## *框架*

### Mybatis

1. *Mybatis批处理：*

**

2. mybatis 使用collection association 关联查询，可以传递多个参数,需要在resultmap中声明（即使是数据库没有的字段，也必须声明才能传递到子查询）

3. mybatis 关于集合的使用，可以是map，list array 但是不能是set集合。

### Spring

1. spring提供文件操作：FileCopyUtil.copy(SerializationUtils.serialie(T),new File(“”))；

2. spring-junit 单元测试：要想使用spring容器的ioc，使用两个注解

@RunWith(SpringJUnit4ClassRunner.class)

@ContextSonfigration(“classpath:aaplicationContext.xml”)

3. Web容器初始化过程

Listener===🡺fileter==🡺servlet

在 Servlet API中有一个ServletContextListener接口，它能够监听ServletContext对象的生命周期，实际上就是监听Web应用的生命周期。当Servlet容器启动或终止Web应用时，会触发ServletContextEvent事件，该事件由ServletContextListener来处理。在ServletContextListener接口中定义了处理ServletContextEvent 事件的两个方法contextInitialized()和contextDestroyed()。

*4. SpringAOp注解：*

*直接调用的service在类上面加上注解就行。Spring会将当前代理对象绑定到当前线程上面，所以service之间的调用并没有用到代理，*

*Service之间调用的实现aop功能，需要将露当前代理对象给当前线程就行了*

*<!-- aspect -->*

*<aop:aspectj-autoproxy proxy-target-class="true" expose-proxy="true"/>*

*protected final T proxy() {*

*return (T) AopContext.currentProxy();*

*}*

*5. 非web项目可以使用spring作为对象管理工具，可以使用aop以及ioc*

## 安全

1. 前端安全问题：

前端安全问题主要存在于cookie机制(自动存储，自动携带)。

XSS攻击：防御方式：配置拦截器，将前端传过来的内容包含特殊字符的屏蔽过滤。

CSRF攻击：防御方式，可以采用tocken机制，服务器返回给用户一个cookie和一个随机字符串(保存在服务器端)，每次验证cookie和随机的字符串。

1. Cookie要设置成httpOnly
2. Shiro安全框架

作用 ：认证、授权、session 管理（非web项目也可以有）、加密

1. cookie session token

cokie session 是解决http无状态的，token更多的扮演安全认证的角色。

**Token（jwt）工作方式：用户第一次登陆，服务端生成token，返回给客户端，客户端保存，以后发送携带token参数，通过比较参数判断客户端的登陆状态。(一致=已登陆，没有=没有登陆 ，不一致=登陆过期)**

## 数据库

##### 数据库执行sql过程：

mysql的连接是基于TCP协议的，而且该协议是二进制协议，不是类似于HTTP的文本协议，其中建立连接的过程具体如下：

第1步：建立TCP连接，通过三次握手实现；（断开4次挥手实现）

第2步：服务器发送给客户端握手信息，客户端响应该握手消息；

第3步：客户端发送认证包，用于用户验证，验证成功后，服务器返回OK响应，之后开始执行命令；

用户验证成功之后，会进行一些连接变量的设置，比如字符集、是否自动提交事务等，其间会有多次数据的交互。完成了这些步骤后，才会执行真正的数据查询和更新等操作。

##### Sql索引

Mysql索引常用的Hash索引和BTree索引不同的引擎对于索引有不同的支持：Innodb和MyISAM默认的索引是Btree索引；而Mermory默认的索引是Hash索引。

所谓Hash索引，当我们要给某张表某列增加索引时，将这张表的这一列进行哈希算法计算，得到哈希值，排序在哈希数组上。所以Hash索引可以一次定位，其效率很高，而Btree索引需要经过多次的磁盘IO，但是innodb和myisam之所以没有采用它，是因为它存在着好多缺点：

1、因为Hash索引比较的是经过Hash计算的值，所以只能进行等式比较，不能用于范围查询

1、每次都要全表扫描

2、由于哈希值是按照顺序排列的，但是哈希值映射的真正数据在哈希表中就不一定按照顺序排列，所以无法利用Hash索引来加速任何排序操作

3、不能用部分索引键来搜索，因为组合索引在计算哈希值的时候是一起计算的。

4、当哈希值大量重复且数据量非常大时，其检索效率并没有Btree索引高的。

Sql优化：

1. 负向查询不能使用索引
2. 前导模糊查询不能使用索引
3. 数据区分不明显的不建议创建索引
4. 字段的默认值不要为 null
5. 在字段上进行计算不能命中索引
6. 最左前缀问题
7. 如果明确知道只有一条记录返回
8. 不要让数据库帮我们做强制类型转换
9. 如果需要进行 join 的字段两表的字段类型要相同

##### 数据库引擎

## Hadoop

### zookeeper

1. zookeeper 特点： 及时性、顺序一致性、原子性、唯一系统映像、可靠性
2. 使用场景：统一配置中心、分布式锁、

### Hdfs

1. 后台进程：

<hdfs>

NameNode： 本地存储文件的元数据：文件名、文件目录结构、文件属性（权限、副本数、生成时间）、文件块列表及文件块位置

DataNode：在本地文件系统存储文件块信息

Secondary NameNode :用来监控HDFS状态的辅助后台程序，每隔一段时间获取NN的元数据快照

<mapReduce>

JobTracker 负责接收用户提交的作业，负责启动、追踪任务执行

TaskTracker 负责执行有JobTracker分配的任务，负责各个任务在每个节点上的执行情况。通过心跳机制与JobTracker联系，TaskTracker包括map task、reduce task

文件写入hdfs步骤：

1. hdfs是基于文件系统的，所以新建文件以及文件夹都是基于fileSystem的。
2. 获取文件系统分两步：读取配置文件Configuration conf=new Configuration()，

获取文件系统 FileSystem fs=FileSystem.get(conf)

（默认文件系统是单例的，多线程情况下会出现问题，需要配置以下）

1. 创建文件保存路径，创建文件。（通过fileSystem Path）

Path path=new Path(<String>filePath);

Fs.mkdirs( path );

1. 创建hdfs输出流 FSDataOutputStream out=fs.creat(fileName);

写出数据 out.write(byte[ ]);

1. 关闭文件系统 以及输出流

### hive

Hive适合用来对一段时间内的数据进行分析查询，例如，用来计算趋势或者网站的日志。Hive不应该用来进行实时的查询。因为它需要很长时间才可以返回结果。

### kafka

kafka 是将所有消息保存在硬盘上，使用操作系统的Sequence I/O:具有很高的I/O性能。

### flume

### hbase

hbase是一个列数据库，

Hbase非常适合用来进行大数据的实时查询。Facebook用Hbase进行消息和实时的分析。它也可以用来统计Facebook的连接数。

Hive和Hbase是两种基于Hadoop的不同技术–Hive是一种类SQL的引擎，并且运行MapReduce任务，Hbase是一种在Hadoop之上的NoSQL 的Key/vale数据库。当然，这两种工具是可以同时使用的。就像用Google来搜索，用FaceBook进行社交一样，Hive可以用来进行统计查询，HBase可以用来进行实时查询，数据也可以从Hive写到Hbase，设置再从Hbase写回Hive。

## 缓存

高并发并发系统架构口诀：读多写少用缓存，写多读少用队列。

1. 缓存存在问题：缓存击穿、缓存失效、缓存并发

缓存击穿/缓存并发：可以设置双缓存(备份缓存)+锁机制 < jvm同步锁，分布式锁>+双重检验

缓存失效：失效时间在统一基础上加一个随机值，防止同一时刻大范围失效

或者统一配置失效时间，是失效时间均匀分布失效

### Redis

Redis 数据类型： String 、List、Set(去重)、sorted Set（排序去重，加权）、hash（map结构）

1. Redis主从中从服务器为只读，同步模式：首次全量同步 以后，增量同步
2. Redis主从复制机制：redis的主从复制是以文件形式进行同步的。redis主从同步期间，master是非阻塞的，slave是阻塞的，不可以进行查询，可以禁止master的文件备份，交给slave去备份。
3. Redis集群是将一种key的hash散布，有固定大小的hash槽，key的hash依照算法存入对应的hash槽对应的redis服务中。
4. 缓存失效：

大面积失效：为有效期增加随机值，避免同时失效

统一规划失效时间，失效时间均匀分布

热点key失效：加锁(互斥锁)<解决不了分布式问题>

分布式锁(双缓存)：基于redis实现的分布式锁(不能再急集群中实现，可用redLock)[ 双缓存，取锁失败，则从备份缓存中取]

基于zookeeper实现的分布式锁

## Web服务器

## Nginx

1. 正向代理与反向代理
   1. 正向代理指的是代理浏览器端，实现请求转发，不掩盖实际url
   2. 反向代理是指代理服务器，实现请求分发，掩藏了实际的请求服务器，用于负载均衡。
2. 负载均衡
   1. 负载均衡的实现必须依赖于反向代理，Nginx的负载均衡策略有两种：内置策略和扩展策略。
3. Ngnix可以添加一些插件，实现更加细腻的负载均衡方案，如iphash，url，请求参数
4. nigix可靠：默认会启动两个守护进程+主进程 两个进程
5. ngnix单点故障解决方法：ngnix集群，使用Lvs虚拟ip（是ngnix集群使用一个虚拟ip，并可以检测ngnix状态，动态分配请求。）这是一种在硬件层面的负载均衡。
6. 负载均衡策略:

DNS(按照一定规则请求不同的服务器，是一种客户端层面的负载均衡)===》LVS虚拟IP实现的硬件层次的负载均衡===》Ngnix实现的软件层面的均衡。

1. Ngnix可以实现服务器的动静分离。前后端单独部署，但是会存在跨域问题，解决方法：跨域注解;设置跨域拦截器。

## Tomcat

1. 单个tomcat 并发能力最多到2W ; nigix占有资源少，并发能力可以达到10W

2. tomcat优化：io模型，连接池 内存参数 垃圾回收参数

3. Jetty当以form发送请求时候，默认最大内容为20000，可以通过设置提高

System.();

## 分布式

1. 分布式理论：

CAP定理：**任何系统都无法满足一致性**(Consistency)**、可用性**(Availability) **、分区容错性**(Partition tolerance) ，一个系统至多支持以上两个属性，但是容错性必须保证的，所以会有AP系统、CP系统。

Base理论：BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：**我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性**（Eventual consistency）。

1. 分布式注意：各个进程之间的通信，协作问题，命名问题。
2. 分布式锁

分布式锁需要满足的条件：

* 在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；
* 高可用的获取锁与释放锁；
* 高性能的获取锁与释放锁；
* 具备可重入特性；
* 具备锁失效机制，防止死锁；
* 具备阻塞锁特性，即没有获取到锁将继续等待获取锁；
* 具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

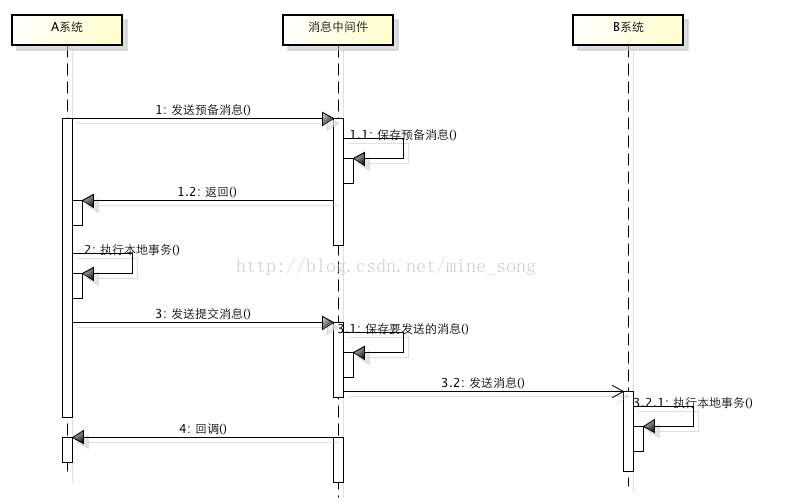
|  |  |  |
| --- | --- | --- |
| 基于数据库 | 基于一条数据的插入 |  |
| 基于数据库锁（for update） |
| 基于redis | 使用sefnx(set if not exist) | Redis集群会存在单节点宕机，瞬间锁失效 |
| 使用relock |  |
| Zookeeper | 创建临时文件 |  |

1. 分布式事务

|  |  |  |
| --- | --- | --- |
| 消息中间件 |  |  |
| XA事务 |  |  |
| TCC |  |  |

消息中间件：（ RocketMQ支持，但是kafka、RabbitMQ 都不支持）

原理：

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务

1、A系统向消息中间件发送一条预备消息  
2、消息中间件保存预备消息并返回成功  
3、A执行本地事务  
4、A发送提交消息给消息中间件

TX协议

TCC，基于

## 协议

1. Soap（Simple Object Access Protocle）简单对象访问协议，soap=http+xml
2. RPC（remote procedure call） 远程过程调用
3. 错误码

400-499：客户端请求错误返回状态码。主要体现在404（资源不存在）、405（方法不允许）、401（访问该资源未被授权）、400（坏请求）等。

500-599：服务端错误导致的状态码，一般很少出现。主要是存在于代理网关上，如500（连不上服务器）、501（请求的方法无法解析）、502（网关连不上）、503（服务器上当前资源暂时不可用）、504（网关超时）等

## 设计模式

设计模式： 开闭原则 ( 代码重用、低耦合、高内聚、易扩展、易维护)

1. 备忘录模式
2. 策略模式：

策略模式：这是一个很重要的设计原则，“**多用组合，少用继承**”。

其实，从更加抽象的角度来讲，例子中的战斗方式实际上是一种算法，通过策略模式，我们分离了使用算法的角色和算法之间的联系。因此我们可以给出策略模式的定义：

***策略模式定义了算法族，分别封装起来，让他们之间可以相互替换，此模式让算法的变化独立于使用算法的客户。***

**策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给别的对象管理。**客户端决定应该使用什么策略，因此客户端需要理解所有具体策略类之间的区别，这也增加了客户端的使用难度。

基于事件的设计：附加于主要任务之上的业务，不能因此影响主要任务的状态。

事件的四大要素：事件：时间发起者；时间执行者；事件监听者

常见的体现：JMS 、观察者模式 、Spring提出对基于事件的设计支持（注解）

## 定时调度(quartz)

1. quartz 线程

## webservice

1. webservice ====soap ====基于xml的=====访问暴露方法。

## linux命令

1．ps命令：ps是显示瞬间进程的状态

2. nohup 命令：不挂断地运行命令。语法：nohup Command [ Arg … ] [　& ]

3. java –cp 相当于 java –classPath 是指定类运行所依赖其他类的路径，通常是类库，jar包之类，需要全路径到jar包，window上分号“;” 分隔，linux上是分号“:”分隔。不支持通配符，需要列出所有jar包，用一点“.”代表当前路径。

语法 java –cp jar2:jar3:jar:4 mainClass >/dev/null 2>&1 (后面表示log重定向)

4. |命令 连接两个进程，将左边的输出结果输入到右边

5. grep 命令 文本搜索工具

6. > file 重置文件

7. sed 流式文本编辑器，语法： sed ‘/ XXXX/‘ file

8. du –sh 显示磁盘信息

9. top 显示linux 各种资源占用情况 –p pid 显示指定进程信息

10. scp命令 跨服务器 进行文件传输

11． whereis redis/ngnix 查询---安装位置

12 批量杀死进程

ps -ef | grep test | grep -v grep | awk '{print $2}' | xargs kill -9   
其中：   
|管道符，用来隔开两个命令，管道符左边命令的输出会作为管道符右边命令的输入。   
ps命令用来列出系统中当前运行的进程， ps -ef显示所有进程信息，联通命令行。   
grep命令用于过滤/搜索特定字符，grep test在这里为搜索过滤所有含有‘test’名称的进程   
grep -v grep -v 显示不包含匹配文本的所有行，在这里为筛选出所有不包含grep名称的进程，对上一步的进程再做一次筛选(因为ps -ef列出了所有的命令，包括命令行)   
awk在文件或字符串中基于指定规则浏览和抽取信息；把文件逐行读入，以空格为默认分隔符将每行切片，然后再进行后序处理。这里利用awk '{print $2}'将上一步中过滤得到的进程进行打印，$2表示打印第二个域(PID，进程号) $0表示所有域,$1表示第一个域，$n表示第n个域。   
xargs命令是给命令传递参数的过滤器，善于把标准数据数据转换成命令行参数。在这里则是将获取前一个命令的标准输出然后转换成命令行参数传递给后面的kill命令。   
kill -9强制关闭进程。   
此外，也有使用cut命令进行处理的，参考如下：   
ps -ef | grep test | grep -v grep | cut -c 9-15 | xargs kill -9   
cut -c 9-15仅显示第9-15个字符(即PID，进程号)

## JVM 和 类加载

### Jvm优化

* + 1. jvm分为堆(对象级别的，存放所有的对象)、

栈、(每个线程执行每个方法的时候都会在栈中申请一个栈帧，每个栈帧包括局部变量区和操作数栈，用于存放此次方法调用过程中的临时变量、参数和中间结果。)

本地栈、（用于支持native方法的执行，存储了每个native方法调用的状态）、

方法区(属于类级别的<静态变量>，存放类对象，常量池<final 类型的常量>)、

程序计数器。

JVM优化内容：调整JVM堆、非堆的大小，以及堆中各区域的大小，调整个位置的垃圾回收机制

1. 所有通过new创建的对象的内存都在堆中分配， Java 中的堆也是 GC 收集垃圾的主要区域，其大小可以通过-Xmx和-Xms来控制。
2. 堆被划分为新生代和旧生,新生代又被进一步划分为Eden和Survivor区，最后Survivor由FromSpace和ToSpace组成, 新建的对象都是用新生代分配内存，Eden空间不足的时候，会把存活的对象转移到Survivor中, JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块Survivor区域是空闲着的。新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例旧生代。用于存放新生代中经过多次垃圾回收仍然存活的对象。
3. JVM垃圾回收GC JVM分别对新生代和旧生代采用不同的垃圾回收机制，GC 分为两种：Minor GC、FullGC ( 或称为 Major GC )，在执行方面分为：

串行收集器（**-XX:+UseSerialGC**）、并行收集器（**-XX:+UseParallelGC**）、并发收集器

* 1. 新生代通常存活时间较短，因此基于复制算法来进行回收，所谓复制算法就是扫描出存活的对象，并复制到一块新的完全未使用的空间中，对应于新生代，就是在Eden和其中一个Survivor，复制到另一个之间Survivor空间中，然后清理掉原来就是在Eden和其中一个Survivor中的对象。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在新生代区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。当连续分配对象时，对象会逐渐从eden到 survivor，最后到老年代，

 Minor GC 是**发生在新生代中**的垃圾收集动作，**所采用的是复制算法**。

* 1. 旧生代与新生代不同，对象存活的时间比较长，比较稳定，因此采用标记（Mark）算法来进行回收，所谓标记就是扫描出存活的对象，然后再进行回收未被标记的对象，回收后对用空出的空间要么进行合并，要么标记出来便于下次进行分配，总之就是要减少内存碎片带来的效率损耗。

Full GC 是**发生在老年代**的垃圾收集动作，标记-清除算法收集垃圾的时候会产生许多的内存碎片 ( 即不连续的内存空间 )shi-XX:+UseSerialGC

1. Jvm优化需要了解jvm的结构，java的内存模型

优化工具: jstat jmap

|  |  |
| --- | --- |
| -Xms | 初始堆大小。如：-Xms256m |
| -Xmx | 最大堆大小。如：-Xmx512m |
| -Xmn | 新生代大小。通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor，即 90% |
| -Xss | JDK1.5+ 每个线程堆栈大小为 1M，一般来说如果栈不是很深的话， 1M 是绝对够用了的。 |
| -XX:NewRatio | 新生代与老年代的比例，如 –XX:NewRatio=2，则新生代占整个堆空间的1/3，老年代占2/3 |
| -XX:SurvivorRatio | 新生代中 Eden 与 Survivor 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10 |
| -XX:PermSize | 永久代(方法区)的初始大小 |
| -XX:MaxPermSize | 永久代(方法区)的最大值 |
| -XX:+PrintGCDetails | GC 信息 |
| -XX:+HeapDumpOnOutOfMemoryError | 让虚拟机在发生内存溢出时 Dump 出当前的内存堆转储快照，以便分析用 |

### 类加载

类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。

类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）

* Bootstrap（核心类库加载器）：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；c++实现的，不能找到次加载器。
* Extension（扩展类库加载器）：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；
* System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。
* AppClassloader（应用程序类加载器）：我们的应用程序类加载器。

类的加载方式：全盘负责、父类委托、缓存机制 双亲委派模型的工作流程是：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，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类

双亲委派机制的优点：

1. 安全：，将子类用父类的类加载器加载，程序启动时候会将rt加载到内存，所以当一个类起名与jdk一致，则不能加载。每个类加载器只能加载指定范围之内的类文件
2. 好处：避免类的重复加载。

JVM是根据类加载器与类名来判别类是否加载的，所以要实现热部署，可以销毁原有的类加载器中的实例，然后创建新的类加载器加载。

 热部署步骤：

1、销毁自定义classloader(被该加载器加载的class也会自动卸载)；

2、更新class

3、使用新的ClassLoader去加载class

JVM中的Class只有满足以下三个条件，才能被GC回收，也就是该Class被卸载（unload）：

   - 该类所有的实例都已经被GC，也就是JVM中不存在该Class的任何实例。  
   - 加载该类的ClassLoader已经被GC。  
   - 该类的java.lang.Class 对象没有在任何地方被引用，如不能在任何地方通过反射访问该类的方法

程序中可以设置接下来的类加载器：Thread.currentThread.setContextClassLoad();

自定义类加载器，可以使用URLClassLoad（）；通过监控文件可以实现热部署。

### NIO

1. 通常，我们可以将 IO 分为两类：面向数据块（block-oriented）的 IO 和面向流（stream-oriented）的 IO。比如文件的读写就是面向数据块的，读取键盘输入或往网络中写入数据就是面向流的。
2. 使用内存映射实现文件复制操作(Memory mapping-内存映射方式)

File f = new File("E:"+File.separator+"test.txt");

FileInputStream in = new FileInputStream(f);

FileOutputStream out = new FileOutputStream("F:"+File.separator+"test2.txt");

FileChannel inChan = in.getChannel();

FileChannel outChan = out.getChannel();

ByteBuffer buf = ByteBuffer.allocate(1024);//开辟缓冲区

while ((inChan.read(buf)) != -1){

buf.flip();//重设缓冲区

outChan.write(buf);//输出缓冲区

buf.clear();//清空缓冲区

}

inChan.close();

outChan.close();

in.close();

out.close();

1. NIO知识：

新的IO操作不再面向 Stream来进行操作了，改为了通道Channel，并且使用了更加灵活的缓存区类Buffer，Buffer只是缓存区定义接口， 根据需要，我们可以选择对应类型的缓存区实现类。在java NIO编程中，我们需要理解以下3个对象Channel、Buffer和Selector。

Channel知识：

Channel即通道，与BIO中的流一个级别。流是单向的，Channel是双向的，既可以用来进行读操作，又可以用来进行写操作。

Channel的话分为四种： FileChannel从文件中读写数据。DatagramChannel以UDP的形式从网络中读写数据。SocketChannel以TCP的形式从网络中读写数据。ServerSocketChannel允许你监听TCP连接。

Buffer知识：

数据是从通道（channel）中写入到缓冲区(buffer)，或从缓冲区写入到通道。缓冲区本质上来说是一个能够进行数据读写的内存块。Buffer对象提供了一系列方法，使得它能够很容易地操控这个内存块。

Buffer重要属性（capacity、position、limit）读写模式下会自动进行切换。

Buffer有（ByteBuffer、MappedBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer、ShortBuffer）几种类型，也就是以不同的数据类型操作缓冲区

Buffer的常用方法：

|  |  |
| --- | --- |
| allocate(int n) | 在内存中分配buffer |
| flip() | 读写转换，三个属性值变换， |
| channel.write()/buffer.get() | 写数据（从缓存往外拿） |
| channel.read()/buffer.put() | 读数据（往缓存中写） |
| clear()/ compact() | 清空缓存/清除已经读取的缓存 |
| equals()/ | 比较两个缓存及里面剩余数据 |

1. Selector知识(注册器)

在传统的架构中，对于客户端的每一次请求，服务器都会创建一个新的线程或者利用线程池复用去处理用户的一个请求，然后返回给用户结果，这样做在高并发的情况下会存在非常严重的性能问题：对于用户的每一次请求都创建一个新的线程是需要一定内存的，同时线程之间频繁的上下文切换也是一个很大的开销。

Selector 是NIO相对于BIO实现多路复用的基础，Selector 运行单线程处理多个 Channel，如果你的应用打开了多个通道，但每个连接的流量都很低，使用 Selector 就会很方便。例如在一个聊天服务器中。要使用 Selector , 得向 Selector 注册 Channel，然后调用它的 select() 方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新的连接进来、数据接收等。

通过使用selector，我们可以通过一个线程来同时管理多个channel，省去了创建线程以及线程之间进行上下文切换的开销。

现在对于客户端的每一次请求到来时我们不再立即创建一个线程进行处理，相反以epool为例子当一个事件准备就绪之后通过回调机制将描述符加入到阻塞队列中，下面只需要通过遍历阻塞队列对相应的事件进行处理就行了，通过这种回调机制整个过程都不需要对于每一个请求都去创建一个线程去单独处理。

1. socketChannel read方法的返回值情况如下：

n 有数据的时候返回读取到的字节数。

0 没有数据并且没有达到流的末端时返回0。

-1 当达到流末端的时候返回-1。

2. Selector会占用一个线程在注册到selector时，可以附加对象，可供后面就绪线程使用

### Java基础

1. 原注解：
   1. target：用于标注该注解添加的位置(class、field、method)
   2. intention:用于表示该注解的元素的作用域
   3. document：用于将该注解打包成文档

2. 数据类型：byte short int long float double char boolean

### Java多线程

1. java内存模型 JMM（java Memory mo）

java内存模型规定所有的共享变量都存储在主内存中，而每条线程有自己的工作内存(本地内存)，工作内存保存了共享变量的副本，而不同内存又无法访问对方的工作内存，所以如果线程在工作内存中修改了变量副本，其它线程是无从得知的。

Java内存模型规范：

* 1. 线程只可操作自己的内存
  2. 共享变量必须放在主内存，
  3. 线程操作共享变量，需从主内存，读取到工作内存，改变值后需同步到主内存。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。

1. 线程之间的通信机制有两种共享内存和消息传递。

在**共享内存**的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过**共享对象**进行通信。

在**消息传递**的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是**wait()**和**notify()**。

1. JVM中运行的每个线程都拥有自己的线程栈，线程栈包含了当前线程执行的方法调用相关信息，我们也把它称作调用栈。随着代码的不断执行，调用栈会不断变化。
2. 支撑java内存模型的原理：指令重排序：简单来说，没有相互依赖关系的指令，执行顺序可能会发生变化，但是Jvm可是使用Memory Barrier来禁止特定类型的编译器重排序和处理器重排序
3. volatile、synchronized原理是基于CPU内存屏障指令实现的

volatile和synchronized的区别

1. volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的
3. volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性
4. volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。
5. volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化
6. 任务性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务配置尽可能少的线程数量，如配置Ncpu+1个线程的线程池。IO密集型任务则由于需要等待IO操作，线程并不是一直在执行任务，则配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。
7. 线程中断机制：Thread. Interrupt(); 但是该方法不能直接中断线程，相当于设置了线程的状态，在非阻塞线程中不起作用，但是可以用Thread.currentThread. isInterrupted() 判断线程状态，然后退出线程，在阻塞线程中，会出现异常，在捕获异常时候退出异常。
8. 2. AbstractQueuedSynchronizer简称AQS，它为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器（信号量等）提供一个基础实现框架。
9. 多线程容易引发的问题：竞争条件、内存一致性（The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement.）

### JMX（Java Management Extensions，即Java管理扩展）

1. 简介：可以方便的管理、监控正在运行中的Java程序。常用于管理线程，内存，日志Level，服务重启，系统环境等。】、
2. 架构：JMX的架构是组件式的，被设计为三层：

* **分布层（Distributed layer）**：包含可以使管理应用与JMX Agents交互的组件。一旦通过交互组件与JMX Agents建立连接，用户可以用管理工具来和注册在Agents中的MBeans进行交互；
* **代理层（Agent layer ）**：包含JMX Agent以及它们包含的MBean Servers。Agent layer的主要组件是MBean server，作为JMX Agents的核心，它充当MBeans的注册中心。该层提供了4个Agent 服务来使对MBean的管理更容易：计时器（Timer）、监控（monitoring）、动态加载MBean（dynamic MBean loading ）、关系服务（relationship services ）；
* **指示层（Instrumentation layer ）**：包含代表可管理资源的MBeans。该层是最接近管理资源的，它由注册在Agents中的MBeans组成，这个MBean允许通过JMX Agent来管理。每个MBean都暴露出来针对底层资源的操作和访问

1. JMX提供了三种访问控制方法：html、java client、 jconsole

### 集合

Java 9，添加了几种集合工厂方法：

List<String> myList = List.of("张三","李四","王五"); 不能再添加数据，适合只读操作。

1. concurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash是为了定位Segment(段)，第二次Hash为了定位元素所在的链表的头部。因此，这一个结构所带来的后果是，Hash的过程会比HashMap要长，但进行写的操作的时候只要对具体的某个segment进行加锁即可（注意：读的时候不需要加锁），这样也不会影响到其他的segment。
2. java 提供了blockingQuerer的消息队列：

### LOCK

**锁的相关概念：可重入锁、可中断锁、公平锁、读写锁。**

**重入锁**：指的是以线程为单位，当一个线程获取对象锁之后，这个线程可以再次获取本对象上的锁，而其他的线程是不可以的。synchronized 和   ReentrantLock 都是可重入锁所谓重入锁，指的是以线程为单位，当一个线程获取对象锁之后，这个线程可以再次获取本对象上的锁，而其他的线程是不可以的，（可重入锁的意义在于防止死锁），

**公平锁：**指获取锁按照先来后到的原则。非公平锁是抢占式

**可中断锁：**线程在等待获取锁期间是否可被中断，synchronized、普通lock是不可中断的，lockInterruptibly可中断。

Lock需要手动的去释放锁。

|  |  |  |
| --- | --- | --- |
| lock() | 如果锁已被其他线程获取，则进行等待。 | 阻塞，不响应线程中断 |
| tryLock() | 如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false | 立刻返回 |
| tryLock(long time, TimeUnit unit) | 同上 | 阻塞指定的时间 |
| lockInterruptibly() | 在等待时，其他线程可以调用Thread.interrupt方法来中断等待线程的等待而直接返回，这时不用获取锁，而会抛出一个InterruptedException。由上层调用者区去处理中断。 | 阻塞，阻塞期间可以相应中断 |

1. 特殊的处理场景，例如读写锁，同时可以多个读获取锁，但是只能有一个写锁。（readwriteLock）

### 泛型

Java泛型支持通配符(Wildcard)，可以单独使用一个“?”表示任意类，也可以使用extends关键字表示某一个类(接口)的子类型，还可以使用super关键字表示某一个类(接口)的父类型**.**

(1)、泛型结构只参与 “读” 操作则限定上界(extends关键字)

(2)、泛型结构只参与“写” 操作则限定下界(使用super关键字)

### Java8新特性

1. Lamabda： 函数式编程：简化代码

用法：Runnable runnable=()->System.print.out();

Lamabda 操作符->,左侧是参数列表，右侧是要执行的功能

1. String： 提供并行计算

注意：流只能读取一次；任何对流的操作都是创建一个新流；流不能保存数据；

对流的操作：nums.stream().filter(num->num>5).count();

创建流 转换流 聚合

## javaSE

### Session一致性：使用场景（sso，分布式应用）

实现方案：

1. 客户端存储session（即浏览器cookie保存），适用与内网环境等安全场所

2. ip\_hash :依赖nginx的负载均衡策略，指定ip的访问服务器。(较大型分布式场景)

优点：1.实现简单，2.hash算法可以实现不同ip的均匀均衡 3.服务器水平扩展方便，4.安全性高

缺点：1.水平扩展会有ip重hash分配到不同的服务器，导致部分session失效

2. 服务器重启会造成session丢失 3.存在单节点负载高的风险

3. session复制：(小型分布下首选)

缺点：1.占用带宽 2. 有一定的同步延迟 3. 占用内存 4. 序列化与发序列化消耗资源 5. 服务器较多不适合

优点：1. 单个节点服务器重启不影响功能 2. 对应用无侵略 3. 能适用各种负载均衡

实现方法：配置应用的web.xml 配置tomcat的server.xml的cluster属性

4. session统一缓存：(大型分布式场景)

优点：服务器重启不影响性能 2 . 适用于集群 3. 适合各种的负债均衡策略

缺点：对应用有侵入，增加一次网络开销 3. 序列化反序列化小号cpu

### 消息中间件两种模式：

生产者消费者模式和发布订阅模式（观察者模式）

好处：基于事件驱动的编程模式，极大的实现了解耦。

区别：两者都可以实现1对多，多对多的需求。生产者消费者模式借助消息队列实现。发布订阅模式可以订阅自己感兴趣的内容，发布者可以发布不同消息。

## 系统架构设计方法

1. 无状态接口设计

解耦 扩容

2. 并行

|  |  |
| --- | --- |
| 单机模式 | 接口并行调用 （fork/Join） |
| 数据并行处理 |
| 分布式 | 集群并行计算 MpReduce |
| MMP架构 |

3. 缓存

|  |  |
| --- | --- |
| 前端缓存 | 浏览器缓存（304）、CDN缓存、 |
| 应用层缓存 | JVM内存（数据结构）、redis |
| 数据层缓存 | Mysql缓存 mongoDB缓存 |

1. 异步操作

|  |  |
| --- | --- |
| 多线程异步操作 | 不关系运行结果 Runnable |
| 关心运行结果 Callable |
| 基于事件驱动编程 | MQ消息队列（生产者消费之模型）（发布订阅模式） |
| 观察者模式（单机状态Spring有支持） |

6 拆分发

|  |  |
| --- | --- |
| 组件化 | 代码服用 |
| 服务化  (dubbo、SpringCloud) | 分布式系统 |
| 实现资源隔离（CPU、内存、网络、磁盘） |

数据分布算法的设计目标：

* 均匀性(Uniformity) ：不同存储节点的负载应该均衡；
* 稳定性(Consistency)：每次一个key通过数据分布算法得到的分布结果应该保持基本稳定，即使再有存储节点发生变化的情况下。

限流方式：

滑动窗口协议

漏桶算法（桶满了策略：直接丢弃请求、可以暂时存放于队列）

令牌桶：解决突发流量（往桶里面生成令牌，请求都需要拿令牌才能继续）

## 英语词汇

1. manipulate 操作 处理 巧妙地处理
2. roughly 大致 大概 粗略
3. constant （n）常量 常数 (adj)不变的
4. reallocation 再分配
5. externally adv.在（或从）外部，在（或从）外面，外表上；
6. explicitly adv.明白地，明确地；
7. merely adv.仅仅，只不过； 只是； 纯粹； 全然
8. guarantee担保； 担保品； 保修单； 抵押品