八股文背诵合集

The Sea and Sheng

2021 年 8 月 9 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 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.

1.0.3 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.

1.0.4 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

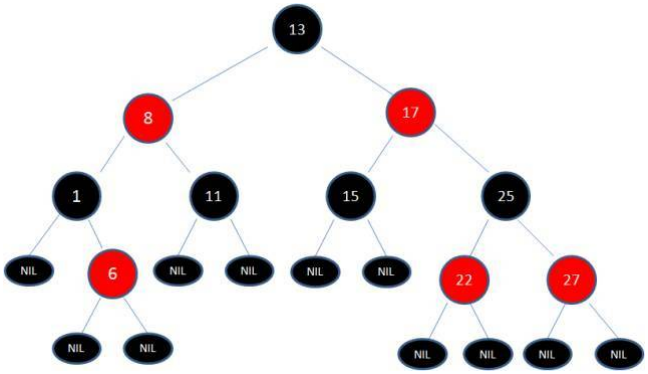

图 1:

1.0.5 第三个三级标题

2 java基础

2.0.1 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.

2.0.2 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
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.

2.0.3 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.

2.0.4 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.

2.0.5 Array 和 ArrayList 的区别

1. Array 大小固定, ArrayList 大小是动态变化的.

2.0.6 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.

2.0.7 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.

2.0.8 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. 接口里面所有的方法都是Public的, 抽象类允许Private, Protected方法
3. JDK接口可以实现默认方法和静态方法, 前面加defalut, static关键字.

```
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    }
```

2.0.9 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.

2.0.10 ==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.

2.0.11 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.

2.0.12 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.

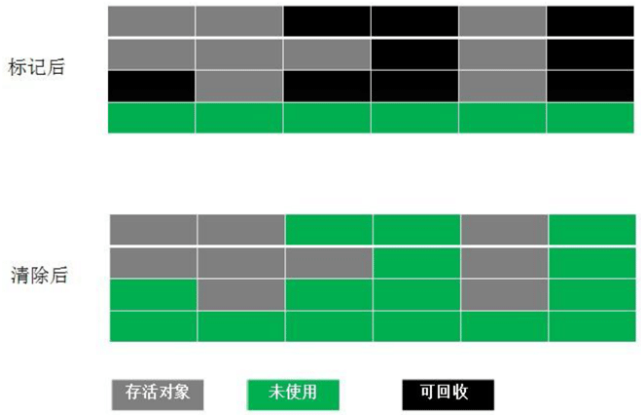

图 2:

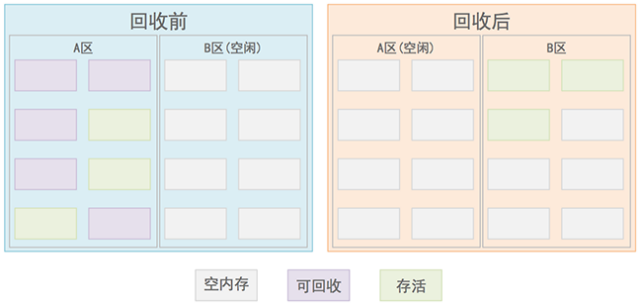

图 3:

3 JVM

3.0.1 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

3.0.2 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.

3.0.3 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.

3.0.4 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.
2. 元空间MetaSpace: 存在于本地内存.

3.0.5 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.

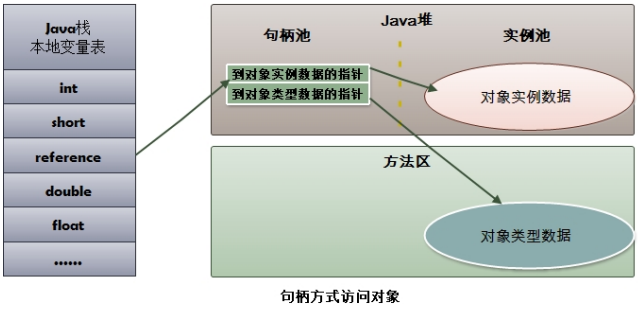

图 4:

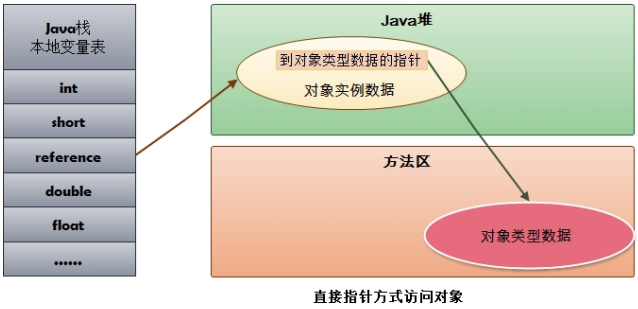

图 5:

3.0.6 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**

3.0.7 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.

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.

3.0.8 在java中, 对象什么时候可以被垃圾回收?

当对象边的不可触及的时候, 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.

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.

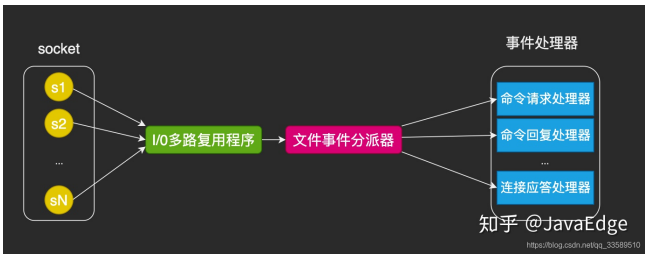

图 6:

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五中类型数据的实现方式

表 1: 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)

字符串结构SDS和C中char[]有什么不同

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

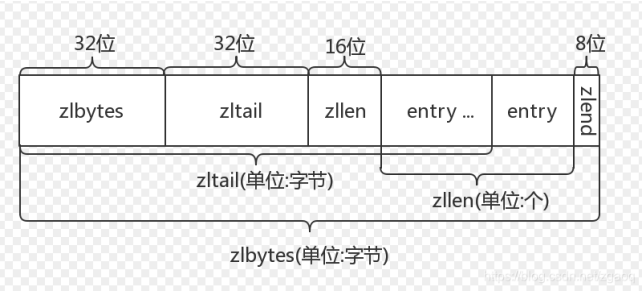

图 7:

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;

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.

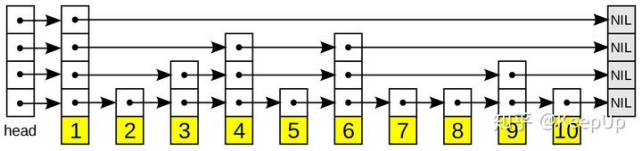

图 8:

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)
```

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,

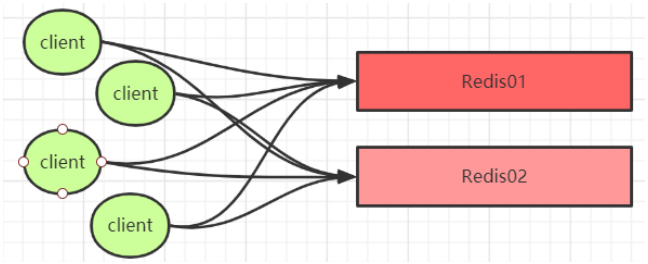

图 9:

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

6 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：

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

6.0.1 小练习

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

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

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

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

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

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

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

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

6.0.2 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：

6.0.3 第二个三级标题

6.0.4 第三个三级标题

7 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：

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

7.0.1 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)

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

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

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

7.0.2 表格的插入

表 2: 标题

指标1	指标2	指标3
居左	居中	居右

表 3: 标题

数据	1	2
甲方	600	700
乙方	800	900