TRANSACTION DSL

发布 *1*

Darwin Yuan

2021 年 07 月 27 日

目录

注解： Transaction DSL 是一套使用 C++ 编写的领域专用语言，通过它，可以简单直观的描述任意复杂的异步通信过程。

CHAPTER 1

主要改进

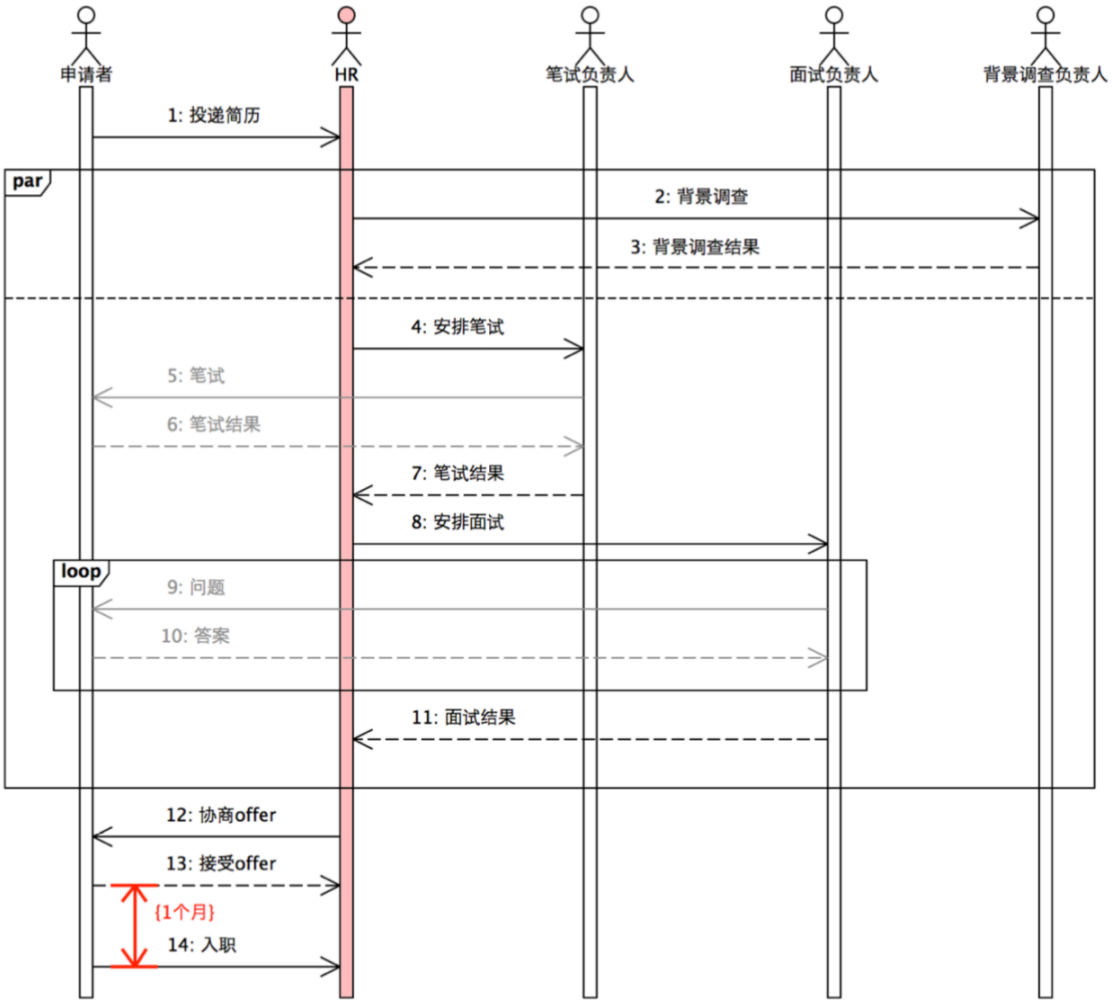
2.0 版本相对于 1.x 版本的主要改进有:

- 简化用户定义方式
- 极小的内存占用
- 极快的性能
- 高度灵活的循环
- 清晰一致的错误处理策略

1.1 动机

我们先从大家都熟悉的一个例子说起。

任何一家企业都需要招聘员工。描述的是某家企业制定的招聘流程图:

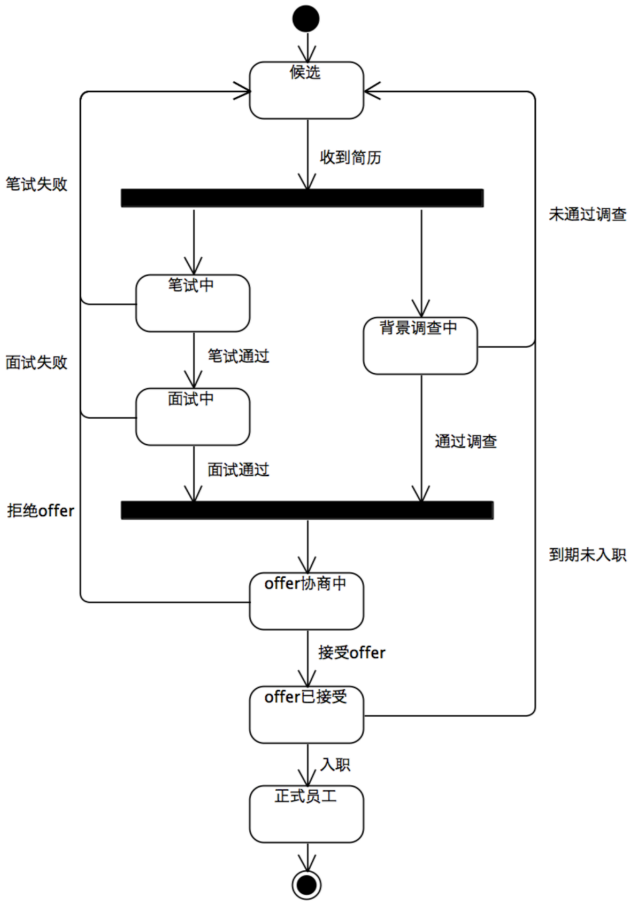


正如图中所描述的，流程的起始是收到应聘者投递的简历，而成功招聘到一个人的标志是：应聘者入职。

现在，我们用软件实现这个过程，其中，每个角色都是一个独立的系统或子系统，他们之间必须通过 **消息**或**事件**进行通信。如果，我们现在需要实现 HR 子系统，该如何做?

1.1.1 状态机

状态机是大多数基于消息的异步系统常用的实现方式。所以 HR 为一个应聘者设计了如图所示的状态机:



状态模型和 **序列模型**都能较好的反映一件事情的本质，但不同的是，**序列模型**更加专注于目标系统的 **行为**，而 **状态模型**更关注被操作对象的 **状态迁移**。

但从实现的角度来看，如果你选择 **序列模型**，当然，你需要实现目标系统的 **行为**；而如果你选择 **状态模型**，那么除了需要管理 **状态机**之外，你仍然需要处理目标系统的 **行为**。这就意味着，在实现层面，一旦选择 **状态模型**，就需要做许多额外的工作。

而这部分额外的工作绝不是轻松和愉快的

首先，**状态机**应该管理每个状态下期望的激励，对于未期望的消息则应该忽略。比如，在本例中，一个正处于 笔试中状态的状态机，如果收到了一个 接受 offer 消息，将是一件很奇怪的事情，所以，只能忽略它。一个成熟的团队，对于状态机的管理往往会引入一个状态机引擎。一个典型的引擎需要程序员自己定义一张状态表，然后注册给引擎。

```
const StateTable states[] = {
  // ...
  { STATE_EXAM,      { {EV_EXAM_RESULT, handleExamResult}}},
```

(下页继续)

(续上页)

```
  { STATE_INTERVIEW,{ {EV_INTERVIEW_RESULT, handleInterviewResult}}}, // ...
  { STATE_OFFERED,  { {EV_ONBOARD, handleOnBoard}
                    , {EV_TIMEOUT, handleTimedout} }}
};
```

有些引擎则呈现出另外一种形式，但本质上没有任何不同:

```
Status STATE_OFFERED_Handler(const Event& event) {
  switch(event.getEventId())
  {
    case EV_ONBOARD : return handleOnBoard(event);
    case EV_TIMEOUT : return handleTimedout(event);
    default: // error log
  }
  return SUCCESS;
}
```

然后，在具体的 **事件处理函数**中，首先要处理这条消息，然后再根据处理的结果进行状态迁移。比如:

```
Status handleExamResult(const Event& event) {
  auto result = (ExamResult*)event.getContent();
  if(result->pass) {
    // 处理消息
    arrangeInterview();
    // 状态迁移
    gotoState(STATE_INTERVIEW);
  } else {
    // 处理消息
    reject();
    // 状态迁移
    gotoState(STATE_IDLE);
  }
  return SUCCESS;
}
```

从上述实现可以看出，对于 **状态模型**来说，其图形描述和代码实现之间存在很大的鸿沟。因为在任何一个成熟的 **状态机引擎**中，由用户所定义的状态表仅仅描述了一个状态机所拥有的所有状态，以及每个状态期望的激励，却没有描述状态之间的 **迁移**。

这是因为，如果想让状态机也能够描述状态迁移，就必须让状态机的迁移是确定的，这就会导致某些状态机的设计不得不进行转换，而转换的结果往往会失去直观性。所以，为了拥有灵活性，状态的迁移只能由各个事件处理函数来完成。

因此，为了在代码层面理解一个 **状态机**的设计，必须仔细的阅读相关代码，并在不同代码间来回跳转（因为状态一直在跳转），才能理解一个状态机的全貌。

另外，**状态机引擎**只能负责状态的管理，功能非常单一。所以，除了状态管理之外的所有其它细节，都必须有用户亲自实现。比如，本例中的时间约束，用户就必须亲自操作定时器:

```
Status handleOfferAccepted(const Event& event) {
  // ...
  // 启动定时器，以确保应聘者可以按时入职
  ASSERT_SUCC(startTimer(TIMER_ONBOARD));

  // ...
  // 状态迁移
  gotoState(STATE_OFFERED);

  // ...
}

Status handleOnBoard(const Event& event) {
  // ...
  // 应聘者已入职，关掉相应的定时器
  ASSERT_SUCC(stopTimer(TIMER_ONBOARD));

  // ...
  // 状态迁移
  gotoState(STATE_HIRED);

  // ...
}
```

而对于的复杂的系统而言，由于各种并发，及并发的丰富组合，要么会导致状态的急剧膨胀，以至于状态机及其的晦涩，难以理解和维护。

为了避免状态机的膨胀，程序员会选择使用一些记录状态会合的标记位（这本质上仍然是一种状态扩展），以表示多个并行是否可以会合，并在执行过程中对这些状态位进行检查。

```
Status handleInterviewResult(const Event& event) {
  auto result = (InterviewResult*)event.getContent();

  // ...
  if(result->pass) {
    // 背景检查已通过？
    if(backgroundCheckPass()) {
       startOfferNegotiation();
       gotoState(STATE_OFFER_NEGO);
    }
  }
```

(下页继续)

(续上页)

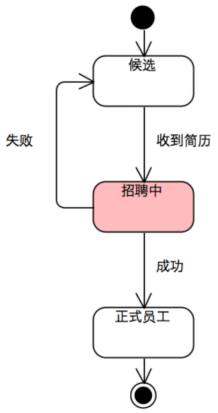
```
  // ...
}

Status handleBackgroundResult(const Event& event) {
  auto result = (BackgroundResult*)event.getContent();

  // ...
  if(result->pass) {
    // 面试已通过?
    if(interviewPass()) {
      startOfferNegotiation();
      gotoState(STATE_OFFER_NEGO);
    }
  }

  // ...
}
```

1.1.2 同步模型

我们已经看到，**状态模型**在实现层面带来了一系列的复杂度。我们如何才能降低这种复杂度?

事实上，稍加思考，就不难发现，在本例中，一个应聘者真正的状态只有两个，一个是 候选状态，一个是 正式员工状态。

而从 候选到 正式员工之间是一个 **转换**（Transition）。众所周知，转换是一个连续的 **过程**，而不是 **状态**。如果非要将其称为一个 **状态**，它的本质属性也和另外两个状态不同，它属于一种不稳定的 **临时中间状态**。如图所示:

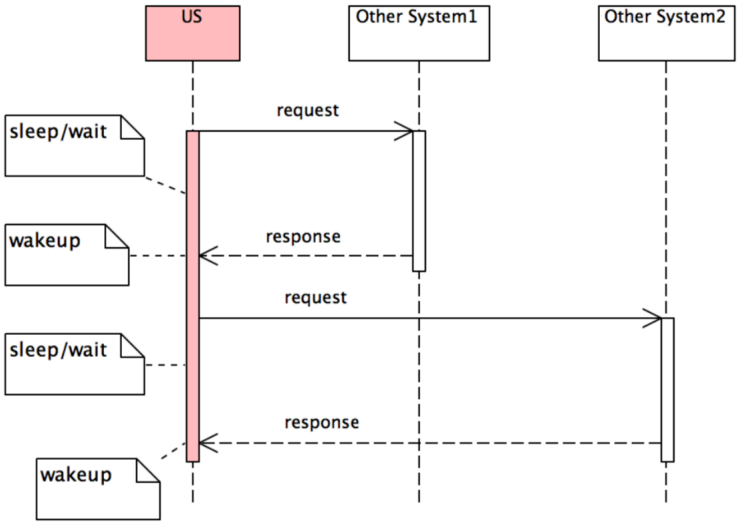


作为一个 **转换**，和简单的转换不同的是，它是一个 **异步过程**，需要大费周章之后才能完成。为了能够控制这个异步过程，才不得不引入状态模型。

但是，如果能将其转化为 **同步过程**的话，那些 **临时中间状态**就失去了意义。我们也就无需再为之引入复杂的状态机。

所以，为了简化异步消息所带来的状态控制，有些设计师会选择通过框架，将一个顺序的 **异步过程**转化为 **同步过程**。

在一个同步过程里，一个系统或子系统一旦发出一个请求消息，并需要等待其应答，则当前进程/线程就会进入休眠态，直到应答消息来临或超时为止。在此期间，所有发给此进程的其它消息将无法得到处理。如图所示:



如果控制过程比较简单，而此应答消息正是其期待的唯一消息的话，这无疑是一种非常聪明而简单的处理。

不幸的是，对于稍微复杂的实时系统而言，这样的做法很多时候无法满足实时性的需要。

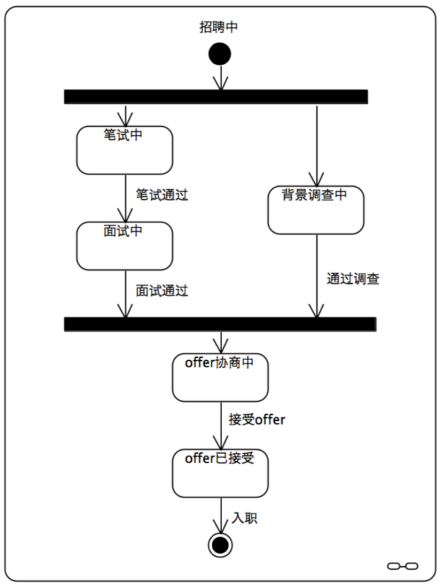
比如并发问题。sleep-wakeup 模型无法做到同时处理多个并发的异步过程。而并发，则是一个异步系统为了满足实时性和性能的必要手段。

这种情况下，一旦需要并发，同步模型就必须启动 OS 线程以进行应对。由此，程序员就不得不编写相关的进程/线程间通信和同步的代码，而这些实现也散乱在系统的各处，无法和序列图中的内容建立起直观的映射关系。

1.1.3 事务

现在，我们似乎陷入了两难的境地：由于异步过程的存在，及异步过程实时性的要求，我们不能总是简单的将异步过程同步化。既然不能同步化，为了良好的控制一个任意复杂度的异步过程，我们似乎只剩下了一种手段——**状态机**；而状态机实现所带来的一系列复杂度又不是我们真正想要的。(这个复杂度难道真是 **内在**的，而不是 **偶发**的吗?)

现在，我们再回到 招聘 中这个临时状态中，仔细的观察一下它的特点，如图所示:



不难发现，这个过程中的任何步骤发生失败都会导致整个转换失败，只有全部成功之后，整个转换才算成功。这就让它成为一个不可分割的原子操作。要么全部成功，要么全部失败（之前通过的考核也统统失去了意义）。

而这正符合一个源自于数据处理的概念：**事务**（Transaction）。

事务: ***from*** **WIKIPEDIA** In computer science, transaction processing is information processing that is divided into individual, indivisible operations, called transactions. Each transaction must succeed or fail as a complete unit; it cannot remain in an intermediate state.

现在，事物的原貌已经浮出水面：图中描述的才是真正的 **状态机**，而图中描述的过程则是一个 **事务**。

对于前者，我们仍然使用 **状态模型**来表示；而对于后者，我们则需要引入更准确的模型——**事务模型**——来解决。

1.1.4 Transaction DSL

事物模型用来描述状态之间的 **转换过程**：它可以由一系列的 **同步**和 **异步**操作（Action）组成。

而 *Transaction DSL* 则是一种用来描述事务的语言。它用来定义状态之间的复杂转换过程。从而避免使用状态机来描述状态转换过程中由于异步而导致的 **临时中间状态**。

Transaction DSL 不是为了取代 **状态模型**，而是为了提供一种方法，以解决那些本来不应该属于状态模型，却在使用状态模型进行解决的问题。从而大大简化实现的复杂度，并缩小用户视图和实现视图之间的距离，让设计和实现更加符合事情的原貌，最终降低开发和维护成本。

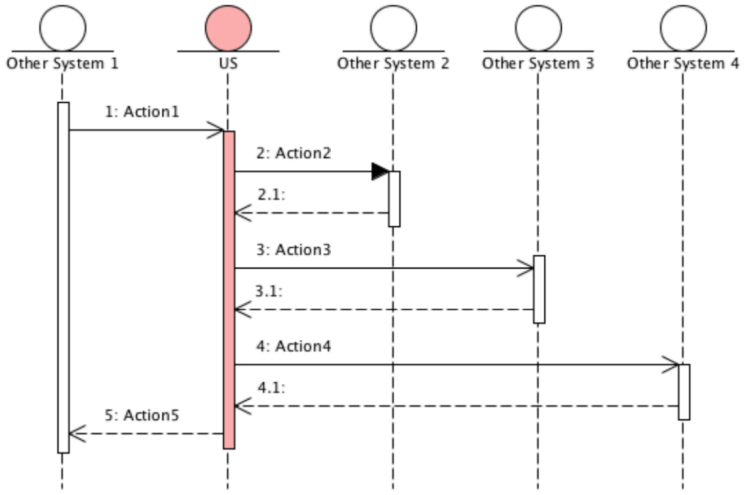
在下面的章节里，我们将会展示：我们如何通过 *Transaction DSL* 来定义一个 **事务**。

1.2 快速开始

我们先通过一系列简单的例子，来对 *Transaction DSL* 有一个直观的认识。

如图所示，站在 *US* 的角度，一个完整的事务操作，共分为 5 个步骤:

1. 从 *Other System 1* 收到一条请求消息;
2. 对 *Other System 2* 进行了一个函数调用;
3. 对 *Other System 3* 发送了一条异步消息，并等待其应答消息;
4. 对 *Other System 4* 发送了一条异步消息，并等待其应答消息;
5. 给 *Other System 1* 发送了一条应答消息。



1.2.1 定义事务

使用 *Transaction DSL* ，我们将上面序列图中的过程定义为:

```
__transaction (
  __sequential
    ( __asyn(Action1)
    , __sync(Action2)
    , __asyn(Action3)
    , __asyn(Action4)
    , __sync(Action5))
);
```

这段代码定义了一个名为 `Transaction` 的事务。它由一个包含了 5 个步骤的 **顺序操作**（Sequential Action）构成。每个步骤都是一个 **基本操作**（*Atom Action*）。

基本操作共分为两类：**同步操作** (*Synchronous Action*) 和 **异步操作** (*Asynchronous Action*)。

任何无需进一步等待后续消息的 *Action*，都称作 **同步操作**。这包括：函数调用，以及只发消息、不等回应的操作。它们都需要通过 `__sync` 来指明。但为了更加清晰的区分究竟一个同步操作属于哪种具体类型，你可以使用 `__call` 来说明一个操作是函数调用，用 `__ind` 来说明这是一个发送指示消息的操作，用 `__rsp` 来说明一个操作是对之前某个请求的一个应答。

任何需要等待异步消息的操作，都称作 **异步操作**。这包括：典型的 **请求——应答**操作，消息触发的操作等。而异步操作则需要通过 `__asyn` 来说明。同样的，你可以使用其它一些更加具体的修饰来明确一个异步操作的类型。比如：你可以使用 `__req` 来说明这是一个消息触发型的请求操作。

另外，由于 `__transaction` 可以直接识别其内部结构是否是一个 `__sequential`，所以 `__sequential` 可以被省略。

所以，之前的例子可以修改为：

```
__transaction
( __req(Action1)
, __sync(Action2)
, __asyn(Action3)
, __asyn(Action4)
, __rsp(Action5));
```

现在，我们可以看出，这段代码是对上述序列图简单而直接的描述。

1.2.2 定义基本操作

为了让上述代码可以工作，你需要实现其中的基本操作。事实上，无论一个事务有多么复杂，最终总是由基本操作组成，你需要做的事情就是定义这些基本操作，然后使用 *Transaction DSL* 来描述它们运行的方式和顺序。

我们之前已经提到，基本操作分为两种：**同步操作**和 **异步操作**。定义它们时需要实现带有如下方法的类。

```
auto exec(TransactionInfo const&) -> Status;
```

```
auto exec(TransactionInfo const&) -> Status;
auto handleEvent(TransactionInfo const&, Event const&) -> Status;
auto kill(TransactionInfo const&, Status cause) -> void;
```

对于 **同步操作**，你需要做的就是定一个原型为 `Status (const TransactionInfo&)` 的函数。

如果你通过类来定一个同步操作，你需要做的就是在你的类中定义一个名为 `exec` 的方法。

对于函数调用型的同步操作，其实现非常简单，其返回值为 `SUCCESS` 代表此操作成功，如果返回错误值则表示此操作失败。

```
auto Action2(TransactionInfo const&) -> Status {
  return OtherSystem::func();
}
```

甚至可以是一个 *lambda*：

```
auto Action2 = [](TransactionInfo const&) -> Status {
  return OtherSystem::func();
}
```

对于 Action5，尽管它发送了消息，却无需等待任何消息，所以它也是一个 **同步操作**。虽然也可以直接用函数直接定义，但基于举例的目的，这次我们用类来定义它：

```
struct Action5 {
  auto exec(TransactionInfo const&) -> Status {
    // 构建并发送消息
    Response1 response;
    response.build();
    return sendResponseTo(OTHER_SYSTEM1_PID, response);
  }
};
```

而对于 **异步操作**，存在一些常用的模式。大多数情况下，你可以根据模式从已存在的基类中继承。比如：如果一个异步操作属于简单的 请求-应答模式，你只需要从 SimpleAsyncAction 继承即可。

```
DEF_SIMPLE_ASYNC_ACTION(Action3) {
  auto exec(TransactionInfo const&) -> Status {
    // 构建并发送请求消息
    Request3 request;
    request.build();
    Status status = sendRequestTo(OTHER_SYSTEM3_PID, request);
    if(status != SUCCESS) return status;

    // 声明自己要等待的应答消息类型，以及对应的处理函数；WAIT_ON会返回CONTINUE
    return WAIT_ON(EV_ACTION3_RSP, handleAction3Rsp);
  }

private:
  // 定义事件处理函数
  auto handleAction3Rsp(TransactionInfo const&, Event const& event) -> Status {
    // 处理应答消息
    handleRsp(event);
    // 返回成功，代表此 Action 成功处理结束
    return SUCCESS;
  }
};
```

而 Action1 则属于一个事件触发的操作，所以它不发送消息，只等待那么触发消息。但它仍然可以继承自 SimpleAsyncAction 。

```
DEF_SIMPLE_ASYNC_ACTION(Action1) {
  auto exec(TransactionInfo const&) -> Status {
    // 声明自己要等待的消息类型，以及对应的处理函数
    return WAIT_ON(EV_ACTION1_REQ, handleAction1Req);
  }

private:
  // 定义事件处理函数
  auto handleAction1Req(TransactionInfo const&, Event const& event) -> Status {
    // 处理触发消息
    handleReq(event);
    // 返回成功，代表此 Action 成功处理结束
    return SUCCESS;
  }
};
```

对于 **异步操作**的所有函数，其返回值有三种: `SUCCESS` 表示此操作成功结束; `CONTINUE` 表示此操作尚未结束，需要进一步的处理；**错误值**则表示此操作已经失败。

而函数 `handleEvent` 则存在一种额外的返回值：`UNKNOWN_EVENT`，说明当前消息不是自己期待的消息。

返回值	语意
`SUCCESS`	Action 成功结束
`CONTINUE`	Action 仍然在工作
`UNKNOWN_EVENT`	Action 收到一个未期待的消息
错误码	Action 失败，并已经中止

1.2.3 约束

用户自定义的基本操作，如果通过类来定义，*Transaction DSL* 要求它们必须是自满足的。即，它们不需要外部通过 **构造函数**或 `set` 函数设置任何外部依赖。所有的依赖，都需要靠类自身到环境中亲自寻找，或亲自创建。所以，这些类必须存在 **默认构造函数**。至于其它带参数的 **构造函数**或 `set` 接口，虽然其存在并不会妨害 *Transaction DSL* 的编译和运行，但它们永远也不会得到调用。

这样的约束，并不会对设计造成任何妨害。因为这些类本来就靠近系统的边界。而边界的代码本身就应该承担寻找或创建目标对象的职责。

1.2.4 运行

现在我们有了 *Transaction*，有了基本操作，一个事务就完整了。由于 *Transaction* 是一个事件驱动的组件。它的基本接口定义如下:

```
auto start() -> Status;
auto start(Event const& event) -> Status;

auto handleEvent(Event const& event) -> Status;
auto stop(Status cause) -> Status;
auto kill(Status cause) -> void;
```

所以，你可以选择任何一个 `start` 接口来启动一个事务。像一个异步操作一样，如果其返回值是 `SUCCESS`，说明此事务已经成功的执行；如果其返回值是一个 **错误值**，则说明此事务已经失败；而如果其返回了 `CONTINUE`，则说明此事务正在工作状态中，尚未结束，仍然需要进一步的消息激励。

在 `start` 接口返回 `CONTINUE` 的情况下，随后每次系统收到一个消息，都需要调用其 `handleEvent` 接口，直到其返回 `SUCCESS` 或一个 **错误值**为止。

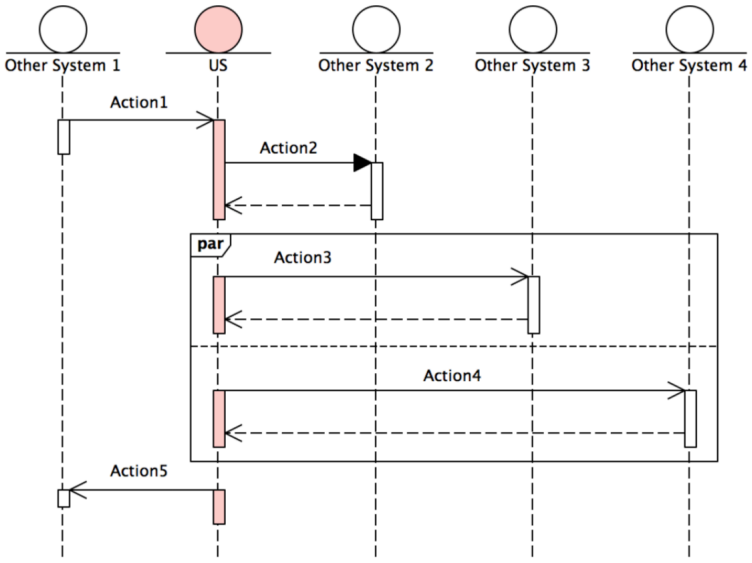
一个可能的实现如下所示:

```
auto runTransaction() -> Status {
  // 将之前定义的 Transaction 实例化 Transaction trans;
  // 启动
  auto status = trans.start();
  if(status != CONTINUE) return status;

  // 消息处理循环
  while(recvEvent(event) == SUCCESS) {
    status = trans.handleEvent(event);
    if(status != CONTINUE && status != UNKNOWN_EVENT) {
       return status;
    }
  }

  return FAILED;
}
```

这个实现是一种简单的处理，主要为了说明一个事务的运行方式。事实上，事务往往不是一个系统的顶层框架，一个事务仅仅是对一个处理过程的描述。在复杂系统中，事务与事务之间可以并发，可以抢占。但那是事务框架之外的事情。在这里我们就不在详细讨论。

1.2.5 并发

一旦系统因为性能要求，需要同时给不同其它系统/子系统发出请求消息，并同时等待它们的应答，如图所示。



在这个例子中，Action3 和 Action4 同时给各自的目标系统发出请求消息，并各自等待应答。这种情况下，简单的策略已经无法处理，实现者仍然不得不回到状态机模型中。

使用 *Transaction DSL*，一个并发过程的定义非常简单，如下所示：

```
__transaction
( __req(Action1)
, __sync(Action2)
, __concurrent(__asyn(Action3), __asyn(Action4))
, __rsp(Action5));
```

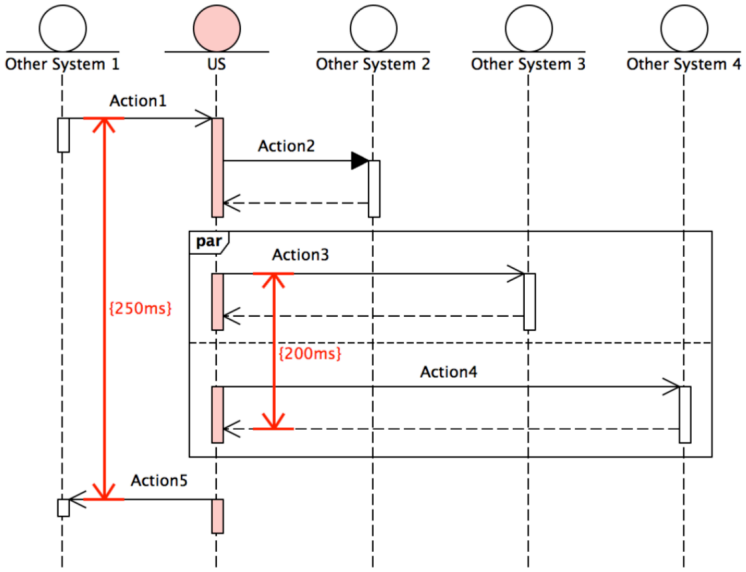
我们只需要将 Action3 和 Action4 放入一个叫做 __concurrent 的盒子里即可。它会保证两者可以得到并发的执行，并发的等待应答，并确保，只有在 Action3 和 Action4 都执行结束后，才会执行 Action5 。

哦…事实上,最后这条不是它单独保证的,因为 __concurrent 这个盒子放在了更大的盒子 __sequential 里面，在 __sequential 看来，它里面有四个操作：分别是 Action1 , Action2 , __concurrent 和 Action4 ，它会来保证这四个操作严格的按照顺序来执行。

1.2.6 时间约束

既然是异步系统，那么发出去的消息就有可能由于各种原因而一去无回；或至少，很晚才回来。为了避免系统被这样的情况挂死，或者不满足实时性要求，设计者往往会对异步过程有着时间的约束；即在一定时间之内，如果一个操作无法完成，则当前操作就失败。

下图所示的过程就属于这样的情况。其中，存在两个时间约束：首先，整个操作必须在 250ms 之内完成，而其中的并发异步过程则必须在 200ms 之内完成。



我们将这个带有时间约束的事务描述如下：

```
const TimerId TIMER_1 = 1;
const TimerId TIMER_2 = 2;

__transaction
( __time_guard
    ( TIMER_1
    , __req(Action1)
    , __sync(Action2)
    , __time_guard(TIMER_2, __concurrent(__asyn(Action3), __asyn(Action4)))
    , __rsp(Action5))
);
```

母语言

在这段代码里，我们首先定义了两个 `TimerId：TIMER_1` 和 `TIMER_2` 。其定义的方式是 *C++* 的常量。为什么这里可以使用 *C++* 的语法?

是的，*Transaction DSL* 本身是就是 *C++* 的代码，它可以被任何成熟的 *C++* 编译器编译。它是 *C++* 代码这个事实，让它可以在需要时，使用任何 *C++* 的元素。

母语法

它采取的语法形式非常接近于 *Lisp* ，除了两点较大的差别:

1. *Lisp* 将一个 *list* 的名字放在括号内;
2. *Lisp* 一个 *list* 内的各个元素之间无需逗号分割。

如果你以前没有接触过 *Lisp* 语言，可通过下面一段 *Lisp* 的例子代码获取直观的印象。

```
(defun find-books (towns)
   (if (null towns) nil
     (let ((shops (bookshops (car towns))))
     (if shops (values (car towns) shops)
               (find-books (cdr towns))))))
```

无论你是否喜欢 *Lisp* 语言的语法形式，但对于我们的问题，这已经是我所能找到的 *C++* 元编程的最好表现形式。但两者相似的地方不仅仅是语法形式，更重要的在于其背后的 **元模型**。对于 *Lisp* 语言而言，一切都是 *List* ，所以，它本身就在表述一颗 **语法树**。

而对于 *Transaction DSL* 而言，一切都是 *Action* ，它本身就在表述一颗 *Action Tree* ，在这颗树上，每一个节点所代表的子树都是一个 *Action* 。而在叶子节点，就是用户所定义的 **同步**或 **异步**的 *Action* (操作)。

Timer ID

你已经看到，我们并没有在 *Transaction* 中直接写 `200ms` 或 `250ms` ，而是使用了 *Timer ID* 。

将时间约束的数值直接写在 *Transaction* 里，从实现上并无难度，但是，这将导致用户将失去运行时修改它的能力。而用户很有可能希望某些时间约束是可配置的，从而在运行时是可修改的。

所以，我们需要定义一个标识来定义这个约束，然后，Timer 管理器会将这个 TimerID 和某项配置关联起来(你当然也可以 hard coding 将其写死)。

另外，无论 *Transaction DSL* 被移植到什么平台上，都必须要满足:

- 在 *Timer* 过期时，必须发送一条消息出来;
- *Timer* 的实现必须能够将 `Timer ID` 和 *Timer* 过期消息之间建立起一一映射关系。而这一切，在我所见过的各种 *Timer* 机制中，都不难实现。

同步操作的时间约束

虽然你可以在一段 *Transaction DSL* 代码中，对一个同步操作进行时间约束，但事实上，这个约束形同虚设。

这是因为，一旦进入一个 **同步操作**的执行，事务框架就失去了控制权。所以，它无法在定时器过期时，抢占或打断一个同步操作的执行。

而 **异步操作**则不同，由于它们需要等待消息激励，在等待期间，事务掌握着控制权，当收到其定时器过期消息时，事务可以马上中止其运行。

1.2.7 回到最初

现在，我们可以使用 *Transaction DSL* 来对我们最初的过程进行描述:

```
__transaction
( __asyn(ApplicationAcceptance)
, __concurrent
    ( __asyn(BackgroundInvestigation)
    , __sequential
        ( __asyn(Exam)
        , __asyn(Interview)))
, __asyn(OfferNegotiation)
, __time_guard(TIMER_ONBOARD, __asyn(OnBoard))));
```

剩下的事情，就是把每一个 **基本操作**进行实现，而它们都是非常简单，原子级别的交互过程。

1.2.8 优势

在本章中，我们初步了解了如何使用 *Transaction DSL* 来定义一个异步过程。即，实现 **原子操作**：分为 **同步**和 **异步**两种。然后通过 *Transaction DSL* 来描述这些 **基本操作**的执行方式。

通过这些例子，我们可以看到 `Transaction DSL` 的一些主要优势:

直观: *Transaction DSL* 的描述，和 *UML* 序列图的描述有着明确直观的映射关系。设计和实现之间的 gap 被大大缩小。

简单: 没有复杂的事无巨细的状态机，没有重复繁琐的定时器操作，没有亲力亲为的并发过程控制——开发这些代码是一个无趣的，重复的，容易出错的，给开发和维护都带来大量成本过程。

不过你仍然需要写一些代码——那就是——将序列图没有冗余的描述一次。但…please…不要告诉我，你打算开发一个图形界面，将 *UML* 序列图自动转化为 *Transaction DSL* ，这样就可以连这些代码也不用写了。

毫无疑问，这将是一笔极其糟糕的投资，你应该将你过剩的精力花在真正有价值的事情上。

事务与操作的分离: 到目前为止，你可能已经注意到，虽然我们的例子从最初的简单顺序执行，到后来改变为并发执行，最后，我们又增加了时间约束，但我们却没有修改基本操作的一行代码。我们只是在修改事务的代码——以一种非常简单、快速的方式。

Native Language: *Transaction DSL* 本身就是 *C++* 代码，这就意味着它可以和 *C++* 的其它代码元素自由地结合使用，与其它 *C++* 代码进行无缝配合。

图灵完备: 随着后续章节的介绍，你就可以发现，*Transaction DSL* 是图灵完备的。也就是说它可以解决一切图灵可计算问题。

1.3 过程控制

在一个较为复杂的事务中，一些操作，只有在一定的条件下才会执行；或者，在不同的条件下执行的时机不同。

如图所示的过程中，`Action2` 就是一个可选的操作，它只有在系统的某个开关打开的情况下才会执行。

1.3.1 __optional

对于这样的情况，可以使用 `__optional` 来描述:

```
auto ShouldExecAction2(const TransactionInfo&) -> bool {
   return SystemConfig::shouldExecAction2();
}

__sequential
  ( __req(Action1)
  , __optional(ShouldExecAction2, __asyn(Action2))
  , __asyn(Action3)
  , __asyn(Action4)
  , __rsp(Action5));
```

从代码中可以看出，`__optional` 有两个参数：第一个参数是一个谓词，即一个返回值是一个 *bool* 的函数，如果为真，则 `__optional` 的第二个参数: 一个 Action 将会得到执行。这是 `C++` 解决这类问题的常用手段。

如果谓词是一个 *仿函数*，即一个类，*Transaction DSL* 对其有着和基本操作一样的约束，即它也必须是自满足的。所以，它必须亲自访问环境来确定一个条件是否成立。

TransactionInfo

无论是谓词还是基本操作，这些需要用户定义的类，都必须是子满足的，所以，它们自身计算所需的信息都必须亲历其为的到环境中查找。由于 *Transaction* 自身也是环境的一部分，所以 *Transaction* 必须通过参数将自身的信息传递给基本操作或谓词，从而让它们有能力得到一切需要的信息，这就是 `TransactionInfo` 的由来。

`TransactionInfo` 是一个接口类。通过它，你首先可以获取到 **实例标识**（*Instance ID* ）。因为有些系统对于同种类型的领域对象会创建多个实例，而每个实例都可能会有自己的 *Transaction*；通过 *Instance ID* ，用户定义的类就可以知道当前的 *Transaction* 属于哪个实例。

另外，*Transaction* 会通过 `TransactionInfo` 告知自身的运行时状态：是成功还是失败，如果失败，是什么原因导致的失败等等信息。

1.3.2 路径选择

现在我们让事情更为复杂一些。

在图所示的事务中，`Action2` 即可以在 `Action3` 之前运行，也可以在它之后运行，究竟选择哪种方式，取决于相应的配置。

```
auto IsAction2RunFirst(TransactionInfo const& info) -> bool {
   Object* object = Respository::findInstance(info.getInstanceId());
   return object->isAction2RunFirst();
}

__transaction
( __req(Action1)
, __optional
    ( IsAction2RunFirst
    , __asyn(Action2)
    , __asyn(Action3))
, __optional
    ( __not(IsAction2RunFirst),
    , __asyn(Action3)
    , __asyn(Action2))
, __rsp(Action4));
```

首先，在这个例子中，谓词的实现使用了 `TransactionInfo` 来获取 *Instance ID* ，进而通过它找到了对象，从而完成了判断。

其次，我们使用了两个顺序的 `__optional` 来进行路径选择，这两条路径是互斥的。而互斥的保证是通过 `__not` 来完成的。

__switch

当一段流程存在多条路径选择时，我们可以选择使用多个 __optional 来决定执行路径。

但是，我们从例子中同样可以看出，每个 __optional 都必须有自己的谓词，当谓词是一种互斥关系时，我们需要在不同的 __optional 里，使用 __not 来描述这种互斥。无疑，这会让事务程序员觉得麻烦。

另外，多个并列的 __optional 无法让事务代码的阅读者直观的看出这些路径之间的互斥关系。

所以，当存在多条互斥的路径时，最好应该使用 __switch 来描述:

```
__transaction
( __req(Action1)
, __switch( __case
              ( IsAction2RunFirst
              , __asyn(Action2)
              , __asyn(Action3))
          , __otherwise
              ( __asyn(Action3)
              , __asyn(Action2)))
, __rsp(Action4));
```

从代码中可以看出，在一个 __switch 里，一条路径可以使用 __case 来描述，而 __case 则和 __optional 一样，存在两个参数：谓词和操作。

当存在多条路径时，__case 的顺序则非常重要：*Transaction DSL* 会按照顺序依次匹配，一旦找到一条路径，将会执行其操作，并忽略其它路径，即便其它路径的谓词也可能匹配。

如果所有的 __case 谓词都不匹配，则 __switch 会返回事务的当前状态。__otherwise 则是一个语法糖，用来描述无条件匹配。所以，它应该作为一个 __switch 的最后一条路径，否则，在它之后的任何 __case 都不会得到调用。

__switch 要求至少两条路径选择。如果只存在一条路径时，使用 __optional 。

找到合适的描述方式

在一个通用编程语言中，在面临路径选择时，你可以找到多种等价的描述方式。为了让程序简洁，直观，我们应该选择最恰当的那一种。

同样的，对于本例，我们可以找到它的等价描述方式。如图所示:

```
__transaction
( __req(Action1)
, __optional(IsAction2RunFirst, __asyn(Action2))
, __asyn(Action3)
, __optional(__not(IsAction2RunFirst), __asyn(Action3))
, __rsp(Action4));
```

1.3.3 异常处理

一个事务是一个不可分割的操作，它或许会包括多个步骤，但这些步骤要么全部成功，要么全部失败。

所以，一个事务从开始到结束，中间发生任何错误，都会导致整个事务的失败。一旦一个事务失败，就会执行 **回滚** (*rollback*) 操作，以将系统恢复到事务开始前的状态。

当整个事务成功执行后，需要执行 **提交** (*commit*) 操作，自此，整个事务对于系统的改动才算真正生效。在提交后，整个系统无法再通过事务的 **回滚**操作恢复系统的状态。

Transaction DSL 提供了同样的机制：使用 *Transaction DSL* 定义的任何事务，在运行时，如果中间某个操作发生了错误，则整个事务就进入失败的状态。

但不幸的是，对于一个具体的，由用户自己定义的事务而言，*Transaction DSL* 无从得知，当失败时，应该执行的具体回滚机制是什么。所以 *Transaction DSL* 无法提供自动的回滚策略。或许对于某些系统，确实存在统一的模式，但另外一些系统则不然。

而在 *Transaction DSL* 的层面，则只能提供相应的机制；如果某些系统确实存在统一的回滚策略，则可以利用这些机制在 *Transaction DSL* 之上层面进行统一定义。

如果没有统一的策略，同样可以利用 *Transaction DSL* 所提供的机制定义差异化的回滚操作。

__procedure

Transaction DSL 提供了 `__procedure` 来定义一个过程，无论这个过程中的所有操作全部成功，还是执行到某一步时发生了失败，都会进入结束模式。用户可以自己定义结束模式里应该执行的操作是什么。如果按照之前对于事务的描述，则用户可以在结束模式里根据过程进入结束模式时的状态，进行提交或回滚操作。

所以，`__procedure` 包含了两个参数：第一个参数是此过程应该执行的正常操作，第二个的参数则是以 `__finally` 修饰的结束模式中应该执行的操作。

比如，对于 `__optional` 中的例子，如果系统要求此事务无论成败最终都应该执行 `Action5` ，但如果失败的话则需要对之前的操作进行 **回滚**。我们就可以将其描述为：

```
__procedure
( __req(Action1)
, __sync(Action2)
, __concurrent(__asyn(Action3), __asyn(Action4))
, __finally
    ( __rsp(Action5)
    , __on_fail(__sync(Rollback))));
```

之所以额外提供 `__procedure` 的概念，是因为，通过它，用户可以在一个事务中定义多个过程，每个过程都可以利用这种机制，从而让用户拥有更细力度的控制。例如，在下面的事务定义中，就存在两个过程：

```
__transaction
( __procedure
```

(下页继续)

(续上页)

```
    ( __asyn(Action1)
    , __asyn(Action2)
    , __finally(__rsp(Action3)))
, __asyn(Action4)
, __procedure
    ( __asyn(Action5)
    , __finally(__sync(Action6))));
```

需要特别指出的是，过程自身也是一个操作，如果一个过程发生了失败，在其 __finally 里定义的操作执行结束之后，仍然会让导致整个事务失败。

比如，在本例中，如果 Action2 发生了失败，将会引起 Action3 的执行；无论 Action3 执行成功还是失败，在它执行结束之后，均导致整个事务以终止运行。所以，其后的操作并不会得到运行，即便它们被定义为 __procedure 。

当然，__procedure 是可以嵌套的，比如:

```
__transaction
( __asyn(Action1)
, __procedure
    ( __asyn(Action2)
    , __finally(__sync(Action3)))
, __sync(Action4)
, __asyn(Action5))
, __finally(__sync(Action6)));
```

由于 __transaction 的最后一行是一个 __finally ，这就意味着本 __transaction 是一个 __procedure ，而这个 __procedure 内部又嵌套了另外一个 __procedure 。

在这个事务中，如果 Action2 发生了错误，Action3 将会得到执行，然后会跳过 Action4 和 Action5 ，直接进入外层过程的 __finally ，执行 Action6 。

__procedure 的恢复

我们前面已经指出，一个 __procedure ，如果其主体部分发生了错误，会跳转到 __finally ，而无论 __finally 里的 *Action* 成功与否，最终整个 __procedure 都会以失败结束。

但是，如果你的确想让一个在主体失败了的 __procedure 有可能以成功方式结束，则不要使用 __finally ，而使用 __recover 。如果主体部分失败，但 __recover 里的 *Action* 却成功了，则整个 __procedure 会在结束时返回成功。

所以，在下面的代码中，如果 Action1 发生失败，则会跳过 Action2 ，转入执行 Action3 ；如果 Action3 执行成功，则会继续执行 Action4 及后续过程。

```
__transaction
( __procedure
    ( __asyn(Action1)
    , __asyn(Action2)
    , __recover(__rsp(Action3)))
, __asyn(Action4)
, __procedure
   ( __asyn(Action5)
   , __finally(__sync(Action6))));
```

但是，如果 Action3 执行失败，则仍然，整个过程就失败了，此时，Action4 及后续过程将不会得到执行。

而对于下面这个事务，如果 Action2 发生了失败，则会执行 Action3 ，如果 Action3 执行成功，则继续执行 Action4 及后续过程; 否则，将跳过 Action4 和 Action5 ，转入执行 Action6 ，如果 Action6 成功，则整个事务将依然是成功的，否则，事务将以失败结束。

```
__transaction
( __asyn(Action1)
, __procedure
    ( __asyn(Action2)
    , __recover(__sync(Action3)))
, __sync(Action4)
, __asyn(Action5))
, __recover(__sync(Action6)));
```

所以，__recover 和 __finally 最大的不同的是，前者比后者多了一个给过程故障恢复的机会。

1.4 循环

循环的控制可以非常灵活。比如，在 *C/C++* 语言中，循环的典型语意有：

```
////////////////////////////
do {
  // ...
} while(cond);

////////////////////////////
while(cond) {
  // ...
}

////////////////////////////
for(init_list; cond; op) {
```

(下页继续)

(续上页)

```
  //
}
```

同时，在任意循环里，可以随时随地进行两种循环控制：

continue： 即重新回到循环的起点开始运行。

break： 中断当前的循环，或从当前的循环跳出。

程序员可以通过上述的语意，组合出来任意复杂的循环控制。

注意： 一个可以控制任意复杂循环控制的基本要素包括：

1. 提供一种或多种循环定义方式：比如，`while(1){}` 或者 `do{}while(1)`；
2. 用户可以在循环里的任何位置，根据当时的条件决定终止循环；即 `if(cond) break` 。其实 `while(cond)` 本身的语意等价于 `if(!cond) break`。因而 `while(cond){}` 和 `do{}while(cond)` 都是语法糖。
3. 用户可在循环里的任何位置，根据当时条件决定重新循环；即 `if(cond) continue` 。
4. 用于循环控制的状态信息：最典型的例子是计数器。这些状态信息，在整个循环进行期间，必须要保证其生命周期的连续性。

1.4.1 __loop

当程序员需要在一个 *Transaction* 里构建一个循环子过程时，可以使用 `__loop(...)` 来进行定义。这也是 *Transaction DSL* 所提供的唯一一种循环定义方式，它相当于 *C/C++* 中的 `while(1){}` 。

```
__loop
( __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5)));
```

这是一个死循环，它自身永远也不会终止，即便内部的某些 *Action* 出现运行时错误，它也不会停止循环。要想终止它，只能依靠 `stop` 或 `kill`。

因而，我们必须提供其它机制，让循环可以自然的终止。

1.4.2 __break_if

__break_if 大致相当于 *C/C++* 的 if(cond) break，即当 cond 所设置的条件得到满足时，即可立即跳出循环。

```
__loop
( __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __break_if(__is_timeout)
, __concurrent(__asyn(Action3), __asyn(Action5)));
```

这个例子中，如果在 __break_if 之前，发生了 TIMEOUT 错误，则循环终止，而整个 __loop 的运行结果也是 TIMEOUT 。

当然，如果用户想在结束循环的同时，让 __loop 的运行结果为另一个错误值，则可以明确进行指定：

```
__loop(
, __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __break_if(__is_timeout, SUCCESS)
, __concurrent(__asyn(Action3), __asyn(Action5)));
```

这样，__loop 的运行结果将是 SUCCESS 。

__while

事实上，正如我们之前所讨论的，我们可以使用 __loop 和 __break_if 描述 while(cond) {} 和 do{} while(cond)：

```
__loop( __break_if(__not(CondSatisfied))
, __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5)));

__loop
( __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5))
, __break_if(__not(CondSatisfied)));
```

也就是说，我们只需要将 __break_if(__not(cond)) 放在 __loop 的最前面和最后面，即等价于 while(cond){...} 和 do{...}while(cond) 。

为了表达的更加直观，*Transaction DSL* 提供了一个语法糖：__while(cond) ，其等价于 __break_if(__not(cond)) 。

```
__loop( __while(CondSatisfied)
, __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5)));

__loop
( __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5))
, __while(CondSatisfied));
```

当然，__while 也可以指定循环结束时的返回值：__while(cond, FAILED) ，如果不指定，循环结束时，则会返回循环所处的 **运行时环境**的状态。

__until

Transaction DSL 所提供的另外一个语法糖是 __until ，它完全等价于 __break_if 。但对于某些程序员来讲，这在循环尾部决定循环是否终止时，更加符合语意理解习惯。

```
__loop
( __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5))
, __until(CondSatisfied));
```

> **注意：** do ... until(cond) 的语意，与 do ... while(cond} 正好相反。

1.4.3 __redo_if

Transaction DSL 所提供的 continue 语意的关键字是 __redo_if，相当于 if(cond) continue。

```
__loop
( __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __redo_if(__is_timeout)
, __concurrent(__asyn(Action3), __asyn(Action5)));
```

在这个例子中，如果发生了 *timeout*，则不再执行后续的其它 *Action*，而是重新开始循环。

1.4.4 用户状态

用户的状态不应该保存在用户定义的 *Action* 中，每一个 *Action* 运行结束后，其所保存的状态信息也会立即失效。用户唯一可以保存信息的地方是那些用在 __break_if，__redo_if 及其语法糖里的 **谓词**。

Transaction DSL 保证，所有这些谓词里所持有的状态信息，和循环的生命周期一致。即只要一个循环没有运行结束，无论其在内部循环了多少次，在循环内对于这些状态的修改，始终保持连续有效。

因而，我们就可以定义这样的谓词：

```
struct ShouldRetry {
   bool operator()(const TransactionInfo& trans) {
      return IsFailed(trans) && retryTime++ < 5;
   }
 private:
   int retryTimes = 0;
}
```

```
__loop(
, __sync(Action1)
, __asyn(Action2)
, __time_guard(TIMER_2, __asyn(Action3))
, __concurrent(__asyn(Action3), __asyn(Action5))
, __while(ShouldRetry));
```

这样，整个循环内部的操作在连续失败 5 次之前，不会结束。

> **注意：** 在 __loop 里，只有与循环控制有关的谓词，其状态的连续性才会得到保证。在正常 Action 的普通谓词，比如：__optional(__not(ShouldRetry), __sync(Action1))，其中的谓词 ShouldRetry 的状态连续性无法得到保证。

1.4.5 错误处理

首先，整个 __loop 有一个自己的 **运行时环境**，而这个运行时环境是一个 *Sandbox* ，即它内部所发生的任何错误，在整个 __loop 没有结束之前，外界无从感知，因而对外界并无任何影响。

动作段与谓词段

在进一步描述 __loop 的错误处理之前，我们先来看两个概念：

动作段：Action Segment 一个或多个 连续的 **动作**（Action）

谓词段：Predicate Segment 一个或多个 连续的 **谓词**（Predicate）

比如下面的代码里，__loop 一共可以划分为 5 个段：3 个 **动作段**，2 个 **谓词段**：

```
__loop(
// Action Segment 1
  __sync(Action1)
, __asyn(Action2)

// Predicate Segment 1
, __break_if(__is_status(FATAL_BUG))
, __redo_if(__is_failed)

// Action Segment 2
, __asyn(Action3)
, __asyn(Action4)
, __time_guard(TIMER_2, __asyn(Action5))

// Predicate Segment 2
, __break_if(__is_timeout)
, __redo_if(__is_failed)

// Action Segment 3
, __concurrent(__asyn(Action6), __asyn(Action7))
);
```

对于任何一个 **动作段**，如果执行到某个 *Action* ，出了错，则此段后续的所有 *Action* 将都会被跳过。比如，本例子中的 *Action Segment 2* 一共包含了 3 个 Action，如果 Action3 的执行出了错，则后续的 Action4 ，Action5 都会被跳过。

当然，如果没有任何错误，一个 **动作段**里的所有 Action 会依次全部执行。

无论一个 **动作段**出没出错，都会进入紧随其后的 **谓词段**（如果存在的话）。如果之前出了错，在进入 **谓词段**之后，此错误总是可以被此 **谓词段**中的所有 **谓词**读取，以做为谓词判断的依据之一。

一个之前发生的错误，一旦离开最近的 **谓词段**之后，便会马上清理。比如，本例中的 **动作段 2** 如果出了错，

谓词段 2 中的所有谓词均可读取此错误。但是，一旦离开 **谓词段 2**，进入 **动作段 3**，此错误将会被清理。在 **动作段 3** 里的任意地方读取运行时上下文状态，总是会得到 SUCCESS 。

除非，**动作段 3** 里又发生了一个新错误，这样，**动作段 3** 将会终止其执行，**谓词段 3** 将可以读到新的错误。

对于最后一个 **动作段**的状态，如果重新回到循环的起始位置，而循环的起始位置是一个 **谓词段**，则此 **谓词段**可以读取最后一个 **动作段**的状态；如果循环的起始位置是一个 **动作段**，则最后一个 **动作段**的错误会首先被清理，以保证起始位置的 **动作段**可以从正确状态开始。比如，本例子中，Action Segment 3 的错误状态，会在 Action Segment 1 开始之前被清理。

但是，对于下面的例子，Action Segment 2 中的错误，在重新回到 Predicate Segment 1 时，依然可以被读取，直到 Predicate Segment 1 运行结束，错误才会被清理，以保证 Action Segment1 可以以正确状态开始。

```
__loop(
// Predicate Segment 1
  __break_if(__is_status(FATAL_BUG))
, __redo_if(__is_failed)

// Action Segment 1
, __asyn(Action3)
, __asyn(Action4)
, __time_guard(TIMER_2, __asyn(Action5))

// Predicate Segment 2
, __break_if(__is_timeout)
, __redo_if(__is_failed)

// Action Segment 2
, __concurrent(__asyn(Action6), __asyn(Action7)));
```

stop

当一个 __loop 被 stop 后，当前正在执行的 *Action* 会被 stop ，此 *Action* 被彻底 stop 后（有可能不能马上结束，需要进一步的消息激励后才能结束），返回的状态，则是整个 __loop 的返回状态。

死循环

如果一个 __loop ，运行一次完整的循环，其间却没有任何消息激励，那么很可能这个循环进入了死循环状态，这种情况下 __loop 会被强制终止，并返回 USER_FATAL_BUG 错误。比如下面的循环隐性的包含死循环状态，但 __loop 会在一个完整的循环之后，将其终止。

```
__loop
( __sync(Action1)
```

(下页继续)

(续上页)

```
, __sync(Action2)
, __sync(Action3));
```

如果用户想避免这样的检查，则可以使用：__loop_max 或者 __forever 以特别说明这的确是用户有意为之，而不是一个无意中犯下的错误。

比如：

```
__loop_max(1000
, __sync(Action1)
, __sync(Action2)
, __sync(Action3));
```

或者：

```
__forever
( __sync(Action1)
, __sync(Action2)
, __sync(Action3));
```

注意： __loop_max 与 __forever 并不意味着循环一定要永远循环下去，或者要循环到最大次数。循环里仍然可以设置谓词，当谓词条件满足时，__break_if 及其语法糖，将可能更早的终止循环。

另外，__loop_max 所指定的次数，指的是无消息激励的情况下，最大的循环次数。在有消息激励的情况下，指定的次数不起作用，因而即便超过指定次数也没有任何关系。当 无消息激励循环的次数达到最大时，会返回 USER_FATAL_BUG 错误。因而，在使用中，必须将次数设置的比所需的更大，才能避免这种错误。

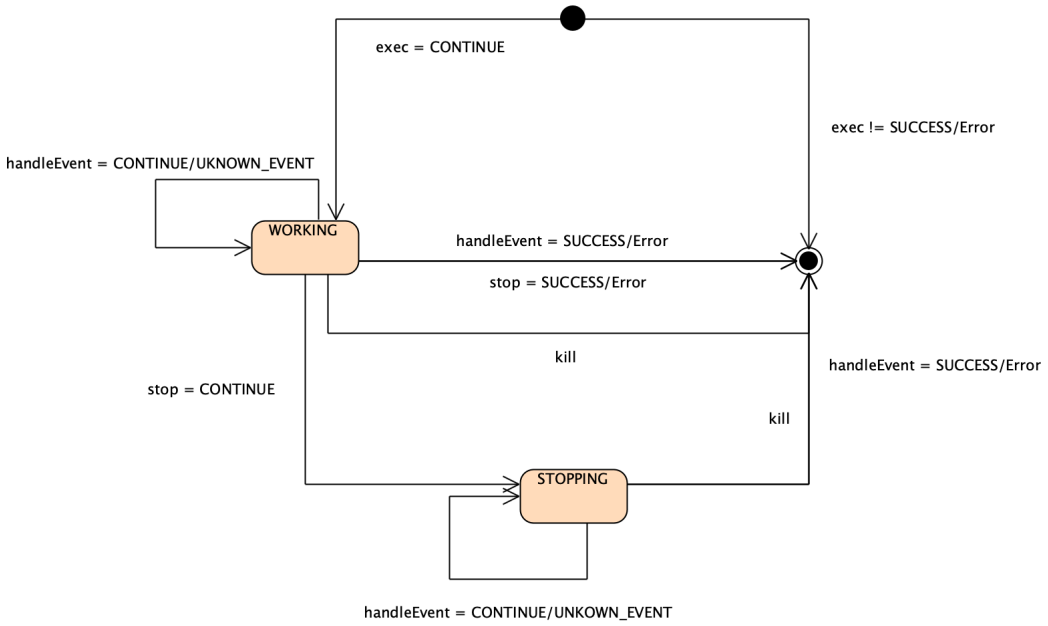
1.5 错误处理

当我们为 *Transaction DSL* 框架添加一个 *Action* 时，这个 *Action* 会组合其它的 *Action* 。比如，__sequential *Action* ，里面会放入一系列的其它 *Action* ，而 __procedure 则会包含两个 *Action* ，一个是 *normal Action* ，一个是 __finally Action。

做为 *Action* 的编写者，你不能假设你的 *Action* 所组合的是那种具体的 *Action* 。因而任何 *Action* 都必须遵从某种约定。所有的 *Action* ，在组合其它 *Action* 时，唯一可以作出的假设是每一个 *Action* 都遵从这些约定。下面我们将会讨论这些约定。

1.5.1 Action 外部行为规范

Action 从外部看，总共有 4 个状态，它们的状态转换关系如下图所示：



下面我们对其进行详细说明。

IDLE

任何一个 *Action* ，单纯从外部看，在没有发生任何调用之前，*Action* 必然处于*IDLE* 状态。

而*IDLE* 状态下，唯一合法的调用是 `exec` ，如果 `exec` 返回 `CONTINUE` 代表 Action 进入*WORKING* 状态。而*WORKING* 的含义是，此 *Action* 需要进一步的异步消息激励。

WORKING

在*WORKING* 状态下，

1. `exec` 不可再被调用，否则应返回 `FATAL_BUG` ；
2. 如果有事件到达，可以调用 `handleEvent` 进行处理；其可能结果如下：

- `SUCCESS` 代表 Action 进入*DONE* 状态；
- 任何错误值，也代表 Action 进入*DONE* 状态；
- `CONTINUE` 代表 Action 依然处于*WORKING* 状态；并且这条消息被 Action 成功的*accepted* 并处理，只是还需要进一步的消息激励；
- `UNKNOWN_EVENT` 表示消息并未被*accepted* ；

3. 如果调用 `stop` ，其可能结果如下：

- 如果返回 `CONTINUE` ，表示 Action 进入*STOPPING* 状态；
- 如果返回 `SUCCESS` ，表示 Action 进入*DONE* 状态；
- 如果返回错误值，表示 Action 进入*DONE* 状态；

4. 如果调用 `kill` ，Action 立即应进入*DONE* 状态。

STOPPING

在*STOPPING* 状态下，

1. `exec` 不可再被调用，否则应返回 `FATAL_BUG` ；
2. 如果调用 `stop`，不应对 Action 产生任何影响，而直接返回 `CONTINUE` ；
3. 如果调用 `kill` ，应立即进入*DONE* 状态
4. 如果调用 `handleEvent` , 其可能结果如下：

- `SUCCESS` 代表 Action 进入*DONE* 状态；
- 任何错误值，也代表 Action 进入*DONE* 状态；
- `CONTINUE` 代表 Action 依然处于*STOPPING* 状态；
- `UNKNOWN_EVENT` 表示消息并未被*accepted* ；

DONE

在*DONE* 状态下，

1. `exec` ， `stop` , `handleEvent` 都不可再被调用，否则应返回 `FATAL_BUG` ；
2. 如果调用 `kill` ，应该对 Action 状态无任何影响，依然处于*DONE* 状态。

注意:

- 一个 Action 的 `handleEvent` ，只要返回 `SUCCESS` ， `CONTINUE` ，包括大部分错误（某些错误，比如 `FATAL_BUG` ，表示在此 Action 已经处于不应该再被调用 `handleEvent` 的状态），都代表这条消息被 **accepted** ；
- 而返回 `UNKNOWN_EVENT` 则明确代表此消息没有被此 Action **accepted** 。
- 一个消息被 **accepted** ，并不代表一个消息被 **consumed** 。如果没有被 **consumed** ，代表此消息依然可以被其它 Action 处理。

当你的 Action 组合其它 Action 时，你对其它 Action 的假设，只需要符合上述外部行为规范即可。但对于我们将要实现的 Action 内部，我们也要进行一些概念上的定义，以保证 Action 与 Action 之间组合时，尤其在进行错误处理时，可以相互协调，保证整个 `Transaction` 行为的正确性。

1.5.2 Action 内部状态

I-IDLE: Action 已经被构造，但尚未调用 `exec` 之前。

I-DONE: Action 已经结束其处理，无论成功还是失败。

如果一个 Action 在调用 `exec` 之后，直接返回 `SUCCESS` 或者任何错误，代表这个 Action 已经进入*I-DONE* 状态。

如果一个 Action 在调用 `exec` 之后，直接返回 `CONTINUE` ，代表这个 Action 已经进入*I-WORKING* 或者*I-STOPPING* 状态。这一点与外部的观察并不一致，因为外部无法从 `CONTINUE` 返回值辨别出其内部处于二者中的哪一种。

无论是哪一种，从外部看，这个 Action 都还没有运行结束，因而需要进一步的消息激励。但从内部看，却有着本质的区别：

I-WORKING: 状态却表示其处于正常处理状态；

I-STOPPING: 则代表 Action 内部已经进入异常处理状态。

如果内部处于*I-WORKING* 状态，如果一个 Action 未处于*免疫模式* ，则 `stop` 调用应强迫 Action 进入失败处理。

1.5.3 错误传播

方式

注意: 错误的传播，主要有三种方式：

1. 最直接，也是最典型的，通过 返回值。这发生于一个 Action 运行结束，进入*I-DONE* 状态时；这属于一个从 内层上下文向 外层上下文传播错误的方式。
2. 但一个 Action 内部发生错误后，并没有直接进入*I-DONE* 状态，而是需要进一步的消息激励，因而会处于*I-WORKING* 或*I-STOPPING* 状态。但此错误需要立即为外界所感知，从而尽快对此错误作出响应。此时，可以通过 运行时上下文的嵌套父子关系，由 内层上下文直接逐级上报，向外传播；
3. 外层上下文由于任何原因，比如，最典型的原因是，通过内层 Action 的返回值，或者 内层上下文的上报，得到了一个错误，需要将错误传递给其它 内层上下文。此时，可以通过 `stop` 调用，带着 cause 值，将错误 由外向内传播。

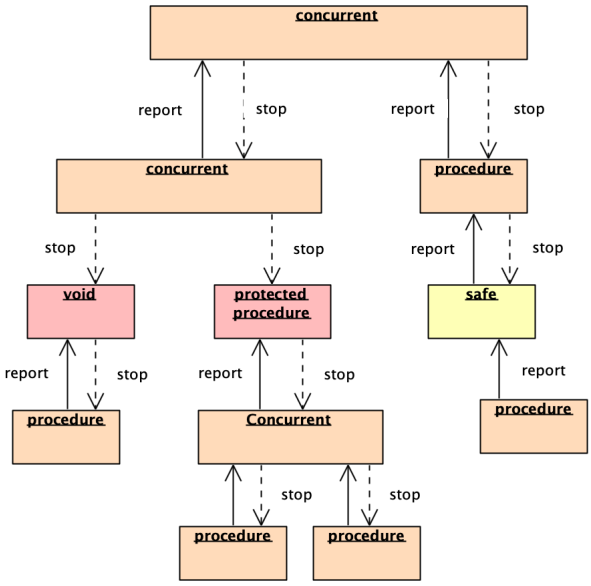
简单的说就是：

1. 由内向外传播

- 内层 Action 的返回值（此时 Action 进入*I-DONE* 状态）
- 内层上下文向 外层上下文的直接传递（此时调用返回值是 `CONTINUE`， 因而 Action 处于*I-WORKING* 或*I-STOPPING* 状态）

2. 由外向内传播：

- `stop(cause)`



模式

每一个可嵌套 Action 都有 4 种模式：

正常模式：Normal Mode 错误既可以向内传播，也可以向外传播；

沙箱模式：Sandbox Mode

- 错误不可通过 运行时上下文向外传播
- 可能允许通过 返回值返回最终的错误；
- 允许外部的错误通过 `stop` 传播进来；

免疫模式：Immune Mode

- 错误不可向内传播
- 但允许内部的错误通过 运行时上下文或者 返回值向外传播

孤岛模式：Island Mode

- 同时处于沙箱模式 和免疫模式

注意： 每一个可嵌套 Action 的设计，必须遵从如下原则：

- 如果本来处于正常模式，一旦被调用 stop，如果 stop 没有导致其进入*I-DONE* 状态，则必然进入免疫模式；随后再次调用其 stop 将会被阻断，直接返回 CONTINUE，而不会对其产生任何影响；
- 如果处于正常模式或免疫模式，在内部发生错误后，如果随后不能立即结束，则必须通过 运行时上下文及时上报错误；
- 一旦通过 运行时上下文上报过一次错误，则随后再发生的错误，禁止再通过 运行时上下文上报。这就意味着，进入了沙箱模式（从正常模式）或孤岛模式（从免疫模式）。

stop 的设计原则

注意： stop (立即结束的情况) 或随后的 handleEvent （经多次消息激励后的情况）的返回值原则如下：

- 如果 stop 并没有导致一个 Action 处理失败，即 Action 依然完成了它本来的职责，则依然返回 SUCCESS；
- 如果 stop 本身没有失败，但 Action 并没有完成它本来应该完成的任务，则返回 stop cause；
- 如果 stop 导致了的其它失败，则返回其它错误 (*Loop* 除外)；

1.5.4 部分 Action 行为定义

__asyn

注意： 当一个 __asyn 处于*I-WORKING* 状态，即其正在等待消息激励时，如果被调用 stop：

- 如果用户实现有错误（返回 CONTINUE 却发现其并没有等待任何消息），直接返回 USER_FATAL_BUG。
- 否则，返回 stop cause。

注意： 当一个 __asyn 处于*I-WORKING* 状态，某次调度时发生一个内部错误，则应该返回此错误，并进入*I-DONE* 状态。

__sequential

注意： 当 __sequential 处于*I-WORKING* 状态，如果此时调用其 stop：

1. 立即对当前 action 调用 stop，将 cause 值透传；
2. 如果其立即返回错误，则直接将此错误返回；进入*I-DONE* 状态；
3. 如果立即返回 SUCCESS，也进入*I-DONE* 状态：
   - 如果这是 __sequential 序列的最后一个 action，则返回 SUCCESS；
   - 否则，返回 stop cause。
4. 如果当前 action 并未直接结束，而是返回 CONTINUE，则进入孤岛模式；
5. 等某次调用 handleEvent 返回 SUCCESS 或错误时，其处理与 2，3 所描述的方式相同。

注意： 当 __sequential 处于*I-WORKING* 状态，如果其中某一个 action 发生错误：

- 直接返回此错误，进入*I-DONE* 状态。

__concurrent

注意： 当 __concurrent 处于*I-WORKING* 状态，如果此时调用其 stop：

- stop 每一个处于*I-WORKING* 状态的线程，将 cause 值继续往内层传递；
- 如果所有的线程都最终以 SUCCESS 结束，则返回 SUCCESS；
- 如果某个或某些线程返回任何错误，整个 __concurrent 结束时，返回最后一个错误。

注意： 当 __concurrent 处于*I-WORKING* 状态，此时某一个线程发生错误：

- 记录下此错误；
- 对其余任何还处于*I-WORKING* 状态的线程，调用其 stop，原因为刚刚发生的错误；
- 如果某个线程最终返回 stop cause，忽略此错误；
- 在整个 stop 过程中，坚持使用同一个原因值；哪怕某些线程立即返回其它错误值；
- 如果在整个 stop 过程中，有一个或多个直接返回其它错误值（非 stop cause），等 stop 调用完成后，将最后一个错误记录下来，更新原来的错误值；

- 如果所有线程都在调用 `stop` 后立即结束，则直接返回最后一个错误值；进入*I-DONE* 状态；
- 如果仍然有一个或多个线程，其 `stop` 调用返回 `CONTINUE` ，则 `__concurrent` 应直接给外层上下文通报最后一个错误，并返回 `CONTINUE` ，由此进入孤岛模式 以及*I-STOPPING* 状态。
- 随后在 `handleEvent` 的过程中，返回的每一个错误，都即不向外扩散，也不向内扩散；仅仅更新自己的 last error；
- 最终结束后，返回最后一个错误值。进入*I-DONE* 状态。

`__procedure`

`__procedure` 分为两个部分：Normal Action，与 `__finally` Action。

注意： Normal Action 的执行如果处于*I-WORKING* 状态，此时进行 `stop`：

1. 直接对 Normal Action 调用 `stop`；
   - 如果直接返回 `SUCCESS` ，则直接以成功状态，进入 `__finally`；
   - 如果直接返回错误，则直接以错误进入 `__finally`；
   - 两种情况下，在 `__finally` 里读到的环境状态都是 Normal Action 结束时的返回值；
2. 如果 Normal Action 返回 `CONTINUE` ，则 `__procedure` 进入孤岛模式 。
3. 随后 Normal Action 的 `__handleEvent` 如果返回 `SUCCESS` 或错误，其处理方式与 1 所描述的情况相同；

注意： Normal Action 的执行如果处于*I-WORKING* 状态，如果此时其内部上报了一个错误，但 Normal Action 的执行并没有立即结束（返回 `CONTINUE` ）：

1. 记录并继续通过 运行时上下文向外传递此错误；并进入孤岛模式 ；
2. 继续调度 Normal Action 运行直到其结束；
3. 如果 Normal Action 最终返回一个错误（理应返回一个错误），记录下此错误；
4. Normal Action 结束后，直接进入 `__finally` ，在 `__finally` 里读到的环境状态之前发生的最后一个错误值；

注意：

- 无论任何原因，一旦开始执行 `__finally` Action，将直接进入免疫模式 （也可能是孤岛模式 ）；

- 在进入 __finally 之后，如果仅仅是免疫模式，而不是孤岛模式，则依然可以给外围环境通报错误；

protected __procedure

注意： 一个处于*I-WORKING* 状态的 protected __procedure 可以被 stop，其处理方式与*procedure stop* 相同。

注意： protected __procedure 天然处于沙箱模式，即，直到其运行结束之前，不会向外围运行时上下文通报任何错误。

__time_guard

注意： 一个处于*I-WORKING* 状态的 __time_guard 被 stop 后，action 会首先被 stop：

1. 如果 stop 导致 action 立即结束，此时 timer 也会被 stop，并返回 action 的执行结果；
2. 如果 stop 后，action 依然没有结束运行（返回 CONTINUE），则定时器也不终止；但 __time_guard 立即进入免疫模式；stop 之后，经过一系列的消息激励，直到运行结束：

- 如果期间没有 timeout，则以 action 的最终返回值做为 __time_guard 的返回值；

注意： 一个处于*I-WORKING* 状态的 __time_guard 在运行期间，监测到一个由 action 上报的一个内部错误，则立即进入免疫模式。之后，经过一系列的消息激励，直到运行结束：

- 如果期间没有 timeout，则以 action 的最终返回值做为 __time_guard 的返回值；

1.6 片段

到目前为止，我们给出的例子虽然并不非常复杂，但某些例子的描述已经开始显的不太清晰。

而良好软件设计的核心原则之一就是 **清晰性**，对于其它编程语言而言，满足清晰性的一个重要措施就是允许用户定义更小的，职责更加单一的小函数，或者小类。

而通过 *Transaction DSL*，你同样可以将一个复杂过程的任何一个局部进行提取，以定义更小的，职责更加单一的子操作：**片段** (*Fragment*)。

1.6.1 __def

定义一个片段的方式，就是通过 __def 。比如:

```
__def(Fragment) __as
( __asyn(Action1)
, __asyn(Action2));
```

从例子中可以看出，一个片段的定义包含两个部分：首先，__def 的参数，定义了一个 **片段**的 **名字**；然后在 __as 里，定义了这个片段所执行的操作。

之所以会有 __as ，主要是作为一门以 *C++* 为母语言的 DSL，在 *C++* 的语法范围内，尚未找到一种更好的实现方法。

1.6.2 __apply

一旦一个 **片段**被定义，就可以在另外一段操作中引用它。比如:

```
__transaction
( __asyn(Action1)
, __apply(Fragment));
```

现在，我们就可以将 procedure 中的例子的两个片段提取出来:

```
__def(Fragment1) __as
( __asyn(Action1)
, __asyn(Action2))
, __finally(__rsp(Action3)));

__def(Fragment2) __as
( __asyn(Action5)
, __finally(__sync(Action6)));
```

最后，我们的事务描述就变成了下面的样子。和之前的例子对比一下，是否清晰了许多。

```
__transaction
( __apply(Fragment1)
, __asyn(Action4)
, __apply(Fragment2));
```

1.6.3 __params

对于软件设计而言，另外一个重要的指标是 **可重用性**。之前定义的子操作，虽然最早的出发点是为了清晰性，但事实上，也间接的让这些片段变得可以复用。

但一旦到了复用的层次，我们就需要更加强大的设施。比如，存在如下两个事务:

```
__transaction
( __time_guard
    ( TIMER_1
    , __asyn(Action2)
    , __asyn(Action3))
, __asyn(Action4));

__transaction
( __concurrent
    ( __asyn(Action1)
    , __time_guard
        ( TIMER_2
        , __asyn(Action4)
        , __asyn(Action3))));
```

稍微仔细的观察，可以发现，两个事务有一段非常相似的过程:

```
__time_guard
( TIMER_1
, __asyn(Action2)
, __asyn(Action3));

__time_guard
( TIMER_2
, __asyn(Action4)
, __asyn(Action3));
```

两段子操作看起来很相似，但中间有两处细节是不同的。对于这类问题，我们可以将它们合二为一，然后将差异参数化:

```
__def(Seq2, __params(__timer_id(TIMER), __action(ACTION))) __as
( __time_guard
    ( TIMER
    , __asyn(ACTION)
    , __asyn(Action3)));
```

在这个例子中，我们定义了一个名为 Seq2 的片段，__params 里放置的是它的两个参数: TIMER 和 ACTION 。前者的修饰 __timer_id 说明参数是一个 Timer ID ，后者则由 __action 修饰，说明它是一个操作。这些修饰就相当于其它编程语言里的参数类型说明。

这两个参数随后在 __as 里所定义的操作里得到了使用，其中 TIMER 被用在了 __time_guard 里，而 ACTION 则用在了第一个 __asyn 里。

1.6.4 __with

调用一个带参数的片段，同样是使用 __apply ，不过，需要通过 __with 来指明实参。现在，我们就可以将之前的两个事务修改为:

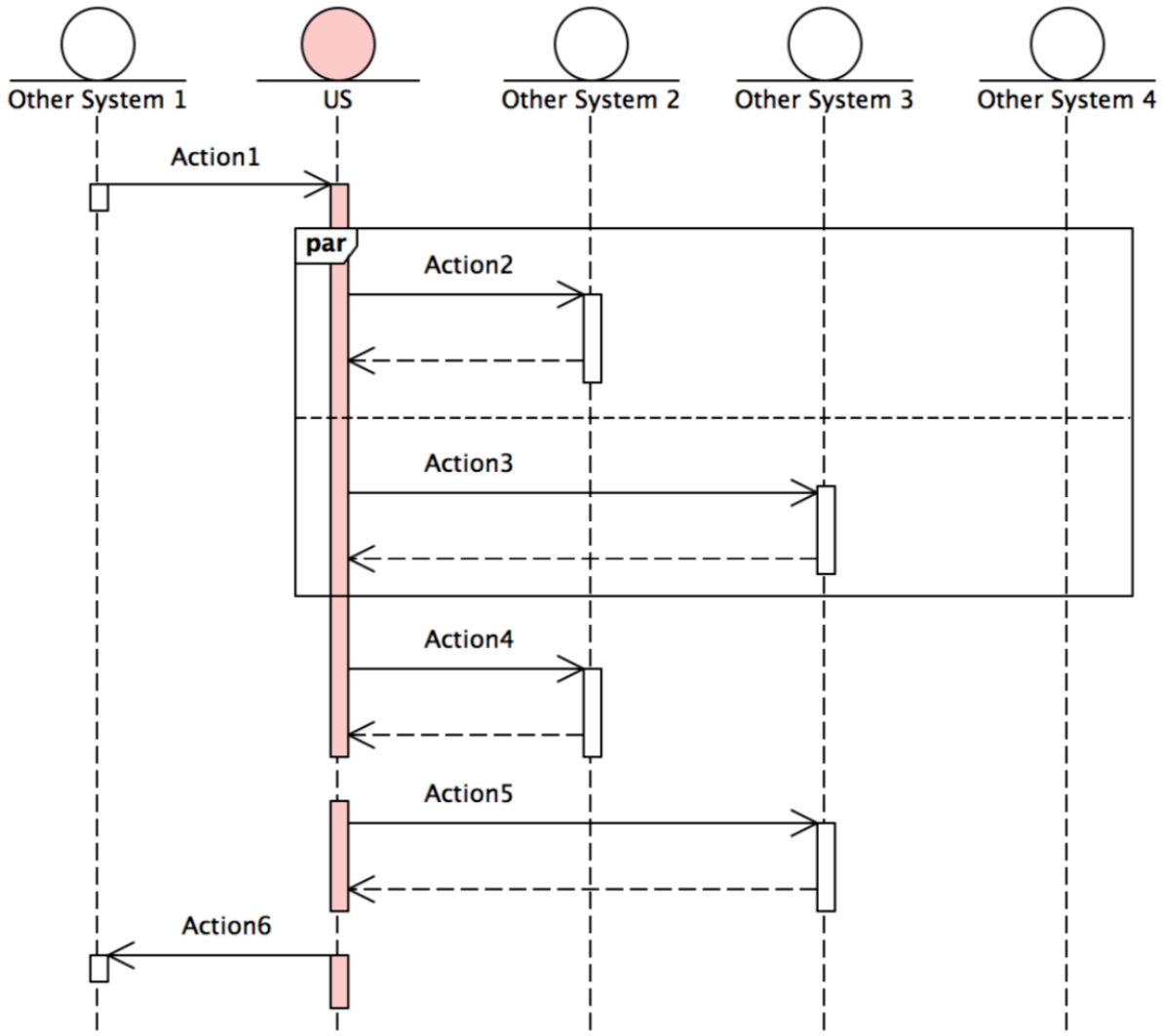
```
__transaction
( __apply(Seq2, __with(TIMER_1, Action2))
, __asyn(Action4));

__transaction
( __concurrent
    ( __asyn(Action1)
    , __apply(Seq2, __with(TIMER_2, Action4)));
```

1.7 多线程

我们先来看一个熟悉的例子。

在下图中，包含了 6 个 *Action* :

1. 首先，*US* 会执行异步操作 *Action1* ，以等待一个请求消息;
2. 随后，*Action2* 和 *Action3* 是两个并发进行的异步操作;
3. 在 *Action2* 和 *Action3* 完成之后，*Action4* 和 *Action5* 同样是两个异步操作, 会依次顺序执行;
4. 最后，执行同步操作 *Action6* ，以回复一个响应消息。

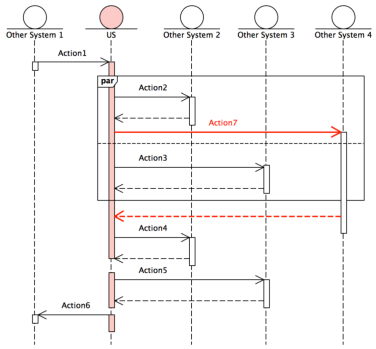


这个过程简单而清晰，我们可以轻易的使用 *Transaction DSL* 对其进行描述。

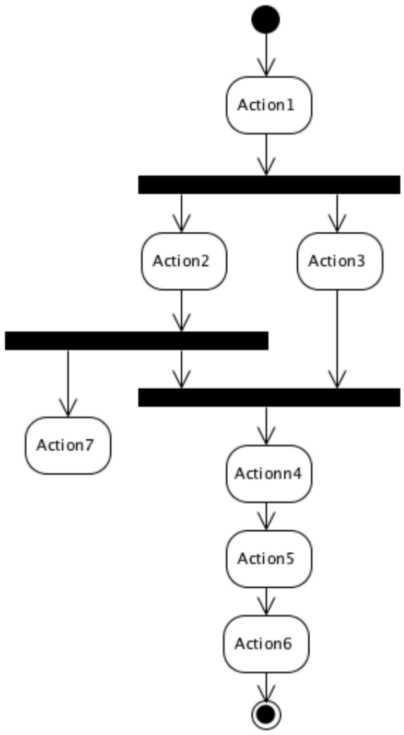
```
__transaction
( __req(Action1)
, __concurrent(__asyn(Action2), __asyn(Action3)) , __asyn(Action4)
, __asyn(Action5)
, __rsp(Action6)));
```

但事情并非总是这么简单。如下图所示，有一天，需求迫使你不得不在原有流程上加入一个新的异步处理：在 *Action2* 收到其应答消息之后，需要马上发出一条新消息，但却无须停下来等待其应答，整个流程仍然按照原有的方式往前走。在这期间，应答消息随时可能到来。一旦其收到这条应答，系统应该马上对其进行处理。

而这个新增的请求-应答异步操作，就是下图中的 *Action7* :



这样的过程，用 **序列图**可能不能准确的表达其过程，而 **活动图**会让其更加直观：



1.7.1 __fork

上图已经揭示了问题的本质：新加入的 *Action7* 已经脱离了本来的控制序列，独自形成了另外一条控制序列。如果我们将一条控制序列看作一个线程，那么，我们已经在面对多线程的问题。

在 *Transaction DSL* 中，我们可以使用 `__fork` 来创建一条新线程。例如:

```
const ActionThreadId ACTION7_THREAD = 1;

__fork(ACTION7_THREAD, __asyn(Action7))
```

正如例子所揭示的，`__fork` 有两个参数：

第一个参数是 *Thread ID* ，随后我们会谈到。

第二个参数则是在这条线程上执行的操作。在一条线程的执行的操作，在本例中非常简单；但事实上，它可以是任意复杂的操作，比如:

```
__fork(THREAD1, __sequential
                  ( __concurrent(__asyn(Action1), __asyn(Action2)) , __asyn(Action2)
                  , __time_guard(TIMER2, __asyn(Action3)))
```

而对于我们这个例子，其完整的描述如下:

```
const ActionThreadId ACTION7_THREAD = 1;

__transaction
( __req(Action1)
, __concurrent
    ( __sequential
        ( __asyn(Action2)
        , __fork(ACTION7_THREAD, __asyn(Action7)))
    , __asyn(Action3))
, __asyn(Action4)
, __asyn(Action5)
, __rsp(Action6));
```

其中，我们将 Action2 和 ACTION7_THREAD 放在了一个 __sequential 操作里。这是因为，我们必须确保，ACTION7_THREAD 的创建发生在 Action2 的成功执行结束之后。同时，我们也必须确保，这个序列，和 Action3 是并行发生的。

此线程非彼线程

尽管 *Transaction DSL* 引入了线程的概念，但这里的线程与操作系统或任何平台所提供的线程之间没有任何关系。它是一个由事务框架创建，管理和调度的实体，操作系统或平台完全意识不到它们的存在。

其执行和调度粒度是以 *Action* 为单位的，不像操作系统级别的线程是以硬件指令为单位的。在一个抢占式的系统中，操作系统级别的线程，可以在允许的任何指令处暂停或中止一个线程的运行，但 *Transaction DSL* 的线程则完全做不到这一点，当然，它也没有任何必要去做到这一点。

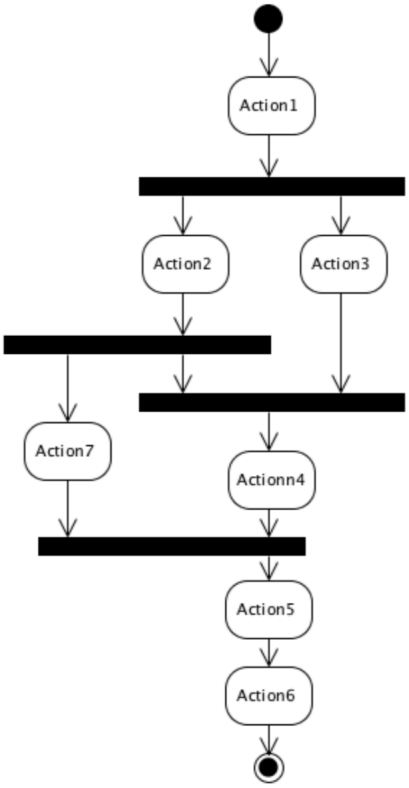
所以，在 *Transaction DSL* 里，其准确的名字是操作线程 (Action Thread)。

1.7.2 __join

当创建了一条线程时，它就和其它的线程——包括主线程——各自独立的并发执行，互不干扰，老死不相往来。

但是，在现实的系统中，往往又存在这这样的需求，即，虽然同时存在多个线程，但其中某个线程的执行过程中，某个步骤的后续执行，是以另外一条线程的执行结束为前提的。

比如，我们可以给之前的例子增加一个约束：`Action5` 开始执行之前，要求 `Action7` 必须执行完毕。如图所示：



对于这样的约束，你可以使用 `__join` 来描述。它的参数，就是某个线程要等待的 *Thread ID* 。例如:

```
// ...
__fork(THREAD1, __asyn(Action1))
// ...
__join(THREAD1)
// ...
```

将其应用于我们的例子，其完整的描述如下:

```
const ActionThreadId ACTION7_THREAD = 1;


__transaction
```

(下页继续)

(续上页)

```
( __req(Action1)
, __concurrent
    ( __sequential
        ( __asyn(Action2)
        , __fork(ACTION7_THREAD, __asyn(Action7)))
    , __asyn(Action3))
, __asyn(Action4)
, __join(ACTION7_THREAD)
, __asyn(Action5)
, __rsp(Action6));
```

在 __join 时，如果被 join 的线程已经执行完毕，则 __join 马上完成。否则，__join 所在线程将在 __join 处一直等待，直到目标线程运行结束。

如果一个线程 __join 它自己，会马上成功完成。

Thread ID

Thread ID ，标示了一个线程的身份，所以，在一个事务中，每个线程的 *Thread ID* 必须唯一。在目前的实现中，其取值范围为 0 到 7 ，但 0 是主线程的 *ID* ，用户不能使用。所以，在一个事务中，用户最多允许创建 7 个线程。

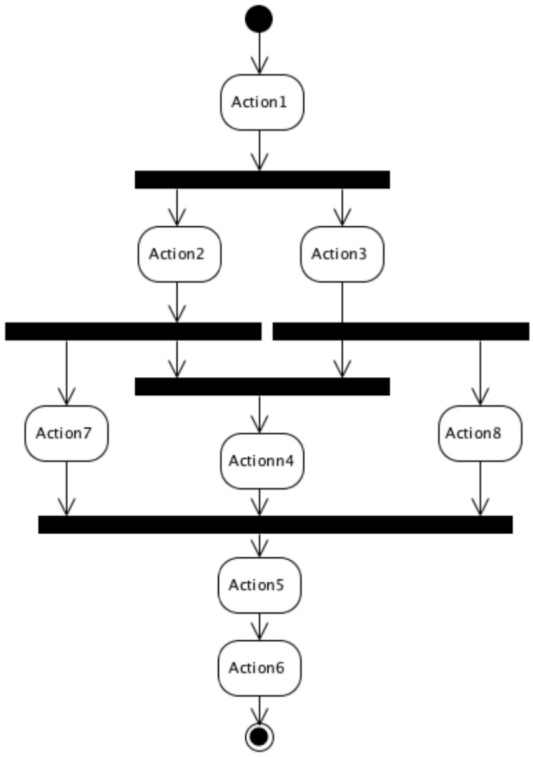
由于 __join 机制的存在，在 __join 时，用户必须有一种手段，来指明具体的线程。所以，每个线程必须有一个唯一的身份标识。

从实现手段上，这个标识可以是一个字符串，从而避免让用户需要亲自来分配和管理 *Thread ID* 。但是，从语言的约束和实现的复杂度上，用整数作为标识，是最为简单的。虽然这略微增加了用户的负担，但却避免了框架实现的复杂度。毕竟，用户最多只能在一个事务中创建 7 个线程，这仍然在人类可轻松管理的范围内。

同时等待多个线程

有些时候，一个线程的继续执行，是以多个线程的执行结束为条件的。这种情况下，你仍然使用 __join 。

比如，在下图中所描述的事务中，Action2 和 Action3 在执行结束后，分别启动了一个线程，并发的运行 Action7 和 Action8 ，随后，在执行 Action5 之前，要求这两个线程都必须执行结束。

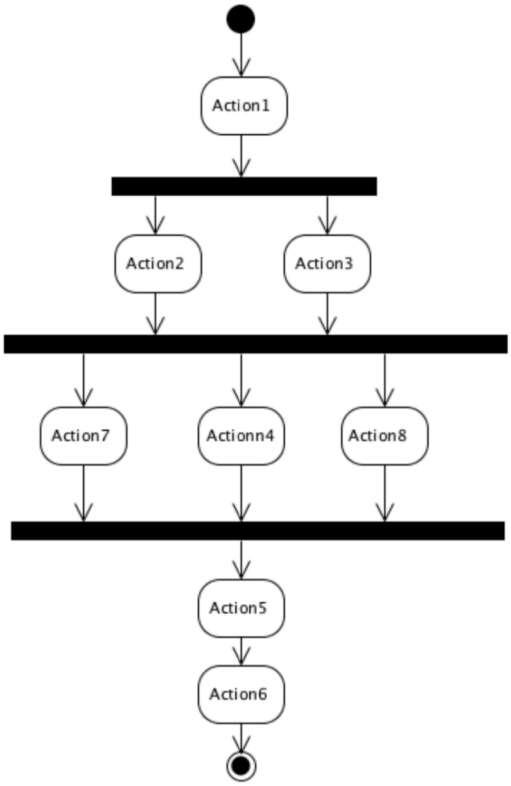


可描述为：

```
const ActionThreadId ACTION7_THREAD = 1;
const ActionThreadId ACTION8_THREAD = 2;

__transaction
( __req(Action1)
, __concurrent
    ( __sequential
        ( __asyn(Action2)
        , __fork(ACTION7_THREAD, __asyn(Action7)))
    , __sequential
        ( __asyn(Action3)
        , __fork(ACTION8_THREAD, __asyn(Action8))))
, __asyn(Action4)
, __join(ACTION7_THREAD, ACTION8_THREAD) , __asyn(Action5)
, __rsp(Action6));
```

__join 是一个变参操作，最多可以等待 7 个线程。因为每个事务的最大线程数量是 8 个。所以，每个线程都可以等待所有其它线程。

或许你会敏锐的发现，对于下图所描述的事务，和上图中所描述的事务是等价的。



所以，你会希望将代码写成这种形式，从而减少对于线程的操作，也让代码看来更加的简洁。

```
__transaction
( __req(Action1)
, __concurrent(__asyn(Action2), __asyn(Action3))
, __concurrent(__asyn(Action7), __asyn(Action4), __asyn(Action8))
, __asyn(Action5)
, __rsp(Action6));
```

不幸的是，尽管它们看起来很相似，但它们的实时性和性能却并不相同（想像一下，Action2 和 Action8 是慢速操作，而 Action3 和 Action7 是快速操作，对比一下两者的性能)。而对于实时性和性能的追求，正是我们使用并发模型的原因，不是吗?

匿名线程

你应该早就已经意识到，在 __concurrent 里的多个操作，和通过 __fork 创建线程执行的操作都是并发操作。所以，__concurrent 里的每个 Action 也都是线程。不同的是，它们没有自己明确的身份：Thread ID 。所以，直接被放在 __concurrent 里的线程被称为匿名线程。之所以它们不需要 Thread ID ，是因为 __concurrent 本身已经保证了这些线程会被自动的 __join ，比如:

```
__concurrent(__asyn(Action1), __asyn(Action2))
```

从控制过程看，就近似的等价于:

```
__fork(TID1, __asyn(Action1)),
__fork(TID2, __asyn(Action2)),
__join(TID1, TID2)
```

但很明显，前一种写法更加简单明确。另外，匿名线程的一个重要优势是：没有数量上的约束。在一个事务内部，你可以根据需要创建任意多个匿名线程。其实，匿名线程和有名线程之间的差别还有很多，我们会在其它相关的部分进行讨论。

1.7.3 调度策略

对于一个事务而言，即便存在多条线程，但只要主线程执行结束，整个事务就执行结束。此时，其它线程执行到什么阶段，都不会影响一个事务的 exec 或 handleEvent 函数的返回值（你应该还记得，其返回值为 CONTINUE 表示一个事务仍在工作，而 SUCCESS 则表示其已经成功结束)。

当主线程结束时，所有其它正在工作的有名线程将会被强行中止。所以，一个用户创建的有名线程 __join 主线程是没有意义的。

join all

如果你期望所有的线程都结束之后，整个事务才能结束，那么你应该在主线程使用 __join ，但不指定任何具体 Thread ID 来等待 **所有**其它线程结束。

```
__transaction
( __fork(THREAD1, __asyn(Action1))
, __fork(THREAD2, __asyn(Action2))
, __asyn(Action3)
, __join());
```

如果主线程是一个 __procedure ，那么就应该在 __finally 里 __join，比如:

```
__transaction
( __fork(THREAD1, __asyn(Action1))
, __fork(THREAD2, __asyn(Action2))
, __asyn(Action3))
, __finally
    ( __asyn(Action4)
    , __join()));
```

注意，__join 并不关心它所等待的线程是以成功还是失败，而只关心它们是否已经结束。

线程错误

Transaction DSL 对于错误的应对哲学是：**尽早失败** (*Fail Fast*)。因为，一旦一个事务中的任何一点发生了不可修复的错误，那么就应该让整个事务的所有线程都进入失败处理。否则，将会导致其它线程的不必要的行为浪费。

有名线程的失败

如果线程启动时发生错误，__fork 以失败结束。比如在下面的过程里：

```
__transaction
( __fork(THREAD1, __asyn(Action1))
, __asyn(Action2))
, __finally(__on_fail(__asyn(Action3))));
```

如果 Action1 的 exec 调用结果为某种错误，则 __fork 失败，从而 Action2 会被调过，直接进入 __finally ； 而由于 __fork 失败，因而 __on_fail 谓词判断结果为 true ， 所以，Action3 会得到执行。

一旦线程的 exec 执行结果是 SUCCESS ， 则 __fork 成功。

如果被创建线程 exec 的结果是 CONTINUE ， __fork 同样成功结束。而被创建线程开始独立运行，从此与创建者线程之间再无任何关系。

如果随后任何线程发生了错误，(无论是通过上下文汇报的错误，还是通过执行结果返回的错误)，都将导致对所有其它线程的 stop 调用。比如：

```
__transaction
  ( __fork(THREAD1, __asyn(Action1))
  , __fork(THREAD2, __asyn(Action2))
  , __asyn(Action3)
  , __asyn(Action4))
  , __recover
      ( __on_fail(__asyn(Action5))
      , __on_succ(__asyn(Action6))));
```

在 THREAD1 和 THREAD2 被成功 __fork 后，如果 Action2 在随后的处理过程中发生了错误，则主线程和 “THREAD1“ 都会被 stop ， 从而导致 THREAD1 的直接终止，而主线程的 Action3 也同样被终止，并跳过 Action4 ， 直接进入 __recover 。而由于这个错误，Action5 得到执行。由于 __recover ， 整个过程最终失败还是成功，取决于 Action5 的执行结果。

但是，如果在 THREAD1 或 THREAD2 发生任何错误之前，主线程已成功进入 __recover ，由于 __recover 让主线程处于 **免疫模式**，所以，其它线程发生的任何错误，都将无法再影响到主线程。从而也无法影响到整个事务的运行结果。

因而，如果你的确想感知其它线程的失败，则务必在进入 __finally 或 __recover 之前，进行 __join 。

匿名线程的失败

而匿名线程则不然，它的错误将会被创建线程捕捉到。如果发生错误的匿名线程处于其创建线程的某个 __procedure 内，则这个错误将可能被 __procedure 的 __recover 捕捉并修复。

比如，在下面的事务中，如果 Action1 发生失败，它将会中止 Action2 的执行，然后转向执行 Action4 ，如果 Action4 成功执行，则整个事务则成功结束。

```
__transaction
( __concurrent
    ( __asyn(Action1)
    , __asyn(Action2))
    , __asyn(Action3)
, __recover(__asyn(Action4)))
```

另外，当一个匿名线程失败后，其宿主有名线程必须等待匿名线程所处的整个 __concurrent 执行结束之后，才能进入 __finally 操作。比如在下面的事务中，如果 Action1 失败，它所在的匿名线程将会马上以失败结束。

由于其所处的 __concurrent 里，还存在另外一条匿名线程，所以，另外一条匿名线程也会进入失败处理，从而跳转执行 Action3 ； 由于 Action3 是一个异步操作，需要等待进一步的消息。所以，到目前为止，整个 __concurrent 并没有执行结束。

等 Action3 等到期待的消息并处理之后，__concurrent 里的两个匿名线程都结束了，从而导致整个 __concurrent 以错误的状态结束。

然后，其所处的 **有名线程**——在这里是 **主线程**——将会跳进 __finally ，去执行 Action5 ; 等 Action5 执行结束后，整个事务将以失败结束。

```
__transaction
( __concurrent
    ( __asyn(Action1)
    , __procedure(__asyn(Action2), __finally(__asyn(Action3))))
, __asyn(Action4)
, __finally(__asyn(Action5)))
```

尽管如此，当一个匿名线程失败时，仍然会及时的通知给整个事务，从而让事务内的其它线程可以尽早进入失败处理。

比如，在下面的事务里，如果匿名线程的 Action2 发生了失败，THREAD1 将会马上意识到这个错误并结束执行。而主线程的错误处理顺序则和上一个例子所描述的过程一样。

```
__transaction
( __fork(THREAD1, __asyn(Action1))
, __concurrent
    ( __asyn(Action2)
```

(下页继续)

(续上页)

```
    , __procedure(__asyn(Action3), __finally(__asyn(Action4))))
, __asyn(Action5)
, __recover(__asyn(Action6)))
```

__multi_thread 约束

1. 创建线程时，*Thread ID* 一定要从 1 开始，并且连续（如果创建多个其它线程时）；
2. 主线程一旦结束，则整个 __multi_thread 都将结束；而运行结果则是主线程的执行结果；
3. 其它线程不能 __join 主线程；
4. 其它线程不能 __join() ，即 *join all* ；
5. 其它线程进行 __join 时，不能得到保证；

1.8 事件

正常来讲，对于具体的事件不应该是 *Transaction DSL* 层面应该关注的事情，它需要做的就是将收到事件派发给由用户编写的基本操作。具体基本操作打算如何处理这些事件，那是那个层面应该关心的问题。

但在某些场景下，将对具体事件的关注放在 *Transaction DSL* 内，将会大大方便用户的实现。

1.8.1 __wait

首先的一种场景是，当一个事务执行到某个点的时候，会期待某个事件的发生，但是，事件的内容并不重要(或干脆没有内容)，此时，一个事件就像一个 signal 一样。如果由基本操作来关心这个事件，则用户不得不按照固定的模式编写一个类，以期待这个事件，并在收到这个事件后，返回 SUCCESS , 比如:

```
DEF_SIMPLE_ASYNC_ACTION(Action1) {
  auto exec(const TransactionInfo&) -> Status {
    return WAIT_ON(EV_SOMETHING, handleSomethingEvent);
  }
private:
  auto handleSomethingEvent(TransactionInfo const&, Event& const) -> Status {
     return SUCCESS;
  }
};
```

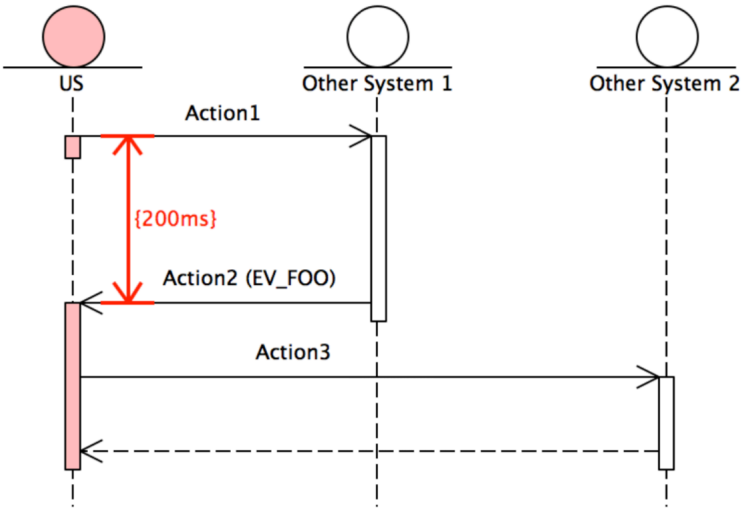
这无疑是一件相当无聊的工作。为了方便于实现这样的场景，用户可以在 *Transaction DSL* 中使用 __wait 来完成相同的事情。如下:

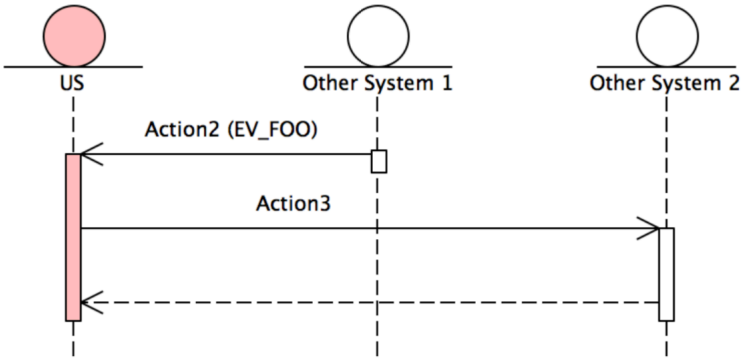
```
__transaction
( ...
, __wait(EV_SOMETHING)
, ... );
```

1.8.2 __peek

在一些场景下，一个消息可能两个操作都关注，但是，放在前面的操作仅仅关心消息的到达，而放在后面的操作则更关心消息的内容。

这种情况下，前者可以使用 __peek 来探测一个消息是否到达，但它并不处理这个消息，所以调度器还会进一步将消息传递给后续的流程。比如，下面的两个事务，事务 2 和事务 1 相比，只是多了一个主动发起的请求，以及等对方回应命令的时间约束。





这种情况下，事务 2 所描述的过程事实上是个可复用的整体。但是，由于事务 1 中的那个定时器约束的是从请求发出到命令到达的时间。所以，你很难从已有的手段中找到一种优雅简洁的表达来进行复用。

而借助于 __peek ，这个问题就迎刃而解。

```
__def(Trans2) __as
( __sync(Action2)
, __asyn(Action3));

__transaction
( __time_guard
    ( TIMER_1
    , __sync(Action1)
    , __peek(EV_FOO))
, __apply(Trans2));

__transaction
( __apply(Trans2) );
```

并不难看出，__peek 和 __wait 的唯一差别是，__wait 会把消息吃掉，而 __peek 则只看一眼，然后把消息留给后续的操作。

1.9 过程监控

在一个现实的系统中，对于个事务的执行过程往往有各种类型的追踪，比如，统计每个操作的运行时间，一个重要的步骤需要汇报给网关软件，以进行相关的指标统计，或者干脆就是调试的目的: 当出现问题时，希望知道究竟是哪个步骤出了问题。

将相关的代码都放入每个操作无疑会严重的污染操作的实现。从单一职责的角度，每个类都只应该关注一件事情。上述的需求都应该作为不同的关注点，从具体的业务代码中分离出去。

很明显，**观察者模式**是解决这类问题的良药。每个不同的过程监控需求都应该当做一个独立的观察者：比如，调试的观察者，某种指标统计的观察者，单步操作运行时间的观察者，等等等等。

1.9.1 观察者接口

每个观察者应该实现一个或者多个如下方法:

```
auto onActionStarting(ActionId, TransactionInfo const&) -> void;
auto onActionEventConsumed(ActionId, TransactionInfo const&, Event const&) -> void;
auto onActionDone(ActionId, TransactionInfo const&, Status) -> void;
auto onActionStopped(ActionId, TransactionInfo const&, Status) -> void;
auto onActionKilled(ActionId, TransactionInfo const&, Status) -> void;
```

注意，你无需继承自任何接口，直接在你的观察者类中选择实现你关心的方法，比如下面的观察者只关心 Action Done 事件:

```
struct MyObserver {
   auto onActionDone(ActionId, TransactionInfo const&, Status) -> void {
      // blah...
   }
};
```

方法	被调用时机
onActionStarting	一个操作开始执行之前 (exec 被调用之前)
onActionEventConsumed	一个事件被 handleEvent 接受 (无论其返回 SUCCESS ， CONTINUE 还是失败)
onActionDone	一个操作已经运行结束，无论是成功还是失败。
onActionStopped	一个操作已经被中止，无论成功或失败。
onActionKilled	一个操作已经被暴力杀掉（kill 被调用之后）

1.9.2 __with_id

你应该已经注意到，观察者的每一个方法都有一个类型为 ActionId 的参数。这个参数是为了唯一的标识一个操作。问题是，这些观察者如何知道哪个 Action ID 对应哪个操作?

答案是：由用户自己指定，指定的方式则是通过 __with_id 。通过 __with_id ，你不仅可以知道一个 Action ID 对应了哪个操作，你还可以选择是否观察一个操作。比如:

```
const ActionId ID_ACTION1 = 1;
const ActionId ID_ACTION4 = 2;

__transaction
( __procedure
    ( __with_id(ID_ACTION1,__asyn(Action1))
    , __asyn(Action2))
    , __finally(__rsp(Action3)))
, __with_id(ID_ACTION4, __asyn(Action4)));
```

在这个例子中，用户指定了要观察 Action1 和 Action4 ，并分别为它们指定了 Action ID 。随后，当此事务运行时，将只向观察者报告这两个操作有关的事件。

任意的粒度

在上一个例子中，两个被观察的操作都是用户定义的基本操作。但这并不是 *Transaction DSL* 的约束。

事实上，你可以观察的粒度可以是任意的。比如, 在下面的例子中，我们设置的观察粒度是整个事务:

```
__transaction
( __with_id
```

(下页继续)

(续上页)

```
    ( ID_TRANS
    , __procedure
        ( __asyn(Action1)
        , __asyn(Action2))
        , __finally(__rsp(Action3)))
    , __asyn(Action4)));
```

无论观察的粒度设为多大，在此粒度上的一切事件，都会汇报给观察者。

1.9.3 嵌套

如果观察者观察了一个大粒度的操作，同时，它依然可以观察此操作中更小粒度的操作。比如，在下面的例子中，ID_SEQ 所代表的操作，就属于 ID_TRANS 所代表的操作的子操作。

```
__transaction
( __with_id
    ( ID_TRANS
    , __procedure
        ( __with_id
            ( ID_SEQ
            , __asyn(Action1)
            , __asyn(Action2))
        , __finally(__rsp(Action3)))
    , __asyn(Action4)));
```

这种情况下，发生在子操作上的一切事件，会同时汇报给子操作和其父操作。

1.9.4 观察者的定义及注册

一个简单的事实是，任何一个 Transaction DSL 的关键字，背后都对应着一个内部 Action 的实现。只要你使用了它，你就潜在地为之付出空间和性能代价。

__with_id 也不例外。并且在一些现实项目中，维测类需求大都可以通过 观察者的方式监控所有 Transaction 的运行。因而会有很多 观察者。每个 观察者关注的 Action ID 也不尽相同。

另外，在运行时，这些维测类 观察者可能会随时被关闭，也可能随时被打开。而我们希望它们被关闭时，Transaction 在运行时无需为之付出任何代价（最好一个指令，一个字节都不付出）。

为了达到这个目标，框架要求你定义每一个 观察者时，需要通过 ObservedActionIdRegistry 来指明你关心的 Action ID 。比如：

```
struct MyListener1 : ObservedActionIdRegistry<ID_TRANS, ID_SEQ> {
  auto onActionDone(ActionId aid, TransactionInfo const&, Status) -> void {
```

(下页继续)

(续上页)

```
    switch(aid) {
    case ID_TRANS: // blah...
    case ID_SEQ:   // blah...
    }
  }
};

struct MyListener2 : ObservedActionIdRegistry<ID_TRANS> {
  auto onActionStarting(ActionId aid, TransactionInfo const&) -> void {
    switch(aid) {
    case ID_TRANS: // blah...
    }
  }
};
```

例子中，MyListener1 关心 2 个 Action ID：ID_TRANS 和 ID_SEQ；而 MyListener2 只关心 ID_TRANS。这需要通过继承 ObservedActionIdRegistry 并在模版参数里指明。

然后，你可以通过 __bind_listener，将这些 观察者注册给一个 Transaction：

```
__bind_listener(Transaction1, __listeners(MyListener1, MyListener2));
```

如果 Transaction1 的定义如下：

```
__def(Transaction1) __as_trans
( __with_id
    ( ID_TRANS
    , __with_id
        ( ID_SEQ
        , __with(ID_1, __sync(Action1))
        , __with(ID_2, __asyn(Action2)))
        , __finally(__rsp(Action3)))
    , __with_id(ID_4, __asyn(Action4))));
```

那么 bind_listener 之后，框架发现 ID_1，ID_2, ID_4 完全没有任何 观察者关心，会立即将对应的 __with_id 给优化掉。也就是说，无论从空间消耗，还是运行时性能，都完全等价于下面的 Transaction：

```
__def(Transaction1) __as_trans
( __with_id
    ( ID_TRANS
    , __with_id
        ( ID_SEQ
        , __asyn(Action1)
```

(下页继续)

(续上页)

```
        , __asyn(Action2))
        , __finally(__rsp(Action3)))
    , __asyn(Action4)));
```

而对于下面这个例子，

```
__def(Transaction2) __as_trans
( __with_id
    ( ID_TRANS
    , __with_id
        ( ID_SEQ
        , __asyn(Action1)
        , __asyn(Action2)
        , __asyn(Action3))
    , __asyn(Action4)));
```

如果连 ID_SEQ 也无人关注，那么优化掉的将不仅仅是 __with_id 。因为这个 __with_id 内部是一个隐含的 __sequential ，由于外面 ID_TRANS 也是一个隐含的 __sequential 结构，所以会发生内部 __sequential 的*inline* ，从而让其结果与如下形式等价：

```
__def(Transaction2) __as_trans
( __with_id
    ( ID_TRANS
    , __asyn(Action1)
    , __asyn(Action2)
    , __asyn(Action3)
    , __asyn(Action4)));
```

而不是：

```
__def(Transaction2) __as_trans
( __with_id
    ( ID_TRANS
    , __sequential
        ( __asyn(Action1)
        , __asyn(Action2)
        , __asyn(Action3))
    , __asyn(Action4)));
```

即便对于剩下的 __with_id ，如果一个 观察者并不关注它，框架同样会知道这一点，为之生成的运行时代码里，将不会有与之有关的任何一个指令。比如：在之前的例子中，MyListener2 只关注 ID_TRANS ，而不关注 ID_SEQ ，那么当与 ID_SEQ 有关的任何事件，框架将不会通知给 MyListener2 ，内部生成的指令完全不会进行任何判断或尝试，而是从机器指令级别，就将其排除出去。

更进一步，由于 MyListener2 只关注 ID_TRANS 里的 onActionStarting ，因而，与此事件无关的任

何其它事件，比如 onActionDone 等等，也会在编译时，从机器指令的层面就消除了与之有关的任何指令。也就是说，你不会为之付出一个指令的代价。

综上所述，通过用户在定义一个 观察者时，明确的指明自己关心的 Action ID，框架将会保证，你无需为你不关注的事情付出任何一丁点代价。

由此，很容易产生一个结论：对于任何一个 Transaction，如果没有 观察者关注它，那么其中所有的 __with_id 都会被优化掉。因而上面的 Transaction1 无论从空间到性能，将完全等价于：

```
__def(Transaction1) __as_trans
( __procedure
    ( __asyn(Action1)
    , __asyn(Action2)
    , __finally(__rsp(Action3)))
, __asyn(Action4));
```

因而，如果你的系统需要在运行时，随时关闭和打开监控类需求。那么你只需要在开关关闭时，使用没有绑定任何 观察者的 Transaction，而在开关打开时，使用绑定了 观察者的 Transaction。从而，让你的系统在开关关闭时，不会为之付出哪怕一个指令的代价。

1.10 语言及框架

```
__def(sub_name [, __params(para [, ...])]) __as(actions...);

para := __action(action_name)  |
        __thread_id(thread_id) |
        __action_id(action_id) |
        __timer_id(timer_id);


action := __sync(synchronous_action)  |
          __asyn(asynchronous_action) |
          __sequential(action, action [, ...]) |
          __concurrent(action, action [, ...]) |
          __optional(pred, action) | __timer_prot(timer_id, action) |
          __sleep(timer_id) |
          __fork(thread_id, action) |
          __join(thread_id , ...)   |
          __procedure(action, __finally(action)) |
          __procedure(action, __recover(action)) |
          __with_id(action_id, action) |
          __apply(sub_name [, __with(param [, ...])]) |
          __wait(event_id) |
          __peek(event_id) |
```

(下页继续)

(续上页)

```
          __loop(...) |
          __time_guard(timer_id, action) |
          __safe(action) |
          __void(action);

timer_id  := BYTE
thread_id := BYTE
signal_id := BYTE
action_id := BYTE
```

1.11 用户接口改进

1.11.1 同步操作或谓词

- 对于 sync action 和 predicate，可以直接使用函数和 lambda，同时也允许使用 仿函数；

比如：

```
// normal function
auto SyncAction1(TransactionInfo const&) -> Status {
   return Result::SUCCESS;
}

// lambda sync action
auto SyncAction2 = [](TransactionInfo const&) -> Status {
   return Result::SUCCESS;
};

// lambda predicate
auto IsTrue = [](TransactionInfo const&) -> bool {
   return true;
};

__sequential
  ( __sync(SyncAction1)
  , __optional(IsTrue, __sync(SyncAction2))
  , ...
  );
```

- 对于 atom action，无需从任何接口类继承，而是直接定义相关函数即可。

比如，对于异步 Action，你只需要定义如下方法，却不需要继承任何接口：

```
struct AsyncAction1 {
   auto exec(TransactionInfo const&) -> Status;
   auto handleEvent(TransactionInfo const&, Event const&) -> Status;
   auto kill(TransactionInfo const&, Status cause) -> void;
};

__sequential
  ( __asyn(AsyncAction1)
  , __sync(SyncAction2)
  , ...
  );
```

这一方面免除了用户的负担，更重要的是，用户无需为不必要的虚函数付出任何额外代价。

1.11.2 __prot_procedure 的重新定义

在 1.x 时，过程分为两种: __procedure 和 __prot_procedure ，它们有完全一致的结构：前面时正常的 *Action* ，最后都有一个 __finally 。它们之间的区别在于，__procedure 无法从错误中恢复。而 __prot_procedure 可以。

但这其中一个麻烦是：由于两者最后都是 __finally ，导致在阅读代码时，无法迅速的知道这是一个 __procedure ，还是一个 __prot_procedure ，另外，用户在自由编写 DSL 代码时，根本无需指明这是一个 __procedure ，这种情况下，如何区分一个过程究竟是 __procedure 还是 __prot_procedure ？

而这两种语意的区别无非是是否可以通过 __finally 里的操作恢复之前的错误而已。所以，2.0 变更了定义它们的方式：不再有 __prot_procedure 。如果你需要从错误中恢复，最后不要使用 __finally ，而使用 __recover 。比如：

```
// 原来的 __procedure 语意
__procedure
( __sync(Action1)
, __asyn(Action2)
, __finally(__asyn(Action3)));

// 原来的 __prot_procedure 语意
__procedure
( __sync(Action1)
, __asyn(Action2)
, __recover(__asyn(Action3)));
```

1.11.3 更加自由的编写代码

在 1.x 时，如果操作是一个 __sequential 或 __procedure ，则必须明确指明，比如：

```
__transaction
( __sequential
    ( __asyn(Action1)
    , __asyn(Action2))
);

__transaction
( __procedure
   ( __sequential
       ( __asyn(Action1)
       , __asyn(Action2)),
   , __finally(
       __sequential
         ( __asyn(Action1)
         , __asyn(Action2)))
);
```

当然，这可以通过让 __transaction 默认就组合了一个 __procedure 或者某些部分默认就组合了 __sequential 来简化用户负担。但这样做至少有如下缺点：

首先，你只能默认组合一个。即你一旦默认组合了 __procedure ，则其必须有 __finally ，这对于只是 __sequential 的情况就很麻烦；反之亦然。

其次，引入了不必要的代价。因为一旦你组合了一个 __procedure 或者 __sequential ，无论你需不需它，你都在付出它们所占用的内存开销和运行时代价。比如下面这种最简单的情况，

```
__transaction
( __apply(Fragment));
```

如果 __transaction 背后默认组合了一个 __sequential ，则在付出不必要的代价。

2.0 对此进行了改进。用户可以自由的编写代码。而不用关心你是一个 __sequential ，还是一个 __procedure ，或干脆就是最简单的单一 action 。比如：

```
// simple case
__transaction
( __asyn(Action1));

// sequential
__transaction
( __asyn(Action1)
, __asyn(Action2));
```

(下页继续)

(续上页)

```
// procedure
__transaction
( __asyn(Action1)
, __finally(__asyn(Action2)));
```

框架可以自动感知你的代码结构。如果你有多个 action 则会自动按照 __sequential 的方式调度，如果最后是 __finally 或 __recover ，则会自动引入 __procedure 的调度，而如果你只是最简单的情况，则按照最简单的情况调度。

也就是说，用户可以在自由书写的同时，却不用担心付出任何不必要的代价。

而这样的能力无处不在，比如 **片段**：

```
// __procedure
__def(Fragment1) __as
( __asyn(Action1)
, __recover(__asyn(Action2)));

// __sequential
__def(Fragment2) __as
( __asyn(Action1)
, __asyn(Action2));

// just a simple one
__def(Fragment3) __as
( __asyn(Action1) );

// __procedure
// main action of __procedure is __sequential
__def(Fragment1) __as
( __asyn(Action1)
, __asyn(Action2)
, __recover(__asyn(Action3)));

// __procedure
// both main action & recover part are __sequential
__def(Fragment1) __as
( __asyn(Action1)
, __asyn(Action2)
, __recover
    ( __asyn(Action3)
    , __asyn(Action4)));
```

或者 __optional ，

```
// __optional with a __sequential
__optional
( __is_failed
, __asyn(Action1)
, __asyn(Action2));

// __optional with a __procedure,
// and the main action of the
// __procedure is a __sequential
__optional
( __is_failed
, __asyn(Action1)
, __asyn(Action2)
, __finally(__asyn(Action3)));
```

事实上，任何可以组合其它 *Action* 的关键字里，比如 __time_guard，__fork，__safe，__void，__case 等等，都拥有这样的能力。

多线程

在一个项目里，会存在多个 __transaction，其中一部分是单线程的，而另外一部分是多线程的。多线程调度器无论从内存，还是运行时资源消耗都要明显高于单线程。

在 1.x 时，为了在单线程场景不必付出多线程代价，需要由用户自己通过 __transaction 和 __mt_transaction 来区分。

2.0 则免除了用户的这种负担。如果你的代码中 __fork 了其它线程，__transaction 会自动选择资源消耗更大的多线程调度器，否则，将不需要付出这种不必要的代价。

```
// multi-thread
__transaction
( __fork(THREAD1, __asyn(Action1))
, __asyn(Action2))
, __finally(__on_fail(__asyn(Action3))))

// single-thread
__transaction
( __asyn(Action1)
, __asyn(Action2))
, __finally(__on_fail(__asyn(Action3))))
```

另外，框架不仅仅能够感知你是一个多线程，还是一个单线程。还能够感知你的线程数量。从而，会按照你实际的线程数量进行空间和性能优化。从而保证空间及性能的最优化。

inline __sequential

在我们编写一个复杂的 __transaction 时，无论是因为 **复用**目的，还是因为 **代码清晰**的目的，往往会抽取很多 **片段**。比如，本来有这样一个 __transaction：

```
// 多个Action，所以背后是一个__sequential
__transaction
( __asyn(Action1)
, __asyn(Action2)
, __asyn(Action3)
, __asyn(Action4)
, __asyn(Action5)
, __asyn(Action6)));
```

由于合理的原因，我们提取了两个片段：

```
// 多个Action，所以背后是一个__sequential
__transaction
( __asyn(Action1)
, __apply(Fragment1)
, __asyn(Action4)
, __apply(Fragment2));

// 多个Action，所以背后是一个__sequential
__def(Fragment1) __as
( __asyn(Action2)
, __asyn(Action3));

// 多个Action，所以背后是一个__sequential
__def(Fragment2) __as
( __asyn(Action5)
, __asyn(Action6));
```

这样，展开之后，会形成这样的结构：

```
// 多个Action，所以背后是一个__sequential
__transaction
( __sequential
    ( __asyn(Action1)
    , __sequential
        ( __asyn(Action2)
        , __asyn(Action3))
    , __asyn(Action4)
    , __sequential
        ( __asyn(Action5)
```

(下页继续)

(续上页)

```
        , __asyn(Action6))));
```

这就意味着，在（为了好的原因）提取片段的同时，你也在额外付出空间和性能代价。而这样的情况，基于现实项目的经验非常常见。

2.0 针对等价语意的 __sequential 进行了自动 inline 处理，即，如果 __sequential 嵌套 __sequential，内层的 __sequential 会被展开（inline）到外层的 __sequential 里。

对于上面的例子，经过 inline 处理之后，会自动恢复到与没有提取片段之前完全一样的结构上。

而在下面例子中的 4 个 transaction 完全等价，无论从语意，内存占用和性能开销，都完全一样。

```
__def(Fork2, __params(__action(ACTION1), __action(ACTION2))) __as
( __fork(1, __asyn(ACTION1))
, __fork(2, __asyn(ACTION2)));

////////////////////////////////////////////////////
__transaction
( __apply(Fork2, __with(AsyncAction1, AsyncAction4))
, __asyn(AsyncAction2)
, __join());

__transaction
( __fork(1, __asyn(AsyncAction1))
, __fork(2, __asyn(AsyncAction4))
, __asyn(AsyncAction2)
, __join());

__transaction
( __sequential
    ( __fork(1, __asyn(AsyncAction1))
    , __fork(2, __asyn(AsyncAction4))
    , __asyn(AsyncAction2)
    , __join()));

__transaction
( __sequential
   ( __sequential
       ( __fork(1, __asyn(AsyncAction1))
       , __fork(2, __asyn(AsyncAction4)))
       , __asyn(AsyncAction2))
   , __join()));
```

这样，就让程序员可以基于好的理由，自由的提取任何片段，而不用担心付出任何资源代价。

除了 __sequential 以外，__loop 里的 __sequential 也可以进行 inline，比如：

```
__def(Fragment) __as
( __asyn(Action2)
, __asyn(Action3));

__loop
( __asyn(Action1)
, __apply(Fragment)
, __asyn(Action4)
, __while(__is_failed));
```

与下面的形式，无论从语意，还是资源消耗，都完全相同：

```
__loop
( __asyn(Action1)
, __asyn(Action2)
, __asyn(Action3)
, __asyn(Action4)
, __while(__is_failed));
```

1.11.4 __with_id 自动感知及消除

参见观察者的定义及注册

1.12 性能对比

```
__procedure
( __sequential
  ( __wait(1)
  , __wait(2)
  , __wait(3)
  , __wait(4)
  , __wait(5)
  , __wait(6))
  , __wait(7))
  , __finally
    ( __sequential
      ( __wait(7)
      , __wait(8)
      , __wait(9)
      , __wait(1)))
) a;
```

1.x: 1225 ns

2.0: 116 ns

```
using ProcedureAction1 =
  __procedure(
    __wait(1),
    __finally(__asyn(AsyncAction2)));

using ProcedureAction2 =
  __procedure(
    __wait(2),
    __finally(__asyn(AsyncAction1)));

using Concurrent =
   __concurrent(
     ProcedureAction1,
     ProcedureAction2);
```

1.x: 2007 ns

2.0: 108 ns

如果我们再增加一个并发:

```
using ProcedureAction3 =
   __procedure(
      __wait(3),
      __finally(__asyn(AsyncAction4)));

using Concurrent2 =
  __concurrent(
    ProcedureAction1,
    ProcedureAction2,
    ProcedureAction3);
```

1.x: 3338 ns

2.0: 199 ns

如果我们将之前的顺序过程和并发过程混合在一起:

```
using Proc = __procedure
   ( __sequential
       ( __wait(1)
       , __wait(2)
       , __wait(3)
       , __wait(4)
```

(下页继续)

(续上页)

```
      , __wait(5)
      , __wait(6)
      , Concurrent2),
    __finally(__sequential(__wait(7), __wait(8), __wait(9)))
  );
```

1.x: `5402 ns`

2.0: `338 ns`

1.13 内存占用

如下所有测试均基于 `64-bit` 主机。

```
__procedure
( __sequential
  ( __wait(1)
  , __wait(2)
  , __wait(3)
  , __wait(4)
  , __wait(5)
  , __wait(6))
  , __finally
    ( __sequential
      ( __wait(7)
      , __wait(8)
      , __wait(9)))
) a;
```

1.x: *sizeof(a) = 432*

2.0: *sizeof(a) = 48*

如果我们增加两个 `__wait`:

```
__procedure
( __sequential
  ( __wait(1)
  , __wait(2)
  , __wait(3)
  , __wait(4)
  , __wait(5)
  , __wait(6))
  , __wait(7))
```

(下页继续)

(续上页)

```
  , __finally
    ( __sequential
      ( __wait(7)
      , __wait(8)
      , __wait(9)
      , __wait(1)))
) a;
```

1.x: `sizeof(a) = 496`

2.0: `sizeof(a) = 48`

```
using ProcedureAction1 =
  __procedure(
    __wait(1),
    __finally(__asyn(AsyncAction2)));

using ProcedureAction2 =
  __procedure(
    __wait(2),
    __finally(__asyn(AsyncAction1)));

using Concurrent = __concurrent(ProcedureAction1, ProcedureAction2);
```

1.x: `sizeof(Concurrent) = 480`

2.0: `sizeof(Concurrent) = 160`

如果我们再增加一个并发。

```
using ProcedureAction3 =
   __procedure(
      __wait(3),
      __finally(__asyn(AsyncAction4)));

using Concurrent2 = __concurrent(ProcedureAction1, ProcedureAction2,␣
↪ProcedureAction3);
```

1.x: `sizeof(Concurrent2) = 688`

2.0: `sizeof(Concurrent2) = 224`

如果我们将之前的顺序过程和并发过程混合在一起：

```
using Proc = __procedure
   ( __sequential
```

(下页继续)

(续上页)

```
      ( __wait(1)
      , __wait(2)
      , __wait(3)
      , __wait(4)
      , __wait(5)
      , __wait(6)
      , Concurrent2),
    __finally(__sequential(__wait(7), __wait(8), __wait(9)))
  );
```

1.x: `sizeof(Proc) = 1144`

2.0: `sizeof(Proc) = 288`