package com.twitter.visibility.rules

import com.twitter.spam.rtf.thriftscala.SafetyResultReason

import com.twitter.util.Memoize

import com.twitter.visibility.common.actions.AppealableReason

import com.twitter.visibility.common.actions.AvoidReason.MightNotBeSuitableForAds

import com.twitter.visibility.common.actions.LimitedEngagementReason

import com.twitter.visibility.common.actions.SoftInterventionDisplayType

import com.twitter.visibility.common.actions.SoftInterventionReason

import com.twitter.visibility.common.actions.LimitedActionsPolicy

import com.twitter.visibility.common.actions.LimitedAction

import com.twitter.visibility.common.actions.converter.scala.LimitedActionTypeConverter

import com.twitter.visibility.configapi.params.FSRuleParams.FosnrFallbackDropRulesEnabledParam

import com.twitter.visibility.configapi.params.FSRuleParams.FosnrRulesEnabledParam

import com.twitter.visibility.configapi.params.RuleParam

import com.twitter.visibility.configapi.params.RuleParams.EnableFosnrRuleParam

import com.twitter.visibility.features.Feature

import com.twitter.visibility.features.TweetSafetyLabels

import com.twitter.visibility.models.TweetSafetyLabel

import com.twitter.visibility.models.TweetSafetyLabelType

import com.twitter.visibility.models.ViolationLevel

import com.twitter.visibility.rules.ComposableActions.ComposableActionsWithInterstitialLimitedEngagements

import com.twitter.visibility.rules.ComposableActions.ComposableActionsWithSoftIntervention

import com.twitter.visibility.rules.ComposableActions.ComposableActionsWithAppealable

import com.twitter.visibility.rules.ComposableActions.ComposableActionsWithInterstitial

import com.twitter.visibility.rules.Condition.And

import com.twitter.visibility.rules.Condition.NonAuthorViewer

import com.twitter.visibility.rules.Condition.Not

import com.twitter.visibility.rules.Condition.ViewerDoesNotFollowAuthorOfFosnrViolatingTweet

import com.twitter.visibility.rules.Condition.ViewerFollowsAuthorOfFosnrViolatingTweet

import com.twitter.visibility.rules.FreedomOfSpeechNotReach.DefaultViolationLevel

import com.twitter.visibility.rules.Reason.\_

import com.twitter.visibility.rules.State.Evaluated

object FreedomOfSpeechNotReach {

val DefaultViolationLevel = ViolationLevel.Level1

def reasonToSafetyResultReason(reason: Reason): SafetyResultReason =

reason match {

case HatefulConduct => SafetyResultReason.FosnrHatefulConduct

case AbusiveBehavior => SafetyResultReason.FosnrAbusiveBehavior

case \_ => SafetyResultReason.FosnrUnspecified

}

def reasonToSafetyResultReason(reason: AppealableReason): SafetyResultReason =

reason match {

case AppealableReason.HatefulConduct(\_) => SafetyResultReason.FosnrHatefulConduct

case AppealableReason.AbusiveBehavior(\_) => SafetyResultReason.FosnrAbusiveBehavior

case \_ => SafetyResultReason.FosnrUnspecified

}

val EligibleTweetSafetyLabelTypes: Seq[TweetSafetyLabelType] =

Seq(ViolationLevel.Level4, ViolationLevel.Level3, ViolationLevel.Level2, ViolationLevel.Level1)

.map {

ViolationLevel.violationLevelToSafetyLabels.get(\_).getOrElse(Set()).toSeq

}.reduceLeft {

\_ ++ \_

}

private val EligibleTweetSafetyLabelTypesSet = EligibleTweetSafetyLabelTypes.toSet

def extractTweetSafetyLabel(featureMap: Map[Feature[\_], \_]): Option[TweetSafetyLabel] = {

val tweetSafetyLabels = featureMap(TweetSafetyLabels)

.asInstanceOf[Seq[TweetSafetyLabel]]

.flatMap { tsl =>

if (FreedomOfSpeechNotReach.EligibleTweetSafetyLabelTypesSet.contains(tsl.labelType)) {

Some(tsl.labelType -> tsl)

} else {

None

}

}

.toMap

FreedomOfSpeechNotReach.EligibleTweetSafetyLabelTypes.flatMap(tweetSafetyLabels.get).headOption

}

def eligibleTweetSafetyLabelTypesToAppealableReason(

labelType: TweetSafetyLabelType,

violationLevel: ViolationLevel

): AppealableReason = {

labelType match {

case TweetSafetyLabelType.FosnrHatefulConduct =>

AppealableReason.HatefulConduct(violationLevel.level)

case TweetSafetyLabelType.FosnrHatefulConductLowSeveritySlur =>

AppealableReason.HatefulConduct(violationLevel.level)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

}

def limitedActionConverter(

limitedActionStrings: Option[Seq[String]]

): Option[LimitedActionsPolicy] = {

val limitedActions = limitedActionStrings.map { limitedActionString =>

limitedActionString

.map(action => LimitedActionTypeConverter.fromString(action)).map { action =>

action match {

case Some(a) => Some(LimitedAction(a, None))

case \_ => None

}

}.flatten

}

limitedActions.map(actions => LimitedActionsPolicy(actions))

}

}

object FreedomOfSpeechNotReachReason {

def unapply(softIntervention: SoftIntervention): Option[AppealableReason] = {

softIntervention.reason match {

case SoftInterventionReason.FosnrReason(appealableReason) => Some(appealableReason)

case \_ => None

}

}

def unapply(

interstitialLimitedEngagements: InterstitialLimitedEngagements

): Option[AppealableReason] = {

interstitialLimitedEngagements.limitedEngagementReason match {

case Some(LimitedEngagementReason.FosnrReason(appealableReason)) => Some(appealableReason)

case \_ => None

}

}

def unapply(

interstitial: Interstitial

): Option[AppealableReason] = {

interstitial.reason match {

case Reason.FosnrReason(appealableReason) => Some(appealableReason)

case \_ => None

}

}

def unapply(

appealable: Appealable

): Option[AppealableReason] = {

Reason.toAppealableReason(appealable.reason, appealable.violationLevel)

}

def unapply(

action: Action

): Option[AppealableReason] = {

action match {

case a: SoftIntervention =>

a match {

case FreedomOfSpeechNotReachReason(r) => Some(r)

case \_ => None

}

case a: InterstitialLimitedEngagements =>

a match {

case FreedomOfSpeechNotReachReason(r) => Some(r)

case \_ => None

}

case a: Interstitial =>

a match {

case FreedomOfSpeechNotReachReason(r) => Some(r)

case \_ => None

}

case a: Appealable =>

a match {

case FreedomOfSpeechNotReachReason(r) => Some(r)

case \_ => None

}

case ComposableActionsWithSoftIntervention(FreedomOfSpeechNotReachReason(appealableReason)) =>

Some(appealableReason)

case ComposableActionsWithInterstitialLimitedEngagements(

FreedomOfSpeechNotReachReason(appealableReason)) =>

Some(appealableReason)

case ComposableActionsWithInterstitial(FreedomOfSpeechNotReachReason(appealableReason)) =>

Some(appealableReason)

case ComposableActionsWithAppealable(FreedomOfSpeechNotReachReason(appealableReason)) =>

Some(appealableReason)

case \_ => None

}

}

}

object FreedomOfSpeechNotReachActions {

trait FreedomOfSpeechNotReachActionBuilder[T <: Action] extends ActionBuilder[T] {

def withViolationLevel(violationLevel: ViolationLevel): FreedomOfSpeechNotReachActionBuilder[T]

}

case class DropAction(violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[Drop] {

override def actionType: Class[\_] = classOf[Drop]

override val actionSeverity = 16

private def toRuleResult: Reason => RuleResult = Memoize { r => RuleResult(Drop(r), Evaluated) }

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(Reason.fromAppealableReason(appealableReason))

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class AppealableAction(violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[TweetInterstitial] {

override def actionType: Class[\_] = classOf[Appealable]

override val actionSeverity = 17

private def toRuleResult: Reason => RuleResult = Memoize { r =>

RuleResult(

TweetInterstitial(

interstitial = None,

softIntervention = None,

limitedEngagements = None,

downrank = None,

avoid = Some(Avoid(None)),

mediaInterstitial = None,

tweetVisibilityNudge = None,

abusiveQuality = None,

appealable = Some(Appealable(r, violationLevel = violationLevel))

),

Evaluated

)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(Reason.fromAppealableReason(appealableReason))

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class AppealableAvoidLimitedEngagementsAction(

violationLevel: ViolationLevel = DefaultViolationLevel,

limitedActionStrings: Option[Seq[String]])

extends FreedomOfSpeechNotReachActionBuilder[Appealable] {

override def actionType: Class[\_] = classOf[AppealableAvoidLimitedEngagementsAction]

override val actionSeverity = 17

private def toRuleResult: Reason => RuleResult = Memoize { r =>

RuleResult(

TweetInterstitial(

interstitial = None,

softIntervention = None,

limitedEngagements = Some(

LimitedEngagements(

toLimitedEngagementReason(

Reason

.toAppealableReason(r, violationLevel)

.getOrElse(AppealableReason.Unspecified(violationLevel.level))),

FreedomOfSpeechNotReach.limitedActionConverter(limitedActionStrings)

)),

downrank = None,

avoid = Some(Avoid(None)),

mediaInterstitial = None,

tweetVisibilityNudge = None,

abusiveQuality = None,

appealable = Some(Appealable(r, violationLevel = violationLevel))

),

Evaluated

)

}

def build(

evaluationContext: EvaluationContext,

featureMap: Map[Feature[\_], \_]

): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(Reason.fromAppealableReason(appealableReason))

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class AvoidAction(violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[Avoid] {

override def actionType: Class[\_] = classOf[Avoid]

override val actionSeverity = 1

private def toRuleResult: Reason => RuleResult = Memoize { r =>

RuleResult(Avoid(None), Evaluated)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(Reason.fromAppealableReason(appealableReason))

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class LimitedEngagementsAction(violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[LimitedEngagements] {

override def actionType: Class[\_] = classOf[LimitedEngagements]

override val actionSeverity = 6

private def toRuleResult: Reason => RuleResult = Memoize { r =>

RuleResult(LimitedEngagements(LimitedEngagementReason.NonCompliant, None), Evaluated)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(Reason.fromAppealableReason(appealableReason))

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class InterstitialLimitedEngagementsAction(

violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[InterstitialLimitedEngagements] {

override def actionType: Class[\_] = classOf[InterstitialLimitedEngagements]

override val actionSeverity = 11

private def toRuleResult: Reason => RuleResult = Memoize { r =>

RuleResult(InterstitialLimitedEngagements(r, None), Evaluated)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(Reason.fromAppealableReason(appealableReason))

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class InterstitialLimitedEngagementsAvoidAction(

violationLevel: ViolationLevel = DefaultViolationLevel,

limitedActionStrings: Option[Seq[String]])

extends FreedomOfSpeechNotReachActionBuilder[TweetInterstitial] {

override def actionType: Class[\_] = classOf[InterstitialLimitedEngagementsAvoidAction]

override val actionSeverity = 14

private def toRuleResult: AppealableReason => RuleResult = Memoize { r =>

RuleResult(

TweetInterstitial(

interstitial = Some(

Interstitial(

reason = FosnrReason(r),

localizedMessage = None,

)),

softIntervention = None,

limitedEngagements = Some(

LimitedEngagements(

reason = toLimitedEngagementReason(r),

policy = FreedomOfSpeechNotReach.limitedActionConverter(limitedActionStrings))),

downrank = None,

avoid = Some(Avoid(None)),

mediaInterstitial = None,

tweetVisibilityNudge = None

),

Evaluated

)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

labelType = label,

violationLevel = violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(appealableReason)

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class SoftInterventionAvoidAction(violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[TweetInterstitial] {

override def actionType: Class[\_] = classOf[SoftInterventionAvoidAction]

override val actionSeverity = 8

private def toRuleResult: AppealableReason => RuleResult = Memoize { r =>

RuleResult(

TweetInterstitial(

interstitial = None,

softIntervention = Some(

SoftIntervention(

reason = toSoftInterventionReason(r),

engagementNudge = false,

suppressAutoplay = true,

warning = None,

detailsUrl = None,

displayType = Some(SoftInterventionDisplayType.Fosnr)

)),

limitedEngagements = None,

downrank = None,

avoid = Some(Avoid(None)),

mediaInterstitial = None,

tweetVisibilityNudge = None,

abusiveQuality = None

),

Evaluated

)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(appealableReason)

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class SoftInterventionAvoidLimitedEngagementsAction(

violationLevel: ViolationLevel = DefaultViolationLevel,

limitedActionStrings: Option[Seq[String]])

extends FreedomOfSpeechNotReachActionBuilder[TweetInterstitial] {

override def actionType: Class[\_] = classOf[SoftInterventionAvoidLimitedEngagementsAction]

override val actionSeverity = 13

private def toRuleResult: AppealableReason => RuleResult = Memoize { r =>

RuleResult(

TweetInterstitial(

interstitial = None,

softIntervention = Some(

SoftIntervention(

reason = toSoftInterventionReason(r),

engagementNudge = false,

suppressAutoplay = true,

warning = None,

detailsUrl = None,

displayType = Some(SoftInterventionDisplayType.Fosnr)

)),

limitedEngagements = Some(

LimitedEngagements(

toLimitedEngagementReason(r),

FreedomOfSpeechNotReach.limitedActionConverter(limitedActionStrings))),

downrank = None,

avoid = Some(Avoid(None)),

mediaInterstitial = None,

tweetVisibilityNudge = None,

abusiveQuality = None

),

Evaluated

)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(appealableReason)

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

case class SoftInterventionAvoidAbusiveQualityReplyAction(

violationLevel: ViolationLevel = DefaultViolationLevel)

extends FreedomOfSpeechNotReachActionBuilder[TweetInterstitial] {

override def actionType: Class[\_] = classOf[SoftInterventionAvoidAbusiveQualityReplyAction]

override val actionSeverity = 13

private def toRuleResult: AppealableReason => RuleResult = Memoize { r =>

RuleResult(

TweetInterstitial(

interstitial = None,

softIntervention = Some(

SoftIntervention(

reason = toSoftInterventionReason(r),

engagementNudge = false,

suppressAutoplay = true,

warning = None,

detailsUrl = None,

displayType = Some(SoftInterventionDisplayType.Fosnr)

)),

limitedEngagements = None,

downrank = None,

avoid = Some(Avoid(None)),

mediaInterstitial = None,

tweetVisibilityNudge = None,

abusiveQuality = Some(ConversationSectionAbusiveQuality)

),

Evaluated

)

}

def build(evaluationContext: EvaluationContext, featureMap: Map[Feature[\_], \_]): RuleResult = {

val appealableReason =

FreedomOfSpeechNotReach.extractTweetSafetyLabel(featureMap).map(\_.labelType) match {

case Some(label) =>

FreedomOfSpeechNotReach.eligibleTweetSafetyLabelTypesToAppealableReason(

label,

violationLevel)

case \_ =>

AppealableReason.Unspecified(violationLevel.level)

}

toRuleResult(appealableReason)

}

override def withViolationLevel(violationLevel: ViolationLevel) = {

copy(violationLevel = violationLevel)

}

}

}

object FreedomOfSpeechNotReachRules {

abstract class OnlyWhenAuthorViewerRule(

actionBuilder: ActionBuilder[\_ <: Action],

condition: Condition)

extends Rule(actionBuilder, And(Not(NonAuthorViewer), condition))

abstract class OnlyWhenNonAuthorViewerRule(

actionBuilder: ActionBuilder[\_ <: Action],

condition: Condition)

extends Rule(actionBuilder, And(NonAuthorViewer, condition))

case class ViewerIsAuthorAndTweetHasViolationOfLevel(

violationLevel: ViolationLevel,

override val actionBuilder: ActionBuilder[\_ <: Action])

extends OnlyWhenAuthorViewerRule(

actionBuilder,

Condition.TweetHasViolationOfLevel(violationLevel)

) {

override lazy val name: String = s"ViewerIsAuthorAndTweetHasViolationOf$violationLevel"

override def enabled: Seq[RuleParam[Boolean]] =

Seq(EnableFosnrRuleParam, FosnrRulesEnabledParam)

}

case class ViewerIsFollowerAndTweetHasViolationOfLevel(

violationLevel: ViolationLevel,

override val actionBuilder: ActionBuilder[\_ <: Action])

extends OnlyWhenNonAuthorViewerRule(

actionBuilder,

And(

Condition.TweetHasViolationOfLevel(violationLevel),

ViewerFollowsAuthorOfFosnrViolatingTweet)

) {

override lazy val name: String = s"ViewerIsFollowerAndTweetHasViolationOf$violationLevel"

override def enabled: Seq[RuleParam[Boolean]] =

Seq(EnableFosnrRuleParam, FosnrRulesEnabledParam)

override val fallbackActionBuilder: Option[ActionBuilder[\_ <: Action]] = Some(

new ConstantActionBuilder(Avoid(Some(MightNotBeSuitableForAds))))

}

case class ViewerIsNonFollowerNonAuthorAndTweetHasViolationOfLevel(

violationLevel: ViolationLevel,

override val actionBuilder: ActionBuilder[\_ <: Action])

extends OnlyWhenNonAuthorViewerRule(

actionBuilder,

And(

Condition.TweetHasViolationOfLevel(violationLevel),

ViewerDoesNotFollowAuthorOfFosnrViolatingTweet)

) {

override lazy val name: String =

s"ViewerIsNonFollowerNonAuthorAndTweetHasViolationOf$violationLevel"

override def enabled: Seq[RuleParam[Boolean]] =

Seq(EnableFosnrRuleParam, FosnrRulesEnabledParam)

override val fallbackActionBuilder: Option[ActionBuilder[\_ <: Action]] = Some(

new ConstantActionBuilder(Avoid(Some(MightNotBeSuitableForAds))))

}

case class ViewerIsNonAuthorAndTweetHasViolationOfLevel(

violationLevel: ViolationLevel,

override val actionBuilder: ActionBuilder[\_ <: Action])

extends OnlyWhenNonAuthorViewerRule(

actionBuilder,

Condition.TweetHasViolationOfLevel(violationLevel)

) {

override lazy val name: String =

s"ViewerIsNonAuthorAndTweetHasViolationOf$violationLevel"

override def enabled: Seq[RuleParam[Boolean]] =

Seq(EnableFosnrRuleParam, FosnrRulesEnabledParam)

override val fallbackActionBuilder: Option[ActionBuilder[\_ <: Action]] = Some(

new ConstantActionBuilder(Avoid(Some(MightNotBeSuitableForAds))))

}

case object TweetHasViolationOfAnyLevelFallbackDropRule

extends RuleWithConstantAction(

Drop(reason = NotSupportedOnDevice),

Condition.TweetHasViolationOfAnyLevel

) {

override def enabled: Seq[RuleParam[Boolean]] =

Seq(EnableFosnrRuleParam, FosnrFallbackDropRulesEnabledParam)

}

}