Part2.　リポジトリを使った共同開発

シューティングゲームを作りながら、リポジトリを使った開発スタイルに慣れる。

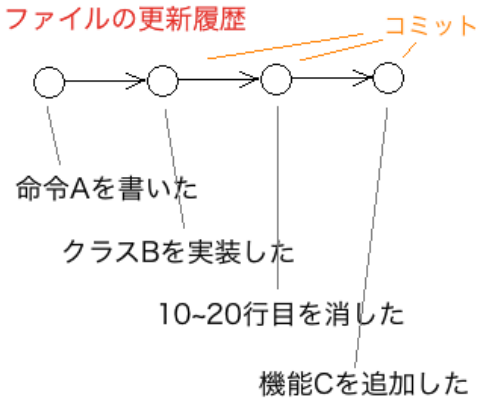
1. リポジトリによる開発サイクル
2. シューティングゲーム
3. 共同開発
4. シューティングゲーム改良案

1.　リポジトリによる開発サイクル

(1) リポジトリってなんですか

・ファイルの更新履歴を保存するところ

大きなプログラムなり文章なりを作るときは、右のようにちょっとずつ改良を重ねていくのが普通である。



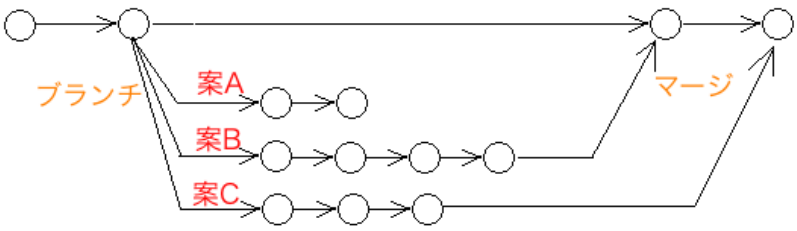
このとき、昔の状態に戻りたい、と思うことがよくある。例えば、「ミスって大事な部分を消してしまった」とか、「新機能を追加したらバグった」とか、「この部分をどうして作ったのか思い出せないからその時の状態を知りたい」とかである。

リポジトリは、このような「昔に戻る」ことを可能にするシステムである。ファイルを更新する度に記録を残しておき（この操作を「コミットする」という）、いつでもその記録にある状態に戻ることが出来る。普通のワードとかにも「もとに戻す」機能はあるが、あれの強いやつだと思えばいい（たとえば一度電源を切っても記録が保持される）。

・新機能を追加する際の実験台

ある程度出来上がったプログラムに、いくつかの改良案を思いついたとしよう。それぞれを別々に試してみて、良かったものを取り入れたい。そんなときにリポジトリは便利である。

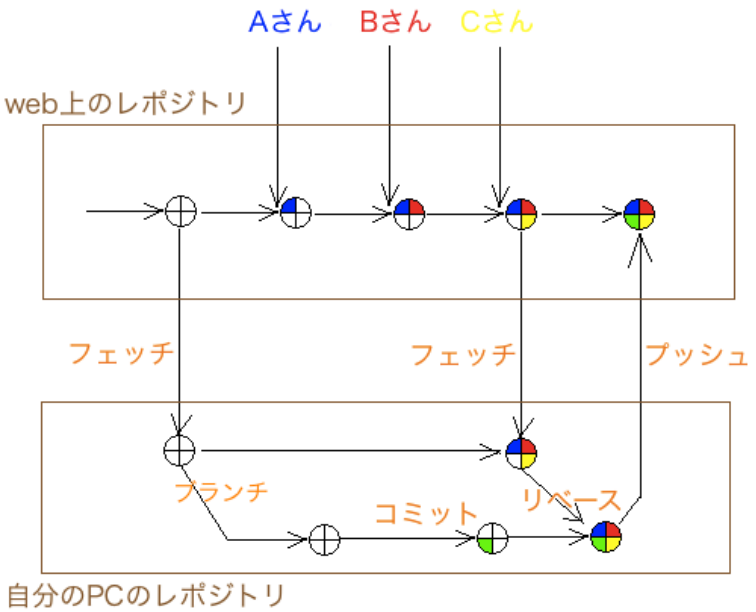
次の図は、3 つの案を同時に並行して試してみて、B が良かったので取り入れ、さらに欲張って C も採用した、というケースを表している。
リポジトリでは、このような操作を可能にするために、ファイルをいくつかの候補に分岐させる機能（ブランチ）と、分岐させたものを統合する機能（マージ）を使うことが出来る。



・ネットを介して多人数で 1 つのものを作るための仕組み
大人数でひとつのプログラムを作るときは、web 上に共有リポジトリを用意して、それを皆で改良していく。開発を始めるときは、まず web 上のリポジトリにあるデータを自分の PC にコピーする（フェッチという）。web 上のリポジトリと、自分の PC のリポジトリとは別物である。だから、フェッチした後の開発は飛行機の中とかネットに繋がらないところでもできる。

下の図のように、自分の PC での開発が一段落した時（下の図で言うと緑の部分が完成した時）に、A, B, C さんが改良を進めていて、青、赤、黄の部分が出来上がっていたとしよう。このとき自分の手元にはまだそれらの変更は反映されていない。
こういったときに、青赤黄と緑が全く別のファイルだったりしたら良いのだが、同じファイルの同じ場所を別々に編集していたりすることがよくある。そうすると、青赤黄の成果にどうやって緑をくっつければいいか困ってしまう（これをコンフリクト(衝突)という）。

そこで、どこに衝突があって矛盾が生じるのかを新たにフェッチしてきた最新データと見比べながらリストアップし、1 つずつ解決していく必要がある。この作業をリベースという。
リベースが終わり衝突が解決したら、その結果を web 上のリポジトリにアップして(プッシュという)仕事を終える。

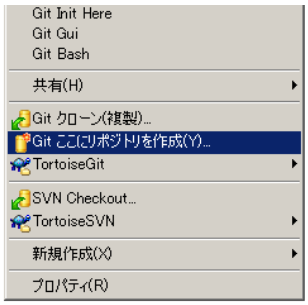


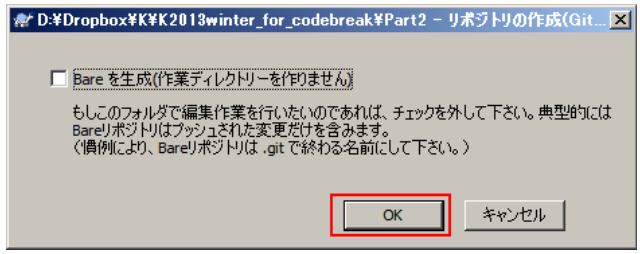
以上がリポジトリの大雑把な説明である。見慣れない言葉が沢山出てきたと思うが、赤色を付けた
　コミット、ブランチ、マージ、フェッチ、コンフリクト、リベース、プッシュはこれから実際に皆さんにやってもらうコマンドである。使いながら慣れていこう。

(2) リポジトリを作ろう
まずは web 上のリポジトリを作る。codebreak のページにログインして、新規リポジトリの作成を押す。作成画面が出るので、
　　リポジトリ名を Part2
　　リポジトリ種類を 非公開
として作成しよう。ちなみに公開リポジトリにすると全世界から見られるようになる。素晴らしいプログラムを公開リポジトリにしてると IT 系企業からスカウトされたりするらしい。

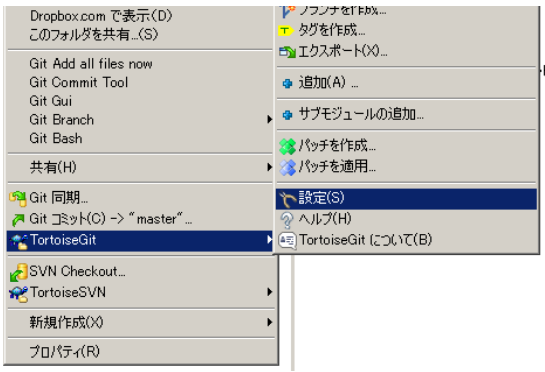
さて、このままではまだ web レポジトリは動いていない。それは、レポジトリのトップに「README.md」という名前のファイルが必要だという決まりがあるからだ。この決まりは、レポジトリに訪れた見ず知らずの人が、そのレポジトリが何なのかを、これを読んで分かるようにするために存在するのだが、今は練習だし非公開だしなので名前だけで中身の無いファイルを作ってアップすることにする。

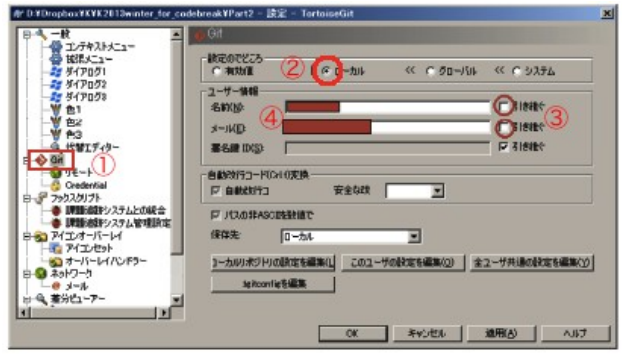
自分の PC のリポジトリを作って、ここから README.md をプッシュすることにする。自分の名前のフォルダの中に Part2 という名前のフォルダを作って、中で右クリックしてみよう。「Git ここにリポジトリを作成」というのがあるので、それをクリックし、次に出てくる画面ではチェックを入れずに OK しよう。

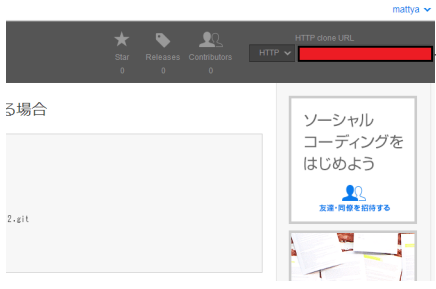


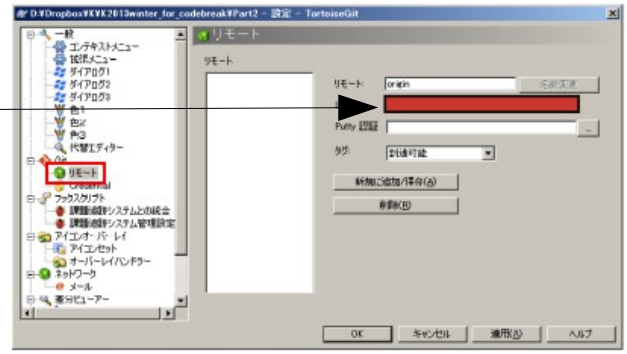


次に、この手元のリポジトリを web リポジトリにリンクさせる。右クリックメニュー→TortoiseGit→設定と進んで、「Git」のところに codebreak への登録に使ったユーザー名とメールアドレスを、「リモート」のところに、ブラウザでリポジトリを開いた時に右上の方に出てくる欄に書かれている URL を記入し、OK を押す。「フェッチしますか」と出てくるので OK しておく（といってもまだ web リポジトリには何もないのでフェッチしても何も起こらない）。





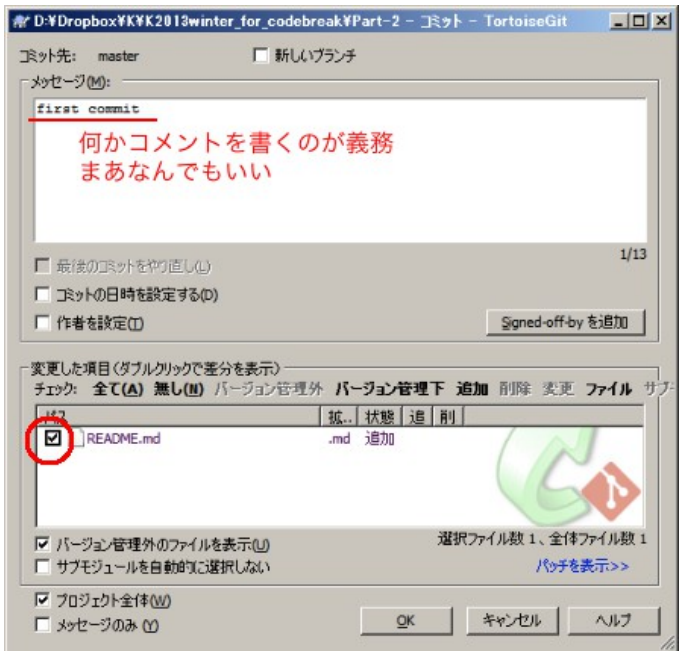




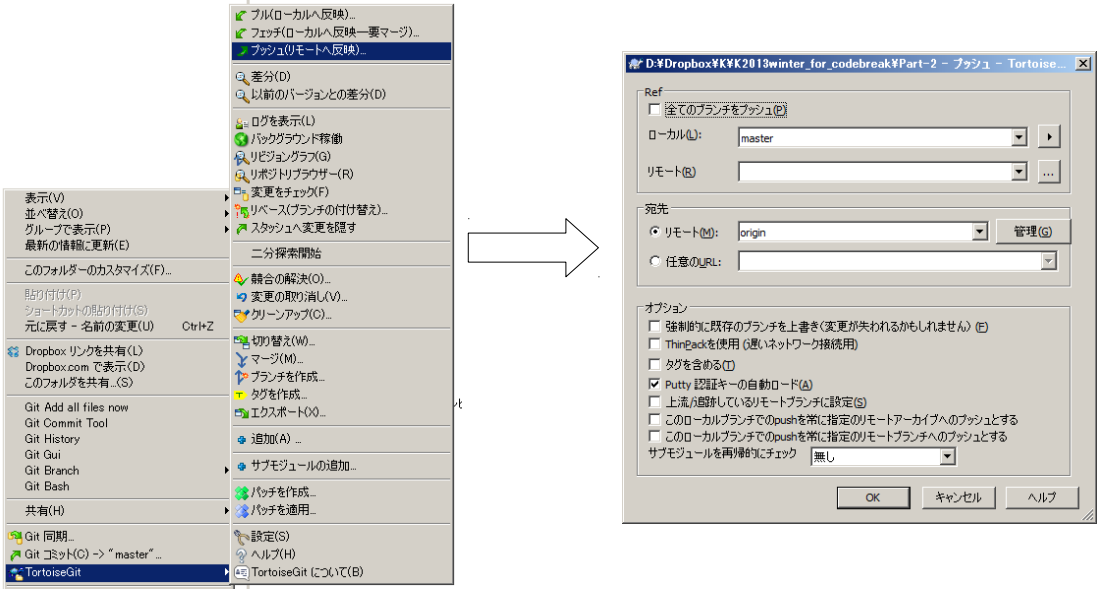
テキストエディタで README.md という名前のファイルを作る。中身はなくていい。作ったら、そのファイルを右クリックして、TortoiseGit→追加を押す。

この「追加」というのは、コミットした時に記録を残す、すなわちリポジトリに記録されるファイルを選ぶということである。追加しなかったファイルはリポジトリには反映されない。一度追加されたファイルは解除しない限りずっとコミットの度に更新履歴が記録され続ける。プログラミングを進めるとゴミファイルがたくさんできるので、追加を使ってほんとうに必要なものだけをリポジトリに入れるようにする。

次に、右クリックメニューから「コミット->master」というのを選ぶ。次のような画面が出るので、下の欄の README.md にチェックを入れて、上の空欄に何か書いて、OK を押す。これで README.md の追加がコミットされる。



最後にプッシュをして、README.md の追加を web リポジトリに反映させる。フォルダの中を右クリックして、右クリックメニューから TortoiseGit→プッシュを選択する。ウインドウが出てくるが、一番上のローカルの欄が master になっていることを確認して、OK を押す。

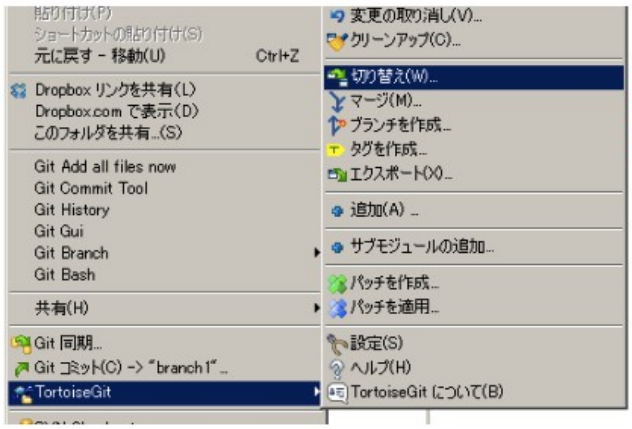


これで web リポジトリに README.md が追加されたはずである。ブラウザからリポジトリに入ると、次のような画面となり、リポジトリが稼働し始めたのが分かる。「ファイル」のタブに行くと、確かに README.md が存在するのが分かるだろう。

README.md の追加により、リポジトリを稼働させることが出来た。
ここまでで、ファイルを作ってから web リポジトリに行くまでに
　追加→コミット→プッシュ
の操作が必要であることがわかった。

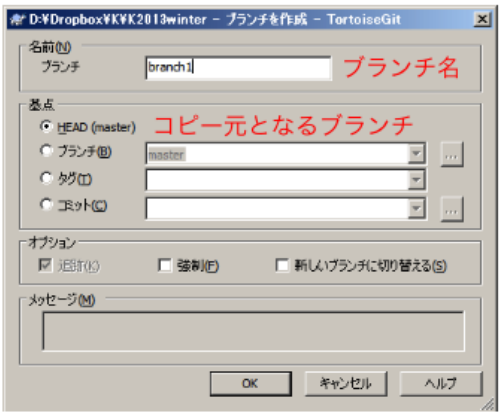
それでは、この手法を使ってもう一つファイルを追加してみよう。
テキストエディタで「hoge.txt」という名前のファイルを作り、中身に
　　hello!
とだけ書こう（まあ名前や中身はなんでもいいのだが）。
このファイルを、追加→コミット→プッシュすることで、web リポジトリに反映させよ。

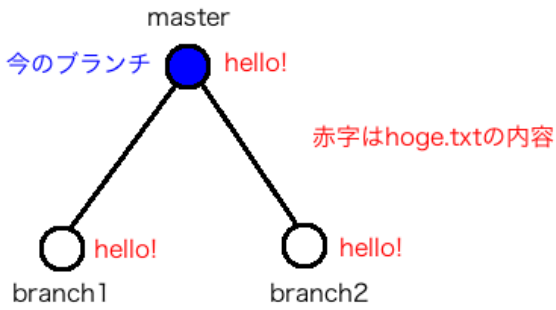
次はブランチとマージの機能を使ってみる。
ブランチに関するコマンドは、いずれも右クリックメニューの TortoiseGit の中にある。



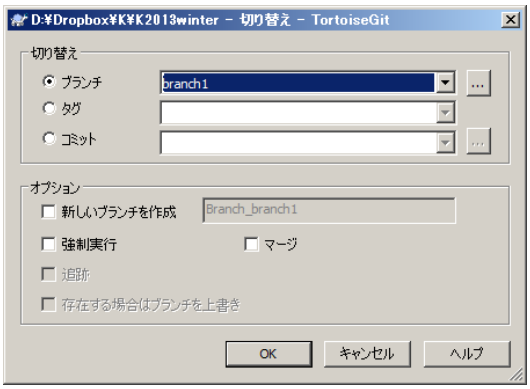
現在、リポジトリには master という名前のブランチのみが存在する。master ブランチというのはリポジトリの最上位に位置するブランチで、通常ここにはバグ等のない、ある程度完成されたプログラムを格納する。

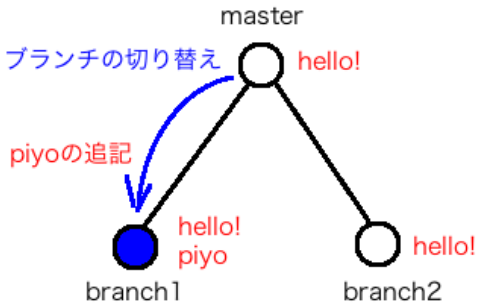
ブランチの作成で、branch1 という名前のブランチと、branch2 という名前のブランチを作ろう。どちらも master が基点となるようにする。





次にブランチの切り替えで branch1 に移動しよう。



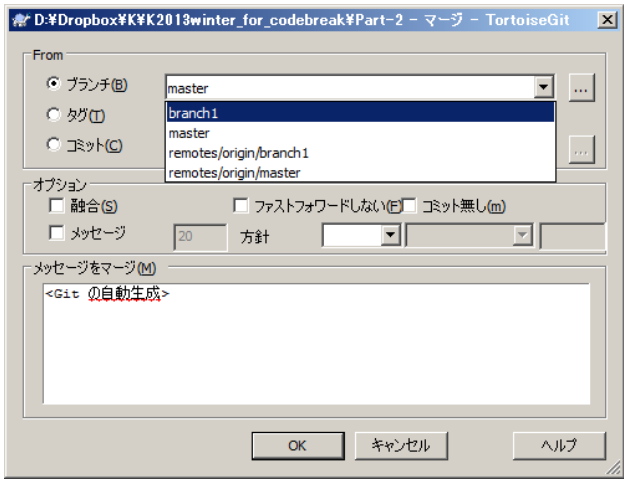


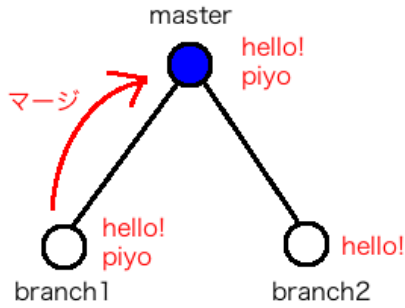
branch1 に移動すると、もうそこで編集したファイルは、マージされるまで master には反映されない。master は編集前の綺麗なまま保たれるのである。branch1 が十分にバグが無くなって綺麗になってから、マージを行うのである。

ここで hoge.txt を編集して、hello!の次の行に「piyo」と書こう。そしてコミットを行うことで branch1 に変更を反映させる。

一旦テキストエディタを閉じて、ブランチの切り替えで master や branch2 に移動してみよう。そこで hoge.txt を開くと、piyo という文字列はなく、branch1 編集前の状態を保っていることが分かる。

次に、この変更を master に反映させることを考える。まずブランチの切り替えで master に移り、右クリックメニューからマージを行う。次のようなウィンドウが出るので、一番上のブランチの選択を branch1 にし、OK しよう。





今 hoge.txt を開くと、master ブランチでもちゃんと piyo が反映されているはずである。
ここで、branch2 に切り替えて、hoge.txt を開こう。piyo はここには反映されていないことが分かるだろう。そこで、hello!の次の行に hoge と書こう。そしてこれをコミットし、branch2 に反映する。

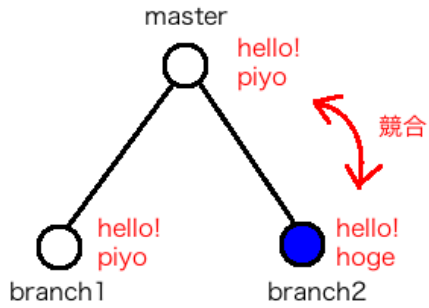
branch2 を master にマージしたいのだが、しかし、考えてみると master の hoge.txt は branch1 がすでに反映されていて

hello!
piyo

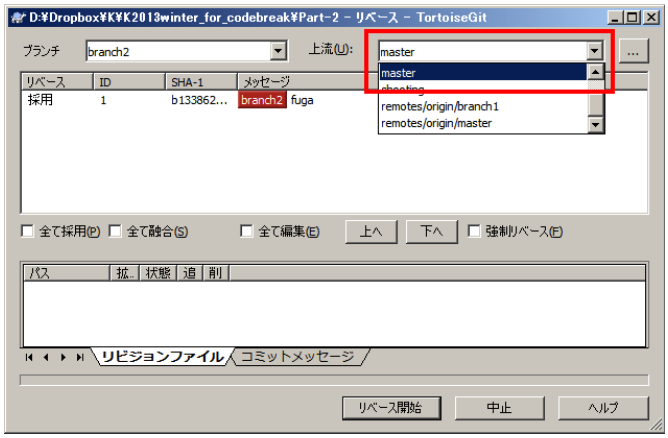
であり、branch2 のほうは

hello!
hoge

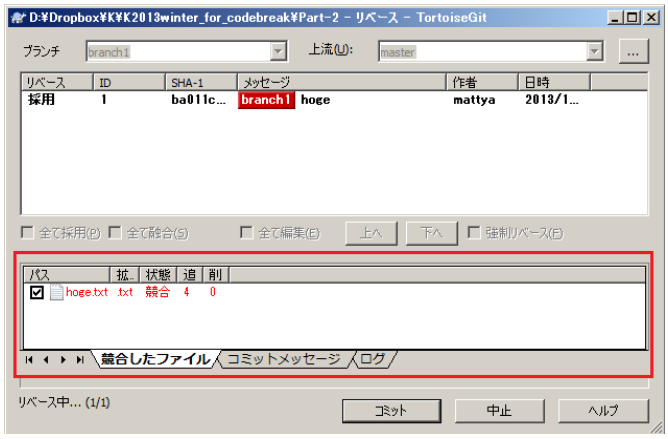
である。この二つをくっつけた時、piyo と hoge とどっちが先に来るべきかは分からない。こういう状態をコンフリクトという。このような状態は「branch で作業していたら、他の人が master を更新して、自分の元にした master と合わなくなってしまった」という状態であり、よく発生する。
これを解決するには、master に生じた更新を branch2 に反映させて、古い master ではなく最新の master を元に branch2 を開発したかのようにしてやればいい。この操作をリベースという。

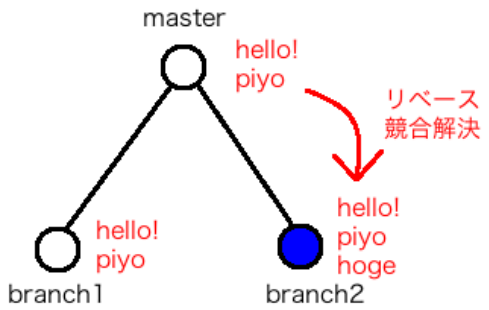


branch2 に切り替え、右クリックメニューからリベースを選ぶ。「上流」の欄を master にして、リベース開始を押す。

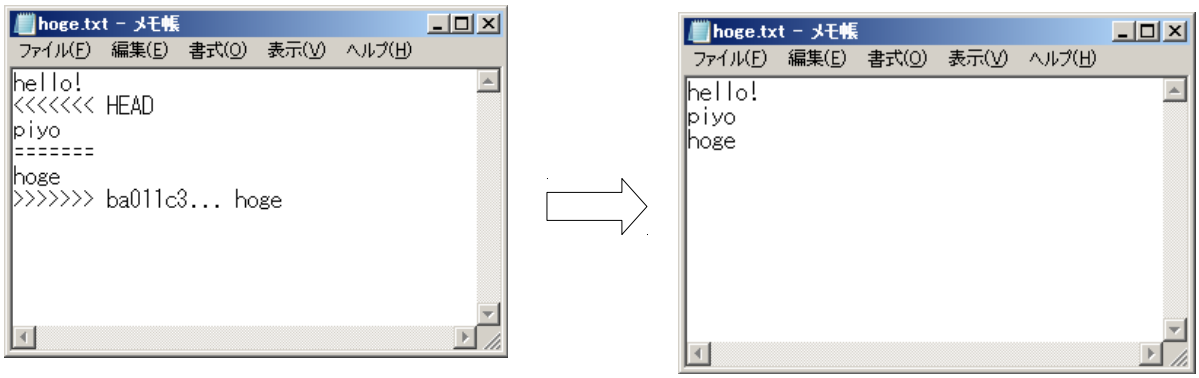


次のような画面が現れて、下の方に「競合したファイル」と表示されるはずである。このとき、このウィンドウを閉じないまま、hoge.txt を開いてみよう。



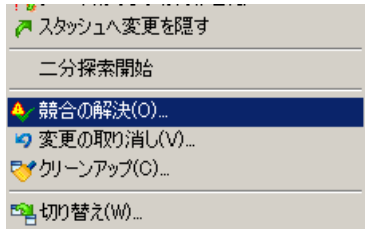


次のように、<<<< HEAD とか====とか変な記号がたくさん書き込まれているはずである。これは、リポジトリのプログラムがどこがコンフリクトしたかを判断して、人間に分かるように印をつけてくれたのである。



コンフリクトの解決は人間の仕事である。今回は piyo が hoge より先に来るということにしよう。右のようにファイルを書き換えて、上書き保存する。

次に、フォルダの中を右クリックして、TortoiseGit→競合の解決を選ぶ。

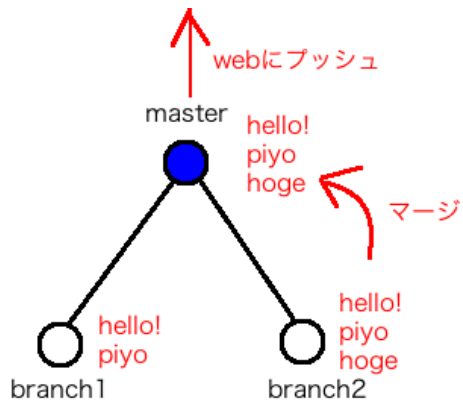


これで競合は解決された。さきほど開きっぱなしだったリベースのウィンドウに戻り、コミットボタンを押す。

master ブランチに移行し、branch2 を master にマージする。コンフリクトなど無くマージに成功すればうまく行っている。最後にプッシュをして web リポジトリに反映させよう。ブラウザからでも、hoge.txt が

hello!
Piyo
hoge

となったことが分かるだろう。



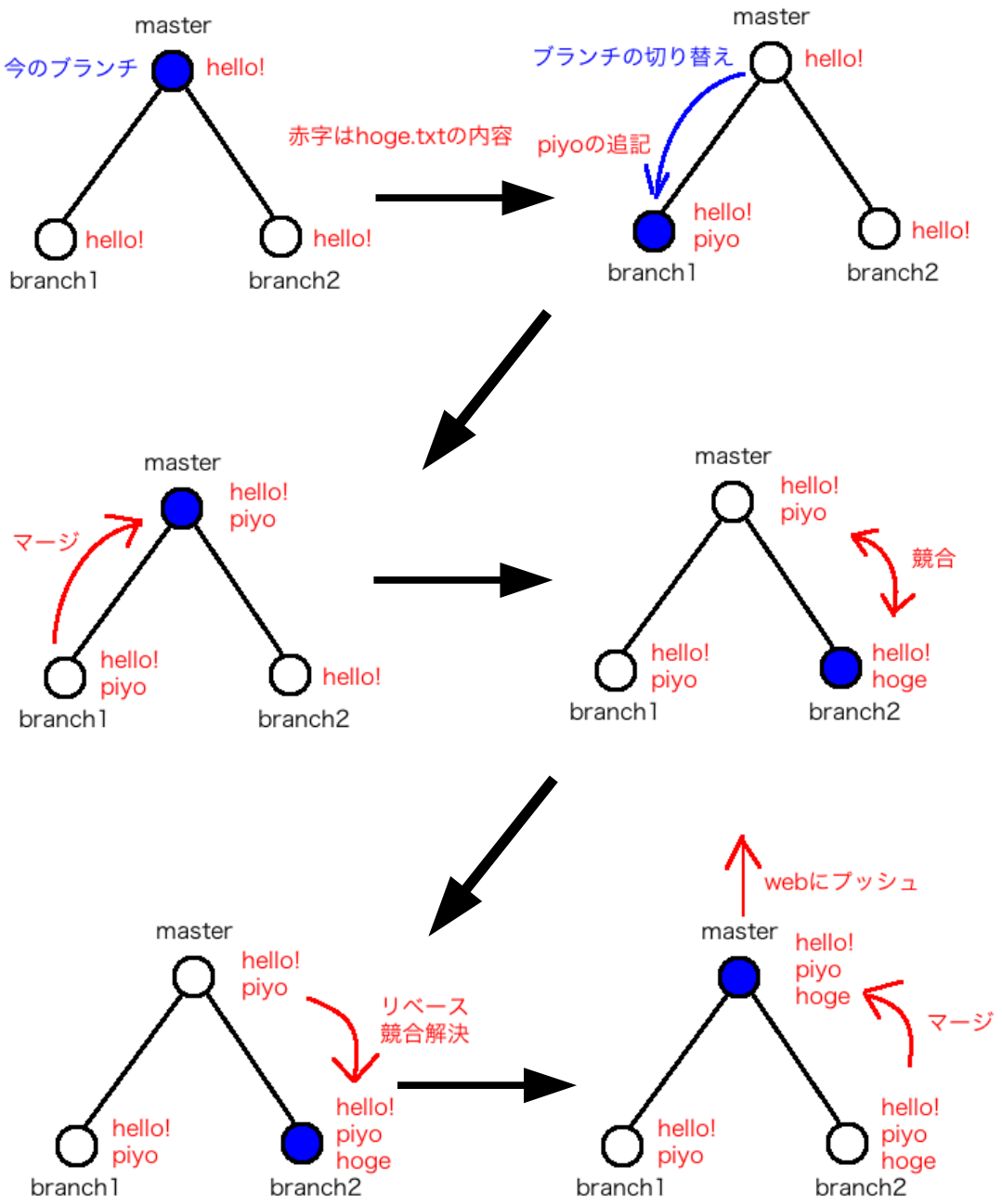
ブラウザから hoge. txt の更新を確認

コミットを見るとちゃんと

```
hello!
piyo
hoge
```

が見える。

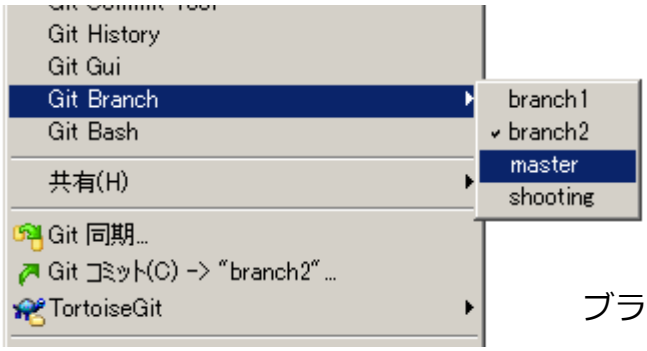
ここで行った操作の図


master
今のブランチ
hello!
赤字はhoge.txtの内容
hello!
hello!
branch1
branch2
master
ブランチの切り替え
hello!
piyoの追記
hello!
piyo
hello!
branch1
branch2
master
hello!
piyo
マージ
hello!
piyo
hello!
branch1
branch2
master
hello!
piyo
競合
hello!
piyo
hello!
hoge
branch1
branch2
master
hello!
piyo
リベース
競合解決
hello!
piyo
hello!
piyo
hoge
branch1
branch2
webにプッシュ
master
hello!
piyo
hoge
マージ
hello!
piyo
hello!
piyo
hoge
branch1
branch2

まとめ

・リポジトリを作る
(1) codebreak のページでリポジトリを作る
(2) 手元の PC でフォルダを作る
(3) フォルダの中で Git リポジトリの作成を行う
(4) フォルダの中に README.md というファイルを作り、追加する
(5) コミット
(6) プッシュ
(7) ブラウザ上で master ブランチができていることを確認

・リポジトリを更新する
(1)　ブランチ develop のデータを改良するとしよう
(2)　develop を元にブランチを作る（branch1 とする）
(3)　ブランチを branch1 に切り替える
(4)　作業
(5)　新しいファイルを作ったなら追加する
(6)　コミットする
(7)　(develop ブランチをフェッチする　これは共同作業するようになってから)
(8)　別の人などにより develop ブランチに変更があったなら、リベースを行う
(9)　このときコンフリクトが起きたら、それを解消する
(10) コンフリクトを解消したら、再びコミットする
(11) ブランチを develop に切り替える
(12) branch1 をマージする
(13) プッシュして web 上に反映させる



ブランチの簡単な切り替え方

2. シューティングゲーム

(1) shooting ブランチを作る
(2) 元となるプログラムを書く
(3) shooting ブランチを元に bullet_basic ブランチと game_main ブランチを作る
(4) bullet_basic ブランチの作業
    1. bullet_basic ブランチに移動
    2. Bullet クラスを埋める
    3. コミット
    4. shooting ブランチに移行してマージ
(5) game_main ブランチの作業
    1. game_main ブランチに移動
    2. メインクラスを埋める
    3. コミット
    4. shooting ブランチをリベース
    5. コンフリクトを解消
    6. コミット
    7. shooting ブランチに移行してマージ
(6) プッシュ

・実装内容

3. 共同作業

(1) 代表者の仕事
1. 新たなリポジトリを作り Shooting と名付ける
2. 他のメンバーを招待する
3. master ブランチから shooting ブランチを作る
4. shooting ブランチに移動
5. 今出来ているシューティングのプログラムを Shooting にコピー
6. コミット
7. プッシュ

(2) 代表でない人の仕事
1. 手元の PC で新たに Shooting フォルダを作っておく
2. 代表者の招待を受け、Shooting リポジトリに行く
3. 右上の欄からアドレスをコピーし、それを使って Shooting フォルダ内にリポジトリの作成を行う
4. 代表者が 7 まで終えたらフェッチを行う
5. shooting ブランチに移動し、プログラムがあることを確認する

以下、「プレイヤーの攻撃」「誘導弾」「キーの同時押し対応」の 3 機能を、2, 3 人で分担して制作していくことにする。

(3) プレイヤーの攻撃を追加する人
1. player_attack ブランチを作る
2. player_attack ブランチに移動
3. メインクラスに弾を撃つ処理を加える
4. コミット
5. shooting ブランチに移動
6. フェッチする。もし更新があったら
    1. player_attack ブランチに戻る
    2. shooting ブランチをリベース
    3. コンフリクトを解消
    4. コミット
    5. shooting ブランチに移動
7. マージ

(4) 誘導弾を作る人
1. bullet_type ブランチを作る
2. bullet_type ブランチに移動
3. Bullet クラスに誘導弾を実装

4. コミット
5. shooting ブランチに移動
6. フェッチする。もし更新があったら
    1. bullet_type ブランチに戻る
    2. shooting ブランチをリベース
    3. コンフリクトを解消
    4. コミット
    5. shooting ブランチに移動
7. マージ

(5) キーの同時押し機能を追加する人

1. key_state ブランチを作る
2. key_state ブランチに移動
3. KeyState クラスを作成し、実装する
4. コミット
5. shooting ブランチに移動
6. フェッチする。もし更新があったら
    1. key_state ブランチに戻る
    2. shooting ブランチをリベース
    3. コンフリクトを解消
    4. コミット
    5. shooting ブランチに移動
7. マージ

(6) プッシュ

・実装内容

4. シューティングゲーム改良案

(1) 自機に弾が当たったらダメージを受けるようにする
(2) 自機が撃った弾が他の弾に当たったら、その弾にダメージ
(3) タイトル画面、ゲームオーバー画面を作る
(4) スコアを計算する
(5) 自機、弾の画像の表示
(6) 爆発のエフェクト
(7) 弾を撃つ弾（要するに敵）の実装
(8) アップグレード機能の実装
(9) 弾の情報を XML ファイルから読み込むようにする
(10)　ステージを XML ファイルで表すようにする