方法内联

## Part1

方法内联：在编译过程中遇到方法调用的时候，将目标方法的方法体纳入编译范围中，并取代原方法调用的优化手段。是一种优化手段。

不仅可以消除调用本身带来的性能开销，还可以进一步触发更多优化。因此可以说是编译过程中最为重要的一环。

以 getter/setter 为例，如果没有方法内联，在调用 getter/setter 时，程序需要保存当前方法的执行位置，创建并压入用于 getter/setter 的栈帧、访问字段、弹出栈帧，最后再恢复当前方法的执行。而当内联了对 getter/setter 的方法调用后，上述操作仅剩字段访问。

### 方法内联的条件

方法内联能够触发更多的优化。通常而言，内联越多，生成代码的执行效率越高。然而，对于及时编译器来说，内联越多，即时编译时间越长，而程序编译到峰值的时刻也被推迟。

内联越多，生成的机器码越长。虚机中，编译生成的机器码会被部署到Code cache中。这个cache是有大小限制的（-XX:ReservedCodeCashSize控制）



因而，即时编译器不会无限制地进行方法内联。

### 内联规则

1. 由-XX:CompileCommand中的inline指令指定的方法，以及由@ForceInline注解的方法（仅限于JDK内部方法），会被强制内联。
2. 由-XX:CompileCommand中的dontinline指令或exclude（表示不编译）指定的方法，以及由@DontInline注解的方法（仅限于JDK内部方法），则始终不会内联
3. 如果调用字节码对应的符号引用未被解析、目标方法所在的类未被初始化，或者目标方法是native方法，都将导致方法调用无法内联。
4. C2不支持内联超过9层的调用（可以通过虚机参数-XX:MaxInlineLevel调整），以及1层的直接递归调用（可通过-XX:MaxRecursiveInlineLevel调整）
5. 即时编译器将根据方法调用指令所在的程序路径热度，目标方法的调用次数以及大小，以及当前IR图的大小来决定方法调用能否被内联。



总的来说，即时编译器中的内联算法更青睐于小方法。

## Part2

第一部分都是静态方法的调用，即时编译器可以轻易地确定唯一的方法。

对于动态绑定的虚方法调用，即时编译器则需要先对虚方法调用进行去虚化（devirtualize），即转换为一个或多个直接调用，然后在进行方法内联。

### 去虚化

1. 完全去虚化（识别虚方法调用的唯一目标方法，从而转换为直接调用的一种优化手段）
   1. 类型推导
   2. 层次分析
2. 条件去虚化（将虚方法调用转换为若干个类型测试以及直接调用的一种优化手段。关键在于找到需要进行比较的类型）