正常和异常

程序执行中可能遇到非正常情况，正常控制流无法继续，需要特殊处理

出现非正常情况的原因可能是由于环境因素（不符合要求的外部输入或用户交互），也可能是程序不同部分间的相互作用

常见实例：存储分配申请无法得到满足，读入数据时找不到媒介或读入出错，调用读栈操作时遇到栈空，压栈操作遇到栈满等等

遇到这类情况时应该如何处理呢？看一个实际例子（数据结构）：

```
int stack::top() {
   if (empty()) {
      // 怎么办?
   }
   ... ...
}
```

常见处理方式1（教科书中常见）：

```
int stack::top() {
   if (empty()) exit(1);
   ... ...
}
```

错误！不能用于实际的程序设计

底层服务绝不能自主决定终止程序，它不掌握做这种决策的信息，也没有这种权力

异常情况和处理

处理方式2（教科书中常见）：

```
int stack::top() {
   if (empty())
     cout << "Stack empty";
   ... ...
}
```

错误！完全不能用于实际程序设计

产生输出信息后继续运行，程序已进入非正常状态，后果无法预料

子程序向程序使用者报告，而人通常无法直接干预程序的内部执行

处理方式3（教科书中可见）：

```
int stack::top() {
   assert(!empty());
   ... ...
}
```

- 通常用于帮助查找错误，在调试执行中出错时assert 产生错误报告并终止程序
- 通常在生成执行代码时关闭断言（定义宏NDEBUD，否则类似于用 exit）
- 如果运行中出现访问空栈的情况：
  - 不关闭断言，效果同上面处理方式1
  - 关闭断言，效果同上面处理方式2

异常情况和处理

处理方式 4（教科书中可见）：

```
int stack::top() {
   if (empty()) return ECODE1;
   ... ...
}
```

使用这种函数：

```
if((n=s.top()) == ECODE1)
      ... // 出错处理
else ... // 正常处理
```

方法正确

- 下层检查错误情况并报告
- 上层检查完成状态并处理

但：

- 能否找到合适的错误码？
- 程序中很难做充分的检查

缺点：如果每个函数都设置出错返回值，在每个调用位置都检查，将严重干扰程序的正常控制流

处理错误的代码和正常处理过程混在一起，程序会变得无法控制。处理逻辑变得很不清晰，使程序难以开发，难以阅读

异常情况和处理

另一常见技术：设置专用于记录执行状态的全局变量，执行中遇异常情况时给变量赋特殊值。程序其他部分可以检查这个变量，并根据情况处理

例：C标准库定义了**errno**，初值为 **0** 表示无异常。一些库函数（特别是数学函数）出现异常情况时把**errno**设为非**0**。检查它就能确定执行情况

问题与返回出错码的方式类似：

- 每时每刻检查，严重干扰正常处理部分的开发、理解和维护
- 不及时检查就可能忽略实际出现的错误

可见：常规的描述手段对错误处理描述的支持不足

1980年代末一项研究显示，程序里处理错误的代码可达总代码量的**2/3**

大量处理异常情况的代码与正常控制流代码混在一起，极大地影响着程序的开发，破坏程序的可读性、可理解性和可维护性

异常情况和处理

问题：

- 发生异常是低级事件，发现异常的通常是底层模块。如硬件、基础运行系统、库模块、底层服务模块。异常的正确处理只能根据应用的需要来确定，只有适当的上层模块才可能知道正确合理的处理方式
- 检查发现异常情况的代码与能处理它的代码之间可能跨越多层调用（按常规的控制流，控制转移的最大步骤是子程序调用和返回）

C 语言标准库里为处理非正常情况提供了两套机制：

- **setjmp/longjmp** 机制。主要功能：保存执行现场，以支持在随后的执行中（主要是从嵌套调用的子程序里）直接跳回前面保存的现场，这时可以换一条路径重新执行（例如处理底层发现的运行异常）
- **signal** 机制。主要功能：可针对一些“信号”定义特殊的处理函数（信号处理器），在信号被引发时自动执行对应处理器。C 标准库定义了若干标准的信号（会自动引发），可定制针对它们的处理器

一些环境提供了类似机制。这类机制的共同缺陷是太低级，而且不安全

异常情况和处理

由于异常（包括错误和其他一些特殊情况）的处理很重要，编程中需要大量写这方面的代码，因此应该在语言层面上为之提供支持

异常处理的目标是尽可能挽救已经出问题的执行过程，使之有可能继续进行下去（换一种方式？抛弃某些枝节？弱化某些功能？等等）

对异常情况的处理，最常见的处理模式是：在正常处理中发现异常情况——控制转至特殊处理阶段——处理完成——返回正常控制流

对语言层解决方案的一些要求：

- 能比较自然地反映从异常检查代码到处理代码，以及从异常处理代码回到正常执行过程的控制转移。将这些转移屏蔽在语言之下自动进行
- 有清晰的语义模型，程序员容易理解，容易正确使用
- 正常控制流与异常处理控制流相互隔离
- 有机地融合到现有的基本语言机制中（以上都是语法/语义方面的考虑）
- 效率较高，不出现异常情况时最好是不影响程序的执行（效率）

异常处理

主流语言里最早的专用错误处理机制是PL/1语言的ON语句。程序出现异常时自动转到相应 ON 语句，处理完毕后通过 goto 转出（控制流很难把握）

1975 年 J. Goodenough 在 CACM 发表论文提出了结构化异常处理的概念，建议的基本处理框架：把异常处理结构附在正常结构上，运行中出现异常时控制自动转过去处理异常，处理完成后回到正常控制流

异常处理机制的研究和发展基本上都是沿着这个方向进行的

异常处理的第一个想法是区分两种控制流：

- 正常控制结构描述正常控制流；异常控制结构描述出现异常时的处理过程
- 定义良好的在两种控制结构之间的自动转移规则
  - 在正常控制结构中出现异常情况，控制自动转入异常控制结构
  - 异常处理完成，运行自动转回正常控制结构（怎么转？转到哪？）
- 结构化异常处理：将异常控制结构附着在某些正常控制结构上，由它们来捕捉所附着的正常结构的执行中发生的异常

异常处理

- 1970年代中后期，人们在 Clu 和 Mesa 语言的开发中对结构化异常处理机制及其实现机制进行了深入研究
- Mesa 提供了非常灵活强大的异常处理机制，人们在其中研究了
  - 异常的自动传播，
  - 异常处理的“终止模型”和“唤醒模型”等
- Clu 采用语义较简单明晰的终止模型，提出了过程/函数的“异常描述”的概念和记法（C++/Java 等的这方面特征设计都从 Clu 汲取了经验）
- 在 Clu 和 Mesa 里，异常传播时可以携带任意复杂的状态信息
- Ada 语言的设计参考了上述经验，其中采用了
  - 较简单清晰的终止语义的异常处理机制
  - 只允许异常携带简单信息

后续主流语言都以类似方式提供异常处理机制，使之成为语言“标准配置”

异常处理模型　　假定异常在函数G中引发，在函数F中捕捉和处理

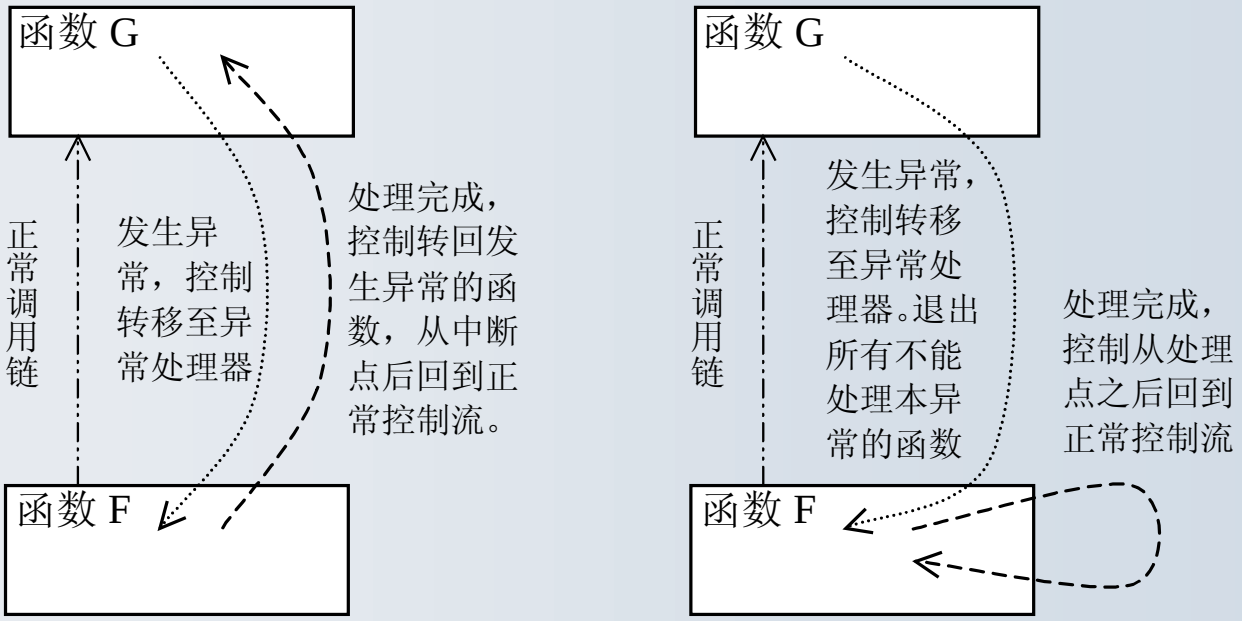


图 1，异常处理的唤醒模型（左）和终止模型（右）

理论和实践说明：唤醒模型太复杂，很难写好。新语言都采用终止模型

异常处理机制的要素

异常处理机制的要素，不同语言有各自的选择：

- 异常引发（**raise**，抛出 **throw**）机制（自动引发，程序引发）
- 预定义异常（有/无），什么机制负责引发这些异常
- 异常捕捉机制（以及监视和捕捉范围）
- 处理器附着位置（表达式/语句/子程序/特定结构）
- 异常辨识和确认：如何决定一个处理单元能否处理当时发生的异常（基于符号，基于类型，等）
- 能否（及如何）捕捉多种异常或者任何异常
- 异常传播时的附加信息传递（无、简单、任意）
- 程序单元（函数/过程等）的异常特征描述
- 异常处理模型（“终止”或“唤醒”）
- 异常的作用域特征（如果传播出定义异常的作用域，怎么办？）

C++ 异常处理

C++ 用 try 块（特定结构）描述异常检查范围，用后附的 catch 子句（异常处理器）描述异常的捕捉和处理。通过 throw 语句抛出异常

```
int f (... ...) // 简单实例
try {... ...
    try {... ...
        throw E(... ...);
        ... ...
    }
    catch (A a) { ... ... }
    catch (B& b) { ... ... }
    ... ...
}
catch (const C& c) { ... ... }
catch (D d) { ... ... }
```

函数体就是一个 try 块

后附异常处理器

可以有带着自己的异常处理器的内嵌 try 块（任意嵌套）

throw E(*e*); 抛出异常，用表达式 *e* 初始化一个匿名的临时对象（异常对象）。异常对象的类型没有限制，可携带任何信息

异常匹配基于类型进行。异常处理器用类型描述其捕捉对象，借助类层次结构，一个处理器可以捕捉多种不同异常

C++ 异常处理

1. catch(A) 捕捉类型为A，处理器里不引用异常对象
2. catch(A a) 捕捉类型为A，并用异常对象初始化局部参数 a
3. catch(const A& a) 建立对异常对象的 const 引用
4. catch(A& a) 建立对异常对象的引用
5. catch(...) 捕捉一切异常，处理器里不引用异常对象

参数类型为A的异常处理器捕捉以A为public基类的所有子类的异常对象

try块抛出的 e 由块后处理器顺序检查。若有处理器能处理 e则控制转入；处理完成后转回正常控制流（“终止模式”）。允许再抛出（支持分步处理）

若 try 块的处理器不能捕捉抛出的 e，e传到外围try块。若当前函数不能处理 e，函数结束，e 在函数调用点再次抛出。（异常在子程序间传播）

未被捕捉的异常最终导致程序结束。 C++ 运行系统不会抛出异常。基本运行错（内存违规、算术错等）不会转为异常，不能通过异常机制处理

异常处理机制：实例

C++ 没有语言预定义的异常

- 标准库定义了异常类（示例），程序可以抛出和捕捉任何类型的对象

Java 的异常处理机制与 C++ 类似，但

- 只能抛出 Throwable 类（及其子类）的对象作为异常对象
- 有预定义的异常类层次结构
- 捕捉子句的参数形式没有多种变化

Ada 采用基于名字的异常抛出和捕捉机制

- 语言定义了一组内部异常，自动抛出（如值溢出、违规访问等）
- 程序里可声明新的异常名（遵循作用域规则）
- 程序里可引发自定义异常，也可引发系统异常
- 异常的匹配是简单的名字匹配
- 定义了异常传播出作用域的语义，提供了捕捉一切异常的结构

异常处理

从控制的角度看，异常可看作一种能够跨过子程序边界的控制转移机制。其最终转移目标由动态调用链的当时情况动态确定

```
void G (int n) {
  ... if (...) throw E(); ...
}
void H (int n) {
  try { ... G(n); ... }
  catch (E e) { ... ... }
}
void F (int n) {
  try {... H(3);... G(4);...}
  catch (E e) { ... ... }
}
```

假定 F 执行期间调用的函数 G 里抛出 E 异常

这个异常可能由 F 里或者 H 里的相应处理器处理，到底由那个处理器处理，根据当时的调用链情况确定

异常与其处理器的匹配，完全根据动态运行时的情况确定

异常处理：实现

现在考虑终止模式的实现技术（唤醒模式已经基本被抛弃了）

动态链方法：

最直接实现方法是在程序运行中维护一个异常处理器链表：

- 程序执行进入一个受监视区域（如 C++ 的 try 块）时，把关联于该区域的各异常处理器的信息结点按规定顺序加在链表前端
- 执行退出受监视区域时，删除链表中与该区域有关的处理器结点
- 这个链表实际上是个栈，加入删除异常处理器结点按后进先出方式，但通常是成组压入弹出。可以在运行栈上实现（增加栈帧里的信息）

一旦运行中抛出异常 E，正常执行流终止，控制转移入异常处理流程

异常管理程序沿处理器链表检查，查找第一个与 E 匹配的异常处理器

在查找处理器的过程中，还要确定和处理尚处于活动状态但因为异常而需要退出的子程序，完成这些子程序退出时应完成的所有动作

异常处理：实现

动态链结点结构图（作为独立结构，可以考虑将其嵌入运行栈）

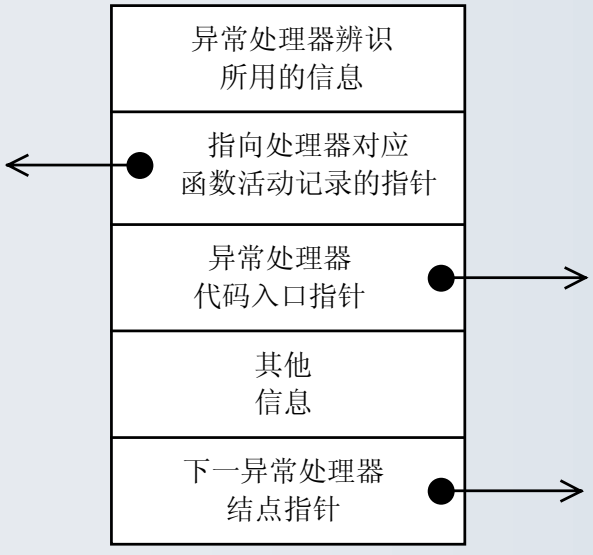


图 3，异常处理器链表结点

采用动态链实现：

每次控制流进出try块都有系统开销（维护处理器链表）

即使运行中不出现异常，异常处理机制的存在也带来运行开销

另一方面（是必然的）

出现异常后查寻处理器的时间代价与需要匹配的处理器个数成正比

如果有深层调用，如深度递归调用，查寻处理器可能很费时间，造成程序执行的一段停顿

异常处理：实现

静态表（字典）方法：

因为程序的目标代码是静态的，运行中不变。因此可以静态构造出一个表格，其中描述各异常监视区域对应的异常处理器（都可以静态确定）

- 表项的索引是被监视区域的地址范围，关联数据中包含用于确定异常处理器的有关信息（与动态链结点中的信息类似）
- 为保证查询的效率，表项按监视区的地址排序，可用二分法查找

若运行中抛出异常 **E**，以抛出点的指令计数器 **IP** 的值查询异常处理表格

- 先确定是否存在对应的处理器（该地址是否位于某个被监视区域中）
- 如果有，再看与该区域关联处理器能否处理异常 **E**
- 如果引发所在的子程序里不能处理**E**，该子程序退出（执行退出动作）
- 退出一层子程序后，**E** 在子程序调用的返回点再次引发，并以这一点的 **IP** 值查找异常处理表格

异常处理：实现

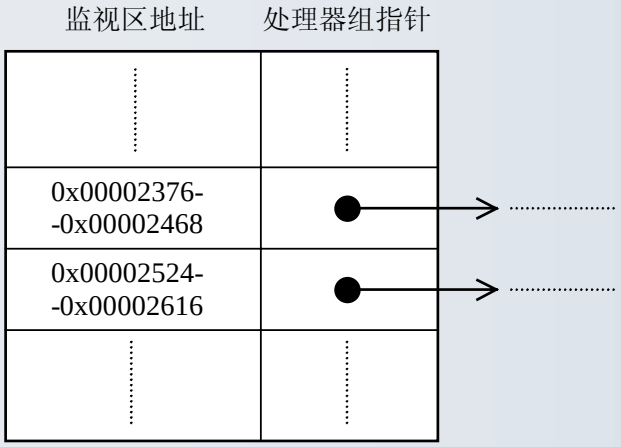


图 4，静态表格数据结构

静态表的基本数据结构大致如图

左边是基于各监视区域创建的索引项，指针指向监视该区域的异常处理器序列的信息结点

索引项按地址排序。

静态表法的最大优点是所用的表格是静态构造好的

如果程序执行中没有出现异常，运行中完全不需要为异常处理做任何事情，“不必为没出现的异常付出任何代价”。这种 **0** 开销性质很吸引人的，是考虑语言的设计与实现技术时的一种重要追求

异常处理：实现比较

不出现异常时：

- 采用静态表方法的程序执行中无须付出任何代价
- 而采用动态链的方法，每次进出异常监视区域都需要维护动态链，其实际开销不可忽略

出现异常时：

- 采用静态表格，在异常的每个引发点做一次二分查找，对数复杂性，与整个程序中监视区域数有关。程序越大，程序中的监视区越多，查表开销也越大（与程序的静态结构有关）
- 同一异常可能在退出多层子程序的过程中多次引发，需要多次查表
- 动态链方法的开销与链表结构（动态调用结构）相关

一般说，采用静态表方法，出现异常时的处理代价可能高于动态链方法

静态表的另一缺点是表格需要静态建立和排序，因此无法很好地与动态连接和装载相容。动态链方法则可以很自然地支持程序的动态连接等机制

异常处理的实现：异常辨识

确定一个处理器能否处理当前异常，称为异常处理器的辨识或者匹配

- 一方是当时引发的异常，另一方处理器
- 需要有一种方式，给出一个肯定或否定的回答

异常用简单符号表示，匹配操作的实现也非常简单

- 可给每个异常一个唯一标识（例如用整数）
- 匹配就是基本标识比较

基于类型的异常表示机制功能更强大，使用更方便

- 匹配就是运行中的子类型判断
- 需要类型的运行时表示的支持，这是类型将不再是简单的静态概念，其运行时的表示（类型对象）必须能支持子类型关系判断
- 基于类型辨识异常处理器的工作比较复杂，但能支持基于类型和子类型的异常处理，支持许多程序技术

异常处理

从控制的角度看，异常可看作一种能跨过子程序边界的控制转移机制（退出机制）。其最终转移目标由动态调用链的当时情况动态确定

有人说异常处理是一种强大的非局部goto："无法确定它从何来，也不知道它跳到哪里去"，一切都是在动态运行中确定的

这种说法过于极端，但也说明使用异常处理机制时应十分小心，尽可能地结构化和规范化，不要用它去搅乱正常控制流

此外，异常处理代价较高，不要用它去处理正常控制结构能处理的问题。这也就是人们提出"只用异常机制处理错误"的缘由

应强调：

如果需要区分错误处理流和正常处理流，处理只有在底层才能发现而又无法局部处理的情况，异常处理机制是最合适的处理工具

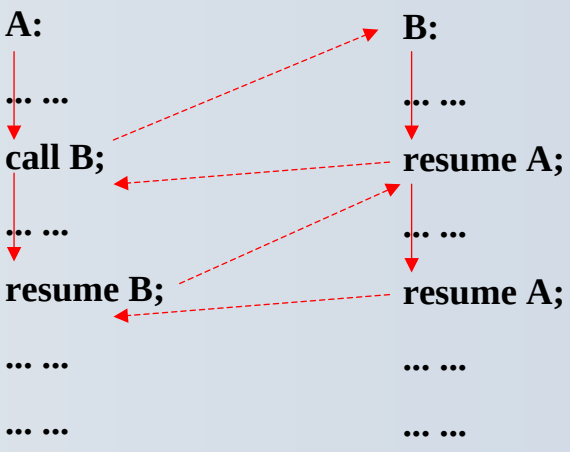
如果用的合理而有节制，异常处理能帮助写出更可靠而强健的程序，这是软件开发的最重要追求之一

其他子程序：协程

采用调用/返回控制方式的过程和函数是子程序中最简单最规范的情况。后进先出活动记录可以用栈式管理

其他子程序需要更复杂的控制方式，其中一种重要控制模式称为协作程序（coroutine），简称协程

协程的特点是显式的主动控制转移：主动交出执行控制，指明要求继续执行的（被唤醒的）协程



如果一个协程执行一个 **resume** 操作去唤醒其他协程，它自己就暂停（挂起）在唤醒操作之后的位置，等待被其他协程唤醒

协程的执行和挂起并没有后进先出或者其他规律性，协程 **B** 被 **A** 唤醒，它完全可能在执行一段后去唤醒另一协程 **C** 且自己挂起

如果一个协程挂起，无论它后来被谁唤醒，都从其挂起位置继续

协程

协程像其他子程序一样有开始和结束

- 程序开始执行时一组协程都处于启动点，但只有一个真正开始执行
- 协程通过唤醒转移控制权，每个时刻只有一个协程是“当前运行协程”
- 宏观地看，所有未结束的协程都在“运行中”，都在逐步推进
- 如果当前运行协程结束，应该有一种机制选择唤醒某个尚未结束的协程

完全可能考虑“动态启动”新协程的机制

- 可以采用启动后即唤醒的设计
- 也可以是启动与唤醒分离，启动只是使协程处于随时可以被唤醒的状态

由于多个协程之间没有后进先出关系，其活动记录（包括被协程调用的子程序的活动记录）无法采用栈式管理，需要用堆或其他复杂的管理技术

协程可以看作多个同时存在的执行进程，**Simula-67** 等语言提供了协程机制，可以用于模拟多个同时进展的活动

其他子程序：并发子程序

另一类子程序是并发子程序：

- 一种子程序，这些子程序的活动（称为进程）可以独立地同时并行执行
- 每个子程序有自己的执行状态（执行点和执行上下文环境）
- 同时执行的并发子程序间可能需要相互协调行为，相互传递信息
- 如果存在多个执行硬件，处于执行中的多个子程序可能各自占用独立的执行硬件（如 **CPU**），得到真正的并行执行
- 如果处于运行中的并行子程序活动多于可用的执行硬件，那就会出现多个活动子程序交替使用一个执行硬件的情况（共享执行硬件）
- 并行子程序可能不是主动交出执行控制，通常不明确指定下一个进入执行的子程序进程，而是由一个独立的调度系统按照一套预定义规则去确定将哪个（哪些）可以执行的活动（进程）投入执行
- 有关并行子程序的详细情况，在后面讨论“并发性”的一章讨论

事务处理

人们在将计算机用于业务处理的过程中，总结出事务处理的概念

- 一个事务（**transaction**）是一个（简单或复杂的）处理动作单元，特点是它对执行的要求：或者这一动作的效果完全实现，或者其执行毫无效果。也就是说，事务要求一种 **all or nothing** 语义
- 人们逐步认识到事务概念的重要性
  - 在数据库领域，希望一个更新操作或者完成，或者没有改动数据库
  - 近年人们一直在研究事务内存，希望通过这种概念支持并发程序
- 如何在语言层面上支持面向事务处理的程序设计，已成为一个重要问题
  - 简单事务操作具有**ACID**性质（原子、一致、独立、耐久，**Atomic**、**Consistent**、**Isolated**、**Durable**），通常采用设置检查点，无法完成就自动回滚（**rollback**）的技术
  - 更一般的事务处理需要考虑时间很长的事务，一批基本事务已经完成后的撤销，不能简单回滚的回退动作等等

事务处理和补偿

一般事务处理需要补偿（**compensation**）的概念。补偿：

- 用户（程序员）根据需要通过编程定义的恢复动作（不是是简单回滚）
- 用于撤销已经部分完成的工作，包括已经结束的子事务
- 对于子事务的撤销也未必是以反序进行

在服务计算领域，事务处理和补偿都是非常重要的编程概念，需要从语言的层面上提供支持。面向服务组合的 **BPEL** 等语言考虑了这方面的需要：

以 **BPEL** 为例，它提供的机制包括：

- 以 **scope** 作为事务处理的描述单元，提供**“all or nothing”**语义
- **scope** 执行中发生异常时，触发异常处理动作。异常处理可以根据需要执行子事务的补偿动作，撤销已经完成的子事务
- 完成的 **scope** 的补偿动作自动注册，以便以后撤销时使用

子程序控制：总结

- 子程序：能独立执行的一段代码。存在多种不同的子程序
- 命名的子程序形成一种控制抽象，扩充了语言的基本词汇表
- 子程序的局部环境形成一种状态隔离，通过活动记录表示
- 参数化使子程序可用于解决一类问题的不同实例
- 不同参数机制各有特点，服务于不同的需要
- 实现子程序（函数和过程）
  - 如果没有递归，可以采用静态实现方式
  - 一般情况下，可利用其后进先出特性，采用栈帧的方式表示其活动记录。子程序实现需要前序和后序代码
- 泛型是子程序（和其他程序结构）的类型参数化
- 异常处理可完成跨过程的控制转移，通常用于处理执行中出错的情况
- 协程和并发子程序各有特点，但一些基本性质与过程和函数类似