3. 【强制】在使用 java.util.stream.Collectors 类的 toMap()方法转为 Map 集合时，一定要使用含有参数类型为 BinaryOperator，参数名为 mergeFunction 的方法，否则当出现相同 key 值时会抛出 IllegalStateException 异常。

   说明：参数 mergeFunction 的作用是当出现 key 重复时，自定义对 value 的处理策略。

   正例：

   ```
   List<Pair<String, Double>> pairArrayList = new ArrayList<>(3);
   pairArrayList.add(new Pair<>("version", 6.19));
   pairArrayList.add(new Pair<>("version", 10.24));
   pairArrayList.add(new Pair<>("version", 13.14));
   Map<String, Double> map = pairArrayList.stream().collect(
   // 生成的 map 集合中只有一个键值对：{version=13.14}
   Collectors.toMap(Pair::getKey, Pair::getValue, (v1, v2) -> v2));
   ```

   反例：

   ```
   String[] departments = new String[] {"iERP", "iERP", "EIBU"};
   // 抛出 IllegalStateException 异常
   Map<Integer, String> map = Arrays.stream(departments)
       .collect(Collectors.toMap(String::hashCode, str -> str));
   ```

4. 【强制】在使用 java.util.stream.Collectors 类的 toMap()方法转为 Map 集合时，一定要注意当 value 为 null 时会抛 NPE 异常。

   说明：在 java.util.HashMap 的 merge 方法里会进行如下的判断：

   ```
   if (value == null || remappingFunction == null)
       throw new NullPointerException();
   ```

   反例：

   ```
   List<Pair<String, Double>> pairArrayList = new ArrayList<>(2);
   pairArrayList.add(new Pair<>("version1", 4.22));
   pairArrayList.add(new Pair<>("version2", null));
   Map<String, Double> map = pairArrayList.stream().collect(
   // 抛出 NullPointerException 异常
   Collectors.toMap(Pair::getKey, Pair::getValue, (v1, v2) -> v2));
   ```

5. 【强制】ArrayList 的 subList 结果不可强转成 ArrayList，否则会抛出 ClassCastException 异常：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。

6. 【强制】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。

7. 【强制】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。

   反例：如果查询无结果，返回 Collections.emptyList()空集合对象，调用方一旦进行了添加元素的操作，就会触发 UnsupportedOperationException 异常。

8. 【强制】在 subList 场景中，高度注意对父集合元素的增加或删除，均会导致子列表的遍历、增加、删除产生 ConcurrentModificationException 异常。

9. 【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致、长度为 0 的空数组。

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。

正例：

```
List<String> list = new ArrayList<>(2);
list.add("guan");
list.add("bao");
String[] array = list.toArray(new String[0]);
```

说明：使用 toArray 带参方法，数组空间大小的 length，

1）等于 0，动态创建与 size 相同的数组，性能最好。

2）大于 0 但小于 size，重新创建大小等于 size 的数组，增加 GC 负担。

3）等于 size，在高并发情况下，数组创建完成之后，size 正在变大的情况下，负面影响与 2 相同。

4）大于 size，空间浪费，且在 size 处插入 null 值，存在 NPE 隐患。

10. 【强制】在使用 Collection 接口任何实现类的 addAll()方法时，都要对输入的集合参数进行 NPE 判断。

说明：在 ArrayList#addAll 方法的第一行代码即 Object[] a = c.toArray(); 其中 c 为输入集合参数，如果为 null，则直接抛出异常。

11. 【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[] { "yang", "hao" };
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] = "changed"; 也会随之修改，反之亦然。

12. 【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，两者在接口调用赋值的场景中容易出错。

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>

13. 【强制】在无泛型限制定义的集合赋值给泛型限制的集合时，在使用集合元素时，需要进行 instanceof 判断，避免抛出 ClassCastException 异常。

说明：毕竟泛型是在 JDK5 后才出现，考虑到向前兼容，编译器是允许非泛型集合与泛型集合互相赋值。

反例：

```
List<String> generics = null;
List notGenerics = new ArrayList(10);
notGenerics.add(new Object());
notGenerics.add(new Integer(1));
generics = notGenerics;
// 此处抛出 ClassCastException 异常
String string = generics.get(0);
```

14. 【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：

```
List<String> list = new ArrayList<>();
list.add("1");
list.add("2");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String item = iterator.next();
    if (删除元素的条件) {
        iterator.remove();
    }
}
```

反例：

```
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？

15. 【强制】在 JDK7 版本及以上，Comparator 实现类要满足如下三个条件，不然 Arrays.sort，Collections.sort 会抛 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，交换两个对象判断结果并不互反，不符合第一个条件，在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;
    }
};
```

16. 【推荐】集合泛型定义时，在 JDK7 及以上，使用 diamond 语法或全省略。

说明：菱形泛型，即 diamond，直接使用<>来指代前边已经指定的类型。

正例：

```
// diamond 方式，即<>
HashMap<String, String> userCache = new HashMap<>(16);
// 全省略方式
ArrayList<User> users = new ArrayList(10);
```

17. 【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，如果暂时无法确定集合大小，那么指定默认值（16）即可。

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。当放置的集合元素个数达千万级别时，不断扩容会严重影响性能。

18. 【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。

19. 【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。

20. 【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。

21. 【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()进行遍历去重或者判断包含操作。

(七) 并发处理

1. 【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。

2. 【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：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;
    }
}
```

3. 【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：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。

4. 【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：

   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。

5. 【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。

6. 【强制】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();
}
```

7. 【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。

8. 【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。

9. 【强制】在使用阻塞等待获取锁的方式中，必须在 try 代码块之外，并且在加锁方法与 try 代码块之间没有任何可能抛出异常的方法调用，避免加锁成功后，在 finally 中无法解锁。

说明一：如果在 lock 方法与 try 代码块之间的方法调用抛出异常，那么无法解锁，造成其它线程无法成功获取锁。

说明二：如果 lock 方法在 try 代码块之内，可能由于其它方法抛出异常，导致在 finally 代码块中，unlock 对未加锁的对象解锁，它会调用 AQS 的 tryRelease 方法（取决于具体实现类），抛出 IllegalMonitorStateException 异常。

说明三：在 Lock 对象的 lock 方法实现中可能抛出 unchecked 异常，产生的后果与说明二相同。

正例：

```
Lock lock = new XxxLock();
// ...
lock.lock();
try {
    doSomething();
    doOthers();
} finally {
    lock.unlock();
}
```

反例：

```
Lock lock = new XxxLock();
// ...
```

```
try {
    // 如果此处抛出异常，则直接执行 finally 代码块
    doSomething();
    // 无论加锁是否成功，finally 代码块都会执行
    lock.lock();
    doOthers();
} finally {
    lock.unlock();
}
```

10. 【强制】在使用尝试机制来获取锁的方式中，进入业务代码块之前，必须先判断当前线程是否持有锁。锁的释放规则与锁的阻塞等待方式相同。

    说明：Lock 对象的 unlock 方法在执行时，它会调用 AQS 的 tryRelease 方法（取决于具体实现类），如果当前线程不持有锁，则抛出 IllegalMonitorStateException 异常。

    正例：

    ```
    Lock lock = new XxxLock();
    // ...
    boolean isLocked = lock.tryLock();
    if (isLocked) {
        try {
            doSomething();
            doOthers();
        } finally {
            lock.unlock();
        }
    }
    ```

11. 【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。

12. 【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。

13. 【推荐】资金相关的金融敏感信息，使用悲观锁策略。

    说明：乐观锁在获得锁的同时已经完成了更新操作，校验逻辑容易出现漏洞，另外，乐观锁对冲突的解决策略有较复杂的要求，处理不当容易造成系统压力或数据异常，所以资金相关的金融敏感信息不建议使用乐观锁更新。

    正例：悲观锁遵循一锁二判三更新四释放的原则

14. 【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。

15. 【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单独的 Random 实例。

16. 【推荐】通过双重检查锁（double-checked locking）（在并发场景下）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（比如修改 helper 的属性声明为`private volatile Helper helper = null;`）。

    反例：

```
public class LazyInitDemo {
    private Helper helper = null;

    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null) { helper = new Helper(); }
            }
        }
        return helper;
    }
    // other methods and fields...
}
```

17. 【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。

    说明：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）。

18. 【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，在开发过程中注意规避此风险。

19. 【参考】ThreadLocal 对象使用 static 修饰，ThreadLocal 无法解决共享对象的更新问题。

    说明：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。

(八) 控制语句

1. 【强制】在一个 switch 块内，每个 case 要么通过 continue/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。

说明：注意 break 是退出 switch 语句块，而 return 是退出方法体。

2. 【强制】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");
        }
    }
}
```

3. 【强制】在 if/else/for/while/do 语句中必须使用大括号。

说明：即使只有一行代码，禁止不采用大括号的编码方式：if (condition) statements;

4. 【强制】三目运算符 condition? 表达式 1：表达式 2 中，高度注意表达式 1 和 2 在类型对齐时，可能抛出因自动拆箱导致的 NPE 异常。

说明：以下两种场景会触发类型对齐的拆箱操作：

1） 表达式 1 或表达式 2 的值只要有一个是原始类型。

2） 表达式 1 或表达式 2 的值的类型不一致，会强制拆箱升级成表示范围更大的那个类型。

反例：

```
Integer a = 1;
Integer b = 2;
Integer c = null;
Boolean flag = false;
// a*b 的结果是 int 类型，那么 c 会强制拆箱成 int 类型，抛出 NPE 异常
Integer result=(flag? a*b : c);
```

5. 【强制】在高并发场景中，避免使用”等于”判断作为中断或退出的条件。

说明：如果并发控制没有处理好，容易产生等值判断被“击穿”的情况，使用大于或小于的区间判断条件来代替。

反例：判断剩余奖品数量等于 0 时，终止发放奖品，但因为并发处理错误导致奖品数量瞬间变成了负数，这样的话，活动无法终止。

6. 【推荐】当某个方法的代码行数超过 10 行时，return / throw 等中断逻辑的右大括号后加一个空行。

说明：这样做逻辑清晰，有利于代码阅读时重点关注。

7. 【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");
}
```

8. 【推荐】除常用方法（如 getXxx/isXxx）等外，不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明：很多 if 语句内的逻辑表达式相当复杂，与、或、取反混合运算，甚至各种方法纵深调用，理解成本非常高。如果赋值一个非常好理解的布尔变量名字，则是件令人爽心悦目的事情。

正例：

```
// 伪代码如下
final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```

反例：

```
public final void acquire ( long arg){
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) {
        selfInterrupt();
    }
```

11. 【参考】创建索引时避免有如下极端误解：

    1） 索引宁滥勿缺。认为一个查询就需要建一个索引。

    2） 吝啬索引的创建。认为索引会消耗空间、严重拖慢记录的更新以及行的新增速度。

    3） 抵制惟一索引。认为惟一索引一律需要在应用层通过“先查后插”方式解决。

(三) SQL 语句

1. 【强制】不要使用 count(列名)或 count(常量)来替代 count(*)，count(*)是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

   说明：count(*)会统计值为 NULL 的行，而 count(列名)不会统计此列为 NULL 值的行。

2. 【强制】count(distinct col) 计算该列除 NULL 之外的不重复行数，注意 count(distinct col1, col2) 如果其中一列全为 NULL，那么即使另一列有不同的值，也返回为 0。

3. 【强制】当某一列的值全是 NULL 时，count(col)的返回结果为 0，但 sum(col)的返回结果为 NULL，因此使用 sum()时需注意 NPE 问题。

   正例：可以使用如下方式来避免 sum 的 NPE 问题：SELECT IFNULL(SUM(column), 0) FROM table;

4. 【强制】使用 ISNULL()来判断是否为 NULL 值。

   说明：NULL 与任何值的直接比较都为 NULL。

   1） NULL<>NULL 的返回结果是 NULL，而不是 false。

   2） NULL=NULL 的返回结果是 NULL，而不是 true。

   3） NULL<>1 的返回结果是 NULL，而不是 true。

   反例：在 SQL 语句中，如果在 null 前换行，影响可读性。select * from table where column1 is null and column3 is not null; 而`ISNULL(column)`是一个整体，简洁易懂。从性能数据上分析，`ISNULL(column)`执行效率更快一些。

5. 【强制】代码中写分页查询逻辑时，若 count 为 0 应直接返回，避免执行后面的分页语句。

6. 【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

   说明：（概念解释）学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

7. 【强制】禁止使用存储过程，存储过程难以调试和扩展，更没有移植性。

8. 【强制】数据订正（特别是删除或修改记录操作）时，要先 select，避免出现误删除，确认无误才能执行更新语句。