Java特种兵

# 功底

## 编译期优化

|  |
| --- |
| String a = "a" + "b" + 1;  String b = "ab1";  println(a == b); // true 编译期优化 |

|  |
| --- |
| String a = "a";  final String c = "a";  //a并不是一个常量,而是局部变量,字节码增强技术就可以修改a的实际赋值  String b = a + "b";  String d = c + "b"; //final  //1.编译器不会看方法内部做了什么,2.递归深度不可预测  //3.返回的是对常量引用的拷贝,不是final的,可以修改  String e = getA() + "b";  String compare = "ab";  println(b == compare); //fasle  println(d == compare); //true  println(e == compare); //fasle |

|  |
| --- |
| private final static String getA() {  return "a";  } |

|  |
| --- |
| //String b = a + "b" 的实际编译效果  StringBuilder temp = new StringBuilder();  String b = temp.append(a).append("b"); |

## Intern

public native String intern();

与string pool关联,加锁寻找string字符串,用equals方法判断是否是目标字符串

Jdk1.6: string pool在 Perm Gen中

Jdk1.7:string pool在堆中

|  |
| --- |
| public static void test3() {  String a = "a";  String b = a + "b"; //a是变量,new StringBuilder().append(a).append("b");  String c = "ab"; //在string pool中新建ab字符串  String d = new String(b);//新对象  println(b == c); //F  println(c == d); //F  println(c == d.intern()); //True:intern:寻找string pool中的ab,返回地址,没有则创建后返回地址  println(b.intern() == d.intern()); //True:intern:寻找string pool中的ab,返回地址,没有则创建后返回地址  } |

Feng:老年代是否可以独自清理?而新生代不清理?

## StringBuilder

|  |
| --- |
| StringBuilder stringBuilder = new StringBuilder();  for (...) {  //最坏的情况是它所占用的内存空间接近Old区域1/3时发生扩容,导致OOM  stringBuilder.append(string);  } |

stringBuilder在append扩容的时候,取max(2倍,count+string.length),所以在小字符串append大字符串时,扩容的空间刚刚足够,而大字符串append小字符串时,就会多出大量的内存空间

## 大量判定是|否操作

举例:java字节码中的类修饰符,以下都是十六进制,只取后四位显示

Public:0001

Static:0100

Final:1000

对数字取&操作,不为0时就是true

若判断 public static final, 先取或 public\_static\_final=public|static|final;判断(value& public\_static\_final)== public\_static\_final

## 数据cache

|  |  |
| --- | --- |
| 类型 | Cache范围 |
| Integer | -128 ~127 |
| Short | -128 ~127 |
| Long | -128 ~127 |
| Float double | 无 |
| Byte | 256个值 |

# 计算机工作原理

## 栈

存储局部变量中的基本数据类型,新建对象时,储存对象引用,而对象是在堆中创建

Jvm发出指令请求,OS完成具体计算,jvm自身无法做计算

## Cache line

通常以连续64位字节为单位进行cache的

如数组获取,当取二维数组a[0][0]时,cache line操作通常会将一些临近的数组元素cache到CPU缓存中,故而连续访问a[0][0],a[0][1]…时,这些连续的数值只要cache一次

## 缓存一致性协议

同一份数据cache在多个cpu中时,要求数据读写一致,多个CPU之间要遵循缓存共享的一致性协议

CPU读写数据时都会广播,其他CPU监听,并保存数据一致性

## 内存

所有程序中使用的地址都是虚拟地址(逻辑地址),在不同的进程中可以重复

物理地址:每个进程有一段内存区域,起始地址+逻辑地址=物理地址

OS预先给jvm分配-xms大小的内存空间,而不是立即分配一个-xmx大小的空间,许多空间是真正使用时才分配的(启动java时,-xmx设置比物理内存大都可以)

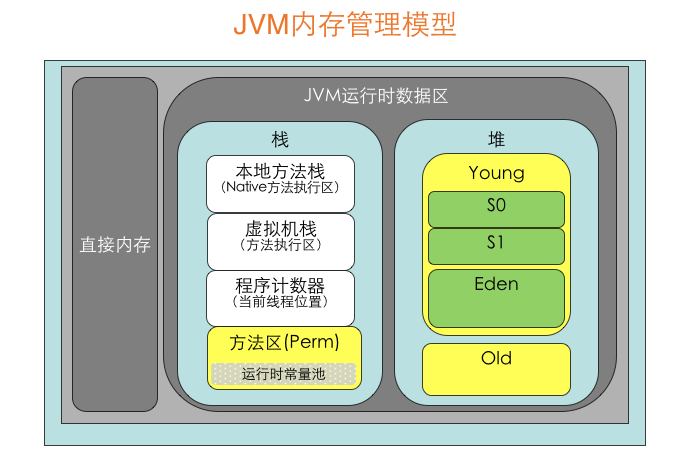
## 磁盘

每秒读取的次数IOPS越大越好

顺序读写,减少定位延迟

Java中的日志读写工具,会将日志替换为buffer的append方式,将日志写入一个缓冲区,由专门的程序实现写操作,或者只在缓冲区已满的时候写入磁盘,尽量一次写入多条数据,顺序IO,减少硬盘寻道寻址

# JVM



Old区域更大一些

静态数据存放于方法区

String对象的intern方法将string对象拷贝到常量池

## 类字节码

类常量池

方法:编译时会自动生成构造方法字节码

略过先

## Class字节码加载器

### 继承关系

ClassLoader.loadClass(“类名”)时,会先从当前ClassLoader查找该类是否已经加载,然后逐步往父类查找类,最后由父类往子类加载类,最后ClassNotFoundException

父类先加载类

BootStrapClassLoader -> ExtClassLoader -> AppClassLoader -> 自定义ClassLoader

### BootStrapClassLoader

加载java自带的核心类,如java.Lang.\*( Object, Class, Number, Thread, System, Throwable…),由jvm内核实现,不能被替换掉

### ExtClassLoader

加载 jre/lib/ext目录下的jar包,

**扩展**

jre

C:\Program Files\Java\jre : 用户运行java程序的jre环境,为jvm用

Jdk

另外安装的jdk则是开发程序时需要用到的

jdk\lib: 包括java开发环境的jar包

jdk\jre: 开发环境下运行的是 jdk 下的 jre

jdk\jre\lib: 开发环境中，运行时需要的jar包,如导入的外部jar包

### AppClassLoader

加载classPath下面的内容

### 自定义ClassLoader

加载class,jar文件,甚至其他文件,加载位置可本地,也可远程

自定义的ClassLoader可以识别到parentClassLoader加载的类,而其他的ClassLoader加载的类需要重新拼接出classpath作为参数动态编译

若未指定parentClassLoader,则parentClassLoader默认为调用者类对应的ClassLoader;初始化时可以设置parentClassLoader为null

### 启动加载

启动时,只加载jvm核心库(如BootStrapClassLoader)和main方法相关类

## class加载过程

所有类在使用前都必须被加载和初始化,初始化过程由<clinit>方法确保线程安全,若多个线程同时尝试获取该类,则必须等到static块执行完成

### 读取文件(ClassNotFoundException)

加载.class文件到方法区内部(包含所有的class和static变量,都是程序中唯一的元素)

先从父classloader加载,找不到就子加载器加载,最后抛出异常

(BootStrapClassLoader -> ExtClassLoader -> AppClassLoader -> 自定义ClassLoader->classnotfoundexception)

### 链接(NoClassDefFoundError)

解析校验字节码,不符合规范就抛出NoClassDefFoundError

为class对象分配内存

### 初始化

调用class对象构造方法,静态变量,static块赋值

**初始化顺序**

**Static块 -> 代码块 -> 构造方法**

|  |
| --- |
| class Parent {  public Parent() {  System.out.println("parent constructor init...."); //4  }  static {  System.out.println("parent static block init...."); //1  }  {  System.out.println("parent normal block call...."); //3  }  }  class Child extends Parent {  static {  System.out.println("child static block call...."); //2  }  {  System.out.println("child block call...."); //5  }  public Child() {  System.out.println("child constructor call...."); //6  }  }  public class LoadObjectStepDemo {  public static void main(String[] args) {  new Child();  }  } |

|  |
| --- |
| parent static block init....  child static block call....  parent normal block call....  parent constructor init....  child block call....  child constructor call.... |

**错误初始化实例ExceptionInInitializerError**

|  |
| --- |
| class B {  // 加载类时先调用static块  private final static B instance = new B();  public static B getInstance() {  return instance;  }  public B() {  instance.test(); // new B()引用instance实例,但又发现这个类未加载完成,instance为NULL  }  public void test() {  System.out.println("test");  }  } |

## class其他知识点

### 容器跨应用访问

在web容器中使用了不同的ClassLoader来加载不同的delopy(不同应用),但可以跨classLoader互相访问信息

### ClassLoader一个类只加载一个

同一个ClassLoader一个类只会加载一个,同一个类可能会被不同ClassLoader加载,在单例模式时应该考虑这个问题

### Full GC释放class

Jvm做fullGC时,只有当相应的ClassLoader下所有的Class都没有实例引用时,可以释放ClassLoader及其下所有class

### ClassLoader加载

ClassLoader本身就是class,在没有ClassLoader时,由jvm内核加载

### Class加载与父类

先加载父类,先初始化父类static方法

### JIT运行时优化

逐步优化,会将优化后的代码存放在codeCache中

-XX:ReservedCodeCacheSize : 修改codeCache大小,64bit server java7默认48M

-XX:+UseCodeCacheFlushing : 清理codeCache

-XX:CICompilerCount : 最大并行编译数,越大提高编译速度

### 同名类加载冲突

同步类出现在不同jar包中,可以使用instance.getClass().getResource("").getPath();获得class来源jar包

### 根引用

GC时的根引用是本地变量引用,操作数栈引用,PC寄存器,本地方法栈引用,静态引用等

即程序运行时栈中的引用+静态引用列表

### 引起Full GC

1. Old区域满 || 小于平均晋升空间大小
2. Perm区域满:class string
3. System.gc()
4. Dump内存

## 回收算法

## 常用GC参数

### 跟 Java 堆大小相关的 JVM 内存参数

-Xms 设置 Java 堆的初始化大小

-Xmx 设置最大的 Java 堆大小

-Xss 设置Java线程堆栈大小

-Xmn 设置新生代空间大小

### 关于打印垃圾收集器详情的 JVM 参数

-verbose:gc 记录 GC 运行以及运行时间，一般用来查看 GC 是否是应用的瓶颈

-XX:+PrintGCDetails 记录 GC 运行时的详细数据信息，包括新生成对象的占用内存大小以及耗费时间等

-XX:-PrintGCTimeStamps 打印垃圾收集的时间戳

### 设置 Java 垃圾收集器行为的 JVM 参数

-XX:+UseParallelGC 使用并行垃圾收集

-XX:-UseConcMarkSweepGC 使用并发标志扫描收集

-XX:-UseSerialGC 使用串行垃圾收集

### JVM调试参数，用于远程调试

-Xdebug -Xnoagent -Xrunjdwp:transport=dt\_socket,server=y,suspend=n,address=8000

### 关于类路径方面的 JVM 参数

-Xbootclasspath用来指定需要加载，但不想通过校验的类路径。JVM 会对所有的类在加载前进行校验并为每个类通过一个int数值来应用。这个是保证 JVM 稳定的必要过程，但比较耗时，如果希望跳过这个过程，就把类通过这个参数来指定。

### 用于修改 Perm Gen 大小的 JVM 参数

下面的这三个参数主要用来解决 JVM 错误： java.lang.OutOfMemoryError:Perm Gen Space.

-XX:PermSize and -XX:MaxPermSize

-XX:NewRatio=2 Ratio of new/old generation sizes.

-XX:MaxPermSize=64m Size of the Permanent Generation.

### 用来跟踪类加载和卸载的信息

-XX:+TraceClassLoading和 -XX:+TraceClassUnloading 用来打印类被加载和卸载的过程信息，这个用来诊断应用的内存泄漏问题非常有用。

### JVM switches related to logging

-XX:+PrintCompilation prints out the name of each Java method Hotspot decides to JIT compile.

### 用于调试目的的 JVM 开关参数

-XX:HeapDumpPath=./java\_pid.hprof Path to directory or file name for heap dump.

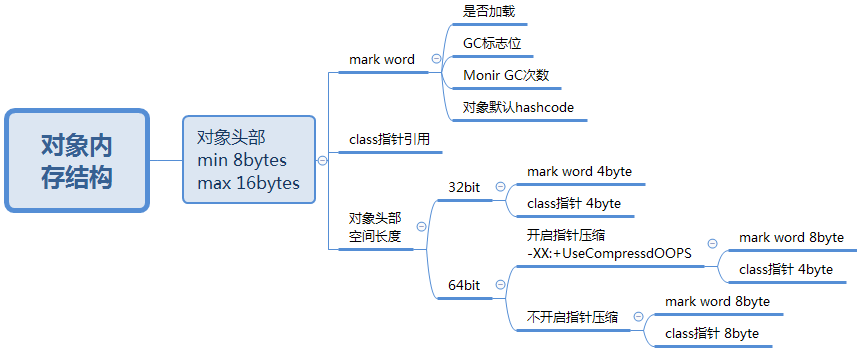
-XX:-PrintConcurrentLocks Print java.util.concurrent locks in Ctrl-Break thread dump.

-XX:-PrintCommandLineFlags Print flags that appeared on the command line.

## Java对象内存结构

Java对象将以8字节对齐在内存中,不足则补齐

静态引用所占的空间通常不计算到对象空间本身的空间上,它的引用在方法区



32bit

|  |
| --- |
| class A{  byte b1;  } |

8字节头部+1字节b1

要对齐,故16字节

* 继承关系的对象属性排布

在内部结构中,父类的属性依然要被分配到相应的子类对象中,这样才能在程序中通过父类访问它的属性

父类的属性不能和子类混用,它们必须单独排布在一个地方

|  |
| --- |
| class A{byte b;}  class B extends A{byte b;}  class C extends B{byte b;} |

|  |  |
| --- | --- |
| mark word | |
| class指针引用 | |
| byte A.b | 3字节属性对齐 |
| byte B.b | 3字节属性对齐 |
| byte C.b | 3字节属性对齐 |
| 整个对象对齐 | |

Table 对象结构图

* 数组占用空间实例(32bit)

|  |
| --- |
| int size = 100 \* 1024 \* 1024;  //1  int[] values = new int[size];  for (int i = 0; i < size; i++) {  values[i] = i;  }  //2  Integer[] valueIntegers = new Integer[size];  for (int i = 0; i < size; i++) {  valueIntegers[i] = i; // 自动装箱了 new Integer(i)  } |

* + 对1:

int[]数组占用空间

8字节头部+4字节描述数组长度+4X100x1024X1024=400MB(int 值4字节)+padding4字节 ≈ 400MB

* + 对2:

每个Integer对象占8字节头部+4字节int值,又需要对齐,故16字节

实例对象一共占用空间16 X 100x1024X1024=1600MB

Integer[] 数组占用空间

8字节头部+4字节描述数组长度+引用空间4X100x1024X1024=400MB(每个引用4字节)+padding4字节 ≈ 400MB

总空间1600+400=2000MB

* Int[2][100] PK int[100][2]

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  Int[2][100] | int[100][2] |
| 第一维数组 | 对象头部 | 8 | 8 |
| 数组长度描述符 | 4 | 4 |
| 引用宽度 | 2X4=8 | 100X4=400 |
| Padding | 4 | 4 |
| 合计 | 24 | 416 |
| 第二维 | 对象头部 | 8 | 8 |
| 数组长度描述符 | 4 | 4 |
| 引用宽度 | 100X4=400 | 2X4=8 |
| Padding | 4 | 4 |
| 合计 | 416 | 24 |
| 总计 |  | 24+2X416=856 | 416+100X24=2816 |

## 常见OOM

### java.lang.OutOfMemoryError: Java heap space

|  |
| --- |
| public static void main(String[] args) {  List<String> list = new ArrayList<String>();  while (true) {  list.add("内存溢出了");  }  } |

* 解决方法

1. 对可能存活较久的大对象:object = null
2. 代码提速:代码运行速度提升,缩短对象生命周期
3. 修改堆大小

### java.lang.OutOfMemoryError: PermGen space

1. jdk1.6 PermGen空间

|  |
| --- |
| int i = 0;  while (true) {  ("在JDK 1.6下运行，在JDK 1.7中运行的结果将完全不同 "  + "string常量在jdk1.7以上就不再存放在PermGen中" + i++).intern();  } |

1. 动态加载class,如使用字节码增强技术,CGlib一直创建加载class

若需要动态加载类,动态编译java代码,最好是有单独的classLoader,当class被替换时,原来的class可以被当做垃圾释放掉

释放class的条件是classLoader下的class都没有活着的对象

### DirectBuffer OOM

java.lang.OutOfMemoryError: Direct buffer memory

DirectBuffer区域不是java的heap,而是C heap的一部分,通常FULL GC时回收

|  |
| --- |
| // -XX:MaxDirectMemorySize=26m  public static void main(String[] args) {  ByteBuffer.allocateDirect(27 \* 1024 \* 1024);  } |

### StackOverflowError

|  |
| --- |
| public void testStackOver() {  testStackOver();  } |

注意递归层数,将递归调用次数作为参数,到达一定次数后结束递归

子类和父类相互间调用方法

# Java通信

## 字符编码转换

若字符编码和解码的方式不一致,很可能损坏源字符,造成无法正确读取

比如变长的UTF-8编码可以由3个字节组成一个汉字,而GBK由2个字节组成汉字,GBK按2个单位 长度读取时,不在其编码范围内的则用?或其他字符替代,这就修改了原来的字符串了

## 流继承嵌套

* 继承

有部分流的具体实现中会继承FilterInputStream, FilterInputStream提供InputStream的默认实现.则流只覆写特定的方法即可

* 嵌套

可以对同一个stream使用多个stream实现类来嵌套,注意当其中一个实现stream close时,底层基础stream关闭了,而其他stream没有调用其close方法,可能会出错

比如多个BufferedInputStream嵌套一个流,关闭其中一个BufferedInputStream后,底层input关闭,但剩余的其他BufferedInputStream没有执行close方法,其buffer的数据不会刷新到磁盘上,造成数据问题

## I/O与内存

### 文件读入与读出

不关闭流会有什么问题?

### 阻塞与非阻塞 同步与异步

同步和异步关注的是消息通信机制,重点在于被调用的结果

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态,重点在于程序自身

* 同步

是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了, 由\*调用者\*主动等待这个\*调用\*的结果

* 异步

\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果。在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用。

* 阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

* 非阻塞

非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程,当前线程继续执行

# 数据库

## 数据库基本原理

### 数据块/页

块,数据存储单位:4KB,8KB,16KB, 块也是虚拟出来的,为数据库规范,可由数据库识别

每个块可存储多条数据,块都有标识符(不唯一,只有和更高层次的标识符组合才唯一)

块内部每条数据通常都有自己的行号

数据性能瓶颈在于磁盘IOPS上,读取1KB和1MB的连续数据,耗时相差无几

慢原因:每个磁盘IO请求都是串行化,每个IO请求需要磁道寻址,故而慢

优化:可以将多个块的数据连续放在一起组成一组数据

当连续块中有部分数据常用,部分不常用时,不常用的数据会被写回磁盘,则程序再次加载这些块时,只是加载未加载的部分,这就造成不连续加载,成为随机读

* 数据字典

通常以表的方式储存,记录数据库信息,包括每个块在文件系统中的偏移量

数据量少,更新少,通常加载到内存中提供访问

数据库通常会以LRU的算法来管理块,让某些访问多的块留在内存,或让访问少的块从内存释放掉

### 分组extends

将多个块组成一个分组,理论上是连续储存的

### segments

多个extends组成一个segments,一个表对象可以划分到多个segments,以实现分区表独立管理

### 修改表

添加一个表格字段

某些数据库会事先在写入每个块时预留少部分空间,一遍添加字段时使用

当预留空间的字段不够用时,会将一行数据的某个部分写入到另一个块中,一般不在连续块上则需要多次IO读取数据了

某些开源数据库是没有数据字典概念,大部分描述信息存放在以表为单位的文件头部中,当表结构修改时,通常会创建一个新表,并将原表中的数据全部拷贝到新表中,为保持数据一致性,拷贝时通常会锁表

对于大数据储存,通常采用分库分表策略

### 删除表

伪删除:删除某些数据库中的表,为了防止误删除,会在默认的处理删除动作时,只是修改一个名字并将修改后的表名与原来的表名做一个简单映射,同时也改变其元数据的状态

### SQL执行

解析获得关联结构信息,执行计划等

* 软解析

缓存SQL解析过程,同样SQL,只有参数不同,可以直接拿解析好的结果直接运行,提高效率

* 硬解析

拼接处参数,而非预编译,SQL不复用,低效

* SQL储存空间

SQL存储空间有限,使用类LRU算法管理内存区域

* 执行计划

单表是否走索引,多表时是jion还是union,排序方法选择,分钟策略确定等

|  |
| --- |
| select \* from table small,table big where small.id =big.id and small.type=’1’; |

small.id =big.id:用小表做驱动表,更快

small.type=’1’:外层先执行,过滤驱动表

### 加锁

读写表时,通常会在数据字典上加锁,保证表结构不会被修改,一般类似与读写锁,不过写时确也可以读,可能会造成读脏数据的问题,解决方法是增加时间戳或版本号做标识

对于某些数据库,锁住的行可能并不仅仅是最终where条件筛选出的行,可能是where条件中有索引字段筛选出的所有行

|  |
| --- |
| update table set … where a=xx and b=xxxx |

若a有索引,而b没有,则可能会锁住所有a条件筛选出的数据,锁的范围更大

有些数据库为了加快加锁的速度,会以”块”为单位加锁,块头部有内部的行信息,每一行都有1bit来标识是否已经加锁

对应大量的数据修改,可能因锁而导致串行速度问题,可以利用cas机制代替锁

### 提交

Commit操作时,会将日志写入到磁盘

数据库会在内存分配一个日志缓冲区,隔一定时间/日志达到一定容量就存盘

日志的存在可以为数据库提供回滚功能

另外,也有可能是数据库记录每一个块的相应版本到某个空间,只要空间够用,就不会将相应版本的数据清空,回滚时直接查询对应版本好/时间点的数据

## 索引原理

索引方法:B+树,也有hash

索引是与数据分开储存的

索引的目的是快速定位数据,索引通过某种标识符与原表关联起来(也加回表),标识符可以是主键,或者建立索引的字段值,也可以是数据物理位置(oracle用rowid做标识符:表空间编号+文件编号+对象编号+块号+块内行号)

除非查询的信息全部在索引上,否则索引至少会走两次操作(索引+回表),所以小规模数据索引表现不好

索引更新时,索引先删除再插入,其中删除是伪删除,索引空间不会变小

重新编译索引才会真正删除索引

### 索引管理

树状管理索引,数据量越大,层数越多,索引间有序,底层所有的叶子块通过双向链表互相关联

Figure 三层索引管理

SQL走索引时,会从树根开始查找,索引块通常也cache在缓存中(若索引块数据量过大,只缓存高层索引块),可能就不需要I/O,另外,由于索引有序,可以通过二分法快速查找,若查完索引后仍需要查询费索引字段,此时回表,否则直接走索引内部数据

SQL某些统计操作可以通过索引完成(比如count),只统计叶子块,就可以得到全表信息

Min,max操作,有where条件,则正常遍历索引,没有条件,直接查第一个和最后一个块

like前缀索引可以通过范围查找,而后缀索引就可能是全表扫描

* 字段有索引并不一定要走索引

in检索时,通常解析为OR方式完成,检索在多个离散块中进行,每个条件都需要单独查找,in条件多就走全表扫描(将in改为exists?)

再如,某些状态字段作为条件,可能也不会走索引

* 位图索引bitMap

但字段只有几种值时,bitmap可以实现高效的统计.

位图索引结构类似于B+树,储存时以字段值类型分开,在每种值的空间中单独储存每个数据行是否有这个值,1有,0没有

如字段类型为A,索引中大致可以储存为101010100,表示第1 3 5 7行有A值

缺点:锁粒度太大

若修改状态1为2,则会锁住所有值为1和2的行,commit或rollback才会释放

## 数据库主从原理

主库更新数据,在同再到从库

* 同步方式:逻辑模式

主库执行SQL,到从库上执行相同SQL

主库可以优化SQL,如生成基于主键更新的SQL,从库执行会简单一些

从库基本只有一条线程写,其余读,压力小,更高效

缺点:SQL较多,从库写速度更不上主库,导致不一致

* 物理模式

基于修改的数据块做复制

提取比从库版本号更高的数据块复制到从库,不用SQL解析,锁,调度等问题,更加高效

## 执行计划

小表做驱动表,大表走索引

* Oracle

1.explain plan for sql,select \* from table(DBMS\_XPLAN.DISPLAY)

2.SET AUTOTRACE ON EXPLAIN , SET AUTOTRACE ON, SET AUTOTRACE TRACEONLY STAT

Eg:

explain plan for select \* from tableAA a where a.tt=:vv1;

:开头代表占位符

INDEX UNIQUE SACN:走唯一索引

INDEX RANCE SCAN:普通索引是走范围索引的,因为可以重复

如果Oracle执行计划不对或不理想,可以通过Hint方式告诉Oracle应该如何走索引

* Mysql

explan<sql>

* 执行计划

在JOIN表之前,需要被JOIN的表能先过滤掉大部分数据

小表驱动大表,嵌套循环时,有序会快一些

多个结果集JOIN,可以使用Hash Join的方式,当表太大,Hash Join key太多,内存会放不下,用普通嵌套方式做

* 函数转换

函数转换通常不属于执行计划的范畴

|  |
| --- |
| select fun1(a),fun(b)… from table |

执行路径是看不到函数的处理

## 模型结构优化

表示树状结构,如省,市,区县,镇,乡,村,门牌,多级结构,可以使用常用的递归方式,只保留父级id

也可以按序保留下所有的父级id,避免递归,索引也方便

# 源码基础

## 源码调用路径

|  |
| --- |
| Thread.currentThread().getStackTrace(); |

|  |
| --- |
| new Exception().printStackTrace(); |

## 反射

### Boolean类型的属性注入

Boolean类型属性名称还是不要加is前缀

|  |
| --- |
| class Node {  private boolean good;  public boolean isGood() {  return good;  }  public void setGood(boolean good) {  this.good = good;  }  } |

当使用PropertyDescriptor去获取属性读写方法时,boolean类型默认都会加上is前缀

若属性名为isGood,则默认调用isIsGood方法,这时类中isGood方法就不管用了

### 反射与范型擦除

|  |
| --- |
| List<Integer> list = new ArrayList<Integer>();  Method method = ArrayList.class.getDeclaredMethod("add",Object.class);  method.invoke(list, 7);  method.invoke(list, "dfsd");  method.invoke(list, new A("aa"));  System.out.println(list); |

1.泛型实参只会在类、字段及方法参数内保存其签名(即可以获得实际的参数类型)，无法通过反射动态获取泛型实例的具体实参,比如入参为范型,方法内部就会擦除为object。

2.需要获取泛型实参的情况下，方法有三；①通过传递实参类型 ②明确定义泛型实参类型，通过反射获取签名 ③通过匿名类捕获相关的泛型实参

### AOP

面向切面,动态代理和字节码增强技术

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP

2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

* + 动态代理

动态代理再生成动态字节码时,并不是通过实现类创建子类的方式,而是通过类所拥有的接口列表来完成,即构造出的类和实现类是没有关系的

在构造动态代理时,类必须基于接口实现动态代理,在最终获得动态代理的的实例引用时,也只能用接口来获取

本质上还是将实现类作为引用传入到Handler中,还是会调用实现类的方法,只是包装了一层

实现接口的方法,方法内再通过引用调用实现类的方法

如果在内部方法再调用内部方法,那么在第二层内部方法中是无法完成AOP切入的,那已经是直接调用了

* + 字节码增强

如果在内部方法再调用内部方法,可以一直完成AOP切入,这是因为字节码增强是构建子类或者直接修改类的字节码,若是子类,由于子类覆写父类方法,就一直调用被修改的子类方法,可以一直AOP;若是直接修改字节码,那就是方法修改了

### Annotation

注解根本上还是依靠反射实现的

# JDBC

## JDBC注册

驱动只需加载一次,不需要反复加载,也不需要自己new,注册的driver以列表的形式保存,驱动加载也与classloader有关,所有也可以在不同的classloader加载同一驱动的不同版本

当程序中存在多个驱动时,DriverManager通过遍历的方式查找其classloader中所有的driver,与jdbcurl匹配,匹配上就返回

## Spring