Kotlin的inline内联函数

# 1.方法调用流程

调用一个方法是一个压栈和出栈的过程，调用方法时将栈针压入方法栈，然后执行方法体，方法结束时将栈针出栈，这个压栈和出栈的过程会耗费资源，这个过程中传递形参也会耗费资源。

# 2.为什么需要inline

有些简单的方法会被频繁调用，什么叫简单的方法呢，举个例子：

fun <T> check(lock: Lock, body: () -> T): T {

lock.lock()

try {

return body()

} finally {

lock.unlock()

}

}

这个check方法的方法体中，不会将它的形参再传递给其他方法。我们调用一下check方法：

check(l, {"我是lambda方法体"})//l是一个Lock对象

对于编译器来说，调用check方法就要将参数l和lambda表达式{"我是lambda方法体"}进行传递，还要将check方法进行压栈出栈处理，这个过程就会耗费资源。

如果我们把check方法删除，直接执行check方法的方法体：

l.lock()

try {

return "我是lambda方法体"

} finally {

l.unlock()

}

这样做的效果和调用check方法是一样的，而且不需要压栈出栈了，但是代码是写给人看的，这样写明显产生了代码坏味道，老司机会告诉你，这几行代码需要抽成一个方法，避免多处调用产生冗余代码。于是你就老老实实把这几行代码抽成了check方法，那么如上所述，一旦这个方法被频繁调用，压栈出栈将会带来性能问题。针对这个问题，kotlin引入了inline关键字。我们在check方法前加上inline关键字：

inline fun <T> check(lock: Lock, body: () -> T): T {

lock.lock()

try {

return body()

} finally {

lock.unlock()

}

}

然后我们再调用check方法，编译器就会在编译期帮我们进行优化：

将我们写的代码

check(l, {"我是lambda方法体"})//l是一个Lock对象

换成

l.lock()

try {

return "我是lambda方法体"

} finally {

l.unlock()

}

也就是说inline关键字实际上增加了代码量，但是提升了性能，而且增加的代码量是在编译期执行的，对程序可读性不会造成影响。

**noinline**

如果check方法中的参数需要传递给其他非inline方法：

inline fun <T> check(lock: Lock, body: () -> T): T {

lock.lock()

try {

otherCheck(body)//会报错

return body()

} finally {

lock.unlock()

}

}

fun <T> otherCheck(body: ()-> T){

}

那么调用otherCheck是会报错的，因为check方法中的形参body现在已经inline了，不是一个函数对象了，也就不能作为一个参数传递了，除非在body参数前加上noinline关键字