Java 面试题整合

# Part 1. Java基础

## 谈谈你对面向对象编程(Object Oriented Programming-OOP)的理解

简单地讲，面向对象设计是一种程序设计技术。它将重点放在数据(即对象)和对象的接口上。用木匠打一个比方，一个“面向对象的”木匠是中关注的是所制作的椅子，第二位才是所使用的工具；一个“非面向对象的”，木匠首先考虑的是所用的工具。也就是说只关心对象的具体实现，而不考虑对象的具体实现。

## 面向对象编程的特性

封装、继承和多态；

### 封装

在面向对象的编程语言中，封装有两个相关但又截然不同或组合的概念。

▪ 一种语言机制-限制直接访问对象的组成部分(成员属性、成员方法)

▪ 一种语言构造，促进了数据与方法的绑定操作数据

封装是面向对象的基本原理。它指的是数据和方法绑定和对数据进行操作的方法绑定。封装是用来在一个类中隐藏一个结构化的数据对象的值或状态，防止未经授权的对象直接访问。public访问权限的方法通常提供访问数据的，提供给其他对象调用这些方法来获取和修改此对象中的值。

封装可用于隐藏数据成员和成员函数。在这个定义下，封装意味着对象的内部表示通常在对象定义之外隐藏起来。通常，只有对象自己的方法才能直接检索或操作字段，使用权限修饰符控制。

隐藏对象的内部保护其完整性，防止用户设置对象内部数据到一个无效的或不一致的状态。封装的好处是，它可以减少系统的复杂性，从而提高鲁棒性，通过允许开发人员限制对象之间的相互依赖关系。

有一种方法可以覆盖这种傲虎机制-通过反射可以获取对象内部的所有成员。

对象的数据封装彻底消除了传统结构方法中数据与操作分离所带来的种种问题，提供了程序的可复用性和可维护性，降低了程序猿保持数据与操作内容的负担。

对象的数据封装特性还可以吧对象的私有数据和公共数据分离开，保护了私有数据，减少了可能的模块之间的干扰，达到降低程序复杂性、提高可控性的目的。

### 继承

#### 概念

继承已存在的类就是继承(复用)这些类的方法和域。在此基础上，还可以添加一些新的方法和域，以满足新的需求。

1.2.2.2 继承的特性

▪ 子类拥有父类非private的属性和方法；

▪ 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展；

▪ 子类可以用自己的方式实现父类的方法；

▪ Java的继承是单继承，但是可以多重继承，单继承就是一个子类只能继承一个父类，重重继承就是，例如：A类继承B类，B类继承C类，所以按照关系C类是B类的父类，B类是A类的父类，这是Java继承区别于C++继承的一个特性；

▪ 提高了类之间的耦合性(继承的缺点，耦合度高就会造成代码之间的联系)

#### 继承关键字

继承可以使用extends和implements这两个关键字来实现继承，而且所有的类都是继承于java.lang.Object，当一个类没有继承显示的继承任何类，则默认继承Object(这个类在java.lang包中，所以不需要import)祖先类。

extends关键字:

在Java中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以extends只能继承一个类。

implements关键字:

使用implements关键字可以变相的使java具有多重继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口(接口跟接口之间采用逗号分隔)

#### super 与 this 关键字

super关键字：我们可以通过super关键字来实现对父类非private成员的访问，用来应用当前对象的父类。

提示

this关键字：指向自己的引用。

#### final类和方法

▪ final类(最终类)，不能被继承；

▪ final 方法，不能被子类重写

#### 构造器

子类不能继承父类的构造器，但是父类的构造器带有参数的，则必须在子类的构造器中显示地通过super关键字调用父类的构造器并配以适当的参数列表。

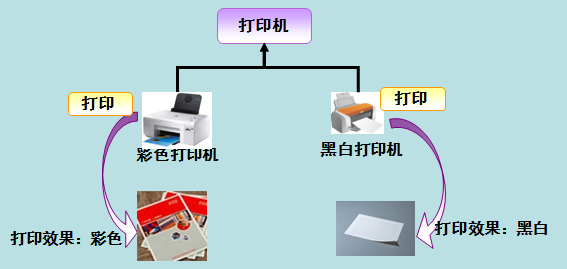
如果父类有午餐构造器，则在子类的构造器中用super调用父类构造器不是必须的，如果没有使用super关键在，系统会自动调用父类的无参构造器。

### 多态

#### 概念

多态是同一个行为具有多种不同表现形式或形态的能力；

多态就是同一个接口，使用不同的实例而执行不同操作；



#### 多态存在的三个必要条件

▪ 继承

▪ 重写

▪ 父类引用指向子类对象

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再在调用子类的同名方法；

多态的好处是可以使程序有良好的扩展性，并可以对所有类的对象进行通用处理。

#### 多态的优点

▪ 消除类型之间的耦合关系；

▪ 可替换性；

▪ 可扩充性

▪ 接口性

▪ 灵活性

▪ 简化性

#### 多态的实现方式

方式一：重写

方式二：接口

▪ 生活中的接口最具代表性的就是插座，例如一个三接头都能接在三孔插座中，因为这个是每个国家都有各自规定的接口规则，有可能到外国就不行，那是因为国外自己定义的接口类型。

▪ java中的接口类似于生活中的几口，就是一些方法特征的集合，但没有方法的实现。

## 面向对象设计的五大原则(SOLID)

### 单一职责原则(SRP-Single responsibility principle)

#### 概念

如果一个类承担的职责过多，就等于把这些职责耦合在了一起。一个职责的变化可能削减或者一直这个类完成其它职责的能力。这种耦合会导致脆弱的设计，当变化时设计会遭受到意想不到的破坏。

什么是职责？在SRP中，我们把职责定义为“变化的原因”。如果你能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责。

#### 反例

Modem.java—违反SRP

|  |
| --- |
| interface Modem{  public void dial(String pno);  public void hangup();  public void send();  public void recv();  } |

该接口中可以看出两个职责。第一个职责是连接管理；第二个职责是数据通信。dial和hangup还书进行调至解调器的连接处理，而send和recv函数进行数据通信。

#### 范例

DataChannel.java和Connection.java—符合SRP

|  |
| --- |
| interface DataChannel{  public void send(char c);  public void recv();  } |

|  |
| --- |
| interface Connection{  public void dial(String pno);  public void changup();  } |

### 开放封闭原则(OCP-Open Close Principle)

#### 概念

开闭原则的核心是对扩展开放，对修改关闭，白话的意思就是我们改变一个软件时(比如扩展其它功能),应该通过扩展的方式来达到软件的改变，而不应该修改原有代码来实现变化。

### 里氏替换原则(LSP)

#### 概念

很严格的原则，规则是“子类必须能够替换基类，否则不应该设计为子类”。也就是说，子类只能去扩展基类，而不是隐藏或覆盖基类。

#### 优点

可以很容易的实现同一父类下各个子类的互换，而客户端可以好不察觉。

### 依赖倒置原则(DIP)

#### 概念

“设计要依赖于抽象而不是具体化”。换句话说就是设计的时候我们要用抽象来考虑，而不是一上来就开始划分我需要哪些类，应为这些事具体的实现。

#### 优点

人的思维本身实际上就是很抽象的，我们分析问题的时候不是一下子就考虑到细节，而是很抽象的将整个问题都构思出来，所以面向抽象设计是符合人的思维的。另外这个原则会很好的支持OCP，面向抽象的设计使我们能够不必太多以来与实现，这样扩展就成为了可能，这个原则也是另一片文章《Design by Contract》的基石。

### 接口分离原则(ISP)

#### 概念

“将大的接口打散成多个小接口”，让系统解耦，从而容易重构，改变和重新部署。

#### 优点

会使一个软件系统功能扩展时，修改的压力不会传到别的对象。

### 迪米特法则

#### 概念

又称为最少知道原则，这个原则首次在Demeter系统中得到正式运用，所以定义为迪米特法则。它讲的是“一个对象应当尽可能少的取了解其他对象”。

#### 优点

消除耦合

## 方法重载和重写

#### 重载

在Java中的2个或2个以上的方法可以有同一个名字，只要他们参数个数或参数类型不同即可(和返回类型无关)

▪ 必须是同一个类；

▪ 方法名一样；

▪ 参数类型或个数或顺序不一样，满足其中一个或多个。

#### 重写

在类层次结构中，当子类中的方法与父类中的方法具有相同的返回类型和签名时，就称子类中的方法重写了父类中的方法。

▪ 继承；

▪ 子类的方法的名称参数和返回类型和父类的方法相同；

▪ 子类不能缩小父类方法的访问权限。

## static 和 final

### static

#### 静态域

如果将域定义为static，每个类中只有一个这样的域。而每个对象对于所有的实例域却都有自己的一份拷贝。例如，嘉定需要给每个雇员赋予唯一的标识码。这里给Employee类添加一个实例域id和一个静态域nextId。

|  |
| --- |
| class Employee{  private static int nextId = 1;  private int id;  } |

现在，每个雇员对象都有一个自己的id域，但这个类的所有实例将共享一个nextId域。换句话说，如果有1000个Employee类的对象，择优1000个实例域id。但是，只有一个静态域nextId。及时没有一个雇员对象，静态域nextId也存在。它数据类，而不属于任何独立对象。

#### 静态常量

使用static final修饰的域称为静态常量。

静态变量使用的比较少，但静态常量却使用的比较多。例如，在很多基本数据类型的封装类型中有许多静态常量。

由于每个类对象都可以对共有域进行修改，所以，最好乣将域设计为public。然而，共有常量却没问题。常量只能赋值一次后不能被修改。

final常量不能被修改，但，本地方法可以绕过Java语言的存取控制机制。这是一种特殊的方法，在自己编写程序时，不应该这样处理。

#### 静态方法

使用static修饰的方法称为静态方法。

▪ 静态方法不能访问实例域(非静态成员)；

▪ 静态方法能够访问自身类中静态域(静态成员);

▪ 静态方法属于类，不属于任何对象；

▪ 静态随着类的加载而装载，随着类的消失而消失；

▪ 静态优先于对象存在，被对象共享；

▪ 静态存在于内存中无法访问后来的对象中的数据，所以静态无法访问非静态；

▪ 静态方法内部无法使用this，应为这时对象有可能不存在，this没有任何指向。

#### 静态内部类

### final

#### 简介

Java中的final关键字非常重要，它可以应用于类、方法、变量。此章节将介绍什么是final关键字？将变量、方法和类声明为final代表了什么？使用final的好处是什么？最后也有一些使用final关键字的实例。final经常和static一起使用来声明常量，也会介绍final是如何改善应用性能的。

#### final关键字的含义

final在Java中是一个保留的关键字，可以声明成员变量、方法、类以及本地变量。一旦你将引用声明为final，你将不能改变这个引用了，编译器会检查代码，如果你视图将变量再次初始化的话，编译器会报编译错误。

#### final变量

凡是对成员变量或本地变量(局部变量)声明为final的都叫做final变量。final经常和static关键字一起使用，作为常量。下面是final变量的例子：

|  |
| --- |
| class A{  private static final Logger LOGGER = LoggerFactory.createLogger(A.class);  } |

#### final方法

final可以声明方法。方法前面加上final关键字，代表这个方法不可以被子类的方法重写。如果你认为一个方法的功能已经足够完整了，子类中不需要改变的话，你可以声明此方法为final。final方法比非final方法要快，因为在编译的时候已经静态绑定了，不需要在运行时再动态绑定。下面是final方法的例子：

|  |
| --- |
| class PersonalLoan{  public final String getName(){  return “personal loan”;  }  }  class CheapPersonalLoan extends PersonalLoan{  @Override  public final String getName(){  return “cheap personal loan”;//会报编译错误：overridden method is final  }  } |

#### final类

使用final来修饰的类叫做final类。final类通常功能是完整的，不能被继承。Java中有许多类是final的，如:String,Interger以及其它保证类。下面是final类的实例：

final类是指它的对象一旦被创建了就不能被更改了。String是不可变类的代表。不可变类有很多好处，譬如它们的对象是只读的，可以在多线程环境下安全的共享，不用额外的同步开销等等。

|  |
| --- |
| final class PersonalLoan{  } |

#### final关键字的好处

▪ final关键字提高了性能。JVM和Java应用都会缓存final变量；

▪ final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销；

▪ 使用final关键字，JVM会对方法、变量、类进行优化。

#### 关于final的重要知识点

1. final关键字可以用于成员变量、局部变量、方法和类；
2. final成员变量必须在声明的时候初始化或者在构造器中初始化，否则会编译错误；
3. 不能对final变量再次赋值；
4. 局部final变量必须在声明时赋值；
5. 在匿名类中所有变量都必须是final变量；
6. final方法不能被重写；
7. final类不能被继承；
8. final关键字不同于finally关键字，后者用于异常处理；
9. final关键字容易与finalize()方法搞混，后者是在Object类中定义的方法，是在垃圾回收之前被JVM调用的方法；
10. 接口中声明的所有变量本身是final的；
11. final和abstract这两个关键字是反相关的，final不能修饰abstract类；
12. final方法在编译阶段绑定，成为静态绑定；
13. 没有在声明时初始化final变量的成为空白final变量，他们必须在构造器中初始化，或者调用this()初始化。不这样做的话，编译器会报错“final变量需要进行初始化”；
14. 将类、方法、变量声明为final能够提高性能，这样JVM就有机会进行估计，然后优化；
15. 按照Java代码管理，final变量就是常量，而且通常常量名要大写；
16. 集合对象声明为final指的是引用不能被更改，但是你可以向其中增加、删除或更改内容。

## 泛型和反射

### 泛型

#### 引言

泛型是Java中一个非常重要的知识点，在Java集合类框架中被广泛应用。本章节将从零开始来看一下Java泛型的设计，将会设计到通配符处理，以及让人苦恼的类型擦除。

#### 泛型类

首先看一下普通的类的做法：

|  |
| --- |
| public class Box{  private String object;  public void set(String object){  this.object = object;  }  public String get(){  return object;  }  } |

这是最常见的做法，这样做的一个坏处是Box类只能装入String类型的元素，后续要装入Integer等其他类型的元素，还必须要另外重写一个Box，代码得不到复用，使用泛型类可以很好的解决这个问题。如下所示：

|  |
| --- |
| public class Box<T>{  //T 代表一种类型  private T t;  public void set(T t){  this.t = t;  }  public T get(){  return t;  }  } |

这样Box类就可以得到复用，我们可以将T替换成任何我们想要的类型。

#### 泛型方法

看完泛型类，接下来我们来了解一下泛型方法。声明一个泛型方法很简单，只要在返回类型前面加上一个类似<K, V>的形式就行了，如下所示：

|  |
| --- |
| public class Util{  public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2){  return p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());  }  }  public class Pair<K, V>{  private K key;  private V value;  public Pair(K key, V value){  this.key = key;  this.value = value;  }  public void setKey(K key){  this.key = key;  }  public void setValue(V value){  this.value = value;  }  public K getKey(){  return key;  }  public V getValue(){  return value;  }  } |

#### 边界符

现在我们要实现这样一个功能，查找一个泛型数组中大于某个特定元素的个数，我们可以这样实现：

|  |
| --- |
| public static <T> int countGreaterThan(T[] anArray, T elem) {  int count = 0;  for (T e : anArray)  if (e > elem) // compiler error 编译报错  ++count;  return count;  } |

但是这样明显是错误的，因为除了八中基本数据类型，其他的类型并以不一定使用操作符 “>”，所以编译器报错，这时候可以使用边界符。

#### 通配符

在了解通配符之前，我们首先必须要澄清一个概念，还是借用上面定义的Box类，假设我们添加一个这样的方法：

|  |
| --- |
| public void boxTest(Box<Number> n) { /\* ... \*/ } |

那么现在Box<Number> n 允许接收什么类型的参数？我们是能够传入Box<Integer>或者Box<Double>呢？答案是否定的，虽然Integer和Double是Number的子类，但是在泛型中Box<Integer>或Box<Double>与Box<Number>之间没有任何的关系。这一点非常重要，接下来我们通过完整的例子来加深一下理解。

首先我们先定义几个简单的类，下面我们将用到它：

|  |
| --- |
| class Fruit {}  class Apple extends Fruit {}  class Orange extends Fruit {} |

下面这个例子中，我们创建了一个泛型类Reader，然后在f1()中当我们尝试Fruit f = fruitReader.readExact(apples);编译器会报错，因为List<Fruit>与List<Apple>之间并没有任何的关系。

|  |
| --- |
| public class GenericReading {  static List<Apple> apples = Arrays.asList(new Apple());  static List<Fruit> fruit = Arrays.asList(new Fruit());  static class Reader<T> {  T readExact(List<T> list) {  return list.get(0);  }  }  static void f1() {  Reader<Fruit> fruitReader = new Reader<Fruit>();  // Errors: List<Fruit> cannot be applied to List<Apple>.  // Fruit f = fruitReader.readExact(apples);  }  public static void main(String[] args) {  f1();  }  } |

按照我们通常的思维习惯，Apple和Fruit之间肯定是存在联系，然而编译器却无法识别，那怎么在泛型代码中解决这个问题呢？我们可以通过使用通配符来解决这个问题：

|  |
| --- |
| static class CovariantReader<T> {  T readCovariant(List<? extends T> list) {  return list.get(0);  }  }  static void f2() {  CovariantReader<Fruit> fruitReader = new CovariantReader<Fruit>();  Fruit f = fruitReader.readCovariant(fruit);  Fruit a = fruitReader.readCovariant(apples);  }  public static void main(String[] args) {  f2();  } |

这样就相当于告诉编译器，fruitReader的readCovariant方法接收的参数只要是满足Fruit的子类就行(包括Fruit)，这样子类和父类之间的关系也就关联上了。

#### PECS原则

上面我们看到了类似<? extends T>的用法，利用它我们可以从list里面get元素，那么我们可不可以往list里边add元素呢？我们来尝试一下：

|  |
| --- |
| public class GenericsAndCovariance {  public static void main(String[] args) {  // Wildcards allow covariance:  List<? extends Fruit> flist = new ArrayList<Apple>();  // Compile Error: can't add any type of object:  // flist.add(new Apple())  // flist.add(new Orange())  // flist.add(new Fruit())  // flist.add(new Object())  flist.add(null); // Legal but uninteresting  // We Know that it returns at least Fruit:  Fruit f = flist.get(0);  }  } |

答案是否定的，Java编译器不允许我们这样做，为什么呢？对于这个问题我们不妨从编译器的角度去考虑。因为List<? extends Fruit> flist 它本身可以有多种含义：

|  |
| --- |
| List<? extends Fruit> flist = new ArrayList<Fruit>();  List<? extends Fruit> flist = new ArrayList<Apple>();  List<? extends Fruit> flist = new ArrayList<Orange>(); |

▪ 当我们尝试add一个Apple的时候，flist可能指向new ArrayList<Orange>();

▪ 当我们尝试add一个Orange的时候，flist可能指向new ArrayList<Apple>();

▪ 当我们尝试add一个Fruit的时候，这个Fruit可以是任何类型的Fruit，而flist可能指向某种特定类型的Fruit，编译器无法识别所以会报错。

所以对于实现了<? extends T>的集合类只能将它视为Producer向外提供(get)元素，而不能作为Consumer对外来获取(add)元素。

如果我们要add应该怎么做呢？可以使用<? super T>:

|  |
| --- |
| public class GenericWriting {  static List<Apple> apples = new ArrayList<Apple>();  static List<Fruit> fruit = new ArrayList<Fruit>();  static <T> void writeExact(List<T> list, T item) {  list.add(item);  }  static void f1() {  writeExact(apples, new Apple());  writeExact(fruit, new Apple());  }  static <T> void writeWithWildcard(List<? super T> list, T item) {  list.add(item)  }  static void f2() {  writeWithWildcard(apples, new Apple());  writeWithWildcard(fruit, new Apple());  }  public static void main(String[] args) {  f1(); f2();  }  } |

这样我们可以往容器里添加元素了，但是使用super的坏处是以后不能get容器里的元素了，原因很简单，我们继续从编译器的角度考虑这个问题，对于List<? super Apple> list，它可以有下面几种含义：

|  |
| --- |
| List<? super Apple> list = new ArrayList<Apple>();  List<? super Apple> list = new ArrayList<Fruit>();  List<? super Apple> list = new ArrayList<Object>(); |

当我们尝试通过list来get一个Apple的时候，可能会get到一个fruit，这个Fruit可以是Orange等其它类型的Fruit。

根据上面的例子，我们可以总结出一条规律，“Producer Extends，Consumer Super”：

▪ “Producer Extends”-如果你需要一个只读list，用它来produce T,那么使用 ? extends T;

▪ “Consumer Super”-如果你需要一个只读list，用它来consume T,那么使用 ? super T;

▪ 如果需要同时读取以及写入，那么我们就不能使用通配符了。

#### 类型擦除

Java泛型中最令人苦恼的地方或许就是类型擦除了，特别是对于C++经验的程序猿。类型擦除就是说Java泛型只能用于在编译期间的静态类型检查，然后编译器生成的代码会擦除相应的类型信息，这样到了运行期间实际上JVM根本就知道泛型所代表的具体类型。这样做的目的是因为Java泛型是1.5之后才被引入的，为了保持向下兼容性，所以只能做类型擦除来兼容以前的非泛型代码。对于这一点，如果阅读Java集合框架的源码，可以发现有些类型其实并不支持泛型。

泛型擦除到底是什么意思呢？看下面这个简单的例子：

|  |
| --- |
| public class Node<T> {  private T data;  private Node<T> next;  public Node(T data, Node<T> next) {  this.data = data;  this.next = next;  }  public T getData() { return data; }  // ...  } |

编译器做完相应的类型检查之后，实际上到了运行期间上面这段代码实际上将转换为：

|  |
| --- |
| public class Node {  private Object data;  private Node next;  public Node(Object data, Node next) {  this.data = data;  this.next = next;  }  public Object getData() { return data; }  // ...  } |

这意味着不管我们生命Node<String>还是Node<Integer>，到了运行期间，JVM统统视为Node<Object>。有没有什么办法可以解决这个问题呢？这就需要我们自己重新设置bounds了，将上面的代码修改为：

|  |
| --- |
| public class Node<T extends Comparable<T>> {  private T data;  private Node<T> next;  public Node(T data, Node<T> next) {  this.data = data;  this.next = next;  }  public T getData() { return data; }  // ...  } |

这样编译器就会将T出现的地方替换成Comparable而不是默认的Object了：

|  |
| --- |
| public class Node {  private Comparable data;  private Node next;  public Node(Comparable data, Node next) {  this.data = data;  this.next = next;  }  public Comparable getData() { return data; }  // ...  } |

上面的概念或许还是比较好理解，但其实泛型擦除带来的问题远远不止这些，接下来我们系统地看一下类型擦除所带来的一系列问题，有些问题在C++的泛型中可能不会遇见，但是在Java中却需要格外小心。

**问题一：**

参考：<http://www.importnew.com/24029.html>

## 接口和抽象类

### 抽象类

使用abstract修饰的类叫做抽象类。使用abstract修饰的方法较抽象方法。抽象类中含有无具体实现的方法，所以不能用抽象类创建对象。

|  |
| --- |
| [public] abstract class ClassName {  abstract void fun();  } |

从这里可以看出，抽象类就是为了继承而存在的，如果定义了一个抽象类，却不去继承它，那么等于白白创建了这个抽象类，因为你不能用它来做任何事情。对于一个父类如果他的某个方法在父类中实现出来没有任何意义，必须根据子类的实际需求来进行不同的实现，那么就可以将这个方法声明为abstract方法，此时这个类也就成为了abstract类了。

包含抽象方法的类成为抽象类，但并不意味着抽象类中只能有抽象方法，它和普通类一样，同样可以拥有成员变量和普通的成员方法。注意，抽象类和普通类的主要由三点区别。

▪ 抽象方法必须为public或者protected，因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public；

▪ 抽象类不能用来创建对象；

▪ 如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为abstract类。

使用抽象类是需注意几点：

▪ 抽象类不能被实例化，实例化的工作应该交由它的子类来完成，它只需要有一个引用即可；

▪ 抽象方法必须有子类来进行重写；

▪ 只要包含一个抽象方法的抽象类，该方法必须定义为抽象类，不管是否还包含其它方法；

▪ 抽象类中可以包含具体的方法，当然也可以不包含抽象方法；

▪ 子类中的抽象方法不能与父类的抽象方法同名；

▪ abstract不能与final并列修饰同一个类；

▪ abstract不能与private、static、final或native并列修饰同一个方法。

### 接口

接口，英文interface，在软件工程中，接口泛指供别人调用的方法或函数。从这里，我们可以体会到Java语言设计者的初衷，它是行为的抽象。在Java中，定一个接口的形式如下：

|  |
| --- |
| [public] interface InterfaceName {    } |

接口中可以含有变量和方法。但是要注意，接口中的变量会被隐示地指定为public static final变量(并且只能是public static final变量，用private修饰会报编译错误),而方法会被隐示地指定为public abstract方法且只能是public abstract方法(用其它关键字，比如private、protected、static、final等修饰会报编译错误),并且接口中所有的方法不能有具体的实现，也就是说，接口中的方法必须都是抽象方法。从这里可以隐约看出接口和抽象类的区别，接口是一种极度抽象的类型，它比抽象类更加“抽象”，并且一般情况下不在接口中定义变量。

要让一个类遵循某组特地的接口需要使用implements关键字，具体格式如下：

|  |
| --- |
| class ClassName implements Interface1,Interface2,[....]{  } |

可以看出，允许一个类遵循多个特定的接口。如果一个非抽象类遵循了某个接口，就必须实现接口中的所有方法。对于遵循某个接口的抽象类，可以不识闲该接口中的抽象方法。

### 抽象类和接口的区别

1. 语法层面上的区别

▪ 抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract方法；

▪ 抽象类中的成员变量可以是各种类型，而接口中的成员变量只能是public static final类型；

▪ 接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；

▪ 一个类只能继承一个抽象类，而一个类可以实现多个接口。

1. 设计层面上的区别

▪ 抽象类是对一种事务的抽象，即对类抽象，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，但是接口缺失对类局部(行为)进行抽象。举个简单的例子，飞机和鸟是不同类的事务，但是他们都是一个共性，就是都会飞。那么在设计的时候，可以将飞机设计为一个类Airplane，将鸟设计为一个类Bird，但是不能将飞行这个特性也设计为类，因此它只是一个行为特性，并不是对一类事务的抽象描述。此时可以将飞行设计为一个接口Fly，包含方法fly()，然后Airplane和Bird分别根据自己的需要实现Fly这个接口。然后至于有不同种类的飞机，比如战斗机、民用飞机等直接继承Airplane即可，对于鸟也是类似的，不同种类的鸟直接继承Bird类即可。从这里可以看出，继承是一个“是不是 is-a”的关系，而接口实现则是“有没有”的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现则是有没有、具备不具备的关系，比如鸟是否能飞(或者是否具备飞行这个特点)，能飞行则可以实现这个接口，不能飞行就不实现这个接口。

▪ 抽象类作为很多子类的父类，他是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。什么是模板式设计？最简单例子，大家都用过ppt里面的模板，如果用模板A设计了ppt B和ppt C公共的部分就是模板A了，如果他们的公共部分需要改动，则只需要改动模板A就可以了，不需要重新对ppt B和ppt C进行改动。而辐射式设计，比如某个电梯都装了某种报警器，一旦要更新报警器，就必须全部更新。也就是说对于抽象类，如果需要添加新的方法，可以直接在抽象类中添加具体的实现，子类可以不进行变更；而对于接口则不行，如果接口进行了变更，则所有实现这个接口的类都必须进行相应的改动。

接口和抽象类组合使用：

▪ 抽象类可以继承抽象类；

▪ 抽象类可以实现接口(并为接口中的行为定义默认行为)；

▪ 接口可以实现接口；

▪ 接口不能继承任何东西，因为它不做具体实现；

## 异常

简介

程序运行时，发生的不被期望的时间，它阻止了程序按照程序猿的预期正常执行，这就是一场。异常发生时，是任程序自生自灭，like退出中止，还是输出错误给用户？或者用C语言风格：用函数返回值作为执行状态？

Java 提供了更加优秀的解决方法：异常处理机制。

异常处理机制能让程序猿在异常发生时，按照代码的欲仙设定的异常处理逻辑，针对性地处理异常，让程序进最大可能恢复正常并继续执行，且保持代码的清洗。

Java 中的异常可以是函数中的语句执行时引发的，也可以是程序猿通过throw语句手动抛出的，只要在Java程序中产生了异常，就会用一个对应的异常对象类封装异常，JRE就会视图寻找异常处理程序来处理异常。

Throwable类是Java异常类型的顶层父类，一个对象只有是Throwable类的(直接或间接)实例，它才是一个异常对象，才能被异常处理机制识别。JDK中内建了一些常用的异常类，我们也可以自定义异常。

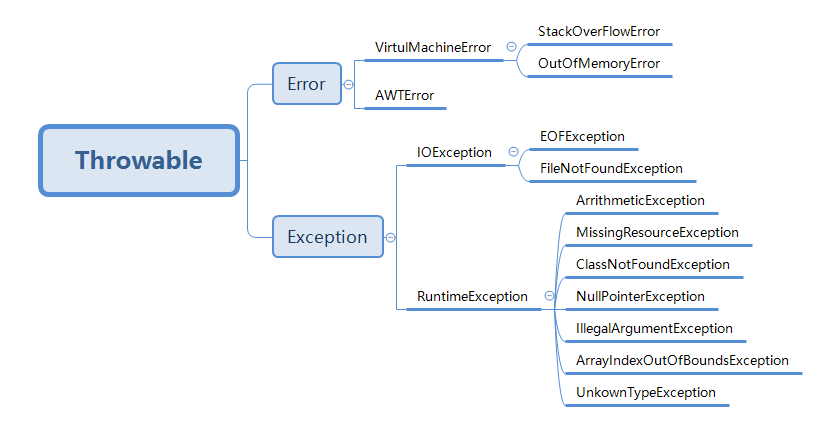
Java异常的分类和类结构图：

Java标准库内建了一些通用的异常，这些类以Throwable为顶层父类。

Throwable又派生出Error类和Exception类。

Error类以及他的子类的实例，代表了JVM本身的错误。错误不能被程序猿通过代码处理，Error很少出现。因此， 程序猿应该关注Exception为父类的分支下的各种异常类型。

Exception以及它的子类，代表程序运行时发生的各种不期望发生的事件。可以被Java异常处理机制使用，是异常处理的核心。



总体上我们根据Javac对异常的处理要求，将异常类分为2类，非检查异常、检查异常。

非检查异常(unckecked exception): Error和RuntimeException以及它们的子类。javac在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常。所以如果愿意，我们可以编写代码处理这样的异常，也可以不处理。对于这些异常，我们应该修正代码，而不是通过异常处理器处理。这样的异常发生的原因多半是代码写的有问题。如除以0错误ArithmeticException，错误的强制类型转换错误ClassCastException ，数组索引越界ArrayIndexOutOfBoundException,使用了空对象NullPointerException等。

检查异常(checked exception)：除了Error和RuntimeException的其他异常。javac强制要求程序猿为这样的异常做预备处理工作(使用try…catch…finally或者throws)。在方法中要么用try-catch语法捕获它并处理，要么用throws字句声明抛出它，否则编译不会通过。这样的异常一般是有程序的运行环境导致的。因为程序猿可能被运行在各种位置的环境下，而程序猿无法干预用户如何使用它编写的程序，于是程序猿就应该为这样的异常时刻准备着。如SQLException，IOException，ClassNotFoundException等。

异常链化

有一些大型的，模块化的软件开发中，一旦一个地方发生异常，则如骨牌效应一样，将导致一连串的异常。假设B模块完成自己的逻辑需要调用A模块的方法，如果A模块发生异常，则B也将不能完成而发生异常，但是B在抛出异常时，会将A的异常信息掩盖掉，这将是的异常的根源信息丢失。异常的链化可以将多个模块的异常串联起来，是的异常信息不回丢失。

异常链化：以一个异常对象为参数构造新的异常对象。新的异常对象包含先前导致异常的信息。这项技术主要是异常类的一个带Throwable参数的函数来实现的。这个当做参数的异常，我们叫他根源异常(cause)。

查看Throwable类源码，可以发现里面有一个Throwable字段cause，就是它保存了构造时传递的根源异常参数。这种设计和链表的节点类设计如出一辙，因此形成链也是自然的了。

|  |
| --- |
| public class Throwable implements Serializable {  private Throwable cause = this;    public Throwable(String message, Throwable cause) {  fillInStackTrace();  detailMessage = message;  this.cause = cause;  }  public Throwable(Throwable cause) {  fillInStackTrace();  detailMessage = (cause==null ? null : cause.toString());  this.cause = cause;  }    //........  } |

下面是一个例子，演示了异常的链化：从命令行输入2个int，将他们相加，输出。输出的数不是int则导致getInputNumbers异常，从而导致add函数异常，则可以在add函数中抛出。

|  |
| --- |
| public static void main(String[] args)  {    System.out.println("请输入2个加数");  int result;  try  {  result = add();  System.out.println("结果:"+result);  } catch (Exception e){  e.printStackTrace();  }  }  //获取输入的2个整数返回  private static List<Integer> getInputNumbers()  {  List<Integer> nums = new ArrayList<>();  Scanner scan = new Scanner(System.in);  try {  int num1 = scan.nextInt();  int num2 = scan.nextInt();  nums.add(new Integer(num1));  nums.add(new Integer(num2));  }catch(InputMismatchException immExp){  throw immExp;  }finally {  scan.close();  }  return nums;  }    //执行加法计算  private static int add() throws Exception  {  int result;  try {  List<Integer> nums =getInputNumbers();  result = nums.get(0) + nums.get(1);  }catch(InputMismatchException immExp){  throw new Exception("计算失败",immExp); /////////////////////////////链化:以一个异常对象为参数构造新的异常对象。  }  return result;  }  请输入2个加数  a 1  java.lang.Exception: 计算失败  at com.daily.dailytest.exception\_test.ExceptionChain.add(ExceptionChain.java:61)  at com.daily.dailytest.exception\_test.ExceptionChain.main(ExceptionChain.java:29)  Caused by: java.util.InputMismatchException  at java.util.Scanner.throwFor(Scanner.java:864)  at java.util.Scanner.next(Scanner.java:1485)  at java.util.Scanner.nextInt(Scanner.java:2117)  at java.util.Scanner.nextInt(Scanner.java:2076)  at com.daily.dailytest.exception\_test.ExceptionChain.getInputNumbers(ExceptionChain.java:41)  at com.daily.dailytest.exception\_test.ExceptionChain.add(ExceptionChain.java:58)  ... 1 more |

自定义异常

如果要自定义异常类，则扩展Exception类即可，因此这样的自定义异常都属于检查异常(checked exception)。如果要自定义非检查异常，则扩展自RuntimeException。

自定义异常应该总是包含如下的构造函数：

▪ 一个无参构造函数；

▪ 一个带有String参数的构造函数，并传递给父类的构造函数；

▪ 一个带有String参数和Throwable参数，并都传递给父类构造函数；

▪ 一个带有Throwable参数的构造函数，并传递给父类的构造函数。

下面是IOException类的完整源代码，可以作为借鉴：

|  |
| --- |
| public class IOException extends Exception {  static final long serialVersionUID = 7818375828146090155L;  public IOException() {  super();  }  public IOException(String message) {  super(message);  }  public IOException(String message, Throwable cause) {  super(message, cause);  }  public IOException(Throwable cause) {  super(cause);  }  } |

异常的注意事项

▪ 当子类重写父类的带有throws声明的函数时，其throws声明的异常必须在父类异常的可控范围内----用于处理父类的throws方法的异常处理器，必须也适用于子类的这个带throws方法。这是为了支持多态。

例如，父类方法throws的是2个异常，子类就不能throws3个及以上的异常。父类throws IOException，子类就必须throws IOException或者IOException的子类。

▪ Java程序可以是多线程的。每一个线程都是一个独立的执行流，独立的函数调用栈。如果程序只有一个线程，那么没有被任何代码处理的异常会导致程序终止。如果是多线程的，那么没有被任何代码处理的异常仅仅会导致异常所在的线程结束。

也就是说，Java中的异常是线程独立的，线程的问题应该有线程自己来解决，而不要委托到外部，也不会直接影响到其他线程的执行。

finally和return

首先如容易理解的事实：在try块中即便有return，break，continue，等改变执行流的语句，finally也会执行。也就是说，try…catch…finally中的return只要能执行，就都执行了，他们共同指向同一个内存地址写入返回值，后执行的将覆盖先执行的数据，而真正被调用者取的返回值就是最后一次写入的。那么，按照这个思想，下面的这个例子也就不难理解了。

|  |
| --- |
| public static void main(String[] args)  {  int result;    result = foo();  System.out.println(result); /////////2    result = bar();  System.out.println(result); /////////2  }    @SuppressWarnings("finally")  public static int foo()  {  trz{  int a = 5 / 0;  } catch (Exception e){  return 1;  } finally{  return 2;  }    }    @SuppressWarnings("finally")  public static int bar()  {  try {  return 1;  }finally {  return 2;  }  } |

finally中的return会抑制(消灭)前面try或者catch块中的异常

|  |
| --- |
| class TestException  {  public static void main(String[] args)  {  int result;  try{  result = foo();  System.out.println(result); //输出100  } catch (Exception e){  System.out.println(e.getMessage()); //没有捕获到异常  }    try{  result = bar();  System.out.println(result); //输出100  } catch (Exception e){  System.out.println(e.getMessage()); //没有捕获到异常  }  }    //catch中的异常被抑制  @SuppressWarnings("finally")  public static int foo() throws Exception  {  try {  int a = 5/0;  return 1;  }catch(ArithmeticException amExp) {  throw new Exception("我将被忽略，因为下面的finally中使用了return");  }finally {  return 100;  }  }    //try中的异常被抑制  @SuppressWarnings("finally")  public static int bar() throws Exception  {  try {  int a = 5/0;  return 1;  }finally {  return 100;  }  }  } |

finally中的异常会覆盖(消灭)前面try或者catch中的异常

|  |
| --- |
| class TestException  {  public static void main(String[] args)  {  int result;  try{  result = foo();  } catch (Exception e){  System.out.println(e.getMessage()); //输出：我是finaly中的Exception  }    try{  result = bar();  } catch (Exception e){  System.out.println(e.getMessage()); //输出：我是finaly中的Exception  }  }    //catch中的异常被抑制  @SuppressWarnings("finally")  public static int foo() throws Exception  {  try {  int a = 5/0;  return 1;  }catch(ArithmeticException amExp) {  throw new Exception("我将被忽略，因为下面的finally中抛出了新的异常");  }finally {  throw new Exception("我是finaly中的Exception");  }  }    //try中的异常被抑制  @SuppressWarnings("finally")  public static int bar() throws Exception  {  try {  int a = 5/0;  return 1;  }finally {  throw new Exception("我是finaly中的Exception");  }    }  } |

上面三个例子都异于常人的编码思维，因此建议如下：

▪ 不要在finally中使用return；

▪ 不要在finally中抛出异常；

▪ 减轻finally的任务，不要在finally中做一些其它的事情，finally块仅仅用来释放资源是最合适的。

▪ 尽量将所有的return写在函数的最后面，而不是try…catch…finally中；

▪ 尽可能的减小try块；

▪ catch语句应当尽量指定具体的异常类型，而不应该指定涵盖范围太广的Exception类；

▪ 既然捕获了异常，就要对它进行适当的处理。不要捕获异常之后又把它丢弃，不予理睬；

▪ 在异常处理模块中提供适量的错误原因信息，组织错误信息使其易于理解和阅读；

▪ 不要在构造函数中抛出异常。

## 集合

Collection接口是最基本的集合接口，它不提供直接的实现。Collection所代表的是一种规则，它所包含的元素都是必须遵循一条或者多条规则。如有些允许重复而有些规则不能重复，有些必须按照顺序插入而有些则是散列，有些支持排序而有些规则不支持排序。

首先要明确，代表了一组对象(和数组一样，但数组长度不能变，而集合能)。Java中的集合框架定义了一套规范，用来表示、操作集合，使具体操作与实现细节解耦。

其实说白了，可以把一个集合看成一个微型数据库，操作不外乎“增删改查”四中操作，我们在学习使用一个具体的集合类时，需要把这四中操作的时/空间福再度弄清楚了，基本上就可以说掌握这个类了。

集合的设计理念概括来说就是：提供一套“小而美”的API。API需要对程序猿友好，增加新功能时能让程序猿门快速上手。

为了保证核心接口足够小，最顶层的接口(也就是Collection与Map接口)并不会区分该集合是否可变，是否可更改，是否可改变大小这些细微的差别。相反，一些操作是可选的，在实现时抛出UNsupportedOperationException即可表示集合不支持该操作。集合的实现者必须在文档中声明哪些操作是不支持的。

为了保证最顶层的核心接口足够小，它们只能包含下面情况下的方法：

▪ 基本操作，像之前说的“增删改查”；

▪ 有一个重要的性能原因，为什么一个重要的实现会要重写它；

此外，所有的集合类都必须能提供友好的交互操作，这包括没有继承Collection类的数组对象。因此，框架提供一套方法，让集合类与数组可以互相转化，并且可以把Map看做集合。

集合基类Collection与Map

在集合框架的类继承体系中，最顶层有两个接口：

▪ Collection表示一组纯数据；

▪ Map表示一组key-value对；

一般继承自Collection或Map的集合类，会提供两个“标准”的构造函数

▪ 无参构造函数，创建一个空白的集合类；

▪ 带有Collection参数的有参构造方法，用于创建一个新的Collection，这个新的Collection与传入进来的Collection具备相同的元素。

因为接口中不能包含构造函数，所以上面这两个构造函数的约定并不是强制性的，但是在目前的集合框架中，所有继承自Collection或Map的子类都遵循这一约定。

Collection



▪ Set表示不允许有重复元素的集合；

▪ List表示允许有重复数据的集合；

▪ Queue JDK1.5新增，与上面两个集合主要的区别在于Queue主要存储数据，而不是处理数据。

Map



Map并不是真正意义上的集合，但是这个接口提供了三种“集合视角”，使得可以像操作集合一样操作他们，具体如下。

▪ 把Map的内容看做key的集合；

▪ 把Map的内容看做value的集合；

▪ 把Map的内容看做key-value映射的集合。

**List接口**

List接口为Collection直接接口。List所代表的是有序的Collection，即用某种特定的插入顺序来维护元素顺序。用户可以对列表中的每个元素的插入位置进行精确地控制，同时可以根据元素的整数索引访问元素，并搜索表中的元素。实现List接口的集合主要由ArrayList、LinkedList、Vector、Stack。

**ArrayList**

ArrayList 是一个动态数组，也是我们最常用的集合。它允许任何符合规则的元素插入甚至包括null。每一个ArrayList都有一个初始容量(10)，该容量代表了数组的大小。随着容器中的元素不断增加，容器的大小也会随着增加。在每次向容器中增加元素的同时都会进行容量检查，当快溢出时，就会进行扩容操作。所以如果我们明确所插入元素的多少，最好指定一个初始容量，避免过多的进行扩容操作而浪费时间、效率。

size、isEmpty、get、set、iterator和listIterator操作都以固定时间运行。add操作以分摊的固定时间运行，也就是说，添加n个元素需要O(n)时间(由于要考虑到扩容，所以这不只是添加元素会带来分摊固定时间开销那样简单)。

ArrayList擅长于随机访问。同时ArrayList是非同步的，线程不安全。

**LinkedList**

同样实现List接口的LinkedList与ArrayList不同，ArrayList是一个动态数组，而LinkedList是一个双向链表。所以它除了有ArrayList的基本操作方法外还额外提供了get、remove、insert、方法在LinkedList的首部或尾部。

由于实现的方式不同，LinkedList不能随机访问，它所有的操作都是要按照双向链表的操作执行。在列表中索引的操作将从头或尾遍历列表(从靠近指定索引的一端)。这样做的好处就是可以通过较低的代价在List中进行插入和删除操作。

与ArrayList一样，LinkedList也是非同步的。如果多个线程同时访问一个List，则必须自己实现访问同步，一种解决方法是在创建List是构造一个同步的List：

List list = Collections.synchronizedList(new LinkedList(…));

**Vector**

与ArrayList相似，但是Vector是同步的。所以说Vector是线程安全的动态数组。它的操作与ArrayList几乎一样。

**Stack**

Stack继承自Vector，实现一个后进先出的堆栈。Stack提供5个额外的方法使得被当做堆栈使用。基本的push和pop方法。还有peek方法得到栈顶的元素，empty方法测试堆栈是否为空，search方法检测一个元素在堆栈中的位置。Stack刚创建后是空栈

**Set接口**

Set是一种不包含重复元素的Collection。它维持他自己的内部排序，所以随机访问没有任何意义。与List一样，它同样允许null的存在但是仅有一个。由于Set接口接口的特殊性，所有传入Set集合中的元素都不许不同，同时要注意任何可变的对象，如果在对集合汇总元素进行操作时，导致e1.equals(e2) == true;则必定会产生某些问题。实现了Set接口的集合有：EnumSet、HashSet、TreeSet。

**EnumSet**

是枚举的专用Set。所有的元素都是枚举类型。

**HashSet**

HashSet堪称查询速度最快的集合，因为其内部是以HashCode来实现的。它内部元素的顺序是由哈希码来决定的，所以他不保证set的顺序；特别是它不保证该顺序恒久不变。

**TreeSet**

基于TreeMap，生成一个总是处于排序状态的set，内部以TreeMap来实现。他是使用元素的自然顺序对元素进行排序，或者根据创建Set时提供的Comparator进行排序，具体取决于使用的构造方法。

**Queue**

队列，它主要分为两大类，一类是阻塞式队列，队列满了以后再插入元素则会抛出异常，主要包括ArrayBlockQueue、PriorityBlockingQueue、LinkedBlockingQueue。另外一种队列则是双端队列，支持在头、尾两端插入和移除元素，主要包括：ArrayDeque、LinkedBlockingDeque、LinkedList。

**Map接口**

Map与List、Set接口不同，他是由一些列键值对组成的集合，提供了key到value的映射。同时它也没有继承Collection。在Map中它保证了key与value之间的一一对应关系。也就是说一个key对应一个value，所以它不能存在相同的key值，当然value值可以相同。实现Map的有：HashMap、TreeMap、HashTable、Properties、EnumMap。

**HashMap**

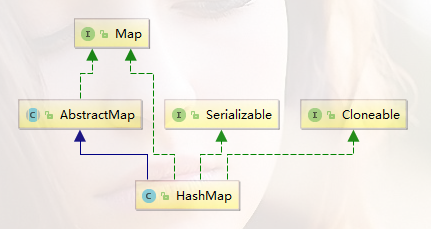
以哈希表数据结构实现，查找对象时通过哈希函数计算其位置，它是为快速查询而设计的，其内部定义了一个hash表数组(Entry[] table)，元素会通过哈希转换函数将元素的哈希地址转换成数组中存放的索引，如果有冲突，则使用散列链表的形式将所有相同哈希地址的元素串起来，可能通过查看HashMap.Entry的源码它是一个单链表结构。

**TreeMap**

键以某种排序规则排序，内部以red-black(红-黑)树数据结构实现，实现了SortedMap接口。

**HashTable**

也是一哈希表数据结构实现的，解决冲突时与HashMap也一样也是采用了散列链表的形式，不过性能比HashMap要地。



**异同点**

**Vector和ArrayList**

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。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，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%，如果在集合汇总使用数据量比较大的数据，用vector有一定的优势。

ArrayList和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要设计到数组元素移动到内存操作，所以索引数据快插入数据慢，Vector由于使用了synchronized方法(线程安全)所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据是只需要记录本项的前后即可，所以插入数据较快。

**ArrayList和LinkedList**

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。

对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

对于新增和删除操作add和remove，LinkedList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的速度法尔优于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没插入一条数据，要移动插入点及之后的所有数据。

**HashMap与TreeMap**

HashMap通过hashCode对其内容进行快速查找，而TreeMap中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用TreeMap(HashMap中元素的排列顺序是不固定的)。HashMap中元素的排列顺序是不固定的。

HashMap通过hashCode对其内容进行快速查找，而TreeMap中所有的元素都保持着某种固定的书序，如果你需要得到一个有序的结果你就应该使用TreeMap(HashMap中元素的排列顺序是不固定的)。集合框架提供两种常规的Map实现：HashMap和TreeMap(TreeMap实现SortedMap接口)。

在Map中插入、删除和定位元素，HashMap是最好的选择。但如果你要按自然顺序或自定义顺序遍历键，那么TreeMap会更好。使用HashMap要求添加的键类明确定义了hashCode()和equals()的实现。这个TreeMap没有调优选项，应为该树总处于平衡状态。

**HashTable与HashMap**

历史原因：HashTable是基于陈旧的Dictionary类，HashMap是Java1.2引进的Map接口的一个实现。

同步性：HashTable是线程安全的，也就是说是同步的，而HashMap是线程不安全的，不是同步的。

值：只有HashMap可以让你将控制作为一个表的key或value。

**对集合的选择**

**对List的选择**

对于随机查询与迭代遍历操作，数组比所有容器都要快。所以在随机访问中使用ArrayList。

LinkedList使用双向链表对元素的增加和删除提供了非常好的支持，而ArrayList执行增加和删除元素需要进行元素位移。

对于Vector而已，我们一般都是避免使用。

将ArrayList当做首选，毕竟对于集合元素而已我们都是进行遍历，只有当程序的性能因为List的频繁插入和删除而降低时，在考虑LinkedList。

**对Set的选择**

HashSet由于使用HashCode实现，所以在某种程度上来说它的性能永远比TreeSet好，尤其是进行增加和查找操作。

虽然TreeSet没有HashSet性能好，但是由于它可以维持元素的排序，所以它还是存在永无之地的。

**对Map的选择**

HashMap与HashSet同样，支持快速查询。虽然HashTable速度也不慢，但是在HashMap面前还是稍微慢一些，所以HashMap在查询方面可以取代HashTable。

由于TreeMap需要维持内部元素的顺序，所以它通常要比HashMap和HashTable慢。

## 并发和多线程

多线程(multithreading)是指从软件或者硬件上实现多个线程并发执行的技术。具有多线程能力的计算机应有硬件支持而能够在同一时间执行多于一个线程，进而提升整体处理性能。具有这种能力的系统包括对称多处理机、多核心处理器以及芯片级多处理货同时多线程处理器。在一个程序中，这些独立运行的程序片段叫做线程，利用它编程的概念就叫做“多线程处理”。具有多线程能力的计算机因有硬件支持而能够在同一时间执行多于一个线程，进而提升整体处理性能。

用多线程只有一个目的，那就是更好的利用cpu的资源，因为所有的多线程代码都可以用单线程来实现。说这个话其实只有一半对，因为反应“多角色”的程序代码，最起码每个角色要给他一个线程吧，否则连实际场景都无法模拟，当然也没法说能用单线程来实现：比如常见的“生产者”，消费者模型。

▪ 多线程：指的是这个程序(一个进程)运行时产生了不止一个线程；

▪ 并行与并发

▪ 并行：多个CPU实例或者多台机器同时执行一段处理器逻辑，是真正的同时；

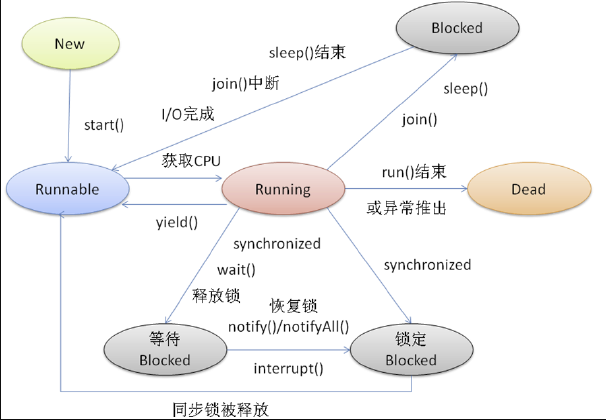
▪ 并发：通过CPU调度算法，让用户看上去同时执行，实际上从CPU操作层面不是真正的同时。并发往往在场景中有公用资源那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。

▪ 线程安全：经常用来描述一段代码。指在并行的情况下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，CPU是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果，如不加事务的转账代码：

|  |
| --- |
| void transferMoney(User from, User to, float amount){  to.setMoney(to.getBalance() + amount);  from.setMoney(from.getBalance() - amount);  } |

▪ 同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入@synchronized关键字。在保证结果准确的同时，提供性能，才是优秀的程序。线程安全的优先级高于性能。

**线程的状态**



线程状态转换各种状态一目了然，值得一提的是“blocked”这个状态：线程在Running的过程中可能会遇到阻塞(Blocked)情况。

▪ 调用join()和sleep()方法，sleep()时间结束或被打断，join()终端，IO完成都会回到 Runnable状态，等待JVM的调度。

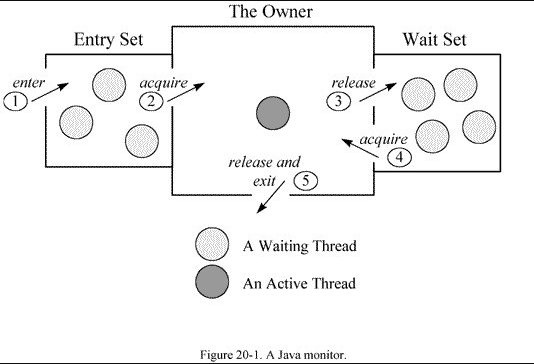
▪ 调用wait()，使该线程处于等待池，直到notify()/notifyAll(),线程被唤醒被放到锁定池，释放同步锁使线程回到可运行状态。

▪ 对Running状态的线程加同步锁使其进入锁定池，同步锁被释放进入可运行状态。

此外，在Runnable状态的线程是处理被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入Runnable。

**每个对象都有的方法(机制)**

synchronized,wait,notify是任何对象都具有的同步工具。让我们先来了解它们。



monitor它们是应用于同步问题的人工线程调度工具。讲其本质，首先就要明确monitor的概念，Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized范围内，监视器发挥作用。

wait/notify必须存在于synchronized快中。并且，这三个关键字针对的是同一个监视器(某对象的监视器)。这意味着wait之后，其他线程可以进入同步块执行。

当某代码并不持有监视器的使用权时(如图中5的状态，即脱离同步块)去wait或notify，就会抛出java.lang.IllegalMonitorStateException。也包括在synchronized块中去调用另一个对象的wait/notify，因为不同对象的监视器不同，同样会抛出异常。

▪ synchronized单独使用：

代码块：如下，在多线程环境下，synchronized块汇总的方法获取了lock实例的monitor，如果实例相同，那么只有一个线程能执行该块内容

|  |
| --- |
| public class Thread1 implements Runnable {  Object lock;  public void run() {  synchronized(lock){  ..do something  }  }  } |

直接用于方法：相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类的所有实例。

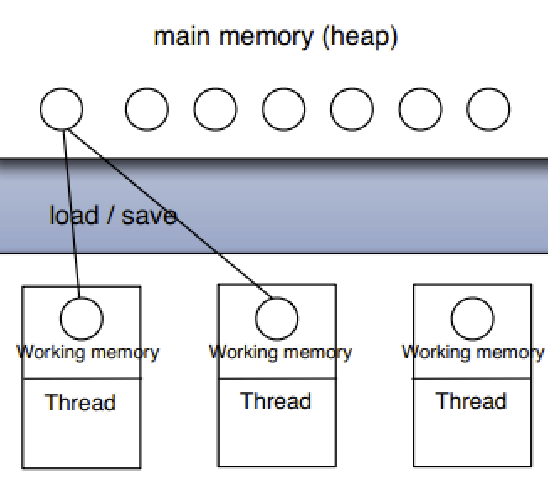
|  |
| --- |
| public class Thread1 implements Runnable {  public synchronized void run() {  ..do something  }  } |

▪ synchronized，wait，notify结合：典型场景生产者消费者问题

|  |
| --- |
| /\*\*  \* 生产者生产出来的产品交给店员  \*/  public synchronized void produce()  {  if(this.product >= MAX\_PRODUCT)  {  try  {  wait();  System.out.println("产品已满,请稍候再生产");  }  catch(InterruptedException e)  {  e.printStackTrace();  }  return;  }    this.product++;  System.out.println("生产者生产第" + this.product + "个产品.");  notifyAll(); //通知等待区的消费者可以取出产品了  }    /\*\*  \* 消费者从店员取产品  \*/  public synchronized void consume()  {  if(this.product <= MIN\_PRODUCT)  {  try  {  wait();  System.out.println("缺货,稍候再取");  }  catch (InterruptedException e)  {  e.printStackTrace();  }  return;  }    System.out.println("消费者取走了第" + this.product + "个产品.");  this.product--;  notifyAll(); //通知等待去的生产者可以生产产品了  } |

volatile：

多线程的内存模型：main memory(主存)、working memory(线程栈)，在处理数据时，线程会把值从主存load到本地栈完成操作后在save回去前(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。



volatile：

针对多线程使用的变量如果不是volatile或者final修饰的，很有可能产生不可预知的结果(另一个线程修改了这个值，但是之后在某个线程看到的是修改之前的值)。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile就是不去缓存，直接取值。在线程安全的情况下加volatile会牺牲性能。

**基本线程类**

基本线程类指的是Thread类，Runnable接口，Callable接口

Thread类实现了Runnable接口，启动一个线程的方法：

|  |
| --- |
| MyThread my = new MyThread();  　　my.start(); |

Thread类相关方法：

|  |
| --- |
| //当前线程可转让cpu控制权，让别的就绪状态线程运行（切换）  public static Thread.yield()  //暂停一段时间  public static Thread.sleep()  //在一个线程中调用other.join(),将等待other执行完后才继续本线程。  public join()  //后两个函数皆可以被打断  public interrupte() |

关于中断：

它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识为，以判断线程是否应该被中断(中断标识值是否为true)。中断只会影响到wait、sleep、join状态。被打断的线程会抛出InterruptedException。

Thread.interrupted()检查当前线程是否发生中断，返回boolean

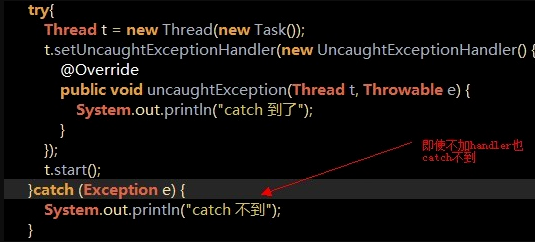
synchronized在获锁的过程中是不能被中断的。

中断是一个状态interrupt()方法知识将这个状态置为true而已。所以说正常运行的程序不去检测状态就不会中止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted())，则同样可以在中断后离开代码体。

Tread类最佳实践：

写的时候最好要设置线程名称Thread.name，并设置线程组ThreadGroup，目的是方便管理。在出现问题的时候，打印线程栈一眼就可以看出是哪个线程出的问题，这个线程是什么的。

如何获取线程中的异常：



不能用try，catch来获取线程中的异常

Runnable

与Thread类似

Callable

future模式：并发模型的一种，可以有两种形式，即无阻塞和阻塞，分别是isDone和get。其中Future对象用来存放该线程的返回值以及状态。

|  |
| --- |
| ExecutorService e = Executors.newFixedThreadPool(3);  //submit方法有多重参数版本，及支持callable也能够支持runnable接口类型.  Future future = e.submit(new myCallable());  future.isDone() //return true,false 无阻塞  future.get() // return 返回值，阻塞直到该线程运行结束 |

**高级多线程控制类**

以上都属于内功心法接下来是实际项目中常用到的工具了，Java1.5提供了一个非常搞笑使用的多线程包：java.util.concurrent，提供了大量高级工具，可以帮助开发者编写搞笑、易维护、结构清晰的Java多线程程序。

ThreadLocal类：

用处：保存线程的独立变量。对一个线程类。

但是用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。常用于用户登录控制，如记录session信息。

实现：每个Thread都持有一个ThreadLocalMap类型的变量(该类是一个轻量级的Map，功能与Map一样，区别是桶里放的是entry而不是entry的链表。功能还是一个map)以本身为key，以目标为value。

主要方法是get()和set(T a)，set之后在map之后在map里维护一个ThreadLocal->a,get时将a返回。

ThreadLocal是一个特殊的容器。

**原子类(AtomicInteger、AtomicBoolean…)**

如果使用atomic wrapper class如atomicInteger，或者使用自己保证原子的操作，则等同于synchronized

|  |
| --- |
| //返回值为boolean  AtomicInteger.compareAndSet(int expect,int update) |

该方法可用于实现乐观锁，考虑文中最初提到的如下场景：a给b付款10元，a扣了10元，b要加10元。此时c给b2元，但是b的加十元代码约为：

|  |
| --- |
| if(b.value.compareAndSet(old, value)){  return ;  }else{  //try again  // if that fails, rollback and log  } |

AtomicReference

对于AtomicReference 来讲，也许对象会出现，属性丢失的情况，即oldObject == current，但是oldObject.getPropertyA != current.getPropertyA。  
这时候，AtomicStampedReference就派上用场了。这也是一个很常用的思路，即加上版本号。

Lock类

lock：在java.util.concurrent包内。共有三个实现

▪ ReentrantLock

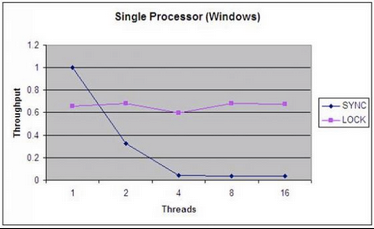
▪ ReentrantReadWriteLock.ReadLock

▪ ReentrantReadWriteLock.WriteLock

主要目的是和synchronized一样， 两者都是为了解决同步问题，处理资源争端而产生的技术。功能类似但有一些区别。

区别如下：

1. lock更灵活，可以自由定义多把锁的枷锁解锁顺序（synchronized要按照先加的后解顺序）
2. 提供多种加锁方案，lock 阻塞式, trylock 无阻塞式, lockInterruptily 可打断式， 还有trylock的带超时时间版本。
3. 本质上和监视器锁（即synchronized是一样的）
4. 能力越大，责任越大，必须控制好加锁和解锁，否则会导致灾难。
5. 和Condition类的结合。
6. 性能更高，对比如下图：



synchronized和Lock性能对比

ReentrantLock

可重入的意义在于持有锁的线程可以继续持有，并且要释放对等的次数后才真正释放该锁。  
使用方法是：

1. 先new一个实例

|  |
| --- |
| static ReentrantLock r=new ReentrantLock(); |

1. 加锁

|  |
| --- |
| r.lock()或r.lockInterruptibly(); |

1. 释放锁

|  |
| --- |
| r.unlock() |

必须做！何为必须做呢，要放在finally里面。以防止异常跳出了正常流程，导致灾难。这里补充一个小知识点，finally是可以信任的：经过测试，哪怕是发生了OutofMemoryError，finally块中的语句执行也能够得到保证。

ReentrantReadWriteLock

可重入读写锁（读写锁的一个实现）

|  |
| --- |
| ReentrantReadWriteLock lock = new ReentrantReadWriteLock()  　　ReadLock r = lock.readLock();  　　WriteLock w = lock.writeLock(); |

两者都有lock,unlock方法。写写，写读互斥；读读不互斥。可以实现并发读的高效线程安全代码

**容器类**

这里就讨论比较常用的两个：

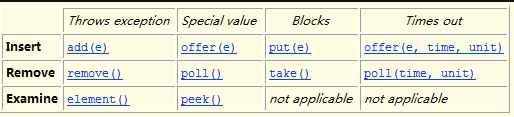
▪ BlockingQueue

▪ ConcurrentHashMap

BlockingQueue

阻塞队列。该类是java.util.concurrent包下的重要类，通过对Queue的学习可以得知，这个queue是单向队列，可以在队列头添加元素和在队尾删除或取出元素。类似于一个管　　道，特别适用于先进先出策略的一些应用场景。普通的queue接口主要实现有PriorityQueue（优先队列），有兴趣可以研究

BlockingQueue在队列的基础上添加了多线程协作的功能：



BlockingQueue

除了传统的queue功能（表格左边的两列）之外，还提供了阻塞接口put和take，带超时功能的阻塞接口offer和poll。put会在队列满的时候阻塞，直到有空间时被唤醒；take在队　列空的时候阻塞，直到有东西拿的时候才被唤醒。用于生产者-消费者模型尤其好用，堪称神器。

常用的阻塞队列有：

▪ ArrayListBlockingQueue

▪ LinkedListBlockingQueue

▪ DelayQueue

▪ SynchronousQueue

ConcurrentHashMap

高效的线程安全哈希map。请对比hashTable , concurrentHashMap, HashMap

**管理类**

管理类的概念比较泛，用于管理线程，本身不是多线程的，但提供了一些机制来利用上述的工具做一些封装。  
了解到的值得一提的管理类：ThreadPoolExecutor和 JMX框架下的系统级管理类 ThreadMXBean  
**ThreadPoolExecutor**  
如果不了解这个类，应该了解前面提到的ExecutorService，开一个自己的线程池非常方便：

|  |
| --- |
| ExecutorService e = Executors.newCachedThreadPool();  ExecutorService e = Executors.newSingleThreadExecutor();  ExecutorService e = Executors.newFixedThreadPool(3);  // 第一种是可变大小线程池，按照任务数来分配线程，  // 第二种是单线程池，相当于FixedThreadPool(1)  // 第三种是固定大小线程池。  // 然后运行  e.execute(new MyRunnableImpl()); |

该类内部是通过ThreadPoolExecutor实现的，掌握该类有助于理解线程池的管理，本质上，他们都是ThreadPoolExecutor类的各种实现版本。请参见javadoc：

ThreadPoolExecutor参数解释

翻译一下：  
**corePoolSize**:池内线程初始值与最小值，就算是空闲状态，也会保持该数量线程。  
**maximumPoolSize**:线程最大值，线程的增长始终不会超过该值。  
**keepAliveTime**：当池内线程数高于corePoolSize时，经过多少时间多余的空闲线程才会被回收。回收前处于wait状态  
**unit**：  
时间单位，可以使用TimeUnit的实例，如TimeUnit.MILLISECONDS  
**workQueue**:待入任务（Runnable）的等待场所，该参数主要影响调度策略，如公平与否，是否产生饿死(starving)  
**threadFactory**:线程工厂类，有默认实现，如果有自定义的需要则需要自己实现ThreadFactory接口并作为参数传入。

乐观锁和悲观锁：

## 1.11 流与文件操作

## 1.12 XML

## 1.13 网络编程

Socket/client

OSI七层模型：

|  |  |  |
| --- | --- | --- |
| OSI层 | 功能 | 协议 |
| 应用层 | 文件传输 电子邮件 文件服务 虚拟终端 | TFTP HTTP SNMP FTP SMTP  DNS TELNET |
| 表示层 | 数据格式化 代码转换 数据加密 |  |
| 会话层 | 解除或建立与其他连接点的联系 |  |
| 传输层 | 提供端对端的接口 | TCP UDP |
| 网络层 | 为数据包选择路由 | IP ICMP RIP OSPF BGP IGMP |
| 数据链路层 | 传输有地址的帧 错误检测功能 | SLIP CSLIP PPP ARP RARP MTU |
| 物理层 | 以二进制数据形式在物理媒体上传输数据 | ISO2110 IEEE802 IEEE802.2 |

五层模型：

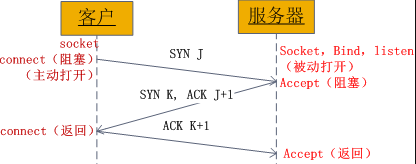
|  |  |
| --- | --- |
| TCP/IP层 | 网络设备 |
| 应用层 |  |
| 传输层 | 四层交换机 也有工作在四层的路由器 |
| 网络层 | 路由器 三层交换机 |
| 数据链路层 | 网桥 以太网交换机 网卡 |
| 物理层 | 中继器 集线器 双绞线 |

其实socket是tcp/ip协议封装的一个api的协议不是同一概念。socket是对tcp/ip协议的封装和应用，tcp/ip协议是传输层协议，主要解决数据如何在网络中传输。

1. 服务器监听：服务器端套接字并不定位具体的客户端套接字，而是出于等待连接的状态，实时监控网络状态，等待客户端的连接请求。
2. 客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，支出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。
3. 连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，简历一个新的线程，把服务端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。

三次握手：

1. 客户端向服务器发送一个SYN J
2. 服务器向客户响应一个SYN K, 并对SYN J 进行确认ACK J+1
3. 客户端再向服务器发送一个确认ACK K+1



从图中可以看出，当客户端调用connect时，出发了连接请求，向服务器发送了SYN J包，这是connect进入阻塞状态；服务器监听到连接请求，即受到SYN J包，调用accept函数接收请求向客户端发送SYN K, ACK J +1,这是accept进入阻塞状态；客户端受到服务器的SYN K, ACK J+1之后，这时connect返回，并对SYN K进行确认；服务器受到ACK K+1时，accept返回，至此三次握手完毕，连接建立。

TCP/IP和UDP的区别

1. TCP是面向链接的，虽然说网络的不安全不稳定特性决定了多少次握手都不能保证连接的可靠性，但TCP的三次握手在最低限度上保证了连接的可靠性；
2. 也正由于a所说的特定，使得UDP的开销更小数据传输速率更高，因为不必进行发送数据的确认，所以UDP的实时性更好。

## 1.14 数据库编程

## 1.15 国际化

## 1.16 JavaBean

## 1.17 安全(类加载、签名、加密)

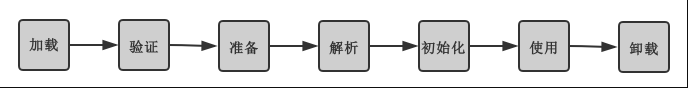
## 1.18 分布式对象

## 1.19 本地方法

## 1.20 JVM

类加载：

Java源代码被变异成class字节码，最终需要加载到虚拟机中才能运行。整个生命周期包括：加载、验证、准备、解析、初始化、使用、卸载7个阶段。



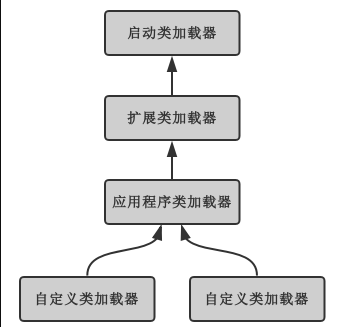
加载：

1. 通过一个类的全限定名获取描述此类的二进制字节流；
2. 将这个字节流所代表的静态存储结构保存为方法去的运行时数据结构；
3. 在Java堆中生成一个代表这个类的java.lang.Class对象，作为访问方法区入口；

虚拟机设计团队吧加载动作放到JVM外部实现，以便让应用程序决定如何获取所需的类，实现这个动作的代码成为“类加载器”，JVM提供了3中类加载器：

1. 启动类加载器(Bootstrap ClassLoader)：负责加载JAVA\_HOME/lib目录中的，或通过Xbootclasspath参数指定路径中的，且被虚拟机认可的类。
2. 扩展类加载器(Extension ClassLoader)：负责加载JAVA\_HOME/lib/ext目录中的，或通过java.ext.dirs系统变量指定路径中的类库。
3. 应用程序类加载器(Application ClassLoder)：负责加载用户路径上的类库。

JVM 基于上述加载器，通过双亲委托模式进行类的加载，当然我们也可以通过继承java.lang.ClassLoader实现自定义的类加载器。



双亲委派模式工作工程：当一个类加载器收到类加载任务，有限交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。

双亲委派模式有什么好处？

比如位于rt.jar包中的类java.lang.Object，无论哪个加载器加载这个类，最终都是委托给顶层的启动器加载器进行加载，确保了Object类在各种加载器环境中都是同一个类。

验证

为了确保Class文件符合当前虚拟机要求，需要对其字节流数据进行验证，主要包括格式验证、元数据验证、字节码验证和符号引用验证。

1. 格式验证

验证自己留是否符合class文件格式的规范，并且能被当前虚拟机处理，如是否已魔数0xCAFEBABE开头，主次版本号是否在当前虚拟机处理范围内，常量池是否有不支持的常量类型等。只有经过格式验证的字节流，才会存储到方法区的数据结构，剩余3个验证都基于方法区的数据进行。

1. 元数据验证

对字节码描述的数据进行语义分析，以保证符合Java语言规范，如是否继承了final修饰的类、是否实现了父类的抽象方法、是否覆盖了父类的final方法或final字段等。

1. 字节码验证

对类的方法体进行分析，确保在方法运行时不会有危害虚拟机的事件发生，如保证操作数据栈的数据类型和指令代码序列的匹配，保证跳转指令的正确性，保证保证类型转换的有效性等。

1. 符号引用验证

为了确保后续的解析动作能够正常执行，对符号引用进行验证，如通过字符串描述的全限定名是都能找到对应的类，在指定类中是否存在符合方法的字段描述等。

准备：

在准备阶段，为类变量(static修饰)在方法区中分配内存并设置初始值。

|  |
| --- |
| private static int var = 100; |

准备阶段完成后，var值为0，而不是100.在初始化阶段，才会把100赋值给val，但是有个特殊情况。

|  |
| --- |
| private static final int VAL= 100; |

在编译阶段会为val生成ConstantValue属性，在准备阶段虚拟机会根据Constantvalue 属性将val赋值为100.

解析：

解析阶段是将常量池中的符号引用替换为直接引用的过程，符号引用和直接引用有什么不同？

1. 符号引用使用一组符号来描述所引用的目标，可以是任何形式的字面量，定义在Class文件格式中。
2. 直接引用可以是直接指向目标的指针，相对偏移量或能间接定位到目标的句柄。

初始化

初始化阶段是执行类构造器<clinit>方法的过程，<clinit>方法由类变量的复制动作和静态语句块按照在源文件出现的顺序合并而成，该合并操作由编译器完成。

|  |
| --- |
| private static int value = 100;  static int a = 100;  static int b = 100;  static int c;    static {  c = a + b;  System.out.println("it only run once");  } |

1. <clinit>方法对于类或接口不是必须的，如果一个类中没有静态代码块，也没有静态变量的复制操作，那么编译器不会生成<clinit>；
2. <clinit>方法与实例构造器不同，不需要显示的调用父类的<clinit>方法，虚拟机会保证父类的<clinit>有限执行；
3. 为了防止多次执行<clinit>，虚拟机会确保<clinit>方法在多线程环境下被正确的加锁同步执行，如果有多个线程同时初始化一个类，那么只有一个线程能够执行<clinit>方法，其他线程进行阻塞等待，直到<clinit>执行完成。
4. 注意：执行接口的<clinit>方法不需要先执行父接口的<clinit>，只有使用父接口中定义的变量时，才会执行。

类初始化场景

虚拟机中严格规定了有且只有5中情况必须对类进行初始化。

1. 执行new、getinstance、putstatic、invokestatic指令；
2. 使用reflect对类进行反射；
3. 初始化一个类的时候，父类还没有初始化，会实现初始化父类；
4. 启动虚拟机时，需要初始化包含main方法的类；
5. 在JDK1.7中，如果java.lang.invoke.MethodHandler实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄对应的类没有进行初始化。

# Part2. Java常用框架

## SpringMVC

//TODO

## 2.2 Struts2

## 2.3 Spring

## 2.4 SpringBoot

//TODO

## 2.5 MyBatis

## 2.6 Hibernate

Hibernate

## 2.7 Servlet

# Part3. Java常用中间件

## 3.1 Tomcat

## 3.2 WebLogic

## 3.3 WebSphere

## 3.4 ZooKeeper

//TODO

## Kafka

//TODO

## 3.6 MQ

//TODO

# Part 4. 内存数据库

## 4.1 Redis

## 4.2 Memcached

## 4.3 H2

h2

# Part 5. 数据库连接池和中间件

## 5.1 数据库连接池

### 5.1.1 c3p0

### 5.1.2 druid

### 5.1.3 BoneCP

### 5.1.4 HiKariCP

## 5.2 数据库中间件

### 5.2.1 Cobar(MySQL-Alibaba)

### 5.2.2 MyCAT(base-Cobar modify)

### 5.2.3 TDDL(Alibaba)

### 5.2.4 Sharding-JDBC

### 5.2.5 Oceanus(58)

### 5.2.6 MySQL Route(MySQL official)

MySQL Route

# Part 6. 关系型数据库

## 6.1 MySQL

## 6.2 Oracle

## 6.3 SQL Server

## 6.4 DB2

db2

# Part 7. 非关系型数据库

## 7.1 MongoDB

//TODO

## 7.2 HBase

//TODO

# Part 8. 分布式数据库

## 8.1 Hive

hive

# Part 9. 微服务

## 9.1 dubbo

//TODO

## 9.2 SpringCloud

//TODO

# Part 10. 其他

## 10.1 数据校验

## 10.2 序列化

## 10.3 加解密