# Java基础

## 面向对象和面向过程的区别

**面向过程**

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护、易复用、易扩展

**面向对象**

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点：性能比面向过程低

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。 面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

面向过程的封装是函数，面向对象的对类做封装

面向对象利于软件的维护，主要体现在面向对象是具有继承和多态等优点，使在软件维护上方便

面向对象与面向过程具体而言，有如下几个方面的不同：

1、出发点不同

面向对象方法是用符合常规思维的方式来处理客观世界的问题，强调把问题域的要领直接映射到对象之间的接口上。而面向过程的方法则强调的则是过程的抽象化与模块化，它是以过程为中心构造或处理客观世界问题的。

2、层次逻辑关系不同

面向对象的方法则是用计算机逻辑来模拟客观世界中的物理存在的，以对象的集合类作为处理问题的单位，尽可能地使计算机世界向客观世界靠拢，以使问题的处理更清晰直接，面向对象方法是用类的层次结构来体现类之间的继承与发展。面向过程方法处理问题的基本单位是能清晰准确地表达过程的模块，用模块的层次结构概括模块或模块间的关系与功能，把客观世界的问题抽象成计算机可以处理的过程。

3、数据处理方式与控制程序方式不同

面向对象方法将数据与对应的代码封装成一个整体，原则上其他对象不能直接修改其数据，即对象的修改只能由自身的成员函数完成，控制程序方式上是通过“事件驱动”来激活和运行程序。而面向过程方法是直接通过程序来处理数据，处理完毕后即可显示处理的结果，在控制方式上是按照设计调用或返回程序，不能自由导航，各模块之间存在着控制与被控制，调动与被调用的关系。

4、分析设计与编码转换方式不同

面向对象方法贯穿于软件生命周期的分析，设计及编码中，是一种平滑过程，从分析到设计再到编码是采用一致性的模型表示，即实现的是一种无缝连接。而面向过程方法强调分析、设计及编码之间按规则进行转换贯穿于软件生命周期的分析、设计及编码中，实现的是一种有缝的连接。

## 面线对象的特性

面向对象的三大特性，封装、继承、多态

封装:

对象数据和操作该对象的指令都是对象自身的一部分，能够实现尽可能对外部隐藏数据。属性私有化（private）、提供相对应的get/set 的方法进行访问(public)、在set/get的方法中对属性的数据 做相对应的业务逻辑的判断。封装外面的程序不能直接访问本类的属性，需要通过本类提供的方法间接访问本类的属性，从而提高代码的安全性。

继承：

子类 extends 父类。多个类所描述的有共同的特征或行为，就可以将这些类中共同的部分抽取出来建立他们的父类。子类共用父类的代码，提高了代码的重用性;代码后期扩展和维护，只需添加对应的子类即可。

多态：

父类 对象名=new 子类()。父类型的引用可以指向内存中的子类对象的内存空间，

就可以方便在设计方法的参数和返回值类型的时候，可以设计成父类对象，这样的方法在实现时比较简洁，在后期进行扩展时，方法比较容易。在代码中，多态的使用更多的是父类引用指向子类对象,会自动类型转换（向上转型）,但是有时候也需要调用子类中自己的东西，这个时候就需要还原子类，叫强制类型转换（向下转型），转的时候要先使用instanceof运算符做类型的判断。

多态的理解：

多态就是对象拥有多种形态：引用多态和方法多态。

②引用多态：父类的引用可以指向本类对象、父类的引用可以指向子类的对象。

③方法多态：创建本类对象时，调用的方法为本类的方法；创建子类对象时，调用的方法为子类重写的方法或者继承的方法。

④存在多态的必要条件：继承、重写。

⑤多态的作用是消除类型之间的耦合关系。

## Java的4个基本特性（抽闲、封装、继承、多态）

抽象：就是把现实生活中的某一类东西提取出来，用程序代码表示，我们通常叫做类或者接口。抽象包括两个方面：一个是数据抽象，一个是过程抽象。数据抽象也就是对象的属性。过程抽象是对象的行为特征。

封装：把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行封装隐藏。封装分为属性的封装和方法的封装。

继承：是对有着共同特性的多类事物，进行再抽象成一个类。这个类就是多类事物的父类。父类的意义在于抽取多类事物的共性。

extends继承父类，implemens实现接口

多态：允许不同类的对象对同一消息做出响应。方法的重载、类的覆盖正体现了多态。

多态存在的三个必要条件：要有继承、要有重写、父类引用指向子类对象

## 重载和重写的区别

重载（Overload）：

发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。无法以返回值类型作为重载的标准。最常用的就是构造器的重载。

被重载的方法必须改变参数列表(参数个数或类型或顺序不一样)；

被重载的方法可以改变返回类型；

被重载的方法可以改变访问修饰符；

被重载的方法可以声明新的或更广的检查异常；

方法能够在同一个类中或者在一个子类中被重载。

无法以返回值类型作为重载函数的区分标准

重写（Override）：

发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛出的异常小于等于父类，访问修饰符大于等于父类；如果父类方法访问修饰符为private则子类中就不是重写。

参数列表必须完全与被重写方法的相同；

返回类型必须完全与被重写方法的返回类型相同；

访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。

父类的成员方法只能被它的子类重写。

声明为final的方法不能被重写。

声明为static的方法不能被重写，但是能够被再次声明。

子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。

子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。

重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。

**构造方法不能被重写，不能用static修饰构造器，只能用public、private、protected这三个权限修饰，且不能有返回语句。**

如果不能继承一个方法，则不能重写这个方法。

## 访问控制符public、protected、private以及默认的区别

private只有在本类中才能访问；

public在任何地方都能访问；

protected在同包内的类及包外的子类能访问；

默认不写在同包内能访问。

## static和final的区别和用途

* static
  + 修饰变量：静态变量随着类加载时被完成初始化，内存中只有一个，且JVM也只会为它分配一次内存，所有类共享静态变量。
  + 修饰方法：在类加载的时候就存在，不依赖任何实例，static 方法必须实现，不能用abstract修饰。
  + 修饰代码块：在类加载完之后就会执行代码块中的内容。
  + 父类静态代码块：->子类静态代码块->父类非静态代码块->父类构造方法->子类非静态代码块->子类构造方法
* final
  + 修饰变量：
    - 编译期常量：类加载的过程完成初始化，编译后带入到任何计算式中，只能是基本类型
    - 运行时常量：基本数据类型或引用数据类型。引用不可变，但引用的对象内容可变。对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
  + 修饰方法：不能被子类修改
  + 修饰类：不能被继承（例如 String 类）
  + 修饰形参：final形参不可变

## String、StringBuilder、StringBuffer区别

**可变性**

String类中使用字符数组保存字符串，private final char value[]，所以string对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[] value，这两种对象都是可变的。

**线程安全性**

String中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如expandCapacity、append、insert、indexOf等公共方法。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。

**性能**

每次对String 类型进行改变的时候，都会生成一个新的String 对象，然后将指针指向新的String 对象。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。相同情况下使用StirngBuilder 相比使用StringBuffer 仅能获得10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

## Java的8种基本数据类型

byte：8位，最大存储数据量是255，存放的数据范围是-128~127之间。

short：16位，最[**大数据**](http://lib.csdn.net/base/hadoop)存储量是65536，数据范围是-32768~32767之间。

int：32位，最大数据存储容量是2的32次方减1，数据范围是负的2的31次方到正的2的31次方减1。

long：64位，最大数据存储容量是2的64次方减1，数据范围为负的2的63次方到正的2的63次方减1。

float：32位，数据范围在3.4e-45~1.4e38，直接赋值时必须在数字后加上f或F。

double：64位，数据范围在4.9e-324~1.8e308，赋值时可以加d或D也可以不加。

boolean：只有true和false两个取值。

char：16位，存储Unicode码，用单引号赋值。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 基本类型 | boolean | byte | char | short | int | long | float | double |
| 二进制位数 |  | 8 | 16 | 16 | 32 | 64 | 32 | 64 |
| 封装器类 | Boolean | Byte | Character | Short | Integer | Long | Float | Double |

## Object类的几个方法 equals、tostring、hashcode重写之前是什么样

1.在重写之前：

（1）toString()方法返回的是对象的：Object 类的 toString 方法返回一个字符串，该字符串由类名（对象是该类的一个实例）、at 标记符“@”和此对象哈希码的无符号十六进制表示组成。换句话说，该方法返回一个字符串，它的值等于：

getClass().getName() + ‘@’ + Integer.toHexString(hashCode())

比如：com.itheima.hah.Dog@2260e277

（2）equals()：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）克隆的对象和原来的对象不是同一个对象。对象地址不一样

（3）hashCode()：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。（这一般是通过将该对象的内部地址转换成一个整数来实现的）。

重写之后

（1）toString()方法一般返回的是：属性的拼接

@Override

public String toString() {

// TODO Auto-generated method stub

return “Person2 [name=” + name + “, age=” + age + “]”;

}

（2）equals()方法：重写之后一般只要自定义为两个对象的所有属性值相等就返回为true；

（3）hasdCode():重写hashCode，

@Override

public int hashCode()

{

final int PRIME = 31;

int result = 1;

result = PRIME \* result + getId();

return result;

}

1) 对象相等则hashCode一定相等；

2) hashCode相等对象未必相等。也就是说两个不同的对象也有可能hashCode（散列码（取余得出的））相同

判断两个对象是否相等的条件：

没有重写的话，只要hashCode不相等那么两个对象一定不相同，

重写的话，首先判断hashCode如果不一样，那么两个对象不相等，如果hashCode一样的话，再比较equals，如果true就相同，否则不相等。

finalize()：当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。子类重写 finalize 方法，以配置系统资源或执行其他清除。也就是手动垃圾回收。

## 反射的使用方法 三种常用 getclass forinstance forname，使用时机，jvm怎么执行，类是怎么加载的，对象是怎么生成的，注意classloader



public class Fanshe {

public static void main(String[] args) {

// 第1种获取Class对象

Student stu1 = new Student(); // 这一new 产生一个Student对象，一个Class对象

Class stuClass = stu1.getClass(); // 获取Class对象

System.out.println(stuClass.getName());

// 第2种获取Class对象

Class stuClass2 = Student.class;

System.out.println(stuClass == stuClass2); //判断和第1中获取的Class对象是否是同一个

// 第3种获取Class对象

try {

Class stuClass3 = Class.forName("com.ning.fanshe.Student"); // 真实路径，包名.类名

System.out.println(stuClass3 == stuClass2); // 判断和第2个获取的是否是同一个Class对象

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

注意： 在运行期间一个类只能有一个Class对象

通过反射获取构造方法

获取成员变量

获取成员方法

反射main方法

反射方法的其它使用之---通过反射越过泛型检查

反射方法的其它使用之---通过反射运行配置文件内容

java中创建对象的5中方式

|  |  |
| --- | --- |
| 使用new关键字 | 调用了构造函数 |
| 使用Class类的newInstance方法 | 调用了构造函数 |
| 使用Constructor类的newInstance方法 | 调用了构造函数 |
| 使用clone方法 | 没有调用构造函数 |
| 使用反序列化 | 没有调用构造函数 |

## 反射的作用，可以通过反射获得什么，final关键字是否一定不能修改(string对象是不是一定不可变、怎么变)

<https://blog.csdn.net/sinat_38259539/article/details/71799078>



大家都知道，要让Java程序能够运行，那么就得让Java类要被Java虚拟机加载。Java类如果不被Java虚拟机加载，是不能正常运行的。现在我们运行的所有的程序都是在编译期的时候就已经知道了你所需要的那个类的已经被加载了。

Java的反射机制是在编译并不确定是哪个类被加载了，而是在程序运行的时候才加载、探知、自审。使用在编译期并不知道的类。这样的特点就是反射

**获取一个对象的类信息.**

**—获取一个类的访问修饰符、成员、方法、构造方法以及超类的信息.**

**—检获属于一个接口的常量和方法声明.**

**—创建一个直到程序运行期间才知道名字的类的实例.**

**—获取并设置一个对象的成员，甚至这个成员的名字是在程序运行期间才知道.**

**—检测一个在运行期间才知道名字的对象的方法**

虽然String对象将value设置为final,并且还通过各种机制保证其成员变量不可改变。但是还是可以通过反射机制的手段改变其值。

// 创建字符串"Hello World",并赋给应用s

String s = "Hello World";

System.out.println("s = " + s);

// 获取String 类中的value字段

Field valueOfString = String.class.getDeclaredField("value");

// 改变value属性的访问权限

valueOfString.setAccessible(true);

// 获取s对象上的value属性的值

char[] value = (char[]) valueOfString.get(s);

// 改变value所引用的数组中的第5个字符

value[5] = '\_';

System.out.println("s = " + s);

打印结果为：

s = Hello World

s = Hello\_World

## 抽象类和接口的区别

**语法层次**

抽象类和接口分别给出了不同的语法定义。

**设计层次**

抽象层次不同，抽象类是对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。

**跨域不同**

抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a"关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的，仅仅是实现了接口定义的契约而已，"like-a"的关系。

**抽象类接口对比**

|  |  |  |
| --- | --- | --- |
| 参数 | 抽象类 | 接口 |
| 默认的实现方法 | 它可以有默认的方法实现 | 接口是完全抽象的，根本不存在方法的实现 |
| 实现 | 子类使用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方法的实现 |
| 构造器 | 抽象类可以有构造器 | 接口不能有构造器 |
| 与正常Java类的区别 | 除了不能实例化抽象类之外，它和普通的Java类无任何区别 | 接口是完全不同的类型 |
| 访问修饰符 | 抽象类可以有public、protected和defaut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 |
| main方法 | 抽象类可以有main方法并且可以运行 | 接口没有main方法，因此我们不能运行它 |
| 多继承 | 抽象类可以继承一个类和实现多个接口 | 接口只可以继承一个或多个其它接口 |
| 速度 | 比接口速度要快 | 接口是稍微有点慢的，因为他需要去寻找在类中实现的方法 |
| 添加新方法 | 如果往抽象类中添加新的方法，可以给它提供默认的实现，因此不需要改变现在的代码 | 如果你往接口中添加方法，那么你必须改变实现该接口的类 |

**什么时候用抽象类和接口**

如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。

如果你想实现多重继承，那么你必须使用接口。由于**Java不支持多继承**，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。

如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

## 对象的clone方法，深拷贝、浅拷贝

将一个对象的引用复制给另外一个对象，一共有三种方式。第一种方式是直接赋值，第二种方式是浅拷贝，第三种是深拷贝。

浅拷贝：Object的Clone()函数了。在这里，我们需要了解clone()主要做了些什么，创建一个新对象，然后将当前对象的非静态字段复制到该新对象，如果字段是值类型的，那么对该字段执行复制；如果该字段是引用类型的话，则复制引用但不复制引用的对象。因此，原始对象及其副本引用同一个对象。

浅拷贝是指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝对象包含的引用指向的对象。深拷贝不仅拷贝对象本身，而且拷贝对象包含的引用指向的所有对象

## 内部类的特点、静态内部类和普通内部类的区别

<https://www.cnblogs.com/chenssy/p/3388487.html>

Java内部类分四种：成员内部类、局部内部类、静态内部类和匿名内部类。

* 成员内部类：作为外部类的一个成员存在，与外部类的属性、方法并列。注意：成员内部类中不能定义静态变量,但可以访问外部类的所有成员。成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类

成员内部类的优点：

⑴ 内部类作为外部类的成员，可以访问外部类的私有成员或属性。（即使将外部类声明为PRIVATE，但是对于处于其内部的内部

类还是可见的。）

⑵ 用内部类定义在外部类中不可访问的属性。这样就在外部类中实现了比外部类的private还要小的[访问权限](https://www.baidu.com/s?wd=%E8%AE%BF%E9%97%AE%E6%9D%83%E9%99%90&tn=67012150_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1Y4myf4uWnzmWbdrjFbrjmL0ZwV5fKWUMw85HmLnjDznHRsgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHT4nWnLPHDv)。 注意：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类。对于一个名为outer的外部类和其内部定义的名为inner的内部类。编译

完成后出现outer.class和outer$inner.class两类。

* 局部内部类： 即在方法中定义的内部类，与局部变量类似，在局部内部类前不加修饰符public或private，其范围为定义它的代码块。 注意：局部内部类中不可定义静态变量，可以访问外部类的局部变量(即方法内的变量)，但是变量必须是final的。在类外不可直接生成局部内部类（保证局部内部类对外是不可见的）。要想使用局部内部类时需要生成对象，对象调用方法，在方法中才能调用其局部内部类。通过内部类和接口达到一个强制的弱耦合，用局部内部类来实现接口，并在方法中返回接口类型，使局部内部类不可见，屏蔽实现类的可见性。
* 静态内部类： 静态内部类定义在类中，任何方法外，用static定义。注意：静态内部类中可以定义静态或者非静态的成员。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
* 匿名内部类 匿名内部类是一种特殊的局部内部类，它是通过匿名类实现接口。

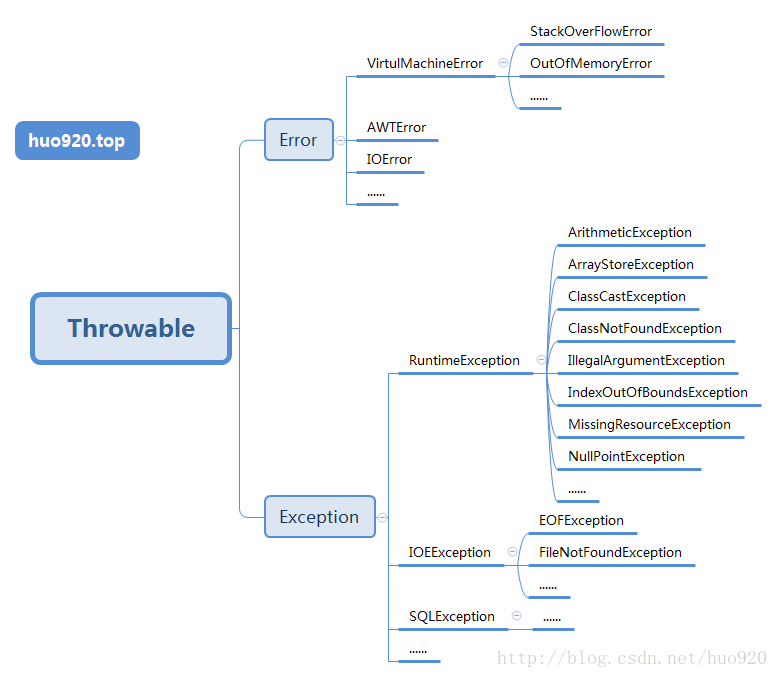
匿名内部类的特点：

一个类用于继承其他类或是实现接口，并不需要增加额外的方法，只是对继承方法的事先或是覆盖。

只是为了获得一个对象实例，不需要知道其实际类型。

类名没有意义，也就是不需要使用到。

## 异常分类图 异常分类图 从throwable开始



Error类和Exception类的父类都是throwable类，他们的区别是：

Error类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。

Exception类表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

**Unchecked Exception:**

a. 指的是程序的瑕疵或逻辑错误，并且在运行时无法恢复。

b. 包括Error与RuntimeException及其子类，如：OutOfMemoryError,

UndeclaredThrowableException, IllegalArgumentException,

IllegalMonitorStateException, NullPointerException, IllegalStateException,

IndexOutOfBoundsException等。

c. 语法上不需要声明抛出异常。

**Checked Exception:**

a. 代表程序不能直接控制的无效外界情况（如用户输入，数据库问题，网络异常，文件丢失等）

b. 除了Error和RuntimeException及其子类之外，如：ClassNotFoundException,

NamingException, ServletException, SQLException, IOException等。

c. 需要try catch处理或throws声明抛出异常

## throws和throw关键字区别 自定义异常

1、throws出现在方法函数头；而throw出现在函数体。

2、throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某种异常对象。程序在执行到throw语句时立即停止，它后面的语句都不执行

3、两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

**编程习惯：**

1.在写程序时，对可能会出现异常的部分通常要用try{...}catch{...}去捕捉它并对它进行处理；

2.用try{...}catch{...}捕捉了异常之后一定要对在catch{...}中对其进行处理，那怕是最简单的一句输出语句，或栈输入e.printStackTrace();

3.如果是捕捉IO输入输出流中的异常，一定要在try{...}catch{...}后加finally{...}把输入输出流关闭；

4.如果在函数体内用throw抛出了某种异常，最好要在函数名中加throws抛异常声明，然后交给调用它的上层函数进行处理。

自定义异常 ：自定义异常通常是定义一个继承自Exception类的子类。一般情况下我们都会直接继承自Exception类，而不会继承某个运行时的异常类。

5.1、创建自定义异常：

public class MyException extends Exception{ public MyException(){ super(); } public MyException(String msg){ super(msg); } }

5.2、在类中使用异常：

public class ExceptionTest { public static void execute(String a) throws MyException { System.out.println("execute..."); if("true".equals(a)){ throw new MyException("参数不能为 true"); } } }

5.3、捕获自定义异常：

public static void main(String[] args) throws MyException { execute("true"); }

## try catch finally执行顺序，注意在try里面有return、finally里面没有return的话，返回值是否会因为finally里面对对象进行改变而改变

try、catch、finally 这个过程也就是这样，如果try catch finally 都有return：

1、在没有异常的情况下，try 中的返回值先保存到临时区域里在去执行finally ，这个finally 有返回值，这个返回值将之前try中的保存到临时区域的值用返回的这个值替换，再将这个临时区域中的值返回给上一级方法。

2、如果有异常，则执行catch中的代码，这里的return 将返回一个返回值放到临时区域，再去执行finally ，这个finally有返回值，这样就将catch中存在临时区域中的值用这个finally 返回的值替换掉，在将这个临时区域的值返回给上一级方法。

finally其实是仅在return 语句执行前执行，如果return一个函数，那么会先执行函数，但如果函数内有（return）语句，那么finally就会在这个return 语句前执行。finally在catch中的return之前执行但是如果catch中有返回值而finally中也有返回值的话finally中的返回值会替换catch中的返回值，因为catch中的返回值是存放在一个临时区中，try 中的过程和catch 是一样的。

如果catch块有异常向外抛出，执行顺序呢：我执行我，你抛你得异常，我finally我的语句，我俩互不干涉，你别管我啥时执行，但我一定会执行。

关于finally，此时，应该很明朗了只需记着一点：除非调用system.exit()让程序退出也就是将调用这个程序的进程断开了退出了这个程序就不会执行或断电等因素致使程序停止进程终止，否则无论任何因素finally块都一定会执行。

**使用注意事项**

Java中try里面不应放太多的代码，只放必须放的就可以，放的太多的话如果检测到了异常，try里面发生异常那句代码之后的代码将不会被执行。所以当我们抛不抛异常都需要执行的代码不能放在try里

尽量不要讲try catch写在循环中，如果能捕获多个异常，那么先捕获小异常，在捕获大异常

尽量避免是用万能代码捕获所有异常

适当配合finally块

## catch多个异常jdk新特性、catch异常的顺序

**在写异常处理的时候，一定要把异常范围小的放在前面，范围大的放在后面**，Exception这个异常的根类一定要放在最后一个catch里面，如果放在前面或者中间，任何异常都会和Exception匹配的，就会报已捕获到...异常的错误

在Java 7中，catch代码块得到了升级，用以在单个catch块中处理多个异常。如果你要捕获多个异常并且它们包含相似的代码，使用这一特性将会减少代码重复度

Java7之前版本 ：

|  |
| --- |
| catch (IOException ex) {  logger.error(ex);  throw new MyException(ex.getMessage());  catch (SQLException ex) {  logger.error(ex);  throw new MyException(ex.getMessage());  }catch (Exception ex) {  logger.error(ex);  throw new MyException(ex.getMessage());  } |

Java7中：

**catch**(IOException | SQLException | Exception ex){

logger.error(ex);

**throw** **new** MyException(ex.getMessage());

}

## 泛型机制(问的不多，了解一下泛型的实现原理、和反射的关系)、泛型的代码怎么写的、几个局限性

<https://blog.csdn.net/s10461/article/details/53941091/>

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

泛型只在编译时有效

泛型类型在逻辑上可以可以看成是 多个不同的类型，实际上都是相同的基本类型

**局限性**

* 不能用基本类型实例化类型参数
* 不能实例化类型参数
* 不能声明参数化的数组
* 类型参数不能进行类型查询
* 不能抛出、不能捕获泛型类实例
* 类型擦除后引起的冲突

## String Arrays工具类

* String

[**https://www.cnblogs.com/ysocean/p/8571426.html**](https://www.cnblogs.com/ysocean/p/8571426.html)

1、String 类的定义

public final class String

implements java.io.Serializable, Comparable<String>, CharSequence {}

是一个用 final 声明的常量类，不能被任何类所继承,而且一旦一个String对象被创建, 包含在这个对象中的字符序列是不可改变的, 包括该类后续的所有方法都是不能修改该对象的，直至该对象被销毁，这是我们需要特别注意的（该类的一些方法看似改变了字符串，其实内部都是创建一个新的字符串，下面讲解方法时会介绍）。接着实现了 Serializable接口，这是一个序列化标志接口，还实现了 Comparable 接口，用于比较两个字符串的大小（按顺序比较单个字符的ASCII码），后面会有具体方法实现；最后实现了 CharSequence 接口，表示是一个有序字符的集合

2、字段属性

/\*\*用来存储字符串 \*/

private final char value[];

/\*\* 缓存字符串的哈希码 \*/

private int hash; // Default to 0

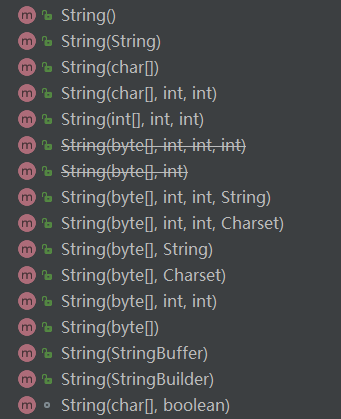
/\*\* 实现序列化的标识 \*/

private static final long serialVersionUID = -6849794470754667710L;

一个String字符串实际上是一个char数组

3、构造方法

String 类的构造方法很多。可以通过初始化一个字符串，或者字符数组，或者字节数组等等来创建一个 String 对象。



4、equals(Object anObject) 方法

String 类重写了 equals 方法，比较的是组成字符串的每一个字符是否相同，如果都相同则返回true，否则返回false。

5、hashCode() 方法

String 类的 hashCode 算法很简单，主要就是中间的 for 循环，计算公式如下：

s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

s 数组即源码中的 val 数组，也就是构成字符串的字符数组。这里有个数字 31 ，为什么选择31作为乘积因子，而且没有用一个常量来声明？主要原因有两个：

①、31是一个不大不小的质数，是作为 hashCode 乘子的优选质数之一。

②、31可以被 JVM 优化，31 \* i = (i << 5) - i。因为移位运算比乘法运行更快更省性能。

6、charAt(int index) 方法

我们知道一个字符串是由一个字符数组组成，这个方法是通过传入的索引（数组下标），返回指定索引的单个字符。

7、compareTo(String anotherString) 和 compareToIgnoreCase(String str) 方法

该方法是按字母顺序比较两个字符串，是基于字符串中每个字符的 Unicode 值。当两个字符串某个位置的字符不同时，返回的是这一位置的字符 Unicode 值之差，当两个字符串都相同时，返回两个字符串长度之差。

compareToIgnoreCase() 方法在 compareTo 方法的基础上忽略大小写，我们知道大写字母是比小写字母的Unicode值小32的，底层实现是先都转换成大写比较，然后都转换成小写进行比较。

8、concat(String str) 方法

该方法是将指定的字符串连接到此字符串的末尾

首先判断要拼接的字符串长度是否为0，如果为0，则直接返回原字符串。如果不为0，则通过 Arrays 工具类（后面会详细介绍这个工具类）的copyOf方法创建一个新的字符数组，长度为原字符串和要拼接的字符串之和，前面填充原字符串，后面为空。接着在通过 getChars 方法将要拼接的字符串放入新字符串后面为空的位置。

注意：返回值是 new String(buf, true)，也就是重新通过 new 关键字创建了一个新的字符串，原字符串是不变的。这也是前面我们说的一旦一个String对象被创建, 包含在这个对象中的字符序列是不可改变的。

9、indexOf(int ch) 和 indexOf(int ch, int fromIndex) 方法

indexOf(int ch)，参数 ch 其实是字符的 Unicode 值，这里也可以放单个字符（默认转成int），作用是返回指定字符第一次出现的此字符串中的索引。其内部是调用 indexOf(int ch, int fromIndex)，只不过这里的 fromIndex =0 ，因为是从 0 开始搜索；而 indexOf(int ch, int fromIndex) 作用也是返回首次出现的此字符串内的索引，但是从指定索引处开始搜索。

10、split(String regex) 和 split(String regex, int limit) 方法

split(String regex) 将该字符串拆分为给定正则表达式的匹配。split(String regex , int limit) 也是一样，不过对于 limit 的取值有三种情况：

①、limit > 0 ，则pattern（模式）应用n - 1 次

1 String str = "a,b,c";

2 String[] c1 = str.split(",", 2);

3 System.out.println(c1.length);//2

4 System.out.println(Arrays.toString(c1));//{"a","b,c"}

②、limit = 0 ，则pattern（模式）应用无限次并且省略末尾的空字串

1 String str2 = "a,b,c,,";

2 String[] c2 = str2.split(",", 0);

3 System.out.println(c2.length);//3

4 System.out.println(Arrays.toString(c2));//{"a","b","c"}

③、limit < 0 ，则pattern（模式）应用无限次

1 String str2 = "a,b,c,,";

2 String[] c2 = str2.split(",", -1);

3 System.out.println(c2.length);//5

4 System.out.println(Arrays.toString(c2));//{"a","b","c","",""}

11、replace(char oldChar, char newChar) 和 String replaceAll(String regex, String replacement) 方法

①、replace(char oldChar, char newChar) ：将原字符串中所有的oldChar字符都替换成newChar字符，返回一个新的字符串。

②、String replaceAll(String regex, String replacement)：将匹配正则表达式regex的匹配项都替换成replacement字符串，返回一个新的字符串。

12、substring(int beginIndex) 和 substring(int beginIndex, int endIndex) 方法

①、substring(int beginIndex)：返回一个从索引 beginIndex 开始一直到结尾的子字符串。

②、 substring(int beginIndex, int endIndex)：返回一个从索引 beginIndex 开始，到 endIndex 结尾的子字符串。

13、常量池

①、通过“字面量”的形式直接赋值

String str = "hello";

②、通过 new 关键字调用构造函数创建对象

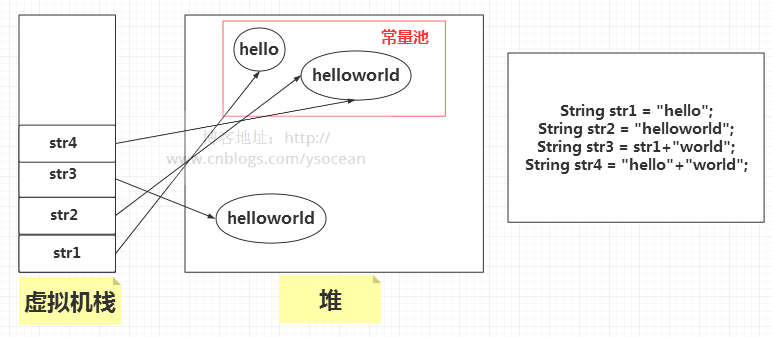
String str = new String("hello");

常量池：Java运行时会维护一个String Pool（String池）， 也叫“字符串缓冲区”。String池用来存放运行时中产生的各种字符串，并且池中的字符串的内容不重复。

①、字面量创建字符串或者纯字符串（常量）拼接字符串会先在字符串池中找，看是否有相等的对象，没有的话就在字符串池创建该对象；有的话则直接用池中的引用，避免重复创建对象。

②、new关键字创建时，直接在堆中创建一个新对象，变量所引用的都是这个新对象的地址。

③、使用包含变量表达式来创建String对象，则不仅会检查维护字符串池，还会在堆区创建这个对象，最后是指向堆内存的对象。



14、intern() 方法

这是一个本地方法： public native String intern();

当调用intern方法时，如果池中已经包含一个与该String确定的字符串相同equals(Object)的字符串，则返回该字符串。否则，将此String对象添加到池中，并返回此对象的引用。

15、String 真的不可变吗?

value 被 final 修饰，只能保证引用不被改变，但是 value 所指向的堆中的数组，才是真实的数据，只要能够操作堆中的数组，依旧能改变数据。而且 value 是基本类型构成，那么一定是可变的，即使被声明为 private，我们也可以通过反射来改变。

String str = "vae";

//打印原字符串

System.out.println(str);//vae

//获取String类中的value字段

Field fieldStr = String.class.getDeclaredField("value");

//因为value是private声明的，这里修改其访问权限

fieldStr.setAccessible(true);

//获取str对象上的value属性的值

char[] value = (char[]) fieldStr.get(str);

//将第一个字符修改为 V(小写改大写)

value[0] = 'V';

//打印修改之后的字符串

System.out.println(str);//Vae

* **Arrays：**

[**https://www.cnblogs.com/ysocean/p/8616122.html**](https://www.cnblogs.com/ysocean/p/8616122.html)

1、asList

public static <T> List<T> asList(T... a) {

return new ArrayList<>(a);

}

作用是返回由指定数组支持的固定大小列表。

①、返回的 ArrayList 数组是一个定长列表，我们只能对其进行查看或者修改，但是不能进行添加或者删除操作

②、返回的列表ArrayList里面的元素都是引用，不是独立出来的对象

③、Arrays.asList 比较适合那些已经有数组数据或者一些元素，而需要快速构建一个List，只用于读取操作，而不进行添加或删除操作的场景。

2、sort

该方法是用于数组排序，在 Arrays 类中有该方法的一系列重载方法，能对7种基本数据类型，包括 byte,char,double,float,int,long,short 等都能进行排序，还有 Object 类型（实现了Comparable接口），以及比较器 Comparator 。

①、基本类型的数组

在 Arrays.sort 方法内部调用 DualPivotQuicksort.sort 方法，这个方法的源码很长，分别对于数组的长度进行了各种算法的划分，包括快速排序，插入排序，冒泡排序都有使用

②、对象类型数组

该类型的数组进行排序可以实现 Comparable 接口，重写 compareTo 方法进行排序。

③、没有实现Comparable接口的，可以通过Comparator实现排序

3、binarySearch

用二分法查找数组中的某个元素。该方法和 sort 方法一样，适用于各种基本数据类型以及对象。

注意：二分法是对以及有序的数组进行查找（比如先用Arrays.sort()进行排序，然后调用此方法进行查找）。找到元素返回下标，没有则返回 -1

4、copyOf

拷贝数组元素。底层采用 System.arraycopy() 实现，这是一个native方法。

public static native void arraycopy(Object src, int srcPos,

Object dest, int destPos,

int length);

src:源数组

srcPos:源数组要复制的起始位置

dest:目的数组

destPos:目的数组放置的起始位置

length:复制的长度

注意：src 和 dest都必须是同类型或者可以进行转换类型的数组。

5、equals 和 deepEquals

①、equals

equals 用来比较两个数组中对应位置的每个元素是否相等。

②、deepEquals

也是用来比较两个数组的元素是否相等，不过 deepEquals 能够进行比较多维数组，而且是任意层次的嵌套数组。

6、fill

该系列方法用于给数组赋值，并能指定某个范围赋值。

//给a数组所有元素赋值 val

public static void fill(int[] a, int val) {

for (int i = 0, len = a.length; i < len; i++)

a[i] = val;

}

//给从 fromIndex 开始的下标，toIndex-1结尾的下标都赋值 val,左闭右开

public static void fill(int[] a, int fromIndex, int toIndex, int val) {

rangeCheck(a.length, fromIndex, toIndex);//判断范围是否合理

for (int i = fromIndex; i < toIndex; i++)

a[i] = val;

}

7、toString 和 deepToString

toString 用来打印一维数组的元素，而 deepToString 用来打印多层次嵌套的数组元素。

int[][] arr = { { 1, 2, 3 }, { 4, 5, 6 } };

System.out.println(Arrays.deepToString(arr));

输出：[[1, 2, 3], [4, 5, 6]]

## Java native方法

* 概念

native关键字说明其修饰的方法是一个原生态方法，方法对应的实现不是在当前文件，而是在用其他语言（如C和C++）实现的文件中。Java语言本身不能对操作系统底层进行访问和操作，但是可以通过JNI接口调用其他语言来实现对底层的访问。

JNI是Java本机接口（Java Native Interface），是一个本机编程接口，它是Java软件开发工具箱（java Software Development Kit，SDK）的一部分。JNI允许Java代码使用以其他语言编写的代码和代码库。Invocation API（JNI的一部分）可以用来将Java虚拟机（JVM）嵌入到本机应用程序中，从而允许程序员从本机代码内部调用Java代码。

* native用法：

1.编写带有native声明的方法的Java类（java文件）

2.使用javac命令编译编写的Java类（class文件）

3.使用javah -jni \*\*\*\*来生成后缀名为.h的头文件（.h的文件）

4.使用其他语言（C、C++）实现本地方法

5.将本地方法编写的文件生成动态链接库（dll文件）

# JavaIO

## Java IO概述

Java IO ：Java IO即Java 输入输出系统。不管我们编写何种应用，都难免和各种输入输出相关的媒介打交道，其实和媒介进行IO的过程是十分复杂的，这要考虑的因素特别多，比如我们要考虑和哪种媒介进行IO（文件、控制台、网络），我们还要考虑具体和它们的通信方式（顺序、随机、二进制、按字符、按字、按行等等）。

流：在Java IO中，流是一个核心的概念。流从概念上来说是一个连续的数据流。你既可以从流中读取数据，也可以往流中写数据。流与数据源或者数据流向的媒介相关联。在Java IO中流既可以是字节流(以字节为单位进行读写)，也可以是字符流(以字符为单位进行读写)。

IO相关的媒介：Java的IO包主要关注的是从原始数据源的读取以及输出原始数据到目标媒介。以下是最典型的数据源和目标媒介

文件

管道

网络连接

内存缓存

System.in, System.out, System.error(注：Java标准输入、输出、错误输出)

## Java IO类库的框架

* Java IO的类型：

虽然java IO类库庞大，但总体来说其框架还是很清楚的。从是读媒介还是写媒介的维度看，Java IO可以分为：

输入流：InputStream和Reader

输出流：OutputStream和Writer

而从其处理流的类型的维度上看，Java IO又可以分为：

字节流：InputStream和OutputStream

字符流：Reader和Writer

|  | 字节流 | 字符流 |
| --- | --- | --- |
| 输入流 | InputStream | Reader |
| 输出流 | OutputStream | Writer |

Source 🡪 InputStream/Reader 🡪 Program

Program 🡪 OutputStream 🡪 Destination

* IO 类库

文件访问

网络访问

内存缓存访问

线程内部通信(管道)

缓冲

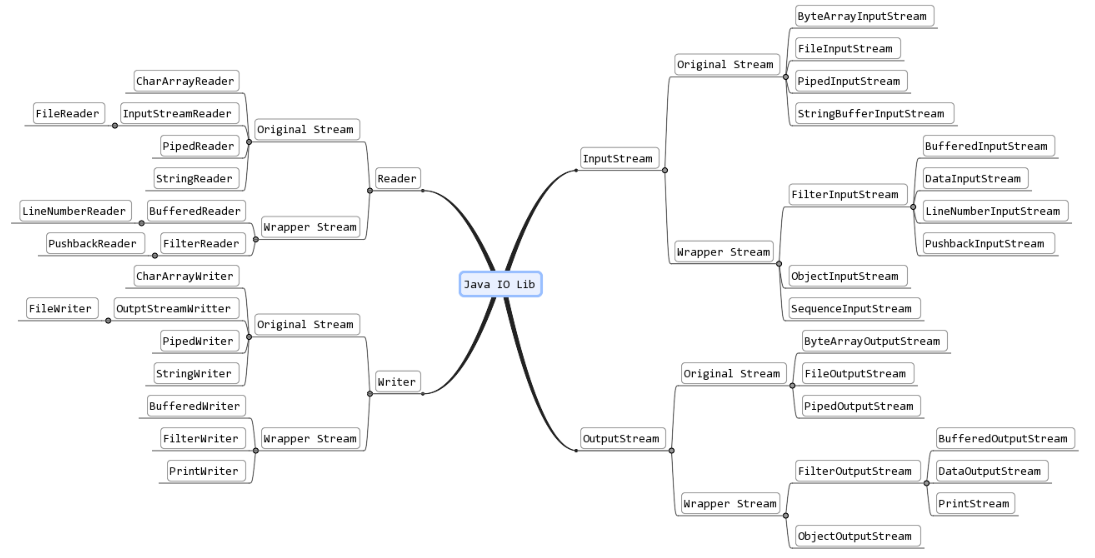
过滤

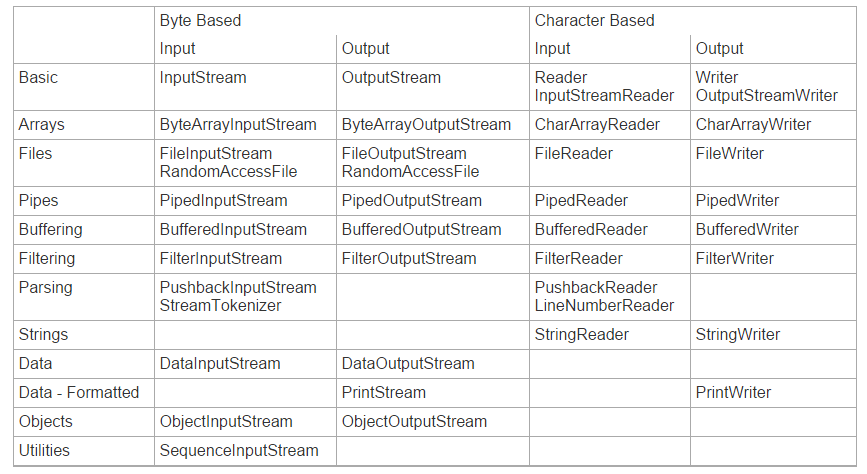
解析

读写文本 (Readers / Writers)

读写基本类型数据 (long, int etc.)

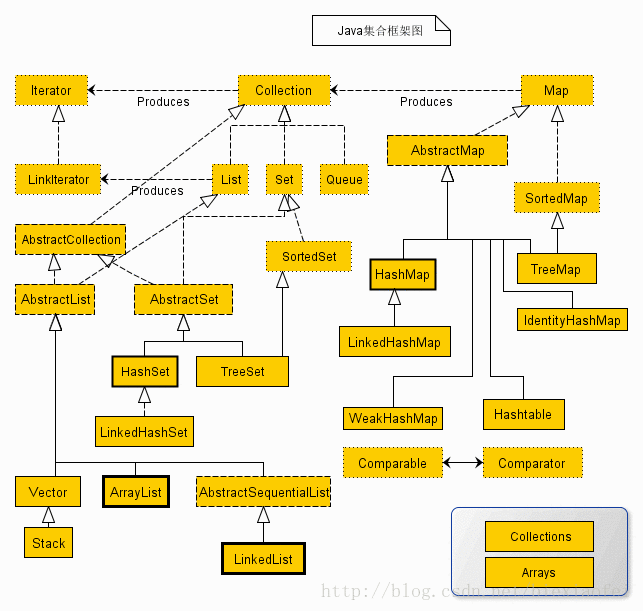
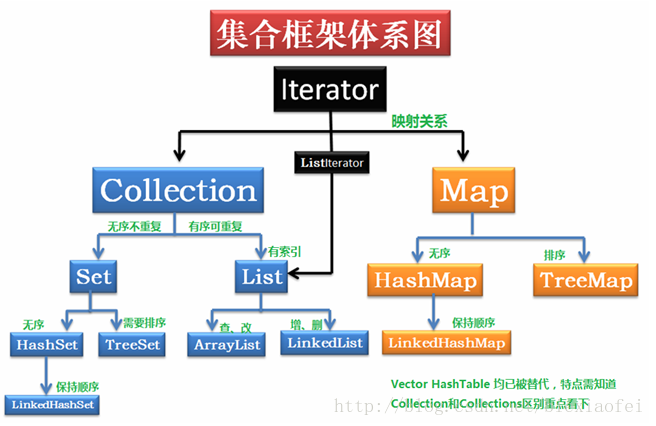
读写对象





# 集合框架

## 集合接口、类的继承关系图

****

## 迭代器和枚举的区别

区别：

(01) 函数接口不同

Enumeration只有2个函数接口。通过Enumeration，我们只能读取集合的数据，而不能对数据进行修改。

Iterator只有3个函数接口。Iterator除了能读取集合的数据之外，也能数据进行删除操作。

(02) Iterator支持fail-fast机制，而Enumeration不支持。

Enumeration 是JDK 1.0添加的接口。使用到它的函数包括Vector、Hashtable等类，这些类都是JDK 1.0中加入的，Enumeration存在的目的就是为它们提供遍历接口。Enumeration本身并没有支持同步，而在Vector、Hashtable实现Enumeration时，添加了同步。

而Iterator 是JDK 1.2才添加的接口，它也是为了HashMap、ArrayList等集合提供遍历接口。Iterator是支持fail-fast机制的：当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。

Java API规范建议，对于较新的程序，Iterator应优先于Enumeration，因为“ Iterator在Java集合框架中代替Enumeration。”

## for foreach关键字的区别，性能

普通for循环语法：

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

    System.out.println(intergers[i]);

}

foreach 循环语法：

**for**(Integer in : integers){

    System.out.println(in);

}

循环ArrayList时，普通for循环比foreach循环花费的时间要少一点；循环LinkList时，普通for循环比foreach循环花费的时间要多很多。

当我将循环次数提升到一百万次的时候，循环ArrayList，普通for循环还是比foreach要快一点；但是普通for循环在循环LinkList时，程序直接卡死。

需要**循环数组结构**的数据时，建议**使用普通for循环**，因为for循环采用下标访问，对于数组结构的数据来说，采用下标访问比较好。

## List，Map，Set分别有什么特点

set(集)、List（列表）、Map(映射)

List:

可以允许重复的对象

可以插入多个null元素

是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序

常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

Set:

不允许重复对象

无序容器，你无法保证每个元素的存储顺序，TreeSet通过Comparator或者Comparable维护了一个排序顺序

只允许一个null

Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

Map:

1.Map不是collection的子接口或者实现类。Map是一个接口。

2.Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。

3. TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。

4. Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。

5.Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

用索引对容器中元素进行访问List

容器中的元素能够按照他们插入的次序进行存储，List

保证插入元素的唯一性，不想有重复值的出现，Set

以键值对的形式进行存储，Map

**总结**

1. 如果涉及到堆栈，队列等操作，应该考虑用List，对于需要快速插入，删除元素，应该使用LinkedList，如果需要快速随机访问元素，应该使用ArrayList。

2. 如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，如果多个线程可能同时操作一个类，应该使用同步的类。

3. 在除需要排序时使用TreeSet,TreeMap外,都应使用HashSet,HashMap,因为他们 的效率更高。

4. 要特别注意对哈希表的操作，作为key的对象要正确复写equals和hashCode方法。

5. 容器类仅能持有对象引用（指向对象的指针），而不是将对象信息copy一份至数列某位置。一旦将对象置入容器内，便损失了该对象的型别信息。

6. 尽量返回接口而非实际的类型，如返回List而非ArrayList，这样如果以后需要将ArrayList换成LinkedList时，客户端代码不用改变。这就是针对抽象编程。

## List接口的实现类（ArrayList, LinkedList，Vector）

public class **ArrayList**<E>

extends AbstractList<E>

implements List<E>,RandomAccess,Cloneable,java.io.Serializable

public class **LinkedList**<E>

extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, java.io.Serializable

public class **Vector**<E>

extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

**ArrayList：**

底层数据结构使数组结构，查询速度快，增删改慢，默认容量为10

public class ArrayList<E>

extends AbstractList<E>

implements List<E>,RandomAccess,Cloneable,java.io.Serializable

* ArrayList<E>：说明ArrayList支持泛型。
* extends AbstractList<E> ：继承了AbstractList。AbstractList提供List接口的骨干实现，以最大限度地减少“随机访问”数据存储（如ArrayList）实现Llist所需的工作。
* implements List<E>：实现了List。实现了所有可选列表操作。
* implements RandomAccess：表明ArrayList支持快速（通常是固定时间）随机访问。此接口的主要目的是允许一般的算法更改其行为，从而在将其应用到随机或连续访问列表时能提供良好的性能。
* implements Cloneable：表明其可以调用clone()方法来返回实例的field-for-field拷贝。
* implements java.io.Serializable：表明该类具有序列化功能。

构造方法：

* ArrayList(int initialCapacity)：构造一个指定容量为capacity的空ArrayList。
* ArrayList()：构造一个初始容量为 10 的空列表。
* ArrayList(Collection<? extends E> c)：构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。

ArrayList有以下核心方法

|  |  |
| --- | --- |
| 方法名 | 时间复杂度 |
| get(int index) | O(1) |
| add(E e) | O(1) |
| add(add(int index, E element)) | O(n) |
| remove(int index) | O(n) |
| set(int index, E element) | O(1) |

get(index):获取指定位置的元素

因为ArrayList底层是数组，所以它的get方法非常简单，先是判断一下有没有越界，之后就直接通过数组下标来获取元素。get方法的**时间复杂度是O(1)**。

add(E e)：将指定元素添加到此列表的尾部

* 空间检查，如果有需要进行扩容
* 插入元素

扩容步骤：

* 进行空间检查，决定是否进行扩容，以及确定最少需要的容量
* 如果确定扩容，就执行grow(int minCapacity)，minCapacity为最少需要的容量
* 第一次扩容，逻辑为newCapacity = oldCapacity + (oldCapacity >> 1);即在原有的容量基础上增加一半。
* 第一次扩容后，如果容量还是小于minCapacity，就将容量扩充为minCapacity。
* 对扩容后的容量进行判断，如果大于允许的最大容量MAX\_ARRAY\_SIZE，则将容量再次调整为MAX\_ARRAY\_SIZE。至此扩容操作结束。

**add( int index, E element)**：在指定位置插入元素，当前位置元素和index之后的元素向后移一位

* 越界检查
* 空间检查，如果有需要进行扩容
* 插入元素

**remove( int index)**：删除此列表指定位置的元素，索引之后的元素向左移一位

* 检查索引是否越界。如果参数指定索引index>=size，抛出一个越界异常
* 将索引大于index的元素左移一位（左移后，该删除的元素就被覆盖了，相当于被删除了）。
* 将索引为size-1处的元素置为null（为了让GC（垃圾回收）起作用）。

**set( int index, E element)**：替换指定位置的元素

**LinkList：**

底层使用双向链表结构，增删速度快，查询稍慢；查找时当给定索引值大于list长度一半时从尾节点开始查找，小于list长度一半时从头节点开始查找。

public class LinkedList<E>

extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, java.io.Serializable

**Vector：**

public class Vector<E>

extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

底层是数组结构，Vector是线程同步的，所以它也是线程安全的。而ArratList是线程异步的，不安全。如果不考虑安全因素，一般用Arralist效率比较高；

构造方法：

* 构造一个指定容量为capacity、自增容量为capacityIncrement的空vector。public Vector(int initialCapacity, int capacityIncrement) ；
* 构造一个指定容量为initialCapacity、自增容量为0的空vector。public Vector(int initialCapacity)
* 构造一个指定容量为10、自增容量为0的空vect，public Vector(int initialCapacity)
* 使用指定的Collection构造vector，public Vector(Collection<? extends E> c)

copyInto:将vector中的所有元素拷贝到指定的数组anArray中

public synchronized void copyInto(Object[] anArray) { System.arraycopy(elementData, 0, anArray, 0, elementCount); }

**trimToSize()**:将底层数组的容量调整为当前vector实际元素的个数，来释放空间。

扩容：

该方法的实现和ArrayList中大致相同。不同的是在第一次扩容时，vector的逻辑是：

newCapacity = oldCapacity + ((capacityIncrement > 0) ?

capacityIncrement : oldCapacity);

即如果capacityIncrement>0，就加capacityIncrement，如果不是就增加一倍。

而ArrayList的逻辑是：

newCapacity = oldCapacity + (oldCapacity >> 1);

即增加现有的一半。

arraylist和linkedlist

1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

2.对于随机访问get和set，ArrayList性能优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数 据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

Vector和ArrayList

1，vector是线程同步的，所以它也是线程安全的，而arraylist是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用arraylist效率比较高。

2，如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度的50%.如过在集合中使用数据量比较大的数据，用vector有一定的优势。

3.Vector可以设置capacityIncrement，而ArrayList不可以，从字面理解就是capacity容量，Increment增加，容量增长的参数。

## Set接口的实现类（HashSet,LinkedHashSet,TreeSet）

Set不允许包含相同的元素，如果试图把两个相同元素加入同一个集合中，add方法返回false。

Set判断两个对象相同不是使用==运算符，而是根据equals方法。也就是说，只要两个对象用equals方法比较返回true，Set就不会接受这两个对象。

HashSet与TreeSet都是基于Set接口的实现类。其中TreeSet是Set的子接口SortedSet的实现类。Set接口及其子接口、实现类的结构如下所示：

Set接口：

HashSet实现类

LinkedHashSet实现类

SortedSet接口——>TreeSet实现类

HashSet：HashSet继承自AbstractSet，实现了Set接口。内部使用HashMap来存储数据，数据存储在HashMap的key中，value都是同一个默认值：

LinkedHashSet：LinkedHashSet继承自HashSet，LinkedHashSet继承了HashSet，又基于LinkedHashMap来实现。LinkedHashSet底层使用LinkedHashMap的key来保存所有元素，从而维护着一个运行于所有元素的双向链表。该双向链表定义了迭代顺序，该顺序分为插入顺序和访问顺序。

TreeSet：TreeSet继承自AbstractSet，又基于TreeMap实现，因为TreeSet底层使用一个TreeMap，**采用树结构实现(红黑树算法)**。元素是按顺序进行排列，它的元素存储在TreeMap的key中，保证了不可重复性。同时，TreeSet中的元素也是排好序的，操作的时间复杂度也与TreeMap相同。与HashSet类似的是，TreeSet的大部分方法也是直接调用TreeMap中的方法。

* HashSet和TreeSet的区别

HashSet

不能保证元素的排列顺序，顺序有可能发生变化

不是同步的

集合元素可以是null,但只能放入一个null

当向HashSet集合中存入一个元素时，HashSet会调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据 hashCode值来决定该对象在HashSet中存储位置。

简单的说，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()方法返回值相等

TreeSet

（1）TreeSet中不允许使用null元素！在添加的时候如果添加null，则会抛出NullPointerException异常。

（2）TreeSet是基于TreeMap实现的。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法。

（3）TreeSet是非同步的方法。 它的iterator 方法返回的迭代器是fail-fast的。

（4）TreeSet不支持快速随机遍历，只能通过迭代器进行遍历！TreeSet判断两个对象不相等的方式是两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0

最重要

1、HashSet是一个无序的集合，基于HashMap实现；TreeSet是一个有序的集合，基于TreeMap实现。

2、HashSet集合中允许有null元素，TreeSet集合中不允许有null元素。

3、HashSet和TreeSet都是非同步！在使用Iterator进行迭代的时候要注意fail-fast。

4、HashSet要求放入的对象必须实现HashCode()方法，放入的对象，是以hashcode码作为标识的，而具有相同内容的 String对象，hashcode是一样，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例 。

## Map接口的实现类

HashMap LinkedHashMap HashTable TreeMap

Map接口：

HashMap（类）——>LinkedHashMap

HaShTable（类）

SortedMap接口——>TreeMap实现类

* HashMap

public class HashMap<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

* LinkedHashMap

public class LinkedHashMap<K,V>

extends HashMap<K,V>

implements Map<K,V>

* HashTable

public class Hashtable<K,V>

extends Dictionary<K,V>

implements Map<K,V>, Cloneable, java.io.Serializable

* TreeMap

public class TreeMap<K,V>

extends AbstractMap<K,V>

implements NavigableMap<K,V>, Cloneable, java.io.Serializable

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。

TreeMap 继承于AbstractMap，所以它是一个Map，即一个key-value集合。

TreeMap 实现了NavigableMap接口，意味着它支持一系列的导航方法。比如返回有序的key集合。

TreeMap 实现了Cloneable接口，意味着它能被克隆。

TreeMap 实现了java.io.Serializable接口，意味着它支持序列化。

TreeMap基于红黑树（Red-Black tree）实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。

另外，TreeMap是非同步的。 它的iterator 方法返回的迭代器是fail-fastl的。

## Queue接口的方法

boolean add(E e); 增加一个元素如果队列已满，则抛出一个IllegalStateException（非法语句）异常

E element(); 返回列表头部元素，如果列表为空，则抛出一个NoSuchElementException 异常

**boolean offer(E e);** 添加一个元素并返回true,如果队列已满，则返回false

**E peek();** 返回队列的头部元素， 如果队列为空，则返回null

**E poll();** 移除并返回队列的头部元素，如果队列的头部为空，则返回null

E remove(); 移除并返回队列的头部元素， 如果队列为空，则抛出NoSuchElementException 异常

## HashMap，Hashtable区别

public class **Hashtable**<K,V> extends Dictionary<K,V>

implements Map<K,V>, Cloneable, java.io.Serializable

public class **HashMap**<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

public class **ConcurrentHashMap**<K,V> extends AbstractMap<K,V>

implements ConcurrentMap<K,V>, Serializable

HashTable的方法前面都有synchronized来同步，是线程安全的；HashMap未经同步，是非线程安全的。

HashTable不允许null值(key和value都不可以) ；HashMap允许null值(key和value都可以)。

二者的基类不一样。HashMap派生于AbstractMap，HashTable派生于Dictionary。它们都实现Map, Cloneable, Serializable这些接口。

HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。HashMap默认的初始化大小为16，之后每次扩充为原来的2倍，大小一定是2第幂

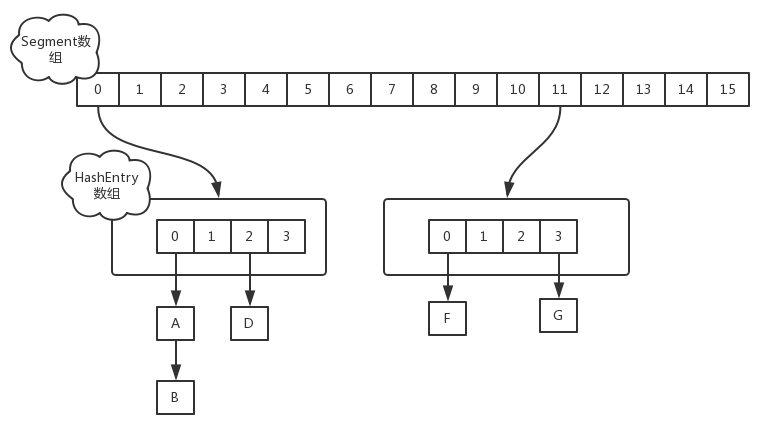
哈希值的使用不同，HashTable直接使用对象的hashCode； HashMap重新计算hash值，而且用与代替求模。

## ConcurrentHashMap

<https://www.cnblogs.com/study-everyday/p/6430462.html#autoid-2-0-0>

JDK1.7 实现：

ConcurrentHashMap的数据结构是由一个Segment数组和多个HashEntry组成，如下图所示：

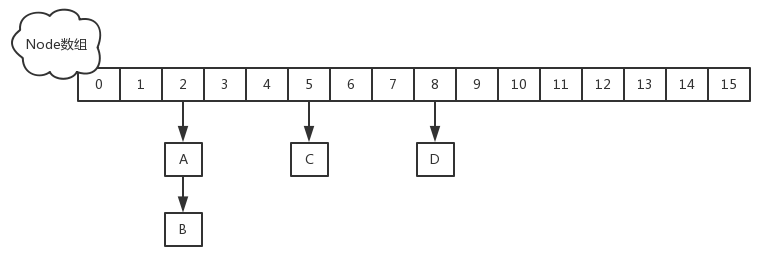


Segment数组的意义就是将一个大的table分割成多个小的table来进行加锁，也就是上面的提到的锁分离技术，而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样

static class Segment<K,V> extends ReentrantLock implements Serializable

JDK1.8实现：

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作，整个看起来就像是优化过且线程安全的HashMap，虽然在JDK1.8中还能看到Segment的数据结构，但是已经简化了属性，只是为了兼容旧版本



put操作过程:

如果没有初始化就先调用initTable（）方法来进行初始化过程

如果没有hash冲突就直接CAS插入

如果还在进行扩容操作就先进行扩容

如果存在hash冲突，就加锁来保证线程安全，这里有两种情况，一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入，

最后一个如果该链表的数量大于阈值8，就要先转换成黑红树的结构，break再一次进入循环

如果添加成功就调用addCount（）方法统计size，并且检查是否需要扩容

get操作：

计算hash值，定位到该table索引位置，如果是首节点符合就返回

如果遇到扩容的时候，会调用标志正在扩容节点ForwardingNode的find方法，查找该节点，匹配就返回

以上都不符合的话，就往下遍历节点，匹配就返回，否则最后就返回null

总结：

其实可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树,相对而言，总结如下思考

JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）

JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了

JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档

JDK1.8为什么使用内置锁synchronized来代替重入锁ReentrantLock，我觉得有以下几点

因为粒度降低了，在相对而言的低粒度加锁方式，synchronized并不比ReentrantLock差，在粗粒度加锁中ReentrantLock可能通过Condition来控制各个低粒度的边界，更加的灵活，而在低粒度中，Condition的优势就没有了

JVM的开发团队从来都没有放弃synchronized，而且基于JVM的synchronized优化空间更大，使用内嵌的关键字比使用API更加自然

在大量的数据操作下，对于JVM的内存压力，基于API的ReentrantLock会开销更多的内存，虽然不是瓶颈，但是也是一个选择依据

# 多线程

## 线程的生命周期、几种状态之间的转换、分别是什么时机

Java中的线程的生命周期大体可分为5种状态。

1.新建(NEW)：新创建了一个线程对象。

2. 可运行(RUNNABLE)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。

3. 运行(RUNNING)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。

4. 阻塞(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

* (一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
* (二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
* (三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

5. 死亡(DEAD)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

**注意：线程的死亡并不意味着线程对象的销毁和回收。线程的死亡指的是当线程的run()方法结束后，该线程就无法被重用和启动，但它的对象还存在并且它的属性和方法还一样可以被使用。**

一.线程的状态图



二、初始状态

* 实现Runnable接口和继承Thread可以得到一个线程类，new一个实例出来，线程就进入了初始状态

三、可运行状态

* 可运行状态只是说你资格运行，调度程序没有挑选到你，你就永远是可运行状态。
* 调用线程的start()方法，此线程进入可运行状态。
* 当前线程sleep()方法结束，其他线程join()结束，等待用户输入完毕，某个线程拿到对象锁，这些线程也将进入可运行状态。
* 当前线程时间片用完了，调用当前线程的yield()方法，当前线程进入可运行状态。
* 锁池里的线程拿到对象锁后，进入可运行状态。

四、运行状态

* 线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。

五.死亡状态

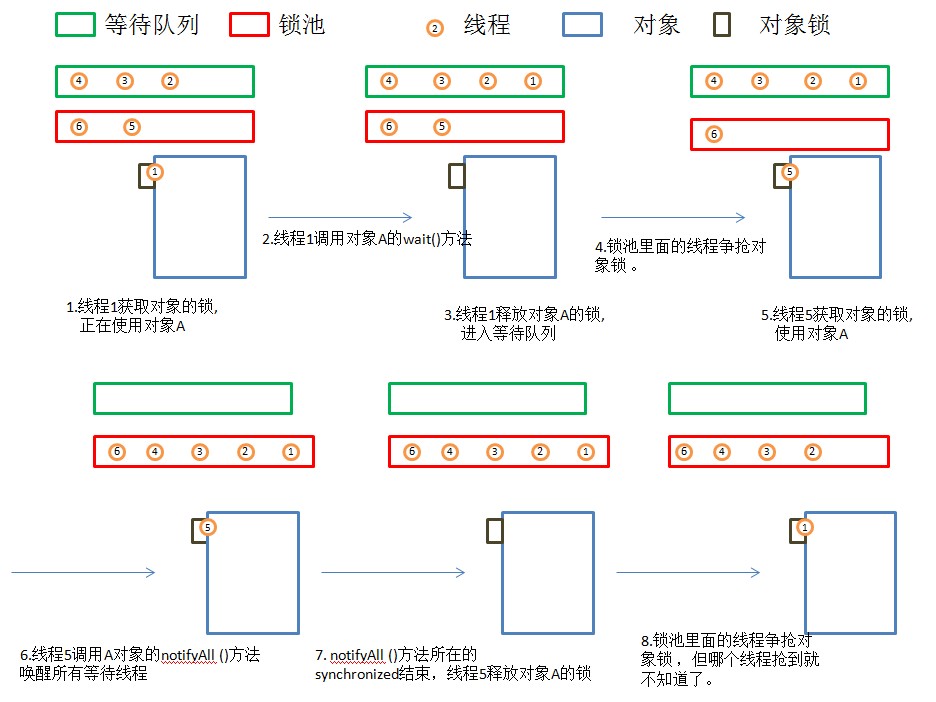
* 当线程的run()方法完成时，或者主线程的main()方法完成时，我们就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。
* 在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

六.阻塞状态

* 当前线程T调用Thread.sleep()方法，当前线程进入阻塞状态。
* 运行在当前线程里的其它线程t2调用join()方法，当前线程进入阻塞状态。
* 等待用户输入的时候，当前线程进入阻塞状态。

七.等待队列(本是Object里的方法，但影响了线程)

* 调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内。
* 与等待队列相关的步骤和图
  + 线程1获取对象A的锁，正在使用对象A。
  + 线程1调用对象A的wait()方法。
  + 线程1释放对象A的锁，并马上进入等待队列。
  + 锁池里面的对象争抢对象A的锁。
  + 线程5获得对象A的锁，进入synchronized块，使用对象A。
  + 线程5调用对象A的notifyAll()方法，唤醒所有线程，所有线程进入锁池。||||| 线程5调用对象A的notify()方法，唤醒一个线程，不知道会唤醒谁，被唤醒的那个线程进入锁池。
  + notifyAll()方法所在synchronized结束，线程5释放对象A的锁。
  + 锁池里面的线程争抢对象锁，但线程1什么时候能抢到就不知道了。||||| 原本锁池+第6步被唤醒的线程一起争抢对象锁。



八.锁池状态

* + 当前线程想调用对象A的同步方法时，发现对象A的锁被别的线程占有，此时当前线程进入锁池状态。简言之，锁池里面放的都是想争夺对象锁的线程。
  + 当一个线程1被另外一个线程2唤醒时，1线程进入锁池状态，去争夺对象锁。
  + 锁池是在同步的环境下才有的概念，一个对象对应一个锁池。

九.几个方法的比较

* + Thread.sleep(long millis)，一定是当前线程调用此方法，当前线程进入阻塞，但不释放对象锁，millis后线程自动苏醒进入可运行状态。作用：给其它线程执行机会的最佳方式。
  + Thread.yield()，一定是当前线程调用此方法，当前线程放弃获取的cpu时间片，由运行状态变会可运行状态，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.yield()不会导致阻塞。
  + t.join()/t.join(long millis)，当前线程里调用其它线程1的join方法，当前线程阻塞，但不释放对象锁，直到线程1执行完毕或者millis时间到，当前线程进入可运行状态。
  + obj.wait()，当前线程调用对象的wait()方法，当前线程释放对象锁，进入等待队列。依靠notify()/notifyAll()唤醒或者wait(long timeout)timeout时间到自动唤醒。
  + obj.notify()唤醒在此对象监视器上等待的单个线程，选择是任意性的。notifyAll()唤醒在此对象监视器上等待的所有线程。

## 创建线程的三种方式，各自特点、优缺点

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。

一、继承Thread类创建线程类

1.重写run方法。该run（）方法的方法体就代表了线程需要完成的任务。

2.创建Thread子类的实例。

3.调用线程对象的start（）方法来启动该线程

二、实现Runnable接口创建线程类

1.定义Runnable的实现类，重写run（）方法。

2.创建Runnable实现类的实例，并以此作为Thread的target来创建对象，该对象才是真正的线程对象。

三、使用Callable和Future创建线程

1.创建Callable接口的实现类，并实现Call（）方法，该方法将作为线程执行体，且该方法有返回值，再创建Callable实现类的实例。从Java8开始，可以直接使用Lambda表达式创建Callable对象。

2.使用FutureTask来包装Callable对象，该FutureTask对象封装了该Callable对象的call方法的返回值。

3.使用FutureTask对象作为Thread对象的target创建并启动新线程。

4.调用FutureTask对象的get（）方法来获取子线程执行结束后的返回值。

四、浅谈三种方式优劣势

通过继承Thread类或实现Runnable、Callable接口都可以实现多线程，不过实现Runnable接口与实现Callable接口的方式基本相同，只是Callable接口里定义的方法有返回值，可以声明抛出异常而已。因此可以将实现Runnable接口和实现Callable接口归为一种方式。这种方式与继承Thread方式之间的主要差别如下。

1.采用实现Runnable、Callable接口的方式创建多线程的优缺点：

优势：（1）线程类只是实现了Runnable接口与Callable接口，还可以继承其他类。

（2）在这种方式下，多个线程可以共享一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势：编程稍稍复杂，如果需要访问当前线程，则必须使用Thread.currentThread（）方法。

2.采用继承Thread类的方法创建多线程的优缺点：

劣势：因为线程类已经继承了Thread类，所以不能再继承其他父类。

优势：编写简单，如果需要访问当前线程，则无须使用Thread.currentThread（）方法，直接使用this即可获得当前线程。

## Callable和Runable的区别，Callable和Future类如何联合使用

Runnable和Callable的区别是

(1)Callable规定的方法是call(),Runnable规定的方法是run().

(2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得

(3)call方法可以抛出异常，run方法不可以

(4)运行Callable任务可以拿到一个Future对象，Future 表示异步计算的结果。

多线程Future和Callable类结合

在多线程编程的时候，要想在开启的线程都结束时，同时获取每个线程中返回的数据然后在做统一处理，在这样的需求下，应使用Future和Callable的组合。

## 守护线程是哈

Java的线程分为两种：User Thread(用户线程)、Daemon Thread(守护线程)。

只要JVM实例中尚存任何一个非守护线程，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作，Daemon作用是为其它线程提供便利服务，守护线程最典型的就是GC（垃圾回收器）。

守护线程是为其他线程的运行提供服务的线程。

setDaemon(boolean on)方法可以方便的设置线程的Daemon模式，true为守护模式，false为用户模式。

## 如何停止一个线程

* 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
* 使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
* 使用interrupt方法中断线程

调用interrupt方法是在当前线程中打了一个停止标志，并不是真的停止线程

Thread.java类中提供了两种方法：

this.interrupted(): 测试当前线程是否已经中断；

this.isInterrupted(): 测试线程是否已经中断；

测试当前线程是否已经中断。线程的中断状态由该方法清除。 换句话说，如果连续两次调用该方法，则第二次调用返回false

## 什么是线程安全，如何保证线程安全

线程安全就是多线程访问同一代码，不会产生不确定的结果。

对非安全的代码进行加锁控制

使用线程安全的类

多线程并发的情况下，线程共享的变量改为方法级的局部变量

## sleep和wait的区别(考察的方向是是否会释放锁)

sleep()方法是Thread类中方法，而wait()方法是Object类中的方法。  
sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态，在调用sleep()方法的过程中，线程不会释放对象锁。而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备。

## 多线程如何进行信息交互

void notify() 唤醒在此对象监视器上等待的单个线程。

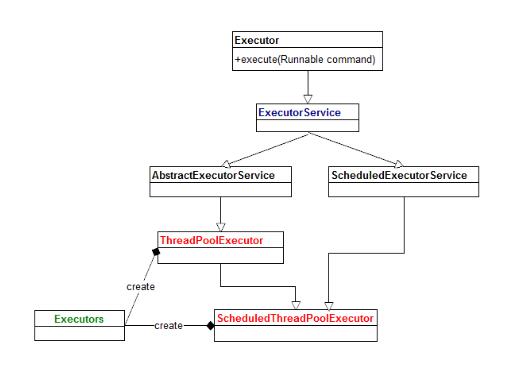
void notifyAll() 唤醒在此对象监视器上等待的所有线程。

void wait() 导致当前的线程等待，直到其他线程调用此对象的notify()方法或notifyAll()方法。

void wait(long timeout) 导致当前的线程等待，直到其他线程调用此对象的notify()方法或notifyAll()方法，或者超过指定的时间量。

void wait(long timeout, int nanos) 导致当前的线程等待，直到其他线程调用此对象的notify()方法或notifyAll()方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

# Executor框架



* **Executor：**是Java线程池的超级接口；提供一个execute(Runnable command)方法。我们一般用它的继承接口ExecutorService
* **Executors：**是java.util.concurrent包下的一个类，提供了若干个静态方法，用于生成不同类型的线程池。Executors一共可以创建下面这四类线程池：
* newFixedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
* newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
* newScheduledThreadPool 创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。
* newSingleThreadExecutor 创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
* **ExecutorService：**它是线程池定义的一个接口，继承Executor。有两个实现类，分别为ThreadPoolExecutor,ScheduledThreadPoolExecutor。

ExecutorService常用的几个方法：

* + execute(Runnable)

从父类继承过来的方法，方法接收一个Runnable实例，并且异步的执行

* + submit(Runnable)

返回一个Future对象，通过返回的Future对象我们可以检查提交的任务是否执行完毕。

* + submit(Callable)

与submit(Callable)类似，也会返回一个Future对象，但是除此之外，submit(Callable)接收的是一个Callable的实现，Callable接口中的call()方法有一个返回值，可以返回任务的执行结果，而Runnable接口中的run()方法是void的，没有返回值。

* + invokeAny(...)

方法接收的是一个Callable的集合，执行这个方法不会返回Future，但是会返回所有Callable任务中其中一个任务的执行结果。这个方法也无法保证返回的是哪个任务的执行结果，反正是其中的某一个。

* + invokeAll(...)

与 invokeAny(...)类似也是接收一个Callable集合，但是前者执行之后会返回一个Future的List，其中对应着每个Callable任务执行后的Future对象。

* + shutdown()

我们使用完成ExecutorService之后应该关闭它，否则它里面的线程会一直处于运行状态。

# 锁

## Java中锁分类

公平锁/非公平锁、可重入锁、独享锁/共享锁、互斥锁/读写锁、乐观锁/悲观锁、分段锁

偏向锁/轻量级锁/重量级锁、自旋锁

上面是很多锁的名词，这些分类并不是全是指锁的状态，有的指锁的特性，有的指锁的设计

Synchronized 非公平锁、可重入锁、独享锁、

* **公平锁/非公平锁**

公平锁是指多个线程按照申请锁的顺序来获取锁。

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。

对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。

对于Synchronized而言，也是一种非公平锁。由于其并不像ReentrantLock是通过AQS的来实现线程调度，所以并没有任何办法使其变成公平锁。

* 可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。说的有点抽象，下面会有一个代码的示例。

对于Java ReentrantLock而言, 他的名字就可以看出是一个可重入锁，其名字是Re entrant Lock重新进入锁。

对于Synchronized而言,也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

synchronized void setA() throws Exception{

Thread.sleep(1000);

setB();

}

synchronized void setB() throws Exception{

Thread.sleep(1000);

}

上面的代码就是一个可重入锁的一个特点，如果不是可重入锁的话，setB可能不会被当前线程执行，可能造成死锁。

* 独享锁/共享锁

独享锁是指该锁一次只能被一个线程所持有。

共享锁是指该锁可被多个线程所持有。

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。

读锁的共享锁可保证并发读是非常高效的，读写，写读 ，写写的过程是互斥的。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

对于Synchronized而言，当然是独享锁

* 互斥锁/读写锁

上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。

互斥锁在Java中的具体实现就是ReentrantLock

读写锁在Java中的具体实现就是ReadWriteLock

* 乐观锁/悲观锁

乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待并发同步的角度。

悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。

乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。

从上面的描述我们可以看出，悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。

悲观锁在Java中的使用，就是利用各种锁。

乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。

* 分段锁

分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。

我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock)。

当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

但是，在统计size的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

* 偏向锁/轻量级锁/重量级锁

这三种锁是指锁的状态，并且是针对Synchronized。在Java 5通过引入锁升级的机制来实现高效Synchronized。这三种锁的状态是通过对象监视器在对象头中的字段来表明的。

偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。

轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。

重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

* 自旋锁

在Java中，自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

## Lock接口的实现类有哪些、分别有什么特点

Lock接口有三个实现类，一个是ReentrantLock,另两个是ReentrantReadWriteLock类中的两个静态内部类ReadLock和WriteLock。

ReentrantLock 可重入锁，对于已经获取到锁的线程，再次调用lock()方法时依然可以获取锁而不被阻塞。

如果在绝对时间上，先对于锁进行获取的请求一定先被满足，那么这个锁就是公平的，反之就是非公平的。该锁同一时刻只允许一个线程来访问

ReentrantReadWriteLock读写锁，实现ReadWriteLock接口，默认构造方法为非公平模式，开发者可以设置为公平模式。读写锁在同一时刻允许可以有多个线程来访问，但在写线程访问时，所有的读线程和其他写线程被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过读写锁分离，使得并发性相比一般的排他锁有了很大的提升

## Lock接口的方法有哪些

在jdk1.5之后，并发包中新增了Lock接口(以及相关实现类)用来实现锁功能，Lock接口提供了与synchronized关键字类似的同步功能，但需要在使用时手动获取锁和释放锁。

* Lock接口的使用

Lock lock = new ReentrantLock();

lock.lock();

try{

//可能会出现线程安全的操作

}finally{

//一定在finally中释放锁

//也不能把获取锁在try中进行，因为有可能在获取锁的时候抛出异常

lock.ublock();

}

* Lock接口与synchronized关键字的区别

主要相同点：Lock能完成synchronized所实现的所有功能

主要不同点：Lock有比synchronized更精确的线程语义和更好的性能。Lock的锁定是通过代码实现的，而synchronized是在JVM层面上实现的，synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且必须在finally从句中释放。Lock还有更强大的功能，例如，它的tryLock方法可以非阻塞方式去拿锁。Lock锁的范围有局限性，块范围，而synchronized可以锁住块、对象、类。Lock接口在指定的截止时间之前获取锁，如果截止时间到了依旧无法获取锁，则返回。

* Lock接口的API

void lock() 获取锁,调用该方法当前线程将会获取锁，当锁获取后，该方法将返回。

void lockInterruptibly() throws InterruptedException 可中断获取锁，与lock()方法不同之处在于该方法会响应中断，即在锁的获取过程中可以中断当前线程

boolean tryLock() 尝试非阻塞的获取锁，调用该方法立即返回，true表示获取到锁

boolean tryLock(long time,TimeUnit unit) throws InterruptedException 超时获取锁，以下情况会返回：时间内获取到了锁，时间内被中断，时间到了没有获取到锁。

void unlock()释放锁

* 读写锁的锁降级

锁降级是指写锁降级成为读锁。**如果当前线程持有写锁，然后将其释放再获取读锁的过程不能称为锁降级。锁降级指的在持有写锁的时候再获取读锁,获取到读锁后释放之前写锁的过程。**

锁降级在某些情况下是非常必要的，主要是为了保证数据的可见性。如果当前线程不获取读锁而直接释放写锁，假设此时另外一个线程获取了写锁并修改了数据。那么当前线程无法感知该线程的数据更新

## synchronized如何使用

synchronized是Java中的关键字，是一种同步锁。它修饰的对象有以下几种：

1).修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；

2).修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；

3).修改一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；

4).修改一个类，其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

## lock和condition的使用

* Lock特性
  + lock不是Java语言内置的
  + synchronized是在JVM层面上实现的，如果代码抛出异常，JVM会自动释放锁，但是lock不行，要保证锁一定会被释放，就必须将unLock方法哦finally{}中（手动释放）
  + 在资源竞争不是很激烈的情况下，synchronized的性能要优于ReetarntLock，但是在很激烈的情况下，synchronized的性能会下降几十倍
  + ReentrantLock增加了锁

a. void lock(); // 无条件的锁;

b. void lockInterruptibly throws InterruptedException;//可中断的锁;

解释:使用ReentrantLock如果获取了锁立即返回,如果没有获取锁,当前线程处于休眠状态,直到获得锁或者当前线程可以被别的线程中断去做其他的事情;但是如果是synchronized的话,如果没有获取到锁,则会一直等待下去;

c. boolean tryLock();//如果获取了锁立即返回true,如果别的线程正持有,立即返回false,不会等待;

d. boolean tryLock(long timeout,TimeUnit unit);//如果获取了锁立即返回true,如果别的线程正持有锁,会等待参数给的时间,在等待的过程中,如果获取锁,则返回true,如果等待超时,返回false;

Condition的特性：

* Condition中的await()方法相当于Object的wait()方法，Condition中的signal()方法相当于Object的notify()方法，Condition中的signalAll()相当于Object的notifyAll()方法。不同的是，Object中的这些方法是和同步锁捆绑使用的；而Condition是需要与互斥锁/共享锁捆绑使用的。
* Condition它更强大的地方在于：能够更加精细的控制多线程的休眠与唤醒。对于同一个锁，我们可以创建多个Condition，在不同的情况下使用不同的Condition。

例如：假如多线程读/写同一个缓冲区：当向缓冲区中写入数据之后，唤醒"读线程"；当从缓冲区读出数据之后，唤醒"写线程"；并且当缓冲区满的时候，"写线程"需要等待；当缓冲区为空时，"读线程"需要等待。

* 如果采用Object类中的wait(), notify(), notifyAll()实现该缓冲区，当向缓冲区写入数据之后需要唤醒"读线程"时，不可能通过notify()或notifyAll()明确的指定唤醒"读线程"，而只能通过notifyAll唤醒所有线程(但是notifyAll无法区分唤醒的线程是读线程，还是写线程)。但是，通过Condition，就能明确的指定唤醒读线程。

## ReentrantLock 源码，为什么要把公平所和非公平所分开（谷歌程序员贴心，方便开发者调用）

* ReentrantLock的创建

默认情况下使用非公平锁

* + 支持公平锁（先进来的线程先执行）
  + 支持非公平锁（后进来的线程也可能先执行）
  + 非公平锁：ReentrantLock()或ReentrantLock(false)

final ReentrantLock lock = new ReentrantLock();

* + 公平锁：ReentrantLock(true)

final ReentrantLock lock = new ReentrantLock(true)

## AQS源码

(装逼必备)网上有一篇很好的讲解一定要看和理解

<https://www.cnblogs.com/waterystone/p/4920797.html>

## Lock和synchronized关键字的区别

主要相同点：Lock能完成synchronized所实现的所有功能

主要不通点：Lock有比synchronized更精确地线程语义和更好的性能。Lock的锁定是通过代码实现的，而synchronized是在jvm层面上实现的，synchronized会自动释放锁，而Lock需要程序员手工释放并且需要在finally从句中释放。Lock的tryLock方法可以非阻塞的方式去拿锁。Lock锁的的范围有局限性，块范围，而synchronized可以锁住块、对象、类。

## 类锁、对象锁的区别及原理，jvm怎么执行的

一段synchronized的代码被一个线程执行之前，他要先拿到执行这段代码的权限，

在Java里边就是拿到某个同步对象的锁（一个对象只有一把锁）；

如果这个时候同步对象的锁被其他线程拿走了，他（这个线程）就只能等了（线程阻塞在锁池等待队列中）。

取到锁后，他就开始执行同步代码(被synchronized修饰的代码）；

线程执行完同步代码后马上就把锁还给同步对象，其他在锁池中等待的某个线程就可以拿到锁执行同步代码了。

这样就保证了同步代码在统一时刻只有一个线程在执行。

事实上，synchronized修饰非静态方法、同步代码块的synchronized (this)用法和synchronized (非this对象)的用法锁的是对象，线程想要执行对应同步代码，需要获得对象锁。

synchronized修饰静态方法以及同步代码块的synchronized (类.class)用法锁的是类，线程想要执行对应同步代码，需要获得类锁。

## 读写锁

<https://blog.csdn.net/yanyan19880509/article/details/52435135>

## 线程间通信，interrupt标志位，抛出异常结束线程等

<https://www.cnblogs.com/hapjin/p/5492619.html>

线程通信的目的是使线程能够互相发送信号。另一方面，线程通信是线程能够等待其它线程的信号。

①同步

多个线程通过synchronized关键字这种方式实现线程间的通信

public class MyObject {

synchronized public void methodA() {

//do something....

}

synchronized public void methodB() {

//do some other thing

}

}

public class ThreadA extends Thread {

private MyObject object;

//省略构造方法

@Override

public void run() {

super.run();

object.methodA();

}

}

public class ThreadB extends Thread {

private MyObject object;

//省略构造方法

@Override

public void run() {

super.run();

object.methodB();

}

}

public class Run {

public static void main(String[] args) {

MyObject object = new MyObject();

//线程A与线程B 持有的是同一个对象:object

ThreadA a = new ThreadA(object);

ThreadB b = new ThreadB(object);

a.start();

b.start();

}

}

由于线程A和线程B持有同一个MyObject类的对象object，尽管这两个线程需要调用不同的方法，但是它们是同步执行的，比如：线程B需要等待线程A执行完了methodA()方法之后，它才能执行methodB()方法。这样，线程A和线程B就实现了 通信。

这种方式，本质上就是“共享内存”式的通信。多个线程需要访问同一个共享变量，谁拿到了锁（获得了访问权限），谁就可以执行。

② while轮询的方式

在这种方式下，线程A不断地改变条件，线程ThreadB不停地通过while语句检测这个条件(list.size()==5)是否成立 ，从而实现了线程间的通信。但是这种方式会浪费CPU资源。之所以说它浪费资源，是因为JVM调度器将CPU交给线程B执行时，它没做啥“有用”的工作，只是在不断地测试 某个条件是否成立。就类似于现实生活中，某个人一直看着手机屏幕是否有电话来了，而不是： 在干别的事情，当有电话来时，响铃通知TA电话来了。关于线程的轮询的影响

③ wait/notify机制

线程A要等待某个条件满足时(list.size()==5)，才执行操作。线程B则向list中添加元素，改变list 的size。

A,B之间如何通信的呢？也就是说，线程A如何知道 list.size() 已经为5了呢？

这里用到了Object类的 wait() 和 notify() 方法。

当条件未满足时(list.size() !=5)，线程A调用wait() 放弃CPU，并进入阻塞状态。---不像②while轮询那样占用CPU

当条件满足时，线程B调用 notify()通知 线程A，所谓通知线程A，就是唤醒线程A，并让它进入可运行状态。

这种方式的一个好处就是CPU的利用率提高了。

但是也有一些缺点：比如，线程B先执行，一下子添加了5个元素并调用了notify()发送了通知，而此时线程A还执行；当线程A执行并调用wait()时，那它永远就不可能被唤醒了。因为，线程B已经发了通知了，以后不再发通知了。这说明：通知过早，会打乱程序的执行逻辑。

④管道通信就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

具体就不介绍了。分布式系统中说的两种通信机制：共享内存机制和消息通信机制。感觉前面的①中的synchronized关键字和②中的while轮询 “属于” 共享内存机制，由于是轮询的条件使用了volatile关键字修饰时，这就表示它们通过判断这个“共享的条件变量“是否改变了，来实现进程间的交流。

而管道通信，更像消息传递机制，也就是说：通过管道，将一个线程中的消息发送给另一个。

## Volatile关键字实现原理，为什么不是线程安全的，使用场景等，优点是什么

原子性、可见性、有序性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性：即程序执行的顺序按照代码的先后顺序执行。

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

**不具备原子性**

原理实现机制

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

使用volatile的场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

事实上，我的理解就是上面的2个条件需要保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行。

①状态标记量

volatile boolean flag = false;

//线程1

while(!flag){

doSomething();

}

//线程2

public void setFlag() {

flag = true;

}

根据状态标记终止线程

②.单例模式中的double check

class Singleton{

private volatile static Singleton instance = null;

private Singleton() {

}

public static Singleton getInstance() {

if(instance==null) {

synchronized (Singleton.class) {

if(instance==null)

instance = new Singleton();

}

}

return instance;

}

}

为什么要使用volatile 修饰instance？

主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情:

1.给 instance 分配内存

2.调用 Singleton 的构造函数来初始化成员变量

3.将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）。

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错

## CAS算法

## 死锁

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的原因：

一.因为系统资源不足。

二.进程运行推进的顺序不合适。

三.资源分配不当。

产生死锁的四个必要条件：

一.互斥条件：所谓互斥就是进程在某一时间内独占资源。

二.请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

三.不剥夺条件:进程已获得资源，在末使用完之前，不能强行剥夺。

四.循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

## 死锁的预防

打破死锁产生的四个必要条件中的一个或几个，保证系统不会进入死锁状态。

一.打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。

二.打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

三.打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。

四.打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。

# web基础

## xml解析方式

一.DOM(JAXP Crimson解析器)

二.SAX

三.JDOM

四.DOM4J

区别：

一.DOM4J性能最好，连Sun的JAXM也在用DOM4J。目前许多开源项目中大量采用DOM4J，例如大名鼎鼎的hibernate也用DOM4J来读取XML配置文件。如果不考虑可移植性，那就采用DOM4J.

二.JDOM和DOM在性能测试时表现不佳，在测试10M

文档时内存溢出。在小文档情况下还值得考虑使用DOM和JDOM。虽然JDOM的开发者已经说明他们期望在正式发行版前专注性能问题，但是从性能观点来看，它确实没有值得推荐之处。另外，DOM仍是一个非常好的选择。DOM实现广泛应用于多种编程语言。它还是许多其它与XML相关的标准的基础，因为它正式获得W3C

推荐(与基于非标准的Java模型相对)，所以在某些类型的项目中可能也需要它(如在JavaScript中使用DOM)。

三.SAX表现较好，这要依赖于它特定的解析方式－事件驱动。一个SAX检测即将到来的XML流，但并没有载入到内存(当然当XML流被读入时，会有部分文档暂时隐藏在内存中)。

## Statement和PreparedStatement之间的区别

一.PreparedStatement是预编译的,对于批量处理可以大大提高效率. 也叫JDBC存储过程

二.使用

Statement 对象。在对数据库只执行一次性存取的时侯，用Statement 对象进行处理。PreparedStatement

对象的开销比Statement大，对于一次性操作并不会带来额外的好处。

三.statement每次执行sql语句，相关数据库都要执行sql语句的编译，preparedstatement是预编译得,

preparedstatement支持批处理

## servlet生命周期及各个方法

1.Servlet 生命周期：Servlet 加载--->实例化--->服务--->销毁。

2.init（）：在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。

3.service（）：它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个HttpServlet对象，该对象的Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。

4.destroy（）： 仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

## servlet中如何自定义filter

1 Filter简介

（1）Filter也称之为过滤器，它是Servlet技术中最实用的技术，WEB开发人员通过Filter技术，对web服务器管理的所有web资源：例如Jsp, Servlet, 静态图片文件或静态 html 文件等进行拦截，从而实现一些特殊的功能。例如实现URL级别的权限访问控制、过滤敏感词汇、压缩响应信息等一些高级功能。

（2）Servlet API中提供了一个Filter接口，开发web应用时，如果编写的Java类实现了这个接口，则把这个java类称之为过滤器Filter。通过Filter技术，开发人员可以实现用户在访问某个目标资源之前，对访问的请求和响应进行拦截

2、Filter是如何实现拦截的？

Filter接口中有一个doFilter方法，当开发人员编写好Filter，并配置对哪个web资源(拦截url)进行拦截后，WEB服务器每次在调用web资源之前，都会先调用一下filter的doFilter方法，因此，在该方法内编写代码可达到如下目的：调用目标资源之前，让一段代码执行，是否调用目标资源（即是否让用户访问web资源）

4、Filter的生命周期

（1）init(FilterConfig filterConfig)throws ServletException：

和我们编写的Servlet程序一样，Filter的创建和销毁由WEB服务器负责。 web 应用程序启动时，web 服务器将创建Filter 的实例对象，并调用其init方法进行初始化（注：filter对象只会创建一次，init方法也只会执行一次。示例 ）

开发人员通过init方法的参数，可获得代表当前filter配置信息的FilterConfig对象。

（2）doFilter(ServletRequest,ServletResponse,FilterChain)

每次filter进行拦截都会执行

在实际开发中方法中参数request和response通常转换为HttpServletRequest和HttpServletResponse类型进行操作

（3）destroy()：

在Web容器卸载 Filter 对象之前被调用。

## JSP原理

1.jsp的运行原理

（1）WEB容器JSP页面的访问请求时，它将把该访问请求交给JSP引擎去处理。Tomcat中的JSP引擎就是一个Servlet程序，它负责解释和执行JSP页面。

（2）每个JSP页面在第一次被访问时，JSP引擎先将它翻译成一个Servlet源程序，接着再把这个Servlet源程序编译成Servlet的class类文件，然后再由WEB容器像调用普通Servlet程序一样的方式来装载和解释执行这个由JSP页面翻译成的Servlet程序。

（3）Tomcat 5把为JSP页面创建的Servlet源文件和class类文件放置在“apache-tomcat-5.5.26\work\Catalina\localhost\<应用程序名>\”目录中，Tomcat将JSP页面翻译成的Servlet的包名为org.apache.jsp(即：apache-tomcat-5.5.26\work\Catalina\localhost\org\apache\jsp\文件下)

2，JSP的执行过程

JSP的执行过程主要可以分为以下几点：

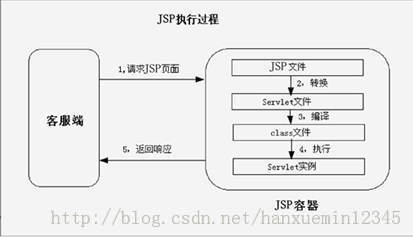
1）客户端发出请求。

2）Web容器将JSP转译成Servlet源代码。

3）Web容器将产生的源代码进行编译。

4）Web容器加载编译后的代码并执行。

5）把执行结果响应至客户端。



3，总结

JSP执行过程：

1）首先，客户端发出请求(request )，请求访问JSP网页

2）接着，JSP Container将要访问的.JSP文件 转译成Servlet的源代码（.java文件）

3）然后，将产生的Servlet的源代码（.java文件）经过编译，生成.class文件，并加载到内存执行

4）最后把结果响应(response )给客户端

执行JSP网页文件时，需要经过两个时期：转译时期(TranslationTime)和请求时期(RequestTime)。

转译时期：JSP转译成Servlet类(.class文件)。

请求时期：Servlet类(.class文件)执行后，响应结果至客户端。

转译期间主要做了两件事情：

(1)将JSP网页转译为Servlet源代码(.java)，此段称为转译时期(Translation time)；

(2)将Servlet源代码(.java)编译成Servlet类(.class)，此阶段称为编译时期(Compilation time)。

其实，JSP就是一个Servlet。

## JSP和Servlet的区别

(1)JSP经编译后就变成了“类servlet”。

(2)JSP由HTML代码和JSP标签构成，更擅长页面显示；Servlet更擅长流程控制。

(3)JSP中嵌入JAVA代码，而Servlet中嵌入HTML代码。

55.JSP的动态include和静态include#

(1)动态include用jsp:include动作实现，如<jsp:include

page="abc.jsp" flush="true" />，它总是会检查所含文件中的变化，适合用于包含动态页面，并且可以带参数。会先解析所要包含的页面，解析后和主页面合并一起显示，即先编译后包含。

(2)静态include用include伪码实现，不会检查所含文件的变化，适用于包含静态页面，如<%@

include file="qq.htm" %>，不会提前解析所要包含的页面，先把要显示的页面包含进来，然后统一编译，即先包含后编译。

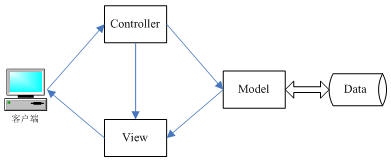
## MVC概念

MVC是Modal View Controler的缩写，原业是用于桌面程序设计中的概念，由以下三个主要部分构成，三部分各司其职。

Model :模型层（用于数据库打交道）

View :视图层（用于展示内容给用户看）

Controller :控制层（控制业务逻辑）



使用MVC的目的是将Model和View的实现代码分离，从而使同一个程序可以使用不同的表现形式。对于同一批数据（Model），可以通过不同的View以不同的形式展示给用户。

## Springmvc与Struts区别

1、Struts2是类级别的拦截， 一个类对应一个request上下文，SpringMVC是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应,所以说从架构本身上SpringMVC就容易实现restful url,而struts2的架构实现起来要费劲，因为Struts2中Action的一个方法可以对应一个url，而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。

2、由上边原因，SpringMVC的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架，方法之间不共享变量，而Struts2搞的就比较乱，虽然方法之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码 读程序时带来麻烦，每次来了请求就创建一个Action，一个Action对象对应一个request上下文。

3、由于Struts2需要针对每个request进行封装，把request，session等servlet生命周期的变量封装成一个一个Map，供给每个Action使用，并保证线程安全，所以在原则上，是比较耗费内存的。

4、 拦截器实现机制上，Struts2有以自己的interceptor机制，SpringMVC用的是独立的AOP方式，这样导致Struts2的配置文件量还是比SpringMVC大。

5、SpringMVC的入口是servlet，而Struts2是filter（这里要指出，filter和servlet是不同的。以前认为filter是servlet的一种特殊），这就导致了二者的机制不同，这里就牵涉到servlet和filter的区别了。

6、SpringMVC集成了Ajax，使用非常方便，只需一个注解@ResponseBody就可以实现，然后直接返回响应文本即可，而Struts2拦截器集成了Ajax，在Action中处理时一般必须安装插件或者自己写代码集成进去，使用起来也相对不方便。

7、SpringMVC验证支持JSR303，处理起来相对更加灵活方便，而Struts2验证比较繁琐，感觉太烦乱。

8、Spring MVC和Spring是无缝的。从这个项目的管理和安全上也比Struts2高（当然Struts2也可以通过不同的目录结构和相关配置做到SpringMVC一样的效果，但是需要xml配置的地方不少）。

9、 设计思想上，Struts2更加符合OOP的编程思想， SpringMVC就比较谨慎，在servlet上扩展。

10、SpringMVC开发效率和性能高于Struts2。

11、SpringMVC可以认为已经100%零配置。

# Java Web

## 序列化与反序列化

把对象转换为字节序列的过程称为对象的序列化。

把字节序列恢复为对象的过程称为对象的反序列化。

对象的序列化主要有两种用途：

一.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；

二.在网络上传送对象的字节序列。

当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。

## transient关键字

一个对象只要实现了Serilizable接口，这个对象就可以被序列化，java的这种序列化模式为开发者提供了很多便利，我们可以不必关系具体序列化的过程，只要这个类实现了Serilizable接口，这个类的所有属性和方法都会自动序列化。

java 的transient关键字为我们提供了便利，你只需要实现Serilizable接口，将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中。

## BIO，NIO， AIO

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 名词 | 解释 | 举例 |
| 1 | 同步 | 指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪 | 自己上街买衣服，自己亲自干这件事，别的事干不了。 |
| 2 | 异步 | 异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知（异步的特点就是通知） | 告诉朋友自己合适衣服的尺寸，大小，颜色，让朋友委托去卖，然后自己可以去干别的事。（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS） |
| 3 | 阻塞 | 所谓阻塞方式的意思是指, 当试图对该文件描述符进行读写时, 如果当时没有东西可读,或者暂时不可写, 程序就进入等待 状态, 直到有东西可读或者可写为止 | 去公交站充值，发现这个时候，充值员不在（可能上厕所去了），然后我们就在这里等待，一直等到充值员回来为止。（当然现实社会，可不是这样，但是在计算机里确实如此。） |
| 4 | 非阻塞 | 非阻塞状态下, 如果没有东西可读, 或者不可写, 读写函数马上返回, 而不会等待， | 银行里取款办业务时，领取一张小票，领取完后我们自己可以玩玩手机，或者与别人聊聊天，当轮我们时，银行的喇叭会通知，这时候我们就可以去了。 |

* BIO（IO）:同步并阻塞，服务器实现模式为**一个连接一个线程**，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
* NIO:同步非阻塞，服务器实现模式为**一个请求一个线程**，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
* AIO:异步非阻塞，服务器实现模式为**一个有效请求一个线程**，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理.AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

## servlet基础

1.servlet声明周期：

* Servlet 生命周期：Servlet 加载--->实例化--->服务--->销毁。
* init（）：在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。
* service（）：它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个HttpServlet对象，该对象的Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。
* destroy（）： 仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

2.get 和post 提交的区别

（1）get一般用于从服务器上获取数据，post一般用于向服务器传送数据

（2）请求的时候参数的位置有区别，get的参数是拼接在url后面，用户在浏览器地址栏可以看到。post是放在http包的包体中。

比如说用户注册，你不能把用户提交的注册信息用get的方式吧，那不是说把用户的注册信息都显示在Url上了吗，是不安全的。

（3）能提交的数据有区别，get方式能提交的数据只能是文本，且大小不超过1024个字节，而post不仅可以提交文本还有二进制文件。

所以说想上传文件的话，那我们就需要使用post请求方式

（4）servlet在处理请求的时候分别对应使用doGet和doPost方式进行处理请求

3. JSP和Servlet的区别

(1)JSP经编译后就变成了“类servlet”。

(2)JSP由HTML代码和JSP标签构成，更擅长页面显示；Servlet更擅长流程控制。

(3)JSP中嵌入JAVA代码，而Servlet中嵌入HTML代码。

4、doGet与doPost方法的两个参数是什么

HttpServletRequest：封装了与请求相关的信息

HttpServletResponse：封装了与响应相关的信息

5、request.getAttribute()和request.getParameter

（1）有setAttribute，没有setParameter方法

（2）getParameter获取到的值只能是字符串，不可以是对象，而getAttribute获取到的值是Object类型的。

（3）通过form表单或者url来向另一个页面或者servlet传递参数的时候需要用getParameter获取值；getAttribute只能获取setAttribute的值

（4）setAttribute是应用服务器把这个对象放到该页面所对应的一块内存当中，当你的页面服务器重定向到另一个页面的时候，应用服务器

会把这块内存拷贝到另一个页面对应的内存当中。通过getAttribute可以取得你存下的值，当然这种方法可以用来传对象。

用session也是一样的道理，这是说request和session的生命周期不一样而已。

6、JSP有哪些内置对象，作用是什么？

|  |  |
| --- | --- |
| 名称 | 作用 |
| request | 包含用户端请求的信息 |
| response | 包含服务器传回客户端的响应信息 |
| session | 与请求有关的会话期 |
| pageContext | 管理网页属性 |
| application | 服务器启动时创建，服务器关闭时停止，为多个应用程序保存信息 |
| out | 向客户端输出数据 |
| config | servlet的架构部件 |
| page | 指网页本身 |
| exception | 针对错误页面才可使用 |

7、四种会话跟踪技术作用域

（1）page：一个页面

（2）request：：一次请求

（3）session：一次会话

（4）application：服务器从启动到停止。

8、forward和redirect的区别

转发与重定向

（1）从地址栏显示来说

forward是服务器请求资源,服务器直接访问目标地址的URL,把那个URL的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,所以它的地址栏还是原来的地址.redirect是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的URL.

（2）从数据共享来说

forward:转发页面和转发到的页面可以共享request里面的数据.

redirect:不能共享数据.

（3）从运用地方来说

forward:一般用于用户登陆的时候,根据角色转发到相应的模块.

redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等.

（4）从效率来说

forward:高.

redirect:低.

## CDN

CDN的全称是Content Delivery Network，即内容分发网络。其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。通过在网络各处放置节点服务器所构成的在现有的互联网基础之上的一层智能虚拟网络，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。其目的是使用户可就近取得所需内容，解决 Internet网络拥挤的状况，提高用户访问网站的响应速度。

## B/S、C/S架构

一、C/S 架构

1、 概念

C/S 架构是一种典型的两层架构，其全程是Client/Server，即客户端服务器端架构，其客户端包含一个或多个在用户的电脑上运行的程序，而服务器端有两种，一种是数据库服务器端，客户端通过数据库连接访问服务器端的数据；另一种是Socket服务器端，服务器端的程序通过Socket与客户端的程序通信。

C/S 架构也可以看做是胖客户端架构。因为客户端需要实现绝大多数的业务逻辑和界面展示。这种架构中，作为客户端的部分需要承受很大的压力，因为显示逻辑和事务处理都包含在其中，通过与数据库的交互（通常是SQL或存储过程的实现）来达到持久化数据，以此满足实际项目的需要。

2 、优点和缺点

优点：

2.1 C/S架构的界面和操作可以很丰富。

2.2 安全性能可以很容易保证，实现多层认证也不难。

2.3 由于只有一层交互，因此响应速度较快。

缺点：

2.4 适用面窄，通常用于局域网中。

2.5 用户群固定。由于程序需要安装才可使用，因此不适合面向一些不可知的用户。

2.6 维护成本高，发生一次升级，则所有客户端的程序都需要改变。

二、B/S架构

1、概念

B/S架构的全称为Browser/Server，即浏览器/服务器结构。Browser指的是Web浏览器，极少数事务逻辑在前端实现，但主要事务逻辑在服务器端实现，Browser客户端，WebApp服务器端和DB端构成所谓的三层架构。B/S架构的系统无须特别安装，只有Web浏览器即可。

B/S架构中，显示逻辑交给了Web浏览器，事务处理逻辑在放在了WebApp上，这样就避免了庞大的胖客户端，减少了客户端的压力。因为客户端包含的逻辑很少，因此也被成为瘦客户端。

2 、优点和缺点

优点：

1）客户端无需安装，有Web浏览器即可。

2）BS架构可以直接放在广域网上，通过一定的权限控制实现多客户访问的目的，交互性较强。

3）BS架构无需升级多个客户端，升级服务器即可。

缺点：

1）在跨浏览器上，BS架构不尽如人意。

2）表现要达到CS程序的程度需要花费不少精力。

3）在速度和安全性上需要花费巨大的设计成本，这是BS架构的最大问题。

4）客户端服务器端的交互是请求-响应模式，通常需要刷新页面，这并不是客户乐意看到的。（在Ajax风行后此问题得到了一定程度的缓解）

三、B/S对C/S的改进和扩展

正如前文所说，C/S和B/S都可以进行同样的业务处理，但是B/S随着Internet技术的兴起，是对C/S结构的一种改进或者扩展的结构。相对于C/S，B/S具有如下优势：

1、分布性：可以随时进行查询、浏览等业务

2、业务扩展方便：增加网页即可增加服务器功能

3、维护简单方便：改变网页，即可实现所有用户同步更新

4、开发简单，共享性强，成本低，数据可以持久存储在云端而不必担心数据的丢失。

## Cookie和Session

ookie 和session 的区别：

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗

考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能

考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：

将登陆信息等重要信息存放为SESSION

其他信息如果需要保留，可以放在COOKIE中

# Spring

## Spring是什么，有什么作用

Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架

Spring是一个轻量级框架，实现了IOC容器和非侵入式框架。提供了AOP的实现方式， 把应用业务逻辑和系统服务分开。面向Bean编程。

spring 的优点？

1.降低了组件之间的耦合性 ，实现了软件各层之间的解耦

2.可以使用容易提供的众多服务，如事务管理，消息服务等

3.容器提供单例模式支持

4.容器提供了AOP技术，利用它很容易实现如权限拦截，运行期监控等功能

5.容器提供了众多的辅助类，能加快应用的开发

6.spring对于主流的应用框架提供了集成支持，如hibernate，JPA，Struts等

7.spring属于低侵入式设计，代码的污染极低

8.独立于各种应用服务器

9.spring的DI机制降低了业务对象替换的复杂性

10.Spring的高度开放性，并不强制应用完全依赖于Spring，开发者可以自由选择spring的部分或全部

## Spring的几个注解

(@Component，@Service，@Controller，@Repository)

传统的Spring做法是使用.xml文件来对bean进行注入或者是配置aop、事物，这么做有两个缺点：

1、如果所有的内容都配置在.xml文件中，那么.xml文件将会十分庞大；如果按需求分开.xml文件，那么.xml文件又会非常多。总之这将导致配置文件的可读性与可维护性变得很低。

2、在开发中在.java文件和.xml文件之间不断切换，是一件麻烦的事，同时这种思维上的不连贯也会降低开发的效率。

为了解决这两个问题，Spring引入了注解，通过"@XXX"的方式，让注解与Java Bean紧密结合，既大大减少了配置文件的体积，又增加了Java Bean的可读性与内聚性。

* @Autowired

@Autowired顾名思义，就是自动装配，其作用是为了消除代码Java代码里面的getter/setter与bean属性中的property。当然，getter看个人需求，如果私有属性需要对外提供的话，应当予以保留。

@Autowired默认按类型匹配的方式，在容器查找匹配的Bean，当有且仅有一个匹配的Bean时，Spring将其注入@Autowired标注的变量中。

* Qualifier（指定注入Bean的名称）

如果容器中有一个以上匹配的Bean，则可以通过@Qualifier注解限定Bean的名称

@Service用于标注业务层组件、

@Controller用于标注控制层组件（如struts中的action）

@Repository用于标注数据访问组件，即DAO组件。

@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Autowired 默认按类型装配，如果我们想使用按名称装配，可以结合@Qualifier注解一起使用。如下：

@Autowired @Qualifier("personDaoBean") 存在多个实例配合使用

@Resource默认按名称装配，当找不到与名称匹配的bean才会按类型装配。

## Bean的几种（依赖）注入方式

在xml文件中配置依赖注入

* 属性注入

属性注入即通过setXxx()方法注入Bean的属性值或依赖对象，由于属性注入方式具有可选择性和灵活性高的优点，因此属性注入是实际应用中最常采用的注入方式

* 构造方法注入

使用构造函数注入的前提是Bean必须提供带参数的构造函数

* 工厂方法注入

非静态工厂方法：有些工厂方法是非静态的，即必须实例化工厂类后才能调用。

静态工厂方法：很多工厂类都是静态的，这意味着用户在无须创建工厂类实例的情况下就可以调用工厂类方法，因此，静态工厂方法比非静态工厂方法的调用更加方便。

使用注解的方式注入：

* 使用@Autowired进行自动注入，Spring通过@Autowired注解实现Bean的依赖注入

## Bean的加载过程，时机，singleton和prototype的不同

1.

获取配置文件、解析配置文件并注册bean、实例化Bean、调用对象的方法

2.

spring管理的bean在默认情况下是会在服务器启动的时候初始化的。

bean设置了scope为prototype（原型）之后，会每次使用时生产一个

bean设置了lazy-init=”true”后，启动服务器不会马上实例化，而是在用到的时候被实例化。

3.

singleton：单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例；注意这里的Spring singleton模式是默认缺省的饿汉模式，启容器时(即实例化容器时),为所有spring配置文件中定义的bean都生成一个实例。

prototype：原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例。

注意：如果不指定Bean的作用域，Spring默认使用singleton作用域。Java在创建Java实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此，prototype作用域Bean的创建、销毁代价比较大。而singleton作用域的Bean实例一旦创建成功，可以重复使用。因此，除非必要，否则尽量避免将Bean被设置成prototype作用域。

## ApplicationContext是啥，作用

<https://www.cnblogs.com/adealjason/p/6500916.html>

ApplicationContext 应用上下文

spring中的BeanFactory与ApplicationContext的作用和区别？

1. BeanFactory负责读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的生命周期。

2. ApplicationContext除了提供上述BeanFactory所能提供的功能之外，还提供了更完整的框架功能：

a. 国际化支持

b. 资源访问：Resource rs = ctx. getResource(“classpath:config.properties”), “file:c:/config.properties”

c. 事件传递：通过实现ApplicationContextAware接口

3. 常用的获取ApplicationContext的方法：

FileSystemXmlApplicationContext：从文件系统或者url指定的xml配置文件创建，参数为配置文件名或文件名数组

ClassPathXmlApplicationContext：从classpath的xml配置文件创建，可以从jar包中读取配置文件

WebApplicationContextUtils：从web应用的根目录读取配置文件，需要先在web.xml中配置，可以配置监听器或者servlet来实现

## IOC的原理

许多应用都是通过彼此间的相互合作来实现业务逻辑的，如类A要调用类B的方法，以前我们都是在类A中，通过自身new一个类B，然后在调用类B的方法，现在我们把new类B的事情交给spring来做，在我们调用的时候，容器会为我们实例化。

* IOC容器的初始化过程

定位并获取资源文件

解析资源文件

注册

* 三种注入方式，通常用setter方法注入

setter方法注入

构造方法注入

接口注入

* bean的创建，工厂模式
* 数据注入，通过反射

## AOP(Aspect Oriented Programming，面向切面编程)

## spring AOP动态代理机制

* jdk动态代理

1、通过实现InvocationHandlet接口创建自己的调用处理器

2、通过为Proxy类指定ClassLoader对象和一组interface来创建动态代理

3、通过反射机制获取动态代理类的构造函数，其唯一参数类型就是调用处理器接口类型

4、通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数参入

JDK动态代理是面向接口的代理模式，如果被代理目标没有接口那么Spring也无能为力，

Spring通过java的反射机制生产被代理接口的新的匿名实现类，重写了其中AOP的增强方法。

* CGLib动态代理

CGLib是一个强大、高性能的Code生产类库，可以实现运行期动态扩展java类，Spring在运行期间通过

CGlib继承要被动态代理的类，重写父类的方法，实现AOP面向切面编程呢。

* 两者对比

JDK动态代理是面向接口，在创建代理实现类时比CGLib要快，创建代理速度快。

CGLib动态代理是通过字节码底层继承要代理类来实现（如果被代理类被final关键字所修饰，那么抱歉会失败），在创建代理这一块没有JDK动态代理快，但是运行速度比JDK动态代理要快。

使用注意：

如果要被代理的对象是个实现类，那么Spring会使用JDK动态代理来完成操作（Spirng默认采用JDK动态代理实现机制）

如果要被代理的对象不是个实现类那么，Spring会强制使用CGLib来实现动态代理。

那么如何选择的使用代理机制了？

通过配置Spring的中<aop:config>标签来显示的指定使用动态代理机制 proxy-target-class=true表示使用CGLib代理，如果为**false就是默认使用JDK动态代理**

## spring用到了哪些设计模式

1、工厂模式，在各种BeanFactory以及ApplicationContext创建中都用到了

2、模版模式，在各种BeanFactory以及ApplicationContext实现中也都用到了

3、代理模式，Spring AOP 利用了 AspectJ AOP实现的! AspectJ AOP 的底层用了动态代理

动态代理有两种

目标方法有接口时候自动选用 JDK 动态代理

目标方法没有接口时候选择 CGLib 动态代理

4、策略模式

加载资源文件的方式，使用了不同的方法，比如：ClassPathResourece，FileSystemResource，ServletContextResource，UrlResource但他们都有共同的借口Resource；在Aop的实现中，采用了两种不同的方式，JDK动态代理和CGLIB代理

5、单例模式，比如在创建bean的时候。

6、观察者（Observer）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

## Beanfactory 和 FactoryBean的区别

BeanFactory

BeanFactory定义了IOC容器的最基本形式，并提供了IOC容器应遵守的的最基本的接口，也就是Spring IOC所遵守的最底层和最基本的编程规范。在Spring代码中，BeanFactory只是个接口，并不是IOC容器的具体实现，但是Spring容器给出了很多种实现，如 DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext等，都是附加了某种功能的实现。

FactoryBean

一般情况下，Spring通过反射机制利用<bean>的class属性指定实现类实例化Bean，在某些情况下，实例化Bean过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring为此提供了一个org.springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化Bean的逻辑。FactoryBean接口对于Spring框架来说占用重要的地位，Spring自身就提供了70多个FactoryBean的实现。它们隐藏了实例化一些复杂Bean的细节，给上层应用带来了便利

3、区别

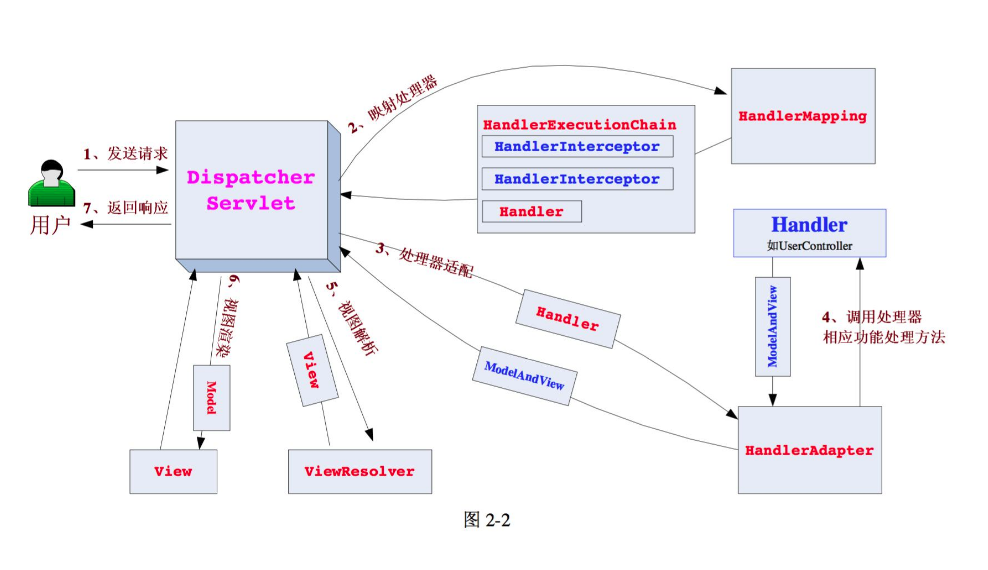
BeanFactory是个Factory，也就是IOC容器或对象工厂，FactoryBean是个Bean。在Spring中，所有的Bean都是由BeanFactory(也就是IOC容器)来进行管理的。但对FactoryBean而言，这个Bean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean,它的实现与设计模式中的工厂模式和修饰器模式类似。

## spring和mybatis整合

## 配置c3p0连接池

# SpringMVC

## SpringMVC请求处理流程



1. 首先用户发送请求——>DispatcherServlet，前端控制器接收到请求后自己不进行处理，而是委托给其它的解析器进行处理，作为统一访问点，进行全局流程控制。
2. DispatcherServlet——>HandlerMapping，HandlerMapping将会把请求映射为HandlerExecutionChain对象（包含一个Handller处理器（页面控制器对象）、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略
3. DispatcherServlet——>HandlerAdapter，HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用
4. HanderAdapter——>处理器功能处理方法的调用，HanderAdapter将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理，并返回一个ModelAndView对象（包含模型数据、逻辑视图名）
5. ModelAndView的逻辑视图名——>ViewResover，ViewResover将把逻辑视图名解析为具体的View，通过这种策略很容易更换其它视图技术
6. View——>渲染，View会根据传进来的Model数据进行渲染，此处的Model实际上是一个Map数据结构，因此很容易支持其它视图技术
7. 返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束

## SpringMVC有哪些部分，哪些需要开发者写，哪些springmvc自带

1. 前端控制器（DispatcherServlet）

中央处理器，负责对各部分任务发放

1. 处理器映射器（HandlerMaooing）

接收到前端控制器发送的url后，对url进行匹配，然后返回给前端控制器一个处理器执行链（HanderExecuteChain）

1. 处理器适配器（HanderAdapter）

接收前端控制器要寻找的Hander，选择对应的Hander进行执行，执行后返回ModelAndView给前端控制器

1. 处理器（Hander）（需要程序员编写）

执行处理器适配器发送的任务，返回ModelAndView给处理器适配器

1. 视图解析器（ViewResolver）

接收前端控制器发送来的逻辑视图（ModelAndView），转换为真正的视图，返回给前端控制器工期渲染

1. 视图（View）（需要程序员编写）

经过视图解析器生成的视图（如JSP、Excel、PDF等），经前端控制器渲染后发送给请求方

## web.xml几个部分(listener,filter,servlet)的加载时机、作用

初始化过程

1、在启动web项目时，容器（比如Tomcat）会读取web.xml配置文件中的两个节点<listener>和<context-param>

2、接着容器会创建一个ServletContext（上下文），应用范围内即整个WEB项目都能使用这个上下文

3、接着容器将会读取到<context-param>转化为键值对，并交给ServletContext。

4、容器创建<listener></listener>中的实例，即创建监听（备注：listener定义的类可以是自动以的类，但必须要继承ServletContextListtener）。

5、在监听的类中会有一个contextInitialized(ServletContextEvent event)初始化方法，在这个方法中可以通过event.getServletContext().getInitParameter("contextConfigLocation") 来得到context-param 设定的值。在这个类中还必须有一个contextDestroyed(ServletContextEvent event) 销毁方法.用于关闭应用前释放资源，比如说数据库连接的关闭。

6、得到这个context-param的值之后，你就可以做一些操作了。注意这个时候你的WEB项目还没有完全启动完成，这个动作会比所有Servlet都要早。

web.xml的加载过程是 context-param >> listener >> filter >> servlet

## springMVC-mvc.xml 配置文件片段讲解

<!-- 自动扫描的包名 -->

<context:component-scan base-package="com.app,com.core,JUnit4" ></context:component-scan>

<!-- 默认的注解映射的支持 -->

<mvc:annotation-driven />

<!-- 视图解释类 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/><!--可为空,方便实现自已的依据扩展名来选择视图解释类的逻辑 -->

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />

</bean>

<!-- 拦截器 -->

<mvc:interceptors>

<bean class="com.core.mvc.MyInteceptor" />

</mvc:interceptors>

<!-- 对静态资源文件的访问 方案一 （二选一） -->

<mvc:default-servlet-handler/>

<!-- 对静态资源文件的访问 方案二 （二选一）-->

<mvc:resources mapping="/images/\*\*" location="/images/" cache-period="31556926"/>

<mvc:resources mapping="/js/\*\*" location="/js/" cache-period="31556926"/>

<mvc:resources mapping="/css/\*\*" location="/css/" cache-period="31556926"/>

## @Controller和@RestController的区别

官方文档中这样解释：

@RestController is a stereotype annotation that combines @ResponseBody and @Controller.

意思是说@RestController=@ResponseBody+@Controller.

使用时的区别就是：

1.@Controller注解是将类标记为一个SpringMVC Controller，它是可以返回一个视图的，而@RestController不可以返回视图。

2.@RestController可以返回JSON/XML或者其他自定义类型到页面，而@Controller需要配合@ResponseBody才可以。

以下二者等价

|  |  |
| --- | --- |
| @RestController  public class HelloController {  @RequestMapping("sayHi")  public String sayHi(){  return "Hello Spring Boot";  }  } | @Controller  public class HelloController {  @RequestMapping("sayHi")  public @ResponseBody String sayHi(){  return "Hello Spring Boot";  }  } |

# Mybatis

## SqlSession,SqlSessionFactory的作用，配置

* SqlSessionFactory

SqlSessionFactory是MyBatis的关键对象，它是单个数据库映射关系经过编译后的内存镜像。SqlSessionFactory对象的实例可以通过SqlSessionFactoryBuilder对象类获得，而SqlSessionFactoryBuilder则可以通过XML配置文件或一个预先定制的Configuration的实例构建出来SqlSessionFactory的实例。每一个MyBatis的应用程序都以一个SqlSessionFactory对象的实例为核心。同时SqlSessionFactory也是线程安全的，SqlSessionFactory一旦被创建，应该在应用执行期间都存在。在应用运行前进不要重复创建多次，建议使用单例模式。SqlSessionFactory是创建SqlSession的工程。

* SqlSession

SqlSession是MyBatis的关键对象，是执行持久化操作的独享，类似于JDBC中的connection，它是应用程序与持久层之间进行交互的一个单线程对象，也是MyBatis执行持久化操作的关键对象。SqlSession对象完全包含以数据库为北京的所有执行SQL操作的方法，它的底层封装了JDBC连接，也可以用SqlSession实例来执行被映射的SQL语句。每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能被共享，同时SqlSession也是线程不安全的，绝对不能将一个SqlSession实例的引用放在一个类的静态字段甚至是实例字段中。也绝对不能将SqlSession实例的引用放在任何类型的管理范围中，比如Servlet当中的HttpSession对象中。使用完SqlSession之后要关闭session，应使用finally块来关闭它。

* SqlSessionFactory和SqlSession实现过程

mybatis框架主要是围绕着SqlSessionFactory进行的，创建过程大概如下：

(1)、定义一个Configuration对象，其中包含数据源、事务、mapper文件资源以及影响数据库行为属性设置settings

(2)、通过配置对象，则可以创建一个SqlSessionFactoryBuilder对象

(3)、通过 SqlSessionFactoryBuilder 获得SqlSessionFactory 的实例。

(4)、SqlSessionFactory 的实例可以获得操作数据的SqlSession实例，通过这个实例对数据库进行操作

mybatis-config.xml单独的配置文件

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 加载类路径下的属性文件 -->

<properties resource="db.properties"/>

<!-- 设置类型别名 -->

<typeAliases>

<typeAlias type="cn.itcast.javaee.mybatis.app04.Student" alias="student"/>

</typeAliases>

<!-- 设置一个默认的连接环境信息 -->

<environments default="mysql\_developer">

<!-- 连接环境信息，取一个任意唯一的名字 -->

<environment id="mysql\_developer">

<!-- mybatis使用jdbc事务管理方式 -->

<transactionManager type="jdbc"/>

<!-- mybatis使用连接池方式来获取连接 -->

<dataSource type="pooled">

<!-- 配置与数据库交互的4个必要属性 -->

<property name="driver" value="${mysql.driver}"/>

<property name="url" value="${mysql.url}"/>

<property name="username" value="${mysql.username}"/>

<property name="password" value="${mysql.password}"/>

</dataSource>

</environment>

</environments>

<!-- 加载映射文件-->

<mappers>

<mapper resource="cn/itcast/javaee/mybatis/app14/StudentMapper.xml"/>

</mappers>

</configuration>

## 如何结合Spring达到自动生成SqlSession的效果

在 MyBatis 中,你可以使用 SqlSessionFactory 来创建 SqlSession。一旦你获得一个 session 之后,你可以使用它来执行映射语句,提交或回滚连接,最后,当不再需要它的时候, 你可以关闭 session。 使用 MyBatis-Spring 之后, 你不再需要直接使用 SqlSessionFactory 了,因为你的 bean 可以通过一个线程安全的 SqlSession 来注入,基于 Spring 的事务配置 来自动提交,回滚,关闭 session。

SqlSessionTemplate 是 MyBatis-Spring 的核心。 这个类负责管理 MyBatis 的 SqlSession, 调用 MyBatis 的 SQL 方法,。 SqlSessionTemplate 是线程安全的, 可以被多个 DAO 所共享使用。

当调用 SQL 方法时, 包含从映射器 getMapper()方法返回的方法, SqlSessionTemplate 将会保证使用的 SqlSession 是和当前 Spring 的事务相关的。此外,它管理 session 的生命 周期,包含必要的关闭,提交或回滚操作。

SqlSessionTemplate 实现了 SqlSession 接口,这就是说,在代码中无需对 MyBatis 的 SqlSession 进行替换。 SqlSessionTemplate 通常是被用来替代默认的 MyBatis 实现的 DefaultSqlSession , 因为模板可以参与到 Spring 的事务中并且被多个注入的映射器类所使 用时也是线程安全的。相同应用程序中两个类之间的转换可能会引起数据一致性的问题。

# 设计模式

## 六大原则

* 单一职责原则

一个类只能负责一个功能领域中的响应职责，就一个类而言，应该只有一个引起它变化的原因。单一职责原则是实现高内聚、低耦合的指导方针，它是最简单而又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关实践经验。

* 开闭原则

一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行修改，软件实体可以值一个软件模块，一个由多各类组成的局部结构或一个独立的类

* 里氏替换原则

所有引用基类（父类）的地方，必须透明地使用其子类对象。在软件中将一个基类对象替换为它的子类队像，程序将不会产生任何错误和异常，反之则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

* 依赖倒置原则

抽象不应该依赖于细节，细节应当依赖于抽象，要针对接口编程而不是针对实现编程

* 接口隔离原则

使用多个专门的接口，而不是使用单一的总接口，即客户端不应该依赖那些它不需要的接口

* 迪米特法则

一个软件实体应当尽可能少的与其它实体发生相互作用

如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。

迪米特法则还有几种定义形式，包括：不要和“陌生人”说话、只与你的直接朋友通信等，在迪米特法则中，对于一个对象，其朋友包括以下几类：

(1) 当前对象本身(this)；

(2) 以参数形式传入到当前对象方法中的对象；

(3) 当前对象的成员对象；

(4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；

(5) 当前对象所创建的对象。

## 单例模式

单例模式是一种常用的软件设计模式。我们只需要某个类同时保留一个对象，不希望有更多对象，此时，我们则应考虑单例模式的设计

单例模式有以下特点：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

单例模式的实现：

* 懒汉模式（线程安全，调用效率不高，但是能延时加载）：

public class SingletonDemo2 {

//类初始化时，不初始化这个对象(延时加载，真正用的时候再创建)

private static SingletonDemo2 instance;

//构造器私有化

private SingletonDemo2(){}

//方法同步，调用效率低

public static synchronized SingletonDemo2 getInstance(){

if(instance==null){

instance=new SingletonDemo2();

}

return instance;

}

}

* 饿汉模式（线程安全，调用效率高，但是不能延时加载）

public class SingletonDemo {

private static SingletonDemo instance=new SingletonDemo();

private SingletonDemo(){

}

public static SingletonDemo getInstance(){

return instance;

}

}

* Double CheckLock实现单例：DCL也就是双重锁判断机制（由于JVM底层模型原因，偶尔会出问题，不建议使用）

public class SingletonDemo5 {

private volatile static SingletonDemo5 SingletonDemo5;

private SingletonDemo5() {

}

public static SingletonDemo5 newInstance() {

if (SingletonDemo5 == null) {

synchronized (SingletonDemo5.class) {

if (SingletonDemo5 == null) {

SingletonDemo5 = new SingletonDemo5();

}

}

}

return SingletonDemo5;

}

}

* 静态内部类实现模式（线程安全，调用效率高，可以延时加载）

public class SingletonDemo3 {

private static class SingletonClassInstance{

private static final SingletonDemo3 instance=new SingletonDemo3();

}

private SingletonDemo3(){}

public static SingletonDemo3 getInstance(){

return SingletonClassInstance.instance;

}

}

* 枚举类（线程安全，调用效率高，不能延时加载，可以天然的防止反射和反序列化调用）

public enum SingletonDemo4 {

//枚举元素本身就是单例

INSTANCE;

//添加自己需要的操作

public void singletonOperation(){

}

}

如何选用：

-单例对象 占用资源少，不需要延时加载，枚举好于饿汉式

-单例对象 占用资源多，需要延时加载，静态内部类好于懒汉式

## 工厂模式

工厂模式主要是为创建对象提供了接口。

应用场景如下：

a、 在编码时不能预见需要创建哪种类的实例。

b、 系统不应依赖于产品类实例如何被创建、组合和表达的细节。

## 策略模式。

策略模式：定义了算法族，分别封装起来，让它们之间可以互相替换。此模式让算法的变化独立于使用算法的客户。

应用场景如下。

a、 一件事情，有很多方案可以实现。

b、我可以在任何时候，决定采用哪一种实现。

c.、未来可能增加更多的方案。

d、 策略模式让方案的变化不会影响到使用方案的客户。

举例业务场景如下。

系统的操作都要有日志记录，通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面。日志写到数据库与文件中是两种算法，但调用方不关心，只负责写就是。

## 观察者模式。

观察者模式又被称作发布/订阅模式，定义了对象间一对多依赖，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

应用场景如下：

a、对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。

b、对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节。

优点： 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

缺点： 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。 3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

## 迭代器模式。

定义：

迭代器模式提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。

迭代器模式的角色构成：

(1)迭代器角色（Iterator）:定义遍历元素所需要的方法，一般来说会有这么三个方法：取得下一个元素的方法next()，判断是否遍历结束的方法hasNext()），移出当前对象的方法remove(),

(2)具体迭代器角色（Concrete Iterator）：实现迭代器接口中定义的方法，完成集合的迭代。

(3)容器角色(Aggregate): 一般是一个接口，提供一个iterator()方法，例如java中的Collection接口，List接口，Set接口等

(4)具体容器角色（ConcreteAggregate）：就是抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkList，Set接口的哈希列表的实现HashSet等。

迭代器模式的优缺点：

迭代器模式的优点有：

1简化了遍历方式，对于对象集合的遍历，还是比较麻烦的，对于数组或者有序列表，我们尚可以通过游标来取得，但用户需要在对集合了解很清楚的前提下，自行遍历对象，但是对于hash表来说，用户遍历起来就比较麻烦了。而引入了迭代器方法后，用户用起来就简单的多了。

2可以提供多种遍历方式，比如说对有序列表，我们可以根据需要提供正序遍历，倒序遍历两种迭代器，用户用起来只需要得到我们实现好的迭代器，就可以方便的对集合进行遍历了。

3封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

迭代器模式的缺点：

对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐，大家可能都有感觉，像ArrayList，我们宁可愿意使用for循环和get方法来遍历集合。

总的来说： 迭代器模式是与集合共生共死的，一般来说，我们只要实现一个集合，就需要同时提供这个集合的迭代器，就像java中的Collection，List、Set、Map等，这些集合都有自己的迭代器。假如我们要实现一个这样的新的容器，当然也需要引入迭代器模式，给我们的容器实现一个迭代器。

应用场景如下：

当你需要访问一个聚集对象，而且不管这些对象是什么都需要遍 历的时候，就应该考虑用迭代器模式。其实stl容器就是很好的迭代器模式的例子。

## 模板方法模式。

模板方法模式定义一个操作中的算法的骨架，将一些步骤延迟到子类中，模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些步骤。

优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

应用场景如下：

对于一些功能，在不同的对象身上展示不同的作用，但是功能的框架是一样的。

注意事项：为防止恶意操作，一般模板方法都加上 final 关键词。

## 生产者消费者模式

生产者消费者问题是线程模型中的经典问题：生产者和消费者在同一时间段内共用同一存储空间，生产者向空间里生产数据，而消费者取走数据。

问题的核心是：

1.要保证不让生产者在缓存还是满的时候仍然要向内写数据;

2.不让消费者试图从空的缓存中取出数据。

# Java设计模式的常见应用场景

<https://blog.csdn.net/xiaokang123456kao/article/details/76268527>

* Java I/O中的设计模式

适配器模式和装饰模式；

适配模式是为了处理两个借口不一致，改变现有接口使其匹配。

装饰模式是在不改变现有接口的前提下，加入新的功能

* Javac中的访问者模式
* Tomcat中的设计模式

门面模式、观察者模式、命令设计模式、责任链模式

* Spring中的设计模式

简单工厂模式、工厂方法模式、单例模式、代理模式、模板方法模式、策略模式

* springMVC中的模板模式

# JVM Java虚拟机

## 内存溢出和内存泄漏的区别

内存溢出是指程序在申请内存时，没有足够的内存空间供其使用，出现out of

memory。

内存泄漏是指分配出去的内存不再使用，但是无法回收。

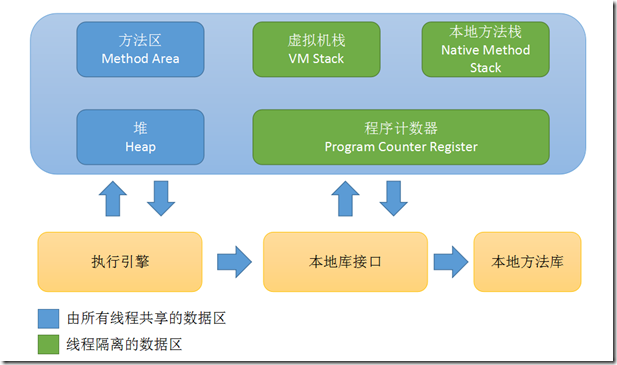
## 45.出现OOM如何解决

一. 可通过命令定期抓取heap dump或者启动参数OOM时自动抓取heap dump文件。

二. 通过对比多个heap dump，以及heap dump的内容，分析代码找出内存占用最多的地方。

三. 分析占用的内存对象，是否是因为错误导致的内存未及时释放，或者数据过多导致的内存溢出。

## jvm内存区域，分别放了啥



Java运行时数据区域

程序计数器、Java虚拟机栈、本地方法栈、Java堆、方法区、运行时常量池、直接内存

* 程序技术器：

程序计数器（Program Counter Register） 是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条执行字节码指令。

每条线程都有一个独立的程序计数器。

如果执行的是java方法，这个计数器记录的是正在执行的虚拟机字节码指令地址。如果是native方法，计数器为空。此内存区域是唯一一个在java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

* Java虚拟机栈

同样是线程私有，描述Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。一个方法对应一个栈帧。

局部变量表存放了各种基本类型、对象引用和returnAddress类型（指向了一条字节码指令地址）。其中64位长度long 和 double占两个局部变量空间，其他只占一个。

规定的异常情况有两种：1.线程请求的栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；2.如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就抛出OutOfMemoryError异常。

* 本地方法栈

和Java虚拟机栈很类似，不同的是本地方法栈为Native方法服务。

* Java堆

是Java虚拟机所管理的内存中最大的一块。由所有线程共享，在虚拟机启动时创建。堆区唯一目的就是存放对象实例。

堆中可细分为新生代和老年代，再细分可分为Eden空间、From Survivor空间、To Survivor空间。

堆无法扩展时，抛出OutOfMemoryError异常

* 方法区

所有线程共享，存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

当方法区无法满足内存分配需求时，抛出OutOfMemoryError

* 运行时常量池

它是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项是常量池（Const Pool Table），用于存放编译期生成的各种字面量和符号引用。并非预置入Class文件中常量池的内容才进入方法运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。

当方法区无法满足内存分配需求时，抛出OutOfMemoryError

* 直接内存

并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。

JDK1.4加入了NIO，引入一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。因为避免了在Java堆和Native堆中来回复制数据，提高了性能。

当各个内存区域总和大于物理内存限制，抛出OutOfMemoryError异常。

## 新生代和老年代GC

Java堆可以分为新生代和老年代两个区，其中新生代又可以分为一个Eden区和两个Survivor区，两个Survivor区分别被命名为From和To以示区分，新生代和老年代的比例为1：2，它们共同组成堆的内存区，所以新生代占堆的1/3，老年代占2/3，但这个比例可以修改，下面分别来介绍一下新生代和老年代。

* 新生代：

新生代分为三个区域，一个Eden区和两个Survivor区，它们之间的比例为（8：1：1），这个比例也是可以修改的。通常情况下，对象主要分配在新生代的Eden区上，少数情况下也可能会直接分配在老年代中。Java虚拟机每次使用新生代中的Eden和其中一块Survivor（From），在经过一次Minor GC后，将Eden和Survivor中还存活的对象一次性地复制到另一块Survivor空间上（这里使用的复制算法进行GC），最后清理掉Eden和刚才用过的Survivor（From）空间。将此时在Survivor空间存活下来的对象的年龄设置为1，以后这些对象每在Survivor区熬过一次GC，它们的年龄就加1，当对象年龄达到某个年龄（默认值为15）时，就会把它们移到老年代中。

在新生代中进行GC时，有可能遇到另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象，这些对象将直接通过分配担保机制进入老年代；

总结：

1、Minor GC是发生在新生代中的垃圾收集，采用的复制算法；

2、新生代中每次使用的空间不超过90%，主要用来存放新生的对象；

3、Minor GC每次收集后Eden区和一块Survivor区都被清空；

* 老年代：

老年代里面存放都是生命周期长的对象，对于一些较大的对象（即需要分配一块较大的连续内存空间），是直接存入老年代的，还有很多从新生代的Survivor区域中熬过来的对象。

老年代中使用的是Full GC，Full GC所采用的是标记-清除算法。老年代中的Full GC不像Minor GC操作那么频繁，并且进行一次Full GC所需要的时间要比Minor GC的时间长。

总结：

1. 老年代中使用Full GC，采用的标记-清除算法

Minor GC ，Full GC 触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

## 哪些对象会被回收、什么时候回收

垃圾收集(Garbage Collection) GC

1、什么时候回收？

GC经常发生的区域是堆区，堆区还可以细分为新生代、老年代，新生代还分为一个Eden区和两个Survivor区。

1.1 对象优先在Eden中分配，当Eden中没有足够空间时，虚拟机将发生一次Minor GC，因为Java大多数对象都是朝生夕灭，所以Minor GC非常频繁，而且速度也很快；

1.2 Full GC，发生在老年代的GC，当老年代没有足够的空间时即发生Full GC，发生Full GC一般都会有一次Minor GC。大对象直接进入老年代，如很长的字符串数组，虚拟机提供一个-XX:PretenureSizeThreadhold参数，令大于这个参数值的对象直接在老年代中分配，避免在Eden区和两个Survivor区发生大量的内存拷贝；

1.3 发生Minor GC时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则进行一次Full GC，如果小于，则查看HandlePromotionFailure设置是否允许担保失败，如果允许，那只会进行一次Minor GC，如果不允许，则改为进行一次Full GC。

2、哪些内存需要回收

jvm对不可用的对象进行回收，哪些对象是可用的，哪些是不可用的？Java并不是采用引用计数算法来判定对象是否可用，而是采用根搜索算法(GC Root Tracing)，当一个对象到GC Roots没有任何引用相连接，用图论的来说就是从GC Roots到这个对象不可达，则证明此对象是不可用的，说明此对象可以被GC。对于这些不可达对象，也不是一下子就被GC，而是至少要经历两次标记过程：如果对象在进行根搜索算法后发现没有与GC Roots相连接的引用链，那它将会第一次标记并且进行一次筛选，筛选条件是此对象有没有必要执行finalize()方法，当对象没有覆盖finalize()方法或者finalize()方法已经被虚拟机调用执行过一次，这两种情况都被视为没有必要执行finalize()方法，对于没有必要执行finalize()方法的将会被GC，对于有必要有必要执行的，对象在finalize()方法中可能会自救，也就是重新与引用链上的任何一个对象建立关联即可。

3、如何回收

选择不同的垃圾收集器，所使用的收集算法也不同。

在新生代中，每次垃圾收集都发现有大批对象死去，只有少量存活，则使用复制算法，新生代内存被分为一个较大的Eden区和两个较小的Survivor区，每次只使用Eden区和一个Survivor区，当回收时将Eden区和Survivor还存活着的对象一次性的拷贝到另一个Survivor区上，最后清理掉Eden区和刚才使用过的Survivor区，Eden和Survivor的默认比例是8：1，可以使用-XX:SurvivorRatio来设置该比例。

而老年代中对象存活率高，没有额外的空间对它进行分配担保，必须使用“标记-清理”或“标记-整理”算法。

## 垃圾回收算法

* 标记-清处算法（Mark-Sweep）

最基本的垃圾回收算法，分为两个阶段:标记阶段和清除阶段。在标记阶段首先通过根节点，标记所有从根节点开始的对象，未被标记的对象就是未被引用的垃圾对象。然后在清除阶段，清除所有未标记的对象。标记清除算法会存在大量的空间碎片，因为收回的空间是不连续的，这样给大对象分配内存的时候可能会触发full gc

* 复制算法（Copying）

将现有内存空间分为两块，每次只使用其中的一块，在垃圾回收时将正在使用的内存中存活的对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。这种算法实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到原本的一半，且存活的对象增多的话，Copying算法的效率会大大降低。

* 标记整理算法（Mark-Campact）

是一种老年代回收算法，在标记-清除算法的基础上做了一些优化，首先也需要从根节点开始对所有可达对象做一次标记，但之后，它并不是简单的清理为标记的对象，而是将所有的存活对象压缩到内存的一端。之后清理边界所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

* 分代收集算法（Generational Collection）

分代收集算法是大部分JVM所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将GC划分为老年代和新生代。老年代的特点是每次垃圾回收时只有少量的对象需要被回收，新生代的特点是每次垃圾回收时都有大量的垃圾需要被回收，因此可以根据不同的区域选择不同的算法。

目前大部分JVM的GC对新生代都采取Copy算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少。一般将新生代划分为一块较大的Eden空间和两个较小的Survivor空间（From space，To Space）,每次使用Eden空间和其中一块Survivor，当进行回收时，将该两块空间还存活的对象复制到另一块Survivor空间中。

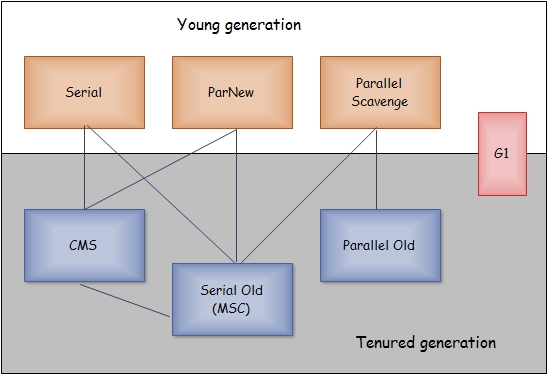
|  |  |  |  |
| --- | --- | --- | --- |
| 新生代 | Eden Space | From Space | To Space |
| Eden区 | Survivor区 | |

老生代因为每次只回收少量对象，因而采用标记整理算法（或标记清除算法）。

对象的内存分配主要在新生代的Eden Space 和 Survivor Space 的 From Space(Survivor目前存放对象的那一块)，少数情况会直接分配到老年代当新生代的Eden Space 和 From Space 的空间不足是就会发生一次GC，进行GC后，Eden Space 和 From Space 区的存活对象会被挪到To Space，然后将Eden Space 和 From Space进行清理。如果To Space无法存储某个对象，则将这个对象存储到老年代。在进行GC后，使用的便是Eden Space 和 To Space ，如此反复循环。当对象在Survivor区躲过一次GC后，其年龄就会加1.默认情况下其年龄到达15的对象就会被迁移到老年代中。

## 典型的垃圾收集器

这里讨论的收集器基于JDK1.7Update14之后的Hotspot虚拟机（在这个版本中正式提出了商用的G1收集器，之前G1仍处于实验状态），这个虚拟机包含的所有收集器如下图所示



如果两个收集器存在连线，说明他们可以搭配使用。

并行和并发的区别

并行（Parallel）：多条垃圾收集线程并行工作，但此时用户线程仍处于等待状态；

并发（Concurrent）：用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行，垃圾收集程序线程运行于；另一个CPU上。

* Serial（串行GC）收集器

Serial收集器是最基本、发展历史最悠久的收集器。是单线程的收集器，使用**复制算法**。它在进行垃圾收集时，必须暂停其它所有工作的线程，直到它收集完成。是JVM client模式下默认的新生代收集器。

* ParNew（并行GC）收集器

Serial收集器的多线程版本，除了使用多线程进行垃圾收集之外，其余行为包括Srial收集器可用的所有控制参数、收集算法（**复制算法**）、Stop The World、对象分配规回收策略等都与Serial收集器一样。

ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，其中一个与性能无关但很重要的原因是，除了Serial收集器之外，目前只有ParNew收集器能与CMS收集器配合工作

* Parallel Scavenge（并行回收）收集器/（并行清除收集器）

Parallel Scavenge收集器也是一个新生代收集器，也是使用**复制算法**的收集器，也是并行多线程收集器。它的关注点与其它收集器不同，CMS等收集器关注点是尽可能的缩短垃圾收集时用户线程的停顿时间。而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量。吞吐量是CPU用于运行用户代码的时间占CPU总消耗时间的比值，吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟器总共运行了100分钟，垃圾收集花掉1分钟，那吞吐量就是99%

* Serial Old收集器

是Serial收集器的老年代版本，同样也是一个单线程收集器，使用**标记整理算法**这个收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，主要有两大用途：

1. 在JDK1.5之前的版本中与Parallel Scavenge收集器搭配使用
2. 作为CMS收集器的后备预案，在并发手机发生Concurrent Mode Failure使用

* Parallel Old收集器

Parallel 收集器是 Parallel Scavengr收集器的老年代版本，使用多线程和**标记整理算法**。这个收集器在jdk1.6中开始提供。

* CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或B/S系统的服务端上，在，这类应用尤其中说服务的响应速度，希望系统停顿时间最短，已给用户带来更好的体验。

CMS收集器基于标记清楚算法实现的，整个手机过程大致分4步：

①初始标记(CMS initial mark)

②并发标记(CMS concurrenr mark)

③重新标记(CMS remark)

④并发清楚(CMS concurrent sweep)

其中初始标记、重新标记这两个步骤任然需要停顿其他用户线程。初始标记仅仅只是标记出GC ROOTS能直接关联到的对象，速度很快，并发标记阶段是进行GC ROOTS 根搜索算法阶段，会判定对象是否存活。而重新标记阶段则是为了修正并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间会被初始标记阶段稍长，但比并发标记阶段要短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以整体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

CMS收集器的优点：并发收集、低停顿，但是CMS还远远达不到完美，其主要有三个显著缺点：

CMS收集器对CPU资源非常敏感。在并发阶段，虽然不会导致用户线程停顿，但是会占用CPU资源而导致引用程序变慢，总吞吐量下降。CMS默认启动的回收线程数是：(CPU数量+3) / 4。

CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure“，失败后而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行，伴随程序的运行自热会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在本次收集中处理它们，只好留待下一次GC时将其清理掉。这一部分垃圾称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，

即需要预留足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分内存空间提供并发收集时的程序运作使用。在默认设置下，CMS收集器在老年代使用了68%的空间时就会被激活，也可以通过参数-XX:CMSInitiatingOccupancyFraction的值来提供触发百分比，以降低内存回收次数提高性能。要是CMS运行期间预留的内存无法满足程序其他线程需要，就会出现“Concurrent Mode Failure”失败，这时候虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置的过高将会很容易导致“Concurrent Mode Failure”失败，性能反而降低。

最后一个缺点，CMS是基于“**标记-清除”算法**实现的收集器，使用“标记-清除”算法收集后，会产生大量碎片。空间碎片太多时，将会给对象分配带来很多麻烦，比如说大对象，内存空间找不到连续的空间来分配不得不提前触发一次Full GC。为了解决这个问题，CMS收集器提供了一个-XX:UseCMSCompactAtFullCollection开关参数，用于在Full GC之后增加一个碎片整理过程，还可通过-XX:CMSFullGCBeforeCompaction参数设置执行多少次不压缩的Full GC之后，跟着来一次碎片整理过程。

* G1收集器

G1(Garbage First)收集器是JDK1.7提供的一个新收集器，G1收集器基于**“标记-整理”算法**实现，也就是说不会产生内存碎片。还有一个特点之前的收集器进行收集的范围都是整个新生代或老年代，而G1将整个Java堆(包括新生代，老年代)。

G1具备如下特点：

并行与并发：G1能更充分的利用CPU，多核环境下的硬件优势来缩短stop the world的停顿时间。

分代收集：和其他收集器一样，分代的概念在G1中依然存在，不过G1不需要其他的垃圾回收器的配合就可以独自管理整个GC堆。

空间整合：G1收集器有利于程序长时间运行，分配大对象时不会无法得到连续的空间而提前触发一次GC。

可预测的非停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

|  |  |
| --- | --- |
| Serial(串行GC)收集器 | 复制算法 |
| ParNew(并行GC)收集器 | 复制算法 |
| Parallel Scavenge(并行回收GC)收集器 | 复制算法 |
| Serial Old(串行GC)收集器 | 标记整理算法 |
| Parallel Old(并行GC)收集器 | 标记整理算法 |
| CMS(并发GC)收集器 | 标记清楚算法 |
| G1收集器 | 标记整理+复制 |

## 垃圾回收参数

|  |  |
| --- | --- |
| **参数** | **描述** |
| -XX:+UseSerialGC | Jvm运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收 |
| -XX:+UseParNewGC | 打开此开关后，使用ParNew + Serial Old的收集器进行垃圾回收 |
| -XX:+UseConcMarkSweepGC | 使用ParNew + CMS + Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。 |
| -XX:+UseParallelGC | Jvm运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge +Serial Old的收集器组合进行回收 |
| -XX:+UseParallelOldGC | 使用Parallel Scavenge +  Parallel Old的收集器组合进行回收 |
| -XX:SurvivorRatio | 新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Subrvivor = 8:1 |
| -XX:PretenureSizeThreshold | 直接晋升到老年代对象的大小，设置这个参数后，大于这个参数的对象将直接在老年代分配 |
| -XX:MaxTenuringThreshold | 晋升到老年代的对象年龄，每次Minor GC之后，年龄就加1，当超过这个参数的值时进入老年代 |
| -XX:UseAdaptiveSizePolicy | 动态调整java堆中各个区域的大小以及进入老年代的年龄 |
| -XX:+HandlePromotionFailure | 是否允许新生代收集担保，进行一次minor gc后, 另一块Survivor空间不足时，将直接会在老年代中保留 |
| -XX:ParallelGCThreads | 设置并行GC进行内存回收的线程数 |
| -XX:GCTimeRatio | GC时间占总时间的比列，默认值为99，即允许1%的GC时间，仅在使用Parallel Scavenge 收集器时有效 |
| -XX:MaxGCPauseMillis | 设置GC的最大停顿时间，在Parallel Scavenge 收集器下有效 |
| -XX:CMSInitiatingOccupancyFraction | 设置CMS收集器在老年代空间被使用多少后出发垃圾收集，默认值为68%，仅在CMS收集器时有效，-XX:CMSInitiatingOccupancyFraction=70 |
| -XX:+UseCMSCompactAtFullCollection | 由于CMS收集器会产生碎片，此参数设置在垃圾收集器后是否需要一次内存碎片整理过程，仅在CMS收集器时有效 |
| -XX:+CMSFullGCBeforeCompaction | 设置CMS收集器在进行若干次垃圾收集后再进行一次内存碎片整理过程，通常与UseCMSCompactAtFullCollection参数一起使用 |
| -XX:+UseFastAccessorMethods | 原始类型优化 |
| -XX:+DisableExplicitGC | 是否关闭手动System.gc |
| -XX:+CMSParallelRemarkEnabled | 降低标记停顿 |
| -XX:LargePageSizeInBytes | 内存页的大小不可设置过大，会影响Perm的大小，-XX:LargePageSizeInBytes=128m |

| **参数** | **描述** |
| --- | --- |
| -Xms | 最小堆大小 |
| -Xmx | 最大堆大小 |
| -Xmn | 新生代大小 |
| -XX:PermSize | 永久代大小 |
| -XX:MaxPermSize | 永久代最大大小 |
| -XX:+PrintGC | 输出GC日志 |
| -verbose:gc | - |
| -XX:+PrintGCDetails | 输出GC的详细日志 |
| -XX:+PrintGCTimeStamps | 输出GC时间戳(以基准时间的形式) |
| -XX:+PrintHeapAtGC | 在进行GC的前后打印出堆的信息 |
| -Xloggc:/path/gc.log | 日志文件的输出路径 |
| -XX:+PrintGCApplicationStoppedTime | 打印由GC产生的停顿时间 |

## JVM调优几个命令

https://www.cnblogs.com/ityouknow/p/5714703.html

* jps

JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

* jstat

JVM statistics Monitoring 是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中类装载、内存、垃圾收集、JIT编译等运行数据。

* jmap

(JVM Memory Map)：命令用于生成heap dump 命令，如果不使用这个命令，还阔以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候·自动生成dump文件。

jmap不仅能生成dump文件，还阔以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。

* jhat

jhat(JVM Heap Analysis Tool)命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看。在此要注意，一般不会直接在服务器上进行分析，因为jhat是一个耗时并且耗费硬件资源的过程，一般把服务器生成的dump文件复制到本地或其他机器上进行分析。

* jstack

jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。 如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。另外，jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。

* jinfo

jinfo(JVM Configuration info)这个命令作用是实时查看和调整虚拟机运行参数。

之前的jps -v口令只能查看到显示指定的参数，如果想要查看未被显示指定的参数的值就要使用jinfo口令

## jvm 虚拟机的server和client模式

Java HotSpot Client VM(-client)，为在客户端环境中减少启动时间而优化；

Java HotSpot Server VM(-server)，为在服务器环境中最大化程序执行速度而设计。

比较：Server VM启动比Client VM慢，运行比Client VM快。

server模式的运行中，垃圾回收处理做的比较好一些。

# 网络 Socket编程

## 网络基础知识

* 两台计算机间进行通讯需要以下三个条件：

IP地址、协议、端口号

* TCP/IP协议

是目前世界上应用最广泛的协议，是以TCP和IP为基础的不同层次上多个协议的集合，也称TCP/IP协议族、或TCP/IP协议栈

TCP：Transmission Control Protocol 传输控制协议

IP：Internet Protocol 互联网协议

* TCP/IP五层模型

应用层：HTTP、FTP、SMTP、Telnet等

HTTP：HyperText Transfer Protocol超文本传输协议

FTP：File Transfer Protocol文件传输协议

SMTP：Simple Mail Transfer Protocol简单邮件传输协议

Telnet：Internet远程登陆服务的标准协议和主要方式，TCP/IP协议族中的一员

传输层：TCP/IP

网络层：

数据链路层：

物理层：网线、双绞线、网卡等

* IP地址：

为实现网络中不同计算机之间的通信，每台计算机都必须有一个唯一的标识---IP地址。

32位

* 端口

区分一台主机的多个不同应用程序，端口号范围为0-65535，其中0-1023位为系统保留。

如：HTTP：80 FTP：21 Telnet：23

IP地址+端口号组成了所谓的Socket，Socket是网络上运行的程序之间双向通信链路的终结点，是TCP和UDP的基础

* Socket套接字：

网络上具有唯一标识的IP地址和端口组合在一起才能构成唯一能识别的标识符套接字。

Socket原理机制：

通信的两端都有Socket

网络通信其实就是Socket间的通信

数据在两个Socket间通过IO传输

* Java中的网络支持

针对网络通信的不同层次，Java提供了不同的API，其提供的网络功能有四大类：

InetAddress:用于标识网络上的硬件资源，主要是IP地址

URL：统一资源定位符，通过URL可以直接读取或写入网络上的数据

Sockets：使用TCP协议实现的网络通信Socket相关的类

Datagram:使用UDP协议，将数据保存在用户数据报中，通过网络进行通信。

## InetAddress、URL类

InetAddress类用于标识网络上的硬件资源，标识互联网协议(IP)地址。该类没有构造方法。

URL类

URI：统一资源标志符（Uniform Resource Identifier）

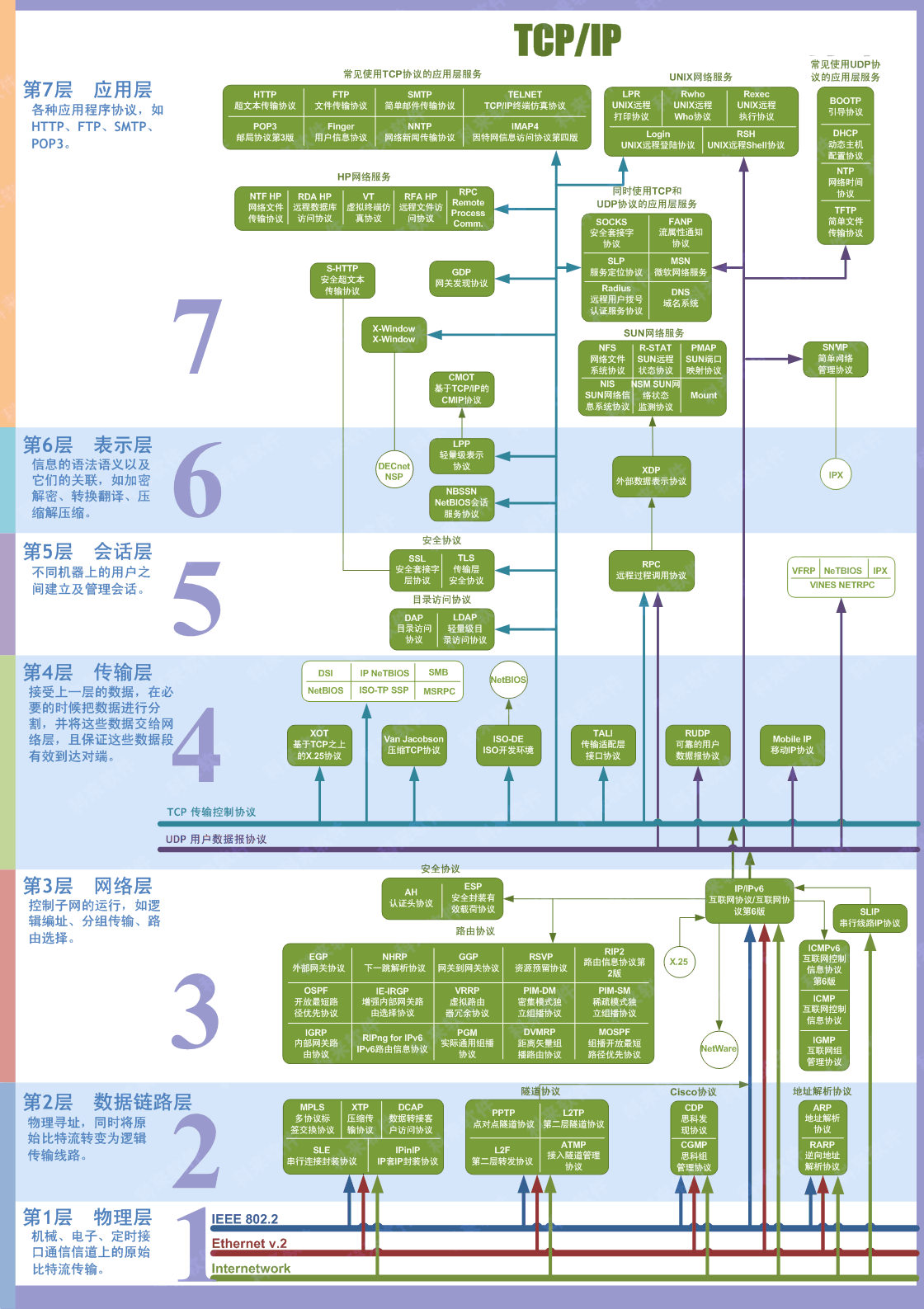
URN：统一资源名称 (Uniform Resource Name, URN)

URL和URN是URI的子集

URL(Uniform Resource Locator)统一资源定位符，表示Internet上某一资源的地址，协议名：资源名称

使用URL读取网页内容，通过URL对象的openStream()方法可以得到指定资源的输入流，通过流能够读取或访问网页上的资源

## 7层模型，每一层分别有哪些协议



七层模型，亦称OSI（Open System Interconnection）参考模型，是国际标准化组织（ISO）制定的一个用于计算机或通信系统间互联的标准体系。

OSI七层和TCP/IP四层的关系

OSI引入了服务、接口、协议、分层的概念，TCP/IP借鉴了OSI的这些概念建立TCP/IP模型

OSI现有模型后有协议，先有标准后进行实践；而TCP/IP则相反，先有协议和应用在提出了模型，且是参照的OSI模型

OSI是一种理论下的模型，TCP/IP已被广泛应用 ，称为互联网事实上的标准

TCP：transmission control protocol 传输控制协议

UDP：user data protocol 用户数据报协议

|  |  |  |
| --- | --- | --- |
| OSI七层模型 | TCP/IP四层模型 | 对应网络协议 |
| 应有层（Application） | 应用层 | HTTP FTP TFTP SMTP SNMP DNS TELNET HTTPS POP3 DHCP |
| 表示层（Presentation） | HTTP/HTML FTP Telnet ASN.1（具有表示层功能） |
| 会话层（Session） | ADSP·ASP·H.245·ISO-SP·iSNS·NetBIOS·PAP·RPC·  RTCP·SMPP·SCP·SSH·ZIP·SDP（具有会话层功能） |
| 传输层（Transport） | 传输层 | TCP · UDP · TLS · DCCP · SCTP ·RSVP · PPTP |
| 网络层（Network） | 网络层 | IP, ICMP, ARP, RARP, AKP, UUCP |
| 数据链路层（Data Link） | 数据链路层 | FDDI, Ethernet, Arpanet, PDN, SLIP, PPP |
| 物理层（Physical） | IEEE 802.1A, IEEE 802.2到IEEE 802.11 |

## 如何通过IP寻址

## DNS整个结构，过程

* 域名系统DNS（Domain Name System）
* 域名结构

每一个域名都是有标号（label）序列组成，而各标号之间用点（小数点）隔开。如下例子

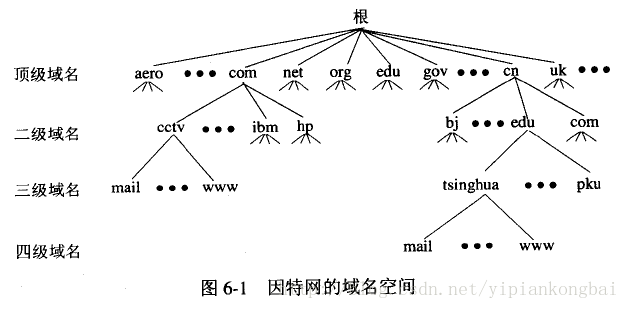


域名只是逻辑概念，并不代表计算机所在的物理地点。据2006年12月统计，现在顶级域名TLD(Top Level Domain)已有265个，分为三大类：

(1)国家顶级域名nTLD：采用ISO3166的规定。如：cn代表中国，us代表美国，uk代表英国，等等。国家域名又常记为ccTLD(cc表示国家代码contry-code)。

(2)通用顶级域名gTLD：最常见的通用顶级域名有7个，即：com(公司企业)，net(网络服务机构)，org(非营利组织)，int(国际组织)，gov(美国的政府部门)，mil(美国的军事部门)。

(3)基础结构域名(infrastructure domain)：这种顶级域名只有一个，即arpa，用于反向域名解析，因此称为反向域名。

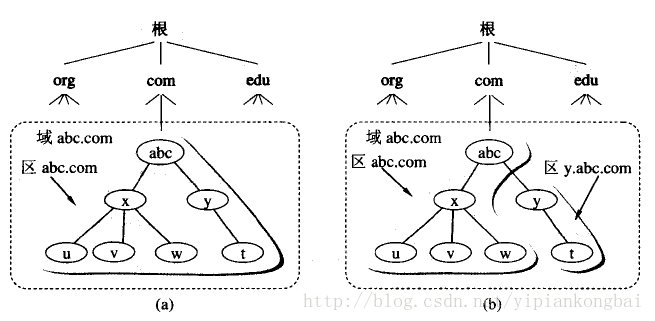


* 域名服务器

如果采用上述的树状结构，每一个节点都采用一个域名服务器，这样会使得域名服务器的数量太多，使域名服务器系统的运行效率降低。所以在DNS中，采用划分区的方法来解决。

一个服务器所负责管辖(或有权限)的范围叫做区(zone)。各单位根据具体情况来划分自己管辖范围的区。但在一个区中的所有节点必须是能够连通的。每一个区设置相应的权限域名服务器，用来保存该区中的所有主机到域名IP地址的映射。总之，DNS服务器的管辖范围不是以“域”为单位，而是以“区”为单位。区是DNS服务器实际管辖的范围。区 <= 域。

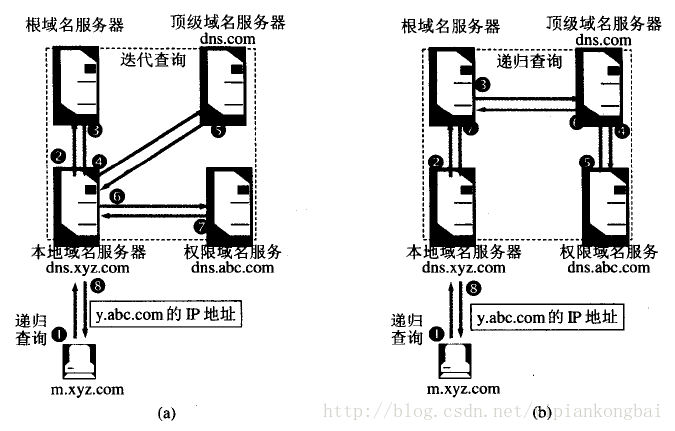
下图是区的不同划分方法的举例。假定abc公司有下属部门x和y，部门x下面有分三个分布们u,v,w，而y下面还有下属部门t。图a表示abc公司只设一个区abc.com。这是，区abc.com和域abc.com指的是同一件事。但图b表示abc公司划分为两个区：abc.com和y.abc.com。这两个区都隶属于域abc.com，都各设置了相应的权限域名服务器。不难看出，区是域的子集。



* 域名解析过程

一、主机向本地域名服务器的查询一般都是采用递归查询。所谓递归查询就是：如果主机所询问的本地域名服务器不知道被查询的域名的IP地址，那么本地域名服务器就以DNS客户的身份，向其它根域名服务器继续发出查询请求报文(即替主机继续查询)，而不是让主机自己进行下一步查询。因此，递归查询返回的查询结果或者是所要查询的IP地址，或者是报错，表示无法查询到所需的IP地址。

二、本地域名服务器向根域名服务器的查询的迭代查询。迭代查询的特点：当根域名服务器收到本地域名服务器发出的迭代查询请求报文时，要么给出所要查询的IP地址，要么告诉本地服务器：“你下一步应当向哪一个域名服务器进行查询”。然后让本地服务器进行后续的查询。根域名服务器通常是把自己知道的顶级域名服务器的IP地址告诉本地域名服务器，让本地域名服务器再向顶级域名服务器查询。顶级域名服务器在收到本地域名服务器的查询请求后，要么给出所要查询的IP地址，要么告诉本地服务器下一步应当向哪一个权限域名服务器进行查询。最后，知道了所要解析的IP地址或报错，然后把这个结果返回给发起查询的主机。



假定域名为m.xyz.com的主机想知道另一个主机y.abc.com的IP地址。例如，主机m.xyz.com打算发送邮件给y.abc.com。这时就必须知道主机y.abc.com的IP地址。下面是上图a的几个查询步骤：

1、主机m.abc.com先向本地服务器dns.xyz.com进行递归查询。

2、本地服务器采用迭代查询。它先向一个根域名服务器查询。

3、根域名服务器告诉本地服务器，下一次应查询的顶级域名服务器dns.com的IP地址。

4、本地域名服务器向顶级域名服务器dns.com进行查询。

5、顶级域名服务器dns.com告诉本地域名服务器，下一步应查询的权限服务器dns.abc.com的IP地址。

6、本地域名服务器向权限域名服务器dns.abc.com进行查询。

7、权限域名服务器dns.abc.com告诉本地域名服务器，所查询的主机的IP地址。

8、本地域名服务器最后把查询结果告诉m.xyz.com。

DNS协议运行在UDP协议之上，使用端口号53。

DNS的获取流程:

DNS是应用层协议，事实上他是为其他应用层协议工作的，包括不限于HTTP和SMTP以及FTP，用于将用户提供的主机名解析为IP地址。

具体过程如下：

①用户主机上运行着DNS的客户端，就是我们的PC机或者手机客户端运行着DNS客户端了。

②浏览器将接收到的url中抽取出域名字段，就是访问的主机名，比如http://www.baidu.com/，并将这个主机名传送给DNS应用的客户端。

③DNS客户机端向DNS服务器端发送一份查询报文，报文中包含着要访问的主机名字段（中间包括一些列缓存查询以及分布式DNS集群的工作）。

④该DNS客户机最终会收到一份回答报文，其中包含有该主机名对应的IP地址。

⑤一旦该浏览器收到来自DNS的IP地址，就可以向该IP地址定位的HTTP服务器发起TCP连接。

## TCP和UDP区别

TCP（Transmission Control Protocol 传输控制协议）

UDP 是User Datagram Protocol的简称， 中文名是用户数据报协议

TCP 与 UDP 区别总结：

* TCP面向连接（如打电话要先拨号建立连接）；UDP是无连接的，即发送数据之前不需要建立连接
* TCP提供可靠的服务。也就是说，通过TCP连接传输数据，无差错，不丢失，不重复，且按序到达；UDP尽量最大努力交付，即不保证可靠交付。TCP通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制，还可以对次序乱掉的分包进行顺序控制。
* UDP具有较好的实时性，工作效率比TCP高，适用于对告诉传输和实时性有较高的通信或广播通信
* 每一条TCP连接只能是点到点的；UDP支持一对一，一对多，多对一和多对多的交互通信
* TCP对系统资源要求较多，UDP对系统资源要求较少。TCP首部开销20字节；UDP的首部开销小，只有8字节
* TCP面向字节流，实际上是TCP吧数据看成一连串无结构的字节流；UDP是面向报文的

UDP的应用场景：

面向数据报方式

网络数据大多为短消息

拥有大量Client

对数据安全性无特殊要求

网络负担非常重，但对响应速度要求高

**具体编程时的区别**

1.socket()的参数不同

2.UDP Server不需要调用listen和accept

3.UDP收发数据用sendto/recvfrom函数

4.TCP：地址信息在connect/accept时确定

5.UDP：在sendto/recvfrom函数中每次均 需指定地址信息

6.UDP：shutdown函数无效

**TCP:**

TCP编程的服务器端一般步骤是：

1、创建一个socket，用函数socket()；

2、设置socket属性，用函数setsockopt(); \* 可选

3、绑定IP地址、端口等信息到socket上，用函数bind();

4、开启监听，用函数listen()；

5、接收客户端上来的连接，用函数accept()；

6、收发数据，用函数send()和recv()，或者read()和write();

7、关闭网络连接；

8、关闭监听；

TCP编程的客户端一般步骤是：

1、创建一个socket，用函数socket()；

2、设置socket属性，用函数setsockopt();\* 可选

3、绑定IP地址、端口等信息到socket上，用函数bind();\* 可选

4、设置要连接的对方的IP地址和端口等属性；

5、连接服务器，用函数connect()；

6、收发数据，用函数send()和recv()，或者read()和write();

7、关闭网络连接；

**UDP:**

与之对应的UDP编程步骤要简单许多，分别如下：

UDP编程的服务器端一般步骤是：

1、创建一个socket，用函数socket()；

2、设置socket属性，用函数setsockopt();\* 可选

3、绑定IP地址、端口等信息到socket上，用函数bind();

4、循环接收数据，用函数recvfrom();

5、关闭网络连接；

UDP编程的客户端一般步骤是：

1、创建一个socket，用函数socket()；

2、设置socket属性，用函数setsockopt();\* 可选

3、绑定IP地址、端口等信息到socket上，用函数bind();\* 可选

4、设置对方的IP地址和端口等属性;

5、发送数据，用函数sendto();

6、关闭网络连接；

TCP和UDP是OSI模型中的运输层中的协议。TCP提供可靠的通信传输，而UDP则常被用于让广播和细节控制交给应用的通信传输。

## TCP三次握手，四次挥手过程，原因

TCP是主机对主机层的传输控制协议，提供可靠的连接服务，采用三次握手确认一个连接：

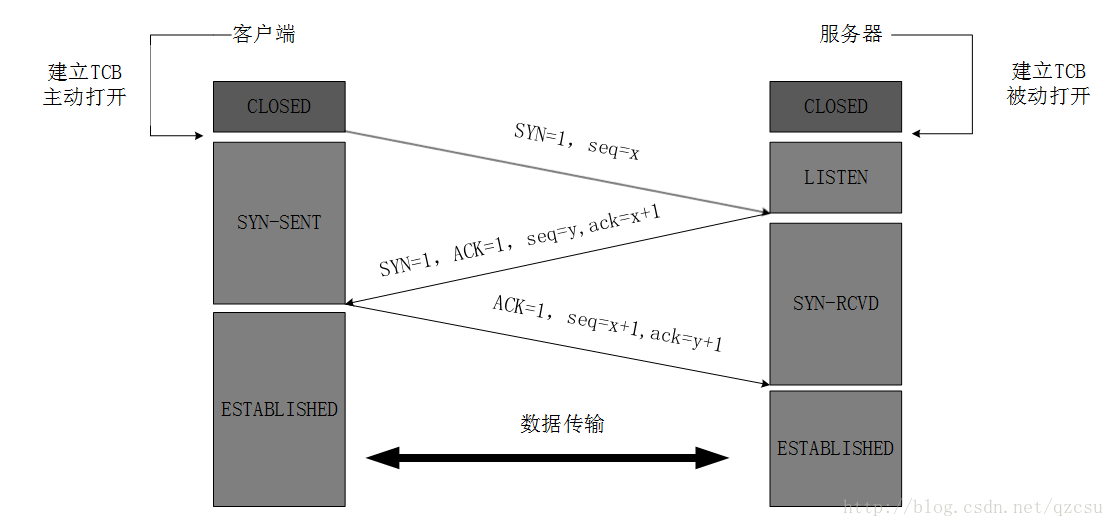
几个概念：

seq：序号，占4个字节，由于TCP是面向字节流的，在一个TCP连接中传送字节流的每一个字节都按照顺序编号，此外序号是循环使用的。

ACK：仅当ACK=1时确认字段才有效，当ACK=0时确认字段无效，并且TCP规定，在连接建立后所有的传送豹纹段都必须要把ACK设置为1

SYN：同步序列号，用来发起一个连接。当SYN=1而ACK-0时表明这是一个请求报文段；若对方同意连接，则相应报文中SYN=1，ACK=1

FIN：用来释放一个连接，当FIN=1表示次报文段的发送方已经发送完毕，并要求释放连接



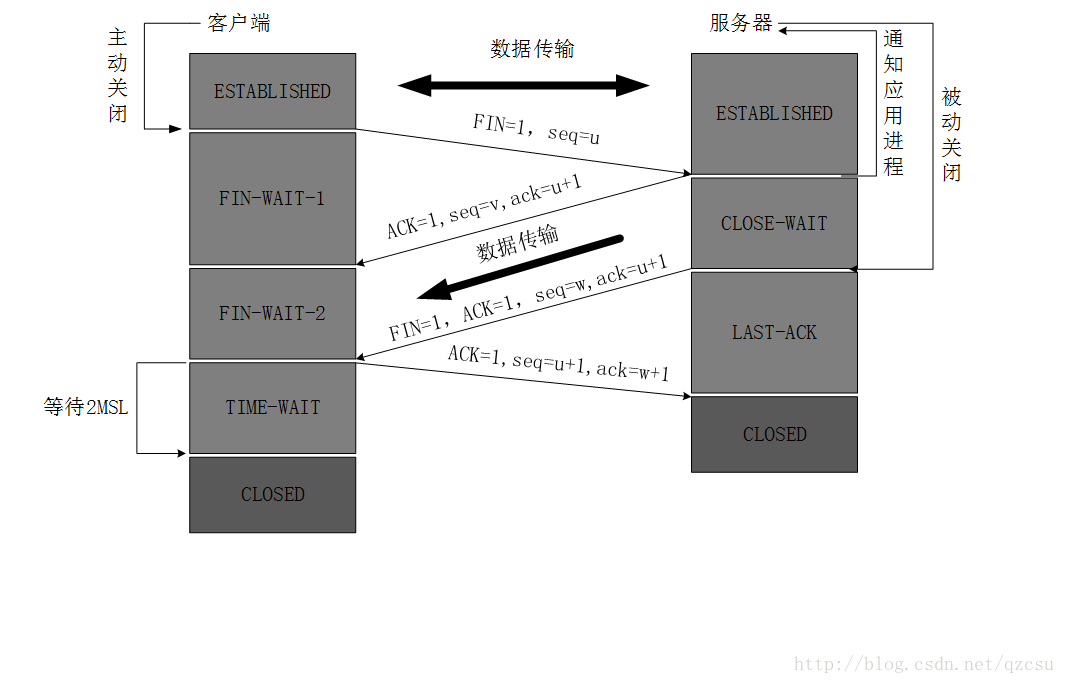
1、客户端的TCP进程也首先创建传输控制模块TCB，然后向服务端发出连接请求报文段，该报文段首部中的SYN=1，ACK=0，同时选择一个初始序号 seq=i。TCP规定，SYN=1的报文段不能携带数据，但要消耗掉一个序号。这时，TCP客户进程进入SYN—SENT（同步已发送）状态，这是 TCP连接的第一次握手。

2、服务端收到客户端发来的请求报文后，如果同意建立连接，则向客户端发送确认。确认报文中的SYN=1，ACK=1，确认号ack=i+1，同时为自己 选择一个初始序号seq=j。同样该报文段也是SYN=1的报文段，不能携带数据，但同样要消耗掉一个序号。这时，TCP服务端进入SYN—RCVD（同 步收到）状态，这是TCP连接的第二次握手。

3、TCP客户端进程收到服务端进程的确认后，还要向服务端给出确认。确认报文段的ACK=1，确认号ack=j+1，而自己的序号为seq=i+1。 TCP的标准规定，ACK报文段可以携带数据，但如果不携带数据则不消耗序号，因此，如果不携带数据，则下一个报文段的序号仍为seq=i+1。这 时，TCP连接已经建立，客户端进入ESTABLISHED（已建立连接）状态。这是TCP连接的第三次握手，可以看出第三次握手客户端已经可以发送携带 数据的报文段了。

当服务端收到确认后，也进入ESTABLISHED（已建立连接）状态。

四次挥手：



1客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为seq=u（等于前面已经传送过来的数据的最后一个字节的序号加1），此时，客户端进入FIN-WAIT-1（终止等待1）状态。 TCP规定，FIN报文段即使不携带数据，也要消耗一个序号。

2服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号seq=v，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

3客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。

5客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

6服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可以接受数据。

第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）。

第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了。

第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手。

TCP的四次挥手过程（简言之）：主动关闭方向被动关闭方发送不会再给你发数据了的信息；被动关闭方对收到的主动关闭方的报文段进行确认；被动关闭方向主动关闭方发送我也不会再给你发数据了的信息；主动关闭方再次对被动关闭方的确认进行确认。

TCP的三次握手过程？为什么会采用三次握手，若采用二次握手可以吗？

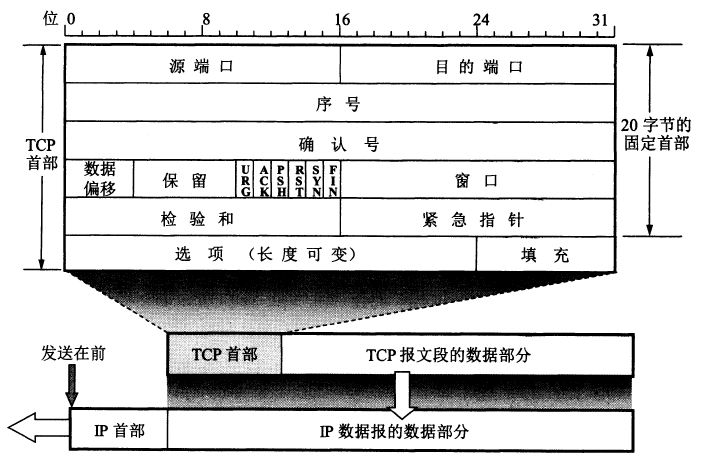
答：建立连接的过程是利用客户服务器模式，假设主机A为客户端，主机B为服务器端。

（1）TCP的三次握手过程：主机A向B发送连接请求；主机B对收到的主机A的报文段进行确认；主机A再次对主机B的确认进行确认。

（2）采用三次握手是为了防止失效的连接请求报文段突然又传送到主机B，因而产生错误。失效的连接请求报文段是指：主机A发出的连接请求没有收到主机B的确认，于是经过一段时间后，主机A又重新向主机B发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种特殊情况，主机A第一次发送的连接请求并没有丢失，而是因为网络节点导致延迟达到主机B，主机B以为是主机A又发起的新连接，于是主机B同意连接，并向主机A发回确认，但是此时主机A根本不会理会，主机B就一直在等待主机A发送数据，导致主机B的资源浪费。

（3）采用两次握手不行，原因就是上面说的失效的连接请求的特殊情况，因此采用三次握手刚刚好，两次可能出现失效，四次甚至更多次则没必要，反而复杂了。

## TCP报文结构



* 端口号：用来表示同一台计算机的不同的应用进程。

源端口号：源端口和IP地址的作用是表示报文的返回地址，占2个字节

目的端口号：指明接收方计算机上的应用程序接口，占2分字节

* 序列号：表示本报文段所发送数据的第一个字节的编号。在TCP连接中所传送的字节流的每一个字节都会按顺序编号。由于序列号由32位表示，所以每2^32个字节，就会出现序列号回绕，再次从 0 开始。那如何区分两个相同序列号的不同TCP报文段就是一个问题了，后面会有答案，暂时可以不管。
* 确认号：表示接收方期望收到发送方下一个报文段的第一个字节数据的编号。也就是告诉发送发：我希望你（指发送方）下次发送的数据的第一个字节数据的编号是这个确认号。也就是告诉发送方：我希望你（指发送方）下次发送给我的TCP报文段的序列号字段的值是这个确认号。

以下6个是控制位

* TCP首部长度/数据偏移：由于TCP首部包含一个长度可变的选项部分，所以需要这么一个值来指定这个TCP报文段到底有多长。或者可以这么理解：就是表示TCP报文段中数据部分在整个TCP报文段中的位置。该字段的单位是32位字，即：4个字节。
* 紧急URG：表示本报文段中发送的数据是否包含紧急数据。URG=1，表示有紧急数据。后面的紧急指针字段只有当URG=1时才有效。
* 确认ACK：表示是否前面的确认号字段是否有效。ACK=1，表示有效。只有当ACK=1时，前面的确认号字段才有效。TCP规定，连接建立后，ACK必须为1。
* 推送PSH：告诉对方收到该报文段后是否应该立即把数据推送给上层。如果为1，则表示对方应当立即把数据提交给上层，而不是缓存起来。
* 复位RST：只有当RST=1时才有用。如果你收到一个RST=1的报文，说明你与主机的连接出现了严重错误（如主机崩溃），必须释放连接，然后再重新建立连接。或者说明你上次发送给主机的数据有问题，主机拒绝响应。
* 同步SYN：在建立连接时使用，用来同步序号。当SYN=1，ACK=0时，表示这是一个请求建立连接的报文段；当SYN=1，ACK=1时，表示对方同意建立连接。SYN=1，说明这是一个请求建立连接或同意建立连接的报文。只有在前两次握手中SYN才置为1。
* 终止FIN：标记数据是否发送完毕。如果FIN=1，就相当于告诉对方：“我的数据已经发送完毕，你可以释放连接了”
* 窗口大小：表示现在运行对方发送的数据量。也就是告诉对方，从本报文段的确认号开始允许对方发送的数据量。
* 校验和：提供额外的可靠性。具体如何校验，参考其他资料。
* 紧急指针：标记紧急数据在数据字段中的位置。
* 选项部分：其最大长度可根据TCP首部长度进行推算。TCP首部长度用4位表示，那么选项部分最长为：(2^4-1)\*4-20=40字节。

## DDoS攻击

分布式拒绝服务(DDoS:Distributed Denial of Service)攻击指借助于客户/服务器技术，将多个计算机联合起来作为攻击平台，对一个或多个目标发动DDoS攻击，从而成倍地提高拒绝服务攻击的威力。通常，攻击者使用一个偷窃帐号将DDoS主控程序安装在一个计算机上，在一个设定的时间主控程序将与大量代理程序通讯，代理程序已经被安装在网络上的许多计算机上。代理程序收到指令时就发动攻击。利用客户/服务器技术，主控程序能在几秒钟内激活成百上千次代理程序的运行。

**攻击方式：**

DDoS攻击通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的。 这种攻击方式可分为以下几种：

通过使网络过载来干扰甚至阻断正常的网络通讯；

通过向服务器提交大量请求，使服务器超负荷；

阻断某一用户访问服务器；

阻断某服务与特定系统或个人的通讯。

⑴、端口扫描攻击：

端口扫描攻击通过对VPS主机进行系统的结构化扫描来实现。例如，某人可能扫描您的 VPS主机上搭建的Web 服务器，其目的是找到暴露的服务或其他可以利用的漏洞。只要具有可从Internet上免费获得的众多端口扫描器之，任何人都可以轻松实现这种攻击。这也是常见的攻击之，因为它很容易实现，脚本小子通常利用它窃取服务器的主机名或IP地址（不过，他们通常不知道如何解析获得的结果）。注意，高的攻击者将利用端口扫描挖掘信息。

⑵、ping洪水（flooding）攻击：

ping洪水攻击是种简单的DDoS攻击。在这种攻击中，个计算机向另个系统发送个包（ping），以找到关于服务或系统的信息。对于低的攻击，ping流可用于偷偷地查找信息，但是如果要截取向目标发送的包，则要求系统离线或停机。这种攻击虽然是“老牌攻击”，但它仍然很有活力，因为很多现代的操作系统都容易受到这种攻击。

⑶、SYN洪水（flooding）攻击：

这种攻击要求了解TCP/IP协议—即整个通信流程是如何工作的。通过个类比就能够很好地解释这种攻击。这种攻击类似于向某人发送封需要回复的信件，但是信封使用虚假的回信地址。收信人回复了信件并等待您的回复，但永远不能收到回信，因为它在某个地方被阻止了。只要针对系统的SYN请求足够多，攻击者就能够使用系统上的所有连接，从而阻止任何东西通过。

⑷、Smurf攻击：

这种攻击类似于ping洪水攻击，但它能巧妙地修改进程。在Smurf攻击中，先向中间网络发送个 ping 命令，然后在自身得到增强之后转发到攻击目标。以前的“点滴”流量现在变成了巨大的流量。幸运的是，这种攻击现在很少见。

⑸、FTP跳转攻击：

文件传输协议（FTP）跳转攻击指攻击者向有漏洞的FTP 服务器上传个结构特殊的文件，然后该服务器将这个文件转发到其他位置（通常是组织内的另个服务器）。被转发的文件通常包含某些代码，其目的是在终服务器上完成攻击者希望做的事情。

⑹、P分片(fragmentation)攻击：

在这种攻击中，攻击者凭借高的TCP/IP协议知识将包分成更小的片段（即分片），从而绕过许多入侵检测系统。在比较严重的情况下，这种攻击会造成挂起、锁定、重启和蓝屏等。不过，这种攻击不是般人能实施的。

⑺、SimpleNetworkManagementProtocol(SNMP)攻击：

SNMP攻击的主要目标是SNMP服务（用于管理网络及其上的设备）。因为SNMP用于管理网络设备，所以通过攻击该服务，攻击者能够详细了解网络的结构，从而为以后的攻击做准备。

**分类：**

按照TCP/IP协议的层次可将DDOS攻击分为基于ARP的攻击、基于ICMP的攻击、基于IP的攻击、基于UDP的攻击、基于TCP的攻击和基于应用层的攻击。

基于ARP

ARP是无连接的协议，当收到攻击者发送来的ARP应答时。它将接收ARP应答包中所提供的信息。更新ARP缓存。因此，含有错误源地址信息的ARP请求和含有错误目标地址信息的ARP应答均会使上层应用忙于处理这种异常而无法响应外来请求，使得目标主机丧失网络通信能力。产生拒绝服务，如ARP重定向攻击。

基于ICMP

攻击者向一个子网的广播地址发送多个ICMP Echo请求数据包。并将源地址伪装成想要攻击的目标主机的地址。这样，该子网上的所有主机均对此ICMP Echo请求包作出答复，向被攻击的目标主机发送数据包，使该主机受到攻击，导致网络阻塞。

基于IP

TCP/IP中的IP数据包在网络传递时，数据包可以分成更小的片段。到达目的地后再进行合并重装。在实现分段重新组装的进程中存在漏洞，缺乏必要的检查。利用IP报文分片后重组的重叠现象攻击服务器，进而引起服务器内核崩溃。如Teardrop是基于IP的攻击。

基于应用层

应用层包括SMTP，HTTP，DNS等各种应用协议。其中SMTP定义了如何在两个主机间传输邮件的过程，基于标准SMTP的邮件服务器，在客户端请求发送邮件时，是不对其身份进行验证的。另外，许多邮件服务器都允许邮件中继。攻击者利用邮件服务器持续不断地向攻击目标发送垃圾邮件，大量侵占服务器资源。

常规的一些防御方法：

1。确保服务器的系统文件是最新的版本,并及时更新系统补丁。

2。关闭不必要的服务。

3。限制同时打开的SYN半连接数目,缩短SYN半连接的time out 时间,限制SYN/ICMP流量

4。正确设置防火墙

5。认真检查网络设备和主机/服务器系统的日志。只要日志出现漏洞或是时间变更,那这台机器就可能遭到了攻击。

7。充分利用网络设备保护网络资源

8。 用足够的机器承受黑客攻击

9。 检查访问者的来源

## Http的传输方式(get,post,head.....)、分别什么作用（）

什么是协议：协议是通信的双方，在通信流程或内容格式上，共同遵守的标准。http协议是互联网中最常见的网络通信标准。

http请求的方法：

1、OPTIONS

返回服务器针对特定资源所支持的HTTP请求方法，也可以利用向web服务器发送‘\*’的请求来测试服务器的功能性

2、HEAD

类似于get请求，只不过响应体将不会被返回。这一方法可以再不必传输整个响应内容的情况下，就可以获取包含在响应小消息头中的元信息。

3、GET

向特定的资源发出请求。它本质就是发送一个请求来取得服务器上的某一资源。资源通过一组HTTP头和呈现数据（如HTML文本，或者图片或者视频等）返回给客户端。GET请求中，永远不会包含呈现数据。

4、POST

向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 Loadrunner中对应POST请求函数：web\_submit\_data,web\_submit\_form

5、PUT

向指定资源位置上传其最新内容

6、DELETE

请求服务器删除Request-URL所标识的资源

7、TRACE

回显服务器收到的请求，主要用于测试或诊断

8、CONNECT

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。



注意：

1）方法名称是区分大小写的，当某个请求所针对的资源不支持对应的请求方法的时候，服务器应当返回状态码405（Mothod Not Allowed）；当服务器不认识或者不支持对应的请求方法时，应返回状态码501（Not Implemented）。

2）HTTP服务器至少应该实现GET和HEAD/POST方法，其他方法都是可选的，此外除上述方法，特定的HTTP服务器支持扩展自定义的方法。

## get和post区别、应用场景

GET：从指定的服务器中获取数据

POST：提交数据给指定的服务器处理

GET方法：

使用GET方法时，查询字符串（键值对）被附加在URL地址后面一起发送到服务器：

/test/demo\_form.jsp?name1=value1&name2=value2

特点：

GET请求能够被缓存

GET请求会保存在浏览器的浏览记录中

以GET请求的URL能够保存为浏览器书签

GET请求有长度限制

GET请求主要用以获取数据

POST方法：

使用POST方法时，查询字符串在POST信息中单独存在，和HTTP请求一起发送到服务器：

POST /test/demo\_form.jsp HTTP/1.1

Host: w3schools.com

name1=value1&name2=value2

特点：

POST请求不能被缓存下来

POST请求不会保存在浏览器浏览记录中

以POST请求的URL无法保存为浏览器书签

POST请求没有长度限制

GET和POST的区别：

|  |  |  |
| --- | --- | --- |
|  | GET | POST |
| **点击返回/刷新按钮** | 没有影响 | 数据会重新发送（浏览器将会提示用户“数据被从新提交”） |
| **添加书签** | 可以 | 不可以 |
| **缓存** | 可以 | 不可以 |
| **编码类型(Encoding type)** | application/x-www-form-urlencoded | application/x-www-form-urlencoded or multipart/form-data. 请为二进制数据使用multipart编码 |
| **历史记录** | 有 | 没有 |
| **长度限制** | 有 | 没有 |
| **数据类型限制** | 只允许ASCII字符类型 | 没有限制。允许二进制数据 |
| **安全性** | 查询字符串会显示在地址栏的URL中，不安全，请不要使用GET请求提交敏感数据 | 因为数据不会显示在地址栏中，也不会缓存下来或保存在浏览记录中，所以看POST求情比GET请求安全，但也不是最安全的方式。如需要传送敏感数据，请使用加密方式传输 |
| **可见性** | 查询字符串显示在地址栏的URL中，可见 | 查询字符串不会显示在地址栏中，不可见 |

## 重定向(redirect))和连接跳转(forward)区别、应用场景

forward（转发）：

是服务器请求资源,服务器直接访问目标地址的URL,把那个URL的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,因为这个跳转过程实在服务器实现的，并不是在客户端实现的所以客户端并不知道这个跳转动作，所以它的地址栏还是原来的地址.

redirect（重定向）：

是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的URL.

转发是服务器行为，重定向是客户端行为。

|  |  |
| --- | --- |
| Forward | Redirect |
| 使用同一个request | 是不同的request |
| Forward后的语句不会继续发送给客户端 | Send后的语句会继续执行，除非return |
| 速度快 | 速度慢 |
| 服务器内部转换 | 需要到客户端的往返，可以转发任何页面 |
| 可以传参数 | 可以传参数，直接写在url后面 |
| 地址栏没有变化 | 地址栏有变化 |

应用场景：

forward仅是容器中控制权的转向，在客户端浏览器地址栏中不会显示出转向后的地址；

redirect则是完全的跳转，浏览器将会得到跳转的地址，并重新发送请求链接。这样，从浏览器的地址栏中可以看到跳转后的链接地址。

所以，forward更加高效，在forward可以满足需要时，尽量使用forward()方法，并且，这样也有助于隐藏实际的链接。

在有些情况下，比如，需要跳转到一个其它服务器上的资源，则必须使用sendRedirect()方法。

forward:一般用于用户登陆的时候,根据角色转发到相应的模块.

redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等.

## http几个状态码(200,301,302,304,400,401,403,404,500,502,503)

1xx：指示信息--表示请求已接收，继续处理

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--信息不完整需要进一步补充

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

|  |  |
| --- | --- |
| 200 | （成功）服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。 |
| 301 | （永久移动） 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。 |
| 302 | （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。 |
| 304 | （未修改） 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。 |
| 400 | （错误请求） 服务器不理解请求的语法。 |
| 401 | （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。 |
| 403 | （禁止） 服务器拒绝请求。 |
| 404 | （未找到） 服务器找不到请求的网页。 |
| 500 | （服务器内部错误） 服务器遇到错误，无法完成请求。 |
| 502 | （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。 |
| 503 | （服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态。 |

## http的request和response报文什么结构，header里面分别是什么，有哪些参数

https://blog.csdn.net/shouwang666666/article/details/70232053/

* http请求报文

一个http请求报文有四个部分组成：请求行、请求头部、空行、请求数据

* 请求行

请求行由请求方法字段、URL字段和HTTP协议版本字段3个字段组成，它们用空格分隔。比如 GET /data/info.html HTTP/1.1

* 请求头部

HTTP客户程序(例如浏览器)，向服务器发送请求的时候必须指明请求类型(一般是GET或者 POST)。如有必要，客户程序还可以选择发送其他的请求头。大多数请求头并不是必需的，但Content-Length除外。对于POST请求来说 Content-Length必须出现。

常见的请求头字段含义：

Accept： 浏览器可接受的MIME类型。

Accept-Charset：浏览器可接受的字符集。

Accept-Encoding：浏览器能够进行解码的数据编码方式，比如gzip。Servlet能够向支持gzip的浏览器返回经gzip编码的HTML页面。许多情形下这可以减少5到10倍的下载时间。

Accept-Language：浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到。

Authorization：授权信息，通常出现在对服务器发送的WWW-Authenticate头的应答中。

Content-Length：表示请求消息正文的长度。

Host： 客户机通过这个头告诉服务器，想访问的主机名。Host头域指定请求资源的Intenet主机和端口号，必须表示请求url的原始服务器或网关的位置。HTTP/1.1请求必须包含主机头域，否则系统会以400状态码返回。

If-Modified-Since：客户机通过这个头告诉服务器，资源的缓存时间。只有当所请求的内容在指定的时间后又经过修改才返回它，否则返回304“Not Modified”应答。

Referer：客户机通过这个头告诉服务器，它是从哪个资源来访问服务器的(防盗链)。包含一个URL，用户从该URL代表的页面出发访问当前请求的页面。

User-Agent：User-Agent头域的内容包含发出请求的用户信息。浏览器类型，如果Servlet返回的内容与浏览器类型有关则该值非常有用。

Cookie：客户机通过这个头可以向服务器带数据，这是最重要的请求头信息之一。

Pragma：指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝。

From：请求发送者的email地址，由一些特殊的Web客户程序使用，浏览器不会用到它。

Connection：处理完这次请求后是否断开连接还是继续保持连接。如果Servlet看到这里的值为“Keep- Alive”，或者看到请求使用的是HTTP 1.1(HTTP 1.1默认进行持久连接)，它就可以利用持久连接的优点，当页面包含多个元素时(例如Applet，图片)，显著地减少下载所需要的时间。要实现这一点，Servlet需要在应答中发送一个Content-Length头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。

Range：Range头域可以请求实体的一个或者多个子范围。例如，

表示头500个字节：bytes=0-499

表示第二个500字节：bytes=500-999

表示最后500个字节：bytes=-500

表示500字节以后的范围：bytes=500-

第一个和最后一个字节：bytes=0-0,-1

同时指定几个范围：bytes=500-600,601-999

但是服务器可以忽略此请求头，如果无条件GET包含Range请求头，响应会以状态码206(PartialContent)返回而不是以200 (OK)。

UA-Pixels，UA-Color，UA-OS，UA-CPU：由某些版本的IE浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和CPU类型。

* 空行

它的作用是通过一个空行，告诉服务器请求头部到此为止。

* 请求数据

若方法字段是GET，则此项为空，没有数据

若方法字段是POST,则通常来说此处放置的就是要提交的数据

比如要使用POST方法提交一个表单，其中有user字段中数据为“admin”, password字段为123456，那么这里的请求数据就是 user=admin&password=123456，使用&来连接各个字段。

总的来说，HTTP请求报文格式就如下图所示



* http响应报文

同样的，HTTP响应报文也由三部分组成：响应行、响应头、响应体

* 响应行

响应行一般由协议版本、状态码及其描述组成 比如 HTTP/1.1 200 OK

其中协议版本HTTP/1.1或者HTTP/1.0，200就是它的状态码，OK则为它的描述。

* 响应头

响应头用于描述服务器的基本信息，以及数据的描述，服务器通过这些数据的描述信息，可以通知客户端如何处理等一会儿它回送的数据。

* 响应体

响应体就是响应的消息体，如果是纯数据就是返回纯数据，如果请求的是HTML页面，那么返回的就是HTML代码，如果是JS就是JS代码，如此之类

响应报文格式如下



## https是什么，什么原理(必须掌握，问的很多)

我们都知道HTTPS能够加密信息，以免敏感信息被第三方获取，所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用HTTPS协议。



客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤，如图所示。

（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

（5）Web服务器利用自己的私钥解密出会话密钥。

（6）Web服务器利用会话密钥加密与客户端之间的通信。



HTTPS的优点：

尽管HTTPS并非绝对安全，掌握根证书的机构、掌握加密算法的组织同样可以进行中间人形式的攻击，但HTTPS仍是现行架构下最安全的解决方案，主要有以下几个好处：

（1）使用HTTPS协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；

（2）HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

（3）HTTPS是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

（4）谷歌曾在2014年8月份调整搜索引擎算法，并称“比起同等HTTP网站，采用HTTPS加密的网站在搜索结果中的排名将会更高”。

HTTPS的缺点：

虽然说HTTPS有很大的优势，但其相对来说，还是存在不足之处的：

（1）HTTPS协议握手阶段比较费时，会使页面的加载时间延长近50%，增加10%到20%的耗电；

（2）HTTPS连接缓存不如HTTP高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

（3）SSL证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

（4）SSL证书通常需要绑定IP，不能在同一IP上绑定多个域名，IPv4资源不可能支撑这个消耗。

（5）HTTPS协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行。

## HTTP和HTTPS的区别

HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## session和cookie的区别

cookie 和session 的区别：

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗，考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能

考虑到减轻服务器性能方面，应当使用COOKIE。

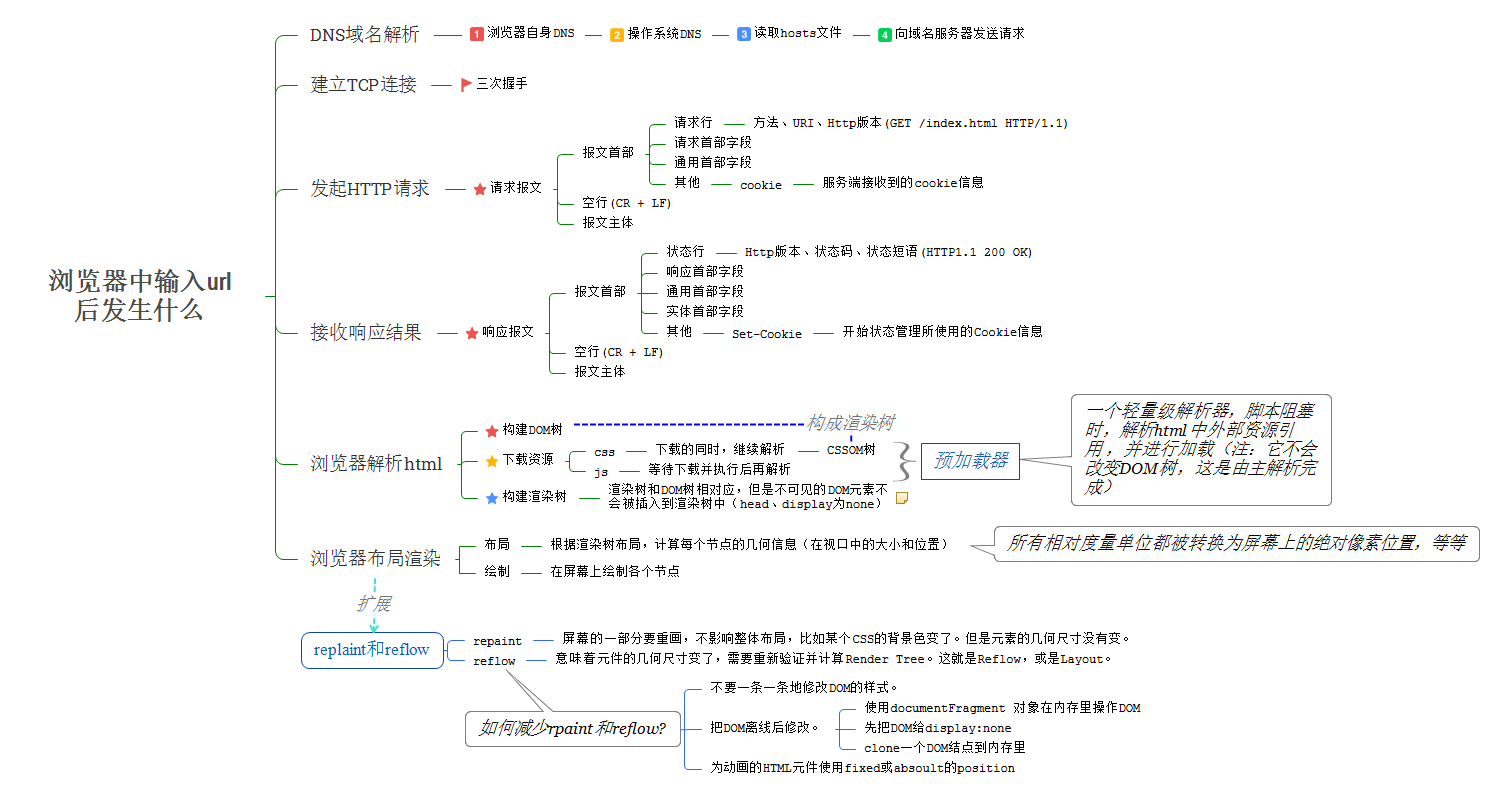
4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：

将登陆信息等重要信息存放为SESSION

其他信息如果需要保留，可以放在COOKIE中

## 浏览器输入url后执行的操作



DNS域名解析、建立TCP连接、发送HTTP请求、接受响应结果、浏览器解析html、浏览器布局渲染

1. DNS域名解析

在浏览器DNS缓存中搜索

在操作系统DNS缓存中搜索

读取系统hosts文件，查找其中是否有对应的ip

向本地配置的首选DNS服务器发起域名解析请求

2. 建立TCP连接

为了准确地传输数据，TCP协议采用了三次握手策略。发送端首先发送一个带SYN（synchronize）标志的数据包给接收方，接收方收到后，回传一个带有SYN/ACK(acknowledegment)标志的数据包以示传达确认信息。最后发送方再回传一个带ACK标志的数据包，代表握手结束。在这过程中若出现问题中断，TCP会再次发送相同的数据包。

TCP是一个端到端的可靠的面向连接的协议，所以HTTP基于传输层TCP协议不用担心数据的传输的各种问题。

3. 发起HTTP请求

请求方法：

GET:获取资源

POST:传输实体主体

HEAD:获取报文首部

PUT:传输文件

DELETE:删除文件

OPTIONS:询问支持的方法

TRACE:追踪路径

4. 接受响应结果

状态码：

1\*\*：信息性状态码

2\*\*：成功状态码

200：OK 请求正常处理

204：No Content请求处理成功，但没有资源可返回

206：Partial Content对资源的某一部分的请求

3\*\*：重定向状态码

301：Moved Permanently 永久重定向

302：Found 临时性重定向

304：Not Modified 缓存中读取

4\*\*：客户端错误状态码

400：Bad Request 请求报文中存在语法错误

401：Unauthorized需要有通过Http认证的认证信息

403：Forbidden访问被拒绝

404：Not Found无法找到请求资源

5\*\*：服务器错误状态码

500：Internal Server Error 服务器端在执行时发生错误

503：Service Unavailable 服务器处于超负载或者正在进行停机维护

5. 浏览器解析html

浏览器按顺序解析html文件，构建DOM树，在解析到外部的css和js文件时，向服务器发起请求下载资源，若是下载css文件，则解析器会在下载的同时继续解析后面的html来构建DOM树，则在下载js文件和执行它时，解析器会停止对html的解析。这便出现了js阻塞问题。

预加载器：

当浏览器被脚本文件阻塞时，预加载器（一个轻量级的解析器）会继续解析后面的html，寻找需要下载的资源。如果发现有需要下载的资源，预加载器在开始接收这些资源。预加载器只能检索HTML标签中的URL，无法检测到使用脚本添加的URL，这些资源要等脚本代码执行时才会获取。

注: 预解析并不改变Dom树，它将这个工作留给主解析过程

浏览器解析css，形成CSSOM树，当DOM树构建完成后，浏览器引擎通过DOM树和CSSOM树构造出渲染树。渲染树中包含可视节点的样式信息（不可见节点将不会被添加到渲染树中，如：head元素和display值为none的元素）

值得注意的是，这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的html都解析完成之后再去构建和布局render树。它是解析完一部分内容就显示一部分内容，同时，可能还在通过网络下载其余内容。

6. 浏览器布局渲染

布局：通过计算得到每个渲染对象在可视区域中的具体位置信息（大小和位置），这是一个递归的过程。

绘制：将计算好的每个像素点信息绘制在屏幕上

在页面显示的过程中会多次进行Reflow和Repaint操作，而Reflow的成本比Repaint的成本高得多的多。因为Repaint只是将某个部分进行重新绘制而不用改变页面的布局，如：改变了某个元素的背景颜色。而如果将元素的display属性由block改为none则需要Reflow。

# 操作系统

## 常用端口号(21,80,8080,443,25,110,23,1080)

<http://cert.sjtu.edu.cn/doc/linux/ch-ports.html>

|  |  |  |
| --- | --- | --- |
| 端口号 | 名称 | 注释 |
| 21 | ftp | 文件传输协议（FTP）端口；有时被文件服务协议（FSP）使用 |
| 23 | telnet | Telnet服务 |
| 25 | smtp | 简单邮件传输协议（SMTP） |
| 80 | http | 用于万维网（WWW）服务的超文本传输协议（HTTP） |
| 110 | Pop3 | 邮局协议版本3 |
| 1080 | socks | SOCKS网络应用程序代理服务 |
| 8080 | Webcache | 万维网（WWW）缓存服务 |

## 几个协议 FTP，SMTP，POPS

FTP：文件传输协议（File Transfer Protocol）是因特网中使用最广泛的文件传输协议。FTP使用交互式的访问，允许客户指定文件的类型和格式（如指明是否使用ASCII码），并允许文件具有存取权限（如访问文件的用户必须经过授权，并输入有效的口令）

SMTP：Simple Mail Transfer Protocol，简单邮件传输协议。它是一组用于从源地址到目的地址传输邮件的规范，通过它来控制邮件的中转方式。SMTP帮助每台计算机在发送或中转信件时找到下一个目的地。SMTP服务器就是遵循SMTP协议的发送邮件服务器。

POPS：Post Office Protocol 3邮局协议的第三个版本，它规定怎样将个人计算机连接到internet的邮件服务器和下载电子邮件的电子协议。它是因特网电子邮件的第一个离线协议标准，POP3允许用户从服务器上把邮件存储到本地主机（即自己的计算机）上，同时删除保存在邮件服务器上的邮件，而POP3服务器则是遵循POP3协议的邮件接收服务器，用来接收电子邮件。

## 进程和线程的区别

进程：是执行中的一段程序，程序一旦运行就是进程，进程是执行程序的一个实例

线程：线程是进程的一个实体，是程序执行的最小单位

* 进程是资源分配的最小单位；线程是程序执行的最小单位
* 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵，而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程小的多
* 线程间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信要以通信的方式（IPC）进行
* 多进程更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间

进程和线程的选择取决以下几点：

需要频繁创建销毁的优先使用线程；因为进程的创建和销毁一个进程代价是很大的。

线程的切换速度快，所以在需要大量计算，切换频繁使用线程是，还有耗时的操作使用线程可提高程序的响应

因为对CPU系统的效率上线程更占优，所以可能要发展到多机分布的用进程，多核分布用线程

并行操作时使用线程，如C/S架构的服务器端并发线程响应用户的请求

需要更稳定安全时，适合选择进程；需要速度时，选择线程更好

|  |  |  |  |
| --- | --- | --- | --- |
| **角度** | **多进程** | **对线程** | **结论** |
| 数据共享、同步 | 数据共享复杂，需要用IPC；数据是分开的，同步简单 | 因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂 | 各有优势 |
| 内存、CPU | 占用内存多，切换复杂，CPU利用率低 | 占用内存少，切换简单，CPU利用率高 | 线程占优 |
| 创建销毁、切换 | 创建销毁、切换复杂，速度慢 | 创建销毁、切换简单，速度快 | 线程占优 |
| 编程、调试 | 编程简单，调试简单 | 编程复杂，调试复杂 | 进程占优 |
| 可靠性 | 进程之间不会相互影响 | 一个线程挂掉导致整个进程挂掉 | 进程占优 |
| 分布式 | 适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单 | 适应于多核分布式 | 进程占优 |

## 进程的生命周期，几个状态的转化

1. 三态模型

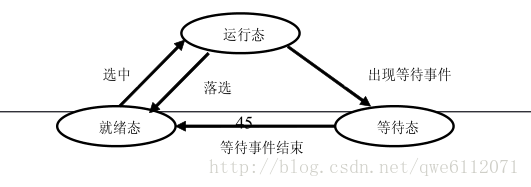
一个进程从创建而产生至撤销而消亡的整个生命周期，可以用一组状态加以刻划，根据三态模型，进程的生命周期可分为如下三种进程状态：

1. 运行态(running):占有处理器正在运行

2. 就绪态(ready):具备运行条件，等待系统分配处理器以便运行

3. 等待态(blocked):不具备运行条件，正在等待某个事件的完成

下面是三个状态的转换图：



运行状态的进程将由于出现等待事件而进入等待状态，当等待事件结束之后等待状态的进程将进入就绪状态，而处理器的调度策略又会引起运行状态和就绪状态之间的切换。

引起进程状态转换的具体原因如下：

运行态—→等待态：等待使用资源；如等待外设传输；等待人工干预。

等待态—→就绪态：资源得到满足；如外设传输结束；人工干预完成。

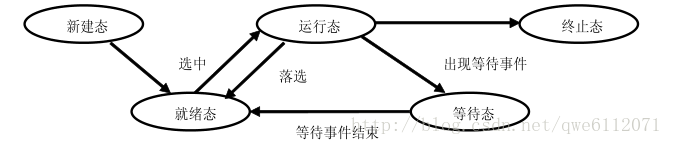
运行态—→就绪态：运行时间片到；出现有更高优先权进程。

就绪态—→运行态：CPU 空闲时选择一个就绪进程。

1. 五太模型

在一个实际的系统里进程的状态及其转换比上节叙述的会复杂一些，例如引入专门的新建态（new）和终止态（exit ）

状态转换图如下所示：



建态对应于进程刚刚被创建的状态。创建一个进程要通过两个步骤，

1. 为一个新进程创建必要的管理信息，

2. 让该进程进入就绪态。此时进程将处于新建态，它并没有被提交执行，而是在等待操作系统完成创建进程的必要操作。需要注意的是，操作系统有时将根据系统性能或主存容量的限制推迟新建态进程的提交

类似地，进程的终止也要通过两个步骤，首先,是等待操作系统进行善后，然后,退出主存。当一个进

程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进

程所终结，它将进入终止态。进入终止态的进程以后不再执行，但依然临时保留在操作系统中等待善后 。

一旦其他进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程。

引起进程状态转换的具体原因如下：

NULL—→新建态：执行一个程序，创建一个子进程。

新建态—→就绪态：当操作系统完成了进程创建的必要操作，并且当前系统的性能和虚拟内存的容量均允许。

运行态—→终止态：当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结。

终止态—→NULL：完成善后操作。

就绪态—→终止态：未在状态转换图中显示，但某些操作系统允许父进程终结子进程。

等待态—→终止态：未在状态转换图中显示，但某些操作系统允许父进程终结子进程。

1. 七太模型

进程的挂起

到目前为止，我们或多或少总是假设所有的进程都在内存中。事实上，可能出现这样一些情况，例如由于进程的不断创建，系统的资源已经不能满足进程运行的要求，这个时候就必须把某些进程挂起（suspend），对换到磁盘镜像区中，暂时不参与进程调度，起到平滑系统操作负荷的目的。

引起进程挂起的原因是多样的，主要有：

1. 系统中的进程均处于等待状态，处理器空闲，此时需要把一些阻塞进程对换出去，以腾出足够的内存装入就绪进程运行。

2. 进程竞争资源，导致系统资源不足，负荷过重，此时需要挂起部分进程以调整系统负荷 ,保证系统的实时性或让系统正常运行。

3. 把一些定期执行的进程（如审计程序、监控程序、记账程序）对换出去，以减轻系统负荷。

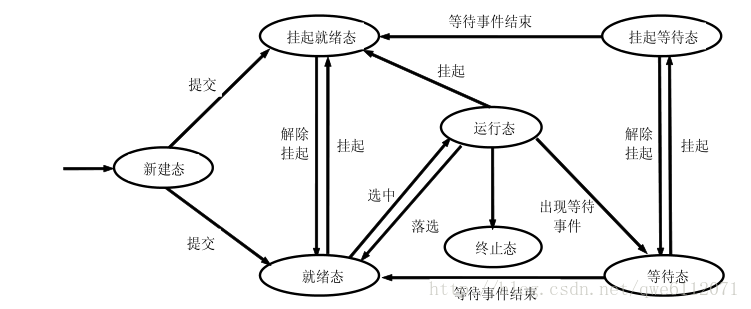
4. 用户要求挂起自己的进程，以便根据中间执行情况和中间结果进行某些调试、检查和改正。

5. 父进程要求挂起自己的后代进程，以进行某些检查和改正。

6. 操作系统需要挂起某些进程，检查运行中资源使用情况，以改善系统性能;或当系统出现故障或某些功能受到破坏时，需要挂起某些进程以排除故障。

七态模型描述

下面是具有七态模型的进程状态及其转换：



在此类系统中，进程增加了两个新状态：

挂起就绪态（ready,suspend）

挂起就绪态表明了进程具备运行条件但目前在二级存储器中，只有当它被对换到主存才能被调度执行

挂起等待态（blocked,suspend）挂起等待态则表明了进程正在等待某一个事件且在二级存储器中。

引起进程状态转换的具体原因如下：

等待态—→挂起等待态：如果当前不存在就绪进程，那么至少有一个等待态进程将被对换出去成为挂起等待态；操作系统根据当前资源状况和性能要求，可以决定把等待态进程对换出去成为挂起等待态。

挂起等待态—→挂起就绪态：引起进程等待的事件发生之后，相应的挂起等待态进程将转换为挂起就绪态。

挂起就绪态—→就绪态：当内存中没有就绪态进程，或者挂起就绪态进程具有比就绪态进程更高的优先级，系统将把挂起就绪态进程转换成就绪态。

就绪态—→挂起就绪态：操作系统根据当前资源状况和性能要求，也可以决定把就绪态进程对换出去成为挂起就绪态。

挂起等待态—→等待态：当一个进程等待一个事件时，原则上不需要把它调入内存。但是在下面一种情况下，这一状态变化是可能的。当一个进程退出后，主存已经有了一大块自由空间,而某个挂起等待态进程具有较高的优先级并且操作系统已经得知导致它阻塞的事件即将结束，此时便发生了这一状态变化。

运行态—→挂起就绪态：当一个具有较高优先级的挂起等待态进程的等待事件结束后，它需要抢占 CPU，，而此时主存空间不够，从而可能导致正在运行的进程转化为挂起就绪态。另外处于运行态的进程也可以自己挂起自己。

新建态—→挂起就绪态：考虑到系统当前资源状况和性能要求，可以决定新建的进程将被对换出去成为挂起就绪态。

## 进程间通信的方式，管道，socket，共享内存

* 管道（pipe）：

管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有血缘关系的进程间使用。进程的血缘关系通常指父子进程关系。

* 命名管道FIFO：

有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间通信。

* 消息队列MessageQueue：

消息队列是由消息组成的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

* 信号量（semaphore）：

信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它通常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

* 共享内存（shared memory）：

共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问，共享内存是最快的IPC方式，它是针对其他进程间的通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。

* 套接字（socket）：

套接口也是一种进程间的通信机制，与其他通信机制不同的是它可以用于不同及其间的进程通信。

## linux常用命令

<https://blog.csdn.net/xufei512/article/details/53321980/>

# 数据库（Mysql）

## 主键、超键、候选键、外键

超键（super key）：

在关系中能唯一表示元素属性的集成为关系模型的超键

候选键（candidate key）：

不含有多余属性的超键称为候选键。也就是说在候选键中再删除属性就不是键了

主键（primary key）：

用户选作元组表示的候选键为主键。一般不做说明，键就是主键

数据库表中对存储数据对象予以唯一和完整表示的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，及不能为空（null）

外键（foreign key）：

在一个表中存在的另一个表的主键称此表的外键

如果关系模式R1中某属性集不是R1的主键，而是另一个关系R2的主键则该属性集是关系模式R1的外键。

学生成绩信息表中有（学号、身份证号、姓名、性别、年龄、系别、专业等）

超键：

学生表中含有学号或者身份证号的任意组合都为此表的超键。如：（学号）、（学号，姓名）、（学号，性别）等

候选键：

学号唯一，而且没有多余属性，所以是一个候选键。身份证号唯一，而且没有多余属性，所以是一个候选键。（学号+性别）不是候选键。

主键：

主键就是候选键里面的一个，是人为规定的，例如学生表中，我们通常会让“学号”做主键，学号能唯一标识这一个元组。

外键：

外键就很简单了，假如我们还有一个教师表，每个教师都有自己的编号，假设老师编号在老师这个层次中是主键，在学生表中它就是外键。

## 视图、游标

**视图：**

是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有一个表或者多个表的行或列的子集。对视图的修改会影响基本表。它使得我们获取数据更容易，相比多表查询。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by 则对视图再次order by将被覆盖。

创建视图：create view XXX as XXXXXXXXXXXXXX;

对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

**游标：**

是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

## 存储过程

存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需创建一次，以后再改程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。调用：可以用一个命令对象调用次存储过程。可以供外部程序调用

存储过程优点：1）存储过程是预编译过的，执行效率高。 2）存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。 3）安全性高，执行存储过程需要有一定权限的用户。 4）存储过程可以重复使用，可减少数据库开发人员的工作量。 缺点：移植性差

## 存储过程与函数的区别

|  |  |
| --- | --- |
| 存储过程 | 函数 |
| 用于在数据库中完成特定的操作或者任务（如插入、删除） | 用于特定的数据（如选择） |
| 程序头部用声明用procedure | 程序头部声明用function |
| 程序头部声明不需要描述返回类型 | 程序头部声明需要描述返回类型，而且PL/SQL块中至少要包括一个有效的return语句 |
| 可以使用in/out/in out 三种模式的参数 | 可以使用in/out/in out三种模式的参数 |
| 可作为一个独立的PL/SQL语句来执行 | 不能独立执行，必须作为表达式的一部分调用 |
| 可以通过out/in out 返回零个或多个值 | 通过return语句返回一个值，且改值要与声明部分一致，也可以是通过out类型的参数带出的变量 |
| SQL语句(DML 或SELECT)中不可调用存储过程 | SQL语句(DML 或SELECT)中可以调用函数 |

## 触发器

触发器是一中特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

## 视图和游标

视图：是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有一个表或者多个表的行或列的子集。对视图的修改会影响基本表。它使得我们获取数据更容易，相比多表查询。

视图的优缺点：

优点： 1）对数据库的访问，因为视图可以有选择性的选取数据库里的一部分。 2 ）用户通过简单的查询可以从复杂查询中得到结果。 3 ）维护数据的独立性，试图可从多个表检索数据。 4 ）对于相同的数据可产生不同的视图。 缺点： 性能：查询视图时，必须把视图的查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，那么就无法更改数据

并非所有的视图都是可更新的。如果MySQL不能正确的确定被更新的基数据，则不允许更新（包括插入和删除）。如果视图定义中有以下操作，则不能进行视图的更新：分组（使用 GROUP BY 和 HAVING）；联结；子查询；并；聚集函数（Min()、Count()、Sum()等）；DISTINCT；导出（计算）列。

游标：是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

## 几种表的连接方式

内连接、自连接、外连接（左、右、全）、交叉连接 内连接：只有两个元素表相匹配的才能在结果集中显示。 外连接： 左外连接:左边为驱动表，驱动表的数据全部显示，匹配表的不匹配的不会显示。 右外连接:右边为驱动表，驱动表的数据全部显示，匹配表的不匹配的不会显示。 全外连接：连接的表中不匹配的数据全部会显示出来。 交叉连接： 笛卡尔效应，显示的结果是链接表数的乘积。

1、内联接（典型的联接运算，使用像 = 或 <> 之类的比较运算符）。包括相等联接和自然联接。

内联接使用比较运算符根据每个表共有的列的值匹配两个表中的行。例如，检索 students和courses表中学生标识号相同的所有行。

2、外联接。外联接可以是左向外联接、右向外联接或完整外部联接。

在 FROM子句中指定外联接时，可以由下列几组关键字中的一组指定：

1）LEFT JOIN或LEFT OUTER JOIN

左向外联接的结果集包括 LEFT OUTER子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。

2）RIGHT JOIN 或 RIGHT OUTER JOIN

右向外联接是左向外联接的反向联接。将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。

3）FULL JOIN 或 FULL OUTER JOIN

完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

3、交叉联接

交叉联接返回左表中的所有行，左表中的每一行与右表中的所有行组合。交叉联接也称作笛卡尔积。

FROM 子句中的表或视图可通过内联接或完整外部联接按任意顺序指定；但是，用左或右向外联接指定表或视图时，表或视图的顺序很重要

## 在数据库中查询语句速度很慢，如何优化

1.建索引 2.减少表之间的关联 3.优化sql，尽量让sql很快定位数据，不要让sql做全表查询，应该走索引,把数据 量大的表排在前面 4.简化查询字段，没用的字段不要，已经对返回结果的控制，尽量返回少量数据 5.尽量用PreparedStatement来查询，不要用Statement

## 数据库三范式

第一范式（1NF）：强调的是列的原子性，即列不能够再分成其他几列。

第二范式（2NF）：首先是 1NF，另外包含两部分内容，一是表必须有主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

第三范式（3NF）：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖

## union和union all的区别

UNION在进行表链接后会筛选掉重复的记录，所以在表链接后会对所产生的结果集进行排序运算，删除重复的记录再返回结果。实际大部分应用中是不会产生重复的记录，最常见的是过程表与历史表UNION。 UNION ALL只是简单的将两个结果合并后就返回。这样，如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据了。 从效率上说，UNION ALL 要比UNION快很多，所以，如果可以确认合并的两个结果集中不包含重复的数据的话，那么就使用UNION ALL。

## Varchar2和varchar有什么区别

Char的长度是固定的，而varchar2的长度是可以变化的，比如，存储字符串“abc”对于char(20)，表示你存储的字符将占20个字节，包含17个空，而同样的varchar2（20）只占了3个字节，20只是最大值，当你存储的字符小于20时，按实际长度存储。 char的效率要被varchar2的效率高。 目前varchar是varchar2的同义词，工业标准的varchar类型可以存储空字符串，但是oracle不能这样做，尽管它保留以后这样做的权利。Oracle自己开发了一个数据类型varchar2,这个类型不是一个标准的varchar，他将在数据库中varchar列可以存储空字符串的特性改为存储null值，如果你想有向后兼容的能力，oracle建议使用varchar2而不是varchar

## drop、delete与truncate的区别

SQL中的drop、delete、truncate都表示删除，但是三者有一些差别

* delete和truncate只删除表的数据不删除表的结构
* 速度,一般来说: drop> truncate >delete
* delete语句是dml,这个操作会放到rollback segement中,事务提交之后才生效;如果有相应的trigger,执行的时候将被触发. truncate,drop是ddl, 操作立即生效,原数据不放到rollback segment中,不能回滚. 操作不触发trigger.

不再需要一张表的时候，用drop

想删除部分数据行时候，用delete，并且带上where子句

保留表而删除所有数据的时候用truncate

## 数据库事务、锁

事务就是被绑定在一起作为一个逻辑工作单元的SQL语句分组，如果任何一个语句操作失败那么整个操作就被失败，以后操作就会回滚到操作前状态，或者是上有个节点。为了确保要么执行，要么不执行，就可以使用事务。要将一组语句作为事务考虑，就需要通过ACID测试，即ACID,原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。

原子性：

整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性：

在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏，一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

隔离性：

隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。

持久性：

在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

原子性（Atomicity）：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。

一致性（Consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束。

隔离性（Isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行。

持久性（Durability）：一个事务一旦提交，他对数据库的修改应该永久保存在数据库中。

**锁：**在所以的DBMS中，锁是实现事务的关键，锁可以保证事务的完整性和并发性。与现实生活中锁一样，它可以使某些数据的拥有者，在某段时间内不能使用某些数据或数据结构。当然锁还分级别的。

## 脏读，不可重复读，幻读

<https://blog.csdn.net/aluomaidi/article/details/52460844>

脏读：

脏读又称无效数据读出。一个事务读取另外一个事务还没有提交的数据叫脏读。

例如：事务T1修改了一行数据，但是还没有提交，这时候事务T2读取了被事务T1修改后的数据，之后事务T1因为某种原因Rollback了，那么事务T2读取的数据就是脏的。

解决办法：把数据库的事务隔离级别调整到READ\_COMMITTED

不可重复读：

不可重复读是指在同一个事务内，两个相同的查询返回了不同的结果。

例如：事务T1读取某一数据，事务T2读取并修改了该数据，T1为了对读取值进行检验而再次读取该数据，便得到了不同的结果。 解决办法：把数据库的事务隔离级别调整到REPEATABLE\_READ

幻读：

例如：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样。这就叫幻读。

解决办法：把数据库的事务隔离级别调整到SERIALIZABLE\_READ

脏读、不可重复读、幻读的级别高低是：脏读 < 不可重复读 < 幻读。所以，设置了最高级别的SERIALIZABLE\_READ就不用在设置REPEATABLE\_READ和READ\_COMMITTED了

| 隔离级别 | 脏读(Dirty Read) | 不可重复读(NoneRepeatable Read) | 幻读（Phantom Read） |
| --- | --- | --- | --- |
| 未提交读(Read uncommitted) | 可能 | 可能 | 可能 |
| 已提交读(Read committed) | 不可能 | 可能 | 可能 |
| 可重复读(Repeatable read) | 不可能 | 不可能 | 可能 |
| 可串行化(Serializable) | 不可能 | 不可能 | 不可能 |

## 数据库隔离

1.READ UNCIMMITTED（读未提交）

事务中的修改，即使没有提交，其他事务也可以看得到，比如说上面的两步这种现象就叫做脏读，这种隔离级别会引起很多问题，如无必要，不要随便使用。

2.READ COMMITTED（读已提交）（Oracle默认）

大多数数据库系统的默认隔离级别是READ COMMITTED，这种隔离级别就是一个事务的开始，只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。这种级别会出现读取旧数据的现象。

3.REPEATABLE READ（可重复读）（MySQL默认）

REPEATABLE READ解决了脏读的问题，该级别保证了每行的记录的结果是一致的，也就是上面说的读了旧数据的问题，但是却无法解决另一个问题，幻行，顾名思义就是突然蹦出来的行数据。指的就是某个事务在读取某个范围的数据，但是另一个事务又向这个范围的数据去插入数据，导致多次读取的时候，数据的行数不一致。

4.SERIALIZABLE（串行化）

SERIALIZABLE是最高的隔离级别，它通过强制事务串行执行（注意是串行），避免了前面的幻读情况，由于他大量加上锁，导致大量的请求超时，因此性能会比较底下，再特别需要数据一致性且并发量不需要那么大的时候才可能考虑这个隔离级别。

## MySQL存储引擎中的MyISAM和InnoDB区别

MyISAM是MySQL的默认数据库引擎（5.5版之前）

MyISAM与InnoDB的区别

* 存储结构

MyISAM：每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。

InnoDB：所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。

* 存储空间

MyISAM：可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。

InnoDB：需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

* 可移植性、备份即恢复

MyISAM：数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。

InnoDB：免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

* 事务支持

MyISAM：强调的是性能，每次查询具有原子性,其执行数度比InnoDB类型更快，但是不提供事务支持。

InnoDB：提供事务支持事务，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

* AUTO\_INCREMENT

MyISAM：可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。

InnoDB：InnoDB中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

* 表锁差异

MyISAM：只支持表级锁，用户在操作myisam表时，select，update，delete，insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。

InnoDB：支持事务和行级锁，是innodb的最大特色。行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。

* 全文索引

MyISAM：支持 FULLTEXT类型的全文索引

InnoDB：不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。

* 表主键

MyISAM：允许没有任何索引和主键的表存在，索引都是保存行的地址。

InnoDB：如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

* 表的具体行数

MyISAM：保存有表的总行数，如果select count(\*) from table;会直接取出出该值。

InnoDB：没有保存表的总行数，如果使用select count(\*) from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。

* CRUD操作

MyISAM：如果执行大量的SELECT，MyISAM是更好的选择。

InnoDB：如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

* 外键

MyISAM：不支持

InnoDB：支持

通过上述的分析，基本上可以考虑使用InnoDB来替代MyISAM引擎了，原因是InnoDB自身很多良好的特点，比如事务支持、存储 过程、视图、行级锁定等等，在并发很多的情况下，相信InnoDB的表现肯定要比MyISAM强很多。另外，任何一种表都不是万能的，只用恰当的针对业务类型来选择合适的表类型，才能最大的发挥MySQL的性能优势。如果不是很复杂的Web应用，非关键应用，还是可以继续考虑MyISAM的，这个具体情况可以自己斟酌。

2. 为什么MyISAM会比Innodb 的查询速度快。

INNODB在做SELECT的时候，要维护的东西比MYISAM引擎多很多；

1）数据块，INNODB要缓存，MYISAM只缓存索引块， 这中间还有换进换出的减少；

2）innodb寻址要映射到块，再到行，MYISAM 记录的直接是文件的OFFSET，定位比INNODB要快

3）INNODB还需要维护MVCC一致；虽然你的场景没有，但他还是需要去检查和维护

MVCC ( Multi-Version Concurrency Control )多版本并发控制

3. 应用场景

MyISAM适合：(1)做很多count 的计算；(2)插入不频繁，查询非常频繁；(3)没有事务。

InnoDB适合：(1)可靠性要求比较高，或者要求事务；(2)表更新和查询都相当的频繁，并且行锁定的机会比较大的情况。

## 聚簇索引和非聚簇索引是啥，区别，应用场景

聚簇索引（聚集索引）：顺序结构（索引项的顺序）与数据存储物理结构一致的一种索引，并且一个表的聚簇索引只能有唯一的一条；对于聚集索引，叶子节点即存储了真实点的数据，不再有另外单独的数据页

非聚簇索引（非聚集索引）：索引顺序与数据的物理存储结构无关。对于非聚集索引，叶结点包含索引字段值及指向数据页数据行的逻辑指针，其行数量与数据表行数据量一致，一个表的非聚集索引可以有多条

应用场景：

不要索引数据量不大的表，对于小表来讲，表扫描的成本并不高

不要设置过多的索引，在没有聚集索引的表中，最大额可以设置249个非聚集索引，过多的索引首先会带来更大的磁盘空间，而在数据发生修改时，对索引的维护是特别消耗时间的

合理应用复合索引，有些情况下可以考虑创建包含所有输出列的覆盖索引

对经常使用范围查询的字段，可以考虑聚集索引

避免对不常用的列，逻辑性列，大字段列创建索引

## 索引

索引就一种特殊的查询表，数据库的搜索可以利用它加速对数据的检索。它很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据。索引可以是唯一的，创建索引允许指定单个列或者是多个列。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

唯一、不为空、经常被查询的字段适合建索引

MySQL数据库几个基本的索引类型：普通索引、唯一索引、主键索引、全文索引

索引加快数据库的检索速度

索引降低了插入、删除、修改等维护任务的速度

唯一索引可以确保每一行数据的唯一性

通过使用索引，可以在查询的过程中使用优化隐藏器，提高系统的性能

索引需要占物理和数据空间

## 索引是啥，什么原理，Innodb是什么样的数据结构

索引是对数据库表中一个或多个列（例如，employee 表的姓氏 (lname) 列）的值进行排序的结构，索引就一种特殊的查询表，数据库的搜索可以利用它加速对数据的检索。它很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据。索引可以是唯一的，创建索引允许指定单个列或者是多个列。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

索引的主要思想是将数据分段，从而减少查询时的无效数据，提高查询效率。比如有1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段……这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。

**索引的类型**，根据数据库的功能，可以在数据库设计器中创建索引：唯一索引、主键索引、聚集索引。尽管唯一索引有助于定位信息，但为获得最佳性能结果，建议改用主键或唯一约束。

唯一索引：UNIQUE

表明此索引值只对应唯一的数据记录，对于单列唯一性索引，这保证单列不包含重复的值。对于多列唯一性索引，保证多个值的组合不会重复

主键索引：primary

数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引（聚簇索引）：cluster

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。 如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

Innodb是什么样的数据结构，选用B+树实现索引

## 联合索引，什么时候索引失效

联合索引：

又名复合索引，有两个或多个列的索引。它规定Mysql从左到右的使用索引字段，对字段的顺序有一定的要求。一个查询可以只使用索引的一部分，更准去的说是最左侧部分（最左优先）。如索引是key index (a,b,c). 可以支持a | a,b| a,b,c 3种组合进行查找，但不支持 b,c进行查找 .当最左侧字段是常量引用时，索引就十分有效。

索引失效：

* 如果条件中有or，即使其中有条件带索引也不会使用，要想索引生效只能让每个or条件中的每个列都加上索引
* 对于多列索引不是使用的第一部分，则不会使用索引
* like查询是以%开头
* mysql使用不等于（!= 或者<>）的时候无法使用索引
* is null，is not null 也无法使用索引
* 如果类型是字符串，那一定要在条件中将数据使用引号引起来，否则不使用索引
* 如果mysql估计使用全表扫描比使用索引要快，则不使用索引

## 数据库索引的优缺点

<https://blog.csdn.net/miracleww/article/details/53352738>

1. 为什么要建立索引，即索引的优点：

① 建立索引的列可以保证行的唯一性，生成唯一的rowId

② 建立索引可以有效缩短数据的检索时间

③ 建立索引可以加快表与表之间的连接

④ 为用来排序或者是分组的字段添加索引可以加快分组和排序顺序

2. 索引的缺点：

① 创建索引和维护索引需要时间成本，这个成本随着数据量的增加而加大

② 创建索引和维护索引需要空间成本，每一条索引都要占据数据库的物理存储空间，数据量越大，占用空间也越大（数据表占据的是数据库的数据空间）

③ 会降低表的增删改的效率，因为每次增删改索引需要进行动态维护，导致时间变长

## 慢查询

1简介

开启慢查询日志，可以让MySQL记录下查询超过指定时间的语句，通过定位分析性能的瓶颈，才能更好的优化数据库系统的性能。

2 参数说明

slow\_query\_log 慢查询开启状态

slow\_query\_log\_file 慢查询日志存放的位置（这个目录需要MySQL的运行帐号的可写权限，一般设置为MySQL的数据存放目录）

long\_query\_time 查询超过多少秒在记录

3.设置步骤：

* 查看慢查询相关参数

mysql> show variables like 'slow\_query%';

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

| Variable\_name | Value |

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

| slow\_query\_log | OFF |

| slow\_query\_log\_file | /mysql/data/localhost-slow.log |

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

mysql> show variables like 'long\_query\_time';

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

| Variable\_name | Value |

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

| long\_query\_time | 10.000000 |

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

* 设置方法
  + 全局变量设置

方法一：全局变量设置

将 slow\_query\_log 全局变量设置为“ON”状态

mysql> set global slow\_query\_log='ON';

设置慢查询日志存放的位置

mysql> set global slow\_query\_log\_file='/usr/local/mysql/data/slow.log';

查询超过1秒就记录

mysql> set global long\_query\_time=1;

* + 方法二：配置文件设置

修改配置文件my.cnf，在[mysqld]下的下方加入

[mysqld]

slow\_query\_log = ON

slow\_query\_log\_file = /usr/local/mysql/data/slow.log

long\_query\_time = 1

* 重启Mysql服务

service mysqld restart

* 查看设置后的参数

mysql> show variables like 'slow\_query%';

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

| Variable\_name | Value |

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

| slow\_query\_log | ON |

| slow\_query\_log\_file | /usr/local/mysql/data/slow.log |

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

mysql> show variables like 'long\_query\_time';

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

| Variable\_name | Value |

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

| long\_query\_time | 1.000000 |

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

## 数据库调优的几种方式

* 选用适当的字段属性

MySQL可以很好的支持大数据量的存取，但是一般说来，数据库中的表越小，在它上面执行的查询也就会越快。因此，在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。

* 使用连接（JOIN）来代替子查询（Sub-Queries）

连接（JOIN）..之所以更有效率一些，是因为MySQL不需要在内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。

* 使用联合（UNION）来代替手动创建的临时表
* 事务

事务的另一个重要作用是当多个用户同时使用相同的数据源时，它可以利用锁定数据库的方法来为用户提供一种安全的访问方式，这样可以保证用户的操作不被其它的用户所干扰。

* 锁定表

由于在事务执行的过程中，数据库将会被锁定，因此其它的用户请求只能暂时等待直到该事务结束。假设有成千上万的用户同时访问一个数据库系统，例如访问一个电子商务网站，就会产生比较严重的响应延迟。可以通过锁定表的方法来获得更好的性能

* 使用索引
* 慢速查询日式

## 各个引擎分别支持哪些索引(b-tree,hash,r-tree,full-text)

FULLTEX全文索引只有MyISAM支持

HASH只有MEMORY引擎显式地支持这种索引

BTREE MyISAM Innodb

BTREE在MyISAM里的形式和Innodb稍有不同

在 Innodb里，有两种形态：一是primary key形态，其leaf node里存放的是数据，而且不仅存放了索引键的数据，还存放了其他字段的数据。二是secondary index，其leaf node和普通的BTREE差不多，只是还存放了指向主键的信息.

而在MyISAM里，主键和其他的并没有太大区别。不过和Innodb不太一样的地方是在MyISAM里，leaf node里存放的不是主键的信息，而是指向数据文件里的对应数据行的信息

RTREE

RTREE在mysql很少使用，仅支持geometry数据类型，支持该类型的存储引擎只有MyISAM、BDb、InnoDb、NDb、Archive几种。

相对于BTREE，RTREE的优势在于范围查找.

各索引使用情况

（1）对于BTREE这种Mysql默认的索引类型，具有普遍的适用性

（2）由于FULLTEXT对中文支持不是很好，在没有插件的情况下，最好不要使用。其实，一些小的博客应用，只需要在数据采集时，为其建立关键字列表，通过关键字索引，也是一个不错的方法，至少愚安我是经常这么做的。

（3）对于一些搜索引擎级别的应用来说，FULLTEXT同样不是一个好的处理方法，Mysql的全文索引建立的文件还是比较大的，而且效率不是很高，即便是使用了中文分词插件，对中文分词支持也只是一般。真要碰到这种问题，Apache的Lucene或许是你的选择。

（4）正是因为hash表在处理较小数据量时具有无可比拟的素的优势，所以hash索引很适合做缓存（内存数据库）。如mysql数据库的内存版本Memsql，使用量很广泛的缓存工具Mencached，NoSql数据库redis等，都使用了hash索引这种形式。当然，不想学习这些东西的话Mysql的MEMORY引擎也是可以满足这种需求的。

# 数据结构

## 常见数据结构

线性表（数组、链表、栈、队列、串、树）

## 算法时间复杂度的计算

定义：在进行算法分析时， 语句总的执行次数T ( n )是关子问题规模n的函数，进而分析T ( n )随n 的变化情况并确定T(n)的数量级。算法的时间复杂度，也就是算法的时间量度，记f作：T ( n )= O(f(n))。它表示示随问题规模n 的增大，算法执行时间的增长率和f (n)的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中f ( n) 是问题规模n 的某个函数。

这样用大写O()来体现算法时间复杂度的记法，我们称之为大O记法。

推到大O阶：

1. 用常数1取代运行时间中的所有加法常数项。
2. 在修改后的运行次数函数中，只保留最高阶项
3. 如果最高阶项存在且不是1，则去除与这个项相乘的常数

得到的结果就是大O阶。

常数阶：O(1)

int sum = 0, n = 100; //执行1次

sum = (1 + n) \* n / 2; //执行第1次

sum = (1 + n) \* n / 2; //执行第2次

sum = (1 + n) \* n / 2; //执行第3次

sum = (1 + n) \* n / 2; //执行第4次

sum = (1 + n) \* n / 2; //执行第5次

sum = (1 + n) \* n / 2; //执行第6次

System.out.printf("%d",sum); //执行1次

线性阶：O(n)

int i;

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

//时间复杂度为O(1)的程序步骤序列

}

对数阶：O(logn)

int count = 1;

while(count < n){

count = count \* 2;

}

优于每次count诚意2以后，就距离n更近了一份。也就是说，有多少个2相乘后大于n,则会推出循环。有2x=n 得到x=log2n.所以此循环的时间复杂度为O(logn)

平方阶：O(n2)

int i, j;

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

for(j = 0; j < n;j++){

// 时间复杂度为O(1)的程序步骤序列

}

}

对于外层的循环，内部这个时间复杂度为O(n)的语句，再循环n次。所以时间复杂度为O(n2)

int i, j;

for(i = 0; i < m;i++){

for(j = 0; j < n;j++){

// 时间复杂度为O(1)的程序步骤序列

}

}

此时时间复杂度为O(m\*n)

|  |  |  |
| --- | --- | --- |
| 执行次函数 | 阶 | 非正式术语 |
| 12 | O(1) | 常数阶 |
| 2n+3 | O(n) | 线性阶 |
| 3n2+2n+1 | O(n2) | 平方阶 |
| 5log2n + 20 | O(logn) | 对数阶 |
| 2n+3nlog2n+19 | O(nlogn) | nlogn阶 |
| 6n3+2n2+3n+4 | O(n3) | 立方阶 |
| 2n | O(2n) | 指数阶 |

常用时间复杂度所耗费的时间右大到小一次是：

O(1)<O(loogn)<O(n) <O(nlogn) <O(n2) <O(n3) <O(2n)<O(n!) <O(nn)

## 二叉树、二叉排序树、平衡二叉树、

* 二叉树：二叉树（Binary Tree）是n(n >= 0)非节点的有限集合，该集合或者为空集（空二叉树），或者由一个根节点和连个互不相交的、分别称为根节点的左子树和有子树的二叉树组成：
* 二叉树特点：每个节点最多有两棵子树，节点的度最大为2.左子树和右子树是有顺序的，次序不能颠倒。即使某节点只有一个子树，也要区分左右子树。
* 二叉树的截个特例：斜树、满二叉树、完全二叉树
  + 斜树：完全二叉树、满二叉树
  + 满二叉树：所有的分支结点都存在左子树和右子树，并且所有的叶子结点都在同一层上，这样就是满二叉树。就是完美圆满的意思，关键在于树的平衡。
  + 完全二叉树：对一棵具有n个结点的二叉树按层序排号，如果编号为i的结点与同样深度的满二叉树编号为i结点在二叉树中位置完全相同，就是完全二叉树。满二叉树必须是完全二叉树，反过来不一定成立。
* 二叉搜索树：二叉搜索树（Binary Search Tree）又称二叉排序树（Binary Sort Tree）。二叉搜索树是一棵空树或者一棵具有下列性质的非空二叉树：

若左子树非空，则左子树上的所有节点的关键字值均小于根节点的关键字的值

若右子树非空，则右子树上的所有节点的关键字值均大于跟节点的关键字的值

左右子树本身也是一棵二叉搜索树

二叉搜索树的中序遍历可以得到一个递增的有序序列

* 平衡二叉树：（Balanced Binary Tree）又被称为AVL树。它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

## 二叉树的前序遍历、中序遍历、后续遍历、层序遍历

前序递归遍历算法：访问根结点-->递归遍历根结点的左子树-->递归遍历根结点的右子树

中序递归遍历算法：递归遍历根结点的左子树-->访问根结点-->递归遍历根结点的右子树

后序递归遍历算法：递归遍历根结点的左子树-->递归遍历根结点的右子树-->访问根结点

层序遍历：从根节点开始，依次向下，对于每一层从左向右遍历。

## 字符串模式匹配算法

在一个文本或较长的一段字符串中，找出一个或多个指定字符串，并返回其位置。

* Brute-Force算法

从目标字符串初始位置开始，依次分别于Pattern的各个位置的字符进行比较，如相同，比较下一个位置的字符直至完全匹配；如果不同则调到目标字符串的下一位置继续如此与Pattern比较，知道找到匹配字符串并返回其位置。

KMP算法、Boyer-Moore算法、Sunday算法

## B树、B+树

B树，B-树就是B树， B-Tree,多叉平衡查找树，

B树的性质（m阶的B树，树的阶数表示一个节点最多能有多少个子节点）：

* 树的每个节点最多有m个字节点（m >= 2），空树除外
* 除根节点外每个节点的关键字数量大于等于ceil(m/2),小于等于m-1个，非跟节点关键字数必须>= 2，(ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2)
* 所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为null对应下图最后一层节点的空格子
* 如果一个非叶节点有N个子节点，则该节点的关键字数等于N-1;
* 所有节点关键字是按递增次序排列，并遵循左小右大原则
* 如果根节点不是叶节点，则其至少有两棵子树。
* 每一个非根的分支节点都有k-1个元素和k个孩子，其中ceil(m/2) <= k <= m，每一个叶子节点都有k-1个元素，其中ceil(m/2) <= k <= m
* 所有叶子节点都位于同一层
* 所有分支几点包含下列信息数据（n,A0,K1,A1,K2,A2,…，Kn,An），其中：Ki(i=1,2，…，n)，为关键字，且Ki<Ki+1，Ai为指向子树根节点的指针，且指针Ai-1所指向子树的关键字均小于Ki,An所指向子树中所有节点的关键字均大于Kn

B+树：

* 有n棵子树的节点中含有n-1个关键字：（此处有争议，B+树到底是与B 树n棵子树有n-1个关键字 保持一致，还是不一致：B树n棵子树的结点中含有n个关键字）
* 所有的叶子节点包含了全部关键字的信息，及指向这些关键字记录的指针，且叶子节点本身以关键字的大小自小而大的顺序连接。

所有非终端节点可以看成是索引部分，节点中仅含有其子树根节点中最大（或最小）关键字。（而B树的非终端节点也包含需要查找的有效信息）

B和B+树主要用在文件系统以及数据库做索引.比如Mysql;

我们知道，机械硬盘耗费时间最多的地方在于寻道地址。在数据结构中，如果我们采用二叉树存数数据，那么频繁的增加和删除数据，会让二叉树退化成一个链表。B树的出现就克服了这个问题，B树的增加和删除节点对于整体的结构来说，改动非常的小，所以十分适合用来作为大数据存储的数据结构。

B+树相比于B树，在文件系统，数据库系统当中，更有优势，原因如下：

* B+树的磁盘读写代价更低

B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说I/O读写次数也就降低了。

* B+树的查询效率更加稳定

由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

* B+树更有利于对数据库的扫描

B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题，而B+树只需要遍历叶子节点就可以解决对全部关键字信息的扫描，所以对于数据库中频繁使用的range query，B+树有着更高的性能。

## Hash算法，几种解决冲突的方式

hash算法：哈希算法可以将任意长度的二进制引用为较短的且固定长度的二进制值，把这个小的二进制值称为哈希值。

https://blog.csdn.net/tanggao1314/article/details/51457585

直接地址法：以数据元素关键字k本身或它的线性函数作为它的哈希地址

数字分析法：假设关键字集合中的每个关键字都是由 s 位数字组成 (u1, u2, …, us)，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。数字分析法是取数据元素关键字中某些取值较均匀的数字位作为哈希地址的方法。即当关键字的位数很多时，可以通过对关键字的各位进行分析，丢掉分布不均匀的位，作为哈希值。它只适合于所有关键字值已知的情况。此法适于能预先估计出全体关键字的每一位上各种数字出现的频度。

折叠法：将关键字分割成若干部分，然后取它们的叠加和为哈希地址。两种叠加处理的方法：移位叠加:将分 割后的几部分低位对齐相加；边界叠加:从一端沿分割界来回折叠，然后对齐相加。

平方取中法：这是一种常用的哈希函数构造方法。这个方法是先取关键字的平方，然后根据可使用空间的大小，选取平方数是中间几位为哈希地址。

减去法：减去法是数据的键值减去一个特定的数值以求得数据存储的位置。

基数转换法：将十进制数X看作其他进制，比如十三进制，再按照十三进制数转换成十进制数，提取其中若干为作为X的哈希值。一般取大于原来基数的数作为转换的基数，并且两个基数应该是互素的。

除留余数法：假设哈希表长为m，p为小于等于m的最大素数，则哈希函数为h（k）=k % p ，其中%为模p取余运算。

随机数法：设定哈希函数为:H(key) = Random(key)其中，Random 为伪随机函数。此法适于：对长度不等的关键字构造哈希函数。

随机乘数法：

亦称为“乘余取整法”。随机乘数法使用一个随机实数f,0≤f<1,乘积f\*k的分数部分在0～1之间，用这个分数部分的值与n（哈希表的长度）相乘，乘积的整数部分就是对应的哈希值，显然这个哈希值落在0～n-1之间。其表达公式为：Hash(k)=「n\*(f\*k%1)」其中“f\*k%1”表示f\*k 的小数部分，即f\*k%1=f\*k-「f\*k」

字符串数值哈希法：

旋转法

旋转法是将数据的键值中进行旋转。旋转法通常并不直接使用在哈希函数上，而是搭配其他哈希函数使用。

解决哈希冲突的几种方法：

* 开放地址法：

所谓的开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入

公式为：fi(key) = (f(key)+di) MOD m (di=1,2,3,……,m-1)

* 再哈希法

再哈希法又叫双哈希法，有多个不同的Hash函数，当发生冲突时，使用第二个，第三个，….，等哈希函数计算地址，直到无冲突。虽然不易发生聚集，但是增加了计算时间。

* 链地址法

链地址法的基本思想是：每个哈希表节点都有一个next指针，多个哈希表节点可以用next指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表连接起来

* 建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表

## 红黑树

红黑树、R-B树：Red-Black Tree，它是一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红（Red）或（Black），红黑树的特性如下：

* 每个节点或是黑色，或是红色。
* 根节点是黑色。
* 每个叶子节点（NIT）是黑色。（注意：这里叶子节点，是指为空（NIT 或NULL）的叶子节点）
* 如果一个节点是红色的，则它的子节点必须是黑色的。
* 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点（从任一节点到其每个叶子节点的所有路径都包含相同数目的黑色节点）

注意：特性3中叶子节点，是指为空（NIT或NULL）的节点

特性5确保没有一条路径会比其他路径长出两倍。红黑树是相对接近平衡的二叉树

红黑树的应用：主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。

例如，Java集合中的**TreeSet和TreeMap**，C++ STL中的set、map，以及Linux虚拟内存的管理，都是通过红黑树去实现的。

## 常用排序算法

手写常用排序算法，尤其是堆排序

插入排序：直接插入排序、希尔排序

选择排序：简单选择排序、堆排序

交换排序：冒泡排序、快速排序

归并排序

排序算法

冒泡排序、插入排序、选择排序、归并排序、希尔排序、堆排序、快速排序

|  |  |  |  |
| --- | --- | --- | --- |
|  | 时间复杂度 | 空间复杂度 | 稳定性 |
| 冒泡排序 | O(n2) | O(1) | √ |
| 插入排序 | O(n2) | O(1) | √ |
| 选择排序 | O(n2) | O(1) | × |
| 希尔排序 | **O(nm)** | O(1) | × |
| 归并排序 | O(nlogn) | O(n) | √ |
| 快速排序 | O(nlogn) | O(logn)~O(n) | × |
| 堆排序 | O(nlogn) | O(1) | × |

**1 <= m < 2**

**快速排序：空间复杂度O(logn)~O(n)**

**真正消耗空间的就是递归调用了，因为每次递归就要保持一些数据；**

**最优的情况下空间复杂度为：O(logn) ；每一次都平分数组的情况**

**最差的情况下空间复杂度为：O( n ) ；退化为冒泡排序的情况**

* **冒泡排序**：
  + S1:从待排序序列的起始位置开始，从前往后一次比较各个位置与其后一个位置的大小，并执行s2
  + S2:如果当前位置的值大于其后一个位置的值，就把它俩的值交换（完成一趟比较后，序列的最后一个位置的值就是该序列的最大值）
  + 将序列的最后一个元素从待排序序列中移除。执行S1，直到待排序序列长度为1
* **选择排序：**
* S1:从0~n-1位置中选择最小的数字与位置0交换
* S2:从1~n-1位置中选择最小的数字与位置1交换
* ……，
* Sn-1:从n-2~n-1位置中选择最小的数字和位置n-2交换
* 插入排序
  + 将待排序序列分为有序部分和无序部分，初始有序部分为第1个元素
  + 每次处理无序数列中的第一个元素，将其和有序序列的元素从后往前比较，找出插入位置，将元素插入到合适位置，是有序序列仍然有序
  + 无序序列中元素个数为0是算法结束
* 归并排序
  + 将待排序序列的每一个元素看成是长度为1的有序区间，将相邻有序区间合并，得到最大长度为2的有序区间
  + 再将相邻有序区间进行合并，得到长度为4的有序区间依次进行下去，4合8,8合16，直到将待排序序列合并成一个有序区间
* 快速排序

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序

* + 从待排序序列中挑选一个元素作为基准（可选择第一个）
  + 重新排序序列，使小于等于基准的数放在基准的前面，大于基准元素的放在基准的后边
  + 对基准元素前后两个部分分别递归调用此过程，使得整个数组有序
* 堆排序
  + 将待排序序列构建成一个大小为n的大根堆，此时堆顶元素为待排序列最大元素。将堆顶元素和堆的最后一个数字进行交换
  + 将最大值脱离堆，放在数组最后的位置，作为有序部分，保留下来
  + 将n-1大小的堆，从堆顶位置开始调整，调整为大根堆。执行上述步骤，直到堆中只剩下一个元素。
* 希尔排序

改进的插入排序。

将待排序序列分为若干子序列，子序列不是逐段分割的，而是相隔特定增量的子序列，对各个子序列进行直接插入排序；然后在选择一个更小的增量，将序列分割为子序列进行排序……最后选择增量为1，即使用直接插入排序，使最终数组有序。

增量的选择一种方式，初始选择为数组长度的一半n/2，每次增量为原来的一半。

* **直接插入排序：**

public static void insertionSort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int j = i - 1;

int temp = arr[i];

for (; j >= 0; j--) {

// 将大于 temp 的往后移动一位

if (arr[j] > temp) {

arr[j + 1] = arr[j];

} else {

break;

}

}

arr[j + 1] = temp;

}

}

* **希尔排序：**

public static void shellSort(int[] arr) {

int j = 0,temp = 0,n = arr.length;

for(int d = n/2;d > 0; d /= 2){

for (int i = d; i < n; i++) {

temp = arr[i];

for (j = i - d; j >= 0; j = j - d) {

if (temp < arr[j]) {

arr[j + d] = arr[j];

} else {

break;

}

}

arr[j + d] = temp;

}

}

}

* **直接选择排序**

public static void selectionSort(int[] arr) {

int index, temp;

for (int i = 0; i < arr.length - 1; i++) {

index = i;

for (int j = i + 1; j < arr.length; j++) {

if (arr[j] < arr[index]) {

index = j;

}

}

if (index != i) {

temp = arr[i];

arr[i] = arr[index];

arr[index] = temp;

}

}

}

* **堆排序：**

public static void heapSort(int[] arr) {

int temp;

for (int i = arr.length / 2 - 1; i >= 0; i--) {

heapAdjust(arr, i, arr.length);

}

for (int i = arr.length - 1; i > 0; i--) {

temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapAdjust(arr, 0, i);

}

}

public static void heapAdjust(int[] arr, int parent, int length) {

int temp = arr[parent];

int child = parent \* 2 + 1;

while (child < length) {

if (child + 1 < length && arr[child + 1] > arr[child]) {

child++;

}

if (temp >= arr[child]) {

break;

}

arr[parent] = arr[child];

parent = child;

child = parent \* 2 + 1;

}

arr[parent] = temp;

}

* **冒泡排序：**

public static void mySort(int[] arr) {

int temp;

for (int i = 0; i < arr.length - 1; i++) {

for (int j = 0; j < arr.length - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

temp = arr[j + 1];

arr[j + 1] = arr[j];

arr[j] = temp;

}

}

}

}

* 快速排序

public static void quickSort(int[] arr) {

quickSort(arr, 0, arr.length - 1);

}

public static void quickSort(int[] arr, int left, int right) {

qSort(arr, 0, arr.length - 1);

}

private static void qSort(int[] arr, int left, int right) {

if (left >= right) {

return;

}

int i, j, temp;

i = left;

j = right;

temp = arr[i]; // 以第一个数为基准

while (i < j) {

//找到右边第一个比基准小的数

while (i < j && arr[j] >= temp) {

j--;

}

if (i < j) {

arr[i++] = arr[j];

}

//找到左边的一个比基准大的数

while (i < j && arr[i] <= temp) {

i++;

}

if (i < j) {

arr[j--] = arr[i];

}

}

arr[i] = temp;

qSort(arr, left, i - 1);

qSort(arr, i + 1, right);

}

* **归并排序**

public static void mySort(int[] arr) {

mergeSort(arr, 0, arr.length - 1);

}

public static void mergeSort(int[] arr, int left, int right) {

if (left >= right) {

return;

}

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

private static void merge(int[] arr, int left, int mid, int right) {

int l = left;

int r = mid + 1;

int j = 0;

int[] temp = new int[right - left + 1];

while (l <= mid && r <= right) {

if (arr[l] <= arr[r]) {

temp[j++] = arr[l++];

} else {

temp[j++] = arr[r++];

}

}

while (l <= mid) {

temp[j++] = arr[l++];

}

while (r <= right) {

temp[j++] = arr[r++];

}

int i = 0;

while (i < j) {

arr[i + left] = temp[i++];

}

}

# 其它

## Java 关键词switch

switch接收的参数类型：

基本参数类型 byte short int char 以及它们的包装类

引用类型String(JDK 1.7 之后，类型也可以是String了)

还需注意的是case后面只能是常量，可以是运算表达式，但一定要符合正确的类型。不能是变量，即便变量在之前进行了赋值，JVM依然会报错。

## 多线程中ArrayList进行add操作，每个线程进行10次add，假设就有两个线程，最后list的size是多少？为什么会出现这样的情况？

* 没发生线程安全问题，最终list的size是20
* size可能小于20：

1、刚开始ArrayList 大小为0，首次扩充大小为10 (通过size获得的是实际存储数据的大小)

2、线程 A 添加元素，此时size 为0。在准备执行elementData[0] = e; 之后执行size++时 CPU 调度线程A暂停

3、线程B 添加元素，此时size 为0，执行了elementData[0] = e; 之后执行size++，此时size为1.

4、此时线程A继续执行size++ ，此时size为1

所以线程B添加的值会被线程A添加的值覆盖

* 发生数组越界异常 在需要扩容时（10,15）

1、当集合中已经添加了14个元素时，线程A率先进入add()方法，在执行ensureCapacityInternal(size + 1)时，发现还可以添加一个元素，故数组没有扩容，但随后线程A被阻塞在此处。

2、接着线程B进入add()方法，执行ensureCapacityInternal(size + 1)，由于线程A并没有添加元素，故size依然为14，依然不需要扩容，所以该线程就开始添加元素，使得size++，变为15，数组已经满了。

3、此时线程A运行，开始执行elementData[size++] = e;，它要在集合中添加第16个元素，而数组容量只有15个，所以就发生了数组下标越界异常！