个好的策略。

- 3.定期删除：由定时删除算法，定期的去检查一定的数据库，删除一定的过期键。

通过合理的删除操作执行的时长和频率，达到合理的删除过期键。

### 六种淘汰策略

- volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰。

- volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰。

- volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰。

- allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰。

- allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰。

- no-enviction（驱逐）：禁止驱逐数据。

-

作为内存数据库，出于对性能和内存消耗的考虑，Redis 的淘汰算法实际实现上并非针对所有 key，而是\*\*抽样一小部分并且从中选出被淘汰的 key\*\*。使用 Redis 缓存数据时，为了提高缓存命中率，需要保证缓存数据都是热点数据。可以将内存最大使用量设置为热点数据占用的内存量，然后启用 allkeys-lru 淘汰策略，将最近最少使用的数据淘汰。

## 6.如何解决Redis和和数据库双写一致性的问题？

https://mp.weixin.qq.com/s/yQjTddLWy-oSYPYxyGVeKw

## 7.如何应对缓存穿透与缓存雪崩的问题？

## 8.如何解决Redis的并发竞争Key问题

## 9.Redis的并发竞争问题如何解决，了解Redis事务的CAS操作吗。

## 10.Redis的选举算法和流程是怎样的。

## 11.redis的集群怎么同步的数据的。

## 12.Reids的主从复制机制原理。

## 13.redis的一致性哈希算法

## 14.当散列类型的value值非常大的时候怎么进行压缩， 用redis怎么实现摇一摇与附近的人功能

## 15.为什么redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。

如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。

但是缺点是，如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

## 16.MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？

redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

## 17.读写分离模型与数据分片模型

# 十、Web与常见框架

## 1.Servlet的生命周期（问的很少）

## 3.分布式Session

## 4.IOC应用及原理

IOC利用Java的反射机制，将本来是由调用者来创建对象实例，统一交由spring来管理创建，将对象交给容器管理，使用时只需要在spring中的配置文件中配置相应的bean，以及设置相关属性，让spring容器来生成类的实例对象以及管理对象。在spring容器启动时，spring会把配置文件中配置的bean都初始化好，然后在需要调用时，就把已经初始化话的那些bean分配给你需要调用的这些bean的类。

IOC的一个重点是在系统运行时，动态的向某个对象提供它所需要的其他对象。这个过程主要通过DI(依赖注入)来实现。

## 5.AOP应用及原理

AOP技术利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“\*\*Aspect\*\*”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如\*\*权限认证\*\*、\*\*日志\*\*、\*\*事务处理\*\*。Aop的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

Spring Aop的实现方式有两种：

### ### 1. 基于接口的动态代理。

在java的动态代理机制中，有两个重要的类或接口，一个是\*\*InvocationHandle\*\*r(Interface)、另一个则是\*\*Proxy(Class)\*\*，这一个类和接口是实现我们动态代理所必须用到的。首先我们先来看看java的API帮助文档是怎么样对这两个类进行描述的：

\*\*1.InvocationHandler:\*\*

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke 方法来进行调用。我们来看看InvocationHandler这个接口的唯一一个方法 invoke 方法Object invoke(Object proxy, Method method, Object[] args) throws Throwable 。

- proxy:指代我们所代理的那个真实对象。

- method:指代的是我们所要调用真实对象的某个方法的Method对象。

- args:指代的是调用真实对象某个方法时接受的参数。

\*\*2.Proxy\*\*

Proxy这个类的作用就是用来动态创建一个代理对象的类，它提供了许多的方法，但是我们用的最多的就是 newProxyInstance 这个方法：

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException

### ### 2. 基于子类化的CGLIB代理。

CGLIB创建的动态代理性能比JDK创建的动态代理对象性能高，但是CGLIB在创建代理对象时所花费的时间比JDK多，所以对于单例对象，因为不需要频繁的创建对象，用CGLIB合适。但是CGLIB是采用动态创建子类的方法，对于final方法，无法进行代理。运行时动态的生成一个被代理的子类(通过ASM字节码处理框架实现)，子类重写了被代理类中所有非final的方法。在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势切入横切逻辑。

两种实现方式的区别：

- JDK动态代理：基于标准JDK的动态代理功能，只针对实现了接口的业务对象。

- CGLIB：通过动态地对目标对象进行子类化来实现AOP代理，需要指定@EnableAspectJAutoProxy(proxyTargetClass = true)来强制使用，当业务对象没有实现任何接口时默认会使用CGLIB。

## 6.Spring涉及的设计模式

第一种：简单工厂

又叫做静态工厂方法（StaticFactoryMethod）模式，但不属于23种GOF设计模式之一。 简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。 spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

第二种：工厂方法（Factory Method）

通常由应用程序直接使用new创建新的对象，为了将对象的创建和使用相分离，采用工厂模式,即应用程序将对象的创建及初始化职责交给工厂对象。

一般情况下,应用程序有自己的工厂对象来创建bean.如果将应用程序自己的工厂对象交给Spring管理,那么Spring管理的就不是普通的bean,而是工厂Bean。

第三种：单例模式（Singleton）

保证一个类仅有一个实例，并提供一个访问它的全局访问点。spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是是任意的java对象。核心提示点：Spring下默认的bean均为singleton，可以通过singleton=“true|false” 或者scope=“？”来指定。

第四种：适配器（Adapter）

在Spring的Aop中，使用的Advice（通知）来增强被代理类的功能。Spring实现这一AOP功能的原理就使用代理模式（1、JDK动态代理。2、CGLib字节码生成技术代理。）对类进行方法级别的切面增强，即，生成被代理类的代理类，并在代理类的方法前，设置拦截器，通过执行拦截器重的内容增强了代理方法的功能，实现的面向切面编程。

第五种：包装器（Decorator）

在我们的项目中遇到这样一个问题：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。我们以往在spring和hibernate框架中总是配置一个数据源，因而sessionFactory的dataSource属性总是指向这个数据源并且恒定不变，所有DAO在使用sessionFactory的时候都是通过这个数据源访问数据库。但是现在，由于项目的需要，我们的DAO在访问sessionFactory的时候都不得不在多个数据源中不断切换，问题就出现了：如何让sessionFactory在执行数据持久化的时候，根据客户的需求能够动态切换不同的数据源？我们能不能在spring的框架下通过少量修改得到解决？是否有什么设计模式可以利用呢？

首先想到在spring的applicationContext中配置所有的dataSource。这些dataSource可能是各种不同类型的，比如不同的数据库：Oracle、SQL Server、MySQL等，也可能是不同的数据源：比如apache 提供的org.apache.commons.dbcp.BasicDataSource、spring提供的org.springframework.jndi.JndiObjectFactoryBean等。然后sessionFactory根据客户的每次请求，将dataSource属性设置成不同的数据源，以到达切换数据源的目的。

spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。基本上都是动态地给一个对象添加一些额外的职责。

第六种：代理（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。从结构上来看和Decorator模式类似，但Proxy是控制，更像是一种对功能的限制，而Decorator是增加职责。

spring的Proxy模式在aop中有体现，比如JdkDynamicAopProxy和Cglib2AopProxy。

第七种：观察者（Observer）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

第八种：策略（Strategy）

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。spring中在实例化对象的时候用到Strategy模式在SimpleInstantiationStrategy中有如下代码说明了策略模式的使用情况：

第九种：模板方法（Template Method）

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method模式一般是需要继承的。这里想要探讨另一种对Template Method的理解。spring中的JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。但是变化的东西是一段代码，而且这段代码会用到JdbcTemplate中的变量。怎么办？那我们就用回调对象吧。在这个回调对象中定义一个操纵JdbcTemplate中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到JdbcTemplate，从而完成了调用。这可能是Template Method不需要继承的另一种实现方式吧。

## 7.谈谈Spring Bean的生命周期和作用域

1. 实例化一个Bean--类似于new。

2. 按照Spring上下文对实例化的Bean进行配置--IOC与DI注入

3. 如果Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String)方法，此处传递的就是Spring配置文件中Bean的id值。

4. 如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory(SetBeanFactory(BeanFactory))传递的是Spring工厂自身(可以用这个方式来获取其它的Bean，只需在Spring配置文件中配置一个普通的Bean就可以)；

5. 如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

6. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj,String s)方法

7. 如果Bean在Spring配置文件中配置了init-method属性会调用其配置的初始化方法。

8. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj,String s)方法。

9. 当Bean不再需要时，会经过清理阶段，如果实现了DisposableBean这个接口，会调用其实现的destory()方法。

10. 最后，如果这个Bean的Spring配置中配置了destory-method属性，会自动调用其配置的销毁方法。

作用域 |字符|描述

---|---|---

单例 singleton | 每次在应用中只创建一个实例

原型 prototype | 每次注入时都新建一个实例

会话 session | 为每次会话创建一个实例

请求 request|为每次请求创建一个实例

## 8.Spring事务

### 事务的实现方式

（1）编程式事务管理对基于 POJO 的应用来说是唯一选择。我们需要在代码中调用beginTransaction()、commit()、rollback()等事务管理相关的方法，这就是编程式事务管理。

（2）基于 TransactionProxyFactoryBean的声明式事务管理

（3）基于 @Transactional 的声明式事务管理

（4）基于Aspectj AOP配置事务

## 9.讲讲Spring事务的传播级别。

1） PROPAGATION\_REQUIRED，

默认的spring事务传播级别，使用该级别的特点是，\*\*如果上下文中已经存在事务，那么就加入到事务中执行\*\*，\*\*如果当前上下文中不存在事务，则新建事务执行\*\*。所以这个级别通常能满足处理大多数的业务场景。

2）PROPAGATION\_SUPPORTS ，从字面意思就知道，supports，支持，该传播级别的特点是，\*\*如果上下文存在事务，则支持事务加入事务，如果没有事务，则使用非事务的方式执行\*\*。所以说，并非所有的包在transactionTemplate.execute中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3）PROPAGATION\_MANDATORY ， 该级别的事务要求上下文中\*\*必须要存在事务\*\*，否则就会抛出异常！配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4）PROPAGATION\_REQUIRES\_NEW ，从字面即可知道，new，\*\*每次都要一个新事\*\*务，该传播级别的特点是，每次都会新建一个事务，并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。这是一个很有用的传播级别，举一个应用场景：现在有一个发送100个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送100封红包，然后再记录发送日志，发送日志要求100%的准确，如果日志不准确，那么整个父事务逻辑需要回滚。

怎么处理整个业务需求呢？就是通过这个PROPAGATION\_REQUIRES\_NEW 级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5）PROPAGATION\_NOT\_SUPPORTED ，这个也可以从字面得知，not supported ，不支持，当前级别的特点就是上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务极可能的缩小。我们知道一个事务越大，它存在的风险也就越多。所以在处理事务的过程中，要保证尽可能的缩小范围。比如一段代码，是每次逻辑操作都必须调用的，比如循环1000次的某个非核心业务逻辑操作。这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑周全的异常情况。所以这个事务这个级别的传播级别就派上用场了。用当前级别的事务模板抱起来就可以了。

6）PROPAGATION\_NEVER，该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，而PROPAGATION\_NEVER传播级别要求上下文中不能存在事务，一旦有事务，就抛出runtime异常，强制停止执行！这个级别上辈子跟事务有仇。

7）PROPAGATION\_NESTED ，字面也可知道，nested，嵌套级别事务。该传播级别特征是，如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

Spring如何管理事务的

## 9.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处理器映射器。

- 3.处理器映射器根据请求url找到具体的处理器，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通过HandlerAdapter处理器适配器调用处理器

- 5.执行处理器(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ViewReslover视图解析器

- 9.ViewReslover解析后返回具体View

- 10.DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。

- 11.DispatcherServlet响应用

### Springmvc 中DispatcherServlet初始化过程。

1.Servlet加载（监听器之后即执行）Servlet的init()

2.加载配置文件

3.从ServletContext拿到spring初始化springmvc相关对象

4.放入ServletContext

## 10.Spring vs Spring Boot

## 11.简单讲讲tomcat结构，以及其类加载器流程，线程模型等。

## 12.tomcat如何调优，涉及哪些参数 。

## 13.说说你对Spring的理解，非单例注入的原理？它的生命周期？循环注入的原理。

## 14.springboot启动机制。

## 15.Mybatis的底层实现原理。

https://mp.weixin.qq.com/s/AqqSRYPdUbruSAhhmqVneA

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 十一、分布式与中间件

## 1.了解微服务，Spring Cloud

## 2.Netty (网络框架)

netty的线程模型，netty如何基于reactor模型上实现的。

什么是TCP粘包，拆包。解决方式是什么。

netty的fashwheeltimer的用法，实现原理，是否出现过调用不够准时，怎么解决。

netty的心跳处理在弱网下怎么办。

netty的通讯协议是什么样的。

## 3.Dubbo

服务调度，涉及服务发现、配置管理、弹性伸缩、故障恢复等

资源调度，涉及对底层资源的调度使用，如计算资源、网络资源和存储资源等。

流量调度，涉及路由、负载均衡、流控、熔断等。

数据调度，涉及数据复本、数据一致性、分布式事务、分库、分表等

容错处理，涉及隔离、幂等、重试、业务补偿、异步、降级等。

自动化运维，涉及持续集成、持续部署、全栈监控、调用链跟踪等

用java自己实现一个LRU。

分布式集群下如何做到唯一序列号。

设计一个秒杀系统，30分钟没付款就自动关闭交易。

如何使用redis和zookeeper实现分布式锁？有什么区别优缺点，会有什么问题，分别适用什么场景。

如果知道redlock，讲讲他的算法实现

分布式事务的原理，优缺点，如何使用分布式事务，2pc 3pc 的区别，解决了哪些问题，还有哪些问题没解决，如何解决，你自己项目里涉及到分布式事务是怎么处理的。

什么是一致性hash。

什么是restful，讲讲你理解的restful。

如何设计一个良好的API。

如何设计建立和保持100w的长连接。

解释什么是MESI协议(缓存一致性)。

说说你知道的几种HASH算法，简单的也可以。

什么是paxos算法， 什么是zab协议。

一个在线文档系统，文档可以被编辑，如何防止多人同时对同

一份文档进行编辑更新。

线上系统突然变得异常缓慢，你如何查找问题。

说说你平时用到的设计模式。

Dubbo的原理，有看过源码么，数据怎么流转的，怎么实现集群，负载均衡，服务注册

和发现，重试转发，快速失败的策略是怎样的 。

一次RPC请求的流程是什么。

自己实现过rpc么，原理可以简单讲讲。Rpc要解决什么问题。

异步模式的用途和意义。

编程中自己都怎么考虑一些设计原则的，比如开闭原则，以及在工作中的应用。

设计一个社交网站中的“私信”功能，要求高并发、可扩展等等。 画一下架构图。

MVC模式，即常见的MVC框架。

聊下曾经参与设计的服务器架构并画图，谈谈遇到的问题，怎么解决的。

应用服务器怎么监控性能，各种方式的区别。

如何设计一套高并发支付方案，架构如何设计。

如何实现负载均衡，有哪些算法可以实现。

Zookeeper的用途，选举的原理是什么。

Zookeeper watch机制原理。

请思考一个方案，实现分布式环境下的countDownLatch。

后台系统怎么防止请求重复提交。

描述一个服务从发布到被消费的详细过程。

讲讲你理解的服务治理。

如何做到接口的幂等性。

如何做限流策略，令牌桶和漏斗算法的使用场景。

什么叫数据一致性，你怎么理解数据一致性。

分布式服务调用方，不依赖服务提供方的话，怎么处理服务方挂掉后，大量无效资源请求的浪费，如果只是服务提供方吞吐不高的时候该怎么做，如果服务挂了，那么一会重启，该怎么做到最小的资源浪费，流量半开的实现机制是什么。

dubbo的泛化调用怎么实现的，如果是你，你会怎么做。

远程调用会有超时现象，如果做到优雅的控制，JDK自带的超时机制有哪些，怎么实现的

消息队列的使用场景。

消息的重发，补充策略。

如何保证消息的有序性。

用过哪些MQ，和其他mq比较有什么优缺点，MQ的连接是线程安全的吗，你们公司的MQ服务

架构怎样的。

MQ系统的数据如何保证不丢失。

rabbitmq如何实现集群高可用。

kafka吞吐量高的原因。

kafka架构和原理

利用mq怎么实现最终一致性。

使用kafka有没有遇到什么问题，怎么解决的。

MQ有可能发生重复消费，如何避免，如何做到幂等。

MQ的消息延迟了怎么处理，消息可以设置过期时间么，过期了你们一般怎么处理。

# 十二、系统设计