پوینتر سرگردان در جاوا

نویسنده:روزبه شریف‌نسب

تاریخ:۲۶ فروردین ۱۳۹۸

# سرخط مطلب

در این مطلب قصد داریم نحوه ساخت و چرایی استفاده و جزيیات پشت پرده استفاده از پوینتر در جاوا را با متد unsafe بررسی کنیم و در نهایت پوینتر سرگردان ساخته و رفتار جاوا با آن را نیز مشاهده کنیم.

# چکیده

در جاوا با آمدن ساختار refrence و Object ها و Grabage Collector قدرتمند، متد unsafe مدتیست deprecate شده و با استفاده از آن چندین warning متفاوت دریافت می‌کنید ولی همچنان با متد‌های allocateMemory و freeMemory می‌توان مموری به برنامه تخصیص داد ‌و آزاد کرد و همچنین با متد putAddress مقداری را درون حافظه تخصیص داده شده، ریخت. همچنین در ادامه می‌بینیم که می‌توان در خانه‌ی حافظه پس از free شدن نیز مقدار نسبت داد و آن را بازخوانی کرد. اگرچه اینکار با پرتاب exception همراه است ولی کد تا پایان اجرا می‌شود!

# مقدمه

یکی از امکانات مهم زبان‌های برنامه‌نویسی سطح پایین و متوسط(مثل C و ++C ) امکان استفاده مستقیم از حافظه با پوینتر است که این امکان با توجه به خطرات ‌آن و مواردی مثل danling pointer در زبان‌های جدید حذف شده‌است ولی با روش پیش‌رو در جاوا می‌توان هم پوینتر ساخت و هم آن را تبدیل به پوینتر سرگردان نمود، در ادامه به نحوه کار و دلیلی که نباید از این روش استفاده کنیم شرح داده می‌شود.

# بخش اول – استفاده از کتاب‌خانه‌های لازم

با توجه به اینکه قرار است در این روش از متدهای غیرمعمول و deprecate شده‌ی زبان جاوا استفاده کنیم طبیعی است که این کلاس‌ها و متد‌هایشان در package های اصلی جاوا همچون java.lang نیستند و نیاز داریم کتاب‌خانه‌های مورد استفاده را import کنیم.

برای اینکار به ۲ کتاب ‌خانه‌ی java.lang.reflect.Field و sun.misc.Unsafe نیاز داریم. بحث reflect.field مربوط به مبحث reflection است که توضیح آن از حوصله این مطلب خارج است و کتاب‌خانه دوم sun.misc.unsafe است که متد‌های کار مستقیم با حافظه در این کلاس پیاده‌سازی شده‌‌اند. توجه کنیم که تنها با import کردن sun.misc.unsafe هشدارهای مربوط به deprecate بودن این کلاس دریافت می‌شود ولی این مانع از ادامه کار نیست.

# بخش دوم – ساخت پوینتر

در این قسمت یک field ساخته و نوع آن را Unsafe قرار می‌دهیم

Field field = sun.misc.Unsafe.class.getDeclaredField("theUnsafe")

در قدم بعدی با استفاده از متد setAccessible، چک‌های سطح دسترسی را غیرفعال می‌کنیم تا با خطای IllegalAccessException مواجه نشویم.

field.setAccessible(true)

در نهایت خروجی field را گرفته و به تایپ sun.misc.Unsafe تبدیل (cast) کرده و در یک Unsafe جدید می‌ریزیم.

Unsafe unsafe = (sun.misc.Unsafe) field.get(null)

# بخش سوم – تخصیص حافظه

اکنون که پوینتر ما ساخته شد لازم است قسمتی از حافظه را از سیستم‌عامل درخواست کنیم، برای این‌کار لازم است تعداد بایت‌های مورد نیاز را به عنوان پارامتر به متد ورودی بدهیم.

درخواست تخصیص حافظه را با دستور allocate انجام می‌دهیم که آدرس حافظه احتصاص داده شده را به عنوان یک متغیر long برمی‌گرداند. آدرس حافظه‌ی تخصیص داده شده را در متغیر memoryAddress می‌ریزیم (در اینجا تعداد بایت های درخواست شده نیز به اندازه یک long است):

long memoryAddress = unsafe.allocateMemory(Long.SIZE)

# بخش چهارم – خواندن و نوشتن در حافظه

اکنون با استفاده از متد putAddress در حافظه‌ای که در قسمت قبل تخصیص داده‌ایم مقداری می‌ریزیم:

unsafe.putAddress(memoryAddress,1)

نتیجه اجرای دستور بالا این است که در آدرس حافظه‌ای که در مرحله قبل گرفتیم عدد یک را ریختیم.

اکنون می‌خواهیم مقداری که به حافظه دادیم را فراخوانی کنیم. از متد getAdress استفاده می‌کنیم و خروجی‌ها را چاپ می‌کنیم.

System.out.println("value of "+memoryAddress+ " is: "+unsafe.getAddress(memoryAddress))

# بخش پنجم – بازگرداندن حافظه اختصاص داده‌شده

آخرین متدی که روی unsafe صدا خواهیم زد متد freeMemory است که آدرس حافظه گرفته شده را به عنوان پارامتر گرفته و آن را به سیستم‌عامل پس می‌دهد(free می کند).

unsafe.freeMemory(memoryAddress)

# بخش ششم - بررسی علت deprecate شدن کلاس Unsafe

همان‌طور که کامپیوترها به سمت قدرتمند شدن پیش‌ می‌روند و زبان‌های برنامه‌نویسی سطح بالاتر به وجود می‌ایند تمایل برنامه‌نویس ها به سر وکار داشتن با جزيیات سخت‌افزار مثل رجیستر cpu و دسترسی مستقیم به منابع حافظه کم‌ می‌شود و با توجه به خطرات این موضوع و سطح بالا بودن زبان جاوا به حذف تدریجی این کلاس منجر شده‌است.

ضمناً این نکته شایان ذکر است که با توجه به ساختار Object و refrence و همچنین garbage collector خوب نیازی به سر و کار داشتن با تخصیص و آزاد نمودن حافظه نیست و وجود این بخش تنها احتمال اشتباه و خطا را زیاد می‌کند.

در بخش بعد یک نمونه از خطای رایج برنامه نویسان C و ++C که با unsafe نیز ممکن است را بررسی می‌کنیم.

# بخش هفتم – ساخت پوینتر سرگردان

در شرح پوینتر سرگردان در ویکی‌پدیا آمده‌است: پوینتری که به خانه‌ای از حافظه اشاره می‌کند که دیگر متعلق به او نیست(deallocate شده است.). با این تعریف برای ساختن پوینتر سرگردان باید از مقدار خانه حافظه بعد از free شدن دسترسی پیدا کنیم و سعی کنیم آن را تغییر داده و دوباره بخوانیم.

با توجه به اینکه آدرس خانه را هنوز در متغیر memoryAddress نگه داشته‌ایم کافیست دستورات قبل را (اینبار بعد از آزاد کردن حافظه تکرار کنیم)

unsafe.putAddress(memoryAddress, 2)

long dangValue = unsafe.getAddress(memoryAddress)

System.out.println("this is danling value : " + dangValue)

همان‌طور که مشاهده می‌کنید در حافظه ای که مربوط به برنامه نبود مقدار ۲ را ریختیم و مجدداً مقدار را خوانده و چاپ کردیم. انتظاری که داریم این است که هیچ کدام از این خط‌ها اجرا نشود و خود jvm یا سیستم عامل جلوی دسترسی برنامه به خانه حافظه خالی شده را بگیرند و خطا رخ دهد (مشابه آنچه در زبان C دیده ایم.) اما با بررسی معلوم می‌شود در حقیقت اتفاق دیگری می‌افتد.

(نکته: با توجه به اینکه متدهای کلاس unsafe مستعد خطا هستند باید حتماً در try, catch قرار گیرند)

پس از اجرا کردن کد بالا یک fatal error از طرف jvm خواهیم داشت و برنامه وارد قسمت catch exception خواهد شد ولی نکته بسیار جالب این است که تمام خط‌های برنامه اجرا می‌شوند و پس از اجرا آخرین خط (System.out.println) برنامه به قسمت catch می‌رسد. نتیجه اجرای کد‌های بالا:

value of 140234709635504 is: 1

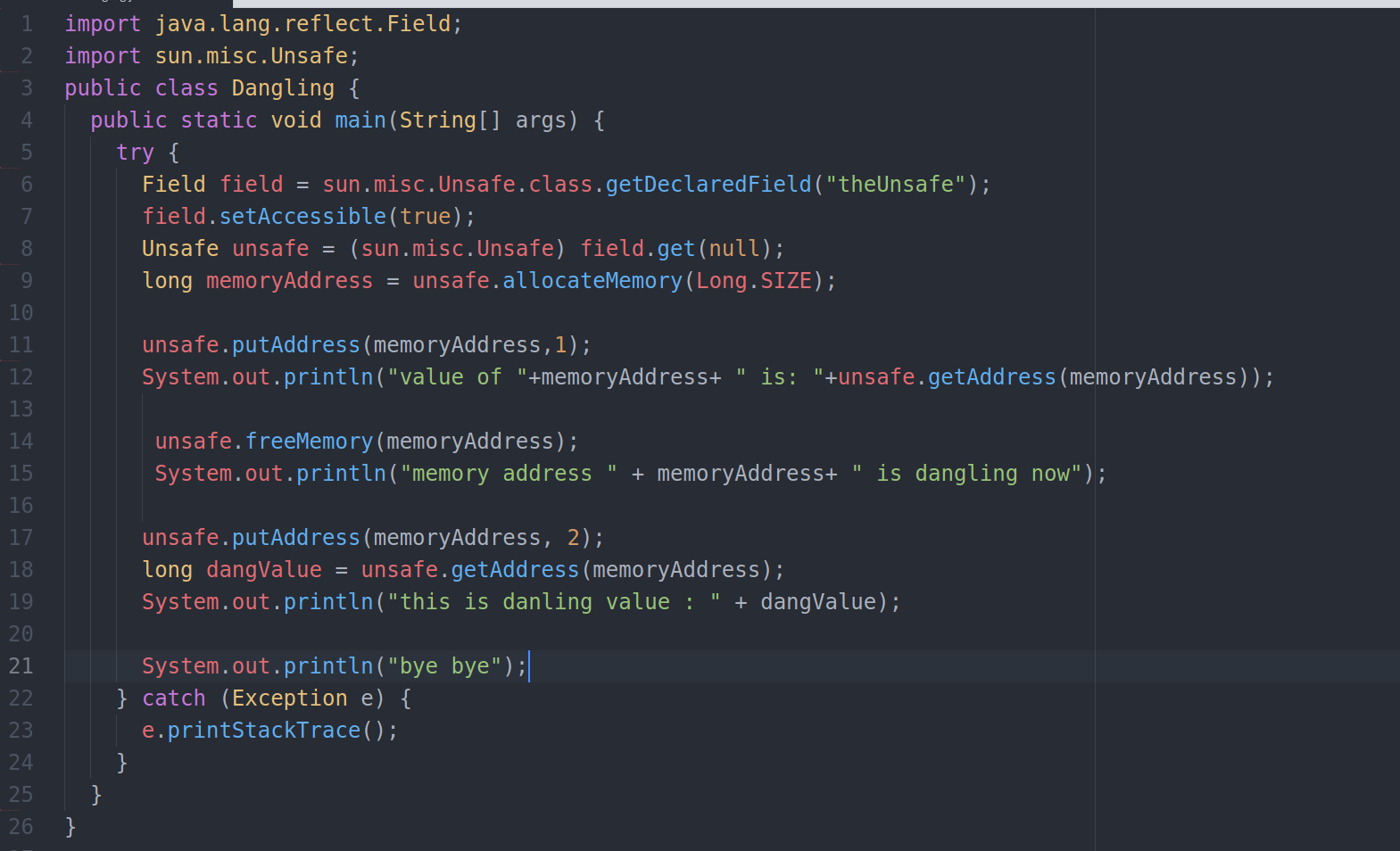
memory address 140234709635504 is dangling now

this is danling value : 2

# A fatal error has been detected by the Java Runtime Environment:

همان‌طور که مشاهده می‌کنید برنامه مقدار ۲ را در حافظه ریخته و سپس از آن بازخوانی کرده و سپس وارد catch شده و Stacktrace را چاپ کرده است.

کد کامل مورد تست و نتیجه اجرا را در زیر مشاهده می‌کنید:



# 

ارائه توجیه برای اتفاق پیش‌ آمده نیاز به مطالعه سورس‌کد ها و داکیومنت‌های متدهای unsafe خواهد داشت که از سطح این مطلب فراتر است.

# جمعبندی

هدف از این مطلب آموزش مختصر دستورات unsafe و بررسی نحوه کار آن‌ها بود که در نهایت رفتار عجیبی از jvm برای مدیریت خطا مشاهده کردیم و احتمالاً برای deprecate کردن این متد‌ها با oracle هم‌نظر شدیم. ضمناً تست‌ها با استفاده از jdk 8 صورت گرفته اند.

# مراجع و منابع

داکیومنت شرکت اوراکل برای متد setAccessible :

[https://docs.oracle.com/javase/10/docs/api/java/lang/reflect/AccessibleObject.html#setAccessible(boolean](https://docs.oracle.com/javase/10/docs/api/java/lang/reflect/AccessibleObject.html" \l "setAccessible(boolean))

توضیح ویکی‌پدیا در مورد dangling pointer:

<https://en.wikipedia.org/wiki/Dangling_pointer>

در نوشتن کدهای استفاده شده از این مطلب کمک گرفته شد:

<https://www.mkyong.com/java/java-write-directly-to-memory/>

برنامه مورد استفاده از این ریپوزیتوری گیتهاب(در اکانت نویسنده مطلب) در دسترس است.

https://github.com/rsharifnasab/java\_pointer

**حقوق معنوی**

این مطلب با هدف آموزشی نوشته شده است. به بهانه نوشتن این مطلب، نویسندگان تشویق به مطالعه و تحقیق شده‌اند و دستاوردهای مطالعاتی یا اجرایی خویش را برای دیگران به اشتراک گذارده‌اند. نشر این مطلب بدون ذکر نویسندگان آن غیرمجاز است. تغییر ترتیب نویسندگان یا حذف/اضافه برخی نویسندگان مجاز نیست. نویسنده/نویسندگان این مطلب اجازه می‌دهند که این مطلب با ذکر نام ایشان منتشر شود. نویسنده/نویسندگان این مطلب اجازه می‌دهند تا ویراست‌های جزئی (مانند ویرایش نگارشی و املایی) در این مطلب برای انتشار اعمال شود.