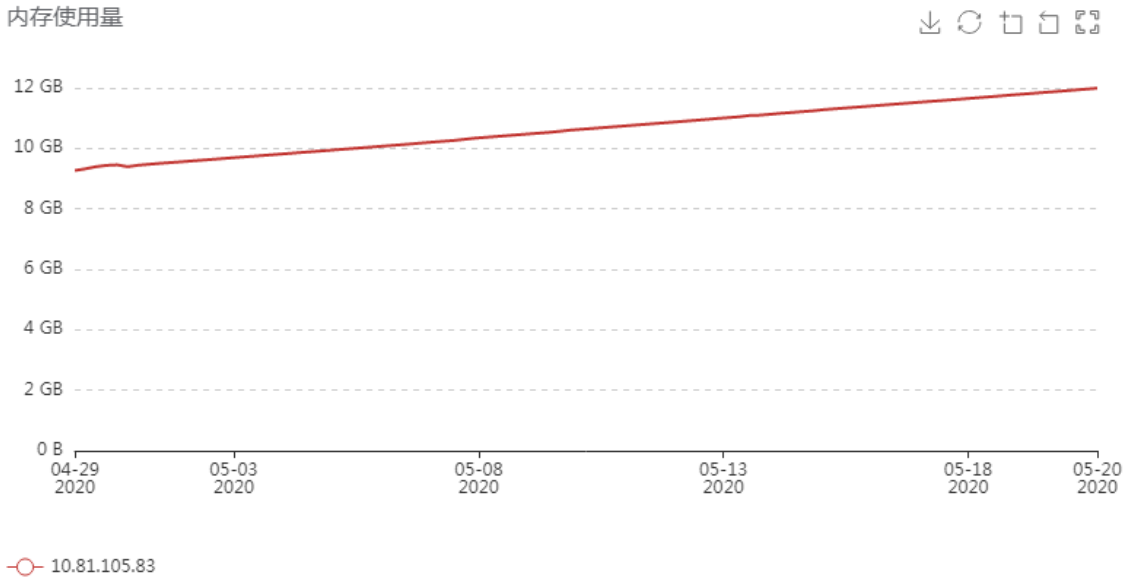
线上堆外内存异常增长 Bug 排查记录

背景

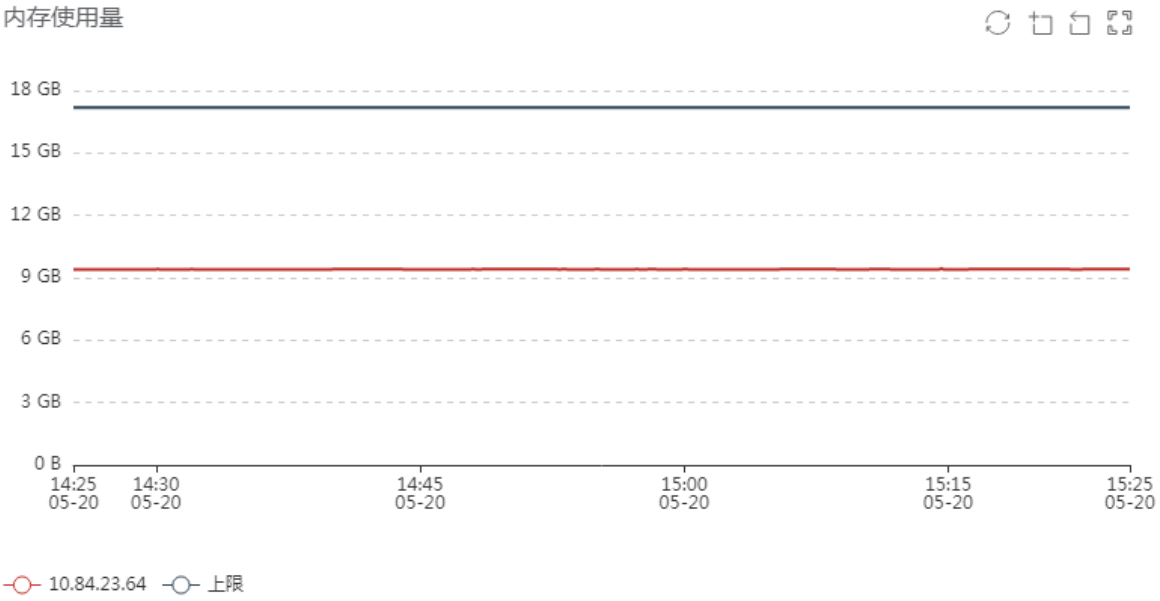
线上长连接服务限制了最大可用堆内存为 8G，并且开启了 `-XX:+AlwaysPreTouch`：

```
Non-default VM flags: -XX:+AlwaysPreTouch -XX:CICompilerCount=4 -XX:CompressedClassSpaceSize=528482304 -XX:ConcGCThreads=6 -XX:ErrorFile=null -XX:G1HeapRegionSize=4194304 -XX:G1MixedGCCountTarget=16 -XX:G1NewSizePercent=40 -XX:G1RSetUpdatingPauseTimePercent=30 -XX:G1ReservePercent=15 -XX:+HeapDumpAfterFullGC -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=null -XX:InitialHeapSize=8589934592 -XX:InitiatingHeapOccupancyPercent=30 -XX:MarkStackSize=4194304 -XX:MaxGCPauseMillis=500 -XX:MaxHeapSize=8589934592 -XX:MaxMetaspaceSize=536870912 -XX:MaxNewSize=5150605312 -XX:MetaspaceSize=536870912 -XX:MinHeapDeltaBytes=4194304 -XX:NativeMemoryTracking=null -XX:ParallelGCThreads=16 -XX:+ParallelRefProcEnabled -XX:+PrintGC -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -XX:+StartAttachListener -XX:ThreadStackSize=256 -XX:+UnlockExperimentalVMOptions -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
```

由于开启上线一段时间后堆外内存一直在增长：



测试环境没这个问题，但测试环境最近也没在集中测试，行为和线上不能说是一致的，所以不能作为依据：



排查

安装 gperftools

为了跟踪堆外内存的分配，这里要了个 root 权限，在线上机器安装 gperftools：

```
yum install libunwind-devel
yum install gcc-c++
wget https://github.com/gperftools/gperftools/releases/download/gperftools-
2.7.90/gperftools-2.7.90.tar.gz
tar xf gperftools-2.7.90.tar.gz
cd gperftools-2.7.90
./configure
make
sudo make install
```

程序运行前添加相关库（LD_PRELOAD），将其位置加入 usr_local_lib.conf 中，执行 ldconfig 生效：

```
export LD_PRELOAD=/usr/local/lib/libtcmalloc.so
sudo echo -e "\n/usr/local/lib" >> /etc/ld.so.conf.d/usr_local_lib.conf
sudo /sbin/ldconfig
```

指定 heap profile 的路径和前缀（HEAPPROFILE）：

```
export HEAPPROFILE=<base>/<prefix>
```

启动后，会在 base 生成 `<prefix>.xxxx.heap`，xxxx 为序号。

```
-rw-rw-r-- 1 service service 1048561 May 25 15:29 perftools.0081.heap
-rw-rw-r-- 1 service service 1048561 May 25 17:07 perftools.0082.heap
-rw-rw-r-- 1 service service 1048572 May 25 18:45 perftools.0083.heap
-rw-rw-r-- 1 service service 1048571 May 25 20:20 perftools.0084.heap
-rw-rw-r-- 1 service service 1048567 May 25 21:42 perftools.0085.heap
-rw-rw-r-- 1 service service 1048572 May 25 23:17 perftools.0086.heap
-rw-rw-r-- 1 service service 1048572 May 26 00:53 perftools.0087.heap
-rw-rw-r-- 1 service service 1048572 May 26 02:31 perftools.0088.heap
-rw-rw-r-- 1 service service 1048573 May 26 04:09 perftools.0089.heap
-rw-rw-r-- 1 service service 1048572 May 26 05:46 perftools.0090.heap
-rw-rw-r-- 1 service service 1048572 May 26 07:24 perftools.0091.heap
-rw-rw-r-- 1 service service 1048565 May 26 09:01 perftools.0092.heap
-rw-rw-r-- 1 service service 1048570 May 26 10:40 perftools.0093.heap
-rw-rw-r-- 1 service service 1048571 May 26 12:09 perftools.0094.heap
-rw-rw-r-- 1 service service 1048568 May 26 13:47 perftools.0095.heap
-rw-rw-r-- 1 service service 1048567 May 26 15:24 perftools.0096.heap
-rw-rw-r-- 1 service service 1048566 May 26 16:58 perftools.0097.heap
-rw-rw-r-- 1 service service 1048569 May 26 18:34 perftools.0098.heap
-rw-rw-r-- 1 service service 1048571 May 26 20:11 perftools.0099.heap
-rw-rw-r-- 1 service service 1048566 May 26 21:47 perftools.0100.heap
-rw-rw-r-- 1 service service 1048560 May 26 23:21 perftools.0101.heap
-rw-rw-r-- 1 service service 1048560 May 27 00:59 perftools.0102.heap
-rw-rw-r-- 1 service service 1048568 May 27 02:36 perftools.0103.heap
-rw-rw-r-- 1 service service 1048569 May 27 04:14 perftools.0104.heap
```

pprof 对比 heap

之后就可以使用 pprof 对这些 heap 文件进行分析了：

```
pprof --text $JAVA_HOME/bin/java <file_name> > out
```

还可以对比两个 heap，看出差异：

```
pprof --text --base=<file1_name> $JAVA_HOME/bin/java <file2_name> > compare.out
```

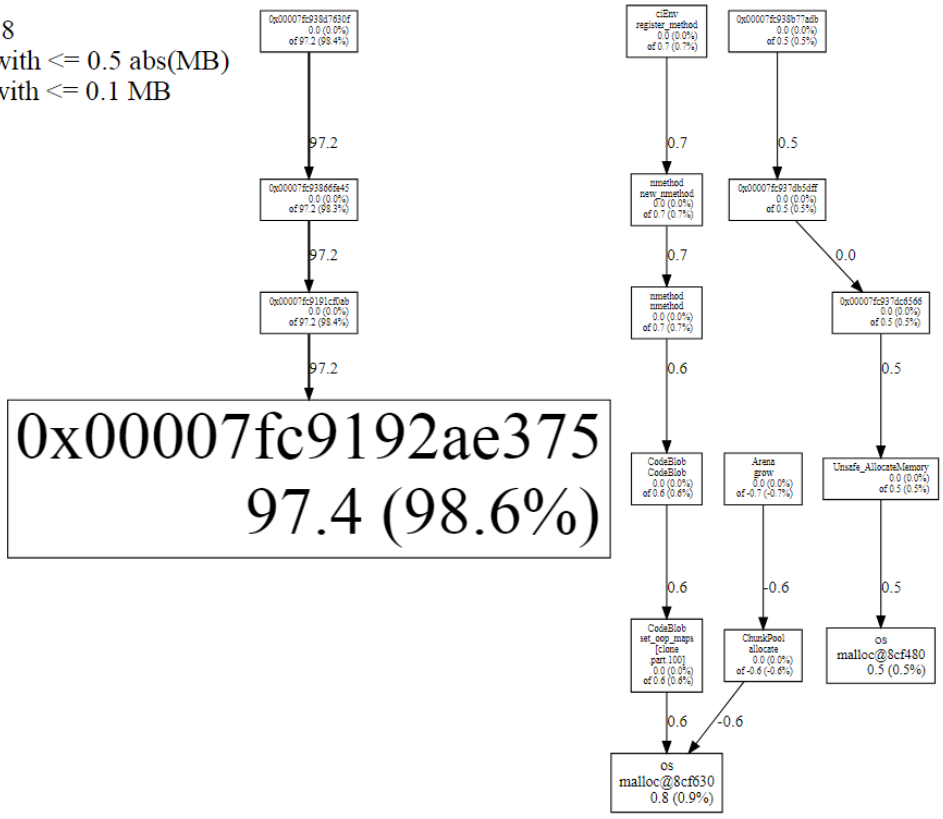
> 具体可以使用 `pprof --list` 看说明。

我们对比 heap，输出为 svg，可视化地查看内存分配：

```
pprof --functions --svg --base=./perftools.0006.heap $JAVA_HOME/bin/java
./perftools.0022.heap > compare_6_22.svg
pprof --functions --svg --base=./perftools.0006.heap $JAVA_HOME/bin/java
./perftools.0033.heap > compare_6_33.svg
```

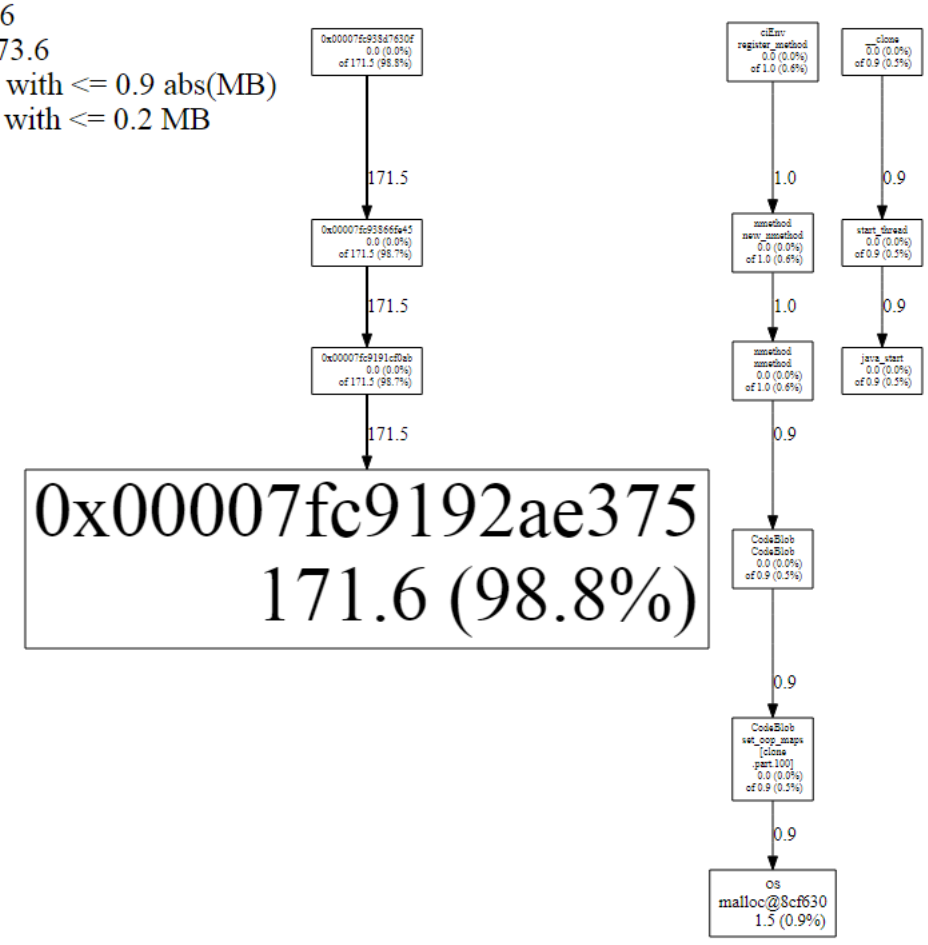
发现有一段内存确实在增长，图中是一天的量：

/usr/local/jdk8/bin/java
Total MB: 98.8
Focusing on: 98.8
Dropped nodes with <= 0.5 abs(MB)
Dropped edges with <= 0.1 MB



2 天的量：

/usr/local/jdk8/bin/java
Total MB: 173.6
Focusing on: 173.6
Dropped nodes with <= 0.9 abs(MB)
Dropped edges with <= 0.2 MB



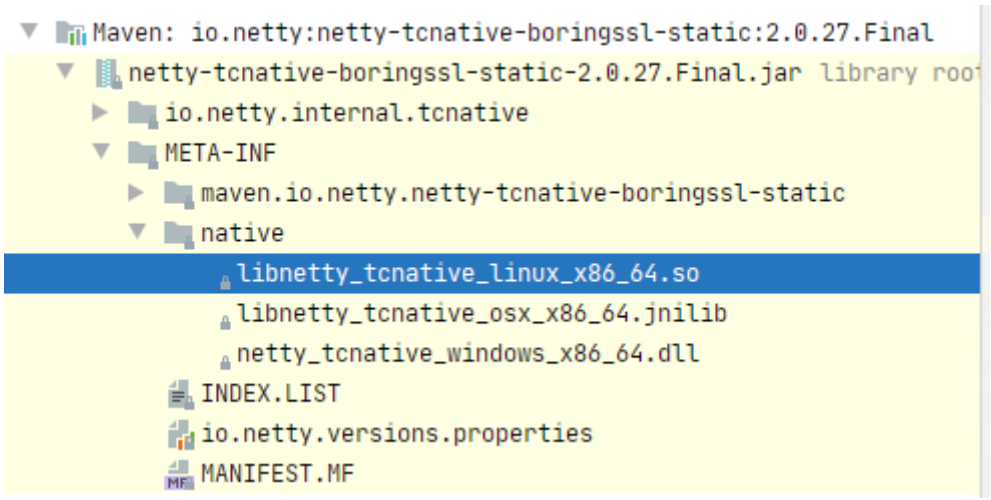
pmap 查看来源

pmap 查询申请 `0x00007fc9192ae375` 这段内存的库：

```
00007fc918d69000      8K r-x-- /usr/lib64/libfreebl3.so
00007fc918d6b000   2044K ----- /usr/lib64/libfreebl3.so
00007fc918f6a000      4K r---- /usr/lib64/libfreebl3.so
00007fc918f6b000      4K rw--- /usr/lib64/libfreebl3.so
00007fc918f6c000     32K r-x-- /usr/lib64/libcrypt-2.17.so
00007fc918f74000   2044K ----- /usr/lib64/libcrypt-2.17.so
00007fc919173000      4K r---- /usr/lib64/libcrypt-2.17.so
00007fc919174000      4K rw--- /usr/lib64/libcrypt-2.17.so
00007fc919175000    184K rw---   [ anon ]
00007fc9191ac000   2188K r-x-- /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_6412789975708419247 6.so (deleted)
00007fc9193cf000   2048K ----- /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_6412789975708419247 6.so (deleted)
00007fc9195cf000     80K rw--- /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_6412789975708419247 6.so (deleted)
00007fc9195e3000      8K rw---   [ anon ]
00007fc9195e5000     12K -----   [ anon ]
00007fc9195e8000    248K rw---   [ anon ]
00007fc919626000     12K -----   [ anon ]
00007fc919629000    248K rw---   [ anon ]
```

```
00007fc918f6c000     32K r-x-- /usr/lib64/libcrypt-2.17.so
00007fc918f74000   2044K ----- /usr/lib64/libcrypt-2.17.so
00007fc919173000      4K r---- /usr/lib64/libcrypt-2.17.so
00007fc919174000      4K rw--- /usr/lib64/libcrypt-2.17.so
00007fc919175000    184K rw---   [ anon ]
00007fc9191ac000   2188K r-x-- /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_6412789975708419247 6.so (deleted)
00007fc9193cf000   2048K ----- /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_6412789975708419247 6.so (deleted)
00007fc9195cf000     80K rw--- /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_6412789975708419247 6.so (deleted)
00007fc9195e3000      8K rw---   [ anon ]
00007fc9195e5000     12K -----   [ anon ]
00007fc9195e8000    248K rw---   [ anon ]
```

可以看到这段内存是 libnetty_tcnative_linux_x86_64127899757084192476.so 这个库申请的，该库来源于 Netty fork 自 Tomcat 的一个项目，该项目提供 SSL 相关的 Native 方法支持：



源码在 Github 上。

回溯源码

我们根据日志回溯 netty-tcnative 包相关的源码，这里包含两个部分：加载和使用。

加载

程序启动时 io.netty.handler.ssl.OpenSsl 其会加载上述库：

```
2020-05-21 18:09:59,706 DEBUG [MqttConnectServer-I/O-Worker-3-1] (i.n.u.i.NativeLibraryLoader:342) - Successfully loaded the library /home/service/app/access-server-mqtt/mhopgup6tjy0/access-server-mqtt-dev-20200520161238/bin/../tmp/libnetty_tcnative_linux_x86_64524764477736046 9980.so
2020-05-21 18:09:59,707 DEBUG [MqttConnectServer-I/O-Worker-3-1] (i.n.h.s.OpenSsl:146) - Initialize netty-tcnative using engine: 'default'
2020-05-21 18:09:59,723 DEBUG [MqttConnectServer-I/O-Worker-3-1] (i.n.h.s.OpenSsl:171) - netty-tcnative using native library: BoringSSL
```

找寻加载相关动作的源码：

```
try {
    // The JNI library was not already loaded. Load it now.
    loadTcNative();
} catch (Throwable t) {
    cause = t;
    logger.debug(
            msg: "Failed to load netty-tcnative; " +
                    OpenSslEngine.class.getSimpleName() + " will be unavailable, unless the " +
                    "application has already loaded the symbols by some other means. " +
                    "See https://netty.io/wiki/forked-tomcat-native.html for more information.", t);
}

try {
    String engine = SystemPropertyUtil.get( key: "io.netty.handler.ssl.openssl.engine",  def: null);
    if (engine == null) {
        logger.debug( msg: "Initialize netty-tcnative using engine: 'default'");
    } else {
        logger.debug("Initialize netty-tcnative using engine: '{}'", engine);
    }
    initializeTcNative(engine);

    // The library was initialized successfully. If loading the library failed above,
    // reset the cause now since it appears that the library was loaded by some other
    // means.
    cause = null;
```

```
private static void loadTcNative() throws Exception {
    String os = PlatformDependent.normalizedOs();
    String arch = PlatformDependent.normalizedArch();

    Set<String> libNames = new LinkedHashSet<~>( initialCapacity: 5);
    String staticLibName = "netty_tcnative";

    // First, try loading the platform-specific library. Platform-specific
    // libraries will be available if using a tcnative uber jar.
    if ("linux".equalsIgnoreCase(os)) {
        Set<String> classifiers = PlatformDependent.normalizedLinuxClassifiers();
        for (String classifier : classifiers) {
            libNames.add(staticLibName + "_" + os + '_' + arch + "_" + classifier);
        }
        // generic arch-dependent library
        libNames.add(staticLibName + "_" + os + '_' + arch);

        // Fedora SSL lib so naming (libssl.so.10 vs libssl.so.1.0.0).
        // note: should already be included from the classifiers but if not, we use this as an
        //       additional fallback option here
        libNames.add(staticLibName + "_" + os + '_' + arch + "_fedora");
    } else {
        libNames.add(staticLibName + "_" + os + '_' + arch);
    }
    libNames.add(staticLibName + "_" + arch);
    libNames.add(staticLibName);

    NativeLibraryLoader.loadFirstAvailable(SSL.class.getClassLoader(),
        libNames.toArray(new String[0]));
}
```

```
/**
 * Loads the first available library in the collection with the specified
 * {@link ClassLoader}.
 *
 * @throws IllegalArgumentException
 *         if none of the given libraries load successfully.
 */
public static void loadFirstAvailable(ClassLoader loader, String... names) {
    List<Throwable> suppressed = new ArrayList<~>();
    for (String name : names) {
        try {
            load(name, loader);
            return;
        } catch (Throwable t) {
            suppressed.add(t);
            logger.debug( format: "Unable to load the library '{}', trying next name...", name, t);
        }
    }
    IllegalArgumentException iae =
            new IllegalArgumentException("Failed to load any of the given libraries: " + Arrays.toString(names));
    ThrowableUtil.addSuppressedAndClear(iae, suppressed);
    throw iae;
}
```

往上追溯日志可以看到：

```
2020-05-21 18:09:59,544 DEBUG [MqttConnectServer-I/O-Worker-3-1] (i.n.u.i.NativeLibraryLoader:141) - netty_tcnative_linux_x86_64 cannot be loaded from java.library.path, now trying export to -Dio.netty.native.workdir: /home/service/app/access-server-mqtt/mhopgup6tjy0/access-server-mqtt-dev-20200520161238/bin/../tmp
java.lang.UnsatisfiedLinkError: no netty_tcnative_linux_x86_64 in java.library.path
```

查看源码：

```
134         try {
135             // first try to load from java.library.path
136             loadLibrary(loader, name,  absolute: false);
137             return;
138         } catch (Throwable ex) {
139             suppressed.add(ex);
140             logger.debug(
141                     "{} cannot be loaded from java.library.path, "
142                     + "now trying export to -Dio.netty.native.workdir: {}", name, WORKDIR, ex);
143         }
144
```

可以看出，Netty 的策略是：如果在 java.library.path 中找不到对应的 .so 文件，则自己从 Jar 包中拷出来输出到临时目录，之后加载到内存。

在 -Djava.io.tmpdir 指定的临时目录创建该文件：

```
String libname = System.mapLibraryName(name);
String path = NATIVE_RESOURCE_HOME + libname;

InputStream in = null;
OutputStream out = null;
File tmpFile = null;
URL url;
if (loader == null) {
    url = ClassLoader.getSystemResource(path);
} else {
    url = loader.getResource(path);
}
try {
    if (url == null) {
        if (PlatformDependent.isOsx()) {
            String fileName = path.endsWith(".jnilib") ? NATIVE_RESOURCE_HOME + "lib" + name + ".dynlib" :
                    NATIVE_RESOURCE_HOME + "lib" + name + ".jnilib";
            if (loader == null) {
                url = ClassLoader.getSystemResource(fileName);
            } else {
                url = loader.getResource(fileName);
            }
            if (url == null) {
                FileNotFoundException fnf = new FileNotFoundException(fileName);
                ThrowableUtil.addSuppressedAndClear(fnf, suppressed);
                throw fnf;
            }
        } else {
            FileNotFoundException fnf = new FileNotFoundException(path);
            ThrowableUtil.addSuppressedAndClear(fnf, suppressed);
            throw fnf;
        }
    }

    int index = libname.lastIndexOf( ch: '.');
    String prefix = libname.substring(0, index);
    String suffix = libname.substring(index);

    tmpFile = File.createTempFile(prefix, suffix, WORKDIR);
    in = url.openStream();
    out = new FileOutputStream(tmpFile);
```

"META-INF/native/"

输出流：

```
tmpFile = File.createTempFile(prefix, suffix, WORKDIR);
in = url.openStream();
out = new FileOutputStream(tmpFile);

if (shouldShadedLibraryIdBePatched(packagePrefix)) {
    patchShadedLibraryId(in, out, originalName, name);
} else {
    byte[] buffer = new byte[8192];
    int length;
    while ((length = in.read(buffer)) > 0) {
        out.write(buffer,  off: 0, length);
    }
}

out.flush();
```

加载到内存：

```
// Close the output stream before loading the unpacked library,
// because otherwise Windows will refuse to load it when it's in use by other process.
closeQuietly(out);
out = null;
loadLibrary(loader, tmpFile.getPath(),  absolute: true);
```

成功：

```
/**
 * Loading the native library into the specified {@link ClassLoader}.
 * @param loader - The {@link ClassLoader} where the native library will be loaded into
 * @param name - The native library path or name
 * @param absolute - Whether the native library will be loaded by path or by name
 */
private static void loadLibrary(final ClassLoader loader, final String name, final boolean absolute) {
    Throwable suppressed = null;
    try {
        try {
            // Make sure the helper is belong to the target ClassLoader.
            final Class<?> newHelper = tryToLoadClass(loader, NativeLibraryUtil.class);
            loadLibraryByHelper(newHelper, name, absolute);
            logger.debug("Successfully loaded the library {}", name);
            return;
        } catch (UnsatisfiedLinkError e) { // Should by pass the UnsatisfiedLinkError here!
            suppressed = e;
            logger.debug( format: "Unable to load the library '{}', trying other loading mechanism.", name, e);
        } catch (Exception e) {
            suppressed = e;
            logger.debug( format: "Unable to load the library '{}', trying other loading mechanism.", name, e);
        }
        NativeLibraryUtil.loadLibrary(name, absolute);  // Fallback to local helper class.
        logger.debug("Successfully loaded the library {}", name);
    } catch (UnsatisfiedLinkError ule) {
        if (suppressed != null) {
            ThrowableUtil.addSuppressed(ule, suppressed);
        }
        throw ule;
    }
}
```

于是有了第一行的输出：

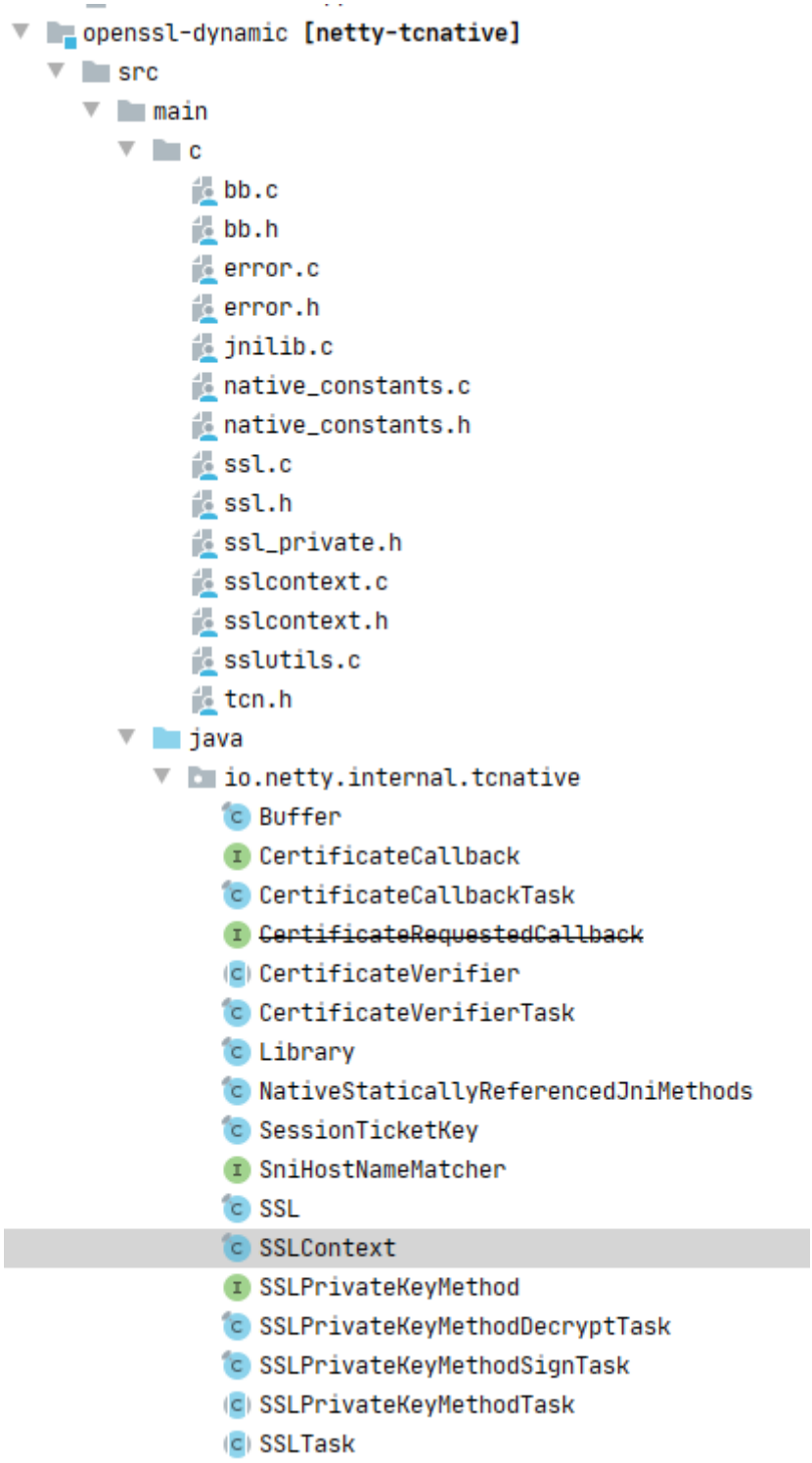
```
2020-05-21 18:09:59,706 DEBUG [MqttConnectServer-I/O-Worker-3-1]
(i.n.u.i.NativeLibraryLoader:342) - Successfully loaded the library
/home/service/app/access-server-mqtt/mhopgup6tjy0/access-server-mqtt-dev-
20200520161238/bin/../tmp/libnetty_tcnative_linux_x86_64527464477736046998O.so
```

最终的加载还是使用 JDK 提供的 JNI 方法，即 `java.lang.System#load`：

```
 */
final class NativeLibraryUtil {
    /**
     * Delegate the calling to {@link System#load(String)} or {@link System#loadLibrary(String)}.
     * @param libName - The native library path or name
     * @param absolute - Whether the native library will be loaded by path or by name
     */
    public static void loadLibrary(String libName, boolean absolute) {
        if (absolute) {
            System.load(libName);
        } else {
            System.loadLibrary(libName);
        }
    }

    private NativeLibraryUtil() {
        // Utility
    }
}
```

使用

好了，加载的代码找到了，现在我们来查找使用此库的代码。



可以看到，SSL 相关的 native 方法均由该库提供。

安装“另一个 pprof”

现在我们需要明确这段内存的方法名，查看 pprof 的版本，发现还有另一个 Google 维护的 仓库，其提供 Symbolization 相关的 demangle 功能。

> Google 牛逼。

我们安装 golang（需要 1.13 以上）后运行 `go get -u github.com/google/pprof`，然后使用 `/home/gopath/bin/pprof`。

> Go 安装只是一种方式。

运行后发现报错：

```
[root@10-81-105-83.access-server-mqtt.bjht release]# /home/gopath/bin/pprof -symbolize=demangle=full $JAVA_HOME/bin/java ./perftools.0033.heap > 33.svg
Local symbolization failed for libnetty_tcnative_linux_x86_6412789975708419247 6.so: stat /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_tcnative_linux_x86_64
12789975708419247 6.so: no such file or directory
Local symbolization failed for libnetty_transport_native_epoll_x86_6457930161379867154 68.so: stat /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/libnetty_transport_na
tive_epoll_x86_6457930161379867154 68.so: no such file or directory
Some binary filenames not available. Symbolization may be incomplete.
Try setting PPROF_BINARY_PATH to the search path for local binaries.
File: java
Type: inuse_space
Entering interactive mode (type "help" for commands, "o" for options)
pprof: Interrupt
```

输出有提醒我们：**这两个 .so 被删除了**，所以 load 不了。

> Netty 默认会在加载完 .so 后将其从临时目录删除，可以配置 `-Dio.netty.native.deleteLibAfterLoading` 来改变行为。

我们手动添加回这两个 .so 到 tmp 中并重命名：

```
[root@10-81-105-83.access-server-mqtt.bjht release]# ll /home/service/app/access-server-mqtt/krwoiskfytfe/access-server-mqtt-master-20200520165919/tmp/
total 2660
drwxrwxr-x 2 service service    4096 May 20 17:01 heracles
-rw------- 1 root    root    2630627 May 22 10:57 libnetty_tcnative_linux_x86_6412789975708419247 6.so
-rw------- 1 root    root      82248 May 22 10:58 libnetty_transport_native_epoll_x86_6457930161379867154 68.so
```

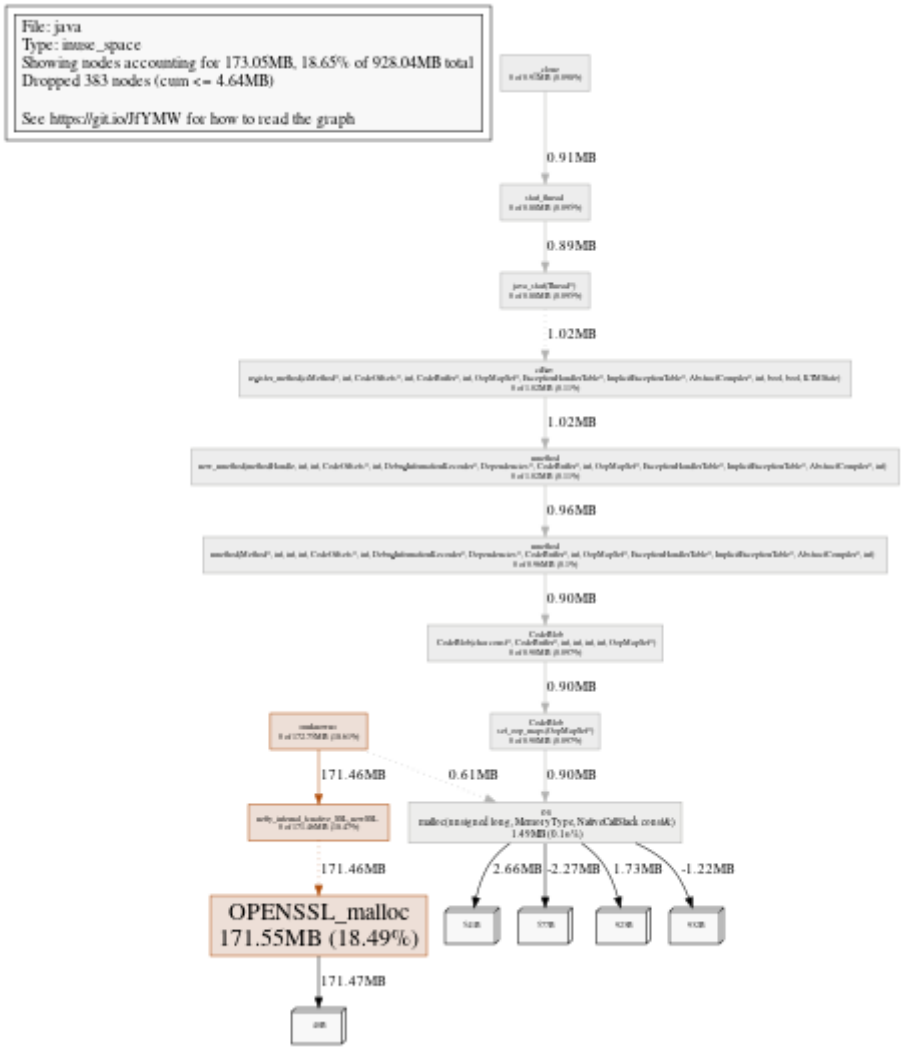
再次执行：

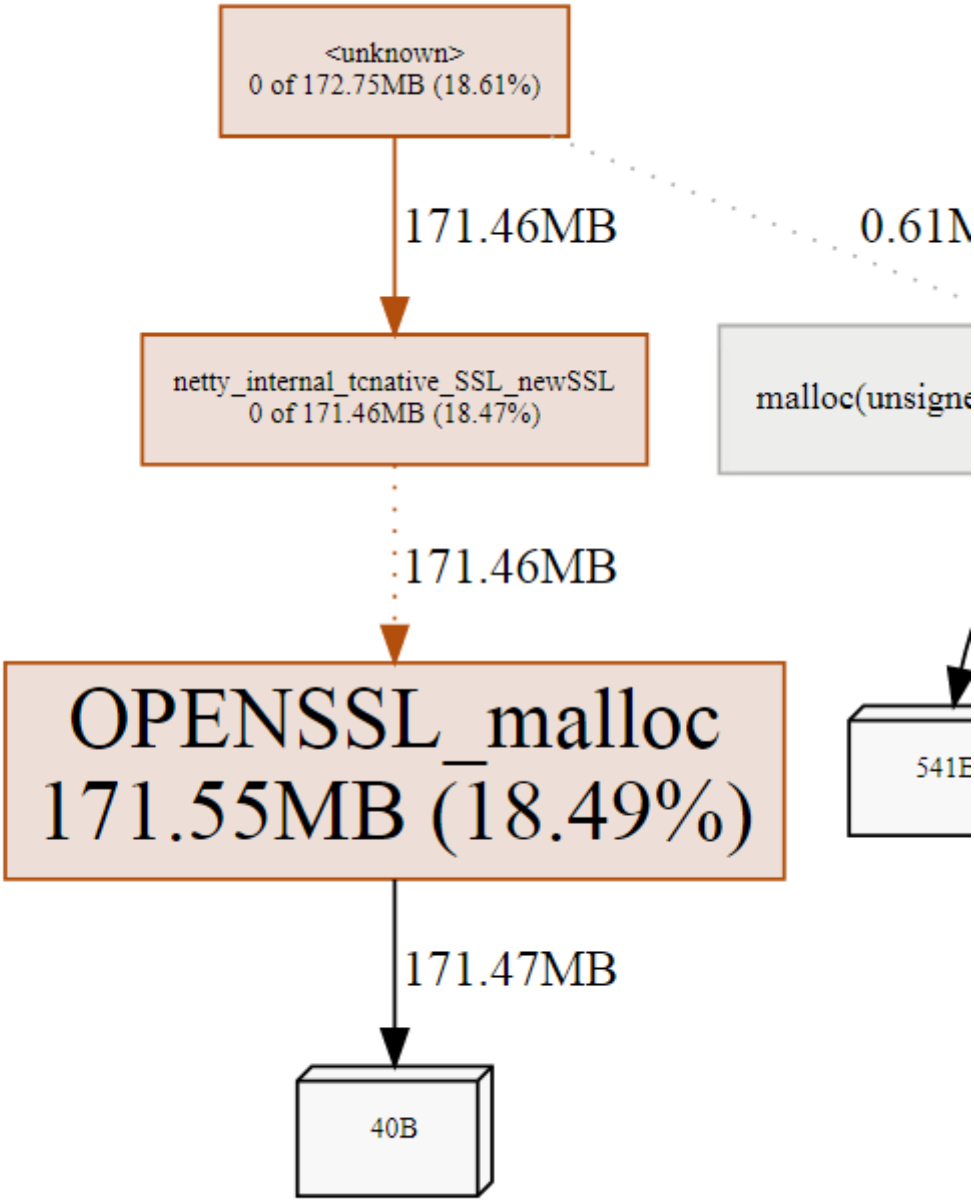
```
/home/gopath/bin/pprof -symbolize=demangle=full --functions --svg --base=./perftools.0006.heap $JAVA_HOME/bin/java ./perftools.0033.heap > compare_6_33.svg
```

找到来源

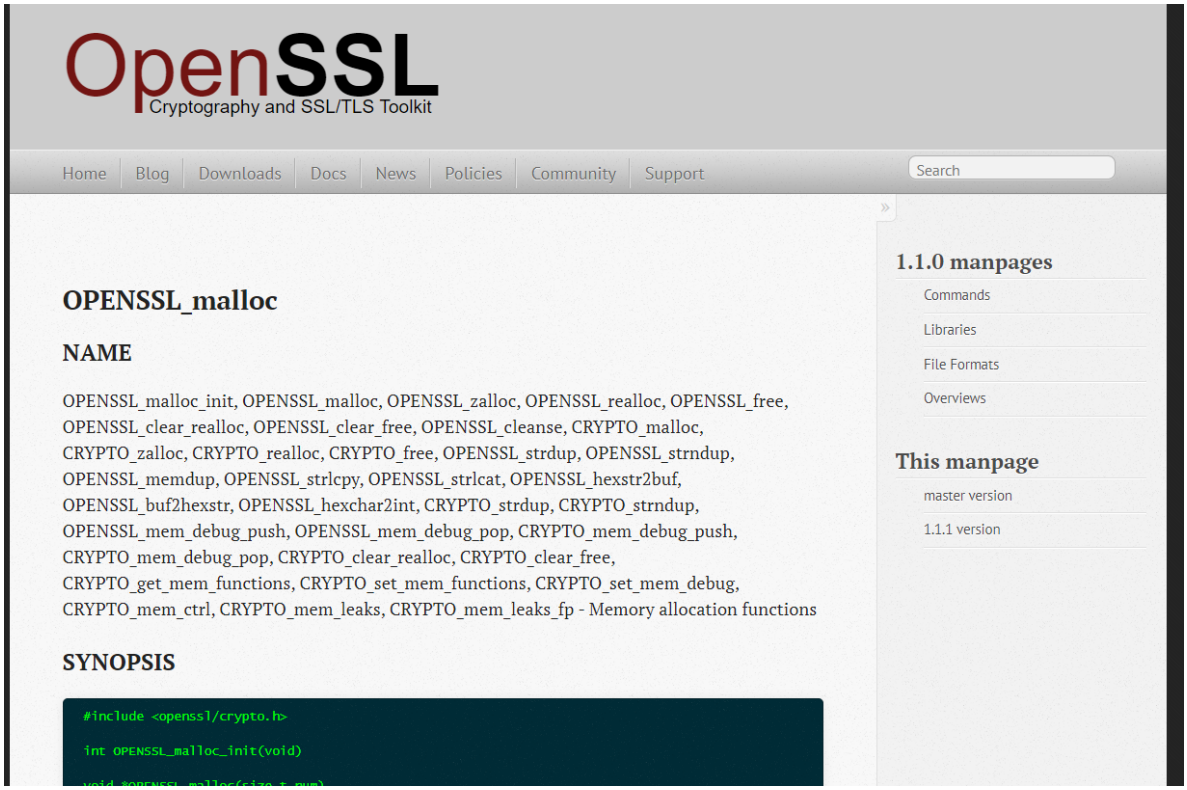
内存申请

终于，我们查到是 OpenSSL 申请的内存：


File: java
Type: inuse_space
Showing nodes accounting for 173.05MB, 18.65% of 928.04MB total
Dropped 383 nodes (cum <= 4.64MB)
See https://git.io/JfYMW for how to read the graph
0.91MB
0.89MB
1.02MB
1.02MB
0.96MB
0.90MB
0.90MB
171.46MB
0.61MB
0.90MB
171.46MB
2.66MB
-2.27MB
1.73MB
-1.22MB
OPENSSL_malloc
171.55MB (18.49%)
171.47MB



`OPENSSL_malloc` 这个方法是 OpenSSL 提供的 API，封装了其申请内存相关的操作：

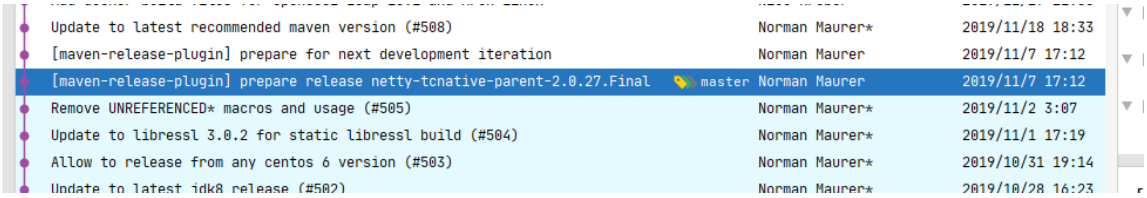


提出疑问

但是从日志上看，实际上我们的 **连接数并没有变化**，那为什么 `OPENSSL_malloc` 申请的内存会不断增长呢?

再次回到源码

回查 netty-tcnative 源码：



> 线上使用的是 2.0.27.Final 版本。

pprof 的输出中，我们看到，调用 `OPENSSL_malloc` 的方法是 `netty_internal_tcnative_SSL_newSSL`，从命名可以直接看出其对应的 Java 全限定名 io.netty.internal.tcnative.SSL#newSSL，查看 SSL.java：

```java
/**
 * SSL_new
 * @param ctx Server or Client context to use.
 * @param server if true configure SSL instance to use accept handshake routines
 *               if false configure SSL instance to use connect handshake routines
 * @return pointer to SSL instance (SSL *)
 */
public static native long newSSL(long ctx, boolean server);
```

SSL.c 中的实现如下：

```
TCN_IMPLEMENT_CALL(jlong /* SSL * */, SSL, newSSL)(TCN_STDARGS,
                                                   jlong ctx /* tcn_ssl_ctxt_t * */,
                                                   jboolean server) {
    SSL *ssl = NULL;
    tcn_ssl_ctxt_t *c = J2P(ctx, tcn_ssl_ctxt_t *);
    tcn_ssl_state_t *state = NULL;

    TCN_CHECK_NULL(c, ctx, 0);

    if ((ssl = SSL_new(c->ctx)) == NULL) {
        tcn_ThrowException(e, "cannot create new ssl");
        return 0;
    }

    if ((state = new_ssl_state(c)) == NULL) {
        SSL_free(ssl);
        tcn_ThrowException(e, "cannot create new ssl state struct");
        return 0;
    }

    // Set the app_data2 before all the others because it may be used in SSL_free.
    tcn_SSL_set_app_state(ssl, state);

    // Add callback to keep track of handshakes.
    SSL_CTX_set_info_callback(c->ctx, ssl_info_callback);

    if (server) {
        SSL_set_accept_state(ssl);
    } else {
        SSL_set_connect_state(ssl);
    }

    return P2J(ssl);
}
```

可以看到，确实有通过 `OPENSSL_malloc` 申请内存的地方，比如创建 tcn_ssl_state_t 结构体时：

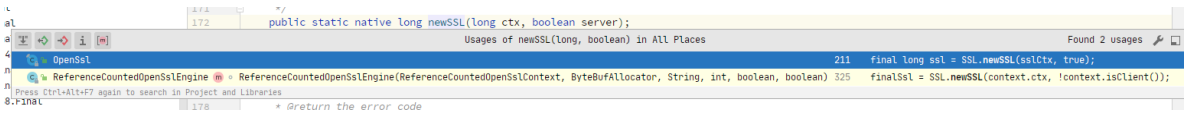
```
static tcn_ssl_state_t* new_ssl_state(tcn_ssl_ctxt_t* ctx) {
    if (ctx == NULL) {
        return NULL;
    }

    tcn_ssl_state_t* state = OPENSSL_malloc(sizeof(tcn_ssl_state_t));
    if (state == NULL) {
        return NULL;
    }
    memset(state, 0, sizeof(tcn_ssl_state_t));
    state->ctx = ctx;

     // Initially we will share the configuration from the SSLContext.
    state->verify_config = &ctx->verify_config;
    return state;
}
```

64 位下每次 newSSL 申请 32 bytes 内存：

```
struct tcn_ssl_state_t {
    int handshakeCount;
    tcn_ssl_ctxt_t *ctx;
    tcn_ssl_task_t* ssl_task;
    tcn_ssl_verify_config_t* verify_config;
};
```

还有其他调用，具体就不看了。

我们回到更熟悉的 Java 代码，调用 `newSSL()` 的地方有两个：



其中，OpenSsl 里的调用只在类加载时触发一次（静态代码块），除非有通过 ClassLoader 去 hot reload，否则只会申请一次，先排除。

剩下就是 io.netty.handler.ssl.ReferenceCountedOpenSslEngine 了。

到底是那个 SslEngine?

看来关键点在 SSLEngine，那么我们是什么时候用到 javax.net.ssl.SSLEngine 的呢？而这个 SSLEngine 的具体实现类是哪个呢？

查看使用到 SSLEngine 的源码：

```
                .getResourceAsStream(trustCertFilePath);
        ctx = SslContextBuilder.forServer(certFileStream, keyFileStream)
                .trustManager(trustCertPath)
                .sslProvider(SslProvider.OPENSSL)
                .ciphers(Collections.singletonList(cipherSuite))
                .protocols(sslVersion)
                .enableOcsp(false)
                .clientAuth(ClientAuth.REQUIRE)
                .startTls(false)
                .build();
    } catch (Throwable e) {
        log.error(String.format("failed to initialize %s", SslFactory.class.getName()), e);
        throw new InitializationException(e);
    }
}

public SslHandler gen() {
    // use thread pool rather than the reactor I/O threads to handle ssl handshake
    SslHandler handler = ctx.newHandler(PooledByteBufAllocator.DEFAULT);
    handler.setHandshakeTimeoutMillis(handshakeTimeout);
    return handler;
}
```

```
if (!Env.LOCAL.equals(EnvUtils.getEnv())) {
    pipeline.addLast( name: "sslHandler", SslFactory.SINGLETON.gen());
}
// FIXME: should be dynamic
```

可以看到，每次建立连接后，根据 ChannelInitializer 的逻辑来初始化 ChannelHandlerContext 上下文内的流水线时会调用。

而 SslHandler close 时其持有的 SslEngine 会一同关闭：

```
/**
 * Sends an SSL {@code close_notify} message to the specified channel and
 * destroys the underlying {@link SSLEngine}. This will <strong>not</strong> close the underlying
 * {@link Channel}. If you want to also close the {@link Channel} use {@link Channel#close()} or
 * {@link ChannelHandlerContext#close()}
 */
public ChannelFuture closeOutbound(final ChannelPromise promise) {
    final ChannelHandlerContext ctx = this.ctx;
    if (ctx.executor().inEventLoop()) {
        closeOutbound0(promise);
    } else {
        ctx.executor().execute(() -> { closeOutbound0(promise); });
    }
    return promise;
}
```

```
private void closeOutbound0(ChannelPromise promise) {
    outboundClosed = true;
    engine.closeOutbound();
    try {
        flush(ctx, promise);
    } catch (Exception e) {
        if (!promise.tryFailure(e)) {
            logger.warn( format: "{} flush() raised a masked exception.", ctx.channel(), e);
        }
    }
}
```

> 这里有两个 close 动作（`closeInbound()` 和 `closeOutbound()`），这里只列了一个示意。

简而言之，就是 **建连时 new 一个，断连时关掉。**

SslContextBuilder 这个建造者设置使用的 SSL 提供方是 SslProvider.OPENSSL，之后生产对应的 io.netty.handler.ssl.SslContext 实现类为 io.netty.handler.ssl.OpenSslServerContext：

```
static SslContext newServerContextInternal(
        SslProvider provider,
        Provider sslContextProvider,
        X509Certificate[] trustCertCollection, TrustManagerFactory trustManagerFactory,
        X509Certificate[] keyCertChain, PrivateKey key, String keyPassword, KeyManagerFactory keyManagerFactory,
        Iterable<String> ciphers, CipherSuiteFilter cipherFilter, ApplicationProtocolConfig apn,
        long sessionCacheSize, long sessionTimeout, ClientAuth clientAuth, String[] protocols, boolean startTls,
        boolean enableOcsp, String keyStoreType) throws SSLException {

    if (provider == null) {
        provider = defaultServerProvider();
    }

    switch (provider) {
    case JDK:
        if (enableOcsp) {
            throw new IllegalArgumentException("OCSP is not supported with this SslProvider: " + provider);
        }
        return new JdkSslServerContext(sslContextProvider,
                trustCertCollection, trustManagerFactory, keyCertChain, key, keyPassword,
                keyManagerFactory, ciphers, cipherFilter, apn, sessionCacheSize, sessionTimeout,
                clientAuth, protocols, startTls, keyStoreType);
    case OPENSSL:
        verifyNullSslContextProvider(provider, sslContextProvider);
        return new OpenSslServerContext(
                trustCertCollection, trustManagerFactory, keyCertChain, key, keyPassword,
                keyManagerFactory, ciphers, cipherFilter, apn, sessionCacheSize, sessionTimeout,
                clientAuth, protocols, startTls, enableOcsp, keyStoreType);
    case OPENSSL_REFCNT:
        verifyNullSslContextProvider(provider, sslContextProvider);
        return new ReferenceCountedOpenSslServerContext(
                trustCertCollection, trustManagerFactory, keyCertChain, key, keyPassword,
                keyManagerFactory, ciphers, cipherFilter, apn, sessionCacheSize, sessionTimeout,
                clientAuth, protocols, startTls, enableOcsp, keyStoreType);
    default:
        throw new Error(provider.toString());
    }
}
lContext › newServerContext()
```

那这个 OpenSslServerContext 的 `newHandler()` 方法生成的 SslHandler 里包裹的那个 SslEngine 到底是哪个实现类呢?

继续查看源码:

```
/**
 * This class will use a finalizer to ensure native resources are automatically cleaned up. To avoid finalizers
 * and manually release the native memory see {@link ReferenceCountedOpenSslContext}.
 */
public abstract class OpenSslContext extends ReferenceCountedOpenSslContext {
    OpenSslContext(Iterable<String> ciphers, CipherSuiteFilter cipherFilter, ApplicationProtocolConfig apnCfg,
                   long sessionCacheSize, long sessionTimeout, int mode, Certificate[] keyCertChain,
                   ClientAuth clientAuth, String[] protocols, boolean startTls, boolean enableOcsp)
            throws SSLException {
        super(ciphers, cipherFilter, apnCfg, sessionCacheSize, sessionTimeout, mode, keyCertChain,
              clientAuth, protocols, startTls, enableOcsp, leakDetection: false);
    }

    OpenSslContext(Iterable<String> ciphers, CipherSuiteFilter cipherFilter,
                   OpenSslApplicationProtocolNegotiator apn, long sessionCacheSize,
                   long sessionTimeout, int mode, Certificate[] keyCertChain,
                   ClientAuth clientAuth, String[] protocols, boolean startTls,
                   boolean enableOcsp) throws SSLException {
        super(ciphers, cipherFilter, apn, sessionCacheSize, sessionTimeout, mode, keyCertChain, clientAuth, protocols,
              startTls, enableOcsp, leakDetection: false);
    }

    @Override
    final SSLEngine newEngine0(ByteBufAllocator alloc, String peerHost, int peerPort, boolean jdkCompatibilityMode) {
        return new OpenSslEngine( context: this, alloc, peerHost, peerPort, jdkCompatibilityMode);
    }

    @Override
    /FinalizeDeclaration/
    protected final void finalize() throws Throwable {
        super.finalize();
        OpenSsl.releaseIfNeeded( counted: this);
    }
}
```

OpenSslServerContext

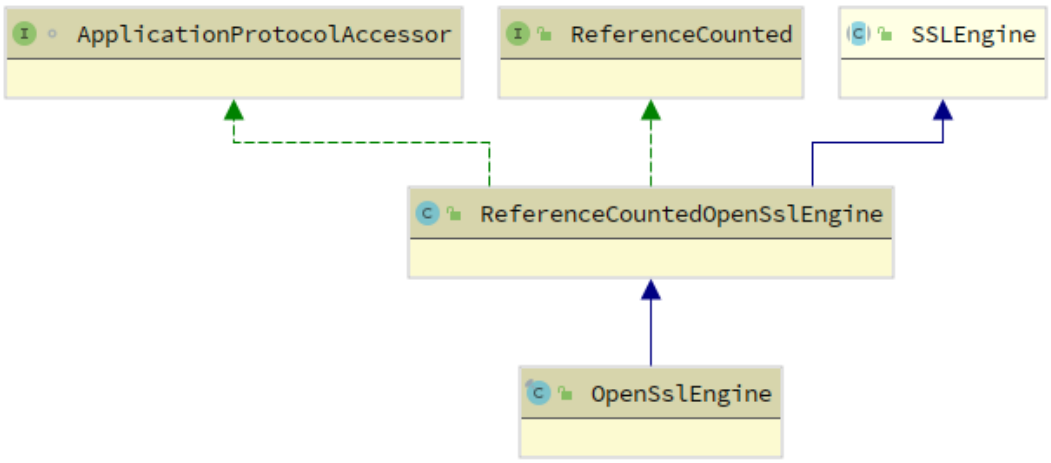
> 从截图还可以看到，这个 OpenSslContext 的回收比较“曲折”，需要从 Finalizer 那里兜一圈（`finalize()` 方法），堆外内存也是 Finalizer 来释放的（`OpenSsl.releaseIfNeeded()`），这个后面会提到，先留个印象。

> finalize 方法在 JDK 9 被标为 @Deprecated 了，这里的 Project 用的 Java 8，所以方法上没有“删除线”。

原来我们使用的是 OpenSslEngine。

申请时机

查看 OpenSslEngine 继承体系：



本身代码也很简单：

```
/**
 * Implements a {@link SSLEngine} using
 * <a href="https://www.openssl.org/docs/crypto/BIO_s_bio.html#EXAMPLE">OpenSSL BIO abstractions</a>.
 * <p>
 * This class will use a finalizer to ensure native resources are automatically cleaned up. To avoid finalizers
 * and manually release the native memory see {@link ReferenceCountedOpenSslEngine}.
 */
public final class OpenSslEngine extends ReferenceCountedOpenSslEngine {
    OpenSslEngine(OpenSslContext context, ByteBufAllocator alloc, String peerHost, int peerPort,
                  boolean jdkCompatibilityMode) {
        super(context, alloc, peerHost, peerPort, jdkCompatibilityMode, leakDetection: false);
    }

    @Override
    /FinalizeDeclaration/
    protected void finalize() throws Throwable {
        super.finalize();
        OpenSsl.releaseIfNeeded( counted: this);
    }
}
```

也重写了 `finalize()` 方法。

在构造器链中，我们找到了 `newSSL()` 被调用的时机：

```
Lock readerLock = context.ctxLock.readLock();
readerLock.lock();
final long finalSsl;
try {
    finalSsl = SSL.newSSL(context.ctx, !context.isClient());
} finally {
    readerLock.unlock();
}
```

释放时机

查看 `finalize()` 方法里的调用：

```
static void releaseIfNeeded(ReferenceCounted counted) {
    if (counted.refCnt() > 0) {
        ReferenceCountUtil.safeRelease(counted);
    }
}
```

OpenSslEngine 是 ReferenceCounted 的实现，这里会触发其重写后者定义的 `release()` 方法。

```
@Override
public final boolean release() {
    return refCnt.release();
}
```

看来 OpenSslEngine 自己又持有一个 ReferenceCounted 属性 refCnt：

```
private final AbstractReferenceCounted refCnt = new AbstractReferenceCounted() {
    @Override
    public ReferenceCounted touch(Object hint) {
        if (leak != null) {
            leak.record(hint);
        }

        return ReferenceCountedOpenSslEngine.this;
    }

    @Override
    protected void deallocate() {
        shutdown();
        if (leak != null) {
            boolean closed = leak.close( trackedObject: ReferenceCountedOpenSslEngine.this);
            assert closed;
        }
        parentContext.release();
    }
};
```

```
@Override
public boolean release(int decrement) {
    return handleRelease(updater.release( instance: this, decrement));
}

private boolean handleRelease(boolean result) {
    if (result) {
        deallocate();
    }
    return result;
}

/**
 * Called once {@link #refCnt()} is equals 0.
 */
protected abstract void deallocate();
```

看来是留了个模板方法，也就是说最终会调用 `deallocate()`。

我们最终在 `shutdown()` 中看到了释放堆外相关的代码：

```
/**
 * Destroys this engine.
 */
public final synchronized void shutdown() {
    if (!destroyed) {
        destroyed = true;
        engineMap.remove(ssl);
        SSL.freeSSL(ssl);
        ssl = networkBIO = 0;

        isInboundDone = outboundClosed = true;
    }

    // On shutdown clear all errors
    SSL.clearError();
}
```

这里的 `SSL.freeSSL()` 会将之前 `newSSL()` 时的内存释放掉。

阶段总结一下：

1. 增长的内存是 `SSL.newSSL()` 申请的堆外内存
2. 只有 new 构造 OpenSslEngine 时会触发申请
3. 在 GC 分析 OpenSslEngine 不可达后，Finalizer 机制会保证 `SSL.freeSSL()` 被调用

猜想

这里关于“为什么没有释放堆外内存”提出几个猜想：

1. OpenSslEngine 还有 GC Root 可达，导致其生命周期较长，迟迟不释放
2. OpenSslEngine 没有 GC Root 可达，但没被 GC 回收
3. OpenSslEngine 被 GC 回收了，但没成功释放堆外内存

排除猜想一、二

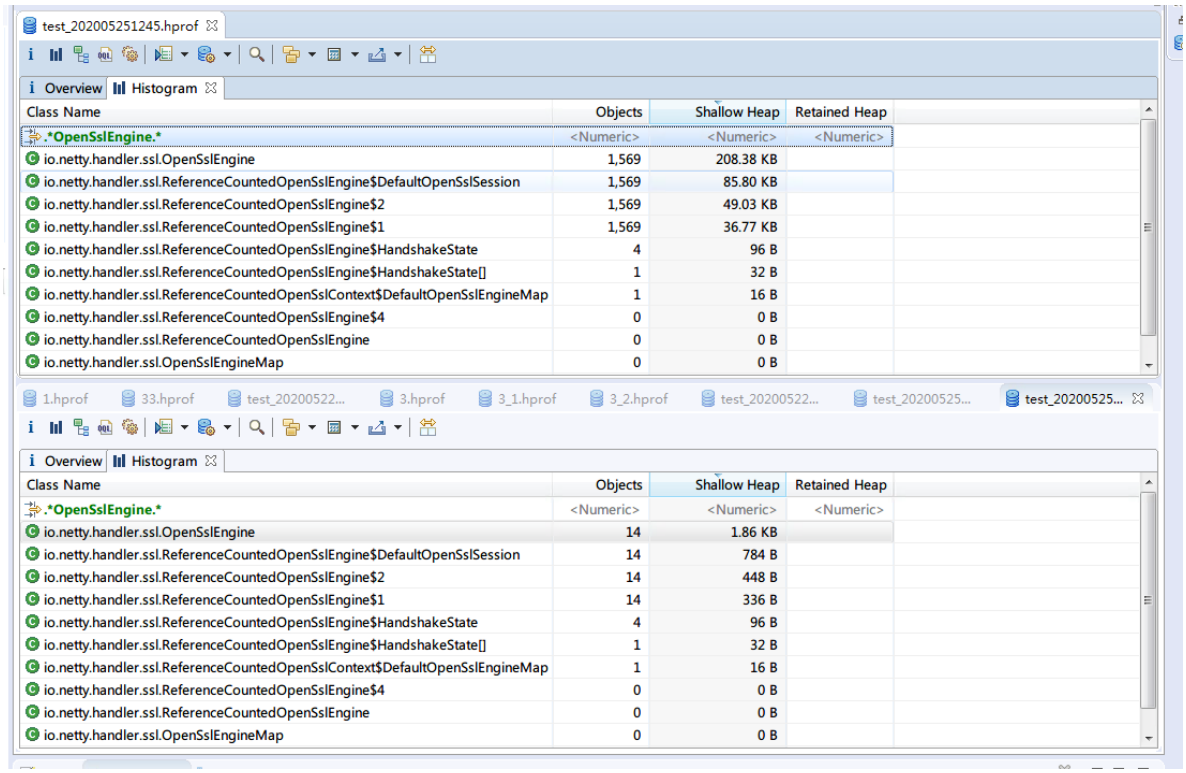
前两个猜想都可以简单的通过 dump 内存来分析确定。

```
jmap -dump:format=b,live,file=dump.hprof <pid>
```

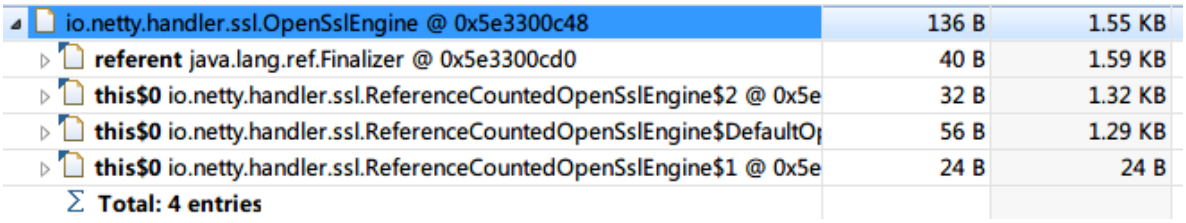
> 加 “live” 会触发 Full GC，否则得到的堆中可能存在大量可被下次 GC 回收的对象。

我们在测试环境 dump 两次内存，此时的连接数统计为 13 个。

```
[service@10-84-23-64.access-server-mqtt.test logs]$ jmap -dump:format=b,live,file=test_202005251245.hprof 338769
Dumping heap to /home/service/var/logs/mhopgup6tjy0/test_202005251245.hprof ...
Heap dump file created
[service@10-84-23-64.access-server-mqtt.test logs]$ jmap -dump:format=b,live,file=test_202005251245_2.hprof 338769
Dumping heap to /home/service/var/logs/mhopgup6tjy0/test_202005251245_2.hprof ...
Heap dump file created
```



可以看到，第一次 dump 得到的 OpenSslEngine 对象个数和此时的连接数相去甚远，很多对象都在 Finalizer 的队列中：



这些对象的 destroyed 为 true：

```java
/**
 * Destroys this engine.
 */
public final synchronized void shutdown() {
    if (!destroyed) {
        destroyed = true;
        engineMap.remove(ssl);
        SSL.freeSSL(ssl);
        ssl = networkBIO = 0;

        isInboundDone = outboundClosed = true;
    }

    // On shutdown clear all errors
    SSL.clearError();
}
```

Statics | Attributes | Class Hierarchy | Value

Type	Name	Value
ref	session	io.netty.handler.ssl.ReferenceCountedOpenSslEngine$2 @ 0x5e33..
ref	parentContext	io.netty.handler.ssl.OpenSslServerContext @ 0x5e0ef7348
ref	apn	io.netty.handler.ssl.ReferenceCountedOpenSslContext$2 @ 0x5e0..
ref	engineMap	io.netty.handler.ssl.ReferenceCountedOpenSslContext$DefaultOpe.
ref	alloc	io.netty.buffer.PooledByteBufAllocator @ 0x5e0957090
boolean	clientMode	false
boolean	jdkCompatibilityMode	false
boolean	outboundClosed	true
boolean	isInboundDone	true
ref	matchers	null
ref	sniHostNames	null
ref	algorithmConstraints	null
ref	endPointIdentificationAlgorithm	null
long	lastAccessed	1590373005256
ref	localCertificateChain	java.security.cert.X509Certificate[2] @ 0x5e3300d98
ref	clientAuth	io.netty.handler.ssl.ClientAuth @ 0x5e0f018d0
ref	refCnt	io.netty.handler.ssl.ReferenceCountedOpenSslEngine$1 @ 0x5e33..
ref	leak	null
boolean	needTask	false
ref	applicationProtocol	null
boolean	destroyed	true
boolean	receivedShutdown	false
ref	handshakeState	io.netty.handler.ssl.ReferenceCountedOpenSslEngine$Handshake...
long	networkBIO	0
long	ssl	0
int	peerPort	-1
ref	peerHost	null

No

这里可以比较清楚的看出，`finalize()` 方法确实被调用了。

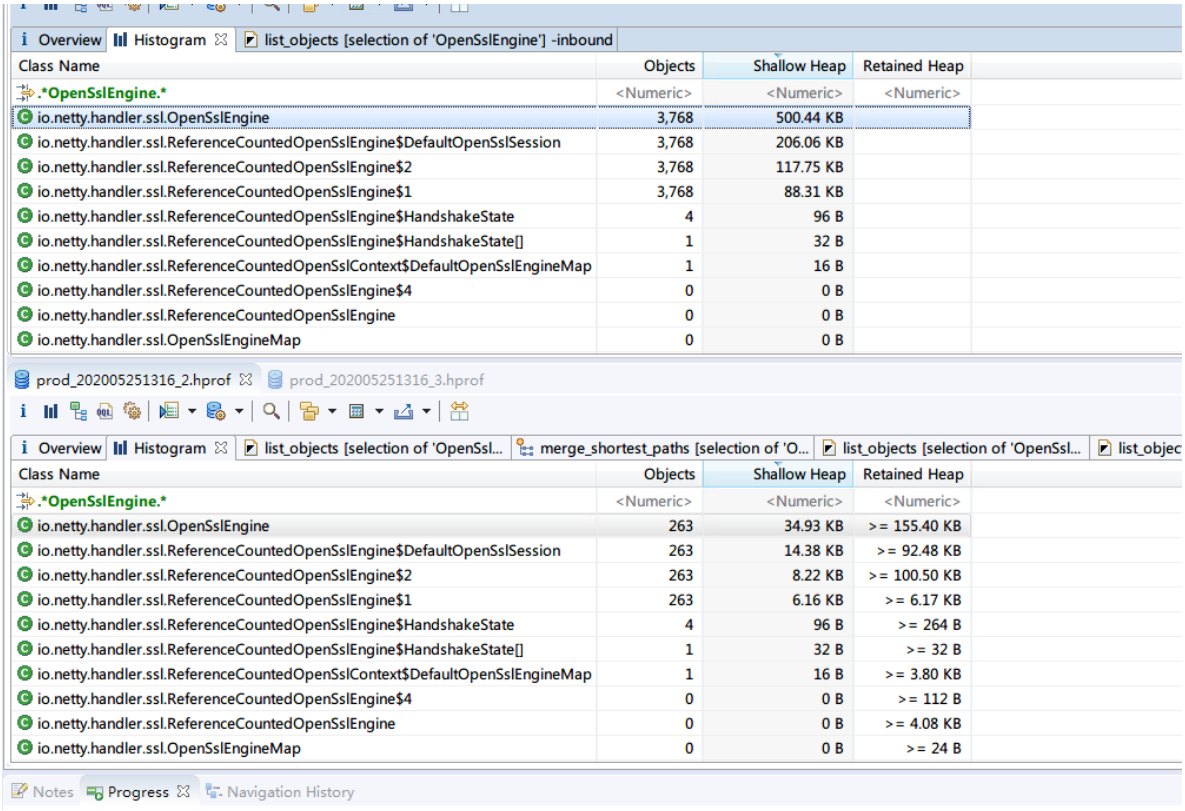
> 我们还发现这些 OpenSslEngine 对象的 Retained Heap 各不一样，已完成握手的大一些，未开始的小一些。

我们换到线上生产环境。

首先，我们确认线上无 Mixed GC：

```
[root@10-81-105-83.access-server-mqtt.bjht logs]# grep mixed gc_20200520170107.log
[root@10-81-105-83.access-server-mqtt.bjht logs]#
```

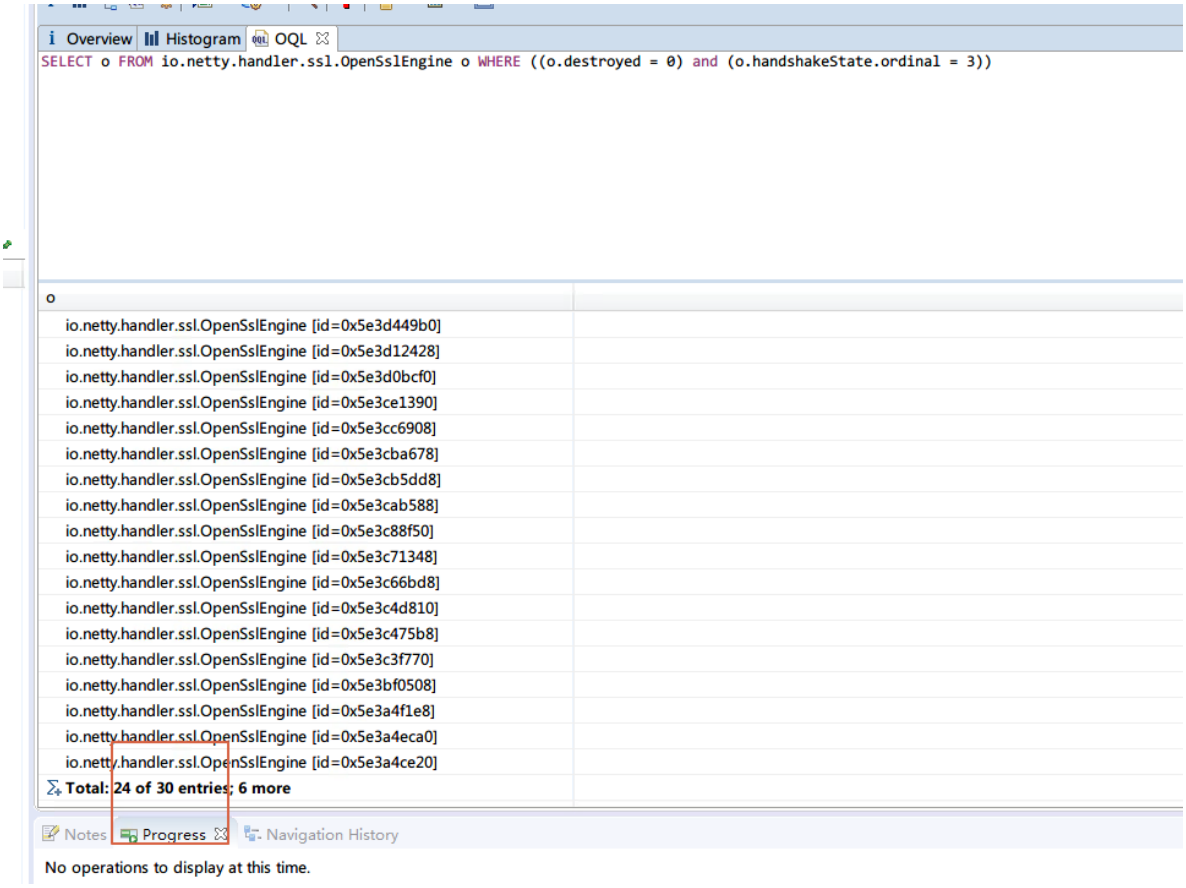
两次 FullGC 的前后对比：



dump 时连接数为 30 个。

> 这里不推荐在线上直接 dump，这里由于产品还在内测期间，连接数不多，所以就大胆 dump 了。

这里有个疑问，“为什么第二次 GC 后剩下 263 个而不是 30 个”，这是因为有些 destroyed 已为 true，如果我们过滤 destroyed 为 FALSE 的 OpenSslEngine 个数，正好是 30 个：

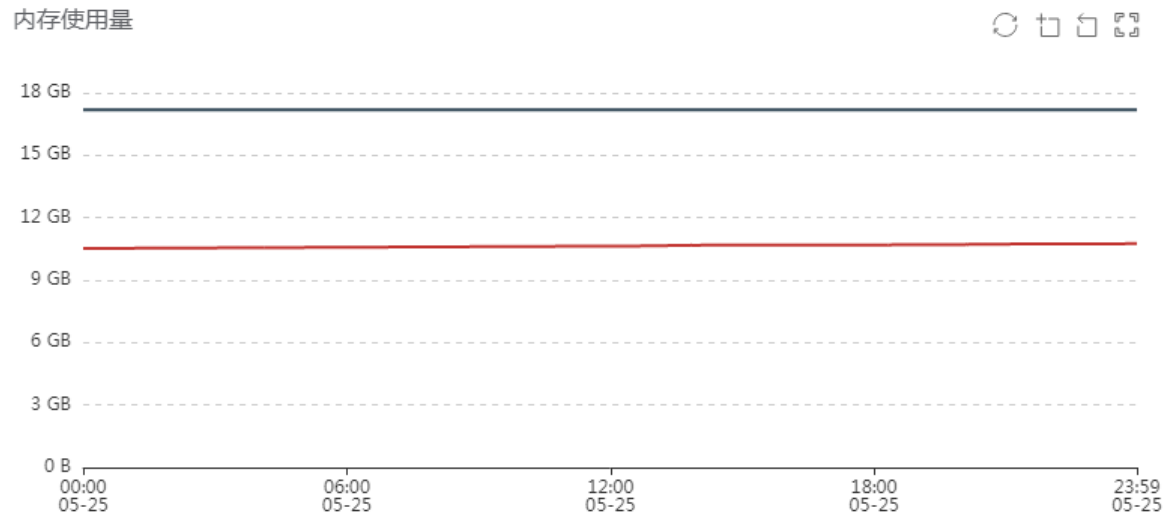


所以，如果一个连接连的时间足够长，其对应的 OpenSslEngine 对象已经成功晋升 Old Gen，那么在一个一直没有 Mixed GC 被触发的线上环境，是很可能堆外溢出的。

我们堆外和堆内的配比接近 1:1，那么，只要 OpenSslEngine 在堆外申请的内存远大于其在堆内占用的内存，这种情况就很有可能发生。

这里作为一个优化项日后完善。

可是，在 dump 两次后，线上的内存曲线却没有变化：



> dump 时间点大约为 12 点左右。

这样，基本可以确定猜想一、二不成立。

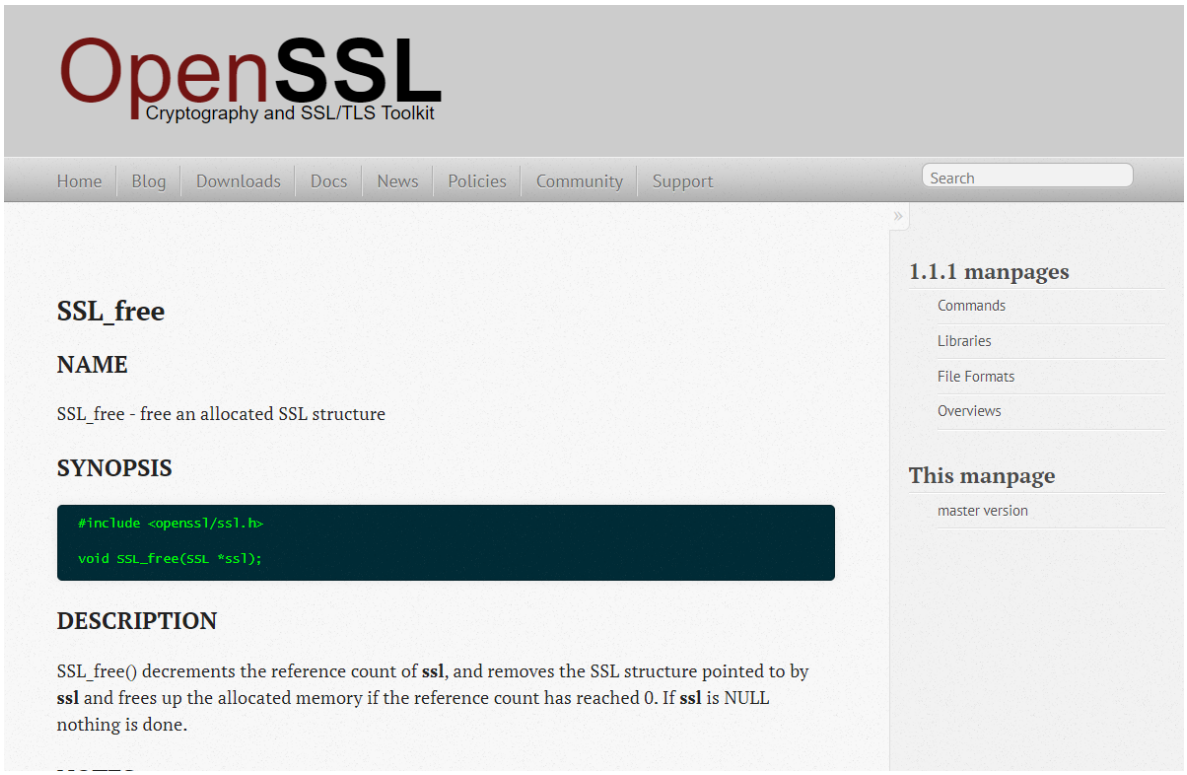
验证猜想三

我们查看 `SSL.freeSSL()` 源码。

```java
/**
 * SSL_free
 * @param ssl the SSL instance (SSL *)
 */
public static native void freeSSL(long ssl);
```

```
// Free the SSL * and its associated internal BIO
TCN_IMPLEMENT_CALL(void, SSL, freeSSL)(TCN_STDARGS,
                                       jlong ssl /* SSL * */) {
    SSL *ssl_ = J2P(ssl, SSL *);

    TCN_CHECK_NULL(ssl_, ssl, /* void */);

    free_ssl_state(tcn_SSL_get_app_state(ssl_));

    SSL_free(ssl_);
}
```

看来这里通过 `SSL_free` 释放了 ssl 结构。



我们看看 `OPENSSL_malloc()` 的 API 说明：

> OPENSSL_malloc(), OPENSSL_realloc(), and OPENSSL_free() are like the C malloc(), realloc(), and free() functions. OPENSSL_zalloc() calls memset() to zero the memory before returning.

看来还应该调 `OPENSSL_free()` 才对啊？查看 `free_ssl_state()` 发现"好像"有相应逻辑：

```
static void free_ssl_state(tcn_ssl_state_t* state) {
    JNIEnv* e = NULL;
    if (state == NULL) {
        return;
    }

    // Only free the verify_config if it is not shared with the SSLContext.
    if (state->verify_config != NULL && state->verify_config != &state->ctx->verify_config) {
        OPENSSL_free(state->verify_config);
        state->verify_config = NULL;
    }


    tcn_get_java_env(&e);
    tcn_ssl_task_free(e, state->ssl_task);
    state->ssl_task = NULL;
}
```

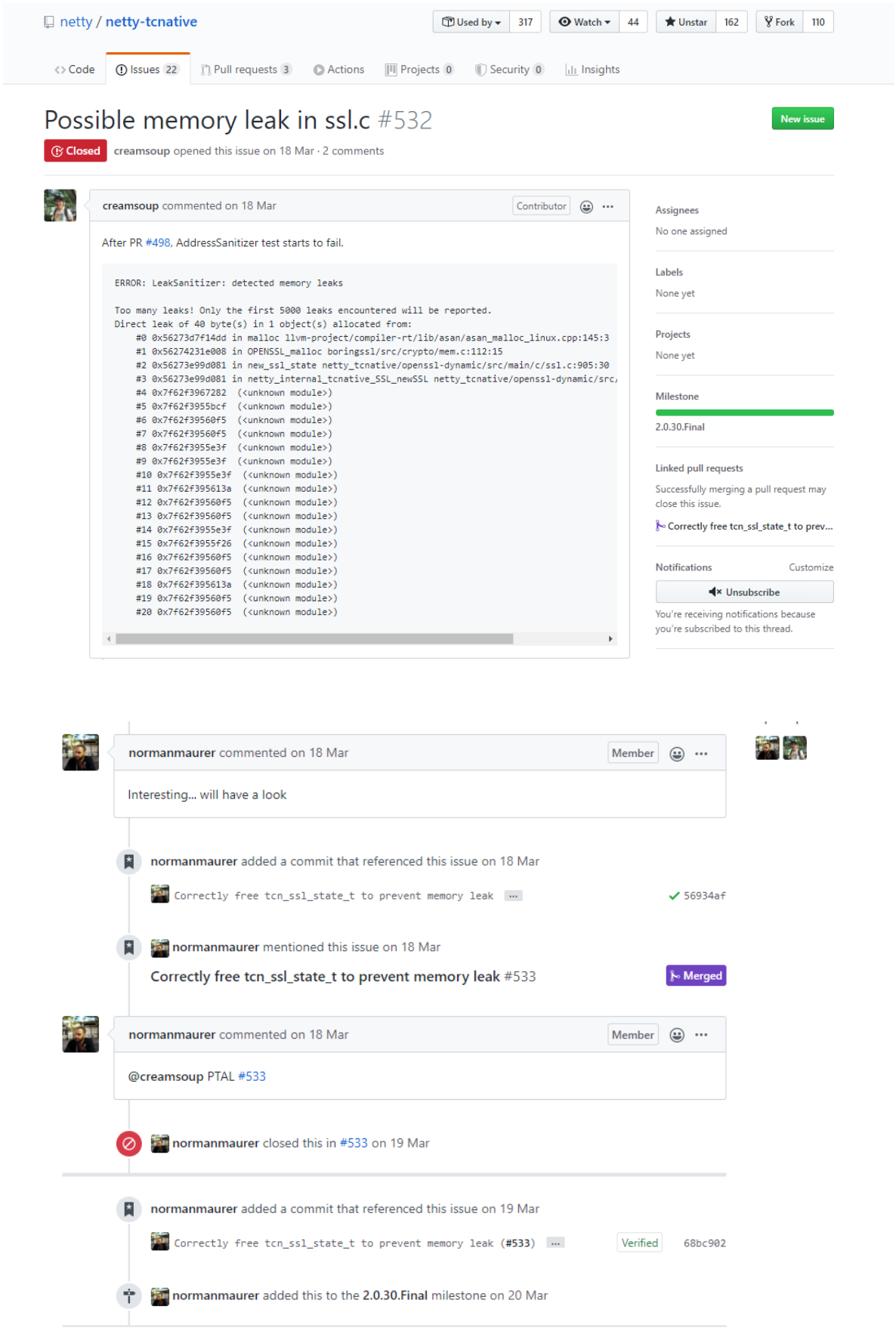
但是这里只释放了 verify_config，state 本身呢？

之前 reset 到了我们使用的 2.0.27.Final，现在我们 pull 一下源码看看有没有修改：

```
static void free_ssl_state(tcn_ssl_state_t* state) {
    JNIEnv* e = NULL;
    if (state == NULL) {
        return;
    }

    // Only free the verify_config if it is not shared with the SSLContext.
    if (state->verify_config != NULL && state->verify_config != &state->ctx->verify_config) {
        OPENSSL_free(state->verify_config);
        state->verify_config = NULL;
    }


    tcn_get_java_env(&e);
    tcn_ssl_task_free(e, state->ssl_task);
    state->ssl_task = NULL;

    // Free the tcn_ssl_state_t itself as it was allocated via OPENSSL_malloc(...) before
    //
    // https://github.com/netty/netty-tcnative/issues/532
    OPENSSL_free(state);
}
```

果然.....

查看相应 issue：



netty / netty-tcnative

Used by 317 | Watch 44 | Unstar 162 | Fork 110

Code | Issues 22 | Pull requests 3 | Actions | Projects 0 | Security 0 | Insights

Possible memory leak in ssl.c #532

New issue

Closed creamsoup opened this issue on 18 Mar · 2 comments

creamsoup commented on 18 Mar — Contributor

After PR #498, AddressSanitizer test starts to fail.

```
ERROR: LeakSanitizer: detected memory leaks

Too many leaks! Only the first 5000 leaks encountered will be reported.
Direct leak of 40 byte(s) in 1 object(s) allocated from:
    #0 0x56273d7f14dd in malloc llvm-project/compiler-rt/lib/asan/asan_malloc_linux.cpp:145:3
    #1 0x56274231e008 in OPENSSL_malloc boringssl/src/crypto/mem.c:112:15
    #2 0x56273e99d081 in new_ssl_state netty_tcnative/openssl-dynamic/src/main/c/ssl.c:905:30
    #3 0x56273e99d081 in netty_internal_tcnative_SSL_newSSL netty_tcnative/openssl-dynamic/src/
    #4 0x7f62f3967282  (<unknown module>)
    #5 0x7f62f3955bcf  (<unknown module>)
    #6 0x7f62f39560f5  (<unknown module>)
    #7 0x7f62f39560f5  (<unknown module>)
    #8 0x7f62f3955e3f  (<unknown module>)
    #9 0x7f62f3955e3f  (<unknown module>)
    #10 0x7f62f3955e3f  (<unknown module>)
    #11 0x7f62f395613a  (<unknown module>)
    #12 0x7f62f39560f5  (<unknown module>)
    #13 0x7f62f39560f5  (<unknown module>)
    #14 0x7f62f3955e3f  (<unknown module>)
    #15 0x7f62f3955f26  (<unknown module>)
    #16 0x7f62f39560f5  (<unknown module>)
    #17 0x7f62f39560f5  (<unknown module>)
    #18 0x7f62f395613a  (<unknown module>)
    #19 0x7f62f39560f5  (<unknown module>)
    #20 0x7f62f39560f5  (<unknown module>)
```

Assignees: No one assigned

Labels: None yet

Projects: None yet

Milestone: 2.0.30.Final

Linked pull requests: Successfully merging a pull request may close this issue. Correctly free tcn_ssl_state_t to prev...

Notifications: Customize — Unsubscribe

You're receiving notifications because you're subscribed to this thread.

normanmaurer commented on 18 Mar — Member

Interesting... will have a look

normanmaurer added a commit that referenced this issue on 18 Mar

Correctly free tcn_ssl_state_t to prevent memory leak ... ✓ 56934af

normanmaurer mentioned this issue on 18 Mar

Correctly free tcn_ssl_state_t to prevent memory leak #533 Merged

normanmaurer commented on 18 Mar — Member

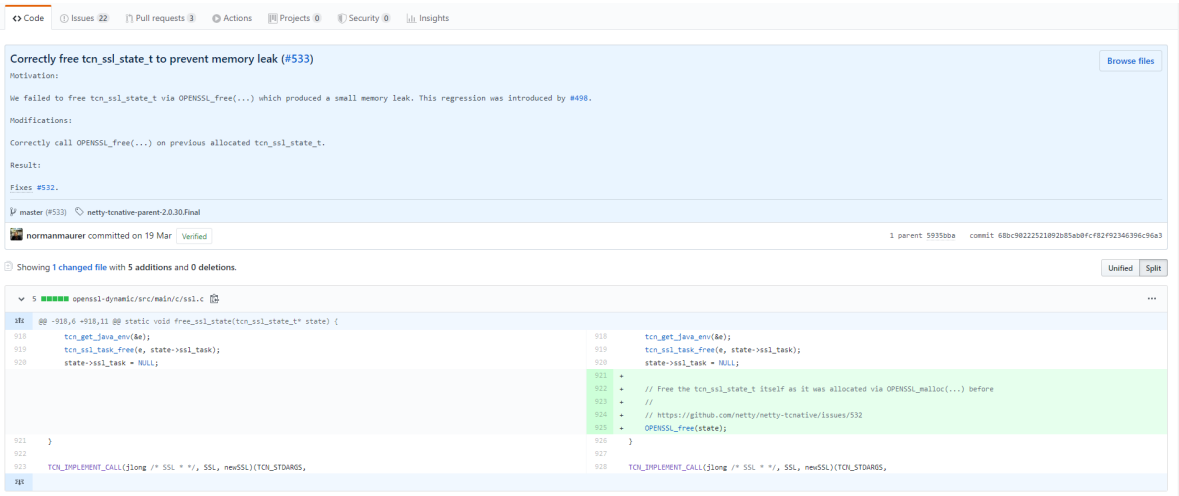
@creamsoup PTAL #533

normanmaurer closed this in #533 on 19 Mar

normanmaurer added a commit that referenced this issue on 19 Mar

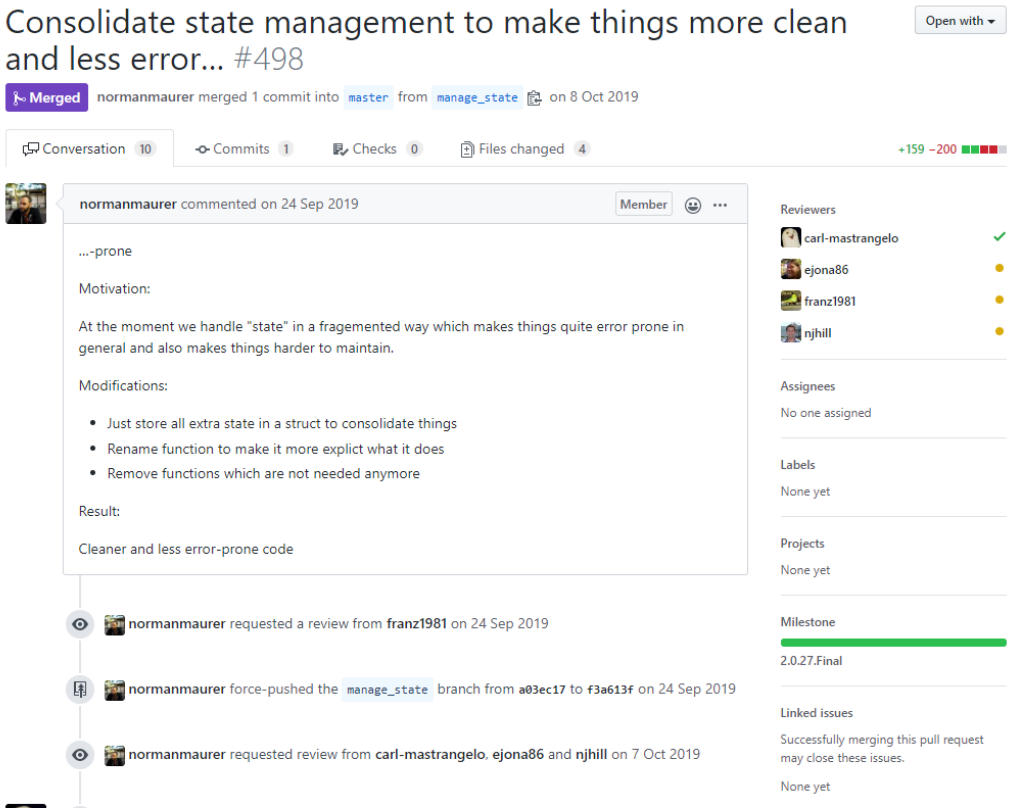
Correctly free tcn_ssl_state_t to prevent memory leak (#533) ... Verified 68bc902

normanmaurer added this to the **2.0.30.Final** milestone on 20 Mar



果然是个 bug，其在 2.0.30.Final 被修复了。

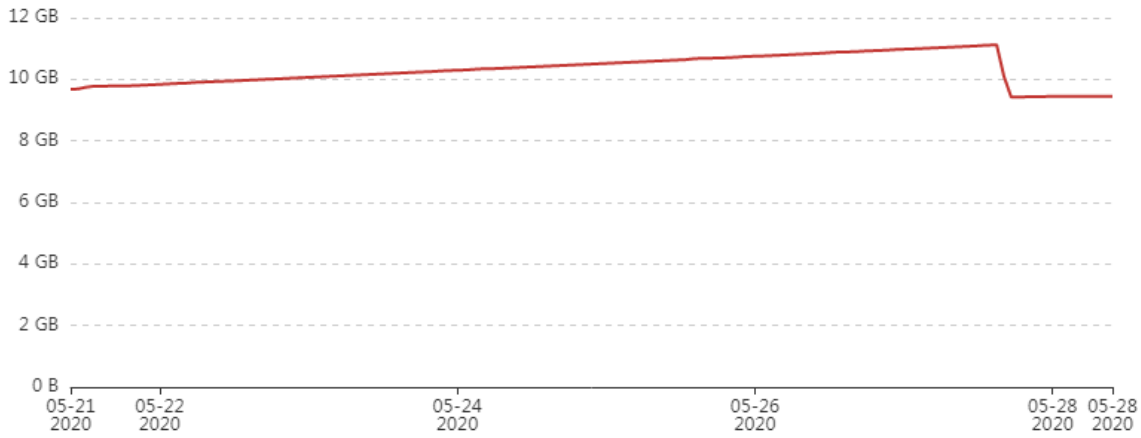
这个 bug 最早是由于 state 这部分代码不好维护，打算做一些小重构：



链接如下：https://github.com/netty/netty-tcnative/pull/498

结果重构过程中 bug 出现了，看起来 2.0.27.Final~2.0.29.Final 都有这个问题？我们将版本替换为 2.0.30.Final，问题解决：

内存使用量


12 GB
10 GB
8 GB
6 GB
4 GB
2 GB
0 B
05-21
2020
05-22
2020
05-24
2020
05-26
2020
05-28
2020
05-28
2020