Apusic 金蝶天燕

生成时间：2026-05-19

用户手册

金蝶Apusic Java开发工具包软件

版权声明

本文档所涉及的软件著作权、版权等知识产权已依法进行了注册，由金蝶天燕云计算股份有限公司合法拥有。受《中华人民共和国著作权法》《计算机软件保护条例》《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的版权信息由金蝶天燕云计算股份有限公司合法拥有，受法律的保护，金蝶天燕云计算股份有限公司对本文档可能涉及到的非金蝶天燕云计算股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经金蝶天燕云计算股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将被视为侵权，金蝶天燕云计算股份有限公司有依法追究其责任的权利。

本文档如有更新，不另行通知。对本文档中的问题您可向金蝶天燕云计算股份有限公司告知或查询。未经本公司明确授予的任何权利均予保留。

商标声明

Apusic是深圳市金蝶天燕云计算股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由金蝶天燕合法拥有，受法律保护。未经金蝶天燕的书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯金蝶天燕商标权的，金蝶天燕将依法追究其法律责任。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

目录

- 1 版本更新说明
- 2 前言
  - 2.1 CRaC介绍
  - 2.2 Apusic JDK 与CRaC
  - 2.3 产品简介
  - 2.4 范围和读者
  - 2.5 约定和术语
- 3 准备与配置 Apusic JDK with CRaC
  - 3.1 环境要求
  - 3.2 获取与配置 Apusic JDK with CRaC Support
  - 3.3 CRaC 相关参数与属性
- 4 通过 Jetty 示例一步步应用 CRaC
  - 4.1 准备 Jetty 应用
  - 4.2 集成 CRaC API 并实现 Resource 接口
    - 4.2.1 添加 org.crac 依赖
    - 4.2.2 实现 org.crac.Resource 接口
  - 4.3 创建检查点 (Checkpoint)
    - 4.3.1 准备应用以创建检查点
    - 4.3.2 触发检查点创建
    - 4.3.3 理解检查点文件
    - 4.3.4 检查点创建过程中的常见异常
  - 4.4 从检查点恢复 (Restore)
    - 4.4.1 启动 JVM 并从检查点恢复
    - 4.4.2 恢复过程详解
- 5 在容器环境中使用 CRaC
  - 5.1 构建基础应用镜像
  - 5.2 在容器中运行并创建检查点
  - 5.3 构建包含检查点的应用镜像
  - 5.4 从包含检查点的镜像启动容器
- 6 文件描述符策略
  - 6.1 配置文件及启用
  - 6.2 规则结构
    - 6.2.1 type (类型) - 必需
    - 6.2.2 action (操作) - 必需
    - 6.2.3 warn: false (可选)
  - 6.3 特定资源类型的匹配属性
    - 6.3.1 文件 (type: file)
    - 6.3.2 管道 (type: pipe)
    - 6.3.3 套接字 (type: socket)
    - 6.3.4 原始文件描述符 (type: filedescriptor)
  - 6.4 规则顺序的重要性
- 7 兼容性
  - 7.1 芯片架构
  - 7.2 Linux系统

1 版本更新说明

本文档根据实际进行更新，最新版本包含历史修改记录。

日期	手册版本	适用产品	更新说明
2025年8月	V17E01F01	金蝶 Apusic JDK with CRaC 17	首次编写

2 前言

2.1 CRaC介绍

Coordinated Restore at Checkpoint (CRaC) 是一项旨在显著提升 Java 应用启动性能的技术。其核心目标是大幅缩短应用从启动到能够处理首次业务请求的时间，并帮助应用更快达到峰值运行效率，同时有效降低传统预热阶段的资源消耗。

CRaC 的工作机制主要围绕两个核心概念：**检查点 (Checkpoint)** 和 **恢复 (Restore)**。

当一个 Java 应用正常运行并达到理想的"预热"状态——例如，所有必要的类均已加载，即时编译器（JIT）已完成关键代码的优化，各类缓存也已填充完毕——此时，CRaC 能够捕获整个 Java 进程在这一瞬间的精确状态，并将其完整地保存为一个或多个文件。这个捕获和保存的过程，我们称之为"创建检查点"。

之后，当需要再次启动该应用时，便可以直接从预先保存的检查点文件中"恢复"其状态，而无需再经历传统、耗时的完整启动流程。Java 虚拟机 (JVM) 会直接加载检查点中的内存镜像，使得应用几乎能够瞬间回到创建检查点时的运行状态。

值得注意的是，"Coordinated"（协调）是 CRaC 技术中的一个关键环节。这意味着应用本身需要与 CRaC 机制进行配合。具体而言，应用开发者需要通过 CRaC 提供的 API（主要是实现 `org.crac.Resource` 接口）来管理应用所使用的外部资源，例如文件句柄、网络连接等。在创建检查点之前，应用必须确保所有这类外部资源都已妥善关闭；而在从检查点成功恢复之后，应用则需要负责重新打开或初始化这些资源，以确保其后续能够正常工作。

采用 CRaC 技术能为 Java 应用带来多项显著优势：

- **大幅缩短启动时间:** 这是 CRaC 最引人注目的优点。通过跳过绝大部分初始化和预热步骤，应用启动时间可以从通常的数秒级别锐减至毫秒级别。
- **快速达到峰值性能:** 由于应用是从一个已经充分预热的状态恢复而来，因此能够迅速达到其最佳运行性能，无需漫长的等待。
- **降低资源消耗:** CRaC 有效减少了应用在启动和预热阶段对 CPU 和内存等系统资源的占用。
- **提升弹性伸缩能力:** 在云计算和微服务架构日益普及的今天，应用实例的快速启动能力对于实现高效的弹性伸缩至关重要，而 CRaC 正好满足了这一需求。

2.2 Apusic JDK 与CRaC

Apusic JDK 是由金蝶天燕（Apusic）公司基于 OpenJDK 项目构建并持续维护的 Java 开发工具包（JDK）发行版。为了满足广大用户对 Java 应用高性能和快速启动的迫切需求，Apusic JDK 团队始终紧密跟进社区的技术前沿，并为多个主流的长期支持（LTS）版本提供了广泛而深入的支持。

Apusic JDK 的上游技术源于华为公司开源的 BiSheng JDK。BiSheng JDK 本身在 OpenJDK 的坚实基础上，已经实施了一系列的性能优化和特性增强。Apusic JDK 不仅继承了这些来自 BiSheng JDK 的技术优势，更融入了自身在中间件领域长年积累的深厚经验，致力于为各类企业级应用提供一个既稳定又高效的 Java 运行时环境。目前，Apusic JDK 为 Java 8, 11, 17, 21 等多个核心的长期支持（LTS）版本提供构建和技术支持，确保用户能够获得持续、可靠的 Java 技术服务。

Apusic 团队敏锐地认识到 CRaC（Coordinated Restore at Checkpoint）技术在解决传统 Java 应用冷启动缓慢以及提升运行时效率方面所展现出的巨大潜力。尽管 CRaC 项目目前尚未正式合并到 OpenJDK 的主线版本中，Apusic 采用了与 Azul 等业界领先厂商相类似的技术路径，主动将其核心功能进行移植（Port），并成功集成到了 Apusic JDK 17 的 LTS 发行版之中。

为了方便用户根据其实际应用场景和具体需求进行灵活选择，针对已经集成了 CRaC 功能的 JDK 17，Apusic 特别提供了两种不同的发行版本：

- **标准的 Apusic JDK (17):** 此版本不包含 CRaC 功能，主要适用于那些不需要 Checkpoint/Restore 特性、运行在标准 Java 应用场景下的用户。
- **Apusic JDK with CRaC Support (17):** 这是一个内置了 CRaC 功能的特殊版本。用户可以选用此版本来开发、测试和部署那些希望利用 CRaC 技术进行启动优化和性能提升的 Java 应用程序。通过这个版本，用户可以充分体验到 CRaC 带来的毫秒级启动速度和快速达到峰值性能的优势。

2.3 产品简介

Apusic JDK 是一款高性能、生产环境就绪的OpenJDK发行版本，完全兼容开源 OpenJDK，基于华为毕昇 JDK发展而来，支持多种运行平台，具备更快的云应用启动速度，更好的性能以及提供更为便捷的分析、诊断工具，适合微服务、云原生应用、大数据等实际应用场景，提供最优的 Java 生产环境及解决方案。此外，Apusic JDK 是 Apusic 应用服务器的运行环境，适配了大量 Java 应用程序，解决了业务实际运行中遇到的多个问题，为 Java 应用程序提供一个安全、稳定、可扩展、高性能的 Java 运行环境。

Apusic JDK with CRaC是Apusic JDK内置了CRaC功能的特定版本，利用CRaC技术能够解决传统 Java 应用冷启动缓慢的问题，降低资源消耗，提升运行效率。

2.4 范围和读者

本手册介绍Apusic JDK产品安装相关的内容，主要适用于实施人员，维护人员等。

2.5 约定和术语

AJDK : 金蝶Apusic JDK（Apusic Java Development Kit，简称：AJDK）

3 准备与配置 Apusic JDK with CRaC

Apusic JDK 产品支持 Linux 下 aarch64 和 x64 CPU 架构下的安装部署。

3.1 环境要求

软件及操作系统环境要求

组件	要求
操作系统	Linux
CPU	64 (amd64), ARM64 (aarch64)
内核	Linux 内核版本建议为 3.11 或更高。CRaC 的核心依赖于 CRIU (Checkpoint/Restore In Userspace) 技术，较低版本的内核可能缺乏必要的支持。你可以运行 criu check 来验证内核是否提供所需支持
内存	4G及以上
硬盘	可用空间10G及以上

3.2 获取与配置 Apusic JDK with CRaC Support

1. 下载JDK

请从金蝶天燕官方渠道获取对应版本的 "Apusic JDK with CRaC Support" 版本。以 JDK 17 for Linux x86_64 为例，下载的文件可能类似于 `apusic-jdk-17.0.x.x-CRaC-linux-x86_64.tar.gz` 。

2. 解压缩 JDK

将下载的压缩包解压到你选择的安装目录下。

```
$ tar -zxvf apusic-jdk-17.0.x.x-CRaC-linux-x86_64.tar.gz
```

解压后，你会得到一个类似 `apusic-jdk-17.0.x.x-CRaC-linux-x86_64` 的目录，这个目录即为你的 JDK 安装目录 (通常称为 JAVA_HOME)。

3. 为 CRIU 设置权限

CRaC 功能依赖的 criu 工具（位于 JDK 的 lib 目录下）在执行检查点和恢复操作时需要特定的系统权限。你需要使用 root 权限执行以下命令，为 criu 可执行文件设置正确的用户所有权和 SUID 位。

```
$ sudo chown root:root ${JAVA_HOME}/lib/criu
$ sudo chmod u+s ${JAVA_HOME}/lib/criu
```

注意： `${JAVA_HOME}` 应指向你在上一步解压得到的 JDK 目录。例如解压到 `/opt/apusic-jdk-17.0.x.x-CRaC-linux-x86_64` ，则命令中的 `${JAVA_HOME}` 就应替换为该路径。

可以通过运行 `ls -la ${JAVA_HOME}/lib/criu` 检查权限是否设置正确，输出应表明 criu 文件归 root 用户所有，并且设置了 SUID 位（如 -rwsr-xr-x）。

4. 验证是否安装成功

执行 `java -version` 查看输出的版本是否为apusic jdk对应的版本。

3.3 CRaC 相关参数与属性

Apusic JDK with CRaC Support 提供了一系列 JVM 启动参数和系统属性，用于控制和配置 CRaC 的行为。

- 主要的 JVM 启动参数:
  - `-XX:CRaCCheckpointTo=<path>`

    指定创建检查点时，检查点文件的保存路径。<path>应为一个目录名。如果目录不存在，JVM 会尝试创建它。
  - `-XX:CRaCRestoreFrom=<path>`

    指定从检查点恢复时，从哪个路径加载检查点文件。<path>应为之前创建检查点时指定的目录。

- `-XX:CRaCMinPid=<value>`

  恢复进程的最小PID值。在容器环境下，为避免恢复时出现 PID 冲突的情况，可使用该参数设置一个较大的值，例如 1024。

- 常用的系统属性 (通过 -D 参数设置):
  - `-Djdk.crac.resource-policies=<file_path>`
  - 指向一个文件描述符策略配置文件。该文件定义了当检查点时遇到未关闭的文件、套接字等资源时的处理规则。详细用法请参考“Apusic JDK CRaC用户手册”。
  - `-Djdk.crac.collect-fd-stacktraces=true`

    当检查点因文件描述符未关闭而失败时，收集并打印打开该文件描述符处的 Java 堆栈跟踪信息，非常有助于调试。
  - `-Djdk.crac.trace-startup-time=true`

    启用此属性可以追踪和打印与 CRaC 启动时间相关的详细诊断信息。

- 更多参数信息

  关于 CRaC 支持的更完整的虚拟机选项列表及其详细说明，你可以参考 Azul 公司提供的 CRaC 文档，Apusic JDK 在此方面的参数基本保持一致。 此外，也请关注 Apusic JDK 官方文档中关于 CRaC 功能的最新说明。

4 通过 Jetty 示例一步步应用 CRaC

本章将通过一个简单的 Jetty Web 服务器示例，详细演示如何使一个 Java 应用支持 CRaC，以及如何创建检查点和从检查点恢复。

一个程序从检查点恢复时，其运行环境可能与创建检查点时的环境不同。程序需要能够感知并适应这些环境变化，例如打开的操作系统资源句柄（文件、套接字）、缓存的主机名或环境变量、在远程服务中的注册信息等。

目前，CRaC 的实现会在创建检查点时检查所有打开的文件和套接字。如果发现有未关闭的句柄，检查点创建过程会中止，并抛出异常，异常信息会描述相关的文件名或套接字地址。

完整的示例代码可以在 example-jetty 代码库中找到。

4.1 准备 Jetty 应用

我们从一个简单的 Jetty 应用开始：

```
// App.java
import org.eclipse.jetty.server.Handler;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.handler.AbstractHandler;
// ...
class ServerManager {
    Server server;

    public ServerManager(int port, Handler handler) throws Exception {
        server = new Server(port);
        server.setHandler(handler);
        server.start();
    }
}

public class App extends AbstractHandler
{
    static ServerManager serverManager;

    public void handle(String target,
                       Request baseRequest,
                       HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException {
        response.setContentType("text/html;charset=utf-8");
        response.setStatus(HttpServletResponse.SC_OK);
        baseRequest.setHandled(true);
        response.getWriter().println("Hello World");
    }

    public static void main( String[] args ) throws Exception {
        serverManager = new ServerManager(8080, new App());
        serverManager.server.join();
    }
}
```

在这个基础应用中，`main` 方法创建了一个 `ServerManager` 实例，该实例启动了一个 Jetty 服务器。

4.2 集成 CRaC API 并实现 Resource 接口

要使 Jetty 应用支持 CRaC，我们需要：

1. 添加 `org.crac` 依赖。
2. 识别应用中需要管理其状态（在检查点前后）的资源。
3. 实现 `org.crac.Resource` 接口来定义检查点前和恢复后的行为。
4. 注册这些资源。

4.2.1 添加 org.crac 依赖

首先，你需要在项目的构建配置中添加 `org.crac` 库的依赖。

Maven 示例 (pom.xml):

```
<dependency>
    <groupId>org.crac</groupId>
    <artifactId>crac</artifactId>
    <version>1.5.0</version>
</dependency>
```

`org.crac` 库的核心功能是作为 CRaC API 的一个适配器。

在编译时，`org.crac` 包提供了与 `jdk.crac` 完全镜像的 API 接口。

在运行时，`org.crac` 使用反射来检测 JDK 中是否存在 CRaC 的实现。如果存在，所有对 `org.crac` API 的调用都会被传递给 JDK 中的实际 CRaC 实现。如果不存在，这些调用会被转发到一个虚拟的实现，允许应用在没有 CRaC 功能的 JVM 上正常运行。

4.2.2 实现 org.crac.Resource 接口

对于 Jetty 服务器，`ServerManager` 类是管理网络连接的关键。因此，我们可以让 `ServerManager` 实现 `org.crac.Resource` 接口。

```
import java.util.Arrays;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

import org.crac.Context;
import org.crac.Core;
import org.crac.Resource;

import org.eclipse.jetty.server.Handler;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.Request;
import org.eclipse.jetty.server.handler.AbstractHandler;
import org.eclipse.jetty.util.component.LifeCycle;

class ServerManager implements Resource {
    Server server;

    public ServerManager(int port, Handler handler) throws Exception {
        server = new Server(port);
        server.setHandler(handler);
        server.start();
        Core.getGlobalContext().register(this);
    }

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) {
        // Stop the connectors only and keep the expensive application running
        Arrays.asList(server.getConnectors()).forEach(c -> LifeCycle.stop(c));
    }
```

```
    @Override
    public void afterRestore(Context<? extends Resource> context) {
        Arrays.asList(server.getConnectors()).forEach(c -> LifeCycle.start(c));
    }
}
```

- **beforeCheckpoint(Context<? extends Resource> context) **
  - 在 JVM 准备创建检查点之前被调用。
  - 此方法的核心职责是释放所有外部资源。对于 Jetty 服务器，这意味着需要停止服务器，从而关闭监听的网络套接字。
- afterRestore(Context<? extends Resource> context)
  - 在 JVM 从检查点成功恢复之后被调用。
  - 此方法的核心职责是重新初始化或重新获取之前释放的外部资源。对于 Jetty 服务器，这意味着需要重新启动服务器，使其能够再次接受网络连接。
- 资源注册：
  - 通过 `Core.getGlobalContext().register(this);`，将 `ServerManager` 实例注册到全局 CRaC 上下文中。这样，CRaC 机制才能在检查点和恢复时回调其 `beforeCheckpoint` 和 `afterRestore` 方法。
- 资源注册顺序：
- 如果应用中有多个 `Resource` 实现，它们的注册顺序非常重要。 `beforeCheckpoint` 方法会按照注册的顺序被调用，而 `afterRestore` 方法会按照注册顺序的**相反顺序**被调用。这种设计有助于处理资源间的依赖关系

4.3 创建检查点 (Checkpoint)

在应用代码支持 CRaC 后，我们就可以创建检查点了。

4.3.1 准备应用以创建检查点

1. 编译并打包应用

   ```
   $ mvn clean package
   # 生成类似 target/example-jetty-1.0-SNAPSHOT.jar 的文件
   ```

2. 启动应用并启用CRaC

   使用 `-XX:CRaCCheckpointTo=<path>` 参数启动应用，指定检查点文件的保存路径。

   ```
   # 启动应用
   $ java -XX:CRaCCheckpointTo=crac-files -jar target/example-jetty-1.0-SNAPSHOT.jar
   ```

3. 预热应用（Warm-up）

   在创建检查点之前，通常建议先对应用进行预热，即发送一些典型的请求，让 JVM 完成 JIT 编译、类加载、缓存填充等操作，使应用达到一个理想的运行状态。

   ```
   $ curl http://localhost:8080
   Hello World
   ```

4.3.2 触发检查点创建

通过 jcmd 命令，可以在应用运行时从外部触发检查点的创建。

```
# 找到应用的进程 PID 或其主类 JAR 名称。
$ jps -l
2204343 target/example-jetty-1.0-SNAPSHOT.jar
2205390 jdk.jcmd/sun.tools.jps.Jps

# 执行 jcmd 命令
$ jcmd 2204343 JDK.checkpoint
```

```
# 或者
$ jcmd target/example-jetty-1.0-SNAPSHOT.jar JDK.checkpoint
```

如果一切顺利，应用控制台会输出类似以下信息，JVM 被 CRIU 暂停并转储，然后原 Java 进程通常会终止。

```
INFO: Starting checkpoint
2025-05-21 08:39:31.369:INFO:oejs.AbstractConnector:Attach Listener: Stopped
ServerConnector@299a06ac{HTTP/1.1, (http/1.1)}{0.0.0.0:8080}
May 21, 2025 8:39:31 AM jdk.internal.crac.LoggerContainer info
INFO: /home/mazhen/works/example-jetty/target/dependency/crac-1.5.0.jar is recorded as always available on
restore
...
[295.210s][info][crac] Checkpoint ...
[1]    2204343 killed     java -XX:CRaCCheckpointTo=crac-files -jar
```

4.3.3 理解检查点文件

在指定的检查点路径（`/tmp/crac-checkpoint-jetty`）下，你会看到生成的检查点文件（如 pages-1.img, core-XXXX.img 等）。

```
$ ls crac-files/
core-2204343.img  core-2204353.img  core-2204365.img  core-2204377.img  core-2205897.img     pages-1.img
core-2204344.img  core-2204354.img  core-2204368.img  core-2204378.img  core-2205898.img     pstree.img
core-2204345.img  core-2204355.img  core-2204369.img  core-2204797.img  dump4.log            seccomp.img
core-2204346.img  core-2204356.img  core-2204370.img  core-2204804.img  fdinfo-2.img         stats-dump
core-2204347.img  core-2204357.img  core-2204371.img  core-2205891.img  files.img            timens-0.img
core-2204348.img  core-2204358.img  core-2204372.img  core-2205892.img  fs-2204343.img       tty-info.img
core-2204349.img  core-2204359.img  core-2204373.img  core-2205893.img  ids-2204343.img
core-2204350.img  core-2204360.img  core-2204374.img  core-2205894.img  inventory.img
core-2204351.img  core-2204361.img  core-2204375.img  core-2205895.img  mm-2204343.img
core-2204352.img  core-2204364.img  core-2204376.img  core-2205896.img  pagemap-2204343.img
```

检查点文件是应用在特定时刻的完整内存镜像和进程状态的快照。这些文件由 CRIU 生成，并存储在 -XX:CRaCCheckpointTo 指定的目录中。主要文件包括：

- `core-<pid>.img`：包含进程核心转储信息。
- `pages-<pid>.img`：包含进程的内存页数据。
- 其他辅助文件，描述了文件描述符、内存映射等。

这些文件合在一起构成了可用于恢复应用状态的完整检查点。

4.3.4 检查点创建过程中的常见异常

如果在 `beforeCheckpoint` 阶段未能正确关闭所有必需的资源，CRaC 会抛出异常，检查点创建会失败。

- CheckpointOpenSocketException

  表示有网络套接字未关闭。在我们的 Jetty 示例中，如果 `ServerManager` 没有实现 `Resource` 接口或 `beforeCheckpoint` 未能停止服务器，就会遇到这个异常。异常信息通常会指明哪个套接字（如 tcp6 localAddr :: localPort 8080 ...）仍然打开。

  ```
  An exception during a checkpoint operation:
  jdk.internal.crac.mirror.CheckpointException
      Suppressed: jdk.internal.crac.mirror.impl.CheckpointOpenSocketException:
  sun.nio.ch.ServerSocketChannelImpl[/[0:0:0:0:0:0:0:0]:8080]
          at
  java.base/jdk.internal.crac.JDKSocketResourceBase.lambda$beforeCheckpoint$0(JDKSocketResourceBase.java
          at java.base/jdk.internal.crac.mirror.Core.checkpointRestore1(Core.java:170)
          at java.base/jdk.internal.crac.mirror.Core.checkpointRestore(Core.java:315)
          at java.base/jdk.internal.crac.mirror.Core.checkpointRestoreInternal(Core.java:328)
  ```

- CheckpointOpenFileException

  表示有文件描述符未关闭。异常信息会指明哪个文件路径（如 `/path/to/some/file.txt`）对应的文件描述符仍然打开。

- CheckpointOpenResourceException

  表示有其他类型的本地资源（非文件或套接字）未关闭。

如果遇到这类异常，可以使用 `-Djdk.crac.collect-fd-stacktraces=true` 系统属性。当检查点因文件描述符未关闭而失败时，该属性会让 JVM 打印出打开该文件描述符时的 Java 堆栈跟踪信息，这对于定位问题的根源非常有帮助。

```
# 应用启动命令
$ java -XX:CRaCCheckpointTo=crac-files -Djdk.crac.collect-fd-stacktraces=true -jar target/example-jetty-1.0-SNAPSHOT.jar

# checkpoint 时异常堆栈信息
An exception during a checkpoint operation:
jdk.internal.crac.mirror.CheckpointException
    Suppressed: jdk.internal.crac.mirror.impl.CheckpointOpenSocketException: sun.nio.ch.ServerSocketChannelImpl[/[0:0:0:0:0:0:0:0]:8080]
        at java.base/jdk.internal.crac.JDKSocketResourceBase.lambda$beforeCheckpoint$0(JDKSocketResourceBase.java:69)
        at java.base/jdk.internal.crac.mirror.Core.checkpointRestore1(Core.java:170)
        at java.base/jdk.internal.crac.mirror.Core.checkpointRestore(Core.java:315)
        at java.base/jdk.internal.crac.mirror.Core.checkpointRestoreInternal(Core.java:328)
    Caused by: java.lang.Exception: This file descriptor was created by main at epoch:1747817822686 here
        at java.base/jdk.internal.crac.JDKFdResource.<init>(JDKFdResource.java:65)
        at java.base/jdk.internal.crac.JDKSocketResourceBase.<init>(JDKSocketResourceBase.java:45)
        at java.base/jdk.internal.crac.JDKSocketResource.<init>(JDKSocketResource.java:40)
        at java.base/sun.nio.ch.ServerSocketChannelImpl$Resource.<init>(ServerSocketChannelImpl.java:740)
        at java.base/sun.nio.ch.ServerSocketChannelImpl.<init>(ServerSocketChannelImpl.java:82)
        at java.base/sun.nio.ch.ServerSocketChannelImpl.<init>(ServerSocketChannelImpl.java:115)
        at java.base/sun.nio.ch.SelectorProviderImpl.openServerSocketChannel(SelectorProviderImpl.java:72)
        at java.base/java.nio.channels.ServerSocketChannel.open(ServerSocketChannel.java:145)
        at org.eclipse.jetty.server.ServerConnector.openAcceptChannel(ServerConnector.java:340)
        at org.eclipse.jetty.server.ServerConnector.open(ServerConnector.java:310)
        at org.eclipse.jetty.server.AbstractNetworkConnector.doStart(AbstractNetworkConnector.java:80)
        at org.eclipse.jetty.server.ServerConnector.doStart(ServerConnector.java:234)
        at org.eclipse.jetty.util.component.AbstractLifeCycle.start(AbstractLifeCycle.java:73)
        at org.eclipse.jetty.server.Server.doStart(Server.java:401)
        at org.eclipse.jetty.util.component.AbstractLifeCycle.start(AbstractLifeCycle.java:73)
        at com.example.ServerManager.<init>(App.java:24)
        at com.example.App.main(App.java:56)
```

4.4 从检查点恢复 (Restore)

创建检查点后，我们就可以从这些检查点文件快速启动应用的新实例。

4.4.1 启动 JVM 并从检查点恢复

使用 `-XX:CRaCRestoreFrom=<path>` JVM 参数来指定从哪个检查点路径恢复。

```
$ java -XX:CRaCRestoreFrom=crac-files
```

4.4.2 恢复过程详解

1. JVM 加载检查点状态

JVM (通过 CRIU) 会读取检查点目录中的文件，将应用的内存镜像和进程状态直接加载到内存中。这跳过了大部分传统的 JVM 初始化和应用启动过程。

2. **调用 afterRestore 方法**

   一旦状态加载完毕，CRaC 机制会按照注册 `Resource` 实例时的**相反顺序**，调用每个 `Resource` 的 `afterRestore` 方法。

   在我们的 Jetty 示例中， `ServerManager` 的 `afterRestore` 方法会被调用，它会重新启动 Jetty 服务器。

应用控制台会输出类似以下信息：

```
$ java -XX:CRaCRestoreFrom=/tmp/crac-checkpoint-jetty
2025-05-21 09:04:16.353:INFO:oejs.AbstractConnector:Attach Listener: Started
ServerConnector@299a06ac{HTTP/1.1, (http/1.1)}{0.0.0.0:8080}
```

此时，Jetty 服务器已经从检查点恢复并开始运行。你可以立即通过 curl http://localhost:8080 访问它。 通过这个 Jetty 示例，我们演示了开发、创建检查点和恢复 CRaC 应用的完整流程。核心在于正确实现 Resource 接口来管理外部资源，并在合适的时机触发检查点创建和恢复。

5 在容器环境中使用 CRaC

CRaC 技术非常适合在容器化环境（如 Docker）中使用，以进一步加速容器的启动和伸缩。

5.1 构建基础应用镜像

首先，创建一个包含 `Apusic JDK with CRaC` 和你的应用程序 JAR 包的 Docker 镜像。

Dockerfile 示例 (用于创建检查点的基础镜像):

```
FROM apusic-jdk:17.0.15-crac-x86_64

ARG APP_DIR=/opt/app
WORKDIR ${APP_DIR}

COPY example-jetty-1.0-SNAPSHOT.jar ${APP_DIR}/example-jetty-1.0-SNAPSHOT.jar

COPY dependency/ ${APP_DIR}/dependency/

EXPOSE 8080

CMD [ "java", \
        "-XX:CRaCCheckpointTo=/opt/crac-files", \
        "-XX:CRaCMinPid=1024", \
        "-jar", \
        "/opt/app/example-jetty-1.0-SNAPSHOT.jar" \
]
```

构建此镜像:

```
$ docker build -t example-jetty .
```

5.2 在容器中运行并创建检查点

1. **运行容器并准备创建检查点**

启动容器时，需要赋予 CRIU 所需的特权，并挂载一个卷用于保存检查点文件。

```
docker run -it --rm -d \
    --cap-add=CHECKPOINT_RESTORE \
    --cap-add=SYS_PTRACE \
    -p 8080:8080 \
    --name example-jetty \
    -v docker/restore/crac-files:/opt/crac-files
    example-jetty
```

- `--cap-add=CHECKPOINT_RESTORE --cap-add=SYS_PTRACE`

  为容器添加必要的 Linux Capabilities。

- `-v docker/restore/crac-files:/opt/crac-files`

  将宿主机的目录挂载到容器的 `/opt/crac-files`，检查点文件将保存在这里。

- `-XX:CRaCCheckpointTo=/opt/crac-files`

  该参数定义在 Dockerfile 中，告诉 JVM 在容器内的 `/opt/crac-files` 目录创建检查点。

2. **预热应用**

容器启动后，可以通过 curl 或其他方式访问应用进行预热。

3. **在容器内触发检查点**

使用 docker exec 在正在运行的容器内执行 jcmd 命令。

首先查看 Java 进程 ID：

```
$ docker exec example-jetty jcmd
1025 /opt/app/example-jetty-1.0-SNAPSHOT.jar
1060 jdk.jcmd/sun.tools.jcmd.JCmd
```

然后触发创建检查点：

```
$ docker exec example-jetty jcmd 1025 JDK.checkpoint
```

检查点文件会保存在容器的 `/opt/crac-files` 目录，由于卷挂载，它们也会出现在宿主机的 `docker/restore/crac-files` 目录中。容器内的 Java 应用在创建检查点后会退出，容器也会随之停止。

5.3 构建包含检查点的应用镜像

现在，我们可以基于上一步创建的检查点文件，构建一个新的 Docker 镜像，这个镜像启动时就会直接从检查点恢复。

```
FROM example-jetty

ARG RESTORE_DIR=/opt/restore
WORKDIR ${RESTORE_DIR}

# 复制检查点文件到镜像中
COPY crac-files ${RESTORE_DIR}/crac-files/

CMD [ "java", "-XX:CRaCRestoreFrom=/opt/restore/crac-files"]
```

构建此镜像：

```
docker build -t example-jetty_restore .
```

5.4 从包含检查点的镜像启动容器

现在，你可以快速启动这个包含检查点的应用镜像了。

```
docker run -it --rm -d \
    -p 8080:8080 \
    --name example-jetty_restore \
    example-jetty_restore
```

容器会非常快地启动，Jetty 服务器几乎立即就能提供服务。`-p 8080:8080` 用于将容器的 8080 端口映射到宿主机的 8080 端口。

注意 PID 冲突问题。

在容器环境中，特别是当 Java 应用作为 PID 1 运行时，恢复时可能会遇到 PID 冲突。可以使用 `-XX:CRaCMinPid=<pid>` 选项在创建检查点时强制将进程的 PID 移动到一个大于指定 `<pid>` 的值，以避免恢复时的冲突。例如，`-XX:CRaCMinPid=1024`。

6 文件描述符策略

CRaC 要求应用程序在创建检查点之前关闭所有打开的文件、网络连接等资源。在 Linux 系统中，这些资源都表现为文件描述符。然而，在某些情况下，修改应用程序（尤其是包含你无法修改的库代码时）以使其与检查点操作正确协调可能会非常困难。针对这些情况，CRaC 提供了一种通过配置文件来有限度地处理这些未关闭文件描述符的机制。

6.1 配置文件及启用

你可以通过设置系统属性 `jdk.crac.resource-policies` 指向一个配置文件来启用此功能。该文件包含一系列规则，规则之间用三个短横线 `(---)` 分隔。以 `#` 号 开头的行将被忽略。每个规则由若干 `key: value` 对组成，这实际上是 YAML 格式的一个子集，因此建议使用 `.yaml` 或 `.yml` 作为文件扩展名，以便在编辑器中获得更好的语法高亮支持。

示例配置文件 (fd-policies.yml):

```
type: file
path: /path/to/my/file
action: close
---
# 这是一个注释示例
type: FILE # 类型不区分大小写
path: "**/*.log" # 支持 Glob 模式
action: reopen
```

6.2 规则结构

每个规则都必须包含 type 和 action 这两个属性，它们的值不区分大小写。

6.2.1 type (类型) - 必需

定义了规则所针对的资源类型：

- `file`：指向本地文件系统上的文件或目录。
- `pipe`：指匿名管道。命名管道应通过 file 类型进行处理。
- `socket`：指网络套接字（如 TCP, UDP）或 Unix 域套接字。
- `filedescriptor`：指那些无法通过上述类型识别的原始文件描述符。

6.2.2 action (操作) - 必需

定义了当匹配到未关闭的资源时应执行的操作：

- `error` (默认操作): 打印错误信息并导致检查点创建失败。
- `ignore`：将打开的句柄交由底层的 CRIU 引擎处理。CRIU 会尝试在恢复时验证并重新打开它。
- `close`：在创建检查点之前关闭该句柄。如果应用程序在恢复后尝试使用这个已关闭的句柄，将会导致运行时异常。
- `reopen`：(主要用于 file 和 socket 类型) 在创建检查点前关闭句柄，并在恢复后尝试重新打开它。
  - 对于文件，会尝试恢复到相同的文件偏移量。
  - 对于监听套接字，会尝试在相同的本地地址上重新监听。

6.2.3 warn: false (可选)

如果设置了 `warn: false`，当规则匹配并执行非 error 操作时，将禁止在日志系统中打印相关的警告信息。默认情况下，会打印警告

6.3 特定资源类型的匹配属性

除了通用的 `type` 和 `action`，不同类型的资源还有其特定的匹配属性。

6.3.1 文件 (type: file)

- `path`：用于指定文件路径，支持 Glob 模式匹配。例如：
  - `path: /var/log/app.log`
  - `path: "/tmp/output_*.txt`
  - `path: "**/cache_files/**"`

6.3.2 管道 (type: pipe)

匿名管道没有特定的标识符供策略文件匹配，因此通常最多只有一个针对所有匿名管道的通用规则。可用的 action 包括 error、ignore 和 close，其含义与文件类型中的相同。

6.3.3 套接字 (type: socket)

可以使用以下一个或多个属性来精确匹配套接字：

- family: 套接字家族。
  - ipv6 或 inet6: IPv6 套接字。
  - ipv4 或 inet4: IPv4 套接字。
  - ip 或 inet: 任意 IPv4/IPv6 套接字。
  - unix: Unix 域套接字。
- localAddress 和 remoteAddress: 本地和远程地址。可以使用 * 来匹配任何已绑定的地址。
- localPort 和 remotePort: 本地和远程端口号（数字）。可以使用 * 来匹配任何端口。
- localPath 和 remotePath: 用于 Unix 域套接字，支持 Glob 模式匹配。
- listening: true: 仅选择处于监听状态的服务器套接字。

对于套接字，action 可以是 error、ignore、close。此外，reopen 操作允许在检查点前关闭监听（服务器）套接字，并在恢复后使用相同的本地地址重新打开它们。

6.3.4 原始文件描述符 (type: filedescriptor)

在某些情况下，文件描述符可能是在没有对应的高级 Java 对象（如 FileOutputStream）的情况下创建的。这类描述符可以通过以下方式识别：

- value: <数字>: 使用其数字值进行匹配，例如 value: 123。
- regex: <正则表达式>: 使用其本地描述字符串进行匹配，遵循 `java.util.regex.Pattern.compile()` 的语法。例如 `regex: ".*some_native_handle.*"`。

对于原始文件描述符，仅支持 error、ignore 和 close 操作。

6.4 规则顺序的重要性

配置文件中规则的顺序至关重要。对于每一个在检查点时发现的打开的文件描述符，CRaC 会从上到下遍历规则列表，并应用第一个匹配到的规则。一旦找到匹配的规则，后续的规则将被忽略。因此，更具体的规则应该放在更通用的规则之前。

文件描述符策略应被视为一种辅助手段，尤其是在无法直接修改代码的情况下。最佳实践仍然是通过实现 `org.crac.Resource` 接口来显式管理应用资源的生命周期。过度依赖策略可能会掩盖应用中潜在的资源管理问题。

7 兼容性

广泛兼容国产化操作系统和芯片。

7.1 芯片架构

- 鲲鹏
- 飞腾
- 海光
- 海思

7.2 Linux系统

- 国产Linux操作系统：麒麟操作系统、统信UOS、深度Linux、优麒麟、红旗Linux、中标麒麟、华为欧拉、中科方德、凝思等
- RedHat 系列
- CentOS 系列
- Ubuntu 系列
- Suse Linux 系列

全国统一服务热线

4008-555-800

金蝶天燕云计算股份有限公司(简称“金蝶天燕云”)成立于2000年，前身为“金蝶中间件公司”，是金蝶集团旗下新一代软件基础云平台服务商，云计算国家标准制定企业，国家信创产业核心软件企业。金蝶天燕是国家863重点研发计划与核高基重大专项承接企业，也是“两网一站四库十二金”国家重点工程的基础平台提供商，产品广泛应用于政府、军工、金融、能源等关键行业，累计服务客户总数超过10万家。

Apusic 金蝶天燕

云计算国家标准制定企业

金蝶集团旗下基础软件企业

信息技术应用创新核心企业

官网：www.apusic.com