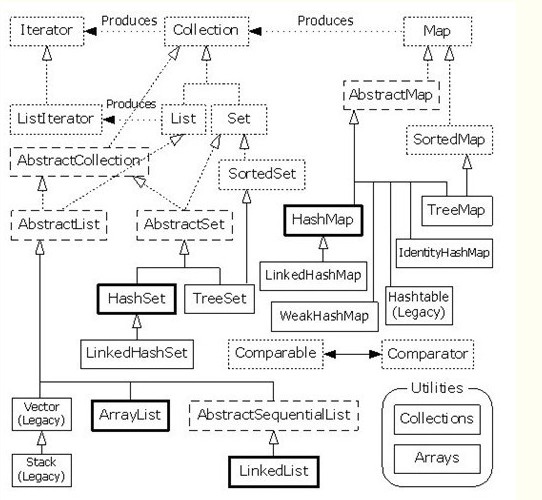
# Java Web学习笔记

## Java基础

### Java 集合



#### ① Collection接口

Collection接口是 Set 、List 和 Queue 接口的父接口，提供了多数集合常用的方法声明，包括 add()、remove()、contains() 、size() 、iterator() 等。

1. List

List 关心的是索引，与其他集合相比，List特有的就是和索引相关的一些方法：get(int index) 、 add(int index,Object o) 、 indexOf(Object o) ；List是有序的Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组；另外List允许有相同的元素。

**ArrayList** 可以将它理解成一个可增长的数组，默认初始大小为10. 成1.5倍加1增长。它提供快速迭代和快速随机访问的能力，其底层使用数组实现的，查询效率较高，但是增删效率低且是线程不安全的。

**LinkedList** 中的元素之间是双链接的，当需要快速插入和删除时LinkedList成为List中的不二选择，其底层使用双向循环链表实现，查询效率低，但增删元素效率比较高。

**Vector** 是ArrayList的线程安全版本，性能比ArrayList要低，现在已经很少使用。

1. Set

**Set关心唯一性，它不允许重复。**

HashSet**当不希望集合中有重复值，并且不关心元素之间的顺序时可以使用此类，使用哈希**[算法](http://lib.csdn.net/base/datastructure)**实现。**

LinkedHashset**当不希望集合中有重复值，并且希望按照元素的插入顺序进行迭代遍历时可采用此类。**

TreeSet**当不希望集合中有重复值，并且希望按照元素的自然顺序进行排序时可以采用此类。（自然顺序意思是某种和插入顺序无关，而是和元素本身的内容和特质有关的排序方式，譬如“abc”排在“abd”前面。）**

1. Queue

Queue用于保存将要执行的任务列表。

**LinkedList** 同样实现了Queue接口，可以实现先进先出的队列。

**PriorityQueue** 用来创建自然排序的优先级队列。

#### ② Map接口

Map关心的是唯一的标识符。他将唯一的键映射到某个元素。当然键和值都是对象。

**HashMap** 当需要键值对表示，又不关心顺序时可采用HashMap，线程不安全，允许key或value为NULL。

**Hashtable** 注意Hashtable中的t是小写的，它是HashMap的线程安全版本，不允许key或value为NULL。

**LinkedHashMap** 当需要键值对，并且关心插入顺序时可采用它。

**TreeMap** 当需要键值对，并关心元素的自然排序时可采用它。

### JAVA 泛型

### JAVA 异常

**① JAVA异常结构**：



**② 异常说明：**

**Throwable：**有两个重要的子类：Exception（异常）和 Error（错误）

**Error（错误）:** 是程序无法处理的错误，表示运行应用程序中较严重问题，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError

**Exception（异常）:**是程序本身可以处理的异常。Java的异常(包括Exception和Error)分为**可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）**。

**运行时异常：**都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常

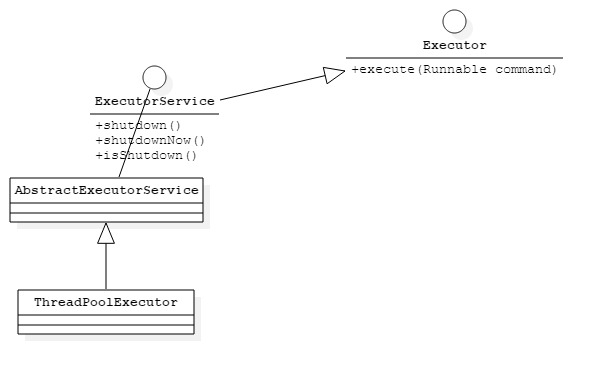
**非运行时异常 （编译异常）：**是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常。

### JAVA IO、NIO

### JAVA 线程

## JAVA 高级

### JAVA线程池



Java通过Executors提供四种线程池，分别为：  
**newCachedThreadPool**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** 创建一个定长线程池，支持定时及周期性任务执行。  
**newSingleThreadExecutor** 创建一个单线程化的线程池，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。

#### newCachedThreadPool

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。示例代码如下：

1. ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. **try** {
5. Thread.sleep(index \* 1000);
6. } **catch** (InterruptedException e) {
7. e.printStackTrace();
8. }
10. cachedThreadPool.execute(**new** Runnable() {
12. @Override
13. **public** **void** run() {
14. System.out.println(index);
15. }
16. });
17. }

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

#### 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。示例代码如下:

1. ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. fixedThreadPool.execute(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. System.out.println(index);
10. Thread.sleep(2000);
11. } **catch** (InterruptedException e) {
12. // TODO Auto-generated catch block
13. e.printStackTrace();
14. }
15. }
16. });
17. }

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。  
定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()。可参考PreloadDataCache。

#### newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

1. ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
2. scheduledThreadPool.schedule(**new** Runnable() {
4. @Override
5. **public** **void** run() {
6. System.out.println("delay 3 seconds");
7. }
8. }, 3, TimeUnit.SECONDS);

表示延迟3秒执行。  
   
定期执行示例代码如下：

scheduledThreadPool.scheduleAtFixedRate(**new** Runnable() {

1. @Override
2. **public** **void** run() {
3. System.out.println("delay 1 seconds, and excute every 3 seconds");
4. }
5. }, 1, 3, TimeUnit.SECONDS);

表示延迟1秒后每3秒执行一次。  
ScheduledExecutorService比Timer更安全，功能更强大。

#### newSingleThreadExecutor

创建一个单线程化的线程池，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。示例代码如下：

1. ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. singleThreadExecutor.execute(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. System.out.println(index);
10. Thread.sleep(2000);
11. } **catch** (InterruptedException e) {
12. // TODO Auto-generated catch block
13. e.printStackTrace();
14. }
15. }
16. });
17. }

结果依次输出，相当于顺序执行各个任务。  
现行大多数GUI程序都是单线程的。Android中单线程可用于数据库操作，文件操作，应用批量安装，应用批量删除等不适合并发但可能IO阻塞性及影响UI线程响应的操作。

### 设计模式

① **静态代理和动态代理**

动态代理相对于静态代理在使用上的优点主要是能够对一个对象的所有方法进行统一包装，而且后期被代理的类添加方法的时候动态代理类不需要改动。

缺点是要求被代理的类必须实现了接口，因为动态代理类在实现的时候继承了Proxy类，java不支持多继承，因此动态代理类只能根据接口来定义方法。

最后动态代理之所以叫做动态代理是因为java在实现动态代理的时候，动态代理类是在运行时动态生成和加载的，相对的，静态代理类和其他普通类一下，在类加载阶段就加载了。

### JAVA 并发编程

## JAVA框架

### Spring

#### 控制反转（IOC）

#### 面向切面（AOP）

### Spring MVC

### Struts2

### MyBatis

### Hibernate

## 数据库（MySQL、Oracle）

### SQL基础

#### 索引

#### 视图

#### 存储过程

#### 触发器

### SQL 优化

### 乐观锁和悲观锁

### 分区和分表

### 事务

 ① 事务的ACID属性：

    1. 原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。  
    2. 一致性（Consistency）

 事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

3. 隔离性（Isolation）

 事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

4. 持久性（Durability）  
    持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。

② 事务隔离级别：

**读操作未提交（Read Uncommitted）**：读取未提交的数据是允许的。说明一个事务在提交前，其变化对于其他事务来说是可见的。这样脏读、不可重读和幻读都是允许的。当一个事务已经写入一行数据但未提交，其他事务都不能再写入此行数据；但是，任何事务都可以读任何数据。这个隔离级别使用排写锁实现。

**读操作已提交（Read Committed）**：读取未提交的数据是不允许的，它使用临时的共读锁和排写锁实现。这种隔离级别不允许脏读，但不可重读和幻读是允许的。

**可重读（Repeatable Read）**：说明事务保证能够再次读取相同的数据而不会失败。此隔离级别不允许脏读和不可重读，但幻读会出现。

**可串行化（Serializable）**：提供最严格的事务隔离。这个隔离级别不允许事务并行执行，只允许串行执行。这样，脏读、不可重读或幻读都可发生。

**1. 1事务隔离与隔离级别的关系**

|  |  |  |  |
| --- | --- | --- | --- |
| **隔离级别** | **脏读（Dirty Read）** | **不可重读（Unrepeatable read）** | **幻读（Phantom Read）** |
| 读操作未提交（Read Uncommitted） | 可能 | 可能 | 可能 |
| 读操作已提交（Read Committed） | 不可能 | 可能 | 可能 |
| 可重读（Repeatable Read） | 不可能 | 不可能 | 可能 |
| 可串行化（Serializable） | 不可能 | 不可能 | 不可能 |

③ 事务的传播机制：

|  |  |
| --- | --- |
| **事务传播行为类型** | **说明** |
| **PROPAGATION\_REQUIRED** | **如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是 最常见的选择。** |
| **PROPAGATION\_SUPPORTS** | **支持当前事务，如果当前没有事务，就以非事务方式执行。** |
| **PROPAGATION\_MANDATORY** | **使用当前的事务，如果当前没有事务，就抛出异常。** |
| **PROPAGATION\_REQUIRES\_NEW** | **新建事务，如果当前存在事务，把当前事务挂起。** |
| **PROPAGATION\_NOT\_SUPPORTED** | **以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。** |
| **PROPAGATION\_NEVER** | **以非事务方式执行，如果当前存在事务，则抛出异常。** |
| **PROPAGATION\_NESTED** | **如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与 PROPAGATION\_REQUIRED 类似的操作。** |

当使用 PROPAGATION\_NESTED 时， 底层的数据源必须基于 JDBC 3.0 ，并且实现者需要支持保存点事务机制。

**readOnly**事务属性中的readOnly标志表示对应的事务应该被最优化为只读事务。这是一个最优化提示 。在一些情况下，一些事务策略能够起到显著的最优化效果，例如在使用Object/Relational映射工具 （如：Hibernate或TopLink）时避免dirty checking（试图“刷新”）。

**Timeout** 在事务属性中还有定义“timeout”值的选项，指定事务超时为几秒。在JTA中，这将被简单地传递到J2EE服务器的事务协调程序，并据此得到相应的解释。

## 缓存（Redis）

## 消息中间件（MQ）

## 网络编程（WebService）

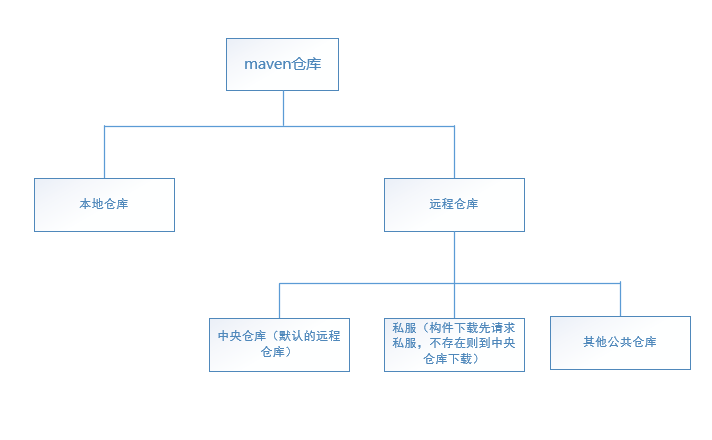
## 工具（Maven,Git）

① maven依赖范围Scope：compile、test、provided、runtime、system。

依赖原则：

1. 路径优先原则
2. 第一声明优先

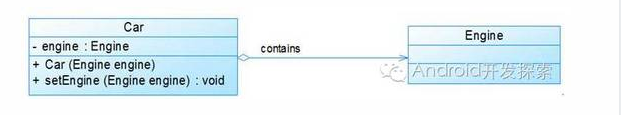
② maven仓库的结构



## UML基础

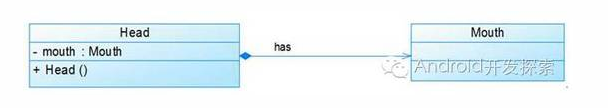
① 聚合关系和组合关系的区别：

聚合关系



上 图中的Car类与Engine类就是聚合关系（Car类中包含一个Engine类型的成员变量）。由上图我们可以看到，UML中 聚合关系用带空心菱形和 箭头的直线表示。聚合关系强调是“整体”包含“部分”，但是“部分”可以脱离“整体”而单独存在。比如上图中汽车包含了发动机，而发动机脱离了汽车也能够 单独存在。

组合关系



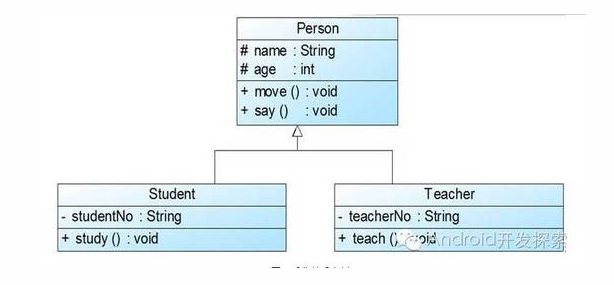
组合关系与聚合关系间的最大不同在于：这里的“部分”脱离了“整体”便不复存在。比如下图：

显然，嘴是头的一部分且不能脱离了头而单独存在。在UML类图中，组合关系用一个带实心菱形和箭头的直线表示。

② 继承关系和接口实现关系

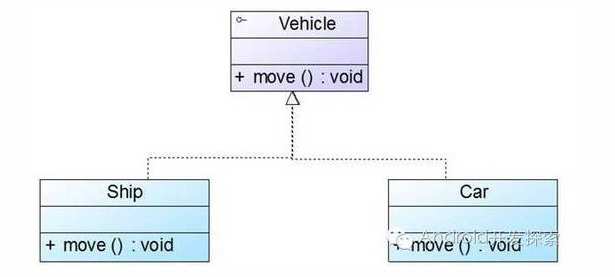
继承关系：

继承关系对应的是extend关键字，在UML类图中用带空心三角形的直线表示，如下图所示中，Student类与Teacher类继承了Person类。

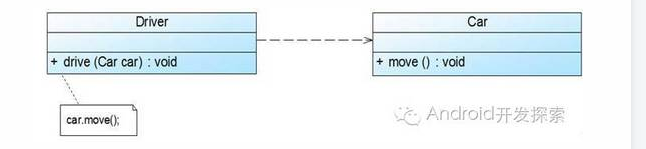


接口实现关系：

这种关系对应implements关键字，比如下图中，Car类与Ship类都实现了Vehicle接口。



③ 依赖关系



从上图中我们可以看到，Driver的drive方法只有传入了一个Car对象才能发挥作用，因此我们说Driver类依赖于Car类。在UML类图中，依赖关系用一条带有箭头的虚线表示。