@[toc](目录)

# 基础

## JDK和JRE

JDK：Java Development Kit的简称，java开发工具包，提供了java的开发环境和运行环境。

JRE：JavaRuntimeEnvironment的简称，java运行环境，为java的运行提供了所需环境。

具体来说JDK其实包含了JRE，同时还包含了编译java源码的编译器javac，还包含了很多java程序调试和分析的工具。简单来说：如果你需要运行java程序，只需安装JRE就可以了，如果你需要编写java程序，需要安装JDK。

## equals和==

基本类型（byte、boolean、char、short、int、long、float、double）都是用\=\=判断相等性。对象引用的话，\=\=判断引用所指的对象是否是同一个。equals是Object的成员函数，有些类会覆盖（override）这个方法，用于判断对象的值的等价性。例如String类，两个引用所指向的String都是”abc”，但可能出现他们实际对应的对象并不是同一个（和jvm实现方式有关），因此用==判断他们可能不相等，但用equals判断一定是相等的。

```java

class StringTest{

public void test(){

String a= "123";

String b = "123";

String c = new String("123");

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

System.out.println(a == c); //false

System.out.println(a.equals(c)); //true

}

}

```

## 字符串

String、StringBuffer、StringBuilder：

String声明的是不可变的对象，每次操作都会生成新的String对象，然后将指针指向新的String对象，而StringBuffer、StringBuilder可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用String。

StringBuffer和StringBuilder最大在于，StringBuffer是线程安全的，而StringBuilder是非线程安全的，但StringBuilder的性能却高于StringBuffer，所以在单线程环境下推荐使用StringBuilder，多线程环境下推荐使用StringBuffer。

```java

//jvm会将其分配到常量池中

String str="i"

//jvm会将其分配到堆内存中

String strt = new String("i")

```

String类的常用方法

函数名 | 作用

-------- | -----

indexOf()|返回指定字符的索引。

charAt()|返回指定索引处的字符。

replace()|字符串替换。

trim()|去除字符串两端空白。

split()|分割字符串，返回一个分割后的字符串数组。

getBytes()|返回字符串的byte类型数组。

length()|返回字符串长度。

toLowerCase()|将字符串转成小写字母。

toUpperCase()|将字符串转成大写字符。

substring()|截取字符串。

equals()|字符串比较。

## 重写和重载

override（重写）

1. 存在于父类和子类之间。

2. 方法名、参数、返回值相同。

3. 子类方法不能缩小父类方法的访问权限。

4. 子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

5. 方法被定义为final不能被重写。

overload（重载）

1. 参数类型、个数、顺序至少有一个不相同。

2. 不能重载只有返回值不同的方法名。

3. 存在于父类和子类、同类中。

## 抽象类

### 抽象类和普通类

1. 普通类不能包含抽象方法，抽象类可以包含抽象方法。

2. 抽象类不能直接实例化，普通类可以直接实例化。

3. 抽象类能不能使用final修饰。定义抽象类就是让其他类继承的，如果定义为final该类就不能被继承。

### 抽象类和接口

1. 接口是公开的，里面不能有私有的方法或变量，用于让别人使用的，而抽象类是可以有私有方法、私有变量、构造函数。

2. 实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现。

3. 接口可以实现多重继承，多个接口可以通过一个类实现；而一个类只能继承一个抽象超类，但可以通过继承多个接口实现多重继承。

4. 访问修饰符：接口中的方法默认使用public修饰；抽象类中的方法可以是任意访问修饰符。

5. 接口还有标识（里面没有任何方法，如Remote接口）和数据共享（里面的变量全是常量）的作用。

## 代理模式

动态代理主要通过反射实现。反射在框架中常用，Java反射的主要功能：查找对象的类；查找类的数据成员、方法、构造器、超类；构造类的对象；调用方法；实现动态代理。

### 静态代理

代理模式实现如下

```java

interface IUserDao {

void save();

}

class UserDao implements IUserDao {

@Override

public void save() {

System.out.println("save data");

}

}

class UserDaoProxy implements IUserDao {

private IUserDao iUserDao;

public UserDaoProxy(IUserDao iUserDao) {

this.iUserDao = iUserDao;

}

@Override

public void save() {

System.out.println("transactions start");

this.iUserDao.save();

System.out.println("transactions end");

}

}

class StaticUserProxy {

public void testStaticProxy() {

IUserDao iUserDao = new UserDao();

UserDaoProxy userDaoProxy = new UserDaoProxy(iUserDao);

userDaoProxy.save();

}

}

```

### 动态代理

定义：当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

应用：Spring的AOP加事务、权限判断、打印日志

实现：JDK动态代理、CGLIB代理

JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理不需要实现接口的类，就可以使用CGLIB实现。

CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP，为他们提供方法的interception拦截。CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。CGLIB代理无需实现接口，通过生成类字节码实现代理，比反射稍快，不存在性能问题，但CGLIB会继承目标对象，需要重写方法，所以目标对象不能为final类。

\*\*CGLIB代理与JDK动态代理区别：\*\*

1. 使用JDK动态代理的对象必须实现一个或多个接口；使用CGLIB代理的对象则无需实现接口，达到代理类无侵入，直接代理类。

2. 使用CGLIB需要引入CGLIB的jar包。

```java

//JDK动态代理和CGLIB代理实现

interface IUserDao {

void save();

}

class UserDao implements IUserDao {

@Override

public void save() {

System.out.println("save data");

}

}

//JDK Dynamic Proxy

class ProxyFactory {

private Object target;

public ProxyFactory(Object target) {

this.target = target;

}

public Object getProxyInstance() {

return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(),

new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("transactions start");

Object returnValue = method.invoke(target, args);

System.out.println("transactions end");

return null;

}

});

}

}

class UserDaoCGLIB {

void save(){

System.out.println("save data");

}

}

//CGLIB Dynamic Proxy

class CGLIBProxyFactory implements MethodInterceptor {

private Object target;

public CGLIBProxyFactory(Object target) {

this.target = target;

}

public Object getProxyInstance() {

Enhancer en = new Enhancer();

en.setSuperclass(target.getClass());

en.setCallback(this);

return en.create();

}

@Override

public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {

System.out.println("transactions start");

Object returnValue = method.invoke(target, args);

System.out.println("transactions end");

return null;

}

}

class DynamicProxy {

public void jdkWay() {

IUserDao iUserDao = new UserDao();

ProxyFactory proxyFactory = new ProxyFactory(iUserDao);

IUserDao proxy = (IUserDao) proxyFactory.getProxyInstance();

proxy.save();

}

public void cglibWay() {

UserDaoCGLIB userDao = new UserDaoCGLIB();

CGLIBProxyFactory proxyFactory = new CGLIBProxyFactory(userDao);

UserDaoCGLIB proxy = (UserDaoCGLIB) proxyFactory.getProxyInstance();

proxy.save();

}

}

```

### 静态代理与动态代理

静态代理在编译时就已经实现，编译完成后代理类是一个实际的class文件。

动态代理是在运行时动态生成的，即编译完成后没有实际的class文件，而是在运行时动态生成类字节码，并加载到JVM中。相对于静态代理，动态代理消耗系统性能，但更灵活。

参考链接: [Java三种代理模式：静态代理、动态代理和cglib代理](https://segmentfault.com/a/1190000011291179).

## 序列化与反序列化

\*\*序列化场景：\*\*

1. 把的内存中的对象保存到一个文件中或者数据库中时候；

2. 用套接字在网络上传送对象的时候，需要将对象序列化；

3. 当你想通过RMI传输对象的时候；

## 对象拷贝

想对一个对象进行处理，又想保留原有的数据进行接下来的操作，就需要使用对象拷贝了。

\*\*实现方式：\*\*

1、实现Cloneable接口并重写Object类中的clone()方法；

2、实现Serializable接口，通过对象的序列化和反序列化实现克隆。

基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

```java

//对象拷贝实现

class Car implements Serializable {

private static final long serialVersionUID = -5713945027627603702L;

private String brand; // 品牌

private int maxSpeed; // 最高时速

public Car(String brand, int maxSpeed) {

this.brand = brand;

this.maxSpeed = maxSpeed;

}

public String getBrand() {

return brand;

}

public void setBrand(String brand) {

this.brand = brand;

}

public int getMaxSpeed() {

return maxSpeed;

}

public void setMaxSpeed(int maxSpeed) {

this.maxSpeed = maxSpeed;

}

@Override

public String toString() {

return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";

}

}

class Person implements Serializable {

private static final long serialVersionUID = -9102017020286042305L;

private String name; // 姓名

private int age; // 年龄

private Car car; // 座驾

public Person(String name, int age, Car car) {

this.name = name;

this.age = age;

this.car = car;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Car getCar() {

return car;

}

public void setCar(Car car) {

this.car = car;

}

@Override

public String toString() {

return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";

}

}

class CloneTest {

public static void main(String[] args) {

try {

Person p1 = new Person("郭靖", 33, new Car("Benz", 300));

// deepy copy

Person p2 = MyUtil.clone(p1);

p2.getCar().setBrand("BYD");

System.out.println(p1);

} catch (Exception e) {

e.printStackTrace();

}

}

}

class MyUtil {

private MyUtil() {

throw new AssertionError();

}

@SuppressWarnings("unchecked")

public static <T extends Serializable> T clone(T obj) throws Exception {

// key code

ByteArrayOutputStream bout = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(bout);

oos.writeObject(obj);

ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());

ObjectInputStream ois = new ObjectInputStream(bin);

return (T) ois.readObject();

// 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义。 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放

}

}

```

\*\*深拷贝和浅拷贝：\*\*

浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝，如赋值引用。深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse()和JSON.stringify()）

## 常见的异常

异常类名|备注

-------- | -----

NullPointerException|当应用程序试图访问空对象时，则抛出该异常。

SQLException|提供关于数据库访问错误或其他错误信息的异常。

IndexOutOfBoundsException|指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

NumberFormatException|当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。

FileNotFoundException|当试图打开指定路径名表示的文件失败时，抛出此异常。

IOException|当发生某种I/O异常时，抛出此异常。此类是失败或中断的I/O操作生成的异常的通用类。

ClassCastException|当试图将对象强制转换为不是实例的子类时，抛出该异常。

ArrayStoreException|试图将错误类型的对象存储到一个对象数组时抛出的异常。

IllegalArgumentException|抛出的异常表明向方法传递了一个不合法或不正确的参数。

ArithmeticException|当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。

NegativeArraySizeException|如果应用程序试图创建大小为负的数组，则抛出该异常。

NoSuchMethodException|无法找到某一特定方法时，抛出该异常。

SecurityException|由安全管理器抛出的异常，指示存在安全侵犯。

UnsupportedOperationException|当不支持请求的操作时，抛出该异常。

RuntimeExceptionRuntimeException|是那些可能在Java虚拟机正常运行期间抛出的异常的超类。

## JDBC

jdbc操作数据库流程如下。

```mermaid

flowchat

st=>start: 开始

e=>end: 结束

op=>operation: 加载JDBC驱动程序

op2=>operation: 创建数据库的连接

op3=>operation: 创建Statement实例

op4=>operation: 通过Statement执行SQL语句

op5=>operation: 处理结果

op6=>operation: 释放资源

st->op->op2->op3->op4->op5->op6->e

```

1、加载JDBC驱动程序

在连接数据库之前，首先要加载想要连接的数据库的驱动到JVM。成功加载后，会将Driver类的实例注册到DriverManager类中。

2、创建数据库的连接

要连接数据库，需要向java.sql.DriverManager请求并获得Connection对象，该对象就代表一个数据库的连接。

3、创建一个Statement实例，Statement实例分为以下3种类型：

- 通过Statement实例实现执行静态SQL语句；

- 通过PreparedStatement实例实现执行动态SQL语句（底层具体啥原理？？？占位符？），可以防止sql注入；

- 通过CallableStatement实例实现执行数据库存储过程。

以上的sql注入和数据库存储过程概念详见链接: [Mysql基础](https://blog.csdn.net/wxd1234567890/article/details/111160003)；

4、执行SQL语句，Statement接口提供了三种执行SQL语句的方法：

- executeQuery返回一个结果集（ResultSet）对象；

- executeUpdate（用于执行INSERT、UPDATE或DELETE语句以及DDL数据库定义语句包括创建表、修改表等）；

- execute用于执行返回多个结果集、多个更新计数或二者组合的语句。

5、处理结果

6、释放JDBC资源，关闭顺序和声明顺序相反：关闭记录集-->关闭声明-->关闭连接对象

```java

class JDBCTest {

public void test() {

//加载MySql的驱动类

try {

Class.forName("com.mysql.jdbc.Driver");

} catch (ClassNotFoundException ex) {

System.out.println("找不到驱动程序类，加载驱动失败！");

ex.printStackTrace();

}

Connection con = null;

Statement stmt = null;

ResultSet rs = null;

try {

String url = "jdbc:mysql://localhost:3306/test?useSSL=false&characterEncoding=utf8";

String username = "root";

String password = "root";

//创建数据库的连接

con = DriverManager.getConnection(url, username, password);

//创建Statement

stmt = con.createStatement();

String pstmtSql = "SELECT \* FROM user WHERE login=? AND password=?";

PreparedStatement pstmt = con.prepareStatement(pstmtSql);

pstmt.setObject(1, "name");

pstmt.setObject(2, "password");

CallableStatement cstmt = con.prepareCall("");

//获取结果集，处理结果

String sqlQueryString = "SELECT \* FROM tablename";

rs = stmt.executeQuery(sqlQueryString);

while (rs.next()) {

String field1 = rs.getString("field1");

System.out.println(field1);

}

String sqlInsertString = "INSERT INTO tablename (field1, field2) VALUES('1', '2')";

int result = stmt.executeUpdate(sqlInsertString);

if (result == 0) {

System.out.println("insert data success");

}

} catch (SQLException ex) {

System.out.println("数据库操作失败！");

ex.printStackTrace();

} finally {

try {

//释放资源，关闭记录集、声明、连接对象

if (rs != null) {

rs.close();

}

if (stmt != null) {

stmt.close();

}

if (con != null) {

con.close();

}

} catch (SQLException ex) {

ex.printStackTrace();

}

}

}

}

```

# 集合

jdk集合的关系拓扑图如下。

![集合接口类的关系图](https://img-blog.csdnimg.cn/20201214141116959.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3d4ZDEyMzQ1Njc4OTA=,size\_16,color\_FFFFFF,t\_70)

## Collection和Collections

java.util.Collection是一个集合接口，集合类的一个顶级父接口。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

## List Set Map

List、Set、Map的比较如下图所示。

![。。。](https://img-blog.csdnimg.cn/20201214141533386.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3d4ZDEyMzQ1Njc4OTA=,size\_16,color\_FFFFFF,t\_70)

## Arraylist与LinkedList

ArrrayList底层的数据结构是数组，使用下标访问一个元素，支持随机访问，而LinkedList的底层数据结构是双向循环链表，不支持随机访问。ArrayList的时间复杂度是O(1)，而LinkedList是O(n)。

\*\*Arraylist：\*\*

优点：ArrayList是实现了基于动态数组的数据结构,因为\*\*地址连续\*\*，一旦数据存储好了，查询操作效率会比较高，在内存里是连着放的。

缺点：因为地址连续，ArrayList要移动数据，所以插入和删除操作效率比较低。

\*\*LinkedList：\*\*

优点：LinkedList基于链表的数据结构,地址是任意的，所以在开辟内存空间的时候不需要等一个连续的地址，对于新增和删除操作add和remove，LinedList比较占优势。LinkedList适用于要头尾操作或插入指定位置的场景

缺点：因为LinkedList要移动指针,所以查询操作性能比较低。

\*\*适用场景分析：\*\*

当需要对数据经常查询的场景下选用ArrayList，当需要对数据进行经常插入删除的场景采用LinkedList。

## ArrayList与Vector

ArrayList和Vector底层都是用数组实现的。

\*\*ArrayList扩容机制：\*\*

在JDK1.8中，如果通过无参构造的话，初始数组容量为0，当真正对数组进行添加时（即添加第一个元素时），才真正分配容量，默认分配容量为10；当容量不足时（容量为size，添加第size+1个元素时），先按照1.5倍的比例扩容能否满足最低容量要求，若能，则以1.5倍扩容，否则以最低容量要求进行扩容。当ArrayList扩容时，根据扩容的大小分配一块新的连续的内存空间，将已经有数组的数据复制到新的存储空间中，并插入新的数据。例如，不指定大小的ArrayList添加第11个元素时，进行扩容到10\*1.5=15，添加第16个元素时，扩容到对应的大小。

\*\*ArrayList与Vector区别：\*\*

- Vector是多线程安全的，而ArrayList不是线程安全的（线程安全就是说多线程修改共享数据时，需求保持同一时刻只有一个修改共享数据，不会产生不确定的结果）。

- 两个都是采用的线性连续的内存空间存储元素，但是当空间不足的时候，两个类的增加方式是不同。ArrayList在内存不够时默认是扩展50% + 1个，Vector是默认扩展1倍或旧容量加增长因子大小。Vector可以设置增长因子，而ArrayList不可以。

- Vector类的源码中的方法很多有synchronized进行修饰，这样就导致了Vector在效率上无法与ArrayList相比，效率较低；

\*\*适用场景分析：\*\*

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。如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用Vector有一定的优势。

## HashMap原理

### HashMap实现原理

HashMap概述：HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap的数据结构：HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

当我们往Hashmap中put元素时,首先根据key的hashcode重新计算hash值,根据hash值（例如假设hash算法为通过hash值对数组的大小取余）得到这个元素在\*\*数组的位置及下标\*\*，如果该数组在该位置上已经存放了其他元素，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头,最先加入的放入链尾。如果数组中该位置没有元素,就直接将该元素放到数组的该位置上。Java8中对HashMap的实现做了优化，当链表中的节点数据超过八个之后，该链表会转为红黑树来提高查询效率,从原来的O(n)到O(logn)。其中红黑树详见链接: [高阶数据结构](https://blog.csdn.net/wxd1234567890/article/details/111170599).

### HashMap扩容机制

\*\*哈希冲突：\*\*

key通过哈希算法得到的哈希值相同，导致数据存储在同一个节点的链表上，当冲突太多链表过长，查询效率就会急剧下降。解决方法需要优化哈希算法，待研究？？？

\*\*扩容应用实例：\*\*

当键值对的实际大小size大于table的实际大小\*负载因子时需要进行扩容。

假设了hash算法就是简单的用key的hash值取余数组的长度，其中的哈希桶数组table的原始size=2，key = 3、7、5的put顺序依次为 5、7、3。在mod取余2以后都冲突在table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小size大于table的实际大小就进行扩容。接下来的三个步骤是哈希桶数组resize扩容成4（后续如果不够，继续扩容8,16，，，），然后所有的Node重新哈希映射的过程。

![在这里插入图片描述](https://img-blog.csdnimg.cn/2020121415443077.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3d4ZDEyMzQ1Njc4OTA=,size\_16,color\_FFFFFF,t\_70)

扩容是一个特别耗性能的操作，所以在使用HashMap的时候，需要估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容影响性能。

\*\*HashMap的长度必须是2的幂原因：\*\*

为了提高性能，待研究？？？

## HashSet实现原理

HashSet底层由HashMap实现，HashSet的值存放于HashMap的key上，HashMap的value统一为PRESENT为空值。

## ConcurrentHashMap实现原理

\*\*ConcurrentHashMap与HashTable比较：\*\*

HashTable里使用的是synchronized关键字，这其实是对整个Map对象加锁，锁住的都是对象整体，当Hashtable的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。ConcurrentHashMap算是对上述问题的优化。

ConcurrentHashMap作为一种线程安全且高效的哈希表的解决方案，尤其其中的"分段锁"的方案，相比HashTable的全表锁在性能上的提升非常之大。ConcurrentHashMap引入了分割Segment技术，把一个大的Map拆分成N个小的HashTable，在put方法中，会根据hash值来决定具体存放进哪个Segment，如果查看Segment的put操作，我们会发现内部使用的同步机制是基于lock操作的，这样就可以对Map的一部分Segment进行上锁，这样影响的只是将要放入同一个Segment的元素的put操作，保证同步的时候，锁住的不是整个Map，相对于HashTable提高了多线程环境下的性能，因此HashTable已经被淘汰了。如下图所示ConcurrentHashMap的分段锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214150637413.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3d4ZDEyMzQ1Njc4OTA=,size\_16,color\_FFFFFF,t\_70)

详见链接: [ConcurrentHashMap实现原理及源码分析](https://www.cnblogs.com/chengxiao/p/6842045.html).

## HashMap和Hashtable

1. hashMap去掉了HashTable的contains方法，但是加上了containsValue（）和containsKey（）方法。

2. hashTable同步的，线程安全的，而HashMap是非同步的，效率上比hashTable要高。

3. hashMap允许空键值，而hashTable不允许。

## HashMap和ConcurrentHashMap

ConcurrentHashMap是线程安全的HashMap的实现。

1. ConcurrentHashMap对整个数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的syn关键字锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。

2. HashMap的键值对都允许有null，但是ConCurrentHashMap都不允许。

## HashMap和TreeMap

- TreeMap<K,V>的Key值是要求实现java.lang.Comparable，所以迭代的时候TreeMap默认是按照Key值升序排序的；TreeMap的实现是基于红黑树结构，适用于按升序或自定义顺序遍历键key。

- HashMap<K,V>的Key值实现散列hashCode()，分布是散列的、均匀的，不支持排序；数据结构主要是数组，链表或红黑树。适用于在Map中插入、删除和定位元素。

- 适用场景分析：需要按照key排序使用TreeMap。除此之外，由于HashMap有更好的性能，所以大多不需要排序的时候我们会使用HashMap。

# 多线程

## 创建线程的方式

1. 继承Thread类创建线程类

定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。

创建Thread子类的实例，即创建了线程对象。

调用线程对象的start()方法来启动该线程。

2. 通过Runnable接口创建线程类

定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。

调用线程对象的start()方法来启动该线程。

3. 通过Callable和Future创建线程

创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。

创建Callable实现类的实例，使用Future类来包装Callable对象，该Future对象封装了该Callable对象的call()方法的返回值。

```java

// create thread 3 ways

class Thread1 extends Thread {

@Override

public void run() {

System.out.println("Thread1 start");

}

}

class Thread2 implements Runnable {

@Override

public void run() {

System.out.println("Thread2 start");

}

}

class Thread3 implements Callable<String> {

@Override

public String call() throws Exception {

System.out.println("Thread3 start");

return "return value";

}

}

class ThreadTest {

public void test() {

Thread thread1 = new Thread1();

thread1.start();

Thread thread2 = new Thread(new Thread2());

thread2.start();

try {

ExecutorService executorService = Executors.newFixedThreadPool(4);

Callable<String> task = new Thread3();

Future<String> future = executorService.submit(task);

//may block

String result = future.get();

System.out.println(String.format("thread3 result %s", result));

} catch (Exception ex) {

}

}

}

```

\*\*runnable和callable:\*\*

- Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；

- Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步任务的执行结果。

\*\*创建线程方式的对比：\*\*

1. 采用实现Runnable、Callable接口的方式创见多线程

优势：线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个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：编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

2. 使用继承Thread类的方式创建多线程

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

劣势：线程类已经继承了Thread类，所以不能再继承其他父类。

## 线程的生命周期

线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。

- 创建状态。在生成线程对象，并没有调用该对象的start方法，这是线程处于创建状态。

- 就绪状态。当调用了线程对象的start方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。线程正在就绪队列中排队等候得到CPU资源。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

- 运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行run函数当中的代码。

- 阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep，wait，suspend（被另一个线程所阻塞）等方法都可以导致线程阻塞。

- 死亡状态。如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪。

## 守护线程

守护线程（Daemon Thread）是指为其他线程服务的线程。在JVM中，所有非守护线程都执行完毕后，无论有没有守护线程，虚拟机都会自动退出。因此，JVM退出时，不必关心守护线程是否已结束。

在守护线程中，编写代码要注意：守护线程不能持有任何需要关闭的资源，例如打开文件等，因为虚拟机退出时，守护线程没有任何机会来关闭文件，这会导致数据丢失。创建守护线程如下代码。

```java

Thread t = new MyThread();

t.setDaemon(true);

t.start();

```

## 线程相关方法

1. sleep()方法

在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。让其他线程有机会继续执行，\*\*但它并不释放对象锁。也就是如果有synchronized同步块，其他线程仍然不能访问共享数据\*\*。比如有两个线程同时执行(没有synchronized同步块)，一个线程优先级为MAX\_PRIORITY，另一个为MIN\_PRIORITY，如果没有Sleep()方法，只有高优先级的线程执行完成后，低优先级的线程才能执行；但当高优先级的线程sleep(5000)后，低优先级就有机会执行了。总之，sleep()可以使低优先级的线程得到执行的机会，当然也可以让同优先级、高优先级的线程有执行的机会。sleep方法是线程类Thread的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。sleep不能改变对象的锁，当一个synchronized同步块中调用了sleep() 方法，线程虽然进入休眠，但是对象的锁没有被释放，其他线程依然无法访问这个对象。

2. wait()方法

wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问同步块，可以通过notify，notifyAll方法来唤醒等待的线程。

3. yield()方法

yield()方法和sleep()方法类似，也不会释放对象锁，区别在于，它没有入参，即yield()方法只是使当前线程重新回到可执行状态（就绪状态？），所以执行yield()的线程有可能在进入到可执行状态后马上又被执行；另外yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和sleep()方法不同。

4. join()方法

如下代码，当main线程对线程对象t调用join()方法时，主线程将等待变量t表示的线程运行结束，即join就是指等待该线程结束，然后才继续往下执行自身线程。所以，上述代码打印顺序可以肯定是main线程先打印start，t线程执行打印，main线程最后再打印end。通过对另一个线程对象调用join()方法可以等待其执行结束，对已经运行结束的线程调用join()方法会立刻返回。

```java

class ThreadTest {

public void threadFunctionTest(){

Thread t = new Thread(()->{

System.out.println("in thread t...");

});

try{

System.out.println("Main thread start");

t.start();

t.join();

System.out.println("Main thread end");

}catch (InterruptedException ex){

}

}

}

```

5. notify和notifyAll

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争。优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。可以通过wait、notify、notifyAll实现多线程协调调用。如下代码。

```java

class TaskQueue{

Queue<String> queue = new LinkedList<>();

public synchronized void addTask(String s){

this.queue.add(s);

this.notifyAll();

}

public synchronized String getTask() throws InterruptedException{

while (queue.isEmpty()) {

this.wait();

}

return queue.remove();

}

}

```

## ThreadLocal

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程的局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。注意点如下：

- 可以把ThreadLocal看成一个全局Map<Thread, Object>，每个线程获取ThreadLocal变量时，总是使用Thread自身作为key。

- ThreadLocal相当于给每个线程都开辟了一个独立的存储空间，各个线程的ThreadLocal关联的实例互不干扰。

- ThreadLocal一定要在finally中清除，因为当前线程执行完相关代码后，很可能会被重新放入线程池中，如果ThreadLocal没有被清除，该线程执行其他代码时，会把上一次的状态带进去。

- ThreadLocal表示线程的局部变量，它确保每个线程的ThreadLocal变量都是各自独立的；

- ThreadLocal适合在一个线程的处理流程中保持上下文（避免了同一参数在所有方法中传递）；

- 使用ThreadLocal可以使用try ... finally结构，并在finally中清除。

```java

class ThreadLocalTest {

static class Context {

private String name;

private int age;

//...

}

static ThreadLocal<Context> threadLocal = new ThreadLocal<>();

public void process() {

try {

threadLocal.set(new Context());

checkPermission();

doWork();

saveStatus();

sendResponse();

} finally {

threadLocal.remove();

}

}

void checkPermission() {

}

void doWork() {

Context ctx = threadLocal.get();

}

void saveStatus() {

Context ctx = threadLocal.get();

}

void sendResponse() {

}

}

```

## 线程池

### 状态

线程池有5种状态：Running、ShutDown、Stop、Tidying、Terminated。状态转换如下图。

![。。。](https://img-blog.csdnimg.cn/20201214193020407.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3d4ZDEyMzQ1Njc4OTA=,size\_16,color\_FFFFFF,t\_70)

### 分类

- newFixedThreadPool(int nThreads)创建一个固定大小的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

- newCachedThreadPool()创建一个可缓存的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

- newSingleThreadExecutor()这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行。

- newScheduledThreadPool(int corePoolSize)创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer定时器。

### 线程池的实现原理

详见链接: [link](https://tech.meituan.com/2020/04/02/java-pooling-pratice-in-meituan.html).待研究？？？

## 其他待研究？？？

### CountDownLatch原理

### CyclicBarrier原理

### Semaphore原理

### Exchanger原理

## 锁机制

### 线程安全

线程安全是指要控制多个线程对某个共享资源的有序访问或修改，而在这些线程之间没有产生冲突。在Java里，线程安全一般体现在两个方面：

1. 多个thread对同一个java实例的访问（read和modify）不会相互干扰，它主要体现在关键字synchronized。如ArrayList和Vector，HashMap和Hashtable（后者每个方法前都有synchronized关键字）。如果你在interator一个List对象时，其它线程remove一个element，问题就出现了。

2. 每个线程都有自己的字段，而不会在多个线程之间共享。它主要体现在ThreadLocal类。

保证多线程安全：

原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作，（atomic,synchronized）；

可见性：一个线程对主内存的修改可以及时地被其他线程看到，（synchronized,volatile）；

有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before原则）。

### synchronized

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个线程可以执行临界区，同时它还可以保证共享变量的内存可见性。synchronized作用的对象包括：普通同步方法，锁是当前实例对象；静态同步方法，锁是当前类的class对象；同步方法块，锁是括号里面的对象

### synchronized和volatile

- volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主内存中读取，保证一个共享变量被一个线程修改，其他线程可见； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

- volatile仅能使用在变量级别；synchronized则可以使用在变量、方法（常规方法和静态方法）、和类级别的。

- volatile仅能实现变量的修改的可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性。

- volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

- volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。

### synchronized和Lock

- synchronized是java内置关键字，在jvm层面，Lock是个java类；

- synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；

- synchronized会自动释放锁(线程执行完同步代码会释放锁 ；线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；

- 用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁超时，线程可以不用一直等待就结束了；

- synchronized的锁可重入、不可中断、非公平（啥意思？？？），而Lock锁可重入、可判断、可公平（两者皆可）；

- Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

### synchronized和ReentrantLock

1. synchronized是关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock可重入锁比synchronized的扩展性体体现如下：

- ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

- ReentrantLock可以获取各种锁的信息

- ReentrantLock可以灵活地实现多路通知？？？

2. 二者的锁机制其实也是不一样的:ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word。java对象在内存中的存储方式见链接: [JVM基础](https://editor.csdn.net/md/?articleId=111184654).

### 常见的Lock

- ReentrantLock

- ReadWriteLock

- StampedLock

- 。。。代码应用？？？

### atomic原理

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。？？？

### 多线程锁升级

JVM优化synchronized的运行机制，当JVM检测到不同的竞争状态时，就会根据需要自动切换到合适的锁，这种切换就是锁的升级。升级是不可逆的，也就是说只能从低到高，也就是偏向-->轻量级-->重量级，不能够降级

Java中锁升级的最佳实例就是synchronized，synchronized把锁信息存放在对象头的MarkWord中。在早期的jdk版本中，synchronized是一个重量级锁，保证线程的安全但是效率很低。后来对synchronized进行了优化，有了一个锁升级的过程：

无锁态-->偏向锁-->轻量级锁（自旋锁）-->重量级锁

锁升级过程详解：

当给一个对象增加synchronized锁之后，相当于上了一个偏向锁。当有一个线程去请求时，就把这个对象MarkWord的ID改为当前线程指针ID（JavaThread），只允许这一个线程去请求对象。

当有其他线程也去请求时，就把锁升级为轻量级锁。每个线程在自己的线程栈中生成LockRecord，用CAS自旋操作将请求对象MarkWord ID改为自己的LockRecord，成功的线程请求到了该对象，未成功的对象继续自旋。

如果竞争加剧，当有线程自旋超过一定次数时，就将轻量级锁升级为重量级锁，线程挂起，进入等待队列，等待操作系统的调度。

？？？

### 可重入锁

JVM允许同一个线程重复获取同一个锁，这种能被同一个线程反复获取的锁，就叫做可重入锁。

由于Java的线程锁是可重入锁，所以，获取锁的时候，不但要判断是否是第一次获取，还要记录这是第几次获取。每获取一次锁，记录+1，每退出synchronized块，记录-1，减到0的时候，才会真正释放锁。

### 死锁

在获取多个锁的时候，不同线程获取多个不同对象的锁可能导致死锁。对于如下释掉的代码注，线程1和线程2如果分别执行add()和dec()方法时：

1. 开始：线程1：进入add()，获得lockA；线程2：进入dec()，获得lockB。

2. 随后：线程1：已经获取lockA，准备获得lockB，失败，等待中；线程2：已经获得lockB，准备获得lockA，失败，等待中。

此时，\*\*两个线程各自持有不同的锁，然后各自试图获取对方手里的锁，造成了双方无限等待下去，这就是死锁\*\*。

防止死锁：

- 线程获取锁的顺序要一致。优化后的代码正确顺序如下。

- 采用Lock对象锁，获取锁设置了超时时间。

- 。。。

```java

class LockTest {

private Object lockA = new Object();

private Object lockB = new Object();

private int value = 0;

private int another = 0;

public void add(int m) {

synchronized (lockA) { // 获得lockA的锁

this.value += m;

synchronized (lockB) { // 获得lockB的锁

this.another += m;

} // 释放lockB的锁

} // 释放lockA的锁

}

// may dead lock

/\* public void dec(int m) {

synchronized (lockB) { // 获得lockB的锁

this.another -= m;

synchronized (lockA) { // 获得lockA的锁

this.value -= m;

} // 释放lockA的锁

} // 释放lockB的锁

}\*/

//优化后的正确顺序

public void dec(int m) {

synchronized (lockA) {

this.another -= m;

synchronized (lockB) {

this.value -= m;

} // 释放lockA的锁

} // 释放lockB的锁

}

}

```

### 悲观锁与乐观锁

乐观锁悲观锁是一种思想。可以用在很多方面。

比如数据库方面。悲观锁就是forupdate（锁定查询的行）乐观锁就是version字段（比较跟上一次的版本号，如果一样则更新，如果失败则要重复读-比较-写的操作。）

JDK方面：悲观锁就是sync乐观锁就是原子类（内部使用CAS实现）

本质来说，就是悲观锁认为总会有人抢我的。乐观锁就认为，基本没人抢。

？？？

### CAS

#### CAS应用

CAS（Compare and Swap）。比如你要操作一个变量，他的值为A，你希望将他修改为B，这期间不会进行加锁，当你在修改的时候，你发现值仍旧是A，然后将它修改为B，如果此时值被其他线程修改了，变成了C，那么将不会进行值B的写入操作，这就是CAS的核心理论。

？？？

#### CAS乐观锁

乐观锁是一种思想，即认为读多写少，遇到并发写的可能性比较低，所以采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读-比较-写的操作。

CAS是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。CAS顶多算是乐观锁写那一步操作的一种实现方式罢了，不用CAS自己加锁也是可以的。

？？？

#### ABA问题

ABA：如果另一个线程修改V值假设原来是A，先修改成B，再修改回成A，当前线程的CAS操作无法分辨当前V值是否发生过变化。

。。。

各种乐观锁的实现中通常都会用版本戳version来对记录或对象标记，避免并发操作带来的问题，在Java中，AtomicStampedReference<E>也实现了这个作用，它通过包装[E,Integer]的元组来对对象标记版本戳stamp，从而避免ABA问题，

https://www.cnblogs.com/549294286/p/3766717.html

？？？