EFFECTIVE JAVA读书笔记

作者：skyesx
原文链接：https://ld246.com/article/1467128778296
来源网站：链滴

<h2 id="toc_h2_0">第一章 引言</h2>
语言与模式之类的书在大学时才看过，而且是.NET版本的，现在给JAVA也补上一课。个人笔记无太大参考意义。
<h2 id="toc_h2_1">第二章 创建和销毁对象</h2>
<h3 id="toc_h3_2">第一条 考虑使用静态工厂方法替代构造器</h3>
优点：

静态工厂方法能提供名称，方便使用者使用
静态工厂方法能够不必每次调用都创建一个新实例
可以返回原返回类型的任意子类型

对外只提供接口，内部实现的类可变，方便以后替换升级
只提供了一个对外接口，屏蔽内部实现类，减轻了使用者的认知负担
编写静态工厂类的时候，实现可以不提供，由外部人员实现并注册到静态工厂类当中。方便 接口用者 使用统一的接口形式 而无需关注具体实现。如JDBC API的实现

使得代码更加简洁易读

缺点：

静态工厂方法如果不含有共有的或者受保护的构造器，那么它就不能被子类化
静态工厂方法本质与其他静态方法没有语法上的区别，只能人为地识别工厂方法

总结：在创建较为复杂的工具类的时候，要优先考虑使用静态工厂方法提供实例，减少 使用者习负担，基于父类接口的契约随时可替换子类实现。
<h3 id="toc_h3_3">第二条 遇到多个构造器参数时要考虑使用Builder模式</h3>
Builder形式如下：
<pre><code class="hljs">public class Example{
    private Integer a;//必
    private Integer b;//必
    private Integer c;//可
    private Integer d;//可
<pre><code class="highlight-chroma">public static class Builder{
    private Integer a;//必填
    private Integer b;//必填
    private Integer c;<span class="hljs-comment" style="box-sizing: border-box; color: #99

988; font-style: italic;">//可选
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">private&l
;/span> Integer d;<span class="hljs-comment" style="box-sizing: border-box; color: #99
988; font-style: italic;">//可选

    <span class=
hljs-function" style="box-sizing: border-box;"><span class="hljs-keyword" style="box-si
ing: border-box; color: #333333; font-weight: bold;">public <span class="h
js-title" style="box-sizing: border-box; color: #990000; font-weight: bold;">Builder</spa
>(Integer a,Integer b)<
span>{
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this&lt
/span>.a = a;
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this&lt
/span>.b = b;
    }

    <span class=
hljs-function" style="box-sizing: border-box;"><span class="hljs-keyword" style="box-si
ing: border-box; color: #333333; font-weight: bold;">public Builder <span c
ass="hljs-title" style="box-sizing: border-box; color: #990000; font-weight: bold;">c</sp
n>(Integer c)</span&gt
{
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this&lt
/span>.c = c;
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retur
 <span class="hljs-keyword" style="box-sizing: border-box; color: #333333; f
nt-weight: bold;">this;
    }

    <span class=
hljs-function" style="box-sizing: border-box;"><span class="hljs-keyword" style="box-si
ing: border-box; color: #333333; font-weight: bold;">public Builder <span c
ass="hljs-title" style="box-sizing: border-box; color: #990000; font-weight: bold;">d</s
an>(Integer d)</span
gt;{
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this&lt
/span>.d = d;
        <span clas
="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">retur
 <span class="hljs-keyword" style="box-sizing: border-box; color: #333333; f
nt-weight: bold;">this;
    }

    <span class=
hljs-function" style="box-sizing: border-box;"><span class="hljs-keyword" style="box-si
ing: border-box; color: #333333; font-weight: bold;">public <span class="h
js-title" style="box-sizing: border-box; color: #990000; font-weight: bold;">build</span

gt;()</span
gt;{

    }

}

<span class="hl
s-function" style="box-sizing: border-box;"><span class="hljs-keyword" style="box-sizi
g: border-box; color: #333333; font-weight: bold;">private <span class="hl
s-title" style="box-sizing: border-box; color: #990000; font-weight: bold;">Example</sp
n>(Builder builder)</s
an>{
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this</
pan>.a = builder.a;
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this</
pan>.b = builder.b;
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this</
pan>.c = builder.c;
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">this</
pan>.d = builder.d;
}
</code></pre>
}
</code> <code class="hljs">useage:
Example.Builder(1,2
).d(3).build();</code> </pre>
以上为Builder模式。
优点：

若某个类有很多的参数的话，构造器模式能够提供一个清晰可读的代码。（使用 很多参数类构造器时，经常出现放错参数的位置的情况，核对参数的位置十分痛苦）
使用这种方法创建出来的对象可以保证对象的完整性*（使用类似JAVABEAN的形式，逐个设置性值会导致对象还没正确初始化就已经被外部可用了，builder模式还可以在build的时候检查对象属的完整性）*
设置好的Builder生成了一个很好的抽象工厂，可以生成多个所需的对象。

缺点：

创建对象需要先创建Builder对象，增加了消耗
增加了码代码的工作量，最好必要时才用

总结： 当有很多初始化参数且要求保证对象完整性的时候就用这个吧。
<h3 id="toc_h3_4">第三条 Singleton的三种实现形式</h3>
Singleton指仅仅被实例化一次的类，通常用来代表那些本质上唯一的系统组件，如文件系统。类变成Singleton会使得客户端很难进行测试，因为singleton难以被MOCK。
第一种实现形式：直接使用静态字段 第二种实现形式：使用静态方法getInstance() 第三种实现式：使用枚举类定义，如下
<pre><code class="hljs">public enu

 Singleton{
INSTANCE;
<pre><code class="highlight-chroma">private Singleton(){
  //init...
}
</code></pre>
</code> <code class="hljs">}</code> </pre>
第二种形式的优势是，提供了灵活性，当我们想改变Singleton的唯一性的时候（如变成线程唯什么的）可以简单的进行修改。
第三种的形式与共有域方法相似，但是更简洁，且无偿的提供了序列化机制，绝对防止多次实例。（第一种，第二种形式可以通过反射等手段新建一个实例。第一二中形式Singleton需要序列化时还要做额外处理，反序列化后才能保证唯一性）
总结：单元素的枚举值是实现Singleton的最佳形式。
<h3 id="toc_h3_5">第四条 为类增加 私有构造器 强调类不可实例化</h3>
某些工具类并不需要实例化，实例化也没有意义，那么可以给它家一个私有的构造器。如java.util Collections
<h3 id="toc_h3_6">第五条 避免创建不必要的对象</h3>
没什么好说的，在对象的职责单一明确的情况下，没必要重复创建相同的对象来进行相同的工作
创建小对象的代价是很小的，JVM很快就会把小对象的内存给回收掉，若创建小对象能提高程序读性，那么久创建一些小对象。
但若创建对象的代价十分高昂，如数据库连接，那么就有必要把这些对象缓存起来，重复使用了
<h3 id="toc_h3_7">第六条 避免内存泄漏</h3>
容易产生内存泄漏的情况：

自己管理内存。如自己维护一个存储池，存储池的内容有没有效，是通过类内部定义的一些数据判断识别的。
使用自定义的缓存，没有定义清理方法。

WeakHashMap，当WeakHashMap外再无对缓存对象的引用，那么对应对象就会被移除
LinkedHashMap有一个removeEldestEntry方法，可以移除最老的一个对象

监听器和回调。客户端注册了回调，但没有显式取消。

<h3 id="toc_h3_8">第七条 避免使用终结方法</h3>
终结方法 执行时间不可预测，而且有很多坑，一般情况下不必使用终结方法。
若存在外部资源需要释放，那么可以提供一个显式关闭的接口，供使用者调用。
但调用者有可能忘记调用这个资源释放的接口，那么对应的对象编写者可以在终结方法中写上释资源的代码。毕竟最终放了好过一直没放。
但引入终结方法的对象JVM回收的执行效率会大大降低。需要认真考虑这个问题。
并且如果子类复写了父类的终结方法，那么除非 子类显式调用父类终结器，否则 父类终结器方并不会执行。
当普通对象需要传给本地方法执行时，普通对象会在本地方法中变成一个本地对等体。这个本地等体不会被JVM回收机制回收掉，因此需要给定终结方法。（本段话不太确定，可能有误）
<h2 id="toc_h2_9">第三章 对于所有对象都通用的方法</h2>
<h3 id="toc_h3_10">第八条 Object.equals</h3>
默认的Object.equals适用的常见场景：

每个实例本质都是唯一的（不同JAVA实例不相等）
不关心是否逻辑相等（不关心不同实例是否指代同一个事物）

需要复写equals的大多是值类，值才是代表他们是否相等的依据。复写equals需要遵循以下约定

自反性（若没有，则Set.contains会判断失败）
对称性(若没有，数组/SET contains判断可能失败)
传递性
一致性（多次调用equals都返回相同的结果）
x非null,那么x.equals(null)必须返回false

结论：尽量保持equals语义简单，尽量不要引入继承比较等复杂语义场景。不要将equals的入参 Object替换成其他的类型，会大大提高复杂度。
<h3 id="toc_h3_11">第九条 覆盖equals时总要覆盖hashCode</h3>
不覆盖的话，会影响哈希算法的使用。哈希算法当遇到碰撞时，需要用到equals
复写hashCode的约定：

应用程序执行期间，只要对象的equals方法的比较操作所用到的信息没有被修改，那么这个对象hashCode方法必须如一的返回同一个整数。
两个对象相等，那么hashCode必须相等。
两个对象equals不等，不一定要返回不等的hashCode,但返回不同的hashCode能减少碰撞，提效率

<h3 id="toc_h3_12">第十条 始终要覆盖toString</h3>
规范建议把对象的所有内容打印出来，方便阅读编程调试
<h3 id="toc_h3_13">第十一条 谨慎的覆盖clone</h3>
没看太懂，不常用，多坑。结论：不复写。
<h3 id="toc_h3_14">第十二条 考虑实现Comparable接口</h3>
实现Comparable接口可以简单的利用很多集合工具类，如 Arrays.sort,TreeSet
实现的约定：

x.compareTo(y) 与 y.compareTo(x)的 正负符号相同
compareTo具有传递性
建议 x.compareTo(y) == 0,那么x.equals(y)

<h2 id="toc_h2_15">第四章 类和接口</h2>
<h3 id="toc_h3_16">第十三条 使类和成员的可访问性最小化</h3>
没什么好说的，就这样
<h3 id="toc_h3_17">第十四条 在共有类中 使用访问方法 而非公有域</h3>
同上，公有域无法修改实现，不够灵活,但在包内部或类内部使用直接访问字段也没啥坏处
<h3 id="toc_h3_18">第十五条 不可变对象</h3>
好处：

线程安全，可以方便地共享
更方便的用到MAP,SET中
容易使用，坑少

缺点：

性能消耗大

一个类应该优先考虑实现成不可变类
实现不可变对象的五条原则：

不要提供任何会修改对象状态的方法
保证类不会被扩展
使所有的域都是final的
使所有的域都是私有的
确保外部不能拿到可变组件的引用

若某个计算过程需要对不可变对象进行大量变更，可提供一个不可变对象的协助类，如 String 应的 StringBuilder
<h3 id="toc_h3_19">第十六条 复合优先于继承</h3>
若父类子类不是同一个人实现，不在同一个包时，尽量避免继承的理由：

如果我们不知道父类实现的细节，继承类复写的方法有可能无法达到预期的效果
父类实现的细节有可能随着版本的不同而不同，子类扩展有可能因此失效

结论:父类实现细节怎么变，子类都无法预测与干预，因此继承不可控的父类，并OVERRIDE或者展方法，干预父类的细节是很危险的,继承这做法违反了封装原则。
复合是解决上述问题的一个好办法，因为复合模式依赖的是接口，而非实现细节。
不用扩展现有的类，而是在新的类中增加一个私有域，它引用现有类的一个实例，这种设计叫复，因为原有的类成为新类的一个组件。 新类的每个实例方法都可以调用现有类实例中的方法，并返回的结果。 新类可以在调用现有方法的前后加入自己的一些额外实现代码。
<h3 id="toc_h3_20">第十七条 要么为继承而设计，并提供文档说明，要么就禁止继承</h3>
若一个类是为继承而设计的，那么在父类中的每个Public或者Protect方法必须包含调用了 哪些覆盖方法 的自用性说明（即说明类内部的实现中是怎样使用这个被覆盖的方法的）。
一旦上述自用性说明发布出去之后，那么这个类以后的修改都必须遵守自用性说明中的契约
一个好的API文档应该描述的是 WHAT TO DO 而不是 HOW TO DO，但这种为继承而生的类文档与这个说法相违背，但 也没啥好办法。
为继承而设计的类还需要遵守一个约束，那就是 构造方法 不要调用可被覆盖的类，否则可能会现意想不到的情况，因为 子类复写的方法 将会在子类的初始化任务还没有完成的时候就被调用了。
<h3 id="toc_h3_21">第十八条 接口优先于抽象类</h3>
接口优点

有共同祖先X的的一些类,若只需少部分扩展某些新功能，那么可以通过新增实现的接口来扩展某功能，而无需担心影响其他的类
接口是实现MIXIN（指代为为类加上一些其他额外属性）的理想选择（貌似跟上一条没啥区别）
接口允许我们构造非层次结构的类型框架。如 某个人 既是 音乐家 ，又是 作家，那么这个人MIXN 音乐家 及 作家 的接口即可。
有了接口，可以更为优雅的实现Wrapper class模式（所以Spring代理最优是接口?果不是的话，代理的对象会侵入实现细节么？待确认）

抽象类有一个好处，就是可以提供一些基础骨架的实现，方便使用者快速开发。对于接口，接口可以提供一些实现了骨架的类。这样，开发者就可以通过 继承骨架类 快速实现对应的接口。如果不使用extends,那么，可以使用wrapper class模式，结合骨架类 给某个现有类快速MIXIN对应的接口
骨架类按照惯例命名方式为 "Abstract" + InterfaceName ,如 AbstractSet,AbstractCollection。
接口缺点

一旦接口被发布且广泛使用后无法新增方法

什么时候选择继承？ 演变的容易性（新增功能之类的）比灵活性和功能更为重要的时候。
<h3 id="toc_h3_22">第十九条 接口只用于表示可进行某种操作，其他任何类型的使用都是不合理的

/h3>
一个接口里只包含一些常量，目的是为了实现该接口的类可以使用这些常量，这种做法是不正确。 若需要导出常量，可以另外定义一个类，并 Static Import
<h3 id="toc_h3_23">第二十条 类层次优先于标签类</h3>
没说啥东西，略过
<h3 id="toc_h3_24">第二十一条 使用 函数对象 实现策略模式</h3>
可以使用 函数对象 来代替 函数指针的作用，实现策略模式。如用到 Comparator这接口的排序等
<h3 id="toc_h3_25">第二十二条 优先考虑使用静态成员类</h3>
讲了 静态内部类，非静态内部类，匿名类，局部类运用的一些场景，运用场景都比较明确，不记录
<h2 id="toc_h2_26">第五章 泛型</h2>
<h3 id="toc_h3_27">第二十三条 不要在新代码中使用原生态类型（要使用泛型）</h3>
略
<h3 id="toc_h3_28">第二十四条 消除非受检警告</h3>
当编译器有 非受检警告 时，需要通过泛型消除。如果无法消除，确认代码安全后，可用SuppresWSarning("unchecked")来禁止告警，并说明为什么。
<h3 id="toc_h3_29">第二十五 列表优于数组（可避免ClassCastException）</h3>
数组是协变的（convariant），指代 如果 Sub是 Super的子类，那么Sub[] 也是 Super[]的子类 这意味着下述的赋值在编译时是合法的
<pre><code class="hljs">String[] sub = new String[]{""};
Object[] super = new Object[0];
super = sub;//能成功值
//以下赋值，编译时无错误，因为Integer确实是Object的子类，能放进去，
//但是运行时会报错，因为 数组会对放进来的类型进行检查
subper[0] = 1;</code></pre>
泛型列表的设计就是为了避免这种 运行时类型转换错误的。List<Sub> List<Super>间并没有任何继承关系，是完全不同的两个类型。
JAVA中还不允许使用 泛型数组 如 List<Sub>[] 是被不能通过编译的。如果能通过编译，么以下场景还是会出现类型转换异常的错误
<pre><code class="hljs">List<String>[] strLists = new List<<spa class="hljs-keyword">String>[1];//编译报，但我们假设能通过
List<Integer&gt intList = Arrays.asArray(1);
Object[] objList = strLists;// Object是List<String>的父类，因此，这个赋值可以成功
Object[0] = intList;//能赋值成功，因为运行时是抹除了泛信息的，运行时List<Strig>和List<Inger>是同一个类型，因此数组赋值的类型检查不会报错
String s = strLists[0].get(0);//会有运行时错误，Iteger无法强转为String</code></pre>
由于上述原因，因此，不允许泛型数组的存在。
<h3 id="toc_h3_30">第二十六 实现优先考虑泛型类</h3>
略
<h3 id="toc_h3_31">第二十七 实现优先考虑泛型方法</h3>
略

<h3 id="toc_h3_32">第二十八 利用有限制通配符来提升API的灵活性</h3>
PECS:producer extends ,customer super. 如下例子：
<pre><code class="hljs">public class
/span> Collections {
    public static <T> voi
 copy(List<? sup
r T> dest,   List<? <span class="hljs-keyw
rd">extends T> src)
    {
        for (int i=<sp
n class="hljs-number">0; i<src.size(); i++)
        dest.set(i,src.get(i))

    }
}</code></pre>
extends有如下特性
<pre><code class="hljs">List<? <span class="hljs-ke
word">extends Number> f = new

ArrayList<Integer>();//不报错
f.add(1);//编译报错</span

Number n = f.get(0);
span class="hljs-comment">//不报错</code></pre>
super有如下特性
<pre><code class="hljs">List<? <span class="hljs-ke
word">super Number> f = new ArrayList&lt
Object>();//不报错</sp
n>
f.add(1);//不报错
Object n = f.get(<spa
 class="hljs-number">0);//只能拿到Object</c
de></pre>
extends能推断出方法自身产生的对象的最精确的值（extends 的那个类），但不能接受对应的
型入参，因为无法保证类型安全
super不能推断出返回对象的类型，但是能接受任何类型的入参
<h3 id="toc_h3_33">第二十九 若要使用异构容器，优先使用类型安全的异构容器</h3>
<pre><code class="hljs">public class Favorites{
    private Map<Class<?>,<span class="hljs-key
ord">Object> map = <span class="hljs-key
ord">new HashMap<Class<?>,<span c
ass="hljs-keyword">Object>();
    public <T> void</
pan> put(Class<T> key,T value){
        map.put(key,<
pan class="hljs-variable">key.cast(value));//cast用于
上类型检测
    }
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">public <T> T <span class="hljs-built_in" style="box-
izing: border-box; color: #0086b3;">get(Class<T> <span cla
s="hljs-variable" style="box-sizing: border-box; color: teal;">key){
    <span class=

hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">return&lt
/span> key&lt
/span>.cast(<span class="hljs-built_in" style="box-sizing: border-box; color: #0086b3;"
gt;map.<span class="hljs-built_in" style="box-sizing: border-box; color: #008
b3;">get(<span class="hljs-variable" style="box-sizing: border-box; color:
eal;">key));
}
</code></pre>
</code> <code class="hljs">}</code> </pre>
<h2 id="toc_h2_34">第六章 枚举和注解</h2>
<h3 id="toc_h3_35">第30条 用enum代替int常量</h3>

类型安全
更高可读性
枚举功能更强大，可附加功能
可用枚举实现策略模式（每个枚举值都是一个类）

例子 public enum Operator{ PLUS("+"){ double apply(double x,double y){ return x + y; } },
MINUS("-"){double apply(double x,double y){ return x - y; } }
<pre><code class="hljs">private <span class="hljs-ke
word">String symbol;
    Operator(String symbol){
        this.symbol = symbol;
    }
<pre><code class="highlight-chroma"><span class="highlight
cl"><span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight:
bold;">abstract apply(<span class="hljs-keyword" style="box-sizing: borde
-box; color: #333333; font-weight: bold;">double x,<span class="hljs-keyw
rd" style="box-sizing: border-box; color: #333333; font-weight: bold;">double</span&gt
 y);

<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">public</
pan> toString(){
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">return&lt
/span> symbol;
}

<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">private</
pan> Map<<span class="hljs-keyword" style="box-sizing: border-box; color: #33
333; font-weight: bold;">String,Operator> <span class="hljs-built_
n" style="box-sizing: border-box; color: #0086b3;">map = <span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">new</sp
n> <span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-wei
ht: bold;">HashMap<>();
<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">static</s
an>{
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">for</s
an>(Opeator op : values){
        <span clas

="hljs-built_in" style="box-sizing: border-box; color: #0086b3;">map.put(op.
oString(),op);
    }
}

<span class="hl
s-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">public</
pan> <span class="hljs-keyword" style="box-sizing: border-box; color: #333333; font-we
ght: bold;">static fromString(<span class="hljs-keyword" style="box-sizing
 border-box; color: #333333; font-weight: bold;">String name){
    <span class=
hljs-keyword" style="box-sizing: border-box; color: #333333; font-weight: bold;">return&lt
/span>
ap.<span class="hljs-built_in" style="box-sizing: border-box; color: #0086b3;
>get(name);
}
</code></pre>
</code> <code class="hljs">}</code> </pre>
<h3 id="toc_h3_36">第31条 用实力域代替序数</h3>
不使用ordinal，其会改变，实例域不会。
<h3 id="toc_h3_37">第32条 使用EnumSet代替位域</h3>
EnumSet.of(Sytle.BOLD,Style.ITALIC);
<h3 id="toc_h3_38">第33条 使用EnumMap代替序数索引</h3>
不使用ordinal，因其会改变，使用EnumMap代替。 反正不要使用ordinal就对了
<h3 id="toc_h3_39">第34条 用接口模拟可伸缩的枚举</h3>
<pre><code class="hljs">public interface Operation{
    double apply(double x,double y);
}
</code> <code class="hljs">public <span class="hljs-keyword
>enum BasicOpeation imp
ements Operation{
...
}</code> </pre>
缺点，骨架代码要写两遍
<h3 id="toc_h3_40">第35条 注解优于命名模式</h3>

注解是显式的，比较容易发现错误，命名模式里命名错了难以发现
注解可以给某个元素附加一些特殊的信息，而命名模式难以做到，做到也不雅

<h3 id="toc_h3_41">第36条 坚持使用Override注解</h3>
略
<h3 id="toc_h3_42">第37条 用标记接口定义类型</h3>
要定义一个任何新方法都不会与之关联的类型，标记接口才是最好的选择，而非 标志注解

标志接口定义的类型是由被标记类的实例实现的，有语言编译的检查，而标记注解则无，只能运
时检查

标志注解的优点

可以一直给类加注解，增加信息，扩展性好，而接口则无法变更

<h2 id="toc_h2_43">第7章 方法</h2>
<h3 id="toc_h3_44">第38条 检查参数的有效性</h3>
尽早检查入参，能及时判断错误的来源

<h3 id="toc_h3_45">第39条 进行保护性复制</h3>
当调用者不可信或者防止意外时，有时需要对传入的参数进行复制并保存到复制的对象。传出对时，也可以视情况返回一个复制的对象，以免外部通过传出的对象修改影响对象内部状态。 当然保护复制是有消耗的，有必要才使用
<h3 id="toc_h3_46">第40条谨慎涉及方法签名</h3>

方法名称：与包风格一致，使用大众认可的名称
不要过于追求提供便利的方法，除非提供的方法特常用
避免过长的参数列表，目标是4个，或者更少

分解大方法，提供小方法
创建辅助类保存参数分组
Builder模式

参数优先使用接口而非类
boolean参数尽量使用枚举类型代替

<h3 id="toc_h3_47">第41条 慎用重载</h3>
重载决定使用哪个方法是编译时决定的，如下
<pre><code class="hljs">public static ex(Integer i){
    ...
}
public static ex(bject i){
...
}
</code> <code class="hljs">public static void main(String[] args){
Integer i = 1;
ex((Object)i);//调用的是第二个方法，而非第一个
}</code> </pre>
这个表现跟Override不一样，Override决定调用哪个方法是在运行时，根据运行类型决定的。
由于上述原因，我们使用重载的时候要特别的小心，尤其是碰上 自动拆包装包，泛型 等情况的候。
要提供类似的方法的时候，优先使用不同名称的方法，而不是重载。
若在构造器等场景，只能使用一个方法名，我们可以尝试使用静态工厂方法来替换掉重载的构造
若一定要使用重载，也尽可能的避免使用相同参数个数的重载方法。
<h3 id="toc_h3_48">第42条 慎用可变参数</h3>
没看懂为什么，不管
<h3 id="toc_h3_49">第43条 返回令长度的数组或集合 而不是NULL</h3>
常识
<h3 id="toc_h3_50">第44条 为所有到处API元素编写文档注释</h3>
略
<h2 id="toc_h2_51">第8章 通用程序设计</h2>
<h3 id="toc_h3_52">第45条 将局部变量的作用于最小化</h3>
常识，最基本的方法就是在第一次使用的地方进行声明

<h3 id="toc_h3_53">第46条 for-each循环优先于传统的FOR循环</h3>
略，已习惯使用for-each循环
<h3 id="toc_h3_54">第47条 了解和使用类库</h3>
优先使用类库，不重复制作轮子OVER 至少要知道java.lang,java.util包里有什么内容，java.io也了解下
<h3 id="toc_h3_55">第48条 如果需要精确的答案，请避免使用float和double</h3>
使用 long,int或者BigDecimal代替 虽然double的精度很高，但使用double进行计算，舍入并总能得到正确的结果。 如：
<pre><code class="hljs">public</pan> static voidspan class="hljs-title">main(String[] args)</spa>{
    double funds = 1.0</spn>;
    int itemBought = 0</spn>;
    for(double pric = 0.1;funds >= price; price+= 0.1){
        funds -= price;
        itemBought++;
    }
}</code></pre>
运行的最后，会剩下 0.39999999999元,itemBought为3，与预期用完所有钱，买到4颗糖果的期不符
<h3 id="toc_h3_56">第49条 基本类型优先于装箱基本类型</h3>

性能
装箱类型==操作有坑
装箱类型自动拆箱可能出现nullPointerExceptiron

结论，平时用基本类型，不能用基本类型才用 装箱类型。
<h3 id="toc_h3_57">第50条 如果其他类型更合适，避免使用字符串</h3>
感觉差不多，略
<h3 id="toc_h3_58">第51条 当心字符串连接的性能</h3>
略
<h3 id="toc_h3_59">第52条 通过接口引用对象</h3>
已形成习惯：可替换，减少暴露，更易于修改
<h3 id="toc_h3_60">第53条 接口优先于反射机制</h3>
反射一般会比原生方法慢 2-10倍。因此尽量避免使用反射。除非要管理一些在编译时不存在的型。 而且也尽量只使用反射来创建实例，获取元信息。操作方法的时候使用接口。
<h3 id="toc_h3_61">第54条 谨慎地使用本地方法</h3>
以下三种情形要使用本地代码

访问特定于运行平台的机制的能力
访问遗留代码库的能力
提升性能

提升性能对现在的JVM来说意义不大。
不使用的原因， 类型不安全，影响移植，胶合代码难看
<h3 id="toc_h3_62">第55条 谨慎地进行优化</h3>
不要因为性能牺牲合理的结构。要努力编写好的程序而不是快的程序。 努力避免那些限制性能设计决策，好的API设计会带来好的性能，如果开发完速度不够，再进行优化
可以用性能剖析器测试性能瓶颈

<h3 id="toc_h3_63">第56条 遵守普遍接收的命名惯例</h3>
命名惯例参考 The Java Language Specification
<h2 id="toc_h2_64">第9章 异常</h2>
<h3 id="toc_h3_65">第57条 只针对异常的情况才使用异常</h3>

基于异常的写法难以让人理解
基于异常捕获的形式的代码会比正常的写法慢

总之，正常的流程控制不应该包含异常
<h3 id="toc_h3_66">第58条 对可恢复的情况使用受检异常，对编程错误使用运行时异常</h3>

如果期望调用者能够适当的恢复，那么使用受检的异常
运行时异常用来表明编程错误，大多数运行时异常都表示 前提违例
ERROR基本都是保留给JVM使用的，因此自己不应该抛出ERROR

<h3 id="toc_h3_67">第59条 避免不必要的使用受检的异常</h3>
我对受检的异常非常痛恨...会大大加大工作量，使用受检异常应该只有在以下两个条件都存在时才使用

正确使用API依然可能出现异常
出现了异常后，调用者能恢复处理这个异常

<h3 id="toc_h3_68">第60条 优先使用标准的异常</h3>

与大家习惯用法一致
便于学习
可以继承某些异常，并增加一些异常信息

<h3 id="toc_h3_69">第61条 抛出与抽象相对应的异常</h3>
更高层的实现应该捕获低层的异常，同时抛出可以按照高层抽象进行解析的异常。 这个叫异常译。异常转译通常会包括异常链，方便排查问题
<h3 id="toc_h3_70">第62条 每个方法抛出的异常都要有文档</h3>
....
<h3 id="toc_h3_71">第63条 在细节消息中包含能捕获失败的信息</h3>
一直这么做
<h3 id="toc_h3_72">第64条 努力使失败保持原子性</h3>
但是基于数据库的一致性倒是容易的。
<h3 id="toc_h3_73">第65 不要忽略异常</h3>
略
<h2 id="toc_h2_74">第10章 并发</h2>
<h3 id="toc_h3_75">第66条 同步访问共享的可变数据</h3>
常识，没有正确同步的数据，JVM并不保证何时可见，可能由于某种优化，某数据就一直看不见。 最好的办法是 不共享的办法，或者 共享不可变变量的办法
<h3 id="toc_h3_76">第67条 避免过度同步</h3>

同步块内不要调用外部方法
同步块内尽量做少的工作
同步会令CPU上市并行的机会，JVM丧失编译优化的机会
一个可变的类要并发使用，应该使得这个类是线程安全的，同步内部同步会比外部锁定整个对象步性能更好。
但同步并不是必须的时候，如经常就一个线程使用，偶尔才多线程，那么，就应该由外部进行同
在多核时代，设计一个可变类的时候要思考这个类是否应该自己实现同步，这比避免过度同步更要

<h4 id="toc_h4_77">第68条 executor和task优先于Thread</h4>
略,常识
<h3 id="toc_h3_78">第69条 并发工具优先于wait和notify</h3>

ConcurrentHashMap(并发集合相关)
CoundownLatch（同步类相关）
若要使用wait那么要加上循环条件判断
如果要用notify最好用notifyALL,牺牲一点性能避免错误发生

<h3 id="toc_h3_79">第70条 线程安全性的文档化</h3>
线程安全的可能的几个级别

不可变的（字符串之类）
无条件线程安全（内部已实现同步）
有条件的线程安全（部分方法需要外部进行同步）
非线程安全（需要外部同步）
线程对立（不能多线程执行，即使外部执行了同步。通常是由于异步修改了静态数据导致）

线程安全的描述需要写在文档注释中，否则使用者将无法正确使用
<h3 id="toc_h3_80">第71条 慎用延时初始化</h3>
因为有坑，且在实例域延迟初始化时，会降低效率
若一定要做这个，那么有以下方法
<h4 id="toc_h4_81">静态域的延迟初始化使用匿名内部类模式</h4>
<pre><code class="hljs">private static class LazyInit{
    static final LazyField = new LazyField();
}
</code> <code class="hljs">public static LazyField getLazyField(){
return LazyInit.lazyField;
}</code> </pre>
<h4 id="toc_h4_82">实力域的延迟初始化方式——双重检查</h4>
<pre><code class="hljs">private volatile LazyField lazyField;//使用volatile的原因代码后的解析
</code> <code class="hljs">public LazyField getLazyField(){
LazyField lazyField = this.lazyField;//避免每次使用都访问 volatile 域，提高性能
if(lazyField == null){//第一次检查，不带锁，避免每次检查都加锁，提升效率
synchronized(this){
if(this.lazyField = null){//第二重检查，带锁保证不重复初始化的关键
this.lazyField = lazyField =new LazyField();

}
}
}
return lazyField;
}</code> </pre>
lazyField声明为volatile是为了禁止lazyField = new LazyField()方法的重排序。若不声明为volatile，这个赋值操作可能就变成了以下伪代码描述的场景
<pre><code class="hljs">memory = allocat();
lazyField = memory;
initInstance(memory);</code></pre>
这种场景的话，lazyField可能还没被初始化，但lazyField已经不为NULL了
<h4 id="toc_h4_83">当实例域可以重复初始化时，可以编程单重检查</h4>
<pre><code class="hljs">private volatile LazyField lazyField;
</code> <code class="hljs">public LazyField getLazyField(){
LazyField lazyField = this.lazyField;//避免每次使用都访问 volatile 域，提高性能
if(lazyField == null){
synchronized(this){
this.lazyField = lazyField =new LazyField();
}
}
return lazyField;
}</code> </pre>
<h4 id="toc_h4_84">volatile总是不能去掉的</h4>
否则会出现难以预测的场景。
<h3 id="toc_h3_85">第72条 不要依赖于线程调度器</h3>
因为这个是与平台相关的，依赖于这个东西，将无法重现对应的结果
<h3 id="toc_h3_86">第73条 避免使用线程组</h3>
没用过。说过时了。那就不用了
<h2 id="toc_h2_87">第11章 序列化</h2>
<h3 id="toc_h3_88">第74条 谨慎地实现Serializable接口</h3>

一旦一个类被发布，就大大降低了“改变这个类的实现”的灵活性，因为类中的内部实例域都会公布出去
增加了出现BUG和安全漏洞的可能性。反序列化机制是一个隐藏的构造器，使用这个构造器的时很容易就会忘记保证原有对象中应有的约束
增加了测试的负担，需要测试二进制的兼容性

表示实体活动的类，一般不应该实现Serializable 为了继承而设计的类，应尽可能少实现Serializable接口
<h3 id="toc_h3_89">第75条 考虑使用自定义的序列化形式</h3>
当对象的逻辑含义与物理表示接近的时候，可以考虑使用默认的序列化形式。
但逻辑含义与物理表示差别很大的时候，就不适合了。
如一个用链表表示的字符串对象。默认序列化会把链表的引用指向关系如实的保存起来，消耗比大。导出的对象实现形式会被束缚在这具体的实现形式上无法变更，除非放弃之前序列化的对象的二制数据。
<h3 id="toc_h3_90">第76条 保护性的编写readObject方法</h3>
避免有人恶意修改被范序列化的对象，破坏约束 常用做法是，readObject的时候检查约束，对

一些可变对象进行保护性复制
<h3 id="toc_h3_91">第77条 对于实例控制，美剧类型优先于readResolve</h3>
略
<h3 id="toc_h3_92">第78条 考虑用序列化代理替代序列化实例</h3>
略