**一、性能测试的概念**  
　　性能测试是通过自动化的测试工具模拟多种正常峰值及异常负载条件来对系统的各项性能指标进行测试。[**负载测试**](javascript:;)和[**压力测试**](javascript:;)都属于性能测试，两者可以结合进行。  
　　通过负载测试，确定在各种工作负载下系统的性能，目标是当负载逐渐增加时，测试系统各项性能指标的变化情况。压力测试时通过确定一个系统的瓶颈或者不能接受的  
　　性能点，来获取系统能提供的最大服务级别的测试。性能测试主要包括负载测试、强度测试、容量测试。  
**二、性能测试的指标**  
　　web服务器：  
　　Avg Rps: 平均每秒的响应次数 = 总请求数 /秒数;  
　　Avg time to last byte per terstion(mstes): 平均每秒业务脚本的迭代次数;  
　　Successful Rounds: 成功的请求;  
　　Failed Rounds: 失败的请求;  
　　Successful Hits: 成功的点击次数;  
　　Failed Hits: 失败的点击次数;  
　　Hits Per Second: 每秒点击次数;  
　　Successful Hits Per Second:每秒成功的点击次数;  
　　Failed Hits Per Second: 每秒失败的点击次数;  
　　Attempted Connections: 尝试连接数;  
　　Throughput: 吞吐率;  
　　数据库服务器：  
　　User Connections: 用户连接数，也就是数据库的连接数量;  
　　Number of Deadlocks: 数据库死锁;  
　　Butter Cache Hit: 数据库Cache 的命中情况;  
**三、性能测试的流程**  
　　1.明确性能测试需求;  
　　2.制定性能测试方案;  
　　2.1.测试范围  
　　2.2.入口标准  
　　2.3.出口标准  
　　2.4.测试策略（测试环境指标、存量数据、业务场景、测试通过标准等）  
　　2.5.测试风险  
　　2.6.测试资源          
　　3.设计性能测试用例;  
　　4.执行性能测试用例;  
　　5.分析性能测试结果;  
　　6.生成性能测试报告;  
**四、性能测试的工具--JMeter**  
　　为什么是JMeter而不是LoadRunner呢  1.更少的投入，针对有限的测试成本；  2.开源工具的可定制性无可比拟；  3.通过社区得到最大程度的支持。  
　　JMeter是Apache组织开发的基于Java的压力测试工具。最初被设计用于web应用的测试，后来扩展到其他测试领域。可用于测试静态和动态资源，如文件、Java服务  
　　程序、Java对象、数据库等。JMeter能够对应用程序做功能/回归测试，通过创建带有断言的脚本来验证被测程序返回了期望的结果。而且为了保证最大限度的灵活性，  
　　JMeter允许使用正则表达式创建断言。  
**五、JMeter的特性**  
　　1.支持对多种服务类型进行测试;  
　　2.支持通过录制/回访方式获取测试脚本;  
　　3.具备高可移植性，是纯Java 程序;  
　　4.采用多线程框架，允许通过多个线程并发取样及通过独立的线程组对不同的功能同时取样;  
　　5.精心设计的GUI支持高速用户操作和精确计时;  
　　6.支持缓存和离线的方式分析/回放测试结果;  
　　7.高扩展性;  
**六、JMeter常用测试元件**  
　　1.线程组  
　　用来管理执行性能测试所需的JMeter线程。  
　　a.可以设置线程数量  
　　b.设置线程启动周期  
　　c.设置执行测试脚本的循环次数  
　　2.控制器  
　　JMeter有两种类型的控制器：采样器和逻辑控制器。  
　　采样器被用来向服务器发送请求。JMeter采样器包含：FTP Request、HTTP Request、JDBC Request等。  
　　逻辑控制器用来控制JMeter的测试逻辑，特别是何时发送请求。  
　　3.监听器  
　　监听器提供了对JMeter在测试期间收集到的信息的访问方法。  
　　4.定时器  
　　JMeter线程在发送请求之间没有间歇，通过添加定时器，设定请求之间应该间隔的时间。  
　　5.断言  
　　可以使用断言来检查从服务器获得的响应内容。  
　　6.配置元件  
　　配置元件与采样器紧密关联。虽然配置元件并不发送请求，但可添加或修改请求。  
　　7.前置处理器  
　　会在采样器发出请求之前做一些操作。  
　　8.后置处理器  
　　会在采样器发出请求之后做一些操作。  
　　JMeter执行顺序：配置元件=》前置处理器=》定时器=》采样器=》后置处理器=》断言=》监听器  
**七、辅助测试工具开发**  
　　下面的代码（工具:sqlexec）是一个用来向数据库（目前支持Oracle、MySQL）插入测试数据的工具。支持多线程，可插入千万级别测试数据。在后续压测中会用到该  
　　工具，工具开发尽量简单，一个工具只完成一个任务，同时不要重复制造轮子。  
　　[java] view plain copy  
　　package d706;   
　　/\*  
　　\* sql处理  
　　\*/   
　　public class Test\_DB\_Insert extends Thread{   
　　public static String SQLTEXT = null;  // 待处理的sql语句   
　　private InputStream ins = null;       // 用于读取配置文件    
　　private Properties property = new Properties(); // 读取数据库配置文件   
　　private String databaseType = null; // 数据库连接类型   
　　private String driver = null;       // 数据库驱动   
　　private String url = null;          // 数据库连接   
　　private String uName = null;        // 数据库登录用户名   
　　private String pwd = null;          // 数据库登录用户密码   
　　private  int numOfTestRecords;      // 插入数据条数   
　　private  Connection con = null;     // 连接数据库   
　　private  PreparedStatement statement = null;  // 获取数据库操作对象   
　　public  Test\_DB\_Insert(String sql){   
　　SQLTEXT = sql;                  // sql语句以参数的形式，在构造实例的时候传入   
　　}   
　　private void init(){   // 初始化配置文件   
　　try{   
　　ins = new FileInputStream("./d706/dbconf.properties");    
　　}catch(FileNotFoundException ffe){   
　　ffe.printStackTrace();   
　　}   
　　try{   
　　property.load(ins);   //   
　　}catch(IOException ie){   
　　ie.printStackTrace();   
　　}   
　　databaseType = property.getProperty("databasetype"); // 获取配置文件中设置的连接数据库类型   
　　if(databaseType.toUpperCase().equals("MYSQL")){      // 判断连接数据库类型   
　　driver = property.getProperty("driver\_mysql");   
　　url = property.getProperty("url\_mysql");   
　　uName = property.getProperty("db\_userName\_mysql"); // 连接数据库的用户信息;   
　　pwd = property.getProperty("db\_pwd\_mysql");   
　　}else if(databaseType.toLowerCase().equals("oracle")){ //   
　　driver = property.getProperty("driver\_oracle");   
　　url = property.getProperty("url\_oracle");   
　　uName = property.getProperty("db\_userName\_oracle");   
　　pwd = property.getProperty("db\_pwd\_oracle");   
　　}   
　　}                  
　　private  synchronized  void Insert\_DB(){     
　　try {   
　　try {   
　　Class.forName( driver );  // 注册驱动;   
　　}catch(ClassNotFoundException cf){   
　　cf.printStackTrace();   
　　}   
　　con = DriverManager.getConnection(url，uName， pwd); // 获取数据库连接   
　　con.setAutoCommit(false);  // 关闭事务自动提交   
　　SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss:SS");  // 记录执行时间   
　　TimeZone t = sdf.getTimeZone();     
　　t.setRawOffset(0);     
　　sdf.setTimeZone(t);     
　　Long startTime = System.currentTimeMillis();     
　　System.out.println("插入数据操作开始...");   
　　statement = con.prepareStatement(SQLTEXT); //创建数据库操作对象   
　　/\*  
　　\* "INSERT INTO TEST\_DB(name，sex，nickname，test1，test2，test3，test4，" +  
　　"test5，test6，test7，test8，test9，test10，test11，test12，test13，test14，" +  
　　"test15，test16，test17，test18，test19，test20，test21，test22，test23，" +  
　　"test24，test25，test26，test27，test28，test29，test30，test31，test32，" +  
　　"test33，test34，test35，test36，test37，test38，test39，test40，test41，" +  
　　"test42) VALUES(?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，" +  
　　"?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?，?)"  
　　\*/   
　　numOfTestRecords = 1000;                               //插入的测试数据量;   
　　for(int i = 0; i<numOfTestRecords; i++) {              //循环   
　　statement.setString(i + 1， "DBTest-" + i);    
　　//statement.setString(2， "" + i%2);          //0表示男 1表示女    
　　statement.addBatch();  // 把一个SQL命令加入命令列表     
　　//statement.executeUpdate(); //执行SQL;   
　　}   
　　statement.executeBatch(); //执行批量更新   
　　con.commit();//语句执行完毕，提交事务                     
　　//int[] ref = statement.executeBatch();                  
　　//if(ref[numOfTestRecords-1] == 0){System.out.println("插入数据操作完成");} //   
　　System.out.println("插入数据操作完成");   
　　Long endTime = System.currentTimeMillis();     
　　System.out.println("插入"+numOfTestRecords+"条数据，"+"用时(时：分：秒：毫秒)" +    
　　sdf.format(new Date(endTime - startTime))); //   
　　}catch(Exception e) {    
　　System.out.println("异常: " + e.toString());   
　　e.printStackTrace();    
　　}finally{   
　　if(statement != null){   // 关闭数据库操作对象                          
　　try{   
　　statement.close();   
　　}catch(SQLException se){   
　　se.printStackTrace();   
　　}   
　　}   
　　if(con != null){         // 关闭数据库连接                        
　　try{   
　　if(con.isClosed()){con.close();}     
　　}catch(SQLException se){   
　　se.printStackTrace();   
　　}   
　　}   
　　}          
　　}   
　　@Override   
　　public void run() {    // 类外调用    
　　Test\_DB\_Insert ti = new Test\_DB\_Insert(SQLTEXT); // 构造实例   
　　ti.init();      // 初始化   
　　ti.Insert\_DB(); // 执行插入数据   
　　}   
　　//  public static void main(String[] args){   
　　//                     
　　//          Test\_DB\_Insert ti = new Test\_DB\_Insert(SQLTEXT);   
　　//          ti.init();  //初始化   
　　//          ti.Insert\_DB();  //执行插入数据   
　　//  }   
　　}   
　　// 针对增删查改，可放到一个SQL处理类（Test\_DB\_crud）中，判断传入的SQL字符串，然后交给对应的方法去执行并在控制台输出结果。 在Test\_DB\_Control类

　　中只是new一个Test\_DB\_crud类，形成了Test\_DB\_crud对Test\_DB\_Control的依赖关系。  
　　[java] view plain copy  
　　package d706;   
　　/\*  
　　\*程序界面  
　　\*按钮事件  
　　\*/   
　　public class Test\_DB\_gui extends JFrame implements ActionListener{   
　　private static final long serialVersionUID = 1L;   
　　public static String SQLTEXT = null; // 界面输入的sql文本   
　　private JScrollPane js = null;   
　　private JPanel jp1 = null;   
　　private JTextArea ta = null;   
　　private JButton jb = new JButton();   
　　public Test\_DB\_gui(){   
　　this.setTitle("sqlExecV1.0");   
　　ta = new JTextArea();   
　　ta.setText("");   
　　js = new JScrollPane(ta);   
　　jp1 = new JPanel();   
　　jp1.setLayout(new BorderLayout());   
　　jp1.add(js，BorderLayout.CENTER);   
　　jb = new JButton();   
　　jb.setText("执行");   
　　jb.addActionListener(this); // 添加监听   
　　jp1.add(jb，BorderLayout.SOUTH);   
　　this.getContentPane().add(jp1);   
　　this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);   
　　this.setBounds(400，200，700，500);   
　　this.setVisible(true);         
　　jb.addActionListener(new ActionListener(){  // 按钮事件   
　　public void  actionPerformed(ActionEvent e) {  // 匿名内部类的形式实现按钮事件   
　　SQLTEXT = ta.getText(); // 将要执行的SQL设置成静态的   
　　Multi\_process mp = new Multi\_process(SQLTEXT); // 在构造实例时，传入sql   
　　mp.run(); // 运行程序   
　　//System.out.println( mp.SQLTEXT +"\*\*\*"); // 控制台输出   
　　}      
　　});   
　　}   
　　public void  actionPerformed(ActionEvent e) {    
　　// 实现 ActionListener 接口，需要实现的方法   
　　}   
　　public void run() {  // 运行方法   
　　Test\_DB\_gui np =  new Test\_DB\_gui();   
　　System.out.println( np.getTitle() );    
　　}   
　　}   
　　[java] view plain copy  
　　public class Multi\_process {   
　　/\*  
　　\*多线程  
　　\*/   
　　public static String SQLTEXT = null;  // 待处理的sql语句   
　　public Multi\_process(String sql){   
　　SQLTEXT = sql;                    // sql语句以参数的形式，在构造实例的时候传入   
　　}   
　　public  void run(){   
　　Test\_DB\_Insert  td1 = new Test\_DB\_Insert(SQLTEXT); // 创建实例   
　　//     Test\_DB\_Insert  td2 = new Test\_DB\_Insert(SQLTEXT);   
　　Thread t1 = td1;  // 创建线程   
　　//         Thread t2 = td2;   
　　t1.start(); // 启动线程    
　　//         t2.start();   
　　//     Test\_DB\_Insert t3 = new Test\_DB\_Insert();   
　　//     Test\_DB\_Insert t4 = new Test\_DB\_Insert();   
　　//     t3.run();   
　　//     t4.run();   
　　}   
　　}   
　　[java] view plain copy  
　　public class Test\_Exec {   
　　/\*  
　　\* 程序入口  
　　\*/   
　　public static void main(String[] args){   
　　Test\_DB\_gui np = new Test\_DB\_gui();   
　　np.run();   
　　}   
　　}   
　　在Linux下执行，需打成jar包，通过shell脚本执行。以下是sqlexec的startup.sh执行脚本.  
　　[plain] view plain copy  
　　#!/bin/sh   
　　#   
　　#Author: bruce   
　　#Version: sqlExecv1.0   
　　#Date:2013-11-20   
　　#   
　　read -p "please input jar file path:" jarpath   
　　java -jar "$jarpath"   
　　echo "running sqlExec."   
**注：**  
　　今天突然想到测试数据的一个问题。即造出来的数据应该更接近真实，而不是都一样。所以想利用配置文件的方式，在配置文件中写好SQL，程序读取执行配置文件并动态生成SQL，完成插入测试数据。这样就解决了测试数据不够真实，可能影响测试结果的问题。