# Java

## web应用中大量使用到缓存，以减少和数据库的交互和提高响应效率。

web应用中大量使用到缓存，以减少和数据库的交互和提高响应效率。

但是，缓存会遇到同时读和写的问题，在写的时候读就会遇到一些意外的情况。

大多数这种情况，就是一言不合就加同步锁synchronized。

一般来说是不会有什么问题的，但是在高并发的时候，可能导致大量线程在等待同步锁。

对于缓存这类读比写多的场景，java.util.concurrent的lock机制比同步锁更为实用。

ReentrantReadWriteLock ，它的读锁和写锁是互斥，写锁和另外的写锁肯定也是互斥的，

也能解决写的时候读的问题； 且它的读锁和其他读锁是可以共存的。

在高并发的情况下能大大减少线程同步锁的开销。

简单举例：

public void put(K key, V value) {  
        try {  
            writeLock.lock();  
            map.put(key, value);  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            writeLock.unlock();  
        }  
    }  
  
    public V get(K key) {  
        try {  
            readLock.lock();  
            return map.get(key);  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            readLock.unlock();  
        }  
        return null;  
    }

[评论 [0]](https://home.amarsoft.com/BLOG/down/entry/%E7%BC%93%E5%AD%98%E7%9A%84%E5%9D%91#comments)

Tags: [缓存](https://home.amarsoft.com/BLOG/down/tags/%E7%BC%93%E5%AD%98)

## 获取本地路径

Test.**class**.getResource("").getPath();

## 加密算法

URL编码和Base64编码都是编码算法，它们不是加密算法；

URL编码的目的是把任意文本数据编码为%前缀表示的文本，便于浏览器和服务器处理；

Base64编码的目的是把任意二进制数据编码为文本，但编码后数据量会增加1/3。

**哈希算法**

哈希算法最重要的特点就是：

* 相同的输入一定得到相同的输出；
* 不同的输入大概率得到不同的输出。

**哈希算法的用途**

1如何判断下载到本地的软件是原始的、未经篡改的文件？我们只需要自己计算一下本地文件的哈希值，再与官网公开的哈希值对比，如果相同，说明文件下载正确，否则，说明文件已被篡改。

2是存储用户口令,要考虑彩虹表攻击 解决方法 对每个口令额外添加随机数，这个方法称之为加盐（salt）加盐的目的在于使黑客的彩虹表失效，即使用户使用常用口令，也无法从MD5反推原始口令 HmacMD5算法，它相当于“加盐”的MD5 推荐使用

Java字符串的hashCode()就是一个哈希算法 这也是为什么当我们自定义一个class时，覆写equals()方法时我们必须正确覆写hashCode()方法 不然当做map的key 时 会存入相同的key或者取不出已经存在的结果

**对称加密算法**

使用同一个密钥进行加密和解密，常用算法有DES、AES和IDEA等；

**非对称加密算法**

**签名算法**

**数字证书**

摘要算法用来确保数据没有被篡改，非对称加密算法可以对数据进行加解密，签名算法可以确保数据完整性和抗否认性，把这些算法集合到一起，并搞一套完善的标准，这就是数字证书 HTTPS协议就是数字证书的应用

## 类初始化顺序

初始化父类中的静态属性 -> 父类中的静态代码块 -> 初始化子类中的静态属性 -> 子

类中的静态代码块 -> 父类中的普通属性初始化 -> 父类中的普通代码块 -> 父类中的

构造方法 -> 子类中的普通属性初始化-> 子类中的普通代码块 -> 子类中的构造方法。

## StringBuilder和StringBuilder

2. StringBuilder可变字符串，主要用于字符串的拼接，属于线程安全的；（StringBuffer的append操作用了synchronized）

3.StringBuilder可变字符串，主要用于字符串的拼接，属于线程不安全的；

String对象串联的效率最慢，单线程下字符串的串联用StringBuilder，多线程下字符串的串联用StrngBuffer；

## 数据库的数据结构

二叉树

红黑树 带平衡的二叉树

b—tree （理解： 改造的红黑树 ，把红黑树的头部 拉长 使树的高度不会太高，性能也得到巨大的优化）

b+tree mysql 的底层数据结构 改造的b—tree 索引加载到内存中比对 叶子节点存放data

mysql 数据都存储在磁盘

存储引擎

innoDB

## 数据结构

栈 后进后出网络浏览器多会将用户最近访问过的网址组织为一个栈。

场景L:

这样，用户每访问一个新页面，其地址就会被存放至栈顶；而用户每按下一次“后退”按钮，即可沿相反的次序访问此前刚访问过的页面。

类似地，主流的文本编辑器也大都支持编辑操作的历史记录功能（ctrl + z：撤销，ctrl + y：恢复），用户的编辑操作被依次记录在一个栈中。

一旦出现误操作，用户只需按下“撤销”按钮，即可取消最近一次操作并回到此前的编辑状态。

队列 先进先出

在需要公平且经济地对各种自然或社会资源做管理或分配的场合，无论是调度银行和医院的服务窗口，

还是管理轮耕的田地和轮伐的森林，队列都可大显身手。

甚至计算机及其网络自身内部的各种计算资源，无论是多进程共享的 CPU 时间，还是多用户共享的打印机，

数据结构：线性和非线性

数组，栈，队列，哈希表，树 Arraylist底层动态数组扩容

数组的优点

查找速度快

数组的缺点

插入和删除效率低

可能浪费内存

内存空间要求高，必须有足够的连续内存空间。

数组大小固定，不能动态拓展

链表的优点

插入删除速度快

内存利用率高，不会浪费内存

大小没有固定，拓展很灵活。

链表的缺点

不能随机查找，必须从第一个开始遍历，查找效率低

## Mysql索引

https://blog.csdn.net/weixin\_43591980/article/details/113833541?spm=1001.2014.3001.5501

## 乐观锁和悲观锁

悲观锁解决：

利用悲观锁的解决思路是，我们认为数据修改产生冲突的概率比较大，所以在更新之前，我们显示的对要修改的记录进行加锁，直到自己修改完再释放锁。

加锁期间只有自己可以进行读写，其他事务只能读不能写。

乐观锁解决

使用乐观锁的解决思路是，我们认为数据修改产生冲突的概率并不大，多个事务在修改数据的之前先查出版本号，在修改时把当前版本号作为修改条件，

只会有一个事务可以修改成功，其他事务则会失败。

悲观锁

优点：悲观锁利用数据库中的锁机制来实现数据变化的顺序执行，这是最有效的办法

缺点：一个事务用悲观锁对数据加锁之后，其他事务将不能对加锁的数据进行除了查询以外的所有操作，如果该事务执行时间很长，那么其他事务将一直等待，

那势必影响我们系统的吞吐量。

乐观锁

优点：乐观锁不在数据库上加锁，任何事务都可以对数据进行操作，在更新时才进行校验，这样就避免了悲观锁造成的吞吐量下降的劣势。

缺点：乐观锁因为是通过我们人为实现的，它仅仅适用于我们自己业务中，如果有外来事务插入，那么就可能发生错误。

应用场景

悲观锁：因为悲观锁会影响系统吞吐的性能，所以适合应用在写为居多的场景下。

乐观锁：因为乐观锁就是为了避免悲观锁的弊端出现的，所以适合应用在读为居多的场景下。

## 逻辑删除和物理删除

一：逻辑删除

逻辑删除的本质是修改操作，所谓的逻辑删除其实并不是真正的删除，而是在表中将对应的是否删除标识（is\_delete）或者说是状态字段（status）做修改操作。比如0是未删除，1是删除。在逻辑上数据是被删除的，但数据本身依然存在库中。

对应的SQL语句：update 表名 set is\_delete = 1 where id = 1；语句表示，在该表中将id为1的信息进行逻辑删除，那么客户端进行查询id为1的信息，服务器就不会提供信息。倘若想继续为客户端提供该信息，可将 is\_delete 更改为 0 。

二：物理删除

物理删除就是真正的从数据库中做删除操作了。

## 不同编码中文对应的字节

编码：GBK 字节数：2

编码：UTF-8 字节数：3

## BufferedReader的readline方法和read方法

Readline（）方法

1该方法读取一行文本，当遇到换行符\n,回车符\r或者\r\n回车换行，该行结束并返回。没有数据时，将会一直处于等待状态 等待一个换行/回车符而一直阻塞

。因此在进行网络连接时，应该避免使用该方法。

2、readLine()只有在数据流发生异常或者另一端被close()掉时，才会返回null值。

read()

功能：读取单个字符的个数，如果已经读完的话会返回-

　　byte[] bytes = new byte[1024];

int i = -1;

while ((i = bis.read(bytes)) != -1) {//单个读取计数，直到结束返回-1

baos.write(bytes, 0, i);

}

注意 read（）也是阻塞

解决方法 1 对方得关闭shutdownOutput

2 定长报文

3 为了防止read操作造成程序永久挂起，还可以给socket设置超时。如果read()方法在设置时间内没有读取到数据，就会抛出一个java.net.SocketTimeoutException异常。例如下面的方法设定超时3秒。socket.setSoTimeout(3000);

2 定长报文 读取写法

int readLen = 0; // 已成功读取的字节数

int dataLen = 0; // 报文前8个字节所标识的完整报文长度

ByteBuffer bbuf = new ByteBuffer();

StringBuffer sb = new StringBuffer();

try {

int needDataLen = 0;

byte[] buffer = new byte[8];

readLen = in.read(buffer, 0, 8);

logger.info("需要读取长度::::::::::::::::::::::::::" + readLen);

if(readLen < 8)

throw new Exception("获取字节数不足8位");

dataLen = Integer.parseInt(new String(buffer));//获取报文长度

needDataLen=dataLen;

logger.info("需要读取长度::::::::::::::::::::::::::" + needDataLen);

byte[] datas = new byte[needDataLen];

int x ;

while ((x = in.read(datas,0, needDataLen)) != -1)

{

bbuf.append(datas,0, x);

logger.info("当前读取长度:"+x);

needDataLen = needDataLen - x;

logger.info("剩余长度" + needDataLen);

if (needDataLen <= 0)

break;//读取完 跳出循环 防止阻塞

}

sb.append(new String(bbuf.toByteArray(), "GBK"));

} catch (Exception e) {

throw new Exception("读取失败");

}

## Bio nio aio 区别

在此之前，先弄清几个概念：

1.同步：使用同步IO时，Java自己处理IO读写。

2.异步：使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS，完成后OS通知Java处理（回调）。

3.阻塞：使用阻塞IO时，Java调用会一直阻塞到读写完成才返回。

4.非阻塞：使用非阻塞IO时，如果不能立马读写，Java调用会马上返回，当IO事件分发器通知可读写时在进行读写，不断循环直到读写完成。

下面是重点了（敲黑板！）！

1.BIO：同步并阻塞，服务器的实现模式是一个连接一个线程，这样的模式很明显的一个缺陷是：由于客户端连接数与服务器线程数成正比关系，可能造成不必要的线程开销，严重的还将导致服务器内存溢出。当然，这种情况可以通过线程池机制改善，但并不能从本质上消除这个弊端。

2.NIO：在JDK1.4以前，Java的IO模型一直是BIO，但从JDK1.4开始，JDK引入的新的IO模型NIO，它是同步非阻塞的。而服务器的实现模式是多个请求一个线程，即请求会注册到多路复用器Selector上，多路复用器轮询到连接有IO请求时才启动一个线程处理。

3.AIO：JDK1.7发布了NIO2.0，这就是真正意义上的异步非阻塞，服务器的实现模式为多个有效请求一个线程，客户端的IO请求都是由OS先完成再通知服务器应用去启动线程处理（回调）。

应用场景：并发连接数不多时采用BIO，因为它编程和调试都非常简单，但如果涉及到高并发的情况，应选择NIO或AIO，更好的建议是采用成熟的网络通信框架Netty。

## Socket和http区别

socket本质是编程接口（API），对TCP/IP的封装，TCP/IP也要提供可供程序员做网络开发所用的接口，这就是Socket编程接口；HTTP是轿车，提供了封装或者显示数据的具体形式；Socket是发动机，提供了网络通信的能力。

## 静态代理和动态代理

JDK动态代理 被代理类必须实现接口

CGLIB（Code Generator Library）动态代理---（需要导入第三方包）。被代理类不需要实现接口

## 第三次握手

我觉得一般不用说标志位，面试官主要想听为什么第三次握手，两次握手不行的原因：第三次握手保证了客户端和服务端既能给对方发送消息，又能响应对方

## ArrayList的新API加上lambda表达式

从Java 8开始，使用ArrayList的新API加上lambda表达式，我们可以这样写：

list.forEach(e -> System.out.println(e));

而这里的lambda表达式的内容其实只不过就是把参数传给了println()方法，而没有做任何别的事情，所以可以进一步简写为：

list.forEach(System.out::println);

System.out::println 可以看作 lambda表达式 e -> System.out.println(e) 的缩写形式。

## 并发的三大特性

原子性，可见性，有序性

例子 i++线程不安全

## maven中的三种工程：

1.pom工程：用在父级工程或聚合工程中。用来做jar包的版本控制。在pom文件中加<packaging>pom</packaging>

2.war工程：将会打包成war，发布在服务器上的工程。如网站或服务。

3.jar工程：将会打包成jar用作jar包使用。

## ##架构演进

架构的发展历程是从单体式架构，到分布式架构，到SOA架构，再到微服务架构。

1单体架构：未做任何拆分的Java Web程序

2分布式架构:按照业务垂直划分，每个业务都是单体架构，通过API互相调用

3 SOA架构：SOA是一种面向服务的架构。其应用程序的不同组件通过网络上的通信协议向其它组件提供服务或消费服务，所以也是分布式架构的一种。

4微服务架构在某种程度上是SOA架构的进一步的发展。

微服务目前并没有比较官方的定义

## ##微服务解决方案

目前最流行的两种微服务解决方案是Spring Cloud和Dubbo。

链接: [、详细来源](https://mp.weixin.qq.com/s/gktF-VYvXdpLsr74oi1msg)

## 构造方法

一个类没有定义构造方法（无参），编译器会自动为我们生成一个默认构造方法

**class Person {**

**public** Person() {

}

}

如果我们自定义了一个构造方法，那么，编译器就不再自动创建默认构造方法

## Kill tomcat 进程

输入netstat -ano|findstr 8080    8080为tomcat端口号

输入taskkill /F /PID 13756

不行 需用管理员cmd

## 多线程用while代替if

个被唤醒的线程就处于就绪状态了，就可以等待被cpu调度了，但是在if语句中，线程被唤醒就从被唤醒的地方执行，不会再次判断，而在while循环中，从被唤醒的地方开始执行代码，至始至终都在while循环中，会继续判断，不满足情况就跳出while，满足就继续wait

## 在全局中定义一个static  数组进行数据缓存

private static List<JSONObject> listBidInfo = new ArrayList<>();

通过这样定义之后，每次访问数据库之后将数据先存储在数组中，从而减少对数据库的访问次数，充当了一定的缓存作用。如果数据过大建议还是使用redis进行缓存。

## 数据库删除

delete是一行一行删除 truncate是将表结构销毁，在重新创建表结构.

如果数据比较多，truncate的性能高。

delete是受事务控制. 可以回滚数据.

truncate是不受事务控制. 不能回滚.

Drop table