八股文背诵合集

The Sea and Sheng

2021 年 8 月 17 日

目录

1 Spring

1.0.1 Spring 框架能带来哪些好处

1. Dependency Injection(DI) 依赖注入是的构造器和JavaBean properties文件中的依赖关系一目了然.

2. IoC容器更加趋向于轻量级.

1.0.2 介绍一下Spring, 读过源码介绍一下大致流程

1. Spring是一个快速开发框架, Spring帮助程序员来管理对象

2. Spring的源码实现的是非常优秀的, 设计模式的应用, 并发安全的实现, 面向接口的设计等

3. 在创建Spring容器, 也就是启动Spring时:

a. 首先会进行扫描, 骚婊得到所有的BeanDefinition对象, 并存在一个Map中

b. 然后筛选出非懒加载的单例BeanDefinition进行创建Bean, 对于多例Bean不需要再启动过程中去进行创建, 对于多例Bean会在每次获取Bean时利用beanDefinition去创建

c. 利用beanDefinition创建Bean就是Bean的创建生命周期, 这期间包括了合并BeanDefinition, 推断构造方法, 实例化, 属性填充, 初始化前, 初始化, 初始化后等步骤, 其中AOP就是发生在初始化后这一步骤中

4. 单例Bean创建完了之后, Spring会发布一个容器启动事件.

5. Spring启动结束

1.0.3 Autowired

使用byType和byName去寻找需要进行注入的对象

1.0.4 什么是控制反转(IOC)

1. 控制反转简单来说, 以前程序开发的时候, 是由程序员通过new来生成对象. 在使用控制反转的情况下, 对象的实例化由Spring框架中的IoC容器来控制对象的创建;

2. 由容器来管理这些对象的生命周期.

3. Spring中的org.springframework.beans包和org.springframework.conext包构成了Spring框架Ioc的基础. 主要使用文件 applicationContext.xml 来进行配置.

1.0.5 什么是依赖注入?

1. Spring 通过反射来实现依赖注入
2. 当我们需要某个功能比如Connection, 至于Connection怎么构造, 何时构造我们不需要知道. 在系统运行时, Spring会在适当的时候制造一个Connection, 我们需要一个Connection, 这个Connection是由Sping注入到A中.

1.0.6 Spring 对对象进行创建流程

class对象反射 —¿ 实例化 —¿ 生成对象 —¿ 属性填充(依赖注入) —¿ 初始化(afterPropertiesSet) —¿ AOP —¿ 代理对象(cglib) —¿ bean

1.0.7 什么是AOP

允许横切业务, 由切面构成, 切面又切入点和通知构成,@Aspect注解的类就是切面.

(1) 目标对象(Target)

要被增强的对象.

(2) 连接点, 哪个目标方法, 相对点, 目标方法的前还是后.

1.0.8 bean的生命周期

什么是bean?

从上面可知，我们可以给Bean下一个定义：Bean就是由IOC实例化、组装、管理的一个对象。 如上图所示，Bean 的生命周期还是比较复杂的，下面来对上图每一个步骤做文字描述:

(1) Spring启动，查找并加载需要被Spring管理的bean，进行Bean的实例化

(2) Bean实例化后对将Bean的引入和值注入到Bean的属性中

(3) 如果Bean实现了BeanNameAware接口的话，Spring将Bean的Id传递给setBeanName()方法

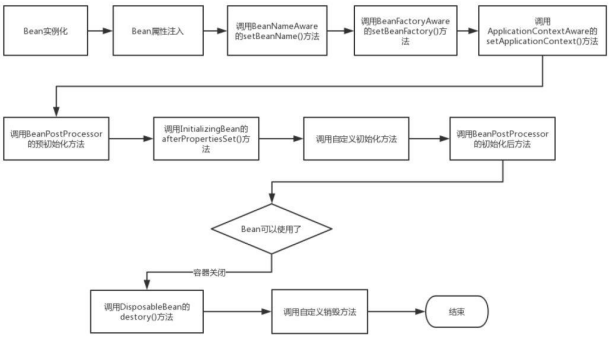

图 1:

(4) 如果Bean实现了BeanFactoryAware接口的话，Spring将调用setBeanFactory()方法，将BeanFactory容器实例传入

(5) 如果Bean实现了ApplicationContextAware接口的话，Spring将调用Bean的setApplicationContext()方法，将bean所在应用上下文引用传入进来。

(6) 如果Bean实现了BeanPostProcessor接口，Spring就将调用他们的postProcessBeforeInitialization(法。

(7) 如果Bean实现了InitializingBean接口，Spring将调用他们的afterPropertiesSet()方法。类似的，如果bean使用init-method声明了初始化方法，该方法也会被调用

(8) 如果Bean 实现了BeanPostProcessor接口，Spring就将调用他们的postProcessAfterInitialization()法。

(9) 此时，Bean已经准备就绪，可以被应用程序使用了。他们将一直驻留在应用上下文中，直到应用上下文被销毁。

(10) 如果bean实现了DisposableBean接口，Spring将调用它的destory()接口方法，同样，如果bean使用了destory-method 声明销毁方法，该方法也会被调用。

1.0.9 简单阐述SpringMVC的流程

SpringMVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架, 通过把Model, View, Controller分离, 将web层进行职责解耦, 把复杂的web应用分成逻辑清晰的几部分, 简化开发.

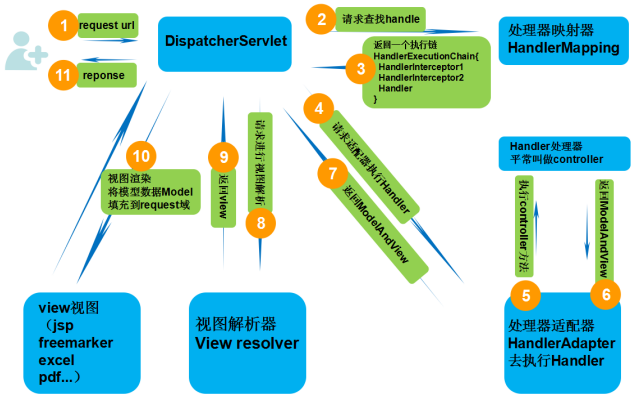

图 2:

(1) 用户发送请求到前端控制器DispatcherServlet;

(2) DispatcherServlet 收到请求后, 调用HandlerMapping处理器映射器, 请求获取Handle

(3) 处理器映射器更具请求url找到具体的处理器, 生成处理器对象以及处理器拦截器(如果有则生成)一并返回给DispatcherServlet;

(4) DispatcherServlet 调用HandlerAdapter处理器适配器;

(5) HandlerAdapter 经过适配调用具体处理器(Handler, 也叫后端控制器);

(6) Handler 执行完成返回ModelAndView;

(7) HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet;

(8) DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析;

(9) ViewResolver解析后返回具体View

(10) DispatcherServlet对View进行渲染视图(即将模型数据填充到视图中)

(11) DispatcherServlet响应用户.

简单来说, 我们需要开发的就是 == 开发处理器（Handler，即我们的Controller, 对于视图jsp我们前后端分离之后也不用写了.

1.0.10 第三个三级标题

2 java基础

2.0.1 如何查看线程死锁

0. 使用ps H -eo pid,tid,%cpu—grep 2783 可以查看java进程中的线程
1. 使用jstack命令来查看
2. 对于mysql 使用select * from INFOMATION_SCHEMA.INNODB_LOCKS
查看正在锁的事物
3. 查看等待锁的事物 SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS

2.0.2 线程之间如何进行通讯的

1. 使用共享内存或基于网络来进行通信
2. 如果是通过共享内存来进行通信

2.0.3 快速失败(fail-fast) 和安全失败(fail-safe) 的区别是什么?

1. java.util包下面的所有的集合类都是快速失败的, 而java.util.concurrent包下面的所有类都是安全失败的. 快速失败的迭代器会抛出ConcurrentModificationException异常, 而安全失败的迭代去永远不会抛出这样的异常.

2.0.4 异常

(1) Throwable(可抛出) 超类, 有两个子类Error 和 exception 错误和异常

2.0.5 synchronized的底层实现细节

1. synchronized作用

原子性: synchronized保证语句内操作是原子的

可见性: synchronized保证可见性(通过在执行unlock之前, 必须先报此变量同步回主内存实现)

有序性: synchronized保证有序性(通过”一个变量在同一时刻只允许一条线程对其进行lock操作”)

可见性补充: 其实真正解决这个问题的是JMM关于Synchronized的两条规定：

1、线程解锁前，必须把共享变量的最新值刷新到主内存中;

2、线程加锁时，讲清空工作内存中共享变量的值，从而使用共享变量是需要从主内存中重新读取最新的值（加锁与解锁需要统一把锁）

2.0.6 线程池参数

ThreadPoolExecutor的创建参数:

(1) corePoolSize, 核心运行的线程个数, 若线程池已创建的线程数小于corePoolSize, 即使此时存在空闲线程, 也会通过创建一个新线程来执行该任务.

(2) maximumPoolSize: 最大线程个数, 当大于这个值就会将准备新加入的异步任务有一个丢弃处理机制来处理, 大于corePoolSize且小于maxmumPoolSize存入等待队列,

(3) workQueue: 任务等待队列, 当达到corePoolSize的时候就向该等待队列放入线程信息.

(4) keepAliveTime: 默认0, 当线程没有任务处理后空闲线程保持多长时间, 不推荐使用, 一般会中止超过corePoolSize数量的线程资源, 空闲线程时间超过keepAliveTime, 线程将会被回收

(5) threadFacory: 构造Thread方法, 使用默认的default实现.

(6) defaultHandler: 当maximumPoolSize达到后丢弃处理的方法实现, java默认是丢出异常.

2.0.7 sychronized和ReentrantLock的区别

1. sychronized是一个关键字, ReentrantLock是一个类
2. synchronized会自动的加锁和释放锁, ReentrantLock是一个类
3. sychronized的底层是jvm层面的锁, ReentrantLock是API层面的锁
4. sychronized是非公平锁, ReentrantLock可以选择公平锁或非公平锁
5. sychronized锁的是对象, 锁信息保存在对象头中, ReentrantLock锁的是线程
6. sychronized底层有一个锁升级的过程

2.0.8 线程池中使用的BlockQueue

(1) 直接提交队列: 简单来说使用SynchronousQueue队列，提交的任务不会被保存，总是会马上提交执行。如果用于执行任务的线程数量小于maximumPoolSize，则尝试创建新的进程，如果达到maximumPoolSize设置的最大值，则根据你设置的handler执行拒绝策略。因此这种方式你提交的任务不会被缓存起来，而是会被马上执行，在这种情况下，你需要对你程序的并发量有个准确的评估，才能设置合适的maximumPoolSize数量，否则很容易就会执行拒绝策略；

(2) 有界的任务队列可以使用ArrayBlockingQueue实现，若有新的任务需要执行时，线程池会创建新的线程，直到创建的线程数量达到corePoolSize时，则会将新的任务加入到等待队列中。若等待队列已满，即超过ArrayBlockingQueue初始化的容量，则继续创建线程，直到线程数量达到maximumPoolSize设置的最大线程数量，若大于maximumPoolSize，则执行拒绝策略。在这种情况下，线程数量的上限与有界任务队列的状态有直接关系，如果有界队列初始容量较大或者没有达到超负荷的状态，线程数将一直维持在corePoolSize以下，反之当任务队列已满时，则会以maximumPoolSize为最大线程数上限。

(3) 使用无界任务队列，LinkedBlockingQueue 实现线程池的任务队列可以无限制的添加新的任务，而线程池创建的最大线程数量就是你corePoolSize设置的数量，也就是说在这种情况下maximumPoolSize这个参数是无效的，哪怕你的任务队列中缓存了很多未执行的任务，当线程池的线程数达到corePoolSize后，就不会再增加了；若后续有新的任务加入，则直接进入队列等待，当使用这种任务队列模式时，一定要注意你任务提交与处理之间的协调与控制，不然会出现队列中的任务由于无法及时处理导致一直增长，直到最后资源耗尽的问题。

(4) 优先任务队列：优先任务队列通过PriorityBlockingQueue实现，

2.0.9 Executors 工厂类实现线程池

通过创建不同的ThreadPoolExecutor参数.

(1) FixedThreadPool 定长, corePoolSize == maxmumPoolSize

(2) SingleThreadExecutor 单一线程无界队列

以上两种可能会堆积大量的请求, 从而引起OOM异常

(3) CachedThreadPool 采用maxmumPoolSize为无限大, 容易创建大量线程, 从而耗尽系统资源.

2.0.10 线程池的拒绝策略

(1) abortPolicy 默认: 直接抛出异常

(2) CallerRunsPolicy: 直接调用主线程来执行任务.

(3) DiscardPolicy: 不能执行的任务被删除, 和abortPolicy一样, 但是不抛出异常.

(4) DiscardOldestPolicy: 位于工作队列头部的任务将被删除, 然后重新执行程序.

2.0.11 8种基本数据类型

表 1: 实现

类型	大小(注释/包装类)
byte	8(Byte)
short	16(Short)
int	32(Integer)
long	64(Long)
float	32(Float)
double	64(Double)
char	16(Character)
boolean	8(Boolean)

2.0.12 Comparable & Comparator 区别

Comparable 是接口能力赋予

```
1 public interface Comparable<T> {
2         public int compareTo(T o);
3 }
```

Comparator 是外部比较器, 也是接口, 类似于 C++sort中自定义的cmp函数

```
1 Collections.sort(list, new Comparator<Person2>() {
2         public int compare(Person o1, Person o2) {
```

```
3                 return o1.getAge() - o2.getAge();
4             }
5 })
```

2.0.13 java采用值传递还是引用传递?

采用值传递, 但是因为采用浅拷贝, 所以会修改传递的对象的相关属性.

2.0.14 java深拷贝和浅拷贝

实现了Coneable接口实现深拷贝.

2.0.15 java"==" 和 equals 的区别

1. "==" : 如果是基本数据类型, 则直接对值进行比较, 如果是引用数据类型, 则是对他们的地址进行比较;

2. equals方法继承Object类, 在具体实现时可以覆盖父类中的实现. 看一下Object中equals的源码发现, 它的实现也是对**对象的地址**进行比较, 可以覆盖实现这个方法, 如果两个对象的类型一致, 并且内容一致, 则返回true.

在实际开始中总结:

(1) 类未复写equals, 则使用equals方法比较两个对象时, 相当于==比较, 及两个地址是否相等. 地址相等, 返回true, 地址不相等, 返回false.

(2) 类复写equals方法, 走复写之后的判断方式. 通常, 我们会将equals复写成: 当两个对象内容相同时, 则equals返回true, 内容不同时, 返回false.

对于set, hashMap, hashset等, 还要重写hashCode值, 比如set判断两个元素是否相等的时候, 会判断hashcode和equals都相等, 则认为相等, 不会添加新元素.

2.0.16 String和StringBuilder, StringBuffer的区别

String是不可变字符串对象(final的char数组), StringBuilder和StringBuffer(线程安全)是可变字符串对象.

为什么String是final修饰的?

1. 为了实现字符串池, 因为只有当字符串是不可变的, 字符串池才有可能实现.

2.0.17 Java反射机制

简单来说就是在, 运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意方法和属性; 并且能改变它的属性. 这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制.

优点: 代码灵活度提高

缺点: 性能瓶颈, 性能较慢.

2.0.18 简述面向对象三大特征, 继承, 封装, 多态

1. 封装

简单来说, 就是使用private方法将没有必要暴露的方法和属性进行隐藏.

2. 继承

继承是从已有的类中派生出行的类, 减少代码冗余.

3. 多态

父类引用指向不同子类对象.

2.0.19 多态

一般使用instance of 来判断对象的子类关系. 增加向下转型的健壮度.

2.0.20 内部类

(1) 静态内部类访问外部变量必须是静态的.

(2)

2.0.21 红黑树

一般考察红黑树: 只考察概念.

1. 节点是红色或黑色
2. 根节点是黑色
3. 所有叶子都是黑色(叶子是NIL节点).
4. 每个红色节点必须有两个黑色节点

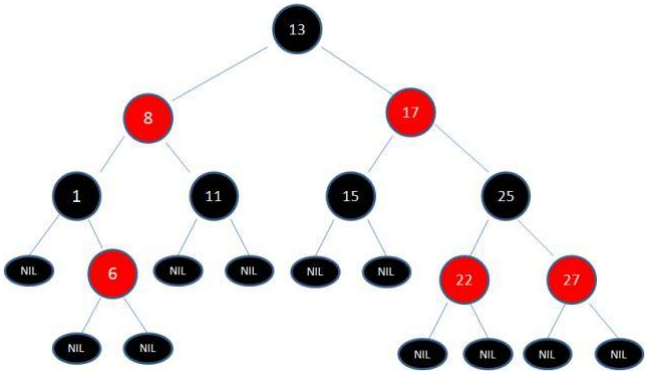

图 3:

5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点.

2.0.22 hashmap的数据结构

1. jdk1.7 由数组 + 链表来构成
2. jdk1.8 由数组 + 链表 + 红黑树来构成
3. jdk1.8的时候, 当元素不超过64个的时候, 不会出现链表转红黑树, 当元素超过64个的时候, 会出现链表转红黑树.
4. jdk1.8 当链表长度达到8个的时候, 链表会转为红黑树, 当红黑树元素长度退回到6个的时候会出现红黑树转为链表.
5. jdk1.7 采用头插法, jdk1.8采用尾插法.

2.0.23 hashmap的put方法

1. 根据Key通过哈希算法与与运算得到数组下标

2. 如果数组下标位置元素为空, 则将key和value封装为Entry对象并放入该位置

3. 如果数组下标元素不为空

1.7, 则先判断是否需要扩容, 如果要扩容就进行扩容, 如果不用扩容就生成Entry对象, 并使用头茶法添加到当前位置的链表中

1.8 先判断当前位置上Node的类型, 看是红黑树Node还是链表Node

a. 如果是红黑树Node, 则将key和value 封装为一个红黑树节点并添加到红黑树中

b. 如果是链表节点, 使用尾插法插入到链表的最后位置去, 插入完后会判断当街链表的个数看是否需要转为红黑树(超过8个)元素.

c. 判断是否需要扩容(0.75*16默认值), 需要扩容就扩容, 不需要就结束PUT方法

2.0.24 介绍一下ThreadLocal

1. ThreadLocal 是java中所提供的的线程本地存储机制, 可以利用该机制将数据缓存在某个线程内部, 该线程可以在任意时刻, 任意方法中获取缓存的数据

2. ThreadLocal 底层是通过ThreadLocalMap来实现的, 每个Thread对象中都存在一个ThreadLocalMap, Map的key为ThreadLoacl对象, Map的value为需要缓存的值

3. 如果在线程池中使用ThreadLocal会造成内存泄露, 因为当ThreadLocal对象使用完后, 应该要报设置的key, value 也就是Entry对象进行回收, 但线程池中的线程不会回收, 而线程对象是通过强引用指向ThreadLocalMap, ThreadLocalMap也是通过强引用指向Entry对象, 线程不被回收, Entry对象也就不会被回收, 从而出现内存泄露, 解决方法是, 在使用了ThreadLocal对象之后, 手动调用ThreadLocal的remove方法, 手动清除Entry对象.

2.0.25 heap和stack有什么区别

1. java的内存分为两类, 一类是堆内存, 一类是栈内存
2. 栈内存是指程序进入一个方法时, 会为这个方法单独分配一块私属存储空间, 用于存储这个方法内部的局部变量. 当这个方法结束时, 分配给这个方法的栈会释放, 这个栈中的变量也随之释放.
3. 使用new创建的对象存放在堆里, 不会随方法的结束二小时. 方法中的局部变量使用final修饰后, 放在堆中, 而不是栈中.

2.0.26 Array 和 ArrayList 的区别

1. Array 大小固定, ArrayList 大小是动态变化的.

2.0.27 Java各种锁：悲观锁，泪管所，自旋锁，偏向锁，轻量/重量锁，读写锁，可重入锁

悲观锁和乐观锁指的是并发情况下的两种不同策略，是一种宏观的描述.

1. 悲观锁和乐观锁指的是并发情况下的两种不同策略，是一种宏观的描述.

2.0.28 Collection 和 Collections 的区别

1. Collection 是集合类的上级接口，继承他的接口主要是set和list
2. Collections 类数针对集合类的一个帮助类. 它提供了一系列的静态方法对各种集合的搜索，排序，线程安全化等操作.

2.0.29 接口与抽象类区别

1. 类可以实现多个接口但只能继承一个抽象类
2. 接口中变量被隐性制定为public static final，方法被指定为 public abstract
3. 接口里面所有的方法都是Public的，抽象类允许Private，Protected方法
4. JDK接口可以实现默认方法和静态方法，前面加defalut，static关键字.
5. 设计层面：抽象类是对事物的抽象，接口是对行为的抽象.

```
1 public interface InterfaceJDK8 {
2
3         /*接口的普通抽象方法*/
4         public void common(String str);
5
6         /*jdk1.8 默认方法:允许在已有的接口中添加新方法，而同
              时又保持了与旧版本代码的兼容性，默认方法与抽象方法不
              同之处在于抽象方法必须要求实现，但是默认方法则没有要
              求实现，相反，接口提供了一个默认实现，这样所有的接口
              实现者将会默认继承他（如果有必要的话，可以覆盖这个默
              认实现）。接口的默认方法：得到接口的实现类对象，直接
              用对象的引用方法名。默认方法可以被实现类覆盖。
```

```
7
8
9
10
11          .
12          */
13          default public void defaultMethod(String str){
14                  System.out.println("InterfaceJDK8:" +
                        str);
15          }
16
17          /*jdk1.8 静态方法：允许在已有的接口中添加静态方法，接
                口的静态方法属于接口本身，不被继承，也需要提供方法的
                现。
18
19          */
20          public static void staticMethod(String str){
21                  System.out.println("InterfaceJDK8:" +
                        str);
22          }
23
24  }
```

2.0.30 ArrayList和LinkedList内部实现大致是怎样的？他们之间的区别和优缺点

1. ArrayList: 内部使用数组的形式实现了存储, 利用数组的小表进行元素的访问, 因此对元素的随机访问速度非常快. 初始化大小为10, 插入新元素的时候, 会判断是否需要扩容, 扩容的步长是0.5倍原容量, 扩容方式是利用数组的复制, 因此有一定的开销

2. LinkedList:内部使用双向链表的结构实现存储, LinkedList有一个内部类作为存放元素的单元, 里面有三个属性, 用来存放元素本身以及前后2个单元的引用, 另外LinkedList内部还有一个Header属性, 用来标识起始位置, LinkedList的第一个单元和最后一个单元都会指向header,

因此形成了一个双向链表结构.

3. LinkedList还额外实现了Deque接口, 所以LinkedList还可以当做队列来使用.

2.0.31 ==和equals的区别

==是运算符, 而equals是Object的基本方法, ==用于基本数据的类型比较, 或者是比较两个对象的引用是否相同, equals用于比较两个对象的值是否相等, 例如字符串的比较.

2.0.32 hashCode方法的作用

1. 如果两个对象equals方法相等, 那么hashCode一定相同
2. 如果两个对象的hashCode相同, 并不表示两个对象相同(只能表示hash碰撞相同), equals方法相同.

2.0.33 反射

简单来说, 在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法, 对于任意一个对象, 都能够调用他的任意方法和属性, 并且能够改变他的属性.

2.0.34 简述Java内存模型(JMM)

简单来说就是, java中存在一个主内存, java中所有变量都存在主内存中, 对所有线程进行共享, 而每个线程又存在自己工作内存, 工作内存存储的是主存中某些变量的拷贝, 线程对所有变量的操作并非发生在主存区, 而是发生在工作内存中, 线程之间是不能直接相互访问, 变量在程序中的传递主要依赖主存完成.

2.0.35 Java内存模型中的可见性, 原子性和有序性

可见性, volatile
原子性, 各种锁
有序性, 线程内有序

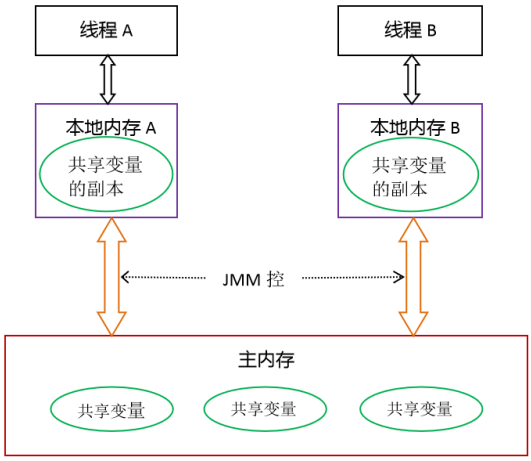

图 4:

2.0.36 happen-before原则

虽然有好几个, 但基本上描述模糊的就不写了

(1) 锁的happen-before, 就是同一个锁的unlock操作happen-before此锁的lock操作.

(2) 传递性: A hb b, b hb c; A happen-before C;

(3) 对象的构造函数在finalize方法之前.

2.0.37 wait/notify, await/singal

Condition 的 await，signal, singalAll 与 Object 的 wait, notify, notifyAll 都可以实现的需求，两者在使用上也是非常类似，都需要先获取某个锁之后才能调用，而不同的是 Object wait,notify 对应的是 synchronized 方式的锁，Condition await，singal 则对应的是 ReentrantLock （实现 Lock 接口的锁对象）对应的锁

下方是Condition的示例

2.0.38 多线程wait和sleep区别

(1) 主要在获得执行权和释放锁之间的区别, wait会释放执行权, 然后释放锁 , sleep 只会释放执行权

(2) 如果notifyAll() 如果有多个线程在等待, 只会有一个线程获得执行权.

2.0.39 Collection¡? extends Person¿ s

这叫泛型上限, 这样取出都是按照上限类型来运算的. 不会出现安全隐患

```
1  public class Message {
2          /** 当前消息数量*/
3          private int count = 0;
4          /** 信息存放最大限数*/
5          private int maximum = 20;
6          /** 重入锁*/
7          private Lock lock;
8          /** 生产者锁控制器*/
9          private Condition producerCondition;
10         /** 消费者锁控制器*/
11         private Condition consumerCondition;
12
13         public Message() {}
14
15         public Message(final Lock lock, final
               Condition producerCondition, final
               Condition consumerCondition) {
16                 this.lock = lock;
17                 this.producerCondition =
                       producerCondition;
18                 this.consumerCondition =
                       consumerCondition;
19         }
20
21         /**
22         * 生产消息
23         * */
```

```
24          public void set() {
25                  /** 获取锁*/
26                  lock.lock();
27                  try {
28                          if (count <= maximum) {
29                                  /** 生产一个消息*/
30                                  System.out.println("生
                                        产者 线程" + Thread.
                                        currentThread().
                                        getName() + "生产了
                                        一个消息，当前有" +
                                        (++count) + "个消
                                        息");
31                                  /** 唤醒等待的消费者*/
32                                  consumerCondition.
                                        signal();
33                          } else {
34                                  try {
35                                          /**
36                                          * 如果当前消息大
                                                于信息最大
                                                数 maximum
37                                          * 生产者进入睡眠
                                                等待状态/
38                                          * */
39                                          producerCondition
                                                .await();
40                                          System.out.
                                                println("生
                                                产者 线程" +
                                                 Thread.
                                                currentThread
                                                ().getName
                                                () + "进入睡
                                                眠");
41                                  } catch (
```

```
                                        InterruptedException
                                         e) {
42                                           e.
                                                 printStackTrace
                                                 ();
43                                      }
44                              }
45                      } finally {
46                              /** 释放锁*/
47                              lock.unlock();
48                      }
49              }
50
51              /**
52              * 消费消息
53              * */
54              public void get() {
55                      /** 获取锁*/
56                      lock.lock();
57                      try {
58                              if (count > 0) {
59                                      /** 消费一个消息*/
60                                      System.out.println("消
                                         费者 线程" + Thread.
                                         currentThread().
                                         getName() + "消费了
                                         一个消息，当前有" +
                                         (--count) + "个消
                                         息");
61                                      /** 唤醒等待的生产者*/
62                                      producerCondition.
                                         signal();
63                              } else {
64                                      try {
```

```
65                                          /** 如果没有消息，消费者进入睡眠等待状态*/
66                                          consumerCondition.await();
67                                          System.out.println("消费者 线程" + Thread.currentThread().getName() + "进入睡眠");
68                                  } catch (InterruptedException e) {
69                                          e.printStackTrace();
70                                  }
71                          }
72                  } finally {
73                          /** 释放锁*/
74                          lock.unlock();
75                  }
76          }
77
78  }
79
80  public class Producer implements Runnable {
81          private Message message;
82          public Producer(Message message) {
83                  this.message = message;
84          }
```

```
85
86          @Override
87          public void run() {
88                  while(true) {
89                          try {
90                                  Thread.sleep(500);
91                          } catch (InterruptedException
                                e) {
92                                  e.printStackTrace();
93                          }
94                          message.set();
95                  }
96          }
97
98  }
99  public class Consumer implements Runnable {
100         private Message message;
101         public Consumer(Message message) {
102                 this.message = message;
103         }
104
105         @Override
106         public void run() {
107                 while(true) {
108                         try {
109                                 Thread.sleep(1000);
110                         } catch (InterruptedException
                                e) {
111                                 e.printStackTrace();
112                         }
113                         message.get();
114                 }
115         }
```

```
116
117 }
118 import java.util.concurrent.locks.Condition;
119 import java.util.concurrent.locks.Lock;
120 import java.util.concurrent.locks.ReentrantLock;
121
122 public class App {
123         public static void main(String[] args) {
124                 /** 重入锁*/
125                 final Lock lock = new ReentrantLock();
126                 /** 生产者锁控制器*/
127                 final Condition producerCondition =
                        lock.newCondition();
128                 /** 消费者锁控制器*/
129                 final Condition consumerCondition =
                        lock.newCondition();
130                 final Message message = new Message(
                        lock, producerCondition,
                        consumerCondition);
131                 /** 建几个生产线程*/
132                 new Thread(new Producer(message)).
                        start();
133                 new Thread(new Producer(message)).
                        start();
134                 new Thread(new Producer(message)).
                        start();
135                 /** 建几个消费线程*/
136                 new Thread(new Consumer(message)).
                        start();
137                 new Thread(new Consumer(message)).
                        start();
138                 new Thread(new Consumer(message)).
                        start();
```

```
139                     new Thread(new Consumer(message)).
                            start();
140             }
141
142 }
```

2.0.40 线程的状态有哪些?

(1) 新建状态(NEW): 线程创建之后

(2) 可运行(RUNNING): 可能正在运行, 也可能正在等待时间片

(3) 阻塞(BLOCKED): 等待获取一个排它锁, 如果期限陈释放了锁就会结束此状态.

(4) 无线等待(WAITING): 等待其他线程显式地唤醒, 否则不会被分配CPU时间片片

(5) 限期等待(TIME_WAITING): 如果没人唤醒在一定时间内系统会自动唤醒

(6) 终止(TERMINATED): 可以是线程结束任务之后自己结束, 或者产生了异常而结束

线程创建之后处于New状态, 调用start()方法后开始运行, 线程这时候处于Ready可运行状态. 可运行状态的线程获得cpu时间片后就处于RUNNING状态. 当线程执行wait()方法之后, 线程进入WAITING(等待)状态. 进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态, 而TIME_WAITING超时等待状态相当于在等待状态的基础上增加了超时限制, 比如通过SLEEP方法或wait放假将java线程至于TIME_WAITING状态, 到超时之后, java线程将会发挥RUNNABLE状态. 当线程调用同步方法是,在没有获取到所的情况下,线程将会进入到BLOCK状态. 执行完Runnable的run()方法之后将会进入到TERMINATED状态.

2.0.41 创建线程的几种方式

(1) 继承Thread类创建线程

定义Thread类的子类, 并重写该类的run方法

创建实例

调用实例start()方法

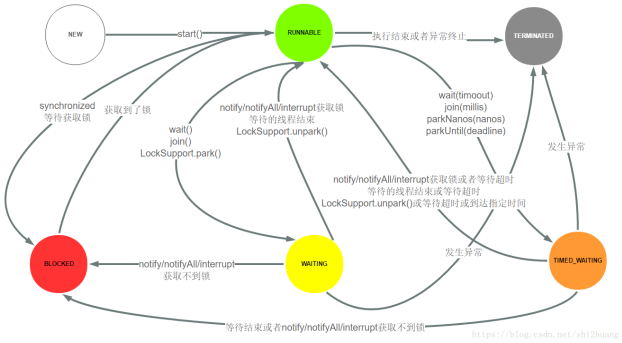

图 5:

(2) 实现Runnable接口创建线程

实现一个接口类, 重新run方法.

创建Runnable实现类实例, 并以此实例作为Thread的target来创建Thread对象, 该Thread对象才是真正的线程对象.

调用线程对象start方法来启动该线程

(3) 使用Callable和Future创建线程: 与Runnable相比Callable是有返回值的, 返回值通过FutureTask进行封装

创建Callable接口的实现类, 并实现call()方法, 该call()方法将作为线程执行体, 兵器有返回值

创建Callbale实现类实例, 使用FutureTask类来包装Callable对象, 该FutureTask对象封装了该Callable对象的call()方法的返回值.

使用FutureTask对象作为Thread对象的target创建biang启动新线程

调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

(4) 使用线程池例如Executor框架(工厂方法)

(5) 创建线程的方式的对比

1. Runnable 不可以抛出异常, Callable可以

2. Runnable 不可以有返回值, Callable 通过封装FutureTask 可以拿到返回值

2.0.42 synchronized锁升级: 无锁, 偏向锁, 轻量级锁, 重量级锁(与锁的优化一起学习

这个叫做锁的膨胀.

(1) 偏向锁, 初次执行到synchronized代码块的时候, 锁对象变成偏向锁, 通过CAS修改对象投里的锁标志位, 字面意思是"偏向于第一个获得它的线程"的锁. 会存储获取锁的线程的地址. 偏向锁解锁, 不需要修改对象头的markword, 减少了一次CAS操作, 锁不会释放, 但是遇到冲突, 会由JVM来进行判断升级. 执行完同步代码块之后, 线程并不会主动释放偏向锁, 当第二次达到同步代码块时, 线程会判断此时持有锁的线程是否就是自己(持有锁的线程ID也在对象头里), 如果是正常往下执行. 由于之前没有释放锁, 这里也就不需要重新加锁. 如果自始至终使用锁的线程只有一个, 很明显偏向锁几乎没有额外开销, 性能极高.

(2) 轻量级锁, 自旋锁, 地担忧第二个线程加入锁竞争, 偏向锁, 就升级为轻量级锁. 只有当某线程获取锁的时候, 发现该锁已经被占用, 只能等待其实方, 这才发生了锁的竞争. 在所竞争下, 没有抢到锁的线程将自旋, 即不停的循环判断锁是否能够被成功获取. 长时间的自选操作是非常消耗资源的, 一个线程持有锁, 其他线程就只能在原地空号CPU. 如果达到某个最大自旋次数, 会将轻量级锁升级为重量级锁. 当后续线程尝试获取锁时, 直接将自己挂起.

(3) 偏向锁, 假定条件只有一个线程去获取锁

(4) 轻量级锁, 假定条件是多个线程交替去获取锁

2.0.43 如何使用synchronized

1. 普通同步方法

3 JVM

3.0.1 说一下JVM中, 哪些是共享区, 哪些可以作为gc root

共享区: 方发区和堆

每个线程独有: 栈, 本地方法栈, 程序技术器

堆中, 从gc root可以找到一连串的对象, 就是正常的对象, 没有被找到的对象可以被回收.

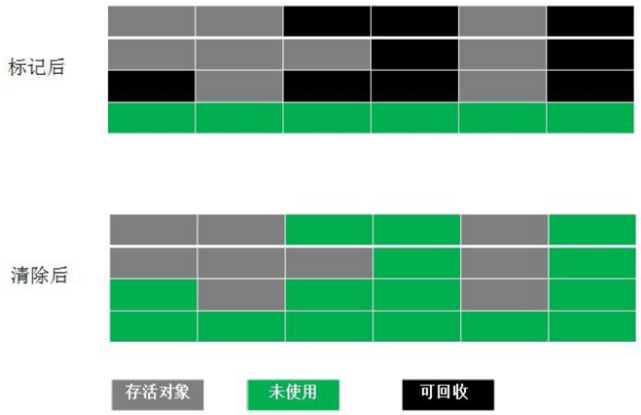

图 6:

3.0.2 你们项目如何排查JVM问题

1. 使用jvisualvm图形化查看内存的变化, 发现频繁的fullgc, 但是并没有出现oom现象, 可能是年轻代的内存不够, 对于大对象如果新生代放不下会直接放入老年代, 导致频繁fullgc, 通过增大新生代内存, fullgc减少, 证明修改有效.

2. 对于已经发生了oom异常的, 生成dump文件(-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/base)

使用jvisualvm等工具来分析dump文件, 更具dump文件找到异常的实例对象, 和异常的线程, 定位到具体的代码, 然后再进行详细的分析和调试

3.0.3 GC的三种收集方法: 标记清除, 标记整理, 复制算法的原理与特点, 分别用在什么地方, 如果让你优化收集方法, 有什么思路

1. 标记清除: 先标记, 标记完毕之后再清除, 缺点: 效率不高会产生碎片.
2. 标记整理: 标记完毕之后, 让所有存活的对象向一端移动
3. 复制算法: 分别8:1的Eden去和survivor区
4. 分代收集算法-重点

   一般将Java分成新生代和老年代, 新生代使用复制算法, 老年代使用标记整理算法.

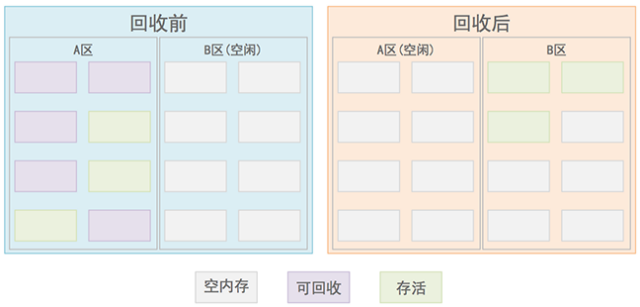

图 7:

3.0.4 JVM的主要组成部分及其作用?

JVM包含两个子系统和两个组件, 两个子系统为Class loader(类加载), Execution engin(执行引擎); 两个组件为 Runtime data area(运行时数据区), native Interface(本地接口)

1. Class loader: 根据给定的额全限定类名(如:java.lang.object)来装在class文件到Runtime data area中的method area.
2. Execution engine(执行引擎): 执行classes中的指令
3. native Interface(本地接口): 与native libraries交互, 是其他编程语言交互的接口.
4. Runtime data area(运行时数据区): 这就是我们常说的jvm的内存

作用: 首先通过编译器吧java代码转换成字节码, 类加载器(ClassLoader)再把字节码加载到内存中, 将其放在运行时数据区(Runtime data area)的方发区内, 而字节码文件知识jvm的一套指令集规范, 并不能直接交给底层操作系统去执行, 因此需要特定的命令解析器执行引擎(Execution Engine), 将字节码翻译成底层系统指令, 在交由CPU去执行, 而这个过程中需要调用其他语言的本地库接口(Native Interface) 来实现整个程序的功能.

Java程序运行机制步骤

1. 编码: IDEA等IDE进行编码java, 后缀.java
2. 编译: javac 将源代码编译成字节码文件,字节码文件的后缀名为.class

类的加载是将类的.class文件中的二进制数据读入到内存中, 将其放在运行时数据区的方法去内, 然后在堆区创建一个java.lang.Class对象, 用来封装类在方区内的数据结构.

3.0.5 JVM运行时数据区

运行时数据区由如下几个区域构成

1. 程序计数器(PC): 当前线程所执行字节码的行号指示器, 字节码解析器的工作是通过改变这个计数器的值, 来选去下一条需要执行的字节码指令.

2. java虚拟机栈(Java Virtual Machine Stacks): 用于存储局部变量表, 操作数栈, 动态链接, 方法出口等信息.

3. 本地方法栈(Native Method Stack) : 与虚拟机栈的作用是一样的, 只不过虚拟机栈是服务Java方法的, 而本地方法栈是为虚拟机调用Native方法服务的.

4. Java堆(Java Heap): Java 虚拟机中内存最大的一块, 是被所有线程共享的, 几乎所有的对象实例, 都在这里分配内存;

5. 方法区(Method Area): 用于存储已被虚拟机加载的类信息, 常量, 静态变量, 及时编译后的代码等数据.

3.0.6 JVM运行时数据区这些方法的关系

可以看到PC指针和虚拟机栈和本地方法栈是线程独有的. 而堆,方法区和运行时常量池是属于线程共享

3.0.7 永久代PermGen 和元空间Metaspace 区别

1. 永久代PermGen : 是jdk1.7 对于方发区的实现. 由于动态生成类的情况比较容易出现永久代的内存溢出, 抛出异常. 而且字符串存储在永久代中容易出现性能问题和内存溢出.

2. 元空间MetaSpace: 存在于本地内存.

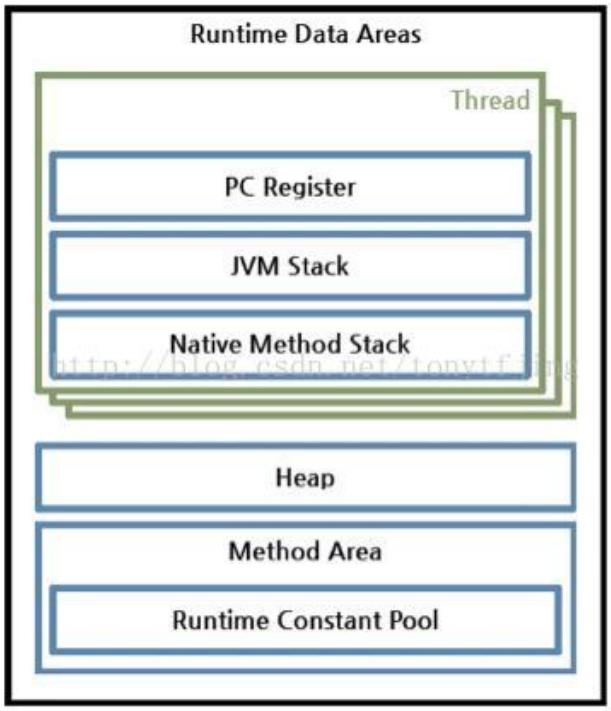
Runtime Data Areas
Thread
PC Register
JVM Stack
Native Method Stack
Heap
Method Area
Runtime Constant Pool

图 8:

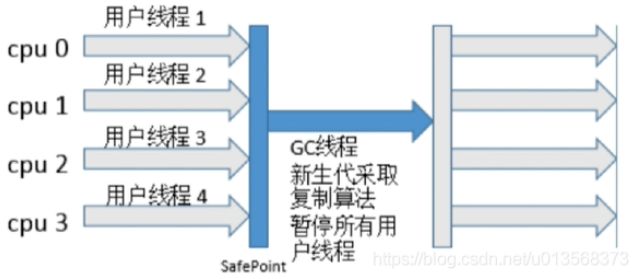

图 9:

3.0.8 说一下堆栈的区别?

物理地址

堆的物理地址分配对对象是不连续的. 因此, 性能慢些. 在GC的时候也要考虑到不连续的分配, 所以后各种算法. 比如, 标记-清除, 复制, 标记压缩, 分代(即新生代生活复制算法, 老年代使用标记压缩算法);

栈使用的是数据结构中的栈, 先进后出的原则, 物理地址分配是连续的. 所以性能快.

内存区别

堆因为是不连续的, 所以分配的内存是在**运行期**确认的, 因此大小不固定. 一般堆大小远大于栈.

栈是连续的, 所以分配的内存大小要在编译器就确认, 大小是固定的.

程序的可见度

堆对于整个应用程序都是共享, 可见的. 栈只对于线程是可见的. 所以也是线程私有. 他的生命周期和线程相同. TIPS:

1. 静态变量放在方法区.
2. 静态的对象还是放在堆.

3.0.9 常见的垃圾收集器?

(1) Serial收集器, 单线程收集器, 会stop the word.

(2) ParNew(Parallel Old)收集器, 是Serial收集器的多线程版本. 然后, 并行收集垃圾工作, 此时用户线程也是停止的状态

(3) Parallel Scavenge 收集器(新生代)

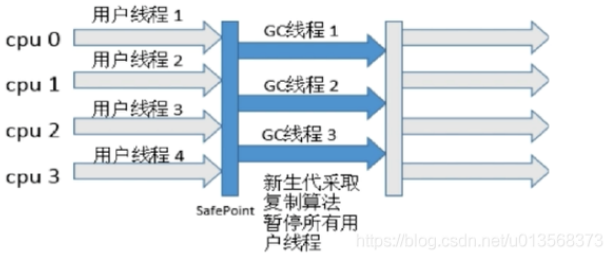

图 10:

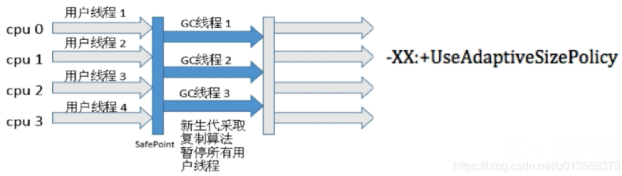

图 11:

多线程收集器, 同样是针对新生代. 停顿时间较短

(4) Serial Old 收集器(老年代)

使用标记整理算法收集老年代垃圾, 单线程.

(5) Parallel old 收集器(老年代)

标记整理算法, 多线程

(6) CMS收集器

简单来说, 在垃圾回收线程几乎能做到与用户线程同时工作, 使用标记清除算法. (7) G1 收集器

使用复制 + 标记 - 整理算法收集新生代和老年代垃圾.

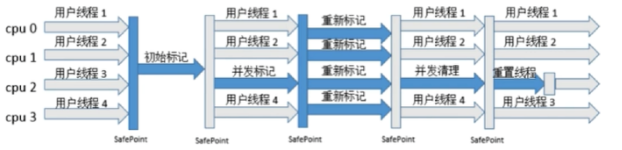

图 12:

3.0.10 内存分配与回收策略.

1. MinorGC 和 Full GC 有什么不同?

MINORGC: 新生代垃圾回收, 回收速度一般较快

MajorGC: 老年代GC, 回收速度较慢

FULLGC: 重GC, 会清理整个空间包括年轻代和老年代.

2. 什么时候对象进入老年代

(1) 大对象直接进入老年代

(2) 空间分配担保: 当TO被填满后当其中的对象还村或者, 剩下的对象直接存入老年代

(3) 年龄判定: 如果Survivor空间中相同年龄多有对象大小的总和大于Survivor空间的一半, 年龄大于或等于改年龄的对象就可以直接进入老年代, 如需达到要求的年龄

3.0.11 虚拟机性能监控和故障处理工具

jvisualvm 可视化监控.

3.0.12 简述JVM中类加载机制

类加载过程: 加载, 验证, 准备, 解析和初始化.

(1) 加载

1. 通过类的全限定名获取此类的二进制字节流

2. 将这个字节流所代表的的静态存储结构转化为方发区的运行时数据结构

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

(2) 验证

为了却表Class文件的字节流中包含的信息符合当前的虚拟机的要求, 并且不会危害虚拟机自身的安全.

(3) 准备

正式为类变量(static修饰的)分配内存并设置类变量初始值的节点, 这些变量所使用的的内存都将在**方法区**中进行分配

(4) 解析

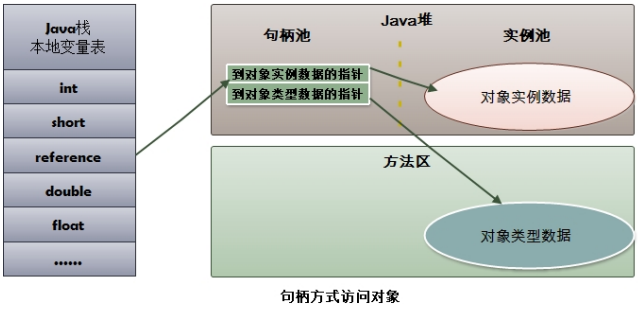

图 13:

虚拟机将常量池内的符号引用替换为直接引用的过程. 主要对类过接口, 字段, 类方法, 接口方法的解析, 主要是静态链接, 方法主要是静态方法和私有方法.

3.0.13 对象的访问定位?

目前主流的访问方式有句柄和直接指针两种方式.

1. 指针: 指向对象, 代表一个对象再内存中的起始地址
2. 句柄: 可以理解为指向指针的指针, 维护者对象的地址. 句柄不直接指向对象, 而是指向对象的地址(句柄不发生变化, 指向固定内存你地址), 再由对象的指针指向对象的真实内存地址.

句柄访问

Java堆中划分出一块内存作为句柄池, 引用中存储对象的句柄地址, 而句柄中包含了**对象实例数据**与**对象类型数据**各自的决堤地址信息, 具体构造如下图所示: **直接指针**

3.0.14 垃圾回收器的基本原理是什么?

可达性分析

GC采用有向图的方式记录和管理堆中的所有对象. 通过这种方式确定哪些对象是"可达的", 哪些对象是"不可达的", 当GC确定一些对象为"不可达"时, GC就有责任回收这些内存空间. 程序员可以手动执行System.gc(), 通知GC运行, 但是Java语言规范并不保证GC一定会执行.

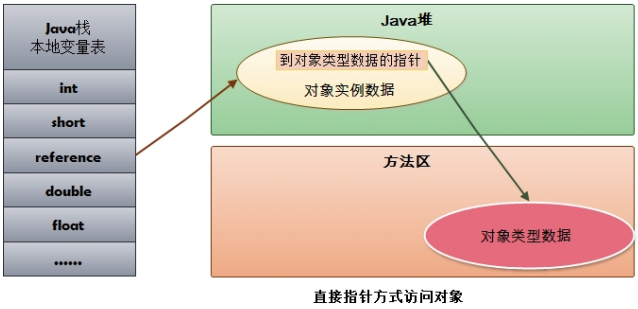

图 14:

引用计数法 为每个对象创建一个引用技术, 有对象引用时计数器+1, 引用被释放是技术-1, 当计数器为0时就可以被回收. 优缺点, 不能解决循环引用的问题.

3.0.15 在java中, 对象什么时候可以被垃圾回收?

当对象变的不可触及的时候, 这个对象就可以被回收了, 垃圾回收不会发生在永久代, 如果永久代满了或者是超过了临界值, 会触发完全垃圾回收(full gc), 会导致Stop-the-world.

3.0.16 如何判断对象已经死亡?

(1) 引用计数法, 标记为零. 缺点难以解决互相引用问题

(2) 可达性分析, 当一个对象到GC Root对象没有任何路径可达.

(3) 上面两种都是暂时处于缓刑阶段, 真正宣告一个对象死亡, 至少要经历两次标记过程.

3.0.17 简述强, 软, 弱, 虚引用?

(1) 强引用

如果一个对象具有强引用, 垃圾回收期绝不会回收它

(2) 软引用(SoftRef)

如果内存足够, 垃圾回收期就不会回收塔, 如果内存不足了, 就会回收这些对象的内存. 可以实现内存敏感的告诉缓存.

(3) 弱引用(WeakRef)

弱引用关联的对象只能生存到下一次垃圾回收之前.

(4) 虚引用(ReferenceQue)

如果一个对象仅持有虚引用, 那么他就和没有任何引用一样, 在任何时候都可能被垃圾回收. 必须和引用队列一起联合使用.

区别:

软弱引用都是发生在垃圾回收动作之后, 虚引用发生在垃圾回收动作之前.

4 Redis

4.0.1 什么是Redis?

1. 高性能非关系型数据库
2. 可以存储五种不同类型的额值之间的映射. 键的类型只能为字符串, 值支持五种类型数据:字符串(string), 列表(list), 集合(set), 散列表(hash), 有序集合(sorted set).
3. redis数据是存在内存中的, 所以读写速度非常快.

4.0.2 简述Redis单线程模型?

实现方式

(1) I/O多路复用

简单来说, 可以使用I/O多路复用来坚定多个socket连接, 然后将感兴趣的时间注册到内核中并监听每个事件是否发生.

(2) 基于事件驱动

服务器需要处理两类事件, 文件事件; 时间事件.

当被监听的套接字准备好执行连接应答(accept), 读取(read), 写入(write), 关闭(close)等操作时, 与操作相对应的文件事件就会产生, 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件.

文件事件处理器(file event handler) 主要包含4个部分: 多个socket(客户端链接), IO多路复用, 文件事件分派; 事件处理;

4.0.3 Redis五种类型数据的实现方式

字符串结构SDS和C中char[]有什么不同

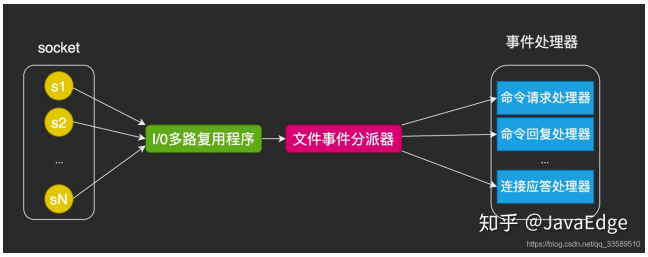

图 15:

表 2: 实现

类型	编码
STRING	INT(整形, 在String中存储整形会是的)
STRING	EMBSTR(简单动态字符串, 对于短小的string(44位字符)会使用这种结构)
STRING	RAW(简单动态字符串, 对于稍微长一点的string会使用(44位字符)这种结构)
LIST	QUICKLIST(快表)
LIST	LINKEDLIST(快表)
SET	INTSET(整数集合)
SET	HT(哈希表)
ZSET	ZIPLIST(压缩列表)
ZSET	SKIPLIST(跳表)
HASH	ZIPLIST(压缩列表)
HASH	HT(哈希表)

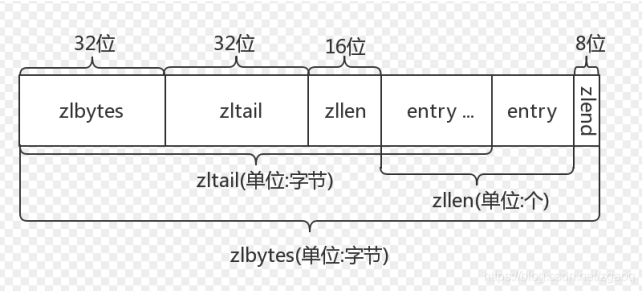

图 16:

1. 获取SDS中字符串的长度因为SDS中存储了字符串的长度len属性, 直接访问, 时间复杂度O(1), 对于C语言获取字符串的长度需要经过遍历, 时间复杂度O(n).
2. 避免缓冲区溢出, 会检查SDS中属性, free(空闲空间)能够实现字符串的扩充判断. 不足会重新申请空间.
3. SDS支持空间预分配, 扩展的内存比实际需要的多
4. SDS支持空间惰性释放, 字符串缩短之后, 不立即进行空间回收操作. SDS也提供相应API, 可以对冗余空间进行回收.
5. 可以存储二进制, 因为SDS不以回车符号进行终止的判定.

4.0.4 redis字典的底层实现hashTable相关问题

1. 解决冲突: 链地址法, 即使用数组+链表的方式实现.
2. 扩容: 有两个指针h[0] 和 h[1], h[1] 用来备份, 当h[0], 满了使用渐进值hash, 插入都插入h[1], 查找两个表都进行查找.

4.0.5 压缩链表原理ziplist

连续内存, 包含多个节点entry.

4.0.6 zset

本质是舵机链表并有序. skiplist与平衡树, 哈希表的比较

1. skiplist和各种平衡树的元素排序是有序的, 而哈希表不是有序的, 因此, 在哈希表上智能做单个key的查找, 不适宜做范围查找.

2. 在做范围查找的时候, 平衡树比skiplist操作要复杂. 在平衡树上, 需要做一步回退操作. 而在skiplist上进行范围查找就非常简单, 只要找到最小值之后对第一层链表进行若干部遍历就可以实现.

3. 平衡树插入和删除操作, 会引起结构调整, 操作复杂, skiplist的插入和删除只需要修改相邻节点的指针, 操作简单又快速.

4.0.7 AOF和RDB两种持久化方式区别

1. AOF存储命令, RDB存储数据.

2. AOF文件因为存储命令, 所以在redis启动的时候加载aof会比加载rdb要慢.

3. Redis 4.0 之后启动了混合模式, AOF不需要是全量日志, 只要保存前一次RDB存储开始到这段时间增量AOF日志即可.

4.0.8 Redis中过期策略和缓存淘汰机制

1. 定期删除: redis默认每隔100ms随机对key检查, 有过期的key则进行删除. 容易导致很多过期的key没被发现

2. 惰性删除: 获取某个key的时候, redis会检查一下, 如果过期了就进行删除.

4.0.9 为什么要使用Redis

1. 高性能: 内存的读取比硬盘乃至固态硬盘的读取速度都要快得多

2. 高并发: 直接操作缓存能够承受的请求是远远大于直接访问数据库的, 所以我们可以考虑把数据库中的部分数据转移到缓存中去, 这样用户的一部分请求会直接请求缓存这里而不用经过数据库.

3. 高性能: 使用多路I/O复用木星, 非阻塞IO;

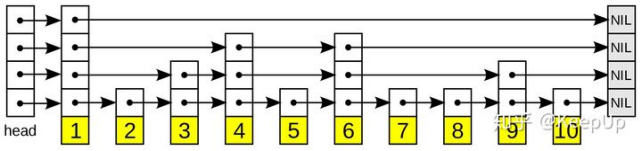

图 17:

4.0.10 Redis底层实现跳表介绍一下

跳表是带多级索引的链表, 时间复杂度O(lgn), 所能实现的功能和红黑树差不多, 但是跳表有一个区间查找的优势, 红黑树没有.

1. 表头: 负责维护跳表的节点指针.
2. 跳跃表节点: 保存着元素值, 以及多个层.
3. 层: 保存着指向其他元素的指针, 高层的指针越过的元素数量大于等于底层的指针, 为了提高查找效率, 程序总是从高层先开始访问, 然后随着元素值范围的缩小, 慢慢降低层次.
4. 表尾: 全部由NULL组成.

4.0.11 为什么要使用Redis而不用map/guavaCache做缓存

1. guavaCache实现的是本地缓存, 最主要的特点是轻量化以及快速, 生命周期随着jvm的销毁而结束, 冰洁在多实例的情况下, 每个实例都需要个字保存一份缓存, 缓存容易出现不一致性.
2. 使用redis之类的缓存称为分布式缓存, 在多实例的情况下, 各实例公用一份缓存数据, 缓存具有一致性.

4.0.12 分布式锁如何使用redis实现

1. setnx命令原子性实现.

4.0.13 Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis用于缓存的内存不足时, 怎么处理需要新写入且需要申额外空间的数据. 全局的键空间选择性移除

1. noeviction: 当内存不足以容纳新写入数据时, 新写入操作会报错.
2. allkeys-lru: 当内存不足以容纳新写入数据时, 在键空间中, 移除最近最少使用的key.
3. allkeys-random: 当内存不足以容纳新写入数据时, 在键空间随机移除某个key.

设置过期时间的键空间选择性移除

1. volatile-lru: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 移除最近最少使用的key.
2. volatile-random: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 随机移除某个key.
3. volatile-ttl: 当内存不足以容纳新写入的数据时, 在设置了过期时间的键空间中, 有更早过期时间的key优先移除.

4.0.14 Redis事物的概念

Redis事物的本质通过MULTI, EXEC, WATCH等一组命令的集合.

1. 事物开始 MULTI
2. 命令入队
3. 事务执行 EXEC

* 事务执行过程中, 如果服务端收到有EXEC, DISCARD, WATCH, MULTI之外的请求, 将会把请求放入队列中排队. 简单介绍一下watch, 当watch的变量在事务过程中发生了改变, 那么事务失败, 拒绝执行事物.

```
1        >watch 'name'
2        >multi
3        >set "name" "peter"
4        >exec
5        (nil)
```

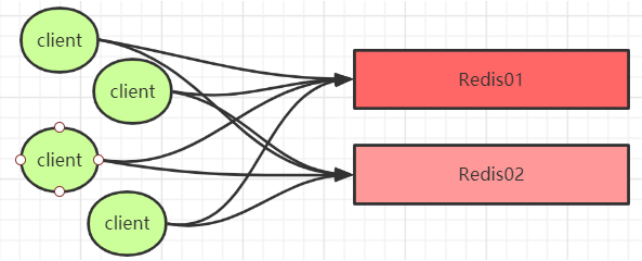

图 18:

4.0.15 RedisSharding

简单来说就是多个client 连接多个redis, 然后通过一致性哈希算法, 来确定访问的key+访问的客户端名字在哪一台redis上. 采用的算法是MURMUR_HASH,

4.0.16 缓存雪崩

缓存同一时间大面积的失效, 所以, 后面的请求都会落到数据库上, 造成数据库短时间内承受大量请求而崩掉

解决方案

1. 缓存数据的过期时间设置随机, 防止同一时间大量数据过期现象发生.

4.0.17 缓存穿透

缓存和数据库中都没有数据, 导致所有的请求都落在数据库中, 造成数据库短时间内承受大量请求而崩掉.

解决方案

1. 从缓存中取不到的数据, 在数据库中也没有取到, 这时也可以将key-value对写为key-null, 缓存时间设定为30s.

4.0.18 缓存击穿

缓存中没有但数据库中有的数据, 由于并发用户特别多, 同时读缓存没读到数据, 又同时去数据库取数据, 引起数据库压力瞬间增大, 造成过大压力, 和缓存雪崩不同的是, 缓存击穿指并发查询同一条数据, 缓存雪崩是不同数据都过期了, 很多数据都查不到, 从而查数据库.

解决方案

1. 设置热点数据永不过期

4.0.19 缓存预热

系统上线后, 将相关的缓存数据直接加载到缓存系统. 这样就可以避免在用户请求的时候, 先查询数据库, 然后再讲数据缓存的问题, 用户直接查询事先被预热的缓存数据.

解决方案

1. 定时刷新缓存;

4.0.20 Redis6.0 为什么要引入多线程呢?

Redis将所有数据放在内存中, 内存的相应市场大约100ns, 对于小数据包, Redis服务器可以处理8w到10wQPS, 这也是Redis处理的极限了, 对于80%的公司来说, 单线程的redis已经足够使用了.

但随着越来越复杂的业务场景, 需要更大的QPS.

1. 可以充分利用服务器CPU资源, 目前主线程只能利用一个核
2. 多线程任务可以分摊Redis同步IO读写负荷.

4.0.21 Redis主从复制模式

1. 完全同步:刚开始, 主服务器发送RDB文件给从服务器, 实现主从同步.
2. 部分同步:当连接由于网络原因断开的时候, 将中间断开的执行的写命令发送给从服务器. 实现同步.

4.0.22 Redis中持久化机制

1.

5 计算机网络

5.0.1 计算机网络分层

计算机网络OSI模型是七层, 如果是TCP/IP 应用层(HTTP/FTP) - 应用层

- 表示层 - 会话层

传输层(UDP/TCP) - 传输层

网际层(IP) - 网络层

主机至网络层 - 数据链路层 - 物理层

5.0.2 TCP和UDP区别?

UDP: 面向报文, 支持1对1, 1对多, 多对1的交互通信

TCP: 面向连接, 提供可靠交付, 有流量控制, 拥塞控制, 提供全双工通信, 面向字节流. TCP是点对点的.

5.0.3 TCP三次握手相关问题

为什么三次握手而不是两次握手: 主要是为了防止已失效的链接请求报文段突然又传送到了B, 因而产生错误. 如果只要两次握手就建立连接, 那么如果A第一次请求丢失了, A 又发送了一次请求, 但是这一次传输结束了, A上一次丢失的请求再次被传送到了B, 如果建立了连接, 但是A现在一直不回应B, 导致B浪费了资源.

5.0.4 TCP四次挥手问题

关闭连接的时候, 当Server端收到FIN报文时候, 很可能并不会立即关闭SOCKET, 所以只能先回复一个ACK报文, 告诉Client端, 你发送的FIN我收到了, 只有等我Server端所有的报文都发送完了, 我才能发送FIN报文. 不能一起发送, 所以需要四步挥手.

5.0.5 TCP协议-如何保证传输的可靠性

超时重传: 简单理解就是发送在发送完数据后等待一个时间, 时间到大没有接收到ACK报文, 那么对刚才发送的数据进行重新发送

连接管理: 使用三次握手和四次挥手

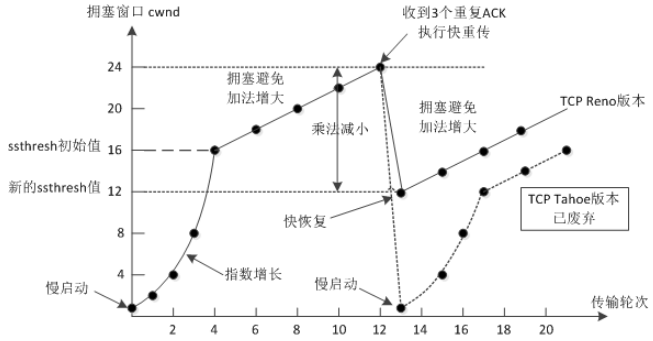

图 19:

流量控制: TCP根据接收端对数据的处理能力, 决定发送端的发送速度, 这个机制就是流量控制. TCP协议的包头信息当中, 有一个16位字段的窗口大小, 发送方更具ACK报文里的窗口大小的值的改变自己的发送数据

拥塞控制: 慢开始, 拥塞避免, 快重传, 快恢复.

慢开始算法的思路就是, 不要一开始就发送大量的数据, 先探测一下网络的拥塞程度, 也就是说由小到大主键增加拥塞窗口的大小.

拥塞避免算法让拥塞窗口缓慢增长, 即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1, 而不是加倍. 这样拥塞窗口按线性归路缓慢增长.

快重传和快恢复: 发送方只要一脸收到三个重复确认就引动立即重传对方尚未收到的报文段, 而不必继续等待设置的重传计时器时间到器.

5.0.6 Cookie作用, 安全性问题和Session的比较

(1) Cookie是服务器发送到用户浏览器并保存在本地的一小块数据, 他会在浏览器之后向统一服务器再次发送请求时被携带上.

(2) Session 存储在服务器端. 使用Session维护用户登录状态的过程如下: (需要cookie作为传输机制) 用户进行登陆是, 用户提交包含用户名和密码的表单, 放入HTTP请求报文中; 服务器校验该用户名和密码, 如果正确则把用户信息存储到Redis中, 他在Redis中的key成为SessionId.

服务器返回的响应报文Set-Cookie首部字段包含了这个SessionID, 客户端收到响应报文之后将该Cookie值存入浏览器中.客户端之后对同一个服务器进行请求时会包含该Cookie值, 服务器收到之后提取出SessionID, 从Redis中取出用户信息, 继续之前的业务操作.

session 的运行依赖 session id, 而 session id 是存在 cookie 中的, 也就是,如果浏览器禁用了 cookie, 同时 session 也会失效(但是可以通过其它方式实现, 比如在 url 中传递 session_id)

cookie存在大小限制, 单个不超过4k, 浏览器中Cookie个数也有限制. Session没有大小限制, 是和服务器内存有关的.

5.0.7 HTTP1.1 和 HTTP1.0的比较

HTTP1.1 默认长链接, 长连接只需要建立一次TCP连接, 进行多次HTTP通信. HTTP1.0 默认短连接, 每进行一次HTTP通信就要新建一个TCP连接.

5.0.8 HTTPS加密

使用SSL连接. HTTPS采用混合加密算法, 使用非对称加密和对称加密, 非对称加密用于传输对称秘钥来保证传输过程的安全性, 之后使用对称秘钥加密进行通信来保证通信过程的效率.

HTTPS加密过程:

1. 客户使用HTTPS的URL访问web服务器, 要求与Web服务器建立SSL链接.
2. Web服务器收到客户端请求后, 会将网站的证书信息(证书中包含公钥)传送一份给客户端.
3. 客户端的浏览器与web服务器开始写上SSL链接的安全登记, 也就是信息加密登记.
4. 客户端的浏览器更具双方同意的安全登记, 建立会话秘钥, 然后利用网站的公钥将会话秘钥加密, 并传送给网站
5. Web服务器利用自己的私钥解密出会话秘钥.
6. Web服务器利用会话秘钥加密与客户端之间的通信.

缺点: 因为需要进行加密解密等过程, 因此速度会更慢, 需要支付证书授权的高额费用.

5.0.9 输入网址发生的事情

1. 浏览器查找该域名的IP地址
2. 浏览器更具解析得到的IP地址向web服务器发送一个HTTP请求.
3. 服务器收到请求并进行处理
4. 服务器返回一个响应.
5. 浏览器对该响应进行解码, 渲染显示.
6. 页面显示完成后, 浏览器发送异步请求.

6 mysql

6.0.1 隔离级别

读已提交(read committed): 简单来说select * from account, 读出来的数据都是已经提交的数据.

可以出现不可重复读(一个事物两次读到的事物不一样)和幻读(插入一条记录)

读未提交(read uncommitted) : 简单来说 select * from account, 读出来的数据是另一个事物还没有提交的数据,

可以出现脏读(无论是脏写还是脏读，都是因为一个事务去更新或者查询了另外一个还没提交的事务更新过的数据。因为另外一个事务还没提交，所以它随时可能会回滚，那么必然导致你更新的数据就没了，或者你之前查询到的数据就没了，这就是脏写和脏读两种场景。): 简单的说, 就是A事物在还没commit数据的情况下, b事物使用了A事物的数据, 但是A事物出现了回滚操作, 导致B事物读出来的数据是脏数据. 可以出现不可重复度, 可以出现幻读. (感觉是最不靠谱的一个级别)

可重复读(RR: Repeatable read) : 简单来说, 就是A事物插入了一条新数据, B事物没看到, 因为可重复读, 但是B是可以操作这条新数据的.

串行化(serializable): 只能一个一个处理, 没什么问题, 但是效率太低一般不考虑.

提示: 不可重复读对应的是修改即Update，幻读对应的是插入即Insert.

TIPS: 可见，幻读就是没有读到的记录，以为不存在，但其实是可以更新成功的，并且，更新成功后，再次读取，就出现了。

6.0.2 ACID

原子性: 不可分割
一致性: 数据完整性, 例如金钱的综述
隔离性: 不被打扰
持久性: 永久保存

6.0.3 乐观锁和悲观锁

悲观锁, 指的是在整个数据处理过程中, 将数据处于锁定状态. 悲观锁的实现, 往往依靠数据库提供的锁机制.

乐观锁的实现, 基于并发控制的CAS理论.

6.0.4 MVCC

MVCC用于提交读和可重复读这两种隔离级别. 使用undolog实现

主要依靠事务版本号来实现读已提交和可重复读.

当开始一个新事物时, 该事物的版本号肯定会大于当前所有数据行快照的创建的版本好

6.0.5 B+/B树之间的比较

(1) B+树空间利用率更高, 可减少IO次数.

(2) B+树所有关键字的查询路径长度相同(因为关键字在叶子节点), 导致每一次查询的效率相当, 更加稳定.

(3) b+叶子节点有指针, 支持between查找很方便.

6.0.6 聚集索引&非聚集索引

简单来说, 一个表只有一个聚集索引, 类似于主键索引.

一个表可以有多个非聚集索引, 打比方, 新华字典A-Z的查询事主键索引, 新华字典偏旁查询事非聚集索引. 是会跳跃的.

6.0.7 创建索引的优点

(1) 当数据量增大的时候, 索引可以极大的加快数据的查找速度

6.0.8 创建索引的缺点

(1) 对于一个表而言, 不单单要维护数据的存储, 也要维护索引的存储. (空间增大)

(2) 对表中的数据进行修改的时候, 索引也要哦进行动态维护, 这样就降低了数据的维护速度.

6.0.9 MYSQL优化

(1) 最左前缀法则

(2) 较长的数据列建立前缀索引

(3) 常查询数字建立索引或者组合索引

(4) 分解大连接查询, 分解成对每一个表进行一次单表查询. 可以对缓存更好效的应用. 即时其中一个表发生变化, 对其他表的查询缓存依然可以使用.

6.0.10 InnoDB & MyISAM

(1) myISAM 支持全文索引, innodb 也支持全文索引, 简单来说innodb比myISAM强太多了

另外inndb 对于全文索引有要求, 有一个最小搜索长度. 和最大搜索长度. 比like 之类快很多.

6.0.11 创建存储过程

(1) 创建存储过程比单独的sql语句要快

(2) 可以快速进行测试

6.0.12 热备份和冷备份

冷备份: 因为mysql是基于文件的, 所以, 我们在mysql关闭的时候, 直接使用 copy 拷贝一份即可.

热备份: 使用mysqldump 命令对正在运行的mysql程序的数据库进行备份

6.0.13 Innode加锁

(1) 对于update,delete和insert语句, Innorb会自动给设计数据集加排它锁(X), 对于普通select语句Innodb不会加任何锁; 事物可以通过以下语句显示给记录集加共享锁或排他锁. (2) 共享锁(S): select * from tableName where ... LOCK IN SHARE MODE(可以查看但无法修改和删除一种数据锁, 其他用户可以加共享锁)

(3) 排它锁(X): SELECT * from tableName where ... FOR UPDATE(独占锁, 其他任何事物都不能对A加任何类型的锁, 直到自己释放了锁.)

共享锁, 主要用在确保没人对这个记录进行UPDATE或者DELETE操作.

6.0.14 INNODB解决死锁

实际上在INNODB发现死锁之后, 会计算出两个事物各自插入,更新或者删除的数据量来判定两个事物的大小. 也就是哪个事物所改变的记录条数越多, 在死锁中就越不会被回滚掉.

6.0.15 Mysql锁你了解哪些

按照锁的粒度区分

1. 行锁, 锁某行数据, 锁力度最小, 并发度最高
2. 表锁, 锁整张表, 锁力度最大, 并发度低
3. 间隙锁, 锁的是一个区间

按照怕他性区分

1. 共享锁: 也就是读锁
2. 排它锁: 也就是写锁

还可以分为:

1. 乐观锁, 并不会真正的去锁某行记录, 而是通过一个版本号来实现的.
2. 悲观锁, 上面所说的行锁, 表锁, 都是悲观锁.

6.0.16 Mysql数据库中, 什么情况下设置了索引但是无法使用?

1. 没有符合最左前缀原则
2. 字段进行了隐私数据类型转化
3. 走索引没有全表扫描效率高

7 操作系统

7.0.1 协程与线程进行比较

协程, 是一种用户态的轻量级线程, 写成的调用完全由用户控制. 协程调度切换时, 将寄存器上下文和栈保存到其他地方, 在切回来的时候, 恢复先前保存的寄存器上下文和栈, 直接操作栈则基本没有内核切换开销, 可以不加锁访问全局变量.

(1) 一个线程可以有多个协程, 一个进程也可以单独拥有多个协程.

(2) 线程进程都是同步机制, 而协程则是异步

7.0.2 进程之间的通信方式有哪些?

(1) 命名管道FIFO: 半双工方式

(2) 消息队列: 克服了信号传递信息少, 管道只能承载无格式字节流以及缓冲区大小受限等缺点.

(3) 共享内存: 共享内存你是最快的IPC方式.

(4) 信号量: 信号量是一个计数器.

(5) 套接字:socket 可以用于不同机器间的进程通信.

(6) 信号: 信号是一种比较复杂的通信方式, 用于通知接收进程某个事件已经发生, 比如linux中的kill命令通知进程进行关闭.

7.0.3 进程调度算法

(1) 先来先服务FCFS

(2) 短作业有限SJF

(3) 高优先权优先, 非抢占式优先权算法 & 强占式优先权调度算法

(4) 高响应比优先调度算法可以克服SJF长作业的饥饿

(5) 基于时间片轮转调度算法

7.0.4 epoll和poll的区别

1. select模型, 使用的是数组来存储Socket连接文件描述符, 容量是固定的, 需要通过轮询来判断是否发生了IO事件

2. poll模型, 使用的是链表来存储Socket连接文件描述符, 容量是不固定的, 童谣需要通过轮询来判断是否发生了IO事件

3. epoll模型, epoll和poll是完全不同的, epoll是一种事件通知模型, 大发生了IO事件时, 应用程序才进行IO操作, 不需要像poll模型那样主动去轮询

8 设计模式

8.0.1 多线程下单例设计模式

(1) 饿汉式不会出现安全问题, 懒汉式会出现(同时创建的时候会有问题, 两个线程).

(2) 懒汉式安全隐患解决

(3) 饿汉式, 在静态属性中就获取了对象, 懒汉式是去得到对象的时候获取, 导致了懒汉式可能有问题.

```
1  package com.lf.shejimoshi;
2
3  /**
4  * @classDesc: 类描述懒汉式单例测试类:()
5  * @author baobaolan
6  * @createTime 年月日2018110
7  * @version v1.0
8  */
9  public class SingletonTest {
10         /**
11         * @functionDesc: 功能描述测试懒汉式单例模式:()
12         * @author baobaolan
13         * @createTime 年月日2018110
14         * @version v1.0
15         */
16         public static void main(String[] args) {
17                 Student s1 = Student.getStudent();
18                 Student s2 = Student.getStudent();
19                 System.out.println(s1==s2);
20         }
21
```

```
22 }
23
24 /**
25 * @classDesc: 类描述学生类:()
26 * @author baobaolan
27 * @createTime 年月日2018110
28 * @version v1.0
29 */
30 class Student{
31
32         //定义全局变量
33         private static Student student;
34
35         //私有化构造函数
36         private Student(){
37
38         }
39
40         /**
41         * @functionDesc: 功能描述对外暴露方法:()
42         * @author baobaolan
43         * @createTime 年月日2018110
44         * @version v1.0
45         */
46         public static Student getStudent(){
47                 if(student==null){
48                         //加上同步锁，保证线程安全
49                         synchronized(Student.class){
50                                 student = new Student
                                        ();
51                         }
52                 }
53                 return student;
```

```
54          }
55  }
```

```
1   package com.lf.shejimoshi;
2
3   /**
4   * @classDesc: 类描述测试类:()
5   * @author baobaolan
6   * @createTime 年月日2018110
7   * @version v1.0
8   */
9   public class Singleton2Test {
10
11          public static void main(String[] args) {
12
13                  Teacher teacher1 = Teacher.getTeacher
                        ();
14                  Teacher teacher2 = Teacher.getTeacher
                        ();
15                  System.out.println(teacher1==teacher2)
                        ;
16
17          }
18
19  }
20
21  /**
22  * @classDesc: 类描述饿汉式单例:()
23  * @author baobaolan
24  * @createTime 年月日2018110
25  * @version v1.0
26  */
27  class Teacher{
```

```
28          //类加载的时候初始化一次
29          private static final Teacher teacher = new
                Teacher();
30          //私有化构造函数
31          private Teacher(){
32                  super();
33          }
34          /**
35          * @functionDesc: 功能描述对外暴露的方法:()
36          * @author baobaolan
37          * @createTime 年月日2018110
38          * @version v1.0
39          */
40          public static Teacher getTeacher(){
41                  return teacher;
42          }
43
44  }
```

8.0.2 为什么在wait代码块中要用while而不用if

因为单个生产者单个消费者, 没什么问题. 如果是一个生产者两个消费者的话会有问题.

因为线程唤醒的话, 会直接在wait() 下面执行, 然后如果是while的话, 可以进入重新判断. 否则可能造成数据溢出.

```
1   /*生产和消费
2
3   */
4   package multiThread;
5
6   class SynStack
7   {
8           private char[] data = new char[6];
```

```
9         private int cnt = 0; //表示数组有效元素的个数
10
11        public synchronized void push(char ch)
12        {
13                if (cnt >= data.length)
14                {
15                        try
16                        {
17                                System.out.println("生产线程"+Thread.currentThread().getName()+"准备休眠");
18                                this.wait();
19                                System.out.println("生产线程"+Thread.currentThread().getName()+"休眠结束了");
20                        }
21                        catch (Exception e)
22                        {
23                                e.printStackTrace();
24                        }
25                }
26                this.notify();
27                data[cnt] = ch;
28                ++cnt;
29                System.out.printf("生产线程"+Thread.currentThread().getName()+"正在生产第%个产品，该产品是d: %c\n", cnt, ch);
30        }
31
32        public synchronized char pop()
33        {
```

```
34                  char ch;
35                  if (cnt <= 0)
36                  {
37                          try
38                          {
39                                  System.out.println("消费线程"+Thread.currentThread().getName()+"准备休眠");
40                                  this.wait();
41                                  System.out.println("消费线程"+Thread.currentThread().getName()+"休眠结束了");
42                          }
43                          catch (Exception e)
44                          {
45                                  e.printStackTrace();
46                          }
47                  }
48                  this.notify();
49                  ch = data[cnt-1];
50                  System.out.printf("消费线程"+Thread.currentThread().getName()+"正在消费第%个产品，该产品是d: %c\n", cnt, ch);
51                  --cnt;
52                  return ch;
53          }
54  }
55
56  class Producer implements Runnable
57  {
58          private SynStack ss = null;
```

```
59          public Producer(SynStack ss)
60          {
61                  this.ss = ss;
62          }
63
64          public void run()
65          {
66                  char ch;
67                  for (int i=0; i<10; ++i)
68                  {
69
70                          ch = (char)('a'+i);
71                          ss.push(ch);
72                  }
73          }
74  }
75
76  class Consumer implements Runnable
77  {
78          private SynStack ss = null;
79
80          public Consumer(SynStack ss)
81          {
82                  this.ss = ss;
83          }
84
85          public void run()
86          {
87                  for (int i=0; i<10; ++i)
88                  {
89                          /*try{
90                                  Thread.sleep(100);
91                          }
```

```
92                          catch (Exception e){
93                          }*/
94
95                          //System.out.printf("ss.pop();
96                  }
97          }
98  }
99
100
101 public class TestPC2
102 {
103         public static void main(String[] args)
104         {
105                 SynStack ss = new SynStack();
106                 Producer p = new Producer(ss);
107                 Consumer c = new Consumer(ss);
108
109
110                 Thread t1 = new Thread(p);
111                 t1.setName("号1");
112                 t1.start();
113                 /*Thread t2 = new Thread(p);
114                 t2.setName号("2");
115                 t2.start();*/
116
117                 Thread t6 = new Thread(c);
118                 t6.setName("号6");
119                 t6.start();
120                 Thread t7 = new Thread(c);
121                 t7.setName("号7");
122                 t7.start();
123         }
124 }
```

8.0.3 Serializable

序列化接口, 只有实现这个接口才能序列化, 默认计算一个serialVersionUID, 可以进行自定义.

9 附录: 相关样例

第一个公式

$$F = ma = aa \tag{1}$$

我们可以知道牛顿得出了与物体质量和加速度之间的关系$F = ma$ 换行公式:

$$\begin{aligned} y &= ax \\ &= bx \end{aligned} \tag{2}$$

9.0.1 小练习

$$v = \frac{x}{y} \tag{3}$$

$$y = e^x \tag{4}$$

$$y = ax^2 + bx + c \tag{5}$$

$$F = G\frac{Mm}{r^2} \tag{6}$$

$$y = 4\pi\frac{\sin x}{\ln x^2} \tag{7}$$

$$y = \sum_{i=1}^{n} x^2 + 1 \tag{8}$$

$$i = \int_1^2 x^2 + \tan x \mathrm{d}x \tag{9}$$

$$A = \begin{bmatrix} 1 & 1 & 3 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{bmatrix} \tag{10}$$

9.0.2 三级标题

接下来将开始书写正文

1. 第一个问题：
2. 第二个问题：
3. 第三个问题：

9.0.3 第二个三级标题

9.0.4 第三个三级标题

10 公式的写作

第一个公式

$$F = ma \tag{11}$$

我们可以知道牛顿得出了与物体质量和加速度之间的关系$F = ma$

换行公式：

$$\begin{aligned} y &= ax \\ &= bx \end{aligned} \tag{12}$$

10.0.1 练习

$$v = \frac{x}{y} \tag{13}$$

$$y = e^x \tag{14}$$

$$y = ax^2 + bx + c \tag{15}$$

$$F = G\frac{Mm}{r^2} \tag{16}$$

$$y = 4\pi\frac{\sin x}{\ln x^2} \tag{17}$$

$$y = \sum_{i=1}^{n} x^2 + 1 \tag{18}$$

现在引用(18)

$$i = \int_1^2 x^2 + \tan x \mathrm{d}x \tag{19}$$

$$A = \begin{bmatrix} 1 & 1 & 3 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{bmatrix} \tag{20}$$

$$A = \begin{cases} x^2 \\ x^2 + x \end{cases} \tag{21}$$

10.0.2 表格的插入

表 3: 标题

指标1	指标2	指标3
居左	居中	居右

表 4: 标题

数据	1	2
甲方	600	700
乙方	800	900