知识汇总

目录

[知识汇总 1](#_Toc21779247)

[1 JAVA技术 3](#_Toc21779248)

[1.1 Servlet 3](#_Toc21779249)

[1.2 JavaEE 4](#_Toc21779250)

[1.3 Java注解 8](#_Toc21779251)

[1.4 Lambda表达式 9](#_Toc21779252)

[1.5 函数式接口 10](#_Toc21779253)

[1.6 集合(Collection) 10](#_Toc21779254)

[1.7 多线程 17](#_Toc21779255)

[1.8 内存模型 30](#_Toc21779256)

[1.9 BlockingQueue使用 30](#_Toc21779257)

[2.0 Java基础知识 31](#_Toc21779258)

[2.0.27 Java (关键字) 66](#_Toc21779259)

[3 框架技术 67](#_Toc21779260)

[3.1 Spring 67](#_Toc21779261)

[3.2 AOP 67](#_Toc21779262)

[4 Web应用 69](#_Toc21779263)

[4.1 REST 69](#_Toc21779264)

[5 JAVA远程通讯 69](#_Toc21779265)

[5.1 远程方法调用(RMI) 69](#_Toc21779266)

[5.2 远程过程调用(RPC) 72](#_Toc21779267)

[5.3 Java消息服务(JMS) 72](#_Toc21779268)

[6 数据结构 77](#_Toc21779269)

[6.1 排序 77](#_Toc21779270)

[6.2 查找 77](#_Toc21779271)

[6.3 逻辑数据结构 80](#_Toc21779272)

[7 前端相关 85](#_Toc21779273)

[7.1 JSP 85](#_Toc21779274)

[7.2 HTML 86](#_Toc21779275)

[7.3 Ajax 87](#_Toc21779276)

[7.4 JavaScript 87](#_Toc21779277)

[8 服务器技术 87](#_Toc21779278)

[WebSphere 87](#_Toc21779279)

[Tomcat 87](#_Toc21779280)

[知识点 87](#_Toc21779281)

[负载均衡 87](#_Toc21779282)

[代理服务器 87](#_Toc21779283)

[9 Linux知识 88](#_Toc21779284)

[9.1 Linux标准输入、标准输出、标准错误与重定向介绍 88](#_Toc21779285)

[9.2 常用命令 90](#_Toc21779286)

[9.3上传文件至linux 98](#_Toc21779287)

[9.4 安装工具 98](#_Toc21779288)

[10 分布式知识 99](#_Toc21779289)

[10.1 分布式缓存 99](#_Toc21779290)

[负载均衡(nginx,读音:engine) 104](#_Toc21779291)

[反向代理 104](#_Toc21779292)

[静态内容缓存 104](#_Toc21779293)

[访问控制 104](#_Toc21779294)

[11 网络编程(计算机网络) 104](#_Toc21779295)

[11.1 TCP/IP协议 104](#_Toc21779296)

[11.2 OSI模型 105](#_Toc21779297)

[11.3 TCP/IP与OSI比较 106](#_Toc21779298)

[11.4 socket编程 106](#_Toc21779299)

[12 前沿技术 106](#_Toc21779300)

[12.1 区块链 106](#_Toc21779301)

[人工智能 110](#_Toc21779302)

[13 测试 110](#_Toc21779303)

[13.1 单元测试(JUnit) 110](#_Toc21779304)

[14 数据库技术 111](#_Toc21779305)

[14.1存储过程 111](#_Toc21779306)

[14.2 函数 118](#_Toc21779307)

[14.3 乐观锁及悲观锁 119](#_Toc21779308)

[14.4 索引 120](#_Toc21779309)

[14.5 数据库优化 120](#_Toc21779310)

[15 设计模式 121](#_Toc21779311)

[16 网络安全 124](#_Toc21779312)

[16.1 公钥密码体制 124](#_Toc21779313)

[16.2 对称加密算法 124](#_Toc21779314)

[16.3数字证书 125](#_Toc21779315)

# 1 JAVA技术

## 1.1 Servlet

全称Java Servlet，未有中文译文。是用Java编写的服务器端程序。狭义的Servlet是指Java语言实现的一个接口，广义的Servlet是指任何实现了这个Servlet接口的类。

### 1.1.1工作模式

客户端发送请求至服务器；

服务器启动并调用Servlet，Servlet根据客户端请求生成响应内容并将其传给服务器；

服务器将响应返回客户端；

其他；

### 1.1.2 HttpServlet

javax.servlet.http.HttpServlet实现了专门用于响应HTTP请求的Servlet，提供了响应对应HTTP标准请求的doGet()、doPost()等方法。

### 1.1.3 Servlet生命周期及工作原理

#### (1)生命周期

Servlet生命周期分为三个阶段：

1.初始化阶段  调用init()方法；

2.响应客户请求阶段　　调用service()方法；

3.终止阶段　　调用destroy()方法；

Servlet初始化阶段：

在下列时刻Servlet容器装载Servlet：

1.Servlet容器启动时自动装载某些Servlet，实现它只需要在web.XML文件中的<Servlet></Servlet>之间添加如下代码；

2.在Servlet容器启动后，客户首次向Servlet发送请求；

3.Servlet类文件被更新后，重新装载Servlet；

Servlet被装载后，Servlet容器创建一个Servlet实例并且调用Servlet的init()方法进行初始化。在Servlet的整个生命周期内，init()方法只被调用一次。

#### (2)Servlet工作原理

首先简单解释一下Servlet接收和响应客户请求的过程，首先客户发送一个请求，Servlet是调用service()方法对请求进行响应的，通过源代码可见，service()方法中对请求的方式进行了匹配，选择调用doGet,doPost等这些方法，然后再进入对应的方法中调用逻辑层的方法，实现对客户的响应。在Servlet接口和GenericServlet中是没有doGet,doPost等等这些方法的，HttpServlet中定义了这些方法，但是都是返回error信息，所以，我们每次定义一个Servlet的时候，都必须实现doGet或doPost等这些方法。

每一个自定义的Servlet都必须实现Servlet的接口，Servlet接口中定义了五个方法，其中比较重要的三个方法涉及到Servlet的生命周期，分别是上文提到的init(),service(),destroy()方法。GenericServlet是一个通用的，不特定于任何协议的Servlet,它实现了Servlet接口。而HttpServlet继承于GenericServlet，因此HttpServlet也实现了Servlet接口。所以我们定义Servlet的时候只需要继承HttpServlet即可。

Servlet接口和GenericServlet是不特定于任何协议的，而HttpServlet是特定于HTTP协议的类，所以HttpServlet中实现了service()方法，并将请求ServletRequest,ServletResponse强转为HttpRequest和HttpResponse。

## 1.2 JavaEE

### 1.2.1 为什么需要JavaEE

我们编写的JSP代码中，由于大量的显示代码和业务逻辑混淆在一起，彼此嵌套，不利于程序的维护和扩展。当业务需求发生变化的时候，对于程序员和美工都是一个很重的负担。

为了程序的易维护性和可扩展性，这就需要我们使用JavaEE技术来进行项目开发。

### 1.2.2 什么是JavaEE

JavaEE是一个开发分布式企业级应用的规范和标准。

Java语言的平台有3个版本：适用于小型设备和智能卡的JavaME（Java Platform Micro Edition，Java微型版）、适用于桌面系统的JavaSE（Java Platform Micro Edition，Java标准版）、适用于企业应用的JavaEE（Java Platform Enterprise Edition，Java企业版）。

### 1.2.3 JavaEE包括的技术

JavaEE应用程序是由组件构成的。J2EE组件是具有独立功能的单元，他们通过相关的类和文件组装成JavaEE应用程序，并与其他组件交互。

JavaEE包括的技术有：Web Service、Struts、Hibernate、Spring、JSP、Servlet、JSF、EJB、JavaBean、JDBC、JNDI、XML、JavaSE。

### 1.2.4 什么框架

框架即Framework，我们可以把框架理解为某种应用的半成品，就是一组组件，供用户选用，完成用户自己的系统。简单地说就是使用别人准备好的组件，来实施自己的项目。

### 1.2.5 JavaEE与框架的关系

框架技术是在JavaEE的基础上形成的，而应用程序是在框架的基础上创建的。



### 1.2.6 JavaEE的体系结构

应用程序的三层结构如下：

表示层：由用户界面和用户生成界面的代码组成。

中间层：包含系统的业务和功能代码。

数据层：负责完成存取数据库的数据和对数据进行封装。

三层体系结构的优点：一个组件的更改不会影响其他两个组件。例如：如果用户需要更换数据库，那么只有数据层组件需要修改代码。同样，如果更改了用户界面设计，那么只有表示层组件需要修改。

由于表示层和数据层相互独立，因而可以方便地扩充表示层，使系统具有良好的可扩展性。

代码重复减少，因为在3个组件之间尽可能地共享代码。

良好的分工与协作。这将使不同的小组能够独立地开发应用程序的不同部分，并充分发挥各自的长处和优势。

### 1.2.7 JavaEE技术内容

JavaEE技术体系结构可分为：表示层技术、中间层技术、数据层技术。JavaEE还涉及到系统集成的一些技术。

1、表示层技术

        HTML

        JavaScript

        Ajax

Ajax是几种技术的整合。Ajax的主要功能是异步地向服务器端发送请求，处理数据或者根据返回的数据重新显示页面。

2、中间层技术

       JSP

       显示动态内容的服务器网页。

       Servlet

       接收客户端请求，并做出响应的Java程序。

       Servlet是中间层技术的重要组成部分，它控制着其他的组件。

       JSTL

       辅助JSP显示动态内容的标准标签库。

       JavaBean

       JavaEE的模型组件。

       中间层的框架技术

       Struts框架：Struts主要是扩展了Servlet。

3、数据层技术

       JDBC技术

       JDBC（Java Database Connectivity）：Java数据库连接。

       使用JDBC操作数据库中的表和数据。

       数据层框架技术

       数据层框架，Hibernate提供了以对象的形式操作关系型数据库数据的功能。

4、系统集成技术

在一些大型项目中，整个应用程序被分成若干个模块，由不同的开发团队同步进行开发，有可能他们使用的技术也是各不相同的，在每个功能模块开发结束后，所要做的事情就是把这些零散的模块，整合成一个整体，这才是我们也是客户所要的东西，这时我们就要用到有关系统集成的技术（Web Service）。

JAX-WS技术

JAX-WS（Java API for XML Web Service）是JavaEE平台的重要组成部分。

JAX-WS简化了使用Java技术开发Web服务的工作。

JNDI

JNDI（Java Naming and Directory Interface，Java命名和目录接口）是一组在Java应用中访问命名和目录服务的API，命名服务把对象和名称联系在一起，并且可以通过名称找到相应的对象。

### HTTP（HyperText Transfer Protocol，超文本传输协议）协议

HTTP是一个无状态协议，它基于客户端/服务器模型。HTTP客户端（如浏览器）需要与服务器建立一个连接，并将一则请求消息通过连接发送到HTTP服务器，以请求相应的资源。然后服务器返回带有请求资源的响应消息。当响应结束后，就断开连接，彼此并不保存连接信息。所以HTTP是无状态协议。

HTTP使用端口发送和接收消息。端口是协议发送和接收数据的信道或机制。80端口是为HTTP开发的，主要用于在WWW（World Wide Web，万维网）服务上传输信息。浏览网页服务默认的端口号是80。

### 使用MyEclipse提供的数据监控工具TCP/IP Monitor来进行数据监控。

TCP/IP Monitor（TCP/IP 监控器）的配置步骤：

Window（窗口）--->Preferences（首选项）--->Run Debug（运行/调式）--->TCP/IP Monitor（TCP/IP 监视器）

注意：如果没有找到，则按以下路径查找：

Window（窗口）--->Preferences（首选项）--->MyEclipse ---> Internet Tools（因特网工具）---> TCP/IP Monitor（TCP/IP 监视器）

然后选中Show the TCP/IP Monitor view there is activity（显示TCP/IP监视器视图，用来查看什么时候有请求和响应）。

单击“Add”按钮，弹出New Monitor窗口。

Local monitoring Port：填写8088。本机监控端口，也就是Monitor实际监听的端口。

Host name：填写localhost或127.0.0.1。欲监控的主机名称，监听对象的设置。

Port：填写8080。欲监控的端口号，你想要监听的端口号，你的请求发送到Local monitoring Port后会转到这个端口。

Type：选择HTTP。要监控的类型。

Timeout：0 。

单击确定按钮，然后单击Start（启动）按钮启动监视器。

然后启动tomcat，在地址栏中输入：<http://localhost:8088/>项目名（或上下文路径）。在MyEclipse中会出现“TCP/IP Monitor”窗口，可以看到监听的数据了。

可以单击窗口右上角的ValidateWS-I Message Log File图标（图标上有个勾）添加记录文件。

下面对某一请求示例进行分析：

客户端发送的消息为文本流，由以下内容组成：

请求行：包括方法、统一资源标识符（URL）和HTTP协议版本。

头信息：典型的请求消息类似于以下内容。

              GET /monitor/test.jsp?username=xinxin&userpassword=1234 HTTP/1.1

              Host: localhost:8080

              User-Agent: Mozilla/5.0(Windows;U;Window NT 5.1;zh-CN;)

              Accept: text/xml,application/xml, text/html; text/plain;

第一行使用GET方法提交请求数据，并指定正在使用的是HTTP协议1.1版本。后续行包括头信息，User-Agent头指示客户端使用的浏览器类型是Mozilla，Accept头信息提供有关客户端可以接受的MIME类型列表。在头信息之后，客户端发送一个空白行，指示请求消息结束。请求消息由服务器进一步处理，并且生成相应的响应消息。

       下面是响应示例：

       HTTP/1.1 200 OK

       Server: JavaWebServer

       Last-modified: Tuesday, 21-Aug-07 11:12:10 GMT

       Content-length: 100

       Content-type: text/html

第一行为状态行。状态行中的状态码200指示已成功处理请求，因此描述为“OK”，404是找不到被请求的网页时遇到的状态码。其他行是头信息，Server头指定服务器上使用的软件，Last-modified头指示被请求文件的最后修改日期，Content-length头指定文件的大小（以字节表示），Content-type头指定文档的MIME类型。例如，text/plain指示未格式化的文本文档。text/html指示格式化的Web文档。

### get和post发送方式

post：post方式会将表单的内容通过http发送，在地址栏中看不到表单的提交信息，比较安全，而且使用post方式发送信息没有字符长度的限制。

get方式：表单内容经过编码之后，通过URL发送，可以在地址栏中看到表单信息，不安全。使用get方式发送信息时有255个字符的限制。get请求的页面可以设置为书签或使用邮件发送，而post不可以。

## 1.3 Java注解

注解：Java代码提供元数据，描述数据的数据(标签)；

#### (1) 基本内置注解

(1)@Override:对覆盖超类中方法的方法进行标记，如果被标记的方法并没有实际覆盖超类中的方法，则编译器会发出错误警告。

(2)@Deprecated:过时代码注解，若使用，则警告；

(3)@SuppressWarnings:去处警告,可传参；

#### (2) 元注解

注解到注解上的注解,主要是应用在注解声明；

(1)@Retention:保留期，注解存活时间；

- RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时它将被丢弃忽视。

- RetentionPolicy.CLASS 注解只被保留到编译进行的时候，它并不会被加载到 JVM 中。

- RetentionPolicy.RUNTIME 注解可以保留到程序运行的时候，它会被加载进入到 JVM 中，所以在程序运行时可以获取到它们。

(2)@Documented 将注解中的元素包含到 Javadoc 中去;

(3)@Target 指定应用场景

参数为枚举ElementType。若应用错误，编译报错；

(4)@Inherited 继承 ，若超类的注解声明包含，则子类默认继承超类的的该注解；

(5)@Repeatable 可重复注解。jdk1.8开始使用 其余1.5后

#### (3)注解的属性

成员变量。只有变量，没有方法。通过实例来说明。

#### (5) java预置注解

(1)@Deprecated 过时代码(方法、类)；

(2)@Override

(3)@SuppressWarnings

(4)@SafeVarargs

(5)@FunctionalInterface 函数式接口，jdk1.8后加入；

## 1.4 Lambda表达式

通过表达式来代替功能接口。和方法一样,它提供了一个正常的参数列表和一个使用这些参数的主体(body,可以是一个表达式或一个代码块。

语法：

// 1. 不需要参数,返回值为 5

() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值

x -> 2 \* x

// 3. 接受2个参数(数字),并返回他们的差值

(x, y) -> x – y

// 4. 接收2个int型整数,返回他们的和

(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)

(String s) -> System.out.print(s)

具体例子，详看：LambdasTest.java

## 1.5 函数式接口

Java 8对一类特殊类型的接口的称呼。 这类接口只定义了唯一的抽象方法的接口（除了隐含的Object对象的公共方法）。

用例详见FunctionInterfaceTest.java

java.lang.Runnable,

java.awt.event.ActionListener,

java.util.Comparator,

java.util.concurrent.Callable

## 1.6 集合(Collection)

java集合(说到底就是一种数据结构或者容器)。如图：



集合的定义: 一组[类](https://en.wikipedia.org/wiki/Class_(computer_science))和[接口](https://en.wikipedia.org/wiki/Interface_(java))实现普遍可重复使用的收集[的数据结构](https://en.wikipedia.org/wiki/Data_structure)。

Java集合大致可以分为Set、List、Queue和Map四种体系，其中Set代表无序、不可重复的集合；List代表有序、重复的集合；而Map则代表具有映射关系的集合

集合和数组的区别：

1.数组长度在初始化时指定，意味着只能保存定长的数据。而集合可以保存数量不确定的数据。同时可以保存具有映射关系的数据（即关联数组，键值对 key-value）。

2.数组元素即可以是基本类型的值，也可以是对象。集合里只能保存对象（实际上只是保存对象的引用变量），基本数据类型的变量要转换成对应的包装类才能放入集合类中。

Java的集合类主要由两个接口派生而出：Collection和Map,Collection和Map是Java集合框架的根接口。

### 1.6.1 Set集合

一个不包含重复元素的 collection。更确切地讲，set 不包含满足 e1.equals(e2) 的元素对 e1 和 e2，并且最多包含一个 null 元素。

Set集合中包含了三个比较重要的实现类:HashSet、TreeSet和EnumSet。

#### HashSet

HashSet是Set接口的典型实现，实现了Set接口中的所有方法，并没有添加额外的方法，大多数时候使用Set集合时就是使用这个实现类。HashSet按Hash算法来存储集合中的元素。因此具有很好的存取和查找性能。

1.不能保证元素的排列顺序，顺序可能与添加顺序不同，顺序也有可能发生变化。

2.HashSet不是同步的，如果多个线程同时访问一个HashSet，则必须通过代码来保证其同步。

3.集合元素值可以是null。除此之外，HashSet判断两个元素是否相等的标准也是其一大特点。HashSet集合判断两个元素相等的标准是两个对象通过equals()方法比较相等，并且两个对象的hashCode()方法返回值也相等。

#### (2) TreeSet

TreeSet是SortedSet接口的实现类，正如SortedSet名字所暗示的，TreeSet可以确保集合元素处于排序状态。此外，TreeSet还提供了几个额外的方法。

TreeSet的排序方式:

1. TreeSet中所谓的有序，不同于之前所讲的插入顺序，而是通过集合中元素属性进行排序方式来实现的。
2. TreeSet支持两种排序方法：自然排序和定制排序。在默认情况下，TreeSet采用自然排序。

#### (4) EnumSet

EnumSet是一个专为枚举类设计的集合类，EnumSet中的所有元素都必须是指定枚举类型的枚举值，该枚举类型在创建EnumSet时显示或隐式地指定。EnumSet的集合元素也是有序的，EnumSet以枚举值在EnumSet类内的定义顺序来决定集合元素的顺序。

EnumSet特点

1.EnumSet集合不允许加入null元素。EnumSet中的所有元素都必须是指定枚举类型的枚举值。

2.EnumSet类没有暴露任何构造器来创建该类的实例，程序应该通过它提供的类方法来创建EnumSet对象。

EnumSet没有其他额外增加的方法，只是增加了一些创建EnumSet对象的方法。

注：HashSet类中包含HashMap实例，该实例是[transient关键字修饰。该关键字作用为防止该字段被序列话，只允许在](https://www.cnblogs.com/huzi007/p/6600559.html)调用者的内存中而不会写到磁盘里持久化。

#### (5) HashSet、TreeSet和EnumSet的性能对比

EnumSet性能>HashSet性能>LinkedHashSet>TreeSet性能

EnumSet内部以位向量的形式存储，结构紧凑、高效，且只存储枚举类的枚举值，所以最高效。HashSet以hash算法进行位置存储，特别适合用于添加、查询操作。LinkedHashSet由于要维护链表，性能比HashSet差点，但是有了链表，LinkedHashSet更适合于插入、删除以及遍历操作。而TreeSet需要额外的红黑树算法来维护集合的次序，性能最次。

但是具体使用要考虑具体的使用场景。

当需要一个特定排序的集合时，使用TreeSet集合。  
当需要保存枚举类的枚举值时，使用EnumSet集合。

当经常使用添加、查询操作时，使用HashSet。

当经常插入排序或使用删除、插入及遍历操作时，使用LinkedHashSet。

### 1.6.2 List集合

List集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。

List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素 。List集合默认按元素的添加顺序设置元素的索引，例如第一个添加的元素索引为0，第二个添加的元素索引为1......

List作为Collection接口的子接口，可以使用Collection接口里的全部方法。而且由于List是有序集合，因此List集合里增加了一些根据索引来操作集合元素的方法。

List典型的两个实现[ArrayList](mk:@MSITStore:G:\帮助文档API\JDK1.6%20API帮助文档.CHM::/java/util/ArrayList.html)、Vector.

#### ArrayList

*/\*\*  
 \* 模拟ArrayList  
 \** ***@param <E>*** *\*/***public class** SimulateArrayList<E> {  
 **transient** Object[] **elementData**;  
 **private int size**;  
 */\*\*  
 \* 构造函数  
 \** ***@param initialCapacity*** *数组长度  
 \*/* **public** SimulateArrayList(**int** initialCapacity) {  
 **if** (initialCapacity > 0) {  
 **this**.**elementData** = **new** Object[initialCapacity];  
 } **else if** (initialCapacity == 0) {  
 **this**.**elementData** = **new** Object[10];  
 } **else** {  
 **throw new** IllegalArgumentException(**"Illegal Capacity: "** +  
 initialCapacity);  
 }  
 }  
 */\*\*  
 \* 默认构造，构造数组长度10  
 \*/* **public** SimulateArrayList() {  
 **this**(10);  
 }  
 */\*\*  
 \* 返回数组长度  
 \** ***@return*** *\*/* **public int** size() {  
 **return size**;  
 }  
 */\*\*  
 \* 判断是否非空  
 \** ***@return*** *\*/* **public boolean** isEmpty() {  
 **return size** == 0;  
 }  
 */\*\*  
 \* 判断是否包含  
 \** ***@param o*** *\** ***@return*** *\*/* **public boolean** contains(Object o) {  
 **return** indexOf(o) >= 0;  
 }  
 */\*\*  
 \* 获取所在数组下标，不包含返回-1  
 \** ***@param o*** *\** ***@return*** *\*/* **public int** indexOf(Object o) {  
 **if** (o == **null**) {  
 **for** (**int** i = 0; i < **size**; i++)  
 **if** (**elementData**[i] == **null**)  
 **return** i;  
 } **else** {  
 **for** (**int** i = 0; i < **size**; i++)  
 **if** (o.equals(**elementData**[i]))  
 **return** i;  
 }  
 **return** -1;  
 }  
 */\*\*  
 \* 获取所在下标元素  
 \** ***@param index*** *\** ***@return*** *\*/* E elementData(**int** index) {  
 **return** (E) **elementData**[index];  
 }  
  
 */\*\*  
 \* 添加元素  
 \** ***@param element*** *\*/* **public void** add(E element) {  
 ensureCapacity();  
 **elementData**[**size**++] = element;  
 }  
  
 */\*\*  
 \* 添加元素至指定下标  
 \** ***@param index*** *\** ***@param object*** *\*/* **public void** add(**int** index, Object object) {  
 rangeCheck(index);  
 ensureCapacity();  
 System.*arraycopy*(**elementData**, index, **elementData**, index + 1, **size** - index);  
 **elementData**[index] = object;  
 **size**++;  
 }  
 */\*\*  
 \* 检验数组下标是否越界  
 \*/* **private void** rangeCheck(**int** index) {  
 **if** (index < 0 || index >= **size**) {  
 **try** {  
 **throw new** Exception(**"下标越界"**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 */\*\*  
 \* 数组扩容和数据拷贝  
 \*/* **private void** ensureCapacity() {  
 **if** (**size** == **elementData**.**length**) {  
 Object[] newArray = **new** Object[**size** \* 2 + 1];  
 System.*arraycopy*(**elementData**, 0, newArray, 0, **elementData**.**length**);  
 **elementData** = newArray;  
 }  
 }  
  
 **public static void** main(String[] args) {  
 SimulateArrayList list = **new** SimulateArrayList();  
 list.add(**"a"**);  
 list.add(**"b"**);  
 list.add(**"c"**);  
 System.***out***.println(**"数组长度:"** + list.size());  
 System.***out***.println(**"数组1："** + list.**elementData**[0]);  
 System.***out***.println(**"数组2："** + list.**elementData**[1]);  
 System.***out***.println(**"数组3："** + list.**elementData**[2]);  
 System.***out***.println(**"list是否包含字符串'b':"** + list.contains(**"b"**) + **";下标:"** + list.indexOf(**"b"**));  
 list.add(1, **"afds"**);  
 System.***out***.println(**"数组长度:"** + list.size());  
 System.***out***.println(**"数组1："** + list.**elementData**[0]);  
 System.***out***.println(**"数组2："** + list.**elementData**[1]);  
 System.***out***.println(**"数组3："** + list.**elementData**[2]);  
 System.***out***.println(**"数组4："** + list.**elementData**[3]);  
 System.***out***.println(**"list是否包含字符串'b':"** + list.contains(**"b"**) + **";下标:"** + list.indexOf(**"b"**));  
 }  
}

1. Vector

1.ArrayList是线程不安全的，Vector是线程安全的。  
2.Vector的性能比ArrayList差。

### 1.6.3 Map集合

将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

## 1.7 多线程

### 1.7.1 Synchronized(同步代码块)

内置锁(相当于互斥锁):java对象都可以用做一个实现同步的锁。同步代码块包括两部分:一个作为锁的对象引用，一个作为由这个锁保护的代码块。

重入：实现方法:为每个锁关联一个获取计数值和一个所有者线程，当计数值为0时，认为没有被任何线程持有。当线程请求一个未被持有的锁时，JVM记下锁持有者，并且将获取计数器加1操作。如果同一个线程再次获取这个锁，计数值递增，当线程退出同步代码块时，计数器递减。为0时锁被释放。

Synchronized保证方法或者代码块以同步方式执行。并且Synchronized是支持可重入的。

当synchronized作用于普通方法是，锁对象是this；

当synchronized作用于静态方法是，锁对象是当前类的Class对象；

当synchronized作用于代码块时，锁对象是synchronized(obj)中的这个obj。

(注：内置锁由synchronized实现，内置锁对应的是显式锁(ReentrantLock)，显式锁实现主要是拥有非阻塞的获取锁、响应中断、超时机制、支持公平性设置等特性。

例如：

**import** java.util.HashMap;  
**import** java.util.Map;  
**import** java.util.concurrent.locks.ReentrantReadWriteLock;  
  
**public class** ReadWriteMap<K, V> {  
 **private** Map<K, V> **map**;  
 **private** ReentrantReadWriteLock **lock** = **new** ReentrantReadWriteLock();  
 **private** ReentrantReadWriteLock.ReadLock **readLock** = **lock**.readLock();  
 **private** ReentrantReadWriteLock.WriteLock **writeLock** = **lock**.writeLock();  
 **public** ReadWriteMap(Map<K, V> map){  
 **this**.**map** = map;  
 }  
 **public** V put(K key, V value){  
 **try**{  
 **writeLock**.lock();  
 System.***out***.println(**"put获取锁成功"**);  
 **try** {  
 Thread.*sleep*(10000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"put准备"**);  
 **return map**.put(key, value);  
 }**finally** {  
 System.***out***.println(**"put释放锁"**);  
 **writeLock**.unlock();  
 }  
 }  
  
 **public** V get(K key){  
 **try**{  
 **readLock**.lock();  
 System.***out***.println(**"get获取锁成功"**);  
 **try** {  
 Thread.*sleep*(5000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"get准备"**);  
 **return map**.get(key);  
 }**finally**{  
 System.***out***.println(**"get释放锁"**);  
 **readLock**.unlock();  
 }  
 }  
}

)

### 1.7.2 Volatile变量

(1)被volatile修饰的共享变量，保证了不同线程对这个变量进行操作(读写)时的可见性(该变量在主存中，一个线程修改了该值，其他线程读取时都能从内存中读取到新值)。

(2)禁止进行指令重排序。

应用场景：

条件：对变量的写操作不依赖于当前值；该变量没有包含在具有其他变量的不变式中；在访问变量时不需要加锁。

即确保他们自身状态的可见性，确保它们所引用对象的状态的可见性，以及标识一些重要的程序生命周期事件（例如初始化或关闭）

eg:

volatile boolean flag;

while(!flag){

...

}

有点难理解，哭。

happens-before原则：

在同一个线程中，书写在前面的操作happen-before后面的操作。

eg:

int a = 1;

int b = a; //此时的b肯定为1，但若声明a = 1;b = 2。两者无依赖关系，赋值顺序可能因为指令重排序导致不一样。

同一个锁的unlock操作happen-before此锁的lock操作。

public class Test{

private int i = 0;

private synchronized void increase(){

i ++;

}

private synchronized void decrease(){

i --;

}

}

对一个volatile变量的写操作happen-before对此变量的任意操作。

如果A操作 happen-before B操作，B操作happen-before C操作，那么A操作happen-before C操作。

### 1.7.3 Synchronized用法

#### (1) 修饰方法

A）public synchronized void add(){};

注：作用于同一个对象内，可防止多个线程访问这个对象的synchronizde方法(同一个实例对象，不同对象的sychronized方法互不干扰)，如果一个对象有多个synchronized方法，最多只能能有一个线程访问其中一个synchronized方法

若将方法声明为synchronized会影响效率，应为synchronized方法被线程锁定后，其他线程进入阻塞状态，直至该线程知识完毕，释放锁。

eg:

/\*\*

\* @Description TODO

\* @ClassName SycTest

\* @Author Administrator

\* @Data 2018/9/16 18:37

\* @Version 1.0

\*/

public class SycTest {

public synchronized void test(){

int num = 1;

while(num <= 10){

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("线程[" + Thread.currentThread() + "]：" + num);

num ++;

}

}

public static void main(String[] args) {

SycTest s = new SycTest();

Thread t1 = new Thread(){

@Override

public void run() {

s.test();

}

};

Thread t2 = new Thread(){

@Override

public void run() {

s.test();

}

};

t1.start();

t2.start();

}

}

输出:

线程[Thread[Thread-0,5,main]]：1

线程[Thread[Thread-0,5,main]]：2

线程[Thread[Thread-0,5,main]]：3

线程[Thread[Thread-0,5,main]]：4

线程[Thread[Thread-0,5,main]]：5

线程[Thread[Thread-0,5,main]]：6

线程[Thread[Thread-0,5,main]]：7

线程[Thread[Thread-0,5,main]]：8

线程[Thread[Thread-0,5,main]]：9

线程[Thread[Thread-0,5,main]]：10

线程[Thread[Thread-1,5,main]]：1

线程[Thread[Thread-1,5,main]]：2

线程[Thread[Thread-1,5,main]]：3

线程[Thread[Thread-1,5,main]]：4

线程[Thread[Thread-1,5,main]]：5

线程[Thread[Thread-1,5,main]]：6

线程[Thread[Thread-1,5,main]]：7

线程[Thread[Thread-1,5,main]]：8

线程[Thread[Thread-1,5,main]]：9

线程[Thread[Thread-1,5,main]]：10

B) public synchronized static void add(){};

注:作用于类范围，防止多个线程同时访问类内的public synchronized static方法。也就是说只要是这个类的对象实例都起到同步作用。

eg:

import java.text.SimpleDateFormat;

import java.util.Date;

/\*\*

\* @Description

\* @ClassName SycTest

\* @Author Administrator

\* @Data 2018/9/16 18:37

\* @Version 1.0

\*/

public class SycTest {

public synchronized static void test\_static(){

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");//设置日期格式

System.out.println("线程[" + Thread.currentThread() + "]:" + df.format(new Date()));

}

public static void main(String[] args) {

for(int i = 0; i < 10; i ++){

Thread t1 = new Thread() {

@Override

public void run() {

SycTest.test\_static();

}

};

t1.start();

}

}

}

输出:

线程[Thread[Thread-0,5,main]]:2018-09-16 22:23:55

线程[Thread[Thread-9,5,main]]:2018-09-16 22:24:00

线程[Thread[Thread-8,5,main]]:2018-09-16 22:24:05

线程[Thread[Thread-7,5,main]]:2018-09-16 22:24:10

线程[Thread[Thread-6,5,main]]:2018-09-16 22:24:15

线程[Thread[Thread-5,5,main]]:2018-09-16 22:24:20

线程[Thread[Thread-3,5,main]]:2018-09-16 22:24:25

线程[Thread[Thread-4,5,main]]:2018-09-16 22:24:30

线程[Thread[Thread-2,5,main]]:2018-09-16 22:24:35

线程[Thread[Thread-1,5,main]]:2018-09-16 22:24:40

#### (2)修饰代码块(比修饰方法的粒度更小)

A) public void add(){synchronized(this){}};

注:作用域为当前对象，表示只对这个区块的资源实行互斥访问，修饰代码块的时候需要一个reference对象作为锁的对象；

synchronized(this)锁的是当前对象，此时仅在有明确对象。否则可以创建一个特殊的instance变量充当锁，例如: private byte[] lock = new byte[0];public void add(){synchronized(lock){}};(注：零长度的byte数组对象创建起来将比任何对象都经济)

eg:

import java.text.SimpleDateFormat;

import java.util.Date;

/\*\*

\* @Description

\* @ClassName SycTest

\* @Author Administrator

\* @Data 2018/9/16 18:37

\* @Version 1.0

\*/

public class SycTest {

public void test(){

SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");//设置日期格式

System.out.println("test执行synchronized块初始化[" + Thread.currentThread() + "]:" + df.format(new Date()));

synchronized(this){

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("test执行synchronized块结束[" + Thread.currentThread() + "]:" + df.format(new Date()));

}

public static void main(String[] args) {

SycTest s = new SycTest();

for (int i = 0; i < 5; i ++){

Thread t = new Thread(){

@Override

public void run() {

s.test();

}

};

t.start();

}

}

}

输出:

test执行synchronized块初始化[Thread[Thread-0,5,main]]:2018-09-16 22:59:12

test执行synchronized块初始化[Thread[Thread-3,5,main]]:2018-09-16 22:59:12

test执行synchronized块初始化[Thread[Thread-2,5,main]]:2018-09-16 22:59:12

test执行synchronized块初始化[Thread[Thread-1,5,main]]:2018-09-16 22:59:12

test执行synchronized块初始化[Thread[Thread-4,5,main]]:2018-09-16 22:59:12

test执行synchronized块结束[Thread[Thread-0,5,main]]:2018-09-16 22:59:17

test执行synchronized块结束[Thread[Thread-4,5,main]]:2018-09-16 22:59:22

test执行synchronized块结束[Thread[Thread-1,5,main]]:2018-09-16 22:59:27

test执行synchronized块结束[Thread[Thread-2,5,main]]:2018-09-16 22:59:32

test执行synchronized块结束[Thread[Thread-3,5,main]]:2018-09-16 22:59:37

public void add(){synchronized(Counter.class){}};

eg:

/\*\*

\* @Description

\* @ClassName SycTest

\* @Author Administrator

\* @Data 2018/9/16 18:37

\* @Version 1.0

\*/

public class SycTest {

public void test(){

SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");//设置日期格式

System.out.println("test执行synchronized块初始化[" + Thread.currentThread() + "]:" + df.format(new Date()));

synchronized(SycTest.class){

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("test执行synchronized块结束[" + Thread.currentThread() + "]:" + df.format(new Date()));

}

public static void main(String[] args) {

for (int i = 0; i < 5; i ++){

SycTest s = new SycTest();

Thread t = new Thread(){

@Override

public void run() {

s.test();

}

};

t.start();

}

}

}

输出:

test执行synchronized块初始化[Thread[Thread-2,5,main]]:2018-09-16 23:12:39

test执行synchronized块初始化[Thread[Thread-3,5,main]]:2018-09-16 23:12:39

test执行synchronized块初始化[Thread[Thread-0,5,main]]:2018-09-16 23:12:39

test执行synchronized块初始化[Thread[Thread-4,5,main]]:2018-09-16 23:12:39

test执行synchronized块初始化[Thread[Thread-1,5,main]]:2018-09-16 23:12:39

test执行synchronized块结束[Thread[Thread-2,5,main]]:2018-09-16 23:12:44

test执行synchronized块结束[Thread[Thread-1,5,main]]:2018-09-16 23:12:49

test执行synchronized块结束[Thread[Thread-4,5,main]]:2018-09-16 23:12:54

test执行synchronized块结束[Thread[Thread-0,5,main]]:2018-09-16 23:12:59

test执行synchronized块结束[Thread[Thread-3,5,main]]:2018-09-16 23:13:04

注：class literal作为锁的时候，跟synchronized static方法一样，作用于该类的所有对象。

#### (3)了解内置锁及对象锁、类锁

1)内置锁：每个java对象都可以用做一个实现同步的锁，这些锁成为内置锁。内置锁是一个互斥锁，最多只有一个线程能够获取该锁。java所有实例对象都含义一个互斥锁，这个锁由jvm管理获取和释放。

注：获取内置锁的唯一途径就是进入该锁的同步代码块或方法。synchronized修饰方法或代码块

2)对象锁:作用上与内置锁类似，但是作用域是对象实例的方法或者一个对象实例上；不同对象的实例的对象锁是互不干扰的。

注：获取内置锁的唯一途径就是进入该锁的同步代码块或方法。synchronized修饰方法或代码块

形式1:public synchronized void add(){};

形式2:public void add(){synchronized(this){}};或

3)类锁:作用类的静态方法或者一个类class对象上；一个类的类锁只有一个。

synchronized 修饰静态的方法或代码块

形式1:public synchronized static void add();

形式2:public void add(){synchronized(Counter.class){}};

对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步。

1. 经测试:

A/B为测试synchronized(this)和synchronized(instance变量)。C/D为测试synchronized方法

1. 不同对象之间多个线程同时执行包含代码块synchronized(this)或synchronized(instance变量)方法，线程之间是执行是不受影响的；两个或两个以上线程执行包含代码块synchronized(this)或synchronized(instance变量)方法，同步块时同步执行的，只有一个线程获取锁(其他线程阻塞状态)，并且释放锁后其它线程才可逐一。说明: **synchronized(this)和synchronized(instance变量)为同一个对象。**
2. 若作用于为同一个对象，多线程既执行包含synchronized(this)的方法也执行包含synchronized(instance变量)的方法,若线程A执行synchronized(this)时，线程B执行synchronized(instance变量)，两者是不受锁影响，异步执行的；若线程A执行synchronized(this)时，线程B执行synchronized(this)，两者是受到锁制约的，等待一线程释放锁，另一线程才可继续执行；若线程A执行synchronized(instance变量)时，线程B执行synchronized(instance变量)，两者是受到锁制约的，等待一线程释放锁，另一线程才可继续执行。说明:对于同一个对象，若synchronized(this)执行于当前对象的作用域；若synchronized(instance变量)执行于instance变量对象域。说明：**synchronized(this)和synchronized(instance变量)作用域为同一对象，synchronized(this)锁的是this当前执行对象，synchronized(instance变量)锁的是instance变量。**
3. 不同对象之间多个线程同时执行synchronized修饰的方法，线程之间不受锁的制约，几乎同时执行synchronized修饰的方法；同一个对象，多个线程同步执行synchronized修饰的方法，线程受到锁(对象锁)的制约，一个线程只有等待另外一个线程释放锁之后才能获取锁，执行synchronized修饰的方法。说明：**synchronized修饰的方法作用域为同一对象**。
4. 两个或两个以上线程同时通过类名.method()执行静态方法,线程受到锁(类锁)的制约一个线程只有等待另外一个线程释放锁之后才能获取锁，执行该静态方法。说明：**synchronized修饰的方法作用域为类。**
5. 两个或两个以上线程同时通过实例对象.method()或类名.method()执行synchronized修饰的静态方法。假如线程A执行实例对象.method()时，线程B同时执行类名.method()，线程A、B受到类锁制约，B执行只有等待A释放锁才能够执行。说明：**实例对象.method()或类名.method()两者是互斥的，因为两者都为对象锁。**
6. 两个或两个以上线程同时通过实例对象.method()执行静态方法或实例对象.method()执行非静态方法。说明：**实例对象.method()执行静态方法为类锁，实例对象.method()为对象锁，两者非互斥，所以是同时执行的。**

### 1.7.4 ThreadLocal类

(1)提供线程局部变量。这些变量不同于它们的普通对应物，因为访问某个变量的每个线程都有自己的局部变量，它独立于变量的初始化副本。ThreadLocal实例通常是类中的private static字段，它们希望将状态与某一个线程相关联。

(2)使用场景：当有一些不是线程安全的对象，但是想避免同步对该对象的访问。

(3)ThreadLocal：线程间的数据隔离(提供变量的副本)，Synchronized：线程间的数据共享。

(4)主要源码导读：

public class ThreadLocal<T> {

/\*\*

\* 返回此线程局部变量的当前线程的“初始值”。线程第一次使用 get() 方法访问变量时将调用此方法，但如果线程之前调用了 set(T) 方法，则不会对该线程再调用 initialValue 方法。通常，此方法对每个线程最多调用一次，但如果在调用 get() 后又调用了 remove()，则可能再次调用此方法。

\* 该实现返回 null；如果程序员希望线程局部变量具有 null 以外的值，则必须为 ThreadLocal 创建子类，并重写此方法。通常将使用匿名内部类完成此操作

\* @return

\* 返回此线程局部变量的初始值

\* 匿名内部类方式:

\* private static final ThreadLocal < Integer > uniqueNum =

\* new ThreadLocal < Integer > () {

\* @Override protected Integer initialValue() {

\* return uniqueId.getAndIncrement();

\* }

\* };

\*

\*/

protected T initialValue() {

return null;

}

public ThreadLocal() {

}

/\*\*

\* 返回此线程局部变量的当前线程副本中的值。如果变量没有用于当前线程的值，则先将其初始化为调用 initialValue() 方法返回的值。

\* @return

\* 此线程局部变量的当前线程的值

\*/

public T get() {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null) {

ThreadLocalMap.Entry e = map.getEntry(this);

if (e != null) {

@SuppressWarnings("unchecked")

T result = (T)e.value;

return result;

}

}

return setInitialValue();

}

/\*\*

\* 将此线程局部变量的当前线程副本中的值设置为指定值。大部分子类不需要重写此方法，它们只依靠 initialValue() 方法来设置线程局部变量的值。

\* @return

\* value - 存储在此线程局部变量的当前线程副本中的值。

\*/

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

/\*\*

\* ThreadLocalMap是一个自定义的哈希映射，仅适用于维护线程本地值。不会在ThreadLocal类之外导出任何操作。

\* 该类是包私有的，允许在Thread类中声明字段。为了帮助处理非常大且长期使用的用法，哈希表条目使用WeakReferences作为键。

\* 但是，由于未使用引用队列，因此只有在表开始空间不足时才能保证删除过时条目

\*/

static class ThreadLocalMap {

static class Entry extends WeakReference<ThreadLocal<?>> {

Object value;

Entry(ThreadLocal<?> k, Object v) {

super(k);

value = v;

}

}

private Entry[] table;

}

}

public class Thread implements Runnable {

/\*

\* 与此线程有关的ThreadLocal值。此映射由ThreadLocal类维护

\*/

ThreadLocal.ThreadLocalMap threadLocals = null;

}

### 1.7.5 进程和线程的区别

进程是cpu资源分配的最小单位，线程是cpu调度的最小单位。

### 1.7.6 悲观锁与乐观锁

|  |  |  |
| --- | --- | --- |
|  | 乐观锁 | 悲观锁 |
| 含义 | 总是假设最好的情况，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，可以使用版本号机制和CAS算法实现 | 总是假设最坏的情况，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阻塞直到它拿到锁 |
| 使用场景 | 1、适用于多读的应用类型；  2、数据库的write\_condition机制；  3、java.util.concurrent.atomic的原子变量类(CAS算法)； | 1、数据库的锁机制，行锁、表锁、读锁、写锁等；  2、java中synchronized修饰的(静态)方法、块；  3、ReentrantLock独占锁； |
| 实现方式 | 1、版本号机制；  2、CAS算法； | 1、数据库使用select for update;  2、java synchronized 方法； |
| 优点 | 省去锁的开销，加大系统的吞吐量 | 保证操作独占性 |
| 缺点 | 若是读写情况比较多的情况下产生冲突，应用会不断重试，导致性能降低。  (1)ABA问题； | 影响性能 |

关于数据库悲观锁与乐观锁，详见[数据库技术-乐观锁及悲观锁。](#_乐观锁及悲观锁)

### 1.7.7 安全发布对象(对象构造)

(1)发布对象:对象能够在当前作用域之外的代码所使用。

eg:将对象的引用保存到一个公有的静态变量中，便于任何类及现场都能看见该对象。

(2)对象逸出:

当一个对象还没有构造完成时 , 就使得它被其他线程看见。

(3)解决办法(参照单例模式)：

A)静态初始化函数中初始化一个对象的引用;

B)将对象的引用保存到volatile类型域或者AtomicReference对象中;

C)将对象的引用保存到某个正确构造对象的 final类型域中;

D)将对象的引用保存到某个由锁保护的域中;

## 1.8 内存模型

详见<java内存模型.docx>。

## 1.9 BlockingQueue使用

阻塞队列。队列主要有两种方式，先进先出，后进先出。是Concurrent包提供的高效并且线程安全的队列类。

两种阻塞情况

（1）当队列对象中无数据时，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列

（2）当队列中的数据满时，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。

### 1.9.1 实现方式

BlockingQueue是一个定义阻塞的接口类，所有阻塞队列都需要实现该接口，它的实现类有ArrayBlockingQueue、DelayQueue、 LinkedBlockingDeque、LinkedBlockingQueue、PriorityBlockingQueue、SynchronousQueue等。BlockingQueue主要基于ReentrantLock（可重入锁，线程可以重复获取已经持有的锁，独占锁，同一时刻只有一个线程能获取到锁，所以在lock.lock()和lock.unlock()之间可能有一次释放锁的操作（同样也必然还有一次获取锁的操作）。在Quene类中，不管take还是put，在线程持有锁之后只有await()方法有可能释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁）。应用场景是生产者和消费者模式。例如线程池的阻塞队列，线程池为空时，消费者阻塞等待，直到有新的线程。线程池满时，生产者堵塞，直至队列中元素非满。

#### 了解LinkedBlockingQueue

LinkedBlockingQueue主要是一个基于链表的可选容量的阻塞队列。

指定容量，内部维持一个队列，所以有一个头节点head和一个尾节点last，内部维持两把锁，一个用于入队，一个用于出队，还有锁关联的Condition对象。

类变量属性

*/\*\*容量界限，默认Integer.MAX\_VALUE\*/***private final int capacity**;  
  
*/\*\* 当前元素个数 \*/***private final** AtomicInteger **count** = **new** AtomicInteger();  
  
*/\*\*  
 \* Head of linked list.  
 \* Invariant: head.item == null  
 \*/***transient** Node<E> **head**;  
  
*/\*\*  
 \* Tail of linked list.  
 \* Invariant: last.next == null  
 \*/***private transient** Node<E> **last**;  
  
*/\*\* Lock held by take, poll, etc \*/***private final** ReentrantLock **takeLock** = **new** ReentrantLock();  
  
*/\*\* Wait queue for waiting takes \*/***private final** Condition **notEmpty** = **takeLock**.newCondition();  
  
*/\*\* Lock held by put, offer, etc \*/***private final** ReentrantLock **putLock** = **new** ReentrantLock();  
  
*/\*\* Wait queue for waiting puts \*/***private final** Condition **notFull** = **putLock**.newCondition();

put方法

take方法

remove方法

## 2.0 Java基础知识

### 2.0.1 java三种移位运算符<<,>>,>>>

(1)<<: 左移运算符，num<<1,相当于num乘以2；

int i = 0x10; //0x代表后面为十六进制，二进制00010000，十进制的16；

System.out.println(i << 1); //左移一位，00100000，十进制的32

输出:32

(2)>>：右移运算符,num>>,相当于num除以2；

int i = 0xf; //0x代表后面为十六进制，二进制00001111，十进制的15；

System.out.println(i >> 1); //右移一位，00000111，十进制的7

输出:7

int i = -129;

/\*\*

\* -23右移五位计算:

\* 1、23的二进制码为:0000 0000 0000 0000 0000 0000 0001 0111;

\* 2、-23的二进制码为23的二进制码的反码并进行+1补码操作

\* 0000 0000 0000 0000 0000 0000 0001 0111

\* --->

\* 1111 1111 1111 1111 1111 1111 1110 1000

\* --->

\* 1111 1111 1111 1111 1111 1111 1110 1001

\* 3、右移五位得到

\* 0100 1111 1111 1111 1111 1111 1111 1111

\* <<以下为逆运算>>

\* 4、取反+1补码运算得到结果

\* 0100 1111 1111 1111 1111 1111 1111 1111

\* --->

\* 1011 0000 0000 0000 0000 0000 0000 0000

\* --->

\* 1011 0000 0000 0000 0000 0000 0000 0001

\* 5、0000 0001的十进制的负数即为结果 为-1

\*/

/\*\*

\* -129右移五位计算:

\* 1、129的二进制码为:0000 0000 0000 0000 0000 0000 1000 0001;

\* 2、-129的二进制码为129的二进制码的反码并进行+1补码操作

\* 0000 0000 0000 0000 0000 0000 1000 0001

\* --->

\* 1111 1111 1111 1111 1111 1111 0111 1110

\* --->

\* 1111 1111 1111 1111 1111 1111 0111 1111

\* 3、右移五位得到

\* 1111 1111 1111 1111 1111 1111 1111 1011

\* <<以下为逆运算>>

\* 4、取反+1补码运算得到结果

\* 1111 1111 1111 1111 1111 1111 1111 1011

\* --->

\* 0000 0000 0000 0000 0000 0000 0000 0100

\* --->

\* 1011 0000 0000 0000 0000 0000 0000 0101

\* 5、0000 0101的十进制的负数即为结果 为-5

\*/

System.out.println(i >> 5);

输出：-5

(3)>>>：无符号右移，忽略符号位，空位都以0补齐

略.

### 2.0.2 java 运算符(位运算符) 与(&)、非(~)、或(|)、异或(^)

int a = 129;

int b = 148;

/\*\*

\* a 二进制码 0000 0000 0000 0000 0000 0000 1000 0001

\* b 二进制码 0000 0000 0000 0000 0000 0000 1001 0100

\* a&b {如果相对应位都是1，则结果为1，否则为0}

\* 0000 0000 0000 0000 0000 0000 1000 0000

\* 即十进制 128

\*/

System.out.println(a & b);

/\*\*

\* a 二进制码 0000 0000 0000 0000 0000 0000 1000 0001

\* ~a {按位取反运算符翻转操作数的每一位，即0变成1，1变成0。}

\* 1111 1111 1111 1111 1111 1111 0111 1110

\* 发现符号位为1，为负数：各位取反，加1,：

\* 0000 0000 0000 0000 0000 0000 1000 0001(取反)

\* 0000 0000 0000 0000 0000 0000 1000 0010(+1)

\* 即二进制 -130

\*/

System.out.println(~a);

/\*\*

\* a 二进制码 0000 0000 0000 0000 0000 0000 1000 0001

\* b 二进制码 0000 0000 0000 0000 0000 0000 1001 0100

\* a|b {如果相对应位都是0，则结果为0，否则为1}

\* 0000 0000 0000 0000 0000 0000 1001 0101

\* 即二进制 149

\*/

System.out.println(a|b);

/\*\*

\* a 二进制码 0000 0000 0000 0000 0000 0000 1000 0001

\* b 二进制码 0000 0000 0000 0000 0000 0000 1001 0100

\* a^b {如果相对应位值相同，则结果为0，否则为1}

\* 0000 0000 0000 0000 0000 0000 0001 0101

\* 即二进制 21

\*/

System.out.println(a^b);

### 2.0.3 拆箱及装箱

首先认识java八种原生数据类型(也叫基本数据类型）: byte(8位，一个字节数)、short(16位，2个字节数)、int(32位，4个字节数)、long(64位，8个字节数)[注:一个字节数8位]、float(32位，4个字节数)、double(64位，8个字节数)、char(16位，2个字节数)、boolean(8位，1个字节数);

包装类:Byte, Short, Integer, Long, Float, Double, Character, Boolean。

装箱:将基本数据类型转换为包装类；

eg：Integer a = new Integer(10);//装箱

Integer a = 10; //自动装箱

拆箱:将包装类转换为基本数据类型；

eg：

Integer a = new Integer(10); int a = a.intValue(); //拆箱

Integer a = new Integer(10);int b = a;/自动拆箱

自动装箱和自动拆箱主要是有编译时候自动转换。

eg:执行以下内容:

Integer aa = 1;

int aaa = aa;

System.out.println(aaa);

执行javap -c com/mjs/test/Test

public static void main(java.lang.String[]);

Code:

0: iconst\_1

1: invokestatic #2 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

4: astore\_1

5: aload\_1

6: invokevirtual #3 // Method java/lang/Integer.intValue:()I

9: istore\_2

10: getstatic #4 // Field java/lang/System.out:Ljava/io/PrintStream;

13: iload\_2

14: invokevirtual #5 // Method java/io/PrintStream.println:(I)V

17: return

注: 装箱自动调用的是Integer的valueOf(int)方法。拆箱自动调用的是Integer的intValue方法。

相关知识点:

（1）使用"=="比较使用Integer装箱数据

Integer i1 = 100;

Integer i2 = 100;

Integer i3 = 200;

Integer i4 = 200;

System.out.println(i1==i2);

System.out.println(i3==i4);

输出： true

false

注:通过valueOf方法创建Integer对象的时候，如果数值在[-128,127]之间，便返回指向IntegerCache.cache中已经存在的对象的引用；否则创建一个新的Integer对象。

### 2.0.4 java中==和equals和hashCode的区别

（1）==

首先认识java八种原生数据类型(也叫基本数据类型）:byte(8个字节)、short(16个字节)、int(32个字节)、long(64个字节)[注:一个字节8位]、float(32个字节)、double(64个字节)、char、boolean;

"==" 比较两个变量的值，若针对基本数据类型，比较的是两个存值是否相同；若是针对引用数据类型，变量存的是地址的值，比较的都是变量直接存储的值；

eg：

int a =1;

int b = 1;

Integer c = new Integer(1);

Integer d = new Integer(1);

System.out.println(a==b);

System.out.println(b==c);

System.out.println(c==d);

输出:

true

true //此处Integer自动拆箱为int

false

(2) equals

通过查看JDK查看Object类的equals方法注释

指示其他某个对象是否与此对象“相等”。

equals 方法在非空对象引用上实现相等关系：

自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。

对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改。

对于任何非空引用值 x，x.equals(null) 都应返回 false。

Object 类的 equals 方法实现对象上差别可能性最大的相等关系；即，对于任何非空引用值 x 和 y，当且仅当 x 和 y 引用同一个对象时，此方法才返回 true（x == y 具有值 true）。

注意：当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

(3)hashCode

主要是返回对象的哈希码值(该值通过将对象的内部地址转换成一个整数来实现)；

hashCode()方法主要是为了更好支持哈希表(HashMap、HashSet、HashTable等)；

哈希表的存储原理:

当我们向哈希表插入一个Object时，首先调用hashCode()方法获取该对象的哈希码值，通过该值定位Object在哈希表中的位置。如果该位置没有对象，将Object插入该位置，如果该位置有对象(可能多个，通过链表实现)，则调用equals()方法将这些对象与Object比较，若相等，则不需要保存，否则，将该对象插入至该链表中。

则若equals相等，hashCode()必须相等；hashCode()相等，equals不一定相等。

### 2.0.5 原生数据类型、位数及字节数

byte(8位，一个字节数)、short(16位，2个字节数)、int(32位，4个字节数)、long(64位，8个字节数)、float(32位，4个字节数)、double(64位，8个字节数)、char(16位，2个字节数)、boolean(8位，1个字节数);

注: 一个字节数8位。

Unicode/GBK： 中文2字节  
UTF-8： 中文通常3字节，在拓展B区之后的是4字节  
综上，中文字符在编码中占用的字节数一般是2-4个字节。

### 2.0.6 HashMap,Hashtable,ConcurrentHashMap和synchronizedMap的原理和区别

(1)HashMap：

基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。注意，此实现不是同步的。如果多个线程同时访问一个哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须 保持外部同步。（结构上的修改是指添加或删除一个或多个映射关系的任何操作；仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过对自然封装该映射的对象进行同步操作来完成。

实现原理:通过数组及指针两个数据结构进行实现，HashMap 底层采用一个 Entry[] 数组{Entry为单向链表}来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据 hash 算法来决定其在数组中的存储位置，在根据 equals 方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry 时，也会根据 hash 算法找到其在数组中的存储位置，再根据 equals 方法从该位置上的链表中取出该Entry。

(2)Hashtable:

Hashtable和HashMap类似，但是Hashtable是基于Dictionary 类，而且HashTable是同步的，HashTable内的同步方法都是使用synchronized实现的。

(3)ConcurrentHashMap:

Hashtable是因为多个线程访问Hashtable时争夺一把锁导致效率低。若容器拥有多把锁，每把锁控制容器的每部分数据。这样能够在多线程访问容器时不存在锁的竞争，能够有效提高并发访问效率(分段锁技术)。将ConcurrentHashMap容器的数据分段存储，每一段数据分配一个Segment（锁），当线程占用其中一个Segment时，其他线程可正常访问其他段数据。

ConcurrentHashMap由Segment和HashEntry组成。

Segment是可重入锁，它在ConcurrentHashMap中扮演分离锁的角色；

HashEntry主要存储键值对；

ConcurrentHashMap包含一个Segment数组，每个Segment包含一个HashEntry数组并且守护它，当修改HashEntry数组数据时，需要先获取它对应的Segment锁；而HashEntry数组采用开链法处理冲突，所以它的每个HashEntry元素又是链表结构的元素。

ConcurrentHashMap实现高并发的主要因素：

1. 使用分段锁(亦叫分离锁)实现多个线程间的更深层次的共享访问，减少了请求同一个锁的频率；
2. 用HashEntery对象的不变形来降低执行读操作的线程在访问遍历链表期间对加锁的需求；
3. 通过对同一个volatile变量的读/写访问，协调不同线程间读/写操作的内存可见性；

(4)synchronizedMap:

Collections.synchronizedMap(hashMap)主要是使用Collections类，将非同步的Map转化为Collections类的内部同步类SynchronizedMap，该同步类主要的所有实现方法都是使用synchronized同步块实现；

因该方式是对整个hash表结构做锁定操作，在锁表的期间，其他线程无法操作。遂性能不高。

### 2.0.7 java多态的理解

多态存在的三个必要条件：

一、要有继承；

二、要有重写；

三、父类引用指向子类对象。

Java中多态的实现方式：

接口实现；

继承父类进行方法重写；

同一个类中进行方法重载；

### 2.0.8 String、StringBuffer、StringBuilder区别

String类长度是不可变的，StringBuffer和StringBuilder类长度是可以改变的；

StringBuffer类是线程安全的，StringBuilder不是线程安全的；

(1)String

String是不可变的对象，因此每次在对String类进行改变的时候都会生成一个新的string对象，然后将指针指向新的string对象，所以经常要改变字符串长度的话不要使用string，因为每次生成对象都会对系统性能产生影响，特别是当内存中引用的对象多了以后，JVM的GC就会开始工作，性能就会降低。

String str = "Hello" + " World";//jvm自动优化，该种拼接被编译器编译成StringBuffer对象的拼接；

str = str + ",I am tonny";//该种方式就会自动创建StringBuffer对象，再使用append方法追加，此时已额外创建StringBuffer

(2)StringBuffer

每次都会对StringBuffer对象本身进行操作，而不是生成新的对象并改变对象引用，

对于String多个字符串使用+时候拼接频繁创建StringBuffer对象，导致性能效率低，此时使用StringBuffer比较理想；

StringBuffer类线程安全原理:使用synchronized同步方法实现(StringBuffer sb = new StringBuffer();若多个线程访问sb的方法，首先获取sb对象锁，才能执行相关方法)

(3)StringBuilder

与StringBuffer一致，但是不能保证同步。在单线程使用时效率比StringBuffer高{因为该类不存在synchronized上下文创建}；

基本原则：如果要操作少量的数据，用String ；单线程操作大量数据，用StringBuilder ；多线程操作大量数据，用StringBuffer。

### 2.0.9 内部类及内部类的作用

封装类一个类(外部类)里面的类，称为内部类；

(1)特点:

2)外部类不允许访问内部类的方法及属性；

3)内部类可以直接访问外部类包含private修饰的属性及方法；

4)可作为回调使用；

5)内部类编译后，会产生外部类.class及外部类$内部类.class等文件；

eg:

/\*\*

\* @Description 外部类

\* @ClassName Test

\* @Author Administrator

\* @Data 2018/9/7 22:32

\* @Version 1.0

\*/

public class Outer {

/\*\*

\* 外部类的变量

\*/

private String outVar;

/\*\*

\* 外部类静态变量

\*/

private static String staticOutVar = "Static Out Var";

/\*\*

\* 获取静态变量值，静态方法

\*/

public static String getStaticVar(){

return staticOutVar;

}

/\*\*

\* Getter方法

\* @return

\*/

public String getOutVar() {

return outVar;

}

public void setOutVar(String outVar) {

this.outVar = outVar;

}

/\*\*

\* 声明内部类

\*/

class Inner{

/\*\*

\* 声明内部类变量

\*/

private String innerVar;

public String getInnerVar() {

return innerVar;

}

public void setInnerVar(String innerVar) {

this.innerVar = innerVar;

}

public boolean addStr(String aStr){

innerVar = Outer.this.outVar + aStr;

//内部类对象通过‘外部类名.this.xxx’的形式访问外部类的属性与方法。

Outer.this.outVar = innerVar;

System.out.println("innerVar:" + innerVar + " ;Outer.this.outVar:" + Outer.this.outVar);

return true;

}

}

/\*\*

\* 执行内部类方法

\* @param aStr

\*/

public void doInner(String aStr){

new Inner().addStr(aStr);

System.out.println("addComplete");

}

public static void main(String[] args) {

/\*\*

\* 声明外部类并且外部类使用内部类做事件驱动

\*/

Outer outer = new Outer();

outer.setOutVar("Hello ");

outer.doInner("world");

//

/\*\*

\* 创建内部类的两种方式

\*/

// Outer outer = new Outer();

// Outer.Inner inner = outer.new Inner();

// Outer.Inner inner2 = new Outer().new Inner(); //匿名外部类创建

}

}

(2)匿名内部类，编写事件驱动程序。

a、java回调机制：

模块之间存在三种调用方式:同步、异步、回调三种方式；

同步调用:阻塞式调用，调用者需等待被调者执行完毕返回才能执行后续流程;

回调:一种双向调用模式，被调者在接口被调用时也会调用对方的接口；

异步调用:接口的服务在收到某种讯息或发生某种事件时，会主动通知客户方；

eg:

/\*\*

\* @Description 回调接口或抽象

\* @ClassName CallBack

\* @Author Administrator

\* @Data 2018/9/11 21:39

\* @Version 1.0

\*/

public abstract class CallBack {

public abstract void execute();

}

/\*\*

\* @Description 外部类

\* @ClassName Test

\* @Author Administrator

\* @Data 2018/9/7 22:32

\* @Version 1.0

\*/

public class Outer {

/\*\*

\* 执行内部类方法

\* @param callBack

\*/

public void doInner(CallBack callBack){

System.out.println("调用前...");

callBack.execute(); //执行回调

System.out.println("调用后...");

}

public static void main(String[] args) {

Outer outer = new Outer();

outer.doInner(new CallBack() {

@Override

public void execute() {

System.out.println("执行回调ing...");

}

});

}

}

执行结果:

调用前...

执行回调ing...

调用后...

编译后产生Outer.class及Outer$1.class：

Outer.class:

import java.io.PrintStream;

public class Outer

{

public void doInner(CallBack callBack)

{

System.out.println("调用前...");

callBack.execute();

System.out.println("调用后...");

}

public static void main(String[] args) {

Outer outer = new Outer();

outer.doInner(new Outer.1());

}

}

Outer$1.class：

import java.io.PrintStream;

final class Outer$1 extends CallBack

{

public void execute()

{

System.out.println("执行回调ing...");

}

}

注:

b、事件驱动编程模型:

事件驱动，是指java应用能够对某些事件作出响应。发出事件的对象为事件源，对事件感兴趣的对象，称为监听器。事件驱动就是事件源发出事件，监听器捕获事件并作出响应的过程。

c、编写事件驱动程序：

Frame frame = new Frame("面板");

Button button = new Button("点击按钮");

// 增加事件处理器，以匿名回调类方式

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("按钮被点击");

String label = e.getActionCommand();

System.out.println("按钮名称:" + label);

}

});

// 将按钮加入frame

frame.add(button, BorderLayout.CENTER);

frame.pack();

frame.setVisible(true);

### 2.0.10 抽象类和接口区别

抽象类的定义:使用abstract修饰符修饰的类。若一个类包含一个或多个抽象方法，则该类必须限定为抽象的。

接口的定义:一个抽象类型，是抽象方法的集合，接口通常以interface来声明。接口自java8后可以有默认实现方法(default定义)，可以定义一个或多个默认方法。

区别:

(1)实现抽象类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。

子类使用关键字implements来实现接口。它需要提供接口中所有声明的方法的实现

(2)抽象类可以有构造器，而接口不能有构造器；

(3)抽象方法可以有public、protected和default这些修饰符；

接口方法默认修饰符是public。你不可以使用其它修饰符；

抽象类在java语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。

(4)抽象方法比接口效率高；

(5)使用场景：

多重继承，必须使用接口；

默认方法实现，使用抽象类；(JDK8以后接口可有默认实现)；

### 2.0.11 java 泛型原理

参数化类型，通过使用参数化类型创建的接口、类、方法，可以指定所操作的数据类型。

有界通配符，可以为参数类型指定上界和下界，限制操作的参数类型。

a)<? extends superclass> 详见java技术-Java基础知识-泛型中extends和super的区别。

b)<? super subclass> 详见java技术-Java基础知识-泛型中extends和super的区别。

(1)泛型类

/\*\*

\* @Description 泛型类示例

\* @ClassName GenericClass

\* @Author Administrator

\* @Data 2018/9/24 20:19

\* @Version 1.0

\*/

/\*\*

\* 若声明多个参数类型，需使用逗号分隔参数列表<T, V, ....>

\* @param <T>

\*/

public class GenericClass<T> {

T obj;

GenericClass(T o){

this.obj = o;

}

public T getObj() {

return obj;

}

public void setObj(T obj) {

this.obj = obj;

}

public static void main(String[] args) {

/\*\*

\* (1)创建对象时，传递一个实际类型，类中只要使用类型参数的地方使用了T,当传递实际类型后，会自动改变成实际类型。

\* new 后将所有泛型类型移除，进行类型转换，从而看似是创建了一个个Generic类版本。移除泛型的过程称为擦除。

\* (2)声明泛型实例时，传递的类型参数必须为引用类型。不能是基本类型(装箱)。

\*

\*/

GenericClass<String> gTest = new GenericClass<String>("Test");

System.out.println(gTest.getObj());

GenericClass<Integer> gTest1 = new GenericClass<Integer>(1);

System.out.println(gTest1.getObj());

}

}

(2)泛型方法

/\*\*

\* @Description 泛型方法实例

\* @ClassName GenericMethod

\* @Author Administrator

\* @Data 2018/9/25 15:17

\* @Version 1.0

\*/

public class GenericMethod {

/\*\*

\* 格式：修饰符 <类型参数列表> 返回类型 方法名(形参列表) { 方法体 }

\* @param t

\* @param <T>

\* @return

\*/

public <T extends Number> void test(T t){

System.out.println("class:" + t.getClass() + " " + t.toString());

}

public <T extends Number> int compareNum(T t){

/\*\*

\* 精度丢失

\*/

return new BigDecimal(t.shortValue()).compareTo(new BigDecimal(0));

}

public static void main(String[] args) {

GenericMethod g = new GenericMethod();

int i = 2;

g.test(i);

System.out.println(g.compareNum(i));

}

}

(3)泛型接口

/\*\*

\* 泛型接口示例

\* 格式 修饰符 interface 接口名<形参列表>

\* @param <T>

\*/

public interface GenericInterface<T> {

public void test();

}

/\*\*

\* @Description 泛型接口实现

\* @ClassName GenericInterfaceImpl

\* @Author Administrator

\* @Data 2018/9/25 15:53

\* @Version 1.0

\*/

/\*\*

\* 格式:

\* class 实现类类名<参数列表> implements 接口名<参数列表>

\* (1)若接口已指定界限

\*

\* (2)若接口未指定界限

\*/

public class GenericInterfaceImpl<T> implements GenericInterface{

private T obj;

public GenericInterfaceImpl(T obj){

this.obj = obj;

}

@Override

public void test() {

System.out.println(this.obj);

}

public static void main(String[] args) {

GenericInterface<String> g = new GenericInterfaceImpl<String>("hello world");

g.test();

}

}

注：类型擦除：

Class c1=new ArrayList<Integer>().getClass();

Class c2=new ArrayList<String>().getClass();

System.out.println(c1==c2); //输出结果为true

### 2.0.12 泛型中extends和super的区别

泛型:泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

关于泛型详见java技术-Java基础知识-泛型。

泛型方法规则:

(1)所有泛型方法都有一个类型参数声明部分(由尖括号分隔)，该类型参数声明部分在方法返回类型之前；

(2)每个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开<E>,<E,T>等待；

eg：

public <E> E c(E t){

return t;

}

A a = new A();

System.out.println(a.c("hello world"));//输出 "hello world"

泛型类规则:

(1)类名后添加类型参数声明部分，声明部分包含一个或多个类型参数，参数间用逗号隔开；

public class A<T>{

private T obj;

public void add(T t){

this.obj = t;

}

public T get(){

return obj;

}

public static void main(String[] args) {

A<Integer> a = new A<Integer>();

a.add(1);

// a.add("hello world"); //编译失败

System.out.println(a.get());

}

}

<? extends T> 限定参数类型的上界:参数类型必须是T或T的子类型；该通配符只能适用于方法返回(返回的类型必须是子类或者本身)；

<? super T> 限定参数类型的下界:参数类型必须是T或T的超类型；该通配符只能适用于方法入参(传参类型必须是本身或者父类)；

List<? extents People> l1 = ...;

List<? super People> l2 = ...;

误区:l1并不是说l1只能放入People对象类型或People派生类对象类型，l2并不是说l2只能放入People对象类型或People继承类对象类型；

正解: List<T> l3 = ...; 表示一个泛型。? extents People和? super People表示l3的泛型T;

针对? extents People：表示的泛型参数必须满足People类型或子类型<People,Chinese,American>

针对? super People：表示泛型参数必须满足为People类型或继承类<People,Object>

当然List<People,Chinese,American>{该种写法是错误的} list = new ArrayList<People>();//当然是不成立的

List<People,Object>{该种写法是错误的} list = new ArrayList<People>();//当然是成立的

eg：

/\*\*

\* 注：该obj对象不能放入如何类型，因编译只知道放入People或的其子类型，但具体类型并不确定，因此除了null之外，不能存放任何类型

\* 总结：<? extends T>只能往外取(可以确定该对象必定是People或的其子类型，向上转型可以获取)，不能往里存

\*/

List<? extends People> obj = new ArrayList<>();

//List<? extends People>包含了List<People>、List<Chinese>、List<American>，三种类型

//List<? extends People> obj\_people = new ArrayList<People>();

//List<? extends People> obj\_chinese = new ArrayList<Chinese>();

//List<? extends People> obj\_american = new ArrayList<American>();

// obj.add(new People("张三", 3)); //不能添加,因为obj有可能是List<Chinese>，也有可能是List<American>

// obj.add(new Chinese("李四", 4)); //不能添加,因为obj有可能是List<People>，也有可能是List<American>

// obj.add(new American("王五", 5)); //不能添加,因为obj有可能是List<Chinese>，也有可能是List<People>

obj.add(null); //能添加,null对象提供一种占位符机制，事实上无任意意义

/\*\*

\* 总结：<? super People>能够往里存，但是取需要Object接收(导致类型信息丢失)

\*/

//<? super People>包含了<Object>、<People>及<People继承类>等多种类型

//因此obj2类型能够添加任何People及其派生类型对象，但不能添加Object、People派生类对象，因为不确定其类型;

//读取，<? super People>获取到的对象都为Object对象

List<? super People> obj2 = new ArrayList<>();

obj2.add(new People("张三", 3));

obj2.add(new Chinese("李四", 4));

obj2.add(new American("王五", 5));

People p = (People) obj2.get(0);

System.out.println(p.toString());

### 2.0.13 final，finally，finalize的区别

三者没有必然联系：

(1)final控制成员变量、方法或者类是否可以被重新或继承等功能。若修饰方法，意味着该类不允许派生出子类；若修饰方法或者变量，可保证在使用中被改变；

注:final修饰的变量，初始化在定义处赋值或者构造函数处赋值，两者只能其一，否则编译错误；

(2)finally主要是异常处理最终出口，finally体内的程序必须被执行，主要用于关闭文件的读写操作，或者是关闭数据库的连接等等。

(3)finalize主要用于垃圾回收。java运行使用finalize(）方法在gc将对象从内存中清理之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是Object类中定义的，因此，所有的类都继承了它。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

### 2.0.14 序列化的方式Serializable 和Parcelable 的区别

对象进行序列话都需要实现的接口。详见java技术-JAVA远程通讯-1.远程方法调用(RMI)内容的序列化及反序列化。Serializable为Java自带，Parcelable 为安卓专用

### 2.0.15 重写、继承及重载

重写(覆盖):子类中重新定义父类中的方法；

继承:子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为；

重载:子类或者自身类中重新定义类的方法的参数或返回值(返回类型、参数属性、参数个数)；

### 2.0.16 静态属性和静态方法是否继承或重写

静态属性和静态方法可以被继承，但是没有被重写(overwrite)而是被隐藏。

原因:

静态属性和静态方法属于类级别(非对象级别)。如果子类里面定义了静态方法和属性，那么这时候父类的静态方法或属性称之为"隐藏"。如果你想要调用父类的静态方法和属性，直接通过父类名.方法或父类名.变量名完成，至于是否继承一说，子类是有继承静态方法和属性，但是跟实例方法和属性不太一样，存在"隐藏"的这种情况。

抄自简书。

### 2.0.17成员内部类、静态内部类、局部内部类和匿名内部类

一个外部类的内部再定义一个类。内部类主要分为4种，成员内部类、静态内部类、局部内部类和匿名内部类

(1)局部内部类:在方法的作用域内定义的的内部类;

/\*\*

\* @Description 局部内部类

\* @ClassName PartOuter

\* @Author Administrator

\* @Data 2018/9/20 10:09

\* @Version 1.0

\*/

public class PartOuter {

private int i = 1;

public void method() {

int a = 2;

final int b = 0;

class Inner {

int i = 3;

void f() {

//内部类方法可以直接访问外类的实例变量

System.out.println(i);

//内部类可以访问final局部变量,非final编译不通过

System.out.println(b);

//内部类中有与外部类同名的变量，直接用变量名访问的是内部类的变量

System.out.println(i);

System.out.println(this.i);

System.out.println(PartOuter.this.i);

}

}

new Inner().f();

}

public static void main(String[] args) {

PartOuter p = new PartOuter();

p.method();

/\*\*

\* 输出：

\* 3

\* 0

\* 3

\* 3

\* 1

\*/

}

}

(2)成员内部类:作为外部类的成员，与外部类的属性、方法并列。(可以直接使用外部类的所有成员和方法，即使是private的。同时外部类要访问内部类的所有成员变量/方法，则需要通过内部类的对象来获取。成员内部类不能含有static的变量和方法。)

/\*\*

\* @Description 成员内部类列举

\* @ClassName MemberOuter

\* @Author Administrator

\* @Data 2018/9/20 10:03

\* @Version 1.0

\*/

public class MemberOuter {

private int i;

class Inner{

//内部类不能定义静态成员

}

}

(3)静态内部类：

非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。

没有这个引用就意味着：

它的创建是不需要依赖于外围类的。

它不能使用任何外围类的非static成员变量和方法。

/\*\*

\* @Description 静态内部类

\* @ClassName StaticOuter

\* @Author Administrator

\* @Data 2018/9/20 10:49

\* @Version 1.0

\*/

public class StaticOuter {

public int a;

public static int b;

static class Inner {

int c;

void a(){

//只能访问外部类静态变量及方法,非静态变量及方法不能访问

System.out.println(StaticOuter.b);

}

}

public static void main(String[] args) {

/\*\*

\* 生成一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成

\*/

StaticOuter.Inner inner = new StaticOuter.Inner();

inner.a();

}

}

匿名函数：详见java基础知识内部类及内部类的作用，匿名函数部分。

### 2.0.18 闭包和局部内部类的区别

闭包:是一个可调用的对象，它记录了一些信息，这些信息来自于创建它的作用域--java编程思想

eg:

javaScript创建闭包(引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外)

/\*\*

\* 函数b嵌套在函数a内部，函数a返回函数b

\* (1)通过调用a函数,返回函数b，并将引用指向变量c；

\* (2)通过指向c()，也就是执行函数b;

\* 当函数a的内部函数b被函数a外的一个变量引用的时候，就创建了一个闭包。

\* 也就是 变量i相对于 函数b来说，就是一个自由变量

\*/

function a(){

var i = 0;

function b(){

alert(++i);

}

return b;

}

var c = a(); //调用此处便形成一个闭包

c();

var powerOfN = function(n){

return function(m){

var res = 1;

for(var i = 0; i < n; i ++){

res \*= m;

}

return res;

}

}

//获取内部函数

var powerOf2 = powerOfN(2); //获取平方计算方法 变量n成为自由变量

var powerOf3 = powerOfN(3); //获取立方计算方法

console.log(powerOf2(3));

console.log(powerOf3(2));

内部类+接口就是Java最常用的闭包实现办法。

public interface ILog {

public void write(String message);

}

public class Outer {

private int length =0;

//private|public

private class InnerClass implements ILog

{

@Override

public void write(String message) {

//DemoClass1.this.length = message.length();

length = message.length();

System.out.println("Outer.InnerClass:" + length);

}

}

public ILog logger() {

return new InnerClass();

}

public static void main(String[] args){

/\*\*

\* 通过外部方法返回内部类时 即调用demoClass1.logger()-->new InnerClass(),就形成闭包

\* 注:存在外部类的一下成员变量(包含私有的)

\*/

Outer demoClass1 = new Outer();

demoClass1.logger().write("abc");

/\*\*

\* 直接通过外部类 构建内部类实例，并且调用其方法

\*/

Outer dc1 = new Outer();

InnerClass ic = dc1.new InnerClass();

ic.write("abcde");

}

}

### 2.0.19 String 转换成 Integer的方式及原理

Integer.valueOf("1");

(1)调用Integer的静态方法valueOf(String s)；

(2)parseInt(String s, int radix)使用第二个参数指定的基数,将字符串解析为有符号的整数：

parseInt("0", 10) 返回 0

parseInt("473", 10) 返回 473

parseInt("-0", 10) 返回 0

parseInt("-FF", 16) 返回 -255

eg： "110"

a)首先边界校验，判断正负

判断首字符是否为"+"或"-",negative为false(非负数)

b)循环字符串中每个值，调用Character.digit(char c, int radix) 返回使用指定基数的字符 ch 的数值。

并以基数来计算

result = 0; radix = 10;

(1)循环第一次，第一位(从左到右)为字符"1";

result \*= radix; 即 result = 0 \* 10 = 0；

result -= digit; 即 result = 0 - 1 = -1;

(2)循环第二次，第二位为字符"1";

result \*= radix; 即 result = -1 \* 10 = -10;

result -= digit; 即 result = -10 - 1 = -11;

(3)循环第三次，第三位为字符"0";

result \*= radix; 即 result = -11 \* 10 = -110;

result -= digit; 即 result = -110 - 0 = -110;

c)得到结果。 return negative ? result : -result; return 110;

### 2.0.20 常见编码方式

常见编码方式:ASCII、Unicode、UTF-8

(1) ASCII (American Standard Code for Information Interchange，美国信息交换标准代码)。定义了128个字符。

其中，图形字符大写字母A-Z对应ASCII的十进制码为:65-90，图形字符小写字母a-z对应ASCII的十进制码为97-122，图形数字0-9对应ASCII的十进制码为48-57；

注:ASCII为8位比特位组成(范围为0000 0000至0111 1111，最高位恒为0)。一个字节(8位)表示。

(2)Unicode

Unicode 只是一个符号集，它只规定了符号的二进制代码。实现方式有多种：UTF-8/UTF-16/UTF-32。

(3)UTF-8

UTF-8编码方式是Unocode的实现方式之一。使用1-4个字节表示一个符号，根据不同的符号而变化字节长度。

A)单字节符号：字节首位设为0，后七位为符号的 Unicode 码，表示ASCII码对应的图形符号；

B)多字节字符(字节数大于1)，若字节数为2(设为n)，在第2(n)个字节(8位ASCII码)的前2(n)位设置为1，第3(n + 1)位设置为0;在非第2(n)个字节的前两位一律设置为10。未设置的二进制位便为对应的unicode码的二进制位。

Unicode符号范围 | UTF-8编码方式

(十六进制) | （二进制）

----------------------+---------------------------------------------

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

所以Unicode符号集转换为UTF-8编码方式的过程:先查字符对应的unicode二进制代码，转十六进制获取Unicode码的范围，确定转换后UTF-8码方式，然后将符号代表Unicode编码方式的二进制依次从右填充到UTF-8码格式中，若位数有余则0。

### 2.0.21 Java中String的了解

#### (1)什么String

(A)String类是final类，也即意味着String类不能被继承，并且它的成员方法都默认为final方法。（注：被final修饰的类是不允许被继承的，并且该类中的成员方法都默认为final方法。）

(B)String类其实是通过char数组来保存字符串的。

(C)String对象一旦被创建就是固定不变的了，对String对象的任何改变都不影响到原对象，相关的任何change操作都会生成新的对象。

#### (2) String为什么要设计成不可变的

(A)字符串常量池:Java堆内存中一个特殊的存储区域, 当创建一个String对象时,假如此字符串值已经存在于常量池中,则不会创建一个新的对象,而是引用已经存在的对象。

若该字符串允许可变，影响另外一个对象发生改变。

(B)允许String对象缓存HashCode

String对象HashCode频繁使用，若改变，HashCode也需重新计算

(C)安全性

多线程访问，不可变对象是线程安全的，即String对象不能被改变。

注：String对象不可变，但是指向引用却是会发生变化的，例如：

a) String a = "Hello";

在字符串常量池新建一个"Hello"字符串对象，并且a指向它。

b) a = "world";

在字符串常量池新建一个"world"字符串对象，并且a指向它。

多线程访问时，a的指向会重定向，但是对于"Hello"字符串对象是不可变的。

保证线程安全性:

a)volatile确保可见性；

b)可以使用fianl修饰；（初始化或或实例化后不可修改）

c)使用synchronized对相关方法或区域进行加锁；

d)使用AtomicReference之类的原子对象；

### 2.0.22 java依赖注入的理解

依赖：ClassA中需要ClassB的实例，则为A对B有一个依赖。

例：

public class ClassB {

}

public class ClassA {

ClassB b;

public ClassA(){

b = new ClassB();

}

}

依赖注入：在依赖的基础上，对象ClassB的实例是通过外部传入依赖的方式(非主动new)，为依赖注入。

例:

public class ClassA {

ClassB b;

public ClassA(ClassB classB){

b = classB;

}

}

spring实现依赖注入(DI)原理：

(1)IOC:控制反转，IOC容器负责创建和获取依赖对象，对象只是被动地接受依赖对象。也就是将依赖方式从内部new对象转移到IOC容器构造。

实现方式:1.XML配置方式；2.注解方式；3.自动装配方式。

(2)DI:依赖注入，将对象所所需的依赖从外部注入。也就是IOC容器自动将对象所需的依赖注入。

### 2.0.23 Object类的equals和hashCode方法重写

(1)hashCode

方法注释翻译：返回该对象的哈希码值。支持此方法是为了提高哈希表的性能。

常规协定：

a)在java应用执行期间，在对同一个对象多次调用hashCode方法时，必须一致地返回相同的整数，前提是将对象进行equals比较时所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。

b)如果根据equals(Object)方法,两个对象是相等的，那么对这两个对象中的每个对象调用hashCode方法都必须生成相同的整数结果。

c)如果根据equlas(Objcect)方法，两个对象不相等，那么对这两个对象中的任一对象上调用hashCode方法不要求一定生成不同的整数结果。但是，程序员应该意识到，为不相等的对象生成不同整数结果可以提高哈希表的性能。

实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。

注：针对c)"为不相等的对象生成不同整数结果可以提高哈希表的性能"。查询时利用哈希方法返回哈希码(每个对象的hashCode在对象生成时就已经存在)，若一致时，会调用equals方法判断，导致性能下降。插入时与查询一致，先判断hash是否相等，再判断equals方法。

(2)equals

方法注释翻译：指示其他某个对象是否与此对象"相等"。

注意：当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

(3)obj1.equals(obj2)和"=="

equals若没有重写，首先会调用继承树最近的节点并已经重写equals，根节点Object对象的equals方法就是返回this == obj,比较的是内存地址是否一致。

(4)针对重写equalss()方法就必须重写hashCode()方法

Object类的equals方法注释已经清楚描述"当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。"。

原因:

导致使用hash函数的数据类型出现错误，无法使用哈希表类似的数据结构(HashMap,HashTable等)；

eg:

针对Map<Key, Value> map = new HashMap<Key, Value>();

(1)map进行put操作时，首先获取Key的hashCod;

(2)判断hashCode是否一致，若一致，使用equals方法判断不相等，则追加在链表后，使用equals方法判断相等，不进行追加；

但是由于只是重写在equals方法，但是没有重写hashCode方法，此时hashCode方法属于Object类。若不同对象得到的hashCode就不一样。

import java.util.HashMap;

import java.util.Map;

/\*\*

\* @Description 若只重写equals不重写hashCode示例

\* @ClassName Person

\* @Author Administrator

\* @Data 2018/9/26 17:05

\* @Version 1.0

\*/

public class Person {

private String name;

public Person(String name){

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj instanceof Person) {

Person anotherSon = (Person)obj;

if(this.getName() == anotherSon.getName()){

return true;

}

}

return false;

}

public static void main(String[] args) {

Person p1 = new Person("张三");

Person p2 = new Person("李四");

Person p3 = new Person("张三");

Map<Person, String> map = new HashMap<Person, String>();

map.put(p1, "三只苹果");

map.put(p2, "四只苹果");

map.put(p3, "五只苹果");

/\*\*

\* 输出：

\* map-size 3

\* null

\* null

\* size长度为3的原因:HashMap使用put的方法时，p1和p3的hash值不一致，会存在hash表的不用index上，所以无法覆盖；

\* 获取Value为空原因:HashMap使用get的方法时，首先调用hashCode()获取hash，因为没重写hashCode方法，得到的hash不一致导致，无法再hash表上获取相匹配的hash值；

\*/

System.out.println("map-size " + map.size());

System.out.println(map.get(new Person("张三")));

System.out.println(map.get(new Person("李四")));

}

}

import java.util.HashMap;

import java.util.Map;

/\*\*

\* @Description 若只重写equals不重写hashCode示例

\* @ClassName Person

\* @Author Administrator

\* @Data 2018/9/26 17:05

\* @Version 1.0

\*/

public class Person {

private String name;

public Person(String name){

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj instanceof Person) {

Person anotherSon = (Person)obj;

if(this.getName() == anotherSon.getName()){

return true;

}

}

return false;

}

/\*\*

\* 重写规则

\* @return

\*/

@Override

public int hashCode() {

int result = 17;

result = result \* 31 + name.hashCode();

return result;

}

public static void main(String[] args) {

Person p1 = new Person("张三");

Person p2 = new Person("李四");

Person p3 = new Person("张三");

Map<Person, String> map = new HashMap<Person, String>();

map.put(p1, "三只苹果");

map.put(p2, "四只苹果");

map.put(p3, "五只苹果");

/\*\*

\* 输出：

\* map-size 2

\* 五只苹果

\* 四只苹果

\*/

System.out.println("map-size " + map.size());

System.out.println(map.get(new Person("张三")));

System.out.println(map.get(new Person("李四")));

}

}

### 2.0.24 深拷贝和浅拷贝

浅拷贝:对基本数据类型进行值传递，对引用数据类型进行引用传递般拷贝；

/\*\*

\* @Description 浅拷贝示例

\* @ClassName Person

\* @Author Administrator

\* @Data 2018/9/26 17:05

\* @Version 1.0

\*/

public class Person implements Cloneable{

private String name;

private Clild clild;

public Person(String name, Clild child){

this.name = name;

this.clild = child;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Clild getClild() {

return clild;

}

public void setClild(Clild clild) {

this.clild = clild;

}

@Override

protected Object clone(){

Person p = null;

try {

p = (Person) super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return p;

}

public static void main(String[] args) {

Clild clild = new Clild("张三儿子");

Person p = new Person("张三", clild);

System.out.println("我叫" + p.getName() + "，我儿子叫" + p.getClild().getName());

Person pc = (Person) p.clone();

Clild clild1 = pc.getClild();

clild1.setName("张三小儿子"); //拷贝一个张三对象，并且改名了

System.out.println("我叫" + p.getName() + "，我儿子叫" + p.getClild().getName());

/\*\*

\* 输出：

\* 我叫张三，我儿子叫张三儿子

\* 我叫张三，我儿子叫张三小儿子

\* 注：此时child1对象和child对象引用的同一个对象，此拷贝为浅拷贝()；

\*/

}

}

class Clild implements Cloneable{

private String name;

public Clild(String name){

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

深拷贝:对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容。

深拷贝实现方式有两种，若存在大量对象，可使用序列化实现对象的拷贝。

例如：

import java.util.HashMap;

import java.util.Map;

/\*\*

\* @Description 深拷贝浅拷贝示例

\* @ClassName Person

\* @Author Administrator

\* @Data 2018/9/26 17:05

\* @Version 1.0

\*/

public class Person implements Cloneable{

private String name;

private Clild clild;

public Person(String name, Clild child){

this.name = name;

this.clild = child;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Clild getClild() {

return clild;

}

public void setClild(Clild clild) {

this.clild = clild;

}

@Override

protected Object clone(){

Person p = null;

try {

p = (Person) super.clone();

p.clild = (Clild) clild.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return p;

}

public static void main(String[] args) {

Clild clild = new Clild("张三儿子");

Person p = new Person("张三", clild);

System.out.println("我叫" + p.getName() + "，我儿子叫" + p.getClild().getName());

Person pc = (Person) p.clone();

Clild clild1 = pc.getClild();

clild1.setName("张三小儿子"); //拷贝一个张三对象，并且改名了

System.out.println("我叫" + p.getName() + "，我儿子叫" + p.getClild().getName());

/\*\*

\* 输出：

\* 我叫张三，我儿子叫张三儿子

\* 我叫张三，我儿子叫张三儿子

\* 注：此时child1对象和child对象引用的非同一个对象，此拷贝为深拷贝；

\* 方法(1)使用clone()方法，针对拷贝对象内存在非基本数据类型，该类(包括子类)需要也需要迭代实现接口Cloneable并重写clone方法，并调用引用对象clone()方法赋值；

\* 方法(2)序列化实现对象的拷贝，略；

\*/

}

}

class Clild implements Cloneable{

private String name;

public Clild(String name){

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

protected Object clone(){

Clild p = null;

try {

p = (Clild) super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return p;

}

}

### 2.0.25 ArrayList和LinkedList的区别，以及应用场景

ArrayList:内部使用数组的形式实现了存储;

LinkedList:内部使用双向链表的结构实现存储;

若频繁读取集合中的元素时，使用ArrayList效率高;若插入和删除操作比较多时，使用LinkedList效率较高；

### 2.0.26 java值传递，引用传递

(1)值传递

是指调用函数时将实际参数复制⼀份传入到函数中，这样函数中如果对参数修改，不会影响实际参数的值。

(2)引用传递

是指调用函数的时候将实际参数的地址传递给函数，这样函数中如果对参数修改，将会影响实际参数的值。 java中方法参数传递方式是按值传递，针对方法的类型为基础数据类型时，传递的是基本类型的字面量值的拷贝。针对传递的对象为非基 础类型时(即引用类型)，传递的是实际参数在栈中地址值的拷贝。 所以得出以下两个结论: (1)当基础数据类型的对象被当作方法参数传递时，该方法不能修改该数据。 (2)当非基础数据类型的对象被当作方法参数传递时，该方法内部可以修改该数据。但是不能让对象参数引用一个新的对象，否则修改无效。

分析 JVM内存模型中有，stack和heap的存在。栈为每个线程独享，所有线程共享一个堆。

*/\*\*  
 \** ***@Description*** *\** ***@ClassName*** *TestValueWay  
 \** ***@Author*** *Administrator  
 \** ***@Data*** *2019/7/13 0:20  
 \** ***@Version*** *1.0  
 \*/***public class** TestValueWay {  
 **public static void** main(String[] args) {  
 CertInfo cert = **new** CertInfo(**"0775"**, **"13333333333"**);  
 *test1*(cert);  
 */\*\*  
 \* sout 077513333333333  
 \*/* System.***out***.println(cert.getCityTel() + cert.getTelPhone());  
 *test2*(cert);  
 */\*\*  
 \* sout 077415888888  
 \*/* System.***out***.println(cert.getCityTel() + cert.getTelPhone());  
 **int** a = 0;  
 *test3*(a);  
 */\*\*  
 \* sout 0  
 \*/* System.***out***.println(a);  
  
 String str = **"a"**;  
 *test4*(str);  
 */\*\*  
 \* sout a  
 \*/* System.***out***.println(str);  
  
 }  
  
 */\*\*  
 \* 值传递-非基本类型传递的是对象的引用的拷贝  
 \* 因为方法内部已经进行形参对象重新构造了一个对象，所以指向新的地址空间，方法内部打印的是新值，但是方法外部打印的确是原值  
 \** ***@param tmpcert*** *\*/* **public static void** test1(CertInfo tmpcert){  
 tmpcert = **new** CertInfo(**"0774"**, **"15888888"**);  
 */\*\*  
 \* sout 077415888888  
 \*/* System.***out***.println(tmpcert.getCityTel() + tmpcert.getTelPhone());  
 }  
 */\*\*  
 \* 值传递-非基本类型传递的是对象的引用的拷贝  
 \* 因为方法内部对形参未进行重新指向新的地址，并且修改了引用指向的值，所以方法外部打印的是修改后的值  
 \** ***@param tmpcert*** *\*/* **public static void** test2(CertInfo tmpcert){  
 tmpcert.setCityTel(**"0774"**);  
 tmpcert.setTelPhone(**"15888888"**);  
 */\*\*  
 \* sout 077415888888  
 \*/* System.***out***.println(tmpcert.getCityTel() + tmpcert.getTelPhone());  
 }  
  
 */\*\*  
 \* 值传递-基本类型传递的是对象值得拷贝(即副本)  
 \* 方法中对该基本数据类型的形参无论何种操作，都不会改变方法外的值  
 \** ***@param tmpa*** *\*/* **public static void** test3(**int** tmpa){  
 tmpa = -1;  
 */\*\*  
 \* sout -1  
 \*/* System.***out***.println(tmpa);  
 }  
  
 */\*\*  
 \* 值传递-非基本类型传递的是对象的引用的拷贝  
 \* 但是由于String、Integer等包装类实际上是使用对应的基本数据类型来存放数据的。所以这些包装类在作为形参传递给函数时，函数同样不会改变实参的值  
 \** ***@param a*** *\*/* **public static void** test4(String a){  
 a = **"b"**;  
 */\*\*  
 \* sout b  
 \*/* System.***out***.println(a);  
 }  
  
}

## 2.0.27 Java (关键字)

#### (1) static

“全局”或者“静态”的意思，用来修饰成员变量和成员方法，当然也可以修饰代码块。

(1)static 变量

static 修饰的变量我们称之为静态变量，没有用 static 修饰的变量称之为实例变量；

静态变量是随着类加载时被完成初始化，它在内存中仅有一个，且 JVM 也只会为它分配一次内存，同时类所有的实例都共享静态变量，可以直接通过类名来访问它。

实例变量是需要构造实例后才能访问该变量；

(2)static 方法

static 修饰的方法我们称之为静态方法，我们通过类名对其进行直接调用。由于他在类加载的时候就存在了，它不依赖于任何实例，所以 static 方法必须实现，也就是说他不能是抽象方法 abstract。

Static修饰的方法出了类名.method()调用以外，也可以使用实例对象.method()调用，但是不能使用this.method()调用。

(3)static 代码块

被 static 修饰的代码块，我们称之为静态代码块，静态代码块会随着类的加载一块执行，而且他可以随意放，可以存在于该类的任何地方。

#### final

final可修饰类、方法和变量（包括成员变量和局部变量）；

(1)修饰类

该类不能被继承，例如String类。

(2)修饰方法

该方法不被继承类重新；

注：类中的private方法隐式指定为final方法

(3)修饰变量

final修饰的基本数据类型的变量，则其数值在初始化之后便不能更改；

final修饰引用类型的变量，初始化后不能再指向其他对象；

注：若final修饰的String类型为常量，编译器就已经分配空间(引用已有)

final修饰的变量必须在构造方法内初始化或直接初始化。

#### (3) transient

(1)仅修饰变量；

(2)若对象已实现Serilizable接口，但是不希望对象中的某些关键字段在网络操作(或者文件保存等)中传输，则使用该字段修饰；

(3)一个静态变量不管是否被transient修饰，均不能被序列化；

(4)若对象实现Externalizable接口(序列化仅有两种方式:Serilizable接口，Externalizable接口)，需要在重写writeExternal方法时，在其内调用out.writeObject指定需序列的内容，与变量是否被transient修饰没有关系。

#### (4) volatile

详见[JAVA技术-多线程-Volatile变量](#_Volatile变量)。

# 框架技术

## 3.1 Spring

## 3.2 AOP

面向切面编程。例如：

A、配置文件方式

<aop:config>

<!-- 定义一个切面，切面是切点和通知的集合-->

<aop:aspect id="do" ref="aspectBean">

<!-- 定义切点 ，后面是expression语言，表示包括该接口中定义的所有方法都会被执行-->

<aop:pointcut id="point" expression="execution(\* com.simple.service..\*.\*(..))" />

<!-- 定义通知 -->

<aop:before method="canLogin" pointcut-ref="point" />

<aop:after method="saveMessage" pointcut-ref="point" />

</aop:aspect>

</aop:config>

注：execution 表达式

(1)通过类包定义切点 execution(\* com.simple.service..\*.\*(..))

第一部分为表达式主体；

第二部分第一个\*代表返回类型，\*表示所有的类型；

第三部分代表包名为com.simple.service，或其子孙包下所有类的方法；

第四部分第二个\*代表类名，\*代表所有的类；

第五部分，其中第三个\*代表方法名，\*代表所有方法，(..)表示方法参数，其中..表示任何参数。

即匹配com.simple.service包及其子孙包下的所有方法。 若\* com.simple.service.\*.\*(..)则仅仅匹配com.simple.service包的所有方法。

(2)通过类定义切点 execution(\* com.simple.service.AopTest.\*(..))

匹配com.simple.service.AopTest接口的所有方法。

....

B、注解方式

略

C、切面、切点、通知

(1)切面(由切点+通知组成)

切面是切点和通知的集合，切点可以映射多个方法。通知类型可以有几种。

(2)切点

配置文件中的pointcut 定义的便是切点，这个切点对应com.simple.service包及子包下面的所有方法，

(3)通知

类型主要有五种，

Before 前置通知

After 返回通知

After-returning 正常返回通知

After-throwing 异常返回通知

Around 环绕通知 围绕在切点前后，能在方法调用前后自定义一些操作。

实现原理：

AOP 来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等。

主要生成代理对象，使用jdk Proxy及Cglib，默认的策略：若目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。详见动态代理。

# 4 Web应用

在不使用IDE情况下构建简单的web应用，首先需要了解web应用的文件结构

## 4.1 REST

Representational State Transfer（表现层状态转换REST）是一种软件架构风格，由创建可伸缩Web服务的指南和最佳实践组成。 REST是一组协调的约束，应用于分布式超媒体系统中的组件设计，可以实现更高性能和可维护的体系结构。

(1)URL可以访问到资源；

(2)状态变化使用HTTP动词实现；

例如，一个简单的网络商店应用，列举所有商品，

GET http://www.store.com/products

呈现某一件商品，

GET http://www.store.com/products/12345

下单购买，

POST http://www.store.com/orders

<purchase-order>

<item> ... </item>

</purchase-order>

摘抄自维基百科。

# 5 JAVA远程通讯

## 5.1 远程方法调用(RMI)

什么是RMI（Remote Method Invocation）？

维基百科：一种用于实现[远程过程调用](https://zh.wikipedia.org/wiki/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8)的[应用程序编程接口](https://zh.wikipedia.org/wiki/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E7%BC%96%E7%A8%8B%E6%8E%A5%E5%8F%A3)。它使客户机上运行的程序可以调用远程服务器上的对象。

(1)序列化：把对象转换为字节序列的过程称为对象的序列化。

(2)反序列化：把字节序列恢复为对象的过程称为对象的反序列化。

对象序列化用途:

1） 把对象的字节序列永久地保存到硬盘上，通常存放在一个文件中；

　　2） 在网络上传送对象的字节序列。

RMI远程调用步骤：

1)，客户对象调用客户端辅助对象（stub对象）上的方法；

2)，客户端辅助对象打包调用信息（变量，方法名），通过网络发送给服务端辅助对象（skeleton对象）；

3)，服务端辅助对象将客户端辅助对象发送来的信息解包，找出真正被调用的方法以及该方法所在对象；

4)，调用真正服务对象上的真正方法，并将结果返回给服务端辅助对象；

5)，服务端辅助对象将结果打包，网络发送给客户端辅助对象；

6)，客户端辅助对象将返回值解包，返回给客户对象；

7)，客户对象获得返回值；

共同部分：

Person.java:

**import** java.io.Serializable;  
**public class** Person **implements** Serializable {  
 */\*\*  
 \* 序列化ID  
 \*/* **private static final long *serialVersionUID*** = -3245478690496182643L;  
 **public** String **name**;  
 **public int age**;  
 **public** Person(String name, **int** age) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 }  
 **public** String getName() {  
 **return name**;  
 }  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
 **public int** getAge() {  
 **return age**;  
 }  
 **public void** setAge(**int** age) {  
 **this**.**age** = age;  
 }  
}

ImyRemote.java  
**import** java.rmi.Remote;  
**import** java.rmi.RemoteException;**public interface** IMyRemote **extends** Remote {  
 String sayHello() **throws** RemoteException;  
 Person getPerson() **throws** RemoteException;  
}

Server实现：

Server.java:

**import** java.net.MalformedURLException;  
**import** java.rmi.Naming;  
**import** java.rmi.RemoteException;  
  
**public class** Server {  
 **public static void** main(String[] args) {  
 **try**{  
*// LocateRegistry.createRegistry(1099);* IMyRemote iRemote = **new** MyRemoteImpl();  
 Naming.*rebind*(**"rmi://198.181.47.219:1099/remoteServer"**, iRemote);  
 System.***out***.println(**"绑定成功"**);  
 }**catch** (RemoteException e){  
 System.***out***.println(**"RemoteException"**);  
 e.printStackTrace();  
 } **catch** (MalformedURLException e) {  
 System.***out***.println(**"MalformedURLException"**);  
 e.printStackTrace();  
 }  
 }  
}

MyRemoteImpl.java  
**import** java.rmi.RemoteException;  
**import** java.rmi.server.UnicastRemoteObject;  
  
**public class** MyRemoteImpl **extends** UnicastRemoteObject **implements** IMyRemote {  
 **public** MyRemoteImpl() **throws** RemoteException {  
 }  
 @Override  
 **public** String sayHello() {  
 **return "server say,hello world"**;  
 }  
  
 @Override  
 **public** Person getPerson() **throws** RemoteException {  
 **return new** Person(**"张三"**,18);  
 }  
  
}

Cliet实现：

**import** java.net.MalformedURLException;  
**import** java.rmi.Naming;  
**import** java.rmi.NotBoundException;  
**import** java.rmi.RemoteException;  
  
**public class** Client {  
 **public static void** main(String[] args) **throws** RemoteException, NotBoundException, MalformedURLException {  
 IMyRemote client = (IMyRemote) Naming.*lookup*(**"rmi://198.181.47.219:1099/remoteServer"**);  
 System.***out***.println(client.sayHello());  
 Person p = client.getPerson();  
 System.***out***.println(**"name:"** + p.getName() + **"age:"** + p.getAge());  
 }  
}

注：

1. 启动远程服务前，需要先启动rmiregistry(Naming.rebind注册服务，rmiregistry必须运行)；
2. 传输的对象的类型必须成为可序列化类型(编译器无法发现)；

## 5.2 远程过程调用(RPC)

RPC(Remote Procedure Call)支持多语言，而RMI只支持JAVA写的应用程序。RMI运行方法返回JAVA对象及基本数据类型。而RPC不支持对象概念，传送到RPC服务的消息由外部数据表示语言标志(XDR)。

注：RMI和RPC没有可比性，RMI是Java分布式框架(基于TCP/IP)。RPC是网络协议

例子略

## 5.3 Java消息服务(JMS)

(1)JMS（Java Message Service）概念

[Java](https://zh.wikipedia.org/wiki/Java)平台中关于[面向消息中间件](https://zh.wikipedia.org/w/index.php?title=%E9%9D%A2%E5%90%91%E6%B6%88%E6%81%AF%E4%B8%AD%E9%97%B4%E4%BB%B6&action=edit&redlink=1)（MOM）的[API](https://zh.wikipedia.org/wiki/API)，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

JMS包含两种消息模式，点对点和发布者/订阅者。

JMS支持面向事件的方法接收消息(事件驱动的程序设计)。

(2)JMS元素

JMS提供者：连接面向消息中间件的，JMS接口的一个实现。提供者可以是Java平台的JMS实现，也可以是非Java平台的面向消息中间件的适配器。

JMS客户：生产或消费消息的基于Java的应用程序或对象。

JMS生产者：创建并发送消息的JMS客户。

JMS消费者：接收消息的JMS客户。

JMS消息：包括可以在JMS客户之间传递的数据的对象

JMS队列：一个容纳那些被发送的等待阅读的消息的区域。队列暗示，这些消息将按照顺序发送。一旦一个消息被阅读，该消息将被从队列中移走。

JMS主题：一种支持发送消息给多个订阅者的机制。

注:关于JMS实现方式(JMS提供者)activemq，详细参照文件夹mq（消息队列）-->apacheActiveMQ。

1. activemq实现JMS例子





A)点对点模式

生产者：

Connection connection = null;

Session session = null;

MessageProducer messageProducer = null;

try{

//1.创建ConnectionFactory 连接工厂

ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://198.181.47.219:61616");

//2.创建Connection 连接对象

connection = connectionFactory.createConnection();

//开启连接

connection.start();

//3.创建Session会话，该会话用于创建消息生产者、消息消费者和消息等

session = connection.createSession(true,Session.AUTO\_ACKNOWLEDGE);

//4.创建Destination 目的地对象

Destination destination = session.createQueue("jms\_producer\_p2p");

//5.创建消息的生产者

messageProducer = session.createProducer(destination);

//6.创建一条消息

ActiveMQObjectMessage msg = (ActiveMQObjectMessage) session.createObjectMessage();

msg.setObject(new MessageDto("p2p", "我是点对点模式发布者"));

//7.发送消息

messageProducer.send(msg);

//提交事务

session.commit();

}catch(Exception e){

e.printStackTrace();

}finally {

//8.释放资源

messageProducer.close();

session.close();

connection.close();

}

消费者：

Connection connection = null;

Session session = null;

MessageConsumer messageConsumer = null;

try{

//1.创建ConnectionFactory

ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://198.181.47.219:61616");

//2.创建Connection

connection = connectionFactory.createConnection();

connection.start();

//3.创建Session

session = connection.createSession(true,Session.AUTO\_ACKNOWLEDGE);

//4.创建Destination 目的地对象

Destination destination = session.createQueue("jms\_producer\_p2p");

//5.创建消费者

messageConsumer = session.createConsumer(destination);

int number = 0;

//6.消费消息，最多消费五条

while(true){

/\*\*

\* 接收消息，并且指定超时时间为10s阻塞时间，超时返回null

\*/

ActiveMQObjectMessage msg = (ActiveMQObjectMessage) session.createObjectMessage();

msg = (ActiveMQObjectMessage) messageConsumer.receive(10000);

if(null != msg){

number ++ ;

MessageDto dto = (MessageDto) msg.getObject();

System.out.println(dto.toString());

}

if (number > 5)

break;

}

}catch(Exception e){

e.printStackTrace();

}finally {

//7.释放资源

messageConsumer.close();

session.close();

connection.close();

}

B)发布订阅模式

生产者：

Connection connection = null;

Session session = null;

MessageProducer messageProducer = null;

try {

//1.创建ConnectionFactory 连接工厂

ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://198.181.47.219:61616");

//2.创建Connection 连接对象

connection = connectionFactory.createConnection();

//开启连接

connection.start();

//3.创建Session会话，该会话用于创建消息生产者、消息消费者和消息等

session = connection.createSession(true, Session.AUTO\_ACKNOWLEDGE);

//4.创建Destination 目的地对象

Destination destination = session.createTopic("jms\_producer\_pub/sub");

//5.创建消息的生产者

messageProducer = session.createProducer(destination);

//6.创建一条消息

ObjectMessage objMessage = session.createObjectMessage(new MessageDto("pub/sub", "我是主题发布者"));

//7.发送消息

messageProducer.send(objMessage);

//提交事务

session.commit();

} catch (Exception e) {

e.printStackTrace();

}finally {

//8.释放资源

messageProducer.close();

session.close();

connection.close();

}

消费者：

Connection connection = null;

Session session = null;

MessageConsumer messageConsumer = null;

try {

//1.创建ConnectionFactory

ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://198.181.47.219:61616");

//2.创建Connection

connection = connectionFactory.createConnection();

connection.start();

//3.创建Session

session = connection.createSession(true,Session.AUTO\_ACKNOWLEDGE);

//4.创建Destination 目的地对象

Destination destination = session.createTopic("jms\_producer\_pub/sub");

//5.创建消费者

messageConsumer = session.createConsumer(destination);

//6.消费消息

while(true){

ActiveMQObjectMessage msg = (ActiveMQObjectMessage) session.createObjectMessage();

msg = (ActiveMQObjectMessage) messageConsumer.receive(10000);

if(null != msg){

MessageDto dto = (MessageDto) msg.getObject();

System.out.println(dto.toString());

}

}

} catch (Exception e) {

e.printStackTrace();

}finally {

//7.释放资源

messageConsumer.close();

session.close();

connection.close();

}

# 6 数据结构

## 6.1 排序

冒泡排序

选择排序

插入排序

归并排序

快速排序

随机快速排序(random quick sort)

计数排序

基数排序

## 6.2 查找

### 6.2.1 二分查找

/\*\*

\* 二分查找(递归实现)

\*

\* @param array 需查找的数组(有序)

\* @param key 查找的值

\* @param left 下界

\* @param right 上界

\* @return

\* key所在数组的下标

\* 若不存在返回-1

\*\*/

private <T extends Comparable<T>> int search(T array[], T key, int left, int right) {

if (right < left) return -1; // key值不存在数组中

int median = (left + right) >>> 1;

int comp = key.compareTo(array[median]);

if (comp < 0) {

return search(array, key, left, median - 1);

}

if (comp > 0) {

return search(array, key, median + 1, right);

}

return median;

}

/\*\*

\* 二分查找(迭代式实现)

\* @param array 有序的数组

\* @param key 需查找的key值

\* @return 返回key在数组中的下标值

\* 若不存在返回-1

\*/

@Override

public <T extends Comparable<T>> int find(T[] array, T key) {

int l, r, k, cmp;

l = 0;

r = array.length - 1;

while (l <= r) {

k = (l + r) / 2;

cmp = key.compareTo(array[k]);

if (cmp == 0) {

return k;

} else if (cmp < 0) {

r = --k;

} else {

l = ++k;

}

}

return -1;

}

### 6.2.2 插补法查找

/\*\*

\* 插值法查找

\* 利用公式:

\* key-array[start] index - start

\* ------------------------- = -------------

\* array[end] - array[start] end - start

\* 即

\* index = start + (end - start) \* (key-array[start]) / array[end] - array[start]

\* @param array 有序的数组

\* @param key 需查找的key值

\* @return

\* key所在数组中的下标

\* 若不存在返回-1

\*/

public int find(int array[], int key) {

// 设置上界及下界

int start = 0, end = (array.length - 1);

while (start <= end && key >= array[start] && key <= array[end])

{

// 使用公式获取到可能的下标索引值

int pos = start + (((end-start) / (array[end]-array[start]))\*(key - array[start]));

// 若相等，则找到key的下标，直接返回

if (array[pos] == key)

return pos;

// 若该下标对应的数组值比查找值小，则该下标索引 + 1作为下一轮查找的下界

if (array[pos] < key)

start = pos + 1;

// 若该下标对应的数组值比查找值大，则该下标索引 - 1作为下一轮查找的上界

else

end = pos - 1;

}

return -1;

}

### 6.2.3 线性查找

/\*\*

\* 线性查找(顺序查找)

\* @param array 查找的数组(该数组无需有序)

\* @param value 需查找的key值

\* @return

\* 返回key在数组中的下标值

\* 若不存在返回-1

\*/

@Override

public <T extends Comparable<T>> int find(T[] array, T value) {

for (int i = 0; i < array.length ; i++) {

if (array[i].compareTo(value) == 0) {

return i;

}

}

return -1;

}

## 6.3 逻辑数据结构

### 6.3.1 线性表

### 6.3.2 树

(1)堆结构

n个元素序列{k1, k2... ki...kn},当且仅当满足下列关系时称之为堆：

(ki <= k2i, ki <= k2i+1){最小堆}或者(ki >= k2i, ki >= k2i+1){最大堆}, (i = 1, 2, 3, 4... n/2)

注：实为二叉树结构，若i为某一节点，则子节点则为(2i或2i+1)。只是比二叉树结构多了条件限制。

(2)堆和树的区别

堆是一种特别的树。

堆根节点的值为最大(或最小)，只允许有两个节点，并且根节点和叶子节点关系有一定的限制；

树节点间无条件限制，可以有多个节点；

### 6.3.3 哈希表(也叫散列表)

根据键(key)而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数据称做散列表。(维基百科)

冲突：对不同的关键字可能得到同一哈希地址，这种现象称为冲突。实现哈希表这种数据结构的时候不仅要设定一个“好”的哈希函数，而且要设定一种处理冲突的方法。

(1)构造哈希函数

a)直接寻址法

取关键字或关键字的某个线性函数值为散列地址。即hash(k)=k或 hash(k)=a\*k+b，其中a b为常数（这种散列函数叫做自身函数）

b)数学分析法

假设关键字是以r为基的数，并且哈希表中可能出现的关键字都是事先知道的，则可取关键字的若干数组组成哈希地址。

eg:统计班上学生的出生年月日，以年为存储地址发生冲突的概率非常大，而以月日作为存储地址，能够有效降低冲突概率。

c)平方取中法

取关键字平方后的中间几位为哈希地址。通常在选定哈希函数时不一定能知道关键字的全部情况，取其中的哪几位也不一定合适，而一个数平方后的中间几位数和数的每一位都相关，由此使随机分布的关键字得到的哈希地址也是随机的。取的位数由表长决定。

d)折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为哈希地址。

e)随机数法

f)除留余数法

取关键字被某个不大于散列表表长m的数p除后所得的余数为散列地址。即hash(k) = k mod p (p小于m，m为表长)。不仅可以对关键字直接取模，也可在折叠法、平方取中法等运算之后取模。对p的选择很重要，一般取素数或m，若p选择不好，容易产生冲突。

视情况选择；

(2)处理冲突

a)开放定址法

b)单独链表法

将散列到同一个存储位置的所有元素保存在一个链表中。实现时，一种策略是散列表同一位置的所有冲突结果都是用栈存放的，新元素被插入到表的前端还是后端完全取决于怎样方便。

c)再散列：

hash\_{i}=hash\_{i}(key), i=1,2...k。 hash\_{i}是一些散列函数。即在上次散列计算发生冲突时，利用该次冲突的散列函数地址产生新的散列函数地址，直到冲突不再发生。这种方法不易产生“聚集”（Cluster），但增加了计算时间。

(3)查找

一些关键码可通过散列函数转换的地址直接找到，另一些关键码在散列函数得到的地址上产生了冲突，需要按处理冲突的方法进行查找。

查找过程中，关键码的比较次数，取决于产生冲突的多少，产生的冲突少，查找效率就高，产生的冲突多，查找效率就低。因此，影响产生冲突多少的因素，也就是影响查找效率的因素。影响产生冲突多少有以下三个因素：

散列函数是否均匀；

处理冲突的方法；

散列表的载荷因子（英语：load factor）。

a)载荷因子

载荷因子的定义：载荷因子 = 填入表中的元素个数/散列表的长度

载荷因子是散列表装满程度的标志因子。由于表长是定制，装载因子与"填入表中的元素个数"成正比，所以，装载因子越大，表明填入表中的元素越多，产生冲突的可能性就越大；反之，载荷因子越小，表明填入表中的元素越少，产生冲突的可能性就越小。实际上，散列表的平均查找长度是载荷因子的函数，只是不同处理冲突的方法有不同的函数。

注：对于开放定址法，载荷因子是特别重要因素，应严格限制在0.7-0.8以下。超过0.8，查表时的CPU不命中按照指数曲线上升。因此，一些采用开放定址法的hash库，如java的系统限制了载荷因子为0.75，超过此值将resize散列表。

HashMap的实例有两个参数影响其性能：初始容量(threshold)和装载因子(载荷因子，DEFAULT\_LOAD\_FACTOR = 0.75f)。。容量 是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 resize 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

(4)HashMap

HashMap快速存取特性。哈希表是一种逻辑数据结构，HashMap是java中的一种数据类型(结构类型)，实现了哈希表这种数据结构，并定义了一系列操作。

a)HashMap是基于数组来实现哈希表；

b)HashMap的每个记录都是一个Entry<K, V>对象(jdk1.8为Node<K,V>)，数组中存储的就是该对象；

c)HashMap的哈希函数为:

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

注：如果Key值为null，返回0；如果Key值不为空，返回key的hashCode值和key的hashCode值无符号右移16位的值按位异或的结果。

将一个数右移16位，即任何小于2的17次方值-1的数(即考虑低16位都为1的情况)，右移结果都为0；也即任何一个hashCode值小于131071(2的17次方值-1)与0异或都为本身的hashCode值；

若此值大于或等于131071，则进行调整。

d)HashMap解决冲突：使用 单独链表法；

### 6.3.4 链表

(1)链表

是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针(Pointer)。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而顺序表相应的时间复杂度分别是O(logn)和O(1)。--维基百科

链表通常由一连串节点组成，每个节点包含任意的实例数据（data fields）和一或两个用来指向上一个/或下一个节点的位置的链接（"links"）。

(2)单向链表

包含两个域，一个信息域和一个指针域。这个链接指向列表中的下一个节点，而最后一个节点则指向一个空值。

(3)双向链表

每个节点有两个连接：一个指向前一个节点，（当此“连接”为第一个“连接”时，指向空值或者空列表）；而另一个指向下一个节点，（当此“连接”为最后一个“连接”时，指向空值或者空列表）

(4)循环链表

首节点和末节点被连接在一起；

eg:

/\*\*

\* @Description 链表模拟简单范例实现(双向链表，参照jdkLinkedList实现)

\* @ClassName LinkedList

\* @Author Administrator

\* @Data 2018/10/6 23:54

\* @Version 1.0

\*/

public class LinkedList<E> {

private int size; //链表长度

private Node<E> first; //首节点

private Node<E> last; //尾节点

public LinkedList() {

}

/\*\*

\* 节点插入(头)

\*/

public void addFirst(E e) {

final Node<E> f = first;

final Node<E> newNode = new Node<>(null, e, f);

first = newNode;

if (f == null)

last = newNode;

else

f.prev = newNode;

size++;

}

/\*\*

\* 节点插入(尾)

\*/

public void addLast(E e) {

final Node<E> l = last;

final Node<E> newNode = new Node<>(l, e, null);

last = newNode;

if (l == null)

first = newNode;

else

l.next = newNode;

size++;

}

/\*\*

\* 节点信息

\* @param <E>

\*/

private static class Node<E> {

E item;

Node<E> next;

Node<E> prev;

Node(Node<E> prev, E element, Node<E> next) {

this.item = element;

this.next = next;

this.prev = prev;

}

}

/\*\*

\* 将元素输出为数组

\* @return

\*/

public Object[] toArray() {

Object[] result = new Object[size];

int i = 0;

for (Node<E> x = first; x != null; x = x.next)

result[i++] = x.item;

return result;

}

public static void main(String[] args) {

LinkedList<Integer> l = new LinkedList<Integer>();

l.addLast(1);

l.addLast(2);

l.addLast(3);

l.addLast(4);

Object[] array = l.toArray();

for (int i = 0; i < array.length; i ++){

if(i == array.length - 1){

System.out.print(array[i]);

}else{

System.out.print(array[i] + "----->");

}

}

}

}

/\*\*

\* 输出：

\* 1----->2----->3----->4

\*/

(5)链表和数组的比较

数组:数组是一块连续的区域。若增加元素，需要重新定义数组。数组的插入数据和删除数据的效率低，但是随即读取效率高；

链表:链表每个数据都保存着下一个数据的信息，环环相扣。增加数据和删除数据效率高，但查找数据效率低。

(6)ArrayList和LinkedList的比较

ArrayList:内部使用数组的形式实现了存储;

LinkedList:内部使用双向链表的结构实现存储;

综(5)所述，若频繁读取集合中的元素时，使用ArrayList效率高;若插入和删除操作比较多时，使用LinkedList效率较高；

(7)链表逆序

/\*\*

\* 将元素输出为数组(逆序)

\* @return

\*/

public Object[] descToArray(){

Object[] result = new Object[size];

int i = 0;

for (Node<E> x = last; x != null; x = x.prev)

result[i++] = x.item;

return result;

}

# 7 前端相关

## 7.1 JSP

全称JavaServer Pages，中文名叫java服务器页面。JSP技术是以Java语言作为脚本语言的，JSP网页为整个服务器端的Java库单元提供了一个接口来服务于HTTP的应用程序。

JSP本质是Servlet,当用户向指定Servlet发送请求时，Servlet利用输出流动态生成HTML页面，包括每一个静态的HTML标签和所有在HTML页面中出现的内容。

当服务器接收一个客户端请求时，需要做一下四件事情。

(1) 加载和实例化

如果Servlet容器还没实例化一个Servlet对象，此时容器装载和实例化一个 Servlet。创建出该 Servlet 类的一个实例。如果已经存在一个Servlet对象，此时不再创建新实例。

(2) 初始化

在产生 Servlet 实例后，容器负责调用该 Servlet 实例的 init() 方法，在处理用户请求之前，来做一些额外的初始化工作。  
(3) 处理请求

当 Servlet 容器接收到一个 Servlet 请求时，便运行与之对应的 Servlet 实例的 service() 方法，service() 方法根据用户的请求调用相对应的doGet或doPost 方法来处理用户请求。然后再进入对应的方法中调用逻辑层的方法，实现对客户的响应。  
(4) 销毁

当 Servlet 容器决定将一个 Servlet 从服务器中移除时 ( 如 Servlet 文件被更新 )，便调用该 Servlet 实例的 destroy() 方法，在销毁该 Servlet 实例之前，来做一些其他的工作。

其中，(1)(2)(4) 在 Servlet 的整个生命周期中只会被执行一次。

Servlet工作原理流程图：



## 7.2 HTML

HTML（Hypertext Markup Language）文本标记语言，它是静态页面。

与JSP的区别：定义上HTML页面是静态页面可以直接运行，JSP页面是动态页它运行时需要转换成servlet。

## 7.3 Ajax

Asynchronous JavaScript and XML(异步JavaScript和XML)

## 7.4 JavaScript

### 7.4.1 正则表达式(同样适用JAVA)

(1)语法

/正则表达式主体/修饰符(可选)

注：/为正则的边界符 \为将下一个字符标记为一个特殊字符、或一个原义字符

(2)修饰符

i 执行对大小写不敏感的匹配；

g 执行全局匹配(查找所有匹配而非在找到第一个匹配后停止)；

m 执行多行匹配

(3)正则表达式主体

[abc] 查找方括号之间的任何字符。

[0-9] 查找任何从 0 至 9 的数字。

(x|y) 查找任何以 | 分隔的选项。

(4)元字符

\d 查找数字。

\s 查找空白字符。

\b 匹配单词边界。

\uxxxx 查找以十六进制数 xxxx 规定的 Unicode 字符。

(5)量词

n+ 匹配任何包含至少一个 n 的字符串。

n\* 匹配任何包含零个或多个 n 的字符串。

n? 匹配任何包含零个或一个 n 的字符串。

eg:

ex1: “2018||2019”.replace(/\|{2}/g, “至”); //输出 “2018至2019”

# 8 服务器技术

## WebSphere

1.dumb及core文件

javacore.\*\*\*.txt : 关于cpu的，javacore文件是java进程的快照，主要保存的是Java应用各线程在某一时刻的运行的位置，即JVM执行到哪一个类、哪一个方法、哪一行上。也即thread dump文件。

heapdump.\*\*\*.phd : 关于memory的，heapdump文件是指定时刻java堆栈的快照，是一个二进制镜像文件，它保存了某一时刻JVM堆中对象的使用情况。

core.\*\*\*.dmp : core文件是java宕掉生成的操作系统级别的进程二进制镜像文件。又叫核心转储，当程序运行过程中发生异常，程序异常退出时，由操作系统把程序当前的内存状况存储在一个core文件中。

Snap.\*\*\*.trc : snap文件是快速追踪的保留在跟踪缓冲区里的追踪点数据，用来分析本地内存的OOM异常。

2.was日志分析步骤：

(1)系统访问异常；

A)通过浏览器通过URL(http://localhost:9080/myWeb)访问服务器,若请求异常，则通过URL(http://localhost:9080)验证HTTP请求是否正常启动。

B)若通过9080端口不能访问应用程序，可以通过URL(http://localhost:9080/snoop)验证服务器是否存货

(2)应用程序报错问题：

应用日志包含SystemOut.log 、SystemErr.log、activity.log定位应用程序引起的问题。

其中SystemOut.log 、SystemErr.log 属于JVM日志。activity.log为IBM日志。

A)SystemOut.log

用于监控应用程序服务器的运行是否正常。

B)SystemErr.log

包含异常堆栈跟踪信息

C)activity.log

应用程序服务器从各种 WebSphere Application Server 组件的活动创建服务或活动日志文件。服务或活动日志文件（activity.log）是二进制文件，它位于 install\_root 的 logs 目录中，我们可以使用日志分析器用于查看服务或活动日志文件。

注：该日志使用find -name SystemOut.log查询所在位置

## Tomcat

## 知识点

服务器集群

## 负载均衡

将负载（工作任务，访问请求）进行平衡、分摊到多个操作单元（服务器，组件）上进行执行。是解决高性能，单点故障（高可用），扩展性（水平伸缩）的终极解决方案。

负载平衡服务通常是由专用软件和硬件来完成.

### 软件(Nginx反向代理)

异步框架的 Web服务器，也可以用作反向代理，负载平衡器 和 HTTP缓存。

Nginx集中策略：

1)轮询，根据Nginx配置文件中的顺序，依次把客户端的Web请求分发到不同的后端服务端。

2)最少连接，当前谁连接最少，分发给谁。

3)IP地址哈希：确定相同IP请求可以转发给同一个后端节点处理，以方便session保持；

4)基于权重的负载均衡：配置Nginx把请求更多地分发到高配置的后端服务器上，把相对较少的请求分发到低配服务器。

Nginx搭建。Eg:

Nginx安装及配置为简单的文件服务器

A)安装

网上有教程。

B)进入安装目录

../nginx/sbin/

C)执行命令

./nginx 开启

./nginx -s stop 停止

./nginx -s quit

./nginx -s reload 重新启动

D)修改后重新启动

/usr/local/webserver/nginx/sbin/nginx -t -c /usr/local/webserver/nginx/conf/nginx.conf

注：

a、停止naginx

ps -ef|grep nginx (查看进程号)

b、杀死进程

kill -QUIT \*\*\*

pkill -9 nginx (强制停止)

查看端口是否占用:

netstat -lnp|grep 80

### 硬件(F5)

流程图(摘抄至网上)：

1、工作原理；

(1)c端向server发出HTTP/HTTPS请求；

(2)BIGIP接收到请求，将数据包中的目的IP地址改为选中的后台server IP地址，然后将数据包发出至后台特定服务器上；

(3)后台server接收到数据包后,将应答包按照其路由发回BIGIP;

(4)BIGIP收到应答包后将其中的源地址改回成c端请求地址，并按照其路由返回；

2、功能:

(1)BIGIP可以确认程序是否能否对请求返回对应的数据(实时监测服务器健康);

(2)BIGIP具有动态session的会话保持功能；

(3)BIGIP的iRules功能可HTTP内容过滤，根据不同域名、URL,将请求发送到不同服务器上；

3、算法:

(1)轮询:顺序循环将请求一次顺序循环地连接每个服务器。 当其中某个服务器发生第二到第7层的故障，BIGIP就把其从顺序循环队列中拿出，不参加下一次的轮询，直到其恢复正常。

(2)优先权:给所有服务器分组，给每个组定义优先权，BIGIP用户的请求，分配给优先级最高的服务器组（在同一组内，采用轮询或比率算法，分配用户的请求）；当最高优先级中所有服务器出现故障，BIGIP才将请求送给次优先级的服务器组。

(3)最少连接方式：传递新的连接给那些进行最少连接处理的服务器。

(4)最快模式：传递连接给那些响应最快的服务器；

....

4、软件负载均衡(Nginx)及硬件负载均衡(F5)优缺点:

(1)Nginx:基于系统与应用的负载均衡，能够更好地根据系统与应用的状况来分配负载。负载能力受服务器本身性能的影响，性能越好，负载能力越大。

(2)F5:能够直接通过智能交换机实现,处理能力更强，而且与系统无关，负载性能强，更适用于一大堆设备、大访问量、简单应用。成本高。

## 代理服务器

客户机在发送请求时，不会直接发送给目的主机，而是先发送给代理服务器，代理服务接受客户机请求之后，再向主机发出，并接收目的主机返回的数据，存放在代理服务器的硬盘中，再发送给客户机。

使用代理服务器的用处:

(1)提高访问速度；

(2)防火墙作用；

(3)通过代理服务器访问不能访问的目标站点(翻墙)；

(一)正向代理

架设在客户机与目标主机之间，只用于代理内部网络对Internet的连接请求，客户机必须指定代理服务器,并将本来要直接发送到Web服务器上的http请求发送到代理服务器中。

(二)反向代理

服务器架设在服务器端，通过缓冲经常被请求的页面来缓解服务器的工作量，将客户机请求转发给内部网络上的目标服务器；并将从服务器上得到的结果返回给Internet上请求连接的客户端，此时代理服务器与目标主机一起对外表现为一个服务器。

可以防止外网对内网服务器的恶性攻击、缓存以减少服务器的压力和访问安全控制之外，还可以进行负载均衡，将用户请求分配给多个服务器。

# 9 Linux知识

## 9.1 Linux标准输入、标准输出、标准错误与重定向介绍

当Linux执行一个程序的时候，会自动打开三个流，标准输入(standard input)，标准输出(standard output)，标准错误(standard error)。比如说你打开命令行的时候，默认情况下，命令行的标准输入连接到键盘，标准输出和标准错误都连接到屏幕。

步骤一：屏幕敲入 $ ls

步骤二：键盘敲击的文本流("ls\n"，\n是回车时输入的字符，表示换行)命令行 (命令行实际上也是一个程序)。命令行随后调用/bin/ls得到结果("a.txt")，最后这个输出的文本流("a.txt")流到屏幕，显示出来

步骤三：假设说我们不想让文本流流到屏幕，而是流到另一个文件，我们可以采用重新定向(redirect)的机制。

(1)命令：$ls > a.txt

注：新建文件a.txt并将ls的命令行的标准输出指向这个文件。



(2)命令：$ls >> a.txt

注：若文件a.txt不存在，则创建并将ls的命令行的标准输出指向这个文件；若文件a.txt存在，则直接将ls的命令行的标准输出追加至该文件内容后面。



(3) 命令：$cat < a.txt (类似cat.txt)

注：将cat标准输入指向a.txt，文本会从文件流到cat，然后再输出到屏幕上。



(4)命令：$cat < a.txt > b.txt

注：a.txt的内容就复制到了b.txt

## 9.2 常用命令

### 9.2.1 rm

1.rm -rf \* 删除[当前目录](https://www.baidu.com/s?wd=%E5%BD%93%E5%89%8D%E7%9B%AE%E5%BD%95&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvnjDvujDvnWR1PjDkPHuW0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHT1PWDkn1cY)下的所有文件,这个命令很危险，应避免使用。

所删除的文件，一般都不能恢复！

2.rm -f 其中的，f参数 （f --force ） 忽略不存在的文件，不显示任何信息

不会提示确认信息。

3.rm 命令删除一个目录中的一个或者多个文件或者目录，只用rm命令不会

删除目录，通常文件通常可以恢。

4.其他参数命令：

    -f, --force    忽略不存在的文件，从不给出提示。

    -i, --interactive 进行交互式删除

    -r, -R, --recursive   指示rm将参数中列出的全部目录和子目录均递归地删除。

    -v, --verbose    详细显示进行的步骤

       --help     显示此帮助信息并退出

       --version  输出版本信息并退出

友情提示：一般的删除操作用rm 即可 。

### 9.2.2 mkdir和touch

新建目录，mkdir 路径加目录名称。

新建文件，touch 文件名

### 9.2.3 tar

压缩解压缩命令

-c: 建立压缩档案  
-x：解压  
-t：查看内容  
-r：向压缩归档文件末尾追加文件  
-u：更新原压缩包中的文件

这五个是独立的命令，压缩解压都要用到其中一个，可以和别的命令连用但只能用其中一个。下面的参数是根据需要在压缩或解压档案时可选的。  
  
-z：有gzip属性的  
-j：有bz2属性的  
-Z：有compress属性的  
-v：显示所有过程  
-O：将文件解开到标准输出

下面的参数-f是必须的

-f: 使用档案名字，切记，这个参数是最后一个参数，后面只能接档案名。

**解压**

tar -xvf file.tar //解压 tar包

tar -xzvf file.tar.gz //解压tar.gz

tar -xjvf file.tar.bz2   //解压 tar.bz2

tar -xZvf file.tar.Z   //解压tar.Z

unrar e file.rar //解压rar

unzip file.zip //解压zip

**总结**

1、\*.tar 用 tar -xvf 解压

2、\*.gz 用 gzip -d或者gunzip 解压

3、\*.tar.gz和\*.tgz 用 tar -xzf 解压

4、\*.bz2 用 bzip2 -d或者用bunzip2 解压

5、\*.tar.bz2用tar -xjf 解压

6、\*.Z 用 uncompress 解压

7、\*.tar.Z 用tar -xZf 解压

8、\*.rar 用 unrar e解压

9、\*.zip 用 unzip 解压

### 9.2.4 vim

vim分为三种模式，命令模式(首先进来)、编辑模式(可输入)、底线命令模式(命令模式下输入:即可进入)

1快速定位文件头及文件尾

底线命令模式 :1 定位至文件首 :$ 定位至文件尾

命令模式 ctrl+F 向下翻页 ctrl+B 向上翻页

2保存文件或退出vi

底线命令模式下操作:

|  |  |
| --- | --- |
| 命令 | 简单说明 |
| :w | 保存编辑后的文件内容，但不退出vim编辑器。这个命令的作用是把内存缓冲区中的数据写到启动vim时指定的文件中。 |
| :w! | 强制写文件，即强制覆盖原有文件。如果原有文件的访问权限不允许写入文件，例如，原有的文件为只读文件，则可使用这个命令强制写入。但是，这种命令用法仅当用户是文件的属主时才适用，而超级用户则不受此限制。 |
| :wq | 保存文件内容后退出vim编辑器。这个命令的作用是把内存缓冲区中的数据写到启动vim时指定的文件中，然后退出vim编辑器。另外一种替代的方法是用ZZ命令。 |
| :wq! | 强制保存文件内容后退出vim编辑器。这个命令的作用是把内存缓冲区中的数据强制写到启动vim时指定的文件中，然后退出vim编辑器。 |
| ZZ | 使用ZZ命令时，如果文件已经做过编辑处理，则把内存缓冲区中的数据写到启动vim时指定的文件中，然后退出vim编辑器。否则只是退出vim而已。注意，ZZ命令前面无需加冒号“：”，也无需按Enter键。 |
| :q | 在未做任何编辑处理而准备退出vim时，可以使用此命令。如果已做过编辑处理，则vim不允许用户使用“:q”命令退出，同时还会输出下列警告信息：  No write since last change (:quit! overrides) |
| :q! | 强制退出vim编辑器，放弃编辑处理的结果。如果确实不需要保存修改后的文件内容，可输入“:q!”命令，强行退出vim编辑器。 |
| :w filename | 把编辑处理后的结果写到指定的文件中保存 |
| :w! filename | 把编辑处理后的结果强制保存到指定的文件中，如果文件已经存在，则覆盖现有的文件。 |
| :wq! filename | 把编辑处理后的结果强制保存到指定的文件中，如果文件已经存在，则覆盖现有文件，并退出vim编辑器。 |

3查找内容

底线命令模式 : /string查找 注:找到后继续查找：n 向下继续搜索 N 反方向继续搜索

### 9.2.5 mv、cp和rm

命令行下的移动、复制与删除：mv ,cp,rm,

(1)mv

语法：mv(选项)(参数)

选项：

--backup=<备份模式>：若需覆盖文件，则覆盖前先行备份；

-b：当文件存在时，覆盖前，为其创建一个备份；

-f：若目标文件或目录与现有的文件或目录重复，则直接覆盖现有的文件或目录；

-i：交互式操作，覆盖前先行询问用户，如果源文件与目标文件或目标目录中的文件同名，则询问用户是否覆盖目标文件。用户输入”y”，表示将覆盖目标文件；输入”n”，表示取消对源文件的移动。这样可以避免误将文件覆盖。

--strip-trailing-slashes：删除源文件中的斜杠“/”；

-S<后缀>：为备份文件指定后缀，而不使用默认的后缀；

--target-directory=<目录>：指定源文件要移动到目标目录；

-u：当源文件比目标文件新或者目标文件不存在时，才执行移动操作。

参数：

源文件：源文件列表。

目标文件：如果“目标文件”是文件名则在移动文件的同时，将其改名为“目标文件”；如果“目标文件”是目录名则将源文件移动到“目标文件”下。

eg:

mv -fi download test

(2)cp

语法：cp(选项)(参数)

选项：

-a:相当于-pdr的意思，请参考后文  
         -d:若源文件为链接文件的属性，则复制链接文件属性而非文件本身  
         -f:强制的意思（force）,也就是说覆盖掉已存在的文件

参数：

源文件：源文件列表。

目标文件：如果“目标文件”是文件名则在移动文件的同时，将其改名为“目标文件”；如果“目标文件”是目录名则将源文件移动到“目标文件”下。

eg:

(3)rm

语法： rm (选项) (参数)

选项：

   -r:递归操作  
         -f:强制(force)  
         -i:删除时给出提示信息

参数：

文件或者目录

eg:

### 9.2.6 sudo

### 9.2.7 find和grep

查找命令, 两者的区别：

(1)find命令是根据文件的属性进行查找如文件名，文件大小，所有者，所属组，是否为空，访问时间，修改时间等

(2)grep是根据文件的内容进行查找，会对文件的每一行按照给定的模式(patter)进行匹配查找。

一find命令

格式 find  (path) (expression)

1按照文件名查找

(1)find / -name httpd.conf　#在根目录下查找文件httpd.conf，表示在整个硬盘查找  
　　(2)find /etc -name httpd.conf　#在/etc目录下文件httpd.conf  
　　(3)find /etc -name '\*srm\*'　#使用通配符\*(0或者任意多个)。表示在/etc目录下查找文件名中含有字符串‘srm’的文件  
　　(4)find . -name 'srm\*' #表示当前目录下查找文件名开头是字符串‘srm’的文件

2按照文件特征查找

(1)find / -amin -10 　　# 查找在系统中最后10分钟访问的文件(access time)  
　　　(2)find / -atime -2　　 # 查找在系统中最后48小时访问的文件  
　　　(3)find / -empty 　　# 查找在系统中为空的文件或者文件夹  
　　　(4)find / -group cat 　　# 查找在系统中属于 group为cat的文件  
　　　(5)find / -mmin -5 　　# 查找在系统中最后5分钟里修改过的文件(modify time)  
　　　(6)find / -mtime -1 　　#查找在系统中最后24小时里修改过的文件  
　　　(7)find / -user fred 　　#查找在系统中属于fred这个用户的文件  
　　　(8)find / -size +10000c　　#查找出大于10000000字节的文件(c:字节，w:双字，k:KB，M:MB，G:GB)  
　　　(9)find / -size -1000k 　　#查找出小于1000KB的文件

二 grep命令

格式：grep expression

(1)grep 'test' d\*　　#显示所有以d开头的文件中包含 test的行  
(2)grep ‘test’ aa bb cc 　　 #显示在aa，bb，cc文件中包含test的行

(3)grep ‘[a-z]\{5\}’ aa 　　#显示所有包含每行字符串至少有5个连续小写字符的字符串的行

(4)grep magic /usr/src　　#显示/usr/src目录下的文件(不含子目录)包含magic的行

(5)grep -r magic /usr/src　　#显示/usr/src目录下的文件(包含子目录)包含magic的行

(6)grep -w pattern files ：只匹配整个单词，而不是字符串的一部分(如匹配’magic’，而不是’magical’)，

### 9.2.8 ls

显示目录列表。格式：ls (选项) (参数)

选项:

-a：显示所有档案及目录（ls内定将档案名或目录名称为“.”的视为影藏，不会列出）；

-A：显示除影藏文件“.”和“..”以外的所有文件列表；

-C：多列显示输出结果。这是默认选项；

-l：与“-C”选项功能相反，所有输出信息用单列格式输出，不输出为多列；

-F：在每个输出项后追加文件的类型标识符，具体含义：“\*”表示具有可执行权限的普通文件，“/”表示目录，“@”表示符号链接，“|”表示命令管道FIFO，“=”表示sockets套接字。当文件为普通文件时，不输出任何标识符；

-b：将文件中的不可输出的字符以反斜线“”加字符编码的方式输出；

-c：与“-lt”选项连用时，按照文件状态时间排序输出目录内容，排序的依据是文件的索引节点中的ctime字段。与“-l”选项连用时，则排序的一句是文件的状态改变时间；

-d：仅显示目录名，而不显示目录下的内容列表。显示符号链接文件本身，而不显示其所指向的目录列表；

-f：此参数的效果和同时指定“aU”参数相同，并关闭“lst”参数的效果；

-i：显示文件索引节点号（inode）。一个索引节点代表一个文件；

参数：

目录：指定要显示列表的目录，也可以是具体的文件。

eg：

### 9.2.9 stat

查看文件状态,包含文件信息 访问时间，修改时间。

eg：

stat debug.log 查看debug.log的文件状态。

### 9.2.10 free和top

查看内存使用情况。

### 9.2.11 ps -ef |grep和kill

ps -ef |grep redis 查找redis的进程；

kill -9 PID 杀死进程；

### 9.2.12 iptables

### 9.2.13 cat

文本输出命令，通常是用于观看某个文件的内容的.

cat主要有三大功能：

命令格式：

cat [-AbeEnstTuv] [--help] [--version] fileName

常用命令：

(1)一次显示整个文件;

$ cat   filename

(2)从键盘创建一个文件;

$ cat  >  filename

$ cat >> filename

(3)将几个文件合并为一个文件。

$cat   file1   file2  > file



(4) 以EOF输入字符为标准输入结束：

$cat<<EOF



(5) 以EOF输入字符为标准输入,并追加至filename文件后面

$cat>filename<<EOF



cat << EOF > test.sh



### 9.2.14 lsof

(1)lsof –i 显示符合条件的进程情况



(2)lsof –i:端口号 查询该端口占用情况



### 9.2.15 **netstat**

打印Linux中网络系统的状态信息。

netstat -a #列出所有端口

netstat -[at](http://man.linuxde.net/at) #列出所有tcp端口

netstat -au #列出所有udp端口

netstat -l #只显示监听端口

netstat -lt #只列出所有监听 tcp 端口

netstat -lu #只列出所有监听 udp 端口

netstat -lx #只列出所有监听 UNIX 端口

### 9.2.16 wget

wget(选项)(参数)

(1)wget <https://archive.apache.org/dist/activemq/5.14.0/apache-activemq-5.14.0-bin.tar.gz>

通过url下载apache-activemq-5.14.0-bin.tar.gz至当前目录

(2) wget -O activemq.tar.gz <https://archive.apache.org/dist/activemq/5.14.0/apache-activemq-5.14.0-bin.tar.gz>

下载activemq，并重命名

1. [tail](http://man.linuxde.net/tail) -f wget-log 查看下载进度

查看下载进度

### 9.2.17 echo

echo [选项] 字符串

例如：echo -n “hello,world” 注:-n 不要在最后自动换行

在显示器上显示一段文字。

### 9.2.18 In

语法：ln [option] source\_file dist\_file

ln abc cde 建立abc的硬连接

ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime创建软链接到localtime

## 9.3上传文件至linux

1. 通过Filezilla工具，建立SSH连接，通过拖拉方式上传，比较方便；
2. 通过SFTP命令上传，自行百度；

## 9.4 安装工具

安装GCC

　　 命令：#yum install gcc gcc-c++

  安装wget网页下载工具

　　 命令：#yum install wget

# 10 分布式知识

## 10.1 分布式缓存

### 10.1.1缓存的定义(数据库层面)

数据交换的缓冲区，数据库在本地内存中保留一个镜像，下次访问的时候只需要从内存中直接获取。

一级缓存：缓存放在session上，属于事务级数据缓冲；

二级缓存：缓存存放在SessionFactory，所有的Session共享同一个二级Cache

### 10.1.2 缓存分类(ehcache)

(1)本地缓存(Ehcache、Guava)：

客户机本地的物理内存划分出一部分空间用来缓冲客户机回写到服务器的数据。

(2)集群缓存(Ehcache集群)：

多台服务器间每个节点独立存储相同的，全部的数据源(存在不同步的风险)。

(3)分布式缓存(Ehcache分布式)：

缓存服务间每个节点存储部分数据源（所有数据源组成服务所有数据源）；

(注：集群与分布式区别：分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。)

### 10.1.3 缓存服务策略

更新策略

(1)Cache Aside模式

先查询缓存数据是否存在，若不存在，读取数据库，将其缓存；

(2)Read/Write Through 模式

应用要读数据和更新数据都直接访问缓存服务，缓存服务同步的将数据更新到数据库；

(3)Write Behind 模式

应用要读数据和更新数据都直接访问缓存服务,缓存服务异步的将数据更新到数据库（通过异步任务）；

缓存淘汰策略

(1)LRU

最近最少使用，缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方来缓存新的元素的时候，那么现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

(2)LFU

最少被使用，缓存的元素有一个hit属性，hit值最小的将会被清出缓存。

(3)FIFO

先进先出。

### 10.1.4 ehcache缓存实现原理及机制

参考：https://www.jianshu.com/p/5a0669d6305e

特性:

快速简单；

多种缓存策略( LRU，LFU，FIFO)；

内存和磁盘缓存数据；

利用RMI、JMS、Jgroup等实现分布式缓存；

具有缓存和缓存管理器的侦听接口；  
支持多缓存管理器实例，以及一个实例的多个缓存区域；  
提供 Hibernate 的缓存实现；

(1)集群实现方式：(a)Terracotta;(b)RMI;(c)JMS;(d)JGroup;(e)Ehcache Server。

(2)如何配置集群?

A)服务提供者

1)自动发现机制；

cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,

multicastGroupPort=4446, timeToLive=32"/>

<!-- timeToLive

0是限制在同一个服务器

1是限制在同一个子网

32是限制在同一个网站

64是限制在同一个region

128是限制在同一个大洲

255是不限制-->

2)手动配置机制；

<!-- server1 -->

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,

rmiUrls=//server2:40001/sampleCache11|//server2:40001/sampleCache12"/>

<!-- server2 -->

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,

rmiUrls=//server1:40001/sampleCache11|//server1:40001/sampleCache12"/>

B)服务监听

<cacheManagerPeerListenerFactory  
 class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"  
 properties="hostName=192.168.1.101,

port=40001,socketTimeoutMillis=2000" />

C)事件监听器

replicatePuts=true | false – 当一个新元素增加到缓存中的时候是否要复制到其他的peers. 默认是true。

replicateUpdates=true | false – 当一个已经在缓存中存在的元素被覆盖时是否要进行复制。默认是true。

replicateRemovals= true | false – 当元素移除的时候是否进行复制。默认是true。

replicateAsynchronously=true | false – 复制方式是异步的（指定为true时）还是同步的（指定为false时）。默认是true。

replicatePutsViaCopy=true | false – 当一个新增元素被拷贝到其他的cache中时是否进行复制指定为true时为复制，默认是true。

<**cacheEventListenerFactory  
 class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"  
 properties="replicateAsynchronously=true, replicatePuts=true, replicateUpdates=true,  
 replicateUpdatesViaCopy=false, replicateRemovals=true,asynchronousReplicationIntervalMillis=1000"**/>

eg(代码添加)：

Configuration cf = **new** Configuration();  
 *//ehcahce 名称(CacheManager)* cf.setName(**"testCache"**);  
 *//ehcache的磁盘存储路径* cf.diskStore(**new** DiskStoreConfiguration().path(**"java.io.tmpdir"**));  
 */\*\*  
 \* 默认配置，其中包含  
 \* name:缓存名称。  
 \* maxElementsInMemory：缓存最大个数。  
 \* eternal:对象是否永久有效，一但设置了，timeout将不起作用。  
 \* timeToIdleSeconds：设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。  
 \* timeToLiveSeconds：设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。  
 \* overflowToDisk：当内存中对象数量达到maxElementsInMemory时，Ehcache将会对象写到磁盘中  
 \*/* CacheConfiguration defaultCf = **new** CacheConfiguration();  
 defaultCf.setMaxElementsInMemory(500);  
 defaultCf.setEternal(**false**);  
 defaultCf.setTimeToIdleSeconds(300);  
 defaultCf.setTimeToLiveSeconds(1200);  
 defaultCf.setOverflowToDisk(**true**);  
 cf.setDefaultCacheConfiguration(defaultCf);  
 */\*\*  
 \*  
 \*/* FactoryConfiguration fcf = **new** FactoryConfiguration();  
 */\*\*  
 \* 节点方式发现类  
 \*/* fcf.setClass(**"net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"**);  
 */\*\*  
 \* 节点发现属性  
 \* peerDiscovery 方式：atutomatic 为自动 manual 手动方式  
 \* 若为自动方式的时候配置  
 \* mulicastGroupAddress 广播组地址：230.0.0.1  
 \* mulicastGroupPort 广播组端口：40001  
 \* timeToLive是指搜索范围：0是同一台服务器，1是同一个子网，32是指同一站点，64是指同一块地域，128是同一块大陆，还有个256，我就不说了  
 \* hostName：主机名或者ip，用来接受或者发送信息的接口  
 \*  
 \* 若为手动方式的时候配置  
 \*  
 \*/  
// fcf.setProperties("peerDiscovery=automatic, multicastGroupAddress=230.0.0.1," +  
// "multicastGroupPort=4446, timeToLive=32,hostName=192.168.1.101");* fcf.setProperties(**"peerDiscovery=manual, rmiUrls=//170.20.10.2:40001/userCache"**);  
 cf.addCacheManagerPeerProviderFactory(fcf);  
 */\*\*  
 \* 配置监听本机40001端口  
 \*/* fcf = **new** FactoryConfiguration();  
 fcf.setClass(**"net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"**);  
 fcf.setProperties(**"hostName=198.181.47.219, port=40001,socketTimeoutMillis=2000"**);  
 cf.addCacheManagerPeerListenerFactory(fcf);  
 *//创建CacheManager对象* CacheManager cm = **new** CacheManager(cf);  
 */\*\*  
 \* 配置一个缓存  
 \* name 缓存名称  
 \* maxEntriesLocalHeap 允许在内存中存放元素的最大数量  
 \* eternal 是否永久(true 不过期过期)  
 \* timeToIdleSeconds 单位是秒，表示一个元素所允许闲置的最大时间，也就是说一个元素在不被请求的情况下允许在缓存中待的最大时间。默认是0，表示不限制  
 \* timeToLiveSeconds 单位是秒，表示无论一个元素闲置与否，其允许在Cache中存在的最大时间。默认是0，表示不限制。  
 \* overflowToDisk 当内存里面的缓存已经达到预设的上限时是否允许将按驱除策略驱除的元素保存在硬盘上。  
 \*/* CacheConfiguration cacheConfig = **new** CacheConfiguration();  
 cacheConfig.setName(**"userCache"**);  
 cacheConfig.setMaxMemoryOffHeap(**"100"**);  
 cacheConfig.setEternal(**true**);  
 cacheConfig.setTimeToIdleSeconds(10000);  
 cacheConfig.setTimeToLiveSeconds(10000);  
 cacheConfig.setOverflowToDisk(**false**);  
 */\*\*  
 \* 配置监听  
 \*/* CacheConfiguration.CacheEventListenerFactoryConfiguration c = **new** CacheConfiguration.CacheEventListenerFactoryConfiguration();  
 c.setClass(**"net.sf.ehcache.distribution.RMICacheReplicatorFactory"**);  
 c.setProperties(**"replicateAsynchronously=true, replicatePuts=true, replicateUpdates=true,replicateUpdatesViaCopy= false, replicateRemovals=true"**);  
 cacheConfig.addCacheEventListenerFactory(c);  
 Cache cache = **new** Cache(cacheConfig);  
 cm.addCache(cache);

## 负载均衡(nginx,读音:engine)

## 反向代理

## 静态内容缓存

## 访问控制

# 11 网络编程(计算机网络)

## 11.1 TCP/IP协议

Transmission Control Protocol/Internet Protocol的简写，中译名为传输控制协议/[因特网](https://baike.baidu.com/item/%E5%9B%A0%E7%89%B9%E7%BD%91)互联协议，又名网络[通讯协议](https://baike.baidu.com/item/%E9%80%9A%E8%AE%AF%E5%8D%8F%E8%AE%AE)，是Internet最基本的协议、Internet国际[互联网](https://baike.baidu.com/item/%E4%BA%92%E8%81%94%E7%BD%91)络的基础，由[网络层](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%B1%82)的IP协议和[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82)的TCP协议组成。TCP/IP 定义了电子设备如何连入[因特网](https://baike.baidu.com/item/%E5%9B%A0%E7%89%B9%E7%BD%91)，以及数据如何在它们之间传输的标准。协议采用了4层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。

TCP/IP参考模型分为四个层次：应用层、传输层、网络互连层和主机到网络层。





## 11.2 OSI模型

开放系统互连参考模型 (Open System Interconnect 简称OSI）是国际标准化组织(ISO)和国际电报电话咨询委员会(CCITT)联合制定的开放系统互连参考模型，为开放式互连信息系统提供了一种功能结构的框架。它从低到高分别是：[物理层](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%B1%82)、[数据链路层](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%93%BE%E8%B7%AF%E5%B1%82)、[网络层](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%B1%82)、[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82)、会话层、[表示层](https://baike.baidu.com/item/%E8%A1%A8%E7%A4%BA%E5%B1%82)和[应用层](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E5%B1%82)。

## 11.3 TCP/IP与OSI比较



## 11.4 socket编程

套接字（socket）是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。

### 11.4.1 Socket编程模型

# 12 前沿技术

## 12.1 区块链

### 12.1.1 区块链技术

区块链是一种比较特殊的分布式数据库。

(1)特点:

(A)去中心化：存在中心点，即类似银行、支付宝、中介等第三方；

(B)开放性：系统数据是公开透明的；

(C)自治性：区块链采用基于协商一致的规范和协议(公开透明的算法)；

(D)信息不可篡改：信息存储到区块链中被永久保存，无法改变；

(E)匿名性：个人信息都被加密；

(2)区块结构

(A)区块头(Head)：记录当前区块的元信息，比如：前一区块的根散列、Merkle树根散列(Hash Tree,存储Hash的一颗树)、时间戳、其他等信息；

(B)区块主体(Body)：实际数据，交易记录信息；

(3)工作原理(以记账为例)

a)拥有第一条账本本记录，该记录包含了交易详细信息；

b)将第一条记录利用哈希函数(哈希方法略)计算得到一个定长的hash值；

c)hash值、时间戳、交易记录组成第一个区块；

d)比特币系统里约10分钟记一次账，即每个区块生成时间大概间隔10分钟；

e)上一个块的Hash值和账本本记录一起作为原始信息进行Hash得到一个定长的hash;

f)该hash值、时间戳、交易记录组成第2个区块；

g)...

以此类推。

(4)比特币挖矿(与5、6点结合)

记账工作的过程便是挖矿，节点记账成功，会获取奖励。

记账时，节点需要的准备工作:

a)收集广播中还没有被记录账本的原始交易信息；

b)检查每个交易信息中付款地址有没有足够的余额；

c)验证交易是否有正确的签名；

d)把验证通过的交易信息进行打包记录；

e)添加一个奖励交易：给自己的地址增加12.5比特币。

(5)工作量证明:

为了保证10分钟左右只有一个人可以记账，就必须要提高记账的难度，使得Hash的结果必须以若干个0开头。同是为了满足这个条件，在进行Hash时引入一个随机数变量。

用伪代码表示一下：

Hash(上一个Hash值，交易记录集) = 456635BCD

Hash(上一个Hash值，交易记录集，随机数) = 0000aFD635BCD

我们知道改变Hash的原始信息的任何一部分，Hash值也会随之不断的变化，因此在运算Hash时，不断的改变随机数的值，总可以找的一个随机数使的Hash的结果以若干个0开头（下文把这个过程称为猜谜），率先找到随机数的节点就获得此次记账的唯一记账权。

(6)验证

在节点成功找到满足的Hash值之后，会马上对全网进行广播打包区块，网络的节点收到广播打包区块，会立刻对其进行验证。

如果验证通过，则表明已经有节点成功解迷，自己就不再竞争当前区块打包，而是选择接受这个区块，记录到自己的账本中，然后进行下一个区块的竞争猜谜。

网络中只有最快解谜的区块，才会添加的账本中，其他的节点进行复制，这样就保证了整个账本的唯一性。

假如节点有任何的作弊行为，都会导致网络的节点验证不通过，直接丢弃其打包的区块，这个区块就无法记录到总账本中，作弊的节点耗费的成本就白费了，因此在巨大的挖矿成本下，也使得矿工自觉自愿的遵守比特币系统的共识协议，也就确保了整个系统的安全。

参考https://learnblockchain.cn/2017/11/04/bitcoin-pow/

### 12.1.2 区块链节点

区块链节点具备着路由、区块链存储、挖矿和钱包四个模块。

(1)路由（network routing）

路由模块承担了发现和维持对等节点的连接、以及广播和接纳新区块的功能实现。理论上，路由模块应该是所有节点都应该具备的模块，否则单一的节点无法参与完整区块链网络的共识和协同。当然也有例外，比如矿池模型中就没有路由模块，但它却十分依赖路由的一些核心协议。

(2)区块链存储（full blockchain database）：

区块链存储模块负责保存完整的、最新的区块链副本，本质上就是数据。

(3)挖矿（miner）

在比特币的区块链网络上，挖矿模块的任务是通过在某种特殊的硬件设备上执行POW共识算法，以相互竞争的方式创建新的区块。其他平台链上也存在类似的概念，可能基于的硬件或者执行的共识算法不一样等等。

(4)钱包（wallet）

钱包模块主要功能是用于管理节点的私钥、资产以及进行交易行为等。

节点的分类

参考https://zhuanlan.zhihu.com/p/39350350

### 12.1.3 Merkle Tree

Merkle Tree，通常也被称作Hash Tree，顾名思义，就是存储hash值的一棵树。Merkle树的叶子是数据块(例如，文件或者文件的集合)的hash值。非叶节点是其对应子节点串联字符串的hash。

(1)哈希函数(hash)

具体参考数据接口-哈希部分(比特币共识算法略)。

(2)Hash List

在点对点网络中作数据传输的时候，会同时从多个机器上下载数据，而且很多机器可以认为是不稳定或者不可信的。为了校验数据的完整性，更好的办法是把大的文件分割成小的数据块（例如，把分割成2K为单位的数据块）。这样的好处是，如果小块数据在传输过程中损坏了，那么只要重新下载这一快数据就行了，不用重新下载整个文件。

怎么确定小的数据块没有损坏哪？只需要为每个数据块做Hash。BT下载的时候，在下载到真正数据之前，我们会先下载一个Hash列表。那么问题又来了，怎么确定这个Hash列表本事是正确的哪？答案是把每个小块数据的Hash值拼到一起，然后对这个长字符串在作一次Hash运算，这样就得到Hash列表的根Hash(Top Hash or Root Hash)。下载数据的时候，首先从可信的数据源得到正确的根Hash，就可以用它来校验Hash列表了，然后通过校验后的Hash列表校验数据块。

(3)Merkle Tree

Merkle Tree可以看做Hash List的泛化（Hash List可以看作一种特殊的Merkle Tree，即树高为2的多叉Merkle Tree）。

在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就结婚生子，得到了一个”子哈希“。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做 Merkle Root.

在p2p网络下载网络之前，先从可信的源获得文件的Merkle Tree树根。一旦获得了树根，就可以从其他从不可信的源获取Merkle tree。通过可信的树根来检查接受到的Merkle Tree。如果Merkle Tree是损坏的或者虚假的，就从其他源获得另一个Merkle Tree，直到获得一个与可信树根匹配的Merkle Tree。

Merkle Tree和Hash List的主要区别是，可以直接下载并立即验证Merkle Tree的一个分支。因为可以将文件切分成小的数据块，这样如果有一块数据损坏，仅仅重新下载这个数据块就行了。如果文件非常大，那么Merkle tree和Hash list都很到，但是Merkle tree可以一次下载一个分支，然后立即验证这个分支，如果分支验证通过，就可以下载数据了。而Hash list只有下载整个hash list才能验证。

(4)Merkle Tree特点

A)MT是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；

B)Merkle Tree的叶子节点的value是数据集合的单元数据或者单元数据HASH。

C)非叶子节点的value是根据它下面所有的叶子节点值，然后按照Hash算法计算而得出的。

(5)Merkle Tree操作

a)创建

从叶子节点往前推，首先计算叶子节点的hasn值，叶子节点单个或者两两结合计算其hasn值，得到新节点，以此类推：

b)检索

略。

c)更新、插入和删除

略。

(6)Merkle Tree的应用

(A)数字签名

最初Merkle Tree目的是高效的处理Lamport one-time signatures。 每一个Lamport key只能被用来签名一个消息，但是与Merkle tree结合可以来签名多条Merkle。这种方法成为了一种高效的数字签名框架，即Merkle Signature Scheme。

(B)P2P网络

在P2P网络中，Merkle Tree用来确保从其他节点接受的数据块没有损坏且没有被替换，甚至检查其他节点不会欺骗或者发布虚假的块。大家所熟悉的BT下载就是采用了P2P技术来让客户端之间进行数据传输，一来可以加快数据下载速度，二来减轻下载服务器的负担。BT即BitTorrent，是一种中心索引式的P2P文件分分析通信协议。

要进下载必须从中心索引服务器获取一个扩展名为torrent的索引文件（即大家所说的种子），torrent文件包含了要共享文件的信息，包括文件名，大小，文件的Hash信息和一个指向Tracker的URL[8]。Torrent文件中的Hash信息是每一块要下载的文件内容的加密摘要，这些摘要也可运行在下载的时候进行验证。大的torrent文件是Web服务器的瓶颈，而且也不能直接被包含在RSS或gossiped around(用流言传播协议进行传播)。一个相关的问题是大数据块的使用，因为为了保持torrent文件的非常小，那么数据块Hash的数量也得很小，这就意味着每个数据块相对较大。大数据块影响节点之间进行交易的效率，因为只有当大数据块全部下载下来并校验通过后，才能与其他节点进行交易。

就解决上面两个问题是用一个简单的Merkle Tree代替Hash List。设计一个层数足够多的满二叉树，叶节点是数据块的Hash，不足的叶节点用0来代替。上层的节点是其对应孩子节点串联的hash。Hash算法和普通torrent一样采用SHA1。其数据传输过程和第一节中描述的类似。

参考http://www.cnblogs.com/fengzhiwu/p/5524324.html

## 人工智能

# 13 测试

## 13.1 单元测试(JUnit)

(维基百科)单元测试（英语：Unit Testing）又称为模块测试, 是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

intellij idea进行JUnit单元测试

(1)环境搭建

略

(2)编写需进行单元测试的类及方法

public class Factorial {

/\*\*

\* 阶乘

\* @param n

\* @return

\*/

public int factorial(int n) throws Exception {

if (n < 0) {

throw new Exception("不能输入负数");

} else if (n <= 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

}

(3)编写测试类

public class MathTest {

@Test

public void testFactorial() throws Exception {

assertEquals(6, new Factorial().factorial(3));

}

}

注: assertEquals为Assert类的一个方法，主要是比较两个值是否相等。

(4)使用chtrl+shift+F10执行MathTest类

若显示绿色线条说明测试通过，红色线条说明不通过。

(5)自动生成测试方法

菜单栏---Code 选择Generate --->JUnit Test --->JUnit4.然后自动生成测试类。

注:

@Test：把一个方法标记为测试方法

Test有两个属性excepted()、timeout

@Before：每一个测试方法执行前自动调用一次

@After：每一个测试方法执行完自动调用一次

@BeforeClass：所有测试方法执行前执行一次，在测试类还没有实例化就已经被加载，所以用static修饰

@AfterClass：所有测试方法执行完执行一次，在测试类还没有实例化就已经被加载，所以用static修饰

@Ignore：暂不执行该测试方法

# 14 数据库技术

## 14.1存储过程

### 14.1.1 定义

存储过程 (Stored Procedure) 是在大型数据库系统中 , 一组为了完成特定功能的 SQL 语句集 , 存储在数据库中 , 经过第一次编译后再次调用不需要再次编译 , 用户通过指定存储过程的名字并给出参数 (如果该存储过程带有参数) 来执行它 , 存储过程是数据库中的一个重要对象 ; 存储过程中可以包含 逻辑控制语句 和 数据操纵语句 , 它可以接受参数 , 输出参数 , 返回单个或多个结果集以及返回值 ;

### 14.1.2 基本语法

1.CREATE OR REPLACE PROCEDURE XX(存储过程名)

2.IS

3.BEGIN

4.NULL;

5.END;

注：行1是一个SQL语句通知数据库创建一个叫做XX的存储过程，若存在则覆盖；

行2 IS关键词表明后面将跟随一个PL/SQL体；

行3 BEGIN关键词表明PL/SQL体的开始；

行4 NULL PL/SQL语句表明什么事都不做，这句不能删去，因为PL/SQL体中至少需要有一句;

行5 END关键词表明PL/SQL体的结束；

例如创建过程：

create or replace procedure 存储过程名（param1 in type，param2 out type）   
as   
变量1 类型（值范围）; --vs\_msg VARCHAR2(4000);   
变量2 类型（值范围）;

Begin

If(判断条件) then

Select count(\*) into 变量2 from 表A where 列名= param1;

Dbms\_output.Put\_line(‘打印信息’);

else if(判断条件) then

Dbms\_output.Put\_line(‘打印信息’);

else

Raise 异常名(NO\_DATA\_FOUND);

End if;

Exception

When others then

Rollback;

End;

注意事项：   
1， 存储过程参数不带取值范围，in表示传入，out表示输出   
类型可以使用任意Oracle中的合法类型。   
2， 变量带取值范围，后面接分号   
3， 在判断语句前最好先用count（\*）函数判断是否存在该条操作记录   
4， 用select 。。。into。。。给变量赋值   
5， 在代码中抛异常用 raise+异常名

### 14.1.3 范例

/\*

需求：现有学生成绩表、实践评分表。若学生实践评分成绩为A,则增加附加分20分，并计算学生的总分及平均分

\*/

/\*==============================================================\*/

/\* DBMS name: ORACLE Version 11g \*/

/\* Created on: 2018/9/3 17:32:43 \*/

/\*==============================================================\*/

drop table stu\_practice cascade constraints;

drop table stu\_score cascade constraints;

/\*==============================================================\*/

/\* Table: stu\_practice \*/

/\*==============================================================\*/

create table stu\_practice

(

stu\_id NUMBER not null,

pra\_score VARCHAR(1),

constraint PK\_STU\_PRACTICE primary key (stu\_id)

);

/\*==============================================================\*/

/\* Table: stu\_score \*/

/\*==============================================================\*/

create table stu\_score

(

stu\_id NUMBER not null,

chinese NUMBER,

math NUMBER,

english NUMBER,

attach NUMBER,

total NUMBER,

average NUMBER,

opertype NUMBER,

constraint PK\_STU\_SCORE primary key (stu\_id)

);

--插入测试数据

insert into stu\_score(stu\_id,chinese,math,english,opertype) values(1000,50,30,60,0)

insert into stu\_score(stu\_id,chinese,math,english,opertype) values(1001,70,80,65,0)

insert into stu\_score(stu\_id,chinese,math,english,opertype) values(1002,95,90,96,0)

insert into stu\_score(stu\_id,chinese,math,english,opertype) values(1003,65,70,96,0)

insert into stu\_score(stu\_id,chinese,math,english,opertype) values(1004,85,70,80,0)

insert into stu\_practice(stu\_id, pra\_score) values(1001, 'B');

insert into stu\_practice(stu\_id, pra\_score) values(1002, 'A');

insert into stu\_practice(stu\_id, pra\_score) values(1003, 'A');

insert into stu\_practice(stu\_id, pra\_score) values(1004, 'B');

insert into stu\_practice(stu\_id, pra\_score) values(1005, 'A');

commit;

--定义包(包可以将任何出现在块声明的语句(过程，函数，游标，类型，变量)放在包中，相当于一个容器，将声明语句放入包中的好处：用户可以从其他PL/SQL块中对其进行引用，因此包为PL/SQL提供了全程变量)

--语法：

--create [or replace] package packname /\*包头名称\*/

--is[as]

--PLSQL子程序体

create or replace package myPackage

is

type stdInfo is record(stu\_id varchar(30),pra\_score varchar(1)); /\*定义组成stdInfo成员变量及类型\*/

type myArray is table of stdInfo index by binary\_integer; /\*,index by binary\_integer:表myArray自增长\*/

end myPackage;

---定义存储过程计算

---语法：

--create [or replace] procedure porname(param)/\*存储过程名称\*/

--is[as] /\*is,as在存储过程/函数中没区别,在视图中只能用as不能用is，在游标(CURSOR)中只能用is不能用as;\*/

--PLSQL子程序体；

create or replace procedure get\_comment(commentArray out myPackage.myArray)

is

rs SYS\_REFCURSOR; --声明变量，用来存放查询表返回数据

record myPackage.stdInfo; --声明变量，用来存放表一组数据

stu\_id number; --声明变量，学生编号

pra\_score varchar(1); --声明变量，课外成绩评分

i number; --声明变量，commentArray输出下标

begin

open rs for select stu\_id,pra\_score from stu\_practice; --定义游标并且返回根据查询语句返回表信息

i :=1; --赋默认值

/\*

循环体语法一：

LOOP 循环开始标识

statement... 循环执行的语句序列

END LOOP;

循环游标语法二：

FOR v in rs LOOP

statement... 循环执行的语句序列

END LOOG

\*/

LOOP --循环体

fetch rs into stu\_id,pra\_score; --游标参数rs返回的值

exit when rs%NOTFOUND; --定义跳出循环的条件

record.stu\_id := stu\_id; --赋值给数组

record.pra\_score := pra\_score;

commentArray(i) := record; --赋值给输出变量

i := i+1;

end LOOP;

end get\_comment;

create or replace procedure autocomputer(step in number)

is

rsCursor SYS\_REFCURSOR; --声明变量，用在过程中返回结果集

commentArray myPackage.myArray; --声明变量，学生课外成绩

math number; --变量声明，数学分数

chinese number; --变量声明，语文分数

english number; --变量声明，英语分数

t\_attach number; --变量声明，附加分数，默认0分，临时变量

t\_total number; --变量声明，总分，临时变量

t\_average number; --变量声明，平均分，临时变量

stu\_id number; --变量声明，学生ID

t\_stu\_id number; --变量声明，学生ID，临时变量

record myPackage.stdInfo;

i number;

begin

i:= 1;

get\_comment(commentArray); --调用get\_comment()的存储过程获取学生课外评分信息

OPEN rsCursor for select stu\_id,math,chinese,english from stu\_score t where t.opertype=opertype;

LOOP

fetch rsCursor into stu\_id,math,chinese,english; --定义游标并且返回根据查询语句返回表信息

exit when rsCursor%NOTFOUND; --定义跳出循环的条件

t\_stu\_id := stu\_id;

t\_total := math+chinese+english; --计算总分

<<SCORE\_LOOP>>

for i in 1..commentArray.count LOOP --循环游标

t\_attach := 0;

/\*

pl/sql if else语法：

语法1：

if 条件 then

语句1；

语句2；

语法2：

if 条件 then

语句序列1；

else

语句序列2；

end if;

语法3:

if 条件 then

语句；

else if 条件 then

语句；

else

语句；

end if

\*/

record := commentArray(i);

if t\_stu\_id = record.stu\_id then

begin

if record.pra\_score = 'A' then

t\_attach:= 20;

exit SCORE\_LOOP; --跳出for循环

end if;

end;

end if;

end LOOP;

t\_total:= t\_total+t\_attach;

<<continue>>

t\_average := t\_total/3;

--注:在sqlplus可以打印，但需先执行set serveroutput on;

dbms\_output.put\_line('学生编号：'||t\_stu\_id||'附加分：'||t\_attach||'总分：'||t\_total||'平均分：'||t\_average);

/\*注：attach命名值不能与表命名值一致，否则更新不成功\*/

update stu\_score t set t.attach=t\_attach,t.total=t\_total,t.average = t\_average,t.opertype=1 where t.stu\_id = t\_stu\_id;

end LOOP;

end autocomputer;

--执行

begin

autocomputer(0);

commit;

end;

执行前stu\_practice：



执行前stu\_score：



执行后stu\_score：



## 14.2 函数

### 14.2.1 基本语法

create [or replace] function function\_name(函数参数列表) --注：参数有IN,OUT,IN OUT三种类型，IN输入，OUT输出，IN OUT输入亦是输出

return datatype --返回类型

[is/AS]

[declare] ----变量声明

PL/SQL代码块；

### 14.2.2 范例

/\*\*

需求：求取平均分分数75分及以上的人数

\*/

/\*

函数语法：

create [or replace] function function\_name(函数参数列表) --注：参数有IN,OUT,IN OUT三种类型，IN输入，OUT输出，IN OUT输入亦是输出

return datatype --返回类型

[is/AS]

[declare] ----变量声明

PL/SQL代码块；

函数可以有返回值！存储过程无。

\*/

create or replace function getNumByScore(score in number)

return number

as

tmp\_sum number; --声明变量

begin

select count(\*) into tmp\_sum from stu\_score a where a.average > score;

return tmp\_sum;

end;

set serveroutput on

declare

score number;

num number;

begin

score := 10;

num := getNumByScore(score);

dbms\_output.put\_line('分数超过'||score||'分的人数有'||num);

score := 80;

num := getNumByScore(score);

dbms\_output.put\_line('分数超过'||score||'分的人数有'||num);

end;

/\*

SQL> set serveroutput on

SQL> declare

2 score number;

3 num number;

4 begin

5 score := 10;

6 num := getNumByScore(score);

7 dbms\_output.put\_line('分数超过'||score||'分的人数有'||num);

8 score := 80;

9 num := getNumByScore(score);

10 dbms\_output.put\_line('分数超过'||score||'分的人数有'||num);

11 end;

12 /

分数超过10分的人数有5

分数超过80分的人数有2

PL/SQL 过程已成功完成。

\*/

## 14.3 乐观锁及悲观锁

数据库悲观锁实现；(锁机制) select for update

(1)session 1执行

select \* from tableA where id='1' for update;

(2)session 1未提交或关闭，session 2执行

select \* from tableA where id='1' for update;

此时session 2的窗口会处于等待状态(注：此时不影响select \* from tableA 执行)；

(3)session 1未提交或关闭, session 3执行

select \* from tableA where id='1' for update nowait;

此时弹出提示框报"资源忙提示"；

注，for upate 为行级锁，若where 后面条件有冲突时，其他session无法获取锁。Insert、Update、Delete与 select for update 效果一致

数据库乐观锁实现；

基于数据版本记录机制实现。

表自定义一个字段作为版本号，每当更新一次记录都会进行自增或者更新，更新时版本号发生变化时，无法更新数据行。只能重新查询并更新

select \* from tableA where id="1";

update tableA set version=version+1, ... where id="1" and version="0";

## 14.4 索引

### 14.4.1 使用索引的目的

提高数据检索的速度。

### 14.4.2 索引的分类

五种:唯一索引，组合索引，反向键索引，位图索引，基于函数的索引

(A)唯一索引

CREATE unique INDEX 索引名 ON 表名 (列名)

TABLESPACE 表空间名;

(B)组合索引

CREATE INDEX 索引名 ON 表名(列名1,列名2)

TABLESPACE 表空间名;

(C)反向键索引

CREATE INDEX 索引名 ON 表名 (列名) reverse

TABLESPACE 表空间名;

注：

表中主键字段自动建立唯一索引。

order by 字段A 字段A可建立索引

where 字段A=? order by 字段B 字段A+字段B可建立组合索引

### 14.4.3 什么时候不用建立索引

(A)表中记录太少

建立索引，会首先通过访问索引表，再访问数据表，数据量比较少的情况下，处理速度相对比较慢。

(B)经常插入、删除、修改的表

(C)对于数据重复且分布平均的表字段

(D)经常和主字段一块查询但主字段索引值比较多的表字段

(E)复合索引中只要一列含有NULL值，那么这一列对于此复合索引就是无效的。

## 14.5 数据库优化

(1)应用层:减少应用请求；

做法:将一些读取频繁的，并且不经常改动的数据放置到缓存中。

(2)数据库层：降低内部资源竞争；

(A)合理利用索引，多建索引导致插入浪费过多时间，少见索引导致查询浪费时间;

(B)对于数据量特别大的表，避免全表扫描；

(C)SQL语句优化；

(D)读写分离到CQRS(可扩展)；

(3)内存层:降低IO读写数量和规模；

(4)存储层:优化IO处理能力；

# 15 设计模式

### 15.1 创建模式-单例模式

(1)普通单例类创建：

public class Singleton

{

private static Singleton instance = new Singleton();

private Singleton(){

…

}

public static Singleton getInstance(){

return instance;

}

}

缺点：实例已经创建，但是消耗资源。

(2)惰性加载创建：

public class Singleton{

private static Singleton instance = null;

private Singleton(){

…

}

public static Singleton getInstance(){

if (instance == null)

instance = new Singleton();

return instance;

}

}

缺点：多线程避免不了同时new两个实例；

(3)Class锁机制

public class Singleton{

private static Singleton instance = null;

private Singleton(){

…

}

public static synchronized Singleton getInstance(){

if (instance == null)

instance = new Singleton();

return instance;

}

}

缺点:影响性能每次调用getInstance都必须获得Singleton的锁，但

(4)double-checked locking

public class Singleton{

private static Singleton instance = null;

private Singleton(){

…

}

public static Singleton getInstance(){

if (instance == null)

synchronized(instance){

if(instance == null)

instance = new Singleton();

}

return instance;

}

}

工作原理：

首先当一个线程发出请求后，会先检查instance是否为null，如果不是则直接返回其内容。

当并发访问时，必会有一线程先执行创建，后续线程非空不会创建。

缺点：

1. B线程先后执行，JVM给A分配空间但未new，直接返回。B线程进入会出错；

注，原因：JVM指令重排特性导致，instance = new Singleton();主要分为三个步骤执行

步骤1：为 instance分配内存空间；

步骤2：初始化 instance；

步骤3：将instance指向分配的内存地址；

执行顺序可能为步骤1🡪步骤3🡪步骤2。若instance变量由volatile修饰，可以保证在多线程环境下也正常进行。

(5)完美解决：

public class Singleton{

private Singleton(){

…

}

private static class SingletonContainer{

private static Singleton instance = new Singleton();

}

public static Singleton getInstance(){

return SingletonContainer.instance;

}

}

原理：(内部类只在需要的时候才会被类加载器加载，实现了懒加载)JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕，这样我们就不用担心3.2中的问题。此外该方法也只会在第一次调用的时候使用互斥机制。最后instance是在第一次加载SingletonContainer类时被创建的，而SingletonContainer类则在调用getInstance方法的时候才会被加载，因此也实现了惰性加载。

### 15.2 代理模式

远程代理实现原理(RMI，具体见RMI)。





静态代理和动态代理的区别

主要区别是静态代理是手写代理类，而动态代理是编译时候动态创建代理类，后者避免了代理类创建过多；

动态代理的使用:spring AOP,详见spring AOP

### 15.3 装饰者模式



# 16 网络安全

## 16.1 公钥密码体制

属于非对称加密的一种。

主要由三部分组成，公钥、私钥、加密解密算法；

加密过程：通过加密算法和公钥对明文进行加密，得到密文；

解密过程：通过解密算法和私钥对密文进行解密，得到明文；

公钥密码体制中的公钥和加密解密算法都是公开的，私钥是保密的。

常见的公钥加密算法有：RSA、ElGamal等。

注：RSA算法中，公钥、私钥都可以用来加密和解密，并且一方加密的内容可以由并且只能由对方进行解密。

## 16.2 对称加密算法

加密及解密都是使用同一个密钥。

在公钥密码算法和对称加密算法中，公钥、私钥已经密钥一般都是字符串或数字。

## 16.3数字证书

为了解决公钥对是否是合法的，故引进数字证书。