**软件开发中心开发规范**

【Java编程规范】



软件开发中心

2014年5月

交通银行保密资料 请勿外泄

文档修改历史

| 版本号 | 修改日期 | 编写 | 评审 | 批准 | 修改内容 |
| --- | --- | --- | --- | --- | --- |
| 0.1 | 2012-10-30 | 秦小波 |  |  | 新建 |
| 0.3 | 2012-11-20 | 王之韡 |  |  | 初稿 |
| 0.5 | 2012-12-07 | 王之韡 |  |  | 讨论修改稿一 |
| 0.6 | 2012-12-12 | 王之韡 |  |  | 添加第8、9、10、11章，完善部分例子。 |
| 0.7 | 2012-12-18 | 王之韡 |  |  | 根据讨论完善部分内容。添加12通信规范。调整错误的格式。 |
| 0.8 | 2013-01-07 | 王之韡 |  |  | 新增规范检查标准100条。调整目录级别。 |
| 0.9 | 2013-01-10 | 秦小波 |  |  | 修订 |
| 1.0 | 2013-03-05 | 王之韡 |  |  | 根据各部门意见进行修订 |
| 1.5 | 2014-05-05 | 汪涛 |  | 李海翔 | 根据各部门意见进行修订 |

**目录**

[**1** **编写目的 6**](#_Toc389228373)

[**2** **读者对象 6**](#_Toc389228374)

[**3** **编码优化规范 6**](#_Toc389228375)

[3.1 创建和销毁对象 6](#_Toc389228376)

[*3.1.1* *考虑用静态工厂方法替代构造方法 6*](#_Toc389228377)

[*3.1.2* *使用私有构造方法强化singleton属性 7*](#_Toc389228378)

[*3.1.3* *用私有构造方法强化不可实例化能力 8*](#_Toc389228379)

[*3.1.4* *避免创建重复对象 8*](#_Toc389228380)

[*3.1.5* *消除对过期对象的引用 8*](#_Toc389228381)

[3.2 对象定义 8](#_Toc389228382)

[*3.2.1* *重载equals 8*](#_Toc389228383)

[*3.2.2* *重载equals时永远重载hashCode 9*](#_Toc389228384)

[*3.2.3* *推荐对自定义对象重载toString() 9*](#_Toc389228385)

[*3.2.4* *谨慎重载clone 9*](#_Toc389228386)

[3.3 接口、抽象类、类 10](#_Toc389228387)

[*3.3.1* *最小化类和成员的可访问能力 10*](#_Toc389228388)

[*3.3.2* *组合优于继承 10*](#_Toc389228389)

[*3.3.3* *接口优于抽象类 11*](#_Toc389228390)

[*3.3.4* *尽可能使用内部类 12*](#_Toc389228391)

[3.4 通用编程 12](#_Toc389228392)

[*3.4.1* *最小化局部变量的作用域 12*](#_Toc389228393)

[*3.4.2* *理解和使用库 12*](#_Toc389228394)

[*3.4.3* *精确数据计算时，不要使用float或是double类型 12*](#_Toc389228395)

[*3.4.4* *了解字符串“+”的性能 12*](#_Toc389228396)

[*3.4.5* *接口优于反射 13*](#_Toc389228397)

[3.5 容器与泛型 14](#_Toc389228398)

[*3.5.1* *遍历容器对象 14*](#_Toc389228399)

[*3.5.2* *尽可能提供参数类型的容器 15*](#_Toc389228400)

[*3.5.3* *列表优先于数组 15*](#_Toc389228401)

[*3.5.4* *变长参数 15*](#_Toc389228402)

[3.6 线程 17](#_Toc389228403)

[*3.6.1* *线程的状态 17*](#_Toc389228404)

[*3.6.2* *同步Synchronized 18*](#_Toc389228405)

[*3.6.3* *考虑死锁 19*](#_Toc389228406)

[3.7 异常 20](#_Toc389228407)

[*3.7.1* *受检异常、运行时异常、错误 20*](#_Toc389228408)

[*3.7.2* *不要忽略异常 20*](#_Toc389228409)

[*3.7.3* *少用异常，尽可能减少自定义异常，优先使用标准异常 21*](#_Toc389228410)

[*3.7.4* *为方法抛出的异常撰写说明文档 22*](#_Toc389228411)

[*3.7.5* *记录所捕获到异常的有用信息 22*](#_Toc389228412)

[*3.7.6* *努力使失败保持原子性 23*](#_Toc389228413)

[*3.7.7* *不要一次捕获所有异常 24*](#_Toc389228414)

[*3.7.8* *针对不同情景使用不同异常 24*](#_Toc389228415)

[*3.7.9* *在finally块中释放资源 24*](#_Toc389228416)

[3.8 通信 25](#_Toc389228417)

[*3.8.1* *HTTP 25*](#_Toc389228418)

[*3.8.2* *JMS 25*](#_Toc389228419)

[*3.8.3* *Socket 26*](#_Toc389228420)

[3.9 错误定位与日志 27](#_Toc389228421)

[*3.9.1* *日志框架 27*](#_Toc389228422)

[*3.9.2* *日志记录 27*](#_Toc389228423)

[*3.9.3* *日志级别 29*](#_Toc389228424)

[*3.9.4* *日志规范 30*](#_Toc389228425)

[*3.9.5* *配置示例 30*](#_Toc389228426)

[3.10 避免内存泄漏 32](#_Toc389228427)

[*3.10.1* *尽早释放无用对象的引用。 32*](#_Toc389228428)

[*3.10.2* *少用finalize函数 32*](#_Toc389228429)

[*3.10.3* *注意集合数据类型 33*](#_Toc389228430)

[*3.10.4* *避免显式申请数组空间 33*](#_Toc389228431)

[**4** **编码格式规范 33**](#_Toc389228432)

[4.1 文件组织 33](#_Toc389228433)

[*4.1.1* *文件名 33*](#_Toc389228434)

[*4.1.2* *文件后缀 33*](#_Toc389228435)

[*4.1.3* *Java源文件 33*](#_Toc389228436)

[*4.1.4* *类和接口声明 34*](#_Toc389228437)

[4.2 缩进排版 35](#_Toc389228438)

[*4.2.1* *行长度 35*](#_Toc389228439)

[*4.2.2* *换行 35*](#_Toc389228440)

[4.3 注释 36](#_Toc389228441)

[*4.3.1* *实现注释的格式 37*](#_Toc389228442)

[*4.3.2* *块注释 37*](#_Toc389228443)

[*4.3.3* *文档注释 38*](#_Toc389228444)

[4.4 声明 39](#_Toc389228445)

[*4.4.1* *每行声明变量的数量 39*](#_Toc389228446)

[*4.4.2* *初始化 39*](#_Toc389228447)

[*4.4.3* *布局 39*](#_Toc389228448)

[*4.4.4* *类和接口的声明 39*](#_Toc389228449)

[4.5 语句 40](#_Toc389228450)

[*4.5.1* *简单语句 40*](#_Toc389228451)

[*4.5.2* *复合语句 40*](#_Toc389228452)

[*4.5.3* *返回语句 40*](#_Toc389228453)

[*4.5.4* *if，if-else，if else-if else语句 40*](#_Toc389228454)

[*4.5.5* *for语句 41*](#_Toc389228455)

[*4.5.6* *while语句 41*](#_Toc389228456)

[*4.5.7* *do-while语句 41*](#_Toc389228457)

[*4.5.8* *switch语句 42*](#_Toc389228458)

[*4.5.9* *try-catch语句 42*](#_Toc389228459)

[4.6 空白 42](#_Toc389228460)

[*4.6.1* *空行 42*](#_Toc389228461)

[*4.6.2* *空格 43*](#_Toc389228462)

[*4.6.3* *命名规范 43*](#_Toc389228463)

[4.7 编程惯例 45](#_Toc389228464)

[*4.7.1* *提供对实例以及类变量的访问控制 45*](#_Toc389228465)

[*4.7.2* *引用类变量和类方法 45*](#_Toc389228466)

[*4.7.3* *常量 45*](#_Toc389228467)

[*4.7.4* *变量赋值 45*](#_Toc389228468)

[*4.7.5* *其它惯例 46*](#_Toc389228469)

[4.8 代码范例 46](#_Toc389228470)

[**5** **规范检查标准100条 48**](#_Toc389228471)

1. **编写目的**

**A good language should help programmers program well ， but no languge could prevent programmers from bad practice.**

一门优秀的语言可以帮助程序员更好的编码，但是并不能避免糟糕的用法。

——《Thinking In Java》

学习一门语言，不仅仅是掌握其语法和用法，更要领悟其设计思想，避开语言设计的不足和陷阱，使用语言的优良特性编写可靠、可维护的系统。

本文档希望通过描述基于JUMP平台开发的各应用系统需要遵守的各项编码规范，达成如下的目标：

* 编程风格一致；
* 程序易于维护；
* 程序易于阅读和理解；
* 程序代码简洁、执行高效；
* 程序中减少可能出现的运行时错误；
* 程序容易定位错误。

1. **读者对象**

本文档的读者期望为采用JUMP平台或基于J2EE规矩进行应用项目开发的各开发部门的开发人员、测试人员、质量管理人员，及各合作公司开发的各相关人员。

1. **编码优化规范**
   1. 创建和销毁对象
      1. **考虑用静态工厂方法替代构造方法**

如：

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

采用静态工厂方法代替构造方法的优势在于：

* 静态方法可以有更清晰明确的名字；
* 不需要在每次调用时都去创建一个新的对象；
* 可以返回任何子类型的对象。
  + 1. **使用私有构造方法强化singleton属性**

Java Singleton模式主要作用是保证在Java应用程序中，一个类Class只有一个实例存在。这种类型典型地被用来表示那些本性上具有唯一特性的系统组件，比如：JNDI的name查找器。在很多操作中，比如建立目录、数据库连接都需要这样的单线程操作。

还有, singleton能够被状态化；这样，多个单态类在一起就可以作为一个状态仓库一样向外提供服务，比如，你要论坛中的帖子计数器，每次浏览一次需要计数，单态类能否保持住这个计数，并且能synchronize的安全自动加1，如果你要把这个数字永久保存到数据库，你可以在不修改单态接口的情况下方便的做到。

另外方面，Singleton也能够被无状态化。提供工具性质的功能，使用Singleton的好处还在于可以节省内存，因为它限制了实例的个数，有利于Java垃圾回收。

实现Singleton模式有以下四点需要注意：

* **提供一个私有构造方法**

|  |
| --- |
| **public** **class** SingletonObjectDemo {  // Note that the constructor is private  **private** SingletonObjectDemo() {  // Optional Code  }  } |

* **添加一个方法用于得到单例对象：**

|  |
| --- |
| **public** **class** SingletonObjectDemo {  **private** **static** SingletonObject *singletonObject*;  // Note that the constructor is private  **private** SingletonObjectDemo() {  // Optional Code  }  **public** **static** SingletonObjectDemo getSingletonObject() {  **if** (*singletonObject* == **null**) {  *singletonObject* = **new** SingletonObjectDemo();  }  **return** *singletonObject*;  }  } |

这里提供了一个公共静态get方法用于在运行时得到单例对象。首次调用该方法时，对象从null变为实例，此后的调用都返回这个单例对象。

* **保证get方法的线程安全**

|  |
| --- |
| **public** **static** **synchronized** SingletonObjectDemo getSingletonObject() |

仍然有可能在两个线程同事调用该方法时创建出2个或以上的对象，而这违反了单例原则。为了避免同时有2个线程调用get方法，可以在该方法上添加synchronized关键字来保证线程安全。

* **重载clone方法避免对象clone**

使用clone方法仍然有可能得到单例对象的一份拷贝，比如：

|  |
| --- |
| SingletonObjectDemo clonedObject = (SingletonObjectDemo) obj.clone(); |

为了避免单例对象被clone，可以重载clone方法，并抛出CloneNotSupportedException异常

|  |
| --- |
| **public** Object clone() **throws** CloneNotSupportedException {  **throw** **new** CloneNotSupportedException(); } |

* + 1. **用私有构造方法强化不可实例化能力**

有些工具类不希望被实例化：对它进行实例化没有意义。如：java.lang.Math。然而在类中缺少显式构造方法的时候，编译器会自动地提供一个公共的无参数的缺省构造方法。在类中包含显式的私有类型构造方法来实现类的不可实例化特性。因为构造方法是私有的，所以它在类的外部不可访问。如果构造方法不会被类自身从内部调用，即能保证类永远不会被实例化。

* + 1. **避免创建重复对象**

如：

|  |
| --- |
| String s = **new** String("abc"); |

这种写法创造了两个对象："abc"和 s(="abc")，前者仅用于给s赋值，随后被垃圾回收器回收，事实上，"abc"对象的创建是完全没有必要的，给程序带来了不必要的开销。

以下的写法可以接受：

|  |
| --- |
| Strong s = "abc"; |

* + 1. **消除对过期对象的引用**

一旦一个对象的引用被不小心地保留，不仅这个对象被排除在垃圾回收之外。而且所有被这个对象引用的对象也不会被垃圾收集，并顺延下去，从而有可能造成内存泄漏问题。

一旦对象过期，必须清除资源以及对它们的引用。尤其自己的程序管理某些对象时，尤其如此。如类中使用对象数组，HashMap 等等。

* 1. 对象定义
     1. **重载equals**

满足下面的条件就不要重载equals：

* 每个类实例本质上是唯一的；
* 不关心类是否提供了“逻辑意义的等同”测试；
* 超类已经重载了equals，而超类继承的行为适合该类；
* 类是私有的或是或是包内私有的，而且可以确定它的equals方法永远不会被调用。

当类有逻辑上的等同意义而不仅仅是对象意义上的等同，而且超类没有重载equals方法以实现期望的行为，这时需要重载。这种情形通常是数值类型的类（Value Object）。这时，equals的重载不仅对满足程序员的需要是必须的，它也能使类实例通过可预知的，期望的行为来做map关键字或set元素，以及Collection中的访问。

* + 1. **重载equals时永远重载hashCode**

equals()反映的是对象或变量具体的值。而hashCode()是对象或变量通过哈希算法计算出的哈希值。之所以有hashCode方法，是因为在批量的对象比较中，hashCode要比equals来得快，很多集合都用到了hashCode，比如HashTable。

* 两个obj，如果equals()相等，hashCode()一定相等；
* 两个obj，如果hashCode()相等，equals()不一定相等（Hash散列值有冲突的情况，虽然概率很低）。

所以，可以考虑在集合中，判断两个对象是否相等的规则是：

第一步，如果hashCode()相等，则查看第二步，否则不相等;

第二步，查看equals()是否相等，如果相等，则两obj相等，否则还是不相等。

* + 1. **推荐对自定义对象重载toString()**

推荐对所有的自定义类重载这个方法，当对象传递给println方法的连接操作(+)时，toString()方法会自动调用。从而有利于调试、有利于日志记录。

* + 1. **谨慎重载clone**

需要重载clone时必须实现以下3条：

* 声明实现Cloneable接口。
* 调用super.clone拿到一个对象，如果父类的clone实现没有问题，则在该对象的内存存储中，所有父类定义的field都已经clone，该类中的primitive和不可变类型引用也已经克隆，可变类型引用都是浅copy。
* 把浅copy的引用指向原型对象新的克隆体。

例：

|  |
| --- |
| **class** User **implements** Cloneable {  String name;  **int** age;  @Override  **public** User clone() **throws** CloneNotSupportedException {  **return** (User) **super**.clone();  }  }  **class** Account **implements** Cloneable {  User user;  **long** balance;  @Override  **public** Account clone() **throws** CloneNotSupportedException {  Account account = **null**;  account = (Account) **super**.clone();  **if** (user != **null**) {  account.user = user.clone();  }  **return** account;  }  } |

* 1. 接口、抽象类、类
     1. **最小化类和成员的可访问能力**

隐藏模块的内部数据和实现细节，仅仅把该暴露的部分暴露给调用者。这也是OO三概念之一：封装。

如果把某些不应该暴露的方法暴露给调用者，一旦某个调用者使用了它，那么为了保证程序兼容，就得继续提供对该方法的支持。

* 私有（private） 仅在类内部可访问；
* 缺省的（default） 对包内的所有类可访问（注意：没有default的关键字，当没有指明任何修饰符时，就是这种类型）；
* 受保护的（protected） 该类的子类和包内所有类可访问；
* 公共的（public） 访问不受限制。

子类重载父类的方法，子类中方法的访问能力不能低于父类中该方法的访问能力；否则编译器产生编译错误。

* + 1. **组合优于继承**

当确实需要继承时才采用继承，否则使用类组合来完成。组合，通过创建一个由其他对象组合的对象来获得新功能的重用方法，新功能的获得是通过调用组合对象的功能实现的，有时又叫聚合。例如：

* 一个对象拥有或者对另外一个对象负责并且两个对象有相同的生命周期；
* 一个对象包含另一个对象集合；
* 被包含对象对其他对象是不可见的并且只能从包含它的对象中访问的特殊组合形式。
  + - 1. **组合**

优点

* 被包含对象通过包含他们的类来访问
* 黑盒重用，因为被包含对象的内部细节是不可见的
* 很好的封装
* 每个类专注于一个任务
* 通过获得和被包含对象的类型相同的对象引用，可以在运行时动态定义组合的方式

缺点

* 结果系统可能会包含更多的对象
* 为了使组合时可以使用不同的对象，必须小心的定义接口
  + - 1. **继承**
* 通过扩展已实现的对象来获得新功能的重用方法
* 基类有用通用的属性和方法
* 子类提供更多的属性和方法来扩展基类

优点

* 新的实现很容易，因为大部分是继承而来的
* 很容易修改和扩展已有的实现

缺点

* 打破了封装，因为基类向子类暴露了实现细节
* 白盒重用，因为基类的内部细节通常对子类是可见的
* 当父类的实现改变时可能要相应的对子类做出改变
* 不能在运行时改变由父类继承来的实现

由此，组合比继承具有更大的灵活性和更稳定的结构。一般情况下，应该优先考虑组合。只有当下列条件满足时才考虑使用继承：

* 子类是一种特殊的类型，而不只是父类的一个角色
* 子类的实例不需要变成另一个类的对象
* 子类扩展，而不是覆盖或者使父类的功能失效
  + 1. **接口优于抽象类**

两种机制最明显的区别是抽象类容许包含某些方法的实现，而接口不行。如：com.bocom.jump.bp.channel.AbstractEndpointMapping，就是抽象类，它具有得到渠道映射的方法：

|  |
| --- |
| **protected** **abstract** Endpoint<Request， Response> doMapping (ChannelContext<Request， Response> ctx); |

但是怎么实现具体映射操作必须是其子类实现。如果设计成抽象类，一旦子类要求继承其他类时就没有办法实现，因为Java中不能多重继承（extends），但是可以多重实现（implements）。换言之，接口是定义混合类型的更理想选择。

* 接口允许非层次类型框架的构造。
* 接口通过封装类方式，能够获得安全、强大的功能。

抽象类有另一个优势：改进抽象类比改进接口更加容易。一旦某个方法的实现发生了变化，仅仅需要更改一个地方；当新加一个方法时，所有子类都具有了该方法。

* + 1. **尽可能使用内部类**

如果该类仅仅在某个调用者使用，那么把它设计成调用者的内部类。不会暴露该内部类，避免滥用而导致的维护问题。

* 1. 通用编程
     1. **最小化局部变量的作用域**
* 最小化局部变量最有效的方法是在它第一次使用时声明。
* 几乎每个局部变量声明都应该包含一个初始值，比如null，否则编译也通不过。
* 方法小而集中，每个方法尽可能仅仅处理一种活动。
  + 1. **理解和使用库**

Java提供了强大而复杂的类库，在完成某个常用功能之前，看看Java类库中是否已经存在类似功能的类库。一方面减少工作量，一方面使得自己的程序更加简洁高效。

例如：向量（Collection，Set，List…）中字符串的排序：Collections.sort(v)；

忽略大小写的排序：Collections.sort(v，String.CASE\_INSENSITIVE\_ORDER)

更多的实用类库可参考JavaDoc。

* + 1. **精确数据计算时，不要使用float或是double类型**

使用int或是long进行精确数据计算，自己管理小数位。

当数目不超过9位时，使用int类型；当数目不超过18位时，使用long；如果超过了18位，使用BigDecimal（注意它是对象，不是基本数据类型）。

* + 1. **了解字符串“+”的性能**

在java中有3个类来负责字符的操作。

* Character 是进行单个字符操作的；
* String 对一串字符进行操作。不可变类；
* StringBuffer 也是对一串字符进行操作，但是可变类。

**String:**

* 是对象不是原始类型；
* 为不可变对象，一旦被创建，就不能修改它的值；
* 对于已经存在的String对象的修改都是重新创建一个新的对象，然后把新的值保存进去；
* String 是final类，即不能被继承。

**StringBuffer:**

是一个可变对象，当对他进行修改的时候不会像String那样重新建立对象，它只能通过构造函数来建立：

StringBuffer sb = **new** StringBuffer();

不能通过付值符号对他进行付值. 如 sb = "welcome to here!"；

对象被建立以后，在内存中就会分配内存空间，并初始保存一个null.向StringBuffer中付值的时候可以通过它的append方法，如：

|  |
| --- |
| sb.append("hello"); |

字符串连接操作中StringBuffer的效率要比String高很多，如：

|  |
| --- |
| String str = **new** String("welcome to ");  str += "here"; |

其处理步骤实际上是通过建立一个StringBuffer，然后调用append()，并且由于String 对象是不可变对象，每次操作Sting 都会重新建立新的对象来保存新的值，而连接之后该对象就丧失作用而被垃圾回收，从而影响到性能。再将StringBuffer toSting(); 因此String的连接操作就比StringBuffer多出了一些对象申请转换的开销，性能相差极大。

* + 1. **接口优于反射**
* 反射失去了在编译期间进行类型检查提供的好处；
* 使用反射使得代码笨拙而冗长，可读性差；
* 使用反射性能损失。反射比正常的方法调用慢40倍左右；

有的时候，付出代价是值得的，因为能够得到更多的益处；一般而言，可以通过反射创建一个实例，然后通过他们的接口或是超类正常访问他们。

* 1. 容器与泛型
     1. **遍历容器对象**
        1. **List的遍历**

**不推荐**用以下方式遍历List：

|  |
| --- |
| List list = **new** ArrayList();  **for** (**int** i = 0; i < list.size(); i++) {  Object obj = list.get(i);  // obj do something;  } |

**推荐**用以下两种方式遍历List:

|  |
| --- |
| **for** (Object obj : list) {  // obj do something;  } |

或者：

|  |
| --- |
| Iterator itr = list.iterator();  **while** (itr.hasNext()) {  Object obj = itr.next();  // obj do something;  } |

* + - 1. **Map的遍历**

仅需要遍历值的情况：

|  |
| --- |
| Collection<Student> c = map.values();  Iterator it = c.iterator();  **while** (it.hasNext()) {  Student s = it.next();  // do something about s;  } |

利用keyset进行遍历

|  |
| --- |
| Set<String> keys = map.keySet();  **for** (Iterator it = keys.iterator(); it.hasNext();) {  String key = (String) it.next();  Student student = map.get(key);  // do something about student  } |

利用Map.Entry进行遍历

|  |
| --- |
| Set<Map.Entry<String， Student>> set = map.entrySet();  **for** (Iterator<Map.Entry<String， Student>> it = set.iterator(); it.hasNext();) {  Map.Entry<String， Student> entry = (Map.Entry<String， Student>) it.next();  Student student = entry.getValue());  // do something about student  } |

* + 1. **尽可能提供参数类型的容器**

推荐使用Set<Object> 或者Set<?>，不推荐以下的写法：

|  |
| --- |
| Set set = **new** Set(); |

这种写法容易造成ClassCastException，同时不利于代码的可读性。

* + 1. **列表优先于数组**

数组相比列表（List）有两个方面的问题：

* 数组的协变性(covariant)是指如果类Base是类Sub的基类，那么Base[]就是Sub[]的基类。而泛型是不可变的(invariant)，List<Base>不会是List<Sub>的基类，更不会是它的子类。
* 数组是具体化的(reified)，而泛型在运行时是被擦除的(erasure)。这句话的意思是数组是在运行时才去判断数组元素的类型约束，而泛型正好相反，在运行时，泛型的类型信息是会被擦除的，只有编译的时候才会对类型进行强化。

比如下面的代码：

|  |
| --- |
| Object[] array = **new** String[10];  array[0] = 10; |

它是可以编译通过的，因为数组是协变的，Object[]类型的引用可以指向一个String[]类型的对象。但是运行的时候是会报出如下异常的：java.lang.ArrayStoreException: java.lang.Integer。但是对于泛型就不会出现这种情况了：

|  |
| --- |
| List<Object> list = **new** ArrayList<String>(); |

这段代码连编译都不能通过。

* + 1. **变长参数**

从Java5开始提供了变长参数，也就是在方法定义中可以使用个数不确定的参数，对于同一方法可以使用不同个数的参数调用，例如：

|  |
| --- |
| print("hello");  print("hello"，"lisi");  print("hello"，"张三"); |

在定义可变长参数以及使用可变长参数时需遵照以下规范：

* **可变长参数方法的定义**

使用...表示可变长参数，例如：

|  |
| --- |
| print(String... args){  ...  } |

在具有可变长参数的方法中可以把参数当成数组使用，例如可以循环输出所有的参数值:

|  |
| --- |
| print(String... args){  for(String temp:args)  System.out.println(temp);  } |

* **可变长参数的方法的调用**

调用的时候可以给出任意多个参数，例如：

|  |
| --- |
| print("hello");  print("hello"，"lisi");  print("hello"，"张三"); |

* **可变参数匹配**

如果要调用的方法可以和两个可变参数匹配，则出现错误，例如下面的代码：

|  |
| --- |
| public static void main(String[] args) {  VarArgsTest test = new VarArgsTest();  test.print("hello");  test.print("hello"，"zhangsan");  }    public void print(String... args){  for(int i=0;i<args.length;i++){  out.println(args[i]);  }  }    public void print(String test，String...args ){  out.println("----------");  }  } |

对于上面的代码，main方法中的两个调用都不能编译通过。

* **一个方法只能有一个可变长参数，并且这个可变长参数必须是该方法的最后一个参数**

以下两种方法定义都是错误的：

|  |
| --- |
| public void test(String... strings，ArrayList list)  public void test(String... strings，ArrayList... list) |

* 1. 线程

在我们的应用程序开发过程中，请总是将多线程并发考虑进去。

* + 1. **线程的状态**

线程的状态转换是线程控制的基础。线程状态总的可分为五大状态：分别是生、死、可运行、运行、等待/阻塞。

可运行

运行

等待阻塞睡眠

创建

终结

* 创建：线程对象已经创建，还没有在其上调用start()方法；
* 可运行：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态；
* 运行：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式；
* 等待/阻塞/睡眠状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，它是可运行的，但是如果某件事件出现，他可能返回到可运行状态；
* 终结态：当线程的run()方法完成时就认为它终结。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦终结，就不能复生。 如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException 异常。
  + 1. **同步Synchronized**
       1. **synchronized 方法**

synchronized 声明是为了定义变量的作用范围和作用域，通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。如：

|  |
| --- |
| public synchronized void accessVal(int newVal); |

synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）。

在 Java 中，不光是类实例，每一个类也对应一把锁，这样我们也可将类的静态成员函数声明为 synchronized ，以控制其对类的静态成员变量的访问。

synchronized 方法的缺陷：若将一个大的方法声明为synchronized 将会大大影响效率，典型地，若将线程类的方法 run() 声明为 synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。当然我们可以通过将访问类成员变量的代码放到专门的方法中，将其声明为 synchronized ，并在主方法中调用来解决这一问题，但是 Java 为我们提供了更好的解决办法，那就是 synchronized 块。

* + - 1. **synchronized 块**

通过 synchronized关键字来声明synchronized 块。语法如下：

|  |
| --- |
| synchronized(syncObject) {  //允许访问控制的代码  } |

synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁方能执行，具体机制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。

* + - 1. **synchronized(this)**
* 当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块；
* 当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞；
* 当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的除synchronized(this)同步代码块以外的部分；
* 上条同样适用其它同步代码块。即是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞；
* 以上规则同样适用于其它对象锁。
  + - 1. **synchronized 类**

如果用synchronized修饰类，则线程在该类中所有操作不能同时进入，包括静态变量和静态方法，实际上，对于含有静态方法和静态变量的代码块的同步，通常用synchronized修饰类来加锁。

锁是和对象相关联的，每个对象有一把锁，为了执行synchronized语句，线程必须能够获得synchronized语句中表达式指定的对象的锁，一个对象只有一把锁，被一个线程获得之后它就不再拥有这把锁，线程在执行完synchronized语句后，将获得锁交还给对象。

在方法前面加上synchronized修饰符即可以将一个方法声明为同步化方法。同步化方法在执行之前获得一个锁。如果这是一个类方法，那么获得的锁是和声明方法的类相关的Class类对象的锁。如果这是一个实例方法，那么此锁是this对象的锁。

* + 1. **考虑死锁**

死锁是指多个线程同时阻塞，其中的一个或者多个都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

导致死锁的根源在于不适当地运用synchronized关键词来管理线程对特定对象的访问。synchronized关键词的作用是，确保在某个时刻只有一个线程被允许执行特定的代码块，因此，被允许执行的线程首先必须拥有对变量或对象的排他性的访问权。当线程访问对象时，线程会给对象加锁，而这个锁导致其它也想访问同一对象的线程被阻塞，直至第一个线程释放它加在对象上的锁。

由于这个原因，为了避免在使用synchronized关键词时出现两个线程互相等待对方释放某个资源的情况，需要注意以下几点：

* 在一个synchronized修饰的代码块中，尽可能的做最小的事情。
* 在一个synchronized修饰的代码块中，不要调用不明确的方法，比如你并不清楚那个方法里会占用到何种资源。
* 将涉及通信访问、文件处理的方法移到synchronized修饰块的外面。
* 考虑采用并发工具来实现wait和notify。
* 有时候，使用线程安全的容器可以代替synchronized块的实现。
  1. 异常
     1. **受检异常、运行时异常、错误**

Java程序设计语言提供了三种可抛出的结构：受检异常（Checked exception）、运行时异常(Runtime exception)和错误(Error)。其继承结构如下图所示：

Throwable

Error

Exception

RuntimeException

Throwable类是所有异常类的父类，实现了Serializable接口。Throwable有两个重要的子类：Exception和Error，分别表示异常和错误。异常一般表示可检测、可恢复或者在编码中可以避免的问题，一般是比较轻度的错误；错误通常表示严重的问题，大多数情况下与代码的执行无关，比如OutOfMemoryError。

其中受检的异常表示期望调用者在异常发生时能够采取一些恢复的措施，或者将该异常传播出去。当一个方法抛出受检异常时，意思就是告诉调用者，要调用我这个方法，你必须捕获我抛出的异常。当然调用者捕获这个异常之后可以忽略，但这通常不是个好办法。

运行时异常和错误属于未受检的可抛出结构，在行为上是相同的：他们都不需要被捕获。这是因为当我们的程序抛出这两种结构时，通常代表的是该错误不可被恢复，继续执行下去有害无益，比如IndexOutOfBoundsException、ArithmeticException和OutOfMemoryError。IndexOutOfBoundsException表示数组或者字符串超出了索引范围，ArithmeticException表示运算异常（比如分母为0），OutOfMemoryError表示JVM没有足够的内存来分配。前两种异常都属于RuntimeException，表明这种异常属于编程错误，我们应该在我们的程序中就避免这种问题，而不是通过捕获来解决。后一种属于错误，是我们无法通过编码来解决的，这种错误通常都是比较严重的问题，一般是JVM出了问题。

* + 1. **不要忽略异常**

这是处理异常最基本的原则，当一个异常被抛出的时候，说明程序中某个地方出了问题，忽略异常可能达不到我们期望的效果。我们应该在程序中对出现的异常进行处理，或者是继续抛出。比如，如下的代码毫无意义：

|  |
| --- |
| **try** {  // do somthing...  }**catch**(Exception e){} |

* + 1. **少用异常，尽可能减少自定义异常，优先使用标准异常**

异常的处理会有一定的开销，除非是在出现异常的情况下，否则不要将异常用于普通的控制流。比如，不要将异常用来检测数组是否越界。如下：

|  |
| --- |
| **try** {  **int** i = 0;  **while** (**true**)  gateway[i++].submit();  } **catch** (ArrayIndexOutOfBoundsException e) {  } |

以上代码完全等价于：

|  |
| --- |
| **for**(Gateway gateway: allGateways){  gateway.submit();  } |

但是性能上要慢了近100倍！此外，不要总是依赖异常来解决空指针问题，我们在调用方法之前就应该判断对象是否为null。注意要判断所有需要的对象是否为null，比如：

|  |
| --- |
| **if**(string.length() >0 && string.equals("astring")) |

但是string对象可能是null！那么该判断就会直接抛出NullPointerException。可以写成这样：

|  |
| --- |
| **if**("astring".equals(string)) |

在使用到异常的地方，优先使用标准异常而不是自定义异常，这样会使你的代码更具可读性（因为标准异常大家都熟悉），同时也减少了异常类。常见的标准异常有：

* IllegalArgumentException：这个异常表示调用者传递了不合适的参数。一般在检测到参数不正确的时候我们可以抛出这个异常，或者是返回null值或false，结束方法；
* NullPointerException：这个是我们最熟悉的异常了。如果调用者在某个不允许null值的参数中传递了null值，习惯的做法就是抛出NullPointerException异常；
* IndexOutOfBoundsException：如果调用者在某个序列下标的参数中传递了越界的值，应该抛出的就是IndexOutOfBoundsException异常。比如访问超过数组下标的数组元素；
* ConcurrentModificationException：这个异常被设计在java.util包中，用来表示一个单线程的对象正在被并发的修改；
* UnsupportedOperationException：这个异常表示当前对象不支持所请求的操作。比如在实现类中没有实现接口定义的方法，就会抛出这个异常；
* NumberFormatException：这个异常表示数据格式有误，还有一个ArithmeticException异常，表示算术异常，比如在除法运算中传递了0作为除数；
  + 1. **为方法抛出的异常撰写说明文档**

不仅要为每个方法建立文档，为每个方法抛出的异常建立完善的文档也是很有必要的。始终要单独声明每个受检的异常，并且利用Javadoc的@throws标记，准确地记录产生每个异常的条件。如下所示：

|  |
| --- |
| /\*\*  \* resolve input data from inputstream  \*  \* **@param** stream the InputStream from which to read the data  \*  \* **@exception** IOException  \* on errors reading, or on invalid data  \*/  **public** **void** resolve(**final** InputStream stream) **throws** IOException |

对于未受检异常，只要使用@throws标签记录下可能抛出的异常以及条件就可。但不要使用throws关键字将该异常包含在方法的声明中。因为使用throws关键字声明的异常必须在调用者的代码中进行捕获，而未受检的异常一般是不需要调用者捕获。

* + 1. **记录所捕获到异常的有用信息**

异常发生时，通常需要查看产生异常的原因来排查问题，所以，在异常中包含尽可能有用的细节信息对解决问题有很大的帮助。一般的异常都会记录异常产生的类名、方法名和发生异常的位置，以及大量的堆栈信息。

实际上仅有这些信息也不能完全满足我们的需求。有时候，通过异常能判断是操作数据库的时候出错了，但还不能确定是SQL语句有错，还是没有操作权限，还是数据库操作过程中发生了其它错误，等等。这时候，就可能希望能看到执行的SQL语句或其它中间状态，记录这些信息也是很有必要的。因此，异常应该包括所有“对该异常有贡献”的参数和域的值。

有两种方式可以实现，一种做法是在捕获异常后再记录这些信息；另一种做法是在自定义异常类的构造器中引入这些信息。比如我们可以在IndexOutOfBoundsException异常中定义一个如下的构造器。

|  |
| --- |
| **public** IndexOutOfBoundsException(**int** lowerBound, **int** upperBound,  **int** index) {  **super**("Lower bound:" + lowerBound + ", Upper bound:"  + upperBound + ", Index:" + index);  **this**.lowerBound = lowerBound;  **this**.upperBound = upperBound;  **this**.index = index;  } |

其实在SQLException里是有比较详细的信息记录的，看一下SQLException的源代码就会发现它有一个如下的构造器：

public SQLException(String reason, String sqlState, int vendorCode, Throwable cause)

这个构造器为我们详细的记录了异常发生的原因。Reason表示异常发生的原因；sqlState是XOPEN或者SQL:2003（一种规范）关于异常分类的代码；vendorCode是数据库特定的异常代码。

有了以上的信息之后，就可以通过SQLException中的getSQLState()和getErrorCode()方法来获得异常更精确的异常信息。

* + 1. **努力使失败保持原子性**

失败的原子性，意思就是失败的方法调用应该使对象保持在调用之前的状态。这一点听起来比较抽象，但是实际上很容易理解。试想想一种情况，在捕获了受检的异常之后，代码并不会终止执行，而是试图从这个异常中恢复。但是有一个对象的值却在异常中被改变了，这样我们的下一步操作可能取到的是一个不期望得到的值。

举个例子，假如要统计进行了某种操作的转发，用下面的代码，或许在转发的时候就失败了，抛出了异常，但是计数器却进行了加1，这可能不是我们期望的结果：

|  |
| --- |
| **for** (String account : accountlist) {  count++;  Object obj = transport.submit(request, ctx);  **if** (baskItem != **null**) {  // ...;  }  } |

为了避免异常所产生的对象改变，可以有很多预防办法：

* 在执行操作之前对参数进行校验，如果参数不对就抛出异常，避免后续的操作改变对象的值。（实际上在一个方法开始之前进行参数校验是很好的做法，不仅可以避免异常带来的对象改变，还可以避免不必要的开销。）
* 调整处理顺序，使得任何可能引起对象改变的操作在可能产生异常的调用之后。
* 在捕获异常之后编写一段恢复的代码，进行回滚；
* 拷贝一份对象，在对象的拷贝上执行操作。当执行完毕后用拷贝的对象替换原来的对象；

并不是针对任何异常都需要保持失败的原子性，如果方法抛出了某种错误，使失败保存原子性就没有多大意义了。比如出现OutOfMemoryError，就没必要再从错误中恢复了。

* + 1. **不要一次捕获所有异常**

不要使用如下一劳永逸的代码：

|  |
| --- |
| **try** {  // …… ……抛出AException  // …… ……抛出BException  // …… ……抛出CException  } **catch** (Exception e) {  logger.error("XXX error", e.getCause());  } |

这段代码的特点是用一个catch子句捕获了所有的异常，看上去省事很多。实际上是存在缺陷的。一是可能针对不同的异常有不同的恢复措施，而这里的代码让我们无法区分异常，也就无法针对性的恢复。二是这种方式掩盖了代码中的RuntimeException异常，通常出现RuntimeException属于编程错误，也就是这种方式掩盖了我们的编程错误。

针对一段代码中可能有多个调用会产生异常，应该分别捕获每种异常，进行不同的处理。

* + 1. **针对不同情景使用不同异常**

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。这一点在明确了受检异常和未受检异常的区别就不难理解了。受检异常是在编程中必须捕获的异常。捕获异常的目的就是在异常发生时希望采取一些措施来恢复。那如果我们希望调用者在调用方法出错之后能有补救措施，就应该抛出受检异常。而对于一些编程错误，应该是调用者能够在代码中就避免，这时候就应该抛出未受检的异常RuntimeException，提示调用者改进代码。比如前面提到的IndexOutOfBoundsException异常，出现这类异常，就应该考虑在代码中避免数组越界了。

* + 1. **在finally块中释放资源**

针对有些操作，比如IO、Socket或者数据库操作，发生异常的时候，我们还需要正确释放占用的资源。一般我们在finally块中完成资源的释放，值得注意的是在finally块中的子句也可能会出现异常，我们在编码的时候要尽量避免出现这种异常。如下所示：

|  |
| --- |
| **try** {  ……  reader = **new** BufferedReader(**new** InputStreamReader(  **new** FileInputStream(file)));  **for** (String record = reader.readLine(); record != **null**;) {  ……  }  } **catch** (FileNotFoundException e) {  log.error("error while reading file " + file.getName());  } **catch** (IOException e) {  log.error("error while reading file " + file.getName() + e);  } **catch** (Exception e) {  log.error("baskitem 任务失败...");  } **finally** {  **try** {  reader.close();  } **catch** (IOException e) {  log.error("error while closeing file " + file.getName());  }  } |

* 1. 通信
     1. **HTTP**
        1. **使用场景**

HTTP协议一般用于同步响应处理环境中。

* + - 1. **使用规范**
* 在http1.1中，是默认长连接的， 如果不使用长链接，则需要在header中指明connection的值为close。
* get和post方法，优先使用post。
* Cookie和Session都为了用来保存状态信息，建议使用SESSION。
* http.socket.timeout：以毫秒为单位定义套接字超时（SO\_TIMEOUT），如果此参数不设置，读操作不会超时，开发时要设置超时时间。
* 无论执行的操作成功与否，一定要释放连接
  + 1. **JMS**
       1. **使用场景**

JMS可以异步也可以同步响应，如果有异步方式，优先考虑JMS传输，并且JMS另一个场景为发布订阅的方式发布消息。JMS通过底层的中间件MQ可以保证对消息的持久性、唯一性及可靠性和事务提交等优点。

* + - 1. **使用规范**
* JMS Session及子对象Producer、Consumer都是非线程安全的
* 应用的MQ依赖jar包的版本一定要和所连接的MQ的版本一样
* 在应用程序中一定要关闭JMS对象
* JMSException抛出时，最好是能获取JMS内部信息的LinkedException，这对问题诊断很有帮助
* 事务Session适用事务的消息场景，非实物Session适用非事务的消息场景
* DUPS\_OK\_ACKNOWLEDGE和AUTO\_ACKNOWLEDGE性能优于CLIENT\_ACKNOWLEDGE
* 如无特殊需求，消息应该设置为非持久化消息(NON\_PERSISTENT)
* 设置消息的有效期时间（TimeToLive）尽量合理，不应太长。
* 尽可能的使用连接池
  + 1. **Socket**
       1. **使用场景**

Socket（套接字）是基于TCP/IP协议的应用编程接口（API）的一种，应用程序通过它来发送和接收数据，就像应用程序打开一个文件句柄，将数据读写到稳定的存储器上一样。一个socket允许应用程序添加到网络中，并与处于同一个网络中的其他应用程序进行通信。一台计算机上的应用程序向socket写入的信息能够被另一台计算机上的另一个应用程序读取，反之亦然。

TCP/IP 协议族中的主要socket类型为流套接字（sockets sockets）和数据报套接字（datagram sockets）。流套接字将TCP作为其端对端协议（底层使用IP协议），提供了一个可信赖的字节流服务。流式套接口定义了一种可靠的面向连接的服务，实现无差错无重复的顺序数据传输。一个TCP/IP流套接字代表了TCP连接的一端；数据报套接字使用UDP协议（底层同样使用IP协议），提供了一个"尽力而为"（best-effort）的数据报服务，数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠，无差错。

一个 TCP/IP 套接字由一个互联网地址，一个端对端协议（TCP或 UDP协议）以及一个端口号唯一确定，Socket适用于基于TCP/IP传输的数据包交互和各个不同系统与应用在网络间交互的环境中。

* + - 1. **使用规范**
* Socket报文长度规范

在使用socket进行通信时，规定报文前几位为标志位,内容填充为报文传输内容长度。例如如果规定前8位为长度位，报文实际内容长度为1024字节，则通信时，需传输00001024+实际报文数据来传输；前8位内容不足以0填充；如：00000010abcdabcdad

* Socket超时时间规范

socket在建立通讯连接时，必须设置超时时间；建议客户端和服务端的超时时间根据具体业务自定义；

* 长连接和短连接规范

短连接是指通信双方有数据交互时就建立一个连接，数据发送接收完成后，则断开此连接。适用于并发量大且交互时间短的应用；如：web网站的http服务；

长连接是指建立socket连接后，不管以后是否使用都保持连接状态；长连接多用户操作频繁，点对点的通讯，而且连接数不是太多且可控的情况；因每个TCP连接都需要三步握手，这需要时间，如每个操作都先连接再操作的话，处理速度肯定会下降很多，所有每次处理完成后，都不断开，以后处理时直接发送数据即可，不需建立连接；如：应用连接数据库的连接。

* 流套接字和数据包套接字规范

Socket类型主要分两种：流套接字和数据包套接字；建议使用流套接字，以保证数数据传输的准确性和应用的稳定性。

* 合理消除缓冲延迟

TCP协议将数据缓存起来直到足够多时一次发送，以避免发送过小的数据包而浪费网络资源。虽这个功能有利于网络，但应用程序可能对所造成的缓冲延迟不能容忍，建议对缓冲延迟进行设置，以提高响应。

* 设置发送和接收缓冲区大小

通常情况下，在没有网络容量或其他限制的情况下，越大的缓冲区通常能够实现越高的吞吐量。建议设置发送和接收缓冲区为合适的值以提高数据吞吐量。

* 设置Keep-Alive

TCP协议提供了一种 keep-alive的机制，该机制在经过一段不活动时间后，将向另一个终端发送一个探测消息。如果另一个终端还出于活跃状态，它将回复一个确认消息。如果经过几次尝试后依然没有收到另一终端的确认消息，则终止发送探测信息，关闭套接字，并在下一次尝试 I/O操作时抛出一个异常。默认情况下，keep-alive机制是关闭的。可以设置为TRUE，开启keep-alive 机制；建议根据实际情况，合理使用Keep-Alive属性；

* 1. 错误定位与日志
     1. **日志框架**

JUMP开发统一采用第三方开源日志组件Logback。

* + 1. **日志记录**

系统日志是系统问题分析、错误定位的重要参考。为了快速定位错误，解决问题，需要合理仔细的进行日志记录。为此，需要遵守以下规范：

* 不得使用System.out.println或者excetion.printStackTrace来打印日志信息，使用日志类。

|  |
| --- |
| System.out.println(“some log message.”) //错误  logger.info(“some log message”); //正确  try{  }catch(Exception e){  e.printStackTrace();  } //错误  try{  }catch(Exception e){  Logger.error(“…处理失败，原因为:”,e);  } //正确 |

* Logger的实例声明在类字段之后，必须为private static final,示例如下。

|  |
| --- |
| **private** **static** **final** Logger *logger* = LoggerFactory.*getLogger*(DataAction.**class**); //正确  **private** Logger *logger* =  LoggerFactory.*getLogger*(DataAction.**class**); //错误 |

* 日志内容要求精确，内容描述详细，每个条日志语句必须同时包含数据和描述。

|  |
| --- |
| logger.debug("Message processed");  logger.debug(message.getJMSMessageID()); //错误    logger.debug(“Message with id ‘{}’ processed”,  message.getJMSMessageID()); *//*正确 |

* 重要方法的参数和返回值需要进行记录。

|  |
| --- |
| **public** String printDocument(Document doc, Mode mode){  logger.debug("Entering printDocument(doc={},mode={})",  doc,mode);  ………………  logger.debug(“Leaving printDocument():{}”,id);  return id;  } |

* 假如需要日志的内容需要大量计算获得，需要使用is\*Enabled方法进行判断。例如：

|  |
| --- |
| if(logger.isDebugEnabled()){  MQMessage message = response.getMessage();  **byte**[] byteMessage = **new** **byte**[message.getDataLength()];  String hexMessage = StringTool.toHexString(byteMessage);  logger.debug(“响应报文字节流为:\n{}”,hexMessage);  } |

* 外部系统调用，需要记录所有系统间交互信息。

|  |
| --- |
| PostMethod post = new PostMethod(url);  post.addRequestHeader("Target-App", "ThisIsSelfDefHeader");  post.setRequestHeader("Content-Type","text/xml; charset=utf-8");  post.setRequestBody(new ByteArrayInputStream(msg));  logger.debug("调用外部系统，请求信息为:{}",msg);  client.executeMethod(post);    if(logger.isDebugEnabled()){  final String responseBody =  new String(post.getResponseBody(),"utf-8");  Header[] headers = post.getResponseHeaders();    logger.debug("外部系统调用结束，返回消息头，返回报文{}",  responseHeader,responseBody);  } |

* 异常信息日志需要发生在异常被最终处理的地方。

|  |
| --- |
| try{  ………………  }catch(IOException e){  logger.error(“IO exception”,e);  throw new MyCustomException(e); } //错误 |

* 日志信息中不能包含敏感信息，如卡号，密码等信息。

|  |
| --- |
| logger.debug(“用户登录成功，用户名={},密码={}。”,  userName,password); //错误。 |

* 禁止使用字符串拼接的方式构建日志信息。

|  |
| --- |
| logger.debug(“Found {} records matching filter :’{}’,  records, fileter ); //正确  logger.debug(“Found ” + records + “ records matching filter : ‘” + filter + “’”); //错误 |

* + 1. **日志级别**

在使用logger进行日志记录时选择合适的日志级别，具体如下：

* ERROR

ERROR级别在系统发生重大错误，需要立刻解决的时候使用。使用这个级别记录的错误时，系统无法继续正常运行。如数据库不可用,队列管理器不可用。

* WARN

用户提示在系统运行过程中违背预设业务逻辑，不会导致异常出现的情况，但是要关注该日志信息,如 “某个字段数据取值的异常，使用默认值”，“无法获取当前数据，使用缓存内容”等。

* INFO

INFO级别用来记录重要的处理过程的完成状态。INFO级别的信息可以给管理员或者高级用户提供应用工作的相关信息。

* DEBUG

DEBUG级别的信息主要供开发人员使用，不建议生产系统使用DEBUG作为日志输出的等级。

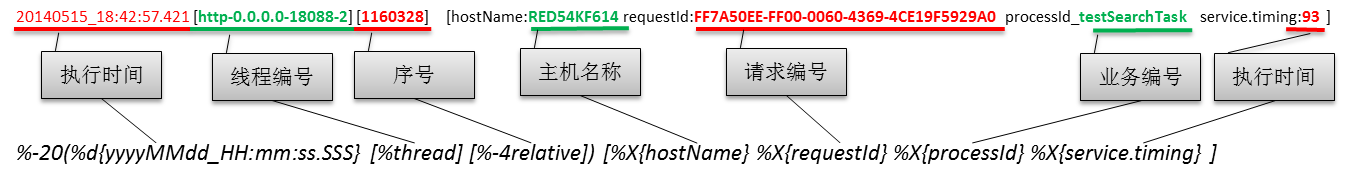
* TRACE

TRACE级别的信息记录程序运行的详细信息，主要供开发人员在开发环节使用，TRACE调试信息在开发测试完成后需要删除。

* + 1. **日志规范**

日志规范是指在原有的Logback的基础，平台约定的日志格式标准配置(logback.xml)，以便于各开放应用在日志输出的过程中，能够统一日志的格式，为日志分析和图形化展示提供基础。具体的日志规范如下：

1. 统一使用Logback进行日志输出，Logback的版本为JUMP-BASE默认提供的版本
2. 根据Logback的约定，日志级别包括TRACE、DEBUG、INFO、WARN和ERROR。
3. 输入到本地磁盘的日志文件。应该以滚动日志结合存档日志的方式进行管理。滚动日志的文件大小应控制在50MB以内。对大于50MB的日志信息应自动转为存档日志。
4. 对应存档日志和滚动日志应该以自然日为单位进行归集，归集的结果应以zip的方式进行压缩。
5. 具体的日志信息应该包括两个部分：日志头和日志体。
6. 日志体为应用控制的需要描述的具体日志信息；
7. 日志头为标准化的日志描述信息，用于说明日志发生的具体场景，标准的日志头格式如下图所示：



1. 通过日志头的信息可以归集同一个事务单元、同一笔业务等统计分析功能，便于分析交易的执行时间和执行次序；
2. 建议将日志信息分模块输出，或者至少应该将错误日志和正常日志分开写入到不同的日志文件中。
   * 1. **配置示例**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <property name="LOG\_DIR" value="logs" /><!-- logs -->  <property name="LOG\_FILE\_NAME" value="jump" /> <!-- 可以由项目组中进行调整 -->  <property name="logFileNamePattern" value="${LOG\_FILE\_NAME}\_%d{yyyy-MM-dd}\_%i" />  <property name="fileLayoutPattern"  value="%-20(%d{yyyyMMdd\_HH:mm:ss.SSS} [%thread] [%-4relative])  [%X{hostName} %X{requestId} %X{processId} %X{service.timing} ] %msg%n" />  <property name="consoleLayoutPattern"  value="%-20(%d{yyyyMMdd\_HH:mm:ss.SSS} [%thread] [%-4relative])  [%X{hostName} %X{requestId} %X{processId} %X{service.timing} ] %msg%n" />  <!-- contextName could use in pattern, like '%contextName' -->  <contextName>osgiApp</contextName>  <appender name="ROLLING"  class="ch.qos.logback.core.rolling.RollingFileAppender">  <file>${LOG\_DIR}/${LOG\_FILE\_NAME}.txt</file>  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <fileNamePattern>${LOG\_DIR}/${logFileNamePattern}.txt.zip  </fileNamePattern>  <timeBasedFileNamingAndTriggeringPolicy  class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">  <maxFileSize>50MB</maxFileSize>  </timeBasedFileNamingAndTriggeringPolicy>  <maxHistory>15</maxHistory><!-- days -->  </rollingPolicy>  <layout>  <pattern>${fileLayoutPattern}</pattern>  </layout>  </appender>  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">  <layout name="StandardFormat" class="ch.qos.logback.classic.PatternLayout">  <pattern>${consoleLayoutPattern}</pattern>  </layout>  </appender>    <logger name="com.bocom.jump.jdbcdslog" level="ERROR" additivity="false">  <appender-ref ref="ROLLING" />  </logger>  <root level="INFO">  <appender-ref ref="CONSOLE" />  <appender-ref ref="ROLLING" />  </root>  </configuration> |

有关日志相关的更详细的开发说明，可以参考《JUMP-BASE开发手册-组件分册》和Logback开源软件的用户手册。

* 1. 避免内存泄漏

由于Java中有内存垃圾收集的机制，并且没有指针类型，开发者就可以从C/C++的内存噩梦中解脱出来。正因为如此，很多开发者就不再关注内存的使用问题。但是，这并不代表程序员可以完全不关注Java程序的内存管理问题，若是代码编写不得当，仍然可能造成内存泄漏的问题。

* + 1. **尽早释放无用对象的引用。**

Java的这个GC机制是内存垃圾收集而非内存收集，因此，需要把使用的对象转换为JVM可以识别的垃圾，要理解变量的作用域。一般而言，在方法中声明使用的变量在方法外面时就会成为垃圾。但是有些却不是，例如：

|  |
| --- |
| **private** HashMap dialogMap = **new** HashMap();  // ...  JDialog dialog = **new** JDialog("XXX");  // ...  dialogMap.put(dialog.getTitle()， dialog);  // ... |

如果之后在整个代码期间都没有调用如下语句：

|  |
| --- |
| Object ref =dialogMap.remove(key);  //...释放资源，如：  (JDialog)ref).dispose()  ref = **null**; |

或者

|  |
| --- |
| dialogMap.clear(); //有时，仅仅这个语句还不行 |

那么就存在内存泄漏问题。几乎所有存储对象的结构都要引起注意，看看是否有资源没有释放，没有成为垃圾而无法回收。比如数据库程序中，以下的对象要严格释放：

* java.sql.Connection;
* java.sql.Statement;
* java.sql.PreparedStatement
* java.sql.CallableStatement
* java.sql.ResultSet

这些对象如果不释放其资源，不仅仅是内存问题；还将引起数据库问题。如果不释放Connection，那么很快就用尽连接或是数据库变慢。

* + 1. **少用finalize函数**

Finalize函数会加大GC的工作量，造成不必要的麻烦。

* + 1. **注意集合数据类型**
* 包括数组、树、图、链表等数据结构，这些数据结构对GC来说，回收更为复杂。
* 避免在类的默认构造器中创建、初始化大量的对象
* 防止在调用其自类的构造器时造成不必要的内存资源浪费。
  + 1. **避免显式申请数组空间**

除非特别必要，尽可能的使用容器对象（如ArrayList，Map）来代替数组。

1. **编码格式规范**
   1. 文件组织

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。超过2000行的程序难以阅读，应该尽量避免。"Java源文件范例"提供了一个布局合理的Java程序范例。

* + 1. **文件名**

本章列出了项目开发中常用的文件名及其后缀。

* + 1. **文件后缀**

Java程序使用下列文件后缀：

* Java源文件 ：.java
* Java字节码文件：.class

其他文件使用下列文件后缀：

* 数据库脚本：.sql
* Shell脚本：.sh
* 数据文件：.dat
* 属性文件：.properties
* 配置文件：.xml
  + 1. **Java源文件**

每个Java源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

Java源文件还遵循以下规则：

* 开头注释（参见"开头注释"）
* 包和引入语句（参见"包和引入语句"）
* 类和接口声明（参见"类和接口声明"）
  + - 1. **开头注释**

所有的源文件都应该在开头有一个注释，其中列出类名、版本信息、日期和版权声明：

|  |
| --- |
| /\*  \* Class name  \*  \* Version information  \*  \* Date  \*  \* Copyright notice  \*/ |

* + - 1. **包和引入语句**

在多数Java源文件中，第一个非注释行是包语句。在它之后可以跟引入语句。例如：

|  |
| --- |
| **package** java.awt;  **import** java.awt.peer.CanvasPeer; |

* + 1. **类和接口声明**

下表描述了类和接口声明的各个部分以及它们出现的先后次序。

* 类/接口文档注释(/\*\*……\*/)：该注释中所需包含的信息，参见"文档注释"；
* 类或接口的声明；
* 类/接口实现的注释(/\*……\*/)：如果有必要的话，该注释应包含任何有关整个类或接口的信息，而这些信息又不适合作为类/接口文档注释；
* 类的(静态)变量：首先是类的公共变量，随后是保护变量，再后是包一级别的变量(没有访问修饰符，access modifier)，最后是私有变量；
* 实例变量：首先是公共级别的，随后是保护级别的，再后是包一级别的(没有访问修饰符)，最后是私有级别的；
* 类构造器；
* 方法：这些方法应该按功能，而非作用域或访问权限，分组。例如，一个私有的类方法可以置于两个公有的实例方法之间。其目的是为了更便于阅读和理解代码。
  1. 缩进排版

4个空格常被作为缩进排版的一个单位。缩进的确切解释并未详细指定(空格 vs. 制表符)。一个制表符等于8个空格(而非4个)。

* + 1. **行长度**

尽量避免一行的长度超过80个字符，因为很多终端和工具不能很好处理之。

注意：用于文档中的例子应该使用更短的行长，长度一般不超过70个字符。

* + 1. **换行**

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开之：

* 在一个逗号后面断开；
* 在一个操作符前面断开；
* 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开；
* 新的一行应该与上一行同一级别表达式的开头处对齐；
* 如果以上规则导致代码混乱或者使代码都堆挤在右边，那就代之以缩进8个空格。

以下是断开方法调用的一些例子：

|  |
| --- |
| someMethod(longExpression1， longExpression2， longExpression3，  longExpression4， longExpression5);  var = someMethod1(longExpression1，  someMethod2(longExpression2，  longExpression3)); |

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

|  |
| --- |
| longName1 = longName2 \* (longName3 + longName4 - longName5)  + 4 \* longname6; //PREFFER  longName1 = longName2 \* (longName3 + longName4  - longName5) + 4 \* longname6; //AVOID |

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进8个空格：

|  |
| --- |
| // CONVENTIONAL INDENTATION  **void** someMethod(**int** anArg, Object anotherArg, String yetAnotherArg, Object andStillAnother) {  // ...  }  // INDENT 8 SPACES TO AVOID VERY DEEP INDENTS  **private** **static** **synchronized** **void** horkingLongMethodName(**int** anArg,  Object anotherArg, String yetAnotherArg, Object andStillAnother) {  // ...  } |

if语句的换行通常使用8个空格的规则，因为常规缩进(4个空格)会使语句体看起来比较费劲。比如：

|  |
| --- |
| // DON’T USE THIS INDENTATION  **if** ((condition1 && condition2) || (condition3 && condition4)  || !(condition5 && condition6)) { // BAD  // WRAPS  doSomethingAboutIt(); // MAKE THIS LINE EASY TO MISS  }  // USE THIS INDENTATION INSTEAD  **if** ((condition1 && condition2) || (condition3 && condition4)  || !(condition5 && condition6)) {  doSomethingAboutIt();  }  // OR USE THIS  **if** ((condition1 && condition2) || (condition3 && condition4)  || !(condition5 && condition6)) {  doSomethingAboutIt();  } |

这里有三种可行的方法用于处理三元运算表达式：

|  |
| --- |
| alpha = (aLongBooleanExpression) ? beta : gamma;  alpha = (aLongBooleanExpression) ? beta  : gamma;  alpha = (aLongBooleanExpression)  ? beta  : gamma; |

* 1. 注释

Java程序有两类注释：实现注释(implementation comments)和文档注释(document comments)。实现注释是那些在C++中见过的，使用/\*...\*/和//界定的注释。文档注释(被称为"doc comments")是Java独有的，并由/\*\*...\*/界定。文档注释可以通过javadoc工具转换成HTML文件。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implementation-free)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如，相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中己清晰表达出来的重复信息。多余的的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意：频繁的注释有时反映出代码的低质量。当要加注释的时候，考虑一下重写代码使其更清晰。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

* + 1. **实现注释的格式**

程序可以有4种实现注释的风格：块(block)、单行(single-line)、尾端(trailing)和行末(end-of-line)。

* + 1. **块注释**

块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

|  |
| --- |
| /\*  \* Here is a block comment.  \*/ |

块注释可以以/\*-开头，这样indent(1)就可以将之识别为一个代码块的开始，而不会重排它。

|  |
| --- |
| / \*  \* Here is a block comment with some very special  \* formatting that I want indent(1) to ignore.  \*  \* one  \* two  \* three  \*/ |

注意：如果不使用indent(1)，就不必在代码中使用/\*-，或为他人可能对代码运行indent(1)作让步。

* + - 1. **单行注释**

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释(参见"块注释")。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

|  |
| --- |
| **if** (condition) {  /\* Handle the condition. \*/  } |

* + - 1. **尾端注释**

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个Java代码中尾端注释的例子：

|  |
| --- |
| **if** (a == 2) {  **return** TRUE; /\* special case \*/  } **else** {  **return** isPrime(a); /\* works only for odd a \*/  } |

* + - 1. **行末注释**

注释界定符"//"，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

|  |
| --- |
| **if** (foo > 1) {  // Do a double-flip.  ...  } **else** {  **return** **false**; // Explain why here.  }  **if** (bar > 1) {  // Do a triple-flip.  ...  } **else** {  **return** **false**;  } |

* + 1. **文档注释**

注意顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(/\*\*)不需缩进；随后的文档注释每行都缩进1格(使星号纵向对齐)。成员，包括构造方法在内，其文档注释的第一行缩进4格，随后每行都缩进5格。

若想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释(见5.1.1)或紧跟在声明后面的单行注释(见5.1.2)。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为Java会将位于文档注释之后的第一个声明与其相关联。

* 1. 声明
     1. **每行声明变量的数量**

推荐一行一个声明，因为这样以利于写注释。亦即，

|  |
| --- |
| **int** level; // indentation level  **int** size; // size of table |

要优于，

|  |
| --- |
| **int** level， size; |

不要将不同类型变量的声明放在同一行，例如：

|  |
| --- |
| **int** foo， fooarray[]; //WRONG! |

* + 1. **初始化**

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

* + 1. **布局**

严格控制变量生命周期(Live time)和跨度(Span)。不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

|  |
| --- |
| **void** myMethod() {  **int** int1 = 0; // beginning of method block  **if** (condition) {  **int** int2 = 0; // beginning of "if" block  ...  }  } |

该规则的一个例外是for循环的索引变量：

|  |
| --- |
| **for** (**int** i = 0; i < maxLoops; i++) { ... } |

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

|  |
| --- |
| **int** count;  ...  myMethod() {  **if** (condition) {  **int** count = 0; // AVOID!  ...  }  ...  } |

* + 1. **类和接口的声明**

当编写类和接口是，应该遵守以下格式规则：

* 在方法名与其参数列表之前的左括号"("间不要有空格；
* 左大括号"{"位于声明语句同行的末尾；
* 右大括号"}"另起一行，与相应的声明语句对齐，除非是一个空语句，"}"应紧跟在"{"之后：

|  |
| --- |
| **class** Sample **extends** Object {  **int** ivar1;  **int** ivar2;  Sample(**int** i， **int** j) {  ivar1 = i;  ivar2 = j;  }  **int** emptyMethod() {}  ...  } |

* 1. 语句
     1. **简单语句**

每行至多包含一条语句，例如：

|  |
| --- |
| argv++; // Correct  argc--; // Correct  argv++; argc--; // AVOID! |

* + 1. **复合语句**

复合语句是包含在大括号中的语句序列，形如"{ 语句 }"。例如下面各段。

* 被括其中的语句应该较之复合语句缩进一个层次；
* 左大括号"{"应位于复合语句起始行的行尾；右大括号"}"应另起一行并与复合语句首行对齐；
* 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如if-else或for控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入bug。
  + 1. **返回语句**

一个带返回值的return语句不使用小括号"()"，除非它们以某种方式使返回值更为显见。例如：

|  |
| --- |
| **return**;  **return** myDisk.size();  **return** (size ? size : defaultSize); |

* + 1. **if，if-else，if else-if else语句**

if-else语句应该具有如下格式：

|  |
| --- |
| **if** (condition) {  statements;  }  **if** (condition) {  statements;  } **else** {  statements;  }  **if** (condition) {  statements;  } **else** **if** (condition) {  statements;  } **else**{  statements;  } |

注意：if语句总是用"{"和"}"括起来，避免使用如下容易引起错误的格式：

|  |
| --- |
| **if** (condition) //AVOID! THIS OMITS THE BRACES {}!  statement; |

* + 1. **for语句**

一个for语句应该具有如下格式：

|  |
| --- |
| **for** (initialization; condition; update) {  statements;  } |

一个空的for语句(所有工作都在初始化，条件判断，更新子句中完成）应该具有如下格式：

|  |
| --- |
| **for** (initialization; condition; update); |

当在for语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在for循环之前(为初始化子句)或for循环末尾(为更新子句)使用单独的语句。

* + 1. **while语句**

一个while语句应该具有如下格式

|  |
| --- |
| **while** (condition) {  statements;  } |

一个空的while语句应该具有如下格式：

|  |
| --- |
| **while** (condition); |

* + 1. **do-while语句**

一个do-while语句应该具有如下格式：

|  |
| --- |
| **do** {  statements;  } **while** (condition); |

* + 1. **switch语句**

一个switch语句应该具有如下格式：

|  |
| --- |
| **switch** (condition) {  **case** ABC:  statements;  /\* falls through \*/  **case** DEF:  statements;  **break**;  **case** XYZ:  statements;  **break**;  **default**:  statements;  **break**;  } |

每当一个case顺着往下执行时(因为没有break语句)，通常应在break语句的位置添加注释。上面的示例代码中就包含注释/\* falls through \*/。

* + 1. **try-catch语句**

一个try-catch语句应该具有如下格式：

|  |
| --- |
| **try** {  statements;  } **catch** (ExceptionClass e) {  statements;  } |

一个try-catch语句后面也可能跟着一个finally语句，不论try代码块是否顺利执行完，它都会被执行。

|  |
| --- |
| **try** {  statements;  } **catch** (ExceptionClass e) {  statements;  } **finally** {  statements;  } |

* 1. 空白
     1. **空行**

空行将逻辑相关的代码段分隔开，以提高可读性。下列情况应该总是使用两个空行：

* 一个源文件的两个片段(section)之间；
* 类声明和接口声明之间。

下列情况应该总是使用一个空行：

* 两个方法之间；
* 方法内的局部变量和方法的第一条语句之间；
* 块注释或单行注释之前；
* 一个方法内的两个逻辑段之间，用以提高可读性。
  + 1. **空格**

下列情况应该使用空格：

* 一个紧跟着括号的关键字应该被空格分开，例如：

|  |
| --- |
| **while** (true) {  ...  } |

注意：空格不应该置于方法名与其左括号之间。这将有助于区分关键字和方法调用。

* 空白应该位于参数列表中逗号的后面；
* 所有的二元运算符，除了"."，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号("-")、自增("++")和自减("--")。例如：

|  |
| --- |
| a += c + d;  a = (a + b) / (c \* d);  **while** (d++ = s++) {  n++;  }  printSize("size is " + foo + "\n"); |

* for语句中的表达式应该被空格分开，例如：

|  |
| --- |
| **for** (expr1; expr2; expr3) |

* 强制转型后应该跟一个空格，例如：

|  |
| --- |
| myMethod((**byte**) aNum， (**Object**) x);  myMethod((**int**) (cp + 5)， ((**int**) (i + 3)) + 1); |

* + 1. **命名规范**

命名规范使程序更易读，从而更易于理解。它们也可以提供一些有关标识符功能的信息，以助于理解代码，例如，不论它是一个常量，包，还是类。

|  |  |  |
| --- | --- | --- |
| **标识符类型** | **命名规则** | **例子** |
| 包(Packages) | 一个唯一包名的前缀总是全部小写的ASCII字母并且是一个顶级域名，通常是com，edu，gov，mil，net，org，或1981年ISO 3166标准所指定的标识国家的英文双字符代码。包名的后续部分根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门(department)，项目(project)，机器(machine)，或注册名(login names)。 | com.sun.eng  com.apple.quicktime.v2  edu.cmu.cs.bovik.cheese |
| 类(Classes) | 命名规则：类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被更广泛使用，像URL，HTML) | class Raster;  class ImageSprite; |
| 接口(Interfaces) | 命名规则：大小写规则与类名相似 | interface RasterDelegate;  interface Storing; |
| 方法(Methods) | 方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。 | run();  runFast();  getBackground(); |
| 变量(Variables) | 除了变量名外，所有实例，包括类，类常量，均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。  变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量。临时变量通常被取名为i，j，k，m和n，它们一般用于整型；c，d，e，它们一般用于字符型。 | char c;  int i;  float myWidth; |
| 实例变量(Instance Variables) | 大小写规则和变量名相似，除了前面需要一个下划线 | int \_employeeId;  String \_name;  Customer \_customer; |
| 常量(Constants) | 类常量和ANSI常量的声明，应该全部大写，单词间用下划线隔开。(尽量避免ANSI常量，容易引起错误) | static final int MIN\_WIDTH = 4;  static final int MAX\_WIDTH = 999;  static final int GET\_THE\_CPU = 1; |

* 1. 编程惯例
     1. **提供对实例以及类变量的访问控制**

若没有足够理由，不要把实例或类变量声明为公有。通常，实例变量无需显式的设置(set)和获取(gotten)，通常这作为方法调用的边缘效应 (side effect)而产生。

一个具有公有实例变量的恰当例子，是类仅作为数据结构，没有行为。亦即，若要使用一个结构(struct)而非一个类(如果java支持结构的话)，那么把类的实例变量声明为公有是合适的。

* + 1. **引用类变量和类方法**

避免用一个对象访问一个类的静态变量和方法。应该用类名替代。例如：

|  |
| --- |
| classMethod(); //OK  AClass.classMethod(); //OK  anObject.classMethod(); //AVOID! |

* + 1. **常量**

位于for循环中作为计数器值的数字常量，除了-1，0和1之外，不应被直接写入代码。

* + 1. **变量赋值**

避免在一个语句中给多个变量赋相同的值。它很难读懂。例如：

|  |
| --- |
| fooBar.fChar = barFoo.lchar = 'c'; // AVOID! |

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如：

|  |
| --- |
| **if** (c++ = d++) { // AVOID! (Java disallows)  ...  } |

不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。例如：

|  |
| --- |
| d = (a = b + c) + r; // AVOID! |

应该写成

|  |
| --- |
| a = b + c;  d = a + r; |

* + 1. **其它惯例**
       1. **圆括号**

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。即使运算符的优先级可能看上去很清楚，但对其他人未必如此。不能假设别的程序员也清楚运算符的优先级。

|  |
| --- |
| **if** (a == b && c == d) // AVOID!  **if** ((a == b) && (c == d)) // RIGHT |

* + - 1. **返回值**

设法让程序结构符合目的。例如：

|  |
| --- |
| **if** (booleanExpression) {  return true;  } **else** {  return false;  } |

应该代之以如下方法：

|  |
| --- |
| **return** booleanExpression; |

类似地：

|  |
| --- |
| **if** (condition) {  **return** x;  }  **return** y; |

应该写做：

|  |
| --- |
| **return** (condition ? x : y); |

* + - 1. **条件运算符"?"前的表达式**

如果一个包含二元运算符的表达式出现在三元运算符" ? : "的"?"之前，那么应该给表达式添上一对圆括号。例如：

|  |
| --- |
| (x >= 0) ? x : -x; |

* + - 1. **特殊注释**

在注释中使用XXX来标识某些未实现(bogus)的但可以工作(works)的内容。用FIXME来标识某些假的和错误的内容。

* 1. 代码范例

下面的例子，展示了如何合理布局一个包含单一公共类的Java源程序。接口的布局与其相似。更多信息参见"类和接口声明"以及"文挡注释"。

|  |
| --- |
| /\*  \* @(#)Blah.java 1.82 99/03/18  \*  \* Copyright (c) 1994-1999 Sun Microsystems， Inc.  \* 901 San Antonio Road， Palo Alto， California， 94303， U.S.A.  \* All rights reserved.  \*  \* This software is the confidential and proprietary information of Sun  \* Microsystems， Inc. ("Confidential Information"). You shall not  \* disclose such Confidential Information and shall use it only in  \* accordance with the terms of the license agreement you entered into  \* with Sun.  \*/  **package** java.blah;  **import** java.blah.blahdy.BlahBlah;  /\*\*  \* Class description goes here.  \*  \* **@version** 1.82 18 Mar 1999  \* **@author** Firstname Lastname  \*/  **public** **class** Blah **extends** SomeClass {  /\* A class implementation comment can go here. \*/  /\*\* classVar1 documentation comment \*/  **public** **static** **int** *classVar1*;  /\*\*  \* classVar2 documentation comment that happens to be  \* more than one line long  \*/  **private** **static** Object *classVar2*;  /\*\* instanceVar1 documentation comment \*/  **public** Object instanceVar1;  /\*\* instanceVar2 documentation comment \*/  **protected** **int** instanceVar2;  /\*\* instanceVar3 documentation comment \*/  **private** Object[] instanceVar3;  /\*\*  \* ...constructor Blah documentation comment...  \*/  **public** Blah() {  // ...implementation goes here...  }  /\*\*  \* ...method doSomething documentation comment...  \*/  **public** **void** doSomething() {  // ...implementation goes here...  }  /\*\*  \* ...method doSomethingElse documentation comment...  \* **@param** someParam description  \*/  **public** **void** doSomethingElse(Object someParam) {  // ...implementation goes here...  }  } |

1. **规范检查标准100条**

* **BigInteger实例**

不要自行创建一个已经存在的BigInteger实例，如BigInteger.ZERO, BigInteger.ONE,BigInteger.TEN 以及 BigDecimal实例，如 BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN。

* **clone抛出clone不支持的异常**

clone方法必须声明抛出CloneNotSupportedException。

* **clone方法应该实现Clonable接口**

重载clone方法的类必须声明实现了Cloneable接口、抛出CloneNotSupportedException。

* **default在switch的最后**

总是把default放在switch条件的最后。

* **equals中无用的null检测**

在判断一个对象是否为空之后，你应该调用这个对象的equlas方法，而不是将此对象作为参数传递给另一个对象的equlas方法。

* **Finalize没有调用父类Finalize**

如果实现了finalize()方法，方法中的最后一行应该是super.finalize。

* **Final属性可以使用Static**

如果的一个final成员是编译时常量，它应该是静态（static）的，从而节省内存。

* **Final类**

如果一个类只含有私有（private）成员，它应该是一个final类。

* **For循环必须使用大括号**

不允许使用for循环省略大括号。

* **If Else语句必须使用大括号**

不允许使用if…else…省略大括号。

* **If语句必须使用括号**

不允许使用if判断省略大括号。

* **StringBuffer实例字符**

不要使用诸如StringBuffer sb = new StringBuffer('c'); 的语句。其中'c'会被转换成为ascii码，从而匹配stringbuffer的长度。

* **String的判等不使用==或!=**

不使用==或者!=判断字符串相等。

* **System.println**

不使用System.(out|err).print/println等语句，使用logger代替。

* **While循环必须使用括号**

不允许使用while循环省略大括号。

* **不变体中的无用操作**

对于不可变对象（Immutable object）的操作不会改变对象本身，这种操作会生成一个新对象，由此，若是不处理该操作的结果，是多余或者错误的。比如，string.toLowerCase()不会改变string对象本身，而是新生成了一个string对象。

* **不必要的大小写转换**

在使用toUpperCase/toLowerCase().equals()的场合应该使用equalsIgnoreCase()，后者更快。

* **不要Catch NPE**

不允许catch NullPointerException，这种catch行为可能会隐藏真实的错误，从而造成其他的问题。

* **不要import java.lang**

import java.lang包下的任何类都可以避免，编译时总是会自动添加这些包依赖。

* **不要使用printStackTrace**

不使用printStackTrace语句，使用logger代替。

* **不要使用重复字符串**

多次重复使用相同内容的不同字符串实例可以改进为使用常量，从而节省内存及对象开销，如：

|  |
| --- |
| **public** **class** Foo {  **private** **void** bar() {  buz("Howdy");  buz("Howdy");  buz("Howdy");  buz("Howdy");  }  **private** **void** buz(String x) {  }  } |

可以改进为：

|  |
| --- |
| **public** **class** Foo {  **final** **static** String *HOWDY* = "howdy";  **private** **void** bar() {  buz(*HOWDY*);  buz(*HOWDY*);  buz(*HOWDY*);  buz(*HOWDY*);  }  **private** **void** buz(String x) {  }  } |

* **不要在BigDecimal构造方法中使用Decimal字符串**

如下代码new BigDecimal(.1)得到的数字并不是正好0.1，而是0.1000000000000000055511151231257827021181583404541015625。这是由于0.1是double型，不可能在double中精确表示。因此，BigDecimal的构造方法推荐使用String参数。

* **不要引入sun的包**

sun的包是为了兼容1.5前的部分java公共库，存在部分性能和内存问题，不推荐使用。

* **不要抛出NullPointerException**

方法中不允许抛出NullPoinerException，因为会没法判断是方法抛出还是由虚拟机抛出，不方便定位问题所在。可以使用IllegalArgumentException代替。

* **不要抛出基础异常类型**

不允许直接抛出RuntimeException, Throwable, Exception, 或 Error这样的异常基类，这种写法太宽泛。推荐使用JumpException代替。

* **不要调用Finalize**

Object.finalize() 是由垃圾回收器调用来确认该对象没有引用的，不要再程序中显式调用。

* **不要在catch块中仅仅抛出包装异常**

catch块中仅仅新包装一个新异常抛出是一件没有意义的事情，同时也增加编写调试和运行的复杂性。

* **低效的StringBuffer**

不要在StringBuffer的构造函数或者append方法中使用非字符型参数。

* **使用ArrayList代替Vector**

ArrayList比Vector高效，而功能完全一致。

* **使用equals比较对象**

除了简单类型用==之外，必须用equlas比较对象。

* **使用Map代替Hashtable**

在没有对同一对象的并发处理时采用Map或HashMap，而不是HashTable。

* **使用StringBuffer.length()**

用 StringBuffer.length()，不要使用 StringBuffer.toString().equals() 或者StringBuffer.toString().length() ==。

* **使用字符索引**

使用String.indexOf(char)来确定一个字符串中是否含有某字符。

* **使用正确的异常日志**

在异常时要输出log，logger的参数应该是错误信息和堆栈日志。

* **修饰符的可见性**

类成员的可见性约束：只有static final成员可以是public的。

* **修饰符顺序**

对类成员的修饰顺序应该是：public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp。

* **关闭资源**

资源类对象如（Connection, Statement, ResultSet）需要在finally代码块中保证关闭。

* **包名**

包名要符合命名规范，正则表达式：^[a-z]+(\.[a-z][a-z0-9]\*)\*$ 。

* **匿名内部类长度**

匿名内部类最长不能超过20行。

* **单一属性**

如果一个类成员域只在一个方法中使用，可以转成方法中的局部对象。

* **命名 - 不要使用$**

禁止在标示符命名中使用$。

* **命名 - 可疑的equals方法名**

不要用不同参数重载equlas方法，如：

|  |
| --- |
| **public** **class** Foo {  **public** **int** equals(Object o) {  // equlas的返回值不应该是int  }  **public** **boolean** equals(String s) {  // equlas的参数应该是Object  }  } |

* **命名 - 可疑的hashCode方法名**

不要用诸如hashcode,Hashcode,HASHCODE的标示符做方法名，容易与hashCode混淆，如：

|  |
| --- |
| **public** **class** Foo {  **public** **int** hashcode() {  // oops, this probably was supposed to be hashCode  }  } |

* **命名 - 可疑的常量属性名**

比如如下的代码：

|  |
| --- |
| **public** **class** Foo {  **double** PI = 3.16; //不应该这样命名  } |

PI如果是一个类成员，就应该小写pi。如果是一个常量，就应该添加final

* **命名 - 方法与内部类名称相同**

不要在一个类中写出一个同名内部类或方法，如：

|  |
| --- |
| **public** **class** MyClass {  // this is bad because it is a method  **public** **void** MyClass() {  }  // this is OK because it is a constructor  **public** MyClass() {  }  } |

* **命名 - 类名称约定**

类名必须满足命名规范。

* **方法的路径复杂度**

方法的路径复杂度定义为方法中的if,while,do,for,?:,catch,switch case, &&, || 所构建成的最小可能执行路径数量。路径数越多说明该方法越复杂，当路径数超过12时应该考虑重构该方法。

* **字符串转化为字符串**

不要对字符串对象调用不必要的toString()方法。

* **安全 - 直接保存数组**

方法接受数组作为参数时应该对其克隆对象进行操作，以限制影像到调用对象。

* **对null进行Equals操作**

不要写出.equlas(null)这样的代码。

* **尾部注释**

尾部注释必须满足注释规范，极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

* **布尔实例**

不需要实例化Boolean对象；可以直接引用Boolean.TRUE, Boolean.FALSE, 或者Boolean.valueOf()。

* **布尔表达式复杂度**

限制嵌套的布尔运算符（&&，| |，^）到指定的深度（默认值<=3）。

* **常量名**

常量需要满足命名规范。

* **幂等操作**

避免幂等操作，比如：

int x = 2;

x = x;

* **异常作为流控制**

异常不应该成为流程控制的一种方式。

* **成员名**

类成员必须满足命名规范。

* **把Arrays当做List使用**

如果需要把Array转成List，不要逐个复制Array成员，使用Array.asList()方法。

* **数组上的类型转换异常**

假设有Collection col

Collection<String> col = new ArrayList();

要将之转换为数组时要写为：

String[] obj = col.toArray(new String[0]);

不要使用强制转换

String[] obj = (String[]) col.toArray();

会抛出ClassCastException异常。

* **整型实例**

在JDK 1.5+中，使用new Integer()会分配新的内存，使用Integer.valueOf()更高效。

* **方法名**

方法名必须满足命名规范。

* **方法（泛型）参数名**

方法泛型名必须满足命名规范。

* **无条件的if语句**

if语句内的条件不可以总是为真。

* **无用的String.valueOf()**

连接字符串的时候不要使用String.valueOf()。

* **无用的修饰符**

在接口（Interface）中的成员域默认是public static final的，方法是public abstract的。接口中的内部类或者内部接口自动是public static的。出于历史原因，对这些成员的修饰可以编译通过，但并不会起作用。

* **无用的局部变量**

如果一个局部变量没有被使用或者赋值，那么它是多余的。

* **无用的私有属性**

如果一个私有域没有被使用或者赋值，那么它是多余的。

* **无用的私有方法**

如果一个私有方法没有被调用，那么它是多余的。

* **未使用的引入**

import了包单是并没有使用。

* **未使用的正式参数**

作为方法的一个参数，但是这个参数没有用到。

* **未实例化类中丢失静态方法**

如果一个类只有私有构造方法但是没有静态实例化方法，这个类无法被使用。

* **松耦合**

尽可能使用接口（如 Set），而不是实现类（如 HashSet）。

* **检查子表达式中是否有赋值操作**

子表达式中不应该有赋值操作，比如String s = Integer.toString(i = 2) 。

* **破坏的null检测**

不要写出object.isNull()这样的方法，调用的时候obj已经是null直接会抛出NullPointerException。使用||或者&&代替之。

* **空Finalizer**

如果finalize方法是空的，就没有必要存在。

* **空if语句**

如果if语句是空的，就没有必要存在。

* **空Switch语句**

如果switch语句是空的，就没有必要存在。

* **空While语句**

如果while语句是空的，就没有必要存在，如果需要时间等待，使用Thread.sleep()。

* **空的Finally块**

如果finally语句是空的，就没有必要存在。

* **空的try块**

如果try语句是空的，就没有必要存在。

* **空的同步块**

如果synchronized语句块是空的，就没有必要存在。

* **空语句**

删除空语句（单行有个;) 。

* **空静态实例化**

空的静态初始化方法可以删除。

* **简化布尔表达式**

简化过于复杂的布尔表达式 。

* **简化布尔返回**

简化过于复杂的布尔返回类型。

* **简化条件**

在instanceof之前没必要检查null，如果对象是null，instanceof会返回false。

* **类（泛型）参数名**

类泛型参数名需要满足命名规范。

* **获取Class的实例**

不要用getClass()来得到类class，用public的.class属性。比如，不要写

Class c = new String().getClass();

应该写成：

Class c = String.class

* **覆盖Finalize**

覆盖的finalize方法不应该有参数，否则很可能是bug。

* **覆盖了equals也会覆盖hashCode**

重写覆盖equlas方法的同时必须重写hashCode方法。

* **避免不用的本地变量**

避免创建不必要的局部变量。

* **避免java包之间的循环依赖**

当很多包形成了一个环（包A > 包B > 包C > 包A，">"意味着依赖）， 这意味着那些包高度耦合，难以在不引用其他所有包的情况下，重用/提取这些包。 这种换可能飞速提升维护应用的需求，也会限制业务变更。 Sonar不知检测包之间的循环依赖，也可以提出打破这些循环的最小影响。 这个规则记录了每个拥有外向依赖源代码文件的违规，目的是为了打破循环。

* **避免使用行注释删除代码**

行注释的代码可以用于偶尔的调试，但是不能留在测试的代码中，因为回头看的时候很容易造成误解。

* **避免数组循环**

不要用循环来复制数组，用System.arrayCopy。

* **谨慎的throws**

不允许抛出另一个声明异常的子类，不运行抛出非受检异常。

* **覆盖类成员域**

检查是否有局部变量覆盖了类成员域的情况。

* **隐藏工具类的构造方法**

工具类是指只有静态方法的类，这样的类不应该有构造函数。

* **静态变量名**

静态变量名必须满足命名规范。

* **非法Throws**

不允许直接throw java.lang.Error 和 java.lang.RuntimeException。

* **魔法数字**

数字直接用在了程序中，而不是设置成常量。