1、【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

**public class** Test {

public static String staticString = “ss”;

}

【正面代码用例】

String str = Test.staticString;

【反面代码用例】

Test test = new Test();

String str = test.staticSring;

2. 【强制】所有的覆写方法，必须加@Override 注解。

说明：getObject()与 get0bject()的问题。一个是字母的 O，一个是数字的 0，加@Override 可以准确判断是否覆盖成功。另外，如果在抽象类中对方法签名进行修改，其实现类会马上编 译报错。

【正面代码用例】

@Override

public class getObject(){}

【反面代码用例】

public class get0bject(){}

3. 【强制】相同参数类型，相同业务含义，才可以使用 Java 的可变参数，避免使用 Object。

说明：可变参数必须放置在参数列表的最后。（提倡同学们尽量不用可变参数编程）

【正面代码用例】

public User getUsers(String type, Integer... ids) {...}

【反面代码用例】

public User getUsers(String type, Integer... ids) {...}

4. 【强制】外部正在调用或者二方库依赖的接口，不允许修改方法签名，避免对接口调用方产生 影响。接口过时必须加@Deprecated 注解，并清晰地说明采用的新接口或者新服务是什么。

@Deprecated 注解:

若某类或某方法加上该注解之后，表示此方法或类不再建议使用，调用时也会出现删除线，但并不代表不能用，只是说，不推荐使用，因为还有更好的方法可以调用。

【正面代码用例】

【反面代码用例】

5. 【强制】不能使用过时的类或方法。 说明：java.net.URLDecoder 中的方法 decode(String encodeStr) 这个方法已经过时，应 该使用双参数 decode(String source, String encode)。接口提供方既然明确是过时接口， 那么有义务同时提供新的接口；作为调用方来说，有义务去考证过时方法的新实现是什么。

@Deprecated 注解使方法过时。

【正面代码用例】

【反面代码用例】

6. 【强制】Object 的 equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals。

说明：推荐使用 java.util.Objects#equals（JDK7 引入的工具类）

**public boolean** equals(Object obj) {  
 **return** (**this** == obj);  
}

【正面代码用例】

"test".equals(object);

【反面代码用例】

object.equals("test");

7. 【强制】所有的相同类型的包装类对象之间值的比较，全部使用 equals 方法比较。 说明：对于 Integer var = ? 在-128 至 127 范围内的赋值，Integer 对象是在IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用==进行 判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑， 推荐使用 equals 方法进行判断。

【正面代码用例】

【反面代码用例】

8. 关于基本数据类型与包装数据类型的使用标准如下：

1） 【强制】所有的 POJO 类属性必须使用包装数据类型。

2） 【强制】RPC 方法的返回值和参数必须使用包装数据类型。

3） 【推荐】所有的局部变量使用基本数据类型。

说明：POJO 类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，任何 NPE 问题，或者入库检查，都由使用者来保证。 正例：数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险。 反例：比如显示成交总额涨跌情况，即正负 x%，x 为基本数据类型，调用的 RPC 服务，调用 不成功时，返回的是默认值，页面显示为 0%，这是不合理的，应该显示成中划线。所以包装 数据类型的 null 值，能够表示额外的信息，如：远程调用失败，异常退出。

【正面代码用例】

【反面代码用例】

9. 【强制】定义 DO/DTO/VO 等 POJO 类时，不要设定任何属性默认值。

反例：POJO 类的 gmtCreate 默认值为 new Date();但是这个属性在数据提取时并没有置入具 体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间。

【正面代码用例】

【反面代码用例】

10. 【强制】序列化类新增属性时，请不要修改 serialVersionUID 字段，避免反序列失败；如 果完全不兼容升级，避免反序列化混乱，那么请修改 serialVersionUID 值。 说明：注意 serialVersionUID 不一致会抛出序列化运行时异常。

【正面代码用例】

【反面代码用例】

11. 【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在 init 方法中。

【正面代码用例】

【反面代码用例】

12. 【强制】POJO 类必须写 toString 方法。

使用 IDE 的中工具：source> generate toString 时，如果继承了另一个 POJO 类，注意在前面加一下 super.toString。

说明：在方法执行抛出异常时，可以直接调用 POJO 的 toString()方法打印其属性值，便于排 查问题。

【正面代码用例】

【反面代码用例】

13. 【推荐】使用索引访问用 String 的 split 方法得到的数组时，需做最后一个分隔符后有无 内容的检查，否则会有抛 IndexOutOfBoundsException 的风险。

说明： String str = "a,b,c,,"; String[] ary = str.split(","); // 预期大于 3，结果是 3 System.out.println(ary.length);

【正面代码用例】

【反面代码用例】

14. 【推荐】当一个类有多个构造方法，或者多个同名方法，这些方法应该按顺序放置在一起， 便于阅读，此条规则优先于第 15 条规则。

【正面代码用例】

【反面代码用例】

15. 【推荐】 类内方法定义顺序依次是：公有方法或保护方法 > 私有方法 > getter/setter 方法。

说明：公有方法是类的调用者和维护者最关心的方法，首屏展示最好；保护方法虽然只是子类 关心，也可能是“模板设计模式”下的核心方法；而私有方法外部一般不需要特别关心，是一个 黑盒实现；因为承载的信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体 最后。

【正面代码用例】

【反面代码用例】

16. 【推荐】setter 方法中，参数名称与类成员变量名称一致，this.成员名 = 参数名。在 getter/setter 方法中，不要增加业务逻辑，增加排查问题的难度。

反例： public Integer getData() { if (true) { return this.data + 100; } else { return this.data - 100; } }

【正面代码用例】

【反面代码用例】

17. 【推荐】循环体内，字符串的连接方式，使用 StringBuilder 的 append 方法进行扩展。

说明：反编译出的字节码文件显示每次循环都会 new 出一个 StringBuilder 对象，然后进行 append 操作，最后通过 toString 方法返回 String 对象，造成内存资源浪费。 反例： String str = "start"; for (int i = 0; i < 100; i++) { str = str + "hello"; }

【正面代码用例】

【反面代码用例】

18. 【推荐】final 可以声明类、成员变量、方法、以及本地变量，下列情况使用 final 关键字： 1） 不允许被继承的类，如：String 类。 2） 不允许修改引用的域对象，如：POJO 类的域变量。 3） 不允许被重写的方法，如：POJO 类的 setter 方法。 4） 不允许运行过程中重新赋值的局部变量。 5） 避免上下文重复使用一个变量，使用 final 描述可以强制重新定义一个变量，方便更好 地进行重构。

【正面代码用例】

【反面代码用例】

19. 【推荐】慎用 Object 的 clone 方法来拷贝对象。 说明：对象的 clone 方法默认是浅拷贝，若想实现深拷贝需要重写 clone 方法实现属性对象 的拷贝。

【正面代码用例】

【反面代码用例】

20. 【推荐】类成员与方法访问控制从严： 1） 如果不允许外部直接通过 new 来创建对象，那么构造方法必须是 private。 2） 工具类不允许有 public 或 default 构造方法。 3） 类非 static 成员变量并且与子类共享，必须是 protected。 4） 类非 static 成员变量并且仅在本类使用，必须是 private。 5） 类 static 成员变量如果仅在本类使用，必须是 private。 若是 static 成员变量，必须考虑是否为 final。 7） 类成员方法只供类内部调用，必须是 private。 8） 类成员方法只对继承类公开，那么限制为 protected。 说明：任何类、方法、参数、变量，严控访问范围。过于宽泛的访问范围，不利于模块解耦。 思考：如果是一个 private 的方法，想删除就删除，可是一个 public 的 service 方法，或者 一个 public 的成员变量，删除一下，不得手心冒点汗吗？变量像自己的小孩，尽量在自己的 视线内，变量作用域太大，无限制的到处跑，那么你会担心的。

【正面代码用例】

【反面代码用例】

21.【强制】关于 hashCode 和 equals 的处理，遵循如下规则： 1） 只要重写 equals，就必须重写 hashCode。 2） 因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的 对象必须重写这两个方法。 3） 如果自定义对象做为 Map 的键，那么必须重写 hashCode 和 equals。 说明：String 重写了 hashCode 和 equals 方法，所以我们可以非常愉快地使用 String 对象 作为 key 来使用。

【正面代码用例】

【反面代码用例】

22. 【强制】 ArrayList的subList结果不可强转成ArrayList，否则会抛出ClassCastException 异常，即 java.util.RandomAccessSubList cannot be cast to java.util.ArrayList. 说明：subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList ，而是 ArrayList 的一个视图，对于 SubList 子列表的所有操作最终会反映到原列表上。

【正面代码用例】

【反面代码用例】

23. 【强制】在 subList 场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、 删除均会产生 ConcurrentModificationException 异常。

【正面代码用例】

【反面代码用例】

24. 【强制】使用集合转数组的方法，必须使用集合的 toArray(T[] array)，传入的是类型完全 一样的数组，大小就是 list.size()。 说明：使用 toArray 带参方法，入参分配的数组空间不够大时，toArray 方法内部将重新分配 内存空间，并返回新数组地址；如果数组元素大于实际所需，下标为[ list.size() ]的数组 元素将被置为 null，其它数组元素保持原值，因此最好将方法入参数组大小定义与集合元素 个数一致。 正例： List<String> list = new ArrayList<String>(2); list.add("guan"); list.add("bao"); String[] array = new String[list.size()]; array = list.toArray(array);

反例：直接使用 toArray 无参方法存在问题，此方法返回值只能是 Object[]类，若强转其它 类型数组将出现 ClassCastException 错误。

【正面代码用例】

【反面代码用例】

25. 【强制】使用工具类 Arrays.asList()把数组转换成集合时，不能使用其修改集合相关的方 法，它的 add/remove/clear 方法会抛出 UnsupportedOperationException 异常。 说明：asList 的返回对象是一个 Arrays 内部类，并没有实现集合的修改方法。Arrays.asList 体现的是适配器模式，只是转换接口，后台的数据仍是数组。

String[] str = new String[] { "you", "wu" }; List list = Arrays.asList(str); 第一种情况：list.add("yangguanbao"); 运行时异常。 第二种情况：str[0] = "gujin"; 那么 list.get(0)也会随之修改。

【正面代码用例】

【反面代码用例】

26. 【强制】泛型通配符<? extends T>来接收返回的数据，此写法的泛型集合不能使用 add 方 法，而<? super T>不能使用 get 方法，做为接口调用赋值时易出错。 说明：扩展说一下 PECS(Producer Extends Consumer Super)原则：第一、频繁往外读取内 容的，适合用<? extends T>。第二、经常往里插入的，适合用<? super T>。

【正面代码用例】

【反面代码用例】

27. 【强制】不要在 foreach 循环里进行元素的 remove/add 操作。remove 元素请使用 Iterator 方式，如果并发操作，需要对 Iterator 对象加锁。 正例： Iterator<String> iterator = list.iterator(); while (iterator.hasNext()) { String item = iterator.next(); if (删除元素的条件) { iterator.remove(); } } 反例： List<String> list = new ArrayList<String>(); list.add("1"); list.add("2"); for (String item : list) { if ("1".equals(item)) { list.remove(item); } } 说明：以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的 结果吗？

【正面代码用例】

【反面代码用例】

28. 【强制】 在 JDK7 版本及以上，Comparator 要满足如下三个条件，不然 Arrays.sort， Collections.sort 会报 IllegalArgumentException 异常。 说明：三个条件如下 1） x，y 的比较结果和 y，x 的比较结果相反。

2） x>y，y>z，则 x>z。 3） x=y，则 x，z 比较结果和 y，z 比较结果相同。 反例：下例中没有处理相等的情况，实际使用中可能会出现异常： new Comparator<Student>() { @Override public int compare(Student o1, Student o2) { return o1.getId() > o2.getId() ? 1 : -1; } };

【正面代码用例】

【反面代码用例】

29. 【推荐】集合初始化时，指定集合初始值大小。 说明：HashMap 使用 HashMap(int initialCapacity) 初始化， 正例：initialCapacity = (需要存储的元素个数 / 负载因子) + 1。注意负载因子（即loader factor）默认为 0.75，如果暂时无法确定初始值大小，请设置为 16（即默认值）。 反例：HashMap 需要放置 1024 个元素，由于没有设置容量初始大小，随着元素不断增加，容 量 7 次被迫扩大，resize 需要重建 hash 表，严重影响性能。

【正面代码用例】

【反面代码用例】

30. 【推荐】使用 entrySet 遍历 Map 类集合 KV，而不是 keySet 方式进行遍历。 说明：keySet 其实是遍历了 2 次，一次是转为 Iterator 对象，另一次是从 hashMap 中取出 key 所对应的 value。而 entrySet 只是遍历了一次就把 key 和 value 都放到了 entry 中，效 率更高。如果是 JDK8，使用 Map.foreach 方法。 正例：values()返回的是 V 值集合，是一个 list 集合对象；keySet()返回的是 K 值集合，是 一个 Set 集合对象；entrySet()返回的是 K-V 值组合集合。

【正面代码用例】

【反面代码用例】

31. 【推荐】高度注意 Map 类集合 K/V 能不能存储 null 值的情况，如下表格：

集合类 Key Value Super 说明

Hashtable 不允许为 null 不允许为 null Dictionary 线程安全

ConcurrentHashMap 不允许为 null 不允许为 null AbstractMap 锁分段技术（JDK8:CAS）

TreeMap 不允许为 null 允许为 null AbstractMap 线程不安全

HashMap 允许为 null 允许为 null AbstractMap 线程不安全

反例： 由于 HashMap 的干扰，很多人认为 ConcurrentHashMap 是可以置入 null 值，而事实上， 存储 null 值时会抛出 NPE 异常。

【正面代码用例】

【反面代码用例】

32. 【参考】合理利用好集合的有序性(sort)和稳定性(order)，避免集合的无序性(unsort)和 不稳定性(unorder)带来的负面影响。 说明：有序性是指遍历的结果是按某种比较规则依次排列的。稳定性指集合每次遍历的元素次 序是一定的。如：ArrayList 是 order/unsort；HashMap 是 unorder/unsort；TreeSet 是 order/sort。

【正面代码用例】

【反面代码用例】

33. 【参考】利用 Set 元素唯一的特性，可以快速对一个集合进行去重操作，避免使用 List 的 contains 方法进行遍历、对比、去重操作。

【强制】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。 说明：资源驱动类、工具类、单例工厂类都需要注意。

【正面代码用例】

【反面代码用例】

34. 【强制】创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。 正例： public class TimerTaskThread extends Thread { public TimerTaskThread() { super.setName("TimerTaskThread"); ... }

【正面代码用例】

【反面代码用例】

35. 【强制】线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。 说明：使用线程池的好处是减少在创建和销毁线程上所花的时间以及系统资源的开销，解决资

源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者 “过度切换”的问题。

【正面代码用例】

【反面代码用例】

36. 【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样 的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。 说明：Executors 返回的线程池对象的弊端如下： 1）FixedThreadPool 和 SingleThreadPool: 允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。 2）CachedThreadPool 和 ScheduledThreadPool: 允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM。

【正面代码用例】

【反面代码用例】

37. 【强制】SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为 static，必须加锁，或者使用 DateUtils 工具类。 正例：注意线程安全，使用 DateUtils。亦推荐如下处理： private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() { @Override protected DateFormat initialValue() { return new SimpleDateFormat("yyyy-MM-dd"); } }; 说明：如果是 JDK8 的应用，可以使用 Instant 代替 Date，LocalDateTime 代替 Calendar， DateTimeFormatter 代替 SimpleDateFormat，官方给出的解释：simple beautiful strong immutable thread-safe。

【正面代码用例】

【反面代码用例】

38. 【强制】高并发时，同步调用应该去考量锁的性能损耗。能用无锁数据结构，就不要用锁；能 锁区块，就不要锁整个方法体；能用对象锁，就不要用类锁。 说明：尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用 RPC 方法。

【正面代码用例】

【反面代码用例】

39. 【强制】对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会造 成死锁。 说明：线程一需要对表 A、B、C 依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序 也必须是 A、B、C，否则可能出现死锁。

【正面代码用例】

【反面代码用例】

40. 【强制】并发修改同一记录时，避免更新丢失，需要加锁。要么在应用层加锁，要么在缓存加 锁，要么在数据库层使用乐观锁，使用 version 作为更新依据。 说明：如果每次访问冲突概率小于 20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次 数不得小于 3 次。

【正面代码用例】

【反面代码用例】

41. 【强制】多线程并行处理定时任务时，Timer 运行多个 TimeTask 时，只要其中之一没有捕获 抛出的异常，其它任务便会自动终止运行，使用 ScheduledExecutorService 则没有这个问题。

【正面代码用例】

【反面代码用例】

42. 【推荐】使用 CountDownLatch 进行异步转同步操作，每个线程退出前必须调用 countDown 方法，线程执行代码注意 catch 异常，确保 countDown 方法被执行到，避免主线程无法执行 至 await 方法，直到超时才返回结果。 说明：注意，子线程抛出异常堆栈，不能在主线程 try-catch 到。

【正面代码用例】

【反面代码用例】

43. 【推荐】避免 Random 实例被多线程使用，虽然共享该实例是线程安全的，但会因竞争同一 seed 导致的性能下降。

说明：Random 实例包括 java.util.Random 的实例或者 Math.random()的方式。 正例：在 JDK7 之后，可以直接使用 API ThreadLocalRandom，而在 JDK7 之前，需要编码保 证每个线程持有一个实例。

【正面代码用例】

【反面代码用例】

44. 【推荐】在并发场景下，通过双重检查锁（double-checked locking）实现延迟初始化的优 化问题隐患(可参考 The "Double-Checked Locking is Broken" Declaration)，推荐解 决方案中较为简单一种（适用于 JDK5 及以上版本），将目标属性声明为 volatile 型。 反例： class Singleton { private Helper helper = null; public Helper getHelper() { if (helper == null) synchronized(this) { if (helper == null) helper = new Helper(); } return helper; } // other methods and fields... }

【正面代码用例】

【反面代码用例】

45. 【参考】volatile 解决多线程内存不可见问题。对于一写多读，是可以解决变量同步问题， 但是如果多写，同样无法解决线程安全问题。如果是 count++操作，使用如下类实现： AtomicInteger count = new AtomicInteger(); count.addAndGet(1); 如果是 JDK8，推 荐使用 LongAdder 对象，比 AtomicLong 性能更好（减少乐观锁的重试次数）。

【正面代码用例】

【反面代码用例】

46. 【参考】 HashMap 在容量不够进行 resize 时由于高并发可能出现死链，导致 CPU 飙升，在 开发过程中可以使用其它数据结构或加锁来规避此风险。

【正面代码用例】

【反面代码用例】

47. 【参考】ThreadLocal 无法解决共享对象的更新问题，ThreadLocal 对象建议使用 static 修饰。这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享 此静态变量 ，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只 要是这个线程内定义的)都可以操控这个变量。

【正面代码用例】

【反面代码用例】

48. 【强制】在一个 switch 块内，每个 case 要么通过 break/return 等来终止，要么注释说明程 序将继续执行到哪一个 case 为止；在一个 switch 块内，都必须包含一个 default 语句并且 放在最后，即使它什么代码也没有。

【正面代码用例】

【反面代码用例】

49. 【强制】在 if/else/for/while/do 语句中必须使用大括号。即使只有一行代码，避免采用 单行的编码方式：if (condition) statements;

【正面代码用例】

【反面代码用例】

50. 【推荐】表达异常的分支时，少用 if-else 方式，这种方式可以改写成：

if (condition) { ... return obj; } // 接着写 else 的业务逻辑代码; 说明：如果非得使用 if()...else if()...else...方式表达逻辑，【强制】避免后续代码维 护困难，请勿超过 3 层。 正例：超过 3 层的 if-else 的逻辑判断代码可以使用卫语句、策略模式、状态模式等来实现， 其中卫语句示例如下：

public void today() { if (isBusy()) { System.out.println(“change time.”); return; }

if (isFree()) { System.out.println(“go to travel.”); return; }

System.out.println(“stay at home to learn Alibaba Java Coding Guidelines.”); return;

【正面代码用例】

【反面代码用例】