1.关于数据库中有大量数据，该怎么优化？

大数据量下的数据库查询与插入如何优化？（整理）

摘要： 数据库经常要做一些查询与插入，但是如果查询和插入的数据量过大的时候就会引发数据库性能问题，降低数据库工作效率。因此性能调优是大家在工作中都能够预见的问题，大到世界五百强的核心系统，小到超市的库存系统，几乎都会有要调优的时候。面对形形色色的系统，林林总总的需求，调优的手段也是丰富多彩。 1.尽量...

数据库经常要做一些查询与插入，但是如果查询和插入的数据量过大的时候就会引发数据库性能问题，降低数据库工作效率。因此性能调优是大家在工作中都能够预见的问题，大到世界五百强的核心系统，小到超市的库存系统，几乎都会有要调优的时候。面对形形色色的系统，林林总总的需求，调优的手段也是丰富多彩。

1.尽量使语句符合查询优化器的规则避免全表扫描而使用索引查询

2.避免频繁创建和删除临时表，以减少系统表资源的消耗。

3.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

4.建立高效的索引

SQL 语句的 Select 部分只写必要的列；尽量将 In 子查询重写为 Exists 子查询；去除在谓词列上编写的任何数学运算；尽可能不用 Distinct；

由于优化工具处理“或”逻辑可能有问题，所以尽量采用其他方式重写；

确保所处理的表中数据分布和其他统计信息正确，并反映当前状况；尽可能用 UNION ALL 取代 UNION；

尽可能减少 DB2 的 SQL 请求；尽量将区间谓词重写为 Between 谓词；不要只是为了排序而选择某一列；

我目前所在的系统就是这么一个有实时插入又需要大数据的查询的一个系统。采用了如下手段：

1，当天的记录会放在一个独立的表中.主要是针对实时的插入的记录，记录不要太多以免插入的时候维护索引的开销稳定在一个范围内。

2，历史的记录会按天分区的形式保存在历史表中。这个表一天只会批量的插入一次数据(用的是分区交换的方法)。

3，分区的索引对我的业务性能不好，因为要跨天 查询。历史查询最长时间段是一个月的时间，如果按照一个月一个分区的话，一个分区差不多是一个亿的记录，就算是按月分区的话，再创建分区的本地索引，如果是时间段跨了月份的话估计分区的本地索引性能估计也不行。

4，后来采用一个方案，DB 层上面再放了一个缓冲层，就是我最近在测试的 Timesten 关系型内存数据库，按照时间的老化策略缓冲一个月的数据。具体不展开说了。涉及的内容很多。

只是对于这个系统，我总结一下有以下需要注意的地方:

1,对于一个系统来说，如果查询性能反应不好的话，第一个调整的地方是思考业务的需求是否是合理的？

一个查询既要分页获取前面一页或者几页的数据，又要根据条件获取总的记录数，如果符合的记录总数是上亿条的话，感觉就是一个不合理的要求。

2，市场需求调研人员业务水平根本不合格。

3，前台开发人员写的 SQL 差，根本没有调优的基本概念,术业有专攻啊。

4，如果业务需求合理，SQL 的调整无非是从执行计划开始，如果是 ORACLE10g,开了 cbo，可以用 SQL 优化器 (***SQL TuningAdvisor*** **:STA)**

分析你的 sql。

5，最近的 nosq 和海量分布式的数据库概念很热。公司也在考虑 HBASE 了。

分区，

读写分离。

细节的优化手段大家都说了很多，我说些几个粗略的方面：

1.使用分区表

2.并行查询

3.定期的数据信息采集

4.可以考虑使用 sql hint（生产库上我个人认为还是少指定 HINT，可以考虑用 SQL_PROFILE 固定执行计划）

1 数据量上亿时我都会使用分区，具体的分区方案看业务需求，如果查询数据是按时间来的且时间跨度小，那么我会按天分

2 读写分离，说实话这个对我来说只是理想方案，实现起来往往有许多困难。磁盘阵列能上则上。

3 SQL 语句的优化，实话说很多程序员写出来的 SQL 真的不忍直视。

4 内存数据库，这个我还没有尝试过，但用过的人说还好。

对于大数据量的处理，我通常采用如下方法：

一、对于大数据存储的处理（以下是假设硬件指标合格的情况）

（1）对大表进行分区，根据不同的业务以及数据特征，采用不同的分区方法。比如，销售，可以考虑采用间隔分区技术，多分公司或是多部门，可以考虑采用列表分区或是上述的组合分区等。

（2）如果硬件具备，对于大表进行分离存储，从而减少磁盘争用。

（3）对历史数据进行定期的归档处理。比如，销售的区间分区，可以采用表与分区交换的技术去处理。对于含有复杂的业务的数据表的归档，采用 PL/SQL 脚本与后台作业定时完成。

二、对于查询的处理（以下是基于成本优化器的情况）

对于查询，优化的方法通常会多一些，但是也是最频繁调整的一块内容。通常，我从如下这几个方面来进行考虑并调优.

（1）对于 SQL 本身的编写是否合理。比如基础写法与高级写法之间的配合，在满足业务要求的基础上，做到尽量减少表的访问。

（2）索引的创建是否合理，优化器是否选择了较正确的索引。在不同的业务场景下，B*树索引与位图索引的相互配合是否合理等。

（3）监控系统中产生的争用，根据产生的不同的闩或锁，对 SQL 或是业务处理逻辑进行调整。

（4）如果有必要，根据当前系统的负载与硬件本身的支持，对 PGA，SGA 进行一个分配，使之更为合理。

（5）如果有必要，在优化器参数进行一个调整。

（6）采用并行处理。

三、对数据插入的处理

这是考虑问题当中最薄弱的一块内容，一般不作必要的优化处理，但有如下一些技巧：

（1）对于大数据量的表与表之间的插入，可以采用并行，直接路径、最小化日志。

（2）如果必要，对表本身参数进行一个修改，如 freelist。

（3）如果是同数据源多目标的插入，可以采用多表插入技术。

（4）如果可以，尽量使用 INSERT INTO SELECT,CREATE TABLE AS SELECT 方法。

1.需求阶段的优化改进

摸清用户的真实需求。发掘隐藏的用户需求。

一般这个很难被真实的发掘出来，因为用户的需求在被调研的时候，往往由于调研者对业务的不深入理解，导致只发掘了用户的表面需求，而没有摸清用户的真实需求。导致后期，开发的时候，每次查询都要查询很多张大表等。

2.在数据库设计阶段的改进

规范数据库的表的设计。

大表尽量要有明确的数据保留策略。

尽量不要在大表上建立影响性能的触发器

大表合理的规划分区。

合理的建立索引，这个很重要。也是优化之核心。

数据库参数的合理设置。

表空间的合理规律。

通过在设计阶段，把

尽量用少做事，或者是不做事情，就可以达到用户的需求。

如果在前期需求调研，设计阶段就做好了工作。那么后面的性能优化问题麻烦就可以少很多。

当然通常由于各种原因，往往前期做不到那么好。

那么找到问题，发现问题，解决问题，通常

1. 通过工具来定位问题，例如选择用 10046 TRACE 工具包来实现

2. 找到问题所在以后去理解需求，探索是否能少做事完成需求（选择用索引来替代全表扫描，从而减少访问路径）；

3. 去思考需求背后的真正需求

例如是否可以用 UNION ALL 取代 UNION ，避免不必要的排序引起资源消耗。

4. 去分析资源如何合理应用 （在系统空闲时使用并行技术）。

5.诊断索引方面的问题。

例如是否由于错误的优化器统计信息导致执行了不正确的执行计划。

一种是分库分表

还有一种就是提前运算好存到数据库中，直接取出。

根据情况实际设计是最合适的~

1 大量数据插入的时候，考虑先删除索引，然后重建索引。这样做的缺点是业务不能同时进行。

2 大量数据的查询优化，根据业务，考虑创建不同的索引。

数据库表进行插入、查询操作当数据达到百万甚至千万条级别的时候， 这一切似乎变得相当困难。几经折腾，总算完成了任务。在此做些简单的小结，不足之处，还望高手们帮忙补充补充！

1、 避免使用 Hibernate 框架

Hibernate 用起来虽然方便，但对于海量数据的操作显得力不从心。

关于插入：

试过用 Hibernate 一次性进行 5 万条左右数据的插入，若 ID 使用 sequence 方式生成，Hibernate 将分 5 万次从数据库取得 5 万个 sequence，构造成相应对象后，再分五万次将数据保存到数据库。花了我十分钟时间。主要的时间不是花在插入上，而是花在 5 万次从数据库取 sequence 上，弄得我相当郁闷。虽然后来把 ID 生成方式改成 increase 解决了问题，但还是对那十分钟的等待心有余

悸。

关于查询：

Hibernate 对数据库查询的主要思想还是面向对象的，这将使许多我们不需要查询的数据占用了大量的系统资源（包括数据库资源和本地资 源）。由于对 Hibernate 的偏爱，本着不抛弃、不放弃的作风，做了包括配 SQL，改进 SQL 等等的相当多的尝试，可都以失败告终，不得不忍痛割爱 了。

2、 写查询语句时，要把查询的字段一一列出

查询时不要使用类似 select * from x_table 的语句，要尽量使用 select id,name from x_table，以避免查询出不需要的数据浪费资源。对于海量数据而言，一个字段所占用的资源和查询时间是相当可观的。

3、 减少不必要的查询条件

当我们在做查询时，常常是前台提交一个查询表单到后台，后台解析这个表 单，而后进行查询操作。在我们解析表单时，为了方便起见，常常喜欢将一些不需要查询的条件用永真的条件来代替（如：select count(id) from x_table where name like '%'），其实这样的 SQL 对资源的浪费是相当可怕的。我试过对于同样的近一千万条记录的查询来说，使用 select count(id) from x_table 进行表查询需要 11 秒，而使用 select count(id) from x_table where name like '%'却花了 33 秒。

4、 避免在查询时使用表连接

在做海量数据查询时，应尽量避免表连接（特别是左、右连接），万不得已要进行表连接时，被连接的另一张表数据量一定不能太大，若连接的另一张表也是数万条的话，那估计可以考虑重新设计库表了，因为那需要等待的时间决不是正常用户所能忍受的。

5、 嵌套查询时，尽可能地在第一次 select 就把查询范围缩到最小

在有多个 select 嵌套查询的时候，应尽量在最内层就把所要查询的范围缩到最

小，能分页的先分页。很多时候，就是这样简单地把分页放到内层查询里，对查询效率来说能形成质的变化。

特别是银行系统的，数量级是亿级别的，所以更要考虑下面的方法。

1，怎样造 Java 对象。有句话说得好：尽可能的少造对象。别说千万级，就是上万级都不要考虑造对象了。因为几个请求一并发，喀嚓，系统肯定完蛋。

2，合理摆正系统设计的位置。大量数据操作，和少量数据操作一定是分开的。大量的数据操作，肯定不是 ORM 框架搞定的。绝对不能 ORM，因为 1，要少造对象；2，数据库资源合理利用。就像博主的例子：id 分配就是一个好例子。

3，合理利用数据库的分区、索引技术。

4，有的时候可以考虑临时表之类的，尤其是大数据量。

5，有人说非常大的数据量，一定要用存储过程：jdbc，效果非常好

6，控制好内存，让数据流起来，而不是全部读到内存再处理，而是边读取边处理；

7，合理利用内存，有的数据要缓存；

30 个 oracle 的查询插入的方法：

1 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：

select id from t where num=0

3.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

4.应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or num=20

可以这样查询：

select id from t where num=10

union all

select id from t where num=20

5.in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

6.下面的查询也将导致全表扫描：

select id from t where name like '?c%'

若要提高效率，可以考虑全文检索。

7.如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时;它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id 可以改为强制查询使用索引：

select id from t with(index(索引名)) where num=@num

8.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2=100

应改为:

select id from t where num=100*2

9.应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用

索引而进行全表扫描。如：

select id from t where substring(name,1,3)='abc'--name 以 abc 开头的 id

select id from t where datediff(day,createdate,'2005-11-30')=0--‘2005-11-30’生成的 id

应改为:

select id from t where name like 'abc%'

select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'

10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

create table #t(...)

13.很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(selectnum from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

14.并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL 查询可能不会去利用索引，如一表中有字段 sex，male、female 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。

15.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

18.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段

存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

19.任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

20.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限(只有主键索引)。

21.避免频繁创建和删除临时表，以减少系统表资源的消耗。

22.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

23.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度;如果数据量不大，为了缓和系统表的资源，应先 create table，然后 insert。

24.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27.与临时表一样，游标并不是不可使用。对小型数据集使用

FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。

29.尽量避免大事务操作，提高系统并发能力。

30.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

"25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考虑改写。"
这是常见的说法，不知是什么场景。
我，恰恰是使用游标，进行千万级的批量插入，效率非常高。
因为，ORACLE OCI，进行批量插入，只能通过游标进行批量绑定，这也是 sqlldr 的做法。
游标一次打开多次使用，效率会非常高，性能超过你的想像。
通常，我们 1000 行一批，每批只需对游标 bind，execute，再加上多线程并行，再加上 RAC，速度无敌。
而且这一切都在一个框架的支撑下进行－－－－C 的 hibernate。。。
离了这个框架，这事还真不好做。

在 C 的世界里，游标可以用，hibernate 也可以用。

数据，其源头都是外来的。多数部门，不允许数据库间的直接访问。必须卸载－传输－加载。

我目前所在的系统就是这么一个有实时插入又需要大数据的查询的一个系统。

我们公司曾经想过用 hbase 开发过这样的系统， 高速写入同时，高速查询，查询的数据是有时间范围，不超过当前点的 5 分钟到 10 分钟。 我们考虑使用 hbase ， 主要 hbase 里有专门内存区域来存放数据，满了 32M/或者根据设定值就会写入到磁盘里面。 如果读磁盘里面的数据就会很慢，如果读内存的数据就会很快。根据业务，增加足够的节点机，保证 5-10 分种的数据缓存在 hbase 的内存中，这样就可以保证高速写入的同时也可以高速查询，这样的结构实际上取巧，使用了 hbase 作为一个小型内存数据库。
我们实现了高速写入。 但是由于其他原因没有做下去，很可惜。

一般大数据的维护多出现在 OLAP 环境中

对于大数据的维护

1、首选 表分区 + 压缩 采用“交换分区”操作，瞬间完成数据的转入转出，数据压缩能减少 I/0, 提高查询性能
2、在 DB2 中，可以采用 MDC，由于维度键值对相同的记录都被放在相邻的块

中，可以很快的进行删除

3、对于业务处理复杂的操作，可 采用“临时表”来存储中间处理过程，分段处理，大表变小表

大数据量的查询：

1、合理设计索引，尽量形成索引覆盖

2、采用簇表

3、使用 ETL 工具，在 DBMS 负载空闲时，将需要查询最终数据（汇总、统计）的结果计算后存入汇总、统计表，减少对大表的多次扫描！

并行多次查询，并且读写分离。

2.Java 的内存管理

JAVA 内存管理总结

1. java 是如何管理内存的

Java 的内存管理就是对象的分配和释放问题。（两部分）

分配 ：内存的分配是由程序完成的，程序员需要通过关键字 new 为每个对象申请内存空间 (基本类型除外)，所有的对象都在堆 (Heap)中分配空间。

释放 ：对象的释放是由垃圾回收机制决定和执行的，这样做确实简化

了程序员的工作。但同时，它也加重了 JVM 的工作。因为，GC 为了能够正确释放对象，GC 必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要进行监控。

2. 什么叫 java 的内存泄露

在 Java 中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连（也就是说仍存在该内存对象的引用）；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为 Java 中的内存泄漏，这些对象不会被 GC 所回收，然而它却占用内存。

3. JVM 的内存区域组成

java 把内存分两种：一种是栈内存，另一种是堆内存 1。在函数中定义的基本类型变量和对象的引用变量都在函数的栈内存中分配；2。堆内存用来存放由 new 创建的对象和数组以及对象的实例变量　在函数（代码块）中定义一个变量时，java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，java 会自动释放掉为该变量所分配的内存空间；在堆中分配的内存由 java 虚拟机的自动垃圾回收器来管理

堆和栈的优缺点

堆的优势是可以动态分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的。

缺点就是要在运行时动态分配内存，存取速度较慢； 栈的优势是，存取速度比堆要快，仅次于直接位于 CPU 中的寄存器。

另外，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。

4. Java 中数据在内存中是如何存储的

a) 基本数据类型

Java 的基本数据类型共有 8 种，即

int, short, long, byte, float, double, boolean, char(注意，并没有 string 的基本类型)。这种类型的定义是通过诸如 int a = 3； long b = 255L；的形式来定义的。如 int a = 3；这里的 a 是一个指向 int 类型的引用，指向 3 这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值定义在某个程序块里面，程序块退出后，字段值就消失了)，出于追求速度的原因，就存在于栈中。

另外，栈有一个很重要的特殊性，就是存在栈中的数据可以共享。比如：

我们同时定义：

```
int a=3;
int b =3;
```

编译器先处理 int a = 3；首先它会在栈中创建一个变量为 a 的引用，然后查找有没有字面值为 3 的地址，没找到，就开辟一个存放 3 这个字面值的地址，然后将 a 指向 3 的地址。接着处理 int b = 3；在创建完 b 这个引用变量后，由于在栈中已经有 3 这个字面值，便将 b 直接指向 3

的地址。这样，就出现了 a 与 b 同时均指向 3 的情况。　定义完 a 与 b 的值后，再令 a = 4；那么，b 不会等于 4，还是等于 3。在编译器内部，遇到时，它就会重新搜索栈中是否有 4 的字面值，如果没有，重新开辟地址存放 4 的值；如果已经有了，则直接将 a 指向这个地址。因此 a 值的改变不会影响到 b 的值。

b)　对象

在 Java 中，创建一个对象包括对象的声明和实例化两步，下面用一个例题来说明对象的内存模型。　　假设有类 Rectangle 定义如下：

```
public class Rectangle {
double width;
double height;
public Rectangle(double w,double h){
w = width;
h = height;
}
}
```

(1)声明对象时的内存模型

用 Rectangle rect；声明一个对象 rect 时，将在栈内存为对象的引用变量 rect 分配内存空间，但 Rectangle 的值为空，称 rect 是一个空对象。空对象不能使用，因为它还没有引用任何"实体"。

(2)对象实例化时的内存模型

当执行 rect=new Rectangle(3,5)；时，会做两件事：　在堆内存中为类的成员变量 width,height 分配内存，并将其初始化为各数据类型的默认值；接着进行显式初始化（类定义时的初始化值）；最后调用构造方

法，为成员变量赋值。 返回堆内存中对象的引用（相当于首地址）给引用变量 rect,以后就可以通过 rect 来引用堆内存中的对象了。

c) 创建多个不同的对象实例

一个类通过使用 new 运算符可以创建多个不同的对象实例，这些对象实例将在堆中被分配不同的内存空间，改变其中一个对象的状态不会影响其他对象的状态。例如：

```
Rectangle r1= new Rectangle(3,5);
Rectangle r2= new Rectangle(4,6);
```

此时，将在堆内存中分别为两个对象的成员变量 width、height 分配内存空间，两个对象在堆内存中占据的空间是互不相同的。如果有：

```
Rectangle r1= new Rectangle(3,5);
Rectangle r2=r1;
```

则在堆内存中只创建了一个对象实例，在栈内存中创建了两个对象引用，两个对象引用同时指向一个对象实例。

d) 包装类

基本型别都有对应的包装类：如 int 对应 Integer 类，double 对应 Double 类等，基本类型的定义都是直接在栈中，如果用包装类来创建对象，就和普通对象一样了。例如：int i=0；i 直接存储在栈中。 Integer i（i 此时是对象） = new Integer(5)；这样，i 对象数据存储在堆中，i 的引用存储在栈中，通过栈中的引用来操作对象。

e) String

String 是一个特殊的包装类数据。可以用用以下两种方式创建：

String str = new String("abc")；String str = "abc";

第一种创建方式，和普通对象的的创建过程一样；

第二种创建方式，Java 内部将此语句转化为以下几个步骤：

(1) 先定义一个名为 str 的对 String 类的对象引用变量：String str；

(2) 在栈中查找有没有存放值为"abc"的地址，如果没有，则开辟一个存放字面值为"abc"

地址，接着创建一个新的 String 类的对象 o，并将 o 的字符串值指向这个地址，而且在栈

这个地址旁边记下这个引用的对象 o。如果已经有了值为"abc"的地址，则查找对象 o，并

回 o 的地址。

(3) 将 str 指向对象 o 的地址。

值得注意的是，一般 String 类中字符串值都是直接存值的。但像

String str = "abc"；这种

合下，其字符串值却是保存了一个指向存在栈中数据的引用。

为了更好地说明这个问题，我们可以通过以下的几个代码进行验证。

String str1="abc"；

String str2="abc"；

System.out.println(s1==s2)；//true

注意，这里并不用 str1.equals(str2)；的方式，因为这将比较两个字符

串的值是否相等。==号，根据 JDK 的说明，只有在两个引用都指向了同一个对象时才返回真值。而我们在这里要看的是，str1 与 str2 是否都指向了同一个对象。

我们再接着看以下的代码。

```
String str1= new String("abc");
String str2="abc";
System.out.println(str1==str2); //false
```

创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。　　以上两段代码说明，只要是用 new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。

f)　数组

当定义一个数组，int x[]；或 int []x；时，在栈内存中创建一个数组引用，通过该引用（即数组名）来引用数组。x=new int[3]；将在堆内存中分配 3 个保存 int 型数据的空间，堆内存的首地址放到栈内存中，每个数组元素被初始化为 0。

g)　静态变量

用 static 的修饰的变量和方法，实际上是指定了这些变量和方法在内存中的"固定位置"－static storage，可以理解为所有实例对象共有的内存空间。static 变量有点类似于 C 中的全局变量的概念；静态表示

的是内存的共享，就是它的每一个实例都指向同一个内存地址。把 static 拿来，就是告诉 JVM 它是静态的，它的引用（含间接引用）都是指向同一个位置，在那个地方，你把它改了，它就不会变成原样，你把它清理了，它就不会回来了。　　那静态变量与方法是在什么时候初始化的呢？对于两种不同的类属性，static 属性与 instance 属性，初始化的时机是不同的。instance 属性在创建实例的时候初始化，static 属性在类加载，也就是第一次用到这个类的时候初始化，对于后来的实例的创建，不再次进行初始化。　　我们常可看到类似以下的例子来说明这个问题：

```
class Student{
static int numberOfStudents=0;
Student()
{
numberOfStudents++;
}
}
```

每一次创建一个新的 Student 实例时,成员 numberOfStudents 都会不断的递增,并且所有的 Student 实例都访问同一个 numberOfStudents 变量,实际上 int numberOfStudents 变量在内存中只存储在一个位置上。

5. Java 的内存管理实例

Java 程序的多个部分(方法，变量，对象)驻留在内存中以下两个位置：即堆和栈，现在我们只关心 3 类事物：实例变量，局部变量和对象：

实例变量和对象驻留在堆上

局部变量驻留在栈上

让我们查看一个 java 程序，看看他的各部分如何创建并且映射到栈和堆中：

```
public class Dog {
Collar c;
String name;
//1. main()方法位于栈上
public static void main(String[] args) {
//2. 在栈上创建引用变量 d,但 Dog 对象尚未存在
Dog d;
//3. 创建新的 Dog 对象，并将其赋予 d 引用变量
d = new Dog();
//4. 将引用变量的一个副本传递给 go()方法
d.go(d);
}
//5. 将 go()方法置于栈上，并将 dog 参数作为局部变量
void go(Dog dog){
//6. 在堆上创建新的 Collar 对象，并将其赋予 Dog 的实例变量
c =new Collar();
}
//7.将 setName()添加到栈上，并将 dogName 参数作为其局部变量
void setName(String dogName){
//8. name 的实例对象也引用 String 对象
name=dogName;
}
//9. 程序执行完成后，setName()将会完成并从栈中清除，此时，局部变量 dogName 也会消失，尽管它所引用的 String 仍在堆上
}
```

6. 垃圾回收机制：

（问题一：什么叫垃圾回收机制？） 垃圾回收是一种动态存储管理技术，它自动地释放不再被程序引用的对象，按照特定的垃圾收集算法来实现资源自动回收的功能。当一个对象不再被引用的时候，内存回收它占领的空间，以便空间被后来的新对象使用，以免造成内存泄露。（问题二：java 的垃圾回收有什么特点？）JAVA 语言不允许程序员直接控制内存空间的使用。内存空间的分配和回收都是由 JRE 负责在后台自动进行的，尤其是无用内存空间的回收操作(garbagecollection,也称垃圾回收)，只能由运行环境提供的一个超级线程进行监测和控制。 （问题三：垃圾回收器什么时候会运行？）一般是在 CPU 空闲或空间不足时自动进行垃圾回收，而程序员无法精确控制垃圾回收的时机和顺序等。 （问题四：什么样的对象符合垃圾回收条件？）当没有任何获得线程能访问一个对象时，该对象就符合垃圾回收条件。（问题五：垃圾回收器是怎样工作的？）垃圾回收器如发现一个对象不能被任何活线程访问时，他将认为该对象符合删除条件，就将其加入回收队列，但不是立即销毁对象，何时销毁并释放内存是无法预知的。垃圾回收不能强制执行，然而 Java 提供了一些方法（如：System.gc()方法），允许你请求 JVM 执行垃圾回收，而不是要求，虚拟机会尽其所能满足请求，但是不能保证 JVM 从内存中删除所有不用的对象。 （问题六：一个 java 程序能够耗尽内存吗？）可以。垃圾收集系统尝试在对象不被使用时把他们从内存中删除。然而，如果保持太多活的对象，系统则可能会耗尽内存。垃圾回收器不能保证有足够的内存，只能保证可用内

存尽可能的得到高效的管理。（问题七：如何显示的使对象符合垃圾回收条件？）（1） 空引用 ：当对象没有对他可到达引用时，他就符合垃圾回收的条件。也就是说如果没有对他的引用，删除对象的引用就可以达到目的，因此我们可以把引用变量设置为 null，来符合垃圾回收的条件。

```
StringBuffer sb = new StringBuffer("hello");
System.out.println(sb);
sb=null;
```

（2） 重新为引用变量赋值：可以通过设置引用变量引用另一个对象来解除该引用变量与一个对象间的引用关系。

```
StringBuffer sb1 = new StringBuffer("hello");
StringBuffer sb2 = new StringBuffer("goodbye");
System.out.println(sb1);
sb1=sb2;//此时"hello"符合回收条件
```

（3） 方法内创建的对象：所创建的局部变量仅在该方法的作用期间内存在。一旦该方法返回，在这个方法内创建的对象就符合垃圾收集条件。有一种明显的例外情况，就是方法的返回对象。

```
public static void main(String[] args) {
Date d = getDate();
System.out.println("d = " + d);
}
private static Date getDate() {
Date d2 = new Date();
StringBuffer now = new StringBuffer(d2.toString());
System.out.println(now);
return d2;
}
```

（4） 隔离引用：这种情况中，被回收的对象仍具有引用，这种情况

称作隔离岛。若存在这两个实例，他们互相引用，并且这两个对象的所有其他引用都删除，其他任何线程无法访问这两个对象中的任意一个。也可以符合垃圾回收条件。

```
public class Island {
Island i;
public static void main(String[] args) {
Island i2 = new Island();
Island i3 = new Island();
Island i4 = new Island();
i2.i=i3;
i3.i=i4;
i4.i=i2;
i2=null;
i3=null;
i4=null;
}
}
```

（问题八：垃圾收集前进行清理------finalize()方法）java 提供了一种机制，使你能够在对象刚要被垃圾回收之前运行一些代码。这段代码位于名为 finalize()的方法内，所有类从 Object 类继承这个方法。由于不能保证垃圾回收器会删除某个对象。因此放在 finalize()中的代码无法保证运行。因此建议不要重写 finalize();

7. final 问题：

final 使得被修饰的变量"不变"，但是由于对象型变量的本质是"引用"，使得"不变"也有了两种含义：引用本身的不变?，和引用指向的对象不变。? 引用本身的不变：

```
final StringBuffer a=new StringBuffer("immutable");
final StringBuffer b=new StringBuffer("not immutable");
```

```
a=b;//编译期错误
final StringBuffer a=new StringBuffer("immutable");
final StringBuffer b=new StringBuffer("not immutable");
a=b;//编译期错误
```

引用指向的对象不变：

```
final StringBuffer a=new StringBuffer("immutable");
a.append(" broken!"); //编译通过
final StringBuffer a=new StringBuffer("immutable");
a.append(" broken!"); //编译通过
```

可见，final 只对引用的"值"(也即它所指向的那个对象的内存地址)有效，它迫使引用只能指向初始指向的那个对象，改变它的指向会导致编译期错误。至于它所指向的对象的变化，final 是不负责的。这很类似==操作符：==操作符只负责引用的"值"相等，至于这个地址所指向的对象内容是否相等，==操作符是不管的。在举一个例子：

```
public class Name {
private String firstname;
private String lastname;
public String getFirstname() {
return firstname;
}
public void setFirstname(String firstname) {
this.firstname = firstname;
}
public String getLastname() {
return lastname;
}
public void setLastname(String lastname) {
this.lastname = lastname;
}
```

```
}

public class Name {
private String firstname;
private String lastname;
public String getFirstname() {
return firstname;
}
public void setFirstname(String firstname) {
this.firstname = firstname;
}
public String getLastname() {
return lastname;
}
public void setLastname(String lastname) {
this.lastname = lastname;
}
}
```

编写测试方法：

```
public static void main(String[] args) {
final Name name = new Name();
name.setFirstname("JIM");
name.setLastname("Green");
System.out.println(name.getFirstname()+" "+name.getLastname());
}
public static void main(String[] args) {
final Name name = new Name();
name.setFirstname("JIM");
name.setLastname("Green");
System.out.println(name.getFirstname()+" "+name.getLastname());
}
```

理解 final 问题有很重要的含义。许多程序漏洞都基于此----final

只能保证引用永远指向固定对象，不能保证那个对象的状态不变。在多线程的操作中,一个对象会被多个线程共享或修改，一个线程对对象无意识的修改可能会导致另一个使用此对象的线程崩溃。一个错误的解决方法就是在此对象新建的时候把它声明为 final，意图使得它"永远不变"。其实那是徒劳的。　　Final 还有一个值得注意的地方：　　先看以下示例程序：

```
class Something {
final int i;
public void doSomething() {
System.out.println("i = " + i);
}
}
class Something {
final int i;
public void doSomething() {
System.out.println("i = " + i);
}
}
```

对于类变量，Java 虚拟机会自动进行初始化。如果给出了初始值，则初始化为该初始值。如果没有给出，则把它初始化为该类型变量的默认初始值。但是对于用 final 修饰的类变量，虚拟机不会为其赋予初值，必须在 constructor (构造器)结束之前被赋予一个明确的值。可以修改为"final int i = 0;"。

8.　如何把程序写得更健壮：

1、尽早释放无用对象的引用。好的办法是使用临时变量的时候，让引用变量在退出活动域后，自动设置为 null，暗示垃圾收集器来收集该

对象，防止发生内存泄露。对于仍然有指针指向的实例，jvm 就不会回收该资源,因为垃圾回收会将值为 null 的对象作为垃圾，提高 GC 回收机制效率；

2、定义字符串应该尽量使用 String str="hello"; 的形式，避免使用 String str = new String("hello"); 的形式。因为要使用内容相同的字符串，不必每次都 new 一个 String。例如我们要在构造器中对一个名叫 s 的 String 引用变量进行初始化，把它设置为初始值，应当这样做：

```
public class Demo {
private String s;
public Demo() {
s = "Initial Value";
}
}

public class Demo {
private String s;
...
public Demo {
s = "Initial Value";
}
...
}
```

而非

```
s = new String("Initial Value");
s = new String("Initial Value");
```

后者每次都会调用构造器，生成新对象，性能低下且内存开销大，并且没有意义，因为 String 对象不可改变，所以对于内容相同的字符串，

只要一个 String 对象来表示就可以了。也就说，多次调用上面的构造器创建多个对象，他们的 String 类型属性 s 都指向同一个对象。

3、我们的程序里不可避免大量使用字符串处理，避免使用 String，应大量使用 StringBuffer ，因为 String 被设计成不可变(immutable)类，所以它的所有对象都是不可变对象，请看下列代码；

```
String s = "Hello";
s = s + " world!";
String s = "Hello";
s = s + " world!";
```

在这段代码中，s 原先指向一个 String 对象，内容是 "Hello"，然后我们对 s 进行了+操作，那么 s 所指向的那个对象是否发生了改变呢？答案是没有。这时，s 不指向原来那个对象了，而指向了另一个 String 对象，内容为"Hello world!"，原来那个对象还存在于内存之中，只是 s 这个引用变量不再指向它了。 通过上面的说明，我们很容易导出另一个结论，如果经常对字符串进行各种各样的修改，或者说，不可预见的修改，那么使用 String 来代表字符串的话会引起很大的内存开销。因为 String 对象建立之后不能再改变，所以对于每一个不同的字符串，都需要一个 String 对象来表示。这时，应该考虑使用 StringBuffer 类，它允许修改，而不是每个不同的字符串都要生成一个新的对象。并且，这两种类的对象转换十分容易。

4、尽量少用静态变量 ，因为静态变量是全局的，GC 不会回收的；

5、尽量避免在类的构造函数里创建、初始化大量的对象 ，防止在调用其自身类的构造器时造成不必要的内存资源浪费，尤其是大对象，

JVM 会突然需要大量内存，这时必然会触发 GC 优化系统内存环境；显示的声明数组空间，而且申请数量还极大。　　以下是初始化不同类型的对象需要消耗的时间：

运算操作	示例	标准化时间
本地赋值	i = n	1.0
实例赋值	this.i = n	1.2
方法调用	Funct()	5.9
新建对象	New Object()	980
新建数组	New int[10]	3100

从表 1 可以看出，新建一个对象需要 980 个单位的时间，是本地赋值时间的 980 倍，是方法调用时间的 166 倍，而新建一个数组所花费的时间就更多了。

6、尽量在合适的场景下使用对象池技术 以提高系统性能，缩减缩减开销，但是要注意对象池的尺寸不宜过大，及时清除无效对象释放内存资源，综合考虑应用运行环境的内存资源限制，避免过高估计运行环境所提供内存资源的数量。

7、大集合对象拥有大数据量的业务对象的时候，可以考虑分块进行处理 ，然后解决一块释放一块的策略。

8、不要在经常调用的方法中创建对象 ，尤其是忌讳在循环中创建对象。可以适当的使用 hashtable，vector 创建一组对象容器，然后从容

器中去取那些对象，而不用每次 new 之后又丢弃。

9、一般都是发生在开启大型文件或跟数据库一次拿了太多的数据，造成 Out Of Memory Error 的状况，这时就大概要计算一下数据量的最大值是多少，并且设定所需最小及最大的内存空间值。

10、尽量少用 finalize 函数，因为 finalize()会加大 GC 的工作量，而 GC 相当于耗费系统的计算能力。

11、不要过滥使用哈希表，有一定开发经验的开发人员经常会使用 hash 表（hash 表在 JDK 中的一个实现就是 HashMap）来缓存一些数据，从而提高系统的运行速度。比如使用 HashMap 缓存一些物料信息、人员信息等基础资料，这在提高系统速度的同时也加大了系统的内存占用，特别是当缓存的资料比较多的时候。其实我们可以使用操作系统中的缓存的概念来解决这个问题，也就是给被缓存的分配一个一定大小的缓存容器，按照一定的算法淘汰不需要继续缓存的对象，这样一方面会因为进行了对象缓存而提高了系统的运行效率，同时由于缓存容器不是无限制扩大，从而也减少了系统的内存占用。现在有很多开源的缓存实现项目，比如 ehcache、oscache 等，这些项目都实现了 FIFO、MRU 等常见的缓存算法

3.操作系统的内存管理

4.怎么管理内存泄漏

5.进程和线程的区别

进程就是一个应用程在处理机上的一次执行过程， 它是一个动态的概念。而线程是进程的一部分，一个进程可以包含多个线程.

进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的代码，还包括当前的活动，通过程序计数器的值和处理寄存器的内容来表示。

进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时，它才能成为一个活动的实体，我们称其为进程。

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度。

线程和进程的区别在于，子进程和父进程有不同的代码和数据空间，而多个线程则共享数据空间，每个线程有自己的执行堆栈和程序计数器为其执行上下文。多线程主要是为了节约 CPU 时间，发挥利用，根据具体情况而定。线程的运行中需要使用计算机的内存资源和 CPU。

线程与进程的区别归纳：

a.地址空间和其它资源：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。

b.通信：进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

c.调度和切换：线程上下文切换比进程上下文切换要快得多。

d.在多线程 OS 中，进程不是一个可执行的实体。

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。线程是进程的一个实体,是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

6.寻址的内存开销