## 集合

Collection

ArrayList、LinkedList、Vector（线程安全）

ArrayList：底层是数组，初始化时，数据量是0，当add时候，默认变成10，扩容机制是上一次容量的1.5倍（新容量=旧容量+旧容量右移1位），但扩容后还是会不满足所需容量，则直接用所需容量当做新容量，最大容量是Integer最大值 - 8，特性查询比较快（通过脚标去查），删除效率低

LinkedList：底层是头结点和尾节点的双向链表，提供了2个方法，头插法LinkedFirst，尾插法LinkedLast，特性非常适用于经常增加和删除的场景，查询效率慢(他会对各个节点进行比较得到查询数据)

Vector：线程安全，底层和arrayList一样是数组，和ArrayList的区别在于，大部分方法都是被synchronized关键字所修饰，扩容机制是上一次容量的2倍

ArrayList查询快插入慢，LinkedList查询慢插入快，那么ArrayList插入一定比LinkedList慢吗？

不一定，如果是定长ArrayList的情况下，会比LinkedList快，ArrayList慢在扩容的时候拷贝数据。

### ArrayList

public class ArrayList<E> extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

RandomAccess：标记接口，用于判断for循环还是用iterator来循环。实现该类支持随机访问，下标访问。

Cloneable：克隆

Serializable：序列化

serialVersionUID：版本号，用于序列化和反序列化。反序列化将二进制转成对象时，会校验版本号，防止反序列化修改类元信息。版本号不匹配，会报错java.io.InvalidClassException，反序列化失败。

浅拷贝：基本类型变，引用类型不变，对象的指针指向同一个地址，内存地址相同。

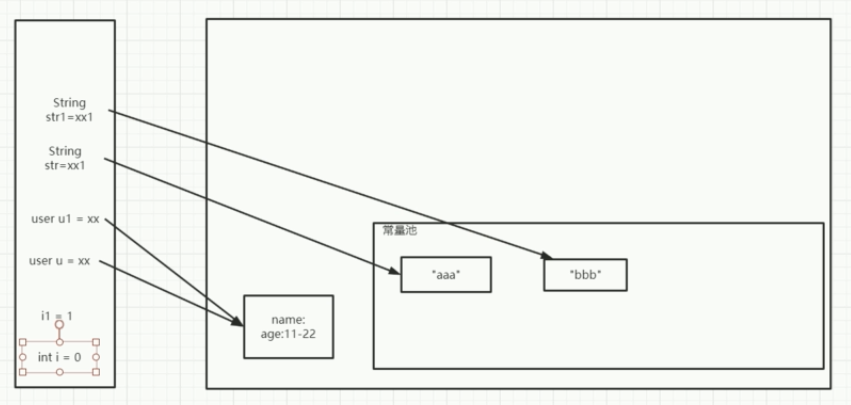
深拷贝：拷贝的对象互补干扰，不管是基本类型还是应用类型。

基本数据类型传递、String类型传递、引用类型传递区别？

基本数据类型传递：在栈中拷贝一份基本数据类型，并更改值

String类型传递：在栈中拷贝一份，然后在常量池中开辟新的值，然后将拷贝的String类型地址指向常量池新开辟的区域

引用类型传递：在栈中拷贝一份引用类型数据，新旧引用类型地址指向堆中同一块区域



不可添加集合：Collections.unmodifiableList(list) 只读，不能添加、修改（会报错），他只受list的变化而变化。一般用来只想框架自己使用，不想让外部代码修改。比如框架配置信息，框架自己可以操作list，提供外部是只读list

### LinkedList

双向链表，插入的数据量越大，LinkedList插入速度不一定比ArrayList要快，ArrayList慢在扩容，而LinkedList慢在创建节点，但LinkedList按照下标插入绝对比ArrayList快。

Iterator遍历比for循环遍历要快，因为for循环每次get都需要比对数据，而Iterator使用游标进行数据的查询，每次next之后，游标会+1

### Vector

Vector是不是线程安全的，取决于开发者是如何使用。

线程安全场景：线程交替执行（所有操作方法都被同步），或者全局锁情况下

线程不安全场景：没有实现全局锁，线程1增加数据，线程2删除数据，此时就是线程不安全的，因为Vector的Synchronized修饰方法，他只能保证方法只能被一个线程占用。

### set

HashSet

### map

HashMap、ConcurrentHashMap线程安全、HashTable线程安全

hashmap：

1.7版本--底层是数组+单链表

1.8版本--数组+单链表+红黑树

1、单链表和红黑色之间的转换，单链表长度>=8，并且哈希桶数组长度>=64，会将单链表转换成红黑树形式进行存储，红黑树节点数量<=6，会重新转换成单链表

2、hash桶数组默认是16个，阈值默认是0.75，当插入16\*0.75=12个元素的时候(简单理解为，已经有12个数组被占用)，就会触发扩容

3、扩容机制是上一次容量的2倍，然后重新进行新的hash运算(hash值&（length -1）),填充到新的hash桶数组

4、链表扩容的数据迁移是将相同下标的节点组合在一起，放到新数组里面；红黑色是直接转移根节点

### ConcurrentHashMap

线程安全，实现分段锁表

1.7版本--底层分片数组，为了保障线程安全，用到了Segment锁，他继承于ReentrantLock

1.8版本--数组+单链表+红黑树，使用synchronized+CAS操作来保证线程安全，里面有2把锁，第一把锁在数据插入时，防止插入时数据并发，第二把锁在链表树化，防止红黑色里面根节点旋转导致根节点转移。

### HashTable

线程安全，全局锁表

## CAS

1、compare and swap的缩写，中文翻译成比较并替换

2、CAS(内存值V，旧的预期值A，要修改的新值B)

3、相当于轻量级的加锁过程，在并发量不是特别大的情况下，当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做

CAS存在问题：

1、CPU开销大；自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销

2、只能保证一个共享变量原子操作；对多个共享变量操作时，循环CAS就无法保证操作的原子性

3、ABA-标志位时间戳；如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

## synchronized

synchronized关键字在编译之后会在同步代码块前后加入2个指令：

1、monitorenter(锁)

2、monitorexit(解锁)

### 偏向锁加锁：

1、将线程id写入到锁对象的对象头中

2、其他线程进入并获取锁之后，锁对象会结束偏向状态，膨胀成轻量锁或重量锁，有资源竞争重量锁，无资源竞争轻量锁

### 轻量级锁加锁：

1、执行monitorenter(锁)

2、新线程进来会在虚拟栈中为当前线程开辟一个空间叫Lock Record

3、将锁对象的Mark Word写入里面

4、CAS尝试将锁对象的Mark Word轻量级锁指针指向当前Lock Record（作用是让其他线程知道，该object monitor已被占用），成功获取锁，失败则挂起线程

5、然后lock record里的owner指针指向object mark word（作用是为了在接下里的运行过程中，识别哪个对象被锁住了）

6、执行monitorexit(解锁)

### 重量级锁同步代码块加锁：synchronized(o)

1、执行monitorenter(锁)

2、偏向锁膨胀成重量锁的过程，偏向锁会先进行撤销偏向成无锁01，此时锁对象的监视器锁monitor为0

3、新的线程获取锁，锁对象的monitor监视器锁会+1（锁支持重入），当前线程成为monitor的owner（所有者），并将monitor的指针写入到锁对象头

4、修改锁对象标志位为10，此时锁就膨胀成重量锁

5、若其他线程已经占有monitor的所有权，那么当前尝试获取monitor的所有权的线程会被阻塞，直到monitor的进入数变为0，才能重新尝试获取monitor的所有权。

6、执行monitorexit(解锁)

7、同步代码块执行完，加了几次锁释放几次，将monitor清零，完成释放锁

### 重量级锁同步方法加锁：public synchronized void test()

JVM使用ACC\_SYNCHRONIZED标识来实现，即JVM通过在方法访问标识符(flags)中加入ACC\_SYNCHRONIZED来实现同步功能。

## ReentrantLock加锁解锁过程

ReentrantLock lock = new ReentrantLock(true);

为true就是公平锁：那个线程等待的时间长就用哪个线程，获取队头线程

为空就是不公平锁：线程抢占资源

基本实现原理：park（等待）+自旋

lock()：一朝排队，永远排队；

公平锁原理：当锁处于自由状态的情况下，有新的线程竞争到锁，他会看队列中有没有排队，没有线程排队，则自己获取到锁，有排队，自己就排到队尾；

非公平锁原理：当锁处于自由状态的情况下，新的线程会通过cas尝试加锁，加锁失败就进入队列

加锁过程：

（1）第一个线程获取到锁，成为当前锁的owner

（2）第二个线程进来，会先创建一个空节点，第二个节点指向空节点，空节点指向自己，然后第二个线程会尝试加锁，加锁成功自己成为队头（此时入队列acquireQueued方法才会有返回值，因为线程被唤醒了），加锁失败，将上一个线程的waitStatus由0修改为-1，因为死循环会在次判断一次，同样竞争失败，然后判断上一个线程的waitStatus为-1，自己就进入队列并park（此时入队列acquireQueued方法不会有返回值，因为线程处于睡眠状态，不能有返回值让线程继续运行）

（3）其余的线程都会先经过第二步然后都park

简单理解为队头获取到锁，第二个线程会进行这几步操作（尝试加锁-失败-入队-判断如果上一个节点在park，那么自己park，如果不是park，则自己竞争锁），第一个线程执行完释放锁，队列中的下一个线程就会尝试加锁。

队列：队列中默认队头的线程、上个节点prev、下个节点next都为空，下一个队列的prev指向队头的next；只有队尾waitStatus为0，其他都是-1

解锁过程：

（1）将自己线程的waitStatus的-1改为0

（2）唤醒队列中的下一个线程

## volatile关键字

CPU和内存之间的线程效率是差好多数量级的，为了保证内存不影响CPU的计算，中间会有好多LLV缓存，我们线程在这个缓存中去工作，首先他取数据会在主内存取到工作内存中去计算，计算完之后在传给主内存，此时就出现了多线程间变量的可见性问题。java为了解决这种问题，他就抽象出了JMM这种模型，被volatile修饰的变量，保证变量能在多线程之间的可见性，也就是变量被更新了，能立刻刷新到主内存中，其他线程能立刻取到最新的变量数据，这是总线嗅探机制，但不保证原子性，简单的说就是volatile保证线程修改了变量能及时被其他线程看到，但不保证多个线程改变同一个变量之后数据的完整。

如何保障数据原子性：使用synchronized关键字、ReentrantLock加锁、AtomicXXX类（多个AtomicXXX类连续调用不能构成原子性）来保证数据原子性。

Volatile可见性：线程修改的变量能立即被其他线程。

Volatile有序性：防止指令重排，不让JVM加载代码使为了优化代码而改变指令顺序。

### 总线嗅探机制

每个处理器通过嗅探在总线上传播的数据来检查自己的缓存值是不是过期了，如果处理器发现自己缓存行对应的内存地址呗修改，就会将当前处理器的缓存行设置无效状态

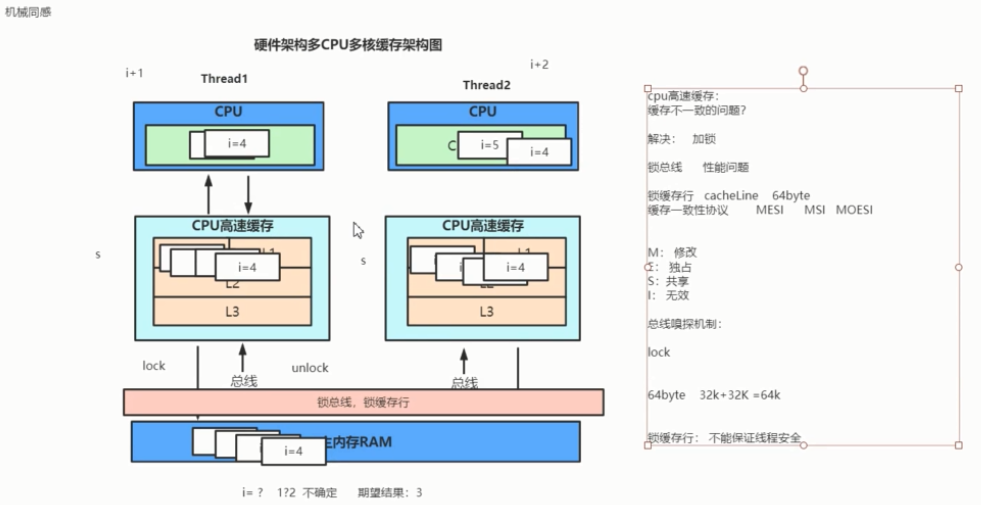
### 总线风暴

由于volatile的mesi缓存一致性协议需要不断的从主内存嗅探和cas不断循环无效交互导致总线带宽达到峰值解决办法：部分volatile和cas使用synchronize

mesi缓存一致性协议：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

资料：https://zhuanlan.zhihu.com/p/137193948

### MESI缓存一致性协议



Modified：修改

Exclusive：独占

Shared：共享

Invalid：无效

由于早期实现线程之间的数据一致性是使用总线锁，但这样会导致线程串行执行，效率低下，后面锁缓存行出现，才有MESI的诞生。

#### 执行过程

（1）线程1获取到a，放到自己的CPU高速缓存中，线程1的a此时处于独占状态

（2）线程2获取到a，放到自己的CPU高速缓存中，线程1和线程2的a此时处于共享状态

（3）线程1操作a，此时线程1的a处于修改状态，线程2的a处于无效状态

（4）线程1完成a的操作，快速回写到主内存中，线程2的a由于是无效状态，重新在主内存中获取a

#### 缺点

无论数据如何快速的回写到主内存中，都会存在性能的损耗，获取到失效状态的线程都必须会重新在主内存中获取数据

### 伪共享

Cache Line大小是64Byte。

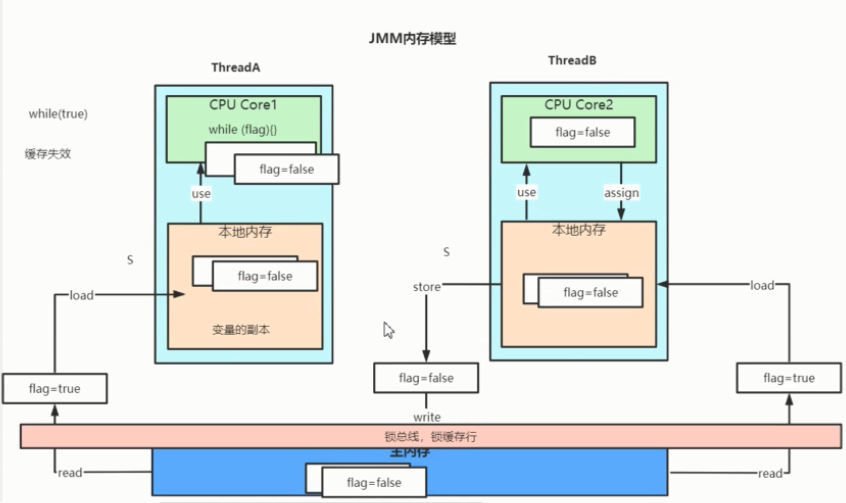
如果多个核的线程在操作同一个缓存行中的不同变量数据，那么就会出现频繁的缓存失效，即使在代码层面看这两个线程操作的数据之间完全没有关系。这种不合理的资源竞争情况就是伪共享。

比如：线程1操作类属性a，线程2操作类属性b，但a和b在同一个缓存行中。如果按照MESI缓存一致性协议流程来操作的话，会频繁导致缓存失效。

JDK8以前在a和b之间增加属性，将类属性达到64Byte，使b在下一个缓存行中，来达到避免伪共享效果。

JDK8使用@Contended + jvm参数 –XX:-RestrictContended 来避免伪共享

## JMM内存模型



JMM内存模型Java Memory Model是为了屏蔽各种硬件和操作系统对内存访问的差异性，简单理解就是为了保证程序在不同的操作系统和硬件，执行的结果是一样的。

Java的一切可见性问题都是因为编译器优化、内存优化、指令优化导致，硬件的底层已经保证了可见性，可以理解Java不存在可见性问题（伪可见性）。

https://hllvm-group.iteye.com/group/topic/34932#post-232535

### JMM的happends-before原则

1、如果操作A和操作B属于同一个线程，那么说明操作A的操作对操作B可见

2、unlock hb lock 线程A的unlock操作，在线程B的lock操作前操作

3、如果操作A是对volatile变量的写操作，操作B是对同一个变量的读操作，那么操作Ahappends-before操作B

4、happends-before具有传递性，操作A hb操作B，操作B hb操作C，操作A hb操作C

## SQL调优

### Sql优化

1、首先考虑where及order条件建立索引

2、少使用is null处理，用=0处理

3、少使用!= <>判断

4、少使用or做条件，用union all做数据合并

5、少使用in/not in，能使用betwen就使用

6、少使用%在前的模糊查询

7、少使用对字段进行表达式操作num/2=100 --> num=100\*2

8、少使用函数

9、不要写一些没意义的查询，比如1=1

10、用exists代替in

11、索引不是越多越好，虽然查询快，但是插入和修改会降低

12、尽量使用数字型字段，因为字符需要对每个字段进行比较

13、尽量用varchar代替char

14、少使用\*，用具体字段和删除临时表

16、大数据备份用select into 新表名 from 旧表名代替 create table

17、数据超过1万条数据，考虑改写脚本

### 索引选择

1. 没主键，需要创建聚族索引
2. 有主键，查询还是慢，创建非聚族索引
3. 非聚族索引规则：
4. 尽量使用where、order by、group by后面的字段来创建
5. 查询最频繁的一列放在左边
6. 查询重复值最少的放在左边
7. 组合索引的字段尽量不要超过3个

Sql调优最基本的是表要有主键，因为有主键的表MySQL会创建聚族索引，聚族索引的好处是它的主键和数据行是在一行的，在explain查询语句的时候会发现，它的type级别是const，这是很高的一种级别。当有主键的时候，如果一条SQL语句很慢，可以去查看他是否创建了相应的索引，建立索引要尽量选择where、order by、group by后面的字段或者join in连接的字段作为索引列，这些索引列也要排序，要符合最左匹配原则，你选的时候要根据他们的索引选择器，你的非重复的数据行和重复的数据行中间的排列，大的放左，小的放右这样的形式，这种多列的索引列，要去建立联合索引而非单个索引。

如果数据查询的非常频繁，可以考虑给它使用覆盖索引，覆盖索引是可以直接在索引中查询到数据，相对来说还是很快的。

聚族索引（数据需要确保唯一）：它确切来说应该是一种数据结构，它是将它的主键和它的数据行放在一块的，找到了索引就找到了数据。

非聚族索引（数据不需要确保唯一）：就是我们说的普通索引，他的索引存放的是叶子节点，叶子节点包含他的主键、索引列、事务ID等信息，它在查询的时候，其实是通过它的索引列查询它的主键，在通过主键去回表查询，还是通过聚族索引查的，这样的速度会相对慢点

### 事务

脏读：一个事务读取了另一个事务改写但未提交的数据，如果这些数据被回滚，则读到的数据时无效的。

不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。在事务1中，第一次读到了A记录数据，中间事务2修改了A记录数据，第二次读到的A记录数据和第一次读到的A记录数据不一致。

幻读：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发生有些原来没有的记录。事务1修改所有A表数据，事务2增加了A表数据，事务1的用户就会发现A表还有未修改的数据。

InnoDB的特性就是它支持事务，支持事务的时候，为了保证事务的并发度，它提供了一个隔离级别，有4种隔离级别：

1、Read Uncommitted：结果什么都不能保证

2、Read Committed：解决事务之间的脏读，但是不能保证不可重复读和幻读

3、Repeatable Read：默认事务，保证不发生脏读、不可重复读，不保证幻读

4、Serializable：事务最高级别，串行化，一个个事务串行排列。事务一个接一个执行，等待前一个事务执行完，后面的事务才可以顺序执行，保证不发生脏读、不可重复读、幻读。

## 线程池的运行机制

线程池他是有一个核心线程数，当线程运行的时候，如果没有设置成预启动加载，首发线程数为0，当提交一个新任务的时候，他会首先建立一个核心线程去执行任务，如果此时一直来任务，之前线程没有执行完，那么就会一直创建核心线程，当达到最大核心线程数时，如果都还在忙，那么此时就会放到BlockingQueue里面作为节点，如果BlockingQueue队列也放满了，而却核心线程都还在忙，那就会去建立新线程，它叫做非核心线程，若一直创建，数量达到非核心线程数max access，此时就会触发一个拒绝策略。

线程构造，在创建线程池的时候，需要提供线程的threadFactory，一定要指定他名称，并且可以设置为守护线程，其他线程执行完，主线程也执行完，JVM才会停止运行。

### JDK内置了四种拒绝策略

1. AbortPolicy，直接抛出异常，这是默认策略
2. DiscardPolicy，悄无声息的丢弃任务
3. DiscardOldestPolicy，丢弃最早的未执行任务
4. CallerRunsPolicy，用调用者所在的线程来执行任务，这种方式会影响新任务提交速度

### 阻塞队列

ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列，使用的是一把锁，性能会慢。

LinkedBlockingQueue：一个由链表结构组成的无界阻塞队列，也可以理解为有界队列，因为他的默认容量是Integer.MAX\_VALUE。使用的是两把锁，入队和出队互相不阻塞，性能相对ArrayBlockingQueue会高。

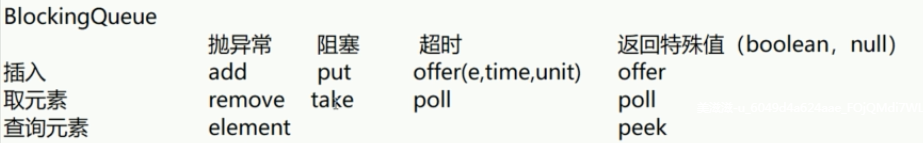
PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列，默认容量11。

DelayQueue：一个支持延时获取元素的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列，生产者线程put，必须要有消费者线程take，不然生产者/消费者线程都会发生阻塞，常用于线程间安全的交换单一元素，优势在于轻量级。

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。



### 线程生命周期

新建(New)：使用new关键字创建了一个线程之后

就绪（Runnable）：象调用了start()方法之后

运行（Running）：就绪状态的线程获得了CPU，开始执行run()方法的线程执行体

阻塞 (Blocked)：线程因为某种原因放弃了cpu使用权

死亡(Dead)：线程结束后就是死亡状态。正常结束、程序报错异常结束、调用stop结束（该方法通常容易导致死锁）

## Spring

### Spring AOP代理方式

1、如果代理类有接口，走的就是JDK的动态代理

2、如果没有，则使用cglib字节码技术去创建代理对象

JDK动态代理：主要是使用了Java反射中的两个类，一个是Proxy，一个是InvocationHandler，它是通过Proxy.newProxyInstance来创建代理对象，然后通过反射invoke去执行那个方法。

### Spring事务隔离级别和传播行为

Spring提供了一个注解，叫做Transactional，并且提供了事务的7个传播行为和5个隔离级别。

Spring事务5个隔离级别

Spring事务隔离级别只比数据库多1个默认的Default，数据库使用什么默认事务隔离级别Spring就用什么，Mysql默认采用REPEATABLE\_READ隔离级别，Oracle默认采用READ\_COMMITTED隔离级别。

Spring事务7个传播行为

事务传播行为（propagation behavior）指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行，解决业务层方法之间的相互调用的问题。

案例：service1、service2有事务。同时调用service1的一个方法和service2的一个方法，才能完成一个业务。

1、Propagation.REQUIRED（默认）

如果当前存在事务，则加入该事务，如果当前不存在事务，则创建一个新的事务。

2、Propagation.SUPPORTS

如果当前存在事务，则加入该事务；如果当前不存在事务，则不使用事务。

3、Propagation.MANDATORY

如果当前存在事务，则加入该事务；如果当前不存在事务，则抛出异常。

4、Propagation.REQUIRES\_NEW

重新创建一个新的事务，如果当前存在事务，延缓当前的事务。

5、Propagation.NOT\_SUPPORTED

以非事务的方式运行，如果当前存在事务，暂停当前的事务。

6、Propagation.NEVER

以非事务的方式运行，如果当前存在事务，则抛出异常。

7、Propagation.NESTED

如果没有，就新建一个事务；如果有，就在当前事务中嵌套其他事务。

### Spring生命周期

粗略的可以分成三个阶段，他有Bean的一次加工、属性设置、Bean的二次加工过程。

第一步：Bean的一次加工，实例化对象，在实例化前后，会用不同的BeanPostProcessor做前置和后置处理。

第二步：依赖注入过程，首先会设置Bean的属性值，然后处理不同的BeanAware（比如：BeanNameAware、ApplicationContextAware），给Bean注入相应的aware属性，这个阶段主要是分为这2步，这2步前后也有不同的BeanPostProcessor后置处理器进行处理。第二步完成之后，Bean已经接近完成期。

第三部：Bean的二次加工前后，可以像ApplicatioinContext一样配置一个自定义的后置处理Bean来处理，在这期间主要是调用了Bean中写的init方法或者Bean实现了InitializingBean，他会回调afterPropertiesSet方法，来对bean做进一步的设置。二次加工完成之后，Bean就已经建立完成，这时候它会根据Bean的scope来对Bean进行单例还是原型处理，默认是单例，如果是单例，生命周期还是由Spring来管理，如果是原型（scope为prototype），就会转交给调用者去管理它后续的生命周期。当Spring容器销毁的时候，它会调用destory相应的方法，或者开发者自己实现了DisposableBean接口，Spring就会调用开发者自己销毁的逻辑。

### BeanFactory和ApplicationContext区别

ApplicationContext是BeanFactory的子接口。

BeanFactory：BeanFactory负责配置、创建、管理Bean。Spring容器最基本的接口就是BeanFactory，他是Spring bean容器的根接口，提供获取bean，是否包含bean,是否单例与原型，获取bean类型，bean 别名的方法。它最主要的方法就是getBean(String beanName)。

ApplicationContext：也被称为Spring上下文，容器同时还管理着Bean和Bean之间的依赖关系。比如：

AnnotationConfigApplicationContext从java注解的配置中加载配置到容器。

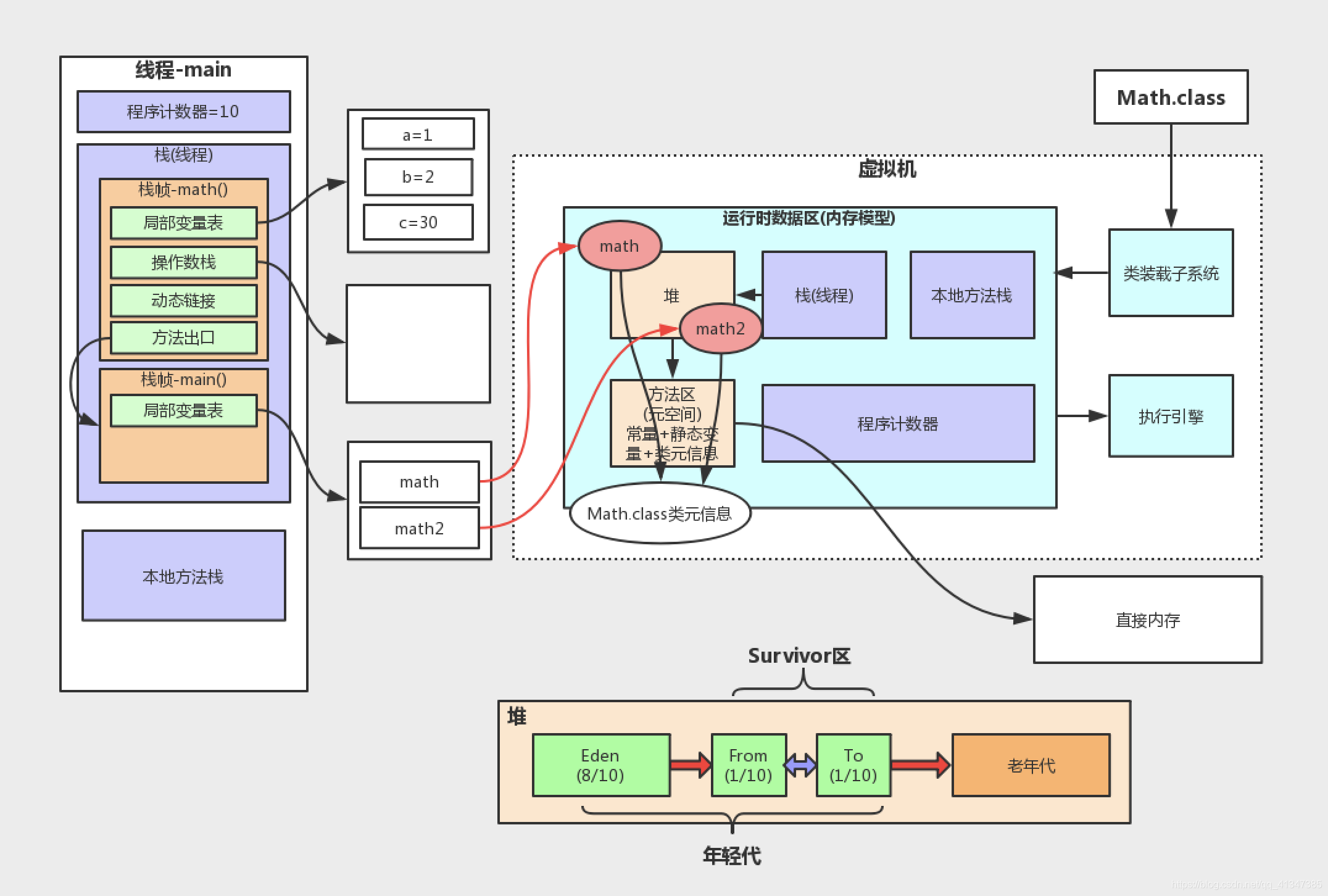
ClassPathXmlApplicationContext加载项目中的spring配置文件到容器。

FileSystemXmlApplicationContext加载系统盘中的配置文件到容器。

## JVM内存模型

JVM有很多，hotspot、jRockit、j9等

JVM运行时的内存区域可以分为两大类，一个是线程的私有区，另外一个是线程共享区。



### 线程私有区

程序计数器：这里在JVM中栈内存比较小的一个地方，同时也是唯一一块不会OOM的区域，这里会告诉你当前线程执行的代码归于哪一行；

虚拟机栈：虚拟机栈主要使用的是栈帧，每一个方法从调用直到执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程，方法程序结束栈内存全部释放，用于存储局部变量、操作数栈、动态链接、方法出口等信息。操作数栈和局部变量关系：比如int a=1程序先将常量1入栈到局部变量表，将常量1出栈后存入局部变量a。如果有常量的计算则会将局部变量值装载到操作数栈，局部变量还在局部变量表中，在操作数栈中计算后，将常量出栈存到局部变量表中。

本地方法栈：与JVM虚拟机的native方法有关，就是JVM本地方法库

局部变量和操作数栈关系

1. 将常量1放到栈顶，从栈顶中将常量数据压栈，压入操作数栈
2. 将常量值1存入到局部变量int a
3. 在局部变量表中将int a的常量值取出，同时从局部变量表中拿到其他数据，计算出结果
4. 将结果数据放到栈顶，然后将常量值压栈
5. 将计算结果常量值存入到局部变量

动态链接: 每个栈帧都保存了一个可以指向当前方法所在类的运行时常量池, 目的是: 当前方法中如果需要调用其他方法的时候, 能够从运行时常量池中找到对应的符号引用, 然后将符号引用转换为直接引用,然后就能直接调用对应方法, 这就是动态链接。简单理解就是方法会有个东西指向运行时常量池中的符号，这个东西就是动态链接。

方法出口：记录当本方法执行完之后，回到主方法时改从哪一行执行。

### 线程共享区

方法区：即我们常说的永久代，方法区用于存储被JVM加载的类信息、常量、静态变量、以及编译器编译后的代码（所以当方法再次被调用时，可以通过堆区对象头信息的Klass Pointer类型指针直接找到类元信息，从而执行类方法的代码）等数据，简单来说所有定义方法的信息都保存在这里。在Java8中，永久代已经被移除，被一个称为元空间的区域所取代（hotspot和jRockit合并推出的元空间）。元空间的本质和永久代类似，元空间与永久代之间大的区别在于：元空间并不在虚拟机中，而是使用本地内存，也叫堆外内存。因此，默认情况下，元空间的大小仅受本地内存限制。

堆：它里面存放是java里产生的对象、对象的实例、数组等信息，它也是GC重点回收的一个区域，也会产生OOM，由于现代虚拟机有采用分代收集算法, 因此Java堆从GC的角度还可以细分为：新生代和老年代比例3：2。新生代又分为Eden区、From survivor区、To survivor区。比例8：1：1。新生代被回收15次之后还存活的对象就转移到老年代中。

运行常量池：是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池。

栈、堆、方法区之间的关系

栈中的对象地址指向堆，堆中的对象是通过方法区获取到类信息，然后new出来的。

Int a = 0; //堆区

Static int b = 1; //常量池

String c = "3"; //常量池

User u = new User();//堆区

Public void m(){

Int d = 4;//栈的局部变量表

String e = "5"//栈的局部变量表

}

## 垃圾回收器

CMS：它可以和用户线程并发操作，就是GC线程与用户线程并发执行的一款垃圾回收，特点就是低延迟，它的整个过程主要是用来回收老年代，算法使用的是标记清除和标记整理，它的过程有四个阶段：1、初始标记 2、并发标记 3、重新标记 4、并发清理；但它在初始标记和重新标记的时候，它不是线程并行，它会短暂的产生Stop The World（用户线程必须暂停），但这个时间很短。它的并发标记和并发清理，这2个阶段的耗时比较长，但它是和用户线程一起并发执行的，所以说它可以最短的实现低延迟。

### 垃圾回收算法

1. 第一种是标记清除，标记一些不可达的对象，比如说将判定为死亡的对象，然后统一回收掉，它因为这种使用可能会造成很多的内存碎片，并且当它对象是比较大的时候，它的标记效率是比较低的
2. 第二种是标记复制，标记复制就是将可用内存按容量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内存上面，然后再把已使用过的内存空间一次清理掉。

缺点是可一次性分配的最大内存缩小了一半。优点是不会产生内存碎片。

1. 第三种是标记整理，将所有存活的对象都移动到一个规整且连续的内存空间中，将已经判定为死亡的对象往另一端移动、复制，然后执行垃圾回收器回收无用对象所占用的内存空间
2. 第四种是分代收集，根据对象存活周期的不同将内存划分为几块。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。

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。而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”或者“标记—整理”算法来进行回收

### 怎么判断对象是否可达

1. 第一种是引用计数，简单说给对象中添加一个引用计数器，每当有一个地方引用他时，计数器值就+1,；当引用失效时，计数器值就-1；任何时刻计数器为0的对象就是不可能在被使用，那么这个对象就是可回收对象。它的缺点是可能产生循环依赖。
2. 第二种可达性分析，就是根据java中可以作为GC Roots根节点来开个链，如果不在这个链上，则称该对象是不可达的，要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

### Java中可以作为GC Roots的对象

1. 虚拟机栈中引用的一些对象
2. 方法区中静态变量所引用的对象
3. 方法区中常量所引用的对象
4. 本地方法栈中JNI（即一般说的native方法）中引用的对象

## Linux的CPU突然飙高排查

1. top找到异常进程
2. ps–ef|grep java 找到异常进程，就可以找到对应的应用
3. top -H -p 进程号查看异常线程
4. printf "%x\n" 线程号将异常线程号转化为16进制
5. jstack进程号|grep 16进制异常线程号 -A90来定位异常代码的位置（最后的-A90是日志行数，也可以输出为文本文件或使用其他数字）。可以看到异常代码的位置。

如果提示 -bash: jstack: command not found，则可以找到jdk的目录，进入bin目录，在执行 ./jstack进程号 | grep 16进制异常线程号 -A90 即可。

## Dubbo

RPC协议：通过不同的技术进行远程方法调用，比如可以通过HTTP发送请求、TCP发送请求、Netty发送请求。

Dubbo协议：通过Netty进行远程方法调用。

多个服务B将服务往ZK进行注册，服务A通过注册中心获取到多个服务B的地址，然后缓存到本地，最后通过Dubbo智能负载均衡技术，进行远程服务B的调用。

Zookeeper注册中心的优势：

1. Zookeeper的数据是存放在内存中的，取数据快。
2. 服务B挂掉了，可以使用zk的临时节点，这样就不影响服务A的调用
3. 新增服务B，因为zk的watch机制，服务A就可以知道服务B的地址有改动，使服务A的缓存和注册中心的地址数据是实时一致的。

### Dubbo服务暴露源码流程

在使用XML方式配置标签的情况下，首先它有个dubbo的DubboBeanDefinitionParser终端解析器去解析标签，像application、registry、protocol、consumer的一些配置，会相应的解析成对应的config对象，但是有个特殊情况是暴露的服务不会写到对应的service标签里，它不是直接解析成ServiceConfig，而是会解析成ServiceConfig下面的一个子类叫做ServiceBean。ServiceBean实现了2个接口，第一个是实现了InitializingBean，到时候Spring会回调afterPropertiesSet方法，第二个是实现了ApplicatioinListener，使用这种观察者模式去监听ContextRefreshEvent，能够去刷新afterPropertiesSet的回调，它会等Spring容器处理完毕后，回调onApplicationEvent方法，但这个里面会进行两个服务的暴露，区别是延迟走afterPropertiesSet方法，不延迟走onApplicationEvent方法。它里面会调用一个暴露方法export，其实这个暴露方法在他的父类ServiceConfig里面比较核心的，它会多次export，然后进行下一次循环调用doExportUrlsForProtocol，这个flow里面有个dubbo核心spi机制，它是跟JDK还是不太一样的，以kv的结构进行存储，可以通过实例化加载某一个类对应所需要的服务。接着protocol，它会调用它的export的一个方法，但是这Protocol它相关的两个实现，第一个是RegistryProtocol，第二个是dubboProtocol，它在Registry的暴露中第一步就是doLocalExport，就是本地暴露，在这个过程中调用dubboProtocol里面将里面的实现全部封装成invoker，然后到dubboProtocol会将invoke封装成一个export，然后将这个export通过netty去暴露出去。暴露完成之后会返回到RegistryProt的暴露口中，去将之前暴露的invoker注册到一个Registry中，并且它还会将已经暴露的invoker保存到它的ProviderConsumer的注册表中，这个注册表之后会被远端引用。

注册表结构

ConcurrentHashMap，key是String，value是set

### 流程总结

1、在使用XML方式配置标签的情况下，首先它有个dubbo的DubboBeanDefinitionParser终端解析器去解析标签，像application、registry、protocol、consumer的一些配置，会相应的解析成对应的config对象

2、但是有个特殊情况是暴露的服务不会写到对应的service标签里，它不是直接解析成ServiceConfig，而是会解析成ServiceConfig下面的一个子类叫做ServiceBean。

3、ServiceBean实现了2个接口

4、第一个是实现了InitializingBean，到时候Spring会回调afterPropertiesSet方法，afterPropertiesSet就是把spring配置文件中dubbo的标签内容保存起来，保存在ServiceBean里面。

5、第二个是实现了ApplicatioinListener<ContextRefreshedEvent>接口，叫应用的监听器，使用这种观察者模式去监听ContextRefreshEvent，能够去刷新afterPropertiesSet的回调，也就是当我们ioc容器里面所有对象都创建完成以后，回调onApplicationEvent方法。

6、onApplicationEvent方法，它里面会调用一个服务暴露方法,也就是ServiceBean的父类ServiceConfig.export方法

7、export里面的doExportUrls执行暴露url地址

8、首先它会获取注册中心的地址，因为注册中心可以配置多个，所以这里是一个集合

9、然后他会获取协议的配置信息，因为协议也可以配置多个，所以这里也是一个集合

10、最后调用doExportUrlsFor1Protocol方法

11、这个方法会根据服务的提供者+接口类型+URL去创建了一个代理的invoker

12、接着会通过protocol.export(invoker)返回Exporter，但是这Protocol它相关的两个实现，第一个是RegistryProtocol，第二个是DubboProtocol

13、在RegistryProtocol.export里面第一步就是doLocalExport，就是本地暴露

14、在本地暴露的过程中调用DubboProtocol.export将invoke封装成一个ExporterChangeableWrapper，同时在openServer里面启动netty服务器，开启端口等着调用者连进来，这是DubboProtocol的export的过程

15、第二步将已经暴露的invoker保存到ProviderConsumerRegTable的注册表中，这个注册表之后会被远端引用

### Dubbo服务暴露通俗流程

调用远程服务过程：

1、生产者注册服务，将接口注册到注册中心。Map<接口类名,url集合>

2、生产者本地注册，将接口的实现类保存起来。Map<接口类名,实现类>

3、生产者启动内置tomcat或者NettyServer，接收并处理请求（将服务暴露出去，供消费者调用）。

3.1、获取用户的协议配置，同时通过protocol.start()启动对应的协议服务（HttpProtocol/NettyProtocol），启动的过程会拦截所有请求，将请求转到dispatch里面。

3.2、接收请求后，反序列化请求参数，根据接口类名在本地注册里面找到实现类class。

3.3、class.getMethod根据方法名和方法参数类型通过反射技术找到类需要调用的方法method。

3.4、method.invoke根据实现类和方法参数值通过反射技术去执行方法。

3.5、将返回值通过resp.getOutputStream()返回出去。

4、消费者通过JDK动态代理调用远程服务

4.1、将接口类名+接口方法名+方法参数类型+方法参数值等这些信息包装在Invocation里面

4.2、从注册中心通过接口类名获取到url集合，然后根据配置的负载均衡策略获取到url配置信息

4.3、序列化Invocation对象

4.4、获取客户配置的协议，ProtocolFactory.getProtocol判断是http、netty还是其他协议，从而获取到对应的Protocol(HttpProtocol/NettyProtocol等)，然后通过protocol.send()将请求发送出去（这里需要定义Protocol接口，里面所有的协议需要实现这个接口的start()、send()方法，方便用户切换协议）。

### 为什么dubbo需要支持多个协议？

不同的协议适用的场景不同。

1. dubbo适用于常规远程服务方法调用，尽量不要用dubbo协议传输大文件或超大字符串。
2. rmi适用于常规远程服务方法调用，与原生RMI服务互操作，可用来传输文件。
3. hessian适用于页面传输，文件传输，或与原生hessian服务互操作。
4. http适用于需同时给应用程序和浏览器 JS 使用的服务，暂不支持文件传输。
5. webServices适用于系统集成，跨语言调用。

### Dubbo的负载均衡策略

Random LoadBalance：随机

RoundRobinLoadBalance：循环

LeastActiveLoadBalance：最少活跃调用数

ConsistentHashLoadBalance：一致性Hash

### 集群容错

Failover Cluster：失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。

## Nginx

### Nginx有那些负载均衡策略

1. 轮询：每个请求会按时间顺序逐一分配到不同的后端服务器
2. 一致性哈希：相同请求参数的情况都会分配到同一台机器上
3. 权重：在轮询策略的基础上指定轮询的几率
4. 最少连接：把请求转发给连接数较少的后端服务器
5. 响应时间（第三方）：按照服务器端的响应时间来分配请求，响应时间短的优先分配
6. 依据URL分配（第三方）：按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，要配合缓存命中来使用。

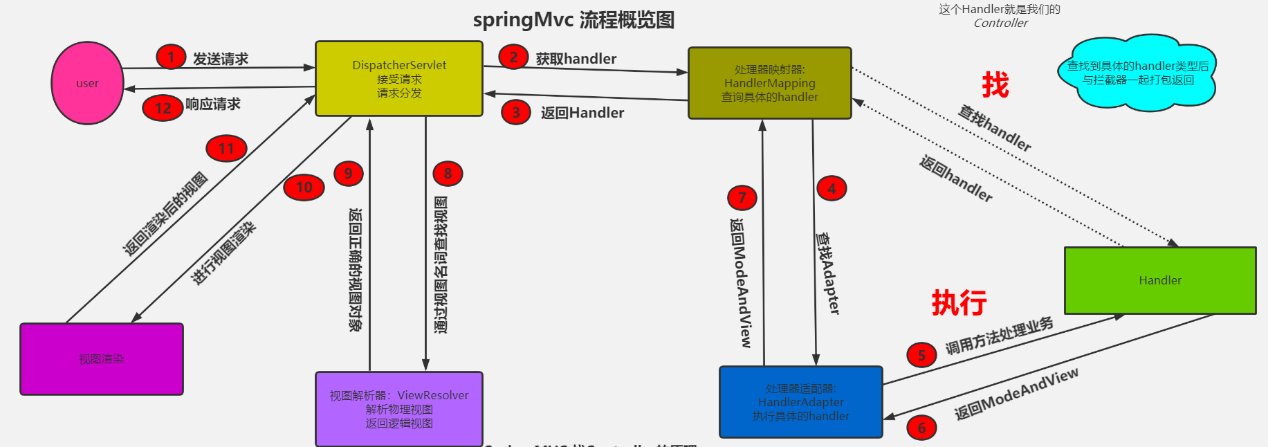
注意：第三方的负载均衡策略的实现需要安装第三方插件

### Nginx缺点

Nginx存在中心化的问题，比如：

1. 当nginx宕机了，需要修改服务A的请求地址，才能调用到服务B
2. 当新增了服务B的地址，需要修改nginx的配置文件，才能实现负载均衡

## Spring MVC



### 源码流程

DispatcherServIet：前端控制器

HandlerMapping：处理器映射器

HandlerExecutionChain：处理器执行链，里面包含了HandlerInterceptor和Handler

HandlerAdapter：处理器适配器

ViewResolver：视图解析器

Handler：Controller

1、客户端请求到前端控制器（DispatcherServIet）

2、前端控制器（DispatcherServIet）请求处理器映射器（HandlerMapping）

3、处理器映射器（HandlerMapping）根据URL查找相应的处理器执行链（HandlerExecutionChain）和处理器（Handler)，返回给前端控制器（DispatcherServlet）

4、前端控制器（Dispatcherservlet）请求处理器适配器（HandlerAdapter）

5、处理器适配器（HandlerAdapter)执行处理器（Handler），调用方法处理业务逻辑，生成ModelAndView，返回ModelAndView给前端控制器（DispatcherServlet）

6、前端控制器（DispatcherServIet〕请求视图解析器（ViewResolver）

7、视图解析器（ViewResovler）返回视图对象给前端控制器（Dispatcherservlet〕

8、最后渲染视图

### 通俗流程

1、tomcat实现servlet的spi规范

1.1 tomcat找到ServletContainerInitializer的实现类，并调用onStartup方法

1.2 Spring的SpringServletContainerInitializer类继承了ServletContainerInitializer类，重写onStartup方法，SpringServletContainerInitializer这个类只用循环调用子类的onStartup方法。

1.3 tomcat调用Spring的SpringServletContainerInitializer.onStartup方法

2、tomcat启动时会在实现SpringServletContainerInitializer子类的onStartup方法中，扫描并加载web.xml配置文件

3、web.xml配置文件配置spring-mvc.xml

4、web.xml会加载DispatcherServlet

5、DispatcherServlet扫描spring-mvc.xml的配置文件，扫描整个项目，根据配置文件给定的目录来扫描

6、扫描所有加了@Controller注解的类

7、当扫描到加了@Controller注解的类之后遍历里面所有的方法

8、拿到方法对象之后，解析方法上面是否加了@RequestMapping注解

9、定义一个map集合（Map<String,Method>），把@RequestMapping的value与方法对象绑定起来

10、拦截到请求之后，拿到请求的URI

11、拿URI去Map里面去get，找到方法

12、消息转换器处理，扫描方法里面的参数，然后根据request.getParameterNames()获取到前台传过来的参数，通过java反射对参数进行赋值

13、视图解析器处理。如果方法加了@ResponseBody，则进行resp.getWriter().write(result)直接将数据通过Response返回出去，如果是ModelAndView则进行视图的处理；如果是String，则通过请求转发req.getRequestDispatcher(前缀+String.valueOf(result)+后缀).forward(req,resp)进行页面跳转；

## 抽象类和接口的区别

1. 抽象类可以有默认的方法实现，接口不存在方法实现
2. 抽象类用extends继承，接口用implement实现
3. 抽象类可以有构造器，接口不能有构造器
4. 抽象类除了不能实例化以外和普通类没有区别，接口和类完全不一样
5. 抽象类可以有main方法，接口不能有
6. 抽象类添加新方法子类可以不用改代码，接口添加新方法子类需要改代码

## IO、BIO、NIO

IO：基于字节流和字符流进行操作。

BIO：同步并阻塞。服务器实现模式为：一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理。

NIO：同步非阻塞。NIO基于 Channel和Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。

知识点1：NIO和传统IO之间第一个大的区别是，IO是面向流的，NIO是面向缓冲区的。

知识点2：BIO和NIO的区别是，BIO是同步阻塞的，NIO是同步非阻塞。所谓的同步就是没有回调方法。

## Redis

### 基础数据类型

string、list、set、hash、zet

### 持久化机制

rdb：redis会单独创建（fork）一个与当前进程一模一样的子进程来进行持久化，这个子进程的所有数据（变量。环境变量，程序程序计数器等）都和原进程一模一样，会先将数据写入到一个临时文件中，待持久化结束了，再用这个临时文件替换上次持久化好的文件，整个过程中，主进程不进行任何的io操作，这就确保了极高的性能，默认15分钟一次。

aof：将Reids的操作日志以追加的方式写入文件，读操作是不记录，它可以设置成每次数据变更同步一次到数据库，或者每秒同步一次到数据库。

注意：官方建议rdb+aof，rdb适合大规模的数据恢复，对数据完整性和一致性不高，在一定间隔时间做一次备份，如果redis意外down机的话，就会丢失最后一次快照后的所有操作 ，aof优化了这一部分缺陷，aof丢失不会超过2秒的数据。

### 缓存几大问题

|  |  |  |
| --- | --- | --- |
| 问题 | 解决方案 | 场景 |
| 缓存穿透 | 缓存空对象，布隆过滤器 | 查询缓存和数据库都没有的数据 |
| 缓存击穿 | 分布式锁或者设置互斥锁（mutex:key:） | 数据刚好失效（缓存当中没有这条），此时来了并发访问 |
| 缓存雪崩 | 搭建高可用集群，错开缓存失效时间 | 大部分数据失效（宕机） |

### 主从复制

主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，mester已写为主，slaver已读为主。

#### 主从复制工作流程

总体分为大三步：

##### 建立连接

1.设置master的地址和端口，发送slaveofip port指令，master会返回响应客户端，根据响应信息保存master ip port信息 (连接测试)

2.根据保存的信息创建连接master的socket

3.周期性发送ping，master会响应pong

4.发送指令 auth password（身份验证），master验证身份

5.发送slave端口信息，master保存slave的端口号

##### 数据同步

1.slave发送指令 psyn2

2.master 执行bgsave

3.在第一个salve连接时，创建命令缓存区

4.生成RDB文件，通过socket发送给slave

5.slave接收RDB，清空数据，执行RDB文件恢复过程

6.发送命令告知RDB恢复已经完成（告知全量复制完成）

7.master发送复制缓冲区信息

8.slave接收信息，执行重写后恢复数据

注意： master会保存slave从我这里拿走了多少数据，保存salve的偏移量

##### 命令传播

slave心跳：replconf ack {offset} 汇报slave自己的offset（偏移量，通过偏移量来判断数据是否有进行操作），获取最新数据指令

命令传播阶段出现断网：

•网络闪断闪连忽略

•段时间断网增量

•长时间断网全量

全量复制核心三个要素

1. 服务器运行id

用于服务器之间通信验证身份，master首次连接slave时，会将自己的run\_id发送给slave，slave保存此ID

2. 主服务器积压的命令缓冲区

先进先出队列

3. 主从服务器的复制偏移量

用于比对偏移量，然后判断出执行全量还是增量

#### 主从复制缺点

1.由于所有的写操作都是先在Master上操作，然后同步更新到Slave上，所以从Master同步到Slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使这个问题更加严重。

2.当主机宕机之后，将不能进行写操作，需要手动将从机升级为主机，从机需要重新制定master

#### 简单总结

一个master可以有多个Slave

一个slave只能有一个master

数据流向是单向的，只能从主到从

### 哨兵模式

哨兵的核心功能是主节点的自动故障转移。通俗来讲哨兵模式的出现是就是为了解决我们主从复制模式中需要我们人为操作的东西变为自动版，并且它比人为要更及时。

#### 哨兵主要功能（做了哪些事）

1、监控（Monitoring）：哨兵会不断地检查主节点和从节点是否运作正常。

2、自动故障转移（Automatic Failover）：当主节点不能正常工作时，哨兵会开始自动故障转移操作，它会将失效主节点的其中一个从节点升级为新的主节点，并让其他从节点改为复制新的主节点。

3、配置提供者（Configuration Provider）：客户端在初始化时，通过连接哨兵来获得当前Redis服务的主节点地址。

4、通知（Notification）：哨兵可以将故障转移的结果发送给客户端。

其中，监控和自动故障转移功能，使得哨兵可以及时发现主节点故障并完成转移；而配置提供者和通知功能，则需要在与客户端的交互中才能体现。

#### 选举领导者哨兵节点

当主节点被判断客观下线以后，各个哨兵节点会进行协商，选举出一个领导者哨兵节点，并由该领导者节点对其进行故障转移操作。监视该主节点的所有哨兵都有可能被选为领导者，选举使用的算法是Raft算法；Raft算法的基本思路是先到先得：即在一轮选举中，哨兵A向B发送成为领导者的申请，如果B没有同意过其他哨兵，则会同意A成为领导者，哨兵选择的过程很快，谁先完成客观下线，一般就能成为领导者。

#### 故障转移

故障转移大致分为三步：

（1）选举出的领导者哨兵，开始进行故障转移操作

（2）在从节点中选择新的主节点遵从的原则如下：

1、首先过滤掉不健康的从节点；

2、过滤响应慢的节点

3、过滤与master断开时间最久的

4、优先原则：先选择优先级最高的从节点（由replica-priority指定）；如果优先级无法区分，则选择复制偏移量最大的从节点；如果仍无法区分，则选择runid最小的从节点。

（3）更新主从状态：通过slaveof no one命令，让选出来的从节点成为主节点；并通过slaveof命令让其他节点成为其从节点。将已经下线的主节点保持关注，当改节点重新上线后设置为从节点。

## Seata AT模式流程

Seata可以管理TM、全局、分支、undo\_log事务，不能管理不在TM下的其他微服务

TC:Seata提供的事务管理中心。

TM:事务的发起者，并且最终与tc通信告诉事务的成功与否

RM:单个的微服务，也就是最终的资源管理者

1.TM 向 TC 申请开启一个全局事务，TC 创建全局事务后返回全局唯一的 XID，XID 会在全局事务的上下文中传播

1.1 开启全局事务的入口：GlobalTransactionScanner extends AbstractAutoProxyCreator implements InitializingBean, ApplicationContextAware

1.2 AbstractAutoProxyCreator.wrapIfNecessary方法作用：为了解决单例bean之间的循环依赖问题，提前将代理Bean对象暴露出去，也就是程序如果需要代理Bean，就可以通过这个方式获取到Bean数据，执行自己代理Bean逻辑并返回出去

1.2.1 往global\_table表插入一条全局事务记录数据

1.2.2 创建全局事务后返回全局唯一的 XID

1.3 InitializingBean.afterPropertiesSet方法作用：初始化Netty客户端

2.RM 向 TC 注册分支事务，该分支事务归属于拥有相同 XID 的全局事务

2.1 执行前置镜像，保存数据库操作前数据源。拼接sql-->select 列名 from 表 where 本地sql条件 for update;

2.2 执行本地sql，操作数据库，不提交事务

2.3 执行后置镜像，保存数据库操作后数据源

2.4 RM 向 TC 注册分支事务，该分支事务归属于拥有相同 XID 的全局事务，XID的数据通过request.header请求头传到各个微服务，然后各个微服务保存起来

2.4.1 往branch\_table表插入一条分支事务数据，每个操作数据库的微服务插入一条

2.4.2 怎么确保每个操作数据库的微服务都往branch\_table表插入一条?此时就用到了DataSourceProxy这个代理数据源

2.4.3 DataSourceProxy代理数据源的主要功能是在执行本地sql前，插入业务逻辑，这样就能做到拦截每个操作数据库的微服务

2.4.4 此时不是所有的微服务都拦截，DataSourceProxy里面的逻辑只拦截在全局事务里面的分支事务，判断标准就是有没有XID,没有XID则直接调用业务方法

2.4.5 本地sql查出来1条数据，就往lock\_table表插入一条id为“表名：id”的数据，存放被锁住的对应数据源，防止多线程操作数据源导致赃数据

2.4.5 防止脏读：@GlobalTransaction + for update

2.4.6 防止脏写：@GlobalLock

2.4.7 Seata是怎么保证分布式事务和分布式事务/分布式事务和本地事务的隔离性？用到全局锁

2.4.7.1 当分布式事务在操作数据库前，会根据自己的xid查询一次branch\_table表数据

2.4.7.2 如果找到，则允许操作数据库，同时往lock\_table表记录操作哪些数据（剔除lock\_table表重复操作记录）

2.4.7.3 如果没找到，则会去lock\_table表查一次

2.4.7.4 没找到，证明不在同一个事务里面，并且操作的也不是同一条记录，运行操作数据库

2.4.7.5 找到了，证明不在同一个事务里面，但是操作有被其他事务锁住的记录，则会抛出异常，客户端接收到这个消息后，会进行重试操作，执行30次，每次间隔10毫秒，重试不成功则进行全局回滚。

2.7 前置镜像和后置镜像的返回结果组装成undoLog对象，插入到undo\_log表中(一个分支事务一条undo\_log数据)，不提交事务

2.8 本地sql事务、undo\_log表事务一起提交

2.9 向 TC 报告一阶段执行情况，改变分支事务状态。

3.TM 向 TC 发起全局提交或回滚；

3.1 在注册分支事务时如果出现报错，则进行全局回滚

4.TC 调度 XID 下的分支事务完成提交或者回滚。

4.1 客户端分支事务提交给服务端之后，服务端异步处理数据

4.1.1 服务端根据xid(全局事务id)和branch\_id(分支事务id)删除lock\_table表数据

4.1.2 服务端根据xid(全局事务id)和branch\_id(分支事务id)删除branch\_table表数据，同时发送一条删除undo\_log表消息给客户端，客户端将这些消息加入到队列

4.1.3 客户端定时任务处理删除undo\_log数据，当队列达到1000的情况下就先删除一次，否则就全部删除

4.1.4 分支事务表删除完之后，服务端根据xid(全局事务id)删除global\_table表数据

4.1.5 表数据都删除完之后在进行全局事务提交