1. 项目描述

项目，是一个基于SpringMVC框架的JavaWeb项目。主要功能是让企业可以在平台上查看旗下船舶的一些信息，如位置信息，航速，和船舶的告警信息。以各种各样的表格形式呈现，我们用的是datatables插件来实现表格的显示，及数据的增删改操作，整个项目业务逻辑比较简单。

1. 项目经常出错的地方

2.1 ajax请求格式

1）简单格式的请求，直接在后端以对应的字段来接收。

2）复杂格式的请求，包含复杂的对象，需要用json格式传输，后端需要用一个pojo类来对应，前段的字段。如前段有个存对象的数组，那么后端pojo中必须用list<pojo>来接收。

2.2 时间格式转换

1）简单格式请求，需要自定义实现convert类的时间格式转换类，来进行转换，需要分别定义String到Date和String到TIMESTAMP两种时间格式转换类，来处理俩种时间格式的转换。

2）Json格式请求，后端json数据映射，默认实现了long型和一些格式化的字符串形式的时间转换，但没有实现精确到秒的格式化字符串形式。我们可以通过@JsonFormat（）注解，来实现和指定一些格式化字符串形式的时间转换（年，月，日，时，分，秒）。

2.3 捕获用户日志

使用springAOP来实现，用户日志的捕获，主要是用户进入了哪个页面，执行了哪些增删改操作。

1. AOP的组成部分

① 通知（Advice）: 自定义的需要增强的功能，比如捕获日志，那么我需要定义一个日志捕获的方法，也就是方法。

② 连接点（joinpoint）：所有类的方法的前后，都是连接点

③ 切入点（pointcut）：具体的某个类的某个方法的前后，即我们需要增强的方法

④ 切面（aspect）：就是我自定义的增强方法所在的类，叫做切面。它有通知和切入点组成。

⑤ 目标（target）：就是被通知的对象。

⑥ 代理（proxy）：我们就是通过用代理模式，将目标对象，代理成新的增强后的对象，将目标对象中需要增强的方法，引入我们的通知，实现增强功能。

1. AOP实现的方式

Bean创建的流程：

① 解析XML文件，创建beanFactory，里面有各个bean的所有信息

② 通过beanFactory中的bean的信息来初始化bean，这个bean是原始的bean，即只进行了属性的注入和依赖的注入。如果有依赖的bean，则先创建依赖的bean

③ 如果bean实现了一些AWare接口，则调用相关的方法，设置一值。

④ 如果bean关联了beanPostProcess接口，就会调用postProcessBeforeInition，对bean进行一些设置。

⑤ 若有init-method方法，执行此方法。

⑥ 如果bean关联了beanPostProcess接口，就会调用postProcessAfterInition，对bean进行一些设置。其中AOP就实现了beanPostProcess接口，在postProcessAferinition方法中对bean进行AOP代理。

代理实现：

① 如果类实现了接口，就用JDK动态代理来创建代理类

② 如果类没有实现接口，就用CGLIB继承原生类来实现代理类

③ 在代理类中需要增强的方法中，织入增强方法，再用反射调用原始类的该方法。

④ 这样用户调用该类的方法，其实执行的是代理类的方法。

springAOP描述：

用代理类包裹切面，把它们织入到spring管理的bean中，也就是说伪装成目标类，通过接口伪装成与目标类有相同方法的类，在相同方法中反射调用目标类的方法和织入增强的方法，即形成了代理类。如果没有接口则使用CGLIB创建一个目标类的子类。

1. 异常处理
2. 配置springMVC自带的simpleMappingExceptionResolver类，再里面配置不同种类异常所返回的错误页面，这里需要自定义一些不同的异常类，如BussinessException，parmaterException。系统将会根据所抛出的异常类型返回特点的错误页面
3. 定义一个全局异常处理类实现HandlerExceptionResolver接口。这样抛出的异常最后会到自定义的全局异常处理类中，我们在这个类里，对不同的异常做处理。
4. 使用@ExceptionHandler和@controllerAdvice注解。定义一个全局异常处理类，用@controllerAdvice注释，使抛出的异常会到此类中来，在用@ExceptionHandler来注释不同的方法，每个方法处理不同的异常信息。和第二种差不多。

第一种的缺点是，只能简单的返回特点的错误页面，不能对数据进行处理。

1. spring上下文

Spring具有子类上下文和父类上下文的概念，子类上下文可以读取父类上下文中的信息，但父类上下文不能读取子类上下文中的信息。

1. 多线程之WAIT、JION、YIELD、SLEEP
2. Thread.sleep(n)

线程释放CPU时间，进入其他阻塞状态，但没有释放锁，时间到了，进入可运行状态，再一次获取到CPU时间，则进入运行状态。

1. Thread.yeild()

释放CPU时间，进入可运行状态，查看是否有相同优先权的线程，如果有将执行权交给它，如果没有继续运行。

1. Object.wait()

必须在Synchronized代码块中，Object是Synchronized中指明的对象。

进入Object对象的等待队列中，需要此Object对象调用notify()|notifyAll()方法唤醒，进入锁池队列中，当某个线程获取到锁时，进入可运行状态，直到获取到CPU时间，进入运行状态。

1. Object.join()

Object为某一线程对象，在当前对象中调用次方法，使当前线程获取了Object线程锁，并调用了Object.wait()方法，进入等待队列中，只要Object线程活着，就一直调用等待，只有Object线程执行完，自动调用notifiyAll()唤醒当前线程。

1. 死锁产生的条件
2. 互斥条件：一个资源只能被一个线程占用，如果此时另一个线程又来请求次资源，则此线程将进入等待队列。若是公平竞争则进入先进先出队列。
3. 请求保持条件：线程在申请其他资源的同时，仍然占有原来的资源，知道获取到其他资源为止。
4. 不可剥夺条件：线程所获得的资源，只能有该线程释放。
5. 循环等待条件：发生死锁时，必然存在一个循环等待队列（P1,P2,..,PN）

我们可以通过破坏死锁产生的4个必要条件来 预防死锁，由于资源互斥是资源使用的固有特性是无法改变的。

1. 破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到 系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行。

2）破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。

3）破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

1. 线程同步辅助类

Synchronized关键字：

1. 作用于方法上时，锁住的是对象实例（this）；
2. 作用于静态方法时，锁住的是类实例；
3. 作用于某一对象实例时，锁住的是对应的对象实例；

自旋锁：

线程在进入等待队列时首先进行自旋尝试获得锁，如果不成功再进入等待队列。这对那些已经在等待队列中的线程来说，稍微显得不公平。

偏向锁：主要解决无竞争下的锁性能问题

现在几乎所有的锁都是可重入的，每次加锁/解锁都会涉及到一些CAS操作（比如对等待队列的CAS操作），CAS操作会延迟本地调用

偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在接下来的运行过程中，若该锁没有被其他的线程访问，则持有偏向锁的线程将永远不需要触发同步。

如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会尝试消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

(偏向锁只能在单线程下起作用)

因此 流程是这样的 偏向锁->轻量级锁->重量级锁

安全点

轻量级锁

**轻量级锁加锁：JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，，**然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

**轻量级锁解锁：**轻量级解锁时，会使用原子的CAS操作来将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。

ReentrantLock，ReentrantReadWriteLock.ReadLock、ReentrantReadWriteLock.WriteLock

reentrantLock有个多条件控制

T1.await()， T1.signal()，T1.singalAll()；

Semaphore（信号量）：流量控制

New Semaphore(n);

Semaphore.acquire();

Semaphore.release();

Acquire 是在try类，release在finally中

CountDownLatch：

CountDownLatch controller;

Controller = new countDownLatch(n);

Controller.countDown();--每个线程走到这减1

Controller.await();--线程在这里等待，countDown减为0为止

有两个run类，一个run类，中执行await，

另一个run类，调用前一个类中的countDown，唤醒前一个线程

CyclicBarrier：使两个或多个线程在某个点上进行同步。

New cyclicBarrier(线程数，runnable类)；

Barrier.await();--当所有线程到达时，继续执行

有两个run类，一个run类中执行await

一个run类是，所有线程到达后，执行的方法

Phaser：执行并发多阶段任务

New phaser(3);

Phaser.arriveAndAwaitAndAwaitAdvance();--所有线程到达，则继续下一阶段任务

Phaser.arriveAndDeregister();--注销此线程

有一个run类，在这run类中执行，或调用有arrive...的方法

重写onAdvance()方法，phaser阶段改变的时候会被自动执行，可以执行一些特定的操作，如打印现在的线程状态

Exchanger：并发间的数据交换

Exchanger.exchange(object);--交换点

有两个run类，一个生产者，一个消费者。

生产者和消费者都到交换点之后，交换数据

Executors.newFixedThreadPool(5);--固定线程池

Executors.newCachedThreadPool();--线程池

1. 事务
2. 事务隔离级别
3. 默认：使用底层数据库的默认隔离级别
4. Read\_uncommitted：读未提交
5. Read\_committed：读已提交
6. Repeatable\_read：可重复读，一个事物对某个数据进行操作，其他事务不能修改
7. Serializable：串行读，所有事务依次执行
8. 事务的传播行为
9. required：如果当前存在事务，则加入该事务，若不存在，创建新事务
10. Required\_new：首先创建一个事务，如果当前存在事务，则挂起当前事务
11. Supports：如果当前存在事务，则加入该事务，若没有，则以非事务运行
12. Not\_supported：以非事务方式运行，如果存在事务，则把当前事务挂起
13. Never：以非事务方式运行，如果存在事务，则抛出异常
14. Mandatory：如果存在事务，则加入该事务，如果没有则抛出异常
15. Nested：嵌套事务，事务中的事务。