系统初始化入口是JNI_CreateJavaVM函数，位于 `YourProjPath/hotspot/src/share/vm/prims/jni.cpp` 中，hotspot代码
的prims模块里面定义的就是一些外部接口，供JDK或者其他应用程序调用。
为了加深印象，我们先复习下JNI_CreateJavaVM函数是怎么来的。入口main函数中创建了运行JavaMain的主线程，在
程中会调用InitializeJVM来对虚拟机进行初始化，InitializeJVM中通过InvocationFunctions结构体中的函数指针来调用
CreateJavaVM，这个CreateJavaVM是通过 `dlsym(libjvm, "JNI_CreateJavaVM")` 获取到的。

系统初始化

JNI_CreateJavaVM函数中做的主要工作：
1.通过CAS判断vm_created变量是否为1，来保证当前只有一个JVM被创建
2.调用Threads::create_vm()来创建虚拟机
3.分别处理创建成功和失败的后续工作
所以想要知道真正的创建虚拟机的过程，我们需要去Threads::create_vm()中一探究竟。这个函数做了大量的工作，初始
绝大多数Hotspot内核模块。由于代码极其的长，贴上之后影响阅读，可自行去 `hotspot/src/share/vm/runtime/thread.`
下查看，从函数名称中可以大概看出都有哪些流程，接下来我们对其中一些重要步骤做详细分析。

初始化一些版本信息
初始化流输出模块
处理launcher属性
包括-Dsun.java.launcher和-Dsun.java.launcher.pid属性
初始化os
os::init()函数，位于 `hotspot/src/os/linux/vm/os_linux.cpp` ，用于初步初始化一些linux系统相关内容，主要包括：
1.初始化随机数产生器种子
2.设置内存页大小
3.初始化系统信息
4.获取原生主线程句柄
5.初始化时钟
初始化系统属性
与操作系统相关的属性，详细代码位于 `hotspot/src/share/vm/runtime/arguments.cpp` 中
的 `void Arguments::init_system_properties()` 函数
初始化JDK版本
初始化/升级JDK版本特定的系统属性
解析程序参数
初始化和应用ergonomics
jvm ergonomics主要用于根据机器配置选择JVM内存参数和gc策略等等。
init_before_ergo主要初始化了大内存页。
Arguments::apply_ergo()根据机器的内存、CPU等等硬件信息，初始化了一些JVM内存参数、gc策略、根据CPU不同选
同策略的BiasedLocking(偏向锁)等等内容，还初始化了JDK8取消持久带新增的Metaspace区。
启动timer，用于统计一些JVM启动信息
再次初始化os
os::init_2()函数，位于 `hotspot/src/os/linux/vm/os_linux.cpp` ，和os::init()不同的是，os::init()用于初始化一些固定配
os::init_2()根据外部传进来的参数来进行配置，os::init_2()初始化了以下内容：
1.初始化快速线程时钟
2.使用mmap分配一个只读单页内存供safepoint polling使用
safepoint是JVM中很重要的一个概念，在很多场景下都会看到，特别是在GC时。safepoint是指一些特定的位置，当线程
行到这些位置时，线程的一些状态可以被确定。
3.使用mmap分配memory_serialize_page
memory_serialize_page是用来在不使用memory barrier系指令的场景下模拟其操作，这样VM Thread可以在Java线程状
生变化时，及时获取到它们的状态，以正确地进行safe point时的管理。
4.初始化内核信号，安装信号处理handler
5.设置线程栈大小，设置线程初始栈
6.初始化libpthread
7.设置linux最大fd数量
8.设置用于串行化线程创建的时钟

9.若开启PerfAllowAtExitRegistration选项，向系统注册atexit函数
10.初始化线程优先级策略
以上可能有很多概念还不是很理解，这没关系，到后面随着一步步对hotspot了解的加深，很多问题都会豁然开朗，这里对hotspot启动时做了哪些工作有个大致了解即可。
配置好os后再做一步参数调整
初始化TLS

```
ThreadLocalStorage::init();
```

初始化输出流log
加载系统库
在解析JVM配置选项的时候，Arguments模块根据虚拟机选项-Xrun将要加载的本地库加入到_libraryList中，根据-agentli
agentpath选项将要加载的本地代理库加入到_agentList中，当这些库被加载进虚拟机进程后，虚拟机将在库中查找函数名
JVM_Onload或者Agent_OnLoad并调用该函数，实现这些库与虚拟机的连接。
初始化全局数据结构

```
void vm_init_globals() {
check_ThreadShadow();
basic_types_init();
eventlog_init();
mutex_init();
chunkpool_init();
perfMemory_init();
}
```

1.初始化java基本类型系统
2.初始化事件队列
3.初始化全局锁
4.初始化chunkpool
这是hotspot实现的内存池，包括_large_pool，_medium_pool，_small_pool和_tiny_pool，这样系统就不必执行malloc/f
5.初始化JVM性能统计数据区(PerfData)，由选项UsePerfData设置。
创建JavaThread
在openjdk中，一个Java线程背后涉及到许多数据结构，Java层面是java.lang.Thread，hotspot层面是JavaThread -> OSThread，native层面是pthread_create()创建的线程。
java中：

```
class Thread implements Runnable {
  private long           eetop;
}
```

eetop就是指向JavaThread的指针。
hotspot中，Thread用于记录平台无关信息，OSThread记录平台相关信息。

```
class Thread: public ThreadShadow {
OSThread* _osthread;

}
```

```
class JavaThread: public Thread {
}
```

有了以上知识，我们来看JavaThread的创建：

```
JavaThread* main_thread = new JavaThread();
main_thread->set_thread_state(_thread_in_vm);
main_thread->record_stack_base_and_size();
```

```
main_thread->initialize_thread_local_storage();
main_thread->set_active_handles(JNIHandleBlock::allocate_block());
if (!main_thread->set_as_starting_thread()) {
vm_shutdown_during_initialization(
  "Failed necessary internal allocation. Out of swap space");
delete main_thread;
*canTryAgain = false; // don't let caller call JNI_CreateJavaVM again
return JNI_ENOMEM;
}
main_thread->create_stack_guard_pages();
```

首先创建了一个JavaThread类型的main_thread，set_thread_state将线程状态设置为_thread_in_vm，表示该线程处于在JVM中执行的状态，record_stack_base_and_size记录线程栈的基址和大小，initialize_thread_local_storage初始化线程存储区TLS，set_active_handles为线程设置JNI句柄，set_as_starting_thread为JavaThread创建了OSThread，并Store pthread info into the OSThread，create_stack_guard_pages初始化这个线程栈。
初始化Java的对象监视器
java的关键字synchronized就是通过对象监视器实现的
init_globals()函数
这个函数实现了对全局模块的初始化，代码位于 `hotspot/src/share/vm/runtime/init.cpp` 。这些模块是Hotspot的整体础，后面会对这些模块分别分析
将创建的main_thread加入线程队列
创建VMThread
VMThread用于执行VMOptions，VMOptions实现了JVM内部的核心操作，为其他运行时模块以及外部程序接口服务。VMThread创建成功后不断等待、接受并执行指定的VMOptions。
初始化主要的JDK类
创建Java层Thread
在java.lang.Thread类初始化好之后，创建了java.lang.Thread，设置到刚才创建的JavaThread中
去， `main_thread->set_threadObj(thread_object);` ，同时这个main_thread也被设置给了java.lang.Thread的eetop。
到这里，java版的main线程就被创建好了，所以现在就是java.lang.Thread中有JavaThread，JavaThread中有OSThread OSThread中又有pthread也就是native thread，这里的pthread才是真正的可以被操作系统执行的线程，其他的都只是一构体，而且这里的pthread就是上一节中分析启动流程时说过的通过pthread_create的主线程，真正的java main函数在这程中执行。
创建一些守护进程
JVM初始化完毕，比较难的是弄清这些线程之间的关系。