**Mysql**

### 事务的基本要素

1. 原子性：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行
2. 一致性：事务开始前和结束后，数据库的完整性约束没有被破坏。
3. 隔离性：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。
4. 持久性：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

### Mysql的存储引擎

1. InnoDB存储引擎：InnoDB存储引擎支持事务，其设计目标主要面向在线事务处理（OLTP）的应用。其特点是行锁设计，支持外键，并支持非锁定锁，即默认读取操作不会产生锁。从Mysql5.5.8版本开始，InnoDB存储引擎是默认的存储引擎。
2. MyISAM存储引擎：MyISAM存储引擎不支持事务、表锁设计，支持全文索引，主要面向一些OLAP数据库应用。InnoDB的数据文件本身就是主索引文件，而MyISAM的主索引和数据是分开的。
3. NDB存储引擎：NDB存储引擎是一个集群存储引擎，其结构是share nothing的集群架构，能提供更高的可用性。NDB的特点是数据全部放在内存中（从MySQL 5.1版本开始，可以将非索引数据放在磁盘上），因此主键查找的速度极快，并且通过添加NDB数据存储节点可以线性地提高数据库性能，是高可用、高性能的集群系统。NDB存储引擎的连接操作是在MySQL数据库层完成的，而不是在存储引擎层完成的。这意味着，复杂的连接操作需要巨大的网络开销，因此查询速度很慢。如果解决了这个问题，NDB存储引擎的市场应该是非常巨大的。
4. Memory存储引擎：Memory存储引擎（之前称HEAP存储引擎）将表中的数据存放在内存中，如果数据库重启或发生崩溃，表中的数据都将消失。它非常适合用于存储临时数据的临时表，以及数据仓库中的纬度表。Memory存储引擎默认使用哈希索引，而不是我们熟悉的B+树索引。虽然Memory存储引擎速度非常快，但在使用上还是有一定的限制。比如，只支持表锁，并发性能较差，并且不支持TEXT和BLOB列类型。最重要的是，存储变长字段时是按照定常字段的方式进行的，因此会浪费内存。
5. Archive存储引擎：Archive存储引擎只支持INSERT和SELECT操作，从MySQL 5.1开始支持索引。Archive存储引擎使用zlib算法将数据行（row）进行压缩后存储，压缩比一般可达1∶10。正如其名字所示，Archive存储引擎非常适合存储归档数据，如日志信息。Archive存储引擎使用行锁来实现高并发的插入操作，但是其本身并不是事务安全的存储引擎，其设计目标主要是提供高速的插入和压缩功能。
6. Maria存储引擎：Maria存储引擎是新开发的引擎，设计目标主要是用来取代原有的MyISAM存储引擎，从而成为MySQL的默认存储引擎。它可以看做是MyISAM的后续版本。Maria存储引擎的特点是：支持缓存数据和索引文件，应用了行锁设计，提供了MVCC功能，支持事务和非事务安全的选项，以及更好的BLOB字符类型的处理性能。

### 事务的并发问题

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：A事务读取了B事务已经提交的新增数据。注意和不可重复读的区别，这里是新增，不可重复读是更改（或删除）。select某记录是否存在，不存在，准备插入此记录，但执行 insert 时发现此记录已存在，无法插入，此时就发生了幻读。

### MySQL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 | 是 | 是 | 是 |
| 不可重复读 | 否 | 是 | 是 |
| 可重复读 | 否 | 否 | 是 |
| 串行化 | 否 | 否 | 否 |

### Mysql的逻辑结构

* 最上层的服务类似其他CS结构，比如连接处理，授权处理。
* 第二层是Mysql的服务层，包括SQL的解析分析优化，存储过程触发器视图等也在这一层实现。
* 最后一层是存储引擎的实现，类似于Java接口的实现，Mysql的执行器在执行SQL的时候只会关注API的调用，完全屏蔽了不同引擎实现间的差异。比如Select语句，先会判断当前用户是否拥有权限，其次到缓存（内存）查询是否有相应的结果集，如果没有再执行解析sql，检查SQL 语句语法是否正确，再优化生成执行计划，调用API执行。

### SQL执行顺序

SQL的执行顺序：from---where--group by---having---select---order by

### MVCC,redolog,undolog,binlog

* undoLog 也就是我们常说的回滚日志文件 主要用于事务中执行失败，进行回滚，以及MVCC中对于数据历史版本的查看。由引擎层的InnoDB引擎实现,是逻辑日志,记录数据修改被修改前的值,比如"把id='B' 修改为id = 'B2' ，那么undo日志就会用来存放id ='B'的记录”。当一条数据需要更新前,会先把修改前的记录存储在undolog中,如果这个修改出现异常,,则会使用undo日志来实现回滚操作,保证事务的一致性。当事务提交之后，undo log并不能立马被删除,而是会被放到待清理链表中,待判断没有事物用到该版本的信息时才可以清理相应undolog。它保存了事务发生之前的数据的一个版本，用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读。
* redoLog 是重做日志文件是记录数据修改之后的值，用于持久化到磁盘中。redo log包括两部分：一是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；二是磁盘上的重做日志文件(redo log file)，该部分日志是持久的。由引擎层的InnoDB引擎实现,是物理日志,记录的是物理数据页修改的信息,比如“某个数据页上内容发生了哪些改动”。当一条数据需要更新时,InnoDB会先将数据更新，然后记录redoLog 在内存中，然后找个时间将redoLog的操作执行到磁盘上的文件上。不管是否提交成功我都记录，你要是回滚了，那我连回滚的修改也记录。它确保了事务的持久性。每个InnoDB存储引擎至少有1个重做日志文件组（group），每个文件组下至少有2个重做日志文件，如默认的ib\_logfile0和ib\_logfile1。为了得到更高的可靠性，用户可以设置多个的镜像日志组（mirrored log groups），将不同的文件组放在不同的磁盘上，以此提高重做日志的高可用性。在日志组中每个重做日志文件的大小一致，并以循环写入的方式运行。InnoDB存储引擎先写重做日志文件1，当达到文件的最后时，会切换至重做日志文件2，再当重做日志文件2也被写满时，会再切换到重做日志文件1中。
* MVCC多版本并发控制是MySQL中基于乐观锁理论实现隔离级别的方式，用于读已提交和可重复读取隔离级别的实现。在MySQL中，会在表中每一条数据后面添加两个字段：最近修改该行数据的事务ID，指向该行（undolog表中）回滚段的指针。Read View判断行的可见性，创建一个新事务时，copy一份当前系统中的活跃事务列表。意思是，当前不应该被本事务看到的其他事务id列表。已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的ReadView,而可重复读隔离级别则在第一次读的时候生成一个ReadView，之后的读都复用之前的ReadView。

### binlog和redolog的区别

1. redolog是在InnoDB存储引擎层产生，而binlog是MySQL数据库的上层服务层产生的。
2. 两种日志记录的内容形式不同。MySQL的binlog是逻辑日志，其记录是对应的SQL语句，对应的事务。而innodb存储引擎层面的重做日志是物理日志，是关于每个页（Page）的更改的物理情况。
3. 两种日志与记录写入磁盘的时间点不同，binlog日志只在事务提交完成后进行一次写入。而innodb存储引擎的重做日志在事务进行中不断地被写入，并日志不是随事务提交的顺序进行写入的。
4. binlog不是循环使用，在写满或者重启之后，会生成新的binlog文件，redolog是循环使用。
5. binlog可以作为恢复数据使用，主从复制搭建，redolog作为异常宕机或者介质故障后的数据恢复使用。

### Mysql读写分离以及主从同步

1. 原理：主库将变更写binlog日志，然后从库连接到主库后，从库有一个IO线程，将主库的binlog日志拷贝到自己本地，写入一个中继日志中，接着从库中有一个sql线程会从中继日志读取binlog，然后执行binlog日志中的内容，也就是在自己本地再执行一遍sql，这样就可以保证自己跟主库的数据一致。
2. 问题：这里有很重要一点，就是从库同步主库数据的过程是串行化的，也就是说主库上并行操作，在从库上会串行化执行，由于从库从主库拷贝日志以及串行化执行sql特点，在高并发情况下，从库数据一定比主库慢一点，是有延时的，所以经常出现，刚写入主库的数据可能读不到了，要过几十毫秒，甚至几百毫秒才能读取到。还有一个问题，如果突然主库宕机了，然后恰巧数据还没有同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。所以mysql实际上有两个机制，一个是半同步复制，用来解决主库数据丢失问题，一个是并行复制，用来解决主从同步延时问题。
3. 半同步复制：semi-sync复制，指的就是主库写入binlog日志后，就会将强制此时立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库ack之后才会认为写完成。
4. 并发复制：指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这样库级别的并行。（将主库分库也可缓解延迟问题）

### Next-Key Lock

InnoDB 采用 Next-Key Lock 解决幻读问题。在insert into test(xid) values (1), (3), (5), (8), (11);后，由于xid上是有索引的，该算法总是会去锁住索引记录。现在，该索引可能被锁住的范围如下：(-∞, 1], (1, 3], (3, 5], (5, 8], (8, 11], (11, +∞)。Session A（select \* from test where id = 8 for update）执行后会锁住的范围：(5, 8], (8, 11]。除了锁住8所在的范围，还会锁住下一个范围，所谓Next-Key。

### InnoDB的关键特性

1. 插入缓冲：对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个Insert Buffer对象中。然后再以一定的频率和情况进行Insert Buffer和辅助索引页子节点的merge（合并）操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索引插入的性能。
2. 两次写：两次写带给InnoDB存储引擎的是数据页的可靠性，有经验的DBA也许会想，如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚地认识到，如果这个页本身已经发生了损坏（物理到page页的物理日志成功页内逻辑日志失败），再对其进行重做是没有意义的。这就是说，在应用（apply）重做日志前，用户需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做。在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是会通过memcpy函数将脏页先复制到内存中的doublewrite buffer，之后通过doublewrite buffer再分两次，每次1MB顺序地写入共享表空间的物理磁盘上，这就是doublewrite。
3. 自适应哈希索引：InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引。
4. 异步IO：为了提高磁盘操作性能，当前的数据库系统都采用异步IO（AIO）的方式来处理磁盘操作。AIO的另一个优势是可以进行IO Merge操作，也就是将多个IO合并为1个IO，这样可以提高IOPS的性能。
5. 刷新邻接页：当刷新一个脏页时，InnoDB存储引擎会检测该页所在区（extent）的所有页，如果是脏页，那么一起进行刷新。这样做的好处显而易见，通过AIO可以将多个IO写入操作合并为一个IO操作，故该工作机制在传统机械磁盘下有着显著的优势。

### Mysql如何保证一致性和持久性

MySQL为了保证ACID中的一致性和持久性，使用了WAL(Write-Ahead Logging,先写日志再写磁盘)。Redo log就是一种WAL的应用。当数据库忽然掉电，再重新启动时，MySQL可以通过Redo log还原数据。也就是说，每次事务提交时，不用同步刷新磁盘数据文件，只需要同步刷新Redo log就足够了。

### InnoDB的行锁模式

* 共享锁(S)：用法lock in share mode，又称读锁，允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。
* 排他锁(X)：用法for update，又称写锁，允许获取排他锁的事务更新数据，阻止其他事务取得相同的数据集共享读锁和排他写锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。在没有索引的情况下，InnoDB只能使用表锁。

### 为什么选择B+树作为索引结构

* Hash索引：Hash索引底层是哈希表，哈希表是一种以key-value存储数据的结构，所以多个数据在存储关系上是完全没有任何顺序关系的，所以，对于区间查询是无法直接通过索引查询的，就需要全表扫描。所以，哈希索引只适用于等值查询的场景。而B+ 树是一种多路平衡查询树，所以他的节点是天然有序的（左子节点小于父节点、父节点小于右子节点），所以对于范围查询的时候不需要做全表扫描
* 二叉查找树：解决了排序的基本问题，但是由于无法保证平衡，可能退化为链表。
* 平衡二叉树：通过旋转解决了平衡的问题，但是旋转操作效率太低。
* 红黑树：通过舍弃严格的平衡和引入红黑节点，解决了 AVL旋转效率过低的问题，但是在磁盘等场景下，树仍然太高，IO次数太多。
* B+树：在B树的基础上，将非叶节点改造为不存储数据纯索引节点，进一步降低了树的高度；此外将叶节点使用指针连接成链表，范围查询更加高效。

### B+树的叶子节点都可以存哪些东西

可能存储的是整行数据，也有可能是主键的值。B+树的叶子节点存储了整行数据的是主键索引，也被称之为聚簇索引。而索引B+ Tree的叶子节点存储了主键的值的是非主键索引，也被称之为非聚簇索引

### 覆盖索引

指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取。也可以称之为实现了索引覆盖。

### 查询在什么时候不走（预期中的）索引

1. 模糊查询 %like
2. 索引列参与计算,使用了函数
3. 非最左前缀顺序
4. where对null判断
5. where不等于
6. or操作有至少一个字段没有索引
7. 需要回表的查询结果集过大（超过配置的范围）

### explain命令概要

1. id:select选择标识符
2. select\_type:表示查询的类型。
3. table:输出结果集的表
4. partitions:匹配的分区
5. type:表示表的连接类型
6. possible\_keys:表示查询时，可能使用的索引
7. key:表示实际使用的索引
8. key\_len:索引字段的长度
9. ref:列与索引的比较
10. rows:扫描出的行数(估算的行数)
11. filtered:按表条件过滤的行百分比
12. Extra:执行情况的描述和说明

### explain 中的 select\_type（查询的类型）

1. SIMPLE(简单SELECT，不使用UNION或子查询等)
2. PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部分，最外层的select被标记为PRIMARY)
3. UNION(UNION中的第二个或后面的SELECT语句)
4. DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)
5. UNION RESULT(UNION的结果，union语句中第二个select开始后面所有select)
6. SUBQUERY(子查询中的第一个SELECT，结果不依赖于外部查询)
7. DEPENDENT SUBQUERY(子查询中的第一个SELECT，依赖于外部查询)
8. DERIVED(派生表的SELECT, FROM子句的子查询)
9. UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)

### explain 中的 type（表的连接类型）

1. system：最快，主键或唯一索引查找常量值，只有一条记录，很少能出现
2. const：PK或者unique上的等值查询
3. eq\_ref：PK或者unique上的join查询，等值匹配，对于前表的每一行(row)，后表只有一行命中
4. ref：非唯一索引，等值匹配，可能有多行命中
5. range：索引上的范围扫描，例如：between/in
6. index：索引上的全集扫描，例如：InnoDB的count
7. ALL：最慢，全表扫描(full table scan)

### explain 中的 Extra（执行情况的描述和说明）

1. Using where:不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤
2. Using temporary：表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询，常见 group by ; order by
3. Using filesort：当Query中包含 order by 操作，而且无法利用索引完成的排序操作称为“文件排序”
4. Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。
5. Impossible where：这个值强调了where语句会导致没有符合条件的行（通过收集统计信息不可能存在结果）。
6. Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行
7. No tables used：Query语句中使用from dual 或不含任何from子句

### 分析sql查询速度慢原因

执行时间长

等待时间长

主要问题：查询语句写的烂

索引失效：

单值索引：(select \* from where name='') name为索引 概念：只为该表的某一个字段创建索引 如:name

复合索引:(select \* from where name='' and email = '') 概念：为该表的多个字段创建复合索引 如：name 和 email

关联查询太多join(设计缺陷或不得已的需求)

服务器各个参数设置（缓存，线程数等）

导致sql慢关联查询太多join分析解决

SQL执行的顺序：

手写顺序：

SELECT DISTINCT <select\_list>

FROM

<left\_table> <join\_type>

JOIN <right\_table> ON <join\_condition>

WHERE

<where\_condition>

GROUP BY

<group\_by\_list>

HAVING

<having\_by\_list>

ORDER BY

<order\_by\_condition>

LIMIT <limit number>

机读顺序：

FROM <left\_table> <join\_type>

ON <join\_condition>

<join\_type> JOIN <right\_table>

WHERE

<where\_condition>

GROUP BY

<group\_by\_list>

HAVING

<having\_by\_list>

SELECT DISTINCT <select\_list>

ORDER BY

<order\_by\_condition>

LIMIT <limit number>

join七种理论

第一种：查询俩张表的共有部分使用内连接：SELECT <select\_list> FROM TableA A INNER JOIN TableB B ON A.Key=B.Kay

第二种：查询A B共有和A的独有使用左连接：SELECT <select\_list> FROM TableA A LEFT JOIN TableB B ON A.Key=B.Kay

第三种：查询A B共有和B的独有使用右连接：SELECT <select\_list> FROM TableA A RIGHT JOIN TableB B ON A.Key=B.Kay

第四种：查询A 的独有使用左连接：SELECT <select\_list> FROM TableA A LEFT JOIN TableB B ON A.Key=B.Kay WHERE B.key is NULL

第五种：查询B 的独有使用右连接：SELECT <select\_list> FROM TableA A RIGHT JOIN TableB B ON A.Key=B.Kay WHERE A.key is NULL

第六种：查询A B所有使用全连接：SELECT <select\_list> FROM TableA A Full OUTER JOIN TableB B ON A.Key=B.Kay

第七种：查询A B独有使用全连接：SELECT <select\_list> FROM TableA A Full OUTER JOIN TableB B ON A.Key=B.Kay WHERE WHERE B.key is NULL OR A.key is NULL

建表练习：

部门表

CREATE TABLE `tbl\_dept`(

`id` INT(11) NOT NULL AUTO\_INCREMENT,

`deptName` VARCHAR(30) DEFAULT NULL,

`locAdd` VARCHAR(40) DEFAULT NULL,

PRIMARY KEY(`id`)

)ENGINE = INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET = utf8;

员工表

CREATE TABLE `tbl\_emp`(

`id`INT(11) NULL AUTO\_INCREMENT,

`name` VARCHAR(20) DEFAULT NULL,

`deptId` INT(11) DEFAULT NULL,

PRIMARY KEY(`id`),

KEY `fk\_dept\_id`(`deptId`)

)ENGINE = INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET =utf8;

数据插入：

INSERT INTO tbl\_dept(deptName,locAdd)VALUES("RD",11);

INSERT INTO tbl\_dept(deptName,locAdd)VALUES("HR",12);

INSERT INTO tbl\_dept(deptName,locAdd)VALUES("MK",13);

INSERT INTO tbl\_dept(deptName,locAdd)VALUES("MIS",14);

INSERT INTO tbl\_dept(deptName,locAdd)VALUES("FD",15);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("z3",1);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("z4",1);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("z5",1);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("w5",2);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("w6",2);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("w7",3);

INSERT INTO tbl\_emp(NAME,deptId)VALUES("w8",4);

根据上述表结构显示join其中情况

第一种：SELECT \* FROM tbl\_dept d INNER JOIN tbl\_emp e ON d.id=e.deptId

第二种：SELECT \* FROM tbl\_dept d LEFT JOIN tbl\_emp e ON d.id=e.deptId 左表全部,右表不满足的补NULL

第三种：SELECT \* FROM tbl\_dept d RIGHT JOIN tbl\_emp e ON d.id=e.deptId 右表全部,左表不满足补NULL

第四种：SELECT \* FROM tbl\_dept d LEFT JOIN tbl\_emp e ON d.id=e.deptId WHERE e.deptId is NULL

第五种：SELECT \* FROM tbl\_dept d RIGHT JOIN tbl\_emp e ON d.id=e.deptId WHERE d.id is NULL

第六种：SELECT \* FROM tbl\_dept d FULL OUTER JOIN tbl\_emp e ON d.id=e.deptId

mysql不支持 FULL OUTER

SELECT \* FROM tbl\_dept d LEFT JOIN tbl\_emp e ON d.id=e.deptId

union

SELECT \* FROM tbl\_dept d RIGHT JOIN tbl\_emp e ON d.id=e.deptId

union(合并去重)

第七种：SELECT \* FROM tbl\_dept d LEFT JOIN tbl\_emp e ON d.id=e.deptId WHERE e.deptId is NULL

union

SELECT \* FROM tbl\_dept d RIGHT JOIN tbl\_emp e ON d.id=e.deptId WHERE d.id is NULL

### 数据库优化指南

## （一）数据库设计优化（三范式）

* 第一范式：确保每列数据保持原子性
* 第二范式：确保表中的每列都和主键相关
* 第三范式：确保每列都和主键列直接相关，而不是间接相关

## （二）索引优化

注意有些使用情境下，虽然建立了索引，但是并没有起到作用。

* 当在where子句中对字段进行null值判断时，将导致引擎放弃使用索引而进行全表扫描。
* 当在where子句中对索引字段进行非等值判断时，如!=,<,>,NOT IN等，将导致引擎放弃使用索引而进行全表扫描。
* 当在where子句中对索引字段进行运算操作时，包括函数，计算表达式等，将导致引擎放弃使用索引而进行全表扫描： 如where age + 10 = 30;where substr(name) = ‘Kevin’;
* 当对索引字段进行like模糊查询时，如果待匹配字符串第一个字符为模糊匹配符%，则不会用到索引：如where name like ‘%kevin’;
* 当在 where 子句中使用or来连接条件时，如果其中一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描。

## （三）语句优化

1. 对于频繁where查询，尽量利用上索引；对于没有建立索引或者索引无效的，建立合适的索引或者修改语句以便利用索引。
2. 合理调整where条件的顺序，将能过滤掉更多数据的条件放在末尾。
3. 避免采用 select \*, 这会占用更多时间。
4. 理论上，尽量使用多表连接（join）查询，避免子查询。
5. 对于LIMIT查询，先在索引上完成排序分页操作，最后根据主键关联回表查询所需要的其他列内容。
6. 充分利用查询优化器，多处存在相同的查询语句时，尽量保持一样（大小写也一致）。
7. 利用explain语句来分析语句执行情况，针对性优化。

## （四）分库分表

MySQL单表的数据量在500w-1000w之间性能比较好，单库数据量在5000w以内性能比较好,,超过阈值性能都会下降。同时单库单表在IO读写和数据库连接上都存在着限制。为了提升性能和解决以上问题, 在数据量和并发访问量大时，分库分表时必要的。

## （五）读写分离

## 一台数据库支持最大连接数是有限的，如果用户的并发访问很多，一台服务器无法满足需求。这时候可以采用读写分离的方式进行集群。读写分离采用主从配置，一台master,多台slave, 所有写操作都往master上写，读操作往从Slave上读。Slave数据库从Master同步数据。 mysql支持读写分离配置。

## （六）缓存利用

## 缓存可以在一定程度上避免磁盘IO读写，节省查询时间。常见的缓存服务有Redis/Tair/Memcached等。

## （七）系统配置优化

## 通过修改数据库的配置，来提升数据库性能，如mysql的my.cnf配置。 有时候优化过程中，需要找到影响性能的sql语句，可以通过配置my.cnf打开mysql的慢查询日志。

**Java JVM**

### 运行时数据区域

1. 程序计数器：程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。是线程私有”的内存。
2. Java虚拟机栈：与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧 ，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。
3. 本地方法栈：本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间的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，而本地方法栈则为虚拟机使用到的Native方法服务。
4. Java堆：对于大多数应用来说，Java堆是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
5. 方法区：方法区用于存储已被虚拟机加载的类信息、常量、静态变量，如static修饰的变量加载类的时候就被加载到方法区中。运行时常量池是方法区的一部分，class文件除了有类的字段、接口、方法等描述信息之外，还有常量池用于存放编译期间生成的各种字面量和符号引用。在老版jdk，方法区也被称为永久代。在1.8之后，由于永久代内存经常不够用或发生内存泄露，爆出异常java.lang.OutOfMemoryError，所以在1.8之后废弃永久代，引入元空间的概念。元空间是方法区的在HotSpot jvm 中的实现，元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。理论上取决于32位/64位系统可虚拟的内存大小。可见也不是无限制的，需要配置参数。

### 分代回收

HotSpot JVM把年轻代分为了三部分：1个Eden区和2个Survivor区（分别叫from和to）。一般情况下，新创建的对象都会被分配到Eden区(一些大对象特殊处理),这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中。

因为年轻代中的对象基本都是朝生夕死的，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，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。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。

### 动态年龄计算

Hotspot在遍历所有对象时，按照年龄从小到大对其所占用的大小进行累积，当累积的某个年龄大小超过了survivor区的一半时，取这个年龄和MaxTenuringThreshold中更小的一个值，作为新的晋升年龄阈值。

JVM引入动态年龄计算，主要基于如下两点考虑：

1. 如果固定按照MaxTenuringThreshold设定的阈值作为晋升条件： a）MaxTenuringThreshold设置的过大，原本应该晋升的对象一直停留在Survivor区，直到Survivor区溢出，一旦溢出发生，Eden+Svuvivor中对象将不再依据年龄全部提升到老年代，这样对象老化的机制就失效了。 b）MaxTenuringThreshold设置的过小，“过早晋升”即对象不能在新生代充分被回收，大量短期对象被晋升到老年代，老年代空间迅速增长，引起频繁的Major GC。分代回收失去了意义，严重影响GC性能。
2. 相同应用在不同时间的表现不同：特殊任务的执行或者流量成分的变化，都会导致对象的生命周期分布发生波动，那么固定的阈值设定，因为无法动态适应变化，会造成和上面相同的问题。

### 常见的垃圾回收机制

1. 引用计数法：引用计数法是一种简单但速度很慢的垃圾回收技术。每个对象都含有一个引用计数器,当有引用连接至对象时,引用计数加1。当引用离开作用域或被置为null时,引用计数减1。虽然管理引用计数的开销不大,但这项开销在整个程序生命周期中将持续发生。垃圾回收器会在含有全部对象的列表上遍历,当发现某个对象引用计数为0时,就释放其占用的空间。
2. 可达性分析算法：这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。

### CMS的执行过程

1. 初始标记(STW initial mark)：这个过程从垃圾回收的"根对象"开始，只扫描到能够和"根对象"直接关联的对象，并作标记。所以这个过程虽然暂停了整个JVM，但是很快就完成了。
2. 并发标记(Concurrent marking)：这个阶段紧随初始标记阶段，在初始标记的基础上继续向下追溯标记。并发标记阶段，应用程序的线程和并发标记的线程并发执行，所以用户不会感受到停顿。
3. 并发预清理(Concurrent precleaning)：并发预清理阶段仍然是并发的。在这个阶段，虚拟机查找在执行并发标记阶段新进入老年代的对象(可能会有一些对象从新生代晋升到老年代， 或者有一些对象被分配到老年代)。通过重新扫描，减少下一个阶段"重新标记"的工作，因为下一个阶段会Stop The World。
4. 重新标记(STW remark)：这个阶段会暂停虚拟机，收集器线程扫描在CMS堆中剩余的对象。扫描从"跟对象"开始向下追溯，并处理对象关联。
5. 并发清理(Concurrent sweeping)：清理垃圾对象，这个阶段收集器线程和应用程序线程并发执行。
6. 并发重置(Concurrent reset)：这个阶段，重置CMS收集器的数据结构状态，等待下一次垃圾回收。

### G1的执行过程

1. 标记阶段：首先是初始标记(Initial-Mark),这个阶段也是停顿的(stop-the-word)，并且会稍带触发一次yong GC。
2. 并发标记：这个过程在整个堆中进行，并且和应用程序并发运行。并发标记过程可能被yong GC中断。在并发标记阶段，如果发现区域对象中的所有对象都是垃圾，那个这个区域会被立即回收(图中打X)。同时，并发标记过程中，每个区域的对象活性(区域中存活对象的比例)被计算。
3. 再标记：这个阶段是用来补充收集并发标记阶段产新的新垃圾。与之不同的是，G1中采用了更快的算法:SATB。
4. 清理阶段：选择活性低的区域(同时考虑停顿时间)，等待下次yong GC一起收集，对应GC log: [GC pause (mixed)]，这个过程也会有停顿(STW)。
5. 回收/完成：新的yong GC清理被计算好的区域。但是有一些区域还是可能存在垃圾对象，可能是这些区域中对象活性较高，回收不划算，也肯能是为了迎合用户设置的时间，不得不舍弃一些区域的收集。

### G1和CMS的比较

1. CMS收集器是获取最短回收停顿时间为目标的收集器，因为CMS工作时，GC工作线程与用户线程可以并发执行，以此来达到降低停顿时间的目的（只有初始标记和重新标记会STW）。但是CMS收集器对CPU资源非常敏感。在并发阶段，虽然不会导致用户线程停顿，但是会占用CPU资源而导致引用程序变慢，总吞吐量下降。
2. CMS仅作用于老年代，是基于标记清除算法，所以清理的过程中会有大量的空间碎片。
3. CMS收集器无法处理浮动垃圾，由于CMS并发清理阶段用户线程还在运行，伴随程序的运行自热会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在本次收集中处理它们，只好留待下一次GC时将其清理掉。
4. G1是一款面向服务端应用的垃圾收集器，适用于多核处理器、大内存容量的服务端系统。G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短STW的停顿时间，它满足短时间停顿的同时达到一个高的吞吐量。
5. 从JDK 9开始，G1成为默认的垃圾回收器。当应用有以下任何一种特性时非常适合用G1：Full GC持续时间太长或者太频繁；对象的创建速率和存活率变动很大；应用不希望停顿时间长(长于0.5s甚至1s)。
6. G1将空间划分成很多块（Region），然后他们各自进行回收。堆比较大的时候可以采用，采用复制算法，碎片化问题不严重。整体上看属于标记整理算法,局部(region之间)属于复制算法。
7. G1 需要记忆集来记录新生代和老年代之间的引用关系，这种数据结构在 G1 中需要占用大量的内存，可能达到整个堆内存容量的 20% 甚至更多。而且 G1 中维护记忆集的成本较高，带来了更高的执行负载，影响效率。所以 CMS 在小内存应用上的表现要优于 G1，而大内存应用上 G1 更有优势，大小内存的界限是6GB到8GB。（Card Table（CMS中）的结构是一个连续的byte[]数组，扫描Card Table的时间比扫描整个老年代的代价要小很多！G1也参照了这个思路，不过采用了一种新的数据结构 Remembered Set 简称Rset。RSet记录了其他Region中的对象引用本Region中对象的关系，属于points-into结构（谁引用了我的对象）。而Card Table则是一种points-out（我引用了谁的对象）的结构，每个Card 覆盖一定范围的Heap（一般为512Bytes）。G1的RSet是在Card Table的基础上实现的：每个Region会记录下别的Region有指向自己的指针，并标记这些指针分别在哪些Card的范围内。 这个RSet其实是一个Hash Table，Key是别的Region的起始地址，Value是一个集合，里面的元素是Card Table的Index。每个Region都有一个对应的Rset。）

### 哪些对象可以作为GC Roots

1. 虚拟机栈（栈帧中的本地变量表）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（即一般说的Native方法）引用的对象。

### GC中Stop the world（STW）

在执行垃圾收集算法时，Java应用程序的其他所有除了垃圾收集收集器线程之外的线程都被挂起。此时，系统只能允许GC线程进行运行，其他线程则会全部暂停，等待GC线程执行完毕后才能再次运行。这些工作都是由虚拟机在后台自动发起和自动完成的，是在用户不可见的情况下把用户正常工作的线程全部停下来，这对于很多的应用程序，尤其是那些对于实时性要求很高的程序来说是难以接受的。

但不是说GC必须STW,你也可以选择降低运行速度但是可以并发执行的收集算法，这取决于你的业务。

### 垃圾回收算法

1. 停止-复制：先暂停程序的运行,然后将所有存活的对象从当前堆复制到另一个堆,没有被复制的对象全部都是垃圾。当对象被复制到新堆时,它们是一个挨着一个的,所以新堆保持紧凑排列,然后就可以按前述方法简单,直接的分配了。缺点是一浪费空间,两个堆之间要来回倒腾,二是当程序进入稳定态时,可能只会产生极少的垃圾,甚至不产生垃圾,尽管如此,复制式回收器仍会将所有内存自一处复制到另一处。
2. 标记-清除：同样是从堆栈和静态存储区出发,遍历所有的引用,进而找出所有存活的对象。每当它找到一个存活的对象,就会给对象一个标记,这个过程中不会回收任何对象。只有全部标记工作完成的时候,清理动作才会开始。在清理过程中,没有标记的对象会被释放,不会发生任何复制动作。所以剩下的堆空间是不连续的,垃圾回收器如果要希望得到连续空间的话,就得重新整理剩下的对象。
3. 标记-整理：它的第一个阶段与标记/清除算法是一模一样的，均是遍历GC Roots，然后将存活的对象标记。移动所有存活的对象，且按照内存地址次序依次排列，然后将末端内存地址以后的内存全部回收。因此，第二阶段才称为整理阶段。
4. 分代收集算法：把Java堆分为新生代和老年代，然后根据各个年代的特点采用最合适的收集算法。新生代中，对象的存活率比较低，所以选用复制算法，老年代中对象存活率高且没有额外空间对它进行分配担保，所以使用“标记-清除”或“标记-整理”算法进行回收。

### Minor GC和Full GC触发条件

* Minor GC触发条件：当Eden区满时，触发Minor GC。
* Full GC触发条件：
  1. 调用System.gc时，系统建议执行Full GC，但是不必然执行
  2. 老年代空间不足
  3. 方法区空间不足
  4. 通过Minor GC后进入老年代的平均大小大于老年代的可用内存
  5. 由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

### 对象什么时候进入老年代

1. 大对象直接进入老年代。 虚拟机提供了一个阈值参数，令大于这个设置值的对象直接在老年代中分配。如果大对象进入新生代，新生代采用的复制算法收集内存，会导致在Eden区和两个Survivor区之间发生大量的内存复制，应该避免这种情况。
2. 长期存活的对象进入老年代。 虚拟机给每个对象定义了一个年龄计数器，对象在Eden区出生，经过一次Minor GC后仍然存活，并且能被Survivor区容纳的话，将被移动到Survivor区中，此时对象年龄设为1。然后对象在Survivor区中每熬过一次 Minor GC，年龄就增加1，当年龄超过设定的阈值时，就会被移动到老年代中。
3. 动态对象年龄判定： 如果在 Survivor 空间中所有相同年龄的对象，大小总和大于 Survivor 空间的一半，那么年龄大于或等于该年龄的对象就直接进入老年代，无须等到阈值中要求的年龄。
4. 空间分配担保： 如果老年代中最大可用的连续空间大于新生代所有对象的总空间，那么 Minor GC 是安全的。如果老年代中最大可用的连续空间大于历代晋升到老年代的对象的平均大小，就进行一次有风险的 Minor GC，如果小于平均值，就进行 Full GC 来让老年代腾出更多的空间。因为新生代使用的是复制算法，为了内存利用率，只使用其中一个 Survivor 空间来做轮换备份，因此如果大量对象在 Minor GC 后仍然存活，导致 Survivor 空间不够用，就会通过分配担保机制，将多出来的对象提前转到老年代，但老年代要进行担保的前提是自己本身还有容纳这些对象的剩余空间，由于无法提前知道会有多少对象存活下来，所以取之前每次晋升到老年代的对象的平均大小作为经验值，与老年代的剩余空间做比较。

### TLAB

在Java中，典型的对象不再堆上分配的情况有两种：TLAB和栈上分配（通过逃逸分析）。JVM在内存新生代Eden Space中开辟了一小块线程私有的区域，称作TLAB（Thread-local allocation buffer）。默认设定为占用Eden Space的1%。在Java程序中很多对象都是小对象且用过即丢，它们不存在线程共享也适合被快速GC，所以对于小对象通常JVM会优先分配在TLAB上，并且TLAB上的分配由于是线程私有所以没有锁开销。因此在实践中分配多个小对象的效率通常比分配一个大对象的效率要高。也就是说，Java中每个线程都会有自己的缓冲区称作TLAB（Thread-local allocation buffer），每个TLAB都只有一个线程可以操作，TLAB结合bump-the-pointer技术可以实现快速的对象分配，而不需要任何的锁进行同步，也就是说，在对象分配的时候不用锁住整个堆，而只需要在自己的缓冲区分配即可。

### Java对象分配的过程

1. 编译器通过逃逸分析，确定对象是在栈上分配还是在堆上分配。如果是在堆上分配，则进入2.
2. 如果tlab\_top + size <= tlab\_end，则在在TLAB上直接分配对象并增加tlab\_top 的值，如果现有的TLAB不足以存放当前对象则3.
3. 重新申请一个TLAB，并再次尝试存放当前对象。如果放不下，则4。
4. 在Eden区加锁（这个区是多线程共享的），如果eden\_top + size <= eden\_end则将对象存放在Eden区，增加eden\_top 的值，如果Eden区不足以存放，则5。
5. 执行一次Young GC（minor collection）
6. 经过Young GC之后，如果Eden区任然不足以存放当前对象，则直接分配到老年代。

### 对象内存分配的两种方法

1. 指针碰撞(Serial、ParNew等带Compact过程的收集器) ：假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”（Bump the Pointer）。
2. 空闲列表(CMS这种基于Mark-Sweep算法的收集器) ：如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”（Free List）。

### JVM类加载过程

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7个阶段。

1. 加载：通过一个类的全限定名来获取定义此类的二进制字节流，将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构，在内存中生成一个代表这个类的Class对象，作为方法去这个类的各种数据的访问入口
2. 验证：验证是连接阶段的第一步，这一阶段的目的是确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟自身的安全。
3. 准备：准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法去中进行分配。这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中。
4. 解析：解析阶段是虚拟机将常量池内的符号（Class文件内的符号）引用替换为直接引用（指针）的过程。
5. 初始化：初始化阶段是类加载过程的最后一步，开始执行类中定义的Java程序代码（字节码）。

### 双亲委派模型

双亲委派的意思是如果一个类加载器需要加载类，那么首先它会把这个类请求委派给父类加载器去完成，每一层都是如此。一直递归到顶层，当父加载器无法完成这个请求时，子类才会尝试去加载。

### 双亲委派模型的"破坏"

一个典型的例子便是JNDI服务，JNDI现在已经是Java的标准服务，它的代码由启动类加载器去加载(在JDK 1.3时放进去的rt.jar)，但JNDI的目的就是对资源进行集中管理和查找，它需要调用由独立厂商实现并部署在应用程序的ClassPath下的JNDI接口提供者(SPI,Service Provider Interface)的代码，但启动类加载器不可能“认识”这些代码那该怎么办?

为了解决这个问题，Java设计团队只好引入了一个不太优雅的设计:线程上下文类加载器(Thread Context ClassLoader)。这个类加载器可以通过java.lang.Thread类的 setContextClassLoaser()方法进行设置，如果创建线程时还未设置，它将会从父线程中继承 一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器。

有了线程上下文类加载器，就可以做一些“舞弊”的事情了，JNDI服务使用这个线程上下 文类加载器去加载所需要的SPI代码，也就是父类加载器请求子类加载器去完成类加载的动 作，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，实际上已经 违背了双亲委派模型的一般性原则，但这也是无可奈何的事情。Java中所有涉及SPI的加载动 作基本上都采用这种方式，例如JNDI、JDBC、JCE、JAXB和JBI等。

### JVM锁优化和膨胀过程

1. 自旋锁：自旋锁其实就是在拿锁时发现已经有线程拿了锁，自己如果去拿会阻塞自己，这个时候会选择进行一次忙循环尝试。也就是不停循环看是否能等到上个线程自己释放锁。自适应自旋锁指的是例如第一次设置最多自旋10次，结果在自旋的过程中成功获得了锁，那么下一次就可以设置成最多自旋20次。
2. 锁粗化：虚拟机通过适当扩大加锁的范围以避免频繁的拿锁释放锁的过程。
3. 锁消除：通过逃逸分析发现其实根本就没有别的线程产生竞争的可能（别的线程没有临界量的引用），或者同步块内进行的是原子操作，而“自作多情”地给自己加上了锁。有可能虚拟机会直接去掉这个锁。
4. 偏向锁：在大多数的情况下，锁不仅不存在多线程的竞争，而且总是由同一个线程获得。因此为了让线程获得锁的代价更低引入了偏向锁的概念。偏向锁的意思是如果一个线程获得了一个偏向锁，如果在接下来的一段时间中没有其他线程来竞争锁，那么持有偏向锁的线程再次进入或者退出同一个同步代码块，不需要再次进行抢占锁和释放锁的操作。
5. 轻量级锁：当存在超过一个线程在竞争同一个同步代码块时，会发生偏向锁的撤销。当前线程会尝试使用CAS来获取锁，当自旋超过指定次数(可以自定义)时仍然无法获得锁，此时锁会膨胀升级为重量级锁。
6. 重量级锁：重量级锁依赖对象内部的monitor锁来实现，而monitor又依赖操作系统的MutexLock（互斥锁）。当系统检查到是重量级锁之后，会把等待想要获取锁的线程阻塞，被阻塞的线程不会消耗CPU，但是阻塞或者唤醒一个线程，都需要通过操作系统来实现。

### 什么情况下需要开始类加载过程的第一个阶段加载

1. 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
2. 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含main（）方法的那个类），虚拟机会先初始化这个主类。

### i++操作的字节码指令

1. 将int类型常量加载到操作数栈顶
2. 将int类型数值从操作数栈顶取出，并存储到到局部变量表的第1个Slot中
3. 将int类型变量从局部变量表的第1个Slot中取出，并放到操作数栈顶
4. 将局部变量表的第1个Slot中的int类型变量加1
5. 表示将int类型数值从操作数栈顶取出，并存储到到局部变量表的第1个Slot中，即i中
   1. ，可以在 浏览器中查看。
   2. jstack(Java堆栈跟踪工具)：jstack命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈 的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循 环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。线程出现停顿 的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些 什么事情，或者等待着什么资源。
6. JDK的可视化工具
   1. JConsole
   2. VisualVM

### JVM常见参数

1. -Xms20M：表示设置JVM启动内存的最小值为20M，必须以M为单位
2. -Xmx20M：表示设置JVM启动内存的最大值为20M，必须以M为单位。将-Xmx和-Xms设置为一样可以避免JVM内存自动扩展。大的项目-Xmx和-Xms一般都要设置到10G、20G甚至还要高
3. -verbose:gc：表示输出虚拟机中GC的详细情况
4. -Xss128k：表示可以设置虚拟机栈的大小为128k
5. -Xoss128k：表示设置本地方法栈的大小为128k。不过HotSpot并不区分虚拟机栈和本地方法栈，因此对于HotSpot来说这个参数是无效的
6. -XX:PermSize=10M：表示JVM初始分配的永久代（方法区）的容量，必须以M为单位
7. -XX:MaxPermSize=10M：表示JVM允许分配的永久代（方法区）的最大容量，必须以M为单位，大部分情况下这个参数默认为64M
8. -Xnoclassgc：表示关闭JVM对类的垃圾回收
9. -XX:+TraceClassLoading表示查看类的加载信息
10. -XX:+TraceClassUnLoading：表示查看类的卸载信息
11. -XX:NewRatio=4：表示设置年轻代（包括Eden和两个Survivor区）/老年代 的大小比值为1：4，这意味着年轻代占整个堆的1/5
12. -XX:SurvivorRatio=8：表示设置2个Survivor区：1个Eden区的大小比值为2:8，这意味着Survivor区占整个年轻代的1/5，这个参数默认为8
13. -Xmn20M：表示设置年轻代的大小为20M
14. -XX:+HeapDumpOnOutOfMemoryError：表示可以让虚拟机在出现内存溢出异常时Dump出当前的堆内存转储快照
15. -XX:+UseG1GC：表示让JVM使用G1垃圾收集器
16. -XX:+PrintGCDetails：表示在控制台上打印出GC具体细节
17. -XX:+PrintGC：表示在控制台上打印出GC信息
18. -XX:PretenureSizeThreshold=3145728：表示对象大于3145728（3M）时直接进入老年代分配，这里只能以字节作为单位
19. -XX:MaxTenuringThreshold=1：表示对象年龄大于1，自动进入老年代,如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代被回收的概率。
20. -XX:CompileThreshold=1000：表示一个方法被调用1000次之后，会被认为是热点代码，并触发即时编译
21. -XX:+PrintHeapAtGC：表示可以看到每次GC前后堆内存布局
22. -XX:+PrintTLAB：表示可以看到TLAB的使用情况
23. -XX:+UseSpining：开启自旋锁
24. -XX:PreBlockSpin：更改自旋锁的自旋次数，使用这个参数必须先开启自旋锁
25. -XX:+UseSerialGC：表示使用jvm的串行垃圾回收机制，该机制适用于单核cpu的环境下
26. -XX:+UseParallelGC：表示使用jvm的并行垃圾回收机制，该机制适合用于多cpu机制，同时对响应时间无强硬要求的环境下，使用-XX:ParallelGCThreads=设置并行垃圾回收的线程数，此值可以设置与机器处理器数量相等。
27. -XX:+UseParallelOldGC：表示年老代使用并行的垃圾回收机制
28. -XX:+UseConcMarkSweepGC：表示使用并发模式的垃圾回收机制，该模式适用于对响应时间要求高，具有多cpu的环境下
29. -XX:MaxGCPauseMillis=100：设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。
30. -XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低响应时间或者收集频率等，此值建议使用并行收集器时，一直打开

### JVM调优目标-何时需要做jvm调优

1. heap 内存（老年代）持续上涨达到设置的最大内存值；
2. Full GC 次数频繁；
3. GC 停顿时间过长（超过1秒）；
4. 应用出现OutOfMemory 等内存异常；
5. 应用中有使用本地缓存且占用大量内存空间；
6. 系统吞吐量与响应性能不高或下降。

### JVM调优实战

1. Major GC和Minor GC频繁

首先优化Minor GC频繁问题。通常情况下，由于新生代空间较小，Eden区很快被填满，就会导致频繁Minor GC，因此可以通过增大新生代空间来降低Minor GC的频率。例如在相同的内存分配率的前提下，新生代中的Eden区增加一倍，Minor GC的次数就会减少一半。

扩容Eden区虽然可以减少Minor GC的次数，但会增加单次Minor GC时间么？扩容后，Minor GC时增加了T1（扫描时间），但省去T2（复制对象）的时间，更重要的是对于虚拟机来说，复制对象的成本要远高于扫描成本，所以，单次Minor GC时间更多取决于GC后存活对象的数量，而非Eden区的大小。因此如果堆中短期对象很多，那么扩容新生代，单次Minor GC时间不会显著增加。

1. 请求高峰期发生GC，导致服务可用性下降

由于跨代引用的存在，CMS在Remark阶段必须扫描整个堆，同时为了避免扫描时新生代有很多对象，增加了可中断的预清理阶段用来等待Minor GC的发生。只是该阶段有时间限制，如果超时等不到Minor GC，Remark时新生代仍然有很多对象，我们的调优策略是，通过参数强制Remark前进行一次Minor GC，从而降低Remark阶段的时间。 另外，类似的JVM是如何避免Minor GC时扫描全堆的？ 经过统计信息显示，老年代持有新生代对象引用的情况不足1%，根据这一特性JVM引入了卡表（card table）来实现这一目的。卡表的具体策略是将老年代的空间分成大小为512B的若干张卡（card）。卡表本身是单字节数组，数组中的每个元素对应着一张卡，当发生老年代引用新生代时，虚拟机将该卡对应的卡表元素设置为适当的值。如上图所示，卡表3被标记为脏（卡表还有另外的作用，标识并发标记阶段哪些块被修改过），之后Minor GC时通过扫描卡表就可以很快的识别哪些卡中存在老年代指向新生代的引用。这样虚拟机通过空间换时间的方式，避免了全堆扫描。

1. STW过长的GC

对于性能要求很高的服务，建议将MaxPermSize和MinPermSize设置成一致（JDK8开始，Perm区完全消失，转而使用元空间。而元空间是直接存在内存中，不在JVM中），Xms和Xmx也设置为相同，这样可以减少内存自动扩容和收缩带来的性能损失。虚拟机启动的时候就会把参数中所设定的内存全部化为私有，即使扩容前有一部分内存不会被用户代码用到，这部分内存在虚拟机中被标识为虚拟内存，也不会交给其他进程使用。

1. 外部命令导致系统缓慢

一个数字校园应用系统，发现请求响应时间比较慢，通过操作系统的mpstat工具发现CPU使用率很高，并且系统占用绝大多数的CPU资 源的程序并不是应用系统本身。每个用户请求的处理都需要执行一个外部shell脚本来获得系统的一些信息，执行这个shell脚本是通过Java的 Runtime.getRuntime().exec()方法来调用的。这种调用方式可以达到目的，但是它在Java 虚拟机中是非常消耗资源的操作，即使外部命令本身能很快执行完毕，频繁调用时创建进程 的开销也非常可观。Java虚拟机执行这个命令的过程是:首先克隆一个和当前虚拟机拥有一样环境变量的进程，再用这个新的进程去执行外部命令，最后再退出这个进程。如果频繁执行这个操作，系统的消耗会很大，不仅是CPU，内存负担也很重。用户根据建议去掉这个Shell脚本执行的语句，改为使用Java的API去获取这些信息后，系统很快恢复了正常。

1. 由Windows虚拟内存导致的长时间停顿

一个带心跳检测功能的GUI桌面程序，每15秒会发送一次心跳检测信号，如果对方30秒以内都没有信号返回，那就认为和对方程序的连接已经断开。程序上线后发现心跳 检测有误报的概率，查询日志发现误报的原因是程序会偶尔出现间隔约一分钟左右的时间完 全无日志输出，处于停顿状态。

因为是桌面程序，所需的内存并不大(-Xmx256m)，所以开始并没有想到是GC导致的 程序停顿，但是加入参数-XX:+PrintGCApplicationStoppedTime-XX:+PrintGCDateStamps- Xloggc:gclog.log后，从GC日志文件中确认了停顿确实是由GC导致的，大部分GC时间都控 制在100毫秒以内，但偶尔就会出现一次接近1分钟的GC。

从GC日志中找到长时间停顿的具体日志信息(添加了-XX:+PrintReferenceGC参数)， 找到的日志片段如下所示。从日志中可以看出，真正执行GC动作的时间不是很长，但从准 备开始GC，到真正开始GC之间所消耗的时间却占了绝大部分。

除GC日志之外，还观察到这个GUI程序内存变化的一个特点，当它最小化的时候，资源 管理中显示的占用内存大幅度减小，但是虚拟内存则没有变化，因此怀疑程序在最小化时它的工作内存被自动交换到磁盘的页面文件之中了，这样发生GC时就有可能因为恢复页面文件的操作而导致不正常的GC停顿。在Java的GUI程序中要避免这种现象，可以 加入参数“-Dsun.awt.keepWorkingSetOnMinimize=true”来解决。

**Spring**

### 什么是三级缓存

1. 第一级缓存：单例缓存池singletonObjects。
2. 第二级缓存：早期提前暴露的对象缓存earlySingletonObjects。（属性还没有值对象也没有被初始化）
3. 第三级缓存：singletonFactories单例对象工厂缓存。

三级缓存详解：[根据 Spring 源码写一个带有三级缓存的 IOC](https://zhuanlan.zhihu.com/p/144627581)

### Spring如何解决循环依赖问题

Spring使用了三级缓存解决了循环依赖的问题。在populateBean()给属性赋值阶段里面Spring会解析你的属性，并且赋值，当发现，A对象里面依赖了B，此时又会走getBean方法，但这个时候，你去缓存中是可以拿的到的。因为我们在对createBeanInstance对象创建完成以后已经放入了缓存当中，所以创建B的时候发现依赖A，直接就从缓存中去拿，此时B创建完，A也创建完，一共执行了4次。至此Bean的创建完成，最后将创建好的Bean放入单例缓存池中。

### BeanFactory和ApplicationContext的区别

1. BeanFactory是Spring里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能。
2. ApplicationContext应用上下文，继承BeanFactory接口，它是Spring的一各更高级的容器，提供了更多的有用的功能。如国际化，访问资源，载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，消息发送、响应机制，AOP等。
3. BeanFactory在启动的时候不会去实例化Bean，中有从容器中拿Bean的时候才会去实例化。ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init=true来让Bean延迟实例化

### 动态代理的实现方式，AOP的实现方式

1. JDK动态代理：利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。
2. CGlib动态代理：利用ASM（开源的Java字节码编辑库，操作字节码）开源包，将代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。
3. 区别：JDK代理只能对实现接口的类生成代理；CGlib是针对类实现代理，对指定的类生成一个子类，并覆盖其中的方法，这种通过继承类的实现方式，不能代理final修饰的类。

### @Transactional错误使用失效场景

1. @Transactional 在private上：当标记在protected、private、package-visible方法上时，不会产生错误，但也不会表现出为它指定的事务配置。可以认为它作为一个普通的方法参与到一个public方法的事务中。
2. @Transactional 的事务传播方式配置错误。
3. @Transactional 注解属性 rollbackFor 设置错误：Spring默认抛出了未检查unchecked异常（继承自 RuntimeException 的异常）或者 Error才回滚事务；其他异常不会触发回滚事务。
4. 同一个类中方法调用，导致@Transactional失效：由于使用Spring AOP代理造成的，因为只有当事务方法被当前类以外的代码调用时，才会由Spring生成的代理对象来管理。
5. 异常被 catch 捕获导致@Transactional失效。
6. 数据库引擎不支持事务。

### Spring的的事务传播机制

1. REQUIRED（默认，常用）：支持使用当前事务，如果当前事务不存在，创建一个新事务。eg:方法B用REQUIRED修饰，方法A调用方法B，如果方法A当前没有事务，方法B就新建一个事务（若还有C则B和C在各自的事务中独立执行），如果方法A有事务，方法B就加入到这个事务中，当成一个事务。
2. SUPPORTS：支持使用当前事务，如果当前事务不存在，则不使用事务。
3. MANDATORY：强制，支持使用当前事务，如果当前事务不存在，则抛出Exception。
4. REQUIRES\_NEW（常用）：创建一个新事务，如果当前事务存在，把当前事务挂起。eg:方法B用REQUIRES\_NEW修饰，方法A调用方法B，不管方法A上有没有事务方法B都新建一个事务，在该事务执行。
5. NOT\_SUPPORTED：无事务执行，如果当前事务存在，把当前事务挂起。
6. NEVER：无事务执行，如果当前有事务则抛出Exception。
7. NESTED：嵌套事务，如果当前事务存在，那么在嵌套的事务中执行。如果当前事务不存在，则表现跟REQUIRED一样。

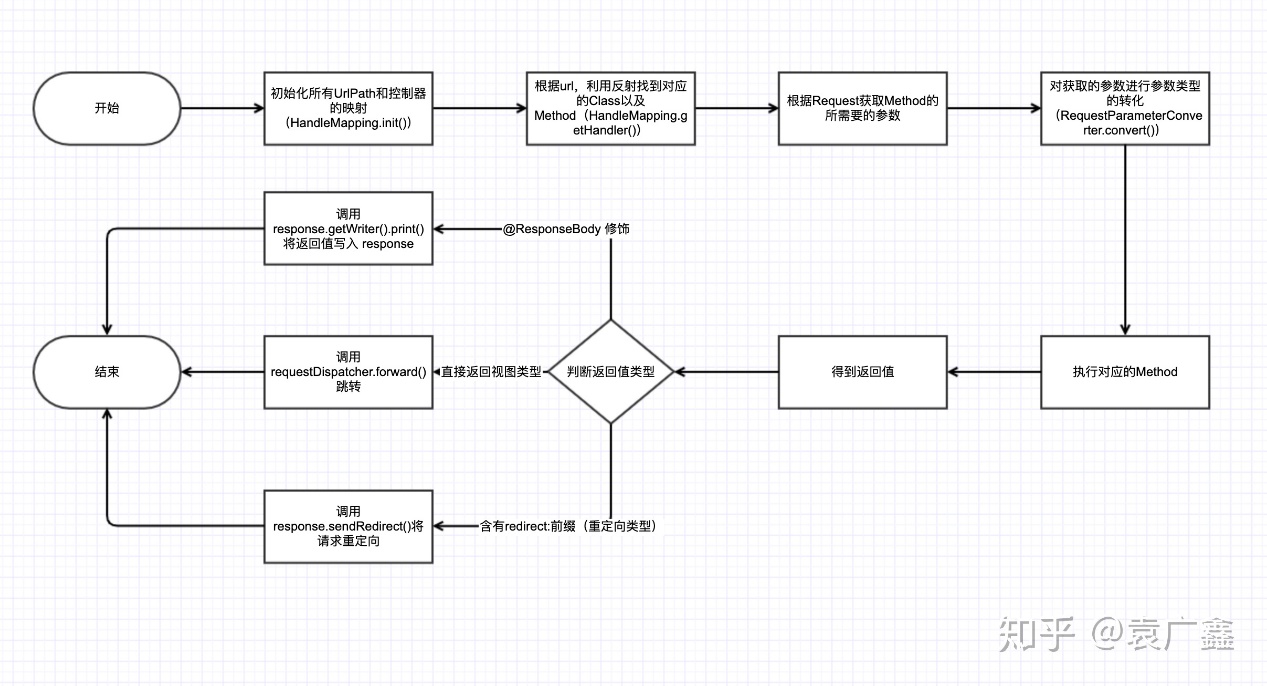
### Spring中Bean的生命周期

1. 实例化 Instantiation
2. 属性赋值 Populate
3. 初始化 Initialization
4. 销毁 Destruction

### Spring的后置处理器

1. BeanPostProcessor：Bean的后置处理器，主要在bean初始化前后工作。（before和after两个回调中间只处理了init-method）
2. InstantiationAwareBeanPostProcessor：继承于BeanPostProcessor，主要在实例化bean前后工作（TargetSource的AOP创建代理对象就是通过该接口实现）
3. BeanFactoryPostProcessor：Bean工厂的后置处理器，在bean定义(bean definitions)加载完成后，bean尚未初始化前执行。
4. BeanDefinitionRegistryPostProcessor：继承于BeanFactoryPostProcessor。其自定义的方法postProcessBeanDefinitionRegistry会在bean定义(bean definitions)将要加载，bean尚未初始化前真执行，即在BeanFactoryPostProcessor的postProcessBeanFactory方法前被调用。

### Spring MVC的工作流程（源码层面）



1. 用户向服务端发送一次请求，这个请求会先到前端控制器DispatcherServlet。
2. DispatcherServlet接收到请求后会调用HandlerMapping处理器映射器。由此得知，该请求该由哪个Controller来处理。
3. DispatcherServlet调用HandlerAdapter处理器适配器，告诉处理器适配器应该要去执行哪个Controller。
4. HandlerAdapter处理器适配器去执行Controller并得到ModelAndView，并层层返回给DispatcherServlet。
5. DispatcherServlet将ModelAndView交给ViewReslover视图解析器解析，然后返回真正的视图。
6. DispatcherServlet将模型数据填充到视图中。
7. DispatcherServlet将结果响应给用户。

## 1. 什么是 Spring 框架?

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。Spring 官网：<https://spring.io/>。

我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。这些模块是：核心容器、数据访问/集成,、Web、AOP（面向切面编程）、工具、消息和测试模块。比如：Core Container 中的 Core 组件是Spring 所有组件的核心，Beans 组件和 Context 组件是实现IOC和依赖注入的基础，AOP组件用来实现面向切面编程。

Spring 官网列出的 Spring 的 6 个特征:

* **核心技术** ：依赖注入(DI)，AOP，事件(events)，资源，i18n，验证，数据绑定，类型转换，SpEL。
* **测试** ：模拟对象，TestContext框架，Spring MVC 测试，WebTestClient。
* **数据访问** ：事务，DAO支持，JDBC，ORM，编组XML。
* **Web支持** : Spring MVC和Spring WebFlux Web框架。
* **集成** ：远程处理，JMS，JCA，JMX，电子邮件，任务，调度，缓存。
* **语言** ：Kotlin，Groovy，动态语言。

## 2. 列举一些重要的Spring模块？

目前最新的5.x版本中 Web 模块的 Portlet 组件已经被废弃掉，同时增加了用于异步响应式处理的 WebFlux 组件。

* **Spring Core：** 基础,可以说 Spring 其他所有的功能都需要依赖于该类库。主要提供 IoC 依赖注入功能。
* **Spring Aspects** ： 该模块为与AspectJ的集成提供支持。
* **Spring AOP** ：提供了面向切面的编程实现。
* **Spring JDBC** : Java数据库连接。
* **Spring JMS** ：Java消息服务。
* **Spring ORM** : 用于支持Hibernate等ORM工具。
* **Spring Web** : 为创建Web应用程序提供支持。
* **Spring Test** : 提供了对 JUnit 和 TestNG 测试的支持。

## 3. @RestController vs @Controller

**Controller 返回一个页面**

单独使用 @Controller 不加 @ResponseBody的话一般使用在要返回一个视图的情况，这种情况属于比较传统的Spring MVC 的应用，对应于前后端不分离的情况。

**@RestController 返回JSON 或 XML 形式数据**

但@RestController只返回对象，对象数据直接以 JSON 或 XML 形式写入 HTTP 响应(Response)中，这种情况属于 RESTful Web服务，这也是目前日常开发所接触的最常用的情况（前后端分离）。

**@Controller +@ResponseBody 返回JSON 或 XML 形式数据**

如果你需要在Spring4之前开发 RESTful Web服务的话，你需要使用@Controller 并结合@ResponseBody注解，也就是说@Controller +@ResponseBody= @RestController（Spring 4 之后新加的注解）。

@ResponseBody 注解的作用是将 Controller 的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到HTTP 响应(Response)对象的 body 中，通常用来返回 JSON 或者 XML 数据，返回 JSON 数据的情况比较多。

### 4.1 谈谈自己对于 Spring IoC 和 AOP 的理解

#### IoC

IoC（Inverse of Control:控制反转）是一种**设计思想**，就是 **将原本在程序中手动创建对象的控制权，交由Spring框架来管理。** IoC 在其他语言中也有应用，并非 Spring 特有。 **IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个Map（key，value）,Map 中存放的是各种对象。**

将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 **IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。** 在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

**Spring IoC的初始化过程：**

其实SpringIOC初始化的过程就是准备好BeanFactory的过程。

（1）**定位并获取资源文件**

ClassPathResource res = new ClassPathResource("my/applicationContext.xml");

因为对象和对象之间的关系存储在xml或properties等语义化配置文件中，首先要定位到配置文件。用资源加载器ResourceLoader将资源文件路径转换为对应的Resource

（2）**解析资源文件**

XmlBeanFactory bf = new XmlBeanFactory(res);

步骤：

1.构造BeanFactory时，首先调用的是BeanDefinitionReader类型的reader属性的loadBeanDefinitions()方法，是整个资源加载的切入点。

* 封装资源文件：当进入BeanDefinitionReader后首先对参数Resource进行EncodedResource类进行封装
* 获取输入流：从Resource中获取InputStream并构造InputSource
* 通过构造器的InputSource实例和Resource实例继续调用loadBeanDefinitions.

2.loadBeanDefinition调用doLoadBeanDefinitons方法，完成以下三个方法

* 对XML文档的验证模式
* 用DocumentLoader处理资源文件，生成Document
* 根据返回的Document信息注册bean信息

首先调用BeanDefinitonDocumentReader 的 doRegisterBeanDefinitions 去注册 bean 定义信息  
  通过实现接口BeanDefinitionDocumentReader 的 DefaultBeanDefinitionDocumentReader 类的 parseBeanDefinitions 来解析 Document ，从 xml 文档根节点递归循环处理各个节点，实际上使用 BeanDefinitionParserDelegate 的 parseBeanDefinitionElement 方法将 bean 节点转换为 BeanDefinitionHolder 对象，完成最终的解析

（3）**注册**

DefaultListableBeanDefiniton.registerBeanDefiniton利用解析好的BeanDefinition对象完成最终的注册。将beanName和BeanDefinition作为键值放到了beanFactory的map中

#### AOP

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，**却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来**，便于**减少系统的重复代码**，**降低模块间的耦合度**，并**有利于未来的可拓展性和可维护性**。

**Spring AOP就是基于动态代理的**，如果要代理的对象，实现了某个接口，那么Spring AOP会使用**JDK Proxy**，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候Spring AOP会使用**Cglib** ，这时候Spring AOP会使用 **Cglib** 生成一个被代理对象的子类来作为代理。

当然你也可以使用 AspectJ ,Spring AOP 已经集成了AspectJ ，AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

### 4.2 Spring AOP 和 AspectJ AOP 有什么区别？

**Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。** Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ ，AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，

如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ ，它比Spring AOP 快很多。

## 5. Spring bean

### 5.1 Spring 中的 bean 的作用域有哪些?

* singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
* prototype : 每次请求都会创建一个新的 bean 实例。
* request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
* session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
* global-session： 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

### 5.2 Spring 中的单例 bean 的线程安全问题了解吗？

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。

常见的有两种解决办法：

1. 在Bean对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

### 5.3 @Component 和 @Bean 的区别是什么？

1. 作用对象不同: @Component 注解作用于类，而@Bean注解作用于方法。
2. @Component通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中（我们可以使用 @ComponentScan 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中）。@Bean 注解通常是我们在标有该注解的方法中定义产生这个 bean,@Bean告诉了Spring这是某个类的示例，当我需要用它的时候还给我。
3. @Bean 注解比 Component 注解的自定义性更强，而且很多地方我们只能通过 @Bean 注解来注册bean。比如当我们引用第三方库中的类需要装配到 Spring容器时，则只能通过 @Bean来实现。

@Bean注解使用示例：

@Configuration

public class AppConfig {

@Bean

public TransferService transferService() {

return new TransferServiceImpl();

}

}

上面的代码相当于下面的 xml 配置

<beans>

<bean id="transferService" class="com.acme.TransferServiceImpl"/>

</beans>

下面这个例子是通过 @Component 无法实现的。

@Bean

public OneService getService(status) {

case (status) {

when 1:

return new serviceImpl1();

when 2:

return new serviceImpl2();

when 3:

return new serviceImpl3();

}

}

### 5.4 将一个类声明为Spring的 bean 的注解有哪些?

我们一般使用 @Autowired 注解自动装配 bean，要想把类标识成可用于 @Autowired 注解自动装配的 bean 的类,采用以下注解可实现：

* @Component ：通用的注解，可标注任意类为 Spring 组件。如果一个Bean不知道属于哪个层，可以使用@Component 注解标注。
* @Repository : 对应持久层即 Dao 层，主要用于数据库相关操作。
* @Service : 对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。
* @Controller : 对应 Spring MVC 控制层，主要用于接受用户请求并调用 Service 层返回数据给前端页面。

### 5.5 Spring 中的 bean 生命周期?

这部分网上有很多文章都讲到了，下面的内容整理自：<https://yemengying.com/2016/07/14/spring-bean-life-cycle/> ，除了这篇文章，再推荐一篇很不错的文章 ：<https://www.cnblogs.com/zrtqsk/p/3735273.html> 。

* Bean 容器找到配置文件中 Spring Bean 的定义。
* Bean 容器利用 Java Reflection API 创建一个Bean的实例。
* 如果涉及到一些属性值 利用 set()方法设置一些属性值。
* 如果 Bean 实现了 BeanNameAware 接口，调用 setBeanName()方法，传入Bean的名字。
* 如果 Bean 实现了 BeanClassLoaderAware 接口，调用 setBeanClassLoader()方法，传入 ClassLoader对象的实例。
* 与上面的类似，如果实现了其他 \*.Aware接口，就调用相应的方法。
* 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行postProcessBeforeInitialization() 方法
* 如果Bean实现了InitializingBean接口，执行afterPropertiesSet()方法。
* 如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
* 如果有和加载这个 Bean的 Spring 容器相关的 BeanPostProcessor 对象，执行postProcessAfterInitialization() 方法
* 当要销毁 Bean 的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy() 方法。
* 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。

### 6.2 SpringMVC 工作原理了解吗?

**原理如下图所示：**

上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 DispatcherServlet 的作用是接收请求，响应结果。

**流程说明（重要）：**

1. 客户端（浏览器）发送请求，直接请求到 DispatcherServlet。
2. DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。
3. 解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。
4. HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑。
5. 处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。
6. ViewResolver 会根据逻辑 View 查找实际的 View。
7. DispaterServlet 把返回的 Model 传给 View（视图渲染）。
8. 把 View 返回给请求者（浏览器）

## 7. Spring 框架中用到了哪些设计模式？

关于下面一些设计模式的详细介绍，可以看笔主前段时间的原创文章[《面试官:“谈谈Spring中都用到了那些设计模式?”。》](https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247485303&idx=1&sn=9e4626a1e3f001f9b0d84a6fa0cff04a&chksm=cea248bcf9d5c1aaf48b67cc52bac74eb29d6037848d6cf213b0e5466f2d1fda970db700ba41&token=255050878&lang=zh_CN#rd) 。

* **工厂设计模式** : Spring使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
* **代理设计模式** : Spring AOP 功能的实现。
* **单例设计模式** : Spring 中的 Bean 默认都是单例的。
* **模板方法模式** : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
* **包装器设计模式** : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
* **观察者模式:** Spring 事件驱动模型就是观察者模式很经典的一个应用。
* **适配器模式** :Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配Controller。
* ......

## 8. Spring 事务

### 8.1 Spring 管理事务的方式有几种？

1. 编程式事务，在代码中硬编码。(不推荐使用)
2. 声明式事务，在配置文件中配置（推荐使用）

**声明式事务又分为两种：**

1. 基于XML的声明式事务
2. 基于注解的声明式事务

### 8.2 Spring 事务中的隔离级别有哪几种?

**TransactionDefinition 接口中定义了五个表示隔离级别的常量：**

* **TransactionDefinition.ISOLATION\_DEFAULT:** 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别.
* **TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED:** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读**
* **TransactionDefinition.ISOLATION\_READ\_COMMITTED:** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生**
* **TransactionDefinition.ISOLATION\_REPEATABLE\_READ:** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**
* **TransactionDefinition.ISOLATION\_SERIALIZABLE:** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读**。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

### 8.3 Spring 事务中哪几种事务传播行为?

**支持当前事务的情况：**

* **TransactionDefinition.PROPAGATION\_REQUIRED：** 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
* **TransactionDefinition.PROPAGATION\_SUPPORTS：** 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
* **TransactionDefinition.PROPAGATION\_MANDATORY：** 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

**不支持当前事务的情况：**

* **TransactionDefinition.PROPAGATION\_REQUIRES\_NEW：** 创建一个新的事务，如果当前存在事务，则把当前事务挂起。
* **TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：** 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
* **TransactionDefinition.PROPAGATION\_NEVER：** 以非事务方式运行，如果当前存在事务，则抛出异常。

**其他情况：**

* **TransactionDefinition.PROPAGATION\_NESTED：** 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

### 8.4 @Transactional(rollbackFor = Exception.class)注解了解吗？

我们知道：Exception分为运行时异常RuntimeException和非运行时异常。事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当@Transactional注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在@Transactional注解中如果不配置rollbackFor属性,那么事务只会在遇到RuntimeException的时候才会回滚,加上rollbackFor=Exception.class,可以让事务在遇到非运行时异常时也回滚。