提纲

FailureHandler

FailureHandler是一个接口类，它提供了onFailure接口，用于处理ignite实例发生failure的情况。

```
public interface FailureHandler {
    /**
     * Handles failure occurred on {@code ignite} instance.
     * Failure details is contained in {@code failureCtx}.
     * Returns {@code true} if kernal context must be invalidated by {@link FailureProcessor}
after calling this method.
     *
     * @param ignite Ignite instance.
     * @param failureCtx Failure context.
     * @return Whether kernal context must be invalidated or not.
     */
    public boolean onFailure(Ignite ignite, FailureContext failureCtx);
}
```

AbstractFailureHandler

AbstractFailureHandler中排除一些当前FailureHandler可以忽略的FailureType。默认的排除SYSTEM_WORKER_BLOCKED和SYSTEM_CRITICAL_OPERATION_TIMEOUT。

AbstractFailureHandler提供了一个抽象的handle方法，供子类去实现具体的针对failure的处理逻辑。

AbstractFailureHandler提供了默认的onFailure的实现，首先检查待处理的FailureType是否是被忽略的FailureTypes集合中的，如果不是则调用子类的handle方法进行处理。

```
public abstract class AbstractFailureHandler implements FailureHandler {
    # 对于ignoredFailureTypes中的FailureType直接忽略，其中SYSTEM_WORKER_BLOCKED
    # 由WorkerRegistry负责处理，SYSTEM_CRITICAL_OPERATION_TIMEOUT不被处理
    @GridToStringInclude
    private Set<FailureType> ignoredFailureTypes =
            Collections.unmodifiableSet(EnumSet.of(SYSTEM_WORKER_BLOCKED,
SYSTEM_CRITICAL_OPERATION_TIMEOUT));

    /**
     * Sets failure types that must be ignored by failure handler.
     *
     * @param failureTypes Set of failure type that must be ignored.
     * @see FailureType
     */
    public void setIgnoredFailureTypes(Set<FailureType> failureTypes) {
        ignoredFailureTypes = Collections.unmodifiableSet(failureTypes);
    }

    /**
     * Returns unmodifiable set of ignored failure types.
     */
    public Set<FailureType> getIgnoredFailureTypes() {
        return ignoredFailureTypes;
    }

    # 当发生故障时，首先排除在ignoredFailureTypes中的FailureType，然后调用相应的处理逻辑
    @Override public boolean onFailure(Ignite ignite, FailureContext failureCtx) {
        return !ignoredFailureTypes.contains(failureCtx.type()) && handle(ignite, failureCtx);
    }

    protected abstract boolean handle(Ignite ignite, FailureContext failureCtx);
}
```

ignite中定义的FailureType

Ignite中定义了FailureType集合：

```
public enum FailureType {
    /** Node segmentation. */
    SEGMENTATION,

    /** System worker termination. */
    SYSTEM_WORKER_TERMINATION,

    /** System worker has not updated its heartbeat for a long time. */
    SYSTEM_WORKER_BLOCKED,

    /** Critical error - error which leads to the system's inoperability. */
    CRITICAL_ERROR,

    /** System-critical operation has been timed out. */
    SYSTEM_CRITICAL_OPERATION_TIMEOUT
}
```

AbstractFailureHandler定义的可忽略的FailureType

AbstractFailureHandler中定义了默认可以忽略的failure type为：SYSTEM_WORKER_BLOCKED和SYSTEM_CRITICAL_OPERATION_TIMEOUT。

自定义可以忽略的FailureType

AbstractFailureHandler中提供了setIgnoredFailureTypes接口去设置可以忽略的FailureType集合。

```
public abstract class AbstractFailureHandler implements FailureHandler {
    @GridToStringInclude
    private Set<FailureType> ignoredFailureTypes =
            Collections.unmodifiableSet(EnumSet.of(SYSTEM_WORKER_BLOCKED,
SYSTEM_CRITICAL_OPERATION_TIMEOUT));

    /**
     * Sets failure types that must be ignored by failure handler.
     *
     * @param failureTypes Set of failure type that must be ignored.
     * @see FailureType
     */
    public void setIgnoredFailureTypes(Set<FailureType> failureTypes) {
        ignoredFailureTypes = Collections.unmodifiableSet(failureTypes);
    }

    ...
}
```

StopNodeOrHaltFailureHandler

StopNodeOrHaltFailureHandler是系统内置的FailureHandler，它通过继承AbstractFailureHandler类，实现了FailureHandler接口，因为AbstractFailureHandler实现了FailureHandler接口。如果tryStop设置为true，则会首先尝试在timeout给定的时间内停止ignite，如果在超时时间内未停止，则调用Runtime.getRuntime().halt，如果tryStop设置为false，则直接调用Runtime.getRuntime().halt。

```
public class StopNodeOrHaltFailureHandler extends AbstractFailureHandler {
    # 故障时，是否stop节点
    private final boolean tryStop;

    # stop超时时间
    private final long timeout;

    # 默认构造方法中tryStop设置为false，则故障时，尝试调用Runtime.getRuntime().halt()
    public StopNodeOrHaltFailureHandler() {
        this(false, 0);
    }

    public StopNodeOrHaltFailureHandler(boolean tryStop, long timeout) {
        this.tryStop = tryStop;
        this.timeout = timeout;
    }

    # failure处理逻辑
    @Override protected boolean handle(Ignite ignite, FailureContext failureCtx) {
        IgniteLogger log = ignite.log();

        if (tryStop) {
            # 设置了tryStop，则首先尝试停止ignite
            # 创建一个CountDownLatch，用于控制stop过程
            final CountDownLatch latch = new CountDownLatch(1);

            new Thread(
                new Runnable() {
                    @Override public void run() {
                        U.error(log, "Stopping local node on Ignite failure: [failureCtx=" +
failureCtx + ']');

                        IgnitionEx.stop(ignite.name(), true, true);

                        # stop完成，则释放栓锁
                        latch.countDown();
                    }
                },
                "node-stopper"
            ).start();

            new Thread(
                new Runnable() {
                    @Override public void run() {
                        try {
                            # 等待timeout的时间，如果还没有stop完成，则halt
                            if (!latch.await(timeout, TimeUnit.MILLISECONDS)) {
                                U.error(log, "Stopping local node timeout, JVM will be halted.");

                                Runtime.getRuntime().halt(Ignition.KILL_EXIT_CODE);
                            }
                        }
                        catch (InterruptedException e) {
                            // No-op.
                        }
                    }
                },
                "jvm-halt-on-stop-timeout"
            ).start();
        }
```

```
        else {
            # 否则，直接halt
            U.error(log, "JVM will be halted immediately due to the failure: [failureCtx=" +
failureCtx + ']');

            Runtime.getRuntime().halt(Ignition.KILL_EXIT_CODE);
        }

        return true;
    }

    public long timeout() {
        return timeout;
    }

    public boolean tryStop() {
        return tryStop;
    }
}
```

StopNodeFailureHandler

```
# 直接stop当前的节点
public class StopNodeFailureHandler extends AbstractFailureHandler {
    @Override public boolean handle(Ignite ignite, FailureContext failureCtx) {
        # 新建并启动一个名为node-stopper的线程，在该线程中直接调用IgnitionEx.stop来停止ignite
        new Thread(
            new Runnable() {
                @Override public void run() {
                    U.error(ignite.log(), "Stopping local node on Ignite failure: [failureCtx=" +
failureCtx + ']');

                    # 停止节点
                    IgnitionEx.stop(ignite.name(), true, true);
                }
            },
            "node-stopper"
        ).start();

        return true;
    }
}
```

Ignite中对不同的FailureType的处理

Ignite中支持以下的FailureType：

```
public enum FailureType {
    /** Node segmentation. */
    SEGMENTATION,

    /** System worker termination. */
    SYSTEM_WORKER_TERMINATION,

    /** System worker has not updated its heartbeat for a long time. */
    SYSTEM_WORKER_BLOCKED,

    /** Critical error - error which leads to the system's inoperability. */
    CRITICAL_ERROR,

    /** System-critical operation has been timed out. */
    SYSTEM_CRITICAL_OPERATION_TIMEOUT
}
```

Ignite中默认不处理SYSTEM_WORKER_BLOCKED和SYSTEM_CRITICAL_OPERATION_TIMEOUT

Ignite中默认忽略SYSTEM_WORKER_BLOCKED和SYSTEM_CRITICAL_OPERATION_TIMEOUT这两种FailureType，当这两种FailureType发生时，是不会被处理的。

Ignite中对SEGMENTATION的处理

在DiscoveryWorker中处理SEGMENTATION，如果SegmentationPolicy设置为RESTART_JVM，则采用RestartProcessFailureHandler进行处理，如果SegmentationPolicy设置为STOP，则采用StopNodeFailureHandler进行处理，否则SegmentationPolicy被设置为NOOP，则什么都不做。

```
public class GridDiscoveryManager extends GridManagerAdapter<DiscoverySpi> {
    private class DiscoveryWorker extends GridWorker {
        ...

        private void onSegmentation() {
            SegmentationPolicy segPlc = ctx.config().getSegmentationPolicy();

            // Always disconnect first.
            try {
                getSpi().disconnect();
            }
            catch (IgniteSpiException e) {
                U.error(log, "Failed to disconnect discovery SPI.", e);
            }

            switch (segPlc) {
                case RESTART_JVM:
                    ctx.failure().process(new FailureContext(FailureType.SEGMENTATION, null),
restartProcHnd);

                    break;

                case STOP:
                    ctx.failure().process(new FailureContext(FailureType.SEGMENTATION, null),
stopNodeHnd);

                    break;

                default:
                    assert segPlc == NOOP : "Unsupported segmentation policy value: " + segPlc;
            }
        }
    }
}
```

Ignite中对SYSTEM_WORKER_TERMINATION的处理

从代码中可以看到，很多组件中都会涉及对SYSTEM_WORKER_TERMINATION的处理，但是所有的地方的处理逻辑都是类似下面的代码：

```
# 其中ctx类型为GridKernalContext
ctx.failure().process(new FailureContext(SYSTEM_WORKER_TERMINATION, e));
```

GridKernalContext.failure()获得的是GridFailureProcessor，GridFailureProcessor.process()实际上会调用系统中配置的FailureHandler来进行处理，默认情况下使用的是StopNodeOrHaltFailureHandler进行处理。

Ignite中对CRITICAL_ERROR的处理

从代码中可以看到，很多组件中都会涉及对SYSTEM_WORKER_TERMINATION的处理，但是所有的地方的处理逻辑都是类似下面的代码：

```
# 其中ctx类型为GridKernalContext
ctx.failure().process(new FailureContext(CRITICAL_ERROR, err));
```

GridKernalContext.failure()获得的是GridFailureProcessor，GridFailureProcessor.process()实际上会调用系统中配置的FailureHandler来进行处理，默认情况下使用的是StopNodeOrHaltFailureHandler进行处理。