Chapter6 - CoroutineContext

CoroutineContext는 코루틴을 실행하는 실행 환경을 설정하고 관리하는 인터페이스이다.

6.1. CoroutineContext의 구성 요소

다양한 CoroutineContext 구성 요소가 있지만 주로 네 가지 구성 요소를 사용해 코루틴을 실행시킨다.

1. CoroutineName: 코루틴의 이름 설정
2. CoroutineDispatcher: 코루틴을 스레드에 할당해 실행한다.
3. Job: 코루틴의 추상체로 코루틴을 제어하는 데 사용
4. CoroutineExceptionHandler: 코루틴에서 발생한 예외를 처리한다.

6.2. CoroutineContext 구성하기

coroutineContext

키	값
CoroutineName 키	CoroutineName 객체
CoroutineDispatcher 키	CoroutineDispatcher 객체
Job 키	Job 객체
CoroutineExceptionHandler 키	CoroutineExceptionHandler 객체

그림 6-1 CoroutineContext의 구성 1

CoroutineContext 객체는 키-값 쌍으로 각 구성 요소를 관리한다. 각 구성 요소는 고유한 키를 가지며, 키에 대해 중복된 값은 허용되지 않는다. 따라서 CoroutineContext 객체는 CoroutineName, CoroutineDispatcher, Job, CoroutineExceptionHandler 객체를 한 개씩만 가질 수 있다.

```
/**
 * Persistent context for the coroutine. It is an indexed set of [Element]
instances.
 * An indexed set is a mix between a set and a map.
 * Every element in this set has a unique [Key].
 */
@SinceKotlin("1.3")
public interface CoroutineContext
```

CoroutineContext 주석을 확인해보면, set과 map이 섞인 indexed set이라고 하는데 이 자료구조가 어떤 건지 잘 모르겠다. 하지만 중복된 값을 허용하지 않는 set의 특성과 키-값 쌍으로 데이터를 관리하는 map의 특성이 섞인 게 아닌가 싶다.

CoroutineContext 구성 요소 덮어씌우기

만약 CoroutineContext 객체에 같은 구성 요소가 둘 이상 더해진다면 나중에 추가된 CoroutineContext 구성 요소가 이전의 값을 덮어씌운다.

```
fun main() = runBlocking<Unit> {
  val coroutineContext: CoroutineContext =
newSingleThreadContext("MyThread") + CoroutineName("MyCoroutine")
  val newCoroutineContext: CoroutineContext = coroutineContext +
CoroutineName("NewCoroutine")

  launch(context = newCoroutineContext) {
    println("[${Thread.currentThread().name}] 실행")
  }
}
/*
// 결과:
[MyThread @NewCoroutine#2] 실행
*/
```

코드의 실행 결과를 보면 스레드는 MyThread로 유지되지만 코루틴의 이름은 NewCoroutine으로 변경된 것을 확인할 수 있다.

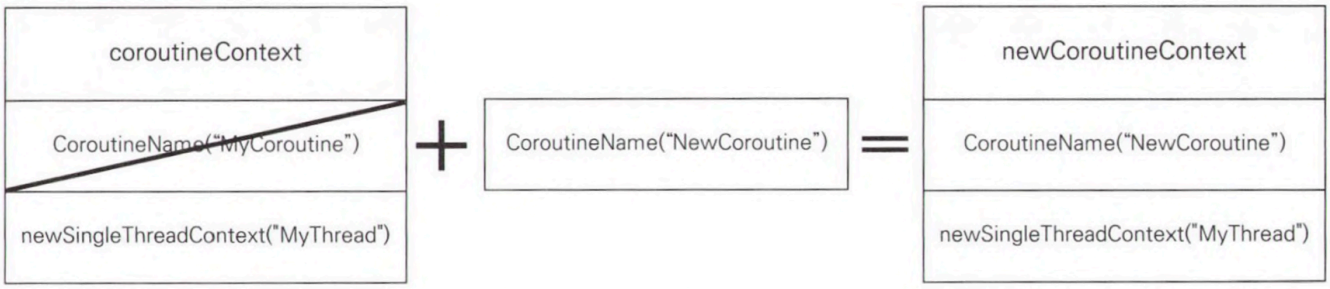

그림 6-3 CoroutineContext의 더하기 연산자가 동작하는 방식

```
val coroutineContext1 = CoroutineName("MyCoroutine1") +
newSingleThreadContext("MyThread1")
val coroutineContext2 = CoroutineName("MyCoroutine2") +
newSingleThreadContext("MyThread2")
val combinedCoroutineContext = coroutineContext1 + coroutineContext2
```

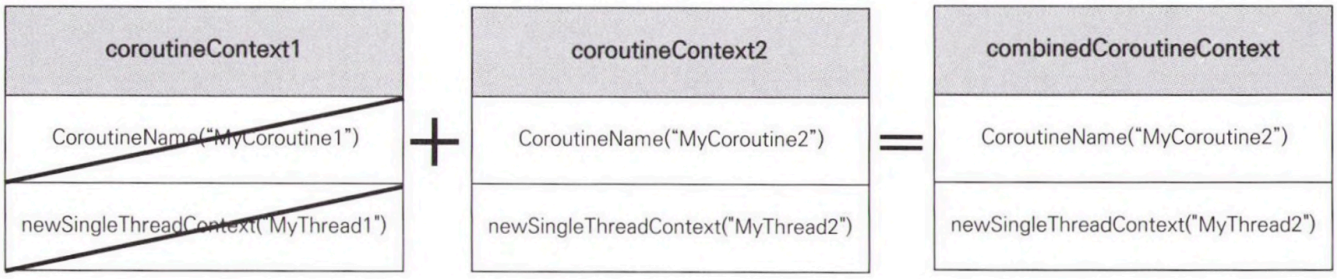

그림 6-4 combinedCoroutineContext

여러 구성 요소로 이루어진 CoroutineContext를 합칠 때도 동일한 원리로 나중에 추가된 구성 요소로 덮어씌워진다.

6.3. CoroutineContext 구성 요소에 접근하기

CoroutineContext 객체의 구성 요소에 접근하기 위해서는 각 구성 요소가 가진 고유한 키가 필요하다. 구성 요소의 키는 CoroutineContex.Key 인터페이스를 구현해 만들 수 있는데 일바적으로 CoroutineContext 구성 요소는 자신의 내부에 키를 싱글톤 객체로 구현한다.

```
public data class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {
    public companion object Key : CoroutineContext.Key<CoroutineName>
    ...
}

public interface Job : CoroutineContext.Element {
    public companion object Key : CoroutineContext.Key<Job>
    ...
```

```
}

public abstract class CoroutineDispatcher :
    AbstractCoroutineContextElement(ContinuationInterceptor),
ContinuationInterceptor {

    /** @suppress */
    @ExperimentalStdlibApi
    public companion object Key :
AbstractCoroutineContextKey<ContinuationInterceptor, CoroutineDispatcher>(
        ContinuationInterceptor,
        { it as? CoroutineDispatcher })
    ...
}

public interface CoroutineExceptionHandler : CoroutineContext.Element {
    public companion object Key :
CoroutineContext.Key<CoroutineExceptionHandler>
    ...
}
```

키를 사용해 CoroutineContext 구성 요소에 접근하기

```
fun main() = runBlocking<Unit> {
  val coroutineContext = CoroutineName("MyCoroutine") + Dispatchers.IO
  val nameFromContext = coroutineContext[CoroutineName.Key]
  println(nameFromContext)
}
/*
// 결과:
CoroutineName(MyCoroutine)
*/
```

싱글톤 객체로 선언된 Key를 사용하여 구성 요소에 접근 가능하다.

```
fun main() = runBlocking<Unit> {
  val coroutineContext = CoroutineName("MyCoroutine") + Dispatchers.IO
  val nameFromContext = coroutineContext[CoroutineName] // '.Key'제거
  println(nameFromContext)
}
/*
// 결과:
CoroutineName(MyCoroutine)
*/
```

구성 요소의 companion object(동반 객체)로 선언되어 있기 때문에 Key를 명시적으로 사용하지 않고 구성 요소 자체를 키로 사용 가능하다.

```
fun main() = runBlocking<Unit> {
  val coroutineName : CoroutineName = CoroutineName("MyCoroutine")
  val dispatcher : CoroutineDispatcher = Dispatchers.IO
  val coroutineContext = coroutineName + dispatcher

  println(coroutineContext[coroutineName.key]) //
CoroutineName("MyCoroutine")
  println(coroutineContext[dispatcher.key]) // Dispatchers.IO
}
/*
// 결과:
CoroutineName(MyCoroutine)
Dispatchers.IO
*/
```

```
public interface Element : CoroutineContext {
    /**
     * A key of this coroutine context element.
     */
    public val key: Key<*>
    ...
}
```

CoroutineContext 구성 요소들은 모두 key 프로퍼티를 가지기 때문에 이를 사용해 구성 요소에 접근 가능하다.

6.4. CoroutineContext 구성 요소 제거하기

CoroutineContext 객체는 구성 요소를 제거하기 위한 minusKey 함수를 제공한다. minusKey 함수는 구성 요소의 키를 인자로 받아 해당 구성 요소를 제거한 CoroutineContext 객체를 반환한다.

```
/**
 * Returns a context containing elements from this context, but without an
element with
 * the specified [key].
 */
public fun minusKey(key: Key<*>): CoroutineContext
```

```
@OptIn(ExperimentalStdlibApi::class)
fun main() = runBlocking<Unit> {
```

```
    val coroutineName = CoroutineName("MyCoroutine")
    val dispatcher = Dispatchers.IO
    val myJob = Job()
    val coroutineContext: CoroutineContext = coroutineName + dispatcher +
myJob

    val deletedCoroutineContext = coroutineContext.minusKey(CoroutineName)

    println(deletedCoroutineContext[CoroutineName])
    println(deletedCoroutineContext[CoroutineDispatcher])
    println(deletedCoroutineContext[Job])
}
/*
// 결과:
null
Dispatchers.IO
JobImpl{Active}@65e2dbf3
*/
```

deletedCoroutineContext

키	값
CoroutineName 키	설정되지 않음
CoroutineDispatcher 키	Dispatchers.IO
Job 키	myJob

그림 6-8 CoroutineContext 구성 요소 제거하기

minusKey 함수 사용 시, minusKey를 호출한 CoroutineContext 객체는 그대로 유지되고, 구성 요소가 제거된 새로운 CoroutineContext 객체가 반환된다는 점을 주의해야 한다.