5.1 简述异常机制能否由判断语句替代。

P44 第一段

举例。如果在错误发生的地方虽然知道了发生的是具体哪一类错误，但却没有足够的信息去将错误很好地呈现和处理，这时程序需要将错误向上传递，直到有足够的信息或者说到了从逻辑上讲应该处理该错误的地方再去处理该错误。异常机制里可以将异常往上抛，而判断语句很难做到这一点，即使能做到，也会使得逻辑结构十分混乱，使代码可读性差。由此可以看出，异常机制大大简化了解决问题的方式，远比判断语句更适合处理此类问题。

10.1 动态链接库的执行方式上有什么特点。

动态链接是相对于静态链接而言的。所谓静态链接是指把要调用的函数或者过程链接到可执行文件中，成为可执行文件的一部分。而动态链接所调用的函数代码并没有被拷贝到应用程序的可执行文件中去，而是仅仅在其中加入了所调用函数的描述信息。仅当应用程序被装入内存开始运行时，才在应用程序与相应的 DLL 之间建立链接关系。当要执行所调用 DLL 中的函数时，根据链接产生的重定位信息，Windows 才转去执行 DLL 中相应的函数代码。

P88 第三段：两种调用方式!!

P88 第二段

P88 第一段：使用 DLL 中的函数与程序自身的函数没有区别…

10.2 反射机制在程序中能起到什么作用

在没有 DLL 文件的源代码情况下使用 C#生成 类库，可应用反射机制获得 DLL 文件中的类方法和属性。

可以使用反射在运行时创建类型实例，调用和访问这些实例

（1）反射可以通过 System.Reflection 命名空间中的类以及 System.Type，可以获取有关已加载的程序集和在其中定义的类型（如类、接口和值类型）的信息

（3）使用 Assembly 定义和加载程序集，加载在程序集清单中列出的模块，以及从此程序集中查找类型并创建该类型的实例。

（4）使用 Module 获得包含模块的程序集以及模块中的类等。

（5）获取在模块上定义的所有全局方法或其他特定的非全局方法。

（6）使用 ConstructorInfo 获得构造函数的名称、参数、访问修饰符（如 public 或 private）和实现详细信息（如 abstract 或 virtual）等。

（7）使用 Type 的 GetConstructors 或 GetConstructor 方法来调用特定的构造函数。

（8）使用 MethodInfo 获得方法的名称、返回类型、参数、访问修饰符（如 public 或 private）和实现详细信息（如 abstract 或 virtual）等。

（9）使用 Type 的 GetMethods 或 GetMethod 方法来调用特定的方法。

（10）使用 FieldInfo 获得字段的名称、访问修饰符和实现详细信息（如 static）等；并获取或设置字段值。

（11）使用 EventInfo 获得事件的名称、事件处理程序数据类型、自定义属性、声明类型和反射类型等；并添加或移除事件处理程序。

（12）使用 PropertyInfo 获得属性的名称、数据类型、声明类型、反射类型和只读或可写状态等；并获取或设置属性值。

（13）使用 ParameterInfo 获得参数的名称、数据类型、参数是输入参数

还是输出参数，以及参数在方法签名中的位置等。

10.3 什么数据类型称为 blittable type

P92

10.4 托管代码调用非托管代码要注意哪些地方。

P92-93

10.5 什么是 DLL 地狱问题。

重新编译生成 DLL 后，用户程序使用新版本的 DLL 库不能正常工作称为 DLL 地狱问题。因为系统文件被覆盖而让整个系统像是掉进了地狱。

简单地讲， DLL Hell 是指当多个应用程序试图共享一个公用组件（如某个动态连接库（DLL）或某个组件对象模型（COM）类）时所引发的一系列问题。最典型的情况是，某个应用程序将要安装一个新版本的共享组件，而该组件与机器上的现有版本不向后兼容。虽然刚安装的应用程序运行正常，但原来依赖前一版本共享组件的应用程序也许已无法再工作。在某些情况下，问题的起因更加难以预料。比如，当用户浏览某些 Web 站点时会同时下载某个 Microsoft ActiveX® 控件。如果下载该控件，它将替换机器上原有的任何版本的控件。如果机器上的某个应用程序恰好使用该控件，则很可能也会停止工作。

这些问题的原因是（+P90 第二段）应用程序不同组件的版本信息没有由系统记录或加强。而且系统为某个应用程序所做的改变会影响机器上的所有应用程序—现在建立完全从变化中隔离出来的应用程序并不容易。

尽量避免 DLL 地狱的方法包括：不直接生成类的实例，不直接访问成员变量，不使用虚函数。

11.1 程序的并发与并行区别是什么

程序的并发是指程序在某一时间段内同时处理多个事务的运行过程。程序并发在宏观上表现为并发运行的形式，在微观上则是串行。

程序的并行是指多个线程在微观上同一时刻在 CPU 的两个或多个核上运行，只有多核或者多 CPU 的机器中才能实现。

并行实际上是指计算机同时做多个任务，而并发则是计算机交替做多个任务。

11.2 HANDEL 值是否就是内核对象的内存指针地址

不是。HANDEL(句柄)在 WinNT.h 中的定义为 typedef PVOID HANDLE

句柄不是系统中对象的指针，而是用于标识对象的特殊值，应用程序通过句柄访问相应对象的信息。

Windows 系统的内存管理经常释放空闲对象内存，再访问时重新提交，对象的物理地址经常变化。因此操作系统禁止程序用物理地址去访问对象内容。
句柄不是对象的指针，虽然句柄的值是确定的，但是必须使用指定的 API 才可操作句柄指向的对象。

系统在进程初始化时分配一个当前进程所有内核的句柄表。内核对象创建函数返回句柄值。

HANDLE 值是进程相关的，同一个 HANDLE 值在不同的进程中意义是不一样的，不代表相同的内核对象。这进一步说明了 HANDLE 值和内存指针地址不同。

13.1 请举例说明最顶窗体可以不是激活的窗体。

最顶窗体是针对窗体的前后关系而言的。Windows 系统通过垂直于屏幕的 Z 坐标给每个对象设置一个 Z-Order 值，用来表示窗体重叠的前后次序，最顶的窗体覆盖所有其他窗体，最低的窗体则被所有窗体覆盖，最顶的窗体具有

WS_EX_TOPMOST 样式值。

激活窗体是针对系统消息队列分派目标而言的。操作系统把用户的输入统一处理为消息并放入系统消息队列，派送到当前激活窗体，隐藏状态的窗体不接受用户输入。Windows 操作系统仅有一个激活窗体，用户通过鼠标、键盘产生的信息都发送到激活窗体。

由此可见，最顶窗体和激活窗体是两个不同的概念，最顶窗体可以不是激活窗体，一个具体的例子就是软键盘窗体。

在软键盘窗体程序中，软键盘需要始终展示在最前端覆盖其他窗体，因此软键盘是一个最顶窗体。但是当用户点击软件盘上的按钮时，软键盘窗体不会获得焦点，而且此时会向程序中真正激活的窗体(可能是一个写字版窗体)发送字符。另外，此时用户的鼠标、键盘的输入也都会被发送到真正激活窗体而不是软键盘窗体。所以软键盘窗体虽然是一个最顶窗体，却并不是一个激活窗体。

补充：

1. DLL 和 COM 的区别：PPT-COM
2. 简述 Windows 应用程序中的消息机制。
   P123
3. 简述 Windows 应用程序中的事件机制
   重点在于事件机制内部的消息机制，主程序收到触发事件后发送消息等。
   事件类型、触发事件、事件处理函数
4. 事件的实现：PPT

```
 public class FireAlarm  {
//将火情处理定义为 FireEventHandler 代理(delegate) 类型，这个代理声明的事件的参数列表
    public delegate void FireEventHandler(object sender, FireEventArgs fe);
//定义 FireEvent 为 FireEventHandler delegate 事件(event) 类型.
    public event FireEventHandler FireEvent;
  //激活事件的方法，创建了 FireEventArgs 对象，发起事件，并将事件参数对象传递过去
   public void ActivateFireAlarm(string room, int ferocity)   {
        FireEventArgs fireArgs = new FireEventArgs(room, ferocity);
       //执行对象事件处理函数指针，必须保证处理函数要和声明代理时的参数列表相同
       FireEvent(this, fireArgs);
   }
}
```

5. 非托管 DLL 的创建和使用

1)　使用 C++创建类库(DLL)

a. 创建应用程序类型为 DLL 的项目

b. 添加头文件*.h，将 API 声明为外部方法

```
extern "C" _declspec(dllexport) int __stdcall test(int a, int b);
```

c. 修改源文件*.cpp，在.cpp 文件中 include 头文件*.h，并编写 API 函数体

```
int __stdcall test(int a, int b) {return a * b;}
```

d. 添加导出定义*.def,并添加以下内容

```
LIBRARY
EXPORTS
test @ 1
```

e. 编译生成dll文件

先将解决方案切换到Release模式，再在项目名称上右击选择【生成】或【重新生成】

f. 使用dll函数查看器查看导出函数和参数是否正确

2）**C#项目调用C++创建的DLL**

将生成的dll放到调用项目的Debug目录下

a. **C#项目中定义DllImport**

```
[DllImport(@"../../../Release/CreateDLL.dll", EntryPoint = "test",
SetLastError = true,CharSet = CharSet.Ansi, ExactSpelling = false,
CallingConvention = CallingConvention.StdCall)]
        public static extern int test(int a, int b);
```

b. 调用api，Console.WriteLine(test(1, 2).ToString());

3）. **C#项目中调试c++项目**

a.在C#工程右键【属性】->【调试】->【启动调试器】中选中【启动本机代码调试】

b. 在C++项目的源码中设置断点，在Debug模式下运行C#程序会自动跳到断点处

1) C#创建DLL

a.创建输出类型为类库的C#项目，在cs文件中定义类和方法

```
public class Class1
    {
        public int test(int a, int b)
         {
             return a * b;
         }
    }
```

b.编译生成dll文件

将解决方案切换到Release模式，再在项目名称上右击选择【生成】

2）　C#项目调用C#创建的DLL

a.项目添加引用，找到DLL文件确认添加

b. c文件引入类库命名空间

```
using CSharpDLL;
```

c. 调用api.

```
CSharpDLL.Class1 class1 = new Class1();
Console.WriteLine(class1.test(1, 2).ToString());
```

3）调用 DLL 的项目可在属性管理器，开启本地断点调试