

# Jakarta Concurrency 안내서

JEUS 9

**TMAXSOFT**

## 저작권 공지

Copyright 2024. TmaxSoft Co., Ltd. All Rights Reserved.

## 제한된 권리

이 소프트웨어(JEUS®) 사용설명서와 프로그램은 저작권법과 국제 조약에 의해 보호됩니다. 사용설명서와 프로그램은 TmaxSoft Co., Ltd.와의 사용권 계약 하에서만 사용할 수 있으며, 사용설명서는 사용권 계약의 범위 내에서만 배포 또는 복제할 수 있습니다. 이 사용설명서의 전부 또는 일부를 TmaxSoft의 사전 서면 동의 없이 전자, 기계, 녹음 등의 수단으로 전송, 복제, 배포하거나 2차적 저작물을 작성할 수 없습니다.

이 소프트웨어 사용설명서와 프로그램의 사용권 계약은 어떠한 경우에도 사용설명서 및 프로그램과 관련된 지적 재산권(등록 여부를 불문)을 양도하는 것으로 해석되지 않으며, 브랜드나 로고, 상표 등을 사용할 권한을 부여하지 않습니다. 사용설명서는 오로지 정보 제공만을 목적으로 하며, 이로 인한 계약상의 직접적 또는 간접적 책임을 지지 않습니다. 또한 사용설명서 상의 내용이 법적 또는 상업적인 특정 조건을 만족시킬 것을 보장하지 않습니다. 사용설명서는 제품의 업그레이드나 수정에 따라 예고 없이 변경될 수 있으며, 내용상의 오류가 없음을 보장하지 않습니다.

## 상표 공지

JEUS®는 TmaxSoft Co., Ltd.의 등록 상표입니다.

Java, Solaris는 Oracle Corporation 및 그 자회사, 관계회사의 등록 상표입니다.

Microsoft, Windows, Windows NT는 Microsoft Corporation의 등록 상표 또는 상표입니다.

HP-UX는 Hewlett Packard Enterprise Company의 등록 상표입니다.

AIX는 International Business Machines Corporation의 등록 상표입니다.

UNIX는 X/Open Company, Ltd.의 등록 상표입니다.

Linux는 Linus Torvalds의 등록 상표입니다.

Noto는 Google Inc.의 상표입니다. Noto 글꼴은 오픈 소스입니다. 모든 Noto 글꼴은 SIL Open Font License, 버전 1.1에 따라 게시됩니다. (<https://www.google.com/get/noto/>)

기타 모든 제품들과 회사 이름은 각각 해당 소유주의 상호, 상표, 또는 등록 상표입니다.

본 사용설명서에 기재된 회사, 시스템, 제품 이름 등에 반드시 상표 표시(™, ®)를 하지는 않습니다.

## 오픈 소스 소프트웨어 공지

본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다.: APACHE2.0, CDDL1.0, EDL1.0, OPEN SYMPHONY SOFTWARE1.1, TRILEAD-SSH2, Bouncy Castle, BSD, MIT, SIL OPEN FONT1.1

관련 상세 정보는 제품의 다음 디렉터리에 기재된 사항을 참고하시기 바랍니다. :

\${INSTALL\_PATH}/license/oss\_licenses

## 기술 지원

홈페이지: <https://www.tmaxsoft.com>

기술 서비스 센터: 1544-8629

## 안내서 이력

제품 버전	안내서 버전	발행일	비고
JEUS 9	3.1.1	2024-09-27	-

# 목차

1. Jakarta Concurrency 표준 .....	1
1.1. 개요 .....	1
1.2. Managed Task .....	1
1.3. 컨테이너 스레드 컨텍스트 .....	1
2. Managed Objects .....	3
2.1. ManagedExecutorService .....	3
2.2. ManagedScheduledExecutorService .....	5
2.3. ContextService .....	7
2.4. ManagedThreadFactory .....	9

# 1. Jakarta Concurrency 표준

본 장에서는 Jakarta Concurrency의 등장 배경 및 기술에 대해서 간략하게 설명한다.

## 1.1. 개요

애플리케이션 서버에서는 EJB나 웹 컴포넌트에서 Java SE에서 제공하는 Concurrency API(Thread, Timer, ExecutorService, ...)를 이용하는 것을 권하지 않는다. Jakarta EE 애플리케이션 컴포넌트(Servlet, EJB 등)는 애플리케이션 서버가 관리하는 스레드에서 실행되고, 같은 스레드에서 컨테이너가 제공하는 기능들이 동작하는 것을 가정한다.

이러한 이유로 APP components는 컨테이너가 관리하지 않는 스레드에서는 Jakarta EE 서비스들을 안전하게 사용할 수 없다. 또한, 컨테이너가 관리하지 않는 스레드에서 리소스를 이용하게 되면 EE에서 사용성, 보안, 신뢰성, 확장성에 잠재적으로 문제를 일으킬 수 있다.

이와 같은 관리되지 않는(unmanaged) 스레드에 의해 발생하는 문제를 해결하기 위해서 기존 Java SE의 Concurrency Utilities를 확장한 표준화된 Jakarta Concurrency 스펙이 제공된다.

이 표준화된 기술을 통해 애플리케이션이 Jakarta EE 환경에서 컨테이너의 integrity를 해치지 않고 동작하는 것을 보장할 수 있다.

## 1.2. Managed Task

컨테이너는 필요없는 리소스의 소모를 줄이기 위해 리소스를 풀링(pooling)하고 생명주기를 관리한다. 그런데 컴포넌트 내에서 Java SE에서 제공되는 Concurrency API를 이용하여 비동기 작업을 수행하면, 컨테이너가 해당 리소스에 대해서 인지할 수 없기 때문에 관리할 수가 없다.

따라서 본 스펙에서는 기존 Java SE의 java.util.concurrent에 정의된 Task를 확장하여 Managed Task를 정의하였다. 이를 통해 컨테이너가 해당 일반 task를 Managed Task로 관리할 수 있게 되고, 비동기로 작업이 실행될 때 Execution Context를 유지하여 실행될 수 있도록 한다.

## 1.3. 컨테이너 스레드 컨텍스트

Jakarta EE 환경에서는 서비스를 실행할 때 각 서비스의 컨텍스트 정보를 가지고 있다. 예를 들면 JDBC에서의 data sources, JMS에서의 프로바이더와 EJB 등이 있다.

컴포넌트 내에서 Java SE의 Concurrency API를 이용하게 되면, 컨테이너가 새롭게 생성된 스레드에서 서비스의 컨텍스트 정보를 유지하기 위해 애플리케이션 개발자는 다음과 같은 방식으로 컨텍스트를 전파해야 한다.

1. 애플리케이션 컴포넌트 스레드의 컨테이너 컨텍스트를 저장한다.
2. 어떤 컨테이너 컨텍스트를 저장하고 전파할지 판단한다.
3. 새롭게 생성된 스레드에 컨테이너 컨텍스트를 적용한다.
4. 원래 컴포넌트 스레드의 컨텍스트를 복구한다.

이와 같은 방식으로 작업이 Java SE의 Concurrency API에서 컨텍스트를 유지하면서 동작할 수 있지만, Jakarta Concurrency 서비스를 이용하면 간단하게 사용자가 정의한 Task를 Managed Objects로 전달하여 내부에서 컨텍스트를 알아서 유지/복구하기 때문에 편리하고 안전하다.

## 컨텍스트를 유지하여 작업을 수행하는 지점

- `java.util.concurrent.Callable`
  - `call()`
- `java.lang.Runnable`
  - `run()`
- `jakarta.enterprise.concurrent.ManagedTaskListener`
  - `taskAborted`
  - `taskSubmitted`
  - `taskStarting`
- `jakarta.enterprise.concurrent.ManagedTaskListener`
  - `taskAborted`
  - `taskSubmitted`
  - `taskStarting`
- `jakarta.enterprise.concurrent.Trigger`
  - `getNextRuntime()`
  - `skipRun()`

## 2. Managed Objects

본 장에서는 Jakarta Concurrency에서 제공하는 Managed Objects에 대해서 간략하게 설명하고 사용 예제를 기술한다.

### 2.1. ManagedExecutorService

jakarta.enterprise.concurrent.ManagedExecutorService 인터페이스는 Java SE의 java.util.concurrent.ExecutorService 인터페이스를 상속한다. ExecutorService와 동일하게 비동기 작업을 수행하기 위해 이용되고, 애플리케이션 서버는 비동기로 실행되는 작업의 컨텍스트 정보를 유지시켜준다.

#### 리소스 정의 예제

다음은 ManagedExecutorService를 리소스로 정의한 예제이다.

ManagedExecutorService를 리소스로 정의한 예제 : <domain.xml>

```
<domain>
  ...
  <server>
    <data-sources>
      <data-source>testdb</data-source>
    </data-sources>
    <managed-executor-service>mes1</managed-executor-service>
  </server>

  <resources>
    <managed-executor-service>
      <export-name>mes1</export-name>
      <long-running-task>true</long-running-task>
      <thread-pool>
        <min>10</min>
        <max>20</max>
        <keep-alive-time>60000</keep-alive-time>
        <queue-size>4096</queue-size>
        <stuck-thread-handling>
          <max-stuck-thread-time>3600000</max-stuck-thread-time>
          <action-on-stuck-thread>None</action-on-stuck-thread>
          <stuck-thread-check-period>300000</stuck-thread-check-period>
        </stuck-thread-handling>
      </thread-pool>
    </managed-executor-service>
  </resources>
  ...
</domain>
```

설정 항목에 대한 설명은 다음과 같다.

- 기본 항목

항목	설명
Export Name	Managed Executor Service를 JNDI Naming Server에 등록할 때 사용할 이름을 설정한다.
Long Running Task	Managed Executor Service에서 동작하는 task가 오래 수행되는지 여부이다. boolean 값으로 설정한다.

#### • Thread Pool

Managed Executor Service에서 사용하는 Thread Pool 설정을 나타낸다.

항목	설명
Min	Thread Pool에서 관리하는 스레드 수의 최솟값이다.
Max	Thread Pool에서 관리하는 스레드 수의 최댓값이다.
Keep Alive Time	설정된 시간 동안 사용하지 않은 Min 개수를 초과한 스레드를 자동적으로 Thread Pool에서 제거한다. 0으로 설정하는 경우 제거하지 않는다.
Queue Size	Thread Pool이 처리할 업무 개체를 저장하는 queue의 크기를 지정한다.

#### • Stuck Thread Handling

스레드가 특정 업무 때문에 일정 시간 계속 점유된 상태일 경우 해당 스레드에 대해 특정한 동작을 취하기 위한 설정이다.

항목	설명
Max Stuck Thread Time	스레드를 Stuck Thread로 판단하는 기준이 되는 값을 설정한다. 설정된 시간 이상 계속 점유된 상태이면 해당 스레드를 Stuck Thread로 간주한다.
Action On Stuck Thread	Stuck Thread로 판단된 경우 해당 스레드에 대해 취할 액션을 설정한다. 다음 옵션 중 하나를 선택할 수 있다. <ul style="list-style-type: none"> <li>• None: 아무 액션도 취하지 않는다.</li> <li>• Interrupt: java.lang.Thread#interrupt() 호출을 통해 Interrupt 신호를 보낸다.</li> <li>• IgnoreAndReplace: Stuck Thread를 무시하고 새로운 스레드로 교체한다. 이후 Stuck 상태에서 풀리면 무시한 스레드는 버려진다.</li> <li>• Warning: Log에 Warning과 함께 thread dump를 남긴다.</li> </ul>
Stuck Thread Check Period	Stuck Thread의 상태를 체크하는 주기를 설정한다.
User Warning Class	action-on-stuck-thread를 Warning으로 설정된 경우에 default는 thread dump를 찍어주도록 되어있으나, 사용자가 원하는 작업을 할 수 있도록 직접 클래스를 작성할 경우 이 설정을 사용한다. 해당 클래스는 jeus.util.pool.Warning을 반드시 implement해야 하며, jeus.util.pool.Warning interface는 jclient.jar에서 찾을 수 있다. 클래스를 작성한 후에 사용할 대상 서버의 SERVER_HOME/lib/application에 위치시킨다.



## 애플리케이션 예제

다음은 ManagedExecutorService를 활용한 애플리케이션 예제이다.

ManagedExecutorService를 활용한 애플리케이션 예제

```
public class AppServlet extends HttpServlet implements Servlet {
    // Retrieve our executor instance.
    @Resource(name="mes1")
    ManagedExecutorService mes;

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        ArrayList<Callable> builderTasks = new ArrayList<Callable>();
        builderTasks.add(new AccountTask(reqID, accountID));
        builderTasks.add(new InsuranceTask(reqID, accountID));

        // Submit the tasks and wait.
        List<Future<Object>> results = mes.invokeAll(builderTasks);

        AccountInfo accountInfo = (AccountInfo) results.get(0).get();
        InsuranceInfo insInfo = (InsuranceInfo) results.get(1).get();
        // Process the results
    }
}
```

## 2.2. ManagedScheduledExecutorService

jakarta.enterprise.concurrent.ManagedScheduledExecutorService 인터페이스는

ManagedExecutorService의 모든 기능을 상속받는 동시에 Java SE의

java.util.concurrent.ScheduledExecutorService의 기능을 상속받아 작업의 지연 실행이나 주기적 실행이 가능하도록 기능을 제공한다. 추가적으로 Trigger와 ManagedTaskListener 인터페이스를 통해 작업의 실행을 제어할 수 있도록 한다.

### 리소스 정의 예제

다음은 ManagedScheduledExecutorService를 리소스로 정의한 예제이다.

ManagedScheduledExecutorService를 리소스로 정의한 예제 : <domain.xml>

```
<domain>
  ...
  <server>
    <data-sources>
      <data-source>testdb</data-source>
    </data-sources>
    <managed-scheduled-executor-service>mses1</managed-scheduled-executor-service>
  </server>
  <resources>
    <managed-scheduled-executor-service>
      <export-name>mses1</export-name>
      <long-running-task>true</long-running-task>
      <thread-pool>
        <min>10</min>
```

```

<max>20</max>
<keep-alive-time>60000</keep-alive-time>
<queue-size>4096</queue-size>
<stuck-thread-handling>
  <max-stuck-thread-time>3600000</max-stuck-thread-time>
  <action-on-stuck-thread>None</action-on-stuck-thread>
  <stuck-thread-check-period>300000</stuck-thread-check-period>
</stuck-thread-handling>
</thread-pool>
</managed-scheduled-executor-service>
</resources>
...
</domain>

```

설정 항목에 대한 설명은 다음과 같다.

- 기본 항목

항목	설명
Export Name	Managed Scheduled Executor Service를 JNDI Naming Server에 등록할 때 사용할 이름을 설정한다.
Long Running Task	Managed Scheduled Executor Service에서 동작하는 task가 오래 수행되는지 여부이다. boolean 값으로 설정한다.

- Thread Pool

Managed Executor Service에서 사용하는 Thread Pool 설정을 나타낸다.

항목	설명
Min	Thread Pool에서 관리하는 스레드 수의 최솟값이다.
Max	Thread Pool에서 관리하는 스레드 수의 최댓값이다.
Keep Alive Time	설정된 시간 동안 사용되지 않은 Min 개수를 초과한 스레드를 자동적으로 Thread Pool에서 제거한다. 0으로 설정하면 스레드를 제거하지 않는다.
Queue Size	Thread Pool이 처리할 업무 개체를 저장하는 queue의 크기를 지정한다.

- Stuck Thread Handling

스레드가 특정 업무 때문에 일정 시간 계속 점유된 상태일 경우 해당 스레드에 대해 특정한 동작을 취하기 위한 설정이다.

항목	설명
Max Stuck Thread Time	스레드를 Stuck Thread로 판단하는 기준이 되는 값을 설정한다. 설정된 시간 이상 계속 점유된 상태이면 해당 스레드를 Stuck Thread로 간주한다.

항목	설명
Action On Stuck Thread	<p>Stuck Thread로 판단된 경우 해당 스레드에 대해 취할 액션을 설정한다. 다음 옵션 중 하나를 선택할 수 있다.</p> <ul style="list-style-type: none"> <li>• None: 아무 액션도 취하지 않는다.</li> <li>• Interrupt: java.lang.Thread#interrupt() 호출을 통해 Interrupt 신호를 보낸다.</li> <li>• IgnoreAndReplace: Stuck Thread를 무시하고 새로운 스레드로 교체한다. 이후 Stuck 상태에서 풀리면 무시한 스레드는 버려진다.</li> <li>• Warning: Log에 Warning과 함께 thread dump를 남긴다.</li> </ul>
Stuck Thread Check Period	Stuck Thread의 상태를 체크하는 주기를 설정한다.
User Warning Class	<p>action-on-stuck-thread를 Warning으로 설정된 경우에 default는 thread dump를 찍어주도록 되어있으나, 사용자가 원하는 작업을 할 수 있도록 직접 클래스를 작성할 경우 이 설정을 사용한다. 해당 클래스는 jeus.util.pool.Warning을 반드시 implement해야 하며, jeus.util.pool.Warning interface는 jclient.jar에서 찾을 수 있다. 클래스를 작성한 후에 사용할 대상 서버의 SERVER_HOME/lib/application에 위치시킨다.</p>

## 애플리케이션 예제

다음은 ManagedScheduledExecutorService를 활용한 애플리케이션 예제이다.

ManagedScheduledExecutorService를 활용한 애플리케이션 예제

```
public class AppServlet extends HttpServlet implements Servlet {
    @Resource(name="mses1")
    ManagedScheduledExecutorService mses;

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        Runnable printTask = new Runnable() {
            @Override
            public void run() {
                System.out.println(System.currentTimeMillis());
            }
        };
        // printTask가 5초마다 주기적으로 실행됨
        mses.schedule(printTask, 5, TimeUnit.SECONDS);
    }
}
```

## 2.3. ContextService

ExecutorService를 이용하지 않고 ContextService를 통해 Managed Task를 생성하는 방식을 제공한다. 본 기능을 이용하여 사용자는 태스크를 만들 때 컨텍스트에 대해서 신경쓰지 않아도 애플리케이션 서버 내부적으로 작업이 실행될 때 컨텍스트를 유지시켜준다. 좀더 상세하게는 Dynamic Proxy를 이용해서 현재 작업을 실행하기

전에 해당 스레드에 컨텍스트를 설정하고 작업을 수행한다. 모든 작업이 완료되면 컨텍스트를 복귀하는 작업을 대신 수행한다.

## 리소스 정의 예제

다음은 ContextService를 리소스로 정의한 예제이다.

ContextService를 리소스로 정의한 예제 : <domain.xml>

```
<domain>
  ...
  <server>
    <data-sources>
      <data-source>testdb</data-source>
    </data-sources>
    <context-service>cs1</context-service>
  </server>

  <resources>
    <context-service>
      <export-name>cs1</export-name>
    </context-service>
  </resources>
  ...
</domain>
```

설정 항목에 대한 설명은 다음과 같다.

- 기본 항목

항목	설명
Export Name	Context Service를 JNDI Naming Server에 등록할 때 사용할 이름을 설정한다.

## 애플리케이션 예제

다음은 ContextService를 활용한 애플리케이션 예제이다.

ContextService를 활용한 애플리케이션 예제

```
public class AppServlet extends HttpServlet implements Servlet {
    @Resource(name="cs1")
    ContextService cs;

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // 일반적인 Runnable task
        Runnable simpleTask = new Runnable() {
            @Override
            public void run() {
                int sum = 0;
                for (int i = 0; i < 10; i++) { sum += i; }
                System.out.println(sum);
            }
        };
    }
};
```

```

// 일반 task를 ContextService를 통해서 Contextual task로 만듦
cs.createContextualProxy(simpleTask, Runnable.class);

// Java SE에서 제공하는 Executor에 Contextual task를 넘김
ExecutorService es = Executors.newFixedThreadPool(1);
es.submit(simpleTask);
}
}

```

## 2.4. ManagedThreadFactory

jakarta.enterprise.concurrent.ManagedThreadFactory 인터페이스는 Java SE의 java.util.concurrent.ThreadFactory 기능을 상속받아 스레드를 생성하는 기능을 제공한다. 일반적으로 ThreadFactory는 ThreadPoolExecutor를 생성할 때 생성자의 파라미터로 넘기는 곳에 사용한다. 따라서, Java SE의 Concurrency API 상에서도 worker 스레드가 작업을 실행할 때 Task에 대한 컨텍스트를 유지해줄 수 있다.

### 리소스 정의 예제

다음은 ManagedThreadFactory를 리소스로 정의한 예제이다.

ManagedThreadFactory를 리소스로 정의 예제 : <domain.xml>

```

<domain>
  ...
  <server>
    <data-sources>
      <data-source>testdb</data-source>
    </data-sources>
    <managed-thread-factory>mtf1</managed-thread-factory>
  </server>

  <resources>
    <managed-thread-factory>
      <export-name>mtf1</export-name>
      <thread-priority>5</thread-priority>
    </managed-thread-factory>
  </resources>
  ...
</domain>

```

설정 항목에 대한 설명은 다음과 같다.

- 기본 항목

항목	설명
Export Name	Managed Thread Factory를 JNDI Naming Server에 등록할 때 사용할 이름을 설정한다.
Thread Priority	스레드의 우선순위를 설정한다. (기본값: 5)

## 애플리케이션 예제

다음은 `ManagedThreadFactory`를 활용한 애플리케이션 예제이다.

`ManagedThreadFactory`를 활용한 애플리케이션 예제

```
public class AppServlet extends HttpServlet implements Servlet {
    // Retrieve our executor instance.
    @Resource(name="mtf1")
    ManagedThreadFactory mtf;

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // 일반적인 Runnable task
        Runnable simpleTask = new Runnable() {
            @Override
            public void run() {
                int sum = 0;
                for (int i = 0; i < 10; i++) { sum += i; }
                System.out.println(sum);
            }
        };

        // simpleTask를 ManagedThreadFactory에서 제공되는 thread에서 실행
        mtf.newThread(simpleTask).start();

        // 혹은 ThreadPoolExecutor의 매개변수로 ThreadFactory를 전달
        Executor e = new ThreadPoolExecutor(5, 10, 6L, TimeUnit.MINUTES,
            new ArrayBlockingQueue<Runnable>(4096), mtf);
        e.execute(new SimpleTask());
    }
}
```