重点篇

MySQL 重点

1. 锁

MyISAM 和 InnoDB 存储引擎使用的锁：
MyISAM 采用表级锁(table-level locking)。
InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁
表级锁和行级锁对比：
表级锁： Mysql 中锁定 粒度最大 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM 和 InnoDB 引擎都支持表级锁。
行级锁： Mysql 中锁定 粒度最小 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。
InnoDB 存储引擎的锁的算法有三种：
Record lock：单个行记录上的锁
Gap lock：间隙锁，锁定一个范围，不包括记录本身
Next-key lock：record+gap 锁定一个范围，包含记录本身

2.大表优化

当 MySQL 单表记录数过大时，数据库的 CRUD 性能会明显下降，一些常见的优化措施如下：

限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。；

读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；

缓存： 使用 MySQL 的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的缓存；

垂直分区：

根据数据库里面数据表的相关性进行拆分。 例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。 如下图所示，这样来说大家应该就更容易理解了。

垂直拆分的优点： 可以使得行数据变小，在查询时减少读取的 Block 数，减少 I/O 次数。此外，垂直分区可以简化表的结构，易于维护。

垂直拆分的缺点： 主键会出现冗余，需要管理冗余列，并会引起 Join 操作，可以通过在应用层进行 Join 来解决。此外，垂直分区会让事务变得更加复杂。

水平分区：

保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。

水平拆分是指数据表行的拆分，表的行数超过 200 万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。

水品拆分可以支持非常大的数据量。需要注意的一点是:分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升 MySQL 并发能力没有什么意义，所以 水品拆分最好分库 。

水平拆分能够 支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决 ，跨界点 Join 性能较差，逻辑复杂。《Java 工程师修炼之道》的作者推荐 尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度 ，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络 I/O。

3.MySQL 的复制原理以及流程

基本原理流程，3 个线程以及之间的关联；

1. 主：binlog 线程——记录下所有改变了数据库数据的语句，放进 master 上的 binlog 中；

2. 从：io 线程——在使用 start slave 之后，负责从 master 上拉取 binlog 内容，放进 自己的 relay log 中；

3. 从：sql 执行线程——执行 relay log 中的语句。

详解：mysql 主从复制

MySQL 数据库自身提供的主从复制功能可以方便的实现数据的多处自动备份，实现数据库的拓展。多个数据备份不仅可以加强数据的安全性，通过实现读写分离还能进一步提升数据库的负载性能。

在一主多从的数据库体系中，多个从服务器采用异步的方式更新主数据库的变化，业务服务器在执行写或者相关修改数据库的操作是在主服务器上进行的，读操作则是在各从服务器上进行。如果配置了多个从服务器或者多个主服务器又涉及到相应的负载均衡问题，关于负载均衡具体的技术细节还没有研究过，今天就先简单的实现一主一从的主从复制功能。

4. Mysql 中的 myisam 与 innodb 的区别

1、InooDB 支持事务，而 MyISAM 不支持事务

2、InnoDB 支持行级锁，而 MyISAM 支持表级锁

3、InnoDB 支持 MVCC，而 MyISAM 不支持

4、InnoDB 支持外键，而 MyISAM 不支持

5、InnoDB 不支持全文索引，而 MyISAM 支持

<2>InnoDB 引擎的四大特性：插入缓冲，二次写，自适应哈希索引，预读

<3>InooDB 和 MyISAM 的 select count（*）哪个更快，为什么

myisam 更快，因为 myisam 内部维护了一个计算器，可以直接调取。MyISAM 的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而 Innodb 是索引和数据是紧密捆绑的，没有使用压缩从而会造成 Innodb 比 MyISAM 体积庞大

不小。

5. InnoDB 的事务与日志的实现方式

错误日志：记录出错信息，也记录一些警告信息或者正确的信息

查询日志：记录所有对数据库请求的信息，不论这些请求是否得到了正确的执行

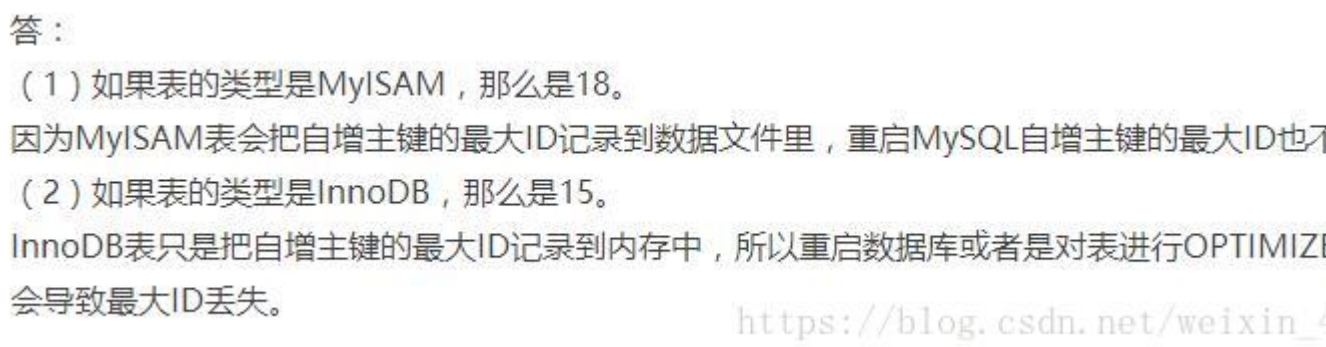
慢查询日志：设置一个阈值，将运行时间超过该值的所有 SQL 语句都记录到慢查询的日志文件中

二进制日志：记录对数据库执行更改的所有操作

中继日志，事务日志。

6. 一张表里面有 ID 自增主键

一张表里面有 ID 自增主键，当 insert 了 17 条记录之后，删除了第 15,16,17 条记录，再把 mysql 重启，再 insert 一条记录，这条记录的 ID 是 18 还是 15 ？



7.哈希索引的优势

等值查询。哈希索引具有绝对优势（前提是：没有大量重复键值，如果大量重复键值时，哈希索引的效率很低，因为存在所谓的哈希碰撞问题。）

哈希索引不适用的场景：

不支持范围查询

不支持索引完成排序

不支持联合索引的最左前缀匹配规则

通常，B+树索引结构适用于绝大多数场景，像下面这种场景用哈希索引才更有优势：

在 HEAP 表中，如果存储的数据重复度很低（也就是说基数很大），对该列数据以等值查询为主，没有范围查询、没有排序的时候，特别适合采用哈希索引。

而常用的 InnoDB 引擎中默认使用的是 B+树索引，它会实时监控表上索引的使用情况，如果认为建立哈希索引可以提高查询效率，则自动在内存中的“自适应哈希索引缓冲区”建立哈希索引（在 InnoDB 中默认开启自适应哈希索引），通过观察搜索模式，MySQL 会利用 index key 的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快等值查询。

注意：在某些工作负载下，通过哈希索引查找带来的性能提升远大于额外的监控索引搜索情况和保持这个哈希表结构所带来的开销。但某些时候，在负载高的情况下，自适应哈希索引中添加的 read/write 锁也会带来竞争，比如高并发的 join 操作。like 操作和%的通配符操作也不适用于自适应哈希索引，可能要关闭自适应哈希索引。

8.B tree/B+tree

1、B 树，每个节点都存储 key 和 data，所有的节点组成这可树，并且叶子节点指针为 null，叶子节点不包含任何关键字信息

2、B+树，所有的叶子节点中包含全部关键字的信息，及指向含有这些关键字记录的指针，且叶子节点本身依关键字的大小自小到大的顺序链接，所有的非终端节点可以看成是索引部分，节点中仅含有其子树根节点中最大（或最小）关键字

3、为什么说 B+比 B 树更适合实际应用中操作系统的文件索引和数据库索引？

B+的磁盘读写代价更低 B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了。

B+-tree 的查询效率更加稳定 由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

9. 数据库范式

1 第一范式(1NF)

在任何一个关系数据库中，第一范式(1NF)是对关系模式的基本要求，不满足第一范式(1NF)的数据库就不是关系数据库。

所谓第一范式(1NF)是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式(1NF)中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

2 第二范式(2NF)

第二范式(2NF)是在第一范式(1NF)的基础上建立起来的，即满足第二范式(2NF)必须先满足第一范式(1NF)。第二范式(2NF)要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键、主码。

第二范式(2NF)要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，第二范式就是非主属性非部分依赖于主关键字。

3 第三范式(3NF)

满足第三范式(3NF)必须先满足第二范式(2NF)。简而言之，第三范式(3NF)要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每

个部门有部门编号(dept_id)、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式(3NF)也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性。(我的理解是消除冗余)

10.如何设计一个高并发的系统

① 数据库的优化，包括合理的事务隔离级别、SQL 语句优化、索引的优化
② 使用缓存，尽量减少数据库 IO
③ 分布式数据库、分布式缓存
④ 服务器的负载均衡

11.锁的优化策略

① 读写分离
② 分段加锁
③ 减少锁持有的时间
④ 多个线程尽量以相同的顺序去获取资源
等等，这些都不是绝对原则，都要根据情况，比如不能将锁的粒度过于细化，不然可能会出现线程的加锁和释放次数过多，反而效率不如一次加一把大锁。

12.了解 XSS 攻击吗？如何防止？

XSS 是跨站脚本攻击，首先是利用跨站脚本漏洞以一个特权模式去执行攻击者构造的脚本，然后利用不安全的 Activex 控件执行恶意的行为。
使用 htmlspecialchars()函数对提交的内容进行过滤，使字符串里面的特殊符号实体化。

13.SQL 注入漏洞产生的原因？如何防止？

SQL 注入产生的原因：程序开发过程中不注意规范书写 sql 语句和对特殊字符进行过滤，导致客户端可以通过全局变量 POST 和 GET 提交一些 sql 语句正常执行。
防止 SQL 注入的方式：
开启配置文件中的 magic_quotes_gpc 和 magic_quotes_runtime 设置
执行 sql 语句时使用 addslashes 进行 sql 语句转换
Sql 语句书写尽量不要省略双引号和单引号。
过滤掉 sql 语句中的一些关键词：update、insert、delete、select、 * 。
提高数据库表和字段的命名技巧，对一些重要的字段根据程序的特点命名，取不易被猜到的。
Php 配置文件中设置 register_globals 为 off,关闭全局变量注册
控制错误信息，不要在浏览器上输出错误信息，将错误信息写到日志文件中。

14. 存储时期

Datatime:以 YYYY-MM-DD HH:MM:SS 格式存储时期时间，精确到秒，占用 8 个字节得存储空间，datatime 类型与时区无关

Timestamp:以时间戳格式存储，占用 4 个字节，范围小 1970-1-1 到 2038-1-19，显示依赖于所指定得时区，默认在第一个列行的数据修改时可以自动得修改 timestamp 列得值

Date:（生日）占用得字节数比使用字符串.datatime.int 储存要少，使用 date 只需要 3 个字节，存储日期月份，还可以利用日期时间函数进行日期间得计算

Time:存储时间部分得数据

注意:不要使用字符串类型来存储日期时间数据（通常比字符串占用得储存空间小，在进行查找过滤可以利用日期得函数）

使用 int 存储日期时间不如使用 timestamp 类型

JVM 重点

15.运行时数据区域

Java 虚拟机管理的内存包括几个运行时数据内存：方法区、虚拟机栈、本地方法栈、堆、程序计数器，其中方法区和堆是由线程共享的数据区，其他几个是线程隔离的数据区

1.1 程序计数器

程序计数器是一块较小的内存，他可以看做是当前线程所执行的行号指示器。字节码解释器工作的时候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的指令，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。如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器则为空。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemotyError 情况的区域

1.2 Java 虚拟机栈

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储存局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

栈内存就是虚拟机栈，或者说是虚拟机栈中局部变量表的部分

局部变量表存放了编辑期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（refrence）类型和 returnAddress 类型（指向了一条字节码指令的地址）

其中 64 位长度的 long 和 double 类型的数据会占用两个局部变量空间，其余的数据类型只占用 1 个。

Java 虚拟机规范对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常。如果虚拟机扩展时无法申请到足够的内存，就会跑出 OutOfMemoryError 异常

1.3 本地方法栈

本地方法栈和虚拟机栈发挥的作用是非常类似的，他们的区别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

本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryErroy 异常

1.4 Java 堆

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，在虚拟机启动的时候创建，此内存区域的唯一目的是存放对象实例，几乎所有的对象实例都在这里分配内存。所有的对象实例和数组都在堆上分配

Java 堆是垃圾收集器管理的主要区域。Java 堆细分为新生代和老年代

不管怎样，划分的目的都是为了更好的回收内存，或者更快地分配内存

Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有完成实例分配，并且堆也无法在扩展时将会抛出 OutOfMemoryError 异常

1.5 方法区

方法区它用于储存已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

除了 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载

当方法区无法满足内存分配需求时，将抛出 OutOfMemoryErroy 异常

1.6 运行时常量池

它是方法区的一部分。Class 文件中除了有关的版本、字段、方法、接口等描述信息外、还有一项信息是常量池，用于存放编辑期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放

Java 语言并不要求常量一定只有编辑期才能产生，也就是可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern()方法

当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常

16.hotspot 虚拟机对象

2.1 对象的创建

1）.检查

虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程

2）分配内存

接下来将为新生对象分配内存，为对象分配内存空间的任务等同于把一块确定的大小的内存从 Java 堆中划分出来。

假设 Java 堆中内存是绝对规整的，所有用过的内存放在一遍，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指向空闲空间那边挪动一段与对象大小相等的距离，这个分配方式叫做“指针碰撞”

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，这种分配方式成为“空闲列表”

选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

3）Init

执行 new 指令之后会接着执行 Init 方法，进行初始化，这样一个对象才算产生出来

2.2 对象的内存布局

在 HotSpot 虚拟机中，对象在内存中储存的布局可以分为 3 块区域：对象头、实例数据和对齐填充

对象头包括两部分：

1). 储存对象自身的运行时数据，如哈希码、GC 分带年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳

2). 另一部分是指类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例

2.3 对象的访问定位

1).使用句柄访问

Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址

优势:reference 中存储的是稳点的句柄地址,在对象被移动(垃圾收集时移动对象是非常普遍的行为)时只会改变句柄中的实例数据指针，而 reference 本身不需要修改

2).使用直接指针访问

Java 堆对象的布局就必须考虑如何访问类型数据的相关信息,而 refreence 中存储的直接就是对象的地址

优势：速度更快，节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本

17.OutOfMemoryError 异常

3.1 Java 堆溢出

Java 堆用于存储对象实例，只要不断的创建对象，并且保证 GCRoots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在数量到达最大堆的容量限制后就会产生内存溢出异常

如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄露对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息及 GC Roots 引用链的信息，就可以比较准确地定位出泄漏代码的位置

如果不存在泄露，换句话说，就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗

3.2 虚拟机栈和本地方法栈溢出

对于 HotSpot 来说，虽然-Xoss 参数（设置本地方法栈大小）存在，但实际上是无效的，栈容量只由-Xss 参数设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常

在单线程下，无论由于栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟

机抛出的都是 StackOverflowError 异常

如果是多线程导致的内存溢出，与栈空间是否足够大并不存在任何联系，这个时候每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。解决的时候是在不能减少线程数或更换 64 为的虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程

3.3 方法区和运行时常量池溢出

String.intern()是一个 Native 方法，它的作用是：如果字符串常量池中已经包含一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用

由于常量池分配在永久代中，可以通过-XX:PermSize 和-XX:MaxPermSize 限制方法区大小，从而间接限制其中常量池的容量。

Intern():

JDK1.6 intern 方法会把首次遇到的字符串实例复制到永久代，返回的也是永久代中这个字符串实例的引用，而由 StringBuilder 创建的字符串实例在 Java 堆上，所以必然不是一个引用

JDK1.7 intern()方法的实现不会再复制实例，只是在常量池中记录首次出现的实例引用，因此 intern()返回的引用和由 StringBuilder 创建的那个字符串实例是同一个。

18.垃圾收集

程序计数器、虚拟机栈、本地方法栈 3 个区域随线程而生，随线程而灭，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了

1.判断对象存活

4.1.1 引用计数器法

给对象添加一个引用计数器，每当由一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器为 0 的对象就是不可能再被使用的

4.1.2 可达性分析算法

通过一系列的成为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到 GC ROOTS 没有任何引用链相连时，则证明此对象时不可用的

Java 语言中 GC Roots 的对象包括下面几种：

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 JNI（Native 方法）引用的对象

2.引用

强引用就是在程序代码之中普遍存在的，类似 Object obj = new Object() 这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象

软引用用来描述一些还有用但并非必须的元素。对于它在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存才会抛出内存溢出异常

弱引用用来描述非必须对象的，但是它的强度比软引用更弱一些，被引用关联的对象只能生存到下一次垃圾收集发生之前，当垃圾收集器工作时，无论当前内存是否足够都会回收

掉只被弱引用关联的对象

虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知

3.Finalize 方法

任何一个对象的 finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize()方法不会被再次执行，因此第二段代码的自救行动失败了

4.3.1 回收方法区

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类

废弃常量：假如一个字符串 abc 已经进入了常量池中，如果当前系统没有任何一个 String 对象 abc，也就是没有任何 Stirng 对象引用常量池的 abc 常量，也没有其他地方引用的这个字面量，这个时候发生内存回收这个常量就会被清理出常量池

无用的类：

1.该类所有的实例都已经被回收，就是 Java 堆中不存在该类的任何实例

2.加载该类的 ClassLoader 已经被回收

3.该类对用的 java.lang.Class 对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法

4.垃圾收集算法

4.4.1 标记—清除算法

算法分为标记和清除两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象、

不足:一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清楚之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后再程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作

4.4.2 复制算法

他将可用内存按照容量划分为大小相等的两块，每次只使用其中的一块。当这块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可

不足：将内存缩小为了原来的一半

实际中我们并不需要按照 1:1 比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor

当另一个 Survivor 空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代

4.4.3 标记整理算法

让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

4.4.4 分代收集算法

只是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

5.垃圾收集器

a)Serial 收集器：

这个收集器是一个单线程的收集器，但它的单线程的意义不仅仅说明它会只使用一个 COU 或一条收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集时，必须暂停其

他所有的工作线程，直到它手机结束

b)ParNew 收集器：

Serial 收集器的多线程版本，除了使用了多线程进行收集之外，其余行为和 Serial 收集器一样

并行：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态

并发：指用户线程与垃圾收集线程同时执行（不一定是并行的，可能会交替执行），用户程序在继续执行，而垃圾收集程序运行于另一个 CPU 上

c)Parallel Scavenge

收集器是一个新生代收集器，它是使用复制算法的收集器，又是并行的多线程收集器。

吞吐量：就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值。即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）

d)Serial Old 收集器：

是 Serial 收集器的老年代版本,是一个单线程收集器，使用标记整理算法

e)Parallel Old 收集器：

Parallel Old 是 Paraller Seavenge 收集器的老年代版本，使用多线程和标记整理算法

f)CMS 收集器：

CMS 收集器是基于标记清除算法实现的，整个过程分为 4 个步骤：

1.初始标记 2.并发标记 3.重新标记 4.并发清除

优点：并发收集、低停顿

缺点：

1.CMS 收集器对 CPU 资源非常敏感，CMS 默认启动的回收线程数是（CPU 数量+3）/4，

2.CMS 收集器无法处理浮动垃圾，可能出现 Failure 失败而导致一次 Full G 场地产生

3.CMS 是基于标记清除算法实现的

g)G1 收集器：

它是一款面向服务器应用的垃圾收集器

1.并行与并发：利用多 CPU 缩短 STOP-The-World 停顿的时间

2.分代收集

3.空间整合：不会产生内存碎片

4.可预测的停顿

运作方式：初始标记，并发标记，最终标记，筛选回收

6.内存分配与回收策略

4.6.1 对象优先在 Eden 分配：

大多数情况对象在新生代 Eden 区分配，当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC

4.6.2 大对象直接进入老年代：

所谓大对象就是指需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串以及数组。这样做的目的是避免 Eden 区及两个 Servivor 之间发生大量的内存复制

4.6.3 长期存活的对象将进入老年代

如果对象在 Eden 区出生并且尽力过一次 Minor GC 后仍然存活，并且能够被 Servivor 容纳，将被移动到 Servivor 空间中，并且把对象年龄设置成为 1.对象在 Servivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认 15 岁），就将会被晋级到老年代中

4.6.4 动态对象年龄判定

为了更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了 MaxTenuringThreshold 才能晋级到老年代，如果在 Servivor 空间中相同年龄所有对象的大小总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入到老年代，无须登到 MaxTenuringThreshold 中要求的年龄

4.6.4 空间分配担保：

在发生 Minor GC 之前，虚拟机会检查老年代最大可 用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么 Minor DC 可以确保是安全的。如果不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。如果允许那么会继续检查老年代最大可用的连续空间是否大于晋级到老年代对象的平均大小，如果大于，将尝试进行一次 Minor GC，尽管这次 MinorGC 是有风险的：如果小于，或者 HandlePromotionFailure 设置不允许冒险，那这时也要改为进行一次 Full GC

19.虚拟机类加载机制

虚拟机吧描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制

在 Java 语言里面，类型的加载。连接和初始化过程都是在程序运行期间完成的

5.1 类加载的时机

类被加载到虚拟机内存中开始，到卸载为止，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

加载、验证、准备、初始化和卸载这 5 个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以再初始化阶段之后再开始，这个是为了支持 Java 语言运行时绑定（也成为动态绑定或晚期绑定）

虚拟机规范规定有且只有 5 种情况必须立即对类进行初始化：

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.当虚拟机启动时候，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

5.当使用 JDK1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化

被动引用：

1.通过子类引用父类的静态字段，不会导致子类初始化

2.通过数组定义来引用类，不会触发此类的初始化

3.常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化

接口的初始化：接口在初始化时，并不要求其父接口全部完成类初始化，只有在正整使用到父接口的时候（如引用接口中定义的常量）才会初始化

5.2 类加载的过程

5.2.1 加载

1)通过一个类的全限定名类获取定义此类的二进制字节流

2)将这字节流所代表的静态存储结构转化为方法区运行时数据结构

3)在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口

怎么获取二进制字节流？

1)从 ZIP 包中读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础

2)从网络中获取，这种场景最典型的应用就是 Applet

3)运行时计算生成，这种常见使用得最多的就是动态代理技术

4)由其他文件生成，典型场景就是 JSP 应用

5)从数据库中读取，这种场景相对少一些（中间件服务器）

数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的

数组类的创建过程遵循以下规则：

1)如果数组的组件类型(指的是数组去掉一个维度的类型)是引用类型，那就递归采用上面的加载过程去加载这个组件类型，数组 C 将在加载该组件类型的类加载器的类名称空间上被标识

2)如果数组的组件类型不是引用类型(列如 int[]组数)，Java 虚拟机将会把数组 C 标识为与引导类加载器关联

3)数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为 public

5.2.2 验证

验证阶段会完成下面 4 个阶段的检验动作：文件格式验证，元数据验证，字节码验证，符号引用验证

1.文件格式验证

第一阶段要验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括：

1.是否以魔数 oxCAFEBABE 开头

2.主、次版本号是否在当前虚拟机处理范围之内

3.常量池的常量中是否有不被支持的常量类型(检查常量 tag 标志)

4.指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量

5.CONSTANT_Itf8_info 型的常量中是否有不符合 UTF8 编码的数据

6.Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息

这个阶段的验证时基于二进制字节流进行的，只有通过类这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面的 3 个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流

2.元数据验证

1.这个类是否有父类(除了 java.lang.Object 之外,所有的类都应当有父类)

2.这个类的父类是否继承了不允许被继承的类（被 final 修饰的类）

3.如果这个类不是抽象类，是否实现类其父类或接口之中要求实现的所有方法

4.类中的字段、方法是否与父类产生矛盾(列如覆盖类父类的 final 字段,或者出现不符合规则的方法重载，列如方法参数都一致，但返回值类型却不同等)

第二阶段的主要目的是对类元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息

3.字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要目的似乎通过数据流和控制流分析，确定程序语言是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件。

1.保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，列如，列如在操作数栈放置类一个 int 类型的数据，使用时却按 long 类型来加载入本地变量表中

2.保证跳转指令不会跳转到方法体以外的字节码指令上

3.保证方法体中的类型转换时有效的，列如可以把一个子类对象赋值给父类数据类型，这个是安全的，但是吧父类对象赋值给子类数据类型，甚至把对象赋值给与它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的

4.符号引用验证

发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。

1.符号引用中通过字符串描述的全限定名是否能找到相对应的类

2.在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段

3.符号引用中的类、字段、方法的访问性是否可被当前类访问

对于虚拟机的类加载机制来说，验证阶段是非常重要的，但是不一定必要（因为对程序运行期没有影响）的阶段。如果全部代码都已经被反复使用和验证过，那么在实施阶段就可以考虑使用 Xverify：none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间

5.2.3 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，这些变量都在方法区中进行分配。这个时候进行内存分配的仅包括类变量(被 static 修饰的变量)，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。其次，这里说的初始值通常下是数据类型的零值。

假设 public static int value = 123；

那变量 value 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把 value 赋值为 123 的动作将在初始化阶段才会执行，但是如果使用 final 修饰，则在这个阶段其初始值设置为 123

5.2.4 解析

解析阶段是虚拟机将常量池内符号引用替换为直接引用的过

5.2.5 初始化

类的初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才正真开始执行类中定义的 Java 程序代码(或者说是字节码)

5.3 类的加载器

5.3.1 双亲委派模型：

只存在两种不同的类加载器：启动类加载器（Bootstrap ClassLoader），使用 C++实现，是虚拟机自身的一部分。另一种是所有其他的类加载器，使用 JAVA 实现，独立于 JVM，并且全部继承自抽象类 java.lang.ClassLoader.

启动类加载器（Bootstrap ClassLoader），负责将存放在<JAVA+HOME>\lib 目录中的，或者被-Xbootclasspath 参数所制定的路径中的，并且是 JVM 识别的（仅按照文件名识别，如 rt.jar，如果名字不符合，即使放在 lib 目录中也不会被加载），加载到虚拟机内存中，启动类加载器无法被 JAVA 程序直接引用。

扩展类加载器，由 sun.misc.Launcher$ExtClassLoader 实现，负责加载<JAVA_HOME>\lib\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader），由 sun.misc.Launcher$AppClassLoader 来实现。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader()方法的返回值，所以一般称它为系统类加载器。负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

这张图表示类加载器的双亲委派模型（Parents Delegation model）. 双亲委派模型要求除了顶层的启动加载类外，其余的类加载器都应当有自己的父类加载器。，这里类加载器之间的父子关系一般不会以继承的关系来实现，而是使用组合关系来复用父类加载器的代码。

5.3.2 双亲委派模型的工作过程是：

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都是应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

5.3.3 这样做的好处就是：

Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object,它存放在 rt.jar 中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.object 的类，并放在程序的 ClassPath 中，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱

就是保证某个范围的类一定是被某个类加载器所加载的，这就保证在程序中同　　一个类不会被不同的类加载器加载。这样做的一个主要的考量，就是从安全层　面上，杜绝通过使用和 JRE 相同的类名冒充现有 JRE 的类达到替换的攻击方式

20.Java 内存模型与线程

6.1 内存间的交互操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java 内存模型定义了以下八种操作来完成：

lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。

unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的 load 动作使用

load（载入）：作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的 write 的操作。

write（写入）：作用于主内存的变量，它把 store 操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行 read 和 load 操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行 store 和 write 操作。Java 内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。也就是 read 和 load 之间，store 和 write 之间是可以插入其他指令的，如对主内存中的变量 a、b 进行访问时，可能的顺　　序是 read a，read b，load b，　load a。

Java 内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

不允许 read 和 load、store 和 write 操作之一单独出现

不允许一个线程丢弃它的最近 assign 的操作，即变量在工作内存中改变了之后必须同步到主内存中。

不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从工作内存同步回主内存中。

一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load 或 assign）的变量。即就是对一个变量实施 use 和 store 操作之前，必须先执行过了 assign 和 load 操作。

一个变量在同一时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一条线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。lock 和 unlock 必须成对出现

如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行 load 或 assign 操作初始化变量的值

如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作；也不允许去 unlock 一个被其他线程锁定的变量。

对一个变量执行 unlock 操作之前，必须先把此变量同步到主内存中（执行 store 和 write 操作）。

6.2 重排序

在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

1.编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。

2.指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3.内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从 Java 源代码到最终实际执行的指令序列，会经过下面三种重排序：

为了保证内存的可见性，Java 编译器在生成指令序列的适当位置会插入内存屏障指令来

禁止特定类型的处理器重排序。Java 内存模型把内存屏障分为 LoadLoad、LoadStore、StoreLoad 和 StoreStore 四种：

6.3 对于 volatile 型变量的特殊规则

当一个变量定义为 volatile 之后，它将具备两种特性：

第一：保证此变量对所有线程的可见性，这里的可见性是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。普通变量的值在线程间传递需要通过主内存来完成

由于 valatile 只能保证可见性，在不符合一下两条规则的运算场景中，我们仍要通过加锁来保证原子性

1.运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。

2.变量不需要与其他的状态变量共同参与不变约束

第二：禁止指令重排序，普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中执行顺序一致，这个就是所谓的线程内表现为串行的语义

Java 内存模型中对 volatile 变量定义的特殊规则。假定 T 表示一个线程，V 和 W 分别表示两个 volatile 变量，那么在进行 read、load、use、assign、store、write 操作时需要满足如下的规则：

1.只有当线程 T 对变量 V 执行的前一个动作是 load 的时候，线程 T 才能对变量 V 执行 use 动作；并且，只有当线程 T 对变量 V 执行的后一个动作是 use 的时候，线程 T 才能对变量 V 执行 load 操作。线程 T 对变量 V 的 use 操作可以认为是与线程 T 对变量 V 的 load 和 read 操作相关联的，必须一起连续出现。这条规则要求在工作内存中，每次使用变量 V 之前都必须先从主内存刷新最新值，用于保证能看到其它线程对变量 V 所作的修改后的值。

2.只有当线程 T 对变量 V 执行的前一个动是 assign 的时候，线程 T 才能对变量 V 执行 store 操作；并且，只有当线程 T 对变量 V 执行的后一个动作是 store 操作的时候，线程 T 才能对变量 V 执行 assign 操作。线程 T 对变量 V 的 assign 操作可以认为是与线程 T 对变量 V 的 store 和 write 操作相关联的，必须一起连续出现。这一条规则要求在工作内存中，每次修改 V 后都必须立即同步回主内存中，用于保证其它线程可以看到自己对变量 V 的修改。

3.假定操作 A 是线程 T 对变量 V 实施的 use 或 assign 动作，假定操作 F 是操作 A 相关联的 load 或 store 操作，假定操作 P 是与操作 F 相应的对变量 V 的 read 或 write 操作；类型地，假定动作 B 是线程 T 对变量 W 实施的 use 或 assign 动作，假定操作 G 是操作 B 相关联的 load 或 store 操作，假定操作 Q 是与操作 G 相应的对变量 V 的 read 或 write 操作。如果 A 先于 B，那么 P 先于 Q。这条规则要求 valitile 修改的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同。

6.4 对于 long 和 double 型变量的特殊规则

Java 模型要求 lock、unlock、read、load、assign、use、store、write 这 8 个操作都具有原子性，但是对于 64 为的数据类型（long 和 double），在模型中特别定义了一条相对宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作分为两次 32 为的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性

6.5 原子性、可见性和有序性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。Java 内存模型是通过在变量修改后将新值同步会主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性，valatile 特殊规则保障新值可以立即同步到祝内存中。Synchronized 是在对一个变量执行 unlock 之前，必须把变

量同步回主内存中（执行 store、write 操作）。被 final 修饰的字段在构造器中一旦初始化完成，并且构造器没有吧 this 的引用传递出去，那在其他线程中就能看见 final 字段的值

可见性：可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性：即程序执行的顺序按照代码的先后顺序执行。

6.6　先行发生原则

这些先行发生关系无须任何同步就已经存在，如果不再此列就不能保障顺序性，虚拟机就可以对它们任意地进行重排序

1.程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确的说，应该是控制顺序而不是程序代码顺序，因为要考虑分支。循环等结构

2.管程锁定规则：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。这里必须强调的是同一个锁，而后面的是指时间上的先后顺序

3.Volatile 变量规则：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作，这里的后面同样是指时间上的先后顺序

4.线程启动规则：Thread 对象的 start()方法先行发生于此线程的每一个动作

5.线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join()方法结束、ThradisAlive()的返回值等手段检测到线程已经终止执行

6.线程中断规则：对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断时间的发生，可以通过 Thread.interrupted()方法检测到是否有中断发生

7.对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize()方法的开始

8.传递性：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论

6.7　Java 线程调度

协同式调度：线程的执行时间由线程本身控制

抢占式调度：线程的执行时间由系统来分配

6.8 状态转换

1.新建

2.运行：可能正在执行。可能正在等待 CPU 为它分配执行时间

3.无限期等待：不会被分配 CUP 执行时间，它们要等待被其他线程显式唤醒

4.限期等待：不会被分配 CUP 执行时间，它们无须等待被其他线程显式唤醒，一定时间会由系统自动唤醒

5.阻塞：阻塞状态在等待这获取到一个排他锁，这个时间将在另一个线程放弃这个锁的时候发生；等待状态就是在等待一段时间，或者唤醒动作的发生

6.结束：已终止线程的线程状态，线程已经结束执行

21.线程安全

1、不可变：不可变的对象一定是线程安全的、无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障。例如：把对象中带有状态的变量都声明为 final，这样在构造函数结束之后，它就是不可变的。

2、绝对线程安全

3、相对线程安全：相对的线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性

4、线程兼容：对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全使用

5、线程对立：是指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码

7.1 线程安全的实现方法

1.互斥同步：

同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一个（或者是一些，使用信号量的时候）线程使用。而互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。互斥是因，同步是果：互斥是方法，同步是目的

在 Java 中，最基本的互斥同步手段就是 synchronized 关键字，它经过编译之后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令，这两个字节码都需要一个 reference 类型的参数来指明要锁定和解锁的对象。如果 Java 程序中的 synchronized 明确指定了对象参数，那就是这个对象的 reference；如果没有指明，那就根据 synchronized 修饰的是实例方法还是类方法，去取对应的对象实例或 Class 对象来作为锁对象。在执行 monitorenter 指令时，首先要尝试获取对象的锁。如果这个对象没有被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加 1，对应的在执行 monitorexit 指令时会将锁计数器减 1，当计数器为 0 时，锁就被释放。如果获取对象锁失败，哪当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止

Synchronized，ReentrantLock 增加了一些高级功能

1.等待可中断：是指当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情，可中断特性对处理执行时间非常长的同步块很有帮助

2.公平锁：是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；非公平锁则不能保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。Synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但可以通过带布尔值的构造函数要求使用公平锁

3.锁绑定多个条件是指一个 ReentrantLock 对象可以同时绑定多个 Condition 对象，而在 synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含的条件，如果要和多余一个的条件关联的时候，就不得不额外地添加一个锁，而 ReentrantLock 则无须这样做，只需要多次调用 newCondition 方法即可

2.非阻塞同步

3.无同步方案

可重入代码：也叫纯代码，可以在代码执行的任何时刻中断它，转而去执行另外一段代码（包括递归调用它本身）而在控制权返回后，原来的程序不会出现任何错误。所有的可重入代码都是线程安全的，但是并非所有的线程安全的代码都是可重入的。

判断一个代码是否具备可重入性：如果一个方法，它的返回结果是可预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的

线程本地存储：如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行？如果能保障，我们就可以把共享数据的可见范

围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题

7.2 锁优化

适应性自旋、锁消除、锁粗化、轻量级锁和偏向锁

7.2.1 自旋锁与自适应自旋

自旋锁：如果物理机器上有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程稍等一下，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁

自适应自旋转：是由前一次在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自过程，以避免浪费处理器资源。

7.2.2 锁消除

锁消除是指虚拟机即时编辑器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。如果在一段代码中。推上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行

7.2.3 锁粗化

如果虚拟机检测到有一串零碎的操作都是对同一对象的加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部

7.2.4 轻量级锁

7.2.5 偏向锁

它的目的是消除无竞争情况下的同步原语，进一步提高程序的运行性能。如果轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把这个同步都消除掉，CAS 操作都不做了

如果在接下俩的执行过程中，该锁没有被其他线程获取，则持有偏向锁的线程将永远不需要在进行同步。

22.逃逸分析

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，成为方法逃逸。甚至还可能被外部线程访问到，比如赋值给类变量或可以在其他线程中访问的实例变量，称为线程逃逸

如果一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可能为这个变量进行一些高效的优化

栈上分配：如果确定一个对象不会逃逸出方法外，那让这个对象在栈上分配内存将会是一个不错的注意，对象所占用的内存空间就可以随栈帧出栈而销毁。如果能使用栈上分配，那大量的对象就随着方法的结束而销毁了，垃圾收集系统的压力将会小很多

同步消除：如果确定一个变量不会逃逸出线程，无法被其他线程访问，那这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以消除掉

标量替换：标量就是指一个数据无法在分解成更小的数据表示了，int、long 等及 refrence 类型等都不能在进一步分解，它们称为标量。

如果一个数据可以继续分解，就称为聚合量，Java 中的对象就是最典型的聚合量

如果一个对象不会被外部访问，并且这个对象可以被拆散的化，那程序正在执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替。

23. 什么是 volatile？

关键字 volatile 是 Java 虚拟机提供的最轻量级的同步机制。当一个变量被定义成 volatile 之后，具备两种特性：

保证此变量对所有线程的可见性。当一条线程修改了这个变量的值，新值对于其他线程是可以立即得知的。而普通变量做不到这一点。

禁止指令重排序优化。普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。

为什么基于 volatile 变量的运算在并发下不一定是安全的？

volatile 变量在各个线程的工作内存，不存在一致性问题（各个线程的工作内存中 volatile 变量，每次使用前都要刷新到主内存）。但是 Java 里面的运算并非原子操作，导致 volatile 变量的运算在并发下一样是不安全的。

为什么使用 volatile？

在某些情况下，volatile 同步机制的性能要优于锁（synchronized 关键字），但是由于虚拟机对锁实行的许多消除和优化，所以并不是很快。

volatile 变量读操作的性能消耗与普通变量几乎没有差别，但是写操作则可能慢一些，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

24. 并发与线程

并发与线程的关系？

并发不一定要依赖多线程，PHP 中有多进程并发。但是 Java 里面的并发是多线程的。

什么是线程？

线程是比进程更轻量级的调度执行单位。线程可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源（内存地址、文件 I/O），又可以独立调度（线程是 CPU 调度的最基本单位）。

实现线程有哪些方式？

使用内核线程实现

使用用户线程实现

使用用户线程+轻量级进程混合实现

Java 线程的实现

操作系统支持怎样的线程模型，在很大程度上就决定了 Java 虚拟机的线程是怎样映射的。

25. Java 线程调度

什么是线程调度？

线程调度是系统为线程分配处理器使用权的过程。

线程调度有哪些方法？

协同式线程调度：实现简单，没有线程同步的问题。但是线程执行时间不可控，容易系统崩溃。

抢占式线程调度：每个线程系统来分配执行时间，不会有线程导致整个进程阻塞的问题。

虽然 Java 线程调度是系统自动完成的，但是我们可以建议系统给某些线程多分配点时间——设置线程优先级。Java 语言有 10 个级别的线程优先级，优先级越高的线程，越容易被系统选择执行。

但是并不能完全依靠线程优先级。因为 Java 的线程是被映射到系统的原生线程上，所以线程调度最终还是由操作系统说了算。如 Windows 中只有 7 种优先级，所以 Java 不得不出现几个优先级相同的情况。同时优先级可能会被系统自行改变。Windows 系统中存在一个“优先级推进器”，当系统发现一个线程执行特别勤奋，可能会越过线程优先级为它分配执行时间。

26.什么是类加载器，类加载器有哪些？

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载 java 核心类库，无法被 java 程序直接引用。

2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

4. 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

27.对象“已死”的判定算法

由于程序计数器、Java 虚拟机栈、本地方法栈都是线程独享，其占用的内存也是随线程生而生、随线程结束而回收。而 Java 堆和方法区则不同，线程共享，是 GC 的所关注的部分。

在堆中几乎存在着所有对象，GC 之前需要考虑哪些对象还活着不能回收，哪些对象已经死去可以回收。

有两种算法可以判定对象是否存活：

1.）引用计数算法：给对象中添加一个引用计数器，每当一个地方应用了对象，计数器加 1；当引用失效，计数器减 1；当计数器为 0 表示该对象已死、可回收。但是它很难解决两个对象之间相互循环引用的情况。

2.）可达性分析算法：通过一系列称为“GC Roots”的对象作为起点，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（即对象到 GC Roots 不可达），则证明此对象已死、可回收。Java 中可以作为 GC Roots 的对象包括：虚拟机栈中引用的对象、本地方法栈中 Native 方法引用的对象、方法区静态属性引用的对象、方法区常量引用的对象。

在主流的商用程序语言（如我们的 Java）的主流实现中，都是通过可达性分析算法来判定对象是否存活的。