The Twitter Service Stack
or: How I Learned to stop Worrying and Love Functional Programming

Marius Eriksen - @marius
BASE 2012/04/09
Agenda

Motivation/background
5-minute Scala intro
Composable futures
Services
Finagle
Conclusion
Twitter infrastructure

Heavy users of open source projects

MySQL, memcached, redis, ruby(on-rails), Hadoop, ZooKeeper, Lucene, etc. etc.

Contribute back. Goal: Twitter agnostic infrastructure is open sourced.

github.com/twitter

A heterogenous world

Languages, protocols, libraries, runtimes
serv·ice stack ˈsərvis stak|

Software infrastructure, for use in services, that is application agnostic.

Comprises a bunch of libraries, loosely coupled.

I’m going to be talking (mostly) about our RPC system.
An RPC system?

Open source = heterogeneity
Antimodular

  Thrift, memcache, mysql, zookeeper share no code, or even interfaces.

Difficult to:

  Reuse code, reason about behavior and performance, address common concerns, observe the system
In fact,

[I claim that] The majority of the difficult problems in RPC systems are entirely protocol agnostic.
Starting from scratch

The JVM

- Hotspot, great GC, instrumentation, experience, support for high level languages
- Must support any protocol
  - HTTP, Thrift, memcache, redis, MySQL
- Must be productive
- Must be performant
- Highly concurrent services
A (short) detour to Scala
Scala elevator pitch

First released in 2003 (Odersky at EPFL)
Runs primarily on the JVM
Mixes OO and FP
Strong static typing
Expressive, concise
Interoperates naturally with Java
Simple in syntax, but with some corners of semantic complexity
Has a REPL
Static typing

```scala
val m: Map[Int, String] = Map[Int, String](
  1 -> "one", 2 -> "two")
```
.. inferred

val m = Map(1 -> "one", 2 -> "two")
Function1<Integer, Integer> f = new Function1<Integer, Integer, Integer>(){
    public Integer apply(Integer x) {
        return x*2
    }
}
val f = new Function1[Int, Int] {
  def apply(x: Int) = x*2
}

function.scala
Lightweight syntax!

```
val f = { x => x*2 }
```
Lightweight syntax!!

```scala
val f = _*2
```
def schedule(f: () => Unit)

schedule({ () =>
    println("hello, world!")
})
Lightweight syntax

def schedule(f: () => Unit)

    schedule { () =>
        println("hello, world!")
    }

Tuesday, April 10, 12
def manytimes(n: Int)(f: => Unit) {
  (0 until n) foreach { _ => f }
}

manytimes(100) {
  println("ok")
}
A pattern

```scala
val f: Int => String = { ... 
val g: String => Float = { ... 
val h: Int => Float =
  g compose f  // g.compose(f)
```
A pattern

val l = List(1,2,3,4)
val l2 =
  l map { elem => elem*2 }
// l.map({elem => elem*2})

l  == List(1,2,3,4)
l2 == List(2,4,6,8)
flatMap[B](f: A⇒List[B]): List[B]
flatMap

Stream.from(1) flatMap { z =>
    Stream.range(1, z) flatMap { x =>
        Stream.range(x, z) flatMap { y =>
            if (x*x + y*y == z*z) Stream((x, y, z))
            else Stream()
        }
    }
}


for-comprehensions

for {
    z <- Stream.from(1)
    x <- Stream.range(1, z)
    y <- Stream.range(x, z)
    if x*x + y*y == z*z
} yield (x, y, z)
Futures
Futures

A deferred value.

A long computation

An RPC

Reading from disk

Computations can fail!

Connection failure

Computation takes too long

Arithmetic exception
Using futures

```scala
val f: Future[String]

// wait indefinitely
val result = f.get()

// wait 1 second
val result = f.get(1.second)
```
Callbacks

val f: Future[String]

f onSuccess { s =>
  println("got string \"\"+s\")
}
onFailure { exc =>
  exc.printStackTrace()
}
Making futures

Values

Future.value[A](a: A): Future[A]
Future.exception[A](e: Throwable): Future[A]

A promise is a writable future

val p = new Promise[Int]
val f: Future[Int] = p

p.setValue(1)
p.setException(new Exception)
val p = new Promise[Int]
p onSuccess { i =>
  println("value: %d".format(i))
}
println("setting the promise")
p.setValue(1)
Callbacks

```scala
val p = new Promise[Int]
p onSuccess { i =>
  println("value: \%d".format(i))
}
println("setting the promise")
p.setValue(1)
```
val p = new Promise[Int]

p onSuccess { i =>
    println("value: %d".format(i))
}

println("setting the promise")
p.setValue(1)
Callbacks

val p = new Promise[Int]
p onSuccess { i =>
  println("value: %d".format(i))
}
println("setting the promise")
p.setValue(1)
val p = new Promise[Int]
p onSuccess { i =>
  println("value: %d".format(i))
}
println("setting the promise")
p.setValue(1)
val p = new Promise[Int]
p onSuccess { i =>
  println("value: %d\n".format(i))
}
println("setting the promise")
p.setValue(1)
Futures are like Lists!

List(1,2,3,4) map { i => i * 2 }  
// == List(2,4,6,8)

Future.value(1) map { i => i * 2}  
// == Future.value(2)
Futures are like Lists!

List(1,2,3,4) flatMap { i => List(i*2) }
  // == List(2,4,6,8)

Future.value(1) flatMap { i =>
  Future.value(i*2)
}  // == Future.value(2)
def auth(token: String): Future[Long]
def getUser(id: Long) : Future[User]

def getUser(token: String): Future[User]
flatMap

def getUser(token: String): Future[User] =
  auth(token) flatMap { id =>
    getUser(id)
  }
Exceptions

Analogous to Scala/Java exceptions
an exception stops a computation
computation can be recovered by the first handler able to do so

Scope

Scala/Java exceptions: stack frames
Futures: transformed “stack”

map, flatMap :: handle, rescue
Exceptions

val f = lookupUser(123) handle {
  case UserNotFound => NullUserObject
}

Tuesday, April 10, 12
Exceptions

``` scala
val f = lookupUser(123) rescue {
  case ServiceUnavailable =>
    lookupUser(123)  // try again
}
```
Functional style

Declarative **transformations** of data

Evaluation strategy is up to implementation

You don’t prescribe control flow, instead you declare the meaning of a value:

auth is the result of looking up a user, then authenticating the user object, recovering UserLookupError into the NullUserObject
val ops: Seq[Future[Res]] = Seq(rpc(), rpc(), rpc())
val res: Future[Seq[Res]] = Future.collect(ops)
val f1: Future[A]
val f2: Future[B]

val f: Future[Unit] =
    Future.join(f1, f2)

f onSuccess { _ =>
    println("everything's done")
}
val a: Future[A]
val b: Future[B]

val c: Future[(A, B)] = a join b
Works as advertised:

```scala
def get(): Future[Item]
def continue(item: Item): Boolean

def collect(items: List[Item]): Future[List[Item]] =
  getItem() flatMap { item =>
    if (continue(item))
      collect(item :: items)
    else
      Future.value(item :: items)
  }

val items: Future[List[Item]] = collect(Nil)
```
trait User {
    def follow(other: User)
    def DM(message: String)
}

def newUser(): Future[User]

val done: Future[Unit] = for {
    (a, b) <- newUser() join newUser()
    _     <- a.follow(b) join b.follow(a)
    _     <- a.DM(b, "hello, world!")
} yield ()
val done: Future[Unit] = for {
  (a, b) <- newUser() join newUser()
  _    <- a.follow(b) join b.follow(a)
  _    <- a.DM(b, "hello, world!")
} yield ()
val f = newUser().join(newUser()).flatMap(
  new Function1[Tuple2[User, User], Future[Unit]] {
    def apply(ab: Tuple2[User, User]): Future[Unit] = {
      ab._1.follow(ab._2).join(ab._2.follow(ab._1)).flatMap(
        new Function1[Unit, Unit] {
          def apply(u: Unit) = {
            ab._1.DM(ab._2, "hello, world!"")
          }
        }
      )
    }
  }
)
We said nothing about execution here; we only declared the meaning of (deferred) values.

Future isAuth is the future that, given the result of getUser, is the result of the RPC to the authentication service.
Services & Filters
What is an RPC?

It’s a function that
  Takes a while
Can fail
Req => Future[Rep]
A service is a function

trait Service[Req, Rep] extends (Req => Future[Rep])

val service: Service[Int, String]
val f: Future[String] = service(123)
val multiplier = 
  new Service[Int, Int] { 
    def apply(i: Int) = 
      Future.value(i*2) 
  }

multiplier(123) onSuccess { x =>
  println("multiplied!", x)
}
What’s a filter?

val timeoutFilter: Filter[...]
val service: Service[Req, Rep]

val serviceWithTimeout =
    timeoutFilter andThen service
// : Service[Req, Rep]
What’s a filter?

```scala
val authFilter: Filter[...]
val service: Service[AuthedReq, Rep]

val serviceThatAuthenticates =
  authFilter andThen service
// : Service[Req, Rep]
```
trait Filter[ReqIn, RepOut, ReqOut, RepIn]  
  (ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])

val authFilter = 
  new Filter[Req, Rep, AuthedReq, Rep] { 
    def apply(req: Req, svc: Service[AuthedReq, Rep]) = 
      if (authenticate(req)) 
        svc(AuthedReq(req)) 
      else 
        Future.exception(new AuthException)
  }

Filters are service transformers
A timeout filter

class TimeoutFilter[Req, Rep](
    timeout: Duration
) extends Filter[Req, Rep, Req, Rep]
{
    def apply(
        request: Req, service: Service[Req, Rep]
    ): Future[Rep] =
        service(request).within(timeout)
}
Authentication

class RequireAuthentication(passbird: Passbird)
  extends Filter[HttpReq, HttpRep, AuthHttpReq, HttpRep]
{
  def apply(
    req: HttpReq,
    service: Service[AuthHttpReq, HttpRep]
  ) = {
    passbird.auth(req) flatMap {
      case AuthResult(AuthResultCode.OK, Some(passport), _) =>
        service(AuthHttpReq(req, passport))
      case ar: AuthResult =>
        Trace.record("Passbird authentication failed with " + ar)
        Future.exception(
          new RequestUnauthenticated(ar.resultCode))
    }
  }
}
They’re stackable, typesafe

```scala
val timeout: Filter[...]
val auth: Filter[...]
val serve: Service[...]

timeout andThen auth andThen serve
```
Finagle is our RPC system
Finagle

Provides Service instances via clients.
Exposes Service instances via servers.
Adds behavior, is largely configurable
  Retrying, connection pooling, load balancing, rate limiting, monitoring, stats keeping, ...

Codecs implement wire protocols.
Manages resources for you.
Builders

**ClientBuilder**
produces a Service instance

**ServerBuilder**
consumes a Service instance
val client = ClientBuilder().name("httploadtest").codec(Http).hosts("host1:80,host2:80,...").hostConnectionLimit(10).build()
val service: Service[HttpReq, HttpResp]
val server = ServerBuilder()
   .name("httpd")
   .codec(Http)
   .bindTo(new InetSocketAddress(8080))
   .build(service)

server.close() // when done
A proxy

val client: Service[HttpRequest, HttpResponse] = ClientBuilder()...build()

ServerBuilder()...build(client)
service Hello {
    string hi(1: string a);
}

=>

public interface ServiceIface {
    public Future<String>
        hi(a: String);
}
Thrift transport

≈ Service[Array[Byte], Array[Byte]]
A DISTRIBUTED SEARCH ENGINE IN 200 SLOC

https://github.com/twitter/scala_school/tree/master/searchbird
service SearchbirdService {
// returned strings are json-encoded
// twitter statuses
list<string> query(1: list<string> tokens)
i64 size() throws(1: SearchbirdException ex)
}
// Generated code
trait Index {
  // Query the index for the set of Statuses that have all of the given tokens.
  def query(tokens: Seq[String]): Future[Seq[String]]

  // Number of Statuses in the index.
  def size: Future[Long]
}
class InmemoryIndex(
    tweets: AsyncQueue[Tweet]
) extends Index {
    tweets foreach { tweet =>
        addToIndex(tweet)
    }
    ..
    // a simple posting list
}
class FanoutIndex(underlying: Seq[Index]) extends Index {
    def query(tokens: Seq[String]) =
        Future.collect(
            underlying map(_.query(tokens))
        ) map(_.flatten)

    def size =
        Future.collect(
            underlying map(_.size)
        ) map(_.sum)
}
query, decomposed

val fs = underlying map { index =>
  index.query(tokens)
} // : Seq[Future[Seq[Seq[String]]]]

val coll = Future.collect(fs)
  // : Future[Seq[Seq[Seq[String]]]]

coll map { sstr => sstr.flatten }
  // : Future[Seq[Seq[String]]]
val mod, count: Int = ..
llocalIndex = {
    val queue: AsyncQueue[Tweet] = makeReader()
    val filteredQueue =
        queue filter { status =>
            (status.id % count) == index
        }

    new InmemoryIndex(filteredQueue)
}
val childAddrs: Seq[SocketAddress] = ..
val childIndices: Seq[Index] = childAddrs map { addr =>
  // This creates a transport service
  val service = ClientBuilder()
    .hosts(addr)
    .codec(ThriftClientFramedCodec())
    .hostConnectionLimit(1)
    .build()

  new Searchbird.ServiceToClient(service)
}
val **index** =
    new FanoutIndex(
        localIndex.toSeq ++ childIndices)

val **service** =
    new Searchbird.Service(index)

ServerBuilder()
    .bindTo(new InetSocketAddr(port))
    .codec(Thrift)
    .name("thriftserver")
    .build(service)
Stats.incr("helloworld")

% curl localhost:9990/stats.txt
counters:
  helloworld: 1
Tracing

Built into finagle

Add arbitrary annotations

Tracing system aggregates all spans, and visualizes them
In practice
The code I showed you is real; most of our services look much like this.
Abstraction buys you...

Computation is “descriptive” (future composition); finagle controls execution strategy.

This has many interesting benefits

- Generic tracing
- Generic end-to-end cancellation
- Per request statistics
- Generic request prioritization
- Can avoid priority inversion
Abstraction buys you...

These have turned out to be extremely productive and versatile primitives.
Most of our services are big “future transformers”
Greatly enhanced modularity and genericity
Easier to reason about semantics
Easier to test: behavior tends to be “pure”
Abstraction costs you...

More indirection is more object allocation is more GC pressure.

Most of our “tuning” time is spent dealing with allocation issues.

But: we have good tools, and it’s usually obvious.

A few places (Promise, codecs) require special attention to efficiency, but generally the JVM just works for you.
Abstraction costs you...

Stack traces don’t make much sense anymore.
Instead, future compositions form “dispatch graphs”
The execution model does matter.
Especially when mixing with legacy code.
This can lead to a lot of confusion.
Harder to reason about performance.
Traditional cost models don’t apply.
Scala School!
From ø to Distributed Service

About
Scala school was started as a series of lectures at Twitter to prepare experienced engineers to be productive Scala programmers. Being a relatively new language, but also one that draws on many familiar concepts, we found this an effective way of getting new engineers up to speed quickly. This is the written material that accompanied those lectures. We have found that these are useful in their own right.

Lessons

- Basics
  - Values, functions, classes, methods, inheritance, try-catch-finally.
  - Expression-oriented programming

- Basics continued
  - Case classes, objects, packages, apply, update, Functions are Objects (uniform access principle), pattern matching

- Collections
  - Lists, Maps, functional combinators (map, foreach, filter, zip, folds)

- Pattern matching & functional composition
  - More functions! PartialFunctions, more Pattern Matching

- Type & polymorphism basics
  - Basic Types and type polymorphism, type inference, variance, bounds, quantification

- Advanced types
  - Advanced Types, view bounds, higher-kindred types, recursive types, structural types

Approach
We think it makes the most sense to approach teaching Scala not as if it's an improved Java but as a new language. Experience in Java is not expected. Focus will be around the interpreter and the object-functional style as well as the style of programming we do here. An emphasis will be placed on maintainability, clarity of
Effective Scala

Marius Eriksen, Twitter Inc.
marius@twitter.com

Table of Contents

- **Introduction**
- **Formatting**: Whitespace, Naming, Imports, Braces, Pattern matching, Comments
- **Types and Generics**: Return type annotations, Variance, Type aliases, Implicit
- **Collections**: Hierarchy, Use, Style, Performance
- **Concurrency**: Futures, Collections
- **Control structures**: Recursion, Returns, for loops and comprehensions, require and assert
- **Functional programming**: Case classes as algebraic data types, Options, Pattern matching, Partial
  functions, Destructuring bindings, Laziness, Call by name, FlatMap
- **Object oriented programming**: Dependency injection, Traits, Visibility, Structural typing
- **Garbage collection**
- **Java compatibility**
- **Twitter’s standard libraries**: Futures

Introduction

Scala is one of the chief application programming languages used at Twitter. Much of our infrastructure is written
in Scala and we have several large libraries supporting our use. While highly effective, Scala is also a large language,
and our experiences have taught us to practice great care in its application. What are its pitfalls? Which features do
we embrace, which do we eschew? When do we employ “purely functional style”, and when do we avoid it? In
other words: what have we found to be an effective use of the language? This guide attempts to distill our experience.
Thanks. Questions?

https://github.com/twitter/util
https://github.com/twitter/finagle
http://twitter.github.com/finagle
finaglers@googlegroups.com
Finagle
under the hood
trait ServiceFactory[−Req, +Rep] {
  def make(): Future[Service[Req, Rep]]
  def close(): Unit
  def isAvailable: Boolean = true
}
Finagle’s two halves

Service[Req, Rep]

Service stack

Netty pipeline

Java NIO
Server pipeline

Channel Bridge
Dispatches requests to service

Channel semaphore
Ensures 1:1 semantics

Codec
Implements protocol

SSL
Just that

Connection manager
Lifetime, timeouts, stats
Server stack

Stats
Maintains request stats

Connection expiration
Expires connection after idle time

Request timeout
Times out slow requests

Tracing
Initiates & stores traces.
Client pipeline

Channel Service
Provides a Service[Req, Rep] interface.

Codec
Protocol implementation

SSL
Just that.

Connection manager
Idle time, reader timeouts, etc.

Snooper (optional)
Debugging output

Tuesday, April 10, 12
Client stack

- **Retrying**
  Retries failed requests

- **Load balancer**
  Balances between pools

- **Stats**
  Maintains request stats

- **Failure accrual**
  Marks services dead

- **Request timeout**
  Times out slow requests

- **Pool**
  Connection pooling.
LET'S MAKE A CODEC!
class StringCodec extends CodecFactory[String, String] {
    def server = Function.const {
        new Codec[String, String] {
            def pipelineFactory = new ChannelPipelineFactory {
                def getPipeline = {
                    val pipeline = Channels.pipeline()
                    pipeline.addLast("line",
                        new DelimiterBasedFrameDecoder(
                            100, Delimiters.lineDelimiter: _*)))
                    pipeline.addLast("stringDecoder",
                        new StringDecoder(CharsetUtil.UTF_8))
                    pipeline.addLast("stringEncoder",
                        new StringEncoder(CharsetUtil.UTF_8))
                    pipeline
                }
            }
        }
    }
}

...
...  

def client = Function.const {
    new Codec[String, String] {
        def pipelineFactory = new ChannelPipelineFactory {
            def getPipeline = {
                val pipeline = Channels.pipeline()
                pipeline.addLast("stringEncode",
                    new StringEncoder(CharsetUtil.UTF_8))
                pipeline.addLast("stringDecode",
                    new StringDecoder(CharsetUtil.UTF_8))
                pipeline
            }
        }
    }
}
val client: Service[String, String] = ClientBuilder()  
  .codec(StringCodec)  
  .hosts(new InetSocketAddress(8080))  
  .hostConnectionLimit(1)  
  .build()

client("hi mom\n") onSuccess { result =>
  println("Received result asynchronously: " + result)
} onFailure { error =>
  error.printStackTrace()
} ensure {
  client.release()
}
val service = new Service[String, String] {
  def apply(request: String) =
    Future.value(request.rot13())
}

val server: Server = ServerBuilder()
  .name("echoserver")
  .codec(StringCodec)
  .bindTo(new InetSocketAddress(8080))
  .build(service)
Backup
OFFER/BROKER
// An offer to communicate
// a value of type ‘T’

trait Offer[T] {
    // Synchronize this offer.
    // Activates the offer, attempting
    // to perform the communication.
    def apply(): Future[T]
}
Synchronization

Happens exactly when both parties agree to synchronize (ie. it is synchronous).

Example:

```scala
val send: Offer[Unit]
val recv: Offer[T]
send(), recv()
```

The resulting Futures are satisfied exactly when `send` and `recv` are both enabled.
Offers are persistent

They are only enabled when synchronized; the original offer value remains valid.

For example, we can “stream” values by continually synchronizing:

```scala
def Offer.foreach(f: T => Unit) {
  this() foreach { v =>
    f(v)
    foreach(f)
  }
}
```
Selective sync

`Offer.choose[T](ofs: Offer[T]*) : Offer[T]`

An offer which – when synchronized – synchronizes exactly one of `ofs` can (whichever is available first, otherwise at random).

Under the hood we must support “retracting” offers.
THIS IS INCREDIBLY POWERFUL
A FIFO “channel” providing brokering offers to send and receive.

class Broker[E] {
    def send(e: E): Offer[Unit]
    def recv: Offer[E]
}
class Broker[E] {
    def !(e: E): Future[Unit] =
        send(e)()
    def ? : Future[E] =
        recv()
}
Other offers

Offer.const[T](t: T): Offer[T]
Offer.never[T]: Offer[T]
Offer.timeout(to: Duration): Offer[Unit]
Future[T].toOffer: Offer[T]
More combinators

Offer.map[U](f: T => U): Offer[U]

Offer.orElse[U >: T](other: Offer[U]): Offer[U]

Offer.or[U](other: Offer[U]): Offer[Either[T, U]]
Example: merge

```scala
val ts: Seq[Offer[T]] = ...

val merged: Offer[T] = Offer.choose(ts:_*)
```
case class ReadMessage(
  bytes: ChannelBuffer,
  ack  : Offer[Unit])

trait ReadHandle {
  val messages: Offer[ReadMessage]
  val error   : Offer[Throwable]
  def close() {
  }
}
// Combine several handles into one.

def grabbyHands(
    handles: Seq[ReadHandle]
): ReadHandle
// These are for the outgoing (merged) view.
val error = new Broker[Throwable]
val messages = new Broker[ReadMessage]
val close = new Broker[Unit]

var underlying = Set() ++ handles
val errors: Offer[ReadHandle] =
  Offer.choose(
    underlying map { h =>
      h.error map { _ => h }
    }
toSeq:_*
  )

// (closes are requested by the user)
val closeOf = close.recv { _ =>
  underlying foreach { _.close() }
  error ! ReadClosedException
}
def loop() {
    // compute the active queues
    if (underlying.isEmpty) {
        error ! AllHandlesDiedException
        return
    }
    val queues = underlying map { _.messages } toSeq

    val of = closeOf orElse {
        Offer.choose(
            closeOf,
            Offer.choose(queues:_*) { m =>
                messages ! m
                loop()
            },
            errors { h =>
                underlying -= h
                loop()
            }
        )
    }
    of()  // synchronize!
}
val addBroker = new Broker[Broker[ChannelBuffer]]
val remBroker = new Broker[Broker[ChannelBuffer]]
val messages = new Broker[ChannelBuffer]
def tee(receivers: Set[Broker[ChannelBuffer]]) {
  Offer.select(
    addBroker.recv { b => tee(receivers + b) },
    remBroker.recv { b => tee(receivers - b) },
    if (receivers.isEmpty) Offer.never else {
      messages.recv { m =>
        Future.join(
          receivers map { _ ! m } toSeq
        ) ensure { tee(receivers) }
      }
    }
  )
}
tee(Set())
class HosebirdService
  extends Service[HttpRequest, StreamResponse]
{
  def apply(request: HttpRequest) = Future {
    val subscriber = new Broker[ChannelBuffer]
    addBroker ! subscriber
    new StreamResponse {
      val httpResponse = new DefaultHttpResponse(
        request.getProtocolVersion,
        HttpResponseStatus.OK)
      def messages = subscriber.recv
      def error = new Broker[Throwable].recv
      def release() = {
        remBroker ! subscriber
        // sink any existing messages, so they
        // don't hold up the upstream.
        subscriber.recv foreach { _ => () }
      }
    }
  }
}
// from previous slides:
val messages: Broker[ChannelBuffer]
val handle: ReadHandle  // kestrel

def read() {
  (handlemsgs.recv?) foreach { m =>
    (messages ! m.bytes) ensure {
      m.ack(); read()
    }
  }
}

read()
package com.twitter.finagle.pool

import scala.annotation.tailrec
import scala.collection.mutable.Queue

import com.twitter.util.{Future, Promise, Return, Throw}
import com.twitter.finagle.{Service, ServiceFactory, ServiceClosedException,
TooManyWaitersException, ServiceProxy,
CancelledConnectionException}
import com.twitter.finagle.stats.{NullStatsReceiver, StatsReceiver}

/**
 * The watermark pool is an object pool with low & high
 * watermarks. This behaves as follows: the pool will persist up to
 * the low watermark number of items (as long as they have been
 * created), and won't start queueing requests until the high
 * watermark has been reached. Put another way: up to `lowWatermark'
 * items may persist indefinitely, while there are at no times more
 * than `highWatermark' items in concurrent existence.
 */

class WatermarkPool[Req, Rep](
  factory: ServiceFactory[Req, Rep],
  lowWatermark: Int, highWatermark: Int = Int.MaxValue,
  statsReceiver: StatsReceiver = NullStatsReceiver,
  maxWaiters: Int = Int.MaxValue)
extends ServiceFactory[Req, Rep] {

  private[this] val queue       = Queue[ServiceWrapper]()
  private[this] val waiters     = Queue[Promise[Service[Req, Rep]]]()
  private[this] var numServices = 0
  private[this] var isOpen      = true
  private[this] val waitersStat =
    statsReceiver.addGauge("pool_waiters") { waiters.size }
  private[this] val sizeStat = statsReceiver.addGauge("pool_size")
    { numServices }

  private[this] class ServiceWrapper(underlying: Service[Req, Rep])
    extends ServiceProxy[Req, Rep](underlying) {
    override def release() = WatermarkPool.this.synchronized {
      if (!isOpen) {
        underlying.release()
        numServices -= 1
      } else if (!isAvailable) {
        underlying.release()
        numServices -= 1
        // If we just disposed of an service, and this bumped us
        // the high watermark, then we are free to satisfy the first
        // waiter.
        if (numServices < highWatermark && !waiters.isEmpty) {
          val waiter = waiters.dequeue()
          val res = make() respond { waiter() = _ }
          waiter.linkTo(res)
        }
      } else if (!waiters.isEmpty) {
        val waiter = waiters.dequeue()
        waiter() = Return(this)
      } else if (numServices <= lowWatermark) {
        queue += this
      } else {
        underlying.release()
        numServices -= 1
      }
    }
  }

  @tailrec private[this] def dequeue(): Option[Service[Req, Rep]] = {
    if (queue.isEmpty) {
      None
    } else {
      val service = queue.dequeue()
      if (!service.isAvailable) {
        // Note: since these are ServiceWrappers, accounting is taken
        // care of by ServiceWrapper.release()
        service.release()
        dequeue()
      } else {
        Some(service)
      }
    }
  }
}

@tailrec private[this] def dequeue(): Option[Service[Req, Rep]] = {
  if (queue.isEmpty) {
    None
  } else {
    val service = queue.dequeue()
    if (!service.isAvailable) {
      // Note: since these are ServiceWrappers, accounting is taken
      // care of by ServiceWrapper.release()
      service.release()
      dequeue()
    } else {
      Some(service)
    }
  }
}
type Waiter = Promise[Service[Req, Rep]]
val addWaiter: new Broker[Waiter]
val returnService: Broker[Service[Req, Rep]]

def make(): Future[Service[Req, Rep]] = {
  val p = new Promise[Service[Req, Rep]]
  addWaiter ! p
  p
}

def wrap(underlying: Service[Req, Rep]) =
  new ServiceProxy[Req, Rep](underlying) {
    override def release() {
      returnService ! underlying
    }
  }
}
val queuedWaiter: Broker[Waiter]

def queue(incoming: Queue[Waiter]) {
  val q = incoming filter { !_.isCancelled }
  Offer.select(
    addWaiter.recv { w =>
      if (q.size < maxWaiters) {
        q enqueue w
      } else {
        w() = Throw(new TooManyWaitersException)
        q
      }
    }
  )
  if (q.isEmpty) Offer.never else {
    val (item, qq) = q.dequeue
    queuedWaiter.send(item) const(qq)
  }
  foreach { queue(_) }
}
def dispatcher(q: Queue[Service[Req, Rep]], n: Int) = Offer.select(
  if (n == highWatermark) Offer.never else queuedWaiter.recv {
    case w if q.isEmpty =>
      factory.make() map { wrap(_) } proxyTo w
      dispatcher(q, n + 1)
    case w =>
      val (s, qq) = q.dequeue
      w() = Return(s)
      dispatcher(qq, n)
  },

  returnService.recv {
    case s if s.isAvailable && n < lowWatermark =>
      dispatcher(q enqueue s, n)
    case s =>
      s.release()
      dispatcher(q, n - 1)
  },

  closePool.recv { _ =>
    addWaiter.recv foreach { _() = Throw(new ServiceClosedException) }
    returnService.recv foreach { _.release() }
    q foreach { _.release() }
    factory.close()
  }
)
Concurrent Programming in ML

John H. Reppy