创建和销毁对象

2018年3月3日　　16:23

一、考虑用静态工厂方法代替构造器

1.静态工厂方法是一个静态方法，用来返回类的实例。

```
public static Boolean valueOf（boolean b）
{
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

2.优点：

（1）静态工厂方法可以有自己的名称，其含义比构造器更清晰

```
PrimeNumber primeNumber1 = new PrimeNumber(int Random); // 返回一个素数
PrimeNumber primeNumber2 = PrimeNumber.newInstance(); // 使用静态工厂方法表示更为清楚
```

（2）可以返回原返回类型的的任何子类型对象

```
class Father {
    private Father() {
    }

    public static Father newInstance(String type) {
        if (type.equals("ChildA")) { // 根据类型判断返回那个子类对象
            return new ChildA();
        } else {
            return new ChildB();
        }
    }

    public void getName() {
        System.out.println("My name is father");
    }

    private static class ChildA extends Father {
        public void getName() {
            System.out.println("My name is child A");
        }
    }

    private static class ChildB extends Father {
        public void getName() {
            System.out.println("My name is child B");
        }
    }
}

public class Test {
    public static void main(String[] args) {
```

```
            Father c1 = Father.newInstance("ChildA");
            c1.getName();
            Father c2 = Father.newInstance("ChildB");
            c2.getName();
        }
    }
```

（3）在使用泛型时，可使代码更简洁

```
    private Map<String, List<String>> map = new HashMap<String, List<String>>();
    public static <K, V> HashMap<K, V> newInstance() {
        return new HashMap<K, V>();
    }
```

3.缺点
（1）如果类中没有公有的或者受保护的构造器，就不能被实例化
（2）与其他的静态方法很难区分，所以规定静态工厂方法使用valueOf、getInstance等名称

二、遇到多个构造器参数时要考虑用构建器
1.如果类的构造器或静态工厂中具有多个参数，而且**大多数参数都是可选的**，这时使用传统的构造器或静态工厂就会不方便，因为不能很好地扩展参数个数。传统的的替代方法有三种：
（1）重叠构造器：第一个构造器的参数是必需的，第二个构造器参数比第一个多一个，第三个又比第二个多一个，以此类推。客户端只使用第一个构造器，第一个构造器在内部调用第二个（使用this），第二个调用第三个，以此类推。
（2）JavaBeans：构造器是无参的，使用setter来给成员属性赋值。
（3）**构建器**（Builder模式）：在类的内部有一个静态类，专门用来初始化成员属性。
2.构建器是比较好的一种方法。

```
    public class A {

        private int a;

        private int b;

        private int c;



        public static class Builder {

            private int a;

            private int b;

            private int c;

            public Builder(){}

            public Builder seta(int a) { this.a = a; return this}

            public Builder setb(int b) { this.b = b; return this}

            public Builder setc(int c) { this.c = c; return this}
```

```
        public A build() { return new A(this)}

    }


    private A(Builder builder) {

        this.a = builder.a;

        this.b = builder.b;

        this.c = builder.c;

    }

}

//调用构造方法:
A a = new A.Builder().seta(1).setb(2).setc(3).build();
```

首先客户端利用构造器或者静态工厂获得一个构建器，构建器再调用类似setter的方法进行赋值，最后调用build方法返回**不可变**对象。构建器的实现代码中有两个注意点：（1）setter中使用this
是为了方便链式调用。（2）利用了内部类和外部类可以方便地互相访问成员的特性。
2.构建器适用于有多个参数（多于4个），特别是当大多数参数都是可选的时候。
3.构建器的缺点是代码冗长，运行效率不高。但是如果符合构建器的使用条件，还是应该一开始就使用构建器。

三、用私有构造器或者枚举强化单例属性
实现单例的方法主要有懒汉模式和饿汉模式。Java 5之后使用单元素的枚举类型已经成为实现单例的最佳方法。

```
public enum Singletion {

    INSTANCE;

    public static Singletion getInstance(){
        return INSTANCE;
    }
}
```

四、通过私有构造器强化不可实例化的能力
1.不应该通过把类定义成抽象类来使其不可实例化，因为其子类仍可以实例化，而且抽象类通常会被理解为专门为继承而设计的。如果要保证一个类不可实例化，只需将其构造器私有化。
2.这种方法的一个弊端是，子类无法通过super调用父类的构造器。

五、避免创建不必要的对象
1.作为一个不应该这样做的极端例子，请考虑以下语句：

```
String s = new String("bikini");  // DON'T DO THIS!
```

正确的做法如下：

```
String s = "bikini";
```

2.对于那些不可变的（上面的字符串）或是已知不会被修改的对象，应该重用而不是总去创建新的对象。

```
public class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }

    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static{
        Calendar gmtCal=Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946,Calendar.JANUARY,1,0,0,0);
        BOOM_START=gmtCal.getTime();
        gmtCal.set(1965,Calendar.JANUARY,1,0,0,0);
        BOOM_END=gmtCal.getTime();

    }

    public boolean isBabyBoomer(){

        return birthDate.compareTo(BOOM_START)>=0
&&birthDate.compareTo(BOOM_END)<0;
    }
}
```

通过把创建对象的代码放在静态代码块中保证代码只执行一次即只创建一个新对象。
3.优先使用基本类型而不是包装类。

六、消除过期的对象引用
1.Java虽然有垃圾回收机制，但是不会消除一些引用的对象，从而造成内存泄漏，称为“无意识的对象保持”。
2.栈这种数据结构会维护对象的过期引用（因为对于垃圾回收器来说栈里面的对象都是等同的），就是说即使某个对象出栈了也不会被清空。这种情况下就需要对出栈的对象进行手动清空。

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack(){
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e){
```

```
        ensureCapacity();
        elements[size ++] = e;
    }

    public Object pop(){
        if (size == 0){
            throw new EmptyStackException();
        }

       Object result = elements[--size];
       //清空引用
       elements[size] = null;
       return result;
    }

    private void ensureCapacity(){
        if (elements.length == size){
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }

}
```

2.内存泄漏的常见来源还有**缓存、监听器和回调**。内存泄漏一般不会造成程序的明显失败，但如果能发现并阻止其发生，那就最好不过了。

七、避免使用终结方法

1.尽量避免在类中使用终结（finalize）方法，在里面写一些释放类中资源的语句。

2.终结方法的缺点：

（1）java语言规范不仅不保证 finalize方法会被及时地执行，而且根本不保证他们会被执行。

（2）System.gc 和 System.runFinalization 这两个方法只是增加了finalizer 方法被执行的机会。

（3）唯一能保证 finalize 方法被执行的方法有两个，System.runFinalizersOnExit 和 Runtime.runFinalizersOnExit ，但是这两个方法已经被废弃。

（4）覆盖并使用终结方法会有严重的性能损失。

（5）及时地执行终结方法是垃圾回收算法的一个主要功能，但是在不同的JVM实现中会大相庭径，使用终结方法可能会丧失平台无关性。

3.一般来说，需要释放资源的有线程或者文件还有涉及到本地资源的对象。我们不去覆盖 finalize 方法，而是自己提供一个显式的终止资源的方法。比如 java.io.FileInputStream 的close方法。当使用完这个FileInputStream对象时，显式调用close()来回收资源。

4.终结方法的主要用途

（1）当对象持有者忘记调用前面段落中建议的显式终结方法 ：close()的情况下，使用终结方法充当“安全网”。

（2）把终结方法放在一个**匿名类**，该匿名类的唯一用途就是终结它的外围实例。外围实例持有对终结方法守卫者的唯一实例，这意味着当外围实例是不可达（引用为零）时，这个终结方法守卫者也是不可达的了，垃圾回收器回收外围实例的同时也会回收终结方法守卫者的实例，而终结方法守卫者的终结方法就把外围实例的资源释放掉，就好像是终结方法是外围实例的一个方法一样。

对象通用方法

2018年3月4日　　10:27

一、覆盖Object.equals

1.不需要覆盖equals的情况：

（1）类的每个实例本质一致。

（2）不关心类是否提供了**逻辑相等**的功能。

（3）父类已经覆盖了equals。

（4）类是私有的或是包级私有。

2.覆盖equals时需遵守的约定：

（1）自反性：若x!=null，则x.equals(x)=true。

（2）对称性：若x!=null且y!=null，则y.equals(x)=true当且仅当x.equals(y)=true。

（3）传递性：若x，y，z均不为null，x.equals(y)=true且y.equals(z)=true，则x.equals(z)=true。

（4）一致性：若x，y均不为null，则当多次equals时返回的结果应一致。

（5）若x!=null，则x.equals(null)=false。

3.实现高质量的equals

（1）如果要检查equals的参数是否为该对象的**引用**，则使用==

（2）使用**instanceof**检查参数是否为正确的类型，如果不是，则返回false；如果是，还要转换成当前类类型（原本是Obejct）。

（3）对于类中的每个“关键域”，检查参数中的域是否与该对象中相应的域相匹配。

（4）遵循对称性、传递性、一致性

（5）不要企图让equals过于智能，即不必把任何一种可能的**别名形式**考虑到等价范围内。

（6）不要把equals声明中的Ocject换成其他类型，使用@Override注解防止出错

（7）**覆盖equals的同时覆盖hashCode**。没有覆盖hashCode会造成的后果是，（逻辑）相等的对象不具有相等的散列码（对象在集合中的位置）。覆盖hashCode的方法如下：

①把某个非零的常数值，比如说17，保存到一个名为result的int类型变量中。

②对于对象中每个关键域f（**指equals方法中涉及的每个域**），完下面的步骤：

a. 为该域计算int类型的散列码c:

i. 如果该域是boolean类型的，则计算(f ? 1 : 0)

ii. 如果该域是byte、char、short、或者int类型的，则计算(int)f。

iii. 如果是long类型的，则计算(int)(f^(f>>>32))。

iv. 如果该域是float类型，则计算FLoat.floatToIntBits(f)。

v. 如果该域是double类型，则计算Double.doubleToLongBits(f)，然后按照步骤a.iii，为得到long类型值计算散列值。

vi. 如果该域是一个对象引用，并且该类的equals方法通过递归地调用equals的方式来比较这个域，则同样为这个域递归地调用hashCode。如果需要更复杂的比较，则为这个域计算一个范式(canonical representation)，然后针对这个范式调用hashCode。一种简单的做法是，如果该对象为null，则返回0；否则，返回这个对象调用hashCode得到的值。

vii. 如果该域是一个数组，则要把每个元素当作单独的域来处理。

b. 按照下面的公式，将a计算得到的散列码c合并到result中： result = 31 * result + c;

③返回result。

④写完了之后，检查是否符合“相等的实例具有相等的散列码”。

```
public class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    public PhoneNumber(int areaCode, int prefix, int lineNumber) {
        this.areaCode = (short) areaCode;
        this.prefix = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }
    //覆盖equals方法
    @Override
    public boolean equals(Object obj) {
        if (obj == this)
            return true;
        if (!(obj instanceof PhoneNumber))
            return false;
        //必须满足如下条件，才能说明为同一个对象
        PhoneNumber pn = (PhoneNumber) obj;
        return pn.areaCode == areaCode && pn.prefix == prefix &&
pn.lineNumber == lineNumber;
    }
@Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + areaCode;
        result = 31 * result + prefix;
        result = 31 * result + lineNumber;
        return result;
    }
}
```

如果一个类是不可变类，并且计算散列码的开销也比较大，就应该考虑把散列码缓存在对象内部，而不是每次请求的时候都重新计算散列码。

```
 private volatile static int hashcode;
```

二、始终覆盖toString

1.toSring的通用约定指出，被返回的字符串应该是一个“简洁的，但信息丰富且易于阅读的表达式”。然而，默认的toString方法是：

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

2.toString方法应该返回对象中包含的所有值得关注的信息，并且应该为每一个信息提供getter。

3.在覆盖之前要想好是否在文档中指定返回值的格式，并且在文档中要有相关的注释信息。

三、谨慎地覆盖clone

1.Cloneable接口的leclone方法的主要作用是创建和返回对象的一个拷贝，通常情况下应满足：

```
x.clone()!= x;           //true
x.clone().getClass() == x.getClass();  //true
x.clone().equals(x);     //true
```

2.所有实现了Cloneable接口的类都应该用一个公有方法覆盖clone，这个方法要先调用**super.clone**，然后根据情况修改任何需要修改的域。
3.实现一个行为良好的clone方法通常都比较复杂，所以除非必需，否则应该提供其他的对象拷贝方法或是干脆不提供这样的方法。
4.另一个实现对象拷贝的好办法就是提供一个**拷贝构造器**或者**拷贝工厂方法**。拷贝构造器的唯一参数类型就是包含该构造器的类，拷贝工厂就是类似拷贝构造器的静态工厂。基于接口的拷贝构造器和拷贝工厂有一大优势，就是允许客户选择拷贝的实现类型，例如将HashSet拷贝为TreeSet：

```
new TreeSet(s);
```

三、考虑实现Comparable接口
1.如果一个值类（实例表示一个值的类）具有非常明显的内在排序关系，就应该实现Comparable接口。
2.compareTo方法的通用约定与equals方法的相似：
（1）满足对称性：必须确保所有的x和y都满足sgn(x.compareTo(y)) == -sgn(y.compareTo(x));
（2）满足传递性：若(x.compareTo(y) >0 && y.compareTo(z)>0)，则x.compareTo(z)>0。
（3）若x.compareTo(y)==0,则sgn(x.compareTo(z)) == sgn(y.compareTo(z));（这一条只是强烈建议，并不是硬性规定）
3.如果一个类并没有实现Comparable接口，或是需要一个非标准的排序关系，可以使用一个显式的比较函数来替代，比如说String.CASE_INSENSITIVE_ORDER.compare（忽略大小写的比较）。
4. 比较整数型基本类型的域，可以使用关系操作符<和>。但是，浮点域用**Double.compare**或者**Float.compare**，而不用关系操作符，当应用到浮点值得时候，它们没有遵守compareTo的通用约定。对于数组域，则要把这些知道原则应用到每个元素上。
5. 如果一个类有多个关键域，那么比较这些关键域的顺序非常关键。必须从最关键的域开始，逐步进行到所有的重要域。如果某个域的比较产生了非零的结果(0代表着相等)，则整个比较操作结束，并返回该结果。如果最关键的域是相等的，则再比较下一个关键域，以此类推，如果所有域都是相等的，那么才返回0。例如下面的例子：

```
public final class PhoneNumber implements Comparable {

    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    public PhoneNumber(int areaCode, int prefix,
                       int lineNumber) {
        this.areaCode = (short) areaCode;
        this.prefix = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }

    @Override
    public int compareTo(PhoneNumber pn) {
        if (areaCode < pn.areaCode)
            return -1;
        if（areaCode > pn.areaCode)
```

```
            return 1;

        if (prefix < pn.prefix)
            return -1;
        if (prefix > pn.prefix)
            return 1;

        if (lineNumber < pn.lineNumber)
            return -1;
        if (lineNumber > pn.lineNumber)
            return 1;

        return 0;
    }
}
```

6.compareTo方法的约定并没有指定返回值的大小(magnitude)，而只是指定了返回值的符号。可以利用这一点来简化代码，或许还可以提高它的运行速度。

```
public int compareTo(PhoneNumber pn) {
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0)
        return areaCodeDiff;

    int prefixDiff = prefix - pn.prefix;
    if (0 != prefixDiff)
        return prefixDiff;

    return lineNumber - pn.lineNumber;
}
```

使用这种方法的时候需要注意，有符号的32位整数还不足以大到能够表达任意两个32位整数的差值，如果i是一个很大的正整数，j是一个很小的负整数，i-j有可能会溢出，并且返回一个负值。

类和接口（一）

2018年3月4日　　15:16

一、使类和成员的可访问性最小化

1.尽可能使每个类或者成员不被外界访问。

2.对于顶层的（非嵌套的）类和接口，应该尽可能地使其成为包级私有的，这样就可以使其成为这个包的实现的一部分；如果声明为公有的，则其成为这个包的API的一部分，这就需要程序员永远要记得保持其兼容性。

3.私有成员和包级私有成员都是一个类的实现的一部分，一般不会影响类的API。如果程序员发现一个类经常需要访问同一个包内另一个类的成员，他就应该重新考虑类的划分，以降低耦合度。

4.如果方法覆盖了父类的一个方法，那么子类中的访问级别应不低于父类中的访问级别。特殊情况是，如果一个类实现了一个**接口**，那么接口中所有的类方法在这个类中也都必须被声明为公有的，因为接口中的所有方法都隐含着公有访问级别。

5.实例成员绝不能是公有的，静态成员不应该是公有的。

6.公有类不应包含公有静态final属性之外的其他公有成员，并且要确保公有静态final属性所引用的对象是不可变的。

二、在公有类中使用访问方法而非公有属性

1.一般情况下，永远不应该直接暴露类中的属性（数据成员），而要通过公有的setter去使用它们。不过暴露不可变的公有final属性危害没有这么大。

2.如果类是包级私有的，或者是私有的嵌套类，直接暴露它的数据域并没有本质的错误。因为在这些情况下数据的访问被限制在包内或者是外围类内。

三、使可变性最小化

1.不可变类是其实例不能被修改的类。Java平台类库中包含许多不可变的类，比如String、包装类、BigInteger和BigDecimal。不可变类所遵循的原则有：

（1）**不要提供任何会修改对象状态的方法**，比如公有setter。为了提高性能，在实际开发中可以有所放松：没有一个方法能够对对象产生外部可见的改变。

（2）**保证类不会被继承**。为了防止子类化，一般做法是使这个类成为final的 。还有一种做法是把类的构造器私有化，然后用公有的静态工厂方法去替代。

（3）**使所有的域都是final的**。一是为了表明意图，二是为了在多线程间确保对对象使用正确的行为。

（4）**使所有的域都成为私有的**。这样可以防止客户端获得访问被域引用的可变对象的权限，并防止客户端直接修改这些对象。

（5）**确保对于任何可变组件的互斥访问**。如果类具有指向可变对象的域，则必须确保该类的客户端无法获得指向这些对象的引用。并且，永远不要用客户端提供的对象引用来初始化这样的域，也不要从任何方法中返回该对象引用。

2.不可变类通常采用**函数式编程**，类中的方法返回一个函数的结果，这些函数对操作数进行运算但是并不修改它。

```
public Complex(double re, double im){
    this.re = re;
    this.im = im;
}

public double realPart(){return re;}
public double imaginaryPart(){return im;}

public Complex add(Complex c){
    return new Complex(re + c.re, im + c.im);
}
```

```
public Complex subtract(Complex c){
    return new Complex(re - c.re, im - c.im);
}

public Complex multiply(Complex c){
    return new Complex(re * c.re - im * c.im, re * c.im + im * c.re);
}

public Complex divide(Complex c){
    double tmp = c.re * c.re + c.im * c.im;
    return new Complex((re * c.re + im * c.im) / tmp, (im * c.re - re * c.im)
/ tmp);
}

@Override
public boolean equals(Object o){
    if (o == this)
        return true;
    if (!(o instanceof Complex))
        return false;
    Complex c = (Complex)o;
    return Double.compare(re, c.re) == 0 && Double.compare(im, c.im) == 0;
}

@Override
public int hashCode(){
    int result = 17;
    result = 31 * result + hashDouble(re);
    result = 31 * result + hashDouble(im);
    return result;
}

private int hashDouble (double val) {
    long longBits = Double.doubleToLongBits(re);
    return (int) (longBits ^ (longBits >>> 32));
}
```

3.不可变对象状态稳定，使用起来较为简单。不可变对象本质上是线程安全的，它们不要求同步，所以不可变对象可以被自由地共享；而这一特性也使得不可变对象不需要进行任何拷贝操作。

4.不可变类的唯一缺点是，对于每个不同的值都需要一个独立的对象。当进行一个多步骤从操作，而且每个步骤都会产生一个新对象时，这就会影响性能。

5.对于一些类，完全不可变的实现使不可能的，但是仍要限制其可变性。除非必要，否则每个域都是final的。而且，在构造器和静态工厂方法之外不应再提供公有的初始化方法，除非必须这么做。

四、复合优先于继承

1.对于包内继承以及继承那些专门设计成为父类且具有良好的文档说明的类来说，继承是非常安全的。然而，对于普通的具体类进行跨包继承，则很危险。继承打破了封装性，子类依赖于其父类中的特定功能的实现细节，父类的变化有可能对子类造成破坏，所以子类必须跟着父类演变。这种错误通常出现在可覆盖的方法上。

2.不继承现有类，而是在新的类中增加一个私有的现有类实例，这种方法称为**复合**。新类中的每个实例方法都可以调用被包含的现有类实例中对应的方法，这被称为**转发**，新类中的方法称

为转发方法。

```
//Wrapper class - use composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E>{

    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override
    public boolean add(E e) {
        addCount ++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}

//Reusable forwarding class
class ForwardingSet<E> implements Set<E> {

    private final Set<E> s;

    public ForwardingSet(Set<E> s) {this.s = s;}

    @Override
    public int size() {return s.size();}

    @Override
    public boolean isEmpty() {return s.isEmpty();}

    @Override
    public boolean contains(Object o) {return s.contains(o);}

    @Override
    public Iterator<E> iterator() {return s.iterator();}

    @Override
    public Object[] toArray() {return s.toArray();}

    @Override
    public <T> T[] toArray(T[] a) {return s.toArray(a);}

    @Override
```

```
        public boolean add(E e) {return s.add(e);}

        @Override
        public boolean remove(Object o) {return s.remove(o);}

        @Override
        public boolean containsAll(Collection<?> c) {return s.containsAll(c);}

        @Override
        public boolean addAll(Collection<? extends E> c) {return s.addAll(c);}

        @Override
        public boolean retainAll(Collection<?> c) {return s.retainAll(c);}

        @Override
        public boolean removeAll(Collection<?> c) {return s.retainAll(c);}

        @Override
        public void clear() {s.clear();}

    }
```

3.如果子类和父类并不存在确切的继承关系，而且子类和父类处于不同的包中，则不应该使用继承。

五、要么为继承而设计，并提供文档说明，要么就禁止继承

1.对于每个公有的或受保护的方法或者构造器，它的文档必须指明该方法或者构造器调用了哪些**可覆盖**的方法（非final的，公有的或者受保护的），是以什么顺序调用的，每个调用的结果又是如何影响后续的处理过程的。更一般的，类必须在文档中说明，在哪些情况下它会调用可覆盖的方法。关于程序文档的格言：好的API文档应该说明一个给定的方法做了什么工作，而不是描述它是如何做到的。

2.为了使程序员能够编写出更加有效的子类，而无需随不必要的痛苦，类必须通过某种形式提供适当的钩子，以便能够进入到它的内部工作流程中，这种形式可以是精心选择的受保护的方法，也可以是受保护的域，后者比较少见。

3.为了允许继承，类必须遵守的一些规则：

（1）**构造器绝对不能调用可覆盖方法**，无论是直接还是间接调用。

（2）如果一个为继承而设计的类实现了Cloneable或者Serializable接口，那么无论是**clone**还是**readObject**方法都不可以调用可覆盖方法。

（3）如果一个为继承而设计的类实现了Serializable接口，并且该类有一个readResolve或者writeReplace方法，就必须**使readResolve或者writeReplace方法成为受保护的方法**，而不是私有方法。

4.对于为了继承而设计的类，唯一的测试方法就是编写子类。经验表明，3个子类通常就足以测试一个可扩展的类，除了超类的创建者外，都要编写一个或者多个这种子类。

类和接口（二）

2018年3月4日　　19:55

六、接口优于抽象类

1.现有的类可以很容易地进行改进来实现一个新的接口。 你只需添加所需的方法（如果尚不存在的话），并向类声明中添加一个**implements**子句。如果你想让两个类继承相同的抽象类，你必须把它放在类型层级结构中的上面位置，让它成为两个类的祖先。不幸的是，这会对类型层级结构造成很大的附带损害，迫使新的抽象类的所有后代对它进行子类化，无论这些后代类是否合适。

2.接口是定义**混合类型**（mixin）的理想选择。一般来说，mixin是一种类，除了它的“主类型”之外，还可以声明它提供了一些可选的行为。这样的接口被称为混合类型，因为它允许可选功能被“混合”到类型的主要功能。

3.接口允许构建**非层级**的类型框架。

4.Java 8之前接口是不可以有方法体的，这就是抽象类相对于接口的优势，为了将抽象类和接口的优势整合起来，“**骨架类**”就诞生了，骨架类的做法是用一个抽象类来实现一个接口，在抽象类中为接口的某些方法提供实现。骨架类的实现的一般步骤是，找出接口中的基本方法，在抽象类中声明为抽象方法，然后用这些基本方法来实现其他方法，所谓基本方法，就是通过将这些方法组合或是变换，可以实现其他的方法。

```
public interface Summation<T> {
  //实现两个对象的相加
  T towEleAdd(T obj01, T obj02);

  //实现List求和
  T listEleSum(List<T> list);

  //实现数组求和
  T arrayEleSum(T[] array);
}

public abstract class AbstractSummation<T> implements Summation<T> {

  @Override
  public abstract T towEleAdd(T obj01, T obj02);

  @Override
  public T listEleSum(List<T> list) {
    T firstEle = null;
    for (T t : list) {

      if (firstEle == null) {
        firstEle = t;
        continue;
      }

      firstEle = towEleAdd(firstEle, t);
    }
    return firstEle;
  }

  @Override
  public T arrayEleSum(T[] array) {
```

```
        T firstEle = null;
        for (T t : array) {

          if (firstEle == null) {
            firstEle = t;
            continue;
          }

          firstEle = towEleAdd(firstEle, t);
        }
        return firstEle;
      }
    }
```

5.一般来说，几乎不可能在不破坏现有实现类的情况下往接口中增加方法，因为现有实现类必须实现每一个新增方法。所以，在设计公有接口时必须非常谨慎，这也说明了接口不易演变。

七、接口只用于定义类型

1.有一种接口被称为常量接口（constant interface）。这种接口没有包含任何方法，只包含静态的final域，每个域都都是一个常量。使用这些常量的类实现这个接口，以避免用类名来修饰常量名。常量接口模式是对接口的不良使用。简而言之，接口应该只被用来定义类型，不应该被用来导出常量。

2.如果要导出常亮，可以有其他的实现方法：

（1）如果这些常量与某个现有的类或者接口紧密相关，就应该把这些常量添加到这个类或者接口中。例如，在java平台类库中所有的数值包装类，比如Integer和Double，都导出了MIN_VALUE和MAX_VALUE常量。

（2）如果这些常量最好被看做枚举类型的成员，就应该用枚举类型来导出这些常量。

（3）使用不可实例化的工具类来导出这些常量。

八、类层次优于标签类

带有两个甚至更多风格的实例的类，并包含表示实例风格的标签域。这种类称为**标签类**，是对类层次的一种简单效仿。

```
    class Figure {
        enum Shape { RECTANGLE, CIRCLE };

        // Tag field - the shape of this figure
        final Shape shape;

        // These fields are used only if shape is RECTANGLE
        double length;
        double width;

        // This field is used only if shape is CIRCLE
        double radius;

        // Constructor for circle
        Figure(double radius) {
            shape = Shape.CIRCLE;
            this.radius = radius;
        }

        // Constructor for rectangle
        Figure(double length, double width) {
            shape = Shape.RECTANGLE;
```

```
            this.length = length;
            this.width = width;
        }

        double area() {
            switch(shape) {
              case RECTANGLE:
                return length * width;
              case CIRCLE:
                return Math.PI * (radius * radius);
              default:
                throw new AssertionError();
            }
        }
    }
```

标签类过于冗长，容易出错，并且效率低下。如果要表示不同风格的实例，实现不同风格的子类就好了。

九、用函数对象表示策略

1.如果一个类仅仅导出一个方法，它的实例实际上就等同于一个指向该方法的指针。这样的实例被称为**函数对象**。

```
    public class StringLengthComparator
    {
        public int compare(String s1, String s2)
        {
            return s1.length() - s2.length();
        }
    }
```

使用StringLengthComparator并不好，因为客户端将无法传递任何其他的策略。相反，我们需要定义一个Comparator接口，并修改StringLengthComparator来实现这个接口。换句话说，我们在设计具体策略类时，还需要一个策略接口。

```
    public interface Comparator<T>
    {
        public int compare(T t1, T t2);
    }
```

2.声明一个接口来表示策略，并且为每个具体策略声明一个实现了该接口的类。当一个具体策略被使用一次时，通常使用**匿名类**来声明和实例化这个具体策略类。

```
    Arrays.sort(stringArray,new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return 0;
        }
    });
```

当一个具体策略是设计用来重复使用的时候，它的类通常就要被实现为私有的静态成员类，并通过公有的**静态final域**导出。

```
    public class Host {
        private static class StrLenCmp implements Comparator<String>,
    Serializable {
            @Override
            public int compare(String o1, String o2) {
                return o1.length() - o2.length();
            }
        }
```

```
    public static final Comparator<String> STRING_LENGTH_COMPARATOR = new
StrLenCmp();
}
```

十、优先考虑静态成员类

1.Java嵌套类指的是定义在另一个类内部的类，分为两大类：**静态成员类和内部类**。内部类又分为三种，非静态成员类、匿名类和局部类。

2.**静态成员类**可看做是普通的类，只是碰巧被声明在另一个类的内部而已，它可以访问外围类的所有成员，包括声明为私有的成员。静态成员类的一种常见用法是作为公有的辅助类，仅当与外部类一起使用时才有意义。例如考虑一个枚举，描述了计算机支持的各种操作。Operation枚举应该是Calculator类的公有静态成员类，然后，Calculator类的客户端就可以用诸如Calculator.Operation.PLUS和Calculator.Operation.MINUS这样的名称来引用这些操作。

```
public class Calculator {
    public enum Operation {
        PLUS, MINUS, TIMES, DIVIDE;

        double apply(double x, double y) {
            switch (this) {
                case PLUS:
                    return x + y;
                case MINUS:
                    return x - y;
                case TIMES:
                    return x * y;
                case DIVIDE:
                    return x / y;
            }
            throw new AssertionError("Unknow op:" + this);
        }
    }

}
```

3.非静态成员类的实例被创建时，它和外围类实例之间的关联关系也被建立起来，而且这种关联关系随后便不能被修改。在没有外围类实例的情况下，无法创建非静态成员类实例。

4.匿名类没有名字。它不是外围类的一个成员。它并不与其他的成员一起被声明，而是在使用的同时被声明和实例化。匿名类可以出现在代码中任何允许存在表达式的地方。当且仅当匿名类出现在非静态的环境中时，它才有外围实例。但是即使它们出现在静态的环境中，也不可能拥有任何静态成员。匿名类的使用要受到诸多限制，匿名类的常见用途有三个：

（1）动态创建函数对象。

（2）创建过程对象，比如Runnable、Thread或者TimerTask实例。

```
    public static void runSomeThing(){

        Runnable runnable = new Runnable() {

            @Override
            public void run() {
                System.out.println("I am running");
            }
        };
        new Thread(runnable).start();
    }
```

（3）用在静态工厂内部。

```
static List<Integer> intArrayAsList(final int[] a) {

    if (a == null)

        throw new NullPointerException();

    return new AbstractList<Integer>() {

        public Integer get(int i) {

            return a[i];

        }

        @Override

        public Integer set(int i, Integer val) {

            int oldVal = a[i];

            a[i] = val;

            return oldVal;

        }

        public int size() {

            return a.length;

        }

    };

}
```

5.局部类是四种嵌套类中用的最少的类。在任何“可以声明局部变量”的地方，都可以声明局部类。与成员类一样，局部类有名字，可以被重复使用。与匿名类一样，只有当局部类实在非静态环境中定义的时候，才有外围实例，它们也不能包含静态成员。与匿名类一样，它们必须简短以便不会影响到可读性。

```
public class outerTolocal {

    public String string;

    public int localInt;

    public void OtoLocal() {}

    public void localMthod(final int m, int n) {
        class local {
            //此类为局部类
            //局部类不需要加public 修饰符，因为这方法执行完这类就消失了
            int methodInt = m;
```

```
                /**
                 * 局部类的变量如果要等于外部类的方法的变量，
                 * 此时外部类的方法变量必须用final 修饰符
                 * 如：
                 */
                final int m;

                void localInner() {
                    System.out.println("local method");
                }
            }
            new local().localInner();//在另外的一个类的中不可以创建局部内部类的实例，只能在局部内部类中来创建。
        }
    }
```

6.简而言之，共有四种不同的嵌套类，每一种都有自己的用途。

（1）如果一个嵌套类需要在单个方法之外仍然可见的，或者它太长了，不适合于放在方法内部，就应该使用成员类。

（2）如果成员类的每个示例都需要一个指向外围实例的引用，就要把成员类做成非静态的；否则，就做成静态的。

（3）假设这个嵌套类属于一个方法的内部，如果你只需要在一个地方创建实例，并且已经有了一个预置的类型可以说明这个类的特征，就要把它做成匿名类；否则，就做成局部类。

泛型

2018年3月5日　　14:33

一、不要在新代码中使用原生态类型

1.原生态类型指的是不带任何实际类型参数的**泛型**名称。这种原生态类型是不安全的，但是因为兼容性的问题才被保留了下来，尽量不要在新代码中使用这种类型。

2.如果要表示一个可以包含任何对象类型的集合，使用**Object**作为类型参数，如Set<Object>。如果要表示只能包含某种未知对象类型的集合，使用通配符**？**，如Set<?>。

3.这条规则有两个例外，原因是泛型信息可以在运行时被擦除。

（1）在**类文字**中必须使用原生态类型。类文字是指在一个类名之后加上.class，类文字表示一个对象，这个对象代表着其自身的类型。比如List.class。

（2）使用instanceof时要用原生态类型。

```
if (o instanceof Set) { // Raw type
     Set<?> m = (Set<?>) o; // Wildcard type
}
```

二、消除非受检警告

1.非受检警告（**unchecked** warnings）会影响到代码的类型安全（抛出ClassCastException异常），要尽可能消除每一个非受检警告。

2.如果无法消除，同时可以证明引起警告的代码是类型安全的，（只有这种情况下才）可以用一个@SuppressWarnings("unchecked")注释来禁止这条警告。但是，要在尽可能小的范围你使用这条注释，尤其不要在整个类上使用，而应该声明一个新的局部变量，然后把注释用在上面。每当使用这条注释时，都要再加一条注释用来说明为什么这么做是安全的。

三、列表优先于数组

1.数组是**协变**的，即子类的数组可以赋值给父类的数组。列表是不可变的，其子类的列表不可以赋值给父类的列表。

2.数组运行时才知道并检查它们的元素类型的约束（这称为可具体化），所以有运行时安全。列表运行时包含的信息比它的编译包含的信息更少（这成为不可具体化），所以没有运行时安全。

3.一般来说，列表和数组不可混用，如果因混用而出错，第一反应是用列表代替数组。

四、优先考虑泛型

1.总而言之，使用泛型比使用需要在客户端代码中进行类型转换的类型来的更加安全，也更加容易。**设计新类型的时候，确保不需要这种类型转换就可以使用。这通常意味着要把类做成泛型的**，这对于这些类型的新用户来说会变得更加轻松，又不会破坏现有的客户端程序。

2.Stack类是泛型化的主要备选对象：

```
public class Stack {

    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
```

```
    public Object pop() {
        if(size == 0) {
            throw new EmptyStackException();
        }
        Object result = elements[--size];
        elements[size] = null;
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if(elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }

}
```

将类泛型化的第一个步骤，就是给明添加一个或者多个类型参数。在这个例子中有一个类型参数，它表示堆栈的元素类型，这个参数的名称通常为E（将这个类强化为带一个或多个泛型参数的类）。但是要考虑到数组不能是泛型的（new E[DEFAULT_INITAL_CAPACITY]是非法的），所以要采取一些办法。

（1）创建一个Object的数组，并将它转换为泛型数组类型。

```
@SuppressWarnings("unchecked")
public StackOfGeneric() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

编译器仍然出现警告，这种用法是合法的,但（整体上而言）不是类型安全的。

（2）直接把属性中的E[]改为Object[]。这样需要在构造器中强制转化，而且在pop()也需要转化。

```
@SuppressWarnings("unchecked")
public StackOfGeneric() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}

public E pop() {
    if (size == 0)
        throw new EmptyStackException();
    @SuppressWarnings("unchecked") E result = (E)elements[--size];
    elements[size] = null;
    return result;
}
```

禁止创建泛型数组，往往偏向于第二种解决策略。但由于第二种需要多次转换，所以实际上第一种更常用。

四、优先考虑泛型方法

1.编写泛型方法与编写泛型类型类似。

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
        Set<E> result = new HashSet<E>(s1);
        result.addAll(s2);
```

```
            return result;
        }
```

为了不产生警告，需要声明类型参数的**类型参数列表**，处在方法的修饰符及其返回类型之间。在这个示例中，类型参数列表为<E>,返回类型为Set<E>。利用**有限制的通配符类型**，可以使这个方法更加灵活。

2.泛型方法的一大特性类型推导。泛型方法的类型参数根据传入的实际参数的类型而定，不需要在调用时指明。

3.可以根据需要进行类型限制。比如要求某个方法实现互相比较的功能：

```
    public class testComparable {
        public <T extends Comparable<T>> T calculateMax(List<T> list) {
            Iterator<T> iterator = list.iterator();
            T result = iterator.next();
            while (iterator.hasNext()) {
                T t = iterator.next();
                // 需要判断大小
                if (result.compareTo(t) < 0) {
                    result = t;
                }
            }
            return result;
        }
```

<T extends Comparable<T>>理解为“针对可以与自身进行比较的每个类型T”

4.总之，泛型方法就像泛型一样，使用起来比要求客户端转换输入参数并返回值的方法来的更加安全，也更加容易。就像类型一样，你应该确保新的方法可以不用转换就能使用，这通常意味着要将它们泛型化。并且就像类型一样，还应该将现有的方法泛型化，使新用户使用起来更加轻松，且不会破坏现有的客户端。

五、利用有限制通配符来提升API的灵活性

1.如果要实现一种效果，可以添加的类型是类型参数的子类型，而不一定使用完全相同的类型，可以通过有限制的通配符来实现：

```
    public class Stack<E>{

        public Stack();
        public void push(E e);
        public E pop();
        public boolean isEmpty();

        public void pushAll(Iterable<? extends E> src);
    }
```

要实现一个功能，将堆栈中的元素弹出来，保存到一个容器中。也就是实现popAll这个API。使用这个API是有要求的，那就是容器的类型要完全跟当前的Stack的类型一致。如果想让堆栈中的元素可以存放在容器Collection<Object>中，可以通过有限制的通配符来实现。

```
    public class Stack<E>{

        public Stack();
        public void push(E e);
        public E pop();
        public boolean isEmpty();

        public void pushAll(Iterable<? extends E> src);
        public void popAll(Collection<? super E> dst);
    }
```

pushAll是数据的生产者；对生产者的进参数使用 <? extends E>，可以接受更多的类型，而

不是只是E这种类型，可以接受E及其子类的类型。
popAll是数据的消费者；对消费者出参数使用<? super E>，可以让堆栈的数据保存在多种类型的容器中，而不只是保存在Collection<E>。它可以保存在类型是E的父类的容器中。
总结起来就是PECS原则：**producer extends，comsumer super**
2.记住所有的Comparable和Comparator都是消费者。

五、优先考虑类型安全的异构容器
1.如果一个容器里面它存放的类型参数数目是不固定的，那么它就是一个**异构**的容器。这是有使用场景的，比如，使用一个容器来存放数据库中有任意列的一个行。因为每行列的数目是不固定的，每个列的类型也是不确定的，那么就可以使用一个异构的容器来表示这个行。一个列元素，用一个键值对来表示，并且键和值之间有有类型关系，键的类型跟值的类型是相同的，因此有不同类型的列。在实现这个目标时，让**键**参数化(也就是键可以表示不同的类型，而不是容器参数化，Set<T>是容器参数化)。

```
Map<Class<?>,Object> row = new HashMap<>();
```

Class在Java 5中已经泛型化了，也就是说String.class属于Class<String>类型，Integer.class属于Class<Integer>类型等。Class<T>就可以用来接受任何合适的类型。
2.这种模式有两个局限性。
（1）恶意的客户端可以通过以原生态形式使用class对象轻松地破坏类型安全。为了确保在运行时类型安全可以使用动态转换。

```
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

（2）它不能用在不可具体化的类型中。换句话说，你可以保存Runnable或Runnable[]，但是不能保存List<Runnable>。

枚举和注解

2018年3月5日　　14:34

一、用enum代替int常量

1.每当需要一组固定常量的时候，使用枚举类型。枚举类型中的常量集并不一定要始终保持不变。

2.除了完善了int枚举模式的不足之外，枚举类型还允许添加任意的方法和域，并实现任意的接口。如果想将数据与它的常量关联起来，可以在枚举中添加方法。

```
public enum Planet {

        MERCURY(3.302e+23, 2.439e6),

        VENUS (4.869e+24, 6.052e6),

        EARTH (5.975e+24, 6.378e6),

        MARS (6.419e+23, B.393e6),

        JUPITER(1.899e+27, 7.149e7),

        SATURN (5.685e+26, 6.027e7),

        URANUS (8.683e+25, 2.556e7),

        NEPTUNE(1.024e+26, 2.477e7);

        private final double mass;

        private final double radius;
       private final double surfaceGravity;

        private static final double G = 6.67300E-11;

      // privvate constructor

     Planet(double mass, double radius) {

              this.mass = mass;

              this.radius = radius;

              surfaceGravity = G * mass / (radius * radius);

      }

      public double mass() { return mass; }

      public double radius() { return radius; }

      public double surfaceCravity() { return surfaceGravity; }
```

```
        public double surfaceWeight(double mass) {

                return mass * surfaceGravity;
        }

    }
```

3.如果要将不同的行为与每个枚举常量关联起来：在枚举类型中声明一个**抽象的apply方法**，并在特定于**常量的类主体**中，用具体的方法覆盖每个常量的抽象apply方法。

```
    public enum Operation {

        PLUS(" + ") {

                double apply (double x, double y) { return x + y; }

        },

        MINUS(" - ") {

                 double apply (double x, double y) { return x - y; }

        },

        TIMES(" * ") {

                 double apply (double x, double y) { return x * y; }

        },

        DIVIDE(" / ") {

                 double apply (double x, double y) { return x / y; }

        };

        private final String symbol;

        Operation(String symbol) { this.symbol = symbol; }

        @Override public String toString() { return symbol; }

        abstract double apply (double x , double y);

    }
```

4.如果多个枚举常量同时共享相同的行为，则考虑**策略枚举**。策略枚举就是在原本的枚举中增加一个用来表示策略的嵌套枚举。

```
    //The strategy enum pattern

    enum PayrollDay {

          MONDAY(PayType.WEEKDAY),

          TUESDAY(PayType.WEEKDAY),
```

```
        WEDNESDAY(PayType.WEEKDAY),
        THURSDAY(PayType.WEEKDAY),
        FRIDAY(PayType.WEEKDAY),
        SATURDAY(PayType.WEEKEND),
        SUNDAY(PayType.WEEKEND);
        private final PayType payType;
        Payroll Day ( PayType payType ) { this.payType = payType; }
        double pay ( double hoursWorked , double payRate ) {
              return payType.pay ( hoursWorked , payRate );
        }
        //The strategy enum type
        private enum PayType {
              WEEKDAY {
                        double overtimePay(double hours, double payRate) {
                              return hours <= HOURS_PER_SHIFT ? 0 :
                                    (hours - HOURS_PER_SHIFT) * payRate /
2;
                        }
               },
              WEEKEND {
                       double overtimePay(double hours, double payRate) {
                                return hours * payRate / 2;
                        }
               };
               private static final int HOURS_PER_SHIFT = 8;
               abstract double overtimePay(double hrs, double payRate);
               double pay(double hoursWorked, double payRate) {
```

```
                        double basePay = hoursWorked*payRate;

                        return basePay + overtimePay ( hoursWorked, payRate );

                    }

                }

    }
```

二、用实例域代替序数

所有枚举都有一个**ordinal**方法用来返回序号（从0开始）。

```
    public enum Ensemble {
        SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET, DECTET;

        public int numberOfMusicians() {
            return ordinal()+1;
        }
    }
```

这个枚举看起来不错，但是维护起来很麻烦，一旦常量重新排序numberOfMusicians方法就被破坏了。解决的方法为：

```
    public enum Ensemble {
        SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5), SEXTET(6), SEPTET(7),
    OCTET(8),
        DOUBLE_QUARTET(8), NONET(9), DECTET(10), TRIPLE_QUARTET(12);

        private final int numberOfMusicians;

        Ensemble(int size) {
            this.numberOfMusicians = size;
        }

        public int numberOfMusicians() {
            return numberOfMusicians;
        }
    }
```

三、用EnumSet代替位域

1.位域是指信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有0和1两种状态，用一位二进位即可。所谓“位域”是把一个字节中的二进位划分为几 个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

```
    public class Test {
        public static final byte STYLE_BOLD          = 1<<0; // 1
        public static final byte STYLE_ITALIC        = 1<<1; // 2
        public static final byte STYLE_UNDERLINE     = 1<<2; // 4
        public static final byte STYLE_STRIKETHROUGH = 1<<3; // 6

        //Parameter is bitwise OR of zero or more STYLE_ constants
        public void applyStyles(int styles) { ... }
    }
```

这种方法让我们将OR位运算将几个常量合并在一个集合中，称作**位域**。

```
text.applyStyles(STYLE_BLOD | STYLE_ITATIC)
```

2.如果枚举类型要用在集合(Set)中,就没有理由用位域来表示它。

```
public class Text{
        public enum Style{BOLD,ITALIC,UNDERLINE,STRIKETHROUGH}
        public void applyStyles(Set<Style> styles){
            //实现方案
        }
    }
```

客户端代码可能为:

```
text.applyStyles(EnumSet.of(Style.BOLD,Style.ITALIC));
```

四、用EnumMap代替序数索引

如果要索引数组，不要用序数，而要用**EnumMap**。注意EnumMap的构造其要用到键类型的Class对象。

```
public static void main(String[] args) {
        Herb[] garden = {new Herb("a", Herb.Type.ANNUAL), new Herb("b",
Herb.Type.BIENNIAL)};
        Map<Herb.Type, Set<Herb>> herbByType = new EnumMap<Herb.Type,
Set<Herb>>(Herb.Type.class);
        for(Herb.Type t : Herb.Type.values())
            herbByType.put(t, new HashSet<Herb>());
        for(Herb h : garden)
            herbByType.get(h.type).add(h);
        System.out.println(herbByType);
}
```

如果要进行两次索引或者多次索引，也是可以实现的。

```
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT(SOLID,LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

        private final Phase src;
        private final Phase dst;

        Transition(Phase src, Phase dst) {
            this.src = src;
            this.dst = dst;
        }

        private static final Map<Phase, Map<Phase, Transition>> m =
                new EnumMap<Phase, Map<Phase, Transition>>(Phase.class);
        static {
            for(Phase p : Phase.values())
                m.put(p, new EnumMap<Phase, Transition>(Phase.class));
            for(Transition t : Transition.values())
                m.get(t.src).put(t.dst, t);
        }

        public static Transition from(Phase src, Phase dst) {
            return m.get(src).get(dst);
```

```
        }
    }
}
```

五、用接口模拟可伸缩的枚举

枚举类型是**不可扩展**的，也就是说为了安全起见无法在一个已有的枚举中增加新的常量。但由于接口是扩展的，所以可以先定义一个接口，然后根据需要定义不同的枚举类型的实现。

```
public interface Operation {
    double apply(double x,double y);
}
public enum BasicOperation implements Operation{
    PLUS("+"){
        public double apply(double x, double y) {
            return x + y;
        }
    },
    MINUS("-"){
        public double apply(double x, double y) {
            return x - y;
        }
    },
    TIMES("*"){
         public double apply(double x, double y){
             return x * y;
         }
    },
    DIVIDE("/"){
        public double apply(double x, double y) {
            return x / y;
        }
    };
    private final String symbol;
    BasicOperation(String symbol) {
        this.symbol = symbol;
    }
}

//增加一些新的实现
public enum ExtendedOperation implements Operation {

    EXP("^"){
        @Override
        public double apply(double x, double y) {
            return Math.pow(x , y);
        }
    },
    REMAINDER("%"){
        @Override
        public double apply(double x, double y) {
            return x % y;
        }

    };
```

```
        private final String symbol;
        ExtendedOperation(String symbol) {
               this.symbol = symbol;
        }
    }
```

六、注解优先于命名模式

1用一个词就可以描述注解，那就是**元数据**，即一种描述数据的数据。所以，可以说注解就是源代码的元数据。Annotation是一种应用于类、方法、参数、变量、构造器及包声明中的特殊修饰符。Annotations仅仅是元数据，和业务逻辑无关。比如最常见的@Override只是声明了某个方法是一个覆盖的方法。

2.Java 5提供了四中预定义的注解：

（1）@Documented－一个简单的Annotations标记注解，表示是否将注解信息添加在java文档中。

（2）@Retention－ 定义该注解的生命周期。

RetentionPolicy.SOURCE － 在编译阶段丢弃。这些注解在编译结束之后就不再有任何意义，所以它们不会写入字节码。@Override, @SuppressWarnings都属于这类注解。

RetentionPolicy.CLASS － 在类加载的时候丢弃。在字节码文件的处理中有用。注解默认使用这种方式。

RetentionPolicy.RUNTIME－ 始终不会丢弃，运行期也保留该注解，因此可以使用反射机制读取该注解的信息。我们自定义的注解通常使用这种方式。

（3）@Target － 表示该注解用于什么地方。如果不明确指出，该注解可以放在任何地方。以下是一些可用的参数。需要说明的是：属性的注解是兼容的，如果你想给7个属性都添加注解，仅仅排除一个属性，那么你需要在定义target包含所有的属性。

ElementType.TYPE:用于描述类、接口或enum声明

ElementType.FIELD:用于描述实例变量

ElementType.METHOD

ElementType.PARAMETER

ElementType.CONSTRUCTOR

ElementType.LOCAL_VARIABLE

ElementType.ANNOTATION_TYPE 另一个注释

ElementType.PACKAGE 用于记录java文件的package信息

（4）@Inherited － 定义该注释和子类的关系

除此之外，程序员也可以自定义注解。

3.除了为程序员开发编程工具的程序员（称为“工具铁匠”）以外，其他程序员没有多大必要自己定义注解类型，但是应该使用Java预定义的注解类型。

七、坚持使用Override注解

1.在Eclipse中覆盖方法：

http://blog.csdn.net/maybe_windleave/article/details/9152963

2.如果在需要覆盖父类方法的方法处添加Override注解，编译器就会防止程序员出错。

3.如果有一些方法，比如抽象类或者接口中的方法，可以不加Override注解。

八、用标记接口定义类型

1.标记接口：**没有包含方法声明的接口**，而只是指明一个类实现了具有某种属性的接口。例如，Serializable接口。

标记注解：特殊类型的注解，其中不包含成员。标记注解的唯一目的就是标记声明。例如，@Override。

2.标记注解和标记接口的选择

（1）如果标记是应用到程序元素而不是类或者接口，就必须用注解；

（2）如果标记只应用给类和接口，那么就标记接口优先于标记注解；

（3）如果要定义一个任何新方法都不会与之关联的类型，标记接口是最好的选择；

（4）如果想要标记程序元素而非类和接口，考虑到未来可能要给标记添加更多的信息；标记更适合于已经广泛使用了注解类型的框架，那么就该使用标记注解。
3.如果想要定义类型，一定要使用接口。

方法

2018年3月8日　　12:18

一、检查参数的有效性

1.对于公有方法，要用**@throws标签**指明违反参数限制时会抛出哪些异常。这样的异常通常为IllegalArgumentException,IndexOutOfBoundsException,NullPointerException。

2.对于非公有的方法，通常应该使用**assert**语句来检查参数。

```
private static void sort(long[] a,int offset,int length){
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    System.out.println("sort do something");
}
```

3.当有效性检查工作非常昂贵，或者根本是不切实际的，而且有效性检查已隐含在计算过程中完成，就不必进行检查了。比如一个方法能为对象列表排序，那这自然要求列表中的对象都是可以互相比较的。

4.在设计方法时，应该使其尽可能通用且能符合实际需要。假如方法对于它所接受的所有参数值都能完成合理的工作，那么对参数的限制就越少越好。

二、必要时进行保护拷贝

1.如果类具有从客户端得到或者返回到客户端的可变组件，类就必须保护性地拷贝这些组件。

```
import java.util.Date;

public final class Period {
    private final Date start;
    private final Date end;
    public Period(Date start,Date end) {
        if(start.compareTo(end) > 0){
            throw new IllegalArgumentException(start + " after " + end);
        }
        this.start = start;
        this.end = end;
    }

    public Date start(){
        return start;
    }

    public Date end(){
        return end;
    }
    //remainder omitted
}
Date start = new Date();
Date end = new Date();
Period period = new Period(start, end);
end.setYear(78);
```

虽然Period类本身想定义为不可变类，但是由于Date类是可变的，所以需要对构造器的每个可变参数进行**保护性拷贝**，用**备份对象**作为Period实例的组件。

```
public Period(Date start,Date end) {
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
```

```
        if(this.start.compareTo(this.end) > 0){
            throw new IllegalArgumentException(this.start + " after " +
    this.end);
        }
    }
```

注意保护性拷贝是在检查参数的有效性之前进行的，并且有效性检查是针对拷贝之后的对象，而不是原始对象。对于参数类型可以被不可信任方子类化的参数，请不要使用**clone**方法进行保护性拷贝。

2.虽然替换构造器可以保护对象，但是由于访问方法提供了对其内部可变成员的访问能力，所以对象还是可以改变。

```
    Date start = new Date();
    Date end = new Date();
    Period period = new Period(start, end);
    period.end().setYear(98);
```

解决办法是让访问方法返回可变内部成员的保护性拷贝：

```
    public Date end(){
        return new Date(end.getTime());
    }
```

三、谨慎设计方法签名

1.谨慎地选择方法的名称。方法的名称应该遵循标准的命名习惯。可以参考Java类库的API。

2.不要过于追求提供便利的方法。应该优先考虑设计功能齐全的方法，对于那些经常使用的操作，才为其提供快捷方法。否则，方法太多会使接口实现者和接口用户难以使用。

3.避免过长的参数列表。最多不超过四个参数。有三种方法可以缩短过长的参数列表：

（1）分解成多个方法，每个方法只需要这些参数的一个子集。

（2）创建辅助类（IO/VO），用来保存参数的分组。

（3）采取**Builder**模式。

4.对于参数类型，要优先使用**接口**而不是类。例如写方法时，禁忌使用HashMap类作为输入，而需要换成Map接口作为输入参数。这可以使你调用此方法时传入HashMap、TreeMap等他的实现。

5.对于boolean类型参数，要优先使用两个元素的枚举类型。

四、慎用重载

1.对于多个具有**相同数目参数**的方法来说，尽量避免使它们成为重载的关系。

```
    public class CollectionClassifier {
        public static String classify(Set<?> s) {
            return "Set";
        }

        public static String classify(List<?> lst) {
            return "List";
        }

        public static String classify(Collection<?> c) {
            return "Unknown Collection";
        }

        public static void main(String[] args) {
            Collection<?>[] collections = {
                new HashSet<String>(),
                new ArrayList<BigInteger>(),
                new HashMap<String, String>().values()
            };
```

```
            for (Collection<?> c : collections)
                System.out.println(classify(c));
        }
    }
```

由于编译器选择重载的依据是被调用方法所在对象的**编译时类型**，所以上述代码的最终输出三次Unknown Collection。重载不同于覆盖，覆盖版本的选择是在**运行时**进行的，所以“最为具体的”那个覆盖版本总是会被执行。

2.对于多个构造器，它们之间必然是重载的关系，在这种情况下，可以选择使用静态工厂方法。

3.如果非要用重载，至少应该避免出现“同一组参数只需经过类型转换就可被传递给不同的重载方法”的情况。如果连这种情况都不能避免，就应该保证当传递同样的参数时，所有重载方法的行为必须一致。

四、慎用可变参数

1.**可变参数方法**接受0个或者多个指定类型的参数。可变参数机制会创建一个数组，这个数组用来保存传递给方法的所有参数，然后再把这个参数数组传递给方法。

```
    static int sum(int… args)
    {
            int sum = 0;
            for(int arg : args)
                sum += arg;
            return sum;
    }
```

这种方法虽然灵活，但是会有缺陷。比如需要至少一个参数，那么就只能在运行时检查而不能在编译时检查。

2.可变参数方法的每次调用都会导致进行一次数组分配和初始化，如果这种性能的损失是必须避免的，但又需要可变参数的灵活性，可以有一种替代的方法：

```
    public void foo() {}
    public void foo() {int a1}
    public void foo() {int a1, int a2}
    public void foo() {int a1, int a2, int a3}
    public void foo() {int a1, int a2, int a3, int... rest}
```

假设确定某个方法大部分调用会有3个或者更少的参数，就声明该方法的5个重载，每个重载带有0至3个普通参数，当参数数目超过3个时，使用可变参数方法。

五、返回零长度的数组或者集合，而不是null

为了避免需要对null进行专门的处理，返回零长度的数组或者集合。

零长度的数组定义：

```
    int[] zeroArray = new int[0];
```

零长度的集合，可以使用集合框架的Collections的**emptyList()**方法或着直接使用共有静态final域EMPTY_LIST。

六、为所有导出的API元素编写文档注释

1.Java环境提供了一种被称为**Javadoc**的实用工具，从而使这项任务变得很容易。Javadoc利用特格式的文档注释，根据源代码自动产生API文档。

如何使用Javadoc？相关资料：

http://blog.csdn.net/nothing0318/article/details/7258523

2.Java 1.5发行版本中增加的三个特性在文档注释中需要特别小心：泛型、枚举和注解。

（1）当为泛型或者方法编写文档时，确保要在文档中说明所有的类型参数。

（2）当为枚举类型编写文档时，要确保在文档中说明常量，以及类型，还有任何公有的方法。

（3）为注解类型编写文档时，要确保在文档中说明所有成员，以及本身类型。对于该类型的

概要描述，要使用一个动词短语，说明当程序元素具有这种类型的注解时它表示什么意思。

通用程序设计

2018年3月8日　　20:47

一、将局部变量的作用域最小化

1.只有到了要使用变量的地方才去声明它，而不要把所有变量的声明放在代码的最前端。

2.**几乎**每个局部变量的声明都应该包含一个初始化表达式，如果还不能对它进行有意义的初始化，就应该推迟这个声明，直到可以初始化为止。

3.优先使用for循环而不是while，因为for语句内可以声明局部变量。

4.如果把两个操作合并到同一个方法之中，那么与其中一个操作相关的局部变量就有可能会出现在执行另一个操作的代码范围内。为了防止出现这种情况，只要把这个方法分为两个，每个方法都只执行一个操作。

二、for-each循环优先于传统的for循环

1.for-each循环的优势：

（1）简洁性。

（2）有效预防bug。

（3）不用担心性能损失。

```
    // 不简洁，同时易误导致bug
            for (Iterator<String> i = list.iterator();i.hasNext();) {
                for (Iterator<String> j = list2.iterator();j.hasNext();) {
                    doSomething(i.next(),j.next());
                }
            }

            // 简洁
            for (String s : list) {
                for (String s1 : list2) {
                    doSomething(s,s1);
                }
            }
```

2.不能使用for-each情况

（1）过滤

如果需要遍历集合，并删除选定的元素，就需要使用显示的迭代器，以便可以调用它的remove方法。

（2）转换

如果需要遍历所有元素，然后重新设定某些值，那么就需要李彪迭代器或者数组索引，以便设定元素的值。

（3）平行迭代

就是并行地同时遍历多个集合，需要显示地控制迭代器或者索引变量，以便所有迭代器或者索引变量都可以得到同步前移。

三、了解和使用类库

1.不需要重新发明轮子，如果要做的工作是十分常见的，那么很可能类库中的某个类就已经完成了这个功能。所以，第一个念头应该是使用标准类库。

2.应该熟悉的类包括**java.lang,java.util,java.io和java.util.concurrent**。

四、如果需要精确的答案，请避免使用float和double

1.如果你想计算十进制小数点，并且不介意非基本类型带来的不便，那就用BigDecimal；使用BigDecimal还有个额外的好处就是它允许你完全控制舍入，每当一个操作涉及舍入的时候它允许你从8种舍入模式中选择一种。如果你正确通过法定要求的舍入行为进行业务计算使用BigDecimal时非常方便的；如果性能十分关键，并且你又不介意自己记录十进制的 小数点，

而且涉及数值不大，就可以使用int或者long； 如果数值范围没有超过9位的十进制数字，可以用int；如果数值范围没有超过18位的十进制数字，用long 如果范围超过了18位数，那我们必须使用BigDecimal。
2.在使用BigDecimal时需要注意两点:
（1）不要使用double类型的参数来创建BigDecimal对象，那样即使使用BigDecimal来运算，运算结果也会产生精度不准的问题。
（2）BigDecimal是**不可变**的对象，这就意味着每次运算的结果都会产生一个新的BigDecimal对象。

五、基本类型优先于装箱基本类型
1.当可以选择的时候，优先使用基本类型而不是其包装类。绝对不要混用基本类型和包装类，因为这种情况下包装类会自动拆箱，而如果这个包装类对象没有初始化，就会抛出异常。
2.自动装箱减少了使用包装类的繁琐性，但是仍有风险。
3.必须使用包装类的情况
（1）在参数化类型中，必须使用包装类。
（2）进行反射的方法调用时，必须使用包装类。

六、如果其他类型更适合，则尽量避免使用字符串
1.字符串不适合代替其他的值类型
当一段数据从文件、网络、或者键盘设备，进入到程序之后，它通常以字符串的形式存在。有一种自然的倾向是让它继续保留这种形式，但是，只有当这段数据本质上确实是文本信息时，这种想法才是合理的。
2.字符串不适合替代枚举类型
枚举类型比字符串更加适合用来表示枚举类型的常量
3.字符串不适合代替聚集类型
如果一个实体有多个组件，用一个字符串来表示这个实体通常是不恰当的
4.字符串不适合代替能力表

七、当心字符串连接的性能
1.除非不关心性能，否则不要使用+连接字符串。因为字符串是不可变对象，当两个字符串连接时它们的内容都要被拷贝。应该使用StringBuilder的append方法进行连接。
2.使用字符数组，或者每次只处理一个字符串，而不是把它们组合起来。

八、通过接口引用对象
1.如果有**合适的接口类型**存在，那么对于参数、返回值、变量和域来说，都应该使用接口进行声明，这样做可以使得程序更为灵活。

```
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

当需求改变时，可以直接更改构造器的名称：

```
List<Subscriber> subscribers = new Vector<Subscriber>();
```

2.不适合使用接口的情况
（1）没有合适的接口存在。
（2）对象属于一个框架，而框架的基本类型是类。比如java.util.TimerTask。
（3）类提供了接口所没有的功能，而程序依赖于这种功能。

九、接口优先于反射机制
1.核心反射机制java.lang.reflect提供了通过程序来访问关于已装载的类的信息。
2.反射机制很强大，能胜任很复杂的任务，比如程序必须要与编译时未知的类一起工作，这时就需要反射机制。但是反射机制也有缺点，所以除非有特殊需求，否则不使用反射。

十、谨慎地使用本地方法
1.Java Native Interface(JNI，Java本地接口)，可以调用本地方法。这里的“本地”是指用其他语言（如C， C++） 编写的特殊方法。
2.可能需要使用使用本地方法的情况

（1）由于Java程序是运行在虚拟机之上的，虚拟机作为中间件，带来的平台无关性的好处的同时，也使得那些要求访问OS甚至硬件的底层操作变得无所适从。通过JNI可以调用C/C++等编写的代码，来提Java完成，比如读写Windows的注册表，取得硬盘的序列号等。
（2）由于一些“古老”的资源要使用“古老”的代码库，为其再开发一个Java版本是不划算的。于是JNI又派上用场了。
（3）最后是为了性能。Java的性能可能是所有语言中几乎最慢的了，关键性的部分可以通过JNI调用性能好的语言，如C/C++/Colob等。但是，随着Java版本的升级，性能在不断提高，如今绝大多数地方已经不值得使用JNI来提高性能了。
3.如果真的要使用本地方法，也要尽可能地少用，并且进行全面测试。

十一、谨慎地进行优化
1.任何优化都存在风险，有时候弄不好反而带来其他的问题
2.并不是性能优先。努力编写好的程序而不是快的程序。
3.对前人的代码，尤其是类似于Java API这样的成熟代码，进行优化，是不明智的（要是能优化，人家早就做了）。

十二、遵守普遍接受的命名惯例
1.Sun推荐包名按照域名的逆序来书写，而且全是小写字母，每一层尽量是一个英文单词（名词最好）。尽量一个单词，而且尽量不大于8个字母，所以鼓励使用缩写。例如，使用util而不是utilities（有人也用utils）。
2.类和接口（接口其实也是类，Java中万物皆类）的名字采用 Pascal命名法，即**每个单词的首字母均大写，各个单词间无连接符**。对于一些英文缩写，也推荐除首字母以外都小写。如，HttpUrl，而不是HTTPURL。这里尤其要注意的是仅有两个字母的缩写，如IO，ID，IP，最好还是写成 DiskIo，UserId，TerminalIp。
3.类成员（属性和方法）名、局部变量名要用**驼峰命**名法，即除首单词的首字母要小写外，其他同Pascal命名法。
4.常量，用全大写，各个单词之间用下划线“_”相连接。
5.getter/setter方法，getter/setter其实就是普通的方法，只是一种特定用途罢了。它们用于将被封装的私有属性对外提供访问的方法。通常是在属性名的前面加上 get 和 set ，再将属性名的首字母变大写。 这里的一个特殊地方是boolean型变量，除了之上的方法，也可以把 get 改为 is，如果属性名本身已经是以 is 开头了，就省掉这个 is 。这是 JavaBean 的规范，广义上讲，也可以用一个名词或名词短语，如：size，hashCode。
6.特殊方法和属性
（1）静态工厂方法：valueOf 和 getInstance 前者广泛用于对值类的类型转换，后者则出现在非值类的单例模式中
（2）类型转换方法：toType，如 toString，toArray
（3）返回当前对象的一个不同的视图：asType，如：asList，常用 Arrays.asList() 来将数组转换成List
（4）一些省略了开头的“is”的boolean型属性，如 initialized 和 composite
（5） 一些常用的通用属性，如：height，digits，size 等
（6）一些常用的通用方法，如：flush()，isEmpty() 等
7.在对英文单词的选择上，也尽量复合大多数人的习惯。 避免使用一些蹩脚的单词，而是使用常见的单词，而且最好是其他人也大多使用的单词。在对两个单词模棱两可时，可以在Javadoc中搜索一下，看看哪一个被类库使用的更多。例如，当你拿不定主意用 delete 还是 remove 时，到Javadoc中搜一下，你会发现 remove 的出现次数远比 delete 要多得多，可能仅仅是在物理上删除如文件或数据库中的记录的时候才用delete，一般对变量中内容的删除都使用remove。

异常

2018年3月9日　　20:34

一、只针对异常的情况使用异常

1. 除非程序可能出现异常，否则不要使用异常，更不要把异常用在正常的控制流上。
2. 如果类具有“状态相关”方法（比如迭代器的next方法），这个类往往也应该有个单独的“状态测试”方法（迭代器的hasNext方法）。另一种进行状态测试的方法是，如果对象处于不适当的对象，那么它就会返回一个可识别的值（比如null）。
3. 一般来说，“状态测试方法”比“返回可识别的值”要优越。但是如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，那么使用可识别的返回值是有必要的。

二、对可恢复的情况使用已检查的异常，对编程错误使用未检查的异常

1. Java语言提供了三种可抛出的结构：已检查的异常、运行时异常和错误，后两种统称为未检查的异常。
2. 如果希望调用者能够适当地恢复，那么就应该使用**已检查的异常**，并且应该提供一些辅助方法，帮助调用者获得一些有助于恢复的信息。
3. 运行时异常大都表示了前提违例的情况，最常见的一种就是数组下标越界；错误往往被JVM保留用于表示资源不足、约束失败等使程序无法继续执行的条件，这已经是个惯例。如果要自定义未检查的异常，那么它应该是**RuntimeException**的子类而不是Error的子类。

三、避免不必要的已检查异常

1.过多地使用已检查的异常会使API用起来很不方便，因为调用者必须在一个或者多个catch块中处理这些异常，或者它必须声明它抛出这些异常并将它们传播出去。

2.可以将已检查的异常变成未检查的异常，就是说把可能抛出异常的方法分为两个部分，第一个方法添加一个状态测试，用来判断是否应该抛出异常，第二部分用来处理异常。

```
try {
    obj.action(args);
} catch (TheCheckedException e) {
      ....
}
```

将这种组织形式重构为：

```
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    //Handle the exceptional condition
}
```

注意，如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，那么这种重构就是不恰当的。

四、优先使用标准的异常

1.Java提供了很多未检查的异常类，如果某个异常类满足自己的需求（经过语义分析之后的结果），那就优先使用这些异常类。重用现有代码的好处使API便于调用，同时也增加了可读性。

2.最常见的标准异常类如下：

异　常	使用场合
IllegalArgumentException	非null的参数值不正确
IllegalStateException	对于方法调用而言，对象状态不合适
NullPointerException	在禁止使用null的情况下参数值为null
IndexOutOfBoundsException	下标参数值越界
ConcurrentModificationException	在禁止并发修改的情况下，检测到对象的并发修改
UnsupportedOperationException	对象不支持用户请求的方法

如果笼统地划分的话，所有错误的方法调用都可归结为非法参数或者非法状态。

五、抛出与抽象相对应的异常

假设方法A要调用方法B，那么A叫做高层方法，B叫做低层方法。假设现在B抛出了已检查的异常，那么可以有以下几种解决方法

1.抛出与抽象相对应的异常——异常转译

更高层的实现应该捕获底层的异常，同时抛出可以按照高层抽象进行解释的异常。这种做法称为异常转译（exception translation）。

如果方法B抛出了NoSuchElementException这个受检异常，然而在方法A中调用方法B时，根据方法A中的逻辑，当遇到NoSuchElementException异常时，抛出一个IndexsOutOfBoundsException异常更为合适。

2.避免低层异常的出现

在调用底层方法之前确保它们会成功执行。 如果高层的A方法能够通过普通的判断语句保证底层的B方法永远也不会抛出异常，那么就可以不必处理B方法的异常。

3.绕开低层异常

如果无法避免低层异常，可以让更高层来悄悄地绕开这些异常，从而将高层方法的调用者与低层的问题隔离开来。使用适当地记录机制来将异常记录下来。 对于上面的例子，如果A方法实在不能避开B方法，只需要try-catch底层的异常，然后偷偷地通过记录机制把异常记录下来，这样A就不用做其他异常处理了。

六、每个方法抛出的异常都要有文档

为自己编写的每个方法所能抛出的异常建立javadoc文档。要为每个已检查的异常提供单独的throws子句，不要为未检查的异常提供throws子句。

七、在细节消息中包含能捕获失败的信息

异常的细节信息应该包含所有对产生该异常有贡献的参数和域的值。一种良好的做好就是在异常的**构造器**中引入这些细节信息。

八、努力使失败保持原子性

1.一般而言，失败的方法调用应该使对象保持在被调用之前的状态。这种属性叫做失败原子性。

2.实现失败原子性的常用方法

(1) 对象为**不可变对象**,那么对象创建出来就不能被修改了，也不需要维护。
(2) 执行操作之前检查参数的有效性。在对象状态被修改之前，先抛出异常。
(3) 调整计算处理的过程，使得任何可能会失败的计算部分都在对象状态被修改之前发生。
(4) 编写恢复代码，由其拦截操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。这样做并不提倡，因为错误代码编写遇到复杂的场景会很繁琐。
(5) 在对象的一份临时拷贝上执行操作，操作完成后，在使用临时拷贝的的结果代替对象的内容。也就是备份操作。

前三种方法更好，因为应该先考虑防患于未然，才考虑如何错误恢复。

3.错误通常是不可恢复的，如果方法抛出的是错误，那就不必保持失败原子性了。

4.一般而言，API文档应该说明如果不能保持失败原子性，那么对象将会处于什么状态。

九、不要忽略异常

绝对不要出现空的catch块，至少也要打印堆栈信息或将其抛给上层调用者，总之不能忽略异

常。

并发

2018年4月1日　　21:21

一、同步访问共享的可变数据

1.同步的作用实际上有两个：第一是阻止一个线程看到对象处于不一致的状态（比如在对同一个对象执行两次读操作的间隔中另一个线程修改了这个变量），第二是可以保证进入同步方法或者同步代码块中的每个线程都看到由同一个锁保护的之前所有的修改效果。

2.Java语言规范保证对一个变量的读写操作是原子的，除非这个变量是long或者double。

3.避免并发引起的问题的最好办法是将可变数据限制在单个线程中。如果一定要让多个线程共享可变数据，那么**每个读或者写方法都必须执行同步**。除了使用synchronized关键字进行加锁，还有一些其他的方法可以进行同步：

（1）将需要共享的数据设置为**volatile**，这个修饰符不执行互斥访问，但它保证数据的改变对任何线程来说都是**立刻可见**的。但如果对一个域的操作不是原子的，那就不能用volatile。

（2）使用java.util.concurrent.atomic，保证操作的原子性。

二、避免过度同步

1.为了避免死锁和数据破坏，不要在一个同步区域内调用外部的方法。更一般的做法是，同步区域内的工作量要尽可能少。

2.如果一个可变的类要并发使用，就应该是这个类变成是线程安全的，但是除非必需，否则不要在类的内部进行同步，而应该让客户端从外部同步（就是说客户必须利用某种同步方法包围每个方法调用）。

三、executor和task优先于线程

1.Excutor框架可以方便地管理任务队列。

2.如果可以，不要使用线程。关键的抽象是工作单元，称为任务，任务有两种：Runnable和Callable。执行任务的通用机制是excutor service。

四、并发工具优先于wait和notify

1.wait和notify方法能不用就不用，如果一定要用，那么必须保证：

(1) wait方法处在一个循环中。

(2) 优先使用notifyAll而不是notify。

2.Java提供了更高级的并发工具来优化代码：

(1) Executor框架

(2) 并发集合：优先使用ConcurrentHashMap、BlockingQueue。

(3) 同步器：CountDownLatch、CyclicBarrier。

五、线程安全性的文档化

1.文档中是否出现synchronized修饰符和线程安全与否毫无关联。

2.线程安全性分为几种常见的级别：

(1) 非可变（immutable）：这个类的实例对于客户代码而言是不变的。所以，不需要外部的同步。这样的例子包括String,Integer和BigInteger。

(2) 线程安全的（thread-safe）:这个类的实例是可变的，但是所有的方法都包含了足够的同步手段，所以这些实例可以被并发使用，无需外部同步。这些并发的调用将表现为按照某种全局一致的顺序，被依次执行。其例子包括了Random和java.util.Timer。

(3) 有条件的线程安全（conditionally thread-safe）：这个类（或者关联的类）包含有某些方法，它们必须被顺序调用，而不能受到其他线程的干扰。除此之外，这种线程安全级别与上一种情形一样。为了消除被其他线程干扰，客户代码在执行此方法序列期间，必须获得一把适当的锁。例子：Hashtable,Vector，他们的迭代器（iterator）要求外部同步。

(4) 线程兼容（thread-compatiable）：在每个方法调用的外部使用外部同步，此时这个类的实例可以被安全并发的使用。如ArrayList和HashMap。

(5) 线程对立（thread-hostile）：这个类不能安全的被多个线程并发使用，即使所有的方法调用都被外部同步包围。这样情况在于这个类的方法要修改静态数据，而这些静态数据可能会影响到其他线程。这样的类在JAVA库中很少，例如：System.runFinalizersOnExit方法是线程对立的，但是已经废弃了。

3.每个类都应该详细说明线程安全性的注解。有条件的线程安全类应该在文档中指明“哪个方法调用序列需要外部同步，以及在执行这些序列的时候要获得哪把锁”。无条件的线程安全类应该使用私有对象锁来代替同步的方法。

六、慎用延迟初始化

1.除非绝对必要，否则就不要延迟初始化。

2.在保证线程安全的情况下延迟初始化。

（1）普通模式

①. 使用final修饰符

```
private final FieldType field = computeFieldValue();
```

②. 使用synchronized

```
private FieldType field;

public synchronized FieldType getField() {
    if (field == null) {
        field = computeFieldValue();
    }
    return field;
}
```

（2）lazy initialization holder class 模式

这种模式适用于静态内部类中的静态域初始化。注意：静态内部类只有**在第一次使用的时候**才会被初始化。

```
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}

//外部类的方法
public static FieldType getField() {
    return FieldHolder.field;
}
```

当调用getField方法时，第一次读取FieldHolder.field，导致静态内部类进行初始化。这种模式的关键在于，getField方法不需要同步（因为只会在第一次调用getFiled方法时导致初始化），并且只执行一个域的访问，因此并没有增加更多的成本。
（3）双重检查模式
这种模式适用于对**实例域**的初始化。这种模式避免了在初始化之后，再次访问这个域时的锁定开销（在普通的方法里面，会使用synchronized对方法进行同步，每次访问方法的时候都要进行锁定）。这种模式的思想是：两次检查域的值，第一次检查时不锁定，查看其是否初始化；第二次检查时锁定，如果其没有被初始化，才会调用computeFieldValue方法对其进行初始化。如果已经被初始化了，就不会锁定了，另外该域被声明为***volatile***非常重要，因为要保证有序性：第一步，给field分配内存；第二步，调用FieldType的构造函数（这里其实是指computeFieldValue）来初始化成员变量；第三步，将field对象指向分配的内存空间（执行完这步field就为非null 了）。

⭐ 详解volatile：http://www.importnew.com/24082.html

```
    private volatile FieldType field;

    public FieldType getField() {
        FieldType result = field;
        if (result == null) {
            synchronized (this) {
                result = field;
                if (result == null) {
                    field = result = computeFieldValue();
                }
            }
        }
        return result;
    }
```

在上面的代码中，事实上，只要该域被初始化以后，无论如何再也不会进入第二次的条件语句判断，也就是说被初始化以后，访问的时候再也不会被synchronized锁定。
另外值得注意的是，result局部变量的使用，是为了保证在已经被初始化的情况下，原来的变量只被读取一次到局部变量result中，否则在比较的时候需要读取一次，返回的时候还需要读取一次。
（4）单重检查模式
对于可以重复初始化的实例域，可以使用单重检查模式（省去第二次检查，此时或许不需要严格同步）。

七、不要依赖于线程调度器
1.线程调度器会决定哪些线程将会运行，以及运行多长时间。但是线程调度器与操作系统有关，这意味着它可能是不可移植的。应该确保可运行线程的平均数量不明显多于处理器的数量，而保证可运行线程数量尽可能少的的主要方法是让每个线程做些有意义的工作。
2.不要依赖Thread.yield和线程优先级。

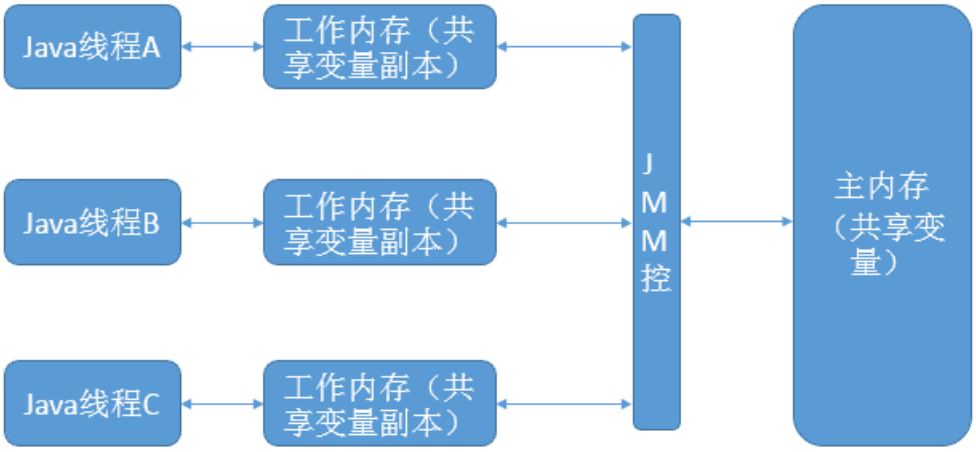
八、避免使用线程组
线程组是一个不成功的试验品，如果需要设计一个处理线程的逻辑组，那就使用线程池。

详解volatile

2018年4月2日　　17:34

http://www.importnew.com/24082.html

预备知识：Java内存模型
Java内存模型规定了所有的变量都存储在**主内存**中。每条线程中还有自己的工作内存，线程的工作内存中保存了被该线程所使用到的变量（这些变量是从主内存中拷贝而来）。线程对变量的所有操作（读取，赋值）都必须在工作内存中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。



一、volatile保证可见性
1.可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。
2.当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。
3.通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

二、volatile不能保证原子性
1.原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。
2.可以通过synchronized或lock，进行加锁，来保证操作的原子性。也可以通过java.util.concurrent.atomic包下提供的原子操作类。

三、volatile保证有序性
1.有序性：即程序执行的顺序按照代码书写的先后顺序执行。
2.指令重排：一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

```
int i = 0;

boolean flag = false;
```

```
i = 1;                  //语句1
flag = true;            //语句2
```

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。
3.volatile能在一定程度上限制指令重排，保证有序性。具体来讲就是，位于volatile变量之前的语句不会被排到volatile变量之后，位于volatile变量之后的语句也不会被排到其之前。

```
//x、y为非volatile变量
//flag为volatile变量

x = 2;          //语句1
y = 0;          //语句2
flag = true;    //语句3
x = 4;          //语句4
y = -1;         //语句5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。