**new关键字执行过程**

1. 在常量池中定位这个类的符号引用。如果这个符号引用代表的类没有被加载则需要执行类加载过程。

2. 在堆中为这个对象分配内存。

如何保证并发情况下多个线程同时在堆中分配内存的线程安全问题？两种解决方案：①采用CAS配上失败重试保证更新操作的原子性；②在堆中为每一个线程分配一个分配缓冲区。每个线程都在自己的缓冲区中分配。只有自己的缓冲区用完了去申请新的缓冲区才需要同步锁定。

3. 将分配的内存空间初始化为零值。

4. 设置对象头。将GC分代年龄、是否使用偏向锁、哈希码等信息放在对象头中。

5. 执行构造函数。

**类加载**

**加载**：获取二进制字节流、从二进制字节流中构建方法区中运行时的数据结构、在堆中生成java.lang.Class对象，作为方法区这个类的各个数据的访问入口

**验证**：确保字节流中包含的信息符合规范，也就是说这一步需要进行语法分析、语义分析以及各种数据的验证操作。

**准备**：正式为静态变量分配内存并设置初始值。

**解析**：将符号引用替换为直接引用的过程

**初始化**：对类的初始化操作，实际上是执行<clinit>()方法。如果一个类没有静态语句块也没有对静态变量的赋值操作，则不会产生<clinit>().

**卸载**：该类所有实例对象已被GC、该类没有在任何其它地方被引用、该类的类加载器的实例被GC。

**重载与重写如何实现**

方法调用分为解析调用和分派调用

解析调用：编译期可知、运行期不可变的方法。有静态方法、私有方法、构造器方法、final修饰的方法、父类独有的方法。

分派调用：编译期可知、运行时可变的方法。用于支持重载和重写。

分派调用分为静态分派和动态分派

**静态分派**（实现重载）：所有依赖静态类型来决定方法执行版本的分派动作，都称为静态分派。

**动态分派**（实现重写）：依赖运行时类型来决定方法执行版本的分派动作。过程是先找到调用方法的实例的运行时类型，记为C。如果类型C中能找到对应方法，则进行访问权限校验。如果不能找到则按照继承关系一次找C的各个父类，直到找到为止，如果没有找到则抛出AbstractMethodError异常。

**InnoDB如何实现MVCC的**

通过undo log实现，undo log中通过roll\_pointer和trx\_id形成一条版本链。每对记录修改一次都会在版本链的最前面添加最新的记录。Roll\_pointer指针会指向旧的记录，trx\_id就是当前的事务id，在MVCC中充当版本的作用。

会通过生成Read View找到符合当前版本的数据。依赖四个参数：当前事务id（创建该Read view的事务id）、最大最小事务id、当前数据库中活跃的事务id。通过这四个参数判断某条记录是否可见。

**索引失效**

联合索引没有遵循**最左前缀原则**

**or前后没有同时使用索引**（因为如果有一个没有使用索引，就会走全表扫描，所以干脆全都走全表扫描）

在查询条件中对索引列做了**计算、函数、类型转换**等操作

**执行优化器**认为没有必要使用索引的情况（可以使用force index强制使用索引）：如查询优化器发现某个值出现在表的数据行中的百分比（惯用的百分比界线是"30%"）很高的时候，它一般会忽略索引，进行全表扫描。

数据类型隐式转换：`select \* from user where name=18;`不使用索引，`select \* from user where name='18';`会使用索引

**InnoDB与MyISAM区别**

MyISAM不支持事务，不支持行级锁，不支持外键。

MyISAM的SELECT性能更高，InnoDB增删改效率更高

InnoDB是聚簇索引，MyISAM是非聚簇索引。

MyISAM保存表的行数，InnoDB不保存。

**Update语句执行流程**

客户端与连接器进行通信，然后依次进行词法分析语法分析，通过之后执行update语句的流程。

先寻找到id=1的记录并提交给执行器，执行器调用InnoDB层实现的接口实现更新操作

开启事务，记录相对应的undo log，InnoDB更新Buffer pool，记录对应的binlog

进行两阶段提交：先写redo log并将状态设置为prepare，再将binlog刷盘，然后将redo log的状态设置为commit。

**Spring Bean生命周期**

实例化

依赖注入

初始化：处理Aware接口、调用初始化方法、登记可销毁的Bean。

销毁

**事务传播机制**

Required(默认)：如果没有事务，则加入

Supports：存在事务则加入，否则以非事务执行

MANDATORY：存在则加入，不存在则抛出异常

REQUIRES\_NEW：创建新事务，挂起旧事务

NOT\_SUPPROTED：存在事务则抛出异常

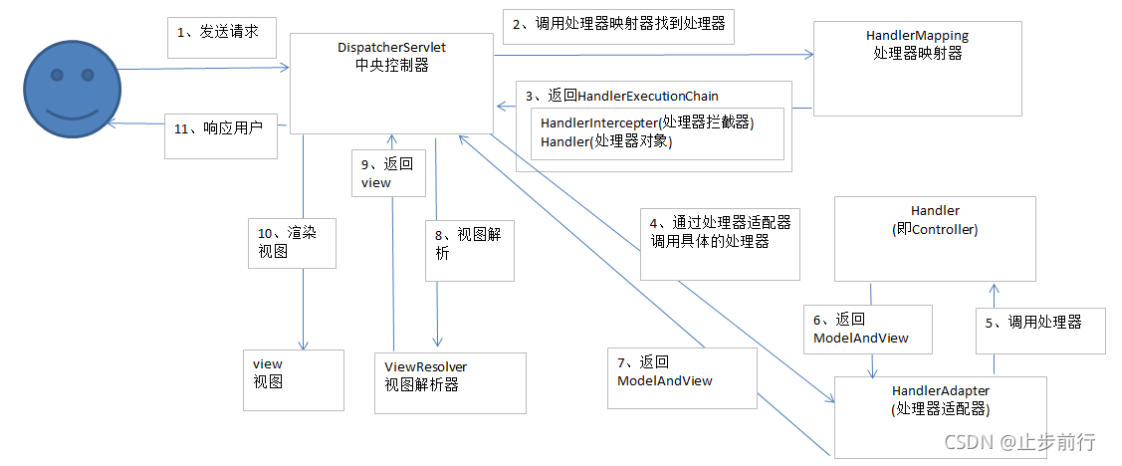
NEVER：不使用事务，如果存在事务则抛出异常

NESTED：没有事务则新建，存在事务则开启一个子事务

**Spring事务失效场景**

异常类型不对、异常没有抛出、本类中的方法调用不会走代理

**SpringMVC执行流程**



**Spring如何解决循环依赖**

利用三级缓存，一级缓存存放成品对象，二级缓存存放半成品对象，三级缓存将半成品封装成工厂。

**CAS详解**

TGC：CAS生成的cookie

TGT是CAS为用户登录签发的票据。用户登录后CAS会在本地存储一个TGT，同时CAS会生成一个cookie（这就是TGC）写入浏览器。浏览器用TGC与TGT是一一对应的。

ST：CAS为用户签发的访问某一特定Service的票据。一个TGT会对应多个ST，主要看用户从哪个应用服务器重定向到CAS服务器。

第一次请求cas-client1：浏览器访问应用服务器，应用服务器发现用户没有携带ST，就引导浏览器重定向到CAS服务器，用户在CAS服务器进行登录。登录成功之后CAS服务器会生成一个TGT，同时向浏览器发送一个TGC（存在cookie里），并在url后加上ST，浏览器拿到CAS服务器生成的cookie与ST就说明登录成功了。

登录成功后拿着ST去cas-client，cas-client拿着ST去cas-server查询有效性，如果有效则返回session给cas-client。至此用户的第一次登录完成。

请求cas-client2：浏览器访问cas-client2，发现没有携带对应的ST，于是重定向到cas-server，发现已经存在TGC了，于是直接生成一个ST给浏览器，浏览器拿着这个ST去cas-client2，然后就是同样的步骤，cas-client2。

拿着这个ST去cas-server进行验证并请求session。

**Redis暴露在外的数据结构**

String：主要有int和SDS实现。可用于缓存对象、计数、分布式锁、共享session信息

List：有quicklist实现。可用于实现消息队列

Hash：hash元素个数小于512个，所有值小于64字节的话，会用ziplist实现（7.0之后被listpack代替）。否则的话会用hash表实现。可用于缓存对象。

Set：元素个数小于512且都是整数，用intset实现，否则使用hash表实现。可用于求集合等操作，如点赞、共同关注等。

Zset：元素个数小于128个且每个元素的值都小于64字节时，会用ziplist。否则使用skiplist。可用于实现排行榜、排序等功能。

**Redis内部数据结构**

SDS、intset、dict、ziplist（新出了listpack来代替ziplist）、quicklist、skiplist、redis object

Ziplist连锁更新问题：prevrawlen会记录前一个entry占用的总字节数，当前一项占用字节数小于254，那么prevrawlen长度为1个字节，如果前一项占用字节数大于254，那么prevrawlen长度为5个字节。此时如果有若干个连续的长度为250~253之间的entry，只要有一个entry的长度变成大于254，那么位于它之后的所有字节长度都会产生变化，那么此时就产生了连锁更新问题。

quicklist是节点为ziplist的双端链表，在3.0之后用来实现list。

**Redis渐进式rehash**：在扩容和收缩的过程中需要将元素进行rehash，然后插入新的table中。具体过程如下：

* 计算扩容或收缩后的realsize
* 按照新的size申请空间，并赋给dict.ht[1]
* 设置rehashidx为0
* 每次执行增删改查的时候，都检查一下rehashidx是否大于-1，如果是代表正在进行rehash，此时当前操作负责将dict.ht[0].table[rehashidx]的所有数据rehash到dict.ht[1]的对应位置上。直至所有数据全部完成rehash
* 将dict.ht[1]的值赋值给dict.ht[0]，并释放相应内存。
* 将rehashidx赋值为-1，代表rehash结束
* 在rehash过程中，新增元素直接写入ht[1]，查询、修改、删除操作会在dict.ht[0]和dict.ht[1]上一次查找并执行。

**Redis内存淘汰**

noeviction：不淘汰任何key，内存满时不允许写入新数据。默认策略

volatile-ttl：对于设置了ttl的key，剩余时间越少越先被淘汰

allkeys-random：对所有key，随机淘汰

volatile-random：对设置了TTL的key，随机淘汰

allkeys-lru：对所有key，采用LRU算法进行淘汰

volatile-lru：对设置了TTL的key，采用LRU算法进行淘汰

allkeys-lfu：对所有key，采用LFU算法进行淘汰

volatile-lfu：对设置了TTL的key，采用LFU算法进行淘汰