# Javaee

## **如果不是以Hash散列查找的集合,即使重写HashCode也没多大实际用处**

## Java序列化的意义（为什么需要序列化）

* 将对象流化，更易在信道上传输或者保存在文件中
* 网络基础结构和硬盘这些硬件组件能够理解bits和bytes，但不了解Java对象

## 1ThreadLocal 是什么？有哪些使用场景？

当执行set方法时，ThreadLocal首先会获取当前线程对象，然后获取当前线程的ThreadLocalMap对 象。再以当前ThreadLocal对象为key，将值存储进ThreadLocalMap对象中。 get方法执行过程类似。ThreadLocal首先会获取当前线程对象，然后获取当前线程的ThreadLocalMap 对象。再以当前ThreadLocal对象为key，获取对应的value。 由于每一条线程均含有各自私有的ThreadLocalMap容器，这些容器相互独立互不影响，因此不会存在 线程安全性问题，从而也无需使用同步机制来保证多条线程访问容器的互斥性。

ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

使用场景：

1、在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。

2、线程间数据隔离

3、进行事务操作，用于存储线程事务信息。

4、数据库连接，Session会话管理。

详细请看以下链接 解释十分完美

<https://blog.csdn.net/weixin_44050144/article/details/113061884>

比较以上两种情况，我们就会发现，内存泄漏的发生跟ThreadLocalMap中的key是否使用弱引用是没有关系的。那么内存泄漏的的真正原因是什么呢？

​细心的同学会发现，在以上两种内存泄漏的情况中，都有两个前提：

1 没有手动删除这个Entry

2 CurrentThread依然运行

**ThreadLocal内存泄漏的根源是**：由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除对应key就会导致内存泄漏。

## 2 HashMap和HashTable区别

HashMap方法没有synchronized修饰，线程非安全，HashTable线程安全； （2）HashMap允许key和value为null，而HashTable不允许

### hashMap是根据key的hashCode来寻找存放位置的，那当key为null时， 怎么存储呢？

当HashMap的put方法，第二个判断就是key为null的判断后进入putForNullKey(V value)这个方法

可以看到，前面那个for循环，是在talbe[0]链表中查找key为null的元素，如果找到，则将value重新赋值给这个元素的value，并返回原来的value。

没找到则将这个元素添加到talbe[0]链表的表头。

HashMap的线程不安全主要体现在下面两个方面：  
1.在JDK1.7中，当并发执行扩容操作时会造成环形链和数据丢失的情况。  
2.在JDK1.8中，在并发执行put操作时会发生数据覆盖的情况。

## HashSet

set 无序，不能索引，所以不能用get（）。可以用迭代器遍历

* HashSet底层原理完全就是包装了一下HashMap
* HashSet的唯一性保证是依赖与hashCode()和equals()两个方法

我们使用Set集合都是需要去掉重复元素的, 如果在存储的时候逐个equals()比较, 效率较低,哈希算法提高了去重复的效率, 降低了使用equals()方法的次数  
当HashSet调用add()方法存储对象的时候, 先调用对象的hashCode()方法得到一个哈希值, 然后在集合中查找是否有哈希值相同的对象  
如果没有哈希值相同的对象就直接存入集合  
如果有哈希值相同的对象, 就和哈希值相同的对象逐个进行equals()比较,比较结果为false就存入, true则不存保证唯一性

## 4线程

五种状态

yield（）执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格， 所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行

join（）执行后线程进入阻塞状态，例如在线程B中调用线程A的join（），那线程B会进入到阻塞队 列，直到线程A结束或中断线程

jvm内存模型

堆是Java虚拟机所管理的内存中最大的一块，是所有线程共享的一块内存区域

栈是每个线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈 互相独立，因此，栈是线程安全的。

Java内存模型

每个线程也有自己的独立工作内存

## 3Hashmap：

理解其底层数据结构

线程不安全

默认容器容量是16，负载因子是0.75,16\*0.75=12，也就是说，当容量达到了12的时候就会进行扩容操作。预估 HashMap 的大小最好，尽量的减少扩容带来的性能损耗

真实数据是存在entry中

* Key :就是写入时的键。
* value :自然就是值。
* next:开始的时候就提到 HashMap 是由数组和链表组成，所以这个 next 就是用于实现链表结构。
* hash:存放的是当前 key 的 hashcode。

hashMap 多线程的话再扩容的时候会造成环形链表，读的时候会造成死循环，cup占用率就会高。

## concurrentHashMap

concurrentHashMap,线程安全，读写快，底层实现是一种以空间换时间的结构，创建的时候直接分了16个segment，每个segment实际上存储的还是哈希表，写入的时候先找到对应的segment，然后对segment加锁，写完，解锁。锁segment的时候其他segment还可以继续工作

1.7：Segment + HashEntry + Unsafe

1.8: 移除Segment，使锁的粒度更小，Synchronized + CAS + Node + Unsafe

### **get没有加锁的话，ConcurrentHashMap是如何保证读到的数据不是脏数据的呢???**

//可以看到这些都用了volatile修饰

volatile V val;

volatile Node<K,V> next;

## JDBC

1. *第一步，加载驱动，创建数据库的连接*
2. *第二步，编写sql*
3. *第三步，需要对sql进行预编译*
4. *第四步，向sql里面设置参数*
5. *第五步，执行sql*
6. *第六步，释放资源*

PreperedStatement

1 Statement会使数据库频繁编译SQL，可能造成数据库缓冲区溢PreparedStatement 可对SQL进行预编译

2 使用占位符 防止sql注入

这样数据库可以减少SQL编译的开销，也能将该类SQL统计在一起，能很容易发现SQL问题。

## 5并发、并行、串行的区别

串行在时间上不可能发生重叠，前一个任务没搞定，下一个任务就只能等着 并行在时间上是重叠的，两个任务在同一时刻互不干扰的同时执行。 并发允许两个任务彼此干扰。统一时间点、只有一个任务运行，交替执行

## 6并发的三大特性

原子性，

可见性， 当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值

有序性 对于那些改变顺序之后不会对最终结果造成影响的代码，虚拟机不一定会按 照我们写的代码的顺序来执行，有可能将他们重排序。

例子 i++线程不安全

synchronized关键字同时满足以上三种特性，但是volatile关键字不满足原子性。

## Synchronized

Synchronized普通方法的锁是this

保证有序性的原理，我们加synchronized后，依然会发生重排序，只不过，我们有同步代码块，可以保证只有一个线程执行同步代码中的代码。保证有序性

可重入一个线程可以多次执行synchronized,重复获取同一把锁。

不可中断一个线程获得锁后，另一个线程想要获得锁，必须处于阻塞或待状态，如果第一个线程不释放锁，第二个线程会一直阻塞或等待，不可被中断。

synchronized的锁对象会关联一个monitor,这个monitor不是我们主动创建的是JVM的线程执行到这个同步代码块,发现锁对象没有monitor就会创建monitor,monitor内部有两个重要的成员变量owner:拥有这把锁的线,

recursions会记录线程拥有锁的次数,当一个线程拥有monitor后其他线程只能等待

monitor 是重量级锁

表格

中度可信度描述已自动生成

面试题synchroznied出现异常会释放锁吗?

会释放锁

面试题：synchronized与Lock的区别

1. synchronized是关键字，而Lock是一个接口。

2. synchronized会自动释放锁，而Lock必须手动释放锁。

3. synchronize是不可中断的，Lock可以中断也可以不中断。

4. 通过Lock可以知道线程有没有拿到锁，而synchronized不能。

5. synchronized能锁住方法和代码块，而Lock只能锁住代码块。

6. Lock可以使用读锁提高多线程读效率。

7. synchronized是非公平锁，ReentrantLock可以控制是否是公平锁。

Linux

操作系统的体系架构分为：用户空间（应用程序的活动空间）和内核

用户态切换至内核态需要传递许多变量, 这种切换就带来了大量的系统资源消耗，这就是在synchronized未优化之前，效率低的原因

### JDK6 做了synchronized优化

不是一下子就是重量级锁

而是 一个优化过程无锁--》偏向锁--》轻量级-》重量级锁

### CAS

的全成是：Compare And Swap(比较相同再交换)。是现代

CPU广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。

CAS操作依赖3个值：内存中的，旧的预估X，要修改的新值B，如果旧的预估值X等于内存中的值V，就将新的值B保存到内存中。

CAS获取共享变量时，为了保证该变量的可见性，需要使用volatile修饰。结合CAS和volatile可以实现无锁并发，适用于竞争不激烈、多核CPU 的场景下

竞争如果激烈 则效率很低

例子 AtomicInteger

### 偏向锁

表格

中度可信度描述已自动生成

偏向锁是在只有一个线程执行同步块时进一步提高性能，适用于一个线程反复获得同一锁的情况。偏向锁可以提高带有同步但无竞争的程序性能。

有线程 竞争 就会撤销偏向锁，升级为轻量级锁

### 轻量级锁

### 自旋锁

循环做一件事，默认循环10次 超过次数就会挂起线程

### 自适应自旋锁

循环时间和循环次数不固定

### 重量级锁

### 锁消除

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

### 锁粗化

图形用户界面, 文本, 应用程序

描述已自动生成

### 平时写代码如何对synchronized优化

1减少synchronized的范围同步代码块中尽量短，减少同步代码块中代码的执行时间

2 降低synchronized锁的粒度

Hashtable 源码在put get remove方法都加了synchronized 锁是this

多线程中 一个线程在put的 其他线程不能进行任何操作

图示

描述已自动生成

一个类中两个没有关联的方法 尽量不要用类名为锁 降低锁的粒度

3读写分离

读取时不加锁，写入和删除时加锁

ConcurrentHashMap，CopyOnWriteArrayList和ConyOnWriteSet

## 7线程池

### 线程池参数

线程池中的几种重要的参数

corePoolSize就是线程池中的核心线程数量，这几个核心线程，只是在没有用的时候，也不会被回收

maximumPoolSize就是线程池中可以容纳的最大线程的数量

keepAliveTime，就是线程池中除了核心线程之外的其他的最长可以保留的时间，因为在线程池中，除了核心线程即使在无任务的情况下也不能被清 除，其余的都是有存活时间的，意思就是非核心线程可以保留的最长的空闲时间，

util，就是计算这个时间的一个单位。

workQueue，就是等待队列，任务可以储存在任务队列中等待被执行，执行的是FIFIO原则（先进先出）。

threadFactory，就是创建线程的线程工厂。

handler,是一种拒绝策略，我们可以在任务满了之后，拒绝执行某些任务。

### 四种拒绝策略

* AbortPolicy(抛出一个异常，默认的)
* DiscardPolicy(直接丢弃任务)
* DiscardOldestPolicy（丢弃队列里最老的任务，将当前这个任务继续提交给线程池）
* CallerRunsPolicy（交给线程池调用所在的线程进行处理)
* 自定义处理方法

### 线程池都有哪几种工作队列？

* ArrayBlockingQueue （有界队列）是一个用数组实现的有界阻塞队列
* LinkedBlockingQueue（可设置容量队列）基于链表结构的阻塞队列，按FIFO排序任务，容量可以选择进行设置，不设置的话，将是一个无边界的阻塞队列，
* DelayQueue （延迟队列）是一个任务定时周期的延迟执行的队列。根据指定的执行时间从小到大排序，否则根据插入到队列的先后排序
* PriorityBlockingQueue 优先级队列）是具有优先级的无界阻塞队列；
* SynchronousQueue

（同步队列）一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，

### 线程池的分类

**1、newCachedThreadPool**

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统OOM。

**2、newFixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

**3、newSingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

**4、newScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。

**5、newSingleThreadScheduledExecutor**

创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。线程池中最多执行1个线程，之后提交的线程活动将会排在队列中以此执行并且可定时或者延迟执行线程活动。

### 线程池线程复用原理

线程池将线程和任务进行解耦，线程是线程，任务是任务，摆脱了之前通过 Thread 创建线程时的 一个线程必须对应一个任务的限制。

其核心原理在于线程池对 Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程

## 双亲委派模型的原理和好处：

双亲委派 把java类加载进jvm内存中

主要两个步骤：1 每个加载器对它加载过的类都有缓存

2 向上查找 ，向下委托加载（每个加载器查找加载路径，有则加载返回，无则向下委托加载）

主要是为了安全性，避免用户自己编写的类动态替换 Java的一些核心类，比如 String。 同时也避免了类的重复加载，因为 JVM中区分不同类，不仅仅是根据类名，相同的 class文件被不 同的 ClassLoader加载就是不同的两个类

自定义加载器 可以加载指定的文件

## JVM

图示

描述已自动生成

### Native 关键字

文本

描述已自动生成

### Jvm内存模型细节

方法区

Static final class 常量池

栈 8大基本类型 对象引用 实例方法 ；

不存在内存回收问题；

堆 实际的对象

日程表

中度可信度描述已自动生成

元空间： 逻辑属于堆 物理不属于堆

### **OOM问题**

全称“Out Of Memory”，翻译成中文就是“内存用完了”

最常见的OOM情况有以下三种：

1 java.lang.OutOfMemoryError: Java heap space ------>java堆内存溢出，此种情况最常见，一般由于内存泄露或者堆的大小设置不当引起。对于内存泄露，需要通过内存监控软件查找程序中的泄露代码，而堆大小可以通过虚拟机参数-Xms,-Xmx等修改。

2 java.lang.OutOfMemoryError: PermGen space ------>java永久代溢出，即方法区溢出了，一般出现于大量Class或者jsp页面，或者采用cglib等反射机制的情况，因为上述情况会产生大量的Class信息存储于方法区。此种情况可以通过更改方法区的大小来解决，使用类似-XX:PermSize=64m -XX:MaxPermSize=256m的形式修改。另外，过多的常量尤其是字符串也会导致方法区溢出。

3 java.lang.StackOverflowError ------> 不会抛OOM error，但也是比较常见的Java内存溢出。JAVA虚拟机栈溢出，一般是由于程序中存在死循环或者深度递归调用造成的，栈大小设置太小也会出现此种溢出。可以通过虚拟机参数-Xss来设置栈的大小

文本, 信件

描述已自动生成

## GC算法

**标记-清除算法、复制算法、标记-整理算法**

## Java spi

SPI全称Service Provider Interface，是Java提供的一套用来被第三方实现或者扩展的接口，它可以用来启用框架扩展和替换组件。

## Linux

通常用ps查看进程PID，用kill命令终止进程。ps命令用于查看当前正在运行的进程。grep是搜索；-aux显示所有状态；

例如：

ps –ef | grep java表示查看所有进程里CMD是java的进程信息。

ps –aux | grep java

kill命令用于终止进程。例如：kill -9 [PID]   -9表示强迫进程立即停止。

# Spirng

**多例模式中，实例只会随着getBean而创建，不会随着容器初始化而创建！也就是说，多例模式只有懒汉**

**单例是饿汉**

## AOP（术语）

​ a）连接点：类里面哪些方法可以被增强，这些方法称为连接点

​ b）切入点：实际被真正增强的方法称为切入点

​ c）通知（增强）：实际增强的逻辑部分称为通知，且分为以下五种类型：

​ 1）前置通知 2）后置通知 3）环绕通知 4）异常通知 5）最终通知

​ d）切面：把通知应用到切入点过程

文本, 表格

描述已自动生成

前置通知（Before）：方法调用之前，对应@Before

后置通知（After）：方法调用之后（不关心方法输出是什么）对应@After

返回通知（After-returning）：目标方法成功执行之后，对应@AfterReturning

异常通知（After-throwing）：目标方法抛出异常之后，对应@AfterThrowing

环绕通知（Around）：方法调用之前和之后，对应@Around

* 后置通知和返回通知的区别

后置通知应用时机在返回通知之后，任何情况下都会应用，而返回通知只有方法正常执行正常返回后执行 也是ruturn 之后

<https://blog.csdn.net/l081499/article/details/119909697>

aop和自定义注解 配合使用

https://blog.csdn.net/weixin\_39816740/article/details/88529433?utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no\_search\_link&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no\_search\_link

aop的应用场景 日志

首先明确我们需要在何时记录日志：

　　1. 查询所有用户时，并没有参数（此示例没有作分页），只有在返回时才会有数据的返回，所以对查询所有用户的方法采用返回通知（AfterReturning）。

　　2. 新增用户时，会带有新增的参数，此时可采用前置通知（Before）。

　　3. 修改用户时，也会带有新增的参数，此时同样采用前置通知（Before）。

　　4. 删除用户时，通常会带有唯一标识符ID，此时采用前置通知（Before）记录待删除的用户ID。

## 注解

### @Autowired：

根据属性类型进行自动装配

### @Qualifier：

根据名称进行注入，这个@Qualifier 注解的使用，和上面@Autowired 一起使用

@Autowired

@Qualifier(value = "userDaoImpl1")

private UserDao userDao;

### @Resource：可以根据类型注入，也可以根据名称注入

Autowired按类型查找，书写方便，不用在后面跟名字，缺点：当一个Service有多个ServiceImpl去实现时，

那么@Autowired会根据名称来查找。如果查询的结果为空，那么会抛出异常

如：@Autowired

private UserRepository userRepository;

有UserRepository多个实现类 那么采用名称匹配方式，在容器中查找名称为userRepository的实例

@Qualifier 虽然用这个个可以解决这个错误，但是引起了效率低下，先按类型查找，再按名字查找

当只有一个Impl实类的时候，随便用哪个都差不多，>=2的时候，最好用@Resource，比@Autowired@Qualifier()效率高

### @Value：注入普通类型属性

@Value(value = "abc")

private String name

## 描述一下Spring Bean的生命周期？

1、解析类得到BeanDefinition 2、如果有多个构造方法，则要推断构造方法 3、确定好构造方法后，进行实例化得到一个对象 4、对对象中的加了@Autowired注解的属性进行属性填充 5、回调Aware方法，比如BeanNameAware，BeanFactoryAware 6、调用BeanPostProcessor的初始化前的方法 7、调用初始化方法 8、调用BeanPostProcessor的初始化后的方法，在这里会进行AOP 9、如果当前创建的bean是单例的则会把bean放入单例池 10、使用bean 11、Spring容器关闭时调用DisposableBean中destory()方法

## 解释下Spring支持的几种bean的作用域。

一般来说，对于无状态的bean使用单例模式，对于有状态的bean使用prototype模式。

request，这种bean在web的request范围内有效，即每次请求时都会产生一个实例。只用于web程序中。  
   4、 session，这种bean在web的session范围内有效。只用于web程序中。  
   5、 global session，这种bean在web的全局session范围内有效。只用于web portlet框架中。

websocket：bean被定义为在websocket的生命周期中复用一个单例对象

## Spring框架中的单例Bean是线程安全的么？

Spring中的Bean默认是单例模式的，框架并没有对bean进行多线程的封装处理。 如果Bean是有状态的 那就需要开发人员自己来进行线程安全的保证，最简单的办法就是改变bean的作 用域 把 "singleton"改为’‘protopyte’ 这样每次请求Bean就相当于是 new Bean() 这样就可以保证线程的 安全了。

Dao会操作数据库Connection，Connection是带有状态的，比如说数据库事务，Spring的事务管理器 使用Threadlocal为不同线程维护了一套独立的connection副本，保证线程之间不会互相影响（Spring 是如何保证事务获取同一个Connection的）

## Spring 框架中都用到了哪些设计模式？

1 工厂方法

2 单例模式

3 适配器模式 这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC 的扩展了。

4 动态代理 AOP

5 观察者模式 监听器的应用

数据库的配置隔离级别是Read Commited,而Spring配置的隔离级别是Repeatable Read，请问这时隔离 级别是以哪一个为准？ 以Spring配置的为准，如果spring设置的隔离级别数据库不支持，效果取决于数据库

## spring事务传播机制

<https://yexindong.blog.csdn.net/article/details/116525588?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no_search_link&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no_search_link>

REQUIRED(Spring默认的事务传播类型)：一种优先按照调用者事务来，

图形用户界面, 文本

描述已自动生成

## spring事务什么时候会失效?

spring事务的原理是AOP，进行了切面增强，那么失效的根本原因是这个AOP不起作用了！常见情况有

如下几种

1、发生自调用，类里面使用this调用本类的方法（this通常省略），此时这个this对象不是代理类，而

是UserService对象本身！

解决方法很简单，让那个this变成UserService的代理类即可！

2、方法不是public的

@Transactional 只能用于 public 的方法上，否则事务不会失效，如果要用在非 public 方法上，可 以开启 AspectJ 代理模式

3、数据库不支持事务

4、没有被spring管理

5、异常被吃掉，事务不会回滚(或者抛出的异常没有被定义，默认为RuntimeException)

# SpringMVC

## 工作流程

1）用户发送请求至前端控制器 DispatcherServlet。

2）DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3）处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器及处理器拦截器

(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 经过适配调用具体的处理器(Controller，也叫后端控制器)

6）Controller 执行完成返回 ModelAndView。

7）HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。8）

DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9）ViewReslover 解析后返回具体 View。

10）DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11）DispatcherServlet 响应用户

图示

描述已自动生成

## 组件简单讲解

1、前端控制器DispatcherServlet

作用：接收请求，响应结果，相当于转发器，中央处理器。

有了DispatcherServlet减少了其它组件之间的耦合度。

2、处理器映射器HandlerMapping

作用：根据请求的url查找Handler

3、处理器适配器HandlerAdapter

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler

4、处理器Handler(需要程序员开发)

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler

5、视图解析器View resolver

作用：进行视图解析，根据逻辑视图名解析成真正的视图

6、视图View(需要程序员开发jsp)

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf…）

## **转发和重定向**

文本

描述已自动生成

## 常用注解

**@RequestParam 用来获得静态URL中传入的参数，@PathVariable 用来获得动态URL中的参数**  
**@RequestParam 表示参数可传可不穿，@PathVariable 表示参数必须传**

@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。

@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给客户。

**如果前台有很多个参数传入，并且这些参数都是一个对象的，那么怎么样快速得到这个对象？**

答：直接在方法中声明这个对象，SpringMvc就自动会把属性赋值到这个对象里面

## 数据显示到前端

String 常用

Model只有寥寥几个方法只适合用于储存数据，简化了新手对于Model对象的操作和理解；

ModelMap 继承了 LinkedMap ，除了实现了自身的一些方法，同样的继承 LinkedMap 的方法和特性；

ModelAndView 可以在储存数据的同时，可以进行设置返回的逻辑视图，进行控制展示层的跳转。

## 拦截器和过滤器区别

过滤器

servlet规范中的一部分，任何java web工程都可以使用

在url-pattern中配置了/\*之后，可以对所有要访问的资源进行拦截

拦截器

拦截器是SpringMVC框架自己的，只有使用了SpringMVC框架的工程才能使用

拦截器只会拦截访问的控制器方法， 如果访问的是jsp/html/css/image/js是不会进行拦截

文本

描述已自动生成

# Mybatis

基本原理

文本

描述已自动生成

## #{}和${}的区别是什么？

${}是字符串替换，#{}是预处理；

Mybatis在处理${}时，就是把${}直接替换成变量的值。而Mybatis在处理#{}时，会对sql语句进行预处理，将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

使用#{}可以有效的防止SQL注入，提高系统安全性

## Mapper接口里的方法，是不能重载的，

因为是使用 全限名+方法名 的保存和寻找策略

# **Mybatis的一级、二级缓存:**

**什么是mybatis二级缓存？**

二级缓存是多个sqlsession共享的，其作用域是mapper的同一个namespace。

即，在不同的sqlsession中，相同的namespace下，相同的sql语句，并且sql模板中参数也相同的，会命中缓存。

第一次执行完毕会将数据库中查询的数据写到缓存，第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。

Mybatis默认没有开启二级缓存，需要在全局配置(mybatis-config.xml)中开启二级缓存。

https://www.cnblogs.com/happyflyingpig/p/7739749.html

**（1）一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。**

**（2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置 ；**

**（3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear 掉并重新更新，如果开启了二级缓存，则只根据配置判断是否刷新。**

# 数据库

## oracle分页和mysql分页的区别：

oracle:

select \* from (

select rownum rn,a.\* from table\_name a where rownum <= x

*//结束行，x = startPage\*pageSize*

)

where rn >= y;

mysql

select \* from stu limit m, n;

*//m = (startPage-1)\*pageSize,n = pageSize*

## 删除多余数据

delete from test a

where a.day in

(select day from test group by day having count(\*) > 1)

and rowid not in

(select min(rowid) from test group by day having count(\*) > 1)

找出重复项，在进行重复同时利用伪列rowid显示最小（或最大）的伪列。 通过唯一标识的伪列进行删除操作。

count(列名)：不包含NULL，但包含重复值项

## 字符串相加

1、采用 || 符号，用'||'来一次串连多个字串。

|  |  |
| --- | --- |
| 1 | [SELECT](https://www.baidu.com/s?wd=SELECT&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao) 'A' || 'B' [from](https://www.baidu.com/s?wd=from&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao) dual;//执行的结果就是AB |

2、采用CONCAT进行连接，CONCAT()只允许两个参数；换言之，一次只能将两个字串串连起来

|  |  |
| --- | --- |
| 1 | [SELECT](https://www.baidu.com/s?wd=SELECT&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao) CONCAT('A','B') [FROM](https://www.baidu.com/s?wd=FROM&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao) dual;//执行结果也是AB |

## Sql优化

<https://blog.csdn.net/it_lihong/article/details/78737296>

in 是把外表和那表作hash join，而exists是对外表作loop，每次loop再对那表进行查询。  
这样的话，in适合内外表都很大的情况，exists适合外表结果集很小的情况。

### 用EXISTS替换DISTINCT：

当提交一个包含一对多表信息(比如部门表和雇员表)的查询时,避免在SELECT子句中使用DISTINCT. 一般可以考虑用EXIST替换, EXISTS 使查询更为迅速,因为RDBMS核心模块将在子查询的条件一旦满足后,立刻返回结果. 例子：

(低效):

SELECT DISTINCT DEPT\_NO,DEPT\_NAME FROM DEPT D , EMP E

WHERE D.DEPT\_NO = E.DEPT\_NO

(高效):

SELECT DEPT\_NO,DEPT\_NAME FROM DEPT D WHERE EXISTS ( SELECT ‘X'

FROM EMP E WHERE E.DEPT\_NO = D.DEPT\_NO);

### 避免在索引列使用not，is null  ,is not null ，以及使用计算。

### ,B-tree索引 is null不会走索引,is not null会走,位图索引 is null,is not null   都会走

# RabbitMQ

1 基础笔记 <https://blog.csdn.net/nihui123/article/details/104534319>

2 交换机类型 笔记 应用场景及实现代码

https://blog.csdn.net/u012060033/article/details/104258895

fanout

交换器就是同步变异步，所有的队列都要发送，因此不需要路由键了

  它会把所有转发到该交换器的消息全部路由到所有与该交换器绑定的队列上，有点像是广播，但是实际上并不是。

direct

  direct 类型的交换器路由规则相对比较简单，是一种一一对应的关系进行路由。将BindingKey和RoutingKey完全匹配的队列进行路由。

系统日志处理场景

* 微服务产生的日志，交给日志处理器处理
* 日志处理服务器有2个服务，分别为info，error
* 服务的直接通信采用direct（发布订阅）

**Topic 模糊匹配**

**header**

依赖与路由键的匹配规则来路由消息，而是根据发送消息的内容headers属性进行匹配

但是这种交换器的性能相对较差、不适用，基本在实际工作中不会用到。

# RPC

* RPC（Remote Procedure Call）远程过程调用，简单的理解是一个节点请求另一个节点提供的服务

核心模块 ：远程调用和序列化

“SpringCloud和Dubbo都是当下流行的RPC框架，各自都集成了服务发现和治理组件。SpringCloud用Eureka，Dubbo用Zookeeper

# Springboot

## 底层原理

**https://blog.csdn.net/qq\_42941671/article/details/108189666**

## 注解

### @ComponentScan

如果扫描到有@Component、@Controller、@Service等这些注解的类，并注册为Bean，可以自动收集所有的Spring组件，包括@Configuration类

## Spring Boot 自动配置原理是什么？

@EnableAutoConfiguration注解、 @Configuration注解和 @ConditionalOnClass注解组成了Spring Boot自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。具体是通过maven读取每个starter中的spring.factories文件，该文件配置了所有需要被创建在spring容器中的bean。

## Spring Boot 是否可以使用 XML 配置 ?

Spring Boot 推荐使用 Java 配置同时支持 XML 配置，通过 @ImportResource 注解加载 XML 配置。

## 什么是 Spring Profiles？

Spring Profiles 允许用户根据配置文件（dev，prod，test等等）来注册 bean。当应用程序在开发环境中运行时，只有某些 bean 可以加载，而在生产环境中，某些其他 bean 也可以加载。比如要求 Swagger 文档仅适用于测试环境，并且禁用所有其他文档，可以使用配置文件来完成

多Profile文件

　　配置文件编写的时，可以是application-{profile}.properties/yml，默认使用application.properties的配置；

激活指定profile(三种方式)

　　1、在配置文件中指定 (xxx.properties/xxx.yml)

**1 spring.profiles.active=dev**