回顾静态链接和动态链接

- 静态链接
  - 编译链接阶段，将目标文件和库文件全部打包进可执行文件。
  - 链接器在那时完成全部符号解析和地址重定位
- 动态链接
  - 不是一次性准备好一切
  - 程序启动时由loader加载，再交给动态链接器
  - 接着符号解析&&重定位，最后运行
- 依赖三个核心机制
  - PLT
  - GOT
  - Lazy Binding

芙莉莲的魔法课：链接的奥秘

性能：到底谁更强？

- 一个粗浅的结论：静态链接快，动态链接慢？
- 慢在哪里？
- 静态链接的性能优势
  - 链接时优化
  - 更少的间接寻址
  - 无需运行时的符号解析
- 适合嵌入式、实时系统、HPC等追求极致性能的应用。

性能：到底谁更强？

- 动态链接的开销？
  - 程序启动阶段，系统需要把共享库映射进内存。
  - 再扫描动态符号表，完成重定位。
  - 函数第一次被调用时，会先跳进PLT表，再进入动态链接器。
  - 找到真正的符号后，再patch回GOT，之后进入真正函数。
- 现代系统通过Lazy Binding，Cache等进行优化。
- 一旦进入steady state，运行期性能其实非常接近静态链接。
- 最大的差距体现在启动阶段，而不是运行阶段。

动态链接为啥这么复杂?

- PIC：位置无关代码
- 必须能被加载到任意地址，因此只能用相对寻址，通过GOT访问数据。
- 共享库可能被多个程序加载，若有硬编码的绝对地址，会出问题。
- 节省内存。
- 但是多出一次间接跳转，多一点寄存器压力，这都是可迁移性和性能之间清晰的工程取舍。

真正的工程难题：可靠性和ABI

- 静态链接的可靠性：它把世界封闭了。
- 即库永远维持它被链接后的样子。
- ABI：二进制层面接口规范
- 规定了函数调用约定（参数传递等），数据布局（对对齐等），系统调用约定，符号和库接口（如何链接）等。
- 动态链接的运行时加载共享库，主要依赖ABI保证。
- 真正危险的不是找不到库。
- 而是，ABI变了，程序还能运行，得到不可预测的结果(DLL/SO Hell)。
- 如何解决?

真正的工程难题：可靠性和ABI

- 非常优雅的解决：Symbol Versioning
- Linux的glibc（GNU C标准库）中常有类似以下代码：

```
printf@@GLIBC_2.2.5
printf@@GLIBC_2.27
```

- 让不同版本的程序能够安全共存和调用动态库。
- 老程序可以继续用旧版本
- 新程序使用新接口
- 系统可以持续升级而不崩溃，是Linux能持久演化的核心基础。

安全视角下的链接

- 静态更安全？动态更安全？
- 它们的安全性来源于什么？
- 静态链接：
  - 依赖封闭，攻击面小。
  - 但是一旦打包进去的库有漏洞，整个漏洞就会永远存在。
- 动态链接：
  - 给OpenSSL打补丁，给glibc修漏洞，使用它们的程序全部受益。
  - 但是loader的复杂逻辑容易被利用。
  - 动态解析链路，运行时临时找相关库，这个过程可被攻击。
  - 可利用的工具库多了，ROP组合可能就变多了。

安全视角下的链接

- 现代安全体系依赖 `动态链接+PIE+ASLR+RELRO` ，而不是某一边的绝对优势。

结论：软件开发与链接

- 事实上，应该按场景权衡动态链接和静态链接的优劣。
- 如嵌入式/实时系统，HPC，ios应用，多采用静态链接，保证性能和可预测性。
- Linux桌面运用，Android应用，插件系统，则必须采用动态链接，还有大量动态库，支持热更新等。
- 新的道路：Docker/容器化
  - 可以冻结整个运行环境，包括动态库版本
  - 仍然使用动态库的灵活性
  - 环境可控，消除动态库的不确定性，增强可预测性和可靠性
  - 同时享受动静态的优势
- 工业实践中，理解权衡，相互融合。

谢谢大家！
祝大家期末顺利！