线程的创建方式？

1. 继承Thread类；
2. 实现Runnable接口，无返回值；new thread(new Runnable(){...}).start();
3. 实现Callable接口，通过Future对象获取结果；
4. 构建线程池；pool.execute(()->{...})

线程让步（yield，一时没想起来）sleep、yeild、join

1. yeild方法，进入就绪状态，把CPU资源让给比它权限更大的进程
2. Sleep方法，进入堵塞状态，释放cpu资源，不会释放锁
3. Join方法，进入堵塞状态， 只有调用的当前线程执行完毕之后，才执行后面的程序代码！！
4. Wait方法，释放cpu，同时也释放锁
5. Interrupt方法中断线程 、 interrupted、isInterrupted

守护线程的作用？

Java中有两类线程：****User Thread(用户线程)、Daemon Thread(守护线程)；****

****用户线程****即运行在前台的线程，而****守护线程****是运行在后台的线程。 守护线程作用是为其他前台线程的运行**提供服务和支持**，而且**仅在非守护线程仍然运行时才需要**，比如垃圾回收线程就是一个守护线程。

总结：线程分为 用户线程和守护线程 ， 其中守护线程主要作用就是：为用户线程的正确运行，提供支持和服务的，比如垃圾回收线程gc

1. IO操作有哪些类？**主要涉及了两类设计模式**
2. **适配器模式（比如字符流和字节流的转换）**
3. **使用filterInputstream实现装饰着模式(继承这个类，重写里面的功能方法，增加额外功能）**

字符流：FileReader、FileWriter（指定编码集或者追加内容）、BufferedReader、BufferedWriter、InputStreamReader、OutputStreamWriter

字节流： FileInputStream、FileOutputStream、

BufferedReader br = new BufferedReader(new fileReader（file））

String st = Br.readLine();

7）异常体系？

Throwable接口：Error（StackOverflowError 重复递归地调用、 outOfMemoryError 重复地产生新的大的对象）和Exception（检查和非检查）

Throws在方法上面 Throw new Exeception（）；

**运行时exeception：比如空指针异常、数组越界、类型转换错误、类没有找到、算术错误、字符串越界等待等 ； 在编译的时候，可以正常编译通过**

**非运行时exeception：比如IO、Sql异常，值得是：在编译的时候，需要使用try catch异常捕获机制处理，不然编译不能通过的异常**

get 和post的区别

1. get方式，提交的数据会在地址栏显示，？前是url地址，？后是传输数据
2. Get方式提交数据大小限制在1kb
3. Get方式数据明文传输，不安全

session和cookie的区别？

1. cookie保存在浏览器端，session数据保存在客户端
2. Cookie只能存储String类型对象，session可以存储任何Java对象
3. cookie的话本身是只能保存string类型的信息的，这就需要先序列化，然后接收的页面反序列化后形成对象调用
4. Cookie不够安全，在浏览器缓存在，能够拿到数据
5. Session存储在服务器端，数据多时，会影响到服务器的性能
6. Cooke只能存放4kb大小的数据

**spring创建bean的方式？**

### **1.调用构造器创建Bean**

使用构造器来创建Bean是最常见的情况，如果是构造注入，则通过配置构造函数来实现创建Bean，如果是设值注入，Spring底层会调用Bean类的无参数构造器来创建实例。

### **2.调用静态工厂方法创建Bean**

所谓静态工厂方法就是建立一个工厂类，工厂类有个静态工厂方法，让产品的实例化在工厂类的静态工厂方法中进行，让整个过程都面向接口编程。

需要配置beans.xml的配置：

package handle;

public interface Being {

public void testBeing();

}

public class Dog implements Being {}

public class Cat implements Being {}

public class BeingFactory {

//返回Being实例的静态工厂方法

public static Being getBeing(String arg)

{

//调用次静态方法的参数是dog，则返回Dog实例

if(arg.equalsIgnoreCase("dog"))

{

return new Dog();

}

else

{

return new Cat();

}

}

}

### **3.调用实例工厂方法创建Bean**

调用实例工厂方法与调用静态工厂方法类似，只是工厂类中的工厂方法有无static之分，以及配置上内容稍有差异，为便于区分，仍然使用调用静态工厂方法的示例。

需要配置beans.xml的配置：

public interface Being {

public void testBeing();

}

public class Dog implements Being {}

public class Cat implements Being {}

public class BeingFactory {

//返回Being实例的静态工厂方法

public Being getBeing(String arg)

{

//调用次静态方法的参数是dog，则返回Dog实例

if(arg.equalsIgnoreCase("dog"))

{

return new Dog();

}

else

{

return new Cat();

}

}

}

# **总结：1、依赖注入三种方式：set注入、构造注入、注解注入**

**2、spring实例化Bean三种方式：构造函数方式、通过静态工厂方法、通过实例工厂方式**

# Spring之依赖注入（设值注入、构造注入、注解注入）

### **两者区别**

### 设值注入：解决循环依赖

a)与传统的Javabean的写法更相似，通过setter方法设定依赖关系显得更加直观自然

b)对于复杂的依赖关系，如果采用构造注入，会导致构造器过于臃肿；多参数情况下使得构造器变得更加笨重

构造注入：

建议采用以设值注入为主，构造注入为辅的注入策略。对于依赖关系无须变化的注入，尽量采用构造注入；而其它依赖关系的注入，则考虑设置注入。

**总结：1、set注入方式可以解决单例作用域的循环依赖问题 2、当依赖关系比较复杂的时候，采用构造注入方式，会使得构造器过于臃肿，最好采用set注入 ，构造注入无法解决循环依赖问题 3、当参数比较多的时候，采用构造注入，比较麻烦**

Spring如何解决循环依赖

Spring容器循环依赖包括**构造器循环依赖和setter循环依赖**

实例：比如testA用到TestB ， TestB用到了TestC ， TestC用到了TestA，造成循环依赖

****1．构造器循环依赖（无法解决，抛出异常）****

如在创建TestA类时，构造器需要TestB类，那将去创建TestB，在创建TestB类时又发现需要TestC类，则又去创建TestC，最终在创建TestC时发现又需要TestA，**从而形成一个环，没办法创建。**

****2．setter循环依赖（可以解决单例作用域的循环依赖问题，通过提前暴露正在创建中的bean来实现） spring容器对单例bean进行缓存，可以提前暴露****

表示通过setter注入方式构成的循环依赖。对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入）的bean来完成的，而且只能解决单例作用域的bean循环依赖。通过提前暴露一个单例工厂方法，从而使其他bean能引用到该bean

****3、prototype范围的依赖处理（无法解决，抛出异常：****BeanCurrentlyInCreationException****）****

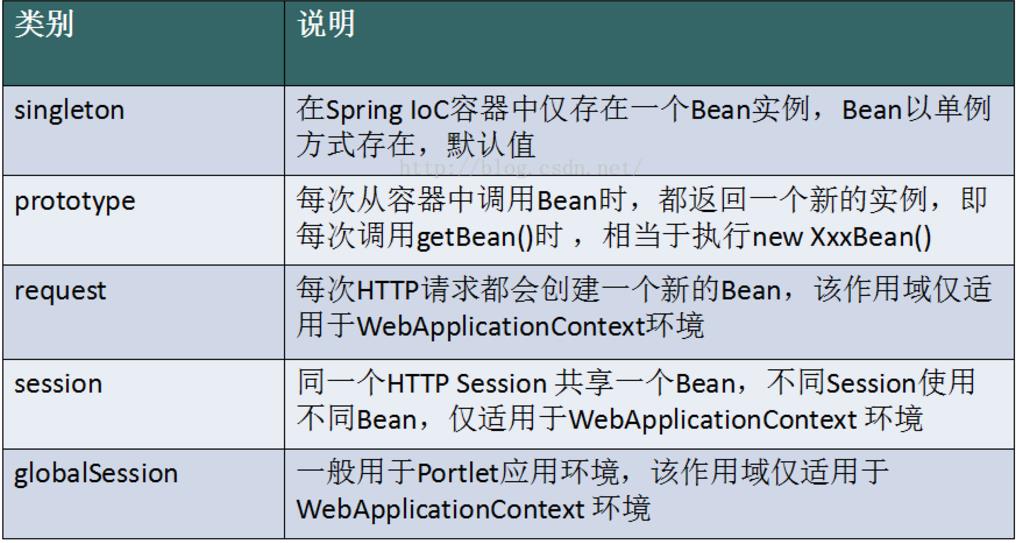
对于"prototype"作用域bean，Spring容器无法完成依赖注入，因为Spring容器不进行缓存"prototype"作用域的bean，因此无法提前暴露一个创建中的bean。

****Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。****

**earlySingletonObjects是为了解决循环依赖设置的，储存的是提前暴露的bean name –> bean instance，而singletonObjects存储的是完全实例化的bean name –> bean instance。**

**Spring中Bean的作用范围：**

**5种，singleton、prototype、request、session、globalSession**

****

**有状态的bean和无状态的bean的区别（单例无状态bean）**

有状态会话bean ：每个用户有自己**特有的**一个实例，在用户的生存期内，bean保持了用户的信息，即“有状态”；

[无状态](http://zhidao.baidu.com/search?word=%E6%97%A0%E7%8A%B6%E6%80%81&fr=qb_search_exp&ie=utf8" \t "http://blog.sina.com.cn/s/_blank)会话bean ：bean一旦实例化就被加进会话池中，各个用户都可以共用，

其中bean作用域中的singleton是无状态的作用域,prototype是有状态的作用域

**Servlet的生命周期**

****（1）实例化（创建一个Servlet的实例）****

当Servlet容器启动或客户端发送一个请求时，Servlet容器会创建一个Servlet实例。

****（2）初始化（调用init方式，进行初始化）****

实例化后，Servlet容器将调用Servlet的init()方法进行初始化（一些准备工作或资源预加载工作）。

****（3）响应请求，提供服务（调用doGet、doPost方式，响应请求、提供服务）****

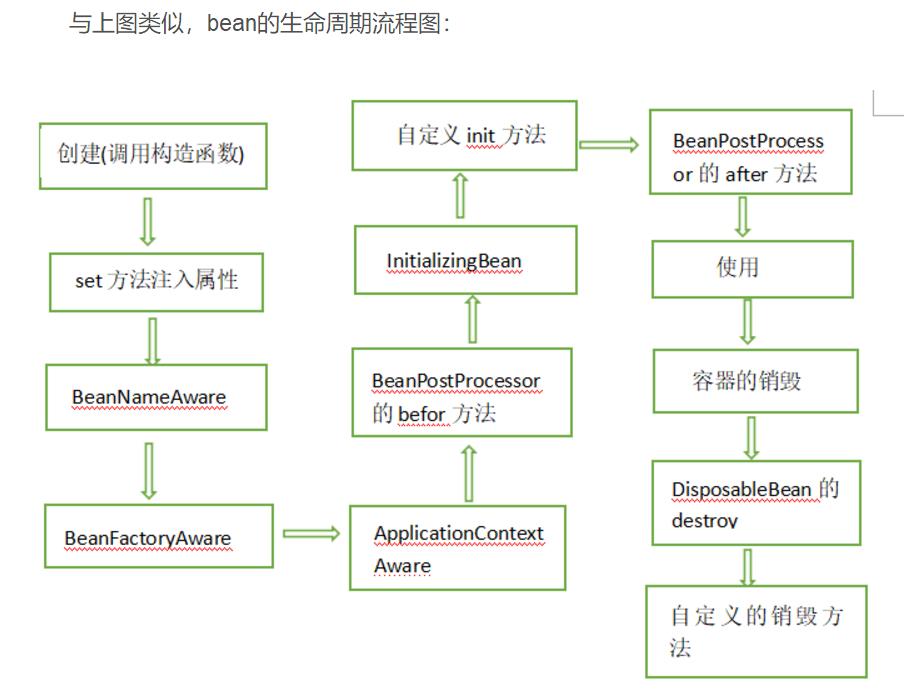
初始化后，当接收到客户端请求时，调用doGet和doPost的方法处理客户端请求，进行服务。

****（4）销毁（调用destroy方式，释放资源）****

当Servlet容器关闭时，Servlet容器会调用Servlet 的destroy()方法释放Servlet（或回收资源）。

**总结：实例化、初始化、响应请求，提供服务、销毁servlet，回收资源**

**SpringBean的生命周期**

****

**Bean实例生命周期的执行过程如下：**

**1）Spring对bean进行实例化，默认bean是单例；（实例化三种方式）**

**2）Spring对bean进行依赖注入；（依赖注入，三种方式）**

**3）如果bean实现了BeanNameAware接口，spring将调用setBeanName()方法，将bean的id传进来；**

**4）如果bean实现了BeanFactoryAware接口，spring将调用setBeanFactory方法，将BeanFactory实例传进来；**

**5）如果bean实现ApplicationContextAware接口，spring将调用setApplicationContext()方法，将ApplicationContext的实例传进来；**

**6）如果bean实现了BeanPostProcessor接口，spring将调用postProcessBeforeInitialization方法；**

**7）如果bean实现了InitializingBean接口，spring将调用init-method方法，该方法也会被调用；**

**8）如果bean实现了BeanPostProcessor接口，spring将调用postProcessAfterInitialization方法；**

**9）此时bean已经准备就绪，可以被应用程序使用了，他们将一直驻留在应用上下文中，直到该应用上下文被销毁；**

**10）若bean实现了DisposableBean接口，spring将调用它的distroy()接口方法。**

**总结：1、实例化bean 2、对bean依赖注入 3、实现beanNameAwere接口，调用setBeanName方法 4、实现beanFactoryAwere接口，调用setBeanfactory方法 5、如果实现applicationContextAwere接口，调用setApplicationContext方式6、实现beanPostprocessor接口，则调用postProcessorInitialictionBefore方法7、实现initializingBean接口，调用init方法8、实现beanPostprocessor接口，则调用postProcessorInitialictionAfter方法9、bean准备就绪 10、调用destroy方式销毁bean**

**Spring之Annotation**

常用的有如下几个标注：

**@Component:**标注一个普通的Spring Bean类 **@Controller:**标注一个控制器组件类

**@Service:**标注一个业务逻辑组件类 **@Repository:**标注一个DAO组件类

所以用@Component标注。另外如果一个Bean实例需要注入另一个Bean实例，则需要使用@Resource来注释，**@Resource有一个name属性**，Spring用这个值标记需要被注入的Bean实例的id，其实@Resource与<property.../>元素的ref属性有异曲同工之妙。之后Spring4.0的自动装配**@Autowired和@Qulified**可以实现同样的功能。

另外，1.如果要指定Bean的作用域，则用@Scope修饰Bean类，默认是singleton，也可将value值定为"prototype"。2.若要定制生命周期行为，则可使用**@PostConstruct和@PreDestroy**来修饰，前者修饰Bean的初始化方法，后者修饰的方法是Bean销毁之前的方法。

**@PostConstruct和@PreDestroy，**前者修饰Bean的初始化方法，后者修饰的方法是Bean销毁之前的方法。

springmvc的请求流程？

restful风格怎么获取URL中的数据，使用哪个注解

@ResponseBody、**@PathVariable**

AOP哪些术语，切入点这个点是指什么？切面@Aspect、切点@PointCut、通知@Advice

AOP：在运行时，动态地将代码切入到类的指定方法、指定位置的编程思想

**下面开始定义切面，先了解下其基本知识要点**

       \* 通知（Advice） ：“切面”对于某个“连接点”所产生的动作，例如，TestAspect中对com.spring.service包下所有类的方法进行日志记录的动作就是一个Advice。其中，一个“切面”可以包含多个“Advice”，例如ServiceAspect。  
      \* 切入点（Pointcut） ：匹配连接点的断言，在AOP中通知和一个切入点表达式关联。例如，TestAspect中的所有通知所关注的连接点，都由切入点表达式execution(\* com.spring.service.impl..\*.\*(..))来决定。   
      \* 目标对象（Target Object） ：被一个或者多个切面所通知的对象。例如，AServcieImpl和BServiceImpl，在实际运行时，Spring AOP采用代理实现，实际AOP操作的是TargetObject的代理对象。   
      \* AOP代理（AOP Proxy） ：在Spring AOP中有两种代理方式，JDK动态代理和CGLIB代理。**默认情况下，TargetObject实现了接口时，则采用JDK动态代理**，例如，AServiceImpl；反之，采用CGLIB代理，例如，BServiceImpl。强制使用CGLIB代理需要将 <aop:config>的 proxy-target-class属性设为true 。

    通知（Advice）类型 5类通知

      \*前置通知（Before advice） ：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。

      \*后通知（After advice） ：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

      \*返回后通知（After return advice） ：在某连接点正常完成后执行的通知，不包括抛出异常的情况。

      \*环绕通知（Around advice） ：贯穿切点执行前后；包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。

      \*抛出异常后通知（After throwing advice） ： 在方法抛出异常退出时执行的通知。

定义一个接口，并编写实现类：

    public interface UserService {  
         public void addUser(String userName, String password);  
    }

    @Service("userService")  
    public class UserServiceImpl implements UserService {  
  
       @Override  
        public void addUser(String userName, String password) {        
             System.out.println("--------------User addUser-------------");  
       }

   }

**定义切面类**

@Aspect//定义切面 = **切点+通知**@Component//声明这是一个组件  
public class Interceptor {  
    private final static Log log = LogFactory.getLog(Interceptor.class);  
    /\*\*  
     \* 这句话是方法切入点  
     \* 1 execution (\* com.wangku.spring.service.impl..\*.\*(..))  
     \* 2 execution ： 表示执行  
     \* 3 第一个\*号 : 表示返回值类型， \*可以是任意类型  
     \* 4 com.spring.service.impl : 代表扫描的包  
     \* 5 .. : 代表其底下的子包也进行拦截    
     \* 6 第二个\*号 : 代表对哪个类进行拦截，\*代表所有类    
     \* 7 第三个\*号 : 代表方法  \*代表任意方法  
     \* 8 (..) : 代表方法的参数有无都可以  
     \*/  
    //配置切入点,该方法无方法体,主要为方便同类中其他方法使用此处配置的切入点  
    @Pointcut("execution (\* com.gcx.service.impl..\*.\*(..))")  
    private void aspect() { //目标方法  
             System.out.println("============进入aspect方法==============");  
    }  
              
    //配置环绕通知,在切入点aspect()上面使用环绕通知 贯穿切点执行前后  
    @Around("aspect()") 贯穿整个目标方法  
    public void around(JoinPoint joinPoint){  
        long start = System.currentTimeMillis();  
        try {  
            ((ProceedingJoinPoint) joinPoint).proceed();  
            long end = System.currentTimeMillis();  
            if(log.isInfoEnabled()){  
                log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms!");//这里顺便记录下执行速度，可以作为监控  
            }  
        } catch (Throwable e) {  
            long end = System.currentTimeMillis();  
            if(log.isInfoEnabled()){  
                log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms with exception : " + e.getMessage());  
            }  
        }  
    }  
     //前置通知等可以没有JoinPoint参数  
    @Before("aspect()")  
    public void doBefore(JoinPoint joinPoint) {  
        System.out.println("==========执行前置通知===============");  
        if(log.isInfoEnabled()){  
            log.info("before " + joinPoint);  
        }  
    }         
    //配置后置通知,使用在方法aspect()上注册的切入点  
    @After("aspect()")  
    public void doAfter(JoinPoint joinPoint) {  
        System.out.println("===========执行后置通知==============");  
        if(log.isInfoEnabled()){  
            log.info("after " + joinPoint);  
        }  
    }     
    //配置后置返回通知,使用在方法aspect()上注册的切入点  
    @AfterReturning("aspect()") 正确执行完后执行  
    public void afterReturn(JoinPoint joinPoint){

            System.out.println("===========执行后置返回通知==============");  
            if(log.isInfoEnabled()){  
                log.info("afterReturn " + joinPoint);  
            }  
    }    
    //配置抛出异常后通知,使用在方法aspect()上注册的切入点，抛出异常后执行该方法  
    @AfterThrowing(pointcut="aspect()", throwing="ex")  
    public void afterThrow(JoinPoint joinPoint, Exception ex){  
            if(log.isInfoEnabled()){  
                log.info("afterThrow " + joinPoint + "\t" + ex.getMessage());  
            }  
    }      
}  
测试类

@Test  
    public void testAOP1(){  
        //启动Spring容器          
        ApplicationContext ctx = new ClassPathXmlApplicationContext(new String[]{"classpath:applicationContext-mvc.xml","classpath:applicationContext-dataSource.xml"});  
        UserService userService = (UserService) ctx.getBean("userService");  
        userService.addUser("zhangsan", "123456");  
    }

@Aspect切面 @PointCut切点 、 @Around、@Before、@After、@AfterReturing、@AfterThrowing通知

**总结：1、切面=切点+通知 2、目标方法 切点 切点表达式 3、5大通知：@Before @After @Around @AfterReturning @AfterThrowing，指定目标方法 4、AOP思想：在运行的时候，动态地将通用的代码切入到类的指定方法、指定位置的编程思想**

redis持久化的方式？（面试官说还有一个虚拟内存的方式）

1. RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。；2）AOF持久化以日志的形式记录服务器所处理的每一个写、删除和更新操作到文件中去，查询操作不会记录，当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。
2. RDB持久化方式：如果redis意外down掉的话，就会丢失最后一次快照后的所有修改；AOF则不会

redis数据淘汰策略？ 6种

voltile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰  
 volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰  
 volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰  
 allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰  
 allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰  
 no-enviction（驱逐）：禁止驱逐数据

tomcat的配置可以配置哪些信息？

linux查看端口使用哪个命令？

Ps -aux | grep

Lsof -i: port|grep

# [String 堆内存和栈内存](https://www.cnblogs.com/kenshinobiy/p/4651990.html)

java的堆是一个运行时数据区，类的对象从中分配空间。java的垃圾回收器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

栈的优势是，存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。栈中主要存放一些基本类型的变量（byte、short、int、long、float、double、char、boolean）和对象句柄（引用变量）。

栈有一个很重要的特殊性，就是存在栈中的数据可以共享。

Int a = 3;

Int b = 3;

a = 4;

编译器会先处理int a = 3;，首先它会在栈中创建一个变量为a 的引用，然后查找栈中是否有3 这个值，如果没找到，就将3存放进来，然后将a 指向3 。接着处理int b = 3; ，在创建完 b 的引用变量后，因为在栈中已经有3 这个值，便将b 直接指向3 。 这样，就出现了 a 与 b 同时均指向 3 的情况。这时，如果再令 a = 4;，那么编译器会重新搜索栈中是否有 4 这个值，如果没有，就将4 存放进来，并令a 指向 4；如果已经有了，则直接将 a 指向这个地址。 因此a 值的改变不会影响到 b 值。 要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的，因为这种情况 a 的修改并不会影响到 b，它是由编译器完成的，它有利于节省空间。而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

例子：

String a = "abc";

String b = "abc";

String c = new String("abc");

**过程：1、在常量池中查看是否有abc的字符串对象，有则返回，无则创建 2、在堆中创建一个new string（）的对象，并生成一个引用指向常量池中的abc对象 3、在栈中生成一个引用c，指向堆中的对象**

String d = c.intern(); **//直接在常量池中查找字符串对象，有则返回，无则添加**

这段代码（code）共创建了几个对象？2

public static void main(String[] args) {

String s1 = "hello";

String s2 = "hello";

String s3 = "he" + "llo";

String s4 = "hel" + new String("lo");

String s5 = new String("hello");

String s6 = s5.intern();

String s7 = "h";

String s8 = "ello";

String s9 = s7 + s8;

System.out.println(s1==s2);//true

System.out.println(s1==s3);//true

System.out.println(s1==s4);//false

System.out.println(s1==s9);//false

System.out.println(s4==s5);//false

System.out.println(s1==s6);//true

}

在jdk1.6,1.7,1.8下运行的结果为：

　　System.out.println(s1==s2);//true  
　　System.out.println(s1==s3);//true  
　　System.out.println(s1==s4);//false  
　　System.out.println(s1==s9);//false  
　　System.out.println(s4==s5);//false  
　　System.out.println(s1==s6);//true

**字符串常量池的设计思想 类似缓存，提供性能、减少内存开销**

JVM为了**提高性能和减少内存开销，**在实例化字符串常量的时候进行了一些优化

1）为字符串开辟一个字符串常量池，类似于缓存区

2）创建字符串常量时，首先坚持字符串常量池是否存在该字符串

3）存在该字符串，返回引用实例，不存在，实例化该字符串并放入池中

实现的基础：实现该优化的基础是因为字符串是不可变的，**可以不用担心数据冲突进行共享**

面试题：String str4 = new String(“abc”) 创建多少个对象？1或者2

1）在常量池中查找是否有“abc”对象

有则返回对应的引用实例

没有则创建对应的实例对象

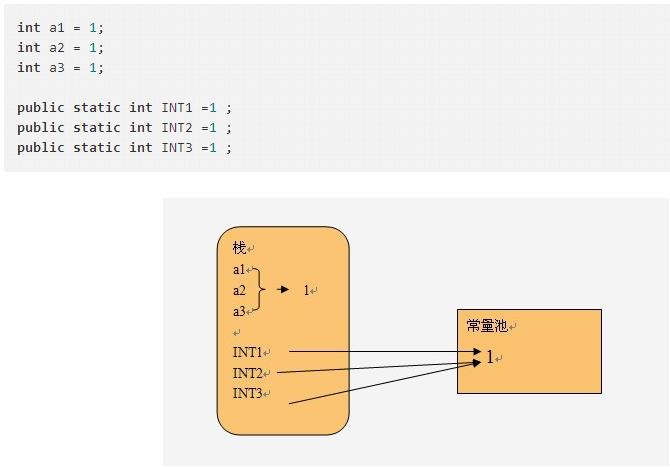
2）在堆中 new 一个 String("abc") 对象

3）将对象地址赋值给str4,创建一个引用

所以，常量池中没有“abc”字面量则创建两个对象，否则创建一个对象，以及创建一个引用

**常量池：类似缓存，提高性能，减少内存开销**

****String str1 = new String("A"+"B") ; 会创建多少个对象?**  1或者2**  
****String str2 = new String("ABC") + "ABC" ; 会创建多少个对象? 1个或者3****



**注意：方法区中存放了：字符串常量池、final常量、static静态变量、类的一些基本信息**

java.lang.String.intern()返回一个**常量池中的字符串**，就是一个在全局字符串池中有了一个入口。如果以前没有在全局字符串池中，那么它就会被添加到里面

 public class Test {

    public static void main(String[] args)

    {

        int a1=1;

        int b1=1;

        int c1=2;

        int d1=a1+b1;

        Integer a = 1;

        Integer b = 2;

        Integer c = 3;   //自动装箱

        Integer d = 3;

        Integer e = 321;    -128-127       ，直接在**IntegerCache**中取Integer对象，否则new一个新的对象

        Integer f = 321;

        Long g = 3L;

       System.out.println(a1==b1);   //true  结果1

       System.out.println(c1==d1);   //true  结果2

       System.out.println(c==d);   //true  结果3

       System.out.println(e==f);   //false  结果4

      }

  }

在自动装箱时，把int变成Integer的时候，是有规则的，当你的int的值在-128-IntegerCache.high(127) 时，返回的不是一个新new出来的Integer对象，而是一个已经缓存在堆 中的Integer对象，如果不在-128-IntegerCache.high(127) 时会返回一个新new出来的Integer对象。

当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并返回此 String 对象的引用。

 Public class StringTest {

   public static void main(String[] args) {

    String str1 = "abc";

    String str2 = new String("abc").intern();

    System.out.println(str1==str2);  //true

  }

}

# [正则表达式口诀](https://www.cnblogs.com/kenshinobiy/p/4809898.html)

正则其实也势利，削尖头来把钱揣； （指开始符号^和结尾符号$）   
特殊符号认不了，弄个倒杠来引路； （指\. \\*等特殊符号）   
倒杠后面跟小w， 数字字母来表示； （\w跟数字字母;\d跟数字）   
倒杠后面跟小d， 只有数字来表示；   
   
倒杠后面跟小s， 空格符号很重要；   
倒杠后面跟大W， 字母数字靠边站；   
倒杠后面跟大S， 空白也就靠边站；   
倒杠后面跟大D， 数字从此靠边站；   
   
单个字符要重复，三个符号来帮忙； （\* + ？）   
0 星加1 到无穷，问号只管0 和1； （\*表0-n;+表1-n;?表0-1次重复）   
花括号里学问多，重复操作能力强； （{n} {n,} {n,m}）   
若要重复字符串，园括把它括起来； （（abc）{3} 表示字符串“abc”重复3次 ）   
   
实在多得排不下，横杠请来帮个忙； （[1-5]）   
尖头放进中括号，反义定义威力大； （[^a]指除“a”外的任意字符 ）   
一些常用的正则表达式   
"^\d+$"　　//非负整数（正整数 + 0）   
"^[0-9]\*[1-9][0-9]\*$"　　//正整数   
"^((-\d+)|(0+))$"　　//非正整数（负整数 + 0）   
"^-[0-9]\*[1-9][0-9]\*$"　　//负整数   
"^-?\d+$"　　　　//整数 

"^[A-Za-z]+$"　　//由26个英文字母组成的字符串   
"^[A-Z]+$"　　//由26个英文字母的大写组成的字符串   
"^[a-z]+$"　　//由26个英文字母的小写组成的字符串   
"^[A-Za-z0-9]+$"　　//由数字和26个英文字母组成的字符串   
"^\w+$"　　//由数字、26个英文字母或者下划线组成的字符串 

匹配首尾空格的正则表达式：(^\s\*)|(\s\*$)   
匹配Email地址的正则表达式：\w+([-+.]\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*   
匹配网址URL的正则表达式：^[a-zA-z]+://([://w+(-//w+)\*)(//.(//w+(-//w+)\*))\*(//?\\S\*)?$](http://www.cnblogs.com/peida/admin/file://w+(-//w+)*)(//.(//w+(-//w+)*))*(//?\\\\S*)?$)   
匹配帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：**^[a-zA-Z][a-zA-Z0-9\_]{4,15}$**  
匹配国内电话号码：(\d{3}-|\d{4}-)?**(\d{8}|\d{7})?**   
匹配腾讯QQ号：^[1-9]\*[1-9][0-9]\*$

HTTP1.0定义了三种请求方法： **GET, POST 和 HEAD方法**。

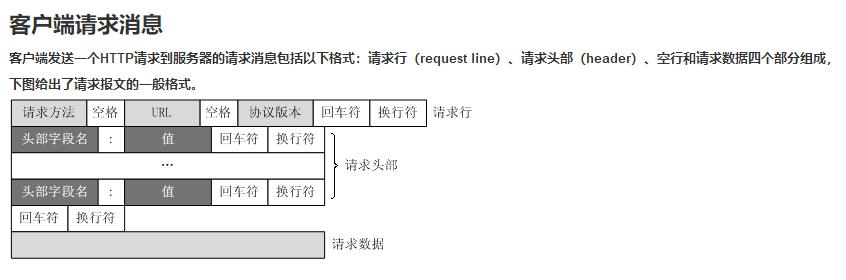
HTTP1.1新增了五种请求方法：**OPTIONS, PUT, DELETE, TRACE 和 CONNECT** 方法。

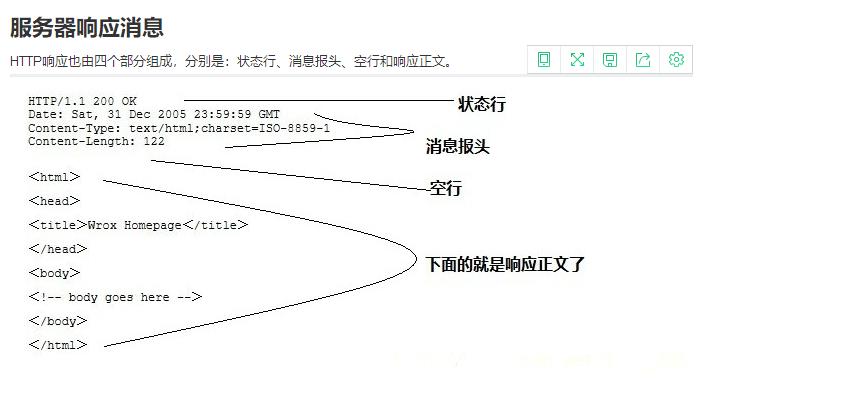
|  |  |  |
| --- | --- | --- |
| **序号** | **方法** | **描述** |
| 1 | GET | 请求指定的页面信息，并返回响应内容。 |
| 2 | HEAD | 类似于get请求，只不过返回的响应中没有具体的内容，主要用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求（例如**提交表单或者上传文件**）。数据被包含在请求体中。 |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| 5 | DELETE | 请求服务器删除指定的页面。 |
| 6 | CONNECT | HTTP/1.1协议中**预留给能够将连接改为管道方式的代理服务器**。 |
| 7 | OPTIONS | 查看服务器的性能，能够支持哪些方法等。 |
| 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断。 |

总结：http1.0 get（请求指定路径的页面信息，并返回响应内容）、post（向指定资源提交数据，比如提交表单，上传文件）、head（类似get请求，但是响应内容不包含具体的内容，主要是获取报头）

Delete：请求服务器删除指定的页面 options：查看服务器的性能，支持哪些方法

Trace：查看服务器收到哪些请求，主要用于诊断和测试





总结：

请求报文组成：请求行（方法、url、http版本）、请求头部、空行、请求数据

响应报文组成：状态行（http版本、状态码）、响应报头、空行、响应数据

总结：

1. spring创建bean三种方式（构造函数、静态工厂的方式、实例工厂方式）
2. Spring依赖注入三种方式（构造注入，set注入、注解注入）
3. Spring的set注入和构造注入区别（当参数比较多或者依赖关系比较复杂的时候，组好使用set注入，并且set注入可以解决单例作用域下的bean的循环依赖问题）
4. Bean的循环依赖问题，单例作用域，是通过“提前暴露正在创建中的bean，来实现的”
5. Spring的IOC（控制反转，IOC容器负责创建bean和管理bean）和AOP（运行的时候，动态地将通用的代码切入到指定的类的指定方法和指定位置上，日志、性能、事务）
6. IOC容器，hashMap，第一种，对于XML配置文件，加载xml文件，并读取所有的bean标签，通过反射，实例化，将实例化对象放入到hashMap中去，并提供getBean方法，第二种，通过properties配置文件，加载， ；； ；；最后通过静态工厂方式，去决定实例化哪一个配置文件对应的实例。。。
7. AOP中主要就是“切面、切点、5大通知、目标方法”，主要是在目标方式执行前后，加入一些操作
8. Spring用到哪些设计模式：AOP（JDK动态代理、Cglib动态代理）IOC（单例、静态工厂模式），策略模式，模板方法模式（JDBCTemplate），观察者模式（监听器）责任链模式（filter）