## 集合

### 集合的类型与各自的特性

|  |
| --- |
| ---|**Collection**: 单列集合  ---|**List**: 有存储顺序, 可重复  ---|**ArrayList**: 数组实现, 查找快, 增删慢  由于是数组实现, 在增和删的时候会牵扯到数组  增容, 以及拷贝元素. 所以慢。数组是可以直接按索引查找, 所以查找时较快  ---|**LinkedList**: 链表实现, 增删快, 查找慢由于链表实现, 增加时只要让前一个元素记住自己就可以, 删除时让前一个元素记住后一个元素, 后一个元素记住前一个元素. 这样的增删效率较高但查询时需要一个一个的遍历, 所以效率较低  ---|**Vector**: 和ArrayList原理相同, 但线程安全, 效率略低  和ArrayList实现方式相同, 但考虑了线程安全问题, 所以效率略低  ---|**Set**: 无存储顺序, 不可重复  ---|**HashSet** 线程不安全，存取速度快。底层是以哈希表实现的。  ---|**TreeSet** 红-黑树的数据结构，默认对元素进行自然排  序（String）。如果在比较的时候两个对象  返回值为0，那么元素重复。  ---| Map: 键值对 键不可重复，键可以重复  ---|**HashMap** 线程不安全，存取速度快。底层是以哈希表实现的.  ---|**TreeMap** 红-黑树的数据结构，默认对元素进行自然排  序（String）。如果在比较的时候两个对象  返回值为0，那么元素重复  ---|**HashTable** 底层也是使用了哈希表 维护的，存取的读取快，存储元素是  无序的。 |

### 遍历集合

#### 遍历集合的几种方式

1. 使用迭代器Iterator的方式。
2. **使用增强for循环的方式。**
3. **如果有下标，则可以使用下标的方式。**

#### 遍历数组

**public** **static** **void** main(String[] args) {

// 遍历数组：

String[] arr = **new** String[] { "xx", "yy", "zz" };

// 1，增强的for循环

**for** (String elt : arr) {

System.*out*.println(elt);

}

// 2，下标的方式

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

System.*out*.println(arr[i]);

}

}

#### 遍历List

**public** **static** **void** main(String[] args) {

// 遍历List：

List<String> list = **new** ArrayList<String>();

list.add("aa");

list.add("bb");

list.add("cc");

// 1，增强的for循环

**for** (String elt : list) {

System.*out*.println(elt);

}

// 2，下标

**for** (**int** i = 0; i < list.size(); i++) {

System.*out*.println(list.get(i));

}

// 3，迭代器

**for** (Iterator<String> iter = list.iterator(); iter.hasNext();) {

String elt = iter.next();

System.*out*.println(elt);

}

}

#### 遍历Set

**public** **static** **void** main(String[] args) {

// 遍历Set：

Set<String> set = **new** HashSet<String>();

set.add("dd");

set.add("ee");

set.add("ff");

// 1，增强的for循环

**for** (String elt : set) {

System.*out*.println(elt);

}

// 2，迭代器

**for**(Iterator<String> iter = set.iterator(); iter.hasNext() ; ){

String elt = iter.next();

System.*out*.println(elt);

}

}

#### 遍历Map

## 泛型(Generic)

**public** **static** **void** main(String[] args) {

// 遍历Map：

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

map.put("aa", "xx");

map.put("bb", "yy");

map.put("cc", "zz");

// 1，增强的for循环（Entry集合）

**for** (Entry<String, String> entry : map.entrySet()) {

System.*out*.println(entry);

}

// 2，增强的for循环（Key集合）

**for**(String key : map.keySet()){

System.*out*.println(key + " = " + map.get(key));

}

// 3，遍历值的集合

**for**(String value : map.values()){

System.*out*.println(value);

}

}

当集合中存储的对象类型不同时，那么会导致程序在运行的时候的转型异常

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  ArrayList arr = **new** ArrayList();  arr.add(**new** Tiger("华南虎"));  arr.add(**new** Tiger("东北虎"));  arr.add(**new** Sheep("喜羊羊"));  System.*out*.println(arr);  Iterator it = arr.iterator();  **while** (it.hasNext()) {  Object next = it.next();  Tiger t = (Tiger) next;  t.eat();  }  }  }  **class** Tiger {  String name;  **public** Tiger() {  }  **public** Tiger(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Tiger@name:" + **this**.name;  }  **public** **void** eat() {  System.*out*.println(**this**.name + "吃羊");  }  }  **class** Sheep {  String name;  **public** Sheep() {  }  **public** Sheep(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Sheep@name:" + **this**.name;  }  **public** **void** eat() {  System.*out*.println(**this**.name + "吃青草");  }  } |

原因 :发现虽然集合可以存储任意对象,但是如果需要使用对象的特有方法,那么就需要类型转换,如果集合中存入的对象不同,可能引发类型转换异常.

|  |
| --- |
| [Tiger@name:华南虎, Tiger@name:东北虎, Sheep@name:喜羊羊]  华南虎吃羊  东北虎吃羊  Exception in thread "main" java.lang.ClassCastException: cn.itcast.gz.map.Sheep cannot be cast to cn.itcast.gz.map.Tiger  at cn.itcast.gz.map.Demo5.main(Demo5.java:17) |

出现问题:

存入的是特定的对象,取出的时候是Object对象,需要强制类型转换,可能诱发类型转换异常.

无法控制存入的是什么类型的对象,取出对象的时候进行强转时可能诱发异常.而且在编译时期无法发现问题.

虽然可以再类型转换的时候通过if语句进行类型检查(instanceof),但是效率较低.(例如吃饭的时候,还需要判断米饭里有没有沙子,吃饭效率低).可以通过给容器加限定的形式规定容器只能存储一种类型的对象.

就像给容器贴标签说明该容器中只能存储什么样类型的对象。

所以在jdk5.0后出现了泛型

泛型应用:

格式

1. 集合类<类类型> 变量名 = new 集合类<类类型>();

|  |
| --- |
| **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  // 使用泛型后,规定该集合只能放羊,老虎就进不来了.  ArrayList<Sheep> arr = **new** ArrayList<Sheep>();  arr.add(**new** Sheep("美羊羊"));  arr.add(**new** Sheep("懒洋洋"));  arr.add(**new** Sheep("喜羊羊"));  // 编译失败  // arr.add(new Tiger("东北虎"));  System.*out*.println(arr);  Iterator<Sheep> it = arr.iterator();  **while** (it.hasNext()) {  // 使用泛型后,不需要强制类型转换了  Sheep next = it.next();  next.eat();  }  }  } |

1. 将运行时的异常提前至编译时发生。

2. 获取元素的时候无需强转类型，就避免了类型转换的异常问题

格式 通过<> 来指定容器中元素的类型.

什么时候使用泛型:当类中操作的引用数据类型不确定的时候,就可以使用泛型类.

JDK5.0之前的Comparable

|  |
| --- |
| **package java.lang;**  **public** **interface** Comparable {  **public** **int** compareTo(Object o);  } |

JDK5.0之后的Comparable

|  |
| --- |
| **package java.lang;**  **public** **interface** Comparable<T> {  **public** **int** compareTo(T o);  } |

这里的<T>表示泛型类型,随后可以传入具体的类型来替换它.

细节一

声明好泛型类型之后,集合中只能存放特定类型元素

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  //创建一个存储字符串的list  ArrayList<String> arr=**new** ArrayList<String>();  arr.add("gz");  arr.add("itcast");  //存储非字符串编译报错.  arr.add(1);  }  } |

细节二:

泛型类型必须是引用类型

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  // 泛型类型必须是引用类型,也就是说集合不能存储基本数据类型  // ArrayList<int> arr2=new ArrayList<int>();  // 使用基本数据类型的包装类  ArrayList<Integer> arr2 = **new** ArrayList<Integer>();      }  } |

细节三: 使用泛型后取出元素不需要类型转换.

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  ArrayList<String> arr = **new** ArrayList<String>();  arr.add("gzitcast");  arr.add("cditcast");  arr.add("bjitcast");  //使用泛型后取出元素不需要类型转换.  String str=arr.get(0);  System.*out*.println();  }  } |

## 泛型方法

需求：写一个函数，调用者传递什么类型的变量，该函数就返回什么类型的变量？

实现一:

由于无法确定具体传递什么类型的数据.那么方法的形参就定义为Object类型.返回值也就是Object类型.但是使用该函数时需要强制类型转换.

|  |
| --- |
| **private** Object getDate(Object obj) {  **return** obj;  } |

当不进行强制类型转换能否写出该功能.?

目前所学的知识无法解决该问题

就需要使用泛型类解决

使用的泛型的自定义来解决以上问题。

泛型： 就是将类型当作变量处理。规范泛型的定义一般是一个大写的任意字母。

|  |
| --- |
| 1. 函数上的泛型定义  当函数中使用了一个不明确的数据类型，那么在函数上就可以进行泛型的定义。  public <泛型的声明> 返回值类型 函数名( 泛型 变量名 ){      } |

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[] arr = { 1, 2, 3, 4, 5 };    new Demo6().*getData*(5);  }  **public** <T> T getData(T data) {  **return** data;  } |

细节：

使用泛型方法前需要进行泛型声明，使用一对尖括号 <泛型>，声明的位置在static后返回值类型前。

当一个类中有多个函数声明了泛型，那么该泛型的声明可以声明在类上。

## 泛型类

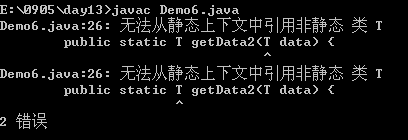
格式

|  |
| --- |
| 2. 类上的泛型声明  修饰符 class 类名<泛型>{    } |

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** Demo6<T> {  **public** **static** **void** main(String[] args) {  // 使用泛型类，创建对象的时候需要指定具体的类型  **new** Demo6<Integer>().getData(5);  }  **public** T getData(T data) {  **return** data;  }  // 反序任意类型数组  **public** **void** reverse(T[] arr) {  **int** start = 0;  **int** end = arr.length - 1;  **for** (**int** i = 0; i < arr.length; i++) {  **if** (start < end) {  T temp = arr[start];  arr[start] = arr[end];  arr[end] = temp;  }  }    } |

在泛型类中定义一个静态方法

|  |
| --- |
| public class Demo6<T> {  public static void main(String[] args) {  System.out.println(getData2(100));  }  public T getData(T data) {  return data;  }  //静态方法  public static T getData2(T data) {  return data;  }  } |



注意：静态方法不可以使用类中定义的泛型

因为类中的泛型需要在对象初始化时指定具体的类型，而静态优先于对象存在。那么类中的静态方法就需要单独进行泛型声明，声明泛型一定要写在static后，返回值类型之前

泛型类细节：

|  |
| --- |
| 1、创建对象的时候要指定泛型的具体类型  2、创建对象时可以不指定泛型的具体类型(和创建集合对象一眼)。默认是Object，例如我们使用集合存储元素的时候没有使用泛型就是那么参数的类型就是Object  3、类上面声明的泛型只能应用于非静态成员函数，如果静态函数需要使用泛型，那么  需要在函数上独立声明。  4、如果建立对象后指定了泛型的具体类型，那么该对象操作方法时，这些方法只能操作一种数据类型。  5、所以既可以在类上的泛型声明，也可以在同时在该类的方法中声明泛型。 |

泛型练习：

定义泛型成员

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) {  Father<String> f = **new** Father<String>("jack");  System.*out*.println(f.getT());  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());  }  }  **class** Father<T> {  **private** T t;  **public** Father() {  }  **public** Father(T t) {  **super**();  **this**.t = t;  }  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  } |

如果Father类有子类，子类该如何实现

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) {  Father<String> f = **new** Father<String>("jack");  System.*out*.println(f.getT());  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());  }  }  **class** Father<T> {  **private** T t;  **public** Father() {  }  **public** Father(T t) {  **super**();  **this**.t = t;  }  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  }  //子类指定了具体的类型  **class** Son **extends** Father<String>{    }  //子类也需要使用泛型  **class** Son3<T> **extends** Father<T>{    }  //错误写法，父类上定义有泛型需要进行处理  **class** Son2 **extends** Father<T>{    } |

## 泛型接口

|  |
| --- |
| **public** **class** Demo8 {  **public** **static** **void** main(String[] args) {  MyInter<String> my = **new** MyInter<String>();  my.print("泛型");  MyInter2 my2 = **new** MyInter2();  my.print("只能传字符串");  }  }  **interface** Inter<T> {  **void** print(T t);  }  // 实现不知为何类型时可以这样定义  **class** MyInter<T> **implements** Inter<T> {  **public** **void** print(T t) {  System.*out*.println("myprint:" + t);  }  }  //使用接口时明确具体类型。  **class** MyInter2 **implements** Inter<String> {  @Override  **public** **void** print(String t) {  System.*out*.println("myprint:" + t);  }  } |

## IO流

### IO流的分类

|  |  |  |  |
| --- | --- | --- | --- |
|  | **输入流** | **输出流** | **说明** |
| **字节流** | InputStream | OutputStream | 字节流是处理字节的（二进制） |
| **字符流** | Reader | Writer | 字符流是处理字符的 |

注：这几个类都是抽象类。

### 读文件的代码

**public** **static** **void** main(String[] args) {

String path = "c:/a.txt";

FileInputStream in = **null**;

**try** {

// 打开流

in = **new** FileInputStream(path);

// 使用流读文件内容

**int** b = in.read();

**while** (b != -1) {

System.*out*.print((**char**) b);

b = in.read();

}

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

} **finally** {

// 释放资源

**if** (in != **null**) {

**try** {

in.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

}

}

### 拷贝文件的代码

**public** **static** **void** main(String[] args) {

String srcPath = "c:/a.txt";

String destPath = "c:/b.txt";

// 一定要使用字节流

InputStream in = **null**;

OutputStream out = **null**;

**try** {

// 打开流

in = **new** FileInputStream(srcPath);

out = **new** FileOutputStream(destPath);

// 使用流

**byte**[] buf = **new** **byte**[1024 \* 8];

**for** (**int** len = -1; (len = in.read(buf)) != -1;) {

out.write(buf, 0, len);

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

// 释放资源

**try** {

**if** (in != **null**) {

in.close();

}

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

} **finally** {

**if** (out != **null**) {

**try** {

out.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

}

}

}

## 多线程

### 启动线程方式

1. 自定义的类继承Thread类。

使用代码为new MyThread().start()

2，自定义的类实现Runnable接口。

使用代码为new Thread(new MyRunnable()).start

### 代码

**以下代码是分别用两种方式启动线程（还是用到了匿名内部类）**

**private** **static** **int** *count* = 100;

**public** **static** **void** main(String[] args) {

// 用继承Thread类的方式启动一个线程

**new** Thread() {

**public** **void** run() {

**synchronized** (StartThreadTest.**class**) {

**while** (*count* > 0) {

*count*--;

System.*out*.println(Thread.*currentThread*() + "卖了一张票，还剩" + *count*);

}

}

}

}.start();

// 用实现Runnable接口的方式启动一个线程

**new** Thread(**new** Runnable() {

**public** **void** run() {

**synchronized** (StartThreadTest.**class**) {

**while** (*count* > 0) {

*count*--;

System.*out*.println(Thread.*currentThread*() + "卖了一张票，还剩" + *count*);

}

}

}

}).start();

}

# ThreadLocal

即线程本地变量，是一个java类。用于隔离多线程之间变量。

为每一个线程创建共享变量的副本。从而解决同步问题。每个线程操作自身的副本。

原理：底层维护了一个map集合，键对应线程，值对应其共享变量。

使用：

在线程中，用ThreadLocal<共享变量> tr=new ThreadLocal<共享变量>

取代 共享变量的声明。

应用： hibernate 使用该技术，实现了在service层实现了，多dao操作，使用同一session（数据库链接）完成事务。

## 锁

Java加锁的两种方式：

1. lock接口
2. synchronize关键字

乐观锁：读操作时候很乐观，认为不会有人同时访问，所以不加锁。写操作使用版本号或CAS实现加锁，如果是版本号，写之前读一次版本号，进行写操作，提交之前再读一次版本号，相同则更新；否则，重复之前操作。

适用场景：读操作远远大于写操作的场景。

存在问题：存在 不可重复读问题。

悲观锁：每次读写操作都加锁。

# Java内存模型

虚拟机栈：虚拟机栈保存了栈帧。栈帧（方法调用时的环境，局部变量表，操作数栈，返回地址），方法调用时栈帧入栈，结束时出栈。

局部变量表：包含基本数据类型，以及涉及到的类的引用。

本地方法栈：作用和虚拟机栈类似

堆：存放java实例对象。类的成员变量也保存在该区域。

方法区： 存放类信息，类方法，静态变量，静态方法，常量。

常量：内部维护一个常量池：编译时产生的常量，和程序运行时String 的intern方法产生的常量

程序计数器：保存下一条指令的地址，该块内存是线程私有的。

线程私有内存块： 仅有程序计数器。

# 垃圾回收算法

# 并发问题算法

# 类加载过程：

1. 加载

通过类的全限定名（包名+类名）从class文件获取二进制字节流，载入虚拟机方法区内存中；

并将字节码文件中的静态数据结构转化为虚拟机运行时的数据结构存储在方法区中；

并在java堆中创建对应的class对象（该对象提供了访问方法区中运行时数据结构的接口）。

二进制字节流获取途径：1.网络 实现了serializable接口的类，可序列化课用于网络传输和存储到数据库 2.数据库 3.class文件 4.动态生成的class文件

1. 连接
2. 验证：验证class文件的结构，是否达到jvm对class文件的规范
3. 文件格式验证：虚拟机能否正确获取字节码文件中的
4. 字段验证
5. 方法体验证
6. 符号引用验证
7. 准备： 为静态字段分配内存，基本数据类型赋值0 ,0L ,0.0f, false, 引用类型赋值null
8. 解析：将编译期字段生成的符号引用转化为直接引用。

直接引用：虚拟机可通过地址直接定位地址，直接引用说明一定被载入内存中了。

1. 类或者接口解析
2. 字段解析，从本类查找字段，若没找到，则去父类和父接口中找。
3. 方法解析，先判断是类方法还是接口方法，从而才去不同的解析方式。
   1. 类方：先搜索本类，再搜索父类，再搜索父接口
   2. 接口方法：接口没有父类，只有父接口。先搜索本接口，再搜索父接口。
4. 初始化：为静态字段初始化，并调用静态代码块。

init方法和clinit方法，是虚拟机在编译.class文件的时候，根据字段和初始化代码段自动生成的函数

init在类的实例化的时候调用

clinit在类初始化的时候调用

1. 使用
2. 卸载

# 三大类加载器

Bootstrap加载器：加载核心类

Extension加载器：加载jre/lib/ext目录下的类

System加载器：加载classpath下的类