# 绑定端口错误 Address in use 分析

Email: geekidea@gmail.com

2019/02

## 0. 问题描述

最近开发同学反馈线上容器业务上线新版时，偶尔启动会失败，并报如下错误：

```
        at org.springframework.boot.loader.MainMethodRunner.run(MainMethodRunner.java:48)
        at org.springframework.boot.loader.Launcher.launch(Launcher.java:87)
        at org.springframework.boot.loader.Launcher.launch(Launcher.java:50)
        at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:51)
Caused by: org.apache.catalina.LifecycleException: Protocol handler start failed
        at org.apache.catalina.connector.Connector.startInternal(Connector.java:1020)
        at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:150)
        ... 21 common frames omitted
Caused by: java.net.BindException: Address in use
        at sun.nio.ch.Net.bind0(Native Method)
        at sun.nio.ch.Net.bind(Net.java:433)
        at sun.nio.ch.Net.bind(Net.java:425)
```

一般由 k8s 自动尝试 2 次（约 3 分钟，视该业务对应的 k8s deployment replica-controller 重试策略相关配置而定）启动后，最终会启动成功。为了减少这种偶发的新版上线（或业务重启）导致的不可用时间，需要对该问题分析处理。

## 1. 问题分析

业务容器基于 spring cloud 微服务架构，容器化后为便于非容器的主机访问该业务，采用主机网络模式（**hostNetwork**），因此该业务使用宿主机的 Linux 网络空间（Network namespace，网卡接口）绑定（或监听）端口，以及建立 socket 连接。通过 **strace** 命令跟踪典型的容器业务进程，该进程会建立以下 socket 连接：

| Sockets | 是否绑定端口 | 是否设置 SO_REUSEADDR | 备注 |
|---|---|---|---|
| 8***端口 | 是，并且绑定所有 ipv4 接口。 | 是 | 监听该端口，向下游业务提供服务。 |
| Redis 连接 | 是，每个 reids 连接均绑定随机源端口。 | 是 | 50-200 个左右，视业务连接池设置而定，程序初始化时创建该连接池，一般在监听端口之前创建好。 |
| HTTP 连接 | 否，源端口随机。 | 否 | 20 个左右，基于 Spring Cloud Feign 默认方式，与上游依赖业务主动建立的 HTTP 连接，会不定期重连。 |

在实际的业务内，除了在表格中列举的 Redis 连接池之外，个别业务内部还会维持其他 DB 连接池，如：Mysql 连接池，Mongodb 连接池，对于业务报错：**java.net.BindException: Address in use**，的堆栈上下文可知该端口被其他进程（或该进程内的其他线程）占用，因此底层执行绑定系统调用（syscall bind）出错，基于 socket

编程基础知识，该系统错误码（errno）为 **EADDRINUSE**（98），对应的头文件（如：/usr/include/asm-generic/errno.h）中定义如下：

```
#define         EADDRINUSE      98      /* Address already in use */
```

由于该业务对应的 k8s deployment 设置了 **hostPort**（值为待绑定监听的端口号，8***），并且 k8s scheduler 已经加载了预选策略 **PodFitsHostPorts**，因此可以排除 k8s 再次调度该业务容器到之前已存在（该业务的）对应宿主机节点的情况，即该宿主机节点之前不会存在相同的容器业务进程；进一步地在该宿主机查看所有端口监听情况：`ss -an | grep tcp | grep LISTEN`，看到有些 30000 以上的端口约 60 个，使用 losf 命令查看这些端口，如：`lsof -nNPi:30667` 确认所属进程为 kube-proxy，这些端口一般为 k8s Service 对应的 nodePort，该端口范围由 kube-apiserver 通过**--service-node-port-range** 指定，默认范围为 **30000-32767**，该范围与业务容器待监听的端口并不冲突。因此在该问题的环境下可以排除该宿主机上其他进程监听同一端口引起的该问题。

**SO_REUSEADDR 选项**：基于 TCP/IP 基本原理对于 TCP TIME-WAIT 状态的连接，系统会保持该连接一段时间（2*MSL，Linux 下一般是 1 分钟），在此期间内该 TIME-WAIT 连接对应的本地 IP 及端口，一般情况下是不允许再绑定的，除非该端口在绑定监听前设置了 SO_REUSEADDR socket 选项。例如：某**业务进程 A** 监听 **8001** 端口，并且（在绑定该端口）之前设置了 SO_REUSEADDR 选项，客户端通过连接该业务进程 IP:8001 请求业务进程处理，该业务进程处理完客户端的业务后主动关闭该连接，则在业务进程所在的宿主机（或容器所在的网络命名空间）该连接会进入 TIME-WAIT 状态，此时业务进程退出并执行重启动作，由于之前该端口绑定监听前设置 SO_REUSEADDR，因此重启后（即使之前该端口有处于 TIME-WAIT 状态的连接），可以再次绑定监听成功，并正常进入业务处理流程，对于现代的服务器程序（如：nginx，redis， tomcat8 等），默认在绑定端口前已经设置了 SO_REUSEADDR 选项。

上面的例子产生的 TIME-WAIT 连接是由之前的同样的业务进程通过 accept 系统调用从 TCP 全连接队列中取出并处理的，假如我们的 TIME-WAIT 连接对应的本地端口来自于同一宿主机另一个业务进程向上游主动打开的 HTTP 请求连接，此时业务进程 A 会监听成功吗？例如：同一宿主机的业务进程 B 主动连接其他机器的 HTTP 服务（例如：使用 Spring Cloud Feign http client 方式），发送 HTTP 请求时 HTTP Header 设置为 Connection: close，这样可以尽量保证业务进程 B 收到应答后，主动关闭该连接并进入 TIME-WAIT 状态，恰巧进程 B 使用的本地源端口也是 8001（因为之前该端口并没有被占用，即宿主机上进程 A 并不存在），在该 TIME-WAIT 状态期间内如果业务进程 A（容器）被调度到了该宿主机上，此时业务进程 A 会绑定监听失败，并报系统错误 EADDRINUSE，可以通过编写 socket 测试程序验证该现象：

```
$ ss -an | grep TIME | less
tcp    TIME-WAIT  0      0      172.23.12.209:56138              172.23.16.15:8715
tcp    TIME-WAIT  0      0      172.23.12.209:30978              172.23.16.218:8706
tcp    TIME-WAIT  0      0      172.23.12.209:8704               172.23.12.215:17701

$ export TT_REUSEADDR=1
$ ./bind_rand 56138 0
bind error fd 3, err(98): Address already in use
```

如上图，测试程序 bind_rand 尝试使用 SO_REUSEADDR 方式绑定监听 TIME-WAIT 状态 56138 端口，并且该端口在此之前并不由 bind_rand 进程监听，最终绑定失败。

以上分析了模拟了 TIME-WAIT 状态的源端口与本机另一进程待绑定监听端口冲突的影响，对于本机上已经建立的连接也是如此，例如：同一宿主机的业务进程 B 主动连接其他机器的 HTTP 服务，发送请求时 HTTP Header 设置为 Connection: keep-alive，这样可以尽量保证业务进程 B 收到应答后，后续会持续使用该 TCP 连接，恰巧进程 B 使用的本地端口也是 8001，在该期间内如果业务进程 A（容器）被调度到了该宿主机上，此时业务进程 A 会绑定监听失败，并报同样的系统错误，EADDRINUSE，通过测试程序验证该现象：

```
$ ss -an | grep EST | less
tcp    ESTAB      0      0      172.21.2.126:30900            100.100.30.25:80
tcp    ESTAB      0      0      172.21.2.126:22               172.21.0.86:49224
tcp    ESTAB      0      0      172.21.2.126:22               172.21.0.86:49177
tcp    ESTAB      0      0      172.21.2.126:22               172.21.0.86:49951
tcp    ESTAB      0      0      172.21.2.126:1027             172.21.0.86:63250

$ export TT_REUSEADDR=1
$ ./bind_rand 30900 0
bind error fd 3, port: 30900, err(98): Address already in use
```

如上图，测试程序 bind_rand 尝试使用 SO_REUSEADDR 方式绑定监听 ESTAB 状态 30900 端口，并且该端口在此之前并不由 bind_rand 进程监听，最终绑定失败。

由于主动发起的连接源端口一般由系统随机生成（除非建立连接前使用 bind 方式指定固定源端口），源端口范围由 **net.ipv4.ip_local_port_range** 指定，如果某业务进程待监听端口在此范围内，并且系统已经存在了来自另一个进程的同样源端口的 TIME-WAIT 或者 ESTAB 状态的连接，此时会监听失败并报系统错 **EADDRINUSE**，该文档所描述的问题正是因为该原因引起：k8s 宿主机上混部了多个容器业务，并且大部分使用了主机网络模式，容器业务进程 B 的 TIME-WAIT 或 ESTAB 的（HTTP）连接源端口与容器业务 A 进程的待监听端口（8***）发生了冲突。

## 2. 解决方案

综上该问题主要是由于系统内核参数 net.ipv4.ip_local_port_range 指定源端口范围与待监听的端口发生了重合，因此调整该系统内核参数，由之前 5000-65535 调整至 10000-65535，避免与业务进程待监听端口 8***的端口冲突，在使用 k8s docker 时需要注意：

1. 容器内执行类似 `sysctl -w "net.ipv4.ip_local_port_range=10000 65535"`命令修改（容器的）内核参数，除容器内需要 root 权限外，还需要该容器具备特权模式：**k8s securityContext privileged** 指定为 **true**；或者使用更安全的 **securityContext sysctls** 方式。

2. 如果容器使用了主机网络模式，修改容器的 net.ipv4.ip_local_port_range 后，会覆盖所在宿主机上的相应内核参数，需要特别引起注意。关于如何在 k8s 中设置容器的 sysctl 内核参数，可以参考官方文档。

## 3. 引申问题

以上通过修改本地随机端口范围避免了本机上源端口与待监听端口的冲突，设想以下几个场景：

**3.1** 进程 A 启动绑定监听指定端口（如：65432）成功后，本机的进程 B 可以使用该端口作为源端口连接其他服务器吗？

a. 当进程 A 不使用 SO_REUSEADDR 选项时，执行测试程序 bind_rand 模拟进程 A，执行测试程序 test_tcplocal 模拟进程 B，如下图：

```
$ export TT_REUSEADDR=0
$ ./bind_rand 65432 0
bind port: 65432

$ ss -an | grep 65432
tcp    LISTEN     0      65535     *:65432                 *:*

$ export TT_REUSEADDR=0
$ ./test_tcplocal 1 100.100.30.25 80 65432
bind error sock 3, err(98): Address already in use

$ export TT_REUSEADDR=1
$ ./test_tcplocal 1 100.100.30.25 80 65432
bind error sock 3, err(98): Address already in use
```

b. 当进程 A 使用 SO_REUSEADDR 选项时，执行测试程序 bind_rand 模拟进程 A，执行测试程序 test_tcplocal 模拟进程 B，如下图：

```
$ export TT_REUSEADDR=1
$ ./bind_rand 65432 0
bind port: 65432

$ ss -an | grep 65432
tcp    LISTEN     0      65535     *:65432                 *:*

$ export TT_REUSEADDR=0
$ ./test_tcplocal 1 100.100.30.25 80 65432
bind error sock 3, err(98): Address already in use

$ export TT_REUSEADDR=1
$ ./test_tcplocal 1 100.100.30.25 80 65432
bind error sock 3, err(98): Address already in use
```

可见当进程 A 已经绑定监听了所有网络接口*:65432 端口，进程 B 尝试使用 65432 作为源端口连接其他服务时（100.100.30.25:80）会失败，无论是否设置 SO_REUSEADDR 选项。

**3.2** 进程 B 先使用指定源端口（如：65432）连接其他服务器成功后，随后进程 A 能绑定监听该端口成功吗？

a. 当进程 B 不使用 SO_REUSEADDR 选项时，执行测试程序 bind_rand 模拟进程 A，执行测试程序 test_tcplocal 模拟进程 B，如下图：

```
$ export TT_REUSEADDR=0
$ ./test_tcplocal 1 100.100.30.25 80 65432
built 1 conn with 100.100.30.25:80 in 29 ms

$ ss -an | grep 65432 | grep EST
tcp    ESTAB      0      0      172.21.2.126:65432            100.100.30.25:80

$ export TT_REUSEADDR=0
$ ./bind_rand 65432 0
bind error fd 3, port: 65432, err(98): Address already in use

$ export TT_REUSEADDR=1
$ ./bind_rand 65432 0
bind error fd 3, port: 65432, err(98): Address already in use
```

b.当进程 B 使用 SO_REUSEADDR 选项时，执行测试程序 bind_rand 模拟进程 A，执行测试程序 test_tcplocal 模拟进程 B，如下图：

```
$ export TT_REUSEADDR=1
$ ./test_tcplocal 1 100.100.30.25 80 65432
built 1 conns with 100.100.30.25:80 in 30 ms

$ ss -an | grep 65432 | grep EST
tcp    ESTAB      0      0      172.21.2.126:65432              100.100.30.25:80

$ export TT_REUSEADDR=0
$ ./bind_rand 65432 0
bind error fd 3, port: 65432, err(98): Address already in use

$ export TT_REUSEADDR=1
$ ./bind_rand 65432 0
bind port: 65432

$ ss -an | grep 65432
tcp    LISTEN     0      65535      *:65432                 *:*
tcp    ESTAB      10     0      172.21.2.126:65432              100.100.30.25:80
```

可见必须在进程 A 和 B 同时使用 SO_REUSEADDR 的前提下，进程 B 使用 65432 源端口连接其他服务（100.100.30.25:80）后，进程 A 才能绑定监听该端口成功。

**3.3** 进程 A 启动时先绑定随机源端口（或指定绑定端口号为 65432）并用此 socket 成功连接远端 redis，随后试图绑定监听端口 65432，此时进程 A 可以绑定监听端口成功吗？

a. 当不使用 SO_REUSEADDR 选项时，执行测试程序 bind_rand，绑定失败，如下图：

```
$ export TT_REUSEADDR=0 TT_SERVIP="172.21.2.126" TT_SERVPORT=6379 TT_LOCALPORT=65432
$ ./bind_rand 65432 0
connected: 172.21.2.120:65432 -> 172.21.2.126:6379
bind error fd 4, port: 65432, err(98): Address already in use
```

b. 当使用 SO_REUSEADDR 选项时， 执行测试程序 bind_rand，绑定成功，如下图：

```
$ export TT_REUSEADDR=1 TT_SERVIP="172.21.2.126" TT_SERVPORT=6379 TT_LOCALPORT=65432
$ ./bind_rand 65432 0
connected: 172.21.2.120:65432 -> 172.21.2.126:6379
bind port: 65432
```

可见当使用 SO_REUSEADDR 时可以避免本进程内，主动连接的源端口与待绑定监听端口相同时（一般发生在 net.ipv4.ip_local_port_range 指定端口范围与待监听的端口发生了重合），导致绑定监听失败的情况。

**3.4** 进程 A 启动时先绑定监听端口 65432 成功后，随后新建 socket 并指定绑定源端口号 65432，该新建的 socket 可以绑定成功吗？

a. 当不使用 SO_REUSEADDR 选项时，执行测试程序 bind_rand，绑定失败，如下图：

```
$ export TT_REUSEADDR=0 TT_SERVIP="172.21.2.126" TT_SERVPORT=6379 TT_LOCALPORT=65432 TT_CONAFTER=1
$ ./bind_rand 65432 0
bind & listen port: 65432
bind error fd 4, port: 65432, err(98): Address already in use
```

b. 当使用 SO_REUSEADDR 选项时， 执行测试程序 bind_rand，绑定失败，如下图：

```
$ export TT_REUSEADDR=1 TT_SERVIP="172.21.2.126" TT_SERVPORT=6379 TT_LOCALPORT=65432 TT_CONAFTER=1
$ ./bind_rand 65432 0
bind & listen port: 65432
bind error fd 4, port: 65432, err(98): Address already in use
```

可见，无论是否设置 SO_REUSEADDR 选项，在绑定监听某端口后，第二次绑定该端口始终会失败。

**3.5** 非主机网络下的容器会出现本文最初描述的问题吗？

因为非主机网络下的容器，会使用各自不同的 Linux 网络命名空间（Network namaspace），因此容器 A 的网络连接源端口号与容器 B 绑定监听的端口号即使相同也是互不影响的，即容器 B 可以正常监听自己所属网络命名空间的 8***端口。

**3.6** 当使用 **SO_REUSEPORT** 选项时，上述问题 3.1 及 3.2 的执行结果如何？

```
$ export TT_REUSEPORT=1 TT_SERVIP="172.21.2.126" TT_SERVPORT=6379 TT_LOCALPORT=65432
$ ./bind_rand 65432 0
connected: 172.21.2.120:65432 -> 172.21.2.126:6379
bind port: 65432
```

```
$ export TT_REUSEPORT=1 TT_SERVIP="172.21.2.126" TT_SERVPORT=6379 TT_LOCALPORT=65432 TT_CONAFTER=1
$ ./bind_rand 65432 0
bind & listen port: 65432
connected: 172.21.2.120:65432 -> 172.21.2.126:6379
```

由测试结果可见，当设置 **SO_REUSEPORT** 选项时，第二次绑定（监听）同一端口都会成功。**SO_REUSEPORT** 选项由 Linux 内核 3.9 引入，旨在通过内核层做同一监听端口的负载均衡，将新晋的客户端连接均衡投递到（监听同一端口的）不同进程的全连接队列中，会避免早期内核下的 prefork 模式的（多进程，多线程）服务器程序中出现的 epoll 惊群问题，在 **SO_REUSEPORT** 选项未引入之前，nginx 通过对 accept 加锁的方式避免惊群，带来的性能问题，使用 SO_REUSEPORT 选项后，监听同一端口的进程可以不必是父子派生（fork）关系。使用 bind_rand 测试程序，启动两个 bind_rand 进程监听 8001 端口，结果如下图：

```
$ export TT_REUSEPORT=1
$ ./bind_rand 8001 1
$ ./bind_rand 8001 1
$ ss -an | grep 8001 | grep LISTEN
tcp    LISTEN     0      65535      *:8001                  *:*
tcp    LISTEN     0      65535      *:8001                  *:*
```

如上可见，ss 状态下有两条监听 8001 端口的条目，分别对应两个 bind_rand 进程，执行客户端连接程序向该 8001 端口发起 100 个连接（为便于观察，bind_rand 中并没有执行 accept 系统调用），此时查看该端口的状态，如下图：

```
$ ss -an | grep 8001 | grep LISTEN
tcp    LISTEN     47     65535      *:8001                  *:*
tcp    LISTEN     53     65535      *:8001                  *:*
```

如下可见，100 个连接最终由内核自动负载投递到了两个不同的 bind_rand 的进程的全连接队列中。

## 4. 后续思考

主机网络下的各容器业务共用宿主机的端口资源，做容器化落地时需要考虑相应的系统资源瓶颈，对于同一主机内的传统（非容器）业务也是如此。