1. 不能在 finally 块中使用 return，finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句

分析：因为 finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句，也就是说 try 块中的 return 值会先保存起来，然后执行完 finally 中的代码后，才会把 try 块中的 return 值返回，所以 finally 中的代码逻辑是不会影响 try 块中的 return 值的。但如果在 finally 中使用 return 了就会导致 try 块中的代码得不到执行而无法返回正确的结果。

2. 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。

分析：静态变量或者静态函数，直接用类来调用。

3. 所有的枚举类型字段必须要有注释，说明每个数据项的用途。

分析：便于代码的阅读以及维护

4. 所有的覆写方法，必须加@Override 注解。

分析：如果不添加改注解，子类会认为是子类内部新建的函数，而不是对父类的方法进行重写。

5. 避免用 Apache Beanutils 进行属性的 copy。

分析：Apache BeanUtils 性能较差，推荐使用 Spring BeanUtils 或者 Cglib BeanCopier 来代替。

6. 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。

分析：Arrays.asList()返回的 arraylist 是静态内部类，继承抽象类 AbstractList，该类没有具体实现以上方法。

7. 线程资源建议通过线程池提供，不建议在应用中自行显式创建线程。

分析：使用线程池技术，可以减少创建和销毁线程的次数，让每个线程可以多次使用,可根据系统情况调整执行的线程数量，防止消耗过多内存,节省系统资源。

线程池的好处是减少在创建和销毁线程上所消耗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。

8. 在subList场景中，高度注意对原列表的修改，会导致子列表的遍历、增加、删除均产生 ConcurrentModificationException异常。

分析： Arraylist 的 subList 返回的是 Arraylist 的内部类 SubList，是 Arraylist 的一个视图，对 subList 的而修改最终会反映到元列表上。

9. 循环体内，字符串的联接方式，使用 StringBuilder 的 append 方法进行扩展。

分析：非多线程环境下，推荐优先使用 StringBuilder，效率高，多线程环境下，推荐使用 StringBuffer，线程安全。

10. 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。

分析：foreach 底层实现还是通过迭代器 Iterator 的方式，Iterator 在迭代的时候，不能使用集合自身的方法进行添加删除操作，可以使用迭代器自身的 remove 方法进行移除操作。

11. 获取当前毫秒数：System.currentTimeMillis(); 而不是 new Date().getTime();

分析：new Date()所做的事情其实就是调用了 System.currentTimeMillis()。如果仅仅是需要或者毫秒数，那么完全可以使用 System.currentTimeMillis()去代替 new Date()，效率上会高一点

12. 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。

分析：逻辑性判断问题，良好的编码风格可以大大的提高代码的可读性。
13. 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。

分析：SimpleDateFormat 是非线程安全的工具类，在多线程环境中需要加同步机制，譬如 synchronized。推荐使用 ThreadLocal 来限制 SimpleDateFormat 只能在线程内共享，这样就避免了多线程导致的线程安全问题，此外，如果在 jdk1.8 环境下，推荐使用 DateTimeFormatter 工具类。

14. 常量命名应该全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长

分析：良好的编码习惯。
15. 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。

分析：对象 object 如果为空的话，不进行判空的话，直接调用 equal 方法，会报空指针，所以需要用确定值的对象来调用 equal 方法。

16. 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 ScheduledThreadPoolExecutor 则没有这个问题。

解析： Timer 线程不会捕获异常，所以 TimerTask 抛出的未检查的异常会终止 timer 线程。如果 Timer 线程中存在多个计划任务，其中一个计划任务抛出未检查的异常，则会引起整个 Timer 线程结束，从而导致其他计划任务无法得到继续执行。Timer 是单线程，如果某个任务很耗时，可能会影响其他计划任务的执行。

ScheduledThreadPoolExecutor 线程会捕获任务重的异常，即使多个计划任务中存在某几个计划任务为捕获异常的情况，也不会影响 ScheduledThreadPoolExecutor 总线程的工作，不会影响其他计划任务的继续执行，ScheduledThreadPoolExecutor 是线程池，如任务数过多或某些任务执行时间较长，可自动分配更多的线程来执行计划任务。
17. 所有编程相关的命名均不能以下划线或美元符号开始

分析：良好的编码习惯。
18. ArrayList 的 subList 结果不可强转成 ArrayList，否则会抛出 ClassCastException 异常

分析：因为函数 subList 返回了一个 SubList，这个类是 ArrayList 中的一个内部类，他们之间并没有继承关系，故无法直接进行强制类型转换。
19. Map/Set 的 key 为自定义对象时，必须重写 hashCode 和 equals。

分析：Map/Set 的 key 值唯一，对于 key 值相等的判断，是判断 hashcode 值以及 equals 方法，hashcode 值默认的计算方式是根据对象的地址值计算的，而不同的对象在内存中的地址值肯定不同，所以会存在逻辑上应该认定为同一 key 值，实际在 Map/Set 会判定为两个 key。
20. 所有的包装类对象之间值的比较，全部使用 equals 方法比较。

分析：对于 Integer 对象来说，初始值在-128 ~ 127 时会将对象放入缓存池（IntegerCache.cache），下次调用相同的值将直接复用。在该区间的 Integer 对象可直接进行判读，该区间以外的对象在 Heap 上产生，不会进行复用。所以推荐使用 equals 方法进行判断

21. 创建线程或线程池时请指定有意义的线程名称，并且日志中输出线程名称的关键字，方便出错时的问题回溯。

分析：良好的编码习惯，有利于出现代码问题之后，问题的快速定位。

例如：自定义线程工厂，并且根据外部特征进行分组，比如，来自同一机房的调用，把机房编号赋值给whatFeaturOfGroup

```
public class UserThreadFactory implements ThreadFactory {
private final String namePrefix;
private final AtomicInteger nextId = new AtomicInteger(1);
// 定义线程组名称，在jstack问题排查时，非常有帮助
UserThreadFactory(String whatFeaturOfGroup) {
namePrefix = "From UserThreadFactory's " + whatFeaturOfGroup + "-Worker-";
}
@Override
public Thread newThread(Runnable task) {
String name = namePrefix + nextId.getAndIncrement();
Thread thread = new Thread(null, task, name, 0, false);
System.out.println(thread.getName());
return thread;
}

}
```

22.if 判断的异常分支处理，表达异常的分支时，少用 if-else 方式。

```
f (condition) {
...
return obj;
}
// 接着写else的业务逻辑代码;
```

说明：如果非使用if()...else if()...else...方式表达逻辑，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 findBoyfriend (Man man){
if (man.isUgly()) {
System.out.println("本姑娘是外貌协会的资深会员");
return;
}
if (man.isPoor()) {
System.out.println("贫贱夫妻百事哀");
return;
}
if (man.isBadTemper()) {
```

```
System.out.println("银河有多远，你就给我滚多远");
return;
}
System.out.println("可以先交往一段时间看看");

} //接着写 else 的业务逻辑代码.
```

23. 在 if/else/for/while/do 语句中必须使用大括号，即使只有一行代码，避免使用下面的形式：if (condition) statements;

分析：良好的编码习惯，便于代码的维护以及提高代码阅读标准。

24. 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()

分析：ArrayList 提供了一个将 List 转为数组的一个非常方便的方法 toArray。toArray 有两个重载的方法，list.toArray();和 list.toArray(T[] a);对于第一个重载方法，是将 list 直接转为 Object[] 数组，第二种方法是将 list 转化为你所需要类型的数组，当然我们用的时候会转化为与 list 内容相同的类型。

第一种方式，如果 list 中存在的 String 类型，很多人会这样做类型转化，String[] array= (String[]) list.toArray()，此时会报 java.lang.ClassCastException，原因 java 中的强制类型转换只是针对单个对象的，想要偷懒将整个数组转换成另外一种类型的数组是不行的，因此需要遍历整个数组，依次做类型转换，如下

```
Object[] arr = list.toArray();
for (int i = 0; i < arr.length; i++) {
    String e = (String) arr[i];
    System.out.println(e);
}
```

第二种方式比较简单，更加方便，直接在调用函数toArray的时候，制定了转换的类型，如下。

```
String[] array =new String[list.size()];
list.toArray(array);
```

25. 异常类命名使用Exception结尾

分析：良好的命名习惯，便于代码维护。

26. 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。

分析：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。

CachedThreadPool： 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。

27. 返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE

分析：例如下面函数

```
public int f(){
        return Integer 对象
        }
```

如果为 null，自动解箱抛 NPE。

出现场景：

1. 数据库的查询结果可能为 null
2. 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null
3. 远程调用返回对象时，一律要求进行空指针判断，防止 NPE
4. 级联调用 obj.getA().getB().getC();一连串调用，易产生 NPE

28. 不允许任何魔法值（即未经定义的常量）直接出现在代码中。

分析：良好的编码习惯，增加代码的可读性，便于代码的后续维护。

29.

所有的类都必须添加创建者和创建日期。

```
/**
* @author WeiWenTao
* @date 2016/10/31
*/
```

分析：便于后续代码的维护以及回溯。

30. 测试类命名以它要测试的类的名称开始，以 Test 结尾

分析：良好的编码习惯，增加代码的可读性，便于代码的维护。

31. 抽象类命名使用 Abstract 或 **Base** 开头

分析：良好的编码习惯，增加代码的可读性，便于代码的维护。

32. 方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase，必须遵从形式

分析：良好的编码习惯，增加代码的可读性，便于代码的维护。

33. long 或者 Long 初始赋值时，必须使用大写的 L，不能是小写的 l，小写容易跟数字 1 混淆，造成误解。

分析：Long a = 2l; 写的是数字的 21，还是 Long 型的 2

34. 在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。

分析：Pattern 要定义为 static final 静态变量，以避免执行多次预编译。

```
// 没有使用预编译

private void func(...) {
    if (Pattern.matches(regexRule, content)) {
        ...
    }
```

```
}

// 多次预编译

private void func(...) {
    Pattern pattern = Pattern.compile(regexRule);
    Matcher m = pattern.matcher(content);
    if (m.matches()) {
        ...
    }

//预编译

private static final Pattern pattern = Pattern.compile(regexRule);
private void func(...) {
    Matcher m = pattern.matcher(content);
    if (m.matches()) {
        ...
    }
```

35. 增加java工程包名的校验，包名前两部分必须为com.cfets,第三部分为工程名称，例如com.cfets.cbs.*
分析：目前中汇的所有项目的包名都是这样的命名格式。

36. 对开发者的注释进行校验，必须按照以下格式添加注释。
start by 修改者-日期-版本号-环境-缺陷号或者需求号-详细内容，
end by 修改者-日期-版本号-环境-缺陷号或者需求号-详细内容.
例如//start by zhangsan-20181121-V1.14.11-ST-XQTM1811060002-调整利率上下限期权市场全市场波动率收盘价导入逻辑分析
//end by zhangsan-20181121-V1.14.11-ST-XQTM1811060002-调整利率上下限期权市场全市场波动率收盘价导入逻辑分析

Commit 提交的标题为：日期|V1.0.0|JiraID|标题内容

例如：

20200602|V2.9.8.5|RMB-2046|辅助交易-WEB-调整利率上下限期权市场全市场波动率收盘价导入逻辑
分析：便于代码后续的维护以及问题的追踪排查。

37. 禁止使用“select *”关键字:
例如
```
sqlString = "select * "
+ " from dual4 ";
+ " where ..... ";
this.getHibernateTemplate().find("select * from BidMstr bid where.....)
this.executeSQL("select * from MMKR_GRP where....）
```
分析：select * 全表扫描，效率比较低。

38. 在一个 try 块内，都必须包含一个 catch 语句，即使它什么代码也没有

分析：良好的编码习惯

39. sql 语句 预编译检测，不允许使用 Statement 对象

分析：推荐使用 preparedStatment 对象，可以防止 sql 注入，预编译，提高运行效率。

40. 避免多线程环境使用静态的 HashMap 对象,防止被多线程调用,导致死锁,替代使用 ConcurrentHashMap。

分析：因为多个线程对同一个 hashmap 进行 put 操作的时候，可能会在某一个元素的位置形成循环链表，当一个进程对该元素 get()操作的时候，会出现死循环，线程表现就是夯住，不进行后面的代码的处理。

41. BigDecimal 对象避免使用 equals 函数。

分析：由于 Bigdecimal 对象与 equals 方法进行了重写，会判断小数点后面的位数，如果小数点位数不等，直接 rerurn false。

例如 string 类型"1.00000"和"1.00"，equals 方法为 false, compareTo 方法为 0.

42 日期格式化字符串应注意使用小写'y'表示当天所在的年，大写'Y'代表 week，YYYYMMdd

分析：日期格式化时，yyyy表示当天所在的年，而大写的YYYY代表是week in which year（JDK7之后引入的概念），意思是当天所在的周属于的年份，一周从周日开始，周六结束，只要本周跨年，返回的YYYY就是下一年。

表示日期和时间的格式如下所示： new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

43. IF 语句块缺少 ELSE 语句块处理，特殊判断异常的情况除外。

分析：if 语句需要写 else 分支，if 中如果添加异常判断直接返回的场景除外，并且需要在每个分支写明注释标注业务逻辑。

44. SimpleDataFormat 在多线程环境下，可能导致线程安全问题

分析：多线程环境下，需要加同步机制判断。

45.if 判断条件中&&和||同时不要超过 3 个，如果超过的话，需要重新起一个 if,嵌套判断。

分析：代码判断逻辑更加清晰。

46.switch 语句中，每一个 case 都必须有 return 或者 break.

分析：良好的编码习惯，便于代码的阅读以及维护。

47. Logger 需要被定义成静态 static

分析：new Logger 对象比较耗费资源，需要定义成静态变量。

48.测试类以及 main 函数做单侧时可以写 system.out.print(), 正常的业务逻辑代码里不要写。

分析：良好的编码习惯。

49. 常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长

分析：良好的编码习惯。 MAX_STOCK_COUNT / CACHE_EXPIRED_TIME

50. 浮点数之间的等值判断，基本数据类型不能用==来比较

分析：浮点数采用"尾数+阶码"的编码方式，类似于科学计数法的"有效数字+指数"的表示方式。二进制无法精确表示大部分的十进制小数

推荐：

使用BigDecimal来定义值，再进行浮点数的运算操作。

```
BigDecimal a = new BigDecimal("1.0");
BigDecimal b = new BigDecimal("0.9");
BigDecimal c = new BigDecimal("0.8");
BigDecimal x = a.subtract(b);
BigDecimal y = b.subtract(c);
if (x.equals(y)) {
System.out.println("true");
}
```

51. 禁止使用构造方法BigDecimal(double)的方式把double值转化为BigDecimal对象。
分析： BigDecimal(double)存在精度损失风险，在精确计算或值比较的场景中可能会导致业务逻辑异常。
如：BigDecimal g = new BigDecimal(0.1f); 实际的存储值为：0.10000000149
正例：优先推荐入参为String的构造方法，或使用BigDecimal的valueOf方法，此方法内部其实执行了Double的toString，而Double的toString按double的实际能表达的精度对尾数进行了截断。

```
BigDecimal recommend1 = new BigDecimal("0.1");
BigDecimal recommend2 = BigDecimal.valueOf(0.1);
```

52. 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。
分析：注意serialVersionUID不一致会抛出序列化运行时异常

53. 判断所有集合内部的元素是否为空，使用isEmpty()方法，而不是size()==0的方式。
分析：前者的时间复杂度为O(1)，而且可读性更好。

```
Map<String, Object> map = new HashMap<>();
if(map.isEmpty()) {
System.out.println("no element in this map.");
}
```

54.
使用Map的方法keySet()/values()/entrySet()返回集合对象时，不可以对其进行添加元素操作，否则会抛出UnsupportedOperationException异常。
分析：keySet()/values()/entrySet() 只做遍历集合使用。
55.Collections类返回的对象，如：emptyList()/singletonList()等都是immutable list，不可对其进行添加或者删除元素的操作。

56.必须回收自定义的ThreadLocal变量，尤其在线程池场景下，线程经常会被复用，如果不清理自定义的ThreadLocal变量，可能会影响后续业务逻辑和造成内存泄露等问题。尽量在代理中使用try-finally块进行回收。 正例：

```
objectThreadLocal.set(userInfo);
try {
// ...
} finally {
objectThreadLocal.remove();
}
```

57.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。
分析：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。
58.
当switch括号内的变量类型为String并且此变量为外部参数时，必须先进行null判断。
反例：如下的代码输出是什么？

```
public class SwitchString {
public static void main(String[] args) {
method(null);
}
public static void method(String param) {
switch (param) {
// 肯定不是进入这里
case "sth":
System.out.println("it's sth");
break;
// 也不是进入这里
case "null":
System.out.println("it's null");
break;
// 也不是进入这里
default:
System.out.println("default");
}
}
}
```

59.
catch时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的catch尽可能进行区分异常类型，再做对应的异常处理。 说明：对大段代码进行try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。
正例：用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。