Compositional Caching

Agenda

- **Why Do We Need A Caching Solution?**
- What Would A Caching Solution Look Like?
- Introducing ZIO Cache

My Interest In Caching

- I got interested in caching because of my work on ZIO Query
- ZIO Query provides automatic pipelining, batching, and **caching** of queries
- Need to have a cache to do caching!

Caching For Compositionality

- If we want our applications to be **compositional** different parts of our application may do overlapping work
- We can refactor our code to try to avoid this but that can make our code harder to understand and maintain
- Caching allows us to avoid this tradeoff

Caching For Performance

- Sometimes we may receive requests to do overlapping work
- If we want our applications to be **performant** we must do this work at most once
- Caching allows us to accomplish this in the face of uncertainty about the requests we will receive

There Is Not A Great Caching Solution For The ZIO Ecosystem Today

- No ZIO native solution
- Limited support for asynchronous code
- Restricted options for caching policies

A Simple Example

```
def effect(key: String): ZIO[Clock with Console, Nothing, String] =
  console.putStrLn("Start") *>
    ZIO.sleep(5.seconds) *>
    console.putStrLn("Done") *>
    ZIO.succeed(s"$key -> Value")

effect.zipPar(effect).flatMap(values => console.putStrLn(values.toString))
```

- Assume that the two concurrent invocations of `effect` occur in different parts of our program
- Implement caching such that `effect` is only executed a single time

ScalaCache

```scala
def memoize[R, K, V](
  key: K
)(f: K => ZIO[R, Throwable, V])(implicit cache: Cache[V]): ZIO[R, Throwable, V] =
    ZIO.runtime[R].flatMap { runtime =>
      ZIO.fromFuture { implicit ec =>
        cachingF[Future, V](key)(None)(runtime.unsafeRunToFuture(f(key)))
      }
    }

memoize("Key")(effect).zipPar(memoize("Key")(effect)).flatMap { values =>
  console.putStrLn(values.toString)
}
```

Caffeine

```
val cache: AsyncLoadingCache[String, String] =
  Caffeine
    .newBuilder()
    .maximumSize(10000L)
    .buildAsync((key, _) =>
      runtime.unsafeRun(effect(key).toCompletableFuture)
    )

  def lookup(key: String): ZIO[Any, Throwable, String] =
    ZIO.fromCompletionStage(cache.get(key))

  lookup("Key").zipPar(lookup("Key")).flatMap { values =>
    console.putStrLn(values.toString)
  }
```

What's Wrong With This?

- Have to unsafely run our effect to a `Future`, losing power of ZIO around features such as interruption
- Loss of type information due to `Future` being able to fail with any `Throwable`
- ScalaCache example doesn't actually prevent effect from being evaluated twice since key is not added to cache until effect completes!

Doing It Ourselves

- We can implement our own solution using a data structure such as a `Ref[Map[K, Promise[E, V]]]`
- But there is a lot we have to get right here, for example removing failed promises from the map
- And what about everything else we expect from a cache like expiration policies and metrics, not to mention performance!
- We were just looking for a basic solution to a common concern and we are having to do a lot of low level work ourselves!

Agenda

- Why Do We Need A Caching Solution?
- **What Would A Caching Solution Look Like?**
- Introducing ZIO Cache

Signature Of A Cache

```
trait Cache[-K, +E, +V] {
  def get(key: K): IO[E, V]
}

object Cache {
  def make[K, R, E, V](
    capacity: Int
    lookup: Lookup[K, R, E, V],
    policy: CachingPolicy[V]
  ): ZIO[R, Nothing, Cache[K, E, V]] =
    ???
}
```

Cache Is Defined In Terms Of A Lookup Function

```
type Lookup[-K, -R, +E, +V] = K => ZIO[R, E, V]
```

- If value already exists in the cache, return that value
- Otherwise, compute its value using the lookup function and return it

Unification Of Synchronous And Asynchronous Caches

- Lookup function can compute value either synchronously or asynchronously
- Either way, key will immediately be added to the cache
- Concurrent lookups will suspend until the value being computed is available

Caching Policy Determines When Values Are Removed From The Cache

```
final case class CachingPolicy[-V](priority: Priority[V], evict: Evict[V])
```

Caching policy has two parts:

1. **Priority** - in what order **may** we remove values if we need to make room in the cache?
2. **Evict** - when **must** we remove values because they are no longer valid?

Optional Removal

```
sealed abstract class Priority[-V] {
  def compare(left: Entry[V], right: Entry[V]): Int
}
```

- Like an `Ordering` specialized for cache entries and with variance
- An `Entry[V]` includes both the value and statistics about the entry, such as the last time it was accessed
- Allows implementing policies such as prioritizing entries by last access

Mandatory Removal

```scala
final case class Evict[-Value](evict: (Instant, Entry[Value]) => Boolean)
```

- A function that determines whether an entry is valid based on the entry and the current time
- Allows implementing policies such as that an entry must not be more than one hour old

Caching Policy Forms A Total Ordering

- A valid entry is worth more than an invalid one and otherwise the one with higher priority is worth more
- Like ordering by `Evict` and then `Priority`
- Allows combining caching policies in a principled way

Caching Policies Compose

```
val evict =
  Evict.olderThan(Duration.ofHours(1L)) &&
    Evict.sizeGreaterThan(100 * 1024 * 1024)

val policy =
  byLastAccess ++ bySize ++ fromEvict(evict)
```

Agenda

- Why Do We Need A Caching Solution?
- What Would A Caching Solution Look Like?
- **Introducing ZIO Cache**

ZIO Cache

```
for {
  cache  <- Cache.make(10000, CachingPolicy.byLastAccess, Lookup(effect))
  values <- cache.get("Key").zipPar(cache.get("Key"))
  _      <- console.putStrLn(values.toString)
} yield ()
```

Key Features

- ZIO native caching solution
- Unification of synchronous and asynchronous APIs
- Composable caching policies
- Cache statistics

Cache Statistics

- Entries
- Memory size
- Hits
- Misses
- Loads
- Evictions
- Total load time

Next Steps

- Open for external contributions
- Excited to get your feedback
- Much more focus on performance!

Conclusion

- ZIO allows unifying across synchronous and asynchronous APIs
- Avoid unnecessarily translating between effect systems
- Composition for the win!

Thank You

- John de Goes for his leadership on this project
- Other contributors
- Sandra for all her work organizing this conference
- Generous sponsors who made this possible
- All of you for attending today