目录

[问题 2](#_Toc460531815)

[附录 13](#_Toc460531816)

[附录一 13](#_Toc460531817)

[附录二 14](#_Toc460531818)

[附录三 14](#_Toc460531819)

# 问题

1. maven项目生命周期

Maven有三套相互独立的生命周期,

* Clean Lifecycle 在进行真正的构建之前进行一些清理工作。
* Default Lifecycle 构建的核心部分，编译，测试，打包，部署等等。
* Site Lifecycle 生成项目报告，站点，发布站点。

它们是相互独立的，你可以仅仅调用clean来清理工作目录，仅仅调用site来生成站点。当然你也可以直接运行 **mvn clean install site** 运行所有这三套生命周期。

每套生命周期都由一组阶段(Phase)组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。

Clean生命周期一共包含了三个阶段：

* pre-clean  执行一些需要在clean之前完成的工作
* clean  移除所有上一次构建生成的文件
* post-clean  执行一些需要在clean之后立刻完成的工作

在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行，也就是说，**mvn clean** 等同于 **mvn pre-clean clean** ，如果我们运行 **mvn post-clean** ，那么 pre-clean，clean 都会被运行。

Site生命周期的各个阶段：

* pre-site     执行一些需要在生成站点文档之前完成的工作
* site    生成项目的站点文档
* post-site     执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
* site-deploy     将生成的站点文档部署到特定的服务器上

Maven的最重要的Default生命周期，绝大部分工作都发生在这个生命周期中，这里，我只解释一些比较重要和常用的阶段：

* validate
* generate-sources
* process-sources
* generate-resources
* process-resources     复制并处理资源文件，至目标目录，准备打包。
* compile     编译项目的源代码。
* process-classes
* generate-test-sources
* process-test-sources
* generate-test-resources
* process-test-resources     复制并处理资源文件，至目标测试目录。
* test-compile     编译测试源代码。
* process-test-classes
* test     使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。
* prepare-package
* package     接受编译好的代码，打包成可发布的格式，如 JAR 。
* pre-integration-test
* integration-test
* post-integration-test
* verify
* install     将包安装至本地仓库，以让其它项目依赖。
* deploy     将最终的包复制到远程的仓库，以让其它开发人员与项目共享。

1. 集合结构图
2. Iterable 获取迭代器
   1. Collection
      1. List
         1. ArrayList 顺序表 内部是一个对象数组
         2. LinkedList 链表 内部包含一个头和一个尾结点，每个结点都包含了该节点存有的对象和prev，next结点
         3. Verctor List子类，线程安全版的ArrayList
            1. Stack，增加了入栈出栈操作，功能没有Deque全面
      2. Set
         1. HashSet 内部一个HashMap，实际操作由Map执行
         2. TreeSet 默认内部维护一个TreeMap，实际操作由Map执行，add时对加入元素排序，通过Comparable或者Comparator
      3. Queue 队列 可实现FIFO结构
         1. 子接口Deque 双向队列 可实现FIFO 栈结构
            1. LinkedList实现了Deque,故也能在头尾结点加入元素
3. Listarraylist set 区别

不解释

1. 多线程sleep wait 区别

1、这两个方法来自不同的类分别是Thread和Object

2、wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在

　　任何地方使用（使用范围）

　　synchronized(x){

　　x.notify()

　　//或者wait()

　　}

最主要区别：

sleep使线程进入了休眠状态，对应Thread.State.TIMED\_WAITING,经过具体等待的时间之后线程重新进入运行状态

wait需要在同步块内使用，将锁释放，并使线程进入等待状态，对应Thread.State. WAITING。当在另一个同步块内对该对象进行notify操作时，监视该对象的线程需要能重新获得对象锁才能进入运行状态。

线程的状态：NEW，RUNNABLE，BLOCKED，WAITING，TIMED\_WAITING，TERMINATED

其中BLOCKED状态是如何进入的？

比如一个线程synchronized(对象)之后若没有及时释放，那么另一个进行synchronized(对象)的线程会进入BLOCKED状态。

1. Stringstringbuffer区别

StringBuffer内部维护一个char[]，toString操作会根据这个数组new一个String对象，属于建造者模式。

1. Redis存储的数据类型

String，HashMap，List，Set，SortedSet

1. Linux常用命令

ll

pwd

ps -ef |grep tomcat 显示包含tomcat的进程

kill -9 杀进程

tail -f xxx.out 显示文本文件尾部，并跟踪最新插入的数据。

rm xx 移除xx文件

cat /etc/crontab

1. Rabbitmq使用类型

点对点(基于queue)，发送消息到队列，监听队列接收，消费端可设置是否ack来实现是否可重复接收

订阅发布(基于exchange-fanout)，一对多，该exchange上绑定的所有队列都接受

路由发送(基于exchange-direct)，一对多，该exchange上绑定的跟routing key相同的队列都接受

按topic发送(基于exchange-topic)，发送消息的routing key不是固定的单词，而是匹配字符串，如"\*.lu.#"，\*匹配一个单词，#匹配0个或多个单词

共四种路由类型

Direct：dog->dog queue

Fanout：dog->dog queue1,dog quque2,.....

Topic：dog.# -> dog.a.b.c.d dog.\*->dog.a

Headers：跟消息头路由，而忽略实际路由键，如果绑定的key和消息头相符则认为符合要求

1. Agurajs

它一个js类库，将html中的标签(称为视图) 关联 到数据模型上(称为model)

比如<div>{{aaa}}</div> 关联模型 aaa，

当你改变模型aaa的值时对应的视图也会发生改变，也就是页面会呈现不一样的数据，就不必通过js的document.getElementByxxx(...)获得然后修改innerHtml了。

如果视图是类似<input type="text" ng-model="aaa"/>这样的可改变对象的视图，那么视图数据的变化会被更新到对应的model(aaa)。

也就是说 视图和模型，无论哪一方数据变化都会导致另一方数据发生改变，这是一种双向绑定。

angular核心就是建立在这样的绑定之上，其它都是常规需求的实现，比如循环输出一个list的model、表单验证、http请求。

10、mybatis

mybatis原本是开源项目iBatis，2010由ASF迁移到google code后改名的。

它是一个orm框架，包括：

1. Sql Maps 实质是对DAO的实现，包含了对象到表的映射关系，以及要执行的sql
2. DAO java接口类，操作数据库对应的对象

有了Map和DAO，相当于有了个实现类了，现在只要new出来就可以实现对数据库的sql操作了。

那么如何new出这个实现类来？

利用SessionFactoryBuilder根据配置的xml可以build出一个SessionFactory实例，通过SessionFacotry打开一个SqlSession实例，通过这个SqlSession实例可以getMapper(DAO接口类.class)就能获得实现对象了。

特点：

1. 动态SQL ——远离拼接sql的烦恼
2. 使用简单，直接使用sql，调优方便，但切换数据库改动较大

11、spring实现机制

为XML配置的、或者加了注解的类生成BeanDefinition，容器加载后根据需要生成单例存在Map里。

容器的启动机制（位于refresh方法中）：

创建一个BeanFactory，配置它，将BeanDefinition读取到BeanFactory中（比如从Xml中读，从注解中生成，从Groovy类读），这属于刷新BeanFactory的操作；得到这个BeanFactory后，对其做默认的配置（如注册通用单例，注册Bean候选，忽视一些依赖，加入Bean拦截器）；配置后完执行BeanFactory的拦截器；又注册BeanDefinition中定义的Bean拦截器，注册一些事件发布/监听者；最关键的一步来了，冻结BeanFactory配置，生产所有非延迟单例，在生产每个对象前后会调用Bean拦截器对应方法，以及特殊创建时期的特定拦截器方法；最后就是一些扫尾工作，如广播给监听者容器已经启动完毕。

上面就是正常的启动流程，对于这个流程只捕获了一个BeansException异常，捕获到后会把BeanFactory已经产生的那些单例给处理掉，然后设置容器状态为非活动的，宣告着本次启动失败。

12、注解的机制

在字节码文件中，注解的access\_flag为2601，对应着它是public的，是个接口，是抽象的，是注解，也就是它属于一种特别的接口。

注解可以设置只能出现的地方（如仅在类上，方法上），通过元注解Target配置，以及它所作用的时机（如编译后擦除，或在class中保存，或在运行时仍有效），通过元注解Retention配置。

那么如何感知到注解被配置到了类、方法、字段上？

通过相应反射类——Class、Method、Field的与Annotation有关的方法，如Class的isAnnotationPresent(xx.class)判断在类上是否标注了xx注解，若有，可以调用getAnnotation(xx.class)获得一个xx注解类的实例对象，就可以访问具体配置的属性了。（前提是该注解类的Retention被配置为运行时仍有效的）

13、spring事务的实现方式

1. 使用AOP（tx标签）的方式引用一个事务管理器，切入到切入点。
2. 注解——Transactional
3. 定义TransactionInterceptor，并在BeanNameAutoProxyCreator中的interceptorNames引用这个拦截器。

14、spring获取bean的方式

1. @Autowired / @Resource
2. 获得applicationContext，调用getBean API
3. xml里面<property><constructor-arg>

怎么获取Application

1. 通过ApplicationContextAware
2. web中WebApplicationContextUtils.getRequiredWebApplicationContext(ServletContext sc)

WebApplicationContextUtils.getWebApplicationContext(ServletContext sc)

ContextLoader.getCurrentWebApplicationContext();

15、map遍历的几种方式

1. 迭代EntrySet 通过迭代Node[]，同时取Key、value值
2. 迭代KeySet，通过迭代Node[]取Key值，根据key经过hash计算和比较得到value
3. 迭代values 通过迭代Node[]，取Value值，无法取得key值

16、spring事务机制

包括声明式事务和编程式事务

1. 声明式事务（无需关注连接、事务提交、回滚等操作，也无需使用try..catch）
   1. 事务属性（在TransactionDefinition中定义）
      1. getPropagationBehaviour()//传播行为
      2. getIsolationLevel()//
      3. getTimeout()
      4. isReadOnly()
2. 编程式事务
   1. 获得TransactionManager
   2. tm.getTansaction()
   3. tm.begin()
   4. tm.suspend(0
   5. tm.resume()
   6. ts.commit()
   7. ts.rollback()

事务传播行为：（指方法所在的环境如何传播给方法）

1. PROPAGATION\_REQUIRED——若存在则支持，若没有则开启 （大部分需求）
2. PROPAGATION\_SUPPORTS——若存在则支持，若没有则不开启
3. PROPAGATION\_MANDATORY——若存在则支持，若没有则异常
4. PROPAGATION\_REQUIRES\_NEW——始终开启新的(需使用JtaTransactionManager)
5. PROPAGATION\_NOT\_SUPPORTED——始终不开启，若存在则挂起(需使用JtaTransactionManager)
6. PROPAGATION\_NEVER——始终不开启，若存在则异常
7. PROPAGATION\_NESTED——若存在则嵌入，若没有则按REQUIRED（外层失败会回滚嵌入事务，而内层事务失败则不会影响导致外层事务失败，外层进入内层之前存有savepoint，内层失败时外层只会回退到这个点）

（若用DataSourceTransactionManager则需JDBC的SavePoint）

参考：<http://www.open-open.com/lib/view/open1350865116821.html>

17、spring事务隔离级别

与JDBC四个级别对应，还一个Default对应数据库默认隔离级别

1. DEFAULT
2. READ\_UNCOMMITED 可能脏读（读取了回滚数据）
3. READ\_COMMITED 可能不可重复读（读取前后不一致）和幻象读
4. REPEATABLE\_READ 可能幻象读（读取到增量）
5. SERIALIZABLE 代价最高

18、httpsession实现机制

session是在服务器第一次执行getSession()语句时才创建的服务器先从浏览器带来的cookie中查找JSESSIONID，是否有相关可用的session，有就调用，如果没有，再从超链接的URL中查找 JSESSIONID，如果还没有，就会创建一个新的session。

服务器会为这个新session分配id，然后以cookie的形式返回给客户端——JSESSIONID=xxx，这样不管用户访问哪一个页面，请求时浏览器都会带上cookie信息，服务端根据id就能获得保存的session对象，也就能访问该用户的信息了。

需要注意的是：

1. 若禁用cookie，session会失效，此时可以通过重写url的方式附带sessionId,示例：xxx.do;JSESSIONID=123?a=b
2. session对象位于服务端，是有生命周期的，cookie存在于客户浏览器，也是有生命周期的，两者生命周期不一致都会导致用户登录失效的现象。比如在一个页面停留很久，没有任何操作，然后刷新一下需要登录了，很可能服务器的session对象生命已到期。

19、session和cookie的区别

session：存在于服务端，服务器生成并保存，是一个内部包含了线程安全HashMap的对象，一般大小不受限制

cookie：由客户浏览器管理，服务器生成并通过相应头让浏览器保存，包含名字、值、过期时间、路径和域，大小一般很小（最多4k），

20、分布式事务实现方式及原理

分布式事务

1. 包含：
   1. 事务管理器TransactionManager
   2. 一或多个资源管理器ResourceManager
2. 规范
   1. XA X/Open组织提出
      1. JTA java规范，定义了对XA的支持
         1. JBOSS容器已实现
         2. Tomcat、Jetty可以选择独立实现——如JOTM、Atomikos
3. 原理
   1. 两阶段提交（事务管理器S和资源管理器C的协作方式）（XA事务）
      1. 过程
         1. 准备阶段
            1. S通知所有C开始执行，耗时的操作都在这一步执行，所有C写本地redo和undo日志，但不提交
         2. 提交阶段
            1. S根据C的反馈判断是否让所有C提交或回滚，最后释放锁资源
      2. 优点
         1. 保证了事务的原子性
      3. 缺点
         1. 多结点之间的协调导致耗时较多，性能下滑，大请求下回导致事务积压甚至死锁
      4. 应用场景
         1. Mongo
            1. 原理(<http://blog.sina.com.cn/s/blog_89d963070102vciw.html>)

对单个文档操作总是原子性的

* + - * 1. 步骤

新建事务记录tran,包含了事务具体操作，状态initial

切换tran到状态pending

将事务具体操作应用到文档，并把当前事务id加入到文档的字段中，并确保不对已执行过的再次执行

切换tran到状态commited

移除受影响文档中的事务数组字段

切换tran到状态done

* 1. 一阶段提交
     1. 过程
        1. 应用像数据库发出提交或回滚请求，然后收到执行结果
     2. 优点
        1. 不存在协调，耗时短于两阶段提交
     3. 缺点
  2. 事务补偿
     1. 方式
        1. 实时补偿
           1. 优点：数据准确
           2. 缺点：并发很低
        2. 最终一致性补偿
           1. 优点：并发高
           2. 缺点：延时很高
     2. 实现
        1. 日志比对
        2. 定期数据同步
        3. 利用MQ落地实现强一致性

1. 概念
   1. 网络通讯危险期：（发出请求后，服务器返回应答前）
      1. 没有成功到达服务器
      2. 已成功到达，但没响应给客户端
2. 意义
   1. 确保了原子性

21、数据库乐观锁，悲观锁

1. 悲观锁
   1. 含义：每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。
   2. 适用范围：读多写少，可提高吞吐量
   3. 数据库场景：
      1. 行锁：select xx from ..where.. for update
      2. 表锁：select xx from ..where.. with(HOLDLOCK(读锁),TABLOCKX(写锁))
      3. MYSQL中有三种
         1. 表级锁
            1. 类别

表共享锁（读锁）

表独占锁（写锁）

* + - * 1. 说明

MyISAM在select前自动加表读锁，更新前自动加表写锁

显示加锁：

LOCK TABLES [NAME] read local,[NAME2] read local

unlock tables

InnoDB采用了行级锁

show status like 'innodb\_row\_lock%' 观察锁争夺情况

行锁

共享锁

select..... lock in share mode

排它锁

selct ... for update

表锁(使用LOCK TABLES...UNLOCK TABLES)

意向共享锁

意向排它锁

* + - 1. 行级锁
      2. 页面锁

1. 乐观锁
   1. 含义：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。
   2. 适用范围：写多的场景，行锁、表锁、读锁、写锁、都是操作前先上锁

22、数据库行级锁，表级锁原理，应用场景

23、数据库索引生效失效场景

1. 生效
   1. 参与join的字段在表中均建立了索引，则join操作会使用
   2. 对建立了索引的字段做sort或group，求min、max也会，=、>、>=、<、<=、between，is null
   3. 存在多个字段的联合索引，这个联合索引的最左前缀匹配字段也会被自动作为索引来加速查找。例如，若为某表创建了3个字段(c1, c2, c3)构成的联合索引，则(c1), (c1, c2), (c1, c2, c3)均会作为索引，(c2, c3)就不会被作为索引，而(c1, c3)其实只利用到c1索引。
   4. 快速查找符合where条件的记录
   5. 优先使用能使记录规模最小的那个索引
2. 失效（总思路：通过日志系统监控各功能执行时间，找到源头并解决）
   1. 对列进行计算（包括执行函数，类型转换）
   2. 数据严重倾斜，亦数据分布不均匀(也就是引擎认为全表扫描比使用索引快)
   3. 前面有%号的like使用 like '%xx'不使用， like 'xx%'可能使用
3. 经验
   1. 强制使用索引 select xx from tb force index(primary) order by ...
   2. 多使用explain确保所写的sql使用了索引

24、消息队列类型，区别，应用场景

队列是消息的安全存放地，队列存储消息直到它被应用程序处理。

JMS规范下消息传递类型

1. 点对点 PTP
   1. 概念
      1. 发布的消息直接进入队列，消费端争抢式消费
2. 发布-订阅 Pub/Sub
   1. 概念
      1. 发布的是topic，1个topic下有 多个Subscription，四种类型如下
         1. SubScription:单个client，client断开则该~自动销毁
         2. Durable ~:单个client,client断开消息会保存到~,client重新连接继续发送
         3. Shared ~:多个client，消息只能被其中一个消费
         4. shared Durable ~:多个client，就算所有client断开，该~继续存在

消息中间件类型

1. ActiveMQ
2. RabbitMQ
3. kafkaMQ
4. zeroMQ
5. metaQ
6. memcached
7. redis

25、忘记密码功能实现原理

用户流程:

　　1.用户忘记密码，来到密码重设界面

　　2.用户输入Email地址，点击重设密码按钮

　　3.用户收到一封密码重设邮件，里面有重设密码的链接，此链接有过期时间

　　4.用户点击链接，来到密码重设页面，输入新密码，完成

　后端实现方式:

　　1.当用户输入Email地址后，验证这个Email，如果存在于数据库，那么取得用户的user\_id

　　2.将user\_id和当前时间戳编码成HASH，需要提前准备好一个KEY，并且这个KEY只存在服务器上。 HASH = md5 ( user\_id + timestamp + KEY )

　　3.生成一个URL，并且附带刚刚生成的HASH和用户id以及timestamp，比如 http://domain.com/reset-password.php?hash=HASH&user\_id=123×tamp=1392121211

　　4.当-三-联-用户访问这个URL, 检查HASH是否合法: HASH == MD5 ( user\_id + timestamp + KEY)

　　5.检查 timestamp 是不是过期了

　　6.如果所有检查通过，那么显示一个新密码表单给用户

这种方式的好处:

　　1.不需要额外的数据表

　　2.不用担心参数被用户恶意修改，因为要检查hash是不是等于那几个参数的md5

　　3.密码重设URL自带时间戳

　　4.只要KEY设置的足够长足够复杂，那么可以认为HASH是绝对安全的

26、redis散列实现机制

27、数据库优化

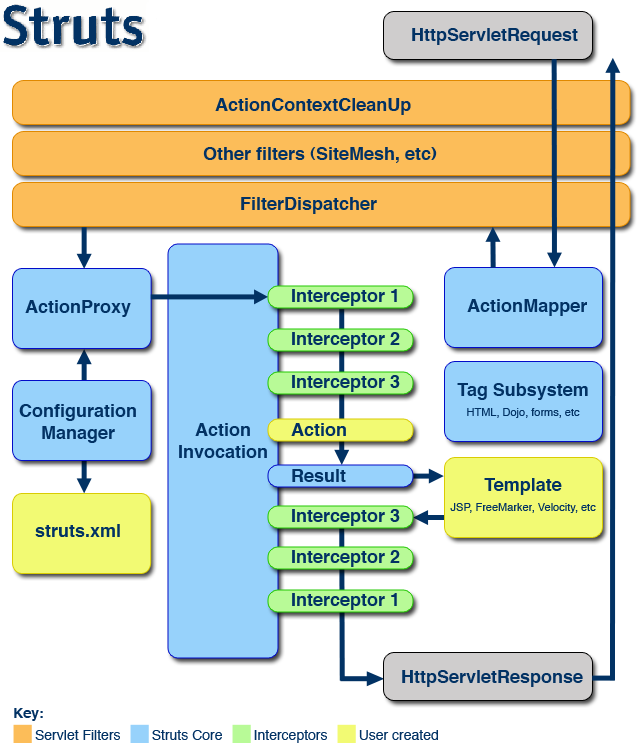
28.Struts2和spring mvc 有什么区别

1. struts2:
   1. 核心是一个filter，StrutsPrepareAndExecuteFilter
   2. 流程:([附录一](#_附录一))
      1. 客户端发起请求
      2. 经过一系列filter最终到FilterDispatcher
      3. FilterDispatcher询问ActionMapper是否要调用某个Action
      4. 若要调用，则把请求交给ActionProxy
      5. ActionProxy通过ConfigurationManager询问框架配置文件(struts.xml)，找到Action类
      6. ActionProxy创建一个ActionInvocation的实例
      7. ActionInvocation依次调用多个拦截器，action返回result，调用多个拦截器
2. Spring mvc：
   1. 核心是一个servlet，DispatcherServlet
   2. 流程: ([附录二](#_/附录二)，[附录三](#_/附录三))
      1. 迭代HandlerMapping
      2. 根据每一个HandlerMapping通过request获取HanlderExecutionChain，不为空时立即返回
      3. 获得HanlderExecutionChain的handler对象
      4. 迭代HandlerAdapter
      5. 根据每一个HandlerAdapter判断是否支持该handler，若支持立即返回
      6. 执行HanlderExecutionChain中的拦截器前置方法
      7. 执行HandlerAdapter方法返回ModelAndView（执行中涉及到数据转换、格式化、验证）
      8. 执行HanlderExecutionChain中的拦截器后置方法
      9. 若ModelAndView的view是引用，则迭代ViewResolver直到能返回一个不为空的view
      10. 对view进行render

29.TCP协议

# 附录

## 附录一



## http://img.my.csdn.net/uploads/201304/13/1365825529_4693.png附录二

## http://img.my.csdn.net/uploads/201304/13/1365825551_8302.png附录三