# Redis缓存问题

## **a. 缓存穿透**

请求查询数据库不存在的数据，也就是缓存和数据库都不存在，请求每次都打到数据库。

问题：黑客会利用来对系统作攻击。数据库压力过大宕机。

解决方法：

### **缓存空值**

为这些key对应的值设置为null 丢到缓存里面去。后面再出现查询这个key 的请求的时候，直接返回null 。并且设置过期时间。

### **布隆过滤器**

在缓存之前在加一层 BloomFilter ，在查询的时候先去 BloomFilter 去查询 key 是否存在，如果不存在就直接返回，存在再走查缓存 -> 查 DB。

**如何选择？**

针对于一些恶意攻击，攻击带过来的大量key 是不存在的，那么我们采用第一种方案就会缓存大量不存在key的数据。

针对这种key异常多、请求重复率比较低的数据，我们就没有必要进行缓存，使用第二种方案直接过滤掉。

而对于空数据的key有限的，重复率比较高的，我们则可以采用第一种方式进行缓存。

## b. 缓存击穿

高并发的系统中，大量的请求同时查询一个 key 时，此时这个key正好失效了，就会导致大量的请求都打到数据库上面去。

**解决方法：**

### 互斥锁

多个线程同时去查询数据库的这条数据，那么我们可以在第一个查询数据的请求上使用一个互斥锁来锁住它。

其他的线程走到这一步拿不到锁就等着，等第一个线程查询到了数据，然后做缓存。后面的线程进来发现已经有缓存了，就直接走缓存。

## c. 缓存雪崩

当某一时刻发生大规模的缓存失效的情况，比如你的缓存服务宕机了或者同一时间很多redis数据缓存失效了，会有大量的请求进来直接打到DB上面。

解决方法：

### 设置不同的失效时间。

# Redis持久化

## rdb

RDB持久化是把**当前进程数据生成快照保存到硬盘**的过程，触发RDB持久化过程分为手动触发和自动触发。

### 1.触发机制

手动触发分别对应save和bgsave命令

·save命令：**阻塞当前Redis服务器**，直到RDB过程完成为止，对于内存比较大的实例会造成长时间阻塞，线上环境不建议使用

·bgsave命令：Redis进程执行**fork操作创建子进程**，RDB持久化过程由子进程负责，完成后自动结束。**阻塞只发生在fork阶段，一般时间很短。**

2）自动触发RDB的持久

1）使用save相关配置，如“**save m n**”。表示**m秒内数据集存在n次修改** 时，自动触发bgsave。

2）如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点。

3）执行debug reload命令重新加载Redis时，也会自动触发save操作。

4）默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则 自动执行bgsave。

### 2.流程

1）执行bgsave命令，Redis父进程判断当前是否存在正在执行的子进 程，如RDB/AOF子进程，如果存在bgsave命令直接返回。

2）父进程执行**fork操作创建子进程**，fork操作过程中父进程会阻塞，通 过info stats命令查看latest\_fork\_usec选项，可以获取最近一个fork操作的耗时，单位为微秒

3）父进程fork完成后，bgsave命令返回“Background saving started”信息*并不再阻塞父进程*，可以继续响应其他命令。

4）**子进程创建RDB文件**，根据父进程内存生成临时快照文件，完成后 对**原有文件进行原子替换**。执行lastsave命令可以获取最后一次生成RDB的 时间，对应info统计的rdb\_last\_save\_time选项。

5）进程发送信号给父进程表示完成，父进程更新统计信息，具体见 info Persistence下的rdb\_\*相关选项。

### 3.优点

RDB是一个**紧凑压缩的二进制文件**，代表Redis在某个时间点上的数据快照。非常适用于备份，全量复制等场景。Redis加载RDB恢复数据**远远快于AOF**的方式。

### 4.缺点

**没办法做到实时持久化/秒级持久化**。因为bgsave每次运 行都要执行**fork操作创建子进程，属于重量级操作**，频繁执行成本过高。

## aof

以独立日志的方式**记录每次写命令**， 重启时再**重新执行AOF文件中的命令达到恢复数据的目的**。AOF的主要作用是解决了数据持久化的**实时性。**

AOF的工作流程操作：**命令写入 （append）、文件同步（sync）、文件重写（rewrite）、重启加载 （load）。**

### 流程

1）所有的**写入命令会追加到aof\_buf（缓冲区）**中。

2）AOF缓冲区根据**对应的策略向硬盘做同步**操作。

AOF为什么把命令追加到aof\_buf中？Redis使用单线程响应命令，如果每次写AOF文件命令都直接追加**到硬盘，那么**性能完全取决于当前硬盘负载。先写入缓冲区aof\_buf中，还有另一个好处，Redis可以提供多种缓冲区同步硬盘的策略，在性能和安全性方面做出平衡

3）随着AOF文件越来越大，需要定期对**AOF文件进行重写**，达到压缩的目的。

4）当Redis服务器重启时，可以加载AOF文件进行数据恢复。

### 2.触发时机

手动触发：直接调用bgrewriteaof命令。

自动触发：根据auto-aof-rewrite-min-size和auto-aof-rewrite-percentage参数确定自动触发时机

·auto-aof-rewrite-min-size：表示运行**AOF重写时文件最小体积**，默认为64MB。

·auto-aof-rewrite-percentage：代表当前AOF文件空间**和上一次重写后AOF文件空间的比值**。

### 3.同步策略

1:Always

服务器**每写入一个命令**,就调用一次fdatasync,**将缓冲区里面的命令写入到磁盘里面**,在这种模式下,服务器即使遭遇意外停机,也不会丢失任何自己已经成功执行的命令数据

2:Everysec

服务器**每一秒重调用一次fdatasync**,将缓冲区里面的命令写入到磁盘里面,在这种模式写,服务器即使遭遇意外停机时,**最多只丢失一秒钟内执行的命令数据**

3:NO

服务器**不主动调用fdatasync**,由操作系统决定任何将缓冲区里面的命令写入磁盘里面,在这种模式写,服务器遭遇意外停机时,丢失命令的数据是不确定的

4:always的速度慢,everysec和no都很快,默认值:everysec`

# Redis分布式锁

执行lua脚本，保证这段复杂业务逻辑执行的原子性。

## 1.加锁机制



KEYS[1]代表的是你加锁的那个key，比如说：RLock lock = redisson.getLock("myLock");

ARGV[1]代表的就是锁key的默认生存时间，默认30秒。

ARGV[2]代表的是加锁的客户端的ID。

第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。hset myLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

接着会执行“pexpire myLock 30000”命令，**设置**myLock这个锁key的生存时间是30秒。

**流程：**

1、判断有没有一个叫“abc”的key

2、如果没有，则在其下设置一个字段为“6f0829ed-bfd3-4e6f-bba3-6f3d66cd176c:Thread-1”，值为“1”的**键值对** ，并**设置它的过期时间**

3、如果存在，则进一步判断“6f0829ed-bfd3-4e6f-bba3-6f3d66cd176c:Thread-1”是否存在，**若存在，则其值加1，并重新设置过期时间**

4、返回“abc”的生存时间（毫秒）

## 锁互斥机制

如果客户端2来尝试加锁，执行了同样的一段lua脚本，第一个if判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl myLock返回的一个数字，这个数字代表了myLock这个锁key的剩余生存时间。比如还剩15000毫秒的生存时间。

此时客户端2会**订阅Channel**，当资源可用时可以及时知道，并抢占，防止无效的轮询而浪费资源。

当资源可用用的时候，**循环去尝试获取锁**，由于多个线程同时去竞争资源，所以这里用了信号量，对于同一个资源只允许一个线程获得锁，其它的线程阻塞

## watch dog自动延期机制

客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢？只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。开启后台功能会降低redis性能，建议不开启

## 可重入加锁机制

第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。

第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：incrby myLock 8743c9c0-0795-4907-87fd-6c71a6b4586:1 1通过这个命令，对客户端1的加锁次数，累加1。

## 解锁机制

如果执行lock.unlock()，每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del myLock”命令，从redis里删除这个key。

## 缺点

如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。

但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，**客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。**

此时就会导致多个客户端对一个分布式锁完成了加锁。这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是redis cluster，或者是redis master-slave架构的**主从异步复制**导致的redis分布式锁的最大缺陷：在**redis master实例宕机的时候，可能导致多个客户端同时完成加锁。**

## 注意点

1. 要为锁的key设置一个唯一的值，要与线程名有关，这样就**避免了任意线程都能释放锁**，因为如果**业务时间小于锁的过期时间，锁被释放而业务没有执行完，另一线程获得锁**，但会因第一个线程最后的释放锁而受到影响
2. **conn.setnx和con.expire应该用lua脚本**，保证其原子操作，上述代码就明显错误了，如果conn.setnx执行完后，redis服务器宕机了那么会导致锁永远无法释放
3. 需要设置过期时间，避免无法解锁。
4. 加锁成功才是获取锁成功。锁字段为null不能作为锁可用的基准。

# Redis集群方式及原理

## 主从模式

在从节点配置文件加上:slaveof 192.168.33.130 6379

通过master server持久化的rdb文件实现的。master server 先dump出内存快照文件，然后将rdb文件传给slave server，slave server 根据rdb文件重建内存表。

### 1.复制过程

1、slave server启动**连接到master** server之后，salve server主动**发送SYNC命令给master** server

2、master server接受SYNC命令之后，判断，**是否有正在进行内存快照的子进程，如果有，则等待其结束，否则，fork一个子进程，子进程把内存数据保存为文件，并发送给slave server**

3、master server子进程进程做数据快照时，父进程可以**继续接收client端请求写数据**，此时，**父进程把新写入的数据放到待发送缓存队列中**

4、slave server **接收内存快照文件之后，清空内存数据，根据接收的快照文件，重建内存表数据结构**

5、master server把快照文件发送完毕之后，**发送缓存队列中保存的子进程快照期间改变的数据给slave** server，slave server**做相同处理，保存数据一致性**

6、master server **后续接收的数据，都会通过步骤1建立的连接，把数据发送到slave server**

需要注意：slave server如果因为网络或其他原因断与master server的连接，当slave server重新连接时，需要重新获取master server的内存快照文件，slave server的数据会自动全部清空，然后再重新建立内存表，这样会让slave server 启动恢复服务比较慢，同时也给master server带来较大压力，可以看出redis的复制没有增量复制的概念，这是redis主从复制的一个主要弊端，在实际环境中，尽量规避中途增加从库

redis2.8之前不支持增量，到**2.8之后就支持增量了**！

### 2. redis什么时候会发生全量复制？

a) redis slave首启动或者重启后，连接到master时

b) redis slave进程没重启，但是掉线了，重连后不满足部分复制条件

### 3. redis什么时候会发生部分复制？

Run id 可通过info server命令查看

a) 主从的redis版本>=2.8

b) redis slave进程没有重启，但是掉线了，重连了master(因为slave进程重启的话，run id就没有了)

c) redis slave保存的run id与master当前run id一致 (注：run id并不是pid，slave把它保存在内存中，重启就消失)

d) redis slave掉线期间，master保存在内存的offset可用，也就是master变化不大，被更改的指令都保存在内存

### 4.优点

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

为了分载Master的读操作压力，**Slave服务器**可以为客户端提供**只读操作**的服务，**写服务仍然必须由Master来完成**

Slave同样可以接受其它Slaves的连接和同步请求，这样可以有效的分载Master的同步压力。

Master Server是以非阻塞的方式为Slaves提供服务。所以在**Master-Slave同步期间，客户端仍然可以提交查询或修改请求。**

Slave Server同样是以非阻塞的方式完成数据同步。**在同步期间，如果有客户端提交查询请求，Redis则返回同步之前的数据**

### 5.缺点

**不具备自动容错和恢复功能**，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。

**主机宕机，宕机前有部分数据未能及时同步到从机**，切换IP后还会引入数据不一致的问题，降低了系统的可用性。

不易动态扩容。

## 哨兵模式

当主服务器中断服务后，可以将一个从服务器升级为主服务器，以便继续提供服务，但是这个过程需要人工手动来操作。为此，Redis 2.8中提供了哨兵工具来实现自动化的系统监控和故障恢复功能。

哨兵的作用就是监控Redis系统的运行状况。它的功能包括以下两个。

（1）**监控主服务器和从服务器是否正常运行**。

（2）主服务器出现故障时**自动将从服务器转换为主服务器**

SDOWN：全拼Subjectively Down，称为**主观下线**，指的是单个sentinel对redis实例作出的下线状态判断。

ODOWN：全拼Objectively Down，称为**客观下线**，指多个 Sentinel 实例在对同一个redis做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的redis实例下线判断。（一个 Sentinel 可以通过向另一个 Sentinel 发送 SENTINEL is-master-down-by-addr 命令来询问对方是否认为给定的redis实例已下线。）

### 1.实现原理

#### 1. 寻找要监控的主数据库

一个sentinel启动时会读取配置文件，并通过sentinel monitor <master-name> <ip> <port> <quorum>配置**寻找要监控的主数据库**。

#### 2.**与监控的数据库建立两条连接**

sentinel启动后，会**与监控的数据库建立两条连接。**

其中一条连接用来订阅master的\_\_sentinel\_\_:hello频道用于获取其他监控该数据库的sentinel节点信息，另外一条用于哨兵定期向主数据库发送INFO等命令获取主库本身信息，原因在于当客户端进入订阅模式以后只能接受消息，不能发送命令，所以还需要建立一条连接。

#### 3.定时执行操作

与监控的主库建立连接完成后，sentinel定时执行操作：

1.每10s会向主数据库和从数据库发送INFO命令；

2.每1s向master、slave以及其他哨兵节点发送PING命令；

3.每2s向master和slave的\_\_sentiel\_\_:hello频道发送自己的信息来宣布自己的存在，同时该过程也是实现哨兵之间自动发现的基础；

##### 1. INFO命令自动发现从数据库变更信息

首先，sentinel启动后，向主库发送INFO命令使得sentinel可以获取当前主库的相关信息（包括运行的ID，复制信息、以及属于该主库的从库节点信息），这也是为什么在配置监控时候只需要配置监控的主库信息sentinel就自动找到其对应的从库，进而实现从库的监控。而后和每个从库同样建立两个连接，这两个连接和上文介绍的与主库的连个连接完全一致，在此之后，哨兵会每10s定时向已知所有主从发送INFO命令获取信息更新并进行相应操作，比如对新增的从库建立连接并加入监控队列、又或者是主库信息发生变化（由failover引起的）进行信息更新等。

##### 2. 向其他哨兵分享自己的信息

接下来哨兵向master和slave的\_\_sentinel\_\_:hello频道发送信息与同样监控该redis示例的其他哨兵分享自己的信息。发送的消息内容为：

<哨兵地址> ,<哨兵端口>,<哨兵运行的ID>,<哨兵配置的版本>,<主库名称>,<主库地址>,<主库端口>,<主库配置版本>，该消息包含了哨兵基本信息以及监控的主库信息，当其他sentinel收到消息后会判断发消息的哨兵是不是新的哨兵，如果是则将其加入已发现的哨兵列表，并创建一个到其的连接（与数据库不同）哨兵与哨兵之间只会创建一条连接用于发送PING命令，同时sentinel会判断主数据库的配置版本，如果该版本比记录数据库版本高，则更新主数据库的数据，其作用在后续介绍。

##### 3. PING命令定时监控这些主从节点和sentinel节点是否在线

实现了自动发现从数据库和其他sentinel节点后，sentinel后续要做的任务是定时监控这些已经发现的主从节点和sentinel节点是否在线。这种监控实现方式是在通过一定时间间隔发送PING命令实现，时间间隔配置通过down-after-milliseconds指定，当超过down-after-milliseconds配置的时间后，如果被PING的数据库或者sentinel未回复，则哨兵认为其主观下线（主观下线在上面已经介绍了），如果该节点是主库sentinel会进一步进行判断是否需要对其进行故障恢复(failover)：sentinel会发送SENTINEL is-master-down-by-addr命令询问其他sentinel节点是否也认为该主库主观下线，如果达到指定数量（在示例配置中也进行了说明，示例配置的是2）时，哨兵会认为其客观下线，并选取领头的哨兵（leader）进行故障恢复，选举过程后续介绍。

选举完零头哨兵后，领头哨兵会开始对主数据库进行故障恢复，这一过程称为failover，在选取新的master时候，sentinel会考虑以下情况：

跟master断开连接的时长

slave的优先级 （由slave-priority配置指定）

复制偏移量offset

实例运行的id（run id）

#### 4.Sentinel状态持久化

Sentinel的状态会被持久化地写入sentinel的配置文件中。每次当收到一个新的配置时，或者新创建一个配置时，配置会被持久化到硬盘中，并带上配置的版本戳。这意味着，可以安全的停止和重启sentinel进程。

## Cluster集群模式

redis的哨兵模式基本已经可以实现高可用，读写分离 ，但是在这种模式下每台redis服务器都存储相同的数据，很浪费内存，所以在redis3.0上加入了cluster模式，实现的redis的分布式存储，也就是说每台redis节点上存储不同的内容。

集群中至少应该有奇数个节点，所以至少有三个节点，每个节点至少有一个备份节点。所以最少6个节点。

Redis-Cluster采用无中心结构。

### 特点

##### **节点彼此互联**

所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽。

##### **fail超过半数的节点检测失效时才生效** 节点的fail是通过集群中超过半数的节点检测失效时才生效。

##### **客户端与redis节点直连,连接集群中任何一个可用节点即可** 客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可。

##### 每个节点保存数据和整个集群状态。物理节点映射到slot，cluster负责维护node<->slot<->value。 redis-cluster把所有的物理节点映射到[0-16383]slot上（不一定是平均分配）,cluster 负责维护node<->slot<->value。

##### CRC16算法计算key的值，对16384求模。决定将一个key放到哪个桶 Redis集群预分好16384个桶，当需要在 Redis 集群中放置一个 key-value 时，根据 CRC16(key) mod 16384的值，决定将一个key放到哪个桶中。

### 2.节点分配

新增一个节点D，redis cluster的这种做法是**从各个节点的前面各拿取一部分slot到D上**。大致就会变成这样：

节点A覆盖1365-5460

节点B覆盖6827-10922

节点C覆盖12288-16383

节点D覆盖0-1364,5461-6826,10923-12287

同样删除一个节点也是类似，移动完成后就可以删除这个节点了。

### 3.主从切换

一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点拉取数据备份，当这个主节点挂掉后，就会有这个从节点选取一个来充当主节点，从而保证集群不会挂掉。

上面那个例子里, 集群有ABC三个主节点, 如果这3个节点都没有加入从节点，如果B挂掉了，我们就无法访问整个集群了。A和C的slot也无法访问。

所以我们在集群建立的时候，一定要为每个主节点都添加了从节点。

# Redis内存淘汰机制

# Redis数据类型

## 1.类型

### 1.String（字符串）

String类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

String类型是Redis最基本的数据类型，一个键最大能存储512MB。

### 2.Hash（哈希）

hash 是一个键值对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

### 3.List（列表）

列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。

列表最多可存储 232 - 1元素 (4294967295, 每个列表可存储40多亿)。

list 是链表结构，所有如果在头部和尾部插入数据，性能会非常高，不受链表长度的影响；但如果在链表中插入数据，性能就会越来越差。

### 4.Set（集合）

Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

sadd 命令:添加一个string元素到,key对应的set集合中，成功返回1,如果元素以及在集合中返回0,key对应的set不存在返回错误。

set 类型提供了多个 set 之间的聚合运算，如求交集、并集、补集，这些操作在 redis 内部完成，效率很高。

### 5.zset(sorted set：有序集合)

zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd 命令：添加元素到集合，元素在集合中存在则更新对应score

命令：zadd key score member

zrangebyscore key score score获取score区间的元素

zcard key 获取key中元素个数

zincrby key increment member将指定key对应的集合中的member元素对应的分值递增加 increment

zrangebyscore key start end [withscores] 同 zrange命令不同的是，zrange命令是索引在startend范围的查询，而zrangebyscore命令是根据分值在startend之间的查询且升序展示

zrank key member返回指定key对应的有序集合中，指定元素member在集合中排名，从0开始切分值是从小到大升序

zcount key min max返回指定key对应的有序集合中，分值在min~max之间的元素个数

## 2.Redis数据类型应用

### 1.排行榜

Zset数据类型

### 2.微信关注

Set

### 3.布隆过滤器、用户日活，月活，留存率的统计

位图

String类型的底层是二进制，可以设置他的bit位。

本质上位只有0和1的区别，所以用位做业务数据记录，就不需要在意value的值。

setbit key offset value

getbit key offset

bitcount key [start] [end] 计算给定key的字符串值中，被设置为1的位bit的数量

对一个或多个保存二进制的字符串key进行元操作，并将结果保存到destkey上。

语法：operation可以是and、or、not、xor的一种。

bitop and destkey key [key...]，对一个或多个key逻辑并，结果保存到destkey。

bitop or destkey key [key...]，对一个或多个key逻辑或，结果保存到destkey。

bitop xor destkey key [key...]，对一个或多个key逻辑异或，结果保存到destkey。

bitop xor destkey key，对一个或多个key逻辑非，结果保存到destkey。

# Synchronized修饰

1. synchronized修饰非静态同步方法时，锁住的是当前实例
2. synchronized修饰静态同步方法时，锁住的是该类的Class对象
3. synchronized修饰静态代码块时，锁住的是synchronized关键字后面括号内的对象。

# 类代码加载顺序

**1.首先加载父类的静态字段或者静态语句块**

**2.子类的静态字段或静态语句块**

**3.父类普通变量以及语句块**

**4.父类构造方法被加载**

**5.子类变量或者语句块被加载**

**6.子类构造方法被加载**

**其中**

**静态代码块（只加载一次）**

**构造方法（创建一个实例就加载一次）**

**静态方法，调用的时候才会加载，不调用的时候不会加载**

**静态语句块和静态变量被初始化的顺序与代码先后顺序有关**

# 微服务解决幂等性问题由于网络延迟避免重复提交的问题

Java后端生成一个token令牌UUID加时间戳，在java返回页面的时候将token存入到redis并返回给页面的隐藏域中，页面校验数据成功且点击提交后台对页面的token和redis的token进行比对成功则清除redis中的token，如果用户在次提交那么redis找不到对应的token就提醒用户此数据已经提交。

# 数据库主从复制

## 2.实现原理

master开启bin-log功能，日志文件用于记录数据库的读写增删。

需要开启3个线程，master IO线程，slave开启 IO线程 SQL线程，

1.Slave 通过IO线程连接master，并且请求某个bin-log，position之后的内容。

2.主服务器将数据更新记录到二进制日志中。

3.从服务器会开启两个线程，即I/O线程和SQL线程。

4.从服务器将主服务器的二进制日志（Binary log）复制到自己的中继日志（Relay log）中，这个是由从服务器的I/O线程来负责的。

5.从服务器实时监控relay-log日志内容是否有更新，将其更新应用到自己的数据库上，这个是由从服务器的SQL线程来负责的。

# MySQL索引的类型有哪些

**FULLTEXT索引：**mysql的引擎为MyISAM的时候才能创建，作用在char,varchar,text上面，为了解决like ‘%name%’搜索慢的问题。利用【分词技术】等多种算法智能分析出文本文字中关键词的频率和重要性，然后按照一定的算法规则智能地筛选出我们想要的搜索结果

**HASH索引：**HASH索引，只有精确匹配索引所有列的查询才会生效，只有Memory引擎显示支持哈希索引，也是默认索引类型。以键值对的形式存在，可以一次定位效率很高，但是仅用于=和in的时候才会生效，如果用在排序，范围组合以及组合索引的情况下效率会大幅度降低

**BTREE索引：**通过一定的算法将索引值存入到二叉树中，每次查询都是从入口root开始，依次遍历node，最终获得leaf（值）,mysql中最常用的索引类型。

**RTREE索引：**只支持geometry集合数据类型，不常用

# [普通索引和唯一索引的区别](https://www.cnblogs.com/wangchunli-blogs/p/10416046.html)

## 对于查询

假设，执行查询的语句是 select id from T where k=5。这个查询语句在索引树上查找的过程，先是通过B+树从树根开始，按层搜索到叶子节点，也就是图中右下角的这个数据页，然后可以认为数据页内部通过二分法来定位记录。

**对于普通索引来说，查找到满足条件的第一个记录(5,500)后，需要查找下一个记录，直到碰到第一个不满足k=5条件的记录。**

**对于唯一索引来说，由于索引定义了唯一性，查找到**第一个满足条件**的记录后，就会**停止继续检索**。**

InnoDB的数据是按数据页为单位来读写的。也就是说，当需要读一条记录的时候，并不是将这个记录本身从磁盘读出来，而是以页为单位，将其整体读入内存。在InnoDB中，每个数据页的大小默认是16KB。

因为引擎是按页读写的，所以说，当找到k=5的记录的时候，它所在的数据页就都在内存里了。那么，对于普通索引来说，要多做的那一次“查找和判断下一条记录”的操作，就只需要一次指针寻找和一次计算。

当然，如果k=5这个记录刚好是这个数据页的最后一个记录，那么要取下一个记录，必须读取下一个数据页，这个操作会稍微复杂一些。

## 对于更新

### 1.change buffer介绍

当需要更新一个数据页时，如果数据页在内存中就直接更新，而如果这个数据页还没有在内存中的话，在不影响数据一致性的前提下，InooDB会将这些更新操作缓存在change buffer中，这样就不需要从磁盘中读入这个数据页了。在下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行change buffer中与这个页有关的操作。通过这种方式就能保证这个数据逻辑的正确性。

虽然名字叫作change buffer，实际上它是可以持久化的数据。也就是说，change buffer在内存中有拷贝，也会被写入到磁盘上。

将change buffer中的操作应用到原数据页，得到最新结果的过程称为merge。除了访问这个数据页会触发merge外，系统有后台线程会定期merge。在数据库正常关闭（shutdown）的过程中，也会执行merge操作。

显然，如果能够将更新操作先记录在change buffer，减少读磁盘，语句的执行速度会得到明显的提升。而且，数据读入内存是需要占用buffer pool的，所以这种方式还能够避免占用内存，提高内存利用率。

那么，什么条件下可以使用change buffer呢？

对于唯一索引来说，所有的更新操作都要先判断这个操作是否违反唯一性约束。比如，要插入(4,400)这个记录，就要先判断现在表中是否已经存在k=4的记录，而这必须要将数据页读入内存才能判断。如果都已经读入到内存了，那直接更新内存会更快，就没必要使用change buffer了。

因此，唯一索引的更新就不能使用change buffer，实际上也只有普通索引可以使用。

change buffer用的是buffer pool里的内存，因此不能无限增大。change buffer的大小，可以通过参数innodb\_change\_buffer\_max\_size来动态设置。这个参数设置为50的时候，表示change buffer的大小最多只能占用buffer pool的50%。

### 2.区别

#### 1. 这个记录要更新的目标页在内存中

第一种情况是，这个记录要更新的目标页在内存中。这时，InnoDB的处理流程如下：

对于唯一索引来说，找到3和5之间的位置，判断到没有冲突，插入这个值，语句执行结束；

对于普通索引来说，找到3和5之间的位置，插入这个值，语句执行结束。

这样看来，普通索引和唯一索引对更新语句性能影响的差别，只是一个判断，只会耗费微小的CPU时间。

但，这不是我们关注的重点。

#### 2. 这个记录要更新的目标页不在内存中。

第二种情况是，这个记录要更新的目标页不在内存中。这时，InnoDB的处理流程如下：

对于唯一索引来说，*需要将数据页读入内存*，判断到没有冲突，插入这个值，语句执行结束；

对于普通索引来说，则是将更新记录在change buffer，语句执行就结束了。

将数据从磁盘读入内存涉及随机IO的访问，是数据库里面成本最高的操作之一。change buffer因为减少了随机磁盘访问，所以对更新性能的提升是会很明显的

## 3.选择

这两类索引在查询能力上是没差别的，主要考虑的是对更新性能的影响。所以，我建议你尽量选择普通索引。

# StringBuilder和StringBuffer的区别

StringBuilder可变的，线程不安全

StringBuffer可变的，线程安全

# final,finally,finalize的区别

## 1.final

1、修饰符（关键字） 如果一个类被声明为final,意味着它不能再派生新的子类，不能作为父类被继承。因此一个类不能及被声明为abstract，又被声明为final的。

2、将变量或方法声明为final,可以保证他们使用中不被改变。被声明为final的变量必须在声明时给定初值，而以后的引用中只能读取，不可修改，被声明为final的方法也同样只能使用，不能重写。

## 2.finally

不管有没有异常被抛出、捕获，finally块都会被执行。try块中的内容是在无异常时执行到结束。catch块中的内容，是在try块内容发生catch所声明的异常时，跳转到catch块中执行。

## 3. **finalize**

java技术允许使用finalize方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在object类中定义的，因此所有的类都继承了它。子类覆盖finalize方法以执行其他清理工作。

# 接口和抽象类的区别

|  |  |  |
| --- | --- | --- |
|  | **抽象类** | **接口** |
| 默认的方法实现 | 可以有默认的方法实现 | jdk1.8之前不能有默认方法实现，jdk1.8开始支持用default进行默认方法的实现，这个默认实现方法并不强迫实现类去实现 |
| 实现 | 子类使用extends关键字来继承抽象类，如果子类不是抽象类的话，那么子类要实现抽象类中的所有声明的方法 | 子类通过implements来实现接口，子类需要实现接口中所有声明的方法 |
| 构造器 | 可以有 | 不可以有 |
| 多继承 | 可以继承一个类和实现多个接口 | 只可以“继承”一个或多个其他接口 |
| 默认修饰符 | 参考下边默认访问权限 | 方法：public abstract ； 属性：public static final |
| 默认访问权限 | JDK 1.8以前，抽象类的方法默认访问权限为protected（可以是public和protected），JDK 1.8时，抽象类的方法默认访问权限变为default（可以是public和protected或者不写） | JDK 1.8以前，接口中的方法默认，也必须是public的（只能用public），JDK 1.8时，接口中的方法默认public的，也可以是default的（可以是public和default），JDK 1.9时，接口中的方法可以是private的（可以是public和default和private） |

# 垃圾回收机制

释放垃圾占用的空间，防止内存泄露。有效的使用可以使用的内存，对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收。

## 1.垃圾判断算法

### 1.引用计数法

给每个对象添加一个计数器，当有地方引用该对象时计数器加1，当引用失效时计数器减1。用对象计数器是否为0来判断对象是否可被回收。缺点：**无法解决循环引用的问题**。

ReferenceCountingGC a = new ReferenceCountingGC("objA");

ReferenceCountingGC b = new ReferenceCountingGC("objB");

a.instance = b;

b.instance = a;

a = null;

b = null;

### 2.可达性分析算法

通过GC ROOT的对象作为搜索起始点，通过引用向下搜索，所走过的路径称为引用链。通过对象是否有到达引用链的路径来判断对象是否可被回收

可作为GC ROOT的对象：

1.虚拟机栈中引用的对象，

2.方法区中类静态属性引用的对象，

3.方法区中常量引用的对象，

4.本地方法栈中JNI引用的对象

## 2.垃圾回收算法

### 1.标记-清除

先把内存区域中的这些对象进行标记，哪些属于可回收标记出来，然后把这些垃圾拎出来清理掉。

缺点：产生内存碎片

### 2.复制

将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。保证了内存的连续可用，内存分配时也就不用考虑内存碎片等复杂情况。

缺点：只能利用一半内存区域

### 3.标记-整理

过程仍然与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，再清理掉端边界以外的内存区域。

标记整理算法解决了内存碎片的问题，也规避了复制算法只能利用一半内存区域的弊端。标记整理算法对内存变动更频繁，需要整理所有存活对象的引用地址，在效率上比复制算法要差很多。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

缺点：对内存变动更频繁，需要整理所有存活对象的引用地址

### 4.分代收集算法

严格来说并不是一种思想或理论，而是融合上述3种基础的算法思想，而产生的针对不同情况所采用不同算法的一套组合拳，根据对象存活周期的不同将内存划分为几块。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

在老年代中，因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记-清理算法或者标记-整理算法来进行回收。

## 3.堆和方法区的垃圾回收

java中的垃圾回收大致在两部分，第一个就是堆、第二个就是方法区。

### 方法区的垃圾回收

方法区的垃圾回收主要有两部分：废弃常量和无用的类。略。

《java虚拟机规范》里面曾经说到过，不要求虚拟机对方法区进行垃圾回收。而且方法区进行垃圾回收性价比比较低。

### **堆的垃圾回收**

#### 新生代－复制 回收机制

新生代特点是：大量新对象死去，只有少量存活

新生代划分了三个部分比例：Eden：S1：S2=8:1:1。

1.首先，Eden对外提供堆内存。当 Eden区快要满了，触发垃圾回收机制，把存活对象放入 Survivor A 区，清空 Eden 区；

2.Eden区被清空后，继续对外提供堆内存；

3.当 Eden 区再次被填满，对 Eden区和 Survivor A 区同时进行垃圾回收，把存活对象放入 Survivor B区，同时清空 Eden 区和Survivor A 区；

4.当某个 Survivor区被填满，把多余对象放到Old 区；

5.当 Old区也被填满时，进行 下一阶段的垃圾回收。

#### 2.老年代－标记整理 回收机制

老年代的特点是：**存活对象多、垃圾少**。因此，根据老年代的特点，这里仅仅通过少量地移动对象就能清理垃圾，而且不存在内存碎片化。也就是标记整理的回收机制。既然是标记整理算法，而且老年代内部也不存在着内存划分，所以只需要根据标记整理的具体步骤进行垃圾回收就好了。

## 4.内存区域与回收策略

对象的内存分配，往大方向讲，就是在堆上分配（但也可能经过JIT编译后被拆散为标量类型并间接地栈上分配），对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中（大对象直接分到老年代），分配的规则并不是百分百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

对象的内存分配，往大方向讲，就是在堆上分配（但也可能经过JIT编译后被拆散为标量类型并间接地栈上分配），对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中（大对象直接分到老年代），分配的规则并不是百分百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

### 1.对象优先在Eden分配

大多数情况下，对象会在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机会发起一次 Minor GC。Minor GC相比Major GC更频繁，回收速度也更快。通过Minor GC之后，Eden区中绝大部分对象会被回收，而那些存活对象，将会送到Survivor的From区（若From区空间不够，则直接进入Old区） 。

4.2 Survivor区

Survivor区相当于是Eden区和Old区的一个缓冲，类似于我们交通灯中的黄灯。Survivor又分为2个区，一个是From区，一个是To区。每次执行Minor GC，会将Eden区中存活的对象放到Survivor的From区，而在From区中，仍存活的对象会根据他们的年龄值来决定去向。（From Survivor和To Survivor的逻辑关系会发生颠倒： From变To ， To变From，目的是保证有连续的空间存放对方，避免碎片化的发生）

### 2.Survivor区存在的意义

如果没有Survivor区，Eden区每进行一次Minor GC，存活的对象就会被送到老年代，老年代很快就会被填满。而有很多对象虽然一次Minor GC没有消灭，但其实也并不会蹦跶多久，或许第二次，第三次就需要被清除。这时候移入老年区，很明显不是一个明智的决定。所以，Survivor的存在意义就是减少被送到老年代的对象，进而减少Major GC的发生。Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

### 3.大对象直接进入老年代

所谓大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组。大对象对虚拟机的内存分配来说就是一个坏消息，经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来 “安置” 它们。

虚拟机提供了一个XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配，这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制（新生代采用的是复制算法）。

### 4.长期存活的对象将进入老年代

虚拟机给每个对象定义了一个对象年龄（Age）计数器，如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中（正常情况下对象会不断的在Survivor的From与To区之间移动），并且对象年龄设为1。对象在Survivor区中每经历一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认15岁），就将会晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 XX:MaxPretenuringThreshold 设置。

### 5.动态对象年龄判定

为了能更好地适应不同程度的内存状况，虚拟机并不是永远地要求对象的年龄必须达到 MaxPretenuringThreshold才能晋升老年代，如果Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需等到MaxPretenuringThreshold中要求的年龄。

## 5.术语

### 1. 吞吐量

CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值。比如说虚拟机总运行了 100 分钟，用户代码时间 99 分钟，垃圾回收 时间 1 分钟，那么吞吐量就是 99%。

### 2.停顿时间

停顿时间 指垃圾回收器正在运行时，应用程序 的 暂停时间。

### 3.GC的名词

新生代GC：Minor GC

老年代GC：Major GC

### 4.并发与并行

（1）串行（Parallel）

垃圾回收线程 进行垃圾回收工作，但此时 用户线程 仍然处于 等待状态。

（2）并发（Concurrent）

这里的并发指 用户线程 与 垃圾回收线程 交替执行。

（3）并行（Parallel）

这里的并行指 用户线程 和多条 垃圾回收线程 分别在不同 CPU 上同时工作。

## **6.垃圾回收器**

### 1.cms

是一种以获取最短回收停顿时间为目标的收集器。这是因为CMS收集器工作时，GC工作线程与用户线程可以并发执行，以此来达到降低收集停顿时间的目的。

仅作用于老年代的收集，是基于标记-清除算法的，它的运作过程分为4个步骤：

1.初始标记（CMS initial mark）

2.并发标记（CMS concurrent mark）

3.重新标记（CMS remark）

4.并发清除（CMS concurrent sweep）

初始标记、重新标记这两个步骤仍然需要Stop-the-world。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，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是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，这个阶段的停顿时间一般会比初始阶段稍长一些，但远比并发标记的时间短。

CMS以流水线方式拆分了收集周期，将耗时长的操作单元保持与应用线程并发执行。只将那些必需STW才能执行的操作单元单独拎出来，控制这些单元在恰当的时机运行，并能保证仅需短暂的时间就可以完成。这样，在整个收集周期内，只有两次短暂的暂停（初始标记和重新标记），达到了近似并发的目的。

CMS收集器优点：并发收集、低停顿。

VM在暂停的时候，需要选准一个时机。由于JVM系统运行期间的复杂性，不可能做到随时暂停，因此引入了安全点的概念。

安全点(Safepoint)

安全点，即程序执行时并非在所有地方都能停顿下来开始GC，只有在到达安全点时才能暂停。Safepoint的选定既不能太少以至于让GC等待时间太长，也不能过于频繁以致于过分增大运行时的负荷。

### 2.G1

重新定义了堆空间，打破了原有的分代模型，将堆划分为一个个区域。这么做的目的是在进行收集时不必在全堆范围内进行，这是它最显著的特点。区域划分的好处就是带来了停顿时间可预测的收集模型：用户可以指定收集操作在多长时间内完成。

| **特征** | **G1** | **CMS** |
| --- | --- | --- |
| 并发和分代 | 是 | 是 |
| 最大化释放堆内存 | 是 | 否 |
| 低延时 | 是 | 是 |
| 吞吐量 | 高 | 低 |
| 压实 | 是 | 否 |
| 可预测性 | 强 | 弱 |
| 新生代和老年代的物理隔离 | 否 | 是 |

#### 特点

##### 1.并行与并发

G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-the-world停顿的时间，部分其他收集器原来需要停顿Java线程执行的GC操作，G1收集器仍然可以通过并发的方式让Java程序继续运行。

##### 2.空间整合

与CMS的标记-清除算法不同，G1从整体来看是基于标记-整理算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的。但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。

##### 3.可预测的停顿

这是G1相对于CMS的一个优势，降低停顿时间是G1和CMS共同的关注点。

G1收集器将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合，每个分区也不会确定地为某个代服务，可以按需在年轻代和老年代之间切换。Region的大小是一致的，数值是在1M到32M字节之间的一个2的幂值数，JVM会尽量划分2048个左右、同等大小的Region，这一点可以参看如下[源码](http://hg.openjdk.java.net/jdk/jdk/file/fa2f93f99dbc/src/hotspot/share/gc/g1/heapRegionBounds.hpp)。其实这个数字既可以手动调整，G1也会根据堆大小自动进行调整。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1会通过一个合理的计算模型，计算出每个Region的收集成本并量化，这样一来，收集器在给定了“停顿”时间限制的情况下，总是能选择一组恰当的Regions作为收集目标，让其收集开销满足这个限制条件，以此达到实时收集的目的。

运作过程大致如下：

初始标记（Initial Marking）：仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这阶段需要停顿线程，但耗时很短。

并发标记（Concurrent Marking）：是从GC Roots开始堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。

最终标记（Final Marking）：是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标记记录，虚拟机将这段时间对象变化记录在线程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（Live Data Counting and Evacuation）：首先对各个Region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。这个阶段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提高收集效率。

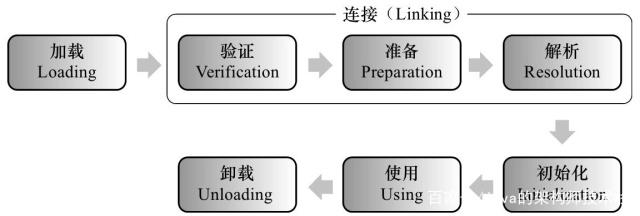
# 类加载机制

## 1.含义

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

## 2.类加载的过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。



### **加载**

（1）通过一个类的全限定名来获取其定义的二进制字节流

（2）将这个字节流所代表的的静态存储结构转化为方法区的运行时数据结构

（3）在堆中生成一个代表这个类的Class对象，作为方法区中这些数据的访问入口。

### 2. 验证

主要作用就是确保被加载的类的正确性。

（1）文件格式的验证：验证.class文件字节流是否符合class文件的格式的规范，并且能够被当前版本的虚拟机处理。这里面主要对魔数、主版本号、常量池等等的校验（魔数、主版本号都是.class文件里面包含的数据信息、在这里可以不用理解）。

（2）元数据验证：主要是对字节码描述的信息进行语义分析，以保证其描述的信息符合java语言规范的要求，比如说验证这个类是不是有父类，类中的字段方法是不是和父类冲突等等。

（3）字节码验证：这是整个验证过程最复杂的阶段，主要是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在元数据验证阶段对数据类型做出验证后，这个阶段主要对类的方法做出分析，保证类的方法在运行时不会做出危害虚拟机安全的事。

（4）符号引用验证：它是验证的最后一个阶段，发生在虚拟机将符号引用转化为直接引用的时候。主要是对类自身以外的信息进行校验。目的是确保解析动作能够完成。

对整个类加载机制而言，验证阶段是一个很重要但是非必需的阶段，如果我们的代码能够确保没有问题，那么我们就没有必要去验证，毕竟验证需要花费一定的的时间。当然我们可以使用-Xverfity:none来关闭大部分的验证。

### 3.准备

为类变量分配内存，并将其初始化为默认值。（此时为默认值，在初始化的时候才会给变量赋值）即在方法区中分配这些变量所使用的内存空间。

（1）类变量（static）会分配内存，但是实例变量不会，实例变量主要随着对象的实例化一块分配到java堆中，

（2）这里的初始值指的是数据类型默认值，而不是代码中被显示赋予的值。比如

public static int value = 1; //在这里准备阶段过后的value值为0，而不是1。赋值为1的动作在初始化阶段。

注意，在上面value是被static所修饰的准备阶段之后是0，但是如果同时被final和static修饰准备阶段之后就是1了。可以理解为static final在编译器就将结果放入调用它的类的常量池中了。

（3）如果类字段的字段属性表中存在ConstantValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。

### 4.解析

解析阶段主要是虚拟机将常量池中的符号引用转化为直接引用的过程。

符号引用：以一组符号来描述所引用的目标，可以是任何形式的字面量，只要是能无歧义的定位到目标就好，就好比在班级中，老师可以用张三来代表你，也可以用你的学号来代表你，但无论任何方式这些都只是一个代号（符号），这个代号指向你（符号引用）

直接引用：直接引用是可以指向目标的指针、相对偏移量或者是一个能直接或间接定位到目标的句柄。和虚拟机实现的内存有关，不同的虚拟机直接引用一般不同。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

### 5.初始化

在初始化阶段，主要为类的静态变量赋予正确的初始值，JVM负责对类进行初始化，主要对类变量进行初始化。在Java中对类变量进行初始值设定有两种方式：

1.声明类变量时指定初始值

2.使用静态代码块为类变量指定初始值

#### 初始化步骤

1、假如这个类还没有被加载和连接，则程序先加载并连接该类

2、假如该类的直接父类还没有被初始化，则先初始化其直接父类

3、假如类中有初始化语句，则系统依次执行这些初始化语句

#### 类初始化时机

只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

1.创建类的实例，也就是new的方式

2.访问某个类或接口的静态变量，或者对该静态变量赋值

3.调用类的静态方法

4.反射（如Class.forName(“com.shengsiyuan.Test”)）

5.初始化某个类的子类，则其父类也会被初始化

6.Java虚拟机启动时被标明为启动类的类（Java Test），直接使用java.exe命令来运行某个主类

## 3.类加载器

站在Java虚拟机的角度来讲，只存在两种不同的类加载器：启动类加载器：它使用C++实现（这里仅限于Hotspot，也就是JDK1.5之后默认的虚拟机，有很多其他的虚拟机是用Java语言实现的），是虚拟机自身的一部分；所有其他的类加载器：这些类加载器都由Java语言实现，独立于虚拟机之外，并且全部继承自抽象类java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。

### 1.种类

1.启动类加载器Bootstrap ClassLoader ：最顶层的加载类，使用C++实现，负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的java.\*开头的类均被Bootstrap ClassLoader加载）。启动类加载器是无法被Java程序直接引用的。

2.扩展类加载器Extention ClassLoader ：扩展的类加载器，加载目录JDK\jre\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。

3.应用程序类加载器Application ClassLoader：也称为SystemAppClass。 该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，我们还可以加入自定义的类加载器。因为JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件。

### 特点

全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

### 类的加载方式

（1）通过命令行启动应用时由JVM初始化加载含有main()方法的主类。

（2）通过Class.forName()方法动态加载，会默认执行初始化块（static{}），但是Class.forName(name,initialize,loader)中的initialze可指定是否要执行初始化块。

（3）通过ClassLoader.loadClass()方法动态加载，不会执行初始化块。

### 双亲委派原则

双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

例子：

1、当AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。

2、当ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。

3、如果BootStrapClassLoader加载失败（例如在$JAVA\_HOME/jre/lib里未查找到该class），会使用ExtClassLoader来尝试加载；

4、若ExtClassLoader也加载失败，则会使用AppClassLoader来加载，如果AppClassLoader也加载失败，则会报出异常ClassNotFoundException。

#### 好处

1.可以避免重复加载，父类已经加载了，子类就不需要再次加载

2.更加安全，很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心api，会带来相关隐患。

### 5.自定义类加载器

主要有两种方式

（1）遵守双亲委派模型：继承ClassLoader，重写findClass()方法。

（2）破坏双亲委派模型：继承ClassLoader,重写loadClass()方法。 通常我们推荐采用第一种方法自定义类加载器，最大程度上的遵守双亲委派模型。

# Java运行时内存模型

共5个。

方法区，虚拟机栈，本地方法栈，堆，程序计数器，直接内存

## 堆

所有线程共享的区域。堆里面存放的都是对象的实例（new 出来的对象都存在堆中），几乎所有的对象实例都在这里分配内存。

堆内存分为两个部分：年轻代和老年代。我们平常所说的垃圾回收，主要回收的就是堆区。更细一点划分新生代又可划分为Eden区和2个Survivor区（From Survivor和To Survivor）。

注意永久代并不属于堆内存中的一部分，同时jdk1.8之后永久代已经被移除。

新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2 ( 该值可以通过参数–XX:NewRatio 来指定 )

默认的，Eden : from : to = 8 : 1 : 1 ( 可以通过参数 –XX:SurvivorRatio 来设定 )

## 方法区

方法区也称”永久代“，它用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。

在JDK8之前的HotSpot JVM，存放这些”永久的”的区域叫做“永久代(permanent generation)”。永久代是一片连续的堆空间，在JVM启动之前通过在命令行设置参数-XX:MaxPermSize来设定永久代最大可分配的内存空间，默认大小是64M（64位JVM默认是85M）。

随着JDK8的到来，JVM不再有永久代(PermGen)。但类的元数据信息（metadata）还在，只不过不再是存储在连续的堆空间上，而是移动到叫做“Metaspace”的本地内存JMM内存模型。

## 虚拟机栈

线程私有，生命周期与线程相同。创建线程的时候就会创建一个java虚拟机栈。

虚拟机执行java程序的时候，每个方法都会创建一个栈帧，栈帧存放在java虚拟机栈中，通过压栈出栈的方式进行方法调用。

栈帧又分为一下几个区域：局部变量表、操作数栈、动态连接、方法出口等。

平时我们所说的变量存在栈中，这句话说的不太严谨，应该说局部变量存放在java虚拟机栈的局部变量表中。

java的8种基本类型的局部变量的值存放在虚拟机栈的局部变量表中，如果是引用型的变量，则只存储对象的引用地址。

### **1.局部变量表**

局部变量表是一组变量值存贮空间，用于存放方法参数和方法内定义的局部变量。

容量以变量槽为最小单位。每个变量槽都可以存储32位长度的内存空间，例如boolean、byte、char、short、int、float、reference。

对于64位长度的数据类型（long，double），虚拟机会以高位对齐方式为其分配两个连续的Slot空间，也就是相当于把一次long和double数据类型读写分割成为两次32位读写。

操作数栈(operand Stack)也常称为操作栈，它是一个后入先出栈。和局部变量表一样，操作数栈的最大深度也在编译的时候写入到Code属性的max\_stacks中。

### **2.操作数栈**

操作数栈的每一个元素可用是任意的Java数据类型，包括long和double。32位数据类型所占的栈容量为1，64位数据类型占用的栈容量为2。

当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法执行的过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈 / 入栈操作。例如，在做算术运算的时候是通过操作数栈来进行的，又或者在调用其它方法的时候是通过操作数栈来进行参数传递的。

操作栈调用其它有返回结果的方法时，会把结果 push 到栈上（通过操作数栈来进行参数传递）。

### ****3.动态链接****

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。

Class文件的常量池中存在大量符号引用，在类加载阶段中的解析阶段会将符号引用转为直接引用，这种转化也称为静态解析。另外的一部分将在运行时转化为直接引用，这部分称为动态链接。

### 方法返回地址

当一个方法开始执行后，只有2种方式可以退出这个方法：

方法返回指令：执行引擎遇到一个方法返回的字节码指令，这时候有可能会有返回值传递给上层的方法调用者，这种退出方式称为正常完成出口。

异常退出：在方法执行过程中遇到了异常，并且没有处理这个异常，就会导致方法退出。

无论采用任何退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信。用来帮助恢复它的上层方法的执行状态。

方法退出的过程实际上就等于把当前栈帧出栈，因此退出可能执行的操作有：

1.恢复上层方法的局部变量表和操作数栈

2.把返回值(如果存在返回值)压入调用者栈帧的操作数栈中

3.调整PC计数器的值以指向方法调用指令后面的一条指令

## 本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

## 程序计数器

程序计数器就是记录当前线程执行程序的位置，改变计数器的值来确定执行的下一条指令，比如循环、分支、方法跳转、异常处理，线程恢复都是依赖程序计数器来完成。

Java虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是线程私有的。

## 直接内存

直接内存并不是虚拟机内存的一部分，也不是Java虚拟机规范中定义的内存区域。jdk1.4中新加入的NIO，引入了通道与缓冲区的IO方式，它可以调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小。

## 常量池

分为两种形态：静态常量池和运行时常量池。

所谓静态常量池，即\*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。这种常量池主要用于存放两大类常量：字面量和符号引用量，字面量相当于Java语言层面常量的概念，如文本字符串，声明为final的常量值等，符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：

1.类和接口的全限定名

2.字段名称和描述符

3.方法名称和描述符

而运行时常量池，则是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在方法区中，我们常说的常量池，就是指方法区中的运行时常量池。

运行时常量池相对于CLass文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入CLass文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的就是String类的intern()方法。

String的intern()方法会查找在常量池中是否存在一份equal相等的字符串,如果有则返回该字符串的引用,如果没有则添加自己的字符串进入常量池。

常量池是为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。

例如字符串常量池，在编译阶段就把所有的字符串文字放到一个常量池中。

# Jmm内存模型

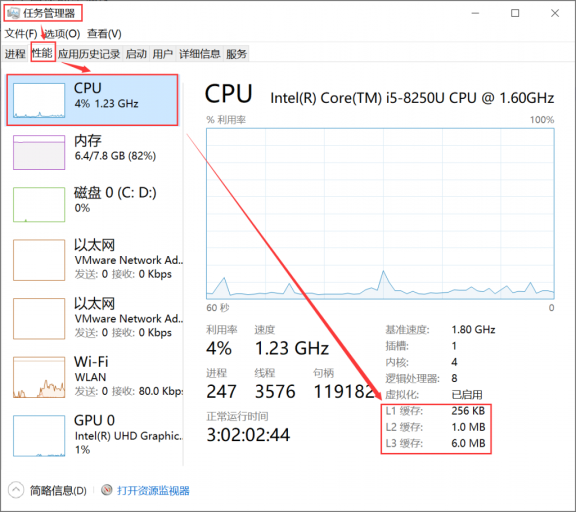
JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存，本地内存中存储了该线程以读/写共享变量的副本。

## 前言

CPU多级缓存。要想完全搞清楚Java内存模型，先要了解计算机硬件架构，特别是计算机CPU和主存之间的架构。在计算机中，cpu和内存的交互最为频繁，相比内存，磁盘读写太慢，内存相当于高速的缓冲区。但是随着cpu的发展，内存的读写速度也远远赶不上cpu。因此cpu厂商在每颗cpu上加上高速多级缓存，用于缓解这种情况。

三级缓存（L1、L2、L3），L1最靠近CPU核心，L2其次，L3再次。运行速度方面：L1最快、L2次快、L3最慢；容量大小方面：L1最小、L2较大、L3最大。CPU会先在最快的L1中寻找需要的数据，找不到再去找次快的L2，还找不到再去找L3，L3都没有那就只能去内存找了。

在多核cpu中，每个处理器都有各自的高速缓存(L1,L2,L3)，而主内存确只有一个。



### 1.缓存一致性问题

CPU需要修改某个数据，是先去Cache中找，如果Cache中没有找到，再去内存中找，然后把数据复制到Cache中，下次就不需要再去内存中寻找了，然后进行修改操作。而修改操作的过程是先在Cache里面修改数据，然后再把数据刷新到主内存。

其他CPU需要读取数据，也是先去Cache中去寻找，如果找到了就不会去内存找了。 所以当两个CPU的Cache同时都拥有某个数据，其中一个CPU修改了数据，另外一个CPU是无感知的，并不知道这个数据已经不是最新的了，它要读取数据还是从自己的Cache中读取，这样就导致了“缓存不一致”。

### 2.解决方法

1.总线加锁（此方法性能较低，现在已经不会再使用）。

2.MESI协议： 这是Intel提出的，MESI协议也是相当复杂，在这里我就简单的说下：当一个CPU修改了Cache中的数据，会通知其他缓存了这个数据的CPU，其他CPU会把Cache中这份数据的Cache Line置为无效，要读取数据的话，直接去内存中获取，不会再从Cache中获取了。

## 带来的问题

可见性问题：

原子性问题：

重排序：

as-if-serial

**不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。编译器、runtime和处理器都必须遵守as-if-serial语义。为了遵守as-if-serial语义，编译器和处理器**不会对存在数据依赖关系**的操作做重排序，因为这种重排序会改变执行结果。**

Happens-Before

**1）如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。(对程序员来说)**

**2）两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种重排序是允许的(对编译器和处理器 来说) （懵逼，这不是矛盾了吗- \_ -#）**

## 2.解决方法

1.内存屏障--禁止重排序

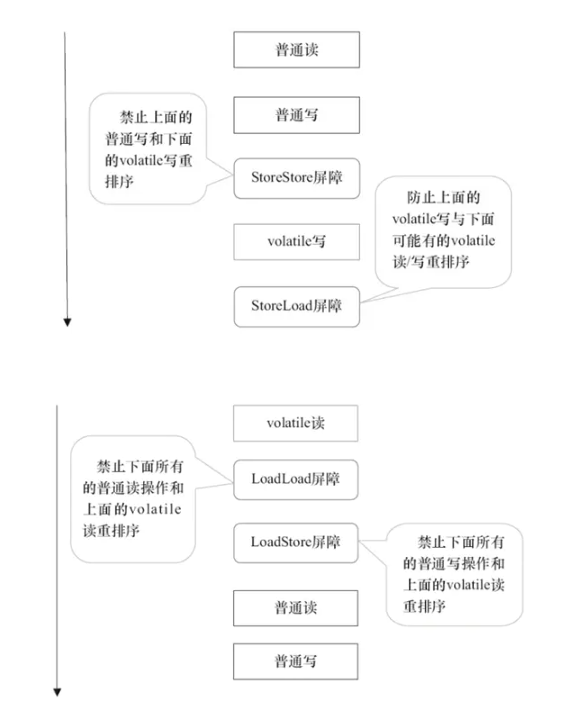


### 1.内存屏障插入策略

JMM对volatile的内存屏障插入策略：

在每个volatile写操作的前面插入一个StoreStore屏障。在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。在每个volatile读操作的后面插入一个LoadStore屏障。



## 3.volatile的内存语义

volatile写的内存语义如下：当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。

volatile读的内存语义如下：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

### 实现原理

有volatile变量修饰的共享变量进行写操作的时候会使用CPU提供的Lock前缀指令：

1.将当前处理器缓存行的数据写回到系统内存

2.这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

# Tcp和udp区别

# TCP三次握手

# Volitate关键字

# Synchronized锁升级过程

# AQS源码

# CAS理解

CAS(compare and swap)即比较并交换。它是解决多线程的环境下使用锁造成性能损耗的一种机制。

CAS操作一般包含三个操作数——内存位置（V）、预期原值（A）和新值(B)

当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false。

CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用，允许java调用其他语言

借助C来调用CPU底层指令实现的。

Unsafe类是CAS的核心类，提供硬件级别的原子操作。

## 缺点

开销大

并发量比较高的情况下，如果反复尝试更新某个变量，却又一直更新不成功，会给CPU带来较大的压力

不能保证代码块的原子性

只能保证一个变量的原子性操作，而多个方法或者代码块之的原子性则无法保证。

ABA问题

当变量从A修改为B在修改回A时，变量值等于期望值A，但是无法判断是否修改，CAS操作在ABA修改后依然成功。

java.util.concurrent包为了解决这个问题，提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证CAS的正确性。

不过目前来说这个类比较"鸡肋"，大部分情况下ABA问题并不会影响程序并发的正确性，如果需要解决ABA问题，使用传统的互斥同步可能回避原子类更加高效。

## AtomicInteger

Unsafe是CAS的核心，也是AtomicInteger的核心

valueOffset是变量值在内存中的偏移地址，而Unsafe提供了相应的方法

value是volatile修饰，其实这是真正存储值的变量

关键代码

private static final Unsafe unsafe = Unsafe.getUnsafe();

// 属性value值再内存中偏移量

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

// AtomicInteger 本身是个整型，所以最重要的属性就是value

private volatile int value;

public final int getAndAddInt(Object o, long offset, int delta) {

int v;

do {

v = getIntVolatile(o, offset);

} while (!compareAndSwapInt(o, offset, v, v + delta));

return v;

}

# Java阻塞队列

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

## 1.处理方法

方法\处理方式 抛出异常 返回特殊值 一直阻塞 超时退出

插入方法 add(e) offer(e) put(e) offer(e,time,unit)

移除方法 remove() poll() take() poll(time,unit)

检查方法 element() peek() 不可用 不可用

抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException(“Queue full”)异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常 。

返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null

一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。

超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出。

## 内置阻塞队列

JDK7提供了7个阻塞队列。分别是

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

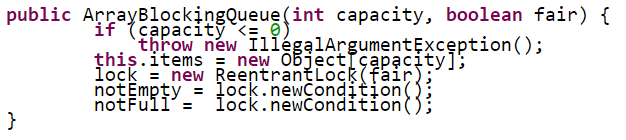
LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

### 1.ArrayBlockingQueue

ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。

访问者的公平性是使用可重入锁实现的。



### 2. LinkedBlockingQueue

LinkedBlockingQueue是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为Integer.MAX\_VALUE。相当与无界了。此队列按照先进先出的原则对元素进行排序。

### 3.PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列，也可以通过比较器comparator来指定元素的排序规则。元素按照升序排列。

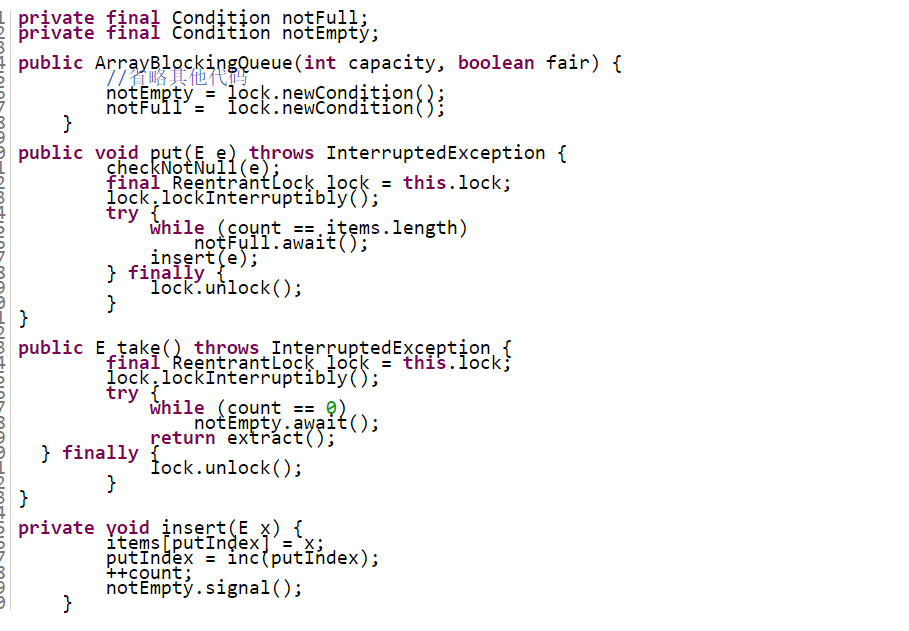
### 4.DelayQueue

DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。

### SynchronousQueue

SynchronousQueue是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。SynchronousQueue可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue的吞吐量高于LinkedBlockingQueue 和 ArrayBlockingQueue。

## 源码



当我们往队列里插入一个元素时，如果队列不可用，阻塞生产者主要通过LockSupport.park(this);来实现。

继续进入源码，发现调用setBlocker先保存下将要阻塞的线程，然后调用unsafe.park阻塞当前线程。

unsafe.park是个native方法，

# Java线程池

## 好处

1、降低资源消耗

可以重复利用已创建的线程降低线程创建和销毁造成的消耗。

2、提高响应速度

当任务到达时，任务可以不需要等到线程创建就能立即执行。

3、提高线程的可管理性

线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。

## 参数

corePoolSize、maximumPoolSize、keepAliveTime、unit、workQueue、threadFactory、handler。

### 1.corePoolSize线程池核心线程大小

线程池中会维护一个最小的线程数量，即使这些线程处理空闲状态，他们也不会被销毁，除非设置了allowCoreThreadTimeOut。这里的最小线程数量即是corePoolSize。

### 2.maximumPoolSize 线程池最大线程数量。

一个任务被提交到线程池后，首先会缓存到工作队列（后面会介绍）中，如果工作队列满了，则会创建一个新线程，然后从工作队列中的取出一个任务交由新线程来处理，而将刚提交的任务放入工作队列。线程池不会无限制的去创建新线程，它会有一个最大线程数量的限制，这个数量即由maximunPoolSize来指定。

### 3.keepAliveTime空闲线程存活时间。

一个线程如果处于空闲状态，并且当前的线程数量大于corePoolSize，那么在指定时间后，这个空闲线程会被销毁，这里的指定时间由keepAliveTime来设定

### 4.unit空间线程存活时间单位。

keepAliveTime的计量单位。

### 5.workQueue工作队列。

新任务被提交后，会先进入到此工作队列中，任务调度时再从队列中取出任务。jdk中提供了多种工作队列。

### 6.threadFactory 线程工厂

创建一个新线程时使用的工厂，可以用来设定线程名、是否为daemon线程等等

### 7.handler 拒绝策略

当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，该如何处理呢。这里的拒绝策略，就是解决这个问题的，jdk中提供了4种拒绝策略：

#### 1.CallerRunsPolicy

该策略下，在调用者线程中直接执行被拒绝任务的run方法，除非线程池已经shutdown，则直接抛弃任务。

#### 2.AbortPolicy

该策略下，直接丢弃任务，并抛出RejectedExecutionException异常。

#### 3.DiscardPolicy

该策略下，直接丢弃任务，什么都不做。

#### 4.DiscardOldestPolicy

该策略下，抛弃进入队列最早的那个任务，然后尝试把这次拒绝的任务放入队列

## 3.任务提交到线程池执行流程

1.如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（需要获得全局锁）

2.如果运行的线程等于或多于corePoolSize ,则将任务加入BlockingQueue

3.如果无法将任务加入BlockingQueue(队列已满)，则创建新的线程来处理任务（需要获得全局锁）

4.如果创建新线程将使当前运行的线程超出maxiumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

## 4.提交任务、关闭线程池

可以向ThreadPoolExecutor提交两种任务：Callable和Runnable。

 1.Callable

该类任务有返回结果，可以抛出异常。

通过submit函数提交，返回Future对象。

可通过get获取执行结果。

2.Runnable

该类任务只执行，无法获取返回结果，并在执行过程中无法抛异常。

通过execute提交。

关闭线程池有两种方式：shutdown和shutdownNow，关闭时，会遍历所有的线程，调用它们的interrupt函数中断线程。但这两种方式对于正在执行的线程处理方式不同。

shutdown()

仅停止阻塞队列中等待的线程，那些正在执行的线程就会让他们执行结束。

shutdownNow()

不仅会停止阻塞队列中的线程，而且会停止正在执行的线程。

## 5.合理配置线程池

## 6.内置线程池

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

new ThreadPoolExecutor(1,1,0L,

TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>())

2.newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

new ThreadPoolExecutor(nThreads, nThreads,

0L,TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>())

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>())

4.newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。

super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,

new DelayedWorkQueue())

5.WorkStealingThreadPool

工作窃取线程池

## 7.源码

# 线程的生命周期、创建线程的三种方式

# 深拷贝和浅拷贝

# Synchronized与ReentrantLock 区别

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化借鉴了ReenTrantLock中的CAS技术。

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器就减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

monitorenter ：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：

## 1.等待可中断

        1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。

## 2.可选择公平锁

  公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁是非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

ReentrantLock的内部类Sync继承了AQS，分为公平锁FairSync和非公平锁NonfairSync。

公平锁：线程获取锁的顺序和调用lock的顺序一样，FIFO；

非公平锁：线程获取锁的顺序和调用lock的顺序无关，全凭运气。

ReentrantLock默认使用非公平锁是基于性能考虑，公平锁为了保证线程规规矩矩地排队，需要增加阻塞和唤醒的时间开销。如果直接插队获取非公平锁，跳过了对队列的处理，速度会更快。

## 3.锁可绑定条件Condition

 synchronized不能绑定； ReentrantLock通过绑定Condition结合await()/singal()方法实现线程的精确唤醒，而不是像synchronized通过Object类的wait()/notify()/notifyAll()方法要么随机唤醒一个线程要么唤醒全部线程。

private Condition c3 = lock.newCondition();

## 4.锁的对象

synchronzied锁的是对象，锁是保存在对象头里面的，根据对象头数据来标识是否有线程获得锁/争抢锁；ReentrantLock锁的是线程，根据进入的线程和int类型的state标识锁的获得/争抢。

## 5.底层实现

synchronized 是JVM层面的锁，是Java关键字，通过monitor对象来完成（monitorenter与monitorexit），对象只有在同步块或同步方法中才能调用wait/notify方法，ReentrantLock 是从jdk1.5以来（java.util.concurrent.locks.Lock）提供的API层面的锁。

synchronized 的实现涉及到锁的升级，具体为无锁、偏向锁、自旋锁、向OS申请重量级锁，ReentrantLock实现则是通过利用CAS（CompareAndSwap）自旋机制保证线程操作的原子性和volatile保证数据可见性以实现锁的功能。

## 6.手动释放

synchronized 不需要用户去手动释放锁，synchronized 代码执行完后系统会自动让线程释放对锁的占用； ReentrantLock则需要用户去手动释放锁，如果没有手动释放锁，就可能导致死锁现象。一般通过lock()和unlock()方法配合try/finally语句块来完成，使用释放更加灵活。

# ReentrantLock 特点及源码

# 悲观锁与乐观锁

# Mysql innodb和myism区别

1.InnoDB支持事务和外键以及行级锁，MyISAM不支持。

2.MyISAM读性能优于InnoDB。

3.MyISAM索引和数据是分开的，而且索引是压缩的，而InnoDB索引和数据是紧密捆绑在一起的，无法压缩，所以InnoDB的体积比MyISAM庞大。

4.MyISAM引擎索引结构的叶子节点的数据域，存放的并不是实际的数据类型，而是数据记录的地址。索引文件与数据文件分离，这样的索引称之：“非聚簇索引”。

5.InnoDB引擎索引结构的叶子节点的数据域，存放是就是实际的数据记录。这样的索引被称为“聚簇索引”，一个表只能有一个聚簇索引。

6.InnoDB并不保存表的具体行数，也就是说，执行select count(\*) from table时，InnoDB要扫描整个表来计算有多少行，但是MyISAM只要简单读出保存好的行数即可。注意的是，当count(\*)语句包含where条件时，两个表的操作是一样的。

7.InnoDB表的行锁也不是绝对的，假如在执行一个SQL语句时MySQL不能确定扫描的范围，InnoDB表同样也会锁全表。

在where条件没有主键时，InnoDB照样会锁全表。

那么mysql何时使用MyISAM，什么时候使用InnoDB?

INNODB和MyISAM是mysql数据库提供的两种存储引擎。INNODB会支持一些关系数据库的高级功能，如事务功能和行级锁；MyISAM不支持，但是MyISAM的性能更优，占用的空间少。

如果应用程序一定要使用事务，那么要选择INNODB引擎。但是INNODB的行级锁是有条件的。在where条件没有使用主键时，照样会锁住全表。比如delete from mytable

若是应用程序对查询性能要求高，就要使用MyISAM。MyISAM索引和数据是分开的，而且索引是压缩的，可以更好的利用内存。所以它的查询性能明细优于Innodb。

# 设计模式

## 策略模式

主要角色如下。

1.抽象策略（Strategy）类：定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。

2.具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现。

3.环境（Context）类：持有一个策略类的引用，最终给客户端调用。

优点：

1.多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句。

2.策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。

3.策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。

4.策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。

## 2.责任链模式

为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。

## 3.代理模式

## 4.原型模式

## 5.工厂模式

## 6.单例模式

## 7.适配器模式

## 装饰者模式

## 模板方法模式

## 观察者模式

## 建造者模式

# JUC包用过哪些

# Java集合用过哪些

# Spring生命周期

1. 实例化 Instantiation

2. 属性赋值 Populate

3. 初始化 Initialization

4. 销毁 Destruction

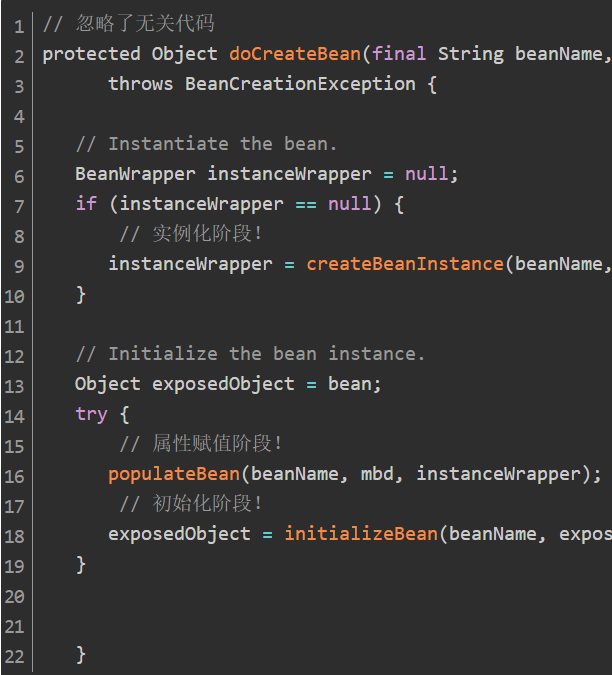
主要逻辑都在doCreate()方法中。

其中关键子方法：

createBeanInstance() -> 实例化

populateBean() -> 属性赋值

initializeBean() -> 初始化

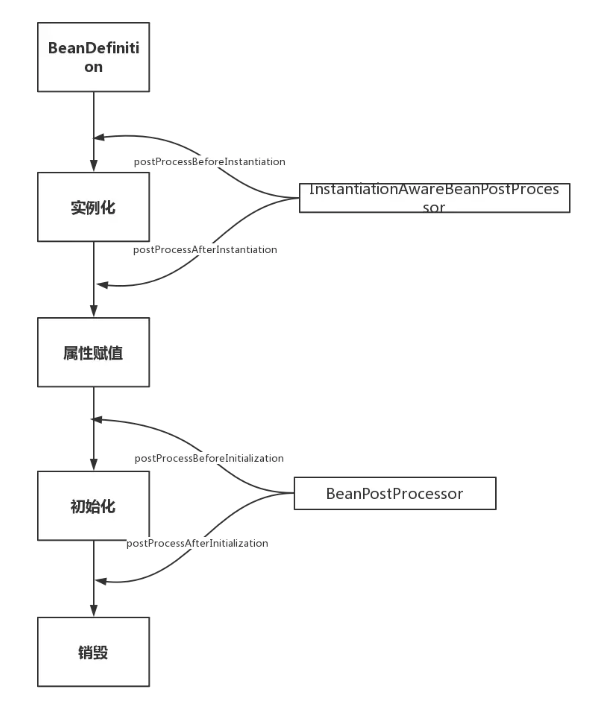


有关aop的接口：

BeanPostProcessor、InstantiationAwareBeanPostProcessor

这两兄弟可能是Spring扩展中最重要的两个接口！InstantiationAwareBeanPostProcessor作用于实例化阶段的前后，BeanPostProcessor作用于初始化阶段的前后。

InstantiationAwareBeanPostProcessor实际上继承了BeanPostProcessor接口。



2.只调用一次的接口

可以分为两类：

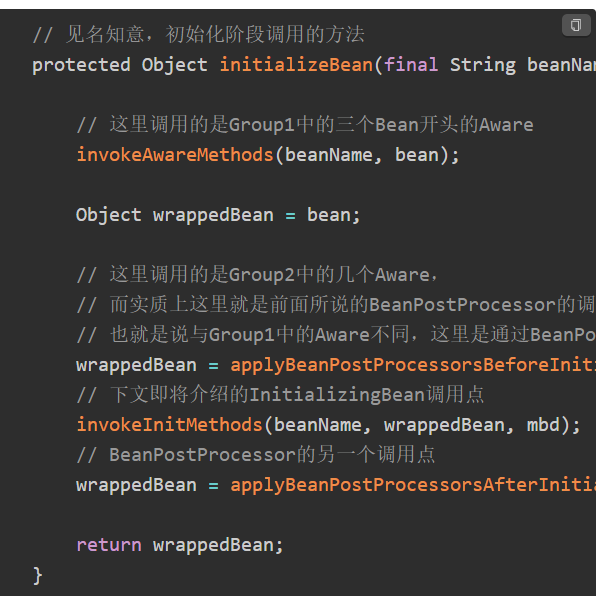
Aware类型的接口、生命周期接口

Aware类型的接口的作用就是让我们能够拿到Spring容器中的一些资源。例如BeanNameAware可以拿到BeanName，以此类推。调用时机需要注意：所有的Aware方法都是在初始化阶段之前调用的

其他的还有：BeanClassLoaderAware， BeanFactoryAware，EnvironmentAware,

EmbeddedValueResolverAware ,ApplicationContextAware

Aware调用时机是在initializeBean方法里，这也说明了Aware都是在初始化阶段之前调用的。

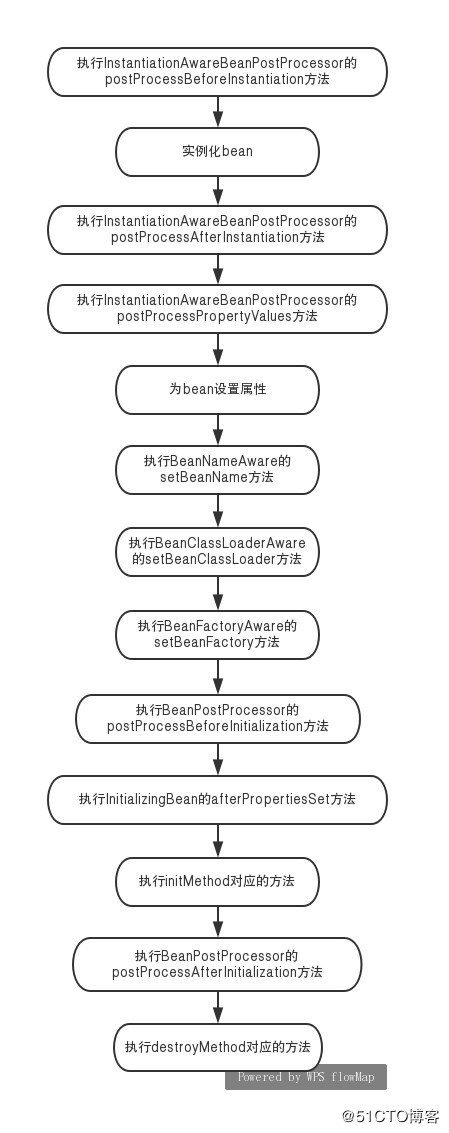


并不是所有的Aware接口都使用同样的方式调用。Bean××Aware都是在代码中直接调用的，而ApplicationContext相关的Aware是通过BeanPostProcessor#postProcessBeforeInitialization()实现的。

InitializingBean 对应生命周期的初始化阶段，在上面源码的invokeInitMethods(beanName, wrappedBean, mbd)。

DisposableBean 类似于InitializingBean，对应生命周期的销毁阶段，以ConfigurableApplicationContext#close()方法作为入口，实现是通过循环取所有实现了DisposableBean接口的Bean然后调用其destroy()方法。

整体流程图：



# Spring的aop和ioc理解及术语名词

## ioc

Inversion of Control

控制反转: 将代码的调用权（控制权）从调用方转移给被调用方（服务提供方），把创建和查找依赖对象的控制权交给了容器。

Dependency Injection（DI）

依赖注入：组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。

## **2.aop**

Aspect Oriented Programming

面向切面编程。

### 1.名词

(1)横切关注点：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

(2)Aspect(切面):通常是一个类，里面可以定义切入点和通知

(3)JointPoint(连接点):程序执行过程中明确的点，一般是方法的调用。被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器

(4)Advice(通知):AOP在特定的切入点上执行的增强处理，有before(前置),after(后置),afterReturning(最终),afterThrowing(异常),around(环绕)

(5)Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式

(6)weave(织入)：将切面应用到目标对象并导致代理对象创建的过程

(7)introduction(引入)：在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段

(8)AOP代理(AOP Proxy)：AOP框架创建的对象，代理就是目标对象的加强。Spring中的AOP代理可以使JDK动态代理，也可以是CGLIB代理，前者基于接口，后者基于子类

### 2.增强类型：

#### **1.前置通知**

Before:在目标方法被调用之前做增强处理,@Before只需要指定切入点表达式即可

#### **2.后置通知**

After:在目标方法完成之后做增强，无论目标方法时候成功完成。

#### **3. 返回通知**

AfterReturning:在目标方法正常完成后做增强,@AfterReturning除了指定切入点表达式后，还可以指定一个返回值形参名returning,代表目标方法的返回值

#### **4. 异常通知**

AfterThrowing:主要用来处理程序中未处理的异常,@AfterThrowing除了指定切入点表达式后，还可以指定一个throwing的返回值形参名,可以通过该形参名

来访问目标方法中所抛出的异常对象

#### **5.环绕通知**

Around:环绕通知,在目标方法完成前后做增强处理,环绕通知是最重要的通知类型,像事务,日志等都是环绕通知。可以控制切入点是否继续执行。

### 3.实现方式

<https://blog.csdn.net/xiaojin21cen/article/details/79487769>

#### 1.静态代理

ApectJ

在编译阶段将Aspect织入Java字节码中，运行的时候就是经过增强之后的AOP对象。

需要特殊的类编译器。

#### 动态代理

动态代理就是说AOP框架不会去修改字节码，而是在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

##### 1.jdk动态代理

JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。

核心类 ：

InvocationHandler 接口

Proxy.newProxyInstance()

过程：

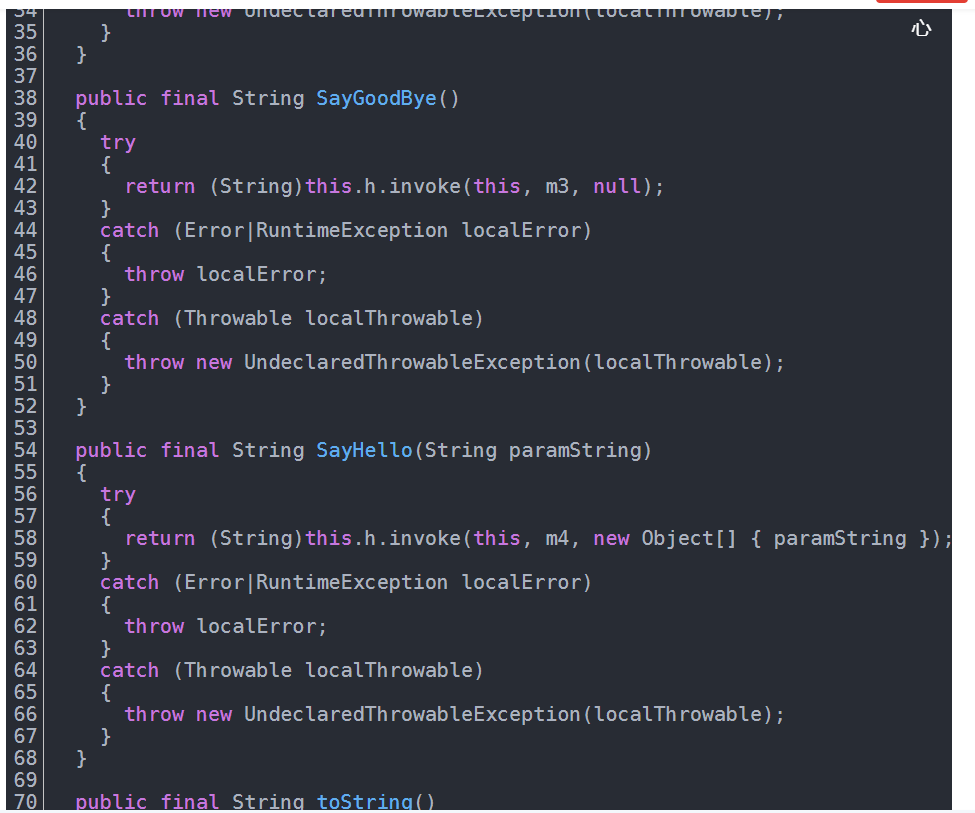
1.通过实现 InvocationHandler 接口创建自己的调用处理器；

2.通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理类；

3.通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型；

4.通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。

生成代理类class文件反编译后的结果：



JDK生成的最终真正的代理类，它继承自Proxy并实现了我们定义的Subject接口，

在实现Subject接口方法的内部，通过反射调用了InvocationHandlerImpl的invoke方法。

##### 2.cglib动态代理

利用ASM开源包，修改代理对象类的字节码来实现。

过程：

1.创建Enhancer实例

2.通过setSuperclass方法来设置目标类

3.通过setCallback 方法来设置拦截对象

4.create方法生成Target的代理类，并返回代理类的实例

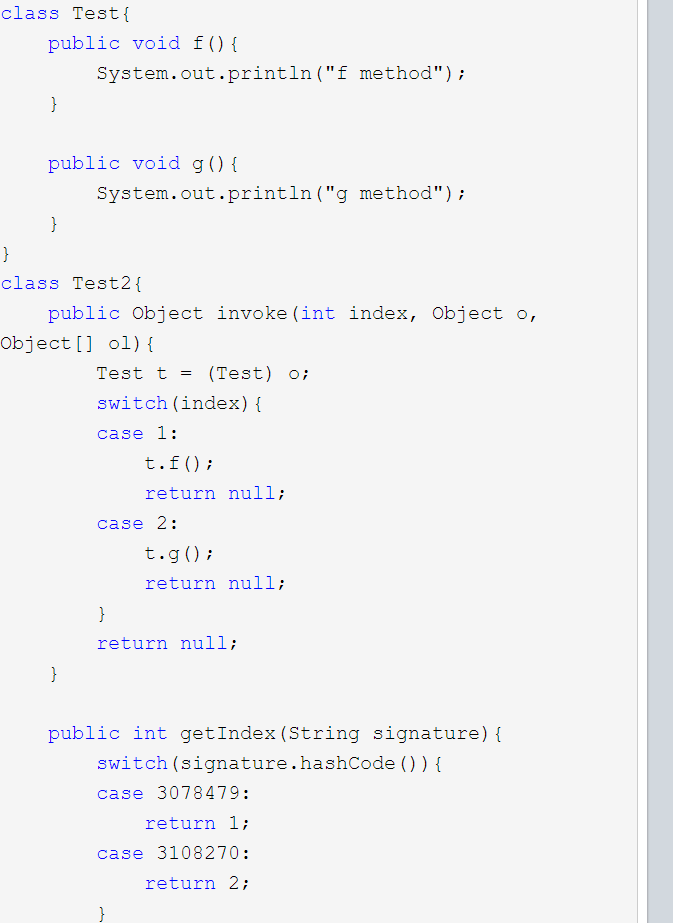
代理类反编译后的结果：

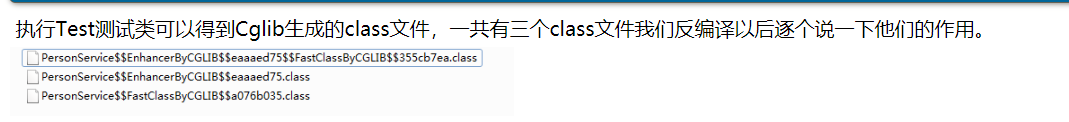


当代理对象的执行sayHello方法时，会首先判断一下是否存在实现了MethodInterceptor接口的CGLIB$CALLBACK\_0;，如果存在，则将调用MethodInterceptor中的intercept方法。

###### **FastClass机制**

Jdk动态代理的拦截对象是通过反射的机制来调用被拦截方法的，反射的效率比较低，所以cglib采用了FastClass的机制来实现对被拦截方法的调用。FastClass机制就是对一个类的方法建立索引，通过索引来直接调用相应的方法。





PersonService$$EnhancerByCGLIB$$eaaaed75就是cglib生成的代理类，它继承了PersonService类。

PersonService$$EnhancerByCGLIB$$eaaaed75$$FastClassByCGLIB$$355cb7ea.class就是代理类的FastClass，

PersonService$$FastClassByCGLIB$$a076b035.class就是被代理类的FastClass。

###### cglib与jdk动态代理对比

1.JDK动态代理是实现了被代理对象的接口，Cglib是继承了被代理对象。

2.JDK和Cglib都是在运行期生成字节码，JDK是直接写Class字节码，Cglib使用ASM框架写Class字节码，Cglib代理实现更复杂，生成代理类比JDK效率低。

3.JDK调用代理方法，是通过反射机制调用，Cglib是通过FastClass机制直接调用方法，Cglib执行效率更高。

4.jdk只能代理接口，cglib可以代理类和接口。

5.cglib不能代理final修饰的方法。

# Spring解决循环依赖

## 三层缓存

1.singletonObjects：用于存放完全初始化好的 bean，从该缓存中取出的 bean 可以直接使用

2.earlySingletonObjects：提前曝光的单例对象的cache，存放原始的 bean 对象（尚未填充属性），用于解决循环依赖

3.singletonFactories：存放我刚实例化的bean，通过ObjectFactory，可以让如果有提前需要bean的需要可以调用该objectfactory ，其会将刚实例化好的bean经过beanPostProcessor的getEarlyReference处理进行返回。

我们先实例化A，实例化好后  
调用addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));  
() -> getEarlyBeanReference(beanName, mbd, bean)是objectFactory的实现匿名类，然后此时我们设置属性的时候会发现我们还依赖B，于是我们去实例化B而当我们此时实例化B的会经过下面这个方法getSingleton()

由此可见 如果之前已经把A存放在singletonFactories，那么B会把其调出来放入earlySingletonObjects，然后整个单例创建完成在塞入singletonObjects。存放在singletonFactories好处是可扩展，我们在这个里面会调用beanPostProcessor 从而可以在我们实现提前获取对象引用的时候进行一些操作。

## 2.注入过程

1.先从一级缓存singletonObjects中去获取。（如果获取到就直接return）

2.如果获取不到或者对象正在创建中（isSingletonCurrentlyInCreation()），那就再从二级缓存earlySingletonObjects中获取。（如果获取到就直接return）

3.如果还是获取不到，且允许singletonFactories（allowEarlyReference=true）通过getObject()获取。就从三级缓存singletonFactory.getObject()获取。（如果获取到了就从singletonFactories中移除，并且放进earlySingletonObjects。其实也就是从三级缓存移动（**剪切**）到了二级缓存）

## 3.三种场景

a.构造器注入循环依赖

项目启动失败抛出异常BeanCurrentlyInCreationException, Spring解决循环依赖依靠的是Bean的“中间态”这个概念，而这个中间态指的是已经实例化，但还没初始化的状态。而构造器是完成实例化的东西，所以构造器的循环依赖无法解决

b.field属性注入（setter方法注入）循环依赖

正常

c.prototype field属性注入循环依赖

启动时是不会报错的（因为非单例Bean默认不会初始化，而是使用时才会初始化）。

手动getBean()或者在一个单例Bean内@Autowired时报错

# Springmvc执行流程

1.用户发送请求至前端控制器DispatcherServlet

2.DispatcherServlet收到请求调用处理器映射器HandlerMapping。

3.处理器映射器根据请求url找到具体的处理器，生成处理器执行链HandlerExecutionChain(包括处理器对象和处理器拦截器)一并返回给DispatcherServlet。

4.DispatcherServlet根据处理器Handler获取处理器适配器HandlerAdapter执行HandlerAdapter处理一系列的操作，如：参数封装，数据格式转换，数据验证等操作

5.执行处理器Handler(Controller，也叫页面控制器)。

6.Handler执行完成返回ModelAndView

7.HandlerAdapter将Handler执行结果ModelAndView返回到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model填充至视图中）。

11.DispatcherServlet响应用户。

# 数据库索引原理

# 聚集索引与非聚集索引

# 怎么优化sql及使用索引

# 事务及mvcc

## 基本要素（ACID）

### 1.原子性

原子性（Atomicity）：事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样。也就是说事务是一个不可分割的整体，就像化学中学过的原子，是物质构成的基本单位。

InnoDB中记录数据的基本单位为页（InnoDB Page，默认16KB）。

原子性通过当undo log**实现。**执行insert/update/delete等写操作时，除了要修改对应数据页之外，还会对之前的数据进行备份（**记录至**undo日志**中）**，记录相反操作的sql及旧的数据行记录。如果事务需要回滚，找到对应的undo 记录进行应用回滚即可。

### 2.一致性

　　 一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏。比如A向B转账，不可能A扣了钱，B却没收到。

### 3.隔离性

　　 隔离性（Isolation）：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

### 4持久性

　　 持久性（Durability）：事务完成后，事务对数据库的所有更新将被持久化保存到数据库，不能回滚。

持久性是通过redo log实现的。事务的所有修改操作(增、删、改)，数据库都会生成一条redo日志记录到redo log。区别于undo log记录SQL语句、redo log记录的是事务对数据库的哪个数据页做了什么修改，属于物理日志。

redo日志应用场景：数据库系统直接崩溃，需要进行恢复，一般数据库都会使用按时间点备份的策略，首先将数据库恢复到最近备份的时间点状态，之后读取该时间点之后的redo log记录，重新执行相应记录，达到最终恢复的目的。

## 2.并发问题

1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

　　2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。

　　3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

　　小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

## 3.事务隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 读已提交（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

　 1、事务隔离级别为读提交时，写数据只会锁住相应的行

　 2、事务隔离级别为可重复读时，如果检索条件有索引（包括主键索引）的时候，默认加锁方式是next-key 锁；如果检索条件没有索引，更新数据时会锁住整张表。一个间隙被事务加了锁，其他事务是不能在这个间隙插入记录的，这样可以防止幻读。

　 3、事务隔离级别为串行化时，读写数据都会锁住整张表

4、 mysql中，默认的事务隔离级别是可重复读。Oracle默认的事务隔离级别是读已提交。

## 4.mvcc

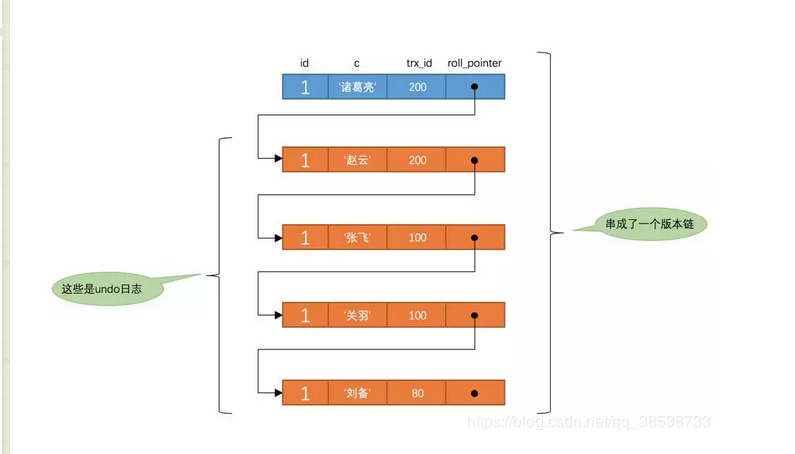
Mvcc多版本并发控制机制（MVCC，Multi-Version Concurrency Control）。MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问。

在Mysql的InnoDB引擎中就是指在已提交读和可重复读这两种隔离级别下的事务对于SELECT操作会访问版本链中的记录的过程。通过保存数据在某个时间点的快照来实现，这意味着一个事务无论运行多长时间，在同一个事务里能够看到数据一致的视图。别的事务可以修改这条记录，反正每次修改都会在版本链中记录。SELECT可以去版本链中拿记录，这就实现了读-写，写-读的并发执行，提升了系统的性能。

在InnoDB引擎表中，它的聚簇索引记录中有两个必要的隐藏列：

trx\_id: 这个id用来存储的每次对某条聚簇索引记录进行修改的时候的事务id。

roll\_pointer: 每次对哪条聚簇索引记录有修改的时候，都会把老版本写入undo日志中。这个roll\_pointer就是存了一个指针，它指向这条聚簇索引记录的上一个版本的位置，通过它来获得上一个版本的记录信息。(注意插入操作的undo日志没有这个属性，因为它没有老版本)。可以将这些undo日志都连起来，串成一个链表，所以现在的情况就像下图一样：



### 1.版本链

对该记录每次更新后，都会将旧值放到一条undo日志中，就算是该记录的一个旧版本，随着更新次数的增多，所有的版本都会被roll\_pointer属性连接成一个链表，我们把这个链表称之为版本链，版本链的头节点就是当前记录最新的值。另外，每个版本中还包含生成该版本时对应的事务id，

### ****2.ReadView****

一致读视图。就是在某一时刻给事务系统trx\_sys打snapshot（快照），把当时trx\_sys状态（包括活跃读写事务数组）记下来，之后的所有读操作根据其事务ID与snapshot中的trx\_sys的状态作比较，以此判断read view对于事务的可见性。

ReadView中主要包含当前系统中还有哪些活跃的读写事务，把它们的事务id放到一个列表中，我们把这个列表命名为为m\_ids。这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

1.如果被访问版本的trx\_id属性值小于m\_ids列表中最小的事务id，表明生成该版本的事务在生成ReadView前已经提交，所以该版本可以被当前事务访问。

2.如果被访问版本的trx\_id属性值大于m\_ids列表中最大的事务id，表明生成该版本的事务在生成ReadView后才生成，所以该版本不可以被当前事务访问。

3.如果被访问版本的trx\_id属性值在m\_ids列表中最大的事务id和最小事务id之间，那就需要判断一下trx\_id属性值是不是在m\_ids列表中，如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本，如果最后一个版本也不可见的话，那么就意味着该条记录对该事务不可见，查询结果就不包含该记录。

在RC隔离级别下，是每个快照读都会生成并获取最新的Read View  
在RR隔离级别下，则是同一个事务中的第一个快照读才会创建Read View, 之后的快照读获取的都是同一个Read View。

**博客：**

<https://blog.csdn.net/qq_38538733/article/details/88902979>

<https://www.jianshu.com/p/d9bdf90005cb#!/xh>

<https://blog.csdn.net/zycxnanwang/article/details/105742160>

## 5.锁分类

Innodb支持表级锁和行级锁，myisam只支持表级锁

### 1.表锁

### 2.行级锁

行级锁又分为：共享锁和排他锁。

#### 共享锁

使用共享锁的方法是在select ... lock in share mode,只适用查询语句。

一个事务获取了共享锁，其他事务可以获取共享锁，不能获取排他锁，其他事务可以进行读操作，不能进行写操作。

#### 排他锁

使用排他锁的方法师在sql末尾加上for update，顺便一提在innodb引擎中，会默认在update,delete这种操作加上for update。

在mysql innodb中，行级锁的实现其实是依靠其对应的索引，所以如果操作的行没用到索引，那么用的还是表级锁。

如果事务T对数据A加上排他锁后，则其他事务不能再对A加任何类型的锁。获取到排他锁的事务既能读数据，又能修改数据。

对于insert、update、delete，InnoDB会自动给涉及的数据加排他锁；

对于一般的Select语句，InnoDB不会加任何锁

例子：

共享锁例子：

事务一：使用lock in share lock获取共享锁，开始事务，不提交也不回滚

事务二：对同一条记录进行修改，会产生阻塞，查询不会阻塞

事务三：对同一记录进行查询（加共享锁），不阻塞，共享锁允许其他事务也获取共享锁

排他锁例子：

事务一：select语句后跟for update获取排他锁，其他事务不能进行查询和修改的操作

事务二：对同一记录进行加锁查询会被阻塞

事务三：对该记录的加锁写操作会阻塞

<https://www.cnblogs.com/shamgod-lct/p/9318032.html>

比较

表级锁：开销小，加锁快；不会出现死锁；锁粒度大。发生冲突几率最大，并发度最低。

行级锁：开销大，加锁慢；会出现死锁，发生锁冲突几率最低，并发度最高。

页面锁：介于表级锁和行级锁之间。

InnoDB什么时候使用表锁，什么时候使用行锁？

表级锁：更适合于查询为主的场景。

行级锁：更适合于大量按索引条件并发更新少量不同的数据，同时又有并发查询的应用。

### 3.页锁

# 数据结构及算法

# Java对象头

# 雪花算法

# Springboot自动装配原理

# Linux基本命令

# Jvm调优命令及参数

# 一个http请求经历的过程

# 网络七层协议模型

# JAVA引用类型

# 死锁

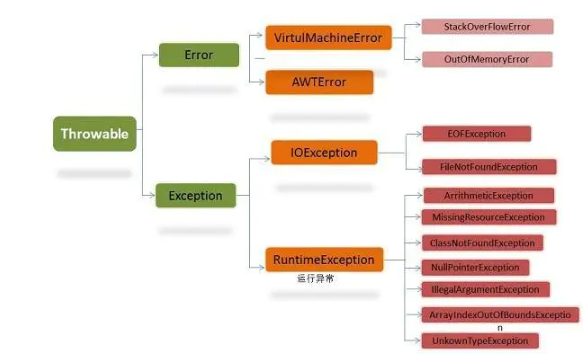
# 排序算法

# 限流算法

# 服务熔断、降级

# 布隆过滤器

# 异常分类



Throwable 类是 Java 语言中所有错误或异常的超类。

Error 是 Throwable 的子类，用于指示合理的应用程序不应该试图捕获的严重问题。Java 程序通常不捕获错误。错误一般发生在严重故障时，它们在Java程序处理的范畴之外。

Exception

一类是 IOException（I/O 输入输出异常），其中 IOException 及其子类异常又被称作「受检异常」

另一类是 RuntimeException（运行时异常），RuntimeException 被称作「非受检异常」。s

受检异常就是指，编译器在编译期间要求必须得到处理的那些异常

所有异常都必须是 Throwable 的子类。

如果希望写一个检查性异常类，则需要继承 Exception 类。

如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

# 分库分表分区

# ThreadLocal

# Kafka消息队列

# 自动装箱、拆箱

# 方法重写、重载

# 为什么要重写equal()和hashCode()

# 单点登录

# 负载均衡算法

# Servlet 生命周期

Servlet 通过调用 init () 方法进行初始化。

Servlet 调用 service() 方法来处理客户端的请求。

Servlet 通过调用 destroy() 方法终止（结束）。

1.被创建：执行init方法，只执行一次

　　1.1Servlet什么时候被创建？

　　--默认情况下，第一次被访问时，Servlet被创建，然后执行init方法；

　　--可以配置执行Servlet的创建时机；

2.提供服务：执行service方法，执行多次

3.被销毁：当Servlet服务器正常关闭时，执行destroy方法，只执行一次

# 过滤器、拦截器、监听器区别

过滤器：是在javaWeb中，前台传入的request,response提前过滤掉一些信息，或者提前设置一些参数，然后再传入servlet或者struts的action进行业务逻辑，比如在非法登录进行过滤掉非法的url（不是.jsp或者.action或者.do的地址请求，如果用户没有登陆都过滤掉）,或者在传入servlet或者struts的action前统一设置字符集，或者去除掉一些非法字符。主要为了减轻服务器负载，减少压力。主要的应用场景是：编码过滤、非法登录过滤、脏话过滤、代码过滤！

拦截器 ：主要是应用于面向切面编程！在程序员的业务逻辑前进行调用一个方法，或者在其方法后调用一个方法。

比如动态代理就是拦截器的简单实现，在你调用方法前输出日志（或者做其它业务逻辑的操作），也可以在你调用方法后输出日志，甚至在你抛出异常的时候做业务逻辑的操作。

监听器：Servlet的监听器Listener，它是实现了javax.servlet.ServletContextListener接口的服务器端程序，它也是随web应用的启动而启动，只初始化一次，随web应用的停止而销毁。

三者区别:

1、拦截器是基于java的反射机制的 - - - - - 过滤器是基于函数回调

2、拦截器不依赖与servlet容器 - - - - - 过滤器依赖与servlet容器，

3、拦截器只能对action请求起作用 - - - - - 过滤器则可以对几乎所有的请求起作用

4、拦截器可以访问action上下文、值栈里的对象 - - - - - 过滤器不能

5、在action的生命周期中，拦截器可以多次被调用 - - - - - 过滤器只能在容器初始化时被调用一次

监听器主要是做一些初始化的内容添加工作、设置一些基本的内容、比如一些参数或者是一些固定的对象等等

执行顺序 ：

过滤前 - 拦截前 - Action处理 - 拦截后 - 过滤后

# 数据库三大范式

# java内存泄漏、内存溢出定位

# 线程start()和run()区别

# 分布式事务

# http哪些请求方式是幂等

# nio，io

# rpc框架

# netty

# springcloud组件