扩展 Erlang 的进程注册表

Ulf T. Wiger

Ericsson AB

ulf.wiger@ericsson.com

摘要

内建的进程注册表早已被实践证明是 Erlang 语言中一项极其有用的特性。它使得开发者能够很轻松地提供具名服务（named services）：用户无需知道服务进程的进程标识符（process identifier，PID）即可使用这些服务。

但目前的进程注册表也有其局限性：进程的名字必须是 atom（不支持有结构的数据），每个进程只能用一个名字注册，并且缺乏有效的搜索和遍历机制。

在 Ericsson 下属的 IMS Gateways 的产品开发中，我们经常需要维护一张映射表，以便根据各种属性找到负责处理调用的进程。我们从中发现了一个通用的模式（一种索引表），并由此开始开发一个扩展的进程注册表。

一开始这个想法并没有立即体现出价值，甚至看不出在实用中提供了多大的便利。但随着开发的进行，程序设计者们越来越多地使用这个扩展的进程注册表，并因此显著减少了代码量、提高了实现一致性。此外，扩展的进程注册表还提供了一种强大的调试机制，能够在数万个进程中进行有效的调试 。

本文介绍了这种扩展的进程注册表，并对其进行检讨，从而提出一种新的实现方式，使之更具一致性、效率更高、并且支持全局命名空间。

分类和主题描述 D.3.3 [**编程语言**]：语言构造和特性——抽象数据类型，模式，控制结构。

总括： 算法，语言

关键字： Erlang，进程注册表

1. 简介

IMS Gateways 是 Ericsson 下属的一个设计单位，负责开发 AXD 301 多服务 ATM 交换机[1]附属的一系列产品。也就是说，其中的很多开发者都曾参与过 AXD 301 的开发，很多概念也来自那里——尽管硬件架构是全新的，很多软件也是重新开发的。

AXD 301 在很大程度上是一块“不明区域”：从来没有人用 Erlang 开发过如此复杂的产品。起初进程是稀缺资源，但后来硬件变得越来越强大，这方面的限制也就迎刃而解了，市场转而要求更快开发出新的功能。从前我们只是专注开发一个复杂的产品，每 1～1.5 年作为一个发布周期；现在我们要同时开发 5～10 个类似的产品，而且发布周期更短。显然我们的编程风格也需要与时俱进，以适应这种新的要求。

我们发现自己的编程风格逐渐朝着“教科书式的 Erlang”方向转变：大量使用进程，尽量避免进行低层面优化。随着在产品中引入多核处理器，这种编程风格会愈发显著地体现出价值。

但问题也同样存在。我们发现在程序的某些部分，对自然并发模式的建模会产生 20～40 万个并发进程。虽然 Erlang 虚拟机处理这个数量级的进程不会有任何性能上的困难，但我们遇到的问题是：

当每个处理器上有 50 万个进程在运行时，如何对系统进行调试？

当数据被分散到几千个进程上、而不是呆在 ETS 表里，如何高效地处理它们？

大量的信息被隐藏在进程状态变量里、而不是内存数据库（ETS 或者 Mnesia）里，这会给调试造成怎样的影响？

当进程数增加到这种程度时，对内存的开销会有多大？

很多情况下，为了解决这些问题，程序设计者们会倾向于减少进程的数量，并把数据保存在 ETS 表中以便读取。不幸的是，这常常会导致令人费解的并发模型——而且情况会随着时间逐渐恶化，会让调试变得困难重重，给产品的演化造成阻碍[2]。但如果选择的一边是优雅、但可能在实地调试中爆炸（或者造成严重的困扰）的代码，另一边是复杂但行为可以预期的代码，对于不允许停机的系统来说后者无疑会胜出。当然了，如果优雅的模式也能扩展应用于极大的系统，那将是最理想的。

在讨论简化代码的办法时，我们发现自己在编程中做的很多事都可以归结为“找到正确的进程”，因此我们冒出了一个念头：创建一个通用的进程索引。本文就介绍了这样的索引机制，也就是“扩展的进程注册表”。和以往的商业产品开发一样，时间逼迫着我们在尚未彻底厘清概念之前就拿出了解决方案，因此代码难免有些臃肿和前后不一，但它已经在之前的编程模型基础上有了显著的改进。在文中我们还会对当前的实现进行检讨，并提出一个更加清晰的模型，它还可以用作全局进程注册表。

2. 现状

Erlang 的进程不仅提供了运行上下文和内存保护，而且还具有全局唯一的标识符（PID），错误处理机制（进程链接、捕捉退出信号、级联退出等）使它们在描述和构建系统时更加健壮。很多时候，或许你应该把 Erlang 的进程想象成“代理”。不过令人吃惊的是，在很多 Erlang 程序中只用到了匿名进程。在调试时，进程主要是通过它注册的名字（如果有的话）、初始化函数和当前函数来识别的，这往往并不足够。sys:get_status/1 函数能够提供更多信息，但只有当进程能够响应请求（responsive，即不被阻塞——译者注）、并且支持 OTP 系统消息时该函数才有效。在万不得已时，也可以生成进程的堆栈转储（stack dump），然后尝试解读其中的内容。

我们把“针对进程的操作”分为进程检视（process inspection）和进程选择（process selection）。

2.1 进程检视

正如 Cronqvist[3]所说，在进程元数据和跟踪工具的基础上可以建立起强大的调试工具。Cronqvist 介绍了调试大型系统时常用的一种方法：扫描所有进程以寻找问题、然后集中研究特定进程的详细信息。他还提到，把相关进程分组会给调试带来便利，而如果进程数量“很多”（600～800 个）则会难以把有错误的进程独立出来。Erlang 提供的进程属性（例如 heap_size、reductions 等）使开发者能够便利地了解资源使用情况。

程序员无法在标准的 process_info()元素集之外给进程增加任何信息元素，除非把数据保存在进程字典里。OTP 提供的库函数 sys:get_status(P)能够读取进程的内部状态，只要这个进程遵循 OTP 系统消息协议[4]。但迭代遍历所有进程、逐一调用 sys:get_status/1 会造成严重的后果：不仅因为这需要所有进程都支持 OTP 系统消息（只有试过才知道它是否支持），而且还因为这需要所有进程都做好了应答请求的准备。如果一个进程正在忙碌或者阻塞等待别的消息，我们的请求也会被阻塞甚至可能超时。在一个庞大的系统中，这种做法至少——在最好的情况下——会造成极大的开销。

2.2 进程选择

经常有这样的情况：一个进程需要给消息找到适当的接收者。

2.2.1 根据唯一的名字找到进程

在最简单的情况下，注册名字是为了将一个已知服务的代号公诸于众。Erlang/OTP 提供两种命名服务：内建的进程注册表，以及全局的命名服务器（也叫全局注册表）。全局命名服务器允许注册带结构的名字，而内建的注册表则不行。内建注册表的效率要比全局注册表高得多。

如果需要访问本地资源，但又不想注册唯一的名字，事情就变得更复杂了。下面这段不太容易理解的代码用于帮助当前进程找到 shell 求值器进程。第一个片段摘自 OTP 内核的 group.erl 模块，不过几乎同样的代码在别的地方（user_drv，user）也可以找到。请注意，这里的关键技巧在于偷窥另一个进程的进程字典：

代码 1：定位 IO 接口进程（group.erl）

```
%% Return the pid of user_drv and the shell process.

%% Note: We can't ask the group process for this info since it

%% may be busy waiting for data from the driver.
interfaces(Group) ->
case process_info(Group, dictionary) of
{dictionary,Dict} ->
get_pids(Dict, [], false);
_ ->
[]
end.
get_pids([Drv = {user_drv,_} | Rest], Found, _) ->
get_pids(Rest, [Drv | Found], true);
get_pids([Sh = {shell,_} | Rest], Found, Active) ->
```

```
get_pids(Rest, [Sh | Found], Active);
get_pids([_ | Rest], Found, Active) ->
get_pids(Rest, Found, Active);
get_pids([], Found, true) ->
Found;
get_pids([], _Found, false) ->
[].
```

这些函数在 OTP stdlib 应用的 shell.erl 模块中被用到。由于 case 语句嵌套太深，文中的代码格式可能不对。

代码 2：找到当前的 shell 求值器（shell.erl）

```
%% Find the pid of the current evaluator process.

whereis_evaluator() ->

      %% locate top group leader,

      %% always registered as user

      %% can be implemented by group (normally)

      %% or user (if oldshell or noshell)

case whereis(user) of
undefined -> undefined;
User ->
%% get user_drv pid from group,
%% or shell pid from user
case group:interfaces(User) of
[] -> % old- or noshell
case user:interfaces(User) of
[] -> undefined;
[{shell,Shell}] ->
whereis_evaluator(Shell)
end;
[{user_drv,UserDrv}] ->
%% get current group pid
%% from user_drv
case user_drv:interfaces(UserDrv) of
[] -> undefined;
[{current_group,Group}] ->
%% get shell pid from group
GrIfs =
group:interfaces(Group),
case lists:keysearch(
shell, 1, GrIfs) of
{value,{shell,Shell}} ->
whereis_evaluator(Shell);
false ->
undefined
end
end
end
```

```
end.
```

我们并不打算说这段代码写得不好，毕竟这是 Erlang 语言的创造者之一所写的代码。不过，我们确实鼓励读者思考更好的解决方案。

不妨来揣测一下，为什么这里没有用到名字注册。可能性最大的一种解释是，作者其实希望使用带结构的名字；或者注册非唯一的名字，然后根据当前在进程上调用 group_leader()函数的结果来进行查询匹配。不过 Erlang/OTP 并不支持这样的做法。

2.2.2 找出具有某个属性的所有进程

Erlang/OTP 并没有提供对进程分组的标准模式，但这种需求确实很常见。具体的做法有几种。

在 OTP Release Handler 中，下列代码会在软升级时执行，其目的是找出在重新装载一个模块时需要挂起的进程——这是在子进程的启动规约里说明的：

代码 3：找出需要挂起的进程（release_handler_1.erl）

```
suspend(Mod, Procs, Timeout) ->
lists:zf(
fun({_Sup, _Name, Pid, Mods}) ->
case lists:member(Mod, Mods) of
true ->
case catch sys_suspend(Pid, Timeout) of
ok -> {true, Pid};
_ ->
..., false
end;
false -> false
end
end, Procs).
```

Procs 变量是由下列函数产生的：

```
get_supervised_procs() ->
lists:foldl(
fun(Application, Procs) ->
case application_controller:get_master(Application) of
Pid when pid(Pid) ->
{Root, _AppMod} =
application_master:get_child(Pid),
case get_supervisor_module(Root) of
{ok, SupMod} ->
get_procs(supervisor:which_children(Root),
Root) ++
[{undefined, undefined,
Root, [SupMod]} | Procs];
{error, _} ->
error_logger:error_msg(…),
get_procs(
supervisor:which_children(Root), Root) ++ Procs
end;
```

```
_ -> Procs
end
end, [],
lists:map(
fun({Application, _Name, _Vsn}) ->
Application
end,
application:which_applications())).
```

release_handler 需要的信息实际上（在大多数时候）是静态定义的，并且保存在监控进程的局部状态里。一个显而易见的局限是：Release Handler 只能找到 OTP 监控树（supervision tree）的成员进程。其实对于并非 OTP 监控树成员的进程，suspend/code_change/resume 等操作也同样有效，只是它们没办法找到这些进程。

重申：这些代码并不糟糕，我们只是认为它有些过于费解，完全可以有改进的余地。此外请注意：这里给出的两个例子用了完全不同的策略来表现进程的属性，其结果是与进程相关的重要信息被散布得到处都是，尝试对系统进行调试的程序员必须掌握大量技巧才能找到这些信息。

2.2.3 进程字典

进程字典（process dictionary）实际上是一组特定的属性，通常是进程内建的 hash 字典，但它在 SASL 生成的崩溃报告中扮演着特殊的角色。所有 OTP 行为（behaviour）都在进程字典中保存关于父进程的数据，这些数据主要用于在进程死亡时输出崩溃报告。

代码 4：收集崩溃报告信息（proc_lib.erl）

```
crash_report(normal,_) -> ok;
crash_report(shutdown,_) -> ok;
crash_report(Reason,StartF) ->
 OwnReport = my_info(Reason,StartF),
 LinkReport = linked_info(self()),
 Rep = [OwnReport,LinkReport],
 error_logger:error_report(crash_report, Rep),
 Rep.

my_info(Reason,StartF) ->
 [{pid, self()},
 get_process_info(self(), registered_name),
 {error_info, Reason},
 {initial_call, StartF},
 get_ancestors(self()),
 get_process_info(self(), messages),
 get_process_info(self(), links),
 get_cleaned_dictionary(self()),
 get_process_info(self(), trap_exit),
 get_process_info(self(), status),
 get_process_info(self(), heap_size),
 get_process_info(self(), stack_size),
 get_process_info(self(), reductions)
 ].

get_ancestors(Pid) ->
 case get_dictionary(Pid,'$ancestors') of
```

```
 {'$ancestors',Ancestors} ->
 {ancestors,Ancestors};
 _ ->
 {ancestors,[]}
 end.

get_cleaned_dictionary(Pid) ->
 case get_process_info(Pid,dictionary) of
 {dictionary,Dict} -> {dictionary,clean_dict(Dict)};
 _ -> {dictionary,[]}
 end.

clean_dict([E|Dict]) when element(1,E) == '$ancestors' ->
clean_dict(Dict);
clean_dict([E|Dict]) when element(1,E) == '$initial_call' ->
clean_dict(Dict);
clean_dict([E|Dict]) ->
[E|clean_dict(Dict)];
clean_dict([]) ->
[].

get_dictionary(Pid,Tag) ->
 case get_process_info(Pid,dictionary) of
 {dictionary,Dict} ->
 case lists:keysearch(Tag,1,Dict) of
 {value,Value} -> Value;
 _ -> undefined
 end;
 _ ->
 undefined
 end.
```

进程字典提供了在进程中使用“全局变量”的方式。正如前面看到的，有时也用进程字典从其他进程中提取信息，但这样做相当麻烦，因为远端进程（remote process）只能把整个字典以一组{Key, Value}元组（tuple）的形式取出。

3. 详细需求

为了在现状之上有所改善，我们识别出扩展进程注册表的下列具体需求。

3.1 带结构的名字注册

在我们的应用中，最适合用于注册为名字以便处理进程的，莫过于 Call ID 了。取决于不同类型的呼叫，Call ID 的结构也会不同，但必定是唯一的。由于我们的系统需要每小时处理上百万次呼叫，所以不可能给每个呼叫分配一个唯一的 atom，因为这会很快超出 atom 表的最大容量。所以一开始我们根本没法注册呼叫处理进程，只得自己实现了一个分配表，把进程标识符与 Call ID 映射起来——说穿了就是一个专用的进程注册表！

3.2 一个进程注册多个名字

在产品演化的过程中，我们尝试过用各种不同的进程模型来处理呼叫，每种模型都有其长处和不足。在处理基于 H.248

的呼叫时，有必要用一个进程来代表“上下文”（context），可能还需要针对每个“终端”[1]（termination）定义一个进程。但在某些方面，终端也可以作为上下文进程的属性来处理。大多数时候，我们并不需要独立处理终端；但有时却又需要这样做。如果必须遵守“每个进程只有一个名字”的规定，那么我们要么注册终端进程、把自己锁定在“每个终端一个进程”的方向上，要么不注册终端、只是处理上下文（后者可能——取决于进程模型——把消息传递给终端进程）。

对于 H.248 呼叫处理来说，这里只有一个小问题：终端必须依赖于上下文存在。但我们还要处理别的协议，包括 SIP、H.323 和其他专有协议，而且经验告诉我们这些协议都会发生变化。

对于要向“负责处理终端的进程”发送信号的实体来说，这个进程是否还在处理别的资源它并不在意。但另一方面，要求只有一个进程负责与一个终端联系是有意义的。对进程注册表来说，“限制每个进程只能注册一定数量的名字”是一种人为的限制，但能够通过别名唯一地标识进程无疑是很有用的。

3.3 注册非唯一的属性

进程常常会依赖于别的资源，例如信号链路。如果信号链路被断开，通过该链路建立的所有呼叫都必须被删除。这并不难做到，只要公布一个“属性”来表示呼叫所使用的信号链路就行了。通常这是用 ETS 表来实现的：把链路 ID 和 Call ID 映射起来。

由于允许注册带结构的名字，我们就可以考虑把这些信息放在注册名字里，不过这最多也只能用于一个或者几个属性。

3.4 高效选择和检视相关进程

在批量处理呼叫和进程时，这些处理必须能够扩展到数万乃至数百万个进程。而且在对运行中的系统进行调试时，支持工程师或者软件设计者需要查看进程列表，同时又不能破坏系统或者给系统带来太大压力。为了降低软件复杂度，进程之间的依赖应该尽量可见并且规整。

4. 解决方案

下列局部解决方案已经可以解决上述的部分问题，下面我们就来介绍它们。

4.1 注册带结构的名字

要注册带结构的名字，最直接的办法当然是修改进程注册的 BIF，使其接受 atom 之外的名字。但这会对所有注册操作的性能造成显著影响，所以 Erlang 才没有这样做。

[1] “上下文”和“终端”都是 H.248 术语。一个上下文通常是指一次电话呼叫，终端则是传送数据的具体数据路径。一个上下文可以包含多个终端。

4.2 Atom 表垃圾收集

很多人希望看到垃圾收集的出现，但这个问题并不容易解决。Thomas Lindgren 提出了一种可行的途径[5]，但并没有真正解决注册带结构名字的问题。

4.3 进程 ETS 表

有一个提议正在被众人讨论：以类似于 ETS 表的形式来表示进程元数据，从而实现 select()形式的搜索操作。这听起来是个不错的主意，但实现并不容易。这个提议的要点在于，它并不给通常的进程处理操作带来明显的性能负担。

4.4 导出进程字典中的元素

要想更方便地用进程字典“公布”元数据，有几件事情可以做。一种——不完备的——解决方案是允许进程检视其他进程字典里的各个对象，实现起来也不会特别困难。另一种方案是增加某些“公布”特定属性的途径、将它们放入全局索引，不过这实现起来的难度要高得多。而且有一种观点认为：如果进程字典太过易用，会导致初学者误入歧途、养成一种 put/get 的编程习惯——这在 Erlang 中通常被认为是有害的。

4.5 分离注册表

这是我们提倡的办法，一定程度上是因为它相对容易实现，也不需要修改运行时系统。一开始我们计划实现一个简单的注册表，允许放入唯一的、带结构的名字和非唯一的属性。但正如人们第一次实现一个想法时经常遇到的，在使用这个简单注册表的过程中我们又有了新的点子，于是又给它加上了新的功能（计数器、通过代理注册、合计计数器等）。

下文将介绍这个注册表的第一版实现，并提出一个在我们看来更加规整的新解决方案——比第一版更快，提供更多的功能（同时支持全局和本地作用域）。

5. 扩展的进程注册表（proc）

我们创建了一个名叫“proc”的模块来实现扩展进程注册表（在我们的产品中叫 sysProc，因为我们一直用前缀名来管理命名空间），在 Jungerl 有一个开源的版本。在实现中我们用了一个 gen_server 和一组 ordered_set ETS 表。由于这只是一个常规的 Erlang 实现，并没有集成到虚拟机里，因此我们还引入了一些额外的函数以便优化之用。

最基本的函数包括：

- reg(Name) – 注册一个唯一的名字，该名字可以是任何 term。
- unreg(Name) – 取消注册一个名字。

- where(Name) -> pid() | undefined – 查询一个名字。
- send(Name, Msg) – 把 Msg 发送给一个注册进程。
- add_property(Property) – 公布一个非唯一的属性。
- del_property(Property) – 取消公布一个属性。

除了这些基本的函数之外，还有一些用于读取信息的函数：

- fold_names(Fun, Acc, Patterns) -> NewAcc
- fold_properties(Fun, Acc, Patterns) -> NewAcc
- select_names(Patterns) -> [{Name, Pid}]
- select_properties(Patterns) -> [{Property, Pid}
- first_name()/next_name(Prev)/last_name()
- first_property()/...
- info(Pid, Item)，其中 Item = properties | names | ...

在这些函数中，Patterns 都是用于匹配的模式，与 ets:select_count(Tab, Pattern)中的 Pattern 参数类似——在这个函数里，Tab 是一个 ordered_set 表，其中的元素是{Name, Pid}对象。

6. 使用范例

6.1 找出具有相同属性的进程

回到 2.2.2 介绍的 release_handler 代码示例：如果改为使用扩展的进程注册表，对应的实现大致如下：

代码 5：代码更换时挂起进程

```
suspend(Mod, T) ->
 proc:fold_properties(
```

```
      fun({_, Pid}, Acc) ->
            case catch sys:suspend(Pid, T) of
                  ok -> [Pid | Acc];
                  _ -> Acc
      end
 end, [], {sasl, suspend_for, Mod}).
```

这个版本不仅速度显著提高，而且适用于任何 Erlang 进程，不管它们是否隶属于 OTP 监控树。

仔细观察这段代码，就能看出其中包含着一个简单的公布/订阅（publish/subscribe）模式。下面需要的就是“订阅”的函数：

代码 6： 注册代码更换时调用的模块

```
suspend_for(Mod) ->
proc:add_property({sasl, suspend_for, Mod}).
```

现在我们已经大大改进了 release_handler，允许需要被挂起的进程在运行过程中增加模块。

而且与当前这个复杂难懂、需要充分了解 OTP 设计原则和 release handler 才能完全理解的实现相比，我们的解决方案建立在一个常见的模式之上，因此看来清爽得多。设计者能够轻松地找到进程的这个——从前是被隐藏起来的——属性，例如通过调用 proc:info(Pid, properties)函数；而且也能够从 Erlang shell 命令行快速获得这些进程，例如借助 proc:pids({sasl, suspend_for, M})函数。

（尽管上述代码看来有些刻意而为的痕迹，但它复用了 ETS 的 select 模式，因此程序员可以很快习惯——特别是当进程注册表成为公开所有进程元数据的主要门户时。）

继续批评当前的解决方案：我们提出的方案对系统中进程的数量相当不敏感。我们的进程注册表使用了一个 order_set ETS 表，因此只要我们搜索的属性至少有一部分是已知的，取出与之关联的子集合就会相当快速，而不必遍历整个进程监控树。

6.2 找出特定服务的特定实例

2.2.1 节给出的例子——找到当前的 shell 求值器——用扩展的进程注册表也可以大大简化。实际上，由于这段代码假设只有一个当前求值器存在，我们完全可以用一个唯一的名字——例如{shell, current_evaluator}——来表示它。当然也可以使用 6.1 节介绍的公布/订阅模式。

代码 7： 注册当前求值器

```
whereis_evaluator() ->
proc:where({?MODULE, current_evaluator}).
```

7. 急速下滑

扩展进程注册表的第一个原型完全模仿 Erlang 内建的进程注册表，只是保存{Key, Value}二元组，因此在处理唯一进程

名时多少显得有些牵强，并且可以用于管理共享属性的好处也没有明显地体现出来。于是我们去掉了其中的 Value 部分，只留下了 Key 部分。

但很快我们就有了在进程注册表中保存计数器的需求，尽管这看上去很像是一种边缘的甚至是错误的用法，但它也有一些合理之处：所有处理呼叫的进程都需要维护计数器，因此经常需要遍历所有的计数器——就像遍历其他属性一样。不幸的是，虽然计数器大多数时候表现得跟其他属性一样，但它必须有一个值。有些计数器是实时计算的，而统计计数器——它负责维护所有同名计数器的总和——则是逐次累加的。为了维护统计计数器，有些进程就需要监视计数器的所有者，如果监视对象死亡就要更新对应的统计计数器。由于进程注册表已经有一个进程在执行这样的监视工作，所以把统计计数器放在这里就显得完全合理了。但这样做了以后，统计计数器的删除必须显式进行，而且在查询属性和/或计数器时也不会被包含在内。

尽管计数器弄得这个“proc”实现快散架了，但我们还是要强调：计数器是极其有用的。内建的进程属性中也有计数器（reductions）和——从更普遍的意义上来说——值属性（heap_size、fullsweep_after 等等）。实际上，只有允许值属性的存在，我们这个模型才算完整。

另一个后来加入的需求是通过代理来注册属性。这需要实现一个访问控制列表，以便控制谁可以针对指定进程添加或删除属性。这是为了避免重写大量代码而作出的妥协，但却让属性注册的成本变得不必要地高昂：如果没有通过代理注册的功能，进程就可以直接把自己的属性放进中央的 ETS 表，不会有任何竞态（race condition）出现。这个问题与 BIF register/2 面临的问题类似：由于可以注册另一个进程，因此很容易写出并非时序安全（timing-safe）的代码来。

我们认为，扩展进程注册表只应该允许进程自己注册名字和共享属性。唯一名字的注册必须顺序进行，但非唯一的属性就无所谓了。稍后我们将看到，通过加入一个 OTP 系统消息，我们就能实现“通过代理注册”的功能。

8. 改进后的模型与实现（gproc）

扩展进程注册表第一个实现版本帮助我们理解了究竟需要哪些功能。现在我们就来介绍第二个实现版本：gproc[6]（其中“g”表示加入对全局作用域的支持）。

我们最主要的改进目标是对进程元数据进行索引，从而高效地遍历具有类似属性的进程。根据下列几组区分标准，我们得以识别出不同类型的元数据（属性）：

- 是唯一的名字还是不唯一的属性
- 属性来自手工插入还是自动生成
- 属性作用域是本地还是全局

8.1 唯一的名字和不唯一的属性

我们对“名字”和“属性”的区分标准是：前者是唯一的，后者不是。出于一致性考虑，两者都包含键和值两部分。对于计数器，我们做了特殊的处理，因为它们特别常用，而且对系统性能至关重要。由于我们已经知道如何用最小的开销

来维护统计计数器，因此也允许这一特殊类型的计数器存在。

8.2 手工创建的属性和自动创建的属性

模拟器给每个进程维护了一组属性，由于可以确保所有进程都有这些属性，因此给它们加上索引似乎没有太多好处，但会有助于执行诸如“列出所有运行在高优先级上的进程”或者“列出系统中所有的 gen_server”这样的查询操作（后者需要查看另一个隐藏的进程属性，这对于理解程序行为至关重要）。

一些自动创建的属性变化太过频繁，以致于无法进行任何形式的索引，例如“reductions”和“current_function”等。如果能在 ets:select()中调用 BIF process_info()，就可以让这些属性看起来好像也在索引之中。

统计计数器也是自动创建属性的一个例子。也许应该有一个通用的模式来安全高效地处理用户定义的自动创建属性，但至少我们还没发现这个模式。

8.3 全局属性和本地属性

此前我们只是在本地范围内使用这个进程注册表，但应该注意到：OTP 提供的全局命名服务器也允许注册带结构的名字，并且也允许一个进程注册多个名字。

我们希望允许所有这些方式的注册，不论在全局还是本地。

8.4 操作许可

我们限制进程只能添加、删除和修改属于自己的属性——绝不允许操作其他进程的属性。这不仅能提高效率，而且从源头上根除了出现竞态的可能性。这与当前的“proc”实现有所不同：后者允许通过代理注册，但我们也可以通过扩展 sys.erl 模块来实现同样的功能。

8.5 一致的数据表现形式

扩展进程注册表的一个主要优点是支持 ets:select()和 ets:fold()形式的搜索操作。可惜属性、名字和计数器的表现形式并不一致，从而导致出现几组不同的搜索函数和 guard 语句等问题。

我们建议大幅增强 Erlang 的模式匹配功能，采用下列表现形式：

```
{Key, Pid, Value}
Key :: {Type, Context, Name}
Type :: n | p | c | a
Context :: g | l
```

类型（Type）和上下文（Context）的定义都采用了简写，这是为了获得更紧凑的查询表达式和列表，而不是为了任何性能上的原因，因为 atom 无论在本地还是在 Erlang 节点之间都是按引用传递的。

可选的类型包括：

```
n = name
p = property
c = counter
a = aggregated counter
```

可选的上下文包括：

```
g = global
l = local
```

我们希望把所有值都保存在同一张表中，以便高效地根据不同条件查询所有对象。但为了区分唯一和不唯一的键，我们需要做一点让步：给键加上一点额外的信息，以便不同的进程能注册同样的键。最简单的做法就是加上 Pid：

代码 8： 唯一对象的存储结构

```
{{Key, Pid}, Pid, Value}
```

为了保持一致性，我们对唯一的键也做了类似的包装，不过加上的额外信息是一个常量（而非 Pid）。我们选择了代表类型的 atom：

代码 9： 非唯一对象的存储结构

```
{{Key, Type}, Pid, Value}
```

我们希望把这层包装隐藏在搜索函数内部，不过这只是简单地重写了选择模式而已。由于它需要在运行时执行，因此会给来自 Erlang shell 的所有查询增加启动时的性能开销。

8.6 反向映射

对于插入注册表的每个对象，我们还同时插入一个反向映射对象{Pid, Key}，这样就可以——譬如在进程死亡的时候——方便地找出属于某个进程的所有对象。只要足够小心，一次写入多个对象也不会损害线程安全性：反向映射对象是唯一的，并且必然与对应的属性对象存在于同一个 ETS 表中，所以进程只要调用下列函数就可以原子地注册一个属性：

代码 10： 插入属性对象和反向映射

```
ets:insert_new(
  ?TAB, [{{Key,self()}, self(), Value},
  {{self(), Key}}]).
```

如果属性对象已经存在，这个函数就会返回 false。“proc”库会在进程重复尝试注册属性时抛出异常，我们觉得没有理由改变这一点。

8.7 处理监视器

如果注册操作经过了服务器，后者就可以针对每个对象启动一个监视器，不过实际上针对每个进程只需要一个监视器。启动多个监视器的成本并不高；但要把来自每个进程的“DOWN”消息都处理很多遍，成本就会相当高了。

我们可以在表里插入一个标记，以确保每个进程只有一个监视器。我们用 ets:insert_new()函数检查这个标记是否存在，如果标记尚不存在就开启监视器，并插入标记。

代码 11： 在需要时开启监视器（服务器端）

```
case ets:insert_new(?TAB, {Pid}) of
false -> ok;
true -> erlang:monitor(process, Pid)
end.
```

当客户端注册属性而又不通知服务器时，就要自己进行类似的检查：

代码 12： 在需要时开启监视器（客户端）

```
case ets:insert_new(?TAB, {self()}) of
 false -> ok;
 true -> cast(monitor_me)
end.
```

一个明显的缺陷是没有取消监视的办法——只要开始监视一个进程，就会一直监视下去。要解决这个问题，可以把标记变成一个计数器，当计数器到达 0 时就清除监视器。不过，如果进程注册表与 Erlang/OTP 完全整合的话，根本就不需要一个显式的监视器。

8.8 统计计数器

统计计数器和普通的计数器很相似，但其计数值是在同名（并且位于同一作用域中）的普通计数器更新时自动更新的：

代码 13： 更新统计计数器

```
update_counter({c,g,_} = K, Incr) ->
leader_call({update_counter,K,self(),Incr});
update_counter({c,l,Name}, Incr) ->
 Prev = ets:update_counter(
 ?TAB, {{c,l,Name},self()}, Incr),
 catch ets:update_counter(
 ?TAB, {{a,l,Name},a}, Incr),
 Prev.
```

为了让 update_counter()的开销保持在可预期的范围内，我们把统计计数器设置为唯一的：每次 update_counter()调用最多只更新一个统计计数器。

8.9 QLC

我们还加上了一个 QLC 生成器，以便支持查询列表解释推断（Query List Comprehensions，QLC）。除了能够支持非常复杂的查询之外，QLC 还使所有查询都可以重入（reentrant），无需任何持续传送循环（continuation-passing loop）。

9. 示范

我们以 OTP R11B-5 为基础，把 gproc 变成了核心应用的一部分，并构造了几个模块来自动注册 grpoc 需要的一些属性。

下面我们就将演示如何简单地显示系统中的重要属性——节点是用命令“erl --boot start_sasl”来启动的。

代码 14：查询监控标志（使用 gproc）

```
=PROGRESS REPORT==== 5-Jul-2007…
 application: sasl
 started_at: nonode@nohost
Eshell V5.5.5 (abort with ^G)
 1> Q1 = qlc:q([{P,Fs} ||
 {{p,l,supflags},P,Fs} <-
 gproc:table(props)]).
{qlc_handle,{qlc_lc,…}}
2> qlc:eval(Q1).
[{<0.10.0>,{one_for_all,0,1}},
 {<0.27.0>,{one_for_one,4,3600}},
 {<0.32.0>,{one_for_one,0,1}},
 {<0.33.0>,{one_for_one,4,3600}}]
```

这样我们就得到了一份简要的列表，其中包含了系统中的监控重启策略。此外我们还能得到所有监控器进程，不过这通过对行为（behaviour）的搜索也可以得到。

代码 15：查找所有具有行为的进程（使用 gproc）

```
3> Q2 = qlc:q([P ||
{{p,l,behaviour},P,supervisor} <-
gproc:table(props)]).
{qlc_handle,…}
4> qlc:eval(Q2).
[<0.10.0>,<0.27.0>,<0.32.0>,<0.33.0>]
```

回顾 5.1 节的例子，现在我们注册的信息**几乎**已经提供了我们想要的答案：

代码 16：找到需要挂起的进程（使用 gproc）

```
8> rr(code:lib_dir(stdlib) ++
"/src/supervisor.erl").
[child,state]
12> Q3 = qlc:q([{C#child.pid,S,
C#child.modules} ||
{{p,l,{childspec,_}},S,C} <-
gproc:table(props)]).
{qlc_handle,…}
13> qlc:eval(Q3).
[{<0.34.0>,<0.33.0>,dynamic},
{<0.20.0>,<0.10.0>,[code]},
{<0.19.0>,<0.10.0>,[file,file_server,
```

```
file_io_server,
prim_file]},
 {<0.18.0>,<0.10.0>,[global_group]},
 {<0.12.0>,<0.10.0>,[global]},
 {<0.9.0>,<0.10.0>,[gproc]},
 {<0.16.0>,<0.10.0>,[inet_db]},
 {<0.26.0>,<0.10.0>,[kernel_config]},
 {<0.27.0>,<0.10.0>,[kernel]},
 {undefined,<0.10.0>,[erl_distribution]},
 {<0.35.0>,<0.33.0>,[overload]},
 {<0.36.0>,<0.32.0>,[]},
 {<0.11.0>,<0.10.0>,[rpc]},
 {<0.33.0>,<0.32.0>,[sasl]},
 {<0.21.0>,<0.10.0>,[user_sup]}]
```

可以看到，release_handler 需要的信息作为子进程启动规约的一部分已经都齐备了。

我们在 5.1 节设想的解决方案在实践中并不容易实现，因为监控模块并不是在子进程中执行代码的，所以要在它的上下文中插入属性并不容易。更严重的问题是监控器无法知道一个进程为何被重启或是重启了多少次，所以即便问题变得严重，它也没办法采取什么不同的措施。使用 gproc，监控器就可以把诸如此类的信息保存在自己的上下文里，从而根据这些信息判断应该做什么。

9.1 通过代理注册

我们还修改了 sys 模块，从而可以要求一个进程注册属性：

代码 17：通过代理注册（使用 gproc+sys）

```
1> gproc:info(whereis(gproc),gproc).
{gproc,[{{p,l,behaviour}, gen_leader}]}
2> sys:reg(gproc, {p,l,test}, 17).
true
3> gproc:info(whereis(gproc),gproc).
{gproc,[{{p,l,behaviour},gen_leader},
{{p,l,test},17}]}
```

sys:handle_system_msg/6 是处理系统消息的推荐方法，它会调用 Mod:system_reg(Misc, Key, Value)函数（如果 Mod 导出了这个函数的话），否则就尝试直接注册。通过回调函数可以有选择地给某些行为禁用通过代理注册。

9.2 性能

评估收集和索引元数据的成本开销是很有必要的。尽管把所有进程和应用属性都索引起来备用确实会很方便，但所需的成本开销很可能令人无法承受。出于示范之用，我们用 gproc 公布了行为信息、子进程规约、监控标志等，现在应该来看看我们的投入能得到多少回报了。

我们把使用 gproc 之后的应用程序与轻量级替代方案（即采用普通的进程注册表，或者在 ETS 表中保存简单对象）以及使用 sysProc（gproc 的前辈）的版本进行性能对比。

9.2.1 属性注册

轻量级替代方案：把映射关系和反向映射保存在 ETS 表中。我们保存了 100 个类似这样的二元组。

Ets	gproc	sysProc
1	3.0	5.4

9.2.2 本地名字注册

轻量级替代方案：register()/unregister() BIFs。由于这两个 BIF 不允许注册多个名字，所以我们先注册一个名字，然后再取消注册，重复 100 次。

register BIFs	gproc	sysProc
1	68.8	74.7

这里出现的巨大差异主要是因为 gproc 和 sysProc 用了一个 gen_server 来防止出现竞态，而 BIFs 在 VM 内的执行是原子的（在非 SMP 版本上甚至不需要互斥体）。如果以 BIF 的形式引入注册 Erlang term（而不仅仅是 atom）的功能，应该会使性能有一定降低，但也不会降低到 gproc 和 sysProc 的程度。

应该注意到，register()/unregister()的开销确实非常低。在大部分主流机器上，即便是使用 gproc 和 sysProc，本地名字注册所需的时间也在 100μ 秒以内。

9.2.3 全局名字注册

轻量级替代方案：global:register_name()。我们在同一台机器上运行 4 个节点，然后执行 100 次注册/取消注册操作，因为如果重复注册名字，“global”模块会发出一个警告。

Global	gproc	sysProc
1	0.16	N/A

显然“轻量级”替代方案并不那么轻量级。使用 gen_leader 的全局注册甚至比“global”模块的投票机制更快。sysProc 没有提供全局注册的功能。

10. 未来的方向

在 IMS Gateways，gproc 原型是否能取代当前的扩展进程注册表仍然需要深入考察。此外值得关注的是 gproc 的本地功能是否能在运行时系统实现、从而获得更高的性能。

gproc 原型是建立在 gen_leader 行为基础上的。gen_leader 最初由 Arts 和 Wiger 开发[7]，后来由 Arts 和 Svensson 重写并验证[8]。gen_leander 简化了同时管理全局和本地作用域的服务器的实现。

在开发 gproc 模块的过程中，我们也遇到了 gen_leader 的一个问题：即便只有一个已知的候选 leader，gen_leader 还是会进入一个选举循环，并等待其他候选 leader 的响应。这场选举将依赖于永远不会到来的响应。还好修复这个问题并不费事：如果只有一个候选 leader，就直接将其选为 leader，跳过后续的协商过程。

更严重的问题是 gen_leader 需要提前知道所有候选节点的信息，但在我们与 OTP 整合的原型中，我们希望尽早启动进程注册表——甚至在 Erlang distribution 之前启动。为此我们对 gen_leader 做了修改，使之以“localonly”模式启动，然后等待用户触发选举模式。这个对 gen_leader 的扩展尚未得到验证，不过看起来它完全满足了我们这个原型的需要。

还可能有更严重的问题：gen_leader 不支持网络的动态重配置，也无法解除两个 leader 之间的冲突状态（这可能是在运行时添加节点的最大阻碍）。给 gen_leader 加上这方面的支持是未来的一个研究方向。我们希望这个原型能激起人们解决此问题的兴趣。

致谢

我要感谢 John Hughes 和 Thomas Arts，他们借助令人惊叹的 QuickCheck 工具无情地找出了第一个原型中的一些致命缺陷。感谢 Thomas Arts 和 Hans Svensson 在 gen_leader 上的精彩工作。感谢 Kurt Jonsson 和 Mats Cronqvist 大量卓有成效的设计讨论。感谢 Bo Fröderberg 和 Mats Andersson 从应用开发的角度为我们详细解释所有的需求。

参考资料

[1] Ulf Wiger, “Four-fold Increase in Productivity and Quality”, FemSYS, Munich, Germany, 2001

http://www.erlang.se/publications/Ulf_Wiger.ppt

[2] Ulf Wiger, “Structured Network Programming”, Erlang User Conference, Stockholm, Sweden, 2005

http://www.erlang.se/euc/05/1500Wiger.ppt

[3] Mats Cronqvist, “Troubleshooting a Large Erlang System”, ACM SIGPLAN Erlang Workshop, Pittsburgh, USA, 2004

http://doi.acm.org/10.1145/1022471.1022474

[4] Erlang/OTP Design Principles,

http://www.erlang.org/doc/design_principles/part_frame.html

[5] Thomas Lindgren, “Atom Garbage Collection”, ACM SIGPLAN Erlang Workshop, Tallinn, Estonia, 2005

http://doi.acm.org/10.1145/1088361.1088369

[6] Ulf Wiger, gproc Source Code Repository (Subversion)

http://svn.ulf.wiger.net/gproc

[7] Arts, Claessen, Svensson, “Semi-formal Development of a Fault-Tolerant Leader Election Protocol in Erlang”

Fourth International Workshop on Formal Approaches to Testing Software ,volume 3395 of *LNCS*, pages 140-154

Linz, Austria, 2004

http://www.ituniv.se/~arts/papers/fates04.pdf

[8] Svensson, Arts, “A New Leader Election Implementation”

ACM SIGPLAN Erlang Workshop, Tallinn, Estonia, 2005

http://doi.acm.org/10.1145/1088361.1088368