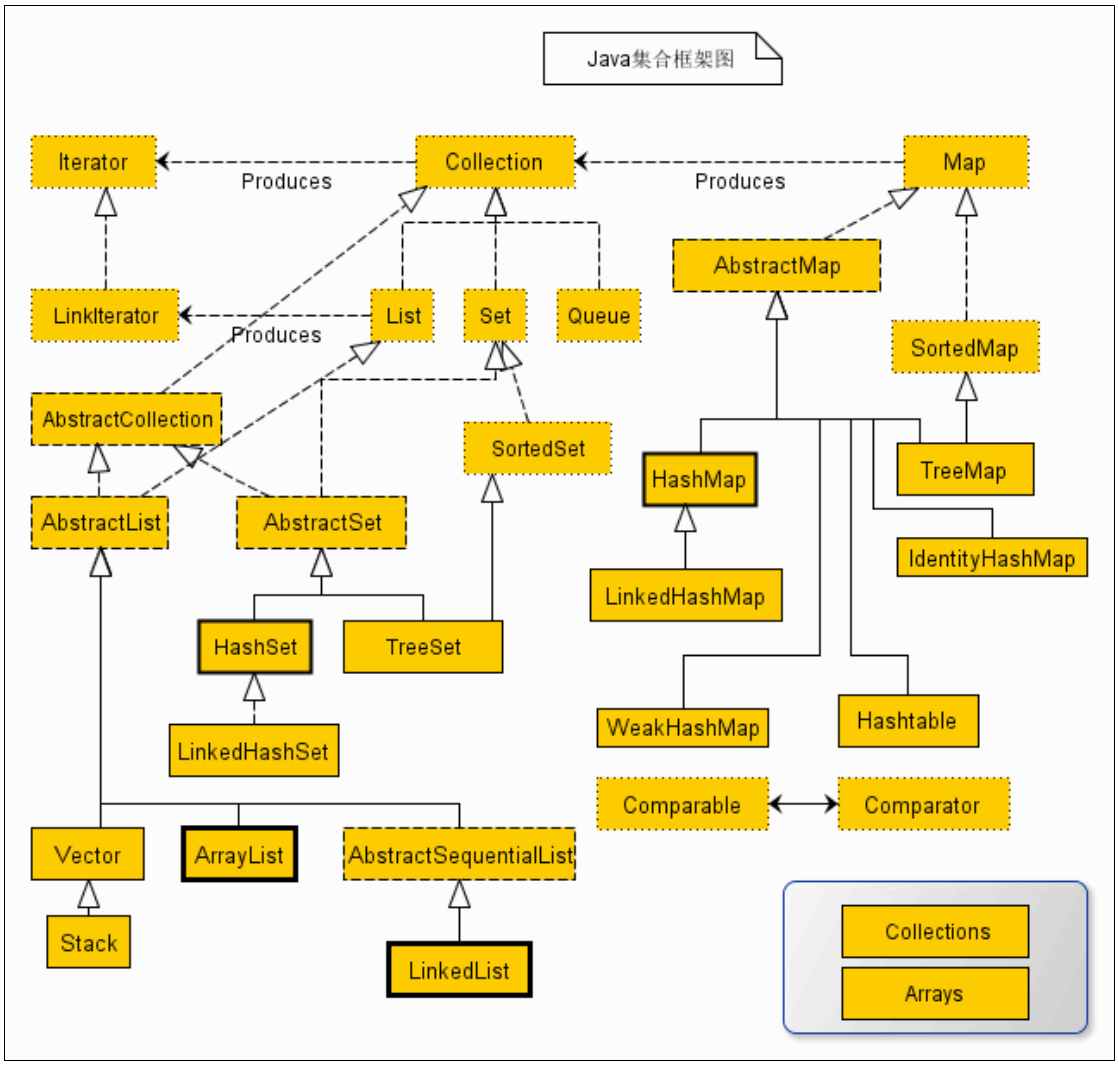
Java 基础增强

目录

1、集合框架

1.1、集合框架体系图



各集合框架概述：

Java 的集合框架主要分为五大类体系：

1、Collection（常用的 List 和 Set，和不常用的 Queue 和 Vector 和 Stack），单元素集合

2、Map（常用的 HashMap 和 TreeMap，不常用的 HashTable），Key-Value 映射

3、Iterator（迭代器）

4、工具类（Collections 和 Arrays）

5、Comparable 和 Comparator 比较器

Java 中的集合和数组的区别：

1、数组长度在初始化时指定，意味着只能保存定长的数据。而集合可以保存数量不确定的数据。同时可以保存具有映射关系的数据（即关联数组，键值对 key-value）。

2、数组元素即可以是基本类型的值，也可以是对象。集合里只能保存对象（实际上只是保

存对象的引用变量），基本数据类型的变量要转换成对应的包装类才能放入集合类中。

Collection 接口中的方法：

方法摘要	
boolean	**add**(E e) 确保此 collection 包含指定的元素（可选操作）。
boolean	**addAll**(Collection<? extends E> c) 将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。
void	**clear**() 移除此 collection 中的所有元素（可选操作）。
boolean	**contains**(Object o) 如果此 collection 包含指定的元素，则返回 true。
boolean	**containsAll**(Collection<?> c) 如果此 collection 包含指定 collection 中的所有元素，则返回 true。
boolean	**equals**(Object o) 比较此 collection 与指定对象是否相等。
int	**hashCode**() 返回此 collection 的哈希码值。
boolean	**isEmpty**() 如果此 collection 不包含元素，则返回 true。
Iterator<E>	**iterator**() 返回在此 collection 的元素上进行迭代的迭代器。
boolean	**remove**(Object o) 从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。
boolean	**removeAll**(Collection<?> c) 移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。
boolean	**retainAll**(Collection<?> c) 仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。
int	**size**() 返回此 collection 中的元素数。
Object[]	**toArray**() 返回包含此 collection 中所有元素的数组。
<T> T[]	**toArray**(T[] a) 返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。

Map 接口中的方法：

方法摘要	
void	**clear**() 从此映射中移除所有映射关系（可选操作）。
boolean	**containsKey**(Object key) 如果此映射包含指定键的映射关系，则返回 true。
boolean	**containsValue**(Object value) 如果此映射将一个或多个键映射到指定值，则返回 true。
Set<Map.Entry<K,V>>	**entrySet**() 返回此映射中包含的映射关系的 Set 视图。
boolean	**equals**(Object o) 比较指定的对象与此映射是否相等。
V	**get**(Object key) 返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。
int	**hashCode**() 返回此映射的哈希码值。
boolean	**isEmpty**() 如果此映射未包含键-值映射关系，则返回 true。
Set<K>	**keySet**() 返回此映射中包含的键的 Set 视图。
V	**put**(K key, V value) 将指定的值与此映射中的指定键关联（可选操作）。
void	**putAll**(Map<? extends K,? extends V> m) 从指定映射中将所有映射关系复制到此映射中（可选操作）。
V	**remove**(Object key) 如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。
int	**size**() 返回此映射中的键-值映射关系数。
Collection<V>	**values**() 返回此映射中包含的值的 Collection 视图。

1.2、常用集合特性概述

1.2.1、List 系

方法摘要	
boolean	**add**(E e) 向列表的尾部添加指定的元素（可选操作）。
void	**add**(int index, E element) 在列表的指定位置插入指定元素（可选操作）。
boolean	**addAll**(Collection<? extends E> c) 添加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序（可选操作）。
boolean	**addAll**(int index, Collection<? extends E> c) 将指定 collection 中的所有元素都插入到列表中的指定位置（可选操作）。
void	**clear**() 从列表中移除所有元素（可选操作）。
boolean	**contains**(Object o) 如果列表包含指定的元素，则返回 true。
boolean	**containsAll**(Collection<?> c) 如果列表包含指定 collection 的所有元素，则返回 true。
boolean	**equals**(Object o) 比较指定的对象与列表是否相等。
E	**get**(int index) 返回列表中指定位置的元素。
int	**hashCode**() 返回列表的哈希码值。
int	**indexOf**(Object o) 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1。
boolean	**isEmpty**() 如果列表不包含元素，则返回 true。
Iterator<E>	**iterator**() 返回按适当顺序在列表的元素上进行迭代的迭代器。
int	**lastIndexOf**(Object o) 返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回 -1。
ListIterator<E>	**listIterator**() 返回此列表元素的列表迭代器（按适当顺序）。
ListIterator<E>	**listIterator**(int index) 返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始。
E	**remove**(int index) 移除列表中指定位置的元素（可选操作）。
boolean	**remove**(Object o) 从此列表中移除第一次出现的指定元素（如果存在）（可选操作）。
boolean	**removeAll**(Collection<?> c) 从列表中移除指定 collection 中包含的其所有元素（可选操作）。
boolean	**retainAll**(Collection<?> c) 仅在列表中保留指定 collection 中所包含的元素（可选操作）。
E	**set**(int index, E element) 用指定元素替换列表中指定位置的元素（可选操作）。
int	**size**() 返回列表中的元素数。
List<E>	**subList**(int fromIndex, int toIndex) 返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的部分视图。
Object[]	**toArray**() 返回按适当顺序包含列表中的所有元素的数组（从第一个元素到最后一个元素）。
<T> T[]	**toArray**(T[] a) 返回按适当顺序（从第一个元素到最后一个元素）包含列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。

List 特点：**元素有放入顺序，元素可重复**

List 接口有三个实现类：LinkedList，ArrayList，Vector

LinkedList：底层基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。**链表增删快，查找慢**

ArrayList 和 Vector 底层都是基于数组实现的，查询快，增删慢，区别是 ArrayList 是非线程安全的，效率高；Vector 是基于线程安全的，效率低

ArrayList 的初始化大小是 10，扩容策略是 1.5 倍原元素数量的大小

```java
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

```java
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

选择标准：

如果涉及到“动态数组”、“栈”、“队列”、“链表”等结构，应该考虑用 List，具体的选择哪个 List，根据下面的标准来取舍。

1、对于需要快速插入，删除元素，应该使用 LinkedList。（增删改）

2、对于需要快速随机访问元素，应该使用 ArrayList。（查询）

3、对于“单线程环境”或者“多线程环境，但 List 仅仅只会被单个线程操作”，此时应该使用非同步的类(如 ArrayList)。对于“多线程环境，且 List 可能同时被多个线程操作”，此时，应该使用同步的类(如 Vector)。

1.2.2、Set 系

方法摘要	
boolean	**add**(E e) 如果 set 中尚未存在指定的元素，则添加此元素（可选操作）。
boolean	**addAll**(Collection<? extends E> c) 如果 set 中没有指定 collection 中的所有元素，则将其添加到此 set 中（可选操作）。
void	**clear**() 移除此 set 中的所有元素（可选操作）。
boolean	**contains**(Object o) 如果 set 包含指定的元素，则返回 true。
boolean	**containsAll**(Collection<?> c) 如果此 set 包含指定 collection 的所有元素，则返回 true。
boolean	**equals**(Object o) 比较指定对象与此 set 的相等性。
int	**hashCode**() 返回 set 的哈希码值。
boolean	**isEmpty**() 如果 set 不包含元素，则返回 true。
Iterator<E>	**iterator**() 返回在此 set 中的元素上进行迭代的迭代器。
boolean	**remove**(Object o) 如果 set 中存在指定的元素，则将其移除（可选操作）。
boolean	**removeAll**(Collection<?> c) 移除 set 中那些包含在指定 collection 中的元素（可选操作）。
boolean	**retainAll**(Collection<?> c) 仅保留 set 中那些包含在指定 collection 中的元素（可选操作）。
int	**size**() 返回 set 中的元素数（其容量）。
Object[]	**toArray**() 返回一个包含 set 中所有元素的数组。
<T> T[]	**toArray**(T[] a) 返回一个包含此 set 中所有元素的数组；返回数组的运行时类型是指定数组的类型。

Set 特点：**元素放入无顺序，元素不可重复**

Set 接口的实现类：HashSet，TreeSet，LinkedHashSet

HashSet（底层由 HashMap 实现）底层通过 hashCode()和 equals()进行去重。

HashSet 中判断集合元素相等，两个对象比较具体分为如下四个情况：
1、如果有两个元素通过 equal()方法比较返回 false，但它们的 hashCode()方法返回不相等，HashSet 将会把它们存储在不同的位置。
2、如果有两个元素通过 equal()方法比较返回 true，但它们的 hashCode()方法返回不相等，HashSet 将会把它们存储在不同的位置。
3、如果两个对象通过 equals()方法比较不相等，hashCode()方法比较相等，HashSet 将会把它们存储在相同的位置，在这个位置以链表式结构来保存多个对象。这是因为当向 HashSet 集合中存入一个元素时，HashSet 会调用对象的 hashCode()方法来得到对象的 hashCode 值，然后根据该 hashCode 值来决定该对象存储在 HashSet 中存储位置。
4、如果有两个元素通过 equal()方法比较返回 true，但它们的 hashCode()方法返回 true，HashSet 将不予添加。
HashSet 判断两个元素相等的标准：
两个对象通过 equals()方法比较相等，并且两个对象的 hashCode()方法返回值也相等

LinkedHashSet，是 HashSet 的子类，在插入元素的时候，同时使用链表维持插入元素的顺序

SortedSet 接口有一个实现类：TreeSet（底层由平衡二叉树实现）确保集合中的元素都是出于排序状态

注意 LinkedHashSet 和 SortedSet 区别，前者存插入顺序，后者存插入之后的顺序

1.2.3、Map 系

Map 特点：存储的元素是键值对，在 JDK1.8 版本中是 Node，在老版本中是 Entry
Map 接口有五个常用实现类：HashMap，HashTable，LinkeHashMap，TreeMap，ConcurrentHashMap

HashMap 非线程安全，高效，支持 null 的 key 和 value，底层实现是数组和链表，通过 hashCode 方法和 equals 方法保证键的唯一性
如果需要使用线程安全的 Map 可以有两种方式：
采用 HashTable
采用 Collections.synchronizedMap(hashMap)方式进行同步
解决冲突主要有三种方法：定址法，拉链法，再散列法。HashMap 是采用拉链法解决哈希冲突的，拉链法是将相同 hash 值的对象组成一个链表放在 hash 值对应的槽位
HashMap 的初始化大小是 16，扩展因子是 0.75，扩容策略是 2 倍原容量大小

put 的大致流程如下：
1、通过 hashcode 方法计算出 key 的 hash 值
2、通过 hash%length 计算出存储在 table 中的 index（源码中是使用 hash&(length-1)，这样结果相同，但是更快）
3、如果此时 table[index]的值为空，那么就直接存储，如果不为空那么就链接到这个数所在的链表的头部。（在 JDK1.8 中，如果链表长度大于 8 就转化成红黑树）

get 的大致流程如下：
1、通过 hashcode 计算出 key 的 hash 值
2、通过 hash%length 计算出存储在 table 中的 index（源码中是使用 hash&(length-1)，这样结果相同，但是更快）
3、遍历 table[index]所在的链表，只有当 key 与该节点中的 key 的值相同时才取出。

ConcurrentHashMap：是从 JDK1.5 之后提供的一个 HashTable 的替代实现，采用分段锁机制，一个 map 中的元素分成很多的 segment，通过 lock 机制可以对每个 segment 加读写锁，从而提高 map 的效率，底层实现采用数组+链表+红黑树的存储结构

HashTable 线程安全，低效，不支持 null 的 key 和 value

SortedMap 有一个实现类：TreeMap 会存储放入元素的顺序

1.2.4、三者的不同应用场景

List 是用来处理序列的，有序可重复
而 Set 是用来处理集的，无序不重复
Map 处理的是键值对

1.3、详细概述

1.3.1、List 的功能方法

实际上有两种 List: 一种是基本的 ArrayList，其优点在于随机访问元素，另一种是更强大的 LinkedList，它并不是为快速随机访问设计的，而是具有一套更通用的方法。

List: 次序是 List 最重要的特点，它保证维护元素特定的顺序。List 为 Collection 添加了许多方法，使得能够向 List 中间插入与移除元素(这只推荐 LinkedList 使用)一个 List 可以生成 ListIterator，使用它可以从两个方向遍历 List，也可以从 List 中间插入和移除元素。

ArrayList: 由数组实现的 List。允许对元素进行快速随机访问，但是向 List 中间插入与移除元素的速度很慢。ListIterator 只应该用来由后向前遍历 ArrayList，而不是用来插入和移除元素。因为那比 LinkedList 开销要大很多。

LinkedList：对顺序访问进行了优化，向 List 中间插入与删除的开销并不大。随机访问则相对较慢。(使用 ArrayList 代替。)还具有下列方 法：addFirst(), addLast(), getFirst(), getLast(), removeFirst() 和 removeLast(), 这些方法 (没有在任何接口或基类中定义过)使得 LinkedList 可以当作堆栈、队列和双向队列使用。

1.3.2、Set 的功能方法

Set 具有与 Collection 完全一样的接口，因此没有任何额外的功能，不像前面有两个不同的 List。实际上 Set 就是 Collection，只是行为不同。(这是继承与多态思想的典型应用：表现不同的行为)。Set 不保存重复的元素(至于如何判断元素相同则较为负责)

Set：存入 Set 的每个元素都必须是唯一的，因为 Set 不保存重复元素。加入 Set 的元素必须定义 equals()方法以确保对象的唯一性。Set 与 Collection 有完全一样的接口。Set 接口不保证维护元素的次序。

HashSet：为快速查找设计的 Set。存入 HashSet 的对象必须定义 hashCode()。

TreeSet：保存次序的 Set，底层为树结构。使用它可以从 Set 中提取有序的序列。

LinkedHashSet ：具有 HashSet 的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是在使用迭代器遍历 Set 时，结果会按元素插入的次序显示。

1.3.3、Map 的功能方法

方法 put(Object key, Object value)添加一个“值”(想要得东西)和与“值”相关联的“键”(key)(使用它来查找)。方法 get(Object key)返回与给定“键”相关联的“值”。可以用 containsKey()和 containsValue()测试 Map 中是否包含某个“键”或“值”。标准的 Java 类库中包含了几种

不同的 Map：HashMap, TreeMap, LinkedHashMap, WeakHashMap, IdentityHashMap。它们都有同样的基本接口 Map，但是行为、效率、排序策略、保存对象的生命周期和判定“键”等价的策略等各不相同。

执行效率是 Map 的一个大问题。看看 get()要做哪些事，就会明白为什么在 ArrayList 中搜索“键”是相当慢的。而这正是 HashMap 提高 速度的地方。HashMap 使用了特殊的值，称为“散列码”(hash code)，来取代对键的缓慢搜索。“散列码”是“相对唯一”用以代表对象的 int 值，它是通过将该对象的某些信息进行转换而生成的。所有 Java 对象都 能产生散列码，因为 hashCode()是定义在基类 Object 中的方法。

HashMap 就是使用对象的 hashCode()进行快速查询的。此方法能够显著提高性能。

Map：维护“键值对”的关联性，使你可以通过“键”查找“值”

HashMap：Map 基于散列表的实现。插入和查询“键值对”的开销是固定的。可以通过构造器设置容量 capacity 和负载因子 load factor，以调整容器的性能。

LinkedHashMap：类似于 HashMap，但是迭代遍历它时，取得“键值对”的顺序是其插入次序，或者是最近最少使用(LRU)的次序。只比 HashMap 慢一点。而在迭代访问时发而更快，因为它使用链表维护内部次序。

TreeMap：基于红黑树数据结构的实现。查看“键”或“键值对”时，它们会被排序(次序由 Comparabel 或 Comparator 决定)。TreeMap 的特点在 于，你得到的结果是经过排序的。TreeMap 是唯一的带有 subMap()方法的 Map，它可以返回一个子树。

WeakHashMap：弱键(weak key)Map，Map 中使用的对象也被允许释放: 这是为解决特殊问题设计的。如果没有 map 之外的引用指向某个“键”，则此“键”可以被垃圾收集器回收。

IdentifyHashMap：使用==代替 equals()对“键”作比较的 hash map。专为解决特殊问题而设计。

2、反射

2.1、反射概述

首先简单说说什么反射，其实就是动态的加载类，我们在写 JDBC 的时候加载驱动 Class.forName("xxxx")，这句话就涉及到了反射。

反射是自 JAVA 诞生就具备的高级特性，其强大的扩展能力使 JAVA 严谨死板的语法变得灵活。但是能够超越一些 JAVA 对普通类的限定，有关反射主要相关的包在 java.lang.reflect 中。

但是反射也有缺点，就是结构代码较为复杂，使人难以理解，在编程中应当注视上普通的实现方式。由于反射将一个类的各种成分都映射成了相应的类和对象，所以相对普通方法来说，比较消耗资源。

为什么要学习反射呢？因为其强大的扩展性，在 java 开发中框架(比如 Spring)大量应用了反射，反射是 Java 开发者非常有必要掌握的一门技能。

Java 的反射机制主要提供了：

1、在运行时判断任意一个对象的所属的类 Class。
2、在运行时判断构造任意一个类的对象 Constructor。
3、在运行时判断任意一个类所具有的成员变量 Field 和方法 Method。
4、在运行时调用任意一个对象的方法。method.invoke(object, args)

2.2、Class

Class 类（反射的基石），Java 中每个类都代表了一类事物，如 Person 类可以有张三李四的具体对象，Class 是用于描述 JAVA 类的一个类，代表类的字节码实例对象，一个类被加载器加载到内存中，占用一片储存空间，在这个空间里的内容就是类的字节码，这样一个个空间可以用一类对象来表示，这些对象具备相同的类型，这个类型就是 Class，不同类的字节码是不同的，所以他们在内存中的内容也是不同的。

Java 中的类用于描述一类事物的共性，该类的属性是由该类的实例对象来决定的，不同的实例对象具备不同的属性。Java 程序中的各个不同类型的 Java 类也属于同一类事物，而 Class 类就是用于描述这类事物的，Class 类描述了类的属性信息，如类名、访问权限、包名、字段名称列表、方法名称列表等

所有 javaer 都知道，所有类都继承自顶级父类 Object，在 Object 类中有 hashCode()，equals()，clone()，toString()，getClass()等。其中 getClass()方法是返回一个实例对象的 Class 对象声明。这里我们需要使用这个 Class 对象，那么可以通过以下三种方式之一去获取：
1、Class.forName("类名字符串") (注意：类名字符串必须是全称，包名+类名)
2、类名.class
3、实例对象.getClass()

2.3、Constructor

用于代表某个 Class 类的构造方法

2.3.1、获取构造方法

获取某个类的所有构造方法：
```
Constructor[] constructor = Class.forName("java.lang.String").getConstructors();
```

获取某个特殊（特定参数）的构造方法：
```
Constructor constructor = Class.forName("java.lang.String").getConstructor(StringBuffer.class);
```

2.3.2、创建实例对象

通常方式，直接调用构造方法：
```
String str = new String("huangbo");
```

反射方式：调用实参构造
String str = (String)constructor.newInstance(new StringBuffer("huangbo"));

反射方式：调用空参构造
String obj = (String)Class.forName("java.lanng.String").newInstance();

只有两个类拥有 newInstance()方法，分别是 Class 类和 Constructor 类 Class 类中的 newInstance()方法是不带参数的，而 Constructor 类中的 newInstance()方法是带参数的(Object)，需要提供必要的参数

2.4、Field

Field 类代表某个类中的一个成员变量，设有一个 obj 对象，Field 对象不是 obj 具体的变量值，而是指代的是 obj 所属类的哪一个变量，可以通过 Field(对象).get(obj)获取相应的变量值
Field field = obj.getClass().getField("变量名")
field.get(obj) //通过反射获取对象的变量值，参数是对象，意思就是获得那个对象的那个变量的值

get 方法只能获取声明为 public 的变量，对于私有变量，可以通过 getDeclaredField()方法获取 private 变量

获取对象后要通过 setAccessible(true)方法将该域设置为可访问
Field field = obj.getClass().getDeclaredField();
field.setAccessible(true); //将 private 变量设置为可访问；继承自父类 AccessibleObject 的方法
field.get(obj); //才可获取变量值
field.set(obj,newValue) //反射替换，把 obj 对象的 field 属性的值替换为 newValue

2.5、Method

Method 类代表某个类中的成员方法
Method 对象不是具体的方法，而是来代表类中哪一个方法，与对象无关

得到类中某一个方法：
Method methodCharAt = Class.forName("java.lang.String").getMethod("charAt",int.class)
getMethod 方法用于得到一个方法对象，该方法接受的参数首先要有该方法名（String 类型），然后通过参数列表来区分重载那个方法，参数类型用 Class 对象来表示(如为 int 就用 int.class)

调用方法：
普通方式：str.charAt(1)
反射方式：methodCharAt.invoke(str,1)
以上两种调用方式等价

2.6、测试代码

见代码 Reflect_Test.java

2.7、反射作业

通过反射编写一个 JDBCUtil.insert(Object obj)方法。实现可以向任意表中插入一条完整的记录。比如 obj 是 Student 类的一个实例对象，则向对应的 Stduent 数据表中插入一条记录，如果 obj 是 Person 类的一个实例对象，反正不管 obj 是哪个类的一个实例对象，都能够向对应的数据表中的插入一条记录。记录的字段值和对象的属性值要对应。如果做不到对应，请提供一个可添加对应关系的配置文件。

3、设计模式

3.1、设计模式--概念

设计模式（Design pattern）代表了面向对象编程中最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

设计模式只不过针对某些具体场景提供了一些效率较高的以复杂度换灵活性的手段而已

推荐学习站点：http://www.runoob.com/design-pattern/design-pattern-tutorial.html

3.2、设计模式--六大原则

总原则：开闭原则（Open Close Principle）
开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类等，后面的具体设计中我们会提到这点。

六大原则：

1、单一职责原则
不要存在多于一个导致类变更的原因，也就是说**每个类应该实现单一的职责**，如若不然，就应该把类拆分。

2、里氏替换原则（Liskov Substitution Principle）
里氏替换原则中，**子类对父类的方法尽量不要重写和重载**。因为父类代表了定义好的结构，

通过这个规范的接口与外界交互，子类不应该随便破坏它。

3、依赖倒转原则（Dependence Inversion Principle）
这个是开闭原则的基础，具体内容：**面向接口编程，依赖于抽象而不依赖于具体**。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

4、接口隔离原则（Interface Segregation Principle）
这个原则的意思是：**每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分**。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

5、迪米特法则（最少知道原则）（Demeter Principle）
就是说：**一个类对自己依赖的类知道的越少越好**。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过 public 方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

6、合成复用原则（Composite Reuse Principle）
原则是尽量**首先使用合成/聚合的方式，而不是使用继承**。

3.3、设计模式--分类

总体来说设计模式分为三大类：

创建型模式，共五种：**工厂方法模式**、抽象工厂模式、**单例模式**、建造者模式、原型模式。

结构型模式，共七种：**适配器模式**、装饰器模式、**代理模式**、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：**策略模式**、模板方法模式、**观察者模式**、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

3.4、常见设计模式

3.4.1、单例模式

单例模式（Singleton Pattern）是 Java 中最简单的,也是最最最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：
1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。
3、单例类必须给所有其他对象提供这一实例。

共有六种实现：
1、懒汉式，线程不安全
2、懒汉式，线程安全
3、饿汉式
4、双检锁/双重校验锁（DCL，即 double-checked locking）
5、登记式/静态内部类
6、枚举

详细：http://www.runoob.com/design-pattern/singleton-pattern.html

3.4.2、装饰器模式

首先看一段代码，大家一定不会对这段代码陌生：

```
/**
 * 作者:  马中华：http://blog.csdn.net/zhongqi2513
 * 日期:  2017年5月9日 上午12:18:56
 *
 * 描述：实现一个简易的逐行读取文件并打印输出的功能
 */
public class TestBufferedReader {
    public static void main(String[] args) throws IOException {

        BufferedReader br = new BufferedReader(new FileReader(new
File("c:/file1.txt")));

        String line = null;
        while( (line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

分析：
1、构造一个缓冲的字符输入流。包装了一个文件字符输入流。
2、事实上，BufferedReader 就是用来增强 FileReader 的读取的功能的。
3、FileReader 只有 read()方法， 但是 BufferedReader 中却增加了一个 readLine()的逐行读取的功能
4、所以这就相当于是 BufferedReader 装饰了 FileReader，让 FileReader 变得更强大

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

那么新需求来了：ArrayList 这个类中是否有函数式编程语言中的 map(Function)这样的方法？答案是没有。那么如果想有，怎么解决呢？

编程实现：ArrayList.map(Function)
解释：
1、 假设现在有一个 ArrayList ： (1,2,3,4,5,6,7,8,9)
2、 Function 是一个函数，它的功能是接受一个数值，然后+1
3、 返回新的 ArrayList

使用装饰模式，轻松解决。

测试代码如下：

```
import java.util.ArrayList;
import java.util.List;

/**
 * 作者:  马中华：http://blog.csdn.net/zhongqi2513
 * 日期:  2017年5月9日 上午12:38:48
 *
 * 描述: 给List实现一个函数式编程语言中的一个 map(Function) 的功能
 */
public class MyList {

    private List<Integer> list;

    public MyList(List<Integer> list) {
        this.list = list;
    }

    public void map() {
        for(int i=0; i<list.size(); i++) {
            list.set(i, list.get(i) + 1);
        }
    }

    public static void main(String[] args) {

        List<Integer> intList = new ArrayList<>();
        intList.add(1);
        intList.add(2);
        intList.add(3);
```

```
        intList.add(4);

        MyList myList = new MyList(intList);
        myList.map();

        for(Integer i : intList) {
            System.out.println(i);
        }
    }
}
```

拓展：那如果不是+1，而是*2 呢？请结合策略模式!!!

3.4.3、代理模式

什么是代理，在 Design patterns In java 这个本书中是这样描述的，简单的说就是为某个对象提供一个代理，以控制对这个对象的访问。在不修改源代码的基础上做方法增强，代理是一种设计模式，又简单的分为两种：

静态代理：代理类和委托类在代码运行前关系就确定了，也就是说在代理类的代码一开始就已经存在了。

动态代理：动态代理类的字节码在程序运行时的时候生成

第一种：静态代理

具体实现请参照课堂代码 **StudentStaticProxyDAO.java**

静态代理的缺点很明显：一个代理类只能对一个业务接口的实现类进行包装，如果有多个业务接口的话就要定义很多实现类和代理类才行。而且，如果代理类对业务方法的预处理、调用后操作都是一样的（比如：调用前输出提示、调用后自动关闭连接），则多个代理类就会有很多重复代码。这时我们可以定义这样一个代理类，它能代理所有实现类的方法调用：根据传进来的业务实现类和方法名进行具体调用。那就是动态代理。

第二种：动态代理

常用的动态代理的实现有两种：

第一种：JDK 动态代理实现

JDK 动态代理所用到的代理类在程序调用到代理类对象时才由 JVM 真正创建，JVM 根据传进来的业务实现类对象以及方法名，动态地创建了一个代理类的 class 文件并被字节码引擎执行，然后通过该代理类对象进行方法调用。我们需要做的，只需指定代理类的预处理、调用后操作即可。

只能对实现了接口的类生成代理，而不是针对类，该目标类型实现的接口都将被代理。原理是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。具体实现步骤：

1、定义一个实现接口 InvocationHandler 的类
2、通过构造函数或者静态工厂方法等，注入被代理类
3、实现 invoke(Object proxy, Method method, Object[] args)方法
4、在主函数中获得被代理类的类加载器
5、使用 Proxy.newProxyInstance(classLoader, interfaces, args)产生一个代理对象
6、通过代理对象调用各种方法

具体实现参照课堂代码 **LoggerDynamicProxy.java**

第二种：CGLIB 动态代理实现：
CGLIB 是针对类来实现代理的，原理是对指定的业务类生成一个子类，并覆盖其中业务方法实现代理。因为采用的是继承，所以不能对 final 修饰的类进行代理，final 的方法也不能

针对类实现代理，对是否实现接口无要求。原理是对指定的类生成一个子类，覆盖其中的方法，因为是继承，所以被代理的类或方法最好不要声明为 final 类型。具体实现步骤：
1、定义一个实现了 MethodInterceptor 接口的类
2、实现其 intercept()方法，在其中调用 proxy.invokeSuper()

第三：静态代理和动态代理的区别

这两种动态代理的**最主要区别**在于 JDK 的动态代理的委托类，必须要实现一个接口，而 CGLIB 不用。

静态代理：自己编写创建代理类，然后再进行编译，在程序运行前，代理类的.class 文件就已经存在了。
动态代理：在实现阶段不用关心代理谁，而在运行阶段（通过反射机制）才指定代理哪一个对象。

4、排序算法

所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。排序算法，就是如何使得记录按照要求排列的方法。排序算法在很多领域得到相当地重视，尤其是在大量数据的处理方面。一个优秀的算法可以节省大量的资源。在各个领域中考虑到数据的各种限制和规范，要得到一个符合实际的优秀算法，得经过大量的推理和分析。 ----From 百度百科

核心概念：**算法复杂度**、**稳定性**

算法复杂度：算法复杂度是指算法在编写成可执行程序后，运行时所需要的资源，资源包括时间资源和内存资源。应用于数学和计算机导论。

稳定性：一个排序算法是稳定的，就是当有两个相等记录的关键字 R 和 S，且在原本的列表

中 R 出现在 S 之前，在排序过的列表中 R 也将会是在 S 之前。

4.1、排序算法分类

各种排序算法：

冒泡排序
选择排序
插入排序
归并排序
堆排序
快速排序
计数排序
基数排序
桶排序
希尔排序
……

排序分类：

按照排序结果是否稳定性分类：
1、稳定排序：插入排序，冒泡排序，归并排序，计数排序，基数排序，桶排序（如果桶内排序采用的是稳定性排序）
2、非稳定排序：选择排序，快速排序，堆排序。

按照排序过程中是否需要额外空间：
1）原地排序：插入排序，选择排序，冒泡排序，快速排序，堆排序。
2）非原地排序：归并排序，计数排序，基数排序，桶排序。

按照排序的主要操作分类：
1）交换类：冒泡排序、快速排序；此类的特点是通过不断的比较和交换进行排序；
2）插入类：简单插入排序、希尔排序；此类的特点是通过插入的手段进行排序；
3）选择类：简单选择排序、堆排序；此类的特点是看准了再移动；
4）归并类：归并排序；此类的特点是先分割后合并；

按照是否需要比较分类：
1、比较排序，时间复杂度 O(nlogn) ~ O(n^2)，主要有：冒泡排序，选择排序，插入排序，归并排序，堆排序，快速排序等。
2、非比较排序，时间复杂度可以达到 O(n)，主要有：计数排序，基数排序，桶排序等。

4.1.3、常用排序算法比较

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						

有趣的排序算法视频：http://v.youku.com/v_show/id_XNjkzODY2NjMy.html

4.2、常见排序算法的核心实现

4.2.1、冒泡排序

见代码 BubbleSort.java

4.2.2、归并排序

见代码 MergeSort.java

4.2.3、快速排序

见代码 QuickSort.java