

# RabbitMQ 源码学习

懒惰的蚂蚁

2016 年 10 月 7 日



# 目录

<b>1</b>	<b>前言</b>	<b>1</b>
<b>2</b>	<b>RabbitMQ Server 启动机制</b>	<b>3</b>
2.1	源码目录结构 . . . . .	3
2.2	资源文件 . . . . .	4
2.3	启动流程 . . . . .	5
<b>3</b>	<b>网络层分析</b>	<b>9</b>
3.1	TODO... . . . .	9
<b>4</b>	<b>存储分析</b>	<b>11</b>
4.1	TODO... . . . .	11
<b>5</b>	<b>AMQP 协议分析</b>	<b>13</b>
5.1	AMQP 协议 . . . . .	13
5.2	Broker . . . . .	13
5.3	Vhost . . . . .	13
5.4	Exchange . . . . .	13
5.5	Queue . . . . .	13
5.6	Message . . . . .	13
<b>6</b>	<b>高可用和集群分析</b>	<b>15</b>
6.1	TODO... . . . .	15



# Chapter 1

## 前言

RabbitMQ 实现了高级消息队列协议，是比较常见的消息中间件。平时工作中，RabbitMQ 也算是常用的组件了，也因为 RMQ 出现过多事件或事故。有些是因为使用姿势不对，有些因为 RMQ 本身的问题，但多次出现问题大多数人摸不着头脑，很是头疼；在网上搜了下发现关于 RMQ 的源码分析只有寥寥几篇，还都是好几年前的，遂下定决心学习下 RMQ 的源码，有很大一部分决心来自于陈帅大的提议。

学习计划主要分为以下几个部分：第二章学习 RMQ 的启动机制；第三章学习 RMQ 的网络层模块；第四章学习 RMQ 的存储管理；第五章学习 RMQ 的 AMQP 协议实现，主要包括 broker、vhost、exchang、queue、messag 等；第六章学习下 RMQ 的高可用和集群实现。

个人能力有限，文中错误之处还请大家斧正；也欢迎大家一起来分析，相互学习进步。

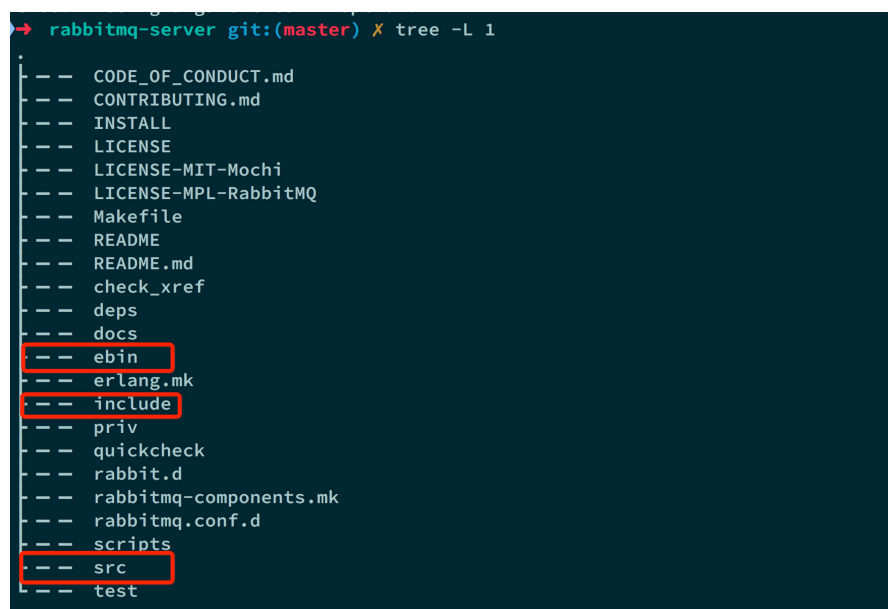


# Chapter 2

## RabbitMQ Server 启动机制

### 2.1 源码目录结构

RabbitMQ Server 本质上是一个 OTP Application, 他有着典型的 OTP app 目录结构, 如图 2.1:



```
→ rabbitmq-server git:(master) X tree -L 1
|-- CODE_OF_CONDUCT.md
|-- CONTRIBUTING.md
|-- INSTALL
|-- LICENSE
|-- LICENSE-MIT-Mochi
|-- LICENSE-MPL-RabbitMQ
|-- Makefile
|-- README
|-- README.md
|-- check_xref
|-- deps
|-- docs
|-- ebin
|-- erlang.mk
|-- include
|-- priv
|-- quickcheck
|-- rabbit.d
|-- rabbitmq-components.mk
|-- rabbitmq.conf.d
|-- scripts
|-- src
|-- test
```

图 2.1: 目录结构

在这个一级目录中红色方框中的目录是一个典型的 OTP app 目录。src

主要是源码目录；ebin 主要是 erlang 编译出的字节码文件，该目录还存放有一个重要的应用资源文件 (AppName.app)；include 存放一些头文件，主要是一些 record 和宏定义。

## 2.2 资源文件

因为资源文件 ebin/rabbit.app 定义了一个应用的基本资源信息，所以我们首先来看下这个文件的内容，如图 2.2:

```
%% -*- erlang -*-
1 {application, rabbit,
2   [{description, "RabbitMQ"},
3    {id, "RabbitMQ"},
4    {vsn, "0.0.0"},
5    {modules, ['background_gc', 'delegate', 'delegate_sup', 'dtree', 'file_ha
6    {registered, [rabbit_amqueue_sup,
7                  rabbit_log,
8                  rabbit_node_monitor,
9                  rabbit_router,
10                 rabbit_sup,
11                 rabbit_direct_client_sup]},
12   %% FIXME: Remove goldrush, once rabbit_plugins.erl knows how to ignore
13   %% indirect dependencies of rabbit.
14   {applications, [kernel, stdlib, sasl, mnesia, goldrush, lager, rabbit
15   %% we also depend on crypto, public_key and ssl but they shouldn't be
16   %% in here as we don't actually want to start it
17   {mod, {rabbit, []}},
18   {env, [{tcp_listeners, [5672]}],
19 +----- 89 li neas: {num_tcp_acceptors, 10},-----
```

图 2.2: rabbit.app

资源文件其实就是 erlang 中的一个 tuple 数据类型；出了 1-5 行主要定义了应用的名字、版本、描述等基本信息外；值得说明的是第五行 modules, ... 罗列了 rabbitmq 中的所有 module 模块。第 6 行 registered, ... 罗列了 rabbit 这个应用所启动的所有进程，这里有一个 Erlang/TOTP 很重要的设计模式 Supervision Principles，基于该特性使得 erlang 可以很轻易的监测



到进程的 crash，并很轻易的重新启动新进程，从而丢弃了大多数语言的防御式编程模式，提出了“就让它崩溃吧”（let it crash），Orz...。第 10 行的 `rabbit_sup` 就是整个 rabbit supervision tree 的根；第 11 行定义了 rabbit 应用所依赖的其他应用。第 12 行定义了 application behavior 的回调模块，这个模块的 `start(Type, StartArgs)` 就是整个应用程序的入口，相当于 `main`。第 18 行 `env,...` 定义和应用的环境变量，rabbit 的各个配置都可以在这里设置，鉴于篇幅问题没有展开剩下的环境变量；当然这些变量也可以通过命令行或者配置文件 `rabbitmq.config` 进行修改。

## 2.3 启动流程

根据前面的分析,我们知道 application behavior 的程序入口是 `./src/rabbit.erl` 中 `start/2` 函数，如图 2.3

```
600 start(normal, []) ->
1   case erts_version_check() of
2   |   ok ->
3   |       rabbit_log:info("~n Starting RabbitMQ ~s on Erlang ~s
4   |                       [rabbit_misc:version(), rabbit_misc:o
5   |                       ?COPYRIGHT MESSAGE, ?INFORMATION_MES
6   |       {ok, SupPid} = rabbit_sup:start_link(),
7   |       true = register(rabbit, self()),
8   |       print_banner(),
9   |       log_banner(),
10  |       warn_if_kernel_config_dubious(),
11  |       warn_if_disc_io_options_dubious(),
12  |       rabbit_boot_steps:run_boot_steps(),
13  |       {ok, SupPid};
14  |   Error ->
15  |       Error
16  end.
17
```

图 2.3: 启动程序入口

该函数其实做了主要住了两件事：第一是创建一个 supervision tree，这里的 rabbit\_sup 实现了 supervisor behavior。第二个是重中之重，rabbitmq 启动方式是通过 boot\_step 的方式启动，在每个 app 中都定义了自己的 boot\_step；这个的 boot\_step 定义个待启动模块的方式，前驱依赖和后继依赖，理解为一个 DAG 图的一条边就好了，下面是一个例子：2.4

```
-rabbit_boot_step({rabbit_alarm,  
  [{description, "alarm handler"},  
   {mfa, {rabbit_alarm, start, []}},  
   {requires, pre_boot},  
   {enables, external_infrastructure}]}).
```

图 2.4: boot\_step 例子

前两行定义了这个 step 的名字和描述，mfa 是 (module, function, arguments) 的缩写，定义了如何启动这个 step；requires 指定了启动这个 step 所以依赖的前置条件，也就是必须等 requires 中的 module 都启动完毕才能启动这个 step；enables 指定了启动这个 step 之后就可以解锁的 module；

在图 2.3 的第 12 行中，通过 rabbit\_boot\_steps 模块来启动所有的 steps；从资源文件中我们知道 rabbitmq 有多个 application，而每个 application 都有自己的 boot\_step，所以必须有一个单独的 tool 能够把这些组织在一起，就是 rabbit\_boot\_steps 啦，如图 2.5：

这里为什么要用 boot\_step 的方式，而不是一行一行的按顺序编写？个人觉得主要是这种方式灵活，解耦解得比较干净，当需要给 rabbitmq 增加一个新的 plugin 时，也只需要编写自己的 app 即可。

顺便说下 DAG，DAG 可以在很多地方进行类似的优化；比如编译原理中代码块的 DAG 优化、spark 中基于 DAG 的多 stage 优化了 Hadoop 的 two stage；我们 soa 服务搞不清依赖关系时其实也可以通过 DAG 拓扑排序来理顺。

```

29
28 run_boot_steps() ->
27   | run_boot_steps(loaded_applications()).
26
25 run_boot_steps(Apps) ->
24   +- 2 | neas: [ok = run_step(Attrs, mfa) || {_, _, Attrs} <- find_steps(A
23
22 run_cleanup_steps(Apps) ->
21   +- 2 | neas: [run_step(Attrs, cleanup) || {_, _, Attrs} <- find_steps(Ap
20
19 loaded_applications() ->
18   | [App || {App, _, _} <- application:loaded_applications()].
17
16 find_steps() ->
15   | find_steps(loaded_applications()).
14
13 find_steps(Apps) ->
12   +- 2 | neas: All -> sort_boot_steps rabbit_misc:all_module_attributes(rab
11
10 run_step(Attributes, AttributeName) ->
9   +- 11 | neas: case [MFA] || {Key, MFA} <- Attributes,-----
8
7 vertices({AppName, _Module, Steps}) ->
6   | [{StepName, {AppName, StepName, Attrs}} || {StepName, Attrs} <- Steps].
5
4 edges({_AppName, _Module, Steps}) ->
3   +- 10 | neas: EnsureList = fun (L) when is_list(L) -> L;-----
2
1 sort_boot_steps(UnsortedSteps) ->

```

查找所有 app 的 boot step

拓扑排序

图 2.5: rabbit\_run\_steps.erl



## Chapter 3

## 网络层分析

### 3.1 TODO...



## Chapter 4

### 存储分析

#### 4.1 TODO...





# Chapter 5

## AMQP 协议分析

### 5.1 AMQP 协议

### 5.2 Broker

### 5.3 Vhost

### 5.4 Exchange

### 5.5 Queue

### 5.6 Message



## Chapter 6

# 高可用和集群分析

### 6.1 TODO...

