1. **如何避免死锁**

那么，通常可以用如下方式避免死锁的情况：

1. 避免一个线程同时获得多个锁；
2. 避免一个线程在锁内部占有多个资源，尽量保证每个锁只占用一个资源；
3. 尝试使用定时锁，使用lock.tryLock(timeOut)，当超时等待时当前线程不会阻塞；
4. 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况
5. **并发和并行的概念**

并发指的是多个任务交替进行，而并行则是指真正意义上的“同时进行”

1. **创建线程的三个方法**

 通过继承Thread类，重写run方法；

 通过实现runable接口；

 通过实现callable接口这三种方式，

1. **线程的状态**

**NEW 初始状态**

**Runnable 运行状态**

**Blocked 阻塞状态**

**Waiting 等待状态**

**Time-Waiting 超时等待状态**

**Terminated 终止状态**

1. **sleep() VS wait()**

sleep()方法是Thread的静态方法，而wait是Object实例方法

wait()方法必须要在同步方法或者同步块中调用，也就是必须已经获得对象锁。而sleep()方法没有这个限制可以在任何地方种使用。另外，wait()方法会释放占有的对象锁，使得该线程进入等待池中，等待下一次获取资源。而sleep()方法只是会让出CPU并不会释放掉对象锁；

sleep()方法在休眠时间达到后如果再次获得CPU时间片就会继续执行，而wait()方法必须等待Object.notift/Object.notifyAll通知后，才会离开等待池，并且再次获得CPU时间片才会继续执行

yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

1. **synchronized实现原理**

**它最大的特征就是在同一时刻只有一个线程能够获得对象的监视器（monitor），从而进入到同步代码块或者同步方法之中，即表现为互斥性（排它性）**

synchronized可以用在**方法**上也可以使用在**代码块**中，锁的对象有所不同。

## 对象锁（monitor）机制

任意一个对象都拥有自己的监视器，**每个对象拥有一个计数器，当线程获取该对象锁后，计数器就会加一，释放锁后就会将计数器减一**。

执行**monitorenter**指令获取monitor锁，执行**monitorexit**指令，释放对象锁

1. **CAS的操作过程**

**V 内存中存放的实际值；**

**O 从内存中获取的旧值；**

**N 更新的新值**

V和O相同时 更新 V和O不相同 返回V

**Synchronized 是一种悲观阻塞同步锁**

**CAS 时一种乐观非阻塞同步锁**

1. **锁一共有4种状态，**

**无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态**

这三种锁是指锁的状态，并且是针对Synchronized。这三种锁的状态是通过对象监视器在对象头中的字段来表明的。

偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。

轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。

重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

**偏向锁首先对比对象头和栈帧中存放线程的id，再看偏向锁标志，没有再使用CAS竞争锁。偏向锁等到竞争出现才释放锁。在释放偏向锁时，会等待全局安全点。**

**轻量级锁，线程使用CAS获取锁，在解锁时，如果出现竞争，升级为重量级锁，会阻塞其他线程。**

1. **Volatile**

**Synchronized：保证可见性和原子性，**

**Volatile：保证可见性，但不保证操作的原子性，volatile++这种复合操作不具有原子性**

**被volatile修饰的变量能够保证每个线程能够获取该变量的最新值，从而避免出现数据脏读的现象。**

写操作的时候会多出**Lock前缀的指令**

**原理（缓存一致性）**

1.Lock前缀的指令会引起处理器缓存写回内存；

2.一个处理器的缓存回写到内存会导致其他处理器的缓存失效；

3.当处理器发现本地缓存失效后，就会从内存中重读该变量数据，即可以获取当前最新值。

1. **AQS(模板方法设计模式)**

同步器是用来构建锁和其他同步组件的基础框架，它的实现主要依赖一个int成员变量来表示同步状态以及通过一个FIFO队列构成等待队列

它的**子类必须重写AQS的几个protected修饰的用来改变同步状态的方法**，其他方法主要是实现了排队和阻塞机制。

AQS同步抽象队列是一个带有头节点的双向链表

**在获取同步状态时，AQS维护一个同步队列，获取同步状态失败的线程会加入到队列中进行自旋；移除队列（或停止自旋）的条件是前驱节点是头结点并且成功获得了同步状态。在释放同步状态时，同步器会调用unparkSuccessor()方法唤醒后继节点**

1. **读写锁**

**读写所允许同一时刻被多个读线程访问，但是在写线程访问时，所有的读线程和其他的写线程都会被阻塞**

1. **Condition简介**

**Object的wait和notify/notify是与对象监视器配合完成线程间的等待/通知机制，Condition与Lock配合完成等待通知机制，**

**Condition能够支持多个等待队列（new 多个Condition对象），而Object方式只能支持一个；**

**Condition实现的是一个单向队列，**

**对象Object对象监视器上只能拥有一个同步队列和一个等待队列，而并发包中的Lock拥有一个同步队列和多个等待队列**。

**垃圾回收**

1. 如何判断一个对象可否被回收

**引用计数算法**

**可达性分析算法**

1. 垃圾收集算法

**标记 - 清除**

**标记 - 整理**

**复制**

HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1，回收新生代。

每次回收有多于 10% 的对象存活，需要依赖于老年代进行空间分配担保

**分代收集**

1. 垃圾收集器

HotSpot 虚拟机中的 7 个垃圾收集器

### CMS 收集器

* 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
* 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
* 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿。
* 并发清除：不需要停顿。

### G1 收集器

G1 把堆划分成多个大小相等的独立区域（Region）

每个小空间可以单独进行垃圾回收

* 初始标记
* 并发标记
* 最终标记：为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要停顿线程，但是可并行执行。
* 筛选回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

**内存分配策略**

**对象优先在 Eden 分配**

**大对象直接进入老年代**

**长期存活的对象进入老年代**

**对象一定年龄进入老年代**

**空间分配担保**

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间

不满足上条，启用空间分配担保，老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小

# 类加载机制

* **加载（Loading）**
* 通过类的完全限定名称获取定义该类的二进制字节流。
* 将该字节流表示的静态存储结构转换为方法区的运行时存储结构。
* 在内存中生成一个代表该类的 Class 对象，作为方法区中该类各种数据的访问入口。
* **验证（Verification）**
* 确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求
* **准备（Preparation）**
* 类变量是被 static 修饰的变量，准备阶段为类变量分配内存并设置初始值，使用的是方法区的内存
* 实例化不是类加载的一个过程，类加载发生在所有实例化操作之前
* **解析（Resolution）**
* 将常量池的符号引用替换为直接引用的过程
* **初始化（Initialization）**
* 初始化阶段是虚拟机执行类构造器 <clinit>() 方法的过程
* **使用（Using）**
* **卸载（Unloading）**

**数据库**

1. **没有事务隔离带来的问题**

**更新数据丢失**：两事务同时更新，一个失败回滚覆盖另一个事务的更新。或事务1执行更细操作，在事务1结束前事务2也更新，则事务1的更细结果被事务2的覆盖了。

**脏读**：事务T2读取到事务T1修改了但是还未提交的数据，之后事务T1又回滚其更新操作，导致事务T2读到的是脏数据。

**不可重复读**：事务T1读取某个数据后，事务T2对其做了修改，当事务T1再次读该数据时得到与前一次不同的值。

**幻读**：事务T1读取在读取某范围数据时，事务T2又插入一条数据，当事务T1再次数据这个范围数据时发现不一样了，出现了一些“幻影行”。

1. **事务隔离的级别**

**未提交读（Read Uncommitted）-----** 一事务写时，禁止其他事务写

**提交读（Read Committed）--------一事务写时，禁止其他事务读写**

**可重复读（Repeatable Read）---一事务写时，禁止其他事务读写、一事务读时，禁止其他事务写**

**串行化（Serializable）---------一事务写时禁止其他事务读写、一事务读时禁止其他事务读写**

1. **MVCC (Multiversion Concurrency Control)，即多版本并发控制技术**

**在mysql的InnoDB中，**

事务以排他锁的形式修改原始数据

把修改前的数据存放于undo log，通过回滚指针与主数据关联

修改成功（commit）啥都不做，失败则恢复undo log中的数据（rollback）

**MVCC**

每行数据都存在一个版本字段，每次数据更新时都更新该版本。

修改时Copy出当前版本随意修改，各个事务之间无干扰

保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃copy（rollback）

二者最本质的区别是，当修改数据时是否要排他锁定

Innodb的实现真算不上MVCC，因为并没有实现核心的多版本共存，undo log中的内容只是串行化的结果，记录了多个事务的过程，不属于多版本共存.

但当事务影响到多行数据时，理想的MVCC据无能为力了.

**例子：**如果T1执行理想的MVCC，修改Row1成功，而修改Row2失败，此时需要回滚Row1，但因为Row1没有被锁定，其数据可能又被Transaction2所修改，如果此时回滚Row1的内容，则会破坏Transaction2的修改结果，导致Transaction2违反ACID。

**InnoDB的MVCC,是通过在每行记录后面保存两个隐藏的列来实现的,这两个列，分别保存了这个行的操作事务id，一个保存的是行的删除事务id**

1. **为什么索引结构默认使用B-Tree，而不是hash，二叉树，红黑树？**

hash：虽然可以快速定位，但是没有顺序，IO复杂度高。

二叉树：树的高度不均匀，不能自平衡，查找效率跟数据有关（树的高度），并且IO代价高。

红黑树：树的高度随着数据量增加而增加，IO代价高。

1. **为什么官方建议使用自增长主键作为索引**

自增主键是连续的，在使用B+树插入过程中尽量减少页分裂，移动的频率

**HTTP**

| **状态码** | **类别** | **含义** |
| --- | --- | --- |
| 1XX | Informational（信息性状态码） | 接收的请求正在处理 |
| 2XX | Success（成功状态码） | 请求正常处理完毕 |
| 3XX | Redirection（重定向状态码） | 需要进行附加操作以完成请求 |
| 4XX | Client Error（客户端错误状态码） | 服务器无法处理请求 |
| 5XX | Server Error（服务器错误状态码） | 服务器处理请求出错 |

**204 No Content** ：请求已经成功处理，不需要返回数据时使用。

**206 Partial Content** ：客户端进行了范围请求。

**301 Moved Permanently** ：永久性重定向

**302 Found** ：临时性重定向

**303 See Other** ：临时性重定向，客户端应该采用 GET 方法

**307 Temporary Redirect** ：临时重定向，客户端应该采用 POST方法

**400 Bad Request** ：请求报文中存在语法错误。

**401 Unauthorized** ：则表示用户认证失败。

**403 Forbidden** ：请求被拒绝。

**500：**表示服务器内部错去

**502：错误网关**

**503：表示服务器超负载或正停机维护**

**505：http版本不支持**

301 302 403 500百度面试

**Java知识**

**Java具有面向对象的三大特性，封装，继承和多态。**

**封装：就是属性的私有化，根据set和get来进行访问，增加安全性和简化编程。**

**继承：将多个相同的属性和方法提取出来，当成父类，实现代码的复用。**

**多态：编译时多态，重载。运行时多态，重写**

**方法的重写和重载**

**方法的重写发生在运行时。因为在编译时，编译器是无法知道我们到底是调用父类的方法还是子类的方法，相反的，只有在实际运行的时候，我们才知道应该调用哪个方法。**

**方法的重载发生在编译时。在编译过程中，编译器必须根据参数类型以及长度来确定到底是调用的哪个方法。**

**三个范式是什么**

**第一范式：字段是最小的的单元不可再分**

**第二范式：表中的字段必须完全依赖于全部主键而非部分主键**

**第三范式：非主键外的所有字段必须互不依赖**

**抽象类和接口**

**抽象类和接口都不能直接实例化**

**接口是设计的结果 ，抽象类是重构的结果**

**接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现**

**接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变**

**接口是like a 抽象类是is a**

**CorrentHashMap的工作原理**

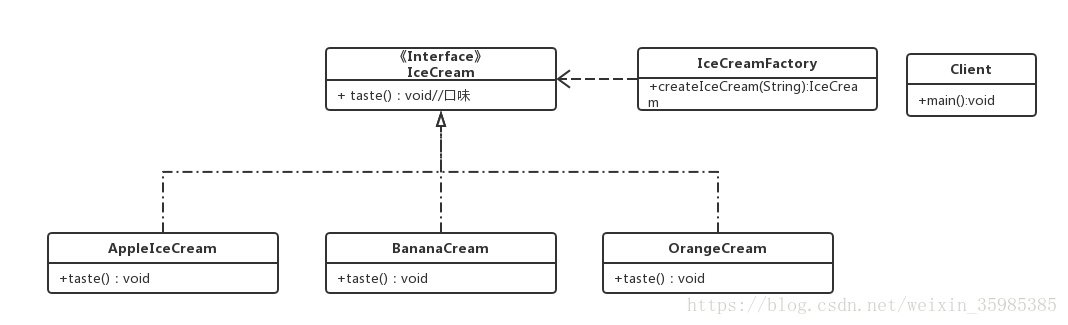
ConcurrentHashMap是线程安全的

ConcurrentHashMap是采用分离锁的方式，局部锁定

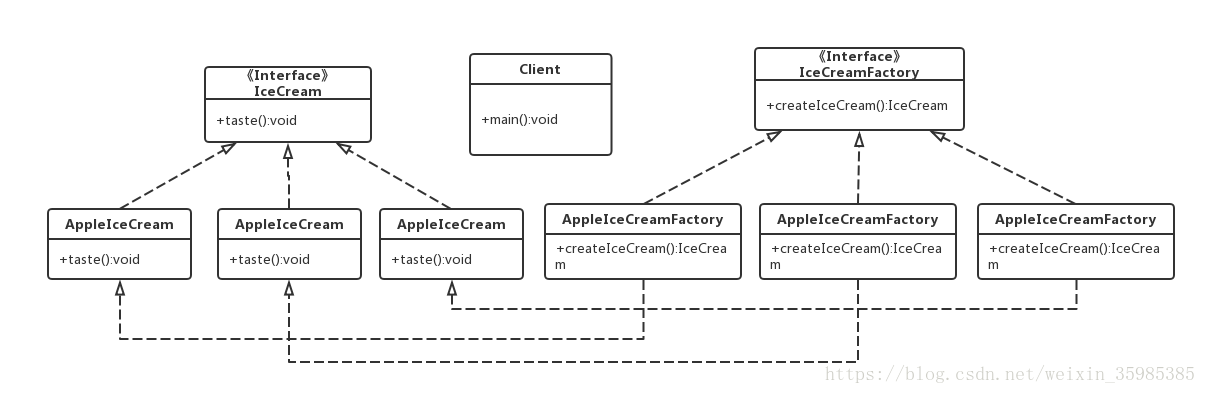
默认并发级下会创建包含16个Segment对象的数组

在Java8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

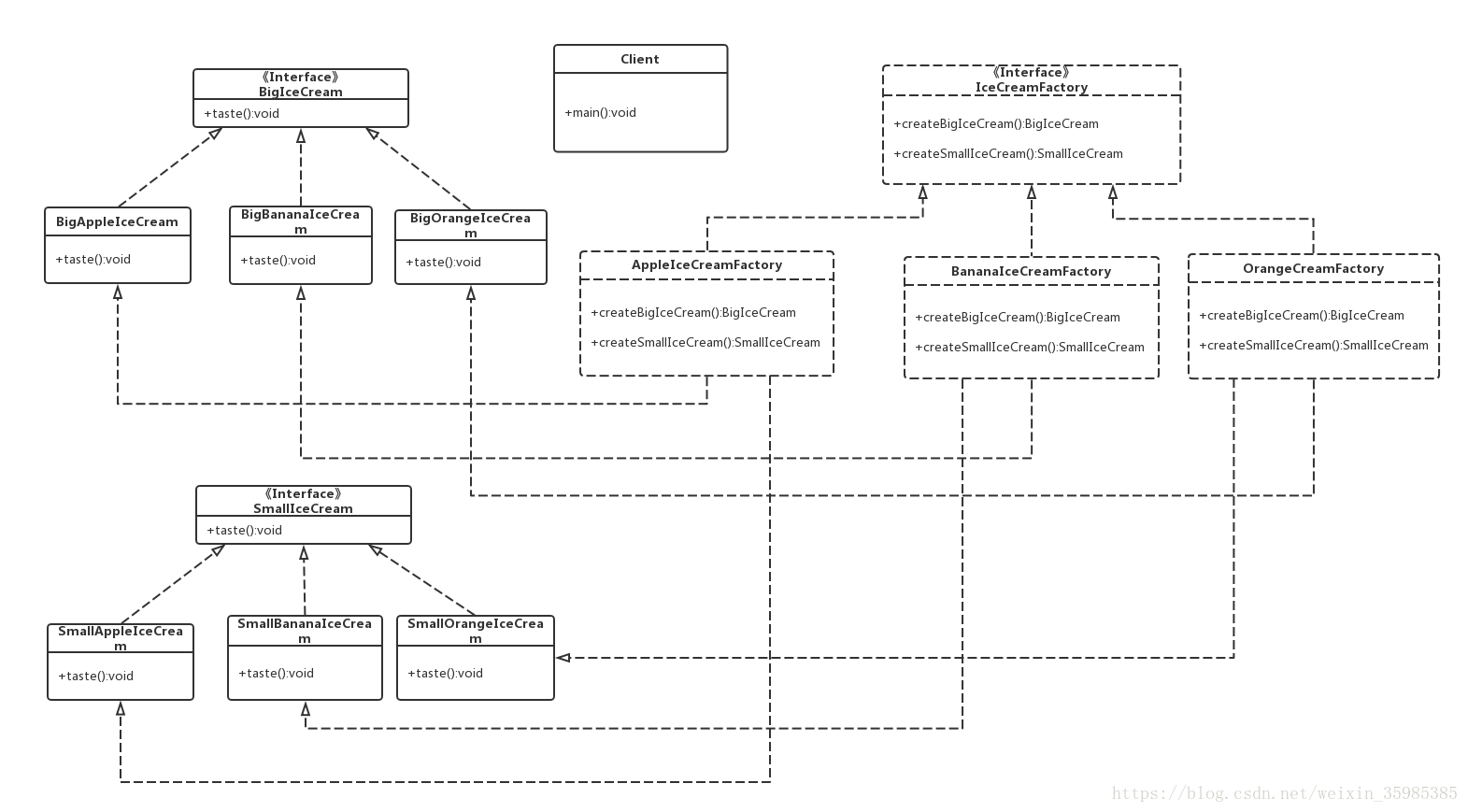
**工厂模式**

**简单工厂模式**

**工厂方法模式**

****

**抽象工厂模式**

****

**分库分表：**存于一个库的数据分散到多个库中，把存于一个表的数据分散到多个表中

**分库分表的方式**

**水平拆分：**如果是因为表中的数据量过于庞大，则可以采用水平切分

采用hash来定位数据分布的表得分位置

（1）每个库（表）的**结构**都一样

（2）每个库（表）的**数据**都不一样，没有交集

（3）所有库（表）的并集是全量数据

**垂直拆分：**将一个属性较多，一行数据较大的表，将不同的属性拆分到不同的表中

（1）每个库（表）的**结构**都不一样

（2）一般来说，每个库（表）的**属性**至少有一列交集，一般是主键

（3）所有库（表）的并集是全量数据

**堆的插入和删除**

**堆是一种逻辑上的数据结构，其物理存储，是按完全二叉树的顺序存储在数组中。**

**插入，是在数组最后面插入一个元素，然后按自下向上进行调整**

**删除，是只能删除根节点，即数组第一个元素，将最后一个元素移到前面，进行自上而下的调整**

**TCP的拥塞控制机制是什么？请简单说说。**

**答：我们知道TCP通过一个定时器（timer）采样计算RTO，但是，如果网络上的延时突然增加，那么，TCP对这个事做出的应对只有重传数据，然而重传会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这就导致了恶性循环，最终形成“网络风暴” —— TCP的拥塞控制机制就是用于应对这种情况。**

**首先需要了解一个概念，为了在发送端调节所要发送的数据量，定义了一个“拥塞窗口”（Congestion Window），在发送数据时，将拥塞窗口的大小与接收端ack的窗口大小做比较，取较小者作为发送数据量的上限。**

**拥塞控制主要是四个算法：**

**1.慢启动：意思是刚刚加入网络的连接，一点一点地提速，不要一上来就把路占满。 呈指数让升**

**阈值ssthresh（slow start threshold），是一个上限，就会进入“拥塞避免算法”**

**2.拥塞避免：当拥塞窗口 cwnd 达到一个阈值时，窗口大小不再呈指数上升，而是以线性上升，避免增长过快导致网络拥塞。**

**拥塞发生：当发生丢包进行数据包重传时，表示网络已经拥塞。分两种情况进行处理： 1.等到RTO超时，重传数据包**

**3.进入慢启动过程**

**2.在收到3个相同ACK时就开启重传，而不用等到RTO超时**

**进入快速恢复算法——Fast Recovery**

1. **快速恢复：至少收到了3个Duplicated Acks，说明网络也不那么糟糕，可以快速恢复。**

**GET和POST区别**

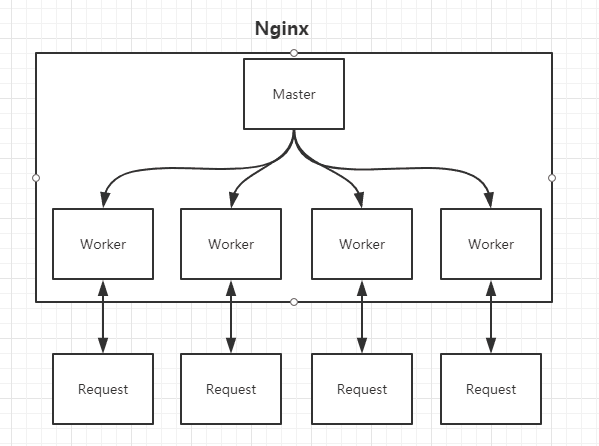
1.最直观的区别就是GET把参数包含在URL中，POST通过request body传递参数。

2.都是HTTP协议中的发送请求的方法，在使用的时候，我们规定各自的传送的方式，但是由于浏览器和服务器，对于url数据长度的限制，而post没有这个烦恼

3.GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，返回200

POST产生两个TCP数据包，浏览器先发送header，服务器正常响应100，浏览器再发送data，返回200

**Nginx请求处理 多进程+异步非阻塞IO事件模型**来处理各种连接请求

****

**线程池的种类和核心参数**

**核心参数 1.**corePoolSize：线程池的核心线程数量

2.maximumPoolSize：线程池中允许的最大线程数

3.keepAliveTime：当池中线程数大于核心线程数时，该时间为余下线程的最大空闲存活时间

4.workQueue：工作队列，存放将要执行的任务的地方

**5.** threadFactory：创建线程的线程工厂

1.固定数量的线程池

2.并行线程池

3.只有一个线程的线程池

4.缓存线程池

5.延时线程池

**char可以存汉字吗，底层怎么存的**

byte、short、char类型在java底层是以32位补码的形式进行存储和表示的

**Spring拦截器怎么使用，Controller是单例**

* 1. SpringMVC的controller默认就是单例的
  2. 在单例的bean中切记声明成员属性（如Map、List集合来缓存数据），是线程不安全的

## 解决线程不安全方案

**1、不要在controller中定义成员变量。**  
**2、万一必须要定义一个非静态成员变量时候，则通过注解@Scope(“prototype”)，将其设置为多例模式。**  
**3、在Controller中使用ThreadLocal变量**

**NIO与AIO的区别以及各自的作用**  
java NIO ：同步非阻塞，服务器实现模式为一个请求一个线程，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。  
Java 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先完成了再通知服务器应用去启动线程进行处理，  
NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器。  
AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器。

**.IOC的实现原理**  
Spring的IOC实现原理就是工厂模式加反射机制，通俗来讲就是根据给出的类名（字符串方式）来动态地生成对象，这种编程方式可以让对象在生成时才被决定到底是哪一种对象。  
把IOC容器的工作模式看做是工厂模式的升华，可以把IOC容器看作是一个工厂，这个工厂里要生产的对象都在配置文件中给出定义，然后利用编程语言提供的反射机制，根据配置文件中给出的类名生成相应的对象。  
Spring支持三种依赖注入方式，分别是属性（Setter方法）注入，构造注入和接口注入。

**数据库调优思路的思路。**  
1）.慢查询的开启并捕获  
2）.explain+慢sql分析  
3）.show profile查询sql在mysql服务器里面的执行细节和生命周期情况  
4）.sql数据库服务器的参数调优

**GC过程**

JVM GC只回收堆区和方法区

堆有存放对象的两大区域，新生代和老年代，新生代又分成三个区域，一个Eden和两个Survivor区，默认空间比例是8：1：1。新实例的对象首先存放在Eden区，当Eden空间不足，则进行一个minorGC，将存活的对象放在其中一个Survivor中，并且每次minorGC对象年龄加一，当这个Survivor中空间不足，将进行minorGC时把存活的对象复制到另一个Survivor，清空此Survivor，当对象的年龄达到阈值时，将把对象放进老年代，当老年代的空间不足时则进行一个FullGC,或当一个大对象（需要大量连续存储空间的对象）进行一次minorGC存活下来，可直接进入老年代，避免Eden与Survivor之间的大量的复制。

持久代，也称之为方法区

用于保存类常量以及字符串常量。注意，这个区域不是用于存储那些从老年代存活下来的对象，这个区域也可能发生GC。发生在这个区域的GC事件也被算为 Major GC 。只不过在这个区域发生GC的条件非常严苛，

必须符合以下三种条件才会被回收：

1、所有实例被回收

2、加载该类的ClassLoader 被回收

3、Class 对象无法通过任何途径访问（包括反射）

**JVM调优命令**

**jstat：JVM**自带命令行工具，可用于统计内存分配速率、GC 次数，GC 耗时。

**jmap：**JVM自带命令行工具，可用于了解系统运行时的对象分布。

**Jinfo：**用来查看正在运行的 Java 应用程序的扩展参数，包括 Java System 属性和 JVM 命令行参数。

**sql查询:部门工资前三高的员工和部门工资最高的员工**

**SELECT Department.Name AS Department, e1.Name AS Employee, e1.Salary AS Salary  
FROM Employee e1  
JOIN Department ON e1.DepartmentId = Department.Id  
WHERE 3 > (  
SELECT COUNT(DISTINCT e2.Salary)   
FROM Employee e2  
WHERE e2.Salary > e1.Salary AND e1.DepartmentId = e2.DepartmentId  
)  
ORDER BY Department.Name, e1.Salary DESC**

**MyBatis将配置文件中的每一个<mapper>**

**节点抽象为一个 Mapper 接口，而这个接口中声明的方法和跟<mapper>**

**节点中的<select|update|delete|insert>节点项对应，即<select|update|delete|insert>**

**节点的id值为Mapper接口中的方法名称，parameterType值表示Mapper对应方法的入参类型，而resultMap值则对应了Mapper接口表示的返回值类型或者返回结果集的元素类型。**

**Mybatis核心组件**

**SqlSession**

**作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能**

**Executor**

**MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护StatementHandler**

**封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。**

**ParameterHandler**

**负责对用户传递的参数转换成JDBC Statement 所需要的参数，**

**ResultSetHandler**

**负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；**

**TypeHandler**

**负责java数据类型和jdbc数据类型之间的映射和转换**

**MappedStatement**

**MappedStatement维护了一条<select|update|delete|insert>节点的封装，**

**SqlSource**

**负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回**

**BoundSql**

**表示动态生成的SQL语句以及相应的参数信息**

**Configuration**

**MyBatis所有的配置信息都维持在Configuration对象之中。**

**Mybatis整个过程：**

**通过SqlSessionFactoryBuilder加载配置文件configuration构建出SqlSessionFactory，SqlSessionFactory的实例直接开启一个SqlSession，通过SqlSession实例获得Mapper对象并运行Mapper映射的SQL语句，完成对数据库的CRUD和事务提交，之后关闭SqlSession。**

**1、加载mybatis全局配置文件，解析配置文件，MyBatis基于XML配置文件生成Configuration，和一个个MappedStatement（包括了参数映射配置、动态SQL语句、结果映射配置），其对应着<select | update | delete | insert>标签项。**

**2、SqlSessionFactoryBuilder通过Configuration对象生成SqlSessionFactory，用来开启SqlSession。**

**3、SqlSession对象完成和数据库的交互：**

**a、用户程序调用mybatis接口层api（即Mapper接口中的方法）b、SqlSession通过调用api的Statement ID找到对应的MappedStatement对象**

**c、通过Executor（负责动态SQL的生成和查询缓存的维护）将MappedStatement对象进行解析，sql参数转化、动态sql拼接，生成jdbc Statement对象**

**d、JDBC执行sql。**

**e、借助MappedStatement中的结果映射关系，将返回结果转化成HashMap、JavaBean等存储结构并返回**

**求n\*m网格内矩形的数目**

**N\*（n+）/2 \*m\*（m+1）/2既包括正方形和长方形**

**Spring中**BeanFactory定义了IOC容器的最基本形式，并提供了IOC容器应遵守的的最基本的接口

FactoryBean一般情况下，Spring通过反射机制利用<bean>的class属性指定实现类实例化Bean，用户可以通过实现该接口定制实例化Bean的逻辑。

**ConcurrentHashMap**

1.数据结构：取消了Segment分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。

2.保证线程安全机制：JDK1.7采用segment的分段锁机制实现线程安全，其中segment继承自ReentrantLock。JDK1.8采用CAS+Synchronized保证线程安全。

3.锁的粒度：原来是对需要进行数据操作的Segment加锁，现调整为对每个数组元素加锁（Node）。

4.链表转化为红黑树:定位结点的hash算法简化会带来弊端,Hash冲突加剧,因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。

5.查询时间复杂度：从原来的遍历链表O(n)，变成遍历红黑树O(logN)。

**StringBuffer和 StringBuilder**

（1）如果要操作少量的数据用 String；

（2）多线程操作字符串缓冲区下操作大量数据 StringBuffer；

（3）**单线程操作字符串缓冲区下操作大量数据 StringBuilder**。

Java为什么不支持多继承

类定义属性和方法 描述某一类事物的抽象 ，一个子类就应该是其父类代表的事物中的某个更具体的类别  
而接口定义的是行为 并不限于任何具体意向

## string不可变的好处

**1. 可以缓存 hash 值**

因为 String 的 hash 值经常被使用，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

**2. String Pool 的需要**

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。

**3. 安全性**

String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 的那一方以为现在连接的是其它主机，而实际情况却不一定是。

**4. 线程安全**

String 不可变性天生具备线程安全，可以在多个线程中安全地使用。

**线程池：**

 线程池按以下行为执行任务

1. 当线程数小于核心线程数时，创建线程。
2. 当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
3. 当线程数大于等于核心线程数，且任务队列已满
   1. 若线程数小于最大线程数，创建线程
   2. 若线程数等于最大线程数，抛出异常，拒绝任务

**核心参数设置：**

* corePoolSize=1/**/核心线程数**
* queueCapacity=Integer.MAX\_VALUE//**队列容量**
* maxPoolSize=Integer.MAX\_VALUE//**最大线程数**
* keepAliveTime=60s//**核心线程空闲时间**
* allowCoreThreadTimeout=false//**核心线程数超时销毁**
* rejectedExecutionHandler=AbortPolicy()/**/拒绝策略**
* 需要根据几个值来决定
  + tasks ：每秒的任务数，假设为500~1000
  + taskcost：每个任务花费时间，假设为0.1s
  + responsetime：系统允许容忍的最大响应时间，假设为1s

**corePoolSize = 每秒需要多少个线程处理**

**queueCapacity = (coreSizePool\*taskcost)\*responsetime**

**maxPoolSize =（最大任务数-队列容量）/每个线程每秒处理能力**

**rejectedExecutionHandler 任务不重要可丢弃，任务重要则要利用一些缓冲机制来处理**

**数据库为什么B+树做索引而不用B-树**

**1.： B+树只有叶节点存放数据，其余节点用来索引，而B-树是每个索引节点都会有Data域。**

**2.： B+树所有的Data域在叶子节点，并且所有叶子节点之间都有一个链指针。 这样遍历叶子节点就能获得全部数据，这样就能进行区间访问啦。在数据库中基于范围的查询是非常频繁的，而B树不支持这样的遍历操作。**

**3.：数据库系统的设计者巧妙利用了磁盘预读原理，**