# Java

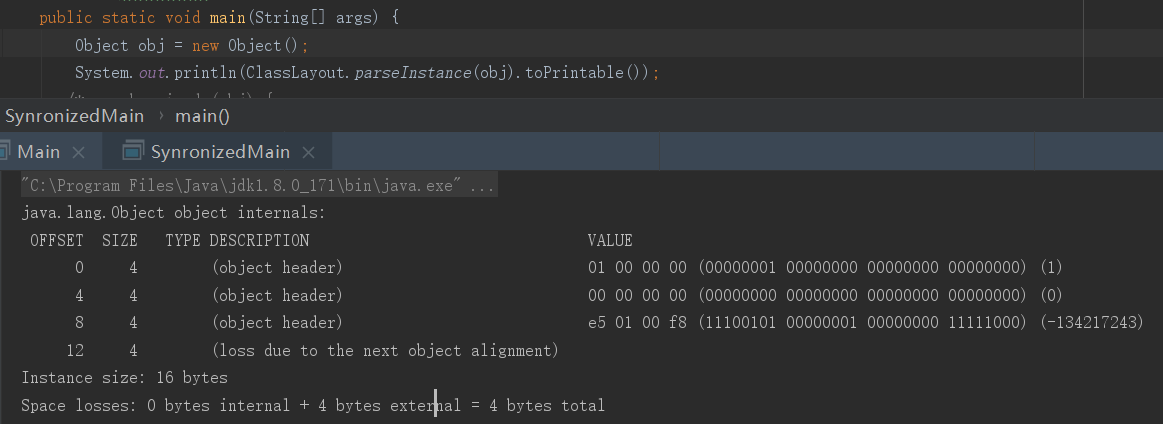
## 对象分配

栈上分配(逃逸分析(在该方法能访问使用，不会逸出到其他地方访问)、标量替换(可以将对象替换成对应的属性值))->2.tlab(thread local alloction buffer)->eden

对象的内存布局(在64位系统，UseCompressedOops来进行压缩指针)

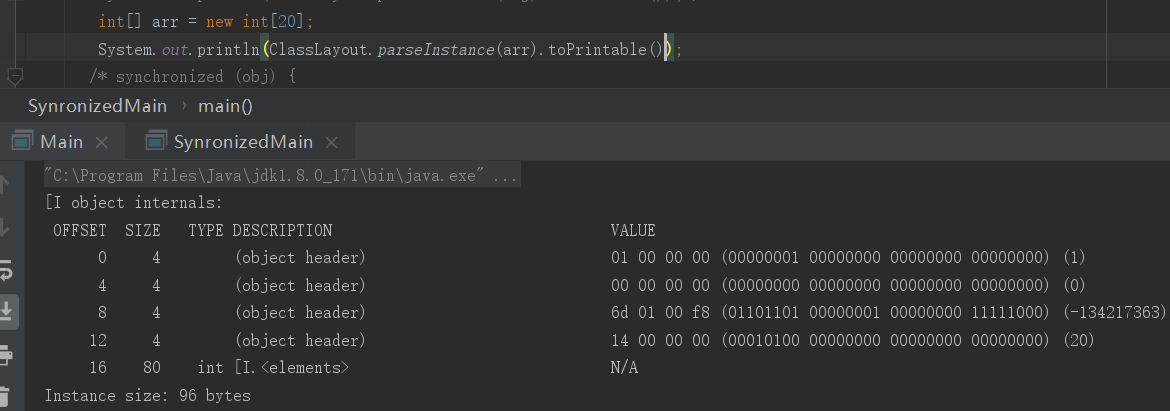
8字节内存头(mark word) + 4字节class point(对象的指针地址，64位系统是8字节) + 实例数据 + 补偿(加起来能够被8整除)

未使用压缩指针：classpoint只用了4个字节来存储(从0-7存储markword，8开始4个字节存储class point)



8字节内存头(mark word) + 4字节class point(对象的指针地址，64位系统是8字节) + 4字节位(数组个数) + 实例数据 + 补偿(加起来能够被8整除)

未使用压缩指针：classpoint只用了4个字节来存储(从0-7存储markword，8-11字节4个字节存储class point，12-15字节4个字节来存储数组的个数，16-95开始80个字节来存储实例数据)



## JAVA线程调度

抢占式调度：由jvm系统来调用，一个线程执行时间片，优先级高的拥有更多的时间片，但不是一直拥有，优先级低的不会得不到时间片。Jvm使用的是抢占式调度

协作式调度：由一个线程执行完，再通知另一个线程执行

## ThreadLocal

Thread中属性threadlocals：ThreadLocalMap来存储。Threadlocal为对应的key去查找对应的值

## JVM

经过fullgc之后，年轻代全部清空，压缩进入老年代；

一般一次major gc都会伴随一次minor gc，但是fullgc是指针对整个堆空间（年轻代、年老代）和永久代

### 年轻代晋升

1. 普通对象会在survivor中存活，直到年龄超过MaxTenuringThreshold，才会晋升到老年代
2. 动态计算年龄，如果超过某个年龄的survivor已有对象超过了survivor的大小的一半，那么超过这个年龄的对象直接进入老年代

### Metaspace

元空间，就是永久代的存储区域，以chunk(块)来分配，

Used: 加载的类空间量()

Capacity：当前分配块的元数据量(由于chunk不一定会被全部使用，所以Capacity>= used)

Commited: 提交空间块的数量（由于提交的chunk也可能被回收，而capacity只代表当前占用的，所以commited >= capacity）

Reserved: 保留的量（在jvm启动的时候，根据参数预留）

### SerialOld

### CardTable:

用于记录跨代引用的，只是记录某一块区域是否需要

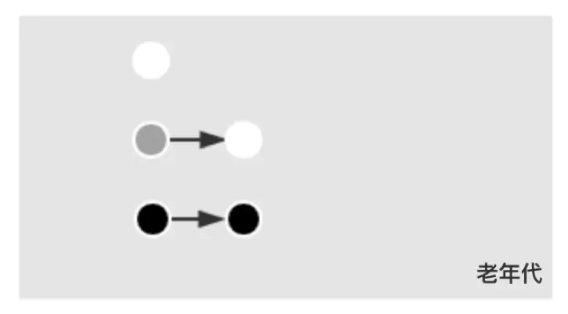
### CMS

分为两种，一种是主动gc(由于担保失败，需要进行一次gc或者System.gc)、另外一种由线程每2s执行一次键查(occupancy是否被超过了)则执行一次gc

**Fullgc是由vm thread发起的，会触发stw(stop the world)**

**周期性，background collection是由后台线程触发的，每2s触发一次检查**

|  |  |
| --- | --- |
| 参数 | 说明 |
| -XX:+CmsInitiatingOccupancyOnly | 如果没有设置，则根据jvm的统计信息进行gc回收。如果设置了，则根据Occupancy的信息统计进行 |
| -XX:CmsInitiatingOccupancyFraction | 占用比例，默认92%，当老年代的空间使用率达到这个值，才进行gc |
| -XX:CmsScheduleRemarkEdenSizeThrehold | 表示当进入重新标记前，如果新生代的占用大小超过了这个阈值，才会触发预处理阶段 |
| -XX:CmsScheduleRemarkEdenPenetration | 当eden的使用率达到这个值，则可停止预处理 |
| -XX:CmsMaxAbortablePrecleanTime | 预处理的最大时间，默认5s，如果超过这个值也会停止预处理 |
| -XX:CmsScavengeBeforeRemark | 是否在重新标记前都执行一次minor gc，年轻代gc。  这样保证在remark能够执行gc，但是会造成remark前eden太小而不必要gc和在预处理时已经处理过了 |
| -XX:UseCmsCompactAtFullCollection | 在进行fullgc时进行碎片整理 |
| -XX:CmsFullGcsBeforeCompaction | 在进行多少次不压缩的fullgc后，进行碎片整理，默认为0 |



白色：未标记对象，则表示可以回收

灰色：标记未处理，则说明还要继续往下处理搜索

黑色：标记已处理，如果搜索完了，则标记为黑色

**初始标记：**GCRoots可达的老年代对象、年轻代中指向老年代的对象，stw

**并发标记：**遍历初始标记的存活对象，这时候和用户线程并发执行，可能产生直接在老年代分配对象，新生代晋升到老年代、老年代的关系发生变更

**预处理：**处理新生代发现的引用(比如新生代的对象引用了老年代对象)、老年代内部结构发生变化

**终止预处理：**由于某些原因终止预处理

**重新标记：**

STW，这里对象可能会产生以下几种变化

老年代未标记被gcRoot关联

老年代未标记被新生代关联

老年代已标记对象关联未标记的老年代

新生代对象关联老年代被删除

可能还有其他场景

遍历新生代对象，重新标记

根据GCRoots重新标记

遍历老年代中的Dirty Card，重新标记

**并发清理：**开始清理。这个时候用户的线程并发执行，这时候产生的垃圾碎片是无法回收的。这边也会产生浮动垃圾。

如果在这个阶段，用户需要的空间，超出了cms预留的空间，那么就会触发full gc，而full gc的时候会去判断当前是否在cms中，如果是，则会产生concurrent mode failure，就会触发备案serial old的进行垃圾回收

**重新调整堆大小：**如果有需要，进行调整

**重置：**重置数据，等待下一轮回收

#### 问题：为什么重新标记阶段不能把并发标记阶段中产生的浮动垃圾给标记出来呢？

并发清理阶段，也会产生用户垃圾，导致无法删除

# 单元测试：

@Mock: mock生成对象，但是不在整个spring context生命周期

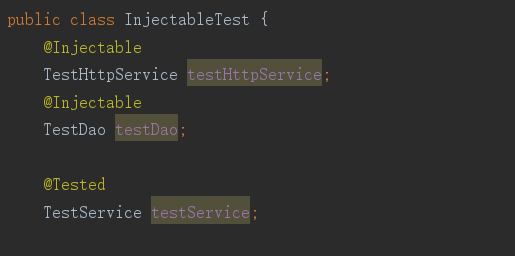
@MockBean: 由mockito对象，注入到整个spring context的生命周期

## Jmockito

Jmockito需要加上javaagent，因为实现是利用asm对类进行代理

@Injectable：需要注解的属性

@Tested 用于需要单元测试的类

实例方法



使用了Expectations进行录制、Verifications进行断言。

### Mockup对类的方法替换执行

这样就能够对静态方法进行mock

New MockUp(XxxClass.class) {

Public type method(….) {

Return xxx;

}

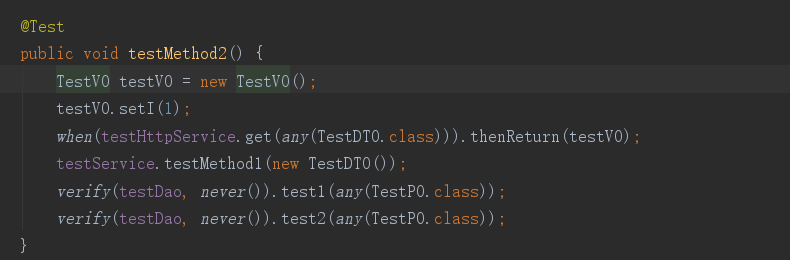
}

## Mockito

使用@Mock和@InjectMocks进行初始化



使用verify进行断言



### @Spy和@Mock

Spy的会实际去触发对应的类的方法运行，mock则不会

利用Mockito.doRealmethod().when(对象).method()；这样也能触发执行对应的mock对象中的method方法逻辑

# SkyWalking

APM: Application Performance Management->应用性能监控

分布式链路跟踪：aop实现跟踪，Javaagent技术

SkyWalkingAgent.premain

->PluginResourcesResolver.loaderPlugins->skywalking-plugin.def（就会针对不同类，来进行类加强）

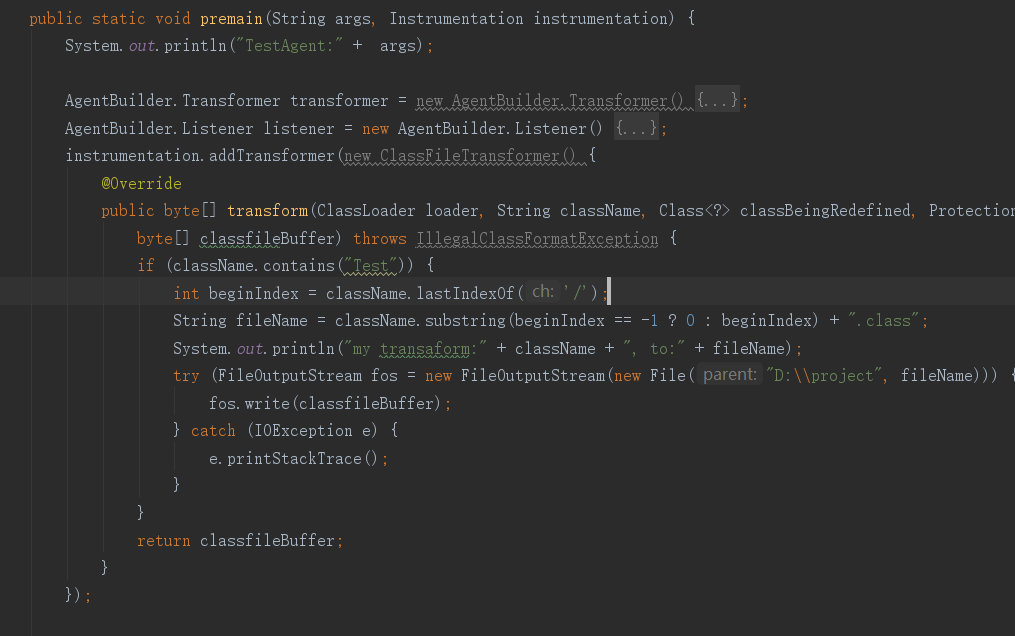
->instrumentation.addTransformer可以对对类的字节码进行修改

-> ByteBuddy进行字节码修改

->AgentClassLoader类加载

## ByteBuddy

## Java探针-agent



# Mysql

## 查询缓存

Have\_query\_cache:是否开启查询缓存，相同的sql查询，可以直接利用缓存进行结果过滤，提升查询效率，但是带来内存的消耗

## Innodb pool buffer

Innodb\_buffer\_pool\_size来设置缓存区的数据，也就是在内存中的数据，数据或者索引都会加载到buffer进行缓存

那就会有一个lru的淘汰策略算法

常规的lru:新读取的数据都放入队首，然后将队尾的数据进行移除。

如果innodb\_buffer\_pool \_size超过1G，则会按innodb\_buffer\_pool\_chunk\_size进行分块，innodb\_buffer\_pool\_instances

### Mysql的lru算法改进：

1. 需要插入新页的话，则插入到mid\_point(3/8，old的数据的head、new数据的tail，配置参数，innodb\_old\_blocks\_pct)
2. 如果插入的数据保留时间超过1s(innodb\_old\_blocks\_time)则会将该数据移到yang的数据区(为了防止读取大量数据，导致污染了之前的分页数据)

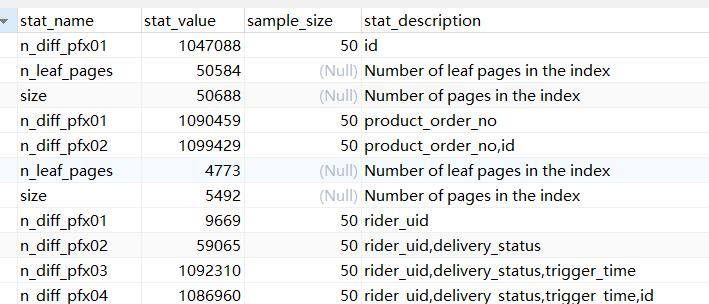
## Mysql统计信息

### 统计相关变量

Show variables like ‘%innodb\_stats\_%’;

|  |  |  |
| --- | --- | --- |
| innodb\_stats\_auto\_recalc | ON | 是否自动统计采样信息 |
| innodb\_stats\_include\_delete\_marked | OFF | 是否统计信息时，将那么delete remark也统计进去， |
| innodb\_stats\_method | nulls\_equal | Null是否可以计算一个值 |
| innodb\_stats\_on\_metadata | OFF | 是否在show table status中进行更新统计信息 |
| innodb\_stats\_persistent | ON | 是否将统计信息持久化，如果关闭，只会保存在缓存 |
| innodb\_stats\_persistent\_sample\_pages | 50 | 持久化采例数量，每x页中取出一个节点，然后(x1+…+xn)/n \* row来估算这一列的cardinality |
| innodb\_stats\_transient\_sample\_pages | 50 | 离线采例数量，每x页中取出一个节点，然后(x1+…+xn)/n \* row来估算这一列的cardinality |

### Mysql数据库中innodb\_index\_stats



可以使用show index from table\_name来查询

|  |  |
| --- | --- |
| Stat\_Name | 说明 |
| N\_leaf\_pages | 叶子节点数量 |
| Size | 总节点数量 |
| N\_diff\_pfx01 | 表示这个字段不同的值数量 |
| n\_diff\_pfx02 | 表示两个字段不同的值数量 |

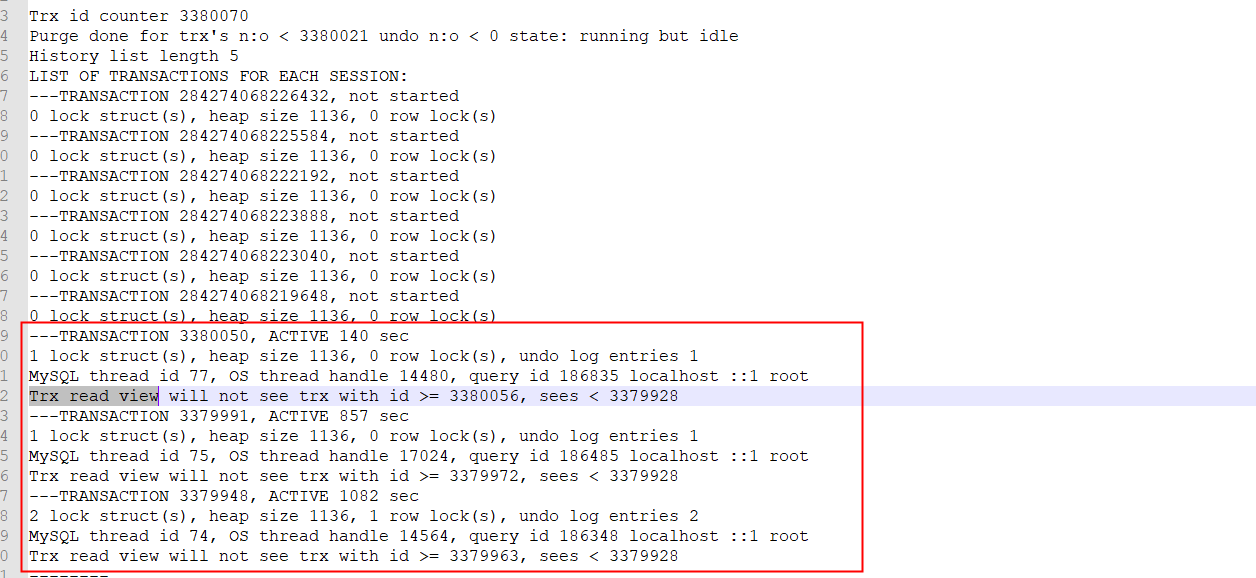
### Mysql数据库中innodb\_table\_stats

### Mysql的mvcc

每个表有data\_trx\_id(当前数据更新的txid)、delete\_bit(删除标识)、data\_ptr(undo日志中的记录)

通过select engine innodb status;

Trx read view:



每次事务启动，都会从当前已有的事务中，创建一个readview，为当前的所有事务列表，里面就有u\_tx\_id和low\_tx\_id。

Rr隔离级别

如3380050这个事务，只能u\_tx\_id就是3379928，low\_tx\_id就是3380056，那么当其他事务的提交，也就无法读取到了

### 快照读

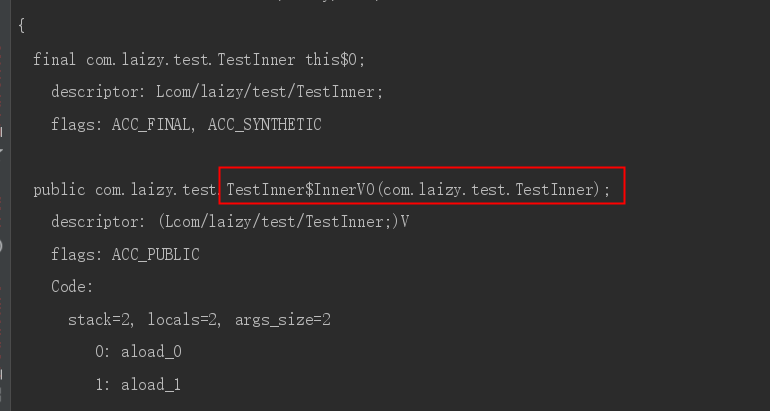
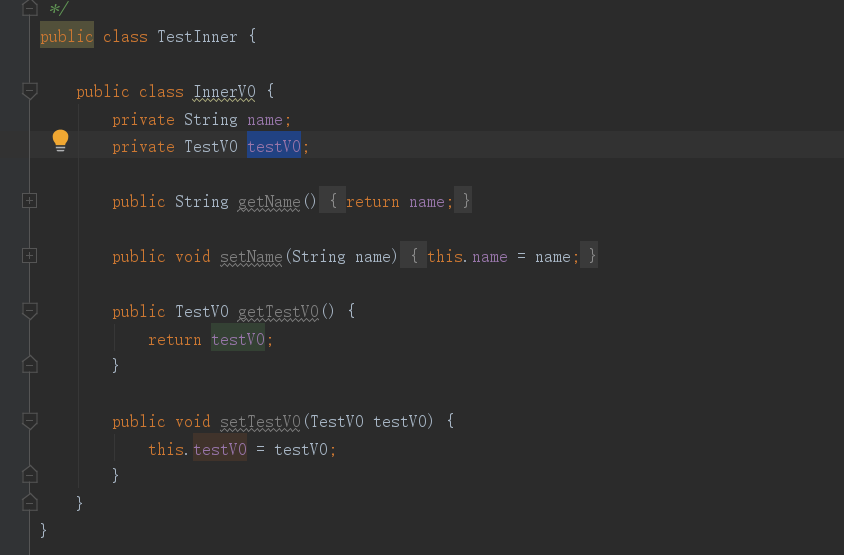
当进行select的时候，才会产生快照，也就是read view。

当rr级别隔离，如果已经产生快照读了，那么就会一直使用这个

Rc隔离级别，则是每次read都重新产生

# 内部类

在编译时，会将内部类编译成另外的类，会在本地目录，找到TestInner$InnerVO.class的类文件。然后查看其字节码，可以看到其实是构造函数加上了TestInner的实例参数。所以可以看到class.forName(“xxxx.TestInner$InnerVO”)来反射找到，然后再通过clazz.getConstruct来找到对应的构造函数，然后实例化对象



## 类加载过程

加载: 找到class文件，然后加载byte字节到内存中

验证：对字节码进行验证

准备：对类变量进行初始化，这里的初始化只是默认值

链接：将符号引用替换成直接引用(将内部的引用直接替换成句柄指针)，但是这一步不一定会在初始化之前，也有可能在初始化之后(动态类型，比如在一个类中有定义了另一个的属性)

初始化：(类变量的初始化)通过class.forName、new、静态变量获取/赋值，静态方法调用、或者加载了子类，导致需要对父类进行加载初始化，这里是对类变量进行初始化(static)。被动初始化(使用了父类的静态属性、常量、定义变量或数组变量)

使用：

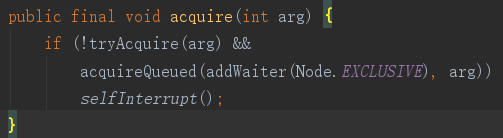
卸载：

定义变量不会触发对应的类加载操作、比如XX.class会触发加载，但是不会触发初始化。

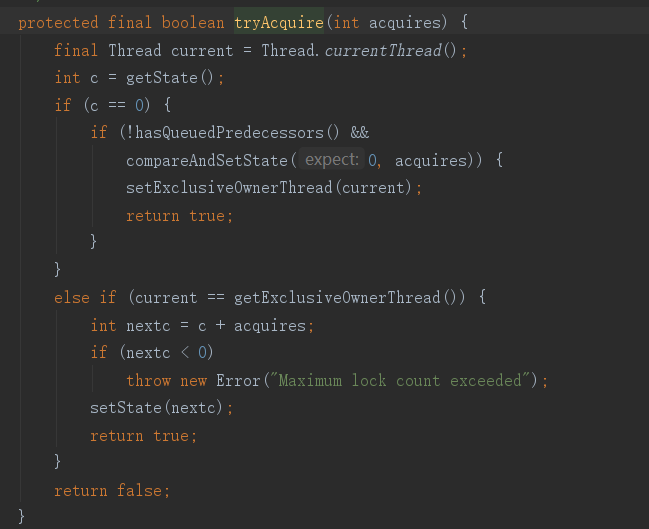
# Lock底层实现分析

都是基于AQS，也是有一个队列机制，也会根据lock进入的次序来逐渐让线程获取锁资源。Lock的非公平是指如果已有队列，还是按顺序来，但是如果当前又有新线程需要获取锁，这时候产生竞争的话，本来已经在队列的线程，并不能保证获取到锁资源，这里表现出不公平性。公平锁的效率会低，因为在tryAcquire的时候，会导致其他资源无法获取，只能等到队首获取

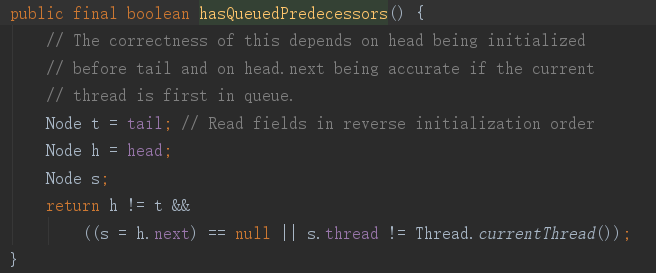
公平锁分析：acquire->tryAcquire（hasQueuePredecessors中判断是否为队首元素，只有队首元素才会成功）->acquireQueued(通过lockSupport中的park来挂起线程，addWaitor来构建队列，首次创建会挂一个空节点new Node())



获取不到锁则添加到队列中

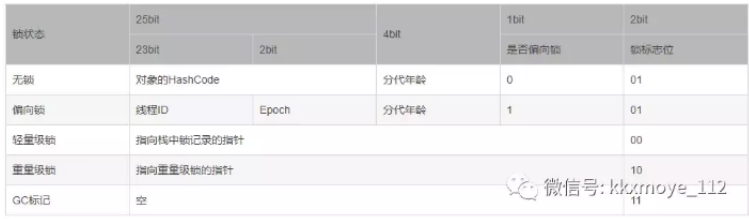


公平锁代码



判断是否为空或者head的下个元素

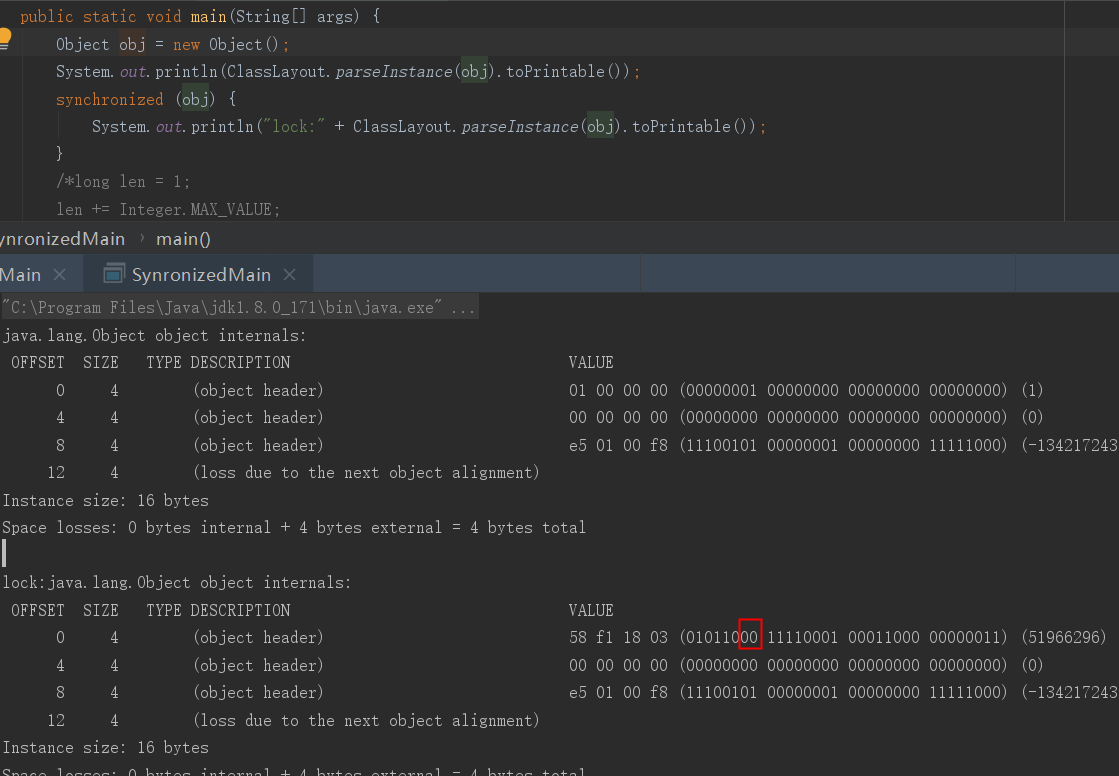
## Synchorinzed的锁升级过程，markword



1. 如果偏向锁的状态为0(biased\_lock)，代表锁未偏向，可通过usingBiaseLock来取消，jdk由于加载偏向锁，需要时间，所以默认会有个延迟时间(4000ms)。BiasedLockingStartupDelay，如果有延迟时间，默认加锁的话，也是优先使用了cas轻量级锁
   1. 已经为偏向状态，判断偏向锁的线程id是否和当前一样，如果一样，直接不用加锁。不一样则需要将锁升级为轻量级锁
   2. 不是偏向状态，通过cas来设置这个状态值，成功了则记录threadId，并执行锁代码；如果失败了，说明存在竞争，则需要将锁升级为轻量级锁
2. 轻量级锁(适合于同步代码执行很快的)
   1. 通过自旋锁(通过preBlockSpin的次数，如果超过了，升级为重量锁)，保证线程会一直循环，直接获取到锁资源
3. 重量锁（通过同步块加锁）

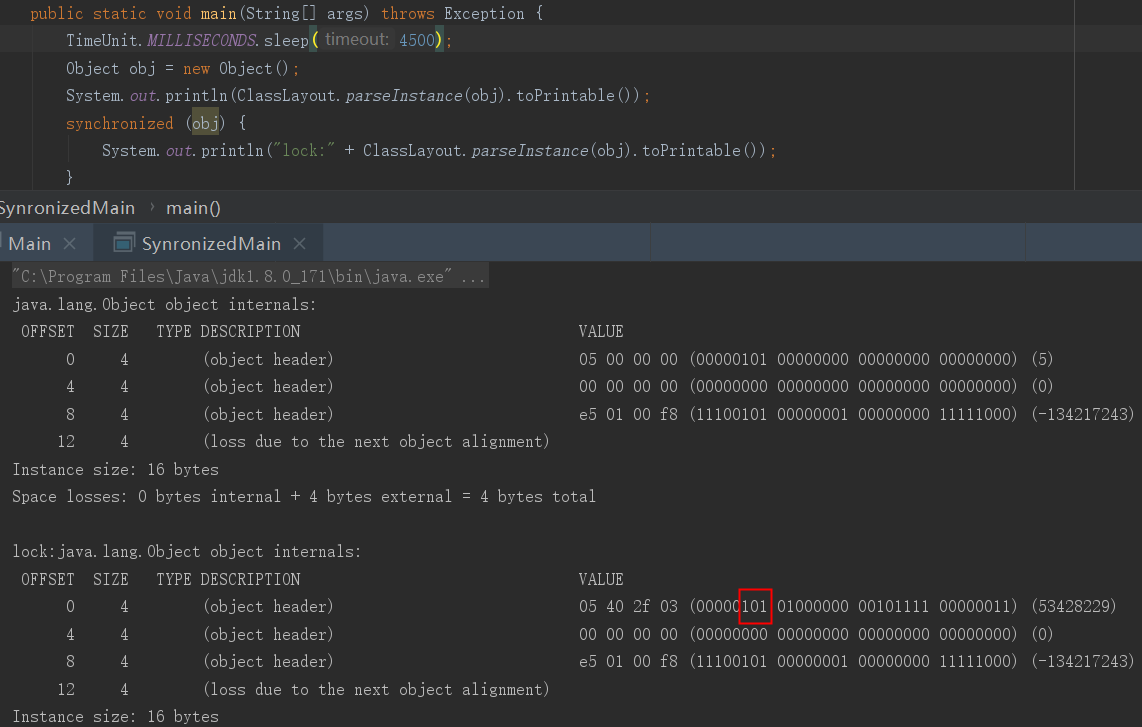
### 例子演示：

1.jvm默认参数



由上图可以看出，并没有使用偏向锁，因为偏向锁加载默认需要时间延迟(4000ms)，而是使用了轻量级锁(cas)

2.在1的例子上面，加上休眠时间



由图可以看出已经使用了偏向锁

# Springboot configuration

spring-autoconfigure-metadata.properties：@Configuration进行解析时，会读取这里的源数据，可以配置条件，相对于@Configuration的效率更高

spring-configuration-metadata.json：元数据，在编写配置文件的时候，可以提供对应的提示

Redis

1. Aof中，记录会写入缓冲区，然后写入文件(这个时候还没有同步是不会到磁盘的)，通过fsync操作，将文件内容同步到磁盘
2. Aof的rewrite误区，之前一直以为是写到rdb，其实不是，而是在到点的时候，开启了另外一个进程，然后将所有的缓存值，转换成对应的aof命令。然后如果这时候主进程继续有更新操作的话，则是会放入aof缓冲池，等到aof重写完成，则会通知主进程，主进程将aof重写缓冲区的记录再写入aof文件，然后覆盖原来的文件。
3. 如果配置appendonlyfile，那么在redis启动，会选择aof文件进行启动

# Spring Bean的实例化过程

## Condition

多个@Condition之间，是互斥，只要一个不满足，则直接跳过

ConfigurationCondition是Condition子类，并多了一个phase，定义判断的时期

@Condition主要在注册类和解析configuration类的时候

多个Condition之间，可以通过PriorityOrder、Order、或者注解@Order，那么顺序是按照优先PriorityOrder，然后通过获取实现Order的值，最后获取@Order的值

## Spring的类注册



按顺序来加载bean

1. @ComponentScan可以传入BeanNameGenetor来重新命名
2. @Import：ImportSelector、DeferImportSelector

注册到beanFactory

1. 普通的类，@Component、@Service等
2. @Bean等方法的注册通过beanFactory的方式
3. @ImportResource
4. ImportRegister的实现(通过@Import标签)

## Context refresh流程



## CreateBean的流程



## @Autowired和@resource的BeanPostProcessor



# Nacos配置中心

## 加载初始化

## 本地更新策略

## 刷新原理



由于刷新将scope的bean实例清空了，所以在get的时候，又重新触发了实例化，那么属性就重新读取赋值

或者通过@ConfigProperties，在fresh的事件中会去触发convironmentRefresh事件，从而触发对应的configProperties的刷新事件，达到属性的刷新操作

# RabbitMQ

生产者和消费者通过exchange+routeKey来找到对应的队列，

Exchange->routeKey->queue

Exchange可分为三种

1. DirectExchange，直连交换机，根据routekey绑定来找到队列
2. FanoutExchange 扇区交换机，没有routekey，只根据队列的key来绑定
3. TopicExchange，主题交换机，可以让routeKey和队列key根据一定规则绑定
   1. \*代表匹配一个单词，必须出现，且出现一次
   2. #代表通配，出现0或者多次

Spring boot 中要取消auto ack的方式，配置

spring.rabbitmq.listener.simple.acknowage-model=manual

代码分析

BlockingConsumerQueue.handleDelivery，将connection中读取到的消息放入内存队列中（本地生产者）

BlockingConsumerQueue.commitIfA…，这会根据设置的模型是否为手动，如果为自动或者不配置，则自动提交ack

BlockingConsumerQueue.rollback，这会将消息提交nack，使得消息进入死信队列中

## BasicNack和BasicReject都是拒绝消息，差别：

Nack可以多条消息，reject只能单条消息

经过了reject和nack之后，如果有requeue的话，那么message的header中的redelivered会为true

## Channel设置no ack

消息在发送之后，不管有没有被消费，都不会出现unacked的数量统计。

## Rabbitmq工作原理

生产者生产消息，

消费者消费消息，受prefetch count=1，则表示一个channel可以取多少消息，在消息没有答复的情况下



图1 消费者读取消息



图2 消息状态

## Rabbitmq源码分析



NIO类图



BIO类图

### BIO消费消息的时序图



### Nio时序图



# MyBatis

## Mapper的注册与查找过程



通过@MapperScan来注册配置

通过Configuration中的registrar来注解对应的mapper的definition

然后通过MapperFactoryBean来获取对应的实例对象

## Mapper的问题

### 什么时候进行解析

在application.properties中配置了configLocation，也会先解析configLocation中的xml

当进行实例化后，进行初始化initialBean，afterPropertySet

### Mapper中方法使用@Select，在xml中也定义了sql，那么最终由哪个生效

方法中的@注解会覆盖

### 有几种缓存，缓存是什么级别

Session级别的缓存

Namespace级别的缓存，可以通过xml(id默认为namespace，cache)或者在mapper的类中用@CacheNamespace注解，也可以通过sqlSessionFactory进行设置，然后使用@CacheNameSpaceRef或者在xml中使用<cacheRef/>

1. 如果在xml中有useCache(select的默认使用cache)，但是此时，还没有在xml定义cache，或者并没有定义cacheRef的话，那么缓存不生效
2. 如果xml即使用了cache，也使用cacheRef，那么缓存生效的是cacheRef的内容
3. 在mapper的类中，使用了@Cache，那么将覆盖xml中定义的cache
4. 同理如果在mapper的类中使用@CacheNamespaceRef，那么后续会使用该缓存

### Insert/update/delete怎么清除缓存，默认清除缓存

在<update>、<insert>、<delete>中配置flushCache为true的属性

Mybatis采用了委托设计模式，会新增一个本地缓存TransactionCacheManage



由于使用了本地缓存，那么必然有分布式的问题。

### 缓存的更新与事务的更新是怎么的顺序



在事务真正commit之前去更新缓存

### Session、connection、mapper、executor、transaction之间的关系

1个事务默认是开启一个session(可以多个session)

1个session对应一个executor

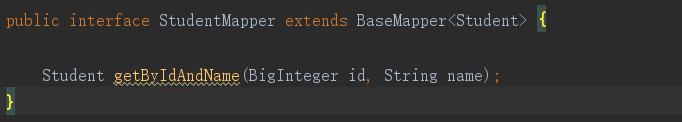
1个session对应n个mapper

## Mapper执行的解析



## 参数解析类





如以上的方法，可以在sql使用，#{param1}、#{param2}参数，或者#{id}、#{name}来使用

## 执行查询语句



这里使用了两种缓存；

1. 基于mapper的缓存，在mapper.xml中cache定义
2. 基于session的缓存，默认启动

### CacheKey的规则

1. id,类名+method，如com.tt.test.dao.mapper.StudentMapper.getByIdAndName
2. offset，起始位置，0
3. limit，默认integer的最大值
4. sql
5. 参数1
6. 参数2
7. …

### 拦截器



目前有的几个拦截器：

statementHandler：

IllegalSQLInterceptor：sql条件

PaginationInterceptor：分页

PerformanceInterceptor：sql性能

Excetor：

OptimisticLockerInterceptor：锁

SqlExplainInterceptor：防止全表更新或者删除

--需要配合BlockAttackSqlParser

### Executor类图



### ParameterHandler、ResultSetHandler



### StatementHandler



### 数据库连接



# Mongodb

## Journal

## 引擎

### MMAPV1

只能集合锁

### wiredTiger

可以文档锁